Supporting OpenMP on Cell

Kevin O'Brien, Kathryn O'Brien, Zehra Sura, Tong Chen and Tao Zhang

IBM T. J Watson Research

Abstract.

The Cell processor is a heterogeneous multi-core processor with one Power Processing Engine (PPE) core and eight Synergistic Processing Engine (SPE) cores. Each SPE has a directly accessible small local memory (256K), and it can access the system memory through DMA operations. Cell programming is complicated both by the need to explicitly manage DMA data transfers for SPE computation, as well as the multiple layers of parallelism provided in the architecture, including heterogeneous cores, multiple SPE cores, multithreading, SIMD units, and multiple instruction issue. There is a significant amount of ongoing research in programming models and tools that attempts to make it easy to exploit the computation power of the Cell architecture. In our work, we explore supporting OpenMP on the Cell processor. OpenMP is a widely used API for parallel programming. It is attractive to support OpenMP because programmers can continue using their familiar programming model, and existing code can be re-used. We base our work on IBM's XL compiler, which already has OpenMP support for AIX multi-processor systems built with Power processors. We developed new components in the XL compiler and a new runtime library for Cell OpenMP that utilizes the Cell SDK libraries to target specific features of the new hardware platform. To describe the design of our Cell OpenMP implementation, we focus on three major issues in our system: 1) how to use the heterogeneous cores and synchronization support in the Cell to optimize OpenMP threads; 2) how to generate thread code targeting the different instruction sets of the PPE and SPE from within a compiler that takes single-source input; 3) how to implement the OpenMP memory model on the Cell memory system. We present experimental results for some SPEC OMP 2001 and NAS benchmarks to demonstrate the effectiveness of this approach. Also, we can observe detailed runtime event sequences using the visualization tool Paraver, and we use the insight into actual thread and synchronization behaviors to direct further optimizations.

1 Introduction

The Cell Broadband Engine ™ (Cell BE) processor [1] is now commercially available in both the Sony PS3 game console and the IBM Cell Blade which represents the first product on the IBM Cell Blade roadmap. The anticipated high volumes for this non-

traditional "commodity" hardware continue to make it interesting in a variety of different application spaces, ranging from the obvious multi-media and gaming domain, through the HPC space (both traditional and commercial), and to the potential use of Cell as a building block for very high end "supercomputing" systems [8].

This first generation Cell processor provides flexibility and performance through the inclusion of a 64-bit multi-threaded Power Processor ™ Element (PPE) with two levels of globally-coherent cache and support for multiple operating systems including Linux. For additional performance, a Cell processor includes eight Synergistic Processor Elements (SPEs), each consisting of a Synergistic Processing Unit (SPU), a local memory, and a globally-coherent DMA engine. Computations are performed by 128-bit wide Single Instruction Multiple Data (SIMD) functional units. An integrated high bandwidth bus, the Element Interconnect Bus (EIB), glues together the nine processors and their ports to external memory and IO, and allows the SPUs to be used for streaming applications [2].

Data is transferred between the local memory and the DMA engine [3] in chunks of 128 bytes. The DMA engine can support up to 16 concurrent requests of up to 16K bytes originating either locally or remotely. The DMA engine is part of the globally coherent memory address space; addresses of local DMA requests are translated by a Memory Management Unit (MMU) before being sent on the bus. Bandwidth between the DMA and the EIB bus is 8 bytes per cycle in each direction. Programs interface with the DMA unit via a channel interface and may initiate blocking as well as non-blocking requests.

Programming the SPE processor is significantly enhanced by the availability of an optimizing compiler which supports SIMD intrinsic functions and automatic simdization [4]. However, programming the Cell processor, the coupled PPE and 8 SPE processors, is a much more complex task, requiring partitioning of an application to accommodate the limited local memory constraints of the SPE, parallelization across the multiple SPEs, orchestration of the data transfer through insertion of DMA commands, and compiling for two distinct ISAs. Users can directly develop the code for PPE and SPE, or introduce new the language extension [5].

In this paper we describe how our compiler manages this complexity while still enabling the significant performance potential of the machine. Our parallel implementation currently uses OpenMP APIs to guide parallelization decisions.

The remainder of the paper is laid out as follows: section two gives an overview of the compiler infrastructure upon which our work is based and presents the particular challenges of retargeting this to the novel features of the Cell platform. The next three sections of the paper look in more depth at each of these challenges and how we have addressed them, and section 6 presents some experimental results to demonstrate the benefit of our approach. We draw our conclusions in the last section.

