Accelerating Dynamic Programming by P-Fold Pipeline Implementation on GPU *

Susumu Matsumae†
Saga University, Saga 840-8502, JAPAN

Abstract

In this paper, we show the effectiveness of pipeline implementations of Dynamic Programming (DP) on Graphics Processing Unit (GPU). We deal with a simplified DP problem where each element of its solution table is calculated in order by semi-group operations among several of already computed elements in the table. We implement the DP program on GPU in a pipeline fashion, i.e., we use GPU cores for supporting pipeline-stages so that several elements of the solution tables are partially computed at one time. Further, to accelerate the pipeline implementation, we propose a p-fold pipeline technique, which enables larger parallelism more than the number of pipeline-stages.

Key Words: Dynamic programming; GPGPU; pipelining.

1 Introduction

In this paper, we show the effectiveness of pipeline implementations of Dynamic Programming (DP) on Graphics Processing Unit (GPU). We deal with a simplified DP problem where each element of its one-dimensional solution table of size n is calculated in order by semi-group computations among several of k already computed elements in the table. Since the size of solution table is n and each element requires computations of k elements, the simplified DP problem is solved sequentially in $O(nk)$ steps.

It has been studied to speed up DP programs using GPU (e.g. [2, 10]), where they mainly focus on optimizing the order of accessing data by proposing novel techniques avoiding memory access conflicts. In our study, however, we consider adopting a pipeline technique and implementing the DP program on GPU in a pipeline fashion. The pipeline computation technique [11] 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 previous studies [5, 6], we solved the simplified DP problem on GPU in a pipeline fashion, i.e., we use GPU cores for supporting pipeline-stages so that several elements of the solution tables are respectively computed partially at one time. Our pipeline implementation determines one output value per one computational step with k threads in a pipeline fashion and solves the simplified DP problem on GPU in $O(n)$ steps, which is k times faster than the $O(nk)$ steps for the sequential algorithm.

Here in this paper, we try to further accelerate the pipeline implementation with larger parallelism. In our previous studies [5, 6], the parallelism is up to k, the number of pipeline-stages. Here, to accelerate the pipeline implementation, we propose a p-fold pipeline technique, and show that it enables larger parallelism more than k. We also discuss the effectiveness of our method with explaining the 0-1 knapsack problem by DP as an example.

The rest of this paper is organized as follows. Section 2 introduces problem definitions and base algorithms. Section 3 explains our pipeline implementations for DP on GPU. Section 4 proposes our p-fold pipeline technique and explains how to accelerate the pipeline implementation by increasing the parallelism more than the number of pipeline-stages. Section 5 provides experimental results. And finally, Section 6 offers concluding remarks.

2 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 a naive and standard GPU implementations of programs.

2.1 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* $\mathtt{ST}[0], \mathtt{ST}[1], \ldots, \mathtt{ST}[n-1]$ *of size n as a solution table, a set* $\mathscr{A} = \{a_0, a_1, \ldots, a_{k-1}\}$ *of k integers representing offset numbers, and a semi-group binary operator* $\otimes$ *over integers are given. Without loss of generality, every element of set* $\mathscr{A}$

*This is the extended version of the paper presented at the 35th International Conference on Computer Applications in Industry and Engineering, 2022 (CAINE 2022) [7].

†Department of Information Science. Email: matsumae@is.saga-u.ac.jp

satisfies the following inequality:

$$a_0 > a_1 > \cdots > a_{k-1} > 0. \quad (1)$$

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

$$\mathtt{ST}[i] = \otimes_{0 \le j < k} \mathtt{ST}[i - a_j] \quad (2)$$

where ST$[0]$,ST$[1]$,...,*and* ST$[a_0 - 1]$ *are preset with initial values.* ■

For example, Fibonacci number problem can be seen as the S-DP problem where $k = 2, a_0 = 2, a_1 = 1$, $\otimes = +$, and ST[0]=ST[1]=1.

Even if a problem is solved by DP with two-dimensional solution table, the problem may be reduced to the S-DP problem of one-dimensional solution table. The interested reader should refer to [6], where the matrix-chain multiplication (MCM) problem (e.g., [1]) is discussed as an example.

2.2 Conventional Approach to S-DP Problem

To begin with, we show a straightforward sequential algorithm that solves the S-DP problem. The algorithm is shown in Figure 1. The outer loop computes values from ST$[a_0]$ to ST$[n-1]$ in order. It should be noted that ST$[0]$,ST$[1]$,...,ST$[a_0 - 1]$ are initially given as input values. The inner loop computes ST[i] for each i by equation (2). Since the outer loop takes $n - a_0 = O(n)$ iterations and the inner loop requires $O(k-1)$ steps, this sequential algorithm solves the S-DP problem in $O(nk)$ steps in total.

A Sequential Algorithm for S-DP Problem

```
for i = a_0 to n − 1 do
    ST[i] = ST[i−a_0];
    for j = 1 to k − 1 do
        ST[i] = ST[i] ⊗ ST[i−a_j];
```

Figure 1: A sequential algorithm for S-DP problem

Next, we consider parallelizing the sequential 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$[\mathtt{i} - a_\mathtt{j}]$, 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. Further, even if those $k-1$ threads could operate in concurrent read and concurrent write (CRCW) mode, obviously we would not get the correct answer because all $k-1$ computations of $\otimes$ in equation (2) would be performed at one time.

