A GPU Parallel Approximation Algorithm for Scheduling Parallel Identical Machines to Minimize Makespan

Yuanzhe Li, Laleh Ghalami, Loren Schwiebert, and Daniel Grosu
Department of Computer Science
Wayne State University
Detroit, MI 48202, USA
{yuanzhe.li, laleh.ghalami, loren, dgrosu}@wayne.edu

***Abstract*—We present a parallel approximation algorithm for the problem of scheduling jobs on parallel identical machines to minimize makespan which is designed and optimized for running efficiently on the GPUs. The algorithm is a Polynomial Time Approximation Scheme (PTAS) based on a higher-dimensional dynamic programming approach, where dimensionality refers to the number of variables in the dynamic programming equation characterizing the problem. The main component of our design consists of a novel data-partitioning technique that is employed to accelerate the higher-dimensional dynamic programming component of the algorithm. We present performance results to demonstrate how our proposed design improves the GPU utilization and makes it possible to solve large higher-dimensional dynamic programming problems with the limited GPU memory. Experimental results show that the GPU implementation outperforms the optimized OpenMP implementation of the approximation algorithm.**

I. Introduction

Approximation algorithms for solving various combinatorial optimization problems have been studied extensively; however, only a few existing studies focus on parallel approximation algorithms. To the best of our knowledge, there is no work on developing efficient GPU implementations of parallel approximation algorithms. In this paper, we present a GPU optimized parallel approximation algorithm for solving the problem of scheduling jobs on parallel identical machines to minimize makespan (i.e., the $P \mid\mid C_{max}$ problem). An OpenMP implementation of an approximation algorithm for $P \mid\mid C_{max}$ is studied in [1]. The algorithm is a Polynomial Time Approximation Scheme (PTAS) based on an exact higher-dimensional dynamic programming approach, where the dimensionality refers to the number of variables in the dynamic programming equation characterizing the problem. Here, higher-dimensional refers to dynamic programming cases with three or more dimensions.

GPUs are widely used for parallel computations because they provide both considerable throughput and energy efficiency. The many-core architecture makes GPUs especially efficient for computationally-intensive problems. Modern GPUs, like the Kepler K40, offer additional hardware features to improve programmability and flexibility [2]. For example, Hyper-Q enables multiple connections between the CPU and the GPU, which allows up to 32 concurrent kernels launched by the CPU through mapping to different streams. Similarly, Dynamic Parallelism [3] enables a more natural decomposition of an application into parallel tasks with an adaptive mapping of resources.

However, porting applications to the GPU can be challenging due to the need to extract fine-grain parallelism, manage the hierarchical memory structure, and synchronize among many threads. For problems with an irregular structure or limited concurrency, a more sophisticated design is required, as a straightforward implementation is unlikely to achieve adequate performance.

In this paper, we not only adapt the parallel approximation algorithm for $P \mid\mid C_{max}$ to make it efficient on the GPU, but also consider the GPU implementation of the higher-dimensional dynamic programming procedure, which is an important component of the algorithm. Although algorithms for several dynamic programming problems have already been ported to the GPU, challenges still remain, especially for higher-dimensional cases. Dynamic programming solutions are built from the solutions to sub-problems limiting the degree of parallelism that can be exploited. Furthermore, the workload imbalance among sub-problems increases with the number of dimensions. In addition, solving higher-dimensional dynamic programming problems can quickly exceed the GPU memory.

Previous work investigated parallel dynamic programming, considering both coarse-grained (multiprocessor clusters) and fine-grained architectures (multicore CPUs, and many-core GPUs). A coarse-grained architecture, such as a multiprocessor cluster, usually achieves efficient local computations because of its powerful computational capabilities and large memory on each processor. But the inefficient inter-cluster communications and unbalanced workload are detrimental to the parallel performance. Some prior research has focused on reducing the inter-cluster communication, such as, partitioning the dynamic programming table into multiple rectangular segments [4, 5, 6]. In addition, work distribution schemes, like block-cyclic [5], have also been employed to balance the workload among processors. Implementing parallel dynamic programming on multicore CPUs and many-core GPUs

requires a more sophisticated design of work distribution among threads, as a fine-grained architecture has many more computing resources to run the same anti-diagonal levels of sub-problems concurrently, especially for the GPU. A strategy for computing successive anti-diagonals of the dynamic programming table was applied to the multicore CPU to maximize parallel execution [1, 7]. Several researchers [8, 9, 10, 11] extended the strategy to achieve more fine-grained parallelism on the GPU, mostly for two-dimensional dynamic programming problems. Very few researchers investigated accelerating higher-dimensional dynamic programming problems on the GPU.

To address "the curse of dimensionality", some researchers proposed approximations for higher-dimensional dynamic programming [12, 13, 14]. Despite the huge advances in parallel computing, the parallel implementation of exact higher-dimensional dynamic programming problems, especially on the GPU, is not as well studied as two-dimensional dynamic programming. Berger and Galea [15] implemented a higher-dimensional knapsack algorithm on the GPU by introducing the idea of combining coarse-grained parallelism and fine-grained parallelism, and improving the memory coalescing by fixing the number of dimensions. However, their technique works only on small problem sizes, as the size of a higher-dimensional table can grow quickly with the number of dimensions and is likely to exceed the GPU global memory.

Therefore, the major challenge of making approximation algorithms such as the one for $P \mid\mid C_{max}$, efficient on the GPU is to improve the execution of the higher-dimensional *DP* procedure. Our proposed techniques resolve the memory issues and improve the thread-level workload balance. Our evaluation on the GPU considers as many as nine dimensions in order to assess the optimal decomposition of the various problem instances. We compare the performance of our GPU implementation with that obtained by the OpenMP implementation on a multicore CPU. The results show that our techniques yield an efficient GPU approximation algorithm for $P \mid\mid C_{max}$, with very good performance on large problem instances.

The rest of the paper is organized as follows. In Section II, we introduce the details of the PTAS and the parallelization of the dynamic programming in the parallel version of the PTAS. In Section III, we present our design of the GPU parallel approximation algorithm for $P \mid\mid C_{max}$. Besides, we address the challenges of developing higher-dimensional *DP* procedures on the GPU and describe the data-partitioning scheme which is specifically designed for solving performance and memory issues. In Section IV, we investigate the performance of our GPU implementation by performing extensive experiments. In Section V, we conclude the paper and describe possible directions for future research.

II. Parallel PTAS and dynamic programming

Our efficient GPU parallel approximation algorithm extends a parallel approximation algorithm for the problem of scheduling jobs on parallel identical machines to minimize makespan proposed by Ghalami and Grosu [1]. Their parallel algorithm requires solving a higher-dimensional dynamic programming problem and is based on parallelizing the PTAS by Hochbaum and Shmoys [16]. In what follows, we call the algorithm presented in [1], the *parallel PTAS*. The basic idea of the PTAS is to partition the set of jobs into two sets, long and short jobs, round down the processing times of the long jobs, and find an optimal schedule for the rounded long jobs using the dynamic programming procedure. The parallelization of the dynamic programming procedure is the core of the *parallel PTAS*.

