LEBANESE AMERICAN UNIVERSITY

A PARALLEL SEARCH TREE ALGORITHM FOR VERTEX
COVER ON GRAPHICAL PROCESSING UNITS

By

RASHAD KARIM KABBARA

A thesis
Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science

School of Arts and Sciences
February 2013

Lebanese American University
School of Arts & Sciences

Thesis Proposal Form

Name of Student: *Rashad Kabbara* I.D.#: *200803588*

Department: *Computer Science and Mathematics*

On *May 11, 2012*, has presented a Thesis proposal entitled:

Parallel Search-Tree Algorithms on Graphical Processing Units

in the presence of the Committee members and Thesis Advisor:

Faisal Abu Khzam, *May 11, 2012*
(Name, signature, and date of the Thesis Advisor)

Ramzi Haraty *11/5/2012*
(Name, signature, and date of Committee Member)

Haidar Harmanani *11/5/2012*
(Name, signature, and date of Committee Member)

Comments/Remarks/Conditions to Proposal:

None.

Date: *14/5/2012* Acknowledged by _____
(Dean of School of Arts & Sciences)

cc: Department Chair
       Thesis Advisor
       Student
       Dean of Graduate Studies/School Dean

ii

LAU

Lebanese American University

LEBANESE AMERICAN UNIVERSITY
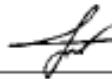School of Arts and Sciences - Beirut Campus

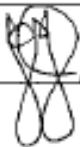### Thesis Defense Result Form

Name of student: Rashad Kabbara   I.D: 200803588

Program / Department: Master of Science in Computer Science

Date of thesis defense: 1-2-2013

Thesis title: A Parallel Search Tree Algorithm for Vertex Cover on Graphical Processing

Result of Thesis defense:

☑ Thesis was successfully defended. Passing grade is granted

☐ Thesis is approved pending corrections. Passing grade to be granted upon review and approval by thesis Advisor

☐ Thesis is not approved. Grade NP is recorded

Committee Members:

Advisor: Faisal Abu Khzam
                     (Name and Signature)

Committee Member: Ramzi A Haraty
                     (Name and Signature)

Committee Member: Haidar Harmanani
                     (Name and Signature)

Advisor's report on completion of corrections (if any):

Changes Approved by Thesis Advisor: Dr.Faisal Abu-Khzam   Signature: _____

Date: _____   Acknowledge by _____
                             (Dean, School of Arts & Sciences)

Cc: Registrar, Dean, Chair, Advisor, Student

iii

LEBANESE AMERICAN UNIVERSITY

School of Arts and Sciences - Beirut Campus

Thesis Approval Form

Student Name: Rashad Kabbara   I.D. #: 200803588

Thesis Title   : A Parallel Search Tree Algorithm for Vertex Cover on Graphical Processing

Units

Program   :   Master of Science in Computer Science

Department   :   Computer Science & Mathematics...
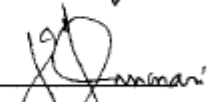
School   :   **School of Arts and Sciences**

Approved by :

Thesis Advisor: Dr. Faisal Abu-Khzam   Signature :

Member   :   Dr. Haidar Harmanani   Signature :

Member   :   Dr. Ramzi Haraty   Signature :

Date   :   Feb. 1, 2013

THESIS COPYRIGHT RELEASE FORM
LEBANESE AMERICAN UNIVERSITY NON-EXCLUSIVE DISTRIBUTION LICENSE

By signing and submitting this license, you (the author(s) or copyright owner) grants to Lebanese American University (LAU) the non-exclusive right to reproduce, translate (as defined below), and/or distribute your submission (including the abstract) worldwide in print and electronic format and in any medium, including but not limited to audio or video. You agree that LAU may, without changing the content, translate the submission to any medium or format for the purpose of preservation. You also agree that LAU may keep more than one copy of this submission for purposes of security, backup and preservation. You represent that the submission is your original work, and that you have the right to grant the rights contained in this license. You also represent that your submission does not, to the best of your knowledge, infringe upon anyone's copyright. If the submission contains material for which you do not hold copyright, you represent that you have obtained the unrestricted permission of the copyright owner to grant LAU the rights required by this license, and that such third-party owned material is clearly identified and acknowledged within the text or content of the submission. IF THE SUBMISSION IS BASED UPON WORK THAT HAS BEEN SPONSORED OR SUPPORTED BY AN AGENCY OR ORGANIZATION OTHER THAN LAU, YOU REPRESENT THAT YOU HAVE FULFILLED ANY RIGHT OF REVIEW OR OTHER OBLIGATIONS REQUIRED BY SUCH CONTRACT OR AGREEMENT. LAU will clearly identify your name(s) as the author(s) or owner(s) of the submission, and will not make any alteration, other than as allowed by this license, to your submission.

Name: Rashad Kabbara

Signature: Date: Feb 1 - 2013

PLAGIARISM POLICY COMPLIANCE STATEMENT

I certify that I have read and understood LAU's Plagiarism Policy. I understand that failure to comply with this Policy can lead to academic and disciplinary actions against me.
This work is substantially my own, and to the extent that any part of this work is not my own I have indicated that by acknowledging its sources.

Name: Rashad Kabbara

Signature: Date: Feb 1 - 2013

ACKNOWLEDGMENTS

A Parallel Search Tree Algorithm for Vertex Cover on Graphical Processing Units

Rashad Karim Kabbara

Abstract

Graphical Processing Units (GPUs) have become popular recently due their highly parallel shared-memory architectures. The computational challenge posed by NP-Hard problems makes them potential targets to GPU-based computations, especially when solved by exact exponential-time algorithms. Using the classical NP-hard Vertex Cover problem as a case study, we provide a framework for GPU-based solutions by exploiting the highly parallel structure of the GPU to accelerate the expansion of search-states in commonly used recursive backtracking algorithms.
Experimental results show that our method can achieve huge speedups on randomly generated sparse graphs, as well as hard instances from the DIMACS benchmark.

Keywords: Parallel programming, Exact algorithms, Graph problems, Graphical processing units, Vertex cover.

TABLE OF CONTENTS

List of Figures

List of Tables

List of Charts

Chapter One

Introduction

A Graphical Processing Unit, henceforth GPU, is a highly parallel architecture initially produced to handle high performance requirements of graphical tasks such as medical imaging, pixel shading and video encoding. Today GPUs are capable to carry out general-purpose computing (i.e., computations that are typically performed by a CPU) where processing of large blocks of data is done in parallel. Their massively parallel architectures along with the newly created programming interfaces made GPU research more popular. The CUDA programming language (NVIDIA C CUDA Programming Guide), a C-based programming platform developed by Nvidia, enables software developers to access the memory and low level instructions of Nvidia GPUs. Similarly STREAM, is the GPU framework used to program ATI GPUs. On the other hand, there exists a generic overlay that provides a common interface API for heterogeneous platforms called OpenCL.

Graphs are very popular data representations that play a role in many application domains including bioinformatics and computational networks. Most problems on graphs are NP-hard, which makes them popular targets to efficient approximation algorithms. Recently, there has been more interest in solving NP-hard problems exactly, even if the running time grows as an exponential function of the input size. Most exact algorithms for NP-complete graph problems adopt the branch and reduce paradigm: a search tree is traversed via a recursive backtracking algorithm.

Due to their super-polynomial running time, exact graph algorithms seem infeasible to be executed sequentially. Therefore, designing parallel graph algorithms has been studied for many years. Some popular search tree based algorithms include exponential algorithms for

graph coloring, dominating set, subgraph isomorphism and vertex cover. Implementing such algorithms efficiently on GPUs is a challenging task due to their backtracking and irregular data access nature.

In this thesis we mainly consider the vertex cover problem, which is undoubtedly the most studied problem in the area of parameterized and exact computation. We start with an overview of existing sequential and parallel vertex cover algorithms (chapter two), and we also provide background information about the architecture of the GPU (chapter three), and highlight the difference between CPU and GPU threads. Then we present our CPU-GPU based vertex cover algorithm (chapter four). The last part (chapter five) will be devoted to experimental studies and the results obtained by comparing the CPU-GPU algorithm running times with the best-known sequential CPU based algorithm.

