Solving Dynamic Programming Problem by Pipeline Implementation on GPU

Susumu Matsumae[1]
Department of Information Science
Graduate School of Science and Engineering
Saga University
Saga, Japan

Makoto Miyazaki[2]
Department of Information Science
Graduate School of Science and Engineering
Saga University
Saga, Japan

Abstract—In this paper, we show the effectiveness of a pipeline implementation of Dynamic Programming (DP) on GPU. As an example, we explain how to solve a matrix-chain multiplication (MCM) problem by DP on GPU. This problem can be sequentially solved in $O(n^3)$ steps by DP where n is the number of matrices, because its solution table is of size $n \times n$ and each element of the table can be computed in $O(n)$ steps. A typical speedup strategy for this is to parallelize the $O(n)$ step computation of each element, which can be easily achieved by parallel prefix computation, i.e., an $O(\log n)$ step computation with n threads in a tournament fashion. By such a standard parallelizing method, we can solve the MCM problem in $O(n^2 \log n)$ steps with n threads. In our approach, we solve the MCM problem on GPU in a pipeline fashion, i.e., we use GPU cores for supporting pipeline-stages so that many elements of the solution table are partially computed in parallel at one time. Our implementation determines one output value per one computational step with n threads in a pipeline fashion and constructs the solution table totally in $O(n^2)$ steps with n threads.

Keywords—*Dynamic programming; pipeline implementation; GPGPU*

I. Introduction

In this paper, we show the effectiveness of a pipeline implementation of Dynamic Programming (DP) on GPU. As an example, we explain how to solve a *matrix-chain multi-plication (MCM)* problem [1] by DP on GPU. This problem can be sequentially solved in $O(n^3)$ steps by DP where n is the number of matrices, because its solution table is of size $n \times n$ and each element of the table can be computed in $O(n)$ steps. A typical speedup strategy for this is to parallelize the $O(n)$ step computation of each element, which can be easily achieved by parallel prefix computation, i.e., an $O(\log n)$ step computation with n threads in a tournament fashion. By such a standard parallelizing method, we can solve the MCM problem in $O(n^2 \log n)$ steps with n threads.

It has been studied well to speed up DP programs using GPU (e.g. [2], [3]), where they mainly focus on optimizing the order of accessing data by proposing novel techniques avoiding memory access conflicts. In this study, we consider adopting a pipeline technique and implementing the DP program on GPU in a pipeline fashion. The pipeline computation technique [4] can be used in situations in which we perform several operations $\{OP_1, OP_2, \ldots, OP_n\}$ in a sequence, where some steps of each OP_{i+1} can be carried out before operation OP_i

is finished. In parallel algorithms, it is often possible to overlap those steps and improve total execution time.

In our approach, we solve the MCM problem on GPU in a pipeline fashion, i.e., we use GPU cores for supporting pipeline-stages so that many elements of the solution table are partially computed in parallel at one time. Our implementation determines one output value per one computational step with n threads in a pipeline fashion and constructs the solution table totally in $O(n^2)$ steps with n threads. This paper is an extended version of our conference paper [5].

The rest of this paper is organized as follows. Section II introduces problem definitions and base algorithms. Section III explains our pipeline implementations for DP on GPU and offers some experimental results. Section IV explains how to apply the pipeline implementation technique to the MCM problem, and finally Section V offers concluding remarks.

II. Preliminaries

In this section, we introduce some preliminary definitions and base algorithms. We first define a simplified DP problem to be solved on GPU, and then explain our GPU implementations of programs.

