PAPER Special Section on Parallel and Distributed Computing and Networking

Blocked United Algorithm for the All-Pairs Shortest Paths Problem on Hybrid CPU-GPU Systems

Kazuya MATSUMOTO[†a)], **Naohito NAKASATO**[†], ***Nonmembers***, ***and*** **Stanislav G. SEDUKHIN**[†], ***Member***

SUMMARY This paper presents a blocked united algorithm for the all-pairs shortest paths (APSP) problem. This algorithm simultaneously computes both the shortest-path distance matrix and the shortest-path construction matrix for a graph. It is designed for a high-speed APSP solution on hybrid CPU-GPU systems. In our implementation, two most compute intensive parts of the algorithm are performed on the GPU. The first part is to solve the APSP sub-problem for a block of sub-matrices, and the other part is a matrix-matrix "multiplication" for the APSP problem. Moreover, the amount of data communication between CPU (host) memory and GPU memory is reduced by reusing blocks once sent to the GPU. When a problem size (the number of vertices in a graph) is large enough compared to a block size, our implementation of the blocked algorithm requires CPU $\rightleftharpoons$ GPU exchanging of three blocks during a block computation on the GPU. We measured the performance of the algorithm implementation on two different CPU-GPU systems. A system containing an Intel Sandy Bridge CPU (Core i7 2600K) and an AMD Cayman GPU (Radeon HD 6970) achieves the performance up to 1.1 TFlop/s in a single precision.

key words: *all-pairs shortest paths problem, path construction, Floyd-Warshall algorithm, blocked algorithm, hybrid CPU-GPU systems*

1. Introduction

The *all-pairs shortest paths* (APSP) problem is to find the shortest paths between all-pairs of vertices in a weighted graph [1], [2]. The problem is one of the most fundamental graph problems, and there are applications of the APSP problem in bioinformatics, social networking, traffic routing, etc. The APSP problem is an instance of a general framework in the so called *algebraic path problem* [3], [4] which covers several graph, matrix, and language processing problems. It means that acceleration of the APSP problem leads to acceleration of all these problems.

A well-known solution of the APSP problem is to apply the classical *Floyd-Warshall* (FW) algorithm [5], [6]. The FW algorithm requires $O(n^3)$ operations on $O(n^2)$ memory space, where n is the number of vertices in a graph. Blocked algorithms of the FW algorithm were proposed to efficiently utilize hierarchical memory structures of current processors [7]–[9]. For a high-speed solution of the APSP problem, different variants of the blocked algorithms have been implemented on CPUs [10]–[12], GPUs [13]–[17], and an FPGA [18].

The previous studies, however, presented implementations of blocked APSP algorithms only for computing the shortest-path distance matrix. In the APSP problem, construction of a path with the shortest distance is required in many cases. Buluç et al. [14] mentioned a possibility to construct the shortest paths by using a matrix which keeps an intermediate vertex between every pair of vertices. To our best knowledge, there have been no report on implementations of a blocked APSP algorithm for computing both the shortest-path distance matrix and the shortest-path construction matrix. We consider high-speed solutions on GPUs as the most prominent because the massively parallel architecture and the high memory bandwidth of GPUs meet requirements for a fast implementation of blocked APSP algorithms.

This paper introduces a blocked united APSP algorithm for computing both matrices at the same time. The presented united algorithm is an extension of the blocked algorithm for a hybrid CPU-GPU system designed in our previous study [17]. The proposed united algorithm can solve the APSP problem of a graph whose required memory size is larger than the capacity of GPU memory. In the algorithm, there are two most compute intensive parts (kernels), both of which run on the GPU. The first kernel is to solve the APSP sub-problem for a block of sub-matrices. The other kernel is a matrix-matrix "multiplication" for the APSP problem. For high utilization of the GPU kernels, we optimized data communication between CPU (host) and GPU. We have evaluated the performance of our APSP implementation on two different systems containing an Intel CPU and an AMD GPU. We show comparison results among our APSP implementations on the CPU-GPU systems and implementations only on a CPU or a GPU.

The rest of this paper is organized as follows. Section 2 describes the APSP problem and the Floyd-Warshall algorithm. Section 3 presents our blocked united algorithm for the APSP problem. Section 4 shows how we implemented the blocked algorithm on the hybrid CPU-GPU systems and benchmark results of the implementation. Section 5 concludes this paper with a mention of future work.

2. All-Pairs Shortest Paths Problem

Let $G = (V, E)$ be a weighted directed graph with a edge-weight function $w : E \to \mathbb{R}$ that assigns a real-valued weight to each edge, where $V = \{1, 2, \ldots, n\}$ is a set of n vertices, $E \subseteq V \times V$ is a set of edges, and $\mathbb{R}$ is a set of real numbers. We use an adjacency matrix representation for the graph G. In an adjacency matrix $A = [a_{i,j}]$, $a_{i,j}$ represents the corresponding value of an edge (i, j).

Manuscript received January 10, 2012.
Manuscript revised May 27, 2012.

[†]The authors are with Distributed Parallel Processing Laboratory, the University of Aizu, Aizu-Wakamatsu-shi, 965–8580 Japan.
a) E-mail: d8121101@u-aizu.ac.jp

DOI: 10.1587/transinf.E95.D.2759

Copyright © 2012 The Institute of Electronics, Information and Communication Engineers

The input of the APSP problem is an $n \times n$ matrix $W = [w_{i,j}]$ which contains edge weights of a given graph G, where

$$w_{i,j} = \begin{cases} \text{weight of edge } (i,j) & \text{if } (i,j) \in E; \\ \infty & \text{if } (i,j) \notin E. \end{cases}$$

Given a weight matrix W, we can compute a weight (or distance) of the shortest paths between all-pairs of vertices. The computed result is an $n \times n$ matrix $D = [d_{i,j}] = [d_{i,j}^{(n)}]$, where $d_{i,j}$ denotes the shortest-path distance from i to j. We can define each $d_{i,j}^{(k)}$ by the following recurrence equation [1]:

$$d_{i,j}^{(k)} = \begin{cases} w_{i,j} & \text{if } k = 0; \\ \min\left(d_{i,j}^{(k-1)}, d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}\right) & \text{if } k \geq 1. \end{cases} \quad (1)$$

Note that algorithms discussed in this paper allow existence of negative-weight edges in a graph G but do not allow negative-weight cycles.

