Exploring Irregular Memory Access Applications on the GPU

Thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science (by Research)
in
Computer Science

by

Mohammed Suhail Rehman
200707016
rehman@research.iiit.ac.in

International Institute of Information Technology
Hyderabad, India
June 2010

INTERNATIONAL INSTITUTE OF INFORMATION TECHNOLOGY
Hyderabad, India

CERTIFICATE

It is certified that the work contained in this thesis, titled "Exploring Irregular Memory Access Applications on the GPU" by Mohammed Suhail Rehman, has been carried out under our supervision and is not submitted elsewhere for a degree.

Date	Advisor: Dr. P. J. Narayanan	Advisor: Dr. Kishore Kothapalli

To my parents,
for supporting my (un)healthy obsession with computers

فَتَعٰلَى اللّٰهُ الْمَلِكُ الْحَقُّ ۗ وَلَا تَعْجَلْ بِالْقُرْءَانِ مِنْ قَبْلِ أَنْ
يُقْضٰىٓ إِلَيْكَ وَحْيُهُۥ ۖ وَقُلْ رَّبِّ زِدْنِى عِلْمًا

"High above all is God, the King, the Truth! Be not in haste with the Qur'an before its revelation to thee is completed, but say, "O my Lord! advance me in knowledge""

- The Holy Qur'an [20:114]

Abstract

General Purpose Computation on Graphics Processors (GPGPU) is a recent development in high performance computing. Graphics hardware has evolved over the past decade or so from fixed-function stream processors to highly programmable computation engines that offer the maximum price-to-performance ratio among current parallel hardware. The GeForce GTX 280 from NVIDIA comes with a peak performance of 1 TFLOP and costs about 400 US$. This peak performance, however is hard to realize for many applications on the GPU. Applications that generally see good performance gains (compared to implementations on similar parallel hardware or CPU) are those which are extremely parallel and have data locality which match well with the memory subsystem of such hardware. A lot of research has been devoted to finding ways to optimally utilize this massively parallel hardware to accelerate a diverse range of applications.

There exists a range of applications which are parallelizable but do not possess 'good' memory access characteristics. These applications are generally classified as *irregular* memory access applications. Developing techniques to implement such applications have been a challenge on such massively parallel hardware. These applications are traditionally not well suited for most modern parallel architectures and even modest gains in performance are appreciated. Development of suitable primitives for this class of applications is crucial in advancing the acceptance of such massively parallel hardware in the high performance computing community.

In our work, we present a few primitives which are fundamental to irregular applications. We start with *pointer jumping* on the GPU, which is a basic technique used in processing lists in parallel. This technique is used to implement the *list ranking* problem on the GPU - which was the original solution for this problem when it was proposed by Wyllie. This technique is also a useful primitive for algorithms which require structural changes to the list during traversal. This implementation, however requires expensive synchronization and is not work-optimal. Some of these techniques we have devised for the implementation of pointer jumping on the GPU are used in the implementation of the Shilovich-Vishkin connected components algorithm on the GPU.

We introduce another technique to solve this problem efficiently on the GPU for large data sets. This technique is a recursive formulation of a recent algorithm for shared memory systems by Helman and JáJá. This recursive formulation allowed us to exploit the massive parallelism on the GPU as we were able to minimize the number of idle threads during computation. The optimal parameters for this algorithm to run on the GPU are also discussed, leading the fastest single-chip implementation of the list ranking problem, at the time of printing. Our optimized algorithm can rank a random list of 8 million elements in about 50 milliseconds which translates to about 15× speedup over a sequential CPU implementation and 5-6× speedup over the Cell Broadband Engine.

We then present an implementation of the list ranking primitive to accelerate a few Euler-tour technique based algorithms - these include tree rooting (parent finding), subtree size calculation, tree traversals and vertex level computation. A major chunk of the actual runtime for these implemenations is the list ranking step, and we obtain highly efficient algorithms for processing trees on the GPU as a result. We see a sustained performance benefit of about 9-10x over sequential BFS on GPU and about 2× over BFS implemented on CUDA.

Optimizing such applications for the GPU is not a trivial task. We explain some of the characteristics of irregular application development for the GPU as well as techniques to model and assess performance for the GPU. Our research has yielded a set of simple equations can aid in quickly estimating the runtime of a particular implementation on the GPU. We discuss list ranking in this context and provide a few pointers in maximizing performance for irregular applications.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A *Graphics Processing Unit* or a GPU is a dedicated processor that offloads graphics rendering from a system's central processing unit. GPUs are primarily used in personal computers and on video game consoles for entertainment and visualization applications. As user interfaces and software applications get more interactive and 3D-like, GPUs are now starting to appear in portable devices, embedded systems and consumer electronics devices.

The reason GPUs are more efficient than CPUs in what they do is because of the fact that they are purpose-built for the demands of real-time graphics rendering and acceleration. Dedicated hardware for handling visual output for computers have been in existence since the 1970's but the power of this hardware has grown exponentially in the last 2 decades with the emergence of graphical user interfaces, multimedia and real-time 3D applications. Today, virtually every personal computer has a GPU - either in the form of a powerful dedicated graphics card or in an integrated chipset on the system motherboard.

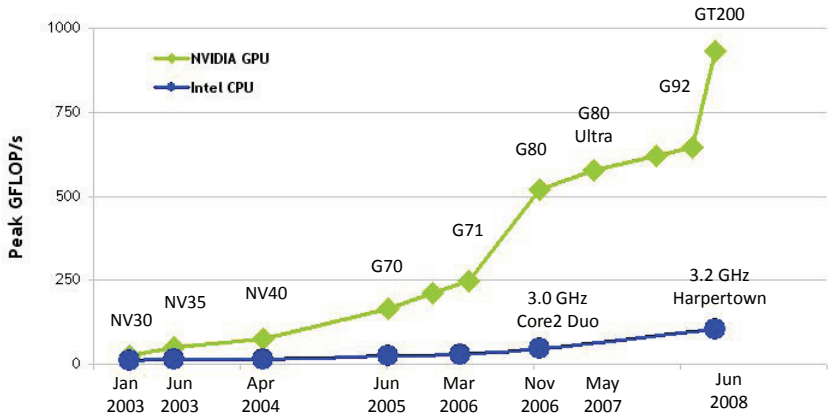


Figure 1.1: Growth of the power of GPUs vs. CPUs in the last few years. Courtesy NVIDIA

GPUs started out as fixed function processors, but gradually got more and more programmable to serve the ever-growing needs of game developers and consumers. An offshoot of enhanced programmability is that GPUs have become capable of general-purpose programming. Due to the large nature of the electronic entertainment market, this hardware is significantly cheaper than similar task-specific specialized processors, and fierce competition among vendors means that

GPU's architecture and products rapidly evolve with a very short product life-cycle. Consequently, GPUs were evolving at a much faster rate than CPUs in terms of raw computational power (Figure 1.1).

1.1 General-purpose Programming on Graphics Processing Units (GPGPU)

Real-time graphics rendering is handled by a graphics pipeline. The graphics hardware would have specific units to handle the processes at each part of this pipeline. GPUs entered a new era of flexibility when major parts of the pipeline were made programmable. Gradually, as researchers and developers could see the massive computational power that was available on these processors, graphics vendors started to support the use of this hardware as general-purpose processors. Today, we have many APIs which allow users to utilize the computational power of GPUs with little or no knowledge of graphics programming. With the advent of specialized programming environments like CUDA[6], Brook+[7], OpenCL[8] and DirectX compute shader[9], general-purpose programming on GPU (or GPGPU) has become mainstream and promises to offer enhanced performance for many applications using existing graphics hardware on end-user systems.

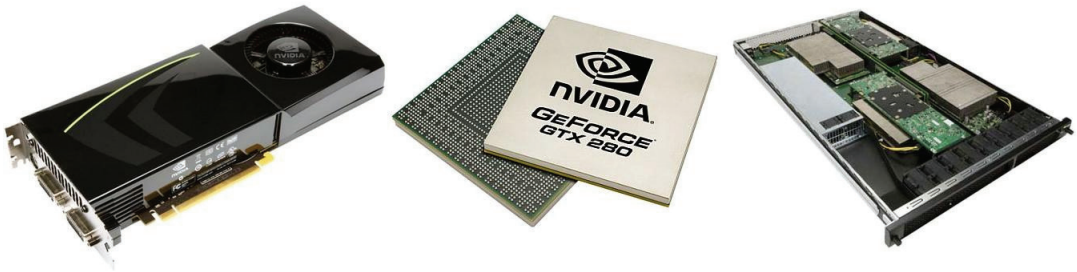


Figure 1.2: NVIDIA GTX280 graphics card, the GTX280 GPU, and 4 GPUs in a rack mountable Tesla unit. Courtesy NVIDIA

The most powerful GPU (the NVIDIA GTX 480) at the time of this writing has a peak computational power of about 2.7 TFLOPs and will be available for around $500. Just a few years back such computational power was available only on supercomputing clusters. Affordable supercomputing is the new mantra to meet the ever-growing requirements of researchers and businesses - and GPGPU has positioned itself as a primary solution in this segment.

1.2 Typical GPGPU Applications and CUDA

GPGPU applications are typically massively parallel applications that can be mapped easily to the GPU architecture or API. GPGPU was first explored with applications that were centered around graphics or image and signal processing. One problem that has been of interest in this area is image segmentation - finding features embedded in 2D or 3D images - these have applications in domains such as medical image processing, video processing and surveillance. In medical image processing, one of the most computationally demanding tasks is to extract 3D information embedded in various image slices obtained from MRI or CT scans - these require mathematical techniques such as the fast Fourier transform (FFT) and linear algebra techniques. Soon enough, general applications in the domains of scientific computing, physics simulation and computational finance started cropping up. A comprehensive survey by Owens *et al.*[10] details some of the early developments of GPGPU.

GPGPU entered mainstream acceptance with the advent of CUDA [6], a C-language API with extensions that allow developers with little or no graphics API experience to program for GPUs. CUDA exposed various facets of GPU hardware (various memories, texture reference options and constant caches) that were previously hidden to developers and enabled programmers to utilize the hardware to it's maximum potential.

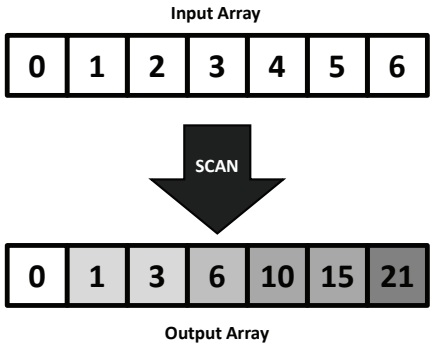


Figure 1.3: Parallel prefix sum (scan)

An early demonstration of CUDA's capability came with a path-breaking paper by Harris *et al.*[11] which describes an optimal implementation of parallel prefix sum (scan) (Figure 1.3) on the GPU. This work proved the viability of the GPU as a parallel architecture that is capable of doing some of the most basic steps in parallel computation and had extended its reach to many parallel application domains. Soon, many primitives for the GPU were developed, many of which are now available in NVIDIA's CUDA data-parallel primitives library (CUDPP) [1]. Suryakant *et al.*recently developed one of the fastest sorts for the GPU by utilizing a primitive called *split*[12] - a method to distribute elements of a list according to a category.

1.3 The PRAM Model and Irregularity in Algorithms and Memory Operations

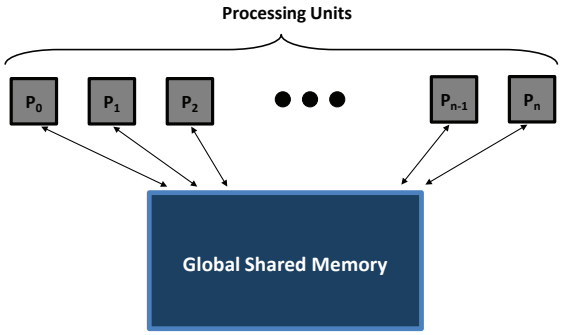


Figure 1.4: The PRAM Computation Model

The PRAM computational model[13] is a simple extension of the RAM computation model

which assumes uniform memory access latency across the architecture. As shown in (Figure 1.4), the PRAM model consists of a number of proecssors, and they exchange data through a global shared memory. Each processor is uniquely identified by an index, called a processor number or processor id. In the PRAM model, the processors are assumed to be operating synchronously under a shared clock.

Although the PRAM model is a simple model which abstracts most implementation details and is applicable to most shared memory architectures, it cannot accurately model modern parallel architectures that have varied data access latencies with respect to spacial and temporal locality of data. An algorithm in the PRAM model which has p processors reading data at random intervals from memory may be exactly as complex (in terms of work and time complexity) to one which reads data from contiguous or from specific patterns in memory. On real-world parallel hardware, however the implications of such memory access patterns may become the single largest factor in the performance of the algorithm. The effect of random memory access is largest on distributed systems which have large latencies (across nodes on a network) and can also affect the tightly integrated multi-core systems of today.

While sequential programming models focus on time and space complexity as a measure of the optimality of an algorithm, PRAM typically includes the additional notion of *work* complexity[14]. The work complexity of an algorithm in the PRAM model is simply the total number of operations done by all processors in a PRAM machine. Typically, a PRAM algorithm is *work optimal* if its work complexity is asymptotically the same as the time complexity of the best-known sequential algorithm.

On today's parallel architecture compute power is cheap. Memory operations are expensive. This has led algorithm implementers to rethink strategies when implementing algorithms on parallel hardware; An algorithm that looks good on paper may not be as efficient on a particular architecture.

Performance degradation of irregular algorithms on current hardware is largely a function of the hardware's architecture. Some generalizations can be made however - these involve varying degrees of spacial and temporal localities of data. Every modern parallel architecture utilizes in one form or another some type of cache - these nowadays are either user-managed or unmanaged (automated) caches. They have their respective advantages and disadvantages. Efficient and optimal code should utilize the various memory locations optimally. Such code is usually referred to as cache-friendly or cache-optimal in case of automated caches. Irregular applications typically have no room for optimization on such architectures, and result in performance degradation on such architectures as the cache's ability to prefetch useful data to the program is compromised. Figure 1.5 illustrates a program which is not cache-optimal as multiple threads are accessing various memory pages at random.

A recent trend in multi-core and parallel architectures is to remove automation from the caches and make them user-managed scratchpad memories. This trend is seen on GPUs such as the G80 and G200 and the Cell Broadband Engine[15]. This gives the developer maximum flexibility in placing the data in the memory hierarchy as required. This does require the programmer to explicitly move data from the main to scratch-pad memories, and incurs an additional coding and algorithm design challenge for the developer. Very recently, NVIDIA announced its next-generation Fermi architecture which features both user-managed scratchpad memory, as well as multilevel (L1 and L2) automated caches.

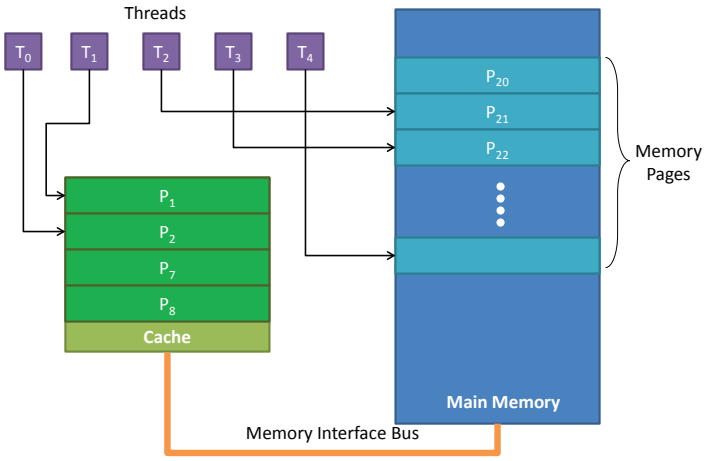


Figure 1.5: An example of a non-optimal memory access pattern in a cached multi-threaded architecture

1.4 List Ranking and Applications

Very often in many processing tasks, we encounter data structures such as lists and trees. A fundamental processing technique that is often performed on these data structures is *list ranking*, an elementary task that ranks the node from either end of the list.

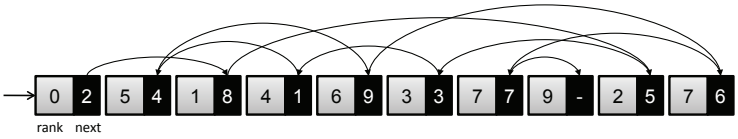


Figure 1.6: Linked list with the rank stored in each node

The algorithm for list ranking sequentially is trivial as it requires a simple list traversal and runs in $O(n)$ time (see Algorithm 1). In parallel though, it is complicated because of the sequential dependency of the data. Most parallel algorithms to solve the problem involve considerable work to divide the data suitably among the processing units. The range of techniques used to arrive at an efficient parallel solution include independent sets, ruling sets, and deterministic symmetry breaking. The non-contiguous structure of the list and irregular access of shared data by concurrent threads/processes make list ranking tricky to parallelize. Unlike the prefix sum or scan operation, there is no obvious way to divide a random list into even, disjoint, continuous sublists without first computing the rank of each node. Concurrent tasks may also visit the same node by different paths, requiring synchronization to ensure correctness. List ranking is a basic step of algorithms such as Euler Tour, load balancing, contraction, expression evaluation, 3-connectivity, planar graph embedding etc.

Algorithm 1 Sequential List Ranking Algorithm

Input: An array L, containing input list. Each element of L has two fields *rank* and *successor*, $n = |L|$, and a pointer to the head of the list, *start*

Output: Array R with ranks of each element of the list with respect to the head of the list

1: Set $count = 0$
2: Set $p = start$
3: **while** p is not the end of the list **do**
4: Set $p.rank = count$
5: Set $p = p.successor$
6: Increment *count*
7: **end while**

1.5 Contributions of this Thesis

In this thesis, we present efficient implementations of two important data-parallel but irregular primitives for the GPU. We first present an efficient implementation of *pointer jumping* on the GPU - an important primitive and technique that is used in many parallel algorithms. We explore its use for the list ranking problem on the GPU by implementing Wyllie's list ranking algorithm[13, 16]. A more efficient implementation of list ranking on GPU is also presented based on the Helman-JáJá algorithm for SMPs[17]. Using this optimized list-ranking algorithm, we implement a few Euler tour technique-based algorithms on the GPU for trees.

The main contributions of this thesis are:

1. Development of an optimized GPU implementation of pointer jumping on the GPU. This implementation is among the first for this parallel paradigm.

