Single Kernel Soft Synchronization Technique for Task Arrays on CUDA-enabled GPUs, with Applications

Shunji Funasaka, Koji Nakano, Yasuaki Ito
Department of Information Engineering,
Hiroshima University
Kagamiyama 1-4-1, Higashi-Hiroshima, 739-8527 Japan

***Abstract*—A task array is a 2-dimensional array of tasks with dependency relations. Each task uses the resulting values of some tasks in the left columns, and so it can be started only after these left tasks are completed. Conventional CUDA implementations repeatedly perform a separated CUDA kernel call for each column from left to right to synchronize the computation for tasks. However, this conventional CUDA implementation has several drawbacks: a CUDA kernel call has a certain overhead, and the running time of a CUDA kernel is determined by a CUDA block that terminates lastly. Also, every task must write and preserve the resulting values in the global memory with low memory access performance for the following tasks. The main contribution of this paper is to introduce task arrays and to present Single Kernel Soft Synchronization (SKSS) technique that significantly reduces such overheads for task arrays. The SKSS performs only one CUDA kernel call and CUDA blocks assigned to each row of a task array using a global counter. To clarify the potentiality of our SKSS technique, we have implemented the dynamic programming for the 0-1 knapsack problem, the summed area table computation, and the error diffusion of a gray-scale image using our SKSS technique and compared with previously published best GPU implementations. Quite surprisingly, the experimental results using NVIDIA Titan X show that, our SKSS implementations are 1.29-2.11 times faster for the 0-1 knapsack problem, 1.08-1.56 times faster for the summed area table computation, and 1.61-2.11 times faster for the error diffusion.**

I. INTRODUCTION

A GPU (Graphics Processing Unit) is a specialized circuit designed to accelerate computation for building and manipulating images. Latest GPUs are designed for general purpose computing and can perform computation in applications traditionally handled by the CPU. Hence, GPUs have recently attracted the attention of many application developers. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [1], the computing engine for NVIDIA GPUs. CUDA gives developers access to the virtual instruction set and memory of the parallel computational elements in NVIDIA GPUs. In many cases, GPUs are more efficient than multicore processors [2], since they have thousands of processor cores and very high memory bandwidth.

A CUDA-enabled GPU has multiple steaming processors, each of which has execution cores with integer and floating point operations, shared memory, register file, and L1 cache.

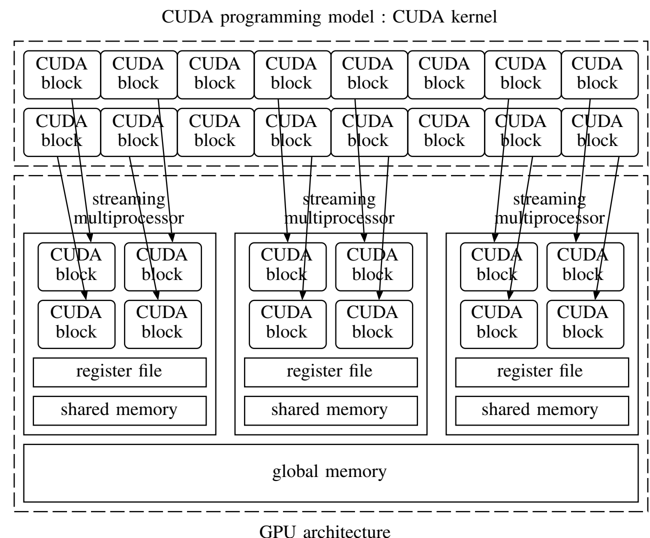

Fig. 1. CUDA programming model and GPU architecture

A CUDA program running on a host PC invokes one or more CUDA kernels executed on a GPU one by one. Each CUDA kernel consists of one or more CUDA blocks, a set of threads running on a streaming multiprocessor. When a CUDA kernel is executed, CUDA blocks are dispatched to streaming multiprocessors in turn. If the number of CUDA blocks in a CUDA kernel that exceeds the total number of CUDA blocks run on a GPU as illustrated in Figure 1, CUDA blocks that are not allocated in a streaming multiprocessor wait for termination of a running CUDA block. Since there is no explicit rule of CUDA block assignment to streaming multiprocessors, we need to design CUDA kernel programs so that they work correctly for any CUDA block assignment to streaming multiprocessors. Hence, there is no direct way to communicate between CUDA blocks in the same CUDA kernel. If we write a CUDA kernel program so that CUDA block A receives a data from CUDA block B, a CUDA kernel may stall due to deadlock; It is possible that CUDA block B can run only after CUDA block A terminates, and CUDA block A terminates only after receiving a data from CUDA

block B. Thus, we should use separated CUDA kernel calls to synchronize execution of CUDA blocks as shown in Figure 2. Since a CUDA block A in a CUDA kernel has been terminated when a CUDA block B in the following CUDA kernel is executed, CUDA block A can transfer a data to CUDA block B through the global memory. Note that such deadlock never occurs in commonly used multiprocessor system running in time sharing mode, because every thread/process will be dispatched and run sooner or later.

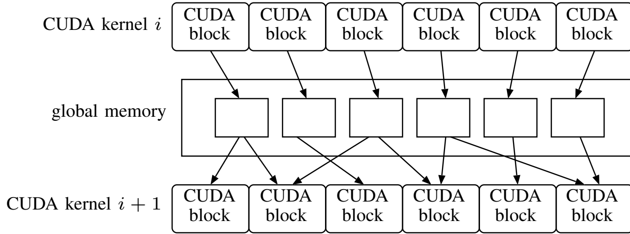

Fig. 2. Synchronization and communication between CUDA blocks in different kernel calls

A task array is a 2-dimensional array of tasks $t_{i,j}$ with dependency relations as illustrated in Figure 3. For example, each task uses the resulting values of some tasks in the left columns, and it can be started only after these left tasks are completed. In the figure, such task dependency is illustrated using an directed edge. Conventional CUDA implementations repeatedly perform a separated CUDA kernel call for each column from left to right to synchronize the computation of tasks so that all tasks in the left rows have been completed when tasks are started. However, this conventional CUDA implementation has several drawbacks. Since each CUDA kernel call has a certain overhead, many CUDA kernel calls degrade the performance. Also, the running time of each CUDA kernel is determined by a CUDA block that terminates lastly. Further, every CUDA block working for a task must write and preserve necessary resulting values in the global memory with low memory access performance, because they are used by following tasks as illustrated in Figure 2.