To avoid the possible memory access conflicts, we can use a standard parallel prefix computation algorithm (e.g., [3, 4]), in which the computations of $\otimes$ over the k values are executed in a tournament fashion. Since the parallel prefix computation runs in $O(\log k)$ steps for k values, the entire time cost can be improved to $O(n \log k)$ steps even when we use 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.

3 Pipeline Implementation on GPU

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

3.1 Pipeline Implementation for S-DP problem

First, we introduce the pipeline implementation technique shown in our previous studies [5, 6]. In the next section, we will further accelerate the algorithm.

In our parallel implementation, we use a group of k threads to establish k-stage pipeline, and this thread group treats k consecutive elements of array ST at one time in parallel. Figure 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 elements calculated by the k-thread group. The inner loop controls each thread's behavior in such a way that the thread j executes computation for ST$[\mathtt{i} - \mathtt{j}]$ using the value stored in ST$[\mathtt{i} - \mathtt{j} - a_\mathtt{j}]$.

A Pipeline Algorithm for S-DP Problem

```
for i = a_0 to n + k − 2 do
    for j = 0 to k − 1 do in parallel
        Thread j executes the following operation
        if a_0 ≤ i_j < n where i_j = i − j:
        ST[i_j] = { ST[i_j − a_j];          (j = 0)
                  { ST[i_j] ⊗ ST[i_j − a_j]; (j > 0)
```

Figure 2: A pipeline algorithm for S-DP problem

An execution example is shown in Figure 3, where $k = 3$, $a_0 = 6$, $a_1 = 3$, and $a_2 = 1$ hold and the initial values are stored in ST[0], ST[1],... , ST[5].

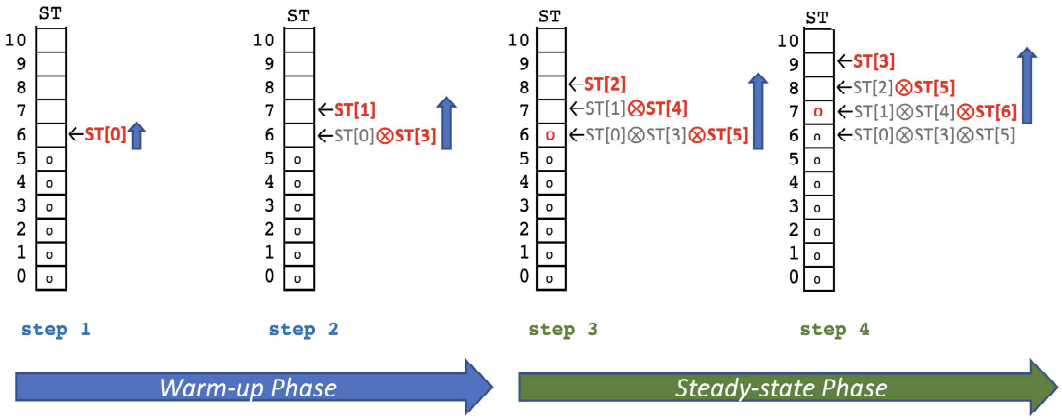

Figure 3: An execution example for the case where $k = 3$, $a_0 = 6$, $a_1 = 3$, and $a_2 = 1$ hold and the initial values are preset to ST[0], ST[1], ..., and ST[5]

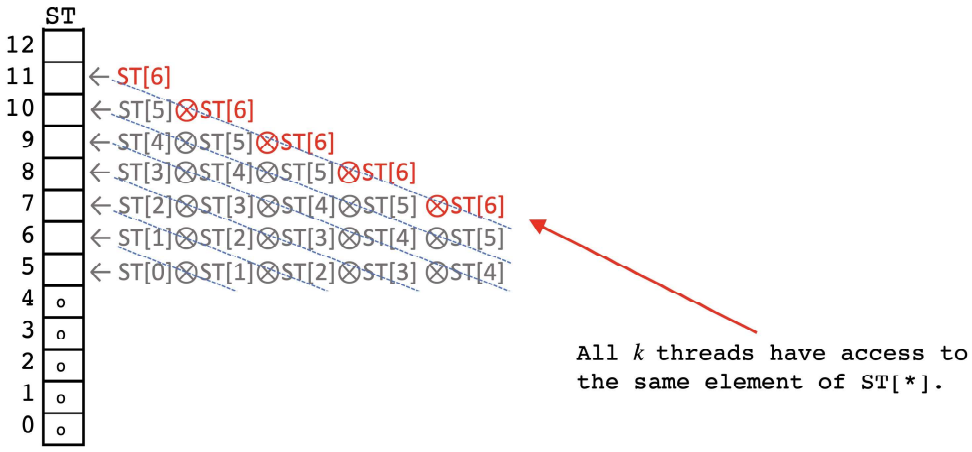

Figure 4: The worst-case example where the offset numbers are consecutively given. In this example, we have $k = 5$ threads and offset numbers are $a_0 = 5$, $a_1 = 4$, $a_2 = 3$, $a_3 = 2$, and $a_4 = 1$

In step 1, the head position i of the elements to be computed is 6. In this step the only one thread is activated and executes ST[6] ← ST[0]. In step 2, the head position is incremented to 7, and two threads are activated. The first thread treats ST[7] and the second thread works on ST[6]. We say that these steps 1 and 2 are in warm-up phase of pipeline, for not all threads are working yet and the number of active threads is gradually incremented one by each step. In step 3, the head position becomes 8, and now all $k = 3$ threads actively execute operations for ST[8], ST[7], and ST[6] respectively. From step 3, all k threads are steadily working and hence we say these steps are in steady-state phase. It should be noted that finally in step 3 the content of ST[6] is completely determined while those of ST[8] and ST[7] are partially computed and not yet fixed. From step 3, all the $k = 3$ threads are active until step $n - a_0$ when the head position i 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 if there is no memory access conflict, because the outer loop takes $n + k - a_0 - 1 = O(n)$ iterations and the inner loop requires $O(1)$ time.

