Compiling for Increasing On-chip Parallelism

Yuan Zhao Ken Kennedy
Computer Science Department, Rice University
6100 Main St, Houston, Texas 77005
Email: [yzhao|ken]@cs.rice.edu

Abstract

It becomes a trend that microprocessor companies are adding more and more parallelism on a chip to increase performance per chip. At the fine granularity level, vector instruction sets are added. While at the coarse granularity level, multiple cores are put on the same chip. This trend presents a challenge for application developers as well for compiler developers: how to exploit the power of these introduced parallelism? In this paper, we present a source-to-source compiler that automatically compiles programs written by ordinary users targeting the on-chip parallelism without users specifying parallelism directives. Initially developed for short vector processors, this compiler is extended to support a heterogeneous multi-core CELL processor. Besides parallelism, these processors also introduced various memory constraints such as data alignment and data movement that will affect an application's performance. Thus we will discuss our compiler strategies for these issues as well.

Keyword: *On-chip Parallelism, Vectorization, Parallelization, Heterogeneous Multi-core, Memory Hierarchy Performance, Data Alignment, Data Movement, Vector Data Reuse*

1 Introduction

It becomes a trend that microprocessor companies are adding more and more parallelism onto a single chip as a different way to improve the performance per chip than scaling up the operating frequencies, which has encountered problems such as heat dissipation. For fine-grained parallelism, vector function units are added to support vector instruction sets such as SSE on x86 and x86-64 processors from Intel and AMD, and AltiVec on PowerPC processors from IBM and Motorola. For coarse-grained parallelism, additional cores are put onto the same chip. These cores are either identical on a homogeneous multi-core processor such as the Intel and AMD ones, or different on a heterogeneous multi-core processor such as the CELL processor developed by Sony, Toshiba and IBM.

Increasing parallelism on chip presents a challenge for both the application and the compiler developers: how to fully exploit the power of these parallelism? To make matters worse, there are additional memory constraints such as data alignment and data movement associated with these parallelism and the performance of an application also depends on how well these memory constraints are addressed. While application developers can manually parallelize and vectorize their code with expert knowledge, we think a general approach for ordinary users is to have a compiler that automatically compiles sequential code onto the targeted architecture.

In this paper, we present such an automatic approach that we developed initially for short vector processors with SSE or AltiVec, and later extended for a CELL processor. The focus of our source-to-source compiler is the loop nests that often represent computation kernels in scientific applications. Given a sequential program written in Fortran 90, based on dependence analysis information, the loop nests are parallelized and vectorized when applicable. The output program is in a mix of Fortran and C code: the sequential part of the program remain unchanged in Fortran, while the transformed loop nests are procedure outlined and rewritten in C to take advantage of the intrinsics for parallelism, as shown in Figure 1. The output is then passed to and compiled by a vendor compiler to generate the executable.

In what follows, Section 2 presents our compiler for short vector processors, Section 3 discusses the extension of the compiler for a CELL processor, Section 4 reviews related work, and Section 5 concludes.

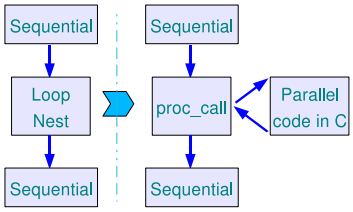

Figure 1: Compilation model

2 Compiling for Short Vector Processors

We focus on the SSE and AltiVec vector instruction sets that are widely available on mainstream microprocessors. Initially designed for multimedia applications, they can be used to speedup scientific applications as well when fully utilized.

2.1 Architecture Characteristics

Short vector length Compared to conventional vector machines, the vector length of SSE and AltiVec vectors is relatively short—16 bytes.

Continuous memory access Vector operations can only fetch or store continuous data in memory, *i.e.* strided vector accesses are not supported.

Data alignment Vector data accesses to data aligned on 16-byte boundaries are much more efficient than those to unaligned ones.

Vector intrinsics Programmers can directly access vector data types and vector operations through vector intrinsics in C.

2.2 Compilation Strategy

We follow the compilation model shown in Figure 1 and vectorize loop nests using the intrinsics. For an innermost loop to be *Short Vectorizable* with SSE or AltiVec, the following conditions need to be met:

- It does not carry true or output dependences or dependence cycles, except that an anti-dependence on a statement itself is allowed;
- Each array reference is either loop invariant or memory continuous to the loop.

Our compiler implementation is based on the D system developed at Rice University for compiling Fortran programs. As shown in Figure 1, the vectorized loop is rewritten in C using the SSE or AltiVec intrinsics. Since we are targeting both SSE and AltiVec, the vector operations are abstracted in the intermediate representation and are converted into real intrinsics during the final stage of code generation.

