Programming Approach for Accelerator-based Heterogeneous Multiprocessors for Scientific Applications

Enes Bajrovic and Eduard Mehofer

Department of Scientific Computing
University of Vienna, Austria
{bajrovic,mehofer}@par.univie.ac.at
http://www.par.univie.ac.at

October 2009

Abstract

We address programming of accelerator-based heterogeneous multiprocessors in the context of computational science. Specifically, we consider stream architectures with explicitly managed memory hierarchies. In this paper we present a programming approach which supports program development for such multiprocessors. The programming approach is based on a coordination model which allows a programmer explicitly to control parallel activities and to manage memory hierarchies. Accelerators are only beneficial, if one succeeds to map the computational kernel efficiently onto the non-general-purpose hardware. Since the target architecture of our programming system are stream multiprocessors, namely Cell/BE, streaming abstractions are provided to improve programmability of stream kernels and to enable profitable compiler optimizations. Parallelization techniques for code generation are presented. First experiences back up the approach.

1 Motivation

Whereas at the beginning of the multi-core era mainly homogeneous multiprocessors have been built, the trend goes now towards heterogeneous multi-core designs. Usually these hybrid systems consist of standard cores enhanced by dedicated non-general-purpose accelerators with explicitly managed memory hierarchies. Such hybrid architectures raise new programming challenges. First, it must be determined which tasks shall be done by the standard CPU and for which tasks is it beneficial to map them onto the accelerators. Second, the tasks must be parallelized for the accelerators. Usually, accelerators are non-general-purpose processors which operate on their local memories only. A major difficulty are the explicit data movements between main memory and local memories. But even if all the data are in the local memory, efficiency is still a challenging problem, since programming of accelerators is not as simple as programming standard CPUs.

Currently, major research efforts are undertaken by academic and industrial institutions to find answers how to deal with these new programming challenges and to leverage the computing power of those multiprocessors. Different approaches are discussed controversially without a consensus within the community.

Even after decades of research, there is still often a large performance gap between automatic parallelization and explicit parallel expert code. One strategy to overcome this problem is to define domain-specific language features for dedicated application classes instead of pursuing general-purpose programming. Such a strategy is obvious for multiprocessors like Cell/BE which are not general-purpose processors either. Our approach reflects three main design goals with respect to processors, applications, and program development.

1. **Processors.** We target stream architectures like Cell/BE with accelerators and explicitly managed hierarchical memories.

2. **Applications.** We target stream-like applications in the field of computational science, i.e. applications with large data arrays ("streams") as input, local calculations on that data, and, optionally, large data arrays written as output. Since usually accelerators have small local memories, data have to be divided into blocks and streamed into / out of the accelerators. Another characteristic is that the calculations can be done independently and are self-contained without accessing main memory randomly. Language extensions for streaming abstractions results in better programmability and enables the compiler to perform more profitable optimizations.

3. **Program development.** Ignoring the parallel control directives shall result in a semantically equivalent sequential version of the program which can be compiled and executed. Hence, all sequential tools like debuggers can be used to analyze the program and to eliminate bugs before the parallel version is executed. This is important particularly for languages like C where it is even for sequential programs challenging to correct pointer errors.

In this paper we present our system VieCell which assists programming hybrid multi-cores, namely Cell/BE multiprocessor. Our approach reflects the principle that parallelization must be under programmer control—efficient parallel algorithms can be developed by programmers only and cannot be generated automatically from sequential algorithms. A small number of directives controls parallel execution. In fact, we only add a coordination model to the sequential programming language C. Basically a parallel program can be separated into *computation* and *coordination* [11]. The computation model allows a programmer to write a single-threaded computational activity, whereas the coordination model supports thread creation, data movement, and synchronization. Instead of integrating both models into one single language with the effect that coordination takes place implicitly, we argue for separating both models and making coordination explicit. This facilitates also to meet our third design point concerning the semantical equivalence between the sequential program and the parallel program.

The basic computational unit for parallel execution are functions which are spawned on the accelerators along with an execution context created by parameter transfers at invocation and return. Hence, data transfers are aggregated to larger pieces which reflects the shopping-list parallelization strategy as proposed by Cell/BE chief scientists [14] for such kind of architectures. Moreover, we aggregate multiple calls of a function to one call and pass parameter lists with *work-arrays* to reduce the overhead of loading and executing a function on an accelerator and to enable overlapping of parameter transfers with computation. Streaming programming features assist developers writing efficient code. The compilation system realizes low-level tasks like thread management, data transfers, or machine specific optimizations—tasks which can be handled by compilers successfully and which should not be dealt with by the programmer for portability reasons. The strategies applied for realizing these tasks are presented in the paper. Our application domain are scientific applications which are usually characterized by floating point operations on large data arrays.

The paper is organized as follows. Section 2 presents system VieCell, starting with the target hardware architecture in Section 2.1 and the model of computation in Section 2.2. The programming approach is illustrated in Section 2.3 with a running example, for which the generated code is shown in Section 2.4. Additional language features are presented in Section 2.5. Runtime measurements and optimizations are discussed in Section 3. The paper concludes with related work in Section 4 and a summary and future work in Section 5.

2 Programming System VieCell

2.1 Target Hardware Architecture

We consider heterogeneous multiprocessors with a main CPU and a number of accelerators or co-processors with local memories as shown in Fig. 1. Whereas the main processor can operate on the main memory, the accelerators can address directly only the local stores. Data transfer between local stores and main memory are managed explicitly and not implicitly with e.g. load/store instructions. Typically, the main processor and the accelerators have different instruction sets. It is the task of the main processor to load the binaries onto the local stores of the accelerators and to initiate execution. In this way, the main processor orchestrates the components of the chip.

