Optimizing Compiler for a Heterogeneous Multi-core Processor with Non-uniform Memory Access

Junggyu Park*, Daehyun Cho, Jongmin Han, Jun Sung Park, Byungchang Cha, and Hyo Jung Song

Software Laboratories, Samsung Electronics,
Maetan3-dong, Youngtong-gu, Suwon, Kuingki, South Korea

Abstract

Multiprocessor-System-on-a-Chips (MPSoCs) have been widely used in today's high performance embedded systems, such as network processors, mobile phones, and MP3 players. Among them, the Cell Broadband Engine (CBE) processor is for high-end computing, targeting such application as multimedia streaming, game applications as in PlayStation 3, and other numerically intensive workloads. It can provide both performance and flexibility to such devices as digital TV by allowing software codec to run as fast as hardware codec. Developing optimized software for the CBE, however, is very difficult because of its complex architecture and multilevel heterogeneous parallelism. In this paper, we present a GCC-based compiler that we developed to address the above problem. For easy of parallel programming, we support OpenMP for heterogeneous multiprocessor. It is different from that of original GCC in that parallel region and serial region are compiled by different target compilers but linked together for final executables. To deal with the data sharing across heterogeneous multi-processors of the CBE, we have come up with software cache. This architecture is platform neutral so that it can be easily ported to other heterogeneous multi-processors. We also present compiler techniques designed to exploit performance potential of the CBE and overcome its constraints, such as instruction scheduling and bundling, auto-SIMD for utilizing data-level parallelism, and auto-overlay. Since our first priority customer is streaming and media applications, we used codecs and some proprietary software module for improving quality of image as main benchmarks, though other general benchmarks are also tested. The results of evaluation indicate that significant speedup can be achieved. Automatic parallelization significantly reduces programming efforts with little performance degradation compared with manual parallelization. Our compiler optimization techniques shown better performance for SPE processor of the CBE, achieved about 26 % improvement in H.264 codec and up to 107 % improvement in other cases. We also compared this result with the result of XLC for SPE, which showed that our compiler outperforms in most cases.

Keyword

Compiler, Multi-core, Optimization, Cell, Parallelizing Compiler, SIMDization, Instruction Scheduling, OpenMP

1. INTRODUCTION

Multiprocessor-System-on-a-Chips (MPSoCs) have been widely used in today's high performance embedded systems, such as network processors, mobile phones, and MP3 players. OMAP from Texas Instruments, Nexperia from Philips, MDSP from Cradle[6], and Cell Broadband Engine (CBE) from Sony/Toshiba/IBM[5] show how MPSoCs can be used successfully in the mobile, portable, or CE devices. Among them, the CBE processor is for high-end computing, targeting such application as multimedia streaming, game applications as in PlayStation3, and other numerically intensive workloads. It can provide both performance and flexibility to such devices as digital TV by allowing software codec to run as fast as hardware codec.

Developing optimized software for the CBE, however, is very difficult because of its complex architecture and multilevel heterogeneous parallelism - it is an asymmetric multiprocessor system with one PowerPC core and eight synergistic processor elements (SPEs) for computation intensive workloads, and each SPE consists of a SIMD processor having two pipelines designed for streaming workloads, a local memory, and a globally coherent DMA engine. Many attempts have been made for the easy development of fast software against the complexity of the CBE. Toshiba has introduced media framework which addressed thread level parallelism [3]. But optimizations by instruction level parallelism (ILP) and data parallelism should be done by optimizing compiler or manual vectorization. RapidMind provides C++ library for parallelization at runtime [7], but it is not easy to learn and has problems in performance and memory foot prints. Also, transforming programs in C to RapidMind is very difficult. IBM introduced compiler techniques to address all of the problems above [9]. What we are trying to do is similar to this, but we used GCC and focused more on multimedia workloads, especially codec and image processing modules.

In this paper, we present GCC-based compiler that we developed to address the above problem. Our contributions in this paper are a set of compiler techniques for both high performance and easy of parallel programming. For easy of parallel programming, we support OpenMP standard for heterogeneous multiprocessor. GCC now supports OpenMP (GOMP [19]) from its release of 4.2, but it is for homogeneous multiprocessors with cache coherent shared memory system. XLC compiler from IBM supports single source compilation for the CBE by using OpenMP [9], but it has two paths for PPE and SPE processor that is almost impossible for GCC. Our approach is different from that of IBM in that parallel and serial region of an OpenMP program are compiled by SPE and PPE compiler respectively (controlled by a compiler driver), then linked together for final executables. This does not need much modification in GCC, which is very important in opensource-based software project in that opensource software changes frequently and maintaining source code is very difficult. To cope with the data sharing across heterogeneous multi-processors of the CBE, we have come up with software cache and a new data type with consideration of minimal data transfer overheads. This architecture is not specific to the CBE so it can be easily ported to other heterogeneous multi-processors with little modification.

We also present compiler techniques designed to exploit performance potential of the SPE and overcome its constraints, such as instruction scheduling and bundling, auto-SIMD for utilizing data-level parallelism, and auto-overlay. We mainly focused on instruction scheduling, such as Swing Modulo Scheduling (SMS) and bundling. Because of large number of register files, we do not care much about register pressure in swing modulo scheduling (SMS, a software pipelining approach), but instead care about blocking situation that often makes SMS fail. Aggressive if-conversion is performed to reduce big penalty of branch, even integrated with SMS. SPE specific features such as memory alignment, 2 pipelines, branch hint, 64-byte unit of instruction fetch are also considered. Automatically generating SIMD code that fully utilizes the functional units of the SPE is supported by GCC, and we ported it to work on the SPE. To overcome the size limit of 256KB for code and data, our SPE compiler supports automatic management of overlay that is otherwise hard to do manually.