2. An implemenation of Wyllie's algorithm on the GPU that utilizes the pointer jumping primitive. The algorithm offers speedup over a sequential implementation for small lists but it is not much when compared to previous implementations on other parallel hardware.

3. A work-optimal, recursive, and efficient formulation of the Helman-JáJá list ranking algorithm for the GPU. This is the fastest single-chip implementation of the list-ranking algorithm on parallel hardware among reported literature. We also discuss some of the running parameters for the algorithm and how to arrive at the most optimal/efficient parameter for the GPU.

4. We present an application that utilizes our list-ranking primitive for the GPU. We present an efficient technique to construct an Euler-tour on the GPU.

5. We also present a few applications of the Euler-tour technique in solving some tree-characterization problems such as tree rooting, traversals and level computations.

6. We discuss, in light of a larger performance model that was developed for the GPU, the analysis and performance/behavior prediction for such irregular algorithms on the GPU.

Chapter 2

Background and Related Work

2.1 Parallel Computing and Architectures

Parallel programming has been a topic of research since the development of modern computers itself. It is the ever-growing need for solving large problems quicker and faster that drives the research and innovation behind parallel computing.

One of the earliest forms of parallel computing was seen in erstwhile supercomputers, usually being a collection of purpose-built processors which are connected together using a specially designed network to facilitate quick communication. These supercomputers would cost to the tune of millions of US dollars and were typically owned and operated by large government entities and would be usually deployed for strategic research. Soon enough large corporations started investing in such supercomputers to gain a competitive advantage in research and development.

During the last two decades or so, many organizations have been constructing their own parallel computers using commercial off-the-shelf (COTS) components and this has become the model for most new supercomputers such as the RoadRunner[18] etc. Some attempts at building very large distributed computing platforms (popularly known as grid computing) using the idle time of personal computers and devices connected to the internet have also been successful, such as the Folding@Home[19] or the SETI@Home[20] project; they are for computationally large problems that that would be too expensive to perform on a dedicated computer system. Patterson *et al.*[21] provides a good introduction and a survey of parallel computing research in general.

2.1.1 Shared Memory and Multicore Architectures

Commercially, parallel computing has entered the personal computing space and the mainstream in the last couple of years with the emergence of multicore and many-core processors - processors that contain more than one processing unit and can run applications in parallel - these systems typically share the same system RAM and are OS-controlled to run multiple threads in parallel.

Shared memory and multicore architectures are the main focus of developers today to create faster and more efficient applications that can take advantage of the ever-increasing number of cores per processor. Also adding to the problem is the fact that single-threaded performance of these processors have not increased significantly for sometime due to the so-called power wall[21]. Some of the software tools and environments used to program these new processors are OpenMP[22], Intel Thread Building Blocks[23], etc. A lot of active research is going into developing simpler and more powerful programming paradigms for programming these multicore chips. Also, a major design goal of newer commercial operating systems such as Windows 7 and Snow Leopard is in improving

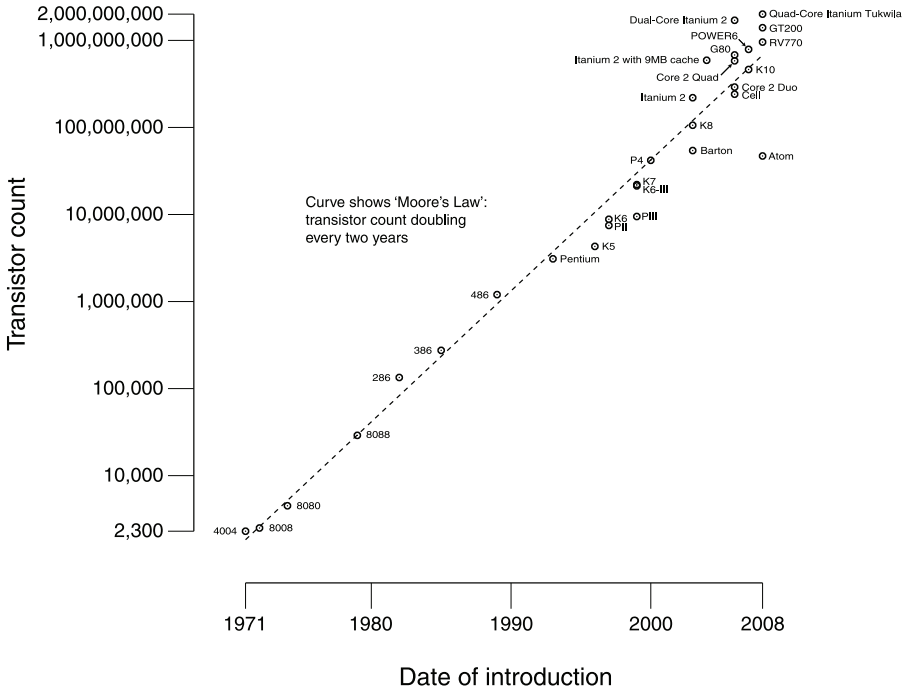


Figure 2.1: Transistor count over time

multi-threaded performance and enabling developers to fully exploit latest muticore chips, including graphics processors.

2.2 The Graphics Pipeline and GPUs

Modern GPUs are parallel processors in their own right. More specifically, they are known as stream processors as they are able to execute various functions on an incoming data stream. They are enhanced architectures that are good at data-parallel computation (primarily graphics). Once fixed function processors, they are now extremely powerful programmable processors with almost multiple instruction multiple data (MIMD) capabilities.

The current DirectX 10 graphics pipeline is pictured in Figure 2.2. This pipeline is designed to allow for efficient hardware implementations for maintaining quick and efficient real-time rendering of graphics applications. As seen in the figure, the pipeline is divided into multiple stages, and each stage is implemented as a separate hardware unit in the GPU (also termed as a task-parallel machine organization).

The input to the pipeline is a list of geometry, usually in the form of object vertex co-ordinates, and the output is an image in the frame-buffer, which is ready for display on the computer. In the first stage (geometry stage), each vertex from the object space is transformed into the screen space, the vertices are assembled into triangles and some lighting calculations are performed for each vertex. Once the geometry stage is complete, we obtain triangles in the screen space. In the next stage (rasterization), the screen position covered by each triangle is determined and the per-vertex parameters are interpolated for each triangle. As a result, a fragment for each pixel

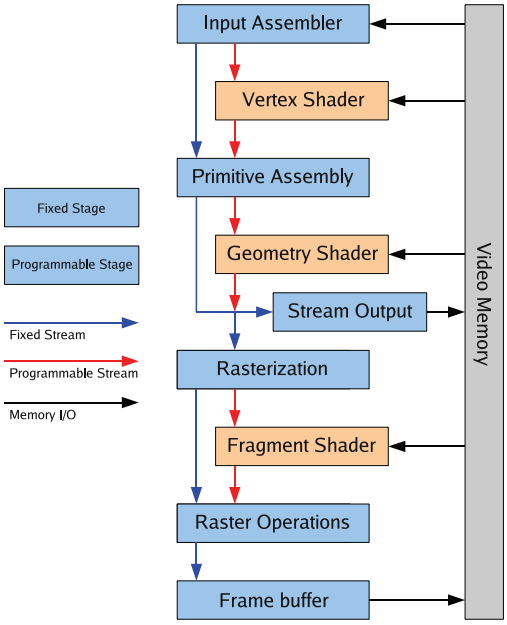


Figure 2.2: The DirectX 10 graphics pipeline

location covered by a triangle is obtained. In the next stage (fragment), a color for each fragment is determined with the help of images cached in the GPU memory called *textures*. Finally, all the fragments are assembled into an image of pixels using an approximation based on the viewing camera location. A GPU must complete these tasks and produce an output image at least 30 times a second to produce real-time 3D graphics. The interested reader is referred to the OpenGL Programming Guide [24] for more details.

2.3 Programmability, Unified Hardware, and GPGPU

The goal with each succeeding generation of GPUs was to produce more and more realistic graphics. This included special effects, especially in the form of lighting and shading. The fixed-function pipeline has transformed over the past decade into a more flexible and programmable pipeline by allowing developers to define small programs that process on each vertex, geometry element or pixel known as *shaders*. Shader programming was first introduced in Shader Model 3.0 with fully programmable vertex and pixel units. High level languages such as Cg, GLSL and HLSL came to being for writing shader programs.

As shading techniques, languages, and hardware evolved, researchers were able to utilize the additional flexibility in deploying non-graphics applications to the GPU, particularly in image processing. A full history of OpenGL and DirectX based GPGPU work can be found in [10].

A unified shader architecture was unveiled in DirectX 10 and Shader Model 4.0, along with hardware that supported it - thereby starting the current wave of GPGPU technologies. The field received a further boost with the advent of CUDA, a C-based GPGPU development environment from NVIDIA. CUDA allows developers unfamiliar with graphics programming to write code that can be executed on the GPU. CUDA provides the necessary abstractions for a developer to write massively multi-threaded programs with little or no knowledge of graphics APIs. Since then, there have been many implementations of parallelizable applications on the GPU[25], many offering substantial speedup over sequential implemenations on a CPU.

Also the adoption of two general vendor-independent stream computing standards : OpenCL[8] and DirectCompute[9] further enhance the stage for substantial growth in this field. The long term success of this computing paradigm may be unclear but it is certain that it will go a long way in shaping the future of parallel computing.

2.3.1 Current GPU Architecture

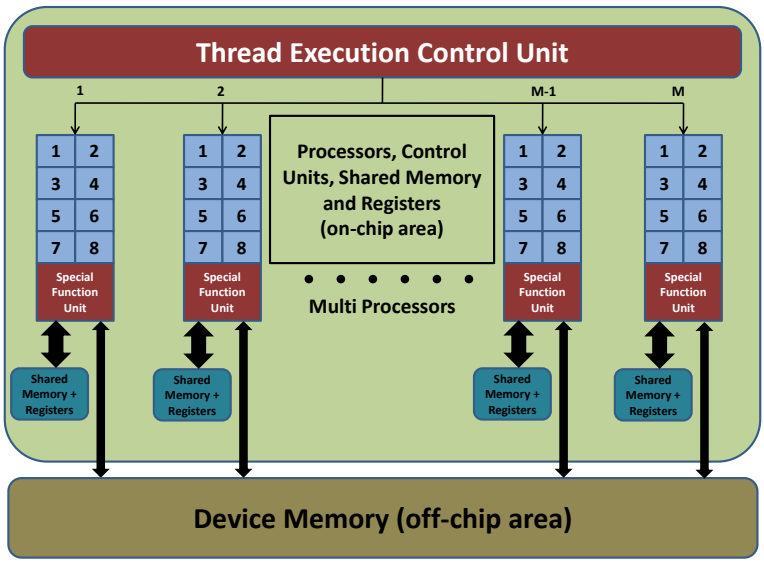


Figure 2.3: An abstracted model of the GT200 architecture from NVIDIA

In Figure 2.3, we can see the basic architecture of the GT200, a powerful GPGPU architecture at the time of this writing. NVIDIA designed this architecture with both graphics and GPGPU in mind.

The GT200 GPU consists of many stream processors or SPs. There are 240 SPs in the GT200. A group of 8 SPs form a streaming multiprocessor (SM). Each SM executes in SIMD mode and each instruction is quad clocked with a SIMD width of 4. The multiprocessors execute instructions asynchronously, without any communication between them. There is, however, a synchronization mechanism for all threads running within a block on an SM. All the threads in an SM also have access to a common high-speed scratchpad memory called *shared memory* which is limited to 16 KB for each SM. The only communication that can take place across SMs is through the use of the high-latency off-chip *global Memory* and due to the asynchronous and unordered execution of threads on this architecture, simultaneous writes to a global memory location will lead to an arbitrary thread succeeding in the write operation (similar to the CRCW Arbitrary PRAM model). This problem can be solved by using hardware atomic operations, which guarantees a write operation for each thread, but the order will still be arbirtrary.

Each SM also contains a special function unit or SFU which contains the logic to perform higher mathematical operations like square, divide, root, trignometric functions etc. Three SM's are grouped into a Texture Processing Cluster (TPC) and share some texture access hardware and cache - an alternate path to access global memory. A number of global memory controllers are also available to access off-chip global memory. More details on the various memory types available and how they are addressed will be discussed in the next section. Each core can store a number of

thread contexts. Data fetch latencies are tolerated by switching between threads. NVIDIA features a zero-overhead scheduling system by quick switching of thread contexts in the hardware.

2.3.2 CUDA Programming Model

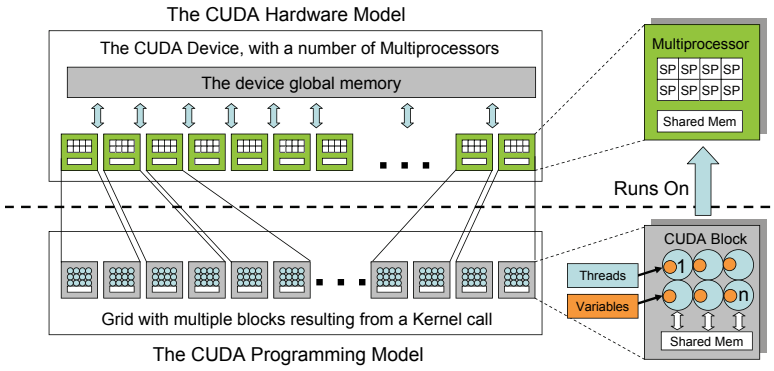


Figure 2.4: The CUDA hardware and programming model

The CUDA API allows a user to create a large number of threads to execute code on the GPU. Threads are also grouped into *blocks* and blocks make up a *grid*. Blocks are serially assigned for execution on each SM. The blocks themselves are divided into SIMD groups called *warps*, each containing 32 threads on current hardware. An SM executes one warp at a time. CUDA has a zero overhead scheduling which enables warps that are stalled on a memory fetch to be swapped for another warp.

The GPU also has various memory types at each level. A set of 32-bit registers is evenly divided among the threads in each SM. 16 KB of *shared memory* per SM acts as a user-managed cache and is available for all the threads in a Block. The GTX 280 is equipped with 1 GB of off-chip *global memory* which can be accessed by all the threads in the grid, but may incur hundreds of cycles of latency for each fetch/store. Global memory can also be accessed through two read-only caches known as the *constant memory* and *texture memory* for efficient access for each thread of a warp.

Computations that are to be performed on the GPU are specified in the code as explicit *kernels*. Prior to launching a kernel, all the data required for the computation must be transferred from the *host* (CPU) memory to the GPU global memory. A kernel invocation will hand over the control to the GPU, and the specified GPU code will be executed on this data. Barrier synchronization for all the threads in a block can be defined by the user in the kernel code. Apart from this, all the threads launched in a grid are independent and their execution or ordering cannot be controlled by the user. Global synchronization of all threads can only be performed across separate kernel launches. More details on the GTX280 GPU architecture and CUDA are available from NVIDIA [6, 26].

2.4 Irregular Algorithms

As mentioned earlier, many algorithms designed in the PRAM model were mainly concerned with work/time efficiency and not about memory locality. Initial efforts in trying to bring this notion of memory locality into parallel complexity analysis included the concept of gross locality - i.e, the ratio of computational complexity vs. the communication complexity [27] - where the efficiency of

the algorithm was based on the ability of the algorithm and the architecture to simulate PRAM by exploiting locality to offset communication costs.

Problems that have regular, oblivious, or predictable patterns of memory accesses easily lead to programs that are organized so that they have low communication costs. It may be argued that the more an algorithm is local the more it is regular. But this is not sufficient[28]; another characteristic for the irregularity of an algorithm is the irregularity of the communication patterns it involves. Irregularity of algorithms also depends on the parallel algorithm scheduling techniques, which has been studied in detail[29]. However on architectures such as the GPU where user-controlled scheduling is limited, the irregularity of the algorithm is mainly based on the memory addresses that each individual thread accesses in a warp of threads[5].

However with newer parallel models such as the Partitioned Global Address Space (PGAS)[30], there is a notion of memory efficiency and of distinct memory spaces such as 'local' and 'global' memory with associated work/time costs. There has been some effort of late to model asynchronous behavior that is seen in current generation multi and many-core architectures like the GPU and the Cell Broadband Engine[31].

2.5 List Ranking and Tree Algorithms

The list ranking problem was first posed by Wyllie[16], and solved it using a method now known as *pointer jumping* using $O(n \log n)$ steps for list of length n. This was gradually improved over subsequent efforts to linearly many steps on the EREW PRAM, most notably by Cole and Vishkin[32]. Anderson and Miller[33] also provided a deterministic solution using independent sets. These algorithms, however did not really provide efficient results on modern parallel hardware - primarily because of the irregular nature of the algorithm's communication costs. An efficient vector-parallel algorithm is described by Reid-Miller[34] and implemented on the Cray C-90. Using the idea of sparse ruling sets proposed in that paper, an efficient algorithm for symmetric multiprocessor systems was designed by Helman and JaJa[17]. More recently, an efficient algorithm for distributed systems is also discussed by Sibeyn[35]. List ranking is an often cited example of a classical irregular algorithm (when used with random or unordered lists), and is of independent interest in evaluating parallel architectures. Table 2.1 summarises the features of these popular list ranking algorithms.

Algorithm	**Time**	**Work**	**Constants**	**Space**
Serial	$O(n)$	$O(n)$	small	c
Wyllie[16]	$O(\frac{n \log n}{p} + \log n)$	$O(n \log n)$	small	$n + c$
Randomized[36, 37]	$O(\frac{n}{p} + \log n)$	$O(n)$	medium	$> 2n$
PRAM Optimal [32, 33]	$O(\frac{n}{p} + \log n)$	$O(n)$	large	$> n$
Miller-Reid[34]	$O(\frac{n}{p} + \log^2 n)$	$O(n)$	small	$5p + c$
Helman-JáJá [17]	$O(\frac{n}{p} + \log n)$	$O(n)$	small	$n + \log n + c$

Table 2.1: Complexities of various list ranking algorithms, for lists of length n and p processors on the standard PRAM model

Results on Parallel Hardware A through comparison of major list ranking algorithms was provided by Miller-Reid[34], and their algorithm has a substantial speedup compared to existing

algorithms at the time on the Cray Y-MP/C90. They were able to achieve an asymptotic execution time of 4.6 nsec per node on an 8 processor Cray C90 supercomputer.

The Helman-JáJá algorithm[17] has reported results on various SMP machines including the HP-Convex Exemplar, the SGI Power Challenge, the IBM SP-2 and the DEC AlphaServer - for a maximum of 4M elements using upto 16 threads. They has a runtime of about 2.4 seconds on the HP-Convex Exemplar for a list of 4 M elements.