The block diagram of Fig. 1 fits exactly to Cell/BE multiprocessor—our first target architecture. Cell/BE is a heterogeneous multiprocessor with an IBM Power processor core, called PPU (Power Processor Unit), and 8 specialized accelerators or co-processors with local stores, called SPU (Synergistic Processor Unit). The PPU and SPU have different instruction sets and the SPU contains a SIMD execution unit. The small local store of 256 KB holds instructions and data and is the only memory directly addressable by the SPU. Therefore it is important

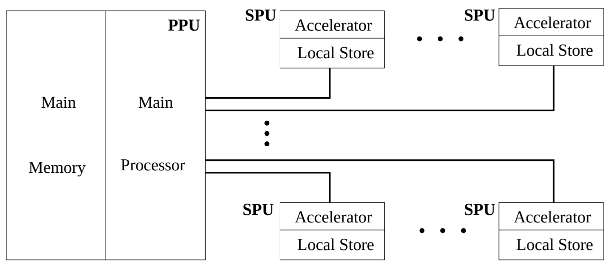

Figure 1: Cell/BE like target architecture with accelerators and explicitly managed memory hierarchies.

that the programming framework provides means to deal with the small local stores instead of delegating it to the programmer.

2.2 Model of Computation

Parallelization is fully controlled by the programmer, since development of parallel algorithms is an inherent task of programmers which cannot be hidden. The parallel code is specified together with the required data transfers. Since the computation model is covered by programming language C, the coordination model has to be addressed only. The required extensions for the coordination model have been realized with directives embedded in the sequential language. Ignoring the directives results in a semantically equivalent sequential version of the program.

The basic computational unit which can be executed in parallel is an SPU function extended by a parameter in/out-description. The data transfers take place at function invocation and return, and constitute the execution context of the SPU function. The parallel execution of an SPU function on an SPU is initiated with a parallel-loop: the loop index space is divided by the number of SPUs and each SPU is assigned an index range.

For Cell-like architectures it is an obvious approach that at program start-up a single master thread is created on the PPU which exists for the duration of the whole program and which starts executing the program sequentially. When the master thread encounters a `parallel` loop, slave threads are created for each SPU to control parallel execution. The task of each slave thread is to load the executable of an SPU function onto the SPU, transfer at the beginning the data to the local memory, and write the result back to main memory. After termination of all slave threads, the master thread in the PPU continues execution. Thus the PPU acts as orchestrator responsible for realizing work distribution and coordinating parallel execution.

Parallel execution is controlled by following directives:

- **pragma parallel**. When the master thread reaches a `parallel` loop, the PPU loads the binary of the SPU function onto the SPUs and distributes the work between the

SPUs. The body of a `parallel` loop contains exactly one SPU function call and the programmer asserts that it is legal to execute the function in parallel. SPUs are not allowed to execute a `parallel` loop which is guaranteed by the compiler.

- **pragma public**. An SPU compilation unit consists of declarations and function definitions whereby one function defintion has the *public*-attribute. Functions with the *public*-attribute are called *SPU functions* and are invokable from the PPU, the remaining functions have internal file scope only. The name of the SPU function must be identical to the source file (cf. Java).

  The *parameter clause* specifies for each parameter whether it is an *in*, *out*, or *inout* parameter together with the number of data elements to be transferred. The semantics of the parameter transfer is call-by-value-result, i.e. the arrays are copied between main memory and local memory forward and backward. The *blockable clause* indicates that it is valid to divide parameter transfers in smaller units in order to prevent memory overflow of the small local memories and to support stream-like processing behavior.

- **pragma comm**. The *communication*-attribute indicates that data structures allocated by the PPU (PPU compilation unit) will be transferred between PPU and SPU. This attribute is used to take care of alignment.

2.3 Programming Approach with Running Example

To illustrate the programming approach with system VieCell, we present and discuss a running example: *matrix-matrix addition* $C_{m,n} = A_{m,n} + B_{m,n}$. We do not start with a sequential version of matrix-matrix addition, but immediately with a parallel version which is based on the fact that the matrix operation can be split up in m independent additions of vectors of length n. The user code is shown in Fig. 2.

The for-loop iterates over all rows and adds row-by-row by calling SPU function `SPU_vec_add` at line 5 along with pointers to the rows. Since the additions of the vectors are independent, VieCell is told to perform the SPU function in parallel on the SPUs with the pragma directive `#pragma vie parallel` at line 3. The master thread creates for each SPU a slave thread which controls execution of the corresponding SPU. The PPU divides the loop index space by the number of SPUs and assigns the work to the SPUs. The pragma directive `#pragma vie comm` at line 1 indicates that the subsequent declared variables will be passed to an SPU function.

The SPU function is shown in Fig. 2. The function gets as input parameter the pointers to the vectors which shall be added and the pointer to the result vector with the sum. Since the SPUs can operate on their local memories only, data transfers between main memory and local stores are required which are controlled by the pragma directive just before the SPU function at line 1. The parameter clauses for `vec1` and `vec2` mark them as *input-parameters* and results in `N` elements of type float to be transferred from main memory to local memory starting from addresses `vec1` and `vec2` just after function activation. Parameter `vec3` is annotated

```
#pragma vie comm
float C[M][N], A[M][N], B[M][N];
⋮ sequential execution
#pragma vie parallel
for (int i=0; i<M; i++)
  SPU_vec_add(&A[i][0],&B[i][0],&C[i][0]);
⋮ sequential execution
```

(a) PPU user code.

```
#pragma vie public vec1(in,N), vec2(in,N),
vec3(out,N)
void SPU_vec_add(float vec1[],
                 float vec2[],
                 float vec3[])
{
  for (int j=0; j<N; j++) {
    vec3[j]=vec1[j]+vec2[j];
  }
}
```

(b) SPU user code.

