A Study on Efficient GPU Implementations to Compute the Diameter of a Graph

(グラフの直径計算のための
効率的なGPU実装に関する研究)

2020年9月

広島大学大学院 工学研究科
情報工学専攻

髙 藤 大 介

Abstract

For analysis of complicated networks, we often use the graphs, consisting of the node set and the edge set. It is known that computing of the distance of all pairs of nodes of the graph helps us analyzing. For example, the diameter of a graph is defined by the minimum distance among the distance of all pairs of nodes of the graph, and this values is useful to evaluate network latency. The Floyd-Warshall algorithm is a well-known algorithm to compute the distance of all pairs of nodes of a graph, and its complexity is $O(n^3)$, where n is the number of nodes. The Blocked Floyd-Warshall algorithm, a variant of the Floyd-Warshall has been proposed to accelerate the Floyd-Warshall algorithm by means of a GPU architecture. 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. GPUs have recently attracted the attention of many application developers.

The first contribution of this thesis is to propose efficient parallel implementations of the Blocked Floyd-Warshall algorithm, and evaluate their capability through computing experiment. The previously published GPU implementations for the Blocked Floyd-Warshall algorithm perform many separated kernel calls for costly barrier synchronization. This thesis proposes GPU implementations, which performs no barrier synchronization and invokes only one kernel call. The previously published GPU implementations are called by multiple kernel implementation, and the implementation proposed in this thesis is called by single kernel one. Experimental results using NVIDIA Tesla V100 GPU show that our implementation runs up to 1.31 times faster than the previously published one. Our implementation with SIMD (Single Instruction, Multiple Data) functions also runs up to 1.28 times faster than it. Secondly, this thesis proposes efficient GPU implementations to execute the Blocked Floyd-Warshall algorithm for many graphs at the same time. A *bulk* computation of a sequential algorithm is to execute it for many independent inputs in turn or in parallel. From the experimental results, single kernel implementation runs up to 1.60 times faster than multiple kernel one. In terms of implementations with SIMD functions, the single kernel implementation runs up to 1.89 times faster than it. Also the low-latency implementations for many graphs are proposed. Finally, the parallel Floyd-Warshall algorithm is implemented on the multicore processors.

Several important tasks, including matrix computation, signal processing, sorting, dynamic programming, encryption and decryption can be performed by *oblivious* sequential algorithms. A sequential algorithm is oblivious if an address accessed at each time does not depend on the input data. The second contribution of this thesis is to present a time-optimal implementation for bulk computation of an oblivious sequential algorithm. This paper presents a tool, named C2CU, which automatically generates a CUDA C program for a bulk computation of an oblivious sequential algorithm. The C2CU has been used to generate CUDA C programs for the bulk computation of the bitonic sorting, Floyd-Warshall, and Montgomery modulo multiplication algorithms. Compared to a sequential implementation on a single CPU, the generated CUDA C programs for the above algorithms run, respectively, 199, 54, and 78 times faster.

Acknowledgements

The author would like to express his deep gratitude to Professor Koji Nakano for his encouragement, guidance and many suggestions in the course of this research efforts. The author wishes to thank Professor Satoshi Fujita and Associate Professor Yasuaki Ito for their comments and reviewing this dissertation.

The author also acknowledges Professor Tadashi Dohi, Professor Tsukasa Hirashima, Professor Chuzo Iwamoto, Professor Kazufumi Kaneda, Professor Takio Kurita, Professor Yasuhiko Morimoto, Professor Hiroaki Mukaidani, Professor Toru Nakanishi, Professor Hiroyuki Okamura, Associate Professor Yusuke Hayashi, Associate Professor Katsunobu Imai, Associate Professor Sayaka Kamei, Associate Professor Teruaki Kitasuka, Associate Professor Jun-ichi Miyao, Associate Professor Bisser Raytchev, Associate Professor Tadashi Shima and Associate Professor Toru Tamaki of the Information Engineering major for their comments.

Special thanks are due to Professor Jacir L. Bordim (at University of Brasilia) for his valuable suggestions and heartfelt support during the entire course of this investigation. The author also thanks his colleagues at Laboratory for Computer Systems and Embedded Systems for their friendly help and discussions.

Finally the author would like to thank Professor Emeritus Toshimasa Watanabe (at Hiroshima University), Professor Qi-Wei Ge (at Yamaguchi University), Dr. Satoshi Taoka (at Hiroshima University) Dr. Toshiya Mashima (at Hiroshima International University), and Dr. Masahiro Yamauchi (at Kinki University), for their assistance.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A Graphics Processing Unit (GPU) is a specialized circuit designed to accelerate computation or building and manipulating images [21]. The most recent 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 applications developers [21]. NVIDIA provides a parallel computing architecture called *CUDA* (Compute Unified Device Architecture) [41], 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, since they have hundreds of processor cores and very high memory bandwidth [16, 48, 49].

The Floyd-Warshall algorithm [9,12,53] is a well-known dynamic programming algorithm to compute the distance (or the length of a shortest path) of all pairs of nodes in a directed graph. Let n denote the number of nodes of a directed graph. More specifically, $A(i, j)$ $(0 \leq i, j \leq n-1)$ represents the length of directed edge (i, j) for the adjacency matrix A of given directed graph. The Floyd-Warshall algorithm outputs the distance matrix D such that each $D(i, j)$ is the length of a shortest path from node i to j. Though its time complexity is $O(n^3)$, it has many applications. For example, it can be used to compute the diameter and the Average Shortest Path Length (ASPL) of a directed graph [37]. If the algorithm is implemented on the GPU as it is, n kernel calls are necessary for barrier synchronization. To accelerate the computation of the Floyd-Warshall algorithm, the Blocked Floyd-Warshall algorithm [52] has been implemented on the GPU [25, 27, 43]. In the Blocked Floyd-Warshall algorithm, the distance matrix D is partitioned into $\frac{n}{W} \times \frac{n}{W}$ tiles of size $W \times W$ each. Usually, $W = 32$ is selected to manipulate a tile of size 32×32 using several warps of 32 threads each. Basically, a CUDA block is assigned to a tile and works for updating the values in it. Their GPU implementations of the Blocked Floyd-Warshall algorithm involve $\frac{n}{W}$ iterations of 3 kernel calls each. Thus, it invokes $3\frac{n}{W}$ kernel calls and so we call it *the multiple kernel implementation*. Also, the first kernel call of each iteration uses only one CUDA block, and so parallelism of it is quite low. The researchers have implemented the parallel Floyd-Warshall algorithm on the GPU [8, 20, 31, 44, 56]. Furthermore, some implementations on other architectures are proposed. Abdelghany K, et al. [1] have present a parallel implementation using

multiprocessors, and Bondhugula U. et al. [7] have proposed an FPGA implementation. Also, Han S-C et al. [18] have proposed a CPU implementation using SIMD instructions. Accelerated algorithms [15,19,55] to find all-pairs shortest paths are proposed. There are a lot of work [30, 46] for the Single-Source Shortest Path problem.

The first main contribution of this thesis is to present efficient implementations of the Blocked Floyd-Warshall algorithm that we call *the single kernel implementation*. We use *the Single Kernel Soft Synchronization (SKSS) technique* [11, 17], which we have presented to execute dynamic programming algorithm on the GPU efficiently. It performs only one kernel call, and CUDA blocks are assigned to tiles dynamically. It has no barrier synchronization by separated kernel call, more CUDA blocks work in parallel and so parallelism is high. The experimental results for graphs with $n = 256, 512, \ldots, 32768$ nodes show that our implementation is 1.05-1.31 faster than the previously published implementation on NVIDIA Tesla V100 GPU. Our implementation with SIMD functions runs also 1.00-1.28 times faster than it. Furthermore, we also propose efficient GPU implementations to execute the Blocked Floyd-Warshall algorithm for many graphs at the same time using the SKSS technique. From the experimental results for $m = 8, 16, \ldots, 1024$ graphs with $n = 128, 256, \ldots, 2048$ nodes each, our implementation runs 1.03-1.60 times faster than the multiple kernel implementation for many graphs. Our implementation with SIMD functions runs 1.01-1.89 times faster than the multiple kernel implementation for many graphs.

We present the low-latency implementations of the Blocked Floyd-Warshall algorithm for many graphs on NVIDIA Tesla V100 GPU. The low-latency implementation outputs the results 1.45-5.90 times faster than multiple kernel implementation, and 1.38-5.87 times faster than the single kernel one. Also, the low-latency implementation with SIMD functions outputs the results 1.11-2.43 times faster than multiple kernel one with SIMD functions, and 1.07-2.36 times faster than single kernel one with SIMD functions. Furthermore, on the multicore processors, we implement the parallel Floyd-Warshall algorithm. We show the experimental results on Intel Skylate-X CPU.

The second contribution of this thesis is to show an implementation of the bulk execution of oblivious sequential algorithms on the UMM. Our implementation runs in $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm. We also prove that this implementation is time-optimal in the sense that any implementation takes at least $\Omega(\frac{pt}{w} + lt)$ time units on the UMM.

As a second contribution, we propose the *C2CU* tool, which allows converting a sequential C program into a CUDA C program. As the resulting CUDA C program makes coalescing memory access, even developers with few knowledge of CUDA C programming and GPU architecture can automatically generate CUDA C programs tailored for the bulk execution. To assess the performance of the C2CU generated programs, we have measured the running time of the bulk execution of three oblivious sequential algorithms: (i) bitonic sort [3, 4]; (ii) Floyd-Warshall algorithm [9, 13, 54]; and (iii) Montgomery modulo multiplication [6, 32, 47]. For this purpose, the aforementioned sequential algorithms have been written in C programming language. These sequential implementations were then fed to the C2CU generator to produce the corresponding CUDA

C programs. The CUDA C programs have been executed on the GeForce GTX Titan GPU. Compared to the sequential implementation, running on a single CPU, the bulk execution of the bitonic sort runs 199 times faster while the Floyd-Warshall algorithm and Montgomery modulo multiplication run, respectively, 54 and 78 times faster. These experimental results are quite surprising since over 100 times acceleration have been obtained.

The bulk execution latency of the generated CUDA C implementation can be improved depending on the execution pattern. Thus, the third contribution of this work is to propose a modified execution pattern of the bulk execution so as to reduce latency. Experimental results show that, using the appropriate parameters, latency of the bulk execution can be reduced without compromising the overall running time.

Chapter 2

GPU Architecture

A *Graphics Processing Unit (GPU)* is a specialized circuit designed to accelerate computation for building and manipulating images [21, 28, 50]. 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 [21, 38, 39, 42, 51]. NVIDIA provides a parallel computing architecture called *CUDA* (*Compute Unified Device Architecture*) [41], 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 [29], since they have hundreds of processor cores and very high memory bandwidth. GPU has multiple streaming multiprocessors (*SMs*) and the *global memory*. Each streaming multiprocessor has multiple cores and the *shared memory*. The shared memory is an extremely fast, on-chip memory, with lower capacity, usually 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, allowing it to have a large capacity, currently about 1.5-6 Gbytes, but its access latency is very long. Hence, efficient usage of the shared memory and the global memory is key for CUDA developers to accelerate applications using GPUs. Figure 2.1 illustrates the GPU hardware architecture. CUDA provides a hierarchy of thread groups: Kernel, CUDA block, Thread.

When a kernel is executed, it launches CUDA blocks, in which the number is limited to the resources of SMs. Note that CUDA blocks wait for being launched. When a CUDA block complete the computation and SM has enough resources to launch CUDA blocks, the waiting CUDA blocks are launched. The order of launched CUDA blocks is arbitrary. After all CUDA blocks complete the computation, the kernel halt.

We need to consider *bank conflicts* of the shared memory access and *coalescing* of the global memory access [29, 39, 40]. The global memory access is *coalescing* if the continuous locations in address space of the global memory are accessed in the same time. The address space of the shared memory is mapped into several physical memory banks. If two or more threads access the same memory bank at the same time, the access requests are processed in turn. The shared memory access is *bank conflicts* if two or more threads access the same memory bank at the same time.

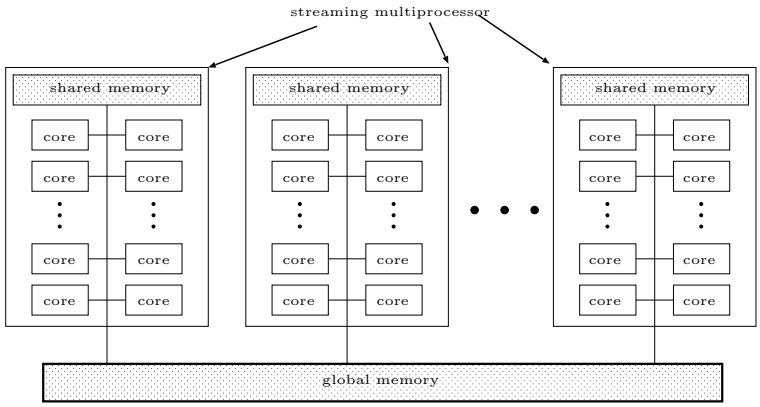


Figure 2.1: GPU hardware architecture

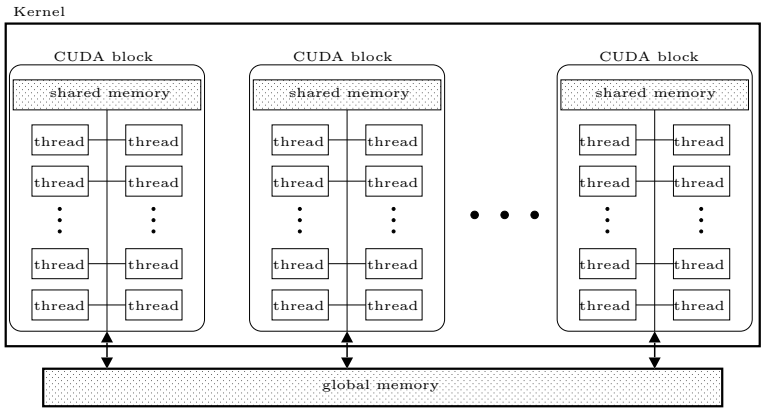


Figure 2.2: CUDA software architecture

Hence, to maximize the memory access performance, CUDA threads should access distinct memory banks to avoid bank conflicts. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform coalesced access when reading/writing to/from the global memory.

In [35], the authors have introduced two models, the *Discrete Memory Machine* (DMM) and the *Unified Memory Machine* (UMM), which reflect the essential features of the shared memory and the global memory of CUDA-enabled GPUs. Since the DMM and the UMM are promising as theoretical computing models for GPUs, several efficient algorithms on the DMM and the UMM are published in [23, 33, 34, 36]. Figure 2.3 illustrates the architectures of the DMM and the UMM. The UMM and the DMM have three parameters: (i) the number p of threads; (ii) width w; and (iii) memory access latency l. Each thread is a *Random Access Machine* (RAM) [2], which can execute fundamental operations in a time unit. Threads are executed in SIMD [14] fashion, and run on the same program and work on the different data. The p threads are partitioned into $\frac{p}{w}$ groups of w threads each, called *warp*. The $\frac{p}{w}$ warps are dispatched for the memory access in turn, and w threads in a dispatched warp send memory access requests to the memory banks (MBs)

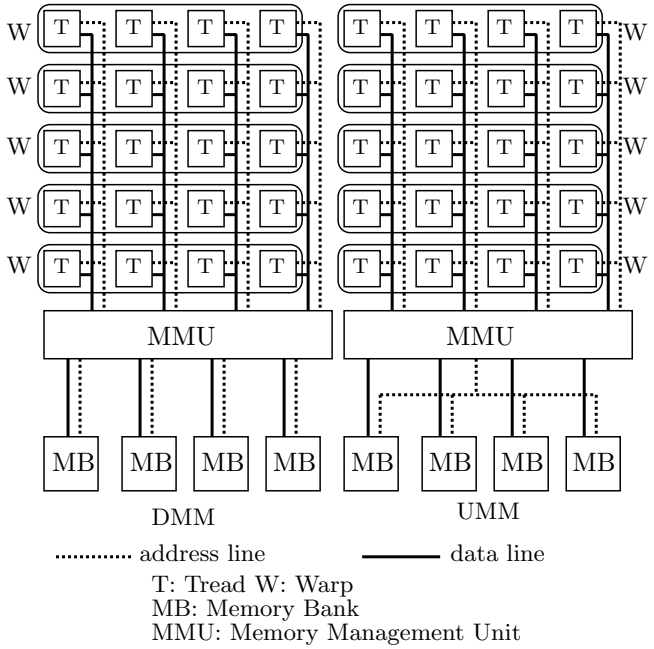


Figure 2.3: The architectures of the DMM and the UMM with width $w = 4$.

through the memory management unit (MMU). The MMU can be viewed as a multistage interconnection network in which memory access requests are moved to destination memory banks in a pipeline fashion. MBs constitute a single address space of the memory. A single address space of the memory is mapped to the MBs in an interleaved way such that a word of data of address i is stored in the $(i \bmod w)$-th bank, where w is the number of MBs. The main difference of the two architectures is the connection of the address line between the MMU and the MBs, which can transfer an address value. In the DMM, the address lines connect the MBs and the MMU separately, while a single address line from the MMU is connected to the MBs in the UMM. Hence, in the UMM, the same address value is broadcast to every MB, and the same address of the MBs can be accessed in each time unit. On the other hand, different addresses of the MBs can be accessed in the DMM. Since the memory access of the UMM is more restricted than that of the DMM, the UMM is less powerful than the DMM. In this work we consider the UMM. As a consequence, the implementations resort only to global memory access.

On evaluating the performance of algorithms on the UMM, four parameters are considered: (i) the size n of the input; (ii) the number p of threads; (iii) the width w; and (iv) the latency l of the memory access. The width w is the number of the MBs as well as the number of threads in a warp. The latency l is the number of time units to complete the memory access. In [34], we have shown that the prefix-sums of n numbers can be computed in $O(\frac{n}{w} + \frac{nl}{p} + l \log n)$ time units. Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. For example, the prefix-sums of an array b of size n can be computed by executing $b[i] \leftarrow b[i] + b[i-1]$ for all i $(1 \leq i \leq n-1)$ in turn. This prefix-sum algorithm is oblivious because the address accessed at each time unit is independent

of the values stored in b. The readers may think that the oblivious memory access is too restricted and that most useful algorithms are not oblivious. However, many important and complicated tasks, including matrix computation, signal processing, sorting, dynamic programming, and encryption/decryption, can be performed by oblivious sequential algorithms.

As defined in [49], the *bulk computation* of a sequential algorithm is to execute it for many different inputs in turn or in parallel. For example, suppose that we have p arrays, $b_0, b_1, \ldots b_{p-1}$, of size n each. We can compute the prefix-sums of each b_j $(0 \leq j \leq p-1)$ by executing the prefix-sum algorithm on a single CPU in turn or concurrently on a parallel computer. The bulk computation has many applications. Indeed, the conventional FFT algorithm [9] for n points, running in $O(n \log n)$ time, is oblivious. In practical signal processing, an input stream is equally partitioned into many blocks, and the FFT algorithm is executed for each block in turn or concurrently. This is exactly the bulk computation of the FFT algorithm. Many works have been devoted to develop and implement parallel algorithms for a single input [21]. A single-input, efficient GPU implementation for the bitonic sort has been presented in [26]. An efficient GPU implementation of the Floyd-Warshall algorithm for a single, large-input graph, has been considered in [10, 24]. However, to the best of our knowledge, there is no efficient GPU implementation for the bulk computation of the above algorithms in the literature.

