Adaptation of Double-Precision Matrix Multiplication to the Cell Broadband Engine Architecture

Krzysztof Rojek and Łukasz Szustak

Czestochowa University of Technology
Institute of Computer and Information Sciences
Dabrowskiego 73, 42-200 Czestochowa, Poland
{krojek,lszustak}@icis.pcz.pl
http://icis.pcz.pl

Abstract. This paper presents an approach to adaptation of the double-precision matrix multiplication to the architecture of Cell processors. The algorithm used for the adaptation on a single SPE is based on $C = C + A * B$ operation performed for matrices of size 64×64; these matrices are further divided into smaller submatrices which correspond to micro-kernel operations. Our approach is based on a performance model which is constructed as a function of submatrix size. The model accounts for such factors as size of local storage, number of registers, properties of double-precision operations, balance between pipelines, etc. This approach allows us to take into consideration properties of the first generation of Cell processors and its successor - PowerXCell 8i.

This adaptation is followed by an optimization phase which includes loop transformations, kernel implementation with SIMD instructions, and other transformations necessary to achieve balance between even and odd pipelines. Finally we present hand-tunings performed with the IBM Assembly Visualizer tool. The proposed adaptation and optimizations allow us to achieve about 96% of the peak performance.

1 Introduction

The Cell Broadband Engine architecture (CBEA) takes a radical departure from conventional multi-core architectures. The Cell Broadband Engine (shortly Cell/B.E) processor is the first implementation of CBEA, developed jointly by IBM, Sony, and Toshiba [1,2]. It was initially used in game consoles, but soon utilized for High Performance Computing as well. This technology we can find in the Sony PlayStation3, and IBM BladeCenter QS21. Each QS21 contains two Cell/B.E. processors.

The PowerXCell 8i processor is a new implementation of CBEA, with improved double-precision floating-point performance, and abilility to increase the main memory capacity up to 16 GB per processor. Each IBM BladeCenter QS22 contains two such processors.

R. Wyrzykowski et al. (Eds.): PPAM 2009, Part I, LNCS 6067, pp. 535–546, 2010.
© Springer-Verlag Berlin Heidelberg 2010

Both generations of Cell processors support a single-precision peak performance of 204.8 GFLOPS per single chip, which gives 25.6 GFLOPS per each of eight SPE (Synergistic Processing Element) cores. At the same time, in case of double-precision arithmetics the peak performance for the first generation of Cell processors is only 1.825 GFLOPS per single SPE, and 14.6 GFLOPS per all the eight SPEs available in the chip. The PowerXCell 8i offers seven times the double-precision performance of the previous Cell/B.E. processor, providing the peak performance of 12.8 GFLOPS per single SPE, and 102.4 GFLOPS per eight SPE cores.

The Cell family of processors provides outstanding opportunities for parallel computing. This allows developers to create high performance applications with the sustained performance close to the peak performance. However, utilizing the full potential of Cell processors is the big challenge for programmers [7,8]. For the matrix multiplication, which is a basic operation in many numerical algorithms, this challenge was investigated in [2,4,5,7,3].

For example, the kernel of the single-precision matrix multiplication algorithm performed on a single SPE was studied in [5]. The proposed algorithm and code optimizations allow the authors to achieve almost the peak performance. The results of implementing the matrix multiplication

$$C_{M\times N} = A_{M\times K} * B_{K\times N} \tag{1}$$

using 4-way SIMD multiply-add operations with 32-bit data were reported in [3], where the performance of 379 GFLOPS was achieved for 16 SPEs and the largest matrix size, as 92.5% of the peak performance delivered by QS21. The performance of 170.7 GFLOPS for the double-precision matrix-multiplication running on the IBM QS22 blade system with 32 GB of DDR2 SDRAM was achieved in [4].

In this paper, we propose an approach to adaptation of the double-precision matrix multiplication to the architecture of Cell processors. This approach is based on a performance model which is constructed as a function of submatrix size. The performance model allows for selecting "the best" size of a micro-kernel, which is used for the adaptation. This adaptation is followed by an optimization phase which includes loop transformations, kernel implementation with SIMD instructions, and other transformations necessary to achieve balance between even and odd pipelines. The proposed adaptation and optimizations allow us to achieve about 96% of the peak performance.

