Front. Comput. Sci., 2020, 14(5): 145316
https://doi.org/10.1007/s11704-019-8184-3

RESEARCH ARTICLE

An efficient GPU-based parallel tabu search algorithm for hardware/software co-design

Neng HOU[1,2], Fazhi HE (✉)[1], Yi ZHOU[3], Yilin CHEN[1]

1 School of Computer Science, Wuhan University, Wuhan 430072, China
2 School of Computer Science, Yangtze University, Jingzhou 434023, China
3 School of Information Science and Engineering, Wuhan University of Science and Technology, Wuhan 430081, China

Abstract Hardware/software partitioning is an essential step in hardware/software co-design. For large size problems, it is difficult to consider both solution quality and time. This paper presents an efficient GPU-based parallel tabu search algorithm (GPTS) for HW/SW partitioning. A single GPU kernel of compacting neighborhood is proposed to reduce the amount of GPU global memory accesses theoretically. A kernel fusion strategy is further proposed to reduce the amount of GPU global memory accesses of GPTS. To further minimize the transfer overhead of GPTS between CPU and GPU, an optimized transfer strategy for GPU-based tabu evaluation is proposed, which considers that all the candidates do not satisfy the given constraint. Experiments show that GPTS outperforms state-of-the-art work of tabu search and is competitive with other methods for HW/SW partitioning. The proposed parallelization is significant when considering the ordinary GPU platform.

Keywords hardware/software co-design, hardware/ software partitioning, graphics processing unit, GPU-based parallel tabu search, single kernel implementation, kernel fusion strategy, optimized transfer strategy

1 Introduction

In embedded systems, a hardware platform is made up of the predominant digital components, which execute application programs. Early in the 1980s, hardware/software co-design (HW/SW co-design) had become a hot topic [1]. In HW/SW co-design, HW/SW partitioning determines which functions to be implemented in hardware and which ones in software. HW/SW partitioning improves the overall performance of embedded systems under the given constraints [2].

In the early stage, the software component refers to general-purpose processing unit, such as RISC-CPU. The hardware component refers to application specification integrated circuit (ASIC). With the emergence of heterogeneous system architectures mixing multiple micro-processors (MP-SoC), digital signal processors (DSP), application-specific instruction set processors (ASIPs), HW/SW partitioning is becoming increasingly complicated [3,4].

There are various metrics (or objects) for HW/SW partitioning, such as energy consumption, area, heat dissipation, cost, etc. Most of metrics are conflicting, hence it is a general way to select one or several typical metrics. According to number of metrics, HW/SW partitioning can be treated as single-objective optimization or multi-objective optimization [5,6].

At the abstract level, the application is represented as task graph with fixed or flexible granularity [7,8]. The topology of task graph also has effects on the partitioning performance. Hence, finding a reducible sub-graph is an effective way to improve the performance [9].

Reference [10] did not aim at partitioning for a given architecture, nor did it present a complete co-design environment.

Received May 14, 2018; accepted August 12, 2019

E-mail: fzhe@whu.edu.cn

Instead, HW/SW partitioning was taken a general description. The system was modeled as an undirected communication graph. Based on it, two different partitioning problems were categorized. One was solved optimally in polynomial time complexity, while the other was NP-hard in strong sense [11,12].

Reference [13] considered that existing HW/SW partitioning problems relied on the approximate estimations of system performance. Therefore, uncertainty theory was utilized to build a generalized problem formulation for resource sharing as well as HW/SW partitioning for embedded systems.

On the algorithmic aspects, when the problem size is small, exact algorithms are used to obtain exact solutions [6,14,15]. When the problem size becomes large, finding the exact solution in reasonable time becomes impractical. Therefore, heuristics become popular alternatives to obtain approximate solutions within the acceptable running time [16–18].

Two classical and complementary domain-specific heuristics at the early stage of HW/SW co-design are noteworthy [19,20]. They were identified as software- oriented approach and hardware-oriented approach. Later on, the general heuristics, such as genetic algorithm [5,16,21], ant colony algorithm [22,23], artificial bees [24], particle swarm optimization [25], simulated annealing [8,26], artificial immune algorithm [27], tabu search [6], and the hybridization of these methods [28–30] were used for HW/SW partitioning. The custom heuristics [31,32] were also proposed for the problem.

Futhermore, parallelizing them to accelerate the solving process is a good option. However, the limitation of the availability and usefulness of parallel computing platforms makes parallel method for HW/SW partitioning scarce. The early researches only include parallel genetic algorithm [33] and parallel particle swarm optimization algorithm [34] for HW/SW partitioning. Both works were implemented on the cluster platform.

Nowadays, muti-core CPU and many-core GPU are widely available and have become a popular and promising parallel computing platform with low power-consuming and high ratio of performance to price. They are playing an important role in science and engineering domains [35–38], such as GPU-based search for traveling salesman problem [39,40], quadratic assignment problem [41], permutation flow shop problem [42] and resource constrained project scheduling [43].

In our previous work, we proposed a tabu search for HW/SW partitioning on GPU [44]. The proposed method could not deal with very large HW/SW partitioning due to the GPU resource limitation, which was then addressed by adding an additional procedure in our second paper [45].

However, since the processes of candidate cost computing and neighborhood compaction on GPU were implemented in a separate way, neither of the previous methods were efficient. This drawback was caused by previous neighborhood compaction, which needed three GPU kernels to finish the whole process. Meanwhile, the efficiency of tabu evaluation on GPU can be further improved.

New technical contributions of this paper on both algorithm framework and details are as follows: a single GPU kernel of compacting neighborhood is proposed. Based on this new implementation, a kernel fusion strategy is further proposed. To further minimize the transfer overhead between CPU and GPU, an optimized transfer strategy for GPU-based tabu evaluation is proposed. To testify the effectiveness of proposed contributions, we design experiments in corresponding sections. For the adaptive strategy, we consider the situation in which all the candidates do not satisfy the given constraint. To highlight the contribution of proposed transfer strategy, the process of tabu evaluation on GPU is analyzed in details.

The remainder of this manuscript is organized as follows. Section 2 introduces the pre-requisites. In Section 3, we present the overview of our method and explore the optimization strategies and implementation details for HW/SW partitioning. In Section 4, enough experiments are conducted to testify our method. Finally, Section 5 discusses the conclusion and future work.

2 Prerequisite

There have been various partitioning models for different hardware platforms and performance metrics. In our work, we research on a fundamental model, which reflects the typical concerns in HW/SW co-design. Based on the fundamental model, the successful experience of our GPU- accelerated algorithm has the generality, which will be helpful in other complex HW/SW co-design.

In [10,11], the application is represented as an undirected graph $G(V, E)$. $V = \{v_1, v_2, \ldots, v_n\}$ denotes the tasks. Each task includes hardware cost $h(v_i)$ and software cost $s(v_i)$. $c(v_i, v_j)$ in the edge set E indicates the communication cost when the two adjacent tasks are in different context (hardware or software). $P = \{V_H, V_S\}$ is a HW/SW partitioning, if it satisfies $V_H \cap V_S = \Phi$ and $V_H \cup V_S = V$. The edge set of P is $E_P = \{(v_i, v_j) | v_i \in V_H, v_j \in V_S \text{ or } v_i \in V_S, v_j \in V_H\}$. In P,

there exists three costs, namely hardware cost H_p, software cost S_p, and communication cost C_p. The total cost of P is defined as $T_p = \alpha H_p + \beta S_p + \gamma C_p$, where α, β and γ are the non- negative weights of three costs and they reflect the relative importance of the three metrics. Based on these costs, Arató et al. in [11] defined two partitioning problems.

Problem $\mathbf{P}_0$ Given a graph G with the cost function s, h, c and the constant α, β, $\gamma \geqslant 0$, find a hardware/software partitioning P with minimum T_P.

Problem P Given a graph G with the cost function s, h, c and $R \geqslant 0$. Finding a hardware/software partitioning P with $S_P + C_P \leqslant R$ that minimize H_P.

As in [11, 46] illustrated, $\mathbf{P}_0$ can be solved optimally in polynomial time, but **P** is NP hard. In our work, we focus on problem **P**.

In problem **P**, hardware cost refers to area utilization or power consumption while software cost refers to timing delay caused by implementing in software plus timing delay between hardware and software, which are both time-dimensional. Problem **P** makes sense because the tradeoff between hardware cost and timing delay is a general problem in the real world [13].

Let $\boldsymbol{x} = (x_1, x_2, \ldots, x_n)$ denote a bi-partitioning of problem P. $x_i = 1 (x_i = 0)$ indicates that v_i is assigned to software (hardware). The hardware cost $H(\boldsymbol{x})$, software cost $S(\boldsymbol{x})$ and communication cost $C(\boldsymbol{x})$ are formulated as

$$H(\boldsymbol{x}) = \sum_{i=1}^{n} h_i(1 - x_i), \tag{1}$$

$$S(\boldsymbol{x}) = \sum_{i=1}^{n} s_i x_i, \tag{2}$$

$$C(\boldsymbol{x}) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij}|x_i - x_j|. \tag{3}$$

The constraint is represented as

$$\sum_{i=1}^{n} s_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij}|x_i - x_j| \leqslant R. \tag{4}$$

As a result, given R, problem **P** is formulated as the following minimization problem.

$$P \begin{cases} \min H(\boldsymbol{x}) = \sum_{i=1}^{n} h_i(1 - x_i), \\ \sum_{i=1}^{n} s_i x_i + \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} c_{ij}|x_i - x_j| \leqslant R. \end{cases} \tag{5}$$

Besides our previous works, typical methods for problem P include [46–48]. These works treated problem P as a variation of knapsack problem. Quan et al. in [49] pointed out a defect of the variation of standard 0-1 knapsack and perfected the theory. In the latest work, Yan et al. in [30] proposed a hybrid meta-heuristic based on particle swarm optimization (PSO). GPU was utilized to compute each particle's communication cost. However, only the speedup of communication cost between sequential implementation and GPU implementation was reported, there was no analysis of the effect of GPU on the performance of whole algorithm.

3 An efficient GPU-based parallel tabu search (GPTS) for HW/SW partitioning

3.1 Framework of GPTS

When comparing with tabu search methods on CPU for HW/SW partitioning, which make a trade-off between solution quality and time [6,47], our method is based on an adaptive strategy. Given a current solution, the neighborhood is generated by sequentially performing 2-flip operation on this current solution. Given that n is the number of nodes, the size of neighborhood is obtained by $n \times (n-1)/2$. In this neighborhood, the dimension of each candidate is n. Fig. 1 shows the neighborhood generated by adaptive strategy.

