AVOIDING DUPLICATED COMPUTATION TO IMPROVE THE PERFORMANCE OF PFSP ON CUDA GPUS

Chao-Chin Wu[1*], Kai-Cheng Wei[1], Wei-Shen Lai[2], Yun-Ju Li[1]

[1]Department of Computer Science and Information Engineering, National Changhua University of Education, Changhua 500, Taiwan
ccwu@cc.ncue.edu.tw, kcwei@cc.ncue.edu.tw, icecloud6666@gmail.com
[2]Department of Information Management,
Chienkuo Technology University, Changhua 500, Taiwan
weishenlai@gmail.com

ABSTRACT

Graphics Processing Units (GPUs) have been emerged as powerful parallel compute platforms for various application domains. A GPU consists of hundreds or even thousands processor cores and adopts Single Instruction Multiple Threading (SIMT) architecture. Previously, we have proposed an approach that optimizes the Tabu Search algorithm for solving the Permutation Flowshop Scheduling Problem (PFSP) on a GPU by using a math function to generate all different permutations, avoiding the need of placing all the permutations in the global memory. Based on the research result, this paper proposes another approach that further improves the performance by avoiding duplicated computation among threads, which is incurred when any two permutations have the same prefix. Experimental results show that the GPU implementation of our proposed Tabu Search for PFSP runs up to 1.5 times faster than another GPU implementation proposed by Czapiński and Barnes.

KEYWORDS

GPU, CUDA, Parallel algorithm, Tabu Search, Permutation Flowshop Scheduling Problem

1. INTRODUCTION

GPUs (Graphics Processing Units) have been emerged as powerful parallel compute platforms for various application domains. A GPU consists of hundreds, even more than one thousand, of processing elements, making it very suitable for executing applications with big data and data-level parallelism [1, 2]. Compute Unified Device Architecture (CUDA) [3-5] is proposed by nVIDIA for easier programming on nVIDIA GPUs. Due to the low cost and the popular GPU-inside desktops and laptops, more and more researchers focus on how to parallelize various algorithms on GPU architecture. On the other hand, computational intelligence has been successfully applied to solve many kinds of applications [6-9]. Researchers have investigated how to use GPU computing to accelerate computational intelligence. For example, *Janiak et al.* [10] proposed the GPU implementations of the Tabu Search algorithm for the Travelling Salesman Problem and the Permutation Flowshop Scheduling Problem. Lots of research has

Natarajan Meghanathan et al. (Eds) : ICAIT, CRYPIS, NC, ITCSE-2016
pp. 13–23, 2016. © CS & IT-CSCP 2016 DOI : 10.5121/csit.2016.60702

reported that the optimized GPU implementations can run tens of times, or even more than one hundred times, faster than their sequential CPU counterparts.

The Tabu Search algorithm is a neighbourhood-based and deterministic metaheuristic, which is proposed to solve many discrete optimisation problems by *Glover* [11, 12]. This algorithm is similar to the function of human's memory. If the solution has been chosen by the previous generation, then it cannot be chosen again until a specified time interval has passed. This way can avoid choosing the local optimal solution to the problems. While computing the flowtime of the permutations, we use the Tabu list to record which permutations have been chosen to produce local optimal solutions during the previous several generations. In addition, users can set an initial value for the so called Tabu value, which determines how many generations the corresponding permutation cannot be used again since the permutation is selected. Whenever a permutation is selected, it is added into the Tabu list and its corresponding Tabu value is set to the user specified input value. Each Tabu value in the Tabu list will be decreased by one whenever proceeding to the next generation. The permutations in the Tabu list cannot be used until its corresponding Tabu value becomes zero. How to optimizing Tabu search on GPUs has been discussed on several projects [13-15].

