Ring pipelined algorithm for the algebraic path problem on the CELL Broadband Engine*

Claude Tadonki
Laboratoire de l'Accélérateur Linéaire/IN2P3/CNRS
University of Orsay, Faculty of Sciences, Bat. 200
91898 Orsay Cedex (France)
claude.tadonki@u-psud.fr

Abstract

The algebraic path problem *(APP) unifies a number of related combinatorial or numerical problems into one that can be resolved by a generic algorithmic schema. In this paper, we propose a linear SPMD model based on the* Warshall-Floyd *procedure coupled with a systematic shift-toroïdal. Our scheduling requires a number of processors that equals the size of the input matrix. With a fewer number of processors, we exploit the modularity revealed by our linear array to achieve the task using a* locally parallel and globally sequential *(LPGS) partitioning. Whatever the case, we just need each processor to have a local memory large enough to house one (probably block) column of the matrix. Considering these two characteristics clearly justify an implementation on the CELL Broadband engine, because of the efficient SPE to SPE communication bandwidth and the absolute power of each SPE. We report our experimentations on a QS22 CELL blade on various input configurations and exhibit the efficiency and scalability of our implementation. We show that, with a highly optimized Warshall-Floyd kernel, we could get close to 80 GFLOPS in simple precision with 8 SPEs (i.e. 80% of the peak performance for the APP).*

1 Introduction

The *algebraic path problem* (APP) unifies a number of related problems (transitive closure, shortest paths, Gauss-Jordan elimination, to name few.) into a generic formulation. The problem itself has been extensively studied at the mathematic, algorithmic, and programming point of view on various technical contexts. Among existing algorithms, the dynamical programming procedure proposed by Floyd [3] for the *shortest paths* and Warshall [15] for the *transitive closure* sor far remains on the spotlight. Due to the wide range of (potential) applications for this problem, sometimes and ingredient to solve other combinatorial problems, providing an efficient algorithm or program to solve the APP is crucial.

In this paper, we consider an implementation on the CELL [10]. The CELL processor has proved to be quite efficient compare to traditional processors when it comes to regular computation. For a more general use, an important programming effort is required in order to achieve the expected performance. A part from providing highly optimized SPU kernels, it is also important to derive a global scheduling in which all participating SPEs efficiently cooperate in order to achieve the global task (managed from the PPU). Most of existing codes for the CELL are based on a master slaves model, where the SPEs get the data from the PPU, perform the computation and send the result back to the main memory through direct memory accesses (DMAs). Such models suffer from lack of scalability, especially on memory intensive applications. Our solution for the APP is based on a linear SPMD algorithm, with quiet interesting properties like *local control*, *global modularity*, *fault-tolerance*, and *work optimal complexity*.

Some attempts on implementing the transitive closure on the CELL can be found in the literature. Among them, we point out the works described in [8] (up to 50 GFLOPS) and [11] (up to 78 GFLOPS in perspective). The two solutions are both based on a block partitioning of the basic Warshall-Floyd together with had-hoc memory optimization and efficient global synchronization. In such *master-slaves* models where all the SPEs compute the same step of the *Warshall-Floyd* procedure at a time, a special care is required for *data alignment* in addition to redundant data management (the pivot elements). Since memory is a critical resource for the SPE, we think it is important to come with a solution which is algorithmically robust regarding both memory re-

*Work done under the PetaQCD project, supported by the french research agency ANR.

quirement and data transfers. We consider this work as one answer to those key points in addition to the absolute performance for which we are also competitive (potential of 80 GFLOPS).

The rest of the paper is organized as follows. The next section provides an overview of the CELL Broadband Engine and its main characteristics. This is followed by a fundamental description of the APP, together with the *Warshall-Floyd* algorithm and the variant of our concern. In section 4, we present our scheduling and discuss various key points. This is followed in section 5 by our implementation strategy on the CELL and related technical issues. We show and comment our experimental results in section 6. Section 7 concludes the papers.

2 Overview of the CELL Broadband Engine

The CELL[1, 10] is a multi-core chip that includes nine processing elements. One core, the *POWER Processing Element* (PPE), is a 64-bit Power Architecture. The remaining eight cores, the *Synergistic Processing Elements* (SPEs), are Single Instruction Multiple Data (SIMD) engines (3.2GHz) with 128-bit vector registers and 256 KB of local memory, referred to as local store (LS). Figure fig. 1 provides a synthetic view of the CELL architecture.

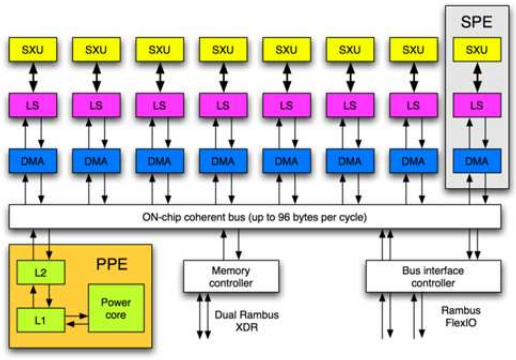

Figure 1. DMA scheme from the SPU

Programming the CELL is mainly a mixture of single instruction multiple data parallelism, instruction level parallelism and thread-level parallelism. The chip was primarily intended for digital image/video processing, but was immediately considered for general purpose scientific programming (see [17] for an exhaustive report on the potential of the CELL BE for several key scientific computing kernels). A specific consideration for QR factorization is presented in [6]. Nevertheless, exploiting the capabilities of the CELL in a standard programming context is really challenging. The programmer has to deal with constraints and performance issues like *data alignment, local store size, double precision penalty, different level of parallelism*. Efficient implementations on the CELL commonly a conjunction of a good computation/DMA overlap and a heavy use of the SPU intrinsics.

3 The algebraic path problem

3.1 Formulation

The *algebraic path problem*(APP) may be stated as follows. We are given a weighted graph $G = \langle V, E, w\rangle$ with vertices $V = \{1, 2, \cdots, n\}$, edges $E \subseteq V \times V$ and a weight function $w : E \to S$, where S is a *closed* semiring $\langle S, \oplus, \otimes, *, \mathbf{0}, \mathbf{1}\rangle$ (*closed* in the sense that $*$ is a unary "closure" operator, defined as the infinite sum $x* = x \oplus (x \oplus x) \oplus (x \oplus x \oplus x) \oplus \cdots$). A *path* in G is a (possibly infinite) sequence of nodes $p = v_1 \cdots v_k$, and the *weight* of a path is defined as the product $w(p) = w(v_1, v_2) \otimes w(v_2, v_3) \otimes \cdots \otimes w(v_{k-1}, v_k)$. Let $P(i, j)$ denotes the (possibly infinite) set of all paths from i to j. The APP is the problem of computing, for all pairs (i, j), such that $0 < i, j \leq n$, the value $d(i, j)$ defined as follows

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

For the *transitive closure*, M is the incidence boolean matrix and $\{\oplus, \otimes\} = \{\vee, \wedge\}$. For the *shortest path*, M is the cost matrix and $\{\oplus, \otimes\} = \{\min, +\}$. In any case, $\oplus$ and $\otimes$ are *commutative* and *associative*. Moreover, $\otimes$ is distributive over $\oplus$. These three properties are very important as they allow to safely permute and factorize the computations as desired.

3.2 Warshall-Floyd algorithm

We reconsider the previous formulation and context. Let $P^{(k)}(i, j)$ be the set of all paths from i to j of length k. Thus, $P(i, j)$ is the union of all $P^{(k)}(i, j)$, $k \geq 0$. This yields equation (2)

$$\bigoplus_{p \in P(i,j)} w(p) = \bigoplus_{k \geq 0} \Big(\bigoplus_{p \in P^{(k)}(i,j)} w(p) \Big) \tag{2}$$

From (1) and (2), we can write

$$d(i,j) = \bigoplus_{k \geq 0} d^{(k)}(i,j), \tag{3}$$

where

$$d^{(k)}(i,j) = \bigoplus_{p \in P^{(k)}(i,j)} w(p). \tag{4}$$

If M is the *incidence or weight* matrix of a finite graph G of order n, then $M^{(k)}$ denotes the matrix of $d^{(k)}(i, j)$, and

M^* the closure matrix (the one we want to compute). By extending the operator $\oplus$ to matrices, we obtain

$$M^* = M^{(0)} \oplus M^{(1)} \oplus M^{(2)} \oplus \cdots \oplus M^{(n)}, \qquad (5)$$

where $M^{(0)} = M$.

The *Warshall-Floyd* dynamical programming procedure to solve the APP formulation is inspired from equation (5), where $P^{(k)}(i,j)$ is now the set of all paths from i to j crossing the nodes $\{1, 2, \cdots, k\}$. Thus, $m^{(k)}(i,j)$ can be computed from $m^{(k-1)}(i,j)$ by considering node k as follows

$$m_{ij}^{(k)} = m_{ij} \oplus (m_{ik}^{(k-1)} \otimes m_{kj}^{(k-1)}). \qquad (6)$$

An important property of this algorithm, which turns to be a memory advantage, is that all the $M^{(k)}$ can be housed inside the same matrix M. So, we perform n *in-place* (matrix) updates within the input matrix M and end with the closure matrix M^*. At step k, row k (resp. column k) is called *pivot row* (resp. *pivot row*). They remain constant at step k and are used to upgrade $M^{(k-1)}$ to $M^{(k)}$. Figure 2 depicts a snapshot of the dataflow of the algorithm, which is of $O(n^3)$ complexity. A part from that $O(n^3)$ complexity, which refers to floating point operations, it is important to notice that the move of the pivot row and the pivot column, although quite regular compare to *gaussian pivoting*, need a special attention. There are mainly two impacts. The first is on the memory access pattern, which sustains a shift from one step to the next one. The second one is on the pipeline scheduling, the pivot elements have to be ready before starting a given step. In order to get ride of the difference between Warshall-Floyd steps, we now consider a shift-toroïdal transformation proposed by Kung, Lo, and Lewis [9].

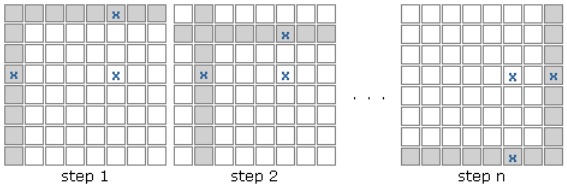

Figure 2. Warshall-Floyd procedure

3.3 Kung-Lo-Lewis mapping

The idea is to maintain the pivots at the same place, preferably at the origins. To do so, Kung, Lo, and Lewis have suggested a shift-toroïdal of the matrix after each application of the standard Warshall-Floyd procedure. Technically, it is equivalent to say that after each step, the nodes are renumbered so that node i becomes node $i-1$ (or $(i-1) \bmod n+1$ to be precise). Thereby, the matrices $M^{(k)}$ are completely identical, with the pivot row (resp. pivot column) remaining the first row (resp. first column). There are two ways to handle such a reindexation. The first one is to explicitly shift the matrix after the standard Warshall-Floyd procedure. The second one is perform the shift-toroïdal on the fly, means after each update of the matrix entries. Figure 3 depicts the two possibilities.

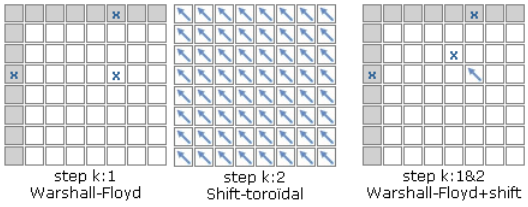

Figure 3. Toroïdal shift

In a formal point of view, we now apply the following rule

$$m_{i-1,j-1}^{(k)} = m_{ij} \oplus (m_{ik}^{(k-1)} \otimes m_{kj}^{(k-1)}), \qquad (7)$$

where operations on subscripts are performed modulo n. When implementing the algorithm in this way, one needs to keep the pivot row and/or the pivot column (its depends on the scheduling) as they could be overwritten due to the shift. In a parallel context, where the data move between the computing units, the memory operations that implement the shift become virtual as they are wisely performed during the transfers. We take all these into account to derive our linear pipeline scheduling.

4 Description of our algorithm

4.1 Scheduling

Given a graph of order n, our scheduling can be intuitively described as follows. The computation of $M^{(k)}$, assigned a single processor, is performed row by row, from the first row (the pivot) to the last one. Each row is computed from the first point (the pivot) to the last one. Figure 4 displays an overview of our array.

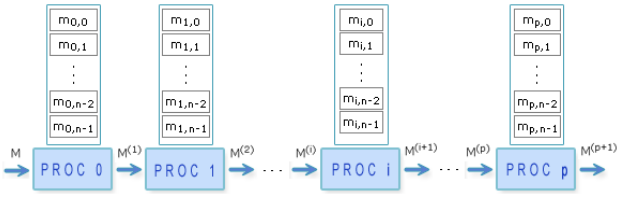

Figure 4. Linear SPMD array for the APP

If (i,j,k) refers to the (i,j) entry of $M^{(k)}$, then our scheduling can be expressed by the timing function t and

the allocation function a given by

$$t(i,j,k) = L(k,n) + (i \times n + j) + 1 \quad (8)$$
$$a(i,j,k) = k \quad (9)$$

where $L(k,n)$ is the logical *computation latency* due to the graph dependencies and our row projected allocation. At this point, we need n processors cooperating on a linear basis. Each processor operates as follows:

- compute the first row (the pivot) and keep on the local memory
- compute + send each of the remaining rows
- send the pivot row

Computing the pivot row requires n steps, which count for the computation latency as any value is sent out during that time. In addition, because of the rotation, a given processor computes a row in the order $0, 1, 2, \cdots, n-1$ and outputs the results in the order $1, 2, \cdots, n-1, 0$. Thus, the total latency between two consecutive processor is $(n+1)$, and we naturally obtain

$$L(k,n) = (k-1)(n+1), k \geq 1. \quad (10)$$

Thus, processor k start its computation at step $L(k,n) = k(n+1)$ and ends n^2 steps after (i.e. at step $n^2 + k(n+1)$). It is important to keep these two values in mind as they will be locally used by each processor to asynchronously distinguish between computing phases and acts accordingly. Our solution originally needs n processors, which is a strong requirement in practice. Fortunately, the conceptual modularity of our scheduling allow to overcome the problem as we now describe.

4.2 Modularity

Recall that processor k computes $M^{(k)}$ and communicates with processors $k-1$ and processor $k+1$. If we have p processors, $p < n$, then we adapt our schedule by just requesting processor k to computes $M^{(k+\alpha p)}$, for all integers α such that $k + \alpha p \leq n$. This is naturally achieved by performing several rounds (n/p to be precise) over our linear array of p processors. This corresponds to the so-called *locally parallel and globally sequential* (LPGS). The fact that our steps are completely identical makes it really natural to implement. Moreover, there is no additional memory need. Indeed, the capability of performing all updates within the same matrix is still valid, processor 0 continuously reads in A and processor $p-1$ continuously writes in A (there will be no conflict as the involved locations will always be different).

From figure **??**, we see that the remaining part of $M^{(\alpha p)}$ and the yet computed part of $M^{((\alpha+1)p)}$ reside on the same matrix space into the main memory. Moreover, the idempotent property of the APP (i.e. $M^{(n+k)} = M^{(n)} = M^*$, $\forall k \geq 0$) provides another simplicity. Indeed, if p does not divides n, then a strict application of our partitioning will ends with $M^{(m)}$, where $m = \lceil (n/p) \rceil \times p$ is greater that n. We will still get the correct result, but with an additional $p - (n \bmod p)$ steps. If we do not want this additional unnecessary computation, we could just dynamically set processor $n \bmod p$ to be the last processor at the ultimate round.

Because of the data communication latency, it is usually better to perform block transfers instead of atomic ones. From a starting fine-grained scheduling, this is achieved by tiling. Although we plan to implement our algorithm at row level, means we compute/send rows instead of single entries, which is already a kind of tiling, we still need to do more by considering a more general tiling for the APP.

4.3 Tiling

On parallel and/or accelerated computing systems, because the communication latency is likely to dominate, the cost of communicating a single data element is only marginally different from that for a "packet" of data. Therefore, it is necessary to combine the results of a number of elementary computations, and send these results together. This is typically achieved by a technique called *supernode partitioning* [5] (also called *iteration space tiling*, *loop blocking*, or *clustering*), where the computation domain is partitioned into (typically parallelepiped shaped) identical "blocks" and the blocks are treated as atomic computations. Such a clustering, which can be applied at various algorithmic level, has also proved to be a good way for a systematic transition from a fine grained parallelism to a coarse grained parallelism [14, 2].

Although tiling [18] is a well known strategy, even used on uniprocessors to exploit hierarchical memories [13], most compilers are only able of tiling rather simple programs (perfectly nested loops with *uniform* dependences). The *Warshall-Floyd* algorithm does not belong to this class. Thus, producing a tiled version in this case is likely to be a manual task.

Considering the original APP formulation and the *Warshall-Floyd* algorithm as previously described, a tile version can be derived by extending the atomic operations $\oplus$ and $\otimes$ to operate on blocks. Now, each step is either the resolution of the APP on a $b \times b$ subgraph, or a "multiply-accumulate" operation $A \oplus (B \otimes C)$, where the operands are $b \times b$ matrices and the operations are the matricial extension of the semiring operators. The only structural change imposed by a block version is that the row pivot (resp. column pivot) need to be explicitly updated too (they do not remain constant as in the atomic formulation). An impor-

tant question raised by the tile implementation of our linear algorithm is the optimal tile size. Indeed, tiling yield a benefit from communication latency, but at the price of the global computation latency (i.e. $p \times n$, which now becomes $p \times (bn)$). We now discuss all these points.

4.4 Performance analysis and prediction

From the formal expression of our scheduling (see (4)), we derive the following results.

Theorem 4.1. *Our algorithm solves the APP of size n in $2n^2 - 1$ steps using n processors.*

Proof. The last result is produced at time $t(n-1, n-1, n) = (n-1)(n+1)+(n-1)n+(n-1)+1 = 2n^2-1$. □

Theorem 4.2. *Using p processors ($p < n$), our algorithm solves the APP of size n in a number of steps given by*

$$T(n,p) = (p-1)(n+1) + \frac{n^3}{p}. \quad (11)$$

Proof. Since the last processor is $p-1$, the global latency is $L(p-1, n) = (p-1)(n+1)$. In addition, there is no idle time in our round-robin partitioning, and the n^3 computations are equally shared among the p processors, thus the additional n^3/p steps. □

Theorem 4.3. *The time to compute the closure matrix of order n and tile b with p processor is given by*

$$T(n,b,p) = [(p-1)(\frac{n}{b}+1)+\frac{(\frac{n}{b})^3}{p}](\alpha b^3+\tau b^2+\beta), \quad (12)$$

where τ is the time for one (scalar) semiring operation, α the transfer bandwidth, and β the communication bandwidth.

Proof. We have simply considered the formula (11) with n replaced by $\frac{n}{b}$ in (11), together with the fact that each operation with a block involves b^3 floating point operations and a communication routine. □

Let now discuss some implementation considerations. First, our "fine grained" operates on rows, this means that the compute/send operation are applied on the rows basis. Tiling is further considered, thus operating on a packet of rows. To be precise, with a tile size b, we manipulate b rows at each step, and the processors operates $b \times b$ blocks with the packet of b consecutive rows. Our algorithm thus uses a mixture of LPGS (round-robin partitioning) and LSGP (tiling). In the timing provides by formula (12), we fully consider the communication overhead. However, in a perfect implementation, the time due to data communication is hidden by that of floating point computation. This is particularly expected with the APP because of the *cubic* floating point complexity compared to the *quadratic* volume of the corresponding transfer. Our algorithm really allows such a perfect overlap because the communications are local and are performed asynchronously from on processor to its successor. With the CELL, this is well implemented by asynchronous DMAs as we now exhibit.

5 Implementation on the CELL BE

5.1 Overview

Among the reasons why the CELL suits for implementing our algorithm, we may point out the followings:

- the local store and its moderate size. Our algorithm just requires a processor to be able to store 2 (block) rows of the matrix. Thus, the modest size of the local store does appear as a constraint. Moreover, its faster and uniform access is an advantage for local performance and global regularity

- our algorithm requires each processor to asynchronously communicate with its forward (send/put) and backward (receive/get) neighbor. This is well implemented with SPE to SPE communication routines. Indeed, a given SPE can write to the local store of the others, thus in the LS of its forward neighbor.

- our algorithm operates with a fewer number of processors. The CELL has up to 16 SPEs. Again, this does not act as an implementation constraint. Moreover, operating with smaller number of processors reduces the global computation latency, thus improves the efficiency of the program (relative to the number of processors).

- DMA is fast and can be performed asynchronously with the computations. Moreover, each SPE can access the main memory, thus the first processor will get the input data from the main memory while the last (reset dynamically if necessary) write outputs results on the main memory.

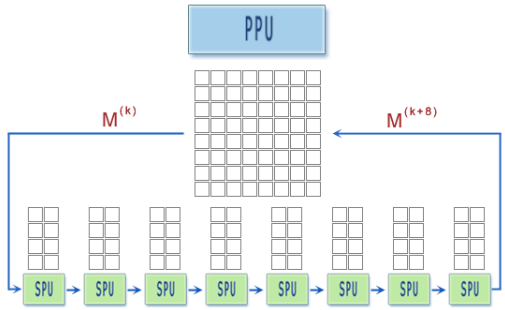

Figure 5. Round-robbin implementation of the APP on the CELL

As usual, the PPE is mainly devoted to mastering the whole computation. The generic SPE structure is the following:

```
struct spe_arg_t {
  unsigned int spe_rank;
  float *A;
  unsigned int n;
  unsigned int base_backward;
  unsigned int base_forward;
  unsigned int last_spe_id;
  unsigned int nb_round;
  unsigned int tile_size;
} __attribute__((aligned(16)));
```

After the number of participating SPE has been specified or retrieved with a call to `spe_cpu_info_get(SPE_COUNT_USABLE_SPES, -1)`, the PPE fills up the structure with appropriate values, prepares and launches SPE threads. The fields `base_backward` and `base_forward` are used for SPE to SPE communication (data and synchronization) following our algorithm. The number of rounds is given by $(n/tile)/p$, where p is the number of SPEs. SPE 0 (resp. SPE `last_spe_id` reads (resp. writes) data from (resp. to) the main memory. There is no synchronization between the PPU and the SPEs. We now discuss individual key points.

5.2 SPE computation

The main floating point computation kernel needed at the SPE level are the APP on a $b \times b$ subgraph (tile), or a "multiply-accumulate" operation $A \oplus (B \otimes C)$ on tiles. The corresponding SIMD code looks like the one displays in figure 6. Indeed, the code could be significantly optimized, this is not the scope of this work but will be done at the soonest.

```
void block_pivot_warshall(
float **pivot,unsigned short b){
vector float w;
vector float *u, *v;
float e;
unsigned int i,j,k;
for(k=0;k<b;k++){
 u = (vector float *)(pivot[k][0]);
 for(i=0;i<b;i++){
   v = (vector float *)(pivot[i][0]);
   e = pivot[i][k];
   w = (vector float){e,e,e,e};
   for(j = 0 ; j < b/4 ; j++)
     v[j] = fminf4(v[j],
                   spu_add(u[j],w));
 }
}
```

Figure 6. SIMD code for the $b \times b$ APP

There are mainly three phases. The first one is the latency phase, where the SPE has not yet received any data. The second phase starts with the reception of the pivot row, which is then updated an stored into a buffer in the local store. In the third phase, the last one, the SPE is also inactive and is just waiting the other ones to finish their remaining computations. Obviously, the overhead of the first and the last phases increases with either a longer array or a bigger tile size. This is the point for the pipeline compromise. Each SPE asynchronously detects the phases using a timer variable (local counter incremented after each step). The program uses the formula of the latency (i.e. $2k$ for processor k), and that of the last step (i.e. $n + 2k + 1$ for processor k), recall we compute/send a (block) row at a step . Thus, the SPE performs an internal control during all its computations. That is why there is no need of any kind of synchronization with or via the PPU.

5.3 Data communication

At a step, the SPE updates a row and puts it (DMA) into the corresponding buffer of its forward neighbor SPE and then waits for its completion. Since its backward SPE neighbor has done the same, the SPE then receives its data for the next step without any additional delay. Thus, all our communications (DMA) are local and fully parallel. For SPE 0, we apply an explicit double buffering as it receives its data from the main memory (DMA get). Although this sounds like a perfect DMA/computation overlap, we think that there is room for improvement, mainly on the SPE to SPE communication. Indeed, the fact that the SPE waits for the completion of its DMA put could be avoided by another double buffering. One way to achieve this is to start with

tiles of size $2b$ and then switch to tiles of size b after each SPE has received its pivot row. This will be studied later.