2 System Overview

In our system, we use compiler transformations in collaboration with a runtime library to support OpenMP. The compiler translates OpenMP pragmas in the source code to

intermediate code that implements the corresponding OpenMP construct. This translated code includes calls to functions in the runtime library. The runtime library functions provide basic utilities for OpenMP on the Cell processor, including thread management, work distribution, and synchronization. For each parallel construct, the compiler outlines the code segment enclosed in the parallel construct into a separate function. The compiler inserts OpenMP runtime library calls into the parent function of the outlined function. These runtime library calls will invoke the outlined functions at runtime and manage their execution.

The compiler is built upon the IBM XL compiler [6][13]. This compiler has front-ends for C/C++ and Fortran, and shares the same optimization framework across multiple source languages. The optimization framework has two components: TPO and TOBEY. Roughly, TPO is responsible for high-level and machine-independent optimizations while TOBEY is responsible for low-level and machine-specific optimizations. The XL compiler has a pre-existing OpenMP runtime library and support for OpenMP 2.0 on AIX multiprocessor systems built with Power processors. In our work targeting the Cell platform, we re-use, modify, or re-write existing code that supports OpenMP as appropriate.

We encountered several issues in our OpenMP implementation that are specific to features of the Cell processor:

- Threads and synchronization: Threads running on the PPE differ in capability and processing power from threads running on the SPEs. We design our system to use these heterogeneous threads, and to efficiently synchronize all threads using specialized hardware support provided in the Cell processor.
- Code generation: The instruction set of the PPE differs from that of the SPE. Therefore, we perform code generation and optimization for PPE code separate from SPE code. Furthermore, due to the limited size of SPE local stores, SPE code may need to be partitioned into multiple overlaid binary sections instead of generating it as a large monolithic section. Also, shared data in SPE code needs to be transferred to and from system memory using DMA, and this is done using DMA calls explicitly inserted by the compiler, or using a software caching mechanism that is part of the SPE runtime library.
- Memory model: Each SPE has a small directly accessible local store, but it needs to use DMA operations to access system memory. The Cell hardware ensures DMA transactions are coherent, but it does not provide coherence for data residing in the SPE local stores. We implement the OpenMP memory model on top of the novel Cell memory model, and ensure data in system memory is kept coherent as required by the OpenMP specification.

In the following sections, we describe how we solve these issues in our compiler and runtime library implementation.

3 Threads and Synchronization

In our system, OpenMP threads execute on both the PPE and the SPEs. The master thread is always executed on the PPE. The master thread is responsible for creating threads, distributing and scheduling work, and initializing synchronization operations. Since there is no operating system support on the SPEs, this thread also handles all OS service requests. The functions specific to the master thread align well with the Cell design of developing the PPE as a processor used to manage the activities of multiple SPE processors. Also, placing the master thread on the PPE allows smaller and simpler code for the SPE runtime library, which resides in the space-constrained SPE local store. Different from the OpenMP standard, if required by users, the PPE thread will not participate in the work for parallel loops.

Currently, we always assume a single PPE thread and use the OMP_NUM_THREADS specification to be the number of SPE threads to use. Specifying the number of PPE and SPE threads separately will need an extension of the OpenMP standard. The PPE and SPE cores are heterogeneous, and there may be significant performance mismatch between a PPE thread and an SPE thread that perform the same work. Ideally, the system can be built to automatically estimate the difference in PPE and SPE performance for a given work item, and then have the runtime library appropriately adjust the amount of work assigned to different threads. We do not have such a mechanism yet, so we allow users to tune performance by specifying whether or not the PPE thread should participate in executing work items for OpenMP work-share loops or work-share sections.

We implement thread creation and synchronization using the Cell Software Development Kit (SDK) libraries [7]. The master thread on the PPE creates SPE threads only when a parallel structure is first encountered at runtime. For nested levels of parallelism, each thread in the outer parallel region sequentially executes the inner parallel region. The PPE thread schedules tasks for all threads, using simple block scheduling for work-share loops and sections. The work sections or loop iterations are divided into as many pieces as the number of available threads, and each thread is assigned one piece. More sophisticated scheduling is left for future work.

