PROGRAMMING MODEL AND EXECUTION MODEL FOR OPENMP ON THE CYCLOPS-64 MANYCORE PROCESSOR

by

Ge Gan

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Electrical and Computer Engineering

Spring 2010

PROGRAMMING MODEL AND EXECUTION MODEL FOR OPENMP ON THE CYCLOPS-64 MANYCORE PROCESSOR

by

Ge Gan

Approved: _____

Kenneth Barner, Ph.D.
Chair of the Department of Electrical and Computer Engineering

Approved: _____

Michael J. Chajes, Ph.D.
Dean of the College of Engineering

Approved: _____

Debra Hess Norris, M.S.
Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Guang R. Gao, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Fouad Kiamilev, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Xiaoming Li, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Jingyi Yu, Ph.D.
Member of dissertation committee

ACKNOWLEDGMENTS

I owe my deepest gratitude to my adviser, Prof. Guang R. Gao. I started my PhD program in CAPSL from Aug. 2004. During the last six years, Prof. Gao spent a lot of time on my study. My research benefit a great deal from his profound learning, great wisdom, and invaluable experience. I thank Prof. Gao for his support, encouragement, and advisement which are essential for the progress I have made in the last six years. I would not have been able to complete this work without his help. His dedication to research and his remarkable professional achievements have always motivated me to do my best.

I thank Dr. Ziang Hu. Dr. Hu is an excellent software engineer. He shared his system software development experience with me and helped me a great deal in writing this thesis.

My sincere thanks also go to numerous colleagues and friends at the CAPSL Laboratory, including Joseph Manzano, Juergen Ributzka, E.J. Park, Fei Chen, Daniel Orozco, Haiping Hwu, Yingping Zhang, Juan Cuvillo, Sunil Shrestha, Tom St. John, Mark Pellegrini, Brian Lucas, Dimitrij Krepis, Handong Ye, Yuhei Hayashi, Andrew Russo, Xiaomi An, Joshua Suetterlein. They are very smart people. It has been a wonderful experience to work together with them.

Last and most, I thank my parents who gave me birth, raised me, and made me to achieve this goal in education. I thank them for their unending support and confidence. Without their support, I am not able to finish my study for PhD.

The work in this thesis was sponsored by these NSF grants: CNS-0509332, CSR-0720531, CCF-0833122, and CCF-0702244.

To Xu

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

During the last ten years, multicore processors have matured from academic research projects to real products in industry. They are now used in across almost the entire spectrum of computer systems, ranging from huge mainframes to small handheld devices. People consider that multicore processor represents the future of computer architecture design [1]. Currently, we may group multicore processor chip into two types: Type-1 and Type-2 [2]. Type-1 multicore processor has several traditional heavy weight processing cores *glued* on a single chip, like the Intel Core 2 Duo processor [3] and the AMD Quad-Core Opteron processor [4]. Basically, the design of Type-1 multicore processor is just a natural and conservative extension of the traditional single core processor architecture. Type-2 multicore processor, instead, represents people's effort to explore the parallel architecture design space and to search for the most suitable multicore processor design model. The IBM Cyclops-64 is a many-core processor falls in this category.

A Cyclops-64 chip has 160 homogeneous on-chip processing cores. They are connected by a 96-port, 7-stage, non-blocking on-chip crossbar switch. The Cyclops-64 chip does not have data cache. Instead, it has 5.2MB on-chip SRAM. In addition, 2GB off-chip DRAM can be connected to the on-chip crossbar switch via four DDR2 controllers. All these memories are located in the same address space and thus are shared by all on-chip cores. However, different memory segments have different access latencies and different bandwidth. It is the programmer's responsibility to orchestra data movement among different memory segments, especially between on-chip and off-chip memory.

As we all know, it is very difficult to program a chip with many processing cores, especially if the chip has user-managed memory hierarchies, like the IBM Cyclops-64 many-core processor. Without considering the heterogeneity of the memory space, Cyclops-64 is similar to the traditional shared memory SMP machine. For this kind parallel machine, OpenMP [5] is the dominate programming language. Although OpenMP provides abundant directives that programmers can use them to decompose loops in a sequential program and make it a parallel program, it provides little support to help programmers to deal with the segmented memory space. Therefore, significant problems will arise if an OpenMP programmer wants to develop OpenMP programs on the Cyclops-64 processor. For example, the problem of manually recoding the original program to add in data movement code; the problem of overlapping the execution of data movement code and computation code may arise. This motivates us to develop a series of *tile aware parallelization* techniques to attack these problems. The basic idea is to enhance the OpenMP API with the concept of data tile so programmers can use the extended OpenMP API to annotate their programs and tell a compiler what is the shape of the data tile and how it would be used in the program, or where the data tiles are located etc. The purpose is to expose more information about program data and their usage so a compiler can have more opportunities to perform some aggressive optimizations that would not be possible (or inefficient, or inaccurate) if without such hints from the programmers.

The major contributions of this thesis are:

- In this thesis, we introduce the concept of *tile aware parallelization*, an extension to the current OpenMP. We analyze and discuss some problems that OpenMP programmers would come across on the Cyclops-64 processor. Then, we use some motivating examples to demonstrate why tile aware parallelization techniques are necessary and also possible to solve these problems. As far as the authors are aware, we are the first that propose *tile aware parallelization* for the OpenMP programming language.

- The thesis proposes and develops *tile percolation*, an OpenMP tile aware parallelization technique that can be used to generate data percolation code for OpenMP programs running on the Cyclops-64 processor. The thesis provides an exploration of the necessity and possibility of developing pragma directives for semi-automatic data movement code generation in OpenMP. The thesis also introduces the techniques used to implement tile percolation, which includes the new programming API, code generation, and the required runtime support. Evaluation results show that tile percolation can make the OpenMP programs run on the Cyclops-64 chip much more efficiently.

- To improve the tile percolation technique, we have designed and developed the *Thread-Level Decoupled Access/Execution* (TL-DAE for short) model for OpenMP programs running on the Cyclops-64 chip. We have designed the TL-DAE programming interfaces that can be used to help OpenMP compiler to generate decoupled code. We have also developed the runtime support that is needed to support the TL-DAE execution model. The experimental results demonstrate the effectiveness of the TL-DAE execution model.

- We have proposed and developed an OpenMP tile aware parallelization technique called *tile reduction*. It can apply parallel reduction on multi-dimensional arrays. We discuss the methods used to implement tile reduction, including the required OpenMP API extension and the associated code generation technique. We evaluate the tile reduction technique with a set of benchmarks. The experimental results show that using tile reduction can make the code parallelization more natural and flexible. It not only can expose more parallelism in the program but also can improve its data locality.

Chapter 1

INTRODUCTION

Programmers have long enjoyed the programming and performance efficiency of the hardware-managed memory hierarchy (i.e. the cache-based memory hierarchy) that most general computer systems have been using. However, due to its scalability and power issue [6], the hardware-managed memory hierarchy is not very suitable to be adopted in multicore, especially manycore processor architecture. It is incredibly difficult to design a hardware-managed memory hierarchy for a 1000-core processor, which is anticipated by many computer scientists, e.g. David Patterson [7] and Fran Allen [8]. The recent cancellation of Intel's Larrabee project is partly due to the difficulty to design an efficient cache-based memory hierarchy for its 80 on-chip cores.

Because of these difficulties, people have to resort to another approach, i.e. the user-managed memory hierarchy approach. Typical examples of this kind are the IBM CELL processor [9, 10] and the IBM Cyclops-64 processor [11]. This kind of manycore processor usually has one or multiple pieces of user-managed on-chip memory [1] , which is much faster and bandwidth-wider than off-chip memory. In order for programmers to manage the on-chip memory in an easier and more effective way, it is necessary to provide a certain amount of help from the programming model side. However, the current mainstream programming languages (FORTRAN, C/C++, Java, etc.) were developed based on the assumption of an *uniform memory access* (UMA) model. Thus, under this

[1] Different segments of memory may be in the same address space, like the Cyclops-64 processor, or in different address space, like in the CELL processor

context, the designers of these programming languages never counted programming the user-managed memory hierarchy as a program design challenge and never tried to propose any language-level solution to it.

In this thesis, we will mainly use the Cyclops-64 processor and the OpenMP programming model as the basis to discuss the possible solutions. The approach we developed in this thesis is termed *tile aware parallelization*. Tiling is widely used by compilers and programmer to optimize scientific and engineering code for better performance. Many parallel programming languages support tile/tiling directly through first-class language constructs or library routines. However, the current OpenMP programming language is *tile oblivious*, although it is the *de facto* standard for writing parallel programs on shared memory systems. In this thesis, we introduce *tile aware parallelization* into OpenMP. Its purpose is to enhance the OpenMP API with the concept of tile/tiling, so more program information can be exposed to the OpenMP compiler. Therefore, more aggressive code transformation can be implemented and thus more parallelism (data & task) can be achieved by the OpenMP compiler.

Three *tile aware parallelization* (TAP for short) techniques will be developed in this thesis. The first TAP technique is called *tile percolation*. This technology is used to help OpenMP compiler to automatically generate data percolation code for OpenMP programs running on the Cyclops-64 manycore processor. The second TAP technique is the *thread-level decoupled access/execution* model. It is a continue improvement of the *tile percolation* technique. It is supposed to be used to generate decoupled code so percolation code and computation code can be executed in parallel on Cyclops-64. The last TAP technique we proposed is called *tile reduction*, which allows reduction to be performed on multi-dimensional arrays. The rest of this chapter will give a brief introduction to the Cyclops-64 manycore processor architecture, the OpenMP programming model, and the three TAP technologies we will discuss in this thesis.

1.1 Features of Manycore Processor Architecture - a Cyclops-64 Example

In this section, we will introduce the IBM Cyclops-64 manycore processor architecture. Figure 1.1 is a diagram of the Cyclops-64 chip and node. A Cyclops-64 chip has 80 homogeneous on-chip processors that are connected to a 96-port, 7-stage, non-blocking on-chip crossbar switch [12]. Each processor consists of two thread units, one floating point unit, and two SRAM memory banks (32KB each). A thread unit is a 64-bit, single issue, in-order RISC core operating at clock rate of 500MHz. Therefore, a Cyclops-64 chip contains 160 processing cores. Not like the traditional RISC processor, the thread running on a Cyclops-64 processing core is not preemptable. Instead, the thread seizes the processing core until it `exit`s or `return`s from the thread execution.

The chip has 512KB instruction cache. Every 10 processing cores (i.e. five processors) share a 32KB instruction cache bank. See Figure 1.1. The chip has no data cache. Instead, each core contains a small amount of on-chip SRAM. For the current generation, the amount of SRAM associated with each processing core is 32KB. Therefore, the whole chip has 5.2MB on-chip memory in total. The SRAM associated with each core can be configured into either Scratchpad Memory (SP), or Global Memory (GM), or both in combination [2]. The configuration is decided by the value stored in a system configuration register that can be specified by programmer. Moreover, the same configuration is applied across all processing cores. Therefore, all processing cores have the same amount of scratchpad memory. In the current configuration, half (16KB) of the SRAM is configured to scratchpad memory; another half (16KB) is configured to global memory. The current system software design (compiler, runtime, library) dedicates the whole scratchpad memory to thread stack storage.

In addition to on-chip SRAM memory, off-chip DRAM are attached onto the crossbar switch through 4 on-chip DRAM controllers. The amount of DRAM that can

[2] Scratchpad memory (SP) is a fast temporary storage that can be used to exploit locality under software control.

be attached to a Cyclops-64 chip is 2GB in. The Cyclops-64 chip does not support virtual memory. Therefore, all threads on the same Cyclops-64 chip are running in the same address space. It is the programmer's job to make sure that threads do not destroy each others text or data segments.

All memory modules are in the same address space and can be accessed directly by all processing cores [13]. However, different segment of the memory address space has different access latency and bandwidth. To a Cyclops-64 on-chip core, the fastest memory segment is its local scratchpad memory. For all other on-chip memory segments (i.e. all remote scratchpad memory and global memory), they have longer access latency than local scratchpad memory. However, they are much faster than off-chip DRAM. See Figure 1.2 for the detailed memory performance parameters of the Cyclops-64 memory hierarchies.

The A-switch interface of the chip connects the Cyclops-64 node to its six neighbors in the 3D-mesh network. In every CPU cycle, A-switch can transfer one double-word (8 bytes) in one direction. The 3D-mesh may scale up to several ten thousands of nodes, which becomes a powerful parallel computing engine that can provide computing power at Petaflops level.

Cyclops-64 is targeted at applications that are highly parallelizable and require enormous amount of computation power. The philosophies behind its architecture design are:

- Explore *thread level parallelism* [14, 15, 16, 17, 18] in the program instead of *instruction level parallelism* [19] in the program.
- Let user manage the memory hierarchies, not hardware.

These design philosophies greatly affects how programmers can program the Cyclops-64 processor efficiently and effectively. Roughly speaking, the Cyclops-64 chip

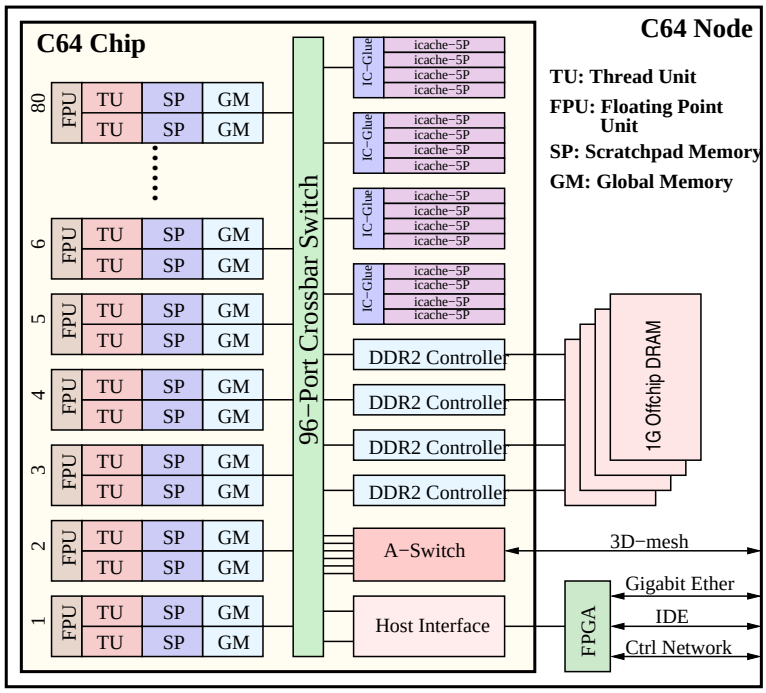


Figure 1.1: The IBM Cyclops-64 Manycore Processor

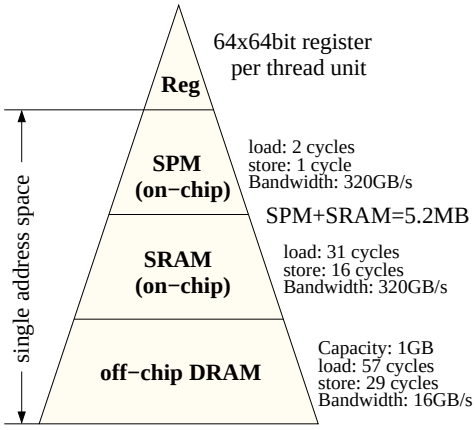


Figure 1.2: The Cyclops-64 Memory Hierarchy

is a single-chip shared memory multiprocessor system. Without considering the heterogeneity of its memory hierarchies, the most suitable programming model for Cyclops-64 is OpenMP [5].

1.2 A Brief Introduction to OpenMP

OpenMP is the *de facto* standard for writing parallel programs on shared memory multiprocessor systems. It is an application programming interface that supports multi-platform shared memory multiprocessing programming in C, C++, and FORTRAN on many architectures, including Unix and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that affect run-time execution behavior.

OpenMP is based upon the existence of multiple threads in the shared memory programming paradigm. It is an explicit parallel programming model, offering the programmer full control over parallelization. It is not an automatic parallel programming model. OpenMP uses the *fork-join* model of parallel execution. An OpenMP program begins as a single process - the *master* thread. The master thread executes sequentially until the first parallel region construct is encountered. The master thread then creates a team of parallel threads The statements in the program that are enclosed by the parallel region construct are then executed in parallel among the threads in the same team. When the team threads complete the statements in the parallel region, they synchronize and terminate, leaving only the master thread. See Figure 1.3

OpenMP is compiler directive based. Most OpenMP parallelism is specified through the use of compiler directives which are embedded in C/C++ or FORTRAN source code. OpenMP supports nested parallelism. This is achieved by placing a parallel construct inside another parallel construct. OpenMP API provides for dynamically altering the number of threads which may be used to execute different parallel regions. OpenMP specifies nothing about parallel I/O. It is entirely up to the programmer to insure that I/O conducted correctly within the context of a multithreaded program. OpenMP

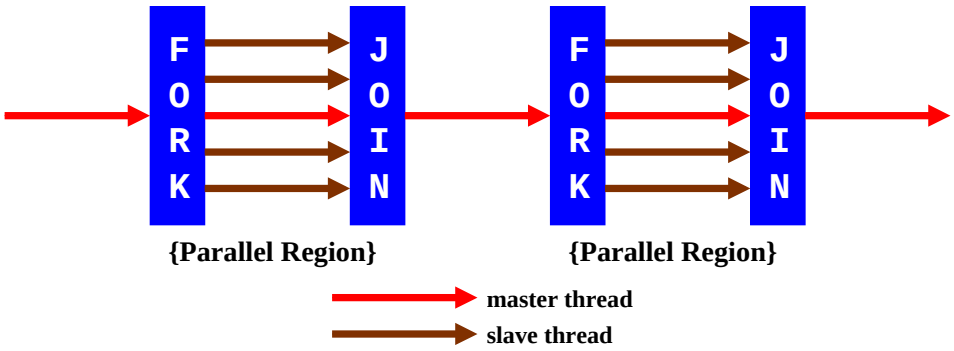


Figure 1.3: OpenMP Fork-Join Execution Model

provides a ”relaxed-consistency” and ”temporary” view of thread memory. Threads can cache their data in their local storage and are not required to maintain exact consistency with read memory all of the time. When it is critical that all threads view a shared variable identically, the programmer is responsible for insuring that the variable is flushed by all threads as needed.

Figure 1.4 shows the major five different categories of OpenMP language constructs and some of the examples. Following is a little bit detailed introduction.

Overview of OpenMP Language Extensions

parallel control structures	**work sharing**	**data environment**	**synchronization**	**runtime funcs env. variables**
governs flow of control in the program	distributes work among threads	scopes variables	coordinates thread execution	runtime environment
directive: **parallel**	directive: **for, single** **section, task ...**	clause: **shared, private** **firstprivate** **lastprivate ...**	directive: **critical** **atomic** **barrier ...**	**omp_set_num_threads()** **omp_get_thread_num()** **OMP_NUM_THREADS** **OMP_SCHEDULE ...**

Figure 1.4: Overview of OpenMP Language Extensions

- **Parallel Control Construct:** This governs the control flow of the program. The only construct that fall in this category is the `parallel` directive. It specifies a parallel region. Upon encountering a *parallel* directive, the master thread will fork a team of slave threads to execute the parallel region task in parallel. See Figure 1.3.

- **Work Sharing:** The work sharing constructs control how works in the parallel region are distributed among the threads. They include these directives: `for`, `single`, `section`, `task`. `for` defines how loop iterations are distributed among the threads; `single` requires that the guarded region can only be executed once; `section` decides how different bodies of the program code are executed by different threads; `task` specifies an explicit task that can be scheduled for execution by one of the threads in the team.

- **Data Environment:** These clauses specify the ”visibility” of the variables used in the OpenMP parallel program. By default, all variables are *shared* variable, which means that all threads access the same piece of data in memory if using the same symbolic variable name. Therefore, *synchronization* primitives are required to guarantee mutual exclusion if a shared variable is accessed by multiple threads and at least one is writing it. By predicating the variable with the `private` clause, the variable becomes private variable. Each thread would have its own local copy of the variable. Thus, no synchronization is required when accessing the variable.

- **Synchronization:** As was just mentioned, the OpenMP API need to provide synchronization primitives to coordinate the execution behavior among threads. The `critical` directive specifies a region of code that must be executed by only one thread at a time. The `atomic` directive specifies that a specific memory location must be updated atomically, rather than letting multiple threads attempt to write to

it. In essence, this directive provides a mini-`critical` section. The `barrier` directive synchronizes all threads in the team. When a `barrier` directive is reached, a thread will wait at that point until all other threads have reached that barrier. All threads then resume executing in parallel the code that follows the barrier.

- **Runtime Functions & Environment Variables:** OpenMP provides a set of runtime functions and environment variables so programmers can use them directly in their OpenMP program. Examples are like thread number or the number of threads in the team. Synchronization directives also have their corresponding runtime functions defined in the OpenMP runtime library.

The OpenMP designers have drawn a very clear line between programmer and compiler regarding the task of parallelizing a sequential program written in C/C++ and FORTRAN. Programmer's job is to use a set of simple and limited number of directives to annotates his/her source code to tell the compiler *where* and *how* [3] to parallelize the sequential program. It is the compiler's responsibility to perform the required code transformation to convert the sequential program into a multithreaded program on the host machine. This is usually achieved by wrapping certain segments of the control-flow of the sequential program into the native thread library functions, so they can be executed in parallel by different native threads at runtime. In this code transformation procedure, `synchronization` directives are mapped to native thread synchronization functions to ensure correct concurrent execution semantics.

1.3 Problems with OpenMP

Roughly, OpenMP APIs are control-flow oriented. OpenMP programmers' job is to use these APIs to identify & annotate potentially parallelizable *control-flow segments*

[3] It is very difficult for the compiler to know *where* and *how* to parallelize a sequential program. If we leave this for compiler, the technique would become *automatic parallelization* [20, 21, 22, 23].

in the code and expose them to OpenMP compiler, so OpenMP compiler can generate correct multithreaded program. Apparently, the OpenMP designers focus only on designing utilities for manipulating control-flow segments in the sequential program. The existing OpenMP APIs do not have constructs that can be used to transfer information about data (memory location, shape, size etc.) to compiler if the program is running on segmented memory address space. Just like what we have found on IBM Cyclops-64 processor. The reasons are:

- Originally, OpenMP was developed to parallelize sequential programs running on shared-memory parallel machines. Most of these machines have an uniform memory address space.

- The base languages on which OpenMP was grown are all control-flow oriented programming languages. They all assume an uniform memory address space in their memory model.

Therefore, the OpenMP designers did not have the motivation to invent directives that can be used to deal with problems that would occur when the parallel program is running on a machine with non-uniform memory address space. An example is given in Figure 1.5.

Figure 1.5 shows a piece of very typical OpenMP code. We assume this piece of code is part of a program that will run on Cyclops-64.

- **Line 1-5:** An array of record pointer is defined. Due to its size, this array and the record objects pointed to by the pointers stored in the array are placed in off-chip DRAM (which is larger) when the program starts running.

- **Line 6:** The `process_record` functions process a single record pointed to by `rp` without touching other global variables. Therefore, it is very suitable to be parallelized. Besides, we assume that data fields in the `record_t` would be accessed multiple times in this function.

```
 0    ...
 1 typedef struct {
 2    ...
 3 } record_t;
 4
 5 record_t *rcd[10000];
 6 void process_record(record_t *rp);
 7    ...
 8 #pragma omp parallel for shared(rcd) private(i)
 9 for (i=0; i<10000; ++i)
10 {
11     process_record(rcd[i]);
12 }
13    ...
```

Figure 1.5: A piece of code that can not be parallelized very well on machines with user-managed memory hierarchies (i.e. non-uniform memory address space)

- **Line 8-12:** A `for` loop traverses the pointer array to process each records. This loop is parallelized by using a typical OpenMP `parallel for` directive.

As we have mentioned, if we ignore the heterogeneity of its memory hierarchies, Cyclops-64 becomes an SMP parallel processing machine. Code between line 8 and 12 is a very standard OpenMP optimization for such a kind of SMP machine. However, due to the poor performance of off-chip memory (both latency and bandwidth, see Figure 1.2), it is desired to move each record into on-chip memory (which is much faster and has much wider bandwidth) and then call function `process_record` to work on the copies located in on-chip memory. In this way, the program execution on Cyclops-64 would have shorter memory access latency and would also save a great amount of off-chip memory bandwidth. But the existing OpenMP API does not provide any directive or clause that can help to achieve this goal. Programmers must add the required data movement code into the OpenMP program manually. Otherwise, the parallelized sequential would not achieve the expected high performance on this platform. Requiring programmers to deal with the heterogeneity among different memory segments would add another level of

complexity for programming a multicore processor.

In addition, even programmers (or compiler) have successfully inserted correct data movement code at the right place in the program, it does not ensure the good performance that the program is supposed to achieve. The problem is that the data movement code inserted is embedded in computation code and thus the two parts may not be able to run in parallel. Usually, people want to hide memory access latency by overlapping the execution of computation code and data movement code. This requires that the two parts, i.e. data movement code and computation code, can be dynamically decoupled [4] and executed in parallel. Sometimes, this can be solved by using the asynchronous DMA supported in hardware [26]. However, due to its hardware design complexity and runtime overhead [27], DMA may not be supported on every manycore processor, especially for those that do not use bus as their on-chip interconnection. As an example, Cyclops-64 is such a kind of processor that does not support asynchronous DMA in hardware. Therefore, it is the programmers job to recode the OpenMP program to make sure that data movements and computations are overlapped in execution on the Cyclops-64 processor. Unfortunately, the current OpenMP programming model and execution model do not provide any support for people to approach this goal.

In one word, the pragma directives and clauses existed in the current OpenMP is not enough to handle the issues that parallel programs would encounter on Cyclops-64. It is desired to extend the OpenMP APIs to solve these new problems. We would propose our solutions in the next section.

1.4 Solution Methodologies

From what have been discussed in the last section, we may draw the conclusion that significant challenges would pop up if programmers use the existing OpenMP APIs

[4] Like the instruction pipeline and out-of-order execution techniques used in modern Superscalar processors [24, 25] for exploring instruction level parallelism.

to parallelize their sequential programs for the Cyclops-64 manycore processor. The difficulties come from the fact that, as a programming model, OpenMP does not have language constructs that can be used to deal with user-managed memory hierarchies on the Cyclops-64 processor. To overcome these difficulties, it is required to extend OpenMP API to handle issues that would arise on manycore processors with user-managed memory hierarchies. This motivates me to develop the *tile aware parallelization* techniques for the OpenMP programming language. The basic idea is to enhance the OpenMP API with the concept of data tile so programmers can use the extended OpenMP API to annotate their programs and tell the compiler how the data tiles are look like and how they are used in the program, or where the data tiles are located etc. The purpose is to expose more information about program data so compiler can have more opportunities to perform some aggressive optimizations that would be impossible (or inefficient, or inaccurate) if without the hints input from programmer.

In the thesis, the first tile aware parallelization technique we developed for OpenMP programming language is called *tile percolation*. The purpose of tile percolation is to enable OpenMP programmers not only the capability to direct the compiler to perform computation decomposition, but also the power to direct the compiler to perform data movement related optimizations. Programmers will be provided with a set of simple OpenMP pragma directives. They can use these directives to annotate their program to instruct the compiler where and how data movement will be performed. Compiler will generate the correct computation and data movement code based on these annotations. At runtime, a set of routines will be provided to perform the dynamic data movement operations. This not only makes the programming on the Cyclops-64 chip easier, but also makes sure that the data movement code inserted into the program has good performance quality. Tile percolation is targeted to array intensive applications. So the major data objects being manipulated are sub-blocks in the multi-dimensional array. That's why this method is termed as tile percolation.

In the tile percolation technique, the data movement code generated by compiler is embedded in computation code. Under the current OpenMP execution model, data movement code can not be executed in parallel with the computation code. Asynchronous DMA can solve this problem. However, as we said, Cyclops-64 does not support DMA. To overcome this disadvantage, we propose the second tile aware parallelization technique for OpenMP. The technique is called *Thread-Level Decoupled Access/Execution*, or TL-DAE for short. Its purpose is to let OpenMP compiler generate the decoupled program so computation code and data movement code can be run in parallel. It is inspired by the original hardware based DAE [28, 29], in which memory access (operands fetch and results store) and computation execution are architecturally decoupled and thus can be maximally overlapped. Not like the hardware based DAE, TL-DAE is developed as a software execution model for OpenMP programs running on the Cyclops-64 processor. In our design, data movement code and computation code are decoupled implicitly by OpenMP compiler at compile time. At runtime, two different groups of threads are spawned: the computation threads and the percolation threads. Computation thread runs computation code while percolation thread runs data movement code. The execution of computation thread and percolation thread can slip with respect to each other, so percolation thread can run further ahead than computation thread and fetch data for it. Thus, computation code and data movement code of different stages can be executed in parallel. To help OpenMP compiler decouple the program, we propose the TL-DAE programming interface for the the programmers. The TL-DAE programming interface is a set of OpenMP tile aware parallelization pragma directives. Programmers can use these directives to annotate their programs to specify where and how data movement would be performed. OpenMP compiler, accordingly, will interpret these directives and generate the correct decoupled data movement code.

