Parallel Computing 35 (2009) 138–150

Contents lists available at ScienceDirect

Parallel Computing

journal homepage: www.elsevier.com/locate/parco

Optimizing matrix multiplication for a short-vector SIMD architecture – CELL processor

Jakub Kurzak [a,*], Wesley Alvaro [a], Jack Dongarra [a,b,c,d]

[a] *Department of Electrical Engineering and Computer Science, University of Tennessee, United States*
[b] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, United States*
[c] *School of Mathematics, University of Manchester, United States*
[d] *School of Computer Science, University of Manchester, United States*

A R T I C L E   I N F O

Article history:
Received 29 May 2008
Received in revised form 18 September 2008
Accepted 16 December 2008
Available online 21 January 2009

Keywords:
Instruction level parallelism
Single Instruction Multiple Data
Synergistic Processing Element
Loop optimizations
Vectorization

A B S T R A C T

Matrix multiplication is one of the most common numerical operations, especially in the area of dense linear algebra, where it forms the core of many important algorithms, including solvers of linear systems of equations, least square problems, and singular and eigenvalue computations. The STI CELL processor exceeds the capabilities of any other processor available today in terms of peak single precision, floating point performance, aside from special purpose accelerators like *Graphics Processing Units* (GPUs).

In order to fully exploit the potential of the CELL processor for a wide range of numerical algorithms, fast implementation of the matrix multiplication operation is essential. The crucial component is the matrix multiplication kernel crafted for the short vector Single Instruction Multiple Data architecture of the Synergistic Processing Element of the CELL processor. In this paper, single precision matrix multiplication kernels are presented implementing the $C = C - A \times B^T$ operation and the $C = C - A \times B$ operation for matrices of size 64×64 elements. For the latter case, the performance of 25.55 Gflop/s is reported, or 99.80% of the peak, using as little as 5.9 kB of storage for code and auxiliary data structures.

1. Introduction

The *CELL Broadband Engine Architecture* (CBEA) has been developed jointly by the alliance of Sony, Toshiba and IBM (STI). The CELL processor is a multi-core architecture consisting of a standard processor, the *Power Processing Element* (PPE), and eight short-vector *Single Instruction Multiple Data* (SIMD) processors, referred to as the *Synergistic Processing Elements* (SPEs). The SPEs are equipped with *scratchpad memory* referred to as the *Local Store* (LS) and a *Memory Flow Controller* (MFC) to perform *Direct Memory Access* (DMA) transfers of code and data between the system memory and the Local Store. This paper is only concerned with the design of computational micro-kernels for the SPE in order to fully exploit *Instruction Level Parallelism* (ILP) provided by its SIMD architecture. Issues related to parallelization of code for execution on multiple SPEs, including intra-chip communication and synchronization, are not discussed here. SPE architectural details important to the discussion are presented in Section 5.1 and also throughout the text, as needed. Plentiful information about the design of the CELL processor and CELL programming techniques is in public the domain [1,2].

* Corresponding author.
E-mail addresses: dongarra@cs.utk.edu, dongarra@utk.edu (J. Kurzak).

0167-8191/$ - see front matter © 2009 Elsevier B.V. All rights reserved.
doi:10.1016/j.parco.2008.12.010

2. Motivation

The current trend in processor design is towards chips with multiple processing units, commonly referred to as *multi-core* processors [3–5]. It has been postulated that building blocks of future architectures are likely to be simple processing elements with shallow pipelines, in-order execution, and SIMD capabilities [6]. It has also been pointed out that direct control over the memory hierarchy may be desired, and software-managed scratchpad memory may be superior to traditional caches [6].

It can be observed that the Synergistic Processing Element of the CELL processor closely matches this description. There is no doubt that future processors will differ significantly from the current designs and will reshape the way of thinking about programming such systems. By the same token, investigation into micro-kernel development for the SPE may have a broader impact by providing an important insight into programming future multi-core architectures.

2.1. Performance considerations

State of the art numerical linear algebra software utilizes *block algorithms* in order to exploit the memory hierarchy of traditional cache-based systems [7,8]. Public domain libraries such as LAPACK [9] and ScaLAPACK [10] are good examples. These implementations work on square or rectangular submatrices in their inner loops, where operations are encapsulated in calls to *Basic Linear Algebra Subroutines* (BLAS) [11], with emphasis on expressing the computation as Level 3 BLAS, *matrix–matrix* type, operations. Frequently, the call is made directly to the matrix multiplication routine _GEMM. At the same time, all the other Level 3 BLAS can be defined in terms of _GEMM and a small amount of Levels 1 and 2 BLAS [12].

A lot of effort has been invested in optimized BLAS by hardware vendors as well as academic institutions through projects such as ATLAS [13] and GotoBLAS [14]. At the same time, the inefficiencies of the BLAS layer have been pointed out [15] as well as the shortcomings of its fork-join parallelization model [16]. Owing to this, the emerging trend in linear algebra is towards the use of specialized data structures such as *Block Data Layout* (BDL) [17,18] and the expression of algorithms directly in terms of specialized inner-kernels [19]. Although application of these techniques is not always straightforward, problems can be often remedied by novel algorithmic approaches [20,21].

The innovation in CELL software has been progressing faster than elsewhere, with direct use of inner-kernels, out-of-order execution and Block Data Layout being a common practice [22–24]. As a result, performance of algorithms comes much closer to the speed of _GEMM for much smaller problem sizes [24]. Any improvement to the _GEMM routine immediately benefits the entire algorithm, which makes the optimization of the _GEMM routine yet more important for the CELL processor.

2.2. Code size considerations

In the current implementation of the CELL BE architecture, the SPEs are equipped with a Local Store of 256 kB. It is a common practice to use tiles of 64×64 elements for dense matrix operations in single precision [22,25,26,23,24]. Such tiles occupy a 16 kB buffer in the Local Store. Between six and eight buffers are necessary to efficiently implement even such a simple operation as matrix multiplication [22,25,26]. Also, more complex operations, such as matrix factorizations, commonly allocate eight buffers [23,24], which consume 128 kB of Local Store. In general, it is reasonable to assume that half of the Local Store is devoted to application data buffers. At the same time, the program may rely on library frameworks like ALF [27] or MCF [28], and utilize numerical libraries such as SAL [29], SIMD Math [30] or MASS [31], which consume extra space for the code. In the development stage, it may also be desirable to generate execution traces for analysis with tools like TATL™ [32] or Paraver [33], which require additional storage for event buffers. Finally, Local Store also houses the SPE stack, starting at the highest LS address and growing towards lower addresses with no overflow protection.