When an SPE thread is created, it performs some initialization, and then loops waiting for task assignments from the PPE, executing those tasks, and then waiting for more tasks, until the task is to terminate. A task can be the execution of an outlined parallel region, loop or section, or performing a cache flush, or participating in barrier synchronization. There is a task queue in system memory corresponding to each thread. When the master thread assigns a task to a thread, it writes information about the task to the corresponding task queue, including details such as the task type, the lower bound and upper bound for a parallel loop, and the function pointer for an outlined code region that is to be executed. Once an SPE thread has picked up a task from the queue, it uses DMA to change the status of the task in the queue, thus informing the master thread that the queue space can be re-used.

The Cell processor provides special hardware mechanisms for efficient communication and synchronization between the multiple cores in a Cell system. The Memory Flow Controller (MFC) for each SPE has two blocking outbound mailbox queues, and one non-blocking inbound mailbox queue. These mailboxes can be used for efficient communication of 32-bit values between cores. When the master thread

assigns tasks to an SPE thread, it uses the mailbox to inform the SPE of the number of tasks available for execution. Each SPE MFC also has an atomic unit that implements atomic DMA commands and provides four 128-byte cache lines that are maintained cache coherent across all processors. We use atomic DMA commands for efficient implementation of OpenMP locks, barriers[1], and cache flush operations.

4 Code Generation

Figure 1 illustrates the code generation process of the Cell OpenMP compiler. The compiler separates out each code region in the source code that corresponds to an OpenMP parallel construct (including OpenMP parallel regions, work-share loops or work-share sections, and single constructs), and outlines it into a separate function. The outlined function may take additional parameters such as the lower and the upper bounds of the loop iteration for parallel loops. In the case of parallel loops, the compiler further transforms the outlined function so that it only computes from the lower bound to the upper bound. The compiler inserts an OpenMP runtime library call into the parent function of the outlined function, and passes a pointer to the outlined function code into this runtime library function. During execution, the runtime function will indirectly invoke the outlined function. The compiler also inserts synchronization operations such as barriers when necessary.

Due to the heterogeneity of the Cell architecture, the outlined functions containing parallel tasks may execute on both the PPE and the SPEs. In our implementation, we clone the outlined functions so that there is one copy of the function for the PPE architecture, and one for the SPE architecture. We perform cloning during TPO link-time optimization when the global call graph is available, so we can clone the whole sub-graph for a call to an outlined function when necessary. We mark the cloned function copies as PPE and SPE procedures, respectively. In later stages of compilation, we can apply machine-dependent optimizations to these procedures based on their target architecture. Auto-simdization is one example. SPE has SIMD units that can execute operation on 128 byte data with one instruction. After cloning, the code for SPE will undergo the auto-simdization to transform scalar code into SIMD code for SPE, while the PPE has totally different SIMD instructions. In other words, we choose to clone the functions to enable more aggressive optimizations.

When a PPE runtime function in the master thread distributes parallel work to an SPE thread, it needs to tell the SPE thread what outlined function to execute. The PPE runtime function knows the function pointer for the PPE code of the outlined function to execute. However, the SPE thread needs to use the function pointer for the SPE code of the same outlined function. To enable the SPE runtime library to determine correct function pointers, the compiler builds a mapping table between corresponding PPE and SPE outlined function pointers, and the runtime looks up this table to determine SPE code pointers for parallel tasks assigned by the master thread.

At the end of TPO, procedures for different architectures are separated into different compilation units, and these compilation units are processed one at a time by the TOBEY backend. The PPE compilation units are processed as for other

[1] We thank Daniel Brokenshire (IBM Austin) for his implementation of barriers on Cell.

architectures and need no special consideration. However, if a single large SPE compilation unit is generated, it may result in SPE binary code that is too large to fit in the small SPE local store all at once. In fact, we observe this to be the case for many benchmark programs. For OpenMP, one way to mitigate this problem is to place all the code corresponding to a given parallel region in one SPE compilation unit, and generate as many SPE compilation units as there are parallel regions in the program. Using this approach, we can generate multiple SPE binaries, one for each SPE compilation unit. We can then modify the runtime library to create SPE threads using a different SPE binary on entry to each parallel region. However, there are two drawbacks to using this approach: first, an individual parallel region may still be too large to fit in SPE local store, and second, we observe through experiments that the overhead for repeatedly creating SPE threads is significantly high.

