Dynamic Programming on Multicore processor

By

Mitul Takodara

10MCEC18

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

March 2012

Dynamic programming on Multicore Processor

Major Project

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science and Engineering

By

Mitul Takodara

(10MCEC18)

Guided By

Dr. S.N. Pradhan

Guided By

Prof. Samir B. Patel

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

AHMEDABAD-382481

March 2012

Certificate

This is to certify that the Major Project entitled "Dynamic Programming on Multicore Processor" submitted by Mitul Takodara (10MCEC18), towards the partial fulfillment of the requirements for the degree of Master of Technology in Computer Science and Engineering of Nirma University, Ahmedabad is the record of work carried out by him under my supervision and guidance. In my opinion, the submitted work has reached a level required for being accepted for examination. The results embodied in this major project, to the best of my knowledge, haven't been submitted to any other university or institution for award of any degree or diploma.

Prof. Samir B. Patel Guide, Associate Professor, Department of C.S.E., Institute of Technology, Nirma University, Ahmedabad.	Dr.S.N.Pradhan Professor and PG-Coordinator, Department of C.S.E, Institute of Technology, Nirma University, Ahmedabad.
Prof.D.J.Patel Professor and Head, Department of C.S.E, Institute of Technology, Nirma University, Ahmedabad.	Dr.K.Kotecha Director, Institute of Technology, Nirma University, Ahmedabad.

Abstract

The strong need for increased computational performance in science and engineering has led to the use of heterogeneous computing, with GPUs, acting as coprocessors to the CPUs for arithmetic intensive data-parallel workloads. CUDA - Compute Unified Device Architecture is a new industry standard for task-parallel and data-parallel heterogeneous computing on NVIDIA GPUs. Basic goal of CUDA is to help programmers focus on the task of parallelization of the algorithms rather than spending time on their implementation. Key to performance on this platform is using massive multithreading to utilize the large number of cores and hide global memory latency. The main objective of the thesis is to obtain the performance gain in execution speed for the dynamic algorithms which generally are complex and takes a very long time for execution and compare results on different gpu processors and CPU and have a comparative study of algorithms. It will require running the CUDA C code in sequential and parallel on GPU consisting of hundreds of core or even more. Also the algorithms C code may require removing dependencies. Hence obtaining all the statistics of various algorithms and achieve performance gain in execution.The contributions of this thesis include a programming language approach to providing transformation abstraction and composition, a unifying framework for general and GPU specific transformations, and demonstration of the framework on standard benchmarks that show it capable of matching or outperforming hand-tuned GPU kernels. This thesis work is mainly concentrated on the computational part of the source code and its optimization. Report contains study of the NVIDIA GeForce GPU architecture, CUDA SDK tool kit, Dynamic Algorithms and different methods to get performance benefit, implementation of intermediate tool to find out functions and their dependencies from the source code and the implementation of the complete algorithm, testing and Obtaining statistics for the same.

Acknowledgements

It gives me great pleasure in expressing thanks and profound gratitude in having done my research work under the efficient supervision of my esteemed guide **Dr. S. N. Pradhan**, Professor and M.Tech. Coordinator, Institute of Technology, Nirma University, Ahmedabad and **Prof. Samir Patel** Senior Associate Professor, Department of Computer Science and Engineering,Institure of Technology,Nirma University,Ahmedabad for his valuable guidance and continual encouragement throughout part one of the Major project. I am heartily thankful to him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during this study.

My sincere thanks and gratitude to **Prof. D.J. Patel**, Professor and Head, Computer Engineering Department, Institute of Technology, Nirma University, Ahmedabad for his continual kind words of encouragement and motivation throughout the Dissertation work.

I am thankful to Nirma University for providing all kind of required resources.

- Mitul Takodara
10MCEC18

Contents

List of Figures

Abbreviation Notation and Nomenclature

ALU . . . . Arithmetic Logic Units
API . . . . Application Programming Interface
CPU . . . . Central Processing Unit
GPGPU . . . . General-Purpose computation on Graphics Processing Unit
GPU . . . . Graphics Processing Unit
CUDA . . . . Compute Unified Device Architecture
MIMD . . . . Multiple Instruction, Multiple Data
SIMT . . . . Single Instruction, Multiple Threads
NVCC . . . . CUDA Compiler
PTX . . . . Parallel Thread Execution
DSP . . . . Digital Signal Processing
HPC . . . . High Performance Computing
GFLOPS . . . . Giga Floating Point Operation Per Second

Chapter 1

Introduction

1.1 General

In the history of microprocessors, the Central Processing Unit (CPU) processor has been the focus of the industry for its powerful ability to run general purpose sequential programs. While hugely successful in meeting the majority of computing needs, there has also been a legacy of programmable accelerators to improve performance for specific domains of applications. In the embedded world, Digital Signal Processors (DSPs) are used to encode and decode audio. In High Performance Computing (HPC), there have been many examples of coprocessors going back to the 1970s designed for floating point calculations or other specific tasks. In consumer computing, discrete video processors have long been included to meet specialized needs of rendering images at the demanding rate of stutter-free video.

Under the pressures of the consumer gaming and professional workstation market, Graphical Processing Units (GPUs) have evolved to deliver ever-increasing amounts of computational performance. Reacting from the market demand to provide more direct means of accessing this potential performance, hardware manufacturers starting providing developer SDKs to treat the GPU as a programmable stream processor, instead of a specialized device only accessible through graphics oriented fixed-function APIs. This opened to door for GPUs to be used for massively parallel computations on non graphics data. Scientific computing has had a long history of using coprocessors and programmable accelerators to serve its seemingly unbounded need for computational performance. In fact, modern high end super computers often include a hybrid of traditional processors and stream processors in the form of GPUs. Today's fastest GPUs can deliver a peak performance in the order of 500 GFLOPS, more than four times the performance of the fastest x86 quad-core processor.

1.2 GPU Hardware Architecture

Present multi-core CPUs usually consist of 2-8 cores. These cores usually work asynchronously and independently. Thus, each core can execute different instructions over different data at the same time. According to the Flynn's taxonomy, we are talking about Multiple Instruction stream, Multiple Data stream (MIMD) class of computer architectures. On the other hand, GPUs are designed for parallel computing with an emphasis on arithmetic operations, which originate from their main purpose - to compute graphic scene which is finally displayed. Current graphic accelerators consist of several multi- processors (up to 30). Each multiprocessor contains several (e.g., 8, 12 or 16) Arithmetic Logic Units (ALUs). Up to 480 processors is in total on the current high-end GPUs. Figure 1.1 shows the general overview of the CPU and GPU.

- CPU cores are designed to execute a single thread of sequential instructions with maximum speed and GPUs are designed for fast execution of many parallel instruction threads.
- CPUs use SIMD (single instruction is performed over multiple data) vector

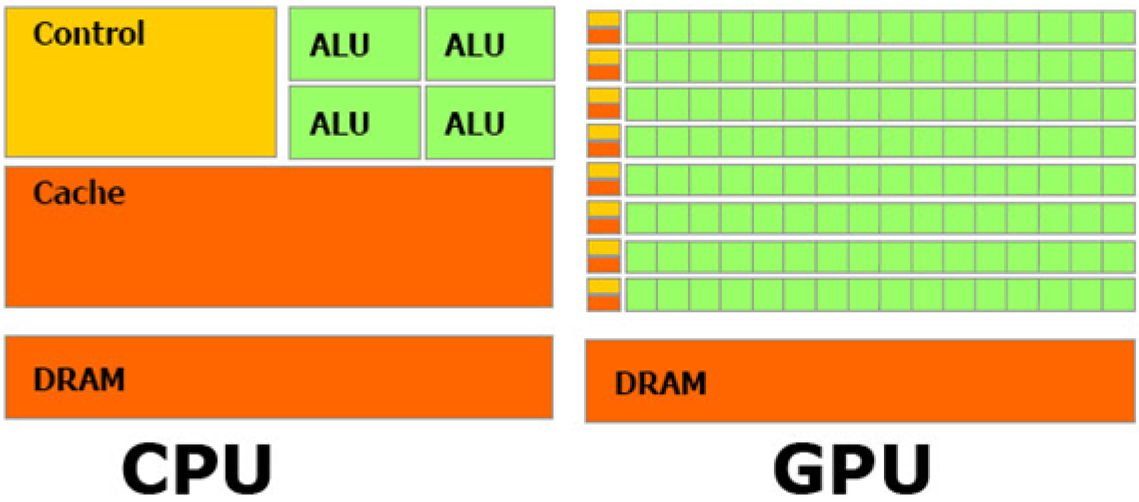


Figure 1.1: GPU Devoted More Transistor To Data Processing [1]

units, and GPUs use SIMT (single instruction, multiple threads) for scalar thread processing.

- GPUs contain extensive support of Stream Processing paradigm. It is related to SIMD (Single Instruction, Multiple Data) processing.

- CPUs use caches to increase their performance owing to reduced memory access latencies and GPUs use caches or shared memory to increase memory bandwidth.

- There exist a lot of differences in multi-threaded operations. CPUs can execute 1-2 threads per core, while GPUs can maintain up to 1024 threads per each multiprocessor. Switching from one thread to another costs hundreds of cycles to CPUs, but GPUs switch several threads per cycle.

- CPUs reduce memory access latencies using large caches as well as branch prediction. GPUs solve the problem of memory access latencies using simultaneous execution of thousands threads when one thread is waiting for data from memory, a GPU can execute another thread without latencies.

1.3 Objective of work

The objective of this research is to provide a better utilization of the resources of CUDA GPU to obtain better performance gain in execution speed of time consuming and complex dynamic algorithms and hence getting performance gain to a greater extent.

1.4 Scope of Work

The scope of this work is to optimize Processor elements and to reduce the computation time with the CUDA enabled GPU which are using Geforce architecture. Work can be extended by developing the software which may directly convert the sequential dynamic c code into parallel with removed dependencies.

1.5 Motivation of the Work

As looking at the advantages of multicore architectures are many, such as higher performance, lower power consumption lower cost and more exibility but can be realized only if the corresponding software is developed to unlock these benefits. The motivation behind doing this research is the need to reduce execution time of complex time consuming dynamic algorithms in a more efficient and optimized way for multicore architecture using CUDA, which best utilizes the available core and other resources on GPUs.

1.6 Thesis Organization

The rest of the thesis is organized as follows.

Chapter 2, *Literature Survey*, Literature survey on CUDA describes history of CUDA. It also describes CUDA Programming Model, Memory Architecture, Hardware implementation.

Chapter 3, *Peformance Optimization Strategies*, Performance Optimization Strategies describes various performance optimization strategies specific to CUDA, which are used to get the maximum utilization of available resources.

Chapter 4, *Preliminary Study*, Preliminary Study includes Study of CUDA C programming language

Chapter 5, *Problem Definition* , Problem Statement and its Proposed Approach

Chapter 6, *Implementation* , Implementation Done

Chapter 8, Future Work

Chapter 2

Literature Survey

GPGPU stands for General-Purpose computation on Graphics Processing Units, also known as GPU Computing. Graphics Processing Units (GPUs) are high-performance many-core processors capable of very high computation and data throughput.

In November 2006, NVIDIA introduced CUDA(Compute Unified Device Architecture), a general purpose parallel computing architecture with a new parallel programming model and instruction set architecture - that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. NVIDIA GPUs with the new Tesla unified graphics and computing architecture run CUDA C programs and are widely available in laptops, PCs, workstations, and servers. The CUDA model is also applicable to other shared-memory parallel processing architectures, including multicore CPUs.

GPU performance is influenced by the architectural organization of the hardware platform. NVIDIA suggests that achieving the highest GPU occupancy and optimizing the use of the memory hierarchy are the two main factors behind GPU performance. In fact, both of them are related since maximizing the occupancy can help to cover latency during global memory loads. We present several experiments aimed at analyzing their relative importance. Our results indicate that code transformations

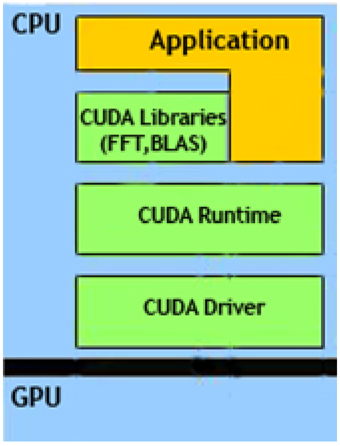


Figure 2.1: CUDA Software Stack [2]

that target efficient memory usage are the major determinant of actual performance.

2.1 NVCC Compilation

The CUDA phase converts a source file coded in the extended CUDA language,into a regular ANSI C source file that can be handed over to a general purpose C compiler for further compilation and linking. The exact steps that are followed to achieve this are displayed in Figure 2.2

2.1.1 Compilation flow

In short, CUDA compilation works as follows: the input program is separated by the CUDA front end (cudafe), into C/C++ host code and the .gpu device code. Depending on the value(s) of the -code option to nvcc, this device code is further translated by the CUDA compilers/assemblers into CUDA binary (cubin) and/or into intermediate ptx code. This code is merged into a device code descriptor which is included by the previously separated host code. This descriptor will be inspected by the CUDA runtime system whenever the device code is invoked ('called') by the

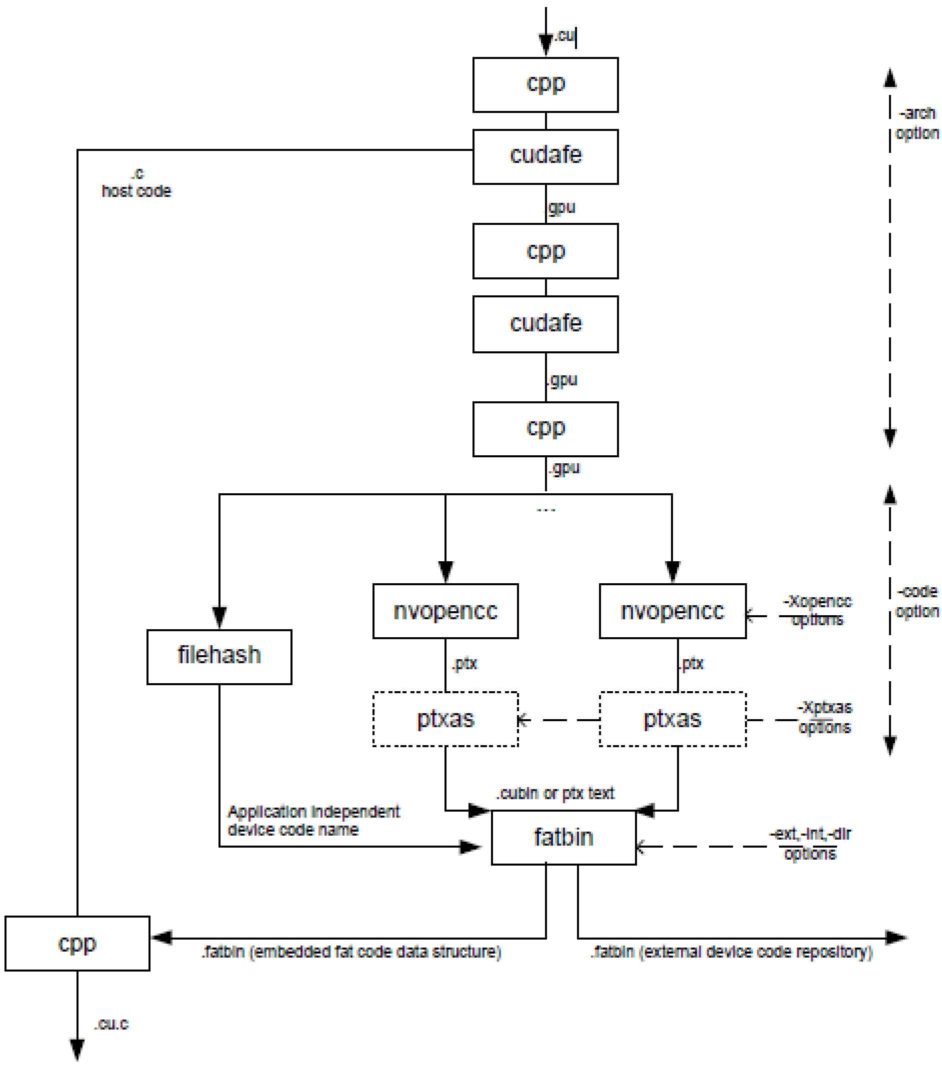


Figure 2.2: CUDA compilation from .cu to .cu.c [4]

host program, in order to obtain an appropriate load image for the current GPU.

2.1.2 CUDA frontend

In the current CUDA compilation scheme, the CUDA front end is invoked twice. The first step is for the actual splitup of the .cu input into host and device code. The second step is a technical detail (it performs dead code analysis on the .gpu generated by the first step), and it might disappear in future releases.

2.1.3 Preprocessing

The trajectory contains a number of preprocessing steps. The first of these, on the .cu input, has the usual purpose of expanding include files and macro invocations that are present in the source file. The remaining preprocessing steps expand CUDA system macros in ('C'-) code that has been generated by preceding CUDA compilation steps. The last preprocessing step also merges the results of the previously diverged compilation flow.

2.1.4 Using cudafe for preprocessing

Figure 2.2 shows that a full CUDA compilation step requires 4 preprocessing steps, which are ultimately performed using the platform compiler. An unfortunate side effect of this on Windows platforms would be a quite noisy CUDA compilation, due to the fact that cl insists on echoing the name of its input file each time it is invoked. For this reason, nvcc will use cudafe for preprocessing whenever it finds this internal CUDA tool on the the executable search PATH (which normally is the case in CUDA releases).

2.2 Advantages

Advantages of CUDA over the traditional approach to GPGPU computing:

- More efficient data transfers between system and video memory.
- Faster downloads and read backs to and from the GPU.
- Scattered reads - code can read from arbitrary addresses in memory.
- Shared memory - CUDA exposes a fast shared memory region (16KB in size) that can be shared amongst threads. This can be used as a user-managed cache, enabling higher bandwidth than is possible using texture lookups.
- Full support for integer and bitwise operations.
- Support for integer texture lookups.
- Programming interface of CUDA applications is based on the standard C language with extensions, which facilitates the learning curve of CUDA.

