Linköping Studies in Science and Technology

Dissertations. No. 1581

Performance-aware Component Composition for GPU-based Systems

by

Usman Dastgeer

Linköping University

Department of Computer and Information Science
Linköping University
SE-581 83 Linköping, Sweden

Linköping 2014

Copyright © Usman Dastgeer 2014

ISBN 978-91-7519-383-0
ISSN 0345–7524
Printed by LiU Tryck 2014

URL: http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-104314

Abstract

This thesis adresses issues associated with efficiently programming modern heterogeneous *GPU-based systems*, containing multicore CPUs and one or more programmable Graphics Processing Units (GPUs). We use ideas from component-based programming to address programming, performance and portability issues of these heterogeneous systems. Specifically, we present three approaches that all use the idea of having multiple implementations for each computation; performance is achieved/retained either a) by selecting a suitable implementation for each computation on a given platform or b) by dividing the computation work across different implementations running on CPU and GPU devices in parallel.

In the first approach, we work on a skeleton programming library (SkePU) that provides high-level abstraction while making intelligent implementation selection decisions underneath either before or during the actual program execution. In the second approach, we develop a composition tool that parses extra information (metadata) from XML files, makes certain decisions offline, and, in the end, generates code for making the final decisions at runtime. The third approach is a framework that uses source-code annotations and program analysis to generate code for the runtime library to make the selection decision at runtime. With a generic performance modeling API alongside program analysis capabilities, it supports online tuning as well as complex program transformations.

These approaches differ in terms of genericity, intrusiveness, capabilities and knowledge about the program source-code; however, they all demonstrate usefulness of component programming techniques for programming GPU-based systems. With experimental evaluation, we demonstrate how all three approaches, although different in their own way, provide good performance on different GPU-based systems for a variety of applications.

This work has been supported by two EU FP7 projects (PEPPHER, EXCESS) and by SeRC.

Populärvetenskaplig sammanfattning

Att få varje generation av datorer att fungera snabbare är viktigt för samhällets utveckling och tillväxt. Traditionellt hade de flesta datorer bara en general-purpose processor (den så kallade CPU:n) som bara kunde exekvera en beräkningsuppgift i taget. Under det senaste årtiondet har dock flerkärniga och mångkärniga processorer blivit vanliga, och datorer har också blivit mer heterogena. Ett modernt datorsystem innehåller vanligtvis fler än en CPU, tillsammans med specialprocessorer såsom grafikprocessorer (GPU:er) som är anpassade för att kunna exekvera vissa typer av beräkningar effektivare än CPU:er. Vi kallar ett sådant system med en eller flera GPU:er för ett GPU-baserat system. GPU:er i sådana system har sitt eget separata minne, och för att kunna köra en beräkning på en GPU så behöver man vanligtvis flytta all indata till GPU:ns minne och sedan hämta tillbaka resultatet när beräkningen är klar.

Programmeringen av GPU-baserade system är icke-trivialt av flera anledningar: (1) CPU:er och GPU:er kräver olika programmeringsexpertis och verktyg, exempelvis kräver koden för en beräkning på en GPU en helt annan struktur än samma beräkning på en CPU. (2) Att snabba upp en beräkning kräver oftast speciella insatser för optimeringen av redan fungerande kod.

I denna avhandling använder vi idéer från komponentbaserad programutveckling för att lösa programmerings- och kodoptimeringsproblem för dessa system. Vi presenterar tre olika ansatser som alla använder idén att kunna välja mellan flera implementationer (varianter) för en och samma beräkning. I alla ansatserna uppnår man bra prestanda genom att antingen: (a) välja den mest lämpliga implementationen bland de valbara varianterna för varje beräkningsuppgift i programmet för ett givet GPU-baserat system, eller (b) dela upp beräkningsarbetet mellan olika implementationer som exekverar på CPU:er och GPU:er parallellt. Den ibland nödvändiga kommunikationen mellan de olika minnesmodulerna optimeras genom att flytta data enbart när det verkligen behövs, vilket avgörs först vid exekveringstid när man har den nödvändiga informationen till hands. De tre ansatserna är:

1. Ett skelettprogrammeringsbibliotek (SkePU) som tillhandahåller färdiga komponenter som programmeraren kan använda för att uttrycka beräkningar som matchar deras beräkningsstruktur.

2. Ett programkompositionsverktyg som tar in extra information om beräkningarna och deras olika implementationer i form av XML-filer, avgör vissa optimeringsbeslut redan före exekveringen, och slutligen genererar kod för att avgöra resterande optimeringar under programmets exekvering.

3. Ett ramverk som använder speciella kommentarer i källkoden och programanalys med hjälp av en kompilator för att analysera beräkningens struktur och generera lämplig kod för att optimera implementeringsurvalet vid exekveringstid.

Dessa ansatser har olika egenskaper, exempelvis är den sistnämda ansatsen mer generisk än den första, medan den första kräver mindre programmeringsarbete. Ändå demonstrerar alla tre att den komponentbaserade programmeringstekniken gynnar en effektiv programmering av GPU-baserade system.

Vi redovisar en omfattande experimentell utvärdering av varje ansats med flera applikationsprogram på olika GPU-baserade system. De experimentella utvärderingen visar att varje ansats, trots uppenbara skillnader, leder till bra prestanda på olika GPU-baserade system för ett spektrum av applikationer.

Acknowledgements

Finally, the moment to write this final part of my PhD thesis has come. It took me more than four years and I have many people to acknowledge for both academic and personal support.

First and foremost, I would like to thank my main supervisor Christoph Kessler for his support and guidance througout this thesis work. He was a perfect advisor to me; accessible, open to new ideas and always available for discussions and advices. I learned a lot during this time with our interactions, and the freedom he gave me as my supervisor groomed my independent thinking.

After that, I thank my co-supervisor Kristian Sandahl for his help and guidance in all matters. I learned many things about software engineering from him which helped me evaluate techniques that I devised during this thesis work. Thanks also to Johan Enmyren, who, together with Christoph Kessler, started the work on the SkePU skeleton framework which part of this thesis is based upon.

Thanks also to all the past and present members of the PELAB and my colleagues at the department of computer and information science, for creating an enjoyable atmosphere. Thank you Lu, Nicolas, Mudassar, Martin, Adeel, Mattias, Mahder, Mikhail, David, Peter, Kristian Ståvaker and other past and present members of IDA. Special thanks to Erik for all those interesting (and often insightful) lunch conversations. A big thanks to Per Östlund for solving technical issues with the Fermi machine that I used mostly for experiments. Thank you Anne Moe, Åsa Kärrman, Liselotte Lundberg, Eva Pelayo Danils, Bodil Mattsson Kihlström, and Inger Norén for taking care of all the administration issues for me.

This work has been financially supported by EU FP7 projects PEPPHER (www.peppher.eu) and EXCESS (www.excess-project.eu and by Swedish e-Science Research Center (SeRC). Especially, I would like to thank all the members of the PEPPHER project, for interesting discussions in the project meetings that I have attended. I have learned a lot from these discussions and many ideas to this research are influenced by our discussions in the project meetings.

For experimental evaluation, we used one machine provided by our

project partner (in the PEPPHER project) at University of Vienna. We would like to thank them for that. Thanks also to Daniel Cederman and Philippas Tsigas for running some experiments on their CUDA machines.

Coming to my personal life, I am afraid I have too many people to list by name but I can try to mention a few. When coming to Sweden for my master studies, I was accompanied by bachelor friends (Hassam, Muteer and Tehman) in our Björnkärrsgatan 5C-12 apartment. Since then, I met lot of new people; some of them became my friends including Khurram, Shahid, Qamar, Qaiser, Inam, Imran, Raheel, Salman, Basit, Mohsin, Kamran, Rizwan and Pasha bahi. I have lot of cheerful memories with these people that will stay with me for the rest of my life. A big thanks to Mansoor bahi and Salma bhabi for inviting me over, every few weeks, for delicious dinner in a family atmosphere.

Last but definitely not the least, I am grateful to my parents and family for their continuous support and encouragement. Talking to them on every weekend gave me a reflection moment, and fueled my energy for the coming working days.

During my PhD, I have met lot of wonderful people and had some amazing experiences, both here in Linköping and during my travel to conferences, meetings and visits to other places. It is difficult to recall and list all those people in few pages. So, to all those who have helped in this thesis work (directly or indirectly) and are not mentioned above: thank you very much!

Contents

1 Introduction **1**
1.1 Motivation 1
1.2 Component-based approach 2
1.3 Optimized composition problem 4
1.4 Our work 7
1.5 Contributions 8
1.6 List of publications 10
1.7 Research Method 13
1.8 Thesis outline 14

2 Background **16**
2.1 Component-based software engineering 16
2.2 Skeleton programming 19
2.3 Runtime scheduling and selection using a runtime system 21
2.4 Multicore and OpenMP 24
2.5 Programming NVIDIA GPUs 26

3 SkePU **33**
3.1 SkePU library 34
3.1.1 Generation of User functions 35
3.1.2 Smart containers 35
3.1.3 Skeletons 38
3.2 Implementation selection 43
3.2.1 Algorithm 44
3.2.2 Implementation details 46
3.2.3 Related work 51

3.3 Runtime support for hybrid execution 51
3.4 Evaluation 54
3.4.1 Tuning implementation selection 54
3.4.2 Hybrid execution 57
3.4.3 SkePU Overhead Analysis 57
3.5 Case study on MapOverlap2D on CUDA 60
3.5.1 Memory optimizations 62
3.5.2 Adaptive tiling optimization 63
3.5.3 Experiments 69
3.6 Summary 77

4 Containers and memory management 78
4.1 Introduction 79
4.2 Memory management in SkePU 80
4.2.1 Initial mechanism 83
4.2.2 Limitations of the initial mechanism 86
4.2.3 New mechanism 90
4.3 Evaluation 102
4.3.1 Effect of communication optimizations 102
4.3.2 New versus initial mechanism 105
4.4 Containers with StarPU runtime system 110
4.4.1 Memory management in StarPU 110
4.4.2 Container interaction with StarPU 111
4.5 SkePU program execution model 112
4.6 Summary 113

5 PEPPHER Composition tool 115
5.1 PEPPHER Component model 115
5.2 Composition tool 118
5.3 Prototype implementation 120
5.3.1 Static composition 121
5.3.2 Component expansion 122
5.3.3 Code generation 122
5.3.4 Usage of smart containers 124
5.3.5 Inter-component parallelism 125
5.3.6 Intra-component parallelism 127

5.3.7 Support for performance-aware component selection . . . 128
5.3.8 Support for PDL (Platform Description Language) . . . 128
5.3.9 Conditional composition . . . 129
5.3.10 Documentation and User manual . . . 132
5.3.11 Utility mode . . . 133
5.4 Composition example . . . 133
5.5 Evaluation . . . 136
5.5.1 Productivity evaluation . . . 136
5.5.2 Hybrid execution . . . 140
5.5.3 Dynamic Scheduling and selection . . . 142
5.5.4 Smart containers and inter-component parallelism 142
5.5.5 Conditional composition . . . 146
5.5.6 PEPPHER Runtime Overhead . . . 147
5.6 Summary . . . 148

6 Global Composition Framework 149
6.1 GCF Component Model . . . 150
6.1.1 Components, interfaces and implementation variants . . . 150
6.1.2 Composition Technique . . . 153
6.2 Global Composition Framework . . . 155
6.2.1 Componentization and Component Repository 156
6.2.2 Component tree builder . . . 157
6.2.3 Composer . . . 159
6.2.4 Performance Modelling API . . . 162
6.3 Evaluation . . . 168
6.3.1 Implementation selection . . . 169
6.3.2 Online learning . . . 171
6.3.3 Hybrid execution . . . 172
6.4 Global composition . . . 173
6.4.1 Greedy scheduling and selection using HEFT . 174
6.4.2 Global Heuristics . . . 178
6.5 Summary . . . 186

7 Discussion and Conclusion **188**
7.1 Three approaches 188
7.1.1 Similarities 189
7.1.2 Differences 190
7.2 Concluding remarks 191

8 Related work **193**
8.1 Skeleton programming 193
8.1.1 Skeleton frameworks for multicore CPU systems and MPI-clusters 193
8.1.2 Skeleton programming for GPU-based systems 195
8.1.3 Other approaches for GPU-based systems . . . 199
8.2 Programming approaches for GPU-based systems . . . 199
8.2.1 Unified programming abstractions 200
8.2.2 Component-based approaches 203
8.2.3 Task-based programming models 204
8.2.4 Code generation 205
8.3 Component Models in HPC and Grid Computing . . . 207
8.3.1 Component Models in Grid Computing 207
8.3.2 Component Models in HPC 208

9 Possible extensions **209**
9.1 SkePU extensions 209
9.2 PEPPHER Composition tool 211
9.3 Global Composition Framework (GCF) 212

"The way the processor industry is going, is to add more and more cores, but nobody knows how to program those things. I mean, two, yeah; four, not really; eight, forget it."

Steve Jobs

Chapter 1

Introduction

1.1 Motivation

For several decades, computer architects kept increasing the speed of a single microprocessor by increasing its clock frequency. However, in 2003, this trend came to a halt due to power and heat dissipation problems, known as *power wall* [17, 3, 50, 16, 174, 93]. Today, the growth in computer architecture is driven by a need for power-efficient computing which has lead to both heterogeneity and parallelism in microprocessors. In the last 10 years, CPU architectures have gone from serial to parallel and a new generation of more power-efficient specialized co-processors, general-purpose programmable Graphic Processing Units (GPUs), have emerged.

On the software side, this disruptive shift in computer architecture lead to a crisis situation [95, 170]. The hardware thread-level parallelism of unprecedented scale ended the traditional paradigm of sequential programming. The sequential software would not run faster with every generation of microprocessors as the clock frequency of microprocessors is either stagnant or decreasing. Software would have to be written exposing parallelism to get performance on modern architectures. Porting legacy software as well as writing new correct software to run efficiently on these systems is still considered a black art [114]. Absence of a single standard parallel programming model

along with diversity and heterogeneity in parallel architectures inflate the problem even further.

Heterogeneous *GPU-based systems* containing multi-core CPU processors optimized for low latency and one or more programmable GPU co-processors, optimized for high throughput, are already mainstream. These systems are becoming popular for their high performance potential at relatively lower (energy) cost, in both traditional and high performance computing domains. However, GPU-based systems expose programmability, performance and portability problems for the application programmer as described below.

The programmer must have programming skills in different programming models to program CPU and GPU devices present in a GPU-based system. Moreover, most of these programming models are at low level, expose architectural features and/or lack proper tooling support for debugging and analysis. Furthermore, getting good performance often requires non-intuitive device-specific optimizations requiring knowledge about cache hierarchy, interconnect and organization of compute units in the system. These low-level optimizations not only make programming a heroic task but also pose questionmarks on the portability of the source code to new architectures with e.g., different GPU and compute unit organization. Even if the code works on the new architecture (i.e., code portability) the performance could deteriorate because of architecture-specific optimizations embedded in the source code.

There exist obvious tradeoffs between programmability, performance and portability aspects. Optimizing for performance often requires coding device-specific optimizations which are very low-level and can restrict the portability to other systems, possibly with different architectural features. A high level of abstraction may yield better programmability support at the expense of performance.

1.2 Component-based approach

In this thesis, we use concepts from component-based programming to tackle the programming, performance and portability problem for

GPU-based systems. The idea is to structure/write applications in the form of components where a component models a specific functionality with a given semantics. A component consists of an interface that represents the functionality and of multiple implementations of that interface that actually implement that functionality in some programming model/language. All of a component's implementations are considered functionally equivalent and can be used interchangeably in a component call context; however, they may have different performance characteristics and can be written in different programming models targeting different compute devices.

The component implementations can come from different sources. The implementations can be provided by expert programmers who have sound programming skills as well as proper domain knowledge to write good implementations. In some cases, implementations can be automatically generated by some tool. For example, there exist source-to-source compilers/code-generators that can automatically generate parallel code for GPU execution from a sequential CPU implementation [28, 213]. An important source for component implementations are libraries that are released by third party vendors. Examples include libraries that provide important dense and sparse matrix kernels such as the BLAS and NVIDIA CUSP (CUda SParse) libraries. In many cases, an implementation can simply wrap calls to these libraries provided by third parties. The last option is to write one from scratch if none of the above sources are available. However, this may not happen that often if a component repository is in place. A component repository can store information about components along with a description of their functionality and different implementations available. With time, the chance of finding an appropriate existing component (or implementation) in the repository increases as more people adopt component-based programming and start putting their components to the repository. In this way, components along with their implementations can be managed and made available as ready-made COTS (Commercial Off-The-Shelf) components [199].

In an application program, the user of components needs to call the component by its interface, without bothering about which actual

implementation is invoked. Selection and invocation of an appropriate implementation for each call can be done automatically by a framework/tool considering information about the target system as well as the actual computation at hand. Performance can be optimized by choosing the (expected) best implementation for each component call in the program such that it either minimizes or maximizes a given objective function (e.g., execution time, energy). When porting to a new platform, performance can be retained for the application program by re-doing these decisions.

1.3 Optimized composition problem

For each component call in a program, the framework needs to select an appropriate implementation considering a given objective function. This is normally referred to as the *implementation selection* problem. On GPU-based systems, the component implementations are normally written either for CPU or GPU execution. The presence of multiple compute devices in the system that can be used simultaneously as well as disjoint memory address spaces complicate this problem further as one needs to consider resource allocation and data management when doing the implementation selection on these systems to reduce overall execution time (or other objective function). This is called the *optimized component composition problem.*

The "component call" expression can refer to the static view of the component call in a program source code (called static component call) or to an execution view of the component call for a certain program invocation (dynamic component call). A static component call can map to zero or more dynamic component calls for a certain program invocation. In this thesis, when not specified explicitly (e.g. component call in a program's source code), a component call mean dynamic view of the component call.

In the following, we describe the optimized component composition (implementation selection) problem in a more formal manner. For simplicity, we consider an application with calls to one component type, but the formulation can easily be extended for multiple

types of components. Let C denote the interface of a component having N component implementation variants[1] $V_C = \{V_1, ..., V_N\}$. The optimization objective in our case is to reduce the overall (expected) execution time T for all component calls in the program[2]. First, we suppose that an application has just one component call $call(C)$ to the component C with execution context CTX. The execution context of a call consists of multiple factors including operand data (size, locality, contents), system workload and underlying architectural features. Suppose that the (expected) execution time of each implementation variant V_i (where $i = 1, ..., N$) for a given call context CTX is given by $t_{V_i}(CTX)$. In this case, the (expected) optimal implementation would be an implementation V_{opt}, where $1 \leq opt \leq N$, that has the shortest (expected) execution time in context CTX, i.e.,

$$t_{V_{opt}}(CTX) \leq t_{V_i}(CTX) \; \forall i \text{ where } 1 \leq i \leq N \tag{1.1}$$

Now, we consider multiple component calls $call_j(C)$, $j = 1, ..., M$, to the component C with data and control dependency. Although calling the same computation, each of these calls has a possibly different execution context CTX_j, $j = 1, ..., M$. For simplicity, we assume that all these component calls are data- and/or control-dependent and should be executed in sequence. The optimal decision would be to choose an implementation variant for each component call $V_{sel,j}$, where $j = 1, ..., M$, in the program in a manner that it reduces the overall (expected) application execution time, i.e.,

$$minimize \; t_{V_{sel,1}}(CTX_1) + ... + t_{V_{sel,M}}(CTX_M) \tag{1.2}$$

In some cases, the chosen $V_{sel,j}$ for a component call $call_j$ would be different from the optimal selection for that call if considered standalone (see Equation 1.1), i.e., $V_{opt,j}$. This is because a greedy decision

[1] In this thesis, we use the terms *implementation*, *implementation variant* and *variant* interchangeably.

[2] We consider reduction of execution time of all component calls inside the program. However, it should directly relate to reduction of overall application execution time considering that all computationally significant parts of an application can be modeled as components.

of selecting the best performing variant for each component call could result in an overall sub-optimal decision, as we will see later in Chapter 6.

In practice, things are even more complicated. Modeling the execution context for a component call in a cost-effective manner is a tricky problem. The resource sharing (cache pollution etc.) between different component calls could affect their execution behavior in a non-intuitive manner. Moreover, component calls in a program normally have arbitrary dependency patterns among each other.

The composition decisions are normally made at runtime, considering that the execution context for a component call includes information about operand data (contents, sizes and locality) which is, in many cases, not known beforehand. This puts a strict limit on the overhead of decision making as this overhead directly affects the application performance and is normally incurred for each component call in the program. What an acceptable level of overhead is, is dependent on computation granularity and potential performance gains of making composition decisions with the implemented mechanism in comparison to random guesses or fixed selections.

The optimized component composition problem can be considered a generalized task scheduling (and selection) problem where each component call is a task with multiple implementations, targeting different kinds of processors (CPU, GPUs) present in the system. A runtime system can then schedule a task by deciding which of its implementations to use on which processor in the system. This decision could be made by considering the execution cost of different task implementations on different kinds of processors. The offline task scheduling problem on a heterogeneous platform (with ≥ 2 kinds of processors) is quite complex in its generality [103]. Even simpler variations of this problem are NP-complete [82]. For example, [123, 98] shows that scheduling m independent tasks on a heterogeneous system (with ≥ 2 kinds of processors), where each task can take different time on different kinds of processors, is NP-complete. Similarly, optimal scheduling of tasks with precedence constraints is known to be NP-complete in general (i.e., except for very restricted architectures and task parameters) [49].

1.4 Our work

In one sentence, the focus of our work can be summarized as follows:

> *We investigate (and build) general, efficient and practical mechanisms for programming GPU-based systems by employing ideas from component-based programming.*

By general, we mean that we are interested in devising techniques that can work for a variety of applications from different application domains. Basically, we can consider any computation that can be modeled as components. Moreover, the techniques devised should be efficient in selecting the (expected) best implementation with low runtime overhead. Finally, the techniques should be practical such that they can be used in real-world situations on existing GPU-based systems. This means that we focus on efficient heuristics that work reasonably well in practice rather than focusing on finding an optimal solution (e.g., by building an integer linear programming (ILP) formulation and solving it offline). Also, the programmer should not be bothered with making such decisions. Instead, the logic to make these decisions automatically can be abstracted away behind a simple component call interface.

Considering the above guidelines, we designed and implemented three different approaches in this thesis work. All these approaches are based on the idea of having multiple (possibly optimized) implementations for each computation where performance is achieved and retained by making appropriate composition decisions on the target platform. These approaches are:

- A C++ skeleton (i.e., generic component) library that provides high-level abstraction to the programmer while having multiple implementations for each skeleton with tunable selection capabilities. It provides performance, in a portable manner, by optimizing communication and by automatically selecting the (expected) best implementation for each skeleton computation in a program. However, it works only for structured computations that can be modeled with the available skeleton set.

- An XML based component annotation approach that relies on extra-information (metadata) provided in XML to generate the composition code for a runtime system to make the final decision. It is non-intrusive to the source code but requires XML annotations specified in separate documents.

- A pragma-based approach with source-code analysis capabilities that can do composition decisions considering both local and global optimization criteria.

Each approach is fully implemented and experimentally evaluated, with different applications, on modern GPU-based systems. In the thesis text, we first discuss these approaches separately in the listing order, and in the end compare and contrast their capabilities.

1.5 Contributions

Following are the major contributions of this work:

- A skeleton programming framework for modern GPU-based systems that supports tunable implementation selection as well as support simultaneous use of multiple kinds of computing resources for a single skeleton call (known as *hybrid execution*).

- An adaptive implementation selection mechanism for our skeleton library based on offline tuning, achieving around 95% selection accuracy with less than 1% training space exploration.

- Design and implementation of *smart containers* that provide a generic high-level interface (similar to C++ STL containers) while tracking operand data copies in different memory units at runtime. When used with our composition frameworks, they enable several runtime optimizations such as the elimination of unnecessary data transfers and asynchronous component execution capability.

- A new memory management mechanism, implemented in the SkePU vector and matrix containers, that provides a stronger

consistency model while reducing the communication cost, especially on systems with two or more GPUs.

- The PEPPHER component model that uses XML-based annotations for specifying performance-relevant information. Furthermore, a *component composition tool* is developed that parses XML-based annotations and platform descriptions and generates composition code, supporting static, dynamic and conditional composition in a non-intrusive manner.

- The concept of *conditional composition* for GPU-based systems that provides a generic method for the programmer to influence the selectability of component implementation variants based on run-time properties of the call context (e.g., the size or sparsity degree of a matrix operand) or static constraints such as the availability of certain software libraries on the target system.

- A pragma-based component model and global composition framework (GCF) that can do performance-aware component composition, program transformations and implementation selection decisions.

- A generic performance modeling API that enables usage of both analytical and empirical performance models while supporting online feedback about runtime execution information to the composition framework.

- Empirical analysis of the HEFT (Heterogeneous Earliest Finish Time) greedy scheduling and selection technique as well as a discussion of scenarios where it can prove sub-optimal. We propose a bulk strategy for one such scenario involving component calls constrained in a data dependency chain.

During this thesis, the author has worked on three major software prototypes:

- The SkePU library has been extended by the author with new features, tuning framework, data types and skeletons (see Chapters 3 and 4). During this process, several new public releases of

the library have been made. The library is currently used in different contexts such as in the EU FP7 ParaPhrase project [47].

- The author rewrote the basic prototype for the PEPPHER composition tool that was initially designed in a master thesis project [144] under his supervision. The prototype has been extended heavily since then to add support for many features (see Chapter 5), and was a major software deliverable in the EU FP7 PEPPHER project (D1.4 [73]).

- The author developed GCF, a source-to-source compilation based composition framework based on the ROSE compiler [182] alongside a light-weight runtime library (see Chapter 6).

1.6 List of publications

The material in this dissertation is based in part on the following publications:

1. Christoph W. Kessler, Sergei Gorlatch, Johan Enmyren, Usman Dastgeer, Michel Steuwer and Philipp Kegel. Skeleton Programming for Portable Many-Core Computing. In: *S. Pllana and F. Xhafa, eds., Programming Multi-Core and Many-Core Computing Systems*, Wiley-Blackwell, New York, USA, accepted 2011 (to appear 2014).

2. Usman Dastgeer, Christoph W. Kessler and Samuel Thibault. Flexible runtime support for efficient skeleton programming on hybrid systems. *ParCo'11: International Conference on Parallel Computing*. Ghent, Belgium, 2011.

3. Christoph Kessler, Usman Dastgeer, Samuel Thibault, Raymond Namyst, Andrew Richards, Uwe Dolinsky, Siegfried Benkner, Jesper Larsson Träff and Sabri Pllana. Programmability and Performance Portability Aspects of Heterogeneous Multi-/Many-core Systems. *DATE-2012: Conference on Design Automation and Testing in Europe*, Dresden, Germany, 2012.

4. Usman Dastgeer and Christoph Kessler. A performance-portable generic component for 2D convolution computations on GPU-based systems. *MULTIPROG-2012 Workshop at HiPEAC-2012*, Paris, France, 2012.

5. Usman Dastgeer, Lu Li, Christoph Kessler. The PEPPHER Composition Tool: Performance-Aware Dynamic Composition of Applications for GPU-based Systems. *MuCoCoS'12: Int. Workshop on Multi-Core Computing Systems*, Supercomputing Conference (SC12), Salt Lake City, Utah, USA, 2012.

6. Usman Dastgeer and Christoph Kessler. A step towards performance-aware global component composition for GPU-based systems. *CPC'13: 17th Int. Workshop on Compilers for Parallel Computers*, Lyon, France, 2013.

7. Usman Dastgeer, Lu Li and Christoph Kessler. Adaptive implementation selection in a skeleton programming library. *APPT-2013: Biennial Conference on Advanced Parallel Processing Technology*, Stockholm, Sweden, 2013.

8. Usman Dastgeer and Christoph Kessler. A Framework for Performance-aware Composition of Applications for GPU- based Systems. *P2S2'13: Intl. Workshop on Parallel Programming Models and Systems Software for High-End Computing*, ICPP-2013, Lyon, France, 2013.

9. Usman Dastgeer, Lu Li and Christoph Kessler. The PEPPHER Composition Tool: Performance-Aware Composition for GPU-based Systems. *Computing journal*, Springer, 2013. doi:10.1007/-s00607-013-0371-8

10. Usman Dastgeer, Johan Enmyren, Mudassar Majeed, Lu Li and Christoph Kessler. SkePU: Efficient and productive skeleton programming for GPU-based systems. *Submitted to Computing journal*, Springer, 2013.

11. Usman Dastgeer and Christoph Kessler. Conditional component composition for GPU-based systems. *MULTIPROG-2014*

workshop at HiPEAC-2014, Vienna, Austria, 2014.

12. Usman Dastgeer and Christoph Kessler. Performance-aware Composition Framework for GPU-based Systems. *Supercomputing journal*, Springer, 2014. doi:10.1007/s11227-014-1105-1

The following papers were published during the dissertation period as well but are not included in the thesis:

- Johan Enmyren, Usman Dastgeer and Christoph W. Kessler. Towards A Tunable Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. *MCC'10: Proceedings of the 3rd Swedish Workshop on Multicore Computing.* Gothenburg, Sweden, 2010.

- Usman Dastgeer, Johan Enmyren and Christoph W. Kessler. Auto-tuning SkePU: A Multi-Backend Skeleton Programming Framework for Multi-GPU Systems. *IWMSE'11: Proceedings of the 4th international workshop on Multicore software engineering.* ACM, New York, USA, 2011.

- Akhtar Ali, Usman Dastgeer and Christoph Kessler. OpenCL on shared memory multicore CPUs. *MULTIPROG-2012 Workshop at HiPEAC-2012*, Paris, France, 2012.

- Lu Li, Usman Dastgeer and Christoph Kessler. Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. *iWAPT-2012: Intl. Workshop on Automatic Performance Tuning*, VECPAR-2012 Conference, Kobe, Japan, 2012.

- Mudassar Majeed, Usman Dastgeer and Christoph Kessler. Structured Development of Scalable Scientific Applications for GPU Clusters. *MCC'12: Fifth Swedish Workshop on Multicore Computing*, Stockholm, Sweden, 2012.

- Lu Li, Usman Dastgeer and Christoph Kessler. Pruning strategies in adaptive off-line tuning for optimized composition of components on heterogeneous systems. *MCC'12: Fifth Swedish Workshop on Multicore Computing*, Stockholm, Sweden, 2012.

- Usman Dastgeer, Lu Li and Christoph Kessler: Performance-Aware Dynamic Composition of Applications or Heterogeneous Multicore Systems with the PEPPHER Composition Tool. *CPC-'12: 16th Int. Workshop on Compilers for Parallel Computers*, Padova, Italy, 2012.

- Mudassar Majeed, Usman Dastgeer and Christoph Kessler. Cluster-SkePU: A Multi-Backend Skeleton Programming Library for GPU Clusters. *PDPTA-2013: Int. Conf. on Parallel and Distr. Processing Techniques and Applications*, Las Vegas, USA, 2013.

Besides the publications listed above, the following PEPPHER project deliverables are co-authored during the dissertation period:

- Usman Dastgeer, Christoph Kessler, Siegfried Benkner, Erich Marth, Martin Sandrieser, Johannes Singler and Jesper Larsson Träff. *D1.1: PEPPHER component model*, Technical report, Dec 29, 2010.

- Siegfried Benkner, Sabri Pllana, Erich Marth, Martin Sandrieser, Christoph Kessler and Usman Dastgeer. *D1.2: PEPPHER Coordination Language and Component Composition Techniques*, Technical report, June 30, 2011.

- Siegfried Benkner, Sabri Pllana, Jesper Larsson Träff, Richard Homonnai, Erich Marth, Martin Sandrieser, Samuel Thibault, Christoph Kessler, Usman Dastgeer and Brendan Barry. *D1.3: Performance Modeling and Portability*, Technical report, June 30, 2011.

- Usman Dastgeer, Lu Li, Christoph Kessler, Siegfried Benkner, Martin Sandrieser, Enes Bajrovic and Sabri Pllana. *D1.4: Research prototype implementation*, Technical report, Dec 31, 2012.

1.7 Research Method

The discipline of computer science [79] has roots in three other disciplines: mathematics, natural science and engineering. The ACM Task

Force on the core of computer science [62] proposes three paradigms that can be used to characterize (research) work in computer science. These paradigms are theory, abstraction and design.

Theory has roots in mathematics where the objects of study are identified, and their relationships are hypothesized, proved and later interpreted. Abstraction is rooted in experimental scientific methods where a hypothesis is formed, models are built to predict and in the end, experiments are carried out to evaluate built models. Finally, the results are analyzed. Design has roots in engineering where requirements are specified, and a solution is designed, built and tested to ensure that it satisfies given requirements.

As pointed out in [62], all three paradigms are equally important and are often intertwined together in computer science. Moreover, all three paradigms model a process which can be repeated and iterated over multiple times until the desired outcome is achieved.

The research presented in this thesis mainly follows the abstraction and design paradigms. Following the abstraction paradigm, we built performance models that predict execution time or best configuration for a program execution. The predictions made by these models are later experimentally evaluated. The work on automatic selection of the best implementation in SkePU also follows the process explained in the abstraction paradigm. On the other hand, the development of three prototypes maps to the design paradigm where the requirements on the proposed solutions are specified and later the solutions are designed, developed and tested to ensure that they meet the specified requirements. Similarly, we first investigate problems in the previous SkePU memory management mechanism and outline specifications for an improved mechanism. Later, we design and implement a new memory management mechanism and evaluate its effectiveness by comparing it to the old mechanism.

1.8 Thesis outline

The remainder of this thesis is structured as follows:

- Chapter 2 provides background information that is important

Table 1.1: Mapping of thesis chapters to publications.

Thesis Chapter	Publications No.
Chapter 3	1-4, 7 and 10
Chapter 5	5, 9 and 11
Chapter 6	6, 8 and 12

for understanding contents presented in later chapters.

- Chapters 3 and 4 discuss the SkePU skeleton programming framework and memory management mechanisms implemented inside the SkePU containers respectively.
- In Chapter 5 we describe the PEPPHER component model and the composition tool prototype. We discuss its different features including support for conditional composition.
- Chapter 6 presents the global composition framework that uses program analysis and source-code transformations for optimized component composition.
- In Chapter 7 we first discuss and compare the different approaches presented, and later conclude the thesis work.
- Chapter 8 discusses the related work in detail.
- Chapter 9 lists some topics for future work.

Text in Chapters 3, 5 and 6 is largely based on the publications that are listed in Section 1.6. Table 1.1 describes the mapping between publications (numbering as of Section 1.6) and these three thesis chapters.

Chapter 2

Background

"... when we start talking about parallelism and ease of use of truly parallel computers, we're talking about a problem that's as hard as any that computer science has faced... I would be panicked if I were in industry."

John Hennessy

This chapter introduces several important concepts that may help the reader in understanding contents presented in other chapters. The chapter starts with an introduction of component based software engineering and its potential benefits. Next, it gives a short introduction to skeleton programming which is important for especially understanding contents in Chapter 3. Afterwards, it introduces runtime component composition and the StarPU runtime system including HEFT scheduling and selection. A brief overview of multicore shared memory programming using OpenMP and GPU computing using CUDA and OpenCL is also presented.

2.1 Component-based software engineering

From the very start, software has been prone to complexity and failures. Charette [52] lists some famous software failures (called "Software Hall of Shame") costing up to few billion US dollars incurred either during software development that was aborted in-between or caused by some erroneous software in operation. To minimize such failures, Doug Mcilroy, at the 1968 NATO conference, proposed a

component-based approach for building software, much like in the fields of electronics or mechanics. He argued that a large library of tested software components would make programming easier, less costly and less prone of failures. This started work on component-based approaches for software engineering [14, 27].

In the last four decades, the idea of component-based software engineering has been actively researched by the software engineering community. There exist several definitions of what a component is. The most famous one is given by Szyperski [199] as follows:

> "A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to third-party composition."

Reuse is an important part of component-based software engineering. A component must be designed in a way that it can be used in any context where its explicitly specified dependencies are met. Normally, a component hides its implementation details behind a well defined functional interface that captures its possible functional interactions with its environment (i.e. the outside world, including other components). This also allows multiple implementations of a component to co-exist behind a single functional interface. Component reuse is supported by the notion of component repositories/marketplaces where components can be published, searched and traded by programmers. Reusing existing components to build software is advantageous for many reasons: It can significantly reduce both cost and time that is spent in developing a new software. A software built as a composite of subparts rather than a monolithic entity is more modular, maintainable and testable. Moreover, considering that published components can be scrutinized and well-tested, the software built using those components can be more reliable. Standardization can also be enforced in a more effective manner for systems built from existing components in a modular fashion. Above all, component-based software development can make software development more predictable and reduce the risk of failures.

Components can be defined at different granularity level. A component can consist of one procedure/method up to multiple procedures organized in a module in a procedural programming environment. In object oriented programming, a component can correspond to one class or to a set of classes. However, components are not objects (i.e., instances of classes) as the former do not have (externally observable) state whereas the latter do have state.

Multiple components are composed together to build a software entity/application. Normally, the interaction between a component and its environment happens via classical caller-callee binding where a method in a component's functional interface is invoked by its environment. A component is usually classified as either black-box, gray-box or white-box when it comes to interaction. A black-box component hides all of its implementation details and can be interacted with via its functional interface only. In contrast, the implementation of white-box components is fully exposed and can be modified by the user of the component. A gray-box component lies in the middle as it exposes selected details of its internal functionality besides its functional interface as variability knobs. These knobs can be used e.g. for certain adaptions and optimizations to affect the internal working of the component in a controlled manner, and hence provide a mechanism to, for instance, affect its performance.

Developing a software using components requires a component model which describes the notion of components and how they can be composed together. As with a component, there exists no universally accepted definition of a component model. According to Lau [140], a component model describes syntax, semantics and composition of components. Semantics of a component include its provided and required services, and whether the components are black-box or gray-box. Information about syntax includes the programming language for implementation as well as syntax guidelines for specifying functional interface, optimization knobs and pre-conditions on the operating environment of the component. Binding mechanisms and other information about usage of components in a software application are also specified in the component model, including the interaction mechanism between different components.

In this thesis work, we propose two component models that are based on the idea of having multiple implementations for each component by separating a component's interface from its implementations. Moreover, the components are gray-box in our case, i.e., they expose certain information about their implementation details which is used for making platform-specific optimization decisions for execution of those components.

2.2 Skeleton programming

Parallel programs are often structured in one or more *computation* and *coordination* parts [106]. The *computation* parts express the calculation describing application logic, control and data flow in a procedural manner. The *coordination* parts consist of managing concurrency issues such as thread management, synchronization, load balancing, memory management and communication between the concurrent threads.

Algorithmic skeletons were introduced in 1989 by Cole [56] as a mechanism to model frequently occuring computation and coordination patterns inside a program. An algorithmic skeleton (or simply called skeleton) is a predefined programming construct, e.g., a generic component, that models a specific computation and/or coordination pattern while providing a sequential high-level interface to the user of the skeleton. Internally, the skeleton implementations may be written using different parallel programming models and other capabilities offered by the underlying system/device (intrinsics, SIMD, etc.).

Using a skeleton in a program involves parameterizing the skeleton with the actual computation or business logic (often called user function). Writing applications with skeletons is advantageous as parallelism and synchronization, as well as leveraging specific target architectural features, come almost for free for skeleton-expressed computations. However, computations that do not fit any predefined skeleton or their combination still have to be written manually.

As an example, a *map* skeleton denotes the element-wise independent application of an operation (also known as *user function*) on each

element of an input array which produces corresponding elements in an output array. For instance, to calculate the element-wise square for an input array, the map skeleton can be used with a square function as operation (aka user function). This is an example of a data-parallel skeleton. Skeletons can be broadly categorized into data- and task-parallel skeletons:

- *Data-parallel skeletons*: The parallelism in these skeletons comes from independent operations on aggregate (possibly large amount of) data elements, e.g., by applying a certain function f independently on each element in a large data structure. The behavior of data-parallel skeletons establishes functional correspondences between data and is usually considered as a type of fine-grained parallelism [106].

- *Task-parallel skeletons*: The parallelism in task-parallel skeletons comes from exploiting independence between different tasks. The behavior of task-parallel skeletons is mainly determined by the interaction between tasks. The granularity of tasks, either fine or coarse, determines the granularity of parallelism.

By arbitrary nesting of both task and data-parallel skeletons, structured hierarchical parallelism can be built inside a skeleton application. This is often referred to as *mixed-mode parallelism* [106].

Skeleton frameworks can provide more than one implementation for each skeleton they provide. For example, for each skeleton, a skeleton framework could provide one implementation for execution on CPU and one for GPU execution. In this way, a skeleton call can be considered as a component call with multiple implementations available from which one suitable implementation should be chosen for a given call context along with appropriate resource allocation (i.e., the component composition problem). The only difference is that skeletons are predefined, generic components with implementations for each skeleton provided as part of the skeleton framework.

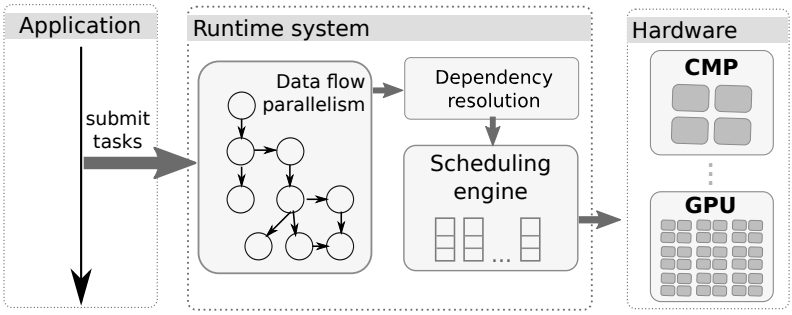


Figure 2.1: Overview of an application execution using a task-based runtime system for a GPU-based system.

2.3 Runtime scheduling and selection using a runtime system

Component composition when done at runtime becomes a task scheduling (and selection) problem if each component call in the program becomes a task operating on some operand data (gets some input and produces some output). Normally, access modes (read and/or written) for each task operand are specified to the runtime system which allows it to automatically detect data dependencies between submitted tasks. Figure 2.1 describes how an application containing multiple component calls is executed using a runtime system such as StarPU. The runtime system serves as a middleware between the application and the hardware as it manages execution of application tasks on the underlying hardware. The application exposes tasks to the runtime system which detects data dependencies between submitted tasks based on information about the access modes of each task's operand data. When a task becomes *ready* (i.e., all its task dependencies are resolved), it is considered for scheduling. Please remember that scheduling a task in our context is combined with selection of an appropriate task implementation besides allocation/selection of compute/storage resources. For practical reasons of keeping runtime overhead low, runtime systems such as StarPU consider one ready task for scheduling at a time. Each runtime system provides different greedy scheduling and selection policies (HEFT, work-stealing etc.)

and a chosen policy is used to schedule the tasks on the underlying hardware.

Heterogeneous Earliest Finish Time (HEFT) was proposed by Topcuoglu et al. [203] originally as a static scheduling heuristic for heterogeneous systems. It has become the most popular dynamic scheduling and selection heuristic in use today for GPU-based systems. Just like other greedy heuristics, it considers one component call at a time when making the implementation selection decision. However, it makes the decision in an intelligent manner by considering previous historical execution records for different implementations, current system workload as well as operand data transfer cost to different memory units considering their current residence.

In our work, we have mostly used the StarPU runtime system and HEFT scheduling and selection for doing runtime component composition on GPU-based systems. In the following, we give a brief introduction to the StarPU runtime system.

StarPU

StarPU [21, 20] is a C-based unified runtime system for heterogeneous multicore platforms with generic scheduling and selection facilities. Three main components of StarPU are task and codelet abstraction, data management, and dynamic scheduling and selection framework.

StarPU task-model StarPU uses the concept of *codelet*, a C structure containing different implementations of the same functionality for different computation units (e.g., CPU and GPU). A StarPU task is then an instance of a codelet applied to some operand data. When using StarPU, the programmer has to explicitly submit all tasks and register all the input and output operands for all tasks. The submission of tasks is asynchronous and termination is signaled through a callback[1]. This lets the application submit several tasks, including tasks which depend on others. Dependencies between different tasks

[1]Task execution can be made synchronous as well, by setting the `synchronous` flag for a StarPU task. This makes the task submission call blocking and returns control after the submitted task finishes its execution.

can either be derived by StarPU implicitly, by considering data dependencies (read-after-write, write-after-read, write-after-write) between submitted tasks, and/or can be explicitly specified by the programmer using integers called *tags*.

Data management StarPU provides a virtual shared memory subsystem for operand data and keeps track of data across different memory units in the machine by implementing a weak consistency model using the MSI (Modified, Shared, Invalid) coherence protocol [84]. This allows StarPU to avoid unnecessary data movement when possible. Moreover, the runtime system can estimate data transfer cost and can do prefetching to optimize the data transfers. The runtime system can also use this information to make better scheduling and selection decisions (e.g., scheduling tasks considering where their operand data resides).

StarPU defines the concept of *filter* to partition data logically into smaller chunks (block- or tile-wise) to suit the application needs. For example, the filter for 1D vector data is the *block filter* which divides the vector into equal-size chunks, while for a dense 2D matrix, the applicable filters include partitioning the matrix into horizontal and/or vertical blocks. Multiple filters can be applied in a recursive manner to partition data in any nested order (e.g., dividing a matrix into 3×3 blocks by applying both horizontal and vertical filters).

Dynamic scheduling and selection Mapping of tasks to different execution units is provided in StarPU by using dynamic scheduling which include implementation selection for each task, as mentioned above. There are several built-in scheduling and selection policies, including greedy (priority, no-priority), work-stealing, and several performance-model aware policies (e.g., based on historical execution records).

2.4 Multicore and OpenMP

In 1965, Gordon Moore from Intel observed that the number of transistors on a chip would double approximately every 18 months [160]. Now known as Moore's law, this observation has guided computer designers for more than four decades. During this period, the exponential increase in transistor count was consistently translated to exponential performance increases. This was possible due to Dennard scaling [77] which showed that energy efficiency improves proportionally with transistors getting smaller; so if transistor size reduced to half, the power it used fell by a factor of four (with voltage and current both halving). But around 2003, chip makers found that they could no longer improve the energy efficiency as sharply as they had done in the past although transistors keep getting smaller and more numerous on a chip [92, 91]. This ended the Dennard scaling era and raised the issue of unsustainable increase in power consumption as transistors get faster — now known as the power wall. Instead, processor chip designers started using the increasing transistor count to put multiple microprocessor cores on a chip. Such chips are called multicore microprocessors (or multicore CPUs).

Many experts believe that the increase in core count on a chip will be exponential and will thus reach to potentially thousands of cores in a few years. We already have general-purpose processor chips available in the consumer market with core count from 2 to a few dozens. Also, some chips have two kinds of cores: few "fat" cores, optimized for low latency, that can run sequential programs at high speed, and lots of "thin" simple cores, optimized for throughput computing, that can run more parallel workloads. For power savings, techniques such as DVFS (Dynamic Voltage Frequency Scaling) and shutting down unused cores are becoming popular.

On the software side, to continue getting increased performance on these multicore systems, an application needs to exploit parallelism so that work can be divided across multiple cores on the chip. Applications need to be written using some parallel programming model that provides constructs to expose different kinds of parallelism in the application [39, 5]. The idea of parallel programming is not new in

computing as it has been present since the early days of programming [78]. However, it was limited to few application domains and few experts (mostly in high-performance computing). In mainstream computing, few applications were using threads, mainly to improve responsiveness by, e.g., doing computations while waiting for a user input. This time, the need for parallel programming is both more prevalent and more critical [174]. It is prevalent as almost every application on every new architecture need to consider this; also it is more critical as parallelism has become the main tool to get decent performance for an application on modern systems. Considering that cores on a mainstream/standard multicore chip often share the same memory (thus being shared memory systems), shared-memory parallel programming models such as OpenMP are becoming widely used.

OpenMP

In the 1980s, many different parallel architectures with their own programming extensions have emerged which created problems for programmers on writing and porting software across different machines. Open Multi Processing (OpenMP) was introduced by the OpenMP Architecture Review Board (ARB) in 1997 as a standard language/-platform to program different shared memory systems built by different vendors [51]. Initially it started as a set of directives mainly targeting loop-level (data) parallelism for Fortran. Within a few years, it extended its features set to model task-level parallelism and support C and C++. Many C, C++ and Fortran compilers now implement support for OpenMP as it has become a de facto standard for shared memory programming.

OpenMP provides preprocessor directives/pragmas, library routines and environment variables [191, 66]. It provides a very easy to use high level application programming interface for multi-threading computationally intensive data and task parallel [22] programs. Most OpenMP implementations use thread pools on top of the fork/join model, which exist throughout execution of the program and therefore avoid the overhead of thread creation and destruction after each parallel region. Its work sharing constructs such as `omp parallel for` in

C/C++ (and `omp parallel do` in Fortran) provide loop-level parallelism while `omp sections` and the `omp task` constructs in OpenMP 3.0 support task parallelism.

Portions of the code that are intended to be multi-threaded need to be revised and modified to remove any dependencies before annotating those portions with OpenMP parallelization constructs. The directive based approach, which supports incremental parallelization, makes this model easy to use for parallelizing new as well as existing applications.

2.5 Programming NVIDIA GPUs

GPUs were traditionally designed to handle rendering of high-definition graphics, which is a problem with tremendous inherent parallelism. Rendering graphics essentially requires applying an operation to each pixel in the image independently, which allows to process possibly thousands of pixels at the same time. The GPU's design is influenced by this use-case of massive parallelism. The demand for high-performance graphics has lead to rapid evolution in GPU architecture resulting in 1000-fold growth in the number of transistors in 12 years (from 3 million to 3 billion between 1997 and 2009) [164]. Earlier GPUs had a highly specialized hardware pipeline which suited graphics and similar applications and hence made it difficult to use them for general-purpose computing. However, with the introduction of programmable shaders and high-level programming models such as CUDA, more and more applications are being implemented for GPUs [135, 164].

One of the big differences between a traditional CPU and a GPU is the difference between how they use the chip area, as shown in Figure 2.2. A CPU, as a multi-tasking general-purpose processing device, uses much of its chip area for other circuitry than arithmetic computations, such as caching, speculation and flow control. This helps it in performing a variety of different tasks at a high speed and also in reducing (average) latency of sequential computations. The GPU, on the other hand, devotes much more space on the chip

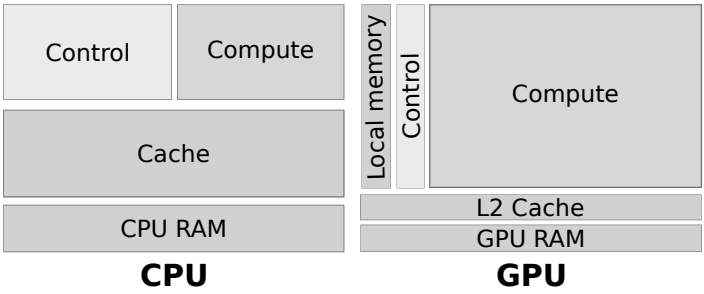


Figure 2.2: Difference between CPU and GPU architectures with respect to area allocated for different type of circuitry.

for pure floating-point calculations since it focuses on achieving high throughput by doing massively parallel computations. This makes the GPU very powerful on certain kinds of problems, especially those that have a data-parallel nature, preferably with much more computation than memory transfers [168]. In GPU computing, performance comes from using a large number of GPU threads, possibly one thread for computing a single value. GPU threads are quite light-weight entities with zero context-switching overhead. This is quite different to CPU threads which are more coarse-grained entities and are usually quite few in numbers.

CUDA

CUDA (Compute Unified Device Architecutre) was released by NVIDIA in 2006 to simplify general-purpose programming of their GPUs. It was initially based on ANSI C with some keyword extensions but gradually many features from C++ are also made available in CUDA. These features include templates, classes, inheritance, operator overloading etc. Since its first public release, CUDA has evolved to a more complete programming language that looks like C++. The GPU computing ecosystem built around CUDA, including a good community-base, early adoption by different industries and strong tooling support, made it the most popular programming language for GPUs. In comparison with OpenCL, CUDA to this day is considered more pop-

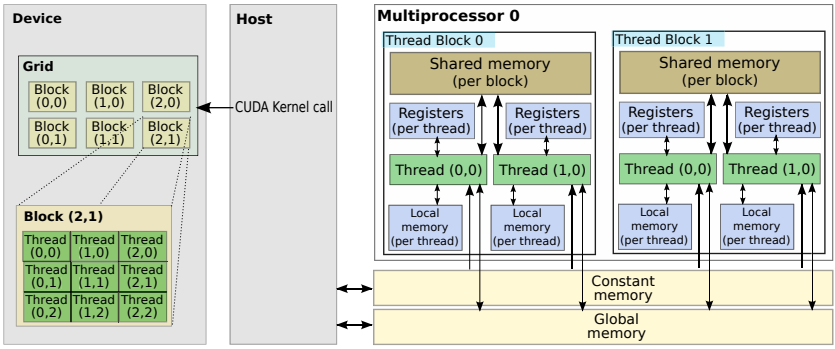


Figure 2.3: A graphical representation of the CUDA programming model, highlighting thread (left) and memory (right) organization. Block is short for Thread-Block.

ular and high-level; however, it is limited to NVIDIA GPUs whereas OpenCL supports CPU, GPU and FPGA devices from different vendors.

In CUDA, the program consists of *host* and *device* code, potentially mixed in a single file that can be compiled by the NVIDIA compiler `nvcc`, which internally uses a conventional C/C++ compiler like *GCC* for compiling the *host* code. The device code is translated by nvcc to an assembly language called PTX (Parallel Thread Execution) which is translated at install time to the target hardware instruction set by a PTX-to-GPU translator/driver. A CUDA program execution starts on a CPU (host thread); afterwards the host thread transfers the input data to GPU device memory before invoking the GPU kernel code. When the kernel execution finishes, output data is transferred back from the GPU device memory to the CPU main memory.