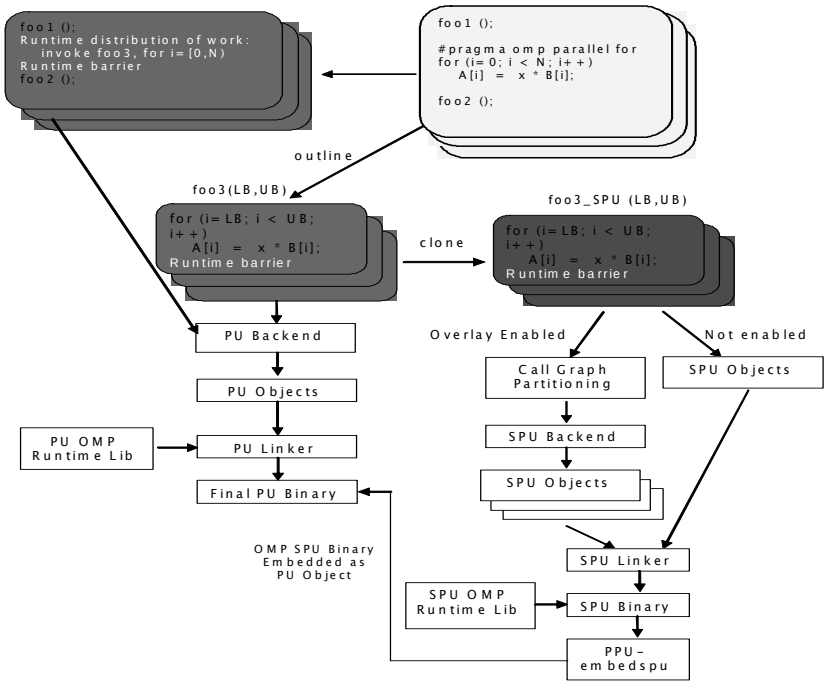


Figure 1 Code generation process

To solve the SPE code size problem, we rely on the technique of call graph partitioning and code overlay. We first partition the sub-graph of the call graph corresponding to SPE procedures into several partitions. Then we create a code overlay for each of these call graph partitions. Code overlays share address space and do not occupy local storage at the same time. Thus, the pressure on local storage due

to SPE code is greatly reduced. To partition the call graph, we weight each call graph edge by the frequency of this edge. The frequency can be obtained by either compiler static analysis or profiling. Then we apply the maximum spanning tree algorithm to the graph. Basically, we process edges in the order of their weight. If merging the two nodes of the edge does not exceed a predefined memory limitation, we merge those two nodes, update the edge weights, and continue. When the algorithm stops, each merged node represents a call graph partition comprising all the procedures whose nodes were merged into that node. Thus, the result is a set of call graph partitions. Our algorithm is a simple greedy algorithm that can be further optimized. After call graph partitions are identified, we utilize SPU code overlay support introduced in Cell SDK 2.0 and place the procedures in each call graph partition into a separate code overlay.

After the TOBEY backend generates an SPE binary (either with or without code overlays), we use a special tool called ppu-embedspu to embed the SPE binary into a PPE data object. This PPE data object is then linked into the final PPE binary together with other PPE objects and libraries. During execution, when code running on the PPE creates SPE threads, it can access the SPE binary image embedded into the PPE data object, and use this SPE image to initialize the newly created SPE threads.

5 Memory model

OpenMP specifies a relaxed-consistency, shared-memory model. This model allows each thread to have its own temporary view of memory. A value written to a variable, or a value read from a variable, can remain in the thread's temporary view until it is forced to share memory by an OpenMP flush operation. We find that such a memory model can be efficiently implemented on the Cell memory structure.

In the Cell processor, each SPE has just 256K directly accessible local memory for code and data. We only allocate private variables accessed in SPE code to reside in the SPE local store. Shared variables reside in system memory, and SPE code can access them through DMA operations. We use two mechanisms for DMA transfers: static buffering and compiler-controlled software cache. In both mechanisms, the global data may have a local copy in the SPE local store. The SPE thread may read and write the local copy. This approach conforms to the OpenMP relaxed memory model and takes advantage of the flexibility afforded by the model to realize memory system performance.

