Parallel Matching and Clustering Algorithms on GPUs

Md. Naim

Thesis for the degree of philosophiae doctor (PhD)
at the University of Bergen

2017

Date of defence: June 27th, 2017

To My Parents

Acknowledgements

First and foremost, I would like to thank my supervisor, Prof. Fredrik Manne, for giving me the opportunity to be his PhD student. He has always been encouraging and optimistic while guiding me through my studies. His enthusiasm and patience for countless discussions helped me significantly to finish my PhD projects. His door has always been open to discuss practical issues that I faced during my stay in Bergen. Last but not least, thank you for your patience in reading my thesis meticulously to correct my bad writing. I am deeply indebted to you for your time, encouragement and generosity.

Second, thanks to my colleagues at the department who created a friendly and welcoming environment. In particular, I would like to thank my officemates Torstein Strømme and Paloma Lima for a lively office. Thanks go to current and former master students at the algorithms group.

I want to thank my coauthors Antonino Tumeo, Håkon Lerring, Johannes Langguth and Mahantesh Halappanavar. Thanks to Jan Arne and Markus Fanebust Dregi for their valuable feedback after my presentations at the algorithms seminar.

Thanks to Antonino Tumeo and Mahantesh Halappanavar for inviting me at Pacific Northwest National Laboratory, USA, where I spent two wonderful summers. It was a great environment to learn from other researchers and make friends. Special thanks to Antonino and Johannes for giving me access to their parallel computers.

I would like to thank the administrative staff, Eli Ertresvaag, Ida Rosenlund, Liljan Myhr, Liv Rebecca A Aae, Maria Marta Lopez, Tor M. Bastiansen, and the former members who always helped in practical issues since I moved to Bergen.

Finally, thanks to my parents, siblings, friends and relatives for their continued support and patience.

Bergen, April 2017
Md. Naim

Contents

1 Introduction

Combinatorial scientific computing (CSC) is an interdisciplinary field that deals with discrete algorithms involved in the efficient solutions of problems arising in computational science and engineering (CSE). Even though combinatorial algorithms have long played a crucial rule in CSE, the field formally came to be known as combinatorial scientific computing in 2002 [14]. The focus includes, but is not limited to, efficient parallelization and balanced load distribution of computational problems on large computing systems, optimization, fast implementation of sparse matrix routines, algorithmic differentiation for numerical computations, as well as combinatorial applications to analyze large-scale social networks [48].

Very often, computational models that are used to solve problems in science and engineering are expressed in terms of continuous mathematics. To solve these problems efficiently, research in CSC aims to identify and solve combinatorial subproblems of the models. The solution process often involves the design, analysis and implementation of graph or hypergraph algorithms to produce high performance software [8, 58]. Such algorithms must be able to solve the combinatorial subproblems efficiently or else these might become bottlenecks in the solution process. Since the involved datasets are often very large, a quadratic running time could be too slow to be useful. In such cases an approximation or a heuristic might be a better choice to provide a fast and sufficiently good solution. In addition, it is also often important that the solution method is amendable to parallelization as a sequential algorithm could dominate the overall computation even if it runs in subquadratic time.

Problems arising in scientific computing can be dauntingly large and often too compute intensive to be solved by a single processor. For this reason scientific computations has been one of the leading forces in developing ever larger and more powerful computing systems. But today it is not only supercomputers that offer parallel processing, this is now something that almost all computers and computing systems are capable of. However, in order to take advantage of multiple processing units, it is necessary to invest sufficient effort in developing code that can run in parallel. As a consequence there exists a long tradition within the scientific community for developing parallel methods and programs that can utilize these resources.

To minimize the overall execution time of a parallel program, a problem needs to be evenly subdivided among the available processors and with as little data dependencies as possible. In this way one can achieve an even load balance, while minimizing the amount of time spent on communication. Other important issues that must be considered are to maximize data locality and to reduce the number of cache misses. For dense and well structured problems much of this is well understood, but when solving sparse and more unstructured problems there are other additional concerns that must be addressed. Sparse matrices and

graphs are harder to partition evenly and dependencies between different parts of computations are typically more irregular. For these reasons it often becomes more challenging to obtain a balanced partitioning that preserves data locality.

Many of the involved problems in designing efficient parallel algorithms for sparse problems are combinatorial by nature. Mapping these to new types of hardware with massive computational power has over the last decade been one of the factors that has maintained and elevated the importance of CSC in the scientific community [14, 48].

Even though there has been a tremendous increase in compute power during the last decades, the fundamental way in which parallel systems are programmed has been fairly static. The two most commonly used programming models, MPI and OpenMP, have both been available for at least 20 years, and even though they have evolved over time, the underlying conceptual layout of shared and distributed memory computers has not changed much. However, one thing that has changed is that dedicated co-processors have become common in large scale systems. These processors are used by the central processing units (CPUs) to offload compute and data intensive parts for which they are particularly well suited. Graphic processing units (GPUs) are by far the most common co-processors. They are fundamentally different from traditional CPUs in that they can process up to thousands of independent threads simultaneously. Programming GPUs is challenging for several reasons. For instance, the hardware only supports synchronization between a limited numbers of threads and also introduces new issues related to memory usage. But due to the massive parallelism and compute power offered by GPUs, there has been a substantial interest in porting and also in developing new solutions onto such systems.

This thesis studies graph algorithms to solve some well known combinatorial problems on GPUs. In particular we study matching problems and the related stable marriage problem, both in which the object is to pair together vertices according to similarity measures. As a generalization of these problems we also study the Louvain clustering method [7]. For all of these problems we develop efficient GPU algorithms and show that this gives significant performance gains using fairly inexpensive hardware. For all considered problems we study issues related to resource utilization, synchronization and load balancing. The thesis consists of following four papers:

I. Md Naim, Fredrik Manne, Mahantesh Halappanavar, Antonino Tumeo and Johannes Langguth. *Optimizing Approximate Weighted Matching on Nvidia Kepler K40* In proceedings of the IEEE International Conference on High Performance Computing (HiPC), 2015.

II. Md Naim and Fredrik Manne. *Scalable b-matching on GPUs*, Submitted, 2017.

III. Md Naim, Fredrik Manne, Mahantesh Halappanavar and Antonino Tumeo. *Community Detection on the GPU*. Accepted for presentation at the IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017.

IV. Fredrik Manne, Md Naim, Håkon Lerring and Mahantesh Halappanavar. *On Stable Marriages and Greedy Matching*, In proceedings of the Seventh SIAM Workshop on Combinatorial Scientific Computing (CSC), 2016. Best paper award.

Paper I presents a redesign and GPU implementation of a practical algorithm for the $\frac{1}{2}$-approximate edge weighted matching problem. The main focus is to address key challenges when implementing graph algorithms on modern GPU architectures. Through extensive experiments the paper documents the impact of synchronization and load balancing on the overall running time.

Paper II presents a new greedy algorithm for the maximum edge weighted b-matching problem. This is a generalization of the problem studied in Paper I. The algorithm is highly efficient for solving the problem on a single GPU. It also extends the implementation to multiple GPUs to handle large inputs. The algorithm avoids dynamic load balancing in order to reduce the burden of synchronization and also to increase the independence between threads.

Paper III presents a new GPU algorithm based on the Louvain method for community detection. To maximize the utilization of resources, the presented algorithm parallelizes the access to individual edges rather than vertices. This provides an extra level of parallelism compared to previous implementations. In addition, the GPU algorithm scales the number of threads assigned to a vertex based on its degree. This helps to achieve a fine grained task distribution among the threads.

Paper IV focuses mainly on relating greedy matching with the well known stable matching problem. It shows that several practical matching problems can be formulated as stable marriage problems. In turn, algorithms for the stable marriage problem, can be parallelized using techniques developed for greedy algorithms for matching problems. The paper presents and tests efficient algorithms for the stable marriage problem both on shared memory computers and GPUs.

The remainder of this thesis consists of the following parts: Section 2 provides a short overview of parallel computing systems. Graphics processors are presented in Section 3. This section covers both the evolution of GPUs and also how current systems are designed and programmed. Section 4 contains a short introduction to generic graph libraries for solving graph problems on GPUs. Finally, Section 5 summarizes the main results of the thesis, before papers I-IV are presented.

2 Parallel Computing

High Performance Computing (HPC) generally refers to the use of multiple processors to solve large and complex computational problems. But it's not limited to the use of multiple processors. In addition HPC also encompasses hardware systems, computing platforms, parallel programming tools, along with various other essential components that help tackle large compute and data intensive problems. To satiate the ever-growing demand for more computational power to solve larger problems in a reasonable amount of time, new technologies keep emerging which in turn change the concepts of HPC [62].

In the last three decades the clock speed of consumer processors has increased by a factor of 1000, from a few MHz in 1970s to a few GHz today. Even though it is not only the clock speed that has boosted the computational power, it has been one of the important contributors to performance, along with optimized execution and faster and larger system memory and caches. But in the last decade increased clock speed has not contributed much to extract more performance. This is due to various fundamental limitations such as power consumption and heat production. Since the introduction of Power4 [70], the worlds first multicore processor [59], the core count has increased from one to around ten to circumvent the need to operate at higher frequencies. Sophisticated instruction pipelining and multilevel cache hierarchies have become commonplace- altogether converting a consumer processor into a parallel processor [26, 62, 68].

Supercomputers and data centers have also obtained massive performance gains with improved processors. But most of the performance gain has come from the use of multiple processors, each with multiple cores. Parallel computer architectures are typically classified in two categories depending on their memory organization. In distributed memory systems, processors with local memory are connected through a network and communicate with each other by sending and receiving data. In shared memory systems, a multi-processor generally consists of one or more processing units that share the same system memory. Each such processing unit is normally composed of multiple physical cores. A processing unit can also have hardware capabilities to time-share the cores [12].

Recently, multiprocessors with significant higher number of cores (up to a few hundreds) have become available. These are known as *many-core* multiprocessors. Typically large systems are built by combining shared memory multiprocessors using a message passing network. These are known as hybrid systems. The current most powerful supercomputer in the world (as of November 2016), the Sunway TaihuLight [19], is built using a network of around fifty thousand *many-core* processors each with 260 cores [74].

2.1 Shared Memory Programming

In a shared memory computing system, multiple cores belonging to one or more processing units (CPUs) share the same system resources. The most commonly used programming and execution model for such systems is the multi-threading model (and it's different variants) where each thread has its own private program counter and stack area while all threads share the same address space and system memory. Having common system memory where each thread has read and write access makes it easier to divide and assign the computational problem to participating threads based on their runtime IDs. Even though threads can communicate with each other through the shared memory, concurrent access from multiple threads to a memory location can cause data races when two or more threads access the same memory location. Depending on the chronological ordering of the operations performed on the location, the output of the computation can differ from run to run. Race conditions can be avoided using mutual exclusion, synchronization or locks that are implemented based on special hardware instructions. Similar kinds of techniques are used to protect memory locations and guide the execution flow to achieve correct computation results. Erroneous usage of mutual exclusions and barriers can cause a deadlock among a subset of participating threads and hence demands special attention while writing programs for multiple threads.

Different application program interfaces (APIs) that provide threading abstractions makes it easier to write programs for shared memory computers. One of the most common abstractions is OpenMP [17] which consists of compiler directives and runtime library routines. Using compiler directives, a programmer can specify parallel work sharing regions, mutual exclusions, locks, synchronization barriers, etc. The runtime library of OpenMP creates a thread pool consisting of a specified number of threads that are assigned with tasks created according to compiler directives in parallel work sharing regions.

2.2 Distributed Memory Programming

Large problem instances might not fit in the limited memory of a single multiprocessor. It could also demand more computational power than that available on a single processor for the results to serve practical purposes. In such cases, a computational problem can be distributed across a network of multi-processors (i.e. a distributed memory system) connected using some type of interconnection network. In this computing paradigm, each independent unit of logical execution is referred to as a process. Typically one process is designated to read and then distribute the original problem to each participating process. It is also possible for each process to independently load its data at the start of the computation. During the course of execution, a process explicitly sends and receives data from

other processes using either point-to-point or collective operations [66]. Any synchronization among the processes is achieved by the means of message passing. MPI is the most commonly used library for communication [23].

2.3 Heterogeneous Computing

Shared memory, distributed memory and hybrid systems typically use processors with the same instruction set architecture and are therefore referred to as homogeneous systems. However, different architectures are often well suited for different kinds of tasks and hence employing a particular processor for particular computations can improve the overall performance in terms of time, power consumption and cost of hardware. The usage of processors with different instruction set architectures is known as heterogeneous computing. A heterogeneous system can consist of any combination of architectures, like CPUs, GPUs, field programmable gate arrays (FPGAs), digital signal processors (DSPs), many-core processors such as Xeon Phi and many others [9, 29, 31, 50]. After the emergence of GPUs for general purpose computations, a typical heterogeneous system consists of several CPUs and GPUs [11] where the GPUs are connected to the CPUs using technologies such as a PCI-Express bus [12, 51] or NV-Link [54, 15]. In this setting, a CPU is labeled as a host and a GPU as a device. The part of the application code that executes on the CPU is called host code while the part that executes on the GPU is called device code. The memory belonging to the host and the device are referred to as host memory and device memory, respectively. The relationship between the host and the device is typically a master-slave model, where the host initializes a heterogeneous application, allocates and manages data for the device and offloads highly parallel tasks on the device along with the associated data. The next section describes the CPU-GPU heterogeneous systems in more detail.

3 GPUs For General Purpose Computing

The introduction of CPU-GPU heterogeneous systems has been a major change in the landscape of HPC in the last decade. Starting from the early 2000s, graphics processing units have evolved to become programmable and highly parallel processors with tremendous computational power.

In recent years, graphics processing units (GPUs) have played a major role both in supercomputers and in personal computers to speed up compute and bandwidth intensive applications across many areas of science including engineering, medicine, finance and others. Application developers and researchers have successfully used GPUs to solve problems that demand high throughput in the field of computer vision [18, 21, 28, 32, 36, 60, 65], machine learning [1, 5, 13, 32, 36, 65], computational fluid dynamics [6, 30, 37, 38, 73], ocean geography,

seismic analysis [20, 35], signal processing [25, 45, 79], climate modeling [3, 22, 39, 40] as well as other areas. The significant speedups that have been achieved with relatively low cost GPUs has lead to a sharply increasing trend of using more GPUs in supercomputers and data centers.

In order to understand how GPUs offer high computational throughput, it is helpful to know their evolution over time and how they have developed into their current forms.

Graphics processors were originally designed to perform bitmap operations to accelerate the display for graphical operating systems. In 1992, OpenGL [63, 64] was introduced with the intention of writing 3D graphics programs using platform independent library methods [49]. But it was not until the mid-1990s that the graphics processors had the capability to process 3D graphics.

In computer graphics, various stages of computations are performed to transform three dimensional object models into pixel values that are displayed on a computer screen. These stages are collectively known as the graphics pipeline. A typical set of stages includes transformation of geometry, triangulation, fragmentation of triangles and shading of fragments. The stages of the graphics pipeline that are responsible for the transformation of geometry and the shading of fragments are called vertex shaders [41] and pixel shaders [41], respectively. The vertex shaders typically transform vertices of graphics primitives into screen space while the pixel shaders compute color and other interpolated attribute values belonging to the interior of such primitives.

Until 1997, all of the stages of the graphics pipeline, except vertex shaders, were implemented inside the CPU. However, in the late 1990s, several improvements made it possible to implement more stages inside a graphics processor. For instance, in 1999 Nvidia introduced the GeForce 256, the first GPU with the capability to perform transformation and lighting [43]. This marked the start of moving the entire graphics pipeline inside GPU hardware. One of the most significant innovations from a programmers viewpoint was the introduction of programmable vertex and pixel shaders [77] by Nvidia in the GeForce 3 GPU in 2001 [43, 53]. This gave programmers more control of the computations in their display accelerators. Along with the graphics data, programmers could send programs (i.e. a set of instructions to perform on the data) to the GPUs.

With the advent of programmable shaders [51, 57, 61, 77] it became possible to perform general purpose computations on GPUs rather than just graphics rendering. However, the process was initially a bit tricky. The programmable pixel shaders were originally designed to compute the color values and other attributes of each pixel using input information like coordinate, color, texture and so on [71]. But these input values could be any numerical data and programmers could control how the computations were performed. As the standard graphics APIs were the only ways to program GPUs, programmers had to disguise their computational problems as rendering tasks. They had to reformulate com-

putations using graphics primitives provided by APIs like OpenGL [64, 49] and DirectX [42]. Since then this trend has been known as General-Purpose computation on Graphics Processing Units (GPGPU) within the scientific community [55, 69].

By the early 2000s, different stages of the graphics pipeline were performed by fixed purpose units of the graphics processors. These hardware units were composed of several simple (i.e. without complicated control flow circuities) but independent processing elements (i.e. cores), which gave them the capability to process multiple inputs (e.g. vertices and pixels) in parallel. Since the vertices and pixels have no dependencies among them, a graphics processor used all of its independent processing elements inside a fixed purpose unit. In addition, different fixed purpose units simultaneously processed different sets of vertices and pixels.

Even though very early programmable GPUs offered significant computational power over then-existing CPUs, writing general purpose programs was much more involved. Limited input options, missing capabilities for random memory access and lack of convenient ways to debug made it difficult to program such devices. On top of this, one not only had to learn OpenGL or DirectX, knowledge of graphics-only shading language [34] was also necessary to program shaders. Altogether, these restrictions limited the popularity and use of GPUs for general purpose computations [62]. To overcome these limitations several general purpose programming languages and APIs such as Sh [69], RapidMind [44] and Brook [10] were developed.

Until 2005 display accelerators had separate hardware units for vertex shaders and pixel shaders which led to under utilization and load imbalance in many circumstances [46]. In 2005 ATI Technologies introduced a GPU-unified shader architecture on the Xenos GPU [56] used in the Xbox 360 game console [2] allowing vertices and pixels to be processed on the same hardware unit. This was subsequently adopted by ATI Technologies's TeraScale and Nvidia's Tesla family of GPU micro-architectures as well as many other display accelerator manufacturers. Along with several other architectural enhancements, GeForce 8800 GTX had capabilities to perform general computations including single precision floating point arithmetics and scatter read-write access to memory [41]. In the GPU-unified shader architectures, vertex shaders, pixel shaders and other shaders are simply multi-threaded programs running on the same hardware cores. Nvidia's GPUs with the capability of running different shaders on the same cores are known as Compute Unified Device Architecture (CUDA)-enabled GPUs.