Chapter 3

Computing of All-Pairs Shortest Paths of a Graph

3.1 The Floyd-Warshall algorithm and The Blocked Floyd-Warshall Algorithm

In this section, we introduce Floyd-Warshall [9, 13, 54] and Blocked Floyd-Warshall Algorithm [52].

Suppose that an $n \times n$ array D is initialized by the adjacency matrix A of an input directed graph G. We describe the (Parallel) Floyd-Warshall algorithm in Algorithm 1.

Algorithm 1 (Parallel) Floyd-Warshall algorithm

```
for k ← 0 to n − 1 do
    for i ← 0 to n − 1 do in parallel
        for j ← 0 to n − 1 do in parallel
            D[i][j] ← min{D[i][j], D[i][k] + D[k][j]}
        end for
    end for
end for
```

For the for-loop with counter k, the Floyd-Warshall algorithm updates the value of each $D[i][j]$ so as to store the length of a shortest path from node i to j with intermediate nodes 0, 1, ..., k in $D[i][j]$. Hence, after it evaluates $\min\{D[i][j], D[i][k] + D[k][j]\}$, the value of $D[i][k] + D[k][j]$ is the length of a shortest path from node i to j via node k with intermediate nodes 0, 1, ..., $k-1$. Also, $D[i][j]$ is stored the length of a shortest path from node i to j with intermediate nodes 0, 1, ..., $k-1$. Thus, the algorithm computes the distance matrix D correctly in $O(n^3)$ time. We also can execute for-loops with counter i and j in parallel.

The Blocked Floyd-Warshall Algorithm [52] is a modification of the Floyd-Warshall algorithm. This algorithm inputs $G = (V, E)$, the distance $D : V \times V \to R$ and a positive integer W, and is similar to the

Floyd-Warshall Algorithm. It divides D into $\frac{n}{W} \times \frac{n}{W}$ tiles, of size $W \times W$, each. It repeats computation of tiles. Let $D_{I,J}$ denote a tile in the I-th row and J-th column. The Blocked Floyd-Warshall Algorithm is described in Algorithm 2. To update the values of tile P of size $W \times W$, it calls procedure $T(P, Q, R)$ for tiles P, Q, and R.

Algorithm 2 Blocked Floyd-Warshall algorithm

```
for Z ← 0 to n/W − 1 do
    T(D_{Z,Z}, D_{Z,Z}, D_{Z,Z})                ▷ Computation of a pivot tile D_{Z,Z}
    for X ← 0 to n/W − 1 and X ≠ Z do
        T(D_{X,Z}, D_{X,Z}, D_{Z,Z})            ▷ Computation of a pivot column tile D_{X,Z}
    end for
    for Y ← 0 to n/W − 1 and Y ≠ Z do
        T(D_{Z,Y}, D_{Z,Y}, D_{Z,Z})            ▷ Computation of a pivot row tile D_{Z,Y}
    end for
    for X ← 0 to n/W − 1 and X ≠ Z do
        for Y ← 0 to n/W − 1 and Y ≠ Z do
            T(D_{X,Y}, D_{X,Z}, D_{Z,Y})        ▷ Computation of a non-pivot tile D_{X,Y}
        end for
    end for
end for
procedure T(P, Q, R)                            ▷ /* P, Q and R are W × W tiles. */
    for k ← 0 to W − 1 do
        for i ← 0 to W − 1 do
            for j ← 0 to W − 1 do
                P[i][j] ← min{P[i][j], Q[i][k] + R[k][j]}
            end for
        end for
    end for
end procedure
```

Suppose that $Z = c$. Tile $D_{c,c}$ is called a *pivot tile*, and $D_{I,c}$ with $I \neq c$ ($D_{c,J}$ with $J \neq c$) is called a *pivot column tile* (*pivot row tile*, respectively). Also $D_{I,J}$ with $I \neq c$ and $J \neq c$ is called a *non-pivot tile*. Figure 3.1 illustrates pivot tiles, pivot column tiles, pivot row tiles and non-pivot tiles of Blocked Floyd-Warshall Algorithm when $\frac{n}{W} = 4$. For example, $D_{0,0}$ is a pivot tile if $Z = 0$, and it is a non-pivot tile if $Z = 1$.

Each execution of the for-loop with counter Z updates D such that each $D[i][j]$ stores the length of a shortest path from node i to j with intermediate nodes 0, 1, ..., $WZ - 1$. Thus, the Blocked Floyd-Warshall algorithm simulates the Floyd-Warshall algorithm and computes the distance matrix D correctly.

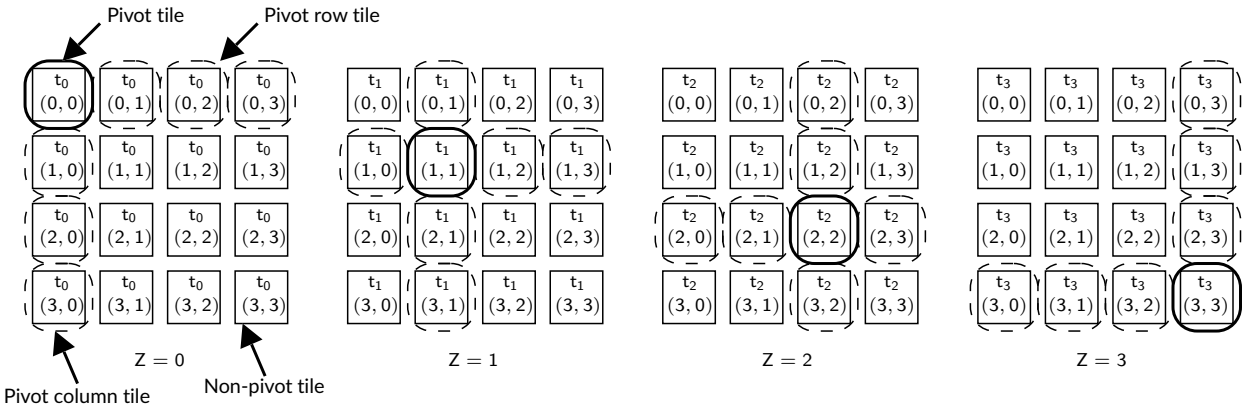


Figure 3.1: Pivot tiles, pivot column tiles, pivot row tiles and non-pivot tiles of Blocked Floyd-Warshall Algorithm when $\frac{n}{W} = 4$. Note that $t_Z(X, Y)$ is a task to execute procedure $T(D_{X,Y}, D_{*,*}, D_{*,*})$, for any loop counter Z.

Let $t_k(X, Y)$ denote *a task* to execute procedure $T(D_{X,Y}, D_{*,*}, D_{*,*})$, which updates the values of $D_{X,Y}$ when the value of loop counter Z is k. In Figure 3.1, we illustrate the computation performed by Blocked Floyd-Warshall algorithm for $\frac{n}{W} = 4$. The computation cost of procedure T for tiles of size $W \times W$ is $O(W^3)$. Thus, that of the Blocked Floyd-Warshall algorithm is $\frac{n}{W}(1 + 2(\frac{n}{W} - 1) + (\frac{n}{W} - 1)^2) \cdot O(W^3) = O(n^3)$, which is the same as the Floyd-Warshall algorithm.

3.2 GPU implementations of the Blocked Floyd-Warshall algorithm

This section first explains the implementation of the computation of procedure $T(P, Q, R)$ by a CUDA block. We then go on to explain previously published implementation [27] that we call *the multiple kernel implementation*. Finally, we present our implementation called *the single kernel implementation*.

To explain the GPU implementations, we introduce memories in the NVIDIA GPUs. CUDA uses two types of memories in the NVIDIA GPUs: *the shared memory* and *the global memory* [41]. The shared memory is an extremely fast on-chip memory with lower capacity, say, 16-48 Kbytes. The global memory is implemented as an off-chip DRAM, and thus, it has large capacity, say, 1.5-6 Gbytes, but its access latency is very long. The efficient usage of the shared memory and the global memory is a key for CUDA developers to accelerate applications using GPUs. In particular, we need to consider *the coalescing* of the global memory access [40]. To maximize the bandwidth between the GPU and the DRAM chips, the consecutive addresses of the global memory must be accessed at the same time. Thus, CUDA threads should perform the coalesced access when they access the global memory.

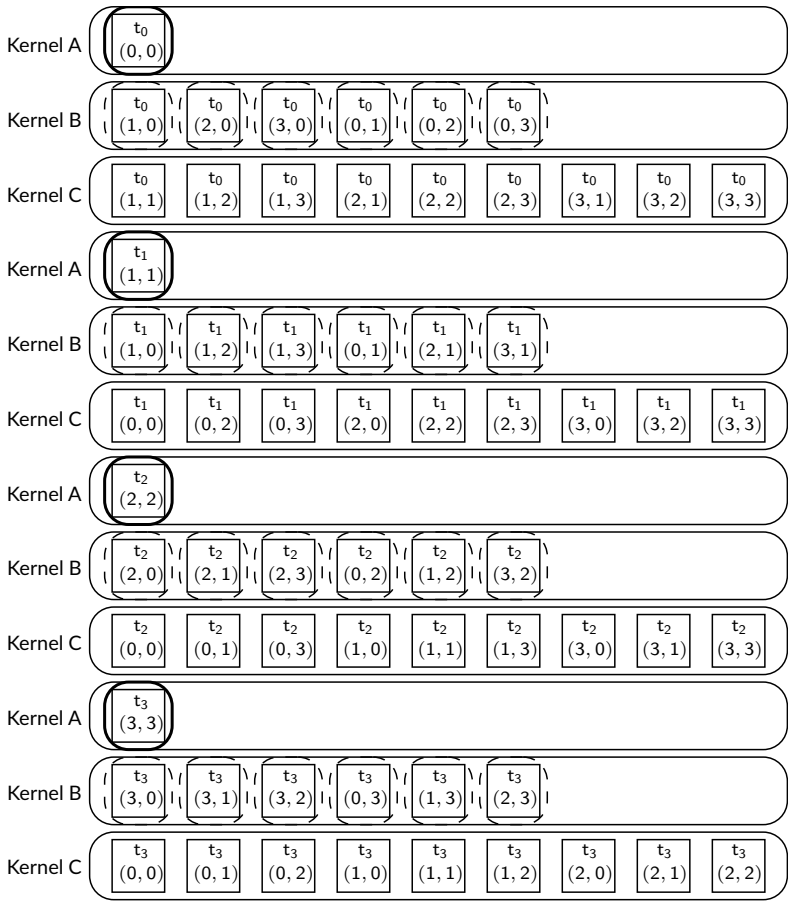


Figure 3.2: The multiple kernel implementation executes Kernel A, B and C when $\frac{n}{W} = 4$

3.2.1 The computation of $T(P, Q, R)$ by a CUDA block

We assume that $W \times W$ matrices P, Q, and R are stored in the global memory. Without loss of generality, we assume that $W = 32$. We use a CUDA block of $32c$ threads, where c ($1 \leq c \leq 32$) is an integer parameter which can be selected to maximize the performance. The computation of procedure $T(P, Q, R)$ can be done in three steps.

Step 1 Copy P, Q, and R in the global memory to the shared memory.

Step 2 Call procedure $T(P, Q, R)$.

Step 3 Copy the resulting values of P back to the global memory.

The space to load the values of P, Q and R are arranged in the shared memory or registers of $32c$ threads. We describe the detail as follows:

(S1) The case that P is a pivot tile. To call procedure $T(D_{Z,Z}, D_{Z,Z}, D_{Z,Z})$, the values of $D_{Z,Z}$ are stored in memory of size $W \times W$, allocated on the shared memory.

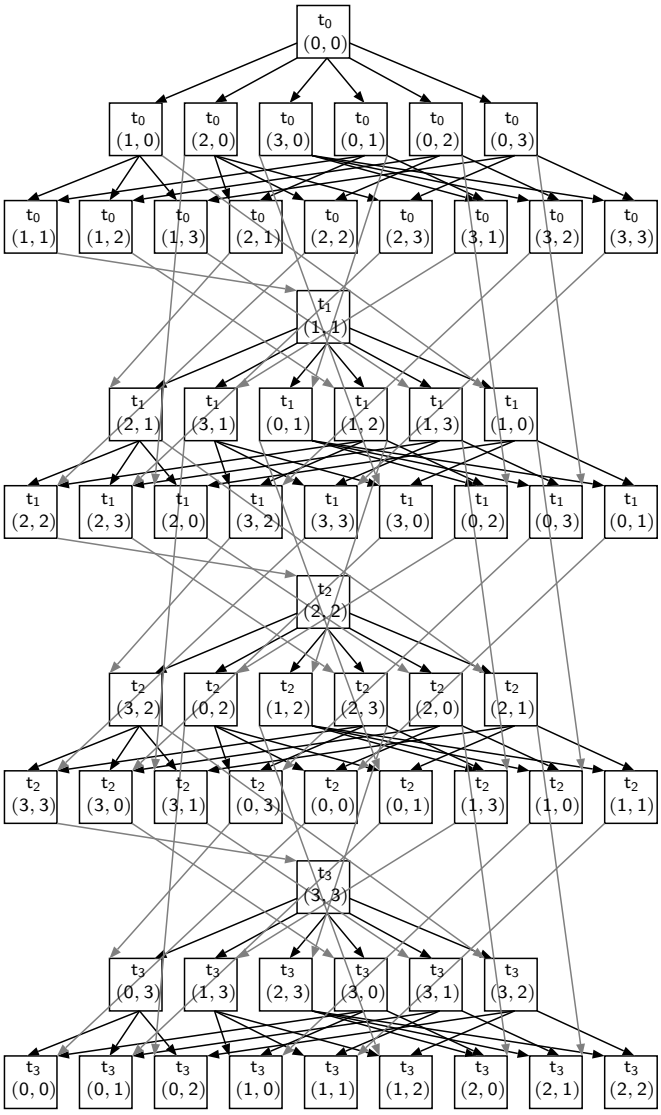


Figure 3.3: The task graph of the Blocked Floyd-Warshall algorithm when $\frac{n}{W} = 4$

(S2) The case that P is a pivot column or row tile. To call procedure $T(D_{X,Z}, D_{X,Z}, D_{Z,Z})$ and procedure $T(D_{Z,Y}, D_{Z,Y}, D_{Z,Z})$, the values of $D_{X,Z}$ (or $D_{Z,Y}$) and $D_{Z,Z}$ are stored in memory of size $2 \cdot W \times W$, allocated on the shared memory.

(S3) The case that P is a non-pivot tile. To call procedure $T(D_{X,Y}, D_{X,Z}, D_{Z,Y})$, the values of $D_{X,Z}$ and $D_{Z,Y}$ are stored in memory of size $2 \cdot W \times W$, allocated on the shared memory. The values of $D_{X,Y}$ are stored in registers of $32c$ threads to accelerate the computation.

The Parallel Floyd-Warshall algorithm is executed using $32c$ threads such that each thread works for updating $\frac{W \times W}{32c} = \frac{32 \times 32}{32c} = \frac{32}{c}$ elements in X. Since the resulting value of an element in P is never used for computing the other elements in P, P can be arranged in registers of threads.

The values of D in each tile are allocated in consecutive addresses on the implementations. When threads

access $W \times W = 32 \times 32$ values in P, the implementations can perform coalesced access.

3.2.2 Multiple kernel implementation on the GPU

We assume that the adjacency matrix A of an input directed graph is stored in the global memory and the distance matrix D must be computed and written in the global memory.

We introduce previously published GPU implementation [27].

For Blocked Floyd-Warshall Algorithm, when the values in a tile is computed, the tile depends on other tiles. It is hard to implement computation of depending tiles with single kernel. The implementation has three CUDA kernels: (1) Kernel A is to compute the values in a pivot tile, that is, to call procedure $T(D_{Z,Z}, D_{Z,Z}, D_{Z,Z})$ using (S1). (2) Kernel B is to do in pivot row tiles and pivot column tiles, that is, to call procedure $T(D_{X,Z}, D_{X,Z}, D_{Z,Z})$ or $T(D_{Z,Y}, D_{Z,Y}, D_{Z,Z})$ using (S2). (3) Kernel C is to do in non-pivot tiles, that is, to call procedure $T(D_{X,Y}, D_{X,Z}, D_{Z,Y})$ using (S3). It invokes these CUDA kernels for each loop Z. Figure 3.2 illustrates tiles computed by each kernel when $\frac{n}{W} = 4$. The number of CUDA blocks of Kernel A is one because there is just one pivot tile. Because the number of pivot row tiles and pivot column tiles is $\frac{n}{W} - 1$, The number of CUDA blocks of Kernel B or C is $2(\frac{n}{W} - 1)$ or $(\frac{n}{W} - 1)^2$, respectively. For the example in Figure 3.2, the number of CUDA blocks of Kernel A, B and C are 1, $2(\frac{n}{W} - 1) = 6$ and $(\frac{n}{W} - 1)^2 = 9$, because of $\frac{n}{W} = 4$.

If the Floyd-Warshall algorithm is implemented on the GPU as it is, then a kernel is invoked for each value of k and so n kernel calls are necessary. Also, a lot of memory access operations are performed for the global memory. Thus, the Blocked Floyd-Warshall algorithm should be used to reduce the number of kernel calls. Also, procedure $T(P, Q, R)$ can be implemented using registers or the shared memory, and the global memory access can be reduced.

The multiple kernel implementation invokes three kernel call for for-loop with counter Z. Thus, it invokes $3\frac{n}{W}$ kernel calls totally. Thus, the overhead of separated kernel calls is quite large. Also, each Kernel A invokes only one CUDA block and so parallelism is quite low.

We introduce the task graph of the Blocked Floyd-Warshall algorithm, as shown in Figure 3.3. Each node of the task graph represents a task of computing values. Each direct edge of the task graph does dependency of tasks. For any directed edges, we must start the task corresponding to its end node, after we complete the task, corresponding to its start node. For the task graph illustrated in Figure 3.3, $t_0(1, 0)$ must be started after $t_0(0, 0)$ is completed, because the graph has a direct edge from $t_0(0, 0)$ to $t_0(1, 0)$. Similarly, $t_0(1, 1)$ must be started after $t_0(1, 0)$ is completed. Therefore, $t_0(1, 1)$ must be started after both $t_0(0, 1)$ and $t_0(1, 0)$ are completed because the graph has direct edges from $t_0(0, 1)$ and $t_0(1, 0)$ to $t_0(1, 1)$.

3.2.3 Our single kernel implementation on the GPU

We explain our implementation for the Blocked Floyd-Warshall algorithm on the GPU. It uses the Single Kernel Soft Synchronization (SKSS) technique [17]. The idea of the SKSS is to use a dependency graph of tasks such that a directed edge from task u to v is drawn if task u must be completed before task v

is processed. We can draw a task graph for $(\frac{n}{W})^3$ tasks $t_K(I,J)$ $(0 \leq I,J,K \leq \frac{n}{W}-1)$ of the Blocked Floyd-Warshall algorithm. For example, we assume that $Z=c$, $I \neq c$, $J \neq c$. Procedure T($D_{I,c}$, $D_{I,c}$, $D_{c,c}$) or T($D_{c,J}$, $D_{c,J}$, $D_{c,c}$) must be executed after T($D_{c,c}$, $D_{c,c}$, $D_{c,c}$) are completed. Procedure T($D_{I,J}$, $D_{I,c}$, $D_{c,J}$) must be done after both T($D_{I,c}$, $D_{I,c}$, $D_{c,c}$) and T($D_{c,J}$, $D_{c,J}$, $D_{c,c}$) are completed. More specifically, the task graph includes directed edges such that

- $(t_{K-1}(I,J), t_K(I,J))$ for all I, J, and K,
- $(t_K(K,K), t_K(I,K))$ and $(t_K(K,K), t_K(K,J))$,
- $(t_K(K,J), t_K(I,J))$ and $(t_K(I,K), t_K(I,J))$

Note that $t_K(K,K)$, $t_K(K,J)$, $t_K(I,K)$ or $t_K(I,J)$ is to compute values in pivot tiles, pivot column tiles, pivot row tiles or non-pivot tiles.