From a practical viewpoint, the inner loop may take more time steps, because the possible memory access conflicts occur. 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 Figure 4 for such a worst-case example. In this example, all five threads try to have access to the same ST[i − 5] at one time in the inner loop. For such a case, we have proposed a *2-by-2 pipeline implementation* technique to avoid memory conflicts, where each thread invoked in the inner loop executes two computations for each element of array ST. The details can be found in [5].

3.2 Correctness of Pipeline Algorithm

Before we improve and accelerate the pipeline implementation in the next section, we consider the sufficient condition for the pipeline algorithm to work correctly and prove that the condition always holds and that our algorithm works correctly.

In the inner loop of the algorithm, in order to (partially) calculate the value of $\mathtt{ST}[i_\mathrm{j}]$, each thread j uses the value of $\mathtt{ST}[i_\mathrm{j} - a_\mathrm{j}]$ assuming that this value has been computed at the time it is referenced. In what follows, we show that the assumption is always true if equation (1) is valid.

In the inner loop (executed in parallel), the values of $\mathtt{ST}[i_\mathrm{j} - a_\mathrm{j}]$ used by the k threads are $\mathtt{ST}[\mathtt{i} - a_0]$, $\mathtt{ST}[\mathtt{i} - 1 - a_1]$, $\mathtt{ST}[\mathtt{i} - 2 - a_2]$,.... , $\mathtt{ST}[\mathtt{i} - (k-1) - a_{k-1}]$. By the equation (1), we can say that the largest value among these k referenced indices is $\mathtt{i} - (k-1) - a_{k-1}$. On the other hand, when the index of the outer loop is i, the elements of array ST have already been calculated up to $\mathtt{ST}[\mathtt{i} - k]$. With these observations, the following inequality must be true for the pipeline algorithm to work correctly:

$$\mathtt{i} - (k-1) - a_{k-1} \ \leq \ \mathtt{i} - k,$$

which leads to

$$1 \leq a_{k-1}.$$

Since the inequality $1 \leq a_{k-1}$ is always true by the equation (1), we can say that our pipeline algorithm correctly calculates the values of array ST.

4 Accelerating Pipeline Implementation on GPU

In the pipeline algorithm introduced in the preceding section, the parallelism is k. In this section, we consider increasing the degree of parallelism further to accelerate the computation.

4.1 Problem with Simultaneous Execution of Multiple Iterations

A simple idea for the acceleration is to let several iterations in the outer loop of the pipeline algorithm be executed in parallel at one time. Such an idea, however, may not work because of the following reason. Each computation of $\mathtt{ST}[i_\mathrm{j}]$ uses the value of $\mathtt{ST}[i_\mathrm{j} - a_\mathrm{j}]$ which had been computed up to that operation. See step 4 of Figure 3 as an example. In step 4, ST[3], ST[5], and ST[6] are referred to, though the computation of ST[6] had just been completed in the preceding step 3. Hence, in the case shown in Figure 3, while step x being executed, the value of ST that had just been calculated in the preceding step $x-1$ is immediately used for the step x, which means that multiple consecutive iterations of the outer loop of the pipeline algorithm cannot be executed at the same time.

However, if the offset values are sufficiently large, it is expected that the required elements for the next calculation had already been computed much earlier. In such a case, there is a possibility that several consecutive iterations of the outer loop may be executed together at the same time. For example, in Figure 5, since all elements up to ST[9] had already been calculated at the time of step 3, all the elements of ST required for the future steps 4, 5, 6, and 7 are already obtained. Hence, in addition to step 3, it seems that it is possible to execute these steps from step 4 to 7 at the same time as well. However, such an idea still arises two problems as follows: possible memory read conflicts and memory write conflicts. The first problem simply leads to the slowdown of execution because those possible memory conflicts should be automatically solved at run-time by the serializing mechanism of GPU. The second problem is from the concurrent write problem, and it is fatal because the calculation cannot be performed correctly. For example, let us consider the case where we simultaneously execute those 5 steps from step 3 to 7 in Figure 5. In such a case, ST[12], for example, is to be written by 3 threads: thread 0 in step 3, thread 1 in step 4, and thread 2 in step 5 try to execute $\mathtt{ST}[12] = \mathtt{ST}[2]$, $\mathtt{ST}[12] = \mathtt{ST}[12] \otimes \mathtt{ST}[4]$, and $\mathtt{ST}[12] = \mathtt{ST}[12] \otimes \mathtt{ST}[7]$, respectively. Obviously, we would not get the correct value by the simultaneous execution of these 3 operations.