Figure 2: Running example.

as *output-parameter* and causes `N` data items to be written from the local memory to main memory address `vec3` just before function return.

Thus parallel execution is completely controlled by the programmer which is essential for the development of efficient parallel algorithms. However, the programmer shall not deal with low level programming details of the accelerators. The accelerators are usually non general-purpose processors with sometimes idiosyncratic instruction set architectures; e.g. in most cases the programmer code of Fig. 2 will not perform well, unless optimizations like vectorization or loop unrolling have been applied—optimizations manageable by compilers successfully.

2.4 Unoptimized Code for Running Example

The generated PPU code is shown in Fig. 3. First, the PPU determines the number of available SPUs and creates for each SPU a thread at line 4. For each SPU an execution context `ctx` is created at line 7 to which specifications like the SPU function handle is added next line. Functions with prefix "spe_" are part of Cell/BE SDK (see below).

The PPU determines the work distribution by dividing the loop index space by the number of available SPUs and assigns each SPU the corresponding index range with a *work-array* which is constructed by the PPU and attached to the SPU execution context at line 10. Since the data transfers are initiated by the SPUs, the work-array contains the addresses of all parameters, i.e. for each loop iteration assigned to an SPU a tuple of three vector pointers is added to the work-array as shown at line 9 based on set notation. The work distribution is a simple block-distribution with the block-size calculated at line 3. Besides the specification of the work distribution, the work-array has a second important task: the work-array reduces the interactions between PPU and SPU and realizes the already mentioned shopping-list parallelization strategy. The SPU function is loaded and called exactly once for

```
01  aligned float C[M][N], A[M][N], B[M][N];
    ⋮
    ⋮  sequential execution
02  num_threads=get_num_SPUs();
03  blksz=M/num_threads;  // for simplicity no remainder
                            assumed
04  pthread_create thread$_t$ ($1 \leq$ t $\leq$ num_threads)  {
05    spe_context_ptr ctx;
07    ctx$_t$ = spe_context_create(...);
08    spe_program_load(ctx, SPU_vec_add);  // add SPU
                                         function to context
09    work_array$_t$=$\bigcup_{i=blksz*(t-1)}^{blksz*t-1}$(&C[i,0],&A[i,0],&B[i,0]);
10    spe_context_run(ctx$_t$, work_array$_t$);  // code loaded
                                         and executed by SPU
11    spe_context_destroy(ctx$_t$);
12  }  // join of pthreads
    ⋮
    ⋮  sequential execution
```

Figure 3: Generated PPU pseudo code for running example.

each SPU, instead of M calls. Moreover, the work-array makes it easy to apply optimizations like double buffering (see below) to overlap parameter transfers with computation.

After all SPUs have terminated, the context is destroyed at line 11 and after all threads have finished (implicit barrier), the master thread continues sequential execution.

The generated SPU code is shown in Fig. 4. In order to execute an SPU function, a main routine is required which is invoked by the PPU. The context of the SPU function is passed by the parameters of the main routine, e.g. the work-array is assigned at line 12. Basically, the generated SPU code iterates over the work-array and performs for each parameter tuple the following three steps: (1) copy the values of the input parameters to the local memory, (2) perform the original SPU kernel function, and (3) copy the values of the output parameters to main memory. Step 1 goes from line 17 to line 19, step 2 takes place at line 20, and step 3 goes from line 21 to line 22. Functions with prefix "vie_" are wrapper functions of our runtime system which call the corresponding Cell/BE SDK functions with prefix "mfc_" (for memory flow controller).

Since the work-array can be huge and only 16KB can be transferred by a DMA transfer, we apply loop tiling (lines 14 and 15) to split the work-array into appropriate tiles. Similar, our runtime functions *vie_mfc_get/put* at lines 17 or 21 split the data transfers into smaller pieces, if they exceed 16KB.

Compilation system and file organization. System VieCell is a source-to-source translator. VieCell translates files with extensions `.ppc` (PPU file) and `.spc` (SPU file) to C-files

```
aligned float vec1[N], vec2[N], vec3[N]; // function
                                          parameters

void SPU_vec_add()
{
    for (int j=0; j<N; j++) {
    vec3[j]=vec1[j]+vec2[j];
    }
}

int main(...argp,...)
{
work_array=argp;
while work_array not empty {
 vie_mfc_get(work_tile, work_array.next_tile,
           TILESZ); //  get tile-by-tile
  for (int i=0; i<TILESZ; i++) {
    (C_ptr,A_ptr,B_ptr)=work_tile[i]; // get next
                       main memory address tuple
    vie_mfc_get(vec1,A_ptr,N*sizeof(float));
    vie_mfc_get(vec2,B_ptr,N*sizeof(float));
    vie_mfc_wait(); // wait on copy completion
    SPU_vec_add();
    vie_mfc_put(vec3,C_ptr,N*sizeof(float));
    vie_mfc_wait(); // wait on copy completion
    }
  }
}
```

Figure 4: Generated SPU pseudo code for running example.