As we can see in Figure 3.3, for all edges $(t_{K'}(I',J'), t_K(I,J))$ in the task graph, task $t_{K'}(I',J')$ must be completed before $t_K(I,J)$ starts, because the resulting value of $D_{I',J'}$ in iteration K' is used in $D_{I,J}$ in iteration K. Clearly, every node is connected with at most three incoming edges.

We assign unique IDs from 0 to $(\frac{n}{W})^3-1$ to all $(\frac{n}{W})^3$ tasks by the topological sort of the task graph. Figure 3.4 shows unique IDs of the task graph in Figure 3.3. More specifically, for all directed edges $(t_{K'}(I',J'), t_K(I,J))$ of the task graph, $d_{K'}(I',J') < d_K(I,J)$ holds, where $d_K(I,J)$ denotes the assigned ID of each task $t_K(I,J)$. In our implementation using the SKSS technique, CUDA blocks are assigned to tasks in the order of task IDs. For this purpose, we use a zero-initialized global counter `x` in the global memory. We invoke enough CUDA blocks and the first thread of every CUDA block increments `x` using CUDA atomic function `atomicAdd(&x,1)`, which exclusively increments the value of `x` by 1 and returns the value of `x` before increment. Clearly, the first threads of CUDA blocks succeeding in executing atomicAdd receive unique return values 0, 1, 2, ... in turn. If the first thread receives the value greater than $(\frac{n}{W})^3-1$, then the CUDA block terminates. Otherwise, the CUDA block performs a task with IDs equal to the return value. When the CUDA block performs the task, it must check if all previous tasks specified by directed edges of the task graph are completed. For example, in Figure 3.3, if the return value is $d_1(2,0)$, then the CUDA block works for task $t_1(2,0)$. It first checks if each of three tasks $t_0(2,0)$, $t_1(1,0)$, and $t_1(2,1)$ is completed. When one of the task is finished, then it copies the resulting values of the task in the global memory. For example, if $t_0(2,0)$ is completed, the resulting values of $D_{2,0}$ are read. After all the resulting values of $D_{2,0}$, $D_{1,0}$, and $D_{2,1}$ are read, the Parallel Floyd-Warshall algorithm is executed for task $t_1(2,0)$ to compute $T(D_{2,0}, D_{1,0}, D_{2,1})$.

Note that for an directed edge $(t_{K'}(I',J'), t_K(I,J))$ of a task graph, the difference of unique IDs, $t_K(I,J) - t_{K'}(I',J')$, should be as large as possible. If it is larger, then more tasks are executed between task $t_{K'}(I',J')$ and $t_K(I,J)$ and task $t_{K'}(I',J')$ is completed with higher probability when task $t_K(I,J)$ starts.

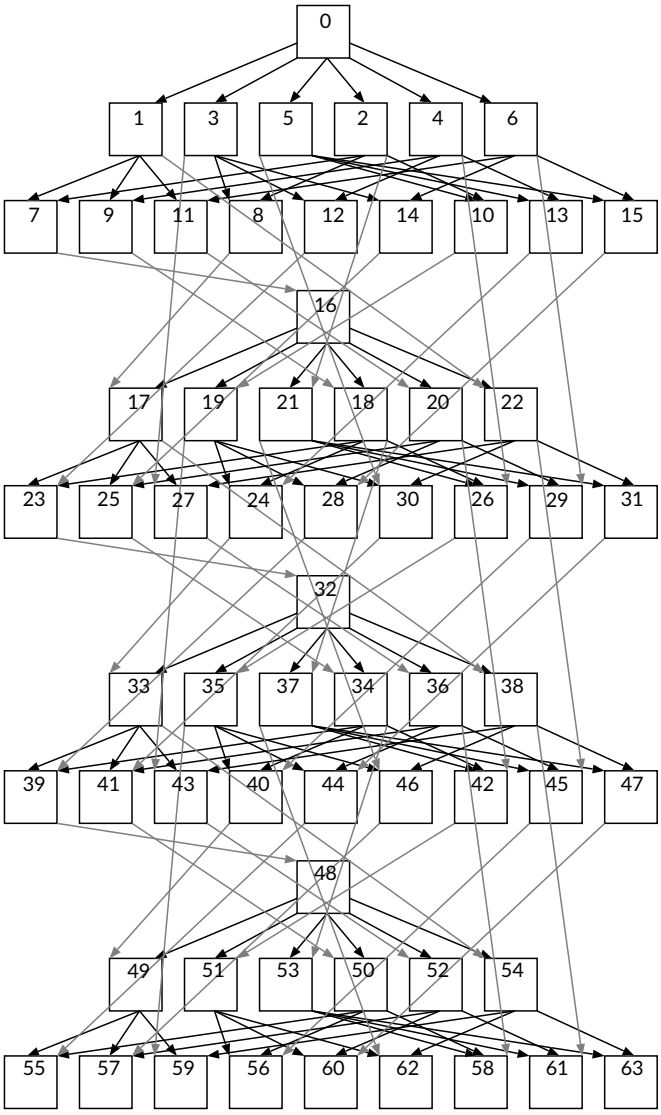


Figure 3.4: Unique IDs assigned by the topological sort when $\frac{n}{W} = 4$

3.2.4 Acceleration by using SIMD functions

In this subsection, we present SIMD (Single Instruction, Multiple Data) functions so as to accelerate computation of addition and minimum selection using the Floyd-Warshall algorithm.

We assume that the length of all edges of graphs is 1. If the length of a shortest path between two nodes is less than 2^b, each value of the distance matrix D can be represented in at most b bits, during the Floyd-Warshall algorithm are executing. Clearly, we can vectorize $\lfloor \frac{32}{b} \rfloor$ values by a 32-bit integer.

Without loss of generality, we set $b = 4$. We can represent values of the distance matrix D as $b = 4$ bits and vectorize 8 values by a 32-bit integer. For example, $D[i][j], D[i][j+1], \ldots, D[i][j+7]$ can be vectorized by a 32-bit integer. If each $D[i][j]$ is stored in a 32-bit integer, we need 8 addition and 8 minimum selection for $D[i][j'] \leftarrow \min\{D[i][k] + D[k][j'], D[i][j']\}$, where $j \leq j' \leq j+7$, illustrated as Figure 3.5 (a). However,

by using saturating addition, we can exchange 8 addition and 8 minimum selection into one saturating addition and one minimum selection, shown in Figure 3.5 (b). Note that saturating addition is defined by $\min\{D[i][k] + D[k][J], 2^b - 1\}$. For example, we assume that 4-bit integers $a[i]$, $b[i]$ and $c[i]$, $0 \leq i \leq 7$, are defined as follows:

$$a[8] = \{1, 14, 13, 12, 2, 2, 2, 2\};$$
$$b[8] = \{1, 15, 1, 2, 14, 4, 5, 6\};$$
$$c[8] = \{15, 15, 1, 10, 1, 10, 1, 11\};$$

We will compute $d[i] = \min\{a[i] + b[i], c[i]\}$, $0 \leq i \leq 7$, by using saturating addition. It is clear that $d[8] = \{2, 15, 1, 10, 1, 6, 1, 8\}$.

When $a[]$, $b[]$ and $c[]$ are vectorized by 32-bit integers x, y and z, we can get the integers, represented as a binary number, as follows:

$$x = 0001\ 1110\ 1101\ 1100\ 0010\ 0010\ 0010\ 0010,$$
$$y = 0001\ 1111\ 0001\ 0010\ 1110\ 0100\ 0101\ 0110, \text{ and}$$
$$z = 1111\ 1111\ 0001\ 1010\ 0001\ 1010\ 0001\ 1011.$$

Computation of saturating addition $s = x + y$ leads to $s = 0010\ 1111\ 1110\ 1110\ 1111\ 0110\ 0111\ 1000$.

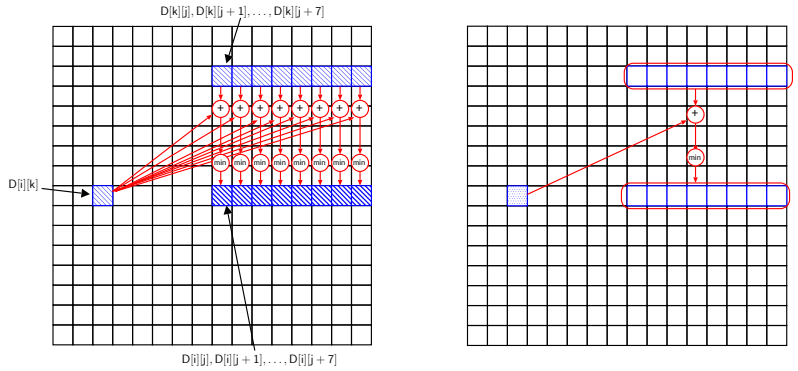
Each of eight 4-bit numbers in s satisfies $\min\{a[i]+b[i], 2^4-1\}$, $0 \leq i \leq 7$. For example, since $a[1]+b[1] = 29 > 2^4 - 1$, the resulting value of the saturating addition is $1111 (= 2^4 - 1)$. After that, by computing $t = \min\{s = x + y, z\}$, we can get the resulting value

$$t = 0010\ 1111\ 0001\ 1010\ 0001\ 0110\ 0001\ 1000.$$

We suppose that eight 4-bit unsigned integers are vectorized by `x` or `y`.

We present saturating addition function `addus8()` and minimum selection function `minus8()` for 4-bit unsigned integers as follows:

```
unsigned addus8(unsigned x,unsigned y)
{
    s = (x & 0x77777777) + (y & 0x77777777);    z = ((x & y) | (y & s) | (
        s & x)) & 0x88888888;
    m = (z >> 3) * 15;
    return ((((x^y) & 0x88888888 )^s) | m);
}
unsigned minus8(unsigned x,unsigned y)
{
    d = (x | 0x88888888) - (y & 0x77777777);    ma = (((~(x^y)) & d) | ((x
        ^y) & x)) & 0x88888888;
    mb = (ma >> 3) * 15;
    return( (x & (~mb)) | ( y & mb ) );
}
```



(a) (b)

Figure 3.5: (a) The computation of $D[i][j'] \leftarrow \min\{D[i][j'], D[i][k] + D[k][j']\}$ for $D[i][j']$, $j \leq j' \leq j+7$. (b) We can compute $D[i][j']$, $j \leq j' \leq j+7$, by one saturating addition and one minimum selection, when $D[i][j]$ is represented as 4 bits and $D[i][j']$, $j \leq j' \leq j+7$, are vectorized by a 32-bit integer.

When two 32-bit integers are input, SIMD functions `addus8()` or `minus8()` returns a resulting 32-bit integer by saturating addition or minimum selection, respectively.

Clearly, we can use these SIMD functions for any other implementations. We can propose single or multiple implementations using these SIMD functions. Also, CUDA has SIMD intrinsic functions for 32-bit integers with $b = 8, 16$. We have written SIMD functions with $b = 4$, and have applied them to our implementation.

3.3 Bulk computation of the Blocked Floyd-Warshall algorithm

Suppose that we need to execute the Blocked Floyd-Warshall algorithm for many directed graphs. We will show how the Blocked Floyd-Warshall algorithm is executed for multiple directed graphs in parallel.

Let m be the number of directed graphs with n nodes each. In the multiple kernel implementation, we can simply invoke $3\frac{n}{W}$ kernels in turn. Recall that for a single directed graph, Kernel A, B, and C invokes 1, $2(\frac{n}{W}-1)$, and $(\frac{n}{W}-1)^2$ CUDA blocks, respectively. For bulk computation, Kernel A is to compute the values in pivot tiles of m directed graphs. So, it invokes m CUDA blocks. Similarly, Kernel B and C is to do in pivot column tiles and pivot row tiles of m directed graphs. They invoke $2m(\frac{n}{W}-1)$ and $m(\frac{n}{W}-1)^2$ CUDA blocks, respectively. Given m directed graphs, we can execute this algorithm at the same time.

Bulk computation for m directed graphs of size n can be done by one kernel call using our single kernel implementation. For this purpose, we assign unique IDs to $m(\frac{n}{W})^3$ tasks for m directed graphs as follows. Let $d_K(I, J)$ denote a unique ID in the range $[0, (\frac{n}{W})^3 - 1]$ assigned for task $t_K(I, J)$ of a directed graph. We assign an ID $m \cdot d_K(I, J) + g$ to a task $t_K(I, J)$ of the g-th $(0 \leq g \leq m-1)$ directed graph. For example, suppose that we have m directed graphs, each of which has the task graph illustrated in Figure 3.3. Clearly, m tasks $t_0(0, 0)$ are assigned unique IDs 0, 1, ..., $m-1$, respectively. Also, m tasks $t_1(1, 1)$ are assigned unique IDs m, $m+1$, ..., $2m-1$, respectively. Thus, the difference of IDs of $t_0(0, 0)$ and $t_1(1, 1)$ for the

Table 3.1: The running time (in milliseconds) of our single and the multiple kernel implementation

	$n = 256$	$n = 512$	$n =$ 1024	$n =$ 2048	$n =$ 4096	$n =$ 8192	$n =$ 16384	$n =$ 32768
# of threads	Multiple kernel implementation [27]							
64	0.545	1.090	2.863	29.37	65.5	454	3508	27852
128	0.336	0.671	1.970	8.15	**52.6**	397	3125	24860
256	0.263	0.488	1.556	**7.21**	53.6	**371**	**2924**	**23433**
512	**0.244**	**0.463**	**1.549**	8.43	54.0	408	3209	25942
1024	0.261	0.537	1.860	9.53	66.1	501	3935	31252
# of threads	Our single kernel implementation							
64	0.487	1.185	2.520	13.45	67.2	482	3977	32123
128	0.256	0.600	1.547	7.61	50.0	365	2955	23699
256	**0.187**	**0.406**	**1.327**	**6.84**	**47.4**	**326**	**2594**	**20722**
512	0.203	0.400	1.541	8.20	56.9	399	3275	26524
1024	0.293	0.543	2.168	12.15	78.2	606	4982	40515
Speedup	1.31x	1.16x	1.17x	1.05x	1.11x	1.14x	1.13x	1.13x

same graph is $d_1(1,1) - d_0(0,0) = m$. If m is large, then task $t_0(0,0)$ is completed with high probability when $t_1(1,1)$ starts. Thus, we can get better performance for the bulk computation. For example, Figure 3.6 shows unique IDs within $[0, (\frac{n}{W})^3 - 1 = 255]$ for 4 graphs and $\frac{n}{W} = 4$.

3.4 Experiment results

In this paper, we use NVIDIA Tesla V100 GPU for the experiment. We show average of the running time in all tables.

We have implemented our single kernel implementation and the multiple kernel implementation published previously [27] for the Blocked Floyd-Warshall algorithm. Note that the computation of the Blocked Floyd-Warshall algorithm is oblivious in the sense that the same computation is performed for any directed graphs. So, the running time only depends on the number of nodes.

Tables 3.1 and 3.2 show the running time of implementations for graphs with $n = 256, 512, \ldots, 32768$ nodes. To execute procedure $T(P, Q, R)$, we use CUDA blocks with 64, 128, ..., 1024 threads. We use the same CUDA program to execute procedure $T(P, Q, R)$ for both our single kernel implementation and multiple kernel implementation. In the tables, the running time is highlighted when it is the best for each n. From the tables, we can see that the running time is the best when 256 or 512 threads per block are used for almost all number of nodes. It also shows the speed-up of the best running time of the single kernel implementation over the multiple kernel implementation. We can see that our single kernel implementation runs 1.05-1.31 times faster than the multiple kernel implementation, because the number of kernel calls in

Table 3.2: The running time (in milliseconds) of our single and the multiple kernel implementation with SIMD functions

	$n = 256$	$n = 512$	$n =$ 1024	$n =$ 2048	$n =$ 4096	$n =$ 8192	$n =$ 16384	$n =$ 32768
# of threads	Multiple kernel implementation [27] with SIMD functions							
64	0.844	1.680	3.359	8.86	64.3	285	1893	15655
128	0.479	0.963	1.800	6.26	71.9	337	2282	17583
256	0.270	0.637	1.198	4.52	43.1	251	1838	14415
512	**0.183**	**0.419**	**0.949**	**4.21**	31.8	213	1684	13450
1024	–	–	–	–	**26.8**	**187**	**1440**	**11531**
# of threads	Our single kernel implementation with SIMD functions							
64	0.831	1.660	4.618	13.04	65.6	432	3154	27664
128	0.422	0.851	2.008	5.48	35.7	449	3325	26830
256	0.213	0.446	1.095	4.15	27.6	223	1673	13112
512	**0.164**	**0.328**	**0.837**	**4.08**	**25.9**	**186**	**1439**	**11402**
1024	–	–	–	–	–	194	1522	12084
Speedup	1.12x	1.28x	1.13x	1.03x	1.04x	1.00x	1.00x	1.01x

Table 3.3: The data transfer time (in milliseconds) of our implementations

	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
Single kernel	0.0623	0.1869	0.683	2.667
Single kernel with SIMD functions	0.0280	0.0430	0.105	0.355
	$n = 4096$	$n = 8192$	$n = 16384$	$n = 32768$
Single kernel	10.60	42.28	169.0	675.6
Single kernel with SIMD functions	1.35	5.31	21.2	84.5

our implementation is smaller than that of the multiple kernel implementation. From Table 3.2, we can see that our single kernel implementation using SIMD functions with $b = 4$ runs 1.00-1.28 times faster than the multiple kernel implementation.

For the implementations without SIMD functions, we set $W = 32$, that is, the size of each tile is 32×32, because of capacity of the shared memory. For the single or multiple kernel implementation with SIMD functions, we define the size of each tiles by 8×64 if $n \leq 4096$ or $n \leq 2048$, respectively. Otherwise we define it by 16×128. They run faster than any other sizes. If the size of each tile is 8×64, the tile includes $8 \times 64 = 512$ values. Each thread of every CUDA block performs updating of values. We cannot use CUDA blocks with the number of threads, which exceeds the number of values. Hence, for single or multiple kernel implementation with SIMD functions, the running time is invalid in Table 3.2, when the number of threads is 1024 and $n \leq 4096$ or $n \leq 2048$, respectively.