To find the all-pairs shortest paths, it is required to compute not only the shortest-path distance matrix D but additionally a shortest-path *construction matrix* $C = [c_{i,j}]$. If at least one shortest path exists between a vertex i and a vertex j, then $c_{i,j}$ indicates the highest-numbered intermediate vertex on the shortest path, and otherwise, it is undefined (`NULL`). Initial values of a construction matrix are all undefined, i.e., all $c_{i,j}^{(0)} =$ `NULL`. The shortest-path construction matrix $C = [c_{i,j}] = [c_{i,j}^{(n)}]$ can be simultaneously computed with computation of the shortest-path distance matrix $D = [d_{i,j}] = [d_{i,j}^{(n)}]$, and each $c_{i,j}^{(k)}$ is defined as the following equation [2]:

$$c_{i,j}^{(k)} = \begin{cases} \texttt{NULL} & \text{if } k = 0; \\ k & \text{if } k \geq 1 \text{ and } d_{i,j}^{(k-1)} > d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}; \\ c_{i,j}^{(k-1)} & \text{otherwise.} \end{cases} \quad (2)$$

The Eqs. (1) and (2) lead to a dynamic-programming solution known as *Floyd-Warshall* algorithm, which is shown as Algorithm 1. In the following algorithm descriptions, we eliminate superscripts used in the Eqs. (1) and (2). The FW algorithm contains a three-nested loop which is similar to the standard matrix-matrix multiply-add algorithm, except the FW algorithm has stricter data dependencies such that the outermost k-loop cannot be interchanged with the other two inner loops. This FW Algorithm 1 requires $4n^3$ operations (addition, comparison, and two conditional assignments) on $2n^2$ data of the two matrices, and our proposed blocked Algorithm 3 described later asymptotically requires the same number of operations.

After computing the shortest-path construction matrix C by solving the APSP problem, the shortest paths between all-pairs can be built. If $d_{i,j} \neq \infty$ and $c_{i,j} \neq$ `NULL` then there exists an intermediate vertex $c_{i,j}$ such that the shortest path from i to j is a shortest path from i to $c_{i,j}$ followed by a shortest path from $c_{i,j}$ to j. The two sub-shortest paths can be determined recursively. Algorithm 2 shows a recursive procedure to print out all the intermediate vertices, in order, between a given pair of vertices i and j.

Algorithm 1: Floyd-Warshall algorithm for computing both the shortest-path distance matrix D and the shortest-path construction matrix C

```
1  FloydWarshall(n, D, C)
2  begin
3      for k ← 1 to n do
4          for all 1 ≤ i ≤ n do
5              for all 1 ≤ j ≤ n do
6                  sum ← d_{i,k} + d_{k,j};
7                  if d_{i,j} > sum then
8                      d_{i,j} ← sum;
9                      c_{i,j} ← k;
10     return {D, C};
```

Algorithm 2: Recursive procedure for printing out intermediate vertices of the shortest path between a pair of vertices (i, j)

```
1  PrintIntermediateVertices(C, i, j)
2  begin
3      if C_{i,j} ≠ NULL then
4          PrintIntermediateVertices(C, i, c_{i,j});
5          print C_{i,j};
6          PrintIntermediateVertices(C, c_{i,j}, j);
```

To construct the shortest paths, there are other methods such as computing the *predecessor matrix* described in [1]. However, we consider that computing the construction matrix is more suitable for a high-speed solution on CPU-GPU systems because the computation for construction matrix requires less memory references.

3. Blocked United Algorithm

This section presents a blocked united algorithm for the APSP problem. This algorithm is based on the blocked algorithm discussed in our previous paper [17], to which we added the computation of shortest-path construction matrix. The blocked algorithm is designed for a fast APSP solution with high GPU utilization on hybrid CPU-GPU systems.

In the blocked united algorithm, we partition both the $n \times n$ distance matrix and construction matrix into blocks of $b \times b$ sub-matrices, where b is a blocking factor. For simplicity, we suppose that n is in multiples of b in the following algorithm description. Let us identify a sub-matrix of block index (I, J) by $A_{I,J} = [a_{i,j}]$, where $1 \leq I, J \leq n/b$ and $1 \leq i, j \leq b$. Figure 1 shows an example of 12×12 matrix with the blocking factor $b = 3$.

Algorithm 3 shows our blocked united APSP algorithm. In this algorithm, each iteration of the outermost loop essentially performs the same number of operations as b iterations of the Floyd-Warshall Algorithm 1; however, the two algorithms may find different shortest paths with an equal minimum distance because the two algorithms have differ-

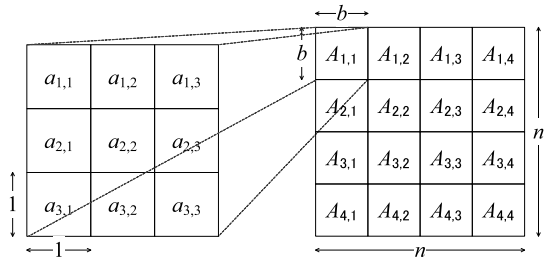

Fig. 1 An $n \times n$ matrix partitioned with a blocking factor b.

Algorithm 3: Blocked united algorithm for the APSP problem

```
1  BlockedAPSP(n, b, D, C)
2  begin
3      N ← n/b;
4      for K ← 1 to N do
           // Phase 1: updating the pivot blocks
5          {D_{K,K}, C_{K,K}} ← SubBlockedAPSP(K, b, D_{K,K}, C_{K,K});
           // Phase 2: updating the pivot column blocks
6          for all 1 ≤ I ≤ N (I ≠ K) do
7              {D_{I,K}, C_{I,K}} ← MMA(K,b,D_{I,K},D_{K,K},D_{I,K},C_{I,K});
           // Phase 3: updating the pivot row blocks
8          for all 1 ≤ J ≤ N (J ≠ K) do
9              {D_{K,J}, C_{K,J}} ← MMA(K,b,D_{K,K},D_{K,J},D_{K,J},C_{K,J});
           /* Phase 4: updating the remaining
              (non-pivot) blocks                          */
10         for all 1 ≤ I, J ≤ N (I ≠ K && J ≠ K) do
11             {D_{I,J}, C_{I,J}} ← MMA(K,b,D_{I,K},D_{K,J},D_{I,J},C_{I,J});
12     return {D, C};
```

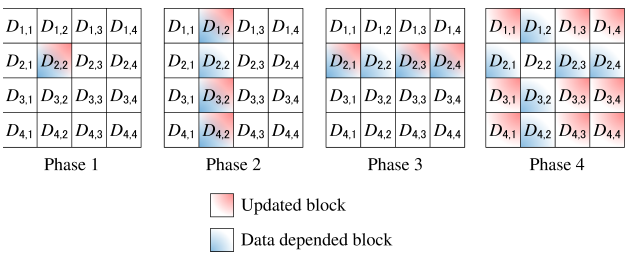

Fig. 2 Visualization of updated and data depended blocks in each phase of the blocked united Algorithm 3 for $n/b = 4$ and $K = 2$.

ent order for computing path construction matrices.

