2008 International Symposium on Parallel and Distributed Processing with Applications

A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-compatible GPU

Tomohiro Okuyama Fumihiko Ino Kenichi Hagihara

Graduate School of Information Science and Technology, Osaka University
1-3 Machikaneyama, Toyonaka, Osaka 560-8531, Japan

Abstract

This paper proposes a fast method for computing the costs of all-pairs shortest paths (APSPs) on the graphics processing unit (GPU). The proposed method is implemented using compute unified device architecture (CUDA), which offers us a development environment for performing general-purpose computation on the GPU. Our method is based on Harish's iterative algorithm that computes the cost of the single-source shortest path (SSSP) for every source vertex. We present that exploiting task parallelism in the APSP problem allows us to efficiently use on-chip memory in the GPU, reducing the amount of data being transferred from relatively slower off-chip memory. Furthermore, our task parallel scheme is useful to exploit a higher parallelism, increasing the efficiency with highly threaded code. As a result, our method is 3.4–15 times faster than the prior method. Using on-chip memory, our method eliminates approximately 20% of data loads from off-chip memory.

1 Introduction

The all-pairs shortest path (APSP) problem is to find paths with the minimum costs for all source-destination pairs in a graph. The cost here is given by the sum of the weights of edges composing the path. Finding such paths is one of the basic operations in graph theory. This fundamental problem plays a key role in many applications in a wide range of fields, such as geographical information systems, networking systems, intelligent transportation systems, and bioinformatics applications. In more detail, there are some practical applications [6] that require the costs instead of the paths.

One challenging issue in solving the APSP problem is that it requires a large amount of computation to find APSPs. For example, the Floyd-Warshall (FW) algorithm finds APSPs for a weighted, directed graph in $O(|V|^3)$ time, where $|V|$ represents the number of vertices in a graph. Therefore, many researchers have proposed fast methods using various accelerators, such as graphics processing units (GPUs) [4, 5], field-programmable gate arrays (FPGAs) [2], and clusters [9].

To the best of our knowledge, Harish et al. [4] show the fastest results by using the GPU [8]. The GPU is off-the-shelf hardware designed to accelerate graphics tasks, such as gaming and rendering. In recent years, this hardware rapidly increases the computational performance with a wide memory bus and hundreds of processing elements, called stream processors (SPs). In addition, nVIDIA has released a development framework, called compute unified device architecture (CUDA) [7], which enhances this special-purpose hardware by allowing C-like programs to be executed on the nVIDIA GPU.

Using the CUDA framework, Harish et al. [4] implement two algorithms for the APSP problem: the FW algorithm and an iterative algorithm that repeatedly computes single source shortest paths (SSSPs) with varying the source vertex. They show that the SSSP-based algorithm is approximately six times faster than the FW algorithm. However, this prior algorithm can be further improved to achieve a full utilization of memory resources available in the GPU. For example, only off-chip memory is used because there is no common data accessed simultaneously from multiple SPs. Thus, higher speed (but small) on-chip shared memory can be used to save the bandwidth between off-chip memory and SPs.

In this paper, we propose a task parallel algorithm capable of exploiting on-chip memory to accelerate the cost computation of APSPs in a graph. Our algorithm is based on Harish's iterative algorithm [4] but employs a different parallelization scheme in order to save the bandwidth between off-chip memory and SPs. To achieve this, 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 use-

978-0-7695-3471-8/08 $25.00 © 2008 IEEE
DOI 10.1109/ISPA.2008.40

ful 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.

The rest of the paper is organized as follows. We begin in Section 2 by introducing related work. Section 3 shows a brief overview of CUDA and Section 4 summarizes the prior SSSP-based method. Section 5 describes our method and Section 6 shows experimental results. Finally, Section 7 concludes the paper.

2 Related Work

Harish et al. [4] 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 the APSP costs for a graph with $|V| = 3072$ vertices. The speedup over the CPU-based FW implementation reaches a factor of 17. With respect to memory consumption, their 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 $|V| = 30720$ 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.