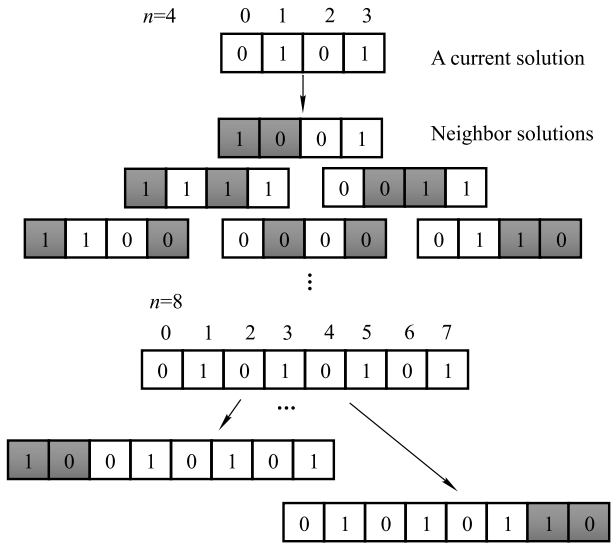


Fig. 1 Neighborhood generation based on sequential 2-flip. When n equals 4, the size of neighborhood is 6. When n equals 8, the size of neighborhood is 28

Though this strategy helps to find a better solution at the cost of longer time, running time can be reduced by GPU. Figure 2 is the framework of proposed efficient GPU-based parallel tabu search algorithm, namely GPTS, for HW/SW co-design. In the figure, the left side of dash line indicates the

host side, and the right side indicates the GPU side.

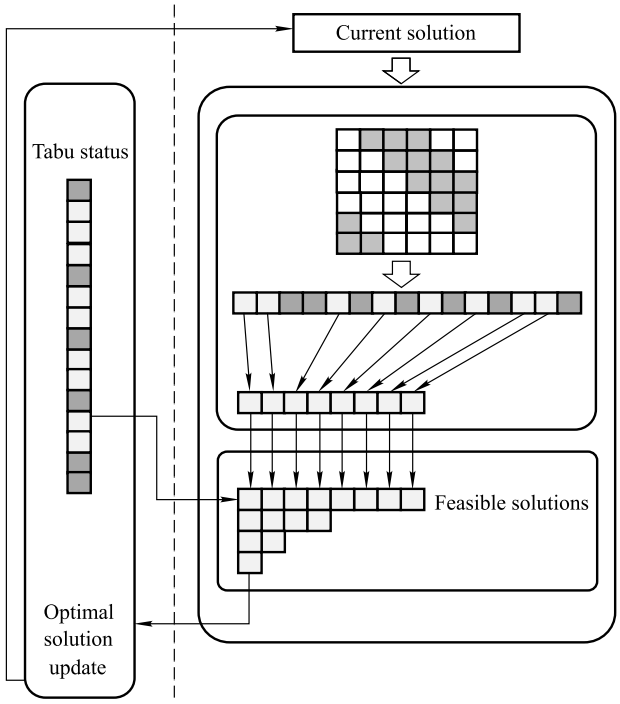


Fig. 2 Framework of GPTS

Firstly, each candidate is mapped to one GPU thread logically to compute the hardware cost, software cost and communication cost in a way of incremental evaluation and to check whether it is subject to the constraint R or not [45].

The next, the feasible candidates are retained by GPU-based compacting neighborhood. The proposed new implementation in this paper will only need single GPU kernel. What is more, the process in which each candidate computes the costs and feasibility will be integrated into this single kernel. This fusion will bring about the significant performance improvement of the whole method. This is the largest difference between previous method in [44] and this method. The reasons will be explained in the following sub-sections.

Thirdly, the optimal candidate is selected by tabu evaluation and is transferred back to the host side. In previous works [44,45], the tabu status table was stored in GPU global memory but updated at host side. To minimize the transfer overhead between CPU and GPU, although a bit-level representation of tabu status was proposed in [45], it was still stored in global memory. In this work, the tabu status table will be stored at the host side. Based on it, a transfer optimization strategy will be proposed to further address the issue of transfer overhead.

In sum, the main contributions in this paper improve the efficiency of GPU-based tabu search for HW/SW co-design. The following subsections will descript these contributions in details.

3.2 Compacting neighborhood of single kernel on GPU

HW/SW partitioning is treated as constraint optimization problem. After each candidate computes three costs, the next step is to check their feasibility according to constraint R. In our method, we retain the feasible candidates and count their number. This procedure is called compacting neighborhood. In our problem, different problem size and constraint R make the number of feasible candidates change adaptively.

Sequential realization of compacting neighborhood is straightforward and has linear time complexity. The algorithm is as follows.

Algorithm 1 Sequential compacting neighborhood

```
pos ← 0
For i← 0 to N-1
  If candidate[i] satisfies formula (4) then
    output[pos] ← candidate [i]
    pos++
  End if
End for
```

Comparing with sequential implementation, GPU implementation is a programming challenge. In previous works [44,45], we presented a GPU version of compacting neighborhood based on existing GPU compaction, which consists of *Prefix-Sum* (*Scan*) and *Scatter* [50,51]. Figure 3 illustrates the main step for HW/SW partitioning. Specially, with the feasibility array, performing *Prefix-Sum* (*Scan*) returns the final position of each feasible candidate in the output array. The next, performing *Scatter* puts them into the output array without changing their relative positions. *Scatter* can be integrated into the process of *Prefix-Sum* to save extra launch overhead of GPU kernel [50,51].

However, this implementation is not efficient enough because it is based on *reduce-scan-scan* pattern of *Prefix-Sum*, in which one procedure corresponds with one GPU kernel call [50,51]. Assuming that the neighborhood size is n, *reduce-scan-scan* pattern involves $3n$ global memory accesses. The first *reduce* kernel reads n data from global memory into register to perform reduction and returns reduce results. The second *scan* kernel performs scan on the reduce result array. The amount of global memory accesses in the second kernel is very small, so its overhead is neglected. In the third *scan* kernel, besides final n global memory write, the whole input data must be read from global memory again. This extra n global memory read is unavoidable because in

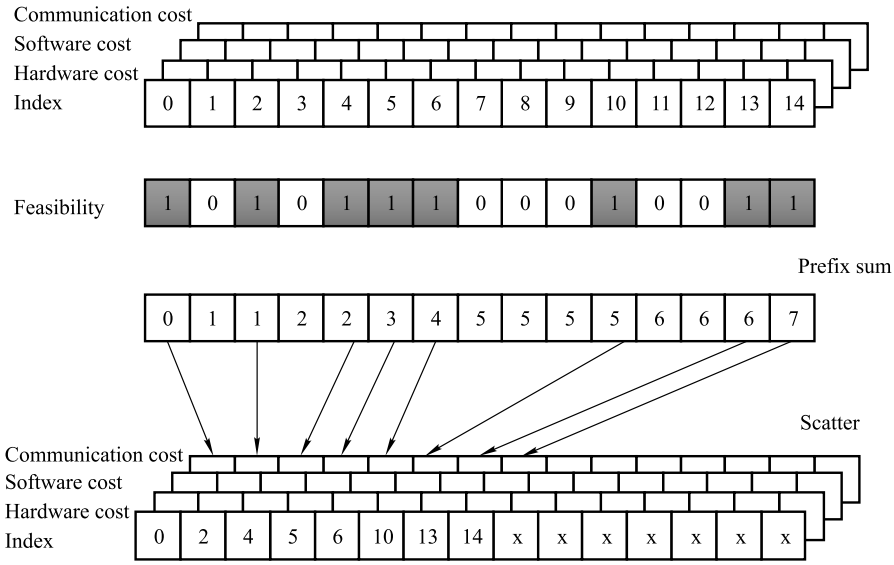


Fig. 3 High-level representation of compacting neighborhood

GPU, once the first *reduce* kernel finishes the work, the data placed in registers will vanish too. This means the input data placed in the register of first kernel cannot be reused.

To reuse the data already placed in register in the first *reduce* kernel, a possible way is performing neighborhood compaction in one kernel. However, in the general pattern of GPU computing, the synchronization of different thread bocks is finished at the host side. This limitation of GPU architecture makes directly integrate three separate kernels into one kernel unrealizable. Therefore, some parallel-hostile tasks, like *Prefix Sum*, have to finish the process by multi-kernel calls.

In this paper, we present a new compacting neighborhood of single kernel on GPU. We adopt *persistent threads style GPU programming*, or *PT* as a framework of compacting neighborhood [52]. *PT* launches as many blocks of maximum size as GPU can simultaneously process, such as 1,024 threads in a block. In GPUs of current generation, the number of simultaneously active blocks is two times that of SMs. For example, the number of active blocks in GTX780 GPU is 24. In this way, the resource utilization of single block on GPU can be maximized.

Next, the neighborhood is divided into chunks of size equaling to that of thread block. Hence, a thread block processes multiple chunks in a stride of 24. In each chunk, intra-block threads cooperatively perform local prefix sum. In Kepler GPUs, shuffle instructions allow intra-warp threads to directly share data placed in thread-private register files.

Now, the challenge of finishing the global compacting neighborhood in a single kernel is how to establish the cooperative relationship among blocks.

To attack this problem, this paper uses two small auxiliary arrays as cyclic buffers being located in GPU global memory. The size is set as a power of two, for example 64, which is much smaller than that of neighborhood. The core five steps are listed as follows.

- At the end of intra-block scan, the last result of each chunk is written in the first auxiliary array.
- Since each thread block do not finish the process of intra-block scan at the same time, to ensure the access consistency of inter-blocks, when one thread block finishes the process, it will call GPU memory fence operation, namely __threadfence(), which ensures that the updated data is visible to the threads in other blocks.
- A self-defined variable of ready flag is set in the second auxiliary array.
- Under PT framework, the number of active thread block is 24. Hence, the intra-warp threads of each block poll the ready flag of prior chunks related to this chunk.
- Once all the local prefix-sums in the first auxiliary array are available, the intra-warp threads of each block will access them and perform reduction.
- Each intra-block thread adds the result to their own local prefix-sum and stores the prefix-sum of last intra-block thread back in shared memory, which is very fast. When processing the next chunk, the thread block can reuse the chunk result of last round and obtains the global prefix sum.
- Finally, the global positions of all feasible candidates are obtained and the feasible candidates are stored in the global memory.

