私立東海大學

資訊工程研究所

碩士論文

指導教授：楊朝棟 博士

多核心系統之 OpenMP 和 CUDA 平行程式效能比較

Performance Comparisons with OpenMP and CUDA Parallel Programming on Multicore Systems

研究生：張子杰

中華民國一〇〇年七月

東海大學碩士學位論文考試審定書

東海大學資訊工程學系 研究所

研究生 張 子 杰 所提之論文

多核心系統之OpenMP和CUDA平行程式效能比較

經本委員會審查，符合碩士學位論文標準。

學位考試委員會

召 集 人 楊武 簽章

委 員 [illegible]

[illegible]

時文中

指 導 教 授 楊朝棟 簽章

中華民國 100 年 6 月 27 日

摘要

當今多核心處理器已經佔據越來越多的市場份額，而且編譯人員也必須面對多核心處理器所帶來的衝擊。由於半導體運作溫度和功耗的問題限制了單核心微處理器效能增長。這個原因導致許多微處理器供應商轉向多核心晶片。不僅處理器走向隨之而來的多核心處理器趨勢，圖形處理單元(GPU)也是。同時，並行處理不僅只是一個機會，也是一個挑戰。程式設計師或編譯器將軟體平行化是在多核心晶片上提高效能的關鍵。在本論文中，我們介紹一些基於 OpenMP 的自動平行化工具，以減少我們重寫運行在多核心系統上之程式碼的時間，我們針對 ROSE 編譯器深入探討並實作一個介面以簡化使用之複雜度。其中有一些工具可以自動平行化成 CUDA 程式碼。另一部分，我們提出了一個混合 OpenMP、CUDA 及 MPI 編譯技術的平行編譯方法，在含有一個 C1060 和一個 S1070 的 GPU 叢集裡，根據 C1060 裝置數量分割迴圈。然後透過 OpenMP 或 MPI 的程序平行指派任務給 GPU 運算。最後，本論文中實驗有兩個部分，首先我們證明了這些自動平行化的工具可行性與正確性，並分別於一般的處理器、圖形處理單元和嵌入式系統上運行並討論其效能比較。另一部分的實驗中，也驗證了混合 OpenMP、CUDA 及 MPI 編譯技術的平行編譯方法確實提高效能。

關鍵字：自動平行、平行編譯、多核心、OpenMP、CUDA、MPI

Abstract

Nowadays, the multicore processor has occupied more and more market shares, and the programming personnel also must face the collision brought by the revolution of multicore processor. Because of the semiconductor operating temperature and power consumption limits performance growth for single-core microprocessors. This reason leads many microprocessor vendors to turn to multicore chip organizations. Not only CPU goes along the trend of multicore processors, but also GPU. At the same time, parallel processing is not only the opportunity but also a challenge. The programmer or compiler explicitly parallelize the software is the key for enhance the performance on multicore chip. In this thesis, we introduce some of the automatic parallel tools based OpenMP, which could to reduce our time on rewrite codes for parallel processing on multicore system. Then we focus on ROSE to explore in depth. And we implement an interface to simplify the complexity of use. And some of these tools can automatic parallelization for CUDA. In other hand, we propose a parallel programming approach using hybrid CUDA OpenMP, and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. Loop iterations assigned to MPI process and processed in parallel by CUDA run by the processor cores in the same computational node. Finally, there are two parts in our experiment in this thesis. First, we verified the available and correctness of the auto-parallel tools, and discussed the performance on CPU, GPU, and embedded system. And in the other part of experiment, we also verify that the hybrid programming could improve performance.

Keywords: Auto-Parallel, Parallel Programming, Multicore, OpenMP, CUDA, MPI

Acknowledgements

I would like to express my gratitude to all the people who have helped me through the completion of this thesis. In particular, I would like to thank my advisor, Professor Chao-Tung Yang, who introduced me to this topic and support in my work. Professor Yang gave me a deep influence and inspiration. I would also like to thank Professor Fang-Yie Leu, Professor Wuu Yang, Professor Yue-Shan Chang and Professor Wen-Chung Shih for their valuable comments and advice given while serving on my reading committee.

I am also grateful to many other classmates and members of my lab in THU. They gave me opportunities to gain more knowledge and shared their knowledge with me. Finally, I'm grateful to my family whose unconditional support and spur me to make this thesis.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivations

Today multicore has become the trend of enhance the processor’s performance, and most industries have considered multicore is the future of development. Not only PC's CPU and GPU have been grown to multicore, but also embedded system. We all think that multicore can give us higher performance than only one core, but it should be in the situation of parallel processing. If a task can be parallel processing by each core, then the efficiency should proportional with number of core. So in this condition, more and more cores must be better. In fact, even the trivial in our life or complex as science computing, parallel processing not only gives us more efficiency on computing but also changes the way we live.

Parallel processing has become more popular. Briefly speaking, parallel processing is the simultaneous use of multiple processor core or CPU to execute a program or more computing threads. Ideally, parallel processing lets the application run faster, because the application runs by more engines such as CPU or core.

GPUs are really “manycore” processors, with hundreds of processing elements. A graphics processing unit (GPU) is a specialized microprocessor that offloads and accelerates graphics rendering from the central (micro-) processor. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. We know that a CPU has only 8 cores at single chip currently, but a GPU has grown to 448 cores. From the number of cores, we know that GPU is appropriately to compute the programs which are suited massive parallel processing. Although the frequency of the core on the GPU is lower than CPU's, we believe that massively parallel can conquer the problem of lower frequency. By the way, GPU has been used on supercomputers. In top 500 sites for November 2010 [23], there are three supercomputers of the first 5 are built with NVIDIA GPU.

1.2 The Goal and Contributions

In this thesis, we introduce some of the automatic parallel tools for parallel processing on multicore system. These tools can automatically transform sequential C/C++ codes to parallel C/C++ codes or to generate parallel programs by using OpenMP directives or CUDA. Then we focus on ROSE to explore in depth. And we implement an interface to simplify the complexity of use. And we must to have an experiment on these tools, to let us know the available of these tools. And when we use them to enhance the performance, how much benefit we can get actually. Then, we used these tools to do parallel programming on some benchmarks, and compared the performance.

And we propose a solution to not only simplify the use of hardware acceleration in conventional general purpose applications, but also to keep the application code portable. We propose a parallel programming approach using hybrid CUDA, OpenMP and MPI [1] [2] [3] programming, which partition loop iterations according to the performance weighting of multicore [4] nodes in a cluster. Because iterations assigned to one MPI process are processed in parallel by OpenMP threads run by the processor cores in the same computational node, the number of loop iterations allocated to one computational node at each scheduling step depends on the number of processor cores in that node.

1.3 Thesis Organization

The rest parts of this work are organized as follows. Chapter 2 describes a background review of parallel programming and auto-parallel tools. Chapter 3 introduces the hardware system used in this work. Chapter 4 depicts the algorithm of automatic parallelization and hybrid the parallel programming of OpenMP, MPI, and CUDA. Chapter 5 presents experimental results of auto-parallel tools and show that the proposed approach improved performance over all previous schemes in heterogeneous and homogeneous clusters environments. Finally, conclusions are discussed in Chapter 6.

Chapter 2

Background Review

2.1 Parallel Programming

Nowadays, we can see that the maximum of CPU’s core is only eight cores, but the GPU has grown to 448 cores. Form the number of cores, we know that GPU is appropriately to compute the program which is suited for massive parallel processing, despite his relatively low frequency of core, but massively parallel can conquer the problem. Now we know that GPU is suitable for parallel computing, how we can use GPU to help us to compute. There are several well-known methods, like CTM, OpenCL, and CUDA.

And in the aspect of CPU, there are several well-known APIs/technologies that can help us to parallel our code like MPI, OpenMP, TBB, Pthread, and OpenCL etc. We will introduce them one by one.

2.1.1 CTM

ATI (now AMD) developed a low-level programming interface called CTM [25], and its goal is enabling GPGPU computing. Developers can access to the native instruction set and memory of the parallel computational units which are in modern AMD video cards by using CTM programming. But CTM is just the name of a beta version, and the first production version of AMD's GPGPU technology is now called Stream SDK.

2.1.2 OpenCL

OpenCL [26] is a framework of programming on heterogeneous platforms which may consist of CPUs, GPUs, and other processors. OpenCL is developed by Apple Inc. and Khronos Group. OpenCL is consisting of a language and APIs. The language based on C99 is used to write kernels; and the APIs are used to define and control the platforms. OpenCL uses task-based and data-based parallelism to parallel computing. AMD/ATI and Nvidia have adopted OpenCL into graphics card drivers.

2.1.3 CUDA

CUDA (Compute Unified Device Architecture) [27][28] is architecture of parallel computing developed by NVIDIA. The architecture is consisting of three parts, library, runtime, and CUDA driver. Developers can access the virtual instruction set and memory of the parallel computational units which are in CUDA GPUs by using CUDA programming. Through the technology of CUDA, users can calculate by using the GPUs from the GeForce 8 series onwards, including Quadro and the Tesla. CUDA architecture is compatible with OpenCL. Neither in CUDA C-language or in OpenCL, the instructions will be transform into the codes of PTX by the driver, and then calculate by the graphics core. CUDA's parallel programming model maintains a low learning curve for programmers familiar with standard programming languages such as C. And current release is CUDA Toolkit 4.0 that supports various operating systems, including Microsoft Windows, Linux, and Mac OS X.

CUDA processing flow is described as **Figure 2-1**[27]. The first step: to copy the data which are on the main memory of CPU to the memory of GPU, second: to instruct the process to GPU by CPU, third: to parallel execute in each core on GPU, finally: to copy the result from the memory of GPU to the main memory of CPU.

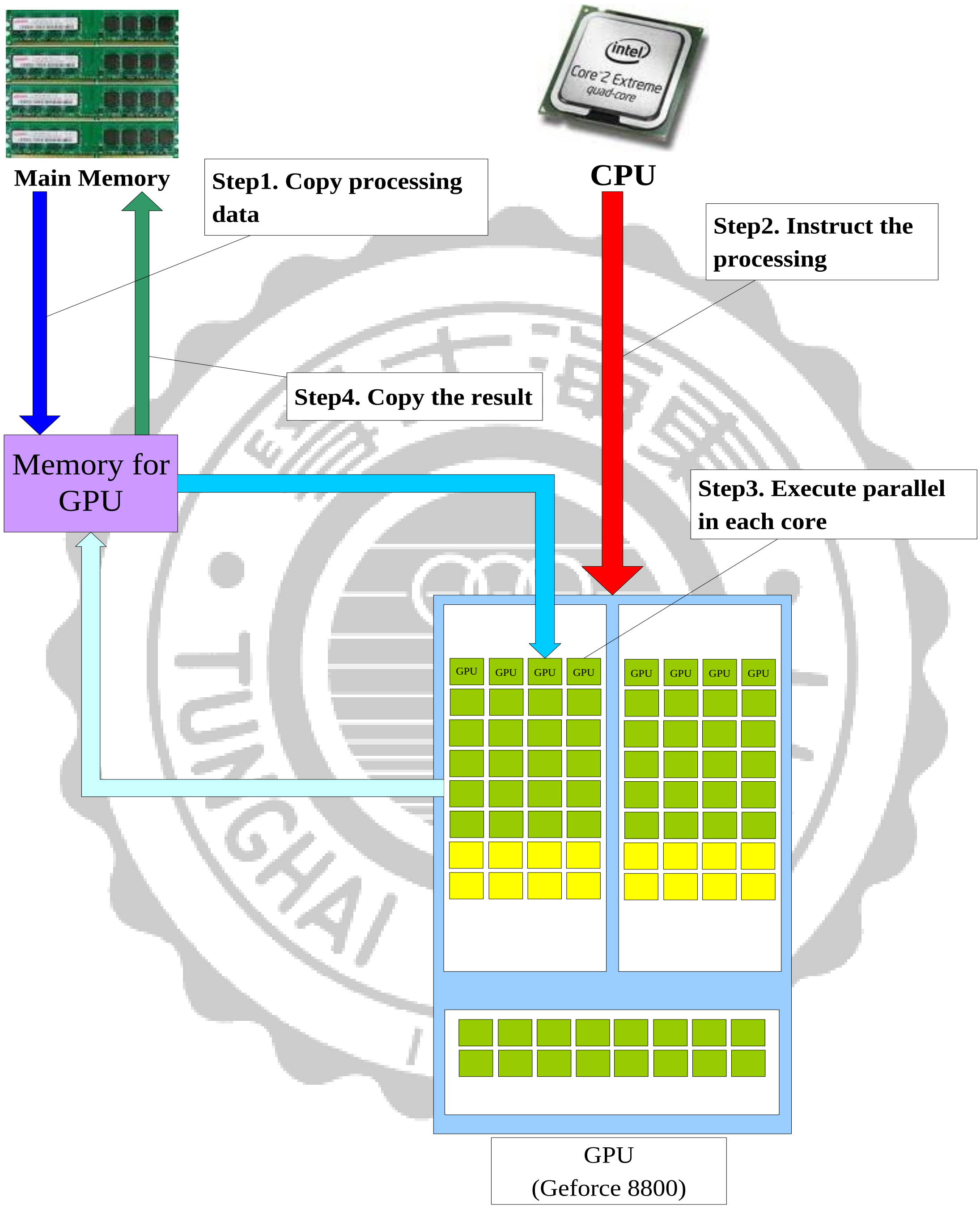


Figure 2-1. Processing flow on CUDA from Wiki

2.1.4 MPI

Message Passing Interface (MPI) [29] is a specification for message passing operations. It defines each worker as a process. MPI is currently the de-facto standard for developing HPC applications on distributed memory architecture. It provides language bindings for C, C++, and FORTRAN. The cluster computations exploit message-passing, because computers in cluster have distributed memory. When one process needs data from another one then you should manage data passing over the network. It is time-consuming operation. The MPI library is often used for parallel programming [5] in cluster systems because it is a message-passing programming language. MPICH is Free Software and is available for most flavors of UNIX (including Linux and Mac OS X) [30] and Microsoft Windows. Moreover, MPICH [31] is a developed program library.

2.1.5 OpenMP

Open Multi-Processing (OpenMP) [32] is an application programming interface (API), a kind of shared memory architecture API which provides a multithreaded capacity. OpenMP supports multi-platform shared memory multiprocessing programming in C, C++ and FORTRAN on much architecture which including UNIX and Microsoft Windows platforms. It consists of a set of compiler directives, library routines, and environment variables that influence run-time behavior. We can parallel a loop easily by invoking subroutine calls from OpenMP thread libraries and inserting the OpenMP compiler directives. In this way, the threads can obtain new tasks, the un-processed loop iterations, directly from local shared memory.

OpenMP is an open specification for shared memory parallelism. The basic idea behind OpenMP is data-shared parallel execution. OpenMP is portable across the shared memory architecture. Thread means that the unit of workers in OpenMP. Every thread can access a variable in shared cache or RAM. It works well, when accessing shared data costs you nothing. But OpenMP is not suitable for the needs of complex inter-thread synchronization and mutual exclusion. And OpenMP cannot be used on the non-shared memory systems (such as computer clusters). In such systems, MPI use more.

2.1.6 Pthread

POSIX Threads [33] is the POSIX standard of threads, and it defines an API for creating and controlling threads and defines a set of C programming language types, functions and constants in the implementation with a pthread.h header and a thread library. The implementation of POSIX Threads library is usually called Pthreads, and it usually used on Unix-like POSIX systems (FreeBSD, NetBSD, GNU/Linux, Mac OS X and Solaris), but also on Microsoft Windows (the pthreads-w32 is used on supporting a subset of the Pthread API for the Windows 32-bit platform).

2.1.7 TBB

Intel TBB (Intel Threading Building Blocks) [34] is a library developed by Intel Corporation, and it provides a rich and complete approach to represent parallelism in a C++ program. While writing software programs, it helps us take the advantage of multicore processor performance. But Intel TBB is not just a threads-replacement library. It represents a higher-level, task-based parallelism that abstracts platform details and threading mechanisms for scalability and performance. And current release is Intel TBB 3.0, it supports various operating systems such as Microsoft Windows, Mac OS X (version 10.4.4 or higher) and Linux.

2.2 Auto-Parallel Tools

About the issue of auto-parallel, there are some tools which could accord to our source code to generate the binary file or parallel code automatically.

2.2.1 ROSE

ROSE is a source-to-source compiler infrastructure of open source which builds source-to-source program transformation and analysis tools for large-scale FORTRAN, C/C++, and OpenMP applications. There are many functions in ROSE such as static analysis, program optimization, arbitrary program transformation, domain-specific optimizations, complex loop optimizations, performance analysis, and cyber-security.

2.2.2 Open64 Compiler

The Open64 [35] compiler is a complier of open source and derived from SGI Pro64(TM) compiler which was released under the GNU GPL in 2000. It supports C/C++, FORTRAN, and OpenMP, and generate code for various architectures (CISC, RISC, VLIW, GPU), including MIPS, IA-32, IA-64, CUDA, and others.

2.2.3 Intel® Composer XE 2011

Intel® Composer XE 2011 [36] is a compiler which combined with high performance libraries for C/C++ and FORTRAN on the operating system of Window or Linux. It can improve the performance with Intel TBB and OpenMP programming on multicore operating systems.

2.2.4 The Portland Group

PGI Workstation™ [37] has many kind of compilers and tools product. PGI Workstation supports C/C++, FORTRAN, OpenMP, and CUDA. It can improve the performance with CUDA and OpenMP programming on multicore operating systems. PGI products run under the operating systems, including various Linux, Mac OS X, and most versions of Microsoft Windows.

2.2.5 PAR4ALL

PAR4ALL [38] is an open-source compiler which could to do source-to-source transformations on C and FORTRAN programs. It can generate OpenMP code from C code and CUDA code from C/FORTRAN code by using the script p4a. And PAR4ALL runs under Linux of operating systems.