Most of the recent performance data for list ranking is provided by Bader[2] for the Sun Enterprise server as well as the Cray MTA. More recent work from him[3] also contains an efficient implementation of JáJá's algorithm on the Cell Broadband Engine using a latency-hiding technique using software managed threads. His implementation on the cell processor can rank a list of 4 M nodes in about 106 milliseconds.

2.5.1 Applications of List Ranking

List ranking is a basic step of algorithms such as the Euler tour tecnhique (ETT) [13], load balancing, tree contraction and expression evaluation [38], connected components [39], planar graph embedding etc. [37].

Techniques for efficient tree traversals in parallel were first described by Wyllie [16] and then by Cole & Vishkin[40]. One of the first applications was expression evaluation[41]. The problem was discussed in the context of the PRAM model by various researchers including Kosaraju and Delcher [42]. Efficient algorithmic implementations of these techniques on the BSP and CGM parallel models were described in detail by Dehne [43] . A practical algorithm for ETT on SMP hardware is described by Bader [44].

2.6 Analytical and Performance Prediction Models for Parallel Computation

The parallel algorithms community has developed several models to design and analyze parallel algorithms such as the network model [13], the PRAM model [45], the Log-P model [46], the QRQW model [47, 48], among others. These models help the algorithm designer to exploit the parallelism in the problem. However, as these are mostly architecture independent they fail to help the algorithm designer leverage any specific advantages offered by the architecture or work around the ill-effects of any architectural limitations on the performance of the algorithm.

The PRAM Model: The PRAM model of parallel computation is a natural extension of the von Neumann model of sequential computation. Consider a set of processors each with a unique identifier called a processor index or processor number. Each processor is equipped with a local memory. Moreover, the processors can communicate with each other by exchanging data via a shared memory. Shared memory is sometimes also referred to as *global memory*. A schematic is shown in Figure 1.4. It is often also assumed that the processors operate in a synchronous manner. This model is called the PRAM (Parallel Random Access Memory) model.

Naturally, when the memory is shared between processors there can be contention for concurrent reads and writes. Depending on whether they are allowed or not, several variants of the PRAM model are possible with rules for resolving concurrent writes:

- EREW model: This stands for Exclusive Read Exclusive Write and in this model, no simultaneous reads or writes to the same memory location are permitted.

- CREW model: This stands for Concurrent Read Exclusive Write and in this model, while simultaneous reads are permitted to the same memory location simultaneous writes are not permitted.
- CRCW model: This stands for Concurrent Read Concurrent Write and in this model both concurrent reads and writes to the same memory location are allowed. Concurrent write needs more explanation as it is not easy to visualize what would be the result of concurrent write. Here, in theory, different variations are studied.

In the *Common CRCW* model, a concurrent write succeeds as long as all the processors are writing the same value to the common location. In the *arbitrary CRCW* model, any arbitrary processor can succeed in the write operation. In the *priority CRCW* model, it is assumed that the processor numbers are linearly ordered and the processor with the minimum index is allowed to succeed in the concurrent write operation. Other variants are also studied along with the relative power of each of these models.

The BSP Model: Valiant [49] proposed a bridging model called the Bulk Synchronous Parallel (BSP) model that aimed to bring together hardware and software practitioners. The model, which Valiant [49] shows can be easily realized also in hardware existing at that time, has three main parameters:

- A number of *components* that can perform computation and memory accesses;
- A *router* that transfers messages between components; and
- A facility for synchronizing all (or a subset) of the components at *regular* intervals of L time units. L is also called as the periodicity parameter.

Valiant suggests hashing to distribute memory accesses uniformly across the components. The performance of a router is captured by its ability to route h-relations, where each component is the source and the destination of at most h messages. Using the model one can then state the runtime of a parallel program in terms of the parameters L, p, g, and the input size n. In the above, p refers to the number of (physical) processors and g is the time taken by the router to route a permutation. The key idea of the model is to find a value of L so that optimality of runtime can be achieved, i.e., truly balance the local computation with message exchange time.

The QRQW Model: To offset the limitation of the PRAM model to handle memory contentions and their effect on the performance of a parallel program, Gibbons, Matias, and Ramachandran [48] introduced the Queue-Read-Queue-Write (QRQW) model. Here, in its simplest form, processors are allowed to contend for reading and writing at the same time. Contending accesses are queued. This, this model falls in between the Exclusive Read Exclusive Write (EREW) PRAM and the Concurrent Read Concurrent Write (CRCW) PRAM. The advantage of this model becomes clear when one sees that the EREW model of the PRAM is too strict and the CRCW model of the PRAM is too powerful. Hence, the QRQW model tries to separate the highly contentious accesses and accesses with very low contention. Most hardware realizations can support the latter better than the former. Moreover, it is observed that most existing machine models behave in a QRQW fashion.
In its general form, the model can also work with a contention function $f(.)$ that governs contentious accesses to the memory. While f being a linear function, we get the QRQW PRAM and, for example, $f(i) = \infty$ for $i > 1$ and $f(1) = 1$ is the EREW PRAM. The work of [48] studies variants such as synchronous and asynchronous QRQW PRAM.

Recent Models for GPUs: As far as modeling for performance on multicore architectures is concerned, there is very little reported work. In [50], the authors discuss the parameter space of GPU kernels and present ways to prune the size of such a space and get a highly optimized code. The result is a gain of about 17% for a particular medical imaging application [50]. However, our work does not target code or runtime optimizations. An extension of this work appears in [51] where the authors consider the multi-GPU design space optimization.

A performance prediction model appears in [52] where also the authors rely on separating memory and compute requirements. But their model is applicable only for a class of programs called "Iterative Stencil Loops". Very recently, Kim [53] presented work that predicts the runtime of a kernel on the GPU. They consider a set of 23 parameters that can be used to predict the runtime of a kernel on the GPU.

Chapter 3

Pointer Jumping

The pointer jumping technique is a fast way of processing data stored in rooted directed trees. In this chapter, we look at various ways by which this technique can be implemented on a massively multi-threaded architecture like the GPU, and use it to get an initial solution for the list ranking problem.

3.1 Pointer Jumping

We shall use the formal definition of the problem as stated in [13]. Let F be a forest consisting of a set of rooted directed trees (such as the one illustrated in Figure 3.1). The forest F is specified by an array P of length n such that $P(i) = j$ if (i, j) is an arc in F; that is, j is the parent of i in a tree of F. For simplicity, if i is a root, we set $P(i) = i$. The problem is to determine the root $S(j)$ of the tree containing the node j for each j between 1 and n.

A simple sequential algorithm to solve this problem is usually based on depth-first or breadth-first search which acomplishes this task in linear time. However, a fast parallel algorithm requires a completely different strategy.

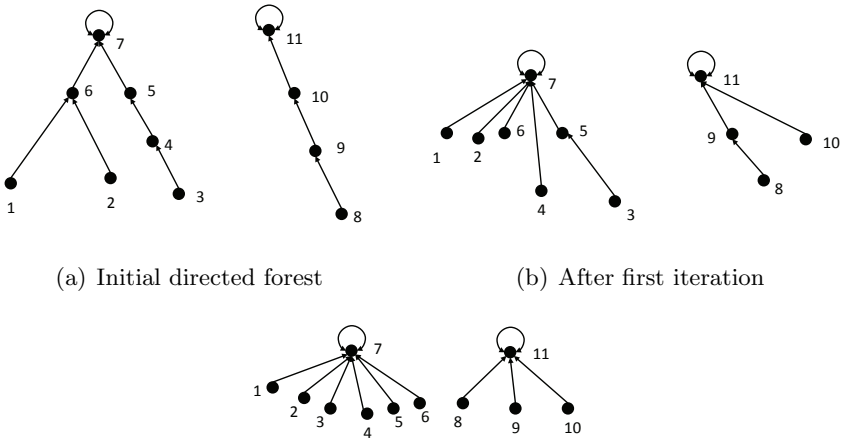
Pointer jumping is a simple technique by which every processor in a parallel machine updates the pointer of an element in a linked structure to the pointer it was pointing to. In other words, the successor of each node is updated to the successor of the successor. If the successor of a node i in a linked structure is $S(i)$ then the pointer jumping operation is defined as given in (Algorithm 2).

Algorithm 2 Pointer Jumping

Input: A forest of rooted directed trees.
Output: For each vertex i, the root $S(i)$ of the tree containing i

```
1: for 1 ≤ i ≤ n do in parallel
2:     while S(i) is not end of list do
3:         Set S(i) = S(S(i))
4:     end while
5: end for
```

An illustration of this technique is given in Figure 3.1. It is easy to see that when this procedure is applied repeatedly, each node gets closer to the root of the tree containing that node. Following k iterations of pointer jumping, the distance between a node i and it's successor $S(i)$ as they originally appeared in a directed tree T would be 2^k unless $S(i)$ is a root node. Figure 3.1 gives an



(a) Initial directed forest (b) After first iteration

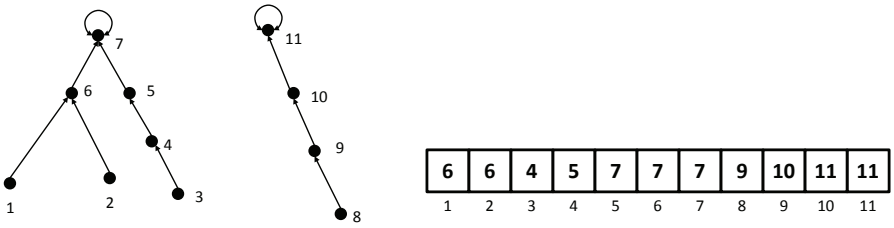
(c) After final iteration

Figure 3.1: The pointer jumping operation applied to a random forest. The numbered dots denote vertices and the arcs denote edges.

illustration of the pointer jumping technique applied to a directed forest.

Pointer jumping is one of the earliest techniques in parallel processing and its work complexity is $O(n \log h)$ operations where h is the maximum height of any tree in the directed forest.

3.1.1 Design for GPU



(a) A directed forest (b) Successor array representation

Figure 3.2: A directed forest and its successor array

Development of a pointer jumping algorithm for the GPU is a direct translation of the PRAM algorithm to the massively multi-threaded context of the GPU. We use one CUDA thread per node in the directed tree and iteratively apply the pointer jumping technique until we reach the end of that list. It must be noted that there is not much scope for the standard optimization techniques for this algorithm on the GPU. Since each element is read exactly once and the successor of a node can exist anywhere in the overall list, there is no useful prefetching technique that can be used in this context (which eliminates the use of shared memory). Since we need to continuously update these successor values in parallel, we also cannot use the read-only texture and constant caches (also the lack of any coherent access pattern also means that the use of these caches will be in vain).

In case an algorithm that uses the pointer jumping technique requires updating two or more values per node per iteration of pointer jumping, we require some sort of synchronization to ensure that all the values per node are updated simultaneously in each step. For example in an algorithm like list ranking, each node of a list will have two values - *rank* and *successor*. Both these values will have to be updated simultaneously, and a write-after-read (WAR) or read-after-write (RAW) hazard can mean that the algorithm will fail. For example, if one thread updated only the *rank* of an element and another thread read that value before *successor* could have been updated, the new thread would increment *rank* and perform an additional pointer jumping step for *successor*. Hence it is critical that both *rank* as well as sucessor are updated simultaneously.

3.2 Implementation of Wyllie's Algorithm on GPU

Wyllie's algorithm[16] was one of the first parallel algorithms used to solve the list ranking problem. The algorithm is very basic in its operation: repeatedly apply pointer jumping to a list and update the ranks of each node as it's pointer is replaced. This simple algorithm provides the rank of a list of size n in $O(\log n)$ time on a PRAM with n processors. It is described formally in Algorithm 3.

Algorithm 3 Wyllie's Algorithm

Input: An array S, containing successors of n nodes and array R with ranks initialized to 1
Output: Array R with ranks of each element of the list with respect to the head of the list

```
1: for each element in S do in parallel
2:     while S[i] and S[S[i]] are not the end of list do
3:         R[i] = R[i] + R[S[i]]
4:         S[i] = S[S[i]]
5:     end while
6: end for
```

Given a list of size n, the implementation of this algorithm (Algorithm 3) is to assign a process/thread per element of the list. If we assume that the GPU is synchronous in it's execution (a PRAM requrirement), each thread performs $O(\log n)$ steps, thereby making the total work complexity $O(n \log n)$. However, due to the fact that the threads are scheduled as warps on the hardware, one warp could complete execution well before another warp starts, which makes the hardware asynchronous in nature. Hence this assumption is not valid.

It is important to note that the rank and successor update (lines 3 and 4) in the algorithm must be performed simultaneously for each thread. If a process updates either the rank or successor value of an element in between the update operation of another process, the algorithm will fail. CUDA does not allow definition of critical sections in the code that can be effected throughout the grid. One solution to this problem is to provide a packed data structure which can be written to the global memory using a single memory transaction (see Figure 3.3).

CUDA does not have any explicit user management of threads or memory synchronization locks. Hence, for the implementation of wyllie's algorithm in CUDA, we need to pack two single precision words (elements R_i and S_i) into a double word and perform a 64 bit write operation(Figure 3.3). It must be noted that 64-bit operations are more expensive on the GPU.

Clearly, for the purpose of list ranking, pointer jumping is work suboptimal as an O(n) work is being performed here using $O(n \log n)$ steps. Also even if we don't consider the time taken to create the packed data structure as shown in Figure 3.3, the overheads of working with 64 bit memory elements at a time also slows the algorithm down on current generation graphics hardware.

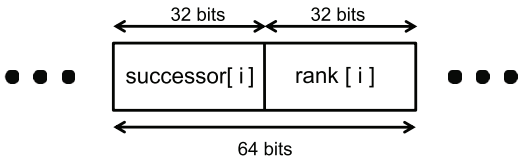
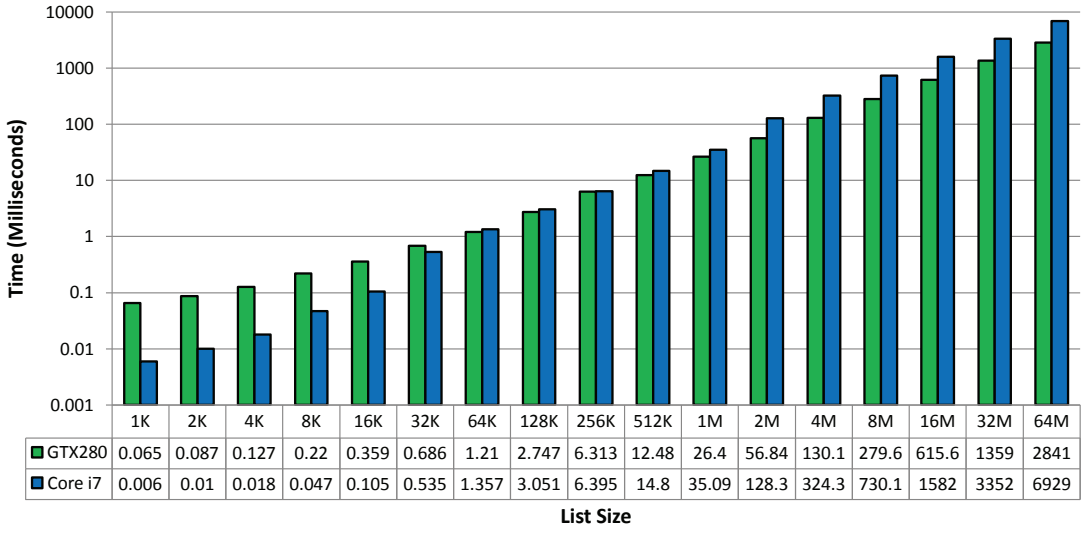


Figure 3.3: Packing two 32-bit values into a double word.

3.3 Results on GPU

Benchmarking Approach: We test our implementation on a PC with Intel Core 2 Quad Q6600 at 2.4 GHz, 2 GB RAM and a NVIDIA GTX 280 with 1 GB of on board Graphics RAM. The host was running Fedora Core 9, with NVIDIA Graphics Driver 177.67, and CUDA SDK/Toolkit version 2.0.



	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M	16M	32M	64M
GTX280	0.065	0.087	0.127	0.22	0.359	0.686	1.21	2.747	6.313	12.48	26.4	56.84	130.1	279.6	615.6	1359	2841
Core i7	0.006	0.01	0.018	0.047	0.105	0.535	1.357	3.051	6.395	14.8	35.09	128.3	324.3	730.1	1582	3352	6929

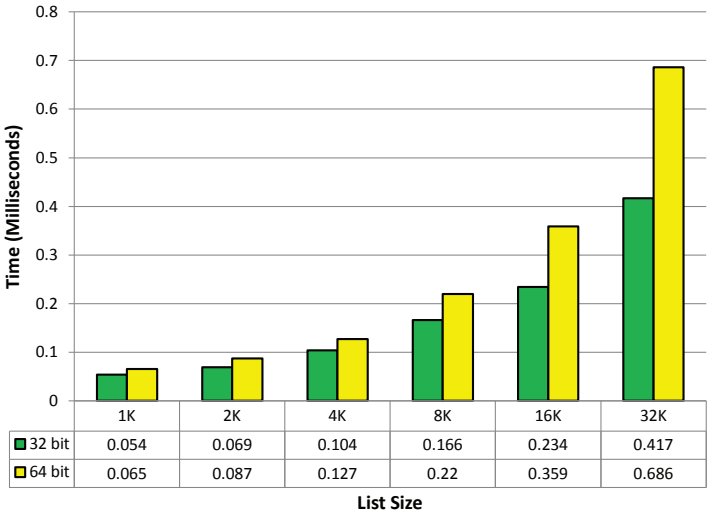
Figure 3.4: Runtime of Wyllie's algorithm on a Tesla C1070 GPU vs. sequential list ranking on Core i7 CPU

Figure 3.4 shows the runtime of the algorithm for various list sizes starting from 1024 to 64 Million elements. The runtime is compared with the $O(n)$ sequential algorithm running on a single core of the Intel Core i7 CPU. As we can see for small list sizes, the overhead of launching a GPU kernel and loading the elements from global memory. We see a small benefit from about 64K and about 2-3x speedup over CPU at the largest list sizes. This shows that the larger the list size, the greater the benefit when it comes to an architecture like the GPU.

3.3.1 Further Improvements and Variations

The following variants of the algorithm can be used to attain some more performance out of the algorithm:

32 bit packing for small lists: For lists of size $\leq 2^{16}$, the two 16-bit integers (representing $R[i]$ and $S[i]$) can be packed into a 32-bit integer and use a 32-bit write operation. As the GPU now has to deal with only 32-bit memory transactions per thread, this can result in time savings of up to 40% on such lists. Results are presented in Figure 3.5 for both the GTX 280 GPU. This has potential applications in sparse forests where an individual tree may not have more than 16K nodes.



