A Task Parallel Algorithm for Finding All-Pairs Shortest Paths Using the GPU

Tomohiro Okuyama*, Fumihiko Ino* and Kenichi Hagihara
Graduate School of Information Science and Technology,
Osaka University, Japan
E-mail: t-okuyam@ist.osaka-u.ac.jp E-mail: ino@ist.osaka-u.ac.jp
*Corresponding author

Abstract: This paper proposes an acceleration method for finding the all-pairs shortest paths (APSPs) using the graphics processing unit (GPU). Our method is based on Harish's iterative algorithm that computes the cost of the single-source shortest path (SSSP) in parallel on the GPU. In addition to this fine-grained parallelism, we exploit the coarse-grained parallelism by using a task parallelization scheme that associates a task with an SSSP problem. This scheme solves multiple SSSP problems at a time, allowing us to efficiently access graph data by sharing the data between processing elements in the GPU. Furthermore, our fine- and coarse-grained parallelization leads to a higher parallelism, increasing the efficiency with highly threaded code. As a result, the speedup over the previous SSSP-based implementation ranges from a factor of 2.8 to that of 13, depending on the graph topology. We also show that the overhead of path recording needed after cost computation increases the execution time by 7.7%.

Keywords: graphics processing unit; all-pairs shortest path; high performance computing; graph algorithm; acceleration; CUDA.

Reference to this paper should be made as follows: Okuyama, T., Ino, F. and Hagihara, K. (xxxx) 'A task parallel algorithm for finding all-pairs shortest paths using the GPU', Int. J. High Performance Computing and Networking, Vol. xx, Nos. xx/xx, pp.xx–xx.

Biographical notes: Tomohiro Okuyama received the B.E. degree in engineering from Osaka University, Osaka, Japan, in 2008. He is currently working toward the M.E. degree at the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. His current research interests include high performance computing and systems architecture and design.

Fumihiko Ino received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1998, 2000, and 2004, respectively. He is currently an Associate Professor in the Graduate School of Information Science and Technology at Osaka University. His research interests include parallel and distributed systems, software development tools, and performance evaluation. He received the Best Paper Award at the 2003 International Conference on High Performance Computing (HiPC'03) and the Best Paper Award at the 2004 Symposium on Advanced Computing Systems and Infrastructures (SACSIS'04).

Kenichi Hagihara received the B.E., M.E., and Ph.D. degrees in information and computer sciences from Osaka University, Osaka, Japan, in 1974, 1976, and 1979, respectively. From 1994 to 2002, he was a Professor in the Department of Informatics and Mathematical Science, Graduate School of Engineering Science, Osaka University. Since 2002, he has been a Professor in the Department of Computer Science, Graduate School of Information Science and Technology, Osaka University. From 1992 to 1993, he was a Visiting Researcher at the University of Maryland. His research interests include the fundamentals and practical application of parallel processing. He received the Best Paper Award at the 2003 International Conference on High Performance Computing (HiPC'03) and the Best Paper Award at the 2004 Symposium on Advanced Computing Systems and Infrastructures (SACSIS'04).

1 Introduction

The all-pairs shortest path (APSP) problem is to find shortest paths between all two vertices in a graph. This operation is one of the basic graph algorithms, which has many applications in a wide variety of fields, such as computer aided design (Shenoy, 1997), intelligent transportation systems (Kim and Lee, 1999), data mining (Jayadevaprakash et al., 2005) and bioinformatics (Nakaya et al., 2001).

However, the APSP problem requires a large amount of computation. For instance, the Floyd-Warshall (FW) (Floyd, 1962; Warshall, 1962) algorithm solves this problem in $O(|V|^3)$ time, where $|V|$ represents the number of vertices in a graph. A straightforward method for solving the APSP problem is to iteratively compute single-source shortest path (SSSP) for every source vertex. Dijkstra's algorithm (Dijkstra, 1959) accelerated with a Fibonacci heap is known as a fast method to find an SSSP for a sparse graph. Using this algorithm, we can find APSPs in $O(|V||E| + |V|^2 \log |V|)$ time, where $|E|$ represents the number of edges in a graph. In contrast to algorithmic studies mentioned above, many researchers are trying to accelerate the algorithms using various accelerators, such as graphics processing units (GPUs) (Harish and Narayanan, 2007; Katz and Kider, 2008; Micikevicius, 2004), field-programmable gate arrays (FPGAs) (Bondhugula et al., 2006), and clusters (Srinivasan et al., 2006).

To the best of our knowledge, Harish and Narayanan (2007) present the fastest method for large graphs. Their method computes the costs of APSPs instead of the paths. The cost of a path here is given by the sum of the weights of edges composing the path. They accelerate the FW algorithm and an SSSP-based iterative algorithm using the compute unified device architecture (CUDA) compatible GPU (nVIDIA Corporation, 2008). CUDA is a development framework for running massively multithreaded parallel applications on the GPU (Owens et al., 2007), which consists of hundreds of processing elements, called stream processors (SPs). They show that the SSSP-based algorithm is approximately six times faster than the FW algorithm and the former demonstrates a more scalable performance with respect to the number $|V|$ of vertices. However, we think that the SSSP-based algorithm can be further accelerated by memory access optimization. For example, the algorithm may be modified such that it fully uses the entire memory hierarchy, including fast but small on-chip shared memory.

In this paper, we propose a variant of the SSSP-based algorithm that enhances the previous method by exploiting not only off-chip memory but also on-chip memory. The proposed method extends the preliminary work presented in (Okuyama et al., 2008). Our method uses a different parallelization scheme for the cost computation of APSPs in order to save the bandwidth between off-chip memory and SPs. In addition to the fine-grained parallelism exploited by the previous method, we exploit the coarse-grained parallelism existing between different SSSP problems. That is, the proposed scheme exploits task parallelism so that it solves in parallel multiple SSSP problems with different sources. This allows SPs to simultaneously access the same data because each SP takes the responsibility for solving one of the task-parallel problems. Such common access leads to an efficient use of on-chip shared memory, which is useful to reduce data accesses to off-chip memory. Furthermore, the proposed scheme contributes to achieve higher speedup with more parallel tasks and less synchronization on the GPU. We also describe how APSPs can be recorded with their costs.

The rest of the paper is organized as follows. Section 2 gives a brief introduction of related work. Section 3 describes an overview of CUDA and Section 4 summarizes the previous SSSP-based method. Section 5 presents our algorithm and Section 6 shows experimental results. Finally, Section 7 concludes the paper.

2 Related Work

Harish and Narayanan (2007) present two APSP algorithms, namely the FW algorithm and the SSSP-based iterative algorithm, both implemented using CUDA. They demonstrate that the SSSP-based implementation takes approximately 10 seconds to obtain APSP costs for a graph of $|V| = 3072$ vertices. The speedup over the CPU-based FW implementation reaches a factor of 17. With respect to memory consumption, their SSSP-based algorithm requires $O(|V|)$ space while the FW algorithm requires $O(|V|^2)$ space. This advantage allows us to deal with larger graphs, up to $|V| = 30,720$ vertices processed within two minutes. However, only off-chip memory is used because (1) there is no data that can be shared between SPs and (2) the entire graph data is too large for 16 KB of on-chip memory.

