Programming Support for Cell/BE Multiprocessor

Enes Bajrović and Eduard Mehofer

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

Abstract. An emerging class of architectures are accelerator-based heterogeneous multiprocessors with software-managed memory hierarchies like Cell/BE. A major difficulty in programming such kind of machines are the explicit data transfers between the different memories raising new programming challenges. In this paper we discuss a programming approach which supports application programmers in writing efficient code for non-cache-based architectures. A crucial role plays the interplay between the three parties programmer, parallelization framework, and native compiler. Based on our experiences with past programming approaches, we propose language extensions to orchestrate parallel execution of threads and to control data transfers. Experiments are performed to analyze the roles of programmer and compiler in more detail.

1 Introduction

Accelerator-based heterogeneous multiprocessors with software-managed memory hierarchies are getting more and more wide-spread for high performance systems. 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. Besides distributing the tasks onto the standard CPU and the accelerators (if beneficial), the explicit data transfers between main memory and local memories have to be realized. 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 academia and industry 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. In this paper we discuss the interplay between programmer, parallelization framework, and native compiler. We present our system VieCell which assists programming heterogeneous multiprocessors with explicitly managed memory hierarchies, namely the 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 code. A small number of directives controls parallel execution. In fact, we only add a coordination model to the sequential programming language C. 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 discusses the interplay between programmer, parallelization framework, and native compiler which is the key for supporting heterogeneous multiprocessors successfully. The programming approach is illustrated in Section 3 together with an example. Runtime measurements and optimizations for Cell/BE are presented in Section 4. The paper concludes with related work in Section 5, and a summary in Section 6.

2 Efficient and Portable Programming of Architectures with Software-Managed Memory Hierarchies

While chip multiprocessors (CMPs) alleviate problems known as *power wall* or *instruction-level parallelism* (ILP) *wall*, they increase the *programmability wall*. On the one hand, program development for multi-core processors, especially for heterogeneous multi-core processors, is significantly more complex than for single-core processors. On the other hand, programmers have been traditionally trained for the development of sequential programs, and only a small percentage of them have experience with parallel programming. In the past, programmers could trust that compilers succeeded to pass the increased computing power of next processor generations to applications without high porting effort. This was due to relatively homogeneous processor designs even from different hardware vendors with instruction-level parallelism supported at hardware level. The architectural change to CMPs, however, affects the programmer in several ways. On the one hand, thread level parallelism (TLP) must be exploited effectively and efficiently. In general, this cannot be done automatically by a compilation system, but requires assistance by the programmer. On the other hand, multi-core architectures differ significantly requiring that applications must be adapted to the various platforms. This porting problem is worsened by the fact that the average lifetime of hardware is about 5 years, whereas the average lifetime of applications is about 20-30 years.

A crucial role in addressing the programmability wall plays the relationship between the three involved parties programmer, parallelization framework, and native compiler. Experiences in the past have shown the limits of parallelizing compilers. Above all, parallelizing compilers will fail for programs which do not exhibit parallelism, since sequential algorithms have been used. Consequently, parallelization must be under programmer control—efficient parallel algorithms can be developed by programmers only and parallelism shall not be hidden. Directives must be provided for the programmer to control the parallel activities and to manage the explicit data transfers at a high level in a machine-independent way.

Basically a parallel program can be separated into *computation* and *coordination* [8]. The computation model allows a programmer to write a single-threaded computational activity, whereas the coordination model supports thread creation, data transfer, and synchronization. A discussion of integration vs. separation can be found in Gelernter and Carriero [8]. It is highly interesting that most of the arguments apply in these days too. In the year 1992 parallel programming was dominated by message-passing and the need for better programming support. In response to the emerging architectures of that time it is said "diversity with respect to language, hardware platform, physical location ... will be normal in the new era"—a sentence which still applies nowadays.[1]

In the following we summarize the distribution of duties between programmer, parallelization framework, and native compiler.

