Concurrent Task Execution on the Intel Xeon Phi

by

Yucheng Zhu

Bachelor of Computer Science, NUAA, 2010

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Master of Computer Science

In the Graduate Academic Unit of Faculty of Computer Science

Supervisor(s):	Eric Aubanel, PhD, Faculty of Computer Science, UNB
Examining Board:	Virendrakumar C. Bhavsar, PhD, Faculty of Computer Science,UNB
	Gerhard Dueck, PhD, Faculty of Computer Science, UNB
	Andrew Gerber, PhD, Faculty of Engineering, UNB

This thesis is accepted

Dean of Graduate Studies

THE UNIVERSITY OF NEW BRUNSWICK

October, 2014

©Yucheng Zhu, 2015

Abstract

The Intel Xeon Phi coprocessor is a new choice for the high performance computing industry and it needs to be tested. In this thesis, we compared the difference in performance between the Xeon Phi and the GPU. The Smith-Waterman algorithm is a widely used algorithm for solving the sequence alignment problem. We implemented two versions of parallel Smith-Waterman algorithm for the Xeon Phi and GPU. Inspired by CUDA stream which enables concurrent kernel execution on Nvidia's GPUs, we propose a socket based mechanism to enable concurrent task execution on the Xeon Phi. We then compared our socket implementation with Intel's offload mode and with an Nvidia GPU. The results showed that our socket implementation performs better than the offload mode but is still not as good as the GPU.

Acknowledgements

Thanks to Kevin Wilcox from Envenio for the help and advice.

Table of Contents

Abstract **ii**

Acknowledgments **iii**

Table of Contents **vi**

List of Figures **ix**

1 Introduction **1**

2 Background **3**

2.1 Task parallelism in HPC projects 3

2.2 The Intel Xeon Phi coprocessor Architecture 5

2.2.1 Offload directives 6

2.2.2 Using socket and pthreads on Xeon Phi 8

2.3 Wavefront Pattern 9

2.3.1 Multiple Sequence Comparisons and our GPU implementation 11

3 Implementation **14**

3.1 GPU Implementation 14
3.1.1 The Naive Implementation 14
3.1.2 Memory Coalescing 15
3.1.3 Tiling and Shared Memory 15
3.1.4 Multiple Sequence Alignment and CUDA Stream . . . 18
3.2 Xeon Phi Kernel Implementation 18
3.2.1 Computation Kernel 18
3.2.2 Disadvantage of the Offload Mode 20
3.3 Xeon Phi Socket based implementation 21
3.3.1 Communication between host and Xeon Phi 22
3.3.2 The Host Program 24
3.3.3 The Server Program 26
3.4 Offload Implementation 28

4 Experimental Results and Analysis 29
4.1 Experiment Platforms 29
4.2 Experimental results 30
4.2.1 Xeon Phi Implementations 30
4.2.2 The Offload Experiment 45
4.2.3 The GPU Experiment 49

5 Conclusion 55

Bibliography 58

Vita

List of Figures

2.1 Architecture of an MIC core[1] 6
2.2 Wavefront pattern and the data dependency 9
3.1 Tiled wavefront 16
3.2 Socket communication illustration 23
4.1 Naive implementation in the native mode for a single comparison 31
4.2 Tiling implementation in the native mode for a single comparison of length 3000 34
4.3 Tiling implementation in the native mode for a single comparison of length 9000 34
4.4 Tiling implementation in the native mode for a single comparison of length 15000 35
4.5 Tiling implementation in the native mode for single comparison of length 21000 35
4.6 Multiple comparisons on the Xeon Phi, 3000 case 38
4.7 Multiple comparisons on the Xeon Phi, 3000 case 38
4.8 Multiple comparisons on the Xeon Phi, 9000 case 39

4.9 Multiple comparisons on the Xeon Phi, 15000 case 39
4.10 Multiple comparisons on the Xeon Phi, 21000 case 40
4.11 Reduced group size for 100 comparisons of size 3000, computation time 43
4.12 Reduced group size for 20 comparisons of size 9000, computation time 45
4.13 Reduced group size for 6 comparisons of size 15000, computation time 46
4.14 Reduced group size for 3 comparisons of size 21000, computation time 46
4.15 Reduced group size for 100 comparisons of size 3000, run time 47
4.16 Reduced group size for 20 comparisons of size 9000, run time . 47
4.17 Reduced group size for 6 comparisons of size 15000, run time . 48
4.18 Reduced group size for 3 comparisons of size 21000, run time . 48
4.19 30 tasks of 3000 length, 20 task of 9000 length, 4 task of 15000 length and 2 tasks of 21000 length mixed together 49
4.20 Offload implementation for sequence size of 3000 50
4.21 The run time and kernel time of GPU tiling version for single comparisons 52
4.22 The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 3000 52
4.23 The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 9000 53

4.24 The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 15000 53
4.25 The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 21000 54

Chapter 1

Introduction

With the development of computer technology, people now can solve problems that used to be too time consuming in a relatively short period of time. This is the goal of high performance computing(HPC) and is achievable using parallel computing systems. Nvidia's CUDA GPUs have played an important role in scientific computing and have become the important HPC devices on the market. Now a new architecture has arrived. It is the Intel Xeon Phi coprocessor. The Xeon Phi coprocessor is more like a CPU than a GPU. It contains 60 Xeon cores so it can be programmed as a conventional x86 processor. The unconventional parts of a Xeon Phi coprocessor are the extensions such as a bidirectional ring interconnect through which the 60 cores can share cache with each other at high speed and a 512-bit per core vector unit to perform many floating-point operations.

The Xeon Phi coprocessor is connected to the CPU via PCI express bus like GPUs. Intel provides several modes to use the Xeon Phi coprocessor. There is a built-in Linux OS in the Xeon Phi so we can compile a binary and run it natively on the Xeon Phi. Alternatively we can use "offload directives" to offload part of a program and the data needed from the host to the coprocessor. This is similar to a Nvidia CUDA kernel. Using native mode we can treat the coprocessor as one network node. In practice, we have found that the offload mode is more suitable to a production project because we can have the non-computational work run on the host and only offload the computational intensive part to the Xeon Phi. Yet the offload mode has its disadvantages. In this thesis, we will illustrate the disadvantages of the offload mode and give our implementation of a socket based solution.

Also, to fully utilize the device, we need the Xeon Phi to run multiple tasks concurrently. Nvidia GPUs are now designed to be able to run multiple tasks concurrently on different streams. This can help users to fully utilize the GPU when a single task is too small to fill the entire GPU. We think the Xeon Phi coprocessor needs this too because not all the tasks may be large enough for the coprocessor and running them one by one is a waste of time and resources. We propose a software solution to help the Xeon Phi process multiple tasks efficiently and concurrently. We then use this solution to achieve multiple sequence comparisons on the Xeon Phi and compare the performance with offload mode and with a GPU implementation.

Chapter 2

Background

In this chapter, we discuss the importance of task parallelism in high performance computing. We show some facts about the Intel Xeon Phi coprocessors and the programming techniques required. We then introduce the wavefront pattern which is used as our application.

2.1 Task parallelism in HPC projects

Multi-tasking is important in practical high performance computing projects. A large problem is often decomposed into smaller sub-problems to provide task parallelism and gain performance. A project using a GPU as a coprocessor may have some multi-task management mechanism on the host and send multiple tasks to the GPU to make sure that the GPU is fully occupied to get the best performance. For applications that consist of several indepen-

dent computational tasks that do not occupy the entire GPU, sequentially using the GPU one task at a time leads to performance inefficiencies[3]. It is therefore important for the programmer to cluster small tasks together for sharing the GPU[3]. In NVIDIA's Fermi architecture, there is a new feature which is called CUDA stream. CUDA stream is the way to perform multiple CUDA operations simultaneously (beyond multi-threaded parallelism). Typically in CUDA a parallel function is executed by different threads as a "kernel". With CUDA stream, we can launch multiple kernels and execute them simultaneously. This is called concurrent kernel execution. This feature can help us solve a group of small tasks by organizing them into different streams. Concurrent task execution is suitable for the Xeon Phi as well. One sub-problem may not fully occupy the Xeon Phi so we need to assign multiple tasks to the target. For Nvidia GPUs when the resources on the GPU are enough for two tasks then the two tasks can be executed on different hardware resources so that there will be no overhead of task switching. However on the Xeon Phi, multiple tasks are managed by the on board Linux system. If we don't specify the right number of threads for each tasks, there may be some overhead. So we need to build a multiple task management mechanism on the Xeon Phi. Another reason why management is important to Xeon Phi is that a program will die if it tries to allocate more memory than there is on the Xeon Phi. The task management module needs to check the available memory amount on the Xeon Phi before it sends a task there.

2.2 The Intel Xeon Phi coprocessor Architecture

The Intel Xeon Phi Coprocessor has up to 61 Intel MIC Architecture cores which can run at 1 GHz. The Intel MIC Architecture is based on the x86 ISA (Instruction Set Architecture) so it can be programmed as a regular x86 CPU. The coprocessor is extended with 64-bit addressing and 512-bit wide SIMD vector instructions and registers. Each core can execute 4 hardware threads in parallel. There are also multiple on-die memory controllers and other components beside the cores[1].

The new-designed Vector Processing Unit(VPU) within each core contains 32 512-bit vector registers. A new 512-bit SIMD ISA was introduced to support the new VPU. To get the best performance on the Xeon Phi, programmers have to fully utilize the vector unit[1]. The best way to achieve this is that programmer can make memory accesses coalesced and 64 bit aligned so that the VPU's register can load data from continuous memory addresses.

As shown in figure 2.1, each core has a 32KB L1 cache, a 32KB L1 instruction cache, and a 512KB L2 cache. There is a high speed bidirectional ring bus to connect all the cores and the memory controller. This mechanism creates a shared last-level cache of up to 32MB. Each core also contains a short in-order pipeline which reduces the overhead for branch misprediction.[1]

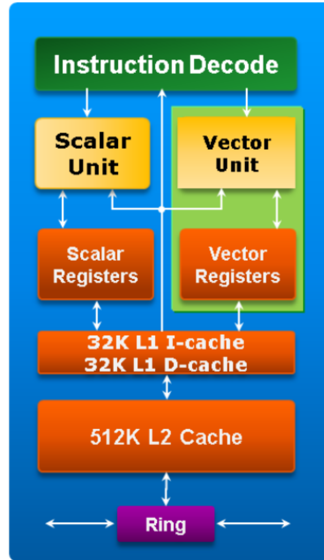

Figure 2.1: Architecture of an MIC core[1]

2.2.1 Offload directives

Programs can run natively on the Xeon Phi. Users can compile the application on the host and send the binary to the Xeon Phi by SSH. Users then can log in the Xeon Phi's on board Linux OS through SSH and run the application natively. Intel also introduces a offload mode. Intel provides several offload directives which can specify certain regions of code to run on the Xeon Phi coprocessor, also called the target. Code for both host and target is compiled within the host and the offloaded code is automatically run within the target environment. Within a offload directive, user can specify certain host memory units to be offloaded to the target. A typical offload consists of five activities:

1. Coprocessor data space allocation
2. Input data copied to the coprocessor memory
3. Offloaded execution on the coprocessor
4. Copying of results back to the host processor memory
5. Deallocation of data space allocated on the processor[2]

The offload mode has some limitations. First, the data types that can be offloaded are restricted. Only arrays with element type of scalar or bitwise copyable derived type in Fortran or bitwise copyable struct or class can be offloaded. So arrays of pointers or structs containing pointers or derived types containing allocatable arrays in Fortran are not supported for offloading to the Xeon Phi.

