Exploring the Limits of GPUs With Parallel Graph Algorithms

Kumanan Yogaratnam
kyogarat@connect.carleton.ca

A thesis submitted tothe Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Computer Science

Carleton University
Ottawa, Ontario, Canada

December 19, 2010

Your file Votre référence
ISBN: 978-0-494-79594-1
Our file Notre référence
ISBN: 978-0-494-79594-1

Abstract

In this thesis we explore the limits of graphics processors (GPUs) for general purpose parallel computing by studying problems that require highly irregular data access patterns: parallel graph algorithms for list ranking and connected components. Such graph problems represent a worst case scenario for coalescing parallel memory accesses on GPUs which is critical for good GPU performance. We identify and explore the impact of irregular memory access on the CUDA GPU platform, specifically looking at the memory bandwidth impact. Our experimental study also indicates that PRAM algorithms are a good starting point for developing efficient parallel GPU methods, though they require modifications, often non-trivial, to ensure good GPU performance. We identify some of the pitfalls of the CUDA GPU platform, particularly when implementing parallel graph algorithms with irregular memory access, and discuss how to adapt such graph algorithms for parallel GPU computation. We point out that the study of parallel graph algorithms for GPUs is of wider interest for discrete and combinatorial problems in general because many of these problems require similar irregular data access patterns.

Acknowledgements

This thesis would not exist without the support, guidance and encouragement of my thesis supervisor Dr. Frank Dehne, to whom I remain forever grateful. At least as importantly I wish to express heartfelt thanks to my wife Trinh, without whose patience and support my attempt to complete my Master's degree while continuing to hold onto my demanding job simply would not have been successful.

Contents

List of Algorithms

List of Tables

List of Figures

List of Listings

Chapter 1

Introduction

Modern graphics processors (*GPUs*) have evolved into highly parallel and fully programmable architectures. Current many-core GPUs can contain hundreds of processor cores and can have an astounding peak performance of up to 1 TFLOP. However, GPUs are known to be hard to program. Since *coalescing* of parallel memory accesses is a critical requirement for maximum performance on GPUs, problems that require irregular data accesses are known to be particularly challenging. Current general purpose (i e. non-graphics) GPU applications concentrate therefore typically on problems that can be solved using fixed and/or regular data access patterns such as image processing, linear algebra, physics simulation, signal processing and scientific computing (see e.g. [11]). In this thesis, we explore the limits of GPU computing for problems that require irregular data access patterns through an experimental study of parallel *graph algorithms* on GPUs. Here, we consider list ranking and connected component computation. Graph problems represent a worst case scenario for coalescing parallel memory accesses on GPUs and the question of how well parallel graph algorithms can do on GPUs is of wider interest for discrete and combinatorial problems in general because many of these problems require similar irregular data access patterns. We also study how the significant body of scientific literature on PRAM graph algorithms can be leveraged to obtain efficient parallel GPU methods.

1.1 Parallel *Graph* Algorithms on GPUs

In this thesis, we will focus on nVIDIA's unified graphics and computing platform for GPUs known as the *Tesla* architecture framework [12] and associated CUDA programming model [3]. However, our discussion also applies to AMD/ATI's Stream Computing model [2]and in general to GPUs that follow the OpenCL standard [5, 4]. For our experimental study, we used an nVIDIA GeForce 260 with 216 processor cores at 2.1Ghz and 896MB memory.

A schematic diagram of the Tesla unified GPU architecture is shown in Figure 2.5. A Tesla GPU consists of an array of streaming processors (SMs), each with eight processor cores. The nVIDIA GeForce 260 has 27 SMs for a total of 216 processor cores. These cores are connected to 896MB global DRAM memory through an interconnection network. The global DRAM memory is arranged in independent memory partitions. The interconnection network routes the read/write memory requests from the processor cores to the respective memory partitions, and the results back to the cores. Each memory partition has its own queue for memory requests and arbitrates among the incoming read/write requests, seeking to maximize DRAM transfer efficiency by grouping read/write accesses to neighboring memory locations. Memory latency is optimized when parallel read/write operations can be grouped into a minimum number of arrays of contiguous memory locations. GPUs are optimized for streaming data access or fixed pattern data access such as matrix based operations in scientific computing (e.g. parallel BLAs [11]). In addition, their 1 TFLOP peak performance needs to be matched with a massive need for floating point operations such as coordinate transformations in graphics applications or floating point calculations in scientific computing [12]. Parallel graph algorithms have neither of those two properties The destinations of pointers (graph edges) that need to be followed are usually by definition irregular and not known in advance. The most basic parallel graph operation, following multiple links in a graph, creates in general a highly irregular data access pattern. In addition, many graph problems have no need at all for floating point operations. The question of how well parallel graph algorithms can do in such a challenging environment is of wider interest for parallel discrete and combinatorial algorithms in general because many of them have similar properties.

1.2 PRAM vs. GPU

The *PRAM* model is a widely used *theoretical* model for parallel algorithm design which has been studied for several decades, resulting in a rich framework for parallel discrete and combinatorial algorithms including many parallel graph algorithms (see e.g. [16]). A PRAM is defined as a collection of synchronous processors executing in parallel on a single (unbounded) shared memory. PRAMs and GPUs are similar in that modern GPUs support large numbers of parallel threads that work concurrently on a single shared memory. In fact, modern GPUs with 200+ cores *require* large numbers of threads to optimize latency hiding. An nVIDIA GeForce 260 has a hardware thread scheduler that is built to manage tens of thousands and even millions of concurrent threads. The PRAM version most closely related to GPUs is the CRCW-PRAM supporting concurrent reads and concurrent writes. Tesla GPUs support concurrent write requests which are aggregated at each memory partition's

memory request queue (using the "arbitrary" model where one of the writes is executed but it is undetermined which one).[12] In fact, concurrent read/write accesses are very efficient on GPUs because they nicely coalesce.

However, GPU and PRAM differ in some important ways. First, as outlined above, parallel memory requests coming from multiple processor cores of a GPU need to be coalesced into arrays of contiguous memory locations [12]. On a PRAM, any set of parallel memory accesses can be executed in O(1) time, regardless of the access pattern. Second, as mentioned above, the cores of Tesla GPUs are grouped into streaming processors (SMs) consisting of eight processor cores each. The cores within an SM share various components, including the instruction decoder. Therefore, parallel algorithms for GPUs need to operate SIMD style. When parallel threads executed on the same SM (and within the same *warp*, see Chapter 3) encounter a conditional branch such as an IF-THEN-ELSE statement, some threads may want to execute the code associated with the "true" condition and some threads may want to execute the code associated with the "false" condition. Since the shared instruction decoder can only handle one branch at a time, different threads cannot execute different branches concurrently and they have to be executed in sequence, leading to performance degradation. This leads to a need for SIMD style, data parallel programming of GPUs which is more similar to classical vector processor programming Unfortunately, data parallel solutions for highly irregular problems are known to be challenging.

We are particularly interested in how efficient parallel graph algorithms and implementations can be obtained by starting from the respective PRAM algorithms. Which parts of PRAM algorithms can be transferred to GPUs and which parts need to be modified, and how?

1.3 Summary Of Results

Our experimental study on parallel *list ranking* and *connected component* algorithms for GPUs indicates that the PRAM algorithms are a good starting point for developing GPU methods. However, they require non-trivial modifications to ensure good GPU performance. It is critical for the efficiency of GPU methods that parallel data access coalescing is maximized and that the number of conditional branching points in the algorithm is minimized. While the number of parallel threads that can be executed concurrently on a GPU is large, it is still significantly smaller than the number of PRAM processors. It is important for the efficiency of GPU methods to appropriately assign groups of PRAM processors to GPU threads and choose an appropriate data layout such that GPU threads access data in a pattern referred to as *striding* (see Section 3.8 for precise definition). Another important difference between the PRAM and GPUs is that PRAM methods assume zero synchronization over-

head while this is not the case for GPUs. Therefore, the number of actually necessary and implemented synchronization points for the GPU implementation needs to be minimized to ensure good performance. The observed GPU performance for parallel list ranking and connected components appears to be very sensitive to the total work of the underlying PRAM method. GPU performance also appears to be more sensitive to the constants in the time complexity of the algorithm than parallel implementations for standard multi-core CPUs. This is because GPUs support so many more threads than multi-core CPUs, each with much less work than a thread for the corresponding multi-core CPU method, that the constants in the time complexity have a much larger relative impact.

For the list ranking problem, we start with a GPU implementation of the straightforward pointer jumping algorithm for the PRAM, also known as *Wylie's Algorithm* [16, p. 64] [23]. This algorithm does provide some limited speedup but, similar to a preliminary result in [14], requires $O(n \log n)$ work. GPU performance appears to be very sensitive to the total work, and while Cole and Vishkin's *deterministic coin tossing* method [6] provides an optimal PRAM list ranking method with $O(n)$ work, it is so complicated and has such high constant factors that it's performance gain would be negated by its complexity. Many other parallel list ranking algorithms have been studied in the literature (see e.g. [19, 18, 9, 16, 15]). However the parallel random splitter PRAM method proposed by Reid-Miller and then adapted for the Cray C90 [15] appears to be a good starting point for an efficient parallel GPU list ranking method (see Algorithm 4.3) as it ensures linear total work with low overhead. We observed that the parts of our code with irregular access patterns (following the list pointers) were the dominating parts with respect to the run time of the entire method. Minimizing the number of data access for these parts, for example through packing of variables and caching in GPU registers was crucial for performance. Reid-Miller's parallel random splitter method is a *randomized* PRAM algorithm and we observed that the the large number of threads that can be executed concurrently on a GPU is very helpful to efficiently implement such methods. The GPU's hardware scheduler was very helpful to make the implementation surprisingly efficient even when the random selection of splitter elements created considerable fluctuations in sub-list lengths. We also observed an interesting inflection point in the running time curve (as function of list size), where the irregular data access pattern and data access volume starts to push the limits of the GPU's on-chip interconnection network. Michael Garland at nVIDIA [10] has used our code to reproduce this effect on their machines and pointed out that this is the first time they have seen such an inflection point.

For the connected component problem, we implemented a GPU adaptation of Shiloach and Vishkin's CRCW-PRAM algorithm [17] following the same guidelines outlined above. Despite the fact that Shiloach and Vishkin's CRCW-PRAM algorithm requires $O((n +$

$m) \log n)$ work for n vertices and m edges, and the sequential method requires only linear work, our parallel GPU implementation was significantly faster than the sequential implementation on a standard sequential CPU. We also analyzed some interesting performance variations of the GPU algorithm when executed for different types of graphs.

A technical report[7] of our two results was posted on 24 February 2010. Our result for parallel list ranking was independently discovered and published by Wei and JaJa[22] in April 2010. Our results for parallel computation for connected components in a graph was submitted to IPDPS on 1 October 2010.

Chapter 2

Review of Parallel Computation Systems

2.1 Overview

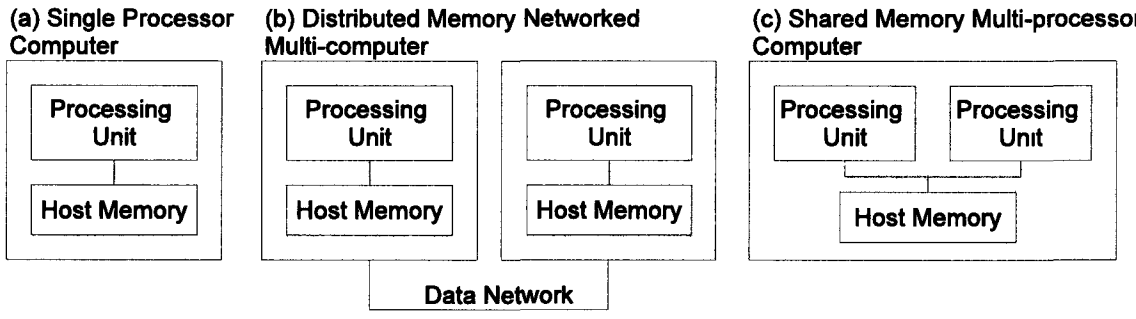


Figure 2.1: Computer Organization Basics.

Parallel computation can be performed in two fundamental ways.

1. Distributed memory systems, where multiple processors each have their own memory and the data they are processing is distributed among them. Examples include cluster computing and grid computing systems; see Figure 2.1 (b).

2. Shared memory systems, where multiple processors have access to a common high-speed memory in which the data they are processing is stored. A common example is the average PC with multiple processors as shown in Figure 2.1 (c), as well as the processors with multiple cores on the same unit as seen in Figure 2.2.

This study is interested primarily in shared memory systems, and particularly with multi-core processors and their use with graph theory algorithms.

2.1.1 Definitions

Information in this chapter was obtained from the following sources: [20, 16, 12].

CPU Central Processing Unit; refers to the processing unit in computers. In this study, the CPU used is a multi-core Intel x86-based processor.

GPU Graphics Processing Unit; refers to the processing unit in today's advanced graphics cards that are added to computers to accelerate graphics performance. When generally speaking, the term is also used to refer to the graphics card itself. In this study, the GPU used in a nVIDIA GTX 260 graphics card.

Core Another term for the processing unit; commonly used to refer to multiple processing units in a multi-core CPU or GPU.

Multi-core Processor A single chip processing unit made up of multiple "cores" or embedded processing units that share the system's main memory.

Thread Refers to a single path of execution. Unlike a process, which has its own distinct memory space, a thread shares the memory space of its parent process; yet otherwise executes with its own program counters and registers.

SISD Single Instruction, Single Data. The traditional computer, as in Figure 2.1 (a), is a SISD machine, with the CPU processing a single instruction stream and operating on a single element of data at a time.

MIMD Multiple Instruction, Multiple Data. A collection of SISD machines together effectively form a MIMD machine, where each CPU processes its own instruction stream and its own data stream, in parallel and independently of the other CPUs. Both multi-core and multi-processor CPU systems (see section 2.1.2), as well as distributed parallel systems like clusters, are fundamentally MIMD systems.

SIMD Single Instruction, Multiple Data. A SIMD machine processes multiple data sets in parallel, executing the same instruction for different data. A common SIMD architecture is the vector processor used by Cray and other super-computer manufacturers.

Data-Parallel An algorithm or computing system is considered data-parallel when it executes the same program in parallel using different data. A SIMD execution is inherently data-parallel; however MIMD systems can be used to execute data-parallel programs as well.

PRAM Parallel Random Access Machine. A widely used theoretical model for parallel computation, a PRAM is a collection of synchronous processors executing in parallel

and communicating via *shared memory*. By definition, each processor is a sequential processor and as a system executes MIMD. There are different PRAM models that vary by how concurrent memory access is resolved.

PRAM-EREW Exclusive Read, Exclusive Write PRAM model, where only a single read or write operation is allowed at any given time, and thus an algorithm needs to ensure that only one processor is reading or writing at any given time.

PRAM-CRCW Concurrent Read, Concurrent Write PRAM model, where concurrent reads and writes are allowed. When multiple processors write to the same location only one of the writes, arbitrarily, will succeed.

PRAM-CREW Concurrent Read, Exclusive Write PRAM model, where concurrent reads are allowed but a algorithm needs to ensure that only one processor is writing at any given time.

PRAM-ERCW Exclusive Read, Concurrent Write PRAM model, where concurrent writes are allowed while only one processor is allowed read memory at any given time. When multiple processors write to the same location only one of the writes, arbitrarily, will succeed.

SIMT Single Instruction, Multiple Thread, a model advanced by nVIDIA as part of their CUDA architecture It describes an execution model where a single instruction processing multiple data is executed by multiple threads, with execution of the threads interleaved over a smaller number of physical processors. (More detail covered in section 2.1.3).

FLOP Floating-point Operation, it is used as a unit of measurement when describing the floating-point processing capacity of a processor.

TFLOP Tera (10^{12}) FLOPs.

2.1.2 Multi-core Multi-processor CPU Systems

Many off-the-shelf CPUs today are made up of multiple cores (or processing units). As shown in Figure 2.2, a multi-core processor is a single processor that is made up of multiple cores which effectively act as a multi-processor computer on a single chip. Each core appears and operates as a fully functional processing unit, and typical operating systems mask the hardware details by mapping each thread created by an application to one of the cores. The current multi-core CPUs have no more than 8 cores each (though the technology is

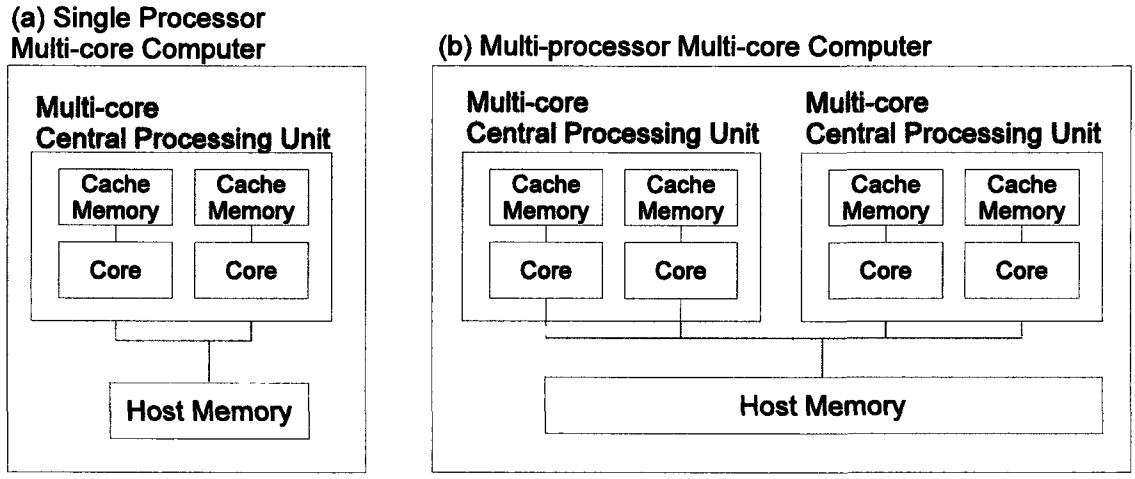


Figure 2.2: Multi-Core Computer Organization Basics

advancing rapidly), and they in turn can be combined by placing multiple CPUs on a single mother-board to provide up to 32 cores in a single system.

A shared memory multi-core multi-processor computer satisfies the very basic definition of the PRAM-CRCW model, i e. multiple processors executing MIMD in parallel and using shared memory with support for concurrent reads and writes. The cores however are not synchronous, as each core can be executing a completely different instruction, where no two instructions necessarily start or end at the same time, at any location in memory at any given time

2.1.3 Multi-core GPU Systems

GPUs are co-processors that are installed within a computer alongside a CPU and main memory, and its purpose is to provide specialized graphics functionality and accelerated performance. Modern GPUs have evolved into high-performance data-parallel multi-core systems, also known as stream processors, that can execute complex data processing algorithms in parallel using thousands, if not millions, of concurrent threads. There are two major vendors of multi-core stream processing GPUs, nVIDIA and ATI, both of whom have graphics cards that can be easily added to a standard Intel PC and accessed via their respective SDKs. A stream computing GPU is typically made up of multiple Streaming Multi-processors (SMs) which are in turn made up of multiple Streaming Processors (SPs) or cores (see Figure 2.3).

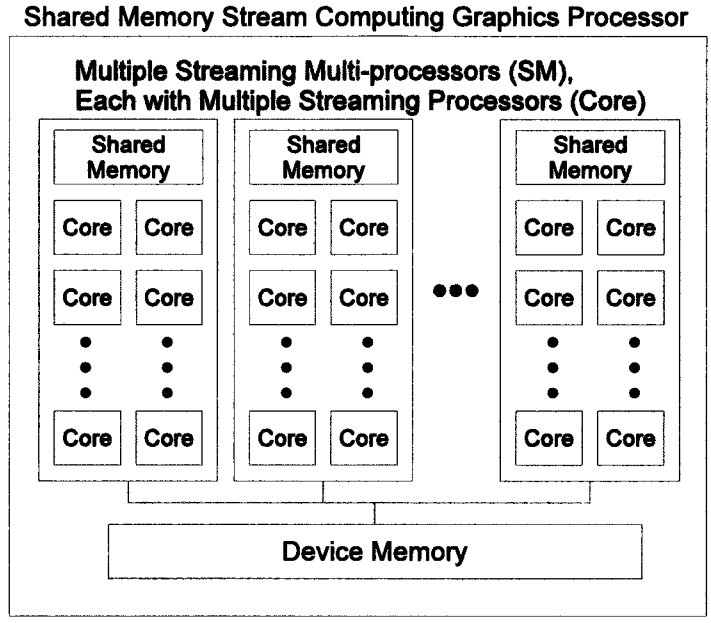


Figure 2.3· Multi-core Graphics Processors

A shared memory multi-core stream computing GPU satisfies some aspects of the definition of the PRAM-CRCW model, i.e. multiple processors executing in parallel and using shared memory with support for concurrent reads and writes. The GPUs are partially synchronous in that within an SM the cores execute SIMD while each SM may be assigned to different work.

Figure 2.4: GPU Interleaving Threads Within Single SM

Programs are written using as many threads as needed, with algorithms allowed and encouraged to use millions of threads. The underlying hardware distributes the threads over multiple SMs such that the threads are divided into blocks (or groups) of threads and each block (group) is assigned to an SM to execute. Typically the number of threads per block (group) will exceed the number of SPs in an SM, which allows the hardware to interleave the instruction execution (see Figure 2.4). Within a single SM, each instruction is executed by all

threads before moving on to the next instruction, a model that has been named SIMT. The interleaving has a desirable side-effect of hiding memory access latency when the instruction being processed is reading or writing memory, since memory access operations typically take longer than the time to execute the instruction itself. By executing more threads than there are cores in an SM, each SM is able to map threads to processor cores in an efficient manner. Each set of threads executes the same instruction on each core in parallel and can be given parallel memory access for the term of the computation. As each thread set finishes, a new set is started on the physical cores until all threads have completed execution. Finally, the SM can move on to next instruction. While the SM does need to make sure that the memory accesses initiated by the first set of threads have finished before moving on, with more threads than cores the time taken with the other threads more often than not ensures that the memory accesses are finished by the time the SM returns to the same set of threads. This is called *latency hiding* and is critical in providing the maximum parallel performance for the system.

Since GPUs are fundamentally graphics accelerators, they can be programmed using a variety of SDKs (e.g. OpenGL, DirectX, PhysX) to take advantage of their various graphics capabilities. As these GPUs have become more and more powerful they have also become more capable of general purpose computing. nVIDIA promoted the first open SDK, called CUDA (Compute Unified Device Architecture) SDK[3] which includes a C/C++ compiler. Later ATI (now AMD) released its own SDK called the ATI Stream SDK [2]. Recently a standard has emerged called OpenCL[5, 4] which provides a common SDK that can be used independently of the underlying GPU hardware.

This study uses the nVIDIA CUDA platform exclusively as its GPU platform. At the time of this work, it was difficult to access ATI and OpenCL development environments as well as compliant hardware. The nVIDIA CUDA platform had been available longer and provided a stable, and well-researched, platform for this study.

2.2 nVIDIA *Tesla* Architecture and *CUDA* Programming Model

Modern graphics processors (*GPU*s) have evolved into highly parallel and fully programmable architectures. The latest GPUs from nVIDIA and AMD contain hundreds of processor cores and have an astounding peak performance of around 1 TFLOP. For this experimental study, the nVIDIA GeForce 260 Graphics Processing Unit (*GPU*) with 216 cores running at 2.1GHz and 896MB of on-board memory was used. The nVIDIA GeForce 260 is based on the *Tesla* architecture [12] shown in Figure 2.5 which is nVIDIA's unified graphics and computing platform. A Tesla GPU consists of an array of streaming processors (SMs), each with eight processor cores. The nVIDIA GeForce 260 has 27 SMs for a total of 216 cores. These cores

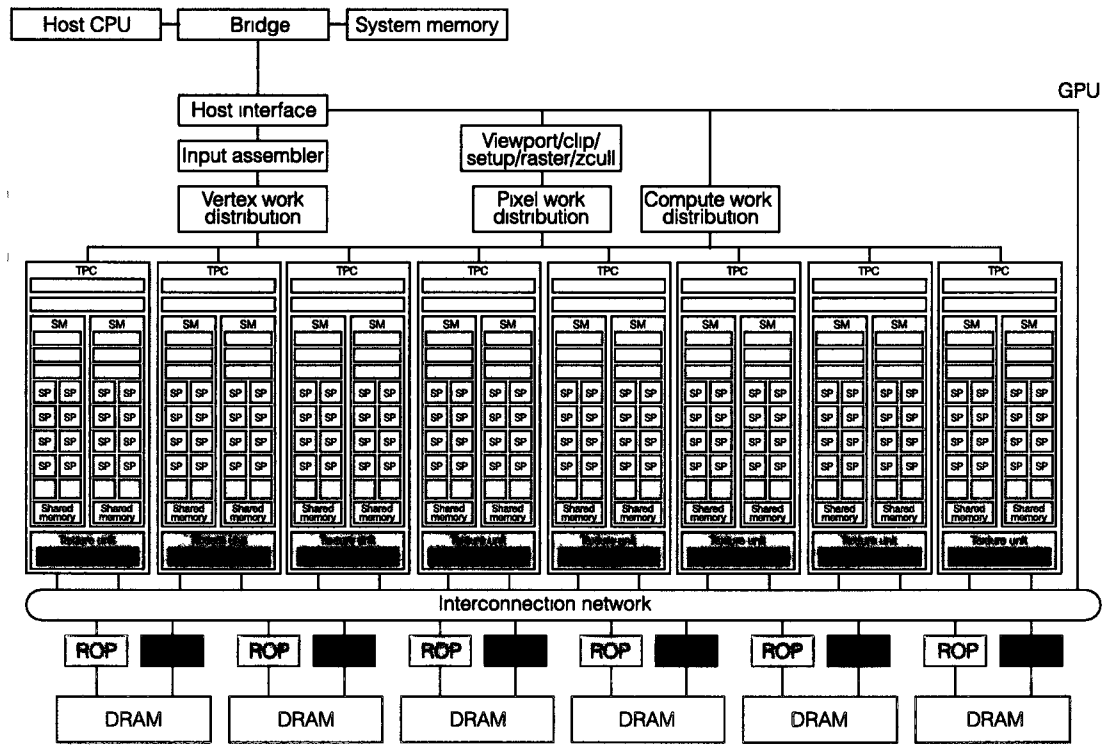


Figure 2.5: nVIDIA Tesla Architecture (from[12])

are connected to the 896MB of on-board memory through an inter-connection network that allows all the SMs to have shared access to the device (aka global) memory. The on-board memory is arranged in independent memory partitions, and the inter-connection network routes the read/write memory requests from the processor cores to the respective memory partitions and back. Each memory partition has its own queue for memory requests[12, p. 49] and arbitrates among the incoming read/write requests, seeking to maximize memory transfer efficiency which favours grouping of read/write accesses to neighboring memory locations. It is important to note that memory latency is lowest with parallel read/write operations that can be grouped into arrays of contiguous memory locations.

Programming the Tesla architecture for general purpose computing is accomplished via the CUDA programming model [1], details of which are explored in detail in Chapter 3. The nVIDIA CUDA GPU platform is made up of multiple SMs, each with multiple (typically 8) cores (SPs). CUDA programs are written using as many threads as needed, with algorithms allowed and encouraged to use millions of threads. The program is aware of the hardware in so much as the CUDA programming model exposes a notion of thread blocks and threads. (See Figure 2.6) Synchronization is available for all threads within a block as well as via

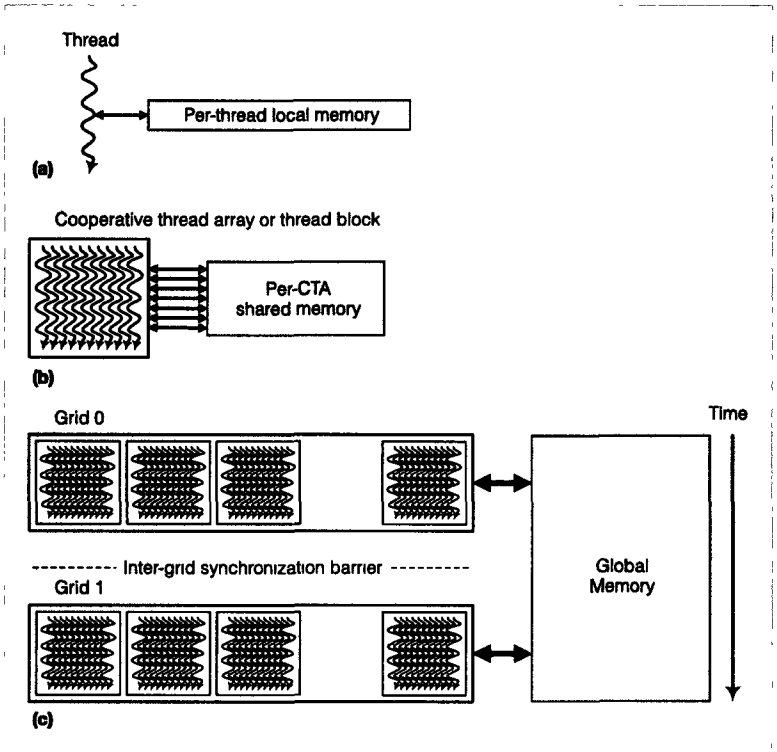


Figure 2.6: CUDA Programming Model (from[12])

barrier synchronization across blocks. The cores themselves execute in SIMD fashion, though that is expensive when code branches occur.