Katz and Kider (2008) propose an optimized version of the FW implementation running on the CUDA-compatible GPU. Their method is based on a blocked FW algorithm proposed by Venkataraman et al. (2000). It takes 13.7 seconds to compute APSPs for a graph of $|V| = 4096$. A multi-GPU version is also presented to demonstrate the scalability of their method. It takes 354 seconds to process a graph of $|V| = 11,264$ vertices on two nVIDIA GeForce 8800 GT cards.

Bleiweiss (2008) implements Dijkstra's algorithm and A* search algorithm (Hart et al., 1968) with a priority queue using CUDA. Their implementations are designed for agent navigation in crowded game scenes, where multiple point-to-point shortest paths are simultaneously computed for smaller graphs. Thus, their problem is slightly different from our target problem.

Micikevicius (2004) presents an OpenGL-based method that implements the FW algorithm on the GPU by mapping it to the graphics pipeline. The implementation runs on an nVIDIA GeForce 5900 Ultra, which demonstrates three times faster results compared with a 2.4 GHz Pentium 4 CPU. It takes approximately 203 seconds to compute APSPs for $|V| = 2048$.

An FPGA-based method is proposed by Bondhugula et al. (2006). They implement a tiled version of the FW algorithm and develop an analytical model to predict the performance for larger FPGAs. As compared with a CPU-based method running on a 2.2 GHz Opteron, their method reduces execution time for $|V| = 16,384$ from approximately four hours to 15 minutes, achieving a speedup of 15.2.

An automated tuning approach is proposed by Han et al. (2006) to accelerate the FW algorithm on the CPU. Their method is optimized by cache blocking and single instruction, multiple data (SIMD) parallelization (Grama et al., 2003; Klimovitski, 2001). Using a 3.6 GHz Pentium 4 CPU, it takes 30 seconds to solve an APSP problem for $|V| = 4096$.

Finally, Srinivasan et al. (2006) show a cluster approach to parallelize the FW algorithm on a distributed memory machine. However, the performance does not scale well with the number of computing nodes, because the data size $|V|$ seems to be small for the deployed cluster. A speedup of 1.2 is observed on a 32-node system when using a graph with $|V| = 4096$.

With respect to the FW implementation, the timing results mentioned above are limited by the memory bandwidth rather than the arithmetic performance: 76.8, 36.8, and 80.1 GB/s (9.6, 4.6, and 10 GFLOPS) on the FPGA (Bondhugula et al., 2006), the CPU (Han et al., 2006), and the GPU (Katz and Kider, 2008), respectively. Similarly, the SSSP-based method can be regarded as a memory-intensive application rather than a compute-intensive application. Actually, it requires two operations and at least two memory accesses to update the cost of a vertex. Thus, considering the ratio of the memory bandwidth to the arithmetic performance, the SSSP-based method requires at least 1 B/FLOP while the GPU employed by Harish and Narayanan (2007) provides 0.25 B/FLOP. Therefore, the performance will be increased if we save the memory bandwidth between off-chip memory and SPs.

3 Compute Unified Device Architecture

CUDA (nVIDIA Corporation, 2008) is a development framework that allows us to write GPU programs without understanding the graphics pipeline. Using this framework, we can assume that the GPU is a SIMD machine that accelerates highly threaded applications by processing thousands of threads in parallel. The GPU program is generally called as kernel, which is launched from the CPU code to process threads in a SIMD fashion. The same kernel is executed for every thread but with different thread IDs to perform SIMD computation.

Figure 1 shows an overview of the GPU architecture. The GPU employs a hierarchical architecture that consists of several multiprocessors (MPs), each having stream processors (SPs) for processing threads. The important point here is that SPs within the same MP are allowed to share on-chip memory called shared memory. This memory hierarchy is useful to save the memory bandwidth between SPs and off-chip memory called device memory, because it can be used as a software cache shared by multiple SPs belonging to the same MP. Accordingly, a hierarchical structure is incorporated into threads to realize efficient data access. That is, threads are structured into equal-sized groups, each called a thread block. Developers have to write their kernel such that there is no data dependence between different thread blocks because each thread block will be independently assigned to an MP. Due to the same reason, the GPU does not have a mechanism that synchronizes all threads. Such global synchronization involves splitting the kernel into two pieces, which are then launched sequentially from the CPU.

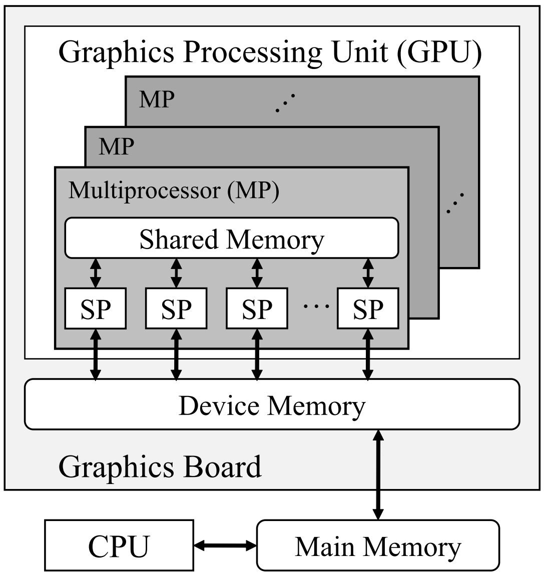


Figure 1: CUDA hardware model. SP denotes stream processor.

Each MP processes a thread block in the following way. Suppose that a block is assigned to an MP. The MP then splits the block into groups of threads called warps. The number of threads in a warp, which is defined as 32 threads in current hardware, is called as the warp size. Each of warps is then processed by the MP in a SIMD fashion. Therefore, branching threads in the same warp will divergent the warp. Such divergent warps (nVIDIA Corporation, 2008) degrade the performance because instructions must be serialized due to different control flows.

While shared memory is almost as fast as registers, device memory takes 400 to 600 clock cycles to access data. Therefore, the GPU architecture is designed to hide this latency with independent computation. This also explains why thread blocks must be independent. Such independent blocks are useful to allow MPs to continue computation by switching the block that has to wait data from device memory. Therefore, it is better to assign multiple thread blocks

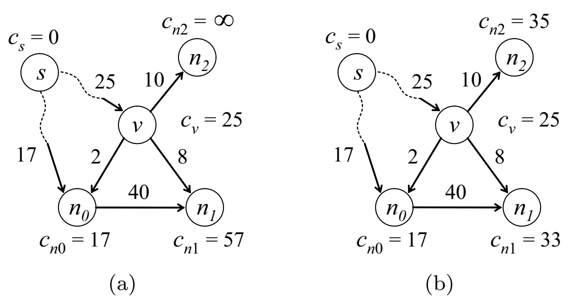


Figure 2: Cost minimization. (a) For each vertex v in the graph, (b) the costs of its neighbours n_0, n_1, and n_2 are updated in the scattering phase.

to every MP. However, memory resources such as shared memory and registers usually limit the number of thread blocks per MP.

Memory coalescing (nVIDIA Corporation, 2008) is also important to achieve the full utilization of the wide memory bus between device memory and SPs. Using this technique, the memory accesses issued from threads in a half-warp can be coalesced into a single access if the source/destination address satisfies an alignment requirement: a thread with ID N within the half-warp should access address $base + N$, where $base$ is a multiple of 16 bytes (nVIDIA Corporation, 2008). Note that this requirement is relaxed in recent GPUs, which automatically perform memory coalescing. Our algorithm assumes the strict requirement to be executable on older GPUs.