	1K	2K	4K	8K	16K	32K
32 bit	0.054	0.069	0.104	0.166	0.234	0.417
64 bit	0.065	0.087	0.127	0.22	0.359	0.686

Figure 3.5: Comparison of the runtimes of the 32 and 64 bit version of Wyllie's algorithm on the Tesla C1070.

CPU-Orchestrated Pointer Jumping (Stepwise): In this version of Wyllie's algorithm, the Pointer-Jumping step is orchestrated by the CPU, and each GPU thread performs a single pointer jump for each kernel launch. This is done by packing an additional bit called *finished* which denotes if a particular node has reached the end of the list or not. We then modify the kernel to do a single step of pointer jumping with each call and orchestrate the execution from CPU. For this we also need a flag in global memory called *done*, which can be altered by any thread. If this value is left unaltered, then we can cease execution of the kernel. This implemenation thus performs pointer jumping in lock-step fashion and could be useful in implementations that require synchronization and additional operations with each jump. As execution of this version of the algorithm closely follows the PRAM model, it runs in $O(\log n)$ time and $O(n \log n)$ steps. Pseudocode for this version of the implementation is given in Algorithm 4:

Restricted Global Memory Operations (Optimized): This algorithm is modified for the GPU to load the various data elements from the global memory only when they are needed, as global memory read operations are expensive. The CUDA kernel pseudocde of this algorithm is presented in Algorithm 5. The designated element of N for a thread, is first read in r_1. The successor node is also loaded in r_2. Using bitwise operations, the rank and successor are read from these variables and updated as required. Finally after packing the new values in r_1, the new node is written back to it appropriate position in N. We synchronize all the threads in a block using the

Algorithm 4 CPU-Orchestrated Wyllie's Algorithm

Input: An array of nodes N, each with fields *rank next* and $finished$.
Output: N with all the node ranks updated

```
1: Set done = 0 in global memory
2: while some thread has set done = 0 do in parallel
3:     Set done = 1
4:     Load the node in N designated to this thread in local register r1
5:     if r1.finished = 0 then
6:         Load the successor node in local register r2
7:         if r2.finished = 0 then
8:             Set r1.next = r2.next
9:             done = 0
10:        end if
11:        Set r1.rank = r1.rank + r2.rank
12:        Synchronize and store the node in N designated to this thread with r1
13:    end if
14: end while
```

`__syncthreads` primitive to force all the eligible threads to read/write in a *coalesced* manner [6]. These operations are looped until all the nodes have set their successors to -1 (which denotes the end of the list).

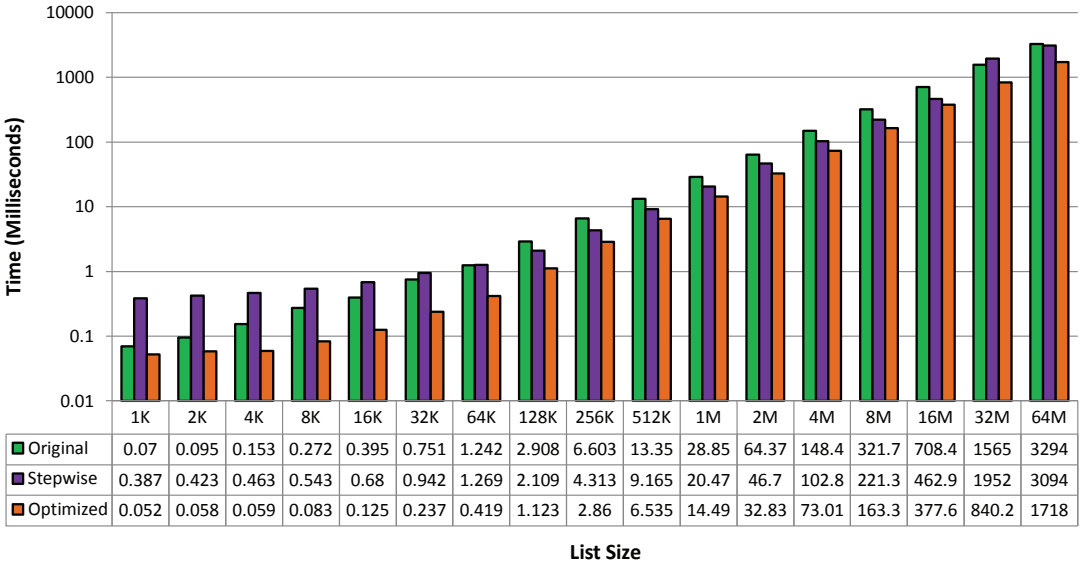
Algorithm 5 Reduced Memory Operations Wyllie's Algorithm

Input: An array of nodes N, each with fields *rank next* and $finished$ in global memory
Output: N with all the node ranks updated

```
1: for each node in list do in parallel
2:     Load the node in N designated to this thread in local register r1
3:     if r1 is end of list then
4:         Stop current thread
5:     end if
6:     Perform block synchronization
7:     Load the node pointed to by r1.next in r2
8:     if r2 is end of list then
9:         Stop current thread
10:    end if
11:    Set r1.next = r2.next
12:    Set r1.rank = r1.rank + r2.rank
13:    Perform block synchronization
14:    Write r1 back to its position in N
15: end for
```

In Figure 3.6, we see that the CPU-orchestrated version (stepwise) is much slower for small list sizes as the overhead to run the kernel $O(\log n)$ times is far greater than the actual time spent in processing such a small list. Towards the bigger sizes, however, we see that it does offer some benefit in runtime over the original implementation. The reduced operations version of the algorithm (optimized) runs best across all list sizes on the GTX280 GPU. This implemenation has better global memory coalescing and ineligible threads exit early, hence reducing the total number

of global memory transactions.



	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M	16M	32M	64M
Original	0.07	0.095	0.153	0.272	0.395	0.751	1.242	2.908	6.603	13.35	28.85	64.37	148.4	321.7	708.4	1565	3294
Stepwise	0.387	0.423	0.463	0.543	0.68	0.942	1.269	2.109	4.313	9.165	20.47	46.7	102.8	221.3	462.9	1952	3094
Optimized	0.052	0.058	0.059	0.083	0.125	0.237	0.419	1.123	2.86	6.535	14.49	32.83	73.01	163.3	377.6	840.2	1718

Figure 3.6: Comparison of the original version of the Wyllie's algorithm with the CPU-orchestrated (stepwise) and reduced operations (optimized) versions on a GTX280 GPU.

Chapter 4

Towards Efficient List Ranking

As described in Section 3.2, solving the list ranking problem on the GPU using pointer jumping is not exactly optimal as its work complexity is $O(n \log n)$. Multiple methods to overcome this have been discussed in Section 2.5. One of the more recent algorithms is the Helman Já Já Algorithm for shared memory systems. This algorithm is based on the sparse ruling set approach that was introduced by Reid-Miller [34]. In this chapter, we shall describe this algorithm and the modifications required to create an optimal implementation for GPUs.

4.1 The Helman-JáJá Algorithm for SMPs

Helman and JáJá's algorithm was originally designed for symmetric multiprocessors (SMP) [17]. The parallel algorithm for a machine with p processors is as follows:

Algorithm 6 Helman and JáJá List Ranking Algorithm

Input: An array L, containing input list. Each element of L has two fields *rank* and *successor*, and $n = |L|$

Output: Array R with ranks of each element of the list with respect to the head of the list

1: Partition L into $\frac{n}{p}$ sublists by choosing a splitter at regular indices separated by p.
2: Processor p_i traverses each sublist, computing the local (with respect to the start of the sublist) ranks of each element and store in $R[i]$.
3: Rank the array of sublists S sequentially on processor p_0
4: Use each processor to add $\frac{n}{p}$ elements with their corresponding splitter prefix in S and store the final rank in $R[i]$

Figure 4.1 illustrates the working of the algorithm on a small list. This algorithm (Algorithm 6) splits a random list into $\frac{n}{p}$ sublists in the splitting phase (Figure 4.1(a)). The rank of each node with respect to the "head" of each sublist is computed by traversing the successor array until we reach another sublist (local ranking phase - Figure 4.1(b)). This is done in parallel for $\frac{n}{p}$ sublists. The length of each sublist is used as the prefix value for the next sublist. A new list containing the prefixes is ranked sequentially (global ranking phase - Figure 4.1(c)) and then subsequently matched and added to each local rank in the final phase (recombination phase). Helman and JáJá proved that for a small constant, when p is small, the worst case run time is $O\left(\log n + \frac{n}{p}\right)$ with $O(n)$ work [17]. Bader presented implementations of this algorithm on various architectures such as the Cray MTA-2 and Sun enterprise servers [2], and recently with the IBM Cell processor [3], a hybrid multi-core architecture.

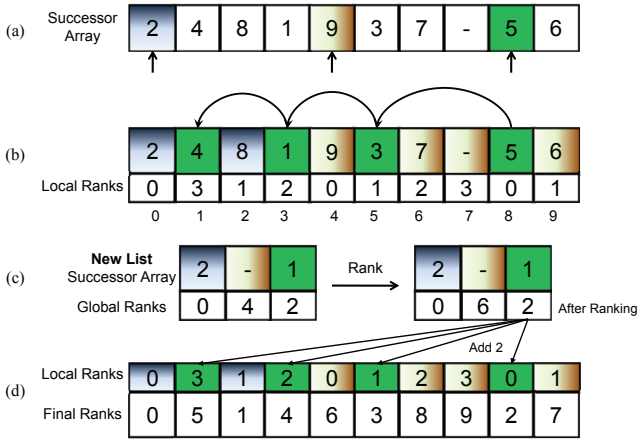


Figure 4.1: A sample run of the Helman and JáJá list ranking algorithm.

4.2 Recursive Helman JáJá Algorithm

The algorithm, as described by Helman and JáJá [17] reduces a list of size n to $\frac{n}{p}$ and then uses one processing node to calculate the prefix of each node of this new list. It would be unwise to apply the same technique to the GPU, as this will leave most of the hardware underutilized after the first step. Instead, we modify the ranking step of the original algorithm to reduce the list recursively until we reach a list that is small enough to be tackled efficiently by the GPU or by handing over to either the CPU for sequential processing or to Wylie's algorithm on the GPU.

Algorithm 7 Recursive Helman-JáJá **RHJ**(L,R,n,$limit$)

Input: Array L, containing the successors of each list element and $n = |L|$
Output: Array R containing the rank of each element in L

1: **if** $n <= limit$ **then**
2: Rank the final list L sequentially
3: **else**
4: Choose p splitters at intervals of $\frac{n}{p}$
5: Define the sublists on L
6: Copy these splitter elements to L_1
7: Perform local ranking on each sublist of L
8: Save the local ranks of each element of L in R.
9: Set successor pointers for L_1
10: Write sublist lengths in R_1
11: Call Procedure **RHJ**(L_1,$R_1, p, limit$)
12: **end if**
13: Each element in R adds the rank of its sublist from R_1 to its local rank.

The recursive version of the Helman JáJá algorithm is listed in Algorithm 7 . Each element of array L contains both a successor and rank field. In step 4 of the algorithm, we select p splitter elements from L. These p elements will denote the start of a new sublist and each sublist will be

represented by a single element in L_1. The record of splitters is kept in the newly created array L_1. Once all the splitters are marked, each sublist is traversed sequentially by a processor from its assigned splitter until another sublist is encountered (as marked in the previous step). During traversal, the local ranks of the elements are written in list R. Once sublist has been traversed, the successor information, i.e. the element (with respect to indices in L_1) that comes next is recorded in the successor field of L_1. It also writes the sum of local ranks calculated during the sublist traversal to the rank field of the succeeding element in L_1.

Step 6 performs the recursive step. It performs the same operations on the reduced list L_1. The condition in step 1 of the algorithm determines when we have a list that is small enough to be ranked sequentially, which is when the recursive call stops.

Finally, in step 8, we add the prefix values of the elements in R_1 to the corresponding elements in R. Upon completion of an iteration of RHJ, the List R will have the correct rank information with respect to the level in the recursive execution of the program.

The recursive Hellman-JáJá algorithm can be analyzed in the PRAM model [13] as follows. For $p = n/\log n$ sublists, i.e., with p splitters, the expected length of each sublist is $O(\log n)$. This is due to the fact that we are working with a randomized list and the splitters are chosen at equal intervals. The actual sublist length can vary, but the expected value will be $O(\log n)$. Hence, the expected runtime can be captured by the recurrence relation $T(n) = T(\frac{n}{\log n}) + O(\log n)$, which has a solution of $O\left(\frac{\log^2 n}{\log \log n}\right)$. Similarly, the work performed by the algorithm has the recurrence relation $W(n) = W(\frac{n}{\log n}) + O(n)$ which has a solution of $O(n)$. So, our adaptation of the algorithm is work-optimal and has an approximate runtime of $O(\log n)$. Notice however that the deviation from the expected runtime for $p = n/\log n$ could be $O(\log^2 n)$, for random lists. The correctness of the algorithm should be apparent from its construction and PRAM style runtime.

4.2.1 Design and Implementation in CUDA

Implementing the RHJ algorithm on the GPU is challenging for the following reasons:

- Programs executed on the GPU are written as CUDA kernels.
- They have their own address space on the GPU's instruction memory, which is not user-accessible. Hence CUDA does not support function recursion.
- Also, each step of the algorithm requires complete synchronization among all the threads before it can proceed. These challenges are tackled in our implementation, which is discussed in the next section.

All operations that are to be completed independently are arranged in kernels. Global synchronization among threads can be guaranteed only at kernel boundaries in CUDA. This also ensures that all the data in the global memory is updated before the next kernel is launched. Since each of the 4 phases of the algorithm require a global synchronization across all threads (and not just of those within a block) we implement each phase of the algorithm as a separate CUDA kernel. Since we are relying purely on global memory for our computation, CUDA blocks do not have any special significance here except for thread management and scheduling. We follow NVIDIA's guidelines to keep the block size as 512 threads per block to ensure maximum occupancy[50][6].

The first kernel is launched with p threads to select p splitters and write to the new list L_1. The second kernel also launches p threads, each of which traverse its assigned sublist sequentially, whilst updating the local ranks of each element and finally writing the sublist rank in L_1. The recursive step, is implemented as the next iteration of these kernels, with the CPU doing the book-keeping

between recursive levels. Finally we launch a thread for each element in L to add the local rank with the appropriate global sublist rank.

Managing Recursion: A wrapper function that calls these GPU kernels is created on the CPU. It takes care of the recursive step and the recursion stack is maintained on the host memory. CUDA also requires that all the GPU global memory be allocated and all input data required by the kernels be copied to the GPU beforehand. Once we obtain the list size, the required memory space is calculated and allocated for the entire depth of recursion before we enter the function. Since we do not know the exact sizes of the sublists in each step of the recursion, we allocate $n \times h$ space on the GPU before executing the algorithm, where n is the size of the input list and h is the depth of recursion.

The final sequential ranking step (which is determined by the variable *limit*) can be achieved in a number of ways. It can either be handed over to Wyllie's algorithm, or be copied to the CPU or be done by a single CUDA thread (provided that the list is small enough).

4.3 Load Balancing RHJ for the GPU

The original algorithm also calls for $\frac{n}{p \log n}$ splitters to be chosen at random from the list, where p is the number processing elements. With $p \ll n$ for SMP architectures, Helman *et al.*[17] went on to prove that with high probability, the number of elements traversed by a processor is no more than α where $\alpha(s) \geq 2.62$.

The GTX280, however, requires around 1024 threads per SM to keep all the cores occupied and offset memory fetch latencies from global memory. For our implementation, the assumption that $p \ll n$ no longer holds good as we have large number of threads that are proportional to the size of the list. For this implementation to succeed we need a good choice of the number of random splitters S such that we are able to get evenly-sized sublists with high probability. For a list of size n and number of sublists p, a few parameters can help us here in deciding the uniformity of the size the sublists: the number of singleton sublists, the standard deviation and the frequency of various sublist sizes.

To understand the impact of these parameters on the runtime, we implemented the RHJ algorithm on the GPU with two choices of splitters: $\frac{n}{\log n}$ and $\frac{n}{2 \log^2 n}$. Observe that with $\frac{n}{\log n}$ splitters, the expected number of singleton sublists for a list of 8 M elements is about 16,000. This means that load is not properly balanced as some lists tend to be larger than the others. With $\frac{n}{2 \log^2 n}$ splitters, the expected number of singletons is under 10. However, this has the effect of increasing the number of elements in each sublist on the average.

In a particular run on a list of 8 M elements, the largest sublist size with $\frac{n}{\log n}$ splitters is 350 while with $\frac{n}{2 \log^2 n}$ the largest sublist has a size of around 10,000 elements. Similarly, the deviation in the case of using $\frac{n}{\log n}$ is higher compared to the deviation with $\frac{n}{2 \log^2 n}$ splitters.

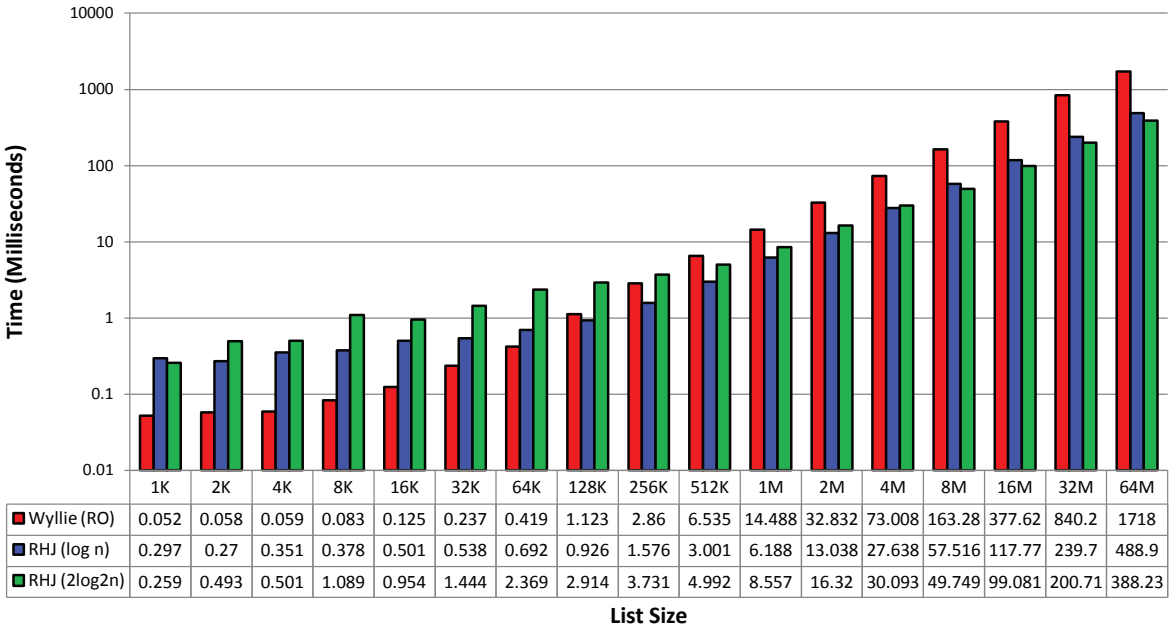
Guided by these observations, for large lists we used $\frac{n}{2 \log^2 n}$ splitters in the first iteration and used $\frac{n}{\log n}$ splitters in the remaining iterations (of the recursion). One hopes that this would give good performance over $\frac{n}{\log n}$ splitters consistently. The performance of RHJ on GPU with $\frac{n}{2 \log^2 n}$ splitters is better than $\frac{n}{\log n}$ for lists of size 8 M and above.