2.3 Optimizations for Performance

Besides vectorization, the performance of an application using SSE or AltiVec is affected by the memory alignment constraint. To make as many array references aligned as possible, the array references in the loop is first partitioned into equivalent classes based on their alignment boundaries, then the following optimizations can be applied accordingly:

Loop peeling makes array references in one equivalent class aligned upon entry to the first iteration of the remaining loop after peeling;

Loop alignment shifts the relative alignment distance between an array reference on the left hand side and one on the right hand side, and could put two references into the same equivalent class;

Software pipelined vector accesses treat each array reference in the loop as a data stream rather than an independent access in each iteration, and pipeline the stream so that only one vector access and one vector shuffling are required in each iteration;

Array padding pads the innermost dimension of a multi-dimensional array so that two array references with the same innermost-dimension index are aligned on the same boundary (*e.g.* $A(I, J)$ and $A(I, J + 1)$).

Another optimization to reduce vector data accesses is data reuse in vector registers. We implemented a version of *vectorized scalar replacement*, which is a straight-forward extension from the scalar replacement for data reuse in scalar registers.

Experimental results are omitted for short vector processors due to the space limit of this paper.

3 Compiling for a CELL Processor

Figure 2 shows a basic diagram of a heterogeneous multi-core CELL processor:

3.1 Architecture Characteristics

Parallelism It has 1 PPE core (PowerPC) and 8 SPE cores that are capable of short vector operations. PPE is responsible for fork-and-joining SPE threads.

Data movement Each SPE has its own 256KB local store memory (LS). The program running on

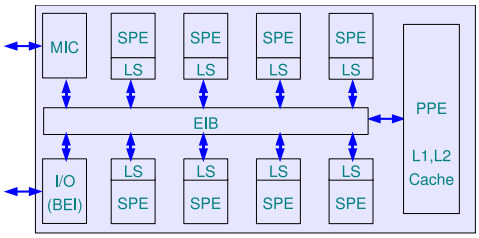

Figure 2: A CELL processor

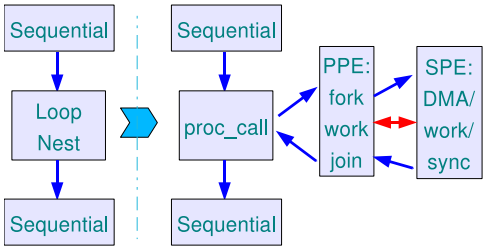

Figure 3: Compilation model on CELL

SPE can only access data in LS, all data transfers in between LS and main memory have to be explicitly controlled by DMA commands. Each DMA operation can transfer $1, 2, 4, 8, 16 * k$ bytes up to $16K$ bytes of data on naturally aligned boundaries.

Intrinsics The mechanisms for fork-and-joining SPE threads, DMA transfers, and synchronization are accessible from intrinsics in C.

Though SPE cores have a different vector instruction set from the PPE core, they have similar memory constraints such as data alignment and continuous memory accesses. On CELL, data alignment also affects the performance of DMA transfers.

3.2 Compilation Strategy

For a loop nest that can be compiled using the parallelism (*CELLizable*) on a CELL processor, we need to find a loop in the nest that is parallelizable and make sure the innermost loop is short vectorizable. Whether a loop is short vectorizable is given in Section 2. To find a parallel loop, a similar algorithm to Allen and Kennedy's vectorization algorithm [3] is developed. The algorithm traverses from the outermost loop towards the innermost loop, searching for a loop that carries no dependence. Such a loop is marked parallel and its iterations will be partitioned later. If no such loop is found, the outermost loop is marked sequential and removed from the loop nest along with all dependences it carries, and the search process is repeated until a parallel loop is found. Note that the innermost loop can be the candidate for both parallelization and vectorization.

Once a CELLizable loop nest is identified, it will be procedure outlined and its computation will be partitioned among PPE and SPEs, as shown in Figure 3. The loops marked sequential will require a barrier synchronization among PPE and SPEs.

We implemented this compilation model as an extension to the short vector compiler presented in Section 2:

- parallel loop detection and code generation;
- multi-buffering DMA data transfer;
- vectorization support for the SPE instruction set (this is fairly easy due to the vector abstraction in the intermediate representation);
- code generation for synchronization among PPE and SPEs.

3.3 Optimizations for Performance

The performance of an application on CELL depends not only on finding efficient amount of parallelism, but also on how well the memory constraints are addressed. We implemented the following optimizations for performance improvement:

Multi-buffering Since data has to be explicitly transferred in between the main memory and SPEs' LS, multi-buffering can overlap data transfers with computation to hide the latency of data transfers.