Since our first priority customer is streaming and media applications, we used H.264 codec, mpeg2 codec, and some proprietary software module for improving quality of image as main benchmarks, though other general benchmarks are also tested. We evaluated the performance of our compiler using those benchmarks, and results indicate that significant speedup can be achieved. Automatic parallelization significantly reduces programming efforts with little performance degradation compared with manual parallelization. Our compiler optimization techniques shown better performance for SPE processor of the CBE, achieved about 26 % improvement in H.264 codec and up to 107 % improvement in other cases. We also compared this result with the result of XLC for SPE, which showed that our compiler outperforms in most cases.

This paper is organized as follows. Next section discusses on the background, while auto-parallelization techniques explained in section 3. In section 4, we describe an in depth description of the SPE specific optimizations. Experimental results are discussed in section 5, followed by related work in section 6. Finally section 7 concludes the paper.

2. BACKGROUND

2.1. Cell Broadband Engine Architecture

The CBE is a relatively new architecture, which uses multilevel heterogeneous parallelism in order to achieve maximum performance of applications. It is designed with such applications in mind as games and media streaming applications which have highly parallel code such as game physics and image processing, providing both flexibility and performance to them.

The CBE is composed of 1 64-bit multithreaded PowerPC Processor Element (PPE) and 8 Synergistic Processor Elements (SPEs). The PPE have two levels of globally-coherent cache and is designed to run operating system and to control task distribution between 8 SPE elements. On the other hand, the SPE is designed to run computation intensive, small, and preferably control reduced tasks. An SPE consists of a SIMD processor having 128 128-bits register file, local non-coherent memory, and globally-coherent DMA engine. Nearly all instructions provided by the SPE operate in a SIMD fashion on 128 bits of data representing either 2 64-bit double float or long integers, 4 32-bit single float or integers, 8 16-bit shorts, or 16 8-bit bytes. An SPE have two pipelines - even pipeline for arithmetic and logic computation, and odd pipeline for load/store and branch instruction. Two pipelines can issue instructions at the same time if the instructions are independent. Because the SPE do not have branch prediction hardware (instead provides branch hint instruction) and pipeline is very deep, care must be taken not to cause severe branch penalty (18 cycles) [14].

The SPE has 256K bytes local memory (LS) which supports 16-byte accesses for memory instructions and 128-byte accesses for instruction fetch and DMA transfers. Because LS is single ported, instruction fetch, memory instructions, and DMA compete for the same port. Instruction fetches occur during idle memory cycles in 64 byte address alignment. A DMA engine is used to transfer data at high speed between an external memory and the SPE local memories, which should be explicitly initiated by programmers [14].

Our work is an attempt to address difficulties in developing efficient software on the CBE. The performance and flexibility supported by the CBE has very important meaning to devices like HDTV/IPTV, since it enables software implementations used in place where built-in hardware implementations have been used. For example, fast software codec can be plugged into the device for new media type, which means low cost and easy of maintenance. For this to happen, however, we need to fully utilize the performance capability of the CBE, especially the SPE. Thread parallelization among the PPE and SPEs, and data/instruction parallelization in a SPE are our main concerns for performance, which we are trying to address by compiler technology.

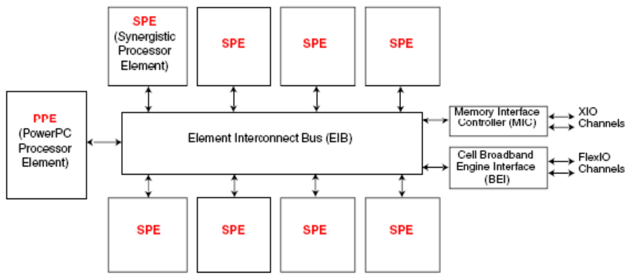
SPE
(Synergistic Processor Element)
SPE
SPE
SPE
PPE
(PowerPC Processor Element)
Element Interconnect Bus (EIB)
Memory Interface Controller (MIC)
XIO Channels
Cell Broadband Engine Interface (BEI)
FlexIO Channels
SPE
SPE
SPE
SPE

Fig. 1. Cell Broadband Engine Architecture.

2.2. Programming for the CBE

In the existing programming model for the CBE architecture, the heterogeneous processor cores, SPEs and the PPE, and the non-uniform memory accesses of the architecture are visible to application programmers. Efficient manual programming of the Cell BE architecture can be a complex task. To make use of the best features of the PPE and SPE cores for an application, programmers must manually partition the application into separate code segments and use the compiler that targets the appropriate ISA [14, 9].

Both the VMX unit of the PPE and the SPE are SIMD processors. Efficient programming therefore requires exploiting parallelism within the SPE and the VMX unit by using the rich set of vector intrinsics that are provided for them. Intrinsics appear to the programming language as functions that mimic the behavior of each SIMD instruction in the target architecture. To further exploit the coarse-grained inter-processor parallelism throughout the Cell BE system, the programmer may choose to partition his application into tasks or parallel work units that may be executed on the SPE by using a pipeline or parallel-execution model.

The resulting PPE and SPE code segments must work together cooperatively and must explicitly manage the transfer of code and data between system memory and the limited SPE local stores. Data and code transfer can be done via DMA. Optimizing data transfer to overlap communication and computation may involve manually programming multi-buffering schemes that take into account the optimal size and numbers of local data buffers and that select the best placement of data transfer requests. The extent to which parallelism is deployed in the application also influences the data transfer decisions.

The final step to effect the seamless execution of a CBE program requires using the SPE linker and an embedding tool to generate a PPE executable that contains the SPE binary embedded within the data section. This object is then linked, using a PPE linker, with the runtime libraries which are required for thread creation and management, to create an executable for the CBE program.

2.3. OpenMP

The OpenMP (Open Multi-Processing) is an application programming interface that supports multi-platform shared memory multiprocessing programming in C/C++ and Fortran on many architectures. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior [20].