Shared memory access is easy and fast and the latency of memory access is hidden by the multiple threads, though that too can be expensive if not managed carefully. Memory latency further is optimized when parallel read/write operations can be grouped into a minimum number of arrays of contiguous memory locations. GPUs are optimized for streaming data access or fixed-pattern data access such as matrix-based operations in scientific computing.

2.3 Applying PRAM Algorithms to GPUs

PRAMs and GPUs are similar in that modern GPUs support large numbers of parallel threads that work concurrently on a single shared memory. With the large number of cores in modern GPUs, a large number of threads is needed to maximize performance and hide memory access latency. An nVIDIA GeForce 260 has a hardware thread scheduler that is built to manage millions of concurrent threads.

2.3.1 SIMD and Data Parallelism

The cores within a CUDA SM share various components, including the instruction decoder. Therefore, parallel algorithms for GPUs need to operate SIMD style. When parallel threads executing on the same SM (and within the same *warp*, see Section 3.3) encounter a conditional branch such as an IF-THEN-ELSE statement, some threads may want to execute the code associated with the "true" condition and some threads may want to execute the code associated with the "false" condition. Since the shared instruction decoder can only handle one branch at a time, different threads can not execute different branches concurrently and they have to be executed in sequence, leading to performance degradation. This leads to a need for SIMD style, data-parallel programming for GPUs.

CUDA GPUs *require* data parallelism within groups of threads, which makes their programming similar to classical vector processors. It is well known that *efficient* data parallelism is relatively easy to obtain for regular problems where data movements are know *a priori*, such as many matrix operations. Graph algorithms, on the other hand, typically have highly irregular data movements which are not known *a priori*.

2.3.2 Concurrent Memory Access

The PRAM model most closely related to GPUs is the CRCW-PRAM supporting concurrent reads and concurrent writes. nVIDIA CUDA GPUs support concurrent read and write requests which are aggregated at each memory partition's memory request queue; as well concurrent writes to the same location are resolved to ensure one of the writes succeeds. In addition, concurrent read/write accesses are very efficient on GPUs because of the latency hiding achieved by the SMs when they interleave thread execution (see Figure 2.4). The PRAM model thus provides a platform that maps well to the world of CUDA.

- Algorithms that run correctly under the CRCW assumption that one of all concurrent writes to the same location will succeed, can be implemented on a CUDA platform (and in fact any multi-core platform) as long as the algorithm can be modified to use fewer parallel processors (threads) than the number of elements in the data being processed.

- Since the CREW, EREW and ERCW models of PRAM are subsets of the CRCW access model, any PRAM algorithms that do not rely on concurrent writing or reading are also promising candidates for conversion to multi-core implementations; however they may not perform optimally because of the extra work they are likely to do to address the non-concurrent read/write restrictions.

While on a PRAM, any set of parallel memory accesses can be executed in O(1) time regardless of the access pattern, implementing a PRAM algorithm for a GPU must take into account the memory access time overhead as well as the impact of the access pattern. Parallel memory requests coming from multiple processor cores of a CUDA GPU can be coalesced into arrays of contiguous memory locations [12]. However that requires either ordered memory access or careful design of an algorithm processing irregularly organized data structures like graphs.

2.3.3 Sensitivity to Overhead

More so than sequential or MIMD multi-core systems, CUDA SIMT implementations are extremely sensitive to the constants hidden inside many theoretically optimal parallel algorithms. These algorithms often first manipulate the data in order to return much faster results during their main execution This initial manipulation adds a fixed overhead cost, but it is usually substantially smaller than the time required to actually process a large number of data elements

From a theoretical point of view, this extra overhead work is hidden in the theoretical notation of an algorithm's performance since it is independent of the the size of the data. The approach is perfectly reasonable as long as the overhead stays small; however, with parallel algorithms, the overhead applies at every processor doing work. When implementing a parallel algorithm on a GPU using a very large number of processors and threads, each handling a much smaller amount of data, the small overhead becomes large relative to the work done per processor.

Given n elements of data processed in parallel by p threads, the optimal work per processor is a function of $\frac{n}{p}$. If an algorithm is considered optimal with a constant overhead of k per processor where the constant is substantially smaller than the size of the data, it is clear that when the number of processors is small that the total overhead across the processors (i.e. $p \cdot k$) is small as well. However as the number of threads increases substantially, in the order of millions, suddenly the total overhead across the processors becomes quite high.

With CUDA, since the best performance is obtained by running with lots of threads, i.e. a very high value for p, the overhead will be amplified that much more as every thread does the extra work. While this does not take away the optimality of the work, it does however mean that the run-time performance will be poor.

Chapter 3

Review of GPU/CUDA Programming Concepts

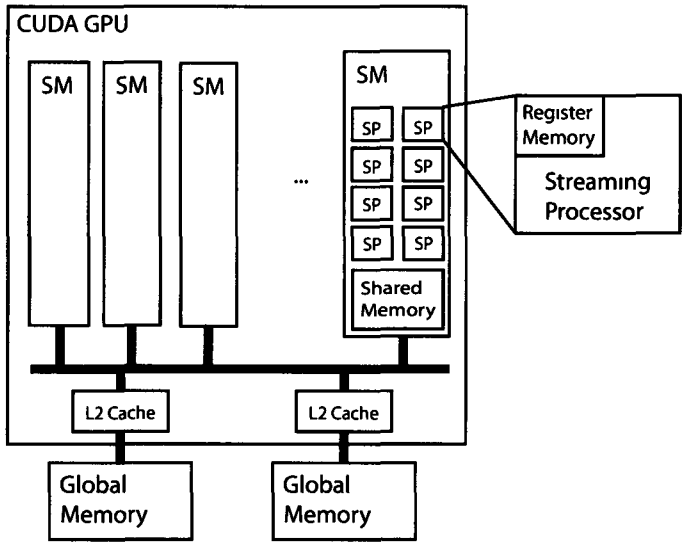


Figure 3.1: Overview of CUDA GPU

The GPU is installed as an add-on within the host computer. Software is developed for NVIDIA GPU cards via the CUDA SDK[3] provided by NVIDIA. Applications developed via the CUDA SDK run code on the CPU which then invokes parallel sub-routines called kernels that are executed on the GPU itself. Since CUDA programs need to be aware of some of the fundamentals of the hardware design, the following facts are important before proceeding to the rest of this chapter.

- As discussed in Section 2.2 and shown by Figure 3.1, an NVIDIA GPU consists of an

array of streaming multi-processors (SMs).

- Each SM is made up of eight streaming processors (SPs or cores).
- The GTX 260 GPU card is made up of 27 SMs; thus a total of 216 cores are available for processing.
- Each *kernel* is executed in parallel using a number of threads grouped into *thread blocks*.
- Each thread block is assigned to an SM and thus all threads in a block execute using the SM's eight cores.
- The execution is grouped into *warps* consisting of 32 threads each.
- Each *warp* is executed SIMD style.
- Synchronization is available in two ways: one method for all threads within a block and another method for synchronization across different thread blocks.
- There is no stack in the GPU architecture and hence there is no recursion supported for the threads within CUDA kernels.

3.1 Memory

3.1.1 GPU Memory

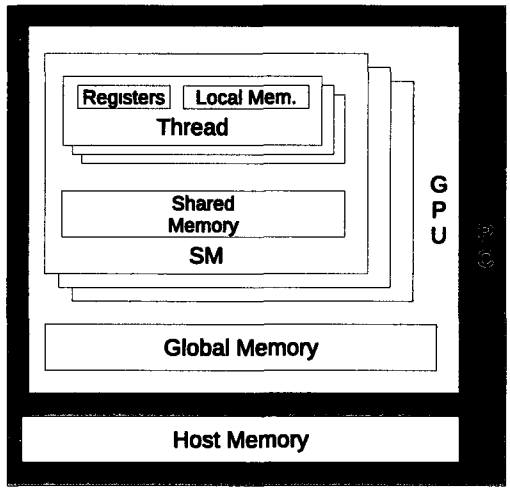


Figure 3.2: Overview of CUDA GPU and Memory Locality

The GPU's on-board memory is divided into the following types[1] that are applicable for general purpose computing:

- *Global* memory, also referred to as *device* memory, is located off the GPU chip and on the GPU card similar to memory on a motherboard for regular CPUs. All global memory locations are accessible by all cores of the GPU through the on-chip interconnection network that routes and schedules all accesses to global memory. Global memory is arranged in independent memory partitions. Each memory partition has its own queue for memory requests and arbitrates among the incoming read/write requests, seeking to maximize memory transfer efficiency by grouping read/write accesses to neighboring memory locations.

- *Shared* memory is on-chip memory local to each SM and accessible by all the cores (and thus all the threads) of the SM. Shared memory in one SM cannot be accessed by cores (and thus threads) in another SM. While access to shared memory is as fast as accessing registers (with some caveats discussed in section 3.6.2), each thread is limited to 16KB of shared memory.

- *Register* memory is on-chip memory local to each SP and thus local to each thread. While register access is extremely fast, there is a limited amount available per thread. Registers are typically used for parameter passing and local variables in kernels.

- *Local* memory is like *global* memory in that it is on-board and thus slow to access; however, like *register* memory, it is local to each thread and is used as a fall-back when a kernel runs out of *register* memory. A caution: when a kernel uses large structures to create local variables, the compiler is likely to move such local variables into *local* memory, resulting in an impact to kernel performance.

Global memory provides the most flexibility as any thread can access any area of the *global* memory; while it is the most abundant source of memory, it is not cached and access times are much slower than using *shared* and *register* memory. While *shared* memory and *register* memory access is extremely fast, only a small amount of each is available. This difference in performance can prove invaluable when optimizing an algorithm for the GPU, and the following steps provide the guidelines to keep in mind when programming the GPU:

- Minimize the use of global memory. Reading once from and writing once to global memory is unavoidable.

[1] There are additional types of memory, including constant memory and texture memory, which are not relevant to this study and thus not explained here Further details can be found in chapters 2 and 5 of [3]

- If the algorithm lends itself to data partitioning then take advantage of shared memory by reading small partitions of data into shared memory and doing all the work on the partition within thread blocks.
- If the algorithm, or the data it works on, isn't amenable to caching, then focus on minimizing repeated access to global memory. Judicious use of registers can significantly reduce the number of global memory accesses.
- Avoid using structures larger than 64 bytes in size where possible.

3.1.2 Host Memory

The host computer's memory is not directly accessible by a kernel executing within the GPU; similarly the program executing on the host CPU cannot directly access any of the device memory on the GPU. The CUDA SDK provides specific APIs that are used to allocate and free *global* memory on the GPU, as well as copy data to and from host memory and the GPU's *global* memory.

When an application allocates memory on the GPU, it is given a pointer that is invalid for use with the host CPU. It is the application's responsibility to remember which pointers refer to GPU memory and which point to host memory. A naming convention is used whereby a pointer variable name uses a different suffix as follows:

```
int *data_h = malloc (...)       // host memory
int *data_d = cudaMalloc (...)   // device global memory
```

3.2 Kernels

A sub-routine that is executed on the GPU is called a *kernel* and is written as a special C function using one of two keywords:

__global__ A prefix used to define a kernel that can be invoked by the main program running on the host CPU.

__device__ A prefix used to define a kernel that can only be invoked by another kernel while it is executing on the GPU.

All kernels must follow the following rules when they execute:

- A kernel can access CUDA registers, *shared* memory and *global* memory.
- Any pointer provided to a kernel during invocation must point to CUDA *global* memory.

- A kernel cannot access host memory[2].
- A kernel cannot use recursion since there is no stack; all local variables are stored in CUDA's registers.
- A kernel is executed using a fixed number of threads which is specified by the host program when it invokes the kernel.

When programming in ANSI C, a global kernel is defined as a C function with a CUDA-defined prefix, as shown below.

```
// host-invokable kernel
__global__ void myKernel (...) {
    ...
}

// kernel-invokable kernel
__device__ void mySubKernel (...) {
    ...
}
```

The host program invokes the kernel by specifying the following values:

- The number of thread blocks to use, as many as desired with no limit imposed.
- The number of threads to use per thread block, up to a maximum of 768.
- The parameters to pass the kernel.

```
...
myKernel <<NUMBLOCKS, BLOCKSIZE>> (p1, p2, ...)
...
```

3.2.1 Kernels using Shared Memory

When using shared memory, the kernel is defined with a shared memory declaration using the CUDA-specific prefix **__shared__** inside the kernel. The following is an example of a kernel where the size of the shared memory is known at compile time.

```
__global__ void myShMemKernel (data_t *gdata) {
    __shared__ data_t shdata [BLOCKSIZE*sizeof (data_t)];
    ...
}
```

[2]There is a way to access host memory from within a kernel via page-locked memory mapping[3, p 32], however it has restrictions that make it unsuitable for the work done as part of this study

The above kernel processes well-defined data elements, and since shared memory can only be initialized by the kernel, we need to pass in the data via *global* memory and have the kernel copy what it needs into the shared memory. The shared memory in a single SM is shared between all the threads of the thread block executed by the SM, and as a result the space must be divided among the threads. Since the size of the shared memory is computed at compile time, invoking the kernel is no different than one that doesn't use shared memory.

Sometimes it is not possible to compute the size of the shared memory at compile time. In that case it is possible to dynamically allocate the shared memory by defining it within the kernel using the additional prefix **extern**, and then passing in the size of the shared memory when invoking the kernel.

```
__global__ void myShDynKernel (data_t *gdata) {
    extern __shared__ data_t shdata [];
    ...
}
...
myShDynKernel <<NUMBLOCKS, BLOCKSIZE, shDataSize>> (...)
...
```

3.2.2 Executing Kernels in Parallel

By default the CUDA SDK associates each program with a single CUDA "stream" such that each kernel invocation is associated with a stream, and only one kernel is allowed to execute at any given time per stream.

The SDK provides APIs that allow an application to initialize multiple streams and thus run kernels in parallel. The use of parallel streams is particularly interesting as a way of processing the same, or overlapping, data with different algorithms at the same time. It is also a way for algorithms that require a small number of threads or thread blocks to take advantage of all the SMs in a GPU by parallelizing more work, thus taking advantage of the latency hiding to improve total performance.

Since the graph algorithms studied in this paper process more data using more thread blocks than there are SMs in the GPUs, they do not require nor use multiple streams; however it is an interesting aspect of CUDA that may be leveraged with the appropriate algorithms.

3.3 Threading

A GPU application does not generally need to be aware of the number of physical cores. It can create thousands or millions of threads, as needed, and is in fact encouraged: an

application needs to use substantially more threads than there are physical cores to take advantage of the performance gains offered by latency hiding (as discussed in section 2.1.3.)

When a kernel is invoked, the number of thread blocks and threads per block specified are multiplied to identify the total number of threads, which are then assigned by thread blocks to the SMs on the GPU. An SM executes a thread block by breaking it into groups of 32 threads called *warps* and executing them in parallel using its eight cores. More precisely, the SM performs context switching between the different warps. This allows the SM to hide the latency of memory access operations performed by the threads and provides increased computational performance. When a warp is being executed, the eight cores also perform context switching between the warp's 32 threads. The eight cores of an SM share various hardware components, including the instruction decoder. Therefore, the threads of a warp are executed in SIMT mode, which is a slightly more flexible version of the standard SIMD mode. The active threads of a warp all need to execute the same instruction as in SIMD mode while operating on their own data.

3.4 Branching

Parallel execution can break down when the threads encounter a conditional branch such as an IF-THEN-ELSE statement. Depending on their data, some threads may want to execute the code associated with the "true" condition and some threads may want to execute the code associated with the "false" condition. Since the shared instruction decoder can only handle one branch at a time, different threads can not execute different branches concurrently and they have to be executed in sequence, leading to performance degradation. The SMs provide a small improvement through an instruction cache that is shared by the eight cores which allows for a "small" deviation between the instructions carried out by the different cores. For example, if an IF-THEN-ELSE statement is short enough so that both conditional branches fit into the instruction cache then both branches can be executed fully in parallel. However, a poorly designed algorithm with many or large conditional branches can result in serial execution and very low performance.

3.5 Thread Synchronization

The PRAM model assumes full synchronization at the level of individual steps and does not account for synchronization overhead; ie. synchronization has no cost. On a GPU this is not the case: CUDA supports two very specific types of synchronization, each with its own constraints and costs.

Threads within a thread block can be synchronized by calling `__syncthreads()` from

inside the kernel. Since threads in different thread blocks can not be synchronized within the same kernel, this approach is useful for algorithms where the thread blocks are independent of each other and do not need to share or process arbitrary locations of global memory while they work.

Barrier synchronization across all threads and thread blocks can be achieved by breaking an algorithm into a sequence of different kernels and invoking them in sequence from the host application. When two kernels are invoked in sequence all threads working on the first kernel must complete their work before the second kernel can be started. The key to this approach is to identify the minimum number of kernels required to implement an algorithm, as kernel invocation can affect performance if abused.

3.6 Memory Access Optimization

While the usage of *shared* memory of each SM is restricted to the threads in a thread block and is limited in size (up to 16K per thread), it is much faster to use than global memory. There is a trade-off that needs to be made during the design of the algorithm to take advantage of the fact that *shared* memory access is as fast as using registers. The use of *shared* memory is ideal for algorithms such as sorting that are able to break up the input into small pieces and then perform most of the work on those small pieces. However not all algorithms can be designed to break their work into well-defined pieces that can be fit into the limited shared memory; and certainly graph algorithms that work on graphs with links arbitrarily distributed throughout the whole graph are not suitable for data partitioning.

Another reason for using *global* memory is to move the data from the host memory into the GPU for processing. This means an algorithm that is able to use *shared* memory effectively still needs to load its data from *global* memory and then write back its results into *global* memory so that it can later be copied to the host memory. Thus at least one read and write operation must be performed from and to *global* memory; and this alone can dominate the computational time of an algorithm.

3.6.1 Coalescing Global Memory Access

To assist with the optimization of global memory access the GPU hardware supports the following notions when accessing global memory:

Memory Access Transactions The GPU hardware has a specialized memory access network (called the inter-connection network) that connects all the SMs to all the global memory modules. When a kernel accesses global memory, the underlying hardware issues memory access transactions to the inter-connection network. Each transaction

is then dispatched to the appropriate memory module for processing, be it a read or a write.

Bulk Transactions Each memory access transaction can read or write more than one unit of a data type. In fact each transaction can process up to 128 contiguous bytes of data at a time from the same memory module.

Coalescing When multiple concurrent memory accesses by different threads are combined into a single memory access transaction, thus avoiding the overhead of issuing and waiting for multiple independent memory accesses.

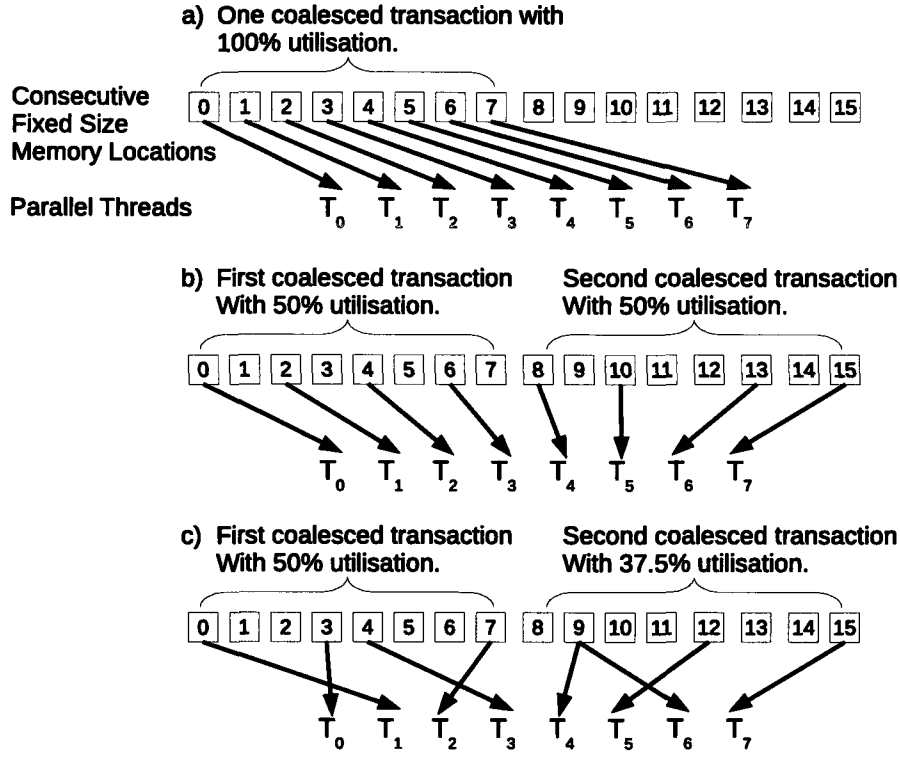


Figure 3.3: Examples of Coalesced Memory Access Transactions

When an application uses *global* memory, a critical requirement for obtaining good performance is to coalesce memory accesses performed concurrently by different threads. The goal is to combine multiple *global* memory access requests executed concurrently by multiple threads into one single bulk transaction for one of the independent memory partitions of *global* memory. The performance improvement that can be gained through coalescing of memory accesses can be substantial [3, 21].

When threads are executing in parallel, if each accesses memory at a consecutive address then the GPU is able to coalesce that access into a smaller number of direct memory accesses,

thus taking advantage of the memory access hardware's capacity to execute 64-byte or 128-byte memory access transactions. This relies on the correct alignment of memory access in the algorithm (see [3, p.81-88] for a detailed technical treatment). To take advantage of this, each thread should access consecutive memory addresses in order relative to the thread index. This is illustrated by example (a) in Figure 3.3 where all the threads access consecutive elements from the global memory, resulting in a single coalesced memory access transaction rather than 8 separate memory access transactions.

As of CUDA 1.2, the GPU is capable of coalescing all requests made by the half-warp's threads that fall within the same segment (i.e. contiguous block of memory), where the size of the segment is governed by the size of the memory access. This means that the order of access matters less than the amount of memory accessed concurrently, and overlapping accesses are allowed as long as they fall within the segment. This is illustrated by examples (b) and (c) in Figure 3.3 where all the threads access non-consecutive elements from the global memory, producing two transactions; even though the two transactions together access 16 memory locations to retrieve data for 8 threads, this is still more efficient that 8 separate memory access transactions accessing less memory in total.

Data Size	Segment Size	Min. Transaction Size
1 byte (8 bits)	32B	16B
2B (16b)	64B	32B
4B (32b)	128B	64B
8B (64b)	128B	64B

Table 3.1: CUDA 1.2 Global Memory Segment & Transaction Size

Table 3.1 shows the segment sizes by data size. For example, if 16 data accesses of 2 bytes each fit exactly into a 32-byte memory segment then this creates one memory transaction of 32 bytes. If those 16 data accesses are not adjacent but fit into a 64-byte memory segment then this creates one memory transaction of 64 bytes. If the transactions fall into multiple segments then this creates multiple memory transactions; this is not desirable and should be avoided where possible to maximize performance.

With graph manipulation algorithms it is not always possible to avoid non-coalesced memory access. Figure 3.4 shows an example of such a memory access where the data being processed is sufficiently random with links distributed in a non-consecutive manner. In the first step, each thread T_i reads a value from the array of indices based on the thread's ID (i.e. $X_i = a[i]$) . As the thread IDs are consecutive, the memory access is easily reduced to a single transaction. In the second step, the value initially read by each thread is now used as a new index into the array (ie. $Y_i = a[X_i]$), resulting in read operations that are neither consecutive, nor within the bounds of a reasonable segment size. As a result, each

new array access has to be issued as a separate memory transaction.

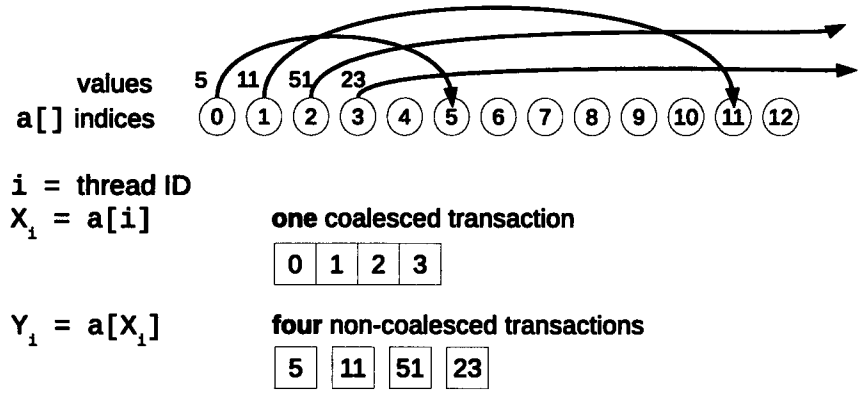


Figure 3.4: Example of Unavoidable Non-coalesced Memory Access

3.6.2 Coalescing and Shared Memory

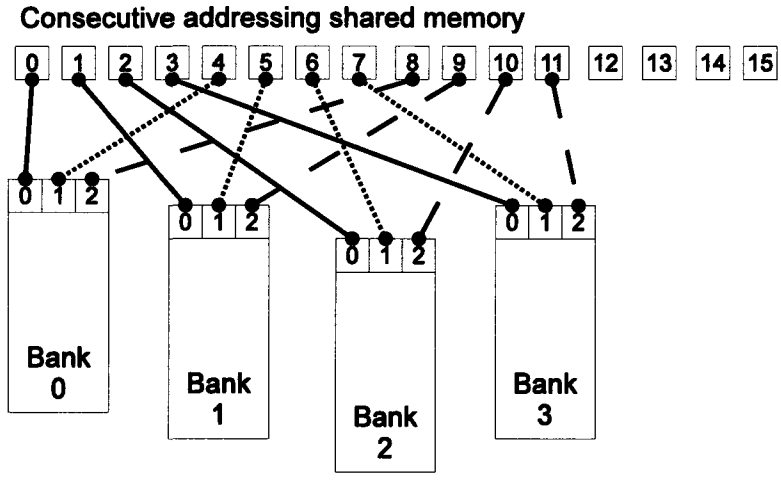


Figure 3.5: Shared Memory Banks and Address Assignment

Shared memory is organized differently: consecutive 32-bit locations from the kernel's point of view are assigned to different memory banks (partitions) and optimal performance is achieved when concurrent access does not cause a bank conflict (see [3, p.89-96] for a detailed technical treatment). This is illustrated by Figure 3.5, where consecutive addresses $0, 4, 8$ are mapped to bank 0, addresses $1, 5, 9$ are mapped to bank 1 and so on. Therefore when four threads consecutively access addresses $0, 1, 2, 3$ each of them will actually read concurrently from a different shared memory bank, but when they access addresses $0, 4, 8, 12$

there will be a bank conflict as all threads attempt to read from the same shared memory bank.

This interleaved address mapping of shared memory means that the most common method of accessing data from shared memory using the thread ID as the index provides zero conflicts, as long the size of each data access is 32 bits. The main difference relative to global memory access is that the size of the data element matters with shared memory; thus sequentially accessing 32-bit integers is ideal for performance, while doing the same with 8-bit or 16-bit values is not. If four threads read four consecutive 8 bit characters in shared memory they will cause a bank conflict as all four characters are in the same 32-bit value in one bank. No conflict occurs, however, if each thread accesses 8-bit characters that are 32 bits apart. This case can be expanded further in that memory accesses do not cause conflicts if they are spread apart in such a way as not to result in concurrent access to the same bank.

Naturally, when all threads access the same location concurrently, no conflict occurs (see [3, p. 91]).

3.7 Memory Write Concurrency

CUDA supports concurrent write attempts to global memory without causing any failure in execution The entire global memory is accessible by all cores of the GPU through the on-chip interconnection network that routes all memory requests to the respective memory modules. Each memory partition has its own queue for memory requests and arbitrates among the incoming read and write requests. When writing to and reading from the same location the following points apply:

- Concurrent writes to the same location are executed in unknown order resulting in one of them succeeding while the others are ignored without errors.
- Concurrent reads from the same location are of course optimal as a result of the coalescing of memory access discussed in section 3.6.1.

On a cautionary note, mixing concurrent reads and writes to the same memory location must be handled carefully, especially when multiple thread blocks are involved. It is important to ensure that all threads are synchronized between reading and writing to the same location unless it can be proven that the arbitrary access will not compromise the outcome (such as the Shiloach-Vishkin algorithm for computing connected components [17]which is discussed in greater detail in Chapter 6).

3.8 Striding & Partitioning

The PRAM model typically assumes one parallel processor per data item. This is not always possible even though GPUs allow an application to allocate millions of concurrent threads. Current nVIDIA Tesla architectures provide up to 4 GB global memory; assuming that 1 billion data elements could be loaded into that memory, to process each element with its own thread means allocating a billion threads, which is not possible with today's hardware. As the same time, even though the hardware thread scheduler is very efficient, there is a cost, however small, associated with thread scheduling, and there is a point at which too many threads hamper the performance. There are also algorithmic reasons for having fewer threads than data items (as in the case of the Parallel Random Splitter List Ranking algorithm discussed in greater detail in Chapter 5).