The paper is organized as follows. Section 2 introduces performance properties of the double-precision arithmetics in both generations of Cell processors. The idea of adaptation of the double-precision matrix multiplication to the Cell architecture is presented in Section 3, while Section 4 describes the micro-kernel used for this aim. The performance model for the double-precision matrix multiplication on Cell processors is derived in Section 5. Section 6 describes systematic implementation and optimizations steps carried out for the micro-kernel, while hand-tunings performed with the IBM Assembly Visualizer tool are illustrated in Section 7. Performance results are presented in Section 8, while Section 9 gives conclusions and future works.

2 Architecture of Cell Processors and Performance of Double-Precision Arithmetics

2.1 First Generation of Cell Processors

The Cell/B.E. processor consists of nine cores on a single chip, all connected to each other and to external devices by a high-bandwidth, memory-coherent bus. This processor has one PowerPC Processor Element (PPE) and eight SPE cores. The SPEs are SIMD processors optimized for data-rich operations allocated to them by the PPE. Each of these identical SPE cores contains a Synergistic Processing Unit (SPU) with a RISC core, 256-KB, software-controlled local store for instructions and data, and large register file with 128 registers, 128-bit each.

For the first generation of Cell processors, double-precision instructions are performed as two double precision operations in 2-way SIMD fashion, but the SPE is capable of performing only one double-precision operation per cycle. Double-precision instructions have 13 clock cycle latencies, but only the final seven cycles are pipelined. No other instructions are dual-issued with double-precision instructions, and no instructions of any kind are issued for six cycles after a double-precision instruction is issued. As a result, two consecutive instructions must be independent to eliminate dependency stalls (Fig. 1).

The SPU has two pipelines, named even (pipeline 0) and odd (pipeline 1). Into these pipelines, the SPU can issue and complete up to two instructions per cycle, one in each of the pipelines. Whether an instruction goes to the even or odd pipeline depends on its instruction type. The instructions responsible for data transfer (textttlqd, `stqd`), and for data organization (`shufb`) are assigned to the odd pipeline. Double-precision floating-point operations (`spu_madd`, `spu_mul`) are assigned to the even pipeline.

2.2 Second Generation of Cell Processors - PowerXCell 8i

The IBM PowerXCell 8i differs from the Cell/B.E. in that each SPE core includes an enhanced, double-precision unit. The second generation of Cell processors support the fully pipelined execution of double-precision instructions, with latency of 9 clock cycles. As a result, it is required to execute at least 9 consecutive, independent instructions to eliminate dependency stalls (Fig. 2).

```
000800 0 0123456789012                              dfma $92,$4,$17
000807 0  ------7890123456789                       dfma $94,$4,$18
000814 0         ------4567890123456                 dfma $92,$4,$19
000821 0                ------1234567890123         dfma $94,$4,$20
000828 0                       ------8901234567890  dfma $98,$4,$21
```

Fig. 1. Latencies of double-precision instructions in the first generation of Cell processors

```
000170 0          012345678                    dfma $75, $16,$47
000171 0           123456789                   dfma $78, $16,$49
000172 0            234567890                  dfma $92, $23,$33
000173 0             345678901                 dfma $94, $23,$36
000174 0              456789012                dfma $96, $23,$39
000175 0               567890123               dfma $98, $23,$43
000176 0                678901234              dfma $99, $23,$47
000177 0                 789012345             dfma $102,$23,$49
000178 0                  890123456            dfma $109,$16,$28
000179 0                   901234567           dfma $75, $20,$33
```

Fig. 2. Latencies of double-precision instructions in the second generation of Cell processors

3 Idea of Adaptation

Due to the limited resources of a SPE core, we base on a block version of the matrix multiplication when the original operation (1) is performed as a set of block matrix multiplications:

$$C_{MB\times NB} = A1_{MB\times KB} * B1_{KB\times NB},\ C_{MB\times NB} += A2_{MB\times KB} * B2_{KB\times NB}, \ldots \tag{2}$$

The block size of 64×64 was chosen, because:

- it is a multiple of maximum transfer size between local storage and main memory (16 KB);
- the amount of memory necessary to store five matrices ($A1$, $B1$, $A2$, $B2$, and C) must be less than the size of local storage (256 KB);
- it is the largest size that satisfies the above assumptions.