Chapter 3

System Hardware

3.1 Tesla C1060 GPU computing processor

The NVIDIAR Tesla™ C1060 [41] transforms a workstation into a high-performance computer that outperforms a small cluster. This gives technical professionals a dedicated computing resource at their desk-side that is much faster and more energy-efficient than a shared cluster in the data center. The NVIDIAR Tesla™ C1060 computing processor board which consists of 240 cores is a PCI Express 2.0 form factor computing add-in card based on the NVIDIA Tesla T10 graphics processing unit (GPU). This board is targeted as high-performance computing (HPC) solution for PCI Express systems. The Tesla C1060 is capable of 933 GFLOPs/s [24] of processing performance and comes standard with 4 GB of GDDR3 memory at 102 GB/s bandwidth.

A computer system with an available PCI Express ×16 slot is required for the Tesla C1060. For the best system bandwidth between the host processor and the Tesla C1060, it is recommended (but not required) that the Tesla C1060 be installed in a PCI Express ×16 Gen2 slot. The Tesla C1060 is based on the massively parallel, many-core Tesla processor, which is coupled with the standard CUDA C programming [42] environment to simplify many-core programming.

3.2 Tesla S1070 GPU computing system

The NVIDIAR Tesla™ S1070 [40] computing system speeds the transition to energy-efficient parallel computing [6]. With 960 processor cores and a standard C compiler that simplifies application development, Tesla S1070 scales to solve the world's most important computing challenges – more quickly and accurately. The NVIDIAR Tesla™ S1070 Computing System is a 1U rack-mount system with four Tesla T10 computing processors. This system connects to one or two host systems via one or two PCI Express cables. A Host Interface Card (HIC) [39] is used to connect each PCI Express cable to a host. The host interface cards are compatible with both PCI Express 1x and PCI Express 2x systems.

The Tesla S1070 GPU computing system is based on the T10 GPU from NVIDIA. It can be connected to a single host system via two PCI Express connections to that host, or connected to two separate host systems via one PCI Express connection to each host. Each NVIDIA switch and corresponding PCI Express cable connects to two of the four GPUs in the Tesla S1070. If only one PCI Express cable is connected to the Tesla S1070, only two of the GPUs will be used. To connect all four GPUs in a Tesla S1070 to a single host system, the host must have two available PCI Express slots and be configured with two cables.

3.3 ARM11 MPCore Processor

The ARM11™ MPCore™ [43] multicore processor implements the ARM11 microarchitecture and brings multicore scalability with 1 to 4 cores from a single RTL base, enabling simple system design with a single macro to integrate with up to 4x the performance of a single core. The ARM11 MPCore processor brings efficient coherency using built-in SCU, and is supported by a wide range of OS with ARM SMP capability. The processor extends the ARMv6 architecture with PIPT caches, supporting 16KB-64KB L1 caches efficiently.

The ARM11 MPCore processor provides enhanced memory throughput of 1.3Gbytes/sec from a single CPU, and a solution that delivers greater performance at lower frequencies than comparable single processor designs, offering significant cost savings to system designers, while maintaining full compatibility with existing EDA tools and flows. The ARM11 MPCore processor also simplifies otherwise complex multiprocessor design, reducing time-to-market and total design cost. Also, the ARM11 MPCore processor supports a fully coherent data cache, providing the designer with a unique level of flexibility across various symmetric multiprocessing (SMP) and asymmetric multiprocessing (AMP), or any combination of either style of multiprocessor design.

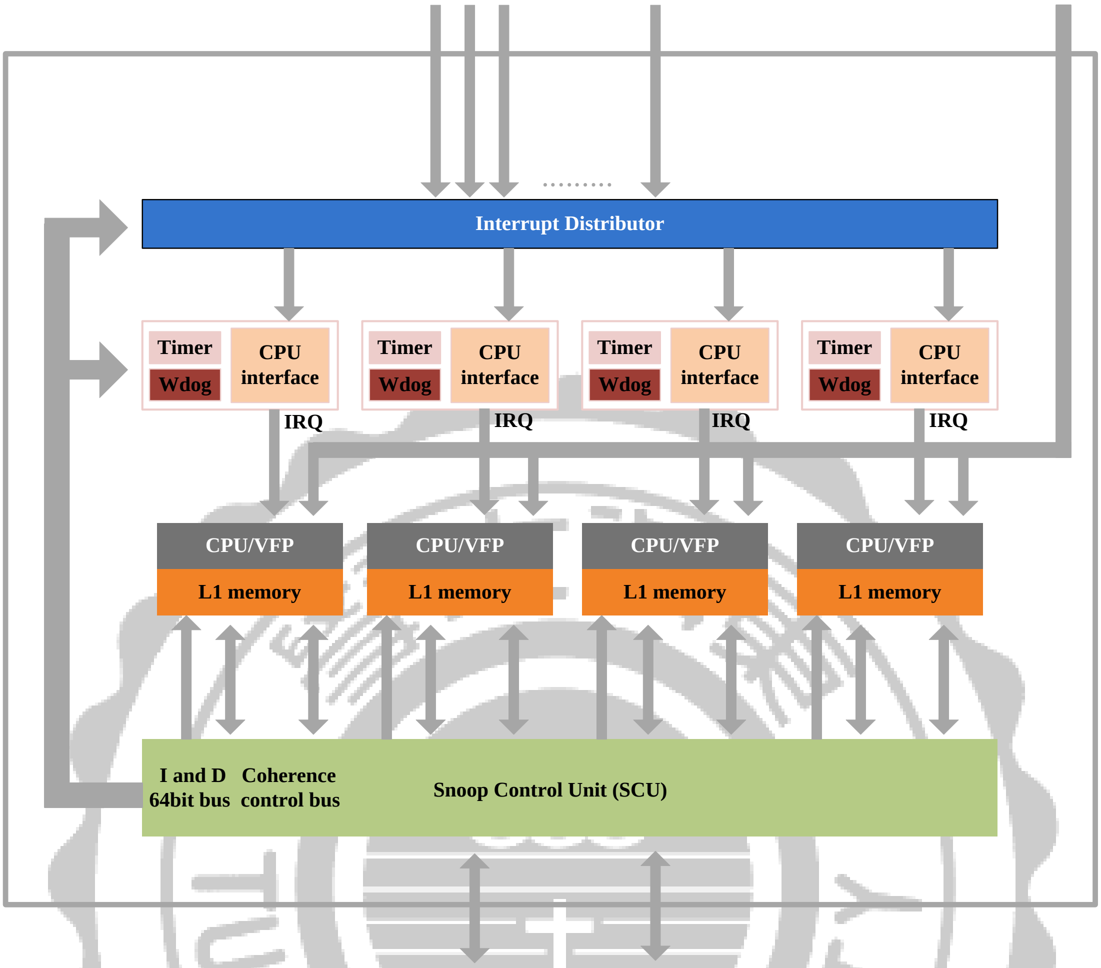


Figure 3-1. Arm11MPcore chip

Chapter 4

System Design and Implementation

4.1 Automatic Parallelization

The implementation of automatic parallelization based on OpemMP is support both C and C++ code. The auto-parallel program will insert OpenMP pragmas into the sequential C/C++ source code automatically, if possible. The system architecture of the auto-parallel tool shows as **Figure 4-1**.

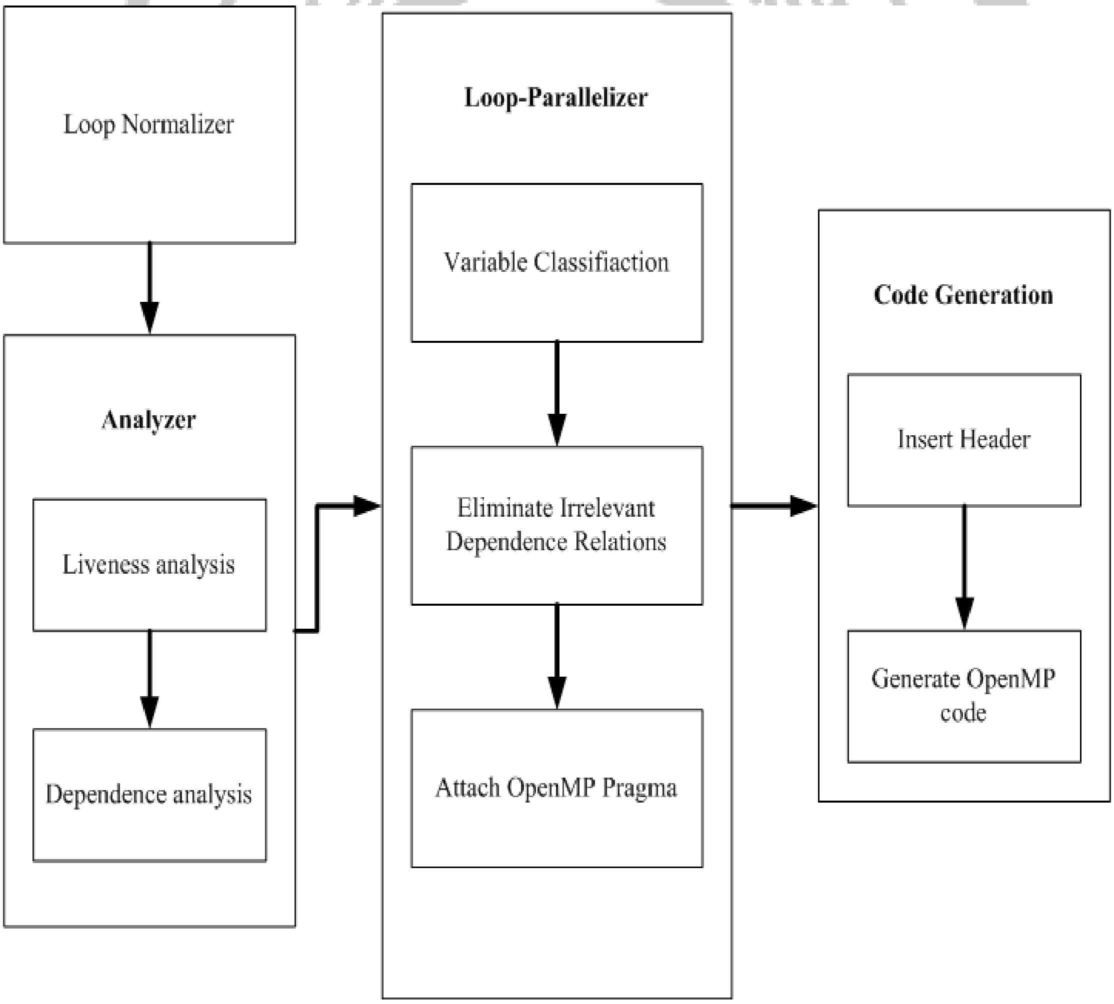


Figure 4-1. System Architecture

4.1.1 Algorithm

Automatic parallelization tool is designed to handle the conventional loops. The loops may include variables of primitive data types or STL container types. To search parts of dependence in a target and eliminate them later on as much as possible based on various rules is the critical idea of the algorithm. If there are no other dependencies, parallelization is safe. The algorithm of the auto-parallel tool shows as **Figure 4-2**.

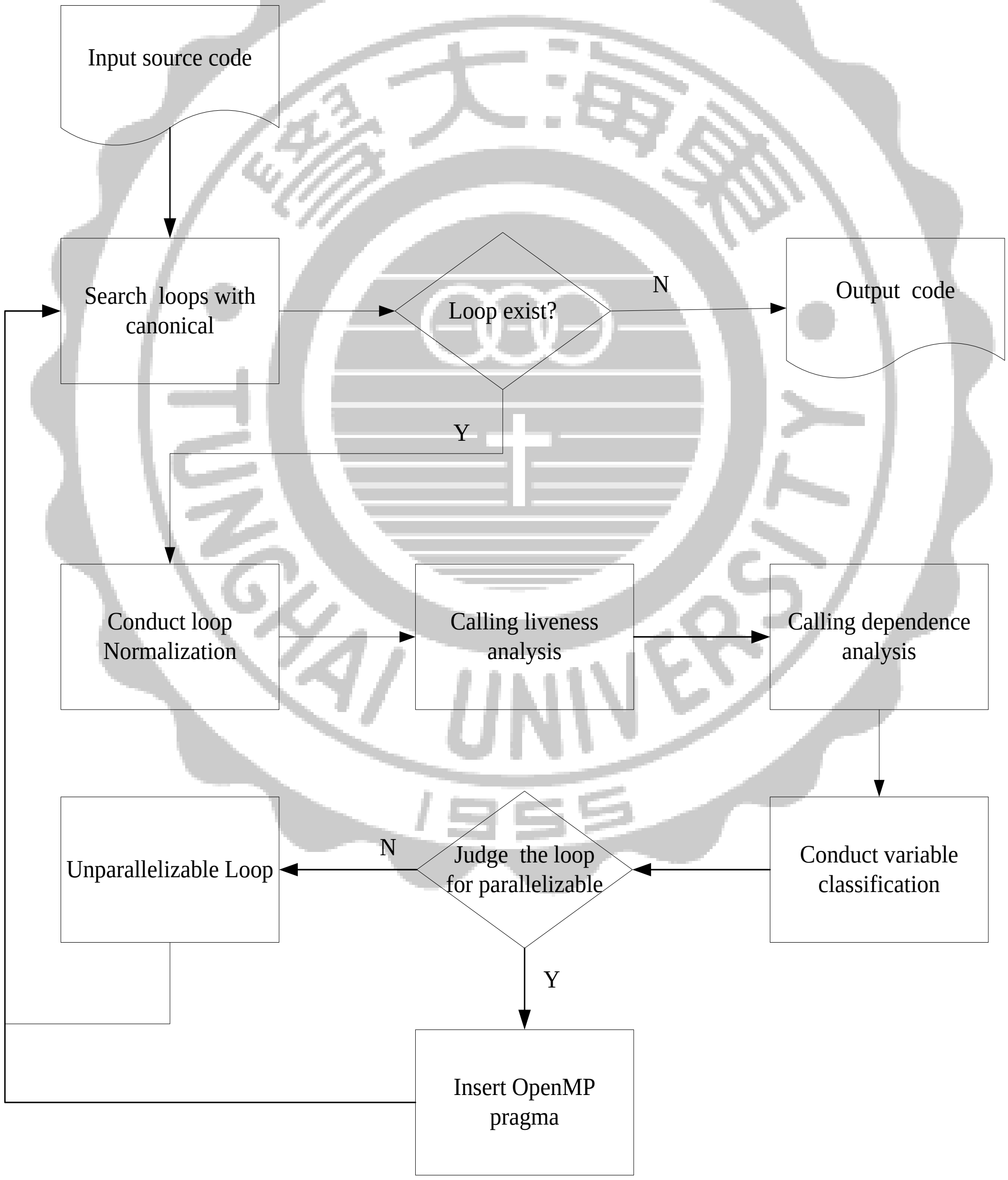


Figure 4-2. Algorithm of auto-parallel

A. To search the loops in source code and normalize the loops.

B. Liveness analysis

C. Dependence analysis

D. To classify variables of OpenMP, and identifying references to the current element, and search order-independent write accesses.

E. To eliminate dependencies associated with variables. Transform the dependent variables to the independent variables, if possible.

F. To insert the corresponding OpenMP directives.

4.1.2 Liveness Analysis

Liveness analysis is a typical data flow analysis, which to calculate the variables for each program point by the compiler. The variables may be potential to read before they write next one. If the variable holds a value which may be needed in the future, it is live at a point in a program's execution path.

And we can access live-in and live-out variables from a translator based on the virtual control flow graph node after calling liveness analysis. The code accesses the control flow graph node of the for statement and retrieve live-in variables of the true edge's target node as the live-in variables of the loop body. Similarly, when getting the live-in variables of the node after the loop, the live-out variables of the loop are obtained (target node of the false edge). Simply, live-in is the set of variables that are live at the entry point of a loop and live-out is the set of variables that are live at the exit point of a loop.

4.1.3 Dependence Analysis

Dependence analysis is used to find the constraints of execution-order between statements/instructions. Broadly speaking, a statement P1 must be executed before a statement P2 if P2 depends on P1. Two statements which access or modify the same resource may incur data dependence. The types of the data dependence included flow dependence (RAW), anti-dependence (WAR), output dependence (WAW), and input dependence (RAR). Most of the data dependence could not be parallel.

Dependence analysis is the basis for the auto-parallel tool, and the analysis is used to judge whether the statements of the loop could be executed independently. **Figure 4-3** shows an example for an input code, in which a statement is surrounded by two loops. It is clear that the example code in **Figure 4-3** cannot be parallelized because of loop-carried dependences in the loop levels.

```
for(i=0;i<m;i++){
    for(j=0;j<n;j++){
        s[i][j]=s[i][j-1] * s[i-1][j];
    }
}
```

Figure 4-3. An example of dependence

4.1.4 Variable Classification

A private variable of a loop is neither live-in nor live-out of the loop. It means the variable is immediately redefined inside the loop and then used inside the loop, but is never used anywhere after the loop. To avoid possible race condition, so all loop index variables are classified as OpenMP private variables. On the other hand, the variables of firstprivate and lastprivate are live respectively at either only the beginning or only the end of the loop. The shared variables are live at both the beginning and the end of the loop.

Reduction variables are used to increase the opportunities for parallelization. For example, the statement of z=z+a[i] in a loop is a typical operation of reduction variables. It would cause loop-carried anti-dependence and loop-carried output dependence. We use an analysis to search such typical operations, and when deciding if a loop is parallelizable, exclude the associated loop-carried dependences.

4.1.5 Interface

In order to simplify the use of ROSE for user, we integrated ROSE into Eclipse. When we want to parallelize the source code, just only need to click a button.