2.3 Limitation

- Threads should be running in groups of at least 32 for best performance, with total number of threads numbering in the thousands. Branches in the program code do not impact performance significantly, provided that each of 32 threads takes the same execution path; the SIMD execution model becomes a significant limitation for any inherently divergent task.
- Texture rendering is not supported.
- It uses a recursion-free, function-pointer-free subset of the C language, plus some simple extensions. However, a single process must run spread across multiple disjoint memory spaces, unlike other C language runtime environments.
- For double precision there are no deviations from the IEEE 754 standard. In single precision, Denormals and signalling NaNs are not supported; only two IEEE rounding modes are supported and those are specified on a per instruction

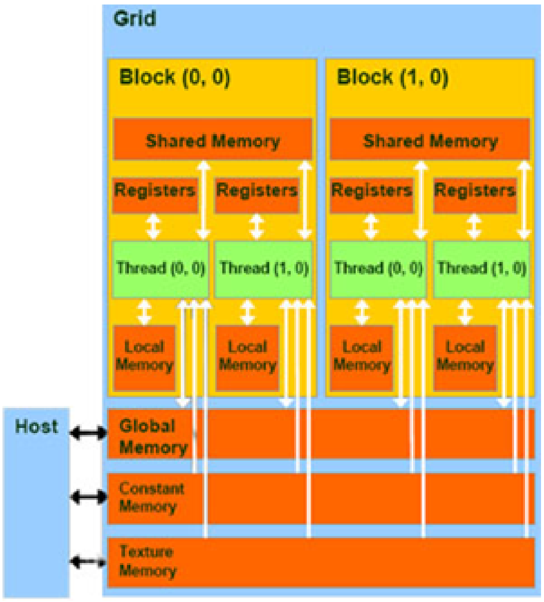


Figure 2.3: CUDA Memory hierarchy [2]

basis rather than in a control word and the precision of division square root are slightly lower than single precision.

2.4 CUDA Memory Model

CUDA threads may access data from multiple memory spaces during their execution as illustrated by Figure 2.3. Each thread has private local memory. Each thread block has shared memory visible to all threads of the block and with the same lifetime as the block. All threads have access to the same global memory. There are also two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces. The global, constant, and texture memory spaces are optimized for different memory usages Texture memory also offers different addressing modes, as well as data filtering, for some specific data formats. The global, constant, and texture memory spaces are persistent across kernel launches by the same application.

a. Local Memory: : is small volume of memory, which can be accessed only by one streaming processor. The local memory space resides in device memory, so local memory accesses have same high latency and low bandwidth as global memory accesses.

b. Global Memory : the largest volume of memory available to all multiprocessors in a GPU, from 256 MB to 1.5 GB in modern solutions (and up to 4 GB in Tesla). It offers high bandwidth, over 100 GB/s for top solutions from NVIDIA, but it suffers from very high latencies (several hundred cycles). Non-catchable supports general load and store instructions, and usual pointers to memory.

c. Shared Memory: is 16-KB memory shared between all streaming processors in a multiprocessor. Because it is on-chip, the shared memory space is much faster than the local and global memory spaces.

d. Constant Memory: is a 64 KB, read only memory for all multiprocessors. It's cached by 8 KB for each multiprocessor. The constant memory space resides in device memory. A constant memory request for a warp is first split into two requests, one for each half-warp, that are issued independently. A request is then split into as many separate requests as there are different memory addresses in the initial request, decreasing throughput by a factor equal to the number of separate requests. The resulting requests are then serviced at the throughput of the constant cache in case of a cache hit, or at the throughput of device memory otherwise. This memory is rather slow latencies of several hundred cycles, if there are no required data in cache.

e. Texture Memory: space resides in device memory and is cached in texture cache, so a texture fetch costs one memory read from device memory only on a cache miss, otherwise it just costs one read from texture cache.

Chapter 3

Performance Optimization Strategies

Performance optimization revolves around four basic strategies:

- Convert CUDA C code in parallel to reduce time exection.
- Maximize parallel execution to achieve maximum utilization
- Try to convert recursion code to Serial code and further parallelize it.
- Remove dependencies in CUDA C code.
- Optimize instruction usage to achieve maximum instruction throughput.

Which strategies will yield the best performance gain for a particular portion of an application depends on the performance limiters for that portion, optimizing instruction usage of a kernel that is mostly limited by memory accesses will not yield any significant performance gain. Optimization efforts should therefore be constantly directed by measuring and monitoring the performance limiters, for example using the CUDA profiler.

3.1 Maximize Utilization

To get the maximum utilization of the available resources, application should be parallelized in such a way that application keeps various components of the system busy most of the time.

3.1.1 Application Level

At a high level, the application should maximize parallel execution between the host, the devices, and the bus connecting the host to the devices, by using asynchronous functions calls and streams. It should assign to each processor the type of work it does best: serial workloads to the host; parallel workloads to the devices.

For parallel execution program is divided into threads, this threads need to share data with each other, there are two cases:

- If this threads belong to same block, they should use syncthreads() and share data through shared memory.

- If threads belong to different blocks, they must share data through global memory. In this case two separate kernel invocations are required, one for writing to and one for reading from global memory.

3.1.2 Device Level

At a lower level, the application should maximize parallel execution between the multiprocessors of a device.

For devices of compute capability 1.x, only one kernel can execute on a device at one time, so the kernel should be launched with at least as many thread blocks as there are multiprocessors in the device. For devices of compute capability 2.0, multiple kernels can execute concurrently on a device, so maximum utilization can also be achieved by using streams to enable enough kernels to execute concurrently.

3.1.3 Multiprocessor Level

At an even lower level, the application should maximize parallel execution between the various functional units within a multiprocessor.

To maximize utilization, a GPU multiprocessor relies on thread-level parallelism. Utilization is therefore directly dependent on the number of resident warps. At every instruction issue time, a warp scheduler selects a warp that is ready to execute, if any, and issues the next instruction to the active threads of the warp. The number of clock cycles it takes for a warp to be ready to execute its next instruction is called latency, and full utilization is achieved when the warp scheduler always has some instruction to issue for some warp at every clock cycle during that latency period, or in other words, when the latency of each warp is completely hidden by other warps. How many instructions are required to hide latency depends on the instruction throughput.

If all input operands are registers, latency is caused by register dependencies. In the case of a back-to-back register dependency (i.e. some input operand is written by the previous instruction), the latency is equal to the execution time of the previous instruction and the warp scheduler must schedule instructions for different warps during that time.

3.2 Maximize Instruction Throughput

If programmer knows, how instructions are executed then it is possible to apply low level optimizations that can be useful. It is good practices to apply lower level optimization after all higher-level optimization have been completed. To maximize instruction throughput the application should:

- Minimize the use of arithmetic instructions with low throughput; this includes trading precision for speed when it does not affect the end result, such as using intrinsic instead of regular functions, single-precision instead of double precision, or flushing de normalized numbers to zero.

- Minimize divergent warps caused by control flow instructions.

- Reduce the number of instructions, for example, by optimizing out synchronization points whenever possible or by using restricted pointers

Chapter 4

Preliminary Study of programming language

4.1 Dynamic Programming

The key idea behind dynamic programming is quite simple. In general, to solve a given problem, we need to solve different parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution. Often, many of these subproblems are really the same. The dynamic programming approach seeks to solve each subproblem only once, thus reducing the number of computations. This is especially useful when the number of repeating subproblems is exponentially large.

Top-down dynamic programming simply means storing the results of certain calculations, which are later used again since the completed calculation is a sub-problem of a larger calculation. Bottom-up dynamic programming involves formulating a complex calculation as a recursive series of simpler calculations.

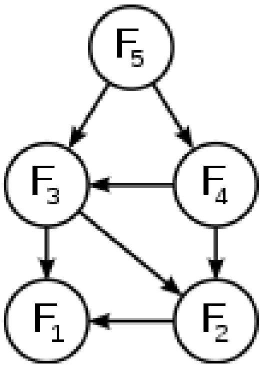


Figure 4.1: The subproblem graph for the Fibonacci sequence. The fact that it is not a tree indicates overlapping subproblems [6]

4.1.1 Dynamic programming in computer programming

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping subproblems. However, when the overlapping problems are much smaller than the original problem, the strategy is called "divide and conquer" rather than "dynamic programming". This is why mergesort, quicksort, and finding all matches of a regular expression are not classified as dynamic programming problems.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its subproblems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion. For example, given a graph G=(V,E), the shortest path p from a vertex u to a vertex v exhibits optimal substructure: take any intermediate vertex w on this shortest path p. If p is truly the shortest path, then the path p1 from u to w and p2 from w to v are indeed the shortest paths between the corresponding vertices (by the simple cut-and-paste argument described in CLRS). Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman-Ford algorithm does.

- **Top-down approach:** This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its subproblems, and if its subproblems are overlapping, then one can easily memoize or store the solutions to the subproblems in a table. Whenever we attempt to solve a new subproblem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly, otherwise we solve the subproblem and add its solution to the table.

- **Bottom-up approach:** This is the more interesting case. Once we formulate the solution to a problem recursively as in terms of its subproblems, we can try reformulating the problem in a bottom-up fashion: try solving the subproblems first and use their solutions to build-on and arrive at solutions to bigger subproblems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger subproblems by using the solutions to small subproblems.

4.2 CUDA C Programming

4.2.1 General-Purpose Parallel Computing Architecture

The advent of multicore CPUs and many core GPUs means that mainstream processor chips are now parallel systems. Furthermore, their parallelism continues to scale with Moores law. The challenge is to develop application software that transparently scales its parallelism to leverage the increasing number of processor cores, much as 3D graphics applications transparently scale their parallelism to many core GPUs with widely varying numbers of cores. CUDAs parallel programming model is designed to overcome this challenge while maintaining a low learning curve for programmers familiar with standard programming languages such as C.

At its core are three key abstractions a hierarchy of thread groups, shared mem-

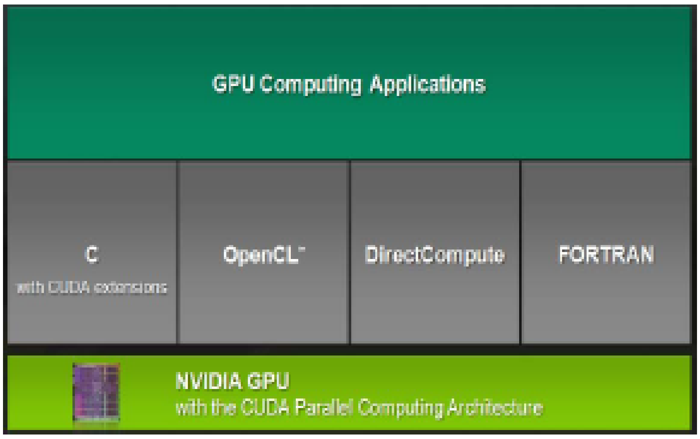


Figure 4.2: CUDA is Designed to Support Various Languages or Application Programming Interfaces [8]

ories, and barrier synchronization that are simply exposed to the programmer as a minimal set of language extensions.

4.2.2 Kernels

C for CUDA extends C by allowing the programmer to define C functions, called kernels, that, when called, are executed N times in parallel by N different CUDA threads, as opposed to only once like regular C functions.

A kernel is defined using the global declaration specifier and the number of CUDA threads for each call is specified using a new

```
"<<<>>>"
```

syntax:

Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel through the built-in threadIdx variable. As an illustration, the following sample code adds two vectors A and B of size N and stores the result into vector C:

Each of the threads that execute VecAdd() performs one pair-wise addition.

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    ...
}

int main()
{
    ...
    // Kernel invocation
    VecAdd<<<1, N>>>(A, B, C);
}
```

Figure 4.3: Kernel call [2]

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
```

Figure 4.4: Kernel call [2]

```
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
```

Figure 4.5: Kernel call [2]

4.2.3 Thread Hierarchy

For convenience, threadIdx is a 3-component vector, so that threads can be identified using a one-dimensional, two-dimensional, or three-dimensional thread index, forming a one-dimensional, two-dimensional, or three-dimensional thread block. This provides a natural way to invoke computation across the elements in a domain such as a vector, matrix, or field. As an example, the following code adds two matrices A and B of size NxN and stores the result into matrix C:

Threads within a block can cooperate among themselves by sharing data through some shared memory and synchronizing their execution to coordinate memory accesses. More precisely, one can specify synchronization points in the kernel by calling the __syncthreads() intrinsic function; __syncthreads() acts as a barrier at which all threads in the block must wait before any is allowed to proceed.

For efficient cooperation, the shared memory is expected to be a low-latency memory near each processor core, much like an L1 cache, __syncthreads() is expected to be lightweight, and all threads of a block are expected to reside on the same processor core. The number of threads per block is therefore restricted by the limited memory resources of a processor core. On current GPUs, a thread block may contain up to 512 threads.

Thread blocks are required to execute independently: It must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores.

4.2.4 Programming Interface

The CUDA driver API is a lower-level C API that provides functions to load kernels as modules of CUDA binary or assembly code, to inspect their parameters, and to launch them. Binary or assembly code are usually obtained by compiling kernels written in C.

C for CUDA comes with a runtime API and both the runtime API and the driver API provide functions to allocate and deallocate device memory, transfer data between host memory and device memory, manage systems with multiple devices, etc.

Chapter 5

Problem Definition

Using GPU architectures for solving large scale or difficult optimization problems like combinatorial optimization problems is nevertheless a great challenge due to the specificities of GPU architectures. The main issues that are to be met is performance issues. In this thesis we will mainly concentrate to some complex algorithms and have comparative study by implementing it on different GPU's and obtain the speedup gain in each case.

5.1 Performance Issues

5.1.1 Communication Bottlenecks

Whether you are on a shared-memory, message-passing or other platform, communication is always a potential bottleneck:

- On a shared-memory system, the threads must contend with each other in communicating with memory.And the problem is exacerbated by cache coherency transactions.
- On a NOW, even a very fast network is very slow compared to CPU speeds.
- GPUs are really fast, but their communication with their CPU hosts is slow.

5.1.2 Load Balancing

Another major issue is load balancing, i.e. keeping all the processors busy as much as possible. A nice, easily understandable example is shown in Multicore Application Programming: for Windows, Linux and Oracle Solaris, Darryl Gove, 2011, Addison-Wesley. There the author shows code to compute the Mandelbrot set. He has a rectangular grid of points in the plane, and wants to determine whether each point is in the set or not; a simple but time-consuming computation is used for this determination. Gove sets up two threads, one handling all the points in the left half of the grid and the other handling the right half. He finds that the latter thread is very often idle, while the former thread is usually busy-severe load imbalance.

5.1.3 Embarrassingly Parallel Application

Consider a matrix multiplication application, for instance, in which we compute AX for a matrix A and a vector X. One way to parallelize this problem would be for have each processor handle a group of rows of A, multiplying each by X in parallel with the other processors, which are handling other groups of rows. We call the problem embarrassingly parallel, with the word ”embarrassing” meaning that the problem is too easy, with is no intellectual challenge involved. It is pretty obvious that the computation Y = AX can be parallelized very easily by splitting the rows of A into groups. By contrast, most parallel sorting algorithms require a great deal of interaction. For instance, consider Merge sort. It breaks the vector to be sorted into two (or more) independent parts, say the left half and right half, which are then sorted in parallel by two processes. So far, this is embarrassingly parallel, at least after the vector is broken in half. But then the two sorted halves must be merged to produce the sorted version of the original vector, and that process is not embarrassingly parallel; it can be parallelized, but in a more complex manner.

Chapter 6

Implementation

6.1 Binary Search

```
binary_search(Array[0..N-1], value, low, high):
      if (high < low):
          return -1 // not found
      mid = (low + high) / 2
      if (A[mid] > value):
          return binary_search(A, value, low, mid-1)
      else if (A[mid] < value):
          return binary_search(A, value, mid+1, high)
      else:
          return mid // found
```

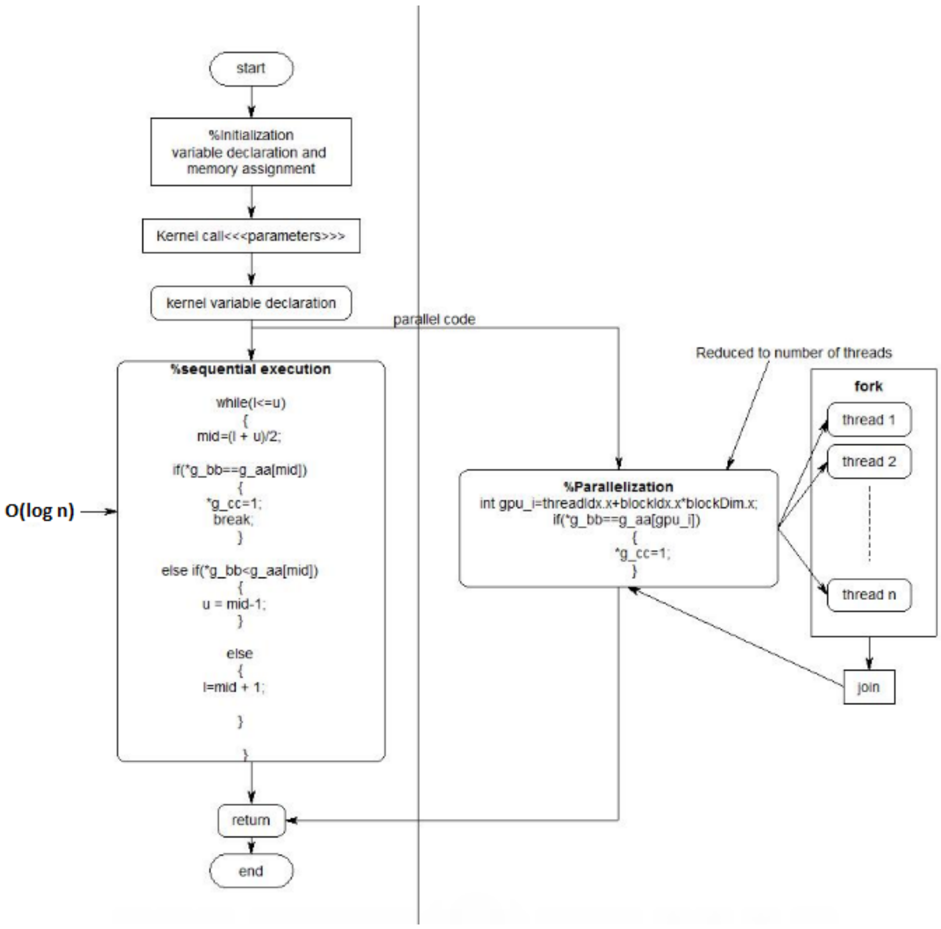


Figure 6.1: Task graph

6.1.1 Comparative Study for Binary Search

The comparative study include execution of Binary Search algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

Implementation on CPU

The fig. 6.3 shows the output in which it considers total time for executing the algorithm on CPU. It takes 592 ms to execute the program by taking the entire program under consideration and fig. 6.4 shows the output of the part of program

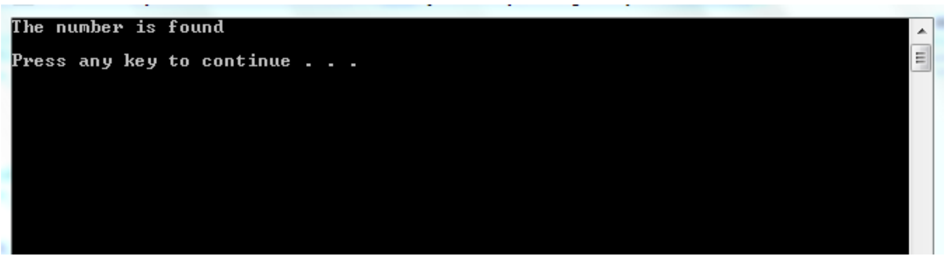


Figure 6.2: Binary Search Output

executed on CPU but it consists of the only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken for the part of the program to execute and it takes 11ms to execute the part of the program.