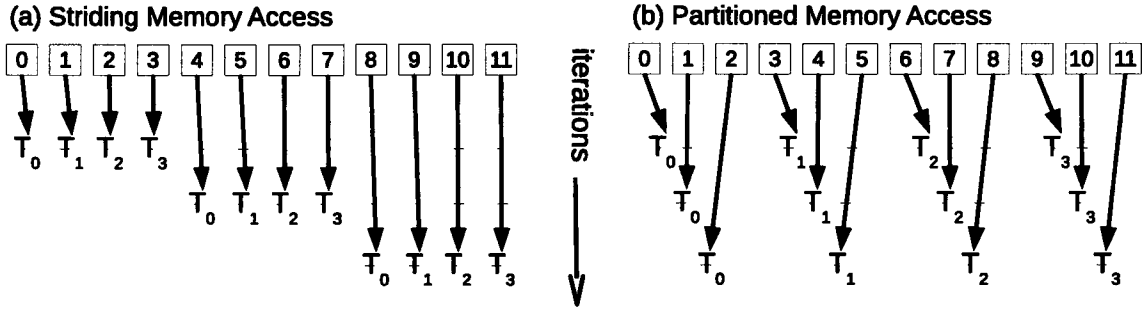


Figure 3.6: Example of Striding Access vs Partitioned Access

Since many PRAM algorithms do make the one-thread-per-data-element assumption, we need a way to modify that algorithm when implementing it to execute on a GPU so that a fewer fixed number of threads are able to process an arbitrarily large amount of data limited only by the amount of available memory. Consider n data items in an array $A[0]$,..., $A[n-1]$ that need to be accessed by p threads T_0, ..., T_{p-1}. There are two implementation strategies to choose from when there are more data than threads (i.e. $n > p$):

1. **Striding**, where each thread over s iterations accesses every pth element in the array: i.e. thread T_i accesses data item $A[i+s \cdot p]$ in iteration s for $0 \leq s < \frac{n}{p}$ and $0 \leq i \leq p-1$, as illustrated by Figure 3.6 (a).

2. **Partitioning**, where each thread over s iterations accesses every consecutive element in a partition of the data assigned to the thread: i.e. thread T_i accesses data item $A[i\frac{n}{p} + s]$ in step s for $0 \leq s < \frac{n}{p}$ and $0 \leq i \leq p-1$, as illustrated by Figure 3.6 (b).

Depending on the multi-processor system's design and memory access caching and optimization support, one of the two strategies must be chosen for optimal run-time performance.

On an Intel processor where there are multiple cores executing in MIMD mode with multiple caching levels for memory access, data partitioning provides the best performance by making sure that each processor gets its own chunk of data to access in sequence without needing to refill the cache. On a GPU where the SMs contain multiple SIMD cores, where memory access is not cached and the system favours coalesced memory accesses across parallel warps and half-warps, striding provides the best performance (see section 3.6.1).

The following pseudo-code illustrates how striding is implemented, where all $p < n$ threads run in parallel and each thread, starting at thread index i, works on data elements $j = \{i, i+p, i+2p, \ldots\}$.

```
j = i
while j < n
    -- do work with data at index j
    -- stride by p to next element
    j = j + p
```

The following pseudo-code illustrates how partitioning is implemented, where all $p < n$ threads run in parallel and each thread, starting at index $i \cdot \frac{n}{p}$, works on data elements $j = \left\{\frac{n}{p}, \frac{n}{p}+1, \frac{n}{p}+2 \ldots\right\}$.

```
j = i*(n/p)
lim = j + (n/p)
while j < lim
    -- do work with data at index j
    -- next consecutive element
    j = j + 1
```

Chapter 4

Review of Studied Graph Algorithms

The following graph algorithms, considered fundamental to other graph applications, were implemented and analyzed as part of this thesis.

- Parallel List Ranking
- Parallel Connected Component Counting

In this chapter we present the basics behind the two problems as well as the algorithms used to solve them on parallel shared-memory systems.

4.1 Parallel List Ranking

A basic operation required by nearly all (parallel or sequential) graph algorithms is to traverse a linked list of edges/pointers. In this section, we will start with the classical parallel *list ranking* problem (see e.g. [16, p. 80]) and study how to convert well-known parallel algorithms for list ranking into efficient GPU implementations by following the guidelines outlined in Chapter 3.

4.1.1 Definition

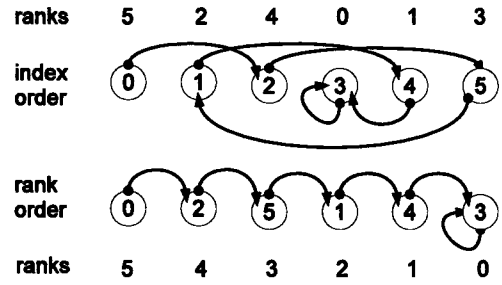


Figure 4.1: Example of List Ranking

Consider a linked list of n vertices represented by an array $succ\,[0 \ldots n-1]$ where each slot $succ[i]$ points to the next element in the linked list. The first element of the linked list is $succ[0]$ and the last element at index x points back to itself such that $succ[x] = x$.

The **rank of a list vertex** is the measure of its distance from the last vertex in the list. The ranks of all list vertices are reported as an array $rank\,[0..n-1]$. Figure 4.1 illustrates list ranking by showing an example of a linked list in index order (i.e. order of list array *succ*) and the list in rank order (i.e. order by following the links).

4.1.2 Sequential List Ranking

List ranking sequentially on a single processor system is trivial: the algorithm, as listed in Algorithm 4.1, starts at the head of the list and follows the links, assigning each vertex in the list its appropriate rank starting with $n-1$ for element at index 0 and decrementing by 1 for each hop. Since the algorithm visits each vertex once, it does $O(n)$ work in $O(n)$ time.

Algorithm 4.1 Serial List Ranking

```
i = 0
r = n - 1
while (succ [i] != i):
    rank [i] = r
    i = succ [i]
    r = r - 1
```

Given the amount of work that is done by the sequential algorithm, a parallel algorithm should ideally do no more than $O(n)$ work regardless of the number of processors and do the work in $O\left(\frac{n}{p}\right)$ time or better with p processors.

4.1.3 Parallel Pointer Jumping

The simplest parallel algorithm, proposed by Wylie [23, 16, p. 64] , relies on pointer jumping, also known as short-cutting. The algorithm relies on a temporary working array $last\,[0 \ldots n-1]$ that is used to represent the links from each vertex to the last vertex in the linked list. There are two broad parts to this algorithm:

1. Initialization, where in parallel for each vertex at index i in the list, its rank $rank\,[i]$ is initialized to 1, or zero in the case of the last vertex, and its link to the last vertex $last\,[i]$ is initialized to its immediate successor.

2. Pointer jumping (as shown by steps 1 through 3 in Figure 4.2), where in parallel for each vertex at index i in the list ...

   (a) The current rank value is added to the rank of the current last vertex: $rank\,[i] = rank\,[i] + rank\,[last\,[i]]$

   (b) The current last link is updated to jump to its successor's last link. $last\,[i] = last\,[last\,[i]]$

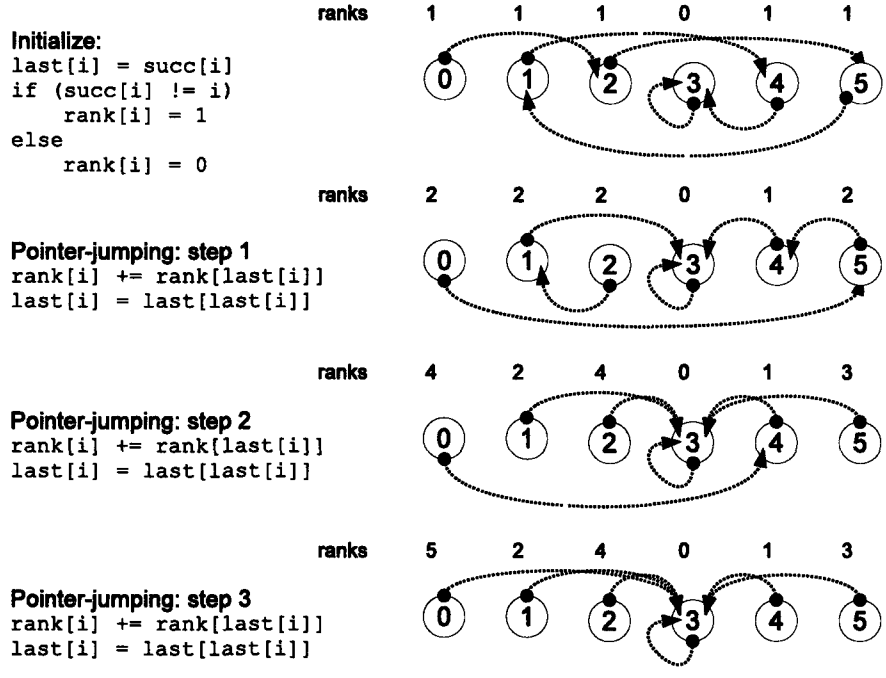


Figure 4.2: Example of List Ranking by Pointer Jumping

The idea behind the algorithm is to execute multiple iterations where, in each iteration, for each vertex in parallel, the distance to the last vertex in the list is counted by jumping in

hops of increasing length. At each iteration s, each vertex jumps over at most 2^s successors, thus taking *short-cuts* over large numbers of vertices. The first vertex in the list has the furthest to jump, with n hops to get to the last vertex. Since the pointer jumping doubles in distance at each iteration, there are at most $\log_2 n$ iterations needed. Thus the whole algorithm does $O(n \log_2 n)$ work; and if we have p parallel processors and an ideal system with no overhead it should run in $O\left(\frac{n \log_2 n}{p}\right)$ time.

Algorithm 4.2 Parallel List Ranking by Pointer Jumping with $p < n$ Processors

```
Kernel PJ1 using p<n processors (processor index i = 0 ... p-1):
    j = i
    while j < n:
        rank [j] = (succ [j] != j)  -- 0 if last vertex, 1 if not
        last [j] = succ [j]
        j = j + p

Kernel PJ2 using p<n processors (processor index i = 0 ... p-1):
    j = i
    while j < n:
        -- each of next 3 lines must be atomic
        r1 = rank [j], l1 = last[j]
        r2 = rank [l1], l2 = last[l1]
        rank [j] = r1 + r2, last [j] = l2
        j = j + p

Host:
    invoke PJ1
    repeat log (n) times:
        invoke PJ2
```

For real world problems, the size of the lists and graphs being processed will always exceed the number of processors available, i.e. $p \lll n$. Algorithm 4.2 show the parallel pointer jumping algorithm adapted for GPU execution as follows:

- *Two kernels* are defined that represent the two broad steps of the algorithm.
- Under the assumption that $p < n$, both kernels employ striding (see section 3.8) to process more data than there are processors. By using p processors and executing *kernel* PJ1 first followed by repeatedly calling PJ2 up to $\log_2 n$ times, each processor effectively strides over $\frac{n}{p}$ elements and does $O(n \log_2 n)$ work.
- Kernel PJ2 implements the pointer-jumping step by breaking it into 3 atomic operations. The algorithm shows each line with either two reads or two writes and requires each line to operate atomically (i.e. both reads or both writes happen together and without any chance of race condition between the two operations in each line). This is important since the GPU provides ordering guarantees only within a single

thread-block and not across multiple thread-blocks. The actual implementation of the algorithm will need to take appropriate measures to ensure that the two operations in each line execute atomically when more than one thread-block is used.

While it is likely that on a parallel system that the algorithm will execute faster than the sequential one given enough processors, this algorithm does $\log_2 n$ times more work than the sequential one.

4.1.4 Parallel Random Splitter List Ranking

The work of the parallel list ranking algorithm can be improved if, instead of ranking the entire list using pointer jumping, we split the list into $r < n$ sub-lists and use pointer jumping to rank a smaller list of r nodes instead. This is very similar to Reid-Miller's algorithm for parallel list ranking on a Cray C90[15], and performs the list ranking as outlined below:

1. Randomly divide the list into r sub-lists by randomly choosing r splitter nodes from the n vertices, creating a linked list of splitters of length r made up of the first vertex of each sub-list.

2. Create a splitter rank array of length r, with the rank of each splitter initialized to the length of its sub-list.

3. Use parallel pointer jumping (see section 4.1.3) with r processors to rank the splitter list, using the pre-initialized splitter ranks from step 2. This will yield the ranks of the splitters relative to their positions in the input list of length n.

4. In parallel, using as many processors as desired, stride over the input list and compute each vertex's rank based on the rank of its sub-list.

Algorithm 4.3 shows the GPU adaptation of the outlined steps. Kernels RS1 and RS2 together initialize and select the random splitters, while kernel RS3 counts the sub-list lengths and constructs the link list of splitters. Kernel RS4 uses a simplified implementation of Wylie's to rank the list of splitters, and then kernel RS5 aggregates the local ranks of the nodes in each sub-list with the final rank of each of their splitters.

The kernels RS1, RS3 and RS5 process n elements and thus do $O(n)$ work each. The kernel RS2 does $O(r)$ work and RS4 (pointer jumping) does $O(r \log_2 r)$ work, thus doing $O(n + r \log_2 r)$ work as a whole. However if we can pick r's value such that $r \log_2 r \leq n$ then kernel RS4 does $O(n)$ work at most, and thus the whole algorithm can be shown to do $O(n)$ work!

Algorithm 4.3 Parallel Random Splitter List Ranking

```
Kernel RS1 using p<=n processors (processor index i = 0 ... p-1):
    j = i       -- initialize owner array
    while j < n:
        owner [j] = -1
        j = j + p

Kernel RS2 using p=r processors (processor index i = 0 ... p-1):
    -- select random splitter
    splitter = random (i*p, ((i+1)*p)-1)
    owner [splitter] = i
    rank [splitter] = 0

Kernel RS3 using p=r processors (processor index i = 0 ... p-1):
    dist = 1    -- walk the sub-lists, marking and counting
    prev = splitter
    j = succ [splitter]
    while (owner [j] = -1) and (prev != j):
        rank [j] = dist
        owner [j] = i
        prev = j
        j = succ [j]
        dist = dist + 1
    if (prev == j) dist = dist - 1
    -- prepare the splitter linked list
    spsucc [i] = owner [j]
    sprank [i] = dist

Kernel RS4 using p<=r processors (processor index i = 0 ... p-1):
    -- rank the splitter list using pointer jumping---
    repeat log (p) times:
        j = i
        while (j < r)
            sprank [j] = sprank [j] + sprank [spsucc [j]]
            spsucc [j] = spsucc [spsucc [j]]
            j = j + p

Kernel RS5 using p<=n processors (processor index i = 0 ... p-1):
    j = i       -- aggregate the local ranks with the splitter ranks
    while j < n:
        rank [j] = sprank [owner [j]] - rank [j]
        j = j + p
```

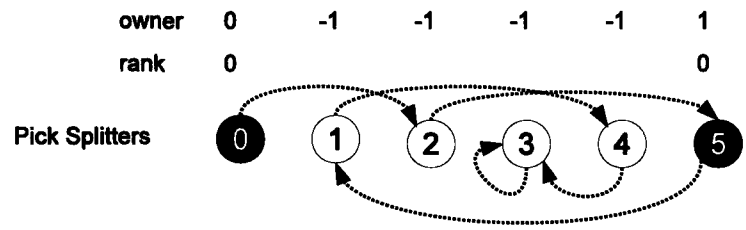
For $n = 1000000$ we can use up to $r = 62700$, i.e. $r \log_2 r = 999198 < n$, and maintain $O(n)$ work. As we increase n to more realistic sizes for the input, so does the upper limit for r that maintains $r \log_2 r \leq n$. When implementing this algorithm we have found that for optimal performance r needs to be a small multiple of the number of physical processors on a GPU, and thus, even for high values of n, we typically use a fixed value for r that is easily much smaller than the maximum needed to maintain linear work. In fact for the GTX 260 GPU with its 27 SMs (see section 2.2 on page 11) and a total of 216 cores, we use $r = 16384$

regardless of the value of n to obtain the best performance for this algorithm while doing $O(n)$ work.

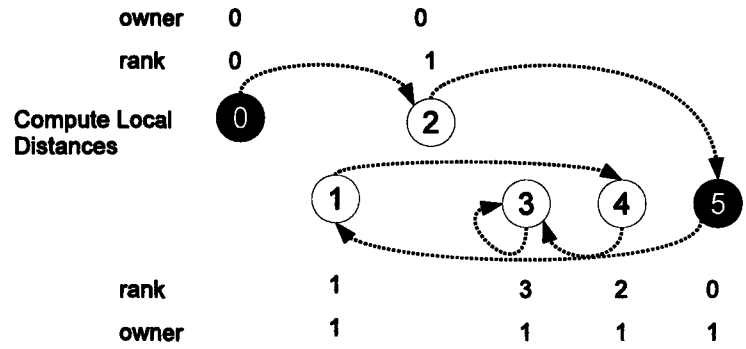
Example of Random Splitter List Ranking in Parallel

Algorithm 4.3 on the previous page is made up of five kernels which are invoked in sequence. The first kernel (RS1) is executed using as many threads as possible and initializes the owner to -1 for each vertex.

The second kernel (RS2) is executed using r threads, where each thread selects one random vertex from the list as its splitter. It is necessary to ensure that no two splitters are the same, the details of which are discussed in the implementation of this algorithm in chapter 5 on page 46. Since r threads are used, the thread index $i = \{0 \ldots r-1\}$ represents the index of the splitter in the list of splitters. The owner of each splitter is initialized to its thread index, and the rank of each splitter is initialized to zero. (See below for the outcome of this kernel for a simple example list.)



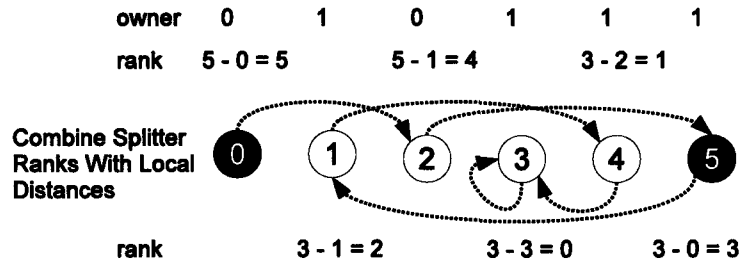
The third kernel (RS3) is also executed using r threads, where each thread selects each splitter and walks along its sub-list counting the length of the sub-list and marking the relative distance of each vertex to the splitter as its rank. Thus for a splitter at index j, its immediate successor $succ[j]$ has rank 1, and the next one 2 and so on.



Once the distances have been computed, the third kernel establishes the linked list of splitters (*spsucc*) and the initial splitter ranks (*sprank*) based on the lengths of the splitters' sub-lists.

The fourth kernel (RS4) is a specialized implementation of the pointer jumping algorithm's second kernel (PJ2). While the kernel in Algorithm 4.2 on page 33 relies on a temporary $last[]$ array, this kernel uses the splitter list $spsucc[]$ array instead, modifying it in the same way until each splitter points to the last splitter in the splitter list. Note that this kernel implements the $\log_2 r$ iterations within the kernel, which imposes the restriction that the kernel must be executed using a single thread-block, with at most 768 threads (see chapter 3 on page 16).

The fifth kernel (RS5) is executed using as many threads as possible, where each thread strides over the elements of the list computing the rank of each vertex. For any vertex at index j at this time we can establish its splitter via $s = owner[j]$ and its distance from its splitter via $rank[i]$ and it's splitter's rank via $sprank[s]$. The rank of the vertex is then computed simply as $rank[j] = sprank[owner[j]] - rank[j]$, i.e. the rank of the splitter less the distance of the vertex from its splitter.



4.2 Parallel Connected Component Counting

Identifying and counting connected components within a graph is a common requirement of many graph processing applications. In this section we will start with a well-known PRAM algorithm by Shiloach and Vishkin for computing the connected components of a graph in parallel and study how to convert it into an efficient GPU implementation.

4.2.1 Definition

According to Diestel [8, p. 10, 11], a non-empty graph G is called *connected* when any two of its vertices are linked by an edge in G. A connected sub-graph of G is called a *component* of G. As shown by the example in Figure 4.3, one graph can be made of one or more connected components. Trees and lists are examples of graphs that have one connected component, viz. themselves, while more complex graphs with different properties can have multiple connected components.

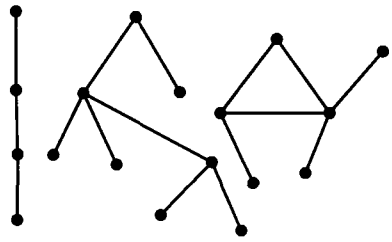


Figure 4.3: An example of a graph with 3 connected components

There are two ways to represent a graph as a data structure:

- In a vertex-representation, a graph is stored as a list of vertices, $V[0 \ldots n-1]$, where each vertex holds a list of the neighbouring vertices that it is connected to. Given an undirected graph, when an edge connects to vertices, each vertex will contain the other vertex in its list of neighbours. Thus a vertex-representation is inherently directional.
- In an edge-representation, a graph is stored as a list of edges, $E[0 \ldots m-1]$, where each edge $E[i] = (a, b)$ is a pair of vertex indices. An edge graph can be both undirected or directed, i.e. edge (a, b) can either mean both a and b point to each other or that only a points to b. In this study, edges in this representation are assumed to be directed unless explicitly stated otherwise.

Linear graphs such as lists and trees always contain fewer edges than they have vertices; i.e. $m = n - 1$, while complex graphs can contain up to $m = O\left(n^2\right)$ edges when every vertex is connected to every other vertex.

4.2.2 Sequentially Counting Connected Components

Sequentially counting or identifying the connected components of a graph is trivial: given a vertex-representation (see Figure 4.4), visit each vertex in the graph and recursively label all vertices connected to it, assigning a unique label only when an unlabeled vertex is visited. Algorithm 4.4 on the next page shows two sub-routines:

- `countConnectedComponents()` initializes an array of labels of length n to zero for each vertex, after which it starts with an initial label of zero and visits each vertex in the graph. When it encounters an unlabeled vertex, it increases the label number and recursively labels the connected components by calling the next sub-routine.

- `recurseLabel()` starts with the supplied vertex and labels each adjoining vertex with the same label. It stops when it encounters a vertex that it has already labeled, thus handling circular paths in the graphs.

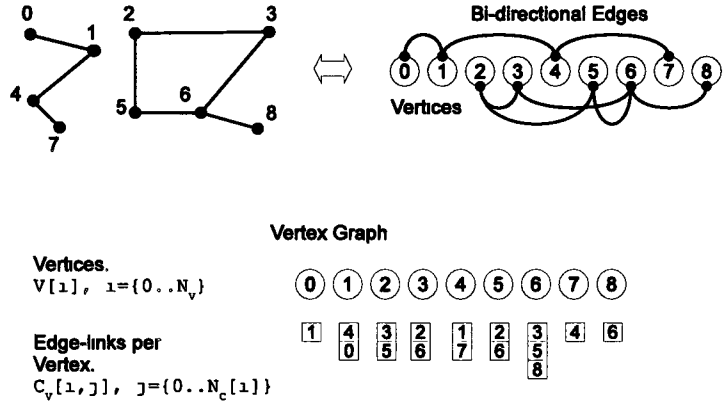


Figure 4.4: Vertex representation of a sample graph

The sequential algorithm visits each vertex at least twice and each edge at most once, thus doing $O(n+m)$ work for a graph with n vertices and m edges in order to obtain a count of the components within the graph.

Algorithm 4.4 Serial Connected Components Counting by Labeling

```
function countConnectedComponents (G):
    l = 0
    for each v in G
        labels [v] = 0
    for each v in G
        if labels [v] = 0 then:
            l = l + 1
            call recurseLabel (v, labels, l)
    return l

function recurseLabel (v, labels, l)
    if (labels [v] != l) then:
        labels [v] = l
        for each c in children [v]:
            call recurseLabel (c, labels, l)
```

Note that this algorithm relies on a vertex-representation of the graph while the parallel algorithm discussed later in this section relies on an edge-representation (see Figure 4.5) Since this study is focused on the parallel algorithms, the input graphs are expected as edge graphs, thus requiring conversion for use with the sequential algorithm. Converting an edge-representation into a vertex-representation can be done with $O(n+m)$ work by walking the edge list and updating the two vertices of each edge.

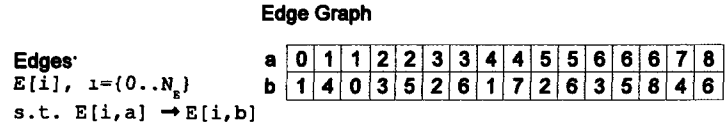


Figure 4 5: Edge Representation of a Graph

4.2.3 Parallel Counting of Connected Components

As with the parallel list ranking problem, the preference is to implement the parallel algorithm such that it does no more work than the sequential algorithm. However, the recursive linear scanning model of the sequential algorithm cannot be efficiently parallelized, and the best known alternate approaches do $O(n \log n)$ work. Chapter 2.2 of [16] presents the general techniques for parallelizing the computation of connected components.

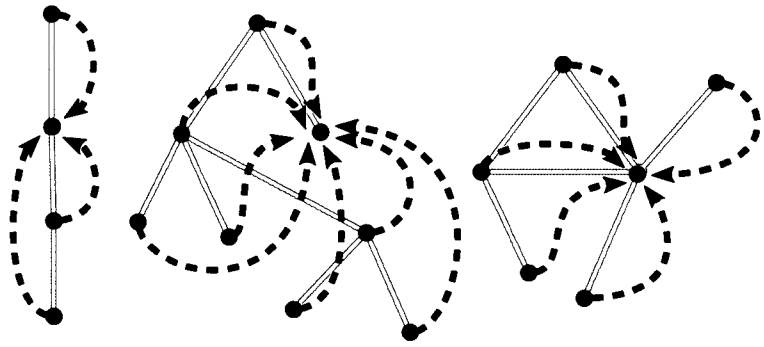


Figure 4.6: Components as Stars (i.e. Trees with height 1)

The strategy is to repeatedly partition the graph's vertices into rooted trees such that, starting with each vertex as a tree of size one and over multiple iterations combining them into larger trees, we end up with each component in the graph as a tree separate from the other components. The trees are generated as an overlay on the graph; i.e. the same vertices are used but the original edges are not modified. It is best to imagine the original graph $G(V,E)$ as the source of the new graph (a forest of trees) $G'(V,E')$ that uses the same vertices but is effectively a different graph with different edges.

Short-cutting, the same pointer-jumping concept used in Wylie's parallel list ranking algorithm (see section 4.1.3), is used to shorten the trees so that each component's tree ends up with a height of 1. Short-cutting works entirely in G'. Hooking is used to join two trees in G' that have a common parent in G, building trees even as short-cutting reduces their depth.

The parallel algorithm's goal is to use the connectivity information in the original graph G to construct the tree graph G' such that each connected component becomes one of the trees in G'. In addition, each tree has a height of 1 and is also known as a star graph (see Figure 4.6 which shows the edges in G as grey lines and the edges in G' as black curved arrows). This helps to quickly count and identify the components in G' since every vertex in a component will have the same parent.

4.2.4 Shiloach and Vishkin's PRAM Connected Components Algorithm

Given a graph with n vertices and m edges, the PRAM algorithm by Shiloach and Vishkin[17] computes the connected components by doing $O((n+m)\log n)$ work

The algorithm requires any two connected vertices to be connected by an edge in both directions, and thus expects there to be $2m$ directed edges. It also requires the availability of $n+2m$ processors using which the algorithm performs $O(n+2m)$ parallel operations during each of its $\log n$ iterations. Each iteration is made up of several steps, and while each step is executed in parallel, barrier synchronization is needed between each step to ensure that each step completes before the next one is started.

Analyzing the original PRAM algorithm with the intent of modifying it to execute on a CUDA GPU reveals some changes that can be made to the above assumptions. Within each iteration, each of the steps either works on the n vertices or the $2m$ directed edges but never on both at the same time. Since CUDA allows the threads to be allocated dynamically, the modified algorithm can be executed with at most $2m$ threads (processors). The PRAM algorithm depends on support for the CRCW model, which allows a CUDA implementation to take advantage of striding (see section 3.8). Thus the modified implementation can run with any number of threads including $p < \min(m, n)$.

Each of the steps in the original PRAM algorithm has been converted into a kernel as shown in the CUDA-based Algorithm 4.5. Using striding in each kernel, during each iteration every one of the p threads does either $\frac{n}{p}$ or $\frac{2m}{p}$ work (rather than the PRAM expectation of each thread doing 1 unit of work per iteration).

