Parallelizing a 2D advection solver on both CPU and GPU

Huang Zhiyuan

A report submitted for the course
COMP8755 Individual Project
Supervised by: Peter Strazdins
The Australian National University

November 2020

Except where otherwise indicated, this report is my own original work.

Huang Zhiyuan
5 November 2020

Acknowledgments

First, I would like to thank my supervisor, Peter Strazdins, for his patience and kindness to support my study throughout this year. I didn't study computer science before I came to the ANU and I didn't even know how to use Linux at the beginning of this project. I have improved on many aspects throughout this project.

Second, I would like to thank my friends Jiang Siyu. He gave me a lot of help not only on this project, but also in the entire study at ANU. He also helped me a lot in the daily life. He is the best friend I have had during the study at ANU.

Next, I want to express my gratitude to my friend Wei Songyan, Alex Pan, Li Shengqi, Shi Mingjie and friends in the Bridge and Bruce Hall. Your friendship made my life at ANU happier.

Finally, I appreciate my family for their unbounded love and unconditional support. Especially during this pandemic, I never felt stressed or intense because their support.

Abstract

We parallelized a 2D advection solver with both CPU parallel programming (OpenMP) and GPU parallel programming (CUDA). We significantly improved the performance by parallelization and then tuned the parameters to find the optimal implementation. We also investigated the properties of different parallel models.

OpenMP is easy to implement and could effectively improve the performance up to 15$\times$ speed-ups. The performance improves with the thread number increases until thread number reaches the limit of the CPU. However, programmers don't have much control over behaviours of each thread and memory management, which causes further optimization troublesome.

CUDA programming is capable of gaining more than 35 $\times$ speed-ups than the original serial program. However, CUDA configurations and implementations are more complicated than OpenMP implementations. Only a minor change in the code may lead to significant changes in the performance. In this thesis, we designed four different implementations and analyzed their performance. Then we chose the best implementation and tuned their parameters to find the optimal configurations. We have shown that parameters such as thread dimensions and iteration orders may have significant impact to the final results. Therefore, it is important for programmers to understand their properties so that they can write efficient parallel programs.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

As the clock rate speedups and transistors number improvement on a single chip gradually slowed down, the industry retreated towards having multiple lower frequency and simpler cores on one chip. It is natural for programmers to start thinking about exploiting the parallelism to optimize the programs for better performance. Stencil computations are widely used in many scientific applications such as computational fluid dynamics, image processing, solving partial differential equations (PDE). It is a good candidate to perform parallel computing because of all of the elements are calculated independently. In this thesis, we use two parallel programming models – OpenMP and CUDA to parallelize stencil computation program. We then investigate their properties and discuss their pros and cons.

1.1 Motivation

Processors are expected to double their transistors every two year according to Moore's law. Recently, however, this trend seems to slow down, and so as the clock rate. Therefore, industry retreated towards having multiple lower frequency and simpler cores on one chip. Theoretically, doubling the cores provides the chip with the capability of performing twice the number of functions in parallel. However, the actual performance is far away from ideal. There are multiple reasons. First, some of the key components don't such as issue logic and cache memories don't scale linearly with the number of cores. Moreover, many cores on same chip will cause overheating problem, which will significantly impact their performance.

In order to take full advantage of the multi-core processing units, programmers should know their properties thus to leverage the increasing number of processors to write high performance parallel programs. Both CPU and GPU are capable of executing programs in parallel. However, their performance is quite different due to their unique architectures. CPU is designed to improve program latency whereas GPU focuses on throughput. Therefore, it is important to understand their properties so that programmers could choose suitable devices when parallelizing a program.

Stencil computations have been employed by a large fraction of scientific applications such as fluid dynamics, electromagnetics and partial differential solvers. In a stencil operation, each element in a data array is updated with a weighted contribu-

tions from its neighbours with a fixed pattern, called stencil. Since all the elements in each step are calculated independently, stencil computations are a great candidate for parallel programming. In this thesis, we parallelize a robust stencil computation program, which is originally designed to "make a 2D advection solver tolerant to soft faults" [Peter Strazdins and Armstrong, 2016], with both CPU parallel programming and GPU parallel programming to optimize its performance. Then we use various parameters and implementations to study their properties. We hope our work will help to better understand different parallel models and their characteristics.

1.1.1 Contributions

This thesis made the following contributions:

- Parallelized the stencil computation program with OpenMP and investigated the impact of different number of thread to the program performance.
- Parallelized the stencil computation program with CUDA using four different implementations and briefly analyzed the results.
- Tuned the parameters of CUDA parallelization and found out the factors that could affect the program performance.

1.2 Thesis Outline

The following is an outline of the thesis:

Chapter 2 gives an overall introduction of parallel programming, including both CPU and GPU architecture/parallel models as well as stencil computations.

Chapter 3 describes how we designed our experiments and how we implement them. We also elaborate implementation details in each program.

Chapter 4 provides details about the software environment and hardware platform used in this thesis.

Chapter 5 evaluates different parallel implementations and discusses their properties. Then we compares the results from OpenMP and CUDA parallelization.

Chapter 6 concludes the results and provides future working extensions about this thesis.

Chapter 2

Background and Related Work

Parallel systems, including multicore CPUs and many-core GPUs models have drawn lots of attention in computer science and smart device applications due to their tremendous potential in performance and efficiency improvement [K. Asanovic and Yelick, 2009]. However, it's still challenging for programmers to fully make use of the increasing number of processor cores. The reasons are various, including both hardware level and software level. In this chapter, we will introduce several techniques that could help developers take advantage of parallel systems. Then, we will discuss their pros and cons and how they are related to this report.

2.1 Parallel computing

Parallel computing is a simple concept that multiple tasks or calculations are running simultaneously. To be able to execute a program in parallel, a program often need to be divided into smaller tasks that are independent of each other. These small tasks are running at the same time until all of them are finished, after that, the program goes to the next step.

There are various models of parallel computing, which can be categorized according to "the level at which the hardware supports parallelism" [Yongpeng Zhang, 2012]. All of these models need both hardware and software work together to effectively utilize the parallel machines. The classes of parallel computer architecture include [Holewinski et al., 2012]:

- Multicore computing: A multicore processor consists of two or more independent processing cores that could executes program instruction separately. If all of these cores are the same, this multicore system is homogeneous. If the multicore system contains different types of cores, this system is heterogeneous.
- Symmetric multiprocessing: Symmetric multiprocessing system has two or more independent, homogeneous processors with a shared main memory connected to all cores, which means all cores share same resources and memories, and they are treated equally. Each core has a private cache memory that are connected using on-chip mesh networks.

- Distributed computing: Distributed computing system is composed of multiple computers connected by HTTP, RPC-like connectors. This system components are all controlled by a central computer to coordinate their actions. Because all of the components are independent, a single component fails couldn't affect the whole system.
- Massively parallel computing: Massively parallel computing uses numerous multicore computers to run in parallel to execute a set of computations. An example is called grid computing, in which computers in different area in the world, communicated via Internet, work together to solve a particular problem.

Among various parallel computing architectures, multicore processor systems are undergoing significant growth "as exploitable instruction-level parallelism is limited and the processor clock frequency cannot be increased any further due to power consumption and heat problems" [Romein, 2012]. In order to solve these problems, engineers and manufactures need to design parallel systems with more efficient processors and less heat generation.

Multiprocessors are gradually replacing traditional, single-core computers due to its tremendous potential for performance and efficiency improvement, and the heterogeneous multicore systems are becoming more common.

2.1.1 Parallel programming models

Flynn' taxonomy classifies computing models into 4 categories based on the resulting data and the control flow of the system [Ikei and Sato, 2014].

- Single-Instruction, Single-Data (SISD): In this model, only one processing unit has access to the program and data storage. This process loads data from memory and executes instructions to get the final result. The result is then stored back to the memory. This is a conventional sequential computing model.
- Multiple-Instruction, Single-Data (MISD): There are multiple processing units are working together, each of which has a private memory. However, only one single global memory is provided in this model. Unfortunately, on commercial computer of this type has been built because of its restrictions.
- Single-Instruction, Multiple-Data (SIMD): There are multiple processing units processing different data using same instructions. Each unit has a private access to memory so that they have different data. These processing units are working synchronously in parallel.
- Multiple-Instruction, Multiple-Data (MIMD): There are multiple processing units working on different instructions with unique data. In this case, every unit is running independently and asynchronously with each other.