A. Simplified DP Problem

In this study, we implement a typical DP program on GPU. To simplify the exposition, we focus on programs that solve such a simplified DP problem defined as follows:

Definition 1: (Simplified DP Problem) A one-dimensional array $\mathrm{ST}[0, \ldots, n-1]$ of size n as a solution table, a set $\mathcal{A} = \{a_1, a_2, \ldots, a_k\}$ of k integers representing offset numbers, and a semi-group binary operator $\otimes$ over integers are given. Every element of set $\mathcal{A}$ satisfies the following inequality:

$$a_1 > a_2 > \cdots > a_k > 0.$$

Then, a *simplified DP problem* (S-DP problem) is to fill all the elements of array ST in such a way that each $\mathrm{ST}[i]$ is computed by the following equation:

$$\mathrm{ST}[i] = \otimes_{1 \leq j \leq k} \mathrm{ST}[i - a_j] \qquad (1)$$

where

$\mathrm{ST}[0]$, $\mathrm{ST}[1]$,$\ldots$,$\mathrm{ST}[a_1 - 1]$ are preset with initial values. ∎

For example, Fibonacci number problem is equal to the S-DP problem where $k = 2, a_1 = 2, a_2 = 1, \otimes = +$, and `ST[0]`=`ST[1]`=1.

B. Conventional Approach to S-DP Problem

To begin with, we show a straightforward sequential algorithm that solves the S-DP problem. Fig. 1 shows the algorithm.

A Sequential Algorithm for S-DP Problem

for `i` = a_1 **to** $n - 1$ **do**
   `ST[i]` = `ST[i−`a_1`]`;
   for `j` = 2 **to** k **do**
      `ST[i]` = `ST[i]` $\otimes$ `ST[i−`a_j`]`;

Fig. 1. A sequential algorithm for S-DP problem

The outer loop computes values from `ST[`a_1`]` to `ST[`$n-1$`]` in order, and the inner loop computes `ST[i]` for each `i` by equation (1). Since the outer loop takes $n - a_1 + 1 = O(n)$ steps and the inner loop requires $O(k-1)$ steps, this algorithm takes $O(nk)$ steps in total.

Next, we consider parallelizing the algorithm for S-DP problem. The straightforward approach is to parallelize the inner loop by using GPU cores. We can easily write a multi-thread program that executes the inner loop-body, `ST[i]` = `ST[i]` $\otimes$ `ST[i−`a_j`]`, for each `j` in parallel using $k-1$ threads at one time. Such an implementation, however, does not improve the time cost, because every thread has access to the same `ST[i]` and thus memory access conflicts occur. As a result, those memory conflicts should be automatically solved at run-time by the serializing mechanism of GPU, and consequently the whole time cost stays in $O(nk)$ steps, which is the same time cost as that of the sequential implementation.

To avoid the memory access conflicts, we can use a well-known standard parallel prefix computation algorithm [6], [7], in which the computations of $\otimes$ over k values are executed in a tournament fashion. Since the parallel prefix computation runs in $O(\log k)$ steps for k values, obviously the entire time cost can be improved to $O(n \log k)$ steps by using k threads.

Although we can successfully reduce the time cost from $O(nk)$ to $O(n \log k)$ by using the parallel prefix computation, it is not work-time optimal because there are many idle threads during the computations in a tournament fashion. In the next section we propose other parallel implementation strategy and show that we can improve the time cost further.

III. PIPELINE IMPLEMENTATION ON GPU

In this section, we explain our proposed parallel implementations for S-DP problem on GPU. Our program runs in a pipeline fashion.

A. Pipeline Implementation for S-DP Problem

In our implementation, we use a group of k threads to establish k-stage pipeline, and this thread group treats k consecutive elements at one time in parallel. Fig. 2 describes our pipeline algorithm for the S-DP problem. The index variable `i` of the outer loop stands for the head position of the k-thread group. The inner loop controls each thread's behaviour in such a way that the `j`-th thread executes computation for `ST[i−j+1]` using the value stored in `ST[i−j+1−`a_j`]`.