To write a parallel program in OpenMP, programmers insert OpenMP directives (i.e., #pragma omp in C/C++) at key locations (usually loops) in the source code. OpenMP compiler interprets the directives and check whether code region noted by the directive can be run in parallel. If so, the compiler generates parallel program code and library calls to parallelize the code region. Creating and managing threads are hidden under the compiler and runtime system.

The current standard is OpenMP 2.5, which was defined by a language committee comprising users and vendors, and is available on many current high performance platforms. As it is widely available today, GCC includes it from GCC release 4.2.0 [19]. Although this work is based on GCC and it has already GOMP, we cannot use it as it is because of some challenging features of the CBE such as heterogeneous processor type and explicit memory transfer.

3. AUTOMATIC PARALLELIZATION

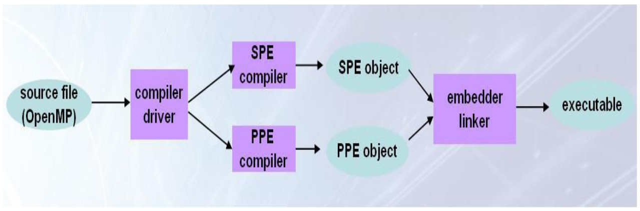

Fig. 2. Proposed compiler architecture

3.1 Overview

To exploit performance potential of the CBE, we first need to distribute workloads to all SPEs so that no SPE is idle. As shown in the section 2.2, doing this manually involves many uneasy tasks such as thread and DMA programming. OpenMP compiler frees programmers from these burdens by automatically generating parallel programs with workloads distributed evenly. To implement OpenMP compiler for the CBE using GCC, there are following issues:

- The compiler should generate both PPE and SPE processor, and link them together to generate a CBE executable. GOMP (existing GCC OpenMP implementation) is for homogeneous multiprocessors with cache coherent shared memory system.
- Argument passing and data sharing between PPE-side and SPE-side functions should be transformed to data transfer (DMA) instructions. This can cause significant performance penalty if implemented poorly.

- Size of LS in SPE is only 256KB which includes code and data. Automatic parallelization should take care of this size limitation.

XLC compiler from IBM supports OpenMP for the CBE [9], but it has two paths for PPE and SPE processor in a compiler. But that is almost impossible to implement in GCC, also not reasonable to modify GCC like that since open-source software changes very frequently and maintaining it is accordingly very difficult. Instead, we take a two-compiler approach: parallel region of an OpenMP program is compiled by SPE compiler and serial region by PPE compiler, and then linked together for final executables (refer to Fig. 2). This does not need much modification in the GCC (details of modification is explained below in GOMP compilation pass). Execution of PPE/SPE compilers and linking to the final executable is controlled by a compiler driver. To handle the data sharing and argument passing between heterogeneous multi-processors of the CBE, we have come up with software cache and a new data type for shared pointer type. This architecture is not specific to the CBE but portable so that it can be easily ported to other heterogeneous multi-processors with little modification (GCC is portable and already ported to many processor architectures).

There are two passes in GOMP compiler to implement OpenMP: lowering pass and expanding pass. Lowering of OpenMP parallel and work-sharing constructs proceeds in two phases. The first phase scans functions in order to look for OpenMP statements and then variables that must be replaced to satisfy data sharing clauses. The second phase expands code for the constructs. During the expanding pass, the flow graph is scanned for parallel regions, which are then moved to a new function, to be invoked by the thread library.

To implement single source code compilation for OpenMP on CBE using GOMP, we split the compilation process into PPE compilation path and SPE compilation path (refer to Fig. 2 and 3). During PPE compilation path, GCC parses OpenMP source program into GIMPLE tree (high-level intermediate code in GCC), then converts OpenMP directives into explicit calls to the runtime library (libgomp) and data marshalling to implement data sharing and copying clauses. Parallel regions are outlined into functions. Related information is updated, e.g. the call graph. After that, the whole GIMPLE tree undergoes all the remaining passes in GCC and output the assembly code for PPE. In this way, the overall structure of the PPE code is not going to be changed. During SPE compilation, we just keep the outlined parallel functions and the functions called by them (the dynamic extent of parallel construct), and discard all other functions.

Fig. 3 shows the outlining. In case target intrinsics are needed for better performance (i.e., SIMD intrinsics in SPE and PPE), this should be guarded by `#ifdef TARGET` expression since a target intrinsic function can compiled only by its target compiler. Due to limited size of SPE LS, we need code overlay mechanism to fully support OpenMP programming on CBE. Since a parallel region can have function calls defined in different source files, the compiler driver keeps information of function calls and its definition (i.e., object file, overlay, and function pointer) while compilation. Details of overlay are explained in section 4.

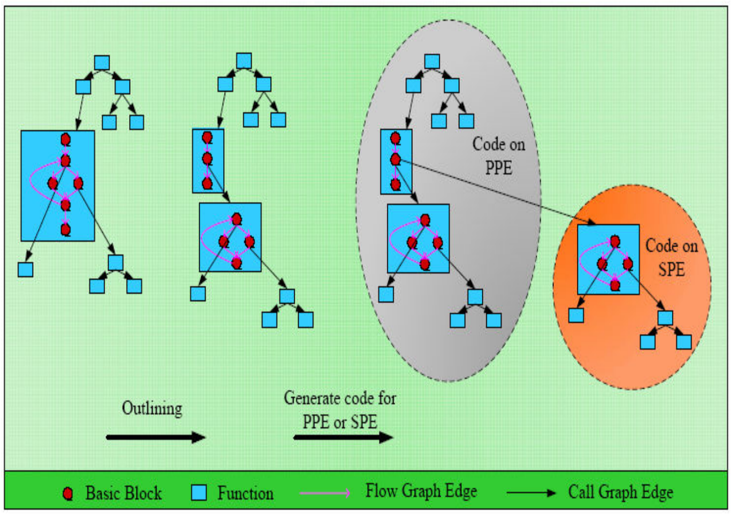

Fig. 3. Parallel region outlining.