MIMD model is the most common parallel model, nearly all general-purpose parallel systems are designed based on it. It may have different memory organization,

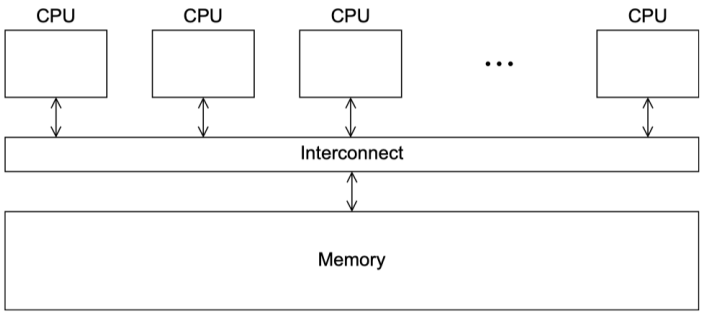


Figure 2.1: A shared-memory system

either a shared memory or distributed memory. A distributed memory organization consists of multiple processing units and their memories are independent. These independent memories are connected by an interconnected network which supports data transfer. In a shared memory organization, all processing units are connected to a global memory. Data is exchanged between processors through the global memory modules [Sano et al., 2011].

2.1.2 Shared-memory programming with OpenMP

OpenMP is a shared-memory parallel programming API in which every thread or process has access to main memory, as in Figure 2.1. Unlike Pthreads and MPI parallel models which require programmer to explicitly specify the behaviours of each thread, OpenMp could do all the complications for the programmers [Luporini et al., 2020]. The programmers only need to specify which part of the code need to be executed in parallel. Moreover, it allows programmers to incrementally parallelize serial programs, which will simplify the program parallelization procedure.

At the beginning of the program, OpenMp creates a master thread that executes instructions consecutively. After the master thread meets the parallel part of the program, it forks a certain number of slave threads that run simultaneously, and the system divides tasks and allocate to each of the thread [Zohouri et al., 2018]. Since each thread is running independently, both task parallelism and data parallelism are feasible.

OpenMP uses directives to tell the runtime environment how to execute the program. Directives are typically added to the source code as a comment to specify which loops were to be parallelized [Soejima et al., 2014]. It will be ignored if you use basic C compiler. The directives serve various purposes in the program, including spawning parallel region, scheduling tasks among threads, allocating loop execution between threads, and so on. The directives are simple and straightforward. Programmers can use these directives to effectively parallelize the programs.

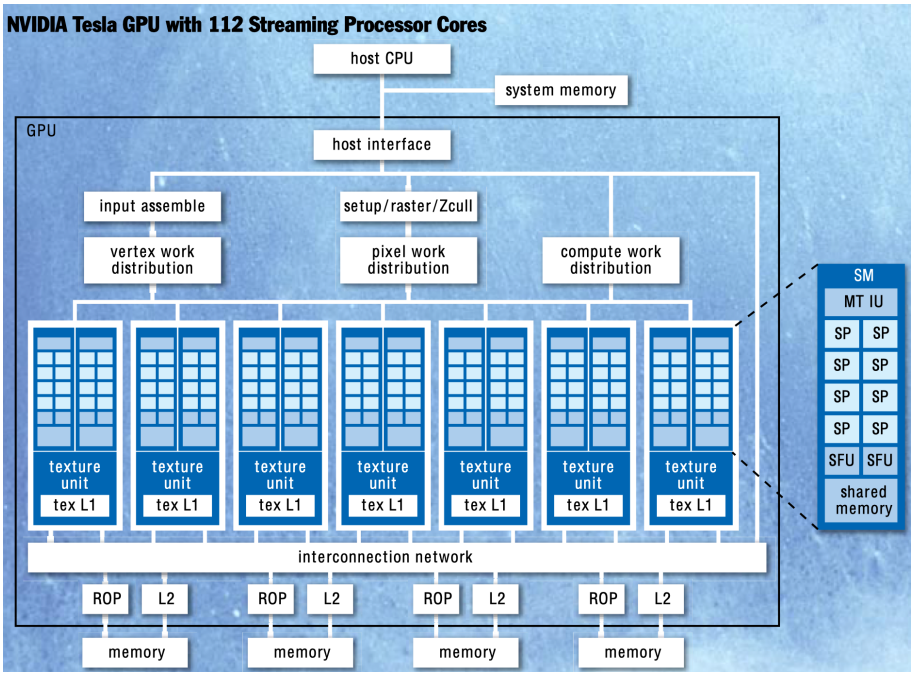


Figure 2.2: NVIDIA Tesla GPU architecture [Nickolls et al., 2008]

2.2 The GPU computing architecture

A graphics processing unit (GPU) is a electronic circuit originally designed to speed up game graphics. However, due to its unique architecture and large amount of processor cores, it shows tremendous advantages in throughput-oriented programs. Since then, GPU gradually evolved into a powerful programmable processor. Nowadays, GPUs have been widely used in various area such as high performance computing and artificial intelligence [Lee et al., 2019]. A modern CPU may only has at most 32 cores, whereas GPU may have up to 5000 cores. This amount of cores make GPUs extremely good at large scale computations. Besides, GPU focuses more on program's throughput rather than latency, which, to some extent, could complement the CPU.

In 2006, NVIDIA released Tesla unified graphics and computing architecture. This architecture makes GPU more powerful parallelization device because of its massively mutithreaded process array. It offers not only graphics computing, but also highly efficient parallel computing. Multithreaded streaming multiprocessors (SMs) are the building blocks for Tesla architecture. Figure 2.2 shows the Tesla architecture with 14 SMs – a total of 112 SP (streaming processor) cores , and they are all connected to 4 external DRAM partitions. The SMs use SIMT (single-instruction, multiple-thread) model to manage hundreds of threads including memory transfer and task scheduling. Every thread in a SP core has private registers and different instruction address, thus, each thread executes independently. Every SM has a control unit to schedule tasks and execute threads in groups of 32 threads called warps. All threads

in a warp start from same program address but execute independently. A SM contains 24 warps, and each warp has 32 threads.

When a program starts, the SM selects a warp to execute the instructions. All threads in a warp share the same instructions and they execute instructions at the same time. If programmers want to obtain full efficiency, they must make sure that threads in a warp do not diverge via data-dependent conditional branch. If threads in a warp diverge, the warp serially disable the threads on the different path. All threads converge back after all threads in a warp complete. This only happens inside a warp. Different warps execute independently [Bucur, 2014].

2.3 CUDA programming

CUDA (Compute Unified Device Architecture), developed by NVIDIA company, is a parallel programming model to enable general-purpose computing on its own GPU. Programmers can use CUDA to directly operate GPU to speed up compute-intensive applications.

2.3.1 CUDA paradigm

CUDA is designed to work with C and C++ programming languages. A typical CUDA program is launched by CPU and then move to GPU, therefore, it has code running on CPU and code running on GPU. The code running on CPU is the same as normal C-base programs, the code running on GPU is called kernel. The kernel executes by numerous threads on GPU, which are organized into a hierarchy of grids and blocks. All threads that are launched in the program form a thread grid, it can be divided into multiple thread blocks with each block has multiple thread in it. All threads in a block have access to a same on-chip memory called local memory, and they can cooperate themselves through barrier synchronization [Buck, 2007].

The programmer need to choose a proper number of thread blocks and threads per block to make the kernel run correctly and efficiently. Each thread in a block has a unique thread ID number in that particular block **threadIdx** and each thread block has a unique block ID **blockIdx**. Progrmmers identify threads based on their IDs. A thread block contains at most 512 threads.

The CUDA kernel is a common C function with a **__global__** specifier. The program launches kernel with three square brackets containing the thread dimensions. The syntax is

kernel«<dimGrid, dimBlock»> (...parameters...);

where dimGrid is the dimension of the grid and dimBlock is the dimension of blocks.

