Technical Report 2010-002

The Algebraic Path Problem on the Cell/B.E. Processor

Kazuya Matsumoto, Stanislav G. Sedukhin

November 30, 2010

Graduate Department of Computer and Information Systems
The University of Aizu
Tsuruga, Ikki-Machi, Aizu-Wakamatsu City
Fukushima, 965-8580 Japan

Technical Report 2010-002

Title:
The Algebraic Path Problem on the Cell/B.E. Processor

Authors:
Kazuya Matsumoto, Stanislav G. Sedukhin

Key Words and Phrases:
algebraic path problem, all-pairs shortest paths problem, Cell Broadband Engine, performance evaluation, parallel computing

Abstract:
The Algebraic Path Problem (APP) unifies well-known matrix, graph, and language problems, such as matrix inversion, all-pairs shortest paths (APSP), maximum capacity paths (MCP), minimum spanning tree, generation of regular languages, etc., into a single algorithmic scheme. The difference between APP instances is in the underlying algebraic structure. This paper explores the APP and presents an implementation of a block algorithm for solving the APP on the Cell Broadband Engine (Cell/B.E.) heterogeneous multicore processor. The block APP algorithm spends the most computing time in a block matrix-matrix multiply-add (MMA) operation in different algebras. In our APP algorithm, a fast dense MMA operation in linear $(+, \times)$-algebra is utilized. The MMA implementation on the Cell/B.E. needs only a single fused multiply-add (FMA) instruction to obtain a single short-vector $(+, \times)$-result in one cycle. APP instances such as APSP and MCP problems are based on $(\min, +)$- and $(\max, \min)$-algebras, respectively, which are different from the linear $(+, \times)$-algebra, and require three and four instructions to obtain a single short-vector result in three and four cycles. Because of that, the maximum sustained performance for MMA operation on Cell/B.E. is 152 Gflop/s whereas for APSP and MCP are 50.7 Gflop/s and 38.1 Gflop/s, respectively.

Report Date:
11/30/2010

Written Language:
English

Any Other Identifying Information of this Report:
Manuscript submitted to Journal of Information Processing (IPSJJIP)

Distribution Statement:
First Issue: 10 copies

Supplementary Notes:

Distributed Parallel Processing Laboratory
The University of Aizu
Aizu-Wakamatsu
Fukushima 965-8580
Japan

The Algebraic Path Problem on the Cell/B.E. Processor

Kazuya Matsumoto Stanislav G. Sedukhin

Abstract

The Algebraic Path Problem (APP) unifies well-known matrix, graph, and language problems, such as matrix inversion, all-pairs shortest paths (APSP), maximum capacity paths (MCP), minimum spanning tree, generation of regular languages, etc., into a single algorithmic scheme. The difference between APP instances is in the underlying algebraic structure. This paper explores the APP and presents an implementation of a block algorithm for solving the APP on the Cell Broadband Engine (Cell/B.E.) heterogeneous multicore processor. The block APP algorithm spends the most computing time in a block matrix-matrix multiply-add (MMA) operation in different algebras. In our APP algorithm, a fast dense MMA operation in linear $(+, \times)$-algebra is utilized. The MMA implementation on the Cell/B.E. needs only a single fused multiply-add (FMA) instruction to obtain a single short-vector $(+, \times)$-result in one cycle. APP instances such as APSP and MCP problems are based on $(\min, +)$- and $(\max, \min)$-algebras, respectively, which are different from the linear $(+, \times)$-algebra, and require three and four instructions to obtain a single short-vector result in three and four cycles. Because of that, the maximum sustained performance for MMA operation on Cell/B.E. is 152 Gflop/s whereas for APSP and MCP are 50.7 Gflop/s and 38.1 Gflop/s, respectively.

1 Introduction

The *Algebraic Path Problem* (APP) is a general framework which unifies several solution procedures for a number of problems into a single algorithmic

formulation. Specific instances of the APP include problems from linear algebra (e.g. computing the inverse of a real non-singular matrix), from graph theory (e.g. the transitive and reflexive closure, the all-pairs shortest paths problem, the minimum-cost spanning tree problem, etc.) as well as from language processing (e.g. generation of regular languages). These problems are among the most important computational problems in computer science and engineering area. The applications of the APP can be found in bioinformatics, network routing, control theory, and many others. As a result, the APP has been widely studied [1, 2, 3, 4, 5, 6]. The difference between many instances of the APP is in the underlying algebraic structure expressed as a *closed semiring*.

There are many algorithms for solving the different instances of the APP. Among APP algorithms, a simple algorithm is well-known because this unified algorithm is generalization of Warshall's algorithm for transitive closure, Floyd's algorithm for all-pairs shortest-paths problem [7], and the Gauss-Jordan method for matrix inversion [8]. The APP algorithm consists of triple-nested loop as in the dense matrix-matrix multiply-add (MMA) algorithm, except it has more strict data dependencies such that the outermost loop cannot be interchanged.

To effectively use memory hierarchy in state-of-art processors, blocking algorithm is necessary to relax the required memory bandwidth and to obtain the high performance. Block Floyd-Warshall (FW) algorithm and its data dependency have been discussed in [9, 10, 11, 7]. The blocking approach can be applied to solve other APP instances. The most compute intensive part of the block FW algorithm is the MMA in different semirings. This fact enables us to leverage a highly tuned MMA implementation for efficient computing of the APP.

In this paper, we extend the previously designed block FW algorithm [7] to other instances of APP and show a parallel implementation of the unified block APP algorithm on the Cell Broadband Engine processor (Cell/B.E.). The Cell/B.E. is a heterogeneous multicore processor consisting of a general purpose processor, the Power Processor Element (PPE), and eight short-vector SIMD processors, so called the Synergistic Processor Elements (SPEs).

As other contribution, this paper discusses how different optimization techniques affect the performance on the Cell/B.E. The techniques include com-

puting with a block data layout, overlapping the computation with the data communication, parallelizing the block algorithm, and reducing the number of required registers in matrix "multiply-add" kernel.

The remainder of this paper is organized as follows. In Section 2, the overview of the APP and its applications are given, and then the scalar and block algorithms for the APP are introduced. In Section 3, the architecture of the Cell/B.E. is described in details. In Section 4, the discussion how the block algorithm for the APP is implemented on the Cell/B.E. is conducted. In Section 5, the results of performance evaluation are presented. Finally, Section 6 describes conclusions and future work.