In a SPE program, a pointer variable points to an address in LS, not in main memory. For a pointer to main memory, a new keyword is introduced a shared pointer variable: `sharedp`. This allows a pointer variable in a SPE program to point to main memory address instead of LS address. This keyword can be used in a function declaration, variable declaration and definition. Also, in case where a `struct` has both shared and non-shared member variable, then we can describe it like this:

```
typedef struct _ROUNDSAMPLING_ {
      int srcWidth;
      int srcHeight;
      sharedp unsigned char* srcBuffer;
} RoundSample;
```

3.2 SW cache

When compiling SPE code, the compiler can identify references to data in main storage that have not been optimized using explicit DMA transfers, and the compiler can insert code to invoke the software-managed cache mechanism before each such reference. The call to the software cache takes a main-storage address and returns an address in the SPE LS that contains the corresponding data. To support the software cache, a separate directory is maintained in each SPE. The compiler replaces a subset of load and store instructions with instructions that explicitly look up the address of the data in a directory. If the directory lookup indicates cache hit (the data is present in the LS), the address of the requested variable cached in the LS is computed and the load or store proceeds using this LS location. Otherwise, a miss-handler is invoked to get the requested data transferred from main storage, and then the directory is updated.

Cache coherence is enforced at synchronization point (e.g, barrier, lock, etc). It's not necessary to keep coherence at other points - data race conditions. Between two synchronization points, if two threads write to the same variable, the result will be random on any SMP machines, even they support sequential consistency. So, we only need to worry about coherence at synchronization point. The big challenge is false-sharing. In order to solve this problem, we associate a dirty bit with each byte of data, and we "merge" the dirty byte with the value in main memory [21]. Therefore, we guarantee no false sharing at byte level.

Performance is a major issue for software cache. Managing software cache is pure overheads and is not a good choice if array of (image) data is cached and its element is accessed sequentially. If the compiler can identify data access patterns, we should use static buffer [18] instead of software cache. Thus, we used static buffer which pre-fetches data for regular accesses with double buffering. In other cases, we used software cache. Code generation of static buffer is not supported automatically by compiler at the time of writing, but will be available soon. Our current implementation of software cache is similar to that of IBM [9] (i.e., set-associative, 32-bit address, cache line size is 128B), but we have 1-way associative cache instead of 4-way. This is because 1-way requires less computation and code size (for performance, software cache code is inlined) than 4-way does. In this way, we reduced significant overheads from lookup and miss-handling.

3.3 OpenMP runtime

We implemented OpenMP runtime for CBE based on GOMP. In our implementation, we propose several novel approaches.

First, master thread and team threads run on PPE and SPEs respectively. Master thread which is on PPE starts to run and creates SPE threads as team threads when it encounters the first PARALLEL construct. Each team thread starts to execute the execution handler and waits a task which is outlined parallel region by master thread. After assignment from master thread, each team thread executes the task, feedbacks the result and waits for the next task assignment if there exists. All the runtime data structures are kept on main memory. These are copied to LS of each SPE or are shared via software cache. To synchronize threads, we use the communication mechanisms of CBE such as mailboxes and DMA transfers.

Second, we provide resource efficient scheme for work-sharing constructs. Since the SPE is designed only for computation-intensive workloads, it should delegate I/O or main memory allocation to the PPE. This feature is provided as PPE-serviced function [14] that allows SPE program to seamlessly use PPE-side services. Although it is easy to use, it is not free of overheads such as DMA and activation (and context switch) of PPE threads to get the service done (i.e., the latency of PPE-serviced malloc function takes more than 100 times of that of DMA operation). For controlling a team of threads on SPEs, however, we need PPE-serviced malloc by which we can create thread control blocks on PPE side. Since frequent allocation and deallocation of memory can be detrimental to the runtime performance, we have come up with a mechanism to reuse previously allocated control blocks as much as possible [23]. The key idea of proposed scheme is a container, which is the unit of reuse in our scheme, consists of a chunk-sized array of work-sharing control blocks, a reusability counter, and a pointer to the next container instance. Containers are in a circular queue, and free containers are found by checking reusability counter of containers in the queue. This is effective especially when the application executes work-sharing constructs without implicit or explicit barrier synchronization [23].

Finally, we provide two selections of work-model: PPE together and SPE only model. In PPE-together mode, the task is executed by PPE and SPEs. In SPE-only mode, the task is executed by SPEs only, while PPE controls and synchronizes team threads. Programmer is able to use options “-mspe-only” or “-mppe-together” to inform compiler to choose the corresponding mode. The default mode is “-mppe-together”.

4. SPE COMPILER

4.1 SPE optimization

- Swing Modulo Scheduling

Swing modulo scheduling (SMS), which is implemented in GCC, is a software pipelining approach which exploits ILP out of loops by overlapping successive iterations of the loops and executing them in parallel. The key idea software pipelining is to find a pattern of operations (kernel code) so that when repeatedly iterating over this pattern, it produces the effect of initiating an iteration before the previous ones have completed [15]. The number of cycles between initiations of successive iterations (i.e., the number of cycles per stage) in software pipelined loop is termed as Initiation Interval (II). The goals of SMS are to find near minimal II, register pressure, and stage count.

Swing Modulo Scheduling takes dependency graph as its input, and after performs ordering of nodes (instructions) in the graph by given priority, and then it schedules all the nodes in the graph using ordering information computed. Scheduling is usually successful if II is big enough, but II should be minimized for better performance. However, when it schedule a node which has its predecessors and successors are already scheduled, since it does not perform backtracking, blocking situations often take places which makes scheduling fails no matter how big is II.