The computation on each K-th iteration in the blocked Algorithm 3 is divided into four phases: Phase 1 updates the pivot $b \times b$ blocks $\{D_{K,K}, C_{K,K}\}$. Phase 2 updates the pivot-column blocks $\{D_{I,K}, C_{I,K}\}$ $(I \neq K)$. Phase 3 updates the pivot-row blocks $\{D_{K,J}, C_{K,J}\}$ $(J \neq K)$. Finally, Phase 4 updates the other (non-pivot) blocks $\{D_{I,J}, C_{I,J}\}$ $(I \neq K$ and $J \neq K)$. Figure 2 visualizes updated and data depended blocks of distance matrix in each phase of the blocked Algorithm 3.

Phase 1 is to solve the APSP sub-problem for $b \times b$ sub-matrices $D_{K,K}, C_{K,K}$. For solving the APSP sub-problem, we can apply either the Floyd-Warshall Algorithm 1 or the blocked united algorithm itself recursively with a small modification. We use the latter approach and Algorithm 4 shows the blocked algorithm modified for the sub-problem. Like Algorithm 3, Algorithm 4 also has four phases but is followed by a process for merging the computed sub-matrices. The scalar Floyd-Warshall Algorithm 1 is used in Phase 1 of Algorithm 4.

Algorithm 4: Blocked united algorithm for the APSP sub-problem

```
1  SubBlockedAPSP(K, b, D, C)
2  begin
3      Let b_s be another blocking factor (b_s ≤ b) and D' and C' be
       b × b matrices;
4      N_s ← b/b_s;
5      D' ← D;
6      for Ks ← 1 to Ns do
           // Phase 1
7          {D'_{K_s,K_s}, C'_{K_s,K_s}} ← FloydWarshall(b_s, D'_{K_s,K_s}, C'_{K_s,K_s});
           // Phase 2
8          for all 1 ≤ I_s ≤ N_s (I_s ≠ K_s) do
9              {D'_{I_s,K_s}, C'_{I_s,K_s}} ←
10                 SubMMA(b_s, D'_{I_s,K_s}, D'_{K_s,K_s}, D'_{I_s,K_s}, C'_{I_s,K_s});
           // Phase 3
11         for all 1 ≤ J_s ≤ N_s (J_s ≠ K_s) do
12             {D'_{K_s,J_s}, C'_{K_s,J_s}} ←
13                 SubMMA(b_s, D'_{K_s,K_s}, D'_{K_s,J_s}, D'_{K_s,J_s}, C'_{K_s,J_s});
           // Phase 4
14         for all 1 ≤ I_s, J_s ≤ Ns (I_s ≠ K_s && J_s ≠ K_s) do
15             {D'_{I_s,J_s}, C'_{I_s,J_s}} ←
16                 SubMMA(b_s, D'_{I_s,K_s}, D'_{K_s,J_s}, D'_{I_s,J_s}, C'_{I_s,J_s});
       /* Merge the computed matrices with the
          pre-computed matrices                           */
17     for all 1 ≤ i, j ≤ b do
18         if d_{i,j} > d'_{i,j} then
19             d_{i,j} ← d'_{i,j};
20             c_{i,j} ← c'_{i,j} + K · b;
21     return {D, C};
```

In Phases 2–4 of both blocked Algorithms 3 and 4, every block can be updated by using a matrix-matrix "multiply-add" (MMA) operation designed for the APSP problem. If the computation of path construction matrix is not taken into consideration, the MMA operation is called matrix-matrix multiply-add in *min-plus algebra* [19], [20] (or *tropical semiring* [21]). The difference between matrix multiply-add in linear algebra and in min-plus algebra is in the type of operations to obtain corresponding result. In linear algebra, we need arithmetic multiplication and addition and an element $z_{i,j}$ of sub-matrix Z is updated with the formula of $z_{i,j} \leftarrow z_{i,j} + \sum_{k=1}^{b} x_{i,k} \cdot y_{k,j}$. In the min-plus algebra, the addition is replaced by minimum operation and the multiplication is replaced by arithmetic addition; thus, an element $z_{i,j}$ is updated with the formula of $z_{i,j} \leftarrow \min(z_{i,j}, \min_{k=1}^{b}(x_{i,k} + y_{k,j}))$.

The MMA in min-plus algebra can be extended to simultaneously compute the path construction matrix and the

Algorithm 5: Matrix-matrix "multiply-add" algorithm for the sub-blocked united APSP Algorithm 4

```
1 SubMMA(b_s, X, Y, Z, C)
2 begin
3     for all 1 ≤ i, j ≤ b_s do
4         for k ← 1 to b_s do
5             sum ← x_{i,k} + y_{k,j};
6             if z_{i,j} > sum then
7                 z_{i,j} ← sum;
8                 c_{i,j} ← k;
9     return {Z, C};
```

Algorithm 6: Matrix-matrix "multiply-add" algorithm for the blocked united APSP Algorithm 3

```
1  MMA(K, b, X, Y, Z, C)
2  begin
3      {Z', C'} ← MINPLUS(b, X, Y);
4      for all 1 ≤ i, j ≤ b do
5          if z_{i,j} > z'_{i,j} then
6              z_{i,j} ← z'_{i,j};
7              c_{i,j} ← c'_{i,j} + K · b;
8      return {Z, C};
9  MINPLUS(b, X, Y)
10 begin
11     Let Z' and C' be b × b matrices;
12     for all 1 ≤ i, j ≤ b do
13         z'_{i,j} ← ∞;
14         for k ← 1 to b do
15             sum = x_{i,k} + y_{k,j};
16             if z'_{i,j} > sum then
17                 z'_{i,j} ← sum;
18                 c'_{i,j} ← k;
19     return {Z', C'};
```

distance matrix, and this extended MMA is described in Algorithm 5, which is invoked from the blocked Algorithm 4 for APSP sub-problem. The MMA Algorithm 5 requires four input matrices X, Y, Z, C to produce two output matrices Z, C. In the blocked Algorithm 3, we use another method for the MMA shown in Algorithm 6. In the MMA Algorithm 6, the matrix-matrix "multiplication" in the extended min-plus algebra (MINPLUS) and the matrix-matrix "addition" are separately executed. The MINPLUS operation is performed on a GPU while the matrix-matrix "addition" (minimum) operation is carried out on a CPU. This separation reduces a data sending of two input matrices Z, C to the GPU. As a result, it makes possible to alleviate a bandwidth requirement for data communication between the CPU (host) memory and the GPU memory.

Table 1 System configurations; BW (bandwidth), CU (Compute Unit), LDS (Local Data Share), SIMD (Single-Instruction Multiple-Data), SP (single-precision), PE (processing element).

