Comparison of CellSs and native programming with a Jacobi solver and triple-matrix-multiply on Cell/B.E.

Liang Yang[1,3], Annika Schiller[1], Godehard Sutmann[1], Brian J. N. Wylie[1]
Ralph Altenfeld[1], and Felix Wolf[1,2]

[1] Jülich Supercomputing Centre, Forschungszentrum Jülich GmbH, Germany
[2] Department of Computer Science, RWTH Aachen University, Germany
[3] Xi'an Microelectronics Technology Institute, China
yungicn@gmail.com
{a.schiller,b.wylie,g.sutmann,r.altenfeld,f.wolf}@fz-juelich.de

Abstract. The Cell Superscalar framework (CellSs), from Barcelona Supercomputing Centre, offers a high-level portable programming model to port, parallelise and tune applications on Cell Broadband Engine. Using the native programming API of the IBM Software Development Kit and CellSs, ease of programming and resulting performance are assessed for a Jacobi solver and a sparse triple-matrix-multiply (TMM).
CellSs is found to be a convenient and efficient tool for achieving parallelisation. The relative simplicity of the CellSs programming model and its efficient automatic implementation of task scheduling and internal data management greatly facilitated development of a novel TMM algorithm with more efficient memory usage, which was necessary to realise the application on Cell/B.E. within the limited SPE memory.

Key words: Programming models & tool environments; application parallelisation & optimisation; performance analysis & evaluation.

1 Introduction

The high potential performance of the Cell Broadband Engine (Cell/B.E.) derives from a broad spectrum of capabilities. Processor characteristics include multiple heterogeneous execution units, SIMD processing engines, fast local store and a software-managed cache. Applications can achieve nearly maximum theoretical performance if specific features are respected [1]. Exploiting the full potential of Cell/B.E., however, is challenging for programmers who are trying to port their applications. Since the memory of the PPE is limited to 2GB and SPE local store is limited to 256kB [6], the amount of data is an important aspect. Parallel tasks are initiated by calling special functions supplied in the IBM Software Development Kit (CellSDK) SPE runtime management library and data transfers to and from the limited local memory of the SPEs have to be managed explicitly via DMA function calls. This means that to port an application to Cell/B.E. a custom parallel program has to be written which can

only run on this specific platform. To facilitate the porting of native code to Cell/B.E., Barcelona Supercomputing Centre is developing CellSs, which proposes a portable programming model for multi-core processors [2, 3].

In this paper a Jacobi solver and the triple-matrix-multiply kernel from a wavelet-based evaluation of Coulomb potentials in molecular systems are ported to Cell/B.E. via CellSs, and results are compared to implementations based on CellSDK native programming API.

2 Programming with CellSs

The Cell Superscalar framework (CellSs) offers a high-level portable programming model to port, parallelize and tune applications on Cell/B.E. In contrast to the implementation using the native programming API, CellSs does not require a complete rewriting of the application. The existing application source code can simply be ported by inserting annotations (pragmas). There are three types of pragmas:

1. Initialization and finalization pragma:
   ```
   #pragma css start
   #pragma css finish
   ```
2. Task specification pragma:
   ```
   #pragma css task [input (<input parameters>)] opt
                    [inout (<inout parameters>)] opt
                    [output (<output parameters>)] opt
                    [highpriority] opt
   ```
3. Synchronization pragma:
   ```
   #pragma css wait on (<list of expressions>)
   ```

The initialization and finalization pragmas indicate the part of the program that will be interpreted by CellSs. If these pragmas are missing they will automatically be inserted by the compiler at the beginning and end of the program. The task specification pragmas are inserted in front of those functions which are intended to be run on the SPEs. The attached parameter list specifies the data that has to be transferred between PPE and SPEs and their direction. CellSs requires that the application is composed of coarse grain functions and that these functions do not have collateral effects (i.e., only local variables and parameters are accessed). With CellSs, the task annotation before a coarse grain function (task) does not indicate a parallel region like OpenMP, but simply indicates the direction of the parameters of this function (input/inout/output). The CellSs runtime system builds a data dependency graph by collecting the information about these parameters and schedules independent tasks to different SPEs concurrently. All data transfers required for computations on the SPEs are also handled automatically. The synchronization pragma is necessary to wait on the data that has to be transferred for the annotated task. The following extract