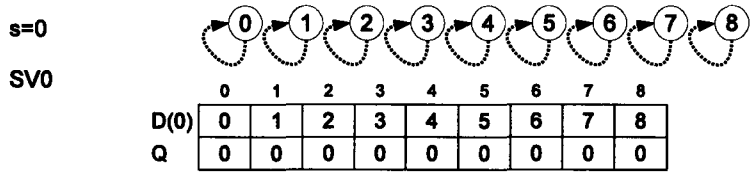
Algorithm 4.5 Shiloach and Viskhin's Algorithm for $p \leq n$ using Striding

```
Kernel SV0 using p<=n processors (processor index i = 0 ... p-1):
    j = i                -- initialize
    while j < n
        D(0) [j] = i
        Q [j] = 0
        j = j + p

s' = s = 1
while s = s':
    Kernel SV1 using p<=n processors (processor index i = 0 ... p-1):
        j = 1            -- step 1: Short-cut & Check/Mark
        while j < n
            D(s) [j] = D(s-1) [D(s-1) [j]]
            if D(s) [j] != D(s-1) [j] then:
                Q [D(s) [j]] = s
            j = j + p

    Kernel SV2 using p<=n processors (processor index i = 0 ... p-1):
        j = i            -- step 2: Hook edges
        while j < 2m
            (a, b) = edge j
            if D(s) [a] = D(s-1) [a] and D(s) [b] < D(s) [a] then:
                    D(s) [D(s) [a]] = D(s) [b]
                    Q [D(s) [b]] = s
            j = j + p

    Kernel SV3 using p<=n processors (processor index i = 0 ... p-1):
        j = i            -- step 3: Hook stagnant roots
        while j < 2m
            (a, b) = edge j
            if Q [D(s) [a]] < s and D(s) [a] = D(s) [D(s) [a]] and
                    D(s) [a] != D(s) [b] then:
                D(s) [D(s) [a]] = D(s) [b]
            j = j + p

    Kernel SV4 using p<=n processors (processor index i = 0 ... p-1):
        j = i            -- step 4: Short-cut again
        while j < n
            D(s) [j] = D(s) [D(s) [j]]
            j = j + p

    Kernel SV5 using p<=n processors (processor index i = 0 ... p-1):
        w = 0, j = i     -- step 5: Check for more work
        while j < n
            if Q[j] = s then
                w = w + 1
            j = j + p
        if w > 0 then
            s' = s' + 1

    s = s + 1
end
```

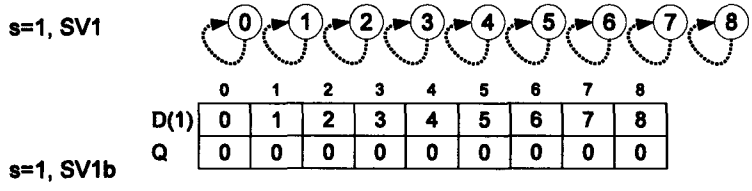
The goal of the algorithm, as described by the general outline in section 4.2.3, is to generate a star graph for each component such that every vertex in a component has a pointer to one root vertex. While in principle the algorithm builds a forest of trees, instead of tracking the children of vertices the algorithm tracks the parent of each vertex. Towards this end the algorithm maintains two arrays, $D_s[0 \ldots n-1]$ and $D_{s-1}[0 \ldots n-1]$, where s is the current iteration. D_0 is simply the result of the initialization step of the algorithm (kernel SV0) after which the algorithm starts at $s = 1$ and repeats until it has no more work to do. To help determine whether the algorithm is finished or if it should continue for another iteration, the algorithm maintains an array $Q[0 \ldots n-1]$ which is used to track for each vertex the last iteration where some change was made to the tree as a result of its state.

Example of Computing Connected Components in Parallel

The initialization step (kernel SV0) sets up D_0 such that each vertex is a tree with itself as its parent. The figure below shows the initialization of D_0 for the graph shown in Figure 4.4.

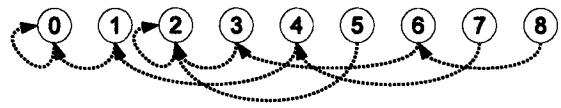


The first step (kernel SV1) within the each iteration attempts to shorten the trees, then checks if the current parent in D_s differs from the parent in D_{s-1} before it updates Q.



The next step (kernel SV2) examines the edges of the original graph and tries to hook the tree of one end-point vertex to the tree of the other. Since the graph has directional edges and other edges may have share one of the vertices, multiple processors may be hooking the same vertex's tree in parallel. The concurrent-write support ensures that only one will succeed, and the algorithm ensures that unsuccessful hookings do not matter.

s=1, SV2

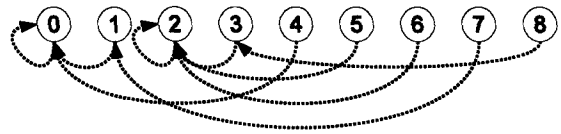


	0	1	2	3	4	5	6	7	8
D(1)	0	0	2	2	1	2	3	4	6
Q	0	1	0	1	1	1	1	1	1

s=1, SV3 **No change.**

The following step (kernel SV3) also examines the edges of the original graph, in this case looking to connect trees that are *stagnant* [17, p.59], where a *stagnant* tree is one that has not changed as a result of the earlier steps of this iteration. One more short-cutting is performed in the next step (kernel SV4), not necessarily because it is required for the correctness of the algorithm but because it helps reduce the number of iterations significantly.

s=1, SV4



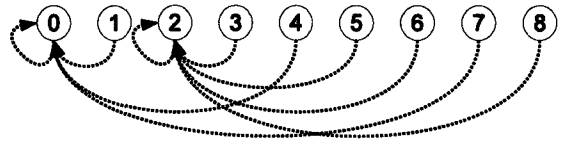
	0	1	2	3	4	5	6	7	8
D(1)	0	0	2	2	0	2	2	1	3
Q	0	1	0	1	1	1	1	1	1

s=1, SV5 **w=7 ⇨ s'=2**

The last step (kernel SV5) checks Q at the end of each iteration to determine if any work was done during the current iteration. If no work was done, then no more iterations are performed.

The diagrams above illustrate the changes for step $s = 1$. Since work was done hooking some of the vertices, the algorithm proceeds to the next iteration, the results of which are shown below. For the sample graph, the short-cutting by kernel SV1 finishes the work and we can see the final star graphs with two roots representing the two components.

s=2, SV1



	0	1	2	3	4	5	6	7	8
D(2)	0	0	2	2	0	2	2	0	2
Q	0	1	0	1	1	1	1	2	2

s=2, SV2 **No change.**

s=2, SV3 **No change.**

s=2, SV4 **No change.**

s=2, SV5 **w=2 ⇨ s'=3**

Since some work was done in this iteration, there is one more iteration that will be executed (not shown) where each kernel does no further work, which then causes the algorithm to terminate.

Chapter 5

Parallel List Ranking and CUDA

We begin our study of the implementation of parallel graph algorithms for a GPU with a basic operation required by nearly all (parallel or sequential) graph algorithms: traversing a linked list of edges. Consider a linked list of length n represented by an array $succ\,[0..n-1]$ where each $succ[i]$ points to the next element in the linked list. The first element of the linked list is $succ[0]$ and the last element has the property $succ[j] = j$. The ranks of all list elements (distances to the last element of the list) are reported as an array $rank\,[0..n-1]$, and computing the ranking of all the nodes in a list requires a traversal of the entire list from the head of the list to its tail by traversing all the links.

In this chapter we present the details of the different list ranking implementations listed below, following which we examine the results comparing the performance differences between the algorithms. In addition to the sequential algorithm, we will study a GPU implementation (see section 4.1.3 for a review) of the straightforward pointer jumping algorithm for the PRAM, also known as *Wylie's Algorithm* [16, p. 64] [23]. This algorithm perform well with the GPU but, similar to a preliminary result in [14], suffers from the fact that it requires $O(n \log n)$ work. We then look at our implementation (see section 4.1.4 for a review) of an adaptation of the *parallel random splitter* algorithm for PRAM made by Reid-Miller for parallel list ranking on a Cray C90[15]. The Cray C90 is a vector processor and shares some features with GPUs such as SIMD style data parallelism. We implement this parallel algorithm on the CPU as well to study how well it fares, unlike the sequential algorithm, using all the cores in a CPU.

5.1 Implementation

5.1.1 Sequential CPU List Ranking

The sequential list ranking implementation is very simple: since the number of elements is known, the implementation walks the list from the first element to the last following each successor, counting down from n to determine each element's rank (see Listing 5.1).

Listing 5.1 Sequential List Ranking Implementation

```
1  void SQ_LR_doListRanking (
2      int32_t *succ_h, int32_t n, int32_t *rank_h)
3  {
4      int32_t cur = 0;
5      do {
6          rank_h [cur] = -- n;
7          cur = succ_h [cur];
8      } while (succ_h [cur] != cur && tmp > 0);
9      rank_h [cur] = 0;
10 }
```

5.1.2 Parallel CUDA Wylie's List Ranking

The CUDA-based implementation of Wylie's list ranking algorithm is made up of three kernels:

- The initialization kernel `LR_PJ_Init()` which sets up the preliminary values for the data structure that is used to compute the rank of each element and track the pointer jumps (see Listing 5.2).
- The pointer jumping kernel `LR_PJ_PtrJumping()` which performs one iteration of each element updating its short-cut pointer. This kernel also tracks if any work to help the host know when to stop iterating (see Listing 5.3).
- The data extraction kernel `LR_PJ_ExtractRank()` which extracts the rank out of the packed data structure that is used to optimize the performance of the pointer jumping (see Listing 5.4).

Implementation Details of the Initialization Kernel

Listing 5.2 Parallel Pointer Jumping List Ranking Initialization Kernel, in C/CUDA

```
1  __global__ void LR_PJ_Init (
2      int32_t *succ, uint64_t *ranklast, int32_t n int32_t p)
3  {
4      ranklast_t r;
5      int32_t ix = blockIdx.x * blockDim.x + threadIdx.x;
6
7      while (ix < n) {
8          r.per.rank = (succ [ix] != ix);
9          r.per.last = ((succ [ix] + 1) * r.per.rank) - 1;
10         ranklast [ix] = r.combo;     // single 64-bit write
11         ix += p;                     // stride by p to next node
12     }
13 }
```

The algorithm expects $n > p$ elements in the list, and thus uses striding to take advantage of memory access coalescing The original algorithm updates two arrays during the pointer jumping: the *last* array and the *rank* array This implementation however uses a packed 64-bit array that encodes the the rank and last pointer. The packing and unpacking are done using C macros (see below) that make the code more readable by hiding the bitwise arithmetic involved.

```
#define packx(last,rank) (\
    (((uint64_t) (rank)) << 32) |    /* upper 32 bits */ \
    ((uint64_t) (last)))             /* lower 32 bits */
#define xrank(p) ((uint32_t) ((p) >> 32))
#define xlast(p) ((uint32_t) ((p) & 0xffffffff))
```

The kernels themselves use the packed data type only as a register, and use it to manipulate the values in register memory. Each value is read once from, and written once to, global memory as a single 64-bit operation

Implementation Details of the Pointer Jumping Kernel

As discussed in section 3.6.1, on a CUDA 1.2 or better system up to 128 bytes can be read or written as a single transaction. There are two benefits of packing the two values into a single 64-bit value: First, the use of 64-bits to encode two 32-bits reduces the total number of memory transactions. While the amount of memory read and written doesn't change, with two arrays there are twice the number of transactions on the memory access network.

Second, the algorithm processes the rank and the last pointer together, and by using a packed value the implementation can atomically read and write 64-bit values. This allows us to ensure data integrity when running the kernel over multiple thread-blocks.

Listing 5.3 Parallel Pointer Jumping List Ranking Kernel, in C/CUDA

```
1  __global__ void LR_PJ_PtrJumping (
2      int32_t *succ, uint64_t *ranklast, int32_t *work,
3      int32_t n, int32_t p)
4  {
5      ranklast_t r, r2;
6      int32_t w = 0, ix = blockIdx.x * blockDim.x + threadIdx.x;
7      while (ix < n) {
8          r.combo = ranklast [ix];
9          if (r.per.last > -1) {
10             r2.combo = ranklast [r.per.last];    // read 64-bits
11             r.per.rank += r2.per.rank;
12             r.per.last = r2.per.last;
13             ranklast [ix] = r.combo;             // write 64-bits
14             w++;     // count work done
15         }
16         ix += p;     // stride by p to next node
17     }
18     if (w > 0) *work = 1;    // if any work done, mark
19 }
```

Implementation Details of the List Rank Extraction Kernel

Listing 5.4 Parallel Pointer Jumping List Ranking Rank Extraction Kernel, in C/CUDA

```
1  __global__ void LR_PJ_ExtractRank (
2      uint64_t *ranklast, int32_t *rank, int32_t n, int32_t p)
3  {
4      ranklast_t r;
5      int32_t ix = blockIdx.x * blockDim.x + threadIdx.x;
6      while (ix < n) {
7          r.combo = ranklast [ix];
8          rank [ix] = r.per.rank;
9          ix += p;                          // stride by p to next node
10     }
11 }
```

Packing the rank and last pointer data in device memory means that the rank data has to be extracted before the host can use it. The extraction is done by a separate kernel which, in parallel, extracts the rank from each packed value.

The pointer jumping kernel needs to be invoked iteratively up to lg (n) times by the host application. Since we know that often less iterations are needed to complete the ranking, the pointer jumping kernel counts the number of short-cuts it performs. Before exiting, it writes 1 to a common address if, and only if, any pointer jumping was done.

With the implementation sub-divided into multiple kernels, additional code is needed to invoke them properly. Listing 5.5 illustrates how the kernels are invoked by the host application.

Implementation Details of the CPU Host Function

Listing 5.5 Host invoking the Pointer Jumping kernels, in C/CUDA

```
1  void LR_PJ_host (...) {
2      cudaMemcpy (..., cudaMemcpyHostToDevice);
3      LR_PJ_Init <<< q, 256 >>> (...);
4      do {
5          cudaMemset (work_d, 0, ...);
6          LR_PJ_PtrJumping <<< q, 256 >>> (..., work_d, ...);
7          cudaMemcpy (&work_h, work_d, ..., cudaMemcpyDeviceToHost);
8      } while (work_h > 0)
9      LR_PJ_ExtractRank <<< q, 256 >>> (...);
10     cudaMemcpy (..., cudaMemcpyDeviceToHost);
11 }
```

The parameter q represents the number of thread blocks used to execute the kernel, each block using 256 threads. While CUDA allows up to 768 threads per block, our experiments showed that using more then 256 threads did not yield any improvement in performance, and often resulted in a reduction in performance.

5.1.3 Parallel CUDA Random Splitter List Ranking

The CUDA-based implementation of Random Splitter list ranking algorithm is made up of four kernels:

- The initialization kernel `CU_RS_LR_64b_PickRandomSplitters()` which sets up the random splitters (see Listing 5.7).
- The sub-list counting kernel `CU_RS_LR_64b_CountSplitterSubLists()` which, for each splitter, walks its sub-list and computes a local ranking (see Listing 5.8), along the way constructing a linked list of the splitters.
- The simplified pointer jumping kernel `CU_PJ_LR_Simple()` which is used to rank the splitters using their local ranks to compute their absolute ranking (see Listing 5.9).

- The rank aggregation kernel `CU_RS_LR_64b_AggregateLocalRanks()` which combines the local rank of each node with the absolute rank of its splitter to compute the rank of each node (see Listing 5.10).

Implementation Details of the CPU Host Function

Listing 5.6 Host invoking the Random Splitter kernels, in C/CUDA

```
1  void CU_RS_LR_64b_host (...) {
2      // copy linked list to device memory
3      cudaMemcpy (..., cudaMemcpyHostToDevice);
4      // initialize owner array
5      cudaMemset (rankowner_d, 0xff, n * sizeof (uint64_t));   // RS1
6      CU_RS_LR_64b_PickRandomSplitters <<<q,256>>> (...);      // RS2
7      CU_RS_LR_64b_CountSplitterSubLists <<<q,256>>> (...);    //RS3
8      // rank the ruler list using parallel Wylie
9      CU_PJ_LR_Simple <<<1,256>>> (...);                       // RS4
10     // use as many threads as possible to aggregate;    i.e. RS5
11     CU_RS_LR_64b_AggregateLocalRank <<<4000000/256,256>>> (...);
12     // copy ranks to host memory
13     cudaMemcpy (..., cudaMemcpyDeviceToHost);
14 }
```

Listing 5.6 illustrates the host function that invokes each of the kernels. The implementation relies on several working data structures in addition to the successor array *succ* and the rank array *rank* used to store the final ranks of each of the nodes. The array *owner* $[0 \ldots n-1] = \{0 \ldots r-1\}$ is used to determine the index of the splitter to whose sub-list each vertex in the input list belongs. The array *spsucc* $[0 \ldots r-1]$ is used to store the linked list of splitters and the array *sprank* $[0 \ldots r-1]$ is used to store the ranks of the splitters in the splitter list.

The implementation details shown here represent an implementation that uses a packed 64-bit array, like the one used by the CUDA Pointer Jumping implementation shown earlier. While in this case the *owner* [] and *rank* [] arrays are combined, the reason is the same: to combine two transactions into one to improve performance. The kernels use the following C macros which hide the bitwise arithmetic associated with packing and unpacking the *owner* and *rank* components. Initializing the packed array is done using CUDA's API for clearing memory (viz. `cudaMemset()`) in parallel rather than writing our own kernel to do the same work.

```
#define packx(owner,rank) (
    (((uint64_t) (rank)) << 32) | ((uint64_t) (owner)))
#define xrank(p) ((uint32_t) ((p) >> 32))          // high 32 bits
```

```
#define xlast(p) ((uint32_t) ((p) & 0xffffffff))     // low 32 bits
```

We also implemented a 48-bit version where the owner and rank fields are not packed, rather tracking two separate arrays: *owner* with 16-bit values and *rank* with 32-bit values. This alternate implementation (not listed here) reduces the total amount device memory accessed during the execution of the algorithm by 25% while doubling the number of access transactions. When we analyze the performance results of the different list raking algorithms in section 5.2, we include the results from both implementations.

Implementation Details of the Random Splitter Selection Kernel

Listing 5.7 Parallel Random Splitter Selection Kernel, in C/CUDA

```
1  __global__ void CU_RS_LR_64b_PickRandomSplitters (
2      uint32_t seed,
3      int32_t *splitters,      // p-length array of splitters
4      uint64_t *rankowner,     // n-length packed rank/owner array
5      int32_t n, int32_t p)
6  {
7      int32_t pIx, workN, workIx0, rem, sel;
8      RAND_t  rng;     // structure for RNG
9      pIx = blockIdx.x * blockDim.x + threadIdx.x;
10     if (pIx < p) {
11         // partition list evenly across threads
12         rem = n % p;
13         workN = n / p;           // start with division of work
14         workIx0 = pIx * workN;   // and use that to index start
15         workN += (pIx < rem);    // add 1 when pIx < rem
16         workIx0 += min (pIx, rem);  // add lesser of pIx or rem
17         // pick random node as local "splitter", unless first node
18         if (workIx0 == 0)
19             sel = 0;
20         else {
21             // initialize RNG and generate a single random value
22             DCU_RandInit (seed, pIx, &rng);
23             sel = workIx0 + (DCU_RandNext (&rng) * (workN - 1));
24         }
25         splitters [pIx] = sel;
26         rankowner [sel] = packx (pIx, 0);    // remember splitter
27     }
28 }
```

The splitter selection kernel (see Listing 5.7) uses a random number generator with a very high period that we implemented (see section A for details) for both the CPU and GPU platforms to select a single splitter from each thread's evenly partitioned array of links.

Given n elements in the array, each thread gets a partition of $\frac{n}{p}$ sequential elements from which it selects a random index. Since only a single random number is needed per thread, the host supplies a global seed which is then altered by each thread using its thread index to ensure that each thread gets a distinct seed from that of any other thread. The RNG itself provides sufficient guarantees of randomness for values generated using different seeds that this approach, in effect, allowed us to generate sufficiently random numbers in parallel.

The *splitters* array is used to note the splitter index for each thread, which is then used by the other kernels to identify the splitter associated with each thread. Each splitter's local rank is set to zero and owner is set to the thread index. Since there is a splitter per thread we can use the thread index as the splitter index and thus the owner index.

Implementation Details of the Sub-List Counting Kernel

Listing 5.8 Parallel Sub-List Counting Kernel, in C/CUDA

```
1  __global__ void CU_RS_LR_64b_CountSplitterSubLists (
2      uint32_t *succ, int32_t *splitters, uint64_t *rankowner,
3      int32_t *splitter_succ_p, int32_t *splitter_rank_p,
4      int32_t n, int32_t p)
5  {
6      int32_t pIx, splitter, j, prevj;
7      uint32_t dist;
8      uint64_t mk;
9      pIx = blockIdx.x * blockDim.x + threadIdx.x;
10     if (pIx < p) {
11         // identify the splitter, then walk and own its sub-lists
12         splitter = splitters [pIx];
13         dist = 1;
14         prevj = splitter;
15         j = succ [splitter];
16         while ((mk = rankowner [j]) == OWNERZERO) {
17             rankowner [j] = packx (pIx, dist);
18             j = succ [prevj = j];
19             dist ++;
20         }
21         // reduce dist by 1 if we've encountered the last node
22         dist -= (prevj == j);
23         // setup the p-length splitter linked listto rank first
24         splitter_succ_p [pIx] = xownr (mk); // the next splitter
25         splitter_rank_p [pIx] = dist;       // and distance to it
26     }
27 }
```

The most labour-intensive part of the algorithm is walking each splitter's sub-list and locally ranking each element in that sub-list. The kernel shown in Listing 5.8 when executed

by each thread uses the thread index to identify the splitter, following which each thread sequentially ranks the sub-list by walking along the successors starting at the splitter. During initialization, every element's *owner* was set to a special value of $0xffffffff$. During splitter selection, every splitter's *owner* was set to the appropriate thread index, a value much smaller than the special value. Thus each thread knows that a successor belongs to its splitter's sub-list or not by checking the *owner* value for the element.

As the kernel walks each splitter's sub-list, it counts the distance from the splitter and sets the element's *rank* value, and sets the *owner* value to the thread index. Thus when the kernel has finished, each splitter's sub-list has been ranked relative to the splitter and every sub-list element knows its owner.

At this point each thread stores the number of elements in its sub-list in *splitterrank* and the owner of the first element that belongs to another splitter as its successor in *splittersucc*. This effectively constructs a linked list of the splitters, with each splitter's sub-list length as an initial rank for each splitter.

This kernel is labour-intensive not because it does $O(n)$ work but because it results in a very large number of un-coalesced memory transactions. In a completely randomly linked list, every element's successor is at a random index in the overall successor array, and when p threads access p elements in parallel and then select their successors, the threads in a particular block are very unlikely to access elements at indices near enough to each other to take advantage of coalescing. Thus with high probability every thread's access results in a unique transaction, in turn resulting in $O(n)$ discrete memory access transactions. As discussed in section 3.6.1, CUDA performance is maximal when memory access is coalesced.

Implementation Details of the Simple Pointer-Jumping List Ranking Kernel

Listing 5.9 shows a single-kernel implementation of Wylie's pointer-jumping list ranking algorithm. Unlike the parallel multi-thread-block implementation that we looked at in section 5.1.2, this one uses a simplified version of parallel pointer jumping that can be executed using only one thread-block, and as a result is suitable for ranking small linked lists. Since there are as many splitters as threads in the Random Splitter-based list ranking algorithm, there are thousands of threads. Recall that we select p in such a way that $O(p \log p) < O(n)$. Thus the linked list of splitters, with its p elements where p is quite small relative to n, is large enough to warrant a CUDA-based ranking algorithm but too small to justify using the more-complex multi-thread-block pointer jumping algorithm, both to maximize the use of the GPU and to avoid copying the splitters to and from host memory.

Since this step needs to rank the linked list of splitters with the caveat that we start with each splitter's sub-list length as the initial rank value instead of zero, this pointer jumping implementation is slightly specialized. It expects a boolean *userank* parameter that it uses

to determine whether to use, or initialize to zero, the initial values in *rank*. When ranking the linked list of splitters, the kernel is invoked with a *true* value to request that it use the incoming values as the initial ranks. Thus when the splitter ranking finishes, each splitter's rank is in fact its final rank relative to the original n-element linked list.

Listing 5.9 Parallel Pointer Jumping List Ranking with One Kernel, in C/CUDA

```
1  __global__ void CU_PJ_LR_Simple (
2      int32_t *succ, int32_t *rank, int32_t *last, int32_t n,
3      int32_t p, bool userank)
4  {
5      int32_t pIx, i, j, dist, more;
6      if (blockIdx.x == 0) {  // ensure only one thread-block is used
7          i = pIx = blockIdx.x * blockDim.x + threadIdx.x;
8          for (; i < n; i += p) {
9              j = succ [i];
10             dist = (j != i);
11             last [i] = ((j + 1) * dist) - 1;
12             if (!userank) rank [i] = dist;
13         }
14         __syncthreads();          // synchronize after init
15         do {
16             more = 0;
17             for (i = pIx; i < n; i += p) {
18                 j = last [i];
19                 if (j >= 0) {
20                     more ++;
21                     dist = rank [i] + rank [ j ];    // read phase
22                     j = last [ j ];
23                     __syncthreads();          // sync before write
24                     last [i] = j;
25                     rank [i] = dist;
26                 }
27                 __syncthreads();              // sync before next read
28             }
29         } while (more > 0);
30     }
31 }
```

Implementation Details of the Rank Aggregation Kernel

The last kernel (see Listing 5.10) is an example of when coalesced memory access can significantly maximize the performance in a CUDA system. At this point each splitter's global rank within the linked list has been established. Since every element knows its *owner*, the aggregation kernel simply strides over the successor array reading the *rankowner* array for each element and using the splitter's global rank to compute each element's global rank

based on each element's local rank relative to the splitter. Since this algorithm doesn't follow the successor links, memory access is maximally coalesced as p threads, over $\frac{n}{p}$ iterations, concurrently access p sequential elements each iteration.

Listing 5.10 Parallel Rank Aggregation Kernel, in C/CUDA

```
1  __global__ void CU_RS_LR_64b_AggregateLocalRanks (
2      uint64_t *rankowner,          // n-length array of rank
3      uint32_t *rank,               // n-length array of rank
4      int32_t *splitter_rank_p,     // r-length array of known splitter ranks
5      int32_t n, int32_t p)
6  {
7      uint64_t mk;
8      int32_t pIx = blockIdx.x * blockDim.x + threadIdx.x;
9      while (pIx < n) {
10         mk = rankowner [pIx];
11         rank [pIx] = splitter_rank_p [xownr (mk)] - xrank(mk);
12         pIx += p;
13     }
14 }
```

Each kernel reads and writes different amounts of data, and there is the additional difference in the amount between the 48-bit and 64-bit implementations of the Random Splitter-based list ranking algorithm for CUDA. As well the two implementations use different numbers of memory access transactions to do their work. Table 5.1 shows the number of bits read and written per relevant kernel as well as the number of transactions used to read and write per relevant kernel, and this information is referenced later in this chapter when the results of the experiments using these implementations are analyzed.

		64-bit Impl				48-bit Impl			
		R bits	R tr	W bits	W tr	R bits	R tr	W bits	W tr
RS2	Pick Random Splitters	0	0	$96r$	r	0	0	$48r$	r
RS3	Count Splitter Sub-Lists	$96n$	$2n$	$64n$	n	$48n$	$2n$	$48n$	$2n$
RS5	Aggregate Local Ranks	$96n$	$2n$	$32n$	n	$80n$	$3n$	$32n$	n

Table 5.1: CUDA Random Splitter Implementations Read/Write Bits & Transactions per Kernel

5.1.4 Parallel CPU Random Splitter List Ranking

To assist and simplify the multi-threaded CPU implementations of parallel algorithms, a generic concurrent API was defined and implemented using the POSIX-based threading library (viz. pthreads) for multi-threading, as outlined below.

- A parallel execution context is defined by the opaque handle type **HPAR**.

- A context based on a specific number of threads can be created by calling the function `bool par_create(int nthreads, HPAR *hpar)`. On success it returns `true` and stores the context handle into the supplied pointer.
- A parallel function can be executed concurrently on all the threads associated with a context by calling `bool par_for_p(HPAR hpar, void *data, par_for_fun_t func)`. This function executes the supplied function on each thread concurrently, returning only when all the threads have finished their work.
- Barrier synchronization can be achieved CUDA-style by breaking up a parallel implementation into separate kernel functions and invoking each one successively using `par_for_p()`.
- Barrier synchronization can also be achieved **within** the execution of the parallel function by calling `void par_for_p_sync(HPAR hpar)`. This function causes all threads to reach the barrier before proceeding past the call.
- When finished, the application can cleanup the context and release the underlying resources by calling `void par_destroy(HPAR hpar)`

The CPU multi-threaded implementation is shown in Listing 5.11, and while the CPU-based multi-threaded implementation continues to use the same steps that were implemented as CUDA kernels, the use CPU multi-threading results in several differences in the implementation details.

Since the concurrency API mentioned earlier allows us to do barrier synchronization from within the threads, the implementation is written as a single parallel function (unlike the CUDA implementation in the following section). The five kernels defined in Algorithm 4.3 on page 35 are implemented within the single function, separated by a call to `par_for_p_sync()` to ensure synchronization (see lines 21, 38 and 53).

There is no barrier synchronization between RS1 and RS2 in the CPU multi-threaded implementation. This is correct because each thread's selected *ruler* index is within the portion of the *owner* array initialized by the thread, and picking the random ruler does not rely on the initialization of the partitions of the data by the the other threads. In fact the same is true for the CUDA implementation; however we chose to use a CUDA-specific API to initialize the data and as a result gained an implicit barrier between the two steps. If we had chosen to perform the initialization within the random splitter selection kernel, we would have the exact same situation as in the CPU implementation.