Threads in a CUDA kernel are organized in a two-level hierarchy, as shown in Figure 2.3(left). At the top level, a kernel consists of a 1D/2D grid of *thread-blocks* where each thread block internally contains multiple *threads* organized in either 1, 2 or 3 dimensions [135]. The maximum number of threads inside a single thread block ranges from 512 to 1024 depending on the compute capability of a GPU.

One or more thread blocks can be executed by a single compute unit called *SM* (Streaming Multiprocessor). The SMs do all the thread management and are able to switch threads with no scheduling overhead. Furthermore, threads inside a thread block can synchronize, via shared memory, as they execute inside the same SM. The multiprocessor executes threads in groups of 32, called *warps*, but each thread executes with its own instruction address, register state and local memory, which allows for separate branching. It is, however, most efficient if all threads in one warp take the same execution path, otherwise the execution in the warp is sequentialized [168]. To measure effective utilization of computational resources of a SM, NVIDIA defined the *warp occupancy* metric. The *warp occupancy* is the ratio of active warps per SM to the maximum number of active warps supported for a SM on a GPU.

A CUDA program can use different types of memory, shown in Figure 2.3(right). The *global device memory* is a large but high-latency memory that is normally used for copying input and output data to and from the main memory. Multiple accesses to this global memory from different threads in a thread block can be coalesced into a single larger memory access. However, the requirements for coalescing differ between different GPU architectures [168]. Besides global memory, each SM has an on-chip read/write *shared memory* whose size ranges from 16KB to 64KB between different generations of GPUs. It can be allocated at thread block level and can be accessed by multiple threads in a thread block in parallel unless there is a bank conflict [168]. In the Fermi architecture [212], a part of the shared memory is used as L1 cache (configurable, either 16KB/48KB or 48KB/16KB L1/shared-memory). *Constant memory* is a small read-only hardware-managed cache, supporting low latency, high speed access when all threads in a thread block access the same memory location. Moreover, each SM has 8,192 to 32,768 32-bit general purpose registers, depending on the GPU architecture [168]. The register and shared memory usage by a CUDA kernel can be analyzed by compiling CUDA code using the `nvcc` compiler with the `--ptxas-options -v` option.

OpenCL

OpenCL (Open Computing Language) is an open, royalty-free and low-level standard by the Khronos group [161] that offers a unified computing platform for modern heterogeneous systems. Vendors such as NVIDIA, AMD, Apple and Intel are members of the Khronos group and have released OpenCL implementations, mainly targeting their own compute architectures.

The OpenCL implementation by NVIDIA runs on all NVIDIA GPUs that support the CUDA architecture. Conceptually, the OpenCL programming style is very similar to CUDA when programming on NVIDIA GPUs as most differences only exist in naming of different concepts, as shown in Table 2.1. Using OpenCL, developers write compute kernels using a C-like programming language. However, unlike CUDA, the OpenCL code is compiled dynamically by calling the OpenCL API functions. The dynamic compilation enables better utilization of the underlying device' latest software and hardware features such as SIMD computing capability of the hardware. At the first invocation, the OpenCL code is automatically uploaded to the OpenCL device memory.

In OpenCL terminology, a program runs on an OpenCL device (CPU, GPU etc.) that holds compute units (one or more cores) which further may include one or more single instruction multiple data (SIMD) processing elements. Besides hiding threads, OpenCL goes a step forward in abstracting hardware architectures and provides a common parallel programming interface. It creates a programming environment comprising a host CPU and connected OpenCL devices which may or may not share memory with the host and might have different machine instruction sets.

The OpenCL memory model consists of four types of memories [128]: global, constant, local and private. Every single element of the problem domain is called work-item while some work-items can be grouped together to form a workgroup. Global memory allows read/write access to all work-items in all workgroups but has high access latency so its use must be kept minimal. Constant memory is a part of global memory which retains its constant values throughout the kernel execu-

CUDA	**OpenCL**
Scalar Core	Stream Core
Streaming Multiprocessor	Compute Unit
Warp	Wavefront
PTX	Intermediate Language
Registers	Private memory
Shared Memory	Local memory
Local Memory	Global memory
Grid	NDRange
Thread Block	Work Group
Thread	Work Item
Block Index	Block ID
Thread Index	Thread ID

Table 2.1: Terminology difference between CUDA and OpenCL.

tion. Local memory can be used to make variables shared for a work-group as all work-items of the workgroup can read/write to it. Private memory is only visible to individual work-items and each work-item can modify or read only its own data. GPUs have an on-chip Local Data Share (LDS) and a separate private memory bank with each compute unit, which are the OpenCL local and private memory respectively. CPUs on the other hand implement private memory as register/L1 cache, local memory as L2/L3 cache, global as main memory, and constant OpenCL memory as main memory/cache but their exact implementations are architecture dependent. The execution starts on the host program which manages one or more OpenCL devices by enqueuing the kernel-execution and memory-transfer commands to the devices' command queues.

In principle, code written in OpenCL should be portable (executable) on all OpenCL platforms (e.g., x86 CPUs, DSPs, AMD and NVIDIA GPUs). However, in reality, certain modifications in the program code may be required when switching between different OpenCL implementations [83]. Furthermore, device-specific optimizations applied to an OpenCL code may negatively impact performance when

porting the code to a different kind of OpenCL device [119, 83].

"Redesigning your application to run multithreaded on a multicore machine is a little like learning to swim by jumping into the deep end."

Herb Sutter

Chapter 3

SkePU

In this chapter, we discuss our work on a skeleton programming library named SkePU. The first version of the SkePU library was designed and developed by Enmyren and Kessler [89], with support for data-parallel skeletons working on one-dimensional data only. Since then, we have extended SkePU's design and its implementation in many ways including:

- Support for a two-dimensional data-type and polymorphic skeleton operations.
- New implementation of SkePU data management (discussed in the next chapter).
- An offline tuning mechanism that enables automatic implementation selection with limited training cost.
- Support for the StarPU runtime system, which enables dynamic tuning of implementation selection and simultaneous execution on CPU and GPU devices for a skeleton call (known as hybrid execution).
- Implementation of a new task-parallel (farm) skeleton.
- Porting of several new applications/kernels to the SkePU framework for a comprehensive experimental evaluation.

- Skeleton-specific parametric auto-tuning mechanisms for the MapOverlap2D skeleton.

Here, we present a unified view of the SkePU library based on its current development status. In Section 3.1, we describe the SkePU library while in Section 3.2, we describe the adaptive implementation selection mechanism implemented in the library. Section 3.3 describes the integration work with the StarPU runtime system [21] which allows hybrid execution and asynchronous execution capabilities. An experimental evaluation is done in Section 3.4 followed by a case study about optimizing the MapOverlap2D CUDA implementation by parametric auto-tuning in Section 3.5. Section 3.6 summarizes the chapter.

3.1 SkePU library

SkePU is a C++ template library that provides a simple and unified interface for specifying data- and task-parallel computations with the help of skeletons on GPUs using CUDA and OpenCL. The interface is also general enough to support other architectures, and SkePU implements both a sequential CPU and a parallel OpenMP backend.

SkePU is designed as a C++ template (include) library. Currently, there exist other skeleton programming libraries for GPU-based systems, like SkelCL and Muesli. SkelCL [197] provides some data-parallel skeletons using OpenCL. However, it does not support concurrent execution on CPU and GPU devices nor an intelligent selection mechanism for using CPU or GPU devices. The Muesli skeleton library [90] supports execution on both single- and multi-node MPI-based clusters using OpenMP, MPI and CUDA. However, it does not have an automated mechanism to decide which implementation (OpenMP, CUDA) to use for a given execution context nor does it support concurrent execution on CPU and GPU compute devices. A more detailed discussion of the related work can be found in the related work chapter (Chapter 8).

SkePU provides a simple and unified interface for specifying data- and task-parallel computations with the help of skeletons on both mul-

ticore CPUs and GPUs. SkePU provides multiple implementations of each skeleton: sequential C++ and OpenMP implementations for single- and multi-core CPUs respectively, and CUDA and OpenCL implementations for doing computations on one or more GPUs[1] present in the system. Three main components of the library are generators for user functions, containers and skeletons.

3.1.1 Generation of User functions

Skeletons are generic components that can be parameterized with actual computations in the form of (usually, sequential) user-functions. SkePU provides a *macro language* to specify user functions; it then process them into a form that can be used with the skeletons regardless of the target architecture. The preprocessor macros expand, depending on the target architectures supported, to a structure that contains the target-specific versions of the user function. Technically, SkePU user function generated from a macro based specification is basically a `struct` containing member functions for CUDA and CPU, and strings for OpenCL. Figure 3.1 shows one of the macros and its expansion, specifying a simple multiplication of two float operands `a` and `b`.

3.1.2 Smart containers

To support skeleton operations, SkePU provides `Vector` and `Matrix` containers for storing 1- and 2-dimensional data respectively. The interface of these containers is very similar to the C++ STL vector container. For instance, the following line creates a SkePU vector of size 100 with all elements initialized to value 10:

```
skepu::Vector<float> input(100,10);
```

The containers can also be used to wrap existing C arrays, for example:

```
float arr[SIZE];
...
skepu::Vector<float> input(arr, SIZE);
```

[1]The OpenCL implementations can be used with CPUs as well but they are optimized for GPUs in our case.

```
BINARY_FUNC(mult_f, float, a, b,
    return a*b;
)

           // EXPANDS TO:  ====>
```

```
struct mult_f
{
    skepu::FuncType funcType;
    std::string func_CL;
    std::string funcName_CL;
    std::string datatype_CL;
    mult_f()
    {
        funcType = skepu::BINARY;
        funcName_CL.append("mult_f");
        datatype_CL.append("float");
        func_CL.append(
        "float mult_f(float a, float b)\n"
        "{\n"
        "   return a*b;\n"
        "}\n");
    }
    float CPU(float a, float b)
    {
        return a*b;
    }
    __device__ float CU(float a, float b)
    {
        return a*b;
    }
};
```

Figure 3.1: User function specification and macro expansion.

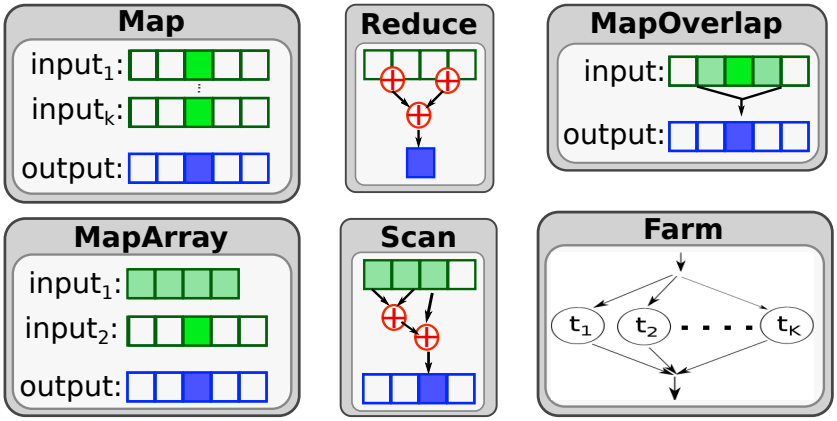


Figure 3.2: A simple illustration of basic skeleton operations.

Although offering an interface very similar to STL containers, SkePU containers internally track different (and possibly overlapping) copies of data in different memory units and do memory management for those copies. This memory management inside SkePU containers allows our framework to optimize data transfers for operands passed to skeleton calls across CPU main memory and GPU device memories. For example, if elements of a container are modified in the GPU memory by a skeleton computation, they are not immediately transferred back to the main memory. The transfer happens only when needed, e.g., when elements are accessed (e.g., via the [] operator) on the host side. This is called *lazy memory copying* and can significantly reduce the communication cost if future GPU computations operate on that data without intermediate CPU accesses. We discuss the memory management and advanced optimizations, that make these containers smart, separately in Chapter 4.

3.1.3 Skeletons

Data parallel skeletons

SkePU provides the data-parallel skeletons `Map`, `Reduce`, `MapReduce`, `MapOverlap`, `MapArray` and `Scan` each with sequential CPU, OpenMP, CUDA and OpenCL implementations. In the object-oriented spirit of C++, the skeleton functions in SkePU are represented by objects (so called functors, overloading `operator()` to resemble the C function call syntax for skeleton calls). A program using SkePU needs to include the SkePU header file(s) for skeleton(s) and container(s) used in the program. A skeleton can be called with whole vectors/matrices as operands, doing the operation on all elements of the vector/matrix. Another way to call them is with iterators [11]. In that case, a start iterator and an end iterator are instead provided which makes it possible to only apply the skeleton on parts of the vector/matrix.

When the skeleton is called the library decides which implementation to use (see Section 3.2). As an example, the following code excerpt

```
skepu::Map<mult_f> elementWiseMult(new mult_f);
```

shows how a skeleton instance called `elementWiseMult` is created by instantiating the Map skeleton with the user function `mult_f` (as described in Figure 3.1) as a parameter. In the current version of SkePU it needs to be provided both as a template parameter and as a pointer to an instantiated version of the user function (remember that the user functions are in fact `structs`). In the following, we briefly describe each of the skeletons (see Figure 3.2 for a graphical representation):

In the *Map*$< f >$ skeleton, every element in the result vector (or matrix) r is computed by a function f applied to the corresponding elements in one or more input vectors (or matrices) $v_1 \dots v_k$. The number of input operands is currently limited to a maximum of three.

The *Reduce*$< \oplus >$ skeleton computes a scalar result by applying an associative binary operator $\oplus$ accumulatively across all elements in the input vector (matrix). For a $R \times C$ matrix operand, an output vector of size R and C is produced instead of a scalar value for row-wise and column-wise reduction respectively.

MapReduce combines mapping and reduction in the same computation kernel and therefore avoids some synchronization, which speeds up the calculation.

MapOverlap is similar to a Map, but each element $r[i]$ of the result vector is a function of several adjacent elements of one input vector that reside within a certain (compile-time) constant maximum distance d from i in the input vector, where d is a skeleton template parameter. For a matrix operand, row-wise (or column-wise) overlap is supported where each element $r[i, j]$ of the result matrix r is a function of several *row adjacent* (or *column adjacent*) elements of one input matrix m that reside within a certain constant maximum distance d from j (or i) in the input matrix. 2D MapOverlap for matrix operands[2] is also supported where each element $r[i, j]$ of the result matrix r is a function of several adjacent elements of one input matrix that reside at a certain constant maximum *logical* distance (i.e., distance in the row or column dimension) from i, j in the input matrix[3]. The maximum row-wise and column-wise logical distance of these elements is controlled by the parameter d_r and d_c respectively. Convolution is an example of a calculation that fits this pattern.

MapArray is another variant of Map where each element of the result vector, $r[i]$, is a function of the corresponding element of one of the input vectors, $v_1[i]$ and any number of elements from the other input vector v_2. For one input matrix m and one input vector v, a result matrix r is produced such that $r[i, j]$ is a function of the corresponding element of the input matrix $m[i, j]$ and any number of elements from the input vector v.

Given a binary associative function $\oplus$, the *Scan*$< \oplus >$ skeleton computes the prefix-$\oplus$ vector of its input vector, such as the prefix sums vector where $\oplus$ is standard addition. Scan is an important basic building block of many scalable parallel algorithms, such as parallel integer sorting.

[2]For denoting full 2D overlaps (e.g., in 2D stencil operations), we use a variant of the MapOverlap skeleton called MapOverlap2D.

[3]The actual access distance between Matrix elements could be different depending on how the Matrix is stored in memory.

Farm - Task parallel skeleton

`Farm` is a task-parallel skeleton which allows the concurrent execution of multiple independent tasks, possibly on different workers. It consists of *farmer* (also called *master*) and *worker* threads. The *farmer* accepts multiple incoming tasks and submits them to different workers available for execution. The overhead of submitting tasks to different workers should be negligible, otherwise the farmer can become the performance bottleneck. The farmer is also responsible for synchronization (if needed) and for returning the control (and possibly results) back to the caller when all tasks finished their execution. The *workers* execute the assigned task(s) and notify the farmer when a task finishes the execution. A *task* is an invocation of a piece of functionality with implementations for different types of workers available in the system[4]. Moreover, a task could itself be internally parallel (e.g., a data parallel skeleton call) or could be another task-parallel skeleton (e.g., another farm), allowing nested parallel executions.

Code example

In Listing 3.1, a complete SkePU source code example of a dot product computation of two vectors is given. It is expressed as a MapReduce data-parallel skeleton with `mult_f` and `plus_f` as provided user functions. In the end, the expected output is shown as a comment.

Multi-GPU support

SkePU has support for carrying out skeleton computations on multiple GPU devices present in the system, using both CUDA and OpenCL. By default, SkePU will utilize as many GPUs as it can find in the system; however, the user can specify an upper limit on the number of GPUs to be used by using the `SKEPU_NUMGPU` flag. The work division across multiple GPUs is based on the idea that data-parallel computations typically have similar computational complexity across

[4]In our farm implementation, a task could define implementations for a subset of worker types (e.g., a task capable of running only on CPU workers).

```
1  #include <iostream>
2
3  #include "skepu/vector.h"
4  #include "skepu/reduce.h"
5
6  BINARY_FUNC(mult_f, float, a, b,
7      return a*b;
8  )
9
10 BINARY_FUNC(plus_f, float, a, b,
11     return a+b;
12 )
13
14 int main()
15 {
16   skepu::MapReduce<mult_f, plus_f> dotProduct(new mult_f,
17                                               new plus_f);
18
19   skepu::Vector<float> v0(1000,2);
20   skepu::Vector<float> v1(1000,3);
21
22   // Call can map to different mapreduce implementations
23   float r = dotProduct(v0, v1);
24
25   std::cout<<"Result: " << r <<"\n";
26
27   return 0;
28 }
29
30 // Output
31 // Result: 6000
```

Listing 3.1: Dot product computation using the MapReduce skeleton.

different elements. Hence, work is divided amongst available GPU devices depending on the computational power of each device which is determined during the initialization phase of the SkePU environment. For example, if we have a map computation on vector objects of size `N` each, executed on two GPUs of the same type, the work is divided equally (to the possible extent) between the two GPUs, i.e., each GPU applies the operation on `N/2` elements. When executing MapOverlap and Scan skeleton calls across multiple GPUs, extra pre-processing is carried out to take care of the data dependencies between element computations in different partitions to enable division of the computation work on multiple devices. Moreover, when executing a Reduce or

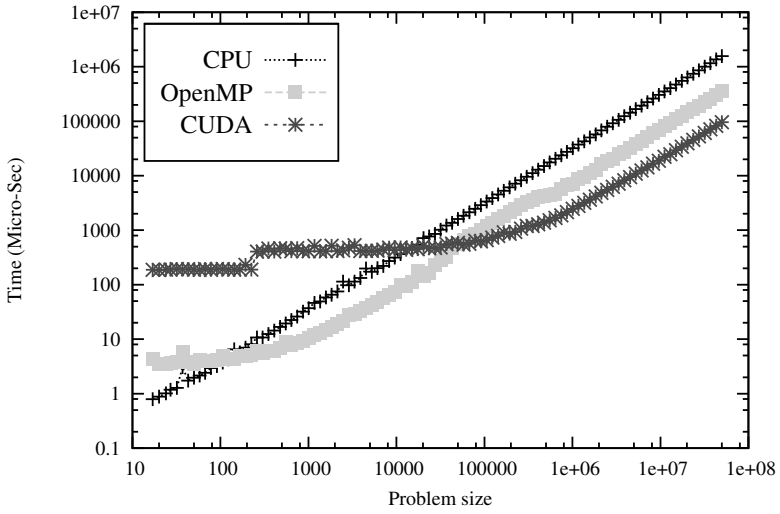


Figure 3.3: Execution of dot product computation for different vector sizes. CPU is sequential execution on 1 CPU core, OpenMP uses 8 CPU threads/cores. Platform: 2 Intel Xeon E5520 CPUs with 1 NVIDIA Tesla C2050 GPU.

a MapReduce skeleton call across multiple GPUs, the final reduction is carried out on a CPU.

We have extended the SkePU framework to optimize multi-GPU execution by leveraging the capabilities of newer CUDA releases. Initially, different CPU threads were required to handle different CUDA GPU devices; however, as this is no longer needed with newer CUDA releases, we removed the threading overhead and use the host thread to manage multiple GPU devices. Moreover, we have improved the memory management (described in Chapter 4) for multi-GPU execution to transfer data between different GPU devices directly if possible. These improvements give significant performance improvements for multi-GPU execution for many applications, as shown later in Section 4.3.

3.2 Implementation selection

SkePU provides skeleton implementations for both CPU and GPU devices present in the system using C++, OpenMP, CUDA and OpenCL. So for each skeleton call in the program, we need to decide whether to execute it on CPU or on a GPU, e.g., using either its C++, OpenMP or CUDA implementation. Figure 3.3 shows execution times of the dot product computation (code in Listing 3.1) on a GPU-based system for different vector sizes. As we can see, the execution with the sequential C++ implementation performs better with small problem sizes; however, for larger problem sizes using OpenMP on 8 cores is relatively faster in this case. For even larger problem sizes, execution on a CUDA GPU supersedes the performance of the other two. The question what type of switching occurs and at what problem sizes cannot be determined statically as it depends upon the skeleton type, actual user computation, operand data locality, target system architecture as well as the problem sizes.

To make this decision automatic for any skeleton call in a program in a portable manner, we devise an offline empirical tuning based technique. Trying out all possible context instances (sizes of operand data) for all implementation variants (CPU, OpenMP, CUDA etc.) using exhaustive search is undesirable and practically infeasible. Instead, our offline tuning technique performs an adaptive hierarchical search [146] based upon a heuristic *convexity assumption* which means that if a certain implementation is performing best on all vertices of a D-dimensional context parameter subspace then we assume it is also the best choice for all points within the subspace. For example, considering a 1-dimensional space (i.e. only 1 input size parameter), if we find out that a certain implementation performs best on two distinct input sizes n_1 and n_2 we consider it best for all points between these points (i.e. for the whole range $[n_1, ..., n_2]$ where $n_1 < n_2$). This concept is extended to D-dimensional spaces as described below.

3.2.1 Algorithm

The training space $C = I_1 \times ... \times I_D$ of context instances for a skeleton with D possibly performance-relevant properties in the context instances is spanned by the D context property axes with considered (user-specified or default) finite intervals I_i of discrete values, for $i = 1, ..., D$. A continuous subinterval of an I_i is called a (context property value) *range*, and any cross product of such subintervals on the D axes is called a *subspace* of C. Hence, subspaces are "rectangular", i.e., subspace borders are orthogonal to the axes of C.

Our idea is to find sufficiently precise approximations by adaptively recursive splitting of subspaces by splitting the intervals I_i, $i = 1, ..., D$. Hence, subspaces are organized in a hierarchical way (following the subspace inclusion relation) and represented by a 2^D-ary tree T_C.

Our algorithm for off-line measurement starts from a trivial tree T_C that has just one node, the root (corresponding to the whole C), which is linked to its 2^D corner points (here, the 2^D outer corners of C) that are stored in a separate table of recorded performance measurements. The implementation variants of the skeleton under examination are run with generated context instances, for each of the 2^D corners; a variant whose execution exceeds a timeout for a context instance is aborted and not considered further for that context instance. Now we know the winning implementation variant for each corner point and store it in the performance table, too, and T_C is properly initialized.

Consider any leaf node v in the current tree T_C representing a subspace $S_v = R_1^v \times ... \times R_D^v$ where $R_i^v \subset I_i$, $i = 1, ..., D$. If the same specific implementation variant runs fastest on all context instances corresponding to the 2^D corners of S_v, we stop further exploration of that subspace and will always select that implementation whenever a context instance at run-time falls within that subspace. Otherwise, the subspace S_v may be refined further. Accordingly, the tree is extended by creating new children below v which correspond to the newly created subspaces of S_v. By iteratively refining the subspaces in breadth-first order, we generate an adaptive tree structure

to represent the performance data and selection choices, which we call *dispatch tree*.

The user can specify a *maximum depth* (training depth) for this iterative refinement of the dispatch tree, which implies an upper limit on the runtime lookup time, and also a maximum tree size (number of nodes) beyond which any further refinement is cut off. Third, the user may specify a timeout for overall training time, after which the dispatch tree is considered final.

At every skeleton invocation, a run-time lookup searches through the dispatch tree starting from the root and descending into subspace nodes according to the current runtime context instance. If the search ends at a *closed leaf*, i.e., a leaf node with equal winners on all corners of its subspace, the winning implementation variant can be looked up in the node. If the search ends in an *open leaf* with different winners on its borders (e.g., due to reaching the specified cut-off depth), we perform an approximation within that range by choosing the implementation that runs fastest on the subspace corner with the shortest Euclidean distance from the current run-time context instance.

The deeper the off-line training algorithm explores the tree, the better precision the dynamic composer can offer for the composition choice; however, it requires more off-line training time and more runtime lookup overhead as well. We give the option to let the user decide the trade-off between training time and precision by specifying the cut-off depth, size and training time limit. Figure 3.4 shows an example for 1-dimensional space exploration. The algorithm can recursively split and refine subspaces until it finds common winners for all points for a subspace (i.e. the subspace becomes closed) or the user-specified maximum depth is reached.

In another work [145], we have described some further optimizations to this iterative refinement process based on the convexity assumption. These optimizations include *oversampling* and *thresholding*. In oversampling, a closed subspace is explored and refined one (or more) level(s) deeper in order to gain additional confidence whether the common winner found on all corner points of a subspace is indeed a good choice for intermediate points within that subspace. In thresholding, we relax the condition on closing a subspace S_v where

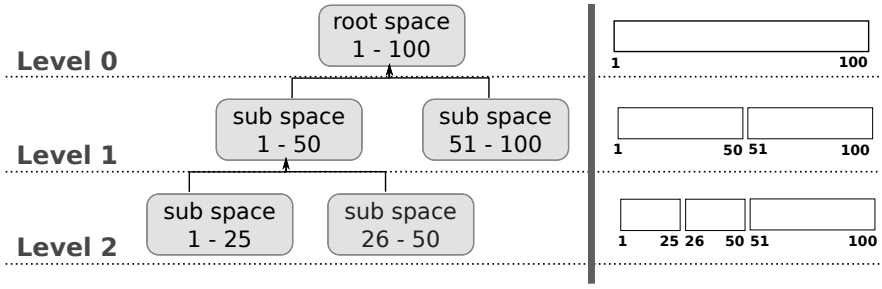


Figure 3.4: Depiction of how a 1-dimensional space is recursively cut into subspaces (right) and the resulting dispatch tree (left).

we do not have a common winner on all 2^D corners of the subspace. A user-specified tolerance threshold value θ is used to close such a subspace (without a common winner on all points) if there exists some specific implementation variant i whose execution time $t_{i,p}$ on each corner point p ($\forall p \in 2^D$ corners of S_v) is either minimal or is within the specified tolerance threshold value in comparison to the execution time of the fastest variant $t_{f,p}$ on that point, i.e., $|t_{i,p} - t_{f,p}| / t_{f,p} \leq \theta$; more details on these optimizations can be found in [145].

3.2.2 Implementation details

A `Tuner` class is introduced (as shown in Listing 3.2) which is parameterized by the skeleton type and user function(s). The user needs to supply a unique `ID` (string) for the *skeletonlet*[5] being tuned as well as lower and upper bounds for the size of each operand. The `ID` decouples the skeletonlet and tuner, and allows, e.g., multiple tuning scenarios even for the same skeletonlet to co-exist. Internally the tuner applies certain optimizations (e.g., dimensionality reduction) and returns one or more execution plans (one for each data locality scenario as discussed later) which are later assigned to the skeleton object (Listing 3.2). An execution plan is a simple data structure that internally tracks the best implementation for each subspace and provides lookup facilities. After the execution plans are set, the expected

[5] A pair of user-function(s), skeleton type.

best implementation for any skeleton in a given call context will be automatically selected.

The `Tuner` supports automatic persistence and loading of execution plans. If an execution plan with same configuration already exists, the `Tuner` loads and returns it from a repository without any tuning overhead; otherwise it invokes the tuning algorithm and constructs an execution plan. The generated execution plans are stored for future usages to avoid re-tuning every time the skeleton program is executed. Furthermore, the tuning and actual execution can happen during the same program execution, as shown in Listing 3.2. When porting the same skeleton program to a new architecture, the tuner would automatically construct execution plans for the new architecture without requiring any changes in the user program.

```
1  ...
2
3  int main()
4  {
5    skepu::Map<mult_f> vecMultiply(new mult_f);
6
7  /* specify lower and upper bounds for training range */
8    const int D = 3; // number of dimensions...
9    int lowerBounds[D] = {10, 10, 10};
10   int upperBounds[D] = {50000000, 50000000, 50000000};
11
12 /* invoke the tuner and get list of execution plans */
13
14   skepu::ExecPlan plans[skepu::MAX_PLANS];
15   skepu::Tuner<mult_f,MAP> tuner("vMult", D, lowerBounds,
16                                              upperBounds);
17   tuner(plans);
18
19 /* assign the execution plans to the skeleton object */
20
21   vecMultiply.setExecPlan(plans);
22
23   ...
24 }
```

Listing 3.2: Tuning the vector multiply skeleton call. Multiple execution plans are created, one for each possible data locality scenario.

Dimensionality reduction

We apply several optimizations based on specific knowledge that each skeleton implementation exposes. For example, considering the fact that all vector operands (inputs, output) in a map skeleton call should be of exactly the same size, we consider it as a 1-dimensional space instead of a 3-dimensional space (with 2 input and 1 output vector operands). This significantly reduces the training cost and is transparently done by considering the type of the skeleton and container being used. Similar optimizations are applied for MapOverlap, MapReduce and Scan skeletons.

Data locality

Current GPU based systems internally have disjoint physical memory and both the vector and matrix containers in our skeleton library can track their payload data on different memory units. The operand data locality matters when measuring the execution time for both CPU and GPU execution for a skeleton implementation. Operand data may or may not exist in the right memory unit; in case it is not available in the right place, extra overhead for data copying needs to be encountered. Selection of the expected best implementation for a given problem size cannot be made without considering where the input data resides at the call execution time and whether the output data needs to be copied back. This is so because the data copying overhead between different memory units could affect the selection of the best performing variant. For example, for the following skeleton call:

```
map(v0, v1, out); // v0,v1: read; out: written
```

for a given vector size (e.g. 1000) on a certain platform, the execution with the CUDA implementation may be the best choice when all input vectors (`v0,v1`) are already present in a GPU device memory and the output vector (`out`) does not need to be transferred back to main-memory; otherwise, perhaps execution using the OpenMP implementation may prove the best choice. Obviously, different data-locality scenarios for operands are possible, such as `v0` being available

in a GPU device memory whereas `v1` is not.

One solution could be to assume that operand data is always located in a specific memory unit (e.g., main memory) and, depending on where the skeleton implementation executes, a copy may or may not be required. This solution is simple but inflexible as even different operands of a single skeleton call may reside at different memory spaces depending on their previous usage with other skeleton calls and program control flow. On the other hand, delaying the decision about operand data locality to runtime is infeasible as we need to know the data transfer cost in order to determine, offline, the best variant for a given problem size, which is used to decide whether to close a subspace or to continue exploring it further by recursively partitioning it into multiple subspaces. We have implemented the following two mechanisms to tackle this issue:

- *User-specified:* The programmer can provide hints about operands' data locality by specifying the memory unit (an integer flag) for each skeleton operand. This gives the programmer more control over the tuning process which could be helpful in more complex data access scenarios. However, asking the programmer for hints about data locality requires knowledge about the underlying system and is thus undesirable.

- *Automatic:* When training offline, we construct one execution plan for each possible data locality scenario and store all possible execution plans. The number of data locality scenarios is limited considering that 1) we have at most 4 operands to a skeleton call and 2) we can reduce the possible scenarios considering the fact that most skeleton operands must have the same size (e.g., all operands to a map call have the same size)[6]. At runtime, the framework can automatically choose the right execution plan, for making implementation selection decisions, by querying the

[6]For example, in a map skeleton call with two input vectors `v0,v1`: scenario where `v0` is available in a GPU device memory whereas `v1` is not, is considered similar from the data locality perspective to the one where `v1` is available in a GPU device memory whereas `v0` is not. This is due to the fact that the size of `v0` and `v1` is same.

operand data state from the containers. Our vector and matrix containers internally track the state of their payload data copies in different memory units, which can be retrieved at runtime. This approach alleviates the burden of specifying data locality hints from the programmer, as shown in Listing 3.2; however, it may require (slightly) higher training cost as a subspace would only be considered closed if it is closed for all possible operands' data locality scenarios, as explained below.

In the *Automatic* case, we do not run each skeleton implementation for all possible data locality scenarios as this could increase the training cost multiple-fold. Rather, we execute each skeleton implementation once with operand data available in the right memory unit for that implementation. This gives us the execution time (without communication time) and then we compute the effect of other possible data locality scenarios by adding the possible communication overhead when some (or all) operand data is not available in the right memory unit. The communication overhead is calculated by multiplying communication size with effective communication bandwidth between different memory units which is determined by running micro-benchrmarks, normally once on each new platform. For example, for a CUDA skeleton implementation, we measure its execution time with all input operands available in the device memory. Afterwards, we add possible communication overhead when some (or all) operands are not available in the device memory and/or when output data needs to be transferred back to main-memory. This way, we keep the training cost similar to the training cost incurred with user-specified hints where only one (user-specified) data-locality scenario is considered.

In the automatic case, a subspace is closed only when we find a common winner on all its 2^D corner points for all possible data locality scenarios; otherwise, we continue exploring and pruning the subspace further until we find such a common winner for each subspace or until the user-specified maximum training depth has been reached (or some other exit criterion is met such as maximum training time etc.). Intuitively, this would require more training exploration as closing a

subspace by finding a common winner on all corner points for all possible scenarios is less probable. The experiments (see Section 3.4) have shown up to 30% increase in training space exploration when considering multiple scenarios. However, it is not much considering that the training space explored is still less than 1% of the total training space, as shown in the experiments later.

3.2.3 Related work

Techniques for automated performance tuning have been considered extensively in previous work for generation of optimized domain-specific libraries (such as basic linear algebra [202, 210], reduction [215], sorting [147] or signal transforms [194, 102]), iterative compilation frameworks [171], or for the component composition [131, 13, 133]. Active Harmony [64] is a generic performance tuning framework for tuning both algorithms' and parameters' choices for a library. It provides several built in strategies (such as random search, exhaustive search and search based on simplex methods) for exploring a search space.

Automated performance tuning usually involves three fundamental preparatory tasks: (1) search through the space of context property values, (2) generation of training data and measurements on the target system, (3) learning a decision function/rule. In our approach, these three tasks are tightly coupled to limit the amount of measurement time and representation size required, while most other approaches decouple at least two of these tasks.

Our approach can be considered as an adaptive variant of decision tree learning. Decision tree learning, often based on C4.5 [183] or similar tools, is also used in many other approaches, e.g. in [202, 194, 215].

3.3 Runtime support for hybrid execution

The implementation selection mechanism described above can help making selection decisions between, e.g., usage of CPU and GPU for a given skeleton call. However, in some cases, better performance can be achieved instead with using both CPU and GPU devices by di-

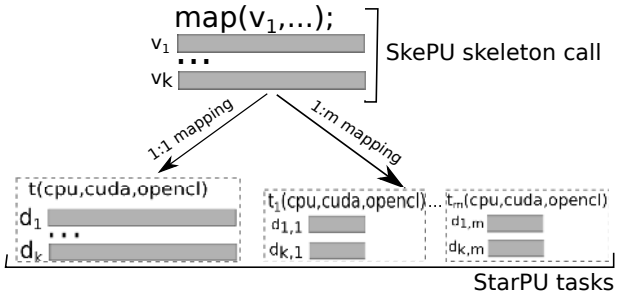


Figure 3.5: Shows mapping of a SkePU skeleton call to one or more StarPU tasks.

viding the computation work of a given skeleton call (called hybrid execution). To allow hybrid execution, we have implemented support for the StarPU task-based runtime system as a possible backend in our library. When enabled, skeleton calls in a normal SkePU skeleton program are translated to tasks (with different implementations) in the StarPU runtime system. Each task is then executed by some StarPU (CPU, CUDA or OpenCL) worker; the decision is made with the help of a scheduling and selection policy inside the runtime system. Figure 3.5 shows how a skeleton call in our library is translated to one or more tasks in the StarPU runtime system. Multiple tasks are created by partitioning input operand data for data-parallel skeletons which consequently divides the computation work. By default, tasks operating on (almost) equal-sized chunks are created. A special mechanism is implemented to allow tasks with unequal-sized chunks of data if necessary.

For the experiments, we have divided the computation into chunks of equal size which are scheduled by the runtime system concurrently on different devices available[7]. However, a more informed decision can be made in case we know a linear performance model for execution time of that computation on different devices. For example, suppose that we want to divide a computation's work between a CPU and a

[7]The number of chunks/tasks can be specified as a last argument to skeleton calls, e.g. `map(v0, 4)` where `4` is the number of chunks/tasks that gets created.

GPU device where performance models for the computation's execution time on both devices are available. Now for a given problem size N, we want to find the fraction of work δ that should be mapped to the CPU; consequently fraction $1-\delta$ of work should be mapped to the GPU. With such division of work, the projected application execution time would be

$$\max(T_{cpu}(\delta N), T_{gpu}((1-\delta)N))$$

where $T_{cpu}(\delta N)$ is the execution time on CPU for a fraction δ of the problem size N and $T_{gpu}((1-\delta)N)$ is the time on the GPU for the remaining fraction. Finding the optimal load balance means finding the value of δ which minimizes the above term. Assuming linear performance models for predicting T_{cpu} and T_{gpu}, as is the case for most of our skeletons, this amounts to finding the value of δ at which $T_{cpu}(\delta N)$ and $T_{gpu}((1-\delta)N)$ intersect [151].

When used with StarPU, the memory management for operand data is done by the StarPU runtime system and not by the SkePU containers. The containers, in this case, handle the interaction with the StarPU data management library to coordinate accesses to their operand data. For example, they implement logic to register their data for usage with runtime tasks and allow read and write accesses from the program source code for data that is registered for usage with the runtime system. Moreover, the StarPU runtime system can execute tasks in an asynchronous manner which allows to exploit parallelism between different independent tasks. Using this mechanism, we implemented support in our library to run skeleton calls asynchronously without bothering the skeleton user with the concurrency issues. When using StarPU as backend, each skeleton execution can be done asynchronously where data dependencies between skeleton calls are implicitly tracked by the runtime system. Implicit synchronization takes place inside the SkePU vector and matrix container implementations when the user accesses data for which some skeleton operations are still in progress (see Section 4.4 for a code example). This enables us to automatically exploit task-level parallelism even across multiple data-independent skeleton calls. In the next chapter (Section 4.4), we discuss the containers' working with StarPU in more

detail.

3.4 Evaluation

We have implemented several applications/kernels using SkePU skeletons such as LU factorization, NBody simulation and Smooth Particle Hydrodynamics (SPH)[8]. The main evaluation is carried out on two GPU-based systems:

A 2 Intel Xeon quad-core E5520 2.27GHz CPUs with 1 NVIDIA Tesla C2050 GPU.

B 2 Intel Xeon quad-core X5550 2.66GHz CPUs with an old Tesla C1060 GPU.

The compilation is carried out using GCC (v4.6) and NVIDIA C (nvcc v0.2.1221 with CUDA 4.2) compiler for C/C++ and CUDA code respectively.

3.4.1 Tuning implementation selection

We used five applications (NBody simulation, **S**mooth **P**article **H**ydrodynamics, LU factorization, Mandelbrot, Taylor series) and four kernels (**M**ean **S**quared **E**rror, **P**eak **S**ignal-to-**N**oise **R**atio, **P**earson **P**roduct-**M**oment **C**orrelation **C**oefficient, dot product) for evaluating implementation selection capability. For each application/kernel, we call the tuner on a given training range (i.e., problem size ranges for each operand as shown in Table 3.1 for different applications) and it internally explores some points in the training space and constructs execution plans using the algorithm described in Section 3.2.1. Afterwards, for the actual execution, we select a set of sample points (different from the training points) within that range and do the actual execution using the tuned version as well as using each implementation variant (CPU, OpenMP, CUDA) on those selected points

[8] All applications' source codes are made available as part of the SkePU library.

[9]. As it is practically infeasible to try out all points in the training range, accuracy is measured by averaging over the ratio of execution time with the tuned configuration to the execution time of the best from direct execution (CPU, OpenMP, CUDA) over all sample points. The same problem size ranges are used for experiments on both systems and no modifications in the program source code are made when porting the applications between both systems. Furthermore, for all experiments, we set the maximum training depth to 20 and Euclidean distance is used to estimate the best variant if no best variant is found for a subspace until depth 20.

Table 3.1: Execution of nine applications/kernels for different problem sizes on System A and System B with respective *accuracy – training space explored* figures (in percent). Training ranges for each application are shown in the first column (*50m* means *50 million*).

Name	System A	System B
Taylor (10 — 50m)	99.3 – 0.0002	97.9 – 0.00008
MSE (10 — 50m)	95.0 – 0.0001	99.3 – 0.00009
PSNR (10 — 50m)	93.6 – 0.0002	97.3 – 0.00009
PPMCC (10 — 50m)	95.5 – 0.0006	94.2 – 0.00005
Dot Product (10 — 50m)	95.7 – 0.0002	93.1 – 0.00009
NBody (8 — 10648)	97.9 – 0.35	71.1 – 0.43
LU (4 — 1600)	94.2 – 1.88	99.7 – 2.38
Mandelbrot (16 — 10000)	81.1 – 0.66	74.5 – 0.004
SPH (4 — 1600)	112.0 – 2.38	111.5 – 2.46
Average	96.0 – 0.58	93.2 – 0.59

As shown in Table 3.1, on average, we achieved 96% and 93.2% accuracy on System A and System B respectively with less than 0.6%

[9]We did not consider the OpenCL implementations for experiments as they are similar to CUDA in performance on NVIDIA GPUs and are primarily written for execution on AMD GPUs not supporting CUDA.

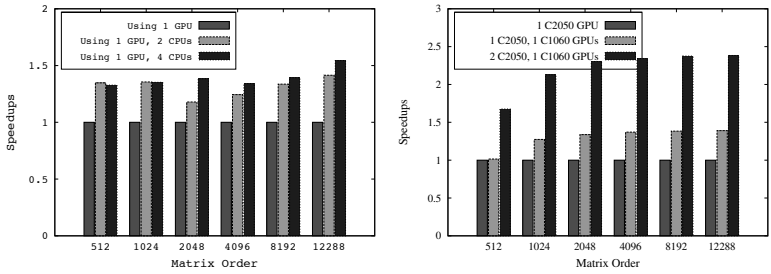


Figure 3.6: Coulombic potential grid execution on System A (left) and System B with two extra C2050 GPUs added (right) for different matrix sizes. The base-line is execution on a single GPU.

training space exploration. Above all, this is achieved without requiring any changes in the program source code when porting to a new system/architecture. For the SPH application, we achieved more than 100% accuracy on both systems which is possible considering that the SPH application internally consists of multiple types of skeleton calls executed inside a loop; selecting the best implementation for each skeleton call individually (as done by our tuner) can supersede any best static implementation choice (CPU, OpenMP, CUDA) in such a scenario.

The performance difference between the two mechanisms to specify data locality information (see Section 3.2.2) is little. The results presented above are for SkePU code with user-specified data locality hints. Execution when decision is made automatically by dynamically looking up operand data state achieves, on average, 94% (91.5%) accuracy with 0.78% (0.71%) training space exploration on System A (System B). Slightly more training space exploration is needed as each training subspace needs to be closed only when it is closed for all possible operand data state/locality scenarios. Also, the lookup at runtime to find the data state of each operand state and querying the right execution plan results in slight overhead which decreases the accuracy, overall, by 1 or 2 percent.

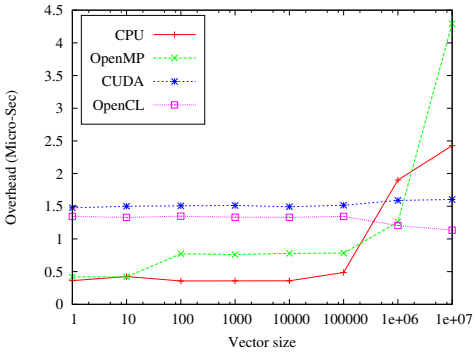


Figure 3.7: The overhead of different SkePU implementations over different vector sizes on System A. Both CUDA and OpenCL executions are made on the GPU; OpenMP is used with 8 threads/cores except for vector size 1.

3.4.2 Hybrid execution

Hybrid execution enables us to use multiple (types of) computing resources present in the system simultaneously for a given computation by dividing the work between them. Figure 3.6 shows the performance of executing a Coulombic potential application [211] written using SkePU skeletons on two GPU based systems. Just like other experiments presented in this thesis, the overhead of data transfer for GPU execution is included in the measurements. Significant speedup is achieved by dividing the work across multicore CPUs and the GPU as shown in Figure 3.6(left). The speedup over GPU execution comes from dividing the work between GPU and one or more CPUs which not only divides the computation work but also *decreases* the data communication to GPU. As shown in Figure 3.6(right), when porting the same code to another system with 3 GPUs, speedups upto `2.5` times are achieved with respect to execution on a single GPU.

3.4.3 SkePU Overhead Analysis

Considering that performance is a critical factor for usage of GPU-based systems, we have designed SkePU as a C++ template library to minimize the runtime overhead.

To measure the overhead of SkePU framework, we execute a simple program with a map skeleton call assigning element-wise square of contents in one input vector (v_{in}) of size N, to an output vector (v_{out}),

i.e.

$$v_{out}[i] = v_{in}[i]^2 \text{ where } i = 0, ..., N-1$$

The actual computation does not matter as we are interested in the SkePU overhead and not the computation itself. To measure the overhead, we execute the map computation with all four skeleton implementations (CPU sequential, OpenMP, CUDA and OpenCL). For each execution, we measure the overhead by deducting the time spent on the actual computation from the total execution time. The actual computation time is measured by placing timers around computation code inside each map implementation inside the SkePU framework. The total time is measured by placing timers around the skeleton call in the main program. Figure 3.7 shows the overhead in micro-seconds for all four implementations over different vector sizes. Both CUDA and OpenCL executions are carried out on the GPU. The time for transferring operand data to GPU device memory for CUDA and OpenCL executions as well as the time for compiling the OpenCL kernel is not considered as part of the SkePU overhead[10]. As shown in the figure, the overhead of different SkePU implementations is usually around one micro-second even for larger problem sizes. This is negligible considering that computations modelled with SkePU skeletons (and even computations considered for GPU execution in general) usually have much coarser granularity such as a few dozen milliseconds or more. With OpenMP and CPU, the overhead slightly increases for larger problem sizes, but it remains negligible in comparison to the time spent on the actual computation, e.g., 0.0007% and 0.0013% of the computation time, for problem (vector) size 10^7, for CPU and OpenMP respectively. Similar behavior is observed for the other SkePU skeletons.

Next, we compare the SkePU overhead by comparing a real-world application execution with SkePU framework to execution with its direct hand-coded implementations. Figure 3.8 compares SkePU execution of a Runge-Kutta ODE solver [138] with direct execution using

[10]OpenCL kernels are built once when the skeleton object gets created. Regarding communication overhead, we transfer the input vector (v_{in}) to the device memory before we start overhead measurements for CUDA/OpenCL executions.

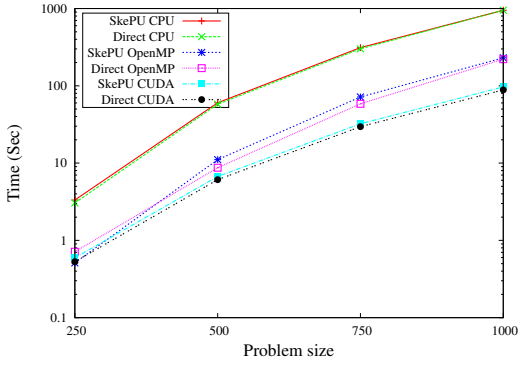


Figure 3.8: Performance comparison for a Runge Kutta ODE Solver application [138] for execution using SkePU in comparison to execution using a hand-coded version for three different backends (CPU C++, OpenMP, CUDA) on System A.

Table 3.2: StarPU task overhead for different number of CPU and GPU workers, measured on System A.

CPU Workers	CUDA Workers	Task Overhead (microseconds)
1	0	3.21
4	0	7.04
8	0	11.03
0	1	3.40
1	1	5.95
4	1	9.12
8	1	44.72

a hand-written implementation for C++, OpenMP and CUDA. The overhead of SkePU implementations is negligible in all cases for even such a large application containing 9 different types of skeleton calls executed over 1160 times, in total.

When executing skeleton programs with StarPU support implemented inside the SkePU library, the main runtime overhead comes from StarPU as it manages both data management for operand data and the execution of skeleton calls. StarPU overhead for a single task is reported to be up to 2 microseconds [19]; however, it could vary depending on the underlying platform and the StarPU configuration

(number/type of workers, scheduling and selection policy etc.) that is used. For example, Table 3.2 shows the overhead that we measured on our system (System A) by running the benchmark given with StarPU package (v1.2, revision 10731) with the default greedy StarPU scheduler. The table shows how the overhead changes when different numbers of CPU and GPU workers are enabled on the target system where the tasks are distributed between all these workers. However, in many cases, some of this overhead is compensated by the StarPU runtime system, by overlapping communication and computation, prefetching data on the target memory unit and several other optimizations.

3.5 Case study on MapOverlap2D on CUDA

Writing CUDA implementations that not only run correctly across different generations of GPUs but also provide reasonable performance in comparison to the maximum achieveable performance on a given GPU is not an easy task. Modern NVIDIA GPUs differ in several performance-relevant architectural features such as availability of L1/L2 caches, number of multiprocessors, number of registers etc. These features significantly affect the performance of running computations, as we will see later.

In this section, we describe our work on optimizing the CUDA implementation for the MapOverlap2D skeleton. As described earlier, in the MapOverlap2D skeleton, each element $r[i, j]$ of the result matrix r is a function of several adjacent elements of one input matrix that reside at a certain constant maximum *logical* distance from position i, j in the input matrix. The maximum row-wise and column-wise logical distance of these elements is controlled by user-function parameters d_r and d_c respectively, as shown in Listing 3.3. For instance, 2D convolution, which is a widely used kernel in image and signal processing applications, can be implemented using this MapOverlap2D skeleton.

We have implemented two different syntaxes to specify computations using the MapOverlap2D skeleton. Listing 3.3 shows how a 2D

```
1  //includes...
2
3  //OVERLAP_FUNC_2D(name, type, d_r, d_c, elem, stride, body)
4  /* Blur kernel (3x3)
5     1 2 1
6     2 4 2
7     1 2 1 */
8  OVERLAP_FUNC_2D(blur_f, float, 1, 1, a, stride,
9    return (a[-1*stride-1] + a[-1*stride]*2 + a[-1*stride+1]
10          + a[-1]*2 + a[0]*4 + a[1]*2 + a[1*stride-1]
11          + a[1*stride]*2 + a[1*stride+1])/16;
12 )
13
14 int main()
15 {
16   skepu::MapOverlap2D<blur_f> conv2D(new blur_f);
17
18   skepu::Matrix<float> m_in(...);
19   skepu::Matrix<float> m_out(...);
20
21   // load input image into m_in...
22
23   conv2D(m_in, m_out);
24
25   // store output image from m_out...
26 }
```

Listing 3.3: SkePU 2D convolution for a blur 3×3 kernel.

convolution computation with 3×3 blur kernel can be implemented using the SkePU MapOverlap2D skeleton. The blur kernel is specified as a user-function (`blur_f`) where d_r and d_c specify row-wise and column-wise overlap on one side (which is 1 in this case) respectively; the overall row-wise and column-wise overlap becomes of size $(d_r \times 2+1) \times (d_c \times 2+1)$. The same computation can be expressed by passing a filter *weight matrix*, with weights for neighbouring and actual elements involved in the overlap (as shown in Listing 3.4), to the skeleton call. The size of the weight matrix, in this syntax, determines the overall overlap size. In this syntax, an explicit user-function is not needed as the computation is implicitly specified by the weight matrix.

In many cases, any of the two syntaxes can be used but the former (user-function based) is in general more flexible whereas the latter

(weight matrix) allows for more aggressive optimizations (putting the weight matrix in GPU constant memory etc.). In the following, we discuss optimizations where the syntax with the weight matrix is used; however, shared memory and tiling optimizations discussed below can also be applied with the other syntax.

```
1 skepu::Matrix<int> m_weights(3,3);
2
3 // set m_weights = [1, 2, 1, 2, 4, 2, 1, 2, 1]
4
5 conv2D(m_in, m_out, m_weights);
```

Listing 3.4: SkePU 2D convolution with blur 3×3 kernel.

3.5.1 Memory optimizations

A simple CUDA implementation of MapOverlap2D is where each CUDA thread processes one element in the output matrix by loading all the neighboring elements of that element (and that element itself) from the GPU global (device) memory. This implementation suffers from too much communication to the global memory as, even for a small 3×3 blur filter (shown in Listing 3.3), it would load each element in the input image as much as 9 times. The communication to GPU global memory can be up to 100 times more expensive than accesses to shared and constant memory [168]. This could really hurt the overall performance especially on GPU architectures with no L1 and L2 caches where high latency is observed for all global memory accesses. On modern GPUs with L1 and L2 caches, global memory accesses can sometimes be served from caches with quite low latency.

We implement the shared memory optimization where threads in a thread block collectively load all input elements that they need in the shared memory and then access it from there. This optimization can significantly reduce the global memory bandwidth consumption, even for small filter sizes. For example, for a thread block of 16×32 and filter size of 3×3, each thread block loads 4608 values without usage of shared memory. With shared memory, the loads from global memory are reduced to $(16 + 3 - 1) \times (32 + 3 - 1) = 612$ values per

thread block, a reduction by a factor of 7.5. The optimization may not yield that much difference in performance while executing on GPUs with caches such as NVIDIA Fermi GPUs.