The Permutation Flowshop Scheduling Problem (PFSP) has been first proposed by Johnson [16] in 1954. The PFSP is to find the best way to schedule many jobs to be processed on several ordered machines, which minimizes the flowtime that is equal to the total processing time of a permutation of the jobs. PFSP can be applied to the manufacturing and resources management in factories and companies. Due to the large number of jobs, the sequential program for PFSP is too slow to be adopted. GPUs have been adopted to solving the PFSP by using the Tabu search [14, 17]. To compute the flowtime of all permutations on GPUs, the previous work proposed placing all the permutations in the global memory initially to avoid branch divergence [14]. These permutations are produced by CPU sequentially. In each generation, each thread will read a permutation from the global memory. For efficient global memory access, the authors of Reference [10] proposed a data placement method that enables coalesced global memory accesses. They arrange all the permutations in an interleaving way. In other words, all the *i*-th elements of C^N_2 permutations are stored in the global memory contiguously. Following the i-th elements are the contiguous C^N_2 (*i*+1)-th elements. Nevertheless, it takes time to read the permutations from the global memory in each generation. The latency of global memory access is about 300 to 400 cycles. Previously, we have address this problem about how to create the appropriate numbers of threads and blocks and efficiently manage the shared memory [17]. Moreover, we propose using a math function to generate all the permutations on the fly, without the need of generating all the permutations by CPU and placing them on the global memory.

To solve the PFSP, in each generation of Tabu search, every thread will exchange two positions of the parent permutation to generate its child permutation. In the previous work [14,17], every thread has to compute the flowtime by constructing the whole completion time table. However, we have observed the following feature. If two child the two corresponding permutations share the same prefix, completion time tables contain several identical column data between them. More precisely, the number of identical columns equals to the legth of the same prefix .Therefore , there is much duplicated computation between threads in the previous work[14.17]. We will address this issue in this paper. Compared with the sequential CPU version, our new approach can run up to 1.5 times faster.

This paper is organized as follows. Section 2 introduces the CUDA architecture, the Permutation Flowshop Scheduling Problem, and related parallel methods. In Section 3, our proposed approach for implementing the PFSP on a CUDA GPU is described in detail. Section 4 demonstrates the experimental results and analyse the performance. Finally, conclusions are given in Section 5.

2. RELATED WORK

2.1. Compute unified device architecture

The CUDA (Compute Unified Device Architecture) development environment is mainly based on a sequential programming language, such as C/C++, and extended with some special functions that hide most issues of GPUs [3-5]. A GPU consists of several streaming multiprocessors (SMs) and each SM has multiple streaming processor cores [1-2]. From the software perspective, a CUDA's device program is organized as a hierarchy of grids, blocks and threads. To design a CUDA device program, programmers must define a C/C++ function, called kernel. While a CPU invokes a kernel to execute the kernel on GPU, the programmer must specify the number of blocks and the number of threads to be created. A block will be allocated to a SM and the threads within a block are able to communicate each other through the shared memory in the SM. Each thread is executed on a streaming processor. One or more blocks can be executed concurrently on a streaming multiprocessor at a time. There are hundreds or even thousands of threads within a block on CUDA. These threads can be organized as a 1-, 2- or 3-dimensional array, as shown in Figure 1. However, blocks can be organized as only a 1-, or 2-dimensional array.

There are many types of memory on GPU. They have different size, access time, and whether they can be written or read by blocks and threads. The description of each memory type is as below. Global memory is the main memory on a GPU, it can be allocated and deallocated explicitly through invoking the CUDA APIs in the kernel to communicate the CPU with the GPU. It has the largest memory space on the GPU, but it requires 400-600 clock cycles to complete a read or write operation. Blocks can communicate with each other via the global memory.

Constant memory is accessible as global memory except it is cached. A read operation takes the same time as that for the global memory in the case of a cache miss, otherwise it is much faster. The CPU can write and read the constant memory. It is read-only for GPU threads. Shared memory is a very fast memory on the GPU, it is used to communicate between threads in the same block. Data in the shared memory of a block cannot be directly accessed by other blocks. Accessing shared memory requires only 2-4 clock cycles. Unfortunately, the memory space of shared memory is limited. The maximum space is 16384 bytes per block for Tesla C1060. When a thread needs more space than the shared memory, the thread has to swap out and in the data in shared memory explicitly. Registers are the fastest memory that can only be used in the thread scope. They are for automatic variables. The number of 32-bit register is limited up to 16384 on each streaming multiprocessor on Tesla C1060. Local memory is used for large automatic variables per-thread, such as arrays. Both read and write operations take the same time as that for the global memory.