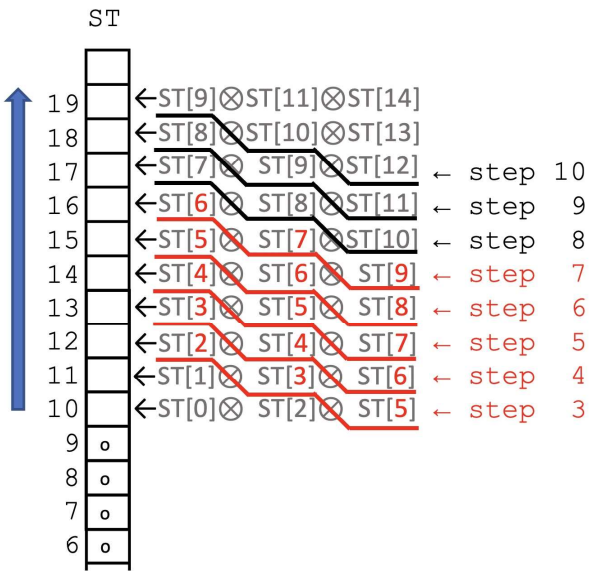

Figure 5: An example of problems with simultaneous execution of multiple iterations, where $k = 3$, $a_0 = 10$, $a_1 = 8$, and $a_2 = 5$

4.2 p-fold Pipeline Implementation on GPU

In this section, we propose a p-fold pipeline technique. Instead of simultaneous execution of multiple iterations of the outer loop in the pipeline algorithm of Figure 2, we modify the operations in the inner loop. The idea is to increase the number of threads for the inner loop.

Figure 6 describes the p-fold pipeline algorithm we propose. Here, the number of threads working in the inner loop is increased to pk from k. Figure 7 shows an execution example when $p = 3$. In the example, 9 ($= pk$) threads are activated

during the inner loop, and the parallelism is increased by a factor of 3 ($= p$) compared to the original pipeline implementation where only 3 ($= k$) threads are activated. In our p-fold pipeline implementation, possible memory read conflicts may occur depending on the offset values, but the memory write conflicts does not. Hence, we can avoid the fatal problem discussed in the preceding subsection and say that our p-fold pipeline algorithm works correctly.

A p-fold Pipeline Algorithm for S-DP Problem

for $\mathtt{i} = a_0 + (p-1)$ **to** $(n-1)+(k-1)p$ **by** p **do**

  for $\mathtt{j} = 0$ **to** $pk-1$ **do** **in parallel**

    Thread $\mathtt{j}$ executes the following operation if $a_0 \le i_\mathtt{j} < n$ where $i_\mathtt{j} = \mathtt{i} - \mathtt{j}$:

$$\mathtt{ST}[i_\mathtt{j}] = \begin{cases} \mathtt{ST}[i_\mathtt{j} - a_{\lfloor \mathtt{j}/p \rfloor}]; & (\lfloor \mathtt{j}/p \rfloor = 0) \\ \mathtt{ST}[i_\mathtt{j}] \otimes \mathtt{ST}[i_\mathtt{j} - a_{\lfloor \mathtt{j}/p \rfloor}]; & (\lfloor \mathtt{j}/p \rfloor > 0) \end{cases}$$

Figure 6: A p-fold pipeline algorithm for S-DP problem

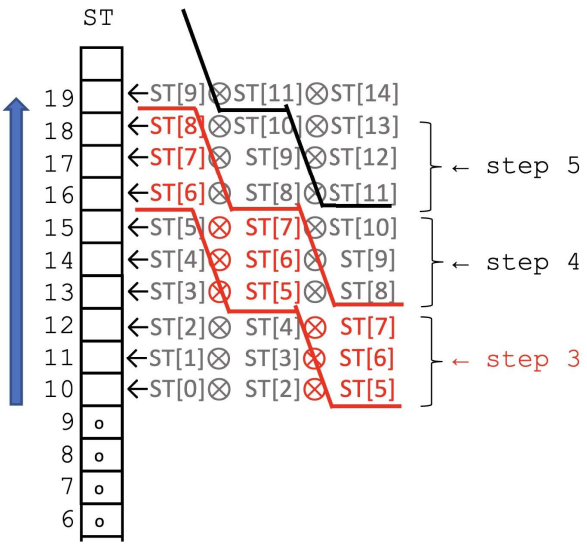

Figure 7: An execution example of the p-fold pipeline algorithm for the case where $p = 3$, $k = 3$, $a_0 = 10$, $a_1 = 8$, and $a_2 = 5$

As for the time-complexity of the p-fold pipeline algorithm, from a theoretical viewpoint, it takes $O(n/p)$ steps if there is no memory read conflict, because the outer loop takes

$$\frac{\{(n-1)+(k-1)p\} - \{a_0 + (p-1)\} + 1}{p} = O\left(\frac{n}{p}\right)$$

iterations and the inner loop requires $O(1)$ time. Since the number of threads is pk, the work is $O(pk \times n/p) = O(nk)$, which is equal to that of sequential algorithm in Figure 1.

4.3 Example: 0-1 Knapsack Problem

Let us consider an example of solving the 0-1 knapsack problem by DP as follows. Given a set of K items (each item is given a weight and a value) and the capacity C of a knapsack, the 0-1 knapsack problem is a problem to find a best way to pack the knapsack so that the total value is maximized within the capacity of the knapsack, by deciding which items to pack and which to exclude.