Chapter Two

Background and Preliminaries

Throughout this thesis we use common graph theoretic terminologies, such as vertices, edges, degree, neighbor, etc. A subgraph H of a graph $G = (V,E)$ is a graph on a subset of V and a subset of E. H is an induced subgraph of G if every edge of G that connects two vertices of H is also an edge of H. A set of vertices is said to be independent if it induces a subgraph with no edges, or an edgeless subgraph. A vertex cover of G is a subset C of V whose complement (in V) is an independent set. C is a minimal vertex cover if no subset of C is a vertex cover. In this thesis we shall be given a graph G and a parameter K and we seek to find a minimal vertex cover of size at most K.

A parameterized problem is said to be fixed-parameter tractable if it can be solved by an algorithm whose running time is a polynomial function of the input size when the input parameter is fixed, or treated as a constant. For example, finding a vertex cover of size K in a graph G on n vertices can be solved in time $O(2^k n^2)$ via a simple recursive backtracking algorithm that branches on the edges of G as follows: pick an edge uv; add u to the solution and recursively find a solution of size K-1; if such a solution cannot be found, backtrack: add v to the solution, while deleting u and not considering it as a solution element.

2.1 The Vertex Cover Problem

The Vertex Cover problem (VC) is one of the most studied NP-Complete graph problems that are widely used in networking, bioinformatics, electrical engineering and many other areas of study. A vertex cover of a graph G is a set of vertices that cover all the edges in the sense that every edge has at least one endpoint in the cover. The complement of a vertex cover induces an edgeless graph, which makes it an independent set. The vertex cover decision problem

takes a graph G and a parameter k as input and asks whether G has a vertex cover of size k or less. This problem is among the classical NP-complete problems mentioned in Richard Karp's historic paper (Karp 85).

Many real world problems can be formulated as instances of the VC problem, including the construction of phylogenetic trees, identity phenotypes and microarray data analysis, to name a few. These various applications have recently caught attention as more and more researchers study the problem to find effective solutions.

Parallel algorithms for the VC problem have been considered recently along with the various parameterized algorithms for the problem (Cheetham, Dehne and Rau-Chaplin), (Hanashiro, Mongelli and Song), (Koufogiannakis and Young), (Zhou, Yang and Li), (Downey and Fellows). Most of the existing parallel algorithms for vertex cover problem use distributed systems, cluster computers, or DNA-based supercomputing as parallel architectures. In our research, we use CUDA to develop a GPU based parallel VC algorithm that runs on NVidia graphical processors.

We adopt a variant of the most used recursive backtracking VC algorithm. Briefly, this common method consists of branching on vertices of maximum degree: select a vertex v of maximum degree; place v in the cover and try to find a solution of size k-1; if such solution cannot be found, backtrack and place all its neighbors are in the cover.

On the other hand, effective preprocessing techniques, known as kernelization techniques in the realm of Parameterized Complexity Theory, are now very well studied as discussed in (Abu-Khzam, Collins, Fellows, et al.). We apply some of these kernelization methods in our CPU/GPU algorithm. Kernelization is a preprocessing technique used to reduce the size of input into a (manageable) function of the parameter k by applying a set of reduction rules. When Vertex Cover Kernelization is applied to a graph (G, k) it results in a reduced instance

4

(G' , k') where k'≤ k. The following three rules have been applied in our parallel algorithm. Starting with a graph of arbitrary number of vertices, applying these three reduction rules exhaustively results in a graph on less than k^2 vertices.

Reduction Rules:

Rule1: any vertex with degree 0 is excluded from the vertex cover since adding this vertex to the cover has no benefit.

Rule2: a vertex whose degree is greater than K should be in the cover. Otherwise all its neighbors are in the cover which is not possible.

Rule3: a vertex with degree 1 should be excluded from the cover while its only neighbor should be added to the cover.

2.2 Graphical Processing Units and Graph Algorithms

Irregular data access algorithms are known to perform poorly on the GPU. Instead, GPUs require coalescing memory access to optimize performance; this is not the case for most graph algorithms represented by either an adjacency list or an adjacency matrix. Nevertheless, graph problems whose sequential algorithms run in polynomial time were implemented on the GPU. Implementations of Breadth first search, single-pair shortest path and all pairs shortest path were studied in (Harish and Narayanan), (Micikevicius), (Joshi and Narayanan). GPU performance speedup was achieved when the size of the graph is huge, in millions of vertices. Some of these algorithms were not only implemented in CUDA on Nvidia GPUs but also using OpenCL which enables the application to execute on other GPU types.

Researchers have also tackled the problem of irregular data access of graph problems and introduced a virtual programming model that can execute at maximal warp along with

exploration of modifying PRAM algorithms to ensure good GPU performance (Hong, Kim and Oguntebi, Olukotun), (Dehne and Yogaratnam).

The work in (Jenkins, Owens and Samatova) is the most related to our work in which an algorithm to solve a hard graph problem was accelerated using the GPU. A GPU based backtracking algorithm for the maximum clique problem was implemented. A clique in a graph is a set of vertices that form a complete subgraph. In their work they have proposed a tree-level based parallelization on GPU, in which multiple subtrees are explored in parallel by GPU processors; this resulted in an order of magnitude increase in performance.

In our work, using the vertex cover problem as a case study; the host (CPU) is used as a buffer scheduler maintaining the search tree, while intensive computation and search tree nodes are constructed by the GPU. The CPU will send tree nodes as tasks to the GPU, every multiprocessor of the GPU will receive its own task and create child nodes in parallel by extensively using the shared and global memory of the GPU. When the global memory is nearly occupied, the data will be transferred back to the main memory and maintained by the CPU for further computations.

Chapter Three

GPU Architecture

The Graphical Processing Unit (GPU) is a highly parallel single chip processor that can reside on a VGA card, on the motherboard, or as an external machine connected to a PC through a communication link. Graphical processing units were mainly used in game consoles and VGA to speed up graphics processing. Several frameworks were developed in order to make it easy to the programmer to control the GPU hardware. GPUs have been modernized from performing pipeline operations to computing floating point procedures. All of the figures bellow are taken from the official CUDA documentation by NVIDIA (NVIDIA C CUDA Programming Guide).
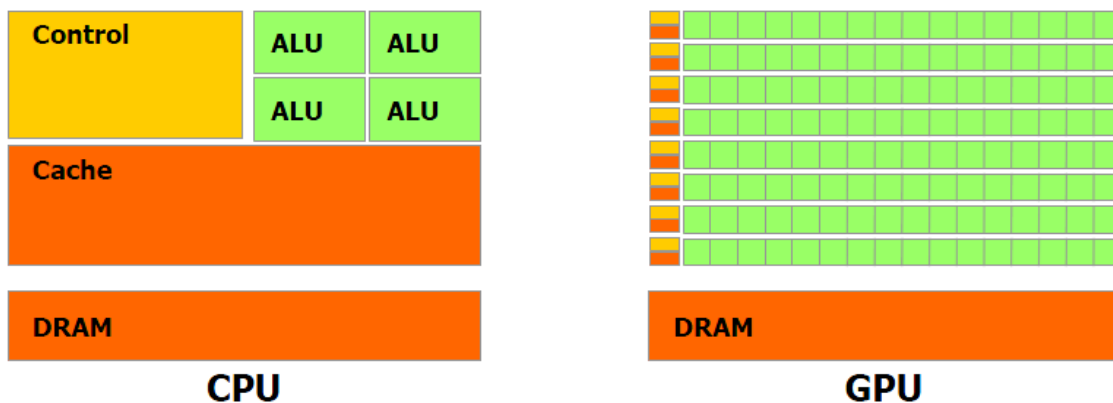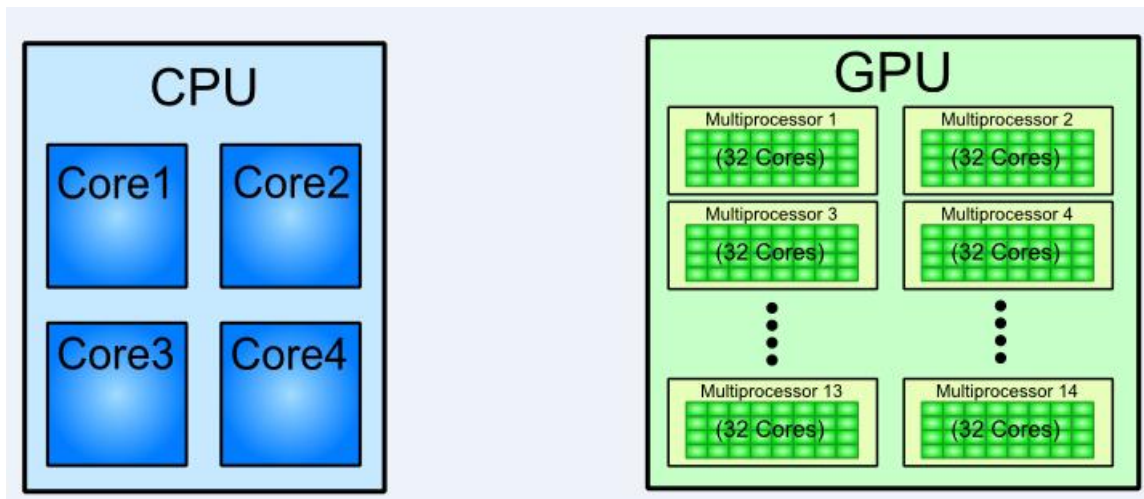
3.1 CPU VS GPU