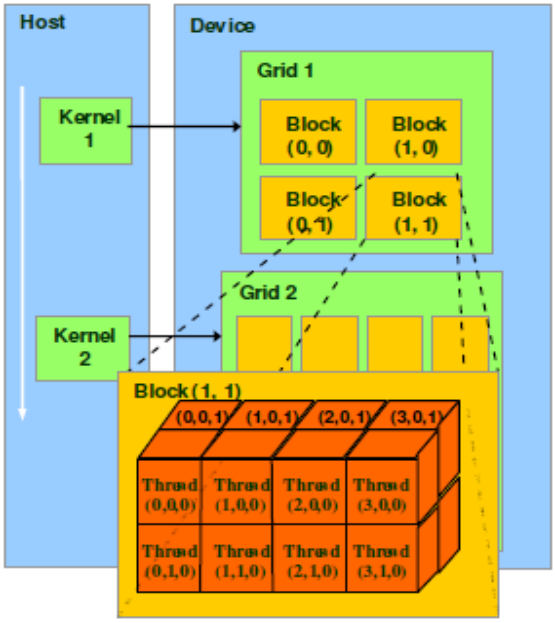

Figure 1. The relations between threads, blocks, and grids

2.2. Permutation Flowshop Scheduling Problem

In the PFSP, a set of N jobs is to be processed on a set of R machines. Each job will be divided into R parts and go through the R machines in a predefined order. Assume M_i is Machine i, and J_k is Job k. Let $P_{i,j}$ denote the processing time of Job k on Machine i. Compute the flowtime, denoted as $C_{i,k}$, for processing J_k on machine M_i, which is defined as the following formula. Each permutation has its own flowtime $C_{m,n}$.

$$C_{0,0} = p_{0,0},$$
$$C_{i,0} = p_{i,0} + C_{i-1,0},$$
$$C_{0,k} = p_{0,k} + C_{0,k-1},$$
$$C_{i,k} = p_{i,k} + \max\{C_{i,k-1}, C_{i-1,k}\},$$
$$\text{where} \quad i \in \{1,2,\ldots, m\}, \quad \text{and} \quad k \in \{1,2,\ldots, n\}$$

To solve the PFSP is to find the minimum of all flowtimes from all permutations. Let ω_i is a permutation, then $C_{m,n}(\omega_i)$ denotes the flowtime of the permutation ω_i. Ω_x denotes the set of all permutations of length x.

$$\forall \omega \in \Omega_x, C_{\max} = \max\{C_{m,n}(\omega_1), C_{m,n}(\omega_2),\ldots, C_{m,n}(\omega_x)\}$$

Because the PFSP is a NP problem, it has been parallelized to shorten its execution time. For instance, *Chakroun et al.* [10] used the branch-and-bound algorithm and the inter-task parallel method to improve the performance of the flowshop problem on GPUs. In the inter-task method, each thread calculates the flowtime for a permutation. Each thread is responsible for sequentially computing the flowtime for a permutation. The advantage is that the threads have no data dependency between each other in the block, so they do not need to synchronize with each other

or wait for another. The disadvantage is that each thread needs a large amount of the shared memory space for processing a permutation. It has low performance when more jobs and machines have to be processed because threads in the same block contend for the use of the shared memory. Due to the limitation of available shared memory space, the maximum number of threads per block cannot be very large.

On the other hand, the intra-task method let all the threads in a block process a permutation together. *Michael et al.* [11] used the intra-task method by well utilizing the characteristic of the GPU memory, such as memory coalescing for accessing the global memory, and avoiding bank conflict on the shared memory. They let each block be responsible for computing the flowtime of a permutation, where multiple threads in a block work together to compute the flowtime for a permutation. The advantage of the method is that a larger number of threads can execute the PFSP concurrently because of using less shared memory when the flowtime of a permutation is processed by a block. In other words, it means the elements in an anti-diagonal have no data dependency between each other. Unfortunately, this method has two drawbasks. First, the number of threads in each phase is not equivalent. It causes the waste of thread resources, due to the idle threads in some phases. Second, the elements in each anti-diagonal have to wait for the results produced by the elements in the previous anti-diagonal. It needs synchronization between threads and blocks, making it necessary to invoke one kernel for each phase.