The main contribution of this paper is to present *Single Kernel Soft Synchronization (SKSS) technique*, which significantly reduces such overheads of conventional CUDA implementations with multiple CUDA kernel calls to complete all tasks in a task array. It performs only one CUDA kernel call, and CUDA blocks in it are assigned to each row of a task array. The assignment is controlled using a global counter so that no deadlock occurs. Also, synchronization is performed by communication through the global memory. To clarify the power and the potentiality of our SKSS technique, we have implemented three parallel algorithms using the SKSS technique: (1) dynamic programming for the 0-1 knapsack problem, (2) summed area table computation, and (3) error diffusion of a gray-scale image. We have also implemented the previously published CUDA implementations for the 0-1

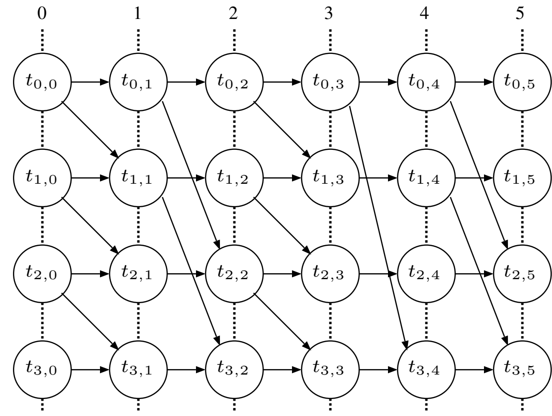

Fig. 3. An example of a forward task array

knapsack problem [3], [4], the summed area table [5], [6], and the error diffusion [7]. The experimental results using NVIDIA Titan X show that, our SKSS implementation is 1.29-2.11 times faster for the 0-1 knapsack problem, 1.08-1.56 times faster for the summed area table computation, and 1.61-2.11 times faster for the error diffusion, than these previously published results.

We can think that the prefix-scan [8], [9], [10] is a 1-dimensional task array that has only the first row of a 2-dimensional one. Separated kernel calls for barrier synchronization are performed in the conventional prefix-sums computation [8]. A synchronization technique using an atomic global counter to control scheduling of prefix-scan has been shown [9], [10]. Our SKSS technique for task arrays is also uses an atomic global counter to ensure that it never stalls.

This paper is organized as follows. In Section II, we introduces forward/fair/backward task arrays and sequential/parallel algorithms to complete all tasks in task arrays. Section III introduces GPU architecture and CUDA programming model to understand the SKSS technique, and shows conventional implementations that performs multiple CUDA kernel calls. We then go on to show our new Single Kernel Soft Synchronization (SKSS) technique that completes all tasks in task arrays by a single CUDA kernel call in Section IV. In Section V, we apply SKSS technique to implement the dynamic programming for the 0-1 knapsack problem, the summed area table computation, and the error diffusion for a gray-scale image, and show the experimental results using NVIDIA Titan X. Section VI concludes our work.

II. TASK ARRAYS AND SEQUENTIAL AND PARALLEL ALGORITHMS

A task array is a 2-dimensional array of $m \times n$ tasks. Let $t_{i,j}$ ($0 \leq i \leq m-1$ and $0 \leq j \leq n-1$) denote a task in the i-th row of the j-th column. *A task graph* is a graph such that nodes are tasks of a task array and directed edges of two tasks represent the dependency of them. A directed edge $(t_{i,j}, t_{i',j'})$ from $t_{i,j}$ to $t_{i',j'}$ means that task $t_{i,j}$ must be completed before task $t_{i',j'}$ starts, because the resulting values of $t_{i,j}$ is used in

the computation of $t_{i',j'}$. We assume that a task graph has *neighbor edges* $(t_{i,j}, t_{i,j+1})$ for all i and j ($0 \le i \le m-1$ and $0 \le j \le n-2$), and *additional edges*. Task arrays can be classified using additional edges as follows:

forward if all additional edges $(t_{i,j}, t_{i',j'})$ of the task graph satisfy $i < i'$ and $j < j'$,

fair if all additional edges $(t_{i,j}, t_{i',j'})$ satisfy $i < i'$ and $j \le j'$, and

backward if all additional edges $(t_{i,j}, t_{i',j'})$ satisfy $i < i'$.

The reader should refer to Figures 3, 4, and 5 illustrating examples of forward/fair/backward task graphs of 4×6 tasks. We call task arrays represented by forward/fair/backward task graphs *forward/fair/backward task array*, respectively. From the definitions, a forward task array is also fair/backward, and a fair task array is backward. Hence, algorithms designed for backward task arrays also work correctly for forward/fair task arrays.

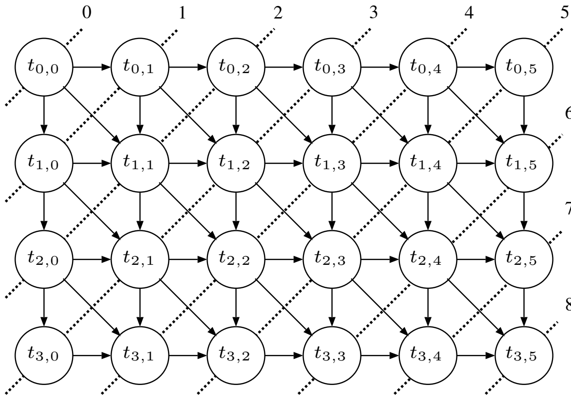

Fig. 4. An example of a fair task array

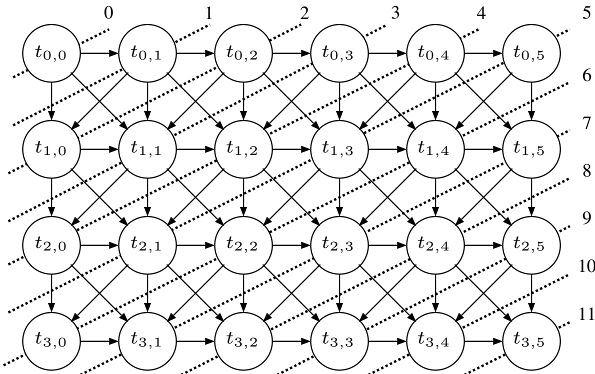

Fig. 5. An example of a backward task array

We can design two straightforward sequential algorithms, Horizontal and Vertical algorithms to complete all tasks in task arrays as follows:

Sequential Algorithm Horizontal
for $i \leftarrow 0$ to $m-1$ **do**
  for $j \leftarrow 0$ to $n-1$ **do**
    Task $t_{i,j}$ is performed;
  end for
end for

Sequential Algorithm Vertical
for $j \leftarrow 0$ to $n-1$ **do**
  for $i \leftarrow 0$ to $m-1$ **do**
    Task $t_{i,j}$ is performed;
  end for
end for

Sequential Algorithm Horizontal works correctly for forward/fair/backward task arrays, because they have no upper-directed edge. However, Sequential Algorithm Vertical may not work for a backward task array because the task graph may have left-directed edge such as $(t_{0,1}, t_{1,0})$ in Figure 5.

Next, we will show parallel algorithms for task arrays. Each task can be started after tasks in the left columns are completed. Thus, we can design a parallel algorithm for a forward task array as follows:

Parallel Algorithm Forward
for $j \leftarrow 0$ to $n-1$ **do**
  for $i \leftarrow 0$ to $m-1$ **do in parallel**
    Task $t_{i,j}$ is performed;
  end for
end for

In Figure 3, tasks in the same dotted line are performed at the same time.

A parallel algorithm for a fair task graph is a little complicated. We say that a task $t_{i,j}$ is *k-diagonal* if $k = i + j$. From Figure 4, k-diagonal tasks can be performed after all $(k-1)$-diagonal tasks are completed. Using this idea, Parallel Algorithm Diagonal performs k-diagonal tasks from $k = 0$ to $m + n - 2$ in turn. The details are spelled out as follows:

Parallel Algorithm Diagonal
for $k \leftarrow 0$ to $m + n - 2$ **do**
  $i \leftarrow \max(0, k - n + 1)$;
  $j \leftarrow \min(k, n - 1)$;
  $L \leftarrow \min(k + 1, (m + n - 2) - k + 1, m, n)$;
  for $l \leftarrow 0$ to $L - 1$ **do in parallel**
    Task $t_{i+l,j-l}$ is performed;
  end for
end for

In Figure 4, tasks in the same dotted line are performed in parallel in Parallel Algorithm Diagonal and numbers correspond to the values of k.