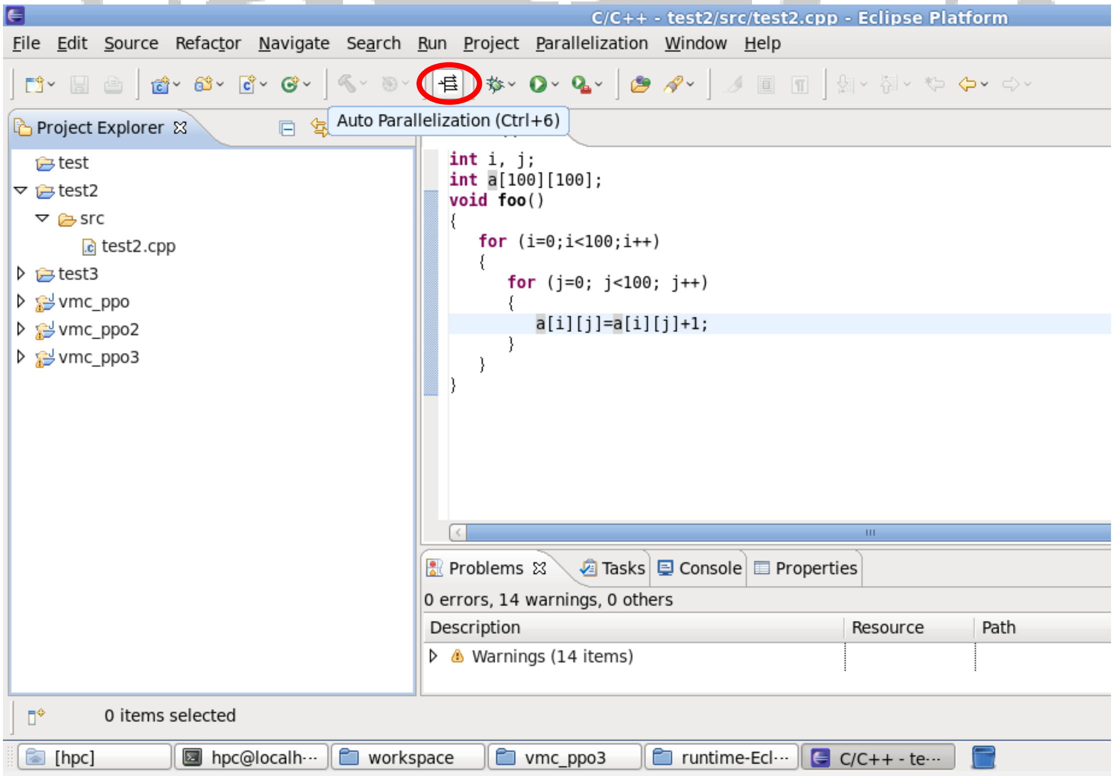


Figure 4-4. The button of automatic parallelization in Eclipse

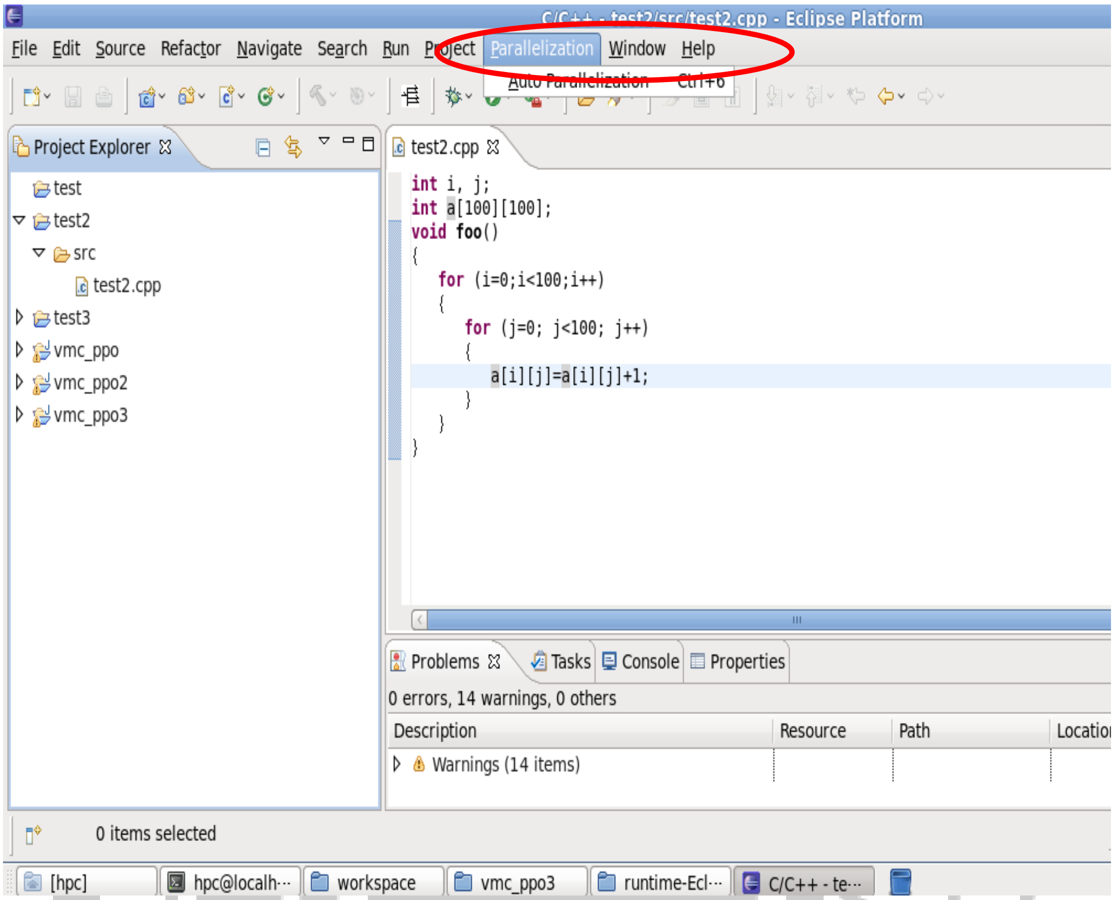


Figure 4-5. Example for auto-parallel

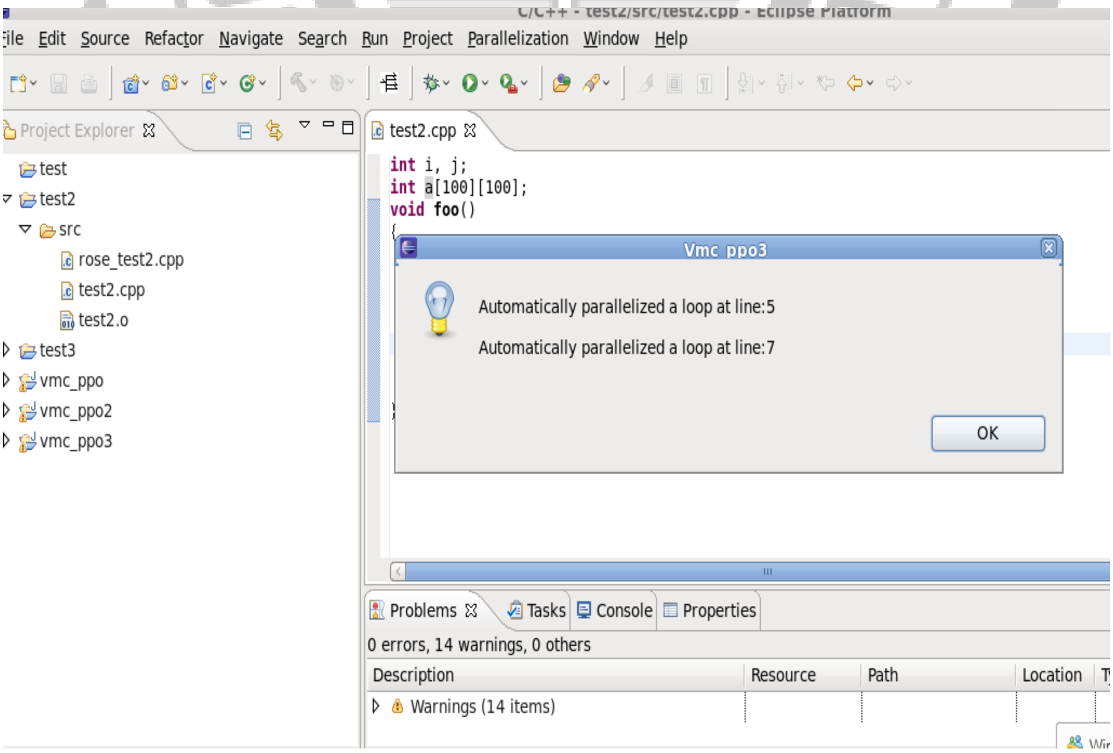


Figure 4-6. Show the lines where be inserted OpenMP pragma

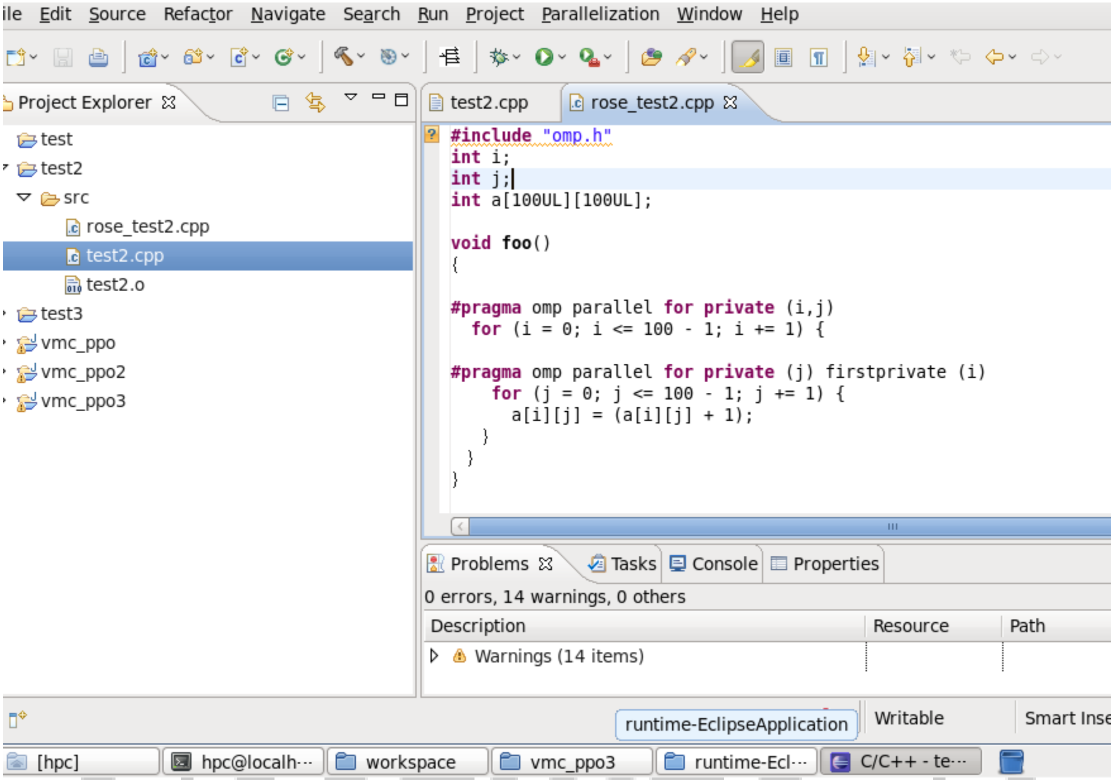


Figure 4-7. Output codes for auto-parallel

And we also implement an interface which shows in **Figure 4-8**. **Figure 4-9** shows that optimization of loop. First, open the file which will be optimized. Second, choose the parameter of optimization. Finally, click the “Optimize” button. **Figure 4-10** shows that auto-parallelization of loop. First, open the file which will be parallelized. And then click the “AutoParallel” button.

