PAPER

A Solution of the All-Pairs Shortest Paths Problem on the Cell Broadband Engine Processor

Kazuya MATSUMOTO[†], ***Nonmember and*** **Stanislav G. SEDUKHIN**[†a)], ***Member***

SUMMARY The All-Pairs Shortest Paths (APSP) problem is a graph problem which can be solved by a three-nested loop program. The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core processor that offers the high single precision floating-point performance. In this paper, a solution of the APSP problem on the Cell/B.E. is presented. To maximize the performance of the Cell/B.E., a blocked algorithm for the APSP problem is used. The blocked algorithm enables reuse of data in registers and utilizes the memory hierarchy. We also describe several optimization techniques for effective implementation of the APSP problem on the Cell/B.E. The Cell/B.E. achieves the performance of 8.45 Gflop/s for the APSP problem by using one SPE and 50.6 Gflop/s by using six SPEs.

key words: *all-pairs shortest paths problem, Floyd-Warshall algorithm, Cell/B.E. processor, performance evaluation*

1. Introduction

The All-Pairs Shortest Paths (APSP) problem is to find the minimum distance between any two nodes in a given weighted graph. The problem is one of the most fundamental problems in graph theory. The applications of the problem can be found in bioinformatics, network routing, computer-aided design for integrated circuits, etc.

The APSP problem can be solved by using the Floyd-Warshall (FW) algorithm with n^3 (min, +) operations on n^2 data where n is the number of nodes in a graph. However, it is difficult to obtain high performance because the FW algorithm has strict data dependencies. Therefore, several attempts have been made previously to optimize the algorithm. A blocked (tiled) FW algorithm has been developed by G. Venkataraman et al. [21], and a recursive implementation was implemented by J.-S. Park et al. [13], [14], [17]. These blocked algorithms utilize memory hierarchy, exploit the data locality, reduce the cache misses, and eventually improve the performance.

Data vectorization can also be used to accelerate the performance of the state-of-the-art processors with single instruction multiple data (SIMD) units. S.-C. Han et al. [7] reported that a four-way floating-point vectorized implementation of the blocked FW algorithm improved the performance around 2.5–3 times on Athlon 64 and 5 times on Pentium 4 over the non-SIMD blocked implementations.

The Cell Broadband Engine (Cell/B.E.) is a heterogeneous multi-core processor consisting of a standard processor, the Power Processor Element (PPE), and eight short-vector SIMD processors, the Synergistic Processor Elements (SPEs). The Cell/B.E. has outstanding single-precision floating point computational performance [22], [23]. The PPE can directly access the main memory with load and store instructions while SPE can do this by using DMA instructions. The SPE does not have a cache, but instead, it has a private local store. The SPE can be considered as a processor which has a 3-level of memory hierarchy (register file, local store, main memory). Utilizing the memory hierarchy and the SIMD units are important to exploit the performance of the Cell/B.E.

This paper presents a solution of the APSP problem on the Cell/B.E. in Sony PlayStation3. The implemented blocked algorithm for the APSP problem is similar to the algorithms in [5], [20]. The most computational intensive part of the blocked algorithm is calculated by matrix multiplication in a corresponding algebraic semiring. This feature enables the program to utilize the power of Cell/B.E.

The rest of this paper is organized as follows. In Sect. 2, the APSP problem is defined. In Sect. 3, the blocked APSP algorithm is presented. In Sect. 4, more detailed description of the Cell/B.E. is described and several optimization techniques for effective implementation of the blocked APSP algorithm are introduced. In Sect. 5, the performance is evaluated and discussed. Finally, we draw the conclusion and the future works in Sect. 6.

2. All-Pairs Shortest Paths Problem

Let $G = (V, E, w)$ be a weighted graph with n nodes or vertices $v \in V = \{1, \cdots, n\}$, edges $(i, j) \in E \subseteq V \times V$, and the positive weight function $w : E \to \mathcal{R}_+$, where $w(i, j)$ means the distance or weight of edge (i, j) and $\mathcal{R}_+$ is the set of positive numbers. Initially, the graph G is represented by an $n \times n$ distance matrix $D = [d(i, j)]$, where

$$d(i,j) = \begin{cases} 0 & \text{if } i = j; \\ w(i,j) & \text{if } i \neq j \text{ and } (i,j) \in E; \\ \infty & \text{if } i \neq j \text{ and } (i,j) \notin E. \end{cases}$$

The shortest distance between any two vertices i and j is

$$d^*(i,j) = \min_{P \in \text{paths}(i,j)} \sum_{(u,v) \in E} w(u,v).$$

Then the problem to find an $n \times n$ matrix $D^* = [d^*(i, j)]$ is called the all-pairs shortest paths problem.