Our model assumes implementation of matrix multiplication using a single SPE core. Computations on huge matrices with high performance are strongly limited by the size of register file. Storing $64 * 64$ double-precision elements requires $64*64/2 = 2048$ registers. By that reason, performance model assumes partition of each matrix into smaller submatrices (Fig. 3).

The proposed adaptation assumes implementation of submatrix multiplications based on a micro-kernel (Fig. 3 and Fig. 4). But, to achieve a maximum

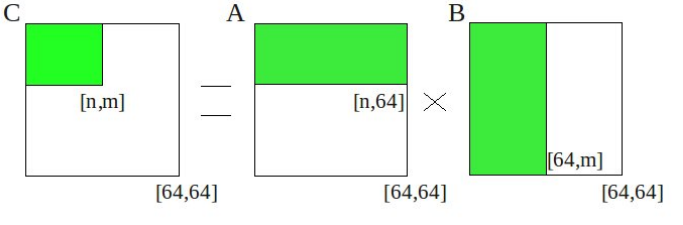

Fig. 3. Illustration of matrix multiplication micro-kernel

```
vector unsigned char pattern[2] = {{0, 1, 2, 3, 4, 5, 6, 7,
                                    0, 1, 2, 3, 4, 5, 6, 7},
                                   {8, 9, 10, 11, 12, 13, 14, 15,
                                    8, 9, 10, 11, 12, 13, 14, 15}};
(...)
for(i=0; i<n; ++i)
  a[i] = spu_shuffle(A[i*m+offsetA], A[i*m+offsetA], pattern[column%2]);

for(i=0; i<m/2; ++i) {
  for(j=0; j<n; ++j)
    c[j*m/2+i] = spu_madd(a[j], B[i+offsetB], c[j*m/2+i]);
}
```

Fig. 4. Basic code of micro-kernel

performance we need to find "the best" values of n and m. To find optimal sizes of submatrices, a performance model is derived.

4 Micro-Kernel of Matrix Multiplication

Fig. 4 shows the basic SIMD implementation of the micro-kernel matrix multiplication [5]. The main task of the micro-kernel is to copy each element of a column of A-submatrix into all two slots of a register that requires `spu_shuffle` instructions, multiply it by a row of B-submatrix, and add to the output C-submatrix. But, to compute all elements of C-submatrix we need to repeat the kernel for all columns of A-submatrix and rows of B-submatrix. We assume that each element of C-submatrix, each column of A-submatrix, and each row of B-submatrix are stored in the register file. The offsets `offsetA` and `offsetB` point out to base addresses of A- anb B-submatrices, respectively.

5 Performance Model for Double-Precision Matrix Multiplication on Cell Processors

5.1 Model Assumptions

Our performance model is designed for a single SPE core. It is formulated for both of the first generation of Cell processors, and PowerXCell 8i architecture. The main challenge of the model is to find "the best" values for the sizes n and m of submatrices involved in the micro-kernel (Fig. 3 and Fig. 4), where $n \leq 64$, $m \leq 64$. Since these values must be a divisor of 64, they belong to the following set of numbers: $1, 2, 4, 8, 16, 32, 64$.

The input parameters for this model are as follows:

1. number of available SPE registers;
2. number of independent double-precision operations.

Also, let us enumerate the assumptions of the model:

1. the number of used registers can not exceed 128;
2. the number of operations on the even-pipeline can not be less than the number of odd-pipeline operations;
3. the number of odd-pipeline operations should be minimized;
4. for the first generation of Cell processors, the ratio between even- and odd-pipeline operations should be maximized.

This model does not take into consideration store operations performed in direction from the register file to the local storage. The number of these operations is constant, and independent from the sizes of submatrices.

5.2 Performance Model

In accordance with our assumptions, each element of A-submatrix is stored in one register. This register contains two copies of one double-precision element. In our algorithm, a single column of A-submatrix is stored in registers. So, at least n registers are required to store A-submatrix. We need two more registers for `spu_shuffle` instructions. These instructions are responsible for distributing (copying) one element across a vector. In addition, the algorithm holds one row of B-submatrix in $m/2$ additional registers. Also, we assume that all elements of C-submatrix are stored in $n * m/2$ registers. So, at least $n + m/2 + n * m/2 + 2$ registers are required to perform the micro-kernel.