In Fig. 5, scheduling phase reads dependency graph as shown in the left and then schedules from V1 to V7 as shown in the right The schedule for each node is computed in such a way that a predecessor should be scheduled before its successor, but a node pointed to by a back-edge node (created by cross-iteration dependency) should be scheduled before that node by the amount of II. As shown in the right of Fig. 5, however, this scheduling algorithm fails when it try to schedule V7 because of conflicting conditions. Such dependency graphs and blocking patterns are found very often in SPU programs due to the 16 bytes memory access size in SPU – load/store of two different memory addresses (i.e., adjacent elements of an array) with the same 16-byte aligned address can be dependent on each other. These patterns can also be caused by cross-iteration loop dependencies of common type, for example, in loop nests. We modify described scheduling rule as follows: if both predecessors and successors of node v are scheduled, then search direction should be based on successors or predecessors depending on whether only forward successors or forward predecessors were scheduled. This allows preventing blocking situations as described above. After described modification, SMS gained 25% improvements in performance when it is applied to the example code in Fig. 4.

```
float a[100],b[100],c[100],d[100];
void foo (int n)
{
int i,m;
float x,y,z,t;
for( i = 0; i<n; i++){
x = a[i+1]; y = b[2*i + m];
z = c[3*i + m];
t = (x + y)/( x - y)*(x* y + z);
d[i] = c[i+1] * t;
d[i+1]= d[i]*( c[i+2]+d[i-1]);
}
}
```

Fig. 4. Compiler architecture.

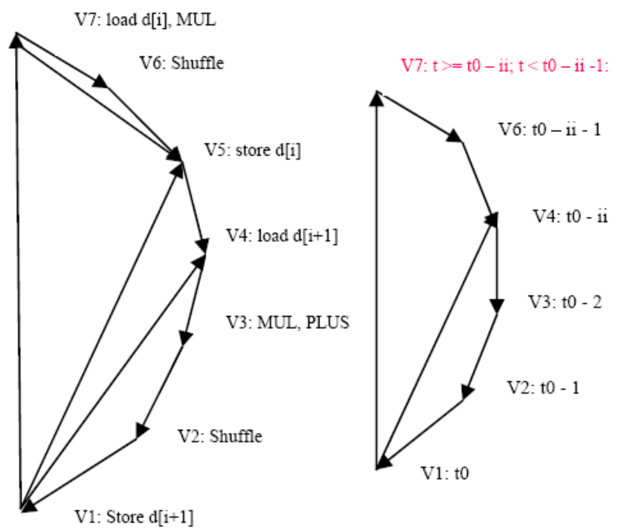

Fig. 5. Compiler architecture.

To get more performance, we integrated SMS with if-conversion so that conditional branches inside a loop removed and just one basic block (BB) remained in the loop. For this to happen, we've merged BBs in an innermost loop into a single BB without considering actual scheduling. Although we do not consider actual scheduling while merging, it generally shows good results since avoiding branch cost is more important in SPE and accurate instruction scheduling heuristics are still very local – does not consider cross-iteration ILP. After merging, CSE pass is run to remove redundancies comes from merging BBs, followed by SMS. Finally, we compared the schedule of instructions before and after this integrated SMS approach, and revert to the original loop body in case that is better. Although there is already a pass performing if-conversion before SMS in GCC, if-conversion itself cannot convert the body of a loop with internal control into a single BB, which makes SMS impossible. Even though SPE does not support predication except for `selb` instructions, this integration has shown good results. Since SPE does not have memory access exceptions, it is possible to speculate load and store instructions while if-conversion. Store speculation is possible as follows:

```
if(cond) *p = x   →
                      val = *p;
                      y = selb(x,val,cond);
                      *p = y
```

- Dependency in 16-byte memory alignment

SPE's memory instructions support only 16-byte aligned and 16-byte wide accesses. In case of accessing scalar variable, compilers must split a scalar load operation to a load-rotate instruction chain, and a scalar store to a load-shuffle-store chain. Because of these splits, two different memory addresses with the same 16-byte aligned address can be dependent on each other. The serial execution of the split instruction chain causes many stalls in SPE pipelines due to dependencies coming from the same operands of the instructions. Thus, we need precise dependency analysis confirming the independency between actually independent operands of different chains. In order to precisely break as many fake dependencies as possible for the SPE, we improved routines for memory dependency check.

First of all, we checked whether two memory references overlap by analyzing the dependency considering SPE's 16-byte access. Specially, in case of elements compactly laid in an array or a struct (or union), we exactly extracted the aligned base addresses, offsets from base, and access sizes. Then we checked whether two memory references can access same 16-byte aligned memory range. Additionally, in case that we can not find the aligned bases due to elements accessed via pointers, we compared the type of their contexts, and checked whether an element can be structurally isolated from the other by the amount of 16 bytes.

In case that we can find the distance between two memory references, basically we need to check whether the distance is greater than the access size of the preceding references, which means independency. Since the access size in SPE is 16 bytes, we modified the independency condition to check whether the distance is greater than 16 bytes.

The original GCC has used only the type-based alias analysis in order to break the dependencies between two memory references including pointers. For better analysis, we propagated the points-to sets including inter-procedural as well as intra-procedural analysis to RTL (low-level intermediate code in GCC) optimization phases, and then checked whether two points-to sets intersect each other.

- Other SPE specific optimizations

The SPE has deep pipelines, which make the penalty of incorrect branch prediction high, namely 18 cycles. Also, because branches are only detected late in the pipeline at a time where there are already multiple fallthru instructions, SPE simply assumes all branches to be not-taken. Thus, we first tried to eliminate taken branches by if-conversion. As shown in the previous sub-section, we aggressively applied if-conversion even inside loop optimization.

In cases taken branches cannot practically be eliminated, such as function calls or loop closing branches, we used a branch hint instruction (refered to as `hbr`) which is provided by SPE. This instruction specifies the location of the branch and its likely target address. When a hint is correct and scheduled at least 11 cycles before its branch, the branch latency is actually 1 cycle. When a hint is scheduled at least 4 instruction pairs, there is some delay before the branch is performed but still no branch penalties apply. Other cases, normal branch penalties occur.