Micikevicius [5] 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. [2]. They implement a tiled version of the FW algorithm and develop a model to predict the performance for larger FPGAs. As compared with a CPU-based method running on 2.2 GHz Opteron, their method reduces execution time for $|V| = 16384$ from approximately four hours to 15 minutes, achieving a speedup of 15.2X.

An automated tuning approach is proposed by Han et al. [3] to accelerate the FW algorithm on the CPU. Their method is optimized by cache blocking and SIMD vectorization. Using a 3.6 GHz Pentium 4 CPU, it takes 30 seconds to solve the APSP problem for $|V| = 4096$.

Finally, Srinivasan et al. [9] show a cluster approach to parallelize the FW algorithm on a distributed memory machine. However, their method 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 1.2X is observed on a 32-node system when using a graph with $|V| = 4096$.

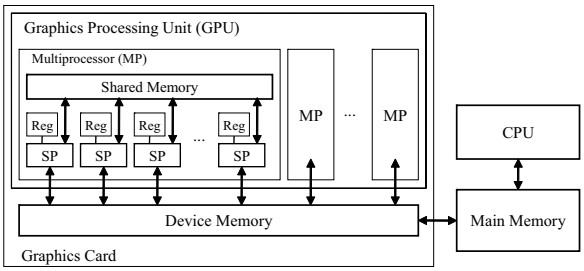

Figure 1. CUDA hardware model. SP and reg denote stream processor and register, respectively.

In summary, the SSSP-based algorithm shows the fastest results while many researchers implement the FW algorithm on various accelerators. With respect to the FW implementation, the timing results mentioned above are equivalent to 9.6, 4.6, and 1.0 GFLOPS (76.8, 36.8, and 8.0 GB/s) on the FPGA [2], the CPU [3], and the GPU [4], respectively. On the other hand, the GPU employed in [4] provides a peak performance of 345.6 GFLOPS and a peak bandwidth of 86.4 GB/s. Thus, we think that the performance on the GPU can be further improved though the SSSP-based algorithm demonstrates the fastest result.

3 Compute Unified Device Architecture

Compute unified device architecture (CUDA) [7] 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 as a thread block, which is independently assigned to an MP. Therefore, developers have to write their kernel such

that there is no data dependence between different thread blocks. 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.

Each MP processes a thread block in the following way. Given a block, it 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 [7] 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 to every MP. However, memory resources such as shared memory and registers usually limit the number of thread blocks per MP.

Memory coalescing [7] is also important to achieve a 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 alignment requirements: a thread with ID N within the half-warp should access address $base + N$, where $base$ is a multiple of 16 bytes [7].

4 SSSP-based Iterative Method

The SSSP-based iterative method [4] computes 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 [4] 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

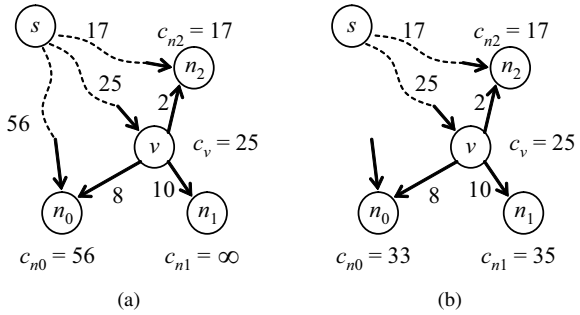

Figure 2. Cost minimization. (a) For each vertex v in the graph, (b) the costs of its neighbors n_0, n_1, and n_2 are updated in the scattering phase.