Similarly to Parallel Algorithm Diagonal, we can write a parallel algorithm for a backward task array. However, it is more complicated, we will show a generic parallel algorithm using a topological ordering of the corresponding task graph. A topological ordering of a task graph is an order of tasks such that, for every directed edge $(t_{i,j}, t_{i',j'})$, task $t_{i,j}$ appears before $t_{i',j'}$. The topological ordering can be obtained by repeatedly removing a task with no incoming edge. For example, for a backward task graph in Figure 5, we have a

topological ordering

$$t_{0,0}t_{0,1}t_{0,2}t_{1,0}t_{0,3}t_{1,1}t_{0,4}t_{1,2}t_{2,0}t_{0,5}t_{1,3}t_{2,1}\cdots t_{3,5}.$$

We can partition the topological ordering such that no two nodes in the same partition are connected. For example, the topological ordering above can be partitioned as follows:

$$t_{0,0}|t_{0,1}|t_{0,2}t_{1,0}|t_{0,3}t_{1,1}|t_{0,4}t_{1,2}t_{2,0}|t_{0,5}t_{1,3}t_{2,1}|\cdots|t_{3,5}.$$

We assign serial numbers from 0 to these partitions of tasks. In Figure 5, each dotted line with a number corresponds to a partition. Let $\mathcal{T}_{k,i}$ denote the i-th task ($0 \leq i \leq m_k - 1$) in partition k ($0 \leq k \leq P - 1$), where P is the number of partitions and m_k is the number of tasks in partition k. For example, $\mathcal{T}_{0,0} = t_{0,0}$, $\mathcal{T}_{1,0} = t_{0,1}$, $\mathcal{T}_{2,0} = t_{0,2}$, $\mathcal{T}_{2,1} = t_{1,0}$, ..., hold in the example above. Clearly, all tasks in the same partition can be done in parallel after all tasks in the previous partitions are completed. Thus, we can design Parallel Algorithm Backward for backward task arrays as follows:

Parallel Algorithm Backward
for $k \leftarrow 0$ to $P - 1$ **do**
  for $i \leftarrow 0$ to $m_k - 1$ **do in parallel**
    Task $\mathcal{T}_{k,i}$ is performed;
  end for
end for

Please note that Parallel Algorithm Backward works correctly not only for backward task arrays but also forward/fair task arrays.

Note that in Parallel Algorithms Forward/Fair/Backward, the barrier synchronization is necessary after each completion of parallel for-loop **for ... do in parallel** if tasks are processed asynchronously. We can also design an asynchronous parallel algorithm for task arrays that uses no barrier synchronization. For each task $t_{i,j}$, let $T(t_{i,j})$ be a set of tasks such that

$$T(t_{i,j}) \quad = \quad \{t_{i',j'} \mid \text{task graph has edge } (t_{i',j'}, t_{i,j})\}.$$

Clearly, we can start task $t_{i,j}$ after all tasks in $T(t_{i,j})$ are completed. Thus, we can design parallel algorithm using $T(t_{i,j})$ as follows:

Parallel Soft Synchronization Algorithm
for $i \leftarrow 0$ to $m - 1$ **do in parallel**
  for $j \leftarrow 0$ to $n - 1$ **do**
    Wait until all tasks in $T(t_{i,j})$ are completed;
    Task $t_{i,j}$ is performed;
  end for
end for

Clearly, in this parallel algorithm, no barrier synchronization is necessary. We can think that synchronization is performed by waiting for completion of all tasks $T(t_{i,j})$. We call it *soft synchronization*.

III. Conventional CUDA implementations for Parallel Algorithms

This section first explains CUDA programming model and GPU architecture briefly to understand CUDA implementations of parallel algorithms and how they are executed on GPUs. We then go on to show conventional CUDA implementations of parallel algorithms for task arrays.

A CUDA program running on a host PC invokes one or more *CUDA kernels* executed on a GPU in turn. Each CUDA kernel has one or more *CUDA blocks*, a set of threads running on a GPU. Threads in a CUDA block are partitioned into groups of 32 threads each called *warps*. All threads in the same warp work completely synchronously, share the program counter, and execute the same instruction. Thus, if Parallel Algorithms Forward/Diagonal/Backward shown in Section II are executed by a warp of 32 threads, no barrier synchronization is necessary.

Each CUDA block is dispatched to a streaming multiprocessor in a GPU as illustrated in Figure 1. Each thread in a CUDA block is assigned to registers in the register file. Also, a CUDA block is assigned to some space of the shared memory which can be accessed by all threads in the same CUDA block. A CUDA block occupies such hardware resources on the streaming multiprocessor until all threads in it terminate For example, NVIDIA TITAN X has 28 streaming multiprocessors, each of which can execute up to 2048 (resident) threads and up to 32 CUDA blocks. Hence, for example, a streaming multiprocessor can run 32 CUDA blocks with 32 threads, 32 CUDA blocks with 64 threads, or 16 CUDA blocks with 128 threads. All threads in CUDA blocks allocated to the same streaming multiprocessor work concurrently. Since each streaming multiprocessor has 128 cores, up to 128 threads in 4 warps run at the same time. Thus, warps in CUDA blocks allocated to the streaming multiprocessor run in time sharing mode such that 4 warps out of $\frac{2048}{32} = 64$ (resident) warps are active. When all threads in all CUDA blocks in a CUDA kernel terminate, the CUDA kernel is completed and the next CUDA kernel is called.

When a CUDA block of a CUDA kernel running on a streaming multiprocessor terminates, a new CUDA block that has not been allocated yet is assigned to it and starts running. For example, suppose that a CUDA kernel with 1000 CUDA blocks with 64 threads each is executed on NVIDIA TITAN X. Each of 28 streaming multiprocessors can run 32 CUDA blocks, $28 \cdot 32 = 896$ CUDA blocks out of 1000 CUDA blocks are allocated to them. The remaining 104 CUDA blocks wait for termination of running CUDA blocks. Note that there is no explicit rule to select CUDA blocks to be allocated to streaming multiprocessor. Thus, direct synchronization and communication between different CUDA blocks in the same CUDA kernel are not possible. If synchronization and/or communication between CUDA blocks is necessary, they must be implemented as separated CUDA kernels. For example, suppose that CUDA kernel $i + 1$ is executed after CUDA kernel i as illustrated in Figure 2. CUDA blocks in CUDA kernel i write the resulting values of their computation in the global memory and terminate. After that, CUDA blocks in kernel call $i + 1$ can read these resulting values. In this manner, synchronization and communication of CUDA blocks in different CUDA kernels are possible. However, when a CUDA block terminates, all data stored in the shared memory

and register files written by threads in it are discarded. Thus, all necessary data used by CUDA blocks in the following CUDA kernel must be written and preserved in the global memory of the GPU. Since memory access capability of the global memory is not high, such data backup operations may degrade the performance.