4 SSSP-based Iterative Algorithm

Harish and Narayanan (2007) compute the costs of APSPs in a directed graph $G = (V, E, W)$ with positive weights, where V is a set of vertices, E is a set of edges, and W is a set of edge weights on the graph G. In the following, let $|V|$ and $|E|$ be the number of vertices and that of edges, respectively. Given a graph G, the method computes an SSSP $|V|$ times with varying the source vertex $s \in V$. This iteration is sequentially processed by the CPU, but each SSSP problem is solved in parallel on the GPU.

To solve an SSSP problem, an iterative algorithm (Harish and Narayanan, 2007) is implemented using CUDA. This algorithm associates every vertex $v \in V$ with cost c_v, which represents the cost of the current shortest path from the source s to the destination v. The algorithm then minimizes every cost until converging to the optimal state. This cost minimization is done by processing two phases alternatively: the scattering phase and the checking phase. In the scattering phase, all vertices try to minimize the costs of their neighbours in parallel. Figure 2 illustrates how this minimization works for a single vertex v. After this, the checking phase confirms whether the previous scattering phase has changed the costs of vertices.

Figure 3 shows this algorithm. Firstly, the cost of every vertex $v \in V$ except the source s is initialized to infinity,

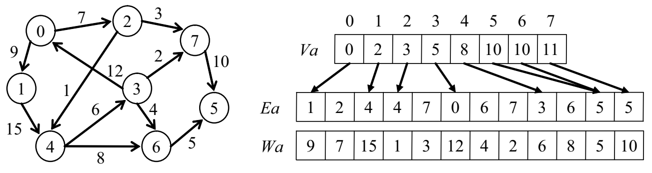


Figure 4: Adjacency list representation. Array Va stores the indices to the head of each adjacency list in Ea. Array Ea and Wa store adjacency lists of every vertex and edge weight, respectively.

which means that v is not reachable from s at the initial state. On the other hand, the cost is set to zero for the source s. The cost minimization then begins at line 4 for a set M of vertices, where M is the modification set, which contains vertices whose neighbour(s) have not yet reached to the optimal state. Given such a vertex $v \in M$, the algorithm updates the cost c_n at line 9, for every neighbour $n \in V$ such that $(v, n) \in E$. The updated cost here is temporally stored to a variable u_n in order to check convergence later at line 13. Vertices that have changed their costs are added to set M for further minimization (line 14). The iteration stops when M becomes empty.

This algorithm requires synchronization between the scattering phase and the checking phase (line 12). Otherwise, some processing elements might overwrite the updated cost u_v after u_v has been confirmed to be minimal. It also should be noted that the algorithm requires atomic instructions to correctly process the scattering phase. Since multiple processing elements can update the same cost u_n at the same time, we have to deal with the consistency of concurrent write access. Atomic instructions solve this issue but they are supported only on GPUs with compute capability 1.1 and higher (nVIDIA Corporation, 2008). If we lack this capability, the minimum cost u_n will be overwritten by a larger cost at line 9, resulting in a wrong result.

We now explain how Harish and Narayanan implement the algorithm on the GPU. As we mentioned earlier, there is no global synchronization mechanism in CUDA. Therefore, they develop two kernels, each for the scattering phase and for the checking phase. In both kernels, a thread is responsible for a vertex $v \in V$ in the graph. Thus, the cost minimization is parallelized using $|V|$ threads.

Figure 4 illustrates how a graph is represented in their kernels. They employ an adjacency list representation to store a graph in device memory. In this representation, each vertex data has a pointer to its adjacency list of edges. The adjacency list of vertex v here contains every neighbouring vertex $n \in V$ such that $(v, n) \in E$. Harish and Narayanan convert these lists into arrays Va, Ea, and Wa, which store vertex set V, edge set E, and weight set W, respectively. As shown in Fig. 4, element $Va[v]$ has an index to array Ea, where the head of the adjacency list of v exists. Since all adjacency lists are concatenated into array Ea of size $|E|$, the adjacency list of vertex v is stored from element $Va[v]$ to $Va[v+1]-1$ in Ea. Similarly, the weight

```
SSSP_Algorithm($s$, $V$, $E$, $W$)                          /* $s$: source vertex */
 1: initialize $c_v := \infty$ and $u_v := \infty$ for all $v \in V$   /* $u_v$: updated cost of vertex $v$ */
 2: $c_s := 0$                                                 /* $c_v$: current cost of vertex $v$ */
 3: $M := \{s\}$
 4: while $M$ is not empty do
 5:     for each vertex $v \in V$ in parallel do
 6:         if $v \in M$ then                                  /* Scattering phase */
 7:             remove $v$ from $M$
 8:             for each neighbouring vertex $n \in V$ such that $(v, n) \in E$ do
 9:                 $u_n := \min(c_n,\ c_v + w_{v,n})$          /* $w_{v,n}$: weight of edge $(v, n)$ */
10:             end for
11:         end if
12:         synchronization
13:         if $c_v > u_v$ then                                /* Checking phase */
14:             add $v$ to $M$
15:             $c_v := u_v$
16:         end if
17:     end for
18: end while
```

Figure 3: Iterative algorithm for finding an SSSP from the source vertex $s \in V$.

```
SSSP_Scattering_Kernel($Va$, $Ea$, $Wa$, $Ma$, $Ca$, $Ua$)
1: $v$ := threadID
2: if $Ma[v]$ = true then
3:     $Ma[v]$ := false
4:     for $i := Va[v]$ to $Va[v + 1] - 1$ do
5:         $n := Ea[i]$
6:         $Ua[n] := \min(Ua[n],\ Ca[v] + Wa[n])$
7:     end for
8: end if
```

Figure 5: Pseudocode of scattering kernel (Harish and Narayanan, 2007). This kernel is responsible for a single vertex v and updates the costs of its adjacent vertices.

of edge $Ea[i]$ is stored in $Wa[i]$, where $0 \le i \le |E| - 1$. In addition to the arrays mentioned above, they use additional arrays Ma, Ca, and Ua to store modification set M, current cost c_v, and updated cost u_v, respectively. Each of these arrays has $|V|$ elements and its index v corresponds to vertex v. They store these three arrays in device memory.

Figure 5 shows a pseudocode of the scattering kernel, which implements lines 6–11 in Fig. 3. This kernel is invoked for every thread t_v, which is responsible for vertex $v \in V$. After this kernel execution, the CPU launches the second kernel to process the checking phase. This checking kernel updates array Ma and also sets a flag to true if any updated cost is found. The CPU then checks this flag to determine if the iteration should be stopped or not. Thus, the flag prevents the CPU from scanning the entire array Ma.

5 Task Parallel Algorithm

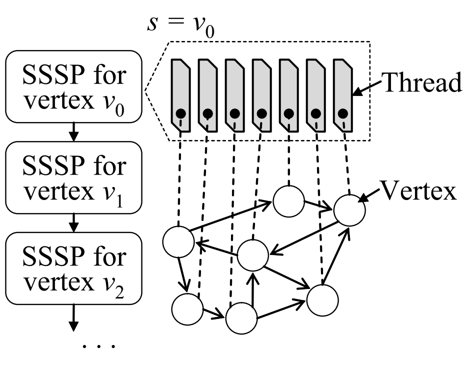
We now describe the proposed algorithm for accelerating the computation of APSPs on a directed, positively weighted graph G. Firstly, we present how our algorithm accelerates the cost computation of Harish's SSSP-based method. We then describe how the algorithm records the paths after the cost computation.