Figure 1 CPU vs GPU Architecture (NVIDIA C CUDA Programming Guide)

The main difference between the GPU and the CPU is the high number of transistors allocated for GPU. On the other side, the CPU has more controllers for cache and flow control.

Figure 2 CPU vs GPU Cores

A GPU has several multiprocessors each containing the same number of physical processors. In other words, the processing units of the GPU are grouped evenly into multiprocessors. In the graph above, the GPU has 448 processors grouped into 14 multiprocessors. This grouping gives the developers a two layer parallelism, one at the level of the multiprocessor and the other at the level of the processors inside every multiprocessor. The GPU is best used to parallelize floating point operations, while the CPU is best in doing sequential operations that has lots of branches and random access to memory. The CPU is different from the GPU on the following roles:

1. **The goal of the design**: The CPU is used to launch operating systems or virtual machines, manage file systems, and control network communications. while the GPU is mainly used for image processing and pixel shading.

2. **The usage of Transistor:** branch prediction hardware, instruction reorder, large on-die cache, and reservation stations are hardware features that are spent by transistors. The aim of such feature design is to increase the speed of execution of each thread.

8

Transistors are spent by the GPU in multithreading hardware, multiple memory controllers, processor arrays, and shared memory. These features help in permitting the chip to support thousands of threads at the same time, assisting thread communication and maintaining high memory bandwidth.

3. **The Role of the Cache**: The cache is used to advance the performance through decreasing the latency of memory access. In addition, the shared memory is used by the GPU in order to increase the bandwidth.

4. **The Management of the Latency:** Memory latency is handled by the CPU through the use of branch prediction hardware and large caches. Doing so result in taking up huge amounts of die-space and increasing the power consumption. While the latency is handled by the GPU through supporting huge numbers of thread simultaneously. The advantage is that there will be no delay time because the GPU can switch to a free thread once one thread is waiting for memory load.

5. **Multithreading**: Multithreading is supported by the CPU through executing one or two threads per one core. Every multiprocessor of a CUDA capable GPUs support up to 1,024 threads The CPU thread cost hundreds of cycles while GPUs switching is cost free.

6. **SIMD vs. MIMD:** The CPU executes instructions in an MIMD (Multiple Instructions, Multiple Data) fashion while GPU has an array of processor that executes the same Instruction on Different/Multiple data (SIMD).

7. **Memory Controller**: There is no on-die memory controller on Intel CPUs. CUDA based GPUs use up to eight on-die controllers which result in increasing the GPU memory bandwidth.
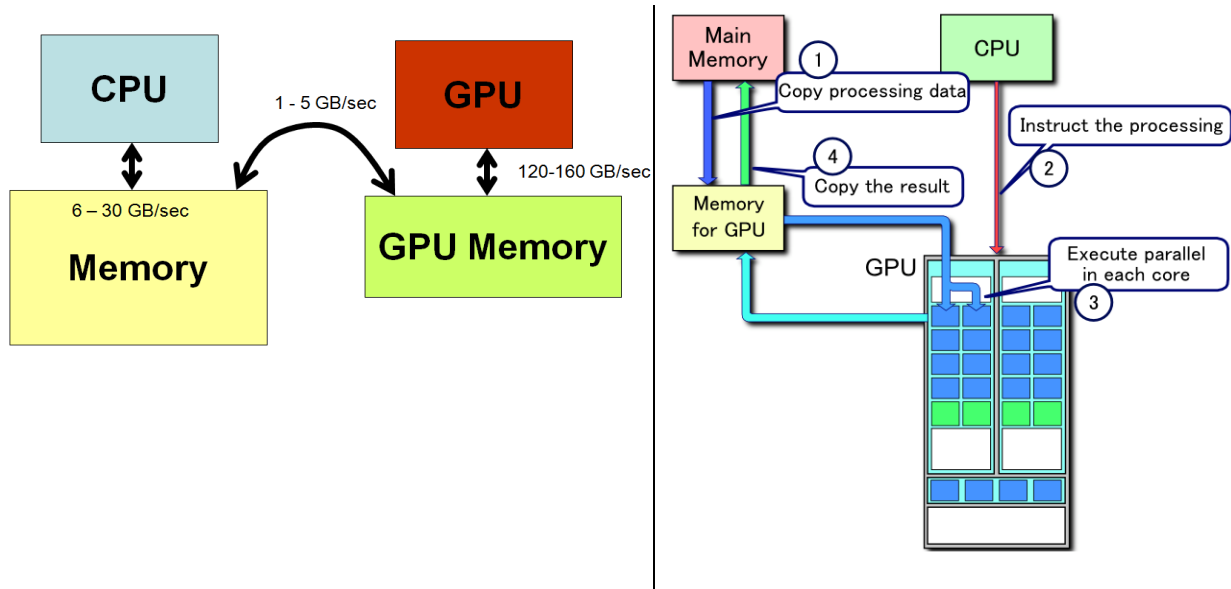
9

3.2 GPU Processing Flow

Figure 3 GPU Processing Flow (Wikipedia GPU)

The GPU can be seen as an external device, independent of the CPU. The first step is to profile the existing application and determine which code segments can be efficiently executed by the GPU, these segments are called kernels. The GPU code is written in CUDA and compiled using nvcc (Nvidia C Compiler). The GPU receives an instruction from the CPU to execute a certain kernel, and instructs its processing units to execute the kernel in parallel. A GPU cannot directly access the main memory; hence input data should be transferred to GPU memory before executing the kernel. After the kernel execution completes, the resulting output is transferred back to the main memory to be available for the CPU. The data transfer back and forth to the GPU slows down the algorithm; hence compute intensive tasks not bound to memory gains the most performance when accelerated by a GPU.