		System A	System B
GPU	Code name	Cayman	Cypress
	Board name	Radeon HD 6970	Radeon HD 5870
	Core clock [GHz]	0.88	0.85
	Number of PEs	1536	1600
	Number of CUs	24	20
	SP peak perf. [GFlop/s]	2703	2720
	Memory clock [GHz]	1.375	1.2
	Memory BW [GB/s]	176	154
	L2 read BW [GB/s]	451	435
	L1 read BW [GB/s]	1352	1088
	LDS read BW [GB/s]	2703	2176
	Memory size [GB]	2	1
	L2 caceh size [kB]	512	512
	L1 cach size / CU [kB]	8	8
	LDS size / CU [kB]	32	32
CPU	Code name	Sandy Bridge	Bloomfield
	Processor name	Core i7 2600K	Core i7 970
	Core clock [GHz]	3.4	3.2
	Num. of cores	4	6
	SP SIMD width	8	4
	SP peak perf [GFlop/s]	217.6	153.6

4. Implementation and Experimentation

4.1 Experimental Environment

We port our program of the blocked APSP Algorithm 3 to two kinds of hybrid systems containing an Intel CPU and an AMD GPU. Table 1 shows the configurations of the two used systems. The prominent difference between the two GPU architectures is that the Cayman GPU uses a four-wide symmetric VLIW (Very Long Instruction Word) instead of a five-wide asymmetric VLIW on the Cypress GPU [22], in addition to a number of other enhancements for both graphics and more general workloads. The single-precision peak performance of the Cayman is a little lower than that of the Cypress; nevertheless, the Cayman is considered to deliver higher performance in most of applications because of the higher memory bandwidth. Both GPUs also support double-precision floating-point instructions with the peak performance of 676 GFlop/s in the Cayman and 544 GFlop/s in the Cypress though, in this study, we only deal with a single-precision. The CPU (host) and the GPU are connected through 16 lanes of PCI-Express 2.0 buses whose aggregated peak bandwidth is 8 GB/s.

Both systems run on Ubuntu 10.04, and the Linux kernel version is 2.6.32-37 in System A and 2.6.32-35 in System B (see Table 1). The installed display driver is AMD Catalyst 11.11. We use `gcc 4.6.2` compiler with `-O2` optimization option for program compilation and AMD Accelerated Parallel Processing (APP) SDK v2.5 for GPU programming. Input graph data were generated by using R-mat random graph generator in GTgraph [23] with the average vertex degree of 20. Note that the performance of our im-

plementations is not sensitive to graph sparsity. All performance values presented in this paper are average of ten times measurements.

4.2 GPU Kernels

The blocked APSP Algorithm 3 requires two different kernels which run on the GPU. The first one is a kernel for the sub-blocked APSP Algorithm 4 (SubBlockedAPSP). The second one is a kernel for the matrix-matrix "multiplication" (MINPLUS in Algorithm 6).

Note that the performance in Flop/s is calculated by the formula of

$$4n^3 \text{ [Flops]} / \text{ running time [sec]}, \quad (3)$$

where n is the problem size and the running time does not include the time for GPU initialization. In addition, the running time of the GPU kernels does not include the communication time for matrix data between the host and the GPU. The GPU kernels for the APSP problem cannot be implemented with high-performance FMA (fused multiply-add) instructions which perform two floating-point operations (multiplication and addition) per clock cycle. In our case of the APSP problem, four instructions are needed for four operations. Thus, although the peak FMA-performance of both GPUs is around 2700 GFlop/s, we can expect at most half of the peak performance for the APSP problem, i.e., 1352 GFlop/s on the Cayman and 1360 GFlop/s on the Cypress.

4.2.1 Kernel for Sub-Blocked APSP Algorithm

The kernel for the sub-blocked APSP Algorithm 4 is to compute the most intensive part (lines 4-16) in Algorithm 4 on the GPU. We have developed the kernel in OpenCL (Open Computing Language) 1.1 [22], [24]. Note that the merge operation (lines 17-20) is performed on the CPU. Separating the merging computation from the whole computation of the algorithm has a positive effect for increasing performance. The separation reduces data communication time because two $b \times b$ matrices D and C are not needed to be sent to the GPU.

Algorithm 4 introduces another blocking factor b_s ($\leq b$). The size of this blocking factor highly influences the performance of the kernel. Phase 1 of the algorithm is to apply the Floyd-Warshall Algorithm 1 for a $b_s \times b_s$ sub-matrix. To implement the FW algorithm on the GPU, a synchronization among GPU threads is required in the beginning of every outermost iteration (immediately after line 3 in Algorithm 1). It means that the upper blocking factor b_s is limited by the size of shared memory called *Local Data Share* (LDS). We have selected 64 as the size of blocking factor b_s because the required size ($32\,\text{kB} = 2 \cdot 64^2 \cdot 4\,\text{Bytes}$) for the two blocks $D'_{K_s,K_s}, C'_{K_s,K_s}$ is equal to the LDS size and the blocking factor should be a power-of-two for easy usage of the kernel invoked in the implementation of the main

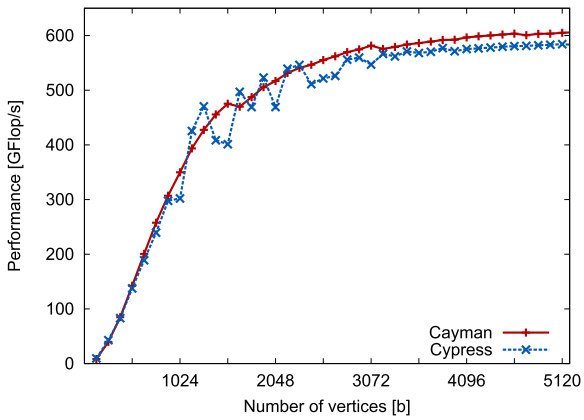

Fig. 3 Performance of the kernel for solving the APSP sub-problem (lines 4-16 in Algorithm 4) on GPUs.

Algorithm 3. Although we can select even smaller blocking factor b_s, this selection results in performance deterioration. For example, the sub-blocked APSP kernel in the case $b_s = 32$ delivers 2.56 times lower performance, on average, than the case $b_s = 64$ on the Cayman.

Figure 3 shows the performance of the implemented kernel on the Cayman and the Cypress as a function of the sub-problem size (the number of vertices) b. The kernel achieves up to 610 GFlop/s (45% of 1353 GFlop/s) on the Cayman and 585 GFlop/s (43% of 1360 GFlop/s) on the Cypress. A performance fluctuation is seen on the Cypress depending on b. The standard deviation of performance is around 65 in the case $b = 1536$ on the Cypress while it is around 4 on the Cayman. When $b \geq 2816$, the standard deviation is mostly less than 6 on both GPUs.