AMD released its GPU programming system CTM (Close To Metal) in 2006 [27, 56], providing a low-level hardware abstraction layer for several ATI GPUs. Subsequently, Nvidia released the CUDA [16] parallel computing platform and programming model that enabled application developers to write programs in C to utilize the massive computational power of CUDA-enabled GPUs [41]. Since then

the field has grown dramatically and successfully attracted developers from many domains of science and engineering with its advanced and easy to use libraries and tools. The current CUDA platform also supports other popular programming languages and open programming standard for parallel computing such as OpenACC [76]. This consists of compiler directives to use accelerators such as GPUs and other many-core processors to speed up the processing of loops and other parts of source code. In recent years, higher level of abstractions and frameworks such as OpenCL [47, 67] have become popular to harness the power of GPUs as well as other types of processors.

3.1 The GPU Architecture and Programming Model

Historically GPUs have been composed of independent but similar components for parallel processing of graphics rendering, with more transistors allocated to arithmetic logic units (ALUs) compared to control logic and caching circuitries. The same set of operations is performed in parallel on each data element without any dependencies among them. This is what has made GPUs into an attractive hardware platform for data parallel applications with large data sets.

In recent years, Nvidia's GPUs have had a dominant position in HPC and big data applications. For this reason we now give a more detailed presentation of their current GPU hardware and computing platform. All work in this thesis has been conducted on Nvidia GPUs using the CUDA computing platform.

Nvidia has developed families of GPU hardware targeting different systems (e.g. GeFore for personal computers, Tesla for data centers and supercomputers, Tegra for mobile and embedded devices), but the architectural features are shared among the product families to accelerate compute intensive tasks.

The Tesla product family has since its introduction undergone significant changes. The code-names for the Tesla GPU micro-architectures are Tesla (same as the product family), Fermi, Kepler, Maxwell and Pascal (in chronological order). For problems presented in this thesis, we have used Kepler and Pascal GPUs and hence we focus on architecture features of these two. To highlight the architectural improvements, we sketch some relevant comparisons.

GPUs are mainly composed of several independent components called *streaming multiprocessor*s (SMs). The number and configuration of the SMs differ from GPU to GPU.

A block diagram of the Pascal GPU is shown in Figure 1. It mainly contains an array of SMs, memory controllers, L_2 cache, global memory and a scheduler. Each memory controller is connected to part of the L_2 cache (even though it's not shown in the figure) and is used to control global memory [15]. The configuration of a single Pascal SM is depicted in Figure 2. For clarity, only half of the SM is shown. Each SM consists of a set of cores, register file, different types of caches and other resources. Each of the cores on the SM supports single precision

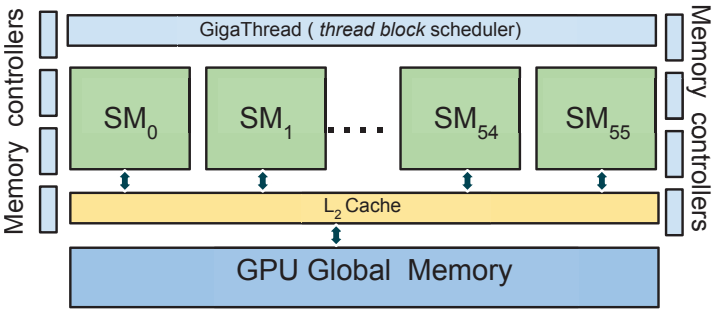

Figure 1: Block diagram of a Pascal GPU

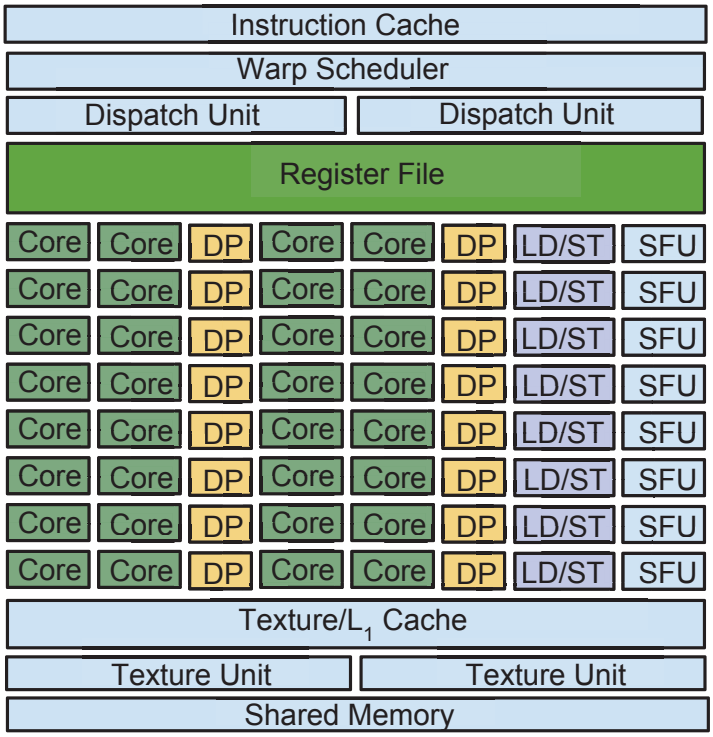

Figure 2: Block diagram of a Pascal SM

(32-bit) arithmetics. Double precision (64-bit) arithmetics are served by double precision units (DP)s while special function units (SFU)s execute intrinsic func-

tions. Load and store units (LD/ST) calculate source and destination addresses of the operands belonging to an instruction. The register file contains a set of registers.

As mentioned earlier, the CUDA programming model was developed to use the unified shader architecture for general purpose computing. It is used to program a heterogeneous system consisting of a host and one or more devices where both the host and the device has their own separate memory. In the model, the functions that are invoked from the host to be executed on the GPU are called *kernel*s. The number of threads that execute in a kernel is specified during the kernel call. These threads are divided into *thread block*s, each with a common shape. All the thread blocks belonging to a kernel call constitute a *grid*. Modern GPUs have the capability to run multiple grids concurrently which provides task parallelism to the programmer. As the kernels execute on device memory, the runtime of the CUDA programming model provides functions to allocate, copy and deallocate device memory. It also provides functions to transfer data between host memory and device memory [16, 52].

Threads are scheduled on the SMs as independent thread blocks. After a thread block has been scheduled on an SM, threads belonging to the thread block are executed in groups of 32. Such groups are called *warp*s, while the hardware units that are responsible to schedule warps to cores are called *warp scheduler*s. Once a warp scheduler selects a warp, instructions belonging to the warp are sent to the cores of the SM by the *dispatch unit*s.

One important feature of the SMs is the programmable memory. This is fast memory that can be used by the threads as random access memory. Threads within a thread block share this memory and hence it is called *shared memory*. Both the shared memory and the register file are located on the GPU chip and are therefore called on-chip memory. In the Kepler GPU, the programmable memory is partitioned between the shared memory and L_1 cache. In the Pascal GPU, there is separate memory for L_1 cache that also caches texture data.

Each SM can run multiple concurrent thread blocks at the same time. A thread block is scheduled on only one SM and stays there until all the member threads of the thread block are done with the designated computations. As the thread blocks are completely independent and scheduled without any particular order, they can be scaled to a GPU with any number of SMs. The scalability of GPUs with varying number of SMs is shown in Figure 3. In the figure, two possible scenarios for the execution of a grid consisting of eight thread blocks is shown, when the grid is executed on a GPU with 2 SMs and on a GPU with 4 SMs. It is assumed that only one thread block can fit on an SM at a time and each of the thread blocks takes the same amount of time to execute. The execution time on a GPU with 4 SMs is then half the time of that on a GPU with 2 SMs.

Once a thread block is scheduled on an SM, on-chip resources such as registers

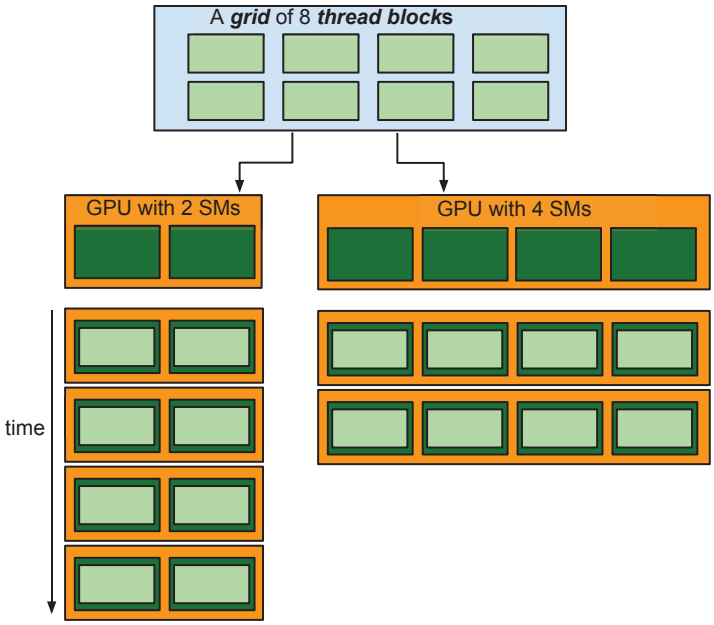

Figure 3: Scalability of CUDA program

and shared memory from the SM are allocated to it. A warp scheduler on the SM selects a warp when there are at least 32 idle cores on the SM and all the arguments to the current instruction to be executed are ready [12, 52]. The logical view of execution flow for a thread block is shown in Figure 4 where the thread block is divided into groups of 32 threads before being scheduled by the warp schedulers to the cores. If a warp is not ready (e.g. waiting for data), it is put into a stall state and other warps gets the opportunity to use the cores. One important point to note is that threads belonging to a stalled warp does not release their on-chip resources until the end of execution. This makes context switching faster than in traditional multi-threaded programming.

The threads within a *warp* execute the same instruction at the same time in a SIMD fashion, but each thread can have an independent execution path using a private instruction address counter [12, 41, 52].

The memory hierarchy of the CUDA programming model is shown in Figure 5. Only one SM with two concurrent thread blocks, each with two threads is shown in the figure. Registers are allocated per thread while shared memory is allocated to an entire thread block. The amount of registers and shared memory on the SM limits how many concurrent thread blocks and warps can run on it. Automatic variables and arrays that are indexed with constant expressions are

Figure 4: Mapping of a *thread block* to physical cores

eligible to be stored in registers. If a thread needs more registers than what is allocated to it, the corresponding data will be stored as private memory of the thread in the global memory of the GPU. This is referred to as *local memory* of the thread. Large private arrays and automatic variables that don't fit in registers are allocated in the local memory.

The amount of shared memory requested by thread blocks is specified during the kernel call. In the CUDA programming model, this amount has to be the same for all thread blocks. Based on the requested amount, the on-chip shared memory of an SM is partitioned among the active thread blocks on the SM [52]. The shared memory of a thread block can be logically divided to its warps or to its threads. If a thread block of a program needs more shared memory than what is available in an SM, the program cannot run on the GPU. Threads within a thread block can cooperate by sharing data in the shared memory. Access to shared memory must be synchronized to avoid race conditions, something which the CUDA API has support for. It is important to note that threads within a thread block can be synchronized but there is no mechanism for inter-thread block synchronization.

The largest and slowest memory of a GPU is the global memory. Each and every thread belonging to any thread block has access to the global memory. Global memory is normally used to store application data and results. As shown in Figure 5, each SM has its private L_1 cache while all SMs inside a GPU share L_2 cache. Memory loads from the global memory are cached while memory stores are not [12]. For simplicity, other forms of memory like texture memory and constant cache have been omitted from the figure.

When the threads in a warp are mapped to the cores of an SM, they can share data between themselves using separate store and load operations in shared memory. Starting from Kepler, subsequent GPUs implement shuffle instructions that makes it possible for the threads within a warp to share data without going through shared memory. The shuffle instructions enable a thread to read from any other thread in the same warp. We found this instruction to be quite useful

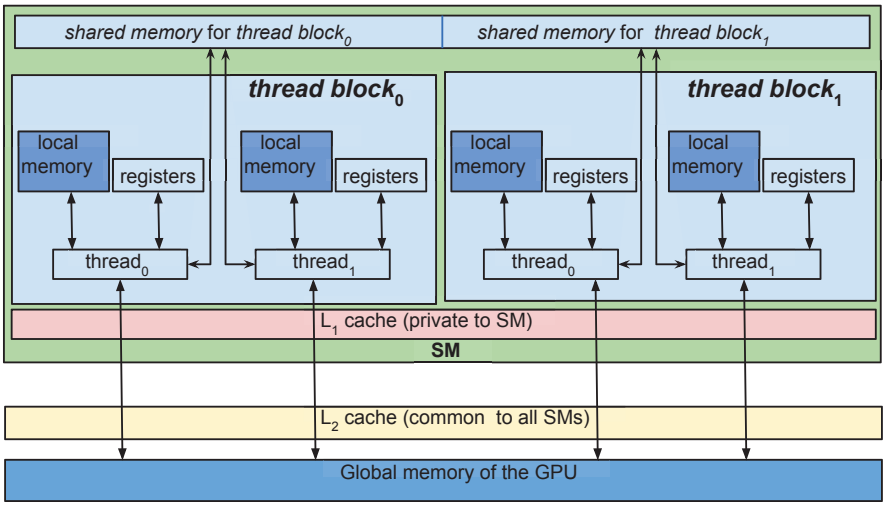

Figure 5: The memory hierarchy on the Pascal GPU

in many of our implementations. A typical scenario would be when each thread within a warp had a separate value that needs to be reduced to a single value using an operator such as sum or min. The use of shuffle instructions helps to reduce the amount of shared memory required per thread and offers improved performance as store-and-load operations are carried out in a single step [15]. When multiple warps inside a thread block are involved in a reduction, the threads inside each warp of the thread block can use shuffle instructions to find a local reduced value and then write this to the shared memory of the thread block. Finally, the values written by the warps can be reduced to a single value.

Atomic operations are important for concurrent programming where multiple threads read and write shared data structures without any predefined order. To ensure a correct result, it is essential that the modification of the same memory location by multiple threads does not lead to different outcomes depending on the order in which these operations are carried out. Fermi and Kepler supports atomic operations that are implemented using a lock-update-unlock pattern while Maxwell and Pascal come with native hardware support for shared memory atomics [15]. Atomic operations are supported both in global and shared memory. For our implementations we used atomic-compare-and-swap (atomicCAS) and atomic add.

Multiple independent SMs inside a GPU, multiple thread blocks inside an SM and multiple threads inside a thread block- this kind of hierarchy directs to split a computational problem into coarse-grained and independent sub-problems that

can be processed in parallel by thread blocks, and each such sub-problem into fine-grained parts that can be processed in coordination by all threads within a thread block [16, 52]. All together, a high number of cores and hardware level parallelism by the means of independent SM units enables modern GPUs to achieve computational throughput for data parallel applications that is competitive compared to traditional shared memory and distributed memory multiprocessors.

Parallel computing systems containing multiple GPUs is already a common trend in HPC. The current way of designing multi-GPU systems is either to have separate GPUs with the same host or to have multiple hosts, each with their own GPU(s) with communication between the hosts handled using MPI. In 2016, Nvidia released their DGX-1 server, a standalone computing node that uses 8 Pascal GPUs connected via fast interconnections to provide up to 85 Tera flops in single precision. This has successfully been used in machine learning and computer vision to reduce the training time to build neural networks [1, 24, 15, 72]. The advantage of the training process is that it mainly involves matrix-vector multiplications and computation of intrinsic functions, something which the GPU architecture has support for.

4 Generic Graph Analytics on GPUs

There has been a substantial effort to develop graph algorithms for GPUs. This has mostly been done for systems with only one GPU, but also for multi-GPU systems. Popular problems that have been studied include breadth-first search (BFS), single-source shortest path (SSSP), betweenness centrality (BC), PageRank (PR) and connected components (CC). Common to all of these is that they can be implemented as some type of graph traversal. When performing a graph traversal there are some common operations that are needed.

- *Advance* Move from one set of vertices to their neighbors.
- *Filtering* Remove specific elements from a set of vertices or edges based on some condition.
- *Compute* A compute operator defines an operation on all vertices or edges in the current input frontier.

As these operations occur in many graph algorithms there has been several efforts to develop generic packages that implement these on GPUs. The intention is then that it will become easier to construct new graph algorithms on GPUs. Medusa was one of the first such generic systems for operating on graphs [80]. It has been tried on PR, BFS, SSSP and maximal bipartite matching. This system can use multiple GPUs connected to the same host. When using multiple GPUs it is advisable to use some kind of graph partitioning software to reduce

the amount of inter GPU communication. Gunrock is a more recent library for developing graph algorithms on a single GPU [75]. This uses a bulk synchronous programming model (BSP) where one performs iterative computational steps interleaved with communication. This has been tried on BFS, SSSP, BC, PR, CC and triangle counting. The BSP model requires synchronization between the computational steps something that might cause threads to idle. To overcome this, the Groute system was developed to support an asynchronous multi-GPU programming environment [4]. In this system the user first sets up a dataflow graph that defines how data is to be moved between the computational devices. During execution the system will then transfer data between these as directed by this graph. Experiments on BFS, SSSP, PR and CC show that it scales on some instances when using up to 8 GPUs connected to a single host. Finally we note the GBTL-CUDA package [78]. This is part of the GraphBLAS effort that aims to reformulate graph problems in the language of linear algebra [33]. It has so far been used to implement BFS and SSSP on a single GPU. This work is a proof of concept that has not yet been optimized for speed.

We note that the algorithms considered in the papers of this thesis do not follow the graph traversal style of programming. One common operation in the considered matching and marriage algorithms is to repeatedly find the current best neighbor for every vertex to match with using a reduction operation. In our work on community detection we also perform a reduction operation on the neighbors of each vertex. But here the neighbors are first reduced to multiple values depending on which community they belong to and then from the set of neighboring communities we pick the best one. It is also important for performance that these operations can be done in an asynchronous fashion while still achieving a deterministic result. Although it might have been possible to implement some of the operations in our algorithms using existing packages, we do not believe that this would have resulted in efficient algorithms.

5 Conclusion

The main focus of this thesis is on developing efficient algorithms on GPUs for certain matching and clustering problems. Through extensive experiments we show that sparse and unstructured problems can benefit greatly from using GPUs as long as the algorithms are carefully designed. Even though none of the presented algorithms are fundamentally new, they still require significant redesign to make them efficient on GPUs. Common to all the developed algorithms is that they emphasis achieving an even load balance and high degree of parallelism, while at the same time avoiding the use of time consuming synchronization operations. Through extensive experiments we verify the performance of the suggested algorithms. In some cases, even a single GPU can outperform tens of multiprocessors

on a state-of-the-art supercomputer.

The area of computing using GPU enhanced systems is changing rapidly. This is especially true for memory management. For instance, simultaneous access to the same memory by a host and a device was not possible until it was recently introduced in the Pascal GPU [15]. However, starting from the early unified shader architecture, the fundamental architectural features of GPUs have not undergone radical changes. Instead what has happened is that features have been added in an incremental fashion while the speed and size of the device has increased gradually. Based on this observation we believe that algorithms that are developed for GPUs today will also be relevant in the near future.