2 Algebraic Path Problem

2.1 Notations and Definitions

Let us consider a weighted graph $G = (V, E, w)$ with a set of n vertices $V = \{1, 2, \ldots, n\}$, a set of edges $E \subseteq V \times V$, and a weight function $w : E \to S$ which assigns to each edge a weight from a special algebraic structure referred to as *closed semiring* $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$ where S is a set of elements; the "addition" $\oplus$ and the "multiplication" $\otimes$ are binary operations $(S \times S \to S)$; $*$ is a unary operation $(S \to S)$ called *closure*; and the zero $\bar{0}$ and the unity $\bar{1}$ are constants in S. The further discussion including the properties of the closed semiring can be found in [12, 3, 13, 14].

Let a path p in G be an arbitrary sequence of vertices $p = (i, k_1, k_2, \ldots, k_m, j)$ which begins with i and ends with j. We then define a weight $w(p)$ of the path p as the product of all edges of the path:

$$w(p) = w(i, k_1) \otimes w(k_1, k_2) \otimes \cdots \otimes w(k_m, j)$$

Let $P(i, j)$ denote the set of all paths from i to j. Then the algebraic path problem (APP) is to compute the "sum" $d_{i,j}$ of $P(i, j)$ for all (i, j) pairs:

$$d_{i,j} = \bigoplus_{p \in P(i,j)} w(p). \tag{1}$$

The APP can also be formulated in a matrix form [12, 3] by introducing a matrix closed semiring $(S^{n \times n}, \oplus, \otimes, *, \bar{O}, \bar{I})$ over the scalar closed semiring

$(S, \oplus, \otimes, *, \bar{0}, \bar{1})$, where $S^{n\times n}$ is a set of $n \times n$ matrices; the matrix "addition" $\oplus$ and the matrix "multiplication" $\otimes$ are binary operations ($S^{n\times n} \times S^{n\times n} \to S^{n\times n}$); $*$ is a unary operation ($S^{n\times n} \to S^{n\times n}$) called *closure of a matrix*; the zero matrix $\bar{O}$ the elements of which are all $\bar{0}$s, and a unity matrix $\bar{I}$ with $\bar{1}$s on the main diagonal and $\bar{0}$s otherwise, are constant matrices.

We associate an initial matrix $A = (a_{i,j})$ in $S^{n\times n}$ with the weighted graph G, where

$$a_{i,j} = \begin{cases} w(i,j) & \text{if } (i,j) \in E, \\ \bar{0} & \text{if } (i,j) \notin E. \end{cases} \tag{2}$$

The matrix operations of "addition" and "multiplication" are defined as in linear algebra: if $A = (a_{i,j})$ and $B = (b_{i,j})$ are $n \times n$ matrices, then the "addition" and the "multiplication" are

$$\begin{aligned} A \oplus B &= (c_{i,j}), \text{ where } c_{i,j} = a_{i,j} \oplus b_{i,j}; \\ A \otimes B &= (c_{i,j}), \text{ where } c_{i,j} = \textstyle\bigoplus_{k=1}^{n} a_{i,k} \otimes b_{k,j}. \end{aligned}$$

When we define an $n \times n$ matrix $D = (d_{i,j})$ of the elements specified in (1), we get the matrix formulation for the APP in terms of the matrix A, as follows:

$$D = A^* = \bigoplus_{m\geq 0} A^m = \bar{I} \otimes A \otimes (A \oplus A) \otimes (A \oplus A \oplus A) \otimes \cdots. \tag{3}$$

The matrix A^*, which is the closure of matrix A, can be rewritten as follows:

$$\begin{aligned} A^* &= \bar{I} \oplus A \oplus A^2 \oplus A^3 \oplus \cdots \\ &= \bar{I} \oplus A \otimes (\bar{I} \oplus A \oplus A^2 \oplus A^3 \cdots) \qquad (4) \\ &= \bar{I} \oplus A \otimes A^*. \qquad (5) \end{aligned}$$

Then the APP in the matrix form is to solve (5) for A^*. Note that for any idempotent semiring[1], the infinite sum (4) converges to a finite sum, because $A^{n+i} = A^n$ for $i > 0$.

2.2 Scalar algorithm

We denote the initial elements of a matrix A by $a_{i,j}^{(0)} = a_{i,j}$. The following recurrence equation (6) is used to update the elements of a matrix $A^{(k)} = (a_{i,j}^{(k)})$

[1] An idempotent semiring holds a property of $a = a \oplus a$ [15].

for $k = 1$ to n **do**
  $a_{k,k}^{(k)} \leftarrow (a_{k,k}^{(k-1)})^*$; % "Black" element update
  for all $1 \le i \le n$ **do**
    $a_{i,k}^{(k)} \leftarrow a_{i,k}^{(k-1)} \otimes a_{k,k}^{(k)}$; % "Red" elements update
  end for
  for all $1 \le i \le n$ $(i \ne k)$ **do**
    for all $1 \le j \le n$ $(j \ne k)$ **do**
      $a_{i,j}^{(k)} \leftarrow a_{i,k}^{(k)} \otimes a_{k,j}^{(k-1)} \oplus a_{i,j}^{(k-1)}$; % "White" elements update
    end for
  end for
  for all $1 \le j \le n$ **do**
    $a_{k,j}^{(k)} \leftarrow a_{k,k}^{(k)} \otimes a_{k,j}^{(k-1)}$; % "Blue" elements update
  end for
end for

Figure 1: Scalar algorithm for solving the algebraic path problem

$(1 \le i, j \le n$ on each iteration $k = 1, 2, \cdots, n)$.

$$a_{i,j}^{(k)} \leftarrow \begin{cases} (a_{i,j}^{(k-1)})^* & \text{if } i = j = k; \\ (a_{k,k}^{(k-1)})^* \otimes a_{k,j}^{(k-1)} & \text{if } i = k \ne j; \\ a_{i,k}^{(k-1)} \otimes (a_{k,k}^{(k-1)})^* \otimes a_{k,j}^{(k-1)} \oplus a_{i,j}^{(k-1)} & \text{if } i \ne k \text{ and } j \ne k; \\ a_{i,k}^{(k-1)} \otimes (a_{k,k}^{(k-1)})^* & \text{if } j = k \ne i. \end{cases} \tag{6}$$

We can cast the recurrence (6) into a scalar algorithm shown in Fig. 1. As can be seen from (6) and Fig. 1, the algorithm consists of four kinds of updates on each k-th iteration $(k = 1, 2, \cdots, n)$. The algorithm updates the "black" element (pivot) first. Then it updates the "red" elements (pivot column), the "white" elements (non-pivot), and the "blue" elements (pivot row) in order.