A Pipeline Algorithm for S-DP Problem

for `i` = a_1 **to** $n + k - 2$ **do**
   for `j` = 1 **to** k **do in parallel**
      Thread `j` executes the following operation if $a_1 \le i_j < n$ where $i_j = $ `i` $- $ `j` $+ 1$:

$$ST[i_j] = \begin{cases} ST[i_j - a_j]; & (j = 1) \\ ST[i_j] \otimes ST[i_j - a_j]; & (j > 1) \end{cases}$$

Fig. 2. A pipeline algorithm for S-DP problem

An execution example is shown in Fig. 3, where $k = 3$, $a_1 = 5$, $a_2 = 3$, and $a_3 = 1$ hold and the initial values are already stored in `ST[0]`, `ST[1]`,..., `ST[4]`. In Step 1, the head position `i` of the thread group is 5. In this step the only one thread is activated and executes `ST[5]` $\leftarrow$ `ST[0]`. In Step 2, the head position is incremented to 6, and two threads are activated. The first thread treats `ST[6]` and the second thread works on `ST[5]`. In Step 3, the head position becomes 7, and now all $k = 3$ threads actively execute operations for `ST[7]`, `ST[6]`, and `ST[5]` respectively. It should be noted that finally in Step 3 the content of `ST[5]` is completely determined while those of `ST[7]` and `ST[6]` are partially computed and not yet determined. From Step 3, all the $k = 3$ threads are active until Step $n - a_1$ when the head position `i` of thread group reaches $n-1$, and after that step the number of active threads decreases one by each step. As you can see there is no memory access conflict in this example.

As for the time-complexity of our pipeline implementation, from a theoretical viewpoint, it takes only $O(n)$ steps, because the outer loop takes $n + k - a_1 - 1 = O(n)$ cycles and the inner loop requires $O(1)$ time if there is no adjacent offset pair (a_m, a_{m+1}) such that $a_m = a_{m+1} + 1$.

However, from a practical viewpoint, because of the memory access conflicts, the inner loop may take more time steps. Actually, in the worst case when consecutive offset numbers are given, those `ST[`$i_j - a_j$`]`, in the right hand side of the assignment statement, coincidentally become the same element of array `ST` and hence the worst memory access conflicts occur. In such a case, all threads in the inner loop are serialized and it takes time proportional to k. See Fig. 4 for such a worst case example. In this example, all four threads try to have access to `ST[i − 4]` at the same time in the inner loop.

Let $seq = (a_p, a_{p+1}, \ldots, a_q)$ be one of the longest subsequences of given offset numbers $(a_1, a_2, \ldots, a_k)$ satisfying $a_r = a_{r+1} + 1$ for all $p \le r < q$. Then, it is easy to check that in the inner loop every thread r $(p \le r < q)$ has access to the

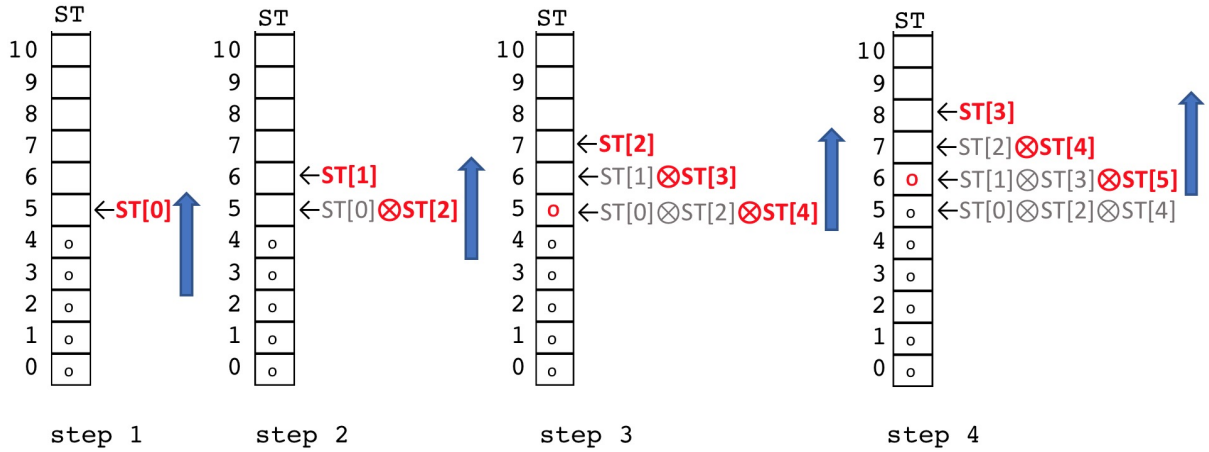Fig. 3. An execution example for the case where $k = 3$, $a_1 = 5$, $a_2 = 3$, and $a_3 = 1$ hold and initial values are preset to ST[0], ST[1], ..., and ST[4].