Although the presented algorithms in this thesis can be viewed as proof of concept that carefully designed GPU algorithms for certain graph problems can compete with implementations on traditional parallel supercomputers, it is not to be underestimated that doing so requires substantial effort. For this to become possible on a more regular basis there is a need to continue and further develop higher level abstractions for graph analytics on GPUs. As an example, we believe that the strategy used in Paper III where vertices were grouped according to their degree before being allocated to different thread blocks, is one such technique that could benefit other applications. Also, the use of various reduction operations could be candidates for generic implementation.

It seems clear that GPUs will play a major role in the future of HPC, with large systems containing several interconnected ones. Designing efficient graph algorithms for such systems is still only starting and likely to generate much interesting work in the future. Just like the distinction between traditional shared and distributed algorithms, we believe that one will see a similar division between different GPU algorithms depending on how the underlying devices are interconnected.

Bibliography

[1] Martín Abadi et al. "Tensorflow: Large-scale machine learning on heterogeneous distributed systems". In: *arXiv preprint arXiv:1603.04467* (2016).

[2] Jeff Andrews and Nick Baker. "Xbox 360 system architecture". In: *IEEE micro* 26.2 (2006), pp. 25–37.

[3] Richard Archibald, KJ Evans, and A Salinger. "Accelerating time integration for climate modeling using GPUs". In: *Procedia Computer Science,(same volume)* (2015).

[4] Tal Ben-Nun et al. "Groute: An asynchronous multi-GPU programming model for irregular computations". In: *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2017, pp. 235–248.

[5] James Bergstra et al. "Theano: A CPU and GPU math compiler in Python". In: *Proc. 9th Python in Science Conf.* 2010, pp. 1–7.

[6] Massimo Bernaschi et al. "A flexible high-performance Lattice Boltzmann GPU code for the simulations of fluid flows in complex geometries". In: *Concurrency and Computation: Practice and Experience* 22.1 (2010), pp. 1–14.

[7] Vincent D Blondel et al. "Fast unfolding of communities in large networks". In: *Journal of statistical mechanics: theory and experiment* 2008.10 (2008), P10008.

[8] Erik G Boman et al. "The Zoltan and Isorropia parallel toolkits for combinatorial scientific computing: Partitioning, ordering and coloring". In: *Scientific Programming* 20.2 (2012), pp. 129–150.

[9] Andre R Brodtkorb et al. "State-of-the-art in heterogeneous computing". In: *Scientific Programming* 18.1 (2010), pp. 1–33.

[10] Ian Buck et al. "Brook for GPUs: stream computing on graphics hardware". In: *ACM Transactions on Graphics (TOG)*. Vol. 23. 3. ACM. 2004, pp. 777–786.

[11] Shuai Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*. Ieee. 2009, pp. 44–54.

[12] John Cheng, Max Grossman, and Ty McKercher. *Professional Cuda C Programming*. John Wiley & Sons, 2014.

[13] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. “Torch7: A matlab-like environment for machine learning”. In: *BigLearn, NIPS Workshop*. EPFL-CONF-192376. 2011.

[14] *Combinatorial Scientific Computing:Tutorial, Experiences, and Challenges*. URL: `https://web.stanford.edu/group/mmds/slides2010/Gilbert.pdf`.

[15] NVIDIA Corporation. *NVIDIA Tesla P100 - GP100 Pascal Whitepaper*. Tech. rep. 2016.

[16] *CUDA C Programming Guide*. URL: `http://docs.nvidia.com/cuda/cuda-c-programming-guide`.

[17] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.

[18] Jean-Philippe Farrugia et al. “GPUCV: A framework for image processing acceleration with graphics processors”. In: *Multimedia and Expo, 2006 IEEE International Conference on*. IEEE. 2006, pp. 585–588.

[19] Haohuan Fu et al. “The Sunway TaihuLight supercomputer: system and applications”. In: *Science China Information Sciences* 59.7 (2016), p. 072001. ISSN: 1869-1919. DOI: `10.1007/s11432-016-5588-7`. URL: `http://dx.doi.org/10.1007/s11432-016-5588-7`.

[20] Xiaodong Fu et al. “Investigation of highly efficient algorithms for solving linear equations in the discontinuous deformation analysis method”. In: *International Journal for Numerical and Analytical Methods in Geomechanics* 40.4 (2016), pp. 469–486.

[21] James Fung and Steve Mann. “OpenVIDIA: parallel GPU computer vision”. In: *Proceedings of the 13th annual ACM international conference on Multimedia*. ACM. 2005, pp. 849–852.

[22] Mark Govett et al. “Parallelization and Performance of the NIM Weather Model for CPU, GPU and MIC Processors”. In: (2016).

[23] William Gropp et al. “A high-performance, portable implementation of the MPI message passing interface standard”. In: *Parallel computing* 22.6 (1996), pp. 789–828.

[24] Song Han et al. “Dsd: Regularizing deep neural networks with dense-sparse-dense training flow”. In: *arXiv preprint arXiv:1607.04381* (2016).

[25] Awni Hannun et al. “Deep speech: Scaling up end-to-end speech recognition”. In: *arXiv preprint arXiv:1412.5567* (2014).

[26] John L Henning. “SPEC CPU2000: Measuring CPU performance in the new millennium”. In: *Computer* 33.7 (2000), pp. 28–35.

[27] Justin Hensley. “Amd ctm overview”. In: *ACM SIGGRAPH 2007 courses*. ACM. 2007, p. 7.

[28] S Heymann et al. “SIFT implementation and optimization for general-purpose GPU”. In: (2007).

[29] Francisco D Igual et al. “Unleashing the high-performance and low-power of multi-core DSPs for general-purpose HPC”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 26.

[30] Dana Jacobsen, Julien Thibault, and Inanc Senocak. “An MPI-CUDA implementation for massively parallel incompressible flow computations on multi-GPU clusters”. In: *48th AIAA Aerospace Sciences Meeting Including the New Horizons Forum and Aerospace Exposition*. 2010, p. 522.

[31] James Jeffers and James Reinders. *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[32] Yangqing Jia et al. “Caffe: Convolutional architecture for fast feature embedding”. In: *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[33] Jeremy Kepner et al. “Graphs, matrices, and the GraphBLAS: Seven good reasons”. In: *Procedia Computer Science* 51 (2015), pp. 2453–2462.

[34] John Kessenich, Dave Baldwin, and Randi Rost. “The opengl shading language”. In: *Language version* 1 (2004).

[35] Dimitri Komatitsch et al. “High-order finite-element seismic wave propagation modeling with MPI on a large GPU cluster”. In: *Journal of computational physics* 229.20 (2010), pp. 7692–7714.

[36] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “ImageNet Classification with Deep Convolutional Neural Networks”. In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105. URL: `http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf`.

[37] Frédéric Kuznik et al. “LBM based flow simulation using GPU computing processor”. In: *Computers & Mathematics with Applications* 59.7 (2010), pp. 2380–2392.

[38] Hyungdo Lee et al. “Acceleration of Computational Fluid Dynamics Analysis by using Multiple GPUs”. In: *Proceedings of the International Conference on Bioinformatics & Computational Biology (BIOCOMP)*. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp). 2016, p. 103.

[39] David Leutwyler et al. “Towards Cloud-Resolving European-Scale Climate Simulations using a fully GPU-enabled Prototype of the COSMO Regional Model”. In: *EGU General Assembly Conference Abstracts*. Vol. 16. 2014, p. 11914.

[40] David Leutwyler et al. “Towards European-scale convection-resolving climate simulations with GPUs: a study with COSMO 4.19”. In: *Geoscientific Model Development* 9.9 (2016), p. 3393.

[41] Erik Lindholm et al. “NVIDIA Tesla: A unified graphics and computing architecture”. In: *IEEE micro* 28.2 (2008).

[42] Michael Macedonia. “The GPU enters computing’s mainstream”. In: *Computer* 36.10 (2003), pp. 106–108.

[43] Chris McClanahan. “History and evolution of gpu architecture”. In: *A Survey Paper* (2010), p. 9.

[44] Michael D McCool and Bruce D’Amora. “Programming using RapidMind on the Cell BE”. In: *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*. ACM. 2006, p. 222.

[45] Yajie Miao, Mohammad Gowayyed, and Florian Metze. “EESEN: End-to-end speech recognition using deep RNN models and WFST-based decoding”. In: *Automatic Speech Recognition and Understanding (ASRU), 2015 IEEE Workshop on*. IEEE. 2015, pp. 167–174.

[46] S. Morein et al. *Graphics processing architecture employing a unified shader*. US Patent 6,897,871. 2005. URL: `https://www.google.com/patents/US6897871`.

[47] Aaftab Munshi. “The opencl specification”. In: *Hot Chips 21 Symposium (HCS), 2009 IEEE*. IEEE. 2009, pp. 1–314.

[48] Uwe Naumann and Olaf Schenk. *Combinatorial scientific computing*. CRC Press, 2012.

[49] Jackie Neider, Tom Davis, and Mason Woo. *OpenGL programming guide*. 1993.

[50] Chris J Newburn et al. “Offload Compiler Runtime for the Intel® Xeon Phi Coprocessor”. In: *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*. IEEE. 2013, pp. 1213–1225.

[51] John Nickolls and William J Dally. “The GPU computing era”. In: *IEEE micro* 30.2 (2010).

[52] John Nickolls et al. “Scalable parallel programming with CUDA”. In: *Queue* 6.2 (2008), pp. 40–53.

[53] *NVIDIA GeForce3 Graphics Cards*. URL: `http://www.nvidia.com/page/geforce3.html`.

[54] *NVIDIA NVLINK HIGH-SPEED INTERCONNECT*. URL: `http://www.nvidia.com/object/nvlink.html`.

[55] John D Owens et al. “A survey of general-purpose computation on graphics hardware”. In: *Computer graphics forum*. Vol. 26. 1. Wiley Online Library. 2007, pp. 80–113.

[56] John D Owens et al. “GPU computing”. In: *Proceedings of the IEEE* 96.5 (2008), pp. 879–899.

[57] Matthew N Papakipos et al. *System, method and article of manufacture for pixel shaders for programmable shading*. US Patent 6,532,013. 2003.

[58] Alex Pothen et al. “Combinatorial algorithms for petascale science”. In: (2007).

[59] *Power 4*
The First Multi-Core, 1GHz Processor. URL: `http://www-03.ibm.com/ibm/history/ibm100/us/en/icons/power4/`.

[60] Kari Pulli et al. “Real-time computer vision with OpenCV”. In: *Communications of the ACM* 55.6 (2012), pp. 61–69.

[61] Timothy J Purcell et al. “Ray tracing on programmable graphics hardware”. In: *ACM Transactions on Graphics (TOG)*. Vol. 21. 3. ACM. 2002, pp. 703–712.

[62] Jason Sanders and Edward Kandrot. *CUDA by Example: An Introduction to General-Purpose GPU Programming, Portable Documents*. Addison-Wesley Professional, 2010.

[63] Mark Segal and Kurt Akeley. “The OpenGL Graphics System: A Specification. Silicon Graphics”. In: *Inc.(30 jul 2006)* (1992).

[64] Mark Segal and Kurt Akeley. *The OpenGL graphics system: A specification (version 1.1)*. 1999.

[65] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).

[66] Marc Snir. *MPI–the Complete Reference: The MPI core*. Vol. 1. MIT press, 1998.

[67] John E Stone, David Gohara, and Guochun Shi. “OpenCL: A parallel programming standard for heterogeneous computing systems”. In: *Computing in science & engineering* 12.3 (2010), pp. 66–73.

[68] Herb Sutter. “The free lunch is over: A fundamental turn toward concurrency in software”. In: *Dr. Dobbs journal* 30.3 (2005), pp. 202–210.

[69] David Tarditi, Sidd Puri, and Jose Oglesby. “Accelerator: using data parallelism to program GPUs for general-purpose uses”. In: *ACM SIGARCH Computer Architecture News*. Vol. 34. 5. ACM. 2006, pp. 325–335.

[70] Joel M Tendler et al. “POWER4 system microarchitecture”. In: *IBM Journal of Research and Development* 46.1 (2002), pp. 5–25.

[71] Chris J Thompson, Sahngyun Hahn, and Mark Oskin. “Using modern graphics architectures for general-purpose computing: a framework and analysis”. In: *Microarchitecture, 2002.(MICRO-35). Proceedings. 35th Annual IEEE/ACM International Symposium on*. IEEE. 2002, pp. 306–317.

[72] Yong-hong Tian et al. “Towards human-like and transhuman perception in AI 2.0: a review”. In: *Front. Inform. Technol. Electron. Eng* 18.1 (2017), pp. 58–67.

[73] Jonas Tölke and Manfred Krafczyk. “TeraFLOP computing on a desktop PC with GPUs for 3D CFD”. In: *International Journal of Computational Fluid Dynamics* 22.7 (2008), pp. 443–456.

[74] *TOP 500 The List*. URL: `htm://www.top500.org/`.

[75] Yangzihao Wang et al. “Gunrock: A high-performance graph processing library on the GPU”. In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. ACM. 2016, p. 11.

[76] Sandra Wienke et al. “OpenACCfirst experiences with real-world applications”. In: *European Conference on Parallel Processing*. Springer. 2012, pp. 859–870.

[77] Harold Robert Feldman Zatz, Henry P Moreton, and John Erik Lindholm. *Programmable pixel shading architecture*. US Patent 6,724,394. 2004.

[78] P. Zhang et al. “GBTL-CUDA: Graph Algorithms and Primitives for GPUs”. In: *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. 2016, pp. 912–920. DOI: `10.1109/IPDPSW.2016.185`.

[79] Xiaohui Zhang et al. “Improving deep neural network acoustic models using generalized maxout networks”. In: *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*. IEEE. 2014, pp. 215–219.

[80] Jianlong Zhong and Bingsheng He. “Medusa: Simplified graph processing on GPUs”. In: *IEEE Transactions on Parallel and Distributed Systems* 25.6 (2014), pp. 1543–1552.

I

Optimizing Approximate Weighted Matching on Nvidia Kepler K40

Md. Naim, Fredrik Manne*,
Mahantesh Halappanavar, Antonino Tumeo**
and Johannes Langguth***

Abstract

Matching is a fundamental graph problem with numerous applications in science and engineering. While algorithms for computing optimal matchings are difficult to parallelize, approximation algorithms on the other hand generally compute high quality solutions and are amenable to parallelization. In this paper, we present efficient implementations of the current best algorithm for half-approximate weighted matching, the Suitor algorithm, on Nvidia Kepler K-40 platform. We develop four variants of the algorithm that exploit hardware features to address key challenges for a GPU implementation. We also experiment with different combinations of work assigned to a warp. Using an exhaustive set of 269 inputs, we demonstrate that the new implementation outperforms the previous best GPU algorithm by 10 to 100× for over 100 instances, and from 100 to 1000× for 15 instances. We also demonstrate up to 20× speedup relative to 2 threads, and up to 5× relative to 16 threads on Intel Xeon platform with 16 cores for the same algorithm. The new algorithms and implementations provided in this paper will have a direct impact on several applications that repeatedly use matching as a key compute kernel. Further, algorithm designs and insights provided in this paper will benefit other researchers implementing graph algorithms on modern GPU architectures.

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: {md.naim, fredrikm}@ii.uib.no

**Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O.Box 999, MSIN J4-30, Richland, WA 99352, USA.
Email: {mahantesh.halappanavar, antonino.tumeo}@pnnl.gov

***High performance computing, Simula Research Laboratory, Oslo, Norway. Email: langguth@simula.no

1 Introduction

Given a graph $G = (V, E)$ with vertex set V, edge set E and a weight function $w : E \rightarrow \mathbf{R}^+$, a matching M is a subset of edges such that no two edges in M are incident on the same vertex. A maximum matching maximizes the number of matched edges (cardinality) in M. The objective for a maximum weighted matching is to maximize the sum of the weights of the matched edges. Further, the solutions can be optimal or approximate. In this paper, we only consider half-approximate weighted matching algorithms that guarantee a solution that is at least half of an optimal solution in terms of the cardinality and weight of the matching. We present and compare two main algorithms, Locally-Dominant and Suitor, on two types of architectures, CPUs and GPUs. We also study four variants of the Suitor algorithm on GPUs. The algorithms are listed in Table 1, and are described in Sections 2 and 4.

Table 1: A list of matching algorithms and variants presented and studied in this paper.

Algorithm	**Description**
OMP-LD	Active vertices are stored in a shared queue. Presented in Algorithm 1.
OMP-Suitor	Uses locks for synchronization. Presented in Algorithm 3.
GPU-LD	Thread-per-vertex based implementation of OMP-LD.
GPU-Suitor	Warp-based implementation of OMP-Suitor. Synchronization and load balancing are not used. Presented in Algorithm 4.
GPU-Suitor-SyncLB	Synchronization and load balancing employed among participating warps.
GPU-Suitor-SyncNoLB	Synchronization, but no load balancing among warps.
GPU-Suitor-Hybrid	Synchronize and load balance only for the first few iterations of the Suitor algorithm.

Matching is a fundamental combinatorial problem with many applications in scientific computing, optimization and data analytics. In scientific computing, matchings are used in the solution of sparse linear systems to place large matrix elements on or close to the diagonal [4]; computation of sparse basis for the null space or column space of under-determined matrices [16]; computation of block-triangular form of a matrix [17]. Approximate weighted matchings are used in multi-level graph algorithms for partitioning and clustering during the coarsening

phase [10]; network alignment [11] and community detection [18]. These applications drive the need for efficient parallel implementations of matching algorithms on emerging multicore and manycore architectures. Many of these applications repeatedly compute matchings several hundreds of times during their execution. Therefore, small improvements in matching performance can lead to large gains in the performance of these applications [11].

An important class of manycore architectures are general purpose graphics programming units (GP-GPUs, or simply GPUs) that are not only powerful but also ubiquitous. The Nvidia Kepler K40 presented in Section 3 is currently one of the best manycore platforms for scientific computing. While many significant performance gains for compute intensive applications with regular and predictable memory access patterns have been demonstrated using GPUs, the efficient implementation of irregular applications such as graph algorithms remains a challenge [21]. Highly irregular degree distributions, poor locality in memory accesses, and minimal computation on accessed data make efficient utilization of compute resources challenging. Using approximate weighted matching as a case study for irregular applications, we introduce several algorithmic ideas that can also be adapted for other graph algorithms.