Manuscript received September 10, 2008.
Manuscript revised February 23, 2009.

[†]The authors are with the Graduate School of Computer Science and Engineering, The University of Aizu, Aizuwakamatsu-shi, 965–8580 Japan.
a) E-mail: sedukhin@u-aizu.ac.jp

DOI: 10.1587/transinf.E92.D.1225

```
for $k = 1 : n$
  for all $i = 1 : n\ (i \neq k)$
    for all $j = 1 : n\ (j \neq k)$
      $d^{(k)}(i, j) = \min(d^{(k-1)}(i, j), d^{(k-1)}(i, k) + d^{(k-1)}(k, j))$;
```

Fig. 1 Modified FW algorithm.

The Floyd-Warshall algorithm [3] can be used to solve the APSP problem with a dynamic programming approach. Let $d^{(k)}(i, j)$ be the distance of the shortest path from vertex i to vertex j composed of the subset of vertices labeled 1 to k. FW algorithm uses the following dynamic programming recurrence:

$$d^{(k)}(i, j) = \begin{cases} w(i, j) & \text{if } k = 0; \\ \min\Big(d^{(k-1)}(i, j), & \\ \qquad d^{(k-1)}(i, k) + d^{(k-1)}(k, j)\Big) & \text{if } k \geq 1. \end{cases}$$

FW algorithm computes the matrix $D^* = [d^*(i, j)] = [d^{(n)}(i, j)]$ as the final result.

Notice that because $d^{(k)}(k, k) = 0$ for any $k \in \{0, 1, \cdots, n\}$ then for $i = k$:

$$\begin{aligned} & d^{(k)}(k, j) \\ & \quad = \min\Big(d^{(k-1)}(k, j), d^{(k-1)}(k, k) + d^{(k-1)}(k, j)\Big) \\ & \quad \equiv d^{(k-1)}(k, j) \end{aligned}$$

and for $j = k$:

$$\begin{aligned} & d^{(k)}(i, k) \\ & \quad = \min\Big((d^{(k-1)}(i, k), d^{(k-1)}(i, k) + d^{(k-1)}(k, k)\Big) \\ & \quad \equiv d^{(k-1)}(i, k). \end{aligned}$$

Therefore, we can modify FW algorithm to exclude the unnecessary operations in $i = k$ and $j = k$ cases. The modified FW algorithm in a form of three-nested loop program is represented in Fig. 1. It can be easily shown that the total number of (min, +) operations in the modified FW algorithm is reduced from n^3 operations of a canonical FW algorithm [3] to

$$S^{\text{scalar}}_{(\min,+)}(n) = n^3 - 2n^2 + n = n(n-1)^2. \tag{1}$$

From Fig. 1, it is clear that the modified algorithm requires $3n(n-1)^2$ load operations and $n(n-1)^2$ store operations, so the total number of load/store operations is

$$S^{\text{scalar}}_{\text{load/store}}(n) = 4n(n-1)^2. \tag{2}$$

The FW algorithm has more strict data dependencies than in the matrix-matrix multiplication case. These dependencies should be resolved to update the entire $n \times n$ matrix $D^{(k)}$ before moving on to the next $(k + 1)$ iteration.

3. Blocked APSP Algorithm

The blocked algorithms are widely known as to be more efficient in many computing environments than the scalar algorithms because block algorithms enable reuse of data in registers and utilize the memory hierarchy. In this section, a blocked APSP algorithm is explained in details.

```
1  $N = n/b$;
2  for $K = 1 : N$ do
3    % Black block update
4    $D^{(K)}_{KK} = (D^{(K-1)}_{KK})^*$;
5    % Red blocks update
6    for all $I = 1 : N\ (I \neq K)$
7      $D^{(K)}_{IK} = \min(D^{(K-1)}_{IK}, D^{(K-1)}_{IK} + D^{(K)}_{KK})$;
8    % Blue blocks update
9    for all $J = 1 : N\ (J \neq K)$
10     $D^{(K)}_{KJ} = \min(D^{(K-1)}_{KJ}, D^{(K)}_{KK} + D^{(K-1)}_{KJ})$;
11   % White blocks update
12   for all $I = 1 : N\ (I \neq K)$
13     for all $J = 1 : N\ (J \neq K)$
14       $D^{(K)}_{IJ} = \min(D^{(K-1)}_{IJ}, D^{(K)}_{IK} + D^{(K)}_{KJ})$;
15 end
```