Some references are *regular* references from the point-of-view of our compiler optimization. These references occur within a loop, the memory addresses that they refer to can be expressed using affine expressions of loop induction variables, and the loop that contains them has no loop-carried data dependence (true, output or anti) involving these references. For such regular reference accesses to shared data, we use a temporary buffer in the SPE local store. For read references, we initialize this buffer with a DMA *get* operation before the loop executes. For write references, we copy the value from this buffer using a DMA *put* operation after the loop executes. The compiler statically generates these DMA get and put operations. The compiler also transforms the loop structure in the program to generate optimized DMA operations for references that it recognizes to be regular. Furthermore, DMA

operations can be overlapped with computations by using multiple buffers. The compiler can choose the proper buffering scheme and buffer size to optimize execution time and space [9].

For irregular references to shared memory, we use a compiler-controlled software cache to read/write the data from/to system memory. The compiler replaces loads and stores using these references with instructions that explicitly look up the effective address in a directory of the software cache. If a cache line for the effective address is found in the directory (which means a cache hit), the value in the cache is used. Otherwise, it is a cache miss. For a miss, we allocate a line in the cache either by using an empty line or by replacing an existing line. Then, for a load, we issue a DMA get operation to read the data from system memory. For stores, we write the data to the cache, and maintain dirty bits to record which byte is actually modified. Later, we write the data back to system memory using a DMA put operation, either when the cache line is evicted to make space for other data, or when a cache flush is invoked in the code based on OpenMP semantics.

We configure the software cache based on the characteristics of the Cell processor. Since 4-way SIMD operations on 32-bit values are supported in the SPE and we currently use 32-bit memory addresses, we use a 4-way associative cache that performs the cache lookup in parallel. Also, we use 128-byte cache lines since DMA transfers are optimal when performed in multiples of 128 bytes and aligned on a 128-byte boundary. If only some bytes in a cache line are dirty, when the cache line is evicted or flushed, the data contained in it must be merged with system memory such that only the dirty bytes overwrite data contained in system memory. One way to achieve this is to DMA only the dirty bytes and not the entire cache line. However, this may result in small discontinuous DMA transfers, and exacerbated by the alignment requirements for DMA transfers, it can result in poor DMA performance. Instead, we use the support for atomic updates of 128-byte lines that is provided in the SPE hardware to atomically merge data in the cache line with data in the corresponding system memory, based on recorded dirty bits.

When an OpenMP flush is encountered, the compiler guarantees that all data in the static buffers in local store has been written back into memory, and that existing data in the static buffers is not used any further. The flush will also trigger the software cache to write back all data with dirty bits to system memory, and to invalidate all lines in the cache.

When an SPE thread uses DMA to get/put data from/to the system memory, it needs to know the address of the data to be transferred. However, global data is linked with the PPE binary and is not directly available in SPE code. The Cell SDK provides a link-time mechanism called CESOF, which makes available to the SPE binary the addresses of all PPE global variables once these addresses have been determined. We also use a facility similar to CESOF when generating SPE code.

Besides global data, an SPE thread may need to know the address of data on the PPE stack when, in source code, the procedure executing in the SPE thread is nested within a procedure executing in a PPE thread, and the SPE procedure accesses variables declared in its parent PPE procedure. Though C and Fortran do not support nested procedures (C++ and Pascal do), this case can occur when the compiler performs outlining. For example, in Figure 1, if the variable "x" were declared in the procedure that contains the parallel loop, after outlining, the declaration of "x"

becomes out of the scope of the outlined function. To circumvent this problem, the compiler considers each outlined function to be nested within its parent function. The PPE runtime, assisted by compiler transformations, ensures that SPE tasks that will access PPE stack variables are provided with the system memory address of those stack variables.

6 Experimental Results

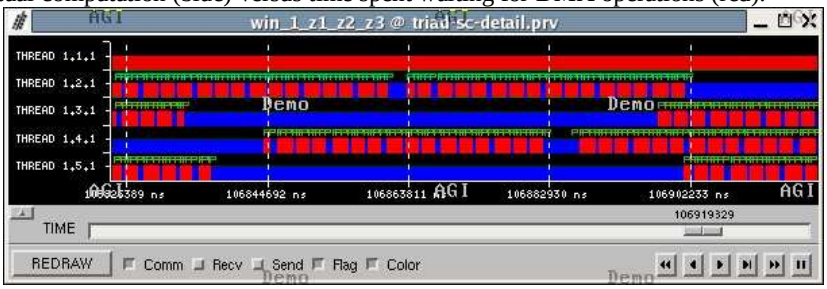
We compiled and executed some OpenMP test cases on a Cell blade that has both the PPU and the SPUs running at 3.2 GHz, and has a total of 1GB main memory. All our experiments used one Cell chip: one PPE and eight SPEs. The test cases include several simple streaming applications, as well as the standard NAS [10] and SPEC OMP 2001 [11] benchmarks. To observe detailed runtime behavior of applications, we instrumented the OpenMP runtime libraries with Paraver [12], a trace generation and visualization tool.