of the CellSs code for scheme III of the triple-matrix-multiply (see section 4) clarifies how the pragmas are applied.

```
#pragma css task input (wblock, A, col_idx) output (B)
void SPE_B (float wblock[BSIZE], float A[N], float B[N], col_idx)
{...}
#pragma css task input (wblock, A, col_idx) output (B)
void SPE_C (float B[N], float wblock[BSIZE], rows, float C[rows])
{...}
int main() {
   ...
   for (blk_j=0; blk_j<w_blocks; blk_j++)
      for (j=0; j<rows_of_this_block[blk_j]; j++){
         SPE_B(W_BLOCK[blk_j], A, Col_B, col_index_of_B);
         for (blk_i=0; blk_i<w_blocks; blk_i++)
            SPE_C(Col_B, W_BLOCK[blk_i], rows_of_this_block[blk_i],
                  C[col_idx_of_B][blk_i]);
      }
   for (i=0; i<N; i++)
      for (j=0; j<w_blocks; j++){
#pragma css wait on (C[i][j])
      }
   ...
}
```

From the pragma equipped application code the source to source compiler generates two source files: one to be run on PPE and another one to be executed on SPEs. These two files are then compiled by the native compiler with the desired optimizations such as automatic vectorisation applied. CellSs at this point only can handle a single source file. Another restriction is that only program parts written in C can be interpreted by the CellSs compiler. That means that for applications written in other programming languages, the parts which are to be ported to Cell/B.E. have to be rewritten in C and equipped with CellSs pragmas, and interfaced to the remainder that remains in the original language. The different source modules can then be compiled separately and linked together to generate the final object file.

Although CellSs handles all data transfers automatically, the programmer should nevertheless carefully calculate if the data will fit into the SPEs' local memory. Alignment requirements of Cell/B.E. also have to be considered: every data object for DMA transfer has to be aligned to 16 byte addresses, and performance is improved when aligned to 128 bytes.

Furthermore, CellSs includes an execution trace generation capability so that the programmer can examine the time for DMA transfers, SPE task computations, waiting on data, user code on the PPE, etc., to find application imbalances and inefficiencies.

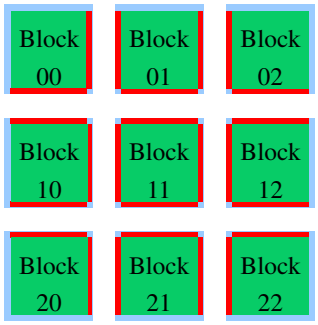

Fig. 1. Schematic of the blocking strategy of the Jacobi solver scheme I. The ghost planes (red) are integrated in the blocks of kernel data (green).

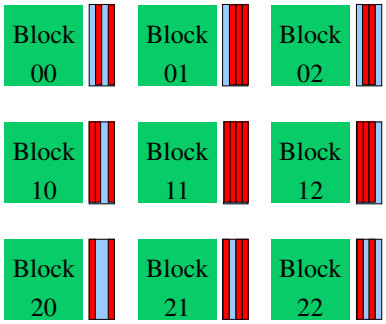

Fig. 2. Schematic of the blocking strategy of Jacobi solver schemes II and III. Ghost planes are separated from kernel data by packing them into an extra array.

3 3D Jacobi solver

The Jacobi solver computes the solution for systems of linear equations, which are used in a wide range of scientific applications. For a 3-dimensional problem, the Jacobi solver basically updates each data point based on its six neighbours. Equation 1 shows the update scheme considered in this work, where $A_{i,j,k}$ are the field values, $f_{i,j,k}$ the given source term and α is constant.

$$A^{n+1}_{i,j,k} = \alpha(f_{i,j,k}+A^n_{i-1,j,k}+A^n_{i+1,j,k}+A^n_{i,j-1,k}+A^n_{i,j+1,k}+A^n_{i,j,k-1}+A^n_{i,j,k+1}) \quad (1)$$

Due to the limited size of the SPE local memory, the whole matrix must be partitioned into smaller blocks. Blocking is also necessary to distribute the work to multiple SPEs. The problem with blocking and distribution is that updating the points which are located at the edge of one block needs the information of points which belong to another block, that are on another SPE. Therefore, the points which are needed to update the kernel data but do not actually belong to the block are also stored. These points are called ghost points and they have to be refreshed after every iteration, which requires data transfer.