Fig. 2 Blocked APSP algorithm.

```
for $i = 1 : b$
  for $j = 1 : b$
    for $k = 1 : b$
      $D_{IK}(i, j) = \min(D_{IK}(i, j), D_{IK}(i, k) + D_{KK}(k, j))$;
```

Fig. 3 Red block update.

```
for $i = 1 : b$
  for $j = 1 : b$
    for $k = 1 : b$
      $D_{KJ}(i, j) = \min(D_{KJ}(i, j), D_{KK}(i, k) + D_{KJ}(k, j))$;
```

Fig. 4 Blue block update.

```
for $i = 1 : b$
  for $j = 1 : b$
    for $k = 1 : b$
      $D_{IJ}(i, j) = \min(D_{IJ}(i, j), D_{IK}(i, k) + D_{KJ}(k, j))$;
```

Fig. 5 White block update.

Figure 2 shows the blocked APSP algorithm. In this algorithm the initial $n \times n$ matrix D is divided into $N \times N$ matrix of blocks where $N = n/b$ and b is the block size. For simplicity, but without loss of generality, the matrix size is assumed to be multiples of b.

On each K-th iteration ($K = 1, 2, \cdots, N$) the blocked APSP algorithm solves, at first, the APSP problem for the *black* block D_{KK} (line 4 in Fig. 2) by applying the scalar algorithm in Fig. 1 to the $b \times b$ subproblem, then updates $N - 1$ *red* blocks D_{IK} (line 7) and $N - 1$ *blue* blocks D_{KJ} (line 10), and, finally, updates $(N - 1)^2$ *white* blocks (line 14). The important fact is that all *red*, *blue* and *white* block updates are the matrix-matrix multiply-add (like $C = A \otimes B \oplus C$) operations in the so called (min, +)-algebra (also known as the tropical semiring [18]). This fact allows the program to use all existing optimization techniques for matrix-matrix multiplication, like loop interchange, loop unrolling, software pipelining, etc. The scalar programs of the *red*, *blue*, and *white* block updates are shown in Figs. 3, 4, and 5, respectively.

As can be seen from Fig. 2, the number of matrices needed for each block update is different. The *black* block update involves only one $b \times b$ matrix. The *red*/*blue* block update needs two $b \times b$ matrices. The *white* block update requires three $b \times b$ matrices. Thus, the *black* block update requires $b(b-1)^2$ scalar (min, +)-operations and $2b^2$ load/store operations (b^2 load and b^2 store operations for each block). The all *red*/*blue* block updates require $b^3(n/b-1)$ (min, +)-operations and $3b^2 \cdot (n/b-1)$ load/store operations ($2b^2$ load and b^2 store operations for each block). The all *white* block updates require $b^3(n/b-1)^2$ (min, +)-operations and $4b^2(n/b-1)^2$ load/store operations ($3b^2$ load and b^2 store operations). Therefore, the total number of (min, +)-operations is

$$\begin{aligned} S^{\text{block}}_{(\min,+)}(n,b) &= n/b(b(b-1)^2 + 2b^3(n/b-1) \\ &\quad + b^3(n/b-1)^2) \\ &= n(n^2 - 2b + 1), \end{aligned} \tag{3}$$

and the total number of load/store operations is

$$\begin{aligned} S^{\text{block}}_{\text{load/store}}(n,b) &= n/b(2b^2 + 2 \cdot 3b^2(n/b-1) \\ &\quad + 4b^2(n/b-1)^2) \\ &= 4n^3/b - 2n^2. \end{aligned} \tag{4}$$

Equations (3) and (4) show that increasing block size b, decreases the number of (min, +) operations as well as the number of load/store operations. Notice that each (min, +)-operation consists of two scalar operations: min and +.

Comparing Eqs. (1) and (3) shows that the blocked APSP algorithm requires more (min, +)-operations than scalar algorithm, and the number of such redundant operations is

$$\Delta = S^{\text{block}}_{(\min,+)}(n,b) - S^{\text{scalar}}_{(\min,+)}(n) = 2n(n-b). \tag{5}$$

These redundant operations are performed in the *red* and *blue* block updates. On each iteration, the blocked algorithm performs b^3 (min, +)-operations, while the scalar algorithm implements $b^3 - b^2$ operations.