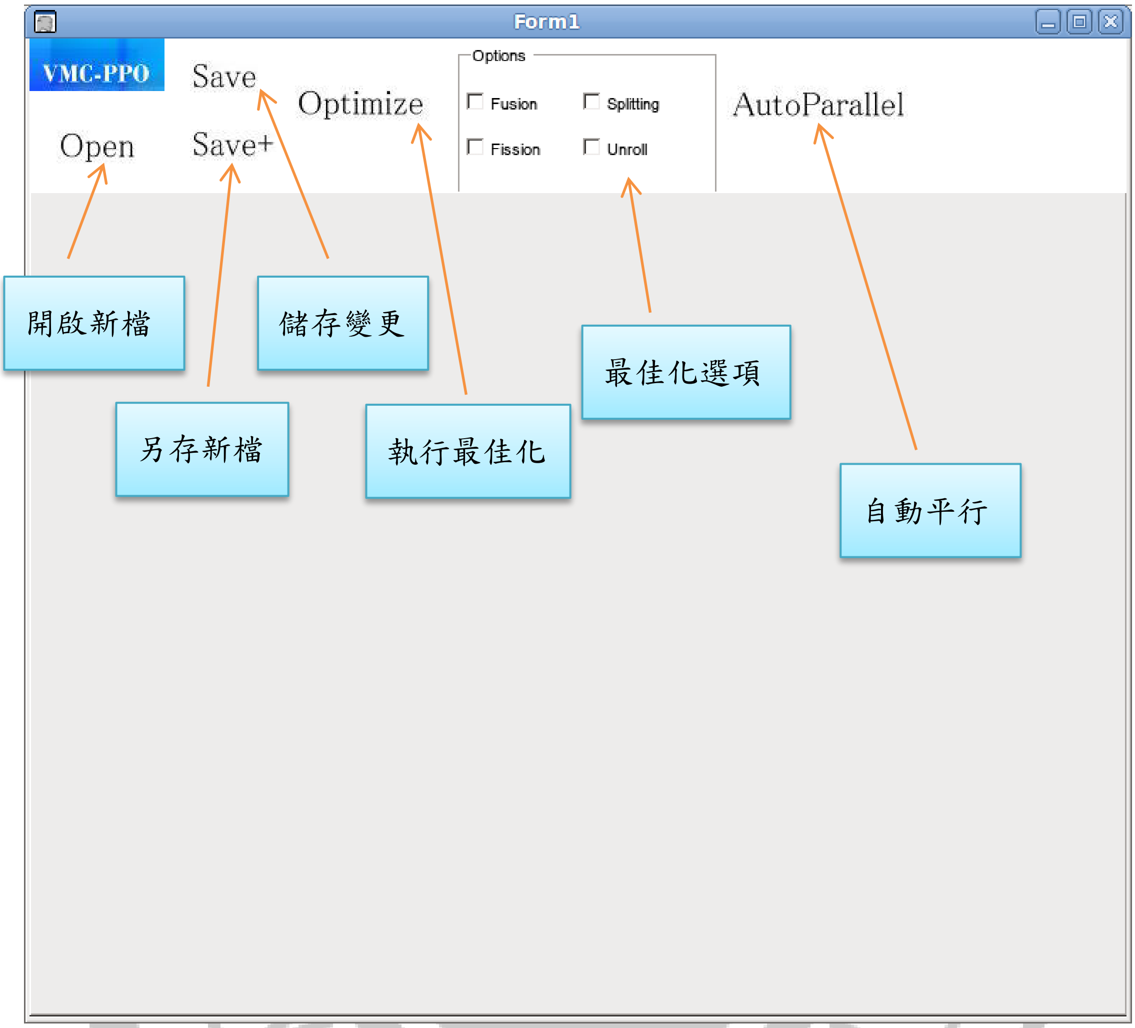


Figure 4-8. Interface of auto-parallel tool

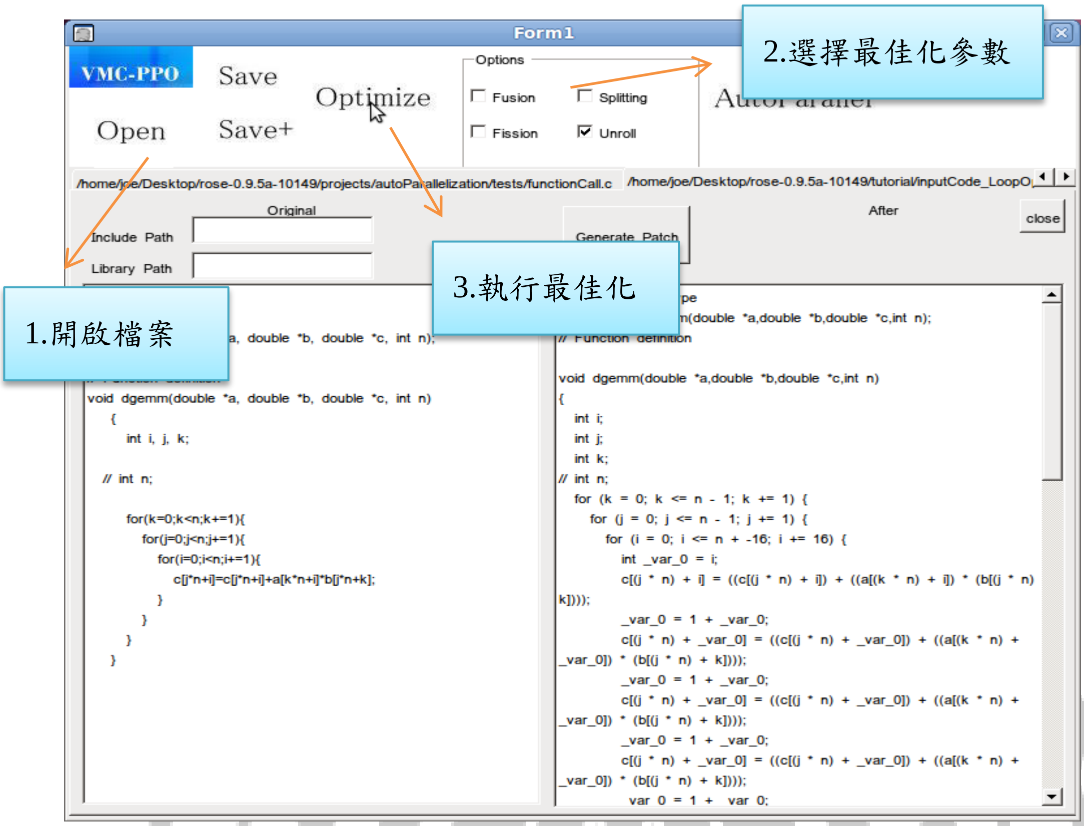


Figure 4-9. Optimization of Loop

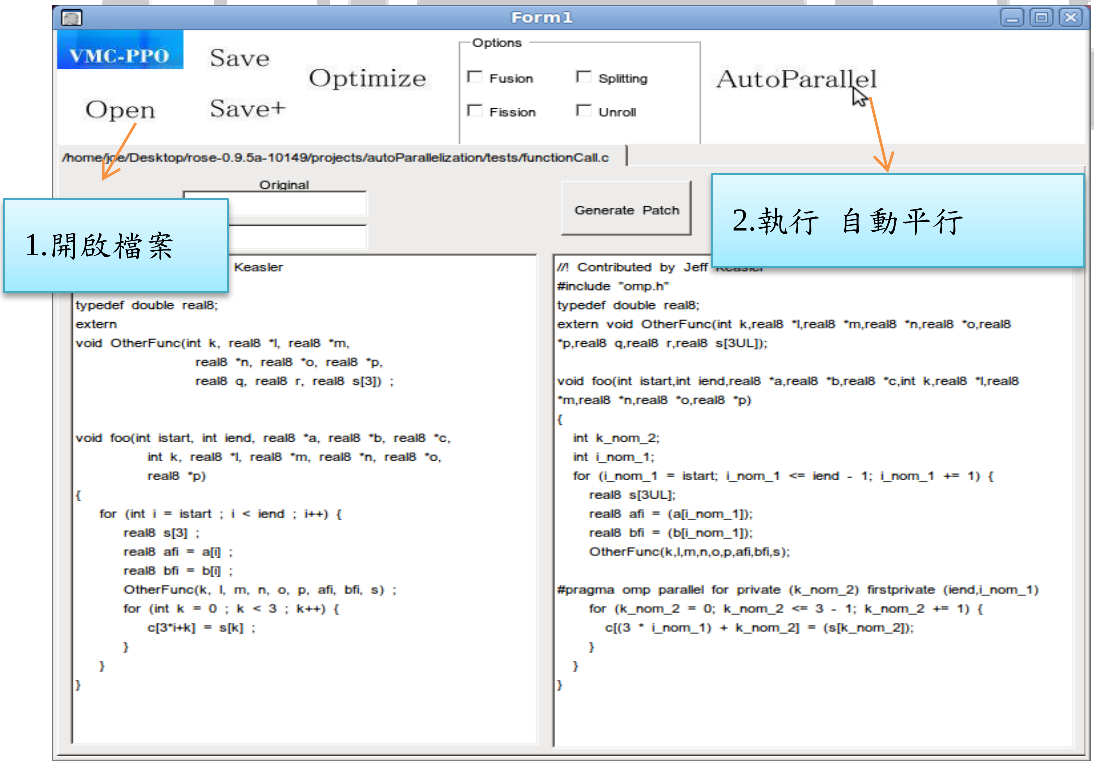


Figure 4-10. Auto-Parallelization of Loop

4.2 Hybrid Parallel Programming

4.2.1 Combining MPI and CUDA

We could use the GPUs which are in different severs by MPI to help us doing parallel computing. A sample source code files as follow:

```
#include "mpi.h"
#include <stdio.h>
#include "kernel.h"
int main(int argc,char **argv)
{
  int myid, numprocs;
  int namelen;
  char processor_name[MPI_MAX_PROCESSOR_NAME];
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Get_processor_name(processor_name,&namelen);
  go(myid,processor_name);
  MPI_Finalize();
}
```

```
#include <string.h>
#include <stdio.h>
#include <math.h>
#include <ctime>
#include <iostream>
using namespace std;
#define BLOCK 16
__global__ void hello(char* s){
      char w[50]="hello CUDA";
       int k;
      for(k=0; w[k]!=0; k++) s[k]=w[k];
       s[k]=0;
       }
```

```
       int go(int cpu_id,char* name){
           char* d;
           char  h[100];
           int gpu_id = -1;
                 if(cpu_id>=3){
                       cudaSetDevice(cpu_id % 4);
                 }
                 else
                       cudaSetDevice(cpu_id);
                 cudaGetDevice(&gpu_id);
           cudaMalloc((void**) &d, 100);
           hello<<<1,1>>>(d);
           cudaMemcpy(h, d, 100, cudaMemcpyDeviceToHost);
           printf("\n%s from Device %d on %s\n", h,gpu_id,name);
           cudaFree(d);
           return 0;
   }
```

4.2.2 Combining OpenMP and CUDA

When there are two or more devices of GPU in a server, we can get more processes through OpenMP to control the devices. A sample source code files as follow:

```
#include <string.h>
#include <math.h>
#include <ctime>
#include <iostream>
#include <omp.h>
using namespace std;
#define BLOCK 16
__global__ void hello(char* s){
       char w[50]="hello CUDA";
       int k;
       for(k=0; w[k]!=0; k++) s[k]=w[k];
       s[k]=0;
}
```

```
int go(int dn){
       char* d;
       char  h[100];
       cudaMalloc((void**) &d, 100);
       hello<<<1,1>>>(d);
       cudaMemcpy(h, d, 100, cudaMemcpyDeviceToHost);
       printf("\n%s from Device %d\n", h,dn);
       cudaFree(d);
       return 0;
   }
```

```
int main(int argc,char *argv[]){
         int numberOfGpuThread=0;
              int deviceCount;
              int device;
              cudaGetDeviceCount(&deviceCount);
             printf("\nThere are %d devices supporting CUDA\n", deviceCount);
        for(int i=0;i<argc;i++){
               if(!strcmp(argv[i],"-openmp")){
              numberOfGpuThread=atoi(argv[i+1]);
              }
        }
        printf("\nYou select %d device(s)\n",numberOfGpuThread);
        #pragma omp parallel for num_threads(numberOfGpuThread)
              for(int i=0;i<numberOfGpuThread;i++){
                     int id=omp_get_thread_num();
                     cudaSetDevice(id % numberOfGpuThread);
            cudaGetDevice(&device);
                     //printf("From device %d\n",device);
                     go(device);
              }
}
```

So maybe we could translate sequential codes into OpenMP codes by our auto-parallel tool, and then manually converted to CUDA code.

4.2.3 System model and approach

The system model is presented in **Figure 4-11**, a hybrid CUDA GPU cluster is built with two GPU Servers as shown as S1070 and C1060, which connected with a

Gigabit Swatch. The S1070 1U server attached to Intel i7 Server is connected with double PCI express channel for enhancing the internal-communication. We take the Intel Core i7 which contains four cores as the control group for comparing with the performance for GPU and CPU. In order to execute MPI and OpenMP application by CUDA, the simplest way forward for combining MPI and OpenMP upon CUDA GPU is to use the CUDA compiler-NVCC [44] for everything. The NVCC compiler wrapper is somewhat more complex than the typical mpicc compiler wrapper, so it's easier to translate MPI and OpenMP codes into.cu and compile with NVCC than the other way around.

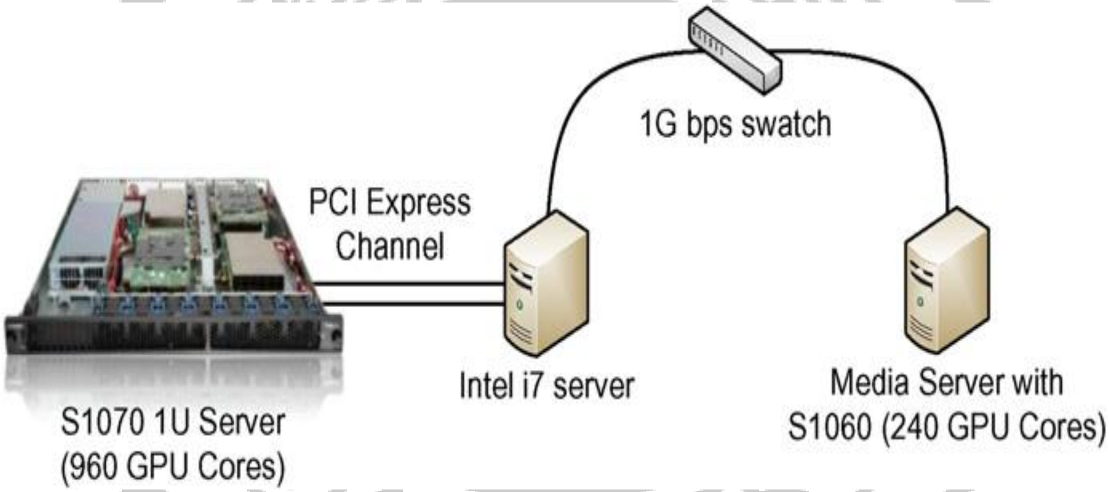


Figure 4-11. System model: The hybrid CUDA GPU cluster.

Chapter 5

Experimental Results

5.1 Part of Auto-parallelism

First, we focused on the availability of these auto-parallel tools, and checked the correctness of the execution results. How much effect of the performance we could get. And try to know that could we get the same effect on embedded system. Experimental procedure as shown in **Figure 5-1**, auto-parallel tools could be separated into two types, one is source-to-binary and the other is source-to-source. The first one means that the kind of auto-parallel tools generate the executable files directly form original codes. The next one means that the kind of auto-parallel tools can get the transformed codes from source codes. If we would have experiment on embedded system, we needed the kind of source-to-source.

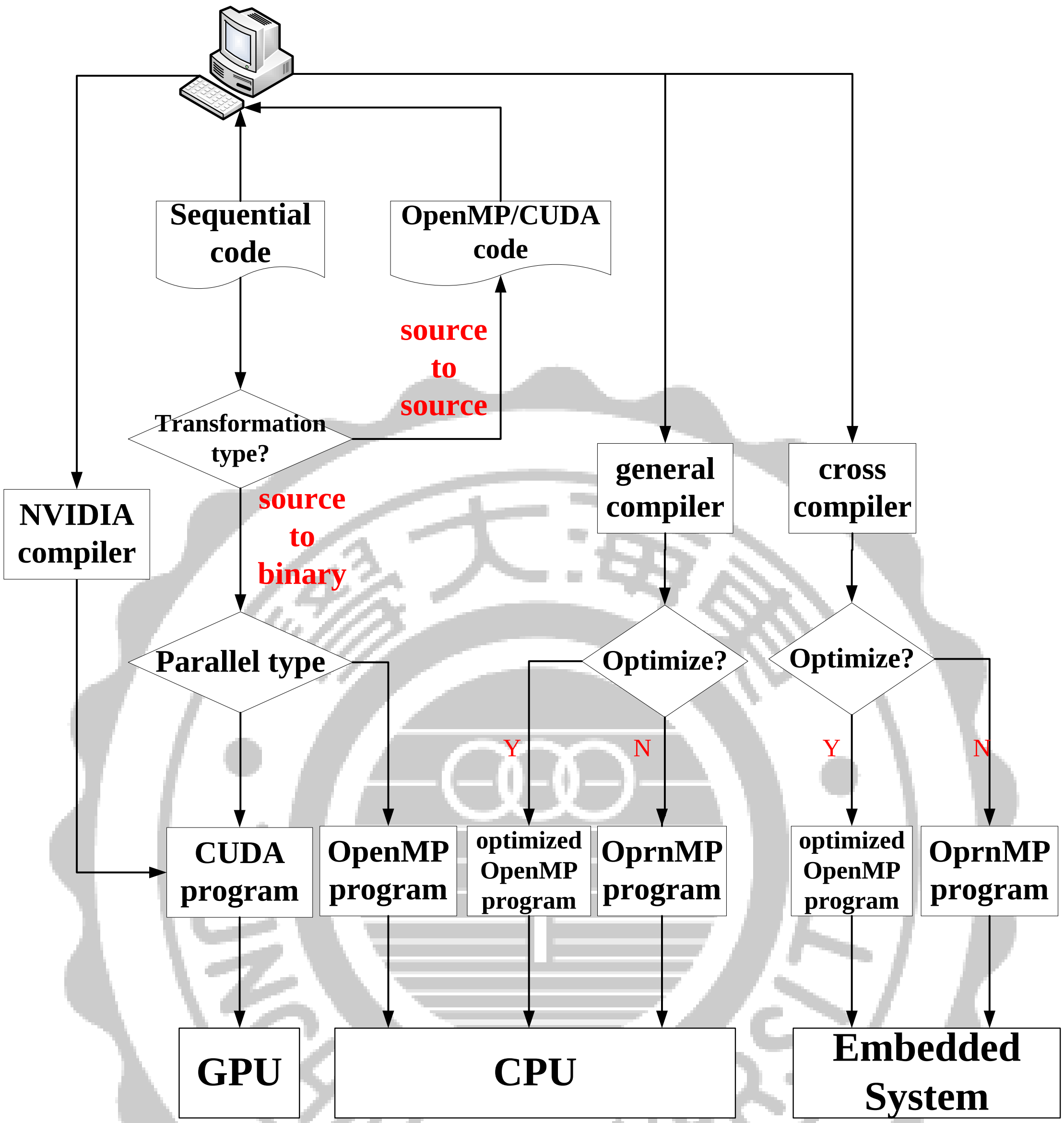


Figure 5-1. Processing flow

Then, we used these auto-parallel tools to generate the codes and the programs which will be used in our experiment, and our experiment programs are matrix multiplication, Nbody, and Jacobi. We exclude any factors which may affect the performance in our experimental environments. And we executed every program 10 times, and took the average of execution time to compare. We also checked the answer that is correct or not. We marked the wrong answer to show the situation. That's all for ensuring the accuracy of the measured data. During the experiment, we used the "time" command to get the execution time of every program. In the experiment, "sequential" means that the program was non-parallel, and "-O2" mean that we used the command to optimize. From **Figure 5-2** to **Figure 5-16**, we measure the execution time to the differences. **Table 5-1** shows the classification of auto-parallel tools.

Table 5-1. The classification of auto-parallel tools

source-to-source	source-to-binary
A. ROSE	A. Open64
B. Par4All	B. Intel
	C. PGI

5.1.1 CPU (OpenMP version)

First, we measure the performance on PC, and the experimental environment as follow:

- CPU: Genuine Intel(R) CPU U7300 @ 1.30GHz(2 cores)
- RAM:3GB

In this part, the auto-parallel tools of source-to-binary just do optimization on the sequential code. **Figure 5-2** shows that the performance on processing the massively parallel execution as the application of Matrix Multiplication from matrix size 512 to 2048. All the tools both improve the performance in the matrix multiplication program, and the tool of Intel has the best performance than others.

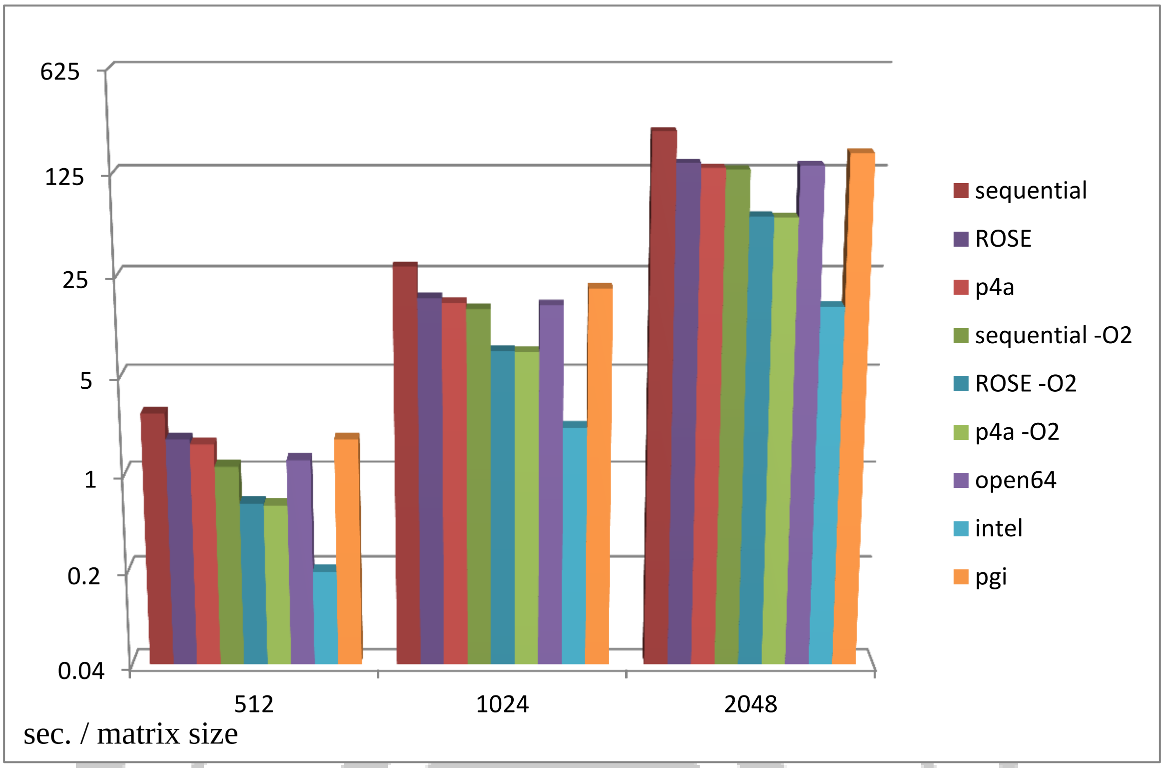


Figure 5-2. Matrix Multiplication runs on CPU

Figure 5-3 shows that the performance measurements of n-body. In the experiment, not any auto-parallel tool could improve the performance form the sequential code. When we compile the code which is transformed by ROSE, we got error. Although we could transform successfully by using PAR4ALL, from the performance result in **Figure 5-3**, PAR4ALL cannot give us a better performance than other compilers, even giving us a lower performance than the program with original code. The **Figure 5-4** shows the performance measurements of Jacobi. All the tools both improve the performance form the sequential code, and the tool of Intel also has the best performance than others.