4.4 Results

Benchmarking Approach As in Section 3.3, the various implementations discussed so far were tested on a PC with Intel Core 2 Quad Q6600 at 2.4 GHz, 2 GB RAM and a NVIDIA GTX 280 with 1 GB of on board Graphics RAM. The host was running Fedora Core 9, with NVIDIA Graphics Driver 185.18, and CUDA SDK/Toolkit version 2.0. These algorithms are also compared to our previous implemention in the last chapter, as well as the standard $O(n)$ sequential algorithm running on the same PC (**Q6600**)- hereby referred to as the CPU Sequential Algorithm. We also test our implementation on both random and ordered lists.

Figure 4.2 shows the RHJ algorithm (using both $\log n$ and $2\log^2 n$ splitters) and compares it to the most optimized version of the Wyllie's algorithm presented in the previous section. RHJ offers substantial speedup over Wyllie's algorithm, with upto 3.5x speedup over the most optimized version. This is primarily because of the reduction in amount of work, and decreased contention among threads as they are now working with independent sublists of the original list.

The results of implementing RHJ on the GPU clearly shows that dividing the input data into non-overlapping segments and introducing global synchronization (through multiple kernels) will clearly benefit performance as compared to a single kernel that relies heavily on atomic operations to implement critical sections.



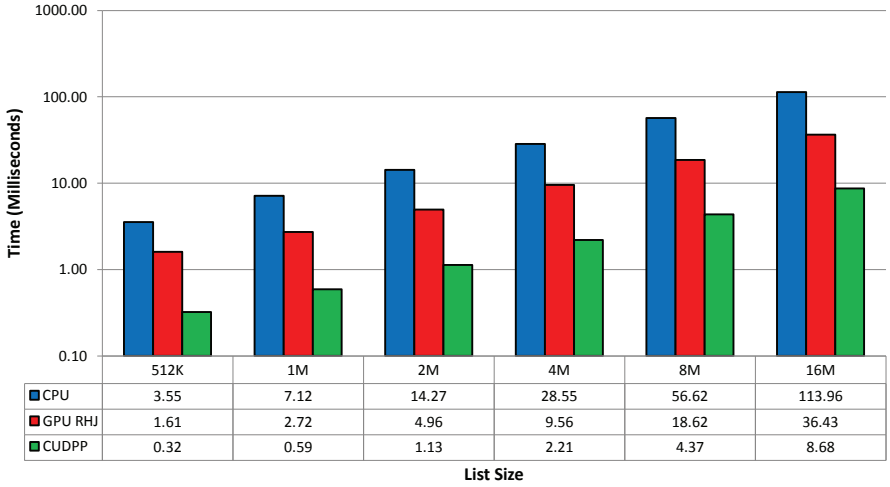
	1K	2K	4K	8K	16K	32K	64K	128K	256K	512K	1M	2M	4M	8M	16M	32M	64M
Wyllie (RO)	0.052	0.058	0.059	0.083	0.125	0.237	0.419	1.123	2.86	6.535	14.488	32.832	73.008	163.28	377.62	840.2	1718
RHJ (log n)	0.297	0.27	0.351	0.378	0.501	0.538	0.692	0.926	1.576	3.001	6.188	13.038	27.638	57.516	117.77	239.7	488.9
RHJ (2log2n)	0.259	0.493	0.501	1.089	0.954	1.444	2.369	2.914	3.731	4.992	8.557	16.32	30.093	49.749	99.081	200.71	388.23

Figure 4.2: Comparison of the most optimized version of Wyllie's algorithm with RHJ versions of list ranking.

4.4.1 RHJ on Ordered Lists

An ordered list is defined as a list where the successor of element i is $i+1$. Applying the list ranking algorithm on such an algorithm is equivalent to parallel prefix-sum or scan. This algorithm takes

advantage of the structure of an ordered list as the indices of each element in the array are enough to determine the order of elements. The algorithm can then take advantage of data locality that is present. For example, the Blelloch scan [54], does in-place adds and swaps of array elements in a tree-based fashion.



	512K	1M	2M	4M	8M	16M
CPU	3.55	7.12	14.27	28.55	56.62	113.96
GPU RHJ	1.61	2.72	4.96	9.56	18.62	36.43
CUDPP	0.32	0.59	1.13	2.21	4.37	8.68

Figure 4.3: Performance of the RHJ algorithm on an ordered list vs. the implementation for ordered lists from CUDPP [1] and a simple CPU sequential scan.

We compare our implementation of RHJ on ordered lists with the CUDPP CUDA ordered scan algorithm by Harris *et al.*[1] (Figure 4.4.1). Scan works well for ordered lists since it has been heavily optimized and takes advantage of the ordered structure to hide global access latencies through the use of shared memory.

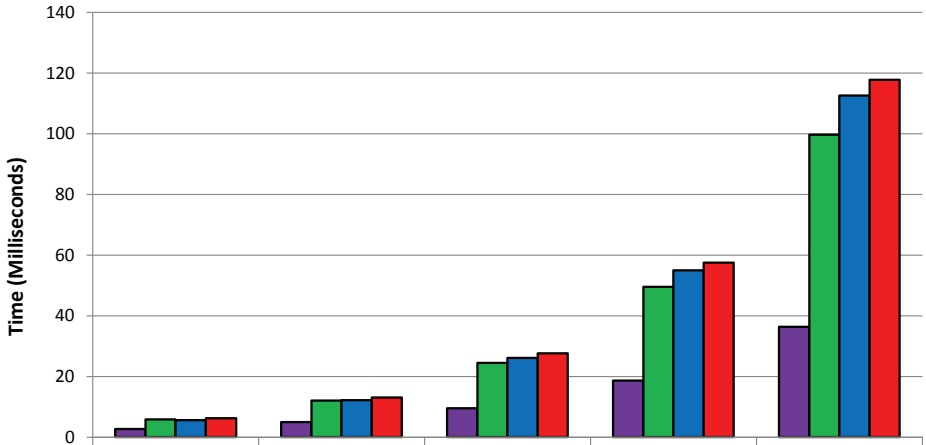
Ordered lists perform 5-10 times better than random lists on RHJ itself due to perfect load balancing among the threads and the high performance of the GPU of coalesced memory accesses. Note that the algorithm does not know that the lists are ordered. CUDPP scan performs about 8-10 times faster than RHJ on ordered lists. Scan operation, however, works only on ordered lists and cannot be used on random lists. A recent paper by Dotsenko [55], claims to be even faster than scan on the GPU.

4.4.2 Effect of Load-Balancing and Memory Coalescing

Random lists perform significantly worse than ordered lists using the RHJ algorithm due to two reasons: unbalanced load in the sequential ranking step and irregular memory access patterns due to the random list. On the other hand, scan [11] has regular and perfectly coalesced memory access as well as even work loads. We studied the impact of access pattern and load balance separately.

We create regular access patterns with unbalanced loads by selecting random splitters on a sequential list. Each thread accesses consecutive elements of the list. The GPU does not use caching on global memory to take advantage of this, however. It coalesces or groups proximage memory accesses from the threads of a half-warp (currently 16 consecutive threads) into a single memory transaction. Coalesced global memory access [6] is a major factor that determines the time taken to service a global memory request for a warp. In the GTX 280, if all the threads of a half-warp access a memory segment of less than 128 bytes, it is serviced with a single memory

transaction. In other words, a single memory transaction fetches an entire segment of memory, and a memory transaction devoted to retrieving a single word will lead to wasted memory bandwidth. Regular access pattern can improve the coalescing performance, but not completely. Chapter 6 discusses irregular global memory accesses and thier impact on performance in detail.



	1M	2M	4M	8M	16M
Ordered List	2.719	4.956	9.56	18.615	36.432
Ordered/Unbalanced	5.897	12.111	24.523	49.585	99.716
Random/Balanced	5.649	12.211	26.164	54.992	112.59
Random List	6.188	13.038	27.638	57.516	117.771

List Size

Figure 4.4: Effect of load balancing and memory coalescing on list ranking for lists of 1 million to 16 million nodes on the recursive RHJ algorithm on a GTX 280.

We create random access with balanced load using a random list, but with splitters pre-selected to be equidistant from each other. This ensures that all threads handle the same number of elements in the sequential ranking step, but the memory access pattern is totally random. The random list is completely uncoalesced as well as load-imbalanced and performs the worst.

Figure 4.4 shows the effects of load imbalance and irregular global memory access on the performance. Regular memory access even with unbalanced load seems to perform marginally better than irregular access with balanced load. It should be noted that regular memory access pattern does not guarantee full coalescing of memory transactions without perfect splitter selection. Both are, however, much closer to the performance on a random list than the performance on a sequential list. Scan does much better and can be thought of as selecting every other element as a splitter on a sequential list.

4.4.3 Profile of RHJ on the GPU

To understand the reason for the observed runtime, we used the CUDA profiler (provided by CUDA SDK) to breakup the runtime into various phases of the algorithm. Table 4.1 shows the complete timing breakup required by the 3 phases of the algorithm for the first iteration along with the total runtime. Notice that a large fraction of the total time is spent in the local ranking step of the first iteration. This implies that optimizations to this step can further improve the performance. In a particular run on a list of 8 M elements, as reported in Section 4.3, the largest sublist had

List Size	Time for First Iteration (μsec)			Total Time (μsec)
	Split	Local Rank	Recombine	
1 M	11	4404	1094	7182
2 M	15	9588	922	13367
4 M	24	19958	1881	26927
8 M	620	40806	11422	57372

Table 4.1: Profile of running time of RHJ on different lists. Total time is the runtime across all iterations.

about 10000 elements, while the number of singleton sublists was around 16000. Using the CUDA documentation [6], the approximate runtime can be computed to be about 100 milliseconds, by taking the global memory access times into account. Since the size of the list ranked in subsequent iterations is very small compared to the size of the original list, the combined runtime of all the subsequent iterations is under 1% of the total.

4.4.4 GPU vs. Other Architectures

In Figure 4.4.4, the running time of the RHJ algorithm is compared to the following implementations:

1. **Q6600:** Sequential implementation on the Intel Core 2 Quad Q6600, 2.4 GHz with 8 MB Cache
2. **Cell BE:** IBM Blade Center QS20 - on 1 Cell BE processor (3.2 GHz, 512 MB Cache) with 1 GB RAM, Cell SDK 1.1 [3]
3. **MTA-8:** 8 x Cray MTA-2 220 MHz (No data cache) processors working in parallel [2]
4. **E4500** 8 x Sun UltraSPARC 400 MHz processors in a SMP System (Sun Enterprise Server E4500) [2]

In Figure 4.4.4, we provide the performance comparison of the GTX 280 vs. the Cell BE for random lists from 1 million to 8 million nodes. We can see that we have a sustained performance benefit of about 3-4x on these lists. The Cell BE implementation is the most efficient till date and features a latency hiding technique. Also, the Cell has user managed threads and an efficient DMA engine. A fair comparison of the various architectural parameters of Cell vs. GPU is really not possible due to the fundamental difference in their architectures.

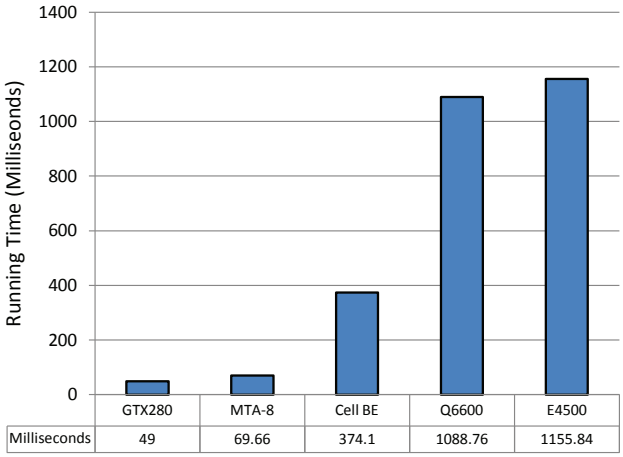


Figure 4.5: Comparison of RHJ list ranking on GTX 280 against other architectures [2, 3].

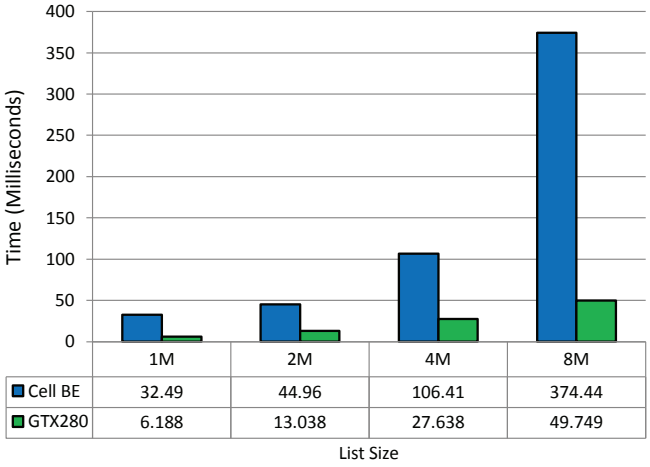


Figure 4.6: Performance of RHJ list ranking on GTX 280 and Cell BE [3] for random lists.

Chapter 5

Applications of List Ranking

In this section we will outline the Euler Tour Technique (ETT) and its applications and their implementations on the GPU. The Euler tour technique is an efficient method used to process graphs in parallel algorithms. It works for trees and is applicable for all computations that can use their spanning trees.

5.1 The Euler Tour Technique (ETT)

An Euler cycle/circuit of a graph G is a cycle that traverses every edge of G exactly once. Since each traversal of a graph node consumes two incident edges, the necessary condition for a connected graph to be Eulerian is that every node has even degree. It is known that this is also a sufficient condition. Similarly, a directed graph is Eulerian iff every node has equal input and output degree. An undirected tree can be made Eulerian if every tree edge is replaced with a pair of anti-parallel arcs.

The Euler Tour Technique (ETT) is a method to do tree computations in parallel by using the properties of an eulerian circuit that is embedded in this newly constructed graph. It is a powerful technique which enables fast computations of tree functions in parallel.

5.1.1 Constructing an Euler Tour

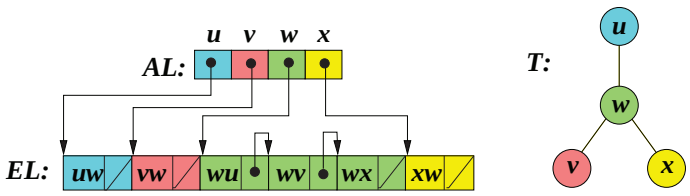


Figure 5.1: A simple tree T and its adjacency and edge list representation.

Consider an undirected tree T represented as an adjacency list in the shared memory of a PRAM machine. This representation consists of an array AL (Adjacency List) indexed by nodes. For $x \in V(T), AL[x]$ is a pointer to the list of edges incident to x. For simplicity, we assume that all these edges are stored in an array EL (Edge List). Each element of EL is a record of two items

specification of the edge (= pair of nodes) and pointer $Next$ to the next edge adjacent from the same node. See Figure 5.1 for an example of an tree and its corresponding adjacency and edge lists.

In order to transform tree T to an eulerian graph T' we need to introduce additional edges that make the in-degree of each vertex equal to its out degree. This is achieved by: Replacing each edge $\langle i, j \rangle$ with a set of anti-parallel arcs $\langle i \rightarrow j \rangle$ and $\langle j \rightarrow i \rangle$. In the edge list, EL we introduce an additional pointer Sib which is a pointer to the arc's anti-parallel sibling.

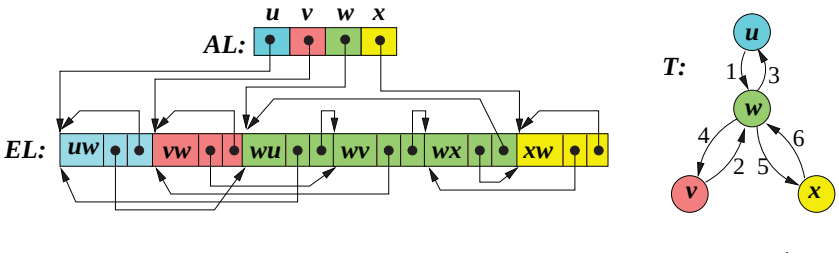
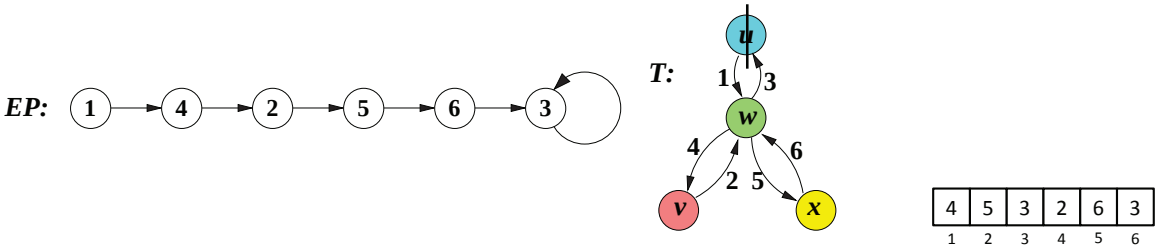


Figure 5.2: An Eulerian graph T' and the corresponding adjacency and edge lists

Once the Eulerian graph is constructed, we can now create an *Euler path* (EP). This is done by numbering all the arcs in the graph in some order and constructing a linked list of these arcs. Similar to the successor array shown in Figure 3.2(b), this linked list can be stored in a successor array as shown in Figure 5.3.