Thread management and parallel execution are automatic [6]. Programmers only need to specify the dimensions of the thread and the underlying system will create thread, schedule, and allocate tasks for the programmers. The programmers could synchronize the all threads in a thread block by calling the function **__syncthreads()**.

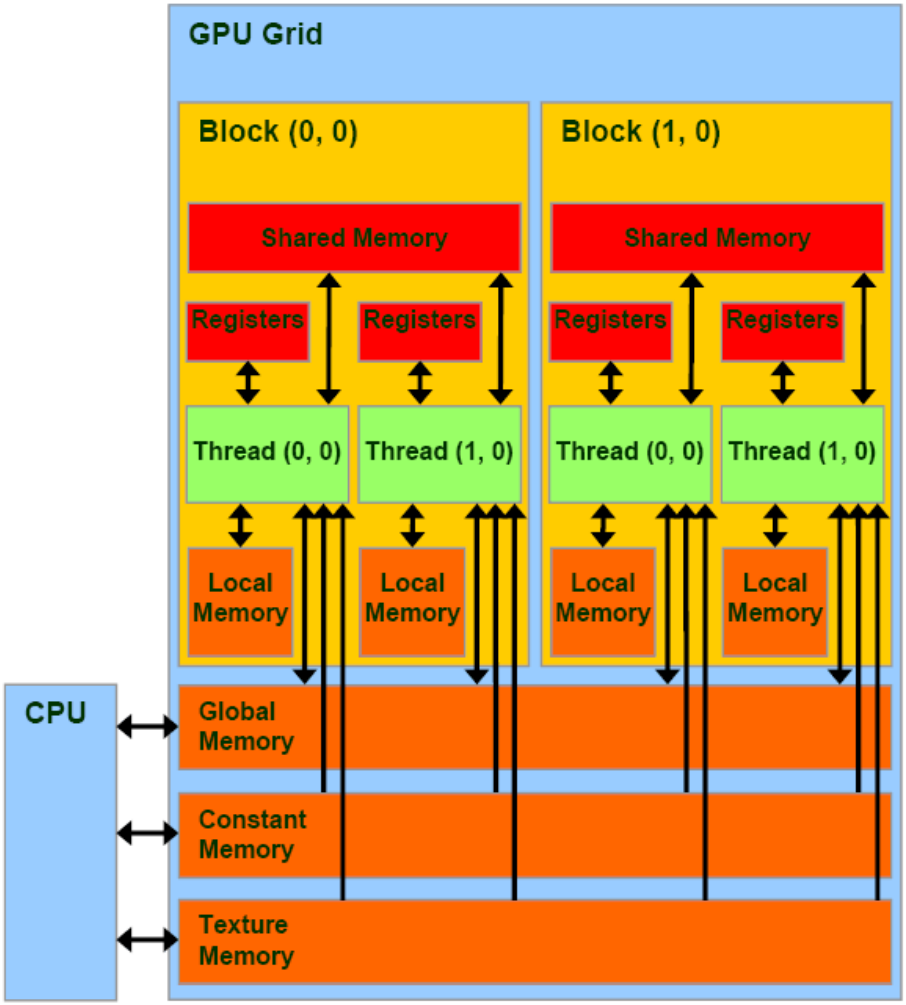


Figure 2.3: Memory structure of GPU [Melchor et al., 2008]

This barrier guarantees that "all threads in the block will not proceed until all participating threads reached the barrier" [Gibbons, 2014].

Since threads in a block may have a shared memory space and a synchronization barrier, therefore, according to the memory structure of GPU, all threads in a block are on the same SM or SP. However, each SM or SP may have 0 or multiple blocks and the programmers don't have to know how the blocks are distributed. The GPU will allocate threads properly and give a block ID to programmers. This virtualization offers programmers the flexibility to choose whatever granularity is convenient for coding because programmers could choose thread dimensions according to data size rather than the number of processors. This also allows the same CUDA program could run on different type of GPUs without modification.

2.3.2 CUDA memory management

There are multiple memory spaces for threads to fetch data during execution. Each thread has its own registers and private local memory. Threads in a block also share a local memory space. Finally, all threads have access to the same global memory. In CUDA programs, qualifiers **__shared__** and **__global__** are used to specify where the variables should be stored at. If a variable was declared in a kernel without qualifiers, then this variable is stored in the private memory, either in registers or in the local memory. The shared memory is on the chip, whereas the global memory is on the graphics board.

Shared memory is on-chip RAM close to processor like L1 cache. It offers low-latency data sharing and communication inside a block. Thread blocks communicate with global memory. Figure 2.3 shows a virtual memory structure of threads, blocks and the grid. It illustrates the relationship between memories and threads as described above.

From Figure 2.3 we can see that only global memory on GPU is connected to CPU and CPU is not allowed to declare variables directly on GPU. Therefore, programmers have to allocate a memory space on GPU and then transfer data from CPU to GPU. After execution, the data need to be transferred back to CPU memory and the memory space allocated on GPU should be freed.

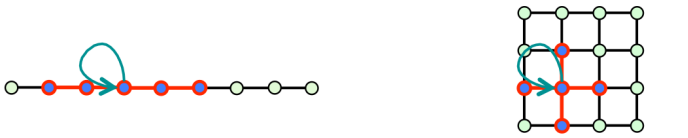
2.3.3 CUDA unified memory

CPU memory and GPU memory are physically separated according to Figure 2.3. Hence, data need to be explicitly copied between CPU memory and GPU memory, which is troublesome and time consuming. In 2014, NVIDIA developed a new memory management technology – unified memory to simplify memory management between CPU and GPU. Unified memory could create a virtual memory pool that is shared by GPU and CPU. Both of them could access the memory by a single pointer, which will simplify the program implementation. While in the actual, the system still transfer data in this unified memory between CPU memory and GPU memory. Hence, it looks like programmers can declare variables directly in the global memory on GPU but it isn't what actually happen in the device.

Unified memory technique is an extremely simple method for a CUDA program to manage memory. Instead of allocate memory on CPU and copy data back and forth, unified memory provides an API **cudaMallocManaged(void* pointer, size_t size)** to allocate memory on the unified memory space so that both CPU and GPU have access to it. This technique hides all complexity and makes memory management much easier.

2.3.4 CUDA restrictions

It's important to understand its restrictions in order to take advantage of its parallel architecture and avoid its drawbacks. From the description above we have known that threads in a block could synchronize by a barrier. However, different thread blocks



(a) 1D 5-Point Stencil **(b)** 2D 5-Point Stencil

Figure 2.4: Two 5-point stencils of 1 dimension and 2 dimension [Datta and Yelick, 2009]

couldn't synchronize [Stout and Jablonowski, 2006]. If programmer wants to combine results from different blocks, he has to launch a second kernel to synchronize all thread blocks to ensure that all threads have completed their jobs.

Recursion is not allowed in a CUDA kernel since it often needs large stack space. If all threads are running a recursive call, the required memory would be huge.

2.3.5 CUDA summary

CUDA is a parallel programming model that allows the programmers use GPU to develop scalable parallel applications. Its easily understood abstractions and straightforward implementation help developers focus on algorithmic efficiency. The programming paradigm allows programmers to harness the parallel architecture of GPU and greatly speed up variety of sophisticated applications.

Since CUDA was first released in 2007, it has attracted tremendous developers, engineers and scientists across the world working on it. Nowadays, many of state-of-the-art computer science technologies including artificial intelligence, big data and computer vision need CUDA to accelerate their applications, demonstrating CUDA a promising future.

2.4 Stencil computation

Stencil computation is a numerical data processing model that the value of the element depends on a fixed pattern, called a stencil. It is widely used in various applications such as fluid dynamics computation, image processing and solving partial differential equations [Luporini et al., 2020].

Stencil computation performs a sequence of sweeps (time steps) of a given data structure which is usually an array. In each sweep, the program iterate through all elements in the array and update its value based on a stencil. Figure 2.4 is a visualization of two stencils with different dimensions. From this figure we can see that the value of the element where the arrow comes from is calculated based on the 5 red neighbour elements including itself. This 5-point pattern is a stencil.