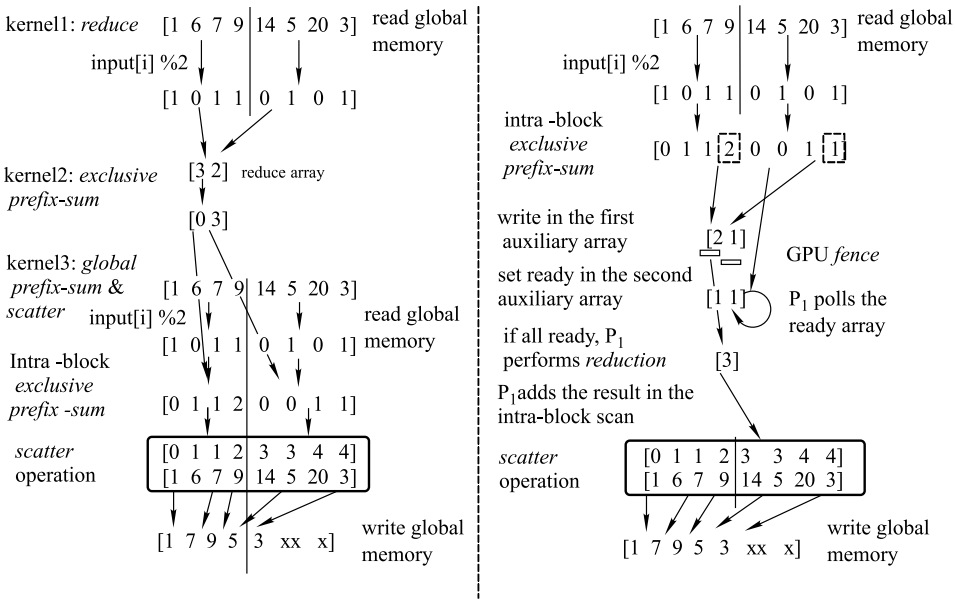


Fig. 4 Different GPU patterns of compacting neighborhood. For simplicity, assuming that a GPU has two thread blocks, each block has one thread (P_0 and P_1). The left process is previous multi-kernel implementation. The right process is proposed single-kernel version

In this way, the whole process is done in a single kernel, and only involves $2n$ global memory accesses except the small auxiliary array accesses. Figure 4 shows the difference between previous implementation in [50,51] and proposed single-kernel implementation. It is seen that the utilization of GPU fence and self-defined poll mechanical cooperatively realize the synchronization of different thread blocks. This implementation of single kernel avoids extra n global memory accesses.

To verify the effectiveness of single-kernel implementation, we design an experiment of picking odd numbers out of an array consisting of random 4-byte integers. The speed-ups are shown in Fig. 5. It should be noticed that when comparing with previous multi-kernel implementation, the speedup is no more than 1.5. It makes sense because this is consistent with the 33.3% improvement ratio of global memory accesses, achieved by $(3n - 2n)/n * 100$.

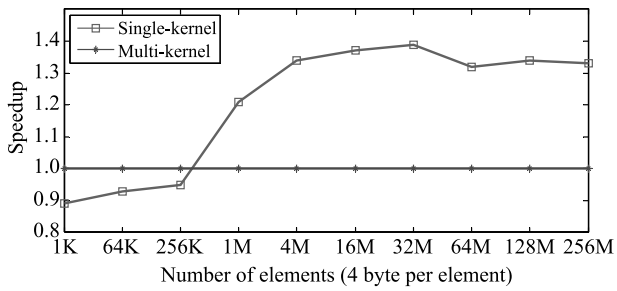


Fig. 5 Speed up of compacting neighborhood of single-kernel implementation

3.3 Kernel fusion

For our problem, each candidate consists of candidate index, software cost, hardware cost, and communication cost and feasibility flag. Assuming that the size of neighborhood is n, after each candidate computes their own costs, their information will be stored in the GPU global memory and the amount of global memory access is $5n$. At the stage of compacting neighborhood, the candidate information will be put into the register again. Therefore, the amount of global memory accesses is up to $10n$, in which $5n$ is for write and $5n$ is for read. This is the traditional pattern of computing candidates' costs plus compacting neighborhood, as is shown in the left subfigure of Fig. 6.

Based on the compacting neighborhood of single kernel, we further propose a fusion strategy in which computing costs and compacting neighborhood are integrated into a single kernel. The advantage of the fusion strategy is that once each candidate obtains the costs, compacting neighborhood is immediately followed. This means $10n$ GPU global memory accesses are avoided. As a result, only the final feasible candidates are written into the global memory. The right subfigure in Fig. 6 clearly highlights this advantage. Figure 7 further shows the code segment of kernel fusion under the framework of new compacting neighborhood.

To testify the fusion strategy, we also design an experiment, in which two procedures are ran on GPU to simulate candidate computing and compacting neighborhood. In the first procedure, each thread does loop accumulation in

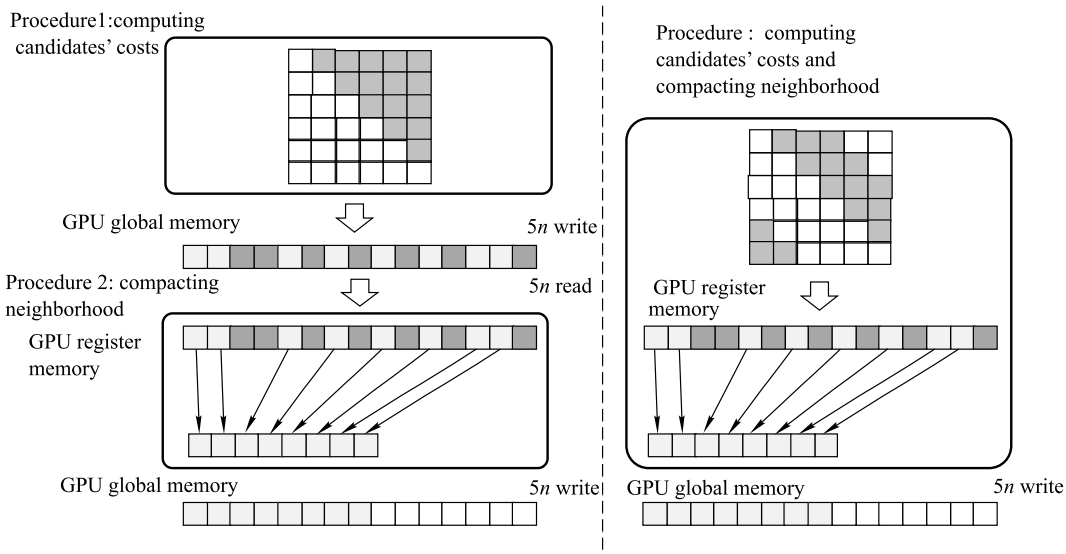


Fig. 6 The difference of computing candidate costs and compacting neighborhood between previous method and proposed method

```
__global__ void computing_and_compacting_kernel(…)
{
  chunks =(items + (1024- 1)) / 1024;
  …
  for (chunk = blockIdx.x;  chunk < chunks;  chunk += SMs * 2){
     neib_id = tid  + chunk * 1024 ;
     if(neib_id<neib_size){
        compute two flipping positions;
        compute hardware cost, software cost and communication cost;
        check the feasibility according to constraint R;
        …
        compacting neighborhood;
        …
       obtain out_pos;
       if(feasible){
          out_neib[out_pos]=neib_id;
          out_hwcost[out_pos]=hareware_cost;
          out_swcost[out_pos]=software_cost;
          out_comcost[out_pos]=communication_cost;
       }
       …
     }
  }
}
```

Fig. 7 Kernel fusion strategy

parallel. Loop accumulation simulates the process of computing communication cost in each candidate. The reason is that although an incremental evaluation strategy was proposed in [44,45], for each candidate, computing communication cost is still the most time-consuming, which involves the traversal of adjacent nodes in the task graph. Its time complexity depends on the number of adjacent nodes of flipped nodes in the current solution. By contrary, computing hardware cost and software cost is efficient, which are always constant.

After loop accumulation, the result of each thread is written into its corresponding positions of 5 arrays, which consist of 4-byte integer allocated on GPU global memory. The second procedure simulates the process of compacting neighborhood. Different from the experiment in Section 3.2, the amount of global memory access in this compacting neighborhood is five times as many as that experiment.

In GPU computing, global memory bandwidth is another way of measuring the performance of GPU kernel [50]. In the first procedure, we change the number of loop accumulation to increase the workload of each thread. This reflects that when the structure of task graph becomes increasingly complex, the time cost of computing communication cost will grow. Figure 8 shows the bandwidth of different data size when comparing against the computing pattern in conference version [44]. In the one hand, the effect of fusion strategy is always better than previous computing pattern. In the other hand, the effect of fusion depends on the workload in loop accumulation. When the loop number is small, such as 1 or 10, the effect of fusion strategy is significant. In the contrary, if the loop number is large, namely 100, the effect becomes less significant. That is to say, the effect of fusion strategy is fixed, namely $10n$ global memory accesses are always saved, regardless of how many loop accumulations are. Therefore, increasing the number of loop accumulation makes computing take up the most of time and the effect of fusion strategy decreases.

3.4 Restart strategy

Since the feasibility of each candidate cannot be known beforehand, it would happen that after compacting neighbor-

hood, all of the candidates do not satisfy the constraint. In order to ensure that there always exist feasible candidates, when all the candidates are infeasible, the current solution is restarted in (0,0,...,0), namely, all tasks are implemented in hardware. By randomly flipping two positions, a new current solution is generated and its new neighborhood is generated by performing sequential 2-flpping. This procedure does not stop until there do exist at least one feasible candidate solution. To some extent, this way ensures the diversity of search space.