Second, the offload directives are synchronous. Any host thread that hits an offload region will be blocked until the offload work has finished on the target and the resulting data has been copied back. Intel also provides an asynchronous offload directives set. The host thread that hits an asynchronous offload region will return immediately and can perform host side work. This sounds like an asynchronous CUDA call but it is not as good as the one CUDA provides. In CUDA after launching an asynchronous CUDA kernel or CUDA memory copy, the host thread can check whether the work on the GPU has been finished using a checking function. If the function returns

with "not finished", the host thread can still do something else. However, in Xeon Phi programming, there is no such checking mechanism. A host thread can only wait for a signal that was specified for a particular asynchronous offload. This wait operation is blocking which means if the work hasn't been finished by the target, the host thread that has called the wait directive can't do anything until the work has been finished. This may cause performance degradation, and negatively impact task management.

2.2.2 Using socket and pthreads on Xeon Phi

The Xeon Phi coprocessor comes with an on-board Linux OS and an implementation of a virtual TCP/IP stack. This means that we can use the POSIX socket API to communicate between the host and the target and also means we can use Pthreads on the Xeon Phi. Unlike GPU's SIMD(Single Instruction Multiple Data) mode, the Phi coprocessors can run MIMD(Multiple Instruction Multiple Data) threads individually on the x86 cores. This means that threads on the Xeon Phi can execute different instructions at the same time. So we will program one Xeon Phi thread as a TCP server in our Xeon Phi implementation. This server thread will handle the socket connection between the host and Xeon Phi and launch other threads for the computations of tasks.

2.3 Wavefront Pattern

Wavefront is a typical pattern that appears in Dynamic Programming (DP) problems. In this kind of DP problem, we need to fill a matrix in memory. In this matrix each element only depends on its northwest, west and north neighbors. This data dependency makes the process of filling this matrix has a pattern of computing advancing anti-diagonals as shown in Figure 2.2. We call this pattern wavefront because it looks like a moving wavefront[6].

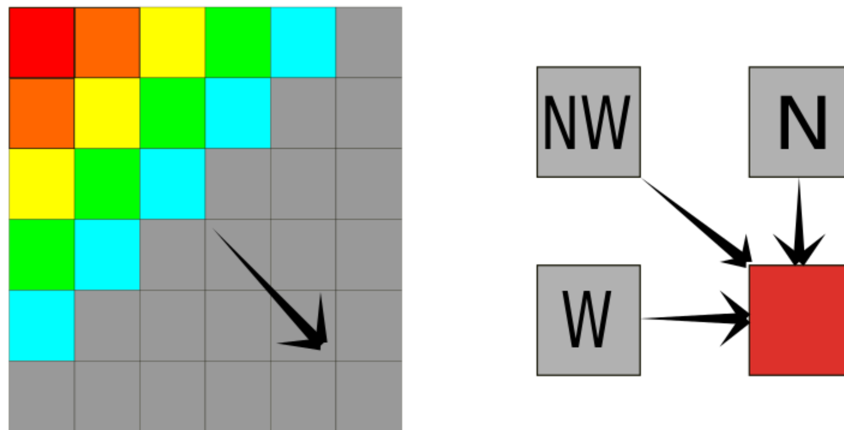

Figure 2.2: Wavefront pattern and the data dependency

From figure 2.2, we can tell that one important property of the wavefront

pattern is that each anti-diagonal in the matrix only depends on its former ones and elements within the same anti-diagonal have no data dependency on each other.

Longest Common Subsequence (LCS) is a typical wavefront problem. It is used to find the longest subsequence common to all sequences, which usually consists of characters such as DNA sequences (4 characters), in a set of sequences (often just two). There are some variations of LCS and among them, we choose the Smith-Waterman (SW) algorithm for our implementation. The SW algorithm is widely used in bioinformatics to determining similar regions between two nucleotide or protein sequences.

The Smith-Waterman algorithm is used in local sequence alignment. Local sequence alignment is performed to find the most similar parts in a pair of sequence by inserting or deleting the elements in the sequences. For example, if we have sequence A as "ATCAT" and sequence B as "ACAGT". Then A will be "ATCA_T" and B will be "A_CAGT" after performing the SW algorithm.

In general, let $A = a_1a_2a_3...a_m$ be the query sequence and $B = b_1b_2b_3...b_n$ be the target sequence. The SW algorithm builds a $(m+1)*(n+1)$ scoring matrix S where $S[i, j]$ stands for the highest similarity score between $a_1a_2a_3...a_i$

and $b_1b_2b_3...b_j$($1 \leq i \leq m$, $1 \leq j \leq n$). S[i,j] can be computed by [12]:

$$S(i,j) = \begin{cases} max(0, S[i,j-1]-p, S[i-1,j]-q, S[i-1,j-1]+W), & \text{if } a_i \neq b_j; \\ S[i-1,j-1]+W, & \text{if } a_i = b_j; \end{cases}$$

Here, p and q stand for the gap penalties between the two sequences and W stands for the cost to substitute a_i with b_j. In our thesis, we set them as $p = 1$, $q = 1$, and $W = 2$ when it's a match and $W = -1$ when it's a mismatch[12]. We also need to set $S[i,0] = S[0,j] = 0(0 \leq i \leq m,\ 0 \leq j \leq n)$. So our simplified formula is [12]:

$$S(i,j) = \begin{cases} max(0, S[i,j-1]-1, S[i-1,j]-1, S[i-1,j-1]-1), & \text{if } a_i \neq b_j; \\ S[i-1,j-1]+2, & \text{if } a_i = b_j; \\ S[i,0] = 0, & \text{if } 0 \leq i \leq m; \\ S[0,j] = 0, & \text{if } 0 \leq j \leq n; \end{cases}$$

2.3.1 Multiple Sequence Comparisons and our GPU implementation

In bioinformatics, comparisons are frequently performed between two DNA/RNA sequences which are not very long. Researchers have performed large sets of comparisons in parallel by scattering them onto different nodes of distributed computational systems or super computers. In [10], Tieng K. Yap

et al. suggested that there are two general parallel methods to perform sequence comparison on parallel computers. The first method is to parallelize the comparison algorithm by making all processors cooperate to compute the similarity matrix. The second one is to perform multiple sequence comparison by making each processor perform a subset of comparisons independently. We propose to enable multiple sequence comparisons on GPU and Xeon Phi. Most work which has been done on GPU only focuses on parallelization of single comparison such as Xiao's work in [5]. There are also a few implementations that focus on multiple sequence comparisons on GPU. In [9], Lukasz Ligowski and Witold Rudnicki gave an implementation of parallel sequences database querying on GPU using CUDA. In their implementation each thread performed a complete comparison of a single pair. Even though it can perform multiple sequence comparisons simultaneously, this implementation still has sequential comparison algorithms. In [11], Yongchao Liu, Douglas L Maskell and Bertil Schmidt showed a new way to perform multiple sequence comparisons on GPU. They introduced their idea of the "inter-task parallelization" and the "intra-task parallelization". The "inter-task parallelization" means that each task is assigned to one thread and all tasks are computed in parallel by different threads in a thread block while the "intra-taks parallelization" means that each task is assigned to a thread block and all threads within a block cooperate to perform the task in parallel. In their implementation all tasks with sequence length below 3072 were performed in the first stage of "inter-task parallelization" and the other tasks were per-

formed in the second stage of "intra-tasks parallelization". The disadvantage of their work is that in their "intra-tasks parallelization" each task was assigned to only one thread block. In our work we want to assign each task to multiple thread blocks based on each task's size.

Recent work in [7] implemented an application on GPU to parallelize the process of comparing a single pair of sequences. We have extended this work to do multiple sequence comparison simultaneously on GPU with a parallel comparison algorithm and use it to test multiple task execution on the Xeon Phi.

Chapter 3

Implementation

3.1 GPU Implementation

3.1.1 The Naive Implementation

First we introduce the simple solution in which each anti-diagonal of the matrix is launched as a kernel. Then elements within the anti-diagonal are assigned to different GPU threads based on the thread's id. In CUDA, threads are organized into blocks for scalability. Threads within a block can communicate directly with each other. A thread is identified by its blockId and threadId. The number of blocks can be read by variable gridDim and the number of thread within a block can be read from blockDim. A thread's id equals to "threadId + blockId * blockDim". The element a[i][j] is assigned to the thread which has id equals to (i+j) mod (blockDim*GridDim). We launch the kernel multiple times from the first anti-diagonal to the last

anti-diagonal.

The naive implementation is a basic mapping from a sequential CPU implementation to a parallel GPU implementation. There are several ways to improve it by make it more suitable to the GPU architecture.

3.1.2 Memory Coalescing

In the C language two dimensional arrays are normally stored in row major order. In the memory, a two dimensional array is stored as a one dimensional array row by row. Accessing a matrix in the row order can be very fast because of this memory layout. However, in our case, we are accessing the matrix in anti-diagonal order. So storing the matrix in anti-diagonal order can provide better caching. We made a function which can interpret the a[i][j] into its actual memory index in the anti-diagonal order array.

3.1.3 Tiling and Shared Memory

As discussed before, in CUDA threads are organized into blocks. If we organize the elements in the matrix into tiles, as shown in figure 3.1, we can then map one block of threads to one tile. The kernel launching computation of an anti-diagonal now becomes the computation of a tile-diagonal. Each

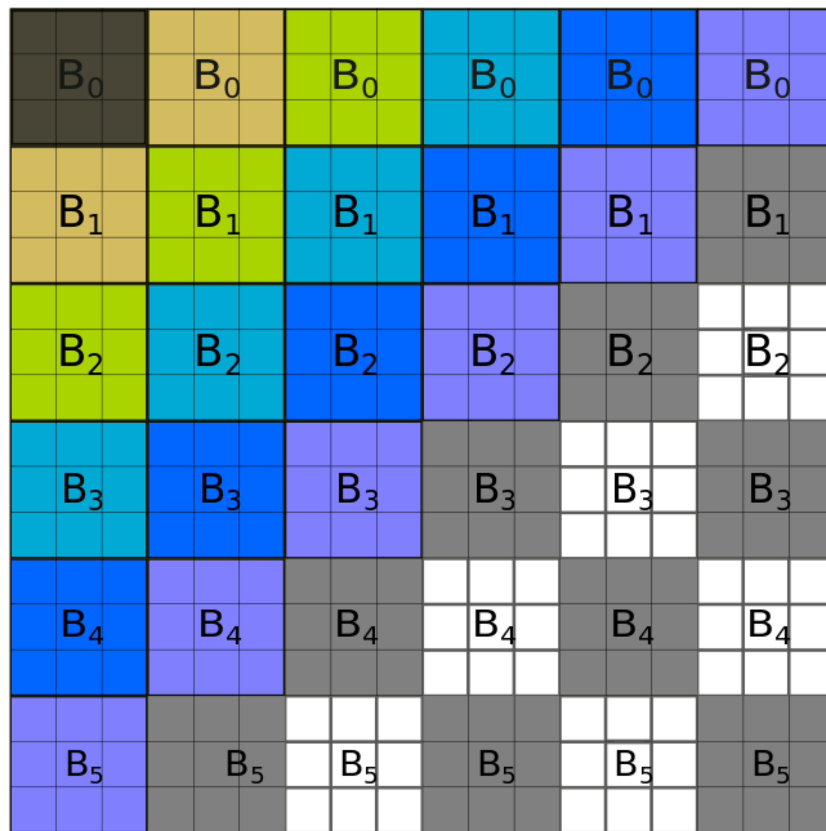

Figure 3.1: Tiled wavefront

kernel computes the results of one tile-diagonal. One thread block takes one tile and threads of that block solve the anti-diagonals of the tile through multiple iterations. This can help the caching and enable the use of shared memory as well.