$K = 0$

Graph G_0			
0	5	7	9
4	28	30	32
6	29	48	50
8	31	49	60

Graph G_1			
1	11	13	15
10	33	35	37
12	34	51	53
14	36	52	61

Graph G_2			
2	17	19	21
16	38	40	42
18	39	54	56
20	41	55	62

Graph G_3			
3	23	25	27
22	43	45	47
24	44	57	59
26	46	58	63

$K = 1$

Graph G_0			
124	72	95	113
73	64	69	71
96	68	92	94
114	70	93	112

Graph G_1			
125	78	100	116
79	65	75	77
101	74	97	99
117	76	98	115

Graph G_2			
126	84	105	119
85	66	81	83
106	80	102	104
120	82	103	118

Graph G_3			
127	90	110	122
91	67	87	89
111	86	107	109
123	88	108	121

$K = 2$

Graph G_0			
176	178	134	157
177	188	136	159
135	137	128	133
158	160	132	156

Graph G_1			
179	181	140	162
180	189	142	164
141	143	129	139
163	165	138	161

Graph G_2			
182	184	146	167
183	190	148	169
147	149	130	145
168	170	144	166

Graph G_3			
185	187	152	172
186	191	154	174
153	155	131	151
173	175	150	171

$K = 3$

Graph G_0			
220	222	224	196
221	240	242	198
223	241	252	200
197	199	201	192

Graph G_1			
225	227	229	202
226	243	245	204
228	244	253	206
203	205	207	193

Graph G_2			
230	232	234	208
231	246	248	210
233	247	254	212
209	211	213	194

Graph G_3			
235	237	239	214
236	249	251	216
238	250	255	218
215	217	219	195

Figure 3.6: Unique IDs within $[0, (\frac{n}{W})^3 - 1 = 255]$ for 4 graphs and $\frac{n}{W} = 4$

Table 3.3 also shows the data transfer time for graphs with $n = 256, 512, \ldots, 32768$ nodes. For the implementations with/without SIMD functions, the data transfer time of single kernel implementation is equal to that of the multiple one, because the transferred data on both implementations are the distance matrix. The size of the distance matrix on the implementation with SIMD functions is $\frac{32}{b}$ times smaller than that of one without SIMD functions. In the implementations with SIMD functions, we set $b = 4$. That is, each value D of the distance matrix is stored in a $b = 4$-bit, and eight 4-bit unsigned integers are vectorized by a 32-bit integer. So, the data transfer time of the simple kernel implementation with SIMD functions is 8 times faster than the ones without them.

3.4.1 Efficiency of computing the diameter and the ASPL of a graph

We show the efficiency of computing the diameter and the ASPL of a graph.

When we start Breadth First Search (BFS) from a node, we can find the length of a shortest path from the node to other nodes. From this fact, we can compute the length of a shortest path of all pairs of nodes by executing BFS from each node. Clearly, its complexity is $O(n(n + |E|))$, and smaller than that of both

the Floyd-Warshall algorithm and the Blocked Floyd-Warshall algorithm where $|E|$ is the number of edges of a given graph. To evaluate our single kernel implementation using SIMD functions, we implement the method of using BFS on the GPU, called the simple implementation with BFS. Each thread of CUDA blocks executes BFS from a node.

Tables 3.4 and 3.5 show the running time of our single kernel implementation with SIMD functions and the simple implementation with BFS. As mentioned above, our single kernel implementation with SIMD functions does not depend on the number of edges. So, we copy the best running time for each n in Table 3.2 to Tables 3.4 and 3.5. For the simple implementation with BFS, we input connected graphs with n nodes and $\frac{n(n-1)}{2} \times \frac{r}{100}$ edges, where $n = 256, 512, 1024, \ldots, 32768$ and $r = 0.05$, 0.1, 0.25, 0.5, 1, 5, 10, 20, 50. For example, a graph with $n = 256$ and $r = 0.5$ has $\frac{n(n-1)}{2} \times \frac{r}{100} = 163$ edges. Because the graphs are connected, any graph with n nodes has at least $n - 1$ edges. Hence, there is no connected graph with $n = 256$ and $r = 0.5$, because of $\frac{n(n-1)}{2} \times \frac{r}{100} = 163 < n - 1 = 255$.

When almost of the graphs are input, our single kernel implementation with SIMD functions is 1.60-352.46 times faster than simple GPU implementation with BFS. When the number of edges is large, that is, ($n \leq 2048$) or ($n = 4096$ and $r \geq 0.1$) or ($n \geq 8192$ and $r \geq 0.25$), our single kernel implementation runs faster than simple implementation with BFS.

The complexity of simple implementation with BFS per node is $O(n + |E|)$. When the number of edges is large, its complexity is large and the running time of each threads is long. Hence, the running time of our implementation is smaller than that of simple implementation with BFS, and we conclude that our single kernel implementation with SIMD functions is efficient.

3.4.2 For Bulk Computation

Tables 3.6 and 3.7 shows the running time of the bulk computation of the Blocked Floyd-Warshall algorithm for $m = 8, 16, \ldots, 1024$ input graphs with $n = 128, 256, \ldots$, 2048 nodes each. For each pair of m and n, we execute both our single kernel implementation and the multiple kernel implementation for CUDA blocks with 64, 128, ..., 1024 threads each, and take the best one. For the implementations without SIMD functions, the bulk computation for 1024 graphs with 2048 nodes cannot be executed since the global memory usage exceeds the capacity of the global memory of Tesla V100 GPU. From Table 3.6, we can see that our single kernel implementation runs 1.03-1.60 times faster than the multiple kernel implementation. From Table 3.7, we can also see that our single kernel implementation with SIMD functions runs 1.01-1.89 times faster.

In general, our single kernel implementation has better advantages over the multiple kernel implementation for small graphs, because the ratio of computation cost of Kernel A in the multiple kernel implementation is large. However, if both the number of nodes and the number of graphs are small and the running time is small, then the speedup ratio is small. This is because the fixed overhead for invoking CUDA program is imposed for both implementations.

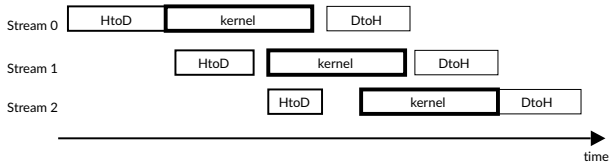


Figure 3.7: The behavior of our proposed low-latency GPU implementation, executing three streams

3.5 Low-Latency GPU Implementation for Bulk Computation

In Section 3.3, we have proposed the GPU implementations of the bulk computation of the Blocked Floyd-Warshall algorithm. Clearly the time to transfer between Host and the device increases if many graphs are given, and latency increases. We propose the low-latency implementation using the stream, which is a sequence of CUDA operations. CUDA may run operations in different streams concurrently, and may interleave them from different streams. Hence, our implementation can decrease latency by hiding the time to transfer between Host and the device. Figure 3.7 illustrates the behavior of the our proposed one, executing three streams. In the figure, "HtoD", "kernel" and "DtoH" correspond to the data transfer from Host to Device (i.e. GPU), kernel computation (i.e. bulk computation); and the data transfer from Device to Host, respectively. When m graphs are given, our implementation invokes m streams. Each stream copies the distance matrix of a graph from Host to Device, and executes the single kernel implementation with/without SIMD functions. Finally, it copies the resulting distance matrix graph to Host.

Tables 3.8 and 3.9 show the running time of the low-latency implementation and the bulk computation by our single or multiple kernel implementation with/without SIMD functions. We input $m = 8, 16, \ldots, 1024$ directed graphs with $n = 128, \ldots$, 2048 nodes each. Note that the running time of the bulk computation includes the data transfer time. When we input 1024 graphs with $n = 2048$, we cannot execute the implementations without SIMD functions because of the shortage of memory space on the GPU. Hence, we write "N/A" in Table 3.8.

From Table 3.8, the low-latency implementation runs 1.45-5.90 times faster than the multiple kernel implementation, and does 1.38-5.87 times faster than the single one. In terms of implementations with SIMD functions, from Table 3.9, the low-latency implementation with SIMD functions runs 1.11-2.43 times faster than the multiple implementation with SIMD functions, and does 1.07-2.36 times faster than the single one. When the number of graphs is large, the speed up of factor is large, and the implementation is efficient. We consider the case that 1024 graphs with $n = 2048$ are input. From Table 3.7, the single kernel implementation with SIMD functions can compute the discance matrix in 3004 ms, From Table 3.9, Low-latency implementation with SIMD functions runs in 3052 ms, and its transfer time is small. We can see that it hides the time to transfer between Host and Device. We conclude that our implementations reduce latency by hiding the time to transfer between Host and the device.

3.6 Multicore Implementations Using SIMD Functions

In previous sections, we have proposed and evaluated GPU implementations of the parallel Blocked Floyd-Warshall algorithm. In this section, we propose multicore implementations of the parallel Floyd-Warshall algorithm using SIMD intrinsics, called Intel Advanced Vector Extensions 512 (Intel AVX-512) [22]. Intel AVX-512 is an instructions set, and available on Intel Xeon Phi Processors and Skylate-X CPUs. It also provides intrinsics by using 512-bits registers, called ZMM registers. When we use Intel AVX-512, we can store sixteen 32-bits integers to one ZMM register. For two values that are vectorized by such the ZMM registers, we can also execute saturating additions and minimum selections. OpenMP is an API that supports multi-platform shared-memory parallel programming in C/C++ and Fortran [45]. By using OpenMP, we can implement the Parallel Floyd-Warshall algorithm on the multicore processors. We propose two implementations of the Parallel Floyd-Warshall algorithm, shown in Algorithm 1 on the multicore processors: One is to use our SIMD functions presented in Section 3.2.4 and the other is to use Intel AVX-512.

Table 3.10 shows the running time of multicore implementations of the parallel Blocked Floyd-Warshall algorithm on Intel Skylate-X CPU with our functions and Intel AVX-512. Intel Skylate-X CPU has 18 cores and we can use Hyper-Threading on it. We executed the implementations with $1, 2, 4, 8, 16, 18, 32, 36$ threads. We input directed graphs with $n = 512, 1024, \ldots, 4096$ nodes each. Table 3.10 (a) and (b) show the running time when each value of the distance matrix D is stored in $2^8 - 1$ and $2^{16} - 1$ bits, respectively. For the tables, we highlight the best running time among that of all threads. From Table 3.10, the implementation with Intel AVX-512 runs faster than one with our functions. We expect that this is caused by the fact that Intel AVX-512 is designed so as to maximize the capability of Intel CPU. Intel Skylate-X CPU includes Hyper-Threading Technology, and this technology enable to launch at most 36 threads simultaneously. In terms of the implementation with our functions, when $n \geq 2048$ except the case $b = 16$ and $n = 4096$, the running time of the implementation with 36 threads is the smallest for each $1, 2, 4, 8, 16, 18, 32, 36$ threads, because of Intel Skylate-X CPU. Even if $b = 16$ and $n = 4096$, the running time of the implementation with 36 threads is nearly equal to that of 18 threads. When $n \leq 1024$, the running time of the implementation with 18 threads is the smallest of all, because the fixed overhead for using OpenMP is larger than computation of the algorithm. Though the implementation with Intel AVX-512 by using multiple threads runs faster than one by using only one thread, we cannot expect the best number of threads, executing the implementation with Intel AVX-512. because the running time depends on performance of Intel AVX-512.

Table 3.4: The running time (in milliseconds) of our single kernel implementation with SIMD functions and simple implementation with BFS when $n = 256, 512, \ldots, 4096$. Note that # of edges is r% edges of clique with n nodes.

		$r =$ 0.05	$r =$ 0.1	$r =$ 0.25	$r =$ 0.5	$r = 1$	$r = 5$	$r = 10$	$r = 20$	$r = 50$
$n = 256$		# of edges								
		16	32	81	163	326	1632	3264	6528	16320
	Our Ker.	Simple implementation with BFS								
time	0.164	N/A	N/A	N/A	N/A	0.402	1.151	1.726	3.799	7.727
Speedup		N/A	N/A	N/A	N/A	2.46x	7.04x	10.55x	23.23x	47.24x
$n = 512$		# of edges								
		65	130	327	654	1308	6540	13081	26163	65408
	Our Ker.	Simple implementation with BFS								
time	0.328	N/A	N/A	N/A	1.595	2.693	7.563	13.452	24.433	54.358
Speedup		N/A	N/A	N/A	4.87x	8.22x	23.08x	41.05x	74.57x	165.90x
$n = 1024$		# of edges								
		261	523	1309	2618	5237	26188	52377	104755	261888
	Our Ker.	Simple implementation with BFS								
time	0.837	N/A	N/A	3.988	5.656	8.612	28.339	51.344	94.925	218.861
Speedup		N/A	N/A	4.77x	6.76x	10.29x	33.86x	61.35x	113.43x	261.52x
$n = 2048$		# of edges								
		1048	2096	5240	10480	20961	104806	209612	419225	1048064
	Our Ker.	Simple implementation with BFS								
time	4.08	N/A	10.17	20.10	30.15	50.46	204.21	371.09	669.36	1436.61
Speedup		N/A	2.50x	4.93x	7.40x	12.38x	50.10x	91.04x	164.22x	352.46x
$n = 4096$		# of edges								
		4193	8386	20966	41932	83865	419328	838656	1677312	4193280
	Our Ker.	Simple implementation with BFS								
time	25.9	25.8	42.3	70.6	120.9	216.0	868.3	1577.8	2813.5	6149.3
Speedup		1.00x	1.64x	2.73x	4.67x	8.36x	33.59x	61.03x	108.82x	237.84x

Table 3.5: The running time (in milliseconds) of our single kernel implementation with SIMD functions and simple implementation with BFS when $n = 8192, 16384, 32768$. Note that # of edges is r% edges of clique with n nodes.

		$r = 0.05$	$r = 0.1$	$r = 0.25$	$r = 0.5$	$r = 1$
$n = 8192$		# of edges				
		16775	33550	83875	167751	335503
	Our Ker.	Simple implementation with BFS				
time	186	120	182	362	654	1226
Speedup		0.65x	0.98x	1.95x	3.52x	6.59x
		$r = 5$	$r = 10$	$r = 20$	$r = 50$	
		# of edges				
		1677516	3355033	6710067	16775168	
		Simple implementation with BFS				
		4815	7994	13256	27080	
Speedup		25.89x	42.97x	71.26x	145.57x	
		$r = 0.05$	$r = 0.1$	$r = 0.25$	$r = 0.5$	$r = 1$
$n = 16384$		# of edges				
		67104	134209	335523	671047	1342095
	Our Ker.	Simple implementation with BFS				
time	1439	658	1089	2395	4555	8774
Speedup		0.46x	0.76x	1.66x	3.17x	6.10x
		$r = 5$	$r = 10$	$r = 20$	$r = 50$	
		# of edges				
		6710476	13420953	26841907	67104768	
		Simple implementation with BFS				
		35291	54276	74139	131598	
Speedup		24.52x	37.72x	51.52x	91.45x	
		$r = 0.05$	$r = 0.1$	$r = 0.25$	$r = 0.5$	$r = 1$
$n = 32768$		# of edges				
		268427	536854	1342136	2684272	5368545
	Our Ker.	Simple implementation with BFS				
time	11402	4463	7919	18226	35257	68960
Speedup		0.39x	0.69x	1.60x	3.09x	6.05x
		$r = 5$	$r = 10$	$r = 20$	$r = 50$	
		# of edges				
		26842726	53685452	107370905	268427264	
		Simple implementation with BFS				
		295724	481468	690765	1179091	
Speedup		25.94x	42.23x	60.58x	103.41x	

Table 3.6: The running time (in milliseconds) of the single and multiple kernel implementation for $m = 8, 16, \ldots, 1024$ directed graphs with $n = 128, 256, \ldots, 2048$ nodes each

# of graphs		$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
8	Multiple kernel	0.1069	0.272	1.145	6.62	46.9
	Single kernel	**0.0999**	**0.194**	**0.901**	**5.83**	**43.4**
	Speedup	1.07x	1.40x	1.27x	1.14x	1.08x
16	Multiple kernel	0.1374	0.434	1.94	12.5	92.5
	Single kernel	**0.0930**	**0.319**	**1.62**	**11.4**	**86.6**
	Speedup	1.48x	1.36x	1.19x	1.10x	1.07x
32	Multiple kernel	0.182	0.665	3.50	24.2	183
	Single kernel	**0.114**	**0.538**	**3.10**	**22.6**	**173**
	Speedup	1.60x	1.24x	1.13x	1.07x	1.06x
64	Multiple kernel	0.239	1.154	6.61	47.7	365
	Single kernel	**0.179**	**0.906**	**6.10**	**45.0**	**345**
	Speedup	1.34x	1.27x	1.08x	1.06x	1.06x
128	Multiple kernel	0.382	1.93	12.9	94.7	729
	Single kernel	**0.313**	**1.75**	**12.1**	**89.8**	**690**
	Speedup	1.22x	1.10x	1.06x	1.06x	1.06x
256	Multiple kernel	0.684	3.69	25.4	189	1456
	Single kernel	**0.532**	**3.44**	**24.1**	**179**	**1380**
	Speedup	1.28x	1.07x	1.05x	1.05x	1.05x
512	Multiple kernel	1.25	7.17	50.4	377	2911
	Single kernel	**1.04**	**6.87**	**48.2**	**359**	**2761**
	Speedup	1.21x	1.04x	1.04x	1.05x	1.05x
1024	Multiple kernel	2.16	14.1	100.3	752	N/A
	Single kernel	**2.06**	**13.7**	**96.4**	**717**	N/A
	Speedup	1.05x	1.03x	1.04x	1.05x	N/A

Table 3.7: The running time (in milliseconds) of the single and multiple kernel implementation with SIMD functions for $m = 8, 16, \ldots, 1024$ directed graphs with $n = 128, 256, \ldots, 2048$ nodes each

# of graphs		$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
8	Multiple kernel with SIMD functions	0.0786	0.187	0.615	3.45	24.8
	Single kernel with SIMD functions	**0.0532**	**0.132**	**0.590**	**3.21**	**23.9**
	Speedup	1.48x	1.42x	1.04x	1.07x	1.04x
16	Multiple kernel with SIMD functions	0.0794	0.286	1.130	6.48	48.8
	Single kernel with SIMD functions	**0.0630**	**0.206**	**0.921**	**6.16**	**46.9**
	Speedup	1.26x	1.39x	1.23x	1.05x	1.04x
32	Multiple kernel with SIMD functions	0.1116	0.352	1.92	12.4	96.5
	Single kernel with SIMD functions	**0.0788**	**0.285**	**1.71**	**11.9**	**93.8**
	Speedup	1.42x	1.24x	1.12x	1.05x	1.03x
64	Multiple kernel with SIMD functions	0.1734	0.542	3.56	24.3	192
	Single kernel with SIMD functions	**0.0915**	**0.466**	**3.13**	**23.6**	**188**
	Speedup	1.89x	1.16x	1.14x	1.03x	1.02x
128	Multiple kernel with SIMD functions	0.203	0.972	6.29	48.2	382
	Single kernel with SIMD functions	**0.146**	**0.866**	**5.94**	**47.1**	**375**
	Speedup	1.39x	1.12x	1.06x	1.02x	1.02x
256	Multiple kernel with SIMD functions	0.306	1.81	12.3	96.1	764
	Single kernel with SIMD functions	**0.244**	**1.68**	**11.9**	**94.3**	**751**
	Speedup	1.25x	1.08x	1.03x	1.02x	1.02x
512	Multiple kernel with SIMD functions	0.519	3.17	24.2	193	1527
	Single kernel with SIMD functions	**0.447**	**3.04**	**23.8**	**189**	**1502**
	Speedup	1.16x	1.04x	1.02x	1.02x	1.02x
1024	Multiple kernel with SIMD functions	0.940	6.19	48.2	383	3052
	Single kernel with SIMD functions	**0.865**	**6.06**	**47.9**	**378**	**3004**
	Speedup	1.09x	1.02x	1.01x	1.01x	1.02x

Table 3.8: The running time (in milliseconds) of our low-latency implementation and the single and multiple kernel implementation for $m = 8, 16, \ldots, 1024$ directed graphs with $n = 128, \ldots, 2048$ nodes each. When we input 1024 graphs with $n = 2048$, we cannot execute the implementations because of the shortage of memory space on the GPU.