The total number of block operations is

$$\begin{aligned} S_{\text{block}}(n,b) &= n/b\left(1 + 2(n/b-1) + (n/b-1)^2\right) \\ &= (n/b)^3. \end{aligned}$$

On the other hand, the number of block matrix-matrix "multiply-add" operations for *red*, *blue*, and *white* block updates is

$$\begin{aligned} S_{\text{RBW}}(n,b) &= n/b\left(2(n/b-1) + (n/b-1)^2\right) \\ &= (n/b)^3 - n/b. \end{aligned}$$

Therefore, the ratio

$$\rho = \frac{S_{\text{RBW}}}{S_{\text{block}}} \times 100\ \% = \left(1 - \frac{1}{(n/b)^2}\right) \times 100\ \% \tag{6}$$

implies that, in case of a relatively large n/b, the blocked APSP algorithm spends the most computing time to update the *red*, *blue*, and *white* blocks (see Fig. 6).

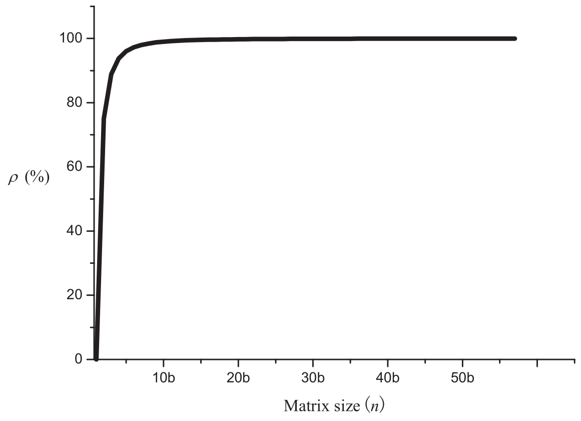

Fig. 6 Ratio $\rho = S_{\text{BRW}}/S_{\text{block}}$ in multiple of b.

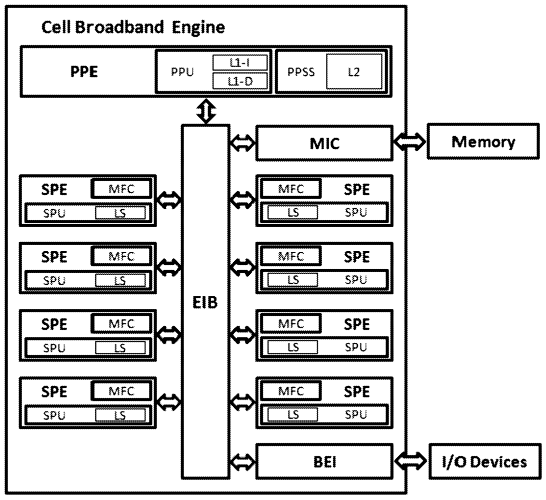

Fig. 7 Structure of the Cell Broadband Engine.

4. APSP Implementation on the Cell/B.E.

4.1 PlayStation3 and Cell Broadband Engine

PlayStation3 (PS3) is a Sony game console equipped with a 256 MB XDR DRAM as the main memory and a Cell Broadband Engine (Cell/B.E.) as the processor at 3.2 GHz clock speed. The Cell/B.E. is the first implementation of a new multiprocessor family of the Cell Broadband Engine Architecture (CBEA) [8]. The CBEA was developed jointly by the alliance of Sony, Toshiba, and IBM (STI). The Cell/B.E. is a single-chip multiprocessor with nine heterogeneous processor elements that operate on a shared, coherent memory. 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). The EIB consisting of four data rings connects all other components. The structure of the Cell/B.E. is shown in Fig. 7.

The PPE is a 64-bit general purpose RISC processor based on PowerPC architecture with vector/SIMD multimedia extensions. The PPE controls a Cell/B.E. system and runs the operating system. The PPE mainly consists of two units: the PowerPC Processor Unit (PPU) and the PowerPC Processor Storage Subsystem (PPSS). The PPU is an instruction execution unit with a 32 kB level-1 (L1) instruction cache and a 32 kB L1 data cache. The PPSS has a 512 kB level-2 unified cache and deals with memory requests from the PPU and external requests from SPEs or I/O devices.

The SPEs are 128-bit RISC processors with a dual-issue pipelined four-way SIMD unit. Each SPE is mainly composed of the Synergistic Processor Unit (SPU) and the Memory Flow Controller (MFC). The SPE does not have caches, but instead, has its 256 kB embedded local store (LS). Each SPU fetches instructions from the LS, and it loads and stores data between the LS and the register file which has 128 registers with 128-bit wide. The SPU can communicate with other processors through its DMA controller (MFC). The single-precision floating point peak performance of each SPE is 25.6 Gflop/s because each SPE can compute eight single-precision floating point operations per clock cycle with a four-way vector fused-multiply-add (`fma`) instruction. Two of eight SPEs cannot be used on the PS3 Linux because one SPE is disabled to improve chip yields and the other SPE is used by the PS3 OS.

4.2 Implementation Details

Several considerations are required to implement the blocked APSP algorithm on the Cell/B.E. effectively. Here, some details of the implementation are mentioned.

4.2.1 Block Size

The proper block size is important for utilizing blocked algorithms. The block size is decided based on several factors such as available memory size and data transfer speed. For the implemented APSP program, the blocks with 64×64 elements in a single precision are used. Several reasons exist for the selection of this block size. For each 64×64 block, 16 kB memory space is needed. At the same time, 16 kB is the maximum size of a single DMA transfer. In addition, if block data layout is used (see below), a whole block of contiguous data can be transferred by a single DMA transfer. Notice that it is a common practice to use blocks of 64×64 elements for blocked algorithms in the Cell/B.E. [6], [12].

4.2.2 Data Layout

The initial matrix data are usually stored in the main memory in a row-major layout, where all row elements are stored continuously. However, the blocked APSP algorithm deals with the matrix data as matrix of blocks, so the implementation is required to process the data as matrix of blocks.

To transfer a block data between the main memory and each LS of SPE, one possible way is to use DMA lists [9] that the Cell/B.E. offers and that gather and scatter data of the other memory storage. DMA lists are as fast as DMA transfers for contiguous data in principle, however, to transfer a block data which is aligned in row-major layout, DMA lists causes more TLB misses than DMA transfers. Thus, for transferring the same amount of data, DMA list takes more time than DMA transfer. The other possible way is to change the data layout to block data layout (BDL) [15], [16] before computing as it is depicted in Fig. 8. The BDL matches the data access pattern for block data, and, as a result, the DMA data transfer for block data becomes more efficient than using DMA lists.

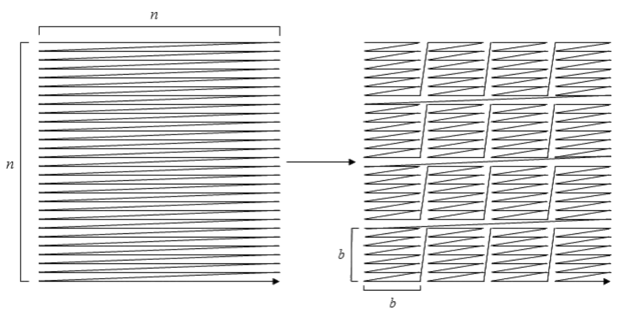

Fig. 8 Converting row-major data layout to block data layout.

4.2.3 Task Assignment

In the Cell/B.E. program, the *black* block update is done in the PPE, and the *red*, *blue*, and *white* block updates are implemented in the SPEs. This task assignment allows the program to overlap processing with all available processor elements except the first *black* block update.

4.2.4 Order of White Block Updates

The order of *white* block updates on each iteration affects the performance. On K−th iteration, the next *black* block update is for the $(K + 1, K + 1)$ block, so the PPE has to wait until SPE completes the *white* $(K + 1, K + 1)$ block update. Therefore, in the program, the $(K + 1, K + 1)$ block is updated firstly on each iteration. To accomplish this, the next updating target block is determined dynamically instead of appointing it in advance. This means that every SPE calls the function to get the block index which is not updated, then gets the block data pointed by the index from the main memory, and updates the block.

Since the data address area which keeps the index is shared with all SPEs, some mutual exclusion technique is needed to synchronize the multiple accesses to this data area. The atomic DMA updates are used in the program to implement this mutual exclusion. The atomic DMA updates ensures the mutual exclusion by managing the "reservation" of the requested address area.

4.2.5 Matrix "multiply-add" in (min, +)-Algebra

The most compute intensive part of the Cell/B.E. program is in non-*black* block updates. A fast matrix multiplication (`matmul`) program [6] was modified so that it can be used for these block updates of the APSP algorithm. The `matmul` program is an assembly language implementation which is very deeply tuned by several optimization techniques such as loop unrolling, SIMDization, software pipelining, etc. For single SPE, the `matmul` program achieves the sustained performance of 25.40 Gflop/s, or 99.22% of the peak performance. Thus, the modified program for the APSP problem can be assumed to demonstrate a good performance.

The difference of the `matmul` program and the APSP program is the type and the number of the needed instructions to obtain the desired result. In reality, the `matmul` program is based on intensive usage of a fused multiply-add instruction (the assembly instruction `fma` [10], [11]); however, the APSP program needs three instructions to implement a single (min, +) operation: an add, a comparison, and a select instruction (the assembly instructions `fa`, `fcgt`, and `selb`, respectively). This means that $c_{ij} = c_{ij} + \sum_{k=1}^{n} a_{ik} b_{kj}$ for the `matmul` program is changed into $c_{ij} = \min_{k=1}^{n}(c_{ij}, a_{ik} + a_{kj})$ for the APSP program. Moreover, the `matmul` program needs only one SIMD instruction to compute four multiply-add results, while for the APSP program, three SIMD instructions are needed, so the expected performance of the APSP program would be at least three times less than that of the `matmul` program.

4.2.6 Barrier Synchronization

To obtain a correct result in the APSP program, two barrier synchronization points are needed. The first point is after *black* block update (in Fig. 2, after line 4), and the second point is before *white* block updates (before line 11). The barrier synchronization is implemented by using a mailbox mechanism [9]. The mailbox mechanism offers a 32-bit data passing between PPE and SPEs. In both barrier synchronization points, each SPE, which is used for computing, 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.

4.2.7 Multi-Buffering

As previously mentioned, some data transfers are required to compute in each SPE. The transfer time is a bottleneck in the Cell/B.E. To decrease and hide the required time, double- or multi-buffering techniques are utilized. In the program, two buffers are reserved for the *black* block update (double buffering), and three buffers for the *red*, *blue*, and *white* block updates (triple buffering), respectively. The triple buffering allows the Cell/B.E. program to overlap loading data, computing, and storing data. For the *black* block update, the storing data is unnecessary, so two buffers are sufficient.

5. Performance Evaluation

The performance evaluation was made on PS3 with Fedora 7 Linux and Cell SDK 3.0. The `gcc` version 4.1.1 based compilers (`ppu-gcc` and `spu-gcc`) were used for the program compilation with the optimization options `-O3` for the PPE program and `-Os` for the SPE program. To evaluate performance, the number in billions of (single precision) floating-point operations per second (Gflop/s) is used. The Gflop/s is calculated by using the following formula:

$$\begin{aligned}\text{Gflop/s} &= \frac{\text{Num. of scalar operations}}{\text{Execution time [sec]}}\\ &= \frac{2 \cdot S^{\text{block}}_{(\min,+)}(n,b)}{\text{Exec. time [sec]}} = \frac{2n(n^2 - 2b + 1)}{\text{Exec. time [sec]}}.\end{aligned}$$

Figure 9 shows the performance of the blocked APSP algorithm, using PPE and one to six SPEs, when the initial matrix data are aligned in block data layout. The experiments were carried out for the problem sizes ranging from $n = 64$ to $n = 7360$ in multiples of 64 (block size).

The maximum performance is 8.45 Gflop/s in the case of using PPE and one SPE. The maximum aggregate performance of using PPE and six SPEs is 50.6 Gflop/s, which is 5.99 times faster than that of using PPE and one SPE. The performance results show that every SPE achieves one-third of the peak performance (25.6 Gflop/s) which is in full agreement with our previous estimation.

The performance for the relatively small matrices on any number of SPEs is not so different because the APSP program spends most of the time in the *black* block update on the PPE. On the other hand, when matrix size is larger than 7104, the performance degrades drastically due to a memory shortage that results in many page faults. For the standard page size of 4 kB, the number of page faults is increased from almost zero to many hundred thousands when the matrix size exceeds 7104. The number of page faults was checked by using the Linux command `/usr/bin/time`.

The performance of the blocked APSP algorithm with row-major data layout was also measured and the performance deterioration compared with BDL was within 1% for

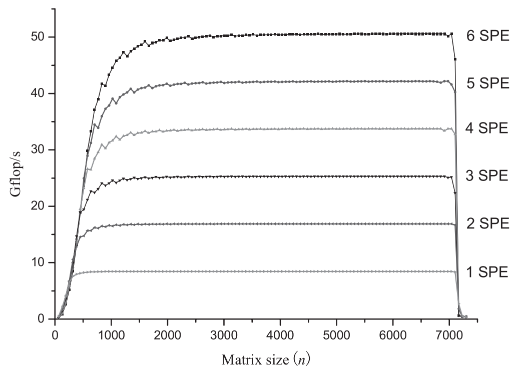

Fig. 9 Performance of the blocked APSP algorithm with BDL.

Table 1 Implementation details.

Processor	Architecture	Algorithm	Source
Cell/B.E.	3.2 GHz 9-core processor (PS3: PPE, 6 SPEs)	blocked with BDL, multi-buffering 4-way SIMDization	[this paper]
Cray XD1/FPGA	0.17 GHz APSP specific FPGA with 32 PEs	blocked with BDL, $n = 4096$	[1]
Pentium 4	3.6 GHz processor	blocked with BDL, 4-way SIMDization, automatic tuning	[7]
Athlon 64	2.4 GHz processor	blocked with BDL, 4-way SIMDization, automatic tuning	[7]
Imagine	0.4 GHz programmable stream processor with 48 parallel ALUs	blocked with BDL	[5]
VIRAM	0.2 GHz processor with 4 lanes	Transitive closure of a directed graph	[4]
Power3	0.375 GHz processor	blocked with BDL	[5]

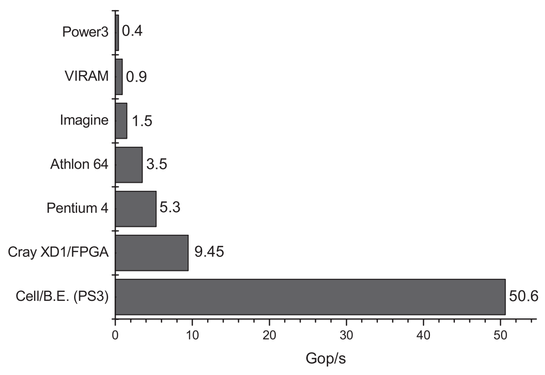

Fig. 10 Performance comparison of the APSP algorithm.

almost all matrix sizes and the number of SPEs. This result indicates that data layout does not strongly affect the performance of the blocked APSP program on the Cell/B.E. of PS3.

We have also discovered that, depending on the number of SPEs, multi-buffering improves the sustained performance from 9% (one SPE) to 13% (six SPEs).

Figure 10 shows the performance comparison of the APSP algorithm implemented on PS3 with other known implementations on different computers: Cray XD1/FPGA, Pentium 4, Athlon 64, Imagine, VIRAM, and Power3. The architectural and algorithmic details of each implementation are summarized in Table 1. The Gop/s is used as the unified performance unit instead of Gflop/s because performance of some computers was measured for integer arithmetic. Notice that the performance of 50.6 Gflop/s obtained on the Cell/B.E. is impressive, achieving roughly 5.4, 9.6, and 14.6 times faster than the Cray XD1/FPGA, Pentium 4, Athlon 64, respectively.

6. Conclusions

This paper has described the blocked APSP algorithm and its implementation on the Cell/B.E processor. It was shown that the Cell/B.E. demonstrates extremely high performance for the APSP problem. The sustained performance of 50.6 Gflop/s on the Cell/B.E. was achieved. To get even more performance, a parallel computing using cluster of PS3s is considered as future research.

In fact, the APSP problem is one of the instances of the so called Algebraic Path Problem (APP) which unifies a number of matrix and graph problems like matrix inversion, APSP, transitive closure, minimum-cost spanning tree, maximum capacity paths, etc. [2], [19], [20]. Thus, the developed APSP program can be used for the solution of these problems with small modifications to satisfy different algebras.

Acknowledgment

The authors would like to thank the anonymous reviewers for their valuable comments and helpful suggestions which have improved this paper.