It is well-known that the 0-1 knapsack problem can be solved by dynamic programming using a two-dimensional solution table $\mathtt{ST}[*,*]$. Each $\mathtt{ST}[i,j]$ stores the best solution for the subproblem $\mathrm{P}(i,j)$, where $\mathrm{P}(i,j)$ is the problem of finding the best way to pack the knapsack that maximizes the value under the two conditions: 1) only up to the i-th item of given items are considered to be packed, and 2) the total weight capacity of the knapsack is limited to j. It should be noted that the original problem is $\mathrm{P}(K,C)$.

The $\mathrm{P}(K,C)$ can be solved by DP as follows. The solution for each $\mathrm{P}(i,j)$ is stored in the two-dimensional solution table $\mathtt{ST}[i,j]$ $(0 \le i \le K, 0 \le j \le C)$. Then, $\mathrm{P}(i,j)$ can be easily solved if the two solutions of its subproblems $\mathrm{P}(i-1,j)$ and $\mathrm{P}(i-1,j-w_i)$ are already known where w_i is the weight of the i-th item. That is, $\mathtt{ST}[i,j]$ can be easily obtained by only checking the two elements $\mathtt{ST}[i-1,j]$ and $\mathtt{ST}[i-1,j-w_i]$. To obtain $\mathrm{P}(K,C)$, we need to fulfill the two-dimensional solution table of size of $(K+1)$ rows and $(C+1)$ columns as Figure 8 in row-major order from the top-left corner to the bottom-right corner, which takes $O(KC)$ steps.

To apply our p-fold pipeline algorithm to this 0-1 knapsack problem, we need to convert the two-dimensional solution table $\mathtt{ST}[*,*]$ to the liner solution table $\mathtt{ST}[*]$. This conversion can be easily done by rearranging elements of $\mathtt{ST}[*,*]$ in row-major order so that each $\mathtt{ST}[i,j]$ is mapped to $\mathtt{ST}[i(C+1)+j]$. It should be noted that in the case of reducing the 0-1 knapsack problem to the S-DP problem, one of the two offset values, a_0, may dynamically change because it depends on w_i. In this sense, the 0-1 knapsack problem is not formulated strictly as an S-DP problem, but it is essentially the same and we can use the p-fold pipeline algorithm. In such a mapping, the calculation of $\mathtt{ST}[i]$ can be executed by using only the two elements of $\mathtt{ST}$ whose indices are at least $(C+1)$ smaller than i. In other words, the offset values of S-DP problem are at least $(C+1)$, and thus we can expect to use the p-fold pipeline algorithm with a large parallelism p (e.g., at least more than C) for solving this 0-1 knapsack problem.

5 Experimental Results

In this section, we show experimental results about the expected number of memory access collision and the number p of the p-fold pipeline implementation.

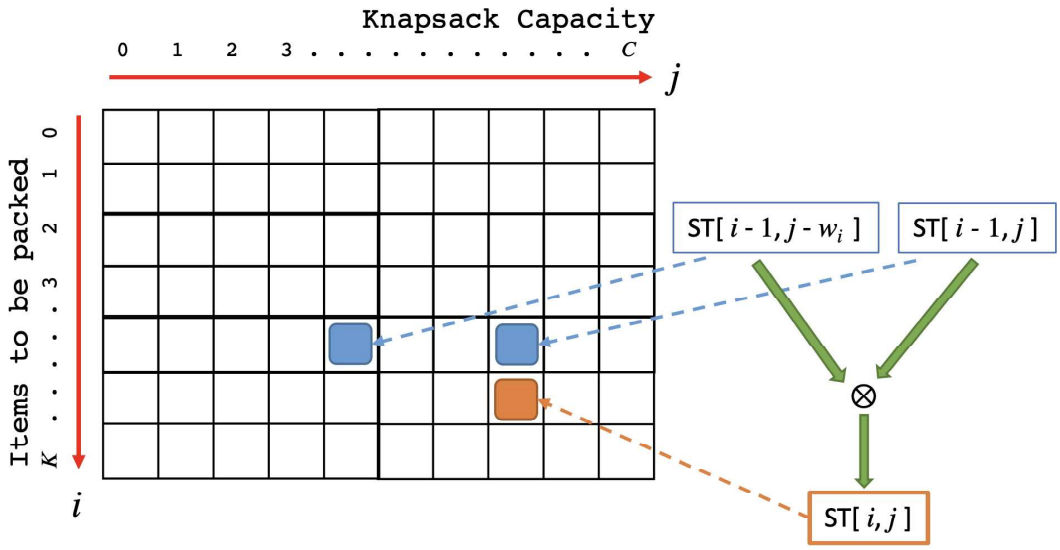

Figure 8: A two-dimensional solution table for the dynamic programming solving 0-1 knapsack problem. Each $\mathtt{ST}[i,j]$ can be calculated by the two elements $\mathtt{ST}[i-1,j]$ and $\mathtt{ST}[i-1,j-w_i]$ in the row one above

5.1 Collision Count of 1-Fold Pipeline Implementation

While the sequential algorithm described in Figure 1 only executes one $\otimes$ operation per step, our pipeline implementation in Figure 2 can process k number of $\otimes$ operations in a step by using k threads. That is, our pipeline implementation solves an S-DP problem at a parallel degree of k. However, from a practical viewpoint, there is a possibility that memory access congestions (concurrent reads to the same element $\mathtt{ST}[i]$) may occur in the pipeline implementation, resulting in performance degradation.