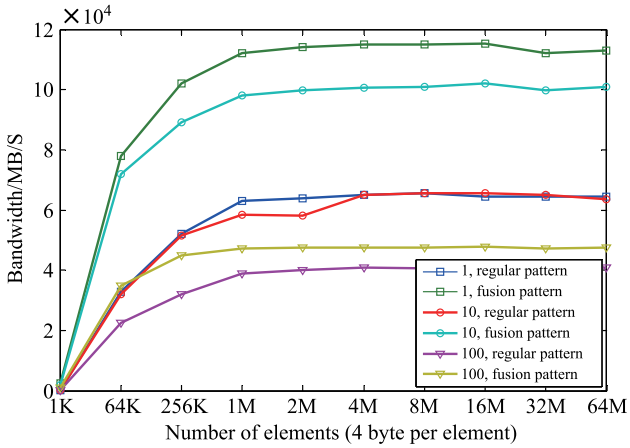


Fig. 8 Memory bandwidth of separate implementation and fusion implementation

3.5 GPU implementation of tabu evaluation with zero copy strategy

After retaining the feasible candidates, tabu evaluation is performed to select an optimal one with smallest hardware cost which is not tabu. Meanwhile, aspiration criterion is triggered if certain tabu candidate has smaller hardware cost than known global cost. In order to efficiently implement this process on GPU, previous work presented a parallel tabu evaluation based on enhanced GPU reduction with tabu status array [44]. In [45], a bit-level representation of tabu status was further proposed. To understand the proposed optimized transfer strategy in this paper, this paper firstly describes tabu evaluation on GPU in detail.

Specially, on GPU, each checking thread runs in parallel on a number of hardware cost array chunks and allocates a hardware cost buffer in register memory by initializing it as the MAX value. Since the hardware cost buffer is frequently accessed at the beginning stage of tabu evaluation, data placement of hardware cost buffer in register memory will greatly improve the performance of the whole process.

For each 2-flip index, it will be checked whether it is tabu or not by accessing the exact position in tabu-status array. If a 2-flip index is tabu, our method will further check whether its corresponding hardware cost is smaller than known global hardware cost or not. If yes, its tabu status is ignored by aspiration criterion and the hardware cost buffer is updated. If not, the initialized MAX value of hardware cost buffer is not updated. By contrary, if a 2-flip index is not tabu, it directly updates the initialized MAX value with the smaller hardware cost. At end of this process, we get a number of buffers, which store the smallest intra-chunk hardware cost.

For a number of buffers, we perform a standard GPU-reduction to find the smallest hardware cost in all of the hardware cost array chunks. Figure 9 illustrates the whole process of tabu evaluation on GPU.

The size of tabu status array is equal to that of neighborhood, namely 15. The tabu list manages six tabu objects. Correspondingly, tabu statuses of these six objects are set as 1. After compacting neighborhood, the number of feasible 2-flip indexes is 8. The 2-flip index is used to position their tabu status. In current iteration, not all tabu candidates in the tabu list are feasible, such as candidates of index 1 and 7. Assuming that the known global hardware cost is 2, it is noteworthy that although the feasible candidate of index 14 is tabu, the hardware cost buffer is still updated by aspiration criteria.

From the procedure of tabu evaluation on GPU, the tabu status array is frequently accessed by each feasible 2-flip candidate. At each iteration, the tabu status is updated at the host side, a straightforward way to perform GPU-based tabu evaluation is to transfer the whole tabu status array from CPU memory to GPU memory.

However, after compacting neighborhood, the number of remaining feasible solutions are generally much less than the size of tabu status table. For example, in an extreme case, there is only one feasible candidate. Therefore, the overhead of transferring the whole tabu status is obviously prohibitive.

After carefully analyzing the above problem, this paper proposes a tabu evaluation method with zero copy strategy, which makes full use of optimized transfer architecture between host side and device side.

With this strategy, the tabu statuses of feasible solutions are fetched directly from main memory through PCI-E instead of GPU global memory. Hence, there is no need to move the overall tabu status table from host side to device side. Just like there is no free lunch, the penalty of this strategy is that the execution time of the kernel will be prolonged. After considering the overall benefit, we think this strategy is well worth being adopted.

To testify the proposed strategy, we perform GPU

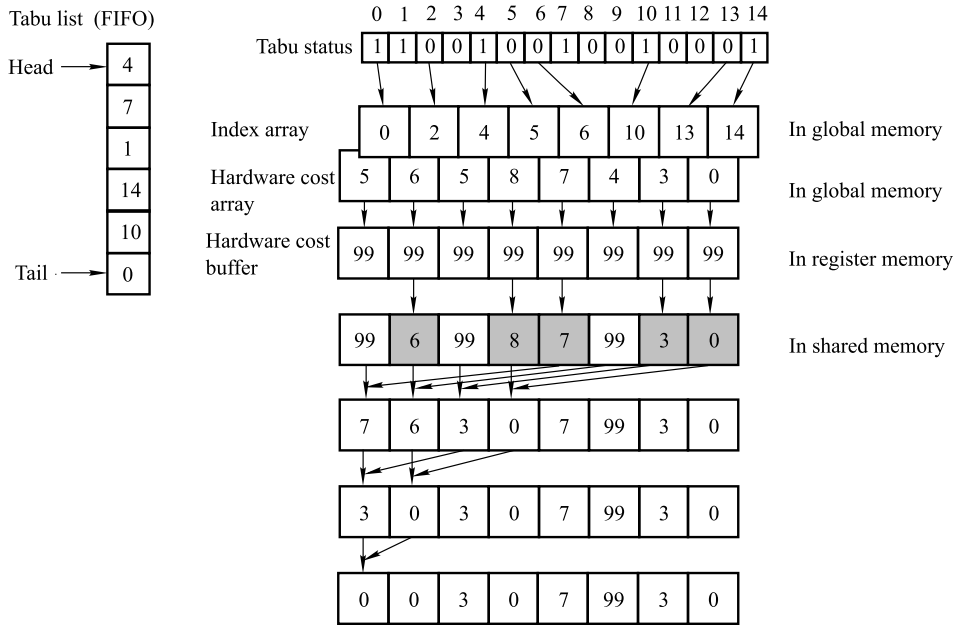


Fig. 9 An example of tabu evaluation on GPU

reduction on an array consisting of 1 or 0 to simulate tabu evaluation on GPU. In previous method, one element takes up one byte. In proposed method, one element takes up one bit. Meanwhile, the transfer overhead is considered. Figure 10 shows that the byte-level reduction with transfer overhead is taken as a baseline method and bit-level reduction with zero copy strategy can achieve the best performance.

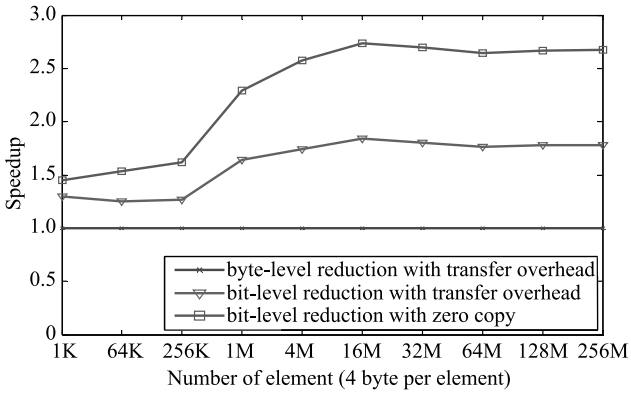


Fig. 10 Speed up of bit-level GPU reduction with zero copy

3.6 Update strategy of tabu list

During tabu evaluation, if all feasible candidates are tabu and no aspiration criteria is triggered, then randomly choosing a tabu feasible candidate.

When updating the tabu list, we firstly check whether the index of optimal candidate is in it or not. If yes, we adjust the relative positions of indexes in the list. Specially, if the index of optimal candidate is not at the tail of list, then all the tabu indexes behind it are moved one step toward the head of list and the index of optimal candidate is put in the tail of list. If no, we further check if the tabu list is full. If yes, we dequeue the index in the head and set its tabu status as 0. Next, we enqueue the index of optimal candidate and set its tabu status as 1.

3.7 Stop criterion

We stop our algorithm when the number of iterations reaches its maximum. If the global hardware cost is not improved in the given number of iterations, we stop the algorithm as well. Once the global hardware cost is improved, we reset the no-improvement count as 0.

Combining the above issues together, Algorithm 2 shows the pseudo-code of proposed efficient GPU-based parallel tabu search (GPTS) for HW/SW partitioning.

4 Experiments

4.1 Benchmark and platform

Because the proposed method is based on the heuristic, we have to empirically determine its performance and effectiveness. In order for fair comparison, we chose very related benchmarks for HW/SW partitioning in references [46–48].

Table 1 shows the number of nodes n and that of edges m in each task graph. In a task graph, each node has soft-

ware cost and hardware cost. Each edge has communication cost and two adjacent node indexes. To evaluate the size of a task graph, formula $size = 2 \times n + 3 \times m$ is utilized. The first five task graphs are abstracted in real-world applications [53]. The other nine task graphs are generated randomly, a pervasive way of comparing the performances between different algorithms.

Algorithm 2 GPTS for HW/SW partitioning

Initialize the current solution and global optimal
Initialize the tabu list at host side
Initialize the tabu status at host side
Allocate the node cost of task graph space on GPU memory
Allocate the neighborhood space on GPU memory
Bind the task graph on GPU texture memory
Transfer the node cost of task graph to GPU memory
While stop criterion does not satisfy ***then***
  Copy the current solution to GPU memory
re-compute:
  Computing the cost in the neighborhood and removing infeasible solutions in a single kernel
  If the number of feasible solutions is 0 ***then***
    Reinitialize the current solution
    Goto recompute
  End if
    Launch tabu Evaluations kernel
  Transfer back the optimal candidate to the host side
  If all feasible solutions are tabu ***then***
    randomly choose a feasible solution
    update the optimal candidate
  End if
  Update the tabu list at the host side;
  Update the tabu status table at the host side;
  Replace the current solution with the optimal candidate solution at the host side;
  If current hardware cost < global hardware cost ***then***
    Update the global solution
    Clear the $count_{\text{no improvement}}$
  Else
    $count_{\text{no improvement}}+1$
  End else
  End if
End while

The related parameters in our experiments are set in the same way as in [11,46,47]. For more details, please see [11,46,47].