References

[1] U. Bondhugula, A. Devulapalli, J. Fernando, P. Wyckoff, E. Stahlberg, and P. Sadayappan, “Hardware/software integration for FPGA-based all-pairs shortest-paths,” Proc. 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp.152–164, April 2006.
[2] E. Fink, “A survey of sequential and systolic algorithms for the algebraic path problem,” Tech. Rep., Department of Computer Science, University of Waterloo, 1992.
[3] R.W. Floyd, “Algorithm 97: Shortest path,” Commun. ACM, vol.5, no.6, p.345, 1962.
[4] B.R. Gaeke, P. Husbands, X.S. Li, L. Oliker, K.A. Yelick, and R. Biswas, “Memory-intensive benchmarks: IRAM vs. cache-based machines,” Int'l Parallel and Distributed Processing Symposium, pp.30–36, April 2002.
[5] G. Griem and L. Oliker, “Transitive closure on the imagine stream processor,” Proc. Fifth Workshop on Media and Stream Processors (MSP-5), 2003.
[6] D. Hackenberg, “Fast matrix multiplication on Cell (SMP) systems,” Technische Universität Dresden; http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/matmul/, Feb. 2008.
[7] S.-C. Han, F. Franchetti, and M. Püschel, “Program generation for the all-pairs shortest path problem,” PACT '06: Proc. 15th International Conference on Parallel Architectures and Compilation Techniques, pp.222–232, ACM, New York, NY, USA, 2006.
[8] IBM, Cell Broadband Engine Architecture, Version 1.01, Oct. 2006.
[9] IBM, Cell Broadband Engine Programming Handbook, Version 1.1, April 2007.
[10] IBM, SPU Assembly Language Specification, Version 1.6, Sept. 2007.
[11] IBM, SPU C/C++ Language Extensions, Version 2.5, Sept. 2007.

[12] J. Kurzak and J. Dongarra, "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.
[13] J.-S. Park, M. Penner, and V.K. Prasanna, "Optimizing graph algorithms for improved cache performance," IPDPS '02: Proc. 16th International Symposium on Parallel and Distributed Processing, IEEE Computer Society, pp.32–41, Washington, DC, USA, 2002.
[14] J.-S. Park, M. Penner, and V.K. Prasanna, "Optimizing graph algorithms for improved cache performance," IEEE Trans. Parallel Distrib. Syst., vol.15, no.9, pp.769–782, 2004.
[15] N. Park, B. Hong, and V.K. Prasanna, "Analysis of memory hierarchy performance of block data layout," ICPP '02: Proc. 2002 International Conference on Parallel Processing (ICPP'02), IEEE Computer Society, p.35, Washington, DC, USA, 2002.
[16] N. Park, B. Hong, and V.K. Prasanna, "Tiling, block data layout, and memory hierarchy performance," IEEE Trans. Parallel Distrib. Syst., vol.14, no.7, pp.640–654, 2003.
[17] M. Penner and V.K. Prasanna, "Cache-friendly implementations of transitive closure," J. Experimental Algorithmics, vol.11, no.1.3, 2006.
[18] J.-E. Pin, Tropical Semirings, In Idempotency, Publ. of the Newton Inst. 11, pp.50–69, Cambridge Univ. Press, 1998.
[19] G. Rote, "Path problems in graphs," Computing Supplementum 7, pp.155–198, Springer, 1990.
[20] A. Takahashi and S.G. Sedukhin, "Parallel blocked algorithm for solving the algebraic path problem on a matrix processor," LNCS, vol.3726, pp.786–795, 2005.
[21] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, "A blocked all-pairs shortest-paths algorithm," J. Experimental Algorithmics, vol.8, no.2.2, 2003.
[22] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the Cell processor for scientific computing," CF '06: Proc. 3rd Conference on Computing Frontiers, pp.9–20, ACM, New York, NY, USA, 2006.
[23] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "Scientific computing kernels on the Cell processor," Int. J. Parallel Program., vol.35, no.3, pp.263–298, 2007.

Kazuya Matsumoto received Bachelor degree in Computer Science and Engineering from the University of Aizu, Japan in 2008. Currently, he is a master student at the University of Aizu. His current research interests include parallel and distributed computing, program optimization and tuning, design and evaluation of parallel algorithms.

Stanislav G. Sedukhin is a professor and head of the Computer Engineering Division at the University of Aizu. He received his Ph.D. in Computer Science and Dr.Sci. (Physics and Mathematics) from the Russian Academy of Sciences in 1982 and 1993, respectively. His research interests are in parallel and distributed computing, architectural synthesis of VLSI-oriented processors, and design of highly-parallel algorithms and application-specific array processors. Dr. Sedukhin is a member of ACM, IEEE Computer Society.