Three different schemes have been investigated applying different blocking strategies for matrix A and different updating methods for the ghost points.

3.1 Algorithm description

Scheme I. In the first scheme the matrix A is partitioned into cubic blocks as displayed in Figure 1. These blocks are not disjoint because the kernel data is extended by the ghost points, so that the blocks overlap at the ghost planes. For each iteration the blocks are then transferred to the SPEs and the kernel points are updated. After that the blocks are transferred back to the PPE so that the

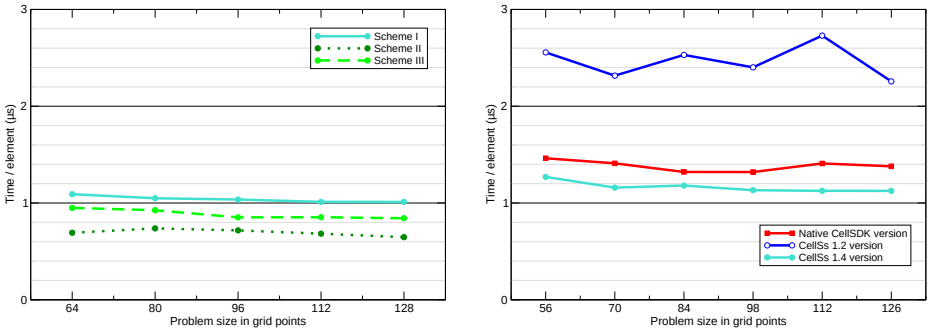

Fig. 3. Comparison of execution time between the three different schemes of the Jacobi solver implementation (left) and between native CellSDK and CellSs implementations of the Jacobi solver (right).

ghost points can be refreshed, and this procedure is repeated for every iteration. This scheme requires several DMA transfers for every iteration (see Table 3.1). The computational speed of this scheme is therefore limited by the speed of the DMA transfers, i.e., by the speed of the bus.

Scheme II. With the second scheme the updating method is kept but the blocking strategy is changed (see Figure 2). The ghost points are now separated from the kernel data by packing them into an extra array. For the side blocks the correspondig places for ghost planes which are missing for this block are filled with zeros. With this modification the ghost points can be transferred to the SPE independently for every iteration.

Table 1. Comparison of the complexities of the three schemes of the Jacobi solver, where M is the problem size, N the block size and Nt the number of operations

	Operations	Transferred data	DMA transfers
Scheme I	$(\lceil M/(N-2)\rceil)^3$	$Nt \cdot (2N^3 + (N-2)^3)$	$3Nt$
Scheme II	$(\lceil M/N\rceil)^3$	$Nt \cdot (3N^3 + 6N^2)$	$4Nt$
Scheme III	$(\lceil M/N\rceil)^3$	$Nt \cdot (3N^3 + 6N^2)$	$9Nt$

Scheme III. The blocking strategy of the third scheme is the same as the second, but modifies the method of updating ghost points. Examining a Paraver visualization of the runtime behaviour of schemes I and II showed that updating ghost points is rather time consuming. Consequently, the exchange of data is replaced by only exchanging pointers to the data. While the time for updating ghost planes then decreases, the shortcoming is that time for DMA transfer increases because the ghost planes belonging to one block cannot always be contiguously stored in memory and therefore require a larger number of transfers.