4.2.2 Kernel for Matrix-Matrix "Multiplication"

In the blocked APSP Algorithm 3, the matrix-matrix "multiplication" (MINPLUS in Algorithm 6) is the most frequent and compute intensive part, which strongly affects the total performance; thus, a fast computation of MINPLUS kernel on the GPU is essential for a high performance implementation of the blocked algorithm. For the MINPLUS kernel, we have modified an SGEMM (single-precision general matrix multiply) kernel which was developed in our previous studies [25], [26]. The SGEMM kernel was written in an assembly-like language called *Intermediate Language* (IL) [27]. The measured performance of the SGEMM kernel is up to 2432 GFlop/s (90% of 2703 GFlop/s) on the Cayman and 2137 GFlop/s (79% of 2720 GFlop/s) on the Cypress.

In the SGEMM kernel, a basic multiply-add operation of $z_{i,j} \leftarrow x_{i,k} \cdot y_{k,j} + z_{i,j}$ can be implemented with a single FMA instruction on the GPU. However, in the MINPLUS kernel, four different instructions are required to implement a single "multiply-add" operation. This indicates that, for the same matrix size, the MINPLUS kernel takes approximately four times longer running time than the SGEMM kernel. The four required instructions are an addition in-

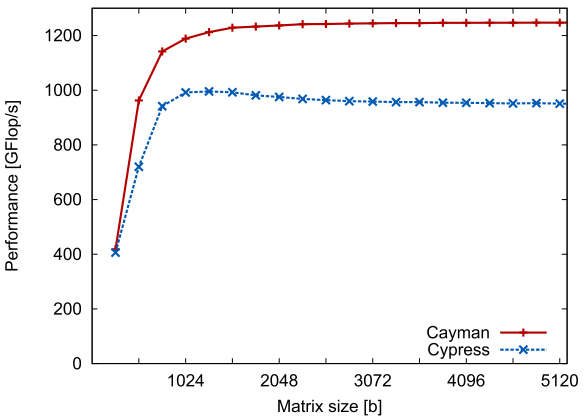

Fig. 4 Performance of the kernel for the matrix-matrix "multiplication" for the APSP problem (MINPLUS in Algorithm 6) on GPUs.

struction (corresponding to line 15 in Algorithm 6), a comparison instruction (line 16), and two conditional move instructions (lines 17, 18).

Figure 4 shows the performance of the MINPLUS kernel on the Cayman and the Cypress as a function of matrix size in multiples of 256. Currently, the kernel does not work if a matrix size is not in multiples of 256 ($= 64 \cdot 4$). It is because a width of global buffer used within a kernel in IL has to be a multiple of 64 elements and each element consists of four single-precision floating point values [28]. The maximum performance of the MINPLUS kernel is 1248 GFlop/s (92% of 1352 GFlop/s) on the Cayman and 995 GFlop/s (73% of 1360 GFlop/s) on the Cypress. A performance variability of the kernel is seen and it is particularly large for small problems sizes ($b \leq 768$).

4.3 Implementation of the Blocked United APSP Algorithm on Hybrid CPU-GPU Systems

We have implemented the blocked APSP Algorithm 3 on the hybrid CPU-GPU systems. Strategies for performance optimization are almost identical to our previous study [17]. In this paper, we show the way how to increase a GPU utilization of the APSP implementation by optimizing data communication between CPU (host) and GPU. Note that we cannot apply the present optimization if $n/b \leq 2$.

Communication latencies of a matrix data transferred between CPU and GPU are the bottleneck for the high GPU utilization. To hide these latencies, we first consider a way to reduce the amount of data communication. The discussed MINPLUS kernel produces two $b \times b$ output matrices Z', C' from two $b \times b$ input matrices X, Y. This means that four $b \times b$ matrices are needed for an execution of the kernel. An input matrix can be reused if a block to be computed is in the same row block or column block as the previously computed block. Our APSP implementation updates blocks ($b \times b$ matrices) on each outermost iteration K ($= 1$ to n/b) of the blocked united Algorithm 3 in the order shown in Fig. 5. Following the order, an input block $D_{K,K}$ is reused in Phases 2 and 3 of the algorithm and either an input block $D_{I,K}$ or

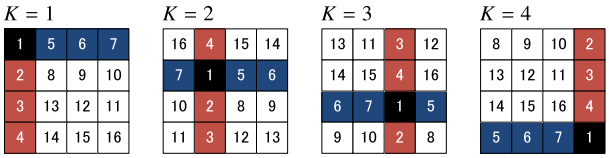

Fig. 5 Order of block computation of our APSP implementation on each outermost iteration (identical to Fig. 8 in [17]).

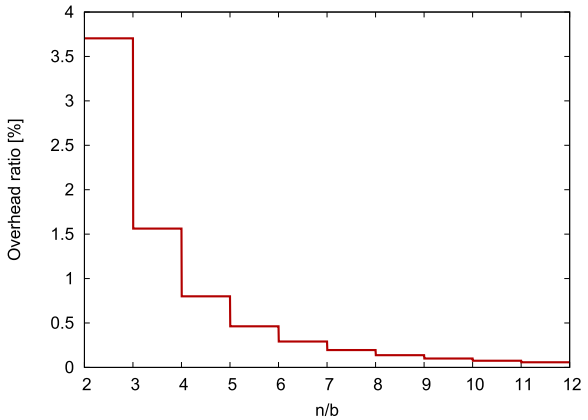

Fig. 6 Overhead ratio b^2/n^2, as a function of n/b, for sending one block ($b \times b$ matrix) on each first block update in Phases 2–4. The problem size n is padded to the size $\lfloor (n + b - 1)/b \rfloor \cdot b$, if n is not in multiples of b.

$D_{K,J}$ is reused in Phase 4, after sending this block to the GPU.