The third tile aware parallelization technique we developed for OpenMP is called *tile reduction*. Reduction is a very common recursive operation that performs aggregation

on a set of data of the same type. Due to the associativity and commutativity of the mathematical operator used in the calculation, reduction can be performed in parallel on better performance. The current OpenMP supports parallel reduction. However, it only supports reduction on scalar variables. We call this kind of reduction as *scalar reduction*. In this thesis, we introduce a new technique called tile reduction, which evolves the current reduction parallelization from scalar variables to multi-dimensional arrays. We have extended the traditional reduction clause to allow the programmers to annotate their code where tile reduction can be applied. We have also developed the required code generation technique to interpret the new reduction clause and generate the required parallel code accordingly.

1.5 Publications

This thesis is based on several published papers. The work on tile percolation [30] was included in the proceedings of Euro-Par 2009. The work on TL-DAE [31] was published in the 22nd International Workshop on Languages and Compilers for Parallel Computing (LCPC 2009). The work on tile reduction [32] was presented in the 5th International Workshop on OpenMP (IWOMP 2009).

1.6 Thesis Organization

The remainder of this thesis is organized as follows. Chapter **2** gives a background introduction of the multicore era and surveys the programming models developed for multicore processors with user-managed memory hierarchy. Chapter **3** introduces the tile percolation technique, which includes the design of the API and the implementation of code generation and runtime routines. Chapter **4** presents the thread-level decoupled access/execution model for OpenMP programs running on Cyclops-64. Chapter **5** introduces the tile reduction technique, which extends the OpenMP concurrent scalar reduction operation to multi-dimensional arrays. Chapter **6** concludes this thesis and also proposes several directions that we can make improvements.

Chapter 2

BACKGROUND

2.1 A Multicore/Manycore Era

With the silicon VLSI technology advancing towards 65nm, 45nm and 32nm [33], computer architecture design is leaping into the multicore (or manycore) era. There are a number of factors that drive this great change. They are: the power and thermal issue, the limits of instruction level parallelism, and the gap between CPU and memory speeds. Usually, people call them *power wall*, *ILP wall*, and *memory wall*.

- **Power Wall:** After four decades of development under Moore's Law, the number of transistors that can be integrated on a chip has reached 1-10 billion-level [34]. Meanwhile, the clock frequency has scaled to several gigahertz and is heading to ten gigahertz. Because of these changes, the power consumption of the chip has increased dramatically [35]. And the power dissipation has become a very high priority issue in all kind computer systems, from handheld devices to high performance computers [36, 37, 38]. Without special cooling system, the chip will burn out quickly under current clock frequency. Therefore, it is now not possible to improve processor performance by clock frequency scaling.

- **ILP Wall:** In 1991, in his famous paper [19], David W. Wall investigated the amount of instruction level parallelism that exists in typical programs. Through simulation, he analyzed a set of programs on which *"impossibly good techniques"* have been applied. These include register renaming, alias analysis, branch prediction, and speculative execution, etc. After analyzing the experimental results, he

found that *"the average parallelism rarely exceeds 7, with 5 more common"* [19]. This means that there is little performance potential left in instruction level parallelism regarding the current dynamic or static instruction scheduling techniques. Instead of increasing issue width, deepening the pipeline, and building complicated speculative execution units, it is time for people to explore performance at a higher level of parallelism, i.e. the *thread level parallelism* [14, 15, 16, 17, 18].

- **Memory Wall:** With the widening gap between CPU and memory speeds (CPU speeds double every eighteen months while memory speeds double only every ten years) [39, 40], the CPU performance suffers a lot from longer memory access latency. Currently, in a typical modern processor, loading from off-chip memory will cost several hundred or a thousand cycles [41]. And this number is still increasing. Apparently, it is not possible to hide such a long latency in the traditional single thread execution model.

Power wall, ILP wall, and memory wall together form the brick wall that stop people from improving CPU performance through old methods. They make some of the old wisdom used in designing the traditional superscalar architecture no longer applicable [7]. For example, clock frequency scaling is limited by power issue; the potential of advanced branch prediction and dynamic scheduling hardware units is constrained by ILP limits; and the speculative execution units can not tolerate such a long memory access latency in a single thread context, no matter how clever and aggressive they are. Computer architects need new wisdom to design computer processors of the next generation. The new wisdom must be able to use the billion-level transistor budge in a more power efficient way and must provide enough headroom for performance improvement.

Under these settings, multicore processor (or chip multiprocessor (CMP)) are emerging as a very promising alternative to the conventional superscalar architecture. Multicore processor usually consists of several, or a great number of processing cores on a single chip. The processing cores are arranged in a decentralized microarchitecture

which simplifies wiring and interconnection logic [42], and thus achieves higher scalability and power efficiency [43]. Meanwhile, it provides direct and efficient support in hardware for the compiler and runtime system to exploit thread level parallelism in the program, in which it was believed to have much more potential for performance improvement [44]. Furthermore, the multithreaded execution model provides an easy way to hide the ever increasing memory access latency [45] and thus improves the program execution efficiency. In short, multicore processor is a very promising computer architecture that can effectively translate the billion number of transistors on the chip to program performance without violating power consumption limits.

Because of the advantages and promising future of multicore processor, all major chip makers have announced their multicore plans or have even started shipping multicore products to the market. For example, the IBM CELL processor [9, 10], IBM POWER5 [46], Sun Niagara [47], Intel Dual-core Montecito [48], Intel Tera-scale 80-core processor [49], ClearSpeed [50], nVidia [51], ATI [52], Tilera [53], and AMD Dual-core Opteron. In addition to the works in industry, people in academia have also initiated many multicore research projects. For example, the Hydra project [54] and the Smart Memory project [55] in Standford, VIRAM [56] in Berkeley, the Cyclops-64 project [11] between Univ. of Delaware and IBM, the RAW [57] and SCALE [58] projects in MIT, the TRIPS project [59] in UT Austin, and the Synchroscalar project [60] in UC Davis. In these research projects, a number of multicore architecture designs have been explored. They come across many major subjects of computer architecture design, which include the high speed on-chip interconnect techniques, the efficient synchronization mechanisms, the scalable and low-latency cache organization, the heterogeneous core vs. homogeneous core choices, and different parallel programming models, etc.

Roughly, all these multicore processors, either from industry or from academia, fall into two categories. For convenience's sake, we call them type-1 and type-2 multicore processors [2].

- **Type-1 Multicore Processor:** The major characteristic of the type-1 multicore processor is that it always glues several heavy weight processing cores on a single chip. Usually, these processing cores are the traditional Superscalar processors. Most of the current multicore processor products in the market are type-1, e.g. Intel Dual Core Montecito and AMD Dual Core Opteron.

- **Type-2 Multicore Processor:** All multicore processors that are not type-1 fall into this category. Type-2 multicore processors represent people's effort to explore the parallel architecture design space and to search for the most suitable multicore processor design model [61, 62]. Usually, a variety of design choices can be observed in different type-2 multicore processors. For example, heterogeneous core vs. homogeneous core, etc. The typical type-2 multicore processors are the IBM CELL processor and the Cyclops-64 processor.

Given fixed die area and power budget, type-1 multicore processor does not scale very well. The reason is that the on-chip processing cores used on type-1 multicore chip are still the traditional Superscalar processors. They rely heavily on complicated out-of-order execution logic and thus consume a significant amount of on-chip silicon resources (which is disproportionally higher than its performance gain). Meanwhile, such a complicated circuit design makes it very power inefficient. For these reasons, the number of on-chip cores integrated on type-1 multicore processors rarely exceeds 16.

An alternative is to integrate many simple processing cores on the multicore chip. These processing cores usually adopt in-order RISC execution engine with instruction issue width of one or two. Each processing core exploits a moderate amount of parallelism within a single thread, but the whole chip can leverage the massive thread level parallelism in the application. Because each processing core is very simple, it consumes much less transistors and is very power efficient. Therefore, this kind of multicore processors have

the potential to scale up to several hundred or thousand cores [1]. They are also given the name *manycore* processor. The IBM Cyclops-64 processor is this kind of type-2 manycore processor. Other multicore/manycore processors that also fall into the type-2 group include the IBM CELL processor [9, 10], ClearSpeed [50], nVidia [51], Tilera Tile64 [53], and Intel Larrabee [63], etc.

These type-2 manycore processors can be further divided into different subgroups based on other design features, like the homogeneity of the on-chip cores (homogeneous vs. heterogeneous); the type of on-chip interconnections (mesh, crossbar, or NoC); and the flavor of the processor's memory hierarchy (hardware-managed or user-managed). The interest of this thesis is not on the architecture design of manycore processors. In stead, our discussion will be focused on the problem of how to program the user-managed memory hierarchy that is widely used in many type-2 manycore processor. In the next section, we will give a brief survey of the programming models proposed for user-managed memory hierarchy.

2.2 Programming Models for User-Managed Memory Hierarchy

The most well known programming model for computer systems with user-managed memory hierarchy is OpenCL [64], i.e. the **O**pen **C**omputing **L**anguage, which is is managed by the non-profit technology consortium Khronos Group. OpenCL is not designed for any specific computer platform. It is a language framework for writing programs that run across heterogeneous platforms consisting of CPUs, GPUs, and other processors. These CPUs, GPUs, and other processors may each contain local/private memory that has different access latency & bandwidth and can be manipulated directly by programmers. OpenCL includes a language (which is based on C99 standard) for writing kernel functions (code that would execute on devices). In addition, OpenCL also

[1] For type-2 multicore processor, the Moore's Law now becomes: *"the number of processing cores on a single chip doubles every generation."*

include a set of APIs that are used to define and then control the whole platforms. OpenCL provides parallel computing using task-based and data-based parallelism.

OpenCL gives any general application the right to access the Graphical Processing Unit for non-graphical computing purpose. The GPU had previously been available for graphical applications only. The GPU memory would be available to the operating system and or applications essentially as faster system memory than the main system memory. Therefore, OpenCL extends the power of the Graphical Processing Unit beyond graphics, i.e. the general-purpose computing on graphics processing units. In other words, in OpenCL, the low level memory details (main memory, device memory, etc.) are exposed to users via standardized programming interfaces. It is the programmers' responsibility to use these APIs efficiently and effectively. Other programming models that also adopt similar policy are nVidia CUDA [51] and Microsoft DirectCompute [65].

CUDA is short for **C**ompute **U**nified **D**evice **A**rchitecture. It is a parallel computing architecture developed by nVidia. CUDA is the computing engine in nVidia graphics processing units or GPUs that is accessible to software developers through industry standard programming languages. CUDA assumes a memory model in which CPU and GPU have separate memory spaces. Data is moved across PCI bus. It is necessary to use special functions to allocate, copy, and deallocate memory on GPU. In addition, pointers in CUDA are just addresses. Programmers can not tell from the pointer value whether the address is on CPU or GPU. Dereferencing CPU pointer on GPU will crash the program. So it is necessary to exercise care before dereferencing. It is the same for vice versa. Microsoft DirectCompute is an application programming interface (API) that supports General-purpose computing on graphics processing units on Microsoft Windows Vista or Windows 7. DirectCompute is part of the Microsoft DirectX collection of APIs. Similar memory model and memory access APIs can also be found in DirectCompute.

UPC [66] (Unified Parallel C) is another parallel programming language that deals with user-managed memory hierarchy at language level. It is an extension of the C

programming language designed for high-performance computing on large-scale parallel machines, including those with a common global address space (SMP and NUMA) and those with distributed memory (e.g. clusters). The programmer is presented with a single shared, partitioned address space, where variables may be directly read and written by any processor, but each variable is physically associated with a single processor. UPC uses a Single Program Multiple Data (SPMD) model of computation in which the amount of parallelism is fixed at program start-up time, typically with a single thread of execution per processor. In UPC memory model, there are two kinds of memory: shared memory and private memory. These two kinds of memory are logically correspond to the main memory and device memory in OpenCL and CUDA. Static and dynamic memory allocation are supported for both shared and private memory. Variables can be declared to be shared using the keyword `shared`. Shared objects are placed in memory based on affinity. Affinity can also be defined based on the ability of a thread to refer to an object by a private pointer. All non-array shared qualified objects have affinity to thread 0. All array shared qualified objects are distributed across the private memory (associated with each thread) via the predefined distribution policy. Threads can access shared and private data. UPC provides standard library functions to move data to/from shared memory. They can be used to move chunks in the shared space or between shared and private spaces. This part is exactly the same as we can see in OpenCL and CUDA.

The IBM CELL Broadband Engine (BE) processor [67] is a heterogeneous many-core processor that provides both flexibility and high performance. The first generation CELL BE processor includes a 64-bit multithreaded PowerPC processor element (PPE) and eight synergistic processor element (SPE). Each SPE has 256-KB local memory (also called local storage) to accommodate both program instructions and data. These local memories and main memory are each in a separate address space. SPEs transfer data between local memory and main memory through the DMA engine, which is controlled by programmers though DMA instructions. It is very clear that it is a great challenge to

the programmers to manage the transfers of code and data between main memory and local memory. To attack this problem, a *single shared memory abstraction* is assumed in the programming model proposed by IBM [68]. This is achieved by using the compiler-controlled software cache in SPE. Like its hardware counterpart, a software directory is maintained in each SPE local storage. The compiler, after inter-procedural analysis, replaces some load and store instructions with instructions that explicitly look up the effective address of the datum in the software cache directory. If the line containing the datum is found in the directory, the address of the requested variable is computed and the load or store continues using this local store location. Otherwise, a subroutine, the cache miss handler, is invoked and the requested data is transferred from system memory. Just like the hardware caches, the directory is updated, and typically another line is selected for eviction in order to make room for the new element. Therefore, by using this approach, the compiler can undertake the task of orchestrating data transfer between local memory and main memory. Moreover, there are many optimizations that the compiler can perform to optimize these data transfers, especially when memory references are regular.

The current implementation supports two variants of software cache. One variant supports only single-threaded code. In this version, as its name suggests, the cache line can be written out to main memory as a whole without worrying about overwriting data that are not supposed to be modified. Therefore, there is not need to keep track which words have been modified. This greatly simplifies the code sequence of cache miss handler and also greatly reduces the runtime overhead of cache line eviction. However, this requires that a multithreaded program must not have shared variable, which makes it not practical for most real parallel applications. The second variant supports multi-threaded shared memory programs. In order to do this, it is necessary to keep track in the cache directory (on each SPE) that which bytes have been written since the line was brought in from system memory. Then, when a line is to be evicted, either in support of a miss or to implement explicit flushes required for conformance with memory consistency rules,

only the modified bytes are written by the DMA engine. Keeping track of dirty bytes requires inserting additional code inline for each store operation in addition to the lookup code discussed above. To accomplish this, the directory entry is extended by additional bits for each bytes in the cache line. This is an non-trivial space overhead in addition to the runtime overhead. If the data access pattern is regular, the program can use static data buffer and thus eliminating all the storage and runtime overhead.

Another programming model developed for the CELL BE architecture is termed CellSs, i.e. the Cell Superscalar framework (CellSs), which is based on a source to source compiler and a runtime library. The supported programming model allows the programmers to write sequential applications and the framework is able to exploit the existing concurrency and to use the different components of the Cell BE (PPE and SPEs) by means of a automatic parallelization at execution time. The only requirement is that annotations (somehow similar to the OpenMP ones) are written before the declaration of some of the functions used in the application. The annotation, on one hand, indicates a parallel task region, on the other hand, specifies the data that need to be copied into or copied out of the SPE local storage at runtime. The compiler generate the parallel program with data movement code inside.

HTA [**?**], i.e. the **H**ierarchically **T**iled **A**rrays, introduces the concept of ”data tile” as the first class data object into the programming language. Such a ”tile oriented” programming model can be used to program the array intensive algorithms. In these algorithms, multi-dimensional arrays are the core data structures. HTA partitions these multi-dimensional arrays into multiple hierarchies of data tiles. Programmers can manipulate these data tiles directly in their programs. The advantages are:

1. The tile hierarchy directly models the memory hierarchy of the computer system. Therefore, HTA facilitates the exploration of different levels of data locality existed in the program.

2. The hierarchical data tiles can be distributed in physically separated memory segments, so communications in the parallel program can be modeled as distributed tile assignment, which is much simpler and less error prone than explicit thread communication routines, e.g. MPI.

3. Data parallelism of the program can be harnessed via the data tile operations, therefore, parallelism is highly structured, which greatly improves the readability over the SPMD paradigm.

All the above methods either rely solely on programmers to manage data movement explicitly in the program or need programmers' hints to direct compiler to generate the required code. Kandemir propose a pure compiler method in [69] to deal with the data movement problem. His work targets single core embedded processors with user-managed scratch-pad memory. His approach is basically a natural extension and application of Monica Lam and Michael Wolfe's loop nest optimization work to embedded processors. The method includes an optimization suite that uses loop and data transformations, an on-chip memory partitioning step, and a code-rewriting phase that collectively transform an input code automatically to take advantage of the on-chip SPM. Compared with previous work, the proposed scheme is dynamic, and allows the contents of the SPM to change during the course of execution, depending on the changes in the data access pattern.

The method proposed in this thesis is not a pure compiler approach. It relies on programmer's intervention to direct compiler to generate the required optimized code. It not only simplifies the development of compiler but also grant more flexibility for programmer to develop port parallel program.

Chapter 3

TILE PERCOLATION

Programming a multicore processor is difficult. It is even more difficult if the processor has user-managed memory hierarchy, e.g. the IBM Cyclops-64 (C64). A widely accepted parallel programming solution for multicore processor is OpenMP. Currently, all OpenMP directives are only used to decompose computation code (such as loop iterations, tasks, code sections, etc.). None of them can be used to control data movement, which is crucial for the C64 performance. In this chapter, we propose a technique called *tile percolation*. This method provides the programmer with a set of OpenMP pragma directives. The programmer can use these directives to annotate their program to specify *where* and *how* to perform data movement. The compiler will then generate the required code accordingly. Our method is a semi-automatic code generation approach intended to simplify a programmer's work. In this chapter, we provides **(a)** an exploration of the possibility of developing pragma directives for semi-automatic data movement code generation in OpenMP; **(b)** an introduction of techniques used to implement tile percolation including the programming API, the code generation in compiler, and the required runtime support routines; **(c)** and an evaluation of tile percolation with a set of benchmarks.

3.1 Introduction

OpenMP [5] is the *de facto* standard for writing parallel programs on shared memory multiprocessor system. For the IBM Cyclops-64 (C64) processor [70, 71, 11], OpenMP is one of the top selected programming model. As shown in Figure 1.1, the C64 chip has 160 homogeneous processing cores. It has instruction cache but no data cache.

Instead, each core contains a small amount of SRAM, 512KB each. So there are total 5.2MB on-chip SRAM. Part of them can be configured into Scratchpad Memory (SPM). The rest are called Global Memory (GM). Off-chip DRAM are attached onto the crossbar switch through 4 on-chip DDR2 memory controllers. All memory modules are in the same address space and can be accessed directly by all processing cores [13]. Therefore, data can be moved from any segment of the address space to any other segment of the address space through the normal `load`/`store` instructions. However, different segment of the memory address space has different access latency and bandwidth. See Figure 1.2 for the detailed parameters of the C64 memory hierarchy. Roughly speaking, the C64 chip is a single-chip shared memory multiprocessor system. Therefore, it is easy to land OpenMP on the C64 chip [72]. However, due to its *user-managed* memory hierarchy, making an OpenMP program run efficiently on the C64 chip is not a trivial task.

Given a processor like C64, it is important for the OpenMP programs to fully utilize the on-chip memory resources. This requires the programmer to insert code in the program to move data back and forth between the on-chip and off-chip memory. Thus, the program can benefit from the short latencies of the on-chip memory and the huge on-chip bandwidth. Unfortunately, this would make the C64 multicore processor more difficult to program. From the OpenMP methodology, we have learned that it would be very helpful if we could annotate the program with a set of OpenMP pragma directives to specify where data movement is beneficial and possible, and let the compiler generate the required code accordingly. This is just like using the `parallel for` directive to annotate a loop and let the OpenMP compiler generate the multithreaded code. This would free the programmer from writing tedious data movement code.

To solve this problem, we developed *tile percolation*, a tile aware parallelization technique [32] for the OpenMP programming model. The programmer will be provided with a set of simple OpenMP pragma directives. They can use these directives to annotate their program to instruct the compiler *where* and *how* data movement will be performed.

The compiler will generate the correct computation and data movement code based on these annotations. At runtime, a set of routines will be provided to perform the dynamic data movement operations. This not only makes the programming on the C64 chip easier, but also makes sure that the data movement code inserted into the program is optimized. Since the major data objects being moved are ”sub-blocks” in the multi-dimensional array, this approach is termed *tile percolation*. The major contributions of this research are as follows: (a) As far as the author is aware, this is the first research that explores the possibility of using pragma for semi-automatic data movement code generation in OpenMP; (b) The research has developed the techniques used to implement tile percolation, including the programming API, the code generation in compiler, and the required runtime support. (c) We have evaluated tile percolation with a set of benchmarks. Our experimental results show that tile percolation can make the OpenMP programs run on the Cyclops-64 chip more efficiently.

The rest of the chapter is organized as follows. In Section 3.2 we use a motivating example to show why tile percolation is necessary. Section 3.3 will discuss how to implement tile percolation in the OpenMP compiler. We present our experimental results in 3.4 and draw our conclusions in Section 3.5.

3.2 A Motivating Example

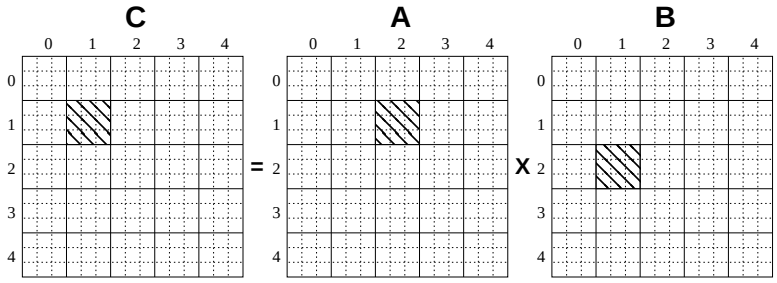
In this section, we use the tiled matrix multiplication code as a motivating example to demonstrate why writing efficient OpenMP program for the user-managed C64 memory hierarchy is not trivial and why a semi-automatic code generation approach is necessary.

Figure 3.1 shows the tiled matrix multiplication code [1] and the data access pattern

[1] Because of the advantages of the *surface-to-volume* effect [73], the *algorithm-by-tile* approach [74, 75, 76] is used intensively in developing scientific and engineering code. For instance, the LAPACT programs [77] use many level-3 BLAS code [78] to leverage the computer’s memory hierarchy, no matter if the memory hierarchy is managed by hardware or software, or if it is managed implicitly or explicitly.

```
1 for (ii=0; ii<n; ii+=b)          controlling loops
2  for (jj=0; jj<n; jj+=b)
3   for (kk=0; kk<n; kk+=b)
4    for (i=ii; i<min(ii+b,n); i++)   tiling loops
5     for (j=jj; j<min(jj+b,n); j++)
6      for (k=kk; k<min(kk+b,n); k++)
7       C[i][j]+=A[i][k]*B[k][j]
```

(a) Tiled Matrix Multiplication Code



(b) Data Access Pattern in the Tiling Loops

Figure 3.1: Tiled Matrix Multiplication: C = A x B

of the kernel loops. On the C64 chip, to make sure that this program fully utilizes the on-chip memory resources, the programmers need to insert tile movement code manually in the source code to move data tiles back and forth between the on-chip and off-chip memory. Figure 3.3 shows the examples of the manually inserted code. In both examples, no matter how the computations in the *controlling loops* are decomposed among the cores, for the tiling loops, small data tiles are moved into the on-chip SRAM memory and the associated computations are performed there. Figure 3.3(a) shows the naive version, in which the array elements are copied into the on-chip memory one by one. A better version is shown in Figure 3.3(b), which utilizes the off-chip memory bandwidth more efficiently. In both versions, the programmers need to study the original source code carefully to figure out how to write correct and efficient data movement code. They are forced to deal with the convoluted array index calculation, which makes their work more complicated.

A simpler approach is to let the compiler to generate the required data movement

```
 0 /* allocate on-chip memory */
 1 AA=(float *)sram_malloc(...);
 2 BB=(float *)sram_malloc(...);
 3 CC=(float *)sram_malloc(...);
 4       ...
 5 /* item-by-item memory copy */
 6 for (i=ii;i<min(ii+b,n);i++)
 7     for (j=jj;j<min(jj+b,n);j++)
 8         for (k=kk;k<min(kk+b,n);k++){
 9             AA[i-ii][k-kk] = A[i][k];
10             BB[k-kk][j-jj] = B[k][j];
11             CC[i-ii][j-jj] = C[i][j];
12         }
13
14 /* MxM performed on-chip */
15 for (i=0;i<min(b,n-ii);i++)
16     for (j=0;j<min(b,n-jj);j++)
17         for (k=0;k<min(b,n-kk);k++)
18             CC[i][j]+=AA[i][k]*BB[k][j];
19
20 /* copy out the results */
21 for (i=ii;i<min(ii+b,n);i++)
22     for (j=jj;j<min(jj+b,n);j++)
23         for (k=kk;k<min(kk+b,n);k++)
24             C[i][j]=CC[i-ii][j-jj];
25          ...
```

Figure 3.2: Examples of Manually Inserted Data Movement Code (Pseudo Code): A naive version

code automatically. In [79, 80], the authors present their implementation of this idea on the IBM CELL processor. However, in [81] it is revealed that the performance of the automatically generated code is not as good as the performance of the manually reformed code [2]. The reason is not because the compiler can not generate the optimal code, but because the static analysis performed by the compiler is not powerful enough to capture all the beneficial cases (which is a well-told story). This motivates us to develop a novel semi-automatic approach: the programmer specifies the places in their program where efficient data movement is needed, while the compiler generates the required high quality

[2] Readers are referred to Figure 12 in [81]

```
 0 /* allocate on-chip memory */
 1 AA=(float *)sram_malloc(...);
 2 BB=(float *)sram_malloc(...);
 3 CC=(float *)sram_malloc(...);
 4       ...
 5 /* mcpy: optimized memory copy routine */
 6 for (i=ii; i<min(ii+b,n); i++)
 7     mcpy(&CC[i-ii][0], &C[i][jj], min(b,n-jj))
 8 for (k=kk; k<min(kk+b,n); k++)
 9     mcpy(&BB[k-kk][0], &C[k][jj], min(b,n-jj))
10 for (i=ii; i<min(ii+b,n); i++)
11     mcpy(&AA[i-ii][0], &A[i][kk], min(b,n-kk))
12             ...
13 /* on-chip calculation */
14 for (i=0; i<min(b,n-ii); i++)
15     for (j=0; j<min(b,n-jj); j++)
16         for (k=0; k<min(b,n-kk); k++)
17             CC[i][j]+=AA[i][k]*BB[k][j];
18               ...
19 /* copy out the results */
20 for (i=ii; i<min(ii+b,n); i++)
21     mcpy(&C[i][jj], &CC[i-ii][0], min(b,n-jj))
22               ...
```

Figure 3.3: Examples of Manually Inserted Data Movement Code (Pseudo Code): An optimized version

code accordingly.

3.3 Tile Percolation

In this section, we will use a simple example to demonstrate how to implement tile percolation. It includes three parts: the programming API, the data movement code generation in the compiler, and the required runtime support.

3.3.1 Programming API

In the design of the programming API for tile percolation, the following criteria should be considered. First, it must be very simple and easy to use. Second, it must be general enough to capture most of the common cases that can benefit from tile percolation.

```
#pragma omp percolate [tile ...]
#pragma omp tile ro (A[j_dim(A), a_dim(A), L_dim(A)]..[j_2, a_2, L_2][j_1, a_1, L_1], ...)
                 wo (B[k_dim(B), b_dim(B), M_dim(B)]..[k_2, b_2, M_2][k_1, b_1, M_1], ...)
                 rw (C[l_dim(C), c_dim(C), N_dim(C)]..[l_2, c_2, N_2][l_1, c_1, N_1], ...)
```

(a) The definition of the tile percolation API

`percolate:`	Directive name. It specifies a percolation region
`tile:`	Directive name. It specifies a tile region and the tile descriptors
`ro:`	Clause name. It specifies the tiles that are read-only in the current percolation region.
`wo:`	Clause name. It specifies the tiles that are write-only in the current percolation region.
`rw:`	Clause name. It specifies the tiles that are read and written in the current percolation region.
`A,B,C:`	Name of the host multi-dimensional data array
j_i,k_i,l_i:	The index variable of the for loop that defines the i_{th} dimension of the tile
a_i,b_i,c_i:	Blocking size of the i_{th} dimension of the host multi-dimensional array (i.e. A, B, and C).
L_i,M_i,N_i:	Size of the i_{th} dimension of the host multi-dimensional array
$\dim(..)$:	The dimension of the multi-dimensional array

(b) The explanation of the tile percolation API

Figure 3.4: The OpenMP API for tile percolation (C/C++)

Third, it should not bring much complexity to code generation and should also not cause too much runtime overhead. According to these criteria, the tile percolation programming API is designed as OpenMP pragma directives, shown in Figure 3.4(a).