Algorithm 1 PTAS for $P||C_{max}$ by Hochbaum and Shmoys [16]

1: **Input:** $n, m, \mathcal{T} = \{t_1, ..., t_n\}, \epsilon$
2: $LB \leftarrow \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil, \max_{j=1,\ldots,n} t_j\right\}$
3: $UB \leftarrow \left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil + \max_{j=1,\ldots,n} t_j$
4: $k = \lceil 1/\epsilon \rceil$
5: **while** $LB < UB$ **do**
6: $\quad T = \lfloor (UB + LB)/2 \rfloor$
7: $\quad$ Partition jobs into short and long jobs
8: $\quad$ Round down long jobs to their nearest multiples of $\lfloor T/k^2 \rfloor$
9: $\quad OPT = DP(N, T)$
10: $\quad$ Obtain the schedule for rounded down long job sizes
11: $\quad$ **if** $OPT \leq m$ **then**
12: $\quad\quad UB = T$
13: $\quad$ **else**
14: $\quad\quad LB = T + 1$
15: Return the schedule

We now briefly describe the PTAS, presented in Algorithm 1. The algorithm requires as input, the number of machines, m; the number of jobs, n; the processing times of the jobs t_i, $i = 1, \ldots, n$; and the relative error $\epsilon > 0$. We denote by $\mathcal{T}$ the multiset of jobs' processing times, i.e., $\mathcal{T} = \{t_1, ..., t_n\}$, and assume that all jobs' processing times are positive integers. The algorithm starts by computing the lower and upper bounds (denoted by LB and UB) on the optimal makespan of the set of n jobs on m identical machines (Lines 2-3).

The algorithm performs a bisection search procedure for a target makespan value T on the interval $[LB, UB]$ and determines a schedule for the long jobs that fits within T. Next, it rounds down the processing times of the long jobs to their nearest multiples of $\lfloor T/k^2 \rfloor$, so that long jobs are classified into k^2 classes, where $k = \lceil 1/\epsilon \rceil$. Then, the algorithm determines the number of jobs of each of the rounded sizes and creates a k^2-dimensional vector $N = (n_1, \ldots, n_{k^2})$, where n_i is the number of rounded long jobs in class i. After creating the vector N, the algorithm finds a schedule for the long rounded jobs with a makespan within time T. This is done by employing the *DP* algorithm which determines the suitable number of machines to achieve a makespan within T. The *DP* algorithm generates the set $\mathcal{C}$ of all possible machine configurations. A *machine configuration* is a k^2-dimensional vector $(s_1, \ldots, s_{k^2})$ specifying an assignment of tasks to one machine and satisfying that the total rounded time is within T.

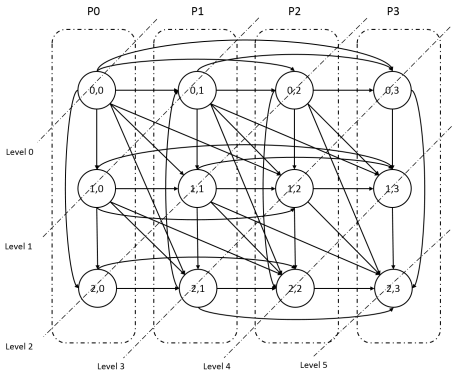


Fig. 1: Dependency graph for $OPT(2,3)$

The recurrence equation of the *DP* is given by

$$OPT(n_1,\ldots,n_{k^2}) = \\ 1 + \min_{(s_1,\ldots,s_{k^2})\in\mathcal{C}} OPT(n_1-s_1,\ldots,n_{k^2}-s_{k^2}). \quad (1)$$

where $OPT(n_1,\ldots,n_{k^2})$ is the minimum number of machines sufficient to schedule the set of jobs given by the vector N and leading to a makespan within T. The idea behind this recurrence is that a schedule assigns some jobs to one machine and then assigns the rest of the jobs to as few machines as possible. Hence, the values of $OPT(n_1,\ldots,n_{k^2})$ are the components of a dynamic programming table. Since k^2 is greater than 3, the *DP* procedure falls within the higher-dimensional dynamic programming. The dynamic programming formulation (Equation 1) also implies that the subproblems at each level depend on subproblems at more than one previous level. These subproblems correspond to the components of a table which we call the *DP-table*. Figure 1 shows the assignment of the subproblems for a two-dimensional *DP-table* to a parallel system composed of four cores.

The parallelization of the higher-dimensional *DP* is presented in Algorithm 2. The goal of the algorithm is to fill out the entire higher-dimensional *DP-table* and find the optimal value of $OPT(N)$. First, the algorithm determines the size of the DP-table, $\sigma = \prod_{i=1}^{k^2}(n_i+1)$ (Line 2). Next, the P processors compute the sums of the distances of the vectors v^i, $i = 1,\ldots,\sigma$ in parallel (Lines 4-8). Because of the dependencies between the anti-diagonals, the parallel *DP* algorithm consists of $n'+1$ sequential iterations, where n' is the number of long jobs. The subproblems on each level l (corresponding to anti-diagonal l) can be identified by the same d_i value (Line 12) and executed by P processors in parallel (Lines 11-25). For computing the optimal value of a subproblem, we need to know its dependencies on the preceding subproblems and use them in Equation (1). Therefore, the algorithm generates the set $\mathcal{C}_{v^i}$ of all possible machine configurations, $(s_1,\ldots,s_{k^2})$, for vector v^i (Line 17). Next, the algorithm finds the location of all subproblems by searching the entire *DP-table* and reads their optimal values $OPT(v_1^i - s_1,\ldots,v_{k^2}^i - s_{k^2})$. Then, it places the values into multiset $\mathcal{O}_{v^i}$ (Lines 18-19) and determines the minimum among all values of the subproblems currently in $\mathcal{O}_{v^i}$, adds 1 to the minimum and assigns the value to subproblem $OPT(v_1^i,\ldots,v_{k^2}^i)$ (Lines 20-25). The ordering of iterations guarantees that at each level the algorithm already computed all the needed preceding subproblems.

Algorithm 2 *Parallel* $DP(N,T)$ by Ghalami and Grosu [1]

1: **Input:** $N=(n_1,\ldots,n_{k^2}),T$
2: $\sigma \leftarrow (n_1+1)(n_2+1)\ldots(n_{k^2}+1)$
3: Let $v^i=(v_1^i,\ldots,v_{k^2}^i)$ and $OPT(v_1^i,\ldots,v_{k^2}^i)$ be the i-th entry of DP-table in row-major order, where $i=0,\ldots,\sigma-1$
4: **parallel for** $i=0,\ldots,\sigma-1$ **do**
5: $\quad d_i=0$
6: $\quad$ **for** $j=0,\ldots,k^2-1$ **do**
7: $\quad\quad d_i=d_i+v_j^i$
8: **end parallel for**
9: $n'=n_1+\ldots+n_{k^2}$
10: **for** $l=0,\ldots,n'$ **do**
11: $\quad$ **parallel for** $i=0,\ldots,\sigma-1$ **do**
12: $\quad\quad$ **if** $d_i=l$ **then**
13: $\quad\quad\quad$ **if** $i=0$ **then**
14: $\quad\quad\quad\quad OPT(0,\ldots,0)\leftarrow 0$
15: $\quad\quad\quad\quad$ **break**
16: $\quad\quad\quad \mathcal{O}_{v^i}\leftarrow\emptyset$
17: $\quad\quad\quad \mathcal{C}_{v^i}\leftarrow$ all machine configurations of vector v^i
18: $\quad\quad\quad$ **for all** $(s_1,\ldots,s_{k^2})\in\mathcal{C}_{v^i}$ **do**
19: $\quad\quad\quad\quad \mathcal{O}_{v^i}\leftarrow\mathcal{O}_{v^i}\cup\{OPT(v_1^i-s_1,\ldots,v_{k^2}^i-s_{k^2})\}$
20: $\quad\quad\quad min\leftarrow\infty$
21: $\quad\quad\quad$ **for all** $o\in\mathcal{O}_{v^i}$ **do**
22: $\quad\quad\quad\quad$ **if** $min>o$ **then**
23: $\quad\quad\quad\quad\quad min=o$
24: $\quad\quad\quad OPT(v_1^i,\ldots,v_{k^2}^i)\leftarrow min+1$
25: $\quad$ **end parallel for**
26: **return** $OPT(n_1,\ldots,n_{k^2})$

III. GPU Implementation and Analysis

Since the *DP* procedure is the most expensive component of the PTAS in terms of running time, the parallelization of the *DP* algorithm becomes the major component of our GPU implementation. Because the OpenMP implementation of the PTAS [1] provides only a simple one-level parallelism, a straightforward port to the GPU cannot generate enough threads to fully utilize the GPU computing resources. Moreover, the imbalanced workload among the subproblems of the higher-dimensional *DP* procedure is also more noticeable on the GPU. In our experiments, a direct GPU translation of the OpenMP implementation is about a hundred times slower than the OpenMP implementation. Thus, sophisticated designs using customized techniques are necessary for achieving good performance on the GPU. Our GPU implementation of the PTAS and the higher-dimensional *DP* procedure are illustrated in Algorithms 3, 4, and 5.