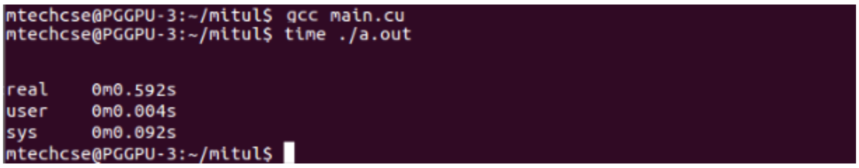


Figure 6.3: Binary CPU Time considering entire program

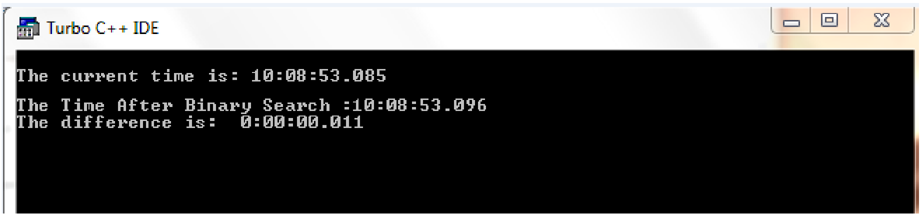


Figure 6.4: Binary CPU Time considering part of the program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 103 ms as shown in the fig. 6.5 and hence the speed up gain as compared with CPU is 400 ms and the time taken to execute the program without considering memory transfer is 0.0534 ms as shown in the fig. 6.6. In order to obtain the result without memory transfer,i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

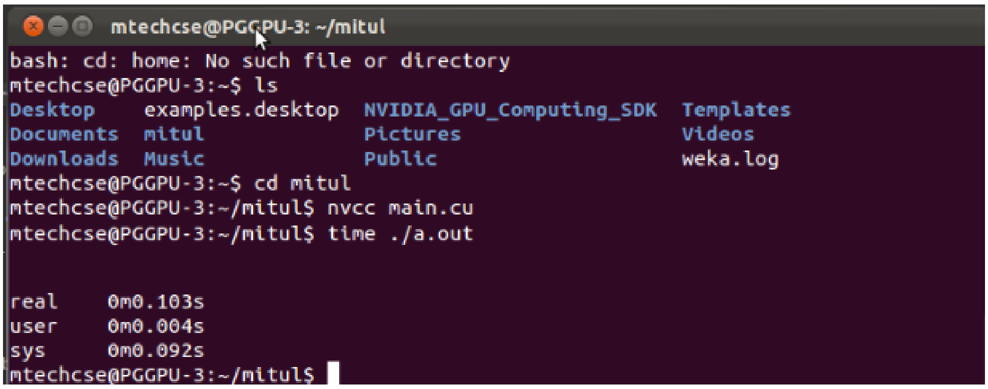
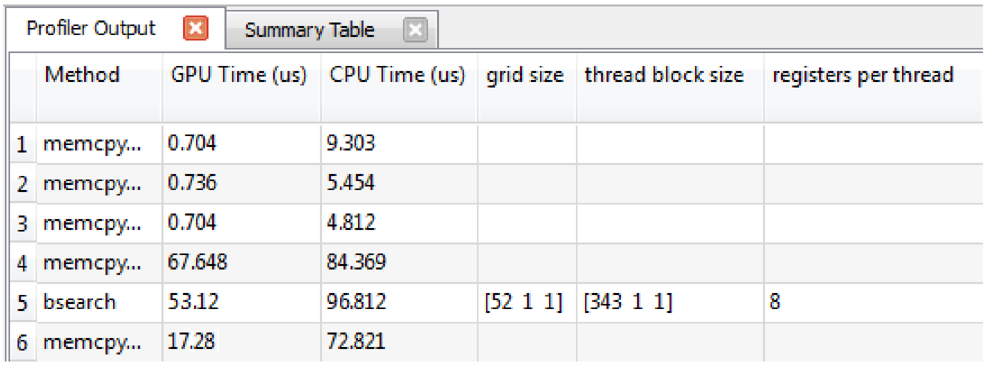


Figure 6.5: GTX 480 Considering Memory Transfer



Profiler Output | Summary Table

	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread
1	memcpy...	0.704	9.303			
2	memcpy...	0.736	5.454			
3	memcpy...	0.704	4.812			
4	memcpy...	67.648	84.369			
5	bsearch	53.12	96.812	[52 1 1]	[343 1 1]	8
6	memcpy...	17.28	72.821			

Figure 6.6: GTX 480 without considering Memory Transfer

Implementation on Tesla C2070

The total time considering memory transfer in Tesla GPU is 73 ms as shown in the fig. 6.7 and hence the speed up gain as compared with CPU is 519 ms and the time taken to execute the program without considering memory transfer is 0.0453 ms as shown in the fig. 6.8. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla GPU compared to that of CPU.

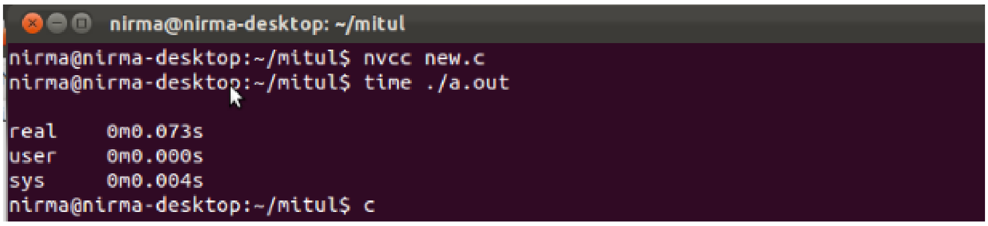
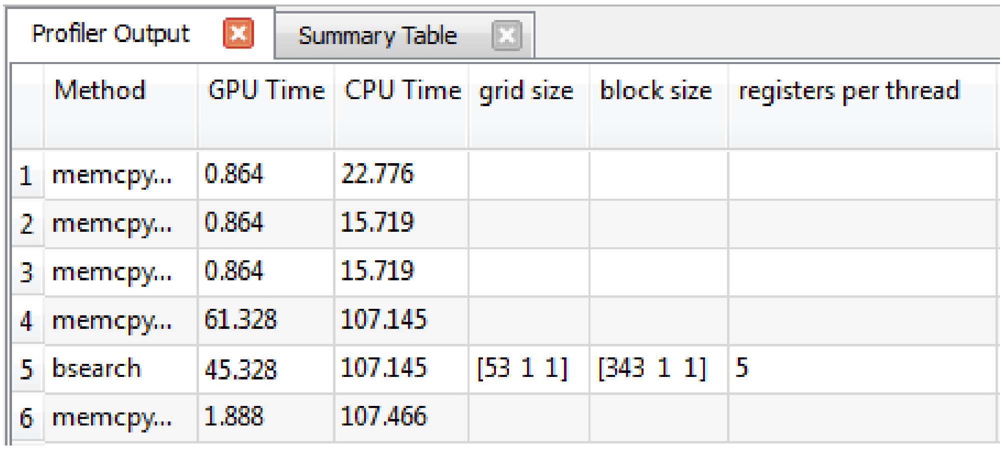


Figure 6.7: Tesla considering Memory transfer



	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1	memcpy...	0.864	22.776			
2	memcpy...	0.864	15.719			
3	memcpy...	0.864	15.719			
4	memcpy...	61.328	107.145			
5	bsearch	45.328	107.145	[53 1 1]	[343 1 1]	5
6	memcpy...	1.888	107.466			

Figure 6.8: Tesla Considering Without Memory Transfer

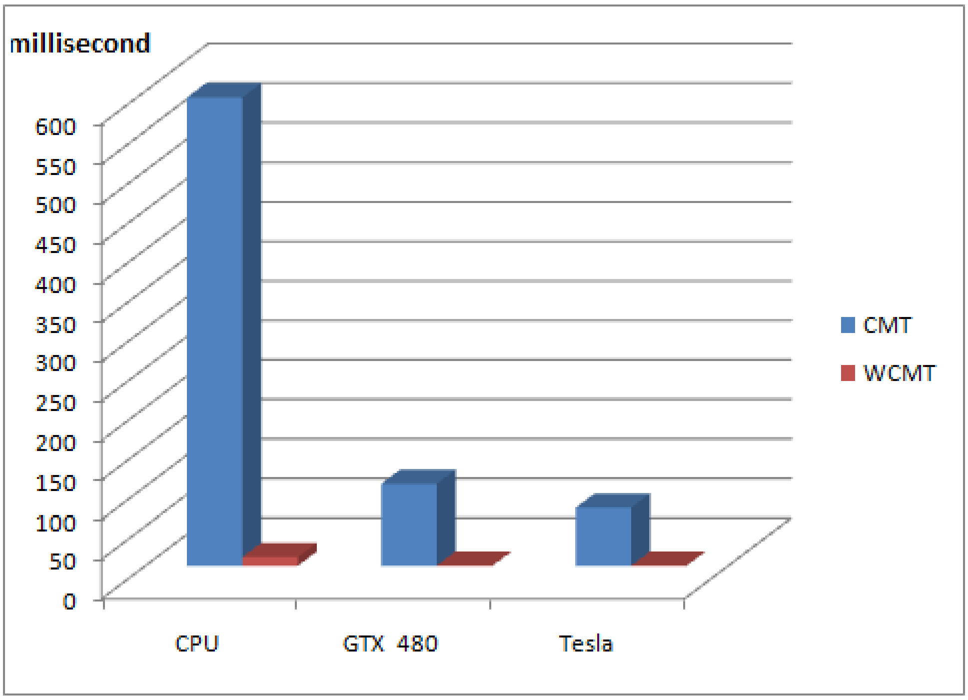


Figure 6.9: Graph Comprising of speedup

Quantitative Comparison

- Here the CPU takes a lot of time then compared to algorithm executed on GPU's which is been reduced from O(log n) to number of threads in parallel.
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 0.053 ms
  - Grid size: [53 1 1]
  - Block size: [342 1 1]
- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 12.21 (Balanced Instruction per byte ratio: 3.79)

- Achieved Occupancy: 0.80 (Theoretical Occupancy: 0.92)

- Also here the limiting factor for GTX 480 GPU is 0.053 milliseconds in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.

- Summary profiling information of Tesla GPU
  - Number of calls: 1
  - GPU time: 0.045 ms
  - Grid size: [53 1 1]
  - Block size: [342 1 1]

- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 10.14 (Balanced Instruction per byte ratio: 3.79)
  - Achieved Occupancy: 0.78 (Theoretical Occupancy: 0.92)

- Also here the limiting factor for Tesla GPU is 0.045 milliseconds in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.

- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.
  - The derived statistics assume all instruction are single precision floating point instruction. If double precision floating point instruction are used then the limiting factor may become incorrect.

- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the tesla is 2988 MHz, hence more speedup is obtained on the Tesla GPU.
- Considering the both the benchmark conditions the best speed up is obtained on Tesla GPU.
- Considering core clock which is highest in Tesla leading to the best performance.
- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on Tesla C2070 GPU.
- Hence from all the above statics and parameters we can theoretically conclude that best performance can be obtained on Tesla C2070 GPU, which is proved practically.

6.2 Knapsack Algorithm

```
Function knapsack(w[1..n],v[1..n],W)
%initialization
for i<-1 upto n do
    x[i] <- 0
    weight <- 0
    sort the objects into descending order of vi/wi
    while(weight<W)
    i <- select remaining object  with maximum vi/wi
        if(weight+w[i]<=W) then
            x[i] <- 1
            weight<- weight + w[i]
        else
            x[i] <- (W - weight)/w[i]
            weight <- W
return x
```

6.2.1 Comparative Study for Knapsack Algorithm

The comparative study include execution of Knapsack algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

Implementation on CPU

The fig. 6.10 & fig 6.11 shows the task graph and the output of the program and fig 6.12 shows the output in which it considers total time for executing the algorithm on CPU. It takes 153 ms to execute the program by taking the entire program under consideration and fig. 6.13 shows the output of the part of program executed on CPU but it consists of only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken by the part of the program to execute and it takes 17 ms to execute the part of the program.

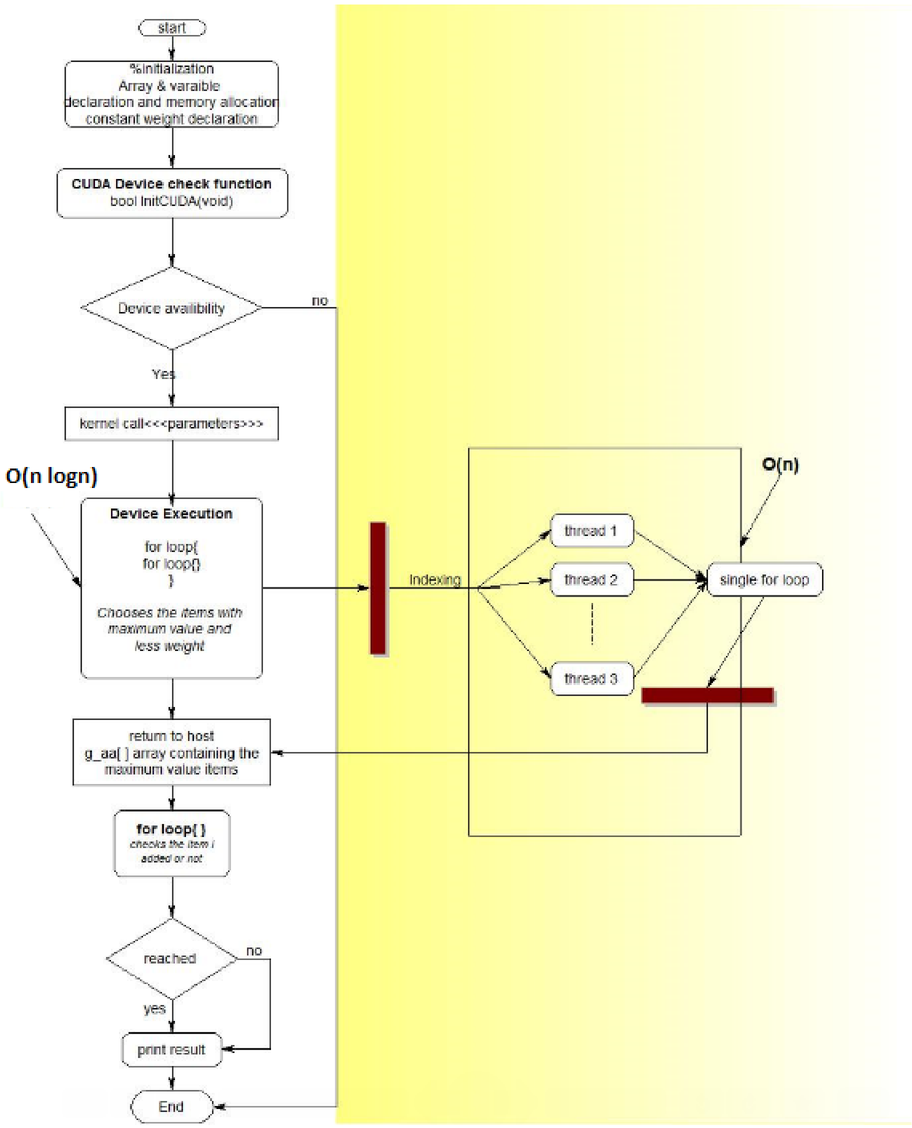


Figure 6.10: Task Graph

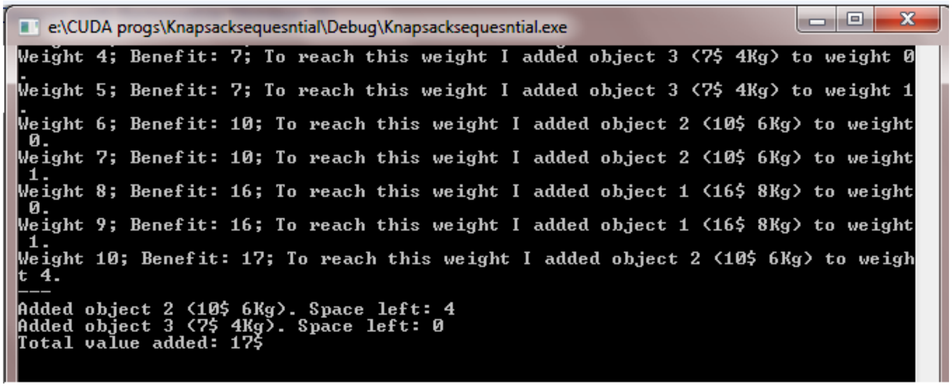


Figure 6.11: Knapsack Output

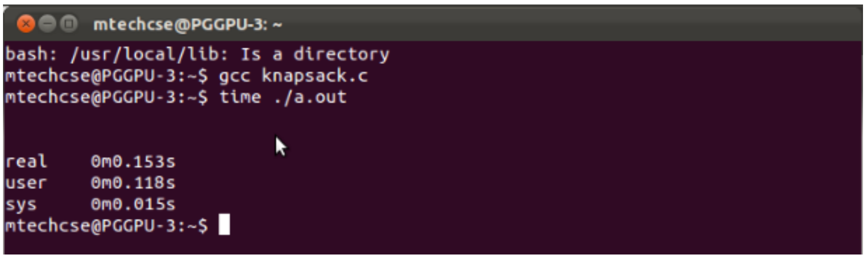


Figure 6.12: knapsack CPU Time considering entire program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 107 ms as shown in the fig. 6.14 and hence the speed up gain as compared with CPU is 46 ms and the time taken to execute the program without considering memory transfer is 12.53 ms as shown in the fig. 6.15. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