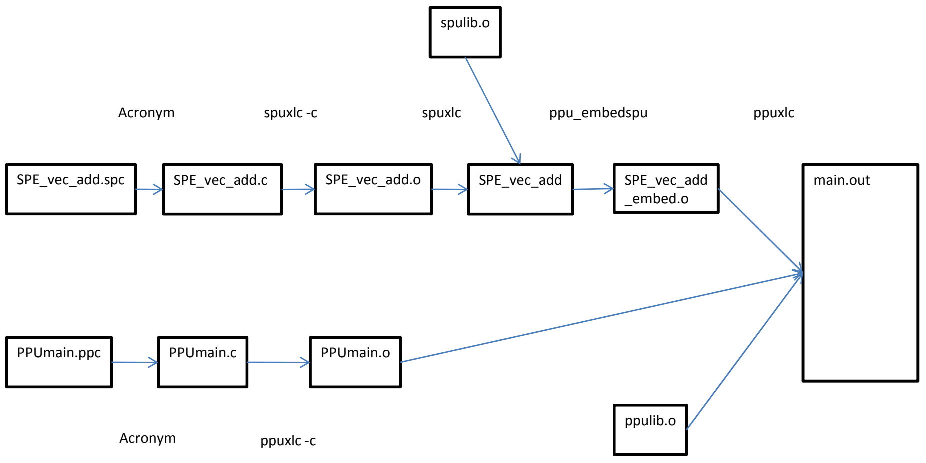

Figure 5: Compilation steps.

with *DaCS* (Data Communication and Synchronization library) calls from Cell/BE SDK. The Cell/BE SDK for Multicore Acceleration architecture is shown in Fig. 6. *DaCS* and *ALF* (Accelerator Library Framework) are IBM libraries with DaCS supporting basic features like process management or data movement primitives and with ALF providing support for parallelizing codes with features like data partitioning (see related work section). The box called "Others" is reserved for third-party developments like VieCell. The compilation steps are shown in Fig. 5. First, VieCell translates PPU files and SPU files into C files, then IBM *ppuxlc* and *spuxlc* generate object files for PPU and SPU, respectively. The SPU object file is compiled with IBM *ppu_embedspu* into an object file which can be embedded in a PPU program. Finally, with *ppuxlc* all object files are linked together and one Cell/BE executable is generated. The object libraries *ppulib.o* and *spulib.o* are part of the VieCell runtime system.

An SPU file may contain several function definitions, but only one of them can be declared `public` and the public function must have the same name as the source file (cf. Java). The public SPU function is the SPU entry function which is invokable from the PPU and which can call local functions.

2.5 More Language Features

Blockable clause. Since scientific applications typically operate on large arrays, techniques are required to partition data such that they fit into the small 256KB local memory. VieCell provides a *blockable clause* as shown in Fig. 7 to indicate that it is semantically valid to split an array in subarrays and apply the kernel to the subarrays.

The `N(blockable)` clause at line 1 indicates that it is semantically valid to partition N into

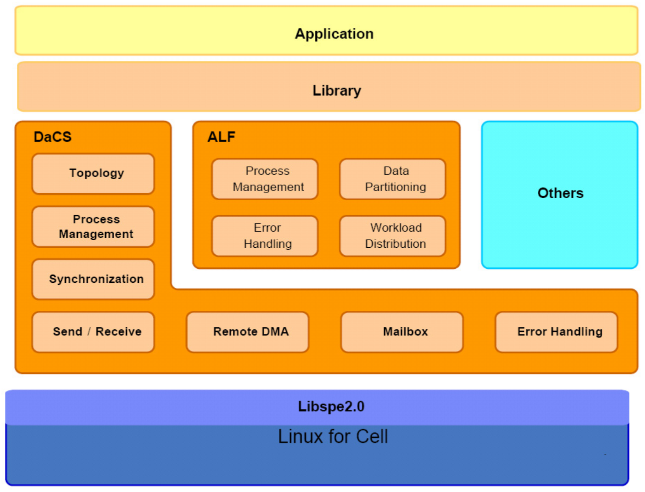

Figure 6: Cell/BE SDK for Multicore Acceleration v3.0 (Source: IBM Corporation).

```
#pragma vie vec(in,N), N(blockable)
void SPU_vec_add(float vec[])  { ... }
```

Figure 7: SPU user code with blockable input streams.

```
for 1 to k number of blocks {
  vie_mfc_get(vec1,A_ptr.next_block,N/k*sizeof(float));
  vie_mfc_get(vec2,B_ptr.next_block,N/k*sizeof(float));
  vie_mfc_wait(); // wait on copy completion
  SPU_vec_add();
  vie_mfc_put(vec3,C_ptr.next_block,N/k*sizeof(float));
  vie_mfc_wait(); // wait on copy completion
}
```

Figure 8: Generated SPU pseudo code for blockable input streams.

```
01  #pragma vie comm
02  float A[L][M][N];
03  float *red, redsum;
04  ...
05  #pragma vie comm
06  red=(float*)calloc(NUM_THREADS,sizeof(float));
07  #pragma vie parallel
08  for (int i=0; i<L; i++)
09    SPU_red(&A[i][0][0],&red[__thread_nr]);
10
11  for (int i=0; i<NUM_THREADS; i++)
12    redsum += red[i];
13  ...
```

(a) PPU user code.

```
01  #pragma vie vec(in,M*N)
02  #pragma vie red(inout,1)
03  void SPU_red(float vec[M][N],
04               float red[1])
05  {
06  ...
07  }
```

(b) SPU user code.