Figure 2 shows the PPE and SPE thread behavior for a small test case comprising of one parallel loop that is repeatedly executed one hundred times. The PPE thread is assigned no loop iterations, and is responsible only for scheduling and synchronization. The figure shows a high-level view of one complete execution of the parallel loop. The first row corresponds to the PPE thread and the remaining rows correspond to SPE threads. Blue areas represent time spent doing useful computation, yellow areas represent time spent in scheduling and communication of work items, and red areas represent time spent in synchronization and waiting for DMA operations to complete. We can clearly identify the beginning and end of one instance of a parallel loop execution from the neatly lined up set of red synchronization areas that indicate the implicit barriers being performed at parallel region boundaries, as specified by the OpenMP standard.

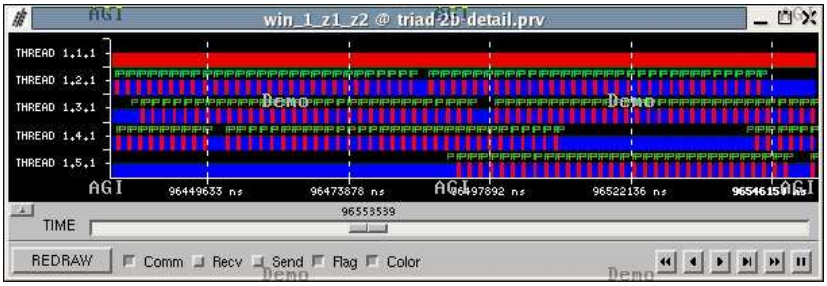
Figure 2 PPE and SPE thread

If we zoom into a portion of the SPE useful computation shown in Figure 2, we can see in greater detail the time actually spent in computing and the time spent in waiting for data transfers. Figure 3(a) shows this detail for a segment of execution when only the software cache is used for automatic DMA transfers. We see many areas in the SPE execution that are dominated by waiting for DMA operations to complete (red areas), and the need to optimize this application is evident. Figure 3(b)

shows a similar segment of execution for the same application when it has been optimized using static buffering. We observe the improved ratio for time spent doing actual computation (blue) versus time spent waiting for DMA operations (red).



(a) software cache only



(b) optimized with static buffer

Figure 3 Data transfer with software cache and static buffer

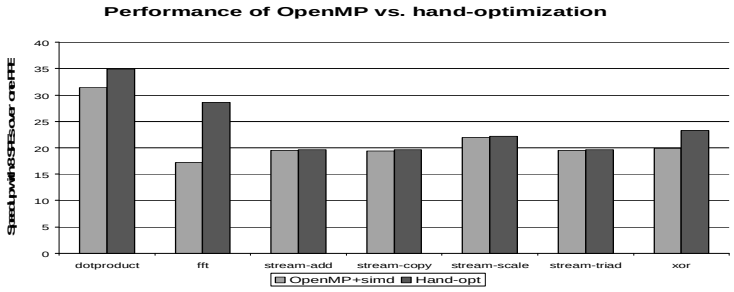


Figure 4 Performance comparison with manually optimized code

To evaluate our approach, we first tried some simple stream benchmarks, comparing the performance of code generated by our compiler with the performance of manually written code that was optimized using Cell SDK libraries and SIMD instructions. The performance comparison is shown in Figure 4. The speedups shown in this figure are the ratios of execution time when using 8 SPUs and execution time when only using the PPE. Both SIMD units and multiple cores contribute to the

speedups observed. We observe that our OpenMP compiler can achieve as good a performance (except for *fft*) as the highly optimized manual code on these stream applications. *fft* performs poorly in comparison because auto-simdization cannot handle different displacements in array subscript expressions for different steps in the FFT computation. Manual code performs slightly better on *dotproduct* and *xor* because the compiler does not unroll the loop an optimal number of times.

We also experimented with some applications from the NAS and Spec OMP 2001 benchmark suites. We report speedups of the whole program normalized to one PPU and one SPU respectively in Figure 5.