Data alignment For the performance of vector computations on both PPE and SPE, optimization discussed in Section 2 such as loop peeling, loop alignment, software-pipelined vector accesses and array padding still apply here. Data alignment also affects the performance of DMA data transfers since naturally aligned boundaries are required. We perform loop peeling on the PPE side so that the data transfer on SPEs are aligned properly.

Synchronization We mentioned that a loop marked sequential in the loop nest needs a barrier synchronization among PPE and SPEs to preserve the semantics of the original program. We also implemented an uni-directional synchronization to parallelize the innermost loop that carries anti-dependence. Coupled with post-store transformation, this approach can improve the performance significantly compared to the parallelization strategy that allocates a temporary array.

Data reuse Again, vectorized scalar replacement can help increase the data reuse in vector registers, especially that each SPE has 128 registers. It can also help reduce the DMA data transfers by only transferring the leaders of the reference groups.

3.4 Experimental Results

Figure 4 shows the speedup of using multiple SPEs to using PPE only for two computation kernels, 1-dimensional and 2-dimensional stencils, on a 3.2GHz CELL blade. Our compiler applied loop peeling, software pipelined vector accesses and vectorized scalar replacement on these two codes, in addition to parallelization and vectorization. Each code is run with a small and a big problem size. We can see that SPE is much faster than PPE, and the best speedup doesn't come when more SPEs are used if the problem size is small. We believe the reason is that the amount of computation per transferred data in the tested kernels is small and the performance is determined by the data transfer. We are currently investigating methods to improve the computation to data transfer ratio.

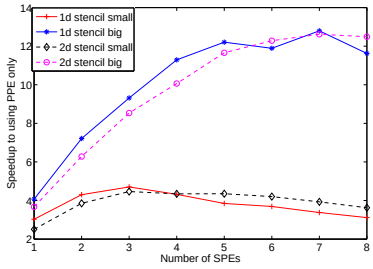

Figure 4: Speedup of using multiple SPEs

4 Related Work

Our compiler strategy is related to various loop nest parallelization and vectorization work done for various architectures [1, 2, 3, 4, 5, 8, 7, 6, 11, 9, 10]. The ones on short vector processors also address data alignment problem. For the CELL processor, OpenMP is the only parallel programming model implemented so far, by an IBM research compiler [6].

5 Conclusion

In this paper, we presented an automatic compilation strategy that compiles user programs without parallelism directives targeting the increasing on-chip parallelism in modern microprocessors, based on the dependence analysis information. Compared to the OpenMP approach, our approach needs no parallelism directives, and thus provides users additional choice when programming for CELL. Our implementation demonstrated that compilers developed for short vector processors can be extended to support multi-core processors.

Acknowledgments

This work is supported by Contract No. 12783-001-0549 from the Regents of University of California (Los Alamos National Laboratory) to William Marsh Rice University, and the CELL development systems (including CELL blades) are provided by IBM to University of Tennessee and Rice University.

References

[1] J. R. Allen. *Dependence Analysis for Subscripted Variables and its Application to Program Transformation*. PhD thesis, Rice University, Houston, Texas, 1983.

[2] Randy Allen and Ken Kennedy. Vector register allocation. *IEEE Transactions on Computers*, 41(10):pp. 1290–1317, 1992.

[3] Randy Allen and Ken Kennedy. *Optimizing Compilers for Modern Architectures*. Morgan Kauffman, October 2001.

[4] Aart J. C. Bik, Milind Girkar, Paul M. Grey, and Xinmin Tian. Automatic intra-register vectorization for the intel architecture. *International Journal of Parallel Programming*, 30(2):pp. 65–98, 2002.

[5] Crescent Bay Software. *VAST/AltiVec*. http://www.crescentbaysoftware.com/vast_altivec.html.

[6] Alexandre E. Eichenberger, Kathryn O'Brien, Kevin O'Brien, Peng Wu, Tong Chen, Peter H. Oden, Daniel A. Prener, Janice C. Shepherd, Byoungro So, Zehra Sura, Amy Wang, Tao Zhang, Peng Zhao, and Michael Gschwind. Optimizing compiler for a cell processor. In *PACT*, 2005.

[7] Alexandre E. Eichenberger, Peng Wu, and Kevin O'Brien. Vectorization for SIMD architectures with alignment constraints. In *PLDI'04*, June 2004.

[8] S. Larsen and S. Amarasinghe. Exploiting superword level parallelism with multimedia instruction sets. In *PLDI*, 2000.

[9] Dorit Nuzman and Richard Henderson. Multi-platform auto-vectorization. In *CGO*, Washington, DC, USA, 2006.

[10] Dorit Nuzman, Ira Rosen, and Ayal Zaks. Auto-vectorization of interleaved data for SIMD. In *PLDI*, 2006.

[11] Yuan Zhao and Ken Kennedy. Scalarization on short vector machines. In *2005 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, March 20–22, 2005.