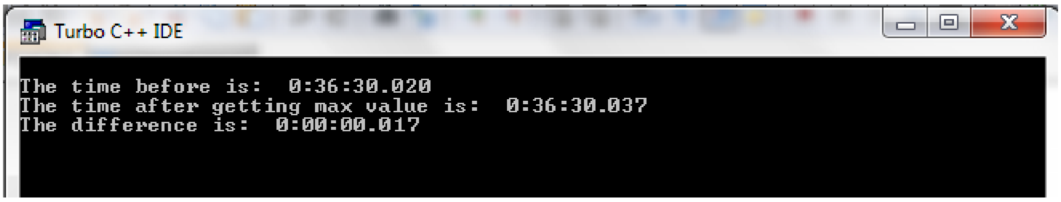


Figure 6.13: Knapsack CPU Time considering part of the program

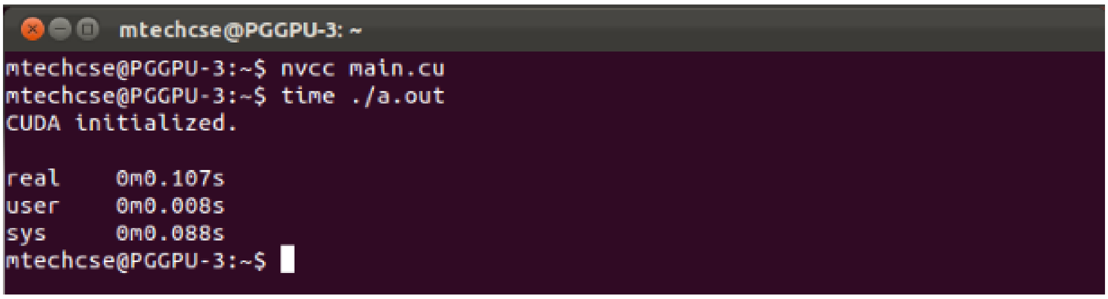
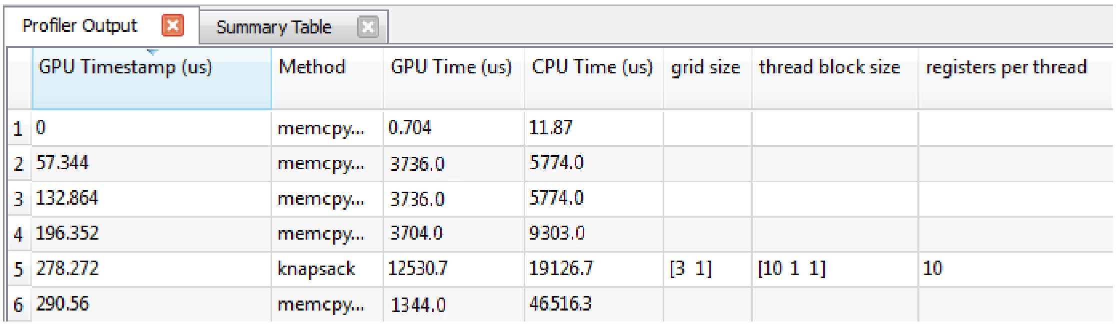


Figure 6.14: GTX 480 Considering Memory Transfer

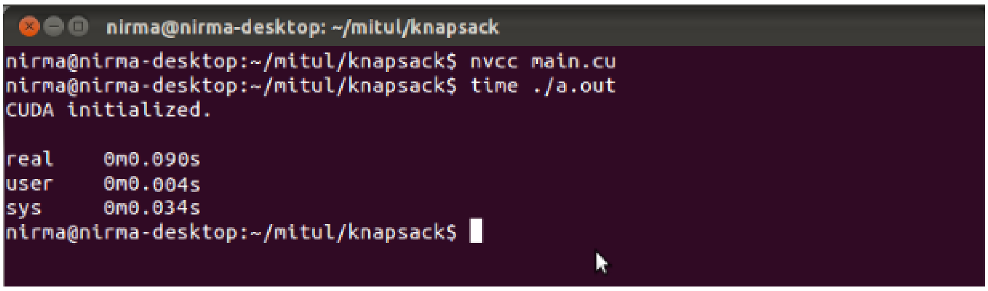
Implementation on Tesla C2070

The total time considering memory transfer in Tesla GPU is 90 ms as shown in the fig. 6.16 and hence the speed up gain as compared with CPU is 63 ms and the time taken to execute the program without considering memory transfer is 11.97 ms as shown in the fig. 6.17. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla C2070 GPU compared to that of CPU.



	GPU Timestamp (us)	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread
1	0	memcpy...	0.704	11.87			
2	57.344	memcpy...	3736.0	5774.0			
3	132.864	memcpy...	3736.0	5774.0			
4	196.352	memcpy...	3704.0	9303.0			
5	278.272	knapsack	12530.7	19126.7	[3 1]	[10 1 1]	10
6	290.56	memcpy...	1344.0	46516.3			

Figure 6.15: GTX 480 Considering Without Memory Transfer



```
nirma@nirma-desktop:~/mitul/knapsack$ nvcc main.cu
nirma@nirma-desktop:~/mitul/knapsack$ time ./a.out
CUDA initialized.

real	0m0.090s
user	0m0.004s
sys	0m0.034s
nirma@nirma-desktop:~/mitul/knapsack$
```


Figure 6.16: Tesla Considering Memory Transfer)

Quantitative Comparison

- Here the CPU takes O(n log n) time time then compared to algorithm executed on GPU's which is been reduced to O(n) times.
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 12.53 ms
  - Grid size: [3 1]
  - Block size: [10 1 1]
- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 166.50
  - Achieved Occupancy: 0.02 (Theoretical Occupancy: 0.04)

Profiler Output

	GPU Timestamp	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1	0	memcpy...	0.8	20.531			
2	111.104	memcpy...	16325.7	16045.4			
3	241.664	memcpy...	0.8	16.04			
4	358.912	memcpy...	0.832	15719			
5	1446.14	knapsack	11970.4	99446	[3 1]	[10 1 1]	7
6	1697.79	memcpy...	18885.4	64.159			
7	1868.03	memcpy...	18885.4	55.497			

Figure 6.17: Tesla Considering Without Memory Transfer

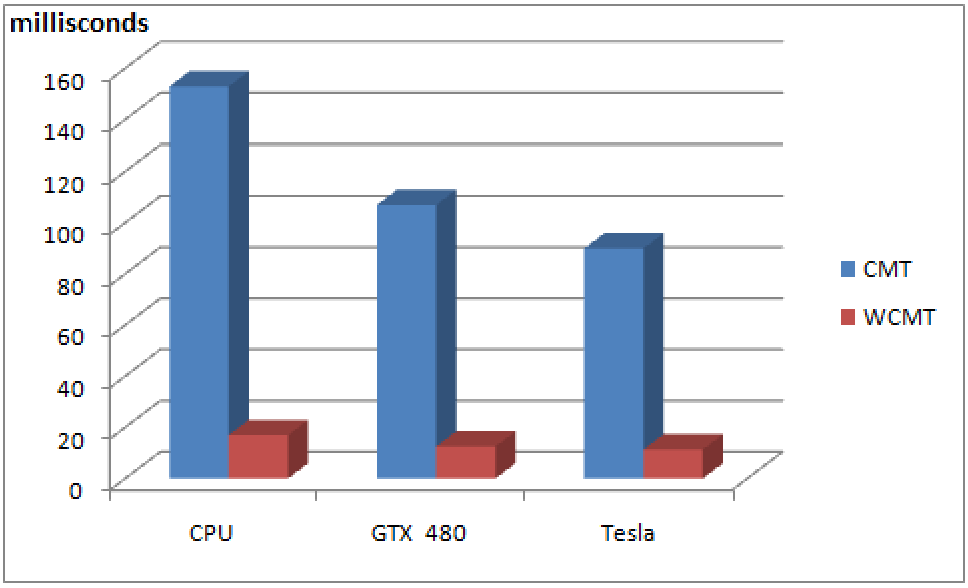


Figure 6.18: Graph Comprising of speedup

- Also here the limiting factor for GTX 480 GPU is 12.53 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call, performance degrades and also the utilization of the GPU decreases.

- Summary profiling information of Tesla C2070 GPU
  - Number of calls: 1
  - GPU time: 11.37 ms

  - Grid size: [3 1]
  - Block size: [10 1 1]

- Limiting factor for Tesla C2070 GPU
  - Achieved instruction per byte ratio: 173.51
  - Achieved Occupancy: 0.167 (Theoretical Occupancy: 0.2)

- Also here the limiting factor for Tesla GPU is 11.37 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.

- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.
  - Also as such as knapsack being converted to iterative form and further to parallel form, due to some dependencies it may affect speedup performance.

- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the tesla is 2988 MHz, hence more speedup is obtained on the Tesla GPU.

- Considering core clock which is highest in Tesla leading to the best performance.

- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on Tesla GPU.

- Hence from all the above statics and parameters we can theoretically conclude that best performance can be obtained on Tesla GPU which is proved practically.

6.3 Longest Common Subsequence

```
LCS-LENGTH(X, Y,m, n)
for i <- 1 to m
    do c[i, 0] <- 0
for j <- 0 to n
    do c[0, j ] <- 0
for i <- 1 to m
    do for j <- 1 to n
        do if xi = yj
            then c[i, j ] <- c[i . 1, j . 1] + 1
                 b[i, j ] <- "\"
            else if c[i - 1, j ] >= c[i, j -1 ]
                 then c[i, j ] <- c[i - 1, j ]
                       b[i, j ] <- "|"
                  else c[i, j ] <- c[i, j - 1]
                       b[i, j ] <- "<-"
return c and b
//
//
PRINT-LCS(b, X, i, j )
if i = 0 or j = 0
    then return
if b[i, j ] = "\"
    then PRINT-LCS(b, X, i - 1, j - 1)
         print xi
elseif b[i, j ] = "|"
    then PRINT-LCS(b, X, i - 1, j )
    else
```

```
PRINT-LCS(b, X, i, j - 1)
```

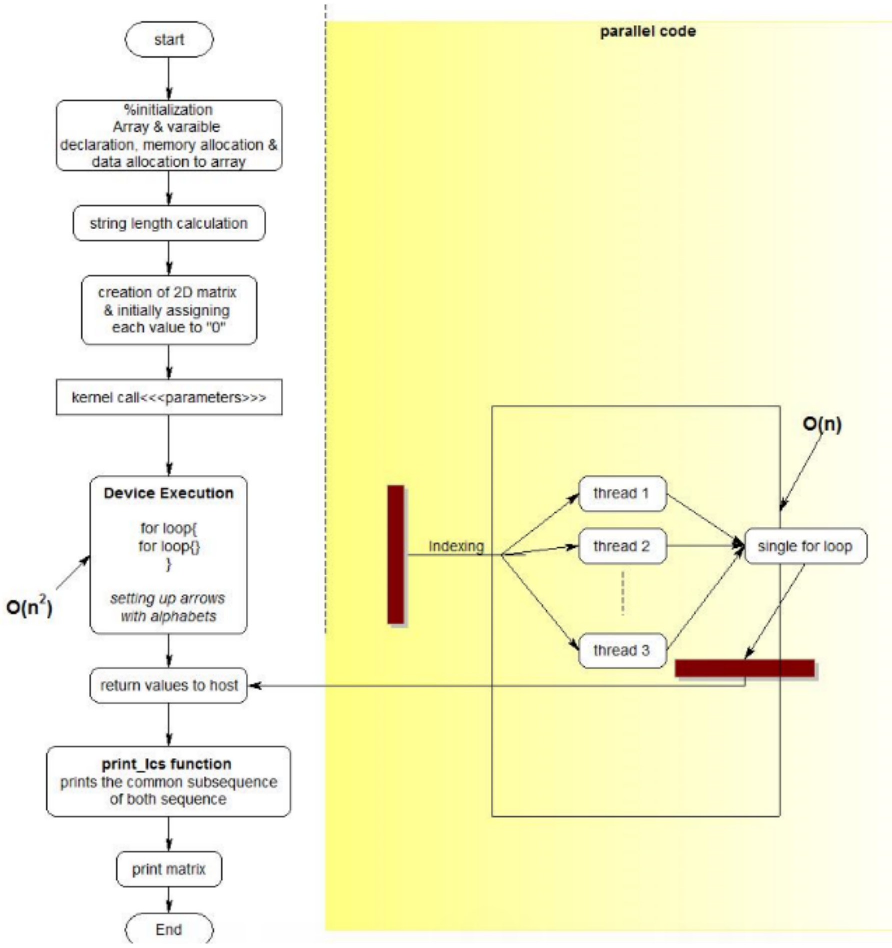


Figure 6.19: Task Graph

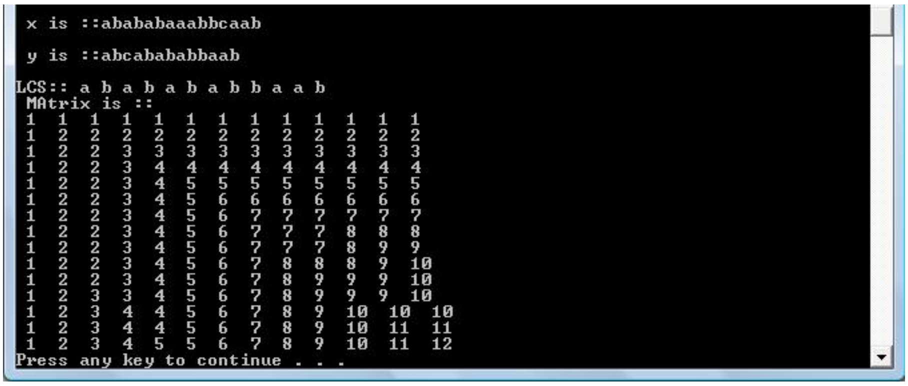


Figure 6.20: LCS Output

6.3.1 Comparative Study for Longest Common Subsequence Algorithm

The comparative study include execution of LCS algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

Implementation on CPU

The fig. 6.19 & fig. 6.20 shows the task graph and the output of the program and fig. 6.21 shows the output in which it considers total time for executing the algorithm on CPU. It takes 321 ms to execute the program by taking the entire program under consideration and fig. 6.22 shows the output of the part of program executed on CPU, but it consists of only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken for the part of the program to execute and it takes 214 ms to execute the part of the program.

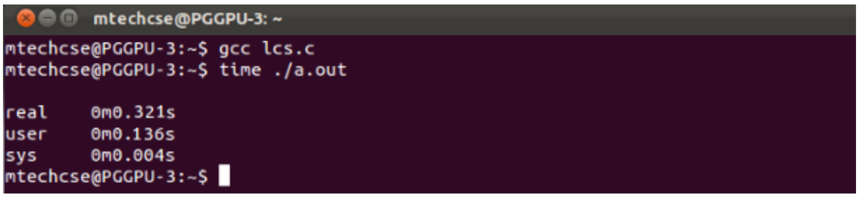


Figure 6.21: LCS CPU Time considering entire program

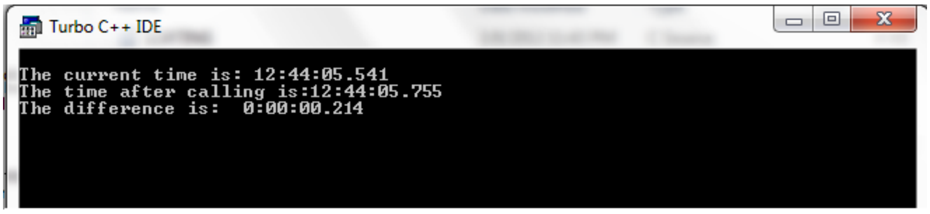


Figure 6.22: LCS CPU Time considering part of the program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 154 ms as shown in the fig. 6.23 and hence the speed up gain as compared with CPU is 167 ms and the time taken to execute the program without considering memory transfer is 96.21 ms as shown in the fig. 6.24. In order to obtain the result without memory transfer,i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

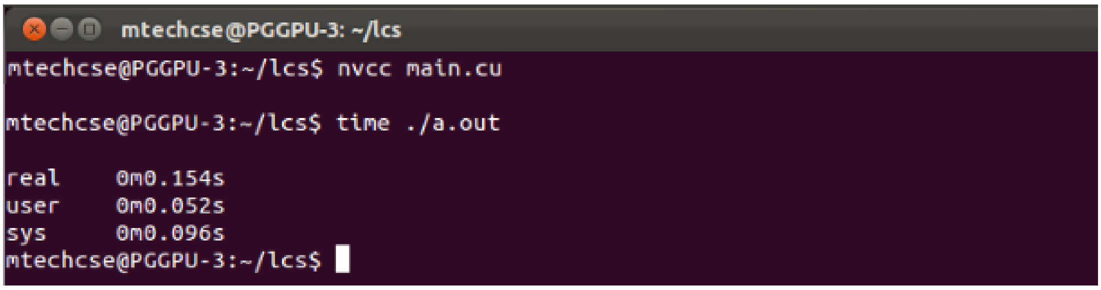


Figure 6.23: GTX 480 Considering Memory Transfer

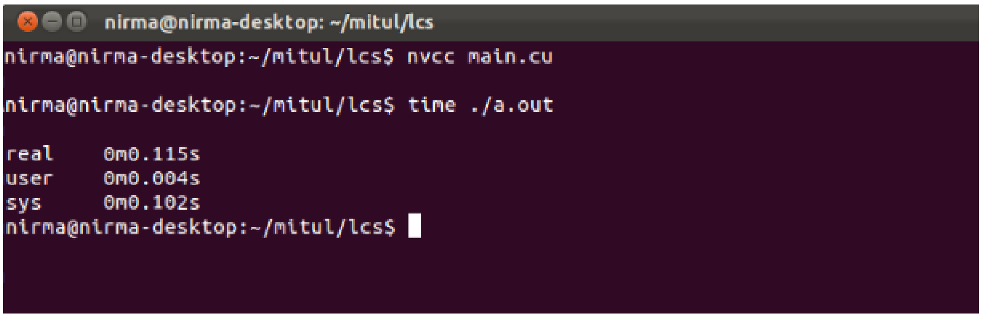
Profiler Output | Summary Table

	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread
1	memcpy...	0.736	8341.7			
2	memcpy...	0.704	5133.4			
3	memcpy...	2624.1	12832.1			
4	memcpy...	23362.8	12832.7			
5	kernel	96210.8	507754.1	[3 1]	[21 1 1]	12
6	memcpy...	1696.1	112921			
7	memcpy...	31360.7	61272.1			

Figure 6.24: GTX 480 Considering without Memory Transfer

Implementation on Tesla C2070

The total time considering memory transfer in Tesla C2070 GPU is 115 ms as shown in the fig. 6.25 and hence the speed up gain as compared with CPU is 206 ms and the time taken to execute the program without considering memory transfer is 89.43 ms as shown in the fig. 6.26. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla C2070 GPU compared to that of CPU.