For any idempotent semiring, the closure is $a_{k,k}^{(k)} = (a_{k,k}^{(k-1)})^* = \bar{1}$ and the pivot is $a_{k,k}^{(k-1)} = \bar{1}$ for any $k \in \{1, 2, \cdots, n\}$. This fact indicates that $a_{k,j}^{(k)} = (a_{k,k}^{(k-1)})^* \otimes a_{k,j}^{(k-1)} \equiv a_{k,j}^{(k-1)}$ for $i = k$, and $a_{i,k}^{(k)} = a_{i,k}^{(k-1)} \otimes (a_{k,k}^{(k-1)})^* \equiv a_{i,k}^{(k-1)}$ for $j = k$. Thus, for the idempotent semiring, the scalar algorithm in Fig. 1 can be simplified as it is shown in Fig. 2. It is clear that the total number of combined $(\oplus, \otimes)$ operations in the simplified algorithm is

$$\mathcal{N}_{(\oplus,\otimes)}^{\text{scalar}}(n) = n^3 - (2n^2 - n) = n(n-1)^2. \tag{7}$$

for $k = 1$ to n **do**

  for all $1 \leq i, j \leq n$ ($i \neq k$ and $j \neq k$) **do**

    $a_{i,j}^{(k)} \leftarrow a_{i,k}^{(k-1)} \otimes a_{k,j}^{(k-1)} \oplus a_{i,j}^{(k-1)}$;

  end for

end for

Figure 2: Scalar algorithm for solving the APP in an idempotent semiring

2.3 Instances of the Algebraic Path Problem

We can define different instances of the algebraic path problem by specializing closed semiring $(S, \oplus, \otimes, *, \bar{0}, \bar{1})$. Several instances are detailed in following:

- *Transitive and reflexive closure*: the weights $a_{i,j}$ are taken from $S \subseteq \{0, 1\}$. $\oplus = \vee, \otimes = \wedge, a^* = 1$ for all a in S, $\bar{0} = 0$, and $\bar{1} = 1$. The closure of a matrix $A = (a_{i,j})$ gives the transitive and reflexive closure.

- *All-pairs shortest paths* (APSP): the weights $a_{i,j}$ are taken from $S \subseteq R_+ \cup \{\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \min, \otimes = +, a^* = 0$ for all a in S, $\bar{0} = \infty$, and $\bar{1} = 0$. The closure of a matrix $A = (a_{i,j})$ gives the all-pairs shortest paths.

- *Critical paths* (CRP), also called *maximum cost paths*: the weights $a_{i,j}$ are taken from $S \subseteq R_+ \cup \{+\infty, -\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \max, \otimes = +, a^* = 0$ for all a in S, $\bar{0} = -\infty$, and $\bar{1} = 0$. The closure of a matrix $A = (a_{i,j})$ gives the maximum cost paths, or $+\infty$ if there are paths of unbounded cost [16].

- *Maximum capacity paths* (MCP), also called *tunnel problem* or *network capacity problem* [1]: the weights $a_{i,j}$ are taken from $S \subseteq R_+ \cup \{\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \max, \otimes = \min, a^* = \infty$ for all a in S, $\bar{0} = 0$, and $\bar{1} = \infty$. The closure of a matrix $A = (a_{i,j})$ gives the maximum capacity paths.

- *Maximum reliability paths* (MRP): the weights $a_{i,j}$ are taken from $S \subseteq \{0, 1\}$ and can be considered as reliabilities of the information transport between the two connected vertices. $\oplus = \max, \otimes = \times, a^* = 1$ for all a in S, $\bar{0} = 0$, and $\bar{1} = 1$. The closure of a matrix $A = (a_{i,j})$ gives the most reliability paths.

- *Minimum cost spanning tree* (MST): the weights $a_{i,j} = a_{j,i}$ are taken from $S \subseteq R_+ \cup \{\infty\}$, where R_+ is the set of positive real numbers. $\oplus = \min, \otimes = \max, a^* = 0$ for all a in S, $\bar{0} = \infty$, and $\bar{1} = 0$. Let $D = (d_{i,j})$ is the closure of a matrix $A = (a_{i,j})$, then its entries $d_{i,j} = a_{i,j}^{(0)}$ are the edges of the minimum spanning tree[17].

- *Inverse of a real non-singular matrix*: the weights $a_{i,j}$ are taken from $S \subseteq R$. $\oplus$ and $\otimes$ are the conventional arithmetic $+$ and $\times$ operations on R respectively, $a^* = 1/(1-a)$ for all a in R with $a \neq 1$ (a^* is undefined for $a = 1$), $\bar{0} = 0$, and $\bar{1} = 1$. The closure of a matrix $A = (a_{i,j})$ gives $(I - A)^{-1}$ (see [3],[8]).

2.4 Block algorithm

Block algorithms are widely known to be more efficient in many computing environments than the corresponding scalar algorithms because block algorithms enable to relax the required memory bandwidth. Figure 3 shows a block algorithm for solving the algebraic path problem. In the block algorithm, the initial $n \times n$ matrix A is divided into an $N \times N$ matrix of $b \times b$ blocks where $N = n/b$ and $b \quad (2 \leq b \leq n/2)$ is the blocking factor. For simplicity, but without loss of generality, the matrix size is assumed to be in multiples of the blocking factor b.

On each K-th iteration ($K = 1, 2, \cdots, N$) of the outermost loop, the block algorithm works as follows: it first updates the "black" block $A_{K,K}$ (line 2) by applying the closure operation to the $b \times b$ subproblem $A_{K,K}$, which is solved by the scalar algorithm of Fig. 2, then it updates $N - 1$ "red" blocks $A_{I,K}$ (lines 3-5) and $N - 1$ "blue" blocks $A_{K,J}$ (lines 6-8), and, finally, it updates $(N-1)^2$ "white" blocks $A_{I,J}$ (lines 9-13).

On each iteration, the "black" block update requires $b(b-1)^2$ ($= \mathcal{N}_{(\oplus,\otimes)}^{\text{scalar}}(b)$ in (7)) "multiply-add" operations; the "red" blocks update and the "blue" blocks update require $b^3(n/b - 1)$ "multiply-add" operations each; and the "white" blocks update requires $b^3(n/b-1)^2$ "multiply-add" operations. Thus the total number of "multiply-add" operations is

$$\begin{aligned} \mathcal{N}_{(\oplus,\otimes)}^{\text{block}}(n,b) &= b(b-1)^2 + 2b^3(n/b-1) + b^3(n/b-1)^2 \\ &= n(n^2 - 2b + 1). \end{aligned} \tag{8}$$