Figure 9: Example with reduction and multi-dimensional arrays.

smaller pieces, i.e. conceptually the compiler can transform one call to `SPU_vec_add` with parameter `vec` of length N into two calls with length $N/2$, or three calls with length $N/3$, and so on. VieCell realizes the *blockable clause* however in a stream-like manner: the SPU function `SPU_vec_add` is called exactly once and inside the function subvectors of size N/k are transferred by a loop resulting in a stream of subvectors and the possibility to perform further optimizations like double buffering. Lines 17-22 of Fig. 4 are changed for k number of blocks as shown in Fig. 8. The loop bounds within `SPU_vec_add` are adjusted accordingly.

Reductions. Reductions are not explictly supported, but can be realized as shown in Fig. 9. The VieCell variable `__thread_nr` denotes the actual thread number and can be used to specify a memory region which is exclusive for an SPU. The length of array `red` corresponds to the number of SPUs and each SPU stores the result of function `SPU_red` into location `red[__thread_nr]` as shown at line 9. After termination of all slave threads, the master thread sums up the values of array `red`. Variable `__thread_nr` is useful for similar programming techniques as well.

Multi-dimensional arrays. Moreover, the code example of Fig. 9 shows the passing of multi-dimensional arrays. At line 9 of the PPU code the first element of a two-dimensional array is passed to function `SPU_red` which is declared as an array at line 3 of the SPU code along with the number of elements to be transferred at line 1. Currently, only contiguous data transfers are supported. Note that passing of one data item is dealt with as an array of length 1, i.e. conceptually we always work on data buffers. Finally, the code example shows the allocation of a dynamic array at lines 5-6 of the PPU code which is passed to the SPU function later.

3 Runtime Measurements and Optimizations

```
01  #pragma vie comm
02  float A[M][N], X[N], Y[M];
    ⋮  sequential execution
03  #pragma vie parallel
04  for (int i=0; i<M; i++)
05    SPU_dot_pr(&A[i][0],X,&Y[i]);
    ⋮  sequential execution
```

(a) PPU user code.

```
01  #pragma vie public vec1(in,N), vec2(in,N), \
02                     vec3(out,1)
03  void SPU_dot_pr(float vec1[],
04                  float vec2[],
05                  float vec3[])
06  {
07    float sum=0;
08    for (int j=0; j<N; j++) {
09      sum+=vec1[j]*vec2[j];
10    }
11   vec3[0]=sum;
12  }
```

(b) SPU user code.

Figure 10: Matrix-vector multiplication.

In this section we report experimental results with our system. First we start with matrix-vector multiplication $A_{m,n} \times X_n = Y_m$. The PPU and SPU user code of the matrix-vector multiplication kernel is shown in Fig. 10.

The most striking difference between our running example matrix-matrix addition and matrix-vector multiplication is the return of a single scalar value of the dot product. Note that the parameter passing mechanism is based on buffers described by arrays. Thus the result of the dot product is transferred in terms of an array of size 1 (cf. lines 2 and 11). Working with data buffers has the advantage that optimizations like splitting large parameter transfers or, the other way round, aggregating small parameter transfers can be done more easily.

The latter optimization—aggregation of data transfers—is an issue for the dot product. Instead of writing single scalars back to main memory, an optimization based on loop unrolling is applied such that the scalars are aggregated and written in one DMA call together back to main memory.

The experiments have been conducted on an IBM BladeCenter QS22 with two IBM PowerXCell 8i processors (3.2GHz/ 1MB L2) mounted in an IBM BladeCenter H chassis. IBM PowerXCell 8i processor is the follow-up model of the Cell processor with much better double-precision floating-point performance. For our experiments we used Cell SDK and IBM *xlc* compiler.

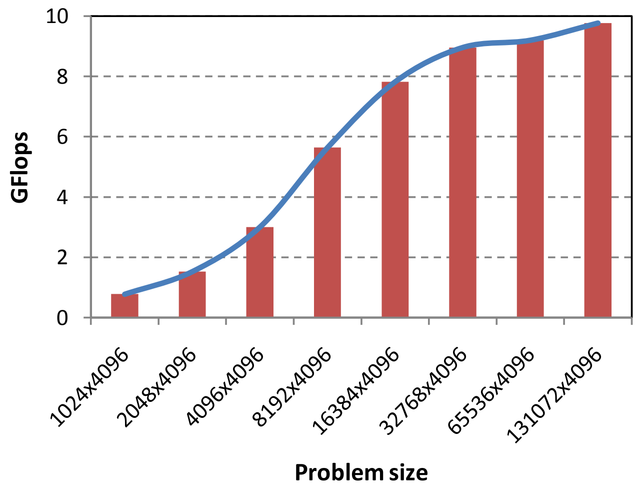

Figure 11: Matrix-vector multiplication with 8 SPUs.

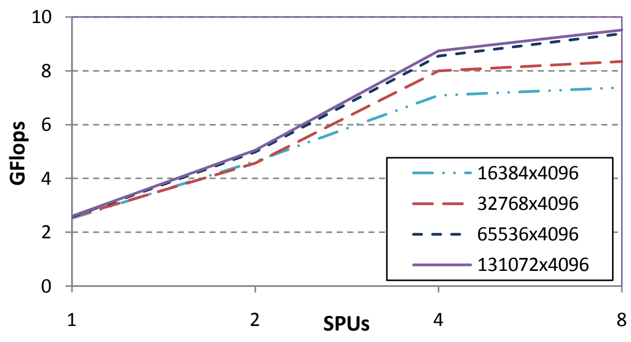

Figure 12: Matrix-vector multiplication: single buffering.