3. AVOIDING DUPLICATED COMPUTATION

In this section, we describe the proposed approach of avoiding duplicated computation. Section 3.1 presents the relation between the completion tables of the parent permutation and the child permutation. Section 3.2 explains how we can use the above important observation to design an algorithm to accelerate the execution of the completion time tables for child permutations.

3.1. Observation

For the Tabu search for PFSP, in each generation, the permutations to be processed are generated based on the best processing order of jobs produced in the previous generation. If there are N jobs, there will be C^N_2 permutations at most to be processed in each generation, where any permutation leading to a job processing order the same as one in the Tabu list will be prohibited in the generation.

In each generation of the Tabu search, each thread will be assigned on permutation to calculate the flow time of the permutation. To accelerate the computation of the flow time, each thread will be allocated with M words on shared memory if there are M machines in the PFSP. Shared memory is fast memory for the scope of a CUDA block. The number of threads is limited by the available space of shared memory if each thread requires shared memory space. In other words, if each thread uses less shared memory space to process and compute the flowtime of a permutation, the block can have more threads. For PFSP, the number of machines is less than that of jobs in general. To keep the required shared memory as much as possible, the number of shared memory words per thread is equal to the number of machines. As shown in Figure 2, if there are 3 machines and 4 jobs, we allocate 3 shared memory words for a thread. Then, it computes sequentially according to the order of the permutation.

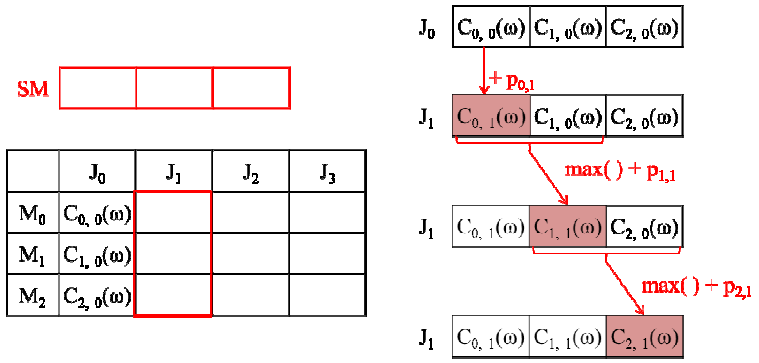

Fig. 2. The space allocation of shared memory, the completion time table, and the register reutilization for calculating completion time within the same column

For the thread to process the permutation, J_0, J_1, J_2, J_3, as show in Figure 2, it will calculate the completion time for J_0 on each machine, from M_0 to M_2 one by one, as show in Figure 2. Next, it continues the computations for the subsequent jobs, J_1, J_2, J_3, one by one, and finally obtain the flow time for the permutation. If we exchange the positions of J_2 and J_3 on the above permutation, new permutation, J_0, J_1, J_3, J_2, is generated. To compute the flow time, a thread will be assigned to perform the same procedures as that illustrated in Figure 2, except the job ordering. We depict two tables in Figure 3 to show the completion times for each job on different machines for the above two permutations. Note that each call of any table contains one completion time and all the completion time in one table are calculated one by one from top to bottom and from left to night. As a result, the first two columns in both tables have the same contents, respectively because the first two jobs in the two permutations are both J_0 and J_1. However, the two third columns in the tables are different because the third job in the original permutation is J_3 but the third Job in the new permutation is J_2. Since the calculation of i-th column depends in the results in *(i-1)-th* column, the results in the two third columns are different. Furthermore, for the following columns, the same columns on both tables have different values.

In general, assume there are a parent permutation, π_0, π_1, π_2, … …, π_n and a child permutation is derived from the parent permutation by exchanging two positions, π_i and π_j , where i<j and $0 \leqq i \leqq n$, $0 \leqq j \leqq n$, The first (i-1) columns in the two corresponding tables of completion time will have the same contents. For the subsequent columns in the two tables any pair of two columns with the same column number must have different completion times. As a result, if we have the completion time table of the parent permutation, we can calculate the flowtime of the child permutation from the *i-th* column after copying the *(i-1)-th* column in the completion table for the parent permutation. In fact the computation of the first *(i-1)* columns in the completion time table for the child permutation is redundant if we are given the completion time table for the parent permutation.