1: **for** $K = 1$ to N **do**
2: $A^{(K)}_{K,K} \leftarrow (A^{(K-1)}_{K,K})^*$; % "Black" block update
3: **for all** $1 \le I \le N$ $(I \ne K)$ **do**
4: $A^{(K)}_{I,K} \leftarrow A^{(K-1)}_{I,K} \otimes A^{(K)}_{K,K} \oplus \bar{O}$; % "Red" blocks update
5: **end for**
6: **for all** $1 \le J \le N$ $(J \ne K)$ **do**
7: $A^{(K)}_{K,J} \leftarrow A^{(K)}_{K,K} \otimes A^{(K-1)}_{KJ} \oplus \bar{O}$; % "Blue" blocks update
8: **end for**
9: **for all** $1 \le I \le N$ $(I \ne K)$ **do**
10: **for all** $1 \le J \le N$ $(J \ne K)$ **do**
11: $A^{(K)}_{I,J} \leftarrow A^{(K)}_{I,K} \otimes A^{(K)}_{K,J} \oplus A^{(K-1)}_{I,J}$;% "White" blocks update
12: **end for**
13: **end for**
14: **end for**

Figure 3: Block algorithm for solving the algebraic path problem

The total number of block operations is

$$\mathcal{N}_{\text{total}}(n,b) = n/b(1 + 2(n/b-1) + (n/b-1)^2) = (n/b)^3.$$

On the other hand, the number of block operations for updating "red" (R), "blue" (B), and "white" (W) blocks is

$$\mathcal{N}_{\text{RBW}}(n,b) = n/b(2(n/b-1) + (n/b-1)^2) = (n/b)^3 - n/b.$$

Then, the ratio is

$$\rho = \frac{\mathcal{N}_{\text{RBW}}}{\mathcal{N}_{\text{total}}} \times 100\% = (1 - \frac{1}{(n/b)^2}) \times 100\%.$$

The ratio shows that, when (n/b) is relatively large, the block algorithm spends the most computing time to update "red", "blue", and "white" blocks; see Fig. 4, where the problem size n is measured in multiples of b.

The "red", "blue" and "white" blocks are updated using matrix-matrix "multiply-add" (MMA) in different algebraic semirings. As it was shown, the MMA operation is the most compute-intensive part (kernel) in the block algorithm. Hence the computational performance of algorithm strongly depends on the performance of MMA operation.

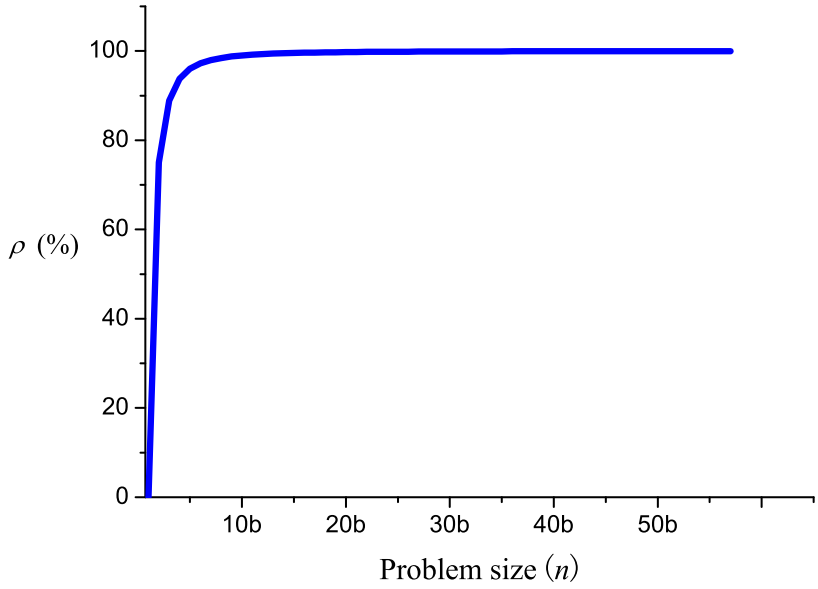


Figure 4: Ratio of the number of operations for updating "red", "blue", and "white" blocks compared to the total number

Note that here the block algorithm is used for a solution of the APP with idempotent semiring, nevertheless, it can also be used with several modifications as a solution for non-idempotent semirings [8]. For instance, to compute the inverse of a matrix, we have to make the following modifications to the algorithm in Fig. 3:

- replacing the statement 4 with $A_{I,K} \leftarrow -A_{I,K} \otimes A_{K,K}$ in the "red" blocks update;
- permuting the order of updates from "black → red → blue → white" to "black → red → white → blue", and changing the superscripts of statements accordingly;
- using the scalar algorithm (Fig. 1), instead of the simplified algorithm (Fig. 2), for the "black" block update.

3 Cell Broadband Engine

The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multicore processor. The Cell/B.E. consists of one Power Processor Element (PPE), eight

Synergistic Processor Elements (SPEs), a Memory Interface Controller (MIC), a Cell Broadband Engine Interface (BEI), and an Element Interconnect Bus (EIB) [18, 19].

PPE is a 64-bit Power Architecture based general purpose processor. PPE usually acts as the control center of the Cell/B.E.; it distributes computational workloads among SPEs and coordinates an entire operation.

SPEs are 128-bit RISC processors with a dual-issue pipelined four-way SIMD unit. Each SPE contains a Synergistic Processor Unit (SPU) and a Memory Flow Controller (MFC) [2]. SPU does not have data caches, but instead, has its own 256 kB local memory called *local store* (LS). The register file of SPU contains 128 128-bit registers. Each SPE can access the host memory and the LS of other SPEs with direct memory access (DMA) transfers. The single-precision floating point peak performance of each SPU at 3.2 GHz clock speed is 25.6 Gflop/s because each SPU can compute eight single-precision floating point operations per clock with the four-way SIMD fused multiply-add (FMA) instruction [20].

EIB is the communication path between all the components of Cell/B.E. and is composed of four data rings, where the theoretical peak data bandwidth at 3.2 GHz is 204.8 GByte/s. The peak data bandwidth between the XDR DRAM memory (host memory) and EIB is, however, 25.6 GByte/s.

4 Porting the APP Algorithm to the Cell/B.E.

4.1 Parallelization

The block algorithm (Fig. 3) has been implemented on the Cell/B.E. so that the PPE updates the "black" block and then the SPEs update the "red", "blue" and "white" blocks. It is essential for high-speed computation considering how to parallelize the algorithm for the heterogeneous multi-core processor. The block algorithm sees an initial $n \times n$ matrix $A = (a_{i,j})$ as the $n/b \times n/b$ matrix $A = (A_{I,J})$ consisting of $b \times b$ blocks, where n is the problem size and

[2]The term "SPU" refers to the instruction set or the unit that executes the instruction set, and the term "SPE" refers generally to functionality of any part of the SPE processing element, including the MFC [18].

b is the block factor. Hence we can parallelize the algorithm in terms of $b \times b$ independent block as a unit of computational workloads.