In the implementation of PTAS, presented in Algorithm 3, the $[LB,UB]$ interval is equally divided into four independent segments. The bisection search and the *DP* procedure is executed concurrently on each of these segments. Algorithm 4 presents the procedure for partitioning the higher-dimensional *DP-table* and the memory restructuring. Algorithm 5 shows

Algorithm 3 GPU implementation of the PTAS

1: **Input:** $n, m, \mathcal{T} = \{t_1, ..., t_n\}, \epsilon, proc = 4, dim \in \{3, ..., 9\}$
2: $LB_p \leftarrow \frac{p}{proc} \max\left\{\left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil, \max_{j=1,...,n} t_j\right\}, p = 0, ..., 3$
3: $UB_p \leftarrow LB_{p+1}, p = 0, ..., 2$
4: $UB_{proc-1} \leftarrow \left\lceil \frac{1}{m}\sum_{j=1}^{n} t_j \right\rceil + \max_{j=1,...,n} t_j$
5: $k = \lceil 1/\epsilon \rceil,\ LB = LB_0,\ UB = UB_{proc-1},\ count = 0$
6: **while** $LB < UB$ **do**
7: **for** $p = 0, ..., proc - 1$ **do**
8: $T_p = \lfloor (UB_p + LB_p)/2 \rfloor$
9: Partition jobs into short and long jobs
10: Round down long jobs to nearest multiples of $\lfloor T/k^2 \rfloor$
11: Create a k^2-dimensional vector $N = (n_1, ..., n_{k^2})$
12: $OPT_p = Partition(N, T_p, dim, p)$
13: **for** $i = 0, ..., proc - 1$ **do**
14: **if** $OPT_0 \leq m$ **then**
15: $UB = T_0$
16: $LB = LB$
17: $OPT = OPT_0$
18: **else if** $OPT_{proc-1} > m$ **then**
19: $UB = UB$
20: $LB = T_{proc-1}$
21: $OPT = OPT_{proc-1}$
22: **else if** $OPT_i > m\ and\ OPT_{i+1} \leq m$ **then**
23: $UB = T_{i+1}$
24: $LB = T_i$
25: $OPT = OPT_{i+1}$
26: $OPT\_Array[count] = OPT$
27: $count \leftarrow count + 1$

the implementation of the higher-dimensional *DP* procedure and the work distribution which is designed for achieving the maximum execution concurrency.

In the rest of the paper, we will use the term *configuration* to refer to a subproblem of the higher-dimensional dynamic programming.

A. Quarter Split for GPU PTAS

The GPU PTAS is designed similarly to Algorithm 1 and differentiated by the distinct execution intervals. As stated in Section II, the original PTAS algorithm finds the best makespan by performing a bisection search procedure on the interval between the lower bound and the upper bound. Thus, it takes $\mathcal{O}(\log_2 N)$ steps to identify the best makespan, where N is the size of the range between the initial lower and upper bounds.

In the GPU PTAS, we split the range between the lower bound and the upper bound into four equal segments and launch four processes $(p_0, ..., p_3)$ to execute these segments. The use of multiple GPU processes is presented in Algorithm 3, line 7. Since the Hyper-Q feature enables multiple work queues between the CPU and the GPU, these four processes can run concurrently on the same GPU. With four target makespans (T_0, T_1, T_2, T_3) in the execution of the quarter split, one for each segment, we develop a new method for updating the lower bound and the upper bound. As shown in Algorithm 3 (Lines 13-25), the new interval is determined by the *OPT* of each segment and can fall into five possible sections, which are $[LB_0, T_0]$, $[T_0, T_1]$, $[T_1, T_2]$, $[T_2, T_3]$, and $[T_3, UB_3]$. Thus, the size of the new interval is a quarter or even one-eighth of that of the current interval. Having the GPU work on the four segments concurrently further maximizes the concurrent execution and reduces the steps required for achieving the best makespan. The reduction of the steps for achieving the minimum makespan also saves the time wasted on repeated computations. In the OpenMP PTAS implementation [1], some scheduling configurations appear multiple times in the process of finding the minimum makespan, which implies repeated calculations on the same configuration. The execution time wasted on the repeated calculations can be noticeable when the configurations are large.

The quarter split achieves good speedup beyond the original implementation by increasing the execution concurrency up to four times; however, it is still not enough to use all the GPU cores efficiently. To address this issue, we implement the higher-dimensional *DP* procedure with fine-grained parallelism. The fine-grained parallelism is achieved by assigning four concurrent streams to each segment. This setting leads to a utilization of sixteen streams in total, which is large enough to fully occupy the GPU computing resources under most circumstances.

B. Higher-Dimensional Dynamic Programming

The higher-dimensionality of dynamic programming poses significant challenges to a GPU implementation and restricts the GPU performance due to two major issues. First, the dependencies among configurations are more complicated than those in the case of two-dimensional dynamic programming algorithms. In a two-dimensional dynamic programming table, a configuration only depends on the sub-configurations corresponding to three directions, horizontal, vertical, and diagonal. When this is applied to n-dimensional tables, a configuration can be updated from the sub-configurations that correspond to $n(n+1)/2$ directions. Thus, a configuration has many more potential sub-configurations, which are stored dispersedly in the higher-dimensional memory structure. The scattered memory access, called strided access, leads to low effective bandwidth (bus) utilization. The worst case happens when only one thread in the warp reads the requested data from each cache line. Thus, the warp reads data from the memory in a sequential manner, which can lead to significant overhead when the warp fetches data from the global memory, which is more likely to happen with large problem instances. Second, the configurations in the same anti-diagonal level may have different numbers of sub-configurations because of the various dimensional structures among them. Consider the following example. The 3-dimensional configurations $(1, 2, 1)$ and $(0, 0, 4)$ are in the same anti-diagonal level because the sums of their dimensional sizes are the same, but configuration $(1, 2, 1)$ has eleven sub-configurations, and $(0, 0, 4)$ has only four. Since we use as many threads as possible to achieve the maximum concurrency when scheduling the configurations, in the same anti-diagonal level, to separate threads, the unequal workloads among the threads result in thread-level

workload balancing issues. Furthermore, if the index of a sub-configuration, which represents its position in the memory of the dynamic programming table, is unknown during run time, it is necessary to iterate through the *DP-table* and search for the sub-configurations, shown in Algorithm 2 (Line 18). Since the maximum size of the iteration is the same as the size of the *DP-table*, the search function can be time-consuming and becomes an additional major bottleneck.

To obtain efficient performance on the GPU, we address all the issues discussed above by developing a data-partitioning scheme and applying the scheme to the dynamic programming component of PTAS.