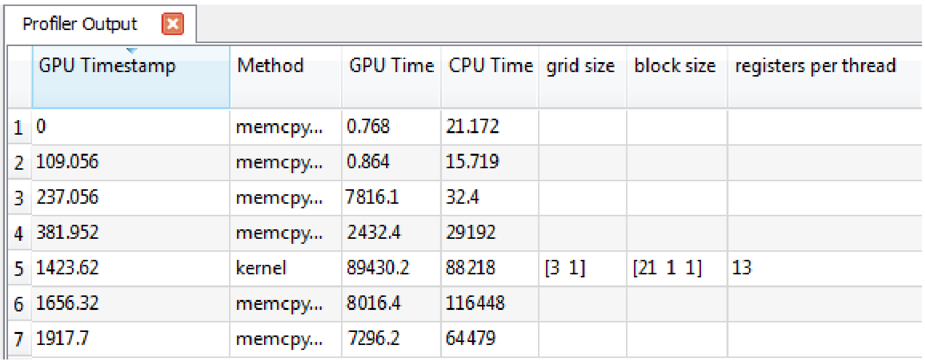


```
nirma@nirma-desktop: ~/mitul/lcs
nirma@nirma-desktop:~/mitul/lcs$ nvcc main.cu

nirma@nirma-desktop:~/mitul/lcs$ time ./a.out

real    0m0.115s
user    0m0.004s
sys     0m0.102s
nirma@nirma-desktop:~/mitul/lcs$
```

Figure 6.25: Tesla Considering Memory Transfer



Profiler Output

	GPU Timestamp	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1	0	memcpy...	0.768	21.172			
2	109.056	memcpy...	0.864	15.719			
3	237.056	memcpy...	7816.1	32.4			
4	381.952	memcpy...	2432.4	29192			
5	1423.62	kernel	89430.2	88218	[3 1]	[21 1 1]	13
6	1656.32	memcpy...	8016.4	116448			
7	1917.7	memcpy...	7296.2	64479			

Figure 6.26: Tesla Considering Without Memory Transfer

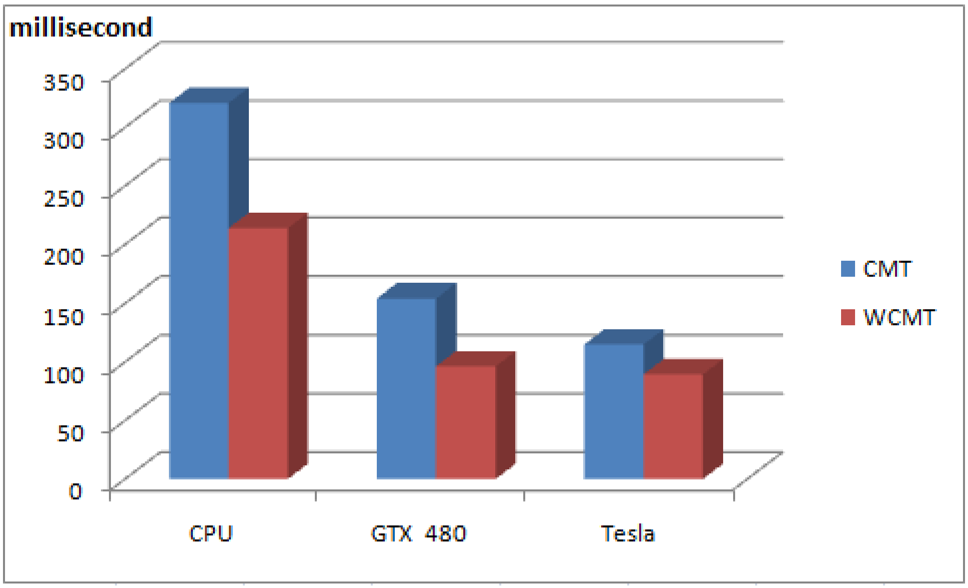


Figure 6.27: Graph Comprising of speedup

Quantitative Comparison

- Here the CPU takes a lot of time then compared to algorithm executed on GPU's which is been reduced from O(n^2) to number O(n).
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 96.21 ms
  - Grid size: [3 1]
  - Block size: [21 1 1]
- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 73.88 (Balanced Instruction per byte ratio: 3.79)
  - Achieved Occupancy: 0.02 (Theoretical Occupancy: 0.06)
- Also here the limiting factor for GTX 480 GPU is 96.21 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block

size during kernel call performance degrades and also the utilization of the GPU decreases.

- Summary profiling information of Tesla C2070 GPU
  - Number of calls: 1
  - GPU time: 89.43 ms
  - Grid size: [3 1]
  - Block size: [21 1 1]
- Limiting factor for Tesla C2050 GPU
  - Achieved instruction per byte ratio: 74.91
  - Achieved Occupancy: 0.04 (Theoretical Occupancy: 0.06)
- Also here the limiting factor for Tesla GPU is 89.43 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.
- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.
  - Longest Common Subsequence algorithm tends to search the longest sequence using backtracking method. Here we may not be able to resolve every dependencies which may lead to lack to optimization and also affect performance speedup.
- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the Tesla C2070 is 2988 MHz, hence more speedup is obtained on the Tesla C2070 GPU.

- Considering core clock which is highest in Tesla C2070 GPU leading to the best performance.
- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on Tesla C2070 GPU.
- Hence from all the above statics and parameters we can theoretically conclude that best performance can be obtained on Tesla C2070 GPU which is proved practically.

6.4 Kruskal's Algorithm

```
Let G = (V, E) be the given graph, with | V| = n

        {

            Start with a graph T = (V,phi) consisting of only the

            vertices of G and no edges; /* This can be viewed as n

            connected components, each vertex being one connected component */

       Arrange E in the order of increasing costs;

        for (i = 1, i<n - 1, i + +)

        { Select the next smallest cost edge;

        if (the edge connects two different connected components)

        add the edge to T;

        }

    }
```

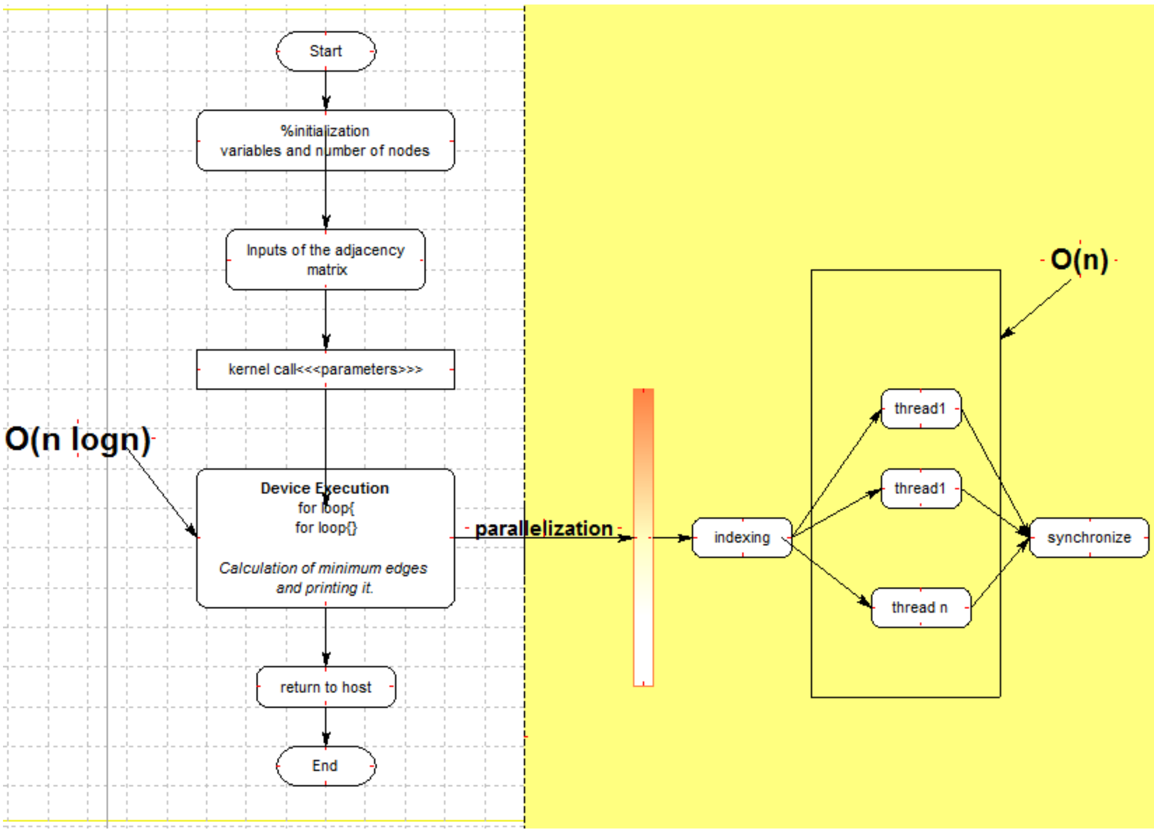


Figure 6.28: Task Graph

6.4.1 Comparative Study for Kruskal's Algorithm

The comparative study include execution of Kruskal's algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

Implementation on CPU

The fig. 6.28 & fig 6.29 shows the task graph and the output of the program and fig 6.30 shows the output in which it considers total time for executing the algorithm on CPU. It takes 833 ms to execute the program by taking the entire program under consideration and fig. 6.31 shows the output of the part of program executed on CPU

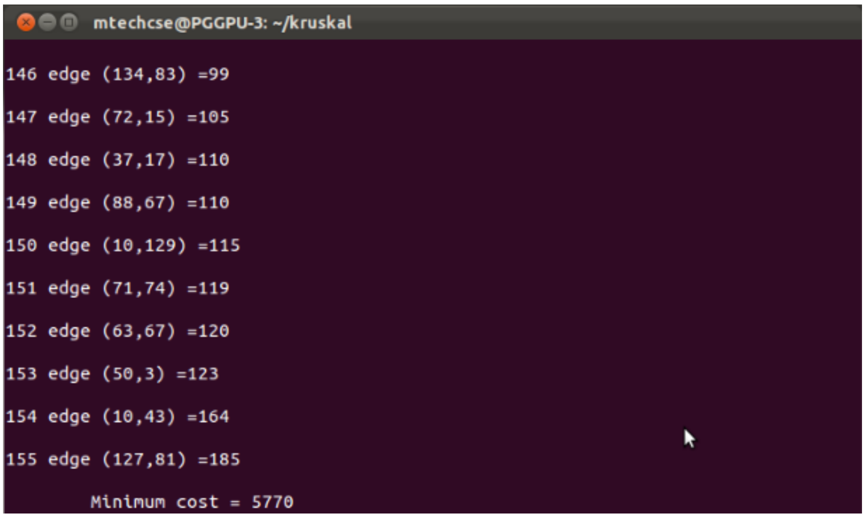


Figure 6.29: Knapsack Output

but it consists of only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken by the part of the program to execute and it takes 739 ms to execute the part of the program.

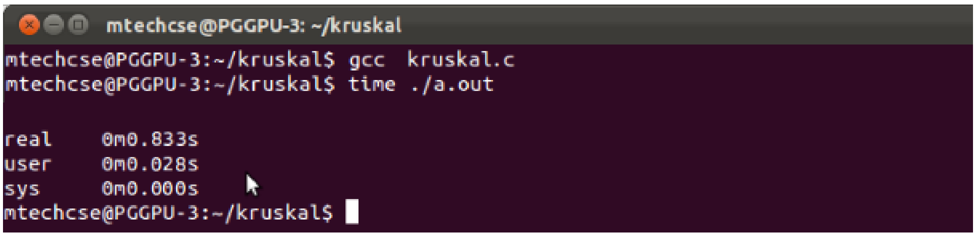


Figure 6.30: kruskal CPU Time considering entire program

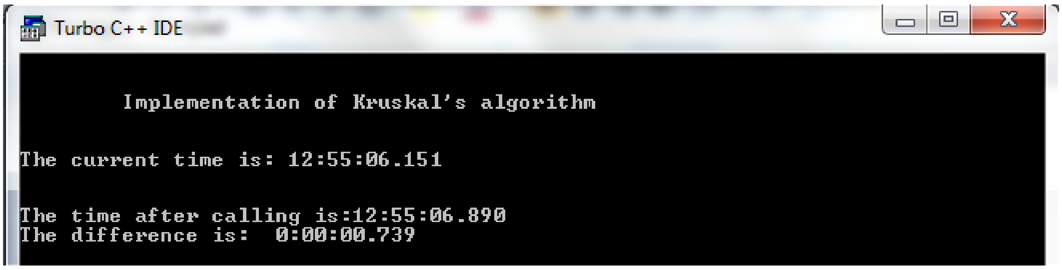


Figure 6.31: Kruskal CPU Time considering part of the program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 493 ms as shown in the fig. 6.32 and hence the speed up gain as compared with CPU is 340ms and the time taken to execute the program without considering memory transfer is 253 ms as shown in the fig. 6.33. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

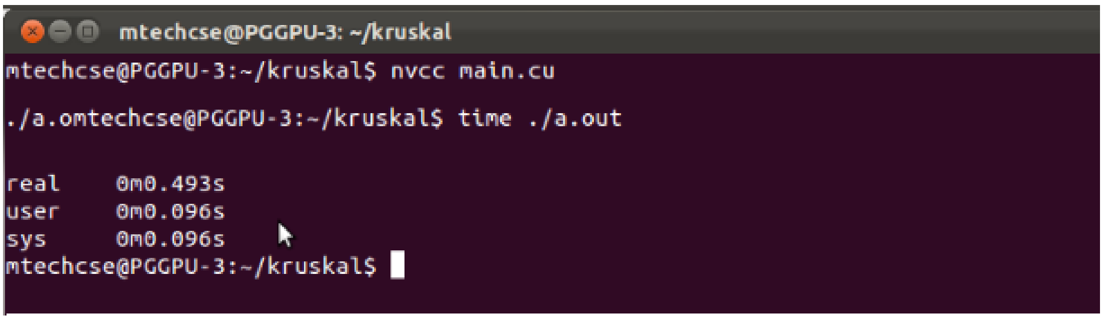
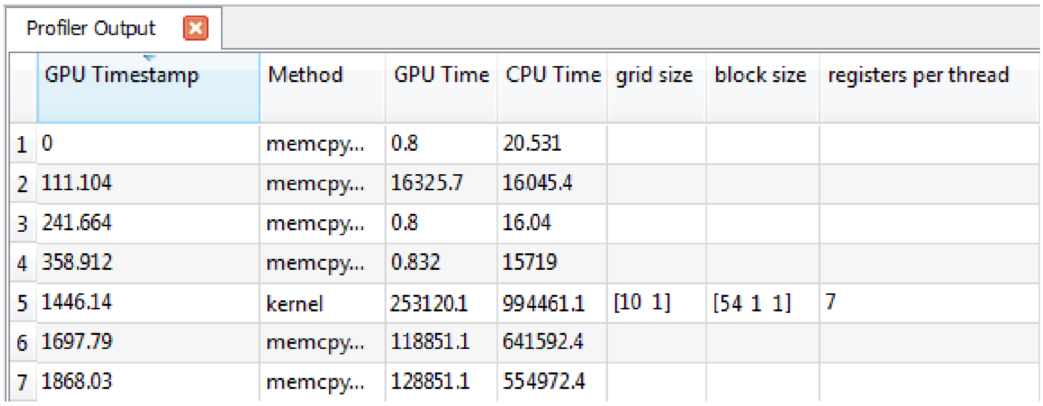


Figure 6.32: GTX 480 Considering Memory Transfer



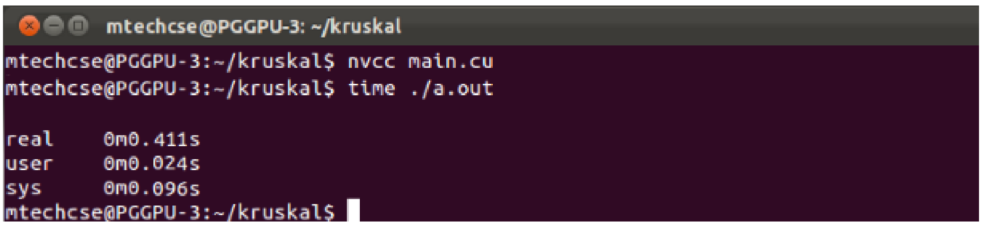
Profiler Output

	GPU Timestamp	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1	0	memcpy...	0.8	20.531			
2	111.104	memcpy...	16325.7	16045.4			
3	241.664	memcpy...	0.8	16.04			
4	358.912	memcpy...	0.832	15719			
5	1446.14	kernel	253120.1	994461.1	[10 1]	[54 1 1]	7
6	1697.79	memcpy...	118851.1	641592.4			
7	1868.03	memcpy...	128851.1	554972.4			

Figure 6.33: GTX 480 Considering Without Memory Transfer

Implementation on Tesla

The total time considering memory transfer in Tesla C2070 GPU is 411 ms as shown in the fig. 6.34 and hence the speed up gain as compared with CPU is 422 ms and the time taken to execute the program without considering memory transfer is 213 ms as shown in the fig. 6.35. In order to obtain the result without memory transfer,i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla C2070 GPU compared to that of CPU.



```
mtechcse@PGGPU-3: ~/kruskal
mtechcse@PGGPU-3:~/kruskal$ nvcc main.cu
mtechcse@PGGPU-3:~/kruskal$ time ./a.out

real    0m0.411s
user    0m0.024s
sys     0m0.096s
mtechcse@PGGPU-3:~/kruskal$
```