To improve performance by allowing more dual issue and less instruction-fetch latency, we used bundler. We implemented bundler in such a simple way that if one instruction should start a new cycle or elements of two consecutive instructions are not on their proper pipelines, `nop` instruction is added to the instruction sequence to make dual issue. Also, if the number of instructions between branch hint and its branch is smaller than that for avoiding branch penalty, other instructions or `nops` are scheduled in between.

To reduce number of instruction fetches, we aligned function body and innermost loop nest in 64 byte alignment which is unit of instruction fetch in SPE. We basically reused what Sony has implemented in GCC4 for SPE – branch hint and bundling [13]. Then we improved branch hint to be added for every branch (for innermost loop, outer loops are sometimes ignored though) with no branch penalty, bundling to make more dual issue. If-conversion, branch hint, bundling and loop optimization is integrated for orchestration of each optimizations, and innermost loop body and function body is aligned for better instruction fetch.

- Profiling support for small LS size

GCC supports profile-directed optimization by automatically add instrumentation code in executable files. GCC adds instrumentation code to count edge probability and stored it in a counter table for a source file. The size of LS in SPE is only 256KB, however, so instrumented programs for profiling often does not fit into LS size. To address this problem, we tried to minimize the size of instrumentation code and counter table to be fit in LS. In our approach, only one source file is compiled with profile-generate option. If there are N source files, minimum N executables are generated. To minimize instrumentation code size, we limit the number of edges to be instrumented. If number of edges (E) in a source file is greater than the limit (MAX_INSTR_EDGES), M = ceiling (E/ MAX_INSTR_EDGES) executables are generated for the source file. After running M executables, complete profile data for the source file is made. The profile data files can be used when compile the source files again. When the instrumented functions are invoked in a SPE, it sends signal to the PPE with command. Then the PPE gets parameters via DMA, and then save the result of profile information. We also provide function level profile support to get profile information of specific functions.

4.2 Automatic SIMDization

Since the SPE is a SIMD unit, converting normal statements in a vector form is essential for improving performance. For automatically generating SIMD code, we used auto-vectorization support from GCC [10, 12]. It supports vectorization of such important statement as reduction, unaligned-access of arrays, and stridden access of arrays, so what we need to do is to define some target instructions and hooks for the SPE.

Because GCC's auto-vectorization is based on loops, however, it is useless for large basic-blocks with no loop which are frequent in codec. To address this problem, we are going to implement superword-level parallelism (SLP) which can vectorize consecutive instructions in a large BB [11].

4.3 Overlay and Partition Manager

An overlay or a partition is a set of function calls that will be loaded as a single entity during execution. In our scheme, partitions can be created by users (using special `#pragma` directives) or automatically by compiler [24]. A pointer to each partition and its size are saved in a list, called the *partition list*. The index in this list is used to load and to identify partitions for several purposes. This index is assigned to the partition during link time and embedded in any call which target is inside a partition. The static libraries (such as libc) and the program entry point always reside in local memory. For this reason, they are referred as *un-partitioned code*.

Under the code layout framework, a function can be called in two ways. If the function resides in local memory then the function call proceeds as usual. If the function is in a partition then the partition manager is called with certain extra information about the partition being called. The partition manager is a small ghost function, whose purpose is to load a code partition when needed and to create or to restore the state that is needed by such a partition. To maintain the state of old partitions in function call chains, a special structure needs to be maintained. This structure is called *partition stack* and it is analogous to a normal function stack. It keeps the caller partition id, return address and other useful information for each partition that has been called. The stack will be operated on at the beginning and at the end of every partition manager call.

When a partition is loaded to local memory, the place in which it resides is called the *partition buffer*. This buffer can be assigned many different partitions during runtime. Furthermore, it can be subdivided into segments by the programming environment. The replacement policies in this scheme are modulus and LRU [24].

5 EXPERIMENTAL RESULTS

5.1 Automatic Parallelization

We evaluated our OpenMP compiler with an alpha blending module (image processing module, 125 frames / HD quality) and mpeg2 codec (both are Samsung proprietary) running on a CBE processor (BladeCenter QS20 from IBM). Alpha blending is a typical application to test whether OpenMP is effective for image processing module or not in that the structure of alpha blending is almost same with other image processing modules (one loop that processes image data). Because the compiler is still under development, however, we do have some compilation problems with H.264 codec. In this test, we configured our OpenMP runtime so that only SPE threads are workers. Manual parallelization is done by one engineer who has experience in developing parallel streaming applications in the CBE.

TABLE 1
Comparison of manual and OpenMP parallelization with an alpha blending benchmark. 1 SPE is used.

	PPE only	Manual Parallelization	OpenMP (sw cache only)	OpenMP (sw cache and static buffer)
Parallelization time (approximation)		1 day	1 min	1 hour
Line of code (line)	84	227 (109 (SPE) 118 (PPE))	86	106
Execution time (sec)	1.45	1.83	3.88	1.48

Table 1 shows that OpenMP with software cache (for irregular data) and static buffer (for regular data) is faster than manual parallelization for alpha blending module. This is because the engineer who implemented manual parallelization does not care much about thread creation overheads and double buffering, which is automatically done by our OpenMP compiler. Since we gained average 1:1 performance ratio between PPE only and manual parallelization (1 SPE with double buffering) in our experience, we expect manual parallelization can be optimized more up to PPE only version level. Since automatic generation of static buffer code is not supported by the compiler at this moment, it takes a bit of manual implementation in the source code (it will be removed in near future). Low performance of software cache only is owing to memory related overheads: if software cache is used for array of image data, it manages each elements of the array which causes frequent DMA access for small data, along with many inefficient memory copy and cache lookup. From the result, it is clear that software cache should be the final choice - for irregular data accesses. If the compiler can identify data access patterns, we should use static buffer instead of software cache. Other overheads, such as thread synchronization, were negligible than expected.