Our algorithm executes $n*m/2$ even-pipeline operations for each C-submatrix. The whole algorithm requires to execute $64 * 64 * 64/2 = 131072$ vectorized operations on the even pipeline. So, each submatrix multiplication must be repeated $131072/(n * m/2)$ times.

Odd-pipeline operations are responsible for data transfers between the local storage and register file, as well as for `spu_shuffle` instructions. A single submatrix of size $n \times m$ requires loading m elements for B-matrix ($m/2$ instructions), and n elements for A-matrix (n instructions). In addition, our algorithm executes n `spu_shuffle` operations (one for each element of A-subbmatrix). So, the total number of odd-pipeline operations is at least $m/2 + 2 * n$.

For the first generation of Cell processors, our model assumes $m/2$ consecutive, independent operations. So, to eliminate stalls, $m/2 \geq 2$ independent operations should be performed, that gives $m \geq 4$. As a result, we have $m \in \{4, 8, 16, 32, 64\}$.

Table 1 presents: (i) the number of required registers; (ii) the numbers of even and odd pipelines operations for a single submatrix multiplication, and for the whole matrix multiplication of size 64×64; (iii) the ratio between the number of even- and odd-pipeline instructions. All these parameters are given as a function of submatrix sizes (m and n). Because of space constraints, the rows corresponding to $m = 4, 64$ are omitted in Table 1.

For the whole algorithm, the number of operations executed on the even pipeline is constant. However, the number of operations executed on the odd pipeline depends on selected sizes of submatrices. For the first generation of Cell

Table 1. Performance model for the first generation of Cell processors

n	m	registers	even	odd	all odd	all even	ratio	$reg. < 128$
2	8	16	8	8	131072	131072	1	TRUE
4	8	26	16	12	98304	131072	1,33	TRUE
8	8	46	32	20	81920	131072	1,6	TRUE
16	8	86	64	36	73728	131072	1,78	TRUE
32	8	166	128	68	69632	131072	1,88	FALSE
64	8	326	256	132	67584	131072	1,94	FALSE
2	16	28	16	12	98304	131072	1,33	TRUE
4	16	46	32	16	65536	131072	2	TRUE
8	16	82	64	24	49152	131072	2,67	TRUE
16	16	154	128	40	40960	131072	3,2	FALSE
32	16	298	256	72	36864	131072	3,56	FALSE
64	16	586	512	136	34816	131072	3,76	FALSE
2	32	52	32	20	83968	131072	1,56	TRUE
4	**32**	**86**	**64**	**24**	**49152**	**131072**	**2,67**	**TRUE**
8	32	154	128	32	32768	131072	4	FALSE
16	32	290	256	48	24576	131072	5,33	FALSE
32	32	562	512	80	20480	131072	6,4	FALSE
64	32	1106	1024	144	18432	131072	7,11	FALSE

processors, the fewer number of odd-pipeline operations the better performance can be achieved.

In our performance model, the register usage depends on chosen values of m and n. Moreover, we can accept only those values of m and n for which the number of required registers is less than 128. From the set of possible solutions, we select that with the biggest ratio between even- and odd-pipeline operations. So, we choose finally $n = 4$ and $m = 32$.

To eliminate stalls for the second generation of Cell processors represented by PowerXCell 8i, it is required to execute $m/2 \geq 9$ independent, consecutive operations, that gives $m \geq 18$. As a result, we have $m \in \{32, 64\}$.

Table 2 presents results given by the performance model for the selected values of m and n, in case of the PowerXCell 8i. This table includes: (i) the number of required registers; (ii) the number of even- and odd-pipeline operations for a single submatrix multiplication, and for the whole matrix multiplication of size 64×64.

To eliminate additional stalls, the number of odd-pipeline operations must not exceed the number of even-pipeline operations. Taking this condition into account, our model indicates three potential solutions for values of n and m. Among them, we choose a solution with the least number of operations on the odd pipeline. Like the first generation of Cell processors, again we have $n = 4$, $m = 32$.