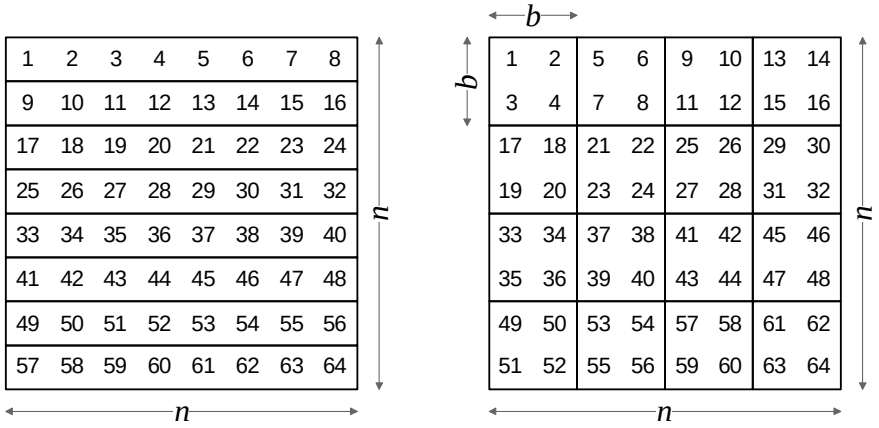
To correctly parallelize the block algorithm, the synchronization between all the processing elements at regular time-points is needed. The algorithm has data dependencies between colored blocks updates: the "red" and "blue" blocks depend on the "black" block; and the "white" blocks depend on both the "red" and "blue" blocks. Thus, we need to make at least two synchronization points: the first point is after the "black" block update; and the second point is before the "white" blocks update. To implement this barrier synchronization, we use a mailbox mechanism, which offers a 32-bit data passing between PPE and SPEs [18]. At either of the synchronization point, each SPE sends a notification mail to the PPE, and after the PPE receives all the notification mails, the PPE sends an acknowledgment mail to each SPE.

The "black" block update for K-th iteration ($K = 2, 3, \cdots, N$) cannot be started till the $A_{K,K}$ in "white" blocks is updated for the previous $(K-1)$-th iteration. This means that PPE can immediately start updating the "black" block when an SPE finishes updating the $A_{K,K}$ block for the previous iteration. For that reason, we have implemented the scheduling that the $A_{K,K}$ block is updated in the beginning of all "white" blocks updates.

4.2 Block size and data layout

In the block algorithm, a given $n \times n$ matrix is partitioned with $b \times b$ square blocks. This raises the question of the optimal block size for given architecture. We use 64×64 as the block size for the implementation. The size of block for data in single precision is 16 kB, which is the maximum data size to transfer with one DMA instruction; also, each the 16 kB block can be fitted in the 256 kB local store of SPE. Moreover, if block data layout [21] is used, which is discussed below, a whole block can be transferred with one DMA transfer instruction. Notice that for other existing block matrix algorithms on the Cell/B.E., it is a common practice to use blocks of 64×64 elements [22, 23, 24, 25].

A two-dimensional array of matrix data in C programming language is usually stored in *row-major data layout* (RDL), where all row elements are



(a) Row-major Data Layout (b) Block Data Layout

Figure 5: (a) Row-major Data Layout and (b) Block Data Layout; the number in each layout indicates the order of memory assignment.

stored continuously (see Fig. 5a). The block algorithm, however, deals with blocks of a matrix data; hence the implementation requires to process the data as the matrix of blocks. For such data access pattern to block data, it is often effective to use a special data layout called as *block data layout* (BDL) [26, 21] (see Fig. 5b).

4.3 Matrix-matrix multiply-add in different semirings

4.3.1 Differences between MMA and APP

As it was discussed above, the most compute intensive part of the block APP algorithm is in the "red", "blue", and "white" blocks updates. We can utilize an existing fast matrix-matrix multiply-add implementation for updating the blocks. The differences of the MMA implementation and the APP implementation are types and the number of instructions to obtain corresponding "multiply-add" result. The MMA implementation only requires a single fused multiply-add (FMA) instruction while other APP implementations need more number of instructions. The number and the type of instructions needed depend on an instance of the APP; each instance needs different kinds of algebraic operations, which include $(+, \times)$ operation for matrix inversion, $(\min, +)$ for shortest paths, $(\max, +)$ for critical paths, $(\max, \min)$ for maximum capacity paths, $(\max, \times)$ for maximum reliability paths, and $(\min, \max)$ for spanning

tree. In the SPU of the Cell/B.E., the standard addition and the multiplication operations are implemented by a single instruction (`fa` and `fm` assembly instruction [27], respectively), but the minimum or maximum operation require two instructions (combination of `fcgt` and `selb` for each). For example, the all-pairs shortest paths (APSP) problem requires three instructions to implement the addition operation and the minimum operation. The fact indicates that the APSP implementation on Cell/B.E. will take three times as many clock cycles as the MMA implementation; in other words, we can expect that the SPU's theoretical peak performance for APSP would be 8.533 (=25.6/3) Gflop/s. Also, the maximum capacity paths (MCP) problem requires four instructions to implement a minimum operation and a maximum operation. The fact indicates that the MCP implementation will take four times as many clock cycles as the MMA implementation; in other words, we can expect that the SPU's theoretical peak performance for MCP would be only 6.4 (=25.6/4) Gflop/s.

4.3.2 MMA implementations on the Cell/B.E.

Currently, there are three available MMA implementations on the Cell/B.E. [28, 24, 23]. Each implementation uses the block size of 64×64 and is deeply tuned by using four-way short-vector SIMD instructions as well as loop unrolling and software pipelining. The first MMA implementation was reported by Chen et al. [28]; it is written in C programming language and is distributed with IBM Cell SDK as a demo program. The other two implementations are written in an assembly language: Hackenberg reported a deeply unrolled implementation [24]; and Kurzak et al. [23] reported an ultimate MMA implementation for the SPU.

We have evaluated the performances of all the three implementations on the experimental environment explained in Section 5.1. Table 1 shows the summary of the different MMA implementations. The implementation by Kurzak et al. [23] is the best one from both aspects of the performance and register usage; hence, we have selected this MMA implementation to be utilized for our APP algorithm.

In fact, we have modified the selected MMA implementation so that the register usage is reduced. We need it because less register usage is desirable

Table 1: Performance of matrix multiply-add on one SPU

Algorithm	Performance (Gflop/s)	% of the peak	Register usage
Chen et al. [28]	23.77	92.85	69
Hackenberg [24]	25.40	99.23	71
Kurzak et al. [23]	25.49	99.56	69
This paper	25.44	99.38	49

for the APP implementation which requires more additional registers to store intermediate results. The modified MMA implementation uses only 49 registers although the obtained performance of 25.44 Gflop/s is almost same as in [23] (see Table 1). When we utilize the modified MMA algorithm for our APP implementation, it uses 65 registers; 16 additional registers are consumed to keep the intermediate results.