The tile percolation API has two new pragma directives: the `percolate` directive and the `tile` directive. The percolate directive specifies a *percolation region*, which is a block of code. At the beginning of the percolation region, on-chip storage will be reserved for all data tiles that will be percolated into the on-chip memory. Then, all or some of the data tiles accessed in the percolation region will be moved into the on-chip memory and the corresponding computations will be performed there. At the end of the percolation region, data tiles that contain the results of the computations are written back to the off-chip memory (if necessary) and the reserved on-chip memory are freed.

The tile directive, on the other hand, provides the detailed information (type,

shape, size, etc.) of the data tiles that will be percolated into the on-chip memory. It is always contained in a percolation region. The tile directive specifies a *tile region*, in which there is a set of `for` loops delimiting the bounds of the data tiles. In the tile directive, following the key word `tile` is a list of *tile descriptors*. The tile descriptors are divided into three groups by the key words **`ro`**, **`wo`**, and **`rw`**, which are the *clause* names that identify *read-only*, *write-only*, and *read-write* data tiles. At the beginning of the percolation region, data tiles specified in the **`rw`** clause will be copied from the off-chip memory into the on-chip memory (after the on-chip memory allocation). At the end of the percolation region, data tiles specified in the **`rw`** and **`wo`** clauses are copied back to their home locations in the off-chip memory. For data tiles specified in the **`ro`** clause, they will be copied into the on-chip memory at the place where the **`ro`** clause is specified. They will not be copied back to the off-chip memory at the end of the percolation region. The associated code in the percolation region are adjusted to access the on-chip data tiles in the computations.

The format of the tile descriptor is similar to the declaration of a multi-dimensional array variable, except that each of the tile descriptor *dimension specifier* is a 3-tuple, not a singleton. The tile descriptor tells the compiler how the data tile is carved out from the multi-dimensional data array that hosts it. To make the thesis easy to follow, we call the multi-dimensional data array that hosts the current data tile as its *host array*. The tile descriptor contains the complete information of the host array. Therefore, the number of dimension specifiers in the tile descriptor is the same as the dimension of the host array. It is not necessarily the same as the dimension of the data tile.

For a dimension specifier $[j_i, a_i, L_i]$ (see Figure 3.4(a)), "L_i" is the size of the i_{th} dimension of the host array (not the data tile). "a_i" is the blocking/tiling size of the i_{th} dimension of the host array. This parameter is used to carve out the data tile from its host array. Normally, if the dimension of the data tile is the same as the dimension of the host array, "j_i" is the index variable of a **`for`** loop in the tile region that traverse the

i_{th} dimension of the data tile. If the dimension of the data tile is smaller than its host array, the element j_i in some dimension specifiers becomes trivial. Currently, we force the programmers to put a "$*$" there as a place holder to let the compiler know that the current dimension of the host array has been squashed away in the dimension space of the data tile. An intuitive example of this case is the expression `A[0][i][j]` guarded by loop `i` and loop `j`. It actually represents a 2-D plane, although the expression has 3 dimension specifiers.

The tile descriptor functions like a template and the associated `for` loops instantiate this template. To make the code generation easy, currently, a writable tile descriptor (specified in the **`rw`** or the **`wo`** clause) can only has one instantiation. The read-only tiles (specified in the **`ro`** clause) can have multiple instantiations. Example is given in Figure 3.5. To put it in a simple way, roughly, the percolate directive and the tile directive tell the compiler *where* the data tiles will be percolated and the tile descriptors tell the compiler *how* the data tiles are percolated. The code example that shows the usage of the tile percolation API is in Figure 3.5. The detailed explanation will be presented in the next sub-section.

3.3.2 Code Generation

The code in Figure 3.5 shows how to use the tile percolation API. The pragma at line 1 is the canonical `parallel for` directive that specifies how the computation iterations are distributed among the parallel threads. The pragma at line 5 is a percolate directive and line 8 is a tile directive. The percolate directive specifies a percolation region, from line 6 to 16. The tile directive specifies a tile region, from line 10 to 15, in which there are there data tiles, represented by "`A[i,b,n][k,b,n]`", "`B[k,b,n][j,b,n]`", and "`C[i,b,n][j,b,n]`". The first two tiles are read-only and the last one is both readable and writable in the current percolation region. They direct the compiler to generate the correct data percolation and computation code.

```
 0
 1 #pragma omp parallel for collapse(2)
 2 for (ii=0; ii<n; ii+=b)
 3   for (jj=0; jj<n; jj+=b)
 4   {
 5     #pragma omp percolate
 6     {
 7       for (kk=0; kk<n; kk+=b)
 8       #pragma omp tile ro (A[i,b,n][k,b,n],B[k,b,n][j,b,n]) \
 9                        rw (C[i,b,n][j,b,n])
10       {
11         for (i=ii; i<min(ii+b,n); i++)
12           for (j=jj; j<min(jj+b,n); j++)
13             for (k=kk; k<min(kk+b,n); k++)
14               C[i][j]+=A[i][k]*B[k][j];
15       }
16     }
17   }
18
```

Figure 3.5: Pseudo Code of the Tile Percolation Example

The tile descriptor "`C[i,b,n][j,b,n]`" specifies a data tile contained in the host array `C`, a 2D $n \times n$ matrix. In this tile descriptor, "`C`" provides the name of the host array, which also tells the compiler the type of the data element of the tile. "`n`" in the dimension specifier tells the size of the each dimension of the host array. "`b`" reveals how the matrix is tiled. "`i`" and "`j`" are two index variables that inform the compiler that the `for` loops at line 11 and 12 are used to construct the data tile. Since the lower and upper bounds of "`i`" and "`j`" are fixed in the current percolation region, there is only one instantiation for this tile template (i.e. descriptor). The clause name "`rw`" indicates that this data tile will be read and written in the current percolation region. So, it will be copied into the on-chip memory at line 6, where the percolation region starts. It will also be copied out to off-chip memory at line 16, where the percolation region ends.

Similarly, data tile "`A[i,b,n][k,b,n]`" and "`B[k,b,n][j,b,n]`" are contained in host array "`A`" and "`B`", which are also 2D $n \times n$ matrix. Since both of them are read-only data tiles, they are copied into the on-chip memory at line 8, where they

are specified in the **ro** clause. They do not need to be copied back to the off-chip memory at the end of the percolation region. Because the lower and upper bounds of "k" are changing (line 7), as we may notice, there are multiple instantiations for these two tile descriptors. All instantiations of the same data tile will reuse the same memory block allocated to it. The example is shown in Figure 3.6.

Figure 3.6 presents the code generated for the tile percolation program in Figure 3.5. First, it allocates on-chip memory for all three data tiles (line 5 to 7). This is achieved by calling the runtime routine `_sram_malloc`, which is inserted in by the compiler. The size of the data tile is calculated by multiplying each of its dimension size, which is obtained from its blocking size. This guarantees that the memory block allocated is big enough to hold the corresponding data tile. If the memory allocations succeed, the read-write data tiles will be copied into the on-chip memory by calling the runtime library routine `_copy2Don` (line 16). Otherwise, no data movement happens and the program execution jumps to the original code (line 12), where computations are performed on off-chip data tiles (line 36).

The other two read-only data tiles are percolated into the on-chip memory between the `for` loops at line 18 and 25. This location corresponds to the place in the original code where they are specified in the **ro** clause. The `for` loops between line 25 and 28 perform matrix multiplication on "`_AA[][]`", "`_BB[][]`", and "`_CC[][]`", which are all located in the on-chip memory. After one kernel computation (line 25 to 28) is finished, the new instantiation of "`_AA[][]`" and "`_BB[][]`" are copied from the off-chip memory into the on-chip memory and are stored in the same memory block. Then it begins the next iteration. Before exiting the percolation region, the **rw** data tile "`_CC[][]`" is copied back to its home location in the off-chip memory (line 32). Meanwhile, the on-chip memory storage allocated to all the percolated data tiles are freed.

To generate the code like in Figure 3.6, the compiler needs to handle three tasks: **(1)** generate code for managing on-chip memory; **(2)** generate code for managing memory

```
 1 {
 2   /* Enter the percolation region.
 3      Allocate on-chip memory for all data tiles  */
 5   _CC=(float *)_sram_malloc(sizeof(float)*b*b);
 6   _AA=(float *)_sram_malloc(sizeof(float)*b*b);
 7   _BB=(float *)_sram_malloc(sizeof(float)*b*b);
 8
 9   if (_CC==NULL || _AA==NULL || _BB==NULL)
10   {
11     _sram_free(_AA); _sram_free(_BB); _sram_free(_CC);
12     goto orig;
13   }
14
15   /* Copy "rw" data tiles from off-chip memory
        to on-chip memory */
16   _copy2Don(sizeof(float), _CC, &C, n, n, ii, jj, \
                                  min(b,n-ii),min(b,n-jj));
17
18   for (kk=0; kk<n; kk+=b)
19   {
20     /* Copy "ro" data tiles from off-chip memory
          to on-chip memory */
21     _copy2Don(sizeof(float), _AA, &A, n, n, ii, kk, \
                                    min(b,n-ii),min(b,n-kk));
22     _copy2Don(sizeof(float), _BB, &B, n, n, kk, jj, \
                                    min(b,n-kk),min(b,n-jj));
23
24     /* on-chip calculation */
25     for (i=0; i<min(b,n-ii); i++)
26         for (j=0; j<min(b,n-jj); j++)
27             for (k=0; k<min(b,n-kk); k++)
28                 _CC[i][j]+=_AA[i][k]*_BB[k][j];
29   }
30
31   /* copy out the results back to off-chip memory */
32   _copy2Doff(sizeof(float), _CC, &C, n, n, ii, jj, \
                                   min(b,n-ii),min(b,n-jj));
33   _sram_free(_AA); _sram_free(_BB); _sram_free(_CC);
34   goto out;
35
36 orig:
37      /* Original code with out percolation */
38         ...
39 out:
40 }
```

Figure 3.6: Code generation example for tile percolation (Pseudo Code)

copy; **(3)** adjust the computation code to access on-chip data tiles. We leave the discussion of the first two items to the next sub-section, because they are mostly related to the runtime. Here, we focus on the third problem.

Adjusting the computation code to access on-chip data tiles includes two sub-tasks: **(i)** calibrate the lower and upper bounds for each `for` loop that is involved in traversing the elements of the data tile; **(ii)** update the tile access expressions accordingly. These tasks are easy because the data tile is copied as one 2D array from its home location (in which, the elements are physically scattered in memory) into a piece of physically contiguous memory block (in which, the elements are consecutive). We only need to know the base address of the memory block and the size of each dimension of the data tile. The value of the tile's dimension size can be easily obtained from its tile descriptor. The base address of the memory block that assigned to the current data tile can be obtained from the corresponding runtime function call (`_sram_malloc`). With this information, it is easy for the compiler to generate the correct code. Most of time, we just perform a kind of simple 1-to-1 replacement. For example, the new lower bound of a `for` loop is always set to zero and the new upper bound is calculated by subtracting the old lower bound from the old upper bound.

3.3.3 Runtime Support

As we have mentioned in the last section, the tile percolation runtime needs to handle the on-chip memory allocation and the memory copy for the percolated data tiles. We provide a set of routines (with clear interface) in the runtime library for the compiler. The compiler, accordingly, would insert the required runtime function calls in the program during code generation.

The runtime routines `_sram_malloc` and `_sram_free` are responsible for allocating on-chip memory for the percolated data tiles. To allocate the correct memory storage for the tile, we need to know three values: (i) the number of dimensions of the tile; (ii) the size of each dimension; and (iii) the type of each data element. The number

of dimensions of the tile is the number of non-trivial dimension specifiers in its tile descriptor. The dimension size is always set to the blocking size. This guarantees that the allocated memory block is big enough to hold any instantiation of the tile descriptor. The type of the data element is obtained from the name of the tile descriptor.

For each percolation region, the ”all-or-none” policy is adopted in memory allocation. The program either continues execution after *all* of its memory allocation requests were satisfied, or, if *any* of its memory allocation request failed, it jumps to the original code to perform the computations on the off-chip data tiles. Because the compiler guarantees that all memory allocations occur at the beginning of the percolation region and all memory frees occur at the end of the percolation region, the memory allocation failure would not cause dead lock among the concurrent OpenMP threads. This greatly simplifies design of the runtime support and also simplifies code generation in compiler.

For the memory copy task, we provide the set of runtime routines presented in Figure 3.7. Currently, we support tile percolation for 1D-, 2D-, and 3D-array. They can cover most of the practical cases. Each kind of multi-dimensional array has its own memory-copy routines (see Figure 3.7(a)). The routines with the suffix ”*on*” are used to copy data tiles from off-chip memory to on-chip memory, while the routines with the suffix ”*off*” are used to copy data tiles from on-chip memory to off-chip memory. The parameters that are required in the address calculation in these memory-copy routines are supplied in the argument list. We use the ”long argument list” instead of the ”packed argument structure” because we try to avoid inserting unnecessary dynamic memory allocation function calls in code generation. We feel that generating dynamic memory allocation code is tricky and error-prone.

According to our design, there are some assumptions on the on-chip and the off-chip data tiles. For the on-chip data tile, it must reside in a contiguous memory block. For the off-chip data tile, it must be a sub-block of a multi-dimensional array and the multi-dimensional array must also reside in a contiguous memory block. Because the percolated

```
_copy1Don(sz,_on,_off,D1,x,b1)
_copy1Doff(sz,_on,_off,D1,x,b1)
_copy2Don(sz,_on,_off,D1,D2,x,y,b1,b2)
_copy2Doff(sz,_on,_off,D1,D2,x,y,b1,b2)
_copy3Don(sz,_on,_off,D1,D2,D3,x,y,z,b1,b2,b3)
_copy3Doff(sz,_on,_off,D1,D2,D3,x,y,z,b1,b2,b3)
...
```

(a) The runtime routines for memory copy

`_copy[1D|2D|3D]on`:	Runtime routines that copy the off-chip data tile into the on-chip memory;
`_copy[1D|2D|3D]off`:	Runtime routines that copy the on-chip data tile back to the off-chip memory;
`sz`:	Size of the element of the data tile;
`_on`:	The address of the on-chip memory block used to hold the percolated data tile;
`_off`:	The address of the home location of the percolated data tile in the off-chip memory;
`D1,D2,D3`:	The size of each dimension of the percolated data tile, from the lowest dimension to the highest dimension;
`x,y,z`:	Position of the percolated data tile in the host array. It is represented by the coordinate of its first element in the host array, from the lowest dimension to the highest dimension;
`b1,b2,b3`:	Blocking size of each dimension of the host array, from the lowest dimension to the highest dimension;

(b) The explanation of the runtime routines

Figure 3.7: The runtime routines for on-chip and off-chip memory copy

data tile is only a sub-block in its host array, its memory layout is not contiguous. Physically, it consists of many data strips (or rows) that are separated by an equal distance. Hence, the parameters provided in the argument list should be able to be used to calculate the start address and the size of each data strip in the tile. With the above assumptions, it is easy to interpret the argument list of the memory-copy routines. For example, the routine `_copy2Don` copies a 2D data tile from off-chip memory to on-chip memory. The argument "`_off`" gives the start address of its host array, (i.e. the address of the first element). "`D1`" and "`D2`" tell the dimension size of the array. "`x`" and "`y`" specify the coordinates of the data tile in its host array. "`b1`" and "`b2`" reveal the blocking size of each dimension of the host array. "`b1`" and "`b2`" are the default size (of each dimension) of the percolated data tile. To handle the corner cases, the routine will calculate the effective size at runtime. The size of the data tile element is shown in "`sz`". With the above information, it is easy for the runtime routine to calculate the address and size of each data strip and copy it around with the optimized library code. All these arguments are provided in the tile percolation directives and can be easily extracted out by the compiler.

In essence, the arguments listed above characterize the position and the size of a data tile and its host array accurately. It doesn't matter whether this data tile and its host array are real multi-dimensional array (in the language sense) or not. As long as all the array access expression are affine functions of the loop indices, they can be declared (physically) as an 1D array but accessed by the programmers (logically) as a multi-dimensional array. The compiler will take care of the convoluted indices calculation.

3.4 Experiments

We have evaluated tile percolation with four scientific kernels (SAXPY, SASUM, SGEMV, and SGEMM) [82] and two NAS benchmarks (EP and MG). Tile percolation was implemented through source-to-source program transformation and was prototyped in the Omni compiler [83]. The experiments were conducted on the FAST simulator [70],

Table 3.1: FAST Simulation Parameters (Courtesy to Juan del Cuvillo!)

Component	**# of units**	**Params./unit**
Threads	160	single in-order issue, 500MHz
FPUs	80	floating point/MAC, divide/square root
I-cache	16	32KB
SRAM (on-chip)	160	30KB
DRAM (off-chip)	4	256MB
Crossbar	1	96 ports, 4GB/s port
A-switch	1	6 ports, 4GB/s port

an execution-driven and binary-compatible C64 simulator with accurate instruction timing. FAST accurately simulates the functional behavior and other hardware components such as thread units, on-chip and off-chip memory, and 3D-mesh network, which are shown in Table 3.1 [84].

Table 3.2 [3] gives the detailed timing of each operation simulated in the FAST simulator [84]. exe is the execution time in the function unit and $delay$ represents the latency before the result of the instruction becomes available to the depending instruction.

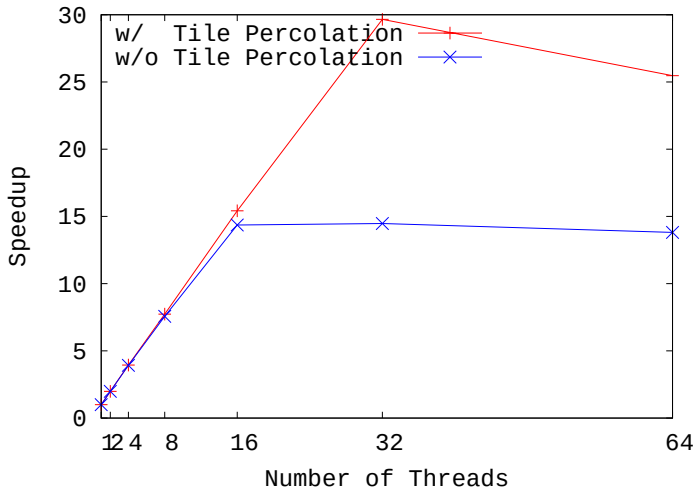
The preliminary experiment results are shown in Figure 3.8, Figure 3.9, and Figure 3.10. After applying tile percolation, the speedup of all testcases get significant improvement. The greatest improvement happens on SGEMM [4]. This testcase has $O(n^3)$ floating-point operations but only access $O(n^2)$ data. However, without reusing the data that have been brought into on-chip memory by the previous computations, the program has very poor scalability. Because it would have $O(n^3)$ number of memory accesses going into the off-chip memory. This would quickly exhaust the off-chip bandwidth. Without using on-chip memory, its diminishing return is 2-thread. After applying the tile percolation

[3] Courtesy to Juan del Cuvillo!

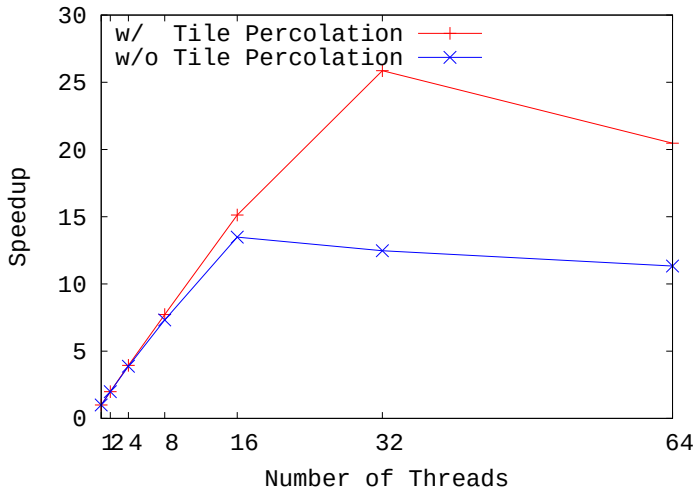
[4] We use 256×256 matrix, the data tile is 16×16

Table 3.2: Instruction Timing of FAST Simulator (Courtesy to Juan del Cuvillo!)

Instruction type	exe	$delay$
Bit gather	1	1
Branches	2	0
Count population	2	0
Integer multiplication	1	6
Integer division signed	1	69
Integer division unsigned	1	68
Integer remainder signed	1	70
Integer remainder unsigned	1	69
Move indirect register	3	0
Floating add, subtract and conv.	1	5
Floating multiplication	1	6
Floating multiply and add	1	11
Floating divide double	1	63
Floating divide single	1	34
Floating square root double	1	62
Floating square root single	1	33
Floating mult. and accumulate	1	6
Memory operation (local SRAM)	1	2
Memory operation (global SRAM)	1	31
Memory operation (off-chip DRAM)	1	57
All other operations	1	0

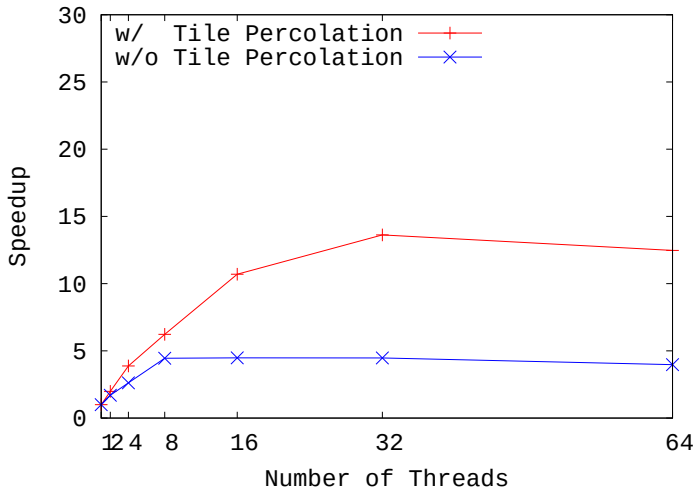


(a) SASUM

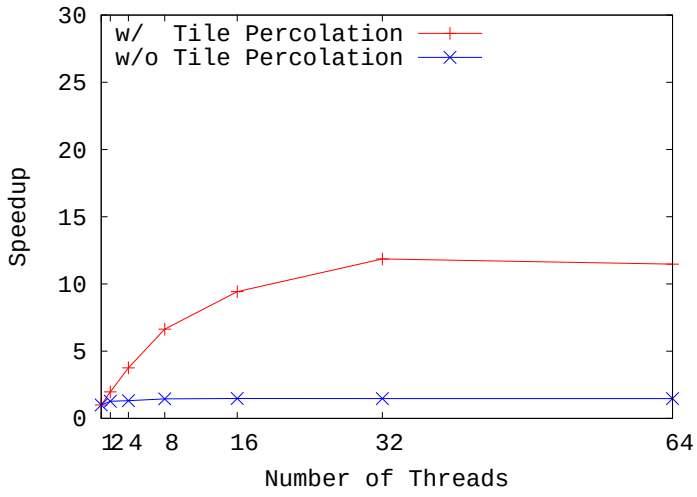


(b) SAXPY

Figure 3.8: Experiment Results of SASUM and SAXPY: Comparison of Speedup

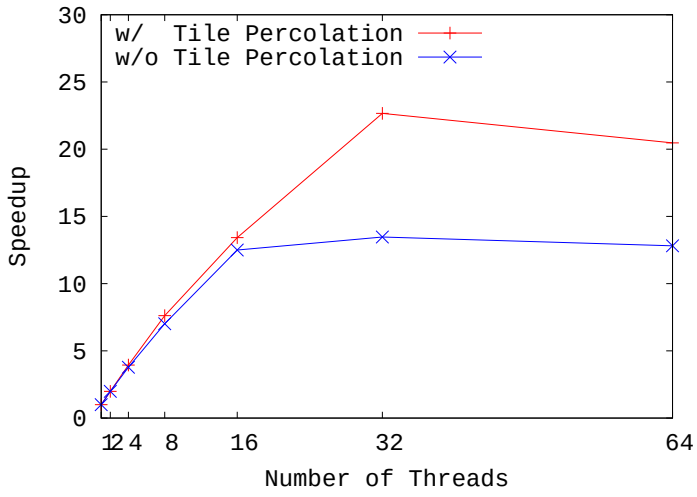


(a) SGEMV

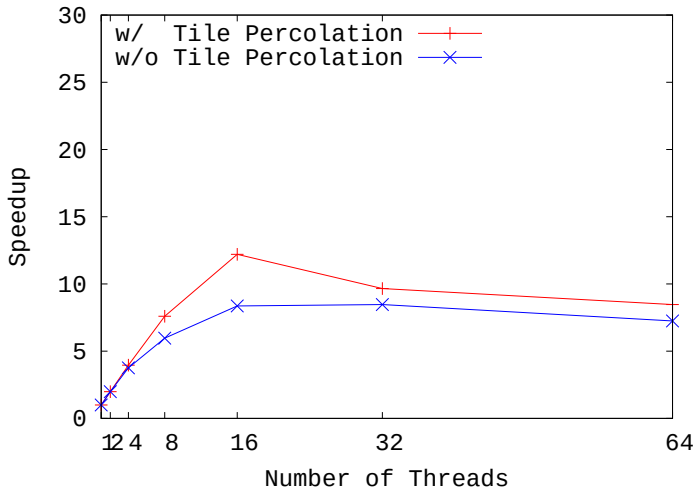


(b) SGEMM

Figure 3.9: Experiment Results of SGEMV and SGEMM: Comparison of Speedup



(a) EP



(b) MG

Figure 3.10: Experiment Results of EP and MG: Comparison of Speedup

optimization, the number of memory accesses has been reduced to $O(n^2(1 + 2n/b))$. Its speedup increased from less than 4 to around 12. For other testcases, their floating-point computations are $O(n^2)$ (SGEMV) or $O(n)$ (EP). So their speedup enhancement is not as big as SGEMM.

An interesting finding is that, without applying tile percolation, most testcases' speedup diminishing return point is at 16-thread. They are SASUM, SAXPY, EP, and MG. The speedup diminishing return point of SGEMV is 8-thread, while for SGEMM, it is 2-thread. For SASUM, its memory accesses and floating-point operations are the same. This reveals that, without on-chip data reuse, the off-chip bandwidth would be saturated when there are more than running 16 threads.

3.5 Summary

Writing a parallel program for multicore processor is already a very difficult task. It is even more difficult if the multicore processor has user-managed memory hierarchy, like the IBM Cyclops-64 processor. On this kind of processor, the programmers not only need to take care of program parallelization, but also need to tackle data movement. Although many efforts have been made to develop automatic data movement code generation [85, 80, 79], it only proves its efficiency on a limited class of problems.

In this chapter, we have proposed a semi-automatic approach to data movement code generation. This novel approach is termed as *tile percolation*. It provides the programmers with a set of OpenMP-like directives. The programmers can annotate the their programs with these directives to tell the compiler *where* and *how* data movement should be performed. Accordingly, the compiler will generate the optimized data movement code and the correct computation code based on the information provided in the tile percolation directives. That way, the programmers can save themselves from writing tedious and error-prone data movement code.

Tile percolation is a kind of OpenMP *tile aware parallelization* technique [32] developed for the IBM Cyclops-64 multicore processor. As far as the author is aware, this is

the first research that try to develop pragma directives for data movement code generation in OpenMP. The tile percolation directives are orthogonal to the canonical OpenMP parallelization directives. This chapter shows that the tile percolation directives can be used together with the traditional OpenMP parallelization directives. Meanwhile, they can also be used independently in the parallel programs written with Pthread library. Experiments conducted on the Cyclops-64 processor show that tile percolation can enhance the utilization of the Cyclops-64 on-chip memory, which turns out to improve the performance and scalability of the programs. This implements an efficient *performance porting* for OpenMP programs developed for the traditional SMP system.

Chapter 4

THREAD-LEVEL DECOUPLED ACCESS/EXECUTION

Cyclops-64 is a many-core processor with user-managed memory hierarchy. For OpenMP programs running on this processor, a frequently used computing paradigm is: (i) copy data into on-chip memory; (ii) perform computations on the chip; (iii) copy results back to off-chip memory. Obviously, hiding memory copy latency is very crucial to the performance of this computing paradigm. The traditional solution is to use the asynchronous DMA transfer to overlap computation and memory copy, like the IBM CELL processor. However, DMA is not supported in the Cyclops-64 processor. Therefore, in this chapter, we propose a software solution, called **T**hread-**L**evel **D**ecoupled **A**ccess/**E**xecution (TL-DAE for short). It is a data-driven execution model for OpenMP programs running on the Cyclops-64 processor. The TL-DAE execution model is inspired by the canonical decoupled architecture. In our design, data movements and computations are decoupled implicitly by OpenMP compiler. At runtime, two different groups of threads are spawned: the *computation* threads and the *percolation* threads. Computation threads execute computation code while percolation threads execute data movement code. The execution of computation thread and percolation thread can slip with respect to each other, so percolation thread can run further ahead than computation thread and fetch data for it. In this chapter, we will not only develop the runtime techniques used to implement the TL-DAE execution model, but also propose the required TL-DAE programming interface that is used by OpenMP compiler to generate the decoupled code. We have evaluated the TL-DAE execution model by using two OpenMP task benchmarks. Experimental results show significant performance enhancement.