# of graph		$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
8	Multiple kernel	0.210	0.624	2.495	11.93	68.0
	Single kernel	0.196	0.560	2.255	11.13	64.5
	Low-Latency	0.107	0.272	1.145	6.62	46.9
	Speedup from Multiple ker.	1.96x	2.29x	2.18x	1.80x	1.45x
	Speedup from Single ker.	1.83x	2.06x	1.97x	1.68x	1.38x
16	Multiple kernel	0.326	1.116	4.60	23.04	134.6
	Single kernel	0.292	1.014	4.30	21.94	128.8
	Low-Latency	0.137	0.434	1.94	12.47	92.5
	Speedup from Multiple ker.	2.38x	2.57x	2.37x	1.85x	1.46x
	Speedup from Single ker.	2.13x	2.34x	2.22x	1.76x	1.39x
32	Multiple kernel	0.537	2.011	8.80	45.3	267.7
	Single kernel	0.478	1.896	8.42	43.7	257.1
	Low-Latency	0.182	0.665	3.50	24.2	183.3
	Speedup from Multiple ker.	2.95x	3.02x	2.51x	1.87x	1.46x
	Speedup from Single ker.	2.63x	2.85x	2.41x	1.81x	1.40x
64	Multiple kernel	0.917	3.819	17.19	89.9	534
	Single kernel	0.875	3.582	16.69	87.2	514
	Low-Latency	0.239	1.154	6.61	47.7	365
	Speedup from Multiple ker.	3.84x	3.31x	2.60x	1.88x	1.46x
	Speedup from Single ker.	3.66x	3.10x	2.52x	1.83x	1.41x
128	Multiple kernel	1.728	7.230	33.96	179.1	1066
	Single kernel	1.672	7.072	33.24	174.2	1028
	Low-Latency	0.382	1.925	12.85	94.7	729
	Speedup from Multiple ker.	4.52x	3.76x	2.64x	1.89x	1.46x
	Speedup from Single ker.	4.38x	3.67x	2.59x	1.84x	1.41x
256	Multiple kernel	3.345	14.27	67.6	357.5	2131
	Single kernel	3.211	14.05	66.4	348.3	2056
	Low-Latency	0.684	3.69	25.4	188.7	1456
	Speedup from Multiple ker.	4.89x	3.87x	2.66x	1.89x	1.46x
	Speedup from Single ker.	4.69x	3.81x	2.61x	1.85x	1.41x
512	Multiple kernel	6.559	28.29	134.8	714	4262
	Single kernel	6.351	28.03	132.7	697	4112
	Low-Latency	1.253	7.17	50.4	377	2911
	Speedup from Multiple ker.	5.23x	3.95x	2.67x	1.89x	1.46x
	Speedup from Single ker.	5.07x	3.91x	2.63x	1.85x	1.41x
1024	Multiple kernel	12.733	56.37	269.2	1428	N/A
	Single kernel	12.660	55.97	265.3	1393	N/A
	Low-Latency	2.158	14.13	100.3	752	N/A
	Speedup from Multiple ker.	5.90x	3.99x	2.68x	1.90x	N/A
	Speedup from Single ker.	5.87x	3.96x	2.65x	1.85x	N/A

Table 3.9: The running time (in milliseconds) of our low-latency implementation and the single and multiple kernel implementation with SIMD functions for $m = 8, 16, \ldots, 1024$ directed graphs with $n = 128, \ldots, 2048$ nodes each.

# of graph		$n = 128$	$n = 256$	$n = 512$	$n = 1024$	$n = 2048$
8	Multiple kernel with SIMD functions	0.110	0.258	0.801	4.14	27.5
	Single kernel with SIMD functions	0.096	0.209	0.791	3.91	26.6
	Low-Latency with SIMD functions	0.079	0.187	0.615	3.45	24.8
	Speedup from Multiple ker.	1.39x	1.38x	1.30x	1.20x	1.11x
	Speedup from Single ker.	1.22x	1.48x	1.29x	1.13x	1.07x
16	Multiple kernel with SIMD functions	0.122	0.407	1.48	7.82	54.1
	Single kernel with SIMD functions	0.117	0.323	1.29	7.52	52.2
	Low-Latency with SIMD functions	0.079	0.286	1.13	6.48	48.8
	Speedup from Multiple ker.	1.54x	1.42x	1.31x	1.21x	1.11x
	Speedup from Single ker.	1.48x	1.13x	1.14x	1.16x	1.07x
32	Multiple kernel with SIMD functions	0.175	0.538	2.61	15.1	107.1
	Single kernel with SIMD functions	0.154	0.484	2.40	14.6	104.4
	Low-Latency with SIMD functions	0.112	0.352	1.92	12.4	96.5
	Speedup from Multiple ker.	1.56x	1.53x	1.36x	1.22x	1.11x
	Speedup from Single ker.	1.38x	1.38x	1.25x	1.18x	1.08x
64	Multiple kernel with SIMD functions	0.277	0.894	4.91	29.6	213
	Single kernel with SIMD functions	0.207	0.831	4.49	28.9	209
	Low-Latency with SIMD functions	0.173	0.542	3.56	24.3	192
	Speedup from Multiple ker.	1.60x	1.65x	1.38x	1.22x	1.11x
	Speedup from Single ker.	1.20x	1.53x	1.26x	1.19x	1.09x
128	Multiple kernel with SIMD functions	0.416	1.656	8.96	58.8	425
	Single kernel with SIMD functions	0.346	1.561	8.62	57.7	417
	Low-Latency with SIMD functions	0.203	0.972	6.29	48.2	382
	Speedup from Multiple ker.	2.05x	1.70x	1.42x	1.22x	1.11x
	Speedup from Single ker.	1.70x	1.61x	1.37x	1.20x	1.09x
256	Multiple kernel with SIMD functions	0.658	3.16	17.6	117.2	848
	Single kernel with SIMD functions	0.608	3.03	17.2	115.4	835
	Low-Latency with SIMD functions	0.306	1.81	12.3	96.1	764
	Speedup from Multiple ker.	2.15x	1.75x	1.43x	1.22x	1.11x
	Speedup from Single ker.	1.99x	1.67x	1.40x	1.20x	1.09x
512	Multiple kernel with SIMD functions	1.205	5.83	34.8	235	1696
	Single kernel with SIMD functions	1.145	5.72	34.4	231	1671
	Low-Latency with SIMD functions	0.519	3.17	24.2	193	1527
	Speedup from Multiple ker.	2.32x	1.84x	1.44x	1.22x	1.11x
	Speedup from Single ker.	2.21x	1.80x	1.42x	1.20x	1.09x
1024	Multiple kernel with SIMD functions	2.285	11.50	69.4	467	3390
	Single kernel with SIMD functions	2.217	11.39	69.1	462	3342
	Low-Latency with SIMD functions	0.940	6.19	48.2	383	3052
	Speedup from Multiple ker.	2.43x	1.86x	1.44x	1.22x	1.11x
	Speedup from Single ker.	2.36x	1.84x	1.43x	1.21x	1.10x

Table 3.10: The running time (in milliseconds) of multicore implementations of parallel Blocked Floyd-Warshall algorithm with our SIMD functions and Intel AVX-512.

# of threads	$n = 512$		$n = 1024$		$n = 2048$		$n = 4096$	
	Our Func.	AVX-512	Our Func.	AVX-512	Our Func.	AVX-512	Our Func.	AVX-512
	(a) Each value D are stored in 8 bits							
1	133.59	2.14	1071.29	10.70	8570.3	324.8	59302.6	1808.7
2	68.47	1.94	547.35	7.04	4312.1	167.6	30111.4	975.8
4	37.78	**1.50**	286.10	5.04	2207.3	56.0	15408.5	501.6
8	21.38	1.55	154.19	**4.81**	1139.0	26.8	8023.6	250.2
16	14.88	2.69	101.52	6.17	667.9	19.3	4976.5	76.2
18	**13.76**	3.00	**84.77**	6.38	584.8	**18.0**	4413.6	**66.6**
32	17.32	5.58	96.59	10.50	626.1	19.7	5091.7	78.1
36	17.99	6.52	85.61	10.78	**559.2**	23.2	**4250.0**	72.5
	(b) Each value D are stored in 16 bits							
1	249.87	4.49	1962.57	81.42	13717.0	651.3	123087	7245
2	124.12	3.54	994.96	29.54	6966.0	337.6	61116	3996
4	72.43	2.45	510.38	11.72	3528.2	171.5	31093	2231
8	37.07	**2.17**	305.89	8.05	1881.1	58.9	15981	1543
16	25.72	2.25	165.64	**7.40**	1163.6	27.6	9102	586
18	**21.37**	2.72	**140.25**	8.15	1143.6	**27.5**	**8245**	551
32	25.41	5.86	158.55	10.64	1218.6	30.6	9056	**547**
36	26.02	7.06	144.38	10.32	**1021.2**	32.6	8255	564

# of threads	$n = 8192$		$n = 16384$	
	Our Func.	AVX-512	Our Func.	AVX-512
	(a) Each value D are stored in 8 bits			
1	524617	36258	4191203	274350
2	264836	20460	2119707	154531
4	135305	16410	1075147	123546
8	68765	14107	543037	**107541**
16	39969	14385	312090	112798
18	35928	14296	282888	115156
32	38185	**14107**	299638	114646
36	**35517**	14753	**266808**	116730
	(b) Each value D are stored in 16 bits			
1	985918	82276	8052052	730061
2	487654	48550	3992515	425921
4	247932	39013	2024565	338038
8	125368	**35903**	1020158	**323986**
16	71800	37450	586676	346229
18	65511	38249	531268	354970
32	71179	38816	576077	361863
36	**63543**	39733	**516402**	369829

Chapter 4

A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm

The first contribution of this paper is to show an implementation of the bulk execution of oblivious sequential algorithms on the UMM. Our implementation runs in $O(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm. We also prove that this implementation is time-optimal in the sense that any implementation takes at least $\Omega(\frac{pt}{w} + lt)$ time units on the UMM.

As a second contribution, we propose the *C2CU* tool, which allows converting a sequential C program into a CUDA C program. As the resulting CUDA C program makes coalescing memory access, even developers with few knowledge of CUDA C programming and GPU architecture can automatically generate CUDA C programs tailored for the bulk execution. To assess the performance of the C2CU generated programs, we have measured the running time of the bulk execution of three oblivious sequential algorithms: (i) bitonic sort [3, 4]; (ii) Floyd-Warshall algorithm [9, 13, 54]; and (iii) Montgomery modulo multiplication [6, 32, 47]. For this purpose, the aforementioned sequential algorithms have been written in C programming language. These sequential implementations were then fed to the C2CU generator to produce the corresponding CUDA C programs. The CUDA C programs have been executed on the GeForce GTX Titan GPU. Compared to the sequential implementation, running on a single CPU, the bulk execution of the bitonic sort runs 199 times faster while the Floyd-Warshall algorithm and Montgomery modulo multiplication run, respectively, 54 and 78 times faster. These experimental results are quite surprising since over 100 times acceleration have been obtained.

The bulk execution latency of the generated CUDA C implementation can be improved depending on the execution pattern. Thus, the third contribution of this work is to propose a modified execution pattern of the

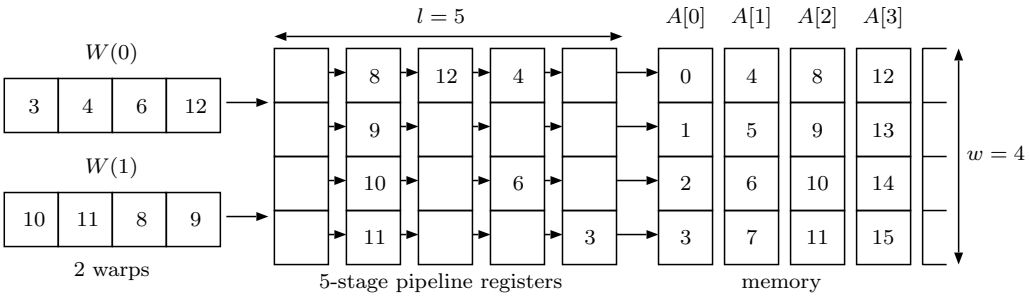


Figure 4.1: The UMM with width $w = 4$ and latency $l = 5$.

bulk execution so as to reduce latency. Experimental results show that, using the appropriate parameters, latency of the bulk execution can be reduced without compromising the overall running time.

The rest of this chapter is organized as follows. Section 4.1 defines the Unified Memory Machine (UMM). Section 4.2 presents the bulk execution of the oblivious sequential algorithms considered in this work. Section 4.3 presents the proposed C2CU generator. Section 4.4 evaluates the performance of the generated CUDA C programs using the GeForce GTX Titan. Section 4.5 proposes a modified execution pattern of the bulk execution so as to reduce execution latency.

4.1 The Unified Memory Machine (UMM)

The main purpose of this section is to define the Unified Memory Machine (UMM) [35]. Let us define the UMM with width w and latency l. The memory of the UMM is partitioned into address groups $A[0], A[1], \ldots$, such that each $A[j]$ $(j \geq 0)$ consists of addresses $j \cdot w$, $j \cdot w + 1$, $\ldots$, $(j + 1) \cdot w - 1$. Figure 4.1 illustrates address groups for UMM with width $w = 4$. Also, the memory access is performed through l-stage pipeline registers as illustrated in Figure 4.1. Let p be the number of threads of the UMM and $T(0)$, $T(1)$, $\ldots$, $T(p-1)$ be the p threads. We assume that p is a multiple of w. The p threads are partitioned into $\frac{p}{w}$ groups, called *warps*, with w threads each. More specifically, p threads are partitioned into $\frac{p}{w}$ warps $W(0), W(1)$, $\ldots$, $W(\frac{p}{w} - 1)$ such that $W(i) = \{T(i \cdot w), T(i \cdot w + 1), \ldots, T((i + 1) \cdot w - 1)\}$. Warps are dispatched for the memory access in turn, and w threads in a warp try to access the memory at the same time. More specifically, $W(0), W(1), \ldots, W(\frac{p}{w} - 1)$ are dispatched in a round-robin manner if at least one thread in a warp requests memory access. If no thread in a warp needs memory access, such warp is not dispatched. When $W(i)$ is dispatched, w threads in $W(i)$ send memory access requests, one request per thread, to the MBs.

Each warp sends memory access requests to the MBs through the l-stage pipeline registers. We assume that each stage can store memory access requests destined for the same address group. For example, since the memory access requests by $W(0)$ are separated in three address groups in Figure 4.1, they occupy three stages of the pipeline registers. Also, those by $W(1)$ are in the same address group, they occupy only one stage. In general, if the memory access requests by a warp are destined for k address groups, they occupy k stages. For simplicity, we assume that the memory access is completed as soon as the request reaches the

last pipeline stage. Thus, all memory access requests by $W(0)$ and $W(1)$ in our example are completed in $3(\text{address groups}) + 1(\text{address group}) + 5(\text{latency}) - 1 = 8$ time units. We also assume that a thread cannot send a new memory access request until the previous memory access request is completed. Hence, if a thread sends a memory access request, it must wait at least l time units to send a new memory access request.

4.2 Oblivious sequential algorithms and the bulk execution

The main purpose of this section is to introduce oblivious sequential algorithms and their corresponding bulk execution. Intuitively, a sequential algorithm is *oblivious* if an address accessed at each time unit is independent of the input. More precisely, there exists a function $a : \{0, 1, \ldots, t-1\} \to \mathcal{N}$, where t is the running time of the algorithm and $\mathcal{N}$ is a set of all non-negative integers such that, for any input of the algorithm, it accesses address $a(i)$ or does not access the memory at each time i $(0 \leq i \leq t-1)$. In other words, at each time i $(0 \leq i \leq t-1)$, it never accesses an address other than $a(i)$.

Let us see an example of oblivious algorithms. Suppose that an array b of n numbers is given. The prefix-sum computation is a task to store each i-th prefix-sum $b[0]+b[1]+\cdots+b[i]$ in $b[i]$. Let r be a register variable. The following algorithm computes the prefix-sum of n numbers.

Algorithm 3 Algorithm Prefix-sums

```
r ← 0;
for i ← 0 to n − 1 do
    r ← r + b[i];
    b[i] ← r;
end for
```

Since $b[0]$, $b[1]$, ..., $b[n-1]$ are added to r in turn, the prefix-sums are stored in b correctly when this algorithm terminates. Let us see the address accessed at each time unit to confirm that this algorithm is oblivious. For simplicity, we ignore access to registers and local computation such as addition and we assume that such operations can be done in zero time unit. Clearly, memory access operations performed in this algorithm are: read $b[0]$, write $b[0]$, read $b[1]$, write $b[1]$, ..., read $b[n-1]$, and write $b[n-1]$. Hence, the memory access function a is $a(2i) = a(2i+1) = i$ for all i $(0 \leq i \leq n-1)$, and thus, this algorithm is oblivious.

Suppose that we need to execute a sequential algorithm for many different inputs on a single CPU in turn or on a parallel machine at the same time. We call such computation as a *bulk execution*. For instance, suppose that we have p arrays $b_0, b_1, \ldots, b_{p-1}$ of size n each on the UMM. The goal of the bulk execution of the prefix-sums is to compute the prefix-sums of every b_j $(0 \leq j \leq p-1)$ on the UMM in parallel. We use p threads and each thread $T(j)$ $(0 \leq j \leq p-1)$ computes the prefix-sums of b_j by using the Parallel Algorithm Prefix-sums. Let r_j $(0 \leq j \leq p-1)$ be a register of thread $T(j)$. The prefix-sums can be computed in parallel by the following algorithm:

Let us consider two arrangements of the p arrays of size n each as follows:

Algorithm 4 Parallel Algorithm Prefix-sums

```
for j ← 0 to p − 1 do in parallel
    r_j ← 0;
    for i ← 0 to n − 1 do
        r_j ← r_j + b_j[i];
        b_j[i] ← r_j;
    end for
end for
```

- **row-wise**: they are arranged in a 2-dimensional array with p rows and n columns, such that each $b_j[i]$ $(0 \leq i \leq n-1, 0 \leq j \leq p-1)$ is stored in the j-th row and in the i-th column, which is allocated in address $j \cdot n + i$;
- **column-wise**: they are arranged in a 2-dimensional array with n rows and p columns, such that each $b_j[i]$ $(0 \leq i \leq n-1, 0 \leq j \leq p-1)$ is stored in the i-th row and in the j-th column, which is allocated in address $i \cdot p + j$.