1.1 Contributions

We make the following contributions in this paper:

- Develop new parallel implementations of a weighted matching algorithm (GPU-Suitor) on the Nvidia Kepler architecture. We present four variants of the algorithm and several combinations of threads-per-block and vertices-per-warp.
- Present detailed experimental results using 269 test cases representing diverse applications and sparsity patterns (graph structures).
- Demonstrate the superiority of our algorithms over the previous best algorithm (GPU-LD) on GPUs [7], as well as shared-memory (OpenMP) implementations. We show that the new implementation outperforms GPU-LD by **10** to **100**× for over 100 instances, and by **100** to **1000**× for 15 instances. We also demonstrate up to **20**× speedup relative to 2 threads, and up to **20**× relative to 16 threads on Intel Xeon platform with 16 cores for the same algorithm.

We organize the presentation in this paper as follows. We first present multithreaded matching algorithms targeting shared-memory architectures in Section 2. The Nvidia Kepler K40 is introduced in Section 3, followed by a discussion on the key challenges and our approaches to overcome them. GPU-Suitor, along with its four variants, are presented in Section 4. Experimental results and

analysis are presented in Section 5, followed by a discussion of related work in Section 6, and our conclusions in Section 7.

2 Parallel Weighted Matching

Matching is a classical topic in combinatorial optimization and has been studied extensively [12, 19, 15]. While many variants of the problem exist, we focus on approximate weighted matching for general graphs. In particular, we focus on the work of Halappanavar *et al.* on the Locally-Dominant Algorithm [7], and the work of Manne and Halappanavar on the Suitor Algorithm [13]. Their work itself was built on the pioneering work of many other researchers in the area, whose parallel algorithms systematically evolved from efficient serial algorithms. Due to space restrictions, we only present these two approximation algorithms in this section. We refer you to the respective papers for details.

2.1 The Locally-Dominant (LD) Algorithm

A half-approx weighted matching can be simply computed by considering the edges in a non-increasing order of weights, and by adding all edges that do not violate the matching condition. However, such an approach imposes a serial order on execution. Therefore, the main idea of the LD Algorithm is to identify and match *locally-dominant* edges in parallel. An edge that is heavier than all the edges incident on its end points is called a locally-dominant edge. Algorithm 1 implements this approach. It takes a graph $G = (V, E)$ as input and returns a matching M as output. The algorithm starts by making a call to Procedure ProcessVertex(v) for each vertex (Lines 6 and 7). In Procedure ProcessVertex, for a given vertex, all its neighbors are scanned to find the current heaviest neighbor that has not been matched already. It is important to break ties (duplicate weight) consistently to prevent deadlocks. For this purpose we use vertex indices, which are guaranteed to be unique (Line 5). The identity of the heaviest neighbor for each vertex is then stored in a vector (candidate). After setting the candidate mate for vertex s, say to vertex t, we check if the candidate mate for t is also set to s: candidate[candidate[s]] $= s$ (Line 9). If this is true, we have found a *locally-dominant* edge $e_{s,t}$. We add this edge to M, and the two vertices s and t to the queue (Line 12). Some of the vertices might end up not having any candidates available to match with.

The second part of the execution begins when every vertex has been processed and matched vertices have been added to the queue Q_C. In this part, we iterate until the queue becomes empty (Line 8 in Algorithm 1). Note that at least one edge (the heaviest edge) would get matched in the first loop, and therefore, Q_C is nonempty if M is nonempty. During each iteration of the **while** loop on Line 8,

Algorithm 1 Parallel Locally-Dominant Algorithm. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a queue, Q_C, listing vertices for processing in current step, and a queue, Q_N, listing vertices to be processed in the next step – both the queues list matched vertices; a vector candidate of size $|V|$ that contains the id of the current heaviest neighbor of each vertex.

```
1: procedure LOCALLY-DOMINANT(G(V, E), mate)
2:     for each v ∈ V in parallel do
3:         mate[v] ← ∅
4:         candidate[v] ← ∅
5:     Q_C ← ∅; Q_N ← ∅
6:     for each v ∈ V in parallel do
7:         PROCESSVERTEX(v, Q_C)
8:     while Q_C ≠ ∅ do
9:         for each u ∈ Q_C in parallel do
10:            for each v ∈ adj(u) do
11:                if candidate[v] = u then
12:                    PROCESSVERTEX(v, Q_N)
13:        SWAP(Q_C, Q_N)                          ▷ Swap the two queues
```

Algorithm 2 ProcessVertex

```
1: procedure PROCESSVERTEX(s, Q)
2:     max_wt ← −∞
3:     max_wt_id ← ∅
4:     for each t ∈ adj(s) do
5:         if (mate[t] = ∅) AND (max_wt < w(e_{s,t})) then
6:             max_wt ← w(e_{s,t})
7:             max_wt_id ← t
8:     candidate[s] ← max_wt_id
9:     if candidate[candidate[s]] = s then
10:        mate[s] ← candidate[s]
11:        mate[candidate[s]] ← s
12:        Q ← Q ∪ {s, candidate[s]}
```

we process vertices matched in the previous iterations while adding new vertices to the queue Q_N that become eligible as edges get matched. Note that we only need to process vertices for which the candidate was set to one of the matched vertices (Line 12). This is achieved by adding the newly matched vertices to the queue and checking if any of their unmatched neighbors point to them. If so, those neighbors will have to find new candidates for matching. The algorithm will terminate when the queue becomes empty. The matching is stored in a vector, mate.

The running time of Algorithm 1 is given by $O(|V| + |E|\Delta)$, where Δ is the

maximum degree in G. The worst case happens when a vertex points to all of its neighbors unsuccessfully, and in order to determine the current heaviest neighbor it needs to check the entire list. However, the runtime can be improved to $\Theta(|V|+|E|)$ if the adjacency list for each vertex is provided in a non-increasing order of edge weights. Under this assumption, the current heaviest neighbor of a vertex can be computed in constant time. The amount of parallelism is determined by the number of vertices in Q_C during each iteration of the **while** loop (Line 8). We use the compressed row storage format (CSR) for storing graphs in memory and therefore benefit from caching effects on adjacency lists on platforms with cache hierarchies. On the x86 platforms we use an intrinsic atomic operation `__sync_fetch_and_add()` to add vertices to the tail of the queue.

2.2 The Suitor Algorithm

We now present the Suitor algorithm, the currently best performing half-approx algorithm for weighted matching [13]. An important distinction of the Suitor algorithm relative to the Locally-Dominant algorithm is the absence of a central queue for active vertices that need to be considered for matching in a given iteration. Elimination of the queue makes the algorithm better suited for parallel implementation. Further, by paying careful attention to the vertex that is being processed, the Suitor algorithm proactively avoids unnecessary work. Similar to the Locally-Dominant algorithm, we again use vertex identities to break ties consistently. We also use the notion of locally-dominant edges in order to find candidate edges for matching. The Suitor algorithm is detailed in Algorithm 3.

Parallelism is achieved by distributing the executions of the outer **for** loop (Line 6 in Algorithm 3) among the threads. Multiple threads will concurrently process different vertices, and attempt to find a suitable candidate for each. Since two variables, mate and ws, are shared among the participating threads, there is a need for explicit synchronization among the threads. We use OpenMP locks for synchronization. To prevent conflicts, we define a lock for each vertex (Line 5) and then require that a thread must acquire a *partner*'s lock before executing lines 20 through 29, at which point the lock is released. Immediately after acquiring the lock we test if $heaviest > \mathsf{ws}[partner]$ is still true as it is possible that some other thread might have increased the value of $\mathsf{ws}[partner]$ after *partner* was determined to be the best match for *current*. If this is not the case, then *current* cannot be the suitor of *partner* and we must continue the search for next best candidate (lines 26 to 28). If a given vertex v ends up replacing another vertex w as the mate, then the thread processing vertex v becomes responsible for finding a suitable mate for w. This is shown in lines 21 to 23. The algorithm terminates when all the vertices have been processed. We note that there is no strict order in which the vertices need to be processed.

The running time of the serial Suitor algorithm is $O(\Sigma_{u\in V}|adj(u)|^2) = O(|E|\Delta)$

Algorithm 3 Parallel Suitor algorithm. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Data structures*: a vector ws of size $|V|$ that stores the weight of the current heaviest neighbor of each vertex.

```
1: procedure OMP-SUITOR(G(V,E), mate)
2:     for each u ∈ V in parallel do
3:         mate[u] ← NULL
4:         ws[u] ← 0
5:         omp_init_lock[u]                          ▷ Initialize the lock for each vertex
6:     for each u ∈ V in parallel do
7:         current ← u
8:         done ← False
9:         while (done = False) do
10:            partner ← mate[current]
11:            heaviest ← ws[current]
12:            next ← ∅
13:            for each v ∈ adj(current) do                ▷ For all neighbors of current
14:                if w(current, v) > heaviest and w(current, v) > ws(v) then
15:                    partner ← v
16:                    heaviest ← w(current, v)            ▷ Weight of edge (current, v)
17:            done ← True
18:            if heaviest ≠ NULL then ▷ True only if there is a candidate to match
                   with
19:                omp_set_lock[partner]                          ▷ Lock the partner
20:                if heaviest > ws[partner] then
21:                    if mate[partner] ≠ NULL then        ▷ Check if partner had a
                           previous offer
22:                        next ← mate[partner]
23:                        done ← False
24:                    mate[partner] ← current
25:                    ws[partner] ← heaviest
26:                else
27:                    done ← False                ▷ The partner already has a better offer
28:                    next ← u
29:                omp_unset_lock[partner]                ▷ Release the lock for partner
30:            if done = False then
31:                current ← next          ▷ Continue the search for next best candidate
```

as a node u might have to traverse its neighbor list $|adj(u)|$ times to find a new partner. Note that Δ is the maximum degree in G.

3 Architecture and Challenges

The NVIDIA Tesla K40, based on the Kepler architecture, is currently the most powerful single chip GPU board for scientific computing. There exists a dual chip board, Tesla K80, which trades off some of the peak performance of each chip to obtain higher combined performance and has better compute-to-shared-memory and register ratios, but requires multi-gpu programming techniques for effective utilization. The new GPUs based on the Maxwell architecture are more power efficient, but they are primarily targeted for single-precision computation and gaming applications. Their double precision performance is $\frac{1}{32}$ of the single precision performance.

The Tesla K40 features the GK110B GPU with 15 streaming multiprocessors (SMX). Each SMX integrates 192 single precision units, 64 double precision units and 32 special function units. Each SMX is equipped with 48 KB of read-only cache, as well as 64 KB of on-chip storage, configurable in splits of 48/16, 32/32 and 16/48 KB between L1 cache or shared-memory. The shared-memory is a directly addressable scratchpad memory. K40 also includes 1.5 MB of L2 cache shared among the 15 SMXes. The board provides 12 GB of GDDR5 memory with a datarate of 6 GHz, and all accesses to this memory are cached in L2 automatically. With a core clock of 745 MHz (and turbo clocks up to 875 MHz), the K40 has a theoretical peak performance of 4.29 TFLOPS (5 with turbo) in single-precision and 1.43 TFLOPS (1.66 with turbo) in double precision. Its peak memory bandwidth is 288 GB/s. Applications can reach about 80% of the peak bandwidth on the Kepler architecture [5]. Thus, in throughput oriented computing, it is significantly more powerful than current CPUs both for bandwidth- and compute-bound problems. However, its memory is limited to 12 GB, and even if K40 employs PCI-E v. 3.0 with bandwidths up to 16 GB/s, transfer rates between host and GPU memory are still an order-of-magnitude smaller than those between the GPU and its memory.

The GPU uses the Single Instruction Multiple Thread (SIMT) model, where threads are issued in warps (groups of 32 threads). A warp executes the same instruction at the same time for all its threads. Warps are further grouped into thread-blocks, which are sets of threads scheduled on the same SMX that can share data through the shared-memory. Finally, thread-blocks are organized in a grid, which comprises all threads launched in an application kernel. Since this is an important parameter, we provide results using several values of threads-per-block in Section 5.3.

3.1 Challenges in parallelization

The main challenges that limit performance on current GPU architectures are: (*i*) un-coalesced memory accesses, (*ii*) thread divergence, and (*iii*) load imbalance

among participating threads. The Kepler architecture somewhat mitigates the performance problems of un-coalesced memory accesses due to a better cache architecture. Memory accesses from the same warp that use the read-only cache can obtain maximum memory bandwidth independent of the thread ordering. Furthermore, shared memory can be used to coalesce other memory accesses.

Threads in the same warp are considered divergent if they take different paths in a branching statement. Because of their lockstep execution, all threads in a warp have to wait until threads that have taken different directions completes. Load imbalance among threads keeps warps executing longer on an SMX, thus wasting resources if only a handful of threads are still computing.

Therefore, efficient implementations on GPUs should address all of these challenges in a systematic manner. With reference to the previous implementation of approximate matching, we address these challenges by implementing the algorithm from the perspective of a warp processing a set of vertices instead of one thread processing a set of vertices. We will now briefly explain how we address the challenges in order to build towards the detailed presentation in Section 4.

Un-coalesced accesses in the previous (Locally-Dominant) implementation resulted from a thread-based approach where threads in a warp accessed neighborhoods of different vertices simultaneously, leading to poor locality of memory accesses and under-utilization of data caches. In our implementation, coalesced memory accesses are achieved by exploring the neighborhood of a vertex in parallel using the threads in a warp.

The performance of the previous implementation was also adversely impacted by thread divergence resulting from variations in the size of neighborhoods and vertex-specific decisions. Using the warp-based approach we minimize the impact of thread divergence. In our implementation, thread divergence is caused by threads in a warp attempting to set the suitor for the vertices that they are responsible for. For example, the availability of locks associated with the candidate-vertices that are chosen for a set of vertices in the warp, the branching of if statements based on the weights, and the replacement of one vertex by another vertex that needs to be processed further. While thread divergence is hard to eliminate, we tackle this challenge by exploring several combinations of vertices-per-warp. As presented in Section 5.3, best performance is observed with 8 vertices-per-warp. We note here that all the threads in a warp process the neighborhood of a vertex in tandem. Each vertex in a warp is processed in a sequential order. This is described in Section 4.

Variations in the sizes of the neighborhood (vertex degree) is a major source of load imbalance for approximate matching. This issue was not addressed in the previous implementation. For example, the slowest thread in a warp determined the speed of the warp. However, by using the warp-based approach to process the neighborhood of a vertex, load imbalance from varying vertex-degrees is minimized. We further address load imbalance by redistributing work among

participating warps of a thread block. The impact of this approach is presented in Section 5.3.

4 Weighted Matching on the GPUs

We now present the GPU implementations of the Suitor Algorithm. We build on the presentation of Suitor in Section 2. The GPU implementation of the Locally-Dominant Algorithm is a straight-forward adaptation of the multithreaded (OpenMP) algorithm, where a single thread processes the entire neighborhood of a vertex. In contrast, the GPU implementation of the Suitor Algorithm utilizes all the threads of a warp to processes the neighborhood of a vertex. Consequently, the vertices themselves are processed in serial on a given warp. In this section, we only present the GPU implementation of the Suitor Algorithm. We refer to Halappanavar *et al.* for details on the GPU implementation of the Locally-Dominant Algorithm [7].

The GPU implementation of the Suitor Algorithm utilizes the nested parallel structure of a GPU – where several warps run in parallel, and in turn each warp consists of parallel threads. Consequently, the fundamental difference between the OpenMP and GPU implementations arise from this nested structure. As an illustration, observe that in Algorithm 3 vertices are processed in parallel (Line 6). In contrast, chunks of vertices are assigned to concurrent warps (Line 2) for parallel execution in Algorithm 4. Each warp processes these vertices in serial (Line 5), but the neighborhood of a vertex is processed in parallel (Line 6). In the following discussion, we present intuition and details of the GPU adaptation of the Suitor Algorithm designed to maximize the nested parallelism of a GPU.

Algorithm 4 GPU-Suitor Algorithm. *Input*: graph $G = (V, E)$. *Output*: A matching M represented in vector mate. *Variables*: V_i represents a chunk of vertices based on vertices-per-warp processed on warp i.

```
1:  procedure GPU-Suitor(G(V, E), mate)
2:      Determine the number of warps required based on |V|, vertices-per-warp and
            threads-per-block
3:      while (there are vertices to process) do
4:          for each V_i in parallel do                          ▷ Across warps
5:              for each v ∈ V_i do
6:                  Process adj(v) in parallel                   ▷ In a warp
7:                  Determine best candidate for v in parallel
8:              Set suitor for each candidate of V_i in parallel
9:              Store self or displaced vertices                 ▷ Within a warp
10:             Synchronize across warps; load balance (optional)
```

We present the overall structure of the GPU-Suitor in Algorithm 4. The details are provided in the following discussion, where we also present the intuition

and differences among the four variants of GPU-Suitor. For ease of presentation, we present the algorithm in two phases: (*i*) Initial phase, and (*ii*) Recurrent phase.

Initial Phase

The algorithm starts by moving vertex indices from global memory of the GPU to the local (logical) shared-memory of each warp for a given chunk of vertices assigned to that warp (Line 2). Once a warp has read the indices of the neighbor lists for all vertices of its chunk into shared memory, it finds the best available candidate for each vertex v in the chunk consecutively. All the 32 threads in a warp collectively read the neighbor list of a vertex v (Line 6), and decide the best candidate using a butterfly reduction on the local best read by each thread in the warp (Line 7). As a result of this reduction, each thread of the warp discovers the best candidate and the weight of the corresponding edge. These values are saved in an intermediate buffer in the global memory or registers. After finding and storing all the candidates, the entire warp reads the intermediate buffer containing the stored candidates and corresponding edge weights in a coalesced manner. Each member thread is responsible for setting one vertex as the suitor of its corresponding candidate (Line 8). As detailed in Algorithm 3, a thread in a warp succeeds in setting its vertex as the mate of its candidate vertex if it has a heavier edge (ties resolved consistently). Similar to OMP-Suitor, locks are used in determining the current highest offer for a candidate stored in $\mathsf{ws}[partner]$.

If a thread fails to set a particular vertex v as the suitor of its best candidate c, or it succeeds in replacing a previously assigned vertex u, then we consider those vertices as unsuccessful. The threads of a warp collectively gather all the unsuccessful vertices in consecutive location of shared memory using parallel prefix sum. This is the same part of the memory that was initially used for storing vertex indices assigned to that warp.

The number of vertices assigned to a warp plays a critical role in determining the overall performance. We therefore use different values for vertices-per-warp and show the impact on performance in Section 5.3.

Recurrent Phase

Once a warp completes processing all the vertices in its chunk, it knows how many vertices need to be processed next (Line 9). Processing of these vertices can potentially lead to other vertices becoming eligible for processing in the next iteration. The warp keeps iterating over the recurrent phase until all of its vertices either obtain suitors or cannot be matched (no candidates are available). It is important to note that while each member thread in a warp can try for a different vertex in parallel, only one thread is allowed to set the suitor of the same vertex

at the same time. In order to avoid race conditions arising from this, we use atomic memory operations.