Programmer. The programmer controls parallelization explicitly. For portability reasons, it is the goal that the code is written in a machine-independent way.

Parallelization framework. The parallelization framework supports the coordination model. The framework realizes tasks like thread management, data transfers, or machine specific optimizations during the parallelization process—tasks which can be handled by an environment successfully and which should not be dealt with by the programmer for portability reasons. Examples of machine specific optimizations are (1) splitting of big data chunks to fit in the small local memories, (2) aggregation of small data transfers to larger pieces, (3) hiding transfers with computation (e.g. double buffering), (4) streaming optimizations, or (5) eliminating synchronization points.

Native compiler. The native compiler optimizes the code assigned to the computing device. In addition to optimizations common for object code compilers, optimizations like vectorization (if a vector unit exists), loop unrolling, or software pipelining shall be supported.

3 Overview of Programming System VieCell

Our system VieCell targets heterogeneous multiprocessors, namely Cell/BE, with a main CPU and a number of accelerators or co-processors with local memories. In case of Cell/BE, the main CPU is the PPU (Power Processor Unit) and the accelerators are called SPUs (Synergistic Processor Units). Data transfers between main memory and local stores are managed explicitly and not implicitly with e.g. load/store instructions. Typically, the main processor and the accelerators have different instruction sets.

Parallelization is fully controlled by the programmer. Since the computation model is covered by programming language C, the coordination model has to be addressed only. The extensions for the coordination model have been realized

[1] By the way, message passing is still the dominating programming paradigm for scientific applications.

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 which are spawned on the SPUs. The data transfers take place at function invocation and return, and constitute the execution context of the SPU function. Hence, data transfers are aggregated to larger pieces which reflects the shopping-list parallelization strategy as proposed by Cell/BE chief scientists [10] for such kind of architectures.

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 a small number of 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.
- **pragma public**. An SPU compilation unit contains exactly one function with the *public*-attribute, called *SPU function*, which is invokable from the PPU.
  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.
- **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.

Since we are targeting stream-like applications in the field of computational science, we provide additional notations to steer optimizations for such kind of applications. As an example a parallel matrix-vector multiplication is shown in Fig. 1.

4 Experiments on Cell/BE

In this section we discuss experimental results performed on single SPU of a Cell/BE using native IBM XL C/C++ XLC compiler version 10.1 and GNU

```
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), \
02                     vec2(in,N), \
03                     vec3(out,1)
04  void SPU_dot_pr(float vec1[],
05                  float vec2[],
06                  float vec3[])
07  {
08    float sum=0;
09    for (int j=0; j<N; j++) {
10      sum+=vec1[j]*vec2[j];
11    }
12   vec3[0]=sum;
13  }
```

(b) SPU user code.

Fig. 1. Matrix-vector multiplication.

GCC open source compiler shipped with Cell SDK 3.1 (both with `-O3` optimization flag).

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.

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.

As example we take vector-vector addition and start with a straight-forward implementation (var. A) as shown in Fig. 2(a) and compile it with XLC and GCC. With XLC we got about 3.72 GFlop/s, whereas for GCC we got only 0.13 GFlop/s. The results of XLC are significantly better than for GCC, but the 3.72 GFlop/s obtained by XLC seem to be low either compared to the peak performance of 25.6 GFlop/s.[2]

For vector-vector addition, however, FMA operations cannot be used resulting in 50% of peak performance. Further, since there are three load/store operations vs. one floating-point operation, only one third can be achieved resulting in a maximum sustained performance of 4.27 GFlop/s.