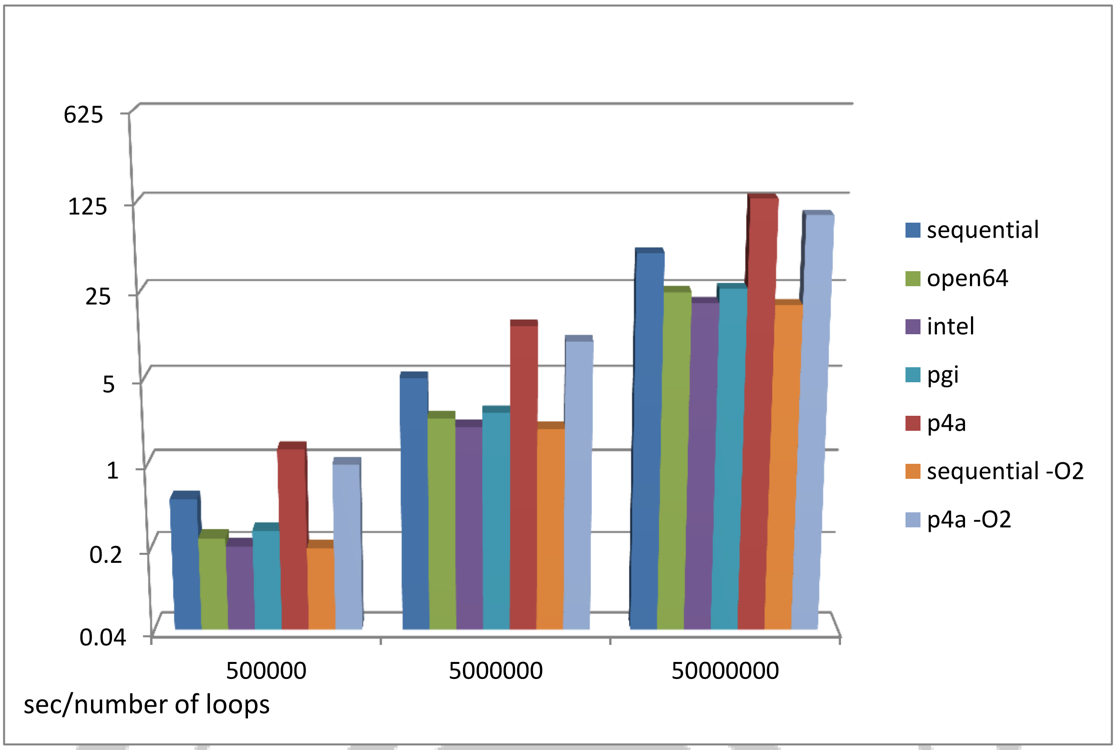


Figure 5-3. Nbody runs on CPU

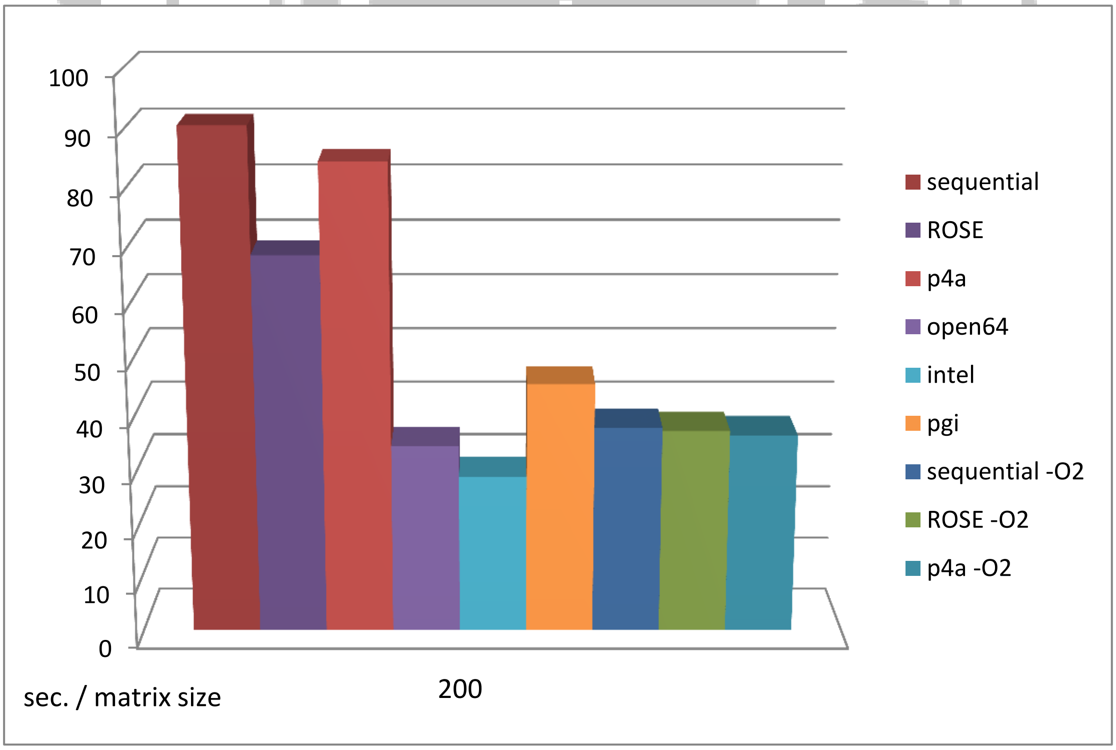
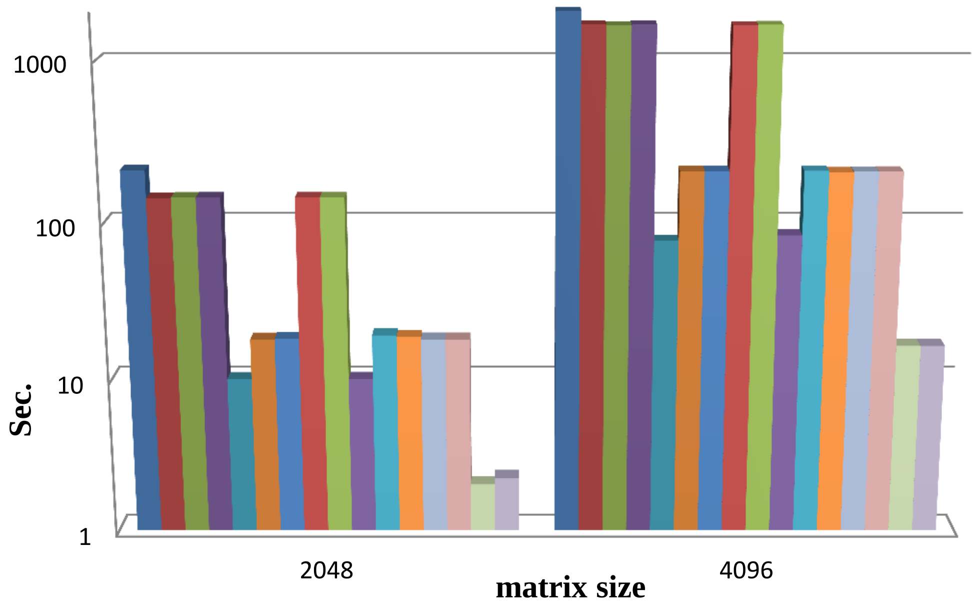


Figure 5-4. Solve problem by Jacobi method on CPU

And then, also measure the performance on our server with 8 cores. The experimental environment as follow:

- CPU: Intel(R) Xeon(R) CPU E5520 @ 2.27GHz (8 cores) [54]
- RAM: 8GB

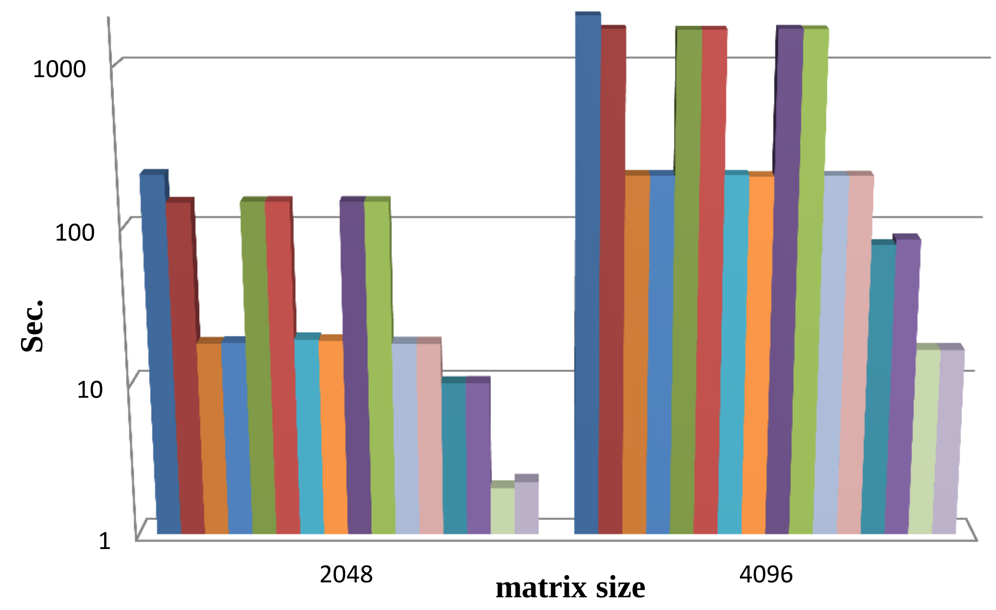
In this part of the experiment, we tried to combine the source-to-source compiler and the source-to-binary compiler. First, generate the OpenMP code through the source-to-source compiler, and then compile the code by using the source-to-binary compiler. In the experiment, we recorded the parameters what we used with each kind of compiler, and also verified that whether the parallelism program execute with multicore. In **Figure 5-5** to **Figure 5-9**, “-apo” is the parameter of Open64 compiler which could do auto-parallel to the sequential code. “-mp” is the parameter of Open64 compiler and PGI compiler which could support OpenMP code. “-fast” is the parameter of PGI compiler which could optimize the sequential code. “-parallel” is the parameter of Intel compiler which could do auto-parallel to the sequential code. “-openmp” is the parameter of Intel compiler which could support OpenMP code. “-Mconcur” is the parameter of PGI compiler which could do auto-parallel to the sequential code.



	2048	4096
sequential	213.143	1991.279
sequential -O2	143.093	1642.686
openCC -O2	143.913	1629.159
pgcc -fast	143.914	1642.224
icpc -O2	9.994	77.178
ROSE -O2	18.094	210.291
p4a -O2	18.246	209.932
opencc -apo -O2	144.044	1632.846
pgcc -Mconcur -fast	143.979	1639.771
icpc -parallel -O2	10.001	83.477
opencc+ROSE -mp -O2	19.24	211.019
opencc+p4a -mp -O2	18.822	206.724
pgcc+ROSE -mp -fast	18.129	208.764
pgcc+p4a -mp -fast	18.113	209.17
icpc+ROSE -openmp -O2	2.026	16.473
icpc+p4a -openmp -O2	2.225	16.471

Figure 5-5. Matrix Multiplication runs on the CPU with 8 cores

In **Figure 5-5**, Intel compiler gets the best performance on optimized version even better than some parallel programs. In parallel version, Intel compiler with OpenMP code of ROSE also gets the best performance.



	2048	4096
sequential	213.143	1991.279
sequential -O2	143.093	1642.686
ROSE -O2	18.094	210.291
p4a -O2	18.246	209.932
openCC -O2	143.913	1629.159
opencc -apo -O2	144.044	1632.846
opencc+ROSE -mp -O2	19.24	211.019
opencc+p4a -mp -O2	18.822	206.724
pgcc -fast	143.914	1642.224
pgcc -Mconcur -fast	143.979	1639.771
pgcc+ROSE -mp -fast	18.129	208.764
pgcc+p4a -mp -fast	18.113	209.17
icpc -O2	9.994	77.178
icpc -parallel -O2	10.001	83.477
icpc+ROSE -openmp -O2	2.026	16.473
icpc+p4a -openmp -O2	2.225	16.471

Figure 5-6. Matrix Multiplication runs on the CPU with 8 cores

In **Figure 5-6** and **Figure 5-9**, it is clearly to show the difference of performance on each kind of compiler. In **Figure 5-6**, the compilers of GNU, PGI, and Open64 have similar performance, only the compiler of Intel improve the performance obviously form the same code.

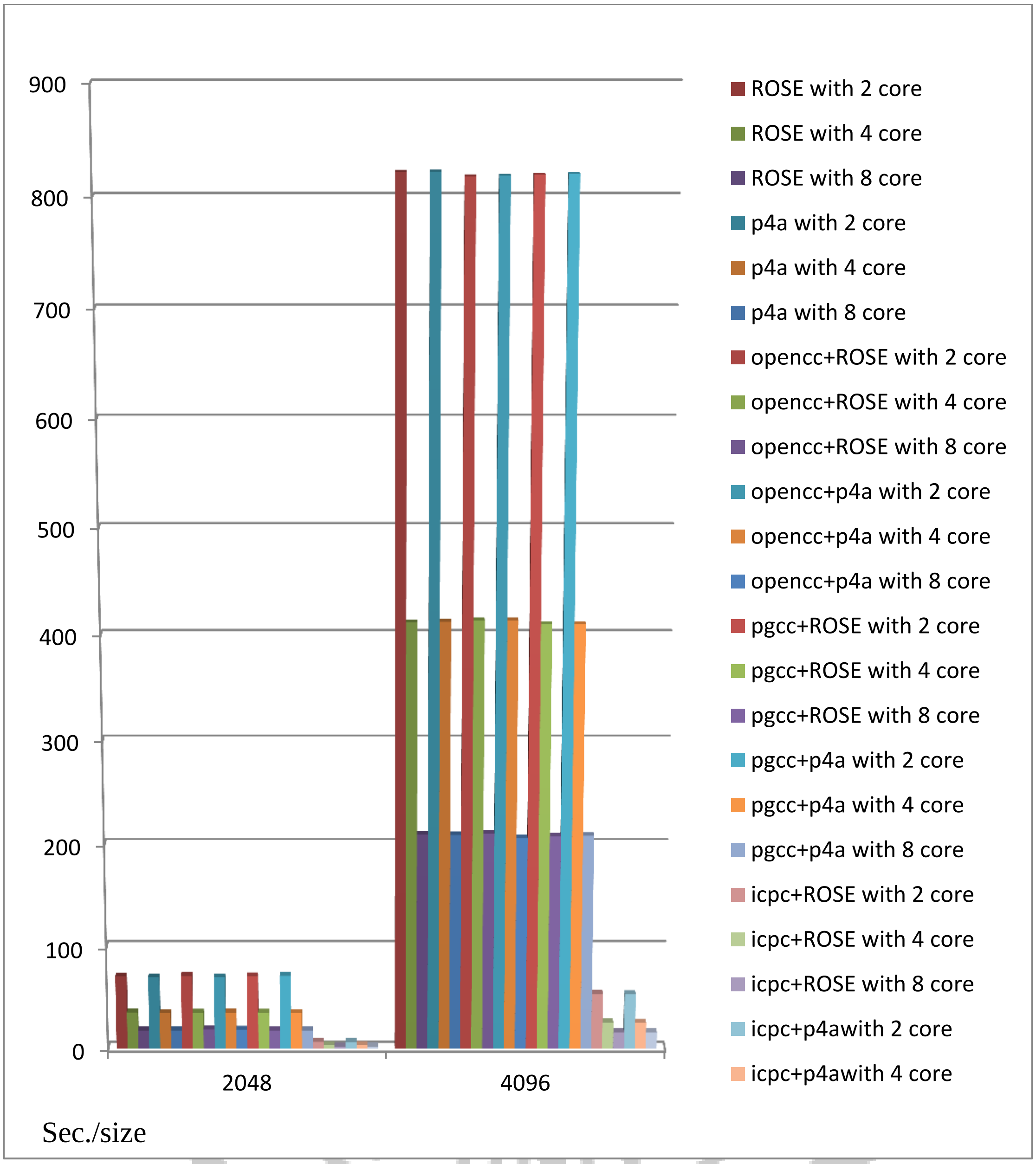
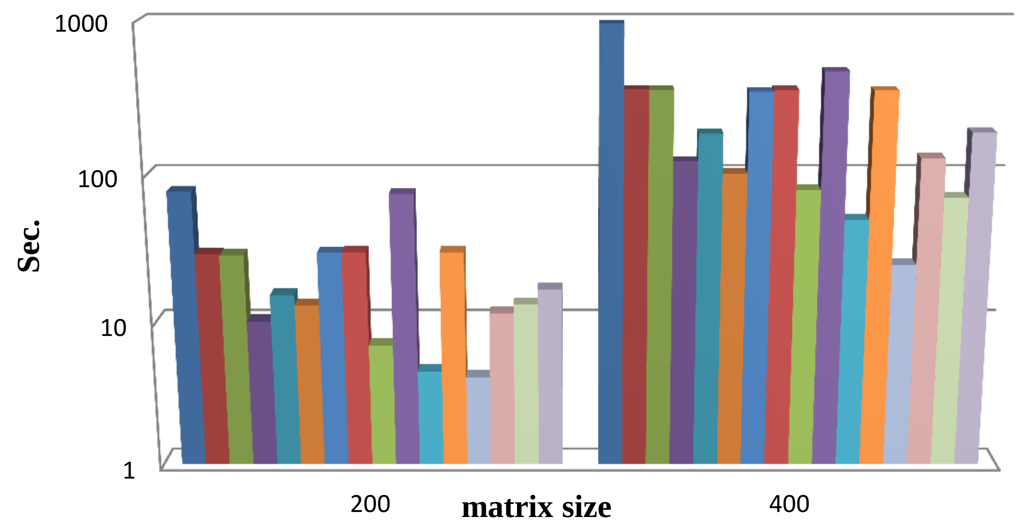


Figure 5-7. Matrix Multiplication runs with 2cores to 8 cores

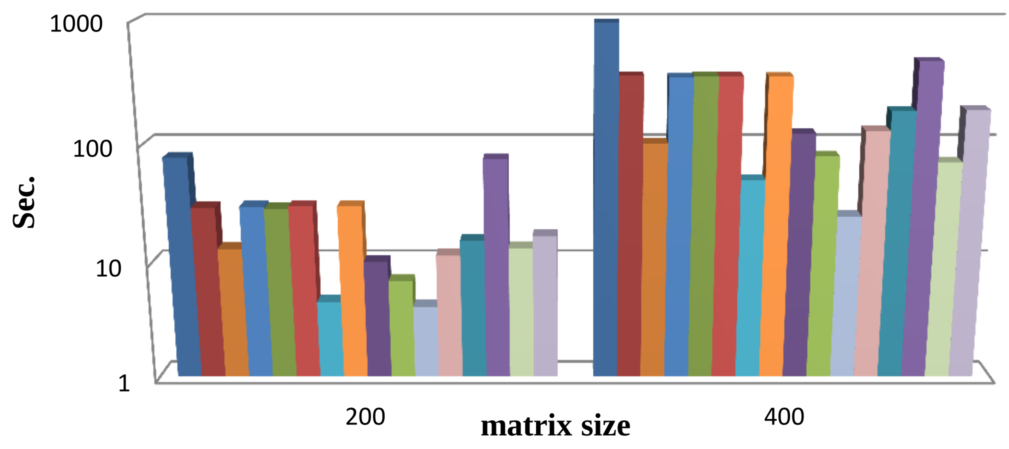
From **Figure 5-7**, we think that the matrix multiplication program is suited be compiled by Intel compiler. In the experiment of matrix multiplication, the program which be compiled by using the Intel compiler, PGI compiler and Open64 compiler with their auto-parallel parameter both execute with just only one core.



	200	400
sequential	77.2412	956.465
sequential -O2	29.3658	360.588
openCC -O2	28.7794	358.246
pgcc -fast	10.1406	121.941
icpc -O2	15.349	185.634
ROSE -O2	13.0842	101.296
p4a -O2	29.9184	349.533
opencc -apo -O2	30.1542	357.469
pgcc -Mconcur -fast	6.886	78.849
icpc -parallel -O2	74.663	471.931
opencc+ROSE -mp -O2	4.5398	50.053
opencc+p4a -mp -O2	30.1676	355.4
pgcc+ROSE -mp -fast	4.105	24.74
pgcc+p4a -mp -fast	11.506	128.314
icpc+ROSE -openmp -O2	13.287	70.423
icpc+p4a -openmp -O2	16.8078	190.255

Figure 5-8. Solve problem by Jacobi method on the CPU with 8 cores

In **Figure 5-8**, PGI compiler gets the best performance on optimized version even better than some parallel programs. In parallel version, PGI compiler with OpenMP code of ROSE gets the best performance.



	200	400
sequential	77.2412	956.465
sequential -O2	29.3658	360.588
ROSE -O2	13.0842	101.296
p4a -O2	29.9184	349.533
openCC -O2	28.7794	358.246
opencc -apo -O2	30.1542	357.469
opencc+ROSE -mp -O2	4.5398	50.053
opencc+p4a -mp -O2	30.1676	355.4
pgcc -fast	10.1406	121.941
pgcc -Mconcur -fast	6.886	78.849
pgcc+ROSE -mp -fast	4.105	24.74
pgcc+p4a -mp -fast	11.506	128.314
icpc -O2	15.349	185.634
icpc -parallel -O2	74.663	471.931
icpc+ROSE -openmp -O2	13.287	70.423
icpc+p4a -openmp -O2	16.8078	190.255

Figure 5-9. Solve problem by Jacobi method on the CPU with 8 cores

From this experiment, no matter source-to-source compilers or source-to-binary compilers almost both improve the performance for us, but not every parallel program all executes with multicore. And the utility rate of CPU would also affect the performance.

In the experiment of solving problem by Jacobi method, the CPU utility rate of the OpenMP program which be compiled by general compiler is lower than the OpenMP program which be compiled by Intel/Open64/PGI compiler. Sometimes, the performance of the OpenMP code which be generated by PAR4ALL is lower than ROSE because of the utility rate of CPU.

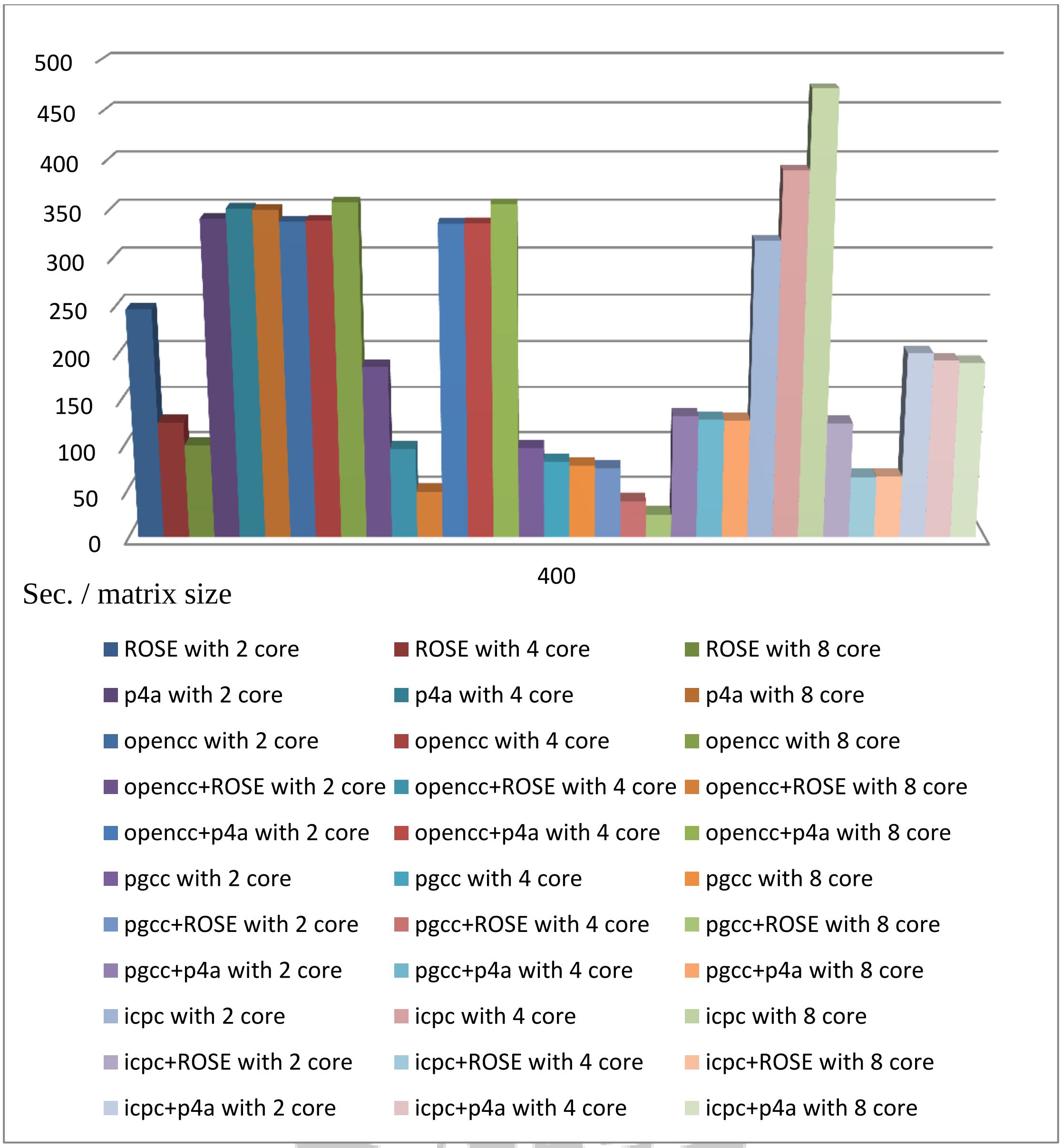


Figure 5-10. Jacobi program runs with 2 cores to 8 cores

In **Figure 5-10**, there are some abnormal performance the OpenMP code of Par4All and the auto-parllel tool of Intel and Open64.

5.1.2 GPU (CUDA version)

Second, we measure the performance on the GPU, and the experimental environment as follow:

- CPU: Intel(R) Xeon(R) CPU E5410 @ 2.33GHz(8 cores) [45]
- RAM: 4GB
- GPU: Tesla C1060

The compilers mentioned earlier, not everyone is support automatically translate to CUDA. PAR4ALL and PGI are the tools that we used in the current environment, and then using these tools to generate CUDA application runs on GPU and comparing with C application runs on CPU. The benchmarks of this part are the same with the benchmarks of 5.1.1, but the method of parallelize is different.

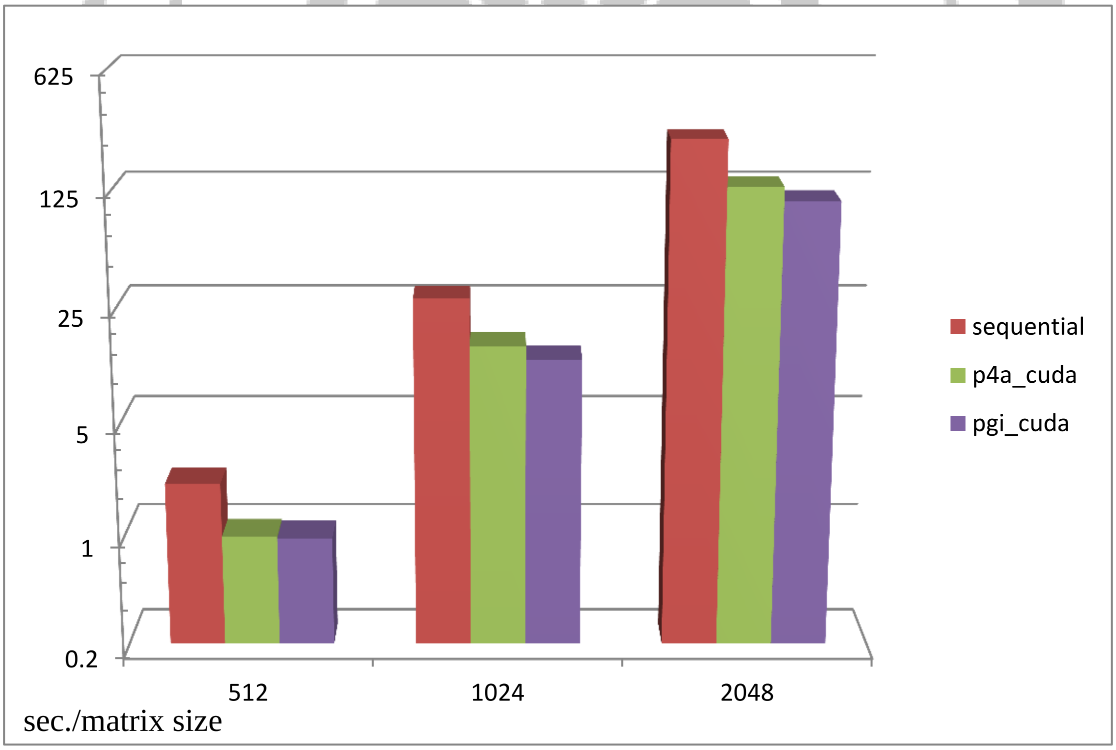


Figure 5-11. Matrix Multiplication runs on GPU

Figure 5-11 shows that the performance on processing the massively parallel execution as the application of Matrix Multiplication from size 512 to 2048.

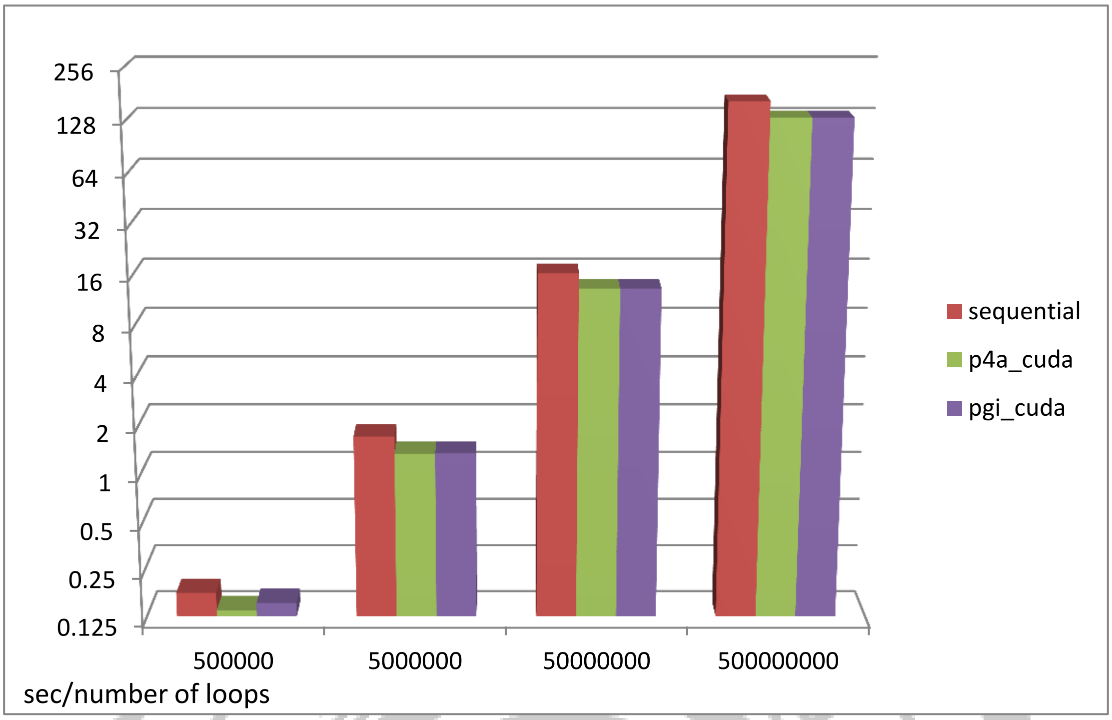


Figure 5-12. Nbody runs on GPU

In **Figure 5-12**, the performance gap between CPU and GPU is small. The accelerator (GPU) in this case has only speed up a little. The results of Par4all and PGI in this example are almost the same.

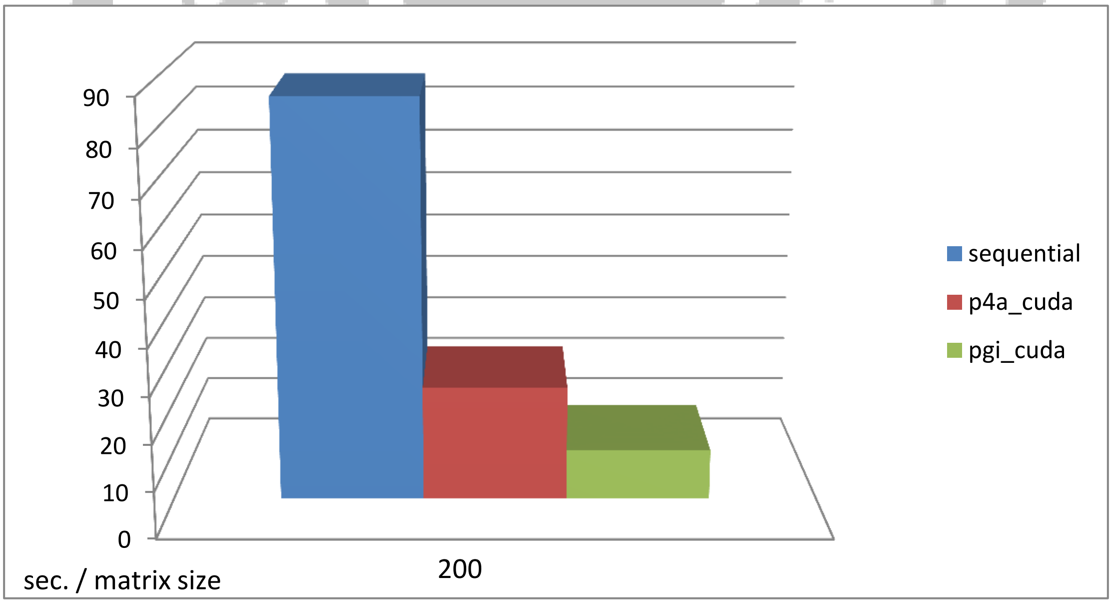


Figure 5-13. Solve problem by Jacobi method on GPU

In **Figure 5-13**, the results of these three applications are obvious. GPU is faster than CPU in this case, and the effective of PGI is better than PAR4ALL.

5.1.3 Embedded System (OpenMP version)

Finally, we measure the performance on Arm11MP Core, and the experimental environment as follow:

- CPU: ARMv6-compatible processor rev 0 (v6l)
- bogoMIPS: 83.76, 83.55, 83.35, 83.35
- RAM: 128M

Only ROSE and PAR4ALL are the compilers of source-to-source, so we measured the performance on the platform of Arm11MP core for these tools. In **Figure 5-14** and **Figure 5-15**, the performance results are similar as the performance results in 5.1.1. In **Figure 5-16**, PAR4ALL didn't give us a better performance, even worse than sequential. From all of the experiment in 5.1, we think the tool of PAR4ALL is unstable because it has more problems than others.

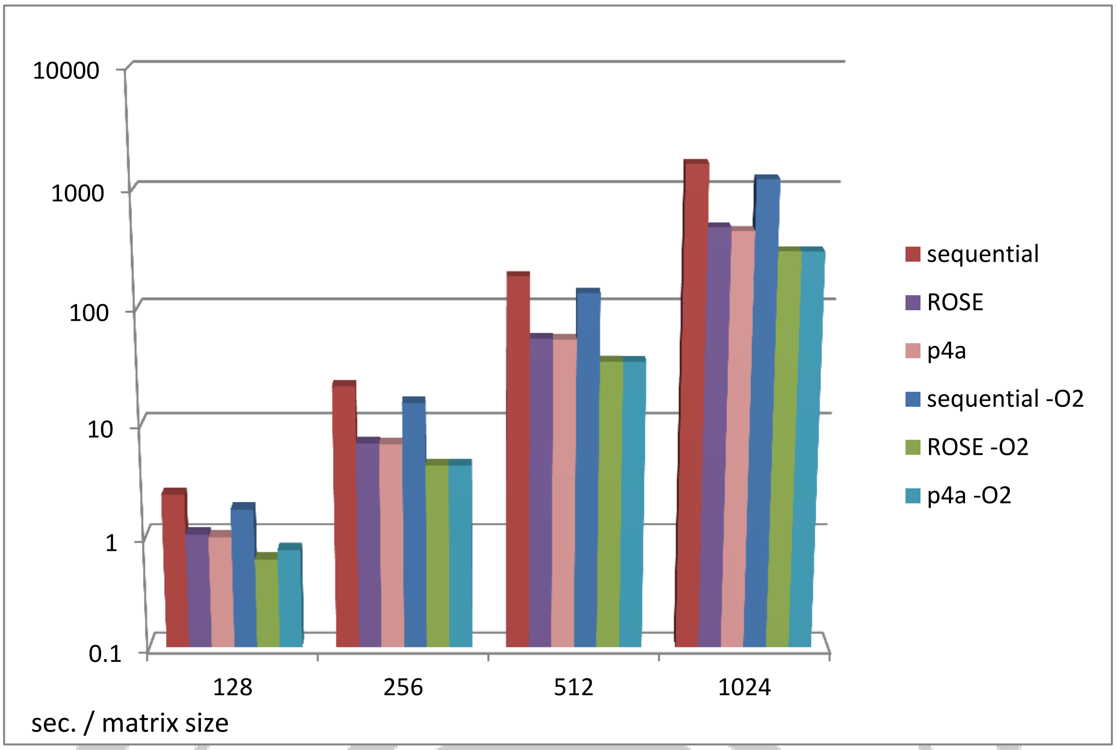


Figure 5-14. Matrix Multiplication runs on embedded system

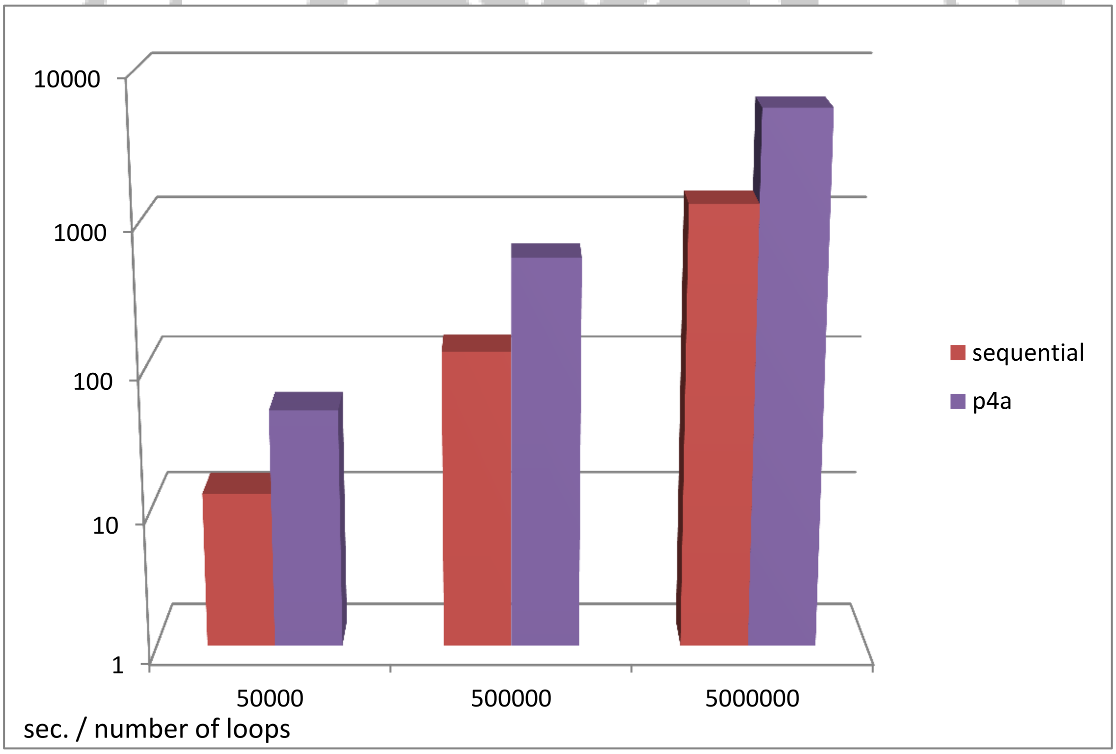


Figure 5-15. Nbody runs on embedded system

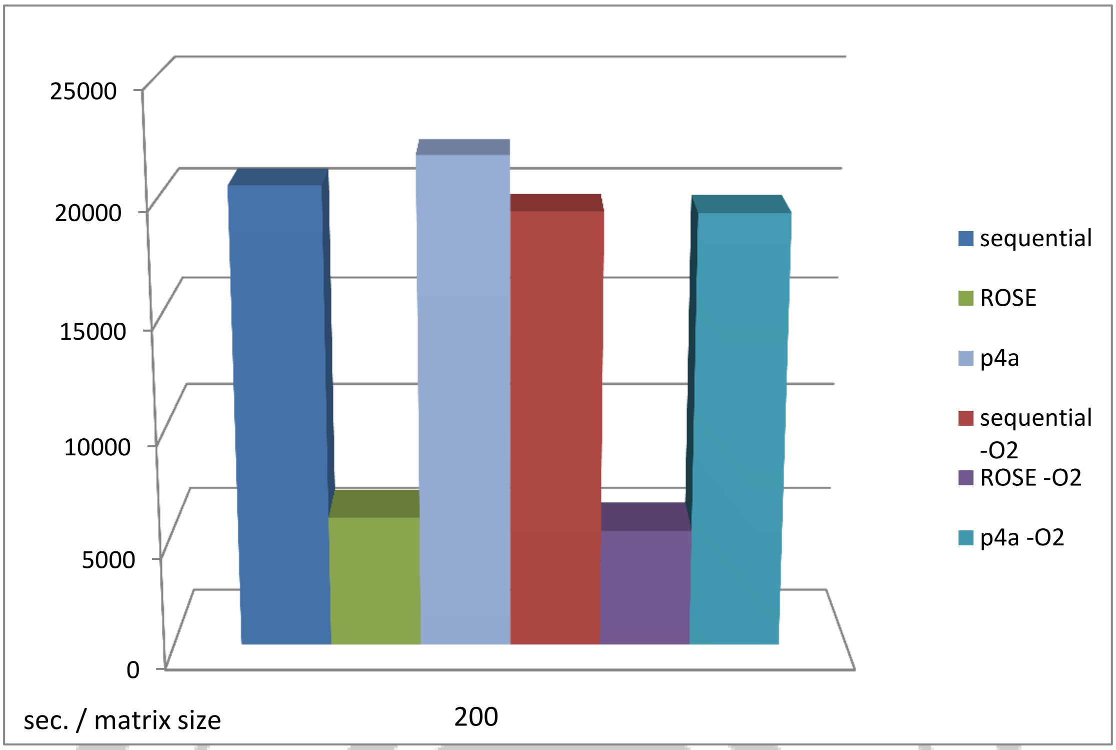


Figure 5-16. Solve problem by Jacobi method on embedded system

Finally, we compared the performance in the three kinds of environment. From Figure 5-17 and Figure 5-18, the best performance is the OpenMP code of ROSE compiled by Intel compiler. We think that perhaps automatic parallel technology of CUDA is not mature, so we get the results. In theory, the performance of CUDA should better than OpenMP.

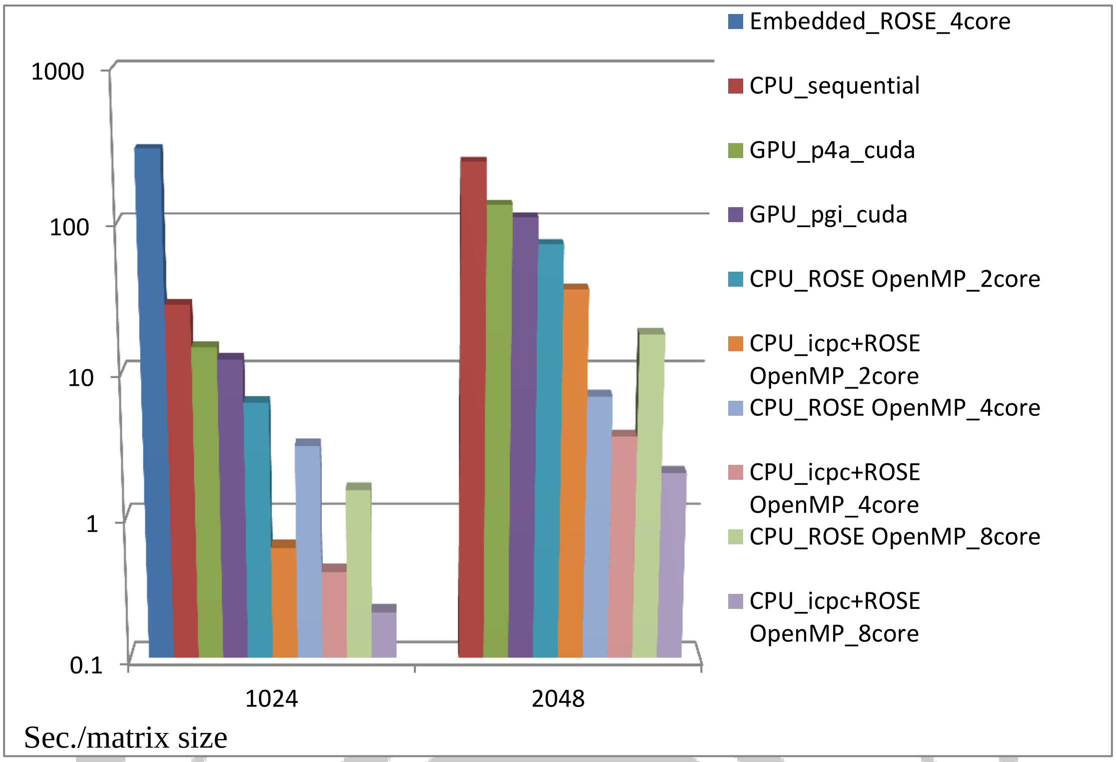


Figure 5-17. Matrix Multiplication executes on three kinds of environment

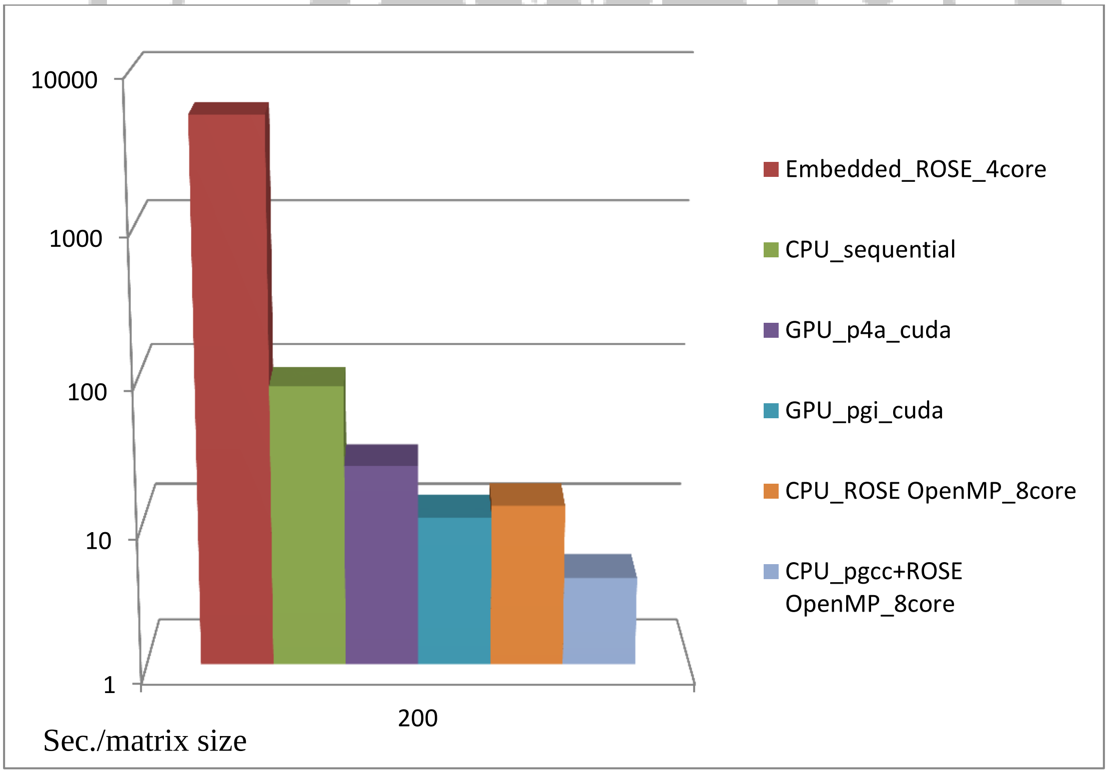


Figure 5-18. Jacobi program executes on three kinds of environment

5.2 Part of Hybrid Parallel Programming

We built a hybrid CUDA GPU cluster consisting of one Tesla C1060 and a Tesla S1070, each with Gigabit Ethernet NIC interconnected via a D-LINK DGS-3100-24 Gigabit switch. To verify our approach, illustrate our cluster environment, and describe the terminology for our application, we implemented programs with MPI/OpenMP for execution on our testbed. We then verify the performance of our scheme upon the hybrid CUDA GPU cluster to solve problems in Matrix Multiplication, MD5 and Merge Sorting. From **Figure 5-19** to **Figure 5-21**we take log of 10 at execution time to emphasize the differences.

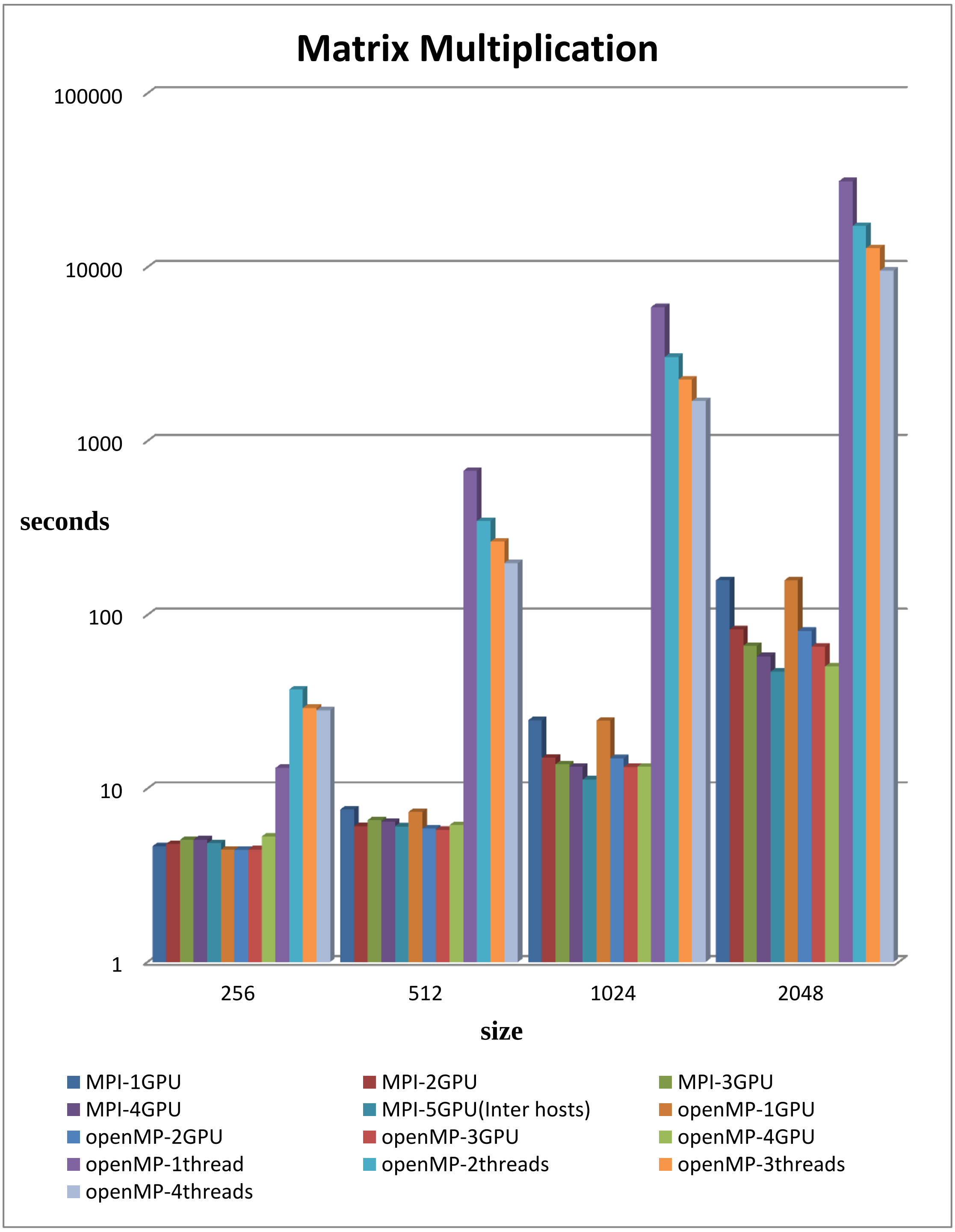
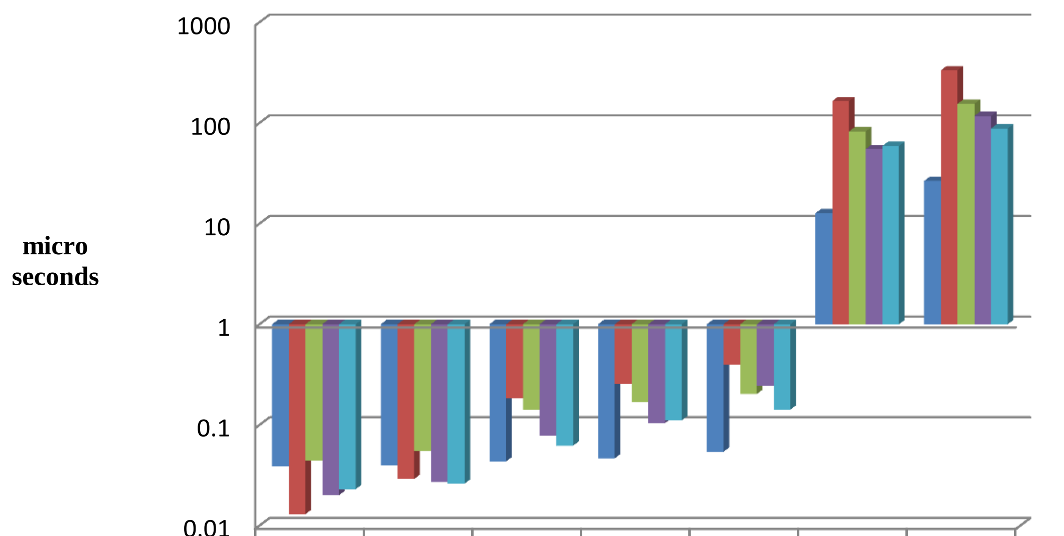


Figure 5-19. Matrix multiplication with problem sizes from 256 to 2048

Figure 5-19 shows that the performance of GPU on processing the massively parallel execution as the application of Matrix Multiplication form 256 to 2048. In this case, the execution results on MPI and OpenMP upon GPU are close. Comparing to the performance between GPU and CPU with this instance, the performance of

GPU obviously exceeds CPU. With the small problem size such as 256 by 256 Matrix Multiplication; the speedup of performance is negligible. The degree of speedup accumulates with the increasing of the problem size.