Listing 5.11 Parallel Random Splitter List Ranking, Multi-threaded C Implementation.

```
1  void MTH_RS_LR_par_fun (int pIx, int p, CALL_t *call)
2  {
3      register int32_t *succ = call->succ;
4      register uint16_t *rsucc = call->ruler_succ, *owner = call->owner;
5      register int32_t *rrank = call->ruler_rank, *rank = call->rank;
6      int32_t workN, workIx0, rem, ruler;
7      /* ------ compute the partitions for each thread ------- */
8      rem = call->n % p;
9      workN = call->n / p;          // start with simple division of work
10     workIx0 = pIx * workN;        // and use that to index start of work load
11     workN += (pIx < rem);         // add 1 to each node whose pIx < rem
12     workIx0 += min (pIx, rem);    // add lesser of pIx or rem to start
13     /* ------ RS1, RS2 - Init & Pick Random Rulers --------- */
14     memset (owner + workIx0, 0xffff, workN * sizeof (uint16_t));
15     ruler = (workIx0 == 0) ? 0
16                 : workIx0 + (call->th [pIx].urnd * (workN - 1));
17     owner [ruler] = pIx;
18     par_for_p_sync (call->hpar);      // BARRIER
19     /* ------ RS3 - Count Rulers' Sub-Lists -------------- */
20     int32_t dist = 1, prevj = ruler, j = succ [ruler];
21     rank [ruler] = 0;    // since separate array, ruler's rank = 0
22     while (owner [j] == 0xffff) {
23         owner [j] = pIx;
24         rank [j] = dist ++;
25         j = succ [prevj = j];
26     }
27     dist -= (prevj == j);    // reduce dist by 1 if we see the last node
28     // setup the p-length ruler linked list, which we need to rank first
29     rsucc [pIx] = owner [j];              // pIx of the next ruler
30     rrank [pIx] = dist;                // and distance to ruler
31     par_for_p_sync (call->hpar);      // BARRIER
32     /* ------ RS4 - Rank the Ruler List ------------------ */
33     if (pIx == 0) { // use seq ranking with one thread
34         register int32_t tmp, sum = call->n - 1;
35         j = 0;
36         while (rsucc [j] != j) {
37             tmp = rrank [j];
38             rrank [j] = sum;
39             sum -= tmp;
40             j = rsucc [j];
41         }
42         rrank [j] = sum;
43     }
44     par_for_p_sync (call->hpar);      // BARRIER
45     /* ------ RS5 - Aggregate Local Ranks with Rulers ----- */
46     for (j = workIx0; j < workIx0 + workN; j++)
47         rank [j] = rrank [owner [j]] - rank [j];
48 }
```

This implementation uses a 16-bit owner array that limits the number of threads that can be used to run this implementation to 65536. Since the maximum number of cores in a multi-core CPU are typically less than 100 and rarely only slightly higher, this does not limit the usefulness of the implementation. Furthermore, as discussed in section 4.1.4 on page 34, the algorithm works best with substantially fewer splitters (and thus threads) than elements in the list.

Under the same restriction that the number of cores available in a multi-core CPU is quite small, the implementation uses sequential list ranking to rank the list of rulers (see RS4 in the implementation). The size of the ruler list did not warrant the overhead of parallel pointer jumping.

5.2 Run Time Comparisons

The CPU implementations were executed on a system with a quad-core CPU while the CUDA implementations were executed using a nVIDIA GTX 260 GPU installed on a quad-core CPU host system (see Appendix A on page 97 for details).

5.2.1 Comparing All Implementations

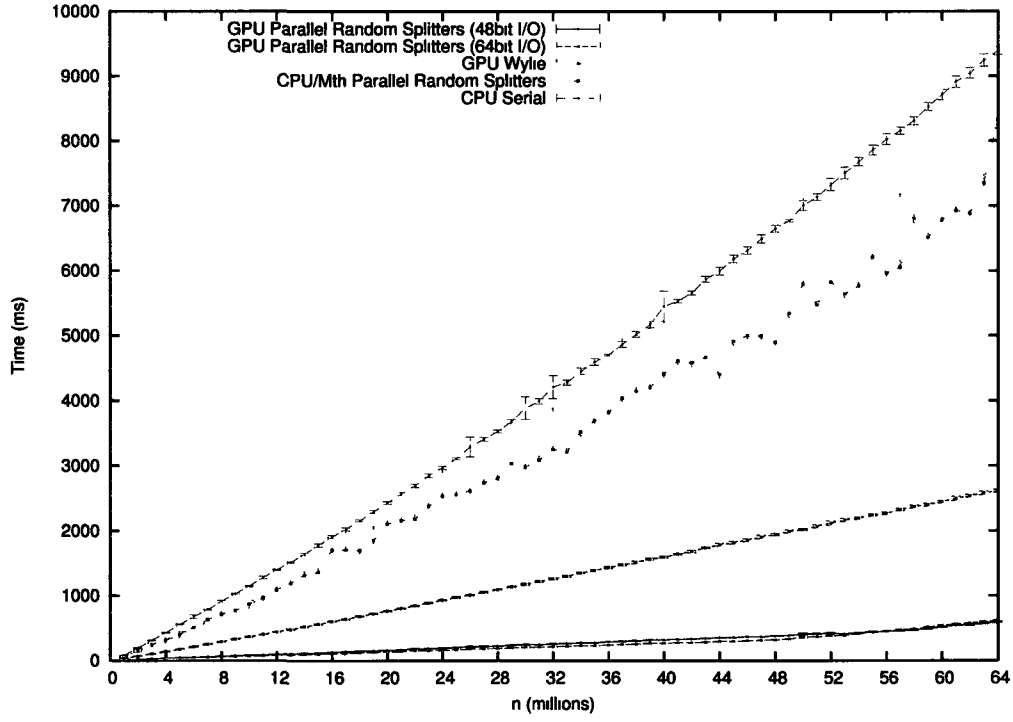


Figure 5.1: Comparing All List-Ranking Implementations

The graph in Figure 5.1 shows a comparison of the runtime performance of all the list ranking implementations described earlier in section 5.1. By comparing the curves for the different implementations we can see that the CUDA implementations out-perform the CPU implementations by a significant margin. This is especially relevant since even the parallel pointer jumping list ranking implementation in CUDA, which does $O(n \log n)$ work, is substantially faster than the best sequential list ranking CPU implementation that does $O(n)$ work. This is a perfect demonstration of the advantage of the GPU's parallel computation.

The margin of improvement between the single-core sequential CPU implementation and the 4-core parallel CPU implementation is quite small, less than 25%. Since both implementations perform $O(n)$ work, any gains from using the extra CPU cores appears to have been negated by the overhead of dealing with the randomized memory accesses of the experimental data and resulting memory cache conflicts within the cores. The large error bars in the parallel CPU implementation's curve is indicative of the variability in performance between the different same-size lists.

Among the CUDA implementations, as expected, the random splitter-based list ranking implementations, both the 64-bit and 48-bit versions, clearly out-perform the pointer jumping algorithm. This is expected as the random-splitter-based algorithm does as much as work in its entirety as the first iteration of the pointer-jumping algorithm.

5.2.2 Comparing Scaling Character of the CUDA Implementations

The graph in Figure 5.2 looks at CUDA-based list ranking algorithms only, expressing the performance run-time per linked list element. The x-axis represents the size of linked lists, n, used for this experiment; the y-axis is $\frac{time}{n}$. The curve is used to illustrate some important points about the performance of the CUDA algorithms.

We know from the algorithm itself that the pointer jumping implementation does $O(n \log n)$ work. This is clearly shown by the curve for the implementation which expresses a logarithmic scaling character, i.e. $\frac{time}{n} = \frac{O(n \log n)}{n} \approx O(\log n)$, when we look at the run-time per element. Similarly, as expected from the random splitter based implementation's $O(n)$ work-load, the curve for both implementations show a flat scaling character, i.e. $\frac{time}{n} = \frac{O(n)}{n} \approx O(1)$, indicative of unit work per element regardless of the size of the list. The random splitter based implementations show a curious inflection in their run time curves as the data size increases, which is analyzed in greater detail in section 5.2.6.

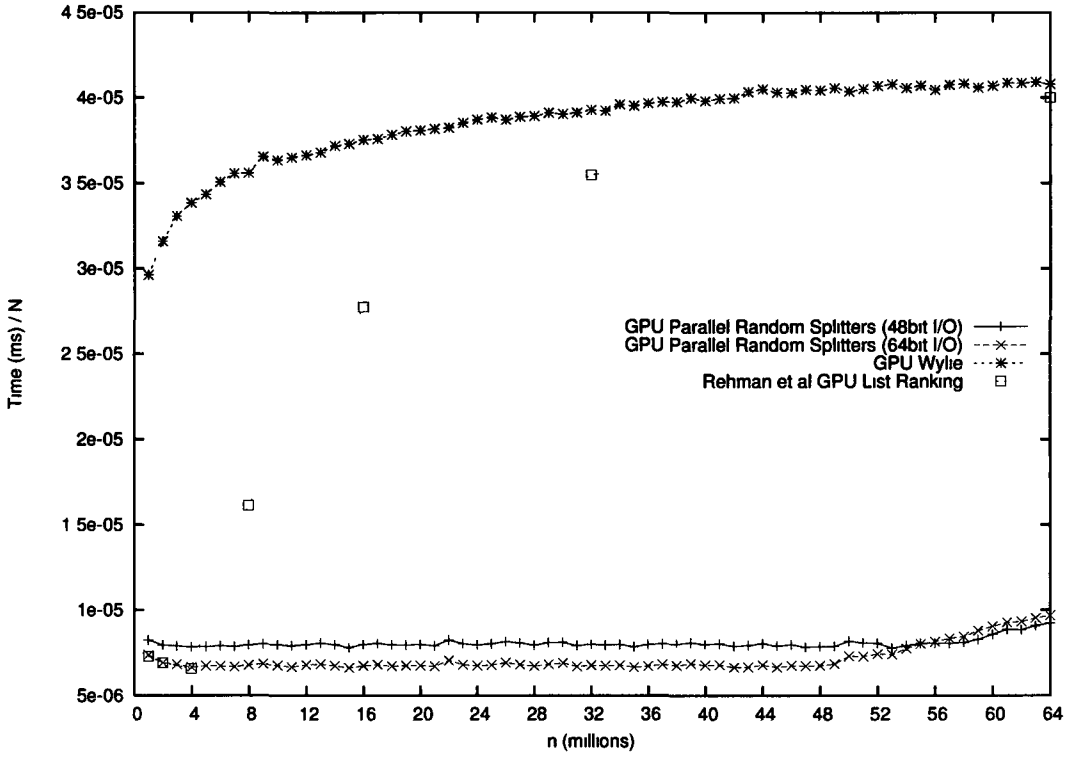


Figure 5.2: Comparing Scaling Character of the CUDA-based List Ranking Implementations

In addition to our three CUDA implementation curves, the graph includes a curve based on published data [14] by Rehman et al who claim to have an optimal parallel list ranking algorithm for CUDA. Since we have not implemented their algorithm, as we have not been able to obtain details regarding their implementation, we have used their published results in the graph. Their results included data points on run-time performance for a subset of the link list sizes that were used in our experiments. As shown in the graph, while their algorithm does appear to out-perform all three of our CUDA implementations for $n < 4M$ elements, with larger linked lists the performance exhibits a logarithmic scaling character, indicating that their recursive algorithm is in fact an $O(n \log n)$ algorithm. Clearly the random splitter-based list ranking algorithm is the best performer.

5.2.3 Comparing Relative Speedup of the CUDA Implementations

The graph in Figure 5.3 looks at CUDA-based list ranking implementations only, expressing the relative speedup of the three CUDA implementations as we increase the number the thread-blocks used to rank lists of size $n = 64M$. The x-axis represents the number of thread-blocks, q, used for this experiment; the y-axis is the speedup relative to the run-time performance when $q = 1$. Thus a speedup of 1 represents the performance when $q = 1$.

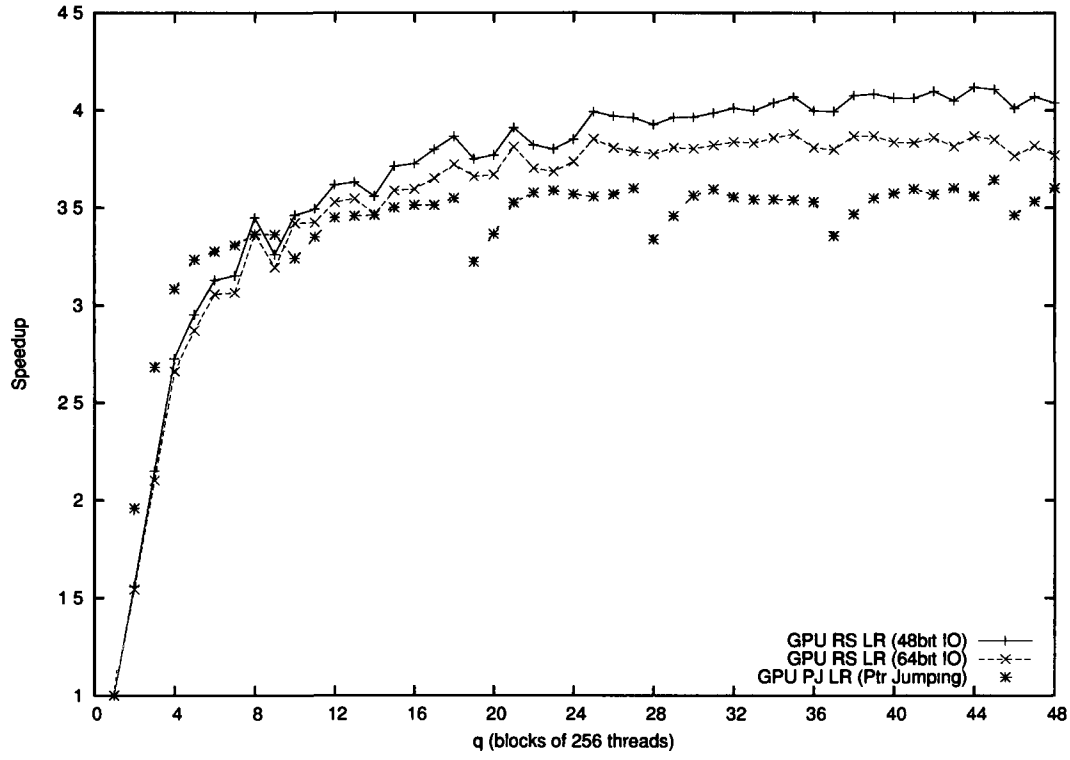


Figure 5.3: Relative Speed-up Over Multiple CUDA Thread-Blocks

The curve is used to study the speedup that can be achieved in a CUDA system when using an increasing number of thread-blocks. As described in section 3.3, in a CUDA system each thread-block is executed by one SM regardless of the number of threads per block. Thus increasing the number of thread-blocks implies increasing use of SMs, which in turn should be reflected in the speedup curve. The curve shows that regardless of the number of thread-blocks used, and regardless of the algorithm we've implemented, the GTX 260 GPU cannot offer more than a 4x speedup relative to $q = 1$ when ranking a randomized linked list. The reduced speedup is a direct result of the overhead of accessing the randomly organized links in the linked lists, the type of work that dominates all three implementations.

CUDA focuses on hiding memory access latency, and this shows when we compare the 64-bit and 48-bit implementations of the random splitter-based algorithm. The 48-bit implementation, which accesses less bytes of memory but requires more access transactions due to the separate *owner* and *rank* arrays, sees better speedup than the 64-bit implementation which accesses more memory but uses less transactions. This is not as counter-intuitive as it seems. The separate *owner* and *rank* values take up less space individually, and thus a single coalesced operation will yield more elements (e.g. twice as many 32-bit values can fit in a coalesced transaction than 64-bit values). In addition, the extra transactions as a

result of not packing the two elements allows CUDA to perform more latency hiding between transactions.

It is worth noting that the worst speedup is for the pointer-jumping implementation which accesses substantially more memory and generates a much higher number of transactions than either of the other two implementations.

5.2.4 Comparing Relative Speedup of CUDA Random Splitter Implementation Kernels

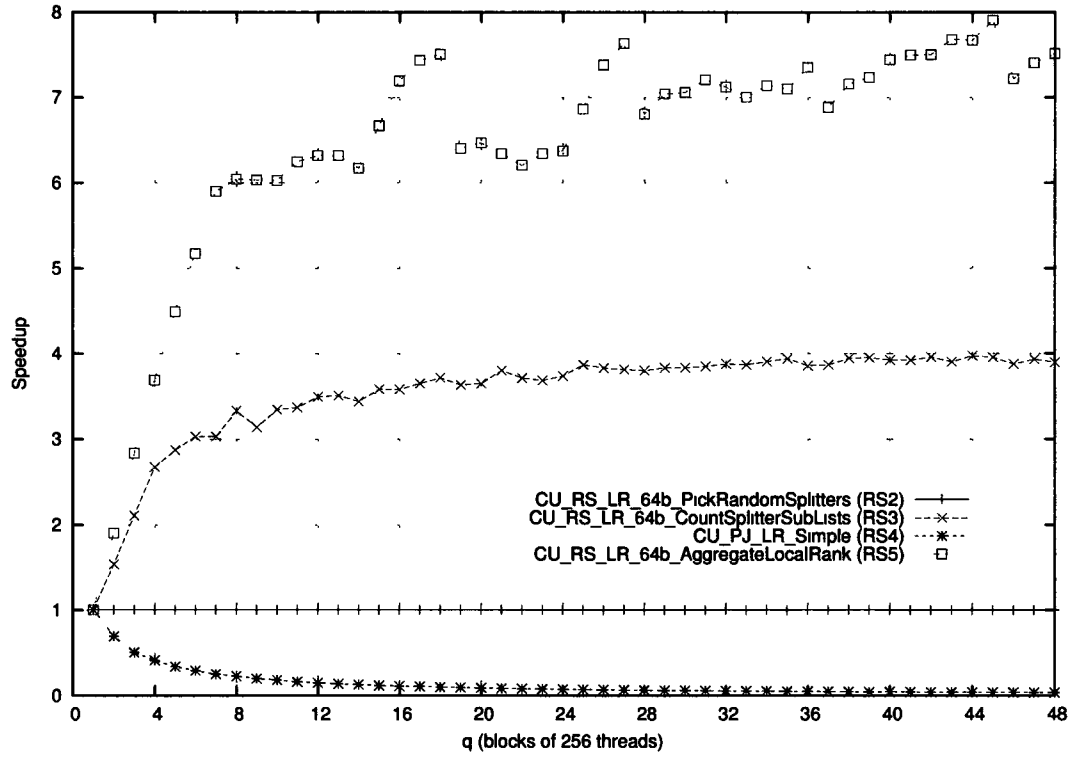


Figure 5.4: Relative Speed-up of Random Splitter List Ranking Kernels over Multiple CUDA Thread-Blocks

The graph in Figure 5.4 looks at the relative speedup of the individual kernels in the CUDA Random Splitter-based list-ranking implementation. When we looked at the relative speedup across the CUDA list-ranking algorithms (see section 5.2.3) it was apparent that overall speedup did not reflect the increase in the number of thread-blocks (and thus physical SMs). To better understand the distribution of speedup within the implementation, we look at the performance of the kernels separately.

- RS2, the random splitter selection kernel (see Listing 5.7), shows no relative speedup

increase with q, which is expected since each of the p threads (acting for each of the p splitters) does $O(1)$ work. Even as p increases with q (remember that we use 256 threads for each thread-block, thus $p = 256q$) every thread in parallel does the same amount of work.

- RS3, the per-splitter sub-list counting kernel (see Listing 5.8), shows a relative speedup curve that matches that of the overall algorithm. As we've discussed before, this kernel does the most time-consuming work and as a result it dominates the performance of the entire algorithm.

- RS4, the sub-list ranking kernel (see Listing 5.9), shows a declining speedup that reflects the nature of the pointer jumping algorithm as affected by the increasing number of p splitters that need to be ranked. This kernel is run using one thread-block regardless of q and thus the relative speedup curve reflects the logarithmic factor in the $O(p \lg p)$ work done by the kernel.

5.2.5 Comparing Performance of CUDA Random Splitter Implementation Kernels

The two graphs in Figure 5.5 show the performance of the individual kernels in the Random Splitter-based list-ranking implementation. These graphs are based on an experiment where we increase the list size n while always using the same number of thread-blocks $q = 32$. By looking at these curves next to each other we see an interesting difference in the performance of kernel RS2 between the two implementations. The better performance of the 48-bit implementation's RS2 kernel is the result of how much data is written by the kernel. In section 5.1.3 we pointed out that the 48-bit implementation uses two separate arrays for *owner*[] and *rank*[] whole elements are 16 bits and 32 bits wide respectively, while the 64-bit implementation uses a single *rankowner*[] array with 64-bit wide elements. Since the purpose of the RS2 kernel is to identify the splitters and update their owner values, one implementation only needs to update the 16-bit *owner*[] array while the other needs to write 64-bits packed values into the *rankowner*[] array.

As discussed earlier, kernel RS3 (see Listing 5.8) is the most labour-intensive and dominates the run-time of the algorithm. There are two differences between the 48-bit and 64-bit implementations of kernel RS3:

- In the 48-bit implementation, one read operation is needed from the 16-bit *owner*[], another read from the 32-bit *succ*[], and two write operations are needed to the 16-bit *owner*[] and 32-bit *rank*[]. Thus 96 bits are read and written using 4 operations for each of n elements in the list.

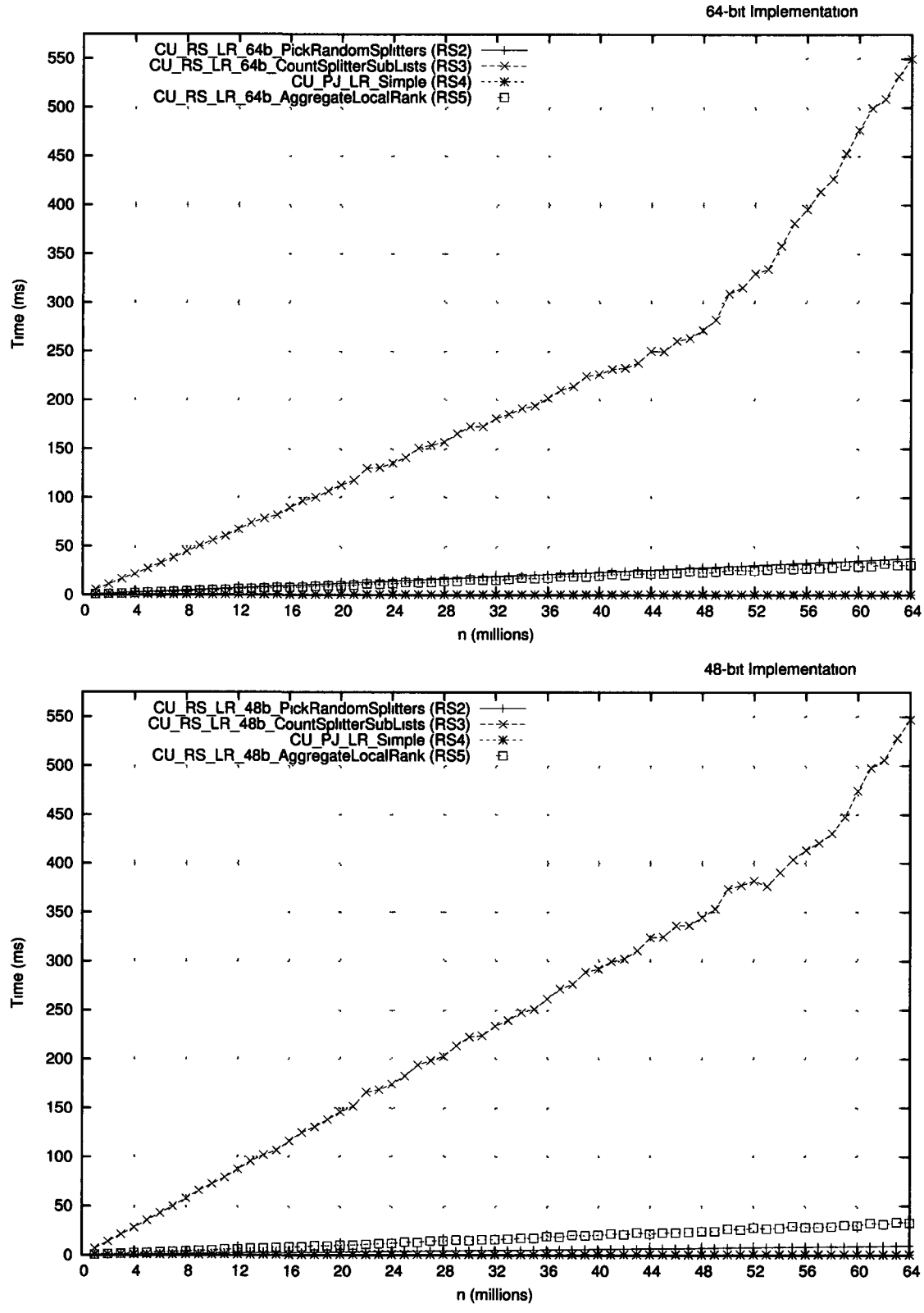


Figure 5.5: Performance of Random Splitter List Ranking Kernels

- In the 64-bit implementation, one 64-bit read operation is needed from *rankowner*[], another read from the 32-bit *succ*[], and one write operation is needed to *rankowner*[]. Thus 160 bits are read and written using 3 operations for each of n elements in the list.

Kernel RS4, the simplified pointer jumping-based list-ranking of the splitters, is identical in performance between the two implementation. This is as expected since there is no difference in the data size or the number of splitters between the two implementations.

While it is difficult to tell by looking at the curves, kernel RS5 is very slightly faster in the 64-bit implementation. This negligible difference is a result of a trade-off between different overheads in the two implementations. The 48-bit implementation needs to read both the *owner*[] and the *rank*[] arrays, thus accessing 48 bits using 2 transactions The 64-bit implementation needs to read the *rankowner*[] array once, thus accessing 64 bits but using 1 transaction. The extra 16 bits do not make enough of a difference compared to the impact of the reduced number of transactions.

5.2.6 The Inflection in the CUDA Random Splitter Implementation

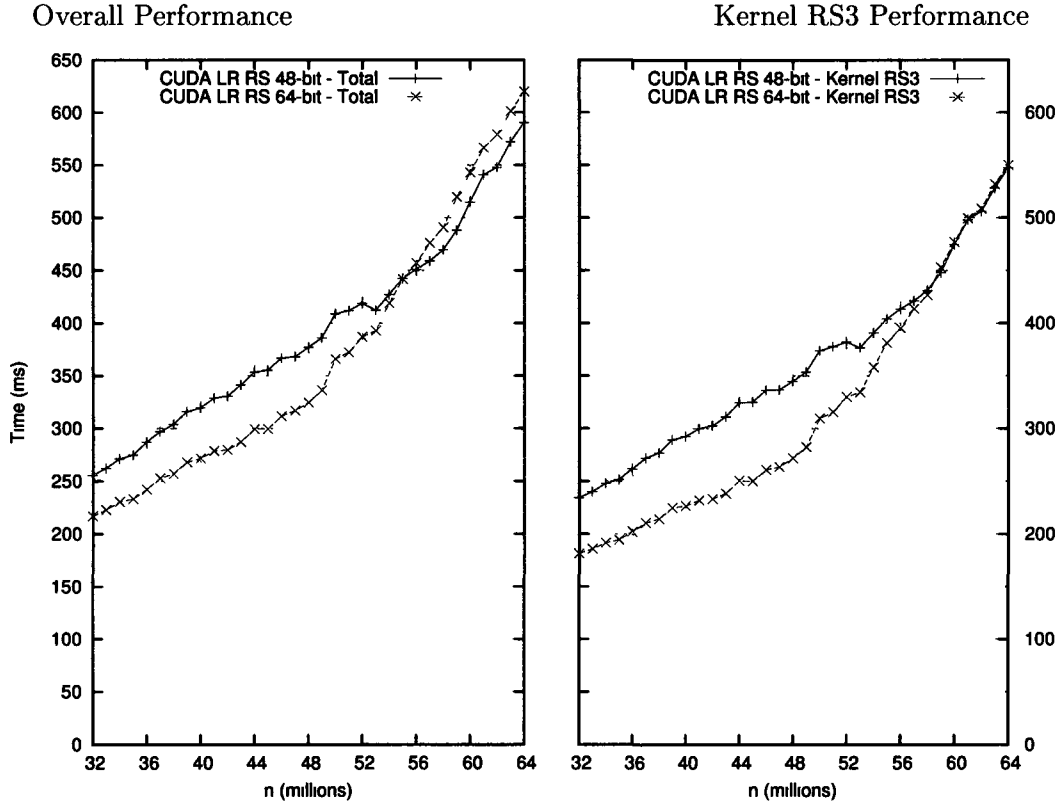


Figure 5.6: Inflection in Parallel Random Splitter List Ranking for CUDA

The graph in Figure 5.6 zooms in on the performance curves of the two random splitter-based implementations (i.e. 48-bit and 64-bit) for CUDA. Side by side we have the overall performance curves and the specific RS3 kernel performance curves.

From our study of the performance differences between all the kernels for the two implementation (see section 5.2.5) we noted that the only kernels RS2 and RS3 showed notable differences between the two implementations and that only the kernel RS3 showed an inflection in its performance. It can also be seen clearly when looking at Figure 5.5 that only kernel RS3 exhibits an inflection in both implementations.