5.1 Cost Computation

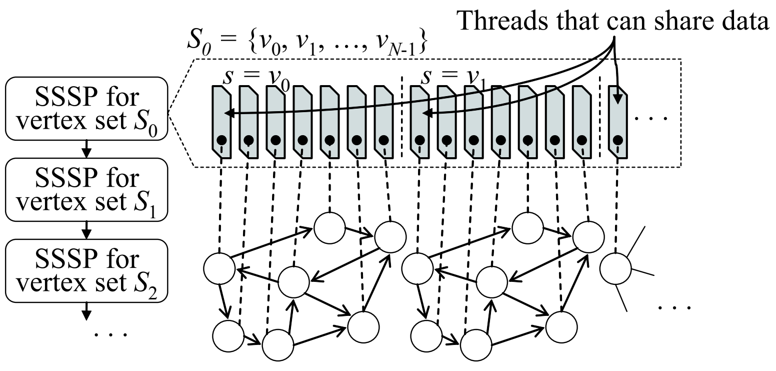
As shown in Fig. 6(b), our algorithm computes N tasks in parallel, where a task deals with an SSSP problem. This task parallel scheme allows us to share graph data between different tasks. Another important benefit is that it allows the kernel to generate more threads at a launch. This leads to an efficient execution on the GPU, which employs a massively multithreaded architecture. Since the algorithm we use for a single SSSP problem is the same one developed by Harish and Narayanan (2007), we explain here how tasks are grouped to share the graph data.

Let p_s denote the SSSP problem with the source vertex $s \in V$. The APSP problem consists of $|V|$ SSSP problems $p_0, p_1, \ldots, p_{|V|-1}$ and there is no data dependence between them. Therefore, we can pack any N problems into a group to solve the group in parallel, where $1 \le N \le |V|$. Thus, the k-th group contains N SSSP problems $p_{kN}, p_{kN+1}, \ldots, p_{(k+1)N-1}$, where $0 \le k \le \lceil |V|/N \rceil - 1$. Let S_k denote the set of source vertices in the k-th group of N SSSP problems, where $0 \le k \le \lceil |V|/N \rceil - 1$. The proposed scheme then computes SSSPs from every source $s \in S_k$ on the GPU while it invokes this computation $\lceil |V|/N \rceil$ times sequentially from the CPU. We assign a vertex to a thread as Harish and Narayanan do in their algorithm. Accordingly, our kernel processes $N|V|$ threads in parallel while the previous kernel does $|V|$ threads, as shown in Fig. 6.

Similar to Harish's algorithm, our algorithm consists of the scattering phase and the checking phase, as shown in Fig. 7. However, our algorithm differs from the previous algorithm with respect to the use of shared memory in



(a)



Threads that can share data
$S_0 = \{v_0, v_1, \ldots, v_{N-1}\}$
$s = v_0$
$s = v_1$
SSSP for vertex set S_0
SSSP for vertex set S_1
SSSP for vertex set S_2
...

(b)

Figure 6: Comparison of parallelization scheme between (a) previous method (Harish and Narayanan, 2007) and (b) proposed method. Our kernel solves N SSSP problems at a time. The graph data is shared between threads that are responsible for the same vertex but in different SSSP problems.

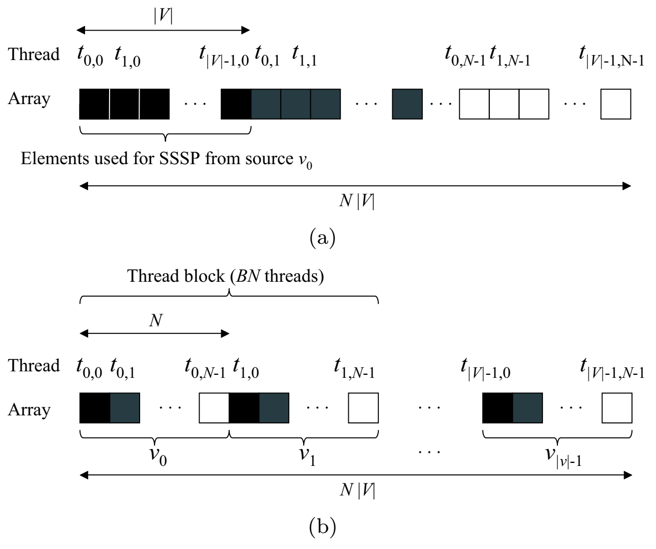
```
N_SSSPs_Algorithm($S_k$, $V$, $E$, $W$)                          /* $S_k$: set of source vertices */
 1: initialize $c_{v,s} := \infty$ and $u_{v,s} := \infty$ for all $v \in V$ and $s \in S_k$     /* $u_{v,s}$: updated cost of vertex $v$ in problem $p_s$ */
 2: initialize $c_{s,s} := 0$ for all $s \in S_k$                 /* $c_{v,s}$: current cost of vertex $v$ in problem $p_s$ */
 3: add pair $\langle s, s\rangle$ to set $M$ for all $s \in S_k$
 4: **while** $M$ is not empty **do**
 5:     **for** each vertex $v \in V$ and each source $s \in S_k$ in parallel **do**   /* Scattering phase */
 6:         **if** $\langle v, s\rangle \in M$ **then**
 7:             remove $\langle v, s\rangle$ from $M$
 8:             **for** each neighbouring vertex $n \in V$ such that $(v, n) \in E$ **do**
 9:                 $u_{n,s} := \min(c_{n,s},\ c_{v,s} + w_{v,n})$
10:             **end for**
11:         **end if**
12:         synchronization
13:         **if** $c_{v,s} > u_{v,s}$ **then**                                /* Checking phase */
14:             add $\langle v, s\rangle$ to $M$
15:             $c_{v,s} := u_{v,s}$
16:         **end if**
17:     **end for**
18: **end while**
```

Figure 7: Algorithm for finding SSSPs from each s of source vertices S_k.

the scattering phase. Let $t_{v,s}$ be the thread, which is responsible for vertex $v \in V$ in problem p_s, where $s \in V$. In our algorithm, thread $t_{v,s}$ tries to update the cost $c_{n,s}$ (variable $u_{n,s}$ at line 9 in Fig. 7), which represents the cost of neighbouring vertex $n \in V$ in problem p_s. The graph data that can be shared between threads is edge (v, n) at line 8 and weight $w_{v,n}$ at line 9, because both variables do not depend on the source vertex s. In order to share such s-independent data between threads, we structure a thread block such that it includes N threads $t_{v,kN}, t_{v,kN+1}, \ldots, t_{v,(k+1)N-1}$, which are responsible for the same vertex v but in different problems. Figure 8 shows the data structure more precisely. Note that every thread block contains a multiple B of such N threads to increase the block size for higher performance. Thus, such threads can save the memory bandwidth if they update the costs of their neighbours. However, it requires additional copy operations to duplicate data to shared memory. Therefore, threads might degrade the performance if such common access rarely occurs during execution.

With respect to the graph representation, our kernel uses a slightly different data structure from the previous kernel. We use the same arrays Va, Ea, and Wa, but they are partially shared between threads as mentioned before. The remaining arrays Ca, Ua, and Ma are separately allocated for every problem p_s, so that these arrays have $N|V|$ elements as shown in Fig. 8(b). The reason why we need such larger arrays is that the GPU does not support dynamic memory allocation though each SSSP problem can have a different number of unoptimized vertices at each iteration. Therefore, we simply use N times more arrays to provide dedicated arrays to each of N problems.