There are three common stencil iteration types: Jacobi, Gauss-Seidel, and Gauss-Seidel Red-Black iterations as shown in Figure 2.5 [Hoffman and Hargrove, 1999]. In a Jacobi iteration, there are two separate grid. The program read values from the

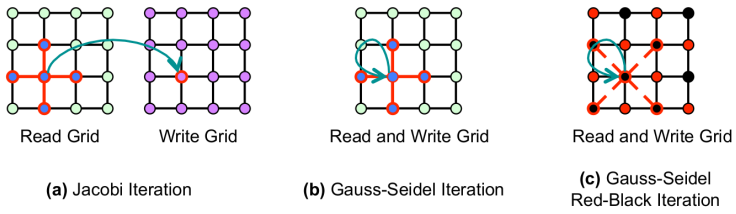


Figure 2.5: Three common stencil iteration types [Datta and Yelick, 2009]

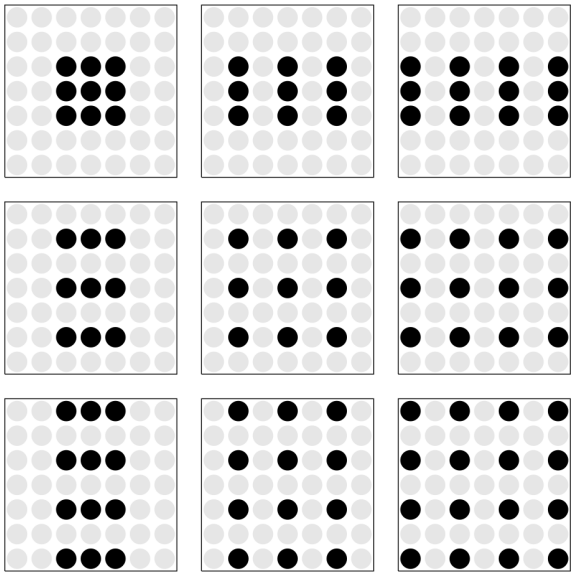


Figure 2.6: A depiction of the nodes used in each stencil. From left to right, top to bottom the stencils depicted are NN, WN, FN, NW, WW, FW, NF, WF and FF [Peter Strazdins and Armstrong, 2016]

original grid, then write the results to the corresponding elements in the other grid. The Gauss-Seidel iteration is in-place, the results are written back to the original grid. The Gauss-Seidel Red-Black iterations are also in-place while calculation occurs every other point. As shown in the Figure 2.5(c), red points are updated first and then black points.

Stencil computation is a good candidate to perform parallel programming since elements in each time step is calculated independently. Therefore, elements can be updated in parallel. However, the results of parallelizing these computations are not satisfied because it is still far away from the theoretical best performance. The main reason is that data transfer is not fast enough to avoid stalling the processors [Ikei and Sato, 2014]. Therefore, it is important to organize the stencil computations to facilitate data transfer so that stencil computation programs could take full advantage of parallelism. Researchers have been working on this for years and a promising solution so far is called tiling optimization. The idea of tiling optimization is to perform calculation in cache-sized blocks of data to improve memory locality.

In summary, stencil computation is a good candidate for parallel programming and there is still room for improvement.

2.5 Advection solver

In this thesis, we parallelize a 2D advection solver which is originally designed to solve partial differential equations. This solver uses a novel technique called robust stencil to make it tolerant to soft faults. A traditional method to solve this problem is called Triple Modular Redundancy (TMR), which involves periodic comparison of the results at each step. TMR uses a voting scheme, in which the results that get majority votes will be determined as correct value. TMR is simple, general and extremely robust, however, it requires extra memory and computations. Robust stencil technique is much more efficient because it can be implemented with negligible memory overhead compared to TMR. Thus, our implementation significantly improve the performance.

In this program, we use several different stencils to solve the PDEs as shown in Figure 2.6. We also have combined stencils, which combined several stencils as one stencil such as C30, C50 and C70. Here, we parallelize all stencils and investigate their performance. They all show very good improvement compared to TMR and maintain robustness at the same time. We believe this technique could be generalized to other PDE solvers [Peter Strazdins and Armstrong, 2016].

Chapter 3

Design and Implementation

We developed our parallel programming application based on a 2D Advection Solver, which was originally designed to solve Partial Differential Equations[Peter Strazdins and Armstrong, 2016]. We optimized this program with both OpenMP and CUDA. Then we investigated the performance of these two parallel programming models and discussed their properties.

3.1 Program analysis

The structure of this program can be divided into three parts. The first step is to initialize variables and data structures that will be used in the following computation. Then the program iterate several times. In each iteration, it first calculates the value of each element based on different stencils. Then the program updates the boundary values in the array. A brief diagram of the program structure is shown in Figure 3.1. From this diagram we can see that the most time consuming part is the node calculation because the program needs to calculate all the elements in the grid until it reaches a certain amount of iterations. For each iteration, the program need to traverse every node in the array and calculate the value based on the stencil. In the original program, all of these are done sequentially. We cannot parallelize the outside loop. Because in each iteration, the values in the array depends on the values in the last iteration, so it has to execute sequentially. Therefore, the code in each iteration is a good candidate to be parallelized.

3.2 CPU parallelization

We use OpenMP to parallelize this program on CPU. The parallelization is easy and straightforward. Instead of iterate through all elements in the array and update their values one by one, we parallelize this function so that multiple elements can be calculated in parallel because their values are independent of each other. For the iteration part, there are two steps. First, the program traverses through the array to calculate the value of each element based on the specified stencil and stores the data in a temporary array. In the next step, the program copies the data back to the grid

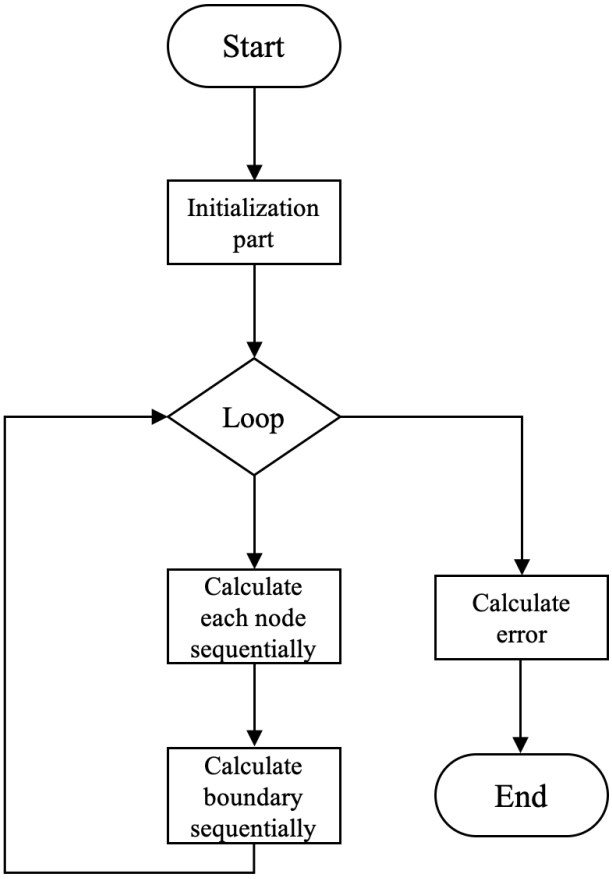


Figure 3.1: A brief structure of this program

data structure and goes to the next iteration. Figure 3.2 shows the first step of the code.

There are two "for" loops, and we only need to add a directive **#pragma omp parallel for default(shared)** in the beginning of the loops, and then the iterations will be executed in parallel. In this directive, the first two words **#pragma omp** indicates that the following code block is a parallel region and it should be compiled using OpenMP compiler. Next, when the master thread reaches **parallel** directive, it forks a group of threads to execute the code in the parallel region. The code in the parallel region is duplicated and executed by all threads including the master thread. If any thread terminates inside the parallel region, the termination will be applied to all threads in that group. Then 'for' directive means the following code should be "for" loops, otherwise it may not work properly. The 'default' directive followed by 'shared' specifies all variables in the parallel region are shared by all threads. Figure 3.3 shows a very similar code snippet, which is to copy data back to the original array in parallel.