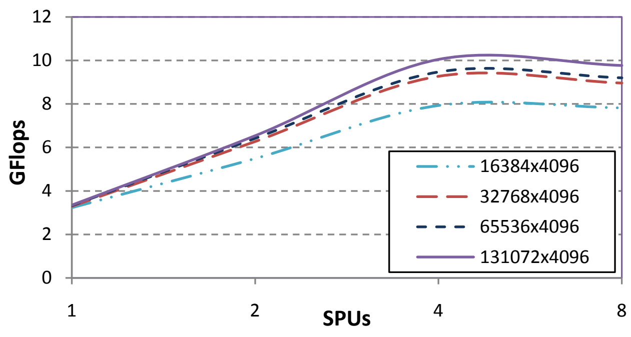

Figure 13: Matrix-vector multiplication: double buffering.

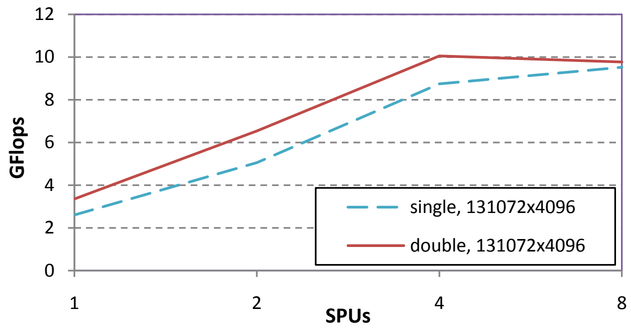

Figure 14: Matrix-vector multiplication: single vs. double buffering.

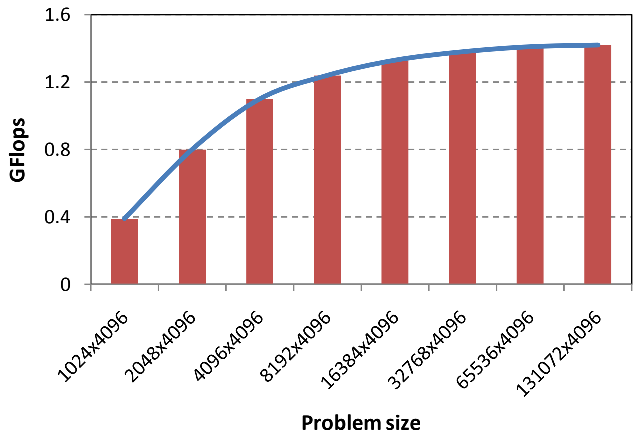

Figure 15: Matrix-matrix addition with 8 SPUs.

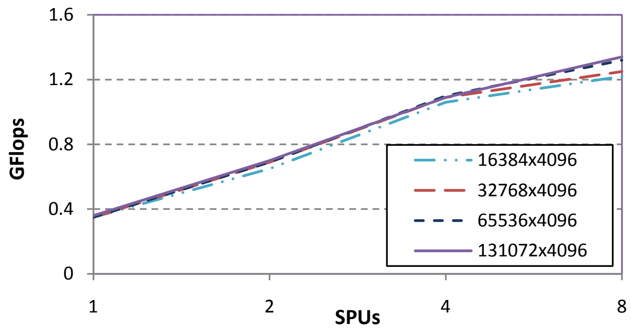

Figure 16: Matrix-matrix addition: single buffering.

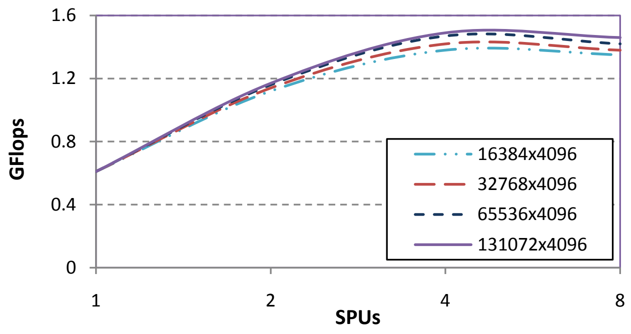

Figure 17: Matrix-matrix addition: double buffering.

On the SPU it is essential to use the vector unit in order to get performance. Note that the SPU code shown in Fig. 10 is not written as vector code, i.e. data type “float” is used instead of “vector float”. Our experiences have shown that IBM *xlc* compiler succeeds in managing vectorization for such codes making hand optimizations unnecessary. However, this is not the case for GNU *gcc*. Vectorization or loop unrolling has to be done in advance before calling GNU *gcc*.

Fig. 11 reports the performance in terms of GFlop/s for different problem sizes MxN on a single PowerXCell with 8 SPUs. The GFlop/s for larger problem sizes are competitive with measurements reported in other publications. The coordination overhead and the DMA transfers are responsible for the rather low GFlop/s for small problem sizes.

Next we analyze the performance of 1 SPU up to 8 SPUs. Fig. 12 shows the graphs for four different problem sizes. Whereas the GFlop/s can be increased from 1 SPU to 2 SPUs by factor of $1.78-1.94$ and from 1 to 4 SPUs by a factor of $2.8-3.3$, there is only a small improvement from 4 to 8 SPUs. The bandwidth problem seems to be caused amongst others by the NOC architecture of PowerXCell which is based on a ring topology with four data rings. The max. bandwidth for this computational kernel of about 18 GB/s is already reached with 4 SPUs.