Figure 4.2 illustrates the column-wise and row-wise arrangements for $p = 8$ arrays of size $n = 6$ each. Let us evaluate the time to execute the prefix-sum algorithm for the row-wise arrangement (row-wise prefix-sums) and column-wise arrangement (column-wise prefix-sums). For simplicity, we assume that p is multiple of w and that n is sufficient large so that $n \geq w$. In the row-wise arrangement, for each i $(0 \leq i \leq n-1)$, $b_0[i], b_1[i], \ldots, b_{p-1}[i]$ stored in addresses $i, i+n, \ldots, i+(p-1)n$, are accessed by p threads. They are in p different address groups and corresponding p memory access requests occupy p pipeline stages. Thus, it takes $p+l-1$ time units to complete them. Figure 4.3 illustrates the memory access requests by warps $W(0)$ and $W(1)$ with $w = 4$ on the row-wise arrangement. Each warp has $w = 4$ memory access requests and their memory addresses are in $p = 8$ address groups. These requests occupy $p = 8$ pipeline stages, taking $p + l - 1 = 8 + 6 - 1$ time units to complete. In the column-wise arrangement, for each i $(0 \leq i \leq n-1)$, $b_0[i], b_1[i], \ldots, b_{p-1}[i]$ stored in continuous addresses $i \cdot p, i \cdot p+1, \ldots, i \cdot p+(p-1)$, are accessed by p threads. They are in $\frac{p}{w}$ address groups. The corresponding $\frac{p}{w}$ memory access requests occupy $\frac{p}{w}$ pipeline stages, taking $\frac{p}{w} + l - 1$ time units to complete them. Figure 4.4 depicts the memory access requests by warp $W(0)$ and $W(1)$ with $w = 4$ for the column-wise arrangement. Warps $W(0)$ and $W(1)$ have $w = 4$ memory access requests whose memory addresses are in $\frac{p}{w} = \frac{8}{4} = 2$ address groups. Their requests occupy $\frac{p}{w} = \frac{8}{4} = 2$ pipeline stages, taking $\frac{p}{w} + l - 1 = \frac{8}{4} + 6 - 1 = 2 + 6 - 1$ time units to complete. A round of the memory access performed by p threads is a *stride* access if the threads access distinct address groups. Figure 4.3 depicts a stride access pattern. Conversely, a *coalesced* memory access occurs when the threads in $\frac{p}{w}$ warps access the same address group (depicted in Figure 4.4). For latter reference, we summarize the above discussion in the following lemma:

Lemma 4.2.1 *A round of the memory access performed by p threads can be completed in $\frac{p}{w} + l - 1$ time units using coalesced access and in $p + l - 1$ time units using stride access.*

0 $b_0[0]$	1 $b_1[0]$	2 $b_2[0]$	3 $b_3[0]$	4 $b_4[0]$	5 $b_5[0]$	6 $b_6[0]$	7 $b_7[0]$
8 $b_0[1]$	9 $b_1[1]$	10 $b_2[1]$	11 $b_3[1]$	12 $b_4[1]$	13 $b_5[1]$	14 $b_6[1]$	15 $b_7[1]$
16 $b_0[2]$	17 $b_1[2]$	18 $b_2[2]$	19 $b_3[2]$	20 $b_4[2]$	21 $b_5[2]$	22 $b_6[2]$	23 $b_7[2]$
24 $b_0[3]$	25 $b_1[3]$	26 $b_2[3]$	27 $b_3[3]$	28 $b_4[3]$	29 $b_5[3]$	30 $b_6[3]$	31 $b_7[3]$
32 $b_0[4]$	33 $b_1[4]$	34 $b_2[4]$	35 $b_3[4]$	36 $b_4[4]$	37 $b_5[4]$	38 $b_6[4]$	39 $b_7[4]$
40 $b_0[5]$	41 $b_1[5]$	42 $b_2[5]$	43 $b_3[5]$	44 $b_4[5]$	45 $b_5[5]$	46 $b_6[5]$	47 $b_7[5]$

Column-wise

0 $b_0[0]$	1 $b_0[1]$	2 $b_0[2]$	3 $b_0[3]$	4 $b_0[4]$	5 $b_0[5]$
6 $b_1[0]$	7 $b_1[1]$	8 $b_1[2]$	9 $b_1[3]$	10 $b_1[4]$	11 $b_1[5]$
12 $b_2[0]$	13 $b_2[1]$	14 $b_2[2]$	15 $b_2[3]$	16 $b_2[4]$	17 $b_2[5]$
18 $b_3[0]$	19 $b_3[1]$	20 $b_3[2]$	21 $b_3[3]$	22 $b_3[4]$	23 $b_3[5]$
24 $b_4[0]$	25 $b_4[1]$	26 $b_4[2]$	27 $b_4[3]$	28 $b_4[4]$	29 $b_4[5]$
30 $b_5[0]$	31 $b_5[1]$	32 $b_5[2]$	33 $b_5[3]$	34 $b_5[4]$	35 $b_5[5]$
36 $b_6[0]$	37 $b_6[1]$	38 $b_6[2]$	39 $b_6[3]$	40 $b_6[4]$	41 $b_6[5]$
42 $b_7[0]$	43 $b_7[1]$	44 $b_7[2]$	45 $b_7[3]$	46 $b_7[4]$	47 $b_7[5]$

Row-wise

Figure 4.2: Column-wise and Row-wise arrangements for $p = 8$ arrays of size $n = 6$ each.

From Lemma 4.2.1, the computation of the row-wise prefix-sums takes $(p+l-1)\cdot O(n) = O(np+nl)$ time units. Likewise, each contiguous access takes $\frac{p}{w} + l - 1$ time units and the computation of the column-wise prefix-sums takes $(\frac{p}{w} + l - 1) \cdot O(n) = O(\frac{np}{w} + nl)$ time units. Thus, we have:

Lemma 4.2.2 *The row-wise prefix-sums of an array of size $p \times n$ and the column-wise prefix-sums of an array of size $n \times p$ can be computed in $O(np + nl)$ and $O(\frac{np}{w} + nl)$ time units, respectively, using p threads on the UMM with width w and latency l.*

We can evaluate the running time of any oblivious algorithm in the same way. Suppose that we have an oblivious sequential algorithm whose execution takes t time units. Without loss of generality, the algorithm works on an array of size n. Similarly to the prefix-sum computation above, suppose that the oblivious sequential algorithm is executed for p inputs of size n each, using p threads on the UMM in parallel. We can consider two arrangements, row-wise and column-wise arrangements for the oblivious sequential algorithm.

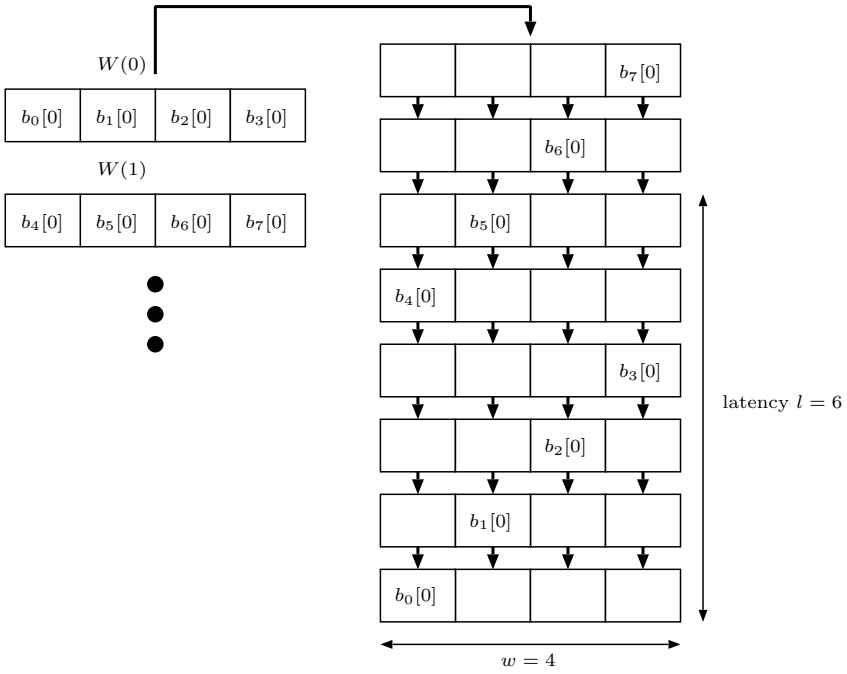


Figure 4.3: Memory access requests executed by warps $W(0)$ and $W(1)$ with $w = 4$ and $p = 8$ for the row-wise arrangement.

We say that such execution is *a row-wise oblivious computation* if the p arrays are arranged in 2-dimensional array of size $p \times n$ such that each row corresponds to an input array of the oblivious sequential algorithm. Likewise, a *column-wise oblivious computation* occurs if the input is arranged in 2-dimensional array of size $n \times p$, such that each column corresponds to an input array.

Let us evaluate the computing time on the UMM for the row-wise and the column-wise arrangements. Let $a(j)$ $(0 \leq j \leq t-1)$ denote the address accessed by the oblivious sequential algorithm. In the row-wise arrangement, for each j $(0 \leq j \leq t-1)$, p threads access $a(j), a(j)+n, \ldots, a(j)+(p-1)n$, which are in p different address groups. According to Lemma 4.2.1, such memory access takes $p+l-1$ time units. Thus, the oblivious algorithm runs in $(p+l-1) \cdot t = O(pt+lt)$ time units on the row-wise arrangement. In the column-wise arrangement, they access $a(j) \cdot p, a(j) \cdot p+1, \ldots, a(j) \cdot p+(p-1)$, which are in $\frac{p}{w}$ address groups. Since such memory access takes $\frac{p}{w}+l-1$ time units from Lemma 4.2.1, the oblivious algorithm runs in $(\frac{p}{w}+l-1) \cdot t = O(\frac{pt}{w}+lt)$ time units. Thus, we have:

Theorem 4.2.1 *Any row-wise oblivious computation of size $p \times n$ and any column-wise oblivious computation of size $n \times p$ run in $O(pt+lt)$ and in $O(\frac{pt}{w}+lt)$ time units, respectively, using p threads on the UMM with width w and latency l, where t is the running time of the corresponding oblivious sequential algorithm.*

We also prove that column-wise oblivious computation for Theorem 4.2.1 is time-optimal on the UMM. Since an oblivious algorithm running t time units is executed p times, it may involve pt memory access operations. Since the width of the UMM is w, it takes at least $\frac{pt}{w}$ time units to complete pt memory access operations. Furthermore, since an oblivious algorithm performs t memory access operations in turn, it takes

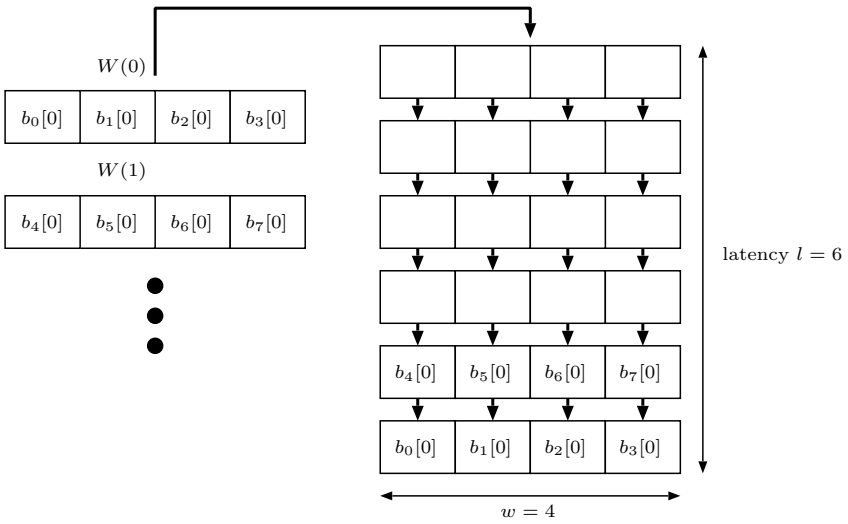


Figure 4.4: Memory access requests executed by warps $W(0)$ and $W(1)$ with $w = 4$ and $p = 8$ for the column-wise arrangement.

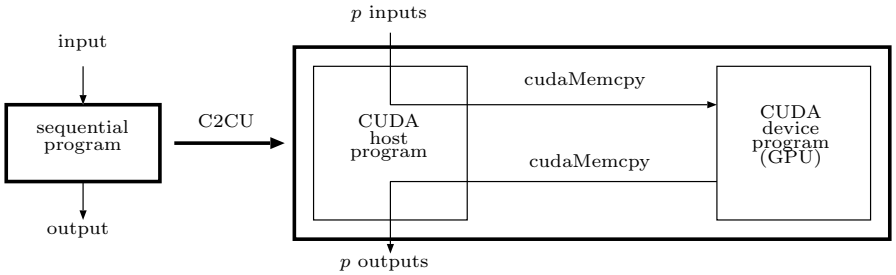


Figure 4.5: The behavior of C2CU generator.

at least lt time units on the UMM. Thus, we have:

Theorem 4.2.2 *Any implementation of bulk execution of an oblivious algorithm for p inputs takes at least $\Omega(\frac{pt}{w} + lt)$ time units using p threads on the UMM with width w and latency l, where t is the running time of the oblivious sequential algorithm.*

4.3 C2CU Generator

The main purpose of this section is to describe C2CU generator. The C2CU takes as input a sequential algorithm, written in C programming language and generates the corresponding CUDA C program for the bulk execution on a CUDA-enabled GPUs. Figure 4.5 illustrates the behavior of C2CU generator. The generated C program accepts p independent inputs, which are copied from the host to the device memory (global memory) of the GPU for bulk execution. The CUDA device program with p threads is spawned, and each thread executes the sequential program for one input. After all threads terminate, p outputs obtained by all threads are copied back to the host.

Let us see how the C2CU generates a CUDA C program using the Floyd-Warshall algorithm [9,13,54] as an example. The Floyd-Warshall algorithm is described in Section 3.1, and computes the distances of shortest paths of all-pairs of nodes in a directed graph. It uses a 2-dimensional array D of size $n \times n$ representing an n-node graph. We assume that, initially, $D[i][j]$ $(0 \leq i, j \leq n-1)$ stores the distance between nodes i and j, if it exists, and $+\infty$ otherwise. The Floyd-Warshall algorithm is described as follows:

Algorithm 5 Floyd-Warshall algorithm

```
for k ← 0 to n − 1 do
    for i ← 0 to n − 1 do
        for j ← 0 to n − 1 do
            D[i][j] ← min{D[i][j], D[i][k] + D[k][j]}
        end for
    end for
end for
```

After termination of the algorithm, $D[i][j]$ stores the distance of the shortest path from node i to j. If no such path exists, it stores $+\infty$. The C program in Figure 4.6 is a direct implementation of the Floyd-Warshall algorithm. The values of D is updated by calling `update_dist`, although it is not necessary to be a function. The reason is to show the C2CU supports function calls. The C program includes the directive `#pragma kernel` in line 22. Most C compilers, such as the GNU C compiler, ignore this directive. Hence, this C program can be compiled correctly, and it computes the all-pairs shortest-distance of an input graph using the Floyd-Warshall algorithm. The above directive is used to specify the function call for the bulk execution on the GPU. Thus, a function call just after directive `#pragma kernel` will be executed on the GPU in the CUDA C program obtained by C2CU.

Figure 4.7 shows the CUDA C program generated by the C2CU from the C program shown in Figure 4.6. Users can specify the number p of inputs (i.e. the number p of threads) and the number of threads in each CUDA block, by using the C2CU options. These values are defined as `__P__` $(= p)$ and `__T__` in lines 2 and 3. In Figure 4.7, they are set to 2048 and 64, respectively. Thus, 32 CUDA blocks, with 64 threads each, are spawned by CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31. Since the generated CUDA C program accepts p inputs, a 3-dimensional array D of size $N \times N \times p$, allocated in the host memory, are used to store them. Also, a 3-dimensional array `__D` of the same size, allocated in the device memory (i.e. the global memory of the GPU), is used. In line 30, the `cudaMemcpyToSymbol` function is used to copy p inputs stored in D to `__D`. After the bulk execution by CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31, the `cudaMemcpyToSymbol` function is used to copy the resulting values from `__D` back to D.

CUDA kernel call `floyd_warshall<<<__B__,__T__>>>()` in line 31 invokes `__B__` CUDA blocks with `__T__` threads each. Thus, `__P__` $(= p)$ threads execute the Floyd-Warshall algorithm on the CUDA-enabled GPU. Since `blockDim.x` is the number `__B__` of threads in a CUDA block and `blockIdx.x` and `threadIdx.x` take values in $[0, \texttt{\_\_B\_\_} - 1]$ and $[0, \texttt{\_\_T\_\_} - 1]$, respectively, `__id__` in line 15 takes value from 0 to $p-1$. Hence, the device function `update_dist(i,j,k,__id__)` is executed for `__id__` in $[0, p-1]$ on the GPU in

```
 1: #define N 1024
 2: float D[N][N];
 3: void update_dist(int i, int j, int k){
 4:   if( D[i][j] > D[i][k] + D[k][j] ) {
 5:     D[i][j] = D[i][k] + D[k][j];
 6:   }
 7: }
 8:
 9: void floyd_warshall(){
10:   int i,j,k;
11:   for(k=0;k<N;k++) {
12:     for(i=0;i<N;i++) {
13:       for(j=0;j<N;j++) {
14:         update_dist(i,j,k);
15:       }
16:     }
17:   }
18: }
19:
20: int main(int argc, char *argv[]){
21:   input_array();
22: #pragma kernel
23:   floyd_warshall();
24:   ...
```

Figure 4.6: A C program implementation of the Floyd-Warshall algorithm.

parallel. The reader should have no difficulty to confirm that the CUDA C program in Figure 4.7 executes the Floyd-Warshall algorithm for p inputs in parallel.

Let us see how C2CU converts a C program into a CUDA C program for general cases and confirm that the generated CUDA C program performs coalesced memory access. If the original C program uses a d-dimensional array a of size $s_1 \times s_2 \times \cdots \times s_d$, a CUDA C program generated by C2CU uses a $(d+1)$-dimensional array a of size $s_1 \times s_2 \times \cdots \times s_d \times p$. If the original C program accesses $a[i_1][i_2]\cdots[i_d]$ then each thread with ID id of the CUDA C program accesses $a[i_1][i_2]\cdots[i_d][\texttt{__id__}]$. Since $a[i_1][i_2]\cdots[i_d][0]$, $a[i_1][i_2]\cdots[i_d][1]$, ..., $a[i_1][i_2]\cdots[i_d][p-1]$ are allocated in consecutive addresses, these memory accesses by the p threads are coalesced.

```
 1: #define N 1024
 2: #define __P__ 2048
 3: #define __T__ 64
 4: #define __B__  __P__/__T__
 5: float D[N][N][__P__];
 6: __device__ float __D[N][N][__P__];
 7:
 8: __device__ void update_dist(int i, int j, int k, int __id__){
 9:   if( __D[i][j][__id__] > __D[i][k][__id__]
                                        + __D[k][j][__id__] ) {
10:     __D[i][j][__id__] = __D[i][k][__id__] + __D[k][j][__id__];
11:   }
12: }
13:
14: __global__ void floyd_warshall(){
15:   int __id__ = blockIdx.x * blockDim.x + threadIdx.x;
16:   int i,j,k;
17:   for(k=0;k<N;k++) {
18:     for(i=0;i<N;i++) {
19:       for(j=0;j<N;j++) {
20:         update_dist(i,j,k,__id__);
21:       }
22:     }
23:   }
24: }
25:
26: int main(int argc, char *argv[])
27: {
28:   input_array();
29: #pragma kernel
30:   cudaMemcpyToSymbol(__D, D, sizeof(float)*N*N*__P__, 0);
31:   floyd_warshall<<<__B__,__T__>>>();
32:   cudaMemcpyFromSymbol(D, __D, sizeof(float)*N*N*__P__, 0);
33:   ...
```

Figure 4.7: A CUDA program for the bulk execution of the Floyd-Warshall algorithm generated by C2CU.

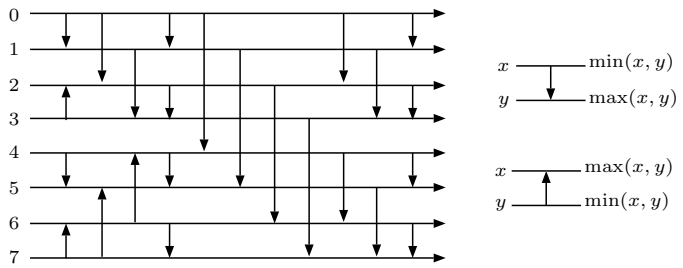


Figure 4.8: Bitonic sort for $n = 8$.