Figure 9 shows simulation results showing collision counts with various K and r. The parameter K is the upper limit of offset values and r is the ratio for controlling the density of selected offset values from the offset value pool $\{1,2,\ldots,K\}$. That is, we choose rK number of offset values from the integer set $\{1,2,\ldots,K\}$. For ratio r we take 0.1, 0.2, 0.3, 0.4, and 0.5. We count the collision counts with moving K from 10 to 640, at a doubling rate of increase. For each pair (r,K), we randomly generate 1000 sets of offset values, and count the maximum number of read collisions occurred in a step. From Figure 9, we can see that the collision counts get larger as K and/or r increase. It should be noted that the rate of increase, however, is not large when K increases.

For more detail on the rate of increase with the parameter r, see Figure 10, which shows the number of expected collisions with increasing the density of offset values in its range $\{1,2,\ldots,K\}$. Here, we fix $K = 50$. When the ratio r is approaching to 1.0, the number of collisions is getting closer to K. This is because we must choose all K offset values from $\{1,2,\ldots,K\}$ when $r = 1.0$, which is the worst-case (the case where all offset values are consecutively selected) scenario.

5.2 Number p of p-Fold Pipeline Implementation

In the p-fold pipeline implementation in Figure 6, the larger the value of p, the greater the degree of parallelism is expected, and the program execution speed becomes faster. However, it is not possible to choose an arbitrarily large value of p. This is because at each step of execution, each of the kp threads respectively reads out $\mathtt{ST}[*]$ for computation and those $\mathtt{ST}[*]$ values must have been completely computed at that point in time. See Figure 11 for example. Although the offset value settings are the same in Figure 7 and 11, in Figure 7 the 3-fold pipeline implementation is feasible, while in Figure 11 the 4-fold pipeline is not.

Figure 12 shows simulation results of how large p can be expected for p-fold pipeline implementation with increasing the lower limit (gap) g of offset values. Here, we fix $K = 100$ and $r = 0.2$, and select rK offset values from integer set $\{g,g+1,\ldots,K+g\}$. As discussed in Section 4.3, we can see that the number p becomes larger as the gap value g increases. For example, it is expected that we can execute 9-fold pipeline implementation for an S-DP problem of $k = 20$ $(= r*K)$ offset values if all offset values are more than 100. In such a case, we can expect higher degree of parallelism like 180 $(= p*k = 9*20)$ by using possibly 9-fold pipeline implementation, while the degree of parallelism is only 20 $(= k)$ for the simple pipeline implementation in Figure 2.

Figure 13 shows simulation results of how large p can be expected for p-fold pipeline implementation with increasing the density of offset values in its range $\{g,g+1,\ldots,K+g\}$. Here, we fix $K = 100$ and $g = 20$. We can see that the p decreases rapidly when ratio r is approaching to 1.0. This is because we must choose all K offset values from $\{g,g+1,\ldots,K+g\}$ when $r = 1.0$, and as a result the worst-case scenario where all offset values are consecutively selected occurs.