To solve the PFSP, at most C^N_2 child permutations will be generated in each generation of Tabu search based on the parent permutation, where each child permutation is obtained by exchanging two positions in the parent permutation. In previous world [14, 17], at most C^N_2 threads will be forked in each generation and each thread is assigned with one child permutation. All threads compute the flowtimes in parallel for their permutations because each flowtime computation depends on only the parent permutation. Since each thread constructs all the table of completion time for its assigned permutation from the scratch based on mainly the parent permutation, too

much redundant computation is performed, resulting in worse performance. Therefore, we propose an approach in the following subsection to accelerate the execution of solving the PFSP by using Tabu search on GPU.

	J_0	J_1	J_2	J_3
M_0	$C_{0,0}(\omega)$	$C_{0,1}(\omega)$	$C_{0,2}(\omega)$	$C_{0,3}(\omega)$
M_1	$C_{1,0}(\omega)$	$C_{1,1}(\omega)$	$C_{1,2}(\omega)$	$C_{1,3}(\omega)$
M_2	$C_{2,0}(\omega)$	$C_{2,1}(\omega)$	$C_{2,2}(\omega)$	$C_{2,3}(\omega)$

(a) The completion time table of the permutation: J_0, J_1, J_2, J_3

	J_0	J_1	J_3	J_2
M_0	$C_{0,0}(\omega)$	$C_{0,1}(\omega)$	$C_{0,3}(\omega)$	$C_{0,2}(\omega)$
M_1	$C_{1,0}(\omega)$	$C_{1,1}(\omega)$	$C_{1,3}(\omega)$	$C_{1,2}(\omega)$
M_2	$C_{2,0}(\omega)$	$C_{2,1}(\omega)$	$C_{2,3}(\omega)$	$C_{2,2}(\omega)$

(b) The completion time table of the permutation: J_0, J_1, J_3, J_2

Fig. 3. Comparison between completion time tables of two permutations, where two positions are different in the permutations

3.2. Our Proposed Approach

Instead of computing the flowtime from the empty completion time table as that adopted in the previous for each child permutation work [14, 17], we start the computation from the *i-th* column after copying the *(i-1)-th* column from the parent's completion time table. To achieve the goal, we have to store all the completion time table for the parent permutation in the global memory, which is not necessary in the previous work [14, 17]. Figure 4 demonstrates an example of our proposed approach. The completion time table for the parent permutation, (1, 2, 3, 4, 5, 6), is calculated and stored in the global memory. Assume the thread T_1, will exchange Jobs 3 and 6 in the parent permutation to produce his child permutation, (1, 2, 6, 4, 5, 3). T_1 has to fetch the whole second column in the completion time table of the parent permutation in the global memory and save the column data into shared memory. Following the similar computation procedures shown in Figure 2, T_1 can calculate the flowtime for its child permutation. In this way, T_1 can avoid the computation of the first two columns, resulting in a shorter execution time. Similarly, if T_2 will exchange Jobs 4 and 6 in the parent permutation, it has to fetch the third column from the global memory, which is used to calculate the completion times for the following columns and derive the flowtime. Totally, the computation of three columns are avoided for T_2. After the flowtimes for all possible child permutation are produced, we will select the permutation with the minimum flowtime to become the parent permutation for the next generation. However, in fact, the completion time table of the newly selected parent permutation does not exist because all column data are stored in the same shared memory of one-table-column size, as shown in Figure 4. At the end, only the last column data are stored in shared memory.

Note that it is impossible to store all the information about the whole completion time table in shared memory for every thread because of the limited shared memory space. Also it is impossible to know which child permutation will become the parent permutation for the next generation before the flowtimes of all possible child permutations are calculated. One possible solution to address the above problem is that every thread writes all its column data to the global memory. However, this solution will result in high overhead due to a large amount of long-latency global memory access.