4.4 Experiment results

The main purpose of this section is to show the experimental results on GeForce GTX Titan. GeForce GTX Titan has 14 streaming multiprocessors with 192 cores each. Hence, it can run 2688 threads in parallel. Note that, a single kernel call to GeForce GTX Titan can run more than 2688 threads in a time sharing manner using CUDA [41] parallel programming platform. All input and output data are stored in the global memory of the GPU. Recall that the UMM does not allow to use the shared memory of the streaming multiprocessors.

In the experiments, we consider the following sequential algorithms:

- bitonic sort [3, 4];
- Floyd-Warshall algorithm [9, 13, 54]; and
- Montgomery modulo multiplication [6, 32, 47].

Bitonic sort is a well-known parallel sorting algorithm developed by K. E. Batcher [4]. It can be described as a sorting network with comparators as illustrated in Figure 4.8. Since the number of compare-exchange elements in each stage is fixed, the bitonic sort can be written as an oblivious sequential algorithm.

Montgomery modulo multiplication is used to speed the modulo multiplication $X \cdot Y \cdot 2^{-R} \bmod M$ for R-bit numbers X, Y, and M. The idea of Montgomery modulo multiplication is not to use direct modulo computation, which is very costly in terms of the computing time and hardware resources. By iterative computation of Montgomery modulo multiplication, the modulo exponentiation $P^E \bmod M$ can be computed, which is a key operation for the RSA encryption and decryption [5]. Since R is at least 1024, to use Montgomery modulo multiplication for RSA encryption and decryption, addition/multiplication is repeated to perform R-bit addition/multiplication. Figure 4.9 illustrates how the product $(a \cdot b)$ of two integers, a and b of large bits, is computed. Both a and b are partitioned into four integers and the sum of pair-wise products is computed. Since Montgomery modulo multiplication repeats computation of the product and the sum of two large integers, it can also be computed by an oblivious sequential algorithm.

We have written a C program for bitonic sort that sorts $n = 32$, 1K (= 1024) and 32K (= 32768) 32-bits float numbers. The C2CU was used to convert it into a CUDA C program for the bulk execution of bitonic sort with parameter $p = 64, 128, \ldots, 4\text{M}$. However, due to the global memory capacity of the GPU, it is

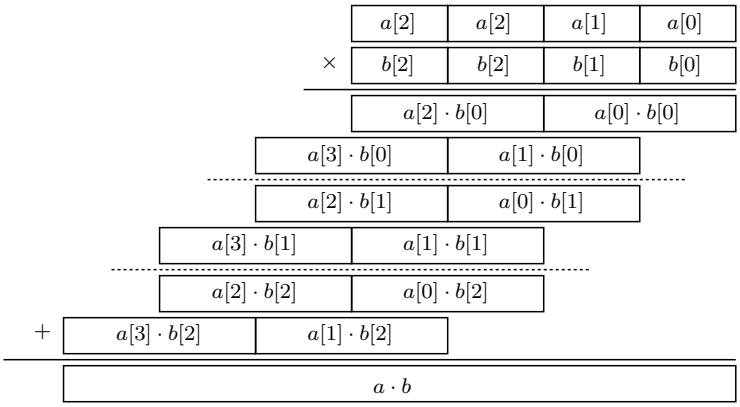


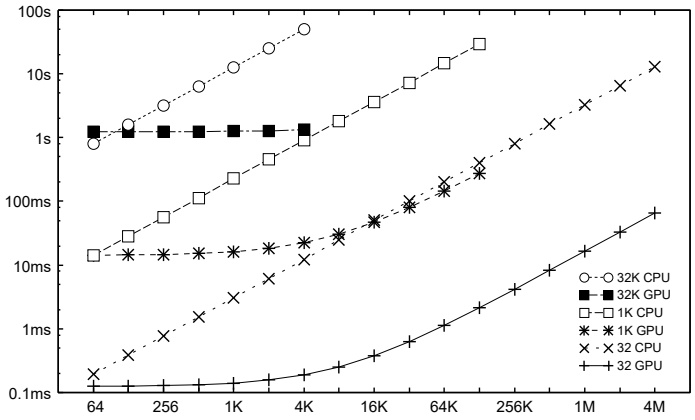
Figure 4.9: Multiplication of two integers with large bits.

executed up to $p = 128$K and $p = 4$K when $n = 1$K and $n = 32$K, respectively. The CUDA C program invokes p threads in $\frac{p}{64}$ CUDA blocks with 64 threads each to sort p inputs of n numbers each.

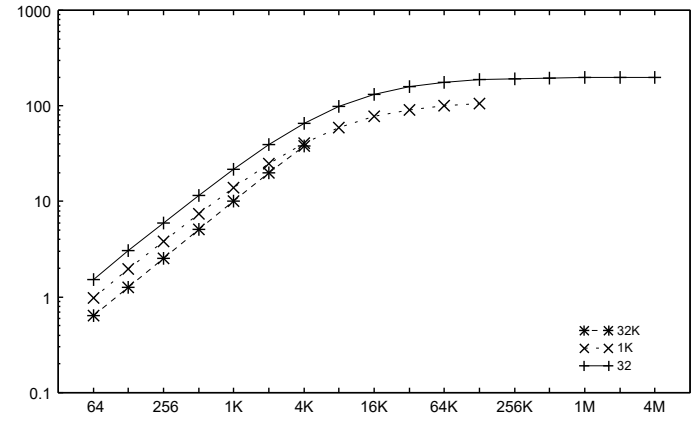
Figure 4.10 (1) shows the computing time for the bulk execution of bitonic sort. Recall that, from Theorem 4.2.1, the bulk execution of a sequential algorithm can be computed in $O(\frac{pt}{w} + lt)$ time units, where p is the total number of threads, l is the memory access latency, and t is the running time of the original sequential algorithm. The bulk execution of bitonic sort for $n = 32$ takes about 0.13ms when $p \leq 1$K. Further, the computing time is proportional to p when $p \geq$16K and it runs 65.1ms when $p = 4$M. Thus, we can think that $O(lt) = 0.13$ms and $O(\frac{pt}{w}) = (15.5p)$ns. More specifically, the bulk execution of bitonic sort for $n = 32$ and $p \geq 1$K can be computed in approximately 0.13ms+$(15.5p)$ns. To see the speed-up factor, the original C program is repeatedly executed p times on a Intel Xeon CPU (2.66GHz). Figure 4.10 (2) shows the speed-up factor of the GPU over the CPU. We can see that the bulk execution of bitonic sort on the GPU can achieve a speed-up of more than 180 times when $n = 32$ and $p \geq 128$K. Furthermore, when $n = 32$ and $p = 4$M, the GPU is 199 times faster than the CPU.

Our C implementation of the Floyd-Warshall algorithm works on graphs having $n = 16$, 64, and 256 nodes. We use 32-bit floating-point numbers to store the length of each edge. As before, the C program is converted into a CUDA C program using C2CU with parameters $p = 16$, 64, and 256. However, due to the global memory capacity of the GPU, it is executed up to $p = 16$K and $p = 1$K when $n = 64$ and $n = 256$, respectively. The CUDA C program also invokes p threads in $\frac{p}{64}$ CUDA blocks with 64 threads each.

Figure 4.11 (1) shows the computing time for the bulk execution of the Floyd-Warshall algorithm. We will show that the bulk execution time of the Floyd-Warshall algorithm matches the $O(\frac{pt}{w} + lt)$ time units of Theorem 4.2.1. As can be observed in Figure 4.11 (1), the bulk execution of the Floyd-Warshall algorithm for $n = 16$ takes about 3.4ms when $p \leq 512$. The computing time is proportional to p when $p \geq$4K, running in 42.6ms when $p = 128$K. Thus, we can think that $O(ln^3) = 3.4$ms and $O(\frac{pn^3}{w}) = (325p)$ns. More specifically, the bulk execution of the Floyd-Warshall algorithm for $n = 16$ and $p \geq 512$ can be computed in approximately 3.4ms+(325)ns. Figure 4.11 (2) shows the speed-up of the GPU over the CPU. We can see that the bulk execution on the GPU can achieve a speed-up of more than 30 times when $n = 16$ and $p \geq 8$K.



(1) The computing time



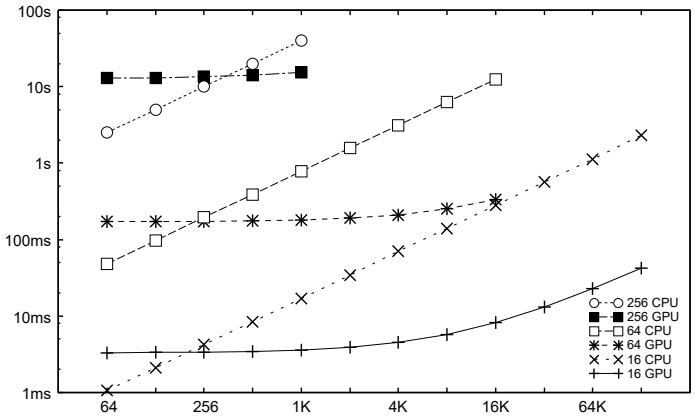
(2) GPU/CPU speed-up factor

Figure 4.10: The computing time (ms) of bitonic sort on CPU and GPU, and the speed-up for $n = 32$, 1K, 32K, and $p = 64$, 128, ..., 4M.

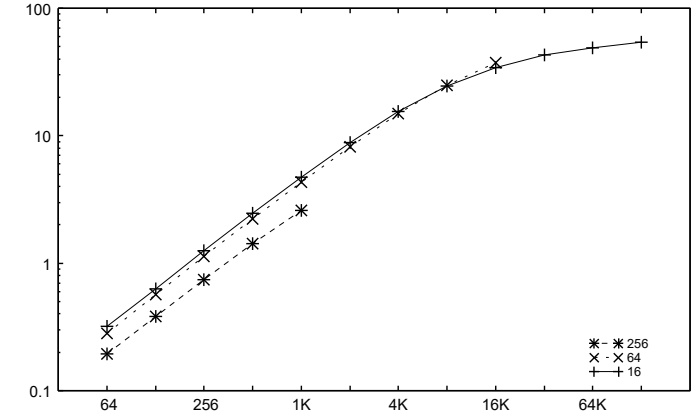
Furthermore, when $n = 16$ and $p = 128$K, the GPU is 54 times faster than the CPU.

Finally, we have written a C program for Montgomery modulo multiplication for $n = 512$, 16K (= 16384), and 1M (= 1048576) bits. The C2CU was used to convert it into a CUDA C program with parameter $p = 64, 128, \ldots, 2$M. However, due to the global memory capacity, it is executed for $p = 64$K and $p = 2$K when $n = 16$K and $n = 1$M, respectively. The CUDA C program also invokes p threads in $\frac{p}{64}$ CUDA blocks with 64 threads each.

Figure 4.12 (1) shows the computing time for the bulk execution of Montgomery modulo multiplication. As above, we will show that the $O(\frac{pt}{w} + lt)$ time units of Theorem 4.2.1 holds for the bulk execution of the Montgomery modulo multiplication. The bulk execution of the algorithm for $n = 512$ takes about 0.45ms when $p \leq 512$. Also, the computing time is proportional to p when $p \geq$128K and it runs 124ms when $p = 2$M. Thus, we can think that $O(ln^2) = 0.45$ms and $O(\frac{pn^2}{w}) = (59.1p)$ns. More specifically, the bulk execution of the algorithm for $n = 512$ can be computed in approximately 124ms+$(5.9p)$ns. Figure 4.12 (2) shows the speed-up of the GPU over the CPU. We can see that the GPU can achieve a speed-up of more than 70 times when $n = 512$ and $p \geq 32$K. Furthermore, when $n = 512$ and $p = 2$M, the GPU is 78 times faster than the



(1) The computing time



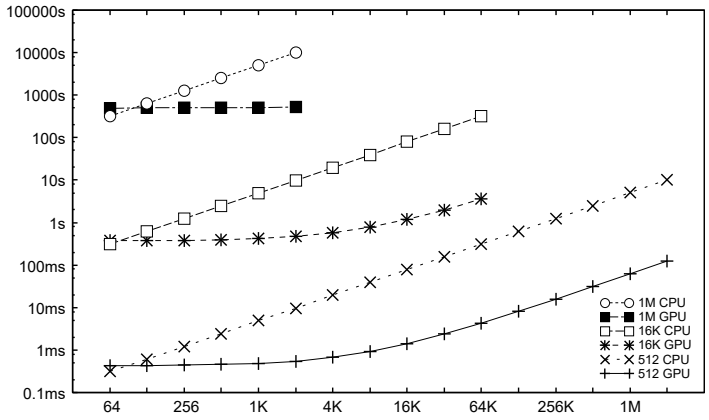
(2) GPU/CPU speed-up factor

Figure 4.11: The computing time (ms) of the Floyd-Warshall algorithm on CPU and GPU, and the speed-up for $n = 16$, 64, 256, and $p = 64$, 128, ..., 128K.

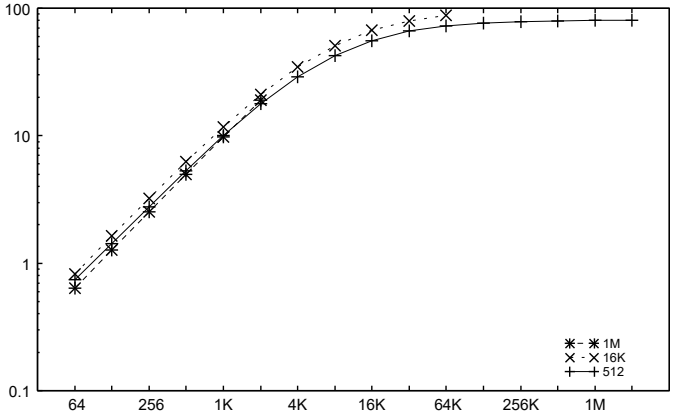
CPU.

4.5 Low-latency implementation for Bulk Execution

In previous sections, we have presented and evaluated the proposed C2CU generator. The obtained results showed substantial speed-up. On the other hand, latency increases due to the fact that the final output is computed at the host only after receiving all the partial results from the device. More specifically, suppose that p inputs $s_0, s_1, \ldots, s_{p-1}$ are given in turn. The CUDA C program generated by the C2CU provides p outputs, $u_0, u_1, \ldots u_{p-1}$, which are available only after the termination of the CUDA C program. An alternative to reduce latency is to obtain each output u_i as soon the corresponding input s_i is given. In this section we explore this idea to reduce the bulk execution latency. To achieve this goal, the input is partitioned into groups and the bulk execution is performed on the GPU for each group in turn. As shown in Figure 4.13, the idea is to partition the p inputs into $\frac{p}{b}$ groups of b inputs each. Then, the generated CUDA C program performs the computation for each group b in turn.



(1) The computing time



(2) GPU/CPU speed-up factor

Figure 4.12: The computing time (ms) of the Montgomery modulo multiplication on CPU and GPU, and the speed-up for $p = 64, 128, \ldots, 4\text{M}$.

We assume that each group of b inputs is given to the main memory of the host PC at each interval of time $T > 0$. From the host PC, each group is then transferred to the global memory of the GPU. The `timeSetEvent()` is used for invoking the computation of each bulk execution after T time units. Clearly, the latency between inputs is bounded by T. After the completion of the bulk execution, the resulting values are copied back to the main memory of the host PC. These operations are repeated for all groups in turn. Figure 4.14 illustrates how the computation and data transfer are performed. In the figure, "HtoD", "kernel" and "DtoH" correspond to the data transfer from Host to Device (i.e. GPU), kernel execution (i.e. bulk execution); and the data transfer from Device to Host, respectively.

As in the previous section, the performance evaluation of the proposed low-latency alternative has been carried out on the GeForce GTX Titan. Figure 4.15 shows the throughput and the latency for the bulk execution of the bitonic sort with $p = 128\text{M}$ (= 134217728) inputs of $n = 32$ numbers for $b = 1\text{K}, 2\text{K}, \ldots, 4\text{M}$. When $b \leq 8\text{K}$, the throughput is proportional to b. The throughput increases slightly when $8\text{K} \leq b \leq 128\text{K}$, and it remains unchanged when $b \geq 128\text{K}$. The latency remains unchanged when $b \leq 8\text{K}$, and it is proportional to b when $b \geq 8\text{K}$. We can see that the throughput saturates when $b \geq 128\text{K}$ due to

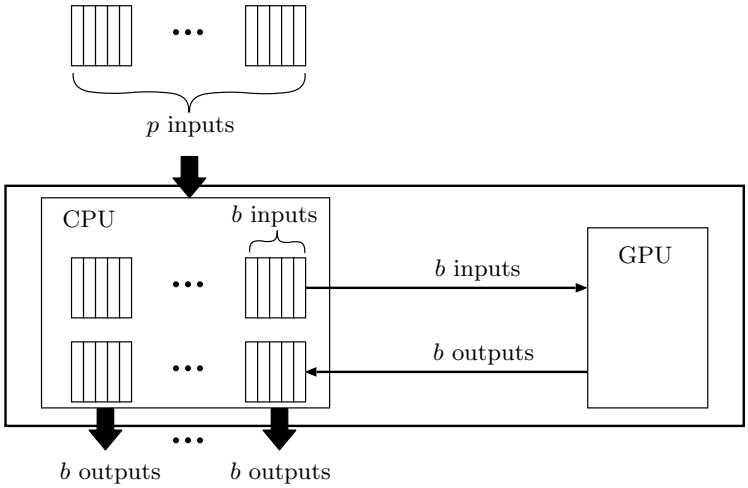


Figure 4.13: Low-latency GPU implementation outline.

Figure 4.14: The behavior of the proposed low-latency GPU implementation.

the bandwidth limitation between the device and the host PC.

We can see that the throughput for $b = 128$K is almost the same as that for $b = 4$M. The latency is 12.0ms when $b = 128$K and 373ms when $b = 4$M. Clearly, by selecting $b = 128$K allows for low-latency computation with nearly maximum throughput.

We have also evaluated the performance for the Floyd-Warshall algorithm with $p =$640K (= 655360) graphs of $n = 16$ nodes each with parameter $b = 64, 128, \ldots, 64$K. Figure 4.16 shows the throughput and the latency. When $b \leq 4$K, the throughput is proportional to b and the latency increases slightly. When $b \geq 4$K, the throughput increases slightly and the latency is proportional to b. We can see that the throughput saturates for $b \geq 4$K. The reason for such behavior is the bandwidth limit between the device and the host PC.

We can see that the throughput for $b = 4$K is slightly smaller than that for $b = 64$K. The latency is 4.82ms when $b = 4$K and 48.3ms when $b = 64$K. Clearly, choosing $b = 64$K allows for low latency and high throughput for the bulk execution in this case.

We have further evaluated the performance for Montgomery modulo multiplication on the GPU. Figure 4.17 shows the throughput and the latency for $p =$4M (= 4194304) inputs of $n = 512$ bits each with parameter $b = 512, 1$K, ..., 2M. When $b \leq 8$K, the throughput is proportional to b. The throughput increases slightly when $8\text{K} \leq b \leq 64$K, and it remains unchanged when $b \geq 64$K. The latency increases slightly when $b \leq 4$K, and it is proportional to b when $b \geq 4$K. We can see that the throughput saturates when $b \geq 64$K due to the bandwidth limitation between the device and the host PC.