Figure 6.34: Tesla Considering Memory Transfer)

Profiler Output

Method	GPU Time	CPU Time	grid size	block size	registers per thread
memcpy...	0.6	20.531			
memcpy...	125.7	13045.4			
memcpy...	0.6	16.04			
memcpy...	0.832	15719			
kernel	213150.4	884461.1	[10 1]	[54 1 1]	7
memcpy...	94891.1	6415.8			
memcpy...	124891.1	554972.4			

Figure 6.35: Tesla Considering Without Memory Transfer

Quantitative Comparison

- Here the CPU takes O(n log n) time time then compared to algorithm executed on GPU's which is been reduced to O(n) times.
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 253 ms
  - Grid size: [10 1]
  - Block size: [53 1 1]
- Limiting factor for GTX 480 GPU
  - Achieved instruction per byte ratio: 176.30
  - Achieved Occupancy: 0.6 (Theoretical Occupancy: 0.9)
- Also here the limiting factor for GTX 480 GPU is 253 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.
- Summary profiling information of Tesla C2070 GPU

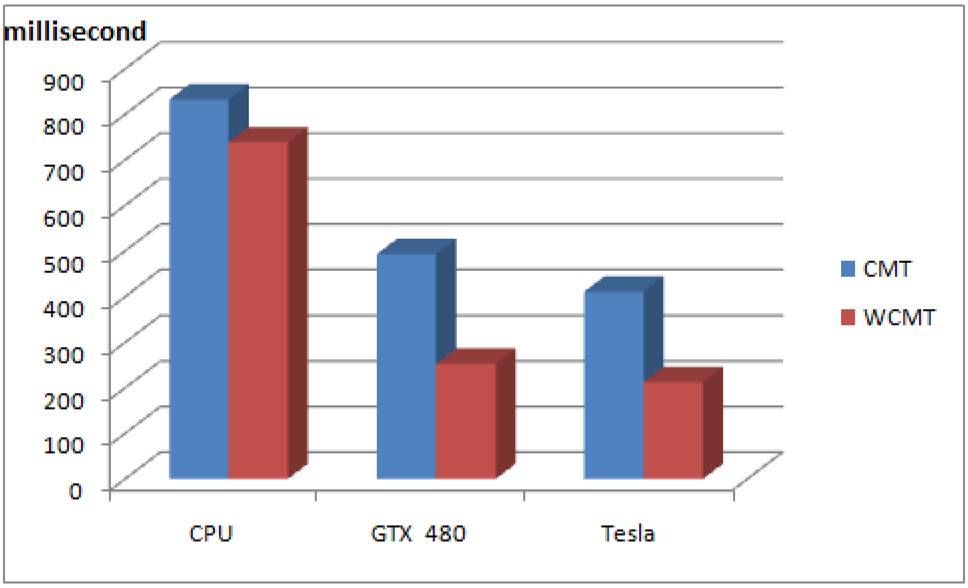


Figure 6.36: Graph Comprising of speedup

- Number of calls: 1
- GPU time: 213 ms
- Grid size: [10 1]
- Block size: [53 1 1]

- Limiting factor for Tesla C2070 GPU
  - Achieved instruction per byte ratio: 193.51
  - Achieved Occupancy: 0.72 (Theoretical Occupancy: 0.91)

- Also here the limiting factor for Tesla C2070 GPU is 213 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.

- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.

  - The derived statistics assume all instruction are single precision floating point instruction. If double precision floating point instruction are used then the limiting factor may become incorrect.

- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the Tesla C2070 is 2988 MHz, hence more speedup is obtained on the Tesla C2070 GPU.

- Considering core clock which is highest in Tesla leading to the best performance.

- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on Tesla GPU.

- Hence from all the above statics and parameters we can theoretically conclude that best performance can be obtained on Tesla GPU which is proved practically.

6.5 Insertion Sort Algorithm

```
begin
    for i := 1 to length(A)-1 do
    begin
        value := A[i];
        j := i - 1;
        done := false;
        repeat
            { To sort in descending order simply reverse
              the operator i.e. A[j] < value }
            if A[j] > value then
            begin
                A[j + 1] := A[j];
                j := j - 1;
                if j < 0 then
                    done := true;
            end
            else
                done := true;
        until done;
        A[j + 1] := value;
    end;
end;
```

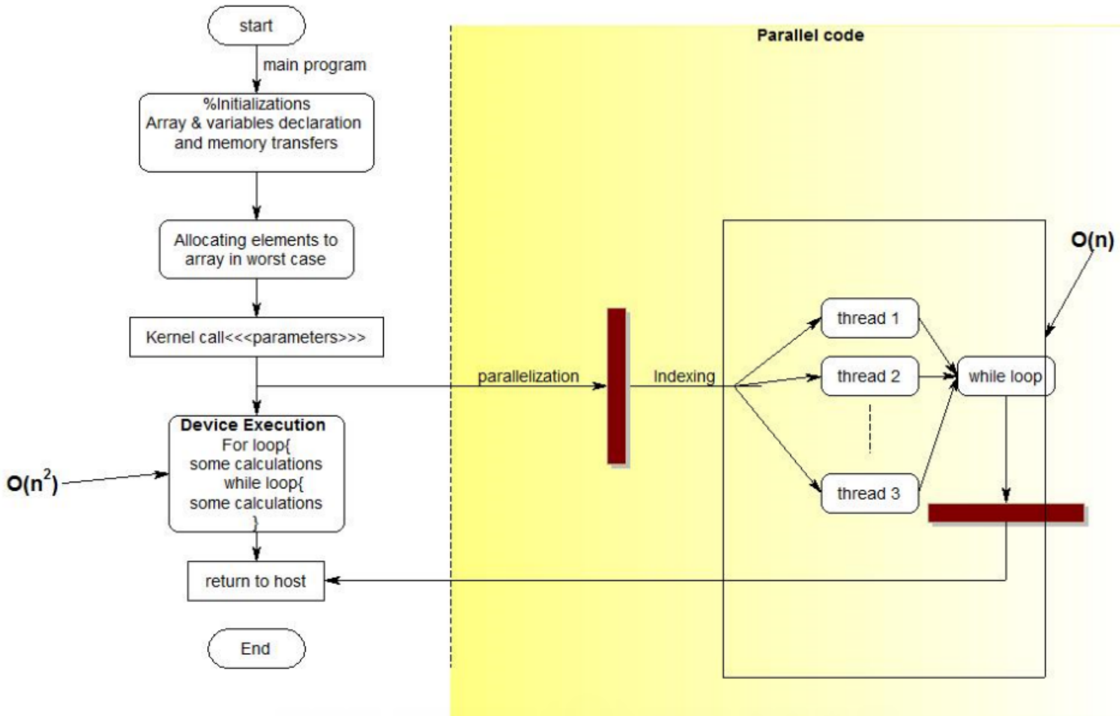


Figure 6.37: Task Graph

6.5.1 Comparative Study for Insertion Sort Algorithm

The Comparative study include execution of Insertion Sort algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

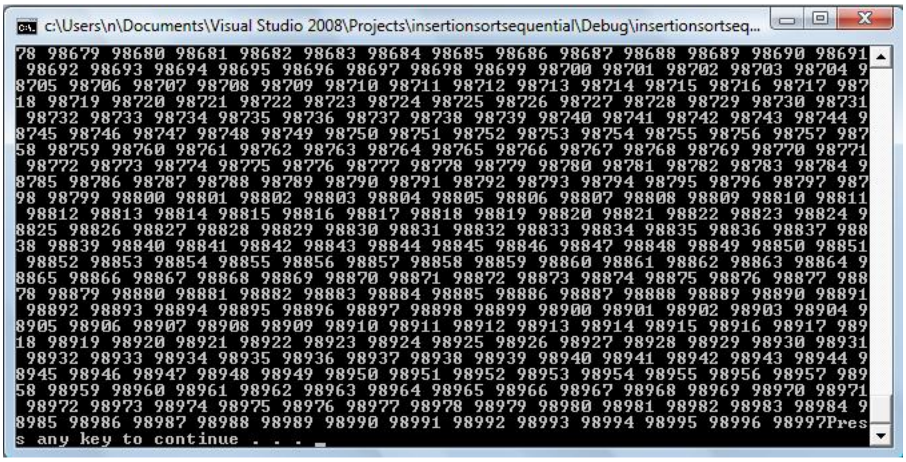


Figure 6.38: Insertion Sort Output

Implementation on CPU

The fig. 6.37 & fig. 6.38 shows the task graph and the output of the program and fig. 6.39 shows the output in which it considers total time for executing the algorithm on CPU. It takes 859 ms to execute the program by taking the entire program under consideration and fig. 6.40 shows the output of the part of program executed on CPU but it consists of only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken for the part of the program to execute and it takes 502 ms to execute the part of the program.

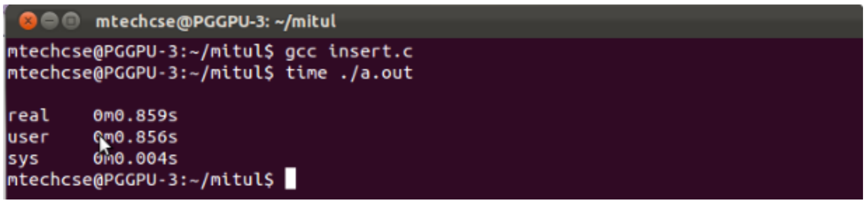


Figure 6.39: Insertion Sort CPU Time considering entire program

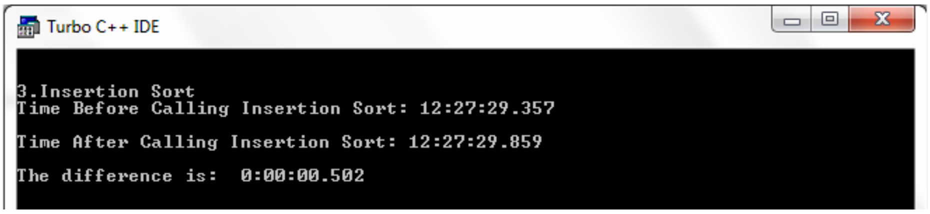


Figure 6.40: Insertion Sort CPU Time considering part of the program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 91 ms as shown in the fig. 6.41 and hence the speed up gain as compared with CPU is 768 ms and the time taken to execute the program without considering memory transfer is 38.44ms as shown in the fig. 6.42. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

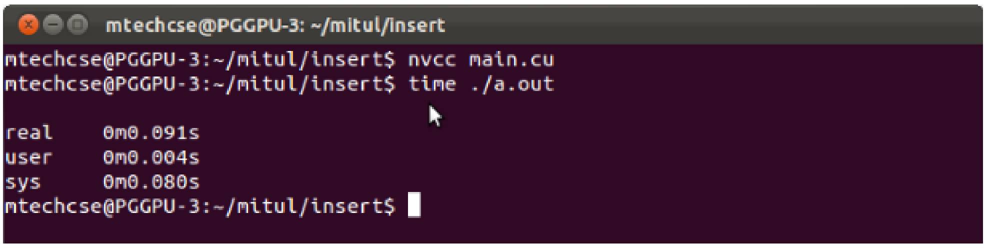
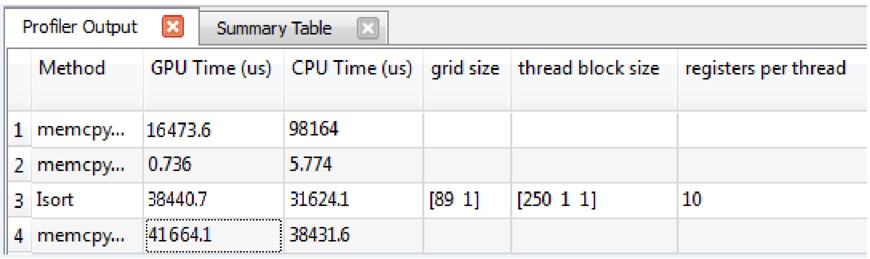


Figure 6.41: GTX 480 Considering Memory Transfer

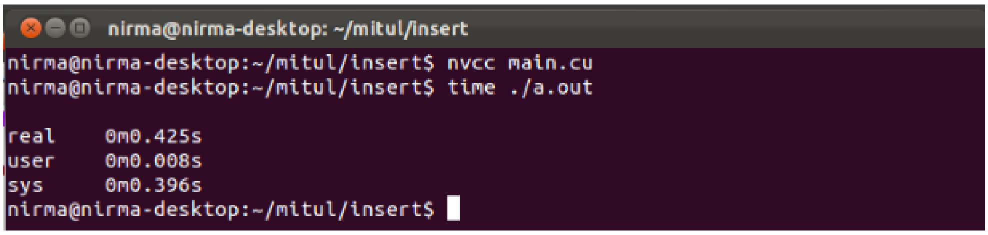

Profiler Output | Summary Table

	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread
1	memcpy...	16473.6	98164			
2	memcpy...	0.736	5.774			
3	Isort	38440.7	31624.1	[89 1]	[250 1 1]	10
4	memcpy...	41664.1	38431.6			

Figure 6.42: GTX 480 Considering Without Memory Transfer

Implementation on Tesla C2070

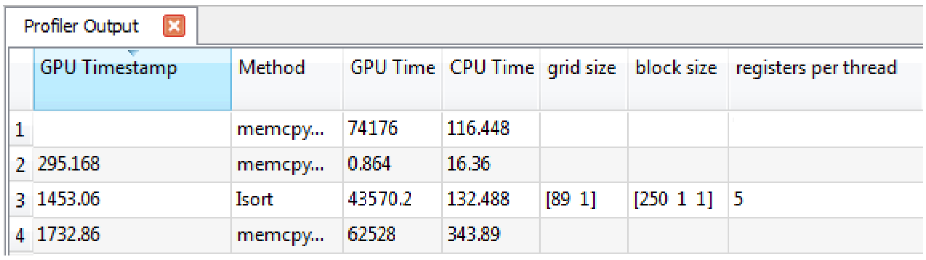
The total time considering memory transfer in Tesla C2070 GPU is 425 ms as shown in the fig. 6.43 and hence the speed up gain as compared with CPU is 425 ms and the time taken to execute the program without considering memory transfer is 43.57 ms as shown in the fig. 6.44. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla C2070 GPU compared to that of CPU.



```
nirma@nirma-desktop: ~/mitul/insert
nirma@nirma-desktop:~/mitul/insert$ nvcc main.cu
nirma@nirma-desktop:~/mitul/insert$ time ./a.out

real    0m0.425s
user    0m0.008s
sys     0m0.396s
nirma@nirma-desktop:~/mitul/insert$
```

Figure 6.43: Tesla Considering Memory Transfer



Profiler Output

	GPU Timestamp	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1		memcpy...	74176	116.448			
2	295.168	memcpy...	0.864	16.36			
3	1453.06	Isort	43570.2	132.488	[89 1]	[250 1 1]	5
4	1732.86	memcpy...	62528	343.89			

Figure 6.44: Tesla Considering Without Memory Transfer

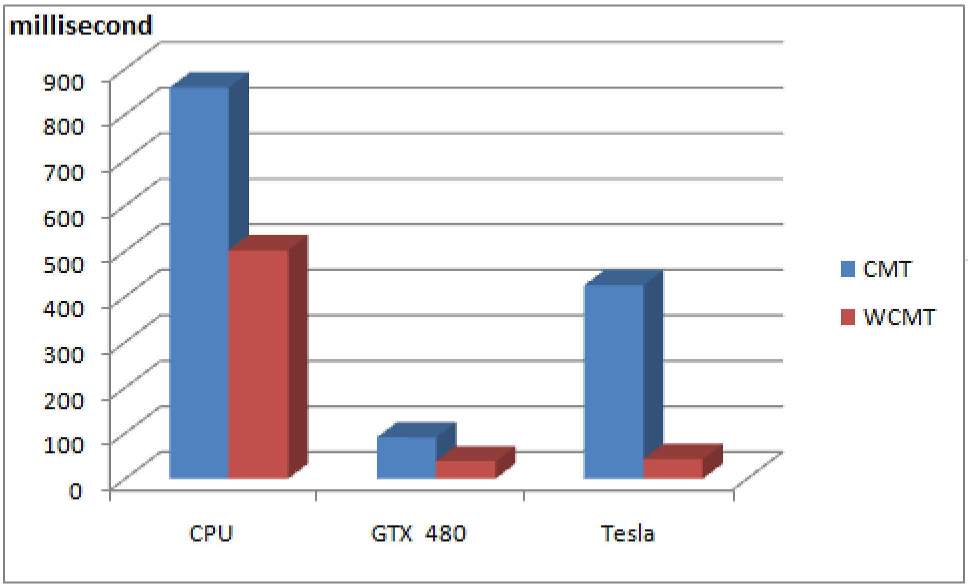


Figure 6.45: Graph Comprising of speedup

Quantitative Comparison

- Here the CPU takes a lot of time then compared to algorithm executed on GPU's which is been reduced from O(log n) to number of threads in parallel.
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 38.84 ms
  - Grid size: [89 1]
  - Block size: [250 1 1]
- Limiting factor for GTX 480 GPU
  - Achieved instruction per byte ratio: 8.51
  - Achieved Occupancy: 0.86 (Theoretical Occupancy: 1.00)
- Also here the limiting factor for GTX 480 GPU is 38.84 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block

size during kernel call performance degrades and also the utilization of the GPU decreases.

- Summary profiling information of Tesla C2070 GPU
  - Number of calls: 1
  - GPU time: 43.57 ms
  - Grid size: [89 1]
  - Block size: [250 1 1]
- Limiting factor for Tesla GPU
  - Achieved instruction per byte ratio: 7.16
  - Achieved Occupancy: 0.83 (Theoretical Occupancy: 0.92)
- Also here the limiting factor for Tesla C2070 GPU is 34.261 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.
- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.
  - Insertion Sort reduces to O(n), but there are still dependencies which if removed will lead to performance gain.
- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the Tesla C2070 is 2988 MHz, hence more speedup is obtained on the Tesla C2070 GPU.