Synchronization and Load Balancing

Synchronization between warps can be avoided in a warp-based implementation, which can lead to minimization of the idle time of the multiprocessors. However, this can lead to an imbalanced load distribution among the warps of a block. For many inputs, we observed that most of the warps finish after a few iterations in the recurrent phase, while a few warps perform a significant number of iterations. To examine these effects further, we implemented intra-block load distribution among participating warps of a thread block. This distribution incurs a cost from synchronization between the warps of a block, and movement of unsuccessful vertices from the shared memory (logical) of one warp to the shared memory of other warp(s). Finding imbalances in load further requires atomic operations and logarithmic to linear operations to find warps with load deficiencies.

Depending on synchronization and load balancing, we have four variations of GPU-Suitor as described below. The differences in performance and their analysis is provided in Section 5.3.

1. **NoSync**: A thread block is not synchronized at all. Each warp works independently to match all of its vertices. This approach has both advantages and limitations. While most of the warps complete their work after a few iterations, only a few warps require tens to hundreds of iterations to complete. These numbers determine the overall performance of the kernel. Thus, load imbalance among the warps of a thread block has a large impact on overall performance. For inputs where most of the warps have approximately the same amount of work, the NoSync approach benefits immensely from avoiding thread synchronization within the blocks, which is an expensive operation on the GPU.

2. **SyncLB**: To alleviate problems arising from load imbalance in NoSync, this approach redistributes load among warps of a thread block during the first k iterations of the recurrent phase, which entails synchronization of all warps and thus of all threads. Here, k is either a predefined number or it is determined based on the number of vertices that haven't been the suitor for other vertices yet. As a result, all warps of a particular thread block are guaranteed to perform more or less the same work for these iterations. For subsequent iterations, warps of a block are synchronized in order to decide on termination of the block without any redistribution of the load. For our implementation, we allow a 25% deviation from the average load when redistributing work during the first k iterations.

3. **SyncNoLB**: In order to examine the impact of synchronization on execution time, this implementation synchronizes warps of a particular block in each iteration, but without balancing the load among them.

4. **Hybrid**: In preliminary experiments, we noticed that after first few iterations with load distribution, subsequent iterations takes more time with synchronization and load distribution than without any synchronization or load balancing. This variant performs load balancing only during the early phases of the execution.

5 Experimental Results and Analysis

We provide the experimental results and analysis in this section. In particular, we demonstrate significant speedup of the new algorithm and implementations relative to the previous best algorithm. We also demonstrate the speedup of GPU implementations relative to CPU (OpenMP) implementations. We further provide results on performance differences between different variants of the GPU implementation. Since we summarize the information in this section, we make the entire result set available at this website: `http://hpc.pnl.gov/people/hala/suitor.html`. The source code is available upon request.

5.1 Hardware Platforms and Dataset

All the experiments are conducted on a server with Intel CPUs and NVIDIA GPUs. The system integrates two sockets and 64 GB of DDR3-1600 memory. Each socket is equipped with an hyperthreaded 8-core Intel Xeon E5-2687W (Sandy Bridge) running at 3.10 GHz (turbo up to 3.8 GHz), thus amounting to a total of 16 cores and 32 threads. Each core has two L1 caches of 32 KB (for instructions and data, respectively) and a private 256 KB L2 cache. Cores in each processor share 20 MB of L3 cache. Each processor has 4 memory channels and a peak memory bandwidth of 51.2 GB/s. We used GCC 4.9.2 to compile our OpenMP implementation of the algorithms. We also used `GOMP_CPU_AFFINITY` to request thread pinning in a `scatter` fashion, and `numactl` for NUMA-aware memory allocation. The GPU is a Tesla K40 system, as described in Section 3, consisting of a GK110B GPU with 15 SMXes (2880 streaming processors) at 745 MHz (turbo up to 875 Mhz) and 12 GB of GDDR5 at 6 GHz. We compiled the code using CUDA version 7.0.

Dataset: We experimented with a large dataset of 269 instances (matrices) downloaded from the University of Florida Sparse Matrix Collection [2]. We downloaded matrices that are symmetric and converted the negative nonzero values to positive. For matrices without weights, weights were added uniformly at random between zero and one, and zero-weight edges were discarded. Given a

matrix A of size $m \times n$, we represent each diagonal entry as a vertex. Each off-diagonal entry is represented as an edge between the vertices representing the row and column of that nonzero entry. The nonzero value is set as the weight of that edge. The diagonal entries are ignored. Thus, the graph representing A has $|m|$ vertices, and the number of edges match the number of nonzeros with diagonal entries ignored. In this paper, we present results only for inputs above a million but less than a billion edges. The number of vertices vary based on the sparsity structure of the matrices. We summarize the size distribution in Figure 1. We also experimented with over 300 problems ranging from hundred thousand to a million edges with similar run time behavior. This large set of inputs represents a wide variety of applications and sparsity patterns. Accordingly, we see a wide variation in run time for different algorithms and their variants. For each input, we run each algorithm at least ten times and capture the minimum time among these runs.

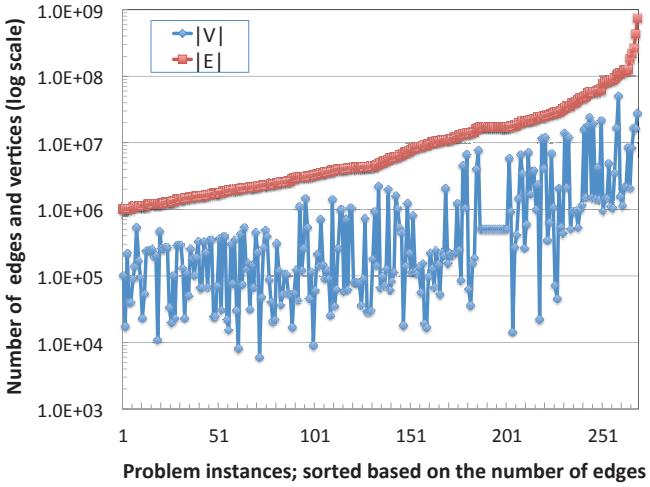

Figure 1: Summary of the sizes of input problems arranged in a non-increasing order of the number of edges. The dataset consists of 269 problems ranging from a million to a billion edges.

5.2 Scaling Comparisons

The two main variants are the Suitor and the Locally-Dominant (LD) algorithms. We implement each algorithm on CPUs using OpenMP (OMP) and on GPUs using CUDA. Thus, we have four main variants to compare: GPU-Suitor, OMP-Suitor, GPU-LD, and OMP-LD. Furthermore, for GPU-Suitor we experiment

with different numbers of threads-per-block and vertices-per-warp as discussed in Section 4. For performance comparisons, we only consider the GPU-Suitor runs with 128 threads-per-block and 8 vertices-per-warp. The impact from variations in these parameters is presented in Section 5.3. Among the four variants presented in Section 4, we present results only for the variant NoSync, the variant with no synchronization and no load balancing. The relative performance of different variants is presented in Section 5.3.

In order to highlight the superior performance of GPU-Suitor, we first present the compute time of GPU and OMP versions of the Suitor and LD algorithms in Figure 2. The run times in milliseconds are presented in log scale on the Y-axis. The times are ordered based on the times of GPU-Suitor. It can be observed that GPU-Suitor outperforms the run time of other variants for most of the problem instances. The OMP run times are for two threads. We present the speedups relative to Suitor and LD algorithms next. The speedup of GPU-Suitor relative

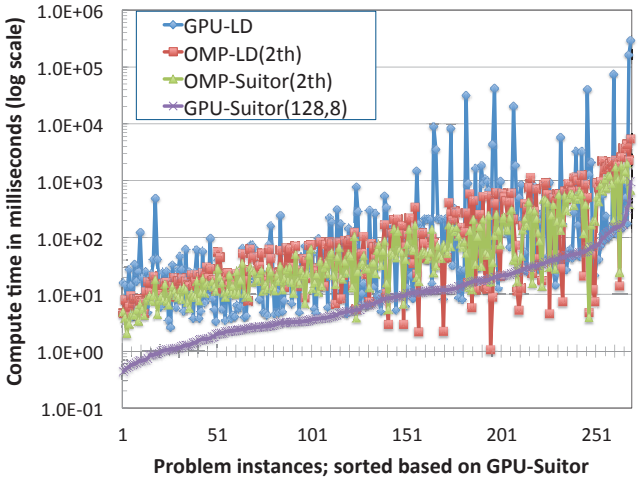

Figure 2: Run time in milliseconds for the two algorithms on two platforms in log scale. The problem instances are shown in non-increasing order of the run times of GPU-Suitor. The OMP times are for two threads.

to GPU-LD and OMP-LD (2 and 16 threads) is presented in Figure 3 on the left, and to OMP-Suitor (2 and 16 threads) on the right. Each speedup curve is ordered individually in non-increasing order of speedup. While we observe positive speedups for GPU-Suitor on a large fraction of the problems against all other algorithms, the largest gains are against GPU-LD. Relative to GPU-LD, the speedups are in the range of 1 to 10× for 115 problems; 10 to 100× for about 100 problems; and above 100× for 15 problems. We run each algorithm for each

input multiple times and pick the minimum time observed among these runs. The experiments were also performed on multiple platforms and we observed similar results. With respect to OMP-Suitor, the best known multithreaded algorithm

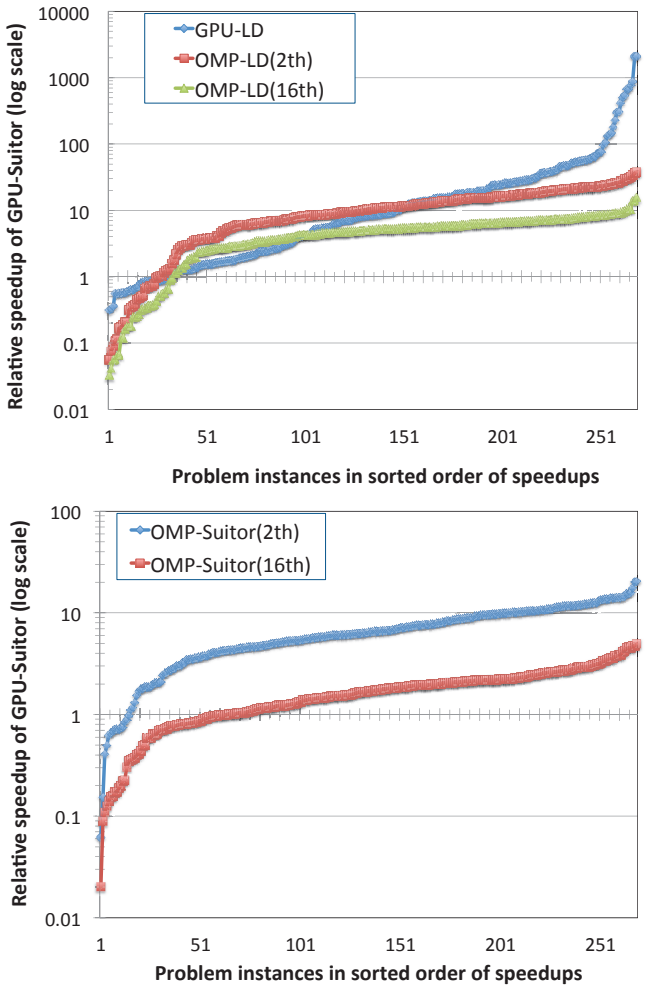

Figure 3: Speedup of GPU-Suitor relative to GPU-LD and OMP-LD on the left, and OMP-Suitor on the right. The speedups are ordered individually for each curve in non-increasing order.

for shared-memory platforms, we observe speedups of up to 20× with two threads

and up to 5× with 16 threads. The speedups with respect to OMP-LD are much higher – up to 40× with two threads and up to 16× with 16 threads.

5.3 Relative Performance

We now present the relative performance of the four variants of GPU-Suitor for different combinations of threads-per-block and vertices-per-warp in this section. The relative performance of variants is presented in Figure 4 in the form of a performance profile. Along the Y-axis we present the fraction of input problems, and along the X-axis we present the relative performance ($\log_2$) to the best variant. For example, we observe that NoSync is the best performing algorithm for about 90% of the problems. However, for about 10% of the problems, NoSync can be up to 4× worse relative to the best variant. We observe that while NoSync stands out as the best variant, the other three variants are similar in performance. The cost of synchronization outweighs the benefits of load balancing.

During the execution of the kernel, most of the blocks finish their work in a few iterations while a few blocks need a significant number of iterations, which in turn determines the overall kernel execution time. If this behavior can be improved without necessitating a large synchronization overhead, GPU-Suitor can perform significantly better.

The second source of difference comes from the variation of threads-per-block and vertices-per-warp. We again present these results in the form of a performance profile captured in Figure 5. We can observe that vertices-per-warp has a large impact on performance, and the threads-per-block has a relatively minor impact. The best performance is obtained with 8 threads-per-warp, and the worst performance is obtained with 256 threads-per-warp, where performance degradation is as high as 30× relative to the best combination. The performance also got worse when less than 8 vertices-per-warp were used.

6 Related Work

Graph algorithm in general, and matching algorithms in particular, are studied extensively. In this section, we present related work that is most relevant to our work. As discussed in Section 2, our work builds on the on multithreaded approximation matching (Locally-Dominant Algorithm) by Halappanavar *et al.* [7], and the Manne and Halappanavar (Suitor Algorithm) [13]. The GPU implementation of Halappanavar *et al.* maintained the general algorithmic structure similar to the implementations on multicore (Intel Xeon) and massively multithreaded (Cray XMT) architectures. In their implementation, the CPU initiates the kernel call considering the number of eligible vertices enqueued in a queue data structure. The actual computation of finding a locally-dominant edge and subsequent

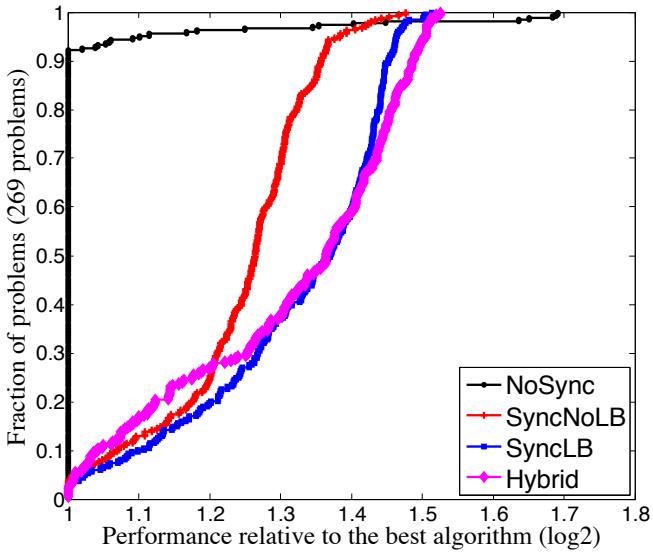

Figure 4: Performance profile depicting the relative performance obtained by different variants of GPU-Suitor. Fraction of input problems are plotted on the Y-axis, and the performance ($\log_2$ scale) relative to the best algorithm are plotted along the X-axis.

matching is done on the GPUs. Matched vertices are concurrently enqueued in a queue for processing in the next iteration. As reported in [8], the increased performance of atomic operations in Fermi-based GPUs provided significant speed ups with respect to a previous generation of hardware. In contrast to the work of Halappanavar *et al.*, we adapt the algorithm of Manne and Halappanavar in this work, which is superior in performance [13]. Further, we consider different combinations of vertices-per-warp and threads-per-block for four variants of the algorithm. The utilization of shared memory is also new in our implementation.

Vasconcelos and Rosenhahn presented GPU adaptation of Bersekas's auction-based algorithm in [20]. However, their implementation is adapted for maximum (unweighted) matching and is limited to bipartite graphs. Fagginger Auer and Bisseling adapt the work of Vasconcelos and Rosenhahn to general graphs by implicitly finding a bipartite graph based on randomly coloring the eligible vertices blue or red [6]. While the blue vertices try to match with one of the neighboring red vertices by bidding, the red vertices select only one bid from the received bids. There are several limitations to this approach, which is not suitable for weighted matching. In our experiments, we found that the quality (in terms of the weight)

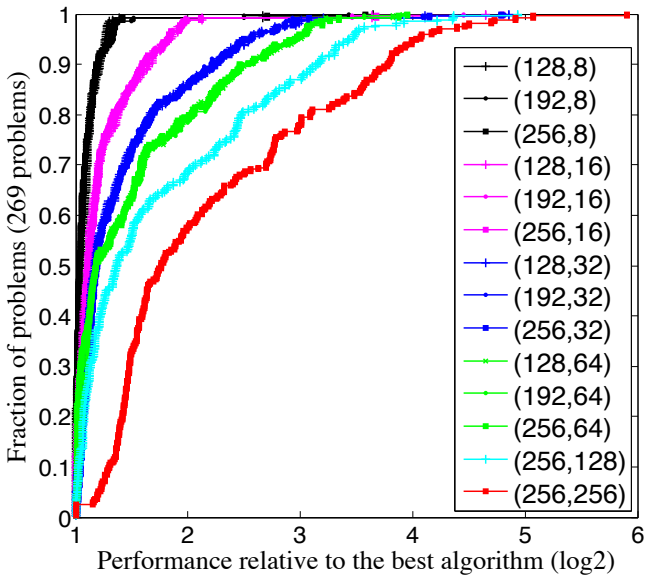

Figure 5: Performance profile depicting the relative performance obtained by different combination of threads-per-block $(128, 192, 256)$ and vertices-per-warp $(8, 16, 32, 64, 128, 256)$. Fraction of input problems are plotted on the Y-axis, and the performance ($\log_2$ scale) relative to the best algorithm are plotted along the X-axis.

of the matching computed with this approach was significantly lower relative to our algorithms. We also note that the algorithm of Auer and Bisseling repeatedly considers all the vertices and is therefore not (work) efficient, but scales better. Xu *et al.* use the algorithm of Auer and Bisseling, along with several other graph algorithms [21]. We address some of the performance issues raised by them in our work. In a similar vein, Devici *et al.* also present adaptation of maximum matching on GPUs [3].

A recent unpublished work of Cohen *et al.* is also relevant to our work [1]. Using a *hand-shaking approach*, Cohen *et al.* identify locally-dominant edges similar to the approach used in Halappanavar *et al.* They further adapt this algorithm by enabling k-way handshake that builds a subgraph by restricting the maximum degree of any vertex to k (k top neighbors of a vertex). However, the performance gain from this extension is not observed in all the inputs. Further, their implementation is specific to bipartite graphs.