5 Performance Evaluation

5.1 Evaluation environment

The examination results presented here were measured on the Cell/B.E. processor of a PlayStation 3 (PS3). In the PS3, six of eight SPEs are only available because the other two are disabled to improve chip yields, and to run some operations on the background of PS3 Linux. We have installed Yellow Dog Linux 6.2 as the operating system and used Cell SDK 3.1 as the software development environment. Cell SDK 3.1 contains `gcc` version 4.1.1 based compilers (`ppu-gcc` and `spu-gcc`). As the optimization options for compilation, we used `-O3` for the PPE program and `-Os` for the SPE program. The input data has been generated by Erdős-Rényi random graph generator in GTgraph [29]. The performance in Gflop/s reported in this section is calculated by using the following formula:

$$\begin{aligned} \text{Performance [Gflop/s]} &= \frac{\text{Num. of floating point operations [flops]}}{\text{Exectution time [ns]}} \\ &= \frac{2\mathcal{N}^{\text{block}}_{(\oplus,\otimes)}(n,b)}{\text{Exec. time}} = \frac{2n(n^2-2b+1)}{\text{Exec. time}}. \end{aligned}$$

Each minimum or maximum operation is counted as a single floating-point operation; in other words, any $(\oplus, \otimes)$-operation is counted as two flops, which is a common practice [30, 31, 32].

5.2 Performance of a 64×64 matrix-matrix "multiply-add" in different semirings on one SPE

This section shows the performance of a 64×64 matrix-matrix "multiply-add" in different semirings on one SPE. Recall that the MMA in linear algebra with $(+, \times)$-algebra delivers 25.44 Gflop/s (99.38% of the peak). The performance of a 64×64 MMA for all-pairs shortest paths (APSP), critical paths (CRP), and maximum reliability paths (MRP) problem delivers 8.502 Gflop/s each (33.21% of the peak). As it was estimated in Section 4.3.1, the sustained performance is close to one-third of the peak. On the other hand, the performance of a 64×64 MMA for maximum capacity paths (MCP) and minimum spanning tree (MST) problem delivers 6.378 Gflop/s (24.91% of the peak). The sustained performance of one-fourth of the peak is also coincident with the previous estimation.

For comparison, we have also measured the performance of a 64×64 matrix-matrix "multiply-add" on PPE. PPE achieves only 1.801 Gflop/s and 1.625 Gflop/s for APSP, CRP and MRP, and for MCP and MST, respectively. PPE has a similar SIMD instruction set to SPE, and the potential peak performance of PPE is the same as the SPE; the achieved performance is, however, several times less than on SPE. The low performance is considered to be due to the less number of registers; the available 32 128-bit vector registers of PPE are not enough to keep all block data elements in the register file.

5.3 Performance on PPE and SPEs

Figure 6 shows the performance plot of the block algorithm for the different APP instances on PPE and six SPEs; the experiments were carried out for the problem sizes in multiples of 64 (blocking factor). Three instances, all-pairs shortest paths (APSP), critical paths (CRP), and maximum reliability paths (MRP) problems, have the same time of solution on the Cell/B.E. The other two instances, maximum capacity paths (MCP) and minimum spanning tree

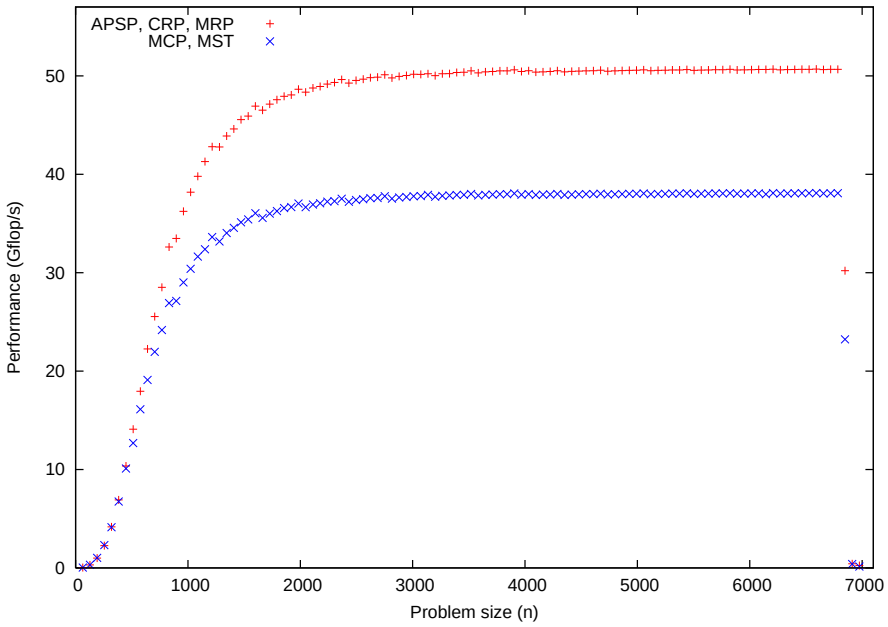


Figure 6: Performance in Gflop/s for the APP instances using PPE and 6 SPEs

(MST) problems, have also the same time of solution on the Cell/B.E.

The performance for relatively small problem size on any number of SPEs is not high. This is due to the lower performance of PPE which updates the "black" block, in addition to the synchronization overhead between PPE and SPEs. On the other hand, when matrix size n is very large (in our case, $n \geq 6912$), the performance degrades drastically due to the memory shortage that results in many page faults. In fact, the number of page faults increases to many hundred thousands for such large matrix size from almost zero for the smaller sizes[3]. 256 MB of the host memory of PS3 is known as a limiting factor for the high performance computing [33].

Table 2 shows the maximum performance using PPE and different number of SPEs. Comparing the performance of six SPEs with one SPE, the speedup ratio is around 5.98, i.e. we have almost linear speedup increasing.

We have made additional experiments to see the effects of two kinds of optimization techniques, which are multi-buffering (MBUF) and computation with a block data layout (BDL) [4]. The *multi-buffering* technique enables us to

[3]The numbers of page faults was counted by Linux `time` command.

[4]The performance shown in Fig. 6 and Table 2 are results of using both the optimization techniques.