In CUDA, programmers can allocate memory in shared memory. Shared memory is allocated per block[8]. Threads can access their block's shared memory much faster than they can access the global memory. So in our early implementation, we allocated shared memory for each tile and computed the tile in shared memory. Only when computation of the tile is finished, threads would write the result from the shared memory to the global memory. The shared memory is limited to 48KB per SM. So to enable the use of a large tile size, we only allocate memory for the current anti-diagonal, its former one, and the one before the former one because they are all we need. This solution enables larger tile size and improves the performance. Consider a tile which has width of n. If we store the whole tile-matrix in the shared memory, then we have: $n^2 < 48KB/4B$ (taking an integer as 4 byte). So we will have $n < 110.85$. This means that we can't have tile size more than 110. By storing only the 3 anti-diagonal in a tile, we can have much larger tile size. This also allows high occupancy because now there can be more blocks residing in a SM during the run time because the number of resident blocks is limited by the size of shared memory. Here is pseudo-code for the kernel:

1: Get the thread id and block id from CUDA API and get the unique thread id by id = blockId * blockDim + threadId
2: Use Block id to map a GPU thread block to an tile in the current tile-diagonal and allocate space for the 3 tile anti-diagonal in the block's shared memory
3: Within each block(now mapped to each tile), map the GPU threads to each entry of the tile matrix based on the id.
4: Load the base case from global memory to shared memory.
5: **loop** through the anti-diagonal
6:     Threads compute the current anti-diagonal in shared memory
7:     Synchronize threads within the block
8: **end loop**
9: Write the result from shared memory back to global memory in parallel

3.1.4 Multiple Sequence Alignment and CUDA Stream

For multiple sequence alignment, each kernel will be launched to a CUDA stream. Multiple alignment can be performed concurrently on the GPU in different streams. Here is pseudo-code:

3.2 Xeon Phi Kernel Implementation

3.2.1 Computation Kernel

The computation kernel in the Xeon Phi implementation is similar to the one on the GPU. There is also a tiling method in the Xeon Phi implementation.

```
 1: Create an array of type cudaStream
 2: Allocate host memory for each task
 3: Allocate device memory on each stream identified by an element in the
    cudaStream array
 4: Copy from each task's host memory to its device memory on specific
    stream
 5: parallel loop  from first task to last task
 6:     loop from the first tile-diagonal to the last one
 7:         Launch the Kernel to the task's stream and specify the arguments
 8:     end loop
 9:     Copy result from device to host for each task
10: end parallel loop
```

The differences are: First, in the Xeon Phi implementation only one thread is assigned to a tile instead of a block of threads as in the GPU. This is because the Xeon Phi has much less threads than the GPU. Instead of computing an anti-diagonal using multiple threads on the GPU, we assign an anti-diagonal within a tile to a thread. This thread computes the anti-diagonal using Xeon Phi's vector processing units. Second, there is no shared memory on the Xeon Phi. However, the Phi implementation can still benefit from tiling because of better caching. Here is the pseudo-code:

The second loop is an OpenMP for loop. It assigns tiles to different OpenMP threads. The inner most loop is a SIMD loop. Each thread only processes its own tiles. Although there is only one thread for the loop from the first element to the last element in an anti-diagonal within a tile, the computation of adjacent 16 elements within that anti-diagonal can be done in parallel.

```
1: loop From the first tile-diagonal to the last tile-diagonal
2:     parallel loop  From first tile to the last tile within a tile-diagonal
3:         loop From the first anti-diagonal to the last one within a tile
4:             loop From first element to the last element
5:                 computation of the element
6:             end loop
7:         end loop
8:     end parallel loop
9: end loop
```

This is because that the Xeon Phi's vector unit can add 16 pairs of integers simultaneously. This is done by marking the inner loop using "#pragma simd" directive.

3.2.2 Disadvantage of the Offload Mode

Normally, a Xeon Phi program is coded in C/Fortran associated with OpenMP directives to use multiple threads. A programmer can code in native mode or offload mode. For a complex problem, Intel recommends to use offload mode and only offload the computationally intense work to the Xeon Phi. Our sequence alignment problem may involve reading sequences from files or databases. These parts should stay on the host and the computation of each score matrix should be offloaded to the Xeon Phi. However, the offload mode may not work well in a multiple sequence alignment problem. First, we need to read the sequences from disk into dynamically allocated arrays and organize them into different structs so we can keep track of each comparison on the Xeon Phi. This means that each task has a struct of its id, a pair

of sequences, and a scoring matrix(also dynamically allocated). Because the offload directives can only offload bit-wise copyable memory units[1], we can not offload a whole struct to the Xeon Phi. So what we need to do is to offload each one of the components with that struct. The offload directive mapping variable on the host and variable on the Xeon Phi by name and there is no function call provided to explicitly allocate memory on the Xeon Phi in offload mode.

Also, as our tests show, the offload mode is not good for performance. First, offload directives have overhead themselves. The overhead can be worse when there are multiple offload directives executing concurrently. Second, the same application runs slower in offload mode than it runs in native mode. We will show these with experiment results later.

3.3 Xeon Phi Socket based implementation

For multiple sequence alignment, we manage each alignment as a task and we organize tasks into an array of structures. Each structure contains the id of the task, two pointers pointing to the two sequences, a pointer pointing to the scoring matrix, and other informations about each task. We then use our own communication method to launch tasks on the Xeon Phi. We also need to manage the memory allocation and data copy on the Xeon Phi.

3.3.1 Communication between host and Xeon Phi

We designed a socket communication mechanism for the Xeon Phi to solve the problem caused by the offload directives while still placing the computationally intense work on the Xeon Phi.

Although the Xeon Phi is connected to the host by PCI, we can still use the POSIX socket to achieve communication between the Xeon Phi and Host. This is achievable because on the host side it sees the Xeon Phi(in our case, mic0) as 172.31.1.1 and on the Xeon Phi side it sees the host as 172.31.1.254. We can verify this by the `cat /etc/hosts` command.
We made a client program on the host and the server program on the Xeon Phi. The client program does the job which should be done on the host, for example, reading the sequences from the disk. The server program runs natively on the Xeon Phi and listens to the clients. Figure 3.2 gives the basic structure of the communication mechanism. Multiple host threads send tasks from the host to the Xeon Phi through sockets and ask the server on the Xeon Phi to compute them there. The server spawns a new thread for a new connection and then this thread spawns more OpenMP threads to compute the tasks. The host thread that sends the task won't block to wait for the completion of the tasks. It can do other things like send another task. The client will launch some other threads to check if the tasks are completed

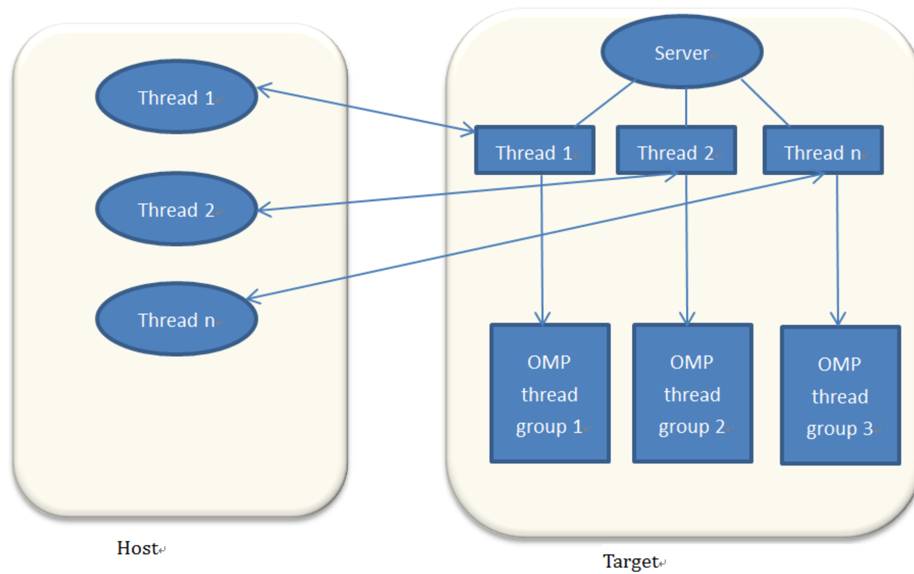

Figure 3.2: Socket communication illustration

on the Xeon Phi and bring the result back.

Here are some request and permission codes between the client and the server:

```
#define SEND_INITIAL_DATA 0x01
#define SEND_INITIAL_PERMITTED 0x02
#define START_CALC 0x03
#define CALC_PERMITTED 0x04
#define RETRIEVE_DATA 0x05
#define RETRIEVE_PERMITTED 0x06
#define DATA_READY 0x07
#define DATA_NOT_READY 0x08
```

Every time the host thread wants to perform something on the Xeon Phi, it will send one of the above requests to the server, the server will give the answer of whether it can perform this action based on the situation. For example, if a host thread sends a RETRIEVE_DATA to the server, the server will first send a RETRIEVE_PERMITTED action back to the client. The client then will know the server has accepted this request and it can send the id of the task which it wants to retrieve to the server. The server then sends DATA_READY or DATA_NOT_READY based on whether the task kernel has been finished. If the data is ready, the sever then sends the result to the socket. The client reads the action from the socket and if it is DATA_READY then it can read the result from the socket.

3.3.2 The Host Program

The host program starts by reading pairs of sequences from files into memory and organizing them into an array of structures. Each struct also contains some additional information about the task such as the length of the two sequences, the id of the task, and the pointers for the sequences and scoring matrix. The next step is to create a socket to send the entire struct array to the Xeon Phi. The server will build a server version of this array. This array on the server is necessary because we will launch tasks on the Xeon Phi asynchronously. The host thread which tries to launch a task on the Xeon Phi will return immediately and leave the tasks to the server. The server

program need this struct array to monitor all the tasks' status on the Xeon Phi. Although the pointers within structs are sent via the socket, the host memoryWe also need to send some other information using this socket such as the number of tasks.

After making sure that the server has successfully received the initial information, the host program closes the socket and then schedules tasks into groups. Tasks are sent to the server one group after another. There will be only one group of tasks running on the Xeon Phi at one time and only after the current group has been finished by the server can the next one be sent. As introduced before, currently a program will die if it tries to allocate more memory than there is on the Xeon Phi. This grouping arrangement can make sure that tasks only ask for a proper amount of memory each time. We have to make sure that the group of tasks we are sending to the server will not ask for more memory than there is currently available on the Xeon Phi.

The host program then sends one group of tasks to the server. Multiple host OpenMP threads will be created and each of them will be assigned a task. Each thread will then try to establish a TCP socket connection to the server and try to send the two sequences to the server. After making sure that the sever has received the sequences, the thread will terminate itself.

After a group of tasks has been sent to the server, the host program creates

another set of threads for retrieving the result(scoring matrix) back. Each thread will try to retrieve a particular task in the group by sending its id to the server via socket. If the computation has been finished by Xeon Phi, it will read the result from the socket buffer and write it into the task structure's scoring matrix field. If the computation hasn't been finished, it will try to retrieve again after a short period of time.

After all the tasks have been finished and all the results have been retrieved, the host program will start to send the next group of tasks(the server should clean up the memory for the new round).

3.3.3 The Server Program

The server program runs natively on the Xeon Phi. It starts by creating a TCP server and put it into listening status. Every time a socket request from the host is heard, the server creates a new thread and this new thread handles the request. We call this thread a working thread. A TCP socket connection will be established between this working thread and the host thread that sent the request unless there is something wrong in the network.

After the connection has been established, the working thread reads the request code of the host from the socket. There are three situations:

1. If the request is "SEND_INITIAL_DATA", the working thread will give

permission by sending a "SEND_INITIAL_PERMITTED" to the host to tell the host that it is ready for the initial information. It then gets the information from the socket. Now it knows the total number of tasks and it can create the task structures array in the Xeon Phi memory for the upcoming computations.

2. If the request is "START_CALC", the working thread will give permission by sending a "CALC_PERMITTED" to the host. It then reads the task's id from the socket to determine which task will be computed. It allocates the memory for the task's two sequences and scoring matrix and then starts the computation kernel for the task. One thing need to be noticed here is that the working thread will create OpenMP threads for the computation kernel. So there are two levels of parallelism on the server. After the computation is finished, the task is marked as finished.

3. If the request is "RETRIEVE_DATA", the working thread will send a "RETRIEVE_PERMITTED" to the host. It then reads the task's id from the socket to determine which task the host wants to retrieve. It then checks that task's status. If it has been finished, it will send the scoring matrix to the host. Otherwise, it will send a "DATA_NOT_READY" to the host. After the scoring matrix has been sent, the working thread frees the memory allocated for that task.