Our work benefited from the work of Hong *et al.* that introduced the notion of utilizing the threads of a warp to process the neighborhood of a vertex [9]. As

discussed in several parts of this paper, we present the benefits of this approach over the thread-per-vertex approach of Halappanavar *et al.* Considerable amount of literature exists on implementations of other graph kernels such as breadth-first search, single-source shortest-path, graph coloring and betweenness centrality on modern GPU platforms. We again refer to the work of Xu *et al.* on this topic. An important area of relevant work is on multi-GPUs. While we restricted our focus on a single GPU in this work, we plan to explore multi-GPU implementations in the near future. We refer the work of Mastrostefano and Bernaschi on distributed multi-GPU implementations of the breadth-first algorithm [14].

We conclude this section by noting that to the best of our efforts, this is the first extensive work on implementing the current best approximate matching algorithm of the current best GPU platform using an exhaustive set of variations and input problems.

7 Conclusions

Using weighted matching as a case study, we presented different strategies to exploit GPU architectures such as coaleased memory access, minimizing thread divergence and load balancing. Supported by experimental results we demonstrated not only excellent scaling on the Nvidia Kepler K-40 platform, but also competitive performance relative to traditional multi-core architectures. We demonstrated speedups relative to previous best GPU algorithm by 10 to 100× for over 100 instances, and from 100 to 1000× for 15 instances. We also demonstrated up to 20× speedup relative to 2 threads, and up to 5× relative to 16 threads on Intel Xeon platform with 16 cores for the same algorithm. We showed the impact of algorithmic variations such as synchronization and load balancing on performance. We also showed the impact of different combinations of threads-per-block and vertices-per-warp on performance.

We conclude this paper by observing that as power limitations impose severe restrictions on architecture design, driving future systems toward larger numbers of weaker cores, this work on a prototypical irregular application (graph algorithm) demonstrates promise of better performance on future low-power architectures. We believe that the algorithmic ideas presented in this paper that exploit architectural features will benefit other researchers implementing their applications on manycore architectures, and that the lessons learned will be applicable to future generations of architectures and other graph algorithms.

Acknowledgment

A part of this work was supported by DoD under project 63810 and the Center for Adaptive Super Computing Software Multithreaded Architectures (CASS-

MT) at the U.S. Department of Energy Pacific Northwest National Laboratory (PNNL). PNNL is operated by Battelle Memorial Institute under Contract DE-AC06-76RL01830. We thank Oreste Villa for lively discussions and access to previous GPU implementation of matching algorithms.

References

[1] Jonathan Cohen and Patrice Castonguay. Efficient graph matching and coloring on the GPU. Presentation at NVIDIA GTC conference, 2012.

[2] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[3] Mehmet Deveci, Kamer Kaya, Bora Uçar, and Ümit V. Çatalyürek. GPU accelerated maximum cardinality matching algorithms for bipartite graphs. *CoRR*, abs/1303.1379, 2013.

[4] Iain S. Duff and Jacko Koster. The design and use of algorithms for permuting large entries to the diagonal of sparse matrices. *SIAM J. Matrix Anal. Appl.*, 20(4):889–901, 1999.

[5] Florent Duguet. Kepler vs Xeon Phi : Nos mesures - et leur code source complet. `http://www.hpcmagazine.fr/en-couverture/kepler-vs-xeon-phi-nos-mesures`, June 2013.

[6] BasO. Fagginger Auer and RobH. Bisseling. A gpu algorithm for greedy graph matching. In Rainer Keller, David Kramer, and Jan-Philipp Weiss, editors, *Facing the Multicore - Challenge II*, volume 7174 of *Lecture Notes in Computer Science*, pages 108–119. Springer Berlin Heidelberg, 2012.

[7] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perf. Comput. App.*, 26(4):413–430, 2012.

[8] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perform. Comput. Appl.*, 26(4):413–430, November 2012.

[9] Sungpack Hong, Sang Kyun Kim, Tayo Oguntebi, and Kunle Olukotun. Accelerating cuda graph algorithms at maximum warp. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming*, PPoPP '11, pages 267–276, New York, NY, USA, 2011. ACM.

[10] George Karypis and Vipin Kumar. Analysis of multilevel graph partitioning. In *Supercomputing '95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing (CDROM)*, page 29, New York, NY, USA, 1995. ACM.

[11] Arif M. Khan, David F. Gleich, Alex Pothen, and Mahantesh Halappanavar. A multithreaded algorithm for network alignment via approximate matching. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 64:1–64:11, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[12] L. Lovasz. *Matching Theory (North-Holland mathematics studies)*. Elsevier Science Ltd, 1986.

[13] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *Proceedings of the 2014 IEEE 28th International Parallel and Distributed Processing Symposium*, IPDPS '14, pages 519–528, Washington, DC, USA, 2014. IEEE Computer Society.

[14] Enrico Mastrostefano and Massimo Bernaschi. Efficient breadth first search on multi-gpu systems. *J. Parallel Distrib. Comput.*, 73(9):1292–1305, September 2013.

[15] Burkhard Monien, Robert Preis, and Ralph Diekmann. Quality matching and local improvement for multilevel graph-partitioning. *Parallel Comput.*, 26(12):1609–1634, 2000.

[16] Ali Pinar, Edmond Chow, and Alex Pothen. Combinatorial algorithms for computing column space bases that have sparse inverses. *Electronic Transactions on Numerical Analysis*, 22:122–145, 2006.

[17] Alex Pothen and Chin-Ju Fan. Computing the block triangular form of a sparse matrix. *ACM Trans. Math. Softw.*, 16(4):303–324, 1990.

[18] E. Jason Riedy, Henning Meyerhenke, David Ediger, and David A. Bader. Parallel community detection for massive graphs. In *Proceedings of the 9th International Conference on Parallel Processing and Applied Mathematics - Volume Part I*, PPAM'11, pages 286–296, Berlin, Heidelberg, 2012. Springer-Verlag.

[19] Alexander Schrijver. *Combinatorial Optimization - Polyhedra and Efficiency*. Springer, 2003.

[20] Cristina Nader Vasconcelos and Bodo Rosenhahn. Bipartite graph matching computation on gpu. In *Proceedings of the 7th International Conference on Energy Minimization Methods in Computer Vision and Pattern Recognition*, EMMCVPR '09, pages 42–55, Berlin, Heidelberg, 2009. Springer-Verlag.

[21] Qiumin Xu, Hyeran Jeon, and M. Annavaram. Graph processing on gpus: Where are the bottlenecks? In *Workload Characterization (IISWC), 2014 IEEE International Symposium on*, pages 140–149, Oct 2014.

III

Community Detection on the GPU

Md. Naim, Fredrik Manne*,
Mahantesh Halappanavar, and Antonino Tumeo**

Abstract

We present and evaluate a new GPU algorithm based on the Louvain method for community detection. Our algorithm is the first for this problem that parallelizes the access to individual edges. In this way we can fine tune the load balance when processing networks with nodes of highly varying degrees. This is achieved by scaling the number of threads assigned to each node according to its degree. Extensive experiments show that we obtain speedups up to a factor of 270 compared to the sequential algorithm. The algorithm consistently outperforms other recent shared memory implementations and is only one order of magnitude slower than the current fastest parallel Louvain method running on a Blue Gene/Q supercomputer using more than 500K threads.

1 Introduction

Community detection is the problem of classifying nodes in a network into sets such that those within each set have more in common than what they share with nodes in other sets. Although there is no standard mathematical definition for what makes up a community, the modularity metric proposed by Newman and Girvan is often used [19]. This is a measurement of the density of links within communities as compared to how connected they would be, on average, in a suitably defined random network. For an overview of different algorithms and metrics for detecting communities see [10].

The Louvain method [1] is one of the more popular strategies for community detection. It uses a greedy approach that optimizes modularity by iteratively moving nodes between communities. Once a sufficiently stable solution is obtained the communities are agglomerated to form a new network on which the process is repeated. Thus the

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: {md.naim, fredrikm}@ii.uib.no

**Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O.Box 999, MSIN J4-30, Richland, WA 99352, USA.
Email: {mahantesh.halappanavar, antonino.tumeo}@pnnl.gov

method computes a multilevel clustering hierarchy of the original network and has applications in several diverse areas such as analyzing social networks [15, 23], mapping of human brain networks [17, 28], and classification of scientific journals [25].

Although the basic Louvain method is relatively fast, computing on large networks that can contain billions of nodes still takes significant time or might not be possible due to memory constraints. Timing issues can also be critical in areas such as dynamic network analytics where the input data changes continuously [6]. For these reasons there has been an interest in developing fast parallel versions of the Louvain method. This has resulted in a wide variety of algorithms suitable for different types of parallel computers. Common to all such implementations is that they are node centered, meaning that only one computational thread is used to process any node of the network. This strategy can give rise to uneven load balance if the network contains nodes of highly varying degrees and if a fine grained partitioning is needed. As this is a common problem in many parallel algorithms operating on sparse graphs and matrices, it has been suggested that one should divide the network using a link based partitioning algorithm [3, 24]. In this way the links adjacent to a node will be split across multiple processing units. However, such partitioning schemes are themselves costly to run and can also gives rise to more complex parallel algorithms.

In this paper we present a new highly scalable GPU algorithm based on the Louvain method. Unlike previous GPU implementations we have parallelized all stages of the method. Compared to all other parallel algorithms we also parallelize the access to individual edges. This allows us to obtain an even load balance by scaling the number of threads assigned to each node depending on its degree. This is done by first putting the nodes in buckets based on their degree. The number of threads assigned to each vertex then depends on which bucket it is in. This binning technique is important for efficient utilization of the compute resources on the GPU. Coordination between the threads operating on the same node is achieved by assigning each node either to threads in the same warp or to all the threads in a thread block.

The main contributions of this paper are:

- We present the first truly scalable GPU implementation based on the Louvain method. This is also the first parallel implementation that parallelizes the access to individual edges, and thus giving a more fine tuned load balance.

- Extensive experiments show that the algorithm gives a speedup of up to a factor of 270 compared to the original sequential Louvain method, without sacrificing solution quality. The algorithm outperforms other recent shared memory implementations, and is only one order of magnitude slower than what is achieved with state of the art supercomputers.

The rest of the paper is organized as follows. In Section 2 we give definitions and describe the sequential Louvain method. Next, in Section 3 we review previous efforts at designing parallel implementations based on the Louvain method. Section

4 presents our new GPU algorithm including how memory and thread allocation is performed. Section 5 contains results from experiments using the new algorithm including comparisons with other parallel algorithms. Finally, we conclude in Section 6. It is assumed that the reader has some knowledge of how GPUs work.

2 The Louvain Method

We model a network using a graph G with vertex set V, edge set E, and a positive weight w_e on each $e \in E$. Let C define a partitioning of V into k disjoint communities $c_1, c_2, \ldots, c_k$. We denote the community that vertex i belongs to by $C(i)$. Let further $k_i = \sum_{j \in N[i]} w_{(i,j)}$. Thus k_i is the sum of the weights of the edges incident on vertex i including any self-loops. For $c \in C$ let $a_c = \sum_{i \in c} k_i$ and let $m = \sum_{e \in E} w_e$ be the sum of all edge weights. Finally, let $e_{i \to C(i)}$ be the sum of the weights of all edges from vertex i to other vertices in community $C(i)$, that is $e_{i \to C(i)} = \sum_{j \in C(i)} w_{i,j}$.

The modularity of a partitioning C measures how much more densely connected the nodes within each community are compared to how connected they would be, on average, in a suitably defined random network. It takes on a value between -1 and 1 and is defined as follows [18].

$$Q = \frac{1}{2m} \sum_{i \in V} e_{i \to C(i)} - \sum_{c \in C} \frac{(a_c)^2}{4m^2}. \tag{1}$$

Finding the partitioning that gives the highest modularity is an NP-hard problem [2].

The gain in modularity when a vertex i is moved from its current community $C(i)$ to another community $C(j)$ is given by

$$\Delta Q_{i \to C(j)} = \frac{e_{i \to C(j)} - e_{i \to C(i) \setminus \{i\}}}{m} + k_i \frac{a_{C(i) \setminus \{i\}} - a_{C(j)}}{2m^2}. \tag{2}$$

The Louvain method is a multi-stage algorithm for computing a hierarchical clustering of the vertices in G. Each stage consists of two phases. In the first phase, the algorithm employs an iterative greedy approach to compute a clustering that optimizes the modularity as given by Eq. 1. In one iteration each vertex is considered in turn and moved to the community that will give the largest increase in modularity as given by Eq. 2. If no positive gain can be obtained the vertex will remain in its current community. The algorithm continues iterating over the vertices until no further gain can be obtained or if the gain falls below some predefined threshold. At this point the first phase ends.

In the second phase the graph is aggregated by merging the vertices of each community into a single new vertex. If there are multiple edges between vertices in two communities then these are also merged into one edge between the corresponding new

vertices. Any edges within a particular community are similarly merged into a self-loop for the corresponding new vertex. The weight of each new edge is set to the sum of the weights of the individual edges that were merged into it. The new aggregated graph is then iteratively given as input to the next stage of the algorithm with every new vertex being a community by itself. This process of a modularity optimization phase followed by an aggregation phase continues until there is no further change in modularity, at which point the algorithm terminates.

When considering what community a vertex i should move to, one needs to evaluate Eq. 2 for every community $C(j)$ for which there exists some vertex $j \in N(i)$. The main challenge in doing this is to compute $e_{i \to C(j)}$ for every $j \in N[i]$. This is typically done by iterating through the neighbors of i and for each neighbor j one accumulates $w_{i,j}$ in a hash table using $C(j)$ as key. The value of each a_c can be computed at the start of a modularity optimization phase and then updated as vertices move between communities, while the values of m and k_i will remain unchanged between contractions of the graph and thus can be computed at the start of modularity optimization phase.

The aggregation phase follows a similar pattern as the modularity optimization. The main difference is that vertices within the same community c are now treated as one unit in that neighbors of any vertex in c are hashed into the same table. In this way one can compute the accumulated weight of all edges from one community to another.

3 Previous work

There has been several previous efforts to parallelize the Louvain method. The main common source of parallelism in all of these is to perform computations on multiple vertices concurrently. The computation of the modularity gain is the most time consuming part of the algorithm, thus this is also where there is the most to gain.

To parallelize the modularity optimization phase the vertices are partitioned into disjoint sets which are then processed concurrently and independently. The different approaches that have been tried can, broadly speaking, be classified depending on the number of vertices in each set.

In the coarse grained approach each set consists of multiple vertices that are typically processed using a sequential modularity optimization algorithm. Only when this algorithm has run to completion on each set, are the results from the different sets merged to compute the final input to the aggregation phase. The coarse grained approach is often used for implementations running on parallel computers with distributed memory. In the fine grained approach each vertex set consists of a single vertex. One iteration of the modularity optimization is now performed on each vertex in parallel. As this is done concurrently, the decision of which community a vertex should belong to is only based on the previous configuration. Once the community membership of each vertex has been computed, the information is made available to the other vertices and the process is repeated. This approach is most commonly used

for parallel computers with shared memory.

Wickramaarachchi et al. [26] presented a coarse grained algorithms based on MPI for communication. This algorithm gave speedups between 3 and 6 when using up to 128 processes. Also using a coarse grained algorithm, Zeng and Yu [27] reported speedups in the range of 2 to 4 when quadrupling the number of cores. The starting point for these measurements was set to 256, 512, and 1024 cores depending on the size of the graph.

Cheong et al. [4] presented a hybrid GPU algorithm that uses a coarse grain model across multiple GPUs, while the execution on each GPU follows a fine grain model where only the modularity optimization phase had been parallelized. They obtained speedups in the range of 1.8 to 5 for single GPU performance and between 3 and 17 when using 4 GPUs. We note that unlike other parallel algorithms this algorithm does not use hashing for computing the modularity gain, but instead sorts each neighbor list based on the community ID of each neighboring vertex. Recently Forster presented a GPU algorithm that uses a fine grained distribution [9]. This algorithm is an adaption of an OpenMP program similar to that in [16] (see below). Compared to the OpenMP program running on 8 cores, the paper reports speedups on four relatively small graphs on up to a factor of 12. There is no information about the quality of the solutions in terms of modularity.

Staudt and Meyerhenke [21] and Lu et al. [16] both presented fine grained implementations using OpenMP. The algorithm in [21] gave a maximal speedup of 9 when using 32 threads, while the speedup obtained in [16] ranged from 1.7 to 16 when using up to 32 threads. Xinyu et al. [20] presented a fine grained implementation running on a computer with distributed memory. This obtained speedups up to 64 when using up to 2K threads on medium sized problems, and a processing rate of up to 1.89 giga TEPS for large graphs when using 1K compute nodes each capable of running 32 threads. Here TEPS is the number of traversed edges per second in the first modularity phase. On a Blue Gene/Q supercomputer with 8192 nodes and 524,288 threads the algorithm had a maximum processing rate of 1.54 giga TEPS.

We note some further variations that have been suggested in the fine grained approach as to which vertices are chosen to participate in each round of the modularity optimization. In [20] only a predetermined fraction of the vertices that gives the highest modularity gain are moved in each iteration. This fraction decreases with each iteration within a phase. In [16] a graph coloring is used to divide the vertices into independent subsets. The algorithm then performs one iteration of the modularity optimization step on the vertices in each color class, with any change in community structure being committed before considering the vertices in the next color class. The motivation for this approach is to reduce contention for shared resources in a shared memory environment. This algorithm also uses further mechanisms to control which vertices participate in the modularity optimization. To prevent neighboring singleton vertices from simultaneously moving to each others communities, a vertex i which is a community by itself, can only move to another vertex j which is also a community by

itself, if $C(j) < C(i)$. In addition, if a vertex i has several possible communities it can move to, each one giving maximal modularity gain, then i will move to the community with the lowest index among these. Since the initial modularity optimization phases are the most costly ones, it can be advantageous if each of these terminate early so that the graph can be contracted. To achieve this, the algorithm initially uses a higher threshold for the net gain in modularity that is required to perform further iterations.

Both the fine grained implementations in [20] and [16] report that the obtained modularity of their algorithms is on par, or even better, than that of the sequential algorithm. However, for both of these algorithms the results depend critically on employing the aforementioned restrictions as to which vertices will participate in each iteration of the modularity optimization. For the coarse grained implementations the multi-GPU implementation in [4] reports a loss of up to 9% in modularity, while [27] and [26] reports results on par with the sequential algorithm.