TABLE 2
Scalability of OpenMP programs. Times in second.

	1 SPE	4 SPEs	Speedup
Alpha Blending	3.88	1.01	3.83
Mpeg2 (352x224, 60frames)	5.76	2.0	2.86
Mpeg2 (720x480, 60frames)	18.06	8.06	2.24

In this test, we observed that OpenMP is very useful tool in that it is very easy to use and the performance of OpenMP application is not very worse than that of manual parallelization. Image processing applications may achieve comparable performance by using it. Also, it can be used for fast prototyping (i.e., performance with either 1SPE or 4 SPEs) or fast development of parallel program structures followed by manual tuning.

But it still has problem in describing complicated details of parallelization. Current version of OpenMP can only deal with `for` loop with number of iterations known. Thus, we should have to transform `while` loop to `for` loop, and prepare buffers for each worker thread to work on independently (This can be solved in OpenMP 3.0). Also, OpenMP cannot be used in case where memory and CPU budget are very tight.

5.2 SPE Compiler

We evaluated the performance of our SPE compiler using well-known benchmarks and a H.264 decoder. The optimizations described in the preceding sections were all applied. Because all benchmarks are already (manually) simdized enough, we gained only small improvements from auto-vectorization. Instead, much of our improvements came from SPE specific optimizations (branch hint and bundling), swing modulo scheduling, if-conversion, and precise dependency analysis.

TABLE 3
SPEEDUPS OF OUR COMPILER OVER H.264 CODEC COMPARED WITH ORIGINAL GCC INTEGRATED WITH SONY'S OPTIMIZATION. BOTH ARE COMPILED WITH MAXIMUM OPTIMIZATION LEVEL.

Decoder Module	Optimized cycle	Original cycle	Speedup
CAVLD	10,979	14,726	1.34
Deblock	7,443	8,837	1.19
MC	4,900	5,870	1.20
IntraP	296	352	1.19
IQIT	1,015	1,258	1.24
Total	24,633	31,043	1.26

Overall, we achieved 26% increases in performance over H.264 codec, up to 107% over other general benchmarks. The main reason our optimization achieved better performance in codec than other benchmarks is that basically we focused our optimization for codec, thus used codec as our main benchmarks for optimizations. Generally we achieved better performance than XLC in most cases, as shown in table 5.

TABLE 4
SPEEDUPS OF OUR COMPILER OVER GENERAL BENCHMARKS. COMPARED WITH ORIGINAL GCC INTEGRATED WITH SONY'S OPTIMIZATION. BOTH ARE COMPILED WITH MAXIMUM OPTIMIZATION LEVEL.

Name	Optimized cycle	Original Cycle	Speedup
Matrix_mul	5,643,442	5,550,991	0.98
oscillator	2,904,141	3,152,589	1.09
vse_subdiv	4,742,792	5,113,267	1.08
Raytrace	28,080,847	28,512,762	1.02
fft_2d	2,568,529	2,623,213	1.02
LU	3,486,027	3,536,415	1.01
convolution	29,875,748	26,480,317	0.89
ip checksum	1,397,323	2,308,622	1.65
linpack single	2,064,373	2,438,692	1.18
linpack double	2,301,677	2,737,000	1.19
whetstone	927,994	1,040,624	1.12
dhrystone	37,601,696	81,701,761	2.17
IDEA	1,031,417	1,047,790	1.02
huff_enc	1,628,001	1,922,909	1.18
Gzip	820,233	953,740	1.16

One of our major concerns when applying optimizations such as loop optimization is the increase in code size. In case we added branch hints for every branch with no branch penalty (by adding 4 instruction pairs including nop) and performed aggressive bundling, we've got 10% increases in code size, which leads us to use this when performance is really important than code size.

TABLE 5
COMPARISON OF RESULTS OF OUR COMPILER WITH THAT OF XLC. BOTH CASES ARE OPTIMIZED WITH MAXIMUM OPTIMIZATION LEVEL.

Decoder Module	Optimized cycle	Original cycle	Speedup
CAVLD	10,979	15,947	1.45
Deblock	7,443	9,176	1.23
MC	4,900	6,062	1.24
IntraP	296	464	1.57
IQIT	1,015	1,268	1.25
Total	24,633	32,917	1.34

6. RELATED WORKS

IBM and Sony, which had participated in development of the CBE, introduced and have implemented compilers for it. IBM provides XL C/C++ Alpha Edition Compiler [5, 9] tuned for CBE. XL C/C++ implemented SPE specific optimizations such as bundling for dual issuing, compiler assisted branch prediction, and instruction fetch. It addressed data parallelization via automatic generation of SIMD codes. It also supports task-level parallelization via code and data partitioning across SPEs, though it is not available at the time of writing. Our project is similar to this, but our compiler is more optimized to media processing workloads. Also, XL for CBE has two paths (each for PPE and SPE) in a compiler, but our compiler has two separate compilers for PPE and SPE. In this aspect, our approach is similar with CUDA [17].

Sony provides tool chains including bin utilities, new libraries, and compilers for the CBE. Sony's compiler is based on GCC open-source compiler. Sony ported it to the SPE and added SPE specific optimizations. As its compiler adopted the versioned-up GCC, the ported compiler could naturally inherit the new compiler optimization techniques, then present good performance in general benchmarks. However, it does not support OpenMP yet, though they have plan [13]. We used some instruction scheduling optimization from Sony, but improved more as shown in section 3.3.

RapidMind provides the software development platform [7] for data parallel programming on the CBE, the GPUs, and multi-core CPUs. The RapidMind platform includes a dynamic compiler and runtime management system for parallel processing. In case of the CBE, the RapidMind platform allows C++ familiar programmers to easily make parallel programs distributing the computations across SPEs without any need to manipulate the low level details. But transforming C programs to RapidMind is very difficult – in our experience, RapidMind seems to be another language.