3.3 GPU Threads
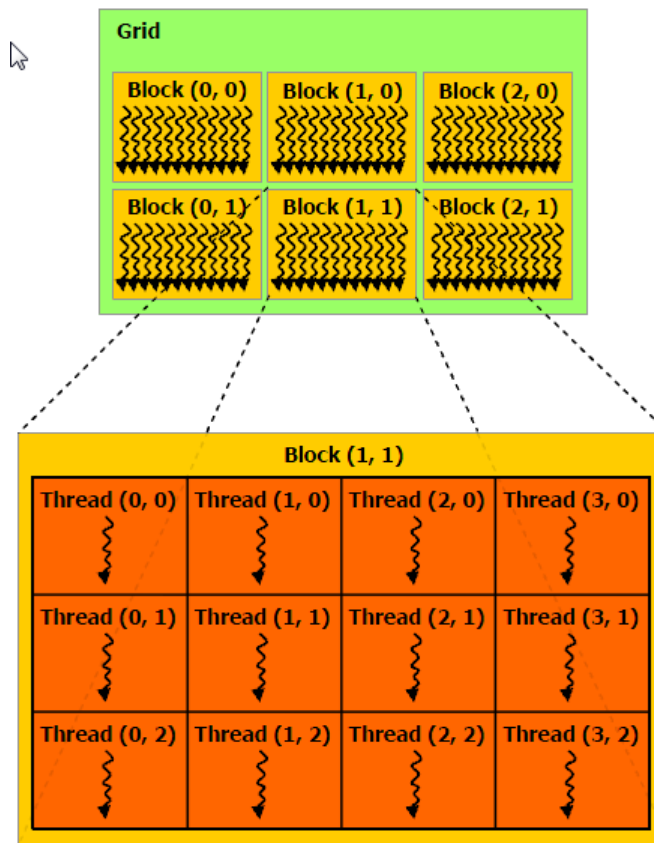
Figure 4 GPU Threads (NVIDIA C CUDA Programming Guide)

A kernel is a C function that is executed by the GPU. The number of threads that should

execute this kernel is specified by the developer. The threads are grouped into blocks. Since

the GPU is a multiprocessor device, where each multiprocessor has the same number of

physical processors, the developer should also specify the number of blocks to be executed.

Every block is executed by a single multiprocessor. The blocks are distributed to the

multiprocessor (SM) depending on the availability of the SM and the ability of the SM to

execute multiple blocks at the same time. The threads of a block launch the kernel in a single

instruction multiple thread (SIMT) mode on a single multiprocessor. When threads of a block

finish execution, the SM is freed and a new block is taken by that SM. Physically each SM is

capable of executing 32 instructions by 32 threads concurrently (used to be 16 on old

devices). This set of instructions is called a warp (old GPUs execute half a warp). Every

thread in a single warp executes the same instruction at the same time. In case there is a

branch scenario, a thread must wait for all other threads in the warp to reach the same code

segment in order to continue its execution.
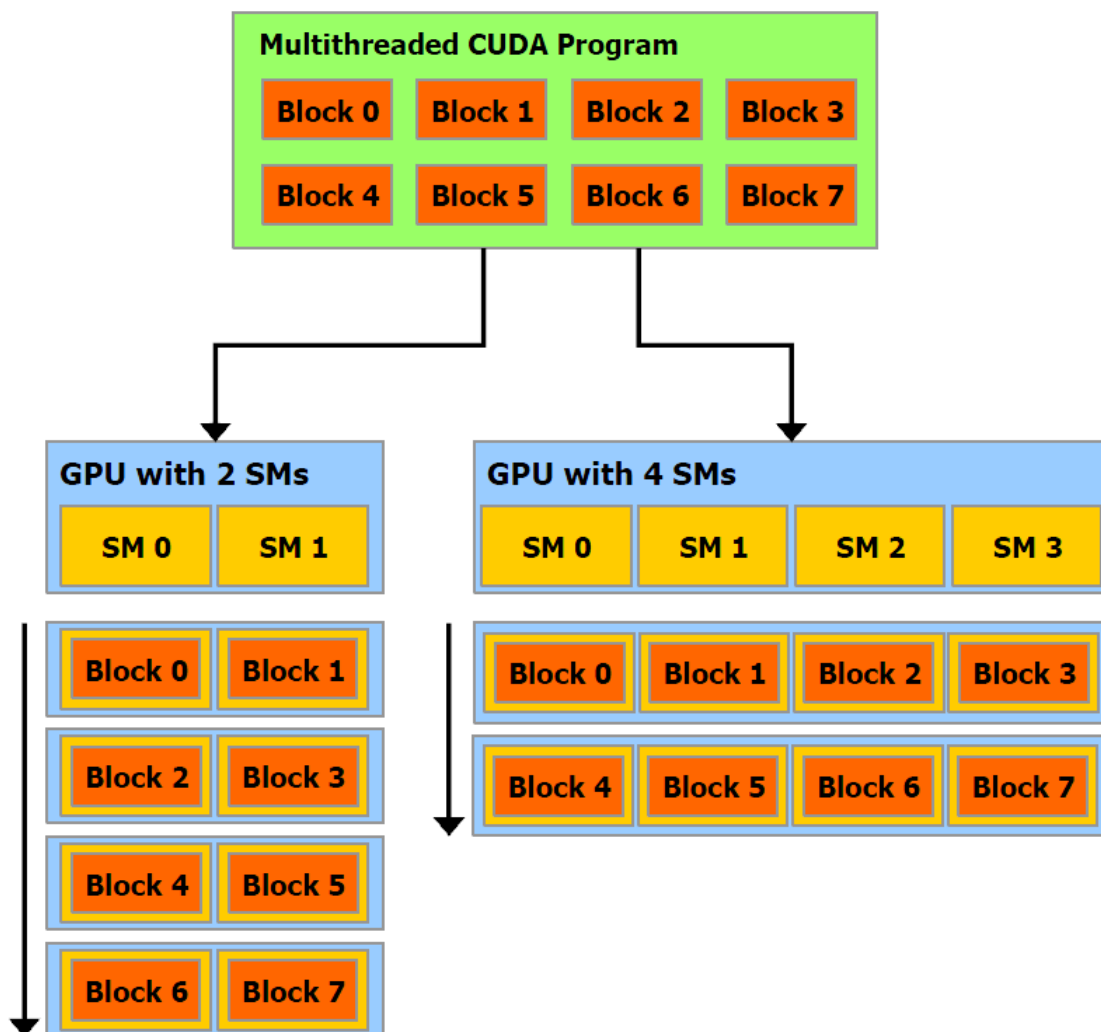
Figure 5 GPU Multiprocessors (NVIDIA C CUDA Programming Guide)

To illustrate further on how blocks are distributed to multiprocessors SMs consider the figure

above. The figure shows how blocks are distributed to multiprocessors in case we have a GPU

12

with two SMs and in case we have a GPU with four SMs. Clearly the more SMs the GPU have the more parallelism we obtain.

3.4 GPU Memory Model

There are five memory regions on the GPU. We discuss the usage, advantages and disadvantages of each.

Figure 6 GPU Memory Model (NVIDIA C CUDA Programming Guide)

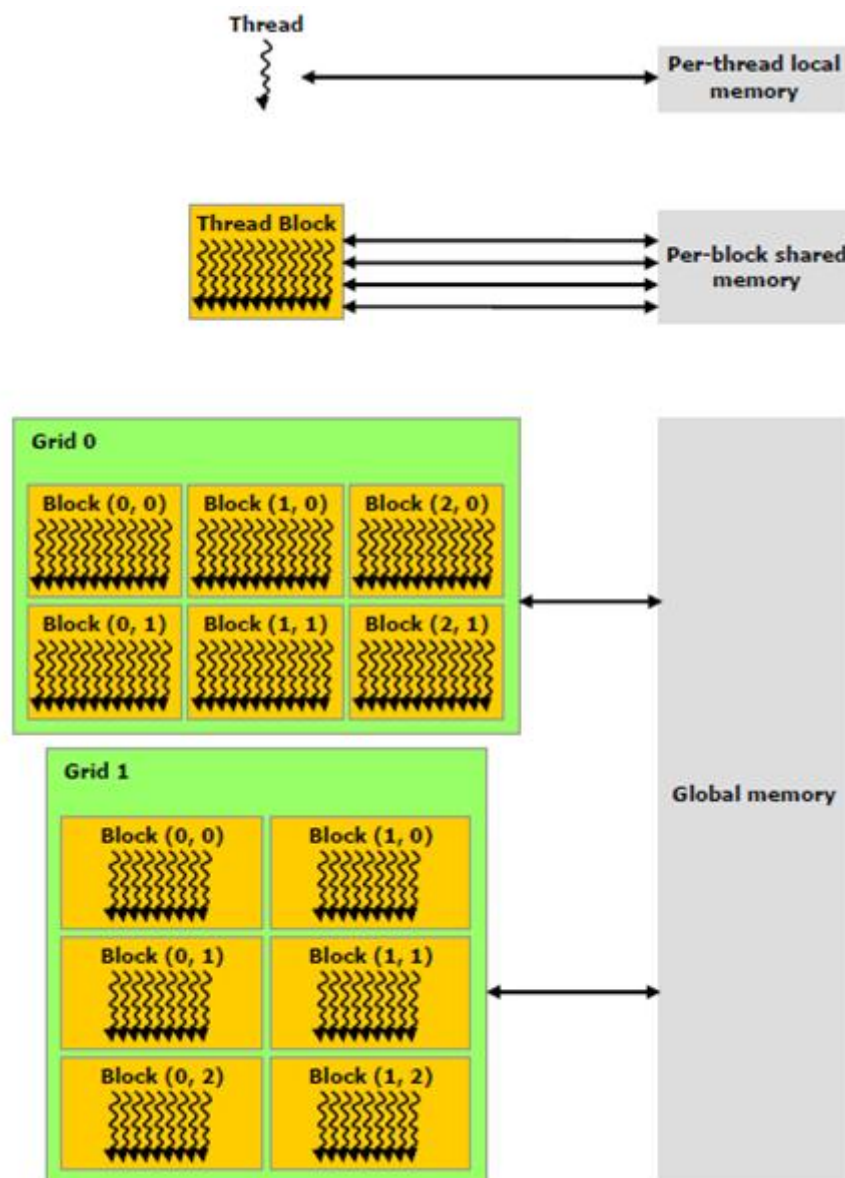Global Memory: Also called Device memory. It is the largest memory (1-4GB) region that all physical threads have access to.