4 The GPU algorithm

In the following we describe our fine grained GPU implementation based on the Louvain method. The main difference compared to previous parallel implementations is that we also parallelize the hashing of individual edges both in the modularity optimization and also in the aggregation phase. In order to obtain an even load balance we let the number of threads assigned to a vertex scale with its degree. In the modularity optimization phase this is achieved by partitioning the vertices into subsets depending on their degrees. These sets are then processed in turn by computing and updating the destination community of each vertex in parallel. For each set we use a different number of threads per vertex. We expand further on memory usage and thread assignment following the presentation of the algorithm.

All computations are run on the GPU and only when a global synchronization is needed is the control returned to the host before calling the next kernel. At the start of the algorithm the graph $G = (V, E)$ is stored in the global memory on the GPU using neighbor lists.

We denote the weight of an edge (i, j) by $w[i, j]$ and use a global table C such that for each vertex i the value $C[i]$ gives the ID of the current community that i belongs to. At the start of each modularity optimization phase $C[i] = i$. Thus every vertex is a community by itself.

The algorithm is lock free, and only uses atomic operations and compare-and-swap (CAS) operations when access to memory location must be sequentialized. In addition, we use optimized routines from the Nvidia's Thrust library for collective operations such as computing prefix sums and for partitioning the vertices according to some criteria.

Although not explicitly outlined in the pseudo code, we use some of the ideas in [16] to control the movements of vertices in the modularity optimization phase. In

particular we only allow a vertex i that is a community by itself to move to another vertex j that is also a community by itself if $C[j] < C[i]$. A vertex always moves to the community with lowest index when there is more than one move that gives a maximal modularity gain. We also employ the idea of using a higher threshold for the modularity gain in the initial rounds.

The main algorithm follows the same outline as the sequential one with a loop that iterates over a modularity optimization phase followed by a graph aggregation phase. This is repeated until the increase in modularity gain from one iteration to the next is below some predefined threshold.

The modularity optimization phase is shown in Algorithm 1. Initially the algorithm computes the values of m and in parallel for each $i \in V$ the value of k_i (line 2). Note that initially $a_C(i) = k_i$ as each vertex starts out as a community by itself. These values are needed in subsequent evaluations of Eq. (2). The outermost loop then iterates over the vertices until the accumulated change in modularity during the iteration falls below a given threshold (lines 3 through 12). For each iteration the vertices are divided into buckets depending on their degrees. The variable $numDBuckets$ holds the number of buckets, while the kth bucket contains vertices of degree ranging from $bucDSize[k-1]$ up to $bucDSize[k]$. To extract the vertices within a certain degree range we use the Thrust method $partition()$. This reorders the elements of an array so that those elements satisfying the given boolean condition can easily be extracted to an array $vSet$ (line 5). Here V contains the vertices, while the function $deg(i)$ gives the degree of vertex i. The selected vertices are then passed in parallel to the $computeMove()$ method that determines to which community each one should belong (line 7). These values are returned in the $newComm$ array. Once the computations for a bucket are completed the community IDs of the associated vertices are updated accordingly (line 9) before a_c is recalculated for each community (line 11).

Algorithm 1 Modularity Optimization

```
1:  procedure MODOPT
2:    Compute m and for each i ∈ V in parallel: k_i;
3:    repeat
4:      for k = 1 to numDBuckets do
5:        vSet = partition(V, bucDSize[k − 1] < deg(i) ≤ bucDSize[k]);
6:        for each i ∈ vSet in parallel do
7:          computeMove(i);
8:        for each i ∈ vSet in parallel do
9:          C[i] = newComm[i];
10:       for each c ∈ C in parallel do
11:         Compute a_c;
12:   until modularity gain <threshold
```

The $computeMove()$ method is given in Algorithm 2. This method takes one

vertex i as argument and computes the community that gives the highest increase in modularity according to Eq. (2) if i was to join it. The computation is carried out by first storing a running sum of the weights from i to each neighboring community c (i.e. $e_{i\to c}$) in a hash table $hashWeight$. These values are then used in the computation of Eq. (2) to select which community i should belong to. In addition to the weight, the algorithm also stores the associated ID of each incident community in a separate hash table $hashComm$ using the same position as is used for indexing $hashWeight$. We use open addressing and double hashing [5] for computing each new index in the hash tables (line 5) . The search of the hash table runs until either the ID of the sought after community is discovered (line 6) or until an empty slot is found (line 8). The size of the hash tables for i is drawn from a list of precomputed prime numbers as the smallest value larger than 1.5 times the degree of i.

As the neighbors of i are considered in parallel, care must be taken when updating the hash tables. If the community of a neighbor of i has already been entered into the hash table then it is sufficient to atomically update the running weight (line 7). This is done using an $atomicAdd()$ operation that takes the position to be updated and the value to be added as arguments. However, if an empty slot is found during the search of the hash table then the community has not been entered into the hash table previously. One must then claim the current position in the $hashComm$ table for this community. This is done by writing the ID of the community in this position of $hashComm$. To avoid race conditions for empty slots, we do this using a CAS operation that tries to replace a $null$ value in $hashComm$ with the new community ID (line 9). Only if this operation is successful will the thread add the edge weight to the corresponding position in the $hashWeight$ table (line 10), otherwise some other thread claimed this position first. If another thread did this and entered the sought after community ID, then the current thread can still add its weight to the hash table (line 12). Otherwise it will continue searching the $hashComm$ table from the current position.

While processing the neighbors of vertex i, all terms in Eq. (2) are known in advance except for $\frac{e_{i\to C(j)}}{m}$ and $-\frac{e_{i\to C(i)\setminus\{i\}}}{m}$. But since $-\frac{e_{i\to C(i)\setminus\{i\}}}{m}$ is identical in each evaluation of Eq. (2) this does not influence which community i will join. Thus for each update of the hash table with an edge (i, j), a thread can locally keep track of the best community it has encountered so far by evaluating the sum of the second term of Eq. (2) and the current value of $\frac{e_{i\to C(j)}}{m}$. Once all neighbors of i has been processed, the algorithm performs a parallel reduction of these values to determine the best community for i (line 14). If this gives a positive modularity gain then the new community ID is stored in the $newComm$ table, otherwise this value is set to the existing community of i.

The aggregation phase is shown in Algorithm 3. The actions in this algorithm can be subdivided into four different tasks: *(i)* Determine the size of each community. *(ii)* Compute a mapping from existing non-empty communities to a consecutive numbering of the new vertices. *(iii)* Set up a data structure for the new graph. *(iv)* Determine the new edges and their weights.

Algorithm 2 Compute next move

```
 1: procedure COMPUTEMOVE(i)
 2:   for each j ∈ N[i] in parallel do
 3:     it = 0;
 4:     repeat
 5:       curPos = hash(C[j], it++);
 6:       if hashComm[curPos] == C[j] then
 7:         atomicAdd(hashWeight[curPos], w[i, j]);
 8:       else if hashComm[curPos] == null then
 9:         if CAS(hashComm[curPos], null, C[i]) then
10:           atomicAdd(hashWeight[curPos], w[i, j]));
11:         else if hashComm[curPos] == C[j] then
12:           atomicAdd(hashWeight[curPos], w[i, j]);
13:     until hashComm[curPos] == C[j]
14:   pos = argmax_curPos{Eq. (2)
        using hashWeight[curPos] as input};
15:   if Eq. (2) using hashWeight[pos] > 0 then
16:     newComm[i] = hashComm[pos];
17:   else
18:     newComm[i] = C[i];
```

In the following we describe how we solve each of these tasks. *(i)* To compute the size of each community we start by initializing a counter $comSize$ to zero for each community (line 2). The algorithm then iterates through the vertices in parallel and for each vertex i it atomically increases the size of the community that i currently belongs to by one (line 5). The number of vertices in the new graph will then be equal to the number of non-empty communities in the original graph. *(ii)* The values in $comSize$ can then be used to compute a consecutive numbering, starting from zero, of the vertices in the contracted graph. This is done by first setting a variable $newID$ for each community to either zero or one, depending on if the community is empty or not (line 8). An ensuing parallel prefix sum on this value gives a consecutive mapping between each non-empty community and the corresponding new vertex (line 12). *(iii)* In order to set up storage for the edges of the new graph one needs a bound on the size of each edge list. It is possible to calculate this number exactly, but this would have required additional time and memory. We therefore chose to use the sum of the individual vertex degrees of the vertices in each community as an upper bound. This value is computed in parallel and stored in the table $comDegree$ (line 6). We copy this table value to $edgePos$ on which we perform a parallel prefix sum to get a pointer to where the edges of each vertex should be stored in a new edge list (line 14). *(iv)* Before computing the edge set of the new graph it is convenient to order the vertices in the original graph according to which community they currently belong

to. In this way the computation of the neighborhood of each new vertex can more easily be assigned to one thread block or one warp. This is explained further in Section 4.1. By performing a parallel prefix sum on the size values of each community in a separate table $vertexStart$ we get a pointer to where the vertices of each community should be stored (line 16). We then process all vertices in parallel (line 17) and for each vertex we fetch and increase the pointer of its community using an atomic add before storing the vertex in the table com. Following this we can start the computation of the edge set of the contracted graph. Similarly to what was done in the modularity optimization, we partition the communities based on the expected amount of work to process each one. To estimate the work we use the already computed upper bound on the size of the neighborhood of each community stored in $comDegree$. The variable $numCBuckets$ holds the number of different buckets, while the table $bucCSize$ gives the bounds specifying in which bucket each community should be stored (line 20). The buckets are processed in turn, and for each bucket the communities in it are processed in parallel using the $mergeCommunity()$ method (line 23).

Once the edge lists have been computed for all communities in a bucket, these lists are compressed to fill consecutive slots in global memory. This entails a parallel prefix sum on the actual size of each edge list before the edges are moved in place. This code is not shown in Algorithm 3.

The $mergeCommunity$ method is similar to $computeMove$ and therefore not shown separately. The method starts by hashing vertices into a table similarly as is done in the for loop in line 2 of $computMove$. The main difference is that $mergeCommunity$ hashes the neighbors of *all* vertices in a community c and not just those of one vertex. When this is done, the entries in the hash table contains the set of edges and associated weights that will be incident on the new vertex replacing c. The method then moves these edges from the hash table to the compressed edge list that will be used for the new graph. While doing so it replaces the ID of each neighboring community nc with the new ID of the vertex that will replace nc (as stored in $newID$ in Algorithm 3). To do the movement of edges efficiently, each threads marks and counts the edges it processed that gave rise to a new entry in the hash table. By performing a prefix sum across all threads on the number of such entries, each thread can look up where in the compressed edge lists it should store the edges it initialized in the hash table. With this knowledge, the movement of edges from the hash table to the global edge lists can be performed in parallel without updating any shared variables.

4.1 Thread and memory allocation

In the following we describe how we allocate tasks to threads and also how memory management is carried out. Due to the relatively complex structure of a GPU these issues are important when it comes to achieving high performance.

The input graph is initially transferred to the device memory. All processing is then carried out on the device. Control is only returned to the host for calling new kernels

Algorithm 3 Aggregation phase

```
1:  procedure CONTRACT
2:      comSize[1 : n] = 0;
3:      commDegree[1 : n] = 0;
4:      for i ∈ V in parallel do
5:          atomicAdd(comSize[c[i]], 1);
6:          atomicAdd(comDegree[c[i]], degree[i]);
7:      for each community c in parallel do
8:          if comSize[c] == 0 then
9:              newID[c] = 0;
10:         else
11:             newID[c] = 1;
12:     prefixSum(newID);
13:     edgePos = comDegree;
14:     prefixSum(edgePos);
15:     vertexStart = comSize;
16:     prefixSum(vertexStart);
17:     for i ∈ V in parallel do
18:         res = atomicAdd(vertexStart[c[i]], 1);
19:         com[res] = i;
20:     for k = 1 to numCBuckets do
21:         comSet = partition(V, bucCSize[k−1] < comDegree[i] ≤ bucCSize[k]);
22:         for each c ∈ comSet in parallel do
23:             mergeCommunity(c);
```

and Thrust library routines. Graphs are stored using compressed neighbor lists. Thus the structure of a graph $G(V, E)$ is represented using two arrays $vertices$ and $edges$ of size $|V| + 1$ and $2|E|$ respectively. The neighbors of vertex i are stored in positions $vertices[i]$ up to position $vertices[i + 1]$. In addition there is an array $weights$ also of length $2|E|$ containing the weight of the corresponding edge in $edges$. These arrays are always stored in global memory due to their size. During the execution of the algorithm the threads use shared memory as far as possible due to speed, but for some computations where there is a large memory requirement, the global memory is used. Also, reading of data from memory is as far as possible done in a coalesced fashion. This is particularly true when reading the neighbor lists of vertices. Due to lack of memory the program only outputs the final modularity, and does not save intermediate clustering information.

The algorithm is centered around the processing of a vertex or a set of vertices. This has implications for how we assign tasks to threads. Since it is not possible to synchronize a subset of thread blocks or a subset of warps within a thread block, we either use a full thread block or a fraction of one physical warp as our unit when assigning tasks to threads. When assigning a task to a fraction of a warp we divide each physical warp into equal sized thread groups containing 2^k threads each, where $k \in [2, 3, 4, 5]$. Throughout the computation we use four physical warps of 32 threads each per thread block.

In the modularity optimization phase the main task is to compute to which community each vertex should belong using the $computeMove$ method, while in the aggregation phase the main task is to compute the neighborhood of each community and the structure of the ensuing new graph using the $mergeCommunity$ method. Thus the execution of each of these routines is never subdivided across multiple thread blocks. Instead each such call is assigned either to a thread block or to a group of threads belonging to the same warp. As shown in algorithms 1 and 3 this is done in batches to allow the number of threads allocated to each task to scale with the expected amount of work.

In $computeMove$ we divide the vertices into seven different groups that are processed one after another. The first six groups contain all vertices with degrees in the range $[1, 4]$, $[5, 8]$, $[9, 16]$, $[17, 32]$, $[33, 84]$, and $[85, 319]$ respectively, while the last group contains all vertices with degree higher than 319. For groups $k = 1$ through 4 we assign 2^{k+1} threads to each vertex. All threads assigned to the same vertex will then belong to the same warp. Together with each vertex we also allocate sufficient space for the hash tables in shared memory. In this way each thread is assigned at most one edge which it has to store in the hash table according to the community membership of the neighboring vertex. A vertex in group five is allocated to all the threads of one warp and with hash table in shared memory, but now each thread has the responsibility for hashing between one and three edges. These are distributed to the threads in an interleaved fashion. For each vertex in group six and seven we assign all the threads (128) of one thread block. The difference between the groups is that in group six the

hash table is stored in shared memory while for vertices in group seven the hash table is stored in global memory. Another difference is that for group seven we might need to assign multiple vertices to each thread block since the size of the global memory is fixed. These will then be processed sequentially, while reusing the same memory location. To ensure a good load balance between the thread blocks, the vertices in group seven are initially sorted by degree before the vertices are assigned to thread blocks in an interleaved fashion.

The allocation of threads to tasks in the aggregation phase is carried out in a similar way as in the modularity optimization phase. But as communities tend to have larger neighborhoods than their individual vertex members, we only subdivide the communities into three different groups depending on if the sum of the degrees of its members are in the range $[1, 127]$, $[128, 479]$, or larger than 479. Each community in the first group is handled by one warp, while communities in the second group are assigned to one thread block, both using shared memory to store the hash table. Finally, the last group is also assigned to one thread block but now storing the hash table in global memory. Similarly as in the modularity optimization, we assign multiple threads to each community. Note that each thread is only assigned to one community in the two first groups, while each thread might participate in processing several communities in the final group. When processing the vertices of one community, all threads participate in the processing of each vertex. Thus vertices within each community are processed sequentially.

5 Experiments

In the following we describe experiments performed to evaluate our implementation based on the Louvain method as described in the previous section. The algorithm was implemented using CUDA C++ and was run on a Tesla K40m GPU with 12 GB of memory, 2880 cores running at 745 MHz, and with CUDA compute capability 3.5. For all sequential experiments we used the original code from [1] running on a 3.30 GHz Intel Xeon i5-6600 processor with 16 Gbytes of memory. For the OpenMP comparisons we used a computer equipped with two Intel Xeon Processor E5-2680 processors. Each processor has 10 cores and can run 20 threads using hyper-threading for a total of 40 threads.

For the experiments we picked 44 graphs from the Florida sparse matrix collection [7] chosen as a representative set among graphs for which the sequential Louvain method required at least 10 seconds to execute, and which gave a relative high modularity. In addition we included 6 graphs from the Snap collection [13] and 5 graphs from the Koblenz collection [12]. Finally, the graph coPapersDBLP was included as it is frequently used in other studies. The first four columns of Table 1 lists the names, number of vertices, number of edges, and sequential running time in seconds respectively of the chosen graphs. The graphs are ordered by decreasing average vertex

degree.

Our first set of experiments was performed to test the effect of changing the threshold value used in Algorithm 1 for determining when an iteration of the modularity optimization should end. As explained in Section 4 we use a larger threshold value t_{bin} when the graph size is above a predetermined limit and a smaller one t_{final} when it is below this limit. Similar to what was done in [16] this limit was set to $100,000$ vertices. We ran the algorithm with all combinations of threshold values (t_{bin}, t_{final}) on the form $(10^k, 10^l)$ where k varied from -1 to -4 and l from -3 to -7.

Figure 1 shows the average modularity over all graphs for each pair of values for th_{final} and th_{bin} compared to the modularity given by the sequential algorithm. As can be seen from the figure the relative modularity of the GPU algorithm decreases when the thresholds increases. Still, the average modularity of the GPU algorithm is never more than 2% lower than that given by the sequential algorithm. In Figure 2 we show the relative speedup compared to the best speedup that was obtained when the threshold values were varied. These numbers were obtained by first computing the best speedup for each graph across all possible threshold configurations. For each threshold configuration we then computed the relative distance from the best speedup for each graph. Finally, we plot the average of these numbers for each threshold configuration.

It is clear from Figure 2 that the speedup is critically dependent on the value of th_{bin}, with higher values giving better speedup. However, this must be compared to the corresponding decrease in modularity. Based on these observations we chose to use a value of 10^{-6} for th_{final} and 10^{-2} for th_{bin}. With these choices we still have an average modularity of over 99% compared to the sequential algorithm and an average speedup of about 63% compared to the best one. The fifth column of Table 1 shows the running time of the GPU algorithm using these parameters. The speedup of the GPU algorithm relative to the sequential one is plotted in Figure 3. The speedup ranges from approximately 2.7 up to 312 with an average of 41.7. However, these results depend on using a higher value for th_{bin} in the GPU algorithm. This value can also be used in the sequential algorithm. In Figure 4 we show the speedup when the sequential algorithm has been modified in this way.

The effect of doing this is that the running time of the sequential algorithm decreases significantly giving an average speedup of 7.3 compared to the original one. The average modularity of the final solution only drops by a factor of 0.13% compared to that of the original algorithm. The obtained speedup of the GPU algorithm is now in the range from just over 1 up to 27 with an average of 6.7.