This decision might prevent us from using small shared memory for arrays Ca, Ua, and Ma. However, it is not a critical problem because each element in these arrays is accessed only by its responsible thread. Instead, as shown in Fig. 8(b), it is important to interleave array Ma to allow threads $t_{v,0}, t_{v,1}, \ldots, t_{v,N-1}$ in the same thread block to



$|V|$
Thread
$t_{0,0}$ $t_{1,0}$ $t_{|V|-1,0}$ $t_{0,1}$ $t_{1,1}$ $t_{0,N-1}$ $t_{1,N-1}$ $t_{|V|-1,N-1}$
Array
Elements used for SSSP from source v_0
$N|V|$
(a)
Thread block (BN threads)
N
Thread
$t_{0,0}$ $t_{0,1}$ $t_{0,N-1}$ $t_{1,0}$ $t_{1,N-1}$ $t_{|V|-1,0}$ $t_{|V|-1,N-1}$
Array
v_0 v_1 $v_{|v|-1}$
$N|V|$
(b)

Figure 8: Array interleaving for coalesced memory accesses. (a) A straightforward layout for Ma, Ca, and Ua stores all data for every SSSP problems into contiguous sequences. (b) The same vertices but for different problems are stored in a contiguous address space. $B = 2$, in this case.

access array elements in a coalesced manner. Similarly, we arrange arrays Ca and Ua into the same structure to realize coalesced accesses in the checking kernel. This also contributes to simplify addressing for data accesses. However, it is not easy to realize coalesced accesses in the scattering kernel because every thread updates different elements of Ua.

Figure 9 shows a pseudocode of our scattering kernel. As we mentioned before, we use shared memory for vertex set V, edge set E, and weight set W: arrays $from$, to, es, and ws at lines 6 and 12. In addition, we also use shared variable ms to perform reductions of modification set M at lines 7–10. That is, set M is shared among N threads, which are responsible for the same vertex v but for different problems. This means that all of such N threads must be engaged in data duplication if any of them has not yet finished the minimization. This cooperative strategy is essential to increase the number of coalesced accesses on the GPU. For memory-intensive applications, we think that SPs must be used for parallelization of memory accesses, namely coalesced accesses, rather than that of computation. Note that this shared space is used only for this purpose, so that we write the new set M directly to device memory at line 22.

Similar to Harish's algorithm, our method uses a flag to check the convergence of cost computation. Since this flag has to be shared between all threads, we store the flag in global memory. However, we initialize per-block flags on shared memory at the beginning of the checking kernel. This duplication minimizes the overhead of serialization, which can occur when many threads set the flag at the same time. After checking the convergence, one of threads

```
N_SSSPs_Scattering_Kernel($Va, Ea, Wa, Ma, Ca, Ua, N$)
 1: $v$ := threadID **div** $N$   /* vertex ID */
 2: $s$ := threadID **mod** $N$ /* source (problem) ID */
 3: /* vertex ID in arrays $Va$, $Ma$ and $Ca$ */
 4: $vg$ := blockID $* B + v$
 5: /* Arrays in shared memory */
 6: **__shared__** $ms[B]$
 7: $ms[v]$ := false
 8: **if** $Ma[vg, s]$ = true **then**
 9:     $ms[v]$ := true
10: **end if**
11: **if** $ms[v]$ = true **then**
12:     **__shared__** $from[N]$, $to[N]$, $es[B, N]$, $ws[B, N]$
13:     /* Copy data to shared memory */
14:     $from[v] := Va[vg]$
15:     $to[v] := Va[vg + 1]$
16:     $neighbours := to[v] - from[v]$
17:     **if** $s < neighbours$ **then**
18:         $es[v, s] := Ea[from[v] + s]$
19:         $ws[v, s] := Wa[from[v] + s]$
20:     **end if**
21:     **if** $Ma[vg, s]$ = true **then**
22:         $Ma[vg, s]$ := false
23:         **for** $i$ := 0 **to** $neighbours - 1$ **do begin**
24:             $n := es[v, i]$
25:             $Ua[n, s] := \min(Ua[n, s],\ Ca[vg, s] + ws[v, i])$
26:         **end for**
27:     **end if**
28: **end if**
```

Figure 9: Pseudocode of proposed scattering kernel. This kernel solves N SSSP problems in parallel.

in each thread block writes the flag back to global memory.

5.2 Task Size Determination

The proposed kernel requires arrays of size $(3N + 1)|V| + 2|E|$ in device memory while the previous kernel requires those of $4|V| + 2|E|$ size. Therefore, our kernel cannot deal with larger graphs as compared with the previous kernel though it has an advantage over the FW algorithm. Thus, it is better to minimize N to compute APSPs for larger graphs. In contrast, we should maximize N to receive the timing benefit of shared memory. Therefore, selection of N is an important issue in our algorithm.

There are two requirements that should be considered when determining N. Firstly, N must be smaller than the maximum size of a thread block to share data between threads in the same thread block. Since the maximum size is currently 512 threads (nVIDIA Corporation, 2008), this requirement will be satisfied when $N \leq 512$. Secondly, N must be a multiple of warp size (32) to achieve coalesced accesses and to reduce divergent warps. The second requirement guarantees coalesced accesses to Ma because every warp will contain threads that process the same vertex v and access data in a contiguous address. Furthermore, such threads have the same number of iterations at line 23 in Fig. 9 because they have to update the costs of the same neighbours. This is useful to avoid divergent warps at line 23. However, we cannot eliminate all divergent warps in the kernel because the branch instructions at lines 17 and 21 depend on each thread.

According to the design considerations mentioned above, we currently use $N = 32$ in our kernel. This configuration allows us to eliminate synchronization from the scattering kernel, because N is equivalent to the warp size. That is, all threads that must be synchronized each other belong to the same warp, where SIMD instructions guarantee implicit synchronization between threads. Otherwise, threads in the same block have to synchronize each other after loading data in shared memory. Finally, we experimentally determine to use $B = 4$. Thus, thread blocks in our kernel have 128 threads.

5.3 Path Recording

Our path recording framework is based on a backtracing strategy that specifies the paths after the cost computation presented in Section 5.1. That is, this strategy computes the shortest path from the destination vertex $d \in V$ to the source vertex $s \in V$. To realize such a backtracing procedure, our method records which vertex has determined the final costs after the cost minimization. Suppose that we compute the shortest path from the source vertex s to the destination vertex n_1 in Fig. 2(b). Our algorithm starts from vertex n_1 and finds that the neighbouring vertex v determines the cost of vertex n_1. In other words, the shortest path consists of vertex v, which is the parent of vertex n_1. The algorithm then moves to parent vertex v and iterates this backtracing procedure until reaching the source vertex s. This backtracing procedure can be processed in $O(|V|)$ time for each path, so that we compute this procedure on the CPU.