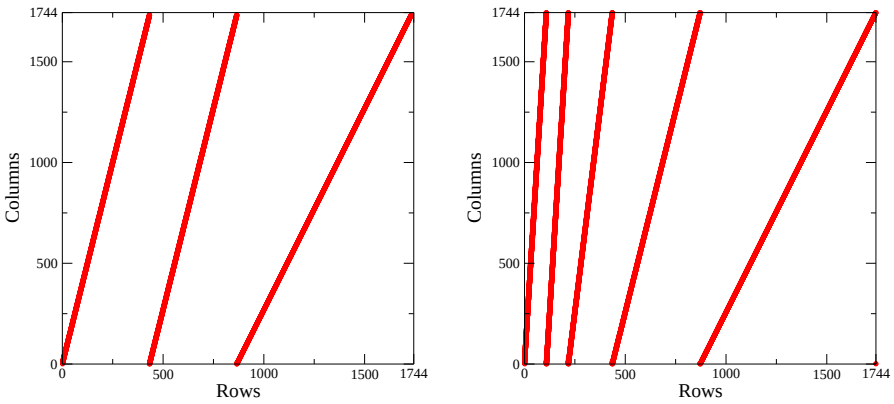

Fig. 4. Structure of the Wavelet matrix $\mathcal{W}$ (Haar Wavelet) for two different levels l of refinement: $l = 2$ (left) and $l = 4$ (right)

3.2 Results

The best speedup was found when five SPEs are used concurrently. Comparing the three schemes according to their execution time as done in the left of Figure 3 show that scheme II is the most efficient. It is about 30% faster than scheme I. In the right diagram of Figure 3 the performance of version 1.2 and 1.4 of CellSs is compared to the performance of the native implementation. It is visible that the implementation with CellSs version 1.2 is considerably slower than the native implementation, however, the CellSs code produced with version 1.4 significantly outperforms the comparable native implementation with CellSDK.

4 Triple-matrix-multiply

Triple-matrix-multiply (TMM) is one part in the kernel of a program which is used to perform a wavelet-based evaluation of Coulomb potentials in molecular systems [4]. The calculation of long-range interactions in molecular systems is computationally very demanding. Since all interactions between particle pairs have to be considered, the complexity scales like $\mathcal{O}(N^2)$ and methods have been developed to reduce the complexity to $\mathcal{O}(NlogN)$ or even $\mathcal{O}(N)$ [5].

In the present work a mesh-based method is considered which uses a Wavelet transform technique to calculate Coulomb interactions. The method's kernel is the calculation of the Wavelet transform of the inverse distance matrix A, which is realized via the following triple-matrix-multiply:

$$\tilde{A} \;= \mathcal{W}\, A\, \mathcal{W}^T \;\;\Rightarrow\;\; \tilde{A}_{ij} = \sum_{k,l} \mathcal{W}_{ik} \cdot A_{kl} \cdot \mathcal{W}_{jl} \tag{2}$$

The transform matrix $\mathcal{W}$ is sparse and has a banded structure (Figure 4), which favours a Compressed Sparse Row (CSR) format for data storage. The inverse

distance matrix A is dense and symmetric with dimension $N \times N$, where $N = np \cdot np \cdot np$ is the number of grid points and np the number of grid points in one Cartesian direction. Elements of matrix A can be calculated as follows:

$$A_{ij} = \begin{cases} \dfrac{1}{|r_i - r_j|} = \dfrac{1}{a \cdot \sqrt{(ix - jx)^2 + (iy - jy)^2 + (iz - jz)^2}} & , \ i \neq j \\ 0 & , \ i = j \end{cases} \tag{3}$$

where

$$ix = i \ \% \ np \qquad iy = (i \ \%(np \cdot np))/np \qquad iz = i/(np \cdot np) \tag{4}$$

In Eq. 3, a is a the grid spacing, i and j are the indices of the elements in A and (ix, iy, iz) are the corresponding index tuples of matrix index i in the three-dimensional $np \cdot np \cdot np$ grid, which can be calculated via Eq. 4.

4.1 Algorithm description

Due to the Wavelet matrix $\mathcal{W}$, the algorithm is based on sparse linear algebra operations. Three different schemes were investigated, which consider different architectural requirements.

Scheme I. The first approach considers the storage requirement for matrix A which can become very large. To minimize storage space and DMA transfers, the elements of matrix A can be calculated on the SPE when they are needed (using Eq. 3). This approach is efficient in its memory usage but computationally inefficient. Measurements for an application with $np = 12$ showed that more than half of the time is spent on the computation of the elements of A and on average each element is calculated more than 80 times.

Suppose that the average number of elements in one line of $\mathcal{W}$ is about N_w. For one element of the result matrix $\tilde{A}$, there are about N_w^2 values of matrix A which have to be calculated. Additionally $2N_w^2$ multiplications and $N_w^2 - 1$ additions are necessary to calculate each element of the result matrix $\tilde{A}$. For the calculation of the whole result matrix $N^2 \cdot N_w^2$ values of A have to be calculated and $N^2 \cdot 2N_w^2$ multiplications and $N^2 \cdot (N_w^2 - 1)$ additions are necessary.