same element of array ST, and as a result those conflicts are serialized at run time by GPU's serializing mechanism and hence the memory access time becomes $(q - p + 1)$ times slower than that of conflict-free case. For such a case, in [5], we proposed a *2-by-2 pipeline implementation* technique where each thread invoked in the inner loop executes two computations for each element of array ST. The details can be found in [5].
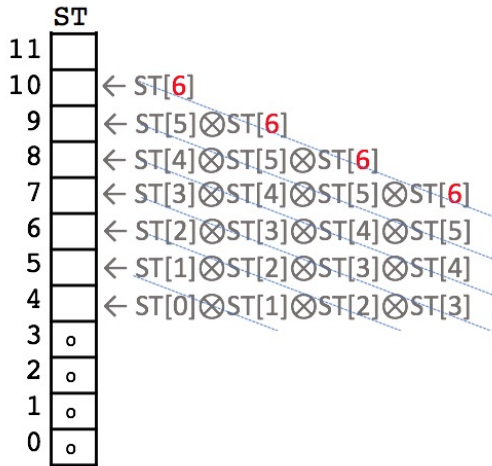

Fig. 4. An example for the worst case where the offset numbers are consecutively given. In this example, we have $k = 4$, $a_1 = 4$, $a_2 = 3$, $a_3 = 2$, and $a_4 = 1$.

B. Experimental Results

Before going to the next section, we show the performance of our pipeline implementation on GPU. We use a computer with 3.40 [GHz] Intel Xeon CPU E3-1245 v3 and an NVIDIA GeForce GTX TITAN Black. The OS is Ubuntu 17.10 and we use g++ 7.2.0 for compiling cpp programs and CUDA 9.2 [8]. The experimental results are shown in TABLE I.

TABLE I shows the average execution time for each implementation on GPU. In the table, SEQUENTIAL, NAIVE-PARALLEL, and PIPELINE respectively stand for the sequential implementation, the naive multi-thread implementation, and our pipeline implementation (for a general case). Here, we use min operation for $\otimes$. The average is computed among 100 executions for each setting.

TABLE I. EXECUTION TIME OF SEQUENTIAL, NAIVE PARALLEL, AND PIPELINE IMPLEMENTATIONS (MSEC)

	SEQUENTIAL	NAIVE-PARALLEL	PIPELINE
$2^{14} \leq n \leq 2^{15}$, $2^{12} \leq k \leq 2^{13}$	274	64	78
$2^{16} \leq n \leq 2^{17}$, $2^{14} \leq k \leq 2^{15}$	4,288	368	386
$2^{18} \leq n \leq 2^{19}$, $2^{16} \leq k \leq 2^{17}$	68,453	3,018	2,408

As for the comparison between the sequential implementation and the parallel ones, parallel implementations are much faster even though it is NAIVE-PARALLEL. Although there is no difference in time between NAIVE-PARALLEL and PIPELINE until $n \leq 2^{17}$, PIPELINE is faster than NAIVE-PARALLEL when $n \geq 2^{18}$.

IV. SOLVING MCM PROBLEM

In this section, we explain how to apply our pipeline implementation technique to more general DP problems. As an example, we deal with the matrix chain multiplication problem (MCM problem) [1].