4.1 Introduction

The IBM Cyclops-64 (C64) [70, 11] is a many-core processor with user-managed memory hierarchy. As shown in Figure 1.1, the C64 chip has 160 homogeneous processing cores. The chip has 512KB instruction cache but no data cache. Instead, each core contains a small amount of SRAM (5.2MB in total) that can be configured into either Scratchpad Memory (SPM), or Global Memory (GM), or both in combination. Off-chip DRAM are attached onto the crossbar switch through 4 on-chip DRAM controllers. All memory modules are in the same address space and can be accessed directly by all processing cores [13]. However, different segments of the memory address space have different access latencies and bandwidths. Apparently, the on-chip memory is faster and has huge access bandwidth, while the off-chip memory is slower and has very limited access bandwidth. See Figure 1.2 for the detailed parameters of the C64 memory hierarchy.

Given an segmented memory model like in the C64 processor, a *3-step computation paradigm* [26, 86] is frequently used, i.e. **(1)** copy data from off-chip memory to on-chip memory; **(2)** perform computations on the data in on-chip memory; **(3)** copy computation results back to off-chip memory Because of the lengthy off-chip memory access latency, it is desired to hide the memory copy latency by overlapping computations and data movements. On C64, the major bottleneck of the multi-level memory hierarchies is between the on-chip and off-chip memory (like in all other processors), so our discussion will only focus on the data movement across this interface.

Usually, the overlapping is achieved by using double buffering [26] and asynchronous DMA transfer that supported in hardware, like we have seen in the IBM CELL processor [87, 88]. However, the hardware design for DMA is very complex and the runtime overhead of DMA is disproportionally high [27]. In [27], authors report that, on a 3.2GHz CELL processor, for a single DMA (128 bytes, 128-byte aligned), the DMA *setup* time is $297.7ns$ ($\approx$ 900 cycles). But the real DMA *transfer* only takes $6.09ns$ (≈ 18 cycles). Taking all latencies into account, it costs more than 1000 cycles to transfer

a 128-byte data block from main memory to local storage in the CELL processor. Similar numbers can also be observed on multi-DMA and DMA-list. Due to these reasons, DMA is not supported in C64. Therefore, we must resort to some kind of software solutions to overlap computation and data movement operations.

In this chapter, we propose such a software solution, which is termed **T**hread-**L**evel **D**ecoupled **A**ccess/**E**xecution (TL-DAE for short). It is inspired by the original hardware based DAE [28, 29], in which memory access (operands fetch and results store) and computation execution are *architecturally* decoupled and thus can be maximally overlapped. Not like the hardware based DAE, TL-DAE is developed as a software execution model for OpenMP programs running on the C64 processor. In our design, data movement code and computation code are decoupled implicitly by OpenMP compiler at compile time. At runtime, two different groups of threads are spawned: the *computation* threads and the *percolation* threads. Computation thread runs computation code while percolation thread runs data movement code. The execution of computation thread and percolation thread can slip with respect to each other, so percolation thread can run further ahead than computation thread and fetch data for it. To achieve dynamic load balancing among the threads, the work-stealing policy [89] is used to schedule both computation tasks and percolation tasks. Besides, a computation thread will not be scheduled until all the data it needs in computation are copied from off-chip memory to on-chip memory. Hence, TL-DAE is also considered as a data-driven execution model for OpenMP programs running on C64.

To help OpenMP compiler decouple the program, we propose the TL-DAE programming interface for the the programmers. The TL-DAE programming interface is a set of OpenMP *tile aware parallelization* [30] pragma directives. Programmers can use these directives to annotate their programs to specify where and how data movement would be performed. OpenMP compiler, accordingly, will interpret these directives and generate the correct decoupled data movement code.

The major contributions of this chapter are as follows: (a) The design and implementation of the Thread-Level Decoupled Access/Execution (TL-DAE) model for OpenMP programs running on a many-core processor with user-managed memory hierarchy; (b) The design of the TL-DAE programming interface that can be used to help OpenMP compiler to generate decoupled code; (c) Detailed experiments and performance analysis of the OpenMP task benchmarks that use the TL-DAE execution model. The experimental results demonstrate the effectiveness of TL-DAE execution model.

The remainder of this chapter is organized as follows. In Section 4.1, we will use a motivating example to demonstrate why thread-level decoupled access/execution is necessary. In Section 4.3, we present the design and implementation of the TL-DAE execution model, including TL-DAE API, TL-DAE code generation, and TL-DAE runtime support. We report our experimental results in Section 4.4 and draw our conclusion in Section 4.5.

4.2 Motivation

In this section, we will use a motivating example to demonstrate why thread-level decoupled access/execution is necessary and also possible for OpenMP programs running on the C64 processor. We will also derive the framework of the TL-DAE execution model in this section.

The code in Figure 4.1 is the major part of the *sparseLU* (Sparse LU Decomposition) benchmark from the Barcelona OpenMP Task Suite (BOTS) [90]. In order not to waste memory storage on sub-matrices with all zero elements, the *sparseLU* program uses a 2-level hierarchical data structure to store the sparse matrix, in which the *all-zero* sub-matrix is shrank to a `nil` pointer. Figure 4.1(b) shows the details of this data structure. The *sparseLU* program performs computations only on *non-zero* sub-matrices [1].

[1] In the sparseLU benchmark, all diagonal sub-matrices are initialized to non-zero matrix

```
 1     ...
 2 #define NB 100
 3 #define B 100
 4     ...
 8 void fwd(float *diag, float *col) { ...; }
 9 void bdiv(float *diag, float *row) { ...; }
10     ...
41 int main(int argc, char* argv[]) //uses A
42 {
43           ...
44     float *A[NB][NB];
45           ...
46 #pragma omp parallel single
47
48     for (kk=0; kk<NB; kk++) {
49         lu0(A[kk][kk]);
50  {
51      for (jj=kk+1; jj<NB; jj++)
52          if (A[kk][jj] != NULL)
53 #pragma omp task firstprivate(kk, jj) shared(A)
54              fwd(A[kk][kk], A[kk][jj]);
55
56      for (ii=kk+1; ii<NB; ii++)
57          if (A[ii][kk] != NULL)
58 #pragma omp task firstprivate(kk, ii) shared(A)
59              bdiv (A[kk][kk], A[ii][kk]);
60  }
61
62 #pragma omp taskwait
77       ...
78 }
79
```

Figure 4.1: The OpenMP Version of the *sparseLU* Source Code

```
 5 void bdiv(float *diag, float *row)
 6 {
 7     int i, j, k;
 8
 9     for (i=0; i<B; i++)
10       for (k=0; k<B; k++) {
11         row[i*B+k] = row[i*B+k] / diag[k*B+k];
12         for (j=k+1; j<B; j++)
13           row[i*B+j] = row[i*B+j] - row[i*B+k]*diag[k*B+j];
14         }
15 }
```

Figure 4.2: `bdiv`: the OpenMP *task* function used in *sparseLU*

```
30 void fwd(float *diag, float *col)
31 {
32     int i, j, k;
33
34     for (j=0; j<B; j++)
35       for (k=0; k<B; k++)
36         for (i=k+1; i<B; i++)
37           col[i*B+j] = col[i*B+j] - diag[i*B+k]*col[k*B+j];
38 }
```

Figure 4.3: `fwd`: the OpenMP *task* function used in *sparseLU*

The code is parallelized by using the OpenMP task directive [91]. According to the specification of OpenMP 3.0 [5], each time the program execution reaches line 53 (and also line 58), an OpenMP task is constructed and is put in the task pool. OpenMP threads in the current team fetch tasks from task pool and execute the functions specified in the task. In this example, function `fwd()` will be executed, with sub-matrix `A[kk][kk]` and `A[kk][jj]` as real arguments. The same scenario repeats at line 58.

To port this piece of OpenMP code to C64, we need to consider the heterogeneity of the C64 memory model and try to leverage the fast on-chip memory as much as we can. This is concerned with moving the associated sub-matrices into and out of on-chip memory. A naive approach is to insert data movement code manually in function `fwd()`,

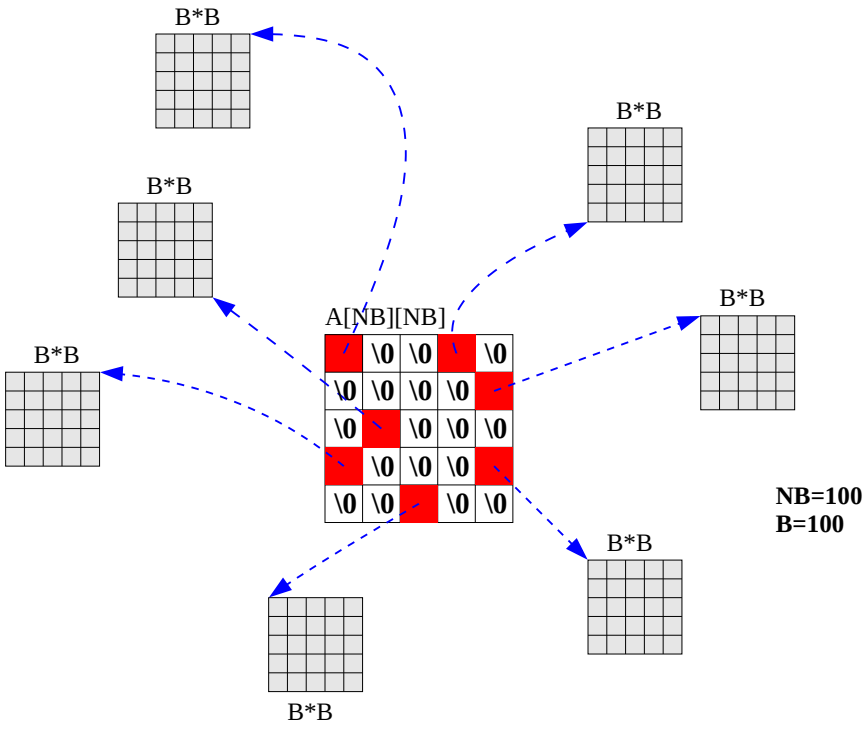


Figure 4.4: The 2-level hierarchical data structure used in *sparseLU* code

```
1 void fwd(float *diag, float *col)
2 {
3     Copy the matrix pointed to by *diag into on-chip memory;
3     Copy the matrix pointed to by *col into on-chip memory;
4     Perform computations on the on-chip copies of the matrices
5     pointed by *diag and *col;
6     Copy computation results back to off-chip memory;
7 }
```

Figure 4.5: An Intuitive Approach: Synchronous Data Movement

just right before and after the computation code. See pseudo code in Figure 4.5. The problem of this approach is obvious. Data movement code are embedded in the main trunk of the thread. The real computations can not start execution until all data it needs are copied into on-chip memory. There is no overlapping between data movement and computation.

The inadequacy of this approach motivates us to explore a new method, i.e. completely decouple computations and data movements of the program and use independent threads to execute them. This is possible for the C64 processor because there are 160 thread units on the chip. One can always find extra or idle thread units that can be used for data movement. By using this method, we will reduce the memory latency hiding problem to a thread scheduling problem, which can be handled efficiently by OpenMP runtime library.

4.3 Thread-Level Decoupled Access/Execution

In this section, we first give an overview on the TL-DAE model. Then we introduce the techniques we proposed to implement the TL-DAE execution model, including the programming interface, code generation, and the TL-DAE runtime design.

4.3.1 Overview

The first step of this approach is to ”separate” computation code and data movement code in the program. Actually, there is *no* data movement code in the original program (that’s why we surround ”separate” in quotation marks). We need to add data movement code into the program either manually or automatically (i.e. let OpenMP compiler generate it). Although it is not difficult for programmers to write data movement code and they also have more flexibility to choose how to manage on-chip memory, we found that such kind of flexibility brings huge amount of difficulties to the compiler to interface the data movement code with OpenMP runtime library. For this reason, we prefer to shipping this task from programmer to compiler.

To help OpenMP compiler generate data movement code, we propose the TL-DAE programming interface. The TL-DAE API is actually a kind of *Tile Aware Parallelization* (TAP) [92] technique proposed for OpenMP programs running on multi/many-core processors with user-managed memory hierarchy. The TL-DAE programming interfaces are OpenMP pragma directives that can be used by programmers to annotate their program, to tell OpenMP compiler where and how data movement would be performed. Compiler will interpret these information and generate correct data movement code. We will present more details about TL-DAE API in the next section.

In the second step, the execution of the decoupled program will be user-managed by the TL-DAE runtime system. The design of TL-DAE runtime follows the idea of OpenMP task runtime. Instead of using one task pool, we use two kinds of task pool: the *percolation* task pool and the *computation* task pool. The master thread first spawn a team of slave threads (i.e. the computation threads) and a group of *percolation* threads. Then, it ”executes” the program, constructs percolation tasks and put them in the percolation task pool. The structure of the percolation task contains enough information for the percolation thread to construct computation tasks. The execution of percolation threads follow OpenMP task model. After a percolation thread finishing one task, it would construct a computation task from the percolation task structure and put it into the computation task pool. The computation threads consume the tasks in the computation task pool. After a computation thread finishing its job, it will construct a percolation task which tells the the percolation thread how to store the computation results back to off-chip memory. This task will be put in the percolation task pool by the current computation thread. We present the design of TL-DAE runtime in details in the next section.

4.3.2 TL-DAE Programming Interface

The purpose of TL-DAE programming interface is to grant programmers a certain extent of power to direct OpenMP compiler to generate decoupled data movement code. In tradition, this part of work is assumed completely by compiler [93, 85], by using the

canonical static analysis techniques, such as *loop nest optimization* [94, 95]. For array-intensive applications, the data access patterns can be easily inferred from loop constructs. However, we found that this approach relies too much on the capabilities of compiler. Sometimes a program's data access pattern is very difficult to analyze by compiler. For instance, for the program in Figure 4.1(a), before the OpenMP compiler can decide the shape and size of the data block pointed to by `float *diag` (the first formal argument of function `fwd()`), it must be able to analyze the code in both function `main()` and function `fwd()` at the same time [2]. This indicates that we need to incorporate some kind of IPA (Inter-Procedure Analysis) techniques [96] in the current OpenMP compiler. As far as we know, such kind of IPA technique does not exist yet. Besides, developing such kind of IPA technique in OpenMP compiler will greatly complicate the compiler design while would achieve only limited effect on a very few number of benchmarks, just like the current *auto-parallelization* techniques.

In order to solve this problem, programmer's interferences could help a great lot, i.e. programmer provides a certain amount of hints by annotating the code while compiler use these hints to perform the required code transformation. This idea is the same as the design philosophy of the OpenMP programming model. Based on this idea, we propose the TL-DAE API as OpenMP pragma directive. See Figure 4.6.

We introduce a new directive: `percolate`. It must be used together with the OpenMP `task` directive. Semantically, the `percolate` directive declares a percolation region. Operand data blocks are copied into on-chip memory at the beginning of the percolation region (including allocation of on-chip memory resources). Result data blocks are copied back to off-chip memory at the end of the percolation region (including release of on-chip memory resources). In between, the corresponding task function is executed. All three parts are executed by independent threads in the TL-DAE model.

[2] This is further complicated by the facts that `fwd()` may be called by more than one function.

```
#pragma omp percolate [altfunc(FUNC_NAME)] ARG_DESC[, ARG_DESC]
ARG_DESC    :: TILE_DESC|SCALAR_DESC
TILE_DESC   :: <rd|wt|rw, ADDR, TY, DIM, std|emb,           \
                          [SZ][[SZ]]|[SZ,HSZ][[SZ,HSZ]]>
SCALAR_DESC :: <SCALAR_NAME=VALUE>
```

(a) TL-DAE percolation API

`percolate:`	Directive name. It specifies a percolation region.
`altfunc:`	Clause name. The ”altfunc” clause declares an alternative function that will be used to perform the computation after the associated data tiles were copied into on-chip memory.
`ARG_DESC:`	Argument descriptor.
`TILE_DESC:`	Tile argument descriptor.
`rd|wt|rw:`	Percolation attributes. `rd` indicates that the data tile will be only read into on-chip memory in the current percolation region; `wt` indicates that the data tile will be only write out of on-chip memory in the current percolation region; `rw` indicates that the data tile will be both read into and write out of on-chip memory.
`ADDR:`	Starting address of the tile.
`TY:`	Tile element type. It specifies the type of the tile element.
`DIM:`	Tile dimension. It specifies the dimension of the tile.
`std:`	Tile attribute. It specifies that the current tile is a *standalone* tile.
`emb:`	Tile attribute. It specifies that the current tile is an *embedded* tile.
`SZ:`	Size of the corresponding dimension of the tile.
`HSZ:`	Size of the corresponding dimension of the host multi-dimension array of the embedded tile.
`SCALAR_DESC:`	Scalar argument descriptor.
`SCALAR_NAME:`	Name of a particular scalar formal argument.
`VALUE:`	Adjusted initial value that assigned to the corresponding scalar argument.

(b) Explanation of the TL-DAE tile percolation API

Figure 4.6: The OpenMP API for TL-DAE tile percolation (C/C++)

The `percolate` directive provides programmers a mechanism to specify which data blocks will be copied into on-chip memory, which data blocks will be copied back to off-chip memory, and what they look like (shape, size, and position). Besides, if the original *task function* (function guarded by the `task` and `percolate` directives) can no longer be used to perform correct computations because of the changes made on data layout, programmers can provide an alternate task function through the `altfunc` clause. Once the `altfunc` is used, the original function guarded by the `task` directive is ignored. The *real* task function becomes the one specified in the `altfunc` clause.

Whether or not the real task function is changed, the associated *argument descriptors* are listed on the `percolate` directive. Each *actual* argument used by the real task function has a corresponding argument descriptor placed on that list, in the same order as it in the function's formal argument list. Although some information about the real arguments can also be found in the task function's declaration or call site, this redundancy would greatly simplifies code generation in the next step.

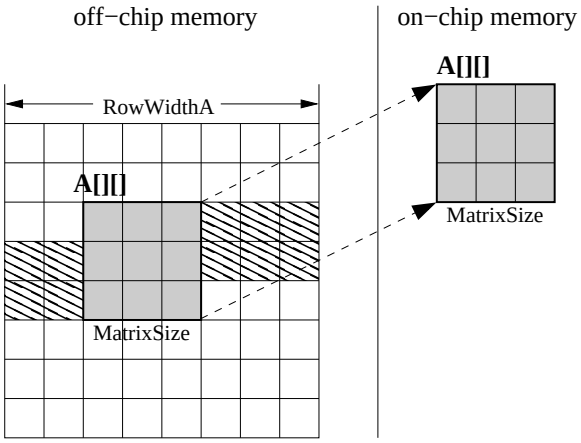
We define two types of argument descriptors: *tile descriptor* and *scalar descriptor*. Tile descriptor is used to depict actual arguments that are multi-dimensional data tiles. Scalar descriptor is used to depict actual arguments of all other types.

Tile descriptor specifies the starting address (`ADDR`), the element type (`TY`), and the dimension (`DIM`) of a particular data tile. Moreover, it also specifies whether the current data tile will be read into on-chip memory *only* (`rd`); or will be written out from on-chip memory *only* (`wt`); or will do both (`rw`). [3]. In our current design, data tiles are categorized into two types: *standalone* data tile (indicated by the keyword `std`) and *embedded* data tile (indicated by the keyword `emb`). A standalone data tile's existence, as its name implies, is independent on any other data object. Semantically, standalone data tile is self contained. Physically, standalone data tile is allocated in contiguous memory

[3] We consider copy into on-chip memory as *read* and copy out of on-chip memory as *write*.

```
 0
 1 void MultiplyByDivideAndConquer (REAL *C,
 2              REAL *A, REAL *B, unsigned MatrixSize,
 3              unsigned RowWidthC, unsigned RowWidthA,
 4              unsigned RowWidthB, int AdditiveMode);
 5
 6  void FastAdditiveNaiveMatrixMultiply (REAL *C,
 7              REAL *A, REAL *B, unsigned MatrixSize,
 8              unsigned RowWidthC, unsigned RowWidthA,
 9              unsigned RowWidthB);
10
```

(a) Functions performing divide-and-conquer matrix-matrix multiplication



(b) Diagram of embedded matrix A

Figure 4.7: An example of *embedded* data tile used in strassen benchmark

space. Examples of standalone data tile are the sub-matrices (pointed to by `A[kk][jj]`) used in the sparseLU program. See Figure 4.1. `[SZ][[SZ]]` specifies the size of each dimension of the data tile. `SZ` can be a constant and can also be a variable. An 1-dimension data tile with dimension size equaling 1 is actually a scalar. Programmers can use this property to represent a big C `struct` in tile descriptor so that it can also be copyin/copyout as a multi-dimensional data tile.

The embedded data tile, as its name implies, is itself part of another bigger data object. In our design, this bigger data object must also be a multi-dimensional data array

of the same type. Logically, the embedded data tile is treated as a computation data unit, like in most divide-and-conquer algorithms [97]. Physically, however, it is distributed inside the body of a bigger multi-dimensional data array. So, embedded data tile is usually not in contiguous memory space. An example of the embedded data tile is shown in Figure 4.7. The code in Figure 4.7(a) is from the *strassen* program of the Barcelona OpenMP task benchmark suite. Both of the two functions are used in the recursive divide-and-conquer algorithms [97]. They are called at the position where recursion termination condition is satisfied. In order to access the elements in the sub-matrix, we not only need to know the position, shape, and size of the sub-matrix, but also need to know the shape and size of the host multi-dimensional data array. In our design, we deem that the dimension of the embedded data tile is the same as the host multi-dimensional data array. So, in the tuple `[SZ,HSZ][[SZ,HSZ]]`, `SZ` is the dimension size of the embedded data tile and `HSZ` is the size of the corresponding dimension of the host multi-dimensional data array. These information are used by compiler to generate correct data movement code.

As Figure 4.7(b) shows, moving an embedded data tile from its original location in off-chip memory to on-chip memory will change its data layout. Usually, the embedded data tile will be copied row by row from non-contiguous locations in off-chip memory into contiguous locations in on-chip memory. Although information in the tuple `[SZ,HSZ][[SZ,HSZ]]` is enough for compiler to generate correct data movement code [4]. It is far from enough for compiler to adjust every data-tile-element reference statement in the original task function. Sometimes, this is too difficult to be carried in

[4] Actually, compiler does not generate the read data movement code. It uses the parameters in the tuple `[SZ,HSZ][[SZ,HSZ]]` to generate stub code to call the correct runtime library function. It is the runtime function that does the real job.

compiler [5]. That is why we provide the `altfunc` clause. Programmer can write an *on-chip version* of the task function and supersede the original one by specifying it in the `altfunc` clause.

Sometimes, programmers do not need to provide an alternate task function. Because the data-tile-element references in the original task function have already been parametrized. Programmers only need to set those parameters correctly when calling the original task function. See examples in Figure 4.7(a). The arguments `RowWidthC`, `RowWidthA`, and `RowWidthB` are used by the function to calculate correct offset of each data tile row. For data tile `A[][]`, `B[][]`, and `C[][]` that are copied into contiguous memory space, we can still call the original function but use the adjusted real arguments. For example, in both functions, because the on-chip data tile `A[][]` is *embedded in itself*, `RowWidthA` equals to `MatrixSize` of `A[][]`. So, we can fix the value of the real argument `RowWidthA` to `MatrixSize`. This also applies on `RowWidthB` and `RowWidthC`. Programmers can specify the fixup value in the scalar descriptor of the corresponding scalar argument. Currently, we only support this in built-in data types. See an use case in Figure 4.9, which is an example of embedded data tile. Figure 4.8 gives an example of standalone data tile.

4.3.3 TL-DAE Code Generation

The design of TL-DAE code generation follows one rule: let TL-DAE runtime do most of the dirty jobs and leave as less work for code generation as possible. Following this rule, most of the real works, like task creation, task scheduling, data movement, and task execution are performed by TL-DAE runtime routines. Therefore, the purpose of TL-DAE code generation is to connect the OpenMP program to the TL-DAE runtime

[5] Two difficulties. OpenMP compiler need to perform some kind of inter-procedure analysis before it can change any code in the task function; Second, OpenMP compiler needs to deal with the code versioning problem. Both issues are far above an OpenMP compiler's capability.

```
13
14 for (jj=kk+1; jj<NB; jj++)
15   if (A[kk][jj] != NULL)
16     #pragma omp task firstprivate(kk, jj) shared(A)
17     #pragma omp percolate  <rw,A[kk][jj],double,2,std,[B][B]>,\
18                            <rd,A[kk][kk],double,2,std,[B][B]>
19     fwd(A[kk][kk], A[kk][jj]);
20     ...
21
```

Figure 4.8: TL-DAE API Example 1: applied on standalone data tile

```
0
1 #pragma omp task firstprivate(MatrixSize) shared(C,A,B)
2 #pragma omp percolate
3  <rw,C,double,2,emb,[MatrixSize,RowWidthC][MatrixSize,RowWidthC]>,\
4  <rd,A,double,2,emb,[MatrixSize,RowWidthA][MatrixSize,RowWidthA]>,\
5  <rd,b,double,2,emb,[MatrixSize,RowWidthB][MatrixSize,RowWidthB]>,\
6  <MatrixSize>, <RowWidthC=MatrixSize>, <RowWidthA=MatrixSize>,    \
7     <RowWidthB=MatrixSize>
8 FastAdditiveNaiveMatrixMultiply(REAL *C, REAL *A, REAL *B,
9            unsigned MatrixSize, unsigned RowWidthC,
10           unsigned RowWidthA, unsigned RowWidthB);
11
```

Figure 4.9: TL-DAE API Example 2: applied on embedded data tile

routines. The compiler will insert stub code in the program to prepare the required arguments and call the correct runtime library functions with the right arguments. Such a design philosophy not only simplifies code generation in compiler but also makes the TL-DAE framework more flexible.

In TL-DAE code generation and also runtime, the most frequently used data structure is `tldae_task{}`. This data structure, as its name implies, represents a task unit that will be scheduled and executed in the TL-DAE execution model. It contains complete information that tells TL-DAE runtime how the current task will be performed. It is also the major interface between the original OpenMP program and TL-DAE runtime. So, one

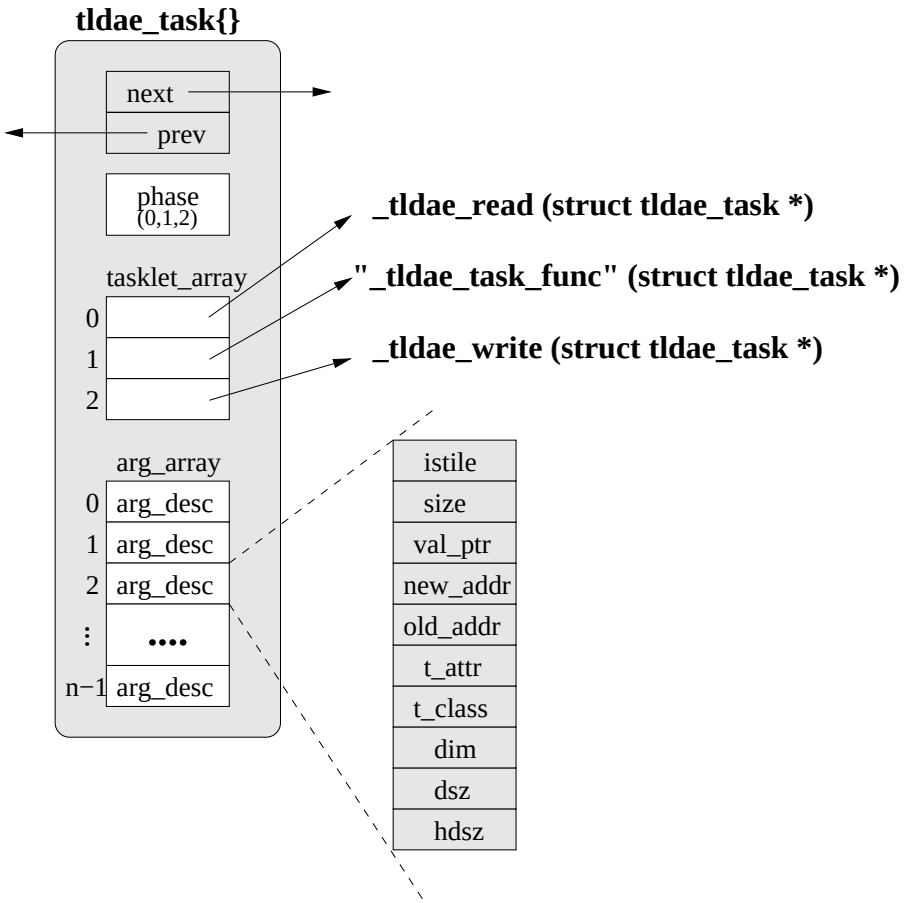


Figure 4.10: Diagram of the `tldae_task{}` structure

```
61
62 struct tldae_task
63 {
64    struct tldae_task *prev;
65    struct tldae_task *next;
66
67    /* index point to tasklet_array[], to indicate the current
68     * task function that will be executed
69     */
70    int phase;
71
72    /* this array stores function pointers that point to task
73     * functions currently, only three functions are supported
74     */
75    void (*tasklet_array[DEFAULT_MAX_TSK])(struct tldae_task *tsk);
76
77    /* this array stores the argument descriptors for
78     * each argument specified on the argument list of the
79     * task function.
80     */
81    struct arg_desc arg_array[DEFAULT_MAX_ARG];
82 };
83
```

Figure 4.11: Definition of `tldae_task{}`

of the major work of TL-DAE code generation is to generate this data object and insert correct runtime function invocations on it. Diagram in Figure 4.10 presents the internal design of the `tldae_task{}` data structure. As shown in the diagram, `tldae_task{}` consists of three parts: *task phase* (`.phase`), *tasklet array* (`.tasklet_array[]`), and *argument descriptor array* (`.arg_array[]`). Figure 4.11 gives the source code of the definition of `tldae_task{}`.