to minimize the costs of their neighbors 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, 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 neighbor(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 neighbor $n \in V$ 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 recent GPUs with compute capability 1.1 and higher [7]. 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 et al. 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 respon-

```
SSSP_ALGORITHM(s, V, E, W)      /* s: source vertex */
 1: initialize c_v := ∞ and u_v := ∞ for all v ∈ 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 ∈ V in parallel do
 6:       if v ∈ M then      /* Scattering phase */
 7:          remove v from M
 8:          for each neighboring vertex n ∈ V such that (v, n) ∈ 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. Algorithm for finding an SSSP from the source vertex $s \in V$.

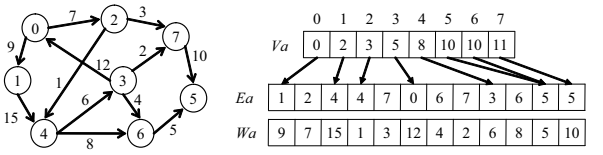

Figure 4. Adjacency list representation. Array Va stores the indices to the head of each adjacency list in Ea. Array Ea and Wa store adjacency lists of every vertex and edge weight, respectively.

```
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 [4]. This kernel is responsible for a single vertex v and updates the costs of its adjacent vertices.

sible 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 all outgoing edges $n \in V$ such that $(v, n) \in E$. Harish et al. convert these lists into arrays Va, Ea, and Wa, which store vertex set V, edge set E, 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 of edge $Ea[i]$ is stored in $Wa[i]$, where $0 \leq i \leq |E| - 1$. 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 Proposed Method

We now describe the proposed algorithm for accelerating the cost computation of APSPs on a directed, positively weighted graph G. 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

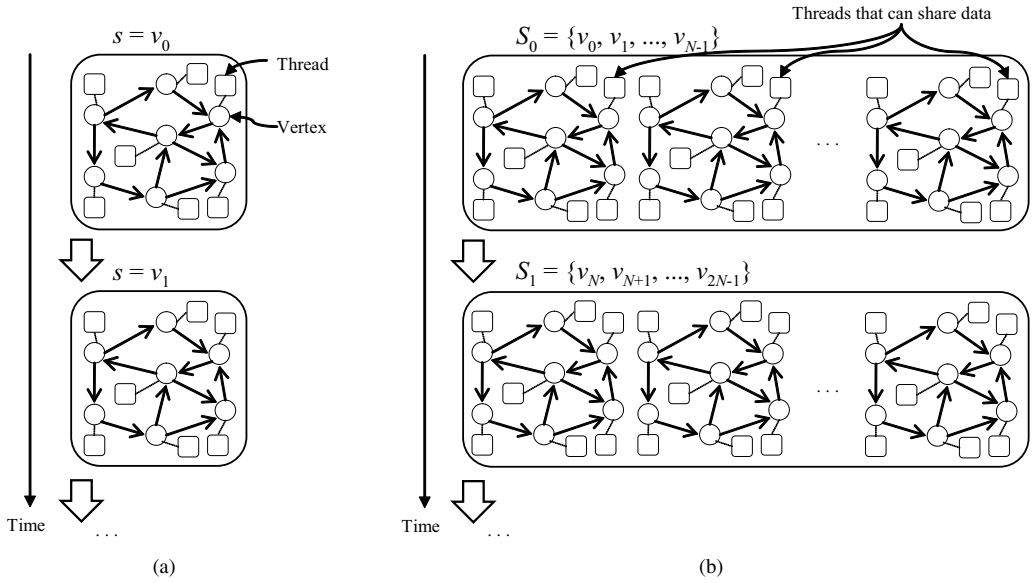

Figure 6. Comparison of parallelization scheme between (a) prior method [4] and (b) proposed method. Our kernel solves N SSSP problems at a time. The graph data is shared between threads that are responsible for the same vertex but in different SSSP problems.

GPU, which employs a massively multithreaded architecture. Since the algorithm we use for a single SSSP problem is the same one developed by Harish et al. [4], we explain here how tasks are grouped to share the graph data.