3.4 Offload Implementation

We also implemented an offload solution. The main thread organizes the tasks into a struct array. Unlike the socket solution, the offload implementation doesn't send the struct array to the Xeon Phi because sending structs which contain pointers is not allowed in offload directives. It then organizes the tasks into groups and creates pthreads for a group of tasks. The thread function contains the computational kernel which is marked with offload directives so that the kernel will run on the Xeon Phi. After all the threads have finished the offload work, the main thread processes the next group.

Chapter 4

Experimental Results and Analysis

4.1 Experiment Platforms

The GPU we are using is Nvidia's Tesla K20[8]. It contains 15 stream multiprocessors. Each multiprocessor can have at most 2048 resident threads or 16 resident blocks, and 48KB of shared memory. The total memory available is 4799MB. The Xeon Phi card we are using contains 60 cores. Each core can execute 2 instructions per cycle and supports 4 threads in hardware[1]. The thread affinity of the Xeon Phi card is set to scatter which means threads are distributed to all the 60 cores[1]. The total amount of memory is 7882316 kB. The sequences we are using are downloaded from the GenBank. The sequence lengths consists of 3000, 9000, 15000, and 21000. We are only

looking at comparisons of sequences of the same length. Unless otherwise specified, the execution time and run time always include the time spent in data transfer and the computation time of the kernels.

4.2 Experimental results

4.2.1 Xeon Phi Implementations

We also implemented the naive and tiling version of the Xeon Phi kernels. The naive implementation is not tiled. It iterates through every anti-diagonals of the scoring matrix. In each iteration, it assigns all the elements within that anti-diagonal to multiple threads using the OpenMP `for` directive. The tiling version divides the matrix into tiles like the GPU tiling version. It iterates through the tile-diagonal and assigns all the tiles within that tile-diagonal to multiple threads using OpenMP `for` directive.

Figure 4.1 shows the computational time of the naive implementation for a single comparison. The x axis is the number of OpenMP threads assigned to the kernel and the y axis is the computation time of the kernel. The time is recorded by setting a timer before the kernel is launched and after the kernel is finished. The number of threads per kernel can affect the execution time. Assigning more threads to a kernel leads to better performance until the total number of threads exceeds the total number of cores which is 60. The best number of threads per kernel is 60 which means threads are distributed to 60 cores. Problems with bigger size benefit more as the number of threads

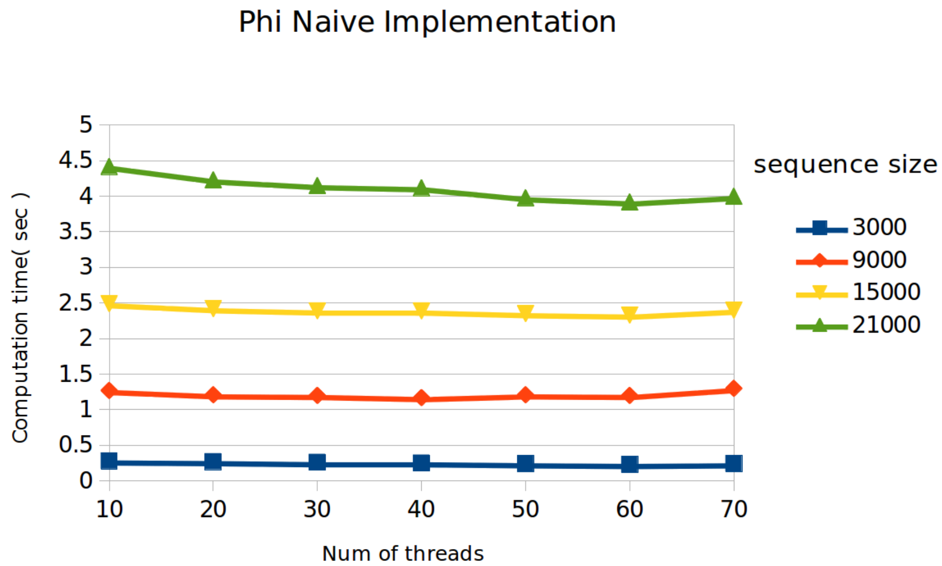

Figure 4.1: Naive implementation in the native mode for a single comparison

increases. We also notice that the run time increases when more than 60 threads are used. This suggests that although the Xeon Phi core is designed to use hyper-threading, this naive implementation does not benefit from it.

Figures 4.2, 4.3, 4.4, and 4.5 show the relationship between the kernel time and the number of threads per kernel in the tiling implementation for a single comparison. As discussed before, the tiling implementation divides the scoring matrix into tiles. During each iteration, the tiles within the current tile-diagonal are assigned to multiple OpenMP threads by the OpenMP loop. Given a scoring matrix of `matrix_width * matrix_width` and the tile width of tile_width, the longest tile-diagonal contains `matrix_width / tile_width`

tiles. If there are `matrix_width / tile_width` OpenMP threads, each OpenMP thread will be assigned to a single tile. The computation of tiles can be fully parallelized in this situation. If there are less threads than `matrix_width / tile_width`, the OpenMP for directive will map each thread to multiple tiles. For example, if there are `0.5 * matrix_width / tile_width` OpenMP threads, each thread will be assigned two tiles. The two tiles will be computed by the thread sequentially.

If we use a large tile size, the number of tiles within a tile-diagonal is small. This means that each thread will have more work to do and the total number of OpenMP threads we need to achieve fully parallel tile processing is less. On the contrary, if we use a small tile size, then the number of tiles goes up. Each thread will have less work to do but we need more OpenMP threads to achieve full parallelism.

Figure 4.2 shows that the best number of OpenMP threads for a single comparison of sequence length 3000 is 10 when the tile size is $10 * 10$. It also shows the best number of thread combined with tile width of 100 is 30. This is because $3000/100 = 30$ tiles and $3000/300 = 10$tiles. In both situation, the number of threads is equal to the number of tiles of the longest tile-diagonal. Tiles are perfectly assigned to different threads so all the tiles can be processed in parallel. Giving a task a smaller number of threads than the number of tiles in its scoring matrix's longest tile can hurt the performance

because some threads will have to process more than one tile and this is done sequentially. Giving a task more threads than needed may hurt the performance because the Xeon Phi may need to spent more time to manage these unnecessary threads.
Figures 4.3, 4.4, and 4.5 all suggest that the best tile size is $300 * 300$. It is large enough for better cache use and also not too large for the processing capacity of a single Xeon Phi thread. With the tile width of 300, the best number of threads for a task of sequences length of 9000 is 30. The best number of threads for a task with size of 15000 is 50. The best number of threads for the 21000 case is 70. We noticed that $9000/300 = 30$, $15000/300 = 50$, and $21000/300 = 70$. In conclusion, theoretically and empirically, given a fixed tile width of `tile_width`, the best number of threads for a single comparison is `matrix_width/tile_width`.

After getting the best combinations of number of threads and tile_width for single comparisons, we applied them to multiple comparisons. As mentioned in Chapter 3, we organize tasks into groups and only send one group to the Xeon Phi at one time. The total amount of memory needed by a group of tasks should not exceed the available memory on Xeon Phi. After some experiments, we found that the maximum number of tasks per group for the 3000 case is 160. For the 9000 case, the maximum number of tasks we can send to the Xeon Phi at one time is 17. For the 15000 case, the number is 6 and for the 21000 case the number is 3.

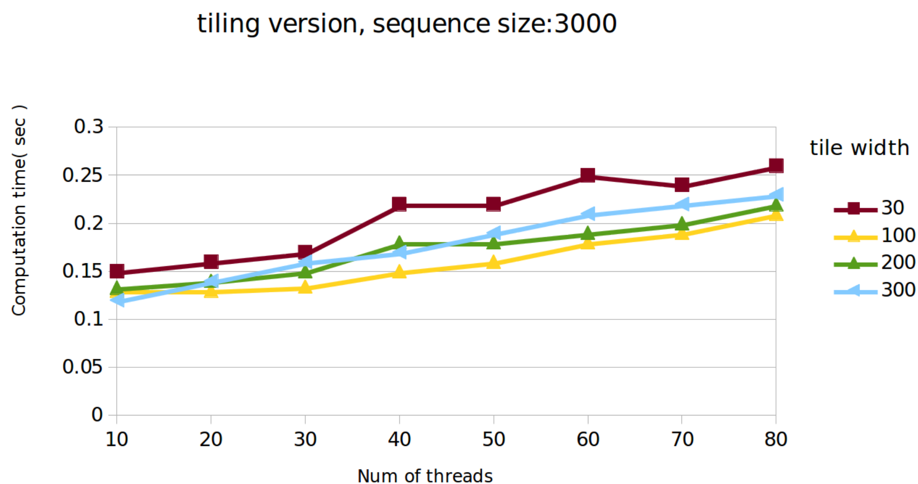

Figure 4.2: Tiling implementation in the native mode for a single comparison of length 3000

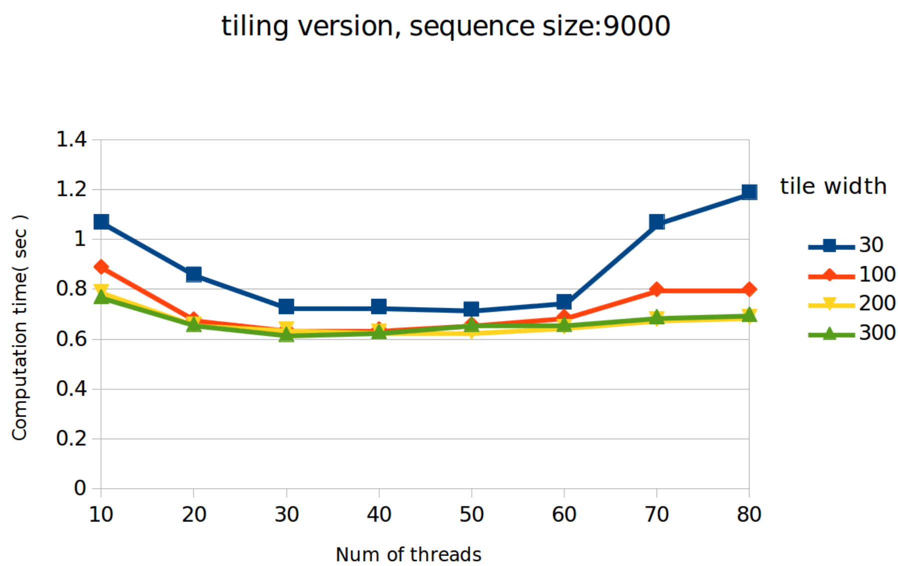

Figure 4.3: Tiling implementation in the native mode for a single comparison of length 9000

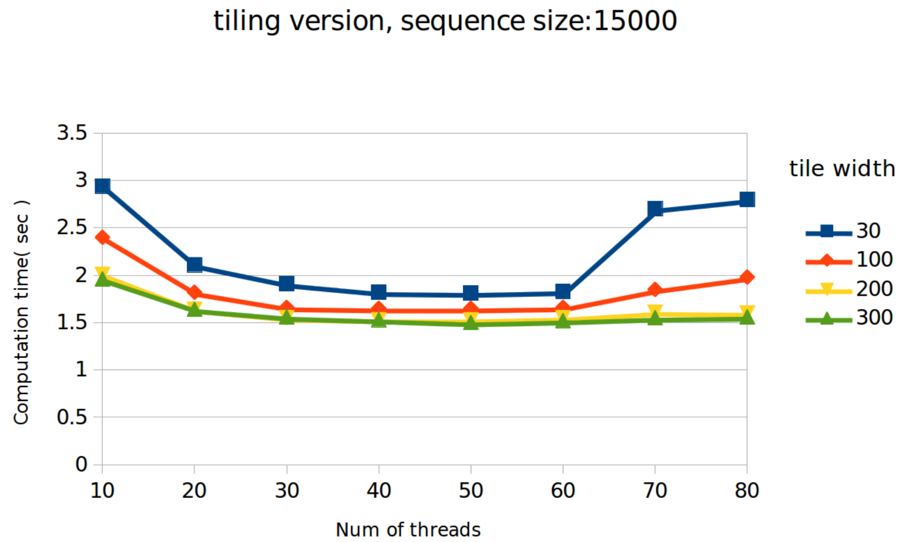

Figure 4.4: Tiling implementation in the native mode for a single comparison of length 15000