C. Proposed Data-Partitioning Scheme

In the higher-dimensional *DP* problem, it is possible that each subproblem in the *DP-table* requires a large chunk of memory for temporarily holding the data of its dependent subproblems, so that even the execution of a relatively small size *DP* problem can also run out of memory. In this paper, we resolve this issue by dividing the huge *DP-table* into many small blocks and performing executions on a number of blocks concurrently. Thus, we can save the memory usage by allocating memory only to the subproblems of these blocks.

From a geometrical point of view, the partitioning evenly divides a higher-dimensional *DP-table* into multiple small blocks of the same size. The number of small blocks and the size of each block is determined by a vector, which we call the $divisor$. A $divisor$ has the same number of dimensions as the higher-dimensional *DP-table*, and the value on each dimension represents the number of segments that this dimension is divided into. Since the subproblems of the higher-dimensional *DP-table* are stored in row-major order, the subproblems of each small block are stored dispersedly in the array of the *DP-table*. Thus, from the data storage point of view, the partitioning scheme reorganizes the storage order of the array to have the subproblems stored within the small blocks.

Similar to Algorithm 2, the flow of computation moves along the main diagonal, and the subproblems on each anti-diagonal are independent. Thus, the subproblems on the same anti-diagonal can be processed in parallel (as shown in Fig. 1). In a partitioned *DP-table*, we develop the same computation flow and parallelization on the blocks and the subproblems in each block separately. In other words, our implementation first processes the blocks on the same level in parallel and then parallelizes the subproblems on the same in-block anti-diagonal level.

As an example, let us consider a 3-dimensional *DP-table* (M, N, L) which is evenly divided by a $divisor$, (a, b, c), and each small block can be represented by a vector (i, j, k), where $i < a$, $j < b$, and $k < c$. Thus, these blocks can be classified into different block-levels, and the vector indicates the block-level $l = i + j + k$ that a block belongs to. Here, the term "block-level" refers to the blocks that can be executed concurrently, which is similar to the term "anti-diagonal level" for concurrent subproblems. We can also index each small block with a unique value, which is calculated from $i \times b \times c + j \times c + k$, so that these small blocks can be stored in a sequence with the index. In addition, a subproblem, represented by (x, y, z), belongs to the small block (i, j, k) if $x \in \left[\frac{M \times i}{a}, \frac{M \times (i+1)}{a}\right]$, $y \in \left[\frac{N \times j}{b}, \frac{N \times (j+1)}{b}\right]$, and $z \in \left[\frac{L \times k}{c}, \frac{L \times (k+1)}{c}\right]$. Moreover, we can indicate the subproblem's in-block anti-diagonal level ($l = x + y + z$) and calculate the vector (i, j, k) of the block it belongs to, where $i = \lfloor x / \frac{M}{a} \rfloor$, $j = \lfloor y / \frac{N}{b} \rfloor$, and $k = \lfloor z / \frac{L}{c} \rfloor$. All the subproblems (x, y, z) that belong to a small block (i, j, k) are stored consecutively in row-major order.

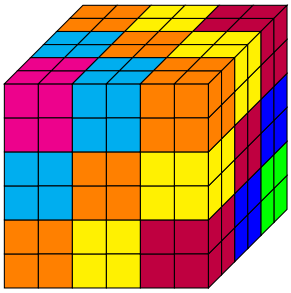


Fig. 2: Partitioning a 3-D *DP-table* by a divisor $(3, 3, 3)$.

Fig. 2 shows an example of the data-partitioning scheme for a 3-dimensional *DP-table*. The table consists of $6 \times 6 \times 6$ subproblems, which are represented by the small cubes. After partitioning the *DP-table* with the $divisor$, all the subproblems are classified into multiple 3-dimensional blocks with a block size of $2 \times 2 \times 2$. Then, these blocks are also grouped into 7 different block-levels which are represented by 7 different colors, and the blocks with the same color are independent and can be executed concurrently. In addition, 8 subproblems, in each block, are also classified into 4 anti-diagonal levels, so that the in-block execution concurrency can also be obtained.

The pseudo-code is illustrated in Algorithm 4. The algorithm divides the *DP-table* into multiple higher-dimensional blocks along a number of specific dimensions, and the number of dimensions is defined by the parameter dim, which is in the range of $(3, ..., 9)$ in our experiments. In Algorithm 4, the *DP-table* is divided along the largest dim dimensions (Line 10), and the number of segments that each dimension is divided into are determined by $divisor$. The entries of the $divisor$ array are computed in lines 4-9, based on the largest configuration, N, of the *DP-table*. To obtain a group of fully functional blocks, the algorithm reorganizes the memory layout of the *DP-table* (Lines 20-27) because the data-partitioning scheme requires the configurations of each block to be stored consecutively. With the newly organized memory layout, the algorithm is able to access the configurations of a block efficiently, which makes the block-level parallelism efficient. The code for classifying the blocks into different block-levels is given in lines 13 and 15. Then, the size of the blocks and the number of configurations of each block are calculated for the purpose of memory access (Lines 18-19). At the end, every four blocks of the same block-level are scheduled into four

Algorithm 4 $Partition(N, T, dim, p)$

Input: $N = (n_1, ..., n_{k^2}), T, dim, p$
$optimal \leftarrow \infty$
$f_1 = 1$, $f_2 = 1$, $block\_offset = 0$, $offset = 0$
$divisor \leftarrow \emptyset$
for $i = 1, ..., k^2$ **do**
  $div = \lfloor \sqrt{n_i + 1} \rfloor$
  while $(n_i + 1) \bmod div \neq 0$ **do**
    $div \leftarrow div - 1$
  $divisor \leftarrow divisor \cup div$
Keep the largest dim dimensions of *divisor*, and set others to 1
Generate the set of configurations $\mathcal{C}$ (*DP-table*) for N
Generate the set $\mathcal{B}$ of all blocks for the set $\mathcal{C}$
for all $(b_1, ..., b_{k^2}) \in \mathcal{B}$ **do**
  $lvl = b_1 + \cdots + b_{k^2}$
  $\mathcal{B}_{lvl} \leftarrow \mathcal{B}_{lvl} \cup \{b_1, ..., b_{k^2}\}$
$\#block\_level \leftarrow$ the number of total block-levels
for $i = 1, ..., k^2$ **do**
  $block\_size[i] = \frac{n_i+1}{divisor[i]}$
  $jobsPerBlock \leftarrow jobsPerBlock \times (block\_size[i] + 1)$
for all $(c_1, ..., c_{k^2}) \in \mathcal{C}$ **do**
  for $i = k^2, ..., 1$ **do**
    $block[i] = \lfloor \frac{c_i}{block\_size[i]} \rfloor$
    $block\_offset \leftarrow block\_offset + block[i] \times f_1$
    $f_1 \leftarrow f_1 \times divisor[i]$
    $offset \leftarrow offset + (c_i - block\_size[i]) \times f_2$
    $f_2 \leftarrow f_2 \times block\_size[i]$
  $\mathrm{M\_offset}_{(c_1,...,c_{k^2})} \leftarrow block\_offset \times jobsPerBlock + offset$
Reorganize $\mathcal{C}$'s memory layout with $\mathrm{M\_offset}_{(c_1,...,c_{k^2})}$
for all $lvl < \#block\_level$ **do**
  for all $(b_1, ..., b_{k^2}) \in \mathcal{B}_{lvl}$ **do**
    $GPU\_DP\langle\langle\langle 1, 1, 0, streams\rangle\rangle\rangle((b_1, ..., b_{k^2}), block\_size)$
$cudaMemcpy(h\_opt, d\_opt, size, DeviceToHost)$
if $optimal > h\_opt$ **then**
  $optimal = h\_opt$
return $optimal + 1$