	10words	100words	1000 words	1500 words	2000 words	1000000 words	2098651words
GPU	0.0389	0.0394	0.0434	0.0466	0.0541	12.6773	26.4995
openMP-1thread	0.013	0.029	0.183	0.255	0.394	164.385	332.687
openMP-2threads	0.044	0.055	0.141	0.168	0.203	82.135	154.648
openMP-3threads	0.02	0.027	0.078	0.104	0.244	54.916	116.991
openMP-4threads	0.023	0.026	0.062	0.111	0.142	58.953	87.909

Figure 5-20. MD5 hashing on 10 to 2,098,651 words

Also, **Figure 5-20** reveals that single GPU presents better performance than single CPU with multiple threads on MD5 hashing computation. Again, the performance of GPU could not be observed in the small problem size due to the constraint on the internal overhead of starting execution.

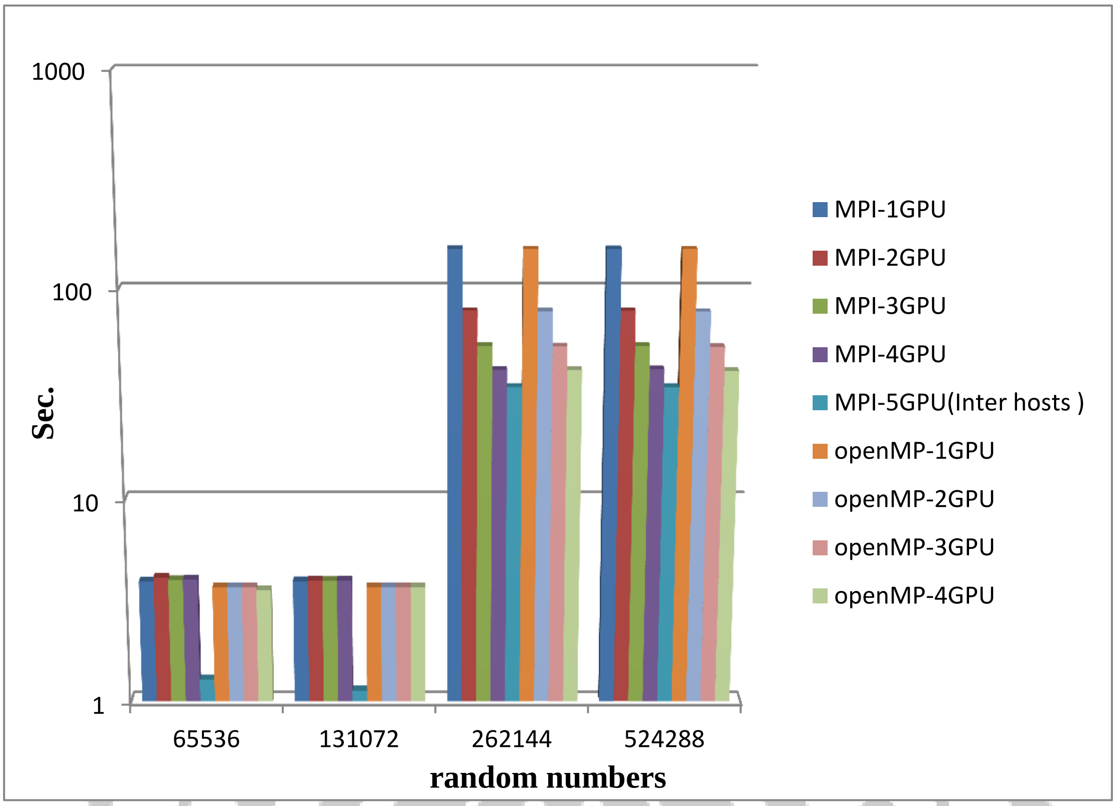


Figure 5-21. Sorting numbers 640 times from 65,536 to 524288 floating point numbers

Finally, **Figure 5-21** shows that the comparison of performance on multiple GPU with MPI and OpenMP. The results of MPI and OpenMP are approximate to each other.

Chapter 6

Conclusions and Future Work

6.1 Concluding Remark

In conclusion, we discovered some of the automatic parallel tools which could translate the sequential codes to the parallel code to reduce our time on rewrite codes for parallel processing on multicore system. And we verified the available of these tools, and then we implement an interface for ROSE to simplify the complexity of use. From our experiment, we know that through these auto-parallel tools almost could both help us easily transform our non-parallel codes to parallel codes and run on multicore system. After comparing, we make a table to show what tool is best in each environment in **Table 6-1**. The perfect auto-parallelizing compiler is yet to be produced. However, there are some cases where auto-parallelization is perfectly suited.

Table 6-1. The best tool in each environment

program / environment	matrix	Nbody	Jacobi
CPU	Intel+ROSE	X	PGI+ROSE
GPU	PGI	PGI	PGI
embedded system	ROSE	X	ROSE

And we propose a parallel programming approach using hybrid CUDA and MPI programming, which partition loop iterations according to the number of C1060 GPU nodes in a GPU cluster which consists of one C1060 and one S1070. During the experiment, loop iterations assigned to one MPI process and processed in parallel by CUDA run by the processor cores in the same computational node. The experiment reveals that the hybrid parallel multicore GPU currently processing with OpenMP and MPI as a powerful approach of composing high performance clusters.

6.2 Future Work

In the future, we may add more experiments especially on GPU and embedded system in the aspect of automatic parallelism. And we will compare the difference between parallelism of automatic and manual. In the aspect of hybrid parallel programming, we think that the algorithm of dispatch task can still be improved. We may accord the computing power of machine to allocate.

Bibliography

[1] C.T. Yang, C.L. Huang and C.F. Lin, "Hybrid CUDA, OpenMP, and MPI Parallel Programming on Multicore GPU Clusters", *Computer Physics Communications*, Vol. 182, Issue 1, pp. 266-269, June 25, 2010.

[2] C.T. Yang, C.L. Huang, C.F. Lin and T.C. Chang, "Hybrid Parallel Programming on GPU Clusters", *International Symposium on Parallel and Distributed Processing with Applications (ISPA) 2010*, pp. 142-147, Sept. 2010.

[3] P. Alonso, R. Cortina, F.J. Martinez-Zaldivar and J. Ranilla, "Neville elimination on multi- and many-core systems: OpenMP, MPI and CUDA", *J. Supercomputing, in press*, doi:10.1007/s11227-009-0360-z, SpringerLink Online Date: Nov. 18, 2009.

[4] F. Bodin and S. Bihan, "Heterogeneous multicore parallel programming for graphics processing units", *Scientific Programming*, Vol. 17, pp. 325-336, 4 Nov. 2009.

[5] R. Dolbeau, S. Bihan and F. Bodin, "HMPP: A hybrid multicore parallel programming environment", *The Proceedings of the Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, Boston, Massachussets, USA, October 4th, 2007

[6] D. Goddeke, R. Strzodka, J. Mohd-Yusof, P. McCormick, S. Buijssen, M. Grajewski, S. Tureka, "Exploring weak scalability for FEM calculations on a GPU-enhanced cluster", *Parallel Computing*, Vol. 33, Issue 10-11, pp. 685–699, 33 Nov. 2007.

[7] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, K. Skadron, "A performance study of general-purpose applications on graphics processors using CUDA", *Journal of Parallel and Distributed Computing*, Volume 68, Issue 10,

pp. 1370-1380, October 2008

[8] C.H. Liao, D. Quinlan, T. Panas and Bronis de Supinski, “A ROSE-based OpenMP 3.0 Research Compiler Supporting Multiple Runtime Libraries”, *International Workshop on OpenMP (IWOMP) 2010*, pp.15-28, accepted in March. 2010

[9] C.H. Liao, D. Quinlan, J. Willcock and T. Panas, “Semantic-Aware Automatic Parallelization of Modern Applications Using High-Level Abstractions”, *International Journal of Parallel Programming*, Vol. 38, No. 5-6, pp. 361-378, Accepted in Jan. 2010

[10] C.H. Liao, D. Quinlan, T. Panas and Bronis de Supinski, “Towards an Abstraction-Friendly Programming Model for High Productivity and High Performance Computing”, *Los Alamos Computer Science Symposium (LACSS) 2009*,

[11] C.H. Liao, D. Quinlan, R. Vuduc and T. Panas, “Effective Source-to-Source Outlining to Support Whole Program Empirical Optimization”, *In Proceedings of LCPC'2009*, pp.308-322, 2009

[12] A. Saebjornsen, J. Willcock, T. Panas, D. Quinlan and Z. Su, “Detecting code clones in binary executables”, *ISSTA '09 Proceedings of the eighteenth international symposium on Software testing and analysis*, pp. 117-127, 2009

[13] T. Panas and D. Quinlan, “Techniques for software quality analysis of binaries: applied to Windows and Linux”, *DEFECTS '09 Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009)*, pp.6-10, 2009

[14] C.H. Liao, D. Quinlan, J. Willcock and T. Panas, “Extending Automatic

Parallelization to Optimize High-Level Abstractions for Multicore", *IWOMP '09 Proceedings of the 5th International Workshop on OpenMP: Evolving OpenMP in an Age of Extreme Parallelism*, pp.28-41, 2009

[15] P. Carribault, M. P´erache and H. Jourdren, "Enabling Low-Overhead Hybrid MPI/OpenMP Parallelism with MPC", *International Workshop on OpenMP (IWOMP) 2010*, pp.1-14, 2010

[16] M.M. Baskaran, J. Ramanujam and P. Sadayappan, "Automatic C-to-CUDA Code Generation for Affine Programs", *In Compiler Construction*, Vol. 6011, pp. 244-263, 2010

[17] S. Gupta and M.R. Babu, "Generating Performance Analysis of GPU compared to Singlecore and Multi-core CPU for Natural Language Applications", *International Journal of Advanced Computer Sciences and Applications*, Vol. 2, Issue 5, pp. 50-53, 2011

[18] S. Rivoire and R. Park, "A breadth-first course in multicore and manycore programming", *SIGCSE '10 Proceedings of the 41st ACM technical symposium on Computer science education*, pp.214-218, 2010

[19] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. Stone, D. B. Kirk and W.M. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded GPU using CUDA", *PPoPP '08 Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pp. 73-82, 2008

[20] M.M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus", *ICS '08 Proceedings of the 22nd annual international conference on Supercomputing*, pp. 225-234, 2008

[21] S. Kamil, C.Y. Chan, L. Oliker, J. Shalf and S. Williams, “An auto-tuning framework for parallel multicore stencil computations”, *Parallel & Distributed Processing (IPDPS) 2010*, pp. 1-12, April 2010

[22] T.P. Chen and Y.K. Chen, “Challenges and opportunities of obtaining performance from multi-core CPUs and many-core GPUs”, *ICASSP '09 Proceedings of the 2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 613-616, April 2009

[23] Top 500 Sites for November 2010, http://www.top500.org/lists/2010/11

[24] Top 500 Super Computer Sites, What is Gflop/s, http://www.top500.org/faq/what_gflop_s.

[25] Close To Metal wiki, http://en.wikipedia.org/wiki/Close_to_Metal.

[26] OpenCL, http://www.khronos.org/opencl/.

[27] CUDA, http://en.wikipedia.org/wiki/CUDA.

[28] Download CUDA, http://developer.nvidia.com/cuda-downloads.

[29] MPI, http://www.mcs.anl.gov/research/projects/mpi/.

[30] Intel 64 Tesla Linux Cluster Lincoln webpage, http://www.ncsa.illinois.edu/UserInfo/Resources/Hardware/Intel64TeslaCluster/

[31] MPICH, A Portable Implementation of MPI, http://www.mcs.anl.gov/research/projects/mpi/mpich1/index.htm.

[32] Open MP Specification, http://openmp.org/wp/about-openmp/.

[33] POSIX Threads Programming, https://computing.llnl.gov/tutorials/pthreads/.

[34] Intel® Threading Building Blocks, http://www.threadingbuildingblocks.org/.

[35] Open64, http://www.open64.net/.

[36] Intel, http://software.intel.com/en-us/articles/intel-parallel-studio-xe/.

[37] The Potland Group, http://www.pgroup.com/index.htm.

[38] PAR4ALL, http://www.par4all.org/.

[39] Specification Tesla S1070 GPU Computing System, http://www.nvidia.com/docs/IO/43395/SP-04154-001_v02.pdf.

[40] The NVIDIA® Tesla™ S1070 Computing System, http://www.nvidia.com/object/product_tesla_s1070_us.html.

[41] NVIDIA Tesla C1060 Computing Processor, http://www.nvidia.com/object/product_tesla_c1060_us.html.

[42] NVIDIA CUDA Programming Guide, http://developer.download.nvidia.com/compute/cuda/2_3/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.3.pdf.

[43] Arm11MP Core http://www.arm.com/products/processors/classic/arm11/arm11-mpcore.php.

[44] The CUDA Compiler Driver NVCC, http://moss.csc.ncsu.edu/~mueller/cluster/nvidia/2.0/nvcc_2.0.pdf.

[45] Intel® Xeon® Processor E5410, http://ark.intel.com/Product.aspx?id=33080

[46] Benchmarcks,
http://shootout.alioth.debian.org/u32q/benchmark.php?test=nbody&lang=gcc.

[47] Cross compiler, http://en.wikipedia.org/wiki/Cross_compiler.

[48] CodeSourcery, http://www.codesourcery.com/.

[49] crosstool-NG, http://linux.softpedia.com/get/System/Shells/crosstool-NG-28833.shtml.

[50] crosstool-ng WIKI, http://ymorin.is-a-geek.org/dokuwiki/projects/crosstool.

[51] How to build cross toolchains for ARM crosstool-NG, http://forum.samdroid.net/wiki/showwiki/How+to+build+cross+toolchains+for+ARM+crosstool-NG.

[52] To build crosscompiler by crosstool-ng http://hi.baidu.com/caicry/blog/item/f306db639c4281680c33fa1b.html.

[53] To build crosscompiler by crosstool-ng, http://blog.chinaunix.net/u3/95743/showart_2067287.html.

[54] Intel® Xeon® Processor E5520, http://ark.intel.com/Product.aspx?id=40200

Appendix

A. Setup of auto-parallel tool

1 ROSE

Environment: Ubuntu 9.10

Before install ROSE, following as command

I. `$ sudo aptget install libboost1.4-*`

II. `$ sudo aptget install sun-jdk-6-sun`

The steps of installation as following:

I. Download package from https://outreach.scidac.gov/projects/rose/

II. Untar package

```
$ tar -zxfv rose-0.9.5a-without-EDG-15163.tar.gz
```

III. Run the configure script

```
$ ./configure -with-java={path of JAVA}
```

IV. Run make

```
$ make
```

V. To install ROSE, type make install

```
$ make install
```

VI. Set PATH

2 Par4All

The steps of installation as following:

I. Download package from http://www.par4all.org/download/

II. Untar package

```
$ tar -zxfv <package>.tar.gz
```

III. It will create a directory named par4all. Move this directory to its final location:

```
$ sudo mv par4all /usr/local
```

IV. In any case, you will then need to source one of the following shell scripts which set up the environment variables for proper Par4All execution:

If you use bash, sh, dash, etc…

```
$ source /usr/local/par4all/etc/par4all-rc.sh
```

If you use csh, tcsh, etc…

```
$ source /usr/local/par4all/etc/par4all-rc.csh
```

3 Intel® Composer XE 2011 for Linux

The steps of installation as following:

I. Download package from http://software.intel.com/en-us/articles/intel-software-evaluation-center/

II. Untar package

```
$ tar -zxfv <package>.tgz
```

III. Run the install script

```
$ sh install.sh
```

IV. Set PATH

4 PGI Accelerator C/C++ Workstation 10.9

The steps of installation as following:

I. Download package from http://www.pgroup.com/support/downloads.php

II. Untar package

```
$ tar -zxfv <package>.tar.gz
```

III. Run the install script

```
$ sh install.sh
```

IV. Set PATH

5 Open64 compiler 4.2.3

The steps of installation as following:

I. Download package from
http://www.open64.net/download/open64-4x-releases.html

II. Untar package

```
$ tar jxfv <package>.tar.bz2
```

III. Set PATH

B. Interface

1 VMC-PPO

I. Get the package of VMC_PPO_GUI.zip

II. Unzip the package

```
$ unzip VMC_PPO_GUI.zip
```

III. Execute the program

```
$ ./VMC_PPO_GUI.sln
```

2 Plug-in of Eclipse

I. Get the package of vmcppo.zip

II. Copy the package to the directory of Eclipse

```
$ cd /usr/lib/eclipse
```

III. Unzip the package and restart Eclipse

```
$ unzip VMC_PPO_GUI.zip
```