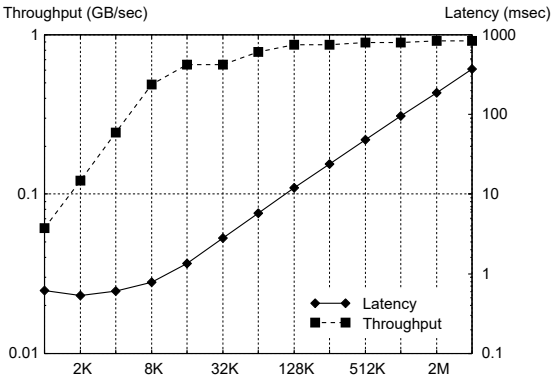


Figure 4.15: Throughput (GB/sec) and latency (ms) for the bulk execution of bitonic sort for $p = 128$M inputs of $n = 32$ numbers each.

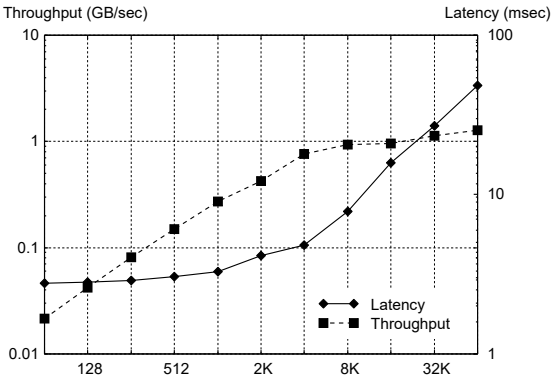


Figure 4.16: Throughput (GB/sec) and latency (ms) for the bulk execution of the Floyd-Warshall algorithm for $p = 640$K inputs of $n = 16$ nodes each.

We can see that the throughput for $b = 64$K is slightly lower than that for $b = 512$K. On the other hand, the latency is 13.3ms when $b = 64$K and 406ms when $b = 2$M. The best performance, in terms of the latency and the throughput, is achieved when $b = 64$K.

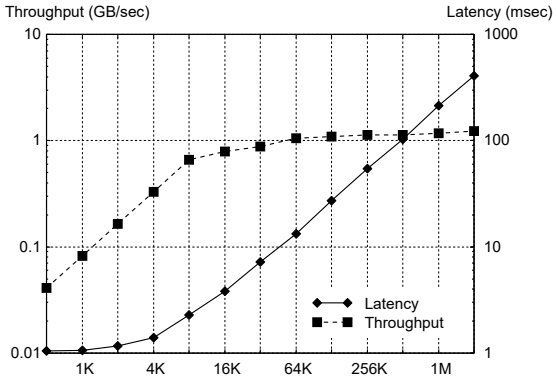


Figure 4.17: Throughput (GB/sec) and latency (ms) for the bulk execution of the Montgomery modulo multiplication for $p = 4$M inputs of $n = 512$ bits each.

Chapter 5

Concluding remarks

For first contributions: First, we have proposed efficient GPU implementations of the Blocked Floyd-Warshall algorithm, and have evaluated them through experimental results using NVIDIA Tesla V100. It performs no barrier synchronization and invokes only one kernel call. Also, we have proposed SIMD functions so as to execute saturating addition and minimum selection for updating values. We have applied these SIMD functions to GPU implementations and multicore implementations.

When we give one graph, we have proposed single kernel implementation for a graph on the GPU. Our Single kernel implementation runs 1.05-1.31 times faster than multiple kernel implementation. Also, we have applied SIMD functions to above GPU implementations. Our Single kernel implementation with SIMD functions runs 1.00-1.28 times faster than multiple kernel implementation with SIMD functions.

Secondly, we have proposed four GPU implementations for many graphs at the same time: multiple kernel implementation without/with SIMD functions and single kernel implementation without/with SIMD functions. In terms of implementations without SIMD functions, single kernel implementation runs 1.03-1.60 times faster than multiple kernel implementation. Also, single kernel implementation runs 1.01-1.89 times faster than multiple kernel one.

If many graphs are given, the time to transfer between Host and the device increases and latency increases. Hence, we have proposed the low-latency implementations. In terms of implementations without SIMD functions, the low-latency implementation outputs the results 1.45-5.90 faster than multiple kernel implementation, and 1.38-5.87 faster than single kernel implementation. Also, the low-latency implementation with SIMD functions outputs the results 1.11-2.43 faster than multiple kernel implementation, and 1.07-2.36 faster than single kernel implementation.

Finally, we have implemented parallel Floyd-Warshall algorithm on the multicore processors.
For second contributions: This work focuses on the performance of the *bulk* execution of an oblivious sequential algorithm. In this regard, the first contribution of this paper was to present a time-optimal implementation for the bulk execution of an oblivious sequential algorithm.

Next, we have proposed a tool, termed C2CU, to assist developers to generate appropriate CUDA C

programs tailored for the bulk execution. The proposed C2CU takes as input a C language program of a sequential algorithm and generates the CUDA C program for the bulk execution on a CUDA-enabled GPU. Experimental results have shown that the generated CUDA C program, executed on the GeForce GTX Titan, can deliver speed-up of 199 times as compared to its original C program running on an Intel Xeon CPU. Although the obtained results showed substantial speed-up, latency has increased since the final result is computed at the host only after receiving all the partial results from the device. Hence, our third contribution was to propose a low-latency GPU implementation for the bulk execution of an oblivious sequential algorithm. Experimental results confirmed that the aforementioned implementation delivers high-throughput at low-latency.

Bibliography

[1] K. Abdelghany, H. Hashemi, and A. Al-nawaiseh, "Parallel all-pairs shortest path algorithm: Network-decomposition approach," *Transportation Research Record*, Vol. 2567, No. 1, pp. 95–104, Jan. 2016.

[2] A. V. Aho, J. D. Ullman, and J. E. Hopcroft, *Data Structures and Algorithms*, Addison Wesley, 1983.

[3] S. G. Akl, *Parallel Sorting Algorithms*, Academic Press, 1985.

[4] K. E. Batcher, "Sorting networks and their applications," in *Proc. AFIPS Spring Joint Comput. Conf.*, Vol. 32, pp. 307–314, 1968.

[5] T. Blum and C. Paar, "High-radix Montgomery modular exponentiation on reconfigurable hardware," *IEEE Trans. on Computers*, Vol. 50, No. 7, pp. 759–764, 2001.

[6] S. Bo, K. Kawakami, K. Nakano, and Y. Ito, "An RSA encryption hardware algorithm using a single DSP block and a single block RAM on the FPGA," *International Journal of Networking and Computing*, Vol. 1, No. 2, pp. 277–289, July 2011.

[7] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," in *Proceedings 20th IEEE International Parallel & Distributed Processing Symposium*. Proceedings 20th IEEE International Parallel & Distributed Processing Symposium, pp. 10 pp.–, April 2006.

[8] A. Buluç, J. R. Gilbert, and C. Budak, "Solving path problems on the GPU," *Parallel Comput.*, Vol. 36, No. 5-6, pp. 241–253, June 2010.

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

[10] H. Djidjev, S. Thulasidasan, G. Chapuis, R. Andonov, and D. Lavenier, "Efficient multi-GPU computation of all-pairs shortest paths," in *Proc. of IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 360–369, May 2014.

[11] Y. Emoto, S. Funasaka, H. Tokura, T. Honda, K. Nakano, and Y. Ito, "An optimal parallel algorithm for computing the summed area table on the GPU," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*. Proc. of International Parallel and Distributed Processing Symposium Workshops, pp. 763–772, May 2018.

[12] R. W. Floyd, "Algorithm 97: Shortest path," *Commun. ACM*, Vol. 5, No. 6, pp. 345, June 1962.

[13] R. W. Floyd, "Algorithm 97: Shortest path," *Communications of the ACM*, Vol. 5, No. 6, pp. 345, June 1962.

[14] M. J. Flynn, "Some computer organizations and their effectiveness," *IEEE Transactions on Computers*, Vol. C-21, pp. 948–960, 1972.

[15] M. L. Fredman, "New bounds on the complexity of the shortest path problem," *SIAM Journal on Computing*, Vol. 5, No. 1, pp. 83–89, 1976.

[16] T. Fujita, K. Nakano, Y. Ito, and D. Takafuji, "An efficient GPU implementation of CKY parsing using the bitwise parallel bulk computation technique," *IEICE Transactions on Information and systems*, pp. 2857–2865, Dec. 2017.

[17] S. Funasaka, K. Nakano, and Y. Ito, "Single kernel soft synchronization technique for task arrays on CUDA-enabled GPUs, with applications," in *Proc. of 2017 Fifth International Symposium on Computing and Networking (CANDAR)*. Proc. of 2017 Fifth International Symposium on Computing and Networking (CANDAR), pp. 11–20, Nov. 2017.

[18] S.-C. Han, F. Franchetti, and M. Püschel, "Program generation for the all-pairs shortest path problem," in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, New York, NY, USA, Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques, PACT '06, pp. 222–232, ACM, 2006.

[19] Y. Han and T. Takaoka, "An $O(n^3 \log\log n/\log^2 n)$ time algorithm for all pairs shortest paths," *J. Discrete Algorithms*, Vol. 38-41, pp. 9–19, 2016.

[20] P. Harish and P. J. Narayanan, "Accelerating large graph algorithms on the GPU using CUDA," in *Proceedings of the 14th International Conference on High Performance Computing*, Berlin, Heidelberg, Proceedings of the 14th International Conference on High Performance Computing, HiPC'07, pp. 197–208, Springer-Verlag, 2007.

[21] W. W. Hwu, *GPU Computing Gems Emerald Edition*, Morgan Kaufmann, 2011.

[22] Intel Corporation, "Intel advanced vector extensions 512," `https://www.intel.com/content/www/us/en/architecture-and-technology/avx-512-overview.html`.

[23] A. Kasagi, K. Nakano, and Y. Ito, "Offline permutation algorithms on the discrete memory machine with performance evaluation on the GPU," *IEICE Transactions on Information and Systems*, Vol. E96-D, No. 12, pp. 2617–2625, Dec. 2013.

[24] G. J. Katz and J. Joseph T. Kider, "All-pairs shortest-paths for large graphs on the GPU," in *Proc. of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pp. 47–55, Jun. 2008.

[25] G. J. Katz and J. T. Kider, "All-pairs shortest-paths for large graphs on the GPU," in *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*. Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware, pp. 47–55, 2008.

[26] P. Kipfer and R. Westermann, "Chapter 46. Improved GPU sorting," in *GPU Gems 2*. Addison-Wesley, 2005.

[27] B. D. Lund and J. W. Smith, "A multi-stage CUDA kernel for Floyd-Warshall," *CoRR*, Vol. abs/1001.4108, 2010.

[28] D. Man, K. Uda, Y. Ito, and K. Nakano, "A GPU implementation of computing Euclidean distance map with efficient memory access," in *Proc. of International Conference on Networking and Computing*. Proc. of International Conference on Networking and Computing, pp. 68–76, Dec. 2011.

[29] 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.

[30] P. J. Martín, R. Torres, and A. Gavilanes, "CUDA solutions for the SSSP problem," Berlin, Heidelberg, Computational Science – ICCS 2009, pp. 904–913, Springer Berlin Heidelberg, 2009.

[31] K. Matsumoto, N. Nakasato, and S. G. Sedukhin, "Blocked united algorithm for the all-pairs shortest paths problem on hybrid CPU-GPU systems," *IEICE Transactions on Information and Systems*, Vol. E95.D, No. 12, pp. 2759–2768, 2012.

[32] P. L. Montgomery, "Modular multiplication without trial division," *Mathematics of Computation*, Vol. 44, No. 170, pp. 519–521, 1985.

[33] K. Nakano, "Efficient implementations of the approximate string matching on the memory machine models," in *Proc. of International Conference on Networking and Computing*, pp. 233–239, Dec. 2012.

[34] K. Nakano, "An optimal parallel prefix-sums algorithm on the memory machine models for GPUs," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*. pp. 99–113, Springer, Sept. 2012.

[35] K. Nakano, "Simple memory machine models for GPUs," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*, pp. 788–797, May 2012.

[36] K. Nakano, "Asynchronous memory machine models with barrier synchronization," *IEICE Transactions on Information and Systems*, Vol. E97-D, No. 3, pp. 431–441, Mar. 2014.

[37] K. Nakano, D. Takafuji, S. Fujita, H. Matsutani, I. Fujiwara, and M. Koibuchi, "Randomly optimized grid graph for low-latency interconnection networks," in *Proc. of International Conference on Parallel Processing (ICPP)*. Proc. of International Conference on Parallel Processing (ICPP), pp. 340–349, 2016.

[38] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the matrix chain product on the GPU," in *Proc. of International Conference on Networking and Computing*, pp. 320–326, Dec. 2011.

[39] K. Nishida, Y. Ito, and K. Nakano, "Accelerating the dynamic programming for the optimal polygon triangulation on the GPU," in *Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP, LNCS 7439)*, pp. 1–15, Sept. 2012.

[40] NVIDIA Corporation, "NVIDIA CUDA C best practice guide version 10.1," https://docs.nvidia.com/cuda/archive/10.1/cuda-c-best-practices-guide/index.html, 2019.

[41] NVIDIA Corporation, "NVIDIA CUDA C programming guide version 10.1," 2019.

[42] K. Ogawa, Y. Ito, and K. Nakano, "Efficient Canny edge detection using a GPU," in *Proc. of International Conference on Networking and Computing*, pp. 279–280, Nov. 2010.

[43] T. Okuyama, F. Ino, and K. Hagihara, "Fast blocked floyd-warshall algorithm on the GPU," *IPSJ Transactions on Advanced Computing Systems*, Vol. 3, No. 2, pp. 57–66, Jun 2010.

[44] T. Okuyama, F. Ino, and K. Hagihara, "A task parallel algorithm for finding all-pairs shortest paths using the GPU," *International Journal of High Performance Computing and Networking*, pp. 87–98, April 2012.

[45] OpenMP, ," `https://www.openmp.org/`.

[46] H. Ortega-Arranz, Y. Torres, D. Ferraris, and A. Gonzalez-Escribano, "A new GPU-based approach to the shortest path problem," Proceedings of the 2013 International Conference on High Performance Computing and Simulation, HPCS 2013, July 2013.

[47] K. Shigemoto, K. Kawakami, and K. Nakano, "Accelerating Montgomery modulo multiplication for redundant radix-64k number system on the FPGA using dual-port block RAMs," in *Proc. of International Conference on Embedded and Ubiquitous Computing(EUC)*, pp. 44–51, 2008.

[48] D. Takafuji, K. Nakano, Y. Ito, and J. Bordim, "C2cu: a cuda c program generator for bulk execution of a sequential algorithm," *Concurrency and Computation: Practice and Experience*, Vol. 29, No. 17, pp. e4022, 2017.

[49] K. Tani, D. Takafuji, K. Nakano, and Y. Ito, "Bulk execution of oblivious algorithms on the unified memory machine, with GPU implementation," in *Proc. of International Parallel and Distributed Processing Symposium Workshops*. Proc. of International Parallel and Distributed Processing Symposium Workshops, pp. 586–595, May 2014.

[50] A. Uchida, Y. Ito, and K. Nakano, "Fast and accurate template matching using pixel rearrangement on the GPU," in *Proc. of International Conference on Networking and Computing*, pp. 153–159, Dec. 2011.

[51] A. Uchida, Y. Ito, and K. Nakano, "An efficient GPU implementation of ant colony optimization for the traveling salesman problem," in *Proc. of International Conference on Networking and Computing*, pp. 94–102, Dec. 2012.

[52] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," *J. Exp. Algorithmics*, Vol. 8, Dec. 2003.

[53] S. Warshall, "A theorem on boolean matrices," *J. ACM*, Vol. 9, No. 1, pp. 11–12, Jan. 1962.

[54] S. Warshall, "A theorem on Boolean matrices," *Journal of the ACM*, , No. 1, pp. 11–12, Jan. 1962.

[55] R. Williams, "Faster all-pairs shortest paths via circuit complexity," *SIAM Journal on Computing*, Vol. 47, No. 5, pp. 1965–1985, 2018.

[56] Q. Wu, C. Tong, Q. Wang, and X. Cheng, "All-pairs shortest path algorithm based on MPI+CUDA distributed parallel programming model," *Journal of Networks*, Vol. 8, pp. 2797–2803, 2013.

Published Works

Journals

J-1 Daisuke Takafuji, Koji Nakano, Yasuaki Ito and Jacir Bordim, "C2CU: a CUDA C program generator for bulk execution of a sequential algorithm," Concurrency and Computation: Practice and Experience, Vol. 29, No. 17, Sep. 2017.

Chapter 4: A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm

J-2 Daisuke Takafuji, Koji Nakano and Yasuaki Ito, "Efficient Parallel Implementations to Compute the Diameter of a Graph", Concurrency and Computation: Practice and Experience, to appear.

Chapter 3: Computing of All-Pairs Shortest Paths of a Graph

International Conferences

C-1 Daisuke Takafuji, Koji Nakano and Yasuaki Ito, "C2CU : A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm," Proc. of International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP LNCS 8631), pp. 178–191, Aug. 2014.

Chapter 4: A CUDA C Program Generator for Bulk Execution of a Sequential Algorithm

C-2 Daisuke Takafuji, Koji Nakano and Yasuaki Ito, "Efficient GPU Implementations to Compute the Diameter of a Graph," Proc. of International Symposium on Computing and Networking (CANDAR) pp. 102–111, Nov. 2019.

Chapter 3: Computing of All-Pairs Shortest Paths of a Graph

Others

1. Toru Fujita, Koji Nakano, Yasuaki Ito and Daisuke Takafuji, "An Efficient GPU Implementation of CKY Parsing Using the Bitwise Parallel Bulk Computation Technique," IEICE Trans. Inf.& Syst., Vol. E100-D, No. 12, pp. 2857–2865, Dec. 2017.

2. Yuji Takeuchi, Koji Nakano, Daisuke Takafuji and Yasuaki Ito, "A character art generator using the local exhaustive search, with GPU acceleration," International Journal of Parallel, Emergent and Distributed Systems, Vol. 31 No. 1, pp. 47–63, Jan. 2016.

3. Koji Nakano, Daisuke Takafuji, Satoshi Fujita, Hiroki Matsutani, Ikki Fujiwara and Michihiro Koibuchi "Randomly Optimized Grid Graph for Low-Latency Interconnection Networks," Proc. of International Conference on Parallel Processing (ICPP), pp. 340–349, 2016.

4. Yuji Takeuchi, Daisuke Takafuji, Yasuaki Ito and Koji Nakano, "ASCII Art Generation using the Local Exhaustive Search on the GPU," Proc. of International Symposium on Computing and Networking, pp. 194–200, Dec. 2013.

5. Daisuke Takafuji, Satoshi Taoka, Yasunori Nishikawa and Toshimasa Watanabe, "Enhanced approximation algorithms for maximum weight matchings of graphs," IEICE Trans. Fundamentals, Vol. E91-A, No. 4, pp. 1129–1139, 2008.

6. Satoshi Taoka, Daisuke Takafuji and Toshimasa Watanabe, "Computing-Based Performance Analysis of Approximation Algorithms for the Minimum Weight Vertex Cover Problem of Graphs," IEICE Trans. Fundamentals, Vol. E96-A, No. 6, pp. 1331–1339, Jun. 2013.

7. Satoshi Taoka, Daisuke Takafuji and Toshimasa Watanabe, "Enhancing PC Cluster-Based Parallel Branch-and-Bound Algorithms for the Graph Coloring Problem," IEICE Trans. Fundamentals, Vol. E91-A, No. 4, pp. 1140–1149, Apr. 2008.

8. Satoshi Taoka, Daisuke Takafuji and Toshimasa Watanabe, "Performance Comparison of Algorithms for the Dynamic Shortest Path Problem," IEICE Trans. Fundamentals, Vol. E90-A, No. 4, pp. 847–856, Apr. 2007.