An usually beneficial optimization for streaming applications is a technique referred to as *double buffering*. The input/output buffers are duplicated and while the current buffer is used in computation, the data transfer for the next buffer has already been initiated to overlap data transfers with computation. The numbers for double buffering are shown in Fig. 13. The increase of GFlop/s for double buffering is similar to single buffering for 1 to 2 SPUs and 1 to 4 SPUs, $1.70-1.95$ and $2.5-3$, respectively. From 4 to 8 SPUs, however, we even encounter a light slowdown. In Fig. 14 single and double buffering for one typical problem size is compared. On the average we get for 1 and 2 SPUs a GFlop/s increase of about 30%, for 4 SPUs about 15%, and for 8 SPUs almost no improvement. Thus, for this computational kernel double buffering is not as benefical as expected.

Next we show the performance numbers of our running example, matrix-matrix addition. Fig. 15 reports the performance in terms of GFlop/s for different problem sizes MxN on a single PowerXCell with 8 SPUs. Striking are the rather low GFlop/s compared to matrix-vector multiplication. The reason is that the amount of transferred data is higher and the amount of computation lower: an add is executed instead of a fused multiply-add operation. Thus the performance of this kernel is bandwidth bound. For higher GFlop/s numbers more compute intensive kernels are required compared to matrix/matrix addition or also matrix-vector multiplication.

Next we analyze the performance of 1 SPU up to 8 SPUs. Fig. 16 and Fig. 17 show the graphs for four different problem sizes with a similar shape as for matrix-vector multiplication. Again there is a knee from 4 SPUs to 8 SPUs. The numbers are similar to matrix-vector multiplication: the GFlop/s can be increased from 1 SPU to 2 SPUs by factor of $1.84 - 1.95$ and from 1 to 4 SPUs by a factor of $2.4 - 3.3$.

The double buffering optimization, however, is more beneficial than it was for matrix-vector multiplication. On the average we get for 1 SPU a GFlop/s increase of about 74%, for 2 SPUs about 65%, for 4 SPUs about 35%, and for 8 SPUs about 10%.

4 Related Work

Currently, many research groups from the parallel computing community as well as graphics community work on programming of accelerator-based heterogeneous multiprocessors. Our approach has been inspired by a discussion of integrating or separating coordination and computation model published by Gelernter and Carriero [11], an early effort with a coordination model called SCHEDULE by Dongarra et al. [9] with computational units scheduled for execution based on execution dependencies with some similarities with dataflow programming in the context of numerical libraries, and new research efforts in this direction [20]. Interestingly, the parallel computing scene at that time resembles in many aspects the situation today and many arguments apply in these days as well. Instead of evolving distributed-memory machines and wide-spread use of low-level MPI, now multi-cores and accelerators are emerging and low-level APIs are available—but neither at that time nor today there is a consensus in the community about the programming approach.

New programming frameworks have been proposed to facilitate use of GPUs as accelerators for general purpose programming. Usually they evolved out of the graphics community with special support for stream processing. OpenCL (Open Computing Language) [12] is an open standard for general-purpose parallel programming of multi-core CPUs, Cell/BE type architectures, or GPUs, and is conducted by the Khronos Group (cf. OpenGL–Open Graphics Library). The design of OpenCL has been primarily driven by the data parallel programming model, however, task parallelism is supported as well. A function call interface is provided for submitting kernels for execution, for defining the execution context, and for performing the data transfers. OpenCL provides a low-level hardware abstraction with many details of the underlying hardware being exposed.

CUDA from Nvidia [17], an extension of ANSI C with stream functions, provides a higher level programming interface than OpenCL, but still many low level tasks like creating kernels and performing data transfers must be done with a function call interface. Brook+ from AMD is an extension of Brook for GPUs from Stanford University [4]. Brook+ specifically targets stream programming for GPUs.

OpenCL or CUDA form a basic layer close to the hardware which are interesting for highler level approaches like ours for target code generation. For non GPU programmers, it takes some time to get used to OpenCL or CUDA coding practices.

One of the first programming environments is the RapidMind platform from the very same company [15] which targets Cell/BE, NVIDIA GPUs, and multi-core CPUs based on the SPMD stream programming model. Another toolkit is HMPP from CAPS entreprise [8] which addresses specifically legacy code issues by allowing the use of target specific software development toolkits and providing interoperability.

Closest related to our programming strategy is the approach taken by Sequoia [10]. Sequoia is also based on explicit communication by argument passing. But whereas Sequoia concentrates on handling memory hierarchies across several levels, our emphasis is placed on supporting scientific programming kernels for stream architectures.

A representative for another class of approaches which is based on OpenMP and which proposes new directives to address architectures like Cell/BE is Cellgen [19]. Differently to our approach, in Cellgen communication is not explicit, but determined by reference analysis based on work distribution with all the difficulties experienced in the past. IBM supports Cell/BE with a dedicated OpenMP version as well [18]. Specifically for scientific applications IBM provides ALF (Accelerator Library Framework) which is part of Cell/BE SDK. ALF provides a set of functions for creating tasks and defining work blocks with the data which are queued for execution. Contrary to our approach, in ALF still many low-level details have to be dealt with.

CellSs (Cell Superscalar framework) [2] is a quite different approach which is based on the automatic exploitation of functional parallelism especially for the Cell processor. A task dependency graph is built at runtime for exploiting parallelism. With this graph, the runtime is able to schedule independent nodes to different SPEs to execute at the same time. Thus, task parallelism can be exploited easily with CellSs.