Shared Memory: There is a shared memory in every GPU multiprocessor. This memory can only be access by the threads in the same multiprocessor. Access to shared memory is faster than global memory but the size of the shared memory is much smaller than the global memory (in KB).

Registers: a very small memory used to store variables for single threads. Every physical processor has its own register memory.

Constant and Texture memory: These memory regions are the same as global memory in which all threads in all multiprocessors can access. But they are read-only memory that differs in caching algorithms.
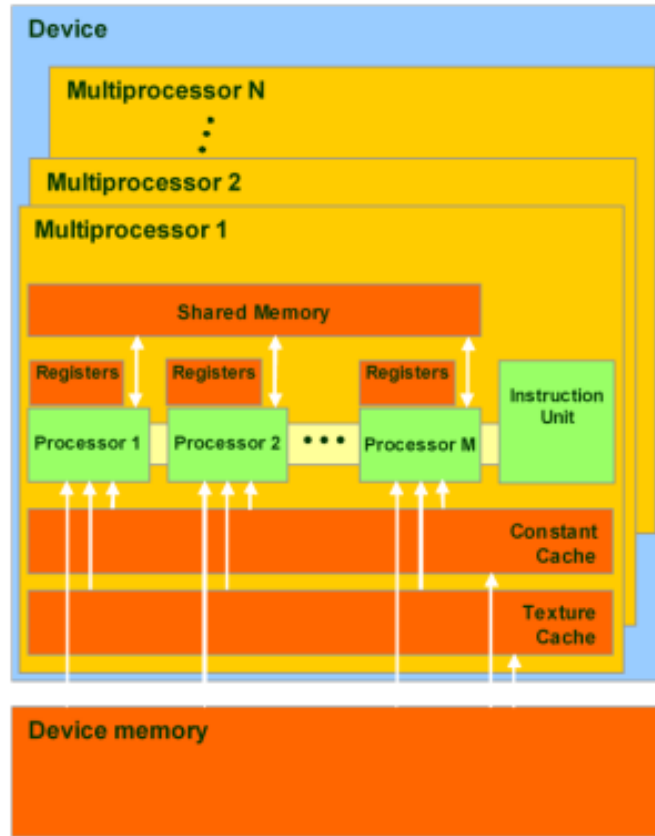
Figure 7 GPU Architecture (NVIDIA C CUDA Programming Guide)

The following can be summarized from the above GPU Memory model figure:

- The Device (aka the GPU) has several Multiprocessors

- Each multiprocessor has a number of physical processors.

- Every physical processor has its own Register memory

- An SM contains a single shared memory that can be accessed by all its processors

- Multiprocessors can access a single large memory space which is called the Device Memory.

Chapter Four

A GPU-CPU Based Algorithm for Vertex Cover

We now describe our approach in details. As the title suggests, we use the CPU as a task manager as well as master process that can also assist in the computation when necessary. Initially, the input graph is stored in the main memory as an adjacency list.

4.1 The Data Structure

The input graph G is represented using the adjacency list format. Since G is a simple unweighted graph, we can use array instead of linked lists to represent the list of neighbors of any vertex. Instead of the two-dimensional structure of the adjacency list, we use a one-dimensional array adjacency list (AL) in CUDA memory. The reason why a single dimension array is used is because it allows easy and faster data transfer between the CPU and GPU. Considering the graph bellow, the single dimension AL is depicted bellow:

AL	1	3	0	2	5	1	4	5	6	0	4	2	3	1	2	2
PL	0	0	1	1	1	2	2	2	2	3	3	4	4	5	5	6
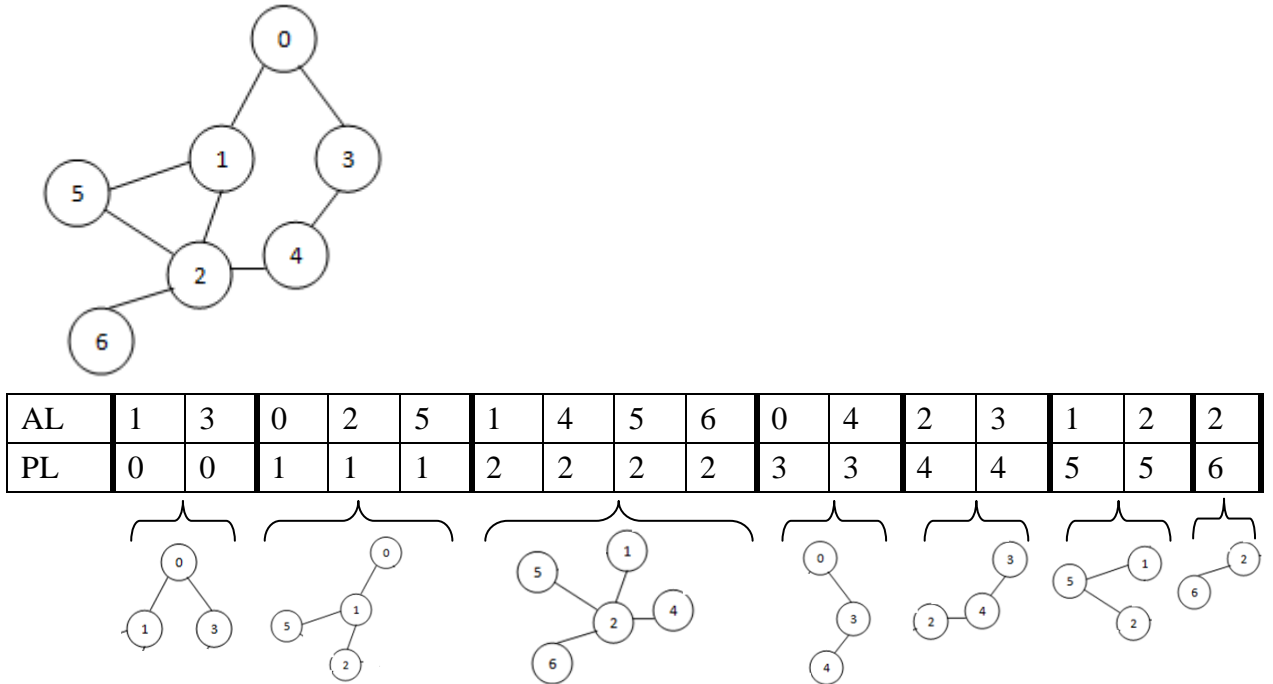
The parent list (PL) array shown in the table is used by the threads to get the source of the vertices they are working on (this will be made clearer later on). Both the AL and the PL arrays are constants (their values are never changed through the program execution). They are kept in the GPU's global memory and only freed when the program exits.