(a) An Euler path EP of tree T

(b) Successor array representation of EP

Figure 5.3: Example of Euler path of a tree and the successor array representation

This Euler path can be constructed easily in parallel using the following relation for all edges e in parallel:

$$EP[e] = (EL[e].sib).next \tag{5.1}$$

Now that an Euler path is constructed for a given tree T, the ranks for each edge in the Euler path can now be obtained - this will be done using the efficient list ranking algorithm that we have presented in Chapter 4. Most of computations that we will cover in this chapter involve some manipulation of these ranks for each edge.

5.1.2 Constructing a Euler Tour on the GPU

In order to minimize global memory overhead, reduce the strided access across a warp and to efficiently map the algorithm on to the GPU, we have used a three level data structure. Specifically, we used the following arrays:

1. A Start Array S which contains the start index of the incident edge list for each vertex v of T
2. The Incident Edge List AL. The incident edge list contains the information regarding edges incident on a given vertex. It contains pointers to edge list EL. The anti-parallel edges are placed in the list such that the incoming edge to a vertex is placed first and in the next array element, the outgoing edge.
3. The Edge List EL, contains the end points of each directed edge. Each edge has its corresponding anti parallel edge duplicated in its adjacent array element.

The number of read accesses for end points of an edge is less than that for the index of the edges in ETT, hence such a format is used. The three arrays are given as input to the program. We must first construct the required eulerian graph by replacing each edge in the original tree T with a set of anti-parallel arcs. We then construct an Euler path contained inside a successor array M. This is done with the help of GPU kernels (Algorithms 8 and 9). Following which M contains an Euler tour of graph T. The work is partitioned into two kernels to reduce effect of thread divergence on performance. Algorithm 8 maps each incoming edge in a vertex to an outgoing edge. We allow the incoming edge to map itself to an outgoing edge of another vertex, this anomaly is rectified in the next kernel. Algorithm 9 maps the last incoming edge in the incident edge list to the first outgoing edge in the list.

Algorithm 8 Mapping Kernel 1 for ETT construction on the GPU

Input: Array EL containing edge list information
Output: Array M with the mapped information.

1: **for** each thread in kernel **do in parallel**
2: Get thread id and store in $index$
3: Set $i = (2 \times index) + 1$
4: Set $j = i + 1$
5: Set $M[EL[i]] = EL[j]$
6: **end for**

Once the array M is obtained, we send the array as input to our optimal RHJ list ranking algorithm. Following which we obtain a rank for each edge in the Euler tour. This is sent for further processing as described in the next section for various tree computations.

5.2 Tree Computations using ETT

Once the Euler path is constructed using the mapping array, we can perform various operations on the path and it's ranks to obtain various tree characterization. We shall discuss a few of them in this section, along with their implementations on the GPU and results.

Algorithm 9 Mapping Kernel 2 for ETT construction on the GPU

Input: Arrays S, EL containing adjacency and edge list information
Output: Array M with the mapped information.

1: **for** each thread in kernel **do in parallel**
2: Get thread id and store in $index$
3: Set $i = S[index + 1] - 1$
4: Set $j = S[index]$
5: Set $M[EL[i]] = EL[j]$
6: **if** this is the first thread **then**
7: Set $M[EL[S[0] + 1]] =$ end of list
8: **end if**
9: **end for**

5.2.1 Tree Rooting and Subtree Size

Once the edges have been mapped to an Euler path and it has been ranked using the list rank kernel, we can determine the parent-successor relationship for each set of adjacent vertices by a few simple calculations that can be carried out for each node in parallel.

Once we have an Euler path in M and the corresponding ranks for each edge R, we can mark each edge as either *forward* or *retreat* edge. All we need to do is compare the rank of an edge with it's sibling. The edge with lower rank a forward edge, and the other is a retreat edge. This can be done independently for each edge and in parallel. The algorithm is formally stated in Algorithm 10.

Algorithm 10 Parent finding Algorithm

Input: Arrays S, AL, EL containing adjacency and edge list information, array M containing the Euler path and array R containing the ranks of M
Output: Array Par with parent of each vertex $v \in V$ and Dir containing the direction of each edge $xy \in E$.

1: **for** each edge $xy \in EL$ **do in parallel**
2: **if** $R[xy] < R[yx]$ **then**
3: Set $Dir[xy] = F$ and $Dir[yx] = R$
4: Set $Par[y] = x$
5: **end if**
6: **end for**
7: Set $Par[root] = NULL$

The size of a subtree rooted in node x is the number of children (descendants) of x. If x is a leaf node, then this is 1. Once the parents of each node have been determined, the difference between the rank of a node and it's parent is double the size of the subtree of that node. Hence, it can be easily calculated (Algorithm 11).

5.2.2 Tree Traversals

We can perform post and preorder traversals on a tree T quickly once we have an Euler path P and the corresponding ranks of each edge and the parent-successor relationship for each node. The preorder number of each node is the number of forward arcs in the Euler tour from the root to that node. To compute this efficiently in parallel, we first construct a weighted array W such that

Algorithm 11 Subtree Size Algorithm

Input: Arrays S, AL, EL containing adjacency and edge list information, array M containing the Euler path , array R containing the ranks of M and array Par with parent of each vertex $v \in V$ and Dir containing the direction of each edge $xy \in E$.

Output: Array $Subtree$ with size of subtree for each vertex $v \in V$

1: **for** each vertex $i \neq root \in V$ **do in parallel**
2: Set $Subtree[i] = (Rank[ij] - Rank[ji] + 1)/2$ where $j = Par[i]$
3: **end for**
4: Set $Subtree[root] = n$

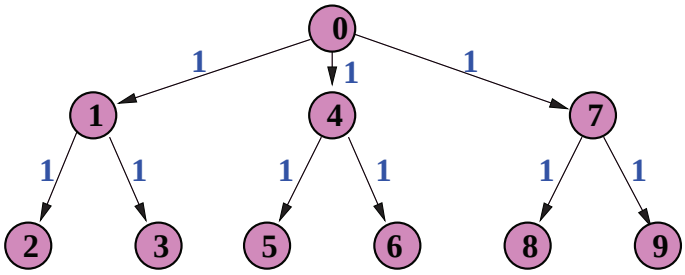


Figure 5.4: Preorder numbering by a parallel scan on the forward flags

a forward edge i gets value $F[i] = 1$ and a backward edge gets $W[i] = 0$. A parallel scan can now be performed on this weighted array, and all the forward arcs will receive a pre-order number. The process is illustrated in Figure 5.4. The algorithm for preorder traversal is descibed in Algorithm 12.

5.2.3 Vertex Levels

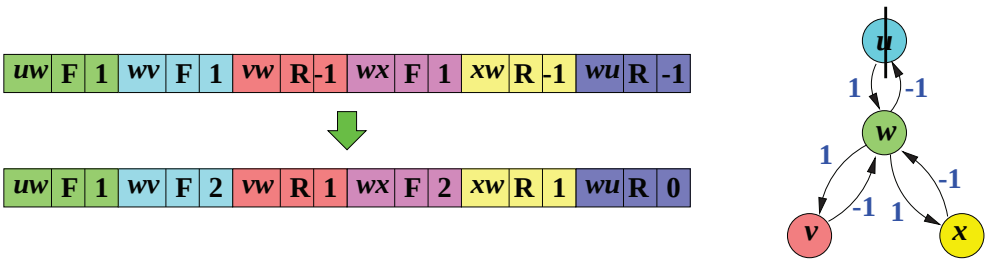


Figure 5.5: Calculating the node levels for each node in a tree

Similar to the traversal, by assigning an appropriate weight function W to each edge $xy \in E$, we can obtain the level for each node. For each forward edge we assign a weight of $+1$ and for each reverse edge we assign a weight of -1 to W. Applying the parallel scan on this algorithm, we get the correct level for each node in the tree. The process is illustrated in Figure 5.5 and the algorithm is given in Algorithm 13.

Algorithm 12 Preorder Traversal by ETT

Input: Arrays S, AL, EL containing adjacency and edge list information, array Dir containing the direction of each edge

Output: Array Pre with the preorder number of each edge.

1: **for** each edge $e \in EL$ **do in parallel**
2: **if** $Dir[e] = F$ **then**
3: Set $W[e] = 1$
4: **else**$W[e] = 0$
5: **end if**
6: **end for**
7: Perform Parallel Scan on W
8: **for** each edge $e \in EL$ **do in parallel**
9: **if** $Dir[e] = F$ **then**
10: Set $Pre[e] = 1$
11: **end if**
12: **end for**

Algorithm 13 Vertex Levels by ETT

Input: Arrays S, AL, EL containing adjacency and edge list information, array Dir containing the direction of each edge

Output: Array L with the level of each vertex $v \in V$

1: **for** each edge $xy \in EL$ **do in parallel**
2: **if** $Dir[xy] = F$ **then**
3: Set $W[xy] = 1$
4: **else**$W[xy] = -1$
5: **end if**
6: **end for**
7: Perform Parallel Scan on W
8: **for** each edge $xy \in EL$ **do in parallel**
9: **if** $Dir[xy] = F$ **then**
10: Set $L[xy] = W[xy]$
11: **end if**
12: **end for**
13: Set $L[root] = 0$

5.3 Implementation and Results on GPU

The various implementations discussed so far were tested on a PC with Intel Core 2 Quad Q6600 at 2.4 GHz, 2 GB RAM and a NVIDIA GTX 280 with 1 GB of on board Graphics RAM. The host was running Fedora Core 9, with NVIDIA Graphics Driver 177.67, and CUDA SDK/Toolkit version 2.0. These algorithms are also compared to the standard BFS sequential algorithm running on the same PC hereby referred to as the CPU BFS Algorithm.

We have implemented the program for the GPU as follows (Figure 5.6). We first load the tree onto GPU memory (the requisite arrays as described in section 5.1.2). We then map the edges onto an Euler path using the two mapping kernels. We then perform list ranking algorithm on this Euler path and then use a GPU kernel to perform the operations required to find the parent of each node. Additionally we may calculate subtree size or create the weighted array of each vertex as required. To calculate vertex levels or to perform a traversal, we need to do a scan on the weighted array. We use the optimized CUDPP scan [1] for this purpose. Finally we launch a corresponding kernel to assign the preorder numbers or vertex level to each vertex as required.

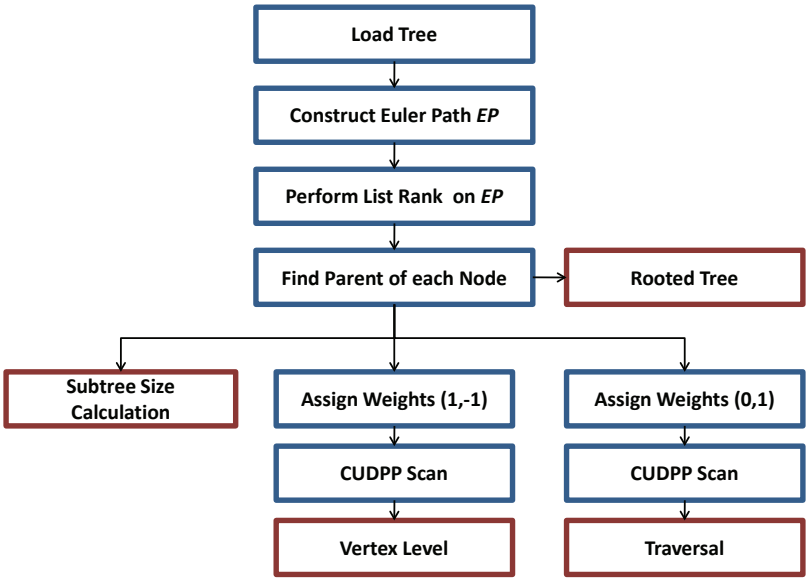
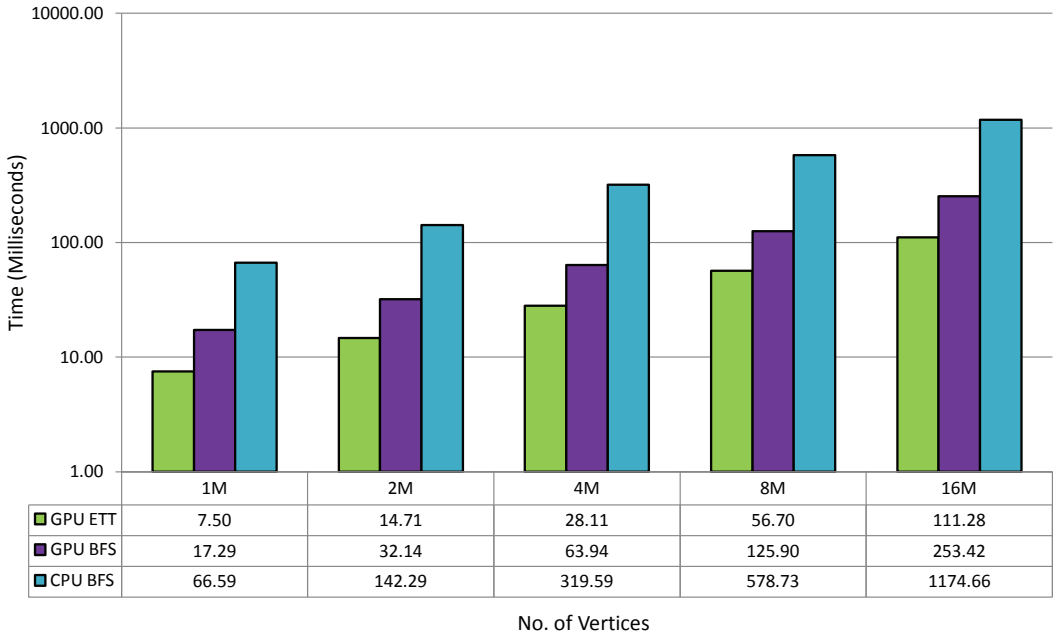


Figure 5.6: Operations performed on the GPU for various ETT algorithms

Results on Random Binary Trees: Figure 5.7 shows the runtime of our ETT construction kernel along with the tree rooting (parent finding) and subtree calculation kernels. Figure 5.8 shows the breakup of runtime for each step while rooting a tree and calculating sublist size using our approach on random binary trees (generated using the random split method) from 1 Million to 16 Million vertices. It can be clearly seen from the timing breakup that list ranking dominates the runtime. Also we see a sustained performance benefit of about 9-11x over sequential BFS and about 2x over parallel BFS on GPU [4].

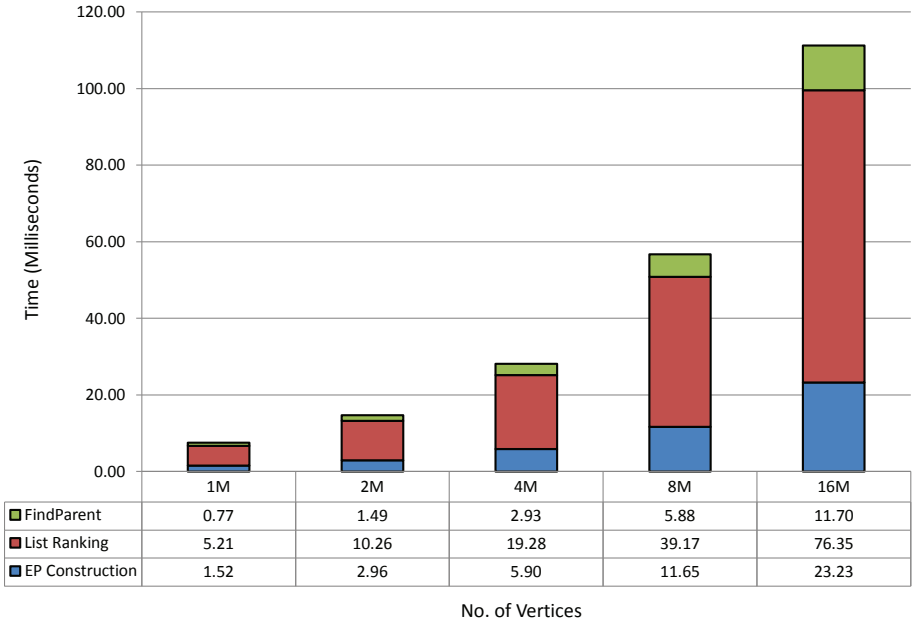
In Figure 5.9, we further extend the algorithm to perform a preorder tree traversal. We have modified the `findparent` kernel to construct the weighted array after the parents are discovered.



	1M	2M	4M	8M	16M
GPU ETT	7.50	14.71	28.11	56.70	111.28
GPU BFS	17.29	32.14	63.94	125.90	253.42
CPU BFS	66.59	142.29	319.59	578.73	1174.66

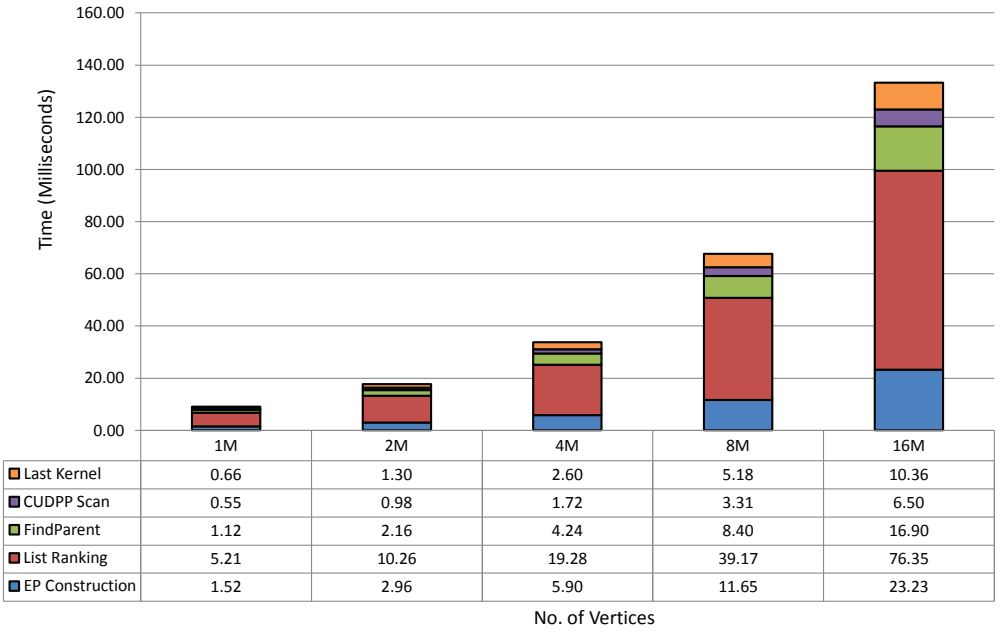
Figure 5.7: ETT-based tree rooting and sublist size calculation (GPU ETT) on GTX 280 GPU compared to sequential BFS on CPU and parallel BFS on GPU [4] on random binary trees

Hence the time taken for the FindParent component has increased from the previous implementation. Our implementation can complete a preorder traversal of a 16 million node random binary tree in about 133 milliseconds.