When parallelizing program with OpenMP, the number of thread has a significant impact on the final results. If more threads are used, the running speed is expected to increase because more calculate is running in parallel. If the thread number exceeds the core number of CPU, the running speed will remain the same

```
1
2  HaloArray3D *uh = new HaloArray3D(u->l, Vec3D<int>(0), 1);
3  double Ux = V.x * dt / delta.x, Uy = V.y * dt / delta.y;
4
5  #pragma omp parallel for default(shared)
6    for (int j=0; j < u->l.y; j++)
7      for (int i=0; i < u->l.x; i++) {
8        double cim1, ci0, cip1;
9        double cjm1, cj0, cjp1;
10       N2Coeff(Ux, cim1, ci0, cip1);
11       N2Coeff(Uy, cjm1, cj0, cjp1);
12       Vh(uh,i,j) =
13         cim1*(cjm1*Vh(u,i-1,j-1) + cj0*Vh(u,i-1,j) + cjp1*Vh(u,i-1,j+1)) +
14         ci0 *(cjm1*Vh(u,i  ,j-1) + cj0*Vh(u,i,  j) + cjp1*Vh(u,i,  j+1)) +
15         cip1*(cjm1*Vh(u,i+1,j-1) + cj0*Vh(u,i+1,j) + cjp1*Vh(u,i+1,j+1));
16     } // for (j...)
```

Figure 3.2: The first step of OpenMP parallelization of a specific stencil function

```
1  #pragma omp parallel for default(shared)
2    for (int j=0; j < u->l.y; j++)
3      for (int i=0; i < u->l.x; i++) {
4        Vh(u,i,j) = Vh(uh,i,j);
5      }
```

Figure 3.3: The second step of OpenMP parallelization of a specific stencil function

because there are not so many actual threads. Programmers could control the number of threads by several methods. They can set the thread number in the directives or call **omp_set_num_threads()** function. Programmers can also set the **OMP_NUM_THREADS** environment variable in the terminal before running the program. In this program, we call **omp_set_num_threads()** in the program so that we can set the thread number by the argument we pass to the program. Here, we use various number of threads to execute the program and investigate their results. The result discussions are in section 5.1.

3.3 GPU parallelization

The original program uses some library that can't be compiled by CUDA compiler, so we rewrite this program and use CUDA to parallelize it. The command to execute this program is In this program, we parallelize every sweep (time step) of the iteration. We are using Jacobi iterations in our program so there are two steps in each sweep. First, the program initializes a writing array and calculates the values of elements in the array in parallel and stored the results in the corresponding positions in the new array. Then the values in the new array are copied back to the original array in parallel. These two steps have to be implemented in two separate kernels to ensure all threads complete before the second step.

Before the iterations begin, we first declare the data grid on the unified memory space using **cudaMallocManaged()** function and then copy the data from CPU mem-

```
cudaMallocManaged((void**) &d_u, sizeof(HaloArray3D));
cudaMallocManaged((void**) &uu, sizeof(double) * sizeu);
for(int i = 0; i < sizeu; i++)
    uu[i] = u->u[i];
d_u->u = uu;
d_u->l = u->l;
d_u->s = u->s;
d_u->halo = u->halo;
d_u->B = u->B;
```

Figure 3.4: Allocate memory on unified memory and copy grid data from CPU to unified memory

```
__global__ void LWN2kernel1(HaloArray3D* u, HaloArray3D* uh, double Ux, double
    Uy,
 double cim1, double ci0, double cip1, double cjm1, double cj0, double cjp1){
    int i0 = blockIdx.x * blockDim.x + threadIdx.x, di = blockDim.x*gridDim.x;
    int j0 = blockIdx.y * blockDim.y + threadIdx.y, dj = blockDim.y*gridDim.y;
    int x = i0 + j0 * di, total = di * dj;
    for (int i = x; i < u->l.x; i += total) {
         for(int j = 0; j < u->l.y; j++){
            Vh(uh,i,j)=cim1*(cjm1*Vh(u,i-1,j-1)+cj0*Vh(u,i-1,j)+cjp1*Vh(u,i-1,j
                +1))
            +ci0*(cjm1*Vh(u,i,j-1) + cj0*Vh(u,i,j)+cjp1*Vh(u,i,j+1))
            +cip1*(cjm1*Vh(u,i+1,j-1)+cj0*Vh(u,i+1,j)+cjp1*Vh(u,i+1,j+1));
        }
    }
}
```

Figure 3.5: Row-wise implementation of the CUDA kernel

ory to the unified memory, as shown in Figure 3.4. Next, we declare an empty write data grid in the unified memory to store data at each iteration. Then the program launch the kernel to compute stencils in parallel.

We use different implementations in the kernel and investigate their performance. A naive implementation would be using one thread to calculate one row or one column in the data grid. The implementation is straightforward. Figure 3.5 shows the kernel that each thread is responsible for calculating one row in the data grid. We call this implementation as row-wise implementation. From the code we can see that we first calculate the ID of each thread **x**. Then we use x as the row index to iterate through all the elements in the row and updating their values. In cases when the number of rows exceeds the total number of the threads, the row index adds the total thread number in the first loop after finishing calculating the first row. If there isn't enough thread for each row, the completed threads will calculate next row. Similarly, Figure 3.6 shows the kernel that each thread calculates one column at a time. These two implementations are very similar and we only need to swap **i** and **j**. We call this as column-wise implementation.

```
__global__ void LWN2kernel1(HaloArray3D* u, HaloArray3D* uh, double Ux, double
    Uy,
 double cim1, double ci0, double cip1, double cjm1, double cj0, double cjp1){
    int i0 = blockIdx.x * blockDim.x + threadIdx.x, di = blockDim.x*gridDim.x;
    int j0 = blockIdx.y * blockDim.y + threadIdx.y, dj = blockDim.y*gridDim.y;
    int x = i0 + j0 * di, total = di * dj;
    for (int j = x; j < u->l.x; j += total) {
         for(int i = 0; i < u->l.y; i++){
            Vh(uh,i,j)=cim1*(cjm1*Vh(u,i-1,j-1)+cj0*Vh(u,i-1,j)+cjp1*Vh(u,i-1,j
                +1))
            +ci0*(cjm1*Vh(u,i,j-1) + cj0*Vh(u,i,j)+cjp1*Vh(u,i,j+1))
            +cip1*(cjm1*Vh(u,i+1,j-1)+cj0*Vh(u,i+1,j)+cjp1*Vh(u,i+1,j+1));
        }
    }
}
```

Figure 3.6: Column-wise implementation of the CUDA kernel

```
__global__ void LWN2kernel1(HaloArray3D* u, HaloArray3D* uh, double Ux, double
    Uy, double cim1,
        double ci0, double cip1, double cjm1, double cj0, double cjp1){
    int i0 = blockIdx.x * blockDim.x + threadIdx.x, di = blockDim.x*gridDim.x;
    int j0 = blockIdx.y * blockDim.y + threadIdx.y, dj = blockDim.y*gridDim.y;

    int size = 16;
    for(int p = j0; p < u->l.y / size + 1; p += dj){
        for(int q = i0; q < u->l.x / size + 1; q += di){
            for(int m = 0; m < size; m++){
                int j = size * p + m;
                for(int n = 0; n < size; n++){
                    int i = size * q + n;
                    if(i >= u->l.x || j >= u->l.y) continue;
                    Vh(uh,i,j)=cim1*(cjm1*Vh(u,i-1,j-1)+cj0*Vh(u,i-1,j)+cjp1*Vh(
                        u,i-1,j+1))
                    +ci0*(cjm1*Vh(u,i,j-1) + cj0*Vh(u,i,j)+cjp1*Vh(u,i,j+1))
                    +cip1*(cjm1*Vh(u,i+1,j-1)+cj0*Vh(u,i+1,j)+cjp1*Vh(u,i+1,j+1)
                        );
                }
            }
        }
    }
}
```

Figure 3.7: Block-wise implementation of the CUDA kernel

```
__global__ void LWN2kernel1(HaloArray3D* u, HaloArray3D* uh, double Ux, double
    Uy,
 double cim1, double ci0, double cip1, double cjm1, double cj0, double cjp1){
    int i0 = blockIdx.x * blockDim.x + threadIdx.x, di = blockDim.x*gridDim.x;
    int j0 = blockIdx.y * blockDim.y + threadIdx.y, dj = blockDim.y*gridDim.y;
    for(int j = j0; j < u->l.y; j+= dj){
        for(int i = i0; i < u->l.x; i += di){
            Vh(uh,i,j)=cim1*(cjm1*Vh(u,i-1,j-1)+cj0*Vh(u,i-1,j)+cjp1*Vh(u,i-1,j
                +1))
            +ci0*(cjm1*Vh(u,i,j-1) + cj0*Vh(u,i,j)+cjp1*Vh(u,i,j+1))
            +cip1*(cjm1*Vh(u,i+1,j-1)+cj0*Vh(u,i+1,j)+cjp1*Vh(u,i+1,j+1));
        }
    }
}
```

Figure 3.8: Element-wise implementation of the CUDA kernel

The third implementation is called block-wise implementation, as shown in Figure 3.7. In this implementation, we divide the data grid into multiple square cells with a certain size. Each thread calculates all the element in that block and then goes to the next block. In the code snippet, we can see there are four loops. The first loop and the second loop determine the thread ID, and the next two loops iterate through elements in the block and calculate their values. After all elements in this block are finished, then the thread goes to the next block and continues. We can set the block size as arbitrary number.