- **Line 70:** `.phase` denotes the current phase of the task. It is an index into the tasklet array at line 75.
- **Line 75:** `.tasklet_array[]` stores the pointers to the functions that would be executed during each phase of the current task. All functions recorded by tasklet array take a pointer to `tldae_task{}` as the only argument and this pointer must

point to the current task. `.tasklet_array[]` and index `.phase` together decide which function would be called if the current task is scheduled for running. In the current design, only three tasklets are allowed [6]. `.tasklet_array[0]` and `.tasklet_array[2]` point to two fixed functions: `_tldae_read()` and `_tldae_write()`. while `.tasklet_array[1]` points to a function generated by compiler, which is not the same in different cases. In Figure 4.10 we use a temporary common name `_tldae_task_func` to refer to all of them. `_tldae_task_func` is simply a wrapper of the original computation function (In Figure 4.8, it is function `fwd()`.). Figure 4.13 and Figure 4.14 will give an example of `_tldae_task_func()`.

- **Line 81:** `.arg_array[]` contains the detailed information of every argument that will be used in the execution of the `_tldae_task_func()` tasklet. Each argument in the argument list of the original computation function has a slot (i.e. `struct arg_desc{}`) in this array, no matter the argument is a scalar or a data tile pointer. The `_tldae_read()` and `_tldae_write()` tasklets will scan `.arg_array[]` and *copyin*/*copyout* the corresponding data tiles from/to off-chip memory. `_tldae_task_func()` uses these arguments to call the original task function to perform the computations.

Figure 4.12 shows the source code of the definition of `arg_desc`, i.e. argument descriptor. This piece of code is part of the TL-DAE runtime. Compiler need to create an argument descriptor for each argument in the argument list of the original task function. Figure 4.13 gives such an example of argument descriptor creation.

- **Line 25:** `.istile` is a flag to tell whether this is a scalar argument descriptor or a tile argument descriptor.

[6] It can be extended to have more tasklets and can deal with more complicated execution patterns.

```
18
19 #define DEFAULT_MAX_TSK 6
20 #define DEFAULT_MAX_DIM 6
21 #define DEFAULT_MAX_ARG 10
22
23 struct arg_desc
24 {
25    bool istile;        /* scalar or tile */
26
27    int size;           /* decided by the type of tile element */
28
29    void* val_ptr;      /* point to arg value */
30
31    void* new_addr;     /* the new start address of the tile after
32                         * it has been copied into on-chip memory.
33                         */
34
35    void* old_addr;     /* the original start address of the tile
36                         * in off-chip memory
37                         */
38
39    enum tile_attr {
40        rd = 0;         /* read-only  */
41        wt = 1;         /* write-only */
42        rw = 2;         /* read-write */
43    } t_attr;
44
45    enum tile_class {
46        std = 0;        /* standalone tile */
47        emb = 1;        /* embedded tile */
48    } t_class;
49
50    int dim;            /* tile dimension */
51
52    int dsz[MAX_DIM];   /* size of each dimension of the tile */
53    int hdsz[MAX_DIM];  /* size of each dimension of the
54                         * multi-dimensional array that hosts
55                         * this tile
56                         */
57 };
58
```

Figure 4.12: Definition of `arg_desc{}`

- **Line 27:** For a scalar argument, `.size` records the size of the argument; for a data tile argument, `.size` records the size of the data tile element. This information comes from the `TY` field in the tile descriptor (`TILE_DESC`) of the guarding pragma.

- **Line 29:** `.val_ptr` points to the memory location where the argument value is stored. For a tile argument, it points to `.new_addr` (line 31) in which the new on-chip memory address of the data tile would be stored. For a scalar argument, it still points to `.new_addr`, to where the value of the scalar argument will be copied. In this case, the rest fields of `arg_desc` are useless. They are only useful when the argument is a data tile.

- **Line 31:** If the argument is a tile, `.new_addr` stores the new address of the data tile after it has been copied into the on-chip memory. Otherwise, `.new_addr` stores the value of the scalar argument. If the size of the scalar argument is bigger than the size of `.new_addr`, it will occupy the bytes following it, because the rest fields in the `arg_desc{}` structure is useless, so is undefined [7].

- **Line 35:** For a data tile argument, `.old_addr` stores its original address in off-chip memory. For a scalar argument, the value of this field is undefined.

- **Line 39:** For a data tile argument, `.t_attr` specifies the attributes of the tile, i.e. whether it is *read-only*, *write-only*, or *read-write*. For a scalar argument, the value of this field is undefined.

[7] Currently, we use the same type of data structure for both data tile argument and scalar argument. This may waste some memory if the argument is a small scalar. But it simplifies code generation. In the future, this can be improved by having two types of data structures for data tile and scalar argument.

- **Line 45:** For a data tile argument, `.t_class` specifies whether it is a *standalone* data tile or an *embedded* data tile. For a scalar argument, the value of this field is undefined.

- **Line 50:** For a data tile argument, `.dim` records the dimensionality of the data tile. For a scalar argument, the value of this field is undefined.

- **Line 52:** For a data tile argument, `.dsz[]` stores the size of each dimension of the tile. It is undefined if the argument is a scalar.

- **Line 53:** For an *embedded* data tile, `.hdsz[]` stores the size of each dimension of the host multi-dimensional array. For other kind of arguments, the content of this array is undefined.

After finishing the introduction of the above two important data structures `tldae_task{}` and `arg_desc{}`, we now present an example of code generation using the real OpenMP program. The original OpenMP source code is shown in Figure 4.1. We will use the TL-DAE programming API to perform tile aware parallelization on the task function `fwd()` at line 54. The usage of the TL-DAE API is shown in Figure 4.8. The generated code is shown in Figure 4.13. The code is generated from the ROSE source-to-source compiler with our modification to support TL-DAE code generation. The generated code without TL-DAE support from the unmodified ROSE compiler is shown in Figure C.2. We illustrate the generated TL-DAE source code in the rest of this section.

- **Line 274:** `OUT__4__1527()` is a function outlined by the ROSE compiler to deal with the `parallel` directive shown at line 46 in Figure 4.1. ROSE compiler will also outline all the private variables that would be used by each OpenMP thread and put them in a separate data structure, i.e. `void *__out_argv[]`, which is an array that stores pointers pointing to each private variable. This greatly simplifies

```
274 void OUT__4__1527__(void **__out_argv)
275 {
276   int *ii = (int *)(__out_argv[3]);
277   int *jj = (int *)(__out_argv[2]);
278   int *kk = (int *)(__out_argv[1]);
279   float *(*A)[100UL][100UL] = \
280                         (float *(*)[100UL][100UL])(__out_argv[0]);
281   for ( *kk = 0;  *kk < 100; ( *kk)++) {
282     lu0((( *A)[ *kk])[ *kk]));
283 //#pragma omp taskgroup
284 {
285       for ( *jj = ( *kk + 1);  *jj < 100; ( *jj)++)
286         if ((( *A)[ *kk])[ *jj]) != ((float *)(((void *)0))))
287         {
288           struct tldae_task *tsk = TLDAE_get_task();
289           ((tsk)->phase) = 0;
290           (((tsk)->tasklet_array)[0]) = (_tldae_read);
291           (((tsk)->tasklet_array)[1]) = (OUT__3__1527__);
292           (((tsk)->tasklet_array)[2]) = (_tldae_write);
293           ((((tsk)->arg_array)[0]).istile) = 1;
294           ((((tsk)->arg_array)[0]).size) = (sizeof(float));
295           ((((tsk)->arg_array)[0]).val_ptr) = \
296                   (void *)(&((((tsk)->arg_array)[0]).new_addr));
297           ((((tsk)->arg_array)[0]).old_addr) = \
298                   (void *)(( *A)[ *kk])[ *kk]);
299           ((((tsk)->arg_array)[0]).t_attr) = 0;
300           ((((tsk)->arg_array)[0]).t_class) = 0;
301           ((((tsk)->arg_array)[0]).dim) = 2;
302           (((((tsk)->arg_array)[0]).dsz)[0]) = 100;
303           (((((tsk)->arg_array)[0]).dsz)[1]) = 100;
304           ((((tsk)->arg_array)[1]).istile) = 1;
305           ((((tsk)->arg_array)[1]).size) = (sizeof(float));
306           ((((tsk)->arg_array)[1]).val_ptr) = \
307                  (void *)(&((((tsk)->arg_array)[1]).new_addr));
308           ((((tsk)->arg_array)[1]).old_addr) = \
309                  (void *)(( *A)[ *kk])[ *jj]);
310           ((((tsk)->arg_array)[1]).t_attr) = 2;
311           ((((tsk)->arg_array)[1]).t_class) = 0;
312           ((((tsk)->arg_array)[1]).dim) = 2;
313           (((((tsk)->arg_array)[1]).dsz)[0]) = 100;
314           (((((tsk)->arg_array)[1]).dsz)[1]) = 100;
315           TLDAE_schedule_task(tsk);
316         }
317      ....
318 }
```

Figure 4.13: Code Generation Example of `fwd()` in sparseLU: the outlined function that creates TL-DAE tasks

the generation of outlined function interface because all outlined functions generate by ROSE only take one argument of the same type.

- **Line 284:** the code from line 284 to line 318 corresponds to the code from line 50 to 78 in Figure 4.1. This piece of code is generated by ROSE compiler to implement the semantics of the OpenMP `task` directive and the TL-DAE `percolate` directive in Figure 4.8.

- **Line 288:** For each OpenMP task, we generate a `tldae_task{}` structure for it. A `tldae_task{}` pool with a small number of free `tldae_task{}` structure is maintained by the TL-DAE runtime in the on-chip SRAM. The function `TLDAE_get_task()` allocate a free TL-DAE task unit from the `tldae_task{}` pool.

- **Line 289-292:** first the task phase is initialized to zero. Then the tasklet array is initialized. The first tasklet is `_tldae_read`, which copies critical data tiles into on-chip memory. The second tasklet is `OUT__3__1527__`, an outlined wrapper function that executes real task computation function. Figure 4.14 shows the source code of this function. The third tasklet is set to `_tldae_write`. It writes the computation results back to off-chip memory. The task is performed starting from the first tasklet to the last tasklet in the array. The traversal of this array is controlled by the variable `.phase`. `.phase` is increased by one after the current tasklet is finished.

- **Line 293-314:** the code between 293 and 314 is used to setup the argument descriptors of the current task function - `fwd()`. This task function has two arguments. Both of them are 2-D standalone data tiles. The values of right hand size of the assignments are coming from the information in the TL-DAE `percolate` directive. See Figure 4.8, line 17 & 18.

```
263
264 void OUT__3__1527__(struct tldae_task *tsk)
265 {
266     fwd((float *)(*(void **)((((tsk)->arg_array)[0]).val_ptr)),\
267         (float *)(*(void **)((((tsk)->arg_array)[1]).val_ptr)));
268
269     ((tsk)->phase)++;
270     TLDAE_schedule_task(tsk);
271     return;
272 }
273
```

Figure 4.14: Code Generation Example of `fwd()` in sparseLU: the outlined TL-DAE task function

- **Line 315:** After the creation of a new TL-DAE task unit, the TL-DAE runtime function `TLDAE_schedule_task()` is called to insert the newly created task unit into one of the task queue owned by a certain percolation thread. See Figure 4.15 for details.

- **Line 317:** The rest of the generated code is performing the same operations on another task function `bdiv()`. Since the these two pieces of code are very similar, we omitted it here to avoid wasteful duplication of effort.

Figure 4.14 shows the source code of the outlined wrapper function `OUT__3__1527__`.

- **Line 266:** the real task function `fwd()` is called to perform the task works. The purpose of outlining a wrapper function is to: **(a)** simplify the generation of the versatile argument list of the task function. For here, the argument list string is very simple. **(b)** have a place to place to put post-execution code.

- **Line 269:** increment `.phase` by one to advance to the next tasklet. Next time if the current task is scheduled for running, it will run the next tasklet in the tasklet array.

- **Line 270:** reschedule the same task unit. See Figure 4.15 for details.

As shown in Figure 4.15, the job of function `TLDAE_schedule_task()` is to insert the task unit into the corresponding task queue according to its phase (line 88, 95, & 102). If the task is finished, the runtime will reclaim the data structure `tldae_task{}`, so it can reused by the next new task.

Figure B.1 in Appendix gives a set of important TL-DAE runtime functions

4.3.4 TL-DAE Runtime Support

The TL-DAE runtime library provides two set of functions. One set of functions deal with data tile movement between on-chip memory and off-chip memory. The other set of functions handle task creation, scheduling, and execution. These functions cooperate with each other at runtime to implement decoupled access/execution for the program.

In the previous subsection, we already introduced two functions: `_tldae_read` and `_tldae_write`. They are actually driver functions that control how data movements are performed. These two functions traverse the argument array in the current `tldae_task`. For each data tile they encounter, different data movement functions are called based on the tile's dimension, type (`std` or `emb`), and attributes (`rd`/`wt`/`rw`). Arguments used by the particular data movement function are obtained from the corresponding fields of the argument descriptor. With accurate and complete information (*shape*, *size*, *position*) of a data tile, it is easy to write the required data movement functions. Due to the page limits, we omit the implementation details of this part.

There are two kinds of threads in TL-DAE runtime, i.e. *percolation* thread and *computation* thread. Percolation threads perform data movement task while computation threads deal with computations. The two groups are in separate name spaces. They are all spawned by OpenMP master thread [8]. The number of each kind threads can be specified

[8] To simplify code generation, the current implementation spawn all percolation threads and computation threads at the very beginning of the program execution, i.e. at the beginning of the `main` function.

```
 80
 81 void
 82 TLDAE_schedule_task(struct tldae_task *tsk)
 83 {
 84 #define TLDAE_READ_PHASE        0
 85 #define TLDAE_COMPUTE_PHASE     1
 86 #define TLDAE_WRITE_PHASE       2
 87
 88     if (tsk->phase == TLDAE_READ_PHASE) {
 89         /* insert the task into the task queue that belongs
 90          * to one of the percolation threads. The task will
 91          * be inserted at the TAIL of the queue.
 92          */
 93         TLDAE_task_enqueue_read(tsk);
 94     }
 95     else if (tsk->phase == TLDAE_COMPUTE_PHASE) {
 96         /* insert the task into the task queue that belongs
 97          * to one of the computation threads. The task will
 98          * be inserted at the TAIL of the queue.
 99          */
100         TLDAE_task_enqueue_compute(tsk);
101     }
102     else if (tsk->phase == TLDAE_WRITE_PHASE) {
103         /* insert the task into the task queue that belongs
104          * to one of the percolation threads. The task will
105          * be inserted at the HEAD of the queue.
106          */
107         TLDAE_task_enqueue_write(tsk);
108     }
109     else {
110         /* the task is finished. release the memory resources
111          */
112         TLDAE_put_task(tsk);
113     }
114
115     return;
116 }
117
```

Figure 4.15: Code Generation Example of `fwd()` in sparseLU: the TL-DAE task scheduling runtime function

by programmer either through environment variable or through runtime library function. Each thread has its own task queue, which is called local task queue. A thread can access any other threads' task queue, which is termed remote task queue. Lock is provided to guarantee mutually exclusive access on the task queues.

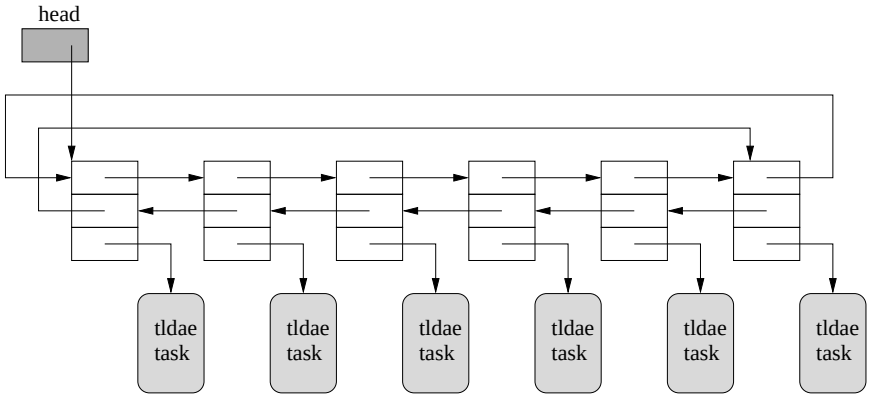


Figure 4.16: Task Queue implemented as a double-linked list

Each element of the task queue stores a pointer to a `tldae_task` data object. The task queue is designed as a circular buffer. `enqueue` and `dequeue` operations can be performed on both the head and the tail of task queue. However, both kind of threads only take a task unit from the head of the task queue. Currently, the task queue is implemented as a double-linked list. Figure 4.16 shows the diagram.

After creating the percolation threads and the computation threads, the master thread traverses the loop (see line 14 in Figure 4.8). In each iteration, it creates a task unit, i.e. a `tldae_task{}` data object. When a `tldae_task` object is created, its task index is set to zero and it is enqueued at the tail of the task queue owned by a percolation thread. Since the task unit's task index is zero, at the first time when it is scheduled for execution by a percolation thread, the thread calls function `_tldae_read()` (pointed to by `task_array[0]`) to copy all `rd`/`rw` data tiles into on-chip memory. After finishing all data movement operations, this percolation thread increases the task index by

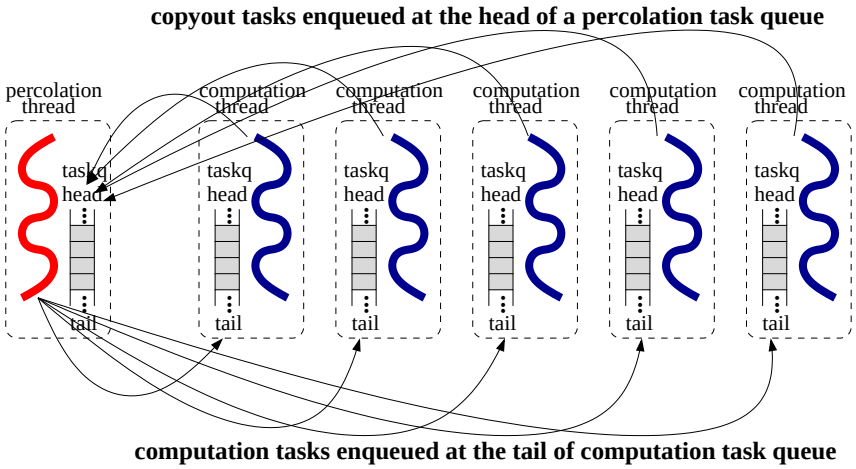


Figure 4.17: 1 Percolation thread and 5 computation threads and their task queues

one and enqueue the same `tldae_task{}` object at the tail of the task queue owned by a computation thread. At the second time when this task unit is scheduled for execution, it is executed by a computation thread which calls `task_array[1]`, i.e. the `_tldae_task_func()`. This function performs the real computation on the data tiles located in on-chip memory. After finishing computation, it increases the task index by one and enqueues the same `tldae_task{}` object at the *head* of the task queue owned by a percolation thread. This time, the `tldae_task{}` object is inserted at the head of the queue (see Figure 4.17) because flushing the result data tiles from on-chip memory to off-chip memory has higher priority than moving data tiles up to on-chip memory. Otherwise, the on-chip memory resources will be exhausted and will cause deadlock. Figure 4.17 shows an example of 1 percolation thread and 5 computation thread. It shows how task units are flowed between percolation thread and computation threads.

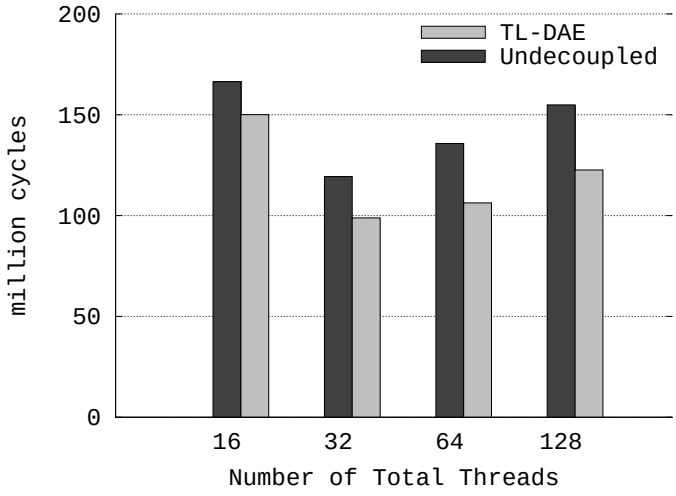
After finishing creating all percolation tasks, the master thread becomes a computation thread and joins other computation threads to perform computation tasks. Each thread works on the task units stored its local task queue. When a new task unit is produced, the producer will use the round-robin policy to insert the new task unit into the

task queue of the corresponding thread group. After a thread finishes all the task units in its local task queue, it will try to steal task unit from other threads of the same group. It also uses the round-robin policy to steal the task unit. These methods are used to achieve load balance among the threads.

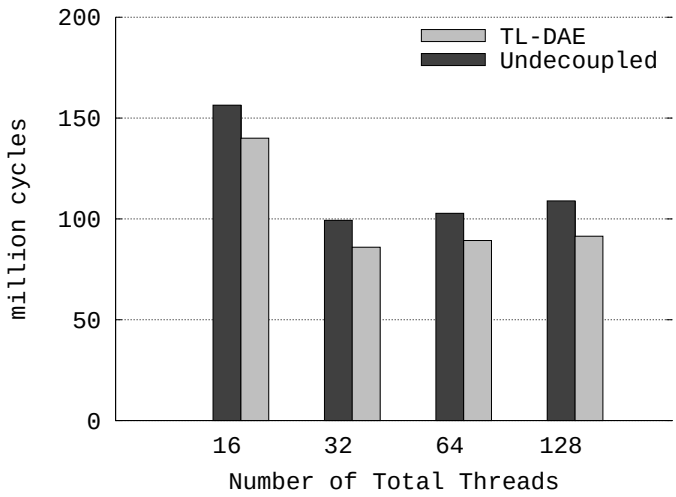
4.4 Experiments

We have used two OpenMP task benchmarks, *sparseLU* and *strassen* to evaluate the TL-DAE execution model. The strassen benchmark was first preprocessed to transform the the recursive invocation of the function to list traversal, just like the *sparseLU* benchmark. We plan to extend the TL-DAE model to accommodate recursive invocation in our future work. The required TL-DAE code generation is achieved by source-to-source translation.The experiments were conducted on FAST [70], an execution-driven C64 simulator with accurate instruction timing. Table 3.1 in Chapter 3 shows the C64 configuration simulated by FAST [84]. Table 3.2 gives the detailed timing of each instruction simulated by FAST [84]. Besides, Figure 1.1(b) shows the detailed latency numbers of the load/store operations when accessing different memory segments. The experimental results are shown in Figure 4.18 and 4.19.

Figure 4.18 (a) and (b) compare the execution time between the code running in TL-DAE execution model and the code running in the base model (Undecoupled, i.e. data movement code embedded in computation code). Both code use the same *total* number of threads. To be more specific, if we use 32 threads to execute the undecoupled code, we use 8 percolation threads and 24 computation threads to execute TL-DAE code. The same number of percolation threads are used in other cases in this experiment. Experimental results show that, the decoupled code execution outperforms the undecoupled code execution in all cases. One of the reason that can explain this is that, when there are a great number of threads performing data movement (in undecoupled code, there are chances that most of the threads are performing data movement at the same time), their competitions for off-chip memory bandwidth would super-linearly increase the latency of data

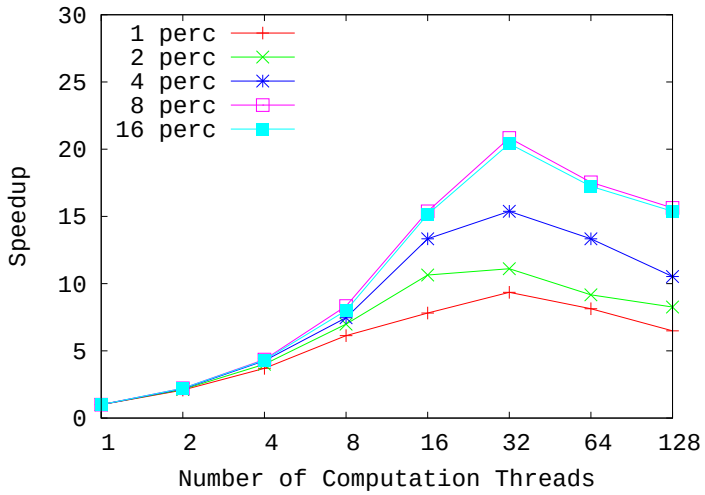


(a) sparseLU (#50 50x50 blocks)

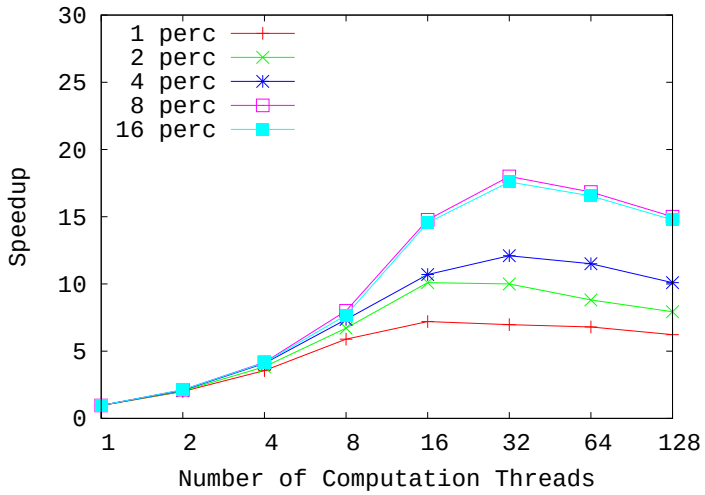


(b) strassen (2048x2048 Matrix)

Figure 4.18: Execution Time Comparison: w/ TL-DAE (8 percolation threads) vs. w/o TL-DAE



(a) sparseLU (#50 50x50 blocks)



(b) strassen (2048x2048 Matrix)

Figure 4.19: Speedup of under different number of percolation threads

transfer. This greatly slows down the threads' computation rate. Another observation is from Figure 4.18 is that, the scalability of both benchmarks are not very good, no matter they are executed in TL-DAE model or undecoupled model. Figure 4.19 clearly shows this.

Figure 4.19 shows the speedup of sparseLU and strassen under different number of percolation threads. Since both benchmarks are data intensive applications, their performances (throughput and scalability) are highly determined by how many data can be transferred from off-chip memory into on-chip memory per unit of time. As shown in both Figure 4.19(a) and (b), when the number of computation threads is below 32, most of the executions get good speedup. This is due to (a) the separation of the off-chip and on-chip memory accesses; (b) the less intensive competition for the shared task queues; and (c) the less intensive competition for the limited on-chip memory capacity. The speedups do not scale after the number of computation threads exceeds 32. The reason is the increased contention for the shared on-chip resources, especially the contention for the on-chip memory storage [9]. The speedup curves for 8 percolation threads and 16 percolation threads almost overlap with each other. This is due to the limited off-chip memory bandwidth, which is saturated at 8 percolation threads. Adding more percolation threads after the saturation point would not make the program run faster. Instead, it might cause performance penalty.

4.5 Summary

In this chapter, we have introduced a new method we developed to hide memory access latency for OpenMP programs running on the IBM Cyclops-64 processor. This new method is named TL-DAE, stands for Thread Level Decoupled Access/Execution. We used an example to explain why decoupled execution is necessary and possible on the C64 processor. We also introduced the techniques used to implement the TL-DAE

[9] On C64, the available global on-chip memory is configured to 2.5M.

execution model, i.e. the TL-DAE programming interfaces and the TL-DAE runtime library. From the experience of developing this execution model and the experimental results, we have the following conclusions:

1. The TL-DAE programming interface is an indispensable part of the TL-DAE execution model that helps OpenMP compiler to generate decoupled program.

2. The thread level decoupled access/execution is an effective execution model to hide memory copy latency on a many-core processor like C64 that has a heterogeneous memory hierarchy.

3. For data intensive applications, off-chip memory bandwidth limit is a more heavy-weight factor in determining the programs' throughput and scalability.