As it can be seen from Algorithm 3 and the order of computing (Fig. 5), the amount of data communication (in bytes) in Phase 1 is $4 \cdot 3b^2$, in both Phase 2 and Phase 3 is $4(b^2 + 3b^2(n/b - 1))$, and in Phase 4 is $4(b^2 + 3b^2(n/b - 1)^2)$, where a single-precision data occupies four bytes. Thus, in total, the amount of data communication of our implementation on each outermost iteration can be expressed as a function of n and b

$$Comm\_Amount(n, b)\ [\text{Bytes}] = 4 \cdot (3n^2 + 3b^2).$$

Since Phase 1, which requires $4 \cdot 3b^2$ data communication, is exceptional (i.e. it needs its own kernel), the amount of an overhead related to one block data sending for each first block update in three Phases 2–4 (blocks associated with order 2, 5, 8 in Fig. 5) is totally $3 \cdot 4b^2$, and the amount of whole data communication in Phases 2–4 is $3 \cdot 4n^2$. Therefore, the ratio of the overhead amount to the whole communication amount equals to b^2/n^2. Figure 6 depicts the overhead ratio as a function of n/b. A staircase pattern is drawn in the figure because our implementation uses a padding technique to solve the APSP problem on a graph whose problem size (number of vertices) n is not in multiples of a blocking factor b. In the padding technique, the values of padded portion in the distance matrix are initialized as infinity ∞ (sufficiently larger value) such that the implementation actually solves the APSP problem on a graph whose size is the next multiple of the blocking factor ($n_{pad} = \lfloor (n + b - 1)/b \rfloor \cdot b$). As it can be seen from Fig. 6, if $n/b > 4$, the overhead ratio is less

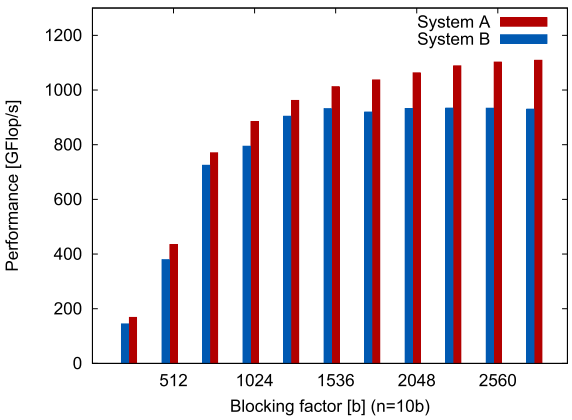

Fig. 7 Performance of the APSP implementation on two different CPU-GPU systems for different blocking factors.

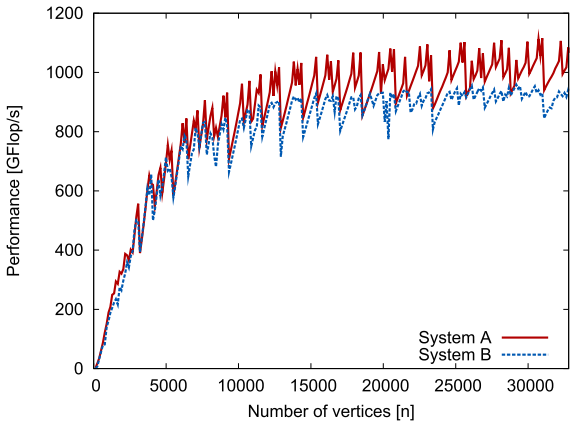

Fig. 8 Performance of the APSP implementation with a near optimal blocking factor as a function of the problem size on the CPU-GPU systems.

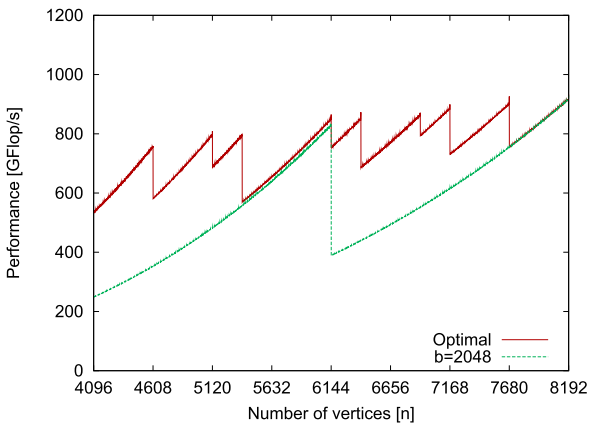

Fig. 9 Performance of the APSP implementation with the near optimal blocking factor as a function of the problem size n and with a predefined blocking factor $b = 2048$ on System A.

than 1%. Moreover, if $n/b > 10$, the ratio is less than 0.1%. Because of it, we can view the implementation of each block update in Phases 2–4 as equal to send one $b \times b$ matrix to the GPU and receives two $b \times b$ matrices from the GPU.

We next consider an overlapping of the data communication and the GPU computation, to hide the communication latencies. The required memory bandwidth for the overlapping is computed by

$$BW\ [\text{Byte/s}] = \frac{\text{Peak performance [Flop/s]}}{Arithmetic\_Intensity\ [\text{Flops/Byte}]}.$$

The value of peak performance is set with the maximum performance of the MINPLUS kernel. The *arithmetic intensity* [29] is the ratio of total floating-point operations to total data communication amount. As our APSP implementation requires three $b \times b$ matrices in single precision for $4b^3$ operations, the arithmetic intensity can be estimated as

$$Arithmetic\_Intensity_{unitedAPSP}(b) = \frac{4b^3}{4 \cdot 3b^2} = \frac{b}{3}. \quad (4)$$

The measured bandwidth between the CPU and the GPU is around 3 GByte/s, and the computed required bandwidth BW for $b = 1280$ is approximately 1248 [GFlop/s]/(1280/3 [Flops/Byte]) $\simeq$ 3 GByte/s. Thus, the communication latencies can be hidden by the overlapping when we use a sufficiently large blocking factor b. Note that the upper size of blocking factor is limited by a capacity of pinned memory (memory mapped to the PCI-Express memory space), and we cannot use an extremely large blocking factor (up to 2816 in the implementation).

Figure 7 shows the performance of the APSP implementation on the CPU-GPU systems for different blocking factors. The performance is saturated on System B which contains a Cypress GPU when the blocking factor $b \geq 1536$. On the other hand, a performance saturation is not seen on System A containing a Cayman GPU. This difference comes from the fact that the overlapping of the GPU computation and the communication cannot be implemented perfectly in the Cayman (an overlapping of the GPU computation and the CPU computation works). The maximum performance is 1116 GFlop/s on System A and 960 GFlop/s on System B, and the efficiency is 89% (1116 of 1248) and 96% (960 of 995), respectively.

As stated above, our implementation uses a padding technique for solving the APSP problem whose size n is not in multiples of a blocking factor b. The disadvantage of the padding technique is that an extra computation is needed for the padded portion. Especially, cases where $n = i \cdot b + 1$ are worst (i is a positive integer). In these cases, a big amount of extra computation is implemented though the ratio of extra computation to whole computation is small for relatively large graphs.

To avoid a drastic performance degradation for such worst cases, our APSP implementation chooses a near optimal blocking factor as a function of a problem size, instead of using a fixed blocking factor. The optimal blocking factors have been found empirically by computer experiments. Figure 8 shows the performance of the APSP implementation with this optimization. Figure 9 compares the performance using the optimal blocking factor with the one using a predefined blocking factor $b = 2048$ on System A for

Table 2 Running time in milliseconds of different APSP implementations computing both shortest-path distance matrix and the shortest-path construction matrix.