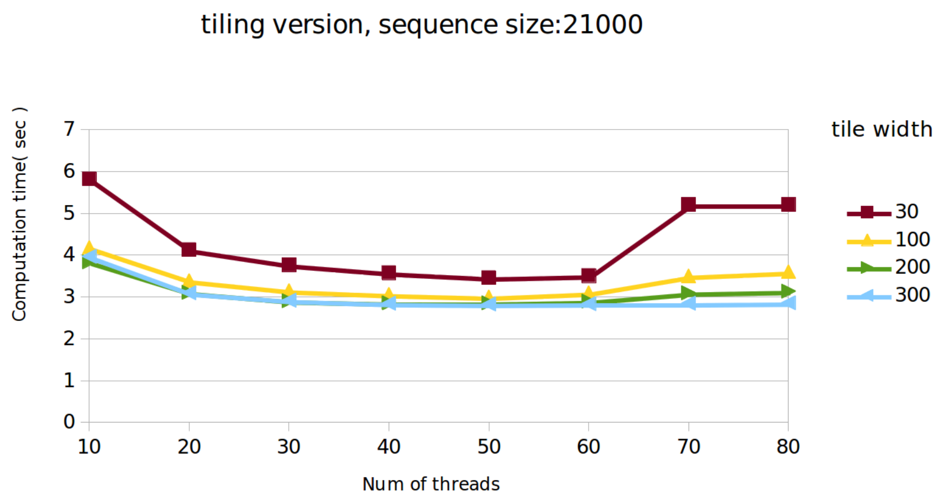

Figure 4.5: Tiling implementation in the native mode for single comparison of length 21000

Figures 4.6, 4.7, 4.8, 4.9, 4.10 show the run time of multiple comparisons on the Xeon Phi. The number of threads per task and the tile size are set to be the ones which gave best performance in the single comparison experiment above.

Figures 4.6 and 4.7 show the execution time of multiple tasks of sequence length of 3000. The x axis is the total number of tasks and the y axis is time in seconds. The computation time means the time between the point when the tasks are launched on the Xeon Phi and the point when all the tasks have been finished on the Xeon Phi. The runtime includes the data transfer time for the output. Ideally the time would be almost constant as the number of tasks increased if everything is done in parallel. In the smaller scale case, the computation time increases by a factor of 3.33 when the number of tasks increase from 1 to 10 while the run time goes up by a similar factor of 2.48. It looks like the growth rate is relatively low when the total number of tasks is small. However, in the larger scale case, the computation time increases by a factor of 9.6 when the number of tasks increase from 10 to 100 while the run time goes up by a similar factor of 9.4. This growth rate is relatively high.

In figure 4.8, when the number of tasks increases from 1 to 10, the run time grows in a factor of 6 and the computation time goes up by a factor of 4.

Figure 4.9 and 4.10 shows that as the problem size goes up, we can send fewer tasks to the Xeon Phi at one time. There is a leap in both run time and kernel time every 6 tasks in figure 4.9 and there is also a leap every 3tasks in figure 4.10. This is because the memory is limited on the Xeon Phi. In both cases, when the number of tasks increases from 1 to 10, the run time grows in a factor of 6 and the computation time goes up by a factor of 4.

We can get some important information from the above results. First, the time spent in sending and receiving data is much higher than the actual computing time. For a task of sequences of length n, the space complexity of the data we send to the Xeon Phi is only $O(n)$ but the complexity of the data(the scoring matrix) we need to receive from the Xeon Phi is $O(n^2)$. For example, the 21000 case has a scoring matrix of $21000 * 21000 * 4/1024^3 = 1.64GB$. Second, the performance of concurrent kernel execution on Xeon Phi is also getting worse when we are sending too many small task to the Xeon Phi.

Figure 4.7 shows very bad performance because we were sending too many tasks to the Xeon Phi at one time so more than 240 threads are used. How we send the tasks of sequence 3000 was then reconsidered. In later experiments, instead of sending as many tasks at one time to the Xeon Phi as the Xeon Phi's memory can handle, we reduced the group size. Each group

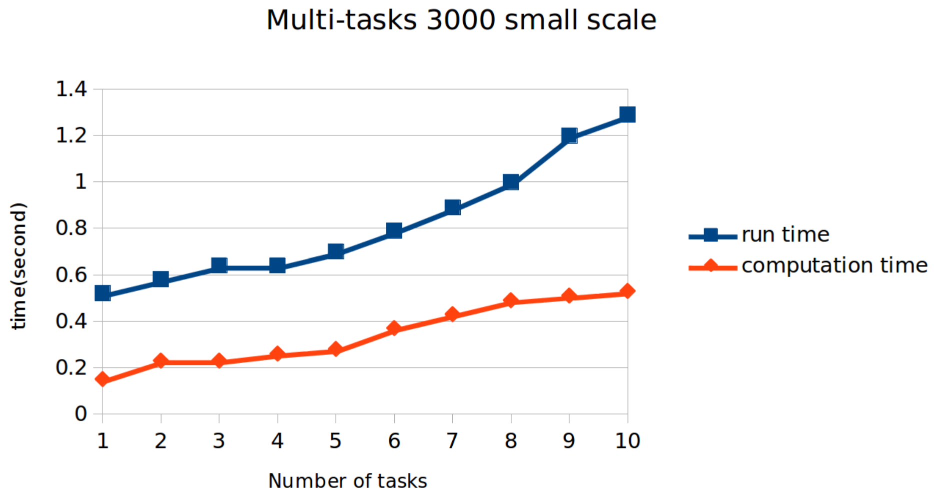

Figure 4.6: Multiple comparisons on the Xeon Phi, 3000 case

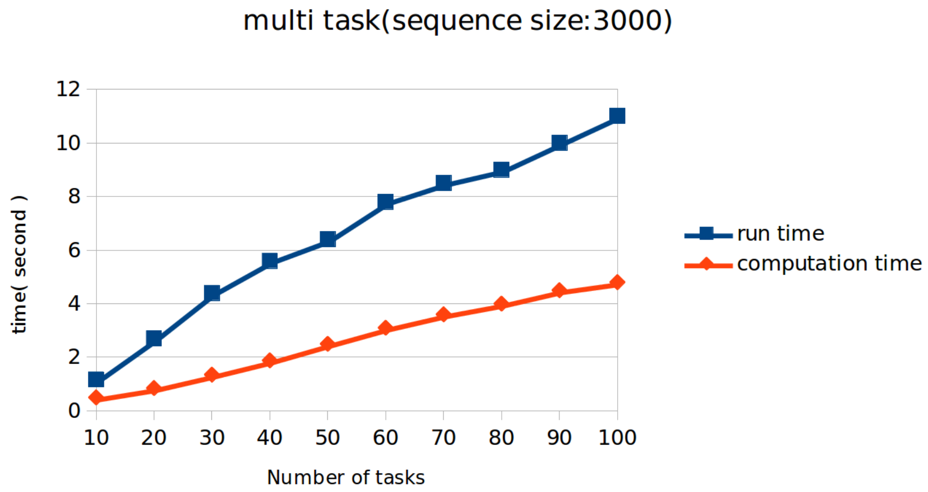

Figure 4.7: Multiple comparisons on the Xeon Phi, 3000 case

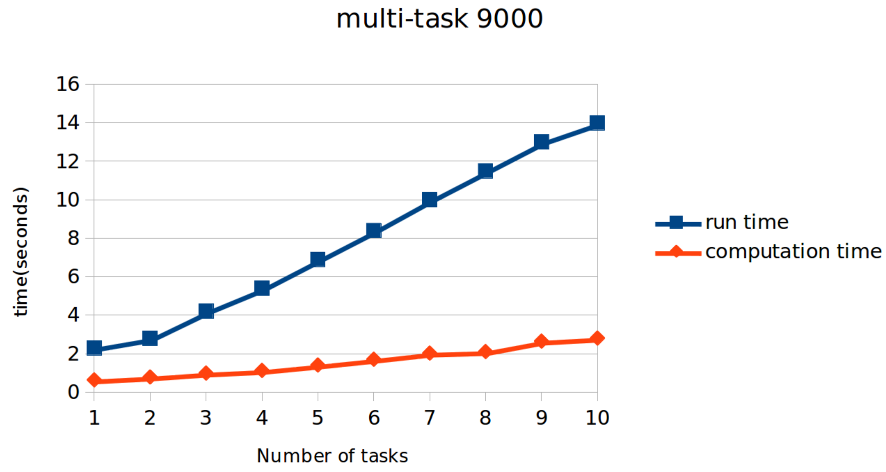

Figure 4.8: Multiple comparisons on the Xeon Phi, 9000 case

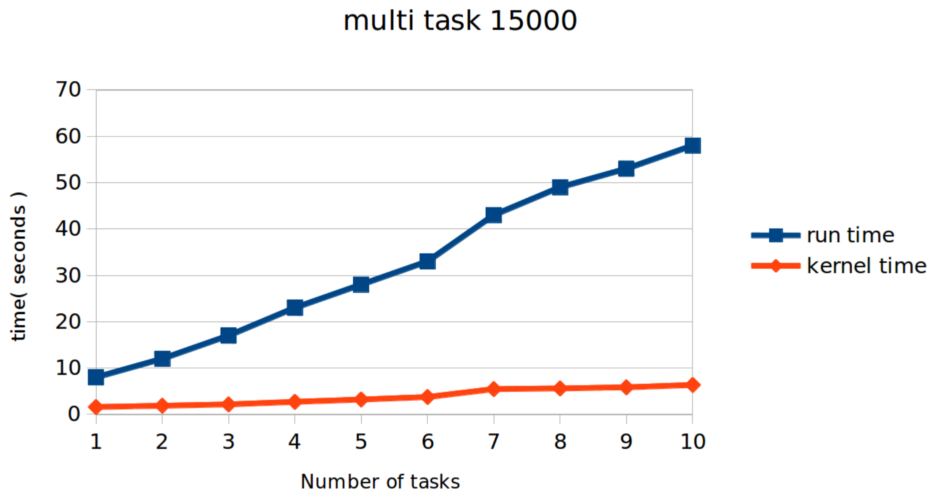

Figure 4.9: Multiple comparisons on the Xeon Phi, 15000 case

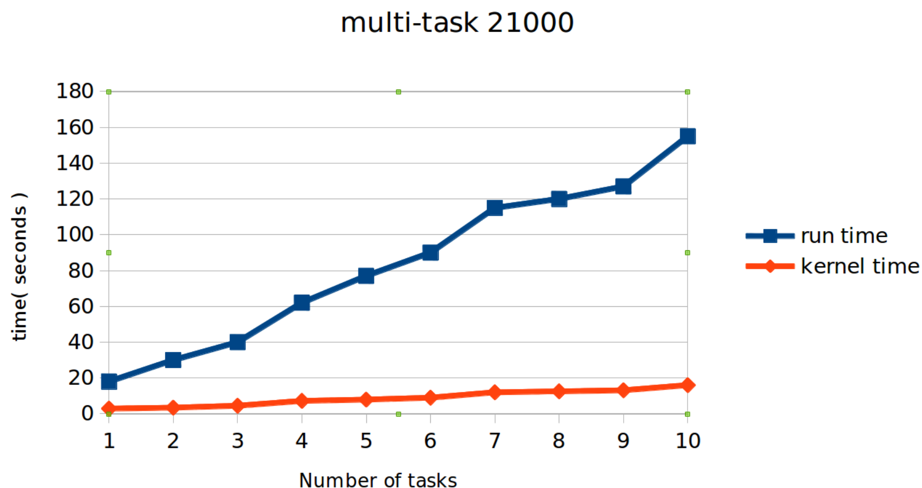

Figure 4.10: Multiple comparisons on the Xeon Phi, 21000 case

contains less tasks and we need to send more groups one after another sequentially. Although more time is spent on the overhead of sending more groups sequentially, this method actually gives better performance, as seen below.