- Considering core clock which is highest in Tesla leading to the best performance, but for the same number of blocks and threads it tends to give best performance on GTX 480 rather than Tesla GPU.
- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on GTX 480.
- Here if we consider the limiting factor for the Tesla then the best performance is obtained on tesla with 75 consisting of 230 threads which is the best performance speedup obtained for this algorithm on this GPU.

6.6 Selection sort algorithm

```
Function selection(Array a,n)
{
 minindex = i
 minvalue = a[i]
for(j=i+1 upto n)
{
    if (a[j]<minval)
    {
    minval<-a[j]
    minindex<-j
    }
}
temp<-a[minindex]
a[minindex]<-a[i]
a[i]<-temp
}
```

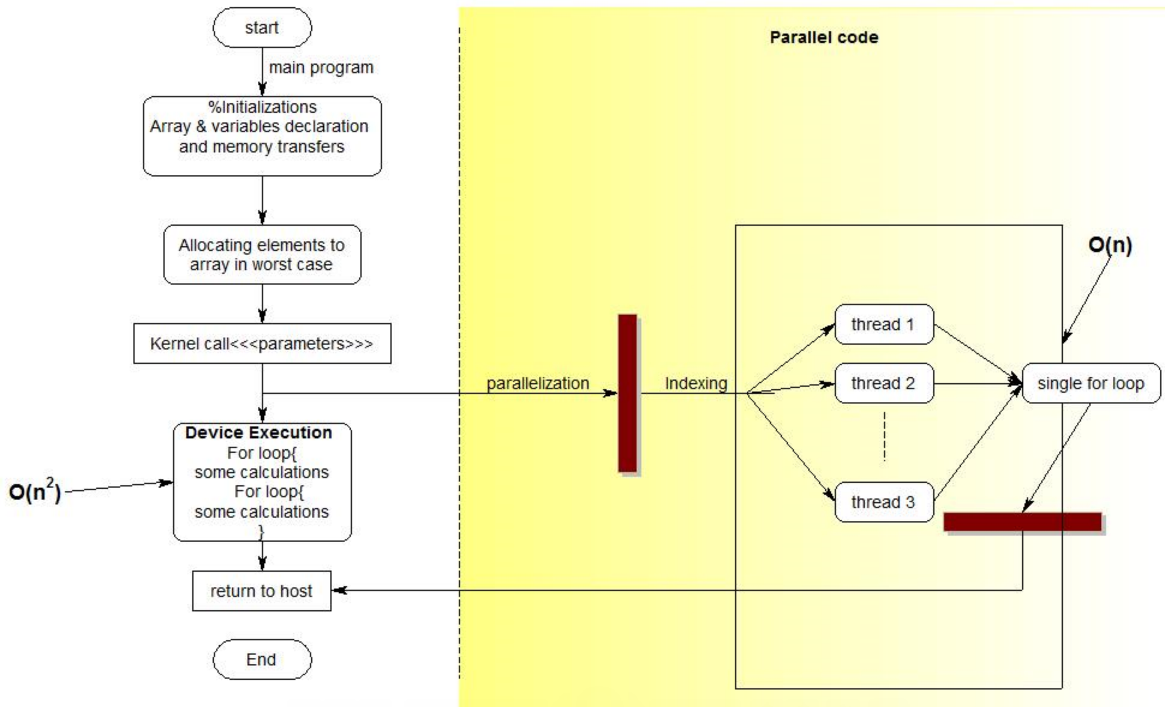


Figure 6.46: Task Graph

6.6.1 Comparative Study for Selection Sort Algorithm

The comparative study include execution of Selection Sort algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

Implementation on CPU

The fig. 6.46 & fig. 6.47 shows the task graph and the output of the program and fig. 6.48 shows the output in which it considers total time for executing the algorithm on CPU. It takes 666 ms to execute the program by taking the entire program under consideration and fig. 6.49 shows the output of the part of program executed on CPU but it consists of only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken for

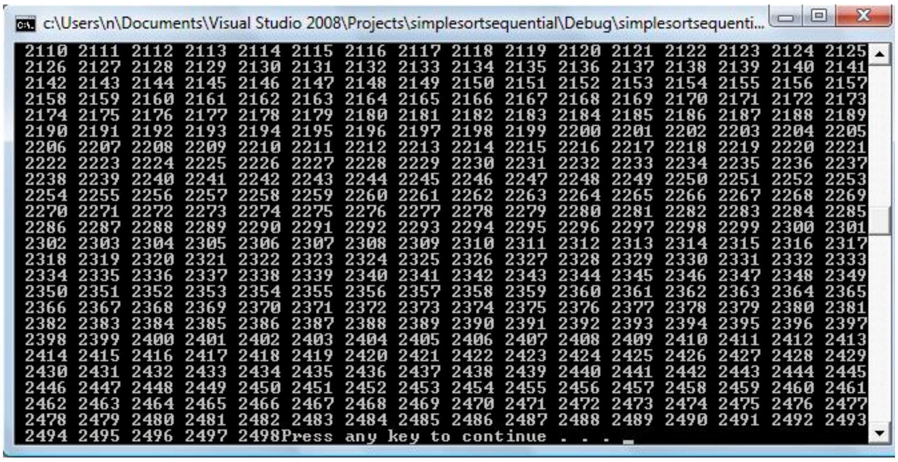


Figure 6.47: Selection Sort Output

the part of the program to execute and it takes 590 ms to execute the part of the program.

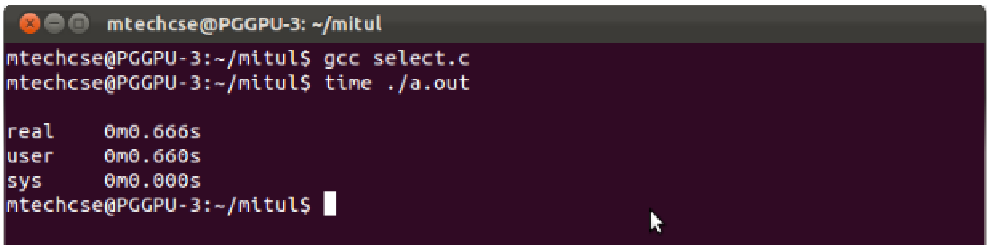


Figure 6.48: Insertion Sort CPU Time considering entire program

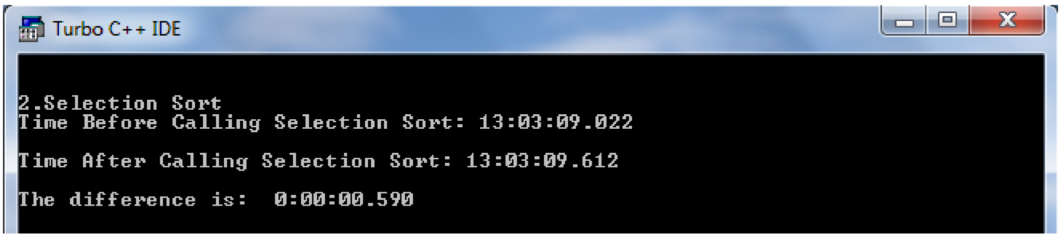


Figure 6.49: Insertion Sort CPU Time considering part of the program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 105 ms as shown in the fig. 6.50 and hence the speed up gain as compared with CPU is 561ms and the time taken to execute the program without considering memory transfer is 27.55 ms as shown in the fig. 6.51. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

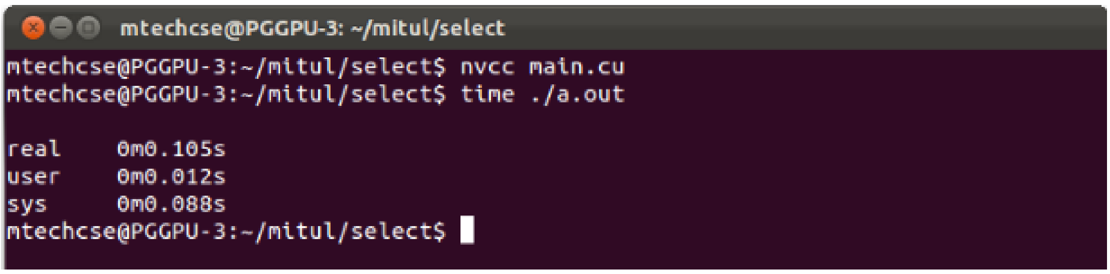
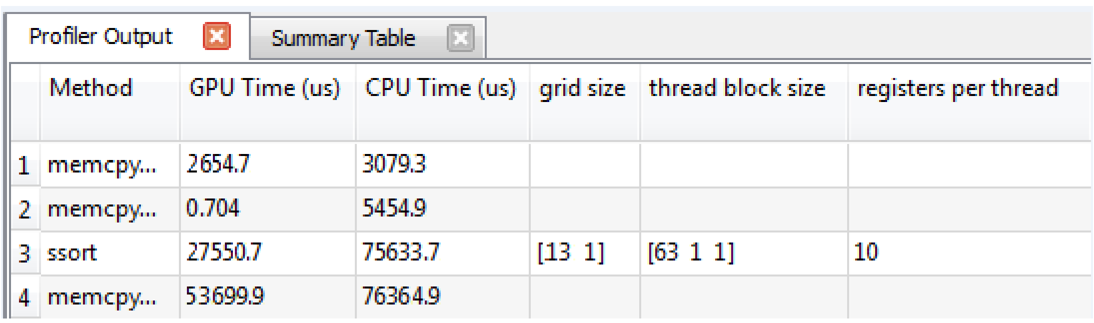


Figure 6.50: GTX 480 Considering Memory Transfer



Profiler Output | Summary Table

	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread
1	memcpy...	2654.7	3079.3			
2	memcpy...	0.704	5454.9			
3	ssort	27550.7	75633.7	[13 1]	[63 1 1]	10
4	memcpy...	53699.9	76364.9			

Figure 6.51: GTX 480 Considering Without Memory Transfer

Implementation on Tesla

The total time considering memory transfer in Tesla GPU is 49 ms as shown in the fig. 6.52 and hence the speed up gain as compared with CPU is 519 ms and the time taken to execute the program without considering memory transfer is 12.63 ms as shown in the fig. 6.53. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla C2070 GPU compared to that of CPU.

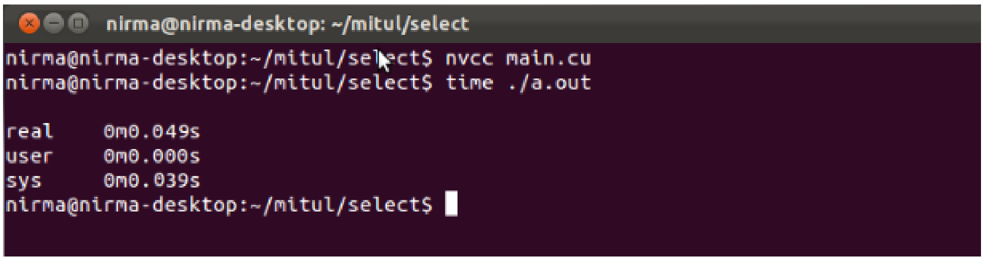
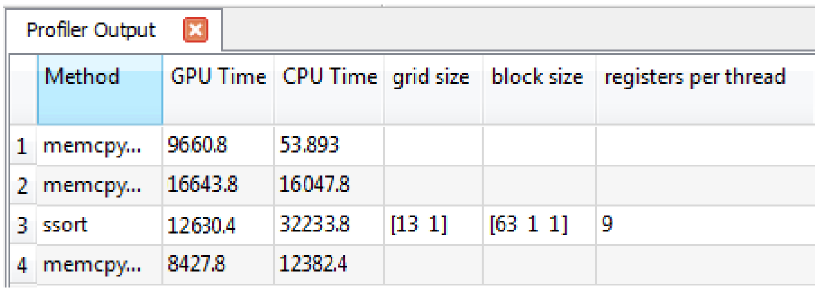


Figure 6.52: Tesla Considering Memory Transfer


Profiler Output

	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1	memcpy...	9660.8	53.893			
2	memcpy...	16643.8	16047.8			
3	ssort	12630.4	32233.8	[13 1]	[63 1 1]	9
4	memcpy...	8427.8	12382.4			

Figure 6.53: Tesla Considering Without Memory Transfer

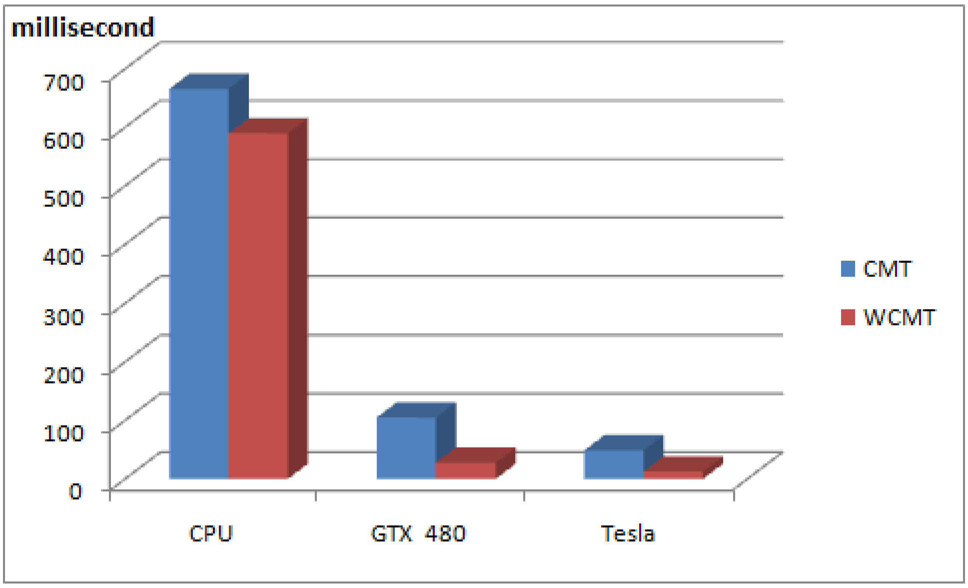


Figure 6.54: Graph Comprising of speedup

Quantitative Comparison

- Here the CPU takes a lot of time then compared to algorithm executed on GPU's which is been reduced from $O(n^2)$ to number $O(n)$.
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 27.55 ms
  - Grid size: [13 1]
  - Block size: [63 1 1]
- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 14.53
  - Achieved Occupancy: 0.091 (Theoretical Occupancy: 0.10)
- Also here the limiting factor for GTX 480 GPU is 0.053 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block

size during kernel call performance degrades and also the utilization of the GPU decreases.

- Summary profiling information of Tesla GPU
  - Number of calls: 1
  - GPU time: 12.63ms
  - Grid size: [13 1]
  - Block size: [63 1 1]
- Limiting factor for Tesla C2070 GPU
  - Achieved instruction per byte ratio: 13.47 (Balanced Instruction per byte ratio: 3.79)
  - Achieved Occupancy: 0.13 (Theoretical Occupancy: 0.20)
- Also here the limiting factor for Tesla C2070 GPU is 0.045 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.
- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.
  - Since the loops for the sorting are reduced which obtains the speedup performance, but also there are dependencies due to which it may not be possible to achieve prosper GPU utilization.
- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the Tesla C2070 is 2988 MHz, hence more speedup is obtained on the Tesla C2070 GPU.

- Considering core clock which is highest in Tesla C2070 GPU leading to the best performance.
- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on Tesla C2070 GPU.
- Hence from all the above statics and parameters we can theoretically conclude that best performance can be obtained on Tesla C2070 GPU which is proved practically.

6.7 Bubble Sort algorithm

```
Function bubble(Array a,i ,j)
for i = i:n,
    swapped = false
    for j = n:i+1,
    if a[j] < a[j-1],
        swap a[j,j-1]
        swapped = true
->invariant:a[1...i] in final position
break if not swapped
end
```

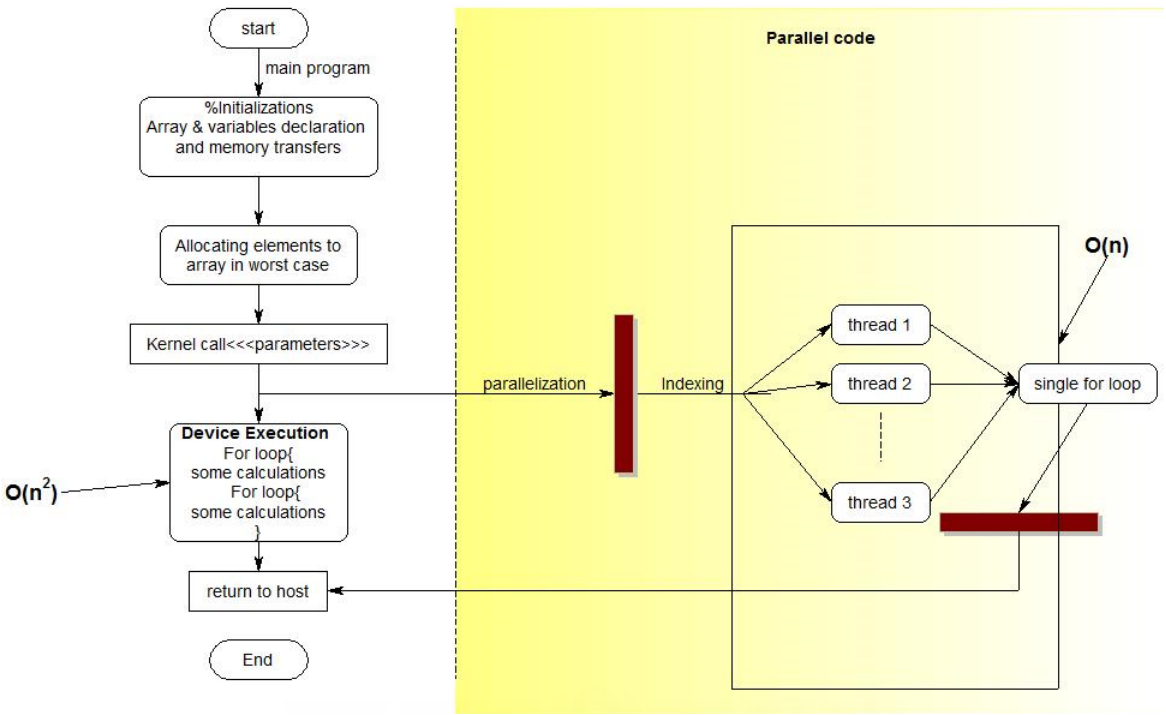


Figure 6.55: Task Graph

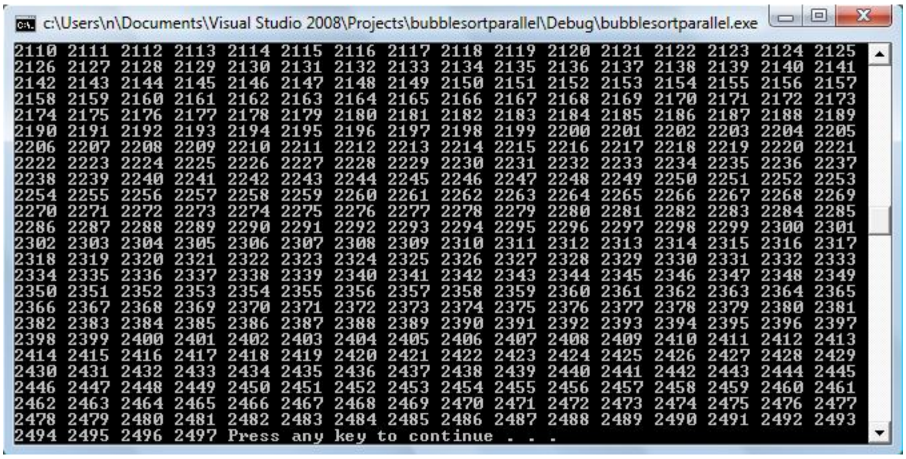


Figure 6.56: Bubble Sort Output

6.7.1 Comparative study for Bubble sort Algorithm

The comparative study include execution of Bubble Sort algorithm between CPU and two different GPU's in parallel forms considering two benchmark criteria.

a. Considering Memory Transfer time.

b. Without considering Memory transfer time.