Let us implement Parallel Algorithm Forward. We assume that each task $t_{i,j}$ can be done by a CUDA block. We use separated CUDA kernels to synchronize computation of tasks performed by multiple CUDA blocks. More specifically, each i-th CUDA blocks of j-th CUDA kernel performs task $t_{i,j}$. The resulting values of tasks are written in the global memory. The details of the implementation are spelled out as follows:

CUDA implementation of Parallel Algorithm Forward

```
/* CUDA kernel */
Parallel-Forward(j)
  i ← blockId;
  task t_{i,j} is performed;
/* Host PC CUDA program */
for j ← 0 to n − 1 do
    Call Parallel-Forward(j) with m − 1 CUDA blocks;
end for
```

In this implementation, blockId returns the index of a CUDA block from 0 to $m-1$ for CUDA kernel call with m CUDA blocks. Parallel Algorithm Fair can be implemented in the same way as Parallel Algorithm Forward.

Parallel Algorithm Backward can be implemented using CUDA kernel for each partition as follows:

CUDA implementation of Parallel Algorithm Backward

```
/* CUDA kernel */
Parallel-Backward(k)
  i ← blockId
  task T_{k,i} is performed;
/* Host PC CUDA program */
for k ← 0 to P − 1 do
    Call Parallel-Backward(k) with m_k CUDA blocks;
end for
```

These conventional CUDA implementations have several drawbacks. First, they perform many CUDA kernel calls with large overhead; CUDA implementation of Parallel Algorithm Backward calls CUDA kernel P times. In particular, a CUDA kernel call can be started after the previous kernel is completed. Hence, the running time of a kernel is determined by a CUDA kernel that terminates lastly. Further, when a CUDA block is terminated, all data stored in the shared memory and registers in a streaming multiprocessor are discarded. Thus, data transfer between tasks must be done through the global memory implemented in very slow off-chip DRAM, even if both tasks are performed by a CUDA block with the same blockId.

IV. Single Kernel Soft Synchronization (SKSS) technique for task arrays

This section explains our *Single Kernel Soft Synchronization (SKSS) technique* to complete all tasks in a backward task array.

To overcome drawbacks of the conventional implementations in Section III, we start with a naive implementation such that each row is assigned to a CUDA block, which performs tasks in it in turn. More specifically, we use a single kernel call in which CUDA block i performs tasks $t_{i,0}$, $t_{i,1}$, $\ldots$, $t_{i,n-1}$. The details are spelled out as follows:

Wrong implementation for Parallel Soft Synchronization Algorithm

```
/* CUDA Kernel */
WrongSoftSynchronization()
i ← blockId;
t_{i,0} is performed;
for j ← 1 to n − 1 do
    Wait until all tasks in T(t_{i,j}) are completed;
    t_{i,j} is performed;
end for
/* Host PC CUDA program */
Call WrongSoftSynchronization() with m CUDA blocks;
```

We use an array of 1-bit flags in the global memory to write the status of tasks. A CUDA block writes 1 in the corresponding flag if a task is completed. The reader may think that this implementation works correctly. However, this implementation may stall due to the deadlock. Let r be the total number of CUDA blocks that can be dispatched to streaming multiprocessors in a GPU. Also, let p be the number of rows reachable from $t_{0,0}$ in a task array. We assume that p are so large that $r < p$ hold, and show that WrongSoftSynchronization may stall. After calling CUDA kernel WrongSoftSynchronization, r CUDA blocks are running and the remaining $m-r$ CUDA blocks wait for termination of running CUDA blocks. Suppose that CUDA block with blockId 0 is waiting and r CUDA blocks are arranged to rows (excluding row 0) reachable from $t_{0,0}$. Since p rows are reachable from $t_{0,0}$ and $r < p$, this arrangement is possible. If $t_{0,0}$ is not completed, none of r CUDA blocks terminates. Also, if none of r CUDA blocks terminates, CUDA block with blockId 0 never runs. Therefore, WrongSoftSynchronization() never terminates.

The main idea of our *Single Kernel Soft Synchronization (SKSS) technique* is to modify WrongSoftSynchronization() such that it never stalls due to deadlock. It uses an integer variable c initialized by 0 in the global memory as a global counter and CUDA atomic function atomicAdd(&c,1) that exclusively increments the value of c by 1 and returns the value of c before increment. Suppose that m threads perform atomicAdd(&c,1). Then, each of m threads receives 0, 1, $\ldots$, $m-1$ and no two threads receive the same value. Thus, we can assign sequential ID from 0 to $m-1$ to m threads. We use this global counter technique to assign CUDA blocks to tasks. The first thread in every CUDA block performs atomicAdd(&c,1). Let i be the return value of atomicAdd(&c,1). The corresponding CUDA block perform tasks $t_{i,0}$, $t_{i,1}$, $\ldots$, $t_{i,n-1}$ in row i one by one. Note that, before starting $t_{i,j}$, it must wait until all tasks in $T(t_{i,j})$ are completed. After $t_{i,n-1}$ is completed, it performs atomicAdd(&c,1) and repeats the same procedure if it is less

than m. The details of the SKSS technique for a backward task array are spelled out as follows:

CUDA implementation for task arrays
/* CUDA Kernel Call */
SKSS()
The first thread performs $i \leftarrow \text{atomicAdd}(\&c, 1)$;
while $i \leq m-1$ **do**
  for $j \leftarrow 0$ to $n-1$ **do**
    Wait until all tasks $T(t_{i,j})$ are completed;
    $t_{i,j}$ is performed;
  end for
  The first thread performs $i \leftarrow \text{atomicAdd}(\&c, 1)$;
end while
/* Host PC CUDA program */
Call SKSS(i, j) with m CUDA blocks;

When a CUDA block performs atomicAdd(&c,1) and receives return value i ($\leq m-1$), running CUDA blocks have been already assigned to rows from 0 to $i-1$. Thus, CUDA kernel call SKSS() can perform all tasks in a task array correctly. Also, since a CUDA block performs all tasks in the same row i, it is not necessary to write and read the resulting values of $t_{i,j-1}$ in the global memory to perform $t_{i,j}$.

Let us see how CUDA blocks in SKSS kernel call work. Again, let r be the total number of CUDA blocks that can be dispatched to streaming multiprocessors at the same time. If $m \leq r$, all m CUDA blocks can be arranged streaming to multiprocessor at the same time. Hence, SKSS can complete all tasks. If $m > r$ then r CUDA blocks arranged to rows 0 to $r-1$ by atomicAdd function for global counter c. The remaining $m-r$ CUDA blocks wait for termination of running r blocks. When one of the running CUDA blocks completes all tasks in the assigned row, the first thread performs atomicAdd(&c,1) again, and receives return value r. Thus, this CUDA block performs tasks in row r. When another running CUDA block completes all tasks, it receives return value $r+1$ of atomicAdd(&c,1) and starts tasks in row $r+1$. The same computation is repeated until return value of atomicAdd(&c,1) exceeds $m-1$. When a running CUDA block receives return value m, it terminates. After that, one of waiting CUDA block is allocated to a streaming multiprocessor and the first thread performs atomicAdd(&c,1). Since the return value is larger than $m-1$, it terminates immediately. After that, each running CUDA block that completes tasks in the assigned row performs atomicAdd(&c,1), receives a return value larger than $m-1$, and terminates. Also, all waiting blocks perform atomicAdd(&c,1), receive return values larger than $m-1$, and terminate. In this way, all tasks are completed. Note that r running CUDA blocks perform m tasks and $m-r$ waiting CUDA blocks perform no task. Further, we can call SKSS(i, j) with r CUDA blocks if m is too large and the exact value of r can be estimated. This may improve the performance a little, because no CUDA block waits and useless $m-r$ atomicAdd function calls are omitted.