Table 2. Performance model for PowerXCell 8i

n	m	reg.	even	odd	all odd	all even	*Reg.* < 128	*even* >= *odd*	
2	32	52	32	20	83968	131072	TRUE	TRUE	TRUE
4	**32**	**86**	**64**	**24**	**49152**	**131072**	**TRUE**	**TRUE**	**TRUE**
8	32	154	128	32	32768	131072	FALSE	TRUE	FALSE
16	32	290	256	48	24576	131072	FALSE	TRUE	FALSE
32	32	562	512	80	20480	131072	FALSE	TRUE	FALSE
64	32	1106	1024	144	18432	131072	FALSE	TRUE	FALSE
2	64	100	64	36	73728	131072	TRUE	TRUE	TRUE
4	64	166	128	40	40960	131072	FALSE	TRUE	FALSE
8	64	298	256	48	24576	131072	FALSE	TRUE	FALSE
16	64	562	512	64	16384	131072	FALSE	TRUE	FALSE
32	64	1090	1024	96	12288	131072	FALSE	TRUE	FALSE
64	64	2146	2048	160	10240	131072	FALSE	TRUE	FALSE

6 Systematic Implementation and Optimization Steps

The canonical algorithm for computing the matrix multiplication (1) is based on three nested loops [5]. Each loop iterates over a certain dimension - m, n, or k. When adapting this algorithm to the Cell/B.E. architecture, the main challenge is a suitable partitioning of a matrix into smaller submatrices.

The technique used for the matrix partitioning is known as a *loop tiling*. It creates three outer loops, which calculate addresses of submatrices, and three 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. In this way, the register reuse is maximized, and the number of loads and stores is minimized. In our approach, the sizes of submatrices are chosen based on the performance model.

The next technique applied by us is *loop unrolling*. It is a loop transformation technique, that attempts to optimize the execution speed of a program at the expense of its size. Loops can be re-written as a sequence of independent statements, which eliminates the loop-control overhead. In our implementation, this is applied for the three most nested loops. The *loop unrolling* allows for eliminating dependency stalls, as well as decreasing the amount of branch and increment instructions. On SPEs cores, this technique allows us to achieve a better balance between pipelines, and maximize the dual-issue rate.

The next challenge is to eliminate stalls caused by latencies of access to the local storage. For each iterations, the algorithm executes the following sequence of operations:

- loading data to the register file;
- submatrix multiplication (textttspu_shuffle and `spu_madd` operations),
- storing data to the local storage.

The optimization technique used to solve this problem is *double-buffering*, applied on the register level and involving two loop iterations [5]. The existing loop

body is duplicated, and two separate C-submatrices take care of the even and odd iterations, respectively.

The algorithm presented in Fig. 4 requires some additional operations to compute addresses of elements of submatrices. Time required to compute these addresses has a significant influence on the performance of the whole algorithm. Computing indices of each elements requires to use even-pipeline instructions, that can not be hidden behind `spu_madd` instructions. In fact, a part of operations on addresses, like multiplication by a power of 2, are computed using *shift* operations, such as `shli` shown in Fig. 5a. However, by changing `shli` operations on `shlqbii` ones (Fig. 5b), which are executed on the odd pipeline, we achieve that a part of address operations can be hidden behind even-pipeline instructions.

```
(a) Before optimization:
000021 0     12        ori $94,$5,0
000022 0      2345     shli $86,$58,0
000023 0       34      ai $sp,$sp,-304
000024 0        4567   shli $44,$58,0
000025 0         56    ori $85,$58,0
000026 0          6789 shli $43,$58,0

(b) After optimization:
000021 0D    12         ori $94,$5,0
000021 1D    1234       shlqbyi $86,$58,0
000022 0D     23        ai $sp,$sp,-304
000022 1D     2345      shlqbyi $44,$58,0
000023 0D      34       ori $85,$58,0
000023 1D      3456     shlqbyi $43,$58,0
```

Fig. 5. Results of `spu_timing` analysis before (a) and after (b) optimization

7 Hand-Tuning with the IBM Assembly Visualizer Tool

The `IBM Assembly Visualizer` tool [9] annotates an assembly source file with static analysis of instruction timing, assuming the linear (branchless) execution. Such an analysis does not account for *branch taken* instructions caused, for example, by loops. The Assembly Visualizer allows the user to safely and interactively edit the assembly in a manner that does not violate the correctness of the original program. The compilation to an assembly code can be done by the `spuxlc` or `spu-gcc` compiler with `-S` flag. The user can then hand-tune and fix pipeline stalls that the compiler has missed. This powerful functionality enables a level of performance optimization that is not possible at the C/C++ level.