There is another implementation called element-wise implementation, as shown in Figure 3.8. In this implementation, the data grid is also divided into blocks whereas each thread calculates only one element in the block. Then, it goes to the next block and calculate the element in the same position. From the code snippet we can see that the data block size is the same as the thread block size. Their indices are the same so the implementation is straightforward.

Chapter 4

Experimental Methodology

4.1 Environment specification

The machine used in the experiments are specified as below:

	Whale
OS	Ubuntu 20.04.1 LTS 64-bit
CPU	Intel(R) Xeon Gold
GPU	NVIDIA Tesla P100-PCIE-12GB
OpenMP	OpenMP 5.0.1
Compiler	gcc 8.4.0
CUDA runtime	CUDA 11.0
CUDA driver	450.80.02

Table 4.1: Software environment

	Whale
Model	2 × NVIDIA Tesla P100
Architecture	Tesla
Memory	16 GB
Clock rate	1328 MHz
Power limit	250.0 W
Streaming multiprocessor	56
CUDA cores	3584

Table 4.2: GPU specifications

Chapter 5

Results and discussion

5.1 OpenMP parallelization

Figure 5.1 shows the running time of this program using OpenMP parallelization with different number of threads. First, the red curve in the top represents the running time when using only one thread to execute this program, which means there is no parallelization. When we use two threads to execute the program, the running time decreases nearly by half. This means we have parallelized most of the time consuming part of the program, the rest part of the program that we didn't parallelize can be negligible. With thread number increases, the running time decreases accordingly. Every time we double the thread number, the running time decreases significantly. However, when we use more than 16 threads, the running speed doesn't improve much. This is limited by the CPU architecture. We are using an Intel Xeon Gold 6134 CPU, which has 16 cores and 2 threads per core. However, there are not 32 threads in total. The two threads in each core use hyper-threading technology, which are two virtual threads created by the operating system. These two virtual threads in one physical core share the workload when necessary to increase the number of independent instructions in the pipeline. These two virtual cores couldn't run in parallel, therefore, excessive threads couldn't help to optimize the program even further. It worth mentioning that the running speed of all stencils are faster than TMR, demonstrating robust stencil better performance.

According to calculation, the average running time of all stencils using 2 threads is 52.23% compared to using 1 thread. Using 4 threads is 56.23% compared to 2 threads. Using 8 threads is 56.30% compared to 4 threads, and using 16 threads is 62.52% compared to 8 threads. When using 32 threads, the running time is 97% compared to 16 threads. At first, when we double the thread number, the running time is reduced by nearly 50%, which means we effectively parallelized the program. However, with the thread number further increases, the improvement gradually becomes slower. This might result from cache misses. The more threads are executing, the more data the CPU is processing. Another possible reason is overheating. More threads working simultaneously means more power generation, thus will lead to overheating for the CPU, which, of course, will effect the efficiency of parallelization.

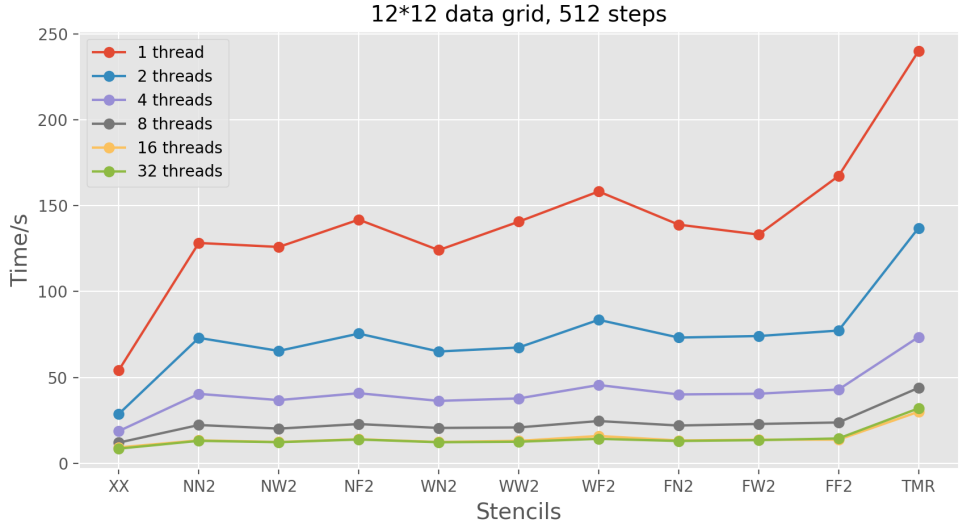


Figure 5.1: The running time of the program using OpenMP parallelization with different number of threads

5.2 CUDA parallelization

We use four different approaches to parallelize the program with CUDA as described in section 3.3. Then we test their performance on different stencils. In the experiments, the data grid size is 10*10 and step number is 512. The thread dimensions are 2*2 blocks and each thread block has 32*32 threads. Figure 5.2 compares the results of four implementations of the program. Different color of bars represent the running time of different implementations. For all stencils, the four implementations show a similar performance. The element-wise implementation shows the best performance, next is the row-wise implementation, the third is block-wise and is very close to the last one – column-wise implementation. Their results vary significantly even their implementations are very similar. In the kernel of a stencil computation program, only a minor change in the code could lead to great difference in the results.

Three stencils on the right – C30, C50, C70 are three combined stencils, which combine several singular stencils together to form one stencil. These combined stencils are more complicated, thus they have longer running time. From the figure we can see only element-wise implementation can successfully calculates these stencils whereas all other three implementation couldn't get the right answer. The reason is that the resources for each thread adding together have exceeded the GPU limitation. Therefore, some of the threads are not activated. The GPU has several limitations. First, the total amount of on-chip shared memory in each SM are limited. If all shared memory of threads in a SM exceeds the limits, some of the threads may not be able to work. Moreover, the number of total registers is also constrained by the hardware. If every thread uses too much registers, there will be fewer active threads. Therefore, we can conclude that threads use least resources in the element-wise implementation,

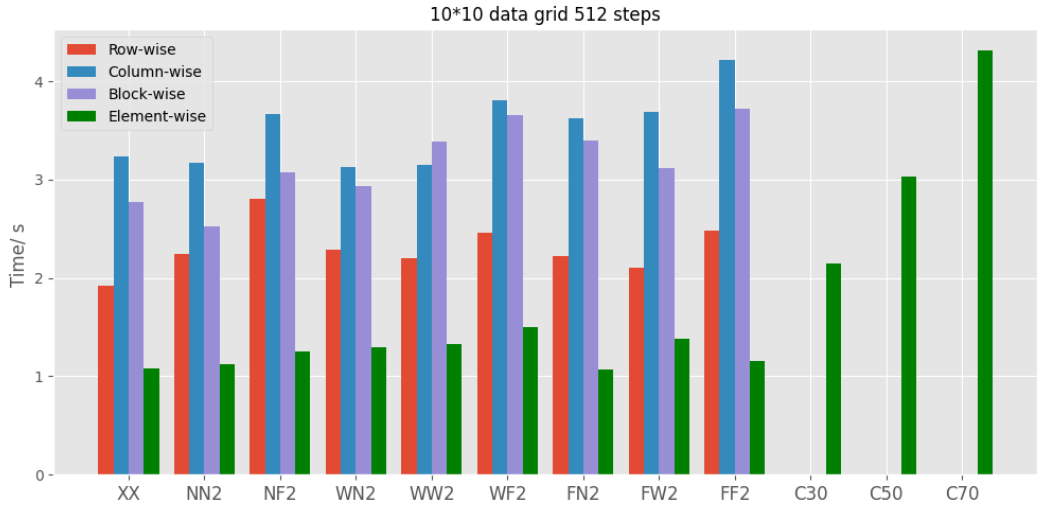


Figure 5.2: The running time of different stencils in the program with 4 different implementations

which is consistent with the results of singular stencils.