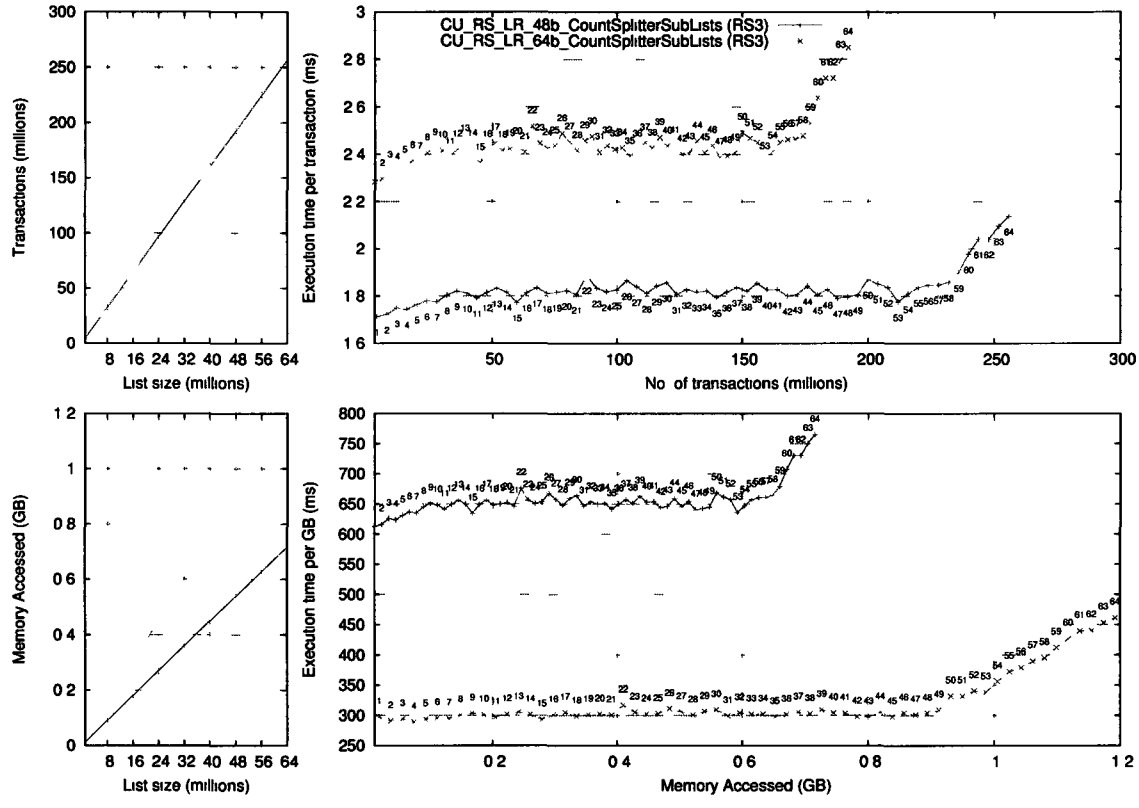


Figure 5.7: Impact of Amount of Memory Accessed vs Number of Access Transactions on Random Splitter List Ranking Kernel RS3

RS3 is the only kernel that examines every element in the list by walking along the successor links. Since these experiments were run using randomly distributed linked lists, we can assume that each element access will result, with very high probability, in a separate un-coalesced transaction. Thus for n elements in the list, every access to read and write from the owner, rank and successor arrays results in individual transactions for each array's index. Earlier we noted that the 48-bit implementation uses $4n$ operations to access $96n$ bits while the 64-bit implementation uses $3n$ operations to access $160n$ bits. With the lack

of coalescing this means that every operation is in fact a single transaction in the CUDA memory access network.

Figure 5.7 shows an analysis of the inflection by memory access and transaction count, separated out to better illustrate the different effects of both on the two implementations. The top row of graphs look at the number of transactions and the time per transaction as the number of transactions increases by list size. The bottom row looks at the the amount memory accessed (in GB) and the time per GB as the memory accesses increases by list size.

From a transactions point of view, the 64-bit implementation's run-time inflection occurs processing fewer transactions than the 48-bit implementation. However from a memory point of view, the 64-bit implementation's run-time inflection occurs accessing more memory. The graphs show that the difference between the two implementations' time-per-transaction prior to the inflection is quite small while the difference between the time-per-GB accessed is quite high, reinforcing the benefit of accessing more memory with fewer transactions.

Since the kernel RS3 implementations are identical otherwise, two conclusions can be made about the impact of memory access on performance.

- Accessing more bits has a penalty, which we can see in the kernel RS3 graph (right-side) where, after the 48-bit implementation encounters the inflection, the 64-bit implementation is only slightly slower.

- Using more access operations, and consequently more transactions, has a much bigger penalty than accessing more bits, which can be seen in the same graph where, before the 48-bit inflection, the 64-bit kernel implementation substantially outperforms the 48-bit implementation.

5.2.7 Memory Access Performance of CUDA Random Splitter Implementation

The graph in Figure 5.8 shows the memory access rate (in GB per second) for the two kernels that walk the entire list: kernels RS3 and RS5. We compare the two because RS3 results in a transaction per memory access but RS5 is designed to maximize coalescing (which is proven by its performing substantially faster than RS3). The y-axes represent memory access throughput in GB/s for increasing list sizes. We know that the 48-bit implementation's RS3 accesses $\frac{n\,96}{8\,1024^3}$ GB of memory and the 64-bit implementation accesses $\frac{n\,160}{8\,1024^3}$ GB. We also know that the 48-bit implementation's RS5 accesses $\frac{n\,112}{8\,1024^3}$ GB of memory and the 64-bit implementation accesses. The y-axis values are computed by dividing the total memory accesses in each kernel by time spent.

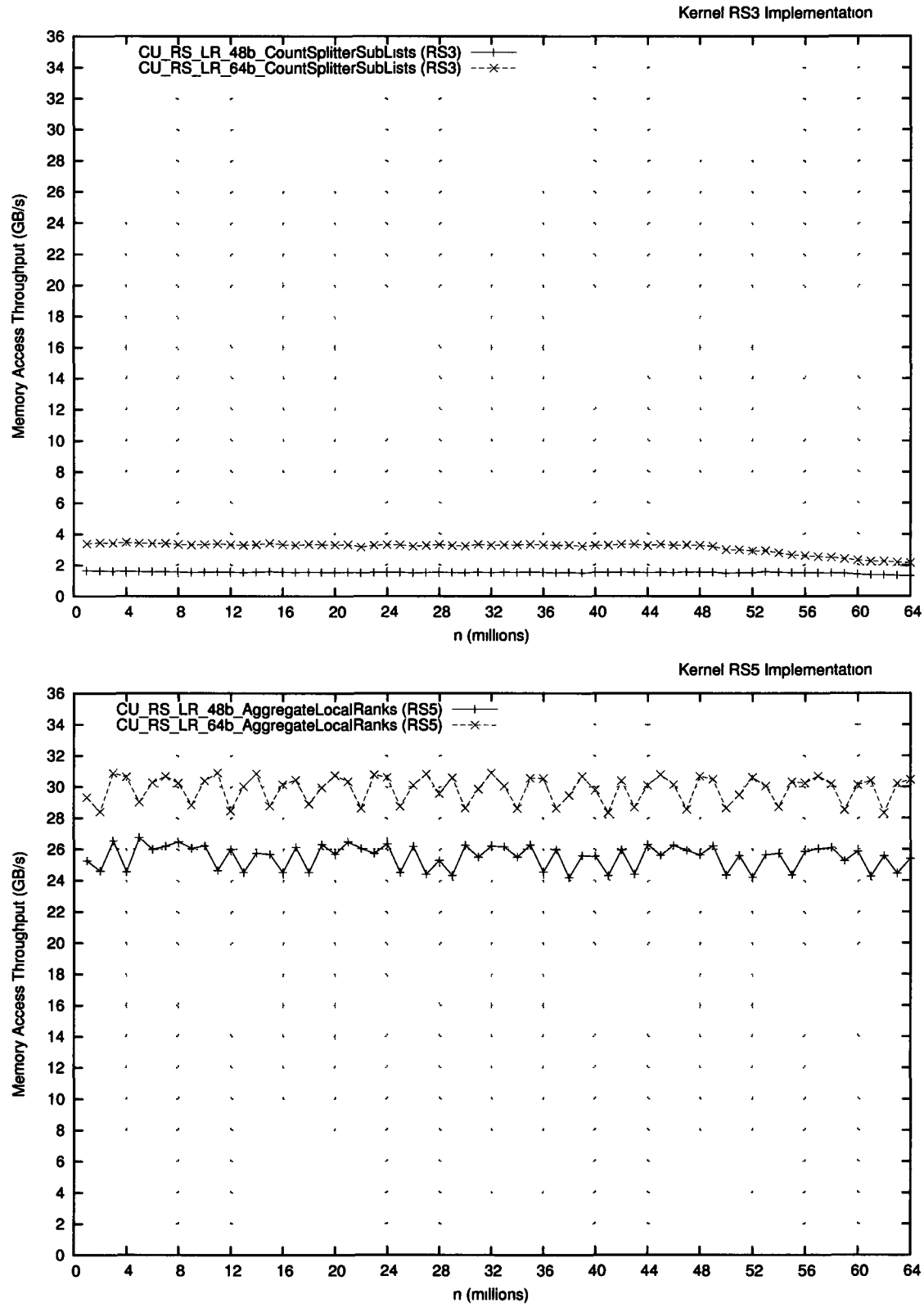


Figure 5.8: Comparing Memory Access Throughput of Random Splitter List Ranking Kernels RS3 and RS5

The 64-bit implementation's RS3 kernel is able to consistently process just under 4GB/s of memory accesses until the inflection point, while the 48-bit kernel processes just under 2GB/s. The RS5 kernels however show nearly an order of magnitude increase in their throughput relative to their RS3 counterparts, with the 64-bit kernels consistently processing $\sim$ 31GB/s and the 48-bit kernel at $\sim$ 27GBps. We know from the specification information for the NVIDIA GTX 260[1] that the GPU has a memory access bandwidth of 111GBps. Clearly, none of the kernels perform close to the expected bandwidth of the GPU.

According to Volkov and Demmel [21, sec. 3.5] the NVIDIA GPUs that they analyzed run at approximately ten times less the bandwidth capacity when copying memory with a stride ≥ 10 and approximately a hundred times less the bandwidth capacity with a stride in the order of 1000. Our experiments rank linked lists with millions of elements and a random link distribution; thus each element's neighbour is with high probability millions of slots away in the *succ*[] array. While Volkov and Demmel's measurement used even strides between elements, our linked list has the added complexity of unpredictable distance per element to its neighbour, providing even less opportunity for optimal access. This easily explains why kernel RS3 implementation's memory access performance is 100 times less than the expected 111GBps for the GTX260.

Explaining why the kernel RS5 implementations performs well below the expected bandwidth requires a deeper look at the algorithm. The clear advantage that RS5 has over RS3 is that it is able to maximize coalesced memory access both when reading each element's owner and local rank as well as writing each element's global rank. To compute the global rank, however, it needs to use each element's owner index to access the splitter's rank. While there are $r \ll n$ splitters, they are distributed randomly relative to each element. On one hand there is the benefit that more than one element can point to the same splitter; on the other hand when two elements refer to different splitters the probability is high that they are not located in proximity to take advantage of coalescing.

To confirm the hypothesis that RS5 is slower than expected because of the randomized splitter-rank access, we implemented a modified version of RS5 that read from the splitter-rank array in thread index order. This made the array access perfectly coalesced while ensuring that same amount of memory was read and written per element (see Listing 5.12). Figure 5.9 shows the memory through-put comparison between the proper RS5 and the coalesced-access RS5, where we can see that removing the randomized access of the splitter-rank array and replacing it with coalesced allows us to substantially increase the memory bandwidth, with the throughput at just under 95GBps.

[1] `http://www.nvidia.com/object/product_geforce_gtx_260_us.html`

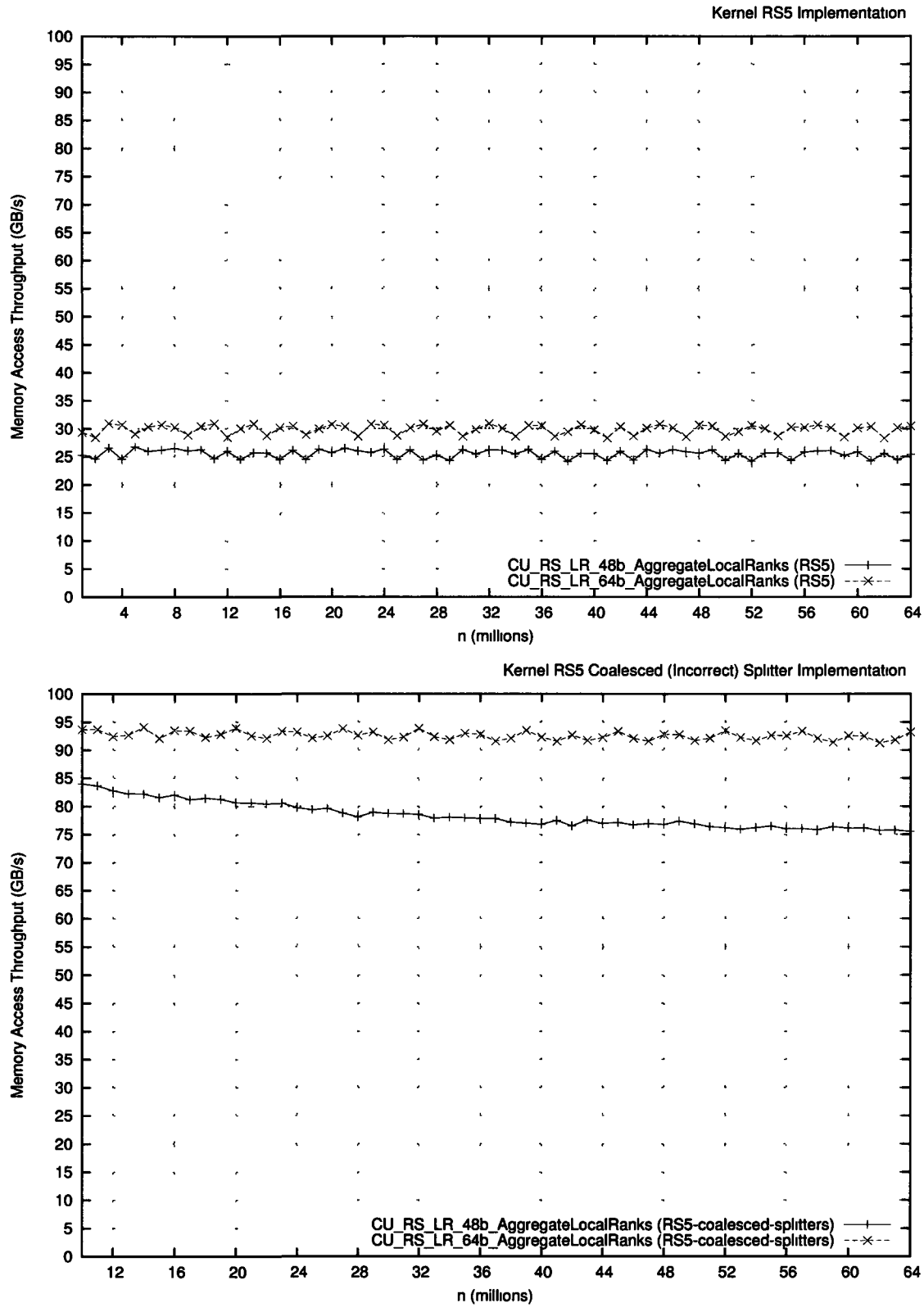


Figure 5.9: Comparing Memory Access Throughput of Random Splitter List Ranking Kernels RS5 with splitter-access coalescing modified RS5.

Listing 5.12 Parallel Rank Aggregation Kernel, Modified to use Coalesced Splitter Rank Access, in C/CUDA

```
1  __global__ void CU_RS_LR_64b_AggregateLocalRanks_CoalsdSplitRanks (
2              uint64_t *rankowner, uint32_t *rank, int32_t *splitter_rank_p,
3              int32_t n, int32_t p)
4  {
5      int32_t pIx, tIx;      // processor (thread) index.
6      uint64_t mk;
7
8      tIx = pIx = blockIdx.x * blockDim.x + threadIdx.x;
9      while (pIx < n) {
10         mk = rankowner [pIx];
11         // coalesced splitter access; INTENTIONALLY WRONG
12         rank [pIx] = splitter_rank_p [tIx] + xownr (mk) + xrank(mk);
13         pIx += p;
14     }
15 }
```

5.3 Conclusions

It is clearly possible to implement a parallel list ranking algorithm that does $O(n)$ work using the CUDA platform. The abilities of the CUDA platform to provide a multi-processing environment where millions of threads can be mapped to hundreds of physical cores, to coalesce parallel memory access at sequential addresses and to hide memory access latency make it an ideal platform to gain the benefits of highly parallel algorithms.

Parallel list ranking using random splitters implemented for CUDA definitely outperforms (see Figures 5.1 and 5.2) other algorithms implemented for CUDA (i.e. Wylie's in 5.1.2, Rehman et al's in [14]) as well as the sequential algorithm on a CPU (see 5.1.1) and the random splitter-based algorithm using multiple CPU threads (see 5.1.4).

The experiments run using randomly distributed linked lists, the worst case data for the algorithms, show that not being able to leverage the ability of the CUDA platform to coalesce memory access can have a substantial negative impact on performance compared to when coalescing can be used (see Figure 5.8). As a result we see that the GPU is definitely under-utilized when kernel RS3 spends most of its time accessing memory. Any increase in memory access performance will have a direct and meaningful impact on the performance of this kernel and thus the whole algorithm.

When accessing a lot of memory in such a way that coalescing cannot be leveraged, we also find that with enough data that we can cause the performance of the GPU to degrade with a clear inflection point in the performance curves (see 5.2.6). Reducing the number of transactions, even as we increase the amount of memory being accessed to do so, allows us

to defer the point at which performance starts to degrade (see Figure 5.7). Clearly there is a limit to the number of memory access transactions that can be pending in the system before performance is impacted.

List ranking is one of the simplest of problems that exhibit highly irregular data access patterns. Yet even with an optimal algorithm implemented on a parallel computing platform with many capabilities designed to maximize parallel memory access and computational performance, and on which the algorithm outperforms other implementations by a substantial margin (nearly 10x faster than the multi-threaded implementation on a CPU) we find that the platform's potential is under-utilized due to the memory access transaction overload.

Chapter 6

Connected Component Counting and CUDA

We now turn our attention to implementing and analyzing parallel graph connected component computation on a GPU. Consider a graph G with n vertices and m edges represented as a list of edges E such that each edge $E[i] = (a, b)$ is composed of two vertices a and b. Given such an edge representation we want to identify and count the number of *connected components* with the graph.

In this chapter we present the details of the implementation of the different connected components algorithms listed below, following which we examine the results comparing the performance differences between the algorithms. In addition to the sequential implementation, we study our GPU implementation (see section 4.2.3 for a review) of the CRCW-PRAM algorithm by Shiloach and Vishkin[17, 16] which computes connected components in $O(\log n)$ time with $O((n+m)\log n)$ work using $O(n+m)$ processors (see section 4.2.4). The algorithm assumes that PRAM concurrent writes are implemented using the "arbitrary" model. Algorithm 4.5 shows an outline of our GPU implementation of Shiloach and Vishkin's CRCW-PRAM algorithm. We implement our adaptation for the CPU using multi-threading as well so that we can compare how well it performs using all the cores of a CPU, unlike the sequential implementation which can only use one core.

6.1 Implementation Details

6.1.1 Sequential CPU Connected Component Counting

The sequential implementation of counting connected components is simply implemented recursively. For each vertex we recursively traverse the links, marking each visited node and labeling each time we start the traversal with an unvisited vertex. This implementa-

tion however relies on a vertex-list-based representation of the graph, while all the parallel algorithms discussed here process edge-list-based graphs. As a result we need an additional routine that can convert a graph from an edge-list into a vertex-list for use with the sequential implementation. Listing 6.1 shows the host function that first invokes the conversion followed by which it starts the sequential counting using a recursive labeling function.

Listing 6.1 Sequential List Ranking Implementation - Main

```
1  uint32_t SQ_CCC_countComponents (edge_graph_t *eG) {
2      vertex_graph_t vG;
3      uint32_t label, i;
4
5      convertEdgeG (eg, &vG);
6      // for each vertex, depth first scan and label
7      for (label = i = 0; i < vG->numV; i++)
8          if (vG->V [i].cclabel == 0) // unlabelled; so new component.
9              recurseLabel (&vG->V [i], ++label);
10     return label;
11 }
12
13 static void recurseLabel (vnode_t *vnode, const uint32_t label) {
14     if (vnode->cclabel != label) {          // if not visited
15         vnode->cclabel = label;             // label self
16         for (int i=0; i<vnode->degree; i++) // visit neighbours
17             recurseLabel (vnode->edges [i], label);
18     }
19 }
```

We note here that when comparing the runtime performance we include both the conversion time as well as the recursive labeling. This is done because the parallel algorithms all operate using an edge list representation for the graph, and we wanted to make a fair comparison such that all implementations take an edge-list as input and thus any processing of that edge-list is included. This has the net effect of changing the $O(n)$ vertex-representation-based sequential labeling algorithm into at best an $O(n+m)$ algorithm due to the overhead of processing the m edges to generate the vertex-representation.

6.1.2 Parallel CUDA Shiloach-Vishkin's Connected Component Counting

Algorithm 4.5 shows an outline of our GPU implementation of Shiloach and Vishkin's CRCW-PRAM algorithm. We start by outlining some of the important aspects of our GPU implementation of Shiloach and Vishkin's CRCW-PRAM algorithm, following which the CUDA code segments for the kernels listed in Algorithm 4.5 are then presented and

analyzed, including an additional kernel for counting the reported components through a parallel sum operation.

Use Of Striding *And* Concurrent Writes

The PRAM algorithm requires $m+n$ processors, one for each edge and vertex. As discussed in Section 3.8, a GPU can execute a large number of concurrent threads but not as many as one thread per data item. In order to improve performance, our CUDA code implements the striding access pattern outlined in Section 3 8 using $p \leq \min(m, n)$ threads. In addition, Shiloach and Vishkin's CRCW-PRAM algorithm requires concurrent write operations which is in principle no problem for a GPU as discussed in Section 3.7. However, when both striding *and* concurrent writes are implemented together, special care must be taken to ensure that the correct semantics are maintained and race conditions avoided.

Step 5 of Shiloach and Vishkin's algorithm ([17, p. 60]) checks whether the previous execution of Steps 1 through 4 resulted in any changes. In our GPU implementation, this is also implemented through a combination of both striding and concurrent writes Each thread first checks whether any of its items have changed and then all those threads that detected a change attempt to update a global variable w, thus implementing a parallel "OR" through a concurrent write.

Memory Access Optimization

As outlined in Section 3.6.1, the GPU's hardware optimizes the use of bandwidth to global memory through coalescing of memory accesses. Our GPU implementation of Shiloach and Vishkin's CRCW-PRAM algorithm makes efforts to save memory bandwidth through pre-fetching to local registers. This ensures that within each kernel the same global memory address is accessed only once, especially since all the working data structures are by necessity maintained in device (global) memory (which is the slowest to access), and we want to avoid accessing them any more than absolutely necessary. The implementation of kernels SV1, SV2 and SV3 apply this optimization.

Implementation Details of the Host Function

The CUDA-based implementation of the Shiloach-Vishkin parallel algorithm is made up of multiple kernels which must be invoked by a single host function running on the CPU. Listing 6.2 illustrates the implementation of this host function.

Listing 6.2 Parallel Shiloach-Vishkin Connected Components Multi-Kernel Outline, in C/CUDA

```
1  uint32_t CU_SV_CCC_host (edge_graph_t *eG) {
2      // allocate device memory
3      // copy edge-list to device memory
4      cudaMemcpy (..., cudaMemcpyHostToDevice);
5      // initialize the D and Q arrays in device memory
6      s = 0;
7      CU_SV_CCC_InitOnce (...);                                    // SV0
8      // multiple <= lg(n) iterations until work is done
9      do {
10         s = s + 1;       // level up
11         CU_SV_CCC_ShortCutAndMark <<<q,256>>> (...);             // SV1
12         CU_SV_CCC_HookEdges <<<q,256>>> (...);                   // SV2
13         CU_SV_CCC_HookStagnantRoots <<<q,256>>> (...);           // SV3
14         work_done = 0;
15         CU_SV_CCC_ShortCutAndCheckWork <<<q,256>>> (...);        // SV4,5
16         // extract work done value from device memory
17         cudaMemcpy (&work_h, work_d, sizeof (uint32_t),
18                                      cudaMemcpyDeviceToHost);
19         // swap Ds_1 and Ds for next iteration
20         vertex_t *tmpD = Ds_1; Ds_1 = Ds; Ds = tmpD;
21     } while (work_done > 0);
22     // count the roots identified, in parallel
23     CU_SV_CCC_CountRoots <<< 1, 512, 512*sizeof (uint32_t) >> (...)
24     // extract sum value from device memory
25     cudaMemcpy (&count, ..., cudaMemcpyDeviceToHost);
26     // free device memory
27     return count;
28 }
```

Unlike the parallel list ranking implementation, the Shiloach-Vishkin algorithm performs $O(\log n)$ iterations, in each of which it repeatedly performs steps 1 through 5. The variable s represents the current step (iteration), and the following sub-steps implemented as separate CUDA kernels are repeatedly executed until we find an iteration at the end of which no change has occurred and thus no further work is needed.

The algorithm (see section 4.2.4) relies on several temporary data structures: two arrays D_s and D_{s-1} that represent the state of the pointers that are used to construct the star graphs rooted at each component, as well as the third array Q used to track when changes have been made in D_s for a particular vertex in the graph. While in theory every iteration gets its own D_s array, the algorithm at any given time relies only on D_s and D_{s-1}. As a result the implementation allocates memory only for two arrays and at the end of each iteration swaps their assignment between D_s and D_{s-1}.

An important difference between the proposed execution of the original PRAM algorithm

and our GPU implementation has to do with how the processors (i.e. CUDA threads) are allocated to the work. Shiloach and Vishkin require that n processors are needed for the steps that process all the vertices in parallel and that m processors are needed for the steps that process all the edges. Since each step must occur in order, theoretically, their algorithm requires at most m processors. Looking at each of the kernels, however, we can see that each processor, whether processing a vertex or edge, relies on the state from the previous steps. This allows us to used a fixed number $p < \max(n, m)$ of processors (i.e. CUDA threads) and modify the kernels to use striding to distribute the work. Thus each kernel, regardless of whether it is processing vertices or kernels, always uses p threads; and when $p < n$ and/or $p < m$ each of the p threads repeatedly process p segments (i.e. $0 \ldots p-1, p \ldots 2p-1, \ldots$). This makes the implementation practical for use with any multi-threaded system, in turn allowing us to implement a non-GPU version with CPU-based multi-threading (as described later in section 6.1.3).

Implementation Details of the Initialization Kernel

Listing 6.3 Parallel Shiloach-Vishkin Initialization Kernel, in C/CUDA

```
1  __global__ void CU_SV_CCC_InitOnce (
2      vertex_t *Ds, step_t *Q, vertex_t nV, uint32_t p)
3  {
4      vertex_t vi = blockIdx.x * blockDim.x + threadIdx.x;
5      while (vi < nV) {
6          Ds [vi] = vi;
7          Q [vi] = 0;
8          vi += p;     // stride
9      }
10 }
```

The initialization kernel (see Listing 6.3) executes once before the iterations begin, initializing D_0 such that each vertex points to itself and initializing Q such that each vertex indicates no work done since level $s = 0$.

Implementation Details of the First Short-Cutting Kernel

The first short-cutting kernel (see Listing 6.4) looks at every vertex and tries to move the pointers one jump closer to the root of the star-graph the algorithm is trying to build. Of course when $s = 1$ all the vertices point to themselves and the first attempt achieves little; however as we progress through the remaining steps the pointers change as edge-hooking occurs (see section 4.2.4 for an example), allowing subsequent short-cutting attempts to make significant progress.

Listing 6.4 Parallel Shiloach-Vishkin Short-Cut And Mark Kernel, in C/CUDA

```
1  __global__ void CU_SV_CCC_ShortCutAndMark (
2      vertex_t *Ds, vertex_t *Ds_1, step_t *Q, step_t s,
3      vertex_t nV, uint32_t p)
4  {
5      vertex_t vi, Ds_vi, Ds_1_vi;
6      vi = blockIdx.x * blockDim.x + threadIdx.x;
7      while (vi < nV) {
8          Ds [vi] = Ds_vi = Ds_1 [Ds_1_vi = Ds_1 [vi]];    // short-cut
9          if (Ds_vi != Ds_1_vi)                 // check ...
10             Q [Ds_vi] = s;                    // and mark if changed
11         vi += p;     // stride
12     }
13 }
```

The kernel also checks whether each short-cutting attempt has some effect or not and updates Q for the vertex to the current value of s if there has been a change. This information is used at the end of the iteration by checking each vertex to determine if another iteration is needed.

Here we see the first explicit use of local variables to optimize memory access so that values from D_s and D_{s-1} are never accessed more than once from global memory. The CUDA compiler does not perform any memory access optimization for repeated accesses via the same pointer, leaving it to the implementor instead to cache an access when the same value from global memory is used multiple times Since local variables in CUDA are actually allocated from register memory, pre-loading $D_s[i]$ and $D_{s-1}[i]$ into local variables `Ds_vi` and `Ds_1_vi` substantially improves runtime performance. This pattern can be seen in the other kernels as well.

Implementation Details of the Edge-Hooking Kernel

Since this algorithm processes an edge-list representation of the graph we represent the entire graph as an array of edges, where each edge is a 64-bit value composed of two 32-bit values that each represent a vertex, as shown in the code fragment below.

```
typedef uint32_t vertex_t;
typedef uint64_t edge_t;
#define MKEDGE(v1,v2) ((((edge_t) v1) << 32) | \
                       (((edge_t) v2) & 0xffffffff))
#define EDGEV1(e) ((vertex_t) (((edge_t) e) >> 32))
#define EDGEV2(e) ((vertex_t) (((edge_t) e) & 0xffffffff))
```

The macro `MKEDGE(v1,v2)` is used to pack the two vertices into a single 64-bit edge, and the macros `EDGEV1(e)` and `EDGEV2(e)` are used to extract the vertices from the packed edge. This packing strategy is motivated by the same performance goals that led us to consider and compare using packed (64-bit) and unpacked (48-bit) implementations of the parallel list ranking algorithms.