SKSS is designed as *CUDA block-wise* in the sense that each task is performed by a CUDA block. We can also modify SKSS to *warp-wise* such that each task is assigned to a warp and the first thread of a warp performs $i \leftarrow \text{atomicAdd}(\&\text{c}, 1)$. If the return value $i \leq m-1$ then 32 threads in the warp performs tasks in the i-th row. This implementation makes sense if each task can be done very efficiently by a warp of 32 threads. If this is the case, we use CUDA blocks with 64 threads each to maximize the number of resident threads running in streaming multiprocessors, because each streaming multiprocessor can have up to 2048 resident threads and 32 resident CUDA blocks. If we use CUDA blocks of 32 threads each, only 1024 resident threads can be allocated to a streaming multiprocessor. Since more resident threads can fully utilize the memory access bandwidth, we should use CUDA blocks with 64 threads each. Actually, all our implementations shown in Section V uses CUDA blocks with 64 threads each to execute warp-wise SKSS.

V. APPLICATION OF TASK ARRAYS AND EXPERIMENTAL RESULTS

This section shows applications of task arrays and experimental results. As applications of forward/fair/backward task arrays, we use the 0-1 knapsack problem, the summed area table computation, and the error diffusion, respectively. To evaluate the performance, we have used NVIDIA Titan X, which has 28 streaming multiprocessors with 128 processor cores, 2048 resident threads, 96K-byte shared memory, and 64K 32-bit registers. The running time of sequential algorithm using a single thread of Core i7 6700K CPU is also shown just for reference.

A. Forward task arrays: dynamic programming for the 0-1 knapsack problem

Suppose that n items with pairs (v_j, w_j) of value v_j and weight w_j for all i $(1 \leq i \leq n)$ and an upper bound W of the total weight are given. The goal of the 0-1 knapsack problem [11] is to find a subset T of $\{1, \ldots n\}$ that

$$\text{maximizes } \sum_{j \in T} v_j \quad \text{subject to } \sum_{j \in T} w_j \leq W.$$

In other words, the best subset of T with the total weight less than W that maximizes the total value must be found. The value of $\sum_{j \in T} w_j$ for the optimal subset T can be obtained efficiently by the dynamic programming technique, which uses a 2-dimensional array V of size $(W+1) \times (n+1)$. In the dynamic programming algorithm, $V_{i,j}$ $(0 \leq i \leq W$ and $0 \leq j \leq n)$ will store the maximum value of $\sum_{k \in T'} w_k$ with subject to $\sum_{k \in T'} w_k \leq i$ for the optimal subset $T' \in \{1, 2, \ldots, j\}$. The reader should have no difficulty to confirm the following recursive formula is correct:

$$\begin{aligned} V_{i,0} &= 0 \\ V_{i,j} &= V_{i,j-1} \quad \text{if } i < w_j \\ V_{i,j} &= \max(V_{i,j-1}, V_{i-w_j,j-1} + v_i) \quad \text{if } i \geq w_j \end{aligned}$$

Figure 6 shows the values of V for the knapsack problem with input $(2, 4), (2, 2), (3, 3), (4, 1)$.

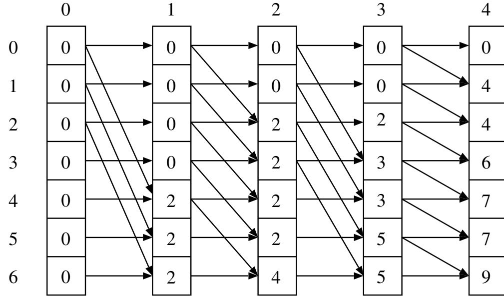

Fig. 6. The values of V for the 0-1 knapsack problem with input $(2, 4), (2, 2), (3, 3), (4, 1)$

Suppose that the computation of each $V_{i,j}$ is task $t_{i,j}$. From the computation illustrated in Figure 6, we can see that the corresponding task array is a forward task array shown in Figure 3.

We partition V of size $(W + 1) \times (n + 1)$ into strips and consider that the computation of each strip is a task. We use two types of partitions.

Single column strip V is partitioned into $\frac{W+1}{32} \times (n + 1)$ strips of size 32×1 each.

Multiple column strip V is partitioned into $\frac{W+1}{32} \times \frac{n+1}{k}$ strips of size $32 \times k$ each. The value of $k \geq 2$ can be any integer.

The resulting task array of single column strip is forward (Figure 3), while that of multiple column strip is fair (Figure 4). Thus, for single column strip, we can use CUDA kernels Parallel-Forward to compute all values of V. Actually, Boyer *et al.* [3] has presented an implementation based on Parallel-Forward for single column strip. O'Connel *et al.* [4] has presented an implementation that uses Parallel-Fair for multiple column strip. We have implemented SKSS for single column strip.

Table I shows the running time for solving the 0-1 knapsack problem. The running time is evaluated for $n + 1 = 4096$ and $W+1 =$ 16K, 32K, 64K, 128K, 256K, and 512K. For efficient coalesced global memory access, transposed V is stored in the global memory, that is, each $V_{i,j}$ is stored as $V[j][i]$ in the global memory. Each value v_i is a randomly generated number less than 4096 and is stored as a 4-byte float. Each weight w_i is a randomly generated integer at most $\frac{4W}{4096}$ and is stored as a 4-byte integer. Since the average of weights is $\frac{2W}{4096}$, the optimal subset T includes approximately half of items. The running time of Parallel-Fair based algorithm shown in [4] are evaluated for $k = 2, 4, 8, 16$ and 32, and selected the minimum running time. For all values of $W+1$, the minimum running time is attained when $k = 4$ or 8. The table also shows the speedup ratio of our SKSS-based implementation over the fastest implementation of Parallel-Forward [3] and Parallel-Fair [4] for each value of $W + 1$. Our SKSS-based implementation is always faster than the previously published implementations and attains a speedup factor of 1.29-2.11.

TABLE I
THE RUNNING TIME FOR SOLVING THE 0-1 KNAPSACK PROBLEM IN MILLISECONDS

$W + 1$	16K	32K	64K	128K	256K	512K
Parallel-Forward [3]	14.83	16.01	19.61	27.13	42.67	87.8
Parallel-Fair [4]	6.484	11.22	20.48	33.78	67.97	126.3
SKSS	5.037	5.515	9.315	15.14	27.24	52.37
Speedup	1.29	2.03	2.11	1.79	1.57	1.68
Sequential	62.87	109.7	261.0	605.5	1259	2584

B. *Fair task arrays: summed area table of a matrix*

The summed area table is a data structure used for texture-map computations [12]. Suppose that a matrix a of size $n \times n$ is given. The summed area table b of a is a matrix of the same size such that

$$b_{i,j} = \sum_{i'=0}^{i} \sum_{j'=0}^{j} a_{i',j'}.$$

The summed area table has many applications in the area of image processing [13]. Using the summed area table, the sum of elements in any rectangular area of a can be computed efficiently in $O(1)$ operations by the following formula:

$$\sum_{i=s}^{t} \sum_{j=u}^{v} a_{i,j} = b_{t,v} - b_{t,u} - b_{s,v} + b_{s,u}.$$

Hence, the summed area table can be used for applying the average filter for an image.