A. Outline of Pipeline Implementation for MCM Problem

It is well-known that the MCM problem can be efficiently solved by DP with a two-dimensional solution table of triangular shape, and that each element is computed along the diagonal direction. See Fig. 5 for an example. In the figure, each number represents the order of elements to be computed.
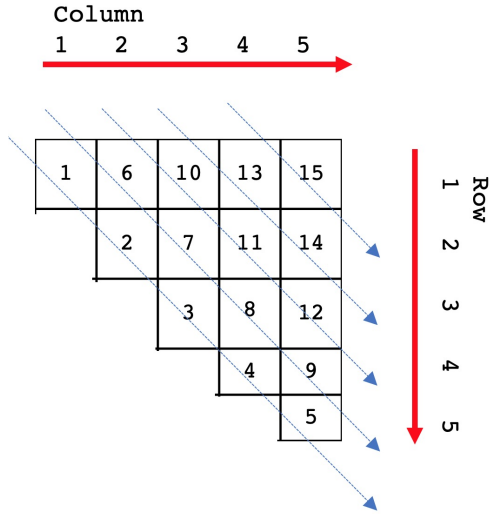
Fig. 5. A solution table example for a matrix chain multiplication problem.

The content of each element is computed by a binary operator $\downarrow$ returning a smaller operand and a binary function $f(*, *)$ as in Fig. 6. The detailed definition of the MCM problem can be found in [1]. In the figure, the element marked 13 is computed by the elements marked by 1, 6, 10, 11, 8, and 4. If we write $\text{ST}[x]$ for the element marked x here, the computation is expressed as

$$\text{ST}[13]$$

$$= f(\text{ST}[1], \text{ST}[11]) \downarrow f(\text{ST}[6], \text{ST}[8]) \downarrow f(\text{ST}[10], \text{ST}[4]).$$
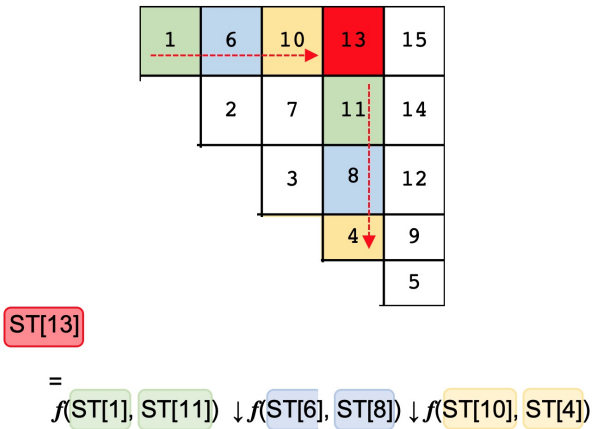

Fig. 6. An example of computing an element of solution table. Here, we write $\text{ST}[x]$ for the element marked x.

Since the elements of two-dimensional solution table are computed in a total order (linear order), we can line up them into a linear array according to that order. Once the solution table is transformed into a linear array, we can apply the pipeline technique to the MCM problem as well. An execution example is shown in Fig. 7.

B. Detailed Pipeline Implementation for MCM Problem

Here, we explain how to solve the MCM problem by pipeline implementation in details.

Firstly, we assume that the original two-dimensional solution table of diagonal shape is already mapped to a one dimensional solution table ST appropriately. That is, we line up all the elements of the two-dimensional table into a linear array ST according to the total (linear) order in which each element is computed along the diagonal direction as in Fig. 5 and 7. Since the number of input matrices is n, the number of elements in the solution table is $n(n+1)/2$. First n elements of array ST are preset with initial values, those correspond to the elements located in the diagonal line of the original two-dimensional table.