Algorithm 5 $GPU\_DP((b_1, ..., b_{k^2}),\ block\_offset)$

1: **Input:** $b^i = \{b_1, ..., b_{k^2}\}$, $block\_offset$
2: Let $v^i = (v^i_1, \ldots, v^i_{k^2})$ and $OPT(v^i)$ be the i-th entry of *DP-table*
3: $b^i\_offset \leftarrow block\_offset$
4: $\#(AntiDiag\_lvl) \leftarrow (block\_size[1] + ... + block\_size[k^2] + 1)$
5: **for** $lvl = 1, ..., \#(AntiDiag\_lvl)$ **do**
6: $sizeof(lvl) \leftarrow$ number of configurations at each lvl
7: $FindOPT\langle\langle\langle gridSize, \frac{sizeof(lvl)}{gridSize}\rangle\rangle\rangle(b^i\_offset)$
8: $b^i\_offset \leftarrow b^i\_offset + sizeof(lvl)$
9: $cudaDeviceSynchronize()$
10:
11: $FindOPT(b^i\_offset)$:
12: $tid = \mathrm{blockDim.x} \times \mathrm{blockIdx.x} + \mathrm{threadIdx.x}$
13: $\#(v^{tid}\_subconfig) = 1$
14: **for** $j = 0, ..., k^2 - 1$ **do**
15: ${v^{tid}}_j \leftarrow$ the value at address $b^j\_offset + tid \times k^2 + j$
16: $\#(v^{tid}\_subconfig) \leftarrow \#(v^{tid}\_subconfig) \times ({v^{tid}}_j + 1)$
17: $\mathcal{C}_{v^{tid}} \leftarrow$ all sub-configurations of $v^{tid} = ({v^{tid}}_1, ..., {v^{tid}}_{k^2})$
18: //Get valid multisets $\mathcal{O}_{v^{tid}}$ from kernel *FindValidSub*
19: *FindValidSub*$\langle\langle\langle 1, \#(v^{tid}\_subconfig)\rangle\rangle\rangle(v^{tid}, \mathcal{C}_{v^{tid}})$
20: //update *OPT* of the configuration v^{tid} from its subsets' *OPT*
21: *SetOPT*$\langle\langle\langle 1, sizeof(\mathcal{O}_{v^{tid}})\rangle\rangle\rangle(\mathcal{O}_{v^{tid}}, ({v^{tid}}_1, ..., {v^{tid}}_{k^2}))$
22:
23: *SetOPT*$(\mathcal{O}_v, (v_1, ..., v_{k^2}))$:
24: $tid = \mathrm{blockDim.x} \times \mathrm{blockIdx.x} + \mathrm{threadIdx.x}$
25: Locate the block b^i of vector $\mathcal{O}_v[tid]$
26: **for all** $(c_1, ..., c_{k^2})$ in b^i **do**
27: **if** $(\mathcal{O}_v[tid]_1, ..., \mathcal{O}_v[tid]_{k^2}) == (c_1, ..., c_{k^2})$ **then**
28: $OPT[\mathcal{O}_v[tid]_1, ..., \mathcal{O}_v[tid]_{k^2}] = OPT[(c_1, ..., c_{k^2})]$
29: $min \leftarrow \infty$
30: **for all** $(ns_1, ..., ns_{k^2}) \in \mathcal{O}_v$ **do**
31: **if** $min > OPT(ns_1, ..., ns_{k^2})$ **then**
32: $min \leftarrow OPT(ns_1, ..., ns_{k^2})$
33: $OPT(v_1, ..., v_{k^2}) = min$

streams separately and executed concurrently.

D. Two-level Fine-grained Parallelism

In Algorithm 2, only the subproblems on the anti-diagonal can be processed in parallel. All other iterations for finding the dependent subproblems of a designated subproblem are executed sequentially. A subproblem, in the dynamic programming table of PTAS, consists of a higher-dimensional vector representing a machine configuration.

In the dynamic programming procedure of PTAS, each configuration has a group of sub-configurations from which the job scheduling can be obtained. Thus, all the configurations of the same anti-diagonal level, and all the sub-configurations of the same configuration can be organized into a parent-child structure. Both "parents" and "children" can be distributed across multiple GPU blocks, which leads to more concurrent execution and results in more fine-grained parallelism. With the benefits of using the GPU feature, dynamic parallelism [3], the parent-child structure can be realized using nested two-level fine-grained parallelism. The two-level nested parallelism is presented in Algorithm 5. Line 5 is the iteration that loops through all anti-diagonal levels of the higher-dimensional block. The kernel function *FindOPT* in line 7 is the "parent" at the first fine-grained level, which is called at every anti-diagonal level and maps all the configurations in the same anti-diagonal level to the GPU threads separately. In the kernel function *FindOPT*, each thread launches two other kernel functions, *FindValidSub* and *SetOPT* (Lines 19, 21). These two kernel functions are the "children" at the second fine-grained level. *FindValidSub* helps finding the valid sub-configurations of the configuration, distributed to the "parent" thread, from all possible options (Line 19). Then the *OPT* of every valid sub-configuration is discovered from the dynamic programming table in function *SetOPT* (Lines 26-28), and the sub-configuration with the minimum *OPT* is used to update the *OPT* of the configuration (Lines 30-32).

E. Analysis of the Data Partitioning Scheme

In a fine-grained parallel dynamic programming implementation, especially when running on the GPU, many cores may stay idle for a considerable amount of time, as many anti-diagonal levels do not have enough work to fully occupy all computing resources. This is inevitable in fine-grained parallelism, but our data-partitioning scheme can alleviate the concurrency loss by improving the bus utilization for each

warp when there are free computing resources available. For example, when no data-partitioning scheme is used, an anti-diagonal level of 32 configurations is scheduled to a warp, and thus, each thread in a warp executes one configuration. With our data-partitioning implementation, the anti-diagonal level is divided into b blocks, and each block assigns a warp to some of the configurations. Thus, each warp has $32/b$ active threads on average. If q memory accesses are required when no data-partitioning scheme is employed, the number of the required memory accesses of each warp of the data-partitioning implementation can be reduced to q/b.

Our data-partitioning scheme also addresses the thread-level workload imbalance issue. In the same example, instead of synchronizing all 32 threads of the same warp at the end of the anti-diagonal level, a synchronization, shown in line 9 in Algorithm 5, of only $32/b$ threads is required by each block. Because the blocks of the same block-level are independent when they are executed concurrently, the in-block synchronization has no effect on other blocks, which implies the warps that have less work finish earlier than running them together in one warp. In addition, we can proceed with the execution of the configurations that are in the next anti-diagonal level but in the same block, without creating a race condition with the other blocks. This may improve the block-level workload balance because the overhead of executing the heavy workload configurations at one anti-diagonal level can be amortized by executing the less workload configurations which are in its following levels but the same block, and vice versa. Therefore, the overhead of the warp-level synchronizations can be reduced. If all anti-diagonal levels that have heavy workload are in the same block, other blocks have to wait the completion of this block. In this case, the overhead of synchronizing these blocks is the same as the overhead of synchronizing the corresponding anti-diagonal levels when no data-partitioning scheme is used. However, using more warps also reduces the maximum throughput because many threads are forcibly scheduled but have no work. Thus, the improvement on the effective bus utilization and the workload balance can only be obtained when there are idle cores available.

In addition, our data-partitioning scheme also improves the performance of the search functions by iterating through a block (lines 26-28 in Algorithm 5), instead of the entire *DP-table* as in the case of the OpenMP implementation (lines 18-19 in Algorithm 2). Since the searching function is designed to locate the memory position of a sub-configuration, the data-partitioning scheme reduces the scope from the entire *DP-table* to the block in which the sub-configuration is stored. Beyond that, unlike the case of no data-partitioning in which the entire *DP-table* is stored in the global memory, our proposed data-partitioning scheme works at the scope of blocks. In addition, the reduction of the working space also reduces the sizes of some data arrays which were designed at table scope. The OpenMP implementation uses an array to store the sub-configurations of every configuration created on the basis of the size of the whole *DP-table*. Instead, we store only the required number of blocks in our proposed scheme. The reduction of the size of the data arrays improves the utilization of the limited GPU global memory.