There might be multiple reasons. First, this implementation is simple. It doesn't need extra variables so that fewer registers are used for each thread. In comparison, block-wise implementation uses several extra variables, which will increase the burden for each thread. Second, this implementation could help increase locality. The GPU schedules tasks in groups of 32 threads called a warp. A warp shares a same cache space. In this implementation, all threads in a warp use same memory blocks in the memory so that memory response is fast. For other implementations such as row-wise implementation, threads in the same warp calculate elements in a column and they are not in the same memory bank.

Figure 5.3 shows running time comparison between serial programming and parallel programming. Here we use the best implementation element-wise as the parallelization results. This figure clearly shows the parallel programming is much faster that serial programming. This conclusion becomes more obvious when it comes to the combined stencils. For the combined stencils, more data and more computations are required, which is more suitable for GPU than CPU. Thus the gaps become even larger. This shows parallel computing has tremendous potential in compute-intensive programs.

Since we have found the best implementation approach, we need to tune the parameters to further optimize the program. The thread dimensions are important because it determines how the device organized memory. Here we use a fixed number of 4096 threads and different configurations to test the impact of thread dimension and find out the best configuration to parallelize the program.

Figure 5.4 depicts the running time of the program with different thread dimensions. The x-axis is the thread number, the first number is the block number and the second number is the thread number in each block. We can see that different thread

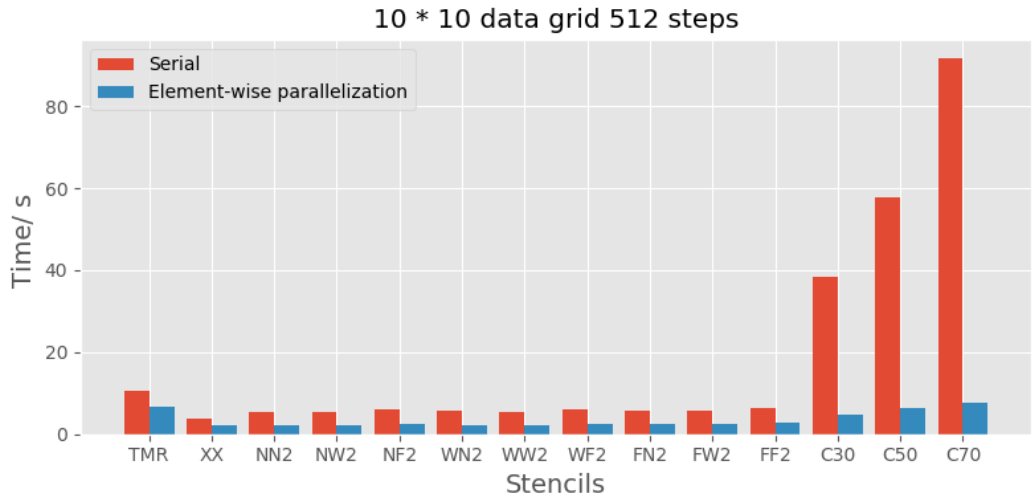


Figure 5.3: The running time comparison of serial execution and parallel execution

structure will have significant impact on the running speed. If we choose 8×512, the running time is less than 5 seconds. If we use 128×32, the running time is about 12 seconds, which is more than two times longer. In this case, 8×512 is the best choice. If the block dimension is less than 128, the running time will increases significantly. From these observations we can conclude that thread dimensions have a significant impact on the program performance. The programmers may need to try different dimensions to find out the most suitable values.

5.3 OpenMP and CUDA results comparison

Figure 5.5 compares the best results from OpenMP parallelization and CUDA parallelization. We can see that both OpenMP and CUDA could effectively optimize the program. The program after parallelization runs more than 10 times faster than the original program. For the simple stencils, the running speed of OpenMP parallelization and CUDA parallelization are very close. For some stencils, OpenMP version is a little faster and some stencils CUDA version is little faster. They both show powerful capability to speedup the program. However, when it comes to combined stencils, the running time of OpenMP parallelization increases whereas CUDA parallelization barely changes, demonstrating CUDA has better optimization. The reason is simple. OpenMP uses CPU to parallelize the program. CPU has much less threads than GPU, but each thread is much powerful than threads in GPU. Therefore, CPU is more suitable for programs with complicated logical structure rather than data-intensive computing. The program we are using doesn't have complex logic, most of the execution time is to calculate the element in the grid, which is a very typical data-intensive program. Therefore, in this case, GPU is more suitable to optimize this program.

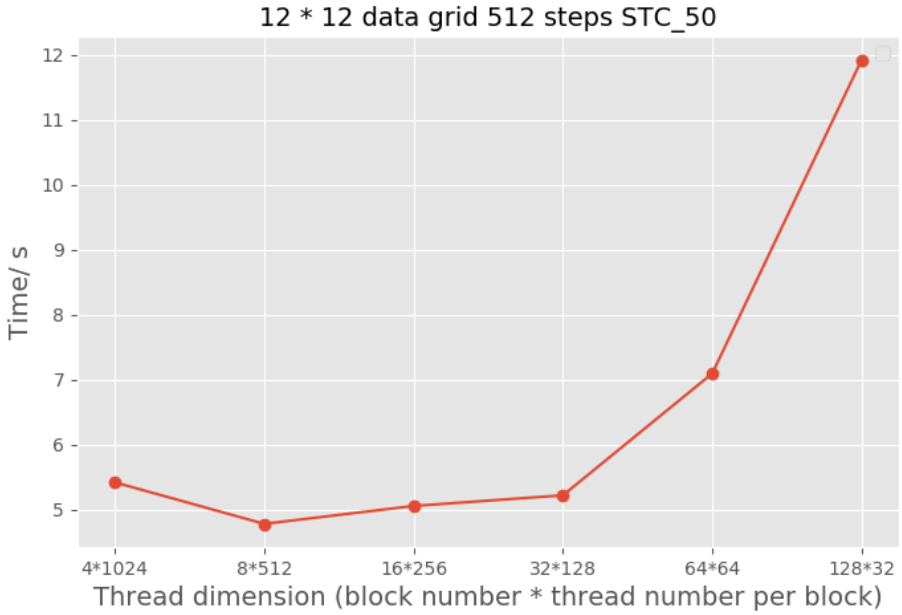


Figure 5.4: The running time using different thread configuration on GPU

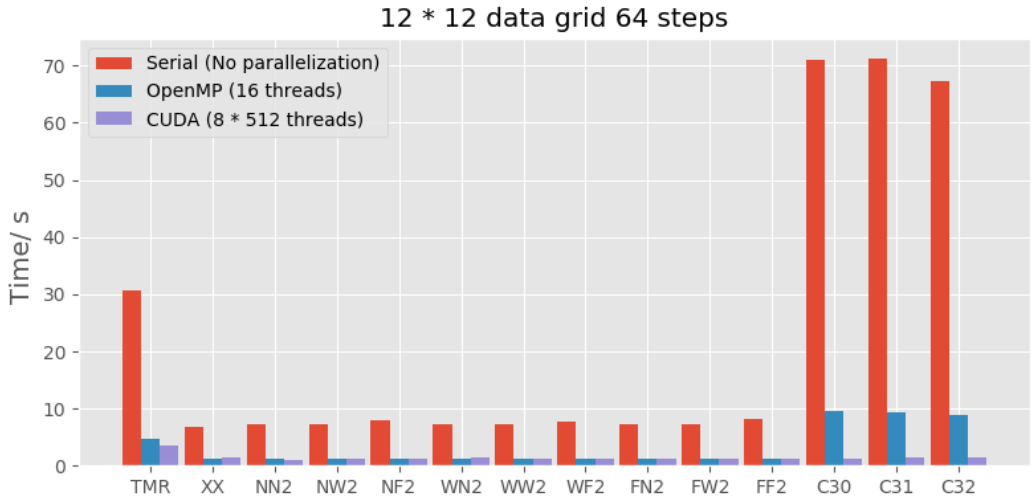


Figure 5.5: The running time of OpenMP prallelization and CUDA parallelization

Chapter 6

Conclusion

In summary, we have successfully parallelized the advection solver using both CPU parallelization (OpenMP) and GPU parallelization (CUDA). Both of them show tremendous potential in optimizing the program. We also investigated their properties and tuned the parameters to find the optimal configuration.