Figure 4.11 shows how group size and number of threads assigned to each task can affect the computation time of the multiple sequence comparison. The total number of tasks is 100 and the tile width is 300. Each task is a comparison between two sequences of size 3000. The x axis is the number of tasks per group. For example, if the group size is 50, then the first group will be sent to the Xeon Phi. After all the tasks in this group have been finished by the Xeon Phi, the next group can be sent. The best performance is

achieved when there are 20 tasks per group and each tasks is assigned with 5 threads. In this situation, the tasks are organized into 3 groups. The first two groups have 40 tasks and the third group has 20 tasks. The interesting thing is that although the best number of threads for a single comparison of size 3000 is 10, 5 threads per task can achieve better performance when multiple tasks are running concurrently on Xeon Phi. Because the total number of threads available in the Xeon Phi is 240 and we can reduce the number of threads for all the tasks by reducing the number of threads assigned to one task. In this case, the maximum number of threads requested by a group is $5 * 20 = 100$. It is much smaller than the maximum 240 threads so all the tasks within a group can run concurrently on the Xeon Phi. Figure 4.12 for sequences of length 9000 shows that the best number of threads per tasks is 10, not 30, when each group contains 10 tasks. This means that the threads on the Xeon Phi are critical resources. When sending multiple tasks to the Xeon Phi, we need to make sure these tasks should not over ask for the threads. Performance gain can be made by giving fewer threads to each task and making sure these tasks can run concurrently. Figure 4.13 for sequences of length 15000 shows that the best number of threads per tasks is 10 but we can send as many tasks of size 15000 as the Xeon Phi has memory, which is 6. Figure 4.14 also suggest for tasks of size 21000, we can just send 3 at a time and the best number of threads in this case is 30. Figures 4.15, 4.16, 4.17, and 4.18 suggests that the total run time favours these arrangements in the same way. With the mechanism of grouping and assigning fewer threads to a

task, the kernel time for 100 tasks of size 3000 reduces from 10.55 seconds to 2.9 seconds. The run time reduces from 14.1 second to 9 seconds. For the 20 tasks of size 9000, the kernel time drops from 5.7 seconds to 2.85 seconds and the run time drops from 29 seconds to 27 seconds. So for tasks of size 3000 and 9000, we recommend to send 10 tasks per group and use fewer threads per task. That is 5 threads per task for the 3000 case and 10 threads per task for the 9000 case. For the 6 tasks of size 15000, the kernel time drops from 4.4 seconds to 2.8 seconds when we change the number of threads per kernel from 50 to 10. For the 3 tasks of size 15000, the kernel time drops from 4.9 seconds to 4.25 seconds when we change the number of threads per task from 70 to 30. For both case, we recommend to send as many tasks as the Xeon Phi's memory can take. That is 6 per group for the 15000 case and 3 per group for the 21000 case.

In practice, there may be tasks of different sequence length. How we schedule these tasks can also affects on the performance. Basically we have two ways: big tasks first and small tasks first. Figure 4.19 shows the case when we try to send 56 tasks to the Xeon Phi. There are 30 tasks of size 3000, 20 tasks of size 9000, 4 tasks of size 15000, and 2 tasks of size 21000.

According to the above experimental results, tasks of size of 3000 sent to the Xeon Phi at one time should not exceed 10. Tasks of size of 9000 sent at one group should not exceed 10 either. For tasks of size 15000 and 21000, we can

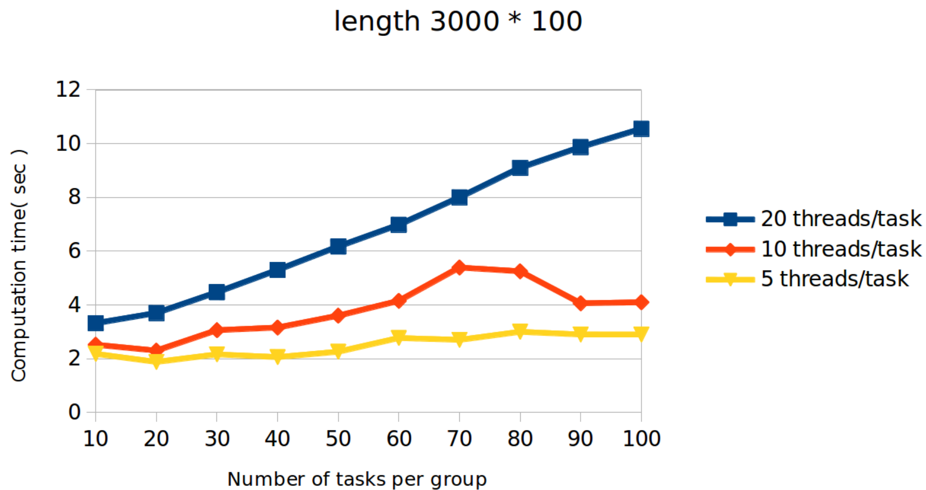

Figure 4.11: Reduced group size for 100 comparisons of size 3000, computation time

send the maximum number Xeon Phi's memory can take, which are 6 tasks of 15000 and 3 tasks of size 21000.

We also did an experiment to decide the best number threads per tasks in concurrent execution situation. We've found that for tasks of size 3000, the best number of threads per tasks should be 5 instead of 10 as long as the number of tasks of size of 3000 is equal or greater than 5. For the 9000 case, the best number of threads per tasks should be 10 instead of 30 as long as the number of tasks is equal or greater than 5. For the 15000 case, the best number of threads per tasks should be 10 instead of 50 as long as the number of tasks is equal or greater than 3. For the 21000 case, the best number of threads per tasks should be 30 instead of 70 as long as the number of tasks

is equal or greater than 2.

Based on all the information above, the big first schedule organizes these 56 tasks into 7 groups. Group 0 contains 2 tasks of size 21000 and 2 tasks of size 15000. Group 1 contains 2 tasks of size 15000 and 8 tasks of size of 9000. Group 2 contains 10 tasks of size 9000. Group 3 contains 2 tasks of size 9000 and 8 tasks of size 3000. Group 4 and 5 both contain 10 tasks of size 3000. Group 6 contains the last 2 tasks of size 3000.

The small first schedule organizes tasks into 7 groups. Group 0, 1, and 2 each contains 10 tasks of size 3000. Group 3 and 4 both contain 10 tasks of size 9000. Group 5 contains 4 tasks of size 15000 and 1 task of size 21000. Group 6 contains the last 1 task of size 21000.

Both of these two schedules send groups one by one to the Xeon Phi. Figure 4.19 shows both the computation time and the run time. The big first schedule is 66% faster than the small first schedule in computation time and 17.4% faster in run time. In the big first schedule, bigger tasks are loaded to the group first. Sometimes there is some room left for smaller tasks. In this case, the group 0 still has room for 2 tasks of size 15000 after loading 2 tasks of size 21000. Both the total threads and total amount of memory needed by group 0 doesn't exceed the Xeon Phi's limit. The similar situation occurs in group 1 and 3 too. This helps to utilize the Xeon Phi better. However,

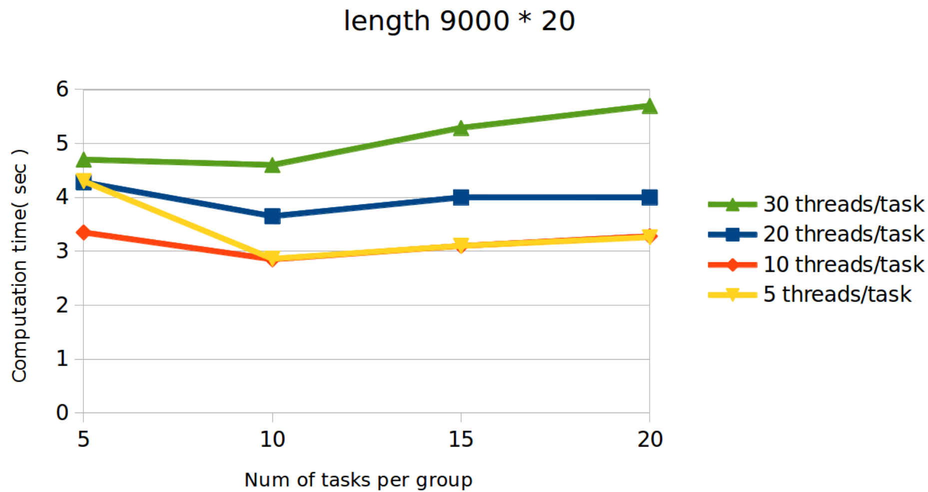

Figure 4.12: Reduced group size for 20 comparisons of size 9000, computation time

this doesn't happen in the small first schedule because small tasks tend to fill the whole group and there is no room for large tasks to fit.

4.2.2 The Offload Experiment

The offload implementation follows a similar structure used in the host program of the socket implementation. It also organizes tasks into groups. One group will not be offloaded to the Xeon Phi unless the current group has been finished. It creates host threads for each task within the current group and each thread offloads its own task to the Xeon Phi. We found that the offload mode has much more overhead than our socket implementation. Figure 4.20 shows the total runtime and the average computation time for the

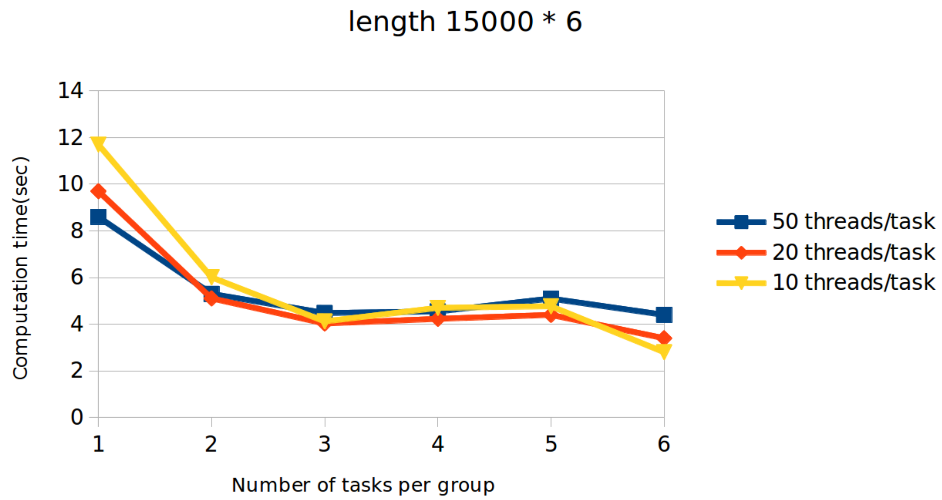

Figure 4.13: Reduced group size for 6 comparisons of size 15000, computation time

Figure 4.14: Reduced group size for 3 comparisons of size 21000, computation time