The summed area table can be obtained by computing the row-wise prefix-sums and then computing the column-wise prefix-sums in parallel as follows:

Parallel Summed Area Table Algorithm
for $j \leftarrow 1$ to $n - 1$ **do**
  for $i \leftarrow 0$ to $n - 1$ **do in parallel**
    $a[i][j] \leftarrow a[i][j] + a[i][j - 1]$;
  end for
end for
for $i \leftarrow 1$ to $n - 1$ **do**
  for $j \leftarrow 0$ to $n - 1$ **do in parallel**
    $a[i][j] \leftarrow a[i][j] + a[i - 1][j]$;
  end for
end for

In this algorithm, each $a_{i,j}$ is stored in $a[i][j]$. Clearly, each $a[i][j]$ stores the value of $b_{i,j}$ when this algorithm terminates. Using this parallel algorithm, a warp of 32 threads can compute the summed area table of size 32×32 on the shared memory very efficiently.

The summed area table b can also be computed by the

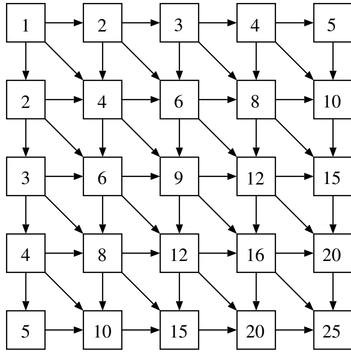

Fig. 7. The values of the summed area table for a matrix with all elements being 1

following recursive formula:

$$
\begin{aligned}
b_{0,0} &= a_{0,0} \\
b_{i,j} &= a_{i,j} + b_{i,j-1} \quad \text{if } i = 0 \text{ and } j \geq 1 \\
b_{i,j} &= a_{i,j} + b_{i-1,j} \quad \text{if } j = 0 \text{ and } i \geq 1 \\
b_{i,j} &= a_{i,j} + b_{i-1,j} + b_{i,j-1} - b_{i-1,j-1} \quad \text{otherwise}
\end{aligned}
$$

Figure 7 shows the values of the summed area table for a matrix with all elements being 1. From this figure, we can see that the computation of the summed area table by the recursive formula above represented as a fair task array illustrated in Figure 4.

We partition the computation of the summed area table b such that $\frac{n}{32} \times \frac{n}{32}$ groups of 32×32 elements each. More specifically, each task $t_{i,j}$ is the computation of $b_{i',j'}$ such that $32i \leq i' \leq 32i + 31$ and $32j \leq j' \leq 32j + 31$. Each task $t_{i,j}$ ($i \geq 1$ and $j \geq 1$) uses the resulting values of

- $b_{32i-1,32j-1}$ computed by $t_{i-1,j-1}$,
- $b_{32i-1,32j}, b_{32i-1,32j+1}, \ldots, b_{32i-1,32j+31}$ computed by $t_{i-1,j}$, and
- $b_{32i,32j-1}, b_{32i+1,32j-1}, \ldots, b_{32i+31,32j-1}$ computed by $t_{i,j-1}$.

We use a warp of 32 threads for each task. To complete task $t_{i,j}$, an assigned warp first computes the local summed area table $b'_{32i+k,32j+l}$ ($0 \leq k, l \leq 31$) such that

$$b'_{32i+k,32j+l} = \sum_{i'=32i}^{32i+k} \sum_{j'=32j}^{32j+l} a_{i',j'}$$

using Parallel Summed Area Table Algorithm. After that, the summed area table assigned to task $t_{i,j}$ can be obtained by computing the following formula for all k and l ($0 \leq k, l \leq 31$)

$$
\begin{aligned}
b_{32i+k,32j+l} = \; & b'_{32i+k,32j+l} + b_{32i-1,32j+l} \\
& + b_{32i+k,32j-1} - b_{32i-1,32j-1}.
\end{aligned}
$$

Thus, each task $t_{i,j}$ computing 32×32 elements of b can be done efficiently by a warp of 32 threads, and both Parallel-Fair as well as SKSS can complete all tasks. Since we have $\frac{n}{32} \times \frac{n}{32}$ groups, CUDA kernel Parallel-Fair is called $2\frac{n}{32} - 1$ times.

TABLE II
THE RUNNING TIME FOR COMPUTING THE SUMMED AREA TABLE IN MILLISECONDS

n	1K	2K	4K	8K	16K	32K
2R1W [6]	0.06287	0.1927	0.6579	2.511	9.763	38.23
1R1W [5]	0.2213	0.4571	1.020	2.786	9.513	31.97
hybrid [5]	0.07124	0.2123	0.6712	2.528	9.244	31.30
SKSS	0.05812	0.1494	0.4219	1.659	6.861	28.79
Speedup	1.08	1.29	1.56	1.51	1.35	1.09
Sequential	1.056	4.339	17.40	71.77	277.5	1021

Nehab *et al.* [6] have presented a sophisticated method to compute the summed area table in three CUDA kernel calls. They also partition the input matrix into $\frac{n}{32} \times \frac{n}{32}$ groups of 32×32 elements each. In the first CUDA kernel call, the row-wise sums $\mathcal{R}$, the column-wise sums $\mathcal{C}$, and the sum $\mathcal{S}$ of each group are computed. The second CUDA kernel computes the row-wise prefix-sums of $\mathcal{R}$, the column-wise prefix-sums of $\mathcal{C}$, and the summed area table of the sums $\mathcal{S}$ for all groups. Finally, in the third CUDA kernel, each warp computes the (global) summed area table for the assigned group by combining the resulting values of the second CUDA kernel and the input elements of the assigned group. We call this implementation *2R1W SAT*, because each input element of an input matrix is read twice, and each resulting value is written once. This parallel algorithm is very efficient for small matrices because the number of warps used in each kernel is so large that each of $\frac{n^2}{32^2}$ groups assigned a warp.

Kasagi *et al.* [5] have presented Parallel-Fair based implementation, which calls Parallel-Fair $2\frac{n}{32} - 1$ times. We call this implementation *1R1W SAT*, because each input element is read and write once each. They also presented a hybrid implementation such that first k calls and the last k calls in $2\frac{n}{32} - 1$ CUDA kernel calls are replaced by 2R1W SAT algorithm. Since earlier and later CUDA kernel calls of 1R1W SAT use fewer warps, this hybrid implementation may run faster than 2R1W SAT and 1R1W SAT. We can select the value of parameter k so that the running time of this hybrid implementation is minimized.

Table II shows the running time of 2R1W SAT, 1R1W SAT, their hybrid, and our SKSS for computing the summed area table for 4-byte float matrices of sizes 1K×1K, 2K×2K, ..., and 32K×32K. It also shows the speedup ratio of SKSS over the fastest of 2R1W, 1R1W, and hybrid for each size. From the table, SKSS always runs faster than the previously published best implementation, and the speedup ratio is 1.08-1.56. The running time of a sequential algorithm is also shown just for reference.