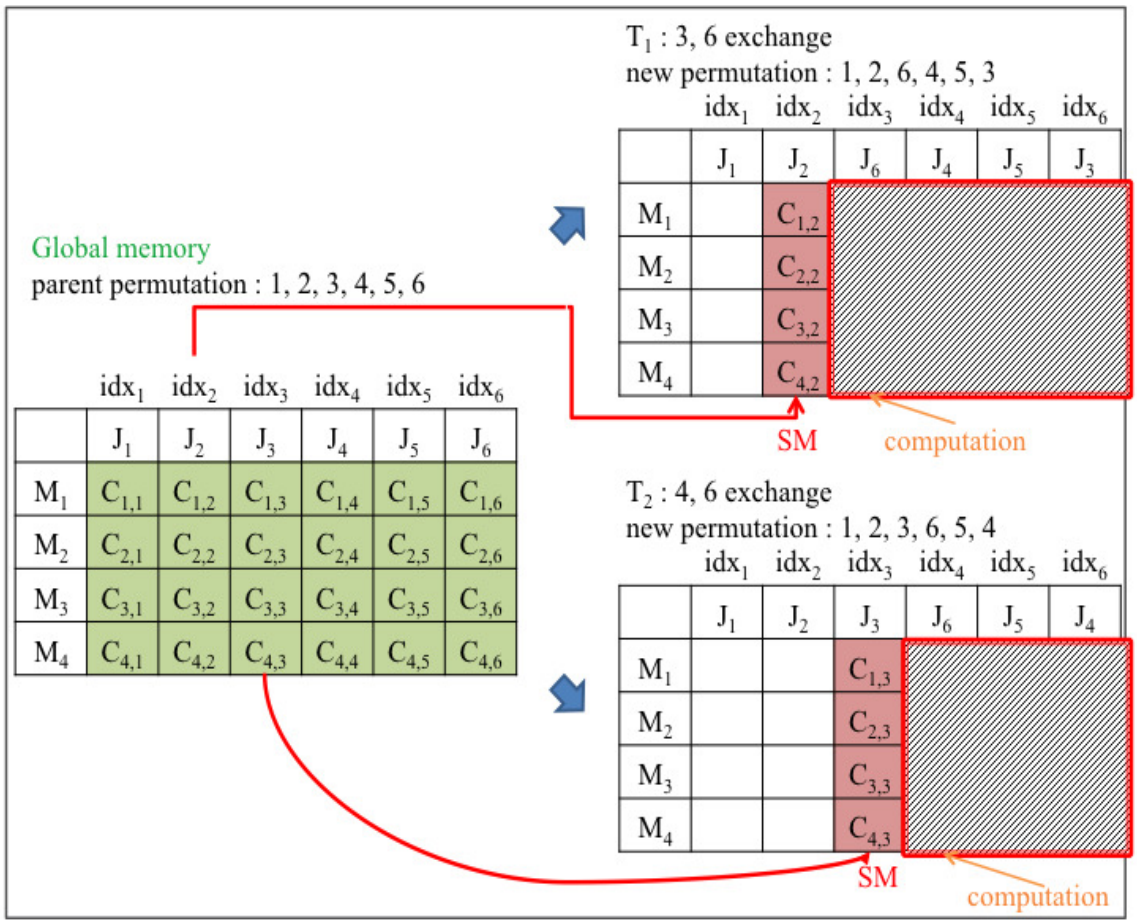

Fig. 4 Avoiding duplicated computation when calculating the completion time table of child permutations from the one of the parent permutation.

We adopt another solution to address the above problem. After the new parent permutation is selected, we use one thread block to calculate the completion time table of the new parent permutation and store the table in global memory. In our proposed approach, we need to construct the completion time table for the next parent permutation, which is the unique overhead for our approach comparing the previous work [14, 17]. Therefore, minimizing the execution time of constructing the table is the key issue of the success of our proposed approach. We parallelize the above table construction with a single thread block [11], as shown in Figure 5. Because of the data dependency, the completion time table construction, is parallelized diagonally. In the example shown in Figure 5, there are 4 machines and 4 jobs. The table construction consists of 7 phases, indicated by dash lines with numbers. The maximum number of threads required is 4. Between any two consecutive phases, we need to insert a synchronization to enforce data consistency between threads.

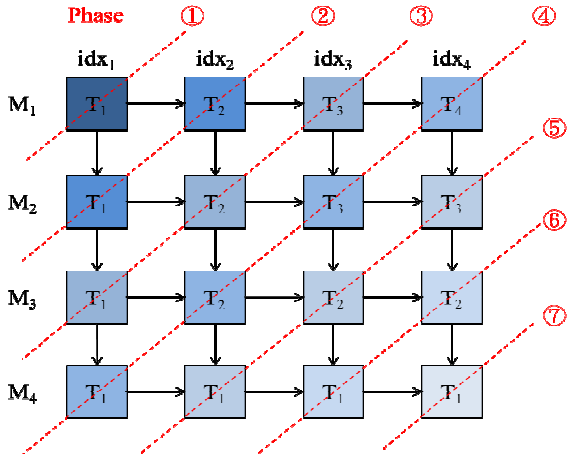

Fig. 5. The parallelization of building the completion time table of the next parent permutation by one thread block.

4. EXPERIMENT RESULTS

The Tabu Search for PFSP is written in C and evaluated on an Intel Pentium 2.5 GHz CPU with 2 GB memory and NVIDIA Tesla C2050 with 448 CUDA cores and 2.6 GB memory. Detailed configurations are shown in Table 1. We use CUDA version 4.2 to implement both the approaches of ours and Czapiński and Barnes', for the Permutation Flowshop Scheduling Problem using the Tabu Search algorithm. The operating system installed is Linux and its version is Ubuntu 11.10, 32-bit.

Table 1. The specifications of the Intel Pentium CPU and the NVIDIA Tesla C2050.

Intel® Pentium® D		NVIDIA Tesla C2050	
# of Cores	2	# of GPUs	1
# of Threads	2	Processor cores	448
Clock Speed	3GHz	Clock Speed	1.15GHz
Memory Size	2GB	Memory Size	2.6GB
Memory Types	DDR2 667	Memory Types	GDDR5
Cache	2MB	Memory Clock	800MHz

We show the speedups of our approach over the Czapiński and Barnes' method in Table 2, where we vary the numbers of the machines, jobs and generations. The speedup is derived from dividing the execution time of our approach by the execution time of Czapiński and Barnes' method. Also, the shared memory size (SMs) is either 16 MB or 48 MB. When the product of (# of machines) and (# of jobs) is smaller than or equal to 3000, our approach would degrade the performance. The reason is as follows. (1) The number of columns that we have no need to re-calculate is rather limited. (2) The computation time of constructing the next parent permutation the completion time table of significantly increases the critical path of the whole execution. On the other hand, when the product is bigger than or equal to 5000, our approach outperforms the previous work. The larger the product, the higher the speedup. The reason is because we can avoid more duplicated computation for larger problem sizes.

Table 2. Speedups of Tabu Search for PFSP, compared with the Czapiński and Barnes' method

# of generations			10		100		1000	
# of ma-chines	# of jobs	SMs m * j	16 MB	48 MB	16 MB	48 MB	16 MB	48 MB
15	100	1500	0.82	0.83	0.65	0.64	0.61	0.61
20	150	3000	0.94	0.95	0.87	0.87	0.85	0.85
25	200	5000	1.01	1.01	1.03	1.03	1.03	1.03
25	350	8750	1.11	1.12	1.28	1.28	1.33	1.33
30	500	15000	1.15	1.14	1.3	1.3	1.34	1.34
30	650	19500	1.15	1.14	1.32	1.32	1.36	1.36
35	800	28000	1.2	1.2	1.35	1.34	1.37	1.37
40	900	36000	1.31	1.31	1.47	1.47	1.5	1.5

5. CONCLUSION

In this paper, an approach of avoiding duplicated computation was presented for the Tabu Search algorithm to solve PFSP on a CUDA GPU. In the previous work, each thread has to calculate the whole completion time table for its assigned child permutation in every iteration. However, we have observed that most child permutations has the same prefix as the parent permutation. Using this observation, we have proposed a new approach. One thread block builds the completion time table of the next parent permutation in parallel and stores the table in the global memory. Each thread fetches the table data of the column, from the global memory, corresponding to the last job in the same prefix. Next, each thread calculates the flowtime according to the column data, without the need of constructing the whole completion time table for its child permutation. Experimental results demonstrated our approach has the best speedup up to 1.5, comparing with the previous work.