- The software cost is generated as uniform random numbers from the interval [1,100]. The hardware cost is generated as random numbers from a normal distribution with expected value $k \cdot s_i$ and a given standard deviation $k \cdot \lambda \cdot s_i$, where s_i is the software cost of the given node. The value of k denotes the choice of units for software and hardware cost. The value of λ denotes the correlation between a node's hardware cost and software cost. In [11,46,47], the authors pointed out that the value of k had no algorithmic implications and that of λ did not impact on the performance of the algorithm.
- The communication costs were generated as uniform random numbers from the interval $[0, 2{\cdot}\rho \cdot s_{\max}]$, where $s_{\max}$ is the highest software cost. Thus, the communication cost has an expected value of $\rho \cdot s_{\max}$, and ρ is the so-called communication to computation ratio (CCR). ρ was taken as 0.1, 1 and 10, corresponding to computation-intensive case, inter- mediate case, and communication-intensive case, respectively.
- R was randomly generated as a uniform random number (1) from the interval $[0, 1/2{\times}\sum s_{\text{i}}]$, corresponding to the strict real-time constraints, (2) from the interval $[1/2{\times}\sum s_{\text{i}}, \sum s_{\text{i}}]$, corresponding to the loose real-time constraint. The two cases are indicated as R=low and R=high, respectively.

Table 1 Benchmark

Name	n	m	Size	Description
crc32	25	34	152	32-bit cyclic redundancy check. From the telecommunicat-ions category of MiBench [53].
patricia	21	50	192	Routine to insert values into patricia tries, which are used to store routing tables. From the Network category of MiBench [53].
dijkstra	26	71	265	Computes shortest paths in a graph. From the Network category of MiBench [53].
clustering	150	333	1299	Image segmentation algorithm in a medical application.
rc6	329	448	2002	RC6 cryptographic algorithm.
random1	1000	1000	5000	random graph
random2	1000	2000	8000	random graph
random3	1000	3000	11000	random graph
random4	1500	1500	7500	random graph
random5	1500	3000	12000	random graph
random6	1500	4500	16500	random graph
random7	2000	2000	10000	random graph
random8	2000	4000	16000	random graph
random9	2000	6000	22000	random graph

The platform of running sequential method is Intel i7-4770 with 3.4GHZ clock frequency. The size of main memory is 16GB. The platform of running parallel method is NVIDIA GTX 780, consisting of 12 SMs, 192 SPs per SM. The clock frequency of each SP is 1.059GHZ. The size of global mem-

ory is 3GB. The development tool is Visual Studio2012. We implement the parallel procedures in CUDA C and the SDK version is CUDA 8.0.

4.2 Comparison method

Because the proposed method is based on the heuristic, we have to empirically determine their performance and effectiveness. We tested the proposed method in different *CCR* and constraint *R*.

We compare our method of sequential implementation and GPU implementation against *Alg-new3* in [46], HEUR and TABU in [47], NODERANK in [48], PDPSO-IWO in [30]. The sequential implementation and GPU implementation of our method in journal version is named as STS and GPTS, respectively. The maximum iteration number M was set as 2000 and the non-improvement threshold N was set as 200, the same as TABU in [47]. The parameters in comparison objects are the same as in literature.

We compare the performance of proposed method with that of existing methods. To make our work convincing, we analyze the results from different respects.

Firstly, we compare the proposed tabu search method with existing tabu search method for HW/SW partitioning [47], which is the most related and comparable. In this comparison, the proposed method outperforms the existing method on both speed and quality.

Secondly, problem-specific heuristic, NODERANK, is also compared. Since NODERANK cannot directly compared with general metaheuristics, in order to make the comparison more reasonable, one choice is to compare the performance within the same time range. A balance between solution quality and time is achieved when the iteration number of NODERANK is set as 4 in [48]. Based on the running time of NODERANK with 4 iterations, the proposed method can get the similar runtime range when the iteration number is 50. In this comparison of same runtime range, our method still has the advantage of solution quality.

4.3 Result analysis

The hardware cost and running time of different methods are showed in Table 2 from (a) to (f). It is seen that there exists no algorithm of absolute advantage. When comparing with three existing heuristics, namely *alg-new3*, HEUR and NODERANK, STS and GPTS can achieve better solution quality in most cases. However, they do not have advantage of efficiency, which is determined by the large iteration number and the characteristic of meta-heuristic. When comparing GPTS with STS, GPTS is much more efficient than STS in relatively large problems such as from *clustering* to *random9*, by fully making use of computing power of GPU. What is more, GPTS outperforms TABU in [47] on both quality and speed. When comparing with PDPSO-IWO in [30], GPTS outperforms PDPSO-IWO in most cases. The most important is, GPTS is significantly faster than PDPSO-IWO.

Table 3 shows the solution improvement by GPTS over *alg-new3* after obtaining the initial solution by *HEUR* in 6 cases. Here, the improvement is defined as

$$\text{Improvements} = \frac{H(\boldsymbol{x}_A) - H(\boldsymbol{x}_B)}{H(\boldsymbol{x}_A)} \times 100. \tag{6}$$

Table 2 Solution quality and time (millisecond) averaged over 30 random instances at different CCR and R, the values of better solution quality are bolded
(a) CCR=0.1, R=low

Size	Alg-new3		HEUR		TABU		NODERANK		PDPSO-IWO		STS		GPTS	
	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time
152	977	0	955	0	947	512	933	1	899	842	911	457	911	800
192	934	0	904	0	892	769	890	1	803	961	848	567	848	1019
265	1129	0	1081	0	1065	791	1059	1	1036	1002	1007	1424	1008	1188
1299	6104	1	5629	2	5507	2549	5421	3	5524	1132	5295	366	5292	290
2002	12858	1	12262	1	12029	4097	11836	4	11083	1362	11690	910	11687	340
5000	38811	11	37335	20	36929	14784	36045	62	37181	2165	35865	8628	35865	815
7500	60198	23	58181	46	57524	18475	56336	88	53516	3666	56163	23684	56164	1377
8000	41586	14	38840	31	38097	15987	37449	87	40047	3088	36950	12701	36952	1016
10000	81445	29	78990	64	78004	28413	76893	124	71020	4821	76604	45200	76605	2397
11000	43482	16	40466	42	39233	16914	39314	153	40134	3766	38822	14315	38806	1471
12000	61186	25	57387	51	56179	19012	55516	167	53048	5511	54658	37121	54669	2455
16000	81746	31	76846	74	74992	32597	74222	193	76187	8011	73234	80824	73223	5664
16500	65330	27	60867	58	59074	26941	59112	249	61280	9756	58361	43368	58341	3464
22000	86499	34	79820	93	78025	38648	77328	291	72054	12125	76265	92062	76281	7329

(b) CCR=0.1, R=high

Size	Alg-new3		HEUR		TABU		NODERANK		PDPSO-IWO		STS		GPTS	
	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time
152	448	0	445	0	383	532	442	1	407	923	**357**	263	**357**	379
192	431	0	443	0	397	697	427	1	405	1125	**305**	253	**305**	255
265	643	1	535	0	453	984	523	2	602	969	**311**	287	**311**	362
1299	3153	2	2971	2	2184	3657	2860	7	2828	1325	2131	437	**2129**	356
2002	5786	2	5570	3	**4213**	5113	5306	11	4947	1397	4292	1264	4291	532
5000	16591	10	16070	17	**12690**	15574	15295	62	15131	2104	12844	11196	12842	948
7500	26894	25	25959	41	**19959**	35471	24753	149	23048	3471	21331	31125	21331	1887
8000	22830	15	20928	28	17101	30347	20075	81	18629	3670	**16512**	16451	16530	1346
10000	30642	31	30073	79	25179	61347	28720	307	23504	5340	23313	63286	**23308**	4043
11000	26469	18	22944	35	**17174**	43564	22216	75	21943	4267	18002	24078	18026	1772
12000	35685	26	32564	65	30780	69842	31182	154	30511	5768	26330	51755	**26304**	3836
16000	48219	30	43742	86	41906	70213	41866	296	44651	9210	**35808**	109758	**35808**	7120
16500	40117	29	34661	81	32184	68412	33514	156	37269	11241	**27895**	76529	27951	4394
22000	53227	36	45798	101	42322	98958	44267	314	48437	12338	35735	151660	**35719**	10503

(c) CCR=1.0, R=low

Size	Alg-new3		HEUR		TABU		NODERANK		PDPSO-IWO		STS		GPTS	
	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time
152	1193	0	1088	0	1063	645	1074	1	1132	824	**1041**	1536	**1041**	2032
192	1031	0	985	0	982	756	983	1	938	893	**913**	3671	918	3839
265	1197	0	1150	1	1146	1039	1140	2	1195	1120	**1041**	1353	1049	1949
1299	7161	1	6584	2	6486	2241	6368	8	6194	1321	6180	358	**6163**	320
2002	15643	2	13983	2	13439	2547	13074	9	**11263**	1236	12970	861	12972	363
5000	46509	12	41606	20	40268	8863	38321	81	**33254**	2165	38535	7187	38536	680
7500	69514	21	61719	54	59891	24129	**56649**	196	62146	3756	56856	21664	56864	1554
8000	47390	16	43884	33	42825	12337	41921	102	43551	3801	**41547**	12078	41550	1068
10000	93026	29	82412	69	79891	24769	75339	283	**71258**	4757	75791	41043	75788	2647
11000	47132	18	44086	44	43629	19567	43063	89	42749	3679	42759	11883	**42708**	1087
12000	70600	24	64854	56	63759	28123	61844	187	**60787**	5153	61160	38744	61152	2822
16000	94186	32	86719	86	85719	28317	82850	310	83072	8102	**81821**	87404	81870	6641
16500	70828	28	67464	68	67464	28789	65720	197	70403	9678	65373	40938	**65365**	3608
22000	94374	34	89324	98	89323	35453	87182	309	**83238**	11245	86712	76958	86709	6157

(d) CCR=1.0, R=high

Size	Alg-new3		HEUR		TABU		NODERANK		PDPSO-IWO		STS		GPTS	
	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time
152	1051	0	846	0	725	764	833	1	886	1105	560	331	**559**	431
192	964	0	802	0	761	989	793	1	790	1100	515	124	**513**	325
265	1201	0	980	2	906	1229	973	7	1077	989	**651**	202	656	338
1299	7063	1	5360	3	**4815**	4892	5272	13	5982	1078	4889	464	4901	266
2002	14840	2	10092	3	8349	7661	9594	15	11353	1297	**7964**	948	8021	368
5000	43068	11	27901	19	24598	19047	26044	91	29803	2446	**20173**	9719	20258	840
7500	64746	25	42831	46	38062	47971	39604	226	43315	3598	31795	26032	**31789**	1522
8000	46807	14	36232	28	33971	42014	34851	87	37867	3433	32979	13109	**32952**	1146
10000	86690	28	58295	78	52776	99873	53704	382	62243	4976	**44341**	52991	44374	3301
11000	46908	17	37899	41	**35508**	48204	37330	114	44797	4793	36591	12664	36594	1187
12000	69986	26	52491	57	49510	72923	50934	219	58298	5894	47787	37248	**47687**	3088
16000	92995	31	70722	94	67612	171980	68386	376	75326	9302	**64867**	77037	64883	5367
16500	70733	27	58804	69	**55347**	89314	57901	211	69318	11287	56821	41683	56856	3147
22000	94400	35	75925	113	**71355**	235270	75022	369	83072	12386	73217	90591	73250	7129