As the elements in the weight matrix are read by different threads in parallel, they can be stored in the GPU constant memory. The constant memory in GPU is a limited (normally 64KB) cache-buffer memory that can be used for read-only data stored in GPU device memory. It is hardware optimized for the case when all threads read the same location. The access latency for constant memory ranges from one cycle for in cache data to hundreds of cycles depending on cache locality. The performance gains from this optimization depend upon the architecture; it could yield up to 100% performance improvement on non-cache GPUs (e.g. NVIDIA GPUs before the Fermi architecture). However, for NVIDIA GPUs with explicit L1 cache, the performance improvement with usage of constant memory can be much less due to implicit caching and reuse capabilities of these architectures. For instance, for NVIDIA C2050, performance improvement up to 30% is observed over different filter sizes.

3.5.2 Adaptive tiling optimization

Besides memory optimizations mentioned above, another optimization that can be applied to this class of applications on modern GPUs is known as $1 \times N$ tiling [187]. A *tile* refers to a block of input data simultaneously processed by multiple threads in a thread block. $1 \times N$ tiling refers to a technique of increasing the workload for a thread block by assigning N tiles to a thread block for processing instead of 1 tile. Figure 3.9 shows how work assigned to a single thread block increases as each thread processes 2 output elements instead of one with 1×2 tiling. Tiling reduces the number of thread blocks by a factor of N. Besides reducing the overhead associated with thread blocks (e.g. array index calculations, loading constant values), this technique also decreases the amount of overall neighborhood loads to the shared memory as the number of thread blocks is decreased.

By default, each thread in a thread block processes one output element; this means that a thread block of size $blockSizeX \times blockSizeY$

processes one tile (i.e., $blockSizeX \times blockSizeY$ output elements). To process one such tile, threads in a thread block collectively load $(blockSizeX + filterSizeX - 1) \times (blockSizeY + filterSizeY - 1)$ input elements from the global memory to the shared memory where $filterSizeX \times filterSizeY$ is the filter size. By increasing horizontal tiling for each thread block by one, as shown in Figure 3.9, each thread in a thread block will process two output elements instead of one, i.e., a thread block of size $blockSizeX \times blockSizeY$ will now process two tiles ($2 \times blockSizeX \times blockSizeY$ output elements). Processing one extra (horizontal) tile would require it to load $blockSizeX \times (blockSizeY + filterSizeY - 1)$ more input elements to the shared memory. The number of loads to the shared memory for processing an extra tile are lower than when this tile is processed by a separate thread block, i.e., $(blockSizeX + filterSizeX - 1) \times (blockSizeY + filterSizeY - 1)$. This could result in major savings especially for very large horizontal filter size ($filterSizeX$) and/or for large $blockSizeY$.

Despite of its potential advantages, tiling can also result in increased shared memory usage by a thread block as now each thread block processes N elements instead of 1. Similarly, register usage can also increase as extra registers are normally used to save intermediate results.

As shown by van Werkhoven et al. [208], using any fixed tiling factor for an image convolution application can result in sub-optimal performance over different filter sizes. Furthermore, with a fixed tiling factor (e.g. 4), the program may simply not work on certain GPUs due to resource limitations (e.g. shared memory size, number of registers). Rather, the tiling factor can be chosen based on computational (filter size, image size etc.) and GPU resource information (size of shared memory, total number of registers etc.) at runtime, i.e., *Adaptive tiling*. The concept is similar to the one introduced in [208], but we design different metrics to decide the adaptive tiling factor, and evaluate them on three different generations of NVIDIA GPUs. Moreover, we focus on a generic skeleton implementation that can work for any computation that can fit the MapOverlap2D skeleton.

The dynamic selection of the tiling factor is interesting for several

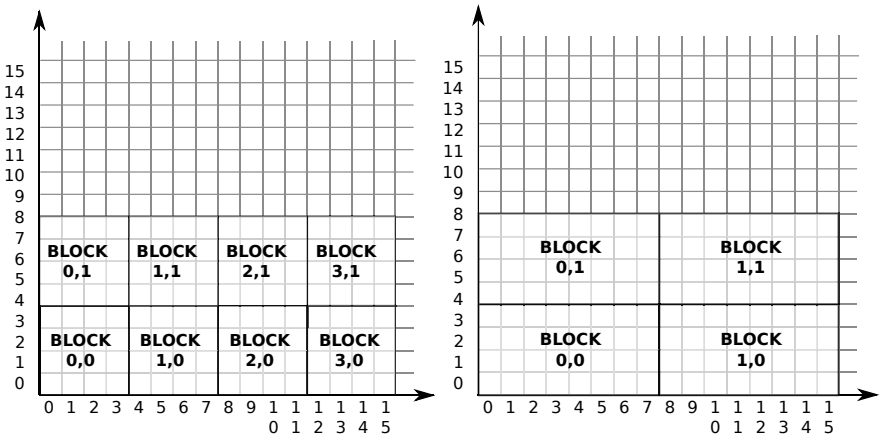


Figure 3.9: The effect of 2-way tiling on thread blocks with 16 threads (4×4): work assigned to a single thread blocks doubles (8×4, see right figure) and the total number of thread blocks is reduced by half. Logical (global) thread indexing is shown on x- and y-axis for x- and y-dimensions respectively.

reasons. First, there could be several different mechanisms to determine the tiling factor based on different performance characteristics. Furthermore, an automatic way of determining the tiling factor over different machine and problem configurations can help in attaining performance when porting the computation across different GPUs.

Maximizing resource utilization

Modern GPUs have many types of resources that can affect the performance of an executing kernel. These resources can be broadly categorized into *computational resources* (e.g., processing cores inside a streaming multiprocessor) and *storage resources* (e.g., registers and shared memory). Effective usage of both kind of resources is important for performance but sometimes a tradeoff exists as both cannot be optimized for at the same time.

The adaptive tiling is focused on maximizing utilization of storage resources of a multiprocessor such as shared memory and registers. On the other hand, *warp occupancy* (or short occupancy) strives for

Algorithm 1: getTilingForMaxOccupancy

Input: $cInf$, information about computation; $gInf$, information about GPU; $maxTiling$, maximum tiling factor
output: $nTiles$, the tiling factor

```
1  begin
2      nTiles ← 1
3      nTrdsPerBlk ← cInf.blockSizeX × cInf.blockSizeY
4      nWarpsPerBlk ← ⌈nTrdsPerBlk / gInf.WARP_SIZE⌉
5
       /* compute max (thread) blocks that can fit in a multiprocessor  */
6      maxBlksPerMP ← min(gInf.NWARPS_MP / nWarpsPerBlk,
7          gInf.NTRDS_MP/nTrdsPerBlk, gInf.NBLKS_MP)
8
       /* adjust #blocks considering # of registers                     */
9      blksPerMPByReg ← gInf.NREGS_MP / (cInf.perTrdNRegFor[nTiles]
10         × nWarpsPerBlk ×gInf.WARP_SIZE)
11     show error if blksPerMPByReg ≤ 0
12     blksPerMPByReg ← min(blksPerMPByReg, maxBlksPerMP)
13
       /* adjust #blocks considering amount of shared memory            */
14     shMemPerBlk ← (cInf.blockSizeX + cInf.filterSizeX - 1) ×
15         (cInf.blockSizeY + cInf.filterSizeY - 1) × sizeof(type)
           /* type is element type (double, int, char etc.)             */
16     blksPerMPByShrdMem ← gInf.SHMEM_SIZE / shMemPerBlk
17     show error if blksPerMPByShrdMem ≤ 0
18     blksPerMP ← min(blksPerMPByReg, blksPerMPByShrdMem)
19
       /* increase tiling as much as possible considering shared memory */
20     shMemPerTile ← cInf.blockSizeX ×
21         (cInf.blockSizeY + cInf.filterSizeY - 1) × sizeof(type)
22     remShMem ← gInf.SHMEM_SIZE - shMemPerBlk × blksPerMP
23     if remShMem > 0 then
24         maxTilByShMem ← 1 + (remShMem / (shMemPerTile × blksPerMP))
25         nTiles ← min(maxTilByShMem, maxTiling)
26
           /* finally, adjust tiling considering # of registers         */
27         while nTiles > 1 do
28             remNRegPerMP ← gInf.NREGS_MP - gInf.WARP_SIZE ×
29                 nWarpsPerBlk × blksPerMP × cInf.perTrdNRegFor[nTiles]
30             if remNRegPerMP ≥ 0 then /* has sufficient registers    */
31                 return nTiles
32             nTiles ← nTiles - 1
33     return nTiles
```

maximizing the computational resource utilization of a multiprocessor[11]. In short, when considering application performance in CUDA, one can either optimize for maximizing computational resource utilization (i.e., maximize occupancy) as the primary objective function, or maximize storage resource utilization (i.e., maximize the tiling factor).

Tiling metrics

We define the following two metrics to dynamically calculate tiling factors considering the two maximization functions:

- In the first metric ($\Phi_{occupancy}$), maximizing occupancy is defined as the primary objective while tiling is maximized as a secondary objective. The objective function first tries to achieve maximum occupancy (possibly 100%) while keeping the tiling factor to 1 and later chooses to increase the tiling factor to the maximum level possible without decreasing the already determined occupancy level.

- In the second metric (Φ_{tiling}), we do the other way around by maximizing tiling as the primary objective while keeping occupancy to the minimum (i.e. assuming only one thread-block per multiprocessor). The occupancy is considered in case tiling cannot be increased any further (e.g., in our case, we use at most 1×16 tiling)[12].

Algorithm 1 shows how the tiling factor is determined when maximizing for occupancy ($\Phi_{occupancy}$). First, it strives to fit as many (thread) blocks as possible inside a single CUDA multiprocessor to maximize occupancy (line 6-7). There exist several constraints, both from CUDA resources (e.g., maximum number of threads/warps/blocks, see Table 3.3 for their description) and from the computation (e.g.,

[11]The warp occupancy, as defined by NVIDIA [65], is the ratio of active warps per multiprocessor to the maximum number of active warps supported for a multiprocessor on a GPU.

[12]The limit on the tiling factor is set to allow the secondary objective (i.e. maximizing occupancy) to be considered besides maximizing tiling only.

Parameter	Description
NTRDS_MP	Number of threads per multiprocessor
NBLKS_MP	Number of (thread) blocks per multiprocessor
NREGS_MP	Number of registers per multiprocessor
NWARPS_MP	Number of warps per multiprocessor
SHMEM_SIZE	Shared memory size (in bytes) per multiprocessor
WARP_SIZE	Number of threads in a warp

Table 3.3: Description of different CUDA architecture specific parameters used inside the algorithms.

number of threads inside a block). Information about the number of registers used by a CUDA thread (`perTrdNRegFor`) inside a kernel can be determined by passing `-Xptxas -v` options to the CUDA compiler (`nvcc`). Afterwards, it adjusts this number by comparing, the total number of registers (line 9-12) and total amount of shared memory (line 14-18) available inside a streaming multiprocessor, to demands from thread blocks for each of those storage resources. After finding the maximum blocks that will fit inside a multiprocessor, it tries to increase tiling for each block by considering the remaining shared memory and registers inside a multiprocessor.

When maximizing for tiling (Φ_{tiling}), as shown in Algorithm 2, we try to increase the tiling factor as much as possible (up to the maximum tiling factor) by considering only one block inside a multiprocessor. This gives us the maximum potential for tiling as at least a block should fit inside a multiprocessor for correct execution. The tiling factor is determined by calculating the shared memory spared (i.e. not used by the block with tiling one), and how many tiles can fit in that amount of shared memory. Then, this tiling factor is adjusted considering the amount of extra registers that will be required to the available number of registers. The tiling size is decreased until it fits the number of registers available in the multiprocessor.

Algorithms for calculating both tiling metrics are implemented in the MapOverlap2D CUDA implementation. The input for both algo-

Algorithm 2: getTilingForMaxTiling

Input: $cInf$, information about computation; $gInf$, information about GPU; $maxTiling$, maximum tiling factor
output: $nTiles$, the tiling factor

```
1  begin
2      nTiles ← 1
3      nTrdsPerBlk ← cInf.blockSizeX × cInf.blockSizeY
4      nWarpsPerBlk ← ⌈nTrdsPerBlk / gInf.WARP_SIZE⌉
5
       /* compute used and remaining shared memory per multiprocessor   */
6      shMemPerBlk ← (cInf.blockSizeX + cInf.filterSizeX - 1) ×
7          (cInf.blockSizeY + cInf.filterSizeY - 1) × sizeof(type)
           /* type is element type (double, int, char etc.)             */
8      remShMem ← gInf.SHMEM_SIZE - shMemPerBlk
9      show error if remShMem ≤ 0
10
       /* get max tiling possible considering shared memory             */
11     shMemPerTile ← cInf.blockSizeX ×
12         (cInf.blockSizeY + cInf.filterSizeY - 1) × sizeof(type)
13     maxTilByShMem ← 1 + remShMem / shMemPerTile
14     nTiles ← min(maxTilByShMem, maxTiling)
15
       /* now adjust tiling considering # of registers                  */
16     while nTiles > 1 do
17         remNRegPerMP ← gInf.NREGS_MP - nWarpsPerBlk
18             × gInf.WARP_SIZE × cInf.perTrdNRegFor[nTiles]
19         if remNRegPerMP ≥ 0 then /* has sufficient registers          */
20             return nTiles
21         nTiles ← nTiles −1
22     return nTiles
```

rithms is the computation's and CUDA architecture parameters. The former are inferred by the program while the latter are determined (via NVIDIA CUDA API) based on the compute capabilities of the CUDA architecture. As shown later with experiments, $\Phi_{occupancy}$ often results in small tiling factors but greater occupancy while Φ_{tiling} often results in relatively large tiling factors with very low occupancy.

3.5.3 Experiments

The experiments are conducted on three different NVIDIA GPUs with different compute capabilities, as shown in Table 3.4. For experiments on C2050, an input image of size 4096×4096 is used with 13 filter dimensions ranging from 3 up to 27 (strided by two). For GTX 280

GPU type	**C2050**	**GTX280**	**8800 GT**
Compute capability	2.0	1.3	1.1
Number of multiprocessors (MP)	14	30	14
Number of cores in a MP	32	8	8
Processor Clock (GHz)	1.15	1.2	1.5
Local memory (KB)	48	16	16
Cache support	yes	no	no
Memory bandwidth (GB/sec)	144	141.7	57.6
Memory interface	384-bit	512-bit	256-bit

Table 3.4: Experiment setup.

and 8800 GT experiments, an input image of 2048×2048 is used with filter dimensions ranging from 3 up to 25 (strided by two).

The C2050 was the development platform while the GTX280 and 8800 GT are used to show performance portability and the effect of L1 and L2 caches. The overhead of calculating the tiling factors for a given objective function is also considered as part of the execution time, which proves to be negligible. The following implementations are referred to in the evaluation:

- *naive* implementation: The very simple CUDA implementation without any explicit optimization, i.e. each CUDA thread processes one output element by loading all input data from the GPU global memory.

- *optimized* implementation: The *naive* implementation with constant memory and shared memory optimizations.

- *tiling-optimized* implementation: The *optimized* implementation with tiling. The tiling factor could be based upon either $\Phi_{occupancy}$ or Φ_{tiling}.

2D map projection is used to present 3D results in Figure 3.10, 3.11 and 3.12. Besides scales on x- and y-axis, please consider the

differences in the color-scale in each (sub-)figure for the correct interpretation of results.

Usage of shared and constant memory

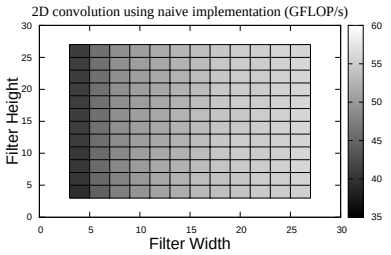
As mentioned earlier, the effect of applying shared memory and constant memory optimizations is largely influenced by the caching capabilities of a GPU. Figure 3.10 shows performance improvements over different GPU architectures for the *optimized* implementation over the *naive* implementation. On the cache-based C2050, performance improvements are, on average, by a factor of almost 1.5. However, on the GTX280 GPU which has no cache, the performance differs by a factor of 3.4. On the 8800 GT, the performance improvement is by a factor of 25 which is much higher than for the GTX280. This is because of a substantial difference in the memory bandwidths of 8800 GT and GTX280 (see Table 3.4) which has a big performance impact for global memory accesses, which are done frequently in the naive implementation.

	$\Phi_{occupancy}$		Φ_{tiling}	
	Tiling factor	Occupancy	Tiling factor	Occupancy
C2050	3.83	100%	14.33	33.34%
GTX280	1.63	75%	3.83	25%
8800 GT	7.35	33.34%	7.35	33.34%

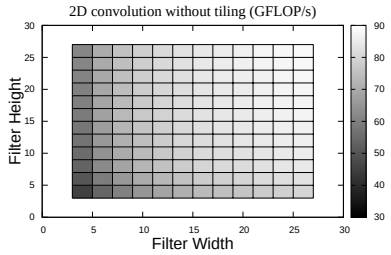
Table 3.5: Average tiling factor and occupancy achieved with 2 metrics on different GPUs.

Tradeoff between $\Phi_{occupancy}$ and Φ_{tiling}

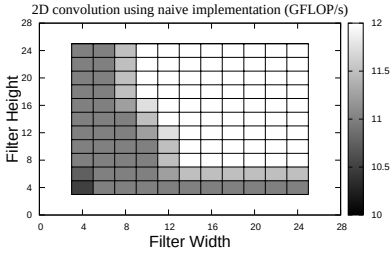
Table 3.5 highlights the tradeoff between the two metrics on different GPUs. For C2050, when maximizing occupancy ($\Phi_{occupancy}$), the tiling factor is reduced by a factor of 3.74 to gain the last 67% in occupancy. Similarly, for GTX280, the occupancy was much less when optimizing for tiling (Φ_{tiling}) in comparison to when optimizing for



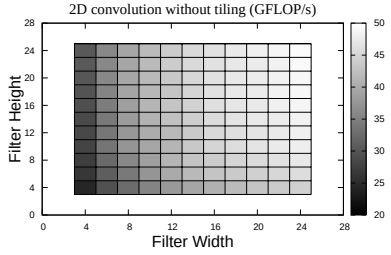
(a) C2050 (50 GFLOP/s)



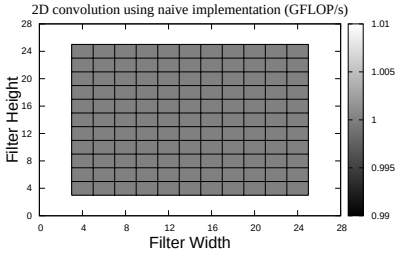
(b) C2050 (76 GFLOP/s)



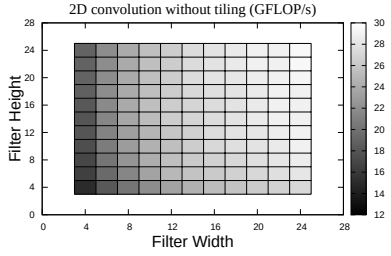
(c) GTX280 (12 GFLOP/s)



(d) GTX280 (41 GFLOP/s)

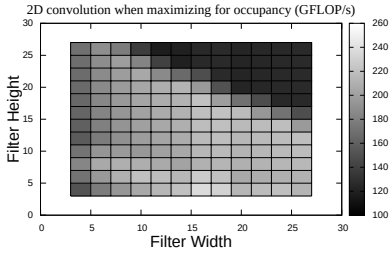


(e) 8800 GT (1 GFLOP/s)

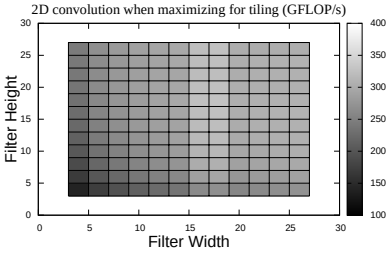


(f) 8800 GT (25 GFLOP/s)

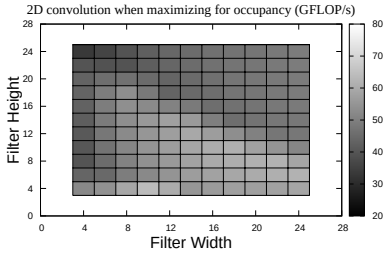
Figure 3.10: 2D convolution with *naive* (a,c,e) and *optimized* (b,d,f) implementations over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.



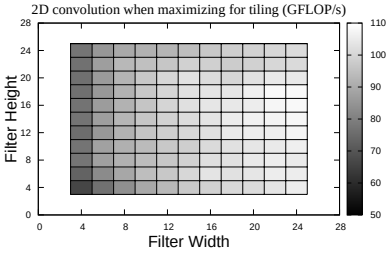
(a) C2050 (180 GFLOP/s)



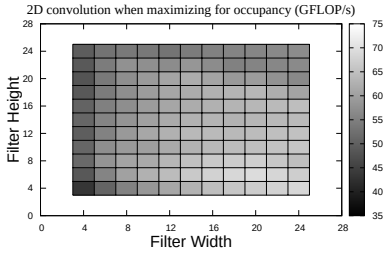
(b) C2050 (274 GFLOP/s)



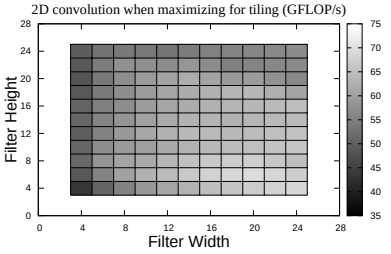
(c) GTX280 (50 GFLOP/s)



(d) GTX280 (95 GFLOP/s)



(e) 8800 GT (59 GFLOP/s)



(f) 8800 GT (59 GFLOP/s)

Figure 3.11: 2D convolution when tiling factors are chosen for maximizing either occupancy ($\Phi_{occupancy}$, a,c,e) or tiling (Φ_{tiling}, b,d,f) over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.

occupancy. However, for 8800 GT, the two metrics do not yield any difference. This is because of constraints in maximizing occupancy ($\Phi_{occupancy}$) any further than what is assumed initially in Φ_{tiling} (i.e. 1 thread block per multi-processor).

As we can see in Table 3.5, 100% occupancy is not always achieved even when optimizing for occupancy. This is possible for multiple reasons. For example, it could happen due to constraints on storage resources (shared memory and registers) that do not allow the maximum number of warps to run inside a multiprocessor. Similarly, the number of warps in a (thread) block may not be a perfect multiple of the maximum number of warps that can possibly fit inside a multiprocessor. For more information about how it works, see Algorithm 1.

The maximization goal may have a profound impact on the tiling factors chosen and consequently on the achieved performance. As shown in Figure 3.11, the overall performance is higher when maximizing for tiling (Φ_{tiling}) than when it is optimized for occupancy ($\Phi_{occupancy}$). Performance with Φ_{tiling} is, on average, 90% and 52% higher than the average performance achieved with $\Phi_{occupancy}$ on GTX-280 and C2050 GPUs. On 8800 GT, there is no difference as both objective functions determine the same tiling factor, as shown in Table 3.5. In short, the higher the tiling factor has been, the better the performance has been achieved. One possible explanation for this could be that with high tiling factors, less redundant neighborhood loads have been carried out from the global memory to shared memory which reduces the overall communication time, as explained earlier in Section 3.5.2.

Performance portability

Table 3.6 shows the CUDA architecture specific parameters for the three GPUs. These parameter values can easily be obtained by querying the device capabilities using CUDA API or by NVIDIA's occupancy calculator excelsheet [65]. Based on these parameters alongside the computation information, the two metrics generate the tiling factors for their corresponding maximization goal. As the tiling factors are automatically calculated and considered for execution, this gives

	C2050	**GTX280**	**8800 GT**
NTRDS_MP	1536	1024	768
NBLKS_MP	8	8	8
NREGS_MP	32768	16384	8192
NWARPS_MP	48	32	24
SHMEM_SIZE	48800	16384	16384
WARP_SIZE	32	32	32

Table 3.6: CUDA architecture specific parameters.

us performance portability when moving from one GPU architecture to another without requiring manual changes in the implementation.

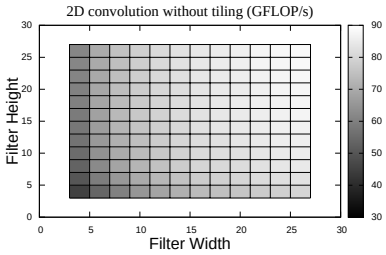
To illustrate performance portability, we compare performance of our `solution` with a `baseline` implementation for a given GPU architecture. For the `solution`, we have used the *tiling-optimized* implementation with Φ_{tiling} as its maximization metric. For the `baseline` implementation, we have considered two alternatives:

1. To use a platform-specific optimized implementation for a given GPU architecture.

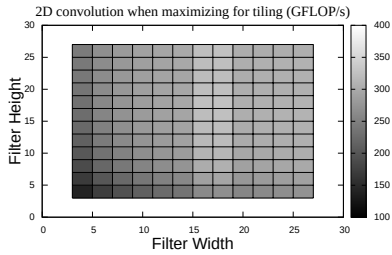
2. To use a generic, fairly optimized implementation that can be executed on different GPU architectures without requiring rewriting the code.

We opted for the second alternative as the first one would require lot of extra effort for writing optimized implementations for each of the three GPUs that we have considered. Following the second alternative, we have chosen our *optimized* implementation as the `baseline` implementation for comparison. The choice is justified as the *optimized* implementation provides significant speedups over *naive* implementation for different class of GPU architectures (see Figure 3.10) and it is also fairly generic as it can run on any modern GPU with a shared memory support.

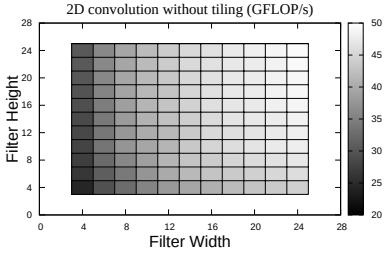
We define *relative performance* on a platform as ratio between the average performance of `solution` and the `baseline` implementa-



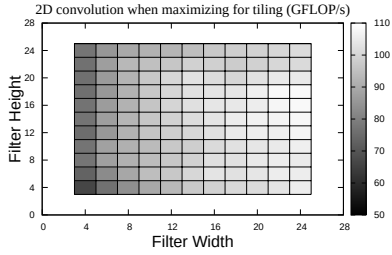
(a) C2050 (76 GFLOP/s)



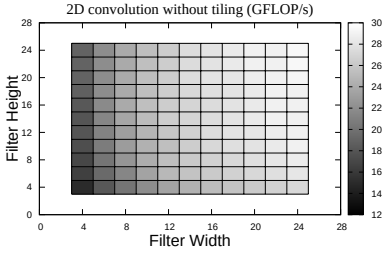
(b) C2050 (274 GFLOP/s)



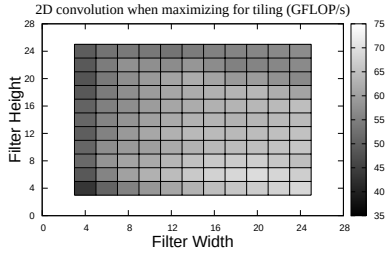
(c) GTX280 (41 GFLOP/s)



(d) GTX280 (95 GFLOP/s)



(e) 8800 GT (25 GFLOP/s)



(f) 8800 GT (59 GFLOP/s)

Figure 3.12: Performance (GFLOP/s) for 2D convolution with *optimized implementation* (a,c,e) and *tiling-optimized implementation* (Φ_{tiling}, b,d,f) over different NVIDIA GPUs. Average GFLOP/s are mentioned in the caption of each sub-figure.

tion. By measuring this ratio, we consider our solution as performance portable if it can retain the relative performance to a potentially higher level (at least `>1`, i.e., better than the baseline implementation for every invocation).

Figure 3.12 compares the performance of 2D convolution with `solution` over `baseline` implementation. The relative performance is `3.6`, `2.3`, and `2.4` on average for C2050, GTX280 and 8800 GT GPUs respectively which is much higher than our threshold value i.e., `1`. This shows that we can achieve good MapOverlap2D performance on different CUDA GPUs in a portable manner by dynamically calculating the tiling factors on that architecture. The same technique can be applied to the MapOverlap2D OpenCL implementation especially when executing on modern GPUs.

3.6 Summary

In this chapter, we presented our work on the design and implementation of a skeleton programming framework named *SkePU* for efficient and portable programming of GPU-based systems. We discussed its design and main features as well as the tuning mechanism that can automatically choose the best implementation for a given skeleton execution in a portable manner. The library also implements support for hybrid execution using the StarPU runtime system that provides dynamic scheduling and selection. Evaluation with several applications showed that, for skeleton expressed computations, our library can provide a high-level abstraction while providing good performance by doing intelligent implementation selection by offline tuning and/or hybrid execution using the runtime system.

In the last section, we have presented a case-study of skeleton-specific parametric auto-tuning, namely on optimizing the CUDA implementation of the MapOverlap2D skeleton in SkePU by applying the $(1 \times N)$ tiling optimization. The case study shows that we can achieve good performance for MapOverlap2D computations on NVIDIA GPUs in a portable manner, by determining the tiling factor at runtime considering architecture and problem-specific parameters.

Chapter 4

Containers and memory management

In this chapter, we present the memory management functionality implemented inside the SkePU containers which enables several communication and memory optimizations. Specifically, we discuss the initial memory management mechanism, its limitations and the improvements that we have made to it as part of this thesis work. When SkePU skeleton calls are executed with the StarPU runtime system, the actual memory management is done by StarPU. SkePU containers, when using StarPU, enable asynchronous execution of skeleton calls in a transparent way while simulating sequential program execution semantics. This and the execution model of SkePU programs is also discussed in this chapter.

The chapter is structured as follows: it starts with a brief introduction in Section 4.1, followed by presentation and evaluation of the memory management implemented inside SkePU containers in Section 4.2 and 4.3 respectively. Section 4.4 discusses how containers work when the SkePU library is used with the StarPU runtime system. We conclude the chapter with a discussion of the execution model for SkePU programs in Section 4.5, followed by a summary in Section 4.6.

4.1 Introduction

In GPU-based systems that we target, GPU compute devices have separate (physical) memories. Communication between these device memories and the CPU main memory is carried out on the (relatively slow) PCIe interconnection network which becomes a performance bottleneck in many applications. Reducing the communication between different device memories and main memory is desirable for performance reasons. One way to reduce data transfers is not to transfer output data of GPU computations back to main memory until and unless it is accessed there. This could be advantageous if, in the meanwhile, more GPU computations on that data can be performed. Obviously, there could be more complex scenarios where partial data is available in different device memories, as we will discuss later in this section.

The SkePU framework optimizes the operand data transfers for its skeleton calls between different device memories and the main memory. Both vector and matrix containers do memory management by internally keeping track of different copies of their data in different memories. If a GPU computation is done, modifying copies of elements of a container in the GPU memory, these are not immediately transferred back to the host memory. Instead, the container delays the transfer until an element is accessed on the host side (for example through the [] operator for `Vector`). This lazy memory copying is of great use if several skeletons are called one after the other, with no modifications of the container operand data by the host in between. In that case, the payload data of the container is kept on the device (GPU) through all the computations, which significantly improves performance.

The memory management functionality yields several other benefits also. Both containers provide lookup functions to find out the state of their data in different memories. This information is useful in deciding which skeleton implementation to invoke. For example, if operand data of a skeleton call is already present in a device memory, using the CUDA skeleton implementation on that GPU device might be more efficient. Similarly, it could help in optimizing com-

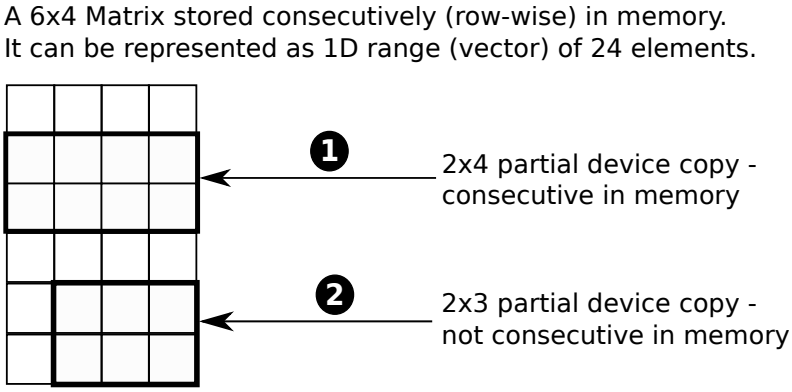


Figure 4.1: Partial copies of a matrix and their mapping to memory locations in the original matrix.

munication patterns between different device memories. As we will discuss in detail, the containers can internally manage data transfers between different device memories without involving the main memory if the underlying GPU platform supports it. Above all, all this can be achieved in a transparent manner behind a generic interface similar to C++ STL containers.

4.2 Memory management in SkePU

In this section, we discuss the memory management that is implemented inside the SkePU containers. We discuss the initial memory management mechanism and its limitations with respect to data consistency and performance issues. Afterwards, we discuss the new, improved mechanism that we have developed as part of this thesis work.

The idea of having multiple (also, partial) copies for a single container object is common in both (memory management) mechanisms described later. However, in the initial mechanism, copies of an object are represented as a 1D (one-dimensional) range for both SkePU vector and matrix containers. Although the contents of a SkePU matrix is stored consecutively (row-wise) and can be represented as a 1D

range, partial (device) copies of a matrix object may not always map to consecutive 1D (sub-)ranges in the matrix. This is shown in Figure 4.1 with two partial device copies representing two sub-matrices. The first partial device copy maps to elements that are stored consecutively in the original matrix and thus can be represented as 1D range. Partial copies that do not map to consecutive elements (such as device copy 2 in Figure 4.1) are not possible in the initial mechanism. In the new mechanism, we have solved this problem by defining a separate mapping for matrix operands which allows partial copies that do not map to consecutive elements in the original matrix. However, in the following, we use the 1D range mapping for explaining the working of both mechanisms. The 1D mapping is used in the initial mechanism for both vector and matrix containers, and for vector container only in the new mechanism. Later, we describe how the new mechanism generalizes this mapping to represent partial copies with non-consecutive elements for matrix containers.

For explanation, suppose that we have a SkePU container object O that could be a vector or matrix object. One main-copy of object O, called *mainCopy(O)*, is created in the main memory when the object is allocated (i.e., when a constructor for object O is called in C++). This is the only copy in main memory created for that container object and all accesses in CPU code (including C++/OpenMP skeleton implementations and other program accesses) are directed to this copy. It covers the complete contents of object O (i.e., *mainCopy(O)* has elements with *index range* $[0, N)$[1] (first inclusive) if the size of object O is N where $N \geq 1$) and remains allocated as long as the object is alive (i.e., until the object destructor is called in C++). Whenever a container object is resized, i.e., N is changed, all other copies on device memories are deleted and the size of the main-copy is adjusted accordingly.

The communication bandwidth from device to host (DTH) and from host to device (HTD) (roughly) doubles if the main-copy of a container object in main memory is allocated as page-locked (pinned)

[1] We use the standard set notation for representing a set of consecutive indices as index range. In set notation, $[a, b)$ is the half-open interval $\{x | a \leq x < b$ and x is a positive integer, i.e., $x \in \mathbb{Z}_{\geq 0}\}$ [18].

memory. As page-locked memory is always stored in physical CPU memory (i.e., RAM) and cannot be swapped out to disk, DMA on the GPU can request transfer of such memory to and from the host memory without the involvement of the CPU. This along the fact that such memory can be transferred without any need for creating intermediate copies makes it quite faster [168]. We have implemented support for using page-locked memory for both containers; with a simple flag, the support can be enabled which also helps in overlapping communication and computation considering that data transfers for page-locked memory can be made asynchronously.

During the program execution, other (partial or full) copies of an object may get created[2], used and later destroyed in different device memories depending upon the object usage with skeleton calls in a program. Let K ($K \geq 0$) denote the number of device memory copies for object O at any given time which are denoted by $dCopy_i(O)$ where $1 \leq i \leq K$. A device copy $dCopy_i(O)$ of the object O is identified by $\langle devID_i, offset_i, n_i \rangle$ where $devID_i$ is an identifier of the device on which the $dCopy_i(O)$ is present, $offset_i$ is the index of the first element of $dCopy_i(O)$ with respect to the index range of *mainCopy(O)*[3], and n_i is the number of elements stored in the copy; i.e., $dCopy_i(O)$ covers O elements with the index range $[offset_i, (offset_i + n_i))$ on device $devID_i$. Furthermore, each device copy $dCopy_i(O)$ has a *modified* flag which is set when contents in that copy is modified (e.g., by a GPU skeleton implementation). The modified contents of this copy is copied to the main-copy (and possibly other device copies) as described later.

Partial copies of a container object can exist in device memories for two reasons. First, an application may have skeleton operations on parts of a container object. Both vector and matrix containers support iterators [11] which allows to call an operation on a subset of

[2]Creation of a device copy involves, besides other things, memory allocation on the device for that copy and either copying of contents from the main-copy (in case it is read) or writing of new contents to it. This means that there could *not* exist a device copy for an object without memory allocation or without contents initialized once (either copied or written).

[3]The *offset* would be 5 if $dCopy_i(O)$ starts from the 6th element of *mainCopy(O)*, i.e., the indexing is zero-based as per C++ style.

the elements as illustrated below:

```
skepu::Vector<int> v0(10); // create vector (v0) of 10 elements
...
skel(v0); // readwrite all 10 elements of v0
skel(v0.begin()+5, vo.end()); // readwrite last 5 elements of v0
skel(v0.begin()+3, v0.end()-3);// readwrite elements [3-7) of v0
```

If we execute the three skeleton calls (listed in the above code) on a single GPU, three device copies of `v0`, with index ranges $[0, 10)$, $[5, 10)$ and $[3, 7)$ respectively, would get created[4]. Secondly, partial copies are created if container objects are passed as arguments to a skeleton call that is executed on multiple GPU devices in parallel. Dividing the computation work of data-parallel skeletons across different GPU devices divides the container objects, passed as operands, into chunks.

SkePU containers are primarily designed for passing operand data to skeleton calls and are not designed to be thread-safe. This means that concurrent memory requests to overlapping elements in a container object are not handled in the memory management mechanisms. We will discuss it in more detail with the SkePU program execution model, in Section 4.5.

The concept of having one main-copy and potentially many overlapping device copies is the same in both the initial and the new memory management mechanism. However, the main difference comes with how read and write accesses to these copies are handled, as described below.

4.2.1 Initial mechanism

In the initial memory management mechanism [89], all accesses to object O in CPU code are mapped to *mainCopy(O)* and are handled in the following manner: Before any element of *mainCopy(O)* is read in main memory, all modified device copies (i.e., device copies with

[4] As a device copy ($dCopy_i$) is identified by a tuple $\langle devID_i, offset_i, n_i \rangle$, two or more device copies of an object O with different starting elements (*offset*) and/or number of elements (n) are considered different copies even on the same device.

modified flag set) of the object are copied back to *mainCopy(O)*, i.e.,

$$\text{Copy } dCopy_i(O) \rightarrow mainCopy(O) \,\forall\, i \text{ where } 1 \leq i \leq K \\ \wedge dCopy_i(O).modified = \text{true} \tag{4.1}$$

Copying contents of a modified device copy to main-copy clears the *modified* flag of that device copy. Similarly, before any element of *mainCopy(O)* is written in main memory, all modified device copies of the object are first copied back to *mainCopy(O)* (Equation 4.1) and then all device copies of the object are removed, i.e., only main-copy remains.

$$\text{Remove } dCopy_i(O) \forall\, i \text{ where } 1 \leq i \leq K \tag{4.2}$$

Removing a copy releases memory allocations for that copy in the device memory as well as deletes it from the copies list, i.e., the device copies count (K) is decremented. Furthermore, data from all modified device copies is copied back no matter which elements in *mainCopy(O)* are read[5]. Obviously, one can think about more fine-grained control where only a chunk of data is updated instead of everything; we have implemented this as improvement to the existing mechanism (more on this in the next section).

Copies on device memories are created, if not present already, when a container is used as operand with a skeleton call that is executed on a GPU with a CUDA or OpenCL skeleton implementation. A GPU read and write access for an object is identified by $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ and is handled in the following manner. When a GPU read access to object O comes with certain $\langle devID_{acc}$, $offset_{acc}, n_{acc} \rangle$ information, a lookup is made to find an existing copy ($dCopy_j(O)$) with same index range on the given device:

$$find\ dCopy_j(O) \text{ where } 1 \leq j \leq K \\ \wedge devID_{acc} = devID_j \wedge offset_{acc} = offset_j \wedge n_{acc} = n_j \tag{4.3}$$

If an existing copy $dCopy_j(O)$ is found, it is used; this leads to con-

[5]For some C++ STL-like container operations (`clear`, `assign` etc.), no modified contents from device copies is copied back to the main-copy as the contents in the container object is discarded anyway.

	CPU (main memory)	GPU (device memory)
Copies	One *main-copy*	Zero or more *device-copies (dCopies)* A device-copy (*dCopy*) identified by <devID, *offset, n*>

Read access	Copy all modified device copies back to *main-copy* and clear their *modified* flags
	Read *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		Copy all modified copies back to *main-copy* and clear their *modified* flags
		Copy [offset, offset+n) from *main-copy* into *dCopy*
		end if
		Read the *dCopy*

Write access	Copy all modified device copies back to *main-copy* and clear their *modified* flags
	Remove all device copies
	Write *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		Set *modified* flag for the *dCopy*
		Write to the *dCopy*

Read/ Write access	Copy all modified device copies back to *main-copy* and clear their *modified* flags
	Remove all device copies
	Read/Write *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		Copy all modified copies back to *main-copy* and clear their *modified* flags
		Copy [offset, offset+n) from *main-copy* into *dCopy*
		end if
		Set *modified* flag for the *dCopy*
		Read/Write the *dCopy*

Figure 4.2: The initial memory management mechanism for a single container object.

sistency problems if the contents of that copy is not the most recent one, as we will discuss later. However, if no existing copy is found, multiple operations are carried out: 1) space for a new device copy $dCopy_{acc}(O)$ is allocated, 2) contents from all modified device copies is copied back to *mainCopy(O)* (as shown in Equation 4.1) and their *modified* flag is cleared, and 3) required contents from *mainCopy(O)* is then copied to $dCopy_{acc}(O)$ in the following manner:

$$\text{Copy } mainCopy(O)[offset_{acc}, (offset_{acc}+n_{acc})) \rightarrow dCopy_{acc}(O) \quad (4.4)$$

The newly created copy $dCopy_{acc}(O)$ is then inserted into the existing device copies collection and the device copies count (K) is incremented. Similar operations are carried out for a GPU write access to object O except that no data is copied (in any way) when a new device copy is allocated. Moreover, the *modified* flag for the copy written (no matter if newly created or an existing one was found) is set.

Figure 4.2 summarizes the initial memory management mechanism in SkePU. When creating a new device copy, the *modified* flag is cleared by default. It is set for a device copy whenever contents in that copy is changed. Note that we track content modifications for a complete device copy of a given size. This is because the elements in a device copy exactly map to the elements that are requested for a GPU read/write access.

4.2.2 Limitations of the initial mechanism

There exist both performance and data consistency issues with this mechanism. When a device copy of a container is modified, other (possibly overlapping) device copies of the same container on the same and other devices are neither marked invalid nor are they removed. A future access to those device copies would access stale data and thus result in data consistency issues. This could happen in both single and multi-GPU based systems. In the following, we discuss two example scenarios where overlapping copies of data in the same or different device memories result in data consistency and performance issues[6].

Scenario one

In the first scenario, we consider three skeleton calls, all executed on a single GPU while operating on different (overlapping) parts of a container object.

```
skepu::Vector<int> v0(10);
map(v0, ...); // read all 10 elements of v0
map(..., v0.begin()+5); // write last 5 elements of v0
map(v0, ...); // read all 10 elements of v0
```

For the first skeleton call, space for a device copy of vector `v0` gets allocated for 10 elements on GPU and data is copied from the main-copy to this new device copy. Later, space for another device copy gets allocated for the second skeleton call, mapping to the second half of the `v0` vector object. As the copy is written, no copying of contents takes place; rather, the *modified* flag for this newly written device

[6]There exists several other possible scenarios where data consistency and performance issues can arise especially when considering multiple GPUs.

copy is set. At this time, two device copies of `v0` exist, one covering full contents of `v0` and one covering the second half of `v0` but with newly written contents. Now, for the third skeleton call executing on the same GPU, an existing device copy (created for the first skeleton call) is found and used. However, the contents of this device copy is stale for the second half which was over-written in the device copy created in the second skeleton call.

The same consistency issues arise even when executing the above code scenario on a multi-GPU system where the first and the third skeleton call are executed on the same GPU. Correct working of the above source code in a GPU-based system would require explicit synchronization by the programmer between the second and third skeleton call, as shown below:

```
skepu::Vector<int> v0(10);
map(v0, ...); // read all 10 elements of v0
map(..., v0.begin()+5); // write last 5 elements of v0
v0.flush(); // update main-copy, discard all device copies
map(v0, ...); // read all 10 elements of v0
```

However, this is inefficient as it would discard all device memory copies of vector `v0` and would require re-allocation and copying of data for the third skeleton call. Moreover, expecting the application programmer to find execution scenarios where data consistency issues might arise in his/her skeleton program is undesirable.

Scenario two

Recently, we have ported NBody, LU factorization and Smooth Particle Hydrodynamics (SPH) applications to use the SkePU skeletons. In all these applications, the main computation is modeled by MapArray skeletons operating on different input and output container objects as shown in the pseudo-code in Listing 4.1.

When executing the computational loop (shown in Listing 4.1) on a single GPU-based system, we do not need any explicit synchronization as there exist no partial device-copies of data, i.e., only one device copy for each entire vector object is created in the GPU device memory. However, if we execute the same computation on a system containing two (or more) GPUs, we get data consistency issues.

```
1  skepu::Vector<...> v0(10);
2  skepu::Vector<...> v1(10);
3  ...
4  loop(...)
5  {
6    maparray1(v0, v1, v1); // read v0 - readwrite v1
7    ...
8    maparray2(v1, v0, v0); // read v1 - readwrite v0
9  }
10 ...
```

Listing 4.1: Pseudo-code with two MapArray skeleton calls in a loop.

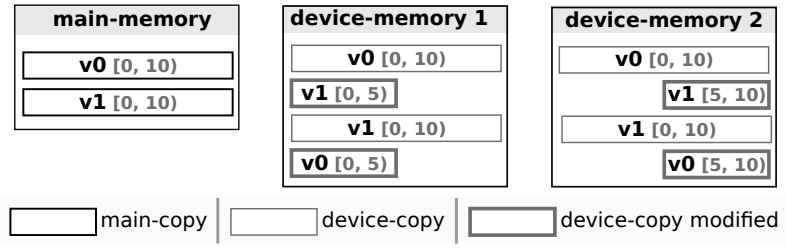


Figure 4.3: Main and device memory copies of two vector objects after execution of the first loop iteration, from Listing 4.1.

Suppose that we execute the code shown in Listing 4.1 on two GPUs of the same generation. Figure 4.3 shows the memory state of both device memories as well as of main memory after execution of the first loop iteration. Initially, the main-copies of `v0` and `v1` are created in main memory (Line 1 and 2 respectively in Listing 4.1). When executing the first skeleton call (Line 6) on two GPUs, the work is divided by partitioning the second input and the output vector (`v1`) between the two device memories; the first input vector (`v0`) is replicated on both GPUs for reading purpose, as per the MapArray skeleton semantics. Similar things happen for the second skeleton call (Line 8) except that `v1` is replicated and `v0` is partitioned across the two device memories.

After the first loop iteration, two copies of each vector object (one partial, one full) are present in each device memory with the *modified*

flag of all partial copies set. The consistency problem arises with the second loop iteration as stale (full device) copies of `v0` and `v1` are read in the first and the second skeleton call respectively. This is because the updated data from partial device copies is never copied to these full device copies of the same vectors. The obvious fix to this scenario would be to `flush` both vectors after each loop iteration, but this is too inefficient and an overkill as it copies all modified contents back to memory as well as removes all device memory copies. Having it inside a loop means that this communication overhead would be incurred for each loop iteration. Moreover, the problem is too specific as it happens only when executing the skeleton calls on multiple GPUs and `flush`ing vectors becomes unnecessary in other execution scenarios.

Remarks

Clearly, the initial memory management mechanism implemented inside the SkePU containers has memory consistency issues with execution on single and multi-GPU systems. Although it is possible to ensure program correctness, the fixes are normally an overkill and can hurt performance by increasing the communication volume. Furthermore, whenever contents of a container object in the main-copy is modified[7], all device copies are removed (i.e. device memory allocations are freed), as in the following example:

```
// v0 with multiple device copies...
...
v0[5] = 3; // removes all device copies
map(v0); // allocate device copy again if executed on a GPU...
```

Any further usage of the container object for GPU execution would require memory re-allocation on that GPU device memory. Marking device copies invalid instead of removing them could be a better idea considering that the device memory allocation is a blocking operation and can prevent asynchronous GPU execution [168].

Multiple device accesses with different offset and/or size information results in different device copies even in the same device memory.

[7] We use proxy elements (from [157]) for some operations to be able to distinguish between read and write accesses in the host code.

This could potentially lead to a situation where the device memory is exhausted and no space in device memory remains available for future allocations. The initial mechanism does not handle this situation explicitly. Last but not the least, contents of a device memory is always copied from the main-copy that exists in main memory. One might think about scenarios where (part of) the contents can be copied from other existing copies of the same object in the same device memory. Next, we present our work on improving the memory management mechanism implemented in SkePU containers to tackle these data consistency and performance issues in an implicit and efficient manner.

4.2.3 New mechanism

We replaced the initial mechanism in order to provide a more robust and efficient memory model for multi-GPU skeleton execution. The basic concept of having one main-copy and possibly multiple device memory copies (identified by $\langle devID, offset, n \rangle$[8]) of each object is the same. However, we introduce a more robust mechanism for updating the state of the different copies when reading and writing contents in a copy. A *valid* flag for each copy (including the main-copy) is added which specifies whether the contents of a given copy is valid for reading purpose or not. If the *valid* flag of a copy is not set, a read operation on that copy would first require copying updated data from either other device memories with valid contents or from main memory, as described later.

In order to control the potential problem of running out of device memory space by creating too many device copies, each device copy has a *lastUsed* flag which tracks the time when that device copy was last accessed (read and/or written). When not enough space is available for allocating a new device copy, one or more least recently used (LRU) device copies can be deallocated to make space for a new device copy.

[8]For simplicity of presentation, we consider here the vector case. Device copies of matrix objects are identified with a different tuple in the new mechanism, as we will see later.

All accesses to object O in CPU code are mapped to its main-copy (*mainCopy(O)*). Initially, only the main-copy for an object is created, which is marked valid. Over time, multiple device copies of object O may get created with contents overlapping with the main-copy, considering that the main-copy covers the full contents. For each device copy of an object, we also have a *modified* flag which marks whether the contents in that copy is modified and not yet copied back to main-copy. Whenever a modified device copy is copied back to main-copy, its *modified* flag is cleared. At a given time, if there exists at least one device copy with modified contents (i.e., with *modified* flag set), the main-copy is marked invalid and any read to the main-copy may require getting data back from one or more modified device copies, as explained below.

In the new mechanism, we identify a read and write access on CPU also by an index range. By default, the index range for an object covers all of its elements (i.e., $[0, N)$ for an object of size N); however, it could be a smaller range, e.g., when a skeleton call executes on CPU with subset of elements such as:

```
skepu::Vector<int> v0(10);
map(v0.begin()+5, v0.end()); // index range [5, 10)
```

or when a specific element of object is accessed/modified on CPU:

```
v0[3] = ...; // index range [3, 4)
```

In the new mechanism, most operations are carried out considering whether the index ranges marked by two object copies (or accesses) overlap. The overlap between two index ranges, $[a, b)$ and $[c, d)$, can be determined by taking the intersection ($\cap$) of both ranges, using the standard set notation [18], i.e.:

$$\text{overlap if } [a, b) \cap [c, d) \neq \emptyset\text{; no overlap otherwise.}$$

Read and write accesses to the main-copy of an object with a given index range are handled in the following manner. If the *valid* flag of the main-copy is set, any read operation to it can proceed without any further processing. Otherwise, before an index range $[P, Q)$ of *mainCopy(O)* is read in main memory, all (if any) modified *overlapping* device copies of the object O are copied back to *mainCopy(O)*,

i.e.,

$$\begin{aligned} \text{Copy } dCopy_i(O) \rightarrow mainCopy(O) \,\forall\, i \text{ where } 1 \leq i \leq K \\ \wedge dCopy_i(O).modified = \text{true} \\ \wedge [offset_i, (offset_i + n_i)) \cap [P, Q) \neq \emptyset \end{aligned} \tag{4.5}$$

The *valid* flag of the main-copy is set if all modified copies have been written back. Notice that only contents of the modified device copies that overlap with the index range of the read access are copied back to the main-copy. This means that the main-copy can still be marked invalid (after the copying) if there exist some modified device copies that do not overlap with the index range required by the current read access and are thus not copied back. In any case, the read access can continue knowing that at least the contents required for the access is up to date in the main-copy.

Similarly, before a write access of index range $[P, Q)$ to *mainCopy(O)*, all modified *overlapping* device copies of the object O, whose index range is not a subset of the index range requested in the write access, are copied back:

$$\begin{aligned} \text{Copy } dCopy_i(O) \rightarrow mainCopy(O) \,\forall\, i \text{ where } 1 \leq i \leq K \\ \wedge dCopy_i(O).modified = \text{true} \\ \wedge [offset_i, (offset_i + n_i)) \cap [P, Q) \neq \emptyset \\ \wedge [offset_i, (offset_i + n_i)) \not\subseteq [P, Q) \end{aligned} \tag{4.6}$$

For simplicity, we copy all elements in the index range marked by that device copy back to the main-copy, as a single copy operation. In some cases, it might be a good idea to copy only those modified contents from a device copy that are not overwritten in the requested write access; investigating its benefits and proper granularity is considered part of future work. After copying contents appropriately, all *overlapping* device copies of the object are marked invalid:

$$\begin{aligned} \text{Set } dCopy_i(O).valid := false \,\forall\, i \text{ where } 1 \leq i \leq K \\ \wedge [offset_i, (offset_i + n_i)) \cap [P, Q) \neq \emptyset \end{aligned} \tag{4.7}$$

Note that clearing the *valid* flag for a device copy (i.e., marking it invalid) also clears its *modified* flag if set. When writing contents in the main-copy of an object, the device copies of the object are marked invalid (i.e., their *valid* flags are cleared) but space is not deallocated. Device copies are only removed and space is deallocated when an object is either destroyed or when it is resized or when no space is available for a new device copy allocation (least recently used device copies of an object are removed in that case). This could yield significant savings in memory (de-) allocation overhead as we will discuss later.