In addition to the global AL and PL lists, every instance of the problem can be uniquely determined by a degrees' array, which simply contains the degrees of all the vertices and whether a vertex is already deleted and added to the vertex cover. The degree of such vertex will be set to -1, while a deleted vertex that is not added to the cover will have degree 0.

4.2 Implementation Details

The algorithm is executed by the GPU after specifying the number of blocks and the number of threads each block should have (the developer specifies the number of blocks and threads). Each block is executed by a single GPU multiprocessor. Each thread within the block is executed by a single processor of the multiprocessor.

Every GPU multiprocessor, when idle, receives a degree array, selects the highest degree vertex (MaxNode) and generates two degree arrays for further processing.

1. The first one contains the result when removing MaxNode from the graph;

2. The second is the resulting degree array when the neighbors of MaxNode are removed.

Block execution could be summarized in the following
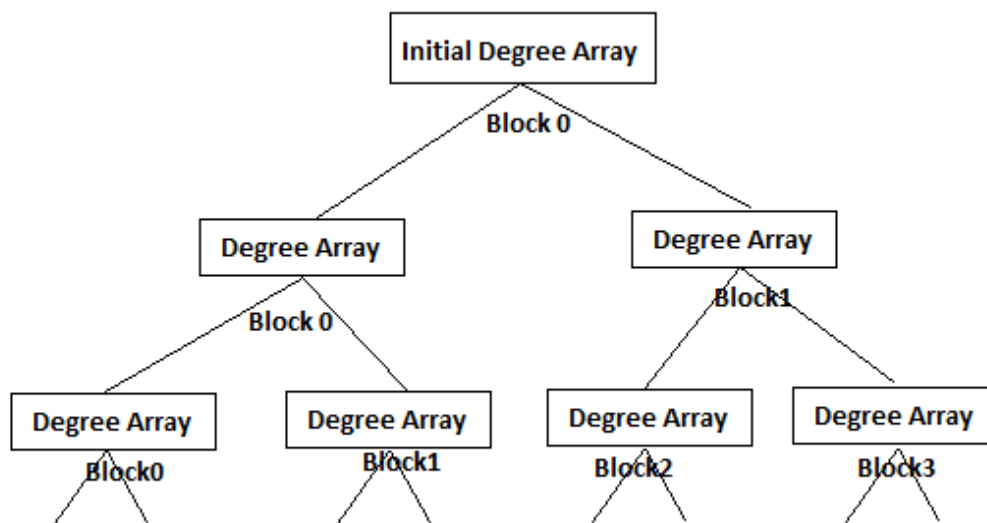
Parallel Tree:

Search:

The algorithm starts with a single degree array. An idle GPU multiprocessor receives the
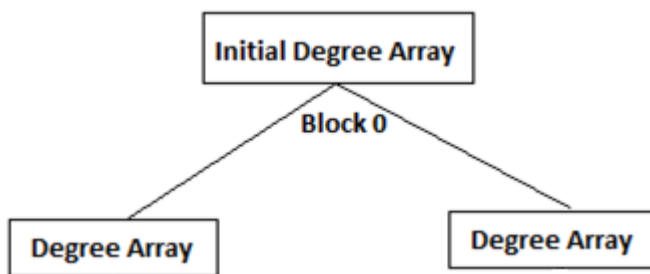
degree array and generates two degree arrays that are then processed by two multiprocessors.

The two multiprocessors will generate two arrays each hence in the third execution we have

four degree arrays to be executed in parallel by four GPU multiprocessors. When the number

of blocks becomes larger than the number of physical multiprocessors, block execution will

be serialized. When a GPU finds a solution it sends a signal to the CPU in order to inform all

GPU multiprocessors to stop execution. If the GPU global memory is full and no solution has

been found yet, data from the GPU's global memory will be transferred to the RAM and maintained by the CPU for future processing.

4.3 Inside the Block

Every block contains a number of threads that run in parallel (maximum is 1024). The job of these threads is to select the vertex with maximum number of neighbors from the degree array and to generate two other degree arrays accordingly.

The degree array of the above graph | 2 | 3 | 4 | 2 | 2 | 2 | 1 | is:

A single multiprocessor of the GPU receives the degree array and generates two new degree arrays for further processing as follows:

Step 1 determining the vertex with maximum number of neighbors.

Performance is improved by using the Prefix-Max algorithm which returns the element with the largest number of neighbors in O(logn) instead of the sequential version which requires O(n).

| Thread 0 | Thread 1 | Thread 2 | Thread 3 |

| 2 | 3 | 4 | 2 | 2 | 2 | 1 |
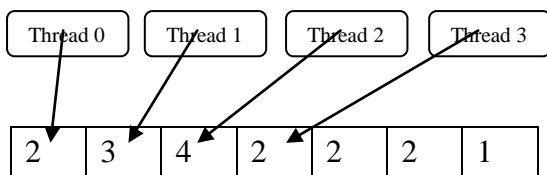
Step 2 in this step two new degree arrays are created by every multiprocessor.

The vertex with maximum number of neighbors (MaxNode) is vertex 2 (having 4 neighbor vertices). An idle multiprocessor receives the degree array as a task and creates 2 new degree arrays. The first (DegreeArray1) contains the result of removing MaxNode from the graph and the second (DegreeArray2) is the resulting degree array when the neighbors of MaxNode are removed.

19

DegreeArray1 starts as the initial degree array sent to an idle multiprocessor:

2	3	4	2	2	2	1

To remove a vertex and place it in the solution, its value in the degree array is set to -1. The degree of each of its neighbors is decremented by one. After removing vertex 2 the DegreeArray1 becomes:

2	3	-1	2	2	2	1

After vertex 2 is removed, we use AL and PL arrays to get the vertices that MaxNode is connected to, and decrement their values in the degree array:

TID	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
AL	1	3	0	2	5	1	4	5	6	0	4	2	3	1	2	2
PL	0	0	1	1	1	2	2	2	2	3	3	4	4	5	5	6

Each thread checks the value of PL[threadid]. If the value is equal to MaxNode, then each thread will look at its corresponding AL[threadid]. AL[threadid] is the vertex that is connected to MaxNode. Therefore:

Thread5 knows that vertex 1 is connected to MaxNode

Thread6 knows that vertex 4 is connected to MaxNode

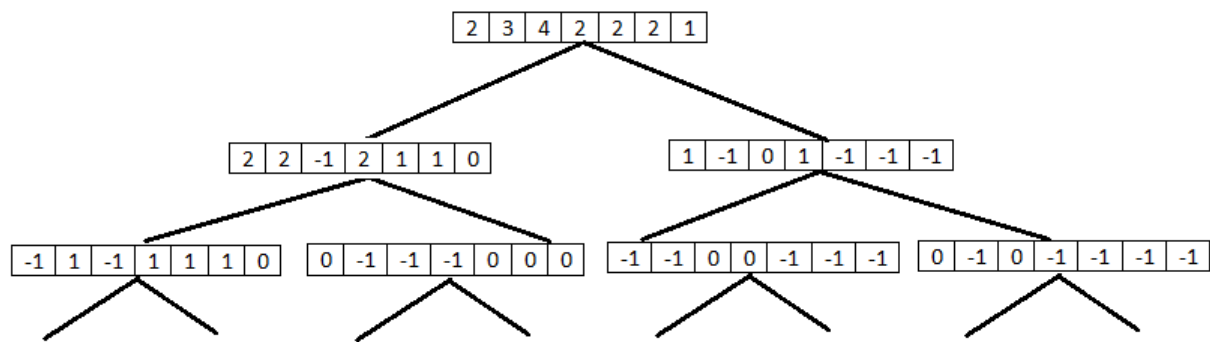*Thread7 knows that vertex 5 is connected to MaxNode*

Thread8 knows that vertex 6 is connected to MaxNode

The above threads can now decrement vertices (1,4,5,6) in DegreeArray1. Hence DegreeArray1 becomes:

2	2	-1	2	1	1	0

A similar procedure takes place to generate **DegreeArray2.** The resulting search space is shown:

2 3 4 2 2 2 1

2 2 -1 2 1 1 0 | 1 -1 0 1 -1 -1 -1

-1 1 -1 1 1 1 0 | 0 -1 -1 -1 0 0 0 | -1 -1 0 0 -1 -1 -1 | 0 -1 0 -1 -1 -1 -1

4.4 Pseudo Code

The described algorithm can be summarized in the following CPU and GPU pseudo code.