Implementation on CPU

The fig. 6.55 & fig. 6.56 shows the task graph and the output of the program and fig. 6.57 shows the output in which it considers total time for executing the algorithm on CPU. It takes 1416 ms to execute the program by taking the entire program under consideration and fig. 6.58 shows the output of the part of program executed on CPU but it consists of only that part which gets executed by GPU without considering memory transfer. We have taken C language time function to get the time taken for the part of the program to execute and it takes 1016 ms to execute the part of the program.

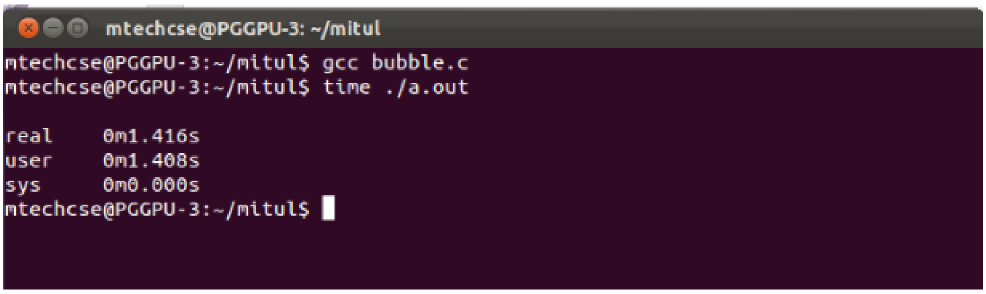


Figure 6.57: Bubble Sort CPU Time considering entire program)

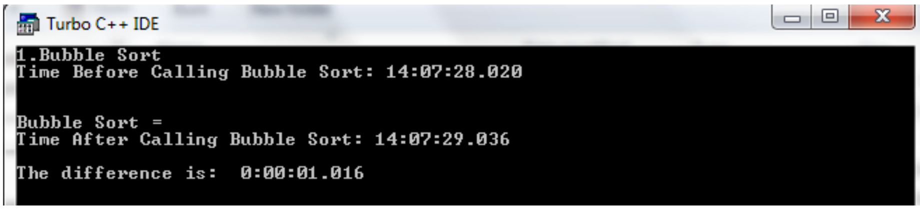


Figure 6.58: Bubble Sort CPU Time considering part of the program

Implementation on GTX 480

The total time considering memory transfer in GTX 480 GPU is 534 ms as shown in the fig. 6.59 and hence the speed up gain as compared with CPU is 882 ms and the time taken to execute the program without considering memory transfer is 149 ms as shown in the fig. 6.60 In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on GTX 480 GPU compared to that of CPU.

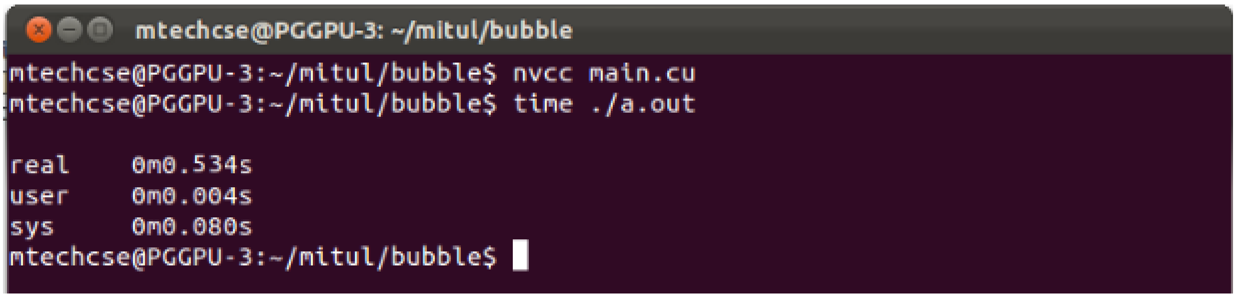
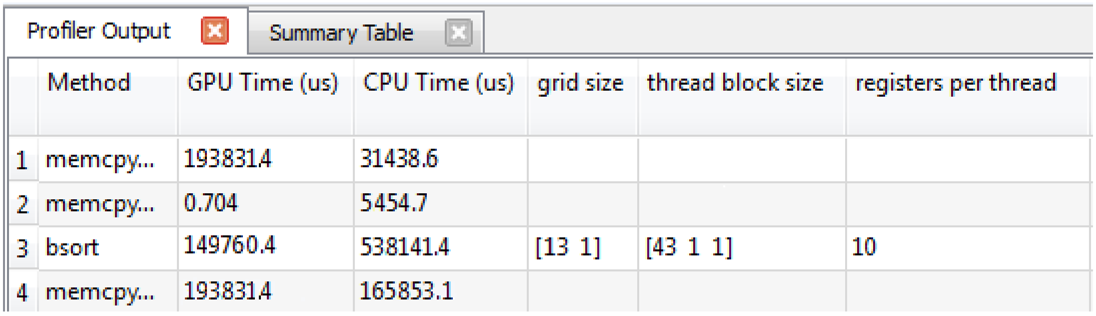


Figure 6.59: GTX 480 Considering Memory Transfer



	Method	GPU Time (us)	CPU Time (us)	grid size	thread block size	registers per thread
1	memcpy...	1938314	31438.6			
2	memcpy...	0.704	5454.7			
3	bsort	149760.4	538141.4	[13 1]	[43 1 1]	10
4	memcpy...	1938314	165853.1			

Figure 6.60: GTX 480 Without Considering Memory Transfer

Implementation on Tesla C2070

The total time considering memory transfer in Tesla C2070 GPU is 497 ms as shown in the fig. 6.61 and hence the speed up gain as compared with CPU is 919 ms and the time taken to execute the program without considering memory transfer is 117 ms as shown in the fig. 6.62. In order to obtain the result without memory transfer, i used the visual studio as the framework and CUDA Visual Profiler gives me the required output. Hence there is wide gain in speedup on Tesla C2070 GPU compared to that of CPU.

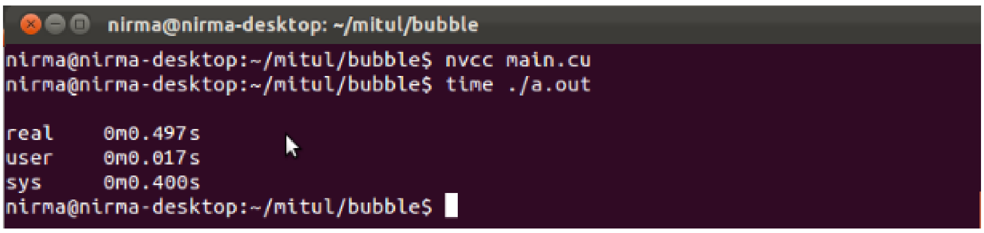


Figure 6.61: Tesla Considering Memory Transfer

Profiler Output | Summary Table

	Method	GPU Time	CPU Time	grid size	block size	registers per thread
1	memcpy...	196435.4	522893.4			
2	memcpy...	0.864	16.04			
3	bsort	116543.4	183495.4	[13 1]	[43 1 1]	5
4	memcpy...	142460.8	96559.6			

Figure 6.62: Tesla Considering Without Memory Transfer

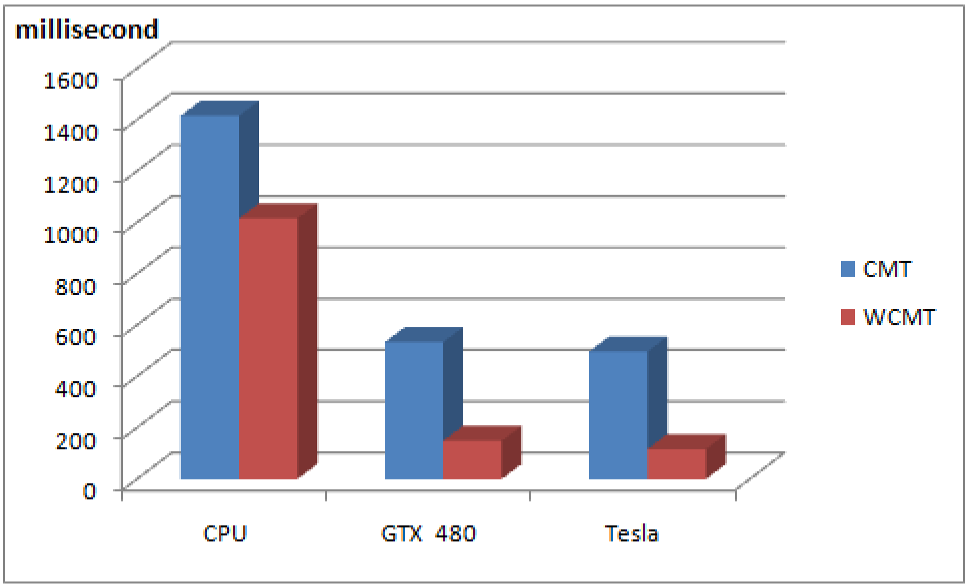


Figure 6.63: Graph Comprising of speedup

Quantitative Comparison

- Here the CPU takes a lot of time then compared to algorithm executed on GPU's which is been reduced from O(log n) to number of threads in parallel.
- Summary profiling information of GTX 480
  - Number of calls: 1
  - GPU time: 534 ms
  - Grid size: [13 1]
  - Block size: [43 1 1]

- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 24.29
  - Achieved Occupancy: 0.02 (Theoretical Occupancy: 0.06)
- Also here the limiting factor for GTX 480 GPU is 534 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.
- Summary profiling information of Tesla GPU
  - Number of calls: 1
  - GPU time: 497 ms
  - Grid size: [13 1]
  - Block size: [43 1 1]
- Limiting factor for GTX 480
  - Achieved instruction per byte ratio: 329.64
  - Achieved Occupancy: 0.21 (Theoretical Occupancy: 0.33)
- Also here the limiting factor for Tesla GPU is 497 ms in which it occupies maximum utilization of GPU, if there is any alteration of threads and block size during kernel call performance degrades and also the utilization of the GPU decreases.
- Factors that may affect the performance gain:
  - The derived statistics are collected in different runs of applications. This may cause some inaccuracy.

  - The derived statistics assume all instruction are single precision floating point instruction. If double precision floating point instruction are used then the limiting factor may become incorrect.

- Processor clock rate in GTX 480 is 1401 MHz, while in Tesla C2070 is 1494 MHz. Also Memory transfer rate in GTX 480 is 1848 MHz,and that of the Tesla C2070 is 2988 MHz, hence more speedup is obtained on the Tesla C2070 GPU.

- Considering core clock which is highest in Tesla leading to the best performance.

- Hence the best result for this algorithm that can be obtained considering both the benchmark criteria is obtained on Tesla GPU.

- Hence from all the above statics and parameters we can theoretically conclude that best performance can be obtained on Tesla C2070 GPU which is proved practically.

6.8 Performance graph

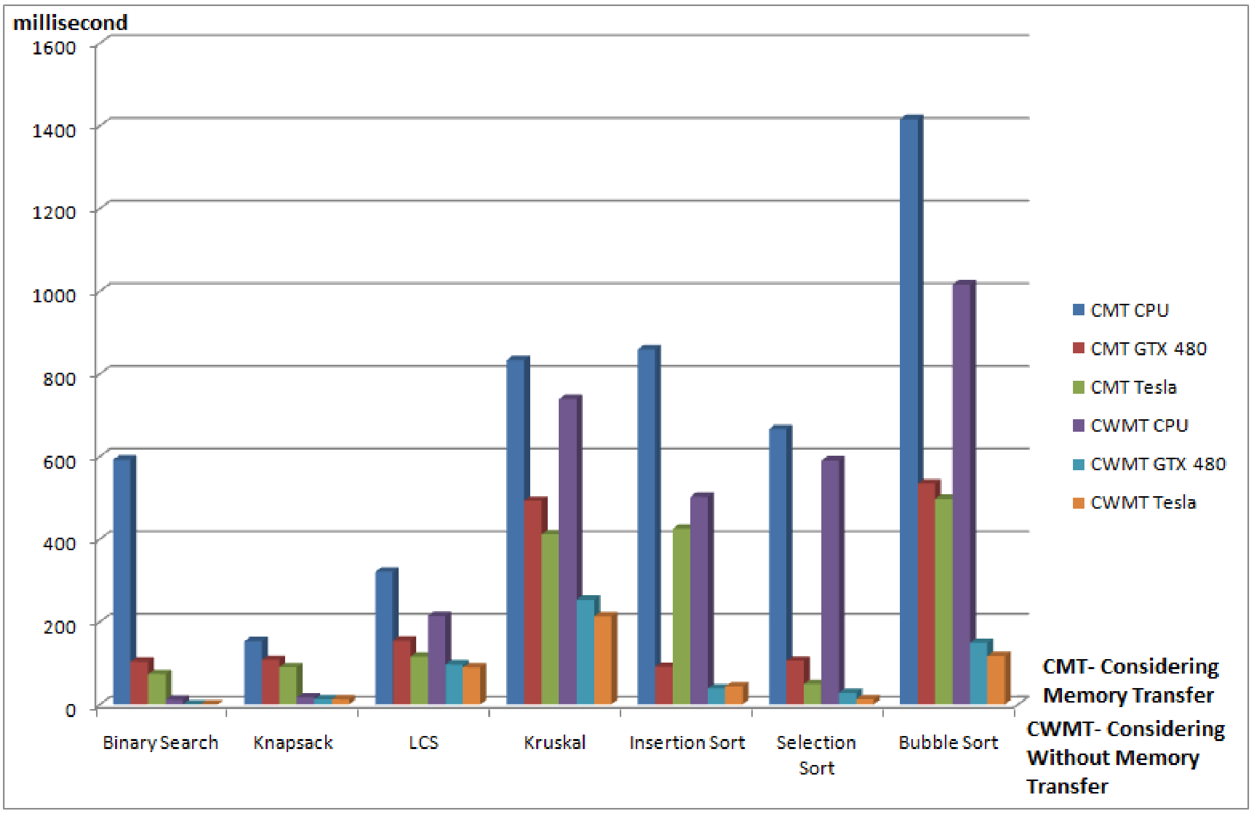


Figure 6.64: Performance Graph(Parallel)

- The graph shows the time taken to execute the algorithms on the CPU and GPU's.
- From the graph, i can conclude that best performance is obtained on Tesla C2070 GPU in most of the cases with both benchmarks taken under consideration.
- Due to some amount of dependencies present leading to speedup performance gain can be achieved further by enhancement to the algorithms.

Chapter 7

Conclusion

By execution of the complex algorithms and obtaining its statistics, i can conclude that GPU's having more number of cores with high clock frequency achieves gain in speedup. The above implementation results concludes that algorithms implemented on GTX 480 and Tesla C2070 takes less time to execute as compared to CPU leading to speedup gains in microseconds. Even i can conclude when these algorithms are optimized by removing dependencies, this enhanced algorithms may achieve more speedup. Also executions of such combinatorial and complex execution loads on high end GPU's leads to more efficient speedup gain.

Chapter 8

Future Work

- Further enhancement can be done to these algorithms in order to achieve more speedup.
- Obtaining different methods for solutions.
- Use of such strategic enhancement in the field of Graph Theory.
- Currently only one kernel can run at time on the hardware device, future work will include extension to multiple kernels simultaneously, so that more parallelism can be achieved.

References

[1] CUDA Tutorials by NVIDIA http://www.nvidia.com/object/cuda_education.html.

[2] NVIDIA CUDA Programming Guide Version 2.2.1.

[3] OpenCL Tutorials by NVIDIA http://www.nvidia.com/object/cuda_opencl.html.

[4] Cuda_Architecture_Overview @2009 by NVIDIA corporation. http://developer.download.nvidia.com/compute/cuda/docs/CUDA_Architecture_Overview.pdf.

[5] AMD - Introduction to OpenGL 3.0 by developers.amd.com.

[6] B. R. Neha Patil, "Fast and parallel implementation of image processing algorithm using cuda technology on gpu hardware", tech. rep., Department of Electrical & Computer and Systems Engineering, Rensselaer Polytechnic Institute,Troy, NY 12180-3590.

[7] D. L. N. Research, "nvidia gpu architecture & implications", NVIDIA Corporation 2007. /cis788-97/h1trnd.htm.

[8] Shane Ryoo, Christopher I. Rodrigue, Sara S. Baghsorkhi, "Optimizing the Fast Fourier Transform on a Multi-core Architecture", University of Illinois at Urbana Champaign, NVIDIA Corporation, 2007.

[9] N. P. Karunadasa & D. N. Ranasinghe, "On the Comparative Performance of Parallel Algorithms on Small GPU/CUDA Clusters", University of Colombo School of Computing, Sri Lanka,2008.

Index