5.4 Synchronization issue

One way to synchronize the SPE is to use mailbox mechanism with the PPU. Such a central synchronization is sufficient, but could be costly as it would be heavily used for our step by step synchronizations. In addition, this has also a hindering impact on scaling. Instead, we chose to implement a local synchronization, thus strictly following the philosophy of our algorithm. To do so, each SPE has two flag variables, `flag_backward` and `flag_forward`. At the end of a given step,

- SPE k writes `counter+1` into the `flag_backward` of SPE $k+1$ to indicate he has finished and has put the data on its local store for further processing.
- SPE k writes `counter+1` into the `flag_forward` of SPE $k-1$ to indicate he has finished and is ready to receive new data on its working buffer.
- SPE k pools its variables `flag_backward` and `flag_forward` to check if they have been updated (if their value are still equal to `counter`).

In order to avoid a deadlock, when a given SPE has performed its latest computation, it performs an additional forward signaling for the next SPE to perform its last step independently.

We now turn to experimental results. The goal is to validate our algorithm and discussions about *latency, tiling* and *scalability*.

6 Performance results

All our experimentations are performed on a QS22 CELL Blade and single precision computation. First, we need to see how tiling affects the performance of our program. In table 1, we use our algorithm to solve the APP with matrices of size 128×128, 256×256, and 512×512 respectively (times are in seconds).

Tile	128×128	256×256	512×512
1	0.0216	0.1321	0.8921
4	0.0110	0.0823	0.6371
8	0.0104	0.0782	0.6082
12	0.0092	0.0754	0.5839
16	0.0105	0.0781	0.6017
20	0.0095	0.0696	0.5757
24	0.0098	0.0704	0.5872
28	0.0088	0.0782	0.5901
32	0.0115	0.0815	0.6119

Table 1. Relative impact of tiling

Recall that tile size b means we operate on the whole matrix by $b \times n$ block rows. What we see is that, a part from the fine grained computation, the difference with different tile sizes is marginal. This is certainly due the fact that the matricial operation dominate as we will see on the next results. However, as previously mentioned, our kernel for block APP is not sufficiently optimized, otherwise we would have certainly observe a more significant difference. Nevertheless, we observe a factor 2 improvement between using tile of size 20 and the fine-grained version for instance. Now, we reconsider our three matrices and perform a scalability test from 1 SPE to 8 SPEs. In figure 7, we display the global execution time (measured from the PPU) with various tile sizes in $\{1, 4, 8, 12, 16\}$, σ refers to the speedup compared to 1 SPE.

	1 SPE	2 SPEs		8 SPEs	
Tile	t(s)	t(s)	σ	t(s)	σ
1	1.3	1.25	1.06	0.31	4.29
4	0.8	0.41	2.00	0.11	7.78
8	0.7	0.39	1.99	0.11	7.21
12	0.7	0.36	2.08	0.10	7.93
16	0.7	0.40	1.97	0.14	5.65

(a) Performance with a 256×256 matrix

	1 SPE	2 SPEs		8 SPEs	
Tile	t(s)	t(s)	σ	t(s)	σ
1	0.892	0.434	2.05	0.213	4.19
4	0.637	0.318	2.00	0.080	7.96
8	0.608	0.304	2.00	0.078	7.79
2	0.584	0.293	1.99	0.074	7.88
6	0.602	0.302	1.99	0.083	7.23

(b) Performance with a 512×512 matrix

	1 SPE	2 SPEs		8 SPEs	
Tile	t(s)	t(s)	σ	t(s)	σ
1	6.67	3.28	2.03	1.60	4.16
4	5.01	2.50	2.00	0.62	7.99
8	4.79	2.39	2.00	0.60	7.95
12	4.70	2.32	2.02	0.56	8.36
16	4.72	2.36	2.00	0.60	7.79

(c) Performance with a 1024×1024 matrix

Figure 7. Timings on a CELL QS22

Again, a part from the fine-grained version, we observe a perfect scaling (sometime superlinear) of our program. In order to illustrate the efficiency of our method (scheduling + DMA + synchronization), we show in table 2 the timings with our 1024×1024 matrix, using a version of our program where the block APP kernel is disable. We clearly see that the overhead due to our method is definitely negligible and that the absolute performance now relies on the block APP kernel. By replacing our APP kernel code by a fast implementation similar to that of the matrix product in [12], our implementation achieves 80 GFLOPS (note that since

the limit for the APP is 102 GFLOPS as the "multiply-add" cannot be used).