As mentioned earlier, a copy of an object on a device memory is created, if not present already, when it is used with a CUDA or OpenCL skeleton implementation. A GPU read and write access for an object is identified by $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ and is handled in the following manner. When a GPU read access to object O occurs, a lookup is made to find an existing copy ($dCopy_j(O)$) with same index range on the given device:

$$\begin{aligned} &\textit{find } dCopy_j(O) \text{ where } 1 \leq j \leq K \\ \wedge devID_{acc} = devID_j \wedge offset_{acc} &= offset_j \wedge n_{acc} = n_j \end{aligned} \tag{4.8}$$

If an existing copy $dCopy_j(O)$ is found and its contents is valid, i.e., $dCopy_j(O).valid = \text{true}$, it is used; otherwise, if the contents of the found copy is not valid, it is copied from other copies in a similar manner as for a newly created device copy (described below). If no existing copy is found, space for a new device copy $dCopy_{acc}(O)$ is allocated; afterwards, contents from other copies are copied to $dCopy_{acc}(O)$ in the following manner.

In the new mechanism, contents to a device copy can be copied from other device copies of the object (present in the same or other device memories) as well as from the main-copy. However, when copying from other device memory copies, it might happen that those copies only have partial overlap with what is required. For example, if we need contents for a device copy in index range $[0, 10)$ and we have two valid device copies of the same object with index range $[0, 7)$ and $[5, 10)$ respectively, then we need two content copies of index sub-

Algorithm 3: UpdateADeviceCopy

Input: $dCopy_{dst}$, device copy to be updated
O, object to which $dCopy_{dst}$ belongs

```
1  begin
2      Init rangesToCheck and copyInfPerRange
3      range_dst ← [offset_dst, (offset_dst + n_dst))
4      rangesToCheck.add(range_dst)
5      while not rangesToCheck.empty() do
6          range ← rangesToCheck.pop()
7          dCopy_src ← getValidDevCopy(O, range, devID_dst)
8          if dCopy_src then
9              subrange ← getOverlapRange(dCopy_src, range)
10             copyInfPerRange.push(dCopy_src, dCopy_dst, subrange)
11             if range ≠ subrange then
12                 rem ← getRemainingRanges(range, subrange)
13                 rangesToCheck.push(rem)
14             go back to start of while loop
15         if mainCopy(O).valid then
16             copyInfPerRange.push(mainCopy(O), dCopy_dst, range)
17             go back to start of while loop
18         foreach dev ∈ AllDevices do
19             if dev ≠ devID_dst then
20                 dCopy_src ← getValidDevCopy(O, range, dev)
21                 if PeerCopyEnabled(devID_dst, dev) ∧ dCopy_src then
22                     subrange ← getOverlapRange(dCopy_src, range)
23                     copyInfPerRange.push(dCopy_src, dCopy_dst, subrange)
24                     if range ≠ subrange then
25                         rem ← getRemainingRanges(range, subrange)
26                         rangesToCheck.push(rem)
27                     go back to start of while loop
28         foreach dev ∈ AllDevices do
29             if dev ≠ devID_dst then
30                 dCopy_src ← getModifiedDevCopy(O, range, dev)
31                 if dCopy_src then
32                     subrange ← getOverlapRange(dCopy_src, range)
33                     range_src ← [offset_src, (offset_dst + n_src))
34                     copyInfPerRange.push(dCopy_src, mainCopy(O), range_src)
                       copyInfPerRange.push(mainCopy(O), dCopy_dst, subrange)
                       if range ≠ subrange then
35                         rem ← getRemainingRanges(range, subrange)
36                         rangesToCheck.push(rem)
37                     dCopy_src.modified ← false
38                     set mainCopy(O).valid flag if no modified copy exists
39                     go back to start of while loop
40         // copy from main-copy (last option)
41         copyInfPerRange.push(mainCopy(O), dCopy_dst, range)
42     foreach copyInf ∈ copyInfPerRange do
43         // do the actual copying using e.g., cudaMemcpy etc.
44     dCopy_dst.valid ← true
```

ranges, e.g., $[0, 7)$ and $[7, 10)$, to copy all contents. Theoretically, we could require from one copy operation (in the best case) up to potentially as many as the size of the index range (n_{acc}) to which we need to copy[9]. However, in practice, the index subranges that occur in skeleton programs are very few (normally up to the total number of GPU devices) as we will see later.

The algorithm for copying data to a device copy ($dCopy_{dst}$) is outlined in Algorithm 3. When copying contents to a device copy, we build a *copy plan* for copying contents from multiple sources in the following precedence order:

1. Copy contents from overlapping valid copies in the same device memory. This is the fastest copy option as it happens within the same device memory.

2. Copy contents from the main-copy if it is valid. No further copy should be needed as the main-copy has the full contents of an object available.

3. If GPU devices support direct data transfer between their memories (e.g., using GPUDirect [193]), copy contents from valid copies in other device memories to the current copy using this feature.

4. Copy contents from modified device copies that exist in other device memories back to the main-copy and then copy required contents from the main-copy.

The *getValidDevCopy()* method in Algorithm 3 finds a valid device copy of an object, on a given device, that has an index range overlapping with the specified index range. The *getModifiedDevCopy()* method does the same except that it looks for a modified device copy. When copying contents from another device copy ($dCopy_{src}$), the *getOverlapRange()* and *getRemainingRanges()* methods find out

[9]This is the worst case scenario. Suppose that we want to copy N elements into a new device copy and we have N existing valid device copies, each containing one element of what is required; we would have to carry out one copy for each required element in this case (N copies in total).

what index range can be copied from a given device copy ($dCopy_{src}$) and what remaining index ranges (if any) need to be copied from other sources[10]. For example, when copying the $[0, 10)$ index range to a device copy ($dCopy_{dst}$), we might find a $dCopy_{src}$ with contents $[3, 17)$; *getOverlapRange()* would return the overlap between the two index ranges, i.e. $[3, 10)$ in this case. As the overlap is not complete, copying elements $[3, 10)$ from $dCopy_{src}$ would yield one remaining index range ($[0, 3)$) to copy further. In a nutshell, considering that the contents of a copy might need to be copied from more than one source, the copy plan includes the ⟨source, destination, index range⟩ information of all the necessary copy operations to copy the required contents. These copies are carried out by the system considering possible asynchronous execution semantics offered by modern GPUs; the *valid* flag of the device copy (the one being written to) is set in the end.

Algorithm 4: UpdateOthersWhenWritingADeviceCopy

```
   Input: dCopy_w, device copy that is written
          O, object to which dCopy_w belongs
1  range_w ← [offset_w, (offset_w + n_w))
2  begin
3      if mainCopy(O).valid then
4          mainCopy(O).valid ← false
5      foreach dev ∈ AllDevices do
6          dCopies ← getValidDevCopies(O, range_w, dev)
7          foreach dCopy_i ∈ dCopies do
8              if dCopy_i ∧ dCopy_i ≠ dCopy_w then
9                  if dCopy_i.modified then
10                     range_i ← [offset_i, (offset_i + n_i))
11                     subrange ← getOverlapRange(range_i, range_w)
12                     if range_i ≠ subrange then
13                         rem ← getRemainingRanges(range_i, subrange)
14                         Copy rem from dCopy_i to mainCopy(O)
15                     dCopy_i.modified ← false
16                 dCopy_i.valid ← false
```

[10]There could be at most two new index ranges returned by *getRemainingRanges()* considering that each object copy stores consecutive elements, marked by an index range.

When a GPU write access, identified by $\langle devID_{acc}, offset_{acc}, n_{acc} \rangle$ for an object O occurs, it is handled in the following manner. First, a lookup is made to find an existing copy with same index range on the given device (Equation 4.8). If not found, a new device copy ($dCopy_{acc}$) is allocated for the given index range on the given device. Before the actual writing of contents in the current device copy can take place, all its overlapping valid copies are marked invalid, as described in Algorithm 4. The *valid* flag for the main-copy and all other device copies that are overlapping with the currently written device copy ($dCopy_{acc}$) is cleared. However, the framework checks for device copies that have modified contents and where the index range covered by $dCopy_{acc}$ is not a proper superset of the index range covered by those device copies. For example, if the $dCopy_{acc}$ index range is $[0, 10)$ and another device copy with modified contents has index range $[5, 15)$, partial contents of the latter copy not overwritten by $dCopy_{acc}$ (i.e., $[10, 15)$) is copied back to the main-copy before its *modified* and *valid* flags are cleared.

Figure 4.4 summarizes the new memory management mechanism. Whenever a new device copy is allocated on device memory, both its *valid* and *modified* flags are cleared. Later, clearing the *valid* flag for a device copy clears (if not already done) its *modified* flag too; similarly, whenever the *modified* flag is set for a device copy, its *valid* flag is also set if not already done. This means that there could not exist a device copy with *valid* flag cleared but *modified* flag set.

Matrix operands

As already discussed, SkePU matrix objects in the new mechanism are treated differently to enable partial device copies with contents not stored consecutively in the original matrix (see Figure 4.1). In the following, we describe how matrix objects are handled in the new mechanism.

The main-copy (*mainCopy(O)*) of a matrix object O with R rows and C columns can be represented with a 1D index range $[0, R \times C)$ knowing that matrix elements are stored consecutively row-wise. There exists a bijective mapping between a specific row and column

	CPU (main-memory)	GPU (device memory)
Copies	One *main-copy*	Zero or more device-copies(*dCopies*) A device-copy (*dCopy*) identified by <devID, offset, n>

Read access [P - Q)	**if** *main-copy* NOT valid **then**
	Copy all modified device copies overlapping with range [P, Q) back to *main-copy* and clear their *modified* flags.
	Set *valid* flag of *main-copy* if no other modified device copy exist.
	endif
	Read *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		if *dCopy* is not valid **then**
		Copy data from other valid copies
		Set *valid* flag for the *dCopy*
		end if
		Update *lastUsed* flag of the dCopy
		Read the *dCopy*

Write access [P - Q)	**if** *main-copy* NOT valid **then**
	Copy all modified device copies overlapping but not subset of range [P, Q) back to *main-copy* and clear their *modified* flags
	Set *valid* flag of *main-copy* if no other modified device copy exist.
	endif
	Clear *valid* flag for all valid device copies overlapping with range [P, Q)
	Write to *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		Clear *valid* flag for all overlapping copies
		Set *modified* flag for the *dCopy*
		Update *lastUsed* flag of the dCopy
		Write the *dCopy*
		Set (if not already) *valid* flag for the *dCopy*

Read/ Write access	**if** *main-copy* NOT valid **then**
	Copy all modified device copies overlapping with range [P, Q) back to *main-copy* and clear their *modified* flags
	Set *valid* flag of *main-copy* if no other modified device copy exist.
	endif
	Clear *valid* flag for all device copies overlapping with range [P, Q)
	Read/Write *main-copy*
		dCopy = *dCopies*.lookup <devID, offset, n>
		if *dCopy* not found **then**
		Allocate a new *dCopy* <devID, offset, n>
		endif
		if *dCopy* is not valid **then**
		Copy data from other valid copies
		Set *valid* flag for the *dCopy*
		end if
		Clear *valid* flag for all overlapping copies
		Set *modified* flag for the *dCopy*
		Update *lastUsed* flag of the dCopy
		Read/Write the *dCopy*

Figure 4.4: The working of new memory management mechanism for a single container object.

index (ind_r, ind_c) of a $R \times C$ matrix to its linearized (1D-equivalent) index (ind) such that we can convert one to the other knowing that the matrix object O has R rows and C columns and is stored row-wise. In the new mechanism, a matrix device copy $(dCopy_i(O))$ for object O can represent any submatrix of the original matrix and is identified by a tuple $\langle devID_i, offset_i, r_i, c_i \rangle$ where:

- $devID_i$ is an identifier of the device on which the $dCopy_i(O)$ is present,
- $offset_i$ is the 1D index of the first element of $dCopy_i(O)$ with respect to the index range of *mainCopy(O)*,
- r_i is the number of rows stored in the device copy, i.e., $1 \leq r_i \leq$

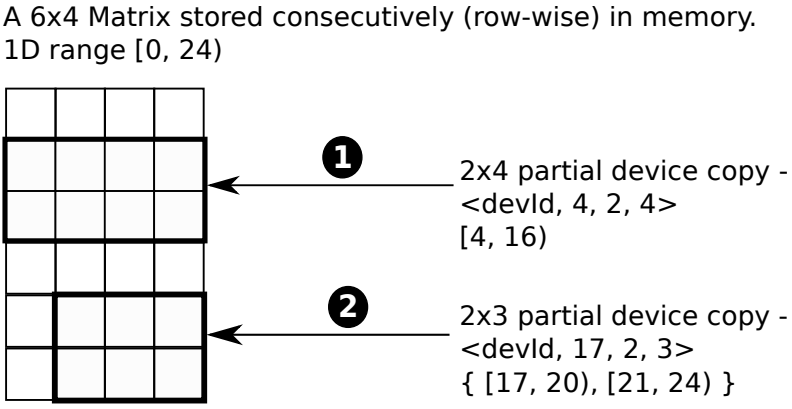


Figure 4.5: Device copies (tuple values, index ranges) in the new memory management mechanism for a matrix object.

R,

- c_i is the number of columns stored in the device copy, i.e., $1 \leq c_i \leq C$,

If $r_i > 1$, c_i should be equal to C if the element range covered by the device copy maps to consecutive elements in the original matrix. Figure 4.5 shows tuple values and index ranges for two partial copies of a 6×4 matrix from Figure 4.1. Considering that the first device copy $dCopy_a(O)$ has $c_a = C$, it can be represented by a single index range:

$$[offset_a,\ offset_a + r_a \times c_a)$$

The second device copy $dCopy_b(O)$ maps to non-consecutive elements in the matrix that can be represented as a sequence of r_b index ranges $\{[range_0), ..., [range_{r_b-1})\}$ where the jth index range in the sequence $(0 \leq j < r_b)$ is calculated as:

$$[start_j,\ start_j + c_a) \text{ where } start_j = offset_a + C \times j$$

Although a matrix device copy may represent multiple (disjoint) index ranges, it is still considered a single device copy in the working

of the new mechanism. The process of marking copies valid, modified etc. remain the same as for vector objects. Contents in the device copy are internally stored consecutively no matter if it represents multiple disjoint index ranges. However, the difference comes in the implementation, e.g., when copying data to/from other copies, a copy operation may be needed for each index range. Also, the overlap between two device copies is checked for each index range that they contain and "no overlap" is defined when there exists no overlap with any index range covered by two device copies.

Remarks

The new mechanism is more powerful when it comes to managing read and write accesses to multiple overlapping copies of an object in different device memories. The consistency problems described with the initial mechanism, as depicted in the two scenarios earlier, are taken care of automatically in the new mechanism. This is achieved by tracking the state of each device copy and by proper handling of other overlapping copies when contents in one device copy is updated. From the programmer's perspective, the new mechanism does not require any new information and works transparently behind the generic container's interface. Besides addressing data consistency issues, the new mechanism improves performance first by not de-allocating device copies whenever contents is modified in the main-copy. Moreover, copying data to a device copy from existing object copies in the same device memory can reduce the communication overhead associated with data transfers between main memory and GPU device memory. Partial device copies of a matrix object are identified with a tuple that allows partial device copies representing any sub-matrix of the original matrix. Last but not the least, support for direct data transfers between two device memories can yield significant savings as only one copy operation is needed rather than two copy operations in the original mechanism (first from one device copy to main-copy and then from main-copy to another device copy).

The granularity of control is more flexible in the new mechanism. Theoretically, the control can happen at individual element level if

there exist device copies consisting of one element. The framework, when managing communication between different copies, considers their index ranges and optimizes data transfers by transferring only required contents. Several optimizations are made inside the implementation of new mechanism, e.g., a list of modified device copies per device memory is maintained for fast lookup.

The new mechanism is very similar to the MSI (Modified, Shared, Invalid) cache coherence protocol [84]. A copy can be in valid (shared in MSI), modified or invalid state. The main difference comes from the concept of (partially) overlapping copies where state of different copies is updated considering their potential overlap, allowing disjoint copies to be modified independently. We will discuss it in more detail when comparing our mechanism with the StarPU MSI implementation in Section 4.4.1.

In both initial and new mechanism, we match a device memory access to an exact copy. For example, if we have a device memory access request $\langle 0, 5, 5\rangle$ (i.e., index range $[5, 10)$ on the device with ID 0) for a vector object, we will check for an exact copy of that vector object with index range $[5, 10)$ in that GPU device memory. If we do not find an existing copy, we will create a new copy for this index range. A possible optimization could be to look for an overlapping copy that totally includes the index range requested, i.e., covers an index range that is a superset of the requested range. However, this might complicate the handling of modifications and invalidations to device copies as partial contents in a device copy might be in different states (valid, invalid, modified). Splitting a device copy into multiple *logical* sub-copies corresponding to different access requests might be an interesting idea and can be investigated in future.

Currently, we keep information about device copies of an object in a C++ STL map container that stores *key,value* pairs in a self-balancing red-black binary tree [36]. It performs operations such as lookup, insertion, and deletion in $\mathrm{O}\big(\log N\big)$ time where N is the total number of entries in the map. There exists other, more specialized data structures that are optimized for comparing ranges, such as segment trees and range trees [35, 214, 85]. The STL map container appears to be a reasonable choice considering that in all SkePU ap-

plications ported so far, we normally get very few copies (e.g. two copies per device memory) per container object. In future, one might investigate other data structures if usage scenarios with hundreds of device copies occur in some application.

4.3 Evaluation

In this section we evaluate the effect of the memory access optimizations implemented inside the SkePU containers with the help of several applications/kernels implemented using SkePU skeletons (see Section 3.4). The evaluation is carried out on a GPU-based system with 2 Intel Xeon quad-core E5520 (2.27GHz) CPUs and 2 NVIDIA Tesla C2050 GPUs. C/C++ and CUDA code is compiled using GCC (v4.8.1) and NVIDIA C (nvcc v0.2.1221 with CUDA 5.0) compilers respectively. We carry out two kinds of evaluation: First, we check the benefits of the memory management implemented in SkePU by comparing with executions where operand data to skeleton calls is transferred back and forth for each call. In the second evaluation, we compare our new memory management mechanism with the initial memory management mechanism found in the initial version of SkePU.

4.3.1 Effect of communication optimizations

In this evaluation, we evaluate the benefits of our memory management inside the SkePU containers for several applications. We do that by comparing the execution time of several SkePU programs with the new memory management mechanism to their execution time where data is not kept on a device memory after a skeleton call execution, i.e., operand data is transferred for each skeleton call back and forth. The benefits of memory optimizations come with GPU execution where communication between different device copies and main-copy of an object can be optimized. Executing all skeleton calls on CPU does not provide any optimization opportunity considering that no data is copied; all accesses are tracked to the main-copy which is already

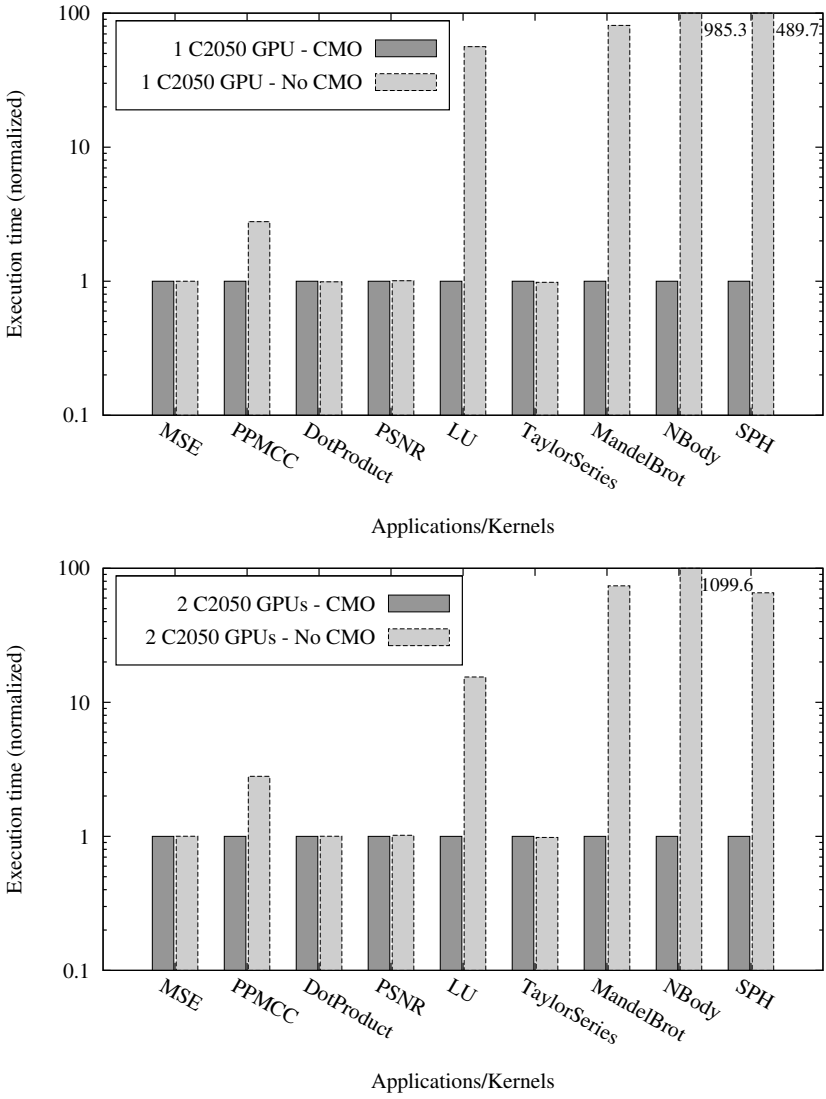


Figure 4.6: Normalized execution times of different SkePU applications/kernels with one or more skeleton calls where each skeleton call is executed on 1 (top) or 2 C2050 GPUs. We compare execution where SkePU communication and memory optimizations (CMO) are enabled to where operand data of skeleton calls is transferred back and forth for each skeleton call (No CMO). The execution time is normalized with CMO execution as baseline.

present in the main memory[11].

Figure 4.6 shows execution of several applications/kernels on two GPU setups (1 C2050 GPU, 2 C2050 GPUs where the work in each skeleton call is equally divided across the 2 GPUs that are of the same type). The execution time is averaged over multiple executions, with different problem sizes (both small and large), for each application/kernel. In both setups, execution with our communication and memory optimizations (CMO), implemented as part of memory management, is compared to one where it is disabled and operand data is transferred back and forth for each skeleton call. There are a few issues to notice in these results:

- For some kernels, there exists no benefit as execution consist of a single skeleton call and transferring data (back and forth) at least once is required in any case. Normally, these kernels are part of a larger application and there we should observe some benefits as we can see with other applications with more than one skeleton call operating on the same data. For these kernels where execution time looks no different between the two alternatives, the overhead of our memory management mechanism in comparison to direct execution (without our memory management enabled) can be observed. The overhead seems to be really negligible (less than 1%) on both GPU setups which shows the efficiency of our memory management mechanism[12].

- For several applications with more than one skeleton call, the benefit is quite substantial for both single and multi GPU executions. In some cases, speedup is more than 100 times which shows the importance of applying communication optimizations in GPU based systems across multiple skeleton calls. For applications such as NBody, passing operand data back and forth for

[11]The effect of caches and NUMA organization are not considered in the SkePU containers and are considered as part of the future work.

[12]In absolute terms, the overhead of SkePU memory management is, on average, less than a microsecond per skeleton call. It is calculated on System A, by averaging the execution time difference of (MSE, PSNR, DotProduct, TaylorSeries) applications over multiple executions with memory management enabled to one where it is disabled.

each time step makes the communication really expensive even for a small simulation (50 timesteps for these experiments).

4.3.2 New versus initial mechanism

Earlier, we have seen the benefits of optimizing communication for operand data across multiple skeleton calls inside the program. In this section, we analyze the improvements of our new memory management mechanism in comparison to the initial mechanism for several applications. In the new mechanism, main improvements come from device copies of an object that (partially) overlap with each other. One possible way to create these partial copies is having one or more skeleton calls in the program that operate on partial elements of a container object, as shown in Scenario One (Section 4.2.2). However, none of the applications that have so far been ported to SkePU has such behavior. The other possibility that results in partial copies in device memory comes with multi-GPU execution where work in each (or some) skeleton call(s) is partitioned across different GPUs. This was shown earlier (Section 4.2.2) with two calls to the MapArray skeleton, and this is found in three of the applications (LU factorization, NBody and SPH) that we have ported to SkePU.

For demonstration, we consider execution of these three applications on the above mentioned GPU-based system. For each application, we do executions over multiple problem sizes with three configurations of memory management, as listed below:

1. *Initial CMO*: Execution with the initial SkePU memory management mechanism.

2. *New CMO*: Execution with our new memory management mechanism. Device-to-device memory transfers between the two peer GPU device memories (using CUDA GPUDirect [193]) are still disabled in this configuration. This means that communication between copies residing in the two device memories will happen indirectly (via the main memory where data will be first transferred from one device memory to main memory and then from main memory to the other device memory).

Table 4.1: Communication volume and device allocation size (in megabytes) for the three applications, accumulated over different problem sizes.

LU Factorization						
CMO Type	HTD	DTD	DTD (peer)	DTH	Total	Alloc
Initial CMO	119.8	0	0	60.1	179.9	359.1
New CMO	60.1	59.4	0	59.7	179.2	4.4
New CMO (peer)	0.6	59.4	59.4	0.3	119.7	4.4
NBody						
CMO Type	HTD	DTD	DTD (peer)	DTH	Total	Alloc
Initial CMO	3121.9	0	0	1135.2	4257.1	6243.9
New CMO	1040.6	1040.6	0	1087.9	3169.1	567.6
New CMO (peer)	0	1040.6	1040.6	47.3	2128.5	567.6
SPH						
CMO Type	HTD	DTD	DTD (peer)	DTH	Total	Alloc
Initial CMO	101.2	0	0	42.7	143.9	203.2
New CMO	33.7	33.7	0	34.1	101.5	4.0
New CMO (peer)	0	33.7	33.7	0.3	67.7	4.0

3. *New CMO (peer)*: Execution with our new memory management mechanism. Device-to-device memory transfers between the two peer GPU device memories (using CUDA GPUDirect [193]) is enabled in this configuration.

The new mechanism transparently finds out (via CUDA API) whether the peer device memory transfers can be enabled or not between different GPU device memories and can optimize communication patterns between such devices. Even if the peer device memory transfers are not available, the new mechanism still improves over the initial mechanism by making execution both consistent (without requiring any `flush` operation[13]) as well as reducing communication to main memory.

Table 4.1 describes the communication volume and device memory allocation size (in megabytes) for the three applications. The values are accumulated over multiple executions of each application with

[13]The `flush` operation is both expensive, as it deallocates all device copies of the object, and exposes data consistency issues to the application programmer.

different problem sizes. For each application, the table lists the size for different types of communication, namely, device to host (DTH), device to device (DTD), device to device between two peer device memories (DTD peer) and host to device (HTD) for each of the three configurations described above. For each application, the table also lists the total communication size (second-last column) for each configuration as well as the accumulated size of device memory allocations (last column) made for different container objects used as operands with the skeleton calls.

Just by looking at the total communication size (second-last column) for each configuration, we can clearly observe the improvements of our new mechanism over the initial one even when the peer device memory transfers are not enabled. The savings are more vital than what one might expect from looking at the total communication size as, in the new mechanism, we do more DTD memory transfers rather than HTD each time. In the initial mechanism, there were no DTD data transfer as can be seen in the table. On modern GPUs (e.g., C2050s that we have used for these experiments), the achieved bandwidth for DTD communication is more than 25 times better than the achieved bandwidth for HTD communication, as found by our experiments on a C2050 GPU.

Enabling the peer device memory transfers, the total communication size reduces even further considering that data can be directly transferred between the two device memories without involving the main-copy that resides in the main memory. This obviously reduces the overall communication size, e.g., communicating 50KB between two device memories via main memory will result in 100KB communication (2 copies of 50KB each) rather than just 50KB when peer transfer is enabled. Furthermore, the achieved bandwidth for peer DTD device transfers is 1.5 to 3 times faster than for HTD. The ratio varies considering whether the HTD transfers are done for page-locked memory or for data with normal (default) allocations; communication with the former is faster than the latter.

Besides the differences in communication size and speed, there comes a substantial saving in the total number of device memory allocations made for the device copies of container objects. The dif-

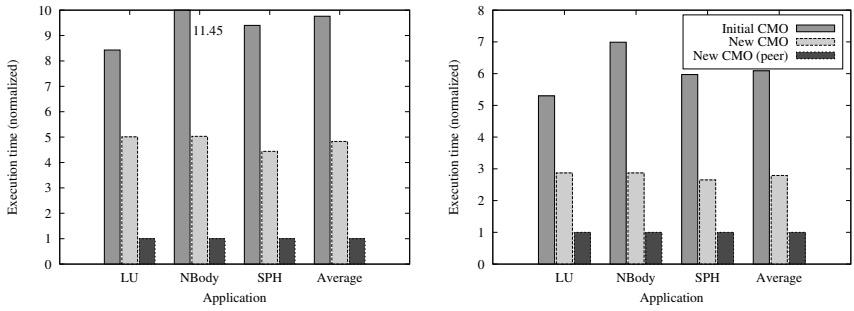


Figure 4.7: Normalized execution time of different memory management configurations for three SkePU application where the main-copy of each object is allocated as pageable/default (left) and as page-locked/pinned (right) memory. An average over all three applications for each configuration is also shown in the end. The execution time is normalized considering *New CMO (peer)* as baseline.

ferences in accumulated device memory allocation size are `82`, `11` and `50` times for the LU, NBody and SPH applications respectively. This is because in the initial mechanism device memory allocations are removed for each write access to the main-copy as well as correct execution requires `flush`ing contents between different skeleton calls. The savings in memory allocations are substantial and can have a major performance impact considering that these device allocations (and corresponding deallocations) are made during the actual skeleton program execution.

It is difficult to quantify the benefits of the new mechanism in terms of performance as the communication is spread over different types (HTD, DTD, DTH etc.). To give a realistic estimate of the communication cost for each memory management configuration in terms of execution time, we do a simple experiment. First, we calculate the effective bandwidth for different communication types on the underlying platform. We do that by modifying the NVIDIA CUDA benchmark for measuring memory bandwidth by adding support for measuring peer device to device data transfer bandwidth. Communication and allocation measurements are carried out for sizes be-

tween 1KB and 64MB with 1KB to 4MB size increments (the same as the ones used by the `bandwidthTest` program in NVIDIA CUDA SDK [168]). We calculate the average for each communication type (HTD, DTH etc.) over different sizes. For HTD and DTH communication, the bandwidth for transferring container objects both with page-locked memory allocation and with default (pageable) memory allocation is calculated. For measuring the execution time of device memory allocations and de-allocations, we design an experiment where we allocate and de-allocate memory for different sizes and record the average number of elements allocated/deallocated within one second, which we call *(de-)allocation bandwidth*.

Once we get the bandwidth numbers by actual experiments, we divide each communication/(de-)allocation size by the bandwidth number that we got for that type, and obtain a time fraction. Afterwards, we calculate the execution time for each configuration by adding fractions for all communication/allocation types in that configuration. Figure 4.7 shows the execution time for the three configurations for each application. The savings are substantial; more than `11` times in the best case. On average, we achieved more than `9` and `6` times reduction in execution time when comparing execution with the initial mechanism to the new mechanism for pageable and page-locked memory allocations respectively.

The new mechanism reduces the total number of data transfers, localizes communication when possible (e.g., DTD instead of HTD) as well as cuts the total number of device memory allocations by `82` times in the best case. Besides making the application execution faster, these savings can have a huge impact on power consumption considering that memory operations consume 1 to 2 orders of magnitude more power than floating-point operations [129, 172]. From the programming perspective, a skeleton program can now be executed on different GPU-based systems without any data consistency issues.

4.4 Containers with StarPU runtime system

When using SkePU atop the StarPU runtime system (see Section 3.3), the skeleton calls are translated to one or more tasks that are submitted to the runtime system. Skeleton calls are executed asynchronously as tasks to exploit possible concurrency between different skeleton calls (e.g., if they operate on different data). Before submitting a task to StarPU, its operand data is registered (if not already done) to the runtime system with information about its starting address, size and access mode in that task. This means that, when SkePU skeleton calls are translated to tasks for the StarPU runtime system, their operand data (in form of container objects) is handled by the StarPU runtime system. Movement of data between different device memories and main memory is managed by StarPU in this case, i.e., our own memory management inside the containers is disabled.

4.4.1 Memory management in StarPU

StarPU implements the MSI protocol [84] for managing accesses to different copies of the same data on different memory units. It optimizes their communication across multiple device memories and can reuse copies across multiple task invocations. Whenever a data present in main memory is registered to StarPU, device copies of this data on all device memories get created; these device allocations are kept until data is used (registered) with the runtime system. The MSI protocol in StarPU works in the following manner:

- A copy is marked invalid when it does not contain the most up-to-date contents.
- A copy of the data is marked as shared when it is valid (invalid flag not set), and there may exist other valid copies as well.
- A copy can be promoted to modified when it is the only valid copy.

The MSI protocol in StarPU is different from our new memory management mechanism. We consider device copies that are not exact replicas of the main-copy of data; device copies in our case can

partially (or fully) overlap with each other as well as with the main-copy of data. For example, in our case, when a device copy is modified, there exists only one valid copy for the index range covered by that modified device copy, but there can exist another modified device copy for the same object if the index ranges of both modified device copies do not overlap with each other. This means that in order to execute a skeleton call that operates on partial elements, we register a new copy for that in StarPU. For multi-GPU execution, where we need to divide the data into equal chunks that are processed on different compute devices, we use the concept of filters in StarPU to partition a registered data entity into smaller chunks that can be processed individually.

```
1 skepu::Vector<...> v0, v1;
2 ...
3 map(v0); // asynchronous - call 1
4 map(v1); // asynchronous - call 2, can overlap with call1
5 ...
6 v1[10] = 11; // blocks until call 2 finished
7 ...
8 std::cout<< v0[i]; // blocks until call 1 finished
```

Listing 4.2: SkePU program execution with StarPU runtime system.

4.4.2 Container interaction with StarPU

We disable the memory management inside our containers when we execute SkePU programs on top of the StarPU runtime system. The containers, in this case, interact with the data management library (DML) present in the runtime system to handle program accesses to container objects which may still be in use, in asynchronous skeleton calls made earlier.

Listing 4.2 shows how SkePU containers provide synchronization when skeleton calls are executed asynchronously with the runtime system. As all skeleton calls are (by default) made asynchronous, both skeleton calls (Lines 3 and 4 in Listing 4.2) can run concurrently as they operate on different operand data. Such inter-call parallelism can be found out by the runtime system automatically considering their

operand data independence. This asynchrony is kept hidden in the main program as, when the user accesses the container objects (e.g., via the [] operator), the container interacts with the runtime DML and ensures that (a) all skeleton calls issued so far operating on this data have finished execution and (b) the most up-to-date data is available in the main memory before continuing with the requested access. In containers, we use proxy elements [157] to distinguish between read and write accesses when, e.g., a vector object is accessed via the [] operator.

We have modified the containers' implementation to achieve integration with the runtime system. For example, when doing our own memory management, the `updateHost()` method copies updated data back to the main-copy from all modified device copies. When using StarPU, it will invoke functions from the runtime DML to make sure that data is available in main memory for reading purposes. Similarly, several other functions have been implemented that internally call methods from the runtime DML API to provide desired semantics behind the same generic STL-like interface to the application programmer.

4.5 SkePU program execution model

SkePU containers are primarily designed to pass operands to skeletons calls in a SkePU program. A typical SkePU program consists of sequential CPU code with skeleton calls nested in the program control flow. The sequential CPU part manages input/output operations and coordinates execution of different skeleton calls in the program. The actual computations are marked by skeleton calls in the program.

Considering the above sequential-style program execution scenario, the SkePU containers are not designed to be thread-safe, i.e., to be used in a concurrent environment outside the skeleton calls. For example, the behavior is undefined if overlapping contents in a vector object $v0$ are read and written simultaneously in multiple threads. Considering our target usage scenarios, making containers thread safe could hurt the performance. The main container operations are per-

formed inside the skeleton implementations which are designed to ensure proper behavior when using multiple threads. For example, the OpenMP implementations of the skeletons in the SkePU library ensure data consistency for the given skeleton semantics. Outside the skeleton implementations, the application programmer needs to either use containers in the sequential source code or provide adequate synchronization when overlapping container data is read and written in a multi-threaded context.

Normally, skeleton calls in a SkePU program are blocking, i.e., control returns from a skeleton call when it is completed and operand data is safe for subsequent program accesses. However, SkePU skeleton calls can be non-blocking when using the StarPU runtime system, i.e., the control returns to the calling thread before the actual call is complete. As shown in Listing 4.2, the containers ensure proper synchronization for program accesses in this case by ensuring that skeleton calls operating on that data complete before those accesses can proceed.

Although the containers are mainly designed for passing operands to SkePU skeleton calls, they can be used with other computations executing on either CPU or GPU. The containers' API used in the SkePU skeleton framework is generic and can be used in other contexts. This could be useful e.g. in programs where not all computations can be modeled with the existing set of skeletons present in SkePU today.

4.6 Summary

In this chapter, we have discussed the role of operand containers for optimizing communication and memory allocation as well as for enabling asynchronous execution for skeleton calls in a transparent manner. Evaluation with several applications/kernels shows benefits for memory optimizations for GPU-based systems. The evaluation also shows that, when not required, the overhead of our memory management inside containers is negligible (less than 1% of program execution time). When used with StarPU, the containers switch their role

from doing memory management on their own to interacting with the data management library in the StarPU runtime system. In any case, they ensure that they provide data consistency in presence of multiple copies of data residing in different memory units (main memory and device memories). Most importantly, accesses to container data from a skeleton program are served in a data consistent manner even in the presence of asynchronous skeleton call executions.

"We stand at the threshold of a many core world. The hardware community is ready to cross this threshold. The parallel software community is not."

Tim Mattson

Chapter 5

PEPPHER Composition tool

In this chapter, we present the *PEPPHER composition tool* that adapts applications written using the PEPPHER component model and offers both static and dynamic composition functionality. The chapter is structured as follows: Section 5.1 describes the PEPPHER framework and component model. Sections 5.2 and 5.3 describe the composition tool, its advanced features and its prototype implementation. Section 5.4 demonstrates composition using the developed composition tool prototype. Section 5.5 presents a comprehensive experimental evaluation of the developed prototype, followed by a summary in Section 5.6.

5.1 PEPPHER Component model

During 2010-2012, we worked in the PEPPHER project [32] which is a European FP7 project. In the PEPPHER project, we proposed a unified framework for programming and optimizing applications for heterogeneous many-core systems to enable both programmability and performance portability. The framework consists of three main parts: (1) a flexible and extensible component model for encapsulating and annotating performance critical parts of the application, (2) adaptive algorithm libraries that implement the same basic functionality

across different architectures, and (3) an efficient run-time system that schedules compiled components across the available resources, using performance information provided by the component layer as well as internal, execution-history-based performance information.

The PEPPHER component model is an essential part of the framework that glues the three parts together. It defines an environment for annotation of native C/C++ based components for homogeneous and heterogeneous multicore and manycore systems, including GPU and multi-GPU based systems. For the same computational functionality, captured as a component, different sequential and explicitly parallel implementation variants using various types of execution units might be provided, together with metadata such as explicitly exposed tunable parameters. The goal is to compose an application from its components and variants such that, depending on the run-time context, the most suitable implementation variant will be chosen automatically for each invocation.

A *PEPPHER component* is an annotated software module that implements a specific functionality declared in a *PEPPHER interface*. A PEPPHER interface is defined by an *interface descriptor*, an XML document that specifies the name, parameter types and access types (read, write or both) of a function to be implemented, and whose performance metrics (e.g., execution time) the prediction functions of component implementations must provide. Interfaces can be generic in static entities such as element types or code; genericity is resolved statically by expansion, as with C++ templates.

Applications for PEPPHER are currently assumed to be written in C/C++. Several implementation variants may implement the same functionality (as represented by a PEPPHER interface), e.g., by different algorithms or for different execution platforms. These implementation variants can exist already as part of some library[1] or can be provided by the programmer (called *expert* programmer by Asanovic et al. [17]). Also, more component implementation variants may be generated automatically from a common source module, e.g., by

[1]For demonstration purpose, we have used CUBLAS [167] and CUSP [29] components for CUDA implementations as shown in Section 5.5.

special compiler transformations or by instantiating or binding tunable parameters. These variants differ by their resource requirements and performance behavior, and thereby become alternative choices for composition whenever the (interface) function is called.

In order to prepare and guide variant selection, component implementations need to expose their relevant properties explicitly to a composition tool (described later). Each PEPPHER component implementation variant provides its own *component descriptor*, an XML document that contains information (meta-data) about the provided and required interface(s), source files, compilation and resource requirements and reference to performance prediction functions. The `main` module of a PEPPHER application is also annotated by its own XML descriptor, which states, e.g., the target execution platform and the overall optimization goal. XML descriptors are chosen over internal code annotations (e.g., pragmas) as the former are non-intrusive to the actual source code and hence provide better separation of concerns. The potential headache associated with writing descriptors in XML is eliminated to a great extent by providing tool support, as shown later. Composition points of PEPPHER components are restricted to calls on general-purpose execution units only. Consequently, all component implementations using hardware accelerators such as GPUs must be wrapped in CPU code containing a platform-specific call to the accelerator.

Component invocations result in *tasks* that are managed by the PEPPHER run-time system and executed non-preemptively. PEPPHER components and tasks are *stateless*. However, the parameter data that they operate on may have state. For this reason, parameters passed in and out of PEPPHER components may be wrapped in special portable, generic, STL-like *container* data structures such as `Vector` and `Matrix` with platform-specific implementations that internally keep track of, e.g., in which memory modules of the target system which parts of the data are currently located or mirrored (*smart containers*)[2]. When applicable, the container state becomes

[2]The smart container concept has co-evolved in SkePU and the PEPPHER component framework during the PEPPHER project. See also Chapter 4.

part of the call context information as it is relevant for performance prediction.

The PEPPHER framework automatically keeps track of the different implementation variants for the identified components, technically by storing their descriptors in repositories that can be explored by the composition tool. The repositories enable organization of source code and XML annotation files in a structured manner and can help keeping files manageable even for a large project.

5.2 Composition tool

Composition is the selection of a specific implementation variant (i.e., callee) for a call to component-provided functionality and the allocation of resources for its execution. Composition is made *context-aware* for performance optimization as performance depends on the current *call context*, which consists of selected input parameter properties (such as size) and currently available resources (such as cores or accelerators). The context parameters to be considered and optionally their *ranges* (e.g., minimum and maximum value) are declared in the PEPPHER interface descriptor. We refer to this considered subset of a call context instance's parameter and resource values shortly as a *context instance*, which is thus a tuple of concrete values for context properties that might influence callee selection.

Composition can be done either statically or dynamically. *Static composition* constructs off-line a *dispatch function* that is evaluated at runtime for a context instance to return a function pointer to the expected best implementation variant. *Dynamic composition* generates code that delegates the actual composition to a context-aware runtime system that records performance history to be used and updated as the application proceeds. Composition can even be done in multiple stages: First, static composition can narrow the set of candidates for the best implementation variant per context instance to a few ones that are registered with the context-aware runtime system that takes the final choice among these at runtime.

Dynamic composition is the default composition mechanism in

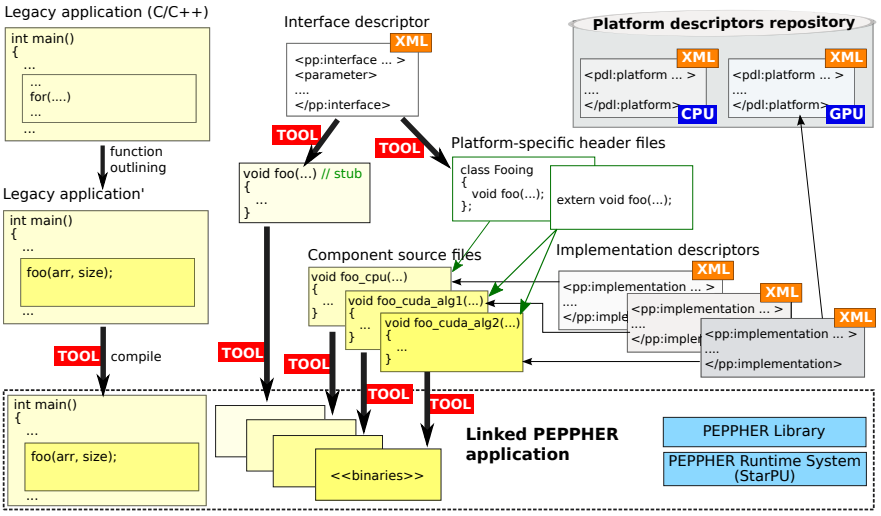


Figure 5.1: Overview of how the composition tool is used to build a PEPPHER application from components. All generation steps marked by "TOOL" are coordinated by the composition tool. The "PEPPHERization" process can be incremental, starting from a sequential legacy application running on a single default execution unit (CPU) for which component implementation variants for CPU and other execution units (such as GPU) are added step by step.

PEPPHER. The *PEPPHER composition tool* deploys the components and builds an executable application. A schematic survey is given in Figure 5.1. It recursively explores all interfaces and components that (may) occur in the given PEPPHER application by browsing the interfaces and components repository. It processes the set of interfaces (descriptors) bottom-up in reverse order of their components' *required interfaces* relation (lifted to the interface level). For each interface (descriptor) and its component implementations, it performs the following tasks:

1. It reads the descriptors and internally represents the metadata of all component implementations that match the target platform, expands generic interfaces and components, and generates platform-specific header files from the interface.

2. It looks up prediction data from the performance data repository or runs microbenchmarking code on the target platform, as specified in the components' performance meta-data.

3. It generates composition code in the form of *stubs* (proxy or wrapper functions) that will perform context-aware composition at runtime. If sufficient performance prediction metadata is available, it constructs performance data and dispatch tables for static composition by evaluating the performance prediction functions for selected context scenarios which could be compacted by machine learning techniques [131]. Otherwise, the generated composition code contains calls to delegate variant selection to runtime, where the runtime system can access its recorded performance history to guide variant selection, in addition to other criteria such as operand data locality.

4. It calls the native compilers, as specified for each component, to produce a binary of every patched component source.

Finally, it links the application's main program and its compiled components together with the generated and compiled stubs, the PEPPHER library and the PEPPHER runtime system to obtain an executable program. The linking step may be architecture dependent (e.g., special handling of different executable formats may be required); the necessary command can be found in the application's `main` module descriptor.

5.3 Prototype implementation

A prototype of the composition tool has been implemented that supports both static and dynamic composition. For dynamic composition, the composition tool generates low-level code to interact with the runtime system in an effective manner. Furthermore it supports component expansion for generic components written using C++ templates as well as user-guided and offline learning based static narrowing of the set of candidates. In this section we describe its design, main features and implementation issues. Figure 5.2 shows a high-

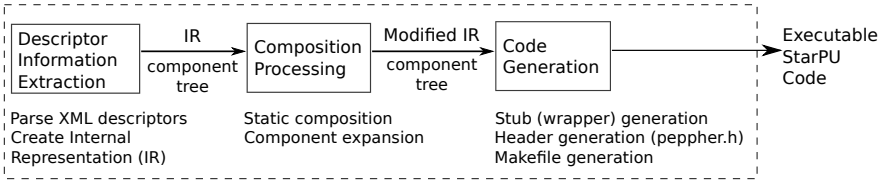


Figure 5.2: Structural overview of the composition tool prototype.

level schematic view of the prototype. Similar to typical compiler frameworks, we decouple composition processing (e.g., static composition decisions) from the XML schema by introducing an intermediate component-tree representation (IR) of the metadata information for the processed component interfaces and implementations. The IR incorporates information not only from the XML descriptors but also information given at composition time. The IR can be processed for different purposes, including:

- Creating multiple concrete components from generic components by expanding template types and tunable parameters,
- Running training executions to prepare for composition decisions,
- Static composition (e.g., using offline learning), and
- Generating code that is executable with the PEPPHER runtime system.

5.3.1 Static composition

Static composition refers to refining the composition choices at compile time, in the extreme case to one possible candidate per call and context instance. Currently, we support two kinds of static composition: The *user-guided static composition* provides a means for the programmer to convey to the composition tool additional expert knowledge about the context that may influence the composition process. For example, a GPU implementation normally runs faster than its CPU counterpart for data parallel problems with large problem size;

thus if such information is statically known, programmers may explicitly specify to select the GPU implementation, and the overhead of the dynamic composition and the risk of wrong dynamic selection can be removed. The composition tool provides simple switches (e.g., `disableImpls`) to enable/disable implementations at composition time without requiring any modifications in the user source code.

Besides user guided static composition, we provide an *adaptive off-line sampling and learning* method to build empirical performance models for components and their implementation variants off-line in order to complement or replace the on-line sampling and learning done in the run-time system. We provide an adaptive sampling method that allows to optimize off-line sampling cost and runtime dispatch overhead while keeping prediction accuracy at a reasonable level. The details of our adaptive off-line sampling and learning method are discussed earlier in Chapter 3.

5.3.2 Component expansion

Component expansion supports genericity on the component parameter types using C++ templates. This enables writing generic components such as sorting that can be used to sort different types of data. The expansion takes place statically. Component expansion for multiple values of tunable parameters to generate multiple implementation variants from a single source is not supported yet and is part of future work.

5.3.3 Code generation

For each component interface, the composition tool generates a wrapper that intercepts each invocation of the component and implements logic to translate the call to one or more tasks for the runtime system. It also performs packing and unpacking of the call arguments to the PEPPHER runtime task handler. The directory structure provides one directory for the main component of the application and one directory for each component used. The different available implementations of a component are organized by platform type (e.g., CPU/OpenMP, CUDA) in different subdirectories. A global registry

of interfaces, implementations and platforms helps the composition tool to navigate this structure and locate the necessary files automatically.

The tool generates wrapper (stub) files providing wrapper functions for different components. Currently, one wrapper file is generated per *component*, containing one *entry wrapper* and multiple *implementation wrappers*. The *entry wrapper* for a component intercepts the component invocation call and implements logic to translate that component call to one or more tasks in the runtime system. A task execution can either be *synchronous* where the calling thread blocks until the task completes or *asynchronous* where the control resumes on the calling thread without waiting for the task completion. The entry wrapper also performs packing and unpacking of arguments of a component call to the PEPPHER runtime task handler. One *implementation wrapper* for a component is generated for each component implementation. Multiple component implementations (and consequently implementation wrappers) can exist for each backend (currently, CPU/OpenMP, CUDA, OpenCL). A implementation wrapper implements the function signature

```
void <name>(void *buffers[],void *arg)
```

that the runtime system expects for a task function[3], and internally delegates the call to the actual component implementation with a different function signature.

Moreover, a header file (`peppher.h`) is generated which internally includes all wrapper files and also contains certain other helping code (e.g., extern declarations). The idea of this header file is to provide a single linking point between the generated code and the normal application code. The *main* program writer only needs to include this header file to enable the composition. Compilation code (`Makefile`) is also generated for compiling and linking the selected (composed) components to build an executable application for a given platform.

[3]As the PEPPHER runtime system is C based and the C language does not permit to call functions with varying types depending on the actual task being run.

5.3.4 Usage of smart containers

A smart container can wrap operand data passed in and out of PEPPHER components while providing a high-level interface to access that data. These containers model and encapsulate the state of their payload data. Four containers are currently implemented: for scalar value (`Scalar`), 1D array (`Vector`), 2D array (`Matrix`) and generic matrix (`GMatrix`). The `GMatrix` container can internally store data in different formats (2D dense, CSR sparse format etc.). All four containers are made generic in the element type, using C++ templates.

We have earlier discussed (in Chapter 4) the usage of vector and matrix containers with the SkePU framework. The vector and matrix containers used here are a more specialized form of their counterparts available in the SkePU library, i.e., these containers always delegate memory management to the StarPU runtime system. This means that these containers internally implement interaction with the data management library (DML) of the PEPPHER runtime system while ensuring data consistency for data that can be accessed both by the runtime system and the application program in an arbitrary fashion. As shown earlier in Section 4.4, implicit synchronization takes place when a container object is accessed in the application program that is in use with asynchronous computations carried out by the runtime system. The containers are made portable and function as regular C++ containers outside the PEPPHER context.

Note that the composition tool also supports parameters of any other C/C++ data types (including user-defined data types) for components. However, for parameters passed using normal C/C++ data types, the composition tool cannot reason about their access patterns in the application program (due to pointer-aliasing and other issues) and hence ensures data consistency by always copying data back to the main memory before returning control back from the component call. Although ensuring consistency, it may prove sub-optimal as data locality cannot be exploited for such parameters across multiple component calls.