C. Backward task arrays: Error diffusion and error collection

Error diffusion[14], [15] is one of the most well-known digital halftoning algorithms to generate a binary image that reproduces an input gray-scale image. The key idea of the error diffusion is to distribute rounding errors to four unprocessed neighboring pixels using coefficients shown in Figure 8. Let

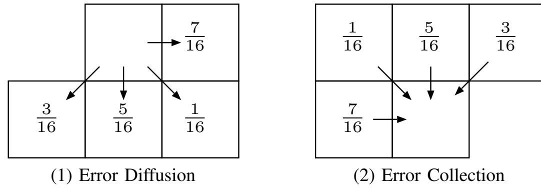

Fig. 8. Error diffusion and Error collection

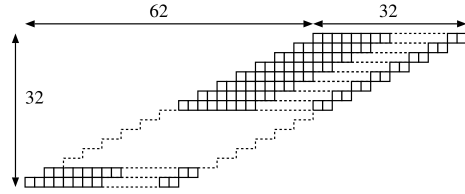

Fig. 9. A group of 32×32 pixels arranged in a parallelogram

a be a gray-scale image of size $n \times n$ such that each pixel $a[i][j]$ $(0 \leq i, j \leq n-1)$ takes an intensity level (i.e. a real number) in the range $[0, 1]$. Error diffusion outputs a binary image b of the same size such that each pixel $b[i][j]$ takes a binary value (i.e. 0 or 1). Error diffusion operation rounds the value of $a[i][j]$ to 0 or 1 and the resulting binary value is stored in $b[i][j]$. Rounding error e $(= a[i][j] - b[i][j])$ is diffused to neighboring unprocessed four pixels. The details of error diffusion is spelled out as follows:

Error diffusion
for $i \leftarrow 0$ to $n-1$ **do**
  for $j \leftarrow 0$ to $n-1$ **do**
    if $a[i][j] \leq \frac{1}{2}$ **then** $r \leftarrow 0$; **else** $r \leftarrow 1$;
    $b[i][j] \leftarrow r$; $e \leftarrow a[i][j] - r$;
    $a[i][j+1] \leftarrow a[i][j+1] + \frac{7}{16} \cdot e$;
    $a[i+1][j+1] \leftarrow a[i+1][j+1] + \frac{1}{16} \cdot e$;
    $a[i+1][j] \leftarrow a[i+1][j] + \frac{5}{16} \cdot e$;
    $a[i+1][j-1] \leftarrow a[i+1][j-1] + \frac{3}{16} \cdot e$;
  end for
end for

For simplicity, we assume that the values of $a[i][j]$ such that $i = -1, n$ or $j = -1, n$ are zero to avoid special treatment for boundary pixels.

Error collection is a more efficient halftoning algorithm developed by Kasagi *et al.* [7]. The resulting binary image of error collection is exactly the same as that obtained by error diffusion. Similar to error diffusion, error collection scans input image a in raster scan order, and for each pixel in a, rounding errors are collected from neighboring processed four pixels using coefficients shown in Figure 8.

Error Collection
for $i \leftarrow 0$ to $n-1$ **do**
  for $j \leftarrow 0$ to $n-1$ **do**
    $s \leftarrow a[i][j] + \frac{7}{16} \cdot a[i][j-1] + \frac{1}{16} \cdot a[i-1][j-1] + \frac{5}{16} \cdot a[i-1][j] + \frac{3}{16} \cdot a[i-1][j+1]$;
    if $s \leq \frac{1}{2}$ **then** $r \leftarrow 0$; **else** $r \leftarrow 1$;
    $a[i][j] \leftarrow s - r$; $b[i][j] \leftarrow r$;
  end for
end for

For each pair of neighboring pixels, errors diffused/collected are the same, and thus resulting binary images b generated by error diffusion and error collection are identical. Error collection performs only one write operation to a, while error diffusion performs four write operations. Thus, error collection is more efficient than error diffusion in terms of memory access.

Suppose that the computation of $a[i][j]$ is a task $t_{i,j}$. The task array is backward as illustrated in Figure 5, because $t_{i,j}$ uses the resulting values of $t_{i,j-1}$, $t_{i-1,j-1}$, $t_{i-1,j}$, and $t_{i-1,j+1}$.

We group all elements in a so that tasks within a group are represented as a fair task array and a warp of 32 threads can complete them very efficiently. For this purpose, each group has 32×32 pixels arranged in a parallelogram as illustrated in Figure 9. Each of 32 rows has 32 pixels and row $i+1$ is slided by two pixels leftwards for row i. As illustrated in Figure 10, we think that the computation of error collection for each pixel in a parallelogram is a task. We can see that the task array illustrated in Figure 10 satisfies the condition of forward task arrays illustrated in Figure 4. Thus, Parallel Algorithm Forward for $m = 32$ and $n = 32$ can complete all tasks in a parallelogram using a warp of 32 threads. More specifically, each thread i $(0 \leq i \leq 31)$ performs $t_{i,0}, t_{i,1}, \ldots, t_{i,31}$ in turn. Since all 32 threads in a warp work completely synchronously, all tasks in the parallelogram can be done correctly and efficiently.

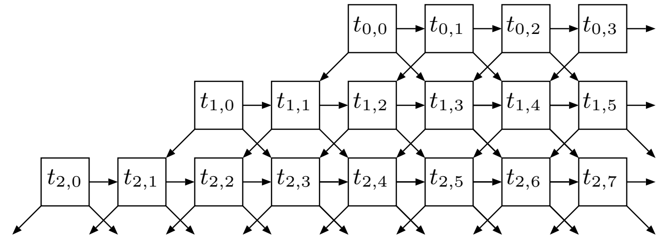

Fig. 10. Tasks for pixels in the parallelogram

Next, let us consider that the computation of a parallelogram is a task to be computed by a warp. As illustrated in Figure 11, we partition the input image into parallelograms. In this figure, an image of 128×128 pixels is partitioned into parallelograms such that $\frac{128}{32} = 4$ strips with 32 rows have $\frac{128}{32} + 2 = 6$ parallelograms each. Thus, we have a task array of size 4×6 and can draw a task graph as illustrated in the figure. For example, the resulting values of $t_{0,3}$ and $t_{0,4}$ are necessary to start $t_{1,2}$, because the pixel values a in the bottom row of parallelograms corresponding to $t_{0,3}$ and $t_{0,4}$ are used to perform $t_{1,2}$. Hence, the task graph has edges $(t_{0,3}, t_{1,2})$ and $(t_{0,4}, t_{1,2})$. In general, the task graph for a $n \times n$ image has