The use of a data-partitioning scheme also has side-effects. First, it reduces the maximum parallelism, which compromises the performance for large problem instances. However, this overhead is much less than the time saved from the search function, so that the scheme is still an efficient solution to this approximation algorithm. Second, to access the correct memory address of a block, the data-partitioning scheme requires some calculations for the memory offset before the block execution is launched. Therefore, the blocks of the same block-level cannot be scheduled concurrently in one kernel call. Instead, these blocks are launched separately in different kernels (Lines 29-31 in Algorithm 4), and these kernels are scheduled sequentially in the default stream. To obtain block-level concurrency, we distribute the blocks of the same block-level, displayed in Fig. 2 as blocks of the same color, into 4 streams in a cyclic distribution manner (Line 31). Because a CUDA stream has its own computing context, these blocks, in the different streams, can be executed concurrently. In our experiments, applying four streams to each data set provides the best performance for the majority of problem instances.

IV. Experimental Results

In this section, we investigate the performance of the proposed techniques by performing extensive experiments with the PTAS algorithm and running it on both the multicore CPU and the many-core GPU. We compare the performance of our proposed GPU algorithm, in terms of execution time, against the performance of the OpenMP algorithm. The comparisons are performed on instances of $P \mid\mid C_{max}$, classified into multiple groups based on their *DP-table* sizes. Since the performance of the sequential PTAS was already compared against the OpenMP implementation in [1], we do not include it in our analysis.

A. Experimental Setup

The experiments with the OpenMP implementations are performed on a dual processor system equipped with two Intel Xeon $E5-2697v3$. The processor has 14 cores and a clock rate of 2.6 GHz for each node. The GPU implementation is evaluated on an Nvidia K40, which has 12 GB memory, 2880 cores and a clock rate of 745 MHz for each core. The performance of the OpenMP implementation is presented for two configurations, 16 cores and 28 cores. The performance data of the GPU implementation is organized by the number of dimensions that the data-partitioning scheme is applied to. We run the experiments, partitioning between 3 and 9 dimensions separately, on the same instances. In the rest of the paper, we use GPU-DIM3 to GPU-DIM9, to denote the cases corresponding to these different partitions. The problem instances are generated using the uniform distribution and considering different numbers of jobs and machines.

According to the approximation algorithm, the maximum number of the *DP-table*'s dimensions is determined by the error rate. In our experiments, we set ϵ to 0.3 as done in [1]

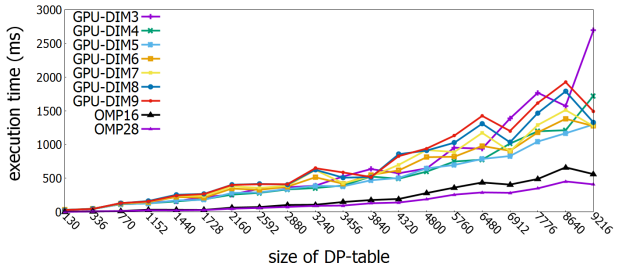
resulting in a table with at most 16 dimensions; however, the number of non-zero dimensions is unknown before the execution because it is determined not only by the jobs' processing times, but also by the target makespan value T. Since each interval $[LB, UB]$ has its unique T in one instance, we can get multiple *DP-tables* of different sizes from each instance during the execution, and the running time of the instance is the addition of the running time of each *DP* execution. To have a better understanding on the higher-dimensional *DP*, we eliminate the speedup obtained on the quarter split technique by evaluating the performance for the different *DP-tables* instead of the scheduling instances. As the sizes of many *DP-tables* are close, we present the typical sizes to shrink the data set for the purpose of making readable figures and tables.

B. Analysis of Results

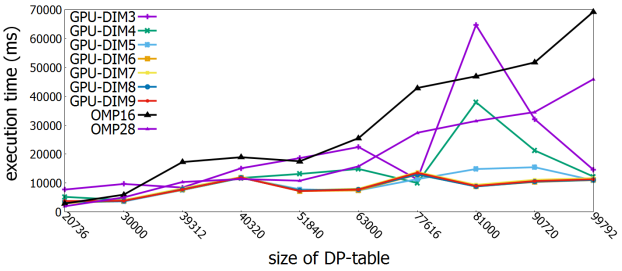
We first analyze the running time of the GPU implementation accelerated by the proposed techniques, by comparing it to the OpenMP implementation. The average running time for each considered size of the DP-table is shown in Fig. 3. We select 36 dynamic programming tables of differing sizes from our much larger data set. These dynamic programming tables are specifically selected for displaying the efficiency of the GPU implementation across the range of sizes in the plots of Fig. 3. The 36 table sizes are divided into three groups, and the ranges are $(100, ..., 10000)$, $(20000, ..., 100000)$, and $(110000, ..., 500000)$. To improve accuracy, we run the same experiment five times and collect all the performance of the selected table sizes, showing the averages of these five runs in the plots.

In the Fig. 3(a), the OpenMP code (denoted by OMP16 and OMP28) performs much better than the GPU code, because the small instances have much less concurrency and get few benefits from the reduction of the search function. Besides, the execution time of the GPU code is dependent on the number of non-zero dimensions in the *DP-table*. When one instance has a small number of non-zero dimensions, dividing along a large number of dimensions cannot obtain further speedup. Consider the instance of table size of 3840 as an example. This instance has 6 non-zero dimensions, which leads to similar performance to the GPU executions that divide along 6 to 9 dimensions. Conversely, in most cases, partitioning along a small number of dimensions cannot obtain good efficiency on the instances that have a large number of non-zero dimensions. Thus, it is not a surprise that the worst performance is obtained in the case of GPU-DIM3 because of the less execution concurrency achieved by this implementation. A similar phenomenon occurs in all the instances, and we can conclude that the trade-off between the block complexity and the in-block workload determines the performance of the GPU implementation.

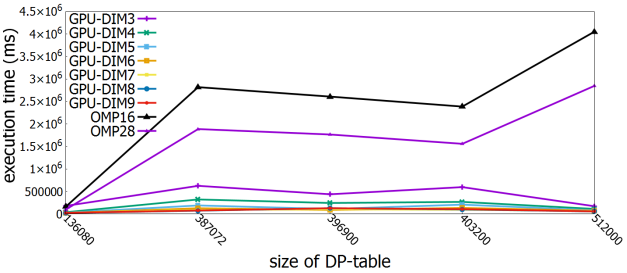
The GPU implementations are more efficient than the OpenMP implementations when the size of the instance's DP-table is larger than 30000. As illustrated in Fig. 3(b) and 3(c), the best GPU performance is obtained by the implementations GPU-DIM6 and GPU-DIM9. Compared to the plots in



(a) Instances with *DP-table* size 100 to 10000.



(b) Instances with *DP-table* size 20000 to 100000.



(c) Instances with *DP-table* size 100000 to 500000.

Fig. 3: Average running time vs. the size of *DP-table*.

Fig. 3(a) and (b), the lines of the plot in Fig. 3(c) are more regular and stable because the size of the instances in the third group are large enough to occupy all the GPU computing resources through the entire execution.

It is also possible that multiple instances share the same *DP-table* size but have a different number of non-zero dimensions. In this case, the same partitioning settings may perform differently on the instances with the same table size. Since the size of the *DP-table* as well as the number of non-zero dimensions of an instance are unknown before the execution, selecting the appropriate instances that can result in an expected table size and different number of non-zero dimensions is impossible. Therefore, due to space limitations, we filter the instances carefully from our data set and select two table sizes from each groups, used in Fig. 3.

Fig. 4 illustrates the effects of different numbers of non-zero dimensions on the GPU performance. We separate the *DP-tables* of the same size according to the number of dimensions *DP-table* has, represented by the values given in the legend. The values along the horizontal axis are the number of dimensions that the *DP-table* is partitioned into.

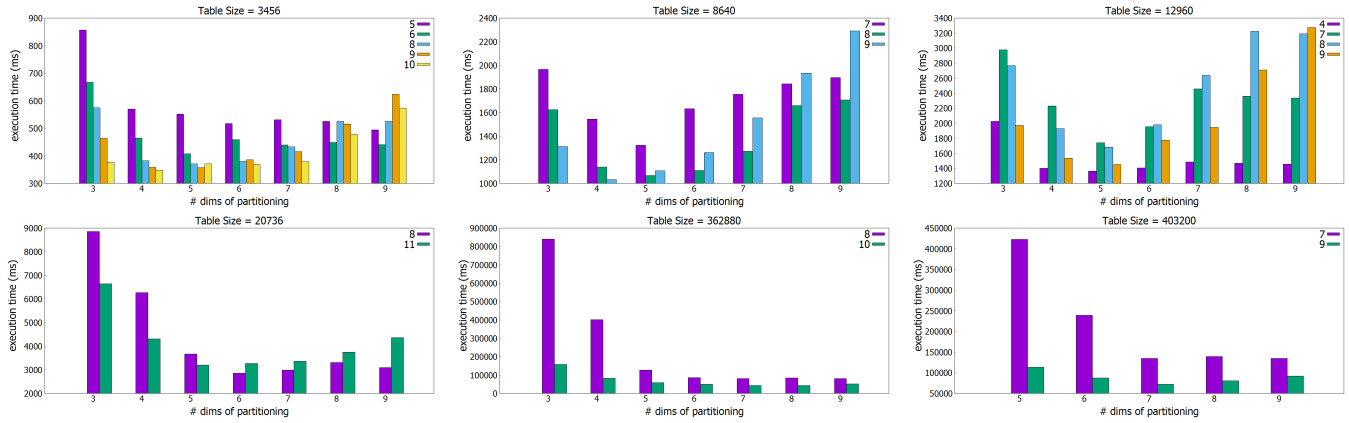


Fig. 4: The number of non-zero dimensions influence the performance.

TABLE I: DP-table Size $= 3456$

#dim	dimension size	GPU-DIM3	GPU-DIM5
5	$(6, 4, 6, 6, 4)$	$(3, 4, 3, 3, 4)$	$(3, 2, 3, 3, 2)$
6	$(2, 6, 3, 4, 6, 4)$	$(2, 3, 3, 2, 3, 4)$	$(2, 3, 1, 2, 3, 2)$
8	$(2, 2, 4, 3, 2, 6, 3, 2)$	$(2, 2, 2, 1, 2, 3, 3, 2)$	$(1, 2, 2, 1, 1, 3, 1, 1)$
9	$(3, 2, 3, 2, 2, 2, 2, 3, 4)$	$(1, 2, 1, 2, 2, 2, 2, 3, 2)$	$(1, 1, 1, 2, 2, 2, 2, 1, 2)$
10	$(2, 3, 2, 2, 3, 3, 2, 2, 2, 2)$	$(2, 1, 2, 2, 1, 1, 2, 2, 2, 2)$	$(2, 1, 1, 1, 1, 1, 2, 2, 2, 2)$

TABLE II: DP-table Size $= 8640$

#dim	dimension size	GPU-DIM3	GPU-DIM5
7	$(5, 3, 6, 3, 4, 4, 2)$	$(1, 3, 3, 3, 2, 4, 2)$	$(1, 1, 3, 3, 2, 2, 2)$
8	$(5, 6, 2, 3, 2, 2, 4, 3)$	$(1, 3, 2, 3, 2, 2, 2, 3)$	$(1, 3, 2, 1, 2, 2, 2, 1)$
9	$(3, 3, 4, 3, 2, 2, 5, 2, 2)$	$(1, 3, 2, 3, 2, 2, 1, 2, 2)$	$(1, 1, 2, 1, 2, 2, 1, 2, 2)$

TABLE III: DP-table Size $= 12960$

#dim	dimension size	GPU-DIM3	GPU-DIM5
4	$(3, 16, 15, 18)$	$(3, 4, 5, 6)$	$(1, 4, 5, 6)$
7	$(4, 5, 3, 6, 4, 3, 3)$	$(2, 1, 3, 3, 4, 3, 3)$	$(2, 1, 1, 3, 2, 3, 3)$
8	$(3, 4, 3, 4, 3, 5, 3, 2)$	$(3, 2, 3, 2, 3, 1, 3, 2)$	$(1, 2, 1, 2, 3, 1, 3, 2)$
9	$(3, 3, 3, 2, 3, 4, 2, 5, 2)$	$(1, 3, 3, 2, 3, 2, 2, 1, 2)$	$(1, 1, 1, 2, 3, 2, 2, 1, 2)$

TABLE IV: DP-table Size $= 20736$

#dim	dimension size	GPU-DIM3	GPU-DIM6
8	$(4, 4, 6, 6, 2, 3, 3, 2)$	$(2, 4, 3, 3, 2, 3, 3, 1)$	$(2, 1, 2, 2, 1, 1, 1, 1)$
11	$(2, 4, 2, 3, 3, 3, 3, 2, 2, 2, 2)$	$(2, 2, 2, 1, 1, 3, 3, 2, 2, 2, 2)$	$(1, 2, 2, 1, 1, 1, 1, 2, 2, 2, 2)$

TABLE V: DP-table Size $= 362880$

#dim	dimension size	GPU-DIM3	GPU-DIM7
8	$(5, 6, 3, 7, 6, 4, 8, 3)$	$(5, 3, 3, 1, 5, 4, 4, 3)$	$(1, 3, 1, 1, 3, 2, 4, 3)$
10	$(3, 3, 3, 4, 5, 7, 2, 3, 4, 4)$	$(3, 3, 3, 2, 1, 1, 2, 3, 4, 4)$	$(3, 3, 1, 2, 1, 1, 2, 1, 2, 2)$

TABLE VI: DP-table Size $= 403200$

#dim	dimension size	GPU-DIM3	GPU-DIM7
7	$(3, 10, 7, 6, 4, 8, 10)$	$(3, 5, 7, 6, 4, 4, 5)$	$(1, 5, 1, 3, 2, 4, 5)$
9	$(4, 5, 4, 2, 3, 5, 7, 3, 8)$	$(4, 1, 4, 2, 3, 5, 1, 3, 4)$	$(2, 1, 2, 2, 1, 1, 1, 3, 4)$

The performance data show that the best GPU performance of these 6 selected table sizes is obtained separately when partitioning the table along 5, 6, and 7 dimensions, which is similar to the results presented in Fig. 3. The execution of the table size of 403200 is too slow when it is partitioned into 3 or 4 dimensions, and the running times exceed the wall clock time, which is set to $10,800,000$ milliseconds. Moreover, the *DP-tables* that have fewer dimensions are usually less efficient than the other *DP-tables* of the same table size having more dimensions. However, exceptions still exist. Thus, we proceed with an in-depth analysis of the size of the in-block dimensions, which appears to be the major factor that can significantly affect the performance.