CPU Pseudo Code:

```
Read Graph From File
Create Data structure and send it to GPU
WHILE NoSolution is False   //loop untill no solution exists
{
    SolutionFound = GPUExpandSearchTree(K); //Kernel Method.
        IF SolutionFound is True
            Define array VC = CopyVCFromGPU(); //copy solution to CPU
            print arrayVC;
            Exit WHILE loop;
        IF GPU Memory is full
            CPUScheduller = CopySearchTreeFromGPU(); //Copy Data to CPU
            Free GPU Memory;
    GPUHasTask = GetGPUStatus(); //if GPU Search Tree can be further expanded
        IF GPUHasTask is FALSE
            IF CPUScheduller is Empty
                Print No Solution Found
                SET NoSolution to True;
            ELSE
                SendTaskToGPU(CPUScheduller); //Send SearchTree nodes to GPU
}
Free Memory on CPU and GPU;
```

GPU Pseudo Code:

```
//The following code is executed in parallel by all GPU Multiprocessors
Copy Degree Array to Shared Memory //Depends on BlockID
Apply Graph Reduction Rules
IF K > 0
{
    SET MAXNODE as result of PrefixMax

    SET array DegreeArray1 = Initial DegreeArray
    IF ThreadID is equal to MAXNODE
        SET DegreeArray1[ThreadID] to -1 //remove MAXNODE
        Decrement K by -1
    IF Neighbor[ThreadID] = MAXNODE //edge connects MAXNODE to ThreadID
        Decrement DegreeArray1[ThreadID] by -1 //remove neighbor links

    SET array DegreeArray2 = Initial DegreeArray
    IF Neighbor[ThreadID] = MAXNODE //edge connects MAXNODE to ThreadID
        SET DegreeArray2[ThreadID] to -1 //remove neighbors of MAXNODE
        Apply Mutex lock on K and decrement K by -1
    IF ThreadID is a Neighbor of the removed Nodes
        Decrement DegreeArray2[ThreadID] by -1 //remove neighbor links

    IF DegreeArray1 or DegreeArray1 is VC a solution
        Send Signal to CPU //to inform CPU that a solution is found
}
Free Shared Memory
```

Chapter Five

Experimental Results

We have divided the experiments in two parts. The first experiment is conducted to test the GPU code on randomly generated graphs. This allows us to compare the effectiveness of the algorithm on graphs of different sizes and densities. The second experiment is conducted using DIMACS-vertex cover benchmarks (cs.hbg.psu.edu/benchmarks/vertex_cover.html).

5.1 Hardware

Throughout the experiments, the following hardware was used:

- Processor: Intel® Core™i7 CPU @ 3.40 GHZ

- System type: 64-bit Operating System (windows 7)

- Installed Memory (RAM): 8GB

- Nvidia GeForce GTX 560. 7 Multiprocessors with 48 physical processors in each

5.2 Random Graphs

Random graphs were created using a graph generator. The generator takes 3 variables:

1. N as the number of vertices in the graph

2. MaxN as the maximum number of neighbors any vertex can have

3. MinN as the minimum number of neighbors a vertex can have

The graph is created such that every vertex has a number of neighbors which is less than MaxN and greater than MinN. The resulted graph consists of N vertices and an average of

MaxN and MinN of neighbors for each vertex. If MaxN and MinN are set to the same number, then every vertex in the graph will have the exact same number of neighbors.

We have compared our CPU/GPU based algorithm to the CPU based algorithm specified in (Abu-Khzam, Collins, Fellows, et al.). The purpose of these tests is to compare the effectiveness of the algorithm on graphs of different sizes and densities. The table lists the graph specifications, the VC size discovered and the timings of each algorithm.

Table 1 Random Graph Experiments

Number of Vertices	Total number of edges	VC size	CPU based timing	GPU timing
100	714	42	0.0001	1.2 sec
100	1282	65	0.001	1.3 sec
100	4143	75	0.01	2.2 sec
500	790	182	534 sec	16 sec
500	1644	191	698 sec	20 sec
500	4683	220	844 sec	48 sec
500	8448	228	922 sec	105.8 sec
500	16966	261	985 sec	198.3 sec
500	53341	322	1218 sec	355.4 sec
1000	16915	658	>2 days	400 sec
1000	68966	747	>2 days	50 min
1000	122524	879	14.4 hrs	2 hrs

The experiments were executed on several graphs having different number of vertices and of different density. On small graphs (50-200 vertices) experiments show that the GPU algorithm was in most cases slower than that of the CPU. On some small graphs the CPU required less than a second to find a solution while the GPU found a solution within seconds.

When the number of vertices increases, better performance results are obtained. When testing on graphs with size greater than 400 vertices, the GPU out-performed the CPU by reaching 1000x speedups. This kind of speedup is comparable to speedups obtained on simpler non-recursive algorithms in several papers (Gregg and Hazelwood).

5.3 Reasons behind the Huge Speedup

The following two charts represent the CPU vs GPU timing illustrated form the table above when the number of vertices in the graph is fixed (500) while the number of edges increases.
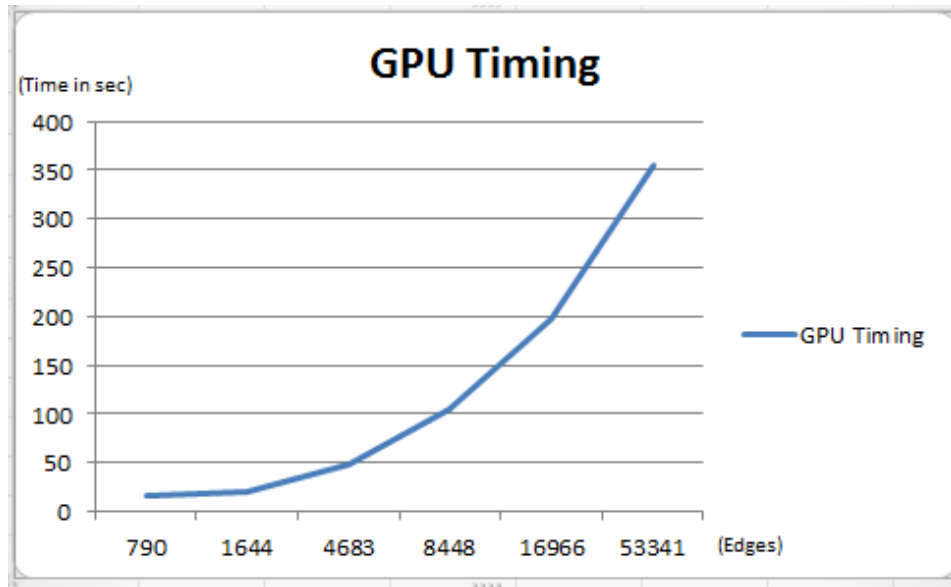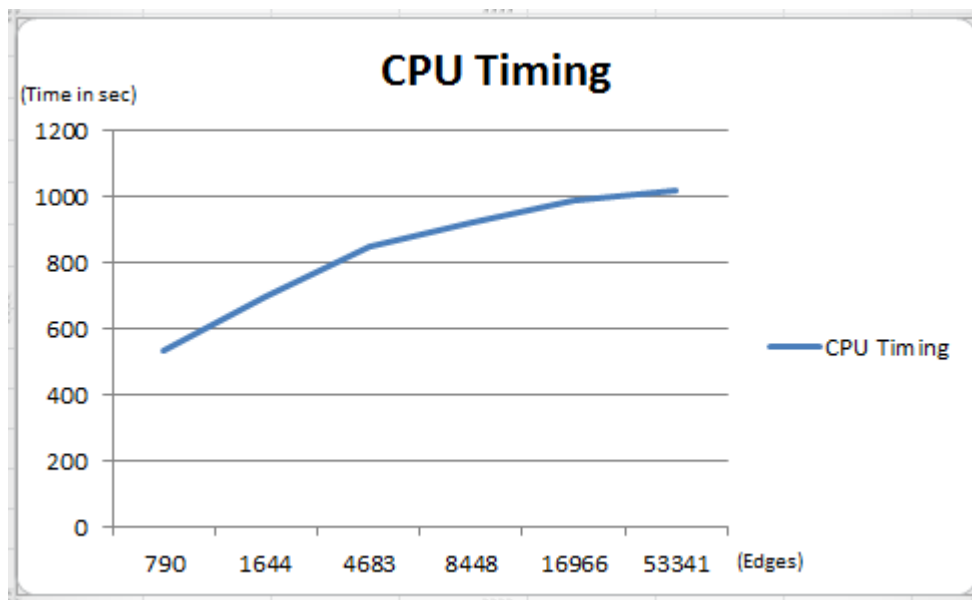