Scheme II. To avoid redundant re-calculation of elements of matrix A, a second approach was to calculate the complete matrix A only once on the PPE and load it line by line into the SPEs. Furthermore, to reduce the number of operations the TMM is now performed in two steps as follows [7]:

$$\tilde{A} \ = \mathcal{W} \ A \ \mathcal{W}^T \ \Rightarrow \ \tilde{a}_q = \mathcal{W} \times (A \times w_q^T) \tag{5}$$

In this equation $\tilde{a}_q$ is the q-th column of the result matrix $\tilde{A}$, and w_q^T is the q-th column of the transposed Wavelet matrix $\mathcal{W}^T$.

For this approach the number of operations is reduced. Calculating the whole result matrix N^2, values for matrix A have to be generated and only $N^2 \cdot 2N_w$ multiplications and $N^2 \cdot (N_w - 1)$ additions are necessary. The shortcoming of this scheme is that it needs so much memory that it can only be applied for problems up to $np = 18$.

Algorithm 1 Scheme III of TMM

```
1: calculate the kernel of A on the PPU and load it to the SPUs
2: for blk_i in w_blocks do                          ▷ loop over all blocks of W
3:     load block blk_i of W to SPU
4:     for i in rows of blk_i do                     ▷ loop over all rows of block i of W
5:         for k in N do
6:             tmp = 0
7:             for l in number of elements in line i of W do
8:                 get value W_il
9:                 map k and l to the three dimensional grid via Eq. 4
10:                get the corresponding A_kl in the kernel of A
11:                calculate tmp = tmp + A_kl · W_jl
12:            end for
13:            B_k = tmp
14:        end for
15:        for blk_j in w_blocks do                  ▷ loop over all rows of block j of W_l
16:            load block blk_j of W to SPU
17:            for j in rows of blk_j do
18:                tmp = 0
19:                for k in number of elements in line j of W_l do
20:                    get value W_jk
21:                    calculate tmp = tmp + W_jk · B_k
22:                end for
23:                Ã_ji = tmp
24:            end for
25:        end for
26:    end for
27: end for
```

Scheme III. To decrease memory usage and redundant computation, a novel approach which is efficient on both aspects was developed. It exploits the fact that the three-dimensional grid of the simulated system has to be mapped onto the two-dimensional matrix A. This matrix contains the inverse distances of every pair of grid points in the three dimensional grid. If two pairs of grid points have the same distance, their values in A calculated via equation 3 are identical. Since many pairs of grid points have the same distance, the matrix A contains a lot of redundant information. That means that it is enough to store only $\mathcal{O}(N)$ values for the kernel of A instead of $\mathcal{O}(N^2)$ values for the whole matrix A. With other words only one row of matrix A has to be stored which still has

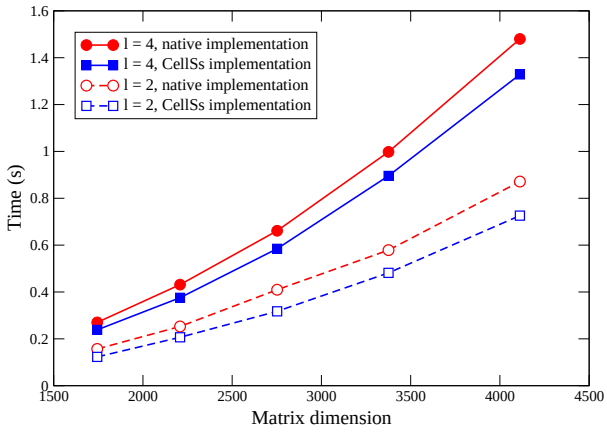

Fig. 5. Execution time comparison between native CellSDK and CellSs implementations of the Wavelet transform with level $l = 2$ and $l = 4$ for $\mathcal{W}$ (Haar Wavelet)

little redundancy. The correct distance can be calculated by mapping the two-dimensional addressing of A back onto the three-dimensional addressing of the system's grid as shown in equation 4.