Listing 6.5 Parallel Shiloach-Vishkin Edge Hooking Kernel, in C/CUDA

```
1  __global__ void CU_SV_CCC_HookEdges (
2      edge_t *E, vertex_t *Ds, vertex_t *Ds_1, step_t *Q, step_t s,
3      edge_t nE, uint32_t p)
4  {
5      edge_t e;
6      vertex_t v1, v2, Ds_v1, Ds_v2;
7      index_t ei = blockIdx.x * blockDim.x + threadIdx.x;
8      while (ei < nE) {
9          e = E [ei];
10         v1 = EDGEV1 (e); v2 = EDGEV2 (e);
11         Ds_v1 = Ds [v1]; Ds_v2 = Ds [v2];
12         if ((Ds_v2 < Ds_v1) && (Ds_v1 == Ds_1 [v1])) {
13             Ds [Ds_v1] = Ds_v2;
14             Q [Ds_v2] = s;       // mark change
15         }
16         ei += p;     // stride
17     }
18 }
```

In the Edge Hooking kernel (see Listing 6.5) we are thus able to use a single 64-bit access operation to load each edge, reducing the total number of transactions. Note that in this kernel, if the two conditional checks succeed, we update both D_s and Q. This reflects the actual hooking operation as well as noting that a change as occurred in this iteration.

Implementation Details of the Stagnant Root Hooking Kernel

Both of the steps of hooking edges as well as hooking stagnant roots (see Listing 6.6) rely on the CRCW property of a PRAM. Shiloach and Vishkin require that all the processors considering edge (j, k) may (under the right conditions) try to update $D_s(D_s(j))$ simultaneously; however only one of them should succeed and it does not matter which [17, p.61].

Listing 6.6 Shiloach-Vishkin Stagnant Root Hooking Kernel, in C/CUDA

```
1  __global__ void CU_SV_CCC_HookStagnantRoots (
2      edge_t *E, vertex_t *Ds, step_t *Q, step_t s,
3      edge_t nE, uint32_t p)
4  {
5      edge_t e;
6      vertex_t v1, v2, Ds_v1, Ds_v2;
7      index_t ei = blockIdx.x * blockDim.x + threadIdx.x;
8      while (ei < nE) {
9          e = E [ei];
10         v1 = EDGEV1 (e); v2 = EDGEV2 (e);
11         Ds_v1 = Ds [v1]; Ds_v2 = Ds [v2];
12         if ((Ds_v1 != Ds_v2) && (Ds_v1 == Ds [Ds_v1])
13                              && (Q [Ds_v1] < s))
14             Ds [Ds_v1] = Ds_v2;
15         ei += p;     // stride
16     }
17 }
```

The CUDA GPU platform provides exactly this guarantee (see section 3 7), and when multiple concurrent writes occur to the same address exactly one of them will succeed. This allows us to make two implementation decisions: First, when p threads execute in parallel, if more than one of them writes to the same address, the GPU will ensure only one write will succeed and no special implementation tricks are needed. Secondly, the use of striding to use $p < m+n$ threads preserves the requirements that one of the concurrent writes must survive. While one write succeeds for each concurrent write per stride, when the kernel completes after $\left\lceil \frac{m}{p} \right\rceil$ strides, the one write (if any) performed by the last stride to the same address as another successful write in an earlier stride will obviously succeed. So while it is possible that the same address is modified in each stride, this merely results in $\left\lceil \frac{m}{p} \right\rceil$ sequential writes to that address of which the last one will persist. This is correct since the algorithm doesn't care which of the concurrent writes succeeds, just that one does.

Implementation Details of the Short-Cutting and Work Counting Kernel

The second short-cut operation in step 4 of the algorithm (see Listing 6.7) is not required for the correct operation of the algorithm. However omitting this step may double the number of iterations before the counting completes [17, p.61]. Since the GPU is very sensitive to any work, especially when that work involves accessing slow global memory, every extra iteration can have a significant impact on execution time. Thus implementing this step is necessary to optimize the run-time of the implementation.

In the PRAM algorithm steps 4 and 5 are executed separately. When examining the

operations, however, it is clear that step 5 (which examines Q for each element to determine if work was done during that iteration) does not depend on the modifications made to D_s by step 4; and Q is updated (if at all) only in steps 1 and 2 of each iteration. Thus by the time we are ready to execute step 4, we can do the work from both of the steps within the same stride of one combined kernel shown in Listing 6.7. This not only simplifies the implementation, it also helps with the run-time of the implementation since reducing kernel calls is always a positive optimization.

Listing 6.7 Shiloach-Vishkin Short-cut And Check Work Kernel, in C/CUDA

```
1  __global__ void CU_SV_CCC_ShortCutAndCheckWork (
2      vertex_t *Ds, step_t *Q, step_t s, uint32_t *work,
3      vertex_t nV, uint32_t p)
4  {
5      uint32_t w = 0;
6      vertex_t vi = blockIdx.x * blockDim.x + threadIdx.x;
7      while (vi < nV) {
8          Ds [vi] = Ds [Ds [vi]]; // short-cut
9          w += (Q [vi] == s);      // count up work
10         vi += p;     // stride
11     }
12     if (w > 0) *work = 1;    // parallel OR.
13 }
```

At the end of each iteration, the loop's condition checks if any work was done. The algorithm states this in a very simple way, that two variables s and s' are maintained and that while s is always incremented, s' is only incremented if work was done during the iteration. The CUDA implementation of step 5 works differently but achieves the same objective. Every vertex is checked for work and a counter is updated by each thread. When each thread has finished checking its vertices, it checks if its sum of work is non-zero, and only if non-zero writes a 1 to a single address, effectively performing a parallel OR. Here again we rely on the CRCW guarantee that one write will succeed. Since we initialize the address to zero before invoking the kernel, and we only write a value of 1 if Q indicates that D_s has changed for one or more of the slots considered by each thread, we know that even if only one thread detects a change and writes the value that the address will definitely be updated, and if no work was done then no write will be attempted at all.

Implementation Details of the Root Counting Kernel

When we exit the loop after the last iteration where no work is done, we have completed constructing a star-graph for each component in the graph. In the pointer array D_s every vertex either points to the root of its star-tree or points to itself if it is the root of the star.

Since we are also interested in counting the number of components, one more step is needed in the implementation.

Listing 6.8 Shiloach-Vishkin Root Counting Kernel, in C/CUDA

```
1  __global__ void CU_SV_CCC_CountRoots (
2      vertex_t *Ds,  vertex_t nV, uint32_t p, uint32_t *sum)
3  {
4      extern __shared__ uint32_t shsums [];    // shared memory
5      uint32_t locsum, work;
6      vertex_t ix = threadIdx.x;
7      if (blockIdx.x == 0) {   // only one block allowed to run kernel
8          for (locsum = 0; ix < nV; ix += p)  // stride
9              locsum += (Ds [ix] == ix);      // add 1 if root
10         shsums [ix = threadIdx.x] = locsum; // store local sums
11         for (work = p/2; work > 1; work /= 2) { // sum by reduction
12             if (ix < work) {
13                 locsum = shsums [ix] + shsums [ix + work];
14                 __syncthreads ();   // sync to finish reads
15                 shsums [ix] = locsum;
16                 __syncthreads ();   // sync to finish writes
17             }
18         }
19         if (ix == 0)     // final sum, by thread 0 only
20             *sum = locsum + shsums [1];
21     }
22 }
```

To implement the root counting kernel (see Listing 6.8) we customize a parallel reduction-based summation algorithm that executes in a single thread-block and computes the sum of an array of numbers by first computing the sum of $\frac{n}{p}$ elements in parallel using p threads. Then, over $\lg(p)$ iterations, we use $\frac{p}{2^{j+1}}$ threads for iteration j to compute $\frac{p}{2^{j+1}}$ sums. Eventually, when $2^{j+1} = p$, one thread will compute the last pair-wise sum and yield the total root count. We chose to implement the simpler single-thread-block summation kernel[1] given the very low run-time cost of he root-counting step when compared to the overall execution of the implementation.

We know from the algorithm that, when it completes, the pointer overlay D_s will contain a link for every vertex such that if a vertex is the root of a component it will point to itself; i.e. $D_s[i] = i$ for a vertex i that is the root of a star-graph. When programming using the C language, a comparison always produces a value of zero (failure) or one (success). We take advantage of this to compute the local sum for each thread without any if-then branching.

[1] A detailed examination of multi-thread-block summation by reduction is available at the nVIDIA developer zone for CUDA `http://developer.download.nvidia.com/compute/cuda/1_1/Website/Data-Parallel_Algorithms.html#reduction`

6.1.3 Parallel CPU Shiloach-Vishkin's Connected Component Counting

The multi-threaded implementation of the Shiloach-Vishkin algorithm for execution on a CPU is shown in Listing 6.9. Like the multi-threaded parallel list ranking implementation (see section 5.1.4) this implementation uses the generic concurrent API that was developed for this project. The following points describe the key characteristics of the implementation as well as some of the differences relative to the algorithm.

Using the concurrency API's barrier synchronization from within the threads, the implementation is written as a single parallel function, with each of the kernels embedded within and separated by a call to `par_for_p_sync()` to ensure synchronization (see lines 15, 22 and 31 for examples).

Unlike the CUDA implementation, loops within each "kernel" in the CPU implementation use partitioning instead of striding (see section 3.8). This takes advantage of the way memory access caching works in CPUs, where the best performance is obtained when each core accesses sequential memory over its individual loop.

There is no notion of thread-blocks when multi-threading with a CPU, so when we need to perform sequential operations we do the work by restricting the work to the thread with id 0. All the other threads skip the work and wait at the next barrier. As a result we are able to perform the iteration management within the threads (unlike CUDA where the iterations are maintained in the host CPU function) and always rely on thread 0 to set up the decision variable which then every thread uses.

The C compiler for the CPU performs memory access optimizations during compilation, and as a result we do not need to explicitly use local variables to cache memory access to the working data to optimize memory access within loops.

Root counting is implemented slightly differently compared to the CUDA implementation. In parallel we still compute the partial sums such that when the implementation ends we have an array of sums with a value from each thread. Then the host function sequentially adds up the values to determine the final sum. Since a CPU does not offer more than a handful of cores (compared to the millions of threads possible with a GPU) we only have a handful of local sums that combine, and the effort of performing this part of the root counting sequentially has no impact on the run-time of the overall CPU implementation.

Listing 6.9 Parallel Random Splitter List Ranking, Multi-threaded C Implementation.

```
1  void MTH_SV_CCC_par_fun (int pIx, int p, CALL_t *call) {
2      graph_t *G = call->G;
3      index_t vsz, esz, vi0, ei0, v1, ei, work;
4      vertex_t *tmpD, *Ds = G->Ds, *Ds_1 = G->Ds_1;
5      //---- computer partition index and size...
6      vsz = G->eG.numV / p; vi0 = i * vsz;
7      if (i == p-1) vsz = G->eG.numV - ((p-1) * vsz);
8      esz = G->eG.numE / p; ei0 = i * esz;
9      if (i == p-1) esz = G->eG.numE - ((p-1) * esz);
10     for (v1=vi0; v1 < vi0 + vsz; vi++) {        // SV0: init Q and Ds_1
11         G->Q[v1] = 0; Ds_1[v1] = v1;
12     }
13     par_for_p_sync (call->hpar);                // barrier sync
14     while (1) {
15         for (v1=vi0; v1 < vi0 + vsz; v1++) {    // SV1: short-cut, check, mark
16             Ds[v1] = Ds_1[Ds_1[v1]];
17             if (Ds[vi] != Ds_1[v1]) G->Q[Ds[v1]] = call->s;
18         }
19         par_for_p_sync (call->hpar);            // barrier sync
20         for (e1=ei0; e1 < ei0 + esz; e1++) {    // SV2: hook edges
21             edge_t e = G->eG.E[ei];
22             vertex_t v1 = EDGEV1 (e), v2 = EDGEV2 (e);
23             if ((Ds[v2] < Ds[v1]) && (Ds[v1] == Ds_1[v1])) {
24                 Ds[Ds[v1]] = Ds[v2];
25                 G->Q[Ds[v2]] = call->s;
26             }
27         }
28         par_for_p_sync (call->hpar);            // barrier sync
29         for (e1=ei0; e1 < ei0 + esz; e1++) {    // SV3: hook stagnant roots
30             edge_t e = G->eG.E[e1];
31             vertex_t v1 = EDGEV1 (e), v2 = EDGEV2 (e);
32             if ((Ds[v1] != Ds[v2]) && (Ds[v1] == Ds[Ds[v1]])
33                                    && (G->Q[Ds[v1]] < call->s))
34                 Ds[Ds[v1]] = Ds[v2];
35         }
36         par_for_p_sync (call->hpar);            // barrier sync
37         for (work=0, v1=vi0; v1 < vi0 + vsz; v1++) {
38             Ds[v1] = Ds[Ds[v1]];                // SV4: short-cut, again
39             work += (G->Q[v1] == call->s);      // SV5: count work done
40         }
41         call->work[i] = work;
42         par_for_p_sync (call->hpar);            // barrier sync
43         /* ---- decide to iterate or not. ---- */
44         tmpD = Ds; Ds = Ds_1; Ds_1 = tmpD;      // swap Ds and Ds_1
45         if (i == 0) {   // one thread sums the work of all threads
46             for (work=0, vi=0; v1<p; v1++) work += call->work[v1];
47             call->work[0] = work; call->s ++;
48         }
49         par_for_p_sync (call->hpar);            // barrier sync
50         if (call->work[0] == 0) break;          // done if no more work.
51     }
52     uint32_t locsum = 0;                        // count the roots
53     for (v1=vi0; v1 < vi0 + vsz; v1++) { locsum += (Ds_1[vi] == vi); }
54     call->work[i] = locsum;
55 }
```

6.2 Run Time Comparison For Different Types Of Graphs: Lists, Trees, and Random Graphs

The CPU implementations were executed on a system with a quad-core CPU while the CUDA implementations were executed using a nVIDIA GTX 260 GPU installed on a quad-core CPU host system (see Appendix A on page 97 for details). The run-time performance of the connected component counting implementations was analyzed by running experiments using four different types of graphs.

1. Graph made up of one or more linked lists.
2. Graph made up of one or more fixed-degree trees.
3. Graph made up of one or more randomly generated sub-graphs, with 0.001% edge density.
4 Graph made up of one or more randomly generated sub-graphs, with 0.01% edge density.

With respect to the random graphs, edge density refers to the number of edges m generated relative to the number of vertices n. For lists and trees, since $n \cong \frac{m}{2}$ we simply specify the number of desired vertices to generate the graphs. However for random graphs we compute $n = \sqrt{\frac{m}{2\delta}}$ where $\delta \in [0 \ldots 1]$ specifies the edge density, such that $\delta = 1$ generates a complete graph.

6.2.1 Comparing All Connected Component Counting Implementations

The graphs in Figure 6.1 on the following page and Figure 6.2 on page 88 show a comparison of the run-time performance of between the sequential CPU algorithm, the multi-threaded CPU version of Shiloach-Vishkin, and the CUDA version of Shiloach-Vishkin. In the case of the CUDA version, the implementation was run using $q = 16$ and $q = 32$ thread blocks.

While we can clearly see that the CUDA implementations outperform the CPU implementations in all cases, we can also see that the difference is much more pronounced for list/tree graphs than random graphs. One reason for the difference between the list/tree graphs and the random graphs is a result of the edge distribution. As mentioned earlier, with list and tree graphs $n \simeq \frac{m}{2}$ while for random graphs $n = \sqrt{\frac{m}{2\delta}}$ where δ is the edge density. Right away we can see that for $m = 64M$ edges the list and tree graphs have $n \simeq 32M$ vertices while the random graphs have $n \simeq 178K$ and $n \simeq 56K$ edges for $\delta = 0.001$ and $\delta = 0.01$ respectively.

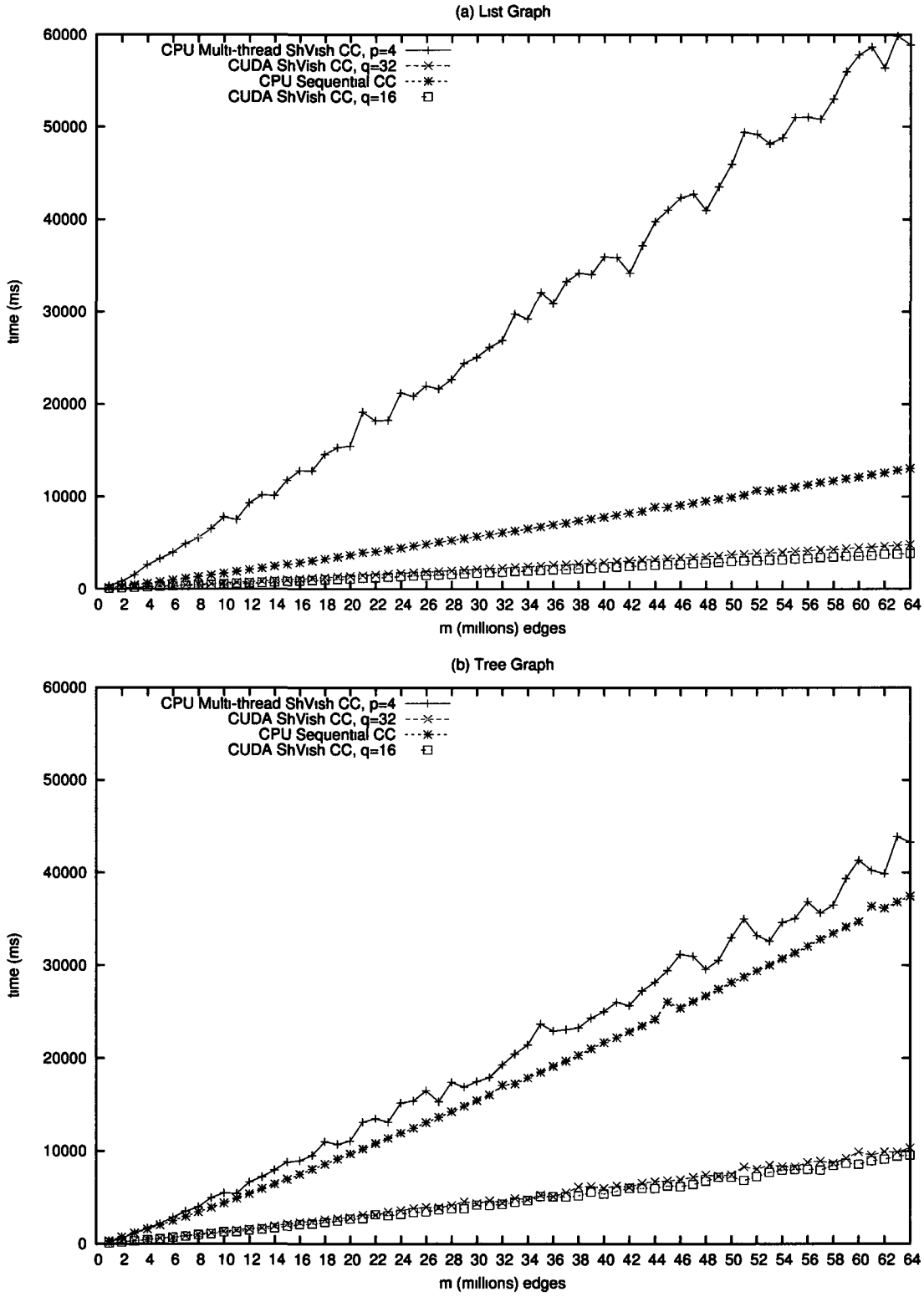


Figure 6.1: Run Time Comparison of the Connected Component Counting Implementations for List and Tree Graphs

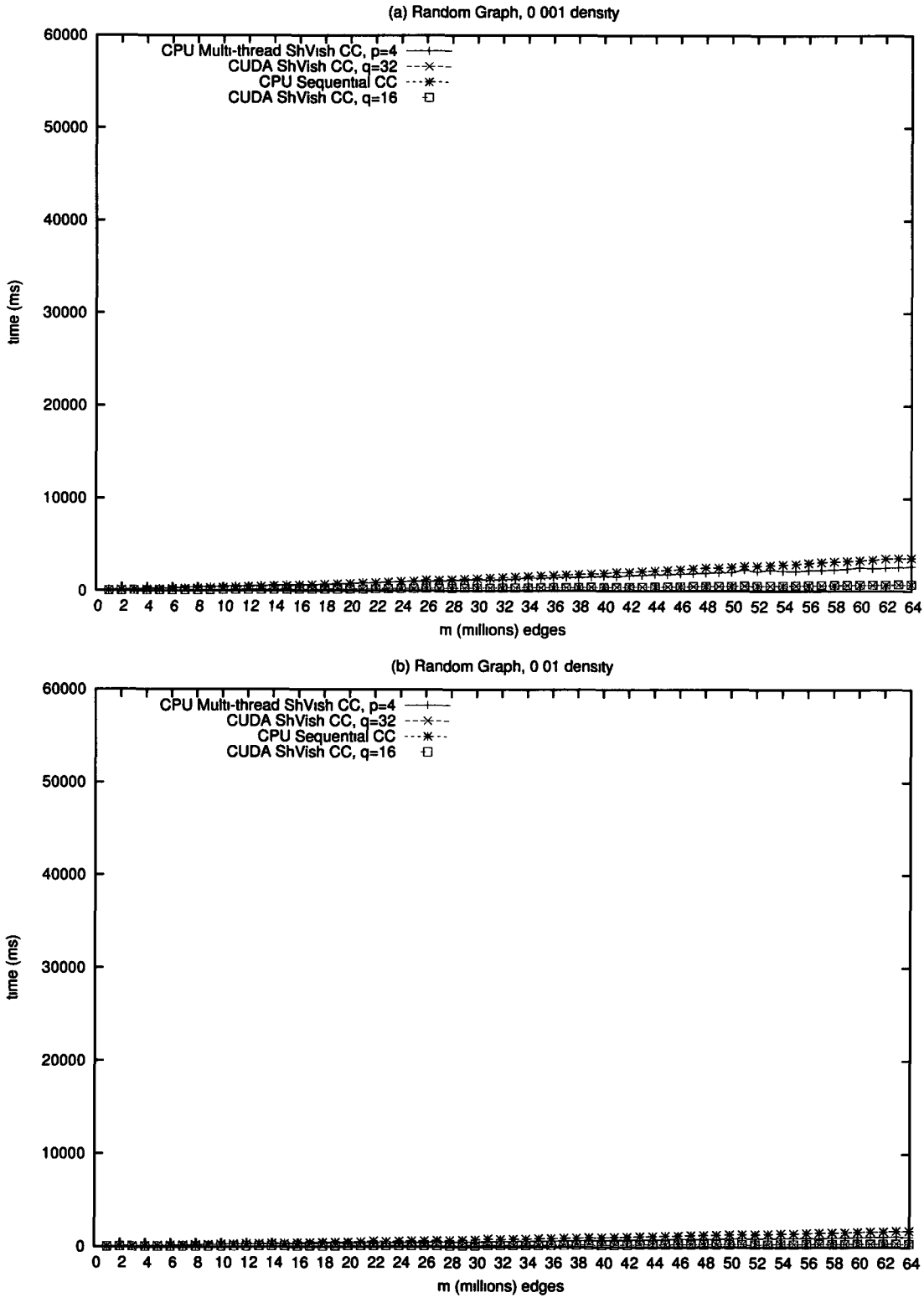


Figure 6.2: Run Time Comparison of the Connected Component Counting Implementations for Random Graphs

Thus random graphs exhibit a much higher edge density, and in general a much smaller diameter, relative to the list/tree graphs. Given the same number of total edges, the much smaller number of vertices in the random graphs significantly reduce the work done by the kernels that process every vertex.

Both the sequential and parallel algorithms benefit from the extra edges in the random graphs, where each node has a significantly higher number of direct links to neighbours in the graph. In the sequential algorithm this allows the algorithm to label all the vertices without necessarily traversing every edge in the graph, and in fact regardless of density does $O(n)$ work. In the parallel algorithms, this allows the pointer jumping to produce the star graphs with fewer iterations (shown by Table 6.1) for random graphs even as the algorithm, by design, continues to do $O(n+m)$ work in each iteration.

	CPU SV ($p = 4$)	**CUDA SV** ($q = 16$)	**CUDA SV** ($q = 32$)
List Graph	6	19	18
Tree Graph	19	25	21
Random Graph $\delta = 0.001$	5	6	6
Random Graph $\delta = 0\ 01$	5	5	5

Table 6.1: Comparing the No. of Iterations in the Parallel Connected Components Implementations for Different Graph Types.

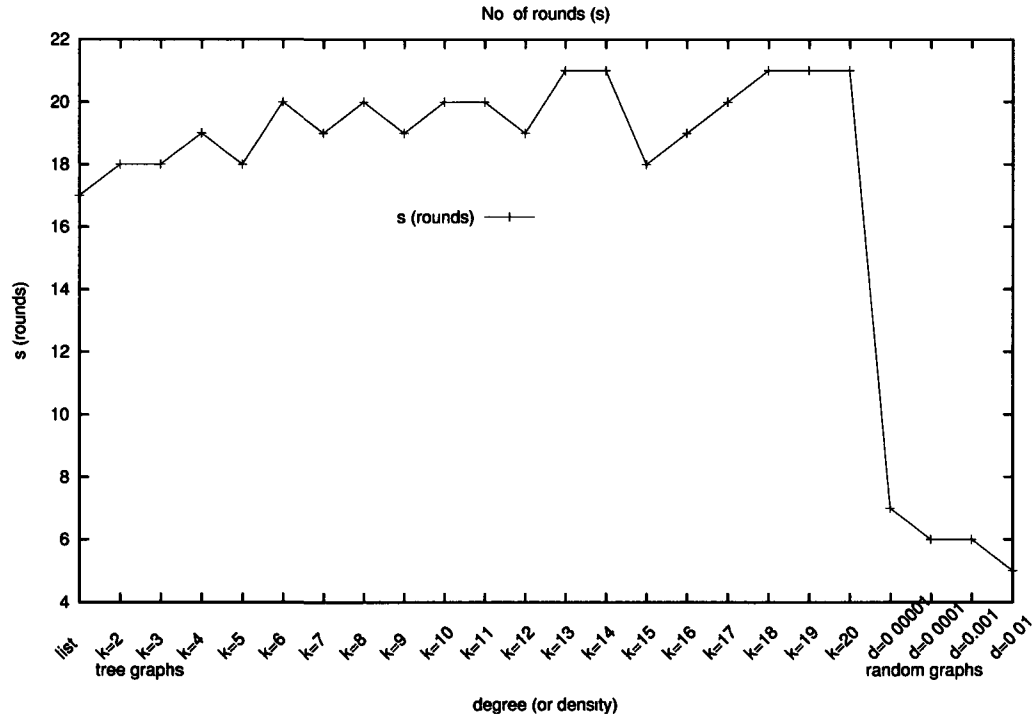


Figure 6.3: No. of Shiloach-Vishkin Iterations for Different Graph Types in CUDA

In the table the CPU-based multi-threaded implementation uses a small number of iterations for list graphs. Our hypothesis is that the small number of parallel threads and the simple linked list structure of the sub-graphs result in pointer jumping that collapses rapidly and with very little concurrent-write interference. When we performed a quick experiment using more threads to execute the CPU implementation we noted an increase in the number of iterations, reinforcing our hypothesis. More detailed analysis is needed in the future to better understand this behaviour.

Figure 6.3 shows the actual number of CUDA iterations for various input graphs: list graphs (degree $k = 1$), tree graphs with degree $k = \{2, 3, \ldots, 20\}$ and random graphs with density $d = \{0.001, 0.01\}$, all of them with $m = 8000000$ edges, we see a clear pattern where the number of iterations in the parallel algorithm execution with the GPU is much smaller for the random graphs that for list/tree graphs. This reinforces the point that an increase in edge density reduces the effort and time needed to identify the components in the parallel implementation.