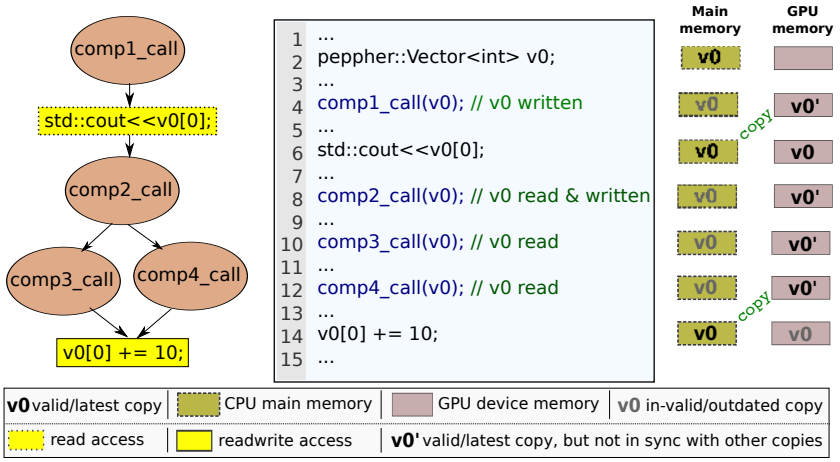


Figure 5.3: Example for usage of smart containers with component calls and other user code. The middle part shows a code scenario with four component calls and one vector operand on a system containing 1 CPU and 1 CUDA GPU. The effect of each instruction on the state of the data is shown to the right of each line, assuming that the four component calls are executed on the GPU. The left part highlights inter-component parallelism for asynchronous component executions based on data dependences.

5.3.5 Inter-component parallelism

In the PEPPHER framework, a major source of parallelism comes from exploitation of independence between different component invocations. This can be obtained when a component invocation is made asynchronous so that subsequent component invocations can overlap in the actual execution. By using the smart containers, the independence between different asynchronous component invocations is implicitly inferred by the PEPPHER runtime system based on data dependences [21].

Figure 5.3 depicts how usage of smart containers can help in achieving inter-component parallelism. The figure shows a simple scenario with four component calls and one vector operand on a system containing 1 CPU and 1 CUDA GPU. When the vector container `v0` is

created, the payload data is placed in the main memory (*main-copy*). Subsequently, depending upon the component calls using that data along their respective data access pattern (read, readwrite or write), other (partial) copies of operand data may get created in different memory units. In this case, we have CUDA device memory which is a separate physical memory. Assuming that all component calls are executed on the GPU, the figure also shows the effect of each instruction execution on the vector data state, i.e., creation/update/invalidation of data copies. As we can see, a PEPPHER container not only keeps track of data copies on different memory units but also helps in minimizing the data communication between different memory units by delaying the communication until it becomes necessary. In this case, only 2 copy operations of data are made in the shown program execution instead of 7 copy operations which are required if one considers each component call independently, as done in, e.g., Kicherer et al. [132, 133].

The first component call (line 4) only writes the data and hence no copy is made. Instead, just a memory allocation is made in the device memory where data is written by the component call. After the completion of the component call (line 4), the main-copy in the main memory is marked outdated which means that any data access to this copy, in future, would first require an update of this copy with the contents of the latest copy. The next statement (line 6) is a read data access[4] from main memory. As the main-copy was earlier marked outdated, a copy from device memory to main memory is implicitly invoked before the actual data access takes place. This is managed by the container in a transparent and consistent manner without requiring any user intervention. The copy in the device memory remains valid as the main-copy is only read. Next, we have a component call (line 8) that both reads and modifies the existing data. As we assume execution of all component calls on the GPU in this scenario, the up-to-date copy already present in the device memory is read and modified. The main-copy again becomes outdated. Afterwards, we

[4]The read and write accesses to container data are distinguished by implementing proxy classes for element data in C++ [11].

have two component calls (line 10 and 12) that both only read the data. Executing these operations on the GPU means that no copy operation is required before or after the component call. Finally the statement in line 14 modifies the data in main memory so data is copied back (implicitly) from the device memory to the main memory before the actual operation takes place. Afterwards, the copy in the device memory is marked outdated and can be de-allocated by the runtime system if it runs short of memory space on the device unit. Doing so would however require re-allocation of memory for future usage.

As all four component calls are asynchronous, the inter-component parallelism is automatically inferred by the runtime system based on data dependencies. In this case, there exists a read-after-write dependency between the first (line 4) and second (line 8) component call; however, independence exists between the third (line 10) and fourth (line 12) component call as they access the same data but in a read only manner.

In the application program, the execution looks no different to the synchronous execution as data consistency is ensured by the smart containers. Blocking is implicitly established for a data access from the application program to data that is still in use with the asynchronous component invocations made earlier (with respect to program control flow) than the current data access.

5.3.6 Intra-component parallelism

A common source of intra-component parallelism is a parallel component implementation, e.g., a CUDA or OpenMP implementation. However, for certain computations, more parallelism can be spawned from a single component invocation by partitioning and dividing the work into several chunks that all can be processed concurrently, possibly on different devices. This is achieved by mapping a single component invocation to multiple independent runtime (sub-)tasks rather than a single task where these (sub-)tasks can be executed on different computing units in parallel. This feature is implemented for data-parallel computations where the final result can be produced either by

simple concatenation or simple reduction of intermediate output results produced by each sub-task (e.g., blocked matrix multiplication, dot product). Extra information about how to `partition` different parameters can be specified in the interface descriptor which allows our framework to determine the partitioning granularity. When applicable, this feature can be used in a transparent manner as it does not require any modification in the user code and/or component implementations.

5.3.7 Support for performance-aware component selection

In the current prototype, the actual implementation variant selection is done by default using the dynamic scheduling and selection capabilities of the PEPPHER runtime system. The actual implementation of performance-aware selection is made transparent in the prototype by providing a simple boolean flag (`useHistoryModels`). The support can be enabled/disabled both for an individual component by specifying the boolean flag in the XML descriptor of that component interface or globally for all components as a command line argument to the composition tool.

5.3.8 Support for PDL (Platform Description Language)

Heterogeneous systems expose many architectural features to the programmer which are important for both performance and portability reasons. Furthermore, different flavors of heterogeneous systems are in use today, ranging from systems with general purpose GPUs to systems with FPGAs and heterogeneous CPUs (e.g., Cell). The PEPPHER Platform Description Language (PDL) [189] allows modeling of key architectural features for these systems, both from hardware and software, and provides a unified interface for both programmers and tool-chains to access that information in a well-defined and explicit manner. We have implemented support for PEPPHER PDL in our framework to allow the component writer to specify constraints/hints by using key architectural properties which are resolved by our system either statically or dynamically.

During composition, a reference to a PEPPHER PDL XML file can be provided that possibly contains information about the underlying hardware/software platform (e.g., compute units, interconnection network). The tool parses information present in the PDL XML file and internally generates code to represent and serialise the information in the form of key-value properties. When using PDL, the tool generates two extra files: `pdl.dat` that stores PDL information used during the composition decisions, and `pdl_reader.h` that reads this information during application initialization time into internal data structures and provides methods for easy information lookup from the C++ composition code. The `pdl_reader.h` gets automatically included in the `peppher.h` header file so no extra include directive is required in the user code. Besides other things, this PDL information can be used for conditional composition.

5.3.9 Conditional composition

Normally, the composition decisions are made at runtime by considering information sources such as historical execution records, data locality for operand data as well as the current system workload. Considering that these decisions are made at runtime, the decision overhead could become the performance bottleneck as the number of implementations increases for a component. In some situations, the (component) implementation writer may provide some useful information that can help in pruning some implementation choices. This may not only help in reducing the runtime decision overhead but also enables making more intelligent user-guided decisions which is not possible otherwise. One such kind of information is constraints on selectability of an implementation variant that an implementation writer can specify. These constraints can be specified based on the following information sources:

- *system resources and topology:* information about number, type and frequency of processing cores; NUMA organization; cache types and sizes; memory organization and sizes; interconnect type and capacity etc.

- *software platform:* information about availability of a particular software library or a certain version of a library; device drivers and their versions; compilers and their versions etc.
- *system runtime characteristics:* information about current CPU and GPU utilization; application memory footprint; cache hit and miss ratio; effective bandwidth etc.
- *component call properties:* information about operand data sizes; actual contents of operand data (sparsity, sortedness) etc.

The first two information sources (i.e. information about system resources and software platform) are known, in many cases before the actual program execution starts. The information about the system runtime characteristics can be collected and made available by a runtime system, during the actual program execution; either by monitoring system counters, or by employing other application monitoring mechanisms. Similarly, information about the actual component call is available for decision making at the component invocation time.

The constraints on selectability of an implementation variant can be of two kinds: soft and hard. *Soft* constraints can be safely ignored/violated without compromising the correctness of the program execution. They are important only for performance reasons. *Hard* constraints are constraints that must be respected to ensure correctness of program execution.

For systematic decision making in a portable manner, information about system resources and software platform should be made available via a standard interface so that the component writer can write constraints for a component implementation that can work across different systems. Underneath that interface, the information about system resources and software platform can be collected via system-/OS-specific routines or by doing micro-benchmarking. Libraries such as hwloc [44] implement a mechanism to lookup such information across different system/OS configurations. One example of providing a generic interface for accessing information about the underlying hardware and software platform is PEPPHER PDL. Similarly, the

runtime system should expose a generic interface to lookup and "refer to" information about runtime system characteristics as well as component call properties.

We have implemented support in our composition tool for specification of constraints on the selectability of an implementation variant. We provide a systematic interface for the user to access information available via PDL as well as information about the component call properties. We do not differentiate between hard and soft constraints, considering that performance is important for us. This means that we only consider an implementation for selection/composition when it satisfies even a soft constraint although it is safe (i.e., functionally correct) to use it otherwise. The constraints are specified in the implementation descriptor using the `validIf` attribute. For example, in the following example:

```
<peppher:implementation name="..." validIf="A.isCSRMatrix()">
```

the implementation is restricted based on value of an input parameter `A` which is an object of a matrix smart container type that internally stores data in either dense or sparse formats. In this case the implementation will only be considered for selection when `A` is stored in a sparse (CSR) format. Multiple constraints can be combined using operators (AND, OR etc.) available in C/C++ [5]. For example:

```
<peppher:implementation name="..."
         validIf="A.isDenseMatrix().AND.size==64">
```

the implementation is considered for selection only when the matrix is stored in dense format and the value of the `size` parameter is exactly `64`.

Information from PDL is made accessible for efficient runtime lookup by loading it at program initialization time (see Section 5.3.8). For example, a CUDA implementation may require availability of at least 16 streaming multiprocessors for execution as specified below:

```
<peppher:implementation name="..."
         validIf='pdl::getIntProperty("numCudaSM") .GE. 16'>
```

[5] As XML restricts usage of certain special characters, we encode them. For example, we represent the logical AND operator `&&` using `.AND.`

Figure 5.4: Tool help available at command-line.

As these constraints use the information about the actual operand values passed to the component call as well as refer to a PDL specification of the target system, they are resolved at runtime. However, the overhead of this runtime lookup is minimal as shown later by experiments.

5.3.10 Documentation and User manual

A user manual is available for the current prototype. It describes the different features and how to use them with the help of code examples. A basic help about the tool usage is also provided at the command-line, as shown in Figure 5.4. Moreover, several small exam-

Figure 5.5: Directory structure before and after generation of the basic PEPPHER component skeleton files from a simple C/C++ header file containing a method declaration.

ple applications are provided to let the users get familiar with different features offered by the current prototype.

5.3.11 Utility mode

Porting existing applications to the PEPPHER framework (PEPPHER-ization) requires writing XML descriptors for interface and implementations as well as adding new implementation variants. To facilitate this process, the current prototype can generate a basic skeleton of these XML and C/C++ source files required for writing PEPPHER components from a simple C/C++ method declaration. This can be really advantageous as writing XML files from scratch can become a tedious task. Our experience with porting several applications shows that the XML skeleton generation by the tool is quite useful as we are required to only fill in certain missing information. For example, the tool can successfully detect template parameters as well as suggest values for the data access mode field of the descriptors by analyzing 'const' and 'pass by reference' semantics of the function arguments.

5.4 Composition example

In the following, we will use the sparse matrix-vector multiplication as an example to describe the complete composition (PEPPHER-ization) process. Figure 5.6 shows a graphical depiction of the complete process and how the composition tool helps by first generating the skele-

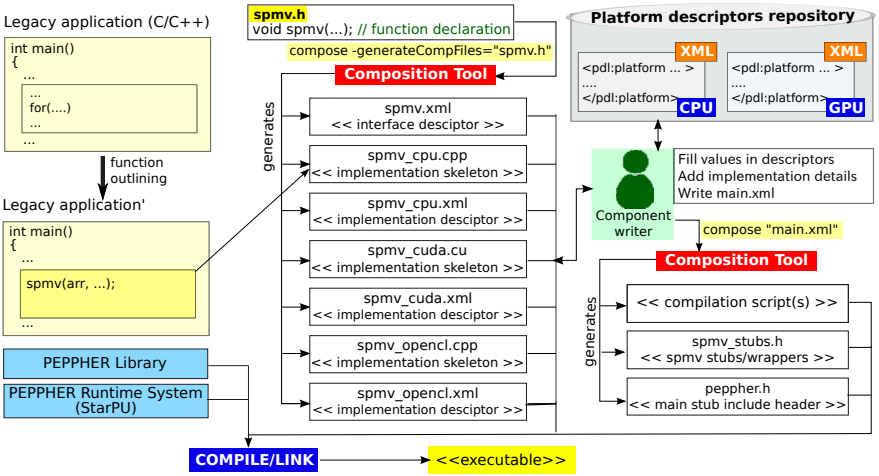


Figure 5.6: The "PEPPHER-ization" process for `spmv` computation using the composition tool prototype starting from a sequential legacy application. The figure also shows how the composition tool helps in writing components, by generating skeleton code, besides doing the composition.

tons for both XML-descriptors and implementation files and later by generating the final composition code.

The process starts with generation of basic skeleton files for components from a C/C++ header file that includes the method signature

```
void spmv(float *values, int nnz, int nrows, int ncols, int
  first, size_t *colidxs, size_t *rowPtr, float *x, float *y);
```

The composition tool can be invoked to generate component skeleton files from this method declaration: As shown in Figure 5.5, the command

```
compose -generateCompFiles="spmv.h"
```

generates the XML descriptors with most information pre-filled as well as basic skeletons for implementation files. The programmer can then fill in the remaining details both in the XML descriptors (e.g. preferences for partitioning of input operands etc.) as well as the implementation files. Listings 5.1 and 5.2 show XML descriptors for the SpMV component interface and a CUDA component implementa-

```
<peppher:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:peppher="http://www.peppher.eu/ComponentMetaData0.1"
 xsi:schemaLocation="http://www.peppher.eu/ComponentMeta0.1/CompMeta.xsd" >

<peppher:interface name=" spmv " impPath="./spmv/">
  <peppher:parameters>
   <peppher:parameter name="values" type="float *" accessMode="read"
        numElements="nnz" csr_nrows="nrows" csr_first="first"
        csr_row_offsets="row_offsets" csr_col_indices="col_indices"
        partition="nnz/2" />
   <peppher:parameter name="nnz" type="unsigned int" accessMode="read" />
   <peppher:parameter name="nrows" type="unsigned int" accessMode="read" />
   <peppher:parameter name="first" type="unsigned int" accessMode="read" />
   <peppher:parameter name="col_indices" type="unsigned int *"
        accessMode="read" numElements="nnz" />
   <peppher:parameter name="row_offsets" type="unsigned int *"
        accessMode="read" numElements="nrows+1" />
   <peppher:parameter name="x" type="float *" accessMode="read"
        numElements="nrows" />
   <peppher:parameter name="y" type="float *" accessMode="write"
        numElements="nrows" partition="nrows/2" />
  </peppher:parameters>
</peppher:interface>
</peppher:component>
```

Listing 5.1: Interface descriptor for the SpMV component.

tion respectively. For this example application, we used one parallel OpenMP implementation for CPUs and a highly optimized CUDA algorithm provided by NVIDIA as part of their CUSP library [29].

In the application's main module `main.cpp`, we only need to add one include statement for `peppher.h` and calls to the macros `PEPPHER_INITIALIZE()` and `PEPPHER_SHUTDOWN()` in the beginning and end of the `main` function, which in this case contains just a call to the `spmv` component. The last step is writing of the XML descriptor (`main.xml`) for the main function. The composition tool can now be called for generating composition code, by giving a reference to the main descriptor:

```
compose main.xml
```

This generates all the wrapper files (shown in Listings 5.3, 5.4) and compilation code (makefile), necessary to compile and build the executable with the runtime support. The resulting executable can then be executed on the given platform.

```
<peppher:component xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
 xmlns:peppher="http://www.peppher.eu/ComponentMetaData0.1"
 xsi:schemaLocation="http://www.peppher.eu/ComponentMeta0.1/CompMeta.xsd" >

 <peppher:implementation name="spmv_cuda">
  <peppher:sourceFiles>
   <peppher:sourceFile name="spmv_cuda.cu" version="1.0" language="CUDA">
    <peppher:compilation>
     <peppher:compiler name="nvcc" flags="-Icuda" />
    </peppher:compilation>
   </peppher:sourceFile>
  </peppher:sourceFiles>

  <peppher:providedInterfaces>
   <peppher:providedInterface name="spmv" />
  </peppher:providedInterfaces>

  <peppher:targetPlatform name="CUDA" />

 </peppher:implementation>
</peppher:component>
```

Listing 5.2: A CUDA implementation descriptor for the SpMV component.

5.5 Evaluation

For evaluation, we implemented (PEPPHERized) several applications from the RODINIA benchmark suite [53], two scientific kernels (dense matrix-matrix and sparse matrix-vector multiplication) and a light-field image refocusing application, using the composition tool. Two evaluation platforms are used: The main evaluation platform is *System A* with Xeon® E5520 CPUs running at 2.27GHz with 2 NVIDIA® C2050 GPUs with L1/L2 cache support. *System B* with Xeon® X5550 CPUs running at 2.67GHz with a lower-end GPU (NVIDIA® C1060 GPU) is used for showing performance portability. The compilation is carried out using GCC (v4.6) and NVIDIA C (nvcc v0.2.1221 with CUDA 4.2) compiler for C/C++ and CUDA code respectively.

5.5.1 Productivity evaluation

The current composition tool prototype generates low-level glue code to use the runtime system. This essentially allows application pro-

```
#ifndef SPMV_WRAPPERR_H
#define SPMV_WRAPPERR_H

typedef struct {
  struct starpu_codelet cl_spmv;
  int cl_spmv_init;
}struct_spmv;

extern void spmv_cpu(float *values, unsigned int nnz, unsigned int nrows, unsigned
int first, unsigned int *col_indices, unsigned int *row_offsets, float *x, float *y);

/* wrapper from StarPU convention to CPU implementation convention */
void spmv_cpu_wrapper(void *buffers[], void *_args)
{
 spmv_cpu ((float *)STARPU_CSR_GET_NZVAL((struct starpu_csr_interface *)buffers[0]),
   STARPU_CSR_GET_NNZ((struct starpu_vector_interface *)buffers[0]),
   STARPU_CSR_GET_NROW((struct starpu_csr_interface *)buffers[0]),
   STARPU_CSR_GET_FIRSTENTRY((struct starpu_csr_interface *)buffers[0]),
   (unsigned int *)STARPU_CSR_GET_COLIND((struct starpu_csr_interface *)buffers[0]),
   (unsigned int *)STARPU_CSR_GET_ROWPTR((struct starpu_csr_interface *)buffers[0]),
   (float *)STARPU_VECTOR_GET_PTR((struct starpu_vector_interface *)buffers[1]),
   (float *)STARPU_VECTOR_GET_PTR((struct starpu_vector_interface *)buffers[2])
  );
}

extern void spmv_cuda(float *values, unsigned int nnz, unsigned int nrows, unsigned
int first, unsigned int *col_indices, unsigned int *row_offsets, float *x, float *y);

/* wrapper from StarPU convention to CUDA implementation convention */
void spmv_cuda_wrapper(void *buffers[], void *_args)
{
 spmv_cuda((float *)STARPU_CSR_GET_NZVAL((struct starpu_csr_interface *)buffers[0]),
   STARPU_CSR_GET_NNZ((struct starpu_vector_interface *)buffers[0]),
   STARPU_CSR_GET_NROW((struct starpu_csr_interface *)buffers[0]),
   STARPU_CSR_GET_FIRSTENTRY((struct starpu_csr_interface *)buffers[0]),
   (unsigned int *)STARPU_CSR_GET_COLIND((struct starpu_csr_interface *)buffers[0]),
   (unsigned int *)STARPU_CSR_GET_ROWPTR((struct starpu_csr_interface *)buffers[0]),
   (float *)STARPU_VECTOR_GET_PTR((struct starpu_vector_interface *)buffers[1]),
   (float *)STARPU_VECTOR_GET_PTR((struct starpu_vector_interface *)buffers[2])
  );
}

void spmv(float *values, unsigned int nnz, unsigned int nrows, unsigned int first,
     unsigned int *col_indices, unsigned int *row_offsets, float *x, float *y)
{
  static struct_spmv * objSt_spmv = NULL;
  if(objSt_spmv == NULL) {
    objSt_spmv = (struct_spmv *) malloc (sizeof(struct_spmv));
    memset(&(objSt_spmv->cl_spmv), 0, sizeof(objSt_spmv->cl_spmv));
    objSt_spmv->cl_spmv_init = 0;
  }

  if(!objSt_spmv->cl_spmv_init) // codelete initialization only at first invocation
  {
    objSt_spmv->cl_spmv.where = 0 | STARPU_CPU | STARPU_CUDA;
    objSt_spmv->cl_spmv.cpu_funcs[0] = spmv_cpu_wrapper;
    objSt_spmv->cl_spmv.cpu_funcs[1] = NULL;

    objSt_spmv->cl_spmv.cuda_funcs[0] = spmv_cuda_wrapper;
    objSt_spmv->cl_spmv.cuda_funcs[1] = NULL;
    objSt_spmv->cl_spmv.nbuffers = 3;
...
```

Listing 5.3: `spmv_wrapper.h`: wrapper code for the SpMV component, generated automatically by our tool.

```
...
    objSt_spmv->cl_spmv.modes[0] = STARPU_R;
    objSt_spmv->cl_spmv.modes[1] = STARPU_R;
    objSt_spmv->cl_spmv.modes[2] = STARPU_W;
    objSt_spmv->cl_spmv_init = 1;
 }

 /* register operands with starpu */
 starpu_data_handle_t values_handle;
 struct starpu_data_filter values_filter_handle;
 memset(&values_filter_handle, 0, sizeof(values_filter_handle));
 values_filter_handle.filter_func = starpu_vertical_block_filter_func_csr;
 values_filter_handle.nchildren = (nnz/(nnz/2));

 starpu_data_handle_t x_handle;

 starpu_data_handle_t y_handle;
 struct starpu_data_filter y_filter_handle;
 memset(&y_filter_handle, 0, sizeof(y_filter_handle));
 y_filter_handle.filter_func = starpu_block_filter_func_vector;
 y_filter_handle.nchildren = (nrows/(nrows/2));


 starpu_csr_data_register(&values_handle, 0, nnz, nrows, (uintptr_t)values,
      (uint32_t *)col_indices, (uint32_t *)row_offsets, first, sizeof(values[0]));
 starpu_data_partition(values_handle, &values_filter_handle);

 starpu_vector_data_register(&x_handle, 0, (uintptr_t)x, nrows, sizeof(x[0]));

 starpu_vector_data_register( &y_handle, 0, (uintptr_t)y, nrows, sizeof(y[0]));
 starpu_data_partition(y_handle, &y_filter_handle);

 /* create multiple tasks by partitioning the work into multiple tasks */
 for(int p=0; p<(nrows/(nrows/2)); p++)
 {
   /* prepare a task */
   struct starpu_task *task = starpu_task_create();
   task->synchronous = 0;
   task->cl = &(objSt_spmv->cl_spmv);
   task->handles[0] = starpu_data_get_sub_data(values_handle, 1, p);
   task->handles[1] = x_handle;
   task->handles[2] = starpu_data_get_sub_data(y_handle, 1, p);

   /* submit the task with CPU and CUDA implementations */
   int ret = starpu_task_submit(task);
   if (ret == -ENODEV) {
     fprintf(stderr, "ERROR: No worker may execute this task\n");
     return;
   }
 }

 starpu_task_wait_for_all();

 starpu_data_unpartition(values_handle, 0);
 starpu_data_unregister(values_handle);

 starpu_data_unregister(x_handle);

 starpu_data_unpartition(y_handle, 0);
 starpu_data_unregister(y_handle);
}

#endif
```

Listing 5.4: `spmv_wrapper.h` (cont.): wrapper code for the SpMV component, generated automatically by our tool.

Application	Savings (LOC, %)	Application	Savings (LOC, %)
bfs	108, 42	cfd	123, 62
hotspot	120, 37	lud	76, 15
nw	90, 25	particlefilter	96, 15
pathfinder	89, 48	nn	101, 41
streamcluster	84, 5	leukocyte	271, 8
b+tree	211, 10	lavaMD	94, 22
spmv	83, 29	sgemm	89, 63
Image Refocusing	113, 11	ODE Solver	452, 57

Table 5.1: Savings in total source LOC (Lines of Code) written by the programmer when using the composition tool compared to an equivalent code written directly using the runtime system.

grams to run using the runtime system without requiring the programmer to actually write code for the runtime system. In the following, we compare how much we gain in terms of programming effort with this code generation functionality.

Table 5.1 shows a simple comparison of the source code written by the programmer when doing hand-written implementations for the runtime system compared to when using the composition tool. The comparison is done using the standard LOC (Lines Of Code) metric [173] for all applications. For the ODE Solver application, we have considered LOC related to the ODE solver, and not the complete LibSolve library which contains more than 12,000 LOC. As we can see in Table 5.1, savings in terms of LOC are significant (up to 63% for `cfd` and `sgemm`). These savings become even more significant considering the fact that the source code for the runtime system is at a low level of abstraction (plain C) and requires handling concurrency issues associated with asynchronous task executions.

The above comparison does not consider the XML descriptors which one needs to provide when using the composition tool. However, this is justified as the XML descriptor skeletons generated automatically from the C/C++ function declaration are already quite concise. For all these applications, we have used the XML descriptor generation feature of the composition tool to generate the XML descriptors and have just filled in the missing information (e.g., values for the

partitioning attribute).

In the remaining part of this section, we will evaluate the efficiency of the generated code by performance evaluation.

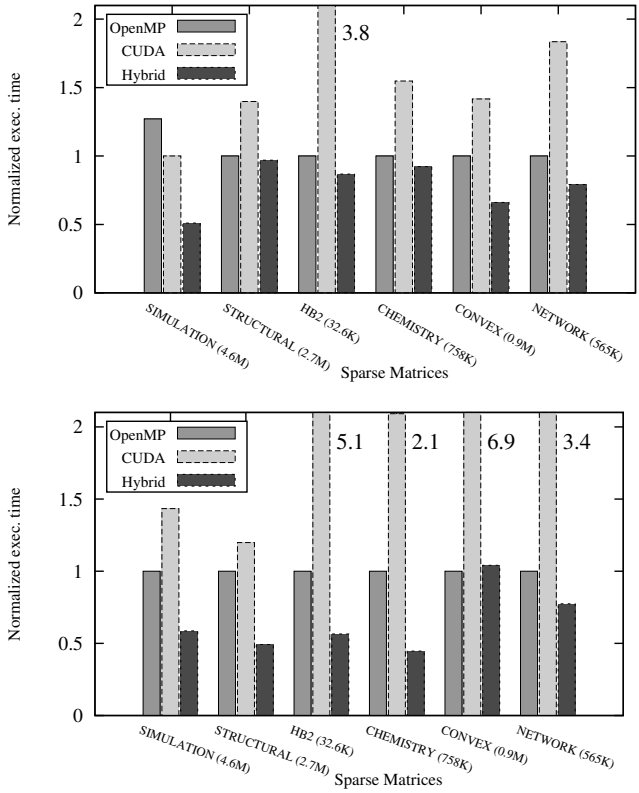


Figure 5.7: Sparse matrix vector product execution for different matrices (with different numbers of non-zero elements) from the UF collection [75] on System A (top: using 1 C2050 GPU) and System B (bottom) respectively.

5.5.2 Hybrid execution

As discussed in Section 5.3.6, a key aspect of the PEPPHER component model is hybrid execution where, instead of using either a GPU

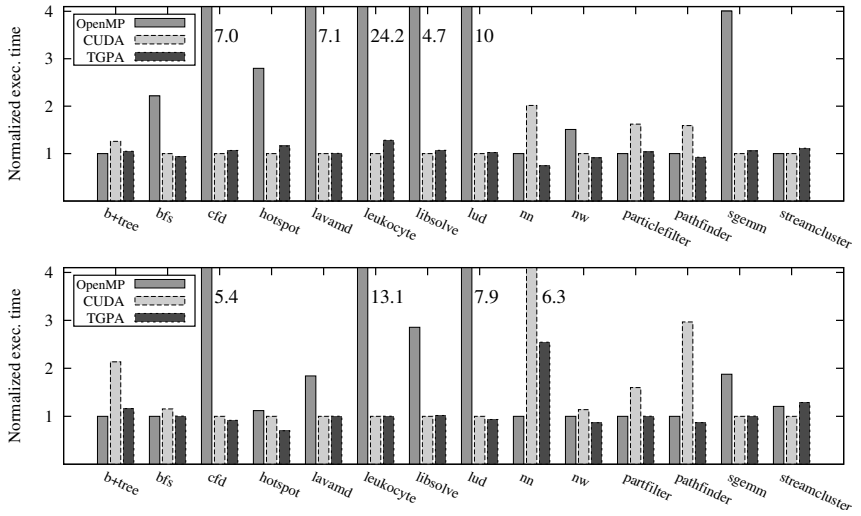


Figure 5.8: Execution times for applications from the Rodinia benchmark suite, an ODE solver and sgemm with CUDA, OpenMP and our tool-generated performance-aware code (TGPA) on System A (top) and B (bottom).

(CUDA) or CPU (OpenMP) implementation, the execution work is partitioned into multiple parts which can be processed concurrently, possibly on different devices by different component implementations. Figure 5.7 shows the advantage of hybrid execution for sparse matrix vector product execution using matrices from the UF collection [75] as example data. The hybrid execution using the code generated by the composition tool supersedes direct OpenMP and direct CUDA execution and, in some cases, reduces the execution time to half of the best performing variant. The speedup by hybrid execution is obtained by dividing the computation work into multiple chunks and executing them on OpenMP and CUDA in parallel rather than using either of them. When applicable (see Section 5.3.6), this feature can be used in a transparent manner (controlled using the `partition` option in the interface descriptor) for an application, as it does not require any modification in the user code and/or component implementations.

5.5.3 Dynamic Scheduling and selection

The architectural and algorithmic differences between different devices (e.g., CPU, GPU with/without cache) and applications often have a profound effect on the achieved performance. However, these execution differences are often hard to model statically as they can originate from various sources (hardware architecture, application/algorithm, input data, problem size etc.). Dynamic scheduling and selection can help in this case by deciding which implementation/device to use by considering previous historical execution information as well as information about current load balance, data locality and potential data-transfer cost (performance-aware dynamic scheduling and selection). Our tool generates the necessary code for using performance-aware scheduling and selection policies offered in the runtime system. Figure 5.8 shows how the usage of dynamic scheduling and selection can help in achieving better performance on two heterogeneous architectures for a variety of applications. The execution time is averaged over different problem sizes. As we can see the execution time with generated code closely follows the best implementation from OpenMP and CUDA for all these applications. In some cases, the execution with dynamic scheduling and selection supersedes the best static selection by making appropriate decisions for each problem size. Above all, it can effectively adjust to the architectural differences as depicted in Figure 5.8. This is achieved by effective utilization of the performance-aware dynamic scheduling and selection mechanism offered by the PEPPHER runtime system.

5.5.4 Smart containers and inter-component parallelism

As mentioned earlier, smart containers provide a high-level abstraction of operand data and underneath optimize data communication for GPU execution while ensuring data consistency for accesses in the user program. Figure 5.9(left) highlights inter-component parallelism (i.e., data independence) for the light-field image refocusing application [163] across three image color channels. Furthermore, a single refocusing call can internally be partitioned into two or more independent parts (sub-tasks) based on input data to further exploit

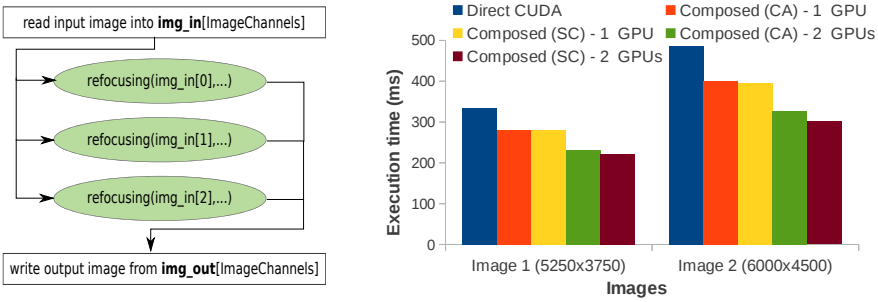
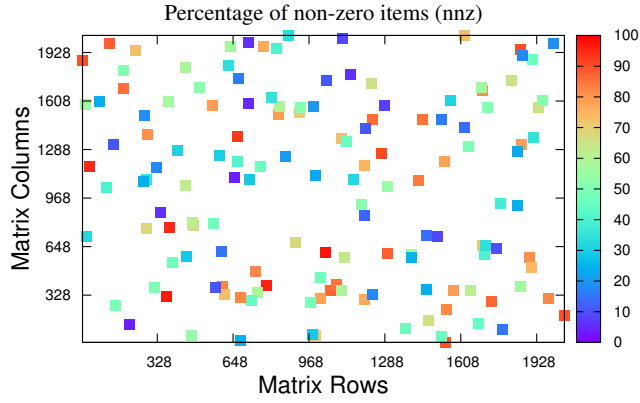


Figure 5.9: Light-field image refocusing application [163]: (left) shows inter-component parallelism; (right) compares direct execution on a CUDA GPU with the composed version, using either C-Arrays (CA) or Smart Containers (SC).

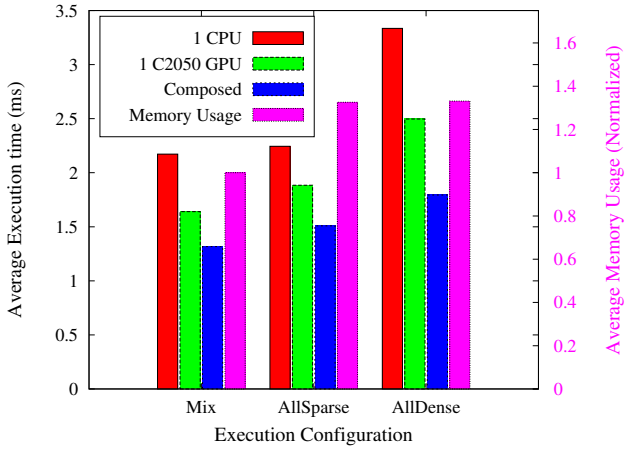
intra-component parallelism (see Section 5.3.6).

Figure 5.9(right) compares the refocusing application using direct execution on the GPU with different versions of the composed application. The composed version using C-Arrays (CA) only exploits intra-component parallelism by internally dividing each refocusing call into *two* parts. This gives a significant performance improvement over direct execution even on a single GPU that supports concurrent kernel executions by overlapping the memory transfers for the second part with computation work of the first part. Using two GPUs further reduces execution time by scheduling both computations on different GPUs.

Besides intra-component parallelism, the version using smart containers (SC) can exploit inter-component parallelism across different image channels and hence performs better than the version using C-Arrays, especially when using multiple GPUs. Once PEPPHERized, this potential for both intra- and inter-component parallelism as well as scaling across multiple GPUs can be exploited without requiring any further modifications in the user code.



(a) Matrices Sparsity



(b) Execution time and memory usage.

Figure 5.10: Conditional composition of `mvmult` computation. (a) shows the distribution of matrices over number of rows, columns and non-zero elements whereas (b) shows execution time (ms), averaged over `mvmult` execution for all matrices. It shows execution on a CPU, a C2050 GPU and composed execution where either the CPU or GPU is used for each `mvmult` call (dynamic performance aware scheduling and selection). Furthermore, it shows memory usage (normalised) for different execution configurations.

Name	**For**	**Valid If**
mvmult_cpu_atlas	CPU	A.isDenseMatrix() .AND. pdl::getBoolProperty("atlasBlasAvailable")
mvmult_cpu_goto	CPU	A.isDenseMatrix() .AND. pdl::getBoolProperty("gotoBlasAvailable")
mvmult_cpu_csr	CPU	A.isSparseMatrix()
mvmult_cuda_cublas	CUDA	A.isDenseMatrix() .AND. pdl::getBoolProperty("cuBlasAvailable")
mvmult_cuda_cusparse	CUDA	A.isSparseMatrix() .AND. pdl::getBoolProperty("cuSparseAvailable")

Table 5.2: Implementations for `mvmult` component.

5.5.5 Conditional composition

The conditional implementation selection implemented in the current prototype enables conditional composition at runtime. To demonstrate conditional composition, we consider a matrix-vector multiplication (`mvmult`) computation. For the evaluation, we generate 200 matrices, randomized over number of rows, columns and non-zero elements (`nnz`). Figure 5.10a shows the randomness distribution of generated matrices based on number of rows, columns and non-zero elements. We use the generic matrix (*GMatrix*) smart container for passing operand data, which considers sparsity in the matrix data and internally stores the matrix data in either dense or sparse[6] format. Currently, the container is parameterized in which format to use for a given operand, but in future this decision could be done automatically.

Following the PEPPHERization process described earlier, we have made a generic `mvmult` component with the following signature:

```
template <typename T>
void mvmult(peppher::GMatrix<T> &A, peppher::Vector<T> &x,
            peppher::Vector<T> &y);
```

The above component can do matrix-vector multiplication for matrices with different number of non-zero elements. However, depending on whether the matrix is stored in sparse or dense format, different implementations can run faster. Table 5.2 lists the implementations that we have devised for this component. Most of them are created by wrapping optimized library functions available for this computation (e.g., AtlasBLAS and GotoBLAS for `sgemv` on CPU). The table also lists the pre-conditions (i.e. the `validIf` attribute in the implementation descriptor, see Section 5.3.9) for selectability of each implementation.

Figure 5.10b shows the execution behavior with three configurations:

1. *AllSparse* when all matrices are stored in sparse format.

[6]For sparse representation, the matrix container currently uses CSR (Compressed Storage Row) format but support for more formats can be added in future.

2. *AllDense* when all matrices are stored in dense format.

3. *Mix* when all matrices are stored in sparse format except those having more than 50% non-zero elements;[7] they are stored in dense format.

These different configurations are orthogonal to the actual component and its implementations as they are not modified in this process. However, these configurations do affect the selection of implementations used for `mvmult` computation (`ValidIf` attribute). For example, when a matrix is stored in sparse format, all implementations with `A.isDenseMatrix()` condition cannot be selected and vice versa.

As shown in the figure, the *Mix* format performs better, on average, than any other configuration considering both execution time and memory usage. We can also see the improvements of dynamic performance aware scheduling and selection over using either CPU or GPU for all executions.

5.5.6 PEPPHER Runtime Overhead

The main runtime overhead of the PEPPHER framework is overhead of the PEPPHER runtime system. Measuring overhead of the runtime system is, in general, a nontrivial task as it depends on the computation structure and granularity, scheduling and selection policy, worker types as well as hardware and software architecture. Micro-benchmarking results reported in [19] show that the task creation and dispatch overhead of the runtime system is less than two microseconds on an Intel Nehalem X5650 CPU. However, according to our measurements on System A (as shown earlier in Section 3.4.3), it could go up to 44 microseconds depending on the number and type of workers used. This overhead is still negligible when considering that the task granularity is normally much higher (in milliseconds). Furthermore, the runtime overhead can be subsidized by potential gains of dynamic

[7] The 50% threshold is chosen arbitrarily in this case as our focus is on demonstrating conditional composition rather than finding the optimum sparsity threshold, which could be a separate tuning parameter [12].

performance-aware scheduling and selection, and overlapping of communication and computation as offered by the runtime system.

5.6 Summary

In this chapter, we have described the component-based approach of the PEPPHER framework that allows the specification of multiple implementation variants for a single functionality where the expected best variant for a given execution context is selected statically and/or dynamically. The PEPPHER composition tool developed during this work allows building applications from annotated components and thereby supports "PEPPHER-izing" both new and legacy applications. It provides a high level abstraction by automatically generating low-level C code for the runtime system. Above all, it combines features available in the runtime system with other mechanisms (e.g., smart containers, platform description language, XML etc.) to expose powerful selection and coordination knobs to the programmer at a higher level of abstraction.

"The wall is there. We probably won't have any more products without multicore processors [but] we see a lot of problems in parallel programming."

Alex Bachmutsky

Chapter 6

Global Composition Framework

This chapter describes the third approach that uses source-to-source code analysis for making effective composition decisions. We developed a Global Composition Framework (GCF) that provides performance-aware composition for different composition needs. The framework is generic both in performance models and target runtime systems and can support composition for applications with pre-calibrated performance models as well as for cases where they are learned during runtime. It also supports hybrid execution as well as global composition decisions considering more than one component call at a time.

The chapter is structured as follows: The GCF component model is described in Section 6.1 followed by description of the framework in Section 6.2. Section 6.3 presents an evaluation of the developed framework with different applications. Section 6.4 discusses global component composition for GPU-based systems and how the composition framework can help in making global decisions. Section 6.5 summarizes the work.

6.1 GCF Component Model

The GCF component model relies on pragma-based source code annotations for modeling meta-information about components. There exists naturally some overlap with the PEPPHER component model (see Chapter 5.1); however, GCF has a completely different annotation mechanism as well as conceptual differences (e.g., relating to performance models).

6.1.1 Components, interfaces and implementation variants

A GCF *component* consists of an *interface* that describes a functionality and multiple *implementations* or *implementation variants* of that interface that actually implement the functionality.

Both interface and implementations have attached meta-data, as shown in Figure 6.1, which provides extra information. In the PEPPHER component model, the meta-data is provided externally in the form of an XML descriptor file; this is non-intrusive to source code but requires some effort of writing and maintaining XML files. Moreover, mapping XML information to specific locations in the source code is highly vulnerable in case the source code gets modified. In the GCF component model, we use pragmas and comments to represent meta-information, which is more flexible and easy to write and maintain during code evolution. The component model is currently realized in the C/C++ language.

In C/C++, an interface is represented by a C/C++ function declaration that represents the functionality. If not specified otherwise, the function name becomes the name of the interface[1]. The meta-information for an interface includes:

- Access mode of each parameter (is it read, written or both inside the component).
- Relationship between different parameters (e.g., one parameter may describe the size of another parameter).

[1]Having one function per component interface is a special case of traditional component models that allow multiple functions per interface.

- Performance model associated with the interface.

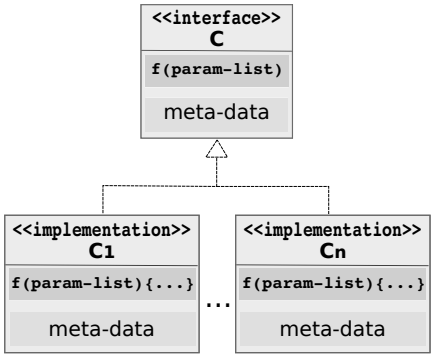


Figure 6.1: The GCF component model.

A component implementation constitutes an implementation of functionality promised by the component interface. Several implementations may exist for the same functionality (interface) by different algorithms or for different execution platforms; also, further component implementation variants may be generated automatically from a common source module, e.g., by special compiler transformations or by instantiating or binding tunable parameters. These variants differ by their resource requirements and performance behavior, and thereby become alternative choices for *composition* whenever the (interface) function is called. A component implementation targeting a particular execution platform assumes its operand data to be present in the memory address space associated with that particular execution platform. In order to prepare and guide variant selection, component implementations need to expose their relevant meta-data explicitly. A component implementation may have the following meta-data:

- The execution platform (e.g., CPU, GPU) it targets.
- The programming environment it uses (e.g., C++, OpenMP, CUDA, OpenCL).
- A performance model for this implementation variant.

Interface meta-data

<type> <funcName> **[/* perfModel(**<model-name> <file-name>**)**
[, name(<interface-name>**)] */]**
({ <param-decl> **[/* {** ***clause*** **[;] } */] });**

where ***clause*** is one of the following:

accessMode('read' | 'write' | 'readwrite'**)**
size({ *expr* **[,] })**
ld(*expr***)**
partition({ *expr* **[,] })**
usedInPerfModel('true' | 'false'**)**

expr is a valid C-expression.

Implementation variant meta-data

#pragma gcf variant [{ ***clause*** **[;] }]**
variant function definition

where ***clause*** is one of the following:

interface(<interface-name>**)**
target('cpu' | 'gpu'**)**
type('c' | 'cpp' | 'openmp' | 'cuda' | 'opencl'**)**
name(<variant-name>**)**
perfModel(<model-name> <file-name>**)**
validIf(*expr***)**

Figure 6.2: Syntax of source code annotations for interface and implementation meta-data.

Call meta-data

#pragma gcf execute [{ ***clause*** **[;] }]**
component call

where ***clause*** is one of the following:

interface(<interface-name>**)**
restrict('cpu' | 'gpu'**)**
hints({ ***phint*** **[,] })**

where
phint *is* <param-name> *op* <value-expr> *and*
op is a C comparison operator.

(a)

Data access annotations

#pragma gcf data [{ ***clause*** **[;] }]**
expression involving data

where ***clause*** is one of the following:

register(<id>**)**
access(<id>**,** ***accessType***)
unregister(<id>**)**

where ***accessType*** *is* 'read' | 'write' | 'readwrite'.

(b)

Figure 6.3: Syntax of source code annotations for component calls (in program source-code) and data accesses respectively.

- Specific requirements about the target platform and execution context that must be satisfied for its successful execution (selectability condititions).

Figure 6.2 shows the syntax of annotations for interface and implementation meta-data, using the standard EBNF (Extended Backus-Naur Form) notation [1]. We use source comments and pragmas to represent interface and implementation meta-data respectively.

The performance model can be specified at both interface and implementation-variant level. A performance model specified for an interface will be shared by all variants of that interface and will supersede individual performance models of any variant if they exist. As

we will see later, the ability to specify performance models at different levels gives us the flexibility to effectively address different application needs.

6.1.2 Composition Technique

Composition is the selection of a specific implementation variant for a call to component-provided functionality and the allocation of resources for its execution. There exist several similarities between the composition technique used here and the one used for PEPPHER components in Section 5.2. For example, the idea of making composition decisions statically and/or dynamically (called static and dynamic composition respectively) is the same in both techniques. Moreover, the ideas of having composition points on general-purpose execution units only, non-preemptive task execution as well as stateless tasks are the same as the ones presented earlier in Section 5.2. The major difference is in the handling of operand data passed to the component calls, as described below.

In the GCF model, the composition points (i.e., component calls locations in the program's source code) can also be annotated with extra information relevant to the call context. This includes information about constraints on the execution (e.g., execution *must* be carried out on a GPU) or on the call properties (e.g., limit on problem size). Figure 6.3a shows annotation syntax for a component call.

As mentioned earlier, a component implementation in the GCF component model assumes operand data to be present in the default memory associated with that execution platform. Whether and when to perform the possibly required operand data transfer is not hard-coded but exposed to the composition framework, as this is an important optimization aspect in a heterogeneous system containing multiple execution platforms with disjoint memory spaces. Factoring out the GPU data transfers from a GPU implementation yields two important benefits. First, it avoids the overhead of data transfers for each component invocation which can have a serious impact on performance [107]. The data transfers can thus be optimized across multiple component invocations. Secondly, it enables further optimizations in

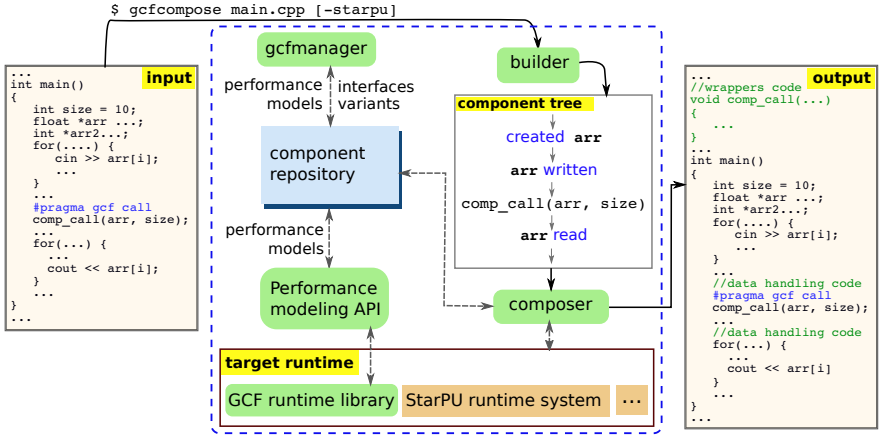


Figure 6.4: The GCF framework and its different parts and their interconnection is shown inside the dashed (blue) box. Moreover, the process of composition for an input program is shown along with the corresponding output modified source file.

data transfers (e.g., overlapping communication with computation) without requiring any changes in the implementation code.

This however requires runtime support for (in-time) data transfers to ensure that data is available in the right address space when an implementation gets called. The runtime support would *place* (i.e., schedule) data transfers (and other data handling code) at the right place to ensure that data is only communicated when really needed while ensuring data consistency and correctness of execution. In our case, we build a static data access analyzer to analyze the program data flow to find out where to *place* data handling code (more on this in the next section). As shown in Figure 6.3b, we also provide annotations to assist the data access analyzer in cases where data access may not be accurately found out using static analysis.

6.2 Global Composition Framework

Components written using the GCF component model can be composed for different composition needs by the GCF framework. Figure 6.4 shows different parts of our GCF framework along with their interaction. Following are the five major parts of the framework.

1. A component repository manager (`gcfmanager`) with support for both registering new components (interfaces, variants, performance models) as well as managing already registered components.

2. A component tree builder (`builder`) that, based on the ROSE compiler [182], first parses the application source code with one or more component calls and generates an AST (Abstract Syntax Tree). Afterwards, it builds a component tree by analyzing data accesses for component operand data in the generated AST.

3. A `composer` which parses the component tree along with the program control flow obtained via the ROSE compiler to generate the composition code. As a component call is made by interface name, a wrapper (proxy) function is generated to intercept the call and internally delegate the call to a certain implementation variant. The composer is generic in the underlying run-time system; currently, the composition code can be generated for both the GCF runtime library, which we have developed in this work, and for the StarPU runtime system [21].

4. A `performance modeling API` that specifies a generic and extensible interaction interface between the performance models and the runtime library.

5. The `GCF runtime library` that can do performance-aware composition by using the performance modeling API along with data handling for component operand data.

In the following we explain each part in more detail.

6.2.1 Componentization and Component Repository

For practical usage, a component model must be complemented with a mechanism for managing component information in an intuitive and scalable manner. We have provided the tool `gcfmanager` that can be used to manage the component repository. It has a simple interface, as shown in Listing 6.1, and can be used to register and manage both interfaces and implementation variants. It internally stores information about registered interface and implementation variants at a specific place specified at the installation time.

Registering a new interface is done by specifying a path to a file containing the interface declaration with annotations. For example, the following command registers an interface of a component from a SPH (Smoothed Particle Hydrodynamics) application:

```
$ gcfmanager -i sph_update_position.h
```

where the interface declaration in `sph_update_position.h` file looks as follows:

```
void sph_update_position(
 Particle *fluidArr /* accessMode(readwrite) size(arrSize)   */,
 int arrSize        /* accessMode(read)      usedInPerfModel */
);
```

Similarly an implementation file of the interface is registered in the following way:

```
$ gcfmanager -v sph_cuda_update_position.cu
```

where the implementation definition in the registered file looks as follows:

```
...
#pragma gcf variant target(gpu) \
                    interface(sph_update_position) \
                    type(cuda) \
                    perfModel(pm_sph_update_position_cuda.h)
void sph_update_position_cuda(Particle *fluidArr, int arrSize)
{
   ...
}
```

6.2.2 Component tree builder

As shown in Figure 6.4, the input program AST contains one or more calls to GCF components which operate on certain operand data. As implementation variants assume that operand data reside in their memory address space, certain data handling must be carried out to ensure that data is available at the right place at the right time. As data handling is carried out by the runtime system, we need to place data handling calls for the runtime system at appropriate places. We achieve this by analyzing the program control and data flow using the ROSE compiler. In fact, we have written a custom pass that provides this functionality; it works in the following manner.

```
1  $ gcfmanager -h
2  The global composition framework (GCF) manager
3  can be used to manage GCF component repository.
4
5  Usage:
6   gcfmanager [options]  <rose-options>
7
8  options*:
9   -h                Shows this message.
10  -i <filename>     Register a new interface.
11  -v <filename>     Register a new variant.
12  -u <inf-name> ... Update existing interface.
13  -U <var-name> ... Update existing variant.
14  -r <inf-name>     Remove existing interface.
15  -R <inf-name>     Remove existing variant.
16  -l                List all interfaces.
17  -L <inf-name>     List all interface variants.
18  -p <inf-name>     Shows interface' performance model.
19  -P <inf-name>     Shows variant' performance model.
20
21  * = option names are case-insensitive.
```

Listing 6.1: The `gcfmanager` options.

First, all component calls in the application code are collected and a list of all variables is maintained that are used (as operands) in one or more component calls. The only exception is scalar variables that are passed by value (e.g., the `size` variable in the input code in Figure 6.4)[2]. The data access analysis is then carried out for each

[2]We don't need any explicit data handling for these scalar passed-by-value

variable in the list. Our data access analysis is different from reaching definitions and liveness analysis [4]. We consider one variable at a time and traverse the AST to record data creation, accesses (read and/or written in non-component code) and usage of that variable in one or more component calls. We use the pointer alias analysis [115] built in the ROSE compiler framework to track aliases for each variable. Currently, data usage for array variables is recorded at the variable level. This means that, for example, a read access to a certain array element is considered a read operation on the complete array. This simplifies data flow analysis and data handling code that is inserted [156, 97]. We analyze data accesses in a recursive manner[3] for variables that are passed by reference or pointer to non-component function calls by tracing their usage inside that function body. The analysis carried out is conservative to ensure program correctness. Incase alias analysis fails [184], we introduce explicit data annotations as a fallback solution (see Figure 6.3b) that allow the user to assist data analysis for complex scenarios[4].