Table 2: Maximum performance in Gflop/s using PPE and SPE(s)

Instances	1 SPE	2 SPEs	3 SPEs	4 SPEs	5 SPEs	6 SPEs
APSP, MRP, CRP	8.47	16.92	25.37	33.82	42.25	50.68
MCP, MST	6.37	12.74	19.06	25.41	31.75	38.09

Table 3: Execution time in milliseconds with and without optimizations

		Problem size				
Instatnces	Optimization	1024	2048	3072	4096	5120
APSP, MRP, CRP	-	76	531	1695	4671	7918
	BDL	62	399	1296	3097	6024
	MBUF	59	374	1213	2994	5558
	BDL + MBUF	56	356	1159	2727	5313
MCP, MST	-	89	641	2046	5454	9556
	BDL	76	510	1666	3967	7738
	MBUF	74	488	1592	3758	7312
	BDL + MBUF	71	470	1538	3625	7068

overlap the computation with the communication, so that the communication latency can be hidden. BDL was explained in Section 4.2. Table 3 shows the results of the experiments. Both of the optimizations increase the performance.

5.4 Related Work

There has been a number of researches related to block algorithms for solving the APP. The researches has been done, in particular, to accelerate the performance for solving the all-pairs shortest paths (APSP) problem because the APSP is the most popular among all APP instances.

Venkataraman et al. [9] presented a block FW algorithm. Park et al. [10, 11] implemented a recursive version of the block FW algorithm. Han et al. [30] reported an implementation of block FW algorithm with auto-tuning technique. Bondhugula et al. [34, 35] proposed an FPGA-based method for a

block FW algorithm. Matsumoto and Sedukhin [7] presented a block algorithm for APSP on the Cell/B.E., and, independently, Vinjamuri and Prasanna [36] also published a research result for APSP with a blocking approach but a different scheduling strategy. In addition, Tadonki [37] showed a ring pipelined algorithm for APSP on the Cell/B.E.

The Graphical Processing Unit (GPU) computing is a popular solution to obtain high performance, and several APSP implementations exist for Nvidia GPUs [38, 39, 40, 41, 42, 43]. Buluç et al. implemented a recursive block FW algorithm [42]; Okuyama et al. compared the recursive block FW algorithm and their iterative block algorithm, and showed the implementation of the iterative algorithm runs around 4% faster than that of the recursive algorithm in the case of relatively small problem sizes of $256 \leq n \leq 1024$. The performance of some implementations on GPUs is over 100 Gflop/s which is more than twice as high as our result on the Cell/B.E. It is because some GPUs contain hundreds of processing elements and, as the result, have higher computational power although the clock speed is lower than in Cell/B.E..

Compared with GPUs, a winning side of the Cell/B.E. is the programmability of the SPEs thanks to the unique embedded local store and the large register file. This programmability makes the performance of the computational kernel (64×64 "MMA") very close to the peak performance whereas it is impossible to obtain such high efficiency on current GPUs.

6 Conclusion

In this paper, we have extended a block Floyd-Warshall algorithm to the instances of Algebraic Path Problem (APP) for idempotent semirings and have showed an efficient algorithm implementation on the Cell/B.E. The results of performance evaluation show that parallel implementation of the block APP algorithm, which is rich in matrix-matrix "multiply-add" operations, is well suited for the Cell/B.E., and the achieved performance is near the estimated peak.

We have also shown the effectiveness of using the optimization techniques. The implementation of the block algorithm with block data layout is faster than that of row-major layout. The evaluation result also shows that overlapping

computations and data transfers by utilizing multi-buffering technique is very effective.

As future work, we will port and evaluate the high performance algorithms to multi-core CPUs and many-core GPUs, and cluster systems with a number of such processors. Alternately, our presented block APP algorithm computes only the distances for all-pairs of vertices; however, finding additionally the paths would also be important for some practical applications of the APP. Therefore, another future work includes a design of efficient methods to compute both the paths and the distances.

References

[1] Fink, E.: A Survey of Sequential and Systolic Algorithms for the Algebraic Path Problem, Technical Report cs-92-37, Department of Computer Science, University of Waterloo (1992).

[2] Rote, G.: Path problems in graphs, *Computing Supplementum*, No. 7, pp. 155–198 (1990).

[3] Rote, G.: A systolic array algorithm for the algebraic path problem (shortest paths; matrix inversion), *Computing*, Vol. 34, No. 3, pp. 191–219 (1985).

[4] Rajopadhye, S., Tadonki, C. and Risset, T.: The algebraic path problem revisited, *Proceedings of the 5th European Conference on Parallel Computing (Euro-Par 1999)*, LNCS, Vol. 1685, Springer, pp. 698–707 (1999).

[5] Sedukhin, S.: Design and Analysis of Systolic Algorithms for the Algebraic Path Problem, *Computers and Artificial Intelligence*, Vol. 11, No. 3, pp. 269–292 (1992).

[6] Litvinov, G. L., Maslov, V. P., Rodionov, A. Y. and Sobolevski, A. N.: Universal algorithms, mathematics of semirings and parallel computations, CoRR, abs/1005.1252, http://arxiv.org/abs/1005.1252v1 (2010).

[7] Matsumoto, K. and Sedukhin, S. G.: A Solution of the All-Pairs Shortest Paths Problem on the Cell Broadband Engine Processor, *IEICE Transactions on Information and Systems*, Vol. E92-D, No. 6, pp. 1225–1231 (2009).

[8] Yokoyama, S., Matsumoto, K. and Sedukhin, S. G.: Matrix Inversion on the Cell/B.E. Processor, *Proceedings of 11th IEEE International Conference on High Performance Computing and Communications (HPCC-09)*, pp. 148–153 (2009).

[9] Venkataraman, G., Sahni, S. and Mukhopadhyaya, S.: A blocked all-pairs shortest-paths algorithm, *Journal of Experimental Algorithmics*, Vol. 8, p. 2.2 (2003).

[10] Park, J.-S., Penner, M. and Prasanna, V. K.: Optimizing graph algorithms for improved cache performance, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 15, No. 9, pp. 769–782 (2004).

[11] Penner, M., Park, J.-S. and Prasanna, V. K.: Optimizing graph algorithms for improved cache performance, *Proceedings of 16th International Parallel and Distributed Processing Symposium*, Washington, DC, USA, IEEE Comput. Soc, pp. 309–318 (2002).

[12] Lehmann, D. J.: Algebraic structures for transitive closure, *Theoretical Computer Science*, Vol. 4, No. 1, pp. 59–76 (1977).

[13] Moller, F.: A survey of systolic systems for solving the algebraic path problem, Technical Report CS-85-2 2, University of Waterloo Computer Science Department (1985).

[14] Sedukhin, S. G., Miyazaki, T. and Kuroda, K.: Orbital Systolic Algorithms and Array Processors for Solution of the Algebraic Path Problem, *IEICE Transactions on Information and Systems*, Vol. E93-D, No. 3, pp. 534–541 (2010).