It should be quite obvious that the Local Store is a scarce resource and any *real-world* application is facing the problem of fitting tightly coupled components together in the limited space. SPE code can be replaced at runtime and the mechanism of *overlays* [34] can be of assistance with dynamic code management. Nevertheless, the use of kernels of tens of thousands of kilobytes in size (Section 3) does not seem adequate for other purposes than to implement *micro-benchmarks*.

3. Related work

This article only discusses canonical implementations of matrix multiplication of $O(N^3)$ complexity. Although lower complexity algorithms exist, such as the one by Strassen [35] or the one by Coppersmith and Winograd [36], they are not discussed here.

Little literature exists about implementing matrix operations using short-vector SIMD capabilities. Implementation of matrix multiplication $C = C + A \times B^T$ using Intel *Streaming SIMD Extensions* (SSE) was reported by Aberdeen and Baxter [37]. Analysis of performance considerations of various computational kernels for the CELL processor, including the _GEMM kernel, was presented by Williams et al. [38,39], with results based mostly on simulation. The first implementation of the matrix multiplication kernel $C = A \times B$ for the CELL processor was reported by Chen et al. [22]. Performance of 25.01 Gflop/s was reported on a single SPE with register usage of 69. The article describes C language implementation, which is now publicly distributed with the IBM CELL SDK. More recently assembly language implementation of the matrix multipli-

Table 1
Selected odd and even pipeline instruction groups and their latencies.

Instructions	Pipeline		Latency (cycles)
	Even	Odd	
Single precision floating point	×		6
Immediate loads, logical operations, integer add/subtract	×		2
Element rotates and shifts	×		4
Byte shuffles, quadword rotates and shifts		×	4
Loads/stores, branch hints		×	6
Branches		×	4

cation $C = C - A \times B$ was reported by Hackenberg[25,26]. Performance of 25.40 Gflop/s was reported. Register usage of 71 can be established by inspection of the publicly available code.

Both codes mentioned above were developed using very aggressive unrolling, resulting in a single loop with a huge body of straight-line code. Multiplication of 64×64 matrices requires $64 \times 64 \times 64 = 262{,}144$ multiplications and additions (or subtractions). In single precision, the calculation can be implemented by 262,144/ 4 = 65,536 fused multiply–add (FMA) SIMD operations or fused multiply–subtract (FNMS) SIMD operations. Both codes place 4096 of these operations in the body of a loop, which iterates 16 times and results in the size of the first code of roughly 32 kB and the size of the second one close to 26 kB. Since the first code is in C, the exact size is compiler dependent.

CELL BLAS library released as part of the SDK 3.0 [40] includes an SPE SGEMM kernel for multiplication of 64×64 matrices. The routine is not available in source form. The size of the object code is over 32 kB.

4. Original contribution

In this publication, an implementation of the _GEMM kernel $C = C - A \times B^T$ is reported, which, to the best knowledge of the authors, has not been reported before. Also, an implementation of the _GEMM kernel $C = C - A \times B$ is reported, which achieves better performance than the kernels reported before, and at the same time occupies more than four times less space. It is also shown that the latter kernel is optimal, in the sense that neither performance can be further improved nor code size decreased.

It is also the intention of the authors to demystify the topic by clearly explaining the careful analysis behind optimized implementation of computational micro-kernels exploiting SIMD ILP, VLIW-like, dual-issue and other low-level architectural features of the computational cores of the CELL processor.

5. Implementation

5.1. SPU architecture overview

The core of the SPE is the *Synergistic Processing Unit* (SPU). The SPU [41–43] is a RISC-style SIMD processor featuring 128 general purpose registers and 32-bit fixed length instruction encoding. SPU includes instructions that perform single precision floating point, integer arithmetic, logicals, loads, stores, compares and branches. SPU includes nine execution units organized into two pipelines, referred to as the odd and even pipeline. Instructions are issued in-order and two independent instructions can be issued simultaneously if they belong to different pipelines (Table 1).

SPU executes code form the Local Store and operates on data residing in the Local Store, which is a fully pipelined, single-ported, 256 kB of *Static Random Access Memory* (SRAM). Load and store instructions are performed within local address space, which is untranslated, unguarded and noncoherent with respect to the system address space. Loads and stores transfer 16 bytes of data between the register file and the Local Store, and complete with fixed six-cycle delay and without exception.

SPU does not perform hardware branch prediction and omits branch history tables. Instead, the SPU includes a *Software Managed Branch Target Buffer* (SMBTB), which holds a single branch target and is loaded by software. A mispredicted branch flushes the pipelines and costs 18 cycles. A correctly hinted branch can execute in one cycle. Since both branch hint and branch instructions belong to the odd pipeline, proper use of SMBTB can result in zero overhead from branching for a compute-intensive loop dominated by even pipeline instructions.

5.2. Loop construction

The main tool in loop construction is the technique of loop unrolling [44]. In general, the purpose of loop unrolling is to avoid pipeline stalls by separating dependent instructions by a distance in clock cycles equal to the corresponding pipeline latencies. It also decreases the overhead associated with advancing the loop index and branching. On the SPE it serves the additional purpose of balancing the ratio of instructions in the odd and even pipeline, owing to register reuse between iterations.