	1 SPE	2 SPEs		8 SPEs	
Tile	t(s)	t(s)	σ	t(s)	σ
1	1.87	1.20	1.56	0.53	3.49
4	0.39	0.47	0.83	0.11	3.70
8	0.19	0.12	1.64	0.06	3.78
12	0.12	0.08	1.65	0.03	4.02
16	0.09	0.06	1.63	0.03	3.78

Table 2. DMA timings with a 1024×1024 matrix

7 Conclusion

Definitely, SPE to SPE communication can be used to derive efficient linear SPMD program on the CELL. This is the case with our solution for the *algebraic path problem*. Our algorithm has lot of interesting properties that match the characteristics and capabilities of the CELL. Moreover, we think that our linear array scheduling can be implemented in the same way with a cluster of CELLs (same array with border communications done with MPI). This has to be investigated, together with an SPU optimization of the APP kernel. We are confident that, together with our scheduling and implementation strategy, this will allow us to offer a solution very close to the peak performance even on huge instances. We also believe that our methodology could be successfully used at algorithmic level to derive efficient CELL programs for many other applications.

References

[1] Cell SDK 3.0. `www.ibm.com/developerworks/power/cell`.

[2] D. Cachera, S. Rajopadhye, T. Risset, C. Tadonki, *Parallelization of the Algebraic Path Problem on Linear SIMD/SPMD Arrays*, IRISA report n 1409, july 2001.

[3] R. N. Floyd, *Algorithm 97 (shortest path)*, Comm. ACM, 5(6):345, 1962.

[4] S.-C. Han, F. Franchetti, and M. Pschel, *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.

[5] F. Irigoin and R. Triolet, *Supernode partitioning*, in 15th ACM Symposium on Principles of Programming Languages, pp. 319-328, ACM, january 1988.

[6] Jakub Kurzak and Jack Dongarra, *QR factorization for the Cell Broadband Engine*, Scientific Programming, vol. 17(1-2), P. 31-42, 2009.

[7] Jakub Kurzak, Alfredo Buttari, and Jack Dongarra, *Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization*, `www.netlib.org/lapack/lawnspdf/lawn184.pdf`

[8] Kazuya MATSUMOTO and Stanislav 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, 2009 .

[9] S.-Y KUNG, S.-C. LO ; LEWIS P. S., *Optimal systolic design for the transitive closure and the shortest path problems*, IEEE transactions on computers ISSN 0018-9340, 1987, vol. 36, no5, pp. 603-614.

[10] H. Peter Hofstee, *Power Efficient Processor Design and the Cell Processor*, `http://www.hpcaconf.org/hpca11/slides/Cell_Public_Hofstee.pdf`.

[11] Vinjamuri, S. Prasanna, V.K., *Transitive closure on the cell broadband engine: A study on self-scheduling in a multicore processor*, IEEE International Parallel & Distributed Processing Symposium (IPDPS), p. 1 - 11, Rome, Italy, 23-29 May 2009.

[12] `http://www.tu-dresden.de/zih/cell/matmul`

[13] S. Sen and S. Chatterjee, *Towards a theory of cache-efficient algorithms*, SODA 2000.

[14] G. Venkataraman, S. Sahni, and S. Mukhopadhyaya, *A blocked all-pairs shortest-paths algorithm*, J. Experimental Algorithmics, vol.8, no.2.2, 2003.

[15] S. Warshall, *A Therorem on Boolean Matrices*, JACM, 9(1):11-12, 1962.

[16] 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.

[17] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, *Scientific Computing Kernels on the Cell Processor*, International Journal of Parallel Programming, 2007.

[18] Jingling Xue, *Loop tiling for parallelism*, Kluwer 2000.