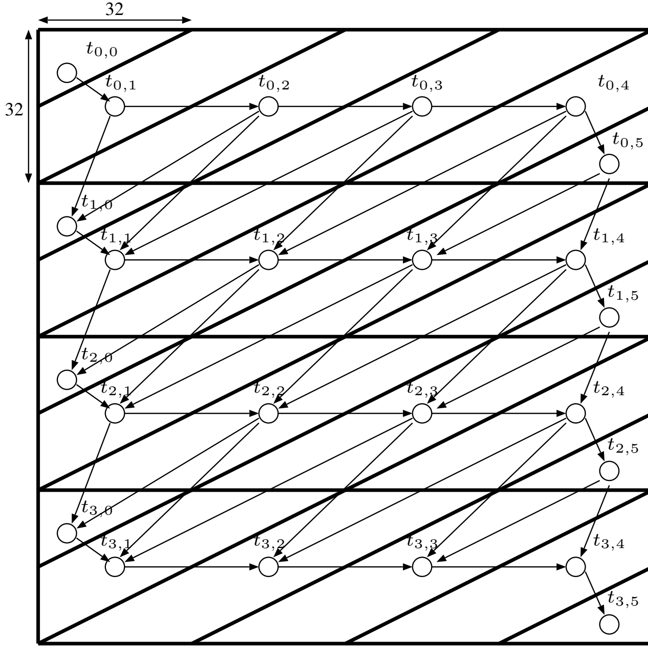

Fig. 11. Tasks for parallelogram in the image of size 128×128

following edges:

- $(t_{i,j}, t_{i,j+1})$ $(0 \leq i \leq \frac{n}{32} - 1$ and $0 \leq j \leq \frac{n}{32})$,
- $(t_{i,j}, t_{i+1,j-1})$ $(0 \leq i \leq \frac{n}{32} - 2$ and $1 \leq j \leq \frac{n}{32} + 1)$, and
- $(t_{i,j}, t_{i+1,j-2})$ $(0 \leq i \leq \frac{n}{32} - 2$ and $2 \leq j \leq \frac{n}{32} + 1)$.

Therefore, the task array thus obtained are backward, and both Parallel-Backward as well as SKSS can complete all tasks. Let us evaluate the number of CUDA kernel Parallel-Backward called in the conventional implementation. For example, a topological ordered tasks of the task graph in Figure 11 have 15 partitions, $t_{0,0}|t_{0,1}|t_{0,2}|t_{1,0}t_{0,3}|t_{1,1}t_{0,4}|t_{1,2}t_{0,5}|$ $t_{2,0}t_{1,3}|t_{2,1}t_{1,4}|t_{2,2}t_{1,5}|t_{3,0}t_{2,3}|t_{3,1}t_{2,4}|t_{3,2}t_{2,5}|t_{3,3}|t_{3,4}|t_{3,5}$. and each partition has at most 2 tasks. In general, the number of partition is $4 \cdot \frac{n}{32} - 1$ with at most $\lceil \frac{n}{96} \rceil$ tasks. Thus, CUDA kernel Parallel-Backward is called $4 \cdot \frac{n}{32} - 1$ times, and each kernel call has at most $\lceil \frac{n}{96} \rceil$.

Kasagi *et al* [7] have presented a Parallel-Backward based implementation for error collection. Table III shows the running time of the implementations using Parallel-Backward and our SKSS. An 8-bit gray-scale image of size $n \times n$ given in the global memory, and the binary image of the same size is written in the global memory as 8-bit pixels. Quite surprisingly, our SKSS is 1.61-2.11 times faster than Parallel-Backward based implementation [7]. Since Parallel-Backward performs too many CUDA kernel calls with few CUDA blocks, it has large overhead for invoking CUDA kernels and resource usage of GPU is not large enough. The running time of a sequential algorithm is also shown for reference.

VI. Conclusion

We have presented Single Kernel Soft Synchronization (SKSS) technique, which reduces the overhead of conventional implementations using multiple CUDA kernel calls. The experimental results using NVIDIA Titan X show that, our SKSS implementation is 1.29-2.11 times faster for the 0-1 knapsack problem, 1.08-1.56 times faster for the summed area table computation, and 1.61-2.11 times faster for the error diffusion, over the previously published best implementations. We believe that SKSS technique is promising and can be applied to many algorithms. In particular, many dynamic programming based algorithms [16] can be implemented very efficiently using our SKSS technique.

TABLE III
The running time for error collection in milliseconds

n	1K	2K	4K	8K	16K	32K
Parallel-Backward [7]	1.119	2.242	4.586	9.272	19.17	48.65
SKSS	0.5982	1.098	2.185	4.391	11.90	29.89
Speedup	1.87	2.04	2.10	2.11	1.61	1.63
Sequential	6.436	25.92	103.3	412.3	1649	6483

References

[1] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 8.0," Mar 2017.

[2] D. Man, K. Uda, H. Ueyama, Y. Ito, and K. Nakano, "Implementations of a parallel algorithm for computing Euclidean distance map in multicore processors and GPUs," *International Journal of Networking and Computing*, vol. 1, no. 2, pp. 260–276, July 2011.

[3] V. Boyer, D. E. Baz, and M. Elkihel, "Solving knapsack problems on GPU," *Computers and Operations Research*, vol. 39, no. 1, pp. 42–49, January 2012.

[4] J. F. O'Connell and C. L. Mumford, "An exact dynamic programming based method to solve optimisation problems using GPUs," in *Proc. of International Symposium on Computing and Networking*, Dec. 2014, pp. 347–353.

[5] A. Kasagi, K. Nakano, and Y. Ito, "Parallel algorithms for the summed area table on the asynchronous hierarchical memory machine, with GPU implementations," in *Proc. of International Conference on Parallel Processing (ICPP)*, Sept. 2014, pp. 251–250.

[6] D. Nehab, A. Maximo, R. S. Lima, and H. Hoppe, "GPU-efficient recursive filtering and summed-area tables," *ACM Trans. Graph.*, vol. 30, no. 6, p. 176, 2011.

[7] A. Kasagi, K. Nakano, and Y. Ito, "Parallelization techniques for error diffusion with GPU implementations," in *Proc. of International Symposium on Computing and Networking*, Dec. 2015, pp. 30–39.

[8] M. Harris, S. Sengupta, and J. D. Owens, "Chapter 39. parallel prefix sum (scan) with CUDA," in *GPU Gems 3*. Addison-Wesley, 2007.

[9] J. Breitbart, "Static GPU threads and an improved scan algorithm," in *Proc. of Euro-Par 2010 Parallel Processing Workshops (LNCS 6586)*, Aug. 2010, pp. 373–380.

[10] S. Yan, G. Long, and Y. Zhang, "StreamScan: fast scan algorithms for GPUs without global barrier synchronization," in *Proc. of ACM SIGPLAN symposium on Principles and practice of parallel programming*, Aug. 2013, pp. 229–238.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, *Introduction to Algorithms*. MIT Press, 1990.

[12] F. C. Crow, "Summed-area tables for texture mapping," *ACM SIGGRAPH Computer Graphics*, vol. 18, no. 3, pp. 207–212, July 1984.

[13] J. Hensley, T. Scheuermann, G. Coombe, M. Singh, and A. Lastra, "Fast summed-area table generation and its applications," *Computer Graphics Forum*, vol. 24, no. 3, pp. 547–555, Sept. 2005.

[14] D. L. Lau and G. R. Arce, *Modern Digital Halftoning, Second Edition*. CRC Press, 2008.

[15] R. W. Floyd and L. Steinberg, "An adaptive algorithm for spatial gray scale," *SID 75 Digest,Society for Information Display*, pp. 36–37, 1975.

[16] Y. Ito and K. Nakano, "A GPU implementation of dynamic programming for the optimal polygon triangulation," *IEICE Transactions on Information and Systems*, vol. E96-D, no. 12, pp. 2596–2603, Dec. 2013.