Platform	Number of vertices [n]								
	128	256	512	1024	2048	4096	8192	16384	32768
System A	3.87	4.13	8.50	23.9	105	446	2,501	16,962	129,786
System B	4.11	4.93	9.56	27.9	114	557	2,840	19,356	148,350
Cayman GPU	3.35	4.30	7.41	20.5	91.3	524	4,096	-	-
Cypress GPU	3.22	4.61	10.0	27.8	115	574	-	-	-
Sandy Bridge CPU	1.09	3.91	11.8	78.8	664	4,984	38,148	295,886	2,396,606
Bloomfield CPU	1.74	5.75	20.4	122	887	6,540	49,269	384,177	3,016,044

Table 3 Running time in milliseconds of different APSP implementations computing a shortest-path distance matrix only.

Platform	Number of vertices [n]								
	128	256	512	1024	2048	4096	8192	16384	32768
System A	4.12	3.98	7.13	16.3	60.2	294	1,494	10,146	75,018
System B	3.89	4.21	7.09	17.4	63.0	241	1,362	9,259	70,816
Cayman GPU	2.80	4.80	8.01	17.4	57.9	304	2,035	-	-
Cypress GPU	3.39	11.1	14.2	17.6	58.7	305	-	-	-
Sandy Bridge CPU	0.54	1.38	4.44	24.3	182	1,371	10,647	83,870	702,600
Bloomfield CPU	0.71	2.38	8.34	41.4	279	2,277	17,516	141,347	1,110,415

$4097 \leq n \leq 8192$. Choosing the optimal blocking factor prevents the drastic performance degradation. For example, when $n = 6145$ ($= 3 \cdot 2048 + 1$), the performance with the optimization is 752 GFlop/s while the one with $b = 2048$ is 389 GFlop/s only. A performance variability is larger on System B than System A, and big differences are seen when $n \leq 4992$ on System B and $n \leq 1408$ on System A.

4.4 Performance Comparison

Finally, we compare the performance of our APSP implementations on the CPU-GPU systems with APSP implementations on a GPU or a CPU (see Table 1 for the specification of the processors). The implementation on the GPU is the SubBlockedAPSP kernel (Algorithm 4), which is described in Sect. 4.2.1, with including the communication of matrix data between the host and the GPU. The implementation on the CPU is also based on the sub-blocked Algorithm 4 without the merging operation. Moreover, it utilizes OpenMP (Open Multi-Processing) directives and eight-way Intel AVX (Advanced Vector Extensions) instructions on the Sandy Bridge CPU (Core i7 2600K) or four-way SSE (Streaming SIMD Extensions) instructions on the Bloomfield CPU (Core i7 970).

To implement a single "multiply-add" operation for computing both shortest-path distance and construction matrices on the CPUs, we need four AVX/SSE instructions (addition, comparison, and two conditional move instructions), which are same as for the GPU. Since the CPUs do not support a dual issue of any combination of the four instructions, we can expect at most half the peak performance, i.e., 108.8 GFlop/s on the Sandy Bridge and 76.8 GFlop/s on the Bloomfield. When we apply the Eq. (3) to calculate the CPU performance, we obtain, as a maximum, 60 GFlop/s on the Sandy Bridge and 46 GFlop/s on the Bloomfield. On the other hand, to implement computing of a shortest-path distance matrix only, two AVX/SSE instructions (addition and minimum) are needed. Therefore, the total number of operations for the computation is $2n^3$, and the performance of the implementation is up to 105 GFlop/s on the Sandy Bridge and 65 GFlop/s on the Bloomfield. The CPU implementation for computing both matrices requires more number of vector registers by price of worsening memory access regularity. It leads to the lower performance than computing a distance matrix only.

Table 2 shows the running time of the different implementations for computing both distance and construction matrices. As it can be seen from the results, the APSP implementation on System A is the fastest when the problem size $n \geq 4096$. When the problem size is relatively small (128–256), the Sandy Bridge outperforms the other implementations.

We also compare APSP implementations for computing a shortest-path distance matrix only (see Table 3). The running time of the implementation on System B is shorter than the one presented in our previous paper [17]. In this work, for updating the pivot block $D_{K,K}$ in Algorithm 3, we use the sub-blocked APSP Algorithm 4 without construction matrix. This algorithm requires $2b^3$ operations while the algorithm used in our previous paper requires $2b^3 \log_2 b$ operations.

Table 3 shows System B runs faster than System A when $n \geq 4096$. The maximum performance of the MIN-PLUS kernel for distance matrix is 1303 GFlop/s on the Cayman and it is higher than 1068 GFlop/s on the Cypress. As mentioned above, the GPU computation cannot be overlapped with the data communication on System A while it is possible on System B. The computing of distance matrix requires two $b \times b$ matrices in single precision for $2b^3$ operations, and thus, the arithmetic intensity can be estimated as

$$Arithmetic\_Intensity_{APSP}(b) = \frac{2b^3}{4 \cdot 2b^2} = \frac{b}{4},$$

which is 4/3 times smaller than the one in Eq. (4). The smaller arithmetic intensity indicates that influence of the data communication time on the running time of System A becomes stronger.

5. Conclusion

This paper has presented a blocked united all-pairs shortest paths (APSP) algorithm for hybrid CPU-GPU systems. The blocked algorithm computes the shortest-path distance matrix and the shortest-path construction matrix at the same time. We have designed the algorithm so that the amount of data communication between CPU (host) and GPU is reduced. In the blocked algorithm, the matrix-matrix "multiplication" for the APSP problem plays the key role for the high performance. The APSP algorithm has been implemented on the two kinds of CPU-GPU systems. The implementation runs close to the peak performance when a problem size (the number of vertices in a graph) is larger than a few thousands. The sustained performance is 1.1 TFlop/s in single precision on a system containing an Intel Sandy Bridge CPU (Core i7 2600K) and an AMD Cayman GPU (Radeon HD 6970). We guess that the proposed blocked algorithm is applicable to other hybrid systems containing GPU-like accelerators.

Future work includes a further extension of the blocked APSP algorithm to cluster systems consisting of multiple CPU-GPU nodes.

Acknowledgment

The authors wish to thank the anonymous reviewers for their insightful comments and helpful suggestions which have improved this paper.