[2] For single precision floating point 2 operations (FMA) are performed on each of the floats in the quad-word per cycle, leading to 3.2GHz * 8 = 25.6 GFlop/s.

```
01 float a[n];
02 float b[n];
03 float c[n];
04
05 for (i = 0; i < n; i++)
06     c[i] = a[i] + b[i];
```

(a) Scalar code (var. A).

```
01 vector float a[n];
02 vector float b[n];
03 vector float c[n];
04
05 for (i = 0; i < n/4; i++)
06   c[i] = spu_add(a[i], b[i]);
```

(b) Vectorized code (var. B).

```
vector float x0,x1,x2,x3,x4,x5;
vector float y0,y1,y2,y3,y4,y5;
vector float z0,z1,z2,z3,z4,z5;
    .
    .  pre-loop code
for (i = 0; i < n/4 - 2; i+=6) {
    // Store [i] - [i+5]
    c[i+0] = z0; c[i+1] = z1; c[i+2]=z2;
    c[i+3] = z3; c[i+4] = z4; c[i+5]=z5;
     // Compute [i+1]  [i+6]
    z0=spu_add(x0, y0); z1=spu_add(x1, y1); z2=spu_add(x2, y2);
    z3=spu_add(x3, y3); z2=spu_add(x2, y2); z3=spu_add(x3, y3);
     // Load next a: [i+12] to [i+17]
    x0 = a[i+12]; x1 = a[i+13]; x2 = a[i+14];
    x3 = a[i+15]; x4 = a[i+16]; x5 = a[i+17];
    // Load next b: [i+12] to [i+17]
    y0 = b[i+12]; y1 = b[i+13]; y2 = b[i+14];
    y3 = b[i+15]; y4 = b[i+16]; y5 = b[i+17];
}
    .
    .  post-loop code
```

(c) Software pipelining with unrolling (var. E).

Fig. 2. Vector-vector addition.

Now we try to hand-optimize the code to get with GCC similar performance results like XLC. First we vectorized the code (var. B) as shown in Fig. 2(b). For XLC nothing changed, while with GCC we got 0.78 which is a speedup by a factor of 6, but still far away from XLC.

Explicit loads and stores to enable software pipelining (var. C) resulted in doubling the performance of the code compiled with GCC, with 1.58 GFlop/s, which is a speedup of 12.15 compared to the initial code variant A.

Next, we apply software pipelining and a technique similar to loop unrolling by a factor 2 (var. D) and factor 6 (var. E) as shown in Fig. 2(c). The SPU has two distinct instruction pipelines supporting dual-issue. Load/store instructions move the data from local store to registers and back with the latency of 6 cycles. On the other hand, floating-point operations take exactly the same amount of cycles. If we consider two pipelines and only loads/stores and floating-add operations, the instruction flow looks like in Fig. 3. With this optimization we got for GCC 3.72 GFlop/s which is a speedup of 30, and for XLC approximately the same result.

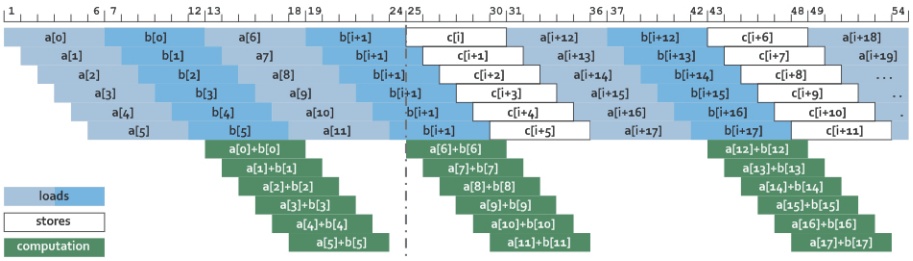

Fig. 3. Software pipelining with unrolling.

Finally, we succeeded for GCC to obtain similar performance numbers like for XLC, however, the hand-optimized code is much more complex compared to the initial code version. A summary of the performance numbers can be found in Fig. 4 and a graphical presentation is shown in Fig. 5.

The experiments have shown that XLC manages the complexity of the low-level optimizations successfully resulting in good performance numbers. Obviously, optimizations performed by the programmer in order to compensate compiler deficiencies results in non-portable complex code. Further, programmer productivity decreases as well.