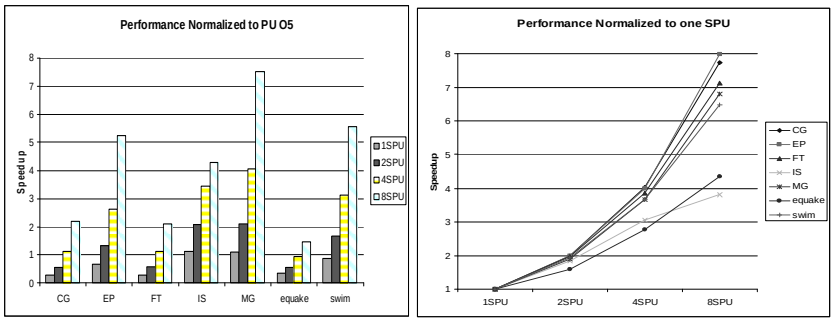


Figure 5 Performance of benchmarks

Compared to the performance of one PPU, many applications show significant speedup with our compiler on 8 SPUs. The performance of some others is unsatisfactory, or even quite bad (such as *equake*, *CG* and *FT*). We analyzed the benchmarks and traced compiler transformations to determine the reasons for the performance shortfall. We identified two main reasons for it:

1. Limitations in our current implementation: A common reason for bad performance is the failure of static buffer optimization. Our current implementation cannot handle references complicated by loop unrolling, done either by the user or the compiler. The absence of precise alias information also prevents static buffering from being applied in many test cases. We are working on improving our compiler.
2. Applications unsuitable for the Cell architecture: *FT* and *CG* contain irregular discontinuous data references to main memory. DMA data transfers become the bottleneck in such cases.

The performance normalized to one SPU shows the scalability. All of them, except IS and equake, show good scalability with speedup more than 6 on 8 SPUs. The reason that IS did not scale up well is that IS contains some computations in either master pragma or critical pragma. Those computations are done sequentially. For equake, some loops are not parallelized due to the current implementation limitation of compiler. Therefore, its speedup of 8 SPUs is only above 4.

7 Conclusions

In this paper, we describe how to support OpenMP on the Cell processor. Our approach allows users to simply reuse their existing OpenMP applications on the powerful Cell Blade, or easily develop new applications with the OpenMP API without worrying about the hardware details of the Cell processor. We support OpenMP by orchestrating compiler transformations with a runtime library that is tailored to the Cell processor. We focus on issues related to three topics: thread and synchronization, code generation, and the memory model. Our compiler is novel in that it generates a single binary that executes across multiple ISAs and multiple memory spaces.

Experiments with simple test cases demonstrate that our approach can achieve performance similar to that of manually written and optimized code. We also experimented with some large, complex benchmark codes. Some of these benchmarks show significant performance gains. Thus, we demonstrate that it is feasible to extract high performance on a Cell processor using the simple and easy-to-use OpenMP programming model. However, we need to further improve our compiler implementation for improved performance on a wider set of application programs.

References

1. D. P. et al. The design and implementation of a first-generation CELL processor. IEEE International Solid-State Circuits Conference (ISSCC), February 2005
2. M. Gordon et. al. Exploiting Coarse-Grained Task, Data, and Pipeline Parallelism in Stream Programs, International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), October, 2006
3. M. Kistler, M. Perrone, and F. Petrini. CELL multiprocessor communication network: Built for Speed. IEEE Micro, 26(3), May/June 2006
4. A. E. et al. Vectorization for SIMD architecture with Alignment Constraints, Conference on Programming Language Design and Implementation (PLDI) 2003.
5. P. Bellens et. al. CellSc: a Programming Model for the Cell BE Architecture, SC, 2006
6. IBM xl compiler for Cell: http://www.alphaworks.ibm.com/tech/cellcompiler
7. SDK for Cell: http://www-128.ibm.com/developerworks/power/cell/
8. Samuel Williams, et al. The Potential of the Cell Processor for Scientific Computing. Conference on Computing Frontiers, 2006
9. T. Chen. et al. Optimizing the use of static buffers for DMA on a CELL chip. Workshop on Language and Compiler for Parallel Computing (LCPC), 2006
10. NAS parallel benchmarks: http://www.nas.nasa.gov/Resources/Software/npb.html
11. Spec OMP benchmarks: http://www.spec.org/
12. Paraver: http://www.cepba.upc.es/paraver/
13. A. E. et al. Optimizing Compiler for the Cell Processor. Conference on Parallel Architecture and Compiler Techniques (PACT), 2005