7. CONCLUSIONS AND FUTURE WORKS

In this paper, we have presented our compiler optimizations and approaches to easily exploit heterogeneous parallelism in the CBE architecture. For automatic parallelization of a single source file, we implemented OpenMP compiler for the CBE. As a result, we showed that for typical image processing application, the performance of our OpenMP compiler is comparable to that of manual parallelization. Also, it can be ported to other platform easily. For SPE, our compiler implements various optimizations. For ILP, we implemented SPE specific optimization, loop optimizations, and instruction scheduling with precise dependency analysis for the SPE. For data parallelization, we ported GCC autovect branch for the SPE and improved a bit. As a result, we achieved 26% improvements in H.264 codec, and up to 107% in other benchmarks.

Though this project is currently working, still there is room for improvement. For the SPE optimization, we are going to implement more optimizations for better performance in multimedia and graphics workload. We found many large BBs in codec, and if-conversion also makes large BBs. Thus, we are going to implement SLP to address large BBs by data parallelization. Furthermore, we are going to implement global scheduling support for better utilize ILP. For overlay, because first segment of code overlay causes page fault and performance penalty accordingly, we're going to implement pre-fetching of overlay to hide page fault overheads. This scheme depends on a special structure called the Partition Graph, which illustrates the relationships between partitions. Giving this graph, many parameters, like partition dependencies, usage and an optimal number of sub-buffers, can be calculated and used to pre-fetch partitions. Finally, improvements, such as dynamically switching between modulus and LRU policies, are planned.

Another direction of our future works is OpenMP and parallel programming model. OpenMP is very easy to use, but its programming model is very limited. While we were implementing OpenMP-mpeg2, one of inconveniences encountered was transforming WHILE loop to FOR loop (with number of iterations known). Care was needed on buffers for each thread to work on independently. Also, in many cases we found that some member variables of a class should be declared as `threadprivate` and others should be shared. But a class cannot have both of them - can have only one of them. Thus, we needed to refactor those classes by first analyzing them and factoring `threadprivate` member variables out of the classes. Both analyzing and factoring out (and modification of references for the factored out members) were tedious job. Finally, many streaming applications (including H.264 codec) needs pipelining of tasks which is not supported by OpenMP.

Our main target for optimization is image processing (i.e., H.264 and Mpeg2), thus we are going to use various codecs and image processing module for our benchmarks. We are going to use our compiler for HDTV/IPTV and portable media devices, especially for downloadable codec.

ACKNOWLEDGEMENT

Much of OpenMP implementation is the result of co-work with ETI international. Special thanks to professor Guang R. Gao and Ziang Hu from the University of Delaware.

REFERENCES

[1] Donald Tanguay, et al., “Nizza: A framework for developing real-time streaming multimedia application”, *http://www.hpl.hp.com/techreports/2004/*
[2] Karl Czajkowski, et al., “A resource management architecture for meta-computing systems”, *Lecture Notes In Computer Science*, *Vol. 1459*, 1997
[3] Seiji Maeda, et al., “A Cell software platform for digital media application”, *COOL ChipsVIII*, 2005
[4] *http://msdn.microsoft.com/*
[5] *http://www-128.ibm.com/developerworks/power/cell/*
[6] *http://www.cradle.com/*
[7] http://www.rapidmind.net/
[8] Steven S. Muchnick, ”Advanced Compiler Design and Implementation”, Morgan Kaufman Publishers, 1997.
[9] Alexandre E. Eichenberger, et. al, “Using advanced compiler technology to exploit the performance of the Cell Broadband Engine architecture”, IBM Systems Journal, VOL 45, NO 1, 2006.
[10] Dorits Nuzman, et. al., “Auto-vectorization of Interleaved Data for SIMD”, In Proceedings of the Programming Language Design and Implementation (PLDI ’06), Jun 2006.
[11] Jaewook Shin, et. al., “Exploting Superword-Level locality in Multimedia Extension Architecture”, Journal of Instruction-Level Parallelism 5 (2003) 1-28
[12] Dorit Nuzman, et. al., “Multi-platform Autovectorization”, In Proceedings of the International Symposium on Code Generation and optimization (CGO ’06), 2006,
[13] Ulrich Weigand, “Porting the GNU Tool Chain to the Cell Architecture”, In Proceedings of the GCC Summit 2005.
[14] “Cell Broadband Engine Programming Handbook”, http://www-128.ibm.com/developerworks/power/cell/docs_ documentation.html
[15] Josep Liosa, et.al., “Lifetime-Sensitive Modulo Scheduling in a Production Environment”, IEEE Transactions On Computer, VOL. 50, No. 3, March 2001.
[16] http://www.mc.com/cell/
[17] *http://developer.nvidia.com/object/cuda.html*
[18] T. Chen. et.al., “Optimizing the use of static buffers for DMA on a CELL chip”, In Proceedings of Workshop on Language and Compiler for Parallel Computing (LCPC), 2006.
[19] *http://gcc.gnu.org/projects/gomp/*
[20] *http://www.wikipedia.org*
[21] Alan H. Karp, et. al., “Data Merging for Shared-Memory Multiprocessors”, In Proceedings of the 26th Hawaii International Conference on System Sciences, Wailea, Hawaii, Volume I (Architecture), pages 244-256, January 1993.
[22] Junggyu Park, et. al., “Compiler Optimization for the Cell Architecture”, In Proceedings of the 5th Workshop on Optimization for DSP and Embedded Systems (ODES-5), in conjunction with CGO 5, March 2007.
[23] Jun Sung Park, et. al., “Implementation of OpenMP Work-sharing on the Cell Broadband Engine Architecture”, In Proceedings of International Workshop on OpenMP (IWOMP 2007), June 2007.
[24] Joseph B. Manzano, et. al., “Toward an Automatic Code Layout Methodology”, In Proceedings of International Workshop on OpenMP (IWOMP 2007), June 2007.