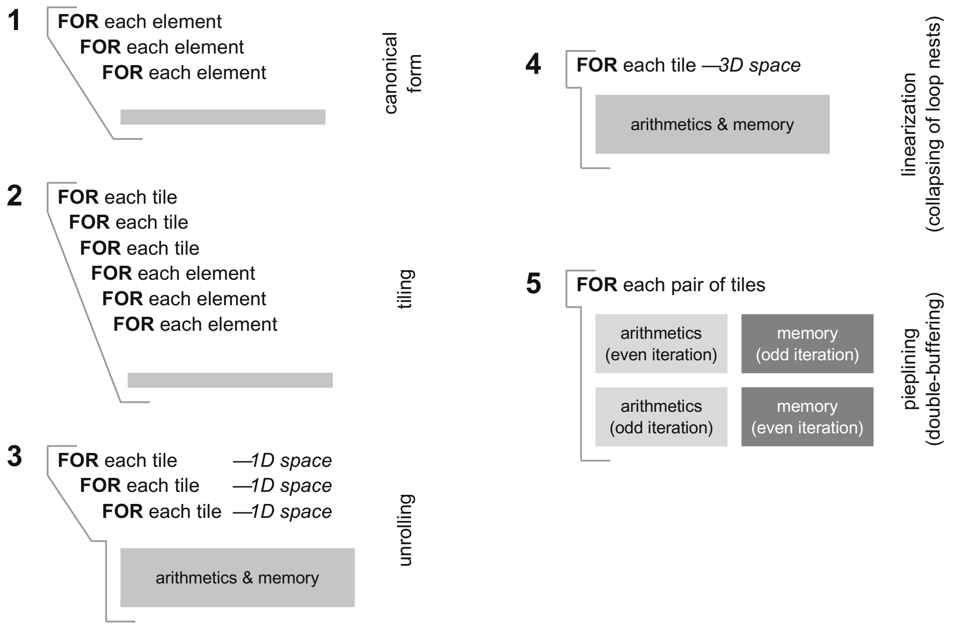

Fig. 1. Basic steps of _GEMM loop optimization.

In the canonical form, matrix multiplication $C_{m\times n} = A_{m\times k} \times B_{k\times n}$ consists of three nested loops iterating over the three dimensions m, n and k. Loop tiling [45] is applied to improve the locality of reference and to take advantage of the $O(n^3)/O(n^2)$ ratio of arithmetic operations to memory accesses. This way register reuse is maximized and the number of loads and stores is minimized.

Conceptually, tiling of the three loops creates three more inner loops, which calculate a product of a submatrix of A and a submatrix of B and updates a submatrix of C with the partial result. Practically, the body of these three inner loops is subject to complete unrolling to a single block of a straight-line code. The tile size is picked such that the cross-over point between arithmetic and memory operations is reached, which means that there is more FMA or FNMS operations to fill the even pipeline than there is load, store and shuffle operations to fill the odd pipeline.

The resulting structure consists of three outer loops iterating over tiles of A, B and C. Inevitably, nested loops induce mispredicted branches, which can be alleviated by further unrolling. Aggressive unrolling, however, leads quickly to undesired code bloat. Instead, the three-dimensional problem can be linearized by replacing the loops with a single loop performing the same traversal of the iteration space. This is accomplished by traversing tiles of A, B and C in a predefined order derived as a function of the loop index. A straightforward row/column ordering can be used and tile pointers for each iteration can be constructed by simple transformations of the bits of the loop index.

At this point, the loop body still contains *auxiliary* operations that cannot be overlapped with arithmetic operations. These include initial loads, stores of final results, necessary data rearrangement with splats (copy of one element across a vector) and shuffles (permutations of elements within a vector), and pointer advancing operations. This problem is addressed by *double-buffering*, on the register level, between two loop iterations. The existing loop body is duplicated and two separate blocks take care of the even and odd iteration, respectively. Auxiliary operations of the even iteration are hidden behind arithmetic instructions of the odd iteration and vice versa, and disjoint sets of registers are used where necessary. The resulting loop is preceded by a small body of *prologue* code loading data for the first iteration, and then followed by a small body of *epilogue* code, which stores results of the last iteration. Fig. 1 shows the optimization steps leading to a high performance implementation of the _GEMM inner kernel.

5.3. $C = C - A \times Btrans$

The BLAS $C = C - A \times B^T$ _GEMM is a very common linear algebra operation. LAPACK relies on this operation for implementation of many matrix transformations, including Cholesky factorization (_POTRF), LDL^T factorization (_SYTRF), QR factorization (_GEQRF-calling _GEMM indirectly through the _LARFB routine), and bidiagonal reduction (_GEBRD). The $C = C - A \times B^T$ micro-kernel is also a building block for Level 3 BLAS routines other than _GEMM, e.g., symmetric rank k update (_SYRK). Specifically, implementation of the Cholesky factorization for the CELL processor, based on this micro-kernel coded in C, has been reported by the authors of this publication [24].

Before going into details, it should be noted, that matrix storage follows C-style row-major format. It is not as much a carefull design decision, as compliance with the common practice on the CELL processor. It can be attributed to C compilers

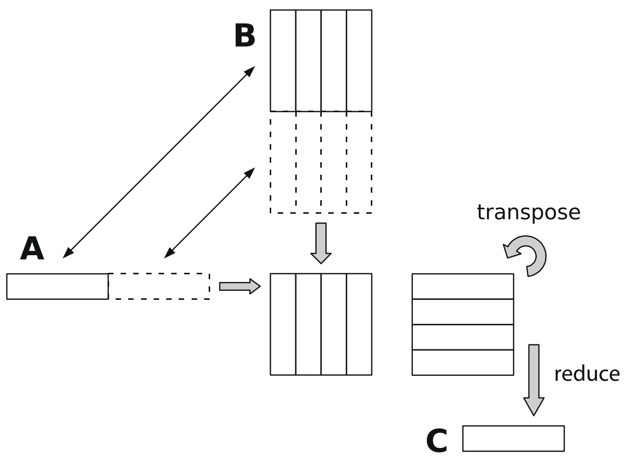

Fig. 2. Basic operation of the $C = C - A \times B^T$ matrix multiplication micro-kernel.

being the only ones allowing to exploit short-vector capabilities of the SPEs through C language SIMD extensions. If compliance with libraries relying on legacy FORTRAN API is required, a translation operation is necessary. However, translation is required anyway, since implementations of dense linear algebra routines on the CELL processor rely on Block Data Layout. Typically, the two conversions are combined in one operation, which introduces an acceptable overhead [23].

An easy way to picture the $C = C - A \times B^T$ operation is to represent it as the standard matrix vector product $C = C - A \times B$, where A is stored using row-major order and B is stored using column-major order. It can be observed that in this case a row of A can readily be multiplied with a column of B to yield a vector containing four partial results, which need to be summed up to produce one element of C. The vector reduction step introduces superfluous multiply–add operations. In order to minimize their number, four row-column products are computed, resulting in four vectors, which need to be internally reduced. The reduction is performed by first transposing the 4×4 element matrix represented by the four vectors and then applying four vector multiply–add operations to produce a result vector containing four elements of C. The basic scheme is depicted in Fig. 2.