Chart 1 GPU Timing

Chart 2 CPU Timing

The charts show that the slope of the GPU timing line increases more than that of the CPU as the number of edges increase. This means that as the graph becomes denser, the speed up

gained by the GPU is being less effective. When the graph is denser, its corresponding search tree space becomes larger. In this case the GPU memory will be filled up faster; hence more transfer to main memory (RAM) is required. The increase in memory transfer from GPU back and forth to the main memory will slow down the GPU algorithm hence leading to less speed up gains.

Moreover, when the number of edges is 790, the GPU code outperforms the CPU by ~33x. But when the number of edges is larger (53341), the GPU code outperforms the CPU by ~3x. The following points summarize the random graph experiments:

1. The CPU code is faster than the GPU code for very small graphs. This is because the additional time spent to initialize the GPU, set data structures and transfer data back and forth to the GPU results in slower execution time compared to the CPU.

2. As the number of edges increase, the GPU code becomes less efficient. This is because the denser the graph, the bigger the search space, hence more memory transfer to the main memory (RAM) is required.

Therefore we conclude that best performance gains by the GPU are achieved when the input graph is a large sparse graph.

5.4 Experiments – DIMACS Graphs

DIMACS benchmarks, taken from the Second DIMACS Implementation Challenge (1992-1993), are real world example graphs used to measure and compare the effectiveness of various algorithms. The following table shows the speedup obtained when comparing our GPU code to the (purely) CPU code using some of the hardest VC instances from the DIMACS benchmark:

Table 2 Dimacs Graphs Experiments

Dimacs Graph	Number Of Vertices	VC found	CPU code	GPU code
frb30-15-1	450	423	>7days	7.25 min
frb30-15-2	450	423	>7days	12.6 min
frb35-17-1	595	564	>7days	8.4 min
frb35-17-2	595	564	>7days	23.16 min
frb40-19-1	760	725	>7days	20.1 min
frb40-19-2	760	725	>7days	13.01 min
frb45-21-1	945	907	>7days	14.9 min
frb45-21-2	945	907	>7days	39.8 min

Chapter Six

Conclusion

Using the classical NP-hard Vertex Cover problem as a case study, we provided a framework for GPU-based solutions for parallel recursive backtracking graph algorithms by exploiting the highly parallel structure of the GPU to accelerate the expansion of search-states. The CPU, which is best used for performing sequential operations that includes branching and random memory accesses, plays the role of a buffer scheduler maintaining the search tree. The GPU is best used for coalescence of floating point operations. It was used to speed up the fast constructions of search states. Moreover, the shared memory of the GPU was extensively used to speed up memory intensive operations. Using the DIMACS benchmark to conduct experiments, our method exhibited notable speedups on graphs whose corresponding search space can fit into the GPU's global memory. This makes our approach suitable for large sparse graphs that are often found in many real scientific applications of Vertex Cover. A natural extension of this work would mainly focus on using multiple GPUs or a cluster of GPUs to accelerate similar recursive backtracking algorithms. Finally, it would be interesting to develop a parallel search tree algorithm that employs a dynamic load balancing strategy on multiple GPUs.

Bibliography

Abu-Khzam, Faisal, Rebecca Collins, Michael Fellows, Michael Langston, Henry Suters, and Christopher Symons. "Kernelization Algorithms for the Vertex Cover Problem: Theory and Experiments." *Proceedings of the Sixth Workshop on Algorithm Engineering and Experiments and the First Workshop on Analytic Algorithmics and Combinatorics, 10 Jan. 2004, New Orleans, LA, USA*. Ed. Lars Arge, Giuseppe F. Italiano, and Robert Sedgewick. LA: SIAM, 10 Jan. 2004. 62-69. Print.

Cheetham, James, Andrew Rau-Chaplin, Ulrike Stege, and Peter Taillon. "Solving Large FPT Problems on Coarse-Grained Parallel Machines." *Journal of Computer and System Sciences - Special issue on Parameterized Computation and Complexity* 67.4 (2003): 691-706. Print.

Dehne, Frank, and KumananYogaratnam. "Exploring the Limits of GPUs With Parallel Graph Algorithms." *CoRR* 1002.4482 (2010): 212-220. Web.

Dinneen, Michael, Masoud Khosravani and Andrew Probert. "Using Opencl for Implementing Simple Parallel Graph Algorithms." *PDPTA* 731.324 (2011): 268-273. Web.

Downey, Rodney and Mickeal Fellows. "The Parametrized Complexity of Some Fundamental Problems in Coding Theory." *SIAM* 29.2 (1999): 545-570. Web.

Gregg, Chris and Kim Hazelwood. "Where Is the Data? Why You Cannot Debate CPU vs. GPU Performance Without the Answer." *ISPASS* 10.11 (2011): 134-144. Web.

Hanashiro, Erik, Henrique Mongelli and Siang Song. "Efficient Implementation of the BSP/CGM Parallel Vertex Cover FPT Algorithm.*" Experimental and Efficient Algorithms, Third International Workshop, WEA, 25-28 May 2004, Angra dos Reis, Brazil*. Ed. Celso C. Ribeiro and Simone L. Martins. Angra dos Reis: Springer, 25 May 2004. 253-268. Print.

Harish , Pawan and P. J Narayanan. "Accelerating Large Graph Algorithms on the GPU Using CUDA." *HiPC* 4873.1 (2007): 197-208. Web.

Hong, Sungpack, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. "Accelerating CUDA Graph Algorithms at Maximum Warp." *PPoPP* 10.1145 (2011): 267-276. Web.

Jenkins, John, John Owens and Nagiza Samatova. "Lessons Learned from Exploring the Backtracking Paradigm on the GPU.*" Euro-Par 2011 Parallel Processing - 17th International Conference, Euro-Par, 29 Aug. – 2 Sept. 2011, Bordeaux, France*. Ed. Emmanuel Jeannot, Raymond Namyst and Jean Roman. Bordeaux: Springer, 29 Aug. 2011. 425-437. Print.

Joshi, Swapnil and P. J Narayanan. "Performance Improvement in Large Graph Algorithms on GPU Using CUDA: An Overview." *IJCA* 10.10 (2010): 10-14. Web.

Karp, Richard. "Reducibility Among Combinatorial Problems*." Proceedings of a Symposium on the Complexity of Computer Computations, 20-22 March 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York*. Ed. Raymond E. Miller and James W. Thatche. New York: Plenum Press, 20 March 1972. 85-103. Print.

Koufogiannakis, Christos and Neal Young. "Distributed and Parallel Algorithms for Weighted Vertex Cover and Other Covering Problems." *PODC* 10.1145 (2009): 171-179. Web.

Micikevicius, Paulius. "General Parallel Computation on Commodity Graphics Hardware: Case Study With the All-Pairs Shortest Paths Problem." *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications,PDPTA , 21-24 June 2004, Las Vegas, Nevada, USA*. Ed. Hamid R. Arabnia. Las Vegas: CSREA Press, 21 June 2004. 1359-1365. Print.

"NVIDIA CUDA C Programming Guide." *docs.nvidia.com*. Nvidia Corporation, 2012. Web. 15 Oct. 2012. <http://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html>.

Zhou, Xu, ZhiBang Yang and Kenli Li. "A New Approach for the Dominating-Set Problem by DNA-Based Supercomputing." *JSW* 5.6 (2010): 662-670. Print.