(e) CCR=10.0, R=low

Size	Alg-new3		HEUR		TABU		NODERANK		PDPSO-IWO		STS		GPTS	
	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time
152	1141	0	1088	0	1087	253	1086	1	1075	801	**1048**	1676	1053	2759
192	982	0	981	0	980	297	981	1	950	913	**934**	7279	**934**	7879
265	1224	0	1221	1	1221	451	1221	2	1198	997	1183	7301	**1179**	8803
1299	6975	2	6766	2	6750	952	6743	5	6682	1023	6682	1230	**6679**	1186
2002	15493	1	14303	3	14251	1257	13882	14	**13386**	1274	13837	668	13843	360
5000	47025	11	43350	16	43055	6149	**39821**	47	44439	2068	40767	5813	40771	612
7500	70604	24	66880	41	66709	8746	**61698**	159	58531	3517	63708	13461	63723	961
8000	47083	15	45461	19	45411	7198	**44688**	51	44870	3079	44695	6654	44711	720
10000	94171	28	86264	79	86168	18943	**78348**	290	74960	4801	80505	30971	80550	1869
11000	47207	18	46481	38	46442	8814	46236	105	**45508**	3754	46177	7320	46175	759
12000	70804	25	68522	54	68463	9835	67157	163	**65281**	5152	67197	20812	67185	1676
16000	94527	30	91643	97	91570	17358	**89381**	271	89517	8106	89474	41549	89473	3341
16500	70801	28	69539	71	69483	11169	68988	191	**68181**	9756	68946	19972	68933	1704
22000	94400	36	92870	121	92796	15451	92200	344	**90530**	10235	92077	45929	92076	3840

(f) CCR=10.0, R=high

Size	Alg-new3		HEUR		TABU		NODERANK		PDPSO-IWO		STS		GPTS	
	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time	quality	time
152	1138	0	1013	0	1008	452	1013	1	1025	1013	**887**	426	897	668
192	954	0	932	0	932	678	932	1	947	1024	787	2287	**785**	2591
265	1216	0	1202	2	1202	856	1202	7	1209	998	996	3838	**953**	3245
1299	7130	2	6624	2	**6491**	1844	6606	9	6688	1165	6617	338	6509	312
2002	15394	2	13070	4	12770	2564	12799	14	13855	1305	12719	690	**12612**	300
5000	47329	12	37738	21	36744	12324	35607	71	37343	2454	**35265**	6559	35267	630
7500	70581	25	55869	48	54734	36148	52858	136	54700	3615	**52346**	15823	52369	1231
8000	47042	16	43314	26	43092	16073	42799	79	43514	3425	**42644**	7227	**42644**	687
10000	94819	30	75353	108	73977	78276	71165	305	74907	5340	**70515**	33461	70586	2168
11000	47683	17	45470	38	45186	12459	45301	108	45394	4397	**45114**	8124	45116	873
12000	70452	25	64809	67	64600	29737	64026	197	67211	5955	63983	17678	**63982**	1383
16000	94215	31	86702	127	86470	65183	85619	341	89787	9103	85446	37248	**85367**	3306
16500	70679	27	67710	79	**67167**	16896	67409	221	67852	11377	67183	21759	67191	1975
22000	94373	34	90545	141	90433	42592	90133	394	90409	12368	89880	46807	**89878**	4010

In the table, the better improvements are bolded. The improvements by STS or GPTS over *alg-new3* are better than that by TABU in 6 cases. Meanwhile, the quality improvements of STS and GPTS are better than NODE- RANK. Although in the first and fifth cases, PDPSO- IWO achieves a little better solution quality. However, our method can achieve significantly better solution quality in the other four cases.

Table 3 Improvement over *alg-new3* averaged over 30 instances

	TABU	NODE-RANK	PDPSO-IWO	STS	GPTS
CCR=0.1, R=low	6.9	7.9	**9.7**	9.6	9.6
CCR=0.1, R=high	21.7	10.6	11.9	**28.7**	**28.7**
CCR=1, R=low	9.3	11.5	12.4	**13.3**	13.2
CCR=1, R=high	30.7	26.6	17.7	**37.5**	37.4
CCR=10, R=low	3.9	6.2	**7.2**	6.5	6.5
CCR=10, R=high	10.5	11.1	8.7	14.4	**14.8**

Figure 11 shows the refinements of our GPTS over HEUR in six cases. It is seen that when R is relaxed, the refinements by GPTS are more than that by TABU in [47]. Especially in the case of CCR=0.1, R=*high*, the refinements are much better.

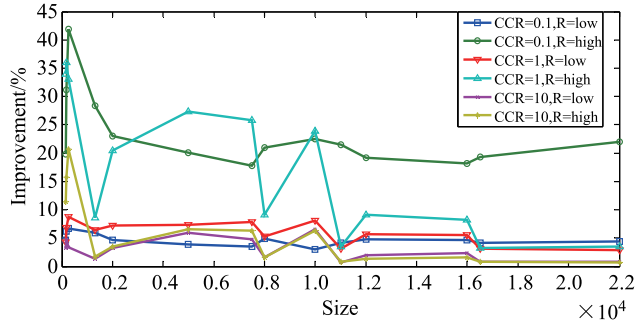


Fig. 11 Refinements of solution quality by GPTS over HEUR in 6 cases

As mentioned before, when generating neighborhood, our adaptive strategy sequentially performs 2-flip operation. Actually, 1-flip, and even 3-flip also can generate neighborhood. The size of neighborhood becomes n and $n\times(n-1)\times(n-2)/6$, respectively. The reason of performing 2-flip is the consideration of both solution quality and execution time. In Fig. 12, we compare the solution refinements over HEUR of three flip operations in *rc6* in six cases. The results of NODERANK measure the actual effects of different flip operations. When comparing with the results of 1-flip, the results of 2-flip and 3-flip operations are significantly better. However, the size of 3-flip neighborhood is 100 times larger than that of 2-flip. Moreover, for each candidate, the workload of computing costs is also increased. These two factors make the running time of GPTS more than 50 seconds. However, its solution refinement is not significant when comparing with 2-flip operation. Therefore, our 2-flip adaptive strategy is reasonable.

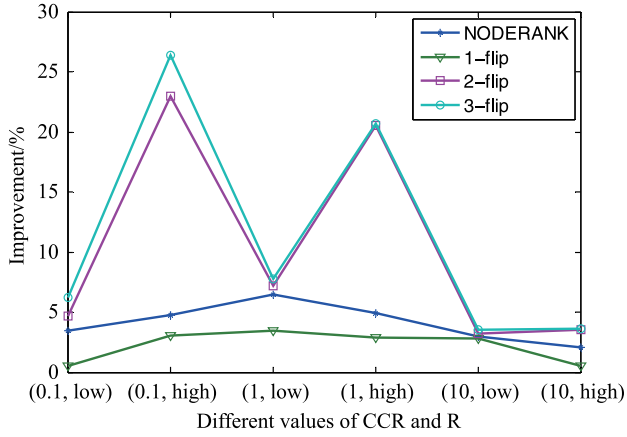


Fig. 12 Solution refinements of three different flip operations over HEUR in 6 different cases in *rc6*

To compare with problem-specific method, NODERANK, which is very different type and is not a comparable method in general, we reduce the iteration number of GPTS. In order to make the comparison more reasonable, one choice is to compare the performance when the running times are very close. In [48], a balance of solution result is achieved when the iteration number of NODERANK is set as 4. After repeated tests, the iteration number of GPTS is set as 50.

In the task graphs of *crc32* to *rc6* in six cases, GPTS achieves better solution quality than NODERANK. However, its solution time is still much slower because of the restart strategy. We mainly focus on the performance of GPTS in *random1*– *random9*.

Figure 13 shows the hardware cost improvement over NODRANK in 6 cases. The solution quality of GPTS is still competitive after reducing the iteration number. Especially when R is high, GPTS achieves better solution quality.

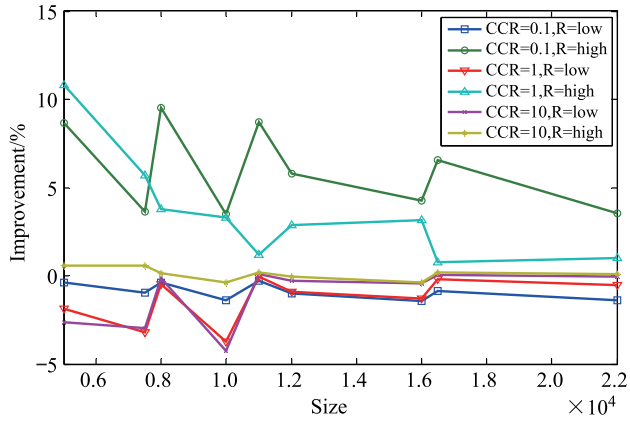


Fig. 13 Refinements of solution quality by GPTS over NODERANK in 6 cases

Although the iteration number is reduced, the solution time of STS is still much slower than that of GPTS and NODERANK. Hence, Fig. 14 shows the execution time of GPTS and NODERANK in six cases. By this way, the performance of GPTS and the advantage of porting STS to GPU are well reflected.

In our method, when current solution can not generate feasible candidates, we adopt a re-start strategy. And when all feasible candidates are tabu and no aspiration criterion is triggered, we randomly set a tabu candidate free. These two random strategies make the solution quality of STS and that of GPTS impaired. Figure 15 shows the error distributions of hardware cost between our sequential implementation and GPU implementation in 6 cases. It is defined as

$$\text{error} = \frac{H(\boldsymbol{x}_{seq}) - H(\boldsymbol{x}_{GPU})}{H(\boldsymbol{x}_{seq})} \times 100\%, \tag{7}$$

$H(\boldsymbol{x}_{seq})$ denotes the hardware cost by sequential implementation while $H(\boldsymbol{x}_{GPU})$ denotes the hardware cost by GPU implementation. Positive error means solution quality by GPU implementation is better while negative error means solution quality by sequential implementation is better.

Finally, to illustrate the difference on efficiency between STS and GPTS, speed-ups, obtained by $\text{Time}_{\text{STS}}/\text{Time}_{\text{GPTS}}$, are summarized, as Fig. 16 shows. To sum up, for the first three task graphs, the speedups are commonly less than 1, meaning that STS is faster than GPTS. When the task graph sizes are 1299 and 2002, the speedups are between 1 and 3. GPTS begins to shows its advantage of efficiency. For the remaining 9 task graphs, the speedups are generally more than 9. It is no doubt that GPTS is much faster than STS.

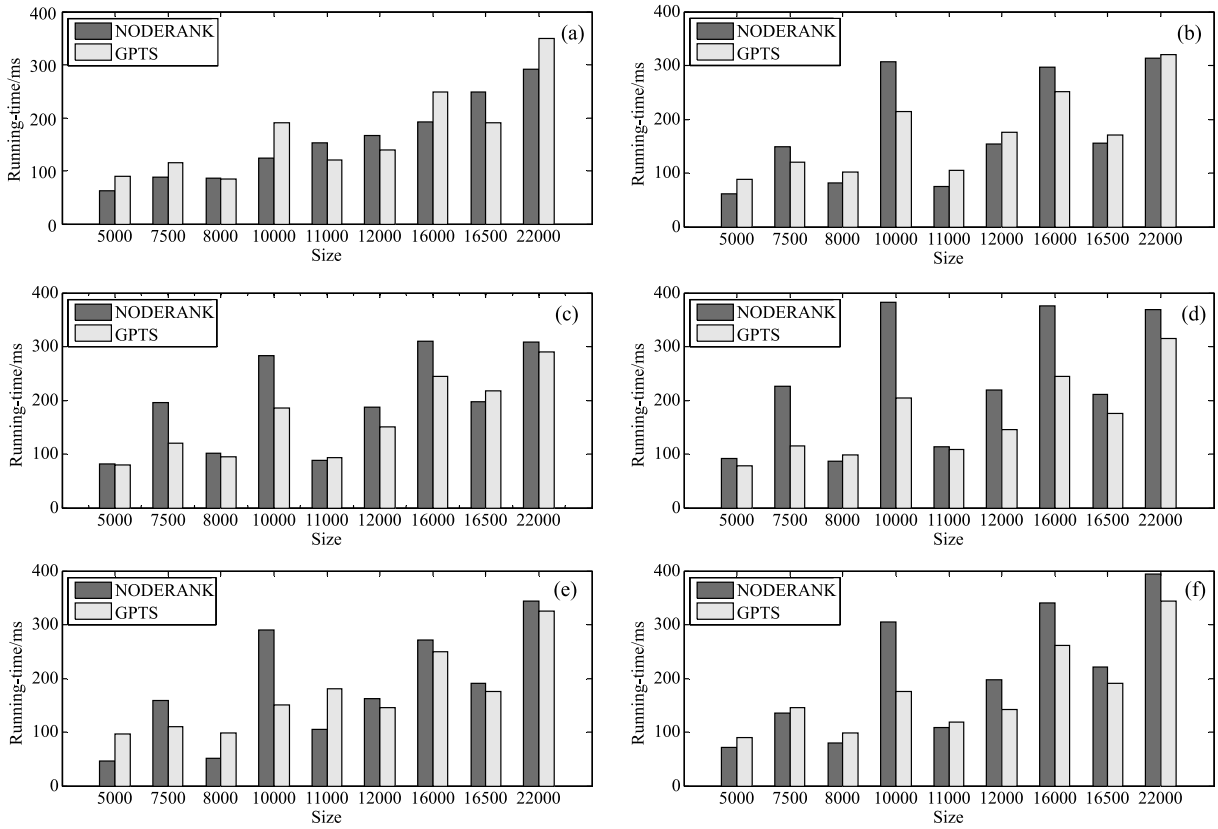


Fig. 14 Running times of NODERANK and GPTS, averaged over 30 instances in different cases. (a) CCR=0.1, R=low; (b) CCR=0.1, R=high; (c) CCR=1, R=low; (d) CCR=1, R=high; (e) CCR=10, R=low; (f) CCR=10, R=high

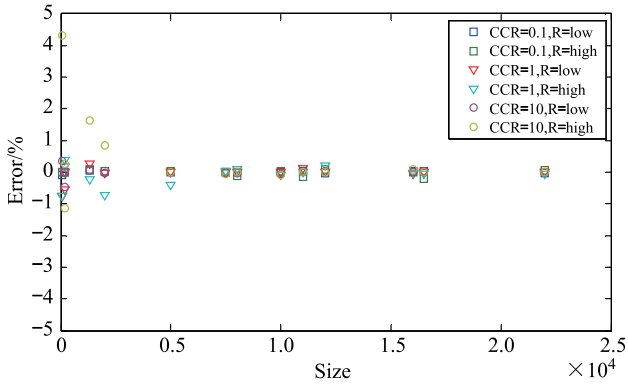


Fig. 15 Errors between sequential implementation and GPU implementation in the six cases

5 Conclusion & future work

This paper presents a GPU-accelerated parallel tabu search for HW/SW partitioning. Different from existing tabu search methods for HWSW partitioning, our GPU-based method can perform the exhaustive search of the entire neighborhood in a reasonable execution time. By combining with the hardware architecture of GPU, we present a single kernel of compacting neighborhood, which reduces the amount of global memory access from $3n$ to $2n$ theoretically. Based on the new compacting neighborhood, we further present a kernel fusion strategy, which further avoids $10n$ global memory accesses. GPU implementation of tabu evaluation with zero copy strategy is also proposed to avoid the transfer overhead of tabu status array from CPU to GPU at each iteration. A number of experiments demonstrate the effectiveness of the proposed method.

In future work, we will continue the research work on following aspects but not limited. Firstly, we will explore other intelligent computing algorithms [54–58] to accelerate HW/SW partitioning on GPU. we will extend the idea of proposed GPU-based methods to other science and engineering domains, such as CAD/Graphics. Image/Video and Computer-Supported Cooperation Work [59–63].

Acknowledgements This paper was supported by the National Natural Science Foundation of China (Grant No. 61472289), National Key Research and Development Project (2016YFC0106305). We also would like to thank the anonymous reviewers for their valuable and constructive comments.

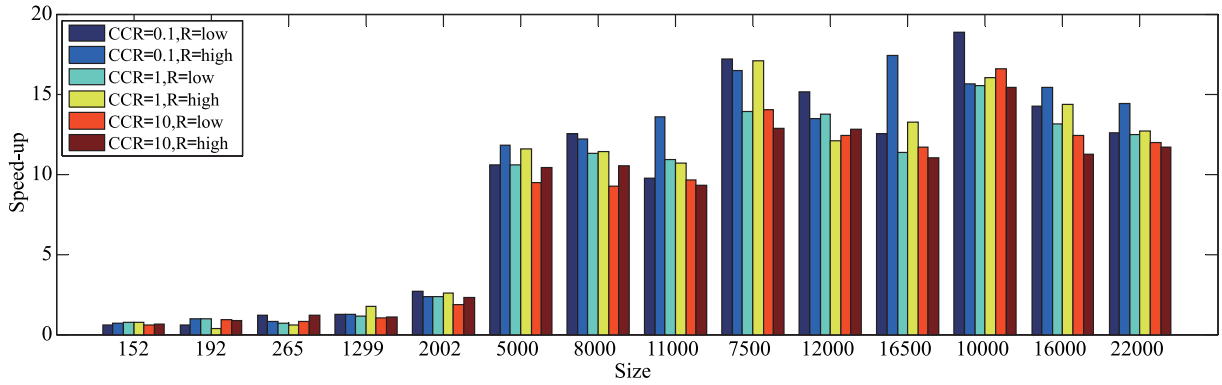


Fig. 16 Speedups in different task graphs in the six cases

References

1. De Michell G, Gupta R K. Hardware/software co-design. Proceedings of the IEEE, 1997, 85(3): 349–365
2. Wolf W. A decade of hardware/software co-design. Computer, 2003, 6(4): 38–43
3. Teich J. Hardware/software co-design: the past, the present, and predicting the future. Proceedings of the IEEE, 2012, 100: 1411–1430
4. Ouyang A, Peng X, Liu J, Sallam A. Hardware/software partitioning for heterogeneous MPSoC considering communication overhead. International Journal of Parallel Programming, 2017, 45(4): 899–922
5. Hou N, Yan X, He F. A survey on partitioning models, solution algorithms and algorithm parallelization for hardware/software co-design. Design Automation for Embedded Systems, 2019, 23(1–2): 57–77
6. Shi W, Wu J, Lam S, Srikanthan T. Algorithms for bi-objective multiple-choice hardware/software partitioning. Computers & Electrical Engineering, 2016, 50: 127–142
7. Dick R P, Rhodes D L, Wolf W. TGFF: task graphs for free. In: Proceedings of the 6th International Workshop on Hardware/Software Co-design. 1998, 97–101
8. Henkel J, Ernst R. An approach to automated hardware/software partitioning using a flexible granularity that is driven by high-level estimation techniques. IEEE Transactions on Very Large Scale Integration Systems, 2001, 9(2): 273–289
9. Jiang G, Wu J, Lam S, Srikanthan T, Sun J. Algorithmic aspects of graph reduction for hardware/software partitioning. The Journal of Supercomputing, 2015, 71(6): 2251–2274
10. Arató P, Juhász S, Mann Z, Orbán A, Papp D. Hardware-software partitioning in embedded system design. In: Proceedings of IEEE International Symposium on Intelligent Signal Processing. 2003, 197–202
11. Arató P, Mann Z, Orbán A. Algorithmic aspects of hardware/software partitioning. ACM Transactions on Design Automation of Electronic Systems, 2005, 10(1): 136–156
12. Zhou Y, He F, Hou N, Qiu Y. Parallel ant colony optimization on multi-core SIMD CPUs. Future Generation Computer Systems, 2018, 79(2): 473–487
13. Wang R, Hung W, Yang G, Song X. Uncertainty model for configurable hardware/software and resource partitioning. IEEE Transactions on Computers, 2016, 66(10): 3217–3223
14. Yan X, He F, Hou N, Ai H. An efficient particle swarm optimization for large scale hardware/software co-design system. International Journal of Cooperative Information Systems, 2018, 27(1): 1741001
15. Trindade A, Cordeiro L. Applying SMT-based verification to hardware/software partitioning in embedded systems. Design Automation for Embedded Systems, 2016, 20(1): 1–19
16. Li H, He F, Yan X. IBEA-SVM: an indicator-based evolutionary algorithm based on pre-selection with classification guided by SVM. Applied Mathematics—A Journal of Chinese Universities, 2019, 34(1): 1–26
17. Luo J, He F, Yong J. An efficient and robust bat algorithm with fusion of opposition-based learning and whale optimization algorithm. Intelligent Data Analysis, 2020, 24(3): 500–519
18. Yong J, He F, Li H, Zhou W. A novel bat algorithm based on cross boundary learning and uniform explosion strategy. Applied Mathematics—A Journal of Chinese Universities, 2019, DOI: 10.1007/s11766-019-3714-1
19. Gupta R, Micheli G. Hardware-software co-synthesis for digital systems. IEEE Design & Test of Computers, 1993, 10(3): 29–41
20. Ernst R, Henkel J, Benner T. Hardware - software co-synthesis for microcontrollers. IEEE Design & Test of Computers, 1993, 10(4): 64–75
21. Dick R, Jha N. MOGAC: a multi-objective genetic algorithm for hardware-software co-synthesis of distributed embedded systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1998, 17(10): 920–935
22. Wang G, Gong W, Kastner R. Application partitioning on programmable platforms using the ant colony optimization. Journal of Embedded Computing, 2006, 2(1): 119–136
23. Ferrandi F, Lanzi P, Pilato C, Sciuto D, Tumeo A. Ant colony optimization for mapping, scheduling and placing in reconfigurable systems. In: Proceedings of IEEE NASA/ESA Conference on Adaptive Hardware and Systems. 2013, 47–54
24. Koudil M, Benatchba K, Tarabet A. Using artificial bees to solve partitioning and scheduling problems in co-design. Applied Mathematics and Computation, 2007, 186(2): 1710–1722
25. Abdelhalim M, Habib S. An integrated high-level hardware/software partitioning methodology. Design Automation for Embedded Systems, 2011, 15(1): 19–50
26. Garg K, Aung Y, Lam S. Knapsim-run-time efficient hardware-software partitioning technique for FPGAs. In: Proceedings of the 28th

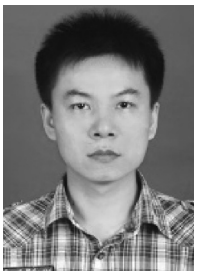
IEEE International Conference on System-on-Chip. 2015, 64–69

27. Zhang Y, Luo W, Zhang Z, Li B, Wang X. A hardware/software partitioning algorithm based on artificial immune principles. Applied Soft Computing, 2008, 8(1): 383–391
28. Jiang Y, Zhang H, Jiao X, Song X, Hung W, Gu M, Sun J. Uncertain model and algorithm for hardware/software partitioning. In: Proceedings of IEEE Computer Society Annual Symposium on VLSI. 2012, 243–248
29. Li G, Feng J, Wang C, Wang J. Hardware/software partitioning algorithm based on the combination of genetic algorithm and tabu search. Engineering Review, 2014, 34(2): 151–160
30. Yan X, He F, Chen Y. A novel hardware/software partitioning method based on position disturbed particle swarm optimization with invasive weed optimization. Journal of Computer Science and Technology, 2017, 32(2): 340–355
31. Kalavade A, Subrahmanyam P. Hardware/software partitioning for multi-function systems. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 1998, 17(9): 819–837
32. Govil N, Shrestha R, Chowdhury S. PGMA: an algorithmic approach for multi-objective hardware software partitioning. Microprocessors and Microsystems, 2017, 54: 83–96
33. Farahani A, Kamal M, Salmani-Jelodar M. Parallel genetic algorithm based HW/SW partitioning. In: Proceedings of International Symposium on Parallel Computing in Electrical Engineering. 2006, 337–342
34. Wu Y, Zhang H, Yang H. Research on parallel HW/SW partitioning based on hybrid PSO algorithm. In: Proceedings of International Conference on Algorithms and Architectures for Parallel Processing. 2009, 449–459
35. Pan Y, He F, Yu H, Li H. Learning adaptive trust strength with user roles of truster and trustee for trust-aware recommender systems. Applied Intelligence, 2019, DOI: 10.1007/s10489-019-01542-0
36. Lv X, He F, Cai W, Cheng Y. An optimized RGA supporting selective undo for collaborative text editing systems. Journal of Parallel and Distributed Computing, 2019, 132: 310–330
37. Li K, He F, Yu H. Robust visual tracking based on convolutional features with illumination and occlusion handing. Journal of Computer Science and Technology, 2018, 33(1): 223–236
38. Yu H, He F, Pan Y. A novel region-based active contour model via local patch similarity measure for image segmentation. Multimedia Tools and Applications, 2018, 77(18): 24097–24119
39. Van Luong T, Melab N, Talbi E. GPU computing for parallel local search meta-heuristic algorithms. IEEE Transactions on Computers, 2013, 62(1): 173–185
40. Zhou Y, He F, Qiu Y. Dynamic strategy based parallel ant colony optimization on GPUs for TSPs. Science China Information Sciences, 2017, 60(6): 068102.
41. Zhu W, Curry J, Marquez A. SIMD tabu search for the quadratic assignment problem with graphics hardware acceleration. International Journal of Production Research, 2010, 48(4): 1035–1047
42. Wei K, Sun X, Chu H, Wu C. Reconstructing permutation table to improve the tabu search for the PFSP on GPU. The Journal of Supercomputing, 2017, 73(11): 4711–4738
43. Bukata L, Šůcha P, Hanzálek Z. Solving the resource constrained project scheduling problem using the parallel tabu search designed for the CUDA platform. Journal of Parallel and Distributed Computing, 2015, 77: 58–68
44. Hou N, He F, Chen Y, Zhou Y. An adaptive neighborhood taboo search on GPU for hardware/software co-design. In: Proceedings of the 20th International Conference on Computer Supported Cooperative Work in Design. 2016, 239–244
45. Hou N, He F, Zhou Y, Ai H. A GPU-based tabu search for very large hardware/software partitioning with limited resource usage. Journal of Advanced Mechanical Design, Systems, and Manufacturing, 2017, 11(5): JAMDSM0060
46. Wu J, Srikanthan T, Chen G. Algorithmic aspects of hardware/software partitioning: 1D search algorithms. IEEE Transactions on Computers, 2010, 59(4): 532–544
47. Wu J, Wang P, Lam S, Srikanthan T. Efficient heuristic and tabu search for hardware/software partitioning. The Journal of Supercomputing, 2013, 66(1): 118–134
48. Chen Z, Wu J, Song G, Chen J. Noderank: an efficient algorithm for hardware/software partitioning. Chinese Journal of Computers, 2013, 36(10): 2033–2040
49. Quan H, Zhang T, Liu Q, Guo J, Wang X, Hu R. Comments on algorithmic aspects of hardware/software partitioning: 1D search algorithms. IEEE Transactions on Computers, 2014, 4(63): 1055–1056
50. Billeter M, Olsson O, Assarsson U. Efficient stream compaction on wide SIMD many-core architectures. In: Proceedings of the Conference on High Performance Graphics. 2009, 159–166
51. Wilt N. The Cuda Handbook: a Comprehensive Guide to GPU Programming. Pearson Education, 2013
52. Gupta K, Stuart J, Owens J. A study of persistent threads style GPU programming for GPGPU workloads. In: Proceedings of Innovative Parallel Computing. 2012, 1–14
53. Guthaus M, Ringenberg J, Ernst D, Austin T, Mudge T, Brown R. MiBench: a free, commercially representative embedded benchmark suite. In: Proceedings of IEEE International Workshop on Workload Characterization. 2001, 3–14
54. Pan Y, He F, Yu H. A novel enhanced collaborative autoencoder with knowledge distillation for top-n recommender systems. Neurocomputing, 2019, 332: 137–148
55. Zhang S, He F, Ren W, Yao J. Joint learning of image detail and transmission map for single image dehazing. The Visual Compute, 2018, DOI: 10.1007/s00371-018-1612-9
56. Chen X, He F, Yu H. A matting method based on full feature coverage. Multimedia Tools and Applications, 2019, 78(9): 11173–11201
57. Yu H, He F, Pan Y. A novel segmentation model for medical images with intensity inhomogeneity based on adaptive perturbation. Multimedia Tools and Applications, 2019, 78(9), 11779–11798
58. Fang F, Yi M, Feng, H, Hu S, Xiao C. Narrative collage of I mage collections by scene graph recombination. IEEE Transactions on Visualization and Computer Graphics, 2018, 24(9): 2559–2572
59. Wu Y, He F, Zhang D, Li X. Service-oriented feature-based data exchange for cloud-based design and manufacturing. IEEE Transactions on Services Computing, 2018, 11(2): 341–353
60. Pan Y, He F, Yu H. A correlative denoising autoencoder to model social influence for top-*N* recommender system. Frontiers of Computer Science, 2020, 14(3): 143301
61. Lv X, He F, Yan X, Wu Y, Cheng Y. Integrating selective undo of

feature-based modeling operations for real-time collaborative CAD systems. Future Generation Computer Systems, 2019, 100: 473–497

62. Li K, He F, Yu H, Chen X. A parallel and robust object tracking approach synthesizing adaptive Bayesian learning and improved incremental subspace learning. Frontiers of Computer Science, 2019, 13(5): 1116–1135

63. Yang L, Yan Q, Fu Y, Xiao C. Surface reconstruction via fusing sparse-sequence of depth images. IEEE Transactions on Visualization and Computer Graphics, 2018, 24 (2): 1190–1203



Neng Hou received PhD degree from Wuhan University, China in 2018. He is currently a lecture at School of Computer Science in Yangtze University, China. His research interests include HW/SW Co-design and GPU computing.

Fazhi He is currently a professor at School of Computer Science in Wuhan University, China. His research interests include computer-aided design, computer graphics, image processing, intelligent computing.

Yi Zhou is currently a lecture at School of Information Science and Engineering in Wuhan University of Science and Technology, China. His research interests include multi-core CPU and Many-core GPU based metaheuristics.

Yilin Chen is currently a PhD candidate at the School of Computer Science in Wuhan University, China. His research interests include GPGPU in computer graphics.