4.6 Related Works

The TL-DAE execution model was inspired by the original hardware based DAE model, which was *Decoupled access/execution* The original DAE model was first proposed as the core hardware technique in decoupled architecture [28, 29, 98]. In decoupled architecture, memory access (operands fetch and results store) and computation execution are architecturally decoupled. An *access processor* (AP) works on the memory access instruction stream and an *execute processor* (EP) works on the computation instruction stream. The two processors communicate data values via architecturally visible queue. The running of AP and EP can *slip* with respect to each other, thus AP can run further ahead than EP and fetch operands for EP. This provides great opportunity for memory latency hiding. Due to this advantage and its hardware simplicity, researchers claimed that the decoupled architecture is more complexity-effective and scalable than the comparable superscalar processors [99]. Later, the initial decoupled architecture were improved by combining with different kinds of multithreading technologies. Dorozhevets' Multi-Threaded Decoupled (MTD) architecture [100, 101] proposes to use speculative multithreading [102] to enable the part of the parallelism that would otherwise be suppressed

by control dependencies; Parcerisa's multithreaded DAE [103, 104] adopts simultaneous multithreading [14] in decoupled architecture to effectively hide function unit latencies; The D^3-machine [105, 106], on the contrary, introduce the concept of decoupled execution into the multithreaded data-flow machine, in which, each actor is decoupled into two parts: the synchronization portion and the computation portion. All of the above works are hardware based DAE and decoupling happens at fine-grain level, i.e. the instruction level. These are quite different from our work which is a software based DAE execution model and decoupling happens at very coarse-grain level.

A set of similar works is software controlled pre-execution or speculative precomputation [107, 108, 109]. These works were extended from the software prefetching technique [110, 111] and were targeted to multithreaded processors. Instead of inserting prefetching instructions in the main thread, compiler (or programmer) constructs a piece of prefetching code (which is termed as *p-slice* or *speculative slice*) and lets a separate thread execute the prefetching code. The thread running the prefetching code is called *helper* thread or *precomputation* thread, which is similar to the *percolation* thread [112] in our work. However, the differences are obvious. The code executed by precomputation thread is carefully extracted from the body of main thread and its main purpose is to calculate (speculatively) the address of the loads that cause the most cache misses. Our percolation thread executes runtime library code and its purpose is to move (with certainty) a specific chunk of data from off-chip memory to on-chip memory. Besides, precomputation thread usually operates on very small number of data values, like 8 bytes or 16 bytes, while percolation thread usually operate on a big chunk of data with pre-defined structure. Usually, there is no communication between precomputation thread and main thread. However, in our work, percolation thread and computation thread communicate with each other through a software queue.

In DSWP [113, 114], a sequential program is splitted into multiple non-speculative threads. One of the thread is dedicated to executing instructions on critical path, which

are the code that traverse the recursive data structure and *produce* the pointers to each node on the recursive data structure. Other threads execute instructions on the off critical path, which *consume* the pointers and perform computations on each node independently. The two kinds of threads communicate via a hardware based *synchronization array*. Here, the thread that execute the code on critical path is also similar to our percolation threads. However, its purpose to discover more ILP in a sequential program.

CellSs [115, 116] proposes an OpenMP-like programming model for the Cell BE architecture, which is similar to the tile aware parallelization proposed for OpenMP in [92]. CellSs also has helper thread at runtime. Its helper thread runs on the PPE side, in parallel with master thread. It is responsible for scheduling active tasks to SPEs and managing the task-dependency graph. It is not responsible for data movement. Data movement is performed by SPE thread via the DMA primitives supported by Cell processor, which is not available in the Cyclops-64 processor [13].

Chapter 5

TILE REDUCTION

In the previous chapters, we have discussed two techniques. One is called *tile percolation*, the other is the *thread-level decoupled access/execution* model. These two *tile aware parallelization* technologies are developed to help compiler to generate data movement code automatically and execute data movement code and computation code concurrently. In this chapter, we will introduce the third tile aware parallelization technique. This technique is not dealing with data movement. However, its purpose is to enable an efficient parallelization which is not possible in the traditional OpenMP program. The name of the new parallelization technique is called *tile reduction*.

Tile reduction is an OpenMP tile aware parallelization technique that allows reduction to be performed on multi-dimensional arrays. This is a natural extension of the *scalar reduction* in the current OpenMP specification. This chapter has three contributions: **(a)** it is the first practice in OpenMP to support parallel reduction on data tiles; **(b)** it presents the methods used to implement tile reduction, which include the OpenMP API extension and the associated code generation techniques; **(c)** it demonstrates the effectiveness of tile reduction with a set of benchmarks. The experimental results show that, in some case, tile reduction can make parallelization more natural and flexible. It not only can expose more parallelism in an OpenMP program, but also can improve its data locality.

5.1 Introduction

Tiling [117, 95, 118, 76] has been used as an effective compiler optimizing technique to generate high performance scientific codes. Tiling not only can improve data

locality for both the sequential and parallel programs [119, 120] , but also can help the compiler to maximize parallelism and minimize synchronization [121, 22] for programs running on parallel machines. Thus, sometimes, it is used by the programmers to hand-tune their scientific programs to get better performance. Tiling is essentially a program design paradigm. It is a natural representation for many important data objects that are heavily used in scientific and engineering algorithms. Scientific code that is written with the concept of tile/tiling in mind usually looks concise and clear, and thus is much easier to understand and less error prone.

Due to these advantages, it is desirable to provide certain high level language constructs in the programming languages to support tile/tiling in program design directly. To meet this requirement, researchers have proposed various designs in many parallel programming languages or sub-languages. The examples include HPF[122], UPC[66], X10[123], ZPL[124], CAF[125], Titanium[126], and HTA[127], which are among the most popular parallel languages. They support the concept of tile aware programming either through the first class language constructs or through library routines with uniform interfaces.

In this chapter, we propose *tile reduction* for the OpenMP programming language. It is an OpenMP tile aware parallelization technique that allows parallel reduction to be performed on multi-dimensional arrays. Its purpose is to enhance the OpenMP API with the concept of tile/tiling so that more data parallelism can be exposed to the OpenMP compiler. It not only grants greater flexibility to the OpenMP compiler to perform more data parallelization, but also brings better data locality into the code.

Basically, reduction is a form of recursive calculation that use mathematically associative and commutative operators to ”aggregate” a set of data. Reduction can be performed in parallel to improve performance. For this reason, many programming languages and sub-languages support parallel reduction. Some examples are UPC [128],

MPI [129], ZPL [130], and OpenMP [5]. According to the current OpenMP API specification, reduction can only be performed on ”named scalar” variables. It cannot be applied on multi-dimensional arrays. We call this kind of reduction *scalar reduction*. In this chapter, we evolve the current reduction parallelization from scalar variables to multi-dimensional arrays, which is termed *tile reduction*. We have extended the traditional `reduction` clause to allow the programmers to annotate their code where tile reduction can be applied. We have also developed the required code generation technique to interpret the new `reduction` clause and generate the required parallel code accordingly. The rest of the chapter is organized as follows. In Section 5.2, we use a motivating example to show why tile reduction is necessary. Section 5.3 will discuss how to implement tile reduction in the OpenMP compiler. We present our experimental data in Section 5.4 and make our conclusions in Section 5.5. The related works are given in Section 5.6.

5.2 Motivation

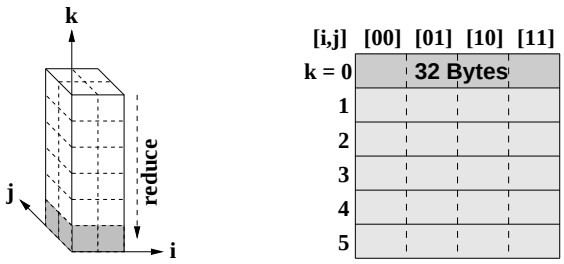
In this section, we use the ”histogram reduction” [131] code as an example to demonstrate the limits of the current OpenMP reduction clause. We will also use the same example to show the advantages of extending *scalar reduction* to *tile reduction*.

Figure 5.1(a) shows the code of the histogram reduction program. The code works on `A[][][]`, a 3-dimensional array with each element containing an 8-byte `long long`. It aggregates all elements along the `k` dimension and stores the results in the `2x2` tile `A[0][][]`. The diagram in Figure 5.1(b) shows these operations. We assume that the cache line size is 32 bytes and that the the array is stored in a row-major order in the memory. Therefore, elements with the same `k` coordinate can be fed into the same cache line, as shown in Figure 5.1(c). There are three nested loops in the code. Each loop traverses one of the `i`, `j`, `k` dimension of the array. Data dependence only exit in loop `k` because of the recursive calculation.

Given the code in Figure 5.1(a), there are many different ways to parallelize it. However, due to the data dependence in loop `k`, we cannot parallelize this loop. Therefore,

```
 1  long long A[][2][2];
 2        ...
 7  for (k=1; k<10000000; k++)
 8      for (i=0; i<2; i++)
 9          for (j=0; j<2; j++)
10              A[0][i][j] += A[k][i][j]
```

(a) Original Histogram Reduction Code



(b) The 3D Diagram (c) A's Memory Layout

Figure 5.1: The Histogram Reduction Example

without changing the code, we can only parallelize loop `i` and `j`, as shown in Figure 5.2(a) and 5.2(b). It is obvious that there are trivial workload and little parallelism in loop `i` and loop `j`. Thus, it is not worthwhile to parallelize these two loops, even while using the `collapse` clause (supported in OpenMP 3.0 [5]).

To get a larger workload and more parallelism, we can interchange the loops manually before parallelizing the code, as shown in Figure 5.3. In Figure 5.3(a) and 5.3(b), the workload that can be assigned to the threads is large enough. However, the available parallelism is still very small (only supports two or four concurrent threads).

Figure 5.4(a) shows a better solution. In Figure 5.4(a), a nested `parallel for` directive is used to parallelize the recursive addition using the `reduction` clause (with trivial code change). Although the code in Figure 5.4(a) can leverage all levels of parallelism in the program, its stride data access pattern would cause a great number of unnecessary cache misses, as shown in Figure 5.4(b). Code in Figure 5.3(a) and 5.3(b) have the

```
0 for (k=1; k<10000000; k++)
1   #pragma omp parallel for
2   for (i=0; i<2; i++)
3     for (j=0; j<2; j++)
4       A[0][i][j] += A[k][i][j]
```

(a) Parallelize loop ”i”

```
0 for (k=1; k<10000000; k++)
1   #pragma omp parallel for collapse(2)
2   for (i=0; i<2; i++)
3     for (j=0; j<2; j++)
4       A[0][i][j] += A[k][i][j]
```

(b) Parallelize loop ”i” and ”j” using the collapse clause

Figure 5.2: Parallelize the Histogram Reduction Program Without Changing the Code

same data locality problem. Apparently, the current OpenMP parallelization techniques cannot harvest the maximum parallelism and data locality in the code at the same time. They suffer from either insufficient parallelism or poor data locality.

The ideal parallelization is shown in Figure 5.5. Logically, the recursive addition can be viewed as being performed on an array of `2x2` data tiles. In theory, these tiles can be added together in parallel by multiple threads, as shown in Figure 5.5(a). In this way, the code can achieve both the maximum parallelism and the best data locality (see Figure 5.5(b)). Besides, from the programmers' angle, this is the most natural way to perform parallelization on this piece of code. However, the current OpenMP specification does not provide any mechanism to support such kind of parallelization. This motivates us to extend the traditional *scalar* reduction to *tile* reduction.

5.3 Tile Reduction

In this section, we will discuss the techniques used to implement tile reduction. They include the extended OpenMP programming interface and the required code generation design. The related runtime support will be mentioned when needed.

```
0 #pragma omp parallel for
1 for (j=0; j<2; j++)
2     for (i=0; i<2; i++)
3         for (k=1; k<10000000; k++)
4             A[0][j][i] += A[k][j][i]
```

(a) Parallelize the outer most loop: ”j”

```
0 #pragma omp parallel for collapse(2)
1 for (j=0; j<2; j++)
2   for (i=0; i<2; i++)
3     for (k=1; k<10000000; k++)
4       A[0][j][i] += A[k][j][i]
```

(b) Parallelize the outer most two loops: ”j” and ”i”

Figure 5.3: Parallelize the Histogram Reduction Program After Performing Loop Interchange

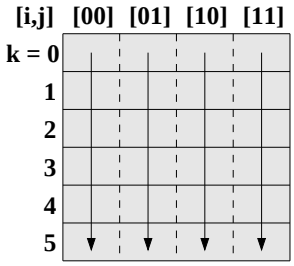
5.3.1 Programming Interface Extension

In order to support tile reduction, we need to extend the current OpenMP programming interface. The extension was made based on three criteria. First, it must be able to cover most of the common cases of tile reduction code. Second, it must be simple and easy to use and provide the programmers with the maximal flexibility. Third, the extension should not complicate the code generation of the OpenMP compiler and the OpenMP runtime. Figure 5.6(a) shows the OpenMP API (C/C++) extension we proposed for the `reduction` clause. Figure 5.6(b) gives a simple example that uses the extended `reduction` clause to parallelize the tile reduction code.

Compared with the current OpenMP API specification, the difference is in the `list` construct. In addition to the ”named scalar” variables, we allow the programmers to put a ”multi-dimensional array” in the `list` construct. This ”multi-dimensional array” is not a real array data structure in the language sense. It is a language construct that conveys important information to the OpenMP compiler. It tells the compiler the shape, the size, and the element type of the tile and how its elements are traversed by the loops.

```
0 #pragma omp parallel for private(sum) collapse(2)
1 for (j=0; j<2; j++)
2   for (i=0; i<2; i++) {
3     sum = 0;
4 #pragma omp parallel for shared(sum) reduction(+:sum)
5     for (k=0; k<10000000; k++)
6       sum += A[k][j][i]
7     A[0][j][i] = sum;
8   }
```

(a) Nested parallelization to harvest more parallelism



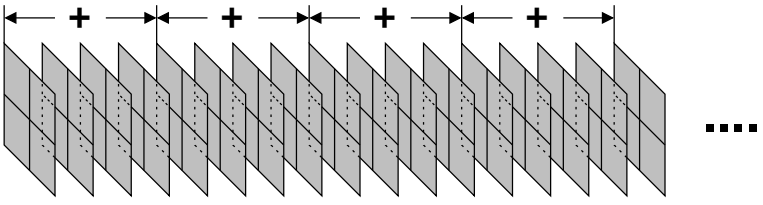
(b) Data access pattern

Figure 5.4: More Parallelization for Histogram Reduction Code

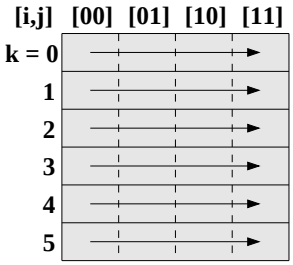
To make the thesis easy to follow, we call the tile under reduction as the *reduction tile*; the ”multi-dimensional array” in the `list` construct as the *tile descriptor*; and the loops involved in performing ”one” recursive calculation as the *reduction kernel loops*. For the example in Figure 5.6(b) [1], the reduction tile is `B[][]`, the tile descriptor is `B[j,0,2][i,0,2]`, and the reduction kernel loops are the `j` and `i` loops (not including the `k` loop, i.e., the parallelized loop). In our design, the shape of the reduction tile must be a rectangle or a high-dimensional rectangle. Triangle or other shapes are not yet supported. The exact shape and size of the reduction tile are determined by the tile descriptor.

The format of the tile descriptor is shown in Figure 5.6(a). It has two parts: the

[1] Index variable k starts from zero because array B[][] is used to store the accumulation results, otherwise it starts from one.



(a) Schema of tile reduction



(b) Better locality

Figure 5.5: The Ideal Parallelization Schema for the Histogram Reduction Code

tile name (i.e., T) and the *dimension descriptor* (i.e., $[j_k, L_k, U_k]...[j_2, L_2, U_2][j_1, L_1, U_1]$). Tile name must be the same as the multi-dimensional array variable on which the recursive calculations are performed. For the example in Figure 5.6(b), this corresponds to the name of the *lhs* variable in line 4, which is B. It tells the OpenMP compiler the data type of the tile element, which must be a built-in scalar type. The dimension descriptor, on the other hand, is an array of 3-tuples. Each 3-tuple corresponds to one dimension of the tile and stores important information of that dimension. These 3-tuples are listed in the dimension descriptor in descendant order (higher dimension first). Each 3-tuple has three elements: loop index variable, upper bound expression, and lower bound expression. The loop index variable identifies a loop in the reduction kernel loops. Since stride accesses are not allowed, the loop stride is always 1, so it is omitted from the tuple. The size of the k-dimensional tile is calculated from equation (5.1).

$$(U_k - L_k) \times ...(U_2 - L_2) \times (U_1 - L_1) \tag{5.1}$$

`reduction(operator : T`$[j_k, L_k, U_k]...[j_2, L_2, U_2][j_1, L_1, U_1]$`)`

`T`:	Tile name
k:	Dimension of the tile
j_i:	the loop index that is used in the traversal of the i^{th} dimension of the tile
L_i:	the lower bound of j_i
U_i:	the strict upper bound of j_i

(a) OpenMP API (C/C++) extension for the `reduction` clause

```
  int B[2][2] = {{0,0},{0,0}};
          ...
0 #pragma omp parallel for reduction(+: B[j,0,2][i,0,2])
1 for (k=0; k<10000000; k++)
2   for (j=0; j<2; j++)
3     for (i=0; i<2; i++)
4       B[j][i] += A[k][j][i]
```

(b) Simple example using the extended API

Figure 5.6: OpenMP API (C/C++) extension and a simple example code

The information stored in the tile descriptor is very important for the OpenMP compiler to generate correct parallel code.

The `operator`, as usual, must be a mathematically associative and commutative operator that performs the recursive calculation. In our current example, it is a "+".

```
0 #pragma omp parallel for reduction(+: A[j,0,2][i,0,2])
1 for (k=1; k<10000000; k++)
2     for (j=0; j<2; j++)
3         for (i=0; i<2; i++)
4             A[0][j][i] += A[k][j][i]
5
```

Figure 5.7: Tile reduction: tile is part of a bigger multi-dimensional array

The reduction tile is not required to be a standalone multi-dimensional array. Instead, it can be part of another larger multi-dimensional array. For example, in the code

in Figure 5.7, the reduction tile is `A[0][j][i]` ($j = \{0, 1\}, i = \{0, 1\}$). It is a 2×2 slice cut out from the 3-dimensional array `A[][][]`;

Besides, as we have mentioned before, the lower and upper bounds in the dimension descriptor are expressions. They are not required to be constants. Generally, the lower and upper bounds can be a function of other variables, as long as the result of the function can be decided at runtime. Figure 5.8 shows such an example. The code in Figure 5.8 is a blocked matrix multiplication program. It is easy to see that there is an opportunity to apply tile reduction on the loop in line 3, i.e., the `kk` loop. The diagram on the right hand side gives an intuitive illustration. In this example, the reduction tiles are blocks cut out from a big 2×2 matrix (`C[][]`). Therefore, the lower and upper bounds of the reduction tiles are not fixed values. In addition, the matrix `C[][]` might not be able to be evenly blocked. So, the tiles located at the margin of the matrix are usually smaller than the tiles located inside of the matrix. Thus, the sizes of the reduction tiles are not necessarily the same. All these information is reflected in the lower and upper bound expressions (or functions) in the dimension descriptor. Moreover, there is a restriction for the lower bound and upper bound expressions. They should not be functions of any index variable in the reduction kernel loops, i.e., they are orthogonal. This is to make sure that the shape of the reduction tile is a rectangle, or high-dimensional rectangle.

An interesting observation of this example code is that the number of the reduction kernel loops (which is 3, from line 6 to line 8) is not the same as the dimension of the reduction tile (which is 2). Generally, we do not require the number of the reduction kernel loops to be the same as the dimension of the reduction tile. We only require that the operations performed by the code in the reduction kernel loops can be viewed as one associative and commutative *macro* operation performed on the entire reduction tile.

5.3.2 Code Generation

Since tile reduction is derived from scalar reduction, its code generation shares the same framework as scalar reduction. Thus, we illustrate the code generation for tile

```
 0
 1 for (ii=0; ii<n; ii+=b)
 2     for (jj=0; jj<n; jj+=b)
 3 #pragma parallel for reduction(+: \
 4              C[i,ii,min(ii+b,n)][j,jj,min(jj+b,n)])
 5         for (kk=0; kk<n; kk+=b)
 6             for (i=ii; i<min(ii+b,n); i++)
 7                 for (j=jj; j<min(jj+b,n); j++)
 8                     for (k=kk; k<min(kk+b,n); k++)
 9                         C[i][j]+=A[i][k]*B[k][j];
10
```

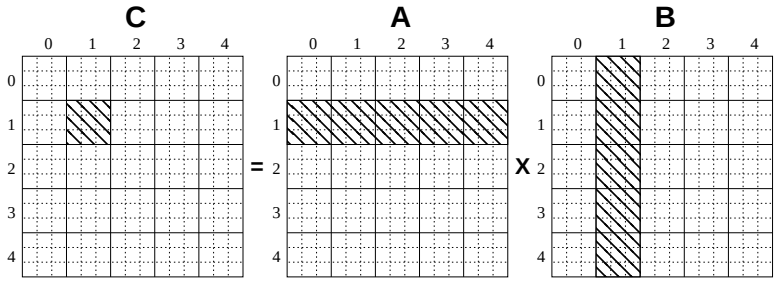


Figure 5.8: Tile reduction: upper and lower bounds are functions

reduction under the same framework as scalar reduction and use the code generation for scalar reduction as a reference. Generally, the code generation needs to deal with the following problems:

1. Distribute the iterations of the parallelized loop among the threads;
2. Allocate memory for the private copy of the tile used in the local recursive calculation;
3. Perform the local recursive calculation which is specified by the reduction kernel loops;
4. Update the global copy of the reduction tile;

Figure 5.9 shows the code generated for the tile reduction example in Figure 5.8. To make the thesis easy to follow, we present the pseudo C code in the figure.

As we have mentioned at the beginning of Section 5.3.1, we try to avoid complicating the code generation when we were developing the extension for the `reduction` clause. A good example is the code generation for distributing the iterations of the parallelized loop among the dynamic threads. Actually, this part of the code generation for tile reduction is the same as that for scalar reduction.

In the tile reduction program, the reduction kernel loops can be viewed as a single statement that performs the recursive calculation, which is the same as its counterpart in the scalar reduction program. So, from the angle of iteration distribution, the scalar reduction code and the tile reduction code are logically the same. Therefore, the method used to generate iteration distribution code for scalar reduction can also be used to generate iteration distribution code for tile reduction. It doesn't matter which `schedule` policy (`static`, `dynamic`, `guided`, or `runtime`) is deployed.

In Figure 5.9, we use `static` scheduling policy as an example. In the code from line 2 to line 7, the iterations of the `kk` loop (line 5 in Figure 5.8) are evenly distributed among the threads. The iterations of the loop are divided into chunks and each chunk is assigned to one dynamic thread. The iteration chunk assigned to the thread is delimited by the lower bound variable `"lb"` and the upper bound variable `"ub"`, which are determined by the *thread number* of the owner thread. This piece of code only deals with the parallelized loop and the user specified OpenMP parameters. It does not even need to look into the code of the reduction kernel loops. This is the same for other schedule policies.

At line 10, the OpenMP runtime routine allocates memory for the the private tile (`private_tile`), which is a 2-dimensional array. This private tile is used by the thread as a temporary storage to perform the local sequential tile reduction. Its size is calculated from the parameters specified in the dimension descriptor (see equation 5.1). Its element data type is inferred from the tile name. All this information is obtained from the extended `reduction` clause.

```
 0
 1  /* statically partition the iteration space among
     * the threads */
 2  num_thr = __builtin_omp_get_num_threads ();
 3  thr_id = __builtin_omp_get_thread_num ();
 4  chunk_size = (((n+(b-1))/(b-1))%num_thr) == 0 ? \
 5   (((n+(b-1))/(b-1))/num_thr):(((n+(b-1))/(b-1))/num_thr)+1;
 6  lb = chunk_size * thr_id;       /* lower bound */
 7  ub = min((lb+chunk_size),n);    /* upper bound */
 8
 9  /* allocate memory for private tile */
10  private_tile = (int *)__builtin_omp_memory_alloc( \
11             (min(ii+b,n)-ii)*(min(jj+b,n)-jj)*sizeof(int));
12
13  /* local tile reduction: serial */
14  for (kk=lb; kk<ub; kk+=b)
15   for (i=ii; i<min(ii+b,n), i++)
16     for (j=jj; j<min(jj+b,n), j++)
17       for (k=kk; k<min(kk+b,n), kk++)
18         private_tile[i-ii][j-jj] += A[i][k]*B[k][j]
19
20  /* update the global reduction tile */
21  __builtin_omp_atomic_start ();
22  for (i=ii; i<min(ii+b,n), i++)
23    for (j=jj; j<min(jj+b,n), j++)
24      C[i][j] += private_tile[i-ii][j-jj];
25  __builtin_omp_atomic_end ();
26
27  free(private_tile);
28
```

Figure 5.9: Pseudo code generated for the matrix multiplication example to perform tile reduction

The local sequential tile reduction is performed by the code from line 14 to line 18. This piece of code is almost the same copy as the original sequential program (line 5 to line 9 in Figure 5.8) except two places. At line 14, the lower and upper bounds of the loop are changed to `"lb"` and `"ub"`. This is to restrict the range of the iteration space in the chunk assigned to the current thread. Besides, at line 18, we replace the original reduction tile with the private tile and update its indices. This index calibration is required because the global reduction tile is cut out from a bigger multi-dimensional array, while the private tile is a standalone array. This piece of code performs local tile reduction sequentially, as in the original un-parallelized code.

After finishing the local tile reduction, the thread must update the global reduction tile. The code is shown from line 21 to line 25. The runtime routines called at line 21 & line 25 ensure atomic access to the global reduction tile. The loops at line 22 and line 23 are extracted from the *reduction kernel loops*. Only the loops listed in the *tile descriptor* are selected. So, the loop `k` in the reduction kernel loops is not included. The *lhs* variable of the statement at line 24 is the same variable as in the original code (line 9 in Figure 5.8). However, the *rhs* variable has been replaced with the private tile, in which the indices have been updated.

From the code in Figure 5.9, it is easy to see that the code generation for the tile reduction is as easy as that for the traditional scalar reduction. Meanwhile, no extra runtime supports is required. These advantages make the implementation of tile reduction in the OpenMP compiler very easy. In the next section, we will present the experimental results of applying the tile reduction on several typical benchmarks.

5.4 Experiments

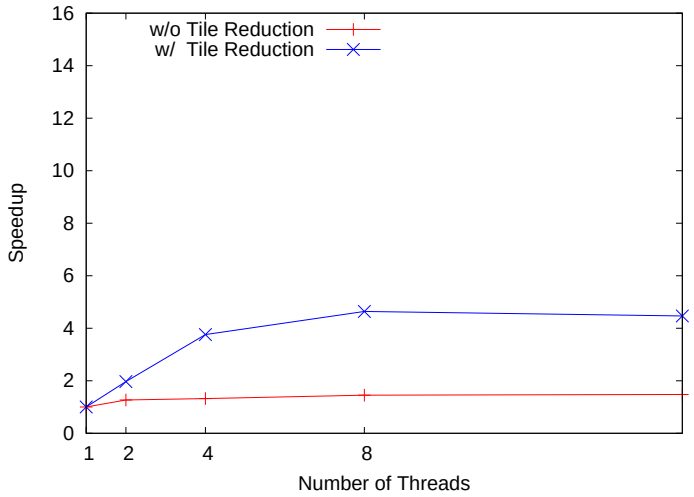
We have applied tile reduction on three benchmarks: the 2D histogram reduction, matrix-matrix multiplication and matrix-vector multiplication. The required code generation was implemented through source-to-source transformation and was prototyped in the Omni-1.6 OpenMP compiler [83]. The machine used in the experiments has 4 Intel

Dual-Core Xeon (Paxville) chips, which are clocked at 3.0 GHz. Each core has Hyper-Threading (HT) enabled. Therefore, the machine can be viewed as a 16-processor shared memory parallel computer. Each chip has 4MB L2 cache (2MB each core) and each core has 16KB L1 cache. The machine runs Linux Ubuntu 7.04.

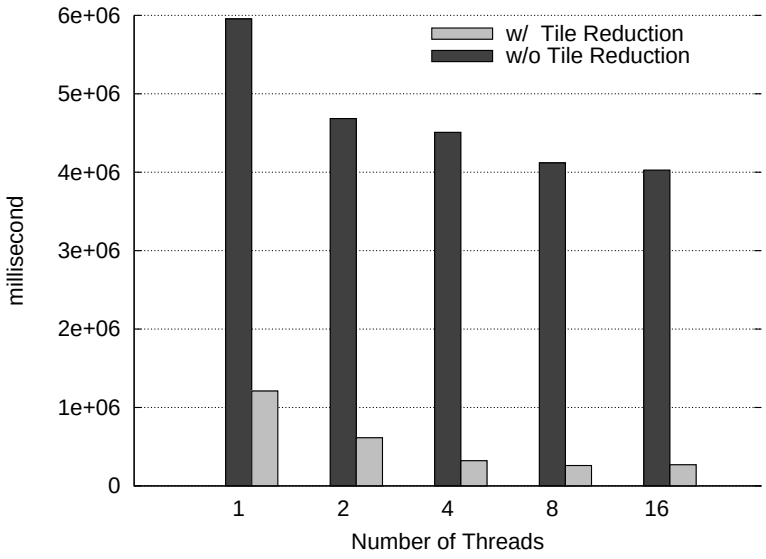
Figure 5.10, Figure 5.11, and Figure 5.12 show the experimental data of the each three benchmarks. The curve graphs in these figures display the speedup of the benchmark programs parallelized either through the tile reduction clause (w/ tile reduction) or through the standard OpenMP directives/clauses (w/o tile reduction). The bar charts, on the other hand, demonstrate the difference of the absolute execution time between the corresponding programs (w/ and w/o tile reduction) of the same set of benchmarks.