For OpenMP parallelization, it is easy to implement and could effectively improve the performance of the program. Every time we double the thread number to execute the program, the running speed nearly doubles until thread number reaches the CPU limits. However, its easy and simple implementations lead to lack of control for the thread behaviours and memory management, which may impede further optimizing the program.

For CUDA parallelization, we concluded that CUDA is capable of speeding up stencil computations up to 35 times. CUDA configuration is much more complicated than OpenMP. Not only the thread dimensions, but also the behaviours of each thread will have great impact on the performance. We have seen that only minor changes in the code, such as the iteration order or compute scope of each thread, may lead to huge differences in the results. Therefore, it is important for programmers to understand its characteristics to write efficient CUDA programs.

6.1 Future Work

There are several approaches to optimize stencil computing programs in multi-core system, such as time-tiling, streaming and grid decomposition. All of these methods have shown great potential in optimizing stencil computing in parallel. I read some papers and tried to optimize this program using time-tiling optimization but failed. In the future, I would like to implement some techniques from the paper to further optimize the program, or maybe we can develop a new technique.

Bibliography

Buck, I., 2007. Gpu computing: Programming a massively parallel processor. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, 17. IEEE Computer Society, USA. doi:10.1109/CGO.2007.13. https://doi.org/10.1109/CGO.2007.13. (cited on page 7)

Bucur, A., 2014. Mobile gpu compute exploring the mobile gpu through opencl. In *ACM SIGGRAPH 2014 Talks*, SIGGRAPH '14 (Vancouver, Canada, 2014). Association for Computing Machinery, New York, NY, USA. doi:10.1145/2614106.2614184. https://doi.org/10.1145/2614106.2614184. (cited on page 7)

Datta, K. and Yelick, K. A., 2009. *Auto-tuning stencil codes for cache-based multicore platforms*. University of California, Berkeley. (cited on pages ix, 10, and 11)

Gibbons, P. B., 2014. Acm transactions on parallel computing: An introduction. *ACM Trans. Parallel Comput.*, 1, 1 (Oct. 2014). doi:10.1145/2661651. https://doi.org/10.1145/2661651. (cited on page 8)

Hoffman, F. and Hargrove, W., 1999. Parallel computing with linux. *XRDS*, 6, 1 (Sep. 1999), 23–27. doi:10.1145/331636.331643. https://doi.org/10.1145/331636.331643. (cited on page 10)

Holewinski, J.; Pouchet, L.-N.; and Sadayappan, P., 2012. High-performance code generation for stencil computations on gpu architectures. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12 (San Servolo Island, Venice, Italy, 2012), 311–320. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2304576.2304619. https://doi.org/10.1145/2304576.2304619. (cited on page 3)

Ikei, M. and Sato, M., 2014. A pgas execution model for efficient stencil computation on many-core processors. In *Proceedings of the 14th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*, CCGRID '14 (Chicago, Illinois, 2014), 305–314. IEEE Press. doi:10.1109/CCGrid.2014.20. https://doi.org/10.1109/CCGrid.2014.20. (cited on pages 4 and 11)

K. Asanovic, J. D. T. K. K. K. J. K. N. M. D. P. K. S. J. W. D. W., R. Bodik and Yelick, K., 2009. A view of the parallel computing landscape. *Commun. ACM*, 52, 10 (2009), 56–67. doi:10.1145/1562764.1562783. (cited on page 3)

Lee, K.; Sullivan, M. B.; Hari, S. K. S.; Tsai, T.; Keckler, S. W.; and Erez, M., 2019. Gpu snapshot: Checkpoint offloading for gpu-dense systems. In *Proceedings*

of the ACM International Conference on Supercomputing, ICS '19 (Phoenix, Arizona, 2019), 171–183. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3330345.3330361. https://doi.org/10.1145/3330345.3330361. (cited on page 6)

Luporini, F.; Louboutin, M.; Lange, M.; Kukreja, N.; Witte, P.; Hückelheim, J.; Yount, C.; Kelly, P. H. J.; Herrmann, F. J.; and Gorman, G. J., 2020. Architecture and performance of devito, a system for automated stencil computation. *ACM Trans. Math. Softw.*, 46, 1 (Apr. 2020). doi:10.1145/3374916. https://doi.org/10.1145/3374916. (cited on pages 5 and 10)

Melchor, C.; Crespin, B.; Gaborit, P.; Jolivet, V.; and Rousseau, P., 2008. High-speed private information retrieval computation on gpu. 263–272. doi:10.1109/SECURWARE.2008.55. (cited on pages ix and 8)

Nickolls, J.; Buck, I.; Garland, M.; and Skadron, K., 2008. Scalable parallel programming with cuda: Is cuda the parallel programming model that application developers have been waiting for? *Queue*, 6, 2 (Mar. 2008), 40–53. doi:10.1145/1365490.1365500. https://doi.org/10.1145/1365490.1365500. (cited on pages ix and 6)

Peter Strazdins, C. L. J. R. M. J. R., Brendan Harding and Armstrong, R. C., 2016. A robust technique to make a 2d advection solver tolerant to soft faults. *Procedia Computer Science*, 80, 10 (2016), 1917–1926. doi:10.1016/j.procs.2016.05.505. (cited on pages ix, 2, 11, 12, and 13)

Romein, J. W., 2012. An efficient work-distribution strategy for gridding radio-telescope data on gpus. In *Proceedings of the 26th ACM International Conference on Supercomputing*, ICS '12 (San Servolo Island, Venice, Italy, 2012), 321–330. Association for Computing Machinery, New York, NY, USA. doi:10.1145/2304576.2304620. https://doi.org/10.1145/2304576.2304620. (cited on page 4)

Sano, K.; Yamamoto, S.; and Hatsuda, Y., 2011. Domain-specific programmable design of scalable streaming-array for power-efficient stencil computation. *SIGARCH Comput. Archit. News*, 39, 4 (Dec. 2011), 44–49. doi:10.1145/2082156.2082168. https://doi.org/10.1145/2082156.2082168. (cited on page 5)

Soejima, R.; Okina, K.; Dohi, K.; Shibata, Y.; and Oguri, K., 2014. A memory profiling framework for stencil computation on an fpga accelerator with high level synthesis. *SIGARCH Comput. Archit. News*, 42, 4 (Dec. 2014), 69–74. doi:10.1145/2693714.2693727. https://doi.org/10.1145/2693714.2693727. (cited on page 5)

Stout, Q. F. and Jablonowski, C., 2006. Parallel computing 101. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing*, SC '06 (Tampa, Florida, 2006), 203–es. Association for Computing Machinery. (cited on page 10)

Yongpeng Zhang, F. M., 2012. Auto-generation and auto-tuning of 3d stencil codes on gpu clusters. *Proceedings of the Tenth International Symposium on Code Generation*

and Optimization, 4, 77 (2012), 155–164. doi:10.1145/2259016.2259037. (cited on page 3)

Zohouri, H. R.; Podobas, A.; and Matsuoka, S., 2018. Combined spatial and temporal blocking for high-performance stencil computation on fpgas using opencl. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '18 (Monterey, CALIFORNIA, USA, 2018), 153–162. Association for Computing Machinery, New York, NY, USA. doi:10.1145/3174243.3174248. https://doi.org/10.1145/3174243.3174248. (cited on page 5)