As per now, data usages are correctly tracked for regular C data types including pointers as well as any combination of C composite types using structures (e.g., structure with array member variables, array of structures etc.). For composite types, if a specific member variable is used inside a component call, the access can be separately tracked only for that member variable. For an array member variable, data usage is tracked at the array level, as said previously.

At this level, all these data operations and component calls are recorded in an AST tree representation which makes data analysis and optimizations simple to carry out. However, using the ROSE compiler, data operations recorded in the tree can be traced back to their location in the program control flow graph. In the next stage, this tree is complemented by the program control flow graph to insert

variables as a *copy* of original contents gets passed.

[3]We consider a recursive function as a special scenario to avoid combinatorial explosion of the solution space. General inter-procedural data-flow analysis for optimization of composition is an issue of future work.

[4]We have not encountered any such scenario yet in any application that we have ported to our framework.

data handling code at the appropriate places in the program control flow.

6.2.3 Composer

The composer takes the component tree generated by the component tree builder as input along with the program control flow graph. By mapping operations listed in the component tree to their source in the program control flow graph[5], it generates the actual composition code for a runtime system. Currently, we support composition using either our own GCF runtime library or using the StarPU runtime system. The choice of a runtime system is specified by the user as a simple switch[6] when doing the composition. Recursive components are not supported yet; partially because the StarPU [21] system does not support them.

At this stage, certain optimizations are carried out at the component tree (i.e. not in the actual source code) with the help of the program control flow graph to simplify the process of placing data handling code. For example, if we have two write accesses in the program's source-code to the same variable in the same (or nested) scope without any component call in between, they are considered as one write access. Similarly if we have one read access in the program's source-code followed by a write access to the same variable with no component call in between, we can consider them a single readwrite access instead.

Finding appropriate places for placing data handling code becomes quite tricky in a program with complex control flow. For instance, the data accesses and component calls can be inside conditional and/or loop constructs. The code must be placed in a way that it preserves the program correctness. The composer follows the following rules when placing the data handling code:

[5]This mapping is facilitated in the ROSE C++ compiler by keeping pointers to objects in the control flow graph.

[6]By default, the system generates composition code for our own GCF runtime library. The user can set the `-starpu` switch to generate code for the StarPU runtime system.

- Calls for registering[7] and unregistering a variable to the runtime system are placed at the same scope level. In case component calls and data accesses span over multiple non-nested scopes, calls can be placed at their immediate common parent scope.
- Placing data handling code inside a conditional statement is avoided, when possible, to simplify execution flow. When not possible, the code may need to be replicated across different control flows (e.g., if-else statement).
- Placing code inside loop constructs is avoided, when possible, to optimize data transfers. For example, if a variable is read inside a loop and no component call using that data is present inside the same (or nested) scope, the necessary code to ensure that data is available for the read purpose is placed just before the loop body to avoid the overhead of executing that code for each loop iteration.

The user can verify the generated/modified code (next step) by inspecting the output of our framework which is a modified source file with extra C/C++ source code added. In the following, we will briefly describe both runtime environments that we currently support for composition.

StarPU

StarPU [21] is a C-based unified runtime system for heterogeneous multicore platforms, especially for GPU-based systems[8]. It supports the explicit notion of CPU and GPU workers, a data management API as well as multiple scheduling and selection policies to decide which implementation to execute at which computing device for a given execution context. It can use the information about runtime workload of each worker to do dynamic load balancing and can effectively use multiple CPU and GPU devices in parallel. We use this capability of

[7] Registering a variable to the runtime system creates a unique data handle (with information about size, memory address etc.) for that data in the runtime system which can be used for controlling its state and data transfers.

[8] See Chapter 2.3 for a more detailed introduction to StarPU.

StarPU to support hybrid execution in our framework. Moreover, it has a performance-aware scheduling and selection policy which considers information about current system workload and data locality, and uses an estimate of data transfer cost as well as previous historical execution information to decide which implementation would be faster.

GCF runtime library

To get more control over runtime decisions, we have developed a simple and light-weight C++ based runtime library. Currently, the library is designed for doing *performance-aware implementation selection* where selection is mainly concerned with which implementation variant to use at which execution platform[9]. The main design goals of this runtime library are to be:

- *light-weight*: We want to reduce the runtime overhead to the minimum possible extent. Our library considers all CPUs as one combined worker, instead of creating one CPU worker for each CPU as StarPU does. Moreover, it does not support asynchronous component executions which alleviates burden of synchronization across different worker queues. When measuring the task overhead with 1 CPU and 1 CUDA worker, the overhead of our runtime library is around 0.02 microseconds which is more than two orders of magnitude lower than the overhead reported for StarPU with same workers configuration on that system (i.e., 5.95 microseconds, see Section 3.4.3 for more information).
- *configurable*: The library uses a configurable performance modeling API which allows it to be used for applications with different kinds of performance models. By specifying his/her own performance models (empirical, analytical etc.), the user can control implementation selection in an effective manner.

[9]In future, the GCF runtime library may be extended to support asynchronous component executions like StarPU.

There exists naturally some overlap in functionality offered by both runtimes; however there are several major differences. The GCF runtime library supports a configurable performance modeling API with multiple types of performance models available; it also allows a user to easily plug any new performance model with his/her components. Also, the GCF library mainly targets performance-aware implementation selection. This is more of an *offloading* decision, i.e., making a choice between using CPU (e.g., OpenMP) and GPU (e.g., CUDA) rather than simultaneous usage of both employed by StarPU, i.e., unlike StarPU, the GCF library does not currently support concurrent component executions on CPU and GPU devices in parallel.

6.2.4 Performance Modelling API

Considering that no single performance model can suit every application need, we have designed a generic API to handle interaction between a performance model and the runtime library, using C++ classes and polymorphism. Listing 6.2 shows the abstract performance model (`PerfModel`) interface which all concrete performance models must inherit. The performance-relevant call context properties are represented by an array of integers. This allows to model any number and kind of performance-relevant properties (e.g., problem size(s), sparsity structure, degree of sortedness of data etc.). Careful selection of performance-relevant properties is important to achieve good prediction accuracy. The `PerfModel` interface can be used to specify either of the following two kinds of performance models:

1. A *Per-variant performance model* targets a single implementation variant. In this kind of performance model, the prediction function `predict` returns a floating-point cost value. A common example of such kind of performance model is the historical execution based performance model used by the StarPU runtime system [20]. For now, we are concerned with performance in terms of execution time so the return value is predicted execution time. However, this could easily be used for other performance objectives such as energy consumption or code size.

2. A *Per-interface performance model* is common for all variants implementing an interface. This kind of model is at a higher granularity level as the performance model internally handles the choice between different implementations. In this case, the prediction function `predict` returns the expected best implementation name, managing the complete selection process internally. Many performance models can fit in here, including models based on decision tree, SVM and one presented in [132].

```
1 class PerfModel
2 {
3 /* for per variant performance models */
4   virtual double predict(int keys[], int numKeys);
5   virtual void addEntry(int keys[], int numKeys, double
        timeVal);
6
7 /* for per interface performance models */
8   virtual string predict(int keys[], int numKeys, double
        *commCost);
9   virtual void addEntry(int keys[], int numKeys, string
        impl, double timeVal);
10
11 /* common for both kinds */
12   virtual void close();
13 };
```

Listing 6.2: The `PerfModel` interface.

When comparing different variants for implementation selection, potential data transfer cost must also be considered besides the computation cost predicted for each variant. For a CUDA implementation variant, the potential data transfer cost is added to its computation cost if input data is not already present in the target GPU memory. The same should be done for CPU or OpenMP implementations if the most updated data resides in some GPU memory. The implementation selection can be configured in two modes. (1) In *pessimistic* mode it considers each component call as a standalone call. The data transfer cost for a variant is estimated assuming that data must be made available in the main memory after the component call. This for example could mean that, for a GPU variant, the cost of transferring input operands from main memory to GPU memory (if not present

already) as well as the cost for transferring modified data back to the main memory must be added to its computation cost. (2) In *optimistic* mode, data transfer cost is estimated by assuming that a component call is part of multiple component calls using that data. The data transfer cost, in this case, is only estimated for transferring (any) input operand data that is not already available in the target memory unit.

The *pessimistic* and *optimistic* modes are mainly suitable for an application with single and multiple component calls using a certain data respectively. The modes can be automatically determined by the framework in the following manner. By default, all component calls are executed in the optimistic mode, i.e., implementation selection decision for a component call is made assuming that output/modified data of that component call needs not to be transferred back to the main memory if the call is executed on a GPU. Transferring cost for modified data back to main-memory is only considered when there exists one or more program accesses, to data written/modified in that component call, on atleast one of the outgoing control flows with no component calls, in between, operating on that data.

For estimating the data transfer cost for operand data between different memory units, we have developed a simple library that estimates the transfer cost given the number of bytes, and source and destination memory unit. For each architecture, the library builds a simple model using latency ($latency$) and average cost of transferring one byte ($costPerByte$) determined by actual measurements on that architecture. Afterwards, it uses a simple formula for predicting data transfer cost between two memory units a,b for B bytes in direction $a \rightarrow b$:

$$time = latency[a][b] + costPerByte[a][b] * B. \tag{6.1}$$

The framework not only uses the performance model to do prediction (using the `predict` method) but also feeds actual measured cost value (execution time in our case) back to the performance model using the `addEntry` method. This enables runtime calibration where the performance model can learn/improve and predict during the same

program execution. One example is online learning where the performance model does not have any previous calibration data available and thus learns and predicts at runtime. The performance models can internally control switching between calibration and prediction mode. One key feature of the API is its genericity as it does not require an empirical model and can work for, e.g., analytical models such as [117], [131].

The process works as follows: The framework creates references to the performance model for each component interface/variant used inside the application during the initialization phase. The performance model can internally load previously calibrated information if any exists. During the actual program execution the performance model is used to predict the best variant as well as the actual execution data is fed back to the performance model. At termination, the framework calls the `close` method for each performance model. This allows a performance model to persist calibration information for future usage.

Currently, four performance models are implemented in our framework. All models can learn and predict both within a program execution or across different program executions. Each model predicts (expected) execution time or the name of the (expected) best implementation, for a given *performance context*, depending upon whether it is a per-variant or a per-interface performance model. A performance context is represented by certain values of performance-relevant properties such as problem sizes. The prediction is made for a given performance context. Moreover, a model is calibrated by feeding measured execution time of an implementation variant for a given performance context (see `addEntry` method in Listing 6.2).

Exact entry model

This performance model targets a single variant and is designed for numerical computations and other applications where few combinations of performance-relevant properties get repeated over different executions. The performance model is similar to the history-based performance model in StarPU [20]. The pseudo-code for this model

```
class HistoryExactPerfModel
{
 public:
  HistoryExactPerfModel(std::string fileName) {
   /* loads calibration data from file to m_estimates */
  }

  double predict(int *keys, int numKeys) {
    std::vector<int> key (keys, keys+numKeys);
    std::map<...>::iterator it = m_estimates.find(key);
    if(it != m_estimates.end()) // found an existing entry
     return it->second.first;
    else
     return -1; /* switch to calibration */
  }

  void addEntry(int *keys, int numKeys, double timeVal) {
   std::vector<int> key (keys, keys+numKeys);
   std::map<...>::iterator it = m_estimates.find(key);
   if(it != m_estimates.end()) // found an existing entry
    /* udpate existing entry avgExecTime and NEntries */
   else
    /* insert new entry to m_estimates */
  }

  void close() {
   /* save calibration data to file */
  }

 private:
  // map<key, pair<avgExecTime, NEntries> >
  std::map<std::vector<int>,std::pair<double,int> > m_estimates;
  ...
}
```

Listing 6.3: Pseudo-code for the Exact entry model.

implementation is shown in Listing 6.3. The model works in the following manner. At initialization, it loads (if any) calibrated information from previous executions. The information is retained in a STL map container with key representing performance context (`key`) and value as a pair of execution time (`avgExecTime`) averaged over number of entries (`NEntries`). During runtime, information about the execution time incurred by a variant for a given performance context is fed to the model via the `addEntry` method. If no entry with the same performance context (`key`) is found, it adds a new entry; otherwise, it

updates the existing entry's execution time estimate by averaging it with the new measured execution time. When predicting for a given performance context, it searches in already recorded data for an entry that has the same performance context. If an exact match is found, the execution time recorded for that entry is returned as (expected) execution time. Otherwise it switches to calibration mode to gather information for possible future prediction.

Euclidean normalized model

This performance model also targets a single variant and is more flexible (but usually less accurate [20]) than the previous model. It can predict for intermediate points[10] based on measured points using Euclidean distance as the decision criteria. The Euclidean distance between any two points a and b is the length of the line segment connecting them ($\overline{ab}$). Considering that it is also a per-variant performance model, it predicts the (expected) execution time of a variant for a given performance context (i.e. predicted context). It works in the following manner. Similar to the previous model, it keeps track of the average execution time incurred by a variant for different performance contexts. When predicting for a context, it searches for an existing performance context that is either same to the predicted context or has the shortest distance to the predicted context. It returns the measured execution time of that found context as (expected) time for the predicted context. The distance is calculated using the Euclidean distance; the performance relevant properties are normalized to the same scale before calculating the Euclidean distance.

Convex model

This model is at the interface level and thus internally manages selection between different variants. It is based on the model from Kicherer et al. [132]. The model aims for execution contexts with limited or no repetitions in performance relevant properties (i.e., almost every execution has a different value for performance relevant properties).

[10] A point represents a certain performance context.

When no calibration data is found, it starts calibration by trying out different variants in a round-robin fashion and collecting their execution time. When a reasonable amount of performance data (e.g., for five performance contexts per each variant) is collected, it builds the initial performance model in the following way: Considering that measured points for different variants would be possibly different, it takes the union of all recorded points for the different variants. It then interpolates measured points for each variant to get potential cost values for missing points in the union. The optimal variant at each point in the union is then recorded by comparing the potential cost values (along with data transfer costs) of each variant at that point. This model is further optimized by making a convexity assumption that if a single variant is found best for two neighboring points, then the same variant is predicted to be best for intermediate points [146].

Generic model

This model does not have any inherent semantics but rather relies on the user to provide an actual implementation. It can be used to plug in any performance model by specifying function pointers for the `predict`, `addEntry` and `close` methods. Two variations of this model are designed for plugging-in *per-interface* and *per-variant* performance models respectively.

6.3 Evaluation

For evaluation, we implemented seven applications from the RODINIA benchmark suite [53], two scientific kernels (*sgemm* and *spmv*), and several other applications from different domains (image processing, fluid dynamics etc.). Two evaluation platforms are used: **System A** with Xeon® E5520 CPUs running at 2.27GHz with 1 NVIDIA® C2050 GPU with L1/L2 cache support. **System B** with Xeon® X5550 CPUs running at 2.67GHz with a lower-end GPU (NVIDIA® C1060 GPU) is used for showing performance portability. The compilation is carried out using GCC (v4.6) and NVIDIA C (nvcc v0.2.1221 with CUDA 4.2) compiler for C/C++ and CUDA code respectively.

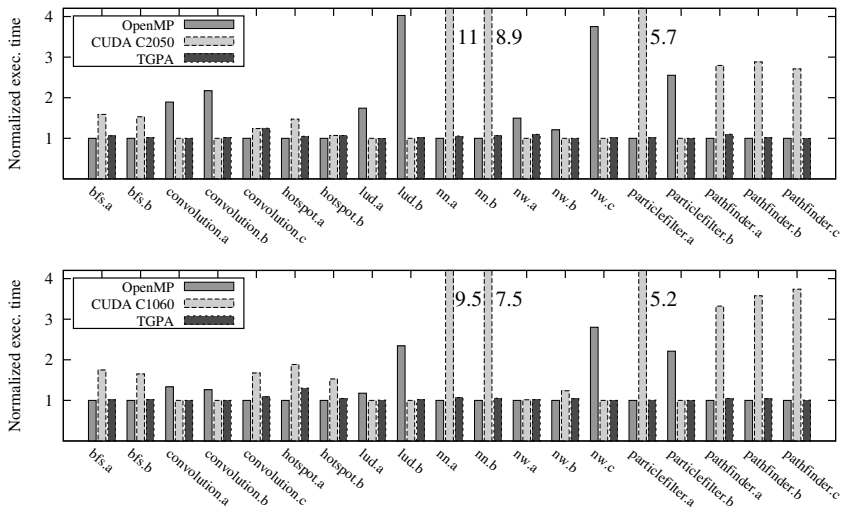


Figure 6.5: Execution times for different applications over multiple problem instances with CUDA, OpenMP and our tool-generated performance-aware code (TGPA) that uses the runtime library to do implementation selection on both platforms. The (exact entry) performance models were calibrated by earlier executions. The baseline is the faster of the two (OpenMP, CUDA).

6.3.1 Implementation selection

Figure 6.5 shows execution of several applications over multiple problem instances on two different GPU-based systems. The main decision here is to choose the implementation that performs better in a given execution context (application, architecture and problem instances etc.). As shown in the figure, different implementations can perform better for the same application on a given architecture but at different problem sizes (e.g., `pathfinder`, `convolution`) or the choice can be different between different architectures for the same application and problem sizes (e.g., `nw.b`). The performance-aware implementation selection effectively adjusts to these differences without requiring any modifications in the user code.

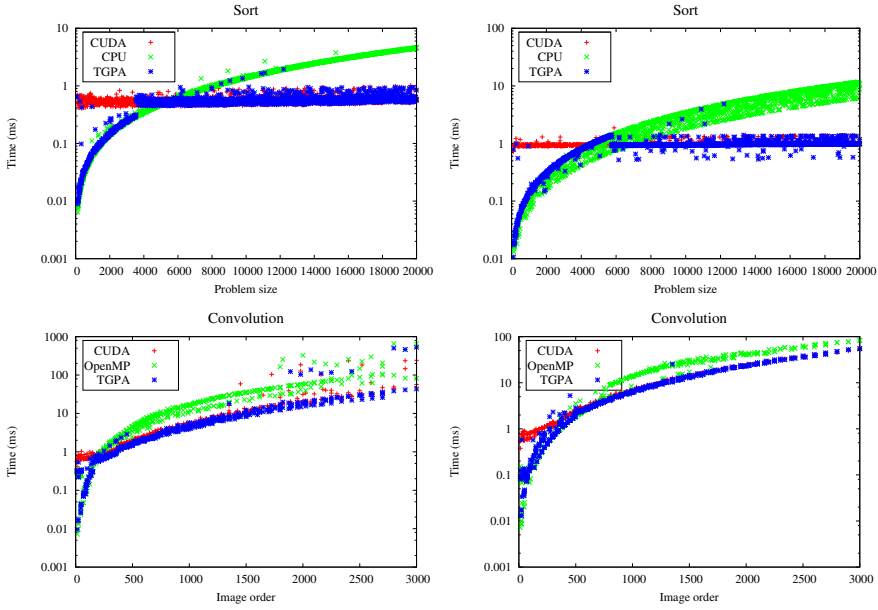


Figure 6.6: Online learning: Performance of our tool generated performance aware (TGPA) code with respect to CPU/OpenMP and CUDA implementation variants, shown with respect to problem sizes/image order for both (convolution, sort) applications on System A (left) and B (right) respectively.

Table 6.1: Online learning - sorting (2000 files).

	System A		System B	
	Time(ms)	Rel.Time	Time(ms)	Rel.Time
Quick sort CPU	3444.96	3.45	7271.36	4.20
Radix sort CUDA	1134.25	1.20	1961.77	1.13
TGPA	997.24	1.0	1732.73	1.0

Table 6.2: Online learning - image convolution (500 images)

	System A		System B	
	Time(ms)	Rel.Time	Time(ms)	Rel.Time
Convolution OpenMP	12276.79	2.75	6951.62	1.74
Convolution CUDA	4997.54	1.12	4598.67	1.15
TGPA	4467.44	1.0	3990.18	1.0

6.3.2 Online learning

There exist some applications where the same operation is applied multiple times over different operand data. Examples of such bulk operations include applying sorting, image filters, or compressions/decompressions over multiple files in a loop. In these applications, performance models can be learned at runtime [127, 132]. Our framework provides support for online feedback which allows performance models to be learned/improved during a program execution. Here, we show two applications (sorting, image convolution) to demonstrate online learning capabilities of our framework.

Table 6.1 shows online learning results for sorting 2000 files with random sizes between 100 and 20000 on both systems. Similarly, Table 6.2 shows online learning results for applying image convolution over 500 images with each dimension between 20 and 3000. The overhead of online learning is also included in the measurements. As the problem sizes are different and non-repetitive, we use the *convex model*. In both applications, we can see that our tool-generated performance aware code can effectively learn completely online with-

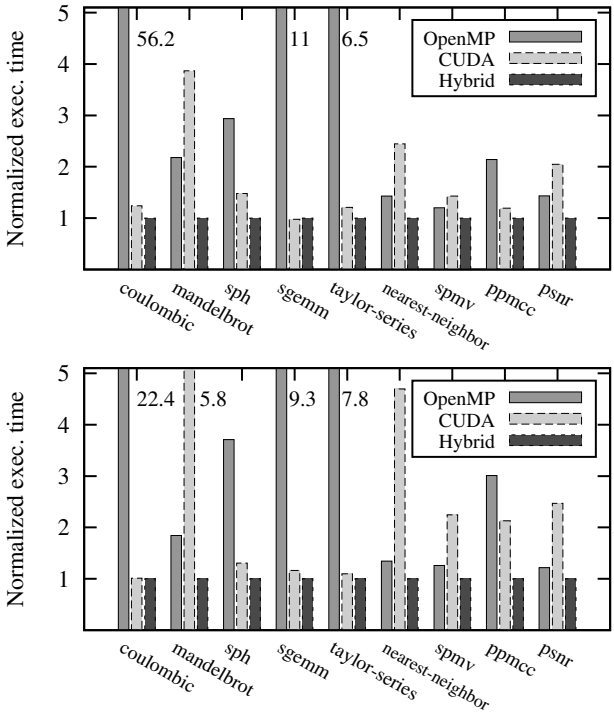


Figure 6.7: Performance benefit of hybrid execution by using our tool-generated performance aware (TGPA) code with respect to OpenMP and CUDA implementation variants. This feature is enabled using the StarPU runtime system. The baseline is hybrid execution time.

out any previous performance data being available on both systems and can perform up to 20% better than the best performing version. Figure 6.6 shows how the learner performs on both systems with respect to different problem and image sizes for convolution and sorting applications respectively.

6.3.3 Hybrid execution

An implementation variant of a component can internally specify parallelism, e.g., by using OpenMP and Pthreads for CPUs and CUDA

for GPUs. However, the implementation is still bound to either CPUs or GPUs. For certain computations, more parallelism can be spawned from a single component invocation by partitioning and dividing the work into several chunks that all can be processed concurrently, possibly on different devices by different implementation variants. This feature is implemented in our framework for data-parallel computations where the final result can be produced either by simple concatenation or simple reduction (using plus, max etc. operator) of intermediate output results produced by each sub-part (e.g., blocked matrix multiplication, dotproduct). When applicable, this feature can be used in a transparent manner (controlled using the `partition` option in the interface annotations, see Figure 6.2) for an application, as it does not require any modification in the user code and/or component implementations.

Figure 6.7 shows how this feature can help in achieving better performance, for different applications, than using any one of the OpenMP and CUDA implementation variants. This feature is enabled with the help of the StarPU runtime system as our runtime library currently does not support this feature yet[11]. More than 100% performance gain is achieved for some applications over the best variant on both systems. This (super-linear) speedup is achievable considering that dividing the computation work of a GPU with multicore CPUs not only divides the computation work but also reduces the communication overhead associated with GPU execution [107].

Besides giving performance improvements, this technique can also enable execution with much larger problem sizes than the one that can fit at once in e.g., a GPU memory (as the work is divided into smaller chunks with associated data).

6.4 Global composition

So far, we have mostly used the HEFT (Heterogeneous Earliest Finish Time) greedy scheduling and selection heuristic for making component

[11] Switching between composition using StarPU and our runtime library does not require *any* change in the user code. It is controlled by just a simple switch to our composition tool (`gcfcompose`).

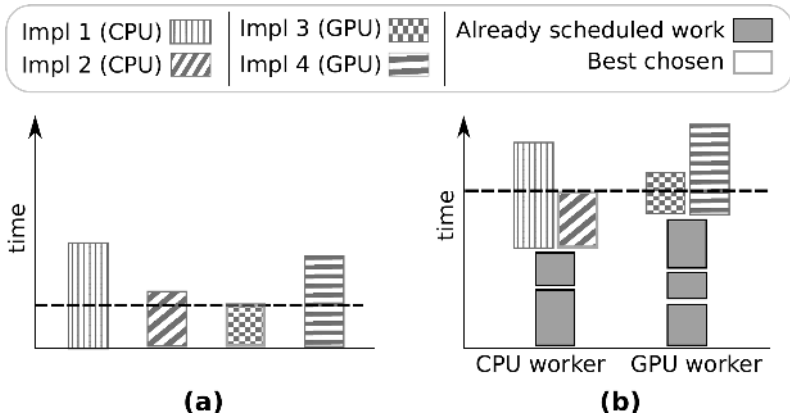


Figure 6.8: Greedy local implementation selection in case of (a) synchronous and (b) asynchronous component execution.

composition decisions at the runtime. In this section, we will discuss the HEFT heuristic and its limitations. Furthermore, we will propose four coordination constructs that can be used to model relationships between different component calls in a hierarchical manner and that can be used for making better composition decisions. In the end, we will describe a global *bulk* scheduling and selection heuristic and evaluate its effectiveness over the HEFT policy.

6.4.1 Greedy scheduling and selection using HEFT

The greedy HEFT scheduler considers one component call at a time when making implementation selection. The selection is normally done by estimating the execution time with the help of a performance model for each component implementation. The performance model could be trained empirically either offline [132] or generated online [20] by the runtime system; it could also be an analytical model supplied by the implementation writer [131]. There exist two variations of this greedy local scheduling and selection technique:

- *Synchronous component calls:* As shown in Figure 6.8(a), with synchronous component executions, each selection decision is made by considering which component implementation has the

expected shortest execution time for the current component call. All compute devices are assumed to be available for execution when making the decision. It is quite simple to implement; however, it cannot exploit parallel execution across different compute units in the system.

- *Asynchronous component calls:* With a runtime system allowing asynchronous component execution, the expected duration for each task (component call) scheduled on different workers is tracked throughout the program execution. As shown in Figure 6.8(b), for a new component call, the component implementation on a worker which results in shortest overall expected timespan is selected. This is a greedy heuristic that tries to reduce the overall timespan of the program execution and is employed by the runtime systems [181, 180, 20].

As GPU-based systems have separate memory address spaces, the operand data locality must be considered when making the selection decision, and the potential cost of data transfers needs to be included in case operand data is not already placed in the right memory unit. This requires runtime handing of operand data to track, transfer and optimize its usage across different memory address spaces, as done by all three approaches that we have proposed.

Performance remarks

As we have seen in the course of this thesis, the HEFT technique, although being a greedy heuristic, works quite well in practice. The main reason for its success is that this heuristic is *practical* and *efficient*. It is generic on the performance model and is normally realized using empirically tuned performance models. These models provide accurate performance estimation for computations with deterministic execution times. Also, the runtime overhead of this heuristic is small, usually up to a few micro seconds, as reported in this thesis and in many earlier works [21, 132]. Even performance models can be calibrated online considering low overhead of performance feedback which allows effective online learning [132, 181]. Results are reported in the

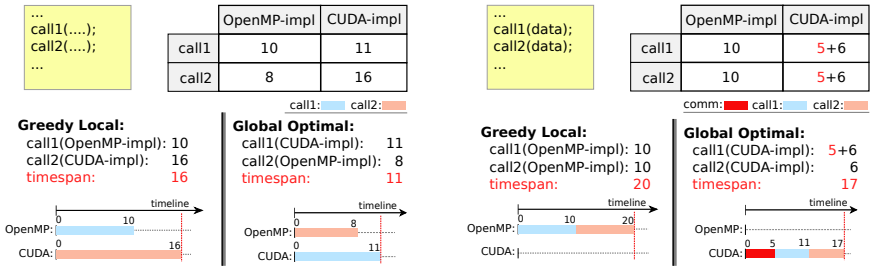


Figure 6.9: Two execution scenarios showing sub-optimal execution of the HEFT greedy scheduler in case of two (a) independent and (b) data-dependent component calls respectively.

literature for both small and medium size applications from different domains [132, 133, 21, 74, 181, 180].

One reason for its success is that many computational applications written for GPU-based systems (e.g., many applications in RODINIA benchmarks) internally have one or few computational kernels (components) normally executed multiple times. For such applications, this heuristic works quite well as reported by many [132, 133, 21, 74]. The main limitation of this heuristic can come for applications with multiple (types of) component calls. Even for many of these applications, this heuristic can still work reasonably well. In the following, we discuss two scenarios where it can perform sub-optimally.

Scheduling and selection for multiple component calls

The HEFT heuristic considering one component call at a time can result in sub-optimal decisions in presence of multiple component calls. Here we discuss two possible scenarios for GPU-based systems where this heuristic can give a sub-optimal global decision.

Independent component calls (*farm*) Two or more component calls that operate on different data can be executed concurrently[12].

[12]This scenario does not happen for approaches which enforce synchronous component execution (i.e. serialize their execution) and thus cannot exploit their in-

Runtime approaches supporting asynchronous component execution (e.g., our second and third approach) can schedule the component calls one after the other and can execute them concurrently. Figure 6.9a shows one such scenario. We suppose two independent component calls, each having an OpenMP and a CUDA implementation to choose from with the corresponding predicted execution time listed for each. For simplicity of demonstration, we assume no overhead of scheduling and selection in this case. The HEFT scheduler in the runtime system (e.g., StarPU) would first schedule the `call1` call on the OpenMP worker by making a greedy decision. The second call (`call2`) would consequently be scheduled on the CUDA worker as the GPU is idle. In this case, this would result in a sub-optimal global decision.

Data dependent computations (*sequence*) This scenario can happen with both synchronous and asynchronous component execution in case runtime data handling of operand data is in place. This is quite relevant to GPU-based systems that expose different memory address spaces with PCIe interconnect. Exploiting data locality and minimizing PCIe communication is important in these architectures to get good performance [107]. Figure 6.9b shows a simple scenario with two component calls constrained in a data dependency chain (i.e. no parallelism). We separately present the communication cost in this case to make the effect of data transfers clear. The corresponding predicted execution time is listed for different implementations and data transfer. Assuming that operand data is initially placed in the main memory and more component calls using this operand data can follow, we just consider data transfer costs from main memory to GPU worker device memory at the start of execution. The greedy scheduler would schedule both calls on the OpenMP worker one after the other whereas scheduling both calls on the CUDA worker would give better results in the end. We will later demonstrate this scenario with a real application along with a bulk scheduling and selection heuristic to make better decisions in such situations.

dependence.

6.4.2 Global Heuristics

In Section 6.4.1, we discussed limitations of HEFT greedy heuristic that considers one component call in isolation when doing scheduling and selection. To make better decisions, we need to consider all or a set of component calls together when making the decision. In the following, we describe a set of *call group* constructs that can represent coordination relationships between different component calls which can help in making better scheduling and selection decisions across different component calls. Specifically, we describe four call group constructs: *farm*, *sequence*, *loop* and *exclusive*. Each construct can be used to model a specific kind of relationship between different component calls. Considering that these four constructs can be nested inside each other in an arbitrary fashion, we use Δ for explaining their relationships where

$$\Delta := \langle farm \rangle \mid \langle sequence \rangle \mid \langle loop \rangle \mid \langle exclusive \rangle \mid ComponentCall$$

Now, we define the four constructs.

1. *farm*: Two or more Δs that are (data) independent of each other and can be executed in parallel if resources permit.

2. *sequence*: Two or more Δs that are constrained inside a data dependency chain (i.e. no parallelism), i.e., they must be executed in sequence.

3. *loop*: A construct that models repetitive execution of a Δ.

4. *exclusive*: A construct that models mutual exclusive execution of two or more Δs. A typical example is `if-else` conditional execution where component calls inside the `then` branch do not execute if the component calls inside the `else` branch execute and vice-versa.

For each construct, the control flow can only enter at the start node and can leave at the end node, i.e., single-entry single-exit. This means that these constructs do not model all possible component call relationships. These constructs can help in reducing complexity of making more informed scheduling and selection decisions and enable modular and hierarchical composition. Figure 6.10 shows the usage

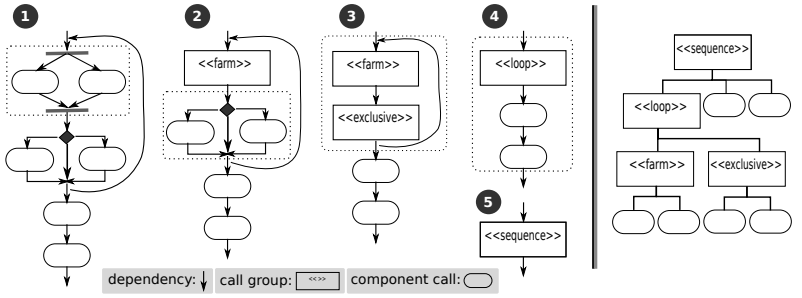


Figure 6.10: A component call graph representation using the grouping constructs in five steps (left) and corresponding group tree that is constructed (right).

of the grouping constructs to represent a component call graph using the four described constructs in a hierarchical manner.

```
1  #pragma gcf group sequence
2  {
3    #pragma gcf group loop
4    for(...)
5    {
6      #pragma gcf group farm
7      {
8        call1(...);
9        call2(...);
10     }
11
12     #pragma gcf group exclusive
13     {
14       if(..)
15         call3(...);
16       else
17         call4(...);
18     }
19   }
20   call5(...);
21   call6(...);
22 }
```

Listing 4: The group annotations for the call graph shown in Figure 6.10. Pragma annotations for component calls are omitted here for brevity.

In many cases, these coordination groups can be detected by source code static analysis for control and data flow for component calls (e.g., using the ROSE compiler in our framework). However, certain programming constructs or `#pragmas` can also be provided for the user to specify information about each group. Listing 4 shows group annotations (using `#pragmas`) for the component call graph shown in Figure 6.10(right). Extra information can be specified using such annotations; for example, a static estimate about the loop iterations count to make some decisions statically.

The four group constructs proposed above can model different scheduling and selection scenarios. As a first step, we will consider three base scheduling and selection scenarios that can be modeled by a farm, a sequence and a loop construct respectively. In future, more complex (hierarchical) composition scenarios with two or more such constructs nested/composed together can be considered.

Farm scheduling and selection

The `farm` group represents independence between its members (component calls and/or other group constructs) that can execute concurrently. In many cases, the farm members would reside adjacent to each other in the source code as we have in Listing 4, and possibly all can start execution at the same time as long as there are resources available. To improve over local greedy scheduling and selection using HEFT, the scheduler would need to consider all component calls inside the farm group at once while making the scheduling and selection decisions. Implementing a practical global heuristic for farm component calls could become difficult if component calls are apart and there exists some source code in between for which no reasoning about its execution time can be made. In some cases, the programmer can restructure the source code if the component calls are independent from that intermediate source code. In other more complex cases where other user code lies in between the component calls with no constraint on its timing, sticking with the local heuristic and scheduling and (implementation) selection for each call as it comes might prove a better solution in practice.

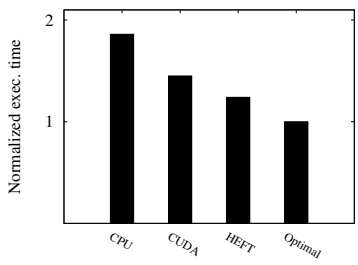


Figure 6.11: Execution of 2 independent tasks (DAXPY and DGEMM) on 1 CPU and 1 CUDA worker with each task having a CPU and a CUDA implementation. CPU and CUDA execution serialize task execution while heft makes sub-optimal decisions in this case just like shown in Figure 6.9a.

A practical scenario for global composition with a farm group is that all farm calls are ready for execution at the same time. In such situations, with a single component implementation available for each worker, finding an optimal schedule would require considering W^N scheduling and selection possibilities where W is the number of workers and N is the number of component calls in the farm group. For the simple and most typical case (e.g., the one shown in Figure 6.9a) with two workers (an OpenMP and a CUDA worker) and two component calls in the farm group, 4 (2^2) combinations need to be considered. However, in situations with more component calls/workers or with multiple implementations available for each worker (e.g., multiple OpenMP implementations), the overhead of considering all possible combinations could supersede the potential gain offered by the improved decision.

Example: Figure 6.11 demonstrates the scenario presented in Figure 6.9a on System A. Two independent tasks (DAXPY and DGEMM) with different execution complexity, each having one CPU and one CUDA implementation, are executed with a runtime system having 1 CPU and 1 CUDA worker. Execution of both tasks on either CPU or GPU worker serializes their execution[13] whereas *HEFT* and *Optimal* execution execute both task in parallel by running one task on the CPU worker and one on the CUDA worker. The optimal schedule for this experiment can be obtained by re-ordering both calls (see Figure 6.9a) and using HEFT afterwards.

[13]The runtime system used for the experiment does not support concurrent kernel execution on GPUs.

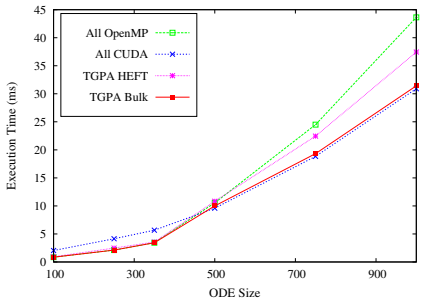


Figure 6.12: Comparison of tool-generated performance-aware (TGPA) composition using bulk scheduling and selection heuristic with HEFT and direct (OpenMP, CUDA) execution for ODE solver component calls [138] on System A.

Finding a practical heuristic: Although possibly sub-optimal, HEFT is still a practical solution for such a set of independent tasks. Finding a practical heuristic that can do a better job than HEFT in such circumstances (without assuming anything about coming tasks pattern or organization) is quite difficult. Some tasks may not be data-ready and deciding whether (and for how long?) to delay a task scheduling and selection decision hoping that some other task might become available (data-ready) is not a wise option as the waiting might supersede the potential gain (if even possible by reordering the ready tasks) in the end. Also, the actual execution time of two independent tasks, when executed together, might look quite different than when executed serially because of resource sharing such as memory bus, caches etc. Modeling such resource sharing for an arbitrary type/number of tasks on modern complex heterogeneous systems is a paramount task in itself. The above example (Figure 6.11) just shows that HEFT *can* be sub-optimal in such cases, but finding a *generic* and *practical* solution that can be implemented as a scheduling and selection heuristic, without knowing the whole task graph statically is rather infeasible. We therefore decided to use HEFT for the tasks in a farm group, and leave a more elaborated heuristic for farms to future work.

Bulk scheduling and selection

The *sequence* group represents a set of component calls (or other group constructs) constrained in a data dependency chain that shall be exe-

```
1  // 1D arrays: y, help1, yscal, temp, err_vector
2
3  #pragma gcf call interface(libsolve_copy)
4  libsolve_copy(y, help1, ode_size);
5  ...
6  #pragma gcf call interface(absaxpy)
7  absaxpy(y, help1, yscal, h, ode_size);
8  ...
9  #pragma gcf call interface(axpy)
10 axpy(help1, yscal, temp, h, ode_size);
11 ...
12 #pragma gcf call interface(libsolve_copy)
13 libsolve_copy(temp, y, ode_size);
14 ...
15 #pragma gcf call interface(scale)
16 scale(y, err_vector, h, ode_size);
17 ...
18 #pragma gcf call interface(absquotMax)
19 absquotMax(err_vector, yscal, &error_max, ode_size);
```

Listing 5: `Libsolve` source code with component calls.

cuted one after the other. As we have seen earlier, if the data operands reside in main memory, scheduling and selection considering one component call at a time might prefer CPU implementations over GPU implementations considering the extra communication cost for GPU data transfers. This could prove sub-optimal in case we have more (upcoming) component calls operating on the same operand data. By amortizing the communication cost over all component calls, going to GPU for the first component call (although locally sub-optimal) may prove an overall better choice in the end.

Listing 5 shows such a scenario with a code portion of a Runge-Kutta ODE Solver from the LibSolve library [138] containing multiple component calls with data dependency between them. The scheduler needs to consider such a group of component calls together when dispatching the first call. When dispatching them together, one choice could be to consider all possible call combinations; however, this could result in big overhead especially considering that the decision is done at runtime. For example, in Listing 5, for six component calls with each call having just two implementations (OpenMP, CUDA), there exist 64 (2^6) different possible combinations. Considering this and the fact that data transfers between different memory units are normally

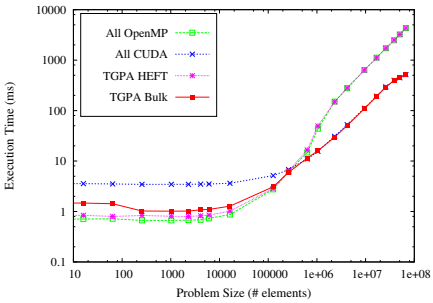


Figure 6.13: Comparison of tool-generated performance-aware (TGPA) composition using bulk scheduling and selection heuristic with HEFT and direct (OpenMP, CUDA) execution for the syntactic application on System A.

expensive, we propose *bulk performance-aware composition* for such a *sequence* (or a *loop* with one or more component calls in sequence) group that only considers combinations possible within implementation variants with same memory address space.

In this case, as OpenMP and CUDA variants have different address spaces, there exist only two choices; either execute everything on 1) OpenMP or 2) CUDA. By using bulk composition, as shown in Figure 6.12, we were able to perform better than the pure local scheduling and selection policy that considers a single component call at a time. The figure also shows that over different problem sizes, the choice between using OpenMP and CUDA also changes, which the bulk composition was effectively able to adjust. As our GCF framework can do program analysis using the ROSE source-to-source compiler, we can implement the bulk scheduler by analyzing the program source code.

```
1   ...
2   #pragma gcf group loop
3   for(....) {
4      #pragma gcf call interface(daxpy)
5      daxpy(a, x, y, N);
6   }
7   ...
```

Listing 6: Loop pseudo-code.

A similar case is for a *loop* group with a sequence of one or more component calls that are repeated multiple times. Amortizing communication overhead over multiple loop iterations may change the

selection of the best variant. Making a greedy choice considering just a single component call might result in overall bad performance. Listing 6 shows pseudo-code of a BLAS Level 1 DAXPY component call (with an OpenMP and a CUDA implementation) executing inside a loop over the same data. Considering one component call at a time with operand data initially placed in the main memory, the HEFT scheduler might prefer an OpenMP implementation over a CUDA implementation if the data transfer overhead to GPU memory supersedes the potential computational advantage of GPU execution. The bulk scheduler can make a better decision by considering data transfer cost amortized over multiple executions of the component call. As the scheduling and selection decision is made at runtime, the loop iteration count needs not be necessarily known statically.

Figure 6.13 shows the execution of the code shown in Listing 6 with bulk scheduling and selection, HEFT as well as direct execution using the OpenMP and CUDA implementations. The bulk scheduler makes a more informed decision by amortizing communication overhead for GPU execution over multiple loop iterations[14]. The overhead of making a bulk scheduling and selection decision at runtime is included in the measurements and it proves to be negligible even for smaller problem sizes.

Bulk heuristic implementation: As our GCF framework uses ROSE source-to-source compiler for program analysis, it can find out such set of component calls automatically. The bulk heuristic is implemented as a runtime superset of HEFT. This means that, e.g., for 2 workers (1 CPU, 1 GPU) with component calls having implementation for both workers, we compare whether it would be better to schedule all of those calls on CPU or GPU over HEFT. Figure 6.14 shows a pseudo-code snippet for bulk composition that can be generated by our GCF framework for three component calls constrained in a data-dependency chain. Three main choices are considered; 1) executing all calls on CPU, 2) executing all calls on GPU and 3) execution using HEFT. The actual decision is made at runtime by considering:

[14]The loop iteration count was set to 50 for this experiment.

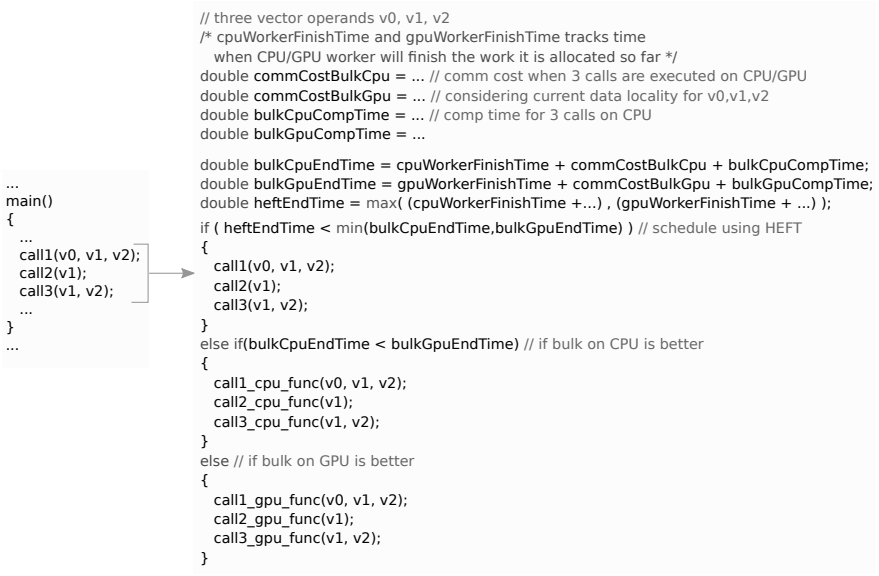


Figure 6.14: An example pseudo-code and the code generated for bulk scheduling and selection by our GCF framework.

- potential data transfer costs calculated based on current data locality for calls operands. Data transfer costs are estimated using equation (6.1).
- computation cost for three calls on CPU or GPU worker.
- current work allocated to both CPU and GPU workers (see Figure 6.8).

6.5 Summary

In this chapter, we have presented a component model and a global composition framework that addresses the implementation selection problem in an integrated manner. It uses the ROSE source-to-source compiler and can do effective composition decisions by doing program (data-flow, control-flow, call-graph etc.) analysis. We have shown its

effectiveness for GPU-based systems with the help of several applications, for doing:

- automatic implementation selection for a given execution context for cases with pre-calibrated performance models to one with pure online learning;
- hybrid execution decisions for data parallel computations, giving up to two times performance improvement for some applications over the best performing variant; and
- bulk composition for multiple component calls constrained inside a data dependency chain.

Furthermore, we have described, with two scenarios, how the global composition decisions can look different than what is determined by the state-of-the-art HEFT scheduler used today. Four group constructs are proposed to model relationship between different component calls that can later be used for guiding the composition decisions.

"... Out of clutter find simplicity. From discord find harmony..."

Albert Einstein

Chapter 7

Discussion and Conclusion

In this thesis, we have studied the problem of efficiently programming GPU-based systems at a high-level of abstraction by marking computations as components with possibly multiple implementations being available that can come from different sources. The idea of optimized component composition is to manage the static and/or dynamic selection of an appropriate implementation for each component call in the program considering the call context, data state and resource allocation/requirements, guided by an objective function such as execution time.

7.1 Three approaches

We have presented, implemented and evaluated three different techniques in the course of this thesis work.

1. *Approach 1*: A C++ skeleton library (SkePU) that provides several data-parallel and one task-parallel skeletons (generic components) with multiple implementations for each skeleton. The application writer needs to choose the appropriate skeleton(s) for a computation and call it using smart containers holding the required operand data. The library can automatically handle the implementation selection, behind the scene, to provide good performance on a given architecture in a portable manner.

2. *Approach 2*: An XML-based composition framework that uses XML descriptors to get extra meta-data for component interfaces and component implementations, which is used to make the component composition decisions for component calls in a program. It can do composition decisions statically where possible as well as generates optimized code for doing them at runtime using the StarPU runtime system. Moreover, it uses smart containers to provide a high level of abstraction while leveraging inter-component parallelism. Using XML, smart containers and features offered in the runtime system, it provides a powerful high-level mechanism for the user to affect component composition decisions (conditional composition).

3. *Approach 3*: A global composition framework (GCF) that uses static program analysis and powerful code generation and transformation capabilities offered by a source-to-source compiler framework for making effective component composition decisions. It provides implementation selection and hybrid execution capabilities with a generic performance modeling API supporting different kinds of performance models. Moreover, by program analysis, it can make composition decisions considering more than one component call at a time, outperforming the state-of-the-art HEFT scheduler, for component calls constrained inside a data-dependency chain.

Unsurprisingly, these three approaches have a lot in common as they all employ the idea of optimized component composition. However, there exist some critical differences too. In the following, we will discuss both similarities and differences between these approaches.

7.1.1 Similarities

The major similarity is that all three approaches leverage the idea of having multiple implementations for each computation (called component or skeleton). Also, they all provide automatic composition capabilities that can be carried out without asking the application writer as he/she can make calls by interface. All approaches provide

Table 7.1: Comparison of the three approaches. SC stands for Smart Containers.

	Approach 1 (SkePU)	**Approach 2 (PEPPHER)**	**Approach 3 (GCF)**
dynamic selection and scheduling	yes	yes	yes
hybrid execution	yes	yes	yes
online learning	no	no	yes
global composition	no	no	yes
general	no	yes	yes
intrusive	yes	no	yes
asynchronous	for SC operands	for SC operands	for all
requires metadata	no	yes	yes

dynamic implementation selection (possibly in combination with dynamic scheduling) and hybrid execution capabilities. Because of these similarities, for many applications, any one of these three techniques can be used to make the decision.

7.1.2 Differences

Besides lots of similarities, as shown in Table 7.1, there exist some inherent differences between the approaches that affect their capabilities. As Approach 1 uses skeleton programming, it offers ready-to-use generic components with implementations made available as part of the library. This gives the user the advantage of not having to bother about providing implementation sources and requires less programming effort; however, it sacrifices on generality as it works only for those computations that can be written using the available set of skeletons. On the other hand, Approaches 2 and 3 are more general as they support any computation whose state can be modelled via its input and output operands.

Approach 2 uses XML to model metadata as the idea is not to modify the application source code. It has the advantage of being non-intrusive to the source code as metadata is represented in a separate file, although maintaining these XML files during code evolution could be a tedious task without special tool support. Also, Approach 2

does not use any program analysis which limits its applicability for making more complex global composition decisions. Last, usage of smart containers for operand data is required to achieve asynchronous component execution and inter-component parallelism.

In Approach 3, we can handle both simple and complex composition scenarios in an integrated manner as it employs source-to-source code analysis and transformation capabilities. It represents metadata for programs using pragmas which provide more flexibility with code maintenance and evolution but are intrusive to the program source code. Asynchronous component executions and inter-component parallelism can be leveraged even for operands of normal C datatypes, by tracking their accesses and injecting data handling code at appropriate places in the program control flow. Last, composition decisions considering multiple component calls together are also possible in this approach.

7.2 Concluding remarks

Enabling efficient and productive programming of heterogeneous GPU-based systems is a hard problem which is far from being solved. In this thesis, we took an integrated approach using constructs for high-level abstraction (skeletons, components, smart containers), program analysis/transformations as well as code generation for a runtime system.

We have seen that there exists a major difference between the performance behavior of CPU and GPU devices and that choosing an appropriate implementation to run on a particular computation resource is of pivotal importance. We used performance models that are based on empirical executions, along with data locality and system workload to decide which implementation to execute on which device. The choice is proved to be quite dynamic as it could change from application to application, system to system and even between different call contexts (e.g., problem sizes) in the same application on a given system. Furthermore, considering CPUs as a first-class computational resource rather than a controller assigning work to GPUs

can give significant performance improvements for many applications.

Although it is completely fine to use analytical performance models for decision making, they often prove too complex to build in practice. Making an analytical model that can model performance effects of chosen algorithm, problem sizes, compiler transformations/optimizations, and micro-architectural features of the underlying system is a hard task. Furthermore, often, this effort cannot be re-used when porting to a new system as it might have a different generation of CPUs (or GPU) with different micro-architectural features. The situation might change in future with architectures and compilers getting more stabilized in their offerings.

Thinking of our three approaches, it is difficult to favour one approach over another as each one has its strong points as shown in Table 7.1. One recommendation could be to use SkePU (Approach 1) when one can model a computation with the existing set of skeletons as it requires no extra effort. Otherwise, Approach 2 is useful in cases where changes in the application source code are either not possible or undesirable. This could happen when, e.g., source code of components is not available for metadata annotations. However, if one has access to the source code and changes in the existing program code are possible, Approach 3 with pragma annotations could be a good idea. Approach 3 is most powerful in terms of its global composition capabilities as it can deduce parallelism across different component calls automatically and can do composition decisions considering control and data flow relationships between component calls.

In all approaches, the achievable performance is determined by the implementations available for that computation. For many computations, one can find good implementations either in the form of libraries or existing source code by other programmers. We hope that, with time, more people will adopt component programming techniques to design their applications, resulting in more and more components and implementations becoming readily available. The benefits of the component-based approach reach beyond programming and performance issues. For example, it can reduce system complexity by modular design, increase system testability as well as increase the software re-use potential.

"Discovery consists of seeing what everybody has seen and thinking what nobody has thought."

Albert Szent-Györgyi

Chapter 8

Related work

There exists a large body of work about skeleton programming, component frameworks and programming approaches for GPU-based systems. In the following, we will discuss the related work for the above mentioned areas.

8.1 Skeleton programming

Although formulated by Cole [56, 57], early work on identification of high-level constructs for expressing basic data and task parallel patterns in parallel applications, has been reported by several researchers during the same time period (i.e., in the late 80s). This includes initial formulations of certain task-parallel skeletons such as *farm* [37, 195], *pipeline* [124], *divide and conquer* [130], and *branch and bound* [100] and some early work on data-parallel skeletons [139] such as scan [38]. Initially, the main focus of skeleton programming was on achieving high abstraction [175], either by usage of activity graphs [59] or by a high-level coordination language [71, 72, 40, 26, 81, 108, 68].