Let p_s denote an 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 $1 \leq N \leq |V|$. Thus, the k-th group contains N SSSP problems $p_{kN}, p_{kN+1}, \ldots, p_{(k+1)N-1}$, where $0 \leq k \leq \lceil |V|/N \rceil - 1$. Let S_k denote the set of source vertices in the k-th group of N SSSP problems, where $0 \leq k \leq \lceil |V|/N \rceil - 1$. 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 et al. do in their algorithm. Accordingly, our kernel processes $N|V|$ threads in parallel while the prior 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 prior algorithm with respect to the use of shared memory in 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 neighboring 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 neighbors. However, it requires additional copy operations to duplicate data to shared memory. Therefore, threads might degrade the performance if such a common access rarely occurs during execution.

With respect to the graph representation, our kernel uses a slightly different data structure from the prior 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. 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 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

```
N_SSSPS_ALGORITHM(S_k, V, E, W)      /* S_k: set of source vertices */
 1: initialize c_{v,s} := ∞ and u_{v,s} := ∞ for all v ∈ V and s ∈ S_k      /* u_{v,s}: updated cost of vertex v in problem p_s */
 2: initialize c_{s,s} := 0 for all s ∈ S_k      /* c_{v,s}: current cost of vertex v in problem p_s */
 3: add pair ⟨s, s⟩ to set M for all s ∈ S_k
 4: while M is not empty do
 5:   for each vertex v ∈ V and each source s ∈ S_k in parallel do      /* Scattering phase */
 6:     if ⟨v, s⟩ ∈ M then
 7:       remove ⟨v, s⟩ from M
 8:       for each neighboring vertex n ∈ V such that (v, n) ∈ 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 ⟨v, s⟩ 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.

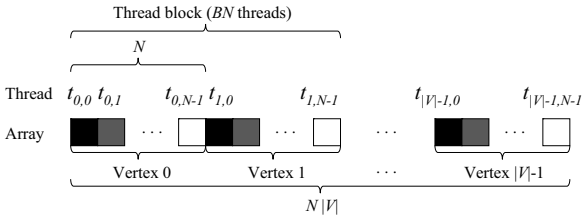

Figure 8. Array interleaving for coalesced memory accesses. The same vertices but for different problems are stored in a contiguous address space. $B = 2$, in this case.

a critical problem because each element in these arrays is accessed only by its responsible thread. Instead, as shown in Fig. 8, 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 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.

6 Experimental Results

To evaluate the performance of our method, we compare it with prior methods: Harish's SSSP-based method [4] running on the GPU and Dijkstra-based method running on the CPU. The latter is accelerated using a binary heap and is known as a fast algorithm for finding an SSSP for sparse graphs. To the best of our knowledge, there is no efficient parallelization for Dijkstra's algorithm. Since the CPU implementation is not multithreaded, it runs on a single core.

We perform all experiments on a PC with an Intel Xeon 5440 2.83 GHz CPU, 12 MB L2 cache, and 8 GB RAM. The GPU employed for experiments is an nVIDIA GeForce 8800 GTS (G92 architecture) with 512 MB of device memory and 16 MPs, each having 8 SPs. All implementations run on Windows XP with CUDA 1.1 and driver version 169.21. Note here that G92 architecture supports atomic instructions to correctly process the scattering kernel.

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 [1]. Using this tool, we generate graphs such that every weight has an integer value within the range $[1, w_{max}]$, where $w_{max} = |V|$.

Table 1 shows timing results obtained with varying the number $|V|$ of vertices. During measurements, the number

Table 1. Timing results for random graphs with a different number $|V|$ of vertices. Results are presented in milliseconds.

Method	Platform	$\|V\|$: Number of vertices					
		1K	2K	4K	8K	16K	32K
Harish SSSP-based [4]	GPU	934	2,130	5,244	13,046	36,670	106,743
Proposed w/o shared memory		71	212	748	2,764	11,128	44,488
Proposed		62	175	586	2,074	8,257	31,052
Dijkstra SSSP-based	CPU	132	590	2,644	11,626	50,478	219,908

```
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__ es[B, N], ws[B, N], 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]
13:    /* Copy data to shared memory */
14:    from[v] := Va[vg]
15:    to[v] := Va[vg + 1]
16:    neighbors := to[v] − from[v]
17:    if s < neighbors 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 neighbors − 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.