In this scheme the TMM is still calculated in two steps as shown in Algorithm 1. The non-redundant part of matrix A is calculated on the PPE and is loaded into the SPEs local memory at the beginning of the calculation. The Wavelet matrix $\mathcal{W}$ is blocked and loaded blockwise onto the SPEs. The result matrix is then calculated in two steps column by column, whereas the intermediate result B_k is buffered on the SPE and not transferred back to the PPE.

4.2 Results

The third approach turned out to be the most efficient one. In Figure 5, the execution time for TMM implemented via CellSs version 1.4 for different matrix dimensions and Wavelet transforms is compared to the implementation using CellSDK. It is apparent that the behaviour of the two implementations for increasing matrix dimensions is nearly the same. The CellSs code, however, is consistently 10–20% faster than the native CellSDK version. The reason for the difference is that double buffering is not yet implemented in the native version while it is automatically applied by CellSs.

5 Conclusions and future work

A Jacobi solver and triple-matrix-multiply were implemented with different algorithms to evaluate parallelisation with CellSs. Although current limitations of CellSs (v1.4) present certain difficulties (particularly with respect to support for

applications written in C++ or Fortran), it provides programmers a flexible programming model with an adaptive parallelism level that provided acceptable performance and portable code, while considerably simplifying Cell/B.E. programming. In particular, CellSs automatically implements efficient double-buffering of data transfers, which are complicated to implement natively with the CellSDK. A novel algorithm was thereby developed for triple-matrix-multiply, which is efficient in both storage requirements and computation, ultimately making the algorithm published in [4] realisable on Cell/B.E.Although significant effort was invested in the current implementations, it is still introductory work without vectorisation and pipelining optimisations. With new versions of Cell/B.E. and CellSs, further performance improvements are also expected.

6 Acknowledgement

The authors would like to thank the Cell group at Jülich Supercomputing Centre (JSC) for providing hardware support and sharing their Cell/B.E. experience. Special thanks to Pieter Bellens, Rosa M. Badia, Josep M. Perez Cancer and Jesus Labarta of Barcelona Supercomputing Centre for their technical support for CellSs and instructive discussion. Liang Yang also thanks the China Scholarship Council for funding his one year visit to JSC.

References

1. David A. Bader, Virat Agarwal and Kamesh Madduri, *On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking*, 21st Int'l Parallel and Distributed Processing Symp. (Long Beach, CA), March 2007
2. Josep M. Perez, Pieter Bellens, Rosa M. Badia and Jesus Labarta, *CellSs: Making it easier to program the Cell Broadband Engine processor*, IBM Journal of R&D, 51(5):593–604, Sep. 2007
3. Pieter Bellens, Josep M. Perez, Rosa M. Badia and Jesus Labarta, *CellSs: A Programming Model for the Cell BE Architecture*, Proc. SC06 (Tampa, FL), Nov. 2006
4. Godehard Sutmann, Silke Waedow, *A Fast Wavelet Based Evaluation of Coulomb Potentials in Molecular Systems*, Computational Biophysics to Systems Biology 2006, NIC Series volume 34, pp. 185–188, John von Neumann Institute for Computing, Forschungszentrum Jülich, Sep. 2006.
5. Paul Gibbon and Godehard Sutmann, *Long range interactions in many-particle simulation.* In *Quantum simulations of many-body systems: from theory to algorithms*, NIC Series, volume 10, pp. 467-506. John von Neumann Institute for Computing, Forschungszentrum Jülich, Feb. 2002.
6. Matthias Bolten, Andreas Dolfen, Norber Eicker, Inge Gutheil, Willi Homberg, Erik Koch, Annika Schiller, Godehard Sutmann, Liang Yang, *JUICE - Jülich Initiative Cell Cluster Report 2007*, FZJ-ZAM-IB-2007-13, Jülich Supercomputing Centre, Forschungszentrum Jülich, Dec. 2007
7. F. Marino and V. Puri and E. E. Swartzlander Jr., *A Parallel Implementation of the 2-D Discrete Wavelet Transform without Interprocessor Communications*, In *IEEE Transactions on Signal Processing*, volume 47, pp. 3179-3184, Nov. 1999