	1M	2M	4M	8M	16M
FindParent	0.77	1.49	2.93	5.88	11.70
List Ranking	5.21	10.26	19.28	39.17	76.35
EP Construction	1.52	2.96	5.90	11.65	23.23

Figure 5.8: Timing profile of ETT-based tree rooting and sublist size calculation on GTX 280 GPU on random binary trees



	1M	2M	4M	8M	16M
Last Kernel	0.66	1.30	2.60	5.18	10.36
CUDPP Scan	0.55	0.98	1.72	3.31	6.50
FindParent	1.12	2.16	4.24	8.40	16.90
List Ranking	5.21	10.26	19.28	39.17	76.35
EP Construction	1.52	2.96	5.90	11.65	23.23

Figure 5.9: Timing profile of ETT-based preorder traversal on GTX 280 GPU for random binary trees

Chapter 6

Performance Prediction for Irregular Applications on GPU

In light of a larger performance model co-developed by the author and described in the following section, this chapter will focus on performance prediction of algorithms which have a significant amount of irregular access portions. We first discuss the performance model in detail, and then focus on global memory coalescing on the GTX280. We then apply the performance model to our RHJ list ranking kernel to predict the runtime and end the chapter with a few notes on optimizing such irregular applications on the GPU.

6.1 A Model for the CUDA GPGPU Platform

A performance prediction model that takes elements from the BSP, PRAM, and QRQW models was developed with inputs from the author. This model will be introduced in this section, and later in the chapter, we shall discuss how the list ranking algorithm can be modeled and how the performance can be predicted using this model.

Let us assume that we want to model the execution time of a program P on the GPU. The BSP model allows us to look at time as the sum of the times across various kernels. Thus, given a CUDA program P with r kernels $K_1, K_2, \cdots, K_r$, the time taken is $\sum_{i=1}^{r} T(K_i)$ sec. where $T(K_i)$ gives the time taken by kernel K_i. Thus, we have:

$$T(P) = \sum_{i=1}^{r} T(K_i) \text{ sec.} \tag{6.1}$$

For a kernel K, we now have to consider the GPU execution model. Recall that *blocks* are assigned to SMs and each block consists of N_w *warps*. Each warp consists of N_b threads and threads in each warp are executed in parallel. Though it is possible that each SM gets blocks in a batch of up to 8 blocks so as to hide idle times, this is equivalent to having all blocks execute in a serial order for the purposes of estimating the runtime. One has to finally take care of the fact that each of the N_c cores (or SPs) in an SM on the GPU has a D-deep pipeline that has the effect of executing D threads in parallel.

In addition, it is also required to estimate the cycle requirement of a single thread. Compute cycles can be estimated easily using published data from NVIDIA[6]. Global memory and shared memory cycles are estimated by taking into account the existance of coalescing (will be described in detail in Section 6.2), and bank conflicts. We take the approach that the number of cycles required

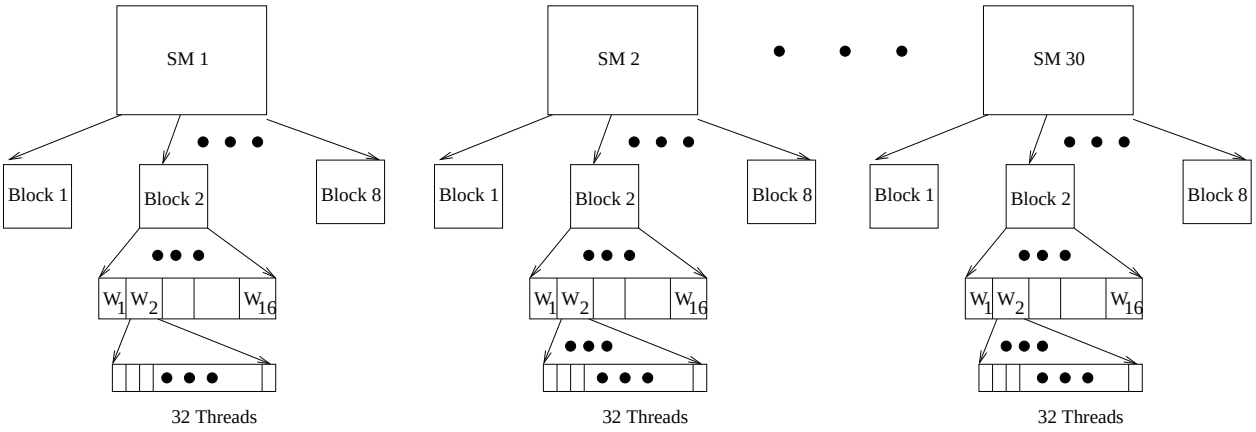


Figure 6.1: The threads in a warp are executed in parallel. Groups of warps are arranged into block and block are assigned to SMs.

by a kernel is the maximum required by some thread in that kernel. Let the maximum number of cycles required by any thread executing the kernel K be $C_T(K)$. Thus, $C_T(K)$ can be expressed as the maximum over all $C(T)$ for T a thread executing the kernel K. Therefore,

$$C_T(K) = \max_T C(T). \tag{6.2}$$

Notice that if we are using the MAX (SUM) model, then the $C_T(K)$ term in the above should be obtained using the MAX (*resp.* SUM) model .

Finally, the time taken for executing kernel K is estimated as follows. Let $N_B(K)$ be the number of blocks assigned to each SM in sequence in the kernel K , $N_w(K)$ be the number of warps in each block in the kernel K, $N_t(K)$ be the number of threads in a warp in the kernel K. Then, the number of cycles required for kernel K, denoted $C(K)$ is:

$$C(K) = N_B(K) \cdot N_w(K) \cdot N_t(K) \cdot C_T(K) \cdot \frac{1}{N_C \cdot D} \tag{6.3}$$

To get the time taken, we have to multiply Equation (6.3) by the clock rate of the GPU as in the equation below, where R is the clock rate of a GPU core.

$$T(K) = \frac{C(K)}{R} \tag{6.4}$$

Since it is possible to have a different structure on the number of blocks, number of warps per block etc., in each kernel, we parameterize these quantities according to the kernel.

To illustrate Equations (6.3, 6.4), Figure 6.1 is useful. Each of the SMs in the GPU get multiple blocks of a kernel. In the picture we consider $N_B = 8$. Each of these blocks are executed on the SM by considering each block as a set of N_w warps. Each warp is then treated as a set of N_t threads. It is these threads that are essentially executed in parallel on the 8 cores of the SM. In Figure 6.1, we have used $N_w = 16$ and $N_t = 32$.

Unlike sequential computation, there is another element that has an impact on the performance of GPU programs. Multiprocessors employ time-sharing as a latency hiding technique. Within the context of the GPU, this time-sharing is in the form of each SM handling more than one block of threads at the same time. To model this situation and its effect, let us assume that each SM gets b blocks of threads that it can time-share. Notice that when we use the MAX or the sum model to

Parameter	Definition
N_w	Number of warps per block
N_t	Number of threads per warp = 32
D	Pipeline depth of a core
N_c	Number of cores per SM
K_i	Kernel i on the GPU
$C_t(K)$	Max. number of cycles required by any thread in kernel K
R	Clock rate of GPU
$T(K)$	Time taken by a kernel K

Table 6.1: List of parameters in our model

estimate the time taken by a kernel, all the b blocks then require b times the time taken by a single block. The number of blocks assigned sequentially to an SM N_B effectively reduces by a factor of b. So there is no net effect of time sharing as long as latencies are hidden well. So, our Equation (6.4) stands good even in the case of time sharing.

6.2 Global Memory Access on the GPU

In the performance model that was presented in Section 6.1, we may estimate the runtime of a particular CUDA kernel by counting compute and memory cycles and choosing either the MAX or SUM of those values (depending on memory operations being overlapped by compute). In irregular applications such as List Ranking it is usually safe to assume that global memory access will take up a large chunk of the overall cycles required. This is because there is limited use for shared memory in such irregular algorithms, especially algorithms like list ranking where data elements are accessed only one per kernel. This section will explain the GPU global memory subsystem in detail (focusing on the G200 architecture), and will also present techniques to estimate the number of global memory transactions required to service a thread or group of threads in a CUDA kernel.

Memory Bandwidth of the GTX280: The NVIDIA GeForce GTX280 uses DDR RAM with a memory clock of 1,107 MHz and 512-bit wide memory interface. The peak memory bandwidth of the NVIDIA GTX 280 can be calculated as:

$$\frac{\left(1107 \times 10^6 \times (\frac{512}{8} \times 2)\right)}{10^9} = 141.6 \text{ GB/sec} \tag{6.5}$$

Effective bandwidth of a kernel K in GBps can be calculated using the following equation:

$$\text{Effective bandwdith} = \left(\frac{B_r + B_W}{10^9}\right) /time \tag{6.6}$$

Where B_r is the bytes read and B_w are the bytes written by a K. Equation 6.6 can be used by a programmer to assess the potential for using the global memory most effectively. If a kernel K can approach the theoretical 141.6 GB/sec, then there can be no further optimization as we have hit the peak transfer capability of the hardware.

6.2.1 Memory Coalescing

Since reading data from the global memory of the GPU incurs about 200 times more latency than reading data on-chip, ensuring that data is fetched from the GPU in the most efficient manner possible is critical to good performance on the GPU. It is even more important for irregular applications as its usually not possible to use the on-chip shared memory as the memory access pattern cannot be determined beforehand. Hence global memory coalescing becomes the single most important consideration in programming for the CUDA architecture.

Global memory loads and stores by threads of a half-warp (16 threads) are coalesced by the device in as few as one global memory transaction if certain access requirements are met. For the GTX 280, the following protocol is used to determine the number of transactions used by a half-warp of threads

- Find the memory segment that contains the address request by the lowest numbered active thread. Segment size is 32 bytes for 8 bit data, 64 bytes for 16-bit data and 128 bytes for 32-, 64-, and 128-bit data.
  - If the transaction is 128 bytes and only the upper or lower half of the segment is used, then transaction size is reduced to 64 bytes
  - If the transaction is 64 bytes and only the lower or upper half is used, reduce the transaction size to 32 bytes
- Carry out the transaction and mark the serviced threads as inactive
- Repeat until all the threads in the half-warp are serviced.

So it is apparent that certain types of memory accesses are bound to be good on the GPU and others will progressively deteriorate as the memory addresses accesed by threads of a half-warp diverge. We shall now explore the two parameters that dictate the effect of coalesced memory access on the GPU.

Effect of Misaligned Access Pattern: Using a simple kernel outlined in Listing 6.1, we can see the effect of a misaligned access pattern using an offset.

Listing 6.1: A copy kernel that illustrates misaligned accesses

```
__global__ void offsetCopy(float *odata, float *idata, int offset)
{
  int xid = blockIdx.x * blockDim.x + threadIdx.x + offset;
  odata[xid] = idata[xid];
}
```

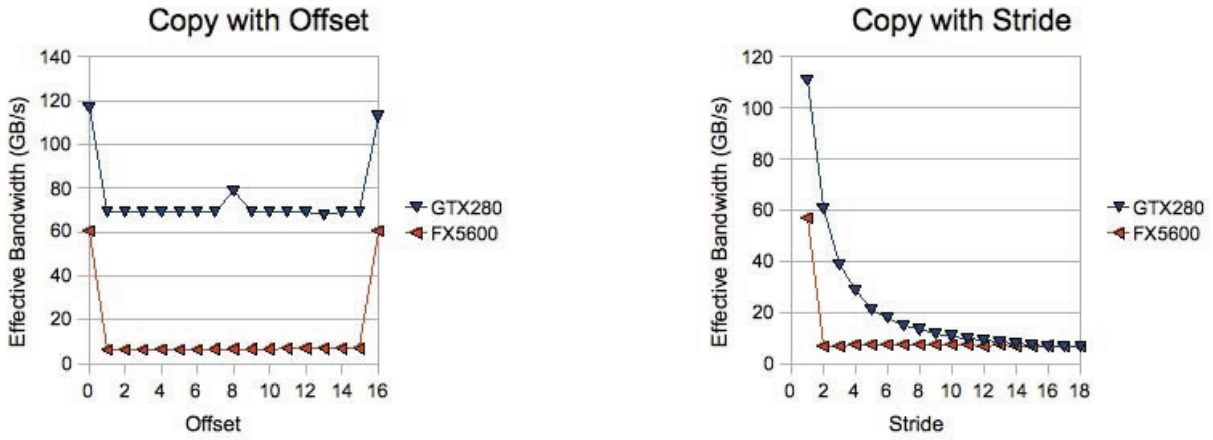
In the code above, data is copied from `idata` to output array, both of which are on global memory. Varying the `offset` variable from 1 to 32, we can see the effect that it has on global memory bandwidth in Figure 6.2(a). In all cases for the GTX280 GPU, the misaligned access can be managed in either one or two transactions. For a `offset` or 0 or 16, the accesses are aligned within a 64-byte boundary and thus results in a single 64-byte transaction. For an `offset` of 8, two 32-byte transactions are performed. For all other offsets two transactions are required, which results in a near halving of the bandwidth.

Effect of Strided Access Pattern: If the threads in a half warp have non-unit strides, then the performance can degrade as the stride increases. In Listing 6.2, we use a variable `stride` in the kernel `strideCopy` to specify the stride across all the elements in a half-warp.

Listing 6.2: A copy kernel that illustrates non-unit stride accesses

```
__global__ void strideCopy(float *odata, float* idata, int stride)
{
  int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
  odata[xid] = idata[xid];
}
```

For example, a stride of 2 results in a single transaction but half the elements remain unused as compared to stride of 1. Likewise, as the stride increases more and more transactions will have to be performed to service all the requests of a half warp, which in the worst case, leads to 16 times slowdown per half warp. Figure 6.2(b) shows the effect of gradually increasing the stride on effective bandwidth.



(a) Performance of the offsetCopy kernel

(b) Performance of the strideCopy kernel

Figure 6.2: Effect of copy parameters on effective bandwidth of a global memory copy kernel. Courtesy NVIDIA [5]

6.2.2 Memory Accesses for Irregular Applications

As discussed before, the inherent nature of irregular applications means that there is little or no data locality that can be exploited. This means that optimization techniques such as shared memory usage, texture caches etc. will have little or no effect on the application.

Consider the following code section from Wyllie's algorithm (Listing 6.3):

Listing 6.3: A portion of Wyllie's Algorithm

```
int temp1=S(i);
int temp2=S(S(i));
```

In the first statement, each thread of a half-warp reads successive elements of the array `S(i)`. This access is guaranteed to be coalesced according to the protocol described in Section 6.2.1, as

there is no misalignment or stride. In the second statement, however, the pointer reference $S(S(i))$ which, in theory, could lie anywhere in the range of array `S`. In the worst case scenario, this would lie in 16 different memory segments in a half-warp (refer Figure 6.3)

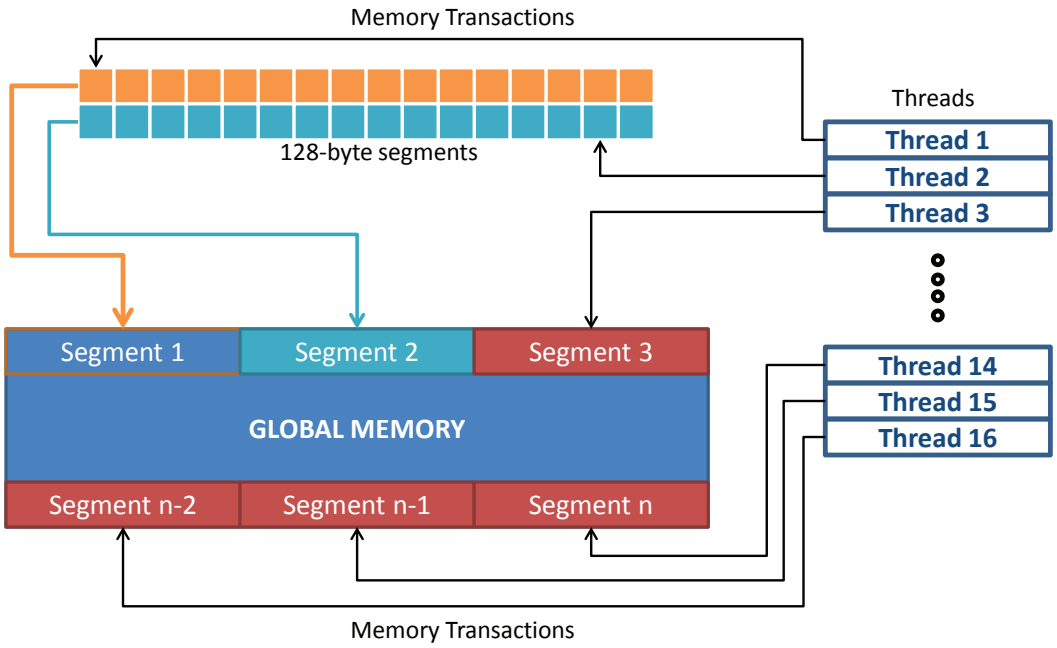


Figure 6.3: Worst-case scenario of 16 threads of a half warp accessing 16 different 128-byte segments in global memory. 16 separate 128-byte memory transactions will be required to service this half-warp.

For exteremly large range of data accessed in random by multiple threads, the probability of threads participating in "accidental" coalescing (also known as the *instantaneous degree of locality*) is very unlikely. Usually it is best to treat them as uncoalesced accesses (which translates to 16 transactions per half warp). We use this assumption in the next section to accurately model the RHJ list ranking algorithm using the performance model discussed in Section 6.1.

6.3 Modeling RHJ List Ranking on the performance Model

Applying the performance model discussed in Section 6.1, the RHJ List Ranking algorithm (section 4.2) is modeled as follows:

With N elements and $p = \frac{N}{\log N}$ splitters, we require $\frac{N}{\log N}$ threads. These threads are grouped into $\frac{N}{512}$ blocks of 512 threads each. Of these at most $N_B = \lceil \frac{N}{512 \cdot 30} \rceil$ blocks are assigned to any single SM on the GPU. Each of these blocks consists of $N_w = 16$ warps of $N_t = 32$ threads each.