8.1.1 Skeleton frameworks for multicore CPU systems and MPI-clusters

Functional approaches Many skeleton programming approaches used the functional programming paradigm, including Higher-order Divide-and-Conquer language (HDC) [112], Eden [150], Concurrent

Clean [118], ML [158], OCamlP3l [55], Skipper [200], and the Hope [70] language. Although achieving higher level of abstraction, all these approaches share a common drawback of functional programming, i.e., trading performance for achieving an easier and shorter programming style. For parallel multicore and cluster architectures, this performance degradation could be up to an order of magnitude, as shown by Loidl et al. [149].

C/C++-based approaches There exist several skeleton programming frameworks that are built using C and C++, including SkeTo, eSkel, Quaff, MALLBA and FastFlow. SkeTo [126] provides data parallel skeletons for distributed data structures such as lists (arrays) [201], matrices (2D arrays), and trees. As a C++ template library, it has limited support for *fusing* consecutive skeleton calls into one call, which can improve performance when applicable. eSkel [58, 34, 33] is a skeleton library that exposes certain parallelism concerns related to distribution directly to the application programmer. It provides a *pipeline* skeleton and a *deal* skeleton where the latter is a variation of the pipeline skeleton with stage replication and fusion capabilities. eSkel focuses on nesting of skeletons and provides two nesting modes, *transient* mode where skeleton objects are destroyed immediately after invocation, and a more lasting *persistent* mode. Quaff [94] mainly provides *farm* and *pipeline* skeletons. It reduces the runtime overhead by performing most of the skeleton instantiations and optimizations at compile-time, using the C++ template meta-programming [2]. MALLBA [6] focuses on the combinatorial optimization problem domain. It defines skeletons such as Dynamic Programming, Simulated Annealing, Tabu Search, and Genetic Algorithms. FastFlow [9] is a C++ template library that targets execution on shared-memory multicore CPUs. It uses lock-free (and memory fence free) Single-Producer-Single-Consumer (SPSC) FIFO queues to implement producer-consumer communication mechanisms with very low synchronization overhead. Furthermore, FastFlow uses Monte Carlo search to tune the values for certain parameters of skeleton implementations [60]. However, tuning in FastFlow is about finding the values for the tuning parameters rather than implementation

selection as we do in our case.

Java-based approaches Calcium [48], JaSkel [99], Lithium [10], Muskel [8], CO2P3S [152] and Skandium [142] are all implemented in Java. These frameworks mainly differ in their skeleton types and the skeleton composition techniques they imply. Most of them have support for distributed memory systems while Skandium is mainly designed for multicore CPUs with shared memory support.

Comparison All skeleton approaches listed above are either written for shared multicore CPU execution or for execution on a standard homogeneous MPI cluster. Our work on SkePU mainly targets single-node GPU-based heterogeneous systems. This difference in target architecture leads to radical differences in library design as for multicore execution, communication is comparatively less expensive (and often implicit) between CPU cores and the granularity of both computation and communication can be fine. In our case, as GPUs are suitable for more compute-intensive applications, SkePU skeletons are designed to leverage these massive computational capabilities while minimizing the communication overhead. Moreover, presence of different kinds of compute devices (CPU, GPU) as well as a distributed memory address space leads to major differences in features such as support for algorithmic selection, hybrid execution and communication optimization (lazy memory copying).

8.1.2 Skeleton programming for GPU-based systems

There exist some related skeleton programming solutions in the GPU computing domain such as SkelCL, Muesli and Marrow.

SkelCL [197] is a skeleton library implemented using OpenCL. Initially, it implemented a one-dimensional vector datatype and four data parallel skeletons named Map (for one input operand), Zip, Reduce and Scan where Zip is basically a Map operation with two input operands. Recently, support for a two-dimensional matrix datatype as well as two new data-parallel skeletons (MapOverlap and Allpairs) has been added [196]. The SkelCL vector and matrix containers provide

memory management capabilities like SkePU containers. However, in SkelCL the data distribution for containers is explicitly specified by the application programmer whereas in SkePU, it is transparent. This gives freedom to the SkePU framework on the actual execution of a skeleton call, including selection of a skeleton implementation to use and the ability to partition the work between different devices in a transparent way. Furthermore, SkelCL has no support for hybrid execution and performance-aware implementation selection.

The Muesli skeleton library, originally designed for MPI/OpenMP execution [54], has been recently ported for GPU execution [90]. However, it does not support skeletons with more global access patterns such as MapOverlap and MapArray which are essential for implementing a large number of applications ranging from N-body simulation to Conjugate Gradient solver. Support for OpenCL, performance-aware implementation selection and hybrid execution are major features that distinguish our work on SkePU from Muesli.

Marrow [154] is a skeleton programming framework for systems containing a single GPU using OpenCL. It provides data (map) and task parallel (stream, pipeline) skeletons that can be composed together to model complex computations. However, it focuses on GPU execution only (i.e., no execution on multicore CPUs) and exposes concurrency and synchronization issues to the programmer.

Support for GPU execution has recently been added [105] in the FastFlow library for pipeline and farm skeletons using OpenCL. Danelutto and Torquati [69] proposed a set of parallel building blocks for modeling and implementation of parallel frameworks that are applied to FastFlow [7]. Moreover, Serban et al. [192] implemented support in FastFlow for distributing map and reduce computation work across multicore CPUs and one GPU using an analytical performance model.

Comparison

In Table 8.1, we compare SkePU with other skeleton approaches that support execution on GPUs using either CUDA and OpenCL. SkelCL and Marrow are both written in OpenCL and mainly target GPU execution. SkePU is the only library that provides both CUDA and

Table 8.1: Comparison of SkePU with other skeleton frameworks supporting GPU execution, based on different capabilities. NA stands for *Not Applicable*

	SkePU	**SkelCL**	**Muesli**	**Marrow**	**FastFlow**
CPU implementations	C++/OpenMP	-	C++/OpenMP	-	C++/OpenMP
GPU implementations	CUDA/OpenCL	OpenCL	CUDA	OpenCL	(CUDA/OpenCL)
Automatic implementation selection	yes	NA	no	NA	limited
Hybrid (CPU/GPU) execution	yes	NA	no	NA	yes
Data-parallel skeletons	yes	yes	yes	yes	no
Task-parallel skeletons	yes	no	yes	yes	yes
Multi-GPU support	yes	yes	yes	no	no
Hide concurrency	yes	yes	yes	no	no
Hide data distribution	yes	no	yes	yes	yes
Skeleton nesting	limited	no	no	yes	yes
MPI support	yes	no	yes	no	no

OpenCL implementations of its skeletons. FastFlow allows the user to write arbitrary CUDA and OpenCL code inside skeletons that can be executed on GPUs, i.e., it does not provide as such any GPU implementations of its skeletons. Furthermore, SkePU is the only library that supports automatic implementation selection between different skeleton implementations as well as hybrid execution of a single data-parallel computation across multiple CPU and/or GPU devices. Some initial support for implementation selection has been implemented in FastFlow to decide whether to use the CPU or (a user-provided) OpenCL implementation for a farm worker [105]. This is done with a simple formula where a new item of work is assigned to a GPU having the least number of assigned (not finished) kernels, assuming that each kernel takes the same execution time. Moreover, in FastFlow, no explicit data-parallel skeletons exist and stream parallelism is mainly expressed using different configurations of farm, pipeline and loop skeletons.

Marrow support both data- and task-parallel skeletons but is limited to single GPU execution. It also exposes concurrency to the application programmer. SkelCL supports only data-parallel skeletons and exposes data distribution to the programmer; however, it can run on multiple GPUs. In FastFlow, the application programmer needs to write CUDA and/or OpenCL code which exposes concurrency and parallelism issues to the programmer.

SkePU hides both concurrency and data distribution from the application programmer. When running SkePU skeleton programs on top of StarPU, skeleton calls are executed asynchronously; however, asynchrony is implicitly handled within the SkePU containers that implement synchronization logic for completing all previously issued skeleton calls, operating on that data, before allowing program accesses to that data. SkePU also supports one-level nesting of data-parallel skeletons inside the farm skeleton. Marrow and FastFlow support arbitrary level nesting of their skeletons. Last but not the least, SkePU and Muesli supports execution on MPI clusters (see [153] for MPI support in SkePU).

8.1.3 Other approaches for GPU-based systems

Thrust [116] is a C++ template library for CUDA that implements functionality like transform (map), reduction, prefix-sum (scan), sorting etc. However, Thrust does not support usage of multiple GPUs for a single library call. *CUDPP* [111] is a library of data-parallel algorithm primitives such as parallel prefix-sum, parallel sort and parallel reduction. It does not however provide higher-order functions which can take any user defined function as an input.

SPOC [41] is a runtime library that enables CUDA and OpenCL programming for the OCaml [113] programming language. It provides a vector data-type to abstract memory management for GPU computations while internally using garbage collection mechanism available in OCaml.

Nugteren et al. [166] introduce a classification of affine loop nests (called algorithmic species) which their tool [165] can automatically identify in a program and can then generate optimized code for execution on GPU. A similar tool for identifying patterns from existing sequential programs that can be represented by SkePU skeleton calls, has been developed by Sarvestani et al. [190].

Vinas et al. [204] propose a high-level programming library with OpenCL code generation facility whereas KFusion [134] can help fusing multiple OpenCL kernel calls for performance reasons. Benkner et al. [31] propose high level constructs (in the form of pragmas) for modeling pipeline patterns in an application; underneath, they generate StarPU code for making decisions at runtime.

8.2 Programming approaches for GPU-based systems

In the last few years, a great deal of work has been published regarding programming solutions for GPU-based heterogeneous systems. It ranges from automatic code generation for execution on GPUs from sequential (or parallel) CPU source code to proposals of new unified high-level abstractions/programming models/environments to program such systems.

8.2.1 Unified programming abstractions

Many programming approaches aim to provide unified programming abstraction for multicore and heterogeneous systems. PetaBricks [13] is an implicitly parallel language and compiler framework that uses empirical auto-tuning for finding algorithmic configurations to get good performance on a variety of architectures. The programming model relies on the programmer providing multiple implementations of each algorithm at possibly different granularity levels and how they can be composed together in the application. The `transform` programming construct models a computation with given input and output operands, internally having possibly multiple `rules` which specify computations on (part of) input data to generate (part of) the final output. The auto-tuner can then explore the search space of possible configurations in an effective manner to determine the best configuration for different execution contexts. The output of the auto-tuner is an application configuration file specifying different configurations and transitions points between them; the configuration file is then used by the runtime library to make choices at runtime. With recent work [178], PetaBricks programs can be executed on GPU-based systems by generating OpenCL code.

There exist some similaritites between our work and PetaBricks as we also use runtime systems to make final decisions, and an auto-tuner is used (e.g., in SkePU) to determine transition points between different configurations. However, there exist some major differences. In our work, we do not propose any new programming model (or programming constructs) but rather rely on existing well-established programming models (OpenMP, CUDA). Secondly, we target mainly the implementation selection problem between different computations and not the tuning of tunable parameters inside each implementation; this is a complementary feature to our approach. Furthermore, component implementations in our case can specify internal explicit parallelism (using OpenMP, CUDA etc.) whereas rules in PetaBricks are written sequentially where parallelism is automatically devised by the PetaBricks compiler based on data dependency analysis.

The Merge framework [148] targets a variety of heterogeneous ar-

chitectures, while focusing on MapReduce [76] as a unified, high-level programming model. Like PetaBricks, it provides a unified programming abstraction while providing data-parallel programming abstractions. The performance is achieved by choosing from potentially many implementations for each computation. The Merge framework provides a predicate mechanism that is used to specify the target platform and restrictions on the execution context for each implementation; this is somewhat similar to the conditional composition capability that we have developed in our PEPPHER composition tool. However, in our work, we consider other computation patterns beside MapReduce. Moreover, in Merge, selection between different implementations is done by issuing work dynamically to the processor available (i.e., a dynamic workload balancing mechanism) while favoring specialized accelerator implementations over sequential CPU implementations. This is quite different to the our selection mechanism based on performance models which is more flexible and generic in its applicability.

In [207], Wang et al. propose EXOCHI, an integrated programming environment for heterogeneous architectures that offers a POSIX shared virtual memory programming abstraction. It extends earlier work on the Multiple Instruction Stream Processor (MISP) [110] by implementing support for heterogeneous accelerators as sequencers, which are similar to processor cores in an SMP. These sequencers are exposed to the application where user-level threads (sequences of instructions) called *shreds* run on these sequencers. The programming model is an extension of C++ and OpenMP where (domain-specific) portions of code, to be executed on a specific accelerator sequencer, are embedded/inlined inside OpenMP parallel blocks. The EXOCHI compiler injects calls to the runtime system and then generates a single fat binary consisting of executable code sections corresponding to the different accelerator-specific ISAs. During execution, the runtime can spread parallel computation across the heterogeneous cores. The evalution in [207] is carried out on a specially built MISP processor research prototype as the MISP architecture is not commercially implemented yet. Comparing to our work, EXOCHI provides offloading of OpenMP parallel regions on accelerators, i.e., it does not support

multiple implementations of a computation as such.

IBM's liquid metal [120] uses a single unified object-oriented programming language called Lime to program both CPUs and FPGAs in the system. The Lime language extends Java with immutable value types and enum-indexed arrays, considering that enum has a bounded number of values in Lime. Like PetaBricks, the idea is to provide a unified abstraction to program different devices present in the system. However, devising one high-level programming abstraction that can be used to *effectively* program different devices considering significant architectural differences between such devices is a difficult task. Because of these reasons, certain programming constructs in Lime language are only meaningful for FPGAs (or CPUs) and should not be used in code meant to execute on other compute devices.

Sequoia [96] is a hierarchical stream programming language, originally developed for the Cell processor. It has been extended to Sequoia-C++ and other target platforms. Unlike C++, Sequoia makes the memory hierarchy/distribution explicit to the programmer. The memory structure is represented in a hierarchical representation (a tree) where nodes closer to the root node represent larger and slower memory (e.g., RAM) and nodes closer to leaves represent smaller but faster memory units (e.g., registers, L1 cache). Each memory node has an associated processing element (e.g. a CPU core or a GPU) that can directly access data from that memory by load and store instructions. Data transfer from parent memory to child memory and vice-versa is done transparently (via MPI calls, DMA transfers, or by using explicit loads and stores). The Sequoia programming model is suitable for divide-and-conquer problems such as sorting or matrix multiplication, where a problem can be partitioned recursively into smaller subproblems by spawning new tasks and partitioning data into smaller chunks that can fit in successor memory (i.e., faster but smaller memories than predecessor memory types in the memory hierarchy tree). Sequoia offers portability for different architectures by tuning appropriately granularity of computation for each machine [185].

8.2.2 Component-based approaches

Elastic Computing [209] is a framework that relies on a library of ready-made implementations for specific computations (called "elastic functions") that are explored offline by an implementation planning tool to determine optimized execution configuration for a given platform. Each elastic function provides a performance model based on a one-dimensional *metric* value that is used to measure execution time of that function for a given input. However, the elastic computing framework does not directly manage execution on GPU devices; rather, elastic functions can internally manage execution on GPU devices along with resource allocation and data management. In our work, the frameworks are aware of GPU devices and manages/optimizes data movements for execution on these devices. Also, our performance modeling framework is more generic and supports models with n-dimensional parameter spaces. Lastly, our frameworks can do execution for computations with no performance models available beforehand where the runtime system can manage execution in a greedy load-balancing manner. In scenarios with repetitive executions, we can even learn the performance models online.

In [131], Kessler and Löwe discuss algorithmic selection for explicitly parallel software components for a multi-processor machine where a component has multiple implementation variants. Each implementation variant of a component contains metacode ($time_f$ method) for each of its performance-aware methods f that is used to predict the execution time of the method f, based on problem and processor group sizes. They use an interleaved dynamic programming algorithm to construct dispatch tables offline; afterwards, these dispatch tables are looked up at runtime to find the expected best implementation variant, processor allocation and schedule for given problem and processor group sizes. Their approach works best for divide-and-conquer algorithms where variant and schedule selection for each component invocation in the call tree often results in significant performance gains compared to any fixed variant execution.

Kicherer et al. [133] also use a component-based approach to address programming issues with modern heterogeneous systems. In [132],

they proposed a performance model that is suited for online learning and implementation selection decision making between available implementation variants for a given invocation context. In their approach, each implementation variant is loaded as a dynamic library at runtime instead of a single binary solution. The major difference, in comparison to our work, comes in data management as their runtime library does not handle (and thus cannot optimize) data transfers for GPU execution. The operand data is always placed in main memory and GPU implementations internally transfer data back and forth for operand data. This could have serious performance implications in GPU-based systems as data transfers cannot be optimized across different invocations in their case. Similarly, overlapping computation and communication is also not possible considering data transfers are not controlled by the runtime system. A further limitation of their work is related to simultaneous execution on both CPU and GPU devices which could yield significant performance benefits for certain data-parallel applications.

8.2.3 Task-based programming models

Support for task parallelism is added in OpenMP version 3.0 [24, 23]. A new `task` construct is introduced which can be used to annotate arbitrary computations as tasks; a task in OpenMP can spawn further tasks dynamically to utilize nested parallelism adaptively with parent-child relationship. Execution of already spawned tasks can be synchronized using the `taskwait` construct. In OpenMP 4.0 [169], more fine grained task synchronization is supported where a group of tasks can be synchronized rather than all spawned tasks. Moreover, dependencies between individual tasks can be specified using the newly introduced `depend` clause. Besides enhancements in the tasking model, the OpenMP 4.0 API has introduced several new features including support for programming accelerators, SIMD programming, and better thread affinity.

One problem with the OpenMP tasking model is that it does not model data flow information across different tasks (input and output operands for a task) which can help in inferring data dependen-

cies between different tasks [104]. This is used in StarPU as well as other tasking models for heterogeneous systems including OmpSs and StarSs, as described below.

StarSs (Star Superscalar) [179] is a task-based programming model that consists of few OpenMP-like pragmas, a source-to-source translator, and a runtime system that schedules tasks for execution while preserving their dependencies. The StarSs programming model has been implemented for different platforms, such as GRIDSs for Grid environments, CellSs for IBM Cell B.E. [177], SMPSs for multicore processors [176] and GPUSs for GPUs [25]. Similar to OpenMP, StarSs has a `task` construct that can be used to mark code as a task for execution; however, unlike OpenMP, it supports implicit synchronization based on data dependencies between different tasks. Each task in StarSs uses `in` (standing for input), `out` (standing for output) and `inout` (standing for input/output) clauses to specify operations on its operand data. This information is used to infer dependencies between different tasks for synchronization; it also helps in exploiting parallelism between multiple tasks that have no data dependency.

OmpSs [86, 46] is a programming model that combines ideas from OpenMP and StarSs. It enhances OpenMP with support for irregular and asynchronous parallelism. It incorporates the idea of disjoint address spaces that allows the compiler/runtime to automatically move data as necessary and perform different kinds of optimizations.

OmpSs and StarSs are quite similar to StarPU which we have used in our work. One major difference is that tasks in OmpSs and StarSs variants have normally one implementation. Recently, one scheduling and selection policy in OmpSs (called versioning) is added that can handle tasks with multiple implementations. It is quite similar to the HEFT scheduling and selection policy that we have used in our runtime library and the one available in StarPU. For a more detail comparison between StarPU and StarSs/OmpSs, we refer to [19].

8.2.4 Code generation

There exists a rich body of work on automatically porting code written in other languages besides CUDA and OpenCL for GPU execu-

tion. GPU language translators have been made available for several languages including Java [213], C [28], C++ [42], OpenMP [141], Python [136], Ruby [155], .NET [137], MATLAB [43] and Fortran [186]. The porting process focuses on exploiting the massive computational capabilities of modern GPUs by porting structured data-parallel parts of the application such as for-loops in C++ with no loop-carried dependences etc. Loops are the primary target considering that they often consume most of the application's execution time, thus promising significant reductions in execution time.

Considering that CPU and GPU have different architectural features, a simple port of existing CPU code for GPU execution may run slowly for several reasons. The code should have sufficient work to supersede the potential overhead associated with GPU execution, data transfers and synchronization. Simply porting all data parallel regions for GPU execution may prove sub-optimal in real situations. One other source of deficiency can come from data structures and access patterns. On CPU, computations on consecutive elements of an array are processed by the same thread to exploit cache and data-locality; however, in GPU execution, consecutive array elements must be processed by different threads running in parallel (i.e., in a warp) to achieve better global memory bandwidth. When porting loops inside OpenMP programs for GPU execution, Grewe et al. [109] used machine learning to decide, for each loop, whether to execute it on a CPU or a GPU. They also apply compiler-based transformations to optimize data access patterns for GPUs.

Our work is orthogonal to the approaches discussed above that generate e.g. CUDA/OpenCL code for GPU execution from CPU code written in languages such as C, C++ and OpenMP. This means that these approaches can be used to generate GPU implementations of components from existing sequential code for that computation. These generated implementations could be hand-tuned further when human resources become available.

8.3 Component Models in HPC and Grid Computing

The traditional COTS (components-off-the-shelf) component models such as CORBA [205] and EJB [88] have been developed for the sequential, concurrent and networked usage context with a standard single-CPU execution platform in mind. Their main task is to abstract from language, platform and location specificity, to provide interoperability and portability, and to interface to legacy systems. Traditionally, these functional and interoperability aspects have been more important to component system designers than performance. However, the component concept is increasingly being adopted also in high-performance and grid computing, where non-functional properties like response times and throughput are more important. We briefly review some approaches below.

8.3.1 Component Models in Grid Computing

GCM (Grid Component Model), developed in the CoreGRID NoE [63], is a component model designed specifically for grid computing. It is based upon FRACTAL [45], which is a generic, extensible and programming-language agnostic component model that can be realized in various different ways (due to its genericity). GCM builds upon FRACTAL by supporting an autonomic controller for each component, which can automatically manage its component by monitoring current status of the component and by executing necessary modifications. Moreover, it supports advanced communication patterns (data, stream, event ports, and collective communication). It extends the existing FRACTAL ADL with new definitions for its added Grid-specific features.

CCA (Common component architecture) [15] is a standardization effort towards a multi-linguistic component framework for high performance computing. It provides a framework for components to interact, thus abstracting away interoperability semantics (including support for location transparency).

8.3.2 Component Models in HPC

There is a growing trend in HPC to benefit from a component oriented programming style that is already widely used in the non-HPC software engineering community. However, the need for maintaining performance while going for component orientation led to several new initiatives in the HPC community. OnRamp [122] addresses the issue of keeping familiar software programming models intact, while achieving component orientation. It is based upon annotating existing software systems with mark-up (annotations) that identify components, their interaction and different composition semantics. It relies on the tool chain of CCA for its working.

Foley *et al.* [101] extend a component interface to model enriched interaction between different components, including file and event-based interaction beyond the classical method invocation. In their component model, the component interface only participates in control flow, where as data is exchanged via files and additional activities are coordinated via the event service. They introduce file interaction descriptors and even interaction descriptors for modeling file and event-based interaction respectively.

Besides general-purpose component frameworks, several domain-specific component frameworks have been created in many domains, including climate modeling [61], rocket simulation [125], and engineering [188]. Contract negotiation and its usage for modeling and fulfilling non-functional requirements is discussed in [198, 67, 143].

"One never notices what has been done; one can only see what remains to be done."

Marie Curie

Chapter 9

Possible extensions

There are of course many ways to improve and extend the work presented in this thesis. Two obvious future work options for all three approaches are: 1) to consider other optimization objectives such as energy consumption for composition decisions and 2) to experiment with other platforms such as embedded devices (smartphones, tablets etc.) and clusters of GPU-based systems. In the following, we will discuss some specific tracks for our three approaches that we consider interesting for further study.

9.1 SkePU extensions

In future extensions, more data- and task-parallel skeletons can be added in SkePU. Task-parallel skeletons require flexible runtime support which might come from existing integration with the StarPU runtime system. Otherwise, our in-house developed runtime library (see Section 6.2.3) could be used (and extended if needed) for such purpose. The most obvious choices for new task-parallel skeletons are `Pipeline`, `Parallel_for`, `Branch-and-bound` and `Divide-and-conquer`.

Regarding data-parallel skeletons, support for data-parallel operations on sparse matrices along with a smart container type for sparse matrix operands could be an interesting idea. The operations on sparse matrices such as sparse matrix-vector multiplication are widely used in solving sparse linear systems and eigenvalue prob-

lems [87, 159, 206]. This work would first require defining exact semantics for skeletons such as MapOverlap for sparse matrix operations. Moreover, sparse matrix operations are typically much less regular in their access patterns and can be tricky to implement efficiently on throughput-oriented processors such as GPUs [30].

The implementation selection mechanism implemented in SkePU uses adaptive offline sampling to decide which implementation to use for a given skeleton call. It currently does not support selection for skeleton calls that can be executed across multiple GPUs. This could be added in future by considering a skeleton call execution on multiple GPUs as another possible implementation variant in the algorithm.

A long-term future work could be to add a pre-processing tool that can parse a skeleton program and optimize the skeleton operations by doing source code analysis and transformations. One simple example could be to analyze data-access patterns in user-functions supplied by the programmer. This analysis could help in, e.g., optimizing the MapOverlap operations depending on the overlap access patterns. With addition of task-parallel skeletons as well as sparse matrix operations, this analysis and transformation mechanism could become even more important for achieving good performance in a portable manner, while providing a high level of abstraction to the programmer.

Another important future work is to carry out further case studies of porting new applications to SkePU. This would help in two ways: first, it would show the genericity of SkePU modeled skeletons, and second, feed-back from such case studies can be used to improve (or even devise new) skeletons. Last but not the least, SkePU can be ported for execution on different platforms. We have already reported our initial work [153] on porting SkePU for execution on MPI-clusters with or without GPU-enhanced nodes. In future, execution on systems with Intel Xeon Phi and FPGAs could be demonstrated. This work might not require any new skeleton implementations, as Intel Xeon Phi and FPGAs support execution using OpenMP and OpenCL respectively.

9.2 PEPPHER Composition tool

Although the PEPPHER project was completed at the end of 2012, it is possible to use (and extend) the composition tool in other contexts in future.

The composition tool composes annotated components using the PEPPHER component model which might be extended to support a more diverse set of performance-related annotations. One idea is to improve annotations for the user to supply his/her own performance model, which could be an analytical model. This would require provision for runtime feedback back to the performance model to update/calibrate it during the actual program execution. We have implemented a similar facility in our global composition framework (see Chapter 6).

The composition tool uses smart containers to provide high-level data abstraction and leverages their data distribution and synchronization capabilities in a transparent manner. With these containers, we have achieved asynchronous component execution which, alongside the implicit data-dependency detection facility for operand data, enabled the exploitation of inter-component parallelism. Currently, we have smart containers for modeling scalar, 1D vector and 2D matrix operations. In future, the set of smart containers could be extended for other commonly used data structures such as stack, queue, tree etc. Some data structures may employ more advanced optimizations. For example, considering the data access patterns for a certain operand, the data storage format can be optimized for that particular operand [162].

A more interesting work would be to investigate further the interface between the runtime system and the application layer including the composition tool. Many existing approaches design their framework in an integrated manner, employing compiler analysis/transformations to inject calls to their runtime system which manages/monitors execution at runtime. This results in tight integration between the runtime system and the outside world. Considering that runtime systems are developed separately by different groups (e.g., StarPU from INRIA), proper interfacing mechanisms would enable high-level

approaches to effectively utilize the runtime system capabilities and (possibly) vice-versa.

9.3 Global Composition Framework (GCF)

There exist several possibilities for future work on the GCF composition framework. First and foremost, the implementation can be improved to do more comprehensive program analysis to extract more information for decision making statically. In some cases, this may enable the propagation of information such as the problem sizes for some operand data, which could help in pruning certain implementations offline. Also, the runtime system library designed and developed in this work can be extended to support concurrent execution on GPU-based systems.

Currently, the composition decisions are made by our runtime system assuming standalone execution of the application on a dedicated system. Recently, StarPU has implemented support for the co-execution of multiple applications each using the StarPU runtime system [121]. The resources can be shared by virtualization and the runtime system needs to consider the global resource allocation when making the scheduling and selection decision for a given component call (or a task). One idea could be to provide a more explicit notion of resource requirements in the component model so that the implementation writer can clearly specify the type and amount of resources required. However, in many cases this could be difficult for a programmer to know as it could be system and input dependent. Another idea could be to profile the application execution to infer such information automatically. This information can be used both by the composition tool and the runtime system to make effective scheduling and selection decisions considering resource sharing and co-execution of multiple applications at the same time.

The composition framework can in future be tried for other systems including MPI- and GPU-clusters as well as systems with e.g. Intel Xeon and FPGA accelerators. As we already support OpenMP and OpenCL, and with the ROSE source-to-source compiler's avail-

able support for MPI, this work could perhaps be done with some modifications in the code generation module (`composer`) of our global composition framework.

One important direction for future work would be to explore more on the global component composition for GPU-based systems. There are primarily two challenges. 1) To find more applications where one can show benefits of global composition decisions (considering multiple component calls at a time) over the local composition decisions (considering one component call at a time) using HEFT. 2) To carry out such composition decisions in an automated manner with limited overhead. In our work, we made composition decisions (mostly) online, which provides greater flexibility in terms of knowledge about problem sizes and system state but puts a restriction on the decision overhead. One idea could be to deduce frequent dependence patterns in the program task graph from one or more program executions and use that along with information about the performance behavior of each task to make certain decisions offline. This could work for applications with a limited number of execution contexts (problem sizes, input data etc.) and with the assumption that executions with the same execution context would generate the same task graph (i.e., generation of tasks does not depend upon some random values). If the task graph is known before execution along with the performance characteristics of each task, the composition decisions can be evaluated offline and stored for lookup during the actual program execution for different execution contexts. However there exist certain problems with this offline approach:

- Size and shape of the task graph of an application can depend on the problem size and even (in some cases) on the actual contents of the input data.
- Knowledge about the system workload and resource sharing needs to be considered. Modeling it as a possible context property would be difficult as system workload can be highly unpredictable in the presence of co-running applications and services.
- Global composition decisions may require program transforma-

tions (e.g., component call reordering etc.) that can require re-compilation; thus it might not be possible to be done on the fly at runtime. One idea could be to include all (or a subset of) possible permutations within the source code, and at runtime jump to the right permutation depending upon the execution context [131].

Other techniques such as speculative execution could be interesting to try for certain application scenarios. Speculative execution [80] allows execution of different component implementations on different compute devices in parallel where the result from the implementation that finishes first can be retained; other implementations' executions can be aborted.

Bibliography

[1] ISO/IEC 14977, EXTENDED BNF, 1996. `http://www.dataip.co.uk/Reference/iso-14977.pdf`.

[2] David Abrahams and Aleksey Gurtovoy. *C++ Template Metaprogramming: Concepts, Tools, And Techniques From Boost And Beyond*. The C++ in-Depth Series. Addison Wesley Professional, 2005.

[3] Sarita Adve et al. Parallel computing research at Illinois: The UPCRC agenda, November 2008. White Paper. University of Illinois, Urbana-Champaign.

[4] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[5] Shameem Akhter and Jason Roberts. *Multi-Core Programming: Increasing Performance Through Software Multi-Threading*. Books by engineers, for engineers. Intel Press, 2006.

[6] Enrique Alba et al. MALLBA: a library of skeletons for combinatorial optimisation (research note). In *8th International Euro-Par Conference on Parallel Processing*, Euro-Par '02, pages 927–932, London, UK, UK, 2002. Springer-Verlag.

[7] Marco Aldinucci, Sonia Campa, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. Design patterns percolating to parallel programming framework implementation. *International Journal of Parallel Programming*, pages 1–20, 2013.

[8] Marco Aldinucci, Marco Danelutto, and Patrizio Dazzi. MUSKEL: An expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4), 2007.

[9] Marco Aldinucci, Marco Danelutto, Peter Kilpatrick, and Massimo Torquati. FastFlow: high-level and efficient streaming on multi-core. In *Programming Multi-core and Many-core Computing Systems*, Parallel and Distributed Computing. Wiley, 2014. to appear.

[10] Marco Aldinucci, Marco Danelutto, and Paolo Teti. An advanced environment supporting structured parallel programming in Java. *Future Generation Computer Systems*, 19(5):611–626, 2003.

[11] Andrei Alexandrescu. *Modern C++ design.* Addison-Wesley Professional; 1 edition, 2001.

[12] Jesper Andersson, Morgan Ericsson, Christoph Kessler, and Welf Löwe. Profile-guided composition. In *Proc. Software Composition (SC'08), Springer LNCS 4954*, 2008.

[13] Jason Ansel, Cy Chan, Yee Lok Wong, Marek Olszewski, Qin Zhao, Alan Edelman, and Saman Amarasinghe. PetaBricks: A language and compiler for algorithmic choice. *SIGPLAN Not.*, 44:38–49, June 2009.

[14] Mikio Aoyama. New age of software development: How component-based software engineering changes the way of software development? In *1st Workshop on Component Based Software Engineering*, 1998.

[15] Rob Armstrong, Dennis Gannon, Al Geist, Katarzyna Keahey, Scott Kohn, Lois McInnes, Steve Parker, and Brent Smolinski. Toward a common component architecture for high-performance scientific computing. In *HPDC '99: 8th IEEE International Symposium on High Performance Distributed Computing*, Washington, DC, USA, 1999. IEEE Computer Society.

[16] Krste Asanovic et al. The landscape of parallel computing research: A view from Berkeley. Technical Report UCB/EECS-2006-183, EECS Department, University of California, Berkeley, December 2006.

[17] Krste Asanovic et al. A view of the parallel computing landscape. *Commun. ACM*, 52:56–67, October 2009.

[18] Richard Aufmann, Vernon Barker, and Joanne Lockwood. *Intermediate Algebra with Applications, Multimedia Edition*. Cengage Learning, 2008.

[19] Cédric Augonnet. *Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective*. Phd thesis, Université Bordeaux 1, 2011.

[20] Cédric Augonnet, Samuel Thibault, and Raymond Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par Workshops 2009 (HPPC 2009)*, volume 6043 of *Lecture Notes in Computer Science*. Springer, 2009.

[21] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[22] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, 2009.

[23] Eduard Ayguadé et al. A proposal for task parallelism in OpenMP. In *A Practical Programming Model for the Multi-Core Era*, volume 4935 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin Heidelberg, 2008.

[24] Eduard Ayguadé et al. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3):404–418, March 2009.

[25] Eduard Ayguadé et al. An extension of the StarSs programming model for platforms with multiple GPUs. In *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 851–862. Springer Berlin Heidelberg, 2009.

[26] Bruno Bacci, Marco Danelutto, Susanna Orlando, Marco Vanneschi, and Salvatore Pelagatti. Summarising an experiment in parallel programming language design. In *High-Performance Computing and Networking*, volume 919 of *Lecture Notes in Computer Science*, pages 7–13. Springer Berlin / Heidelberg, 1995.

[27] Felix Bachman, Len Bass, Charles Buhman, Santiago Comella-Dorda, Fred Long, John Robert, Robert Seacord, and Kurt Wallnau. Volume II: Technical concepts of component-based software engineering. Technical report, 2000. CMU/SEI-2000-TR-008, ESC-TR-2000-007.

[28] Muthu M. Baskaran, J. Ramanujam, and P. Sadayappan. Automatic C-to-CUDA code generation for affine programs. In *Compiler Construction*, volume 6011 of *Lecture Notes in Computer Science*, pages 244–263. Springer Berlin Heidelberg, 2010.

[29] Nathan Bell and Michael Garland. CUSP library v0.2: Generic parallel algorithms for sparse matrix and graph computations. http://code.google.com/p/cusp-library/. `http://code.google.com/p/cusp-library`.

[30] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 18:1–18:11, New York, NY, USA, 2009. ACM.

[31] Siegfried Benkner, Enes Bajrovic, Erich Marth, Martin Sandrieser, Raymond Namyst, and Samuel Thibault. High-level support for pipeline parallelism on many-core architectures. In *Proceedings of the 18th International Conference on Parallel Processing*, Euro-Par'12, pages 614–625, Berlin, Heidelberg, 2012. Springer-Verlag.

[32] Siegfried Benkner, Sabri Pllana, Jesper Larsson Traff, Philippas Tsigas, Uwe Dolinsky, Cedric Augonnet, Beverly Bachmayer, Christoph Kessler, David Moloney, and Vitaly Osipov. PEPPHER: Efficient and productive usage of hybrid computing systems. *IEEE Micro*, 99(PrePrints), 2011.

[33] Anne Benoit and Murray Cole. Two fundamental concepts in skeletal parallel programming. In *The International Conference on Computational Science (ICCS 2005) , Part II, LNCS 3515*, pages 764–771. Springer Verlag, 2005.

[34] Anne Benoit, Murray Cole, Stephen Gilmore, and Jane Hillston. Flexible skeletal programming with eSkel. In *Euro-Par 2005 Parallel Processing*, volume 3648 of *Lecture Notes in Computer Science*, pages 613–613. Springer Berlin / Heidelberg, 2005.

[35] Jon Louis Bentley and Jerome H. Friedman. Data structures for range searching. *ACM Comput. Surv.*, 11(4):397–409, December 1979.

[36] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[37] Richard S. Bird. Lectures on Constructive Functional Programming. In *Constructive Methods in Computing Science*, NATO ASI Series F: Computer and Systems Sciences, pages 151–216. Springer-Verlag, 1989.

[38] Guy E. Blelloch. *Vector models for data-parallel computing*. MIT Press, Cambridge, MA, USA, 1990.

[39] Barry Boehm. Some future software engineering opportunities and challenges. In *The Future of Software Engineering*, pages 1–32. Springer Berlin Heidelberg, 2011.

[40] George H. Botorog and Herbert Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *5th IEEE International Symposium on High Performance Distributed Computing*, pages 243–252, August 1996.

[41] Mathias Bourgoin, Emmanuel Chailloux, and Jean L. Lamotte. SPOC: GPGPU programming through stream processing with OCaml. *Parallel Processing Letters*, 22, 2012.

[42] Jens Breitbart. CuPP - a framework for easy CUDA integration. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–8, 2009.

[43] Andre Rigland Brodtkorb. The graphics processor as a mathematical coprocessor in MATLAB. In *International Conference on Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 822–827, 2008.

[44] François Broquedis, Jérôme Clet-Ortega, Stephanie Moreaud, Nathalie Furmento, Brice Goglin, Guillaume Mercier, Samuel Thibault, and Raymond Namyst. hwloc: A generic framework for managing hardware affinities in HPC applications. In *18th Euromicro Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 180–186. IEEE Computer Society, 2010.

[45] Eric Bruneton, Thierry Coupaye, and Jean-Bernard Stefani. The fractal component model, 2004. `http://fractal.ow2.org/specification/fractal-specification.pdf`.

[46] Javier Bueno et al. Productive cluster programming with OmpSs. In *Euro-Par 2011 Parallel Processing*, volume 6852 of *Lecture Notes in Computer Science*, pages 555–566. Springer Berlin Heidelberg, 2011.

[47] Sonia Campa et al. Heterogeneous implementation of initial generic patterns. Technical report, ©the ParaPhrase Consortium, 2013. Date of preparation (latest version) March 31, 2013.

[48] Denis Caromel and Mario Leyton. Fine tuning algorithmic skeletons. In *Euro-Par 2007 Parallel Processing*, volume 4641 of *Lecture Notes in Computer Science*, pages 72–81. Springer Berlin / Heidelberg, 2007.

[49] Thomas L. Casavant and Jon G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Trans. Softw. Eng.*, 14(2):141–154, February 1988.

[50] Byran Catanzaro, Armando Fox, Kurt Keutzer, David Patterson, Bor-Yiing Su, Marc Snir, Kunle Olukotun, Pat Hanrahan, and Hassan Chafi. Ubiquitous parallel computing from Berkeley, Illinois, and Stanford. *Micro, IEEE*, 30(2):41–55, March 2010.

[51] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming.* Number v. 10 in Scientific Computation Series. MIT Press, 2008.

[52] Robert N. Charette. Why software fails [software failure]. *IEEE Spectr.*, 42(9):42–49, September 2005.

[53] Shuai Che, Michael Boyer, Jiayuan Meng, David Tarjan, Jeremy W. Sheaffer, Sang-Ha Lee, and Kevin Skadron. Rodinia: A benchmark suite for heterogeneous computing. In *IEEE International Symposium on Workload Characterization (IISWC)*, pages 44–54, 2009.

[54] Philipp Ciechanowicz, Michael Poldner, and Herbert Kuchen. The Munster skeleton library Muesli - a comprehensive overview, 2009. ERCIS Working Paper No. 7.

[55] Francois Clément, Vincent Martin, Arnaud Vodicka, Roberto Di Cosmo, and Pierre Weis. Domain decomposition and skeleton programming with OCamlP3l. *Parallel Computing*, 32(7–8):539–550, 2006.

[56] Murray Cole. A "skeletal" approach to the exploitation of parallelism. In *The CONPAR Conference*, pages 667–675, New York, NY, USA, 1989. Cambridge University Press.

[57] Murray Cole. *Algorithmic skeletons: Structured management of parallel computation*. MIT Press, Cambridge, MA, USA, 1991.

[58] Murray Cole. Bringing skeletons out of the closet: A pragmatic manifesto for skeletal parallel programming. *Parallel Comput.*, 30:389–406, March 2004.

[59] Murray Cole and Andrea Zavanella. Coordinating heterogeneous parallel systems with skeletons and activity graphs. *Journal of Systems Integration*, 10:127–143, 2001.

[60] Alexander Collins, Christian Fensch, and Hugh Leather. Autotuning parallel skeletons. *Parallel Processing Letters*, 22(02), 2012.

[61] Nancy Collins et al. Design and implementation of components in the earth system modeling framework. *International Journal of High Performance Computing Applications*, 19(3):341–350, 2005.

[62] Douglas E. Comer, David Gries, Michael C. Mulder, Allen Tucker, A. Joe Turner, and Paul R. Young. Computing as a discipline. *Commun. ACM*, 32(1):9–23, January 1989.

[63] Coregrid: Basic features of the grid component model, 2007. Programming Model Institute, Deliverable D.PM.04, `http://www-sop.inria.fr/oasis/Ludovic.Henrio/CoreGrid/GCM-assessed-DPM04.pdf`.

[64] Cristian Ţăpuş, I-Hsin Chung, and Jeffrey K. Hollingsworth. Active Harmony: Towards automated performance tuning. In *Proceedings of the ACM/IEEE Supercomputing Conference*, pages 44–44, Nov 2002.

[65] CUDA occupancy calculator, 2014. NVIDIA Corporation, `http://developer.download.nvidia.com/compute/cuda/CUDA_Occupancy_calculator.xls`.

[66] Leonardo Dagum and Ramesh Menon. OpenMP: an industry standard API for shared-memory programming. *Computational Science Engineering, IEEE*, 5(1):46–55, 1998.

[67] Tammy Dahlgren and Irina Abramova. Toward multi-language, multi-component interface contract enforcement. In *CBHPC '09: Workshop on Component-Based High Performance Computing*, pages 1–4, New York, NY, USA, 2009. ACM.

[68] Marco Danelutto, Roberto Di Meglio, Salvatore Orlando, Susanna Pelagatti, and Marco Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3):205–220, 1992.

[69] Marco Danelutto and Massimo Torquati. A RISC building block set for structured parallel programming. In *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 46–50, 2013.

[70] John Darlington, John A. Field, Peter G. Harrison, Paul H. J. Kelly, David W. N. Sharp, and Qian Wu. Parallel programming using skeleton functions. In *5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, pages 146–160, London, UK, 1993. Springer-Verlag.

[71] John Darlington, Yi-ke Guo, Hing To, and Jin Yang. Functional skeletons for parallel coordination. In *EURO-PAR '95 Parallel Processing*, volume 966 of *Lecture Notes in Computer Science*, pages 55–66. Springer Berlin / Heidelberg, 1995.

[72] John Darlington and Hing W. To. Building parallel applications without programming. In *Abstract machine models for highly parallel computers*, pages 140–154. Oxford University Press, Oxford, UK, 1995.

[73] Usman Dastgeer, Lu Li, and Christoph Kessler. D1.4: Research prototype implementation. Technical report, ©the PEPPHER Consortium, 2012. Date of preparation (latest version) December 31, 2012.

[74] Usman Dastgeer, Lu Li, and Christoph Kessler. The PEPPHER composition tool: Performance-aware dynamic composition of applications for GPU-based systems. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 711–720, 2012.

[75] Timothy A. Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[76] Jeffrey Dean and Sanjay Ghemawat. MapReduce: Simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[77] Robert H. Dennard et al. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974.

[78] Peter J. Denning. The evolution of parallel processing. *Am. Sci.*, 73(5):414–416, September 1985.

[79] Peter J. Denning. Computer science. In *Encyclopedia of Computer Science*, pages 405–419. John Wiley and Sons Ltd., Chichester, UK, 2003.

[80] Gregory Diamos and Sudhakar Yalamanchili. Speculative execution on multi-GPU systems. In *IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12, 2010.

[81] Antonio J. Dorta, Jesus A. Gonzalez, Casiano Rodriguez, and Francisco De Sande. LLC: A parallel skeletal language. *Parallel Processing Letters*, 13:437–448, July 2003.

[82] Maciej Drozdowski. *Scheduling for Parallel Processing*. Springer Publishing Company, Incorporated, 1st edition, 2009.

[83] Peng Du, Rick Weber, Piotr Luszczek, Stanimire Tomov, Gregory Peterson, and Jack Dongarra. From CUDA to OpenCL: Towards a performance-portable solution for multi-platform GPU programming. *Parallel Comput.*, 38(8):391–407, August 2012.

[84] Michel Dubois, Murali Annavaram, and Per Stenström. *Parallel Computer Organization and Design*. Parallel Computer Organization and Design. Cambridge University Press, 2012.

[85] Amalia Duch and Conrado Martinez. On the average performance of orthogonal range search in multidimensional data structures. In *29th International Colloquium on Automata, Languages and Programming*, ICALP '02, pages 514–524, London, UK, UK, 2002. Springer-Verlag.

[86] Alejandro Duran et al. OmpSs: A proposal for programming heterogenenous multi-core architectures. *Parallel Processing Letters*, 21(02):173–193, 2011.

[87] Adam Dziekonski, Adam Lamecki, and Maciej Mrozowski. A memory efficient and fast sparse matrix vector product on a GPU. *Progress In Electromagnetics Research*, 116:49–63, 2011.

[88] Wolfgang Emmerich and Nima Kaveh. Component technologies: Java beans, COM, CORBA, RMI, EJB and the CORBA component model. In *24rd International Conference on Software Engineering (ICSE)*, pages 691–692, May 2002.

[89] Johan Enmyren and Christoph W. Kessler. SkePU: A multi-backend skeleton programming library for multi-GPU systems. In *Proc. 4th Int. Workshop on High-Level Parallel Programming and Applications*, HLPP '10. ACM, September 2010.

[90] Steffen Ernsting and Herbert Kuchen. Algorithmic skeletons for multi-core, multi-GPU systems and clusters. *International Journal of High Performance Computing and Networking*, 7:129–138, 2012.

[91] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Commun. ACM*, 56(2):93–102, 2013.

[92] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power limitations and dark silicon challenge the future of multicore. *ACM Trans. Comput. Syst.*, 30(3):11:1–11:27, 2012.

[93] Hadi Esmaeilzadeh, Ting Cao, Xi Yang, Stephen M. Blackburn, and Kathryn S. McKinley. Looking back and looking forward: Power, performance, and upheaval. *Commun. ACM*, 55(7):105–114, 2012.

[94] Joel Falcou, Jocelyn Sérot, Thierry Chateau, and Jean-Thierry Lapresté. Quaff: Efficient C++ design for parallel skeletons. *Parallel Computing*, 32(7-8):604–615, 2006.

[95] Robert M. Farber. Topical perspective on massive threading and parallelism. *Journal of Molecular Graphics and Modelling*, 30(0):82–89, 2011.

[96] Kayvon Fatahalian et al. Sequoia: Programming the memory hierarchy. In *ACM/IEEE Conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[97] Paul Feautrier. Dataflow Analysis of Array and Scalar References. *International Journal of Parallel Programming*, 20(1), 1991.

[98] David Fernandez-Baca. Allocating modules to processors in a distributed system. *IEEE Transactions on Software Engineering*, 15(11):1427–1436, 1989.

[99] Joâo F. Ferreira, Joâo L. Sobral, and Alberto J. Proenca. JaSkel: A Java skeleton-based framework for structured cluster and grid computing. In *CCGRID, IEEE Computer Society*, pages 301–304, 2006.

[100] Raphael Finkel and Udi Manber. DIB - A distributed implementation of backtracking. *ACM Trans. Program. Lang. Syst.*, 9:235–256, March 1987.

[101] Samantha S. Foley, Wael R. Elwasif, David E. Bernholdt, Aniruddha G. Shet, and Randall Bramley. Extending the concept of component interfaces: experience with the integrated plasma simulator. In *CBHPC '09: Workshop on Component-Based High Performance Computing*, pages 1–4, New York, NY, USA, 2009. ACM.

[102] Matteo Frigo and Steven G. Johnson. FFTW: an adaptive software architecture for the FFT. In *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing*, volume 3, pages 1381–1384 vol.3, May 1998.

[103] Michael R. Garey and David S. Johnson. *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1990.

[104] Priyanka Ghosh, Yonghong Yan, and Barbara Chapman. Support for dependency driven executions among OpenMP tasks. In *Data-Flow Execution Models for Extreme Scale Computing (DFM)*, pages 48–54, Sept 2012.

[105] Mehdi Goli and Horacio Gonzalez-Velez. Heterogeneous algorithmic skeletons for FastFlow with seamless coordination over hybrid architectures. In *21st Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, pages 148–156, 2013.

[106] Horacio González-Vélez and Mario Leyton. A survey of algorithmic skeleton frameworks: High-level structured parallel programming enablers. *Softw. Pract. Exper.*, 40:1135–1160, November 2010.

[107] Chris Gregg and Kim Hazelwood. Where is the data? Why you cannot debate CPU vs. GPU performance without the answer.

In *IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 134–144, April 2011.

[108] Clemens Grelck. Shared memory multiprocessor support for functional array processing in SAC. *J. Funct. Program.*, 15:353–401, May 2005.

[109] Dominik Grewe, Zheng Wang, and Michael F.P. O'Boyle. Portable mapping of data parallel programs to OpenCL for heterogeneous systems. In *CGO '13: 11th International Symposium on Code Generation and Optimization*. ACM, 2013.

[110] Richard A. Hankins et al. Multiple instruction stream processor. In *33rd International Symposium on Computer Architecture (ISCA)*, pages 114–127, 2006.

[111] Mark Harris, John Owens, Shubho Sengupta, Yao Zhang, and Aal Davidson. CUDPP: CUDA data parallel primitives library, 2011. `http://code.google.com/p/cudpp/`.

[112] Christoph A. Herrmann and Christian Lengauer. HDC: A higher-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2/3):239–250, 2000.

[113] Jason Hickey, Anil Madhavapeddy, and Yaron Minsky. *Real World OCaml*. O'Reilly Media, Incorporated, 2013.

[114] Mark D. Hill and Michael R. Marty. Amdahl's law in the multicore era. *Computer*, 41(7):33–38, July 2008.

[115] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[116] Jared Hoberock and Nathan Bell. Thrust: C++ template library for CUDA, 2011. `http://code.google.com/p/thrust/`.

[117] Sunpyo Hong and Hyesoon Kim. An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness. In *Proc. Annual International Symposium on Computer Architecture (ISCA)*, 2009.

[118] Zoltán Horváth, Viktória Zsók, Pascal Serrarens, and Rinus Plasmeijer. Parallel element wise processable functions in Concurrent Clean. *Mathematical and Computer Modelling*, 38(7-9):865–875, 2003.

[119] Lee Howes, Anton Lokhmotov, Alastair F. Donaldson, and Paul H. J. Kelly. Towards metaprogramming for parallel systems on a chip. In *International conference on Parallel processing*, Euro-Par'09, pages 36–45, Berlin, Heidelberg, 2010. Springer-Verlag.

[120] Shan Shan Huang, Amir Hormati, David F. Bacon, and Rodric Rabbah. Liquid metal: Object-oriented programming across the hardware/software boundary. In *22nd European conference on Object-Oriented Programming (ECOOP)*, 2008.

[121] Andra-Ecaterina Hugo, Abdou Guermouche, Raymond Namyst, and Pierre-André Wacrenier. Composing multiple StarPU applications over heterogeneous machines: a supervised approach. In *Third International Workshop on Accelerators and Hybrid Exascale Systems*, 2013.

[122] Geoffrey C. Hulette, Matthew J. Sottile, Robert Armstrong, and Benjamin Allan. OnRamp: enabling a new component-based development paradigm. In *CBHPC '09: Workshop on Component-Based High Performance Computing*, pages 1–10, New York, NY, USA, 2009. ACM.

[123] Oscar H. Ibarra and Chul E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *J. ACM*, 24(2):280–289, April 1977.

[124] Leah H. Jamieson, Dennis B. Gannon, and Robert J. Douglass, editors. *The characteristics of parallel algorithms*. Massachusetts Institute of Technology, Cambridge, MA, USA, 1987.

[125] Xiangmin Jiao et al. A system integration framework for coupled multiphysics simulations. *Engineering with Computers*, 22(3–4):293–309, 2006.

[126] Yuki Karasawa and Hideya Iwasaki. A parallel skeleton library for multi-core clusters. In *Proc. International Conference on Parallel Processing (ICPP)*, pages 84–91, 2009.

[127] Thomas Karcher and Victor Pankratius. Run-time automatic performance tuning for multicore applications. In *Euro-Par*, volume LNCS 6852. Springer Berlin Heidelberg, 2011.

[128] Kamran Karimi, Neil G. Dickson, and Firas Hamze. A performance comparison of CUDA and OpenCL. *CoRR*, abs/1005.2581, 2010.

[129] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the future of parallel computing. *Micro, IEEE*, 31(5):7–17, 2011.

[130] Paul H. Kelly. *Functional Programming for Loosely-Coupled Multiprocessors*. MIT Press, Cambridge, MA, USA, 1989.