Cilk [3] developed by MIT and licensed to Cilk Arts, a venture-funded start-up, is a language for multithreaded parallel programming based on ANSI C which addresses homogeneous multi-core processors.

Other well-known parallel programming languages include the PGAS (partitioned global address space) languages UPC (Unified Parallel C) [7], CAF (Co-array Fortran) [16], and Titanium (Java-based) [13]. Higher level of abstractions provide the languages of the HPCS (High Productivity Computing Systems) program of DARPA: X10 (IBM) [6], Chapel (Cray) [5], and Fortress (Sun) [1]. These parallel programming languages address general parallel applications as well as general hardware architectures and do not specifically target stream processing or heterogeneous multi-cores.

5 Conclusion and Future Work

We presented a programming approach for accelerator-based heterogeneous multiprocessors and parallelization techniques to realize the approach. Of key importance is the interplay between the three parties programmer–parallelization framework–native compiler. In our approach the programmer is responsible for developing parallel algorithms and writing explicitly parallel code in a hardware independent way.

The parallelization framework VieCell generates parallel code based on the programming directives and native libraries and hides all high-level machine characteristics. For example partitioning of large data arrays into smaller pieces to fit into the local stores shall be done by the parallelization framework and not by the programmer, since it is hardware dependent and would prevent portability.

The task of the native compiler is to cover all low-level machine characteristics and perform optimizations like vectorization or loop unrolling. Again, manual vectorization might hamper portability and compilers already succeeded in managing such kind of optimizations.

The examples show that computations fitting the stream hardware architectures can be translated in a straight-forward way onto the processors based on the directives without complex analysis, but instead with techniques like the work-arrays to support efficient streaming. Performance measurements have shown that efficient programs can be obtained in this way.

Basically, VieCell acts as a research platform to analyze the interplay programmer–parallelization framework–native compiler. In our future work, we plan to port codes onto Cell/BE which cannot be parallelized efficiently in a straight-forward way, but require either by the programmer or by the parallelization framework some support, which is investigated with VieCell. Finally, we plan to extend VieCell to support related heterogeneous processor designs like GPUs.

References

[1] Eric Allen, David Chase, Joe Hallett, Victor Luchangco, Jan-Willem Maessen, Sukyoung Ryu, Guy L. Steele Jr., and Sam Tobin-Hochstadt. *The Fortress Language Specification*. Sun Microsystems, Inc., 1.0 edition, March 2008.

[2] Pieter Bellens, Josep M. Perez, Rosa M. Badia, and Jesus Labarta. CellSs: a Programming Model for the Cell BE Architecture. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC'06)*, New York, NY, USA, 2006. ACM.

[3] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk: an efficient multithreaded runtime system. *J. Parallel Distrib. Comput.*, 37(1):55–69, 1996.

[4] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[5] B.L. Chamberlain, D. Callahan, and H.P. Zima. Parallel programmability and the chapel language. *Int. J. High Perform. Comput. Appl.*, 21(3):291–312, 2007.

[6] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *OOPSLA '05: Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 519–538, New York, NY, USA, 2005. ACM.

[7] UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.

[8] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A Hybrid Multi-core Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.

[9] Jack J. Dongarra, Danny C. Sorensen, Kathryn Connolly, and Jim Patterson. Programming methodology and performance issues for advanced computer architectures. *Parallel Computing*, 8:41–58, 1988.

[10] Kayvon Fatahalian, Timothy J. Knight, Mike Houston, Mattan Erez, Daniel Reiter Horn, Larkhoon Leem, Ji Young Park, Manman Ren, Alex Aiken, William J. Dally, and Pat Hanrahan. Sequoia: Programming the memory hierarchy. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, 2006.

[11] David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.

[12] Khronos OpenCL Working Group. OpenCL Specification, v1.0. Technical report, Khronos Group, 2009. http://www.khronos.org.

[13] Paul N. Hilfinger, Dan Oscar Bonachea, Kaushik Datta, David Gay, Susan L. Graham, Benjamin Robert Liblit, Geoffrey Pike, Jimmy Zhigang Su, and Katherine A. Yelick. Titanium language reference manual, version 2.19. Technical Report UCB/EECS-2005-15, EECS Department, University of California, Berkeley, Nov 2005.

[14] Peter Hofstee – *An Interview.* Custom Processing. *ACM Queue*, 5(1), 2007.

[15] Michael D. McCool. Data-Parallel Programming on the Cell BE and the GPU using the RapidMind Development Platform. In *GSPx Multi-core Applications Conference*, Santa Clara, CA, October-November 2006.

[16] Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.

[17] NVIDIA CUDA. NVIDIA, http://developer.nvidia.com/ object/cuda.html.

[18] Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen, and Tao Zhang. Supporting OpenMP on Cell. *Int. J. Parallel Program.*, 36(3):289–311, 2008.

[19] Scott Schneider, Jae-Seung Yeom, Benjamin Rose, John C. Linford, Adrian Sandu, and Dimitrios S. Nikolopoulos. A comparison of programming models for multiprocessors with explicitly managed memory hierarchies. In *PPoPP '09: Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 131–140, New York, NY, USA, 2009. ACM.

[20] George Wells. Coordination Languages: Back to the Future with Linda. In *Workshop on Coordination and Adaptation Techniques for Software Entities (WCAT'05)*, Glasgow, UK, 2005.