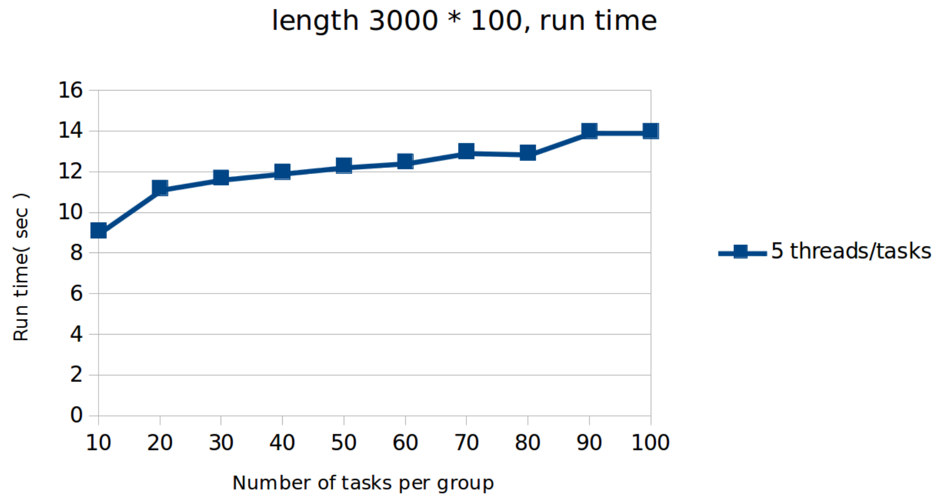

Figure 4.15: Reduced group size for 100 comparisons of size 3000, run time

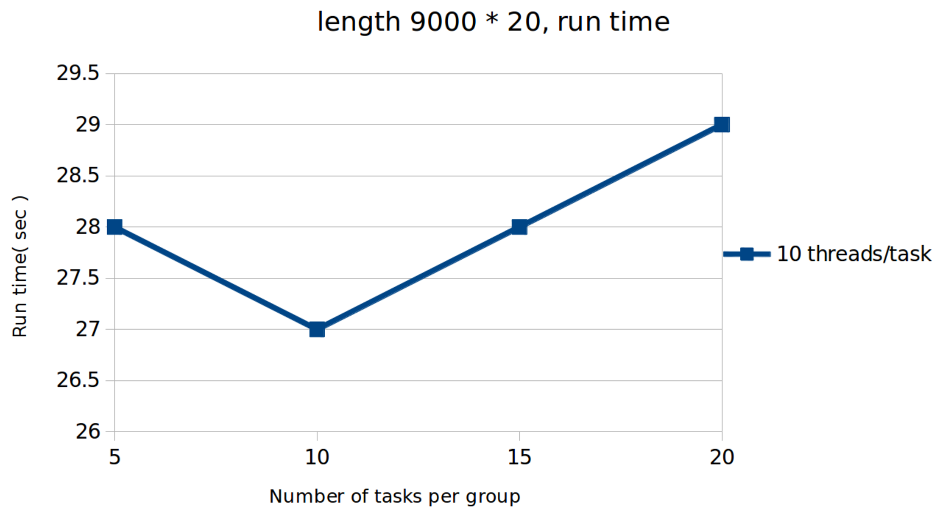

Figure 4.16: Reduced group size for 20 comparisons of size 9000, run time

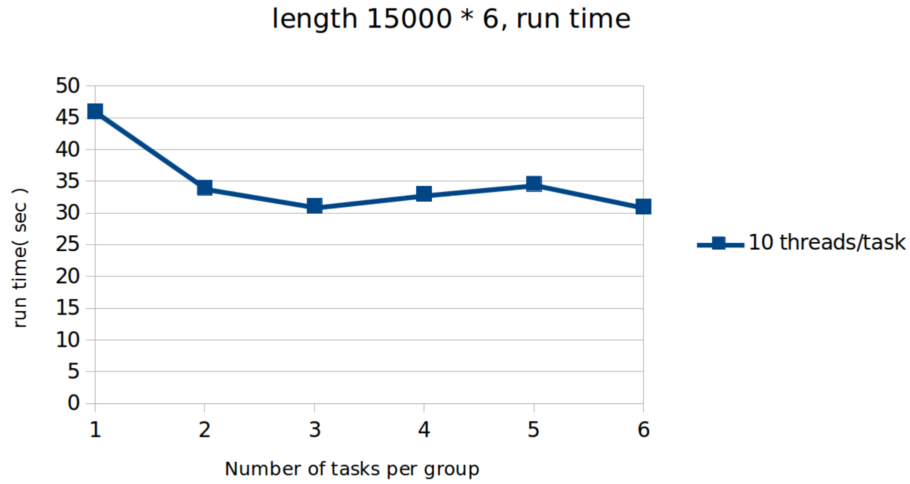

Figure 4.17: Reduced group size for 6 comparisons of size 15000, run time

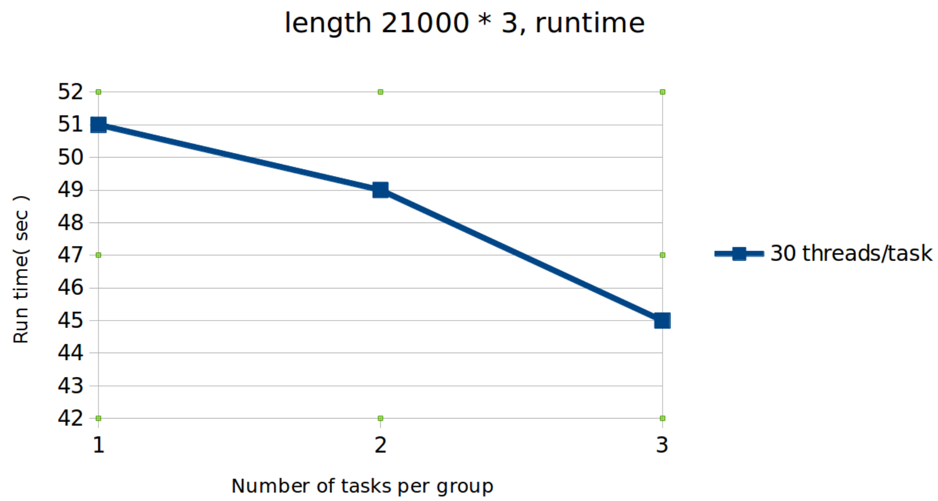

Figure 4.18: Reduced group size for 3 comparisons of size 21000, run time

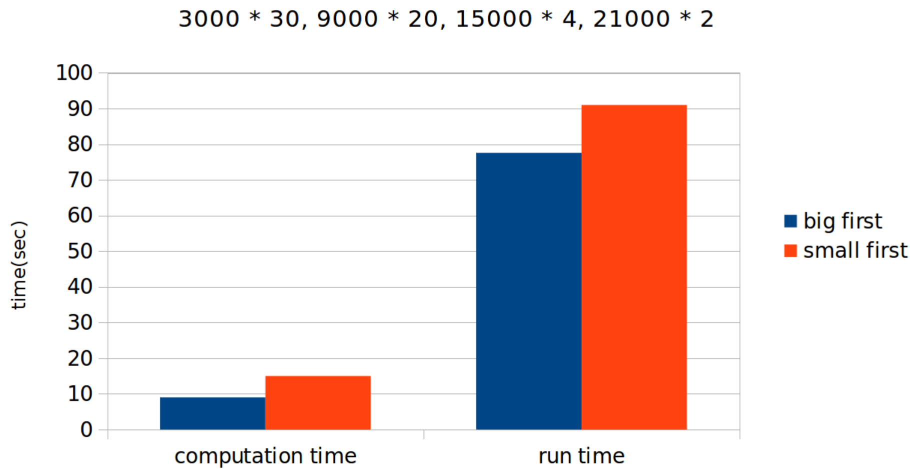

Figure 4.19: 30 tasks of 3000 length, 20 task of 9000 length, 4 task of 15000 length and 2 tasks of 21000 length mixed together

offloaded kernel. The tile width is 100 and the number of threads per kernel is 20. Comparing to figure 4.2, we can find that the same kernel runs 22% slower in offload mode when it runs in native mode. Comparing to figure 4.7, the total runtime for 10 tasks is almost 90% slower than our implementation. This suggests that the offload mode has more overhead than our socket implementation.

4.2.3 The GPU Experiment

We also performed several experiments on the GPU. First we tried to find the best tile size for the single comparison. Because the number of threads per block is equal to the width of the tile and the number of blocks is equal to the number of tiles in our tiling implementation, the tile size is the only

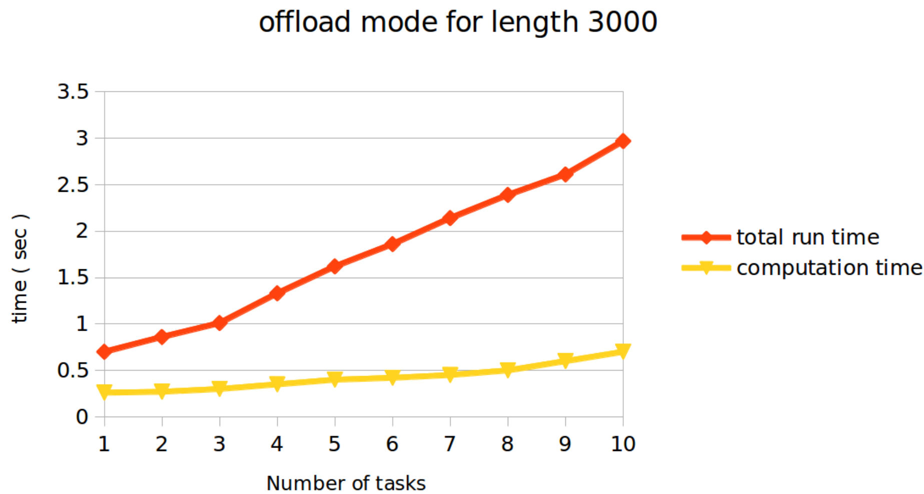

Figure 4.20: Offload implementation for sequence size of 3000

parameter that can affect on the performance of a single comparison. After several experiments, we found that a tile size of $300 * 300$ is the best for all four sequence lengths.

Figure 4.21 shows the run time and kernel time of the GPU tiling version for single comparisons. We can find that the GPU version is much faster than the Xeon Phi version.

We then performed experiments for multiple comparison. Figure 4.22, 4.23, 4.24, and 4.25 show the run time and kernel time for different sequence length. We can find that the kernel time grows slowly in a constant rate as the number of tasks increased. The run time grows faster. Figure 4.22

shows the situation when we send a large number of small tasks to the GPU. Because only the default stream handles the memory copy, the time spent in copying data between the host and GPU can not be parallelized at all. But still GPU is 10 time faster than the Xeon Phi for length 3000. From figure 4.23 and 4.24 we can see the kernels are running concurrently on the GPU while there is still a little overhead of doing so. The run time spent in copying the scoring matrix back to the host keeps high but it is necessary in this application. Figure 4.25 shows that there can be no more than 2 tasks of length 21000 running concurrently on the GPU due to the memory. This number is 3 on the Xeon Phi. This is because the K20 has less memory than the Xeon Phi.

All the results suggest that the GPU version works much better than the Xeon Phi version when we apply the same design to them. The kernel runs faster on the GPU than on the Xeon Phi. Also the GPU has a set of well designed mechanisms to enable concurrent tasks unlike the Xeon Phi. The time spent in sending data through sockets to the Xeon Phi is also much longer than that on the GPU. The communication time is equal to `run time - kernel time`. We found in most of our experiments, the communication time for a GPU implementation is less or a little higher than 1 second, while the communication time of Xeon Phi implementations are always higher than 1 second. For example, the run time for 10 tasks of 9000 can be as high as $11.5(14 - 2.5)$ seconds while the GPU only need $0.49(0.77 - 0.28)$ second.