To better understand the performance results, we need to investigate the effect of the dimensional sizes of the blocks. Again, we use the 6 selected table sizes from the example data set and compare the block dimensional sizes of different partition settings for each *DP-table*. In Table I, we compare the block's dimensional sizes of GPU-DIM3 to the block's dimensional sizes of GPU-DIM5 for each of the *DP-tables* with different non-zero dimensions. Relating the differences to the performance, we can discover how a block's dimensional sizes influence the performance. The size of each block dimension is calculated according to the division, presented in Algorithm 4 (Lines 4-9). The related data is presented in the Tables I-VI. The four columns in each table represent the number of non-zero dimensions, the dimensional sizes of the *DP-table*, the dimensional sizes of the block after partitioning the *DP-table* along three dimensions, and the dimension sizes of the block after partitioning the *DP-table* along a specific number of dimensions, from which the best performance is obtained.

We can conclude from Tables I-VI and Fig. 4 that the best performance is usually obtained by the execution that has the most regular shaped blocks and the smallest in-block workload. Generally, a large number of non-zero dimensions is helpful to the block's regularity because the high-density dimensions can be scattered by the extra dimensions. In Table I, even if the *DP-tables* of 5 or 6 non-zero dimensions have the same table size, and the executions of these two *DP-tables* have the same number of launched GPU blocks, we can still observe from Fig. 4 that the execution of 6 non-zero dimensions is more efficient because the one additional non-zero dimension further improves the block regularity, as it is shown in column GPU-DIM3. In addition, the block's dimensional sizes of GPU-DIM5 has more regular shapes and less workload than GPU-DIM3 for all the *DP-tables*, and we can see from the Fig. 4 that the performance of GPU-DIM5 is

TABLE VII: Runtime and number of iterations performed.

table size	#itr GPU	runtime GPU	#itr OpenMP	runtime OpenMP
12960	8	13,183	13	11,160
20736	4	13,031	6	13,072
27360	1	4,559	3	15,238
30240	3	11,139	5	34,098
403200	3	300,881	5	9,654,220

better on all the *DP-tables* of different non-zero dimensions. This conclusion also applies to other *DP-table* sizes.

We now present the achievement of the quarter split technique. We run the experiments on some selected instances and use the configurations, executed multiple times, as our comparison objects. In Table VII, a designated configuration is represented by its unique *DP-table* size as shown in column "table size". We count the number of iterations for getting the best makespan value and the total running time for each designated configuration. The number of the iterations and the running time are titled as "#itr" and "runtime" in Table VII and applied to both the GPU implementation and the OpenMP implementation. For the configurations of smaller *DP-table* size, like 12960 and 20736, the GPU runtime are close or even longer than the OpenMP runtime, even if the OpenMP implementation runs more iterations to find the best makespan. It is because of either the low GPU concurrency or the limited benefit that GPU can obtain from partitioning a small data block. As expected, the speedup becomes more considerable on the large table size subproblems, as shown in the rows corresponding to table sizes 27360, 30240, and 403200.

V. Conclusion

We proposed a customized parallel approximation algorithm which is the first parallel approximation algorithm for solving $P \mid\mid C_{max}$ problem on the GPU. Our work indicates that GPU parallel programming has a great potential for accelerating approximation algorithms. Besides, we proposed a data-partitioning scheme for accelerating the higher-dimensional dynamic programming implementations on the GPU. The proposed technique improves the GPU performance significantly and makes the GPU implementation perform better than the OpenMP implementation on large-scale higher-dimensional dynamic programming problems. To our knowledge, this is the first data-partitioning scheme specifically designed for addressing the performance and memory issue of higher-dimensional dynamic programming on the GPU.

In our future work, we plan to apply the proposed data-partitioning scheme to other higher-dimensional dynamic programming problems, like higher-dimensional knapsack problems, and eventually extend it to a general technique that can be used to accelerate similar problems. In addition, it is possible that the data-partitioning scheme can further reduce the memory usage. Since the values of the subproblems are required on the GPU for finding the optimal result, the values of the entire *DP-table* are now stored on the GPU. If the blocks that include the required subproblems can be located, only the values of the subproblems in these blocks are needed on the GPU.

Acknowledgments

We thank the Advanced Computing team of Wayne State University for supporting our research with the high performance clusters available on the WSU grid.

References

[1] L. Ghalami and D. Grosu, "A parallel approximation algorithm for scheduling parallel identical machines," in *Proc. IEEE Intl. Parallel and Distributed Processing Symp. Workshops*, 2017, pp. 442–451.

[2] Nvidia Corporation. (2012) NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK110. [Online]. Available: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf

[3] Y. Li, L. Schwiebert, E. Hailat, J. Mick, and J. Potoff, "Improving performance of gpu code using novel features of the nvidia kepler architecture," *Concurrency and Computation: Practice and Experience*, vol. 28, no. 13, pp. 3586–3605, 2016.

[4] C. E. Alves, E. N. Cáceres, F. Dehne, and S. W. Song, "A parallel wavefront algorithm for efficient biological sequence comparison," in *Proc. Intl. Conf. on Computational Science and Its Applications*. Springer, 2003, pp. 249–258.

[5] M. Low, W. Liu, and B. Schmidt, "A parallel bsp algorithm for irregular dynamic programming," *Advanced Parallel Processing Technologies*, pp. 151–160, 2007.

[6] C. E. Alves, E. N. Cáceres, and F. Dehne, "Parallel dynamic programming for solving the string editing problem on a cgm/bsp," in *Proc. 4-th Annual ACM Symp. on Parallel Algorithms and Architectures*, 2002, pp. 275–281.

[7] B. Schmidt, H. Schröder, and M. Schimmler, "Massively parallel solutions for molecular sequence analysis." in *Proc. 16th IEEE Intl. Parallel and Distributed Processing Symp.*, 2002.

[8] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the gpu," in *Proc. 2nd Intl. Conf. on Networking and Computing*, 2011, pp. 320–326.

[9] C.-C. Wu, J.-Y. Ke, H. Lin, and W.-c. Feng, "Optimizing dynamic programming on graphics processing units via adaptive thread-level parallelism," in *Proc. 17th IEEE Intl. Conf. on Parallel and Distributed Systems*, 2011, pp. 96–103.

[10] K. Nishida, K. Nakano, and Y. Ito, "Accelerating the dynamic programming for the optimal polygon triangulation on the gpu," in *Proc. Intl. Conf. on Algorithms and Architectures for Parallel Processing*. Springer, 2012, pp. 1–15.

[11] V. Boyer, D. El Baz, and M. Elkihel, "Solving knapsack problems on gpu," *Computers & Operations Research*, vol. 39, no. 1, pp. 42–47, 2012.

[12] D. A. Castanon, "Approximate dynamic programming for sensor management," in *Proc. 36th IEEE Conf. on Decision and Control*, vol. 2, 1997, pp. 1202–1207.

[13] D. Bertsimas and R. Demir, "An approximate dynamic programming approach to multidimensional knapsack problems," *Management Science*, vol. 48, no. 4, pp. 550–565, 2002.

[14] V. Boyer, D. El Baz, and M. Elkihel, "Solution of multidimensional knapsack problems via cooperation of dynamic programming and branch and bound," *European Journal of Industrial Engineering*, vol. 4, no. 4, pp. 434–449, 2010.

[15] K.-E. Berger and F. Galea, "An efficient parallelization strategy for dynamic programming on gpu," in *Proc. 27th IEEE Intl. Parallel and Distributed Processing Symp. Workshops & PhD Forum*, 2013, pp. 1797–1806.

[16] D. S. Hochbaum and D. B. Shmoys, "Using dual approximation algorithms for scheduling problems theoretical and practical results," *Journal of the ACM*, vol. 34, no. 1, pp. 144–162, 1987.