$|E|$ 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.

As compared with the prior SSSP-based method, the proposed method achieves the best speedup of 15X when $|V| = 1$K. In particular, our method runs more efficiently than the prior 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 $|V| = 1$K, which is equivalent to 2K threads per MP. In contrast, the prior 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 prior method when $|V| \leq 8$K.

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 prior 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, we can see that the speedup achieved by shared memory ranges from a factor of 1.1 to that of 1.4. This speedup is achieved by shared memory, which eliminates approximately 20% of the data access between SPs and device memory. On the other hand, the unshared version of the proposed method is at least 2.4 times faster than the prior 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 prior 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 10 shows timings results for graphs with a different number $|E|$ of edges. Every graph has the same number of vertices: $|V| = 4$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

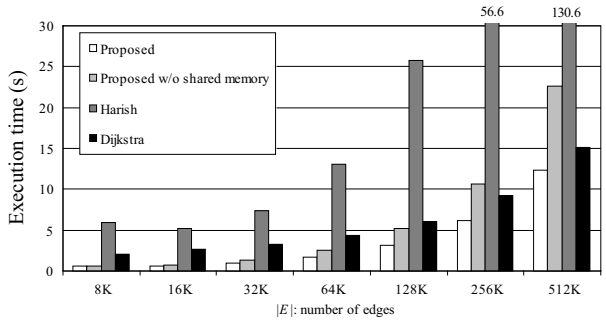

Figure 10. Timing results with different number $|E|$ of edges. The number $|V|$ of vertices is fixed to $|V| = 4K$.

of shared memory.

Another interesting point in Fig. 10 is that the prior method decreases execution time when $|E|$ increases from 8K to 16K. This is due to the reduction of kernel launches. It requires 27 launches per SSSP problem when dealing with $|E| = 8$K edges, but this is reduced to 19 launches when $|E| = 16$K edges. Similar reductions are also observed when $|E| \geq 16$K. However, they do not significantly affect the execution time, because each launch takes longer execution time as $|E|$ increases. Actually, the scattering kernel has a loop repeated for every outgoing edge and has to update more vertices as $|E|$ increases.

7 Conclusion

We have presented a fast algorithm for solving the APSP problem on the CUDA-enabled GPU. Our algorithm is based on an SSSP-based algorithm [4] but has a different parallelization scheme to exploit the task parallelism in the APSP problem. The proposed scheme solves multiple SSSP problems in parallel, leading to an efficient use of on-chip shared memory. Thus, our task parallel scheme reduces the amount of data being transferred from off-chip memory. Furthermore, the scheme can run many threads with less kernel launches, which is suitable to highly multithreaded architecture of the GPU.

The experimental results show that our method is 3.4–15 times faster than the prior method running on the same GPU. As compared with the prior method, the task parallel scheme shows 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.

One future work is to store the APSPs themselves because our method currently computes the costs instead of the paths.

Acknowledgments

This work was partly supported by JSPS Grant-in-Aid for Scientific Research (A)(2)(20240002), Young Researchers (B)(19700061), and the Global COE Program "in silico medicine" at Osaka University. We would like to thank the anonymous reviewers for their valuable comments.

References

[1] 9th DIMACS implementation challenge - Shortest paths. `http://www.dis.uniroma1.it/~challenge9/download.shtml`.

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

[3] S.-C. Han, F. Franchetti, and M. Püshel. Program generation for the all-pairs shortest path problem. In *Proc. 15th Int'l Conf. Parallel Architectures and Compilation Techniques (PACT'07)*, pages 222–208, Sept. 2006.

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

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

[6] A. Nakaya, S. Goto, and M. Kanehisa. Extraction of correlated gene clusters by multiple graph comparison. *Genome Informatics*, 12:34–43, Dec. 2001.

[7] nVIDIA Corporation. CUDA Programming Guide Version 1.1, Nov. 2007.

[8] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, Mar. 2007.

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