In further work, we will apply more optimization techniques of CUDA and utilize the features of a GPU workstation to optimize the Tabu search algorithm, such as how to efficiently manage device memories, synchronize blocks, and reduce the number of computing subtasks.

ACKNOWLEDGMENT

The authors would like to thank the Ministry of Science and Technology, Taiwan, for financially supporting this research under Contract No. MOST104-2221-E-018-007.

REFERENCES

[1] Owens, J.D., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A.E., Purcell, T.J.: A survey of general-purpose computation on graphics hardware. Computer Graphics Forum 26, pp. 80–113, (2007)

[2] NVIDIA GPU, http://www.nvidia.com/object/cuda_home_new.html.

[3] NVIDIA GPU Programming Guide, http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html.

[4] Kirk, D.B., Hwu, W.W.: Programming Massively Parallel Processors. NVIDIA.

[5] Oster, Brent: Programming the CUDA Architecture: A Look at GPU Computing. Electronic Design, Vol. 57, Issue 7. (2009)

[6] Ge, M., Wang, Q.-G., Chiu, M.-S., Lee, T.-H., Hang, C.-C., Teo, K.-H.: An effective technique for batch process optimization with application to crystallization. Chemical Engineering Research and Design, Vol. 78, No. 1, pp. 99-106. (2000)

[7] Precup, R.-E., David, R.-C., Petriu, E. M., Preitl, S. Radac, M.-B.: Novel adaptive gravitational search algorithm for fuzzy controlled servo systems. IEEE Transactions on Industrial Informatics, Vol. 8, No. 4, pp. 791–800. (2012)

[8] Saha, S. K., Ghoshal, S. P., Kar, R. Mandal, D. Cat swarm optimization algorithm for optimal linear phase FIR filter design. ISA Transactions, Vol. 52, No. 6, pp. 781-794. (2013)

[9] Yazdani, D., Nasiri, B., Azizi, R. Sepas-Moghaddam, A., Meybodi, M. R.: Optimization in dynamic environments utilizing a novel method based on particle swarm optimization. International Journal of Artificial Intelligence, Vol. 11, No. A13, pp. 170-192. (2013)

[10] Bożejko, W., Wodecki, M.: Parallel genetic algorithm for the flow shop scheduling problem. Lecture Notes in Computer Science, Vol.3019, pp.566–571. (2004)

[11] Glover, F.: Tabu search—part I. ORSA Journal on Computing 1, Vol.3, pp.190-206. (1989)

[12] Glover, F.: Tabu search—part II. ORSA Journal on Computing 2, Vol.1, pp.4-32. (1990)

[13] Janiak, A., Janiak, W., Lichtenstein, M.: Tabu search on GPU. Journal of Universal Computer Science 14, Vol.14, pp.2416–2427. (2008)

[14] Czapiński, M., Barnes, S.: Tabu Search with two approaches to parallel flowshop evaluation on CUDA platform. J. Parallel Distrib. Comput., Vol.71, pp.802-811. (2011)

[15] Chakroun, I. Bendjoudi, A. Melab, N. Reducing Thread Divergence in GPU-Based B&B Applied to the Flow-Shop Problem. PPAM. (2011)

[16] Johnson, S.M.: Optimal two- and three-stage production schedules with setup times included. Naval Research Logistics Quarterly 1, Vol.1, pp.61-68. (1954)

[17] Liang-Tsung Huang, Syun-Sheng Jhan, Yun-Ju Li, Chao-Chin Wu, “Solving the Permutation Problem Efficiently for Tabu Search on CUDA GPUs,” 6th International Conference on Computational Collective Intelligence Technologies and Applications, LNAI 8733, pp. 342-352, 24th-26th September 2014, Seoul, Korea.