References

[1] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein, Introduction to Algorithms, 3rd ed., The MIT Press, Massachusetts, USA, 2009.
[2] C.H. Papadimitriou and K. Steiglitz, Combinatorial Optimization: Algorithms and Complexity, Dover Publications, NY, USA, 1998.
[3] G. Rote, "Path problems in graphs," Computing Supplementum, vol.7, pp.155–198, 1990.
[4] K. Matsumoto and S.G. Sedukhin, "The algebraic path problem on the Cell/B.E. processor," Tech. Rep. 2010-002, The University of Aizu, 2010. ftp://ftp.u-aizu.ac.jp/pub/u-aizu/doc/TechReport/2010/2010-002.pdf
[5] R.W. Floyd, "Algorithm 97: Shortest path," Commun. ACM, vol.5, no.6, p.345, 1962.
[6] S. Warshall, "A theorem on boolean matrices," J. ACM, vol.9, no.1, pp.11–12, 1962.
[7] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," J. Experimental Algorithmics, vol.8, p.2.2, 2003.
[8] M. Penner and V.K. Prasanna, "Cache-friendly implementations of transitive closure," J. Experimental Algorithmics, vol.11, p.1.2, 2006.
[9] P. D'Alberto and A. Nicolau, "R-Kleene: A high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks," Algorithmica, vol.47, no.2, pp.203–213, Feb. 2007.
[10] S.C. Han, F. Franchetti, and M. Püschel, "Program generation for the all-pairs shortest path problem," Proc. 15th International Conference on Parallel Architectures and Compilation Techniques (PACT '06), pp.222–232, Seattle, Washington, USA, Sept. 2006.
[11] K. Matsumoto and S.G. Sedukhin, "A solution of the all-pairs shortest paths problem on the Cell Broadband Engine processor," IEICE Trans. Inf. & Syst., vol.E92-D, no.6, pp.1225–1231, June 2009.
[12] S. Vinjamuri and V.K. Prasanna, "Transitive closure on the cell broadband engine: A study on self-scheduling in a multicore processor," Proc. 23rd IEEE International Symposium on Parallel and Distributed Processing (IPDPS 2009), pp.1–11, Rome, Italy, IEEE Computer Society, May 2009.
[13] G.J. Katz and J.T. Kider, Jr., "All-pairs shortest-paths for large graphs on the GPU," Proc. 23rd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware (GH 2008), pp.47–55, Sarajevo, Bosnia and Herzegovina, June 2008.
[14] A. Buluç, J.R. Gilbert, and C. Budak, "Solving path problems on the GPU," Parallel Comput., vol.36, no.5-6, pp.241–253, June 2010.
[15] P. Harish, V. Vineet, and P.J. Narayanan, "Large graph algorithms for massively multithreaded architectures," Tech. Rep. IIIT/TR/2009/74, International Institute of Information Technology, 2009. http://cvit.iiit.ac.in/papers/pawan09GraphAlgorithms.pdf
[16] T. Okuyama, F. Ino, and K. Hagihara, "Fast blocked Floyd-Warshall algorithm on the GPU," IPSJ Transactions on Advanced Computing Systems, vol.3, no.2, pp.57–66, June 2010 [in Japanese].
[17] K. Matsumoto, N. Nakasato, and S.G. Sedukhin, "Blocked all-pairs shortest paths algorithm for hybrid CPU-GPU system," Proc. 13th IEEE International Conference on High Performance Computing and Communications (HPCC-2011), pp.145–152, Banff, Canada, IEEE Computer Society Press, Sept. 2011.
[18] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, and P. Sadayappan, "Parallel FPGA-based all-pairs shortest-paths in a directed graph," Proc. 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2006), pp.1–10, Rhodes Island, Greece, April 2006.
[19] F. Baccelli, G. Cohen, G.J. Olsder, and J.P. Quadrat, Synchronization and Linearity, An Algebra for Discrete Event Systems, John Wiley & Sons, Chichester, West Sussex, UK, July 1993.
[20] K. Matsumoto and S.G. Sedukhin, "Matrix multiply-add in min-plus algebra on a short-vector SIMD processor of Cell/B.E.," WANC: Proc. First International Conference on Networking and Computing (ICNC'10), pp.272–274, Hiroshima, Japan, Nov. 2010.
[21] J.E. Pin, "Tropical semirings," in Idempotency, Publications of the Newton Institute, no.11, ed. J. Gunawardena, pp.50–69, Cambridge University Press, Cambridge, UK, 1998.
[22] AMD Inc., "AMD Accelerated Parallel Processing OpenCL Programming Guide, rev1.3f," Aug. 2011.
[23] K. Madduri and D.A. Bader, "GTgraph: A suite of synthetic random graph generators." http://www.cse.psu.edu/~madduri/software/GTgraph, accessed Jan. 4, 2012.
[24] Khronos Group, "OpenCL - The open standard for parallel programming of heterogeneous systems." http://www.khronos.org/opencl, accessed Jan. 4, 2012.
[25] N. Nakasato, "A fast GEMM implementation on the Cypress GPU," ACM SIGMETRICS Performance Evaluation Review, vol.38, no.4, pp.50–55, March 2011.
[26] K. Matsumoto, N. Nakasato, T. Sakai, H. Yahagi, and S.G. Sedukhin, "Multi-level optimization of matrix multiplication for GPU-equipped systems," Procedia Computer Science, vol.4, pp.342–351, June 2011.
[27] AMD Inc., "AMD Intermediate Language (IL) Reference Guide, rev 2.4," Oct. 2011.
[28] AMD Inc., "AMD Compute Abstraction Layer (CAL) Program-

ming, rev2.03," Dec. 2010.
[29] S.W. Williams, "The roofline model," in Performance Tuning of Scientific Applications, ed. D.H. Bailey, R.F. Lucas, and S.W. Williams, ch.9, pp.195–215, CRC Press, FL, USA, 2011.

Kazuya Matsumoto is a Ph.D. student at the University of Aizu. He received Bachelor degree and Master degree in Computer Science and Engineering from the University of Aizu in 2008 and 2010, respectively. His current research interests include parallel and distributed computing, program optimization and tuning, and design and evaluation of parallel algorithms. He is a student member of IEEE Computer Society and IPSJ.

Naohito Nakasato is assistant professor of Department of Computer Science and Engineering at University of Aizu, Japan and visiting associate professor of Center for Computational Sciences at University of Tsukuba, Japan. He had obtained Ph.D in Science (Astronomy) from University of Tokyo, 2000. He is a member of International Astronomical Union, Astronomical Society of Japan, Information Processing Society of Japan and IEEE. His research interests are high performance computing and numerical modeling in astrophysics.

Stanislav G. Sedukhin is a Professor and the Vice-President of the University of Aizu. He received his Ph.D. and Dr.Sci. (Habilitation) from the Russian Academy of Sciences in 1982 and 1993, respectively. His research interests are in architectural co-design of application-specific array processors and massively-parallel algorithms. Dr. Sedukhin is a member of ACM, and IEEE CS.