[15] Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems, *Journal of Automata, Languages and Combinatorics*, Vol. 7, No. 3, pp. 321–350 (2002).

[16] Lehmann, D. J.: Algebraic structures for transitive closure, *Theoretical Computer Science*, Vol. 4, No. 1, pp. 59–76 (1977).

[17] Maggs, B. M. and Plotkin, S. A.: Minimum-cost spanning tree as a path-finding problem, *Information Processing Letters*, Vol. 26, No. 6, pp. 291–293 (1988).

[18] IBM Corporation: *Cell Broadband Engine Programming Handbook, Version 1.1* (2007).

[19] IBM Redbooks: *Programming the Cell Broadband Engine Architecture: Examples and Best Practices*, Vervante, California, USA (2008).

[20] Mueller, S., Jacobi, C., Oh, H.-J., Tran, K., Cottier, S., Michael, B., Nishikawa, H., Totsuka, Y., Namatame, T., Yano, N., Machida, T. and Dhong, S.: The Vector Floating-Point Unit in a Synergistic Processor Element of a CELL Processor, *17th IEEE Symposium on Computer Arithmetic (ARITH'05)*, pp. 59–67 (2005).

[21] Prasanna, V. K., Park, N. and Hong, B.: Tiling, block data layout, and memory hierarchy performance, *IEEE Transactions on Parallel and Distributed Systems*, Vol. 14, No. 7, pp. 640–654 (2003).

[22] Kurzak, J. and Dongarra, J.: Implementation of mixed precision in solving systems of linear equations on the Cell processor, *Concurrency and Computation: Practice and Experience*, Vol. 19, No. 10, pp. 1371–1385 (2007).

[23] Kurzak, J., Alvaro, W. and Dongarra, J.: Optimizing matrix multiplication for a short-vector SIMD architecture CELL processor, *Parallel Computing*, Vol. 35, No. 3, pp. 138–150 (2009).

[24] Hackenberg, D.: Fast Matrix Multiplication on Cell (SMP) Systems, http://www.tu-dresden.de/zih/cell/matmul (2009).

[25] Saxena, V., Agrawal, P., Sabharwal, Y., Garg, V. K., Kuruvilla, V. A. and Gunnels, J. A.: Optimization of BLAS on the Cell Processor, *Proceedings of the 15th International Conference on High Performance Computing (HiPC 2008)*, LNCS, Vol. 5374, Springer, pp. 18–29 (2008).

[26] Prasanna, V. K., Park, N. and Hong, B.: Analysis of memory hierarchy performance of block data layout, *Proceedings of International Conference on Parallel Processing*, Washington, DC, USA, IEEE Comput. Soc, pp. 35–44 (2002).

[27] IBM Corporation: *SPU Assembly Language Specification, Version 1.6* (2007).

[28] Chen, T., Raghavan, R., Dale, J. N. and Iwata, E.: Cell Broadband Engine Architecture and its first implementation: A performance view, *IBM Journal of Research and Development*, Vol. 51, No. 5, pp. 559–572 (2007).

[29] Madduri, K. and Bader, D. A.: GTgraph: A suite of synthetic random graph generators, https://sdm.lbl.gov/˜kamesh/software/GTgraph.

[30] Han, S.-C., Franchetti, F. and Püschel, M.: Program generation for the all-pairs shortest path problem, *Proceedings of the 15th international conference on Parallel architectures and compilation techniques - PACT '06*, New York, ACM Press, pp. 222–232 (2006).

[31] Buluç, A., Gilbert, J. R. and Budak, C.: Gaussian Elimination Based Algorithms on the GPU, Technical Report UCSB/CS-2008-15, CS Department, University of California, Santa Barbara, USA (2008).

[32] Gaeke, B., Husbands, P., Li, X., Oliker, L., Yelick, K. and Biswas, R.: Memory-intensive benchmarks: IRAM vs. cache-based machines, *Proceedings of 16th International Parallel and Distributed Processing Symposium*, IEEE Comput. Soc, pp. 290–296 (2002).

[33] Kurzak, J., Buttari, A., Luszczek, P. and Dongarra, J.: The PlayStation 3 for High-Performance Scientific Computing, *Computing in Science & Engineering*, Vol. 10, No. 3, pp. 84–87 (2008).

[34] Bondhugula, U., Devulapalli, A., Dinan, J., Fernando, J., Wyckoff, P., Stahlberg, E. and Sadayappan, P.: Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths, *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, IEEE, pp. 152–164 (2006).

[35] Bondhugula, U., Devulapalli, A., Fernando, J., Wyckoff, P. and Sadayappan, P.: Parallel FPGA-based All-Pairs Shortest-Paths in a Directed

Graph, *Proceedings of 20th IEEE International Parallel & Distributed Processing Symposium*, IEEE (2006).

[36] Vinjamuri, S. and Prasanna, V. K.: Transitive closure on the cell broadband engine: A study on self-scheduling in a multicore processor, *Proceedings of the 2009 IEEE International Symposium on Parallel and Distributed Processing*, IEEE Computer Society, pp. 1–11 (2009).

[37] Tadonki, C.: Ring pipelined algorithm for the algebraic path problem on the CELL Broadband Engine, *1st Workshop on Applications for Multi and Many Core Architectures* (2010).

[38] Micikevicius, P.: 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)*, pp. 1359–1365 (2004).

[39] Harish, P. and Narayanan, P. J.: Accelerating large graph algorithms on the GPU using CUDA, *14th International Conference on High-Performance Computing (HiPC 2007)*, LNCS, Vol. 4873, Springer, pp. 197–208 (2007).

[40] Katz, G. J. and Kider Jr, J. T.: All-pairs shortest-paths for large graphs on the GPU, *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, Aire-la-Ville, Switzerland, Switzerland, Eurographics Association, pp. 47–55 (2008).

[41] Okuyama, T., Ino, F. and Hagihara, K.: A Task Parallel Algorithm for Computing the Costs of All-Pairs Shortest Paths on the CUDA-Compatible GPU, *Proceedings of 2008 IEEE International Symposium on Parallel and Distributed Processing with Applications*, IEEE, pp. 284–291 (2008).

[42] Buluç, A., Gilbert, J. R. and Budak, C.: Solving Path Problems on the GPU, *Parallel Computing*, No. 7000012980 (2009).

[43] Okuyama, T., Ino, F. and Hagihara, K.: Fast Blocked Floyd-Warshall Algorithm on the GPU [in Japanese], *IPSJ Transactions on Advanced Computing Systems*, Vol. 3, No. 2, pp. 57–66 (2010).