Let each computation for $\text{ST}[i]$ be represented as

$$\text{ST}[i] = \downarrow_{1 \le j \le k_i} f(l_{(i,j)}, r_{(i,j)}). \tag{2}$$

It should be noted that here each k_i may differ from others and that it is $n-1$ at largest. See Fig. 5 and 7 for an example. Here, we have $n = 5$, and $\text{ST}[13]$ and $\text{ST}[12]$ can be represented as follows:

$$\text{ST}[13]$$

$$= f(l_{(13,1)}, r_{(13,1)}) \downarrow f(l_{(13,2)}, r_{(13,2)}) \downarrow f(l_{(13,3)}, r_{(13,3)})$$

$$= f(\text{ST}[1], \text{ST}[11]) \downarrow f(\text{ST}[6], \text{ST}[8]) \downarrow f(\text{ST}[10], \text{ST}[4])$$

and

$$\text{ST}[12]$$

$$= f(l_{(12,1)}, r_{(12,1)}) \downarrow f(l_{(12,2)}, r_{(12,2)})$$

$$= f(\text{ST}[3], \text{ST}[9]) \downarrow f(\text{ST}[8], \text{ST}[5]).$$

To solve such defined MCM problem by the pipeline implementation on GPU, we have to modify the pipeline algorithm designed for the S-DP problem, because the offset numbers for each $\text{ST}[i]$ may differ from others. Thus, for the MCM problem, we propose a modified pipeline algorithm in Fig. 8. In the MCM-pipeline algorithm, each element of ST is computed by equation (2), and $\text{ST}[1], \text{ST}[2], \ldots, \text{ST}[n]$ are preset with initial values. In substep 1, 2, and 3, the computation of $f(l_{(i,j)}, r_{(i,j)})$ is executed, and in substep 4, that obtained value is used for the computation by $\downarrow$ and the result is stored to $\text{ST}[i]$.

C. Conflict-Free Memory Access of MCM Algorithm

In this subsection, we prove that no memory access conflict occurs during the execution of the MCM-pipeline algorithm described in Fig. 8.

To begin with, we prove the following lemma.

Lemma 1: In substep 1 of an execution of the inner loop-body of the MCM algorithm, each thread has access to a distinct element of the array ST.

Proof: Assume that threads p and q try to read the same element of ST in substep 1 and that $p < q$ holds. Let (row_p, col_p) (resp. (row_q, col_q)) be the pair of row and column indexes of the elements in the original two-dimensional solution table of

Fig. 7.   An execution example of pipeline implementation for the matrix chain multiplication problem.

A Pipeline Algorithm for MCM Problem

for $i = n + 1$ **to** $n(n+1)/2 + n - 2$ **do**
  for $j = 1$ **to** $(n-1)$ **do in parallel**
    Thread j executes the following operation if $i_j \leq k_{i_j}$ where $i_j = i - j + 1$:
    (substep 1)

$$v_l = l_{(i_j, j)};$$

    (substep 2)

$$v_r = r_{(i_j, j)};$$

    (substep 3)

$$v_s = f(v_l, v_r);$$

    (substep 4)

$$\text{ST}[i_j] = \begin{cases} v_s; & (j = 1) \\ \text{ST}[i_j] \downarrow v_s; & (j > 1) \end{cases}$$

    where $v_l, v_r,$ and v_s are local variables in a thread.

Fig. 8.   A pipeline algorithm for MCM problem

triangle shape of the MCM problem for which thread p (resp. q) is now computing. Since threads p and q try to read the same element of ST and in substep 1 they read the value for the left argument of the function f, the relation $row_p = row_q$ must hold. Then, since threads p and q respectively read the p-th and q-th elements from the left in the same row of the original two-dimensional solution table, the relation $p = q$ must hold if the two threads read the same element of ST, which contradicts the assumption $p < q$. ∎

Next, we prove the following lemma, which can be proved in a similar way to the proof of Lemma 1.

Lemma 2: In substep 2 of an execution of the inner loop-body of the MCM algorithm, each thread has access to a distinct element of the array ST.