6.2.2 Comparing Relative Speedup of the CUDA Implementations

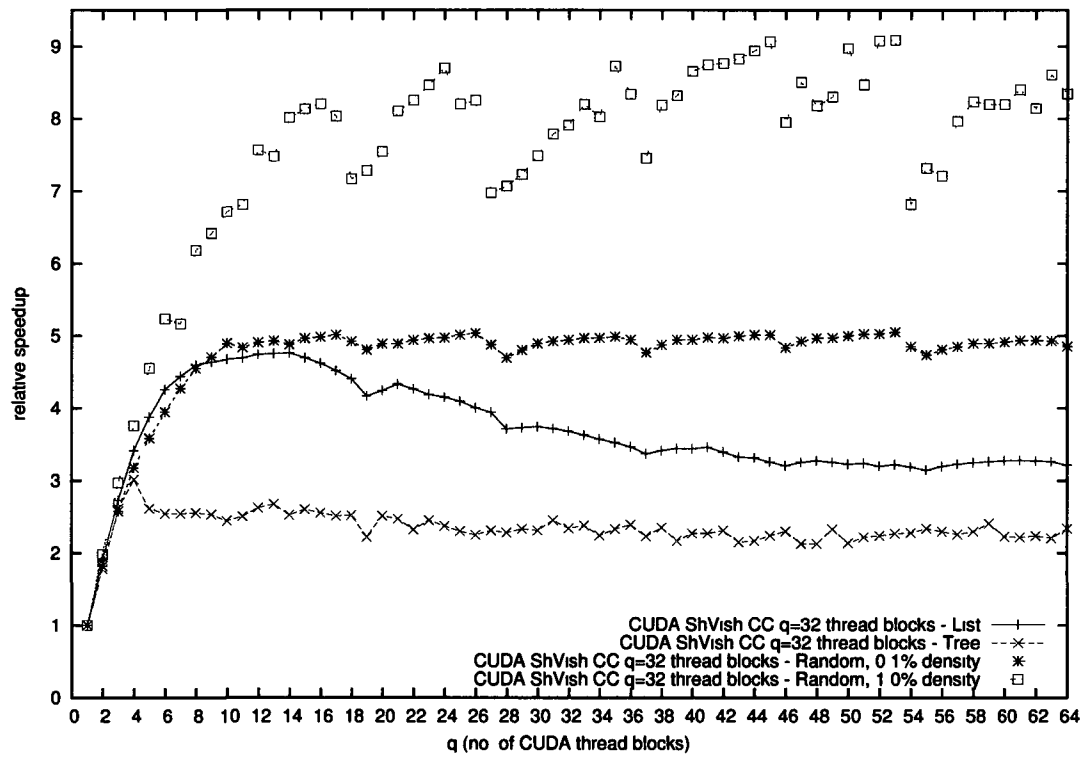


Figure 6.4: Comparing Relative Speedup of Connected Component Counting CUDA Implementation

Figure 6.4 shows in more detail the performance of our GPU implementation in terms of relative speedup as a function of the number of thread blocks. This provides further evidence that the more dense random graphs are processed more quickly than the sparse lists and trees. We can also see that there is a marked difference in the relative speedup when computing the connected components for list graphs versus tree graphs, with lists yielding a better speedup as we increase the number of thread-blocks.

All the speedup curves show no further improvement when we start to use more than $\sim$ 25 thread blocks. We noted similar behaviour when analyzing the parallel list ranking performance with the GPU, where a similar limit can be correlated to the 27 physical SMs available for our nVIDIA GeForce 260 GPU.

The effect on individual kernel performance is further illustrated by Figure 6.5. Complementing the graph in Figure 6.3, the diagram shows the time per iteration spent in each kernel (or part of a kernel). The time spent for Kernels SV2 and SV3 dominates the time for the entire Algorithm 4.5. These kernels, with examine every edge in the graph, appear to perform the most work per iteration for trees, less work per iteration for lists and the least work per iteration for random graphs.

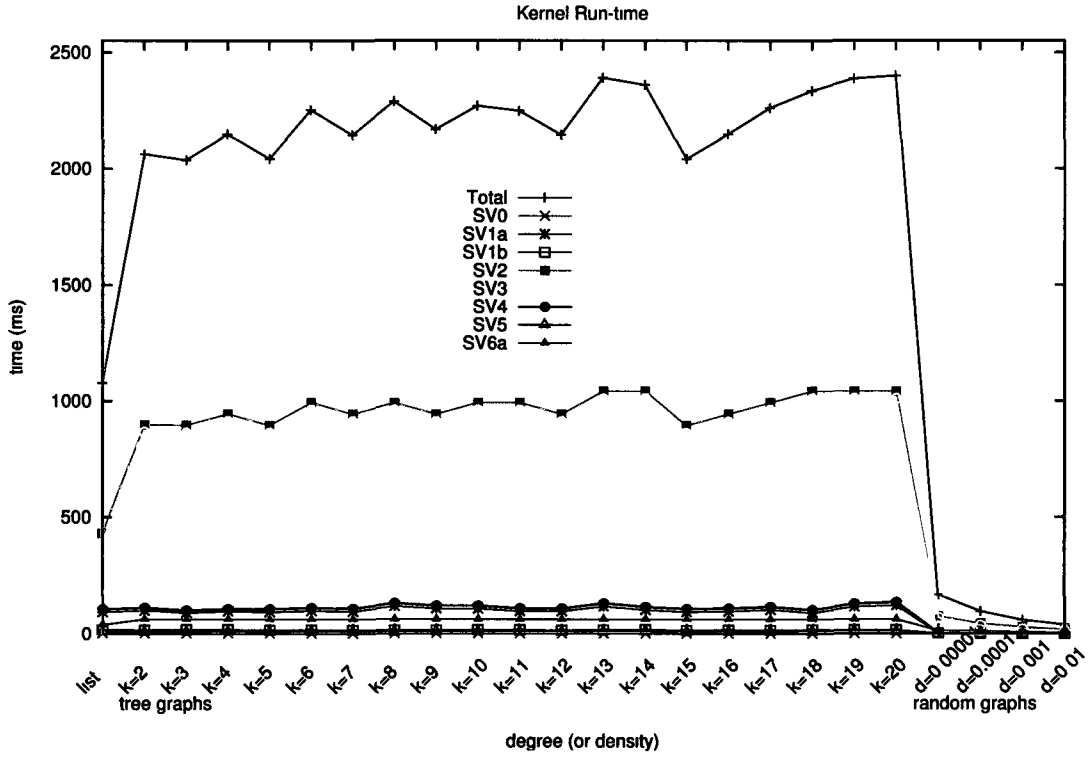


Figure 6.5: Kernel Run-time Comparison of Shiloach-Vishkin CUDA Implementation, by Graph Types

While lists and trees, when processed by the CUDA implementation, require approximately similar number of iterations, why does it take less time per iteration for lists? Observe that, inside Kernels SV2 and SV3 of Algorithm 4.5, for each edge there are 2 and 3 conditions respectively that all need to succeed to enter the if-block where D_s and Q are modified. The failure of any one condition implies that no global memory write operation is performed for that edge in those kernels. With lists the conditions fail more than with trees, thus allowing kernels SV2 and SV3 to do less work and thus require less time to complete.

The total performance is thus the product of number of iterations and time per iteration, which explains why the algorithm performs better for lists than for trees, and best for random graphs.

6.3 Conclusions

In general, our GPU implementation for parallel connected component computation seems to be successful. Despite the highly irregular data access patterns which complicate memory access coalescing, our GPU implementation of Shiloach and Vishkin's algorithm appears to be considerably faster than the sequential method. Note that the sequential method requires only linear work while Shiloach and Vishkin's algorithm requires $O((n+m)\log n)$ work. Another important observation is that performance is different for different types of graphs. Random graphs are processed more quickly than lists, and lists are processed faster than trees. All speedup curves show no further improvement for more than 25 thread blocks which reflects the number of physical SMs available for our nVIDIA GeForce 260 GPU.

Kernels SV0, SV1, SV4 and SV5 of Algorithm 4.5 process vertices and perform $O(n)$ work while Kernels SV2 and SV3 process edges and perform $O(m)$ work. As shown in [17], the algorithm will iterate at most $\left\lfloor \log_{\frac{3}{2}} n \right\rfloor + 2$ rounds but the actual number of rounds will differ according to the actual graph. The number of actual rounds is in general smaller for random graphs than for for trees. That explains the better performance of denser graphs as compared to sparser graphs as compared to trees.

Chapter 7

Conclusion

7.1 Summary Of Results

Our experimental study on parallel *list ranking* and *connected component* algorithms for GPUs indicated that the PRAM algorithms are a good starting point for developing GPU-based implementations.

While they did require non-trivial modifications to ensure good GPU performance, by understanding the characteristics and limitations of the CUDA GPU platform it was possible to implement the original algorithms without impacting their complexity.

It was critical for the efficiency of GPU methods that parallel data access coalescing was maximized. Every step in the algorithms was carefully studied to ensure that the implementation kernels were able to take advantage of memory coalescing where possible. While chasing links in a graph inevitably led to irregular memory access, we were able to ensure by using striding within the kernels that any opportunity for regular access was not wasted.

A more difficult problem was to reduce the number of conditional branching points in the algorithm. As discussed, minimizing conditions is critical for good performance; however replacing conditions is not always possible without impacting either the correctness of the algorithm, or without significantly impacting performance by doing more work unnecessarily. For example, with parallel list ranking we were able to convert a condition into a simple boolean operation, while eliminating any of the conditions in the connected components implementation would have led to more writes to global memory than with the conditions.

While the number of parallel threads that can be executed concurrently on a GPU is large, it was still significantly smaller than the number of PRAM processors needed by the original algorithms. Since the PRAM is a theoretical model with very specific assumptions regarding performance and synchronization and atomicity, the algorithms designed for

PRAM tend to require n processors to process n data elements. In fact the Shiloach-Vishkin algorithm for connected components explicitly requires $m + n$ processors since some steps process n vertices and others process m edges. However, regardless of the availability of millions of threads, the problem inputs will always have more data than processors. It is therefore important for the efficiency of GPU implementations to appropriately assign groups of PRAM processors to GPU threads. Fortunately, by preserving the concurrency and atomicity requirements of the algorithms in the multi-threaded implementations, we were able to execute them using $p < n$ threads on the GPU, and in fact using $p \ll n$ threads on the CPU.

Another important difference between the PRAM and GPUs is that PRAM methods assume zero synchronization overhead while this is not the case for GPUs. Therefore, the number of actually necessary and implemented synchronization points for the GPU implementation needs to be minimized to ensure good performance. In the implementation this manifests in the number of kernels that need to be implemented, since kernel-level separation is needed when executing CUDA implementations that using multiple thread-blocks and thus barrier-synchronization. The good news here is that the overhead of invoking kernels is negligible compared to the amount of work done by the kernels in our implementations.

The observed GPU performance for parallel list ranking and connected components appears to be very sensitive to the total work of the underlying PRAM method. The performance of these algorithms is dominated by the irregular memory access required to traverse the graphs. Since these algorithms also rely on every processor (or thread) being able to access every node or edge at any time, it meant that we were not able to partition the memory allocation into chunks for targeted processing, which in turn meant that these algorithms cannot take advantage of the faster, albeit very limited, shared memory locally available to all threads operating within each SM.

GPU performance is much more sensitive to the constants in the time complexity of the algorithm than parallel implementations for standard multi-core CPUs. This is because GPUs support so many more threads than multi-core CPUs, each with much less work than a thread for the corresponding multi-core CPU method, that the constants in the time complexity have a much larger relative impact.

When executing the $O(n)$ parallel list ranking implementation, we observed an inflection point in performance that indicated a bottle-neck in the system. However no such inflection was observed in the performance of $O(n \log n)$ parallel pointer-jumping-based list ranking implementation. We know that the GPU's hardware scheduler is very efficient at distributing the parallel threads across the SMs and hiding the latency of concurrent memory accesses. Since we maximized the use of coalesced memory access where possible, a detailed performance analysis of the kernel was required to better understand the effect.

This revealed that the inflection point occurs only in the kernel that was unable to take advantage of coalescing, which as a result could generate up to n separate transactions to access n elements. This revealed that the absolute worst case scenario for irregular memory access does in fact pushes the GPU beyond its capabilities, resulting in a definite and drastic impact on the performance.

7.2 Future Work

While we did not observe an inflection in the performance of pointer-jumping-based parallel list ranking algorithm, it is important to note that it doesn't mean there wasn't an inflection. While the algorithm does $O(\log n)$ iterations, each with $O(n)$ work, the pointer jumping results in a natural coalescing of memory access such that in each iteration i the n irregular memory accesses actually access $\frac{n}{2^i}$ distinct memory addresses. However during the first iteration, when $i = 0$, there are n irregular accesses which should result in n separate transactions. It would be interesting to extract the run-time of each iteration and (a) determine if the expected inflection occurs when $i = 0$, and (b) observe the impact on the inflection for subsequent iterations.

The available memory on the GTX 260 (< 1 GB) imposed a restriction on the size of the graphs that could be used in our experiments. As a result it was not clear based on the performance results the characteristic curve of the degradation of the performance following the inflection point. It would be interesting to run the experiments using larger data sets on a GPU with more memory to (a) identify this characteristic and (b) determine if it actually degrades to such a degree that it intersects with the pointer-jumping-based list ranking algorithm.

In section 5.2.7 we analyzed the memory bandwidth of the rank aggregation kernel. While the GPU hardware is capable of memory access throughput of 110 GBps our implementation was unable to exceed 31 GBps. On one hand we were able to show that this loss of throughput was a direct result of the un-coalesced access of the splitter ranks array. The intentionally incorrect test kernel shows that by forcing coalesced access brought the throughput close to 95 GBps. It would be interesting to determine if it would be possible to increase the throughput of the correct implementation any further, perhaps through the use of loop-unrolling and other optimizations. There is room even for the modified, yet incorrect, kernel to reach the desired 110 GBps, and this may indicate some additional gain possible for the rank aggregation kernel.

As mentioned in section 6.1.1, the sequential algorithm for identifying the connected components requires a vertex-representation while the parallel algorithms all operate on an edge-representation of the graph. The conversion process used in our sequential experiments

adds a substantial overhead to the performance of the sequential algorithm. If we simply remove the overhead of the conversion, the comparison becomes unfair as the inputs are quite different. However when we do remove it, we suspect that the result will substantially reduce the performance difference with the parallel GPU implementation for list and tree graphs, and very likely outperform the parallel GPU implementation with the denser random graphs. It would be interesting to (a) identify whether an edge-representation-based sequential algorithm that does $O(m+n)$ at worst is possible, and (b) compare such an implementation with the parallel GPU implementations.

For this thesis we explored two fundamental graph theory problems: list ranking and computing connected components. Having shown that the parallel algorithms to solve these problems can not only be adapted for execution using modern general-purpose GPUs but also that they outperform sequential and parallel implementations that use the CPU, the natural next step is to expand the study to look at other graph theory algorithms. The PRAM literature is a treasure trove of parallel algorithms that until now have simply not been practical. For example, Reid-Miller's parallel list-ranking algorithm was originally implemented using a Cray C90 supercomputer, while today we have been able to implement it on a many-core GPU that can be plugged into any desktop computer. Clearly it would be interesting future research to at the very least attempt to implement these PRAM algorithms and take advantage of their promise of better performance through parallelism.

Appendix A

Experimental Setup and Data Generation

System

The parallel list ranking and connected components experiments were executed on two systems.

- The serial and multi-thread (for PC) experiments were executed on a PC with Intel Core 2 Quad CPU Q6600 @ 2.40GHz with 4 cores and 8GB of RAM. Clearly the serial algorithm only uses one of the cores, while the multi-thread algorithm uses all the cores.
- The CUDA experiments were executed on an identical PC that has a NVIDIA GeForce GTX 260 card installed. The GTX 260 has 27 SMs with a total of 216 cores, approximately 900MB of built-in global memory, a warp size of 32, 16KB of shared memory per thread block, and 16384 registers per block.

All the timings were measured by the experiments through the use of well defined timing functions as described below in section A. Each experiment was run multiple times; both averages and standard deviations were calculated to build the graphs.

Structure of Each Experiment

Each experiment was written using ANSI C using the following general pattern:

1. Include header files.
2. Define and implement CUDA or `pthread` kernels

3. Main routine

   (a) Process experiment parameters supplied via command line

   (b) Load data file to process for the experiment (e.g. linked lists for List Ranking, graphs for Connected Components)

   (c) Copy data from host memory into CUDA global memory

   (d) Start timers

   (e) Invoke the CUDA or `pthread` kernels

   (f) Stop timers

   (g) Copy results from CUDA global memory into host memory

   (h) Verify correctness of results automatically (when possible)

   (i) Send experiment details and timing results to standard output

It is important to note that the performance measurement was done on the computation work and does not include the cost of copying memory to and from CUDA global memory. The intent was to measure the computation time; as well, it is expected that in the near future the cost of moving data between host and device memories will be replace with shared memory addressing techniques, a form of which is already supported by CUDA devices in a limited way today. The measured computation time does include time taken to initialize any memory that is part of the parallel algorithm; i.e. if the algorithm itself requires a particular initial state beyond the input data, then the work done (usually in parallel) is included in the timing.

Measuring time

The CUDA APIs provide a well-defined way to measure the elapsed time by setting up an "event" and measuring the start and end time of that event. The following fragment illustrates the code used to set up such a measurement. The elapsed time is returned in milliseconds.

```
cudaEvent_t start, stop;
float elapsedTime;

cudaEventCreate (&start);
cudaEventCreate (&stop);
// start timing
cudaEventRecord (start);
...
```

```
        // invoke kernels here so we can measure the time
        ...
        // stop the timing
        cudaEventRecord (stop);
        cudaEventSynchronize (stop);

        cudaEventElapsedTime(&elapsedTime, start, stop);
```

With non-CUDA experiments, the time measurement was done differently using a BSD C function to get the time of day with very high precision. The following convenience routine was implement to help simplify the timing.

```
float getTimeNowMillis (void)
{
        static struct timeval first;
        static bool first_set = false;
        if (!first_set) {
                first_set = true;
                (void) gettimeofday (&first, NULL);
        }
        struct timeval now;
        (void) gettimeofday (&now, NULL);
        return ((now.tv_sec - first.tv_sec) * 1000.0f) +
                                ((now.tv_usec - first.tv_usec) / 1000.0f);
}
```

Using the above routine, the following pattern was used to measure the elapsed time of an experiment run without CUDA. A serial experiment was simply run by invoking the function implementing the experiment; a pthread-based parallel experiment was run by setting up the threads before the starting the timing, and then starting the threads and blocking on their completion.

```
        float start, stop;
        float elapsedTime;

        // start timing
        start = getTimeNowMillis ();
        ...
        // invoke the kernels
        ...
        // stop timing
        stop = getTimeNowMillis ();

        elapsedTime = stop - start;
```

Generating Random Numbers

A typical C program can use existing pseudo-random number generators such as `rand()` or hardware random number generators via `/dev/random` on Linux platforms to obtain random numbers. While historically the implementation of `rand()` has been problematic, today's `rand()` implementations on Linux are high-quality linear congruential generators with a period of at least 2^{32}, making them suitable when the number random numbers needed is much less than 2^{32}. The randomized list and graph data generators, which are used to generate lists and graphs with $n \ll 2^{32}$ elements, are run on a Linux PC and using the `rand()` function seeded using a command line parameter.

However none of these methods apply when executing code inside a CUDA kernel since none of the standard C functions can be called from within a CUDA function. Since random numbers are needed in exactly one kernel in the Random Splitter List Ranking implementation, we use an algorithm called KISS by Marsaglia and Zaman[13] that has a very high period of 2^{123} while using straight-forward 64bit integer operations that make it highly performant on a CUDA system. The KISS algorithm has also been implemented for use with the list and graph generators and can be used via a command line option. The following is the C-language implementation of the algorithm for CUDA.

```
typedef struct {
        uint32_t x, y, z;
        uint32_t carry;
} RAND_t;

__device__ void DCU_RandInit (uint32_t seed, uint32_t pIx, RAND_t *kiss) {
        uint32_t pseed = seed + (pIx + (pIx << 16));
        // rotate left by 1 bit
        kiss->x = (pseed << 1) | (pseed >> 31);
        // rotate left by 2 bits
        kiss->y = (pseed << 2) | (pseed >> 30);
        // rotate left by 4 bits
        kiss->z = (pseed << 4) | (pseed >> 28);
        // rotate left by 8 bits
        kiss->carry = (pseed << 8) | (pseed >> 24);
}

__device__ uint32_t DCU_RandNextU32 (RAND_t *kiss) {
        uint64_t t;
        uint64_t a = 698769069LL;
        kiss->x = 69069 * kiss->x + 12345;
        kiss->y ^= (kiss->y << 13);
```

```
        kiss->y ^= (kiss->y >> 17);
        kiss->y ^= (kiss->y << 5);
        t = a * kiss->z + kiss->carry;
        kiss->carry = (t >> 32);
        return kiss->x + kiss->y + (kiss->z = t);
}

__device__ float DCU_RandNext (RAND_t *kiss) {
        return ((float) DCU_RandNextU32 (kiss)) / ((float) 4294967295);
}
```

Generating the Experimental Data

Fairly early in the process of implementing the experiments we decided to generate the data to be used with the experiments and then run the experiments by loading the data files. On one hand, this was motivated by the very large data sets and the time it to generate them for each experiment. At the same time, we ran into stability issues with the CUDA GPU which led to crashed experiments that required re-running experiments and we wanted to minimize the time spent doing that. This decision was particularly relevant when implementing the connected components algorithms where the graph generation itself became very time-consuming for the data sizes we were using.

Randomized List Generator for List Ranking

A random list generator, which is invoked as $genList\,(n, seed)$, where n is the size of the linked list and *seed* specifies the seed to be used with the random number generator. The list is constructed by starting at index 0 and randomly selecting the successive links from the n slots; this represents a properly randomized linked list across the entire n nodes, the worst case linked list for parallel list ranking.

Randomized Graph Generator for Counting Connected Components

A random graph generator used to generate different types of random graphs, as follows.

1. A randomized list graph generator, invoked as $genListGraph(n, count, seed)$, where m is the total number of vertices in the generated graph, *count* specifies the number of lists within the graph, and *seed* specifies the seed to be used with the random number generator. This function generates *count* sub-graphs where each sub-graph is a linked list made up of approximately $\frac{n}{count}$ nodes randomly selected across the n vertices; thus each list is randomly evenly distributed.

2. A randomized tree graph generator, invoked as $genTreeGraph(n, count, k, seed)$, where *count* is the number of trees within the graph and k specifies the maximum degree per node in each tree in the graph. This function generates *count* tree sub-graphs where each tree is made up of approximately $\frac{n}{count}$ nodes randomly selected across the n vertices. To ensure maximum random-ness in the tree node distribution and size, the generator randomly selects *count* root nodes and adds them to a parent list P. Then, until $|P| = n$, it (a) randomly picks a node $x \in P$ with less then k children, (b) randomly picks a node $y \notin P$ from the remaining $n - |P|$ nodes, and (c) makes y a child of x, following which y is added to P.

3. An edge-density-based random graph generator, invoked as $genRandomGraph\ (m, \delta, count, seed)$, where m specifies to the desired number of total edges in the graph, δ specifies the edge-density which is used to compute the number of vertices in the graph $n = \sqrt{\frac{m}{2\delta}}$, *count* specifies the number of sub-graphs and *seed* specifies the seed to be used with the random number generator. This function generates *count* sub-graphs by individually generating random graphs made up of $\frac{n}{count}$ vertices and $\frac{m}{count}$ edges each.

Automating the Experiments

The individual experiments were written as stand-alone C programs that, when executed, would read one data file, perform the experiment on the data, measure the time taken, verify correctness of the experiment and print out a single row of results. For example:

```
GPURULER64 18976 1 64000000 256 2196.485107 0 59.310719 2082.887695 0.031968
59.310719
```

The above is a single RAW result from a single execution of the CUDA GPU Random Splitter List Ranking experiment, using a linked list with 64 million elements generated using a seed of 18976, invoked with 1 thread block of 256 threads. The rest of the numbers are timing results showing the total time as well as a detailed break-down of the time taken by the various kernels that make up each experiment. The overall experiment is made up of multiple such experiments executed using different parameters, the results of which are then averaged and aggregated to produce the final results analyzed in the earlier chapters.

The following Table A.1shows the various list ranking experiments that were run, and the total number of executions per experiment.

The following Table A.2 shows the various connected components experiments that were run for each of the four types of graphs. The table shows the experiments only; each shown

Table A.1: All the List Ranking Experiments

Purpose	Algorithm	N range	p range	Count	Reps	**Runs**
Both	CPU Multi-thread Random Splitter	1M..64M	2..16	3	3	**3072**
Speed-up	GPU Random Splitter 64bit	64M	256...12288	3	3	**144**
Scalability	GPU Random Splitter 64bit	1M 64M	8192	3	3	**192**
Speed-up	GPU Random Splitter 48bit	64M	256 12288	3	3	**144**
Scalability	GPU Random Splitter 48bit	1M..64M	8192	3	3	**192**
Speed-up	GPU Pointer Jumping	64M	256...32768	3	n/a	**384**
Scalability	GPU Pointer Jumping	1M. 64M	8192	3	n/a	**192**
Baseline	CPU Serial Ranking	1M 64M	1	3	n/a	**192**

was run for the four different graph types with 11 different sub-graph sizes per type, and that is taken into account in the total number of runs.

Table A.2: All the Connected Components Experiments (x 4 graph types)

Purpose	Algorithm	M range	p range	Count	Types	**Runs**
Baseline	CPU Serial Vertex-Based	1M 64M	1	3	4 x 11	**8448**
Scalability	CPU Multi-thread Shiloach-Vishkin	1M .64M	4	3	4 x 11	**8448**
Scalability	CPU Multi-thread Shiloach-Vishkin	1M 64M	16	3	4 x 11	**8448**
Speed-up	CPU Multi-thread Shiloach-Vishkin	64M	2 .16	3	4 x 11	**2112**
Scalability	GPU Shiloach-Vishkin	1M .64M	65536	3	4 x 11	**8448**
Speed-up	GPU Shiloach-Vishkin	1M..64M	1048576	3	4 x 11	**8448**
Speed-up	GPU Shiloach-Vishkin	64M	1024 16384	3	4 x 11	**8052**

To automate the execution of these experiments, we selected the use of the Bean Shell[1] scripting language. Bean Shell is an interpretive variant of the Java[2] programming language implemented using Java. It is readily available on all platforms and provides a rich yet easy to use scripting language that is easier to maintain than the traditional Unix-derived scripting languages such as Bash, Perl and Python.

A common utilities script was written for each problem, i.e. one for List Ranking and one for Connected Components, and then individual scripts were created for each algorithm. The scripts themselves were configurable via command-line parameters to control the N, M and p variables and the results were gathered in two forms:

- RAW results from each run was collected and stored. This was important to deal with the GPU crashes that, while infrequent, occurred often enough that the raw data was valuable in helping us identify at which point to re-start the experiment.

- Aggregated and averaged results, which was generated by a separate script that would

[1]See http://www.beanshell.org/ for more information on Bean Shell scripting
[2]See http://www.java.com for more information on the Java programming language

be run to process the RAW data and perform the averaging and standard deviation calculations.

The aggregated results were then converted by hand into plotting data files for use with the **gnuplot**[3] to produce the graphs that are presented and analyzed in the remaining sections of this chapter.

[3]See **http://www.gnuplot.info/** for more information on the extremely awesome **gnuplot** graphing tool

Bibliography

[1] *NVIDIA CUDA Developer Zone.* Available from: `http://developer.nvidia.com/object/gpucomputing.html`.

[2] *ATI Stream Computing User Guide.* AMD, Inc., 2009.

[3] *NVIDIA CUDA Programming Guide.* NVIDIA Corporation, May 2009.

[4] *OpenCL and the ATI Stream SDK v2.0.* AMD, Inc., 2009.

[5] *The OpenCL Specification 1.0.* Khronos OpenCL Working Group, 2009.

[6] Richard Cole and Uzi Vishkin. Deterministic coin tossing with applications to optimal parallel list ranking. *Inform. and Control*, 70(1):32–53, 1986. Available from: `http://dx.doi.org/10.1016/S0019-9958(86)80023-7`, `doi:10.1016/S0019-9958(86)80023-7`.

[7] Frank Dehne and Kumanan Yogaratnam. Exploring the limits of GPUs with parallel graph algorithms. *CoRR*, abs/1002.4482, 2010. Available from: `http://arxiv.org/abs/1002.4482`.

[8] Reinhard Diestel. *Graph Theory (Graduate Texts in Mathematics).* Springer, August 2005.

[9] F.Dehne and S.Song. Randomized parallel list ranking for distributed memory multiprocessors. *Int. Journal of Parallel Programming*, 25 (1):1–16, 1997.

[10] Michael Garland. private communication. NVIDIA Corporation, 2009.

[11] GPGPU.ORG. General purpose computation on graphics hardware. Available from: `http://gpgpu.org/tag/papers`.

[12] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. 28(2):39–55, March 2008. `doi:10.1109/MM.2008.31`.

[13] G. Marsaglia and A. Zaman. The KISS generator. In *Technical Report*. Department of Statistics, University of Florida., 1993.

[14] M. Suhail Rehman, Kishore Kothapalli, and P. J. Narayanan. Fast and scalable list ranking on the GPU. In *ICS '09: Proceedings of the 23rd international conference on Supercomputing*, pages 235–243, New York, NY, USA, 2009. ACM. `doi:http://doi.acm.org/10.1145/1542275.1542311`.

[15] Margaret Reid-Miller. List ranking and list scan on the CRAY C90. In *Journal of Computer and System Sciences*, volume 53, pages 344–356, 1996.

[16] John H. Reif, editor. *Synthesis of Parallel Algorithms*. Morgan Kaufman Publishers Inc., 1993.

[17] Yossi Shiloach and Uzi Vishkin. An O(logn) parallel connectivity algorithm. In *Journal of Algorithms*, volume 3, pages 57–67, 1982.

[18] J.F. Sibeyn, F. Guillaume, and T. Seidel. Practical parallel list ranking In *IRREGULAR'97*, volume 1253, pages 25–36. Springer LNCS, 1997.

[19] Jop F. Sibeyn. Better trade-offs for parallel list ranking. In *ACM SPAA*, pages 221–230, 1997.

[20] Andrew S. Tanenbaum. *Structured Computer Organization*. Prentice Hall PTR, 1990.

[21] Vasily Volkov and James W. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press. Available from: `http://portal.acm.org/citation.cfm?id=1413370.1413402`.

[22] Zheng Wei and J. JaJa. Optimization of linked list prefix computations on multithreaded GPUs using CUDA. In *2010 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, 2010.

[23] J. C. Wylie. *The Complexity of Parallel Computations*. PhD thesis, Cornell University, 1979.