The crucial design choice to be made is the right amount of unrolling, which is equivalent to deciding the right tile size in terms of the triplet $\{m, n, k\}$ (Here sizes express numbers of individual floating-point values, not vectors). Unrolling is mainly used to minimize the overhead of jumping and advancing the index variable and associated pointer arithmetic. It has been pointed out in Section 5.1 that both jump and jump hint instructions belong to the odd pipeline and, for compute intensive loops, can be completely hidden behind even pipeline instructions and thus introduce no overhead. In terms of the overhead of advancing the index variable and related pointer arithmetic, it will be shown in Section 5.5 that all of these operations can be placed in the odd pipeline as well. In this situation, the only concern is balancing even pipeline, arithmetic instructions with odd pipeline, data manipulation instructions.

Simple analysis can be done by looking at the number of floating-point operations versus the number of loads, stores and shuffles, under the assumption that the size of the register file is not a constraint. The search space for the $\{m, n, k\}$ triplet is further truncated by the following criteria: only powers of two are considered in order to simplify the loop construction; the maximum possible number of 64 is chosen for k in order to minimize the number of extraneous floating-point instructions performing the reduction of partial results; only multiplies of four are selected for n to allow for efficient reduction of partial results with eight shuffles per one output vector of C. Under these constraints, the entire search space can be easily analyzed.

Table 2 shows how the number of each type of operation is calculated. Table 3 shows the number of even pipeline, floating-point instructions including the reductions of partial results. Table 4 shows the number of even pipeline instructions minus the number of odd pipeline instructions including loads, stores and shuffles (not including jumps and pointer arith-

Table 2
Numbers of different types of operations in the computation of one tile of the $C = C - A \times B^T$ micro-kernel as a function of tile size ($\{m, n, 64\}$ triplet).

Type of operation	Pipeline		Number of operations
	Even	Odd	
Floating point	×		$(m \times n \times 64)/4 + m \times n$
Load A		×	$m \times 64/4$
Load B		×	$64 \times n/4$
Load C		×	$m \times n/4$
Store C		×	$m \times n/4$
Shuffle		×	$m \times n/4 \times 8$

Table 3
Number of even pipeline, floating-point operations in the computation of one tile of the micro-kernel $C = C - A \times B^T$ as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	68	136	272	544	1088
2	136	272	544	1088	2176
4	272	544	1088	2176	4352
8	544	1088	2176	4352	8704
16	1088	2176	4352	8704	17,408
32	2176	4352	8704	17,408	34,816
64	4352	8704	17,408	34,816	69,632

Table 4
Number of spare odd pipeline slots in the computation of one tile of the $C = C - A \times B^T$ micro-kernel as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	−22	−28	−40	−64	−112
2	20	72	176	384	800
4	104	272	608	1280	2624
8	272	672	1472	3072	6272
16	608	1472	3200	6656	13,568
32	1280	3072	6656	13,824	28,160
64	2624	6272	13,568	28,160	57,344

metic). In other words, Table 4 shows the number of spare odd pipeline slots before jumps and pointer arithmetic are implemented. Finally, Table 5 shows the size of code involved in calculations for a single tile. It is important to note here that the double-buffered loop is twice the size.

It can be seen that the smallest unrolling with a positive number of spare odd pipeline slots is represented by the triplet $\{2, 4, 64\}$ and produces a loop with 136 floating-point operations. However, this unrolling results in only 20 spare slots, which would barely fit pointer arithmetic and jump operations. Another aspect is that the odd pipeline is also used for instruction fetch and near complete filling of the odd pipeline may cause instruction depletion, which in rare situations can even result in an indefinite stall [46].

The next larger candidates are triplets $\{4, 4, 64\}$ and $\{2, 8, 64\}$, which produce loops with 272 floating-point operations, and 104 or 72 spare odd pipeline slots, respectively. The first one is an obvious choice, giving more room in the odd pipeline and smaller code. It turns out that the $\{4, 4, 64\}$ unrolling is actually the most optimal of all, in terms of the overall routine footprint, when the implementation of pointer arithmetic is taken into account, as further explained in Section 5.5.

It can be observed that the maximum performance of the routine is ultimately limited by the extra floating-point operations, which introduce an overhead not accounted for in the formula for operation count in matrix multiplication: $2 \times m \times n \times k$. For matrices of size 64×64, every 64 multiply–add operations require four more operations to perform the intra-vector reduction. This sets a hard limit on the maximum achievable performance to $64/(64 + 4) \times 25.6 = 24.09$ [Gflop/s], which is roughly 94% of the peak.

5.4. C=C – A×B

Perhaps the most important usage of the BLAS $C = C - A \times B$ _GEMM operation is in Gaussian elimination. This operation is employed by LAPACK's implementation of LU factorization (_GETRF), which is also a basis for the Linpack benchmark [47] used to rank supercomputers on the Top500 list [48]. The $C = C - A \times B$ micro-kernel is also a building block for Level 3 BLAS routines other than _GEMM, e.g., triangular solve (_TRSM). Specifically, implementation of LU factorization for the CELL processor, based on this micro-kernel coded in C has been reported by Chen et al. [22].

Table 5
The size of code for the computation of one tile of the $C = C - A \times B^T$ micro-kernel as a function of tile size ({m, n, 64} triplet).

M/N	4	8	16	32	64
1	1.2	1.2	2.3	4.5	8.9
2	1.0	1.8	3.6	7.0	13.9
4	1.7	3.2	6.1	12.0	23.8
8	3.2	5.9	11.3	22.0	43.5
16	6.1	11.3	21.5	42.0	83.0
32	12.0	22.0	42.0	82.0	162.0
64	23.8	43.5	83.0	162.0	320.0

Here, same as before, row major storage is assumed. The key observation is that multiplication of one element of A with one row of B contributes to one row of C. Owing to that, the elementary operation splats an element of A over a vector, multiplies this vector with a vector of B and accumulates the result in a vector of C (Fig. 3). Unlike for the other kernel, in this case no extra floating-point operations are involved.