Summarizing, the static timing analysis of assembly codes:

- is based on dual-issue rules, and static dependency stalls of instructions;
- accounts for *branch not taken* instructions;
- does not account for local store transfers, and branching.

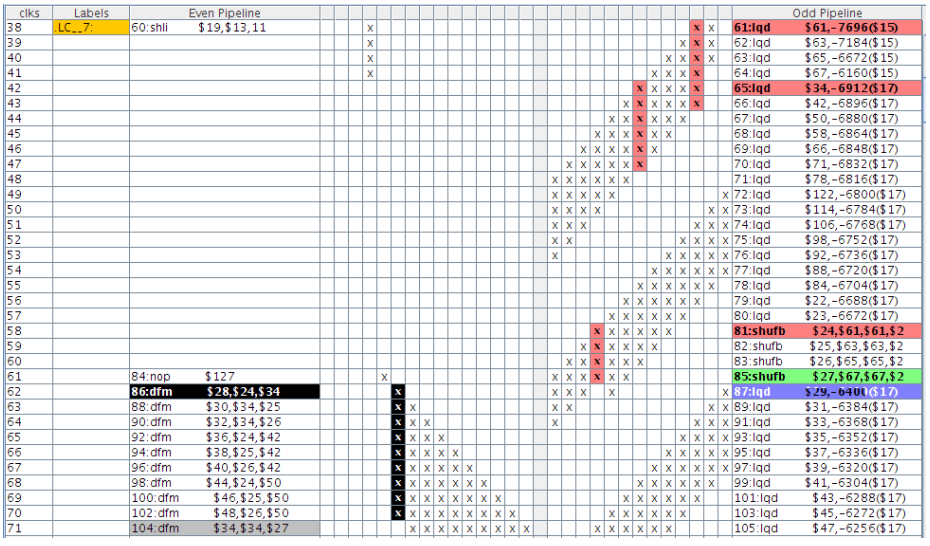

Fig. 6. Timing analysis before hand-tuning

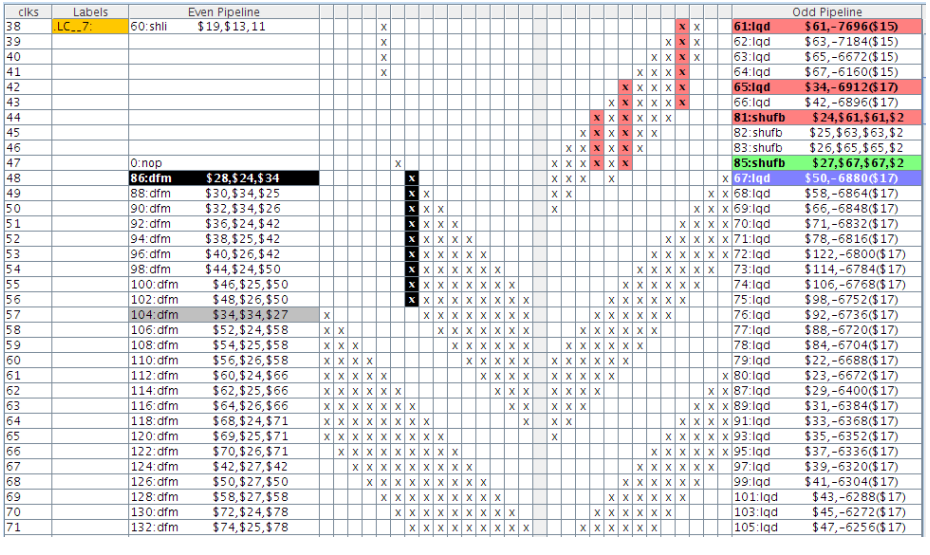

Fig. 7. Timing analysis after code hand-tuning with the IBM Assembly Visualizer

Fig. 6 shows the static timing analysis of the SPE code before its hand-tuning, while Fig. 7 demonstrates the result of the code hand-tuning with the IBM Assembly Visualizer tool.