5 Related Work

Many research groups from the parallel computing community as well as graphics community work on programming of accelerator-based heterogeneous multiprocessors. Related approaches include CUDA from Nvidia [13], Brook for GPUs

Program variants	GCC	XLC	Speedup GCC/XLC
Scalar (A)	0.13	3.72	1.00/1.00
Vectorized (B)	0.78	3.72	6.00/1.00
SW Pipelining (C)	1.58	3.72	12.15/1.00
SW pipelining with prefetching 2 elemets (D)	1.71	3.81	13.15/1.00
SW pipelining with prefetching 6 elemets (E)	3.72	3.84	30.00/1.03

Fig. 4. Performance results as table.

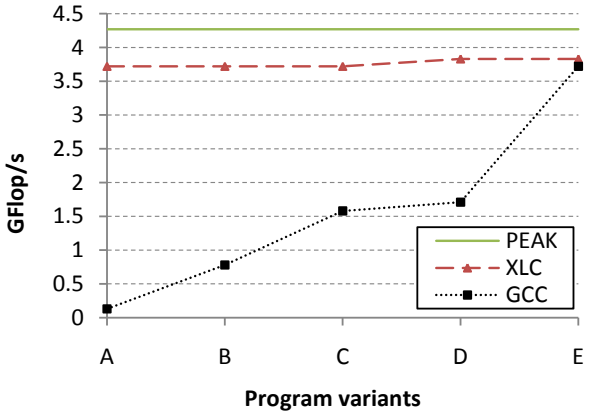

Fig. 5. Performance results: graphical presentation.

from Stanford University [2], StreamIt from MIT [14], HMPP from CAPS entreprise [6], and RapidMind platform from the very same company [11]. Other well-known parallel programming languages include UPC (Unified Parallel C) [5], CAF (Co-array Fortran) [12], Titanium (Java-based) [9], and Sequoia [7]. Higher level of abstractions provide the languages of the HPCS (High Productivity Computing Systems) program of DARPA: X10 (IBM) [4], Chapel (Cray) [3], and Fortress (Sun) [1].

6 Conclusion

We discussed the importance of the interplay between the three parties programmer, parallelization framework, and native compiler which is the key for supporting heterogeneous multiprocessors successfully. The programming approach presented in this paper addresses Cell/BE like architectures and is based on a coordination model added to C. The separation between coordination and computation fits specifically to architectures like Cell/BE with PPU as main unit orchestrating the parallel activities and SPUs as accelerators; or even in bigger contexts like the Los Alamos Roadrunner architecture[3] with Opterons as main units and orchestrators and Cell/BE multiprocessors as accelerators.

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. 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.
3. 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.
4. 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.
5. UPC Consortium. UPC Language Specifications, v1.2. Technical Report LBNL-59208, Lawrence Berkeley National Lab, 2005.
6. Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP: A Hybrid Multicore Parallel Programming Environment. In *Workshop on General Purpose Processing on Graphics Processing Units*, Boston, MA, October 2007.
7. 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.

[3] Since Nov 2008 the fastest supercomputer in the TOP500 list and the first computer breaking the petaflop/s performance barrier.

8. David Gelernter and Nicholas Carriero. Coordination languages and their significance. *Commun. ACM*, 35(2):97–107, 1992.
9. 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.
10. Peter Hofstee – *An Interview*. Custom Processing. *ACM Queue*, 5(1), 2007.
11. 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.
12. Robert W. Numrich and John Reid. Co-array fortran for parallel programming. *SIGPLAN Fortran Forum*, 17(2):1–31, 1998.
13. NVIDIA CUDA. NVIDIA, http://developer.nvidia.com/ object/cuda.html.
14. William Thies. *Language and Compiler Support for Stream Programs*. Ph.d. thesis, Massachusetts Institute of Technology, Cambridge, MA, Feb 2009.