Proof: Assume that threads p and q try to read the same element of ST in substep 2 and that $p < q$ holds. Let (row_p, col_p) (resp. (row_q, col_q)) be the pair of row and column indexes of the elements in the original two-dimensional solution table of triangle shape of the MCM problem for which thread p (resp. q) is now computing. Since threads p and q try to read the same element of ST and in substep 2 they read the value for the right argument of the function f, the relation $col_p = col_q$ must hold. On the other hand, thread p reads the p-th elements below of the row row_p of the original two-dimensional solution table, and thread q does the q-th elements below of the row row_q of the table. Since threads p and q read row $(row_p + p)$ and row $(row_q + q)$ respectively, the relation $row_p + p = row_q + q$ must hold if the two threads read the same element of ST. Since $p < q$ and $col_p = col_q$ hold, the relation $row_p < row_q$ must hold from the way of mapping from the original two-dimensional solution table to the linear array ST. This leads to the relation $row_p + p \neq row_q + q$, which contradicts the assumption that threads p and q read the same element of array ST. ∎

In substep 3, each thread simply executes computation using local variables. In substep 4, it is obvious that each thread has access to a distinct element of ST. Therefore, by Lemma 1 and Lemma 2, we obtain the following theorem.

Theorem 1: No memory access conflict occurs during the execution of the MCM-pipeline algorithm described in Fig. 8. ∎

As for the time-complexity of the MCM-pipeline implementation, from a theoretical viewpoint, it takes only $O(n^2)$ steps with $(n-1)$ threads, because the outer loop takes $O(n^2)$ cycles and the inner loop requires $O(1)$ time (Theorem 1).

V. CONCLUDING REMARKS

In this study, we examined the effectiveness of pipeline implementations of Dynamic Programming (DP) on GPU. As an example, we explained how to solve a matrix-chain multiplication (MCM) problem by DP on GPU. This problem can be sequentially solved in $O(n^3)$ steps by DP where n is the number of matrices. In our approach, we solve the MCM problem on GPU in a pipeline fashion, i.e., we use GPU cores for supporting pipeline-stages so that many elements of the solution table are partially computed in parallel at one time. Since our implementation determines one output value per one computational step with $O(n)$ threads, we can solve the MCM problem in $O(n^2)$ steps using $O(n)$ threads, which is an ideal speedup from the $O(n^3)$-step sequential DP algorithm.

For future work, we plan to evaluate the performance of our pipeline implementations. From the experimental results shown in Section III, it is obvious that the ideal speed up is not attained here. This is mainly due to the limitations on the bandwidth of memory on GPU. That is, as the problem size becomes large, all threads cannot always have access to the target memory at one time, because unavoidable access conflicts occur. We also plan to study the relation between the memory bandwidth and the performance of our pipeline implementation on some theoretical GPU models (e.g., [9]).

ACKNOWLEDGMENT

REFERENCES

[1] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[2] Y. Ito and K. Nakano. A gpu implementation of dynamic programming for the optimal polygon triangulation. *IEICE Transactions on Information and Systems*, E96.D(12):2596–2603, 2013.

[3] K. Nakano. A time optimal parallel algorithm for the dynamic programming on the hierarchical memory machine. In *2014 Second International Symposium on Computing and Networking (CANDAR)*, pages 86–95, 2014.

[4] S. H. Roosta. *Parallel Processing and Parallel Algorithms: Theory and Computation*. Springer, 1999.

[5] M. Miyazaki and S. Matsumae. A pipeline implementation for dynamic programming on gpu. In *International Workshop on Parallel and Distributed Algorithms and Applications (in conjunction with CANDAR'18)*, Nov. 2018.

[6] F. T. Leighton. *Introduction to Parallel Algorithms and Architectures: Array, Trees, Hypercubes*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1992.

[7] V. Kumar, A. Grama, A. Gupta, and G. Karypis. *Introduction to Parallel Computing: Design and Analysis of Algorithms*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[8] Cuda toolkit 9.2 download.
https://developer.nvidia.com/cuda-downloads.

[9] K. Nakano and S. Matsumae. The super warp architecture with random address shift. In *2013 20th International Conference on High Performance Computing (HiPC)(HIPC)*, volume 00, pages 256–265, Dec. 2013.