In order to allow the CPU to perform backtracing, our algorithm records all of parents after the cost computation. For each source vertex s, we store parent vertices in array Pa of size $|V|$ such that element $Pa[v, s]$ has the parent of vertex v. This is done by an additional kernel, namely the recording kernel, which is invoked after the convergence of cost minimization. Since our method solves N SSSPs at a time, the recording kernel also records all parents needed for N SSSPs. A thread in the recording kernel is responsible for a vertex $v \in V$ as same as the scattering kernel. Each thread $t_{v,s}$ records a direct predecessor $u \in V$ as its parent such that u satisfies $c_{v,s} = c_{u,s} + w_{u,v}$. To find such a parent, thread $t_{v,s}$ simply checks every direct predecessor $u \in V$ such that $(u, v) \in E$. Since this operation requires predecessors for each vertex, we use inverted graph $G' = (V, E', W')$ to simplify the operation, where E' contains the inverted edge (v, u) of edge $(u, v) \in E$ and W' is a set of corresponding edge weights. Therefore, our method further requires $|V| + 2|E|$ space to store the adjacency list of G'. With respect to array Pa, we reuse the memory space for array Ua, so that we do not allocate additional space.

After recording all parents in Pa, the CPU backtraces the shortest path from the destination vertex d to the source vertex s (Cormen et al., 2001). Firstly, the CPU refers $Pa[d, s]$ to find the parent vertex u of vertex d. The CPU then records u and finds the parent vertex of u from $Pa[u, s]$. In this way, the CPU recursively backtraces the path until reaching the source vertex s such that $Pa[u, s] = s$.

6 Experimental Results

We evaluate the performance of our method by comparing it with other two methods: the SSSP-based method (Harish and Narayanan, 2007) and the Dijkstra-based method (Dijkstra, 1959) running on the GPU and the multi-core CPU, respectively. Note that the comparison is done with respect to the cost computation. After this comparison, we also evaluate the overhead of path recording.

The Dijkstra-based method is accelerated using a binary heap. Furthermore, we parallelize the method using all CPU cores. In more detail, we do not parallelize this algorithm but run a serial program with different sources on each CPU core, because there is no efficient parallelization for Dijkstra's algorithm. Thus, a CPU thread is responsible for solving SSSP problems for a subset of source vertices. CPU threads are managed using OpenMP (Chandra et al., 2000).

We execute GPU-based implementations on a PC with an nVIDIA GeForce 8800 GTS (G92 architecture). This graphics card has 512 MB of device memory and 16 MPs, each having 8 SPs. It also should be mentioned that G92 architecture supports atomic instructions to correctly process the scattering kernel. CUDA-based implementations run on Windows XP with CUDA 2.0 and driver version 178.28. The Dijkstra-based method is executed on an Intel Xeon 5440 2.83 GHz quad-core CPU, 12 MB L2 cache, and 8 GB RAM.

6.1 Performance Scalability on Graph Size

We investigate the performance with varying the graph size in terms of the number $|V|$ of vertices and that $|E|$ of edges. The graph data we used in this experiment is random graphs generated by a tool (DIMACS implementation challenge). Using this tool, we generate graphs such that every weight has an integer value within the range $[1, w_{max}]$, where $w_{max} = |V|$.

Figure 10 shows timing results obtained with varying the number $|V|$ of vertices. During measurements, the number $|E|$ of edges is fixed to $|E| = 4|V|$, meaning that a single vertex has four outgoing edges in average. In addition to the three methods mentioned before, we also implement an unshared version of the proposed method that uses device memory instead of shared memory. Due to the capacity of on-board memory, our method fails to solve the problem for $|V| = 1.2$M while Harish's method can deal with $|V| = 10.7$M on our machine.

As compared with the previous SSSP-based method, the proposed method achieves the best speedup of 13 when

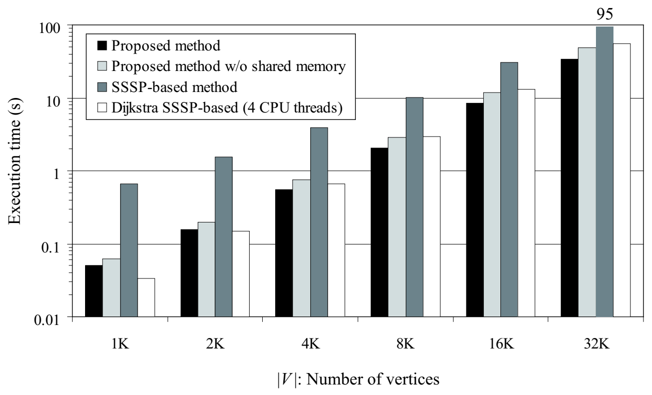


Figure 10: Timing results for random graphs with a different number $|V|$ of vertices. Results are presented in seconds.

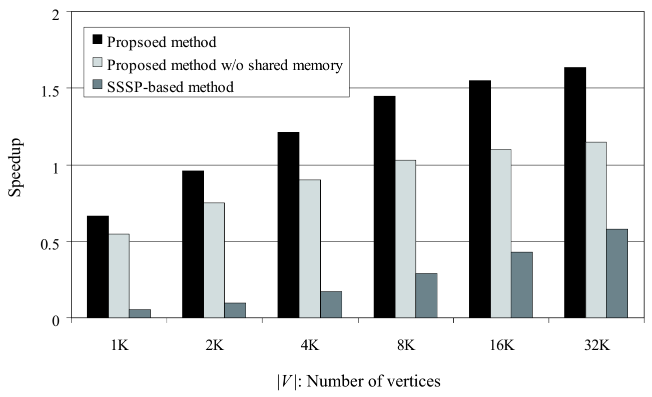


Figure 11: Speedup over the SSSP-based implementation running on the CPU.

$|V| = 1\text{K}$. In particular, our method runs more efficiently than the previous method when the graph has fewer vertices. The reason for this is that our method has many threads that can be used for hiding the memory latency. For example, it generates 32K threads for 16 MPs when $|V| = 1\text{K}$, which is equivalent to 2K threads per MP. In contrast, the previous method has 64 threads per MP. Thus, more threads belonging to different thread blocks are assigned to every MP in our method. Such multiple assignments are essential to hide the latency with other computation, making the GPU-based methods faster than the CPU-based Dijkstra method, which is faster than the previous method (Figure 11).

Since the proposed method solves N SSSP problems at a time, the number of kernel launches is reduced to approximately $1/N$ as compared with the previous method. This implies that we can reduce the synchronization overhead needed at the end of a kernel execution. This reduction effects are observed clearly when $|V|$ is small, where synchronization cost accounts for a relatively large portion of total execution time. Thus, less synchronization allows threads to have shorter waiting time at the kernel completion.

By comparing the proposed two methods, the speedup

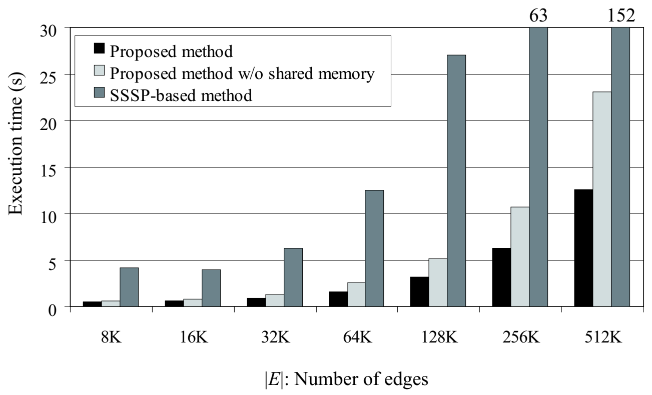


Figure 12: Timing results with different number $|E|$ of edges. The number $|V|$ of vertices is fixed to $|V| = 4K$.

achieved by shared memory ranges from a factor of 1.2 to that of 1.4. This speedup is achieved by shared memory, which eliminates approximately 16% of the data access between SPs and device memory compared to the proposed method without shared memory. On the other hand, the unshared version of the proposed method is at least 1.9 times faster than the previous method. Thus, the acceleration is mainly achieved by the task parallel scheme rather than shared memory. However, the speedup achieved by the task parallel scheme decreases as $|V|$ increases, because the increase allows the previous method to assign many threads to MPs, as we do in our method. In contrast, the speedup given by shared memory increases with $|V|$. Therefore, we think that shared memory is useful to deal with larger graphs.