Figure 5.10(b) shows great performance enhancement if we parallelize the 2D histogram reduction benchmark with the tile reduction clause. Generally, compared with the program parallelized with standard OpenMP pragma, the absolute execution time of the tile reduction version decreased about 90% and its speedup on 8 threads increased from 1.5 to 4.5. The performance gain comes from the improved data locality, which owes to the tile reduction optimization. Without using tile reduction, the 2D histogram reduction program exhibit very poor scalability (shown in Figure 5.3). The tile reduction parallelization successfully rectifies the data access pattern and thus significantly improves its scalability. However, no matter what kind of optimizations are used, this benchmark stops scaling beyond 8 threads. This is because of the huge number of memory references in the code, which results in that its performance is finally restricted by the bandwidth of the shared memory bus.

The same phenomena are also observed in the matrix-matrix multiplication benchmark (see Figure 5.11(a) and 5.12(b)). Tile reduction can also decrease its execution time and improve its scalability. However, the magnitude of the performance enhancement caused by tile reduction is not as big as that of the 2D histogram reduction benchmark.

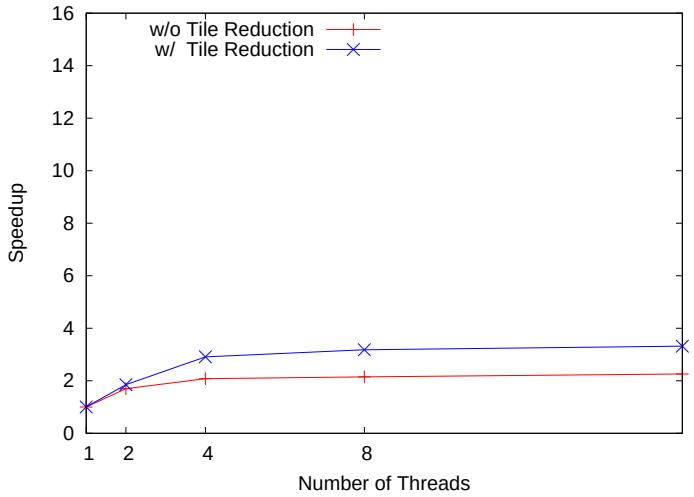


(a) speedup

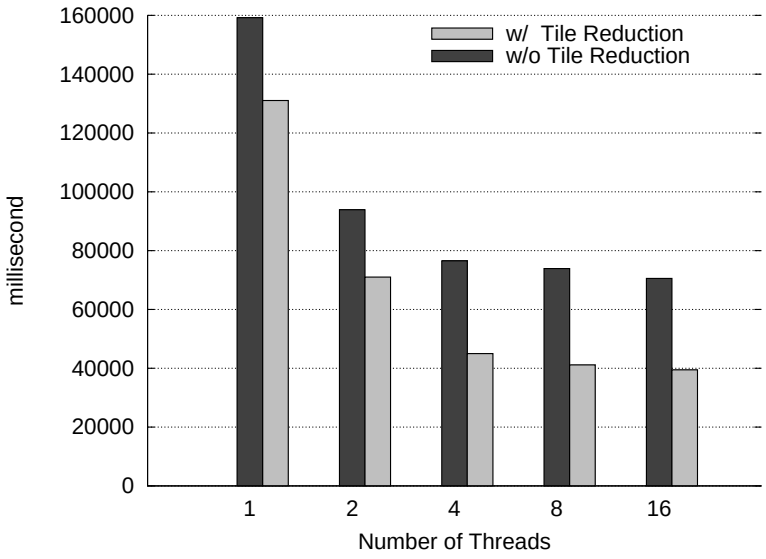


(b) execution time

Figure 5.10: 2D histogram reduction: Comparison of the speedup and execution time between the code parallelized with tile reduction and the code parallelized with the standard OpenMP pragma.

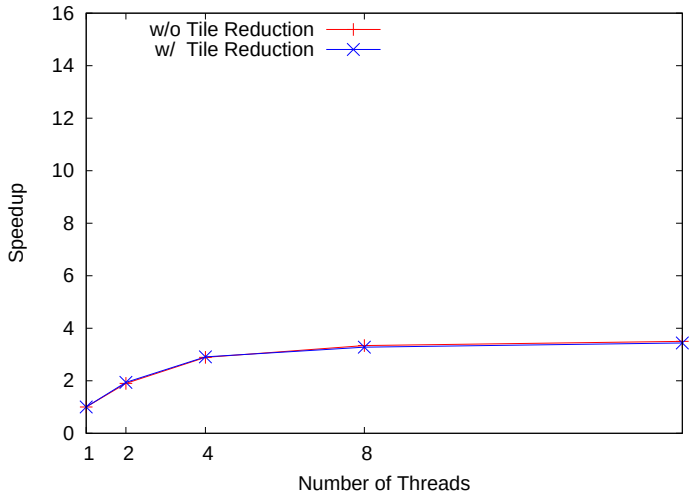


(a) speedup

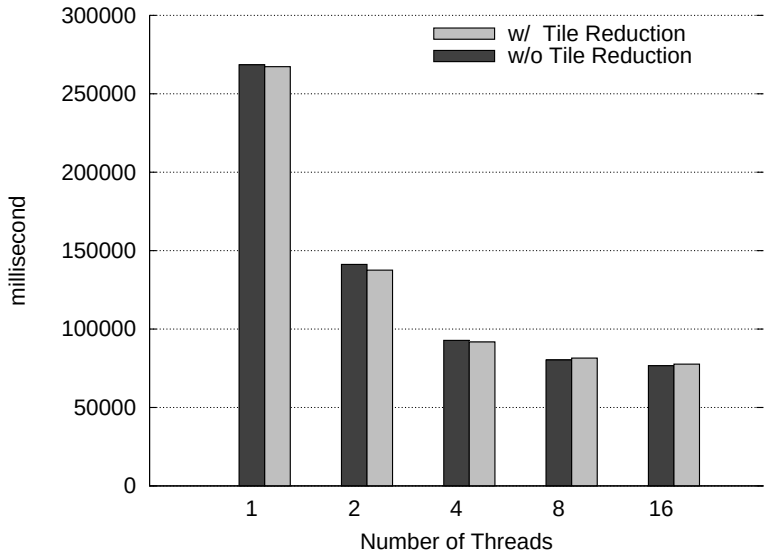


(b) execution time

Figure 5.11: Matrix-matrix multiplication: Comparison of the speedup and execution time between the code parallelized with tile reduction and the code parallelized with the standard OpenMP pragma.



(a) speedup



(b) execution time

Figure 5.12: Matrix-vector multiplication: Comparison of the speedup and execution time between the code parallelized with tile reduction and the code parallelized with the standard OpenMP pragma.

This is also the same for the scalability enhancement. The reason is that the data locality of the tiled matrix-matrix multiplication program is better than the 2D histogram reduction benchmark. Therefore, the performance gain from tile reduction in the matrix multiplication program is less than that in the 2D histogram reduction program. On average, the execution time decreased 34% after applying tile reduction and its speedup increased from 2.15 to 3.18 on 8 threads and from 2.26 to 3.32 on 16 threads.

For the matrix-vector multiplication case, the performance enhancement brought about by tile reduction is smaller than that of the previous two benchmarks. The reason is the same as the previous one. Moreover, compared with the other two benchmarks, there are less data memory references in this benchmark. So, the program's performance degrades a little bit when it runs with 8 or 16 threads. This is because of the synchronization overhead caused by the code in line 21 and 25 in Figure 5.9. In average, its execution time decreased 0.28%.

5.5 Summary

In this chapter, we introduced the concept of tile aware parallelization for OpenMP. Meanwhile, we developed the first tile aware parallelization technique - tile reduction, and illustrated the details of code generation for the tile reduction clause. We also designed a series of experiments to evaluate the tile reduction technique. From the experimental results and our experience of parallelizing the benchmarks, we have the following conclusions:

1. As a building block of the tile aware parallelization theory, tile reduction brings more opportunities to parallelize dense matrix applications.

2. For some benchmarks, tile aware parallelization is a more natural and intuitive way to reason about the best parallelization decision.

3. Tile reduction not only can improve data locality for some programs, but also can expose more parallelism.

5.6 Related Works of Parallel Reduction

Parallel reduction operations are supported in many parallel programming languages. They include C**[132], SAC [133], ZPL [131], UPC [128], and MPI [129]. Most of them support user-defined reduction operations, either through language constructs or through library routines. User-defined reduction operation provides a flexible way to implement tile reduction. However, programmers need to change both data structures and algorithms, which, sometimes, is not a trivial job.

Another piece of work that we need to mention is [134]. In [134], the authors propose to extend the OpenMP `reduction` clause to parallelize C++ generic algorithms. They propose to support user-defined types, overloaded operators, and function objects in the same way as the built-ins supported in the current OpenMP `reduction` clause. Their work is very close to that presented in this chapter. However, we study the reduction problem from a different angle. We propose tile reduction as one of the tile aware parallelizing technique for OpenMP, while [134] proposes user-defined reduction operation to complete their OpenMP extensions for parallelizing generic libraries. In our tile aware parallelization technique, we are concerned with the data partition, locality and a more flexible and efficient way to parallelize dense matrix programs written in canonical C syntax, while the purpose of [134] is to allow people to parallelize programs written in modern C++ idioms such as *iterators* and *function objects*, which are not canonical C syntax. Second, due to the non-trivial dynamic overhead of the generic techniques, generic libraries are not widely used in programming high performance scientific and engineering algorithms. Finally, there are no experimental data in [134].

Chapter 6

CONCLUSIONS AND FUTURE DIRECTIONS

In this thesis, we have proposed a set of *Tile Aware Parallelization* (TAP for short) techniques for OpenMP programs running on many-core processors with software managed memory hierarchies, like the IBM Cyclops-64 processor. The purpose of TAP is to grant OpenMP programmers the ability to interact with OpenMP compiler to orchestrate data movement in a parallel program running in the segmented memory space, thus the program can take full advantage of the fast on-chip memory. Although, for Cyclops-64, there are three different kinds of memory segments in the same address space, TAP only focus on the interface between on-chip and off-chip memory. The reason is that, among all the memory segment separators, the interface between on-chip and off-chip memory is the most critical one due to the memory bandwidth issue that exists at this interface. In the following two sections, we will draw our conclusion and point out the possible future work directions.

6.1 Summary and Conclusions

In Chapter 3, we have designed and developed *Tile Percolation*, a TAP technique used to generate data movement code for OpenMP programs running on the Cyclops-64 many-core processor. To make sure the generated data movement code can be executed in parallel with the computation code, we developed another TAP technique (in Chapter 4) called *Thread-Level Decoupled Access/Execution*, i.e. the TL-DAE execution model

for executing OpenMP programs on Cyclops-64. In Chapter 5, we developed a TAP technique called *Tile Reduction* to perform parallel reduction operation on multi-dimensional arrays.

From our experience in developing these TAP techniques, we got the following conclusions:

1. As more and more many-core processors adopt software managed memory hierarchy design, many new problems would come out if we use the existing OpenMP APIs to parallelize a sequential program for these many-core processors. Our development of tile aware Parallelization techniques show that, it is not only necessary but also possible to solve some of these new problems by introducing new data tile directives/clauses into the current OpenMP programming model.

2. The tile percolation technique can protect the programmers from involving in the hassles of dealing with the heterogeneity of the memory address space. The TL-DAE execution model makes sure that the data movement code generated by tile percolation can be executed in parallel with computation code. Both techniques can make programming on Cyclops-64 easier and make program execution on Cyclops-64 more efficient.

3. The tile reduction technique grant OpenMP the power to perform parallel reduction on multi-dimensional arrays. Experimental results show that, this technique can both improve parallelism and optimize data locality.

6.2 Future Works

The followings are the possible future directions for the design and implementation of the Tile Aware Parallelization techniques.

1. At the current stage, the design of the tile aware parallelization techniques is restricted by its *ad hoc* implementation through a source-to-source transformation

approach. This affects the design of the tile aware parallelization API. For example, in the API for the TL-DAE programming model, there is a field called `TY` in the definition of tile descriptor `TILE_DESC`. It tells the compiler the type of the data tile element. Actually, in a decent compiler implementation, the compiler can infer the type of the data tile element from the intermediate representation of the guarded function call. Therefore, it is not necessary to ask the programmer to respecify it in the API. So, the API does not need to have this field, which would make the API simpler and more clean. There are other similar issues in the current design which can be optimized in a decent compiler implementation. Therefore, a final decision on the design of the API needs a comprehensive discussion to decide which fields need to be kept in the API.

2. The current TL-DAE programming and execution model only support three tasklets, i.e. the 3-step operation *copyin*, *compute*, and *copyout*. However, the design of the implementation allows it to be extended to support more complicated computation models. For instance, the stream processing [135] model. In order to achieve a more general and flexible framework to support versatile computation models, efforts must be exerted to improve in the following directions: **(a)** First, the programming interface must be extended to give programmers the power to annotate a task that may have more tasklet functions and may possess more complicated relationships. In addition, the programmer also need the utilities to specify how data are exchanged among these tasklets. The current OpenMP pragma/directives do not support these; **(b)** Second, the data structure `tldae_task{}` needs to be extended. The tasklet array `.tasklet_array[]` should have more entries to store more tasklet functions that might be specified by programmer. Accordingly, we need to add more argument descriptor arrays in the `tldae_task{}` to accompany the multiple numbers of tasklet functions specified by programmer; **(c)** Currently, the dependence between tasklets in TL-DAE execution model is enforced by their

orders in the `.tasklet_array[]`. This array is an one dimension structure, so it can only be used to enforce dependence relationship between two tasklets in a total order. If the execution of tasklet `A` is dependent on tasklet `B` and tasklet `C`, we must insert `A` and `B` in front of `C` in the same tasklet array. Therefore, we would serialize the execution of tasklet `B` and tasklet `C` although they can be executed in parallel. We need a scheme to express such a dependence relationship in a partial order data structure, which means that a DAG is needed instead of an array.

3. In the current TL-DAE model, the data movement tasks and computation tasks are created and executed in parallel. However, the current work does not intend to leverage the data tile reuse that might exist among the different tasks. This actually can also be solved under the TL-DAE framework. Efforts are needed to **(a)** first provide a method that programmer can annotate data tile re-usage; **(b)** second, compiler needs to interpret the annotations and generate the correct code and data structure; **(c)** third, create partial order dependence graph so the tasks are scheduled to reuse the computation results produced by the leading tasks.

4. As we mentioned, it is very desirable to extend the current OpenMP programming model to deal with the issues brought up by the software managed memory hierarchy design. So, a set of simple and uniform new OpenMP APIs is required to be added into OpenMP of the next generation. Therefore, more practical OpenMP benchmarks need to be studied to improve the current tile aware parallelization directives/clauses.

Appendix A

DIAGRAM OF THE CYCLOPS-64 SOFTWARE TESTBED

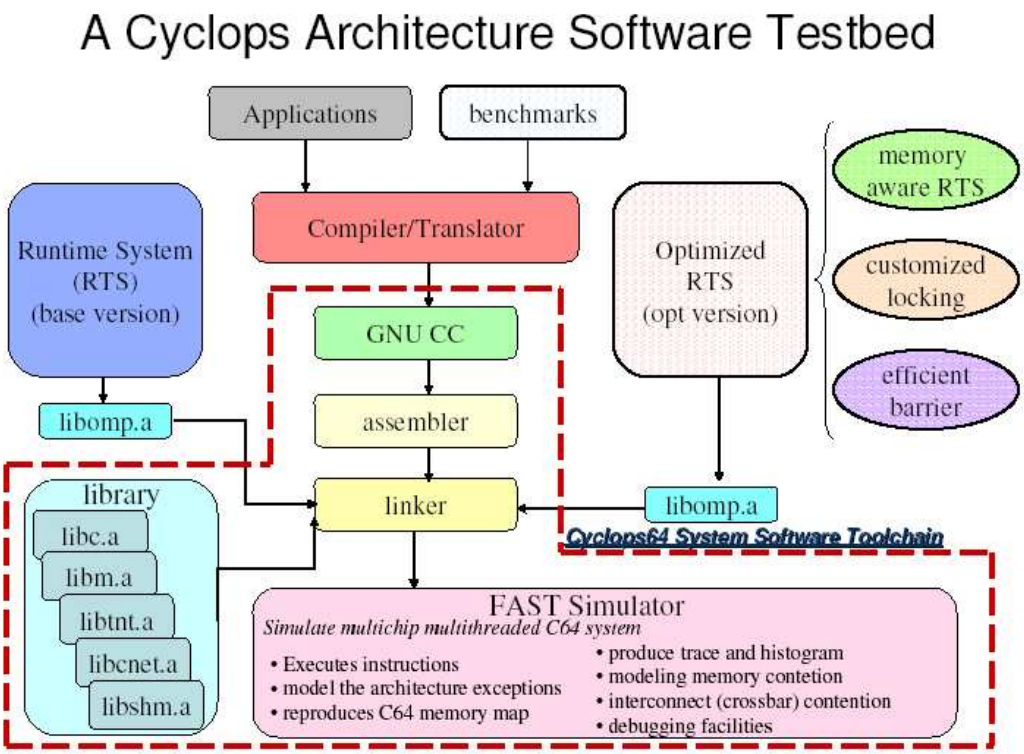


Figure A.1: Cyclops-64 Softeare Testbed (Courtesy to Ziang Hu)

Appendix B

IMPORTANT TL-DAE RUNTIME ROUTINES

```
10
11 void TLDAE_schedule_task(struct tldae_task *tsk);
12
13 void _tldae_read(struct tldae_task *tsk);
14
15 void _tldae_write(struct tldae_task *tsk);
16
17 void TLDAE_task_enqueue_read(struct tldae_task *tsk);
18
19 void TLDAE_task_enqueue_compute(struct tldae_task *tsk);
20
21 void TLDAE_task_enqueue_write(struct tldae_task *tsk);
22
23 struct tldae_task* TLDAE_get_task();
24
25 void TLDAE_put_task(struct tldae_task *tsk);
26
```

Figure B.1: TL-DAE Runtime Routines

Appendix C

ROSE COMPILER CODE GENERATION EXAMPLE

C.1 Original sparseLU code with OpenMP task pragma

```
156
157          ...
158
159 #pragma omp parallel single
160
161     for (kk=0; kk<NB; kk++) {
162         lu0(A[kk][kk]);
163 //#pragma omp taskgroup
164  {
165
166      for (jj=kk+1; jj<NB; jj++)
167          if (A[kk][jj] != NULL)
168 #pragma omp task firstprivate(kk, jj) shared(A)
169              fwd(A[kk][kk], A[kk][jj]);
170
171      for (ii=kk+1; ii<NB; ii++)
172          if (A[ii][kk] != NULL)
173 #pragma omp task firstprivate(kk, ii) shared(A)
174              bdiv (A[kk][kk], A[ii][kk]);
175  }
176
177 #pragma omp taskwait
178
179           ...
180
```

Figure C.1: Original OpenMP task code segment from sparseLU `main` function

C.2 Code Generated from ROSE Compiler with OpenMP Task Support

```
47 void OUT__4__1527__(void **__out_argv)
48 {
49   int *ii = (int *)(__out_argv[3]);
50   int *jj = (int *)(__out_argv[2]);
51   int *kk = (int *)(__out_argv[1]);
52   float *(*A)[100UL][100UL] = \
53           (float *(*)[100UL][100UL])(__out_argv[0]);
54   for ( *kk = 0;  *kk < 100; ( *kk)++) {
55     lu0((( *A)[ *kk])[ *kk]));
56 //#pragma omp taskgroup
57 {
58  for ( *jj = ( *kk + 1);  *jj < 100; ( *jj)++)
59    if ((( *A)[ *kk])[ *jj]) != ((float *)(((void *)0)))) {
60      void *__out_argv3__1527__[3];
61      __out_argv3__1527__[0] = ((void *)(&( *A)));
62      __out_argv3__1527__[1] = ((void *)(&( *kk)));
63      __out_argv3__1527__[2] = ((void *)(&( *jj)));
64      GOMP_task(OUT__3__1527__,&__out_argv3__1527__,0,12,4,1,0);
65    }
66  for ( *ii = ( *kk + 1);  *ii < 100; ( *ii)++)
67    if ((( *A)[ *ii])[ *kk]) != ((float *)(((void *)0)))) {
68      void *__out_argv2__1527__[3];
69      __out_argv2__1527__[0] = ((void *)(&( *A)));
70      __out_argv2__1527__[1] = ((void *)(&( *kk)));
71      __out_argv2__1527__[2] = ((void *)(&( *ii)));
72      GOMP_task(OUT__2__1527__,&__out_argv2__1527__,0,12,4,1,0);
73    }
74 }
75 GOMP_taskwait();
76
77 for ( *ii = ( *kk + 1);  *ii < 100; ( *ii)++)
78   if ((( *A)[ *ii])[ *kk]) != ((float *)(((void *)0))))
79     for ( *jj = ( *kk + 1);  *jj < 100; ( *jj)++)
80       if ((( *A)[ *kk])[ *jj]) != ((float *)(((void *)0)))) {
81         void *__out_argv1__1527__[4];
82         __out_argv1__1527__[0] = ((void *)(&( *A)));
83         __out_argv1__1527__[1] = ((void *)(&( *kk)));
84         __out_argv1__1527__[2] = ((void *)(&( *jj)));
85         __out_argv1__1527__[3] = ((void *)(&( *ii)));
86         GOMP_task(OUT__1__1527__,&__out_argv1__1527__,0,16,4,1,0);
87       }
88     GOMP_taskwait();
89   }
90 }
```

Figure C.2: Code Generated by ROSE Compiler with OpenMP Task Support: the master thread code

```
02 void OUT__1__1527__(void **__out_argv)
03 {
04   int *ii = (int *)(__out_argv[3]);
05   int *jj = (int *)(__out_argv[2]);
06   int *kk = (int *)(__out_argv[1]);
07   float *(*A)[100UL][100UL] = \
08           (float *(*)[100UL][100UL])(__out_argv[0]);
09   int _p_ii;
10   _p_ii =  *ii;
11   int _p_jj;
12   _p_jj =  *jj;
13   int _p_kk;
14   _p_kk =  *kk;
15   if ((( *A)[_p_ii])[_p_jj]) == ((float *)(((void *)0))))
16     (( *A)[_p_ii])[_p_jj] = allocate_clean_block();
17   bmod((( *A)[_p_ii])[_p_kk]),((( *A)[_p_kk])[_p_jj]), \
18        ((( *A)[_p_ii])[_p_jj]));
19 }
20
21 void OUT__2__1527__(void **__out_argv)
22 {
23   int *ii = (int *)(__out_argv[2]);
24   int *kk = (int *)(__out_argv[1]);
25   float *(*A)[100UL][100UL] = \
26           (float *(*)[100UL][100UL])(__out_argv[0]);
27   int _p_ii;
28   _p_ii =  *ii;
29   int _p_kk;
30   _p_kk =  *kk;
31   bdiv((( *A)[_p_kk])[_p_kk]),((( *A)[_p_ii])[_p_kk]));
32 }
33
34 void OUT__3__1527__(void **__out_argv)
35 {
36   int *jj = (int *)(__out_argv[2]);
37   int *kk = (int *)(__out_argv[1]);
38   float *(*A)[100UL][100UL] = \
39           (float *(*)[100UL][100UL])(__out_argv[0]);
40   int _p_jj;
41   _p_jj =  *jj;
42   int _p_kk;
43   _p_kk =  *kk;
44   fwd((( *A)[_p_kk])[_p_kk]),((( *A)[_p_kk])[_p_jj]));
45 }
```

Figure C.3: Code Generated by ROSE Compiler with OpenMP Task Support: three outlined wrapper functions of the task functions

BIBLIOGRAPHY

[1] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "Parallel tiled QR factorization for multicore architectures," LAPACK Working Note, Tech. Rep. 190, July 2007. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn190.pdf

[2] G. R. Gao, "Developing Program Execution Models for 1,000,000 cores and Beyond: Issues and Challenges," 2008, slides presented at Sandia.

[3] Intel, "Intel Core 2 Duo Processor," 2007. [Online]. Available: http://www.intel.com/products/processor/core2duo/index.htm

[4] AMD, "AMD Quad-Core Opteron Processors," 2007. [Online]. Available: http://multicore.amd.com/us-en/AMD-Multi-Core/Products/Multi-Core-S-WS.aspx

[5] OpenMP Architecture Review Board, "OpenMP Application Program Interface Version 3.0," May 2008, http://www.openmp.org/mp-documents/spec30.pdf.

[6] D. E. Culler, J. P. Singh, and A. Gupta, *Parallel Computer Architecture – A Hardware/Software Approach*. San Francisco: Morgan Kaufmann Publishers, Inc., 1999.

[7] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, "The landscape of parallel computing research: A view from berkeley," EECS Department, University of California, Berkeley, Technical Report UCB/EECS-2006-183, December 2006. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf

[8] F. Allen, "Compiling for performance a personal tour," San Diego, CA, 2007, turing Award Lecture given at PLDI 2007. [Online]. Available: http://awards.acm.org/images/awards/140/vstream/2006/turingaward2006.mov

[9] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy, "Introduction to the cell multiprocessor," *IBM J. Res. Dev.*, vol. 49, no. 4/5, pp. 589–604, 2005.

[10] H. P. Hofstee, "Power efficient processor architecture and the Cell processor." in *11th International Conference on High-Performance Computer Architecture (HPCA-11 2005)*, San Francisco, CA, USA, February 2005, pp. 258–262.

[11] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, "Towards a software infrastructure for cyclops-64 cellular architecture," in *HPCS 2006*, Labroda, Canada, June 2005.

[12] Y. Zhang, T. Jeong, F. Chen, H. Wu, R. Nitzsche, and G. R. Gao, "A study of the on-chip interconnection network for the ibm cyclops64 multi-core architecture," in *IPDPS'06: Proceedings of the 20th International Parallel and Distributed Processing Symposium, 25-29 April 2006, Rhodes Island, Greece*, April 2006.

[13] Z. Hu, J. del Cuvillo, W. Zhu, and G. R. Gao, "Optimization of dense matrix multiplication on ibm cyclops-64: Challenges and experiences," in *Euro-Par 2006, Parallel Processing, 12th International Euro-Par Conference, Dresden, Germany, August 28 - September 1, 2006, Proceedings*, 2006, pp. 134–144.

[14] D. M. Tullsen, S. J. Eggers, and H. M. Levy, "Simultaneous multithreading: Maximizing on-chip parallelism," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. Santa Margherita Ligure, Italy: ACM SIGARCH and IEEE Computer Society, June 22–24, 1995, pp. 392–403, *Computer Architecture News*, 23(2), May 1995.

[15] H. Akkary and M. A. Driscoll, "A dynamic multithreading processor," in *Proceedings of the 31st Annual International Symposium on Microarchitecture*. Dallas, Texas: IEEE-CS TC-MICRO and ACM SIGMICRO, November 30–December2, 1998, pp. 226–236.

[16] P. Marcuello, A. González, and J. Tubella, "Speculative multithreaded processors," in *Conference Proceedings of the 1998 International Conference on Supercomputing*. Melbourne, Australia: ACM SIGARCH, July 13–17, 1998, pp. 77–84.

[17] G. S. Sohi, S. E. Breach, and T. N. Vijaykumar, "Multiscalar processors," in *Proceedings of the 22nd Annual International Symposium on Computer Architecture*. Santa Margherita Ligure, Italy: ACM SIGARCH and IEEE Computer Society, June 22–24, 1995, pp. 414–425, *Computer Architecture News*, 23(2), May 1995.

[18] E. Rotenberg, Q. Jacobson, Y. Sazeides, and J. Smith, "Trace processors," in *Proceedings of the 30th Annual International Symposium on Microarchitecture*. Research Triangle Park, North Carolina: IEEE-CS TC-MICRO and ACM SIGMICRO, December 1–3, 1997, pp. 138–148.

[19] D. W. Wall, "Limits of instruction-level parallelism," in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages*

and Operating Systems. Santa Clara, California: ACM SIGARCH, SIGPLAN, SIGOPS, and the IEEE Computer Society, April 8–11, 1991, pp. 176–188, *Computer Architecture News*, 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices*, 26(4), April 1991.

[20] C. Ancourt and F. Irigoin, "Scanning polyhedra with DO loops," in *Proceedings of the Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Williamsburg, Virginia, April 21–24, 1991, pp. 39–50, *SIGPLAN Notices*, 26(7), July 1991.

[21] P. Feautrier, "Toward automatic partitioning of arrays on distributed memory computers," in *Conference Proceedings, 1993 International Conference on Supercomputing*. Tokyo: ACM, July 20–22, 1993, pp. 175–184.