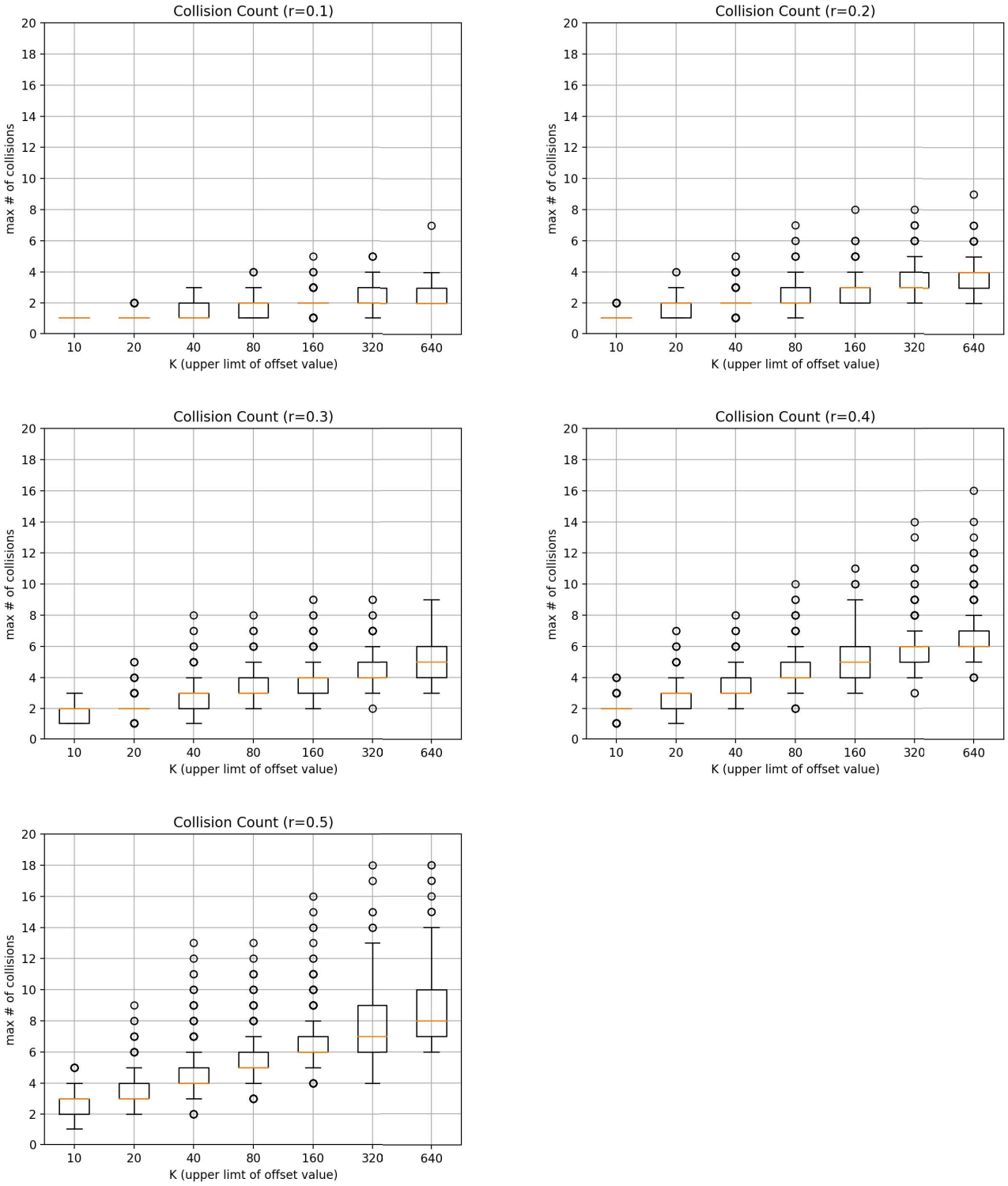

Figure 9: Number of collisions with increasing the upper limit K of offset values. From integer set $\{1,2,\ldots,K\}$, $r*K$ number of offset values are selected. The horizontal axes are of logarithmic scale

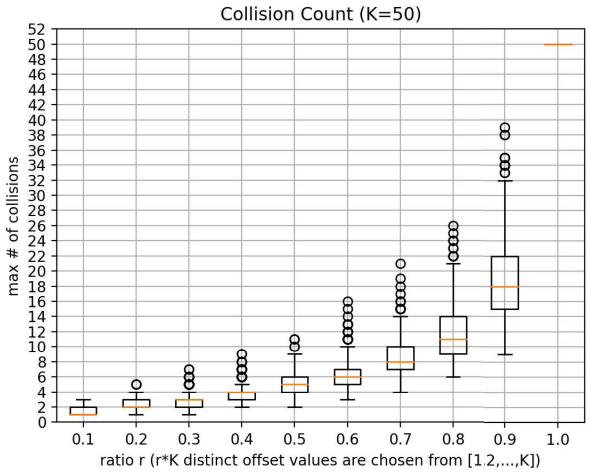

Figure 10: Number of collisions with increasing the density of offset values in its range $\{1,2,\ldots,K\}$. Here, we set $K = 50$

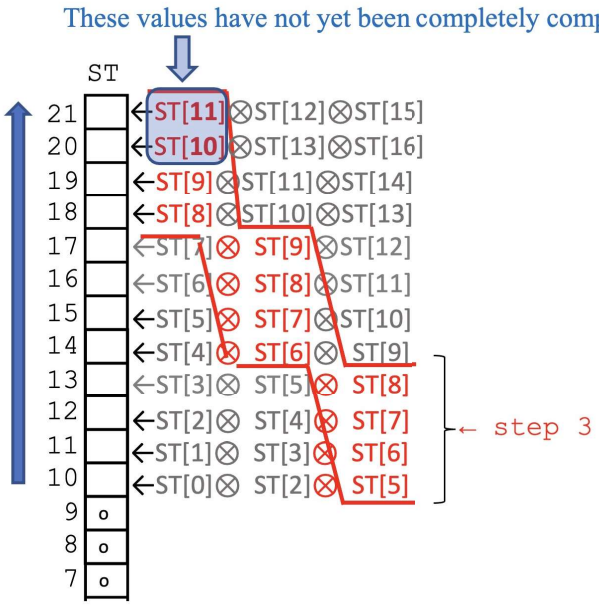

Figure 11: An IMPOSSIBLE execution example of the p-fold pipeline algorithm for the case where $p = 4$, $k = 3$, $a_0 = 10$, $a_1 = 8$, and $a_2 = 5$

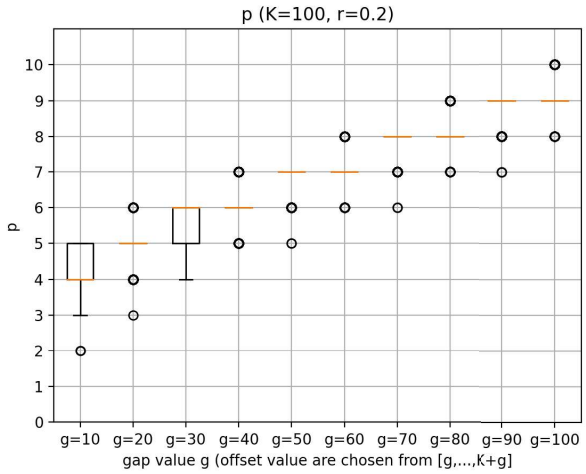

Figure 12: Number p of p-fold pipeline implementation with increasing the lower limit (gap) g of offset values. From integer set $\{g, g+1, \ldots, K+g\}$, $r * K$ offset values are selected. Here, we set $K = 100$ and $r = 0.2$