Since we are working with a random list, some threads process more work than others. Typically, the most likely size of the sublist was observed to be $4 \log N$ elements, which also confirms to known results on probability. The memory cycles taken by a thread can be computed as follows. Each thread involves 3 reads/writes to the global memory for each element that this thread is traversing. All these accesses are non-coalesced. So with about $4 \log N$ elements per sublist, $N_{\text{memory}} = 4 \log N \cdot 3 \cdot 500$. The compute in each thread is very minimal. So we ignore this completely and set $C_T(K) = N_{\text{memory}}$.

List Size	No of Blocks	Blocks per SM	Compute Cycles	Memory Cycles	Expected Runtime	Actual Runtime
256K	28	1	288	108000	1.333	0.773
512K	54	2	304	114000	2.814	1.909
1M	102	3	320	120000	4.443	4.404
2M	195	7	336	126000	10.884	9.588
4M	372	13	352	132000	21.176	19.958
8M	712	25	368	138000	42.575	40.806
16M	1365	47	384	144000	83.521	82.542

Table 6.2: Calculations of expected runtime (calculated using Equation 6.4) and actual runtime by the local ranking phase of the RHJ list ranking algorithm on the GTX 280 using the performance prediction model. Runtimes are in milliseconds.

The overall time taken by the kernel to compute the local ranks for each sublist can then be computed using Equations 6.3 and 6.4 as:

$$\frac{N}{512 \cdot 30} \cdot 16 \cdot \frac{32}{8 \times 4} \cdot 4 \log N \cdot 3 \cdot 500 \cdot \frac{1}{1.3 \times 10^9} \text{ sec.}$$

For $N = 2^{22}$ we get the time per SM to be ≈ 21.0 millisecond. This compares favorably with the actual time for this kernel for $N = 2^{22}$ at 24 millisecond. Figure 6.3 and Table 6.3 show the comparison of the estimate and the actual times over various list sizes, ranging from 256 K to 16 M elements. We note that since the computation in each thread is minimal to the memory access cost, the models MAX and SUM exhibit identical behavior. So we show only the estimates from the MAX model.

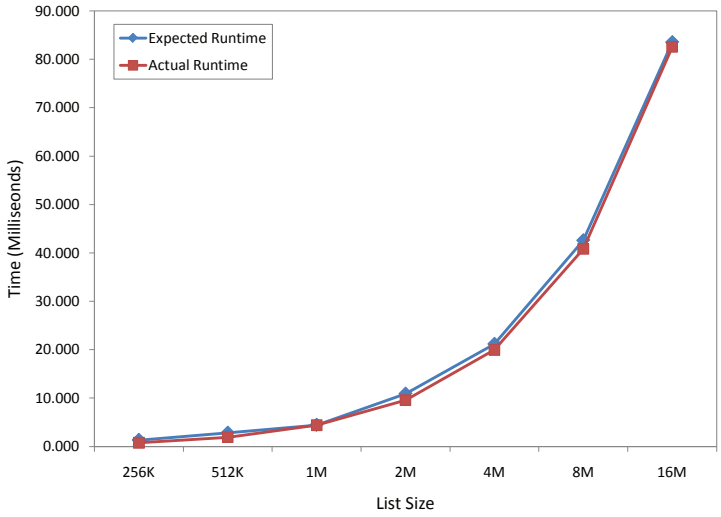


Figure 6.4: The estimated and the actual time taken by the local ranking phase of the list ranking kernel on lists of size varying from 256K elements to 16M elements.

6.4 Strategies for Implementing Irregular Algorithms

In this section, we aim to provide some important pointers for implementing irregular algorithms on massively multi-threaded architectures like the GPU. Some of the lessons learned for implementing irregular algorithms are:

1. Exploiting massive parallelism of the GPU is key to maximizing performance. In case of irregular algorithms, a performance benefit can be seen only if there are enough threads to offset the memory access latencies and the overhead of transferring data to the GPU and launching a kernel. For small input data, a CPU might be much more efficient as the data might fit onto the CPU cache and thereby give good performance on a sequential algorithm.

2. Over-reliance on expensive synchronization should be avoided. Use of atomic operations might be required in applications that have a lot of irregular access, but it usually does not give good performance. It would be wise to find out if there is any way of breaking down the problem into independent subsets which can be worked upon using individual threads or blocks.

3. Some effort in improving load-balancing through probabilistic means of dividing the work among various threads may be helpful in improving performance. As shown using the RHJ list ranking implementation, it would be wise to improve load balancing and try to give each thread enough useful work in order to improve efficiency.

   A key optimization for CUDA kernels is the use of Shared memory. However, in the context of irregular algorithms this is unlikely to be helpful as we cannot accurately predict and pre-fetch required data from main memory to the shared memory cache. With larger caches and support for automated caching in future GPU architectures, one may hope to see improved performance for such classes of algorithms.

Chapter 7

Conclusions and Future Work

In this thesis, we presented a few implementations of the pointer jumping and list ranking primitives, which are irregular algorithms used frequently in various graph and tree computations on parallel architecture. An efficient implementation of pointer jumping in the form of Wyllie's algorithm was gradually developed using multiple optimization techniques. This implementation can be used to deploy other algorithms that require list processing in parallel like the Shilloach-Vishkin connected components algorithm. We were able to get about 2x speedup over the original implementation through reduced operations optimizations on the GPU.

The pointer jumping primitive, while useful for various list processing computations, is work-suboptimal for list ranking and is also inefficient in its use of global memory. We explored a more recent algorithm by Helman and JáJá which works on independent sublists of the original list. This algorithm was modified for the GPU and we recursively reduce the list down to a size which could be efficiently ranked sequentially, and the results was recombined. The implementation was about 3-4x faster than the most efficient implementation of Wyllie's algorithm and is also the fastest single-chip parallel implementation of list ranking that we are aware of.

We then presented a few implementations of list ranking using an efficient Euler tour construction technique (ETT) on the GPU to complete a few tree characterization problems. Our implementations were about 9-11x faster than sequential BFS on the CPU. There are many other applications of list ranking and ETT which can now be implemented on the GPU such as bi-connected components, least-common ancestor problems as well as expression evaluation and ear decomposition.

Optimized implementations of pointer jumping, RHJ list ranking and ETT have been released for general use. We hope that these primitives will come handy for application developers and researchers for processing irregular data such as graphs and trees and also serve as examples for other irregular algorithm implementations on the GPU.

Finally, in light of a performance model for GPUs developed recently, we also discussed the implications of irregular algorithm memory accesses on the GPU architecture and accurately predicted the runtime of the RHJ list ranking implementation on the GPU. We also discussed some of the strategies that we've discovered to implement such irregular algorithms on the GPU. We hope that it will be much easier to deploy such algorithms on the next generation of GPUs such as NVIDIA's Fermi, which has support for automated L1 and L2 caches, possibly providing more efficient accesses to global memory.

Related Publications

- **M. Suhail Rehman**, Kishore Kothapalli, P. J. Narayanan, "Fast and Scalable List Ranking on the GPU", in *Proceedings of the 23rd International Conference on Supercomputing (ICS)*, June 2009, pp. 235–243.
- Kishore Kothapalli, Rishabh Mukherjee, **M. Suhail Rehman**, Suryakant Patidar, P. J. Narayanan, Kannan S. "A Performance Prediction Model for the CUDA GPGPU Platform", to appear in *16th International Conference on High Performance Computing (HiPC)*, December 2009.

Bibliography

[1] M. Harris, J. Owens, S. Sengupta, Y. Zhang, and A. Davidson. "CUDPP: CUDA Data Parallel Primitives Library,",". `http://gpgpu.org/developer/cudpp`.

[2] D. Bader, G. Cong, and J. Feo, "On the Architectural Requirements for Efficient Execution of Graph Algorithms," *International Conference on Parallel Processing (ICPP), 2005*, pp. 547–556, June 2005.

[3] D. A. Bader, V. Agarwal, and K. Madduri, "On the Design and Analysis of Irregular Algorithms on the Cell Processor: A Case Study of List Ranking," in *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, IEEE, 2007, pp. 1–10.

[4] P. Harish and P. Narayanan, "Accelerating Large Graph Algorithms on the GPU Using CUDA," in *High Performance Computing HiPC 2007*, Lecture Notes in Computer Science, Springer, 2007, pp. 197–208.

[5] NVIDIA Corporation, "NVIDIA CUDA C Programming Best Practices Guide," tech. rep., NVIDIA, 2009.

[6] NVIDIA Corporation, "CUDA: Compute Unified Device Architecture Programming Guide," tech. rep., NVIDIA, 2007.

[7] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for gpus: stream computing on graphics hardware," in *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, New York, NY, USA, ACM, 2004, pp. 777–786.

[8] A. Munshi, "OpenCL Specification V1.0," tech. rep., Khronos OpenCL Working Group, 2008.

[9] C. Boyd, "DirectX 11 Compute Shader," in *ACM SIGGRAPH 2008 classes*, 2008.

[10] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, pp. 80–113, Mar. 2007.

[11] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens, "Scan Primitives for GPU Computing," in *GH '07: Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS Symposium on Graphics Hardware*, Switzerland, Eurographics Association, 2007, pp. 97–106.

[12] S. Patidar and P. Narayanan, "Scalable split and gather primitives for the gpu," tech. rep., Tech. Rep. IIIT/TR/2009/99, 2009.

[13] J. JáJá, *An Introduction to Parallel Algorithms*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1992.

[14] R. P. Brent, "The Parallel Evaluation of General Arithmetic Expressions," *J. ACM*, vol. 21, no. 2, pp. 201–206, 1974.

[15] T. Chen, R. Raghavan, J. N. Dale, and E. Iwata, "Cell broadband engine architecture and its first implementation: a performance view," *IBM Journal of Research and Development*, vol. 51, no. 5, pp. 559–572, 2007.

[16] J. C. Wyllie, "The Complexity of Parallel Computations," Ph.D. dissertation, Cornell University, Ithaca, NY, USA, 1979.

[17] D. R. Helman and J. JáJá, "Designing Practical Efficient Algorithms for Symmetric Multiprocessors," in *ALENEX '99: Selected papers from the International Workshop on Algorithm Engineering and Experimentation*, Lecture Notes in Computer Science, Springer-Verlag, 1999, pp. 37–56.

[18] K. J. Barker, K. Davis, A. Hoisie, D. J. Kerbyson, M. Lang, S. Pakin, and J. C. Sancho, "Entering the petaflop era: the architecture and performance of Roadrunner," in *SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, Piscataway, NJ, USA, IEEE Press, 2008, pp. 1–11.

[19] A. L. Beberg, D. L. Ensign, G. Jayachandran, S. Khaliq, and V. S. Pande, "Folding@home: Lessons from eight years of volunteer distributed computing," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, Washington, DC, USA, IEEE Computer Society, 2009, pp. 1–8.

[20] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "Seti@home: an experiment in public-resource computing," *Communications of the ACM*, vol. 45, no. 11, pp. 56–61, 2002.

[21] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, *et al.*, "The Landscape of Parallel Computing Research: A view from Berkeley," tech. rep., University of California, Berkeley: Technical Report UCB-EECS-2006-183, 2006.

[22] L. Dagum and R. Menon, "Openmp: An industry-standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[23] J. Reinders, *Intel threading building blocks: outfitting C++ for multi-core processor parallelism*. O'Reilly Media, Inc., 2007.

[24] D. Shreiner, M. Woo, J. Neider, and T. Davis, *OpenGL programming guide*. Addison-Wesley, 2005.

[25] NVIDIA Corporation. "CUDA Zone,",". `http://www.nvidia.com/cuda`.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue*, vol. 6, no. 2, pp. 40–53, 2008.

[27] A. Ranade, "A Framework for Analyzing Locality and Portability Issues in Parallel Computing," in *Parallel Architectures and Their Efficient Use*, Lecture Notes in Computer Science, Springer, 1993, pp. 185–194.

[28] M. Cosnard, "A Comparison of Parallel Machine Models from the Point of View of Scalability," in *Massively Parallel Computing Systems, 1994., Proceedings of the First International Conference on*, IEEE, May 1994, pp. 258–267.

[29] T. Gautier, J. Roch, and G. Villard, "Regular versus Irregular Problems and Algorithms," in *Proceedings of Workshop on Parallel Algorithms for Irregularly Structured Problems (IRREGULAR 95)*, Lecture Notes on Computer Science, Springer, 1995.

[30] K. Yelick, D. Bonachea, W.-Y. Chen, P. Colella, K. Datta, J. Duell, S. L. Graham, P. Hargrove, P. Hilfinger, P. Husbands, C. Iancu, A. Kamil, R. Nishtala, J. Su, M. Welcome, and T. Wen, "Productivity and performance using partitioned global address space languages," in *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, New York, NY, USA, ACM, 2007, pp. 24–32.

[31] A. Shet, V. Tipparaju, and R. Harrison, "Asynchronous Programming in UPC: A Case Study and Potential for Improvement," in *First Workshop on Asynchrony in PGAS languages - 23rd International Conference in Supercomputing (ICS)*, ACM, 2009.

[32] R. Cole and U. Vishkin, "Faster Optimal Parallel Prefix Sums and List Ranking," *Information and Computation*, vol. 81, no. 3, pp. 334–352, 1989.

[33] R. Anderson and G. Miller, "Deterministic Parallel List Ranking," *Algorithmica*, vol. 6, no. 1, pp. 859–868, 1991.

[34] M. Reid-Miller, "List Ranking and List Scan on the Cray C-90," in *SPAA '94: Proceedings of the Sixth Annual ACM Symposium on Parallel Algorithms and Architectures*, New York, NY, USA, ACM, 1994, pp. 104–113.

[35] J. Sibeyn, "Minimizing Global Communication in Parallel List Ranking," in *Euro-Par Parallel Processing*, Lecture Notes in Computer Science, Springer, 2003, pp. 894–902.

[36] R. J. Anderson and G. L. Miller, "A Simple Randomized Parallel Algorithm for List-Ranking," *Information Processing Letters*, vol. 33, no. 5, pp. 269–273, 1990.

[37] G. L. Miller and J. H. Reif, "Parallel Tree Contraction and its Application," *26th Annual IEEE Symposium on Foundations of Computer Science, 1984*, pp. 478–489, Oct. 1985.

[38] D. A. Bader, S. Sreshta, and N. R. Weisse-Bernstein, "Evaluating Arithmetic Expressions Using Tree Contraction: A Fast and Scalable Parallel Implementation for Symmetric Multiprocessors (SMPs)," in *HiPC '02: Proceedings of the 9th International Conference on High Performance Computing*, Lecture Notes in Computer Science, London, UK, Springer-Verlag, 2002, pp. 63–78.

[39] D. A. Bader and G. Cong, "A Fast, Parallel Spanning Tree Algorithm for Symmetric Multiprocessors (SMPs)," *Journal of Parallel and Distributed Computing*, vol. 65, no. 9, pp. 994 – 1006, 2005.

[40] R. E. Tarjan and U. Vishkin, "An efficient parallel biconnectivity algorithm," *SIAM Journal on Computing*, vol. 14, no. 4, pp. 862–874, 1985.

[41] R. Brent, "The parallel evaluation of general arithmetic expressions," *Journal of the Association for Computing Machinery*, vol. 21, no. 2, pp. 201–206, 1974.

[42] S. Kosaraju and A. Delcher, "Optimal parallel evaluation of tree-structured computations by raking," in *Proceedings of the 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures*, Springer-Verlag London, UK, 1988, pp. 101–110.

[43] F. Dehne, A. Ferreira, E. Caceres, S. Song, and A. Roncato, "Efficient parallel graph algorithms for coarse-grained multicomputers and BSP," *Algorithmica*, vol. 33, no. 2, pp. 183–200, 2002.

[44] G. Cong and D. Bader, "The Euler tour technique and parallel rooted spanning tree," in *Parallel Processing, 2004. ICPP 2004. International Conference on*, 2004, pp. 448–457.

[45] S. Fortune and J. Wyllie, "Parallelism in Random Access Machines," in *Proceedings of 10th Annual ACM Symposium on Theory of Computing (STOC)*, ACM New York, NY, USA, 1978, pp. 114–118.

[46] D. Culler, R. Karp, D. Patterson, K. E. S. A. Sahay, E. Santos, R. Subramonian, and T. von Eicken, "LogP: Towards a Realistic Model of Parallel Computation," in *Proceedings of ACM SIGPLAN Annual Symposium on Principles and Practices of Parallel Programming (PPoPP)*, 1993, pp. 1–12.

[47] P. B. Gibbons, Y. Matias, and V. Ramachandran, "The Queue-Read Queue-Write Asynchronous PRAM model," in *Proceedings of EURO-PAR*, 1996.

[48] P. B. Gibbons, Y. Matias, and V. Ramachandran, "The Queue-Read Queue-Write PRAM Model: Accounting for Contention in Parallel Algorithms," *SIAM Journal of Computation.*, vol. 28, no. 2, pp. 733–769, 1999.

[49] L. G. Valiant, "A Bridging Model for Parallel Computation," *Communications of the ACM*, vol. 33, no. 8, pp. 103 – 111, 1990.

[50] S. Ryoo, C. I. Rodrigues, S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W. W. Hwu, "Program Optimization Space Pruning for a Multithreaded GPU," in *Proceedings of the Sixth Annual IEEE/ACM International Symposium on Code Generation and Opimization*, 2008, pp. 195–204.

[51] D. Schaa and D. Kaeli, "Exploring the Multiple-GPU Design Space," in *IPDPS '09: Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing*, Washington, DC, USA, IEEE Computer Society, 2009, pp. 1–12.

[52] J. Meng and K. Skadron, "Performance Modeling and Automatic Ghost Zone Optimization for Iterative Stencil Loops on GPUs," in *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, New York, NY, USA, ACM, 2009, pp. 256–265.

[53] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-Level and Thread-Level Parallelism Awareness," in *ISCA '09: Proceedings of the 36th Annual International Symposium on Computer Architecture*, New York, NY, USA, ACM, 2009, pp. 152–163.

[54] G. E. Blelloch, "Scans as primitive parallel operations," *IEEE Trans. Comput.*, vol. 38, no. 11, pp. 1526–1538, 1989.

[55] Y. Dotsenko, N. Govindaraju, P. Sloan, C. Boyd, and J. Manferdelli, "Fast Scan Algorithms on Graphics Processors," in *Proceedings of the 22nd Annual International Conference on Supercomputing (ICS)*, ACM New York, NY, USA, 2008, pp. 205–213.