Same as before, the size of unrolling has to be decided in terms of the triplet $\{m, n, k\}$. This time, however, there is no reason to fix any dimension. Nevertheless, similar constraints to the search space apply: all dimensions have to be powers of two, and additionally only multiples of four are allowed for n and k to facilitate efficient vectorization and simple loop construction. Table 6 shows how the number of each type of operation is calculated. Table 7 shows the number of even pipeline, floating-point instructions. Table 8 shows the number of even pipeline instructions minus the number of odd pipeline instructions including loads, stores and splats (not including jumps and pointer arithmetic). In other words, Table 8 shows the number of spare odd pipeline slots before jumps and pointer arithmetic are implemented. Finally, Table 9 shows the size of code involved in calculations for a single tile. It is should be noted again that the double-buffered loop is twice the size.

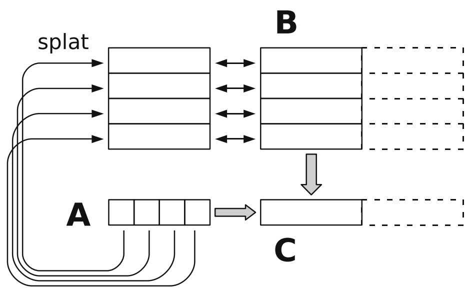

Fig. 3. Basic operation of the $C = C - A \times B$ matrix multiplication micro-kernel.

Table 6
Numbers of different types of operations in the computation of one tile of the $C = C - A \times B$ micro-kernel as a function of tile size ($\{m, n, k\}$).

Type of operation	Pipeline		Number of operations
	Even	Odd	
Floating point	×		$(m \times n \times k)/4$
Load A		×	$m \times k/4$
Load B		×	$k \times n/4$
Load C		×	$m \times n/4$
Store C		×	$m \times n/4$
Splat		×	$m \times k$

Table 7
Number of even pipeline operations in the computation of one tile of the micro-kernel $C = C - A \times B$ as a function of tile size ($\{m, n, k\}$).

K	M/N	4	8	16	32	64
4	1	4	8	16	32	64
4	2	8	16	32	64	128
4	4	16	32	64	128	256
4	8	32	64	128	256	512
4	16	64	128	256	512	1024
4	32	128	256	512	1024	2048
4	64	256	512	1024	2048	4096
8	1	8	16	32	64	128
8	2	16	32	64	128	256
8	4	32	64	128	256	512
8	8	64	128	256	512	1024
8	16	128	256	512	1024	2048
8	32	256	512	1024	2048	4096
8	64	512	1024	2048	4096	8192
16	1	16	32	64	128	256
16	2	32	64	128	256	512
16	4	64	128	256	512	1024
16	8	128	256	512	1024	2048
16	16	256	512	1024	2048	4096
16	32	512	1024	2048	4096	8192
16	64	1024	2048	4096	8192	16,384

Table 8
Number of spare odd pipeline slots in the computation of one tile of the $C = C - A \times B$ micro-kernel as a function of tile size ({m, n, k}).

K	M/N	4	8	16	32	64
4	1	−7	−9	−13	−21	−37
4	2	−10	−10	−10	−10	−10
4	4	−16	−12	−4	12	44
4	8	−28	−16	8	56	152
4	16	−52	−24	32	144	368
4	32	−100	−40	80	320	800
4	64	−196	−72	176	672	1664
8	1	−12	−14	−18	−26	−42
8	2	−16	−12	−4	12	44
8	4	−24	−8	24	88	216
8	8	−40	0	80	240	560
8	16	−72	16	192	544	1248
4	32	−136	48	416	1152	2624
4	64	−264	112	864	2368	5376
16	1	−22	−24	−28	−36	−52
16	2	−28	−16	8	56	152
16	4	−40	0	80	240	560
16	8	−64	32	224	608	1376
16	16	−112	96	512	1344	3008
16	32	−208	224	1088	2816	6272
16	64	−400	480	2240	5760	12,800

Table 9
The size of code for the computation of one tile of the $C = C - A \times B$ micro-kernel as a function of tile size ({m, n, k}).

K	M/N	4	8	16	32	64
4	1	0.1	0.1	0.2	0.3	0.6
4	2	0.1	0.2	0.3	0.5	1.0
4	4	0.2	0.3	0.5	1.0	1.8
4	8	0.4	0.6	1.0	1.8	3.4
4	16	0.7	1.1	1.9	3.4	6.6
4	32	1.4	2.2	3.7	6.8	12.9
4	64	2.8	4.3	7.3	13.4	25.5
8	1	0.1	0.2	0.3	0.6	1.2
8	2	0.2	0.3	0.5	1.0	1.8
8	4	0.3	0.5	0.9	1.7	3.2
8	8	0.7	1.0	1.7	3.1	5.8
8	16	1.3	1.9	3.3	5.9	11.1
4	32	2.5	3.8	6.4	11.5	21.8
4	64	5.0	7.6	12.6	22.8	43.0
16	1	0.2	0.3	0.6	1.1	2.2
16	2	0.4	0.6	1.0	1.8	3.4
16	4	0.7	1.0	1.7	3.1	5.8
16	8	1.3	1.9	3.1	5.6	10.6
16	16	2.4	3.6	6.0	10.8	20.3
16	32	4.8	7.1	11.8	21.0	39.5
16	64	9.6	14.1	23.3	41.5	78.0

It can be seen that the smallest unrolling with a positive number of spare odd pipeline slots produces a loop with 128 floating-point operations. Five possibilities exist, with the triplet $\{4, 16, 8\}$ providing the highest number of 24 spare odd pipeline slots. Again, such unrolling would both barely fit pointer arithmetic and jump operations and be a likely cause of instruction depletion.

The next larger candidates are unrollings producing loops with 256 floating-point operations. There are 10 such cases, with the triplet $\{4, 32, 8\}$ being the obvious choice for the highest number of 88 spare odd pipeline slots and the smallest code size. It also turns out that this unrolling is actually the most optimal in terms of the overall routine footprint, when the implementation of pointer arithmetic is taken into account, as further explained in Section 5.5.

Unlike for the other routine, the maximum performance is not limited by any extra floating-point operations, and performance close to the peak of 25.6 Gflop/s should be expected.

5.5. Advancing tile pointers

The remaining issue is the one of implementing the arithmetic calculating the tile pointers for each loop iteration. Due to the size of the input matrices and the tile sizes being powers of two, this is a straightforward task. The tile offsets can be calculated from the tile index and the base addresses of the input matrices using integer arithmetic and bit manipulation instructions (bitwise logical instructions and shifts). Fig. 4 shows a sample implementation of pointer arithmetic for the kernel $C = C - A \times B^T$ with unrolling corresponding to the triplet $\{4, 4, 64\}$. *Abase*, *Bbase* and *Cbase*, are base addresses of the input matrices and the variable *tile* is the tile index running from 0 to 255; *Aoffs*, *Boffs* and *Coffs* are the calculated tile offsets.

Fig. 5 shows the result of compiling the sample C code from Fig. 4 to assembly code. Although a few variations are possible, the resulting assembly code will always involve a similar combined number of integer and bit manipulation operations. Unfortunately, all these instructions belong to the even pipeline and will introduce an overhead, which cannot be hidden behind floating point operations, like it is done with loads, stores, splats and shuffles.

One way of minimizing this overhead is extensive unrolling, which creates a loop big enough to make the pointer arithmetic negligible. An alternative is to eliminate the pointer arithmetic operations from the even pipeline and replace them with odd pipeline operations. With the unrolling chosen in Sections 5.3 and 5.4, the odd pipeline offers empty slots in abundance. It can be observed that, since the loop boundaries are fixed, all tile offsets can be calculated in advance. At the same time, the operations available in the odd pipeline include loads, which makes it a logical solution to precalculate and tabulate tile offsets for all iterations. It still remains necessary to combine the offsets with the base addresses, which are not known beforehand. However, under additional alignment constraints, offsets can be combined with bases using shuffle instructions, which are also available in the odd pipeline. As will be further shown, all instructions that are not floating point arithmetic can be removed from the even pipeline.

The precalculated offsets have to be compactly packed in order to preserve space consumed by the lookup table. Since tiles are 16 kB in size, offsets consume 14 bits and can be stored in a 16-bit halfword. Three offsets are required for each loop iteration. With eight halfwords in a quadword, each quadword can store offsets for two loop iterations or a single interation of the pipelined, double-buffered loop. Fig. 6 shows the organization of the offset lookup table.

The last arithmetic operation remaining is the advancement of the iteration variable. It is typical to decrement the iteration variable instead of incrementing it, and branch on non-zero, in order to eliminate the comparison operation, which is also the case here. This still leaves the decrement operation, which would have to occupy the even pipeline. In order to

```
int tile;

vector float *Abase;
vector float *Bbase;
vector float *Cbase;

vector float *Aoffs;
vector float *Boffs;
vector float *Coffs;

Aoffs = Abase + ((tile & ~0x0F) << 2);
Boffs = Bbase + ((tile &  0x0F) << 6);
Coffs = Cbase +  (tile &  0x0F)
              + ((tile & ~0x0F) << 2);
```

Fig. 4. Sample C language implementation of pointer arithmetic for the kernel $C = C - A \times B^T$ with unrolling corresponding to the triplet {4, 4, 64}.

```
lqa  $2,tile
lqa  $3,Abase
andi $4,$2,-16
andi $2,$2,15
shli $6,$4,2
shli $4,$4,6
shli $5,$2,10
a    $2,$2,$6
a    $4,$4,$3
shli $2,$2,4
lqa  $3,Bbase
stqa $4,Aoffs
a    $5,$5,$3
lqa  $3,Cbase
stqa $5,Boffs
a    $2,$2,$3
stqa $2,Coffs
```

Fig. 5. The result of compiling the code from Fig. 4 to assembly language, with even pipeline instructions in bold.

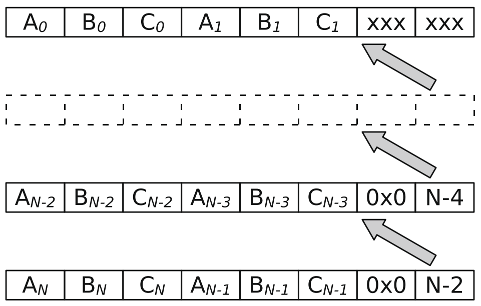

Fig. 6. Organization of the tile offset lookup table. N is the number of tiles.

annihilate the decrement, each quadword containing six offsets for one iteration of the double-buffered loop also contains a seventh entry, which stores the index of the quadword to be processed next (preceding in memory). In other words, the iteration variable, which also serves as the index to the lookup table, is tabulated along with the offsets and loaded instead of being decremented.

Normally, the tile pointers would have to be calculated as a sum of an 18-bit base address and a 14-bit offset, which would require the use of integer addition residing in the even pipeline. With the additional constraint of 16 kB alignment of the base addresses, 14 less significant bits of the base are zero and can be simply replaced with the bits of the offset. The replacement could be implemented with the logical *AND* operation. This would however, again, involve an even pipeline instruction. Instead, both the base addresses and the offsets are initially shifted left by two bits, which puts the borderline between offsets and bases on a byte boundary. At this point the odd pipeline shuffle instruction operating at byte granularity can be used to combine the base with the offset. Finally, the result has to be shifted right by two bits, which can be accomplished by a combination of bit and byte quadword rotations, which also belong to the odd pipeline. Overall, all the operations involved in advancing the double-buffered loop consume 29 extra odd pipeline slots, which is small, given that 208 is available in the case of the first kernel and 176 in the case of the second.

This way, all operations involved in advancing from tile to tile are implemented in the odd pipeline. At the same time, both the branch instruction and the branch hint belong to the odd pipeline. Also, a correctly hinted branch does not cause any stall. As a result, such an implementation produces a continuous stream of floating-point operations in the even pipeline, without a single cycle devoted to any other activity.

The last issue to be discussed is the storage overhead of the lookup table. This size is proportional to the number of iterations of the unrolled loop and reciprocal to the size of the loop body. Using the presented scheme (Fig. 6), the size of the lookup table in bytes equals $N^3/(m \times n \times k) \times 8$. Table 10 presents the overall footprint of the $C = C - A \times B^T$ micro-kernel as a function of the tile size. Table 11 presents the overall footprint of the $C = C - A \times B$ micro-kernel as a function of the tile size. As can be clearly seen, the chosen tile sizes result in the lowest possible storage requirements for the routines.

5.6. Results

Both presented SGEMM kernel implementations produce a continuous stream of floating-point instructions for the duration of the pipelined loop. In both cases, the loop iterates 128 times, processing two tiles in each iteration. The $C = C - A \times B^T$ kernel contains 544 floating-point operations in the loop body and, on a 3.2 GHz processor, delivers 25.54 Gflop/s (99.77% of peak) if actual operations are counted, and 24.04 Gflop/s (93.90% of peak) if the standard formula, $2N^3$, is used for operation count. The $C = C - A \times B$ kernel contains 512 floating-point operations in the loop body and delivers 25.55 Gflop/s (99.80% of peak). Here, the actual operation count equals $2N^3$. At the same time, neither implementation overfills the odd pipeline, which is 31% empty for the first case and 17% empty for the second case. This guarantees no contention between loads

Table 10
The overall footprint of the micro-kernel $C = C - A \times B^T$, including the code and the offset lookup table, as a function of tile size ($\{m, n, 64\}$ triplet).

M/N	4	8	16	32	64
1	9.2	6.3	6.6	10.0	18.4
2	6.0	5.7	8.1	14.5	28.0
4	5.4	7.4	12.8	24.3	47.6
8	7.4	12.3	22.8	44.1	87.1
16	12.8	22.8	43.1	84.1	166.0
32	24.3	44.1	84.1	164.0	324.0
64	47.6	87.1	166.0	324.0	640.0

Table 11
The overall footprint of the micro-kernel $C = C - A \times B$, including the code and the offset lookup table, as a function of tile size ({m, n, 64} triplet).

K	M/N	4	8	16	32	64
4	1	128.1	64.2	32.4	16.7	9.3
4	2	64.2	32.3	16.6	9.1	6.1
4	4	32.4	16.6	9.0	5.9	5.7
4	8	16.7	9.1	5.9	5.6	7.8
4	16	9.4	6.2	5.8	7.9	13.6
4	32	6.8	6.3	8.4	14.0	26.0
4	64	7.5	9.6	15.1	27.0	51.1
8	1	64.2	32.4	16.6	9.2	6.3
8	2	32.4	16.6	9.0	5.9	5.7
8	4	16.7	9.1	5.8	5.3	7.3
8	8	9.3	6.0	5.4	7.1	12.1
8	16	6.6	5.9	7.5	12.3	22.5
4	32	9.1	9.6	13.8	23.5	43.8
4	64	12.1	16.1	25.8	45.8	86.1
16	1	32.4	16.7	9.2	6.3	6.4
16	2	16.7	9.1	5.9	5.6	7.8
16	4	9.3	6.0	5.4	7.1	12.1
16	8	6.5	5.8	7.3	11.8	21.5
16	16	6.9	8.3	12.5	21.8	40.6
16	32	10.6	14.8	23.8	42.1	79.1

Table 12
Summary of the properties of the SPE SIMD SGEMM micro-kernels.

Characteristic	$C = C - A \times B^T$	$C = C - A \times B$
Performance (Gflop/s)	24.04	25.55
Execution time (μs)	21.80	20.52
Fraction of peak *using the* $2 \times M \times N \times K$ *formula* (%)	93.90	99.80
Fraction of peak *using actual number of floating-point instructions* (%)	99.77	99.80
Dual issue rate *odd pipeline workload* (%)	68.75	82.81
Register usage	69	69
Code segment size	4008	3992
Data segment size	2192	2048
Total memory footprint	6200	6040

and stores and DMA operations, and no danger of instruction fetch starvation. Table 12 shows the summary of the kernels' properties.

5.7. Conclusions

Computational micro-kernels are architecture specific codes, where no portability is sought. It has been shown that systematic analysis of the problem combined with exploitation of low-level features of the Synergistic Processing Unit of the CELL processor leads to dense matrix multiplication kernels achieving peak performance without code bloat.

5.8. Future work

This article shows that great performance can be achieved on a SIMD architecture by optimizing code manually. The question remains, whether similar results can be accomplished by automatic vectorization techniques or a combination of auto-vectorization with heuristic techniques based on searching the parameter space. The authors plan to investigate a combination of the *Superworld Level Parallelism* technique for auto-vectorization [49] with heuristic search similar to the ATLAS [13] methodology.

5.9. Code

The code is freely available, under the BSD license and can be downloaded from the author's web site http://icl.cs.utk.edu/alvaro/. A few comments can be useful here. In absence of better tools, the code has been developed with a help of a spreadsheet, mainly for easy manipulation of two columns of instructions for the two pipelines of the SPE. Other useful features were taken advantage of as well. Specifically, color coding of blocks of instructions greatly improves the readability of the code. It is the hope of the authors that such visual representation of code considerably helps the reader's understanding of the techniques involved in construction of optimized SIMD assembly code. Also, the authors put considerable effort in making the software self-contained, in the sense that all tools involved in construction of the code are distributed alongside. That includes the lookup table generation code and the scripts facilitating translation from spreadsheet format to SPE assembly language.

References

[1] IBM Corporation, Cell BE Programming Tutorial, November 2007.
[2] IBM Corporation, Cell Broadband Engine Programming Handbook, Version 1.1, April 2007.
[3] S. Borkar, Design challenges of technology scaling, IEEE Micro 19 (4) (1999) 23–29.
[4] D. Geer, Industry trends: chip makers turn to multicore processors, Computer 38 (5) (2005) 11–13.
[5] H. Sutter, The free lunch is over: a fundamental turn toward concurrency in software, Dr. Dobb's J. 30(3).
[6] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, K.A. Yelick, The Landscape of Parallel Computing Research: A View from Berkeley, Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences Department, University of California at Berkeley, 2006.
[7] J.J. Dongarra, I.S. Duff, D.C. Sorensen, H.A. van der Vorst, Numerical Linear Algebra for High-performance Computers, SIAM, 1998.
[8] J.W. Demmel, Applied Numerical Linear Algebra, SIAM, 1997.
[9] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J.W. Demmel, J.J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, D. Sorensen, LAPACK Users' Guide, SIAM, 1992.
[10] L.S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J.J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, R.C. Whaley, ScaLAPACK Users' Guide, SIAM, 1997.
[11] Basic Linear Algebra Technical Forum, Basic Linear Algebra Technical Forum Standard, August 2001.
[12] B. Kågström, P. Ling, C. van Loan, GEMM-based Level 3 BLAS: high-performance model implementations and performance evaluation Benchmark, ACM Trans. Math. Soft. 24 (3) (1998) 268–302.
[13] ATLAS. <http://math-atlas.sourceforge.net/>.
[14] GotoBLAS. <http://www.tacc.utexas.edu/resources/software/>.
[15] E. Chan, E.S. Quintana-Orti, G. Gregorio Quintana-Orti, R. van de Geijn, Supermatrix out-of-order scheduling of matrix operations for SMP and multicore architectures, in: Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07, 2007, pp. 116–125.
[16] J. Kurzak, J.J. Dongarra, LAPACK Working Note 178: Implementing Linear Algebra Routines on Multi-Core Processors, Tech. Rep. CS-07-581, Electrical Engineering and Computer Science Department, University of Tennessee, 2006.
[17] N. Park, B. Hong, V.K. Prasanna, Analysis of memory hierarchy performance of block data layout, in: International Conference on Parallel Processing, 2002.
[18] N. Park, B. Hong, V.K. Prasanna, Tiling, block data layout, and memory hierarchy performance, IEEE Trans. Parallel Distrib. Syst. 14 (7) (2003) 640–654.
[19] J.R. Herrero, J.J. Navarro, Using nonlinear array layouts in dense matrix operations, in: Workshop on State-of-the-Art in Scientific and Parallel Computing PARA'06, 2006.
[20] A. Buttari, J. Langou, J. Kurzak, J.J. Dongarra, LAPACK Working Note 190: Parallel Tiled QR Factorization for Multicore Architectures, Tech. Rep. CS-07-598, Electrical Engineering and Computer Science Department, University of Tennessee, 2007.
[21] A. Buttari, J. Langou, J. Kurzak, J.J. Dongarra, LAPACK Working Note 191: A Class of Parallel Tiled Linear Algebra Algorithms for Multicore Architectures, Tech. Rep. CS-07-600, Electrical Engineering and Computer Science Department, University of Tennessee, 2007.
[22] T. Chen, R. Raghavan, J. Dale, E. Iwata, Cell Broadband Engine Architecture and its First Implementation, A Performance View, November 2005. <http://www-128.ibm.com/developerworks/power/library/pa-cellperf/>.
[23] J. Kurzak, J.J. Dongarra, Implementation of mixed precision in solving systems of linear equations on the CELL processor, Concurrency Comput. Pract. Exper. 19 (10) (2007) 1371–1385.
[24] J. Kurzak, A. Buttari, J.J. Dongarra, Solving systems of linear equations on the CELL processor using Cholesky factorization, IEEE Trans. Parallel Distrib. Syst. 19 (9) (2008) 1175–1186.
[25] D. Hackenberg, Einsatz und Leistungsanalyse der Cell Broadband Engine, Institut für Technische Informatik, Fakultät Informatik, Technische Universität Dresden, Großer Beleg, February 2007.
[26] D. Hackenberg, Fast Matrix Multiplication on CELL Systems, July 2007. <http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/>.
[27] IBM Corporation, ALF for Cell BE Programmer's Guide and API Reference, November 2007.
[28] M. Pepe, Multi-Core Framework (MCF), Mercury Computer Systems, Version 0.4.4, October 2006.
[29] Mercury Computer Systems, Inc., Scientific Algorithm Library (SAL) Data Sheet, 2006. <http://www.mc.com/uploadedfiles/SAL-ds.pdf>.
[30] IBM Corporation, SIMD Math Library API Reference Manual, November 2007.
[31] I. Corporation, Mathematical Acceleration Subsystem-product Overview, March 2007. <http://www-306.ibm.com/software/awdtools/mass/>.
[32] Mercury Computer Systems, Inc., Trace Analysis Tool and Library (TATL^{TM}) Data Sheet, 2006. <http://www.mc.com/uploadedfiles/tatl-ds.pdf>.
[33] European Center for Parallelism of Barcelona, Technical University of Catalonia, Paraver, Parallel Program Visualization and Analysis Tool Reference Manual, Version 3.1, October 2001.
[34] IBM Corporation, Software Development Kit 2.1 Programmer's Guide, Version 2.1, March 2007.
[35] V. Strassen, Gaussian elimination is not optimal, Numer. Math. 13 (1969) 354–356.
[36] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, J. Symbol. Comput. 9 (3) (1990) 251–280.
[37] D. Aberdeen, J. Baxter, Emmerald: A fast matrix–matrix multiply using Intel's SSE instructions, Concurrency Comput. Pract. Exper. 13 (2) (2001) 103–119.
[38] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, The potential of the cell processor for scientific computing, in: ACM International Conference on Computing Frontiers, 2006.
[39] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, K. Yelick, Scientific computing kernels on the cell processor, Int. J. Parallel Prog. 35 (3) (2007) 263–298.
[40] IBM Corporation, Basic Linear Algebra Subprograms Programmer's Guide and API Reference, November 2007.
[41] B. Flachs, S. Asano, S.H. Dhong, P. Hofstee, G. Gervais, R. Kim, T. Le, P. Liu, J. Leenstra, J. Liberty, B. Michael, H. Oh, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yanoz, A streaming processing unit for a CELL processor, in: IEEE International Solid-State Circuits Conference, 2005, pp. 134–135.
[42] B. Flachs, S. Asano, S.H. Dhong, H.P. Hofstee, G. Gervais, K. Roy, T. Le, L. Peichun, J. Leenstra, J. Liberty, B. Michael, O. Hwa-Joon, S.M. Mueller, O. Takahashi, A. Hatakeyama, Y. Watanabe, N. Yano, D.A. Brokenshire, M. Peyravian, T. Vandung, E. Iwata, The microarchitecture of the synergistic processor for a cell processor, IEEE J. Solid-State Circ. 41 (1) (2006) 63–70.
[43] M. Gschwind, H.P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, Synergistic processing in cell's multicore architecture, IEEE Micro 26 (2) (2006) 10–24.

[44] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, fourth ed., 2006.
[45] S. Muchnick, Advanced Compiler Design and Implementation, Morgan Kaufmann, 1997.
[46] IBM Corporation, Preventing synergistic processor element indefinite stalls resulting from instruction depletion in the Cell Broadband Engine Processor for CMOS SOI 90 nm, Applications Note, Version 1.0, November 2007.
[47] J.J. Dongarra, P. Luszczek, A. Petitet, The LINPACK Benchmark: past, present and future, Concurrency Comput. Pract. Exper. 15 (9) (2003) 803–820.
[48] TOP500 Supercomputing Sites. <http://www.top500.org/>.
[49] J. Shin, J. Chame, M.W. Hall, Exploiting superword-level locality in multimedia extension architectures, J. Instr. Level Parallel. 5 (2003) 1–28.