Figure 12 shows timings results for graphs with a different number $|E|$ of edges. Every graph has the same number of vertices: $|V| = 4\text{K}$. These results indicate that all methods increase the execution time with $|E|$. In particular, the shared version of the proposed method shows better acceleration results as $|E|$ increases. Thus, shared memory can effectively reduce the number of data reads from global memory for larger graphs with many edges and vertices. It also should be noted that the proposed method uses $O(BN)$ space in shared memory, which is independent from the graph size $|V|$ and $|E|$. Thus, the graph size is limited by the capacity of device memory rather than that of shared memory.

6.2 Performance Stability on Graph Attributes

Figure 13 shows the results obtained using graphs with a different maximum weight w_{max}. All graphs have the same numbers of vertices and edges: $|V| = 4\text{K}$ and $|E| = 16\text{K}$. These results show that the execution time increases with w_{max}. However, this increasing behaviour is not so sharp as compared with that shown in Fig. 12, because w_{max} does not directly affect the execution time. The increase of w_{max} means that the graph has edge weights with a larger distribution. In such a case, we need more iterations to minimize the costs, because shorter paths hav-

Table 1: Timing results for some graph topologies. Results are presented in seconds.

Method	Platform	Topology			
		Random	Power law	Ring	Complete
SSSP-based (Harish and Narayanan, 2007)	GPU	4.39	6.37	687	5730
Proposed w/o shared memory	GPU	0.884	1.73	53.7	175
Proposed	GPU	0.669	0.942	58.2	77.3
Dijkstra SSSP-based (4 threads)	CPU	0.835	0.437	0.0962	158

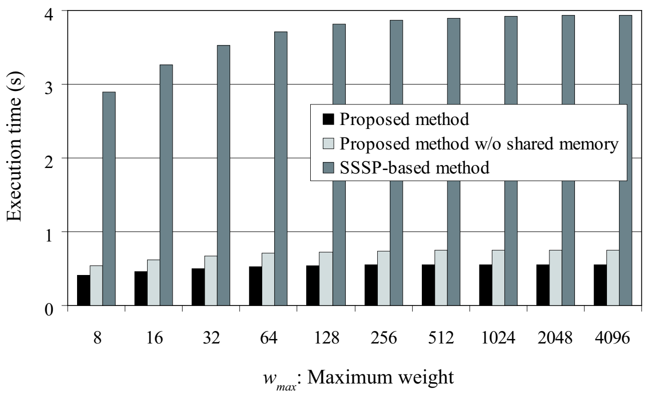


Figure 13: Timing results with different maximum weight w_{max}. The number $|V|$ of vertices and the number $|E|$ of edges are fixed to $|V| = 4K$ and $|E| = 16K$.

ing many edges can overwrite costs that have updated by longer paths having a few edges with larger weights. In addition, a change of a cost of vertex u affects the cost of vertex v, where the temporal shortest paths from source vertex to v pass through u.

We next investigate the performance on different topologies. Table 1 shows timing results for various graph topologies. Every graph data has the same number $|V| = 4677$ of vertices but with different topologies: a random graph (DIMACS implementation challenge), a power law graph (Bader and Madduri; Chakrabarti et al., 2004), a ring, and a complete graph. The random graph and the power law graph have $|E| = 16\text{K}$ edges with $w_{max} = 4096$. The remaining two graphs have the same weight $w_{max} = 1$ for all edges. The power law graph has many low-degree vertices but also has less high-degree vertices. In more detail, the maximum and average outdegrees in our graph are 834 and 3.5, respectively, and 84% of vertices have a lower degree than 3.

In this table, we can see that all methods significantly vary their performance depending on the graph topology. The GPU-based methods outperform the CPU-based method with respect to the random graph and the complete graph. However, the CPU-based method provides the fastest result for the ring graph. This is due to the employed parallel algorithm rather than the GPU implementation, because every vertex in the ring graph has a single outgoing edge, which serializes the scattering phase. Thus, the problem is left on the parallel algorithm rather than the implementation.

For the power law graph, the GPU-based methods basically decrease their performance as compared with that for the random graph. We think that this is due to the load imbalance in the scattering kernel, because the power law graph has wide outdegrees ranging from 0 to 834. Thus, the most loaded thread has to update 834 neighbours while 84% of threads do this only for at most 3 neighbours. Such a load imbalanced situation will increase the kernel execution time. Actually, the power law graph requires 28% less kernel launches than the random graph. Thus, the GPU-based methods can vary the performance according to the graph degree. The workload will be balanced if every vertex has the same number of outgoing vertices. Otherwise, a sorting mechanism will help us improve the performance.

The previous method takes 156 times longer time to solve the ring graph as compared with the random graph. This increasing time can be explained by the number of kernel launches. The ring graph has a unique topology where every vertex has a single edge. Since the previous kernel updates the neighbouring cost of a single vertex, it has to launch the kernel $|V| = 4677$ times to compute an SSSP in this case. In contrast, it requires only 23 launches per SSSP for the random graph. Thus, the ring graph requires 203 times more launches than the random graph, increasing the execution time.

The proposed method also takes longer time for the ring graph but it is approximately 12 times faster than the previous method. We expect a 32-fold speedup over the previous method, because our method simultaneously processes $N = 32$ vertices using 128 SPs. Thus, there is a gap between measured performance and expected performance. This gap can be explained by the branch overhead and the duplication overhead. Our method has more branch instructions in the scattering kernel due to the use of shared memory. Moreover, it requires duplication operations to use shared memory, but this overhead can degrade the performance as we mentioned in Section 5.1. The duplication overhead also explains why the unshared version outperforms the shared version when processing the ring graph.

The complete graph shows the worst result among the four topologies used in this experiment. As compared with the random graph, the previous method and our method spend 1310 times and 116 times longer time for the complete graph, respectively. In this graph, every vertex has $|V| - 1$ neighbours, so that $|V| - 1$ repetitions are required to write new costs of neighbours in the scattering kernel. Since each thread sequentially processes these repetitions, the scattering kernel spends relatively longer

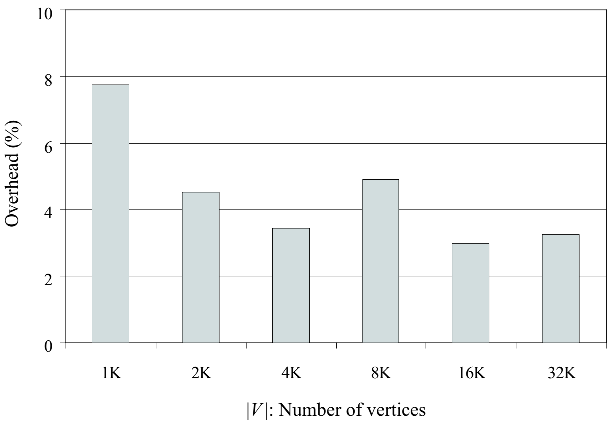


Figure 14: Overhead of path recording for random graphs with a different number $|V|$ of vertices. Overhead explains the increased time due to the recording kernel called after cost computation.

time. Although these repetitions are common to both of the proposed and previous kernels, our kernel demonstrates a higher tolerance to these repetitions. This tolerance is given by shared memory because each thread refers $|V|-1$ elements of Ea and Wa, which is duplicated in on-chip shared memory.