[22] A. W. Lim and M. S. Lam, "Maximizing parallelism and minimizing synchronization with affine transforms," in *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, Paris, January 15–17, 1997, pp. 201–214.

[23] J. Xue, *Loop Tiling for Parallelism*. Kluwer Academic Publishers, 2000.

[24] John Paul Shen and Mikko H. Lipasti, *Modern Processor Design: Fundamentals of Superscalar Processors*. McGraw-Hill Book Company, 2005.

[25] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 4th ed. San Francisco: Morgan Kaufmann Publishers, Inc., 2006.

[26] T. Chen, Z. Sura, K. M. O'Brien, and J. K. O'Brien, "Optimizing the use of static buffers for dma on a cell chip," in *LCPC*, ser. Lecture Notes in Computer Science, G. Almási, C. Cascaval, and P. Wu, Eds., vol. 4382. Springer, 2006, pp. 314–329. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-72521-3_23

[27] Tao Liu and Haibo Lin and Tong Chen and Kevin O'Brien and Ling Shao, "DBDB: optimizing DMATransfer for the cell be architecture," in *Proceedings of the 23rd international conference on Supercomputing, ICS 2009, Yorktown Heights, NY, USA, June 8-12, 2009*. ACM, 2009, pp. 36–45.

[28] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, no. 4, pp. 289–308, 1984. [Online]. Available: http://doi.acm.org/10.1145/357401.357403

[29] J. E. Smith, S. Weiss, and N. Y. Pang, "A simulation study of decoupled architecture computers," *IEEE Trans. Comput.*, vol. 35, no. 8, pp. 692–702, 1986. [Online]. Available: http://dx.doi.org/10.1109/TC.1986.1676820

[30] G. Gan, X. Wang, J. Manzano, and G. R. Gao, "Tile Percolation: an OpenMP Tile Aware Parallelization Technique for the Cyclops-64 Multicore Processor," in *Euro-Par 2009, Parallel Processing, 15th International Euro-Par Conference, Delft, Netherlands, August 25 - August 28, 2009, Proceedings*, ser. Lecture Notes in Computer Science. Springer, 2009. [Online]. Available: http://dx.doi.org/10.1007/11823285_14

[31] G. Gan and J. Manzano, "TL-DAE: Thread-Level Decoupled Access/Execution for OpenMP on the Cyclops-64 Many-core Processor," in *Languages and Compilers for Parallel Computing, 22nd International Workshop, LCPC 2009, Newark, Delaware, US, October 8-10, 2009, Revised Selected Papers*, ser. Lecture Notes in Computer Science. Springer, 2009.

[32] G. Gan, X. Wang, J. Manzano, and G. R. Gao, "Tile Reduction: the First Step towards OpenMP Tile Aware Parallelization," in *Lecture Notes in Computer Science: OpenMP in a New Era of Parallelism, IWOMP'09, International Workshop on OpenMP*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2009.

[33] "International Technology Roadmap for Semiconductors," Oct 2007. [Online]. Available: http://www.itrs.net/Links/2007ITRS/Home2007.htm

[34] K. Krewell, "Best servers of 2004," *Microprocessor Report*, Jan 2005.

[35] P. P. Gelsinger, "Microprocessors for the new millennium: Challenges, opportunities, and new frontiers," in *Solid-State Circuits Conference, 2001. Digest of Technical Papers. ISSCC. 2001 IEEE International*. IEEE Computer Society Press, 2001, pp. 22–25.

[36] J. Li and J. F. Martinez, "Power-performance considerations of parallel computing on chip multiprocessors," *ACM Trans. Archit. Code Optim.*, vol. 2, no. 4, pp. 397–422, 2005. [Online]. Available: http://doi.acm.org/10.1145/1113841.1113844

[37] T. Agerwala and S. Chatterjee, "Computer architecture: Challenges and opportunities for the next decade," *IEEE Micro*, vol. 25, no. 3, pp. 58–69, 2005. [Online]. Available: http://dx.doi.org/10.1109/MM.2005.45

[38] D. M. Brooks, P. Bose, S. E. Schuster, H. Jacobson, P. N. Kudva, A. Buyuktosunoglu, J.-D. Wellman, V. Zyuban, M. Gupta, and P. W. Cook, "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," *IEEE Micro*, vol. 20, no. 6, pp. 26–44, 2000. [Online]. Available: http://dx.doi.org/10.1109/40.888701

[39] W. A. Wulf and S. A. McKee, “Hitting the memory wall: implications of the obvious,” *SIGARCH Comput. Archit. News*, vol. 23, no. 1, pp. 20–24, 1995. [Online]. Available: http://doi.acm.org/10.1145/216585.216588

[40] S. A. McKee, “Reflections on the memory wall,” in *CF ’04: Proceedings of the 1st conference on Computing frontiers*. New York, NY, USA: ACM Press, 2004, p. 162. [Online]. Available: http://doi.acm.org/10.1145/977091.977115

[41] C. Grassl, “Optimizing for performance on ibm power4 systems,” in *The 7th SCICOMP Meeting: IBM System Scientific User Group*, 2003.

[42] K. I. Farkas, P. Chow, N. P. Jouppi, and Z. Vranesic, “The multicluster architecture: reducing cycle time through partitioning,” in *MICRO 30: Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 1997, pp. 149–159.

[43] F. Li, C. Nicopoulos, T. Richardson, Y. Xie, V. Narayanan, and M. Kandemir, “Design and management of 3d chip multiprocessors using network-in-memory,” *SIGARCH Comput. Archit. News*, vol. 34, no. 2, pp. 130–141, 2006. [Online]. Available: http://doi.acm.org/10.1145/1150019.1136497

[44] N. Mitchell, L. Carter, J. Ferrante, and D. Tullsen, “ILP versus TLP on SMT,” in *Supercomputing ’99: Proceedings of the 1999 ACM/IEEE conference on Supercomputing (CDROM)*. New York, NY, USA: ACM Press, 1999, p. 37. [Online]. Available: http://doi.acm.org/10.1145/331532.331569

[45] P. Machanick, “How multithreading addresses the memory wall,” School of IT and Electrical Engineering, University of Queensland,” Technical Report, 2002.

[46] J. Clabes, J. Friedrich, M. Sweet, J. DiLullo, S. Chu, D. Plass, J. Dawson, P. Muench, L. Powell, M. Floyd, B. Sinharoy, M. Lee, M. Goulet, J. Wagoner, N. Schwartz, S. Runyon, G. Gorman, P. Restle, R. Kalla, J. McGill, and S. Dodson, “Design and implementation of the POWER5 microprocessor,” in *DAC ’04: Proceedings of the 41st annual conference on Design automation*. New York, NY, USA: ACM Press, 2004, pp. 670–672. [Online]. Available: http://doi.acm.org/10.1145/996566.996749

[47] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005. [Online]. Available: http://dx.doi.org/10.1109/MM.2005.35

[48] C. McNairy and R. Bhatia, “Montecito: A dual-core, dual-thread itanium processor,” *IEEE Micro*, vol. 25, no. 2, pp. 10–20, 2005. [Online]. Available: http://dx.doi.org/10.1109/MM.2005.34

[49] J. Held, J. Bautista, and S. Koehl, “From a few cores to many: A tera-scale computing research overview,” Intel White Paper, 2006.

[50] ClearSpeed Technology, “CSX processor architecture whitepaper,” 2006.

[51] “NVIDIA CUDA Revolutionary GPU Computing,” Oct 2007. [Online]. Available: http://developer.nvidia.com/object/cuda.html

[52] “ATI Radeon HD 2900 XT - Overview,” 2007. [Online]. Available: http://ati.amd.com/products/radeonhd2900/radeonhd2900xt/index.html

[53] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, L. Bao, J. Brown, M. Mattina, C.-C. Miao, C. Ramey, D. Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook, “Tile64 - processor: A 64-core soc with mesh interconnect,” in *Solid-State Circuits Conference, 2008. ISSCC 2008. Digest of Technical Papers. IEEE International*, Feb. 2008, pp. 88–598. [Online]. Available: http://dx.doi.org/10.1109/ISSCC.2008.4523070

[54] L. Hammond and K. Olukotun, “Considerations in the design of hydra: A multiprocessor-on-a-chip microarchitecture, Tech. Rep. CSL-TR-98-749, 1998. [Online]. Available: citeseer.ist.psu.edu/hammond98considerations.html

[55] K. Mai, T. Paaske, N. Jayasena, R. Ho, W. J. Dally, and M. Horowitz, “Smart memories: a modular reconfigurable architecture,” in *ISCA ’00: Proceedings of the 27th annual international symposium on Computer architecture*. New York, NY, USA: ACM Press, 2000, pp. 161–171. [Online]. Available: http://doi.acm.org/10.1145/339647.339673

[56] C. E. Kozyrakis and D. Patterson, “A new direction for computer architecture research,” *IEEE Computer*, vol. 30(9), 1997.

[57] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal, “The raw microprocessor: A computational fabric for software circuits and general-purpose programs,” *IEEE Micro*, vol. 22, no. 2, pp. 25–35, 2002. [Online]. Available: http://dx.doi.org/10.1109/MM.2002.997877

[58] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal, “Baring it all to software: Raw machines,” *IEEE Computer*, vol. 30(9), pp. 86–93, Sept. 1997.

[59] K. Sankaralingam, R. Nagarajan, R. McDonald, R. Desikan, S. Drolia, M. S. Govindan, P. Gratz, D. Gulati, H. Hanson, C. Kim, H. Liu, N. Ranganathan, S. Sethumadhavan, S. Sharif, P. Shivakumar, S. W. Keckler, and D. Burger, "Distributed Microarchitectural Protocols in the TRIPS Prototype Processor," in *MICRO 39: Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 480–491. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.19

[60] J. Oliver, R. Rao, P. Sultana, J. Crandall, E. Czernikowski, L. W. J. IV, D. Franklin, V. Akella, and F. T. Chong, "Synchroscalar: A multiple clock domain, power-aware, tile-based embedded processor," in *ISCA '04: Proceedings of the 31st annual international symposium on Computer architecture*. Washington, DC, USA: IEEE Computer Society, 2004, p. 150.

[61] G. R. Gao, "Programming and Compiling for TiNy Threads (TNT) – Experience with Cyclops-64 Architecture," Dec 2006.

[62] P. M. Kogge, "Past predictions, the present, and future trends," Oct 2006.

[63] L. Seiler, D. Carmean, E. Sprangle, T. Forsyth, M. Abrash, P. Dubey, S. Junkins, A. Lake, J. Sugerman, R. Cavin, R. Espasa, E. Grochowski, T. Juan, and P. Hanrahan, "Larrabee: a many-core x86 architecture for visual computing," *ACM Trans. Graph.*, vol. 27, no. 3, pp. 1–15, 2008. [Online]. Available: http://doi.acm.org/10.1145/1360612.1360617

[64] Khronos, "OpenCL - The open standard for parallel programming of heterogeneous systems," 2010. [Online]. Available: http://www.khronos.org/opencl/

[65] Microsoft, "Example of DirectCompute for Next Generation Game Effects," 2009. [Online]. Available: http://www.microsoft.com/showcase/en/us/details/6ef116dc-b1d9-41db-8a7b-db1932ff72a5

[66] T. El-Ghazawi, W. Carlson, T. Sterling, and K. Yelick, *UPC: Distributed Shared-Memory Programming*. Wiley-Interscience, 2003.

[67] A. E. Eichenberger, J. K. O'Brien, K. M. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao, M. K. Gschwind, R. Archambault, Y. Gao, and R. Koo, "Using advanced compiler technology to exploit the performance of the cell broadband enginetm architecture," *IBM Syst. J.*, vol. 45, no. 1, pp. 59–84, 2006.

[68] A. E. Eichenberger, K. O'Brien, K. O'Brien, P. Wu, T. Chen, P. H. Oden, D. A. Prener, J. C. Shepherd, B. So, Z. Sura, A. Wang, T. Zhang, P. Zhao,

and M. Gschwind, “Optimizing compiler for the cell processor,” in *PACT ’05: Proceedings of the 14th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 161–172. [Online]. Available: http://dx.doi.org/10.1109/PACT.2005.33

[69] M. Kandemir and A. Choudhary, “Compiler-directed scratch pad memory hierarchy design and management,” in *DAC ’02: Proceedings of the 39th annual Design Automation Conference*. New York, NY, USA: ACM, 2002, pp. 628–633. [Online]. Available: http://doi.acm.org/10.1145/513918.514077

[70] J. del Cuvillo, W. Zhu, Z. Hu, and G. R. Gao, “Fast: A functionally accurate simulation toolset for the cyclops-64 cellular architecture,” in *Workshop on Modeling, Benchmarking and Simulation (MoBS’05) of ISCA’05*, Madison, Wisconsin, June 2005.

[71] ——, “TiNy Threads: A thread virtual machine for the Cyclops64 cellular architecture,” in *Fifth Workshop on Massively Parallel Processing, in conjuction with 19th International Parallel and Distributed Processing Symposium (IPDPS 2005)*, Denver, Colorado, USA, April 2005, p. 265.

[72] J. del Cuvillo, W. Zhu, and G. Gao, “Landing openmp on cyclops-64: an efficient mapping of openmp to a many-core system-on-a-chip,” in *CF ’06: Proceedings of the 3rd conference on Computing frontiers*. New York, NY, USA: ACM, 2006, pp. 41–50. [Online]. Available: http://doi.acm.org/10.1145/1128022.1128030

[73] E. Anderson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. D. Croz, S. Hammerling, J. Demmel, C. Bischof, and D. Sorensen, “Lapack: a portable linear algebra library for high-performance computers,” in *Supercomputing ’90: Proceedings of the 1990 conference on Supercomputing*. Los Alamitos, CA, USA: IEEE Computer Society Press, 1990, pp. 2–11.

[74] E. Anderson and J. J. Dongarra, “Evaluating block algorithm variants in LAPACK,” LAPACK Working Note, Tech. Rep. 19, April 1990. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn19.pdf

[75] H. Ltaief, J. Kurzak, and J. Dongarra, “Parallel block hessenberg reduction using algorithms-by-tiles for multicore architectures revisited,” LAPACK Working Note, Tech. Rep. 208, August 2008. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn208.pdf

[76] M. S. Lam, E. E. Rothberg, and M. E. Wolf, “The cache performance and optimizations of blocked algorithms,” in *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. Santa Clara, California: ACM SIGARCH, SIGPLAN, SIGOPS, and

the IEEE Computer Society, April 8–11, 1991, pp. 63–74, *Computer Architecture News*, 19(2), April 1991; *Operating Systems Review*, 25, April 1991; *SIGPLAN Notices*, 26(4), April 1991.

[77] A. Buttari, J. Langou, J. Kurzak, and J. J. Dongarra, "A class of parallel tiled linear algebra algorithms for multicore architectures," LAPACK Working Note, Tech. Rep. 191, September 2007. [Online]. Available: http://www.netlib.org/lapack/lawnspdf/lawn191.pdf

[78] J. J. Dongarra, I. S. Duff, D. C. Sorensen, and H. A. van der Vorst, *Numerical Linear Algebra for High-Performance Computers*. Philadelphia: Society for Industrial and Applied Mathematics, 1998.

[79] T. Chen, T. Zhang, Z. Sura, and M. G. Tallada, "Prefetching irregular references for software cache on cell," in *CGO '08: Proceedings of the sixth annual IEEE/ACM international symposium on Code generation and optimization*. New York, NY, USA: ACM, 2008, pp. 155–164. [Online]. Available: http://doi.acm.org/10.1145/1356058.1356079

[80] T. Chen, H. Lin, and T. Zhang, "Orchestrating data transfer for the cell/b.e. processor," in *ICS '08: Proceedings of the 22nd annual international conference on Supercomputing*. New York, NY, USA: ACM, 2008, pp. 289–298. [Online]. Available: http://doi.acm.org/10.1145/1375527.1375570

[81] J. Lee, S. Seo, C. Kim, J. Kim, P. Chun, Z. Sura, J. Kim, and S. Han, "Comic: a coherent shared memory interface for cell be," in *PACT '08: Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. New York, NY, USA: ACM, 2008, pp. 303–314. [Online]. Available: http://doi.acm.org/10.1145/1454115.1454157

[82] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: using data parallelism to program gpus for general-purpose uses," in *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 2006, pp. 325–335. [Online]. Available: http://doi.acm.org/10.1145/1168857.1168898

[83] K. Kusano, S. Satoh, and M. Sato, "Performance evaluation of the omni openmp compiler," in *ISHPC '00: Proceedings of the Third International Symposium on High Performance Computing*. London, UK: Springer-Verlag, 2000, pp. 403–414.

[84] J. del Cuvillo, "Breaking away from the os shadow: a program execution model aware thread virtual machine for multicore architectures," Ph.D. dissertation, Newark, DE, USA, 2008, chair-Guang R. Gao.

[85] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "Automatic data movement and computation mapping for multi-level parallel architectures with explicitly managed memories," in *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345210

[86] I. E. Venetis and G. R. Gao, "Mapping the lu decomposition on a many-core architecture: challenges and solutions," in *CF '09: Proceedings of the 6th ACM conference on Computing frontiers*. New York, NY, USA: ACM, 2009, pp. 71–80. [Online]. Available: http://doi.acm.org/10.1145/1531743.1531756

[87] Michael Kistler and Michael Perrone and Fabrizio Petrini, "Cell multiprocessor communication network: Built for speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, 2006. [Online]. Available: http://dx.doi.org/10.1109/MM.2006.49

[88] Tong Chen and Haibo Lin and Tao Zhang, "Orchestrating data transfer for the cell/B.E. processor," in *Proceedings of the 22nd Annual International Conference on Supercomputing, ICS 2008, Island of Kos, Greece, June 7-12, 2008*. ACM, 2008, pp. 289–298.

[89] R. D. Blumofe and C. E. Leiserson, "Scheduling multithreaded computations by work stealing," in *Proceedings of the 35th Annual Symposium on Foundations of Computer Science*. Santa Fe, New Mexico: IEEE, November 20–22, 1994, pp. 356–368.

[90] The NANOS Group at Universitat Politécnica de Catalunya, "Barcelona OpenMP Task Suite," May 2009, http://nanos.ac.upc.edu/content/barcelona-openmp-task-suite.

[91] E. Ayguadé, N. Copty, A. Duran, J. Hoeflinger, Y. Lin, F. Massaioli, X. Teruel, P. Unnikrishnan, and G. Zhang, "The design of openmp tasks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 20, no. 3, pp. 404–418, 2009.

[92] Ge Gan, Xu Wang, Joseph Manzano, Guang R. Gao, "Tile percolation: an openmp tile aware parallelization technique for the cyclops-64 multicore processor," in *Euro-Par 2009, Parallel Processing, 15th International Euro-Par Conference, Delft, Netherlands, August 25 - August 28, 2009, Proceedings*, 2009. [Online]. Available: http://dx.doi.org/10.1007/11823285_14

[93] M. T. Kandemir, J. Ramanujam, M. J. Irwin, N. Vijaykrishnan, I. Kadayif, and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems," *IEEE Trans. on CAD of Integrated*

Circuits and Systems, vol. 23, no. 2, pp. 243–260, 2004. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/TCAD.2003.822123

[94] M. E. Wolf and M. S. Lam, "A loop transformation theory and an algorithm to maximize parallelism," *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, October 1991.

[95] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam, "Data and computation transformations for multiprocessors," in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, Santa Barbara, California, July 19–21, 1995, pp. 166–178, *SIGPLAN Notices*, 30(8), August 1995.

[96] S. S. Muchnick, *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[97] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran, "Cache-oblivious algorithms," in *FOCS '99: Proceedings of the 40th Annual Symposium on Foundations of Computer Science*. Washington, DC, USA: IEEE Computer Society, 1999, p. 285.

[98] L. Kurian, P. T. Hulina, and L. D. Coraor, "Memory latency effects in decoupled architectures with a single data memory module," in *Proceedings of the 19th Annual International Symposium on Computer Architecture*. Gold Coast, Australia: ACM SIGARCH and IEEE Computer Society, May 19–21, 1992, pp. 236–245, *Computer Architecture News*, 20(2), May 1992.

[99] M. Sung, R. Krashinsky, and K. Asanović, "Multithreading decoupled architectures for complexity-effective general purpose computing," *SIGARCH Comput. Archit. News*, vol. 29, no. 5, pp. 56–61, 2001.

[100] M. N. Dorozhevets and P. Wolcott, "The el'brus-3 and mars-m: recent advances in russian high-performance computing," *J. Supercomput.*, vol. 6, no. 1, pp. 5–48, 1992. [Online]. Available: http://dx.doi.org/10.1007/BF00128641

[101] M. N. Dorojevets and V. G. Oklobdzija, "Multithreaded decoupled architecture," *International Journal of High Speed Computing*, vol. 7, no. 3, pp. 465–480, 1995.

[102] J. G. Steffan, C. B. Colohan, A. Zhai, and T. C. Mowry, "A scalable approach to thread-level speculation," in *Proceedings of the 27th Annual International Symposium on Computer Architecture*. Vancouver, British Columbia: IEEE Computer Society and ACM SIGARCH, June 12–14, 2000, pp. 1–12, *Computer Architecture News*, 28(2), May 2000.

[103] J.-M. Parcerisa and A. González, "Multithreaded decoupled access/execute processors," Universitat Politécnica de Catalunya, Departament dArquitectura de Computadors, Technical Report UPC-DAC-1997-83, 1997.

[104] J.-M. Parcerisa and A. González, "The synergy of multithreading and access/execute decoupling," in *Proceedings of the Fifth International Symposium on High-Performance Computer Architecture*. Orlando, Florida: IEEE Computer Society, January 9–13, 1999, pp. 59–63.

[105] Paraskevas Evripidou, "D3-machine: a decoupled data-driven multithreaded architecture with variable resolution support," *Parallel Comput.*, vol. 27, no. 9, pp. 1196–1225, 2001.

[106] P. Evripidou and J.-L. Gaudiot, "The USC decoupled multilevel data-flow execution model," in *Advanced Topics in Data-Flow Computing*, J.-L. Gaudiot and L. Bic, Eds. Englewood Cliffs, New Jersey: Prentice-Hall, 1991, ch. 13, pp. 347–379, book contains papers presented at the First Workshop on Data-Flow Computing, held in conjunction with the 16th Annual International Symposium on Computer Architecture in Eilat, Israel, May 1989.

[107] C.-K. Luk, "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*. Göteborg, Sweden: IEEE Computer Society and ACM SIGARCH, June 30–July 4, 2001, pp. 40–51, *Computer Architecture News*, 29(2), May 2001.

[108] J. D. Collins, H. Wang, D. M. Tullsen, C. Hughes, Y.-F. Lee, D. Lavery, and J. P. Shen, "Speculative precomputation: long-range prefetching of delinquent loads," in *ISCA '01: Proceedings of the 28th annual international symposium on Computer architecture*. New York, NY, USA: ACM, 2001, pp. 14–25. [Online]. Available: http://doi.acm.org/10.1145/379240.379248

[109] C. Zilles and G. Sohi, "Execution-based prediction using speculative slices," in *Proceedings of the 28th Annual International Symposium on Computer Architecture*. Göteborg, Sweden: IEEE Computer Society and ACM SIGARCH, June 30–July 4, 2001, pp. 2–13, *Computer Architecture News*, 29(2), May 2001.

[110] T. C. Mowry, M. S. Lam, and A. Gupta, "Design and evaluation of a compiler algorithm for prefetching," in *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1992, pp. 62–73. [Online]. Available: http://doi.acm.org/10.1145/143365.143488

[111] D. Callahan, K. Kennedy, and A. Porterfield, “Software prefetching,” in *ASPLOS-IV: Proceedings of the fourth international conference on Architectural support for programming languages and operating systems*. New York, NY, USA: ACM, 1991, pp. 40–52. [Online]. Available: http://doi.acm.org/10.1145/106972.106979

[112] A. Jacquet, V. Janot, C. Leung, G. R. Gao, R. Govindarajan, and T. L. Sterling, “An executable analytical performance evaluation approach for early performance prediction,” in *17th International Parallel and Distributed Processing Symposium (IPDPS 2003), 22-26 April 2003, Nice, France*. IEEE Computer Society, 2003, pp. 268–276.

[113] R. Rangan, N. Vachharajani, M. Vachharajani, and D. I. August, “Decoupled software pipelining with the synchronization array,” in *PACT ’04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 177–188. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.14

[114] G. Ottoni, R. Rangan, A. Stoler, and D. I. August, “Automatic thread extraction with decoupled software pipelining,” in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 105–118. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2005.13

[115] J. M. Perez, P. Bellens, R. M. Badia, and J. Labarta, “Cellss: making it easier to program the cell broadband engine processor,” *IBM J. Res. Dev.*, vol. 51, no. 5, pp. 593–604, 2007.

[116] P. Bellens, J. M. Perez, R. M. Badia, and J. Labarta, “Cellss: a programming model for the cell be architecture,” in *SC ’06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. New York, NY, USA: ACM, 2006, p. 86. [Online]. Available: http://doi.acm.org/10.1145/1188455.1188546

[117] F. Irigoin and R. Triolet, “Supernode partitioning,” in *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*. San Diego, California: ACM SIGACT and SIGPLAN, January 13–15, 1988, pp. 319–329.

[118] J. M. Anderson and M. S. Lam, “Global optimizations for parallelism and locality on scalable parallel machines,” in *Proceedings of the ACM SIGPLAN ’93 Conference on Programming Language Design and Implementation*, Albuquerque, New Mexico, June 23–25, 1993, pp. 112–125, *SIGPLAN Notices*, 28(6), June 1993.

[119] A. W. Lim, S.-W. Liao, and M. S. Lam, “Blocking and array constraction across arbitrarily nested loops using affine partitioning,” in *PPoPP’01*, Snowbird, Utah, USA, June 2001.

[120] M. E. Wolf and M. S. Lam, “A data locality optimizing algorithm,” in *Proceedings of the ACM SIGPLAN ’91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 26–28, 1991, pp. 30–44, *SIGPLAN Notices*, 26(6), June 1991.

[121] A. W. Lim and M. S. Lam, “Communication-free parallelization via affine transformations,” in *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds., no. 892. Ithaca, New York: Springer-Verlag, August 8–10, 1994, pp. 92–106.

[122] High Performance Fortran Forum, “High-performance fortran language specification version 2.0,” Rice University,” Technical Report, 1997.

[123] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: an object-oriented approach to non-uniform cluster computing,” in *OOPSLA ’05: Proceedings of the 20th annual ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*. New York, NY, USA: ACM, 2005, pp. 519–538. [Online]. Available: http://doi.acm.org/10.1145/1094811.1094852

[124] S. J. Deitz, “High-level programming language abstractions for advanced and dynamic parallel computations,” Ph.D. dissertation, Seattle, WA, USA, 2005, chair-Lawrence Snyder.

[125] Y. Dotsenko, C. Coarfa, and J. Mellor-Crummey, “A multi-platform co-array fortran compiler,” in *PACT ’04: Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 29–40. [Online]. Available: http://dx.doi.org/10.1109/PACT.2004.3

[126] P. N. Hilfinger, D. Bonachea, D. Gay, S. Graham, B. Liblit, G. Pike, and K. Yelick, “Titanium language reference manual,” Berkeley, CA, USA, Tech. Rep., 2001.

[127] J. Guo, G. Bikshandi, B. B. Fraguela, M. J. Garzaran, and D. Padua, “Programming with tiles,” in *PPoPP ’08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2008, pp. 111–122. [Online]. Available: http://doi.acm.org/10.1145/1345206.1345225

[128] UPC Consortium, “UPC Collective Operations Specifications V1.0 A publication of the UPC Consortium,” 2003.

[129] M. P. I. Forum, “MPI: A message-passing interface standard (version 1.0),” Tech. Rep., May 1994, uRL http://www.mcs.anl.gov/mpi/mpi-report.ps.

[130] S. J. Deitz, B. L. Chamberlain, S.-E. Choi, and L. Snyder, “The design and implementation of a parallel array operator for the arbitrary remapping of data,” in *PPoPP ’03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. New York, NY, USA: ACM, 2003, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/781498.781526

[131] S. J. Deitz, B. L. Chamberlain, and L. Snyder, “High-level language support for user-defined reductions,” *J. Supercomput.*, vol. 23, no. 1, pp. 23–37, 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1015781018449

[132] G. Viswanathan and J. R. Larus, “User-defined reductions for efficient communication in data-parallel languages,” University of Wisconsin-Madison, Technical Report 1293, Jan 1996.

[133] S.-B. Scholz, “On defining application-specific high-level array operations by means of shape-invariant programming facilities,” in *APL ’98: Proceedings of the APL98 conference on Array processing language*. New York, NY, USA: ACM, 1998, pp. 32–38. [Online]. Available: http://doi.acm.org/10.1145/327559.327613

[134] P. Kambadur, D. Gregor, and A. Lumsdaine, “Openmp extensions for generic libraries,” in *Lecture Notes in Computer Science: OpenMP in a New Era of Parallelism, IWOMP’08, International Workshop on OpenMP*, vol. 5004/2008. Springer Berlin / Heidelberg, 2008, pp. 123–133.

[135] J. Gummaraju and M. Rosenblum, “Stream programming on general-purpose processors,” in *MICRO 38: Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 343–354. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2005.32