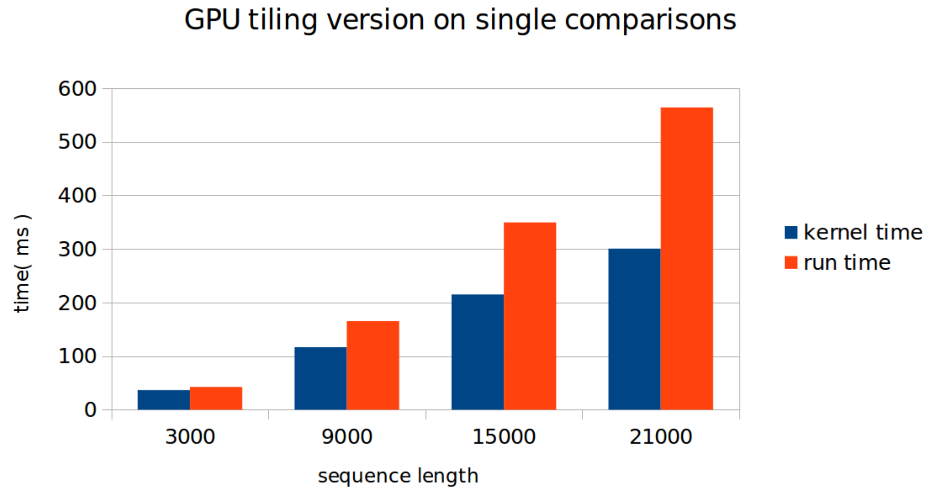

Figure 4.21: The run time and kernel time of GPU tiling version for single comparisons

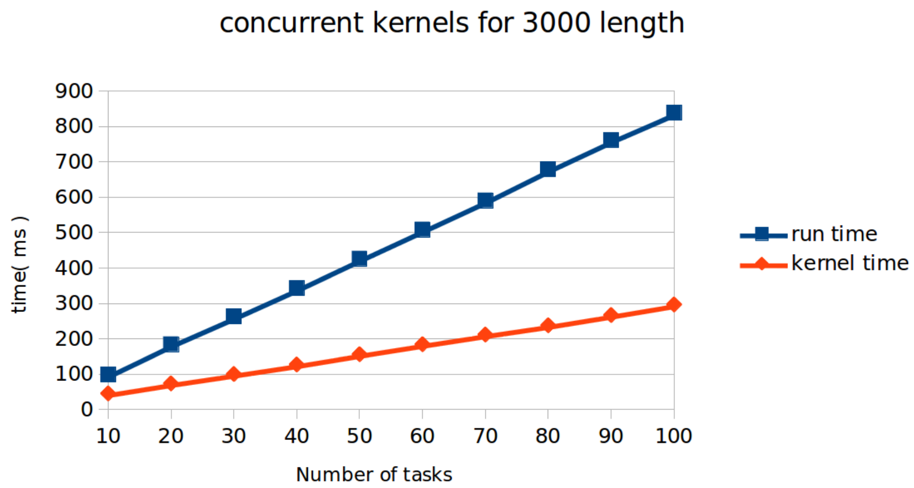

Figure 4.22: The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 3000

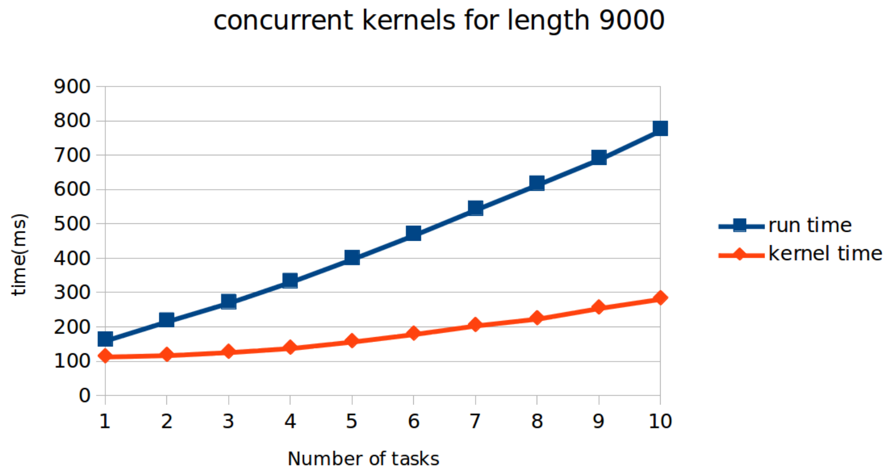

Figure 4.23: The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 9000

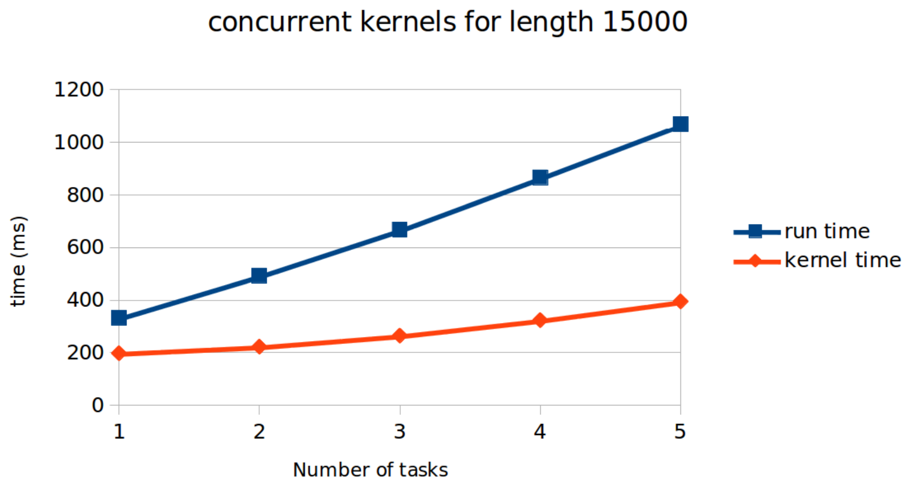

Figure 4.24: The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 15000

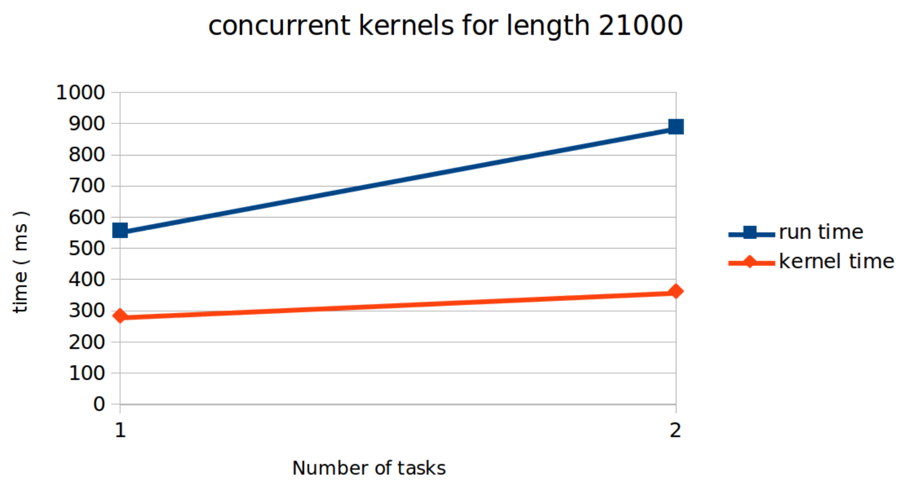

Figure 4.25: The run time and kernel time of GPU concurrent tiling kernels for comparison of sequence length 21000

Chapter 5

Conclusion

In this thesis, we discussed the differences between the architectures of the GPUs and the Xeon Phis. We then introduced the wave-front pattern and the Smith-Waterman algorithm used in sequence alignment problems. We then implemented two versions of single sequence comparison application based on the Smith-Waterman algorithm for the GPU and the Xeon Phi. To improve their performance, we applied the tiling method to both of them. In the GPU version, we also developed a new mechanism for using the shared memory, which stores only three anti-diagonal in the shared memory instead of the whole tile. This helps us reduce the memory use from $O(tileWidth^2)$ to $O(tileWidth)$ to enable larger tile sizes. The Xeon Phi version also has the tiling method but not this 3-diagonal mechanism because there is no shared memory available. We also found the best combination of tile size and number of threads for a single comparison on the Xeon Phi. The GPU

version works 5 to 8 times faster than the Xeon Phi version. This suggests that for the Smith-Waterman application, the GPU works better than the Xeon Phi because the GPU organizes the threads into blocks and the blocks can be easily mapped to the tiles in this application. Threads of the same block can compute the whole tile together efficiently. However, on the Xeon Phi, threads works more independently, communications between threads are not well designed like the communications between threads within the same group in the GPU. We have to assign more work to each thread on the Xeon Phi, and try to make the best use of the vector unit. However, the use of shared memory on the GPU makes the GPU much faster than the Xeon Phi's vector unit in this application.

We then implemented multiple sequence comparison for both devices. On the GPU, we used the well established CUDA stream. We send different tasks to different CUDA streams. Because the CUDA stream is designed by Nvidia for concurrent kernel execution, tasks can run concurrently on the GPU as long as there are enough resources. However, on the Xeon Phi, because Intel didn't provide such efficient design for concurrent tasks, the performance is bad. In the offload implementation, we used the offload directives provided byIntel. Our socket implementation performed much better than the offload mode both in runtime and computation time. However, the Xeon Phi implementation is still not as good as the GPU version. The `cudaMemCpy` works more efficiently than copying data through sockets. Using multiple

host threads to perform multiple offload results in bad performance. So we designed a new socket mechanism to send multiple tasks to run natively on the Xeon Phi. By organizing tasks into groups and reducing the number of threads per task when running multiple tasks, we greatly reduced the computation time for concurrent task execution. The computation time suggests that concurrent tasks execution works well on the Xeon Phi as long as there are enough threads on the Xeon Phi. The run time suggests that we need to avoid large size data transfers between the host and the Xeon Phi. In this particular application, a good solution could be to do the back trace of the scoring matrix on the Xeon Phi. Because the back tracing process has to be done in a sequential way, the back tracing process can be done easily on the Xeon Phi while it is hard and inefficient on the GPU.

The socket implementation itself gives a new way of utilize the Xeon Phi. For applications which suit Xeon Phi better than the GPU, this socket communication mechanism can help the user to send more tasks to the Xeon Phi concurrently. Our API is not very loosely coupled. Building a loosely coupled and open API for this socket communication mechanism is suggested future work.

Bibliography

[1] The Intel Xeon Phi Coprocessor Developer's Quick Start Guide. 2013.

[2] Jim Jeffers, James Reinders. Intel Xeon Phi Coprocessor High-performance Programming(2013). Morgan Kaufmann Publishers.

[3] T. Li, V. K. Narayana and T. El-Ghazawi. A static task scheduling framework for independent tasks accelerated using a shared graphics processing unit. IEEE 17th International Conference on Parallel and Distributed Systems (ICPADS), 2011.

[4] F. Wende, F. Cordes and T. Steinke. On improving the performance of multi-threaded CUDA applications with concurrent kernel execution by kernel reordering. Symposium on Application Accelerators in High Performance Computing (SAAHPC), 2012.

[5] S. Xiao, A. M. Aji and W. Feng. On the robust mapping of dynamic

programming onto a graphics processing unit. 15th International Conference on Parallel and Distributed Systems (ICPADS), 2009.

[6] Wavefront pattern, by Mark Snir.
http://www.cs.uiuc.edu/homes/snir/PPP/patterns/wavefront.pdf

[7] Wenzhen Zhou, MCS thesis, Wavefront Pattern for Dynamic Programming on GPUs, University of New Brunswick, 2013.

[8] Nvidia Kepler White Paper, 2012.

[9] L. Ligowski and W. Rudnicki. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. IEEE International Symposium on Parallel & Distributed Processing, 2009.

[10] T. K. Yap, O. Frieder and R. L. Martino. Parallel computation in biological sequence analysis. IEEE Transactions on Parallel and Distributed Systems, 9(3), pp. 283-294. 1998.

[11]Y. Liu, D. L. Maskell and B. Schmidt. CUDASW++: Optimizing smith-waterman sequence database searches for CUDA-enabled graphics processing units. BMC Res. Notes 2, pp 73-0500-2-73, 2009.

[12] F. Ino, Y. Kotani, Y. Munekawa and K. Hagihara. Harnessing the power of idle GPUs for acceleration of biological sequence alignment. Parallel Processing Letters 19(04), pp. 513-533. 2009.

Vita

Candidate's full name:
Yucheng Zhu
University attended:
University of New Brunswick, Master of Computer Science, 2012 to 2014
Nanjing University of Aeronautics and Astronautics, Bachelor, 2006 to 2010
Publications:
Conference Presentations: