Fast and Small Short Vector SIMD Matrix Multiplication Kernels for the Synergistic Processing Element of the CELL Processor

Wesley Alvaro[1], Jakub Kurzak[1], and Jack Dongarra[1,2,3]

[1] University of Tennessee, Knoxville TN 37996, USA
[2] Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA
[3] University of Manchester, Manchester, M13 9PL, UK
{alvaro, kurzak, dongarra}@eecs.utk.edu

Abstract. 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. 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 crutial 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 percent of the peak, using as little as 5.9 KB of storage for code and auxiliary data structures.

1 Introduction

The *CELL Broadband Engine* (CBE) processor has been developed jointly by the alliance of Sony, Toshiba and IBM (STI). The CELL processor is an innovative 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. All components are interconnected with the *Element Interconnection Bus* (EIB).

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

M. Bubak et al. (Eds.): ICCS 2008, Part I, LNCS 5101, pp. 935–944, 2008.

not discussed here. SPE architercural details important to the discussion are presented in Sect. 4.1 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 Motivation

The current trend in processor design is towards chips with multiple processing units, commonly referred to as *multi-core* processors [2]. 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 [3]. It can be observed that the Synergistic Processing Element of the CELL processor closely matches this description. By the same token, investigation into microkernel 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 [4,5]. Public domain libraries such as LAPACK and ScaLAPACK 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), 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 Level 1 and Level 2 BLAS [6].

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, which occupy 16 KB buffers in the Local Store. Between six and eight such buffers are necessary to efficiently implement common matrix operations. In general, it is reasonable to assume that half of the Local Store is devoted to application data buffers leaving only 128 KB for the application code, necessary libraries and the stack. Owing to that, the Local Store is a scarse resource and any *real-world* application is facing the problem of fitting tightly coupled components together in the limited space.

3 Related Work

Implementation of matrix multiplication $C = C + A \times B^T$ using Intel *Streaming SIMD Extensions* (SSE) was reported by Aberdeen and Baxter [7]. Analysis

of performance considerations of various computational kernels for the CELL processor, including the _GEMM kernel, was presented by Williams et al. [8]. The first implementation of the matrix multiplication kernel $C = A \times B$ for the CELL processor was reported by Chen et al. [9]. Performance of 25.01 Gflop/s was reported on a single SPE, with code size of roughly 32 KB. More recently assembly language implementation of the matrix multiplication $C = C - A \times B$ was reported by Hackenberg[10,11]. Performance of 25.40 Gflop/s was reported with code size close to 26 KB.

4 Implementation

4.1 SPU Architecture Overview

The core of the SPE is the *Synergistic Processing Unit* (SPU). The SPU is a RISC-style SIMD processor feturing 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.

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.

4.2 Loop Construction

The main tool in loop construction is the technique of loop unrolling. 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 interations.

In the canonical form, matrix multiplication $C_{m\times n} = A_{m\times k} \times B_{k\times n}$ coinsists of three nested loops iterating over the three dimensions m, n and k. Loop tiling 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 or splat 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 and shuffles, 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 preceeded 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.

4.3 $\mathbf{C = C - A \times B^T}$

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 being the only ones allowing to exploit short-vector capabilities of the SPEs through C language SIMD extensions.

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. 1 (left).

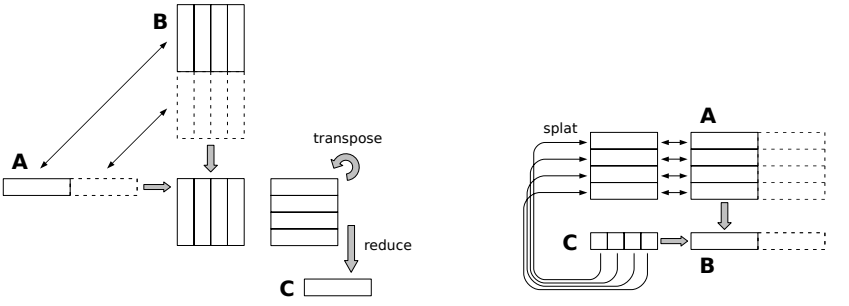

Fig. 1. Basic operation of the $C = C - A \times B^T$ micro-kernel (left). Basic operation of the $C = C - A \times B$ micro-kernel (right).

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 Sect. 4.1 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 Sect. 4.5 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 1 (left) shows how the number of each type of operation is calculated. Table 2 (left) shows the number of even pipeline, floating-point instructions including the reductions of partial results. Table 2 (center) shows the number of even pipeline instructions minus the number of odd pipeline instructions including loads, stores and shuffles (not including jumps and pointer arithmetic). In other words, Table 2 (center) shows the number of spare odd pipeline slots before jumps and pointer arithmetic are implemented. Finally, Table 2 (right) 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.

Table 1. Numbers of different types of operations in the computation of one tile of the $C = C - A \times B^T$ micro-kernel (left) and the $C = C - A \times B$ micro-kernel (right) as a function of tile size ({m, n, 64} triplet)

Type of Operation	Pipeline		Number of Operations
	Even	Odd	
Floating point	✘		(m × n × 64)/4 + m × n
Load A		✘	m × 64 /4
Load B		✘	64 × n /4
Load C		✘	m × n /4
Store C		✘	m × n /4
Shuffle		✘	m × n /4 × 8

Type of Operation	Pipeline		Number of Operations
	Even	Odd	
Floating point	✘		(m × n × k)/4
Load A		✘	m × k /4
Load B		✘	k × n /4
Load C		✘	m × n /4
Store C		✘	m × n /4
Splat		✘	m × k

Table 2. Unrolling analysis for the $C = C - A \times B^T$ micro-kernel: left - number of even pipeline, floating-point operations, center - number of spare odd pipeline slots, right - size of code for the computation of one tile

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	17408
32	2176	4352	8704	17408	34816
64	4352	8704	17408	34816	69632

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	13568
32	1280	3072	6656	13824	28160
64	2624	6272	13568	28160	57344

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

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.

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.

4.4 C = C - A × B

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. 1). 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 multiplies of four are allowed for n and k to facilitate efficient vectorization and simple loop construction. Table 1 (right) shows how the number of each type of operation is calculated. Table 3 (left) shows the number of even pipeline, floating-point instructions. Table 3 (center) 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 3 (center) shows the number of spare odd pipeline slots before jumps and pointer arithmetic are implemented. Finally, Table 3 (right) 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.

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.

Table 3. Unrolling analysis for the $C = C - A \times B$ micro-kernel: left - number of even pipeline, floating-point operations, center - number of spare odd pipeline slots, right - size of code for the computation of one tile

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	16384

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	12800

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

4.5 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). 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 Sect. 4.3 and Sect. 4.4, 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.

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. The size of the lookup table constructed in this manner equals $N^3/(m \times n \times k) \times 8$ bytes.

The last arithmetic operation remaining is the advancement of the itaration 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 annihilate the decrement, each quadword containing six offsets for one itaration of the double-buffered loop also contains a seventh entry, which stores the index of the quadword to be processed next (preceeding 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.

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.

5 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 $2N^3$. If used on the whole CELL processor with 8 SPEs, performance in excess of 200 Gflop/s should be expected. Table 4 shows the summary of the kernels' properties.

Table 4. Summary of the properties of the SPE SIMD SGEMM mikro-kernels

Characteristic	C=C-A×B^T	C=C-A×B
Performance	24.04 Gflop/s	25.55 Gflop/s
Execution time	21.80 μs	20.52 μs
Fraction of peak *USING THE 2×M×N×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

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/`.

6 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.

References

1. IBM Corporation: Cell Broadband Engine Programming Handbook, Version 1.1 (April 2007)
2. Geer, D.: Industry Trends: Chip Makers Turn to Multicore Processors. Computer 38(5), 11–13 (2005)
3. Asanovic, K., Bodik, R., Catanzaro, B.C., Gebis, J.J., Husbands, P., Keutzer, K., Patterson, D.A., Plishker, W.L., Shalf, J., Williams, S.W., Yelick, K.A.: The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences Department, University of California, Berkeley (2006)
4. Dongarra, J.J., Duff, I.S., Sorensen, D.C., van der Vorst, H.A.: Numerical Linear Algebra for High-Performance Computers. SIAM, Philadelphia (1998)
5. Demmel, J.W.: Applied Numerical Linear Algebra. SIAM, Philadelphia (1997)
6. Kågström, B., Ling, P., van Loan, C.: GEMM-Based Level 3 BLAS: High-Performance Model Implementations and Performance Evaluation Benchmark. ACM Trans. Math. Soft. 24(3), 268–302 (1998)
7. Aberdeen, D., Baxter, J.: Emmerald: A Fast Matrix-Matrix Multiply Using Intel's SSE Instructions. Concurrency Computat.: Pract. Exper. 13(2), 103–119 (2001)
8. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The Potential of the Cell Processor for Scientific Computing. In: ACM International Conference on Computing Frontiers (2006)
9. Chen, T., Raghavan, R., Dale, J., Iwata, E.: Cell Broadband Engine architecture and its first implementation, A performance view (November 2005), http://www-128.ibm.com/developerworks/power/library/pa-cellperf/
10. Hackenberg, D.: Einsatz und Leistungsanalyse der Cell Broadband Engine. Institut für Technische Informatik, Fakultät Informatik, Technische Universität Dresden, Großer Beleg (February 2007)
11. Hackenberg, D.: Fast matrix multiplication on CELL systems (July 2007), http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschun/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/