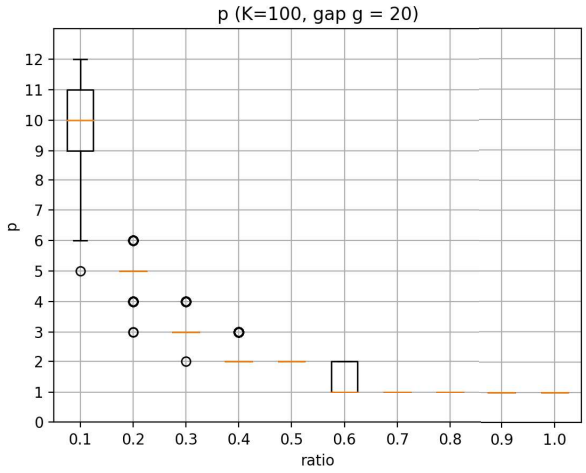

Figure 13: Number p of p-fold pipeline implementation with increasing the density of offset values in its range $\{g, g+1, \ldots, K+g\}$. Here, we set $K = 100$ and $g = 20$

5.3 Collision Count of p-Fold Pipeline Implementation

Figure 14 shows simulation results of the collision count with increasing the lower limit (gap) g of offset values. We can see that collision counts tend to decrease as g increases. The intuitive explanation is as follows. In the p-fold pipeline implementation, at each step, let (idx_1, idx_2, ..., idx_{kp}) be the index sequence of ST[$*$] read by those kp threads in order of thread number th. Then, if those indices are grouped by separating every p indices in order from the first idx_1, it is guaranteed that no index collision will occur within such a group of size p. Thus, since the increase of p implies the increase of size of collision-free index groups, the collision count is expected to decrease.

6 Concluding Remarks

In this study, we examined the effectiveness of pipeline implementations of Dynamic Programming (DP) on GPU. We dealt with a simplified DP problem where each element of its

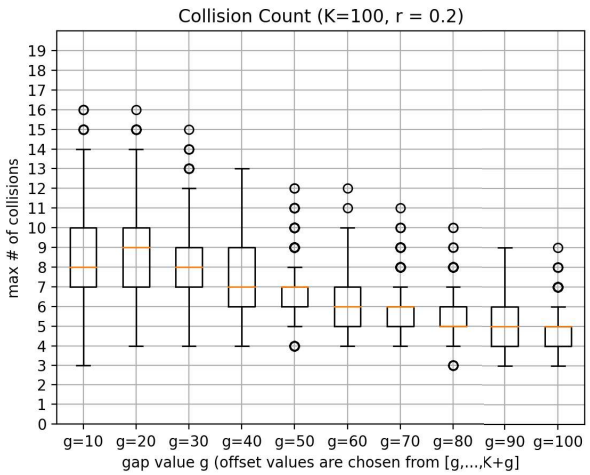

Figure 14: Number of collisions with p-fold pipeline implementation with increasing the lower limit (gap) g of offset value. From integer set $\{g, g+1, \ldots, K+g\}$, $r*K$ offset values are selected. Here, we set $K = 100$ and $r = 0.2$

one-dimensional solution table of size n is calculated in order by semi-group computations among several of k already computed elements in the table and proposed pipeline implementations on GPU model. In our previous studies [5, 6], the degree of parallelism of our pipeline is up to k, but in this paper, we proposed the p-fold pipeline implementation technique and successfully increased the parallelism toward pk, though the parameter p depends on the problem to be solved. As an application example, we explained how to apply our p-fold pipeline technique to the 0-1 knapsack problem. And our experimental results showed that we can expect a large p for the p-fold pipeline.

For future work, we plan to evaluate the performance of our pipeline implementations by conducting experiments on GPU. We also plan to study the performance of our pipeline implementation on theoretical GPU models (e.g., [8, 9]).

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] 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.

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

[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. doi: 10.1109/CANDARW.2018.00063

[6] S. Matsumae and M. Miyazaki. "Solving Dynamic Programming Problem by Pipeline Implementation on GPU." *International Journal of Advanced Computer Science and Applications*, 9(12):518–523, 2018. doi: 10.14569/IJACSA.2018.091272

[7] S. Matsumae. "Accelerating Pipeline Implementation of Dynamic Programming on GPU." In *the 35th International Conference on Computer Applications in Industry and Engineering (CAINE 2022)*, EPiC Series in Computing, 89:52–61, 2022. doi: 10.29007/6mgp

[8] K. Nakano, S. Matsumae, and Y. Ito. "The Random Address Shift to Reduce the Memory Access Congestion on the Discrete Memory Machine." In *2013 First International Symposium on Computing and Networking*, pp.95–103, Dec. 2013. doi: 10.1109/CANDAR.2013.21

[9] K. Nakano and S. Matsumae. "The Super Warp Architecture with Random Address Shift." In *2013 20th International Conference on High Performance Computing (HiPC)*, pp.256–265, Dec. 2013. doi: 10.1109/HiPC.2013.6799118

[10] 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)*, pp.86–95, 2014.

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

Susumu Matsumae is a professor at the Department of Information Science, Faculty of Science and Engineering, Saga University, Japan. He received the M.E. and PhD degrees in computer science from Osaka University in 1996 and 2000, respectively. From 2000 to 2001, he was a research associate of Graduate School of Engineering Science, Osaka University. From 2001 to 2007, he was with Department of Information Systems, Tottori University of Environmental Studies. In 2007, he joined Department of Information Science, Saga University. His research interests include parallel algorithms and architectures, logic, and computational complexity.