8 Performance Results

For the first generation of Cell processors, the basic implementation of our matrix multiplication algorithm allows us to achieve 35.53% of the peak performance. After including optimizations such as loop reorganizations (loop tiling and loop unrolling), but without double buffering, the performance increases to 94.63%.

For the PowerXCell 8i architecture, the basic implementation allows for achieving 39.2% of the peak performance. The loop reorganizations increase the performance to 86.41%. The next optimization, which is the double buffering technique used on the register level, improves the performance up to 95.14%. Finally, optimizations included on the assembly level with the IBM Assembly Visualizer tool allows for achieving 96.05% of the peak performance.

9 Conclusions and Future Works

This paper presents the approach to adaptation of the double-precision matrix multiplication to the architecture of both generations of Cell processors. The algorithm used for the adaptation on a single SPE core is based on $C = C + A * B$ operation performed for matrices of size 64×64; these matrices are further divided into smaller submatrices which correspond to micro-kernel operations. Our approach is based on the performance model which accounts for such factors as size of local storage, number of registers, properties of double-precision operations, balance between pipelines, etc. The proposed approach allows us to take into consideration properties of the first generation of Cell processors, and its successor - PowerXCell 8i.

This adaptation is followed by an optimization phase which includes loop transformations, kernel implementation with SIMD instructions, and other transformations necessary to achieve balance between even and odd pipelines. Finally we present hand-tunings performed with the IBM Assembly Visualizer tool. The proposed adaptation and optimizations allow us to achieve 94.63% and 96.05% of the peak performance for the first and second generations of Cell processors, respectively.

The performance model derived in this work allows for selecting "the best" size of the micro-kernel, which is used for the adaptation. We have investigated other solutions extracted from the performance model, however, performance results for these solutions are worse by about 10

This paper focuses on multiplying matrices of size 64×64 on a single SPE. In future works the proposed approach will be extended for multiplying large matrices on all 16 SPEs available in both of the QS21 and QS22 blade centers.

References

1. Buttari, A., Dongarra, J., Kurzak, J.: Limitations of the PlayStation3 for High Performance Cluster Computing, http://www.netlib.org/lapack/lawnspdf/lawn185.pdf
2. Chen, T., Raghavan, R., Dale, J.N., Iwata, E.: Cell Broadband Engine Architecture and its first implementation - A performance view. IBM Journal of Research and Development 51(5), 559–572 (2007)
3. Dolfen, A., Gutheil, I., Homberg, W., Koch, E.: Applications on Juelich's Cell-based Cluster JUICE, http://www.fz-juelich.de/jsc/datapool/cell/Para08_apps_on_juice.pdf
4. Kistler, M., Gunnels, J., Brokenshire, D., Benton, B.: Programming the Linpack Benchmark for the IBM PowerXCell 8i Processor. Scientific Programming 17(1-2), 43–57 (2009)
5. Kurzak, J., Alvaro, W., Dongarra, J.: Optimizing matrix multiplication for a short-vector SIMD architecture - CELL processor. Parallel Computing 35(3), 138–150 (2009)
6. Wang, H., Takizawa, H., Kobayashi, H.: A Performance Study of Secure Data Mining on the Cell Processor. Int. Journal of Grid and High Performance Computing 1(2), 30–44 (2009)
7. Williams, S., Shalf, J., Oliker, L., Kamil, S., Husbands, P., Yelick, K.: The potential of the Cell Processor for Scientific Computing. In: Proc. 3rd Conf. on Computing Frontiers, Ischia, Italy, pp. 9–20 (2006)
8. Woodward, P.R., Jayaraj, J., Lin, P., Yew, P.: Moving Scientific Codes to Multicore Microprocessor CPUs. Computing in Science and Engineering 10(6), 16–25 (2008)
9. IBM Assembly Visualizer for Cell Broadband Engine, http://www.alphaworks.ibm.com/tech/asmvis