[131] Christoph Kessler and Welf Löwe. Optimized composition of performance-aware parallel components. *Concurrency and Computation: Practice and Experience*, 24(5):481–498, 2012.

[132] Mario Kicherer, Rainer Buchty, and Wolfgang Karl. Cost-aware function migration in heterogeneous systems. In *6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 137–145, New York, NY, USA, 2011. ACM.

[133] Mario Kicherer, Fabian Nowak, Rainer Buchty, and Wolfgang Karl. Seamlessly portable applications: Managing the diversity of modern heterogeneous systems. *ACM Trans. Archit. Code Optim.*, 8(4):42:1–42:20, January 2012.

[134] Liam Kiemele, Celina Berg, Aaron Gulliver, and Yvonne Coady. Kfusion: Optimizing data flow without compromising modularity. In *12th Annual International Conference on Aspect-oriented Software Development*, AOSD '13, pages 25–36, New York, NY, USA, 2013. ACM.

[135] David Kirk and Wen mei Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann, 1st edition, February 2010.

[136] Andreas Klöckner, Nicolas Pinto, Yunsup Lee, Bryan Catanzaro, Paul Ivanov, and Ahmed Fasih. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174, 2012.

[137] Nick Kopp. CUDAfy.NET library: GPGPU from Microsoft .NET framework, 2012. `http://cudafy.codeplex.com`.

[138] Matthias Korch and Thomas Rauber. Optimizing locality and scalability of embedded Runge–Kutta solvers using block-based pipelining. *J. Parallel Distrib. Comput.*, 66:444–468, March 2006.

[139] Hong T. Kung. *Computational models for parallel computers*, pages 1–15. Prentice Hall Press, Upper Saddle River, NJ, USA, 1989.

[140] Kung-Kiu Lau and Zheng Wang. Software component models. *IEEE Transactions on Software Engineering*, 33(10):709–724, 2007.

[141] Seyong Lee, Seung-Jai Min, and Rudolf Eigenmann. OpenMP to GPGPU: A compiler framework for automatic translation and optimization. *SIGPLAN Not.*, 44(4):101–110, 2009.

[142] Mario Leyton and José M. Piquer. Skandium: Multi-core programming with algorithmic skeletons. In *18th Euromicro International Conference on Parallel, Distributed and Network-Based Processing*, PDP '10, pages 289–296, February 2010.

[143] Li Li, Tamara L. Dahlgren, Lois C. McInnes, and Boyana Norris. Interface contract enforcement for improvement of computational quality of service (cqos) for scientific components. In *CBHPC '09: Workshop on Component-Based High Performance Computing*, pages 1–5, New York, NY, USA, 2009. ACM.

[144] Lu Li. Translating C/C++ applications to a task-based representation. Master's thesis, Linköping University, 2011. LIU-IDALITH-EX-A–11/036–SE.

[145] Lu Li, Usman Dastgeer, and Christoph Kessler. Pruning strategies in adaptive off-line tuning for optimized composition of components on heterogeneous systems. In *5th Swedish Workshop on Multicore Computing (MCC-2012), Stockholm, Sweden*, November 2012.

[146] Lu Li, Usman Dastgeer, and Christoph Kessler. Adaptive off-line tuning for optimized composition of components for heterogeneous many-core systems. In *High Performance Computing for Computational Science - VECPAR 2012*, volume 7851 of *Lecture Notes in Computer Science*, pages 329–345. Springer Berlin Heidelberg, 2013.

[147] Xiaoming Li, María Jesús Garzarán, and David Padua. A dynamically tuned sorting library. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 111–, Washington, DC, USA, 2004. IEEE Computer Society.

[148] Michael D. Linderman, Jamison D. Collins, Hong Wang, and Teresa H. Meng. Merge: A programming model for heterogeneous multi-core systems. *SIGPLAN Not.*, 43:287–296, March 2008.

[149] Hans W. Loidl et al. Comparing parallel functional languages: Programming and performance. *Higher Order Symbol. Comput.*, 16:203–251, September 2003.

[150] Rita Loogen, Yolanda Ortega-mallén, and Ricardo Peña marí. Parallel functional programming in Eden. *J. Funct. Program.*, 15:431–475, May 2005.

[151] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *42nd Annual IEEE/ACM International*

Symposium on Microarchitecture, MICRO 42, pages 45–55, New York, NY, USA, 2009. ACM.

[152] Steve MacDonald, John Anvik, Steven Bromling, Jonathan Schaeffer, Duane Szafron, and Kai Tan. From patterns to frameworks to parallel programs. *Parallel Comput.*, 28:1663–1683, December 2002.

[153] Mudassar Majeed, Usman Dastgeer, and Christoph Kessler. Cluster-SkePU: A multi-backend skeleton programming library for GPU clusters. In *Proc. Int. Conf. on Parallel and Distr. Processing Techniques and Applications(PDPTA)*, 2013.

[154] Ricardo Marques, Hervé Paulino, Fernando Alexandre, and Pedro D. Medeiros. Algorithmic skeleton framework for the orchestration of GPU computations. In *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 874–885. Springer Berlin Heidelberg, 2013.

[155] Hidehiko Masuhara and Yusuke Nishiguchi. A data-parallel extension to Ruby for GPGPU: Toward a framework for implementing domain-specific optimizations. In *9th ECOOP Workshop on Reflection, AOP, and Meta-Data for Software Evolution*, RAM-SE '12, pages 3–6, New York, NY, USA, 2012. ACM.

[156] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 2–15, New York, NY, USA, 1993. ACM.

[157] Scott Meyers. *More Effective C++: 35 New Ways to Improve Your Programs and Designs.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[158] Greg Michaelson, Norman Scaife, Paul Bristow, and Peter King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications (special issue on High Level Models and Languages for Parallel Processing)*, 16(2–3):181–206, 2000.

[159] Alexander Monakov and Arutyun Avetisyan. Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation*, volume 5657 of *Lecture Notes in Computer Science*, pages 289–297. Springer Berlin / Heidelberg, 2009.

[160] Gordon E. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[161] Aaftab Munshi. The OpenCL specification version 1.1. Technical report, Khronos OpenCL Working Group, 2011.

[162] Craig Mustard, Svetozar Miucin, Yuan Liu, Karol Swietlicki, Yuxuan Li, Alexandra Fedorova, and Arrvindh Shriraman. Access declarations: Revealing data access patterns to hardware. In *Proc. 5th USENIX Workshop on Hot Topics in Parallelism (HOTPAR-2013)*, 2013.

[163] Ren Ng, Marc Levoy, Mathieu Brédif, Gene Duval, Mark Horowitz, and Pat Hanrahan. Light Field Photography with a Hand-Held Plenoptic Camera. Technical report, Stanford University, 2005.

[164] John Nickolls and William J. Dally. The GPU computing era. *Micro, IEEE*, 30(2):56–69, 2010.

[165] Cedric Nugteren and Henk Corporaal. Introducing 'bones': A parallelizing source-to-source compiler based on algorithmic skeletons. In *5th Annual Workshop on General Purpose Processing with Graphics Processing Units*, GPGPU-5, pages 1–10, New York, NY, USA, 2012. ACM.

[166] Cedric Nugteren, Pieter Custers, and Henk Corporaal. Algorithmic species: A classification of affine loop nests for parallel programming. *ACM Trans. Archit. Code Optim.*, 9(4):40:1–40:25, January 2013.

[167] NVIDIA Corporation. CUBLAS library: NVIDIA CUDA basic linear algebra subroutines. `http://developer.nvidia.com/cublas`.

[168] NVIDIA Corporation. *NVIDIA CUDA C Programming Guide*, July 2013. `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[169] OpenMP application program interface, July 2013. OpenMP specifications v4.0 `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`.

[170] Victor Pankratius, Christoph Schaefer, Ali Jannesari, and Walter F. Tichy. Software engineering for multicore systems: An experience report. In *1st International Workshop on Multicore Software Engineering*, IWMSE '08, pages 53–60, New York, NY, USA, 2008. ACM.

[171] Eunjung Park, Sameer Kulkarni, and John Cavazos. An evaluation of different modeling techniques for iterative compilation. In *Proceedings of the 14th International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES '11, pages 65–74, New York, NY, USA, 2011. ACM.

[172] Jongsoo Park. *Memory Optimizations of Embedded applications for energy efficiency*. PhD thesis, Dept. of Electrical Engineering. University of Stanford, 2011.

[173] Robert E. Park. Software size measurement: A framework for counting source statements. Technical report, Software Engineering Institute, CMU, 1992.

[174] David Patterson. The trouble with multi-core. *Spectrum, IEEE*, 47(7):28–32, 53, 2010.

[175] Susanna Pelagatti. *Structured development of parallel programs*. Taylor & Francis, Inc., Bristol, PA, USA, 1998.

[176] Josep M. Perez, Rosa M. Badia, and Jesus Labarta. A dependency-aware task-based programming environment for multi-core architectures. In *IEEE International Conference on Cluster Computing*, pages 142–151, Sept 2008.

[177] Josep M. Pérez, Pieter Bellens, Rosa M. Badia, and Jesus Labarta. CellSs: Making it easier to program the cell broadband engine processor. *IBM Journal of Research and Development*, 51(5):593–604, 2007.

[178] Phitchaya Mangpo Phothilimthana, Jason Ansel, Jonathan Ragan-Kelley, and Saman Amarasinghe. Portable performance on heterogeneous architectures. *SIGARCH Comput. Archit. News*, 41(1):431–444, 2013.

[179] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesus Labarta. Hierarchical task-based programming with StarSs. *Int. J. High Perform. Comput. Appl.*, 23(3):284–299, August 2009.

[180] Judit Planas, Rosa M. Badia, Eduard Ayguadé, and Jesús Labarta. Selection of task implementations in the Nanos++ runtime. Technical report, Barcelona Supercomputing Center, Barcelona, Spain, 2013. PRACE WP5.

[181] Artur Podobas, Mats Brorsson, and Vladimir Vlassov. Exploring heterogeneous scheduling using the task-centric programming model. In *Euro-Par 2012: Parallel Processing Workshops*, volume 7640 of *Lecture Notes in Computer Science*, pages 133–144. Springer Berlin Heidelberg, 2013.

[182] Dan Quinlan and Chunhua Liao. The ROSE source-to-source compiler infrastructure. In *Cetus Users and Compiler Infrastructure Workshop, in conjunction with PACT*, Galveston Island, Texas, USA, 2011.

[183] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[184] Ganesan Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[185] Manman Ren, Ji Young Park, Mike Houston, Alex Aiken, and William J. Dally. A tuning framework for software-managed

memory hierarchies. In *17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 280–291, New York, NY, USA, 2008. ACM.

[186] Greg Ruetsch and Massimiliano Fatica. CUDA Fortran for Scientists and Engineers. Technical report, NVIDIA Corporation, 2011. `http://www.phy.ohiou.edu/~jung/computer_club/cuda/Book-Fatica.pdf`.

[187] Shane Ryoo, Christopher I. Rodrigues, Sam S. Stone, Sara S. Baghsorkhi, Sain-Zee Ueng, John A. Stratton, and Wen-mei W. Hwu. Program optimization space pruning for a multithreaded GPU. In *6th Annual IEEE/ACM international symposium on Code generation and optimization*, CGO '08, pages 195–204, New York, NY, USA, 2008. ACM.

[188] SALOME: The open source integration platform for numerical simulation, 2013. SALOME version 7.3.0, `http://www.salome-platform.org`.

[189] Martin Sandrieser, Siegfried Benkner, and Sabri Pllana. Using explicit platform descriptions to support programming of heterogeneous many-core systems. *Parallel Computing*, 38(1-2):52–65, 2012.

[190] Amin S. Sarvestani, Erik Hansson, and Christoph Kessler. Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *International Journal of Parallel Programming*, 41(6):806–824, 2013.

[191] Mitsuhisa Sato. OpenMP: parallel programming API for shared memory multiprocessors and on-chip multiprocessors. In *15th International Symposium on System Synthesis*, pages 109–111, 2002.

[192] Tudor Serban, Marco Danelutto, and Peter Kilpatrick. Autonomic scheduling of tasks from data parallel patterns to cpu/gpu core mixes. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 72—79, July 2013.

[193] Gilad Shainer, Ali Ayoub, Pak Lui, Tong Liu, Michael Kagan, ChristianR. Trott, Greg Scantlen, and PaulS. Crozier. The development of Mellanox/NVIDIA GPUDirect over InfiniBand - A new model for GPU to GPU communications. *Computer Science - Research and Development*, 26(3-4):267–273, 2011.

[194] Bryan Singer and Manuela Veloso. Learning to construct fast signal processing implementations. *J. Mach. Learn. Res.*, 3:887–919, March 2003.

[195] David B. Skillicorn. Architecture-independent parallel computation. *Computer*, 23:38–50, December 1990.

[196] Michel Steuwer and Sergei Gorlatch. SkelCL: Enhancing OpenCL for high-level programming of multi-GPU systems. In *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 258–272. Springer Berlin Heidelberg, 2013.

[197] Michel Steuwer, Philipp Kegel, and Sergei Gorlatch. SkelCL - A portable skeleton library for high-level GPU programming. In *16th International Workshop on High-Level Parallel Programming Models and Supportive Environments*, HIPS '11, May 2011.

[198] Daniel Sykes, William Heaven, Jeff Magee, and Jeff Kramer. Exploiting non-functional preferences in architectural adaptation for self-managed systems. In *SAC '10: ACM Symposium on Applied Computing*, pages 431–438, New York, NY, USA, 2010. ACM.

[199] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 2002.

[200] Jocelyn Sérot and Dominique Ginhac. Skeletons for parallel image processing: An overview of the SKIPPER project. *Parallel Computing*, 28(12):1685–1708, 2002.

[201] Haruto Tanno and Hideya Iwasaki. Parallel skeletons for variable-length lists in SkeTo skeleton library. In *15th International Euro-Par Conference on Parallel Processing*, Euro-Par '09, pages 666–677, Berlin, Heidelberg, 2009. Springer-Verlag.

[202] Nathan Thomas, Gabriel Tanase, Olga Tkachyshyn, Jack Perdue, Nancy M. Amato, and Lawrence Rauchwerger. A framework for adaptive algorithm selection in STAPL. In *10th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '05, pages 277–288, New York, NY, USA, 2005. ACM.

[203] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Trans. Parallel Distrib. Syst.*, 13:260–274, 2002.

[204] Moisés Vinas, Zeki Bozkus, and Basilio B. Fraguela. Exploiting heterogeneous parallelism with the heterogeneous programming library. *Journal of Parallel and Distributed Computing*, 73(12):1627–1638, 2013.

[205] Nanbor Wang, Douglas C. Schmidt, and Carlos O'Ryan. *Component-based Software Engineering*, chapter Overview of the CORBA Component Model, pages 557–571. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[206] Zhuowei Wang, Xianbin Xu, Wuqing Zhao, Yuping Zhang, and Shuibing He. Optimizing sparse matrix-vector multiplication on CUDA. In *2nd International Conference on Education Technology and Computer*, volume 4 of *ICETC '10*, pages V4–109–V4–113, June 2010.

[207] Perry H. Wang et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Programming Language Design and Implementation (PLDI)*, pages 156–166. ACM, 2007.

[208] Ben Van Werkhoven, Jason Maassen, and Frank Seinstra. Optimizing convolutions operations in CUDA with adaptive tiling. In *2nd Workshop on Applications for Multi and Many Core Processors*, A4MMC '11, 2011.

[209] John Robert Wernsing and Greg Stitt. Elastic computing: a framework for transparent, portable, and adaptive multi-core heterogeneous computing. In *ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems (LCTES)*, pages 115–124. ACM, 2010.

[210] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[211] Jerry L. Whitten. Coulombic potential energy integrals and approximations. *The Journal of Chemical Physics*, 58:4496–4501, May 1973.

[212] Craig M. Wittenbrink, Emmett Kilgariff, and Arjun Prabhu. Fermi GF100 GPU architecture. *IEEE Micro*, 31(2):50–59, 2011.

[213] Yonghong Yan, Max Grossman, and Vivek Sarkar. JCUDA: A programmer-friendly interface for accelerating java programs with CUDA. In *Euro-Par 2009 Parallel Processing*, volume 5704 of *Lecture Notes in Computer Science*, pages 887–899. Springer Berlin Heidelberg, 2009.

[214] Cui Yu. *High-dimensional Indexing: Transformational Approaches to High-dimensional Range and Similarity Searches*. Springer-Verlag, Berlin, Heidelberg, 2002.

[215] Hao Yu and Lawrence Rauchwerger. An adaptive algorithm selection framework for reduction parallelization. *IEEE Transactions on Parallel and Distributed Systems*, 17(10):1084–1096, October 2006.

Department of Computer and Information Science
Linköpings universitet

Dissertations

Linköping Studies in Science and Technology
Linköping Studies in Arts and Science

Linköping Studies in Statistics
Linköpings Studies in Informatics

Linköping Studies in Science and Technology

No 14 **Anders Haraldsson:** A Program Manipulation System Based on Partial Evaluation, 1977, ISBN 91-7372-144-1.

No 17 **Bengt Magnhagen:** Probability Based Verification of Time Margins in Digital Designs, 1977, ISBN 91-7372-157-3.

No 18 **Mats Cedwall**: Semantisk analys av process-beskrivningar i naturligt språk, 1977, ISBN 91- 7372-168-9.

No 22 **Jaak Urmi:** A Machine Independent LISP Compiler and its Implications for Ideal Hardware, 1978, ISBN 91-7372-188-3.

No 33 **Tore Risch:** Compilation of Multiple File Queries in a Meta-Database System 1978, ISBN 91- 7372-232-4.

No 51 **Erland Jungert:** Synthesizing Database Structures from a User Oriented Data Model, 1980, ISBN 91-7372-387-8.

No 54 **Sture Hägglund:** Contributions to the Development of Methods and Tools for Interactive Design of Applications Software, 1980, ISBN 91-7372-404-1.

No 55 **Pär Emanuelson:** Performance Enhancement in a Well-Structured Pattern Matcher through Partial Evaluation, 1980, ISBN 91-7372-403-3.

No 58 **Bengt Johnsson, Bertil Andersson:** The Human-Computer Interface in Commercial Systems, 1981, ISBN 91-7372-414-9.

No 69 **H. Jan Komorowski:** A Specification of an Abstract Prolog Machine and its Application to Partial Evaluation, 1981, ISBN 91-7372-479-3.

No 71 **René Reboh:** Knowledge Engineering Techniques and Tools for Expert Systems, 1981, ISBN 91-7372-489-0.

No 77 **Östen Oskarsson:** Mechanisms of Modifiability in large Software Systems, 1982, ISBN 91- 7372-527-7.

No 94 **Hans Lunell:** Code Generator Writing Systems, 1983, ISBN 91-7372-652-4.

No 97 **Andrzej Lingas:** Advances in Minimum Weight Triangulation, 1983, ISBN 91-7372-660-5.

No 109 **Peter Fritzson:** Towards a Distributed Programming Environment based on Incremental Compilation, 1984, ISBN 91-7372-801-2.

No 111 **Erik Tengvald:** The Design of Expert Planning Systems. An Experimental Operations Planning System for Turning, 1984, ISBN 91-7372- 805-5.

No 155 **Christos Levcopoulos:** Heuristics for Minimum Decompositions of Polygons, 1987, ISBN 91-7870-133-3.

No 165 **James W. Goodwin:** A Theory and System for Non-Monotonic Reasoning, 1987, ISBN 91-7870-183-X.

No 170 **Zebo Peng:** A Formal Methodology for Automated Synthesis of VLSI Systems, 1987, ISBN 91-7870-225-9.

No 174 **Johan Fagerström:** A Paradigm and System for Design of Distributed Systems, 1988, ISBN 91-7870-301-8.

No 192 **Dimiter Driankov:** Towards a Many Valued Logic of Quantified Belief, 1988, ISBN 91-7870-374-3.

No 213 **Lin Padgham:** Non-Monotonic Inheritance for an Object Oriented Knowledge Base, 1989, ISBN 91-7870-485-5.

No 214 **Tony Larsson:** A Formal Hardware Description and Verification Method, 1989, ISBN 91-7870-517-7.

No 221 **Michael Reinfrank:** Fundamentals and Logical Foundations of Truth Maintenance, 1989, ISBN 91-7870-546-0.

No 239 **Jonas Löwgren:** Knowledge-Based Design Support and Discourse Management in User Interface Management Systems, 1991, ISBN 91-7870-720-X.

No 244 **Henrik Eriksson:** Meta-Tool Support for Knowledge Acquisition, 1991, ISBN 91-7870-746-3.

No 252 **Peter Eklund:** An Epistemic Approach to Interactive Design in Multiple Inheritance Hierarchies, 1991, ISBN 91-7870-784-6.

No 258 **Patrick Doherty:** NML3 - A Non-Monotonic Formalism with Explicit Defaults, 1991, ISBN 91-7870-816-8.

No 260 **Nahid Shahmehri:** Generalized Algorithmic Debugging, 1991, ISBN 91-7870-828-1.

No 264 **Nils Dahlbäck:** Representation of Discourse-Cognitive and Computational Aspects, 1992, ISBN 91-7870-850-8.

No 265 **Ulf Nilsson:** Abstract Interpretations and Abstract Machines: Contributions to a Methodology for the Implementation of Logic Programs, 1992, ISBN 91-7870-858-3.

No 270 **Ralph Rönnquist:** Theory and Practice of Tense-bound Object References, 1992, ISBN 91-7870-873-7.

No 273 **Björn Fjellborg:** Pipeline Extraction for VLSI Data Path Synthesis, 1992, ISBN 91-7870-880-X.

No 276 **Staffan Bonnier:** A Formal Basis for Horn Clause Logic with External Polymorphic Functions, 1992, ISBN 91-7870-896-6.

No 277 **Kristian Sandahl:** Developing Knowledge Management Systems with an Active Expert Methodology, 1992, ISBN 91-7870-897-4.

No 281 **Christer Bäckström:** Computational Complexity of Reasoning about Plans, 1992, ISBN 91-7870-979-2.

No 292 **Mats Wirén:** Studies in Incremental Natural Language Analysis, 1992, ISBN 91-7871-027-8.

No 297 **Mariam Kamkar:** Interprocedural Dynamic Slicing with Applications to Debugging and Testing, 1993, ISBN 91-7871-065-0.

No 302 **Tingting Zhang:** A Study in Diagnosis Using Classification and Defaults, 1993, ISBN 91-7871-078-2

No 312 **Arne Jönsson:** Dialogue Management for Natural Language Interfaces - An Empirical Approach, 1993, ISBN 91-7871-110-X.

No 338 **Simin Nadjm-Tehrani:** Reactive Systems in Physical Environments: Compositional Modelling and Framework for Verification, 1994, ISBN 91-7871-237-8.

No 371 **Bengt Savén:** Business Models for Decision Support and Learning. A Study of Discrete-Event Manufacturing Simulation at Asea/ ABB 1968-1993, 1995, ISBN 91-7871-494-X.

No 375 **Ulf Söderman:** Conceptual Modelling of Mode Switching Physical Systems, 1995, ISBN 91-7871-516-4.

No 383 **Andreas Kågedal:** Exploiting Groundness in Logic Programs, 1995, ISBN 91-7871-538-5.

No 396 **George Fodor:** Ontological Control, Description, Identification and Recovery from Problematic Control Situations, 1995, ISBN 91-7871-603-9.

No 413 **Mikael Pettersson:** Compiling Natural Semantics, 1995, ISBN 91-7871-641-1.

No 414 **Xinli Gu:** RT Level Testability Improvement by Testability Analysis and Transformations, 1996, ISBN 91-7871-654-3.

No 416 **Hua Shu:** Distributed Default Reasoning, 1996, ISBN 91-7871-665-9.

No 429 **Jaime Villegas:** Simulation Supported Industrial Training from an Organisational Learning Perspective - Development and Evaluation of the SSIT Method, 1996, ISBN 91-7871-700-0.

No 431 **Peter Jonsson:** Studies in Action Planning: Algorithms and Complexity, 1996, ISBN 91-7871-704-3.

No 437 **Johan Boye:** Directional Types in Logic Programming, 1996, ISBN 91-7871-725-6.

No 439 **Cecilia Sjöberg:** Activities, Voices and Arenas: Participatory Design in Practice, 1996, ISBN 91-7871-728-0.

No 448 **Patrick Lambrix:** Part-Whole Reasoning in Description Logics, 1996, ISBN 91-7871-820-1.

No 452 **Kjell Orsborn:** On Extensible and Object-Relational Database Technology for Finite Element Analysis Applications, 1996, ISBN 91-7871-827-9.

No 459 **Olof Johansson:** Development Environments for Complex Product Models, 1996, ISBN 91-7871-855-4.

No 461 **Lena Strömbäck:** User-Defined Constructions in Unification-Based Formalisms, 1997, ISBN 91-7871-857-0.

No 462 **Lars Degerstedt:** Tabulation-based Logic Programming: A Multi-Level View of Query Answering, 1996, ISBN 91-7871-858-9.

No 475 **Fredrik Nilsson:** Strategi och ekonomisk styrning - En studie av hur ekonomiska styrsystem utformas och används efter företagsförvärv, 1997, ISBN 91-7871-914-3.

No 480 **Mikael Lindvall:** An Empirical Study of Requirements-Driven Impact Analysis in Object-Oriented Software Evolution, 1997, ISBN 91-7871-927-5.

No 485 **Göran Forslund:** Opinion-Based Systems: The Cooperative Perspective on Knowledge-Based Decision Support, 1997, ISBN 91-7871-938-0.

No 494 **Martin Sköld:** Active Database Management Systems for Monitoring and Control, 1997, ISBN 91-7219-002-7.

No 495 **Hans Olsén:** Automatic Verification of Petri Nets in a CLP framework, 1997, ISBN 91-7219-011-6.

No 498 **Thomas Drakengren:** Algorithms and Complexity for Temporal and Spatial Formalisms, 1997, ISBN 91-7219-019-1.

No 502 **Jakob Axelsson:** Analysis and Synthesis of Heterogeneous Real-Time Systems, 1997, ISBN 91-7219-035-3.

No 503 **Johan Ringström:** Compiler Generation for Data-Parallel Programming Languages from Two-Level Semantics Specifications, 1997, ISBN 91-7219-045-0.

No 512 **Anna Moberg:** Närhet och distans - Studier av kommunikationsmönster i satellitkontor och flexibla kontor, 1997, ISBN 91-7219-119-8.

No 520 **Mikael Ronström:** Design and Modelling of a Parallel Data Server for Telecom Applications, 1998, ISBN 91-7219-169-4.

No 522 **Niclas Ohlsson:** Towards Effective Fault Prevention - An Empirical Study in Software Engineering, 1998, ISBN 91-7219-176-7.

No 526 **Joachim Karlsson:** A Systematic Approach for Prioritizing Software Requirements, 1998, ISBN 91-7219-184-8.

No 530 **Henrik Nilsson:** Declarative Debugging for Lazy Functional Languages, 1998, ISBN 91-7219-197-x.

No 555 **Jonas Hallberg:** Timing Issues in High-Level Synthesis, 1998, ISBN 91-7219-369-7.

No 561 **Ling Lin:** Management of 1-D Sequence Data - From Discrete to Continuous, 1999, ISBN 91-7219-402-2.

No 563 **Eva L Ragnemalm:** Student Modelling based on Collaborative Dialogue with a Learning Companion, 1999, ISBN 91-7219-412-X.

No 567 **Jörgen Lindström:** Does Distance matter? On geographical dispersion in organisations, 1999, ISBN 91-7219-439-1.

No 582 **Vanja Josifovski:** Design, Implementation and Evaluation of a Distributed Mediator System for Data Integration, 1999, ISBN 91-7219-482-0.

No 589 **Rita Kovordányi:** Modeling and Simulating Inhibitory Mechanisms in Mental Image Reinterpretation - Towards Cooperative Human-Computer Creativity, 1999, ISBN 91-7219-506-1.

No 592 **Mikael Ericsson:** Supporting the Use of Design Knowledge - An Assessment of Commenting Agents, 1999, ISBN 91-7219-532-0.

No 593 **Lars Karlsson:** Actions, Interactions and Narratives, 1999, ISBN 91-7219-534-7.

No 594 **C. G. Mikael Johansson:** Social and Organizational Aspects of Requirements Engineering Methods - A practice-oriented approach, 1999, ISBN 91-7219-541-X.

No 595 **Jörgen Hansson:** Value-Driven Multi-Class Overload Management in Real-Time Database Systems, 1999, ISBN 91-7219-542-8.

No 596 **Niklas Hallberg:** Incorporating User Values in the Design of Information Systems and Services in the Public Sector: A Methods Approach, 1999, ISBN 91-7219-543-6.

No 597 **Vivian Vimarlund:** An Economic Perspective on the Analysis of Impacts of Information Technology: From Case Studies in Health-Care towards General Models and Theories, 1999, ISBN 91-7219-544-4.

No 598 **Johan Jenvald:** Methods and Tools in Computer-Supported Taskforce Training, 1999, ISBN 91-7219-547-9.

No 607 **Magnus Merkel:** Understanding and enhancing translation by parallel text processing, 1999, ISBN 91-7219-614-9.

No 611 **Silvia Coradeschi:** Anchoring symbols to sensory data, 1999, ISBN 91-7219-623-8.

No 613 **Man Lin:** Analysis and Synthesis of Reactive Systems: A Generic Layered Architecture Perspective, 1999, ISBN 91-7219-630-0.

No 618 **Jimmy Tjäder:** Systemimplementering i praktiken - En studie av logiker i fyra projekt, 1999, ISBN 91-7219-657-2.

No 627 **Vadim Engelson:** Tools for Design, Interactive Simulation, and Visualization of Object-Oriented Models in Scientific Computing, 2000, ISBN 91-7219-709-9.

No 637 **Esa Falkenroth:** Database Technology for Control and Simulation, 2000, ISBN 91-7219-766-8.

No 639 **Per-Arne Persson:** Bringing Power and Knowledge Together: Information Systems Design for Autonomy and Control in Command Work, 2000, ISBN 91-7219-796-X.

No 660 **Erik Larsson:** An Integrated System-Level Design for Testability Methodology, 2000, ISBN 91-7219-890-7.

No 688 **Marcus Bjäreland:** Model-based Execution Monitoring, 2001, ISBN 91-7373-016-5.

No 689 **Joakim Gustafsson:** Extending Temporal Action Logic, 2001, ISBN 91-7373-017-3.

No 720 **Carl-Johan Petri:** Organizational Information Provision - Managing Mandatory and Discretionary Use of Information Technology, 2001, ISBN-91-7373-126-9.

No 724 **Paul Scerri:** Designing Agents for Systems with Adjustable Autonomy, 2001, ISBN 91 7373 207 9.

No 725 **Tim Heyer:** Semantic Inspection of Software Artifacts: From Theory to Practice, 2001, ISBN 91 7373 208 7.

No 726 **Pär Carlshamre:** A Usability Perspective on Requirements Engineering - From Methodology to Product Development, 2001, ISBN 91 7373 212 5.

No 732 **Juha Takkinen:** From Information Management to Task Management in Electronic Mail, 2002, ISBN 91 7373 258 3.

No 745 **Johan Åberg:** Live Help Systems: An Approach to Intelligent Help for Web Information Systems, 2002, ISBN 91-7373-311-3.

No 746 **Rego Granlund:** Monitoring Distributed Teamwork Training, 2002, ISBN 91-7373-312-1.

No 757 **Henrik André-Jönsson:** Indexing Strategies for Time Series Data, 2002, ISBN 917373-346-6.

No 747 **Anneli Hagdahl:** Development of IT-supported Interorganisational Collaboration - A Case Study in the Swedish Public Sector, 2002, ISBN 91-7373-314-8.

No 749 **Sofie Pilemalm:** Information Technology for Non-Profit Organisations - Extended Participatory Design of an Information System for Trade Union Shop Stewards, 2002, ISBN 91-7373-318-0.

No 765 **Stefan Holmlid:** Adapting users: Towards a theory of use quality, 2002, ISBN 91-7373-397-0.

No 771 **Magnus Morin:** Multimedia Representations of Distributed Tactical Operations, 2002, ISBN 91-7373-421-7.

No 772 **Pawel Pietrzak:** A Type-Based Framework for Locating Errors in Constraint Logic Programs, 2002, ISBN 91-7373-422-5.

No 758 **Erik Berglund:** Library Communication Among Programmers Worldwide, 2002, ISBN 91-7373-349-0.

No 774 **Choong-ho Yi:** Modelling Object-Oriented Dynamic Systems Using a Logic-Based Framework, 2002, ISBN 91-7373-424-1.

No 779 **Mathias Broxvall:** A Study in the Computational Complexity of Temporal Reasoning, 2002, ISBN 91-7373-440-3.

No 793 **Asmus Pandikow:** A Generic Principle for Enabling Interoperability of Structured and Object-Oriented Analysis and Design Tools, 2002, ISBN 91-7373-479-9.

No 785 **Lars Hult:** Publika Informationstjänster. En studie av den Internetbaserade encyklopedins bruksegenskaper, 2003, ISBN 91-7373-461-6.

No 800 **Lars Taxén:** A Framework for the Coordination of Complex Systems´ Development, 2003, ISBN 91-7373-604-X

No 808 **Klas Gäre:** Tre perspektiv på förväntningar och förändringar i samband med införande av informationssystem, 2003, ISBN 91-7373-618-X.

No 821 **Mikael Kindborg:** Concurrent Comics - programming of social agents by children, 2003, ISBN 91-7373-651-1.

No 823 **Christina Ölvingson:** On Development of Information Systems with GIS Functionality in Public Health Informatics: A Requirements Engineering Approach, 2003, ISBN 91-7373-656-2.

No 828 **Tobias Ritzau:** Memory Efficient Hard Real-Time Garbage Collection, 2003, ISBN 91-7373-666-X.

No 833 **Paul Pop:** Analysis and Synthesis of Communication-Intensive Heterogeneous Real-Time Systems, 2003, ISBN 91-7373-683-X.

No 852 **Johan Moe:** Observing the Dynamic Behaviour of Large Distributed Systems to Improve Development and Testing – An Empirical Study in Software Engineering, 2003, ISBN 91-7373-779-8.

No 867 **Erik Herzog:** An Approach to Systems Engineering Tool Data Representation and Exchange, 2004, ISBN 91-7373-929-4.

No 872 **Aseel Berglund:** Augmenting the Remote Control: Studies in Complex Information Navigation for Digital TV, 2004, ISBN 91-7373-940-5.

No 869 **Jo Skåmedal:** Telecommuting's Implications on Travel and Travel Patterns, 2004, ISBN 91-7373-935-9.

No 870 **Linda Askenäs:** The Roles of IT - Studies of Organising when Implementing and Using Enterprise Systems, 2004, ISBN 91-7373-936-7.

No 874 **Annika Flycht-Eriksson:** Design and Use of Ontologies in Information-Providing Dialogue Systems, 2004, ISBN 91-7373-947-2.

No 873 **Peter Bunus:** Debugging Techniques for Equation-Based Languages, 2004, ISBN 91-7373-941-3.

No 876 **Jonas Mellin:** Resource-Predictable and Efficient Monitoring of Events, 2004, ISBN 91-7373-956-1.

No 883 **Magnus Bång:** Computing at the Speed of Paper: Ubiquitous Computing Environments for Healthcare Professionals, 2004, ISBN 91-7373-971-5

No 882 **Robert Eklund:** Disfluency in Swedish human-human and human-machine travel booking dialogues, 2004, ISBN 91-7373-966-9.

No 887 **Anders Lindström:** English and other Foreign Linguistic Elements in Spoken Swedish. Studies of Productive Processes and their Modelling using Finite-State Tools, 2004, ISBN 91-7373-981-2.

No 889 **Zhiping Wang:** Capacity-Constrained Production-inventory systems - Modelling and Analysis in both a traditional and an e-business context, 2004, ISBN 91-85295-08-6.

No 893 **Pernilla Qvarfordt:** Eyes on Multimodal Interaction, 2004, ISBN 91-85295-30-2.

No 910 **Magnus Kald:** In the Borderland between Strategy and Management Control - Theoretical Framework and Empirical Evidence, 2004, ISBN 91-85295-82-5.

No 918 **Jonas Lundberg:** Shaping Electronic News: Genre Perspectives on Interaction Design, 2004, ISBN 91-85297-14-3.

No 900 **Mattias Arvola:** Shades of use: The dynamics of interaction design for sociable use, 2004, ISBN 91-85295-42-6.

No 920 **Luis Alejandro Cortés:** Verification and Scheduling Techniques for Real-Time Embedded Systems, 2004, ISBN 91-85297-21-6.

No 929 **Diana Szentivanyi:** Performance Studies of Fault-Tolerant Middleware, 2005, ISBN 91-85297-58-5.

No 933 **Mikael Cäker:** Management Accounting as Constructing and Opposing Customer Focus: Three Case Studies on Management Accounting and Customer Relations, 2005, ISBN 91-85297-64-X.

No 937 **Jonas Kvarnström:** TALplanner and Other Extensions to Temporal Action Logic, 2005, ISBN 91-85297-75-5.

No 938 **Bourhane Kadmiry:** Fuzzy Gain-Scheduled Visual Servoing for Unmanned Helicopter, 2005, ISBN 91-85297-76-3.

No 945 **Gert Jervan:** Hybrid Built-In Self-Test and Test Generation Techniques for Digital Systems, 2005, ISBN: 91-85297-97-6.

No 946 **Anders Arpteg:** Intelligent Semi-Structured Information Extraction, 2005, ISBN 91-85297-98-4.

No 947 **Ola Angelsmark:** Constructing Algorithms for Constraint Satisfaction and Related Problems - Methods and Applications, 2005, ISBN 91-85297-99-2.

No 963 **Calin Curescu:** Utility-based Optimisation of Resource Allocation for Wireless Networks, 2005, ISBN 91-85457-07-8.

No 972 **Björn Johansson:** Joint Control in Dynamic Situations, 2005, ISBN 91-85457-31-0.

No 974 **Dan Lawesson:** An Approach to Diagnosability Analysis for Interacting Finite State Systems, 2005, ISBN 91-85457-39-6.

No 979 **Claudiu Duma:** Security and Trust Mechanisms for Groups in Distributed Services, 2005, ISBN 91-85457-54-X.

No 983 **Sorin Manolache:** Analysis and Optimisation of Real-Time Systems with Stochastic Behaviour, 2005, ISBN 91-85457-60-4.

No 986 **Yuxiao Zhao:** Standards-Based Application Integration for Business-to-Business Communications, 2005, ISBN 91-85457-66-3.

No 1004 **Patrik Haslum:** Admissible Heuristics for Automated Planning, 2006, ISBN 91-85497-28-2.

No 1005 **Aleksandra Tešanovic:** Developing Reusable and Reconfigurable Real-Time Software using Aspects and Components, 2006, ISBN 91-85497-29-0.

No 1008 **David Dinka:** Role, Identity and Work: Extending the design and development agenda, 2006, ISBN 91-85497-42-8.

No 1009 **Iakov Nakhimovski:** Contributions to the Modeling and Simulation of Mechanical Systems with Detailed Contact Analysis, 2006, ISBN 91-85497-43-X.

No 1013 **Wilhelm Dahllöf:** Exact Algorithms for Exact Satisfiability Problems, 2006, ISBN 91-85523-97-6.

No 1016 **Levon Saldamli:** PDEModelica - A High-Level Language for Modeling with Partial Differential Equations, 2006, ISBN 91-85523-84-4.

No 1017 **Daniel Karlsson:** Verification of Component-based Embedded System Designs, 2006, ISBN 91-85523-79-8

No 1018 **Ioan Chisalita:** Communication and Networking Techniques for Traffic Safety Systems, 2006, ISBN 91-85523-77-1.

No 1019 **Tarja Susi:** The Puzzle of Social Activity - The Significance of Tools in Cognition and Cooperation, 2006, ISBN 91-85523-71-2.

No 1021 **Andrzej Bednarski:** Integrated Optimal Code Generation for Digital Signal Processors, 2006, ISBN 91-85523-69-0.

No 1022 **Peter Aronsson:** Automatic Parallelization of Equation-Based Simulation Programs, 2006, ISBN 91-85523-68-2.

No 1030 **Robert Nilsson:** A Mutation-based Framework for Automated Testing of Timeliness, 2006, ISBN 91-85523-35-6.

No 1034 **Jon Edvardsson:** Techniques for Automatic Generation of Tests from Programs and Specifications, 2006, ISBN 91-85523-31-3.

No 1035 **Vaida Jakoniene:** Integration of Biological Data, 2006, ISBN 91-85523-28-3.

No 1045 **Genevieve Gorrell:** Generalized Hebbian Algorithms for Dimensionality Reduction in Natural Language Processing, 2006, ISBN 91-85643-88-2.

No 1051 **Yu-Hsing Huang:** Having a New Pair of Glasses - Applying Systemic Accident Models on Road Safety, 2006, ISBN 91-85643-64-5.

No 1054 **Åsa Hedenskog:** Perceive those things which cannot be seen - A Cognitive Systems Engineering perspective on requirements management, 2006, ISBN 91-85643-57-2.

No 1061 **Cécile Åberg:** An Evaluation Platform for Semantic Web Technology, 2007, ISBN 91-85643-31-9.

No 1073 **Mats Grindal:** Handling Combinatorial Explosion in Software Testing, 2007, ISBN 978-91-85715-74-9.

No 1075 **Almut Herzog:** Usable Security Policies for Runtime Environments, 2007, ISBN 978-91-85715-65-7.

No 1079 **Magnus Wahlström:** Algorithms, measures, and upper bounds for Satisfiability and related problems, 2007, ISBN 978-91-85715-55-8.

No 1083 **Jesper Andersson:** Dynamic Software Architectures, 2007, ISBN 978-91-85715-46-6.

No 1086 **Ulf Johansson:** Obtaining Accurate and Comprehensible Data Mining Models - An Evolutionary Approach, 2007, ISBN 978-91-85715-34-3.

No 1089 **Traian Pop:** Analysis and Optimisation of Distributed Embedded Systems with Heterogeneous Scheduling Policies, 2007, ISBN 978-91-85715-27-5.

No 1091 **Gustav Nordh:** Complexity Dichotomies for CSP-related Problems, 2007, ISBN 978-91-85715-20-6.

No 1106 **Per Ola Kristensson:** Discrete and Continuous Shape Writing for Text Entry and Control, 2007, ISBN 978-91-85831-77-7.

No 1110 **He Tan:** Aligning Biomedical Ontologies, 2007, ISBN 978-91-85831-56-2.

No 1112 **Jessica Lindblom:** Minding the body - Interacting socially through embodied action, 2007, ISBN 978-91-85831-48-7.

No 1113 **Pontus Wärnestål:** Dialogue Behavior Management in Conversational Recommender Systems, 2007, ISBN 978-91-85831-47-0.

No 1120 **Thomas Gustafsson:** Management of Real-Time Data Consistency and Transient Overloads in Embedded Systems, 2007, ISBN 978-91-85831-33-3.

No 1127 **Alexandru Andrei:** Energy Efficient and Predictable Design of Real-time Embedded Systems, 2007, ISBN 978-91-85831-06-7.

No 1139 **Per Wikberg:** Eliciting Knowledge from Experts in Modeling of Complex Systems: Managing Variation and Interactions, 2007, ISBN 978-91-85895-66-3.

No 1143 **Mehdi Amirijoo:** QoS Control of Real-Time Data Services under Uncertain Workload, 2007, ISBN 978-91-85895-49-6.

No 1150 **Sanny Syberfeldt:** Optimistic Replication with Forward Conflict Resolution in Distributed Real-Time Databases, 2007, ISBN 978-91-85895-27-4.

No 1155 **Beatrice Alenljung:** Envisioning a Future Decision Support System for Requirements Engineering - A Holistic and Human-centred Perspective, 2008, ISBN 978-91-85895-11-3.

No 1156 **Artur Wilk:** Types for XML with Application to Xcerpt, 2008, ISBN 978-91-85895-08-3.

No 1183 **Adrian Pop:** Integrated Model-Driven Development Environments for Equation-Based Object-Oriented Languages, 2008, ISBN 978-91-7393-895-2.

No 1185 **Jörgen Skågeby:** Gifting Technologies - Ethnographic Studies of End-users and Social Media Sharing, 2008, ISBN 978-91-7393-892-1.

No 1187 **Imad-Eldin Ali Abugessaisa:** Analytical tools and information-sharing methods supporting road safety organizations, 2008, ISBN 978-91-7393-887-7.

No 1204 **H. Joe Steinhauer:** A Representation Scheme for Description and Reconstruction of Object Configurations Based on Qualitative Relations, 2008, ISBN 978-91-7393-823-5.

No 1222 **Anders Larsson:** Test Optimization for Core-based System-on-Chip, 2008, ISBN 978-91-7393-768-9.

No 1238 **Andreas Borg:** Processes and Models for Capacity Requirements in Telecommunication Systems, 2009, ISBN 978-91-7393-700-9.

No 1240 **Fredrik Heintz:** DyKnow: A Stream-Based Knowledge Processing Middleware Framework, 2009, ISBN 978-91-7393-696-5.

No 1241 **Birgitta Lindström:** Testability of Dynamic Real-Time Systems, 2009, ISBN 978-91-7393-695-8.

No 1244 **Eva Blomqvist:** Semi-automatic Ontology Construction based on Patterns, 2009, ISBN 978-91-7393-683-5.

No 1249 **Rogier Woltjer:** Functional Modeling of Constraint Management in Aviation Safety and Command and Control, 2009, ISBN 978-91-7393-659-0.

No 1260 **Gianpaolo Conte:** Vision-Based Localization and Guidance for Unmanned Aerial Vehicles, 2009, ISBN 978-91-7393-603-3.

No 1262 **AnnMarie Ericsson:** Enabling Tool Support for Formal Analysis of ECA Rules, 2009, ISBN 978-91-7393-598-2.

No 1266 **Jiri Trnka:** Exploring Tactical Command and Control: A Role-Playing Simulation Approach, 2009, ISBN 978-91-7393-571-5.

No 1268 **Bahlol Rahimi:** Supporting Collaborative Work through ICT - How End-users Think of and Adopt Integrated Health Information Systems, 2009, ISBN 978-91-7393-550-0.

No 1274 **Fredrik Kuivinen:** Algorithms and Hardness Results for Some Valued CSPs, 2009, ISBN 978-91-7393-525-8.

No 1281 **Gunnar Mathiason:** Virtual Full Replication for Scalable Distributed Real-Time Databases, 2009, ISBN 978-91-7393-503-6.

No 1290 **Viacheslav Izosimov:** Scheduling and Optimization of Fault-Tolerant Distributed Embedded Systems, 2009, ISBN 978-91-7393-482-4.

No 1294 **Johan Thapper:** Aspects of a Constraint Optimisation Problem, 2010, ISBN 978-91-7393-464-0.

No 1306 **Susanna Nilsson:** Augmentation in the Wild: User Centered Development and Evaluation of Augmented Reality Applications, 2010, ISBN 978-91-7393-416-9.

No 1313 **Christer Thörn:** On the Quality of Feature Models, 2010, ISBN 978-91-7393-394-0.

No 1321 **Zhiyuan He:** Temperature Aware and Defect-Probability Driven Test Scheduling for System-on-Chip, 2010, ISBN 978-91-7393-378-0.

No 1333 **David Broman:** Meta-Languages and Semantics for Equation-Based Modeling and Simulation, 2010, ISBN 978-91-7393-335-3.

No 1337 **Alexander Siemers:** Contributions to Modelling and Visualisation of Multibody Systems Simulations with Detailed Contact Analysis, 2010, ISBN 978-91-7393-317-9.

No 1354 **Mikael Asplund:** Disconnected Discoveries: Availability Studies in Partitioned Networks, 2010, ISBN 978-91-7393-278-3.

No 1359 **Jana Rambusch**: Mind Games Extended: Understanding Gameplay as Situated Activity, 2010, ISBN 978-91-7393-252-3.

No 1373 **Sonia Sangari**: Head Movement Correlates to Focus Assignment in Swedish,2011,ISBN 978-91-7393-154-0.

No 1374 **Jan-Erik Källhammer**: Using False Alarms when Developing Automotive Active Safety Systems, 2011, ISBN 978-91-7393-153-3.

No 1375 **Mattias Eriksson**: Integrated Code Generation, 2011, ISBN 978-91-7393-147-2.

No 1381 **Ola Leifler**: Affordances and Constraints of Intelligent Decision Support for Military Command and Control – Three Case Studies of Support Systems, 2011, ISBN 978-91-7393-133-5.

No 1386 **Soheil Samii**: Quality-Driven Synthesis and Optimization of Embedded Control Systems, 2011, ISBN 978-91-7393-102-1.

No 1419 **Erik Kuiper**: Geographic Routing in Intermittently-connected Mobile Ad Hoc Networks: Algorithms and Performance Models, 2012, ISBN 978-91-7519-981-8.

No 1451 **Sara Stymne**: Text Harmonization Strategies for Phrase-Based Statistical Machine Translation, 2012, ISBN 978-91-7519-887-3.

No 1455 **Alberto Montebelli**: Modeling the Role of Energy Management in Embodied Cognition, 2012, ISBN 978-91-7519-882-8.

No 1465 **Mohammad Saifullah**: Biologically-Based Interactive Neural Network Models for Visual Attention and Object Recognition, 2012, ISBN 978-91-7519-838-5.

No 1490 **Tomas Bengtsson**: Testing and Logic Optimization Techniques for Systems on Chip, 2012, ISBN 978-91-7519-742-5.

No 1481 **David Byers**: Improving Software Security by Preventing Known Vulnerabilities, 2012, ISBN 978-91-7519-784-5.

No 1496 **Tommy Färnqvist**: Exploiting Structure in CSP-related Problems, 2013, ISBN 978-91-7519-711-1.

No 1503 **John Wilander**: Contributions to Specification, Implementation, and Execution of Secure Software, 2013, ISBN 978-91-7519-681-7.

No 1506 **Magnus Ingmarsson**: Creating and Enabling the Useful Service Discovery Experience, 2013, ISBN 978-91-7519-662-6.

No 1547 **Wladimir Schamai**: Model-Based Verification of Dynamic System Behavior against Requirements: Method, Language, and Tool, 2013, ISBN 978-91-7519-505-6.

No 1551 **Henrik Svensson**: Simulations, 2013, ISBN 978-91-7519-491-2.

No 1559 **Sergiu Rafiliu**: Stability of Adaptive Distributed Real-Time Systems with Dynamic Resource Management, 2013, ISBN 978-91-7519-471-4.

No 1581 **Usman Dastgeer**: Performance-aware Component Composition for GPU-based Systems, 2014, ISBN 978-91-7519-383-0.

Linköping Studies in Arts and Science

No 504 **Ing-Marie Jonsson:** Social and Emotional Characteristics of Speech-based In-Vehicle

Information Systems: Impact on Attitude and Driving Behaviour, 2009, ISBN 978-91-7393-478-7.

No 586 **Fabian Segelström:** Stakeholder Engagement for Service Design: How service designers identify and communicate insights, 2013, ISBN 978-91-7519-554-4.

Linköping Studies in Statistics

No 9 **Davood Shahsavani:** Computer Experiments Designed to Explore and Approximate Complex Deterministic Models, 2008, ISBN 978-91-7393-976-8.

No 10 **Karl Wahlin:** Roadmap for Trend Detection and Assessment of Data Quality, 2008, ISBN 978-91-7393-792-4.

No 11 **Oleg Sysoev:** Monotonic regression for large multivariate datasets, 2010, ISBN 978-91-7393-412-1.

No 13 **Agné Burauskaite-Harju:** Characterizing Temporal Change and Inter-Site Correlations in Daily and Subdaily Precipitation Extremes, 2011, ISBN 978-91-7393-110-6.

Linköping Studies in Information Science

No 1 **Karin Axelsson:** Metodisk systemstrukturering- att skapa samstämmighet mellan informationssystemarkitektur och verksamhet, 1998. ISBN-9172-19-296-8.

No 2 **Stefan Cronholm:** Metodverktyg och användbarhet - en studie av datorstödd metodbaserad systemutveckling, 1998, ISBN-9172-19-299-2.

No 3 **Anders Avdic:** Användare och utvecklare - om anveckling med kalkylprogram, 1999. ISBN-91-7219-606-8.

No 4 **Owen Eriksson:** Kommunikationskvalitet hos informationssystem och affärsprocesser, 2000, ISBN 91-7219-811-7.

No 5 **Mikael Lind:** Från system till process - kriterier för processbestämning vid verksamhetsanalys, 2001, ISBN 91-7373-067-X.

No 6 **Ulf Melin:** Koordination och informationssystem i företag och nätverk, 2002, ISBN 91-7373-278-8.

No 7 **Pär J. Ågerfalk:** Information Systems Actability - Understanding Information Technology as a Tool for Business Action and Communication, 2003, ISBN 91-7373-628-7.

No 8 **Ulf Seigerroth:** Att förstå och förändra systemutvecklingsverksamheter - en taxonomi för metautveckling, 2003, ISBN91-7373-736-4.

No 9 **Karin Hedström:** Spår av datoriseringens värden – Effekter av IT i äldreomsorg, 2004, ISBN 91-7373-963-4.

No 10 **Ewa Braf:** Knowledge Demanded for Action - Studies on Knowledge Mediation in Organisations, 2004, ISBN 91-85295-47-7.

No 11 **Fredrik Karlsson:** Method Configuration method and computerized tool support, 2005, ISBN 91-85297-48-8.

No 12 **Malin Nordström:** Styrbar systemförvaltning - Att organisera systemförvaltningsverksamhet med hjälp av effektiva förvaltningsobjekt, 2005, ISBN 91-85297-60-7.

No 13 **Stefan Holgersson:** Yrke: POLIS - Yrkeskunskap, motivation, IT-system och andra förutsättningar för polisarbete, 2005, ISBN 91-85299-43-X.

No 14 **Benneth Christiansson, Marie-Therese Christiansson:** Mötet mellan process och komponent - mot ett ramverk för en verksamhetsnära kravspecifikation vid anskaffning av komponentbaserade informationssystem, 2006, ISBN 91-85643-22-X.