6.3 Overhead of Path Recording

Finally, we analyze the overhead of path recording. Figure 14 shows the ratio of the recording kernel to the remaining two kernels in terms of execution time. That is, the execution time in Fig. 10 will be increased by the ratio in Fig. 14 if APSPs are recorded after the cost computation. Note that the overhead does not include the execution time spent for backtracing on the CPU. According to Fig. 14, the overhead of path recording ranges from 3.0% to 7.7%. This overhead is mainly due to the recording kernel. In addition to this additional kernel execution, we also need to send array Pa back to the main memory.

7 Conclusion

In this paper, we have proposed a fast algorithm for finding APSPs using the CUDA-compatible GPU. The proposed algorithm is based on Harish's SSSP-based method and increases the performance by on-chip shared memory. We exploit the coarse-grained task parallelism in addition to the fine-grained data parallelism exploited by the previous method. This combined parallelism makes it possible to share graph data between processing elements in the GPU, saving the bandwidth between off-chip memory and processing elements. It also allows us to run more threads with less kernel launches, leading to an efficient method for highly multithreaded architecture of the GPU. The method is also capable of recording shortest paths as well as their costs.

The experimental results show that the proposed method is 2.8–13 times faster than the previous SSSP-based method. As compared with the previous method, the task parallel scheme demonstrates higher performance for smaller graphs. However, this advantage becomes smaller when dealing with larger graphs with more vertices. In contrast, shared memory increases its effects for larger graphs with more vertices and edges. With respect to the graph topology, we show that the ring graph serializes both the previous and proposed methods. We also demonstrate that both methods vary their performance according to the graph degree. The overhead of path recording is at most 7.7% for random graphs.

One future work is to deal with large graphs that cannot be stored entirely on device memory. Such large graphs involve data decomposition due to the lack of memory capacity.

ACKNOWLEDGMENT

This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University.

REFERENCES

9th DIMACS implementation challenge - Shortest paths. `http://www.dis.uniroma1.it/~challenge9/download.shtml`.

Bader, D. A. and Madduri, K. GTgraph. `http://www.cc.gatech.edu/~kamesh/GTgraph/`.

Bleiweiss, A. (2008) 'GPU accelerated pathfinding', *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware (GH'08)*, pages 65–74.

Bondhugula, U., Devulapalli, A., Dinan, J., Fernando, J., Wyckoff, P., Stahlberg, E., and Sadayappan, P. (2006) 'Hardware/software integration for FPGA-based all-pairs shortest-paths', *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines (FCCM'06)*, pages 152–164.

Chakrabarti, D., Zhan, Y., and Faloutsos, C. (2004) 'R-MAT: A recursive model for graph mining', *Proc. 4th SIAM Int'l Conf. Data Mining (SDM'04)*, pages 442–446.

Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., and Menon, R. (2000) *Parallel Programming in OpenMP*. Morgan Kaufmann, San Mateo, CA.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2001) *Introduction to Algorithms*. The MIT Press, second edition.

Dijkstra, E. W. (1959) ‘A note on two problems in connexion with graphs’, *Numerische Mathematik*, 1:269–271.

Floyd, R. W. (1962) ‘Algorithm 97: Shortest path’, *Communications of the ACM*, 5(6):345.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003) *Introduction to Parallel Computing.* Addison-Wesley, Reading, MA, second edition.

Han, S.-C., Franchetti, F., and Püshel, M. (2006) ‘Program generation for the all-pairs shortest path problem’, *Proc. 15th Int’l Conf. Parallel Architectures and Compilation Techniques (PACT’07)*, pages 222–232.

Harish, P. and Narayanan, P. J. (2007) ‘Accelerating large graph algorithms on the GPU using CUDA’, *Proc. 14th Int’l Conf. High Performance Computing (HiPC’07)*, pages 197–208.

Hart, P., Nilsson, N., and Raphael, B. (1968) ‘A formal basis for the heuristic determination of minimum cost paths’, *IEEE Trans. Systems Science and Cybernetics*, 4(2):100–107.

Jayadevaprakash, N., Mukhopadhyay, S., and Palakal, M. (2005) ‘Generating association graphs of non-cooccurring text objects using transitive methods’, *Proc. 20th ACM Symp. Applied computing (SAC’05)*, pages 141–145.

Katz, G. J. and Kider, J. T. (2008) ‘All-pairs shortest-paths for large graphs on the GPU’, *Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symp. Graphics Hardware (GH’08)*, pages 47–55.

Kim, S.-S. and Lee, J.-H. (1999) ‘A study on design of dynamic route guidance system using forecastedtravel time based on GPS data and modified shortest path algorithm’, *Proc. 2nd IEEE/IEEJ/JSAI Int’l Conf. Intelligent Transportation Systems (ITSC’99)*, pages 44–48.

Klimovitski, A. (2001) Using SSE and SSE2: Misconceptions and reality. In *Intel Developer Update Magazine.*

Micikevicius, P. (2004) ‘General parallel computation on commodity graphics hardware: Case study with the all-pairs shortest paths problem’, *Proc. Int’l Conf. Parallel and Distributed Processing Techniques and Applications (PDPTA’04)*, volume 3, pages 1359–1365.

Nakaya, A., Goto, S., and Kanehisa, M. (2001) ‘Extraction of correlated gene clusters by multiple graph comparison’, *Genome Informatics*, 12:34–43.

nVIDIA Corporation (2008) CUDA Programming Guide Version 2.0.

Okuyama, T., Ino, F., and Hagihara, H. (2008) ‘A task parallel algorithm for computing the costs of all-pairs shortest paths on the CUDA-compatible GPU’, *Proc. 6th Int’l Symp. Parallel and Distributed Processing with Applications (ISPA’08)*, pages 284–291.

Owens, J. D., Luebke, D., Govindaraju, N., Harris, M., Krüger, J., Lefohn, A. E., and Purcell, T. J. (2007) ‘A survey of general-purpose computation on graphics hardware’, *Computer Graphics Forum*, 26(1):80–113.

Shenoy, N. (1997) ‘Retiming: theory and practice’, *Integration, the VLSI J.*, 22(1/2):1–21.

Srinivasan, T., Balakrishnan, R., Gangadharan, S. A., and Hayawardh, V. (2006) ‘A scalable parallelization of all-pairs shortest path algorithm for a high performance cluster environment’, *Proc. 13th Int’l Conf. Parallel and Distributed Systems (ICPADS’07)*, volume 1. CD-ROM (8 pages)

Venkataraman, G., Sahni, S., and Mukhopadhyaya, S. (2000) ‘A blocked all-pairs shortest-path algorithm’, *Proc. 7th Scandinavian Workshop Algorithm Theory (SWAT’07)*, pages 419–432.

Warshall, S. (1962) ‘A theorem on boolean matrices’, *J. ACM*, 9(1):11–12.