Next, we consider how the time is spent in the GPU algorithm. In figures 5 and 6 we show the breakdown of the running time over the different stages for the road_usa and nlpkkt200 graphs respectively. For each stage the time is further divided into the time spent in the modularity optimization phase and in the aggregation phase.

Figure 5 gives the typical behaviour we experienced, with the first stage being the most time consuming followed by a tail of less expensive stages. On a few graphs this tail could go on up to a few hundred stages. On average the algorithm spends

Graph	#V	#E	Time sequential	Time GPU
out.actor-collaboration	382,220	33,115,812	6.81	2.53
hollywood-2009	1,139,905	56,375,711	17.49	4.69
audikw_1	943,695	38,354,076	42.42	1.90
dielFilterV3real	1,102,824	44,101,598	21.99	1.54
F1	343,791	13,246,661	9.81	0.75
com-orkut	3,072,627	117,185,083	197.98	16.83
Flan_1565	1,564,794	57,920,625	115.55	3.39
inline_1	503,712	18,156,315	9.07	1.29
bone010	986,703	35,339,811	58.14	0.94
boneS10	914,898	27,276,762	24.48	0.97
Long_Coup_dt6	1,470,152	42,809,420	41.51	1.40
Cube_Coup_dt0	2,164,760	62,520,692	68.84	2.70
Cube_Coup_dt6	2,164,760	62,520,692	67.35	2.69
coPapersDBLP	540,486	15,245,729	3.33	0.73
Serena	1,391,349	31,570,176	38.15	0.76
Emilia_923	923,136	20,041,035	22.39	0.57
Si87H76	240,369	5,210,631	2.60	0.77
Geo_1438	1,437,960	30,859,365	40.94	1.09
dielFilterV2real	1,157,456	23,690,748	39.60	0.62
Hook_1498	1,498,023	29,709,711	36.49	0.71
soc-pokec-relationships	1,632,803	30,622,562	36.61	4.52
gsm_106857	589,446	10,584,739	8.48	0.34
uk-2002	18,520,486	292,243,663	385.34	8.21
soc-LiveJournal1	4,847,571	68,475,391	117.61	8.15
nlpkkt200	16,240,000	215,992,816	327.42	26.11
nlpkkt160	8,345,600	110,586,256	168.56	11.54
nlpkkt120	3,542,400	46,651,696	78.08	3.97
bone010_M	986,703	11,451,036	63.50	0.52
cnr-2000	325,557	3,128,710	2.27	0.26
boneS10_M	914,898	8,787,288	27.42	0.52
out.flickr-links	1,715,256	15,551,249	9.25	2.64
channel-500x100x100-b050	4,802,000	42,681,372	934.17	6.67
com-lj	4,036,538	34,681,189	78.09	5.25
packing-500x100x100-b050	2,145,852	17,488,243	360.42	1.19
rgg_n_2_24_s0	16,777,216	132,557,200	132.87	4.95
offshore	259,789	1,991,442	13.14	0.15
rgg_n_2_23_s0	8,388,608	63,501,393	60.44	2.42
rgg_n_2_22_s0	4,194,304	30,359,198	30.48	1.20
StocF-1465	1,465,137	9,770,126	177.86	0.57
out.flixster	2,523,387	7,918,801	16.90	2.11
delaunay_n24	16,777,216	50,331,601	95.60	1.60
out.youtube-u-growth	3,223,585	9,375,369	18.46	2.62
com-youtube	1,157,828	2,987,624	4.58	1.00
com-dblp	425,957	1,049,866	2.40	0.22
com-amazon	548,552	925,872	2.53	0.26
hugetrace-00020	16,002,413	23,998,813	101.84	1.43
hugebubbles-00020	21,198,119	31,790,179	126.79	2.01
hugebubbles-00010	19,458,087	29,179,764	116.90	1.87
hugebubbles-00000	18,318,143	27,470,081	115.88	1.60
road_usa	23,947,347	28,854,312	132.38	1.93
germany_osm	11,548,845	12,369,181	42.48	1.64
asia_osm	11,950,757	12,711,603	42.86	7.22
europe_osm	50,912,018	54,054,660	197.07	22.21
italy_osm	6,686,493	7,013,978	24.33	4.82
out.livejournal-links	5,204,175	2,516,088	25.33	1.39

Table 1: Graphs used for the experiments

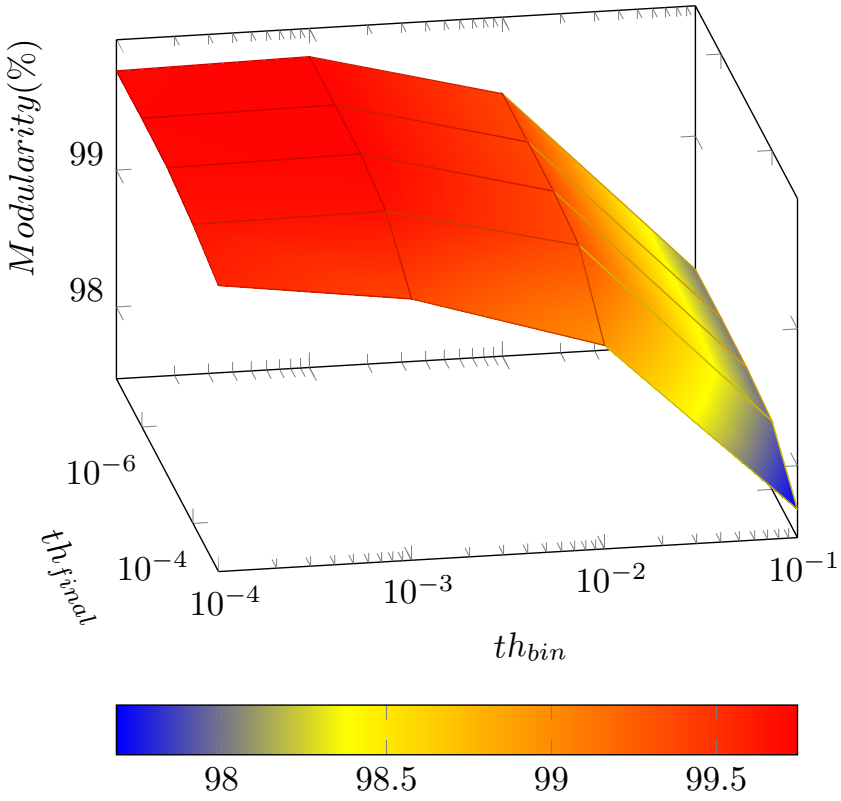

Figure 1: Relative modularity for different threshold values

about 70% of the total time in the optimization phase and 30% in the aggregation phase. Thus the algorithm is spending most of the time in the optimization phase even though this part has been more carefully tuned to utilize GPU resources compared to the aggregation phase. It follows that further fine tuning of the load balance in the aggregation phase would most likely only give a limited effect.

The behaviour shown in Figure 6 occurred only in the channel-500 graph and the three nlpkkt graphs. For all of these graphs we observed that in the first few stages the graph size did not decrease significantly. We then get a time consuming modularity optimization phase where the largest remaining community is up to two orders of magnitude larger than what was seen in the previous stages. Following this stage the size of the graph decreases considerably. We note that this effect happens while we are still using the t_{bin} threshold value. We suspect that this effect might occur on graphs that lack a natural initial community structure.

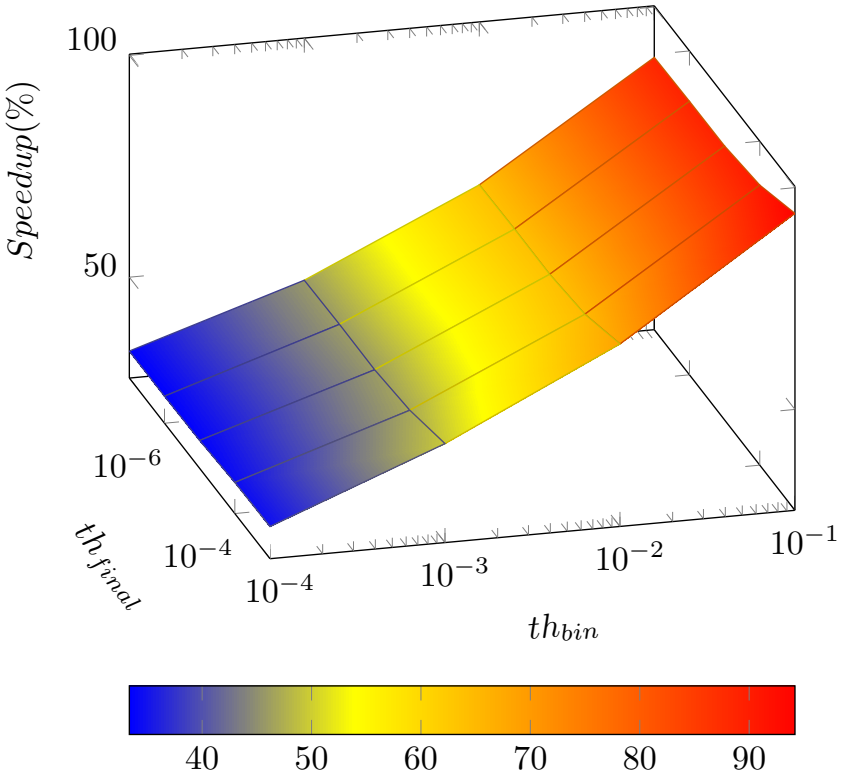

Figure 2: Relative speedup for different threshold values

In shared memory implementations such as those in [16] and [21] a thread is responsible for determining the next community for several vertices in the modularity optimization phase. Once a new community has been computed for a vertex, it is immediately moved to it. In this way each thread has knowledge about the progress of other threads through the shared memory. At the other end of the scale, a pure fine grained implementation would compute the next community of each vertex only based on the previous configuration and then move all vertices simultaneously. Our algorithm operates somewhere in between these two models by updating the global community information following the processing of the vertices in each bin. A natural question is then how the algorithm is affected by this strategy. To test this we ran experiments where we only updated the community information of each vertex at the end of each iteration in the optimization phase. We label this as the "relaxed" approach. The results showed that the average difference in modularity was less than 0.13% between the two strategies. However, the running time would in some cases increase by as much as a

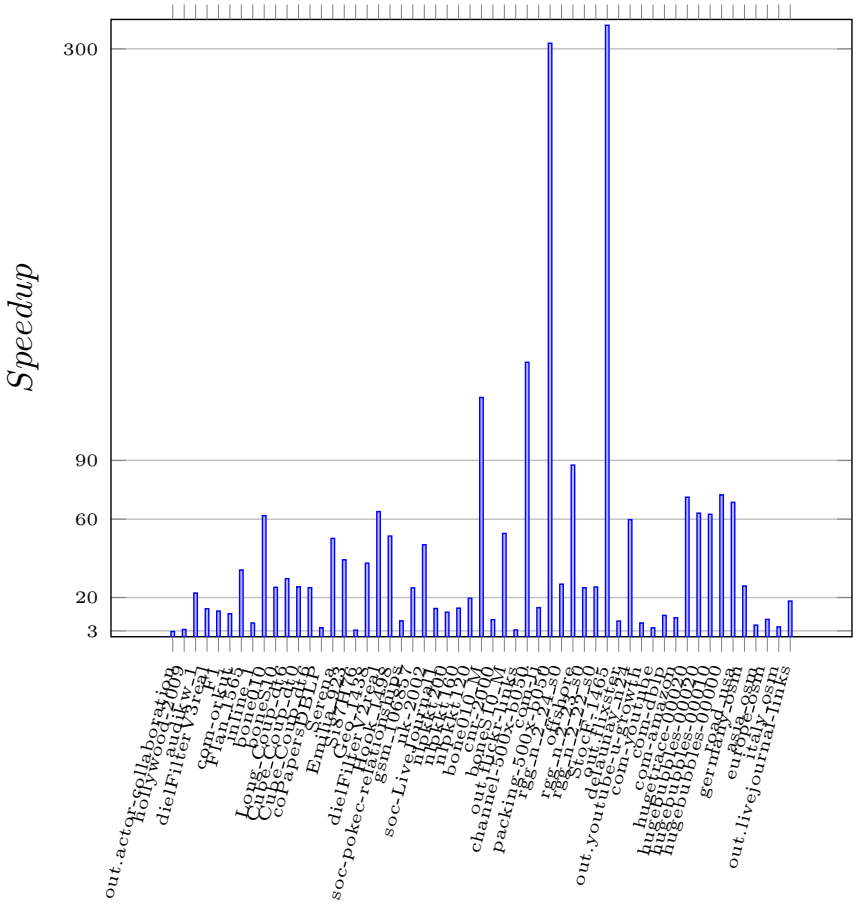

Figure 3: Speedup of the GPU algorithm compared to the sequential algorithm

factor of ten when using the relaxed strategy. This was typically due to the optimization phase immediately following the switch from t_{bin} to t_{final}. One other observed difference was that the number of phases was in some instances significantly smaller with the relaxed strategy, although this did not result in a clear trend in how the running time was affected. In several cases the reduced number of iterations would be offset by the algorithm spending more time in each iteration.

To see how our algorithm compares to other parallel implementations we have compared our results with those of the parallel Louvain method (PLM) by Staudt and Meyerhenke [21] using 32 threads. Our test sets contains four common graphs, coPaperDBLP, soc-LiveJournal1, europe_osm, and uk-2002. On these graphs the average

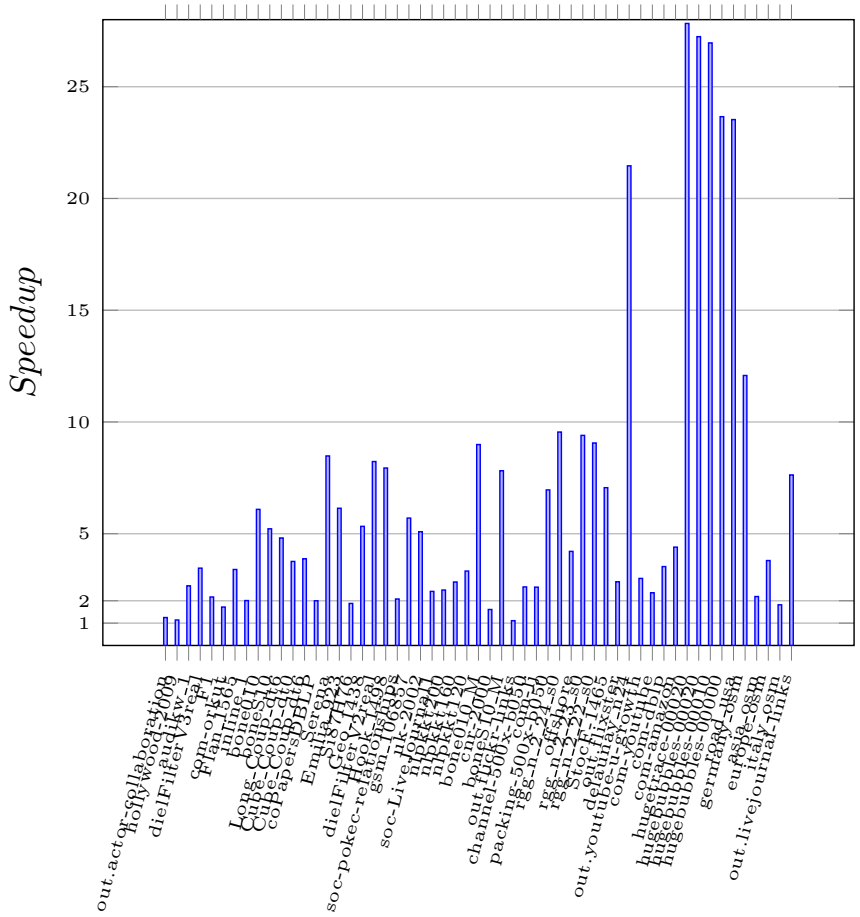

Figure 4: Speedup compared to the adaptive sequential algorithm

modularity of the methods differed by less than 0.2%. The running time of coPaperDBLP in [21] was less than one second, while the other graphs all required more than 10 seconds each. On the three largest graphs our algorithm gave speedups ranging from a factor of 1.3 to 4.6 with an average of 2.7 compared to [21].

We have also compared our results with the OpenMP code from [16]. Out of our test set in Table 1 we were able to run 30 graphs on the computer equipped with two Intel Xeon E5-2680 processors using 20 threads. For these tests we used NUMA aware thread allocation and ”scatter mode” for thread pinning. Figure 7 gives the relative performance of our GPU implementation compared to the OpenMP code on these graphs.

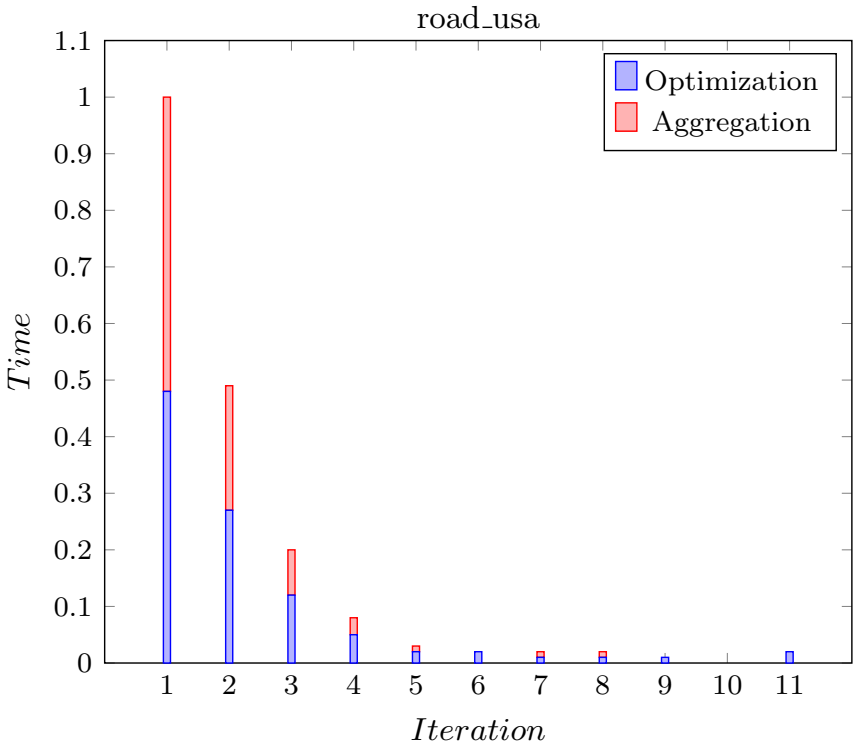

Figure 5: Time spent on the road_usa graph

Our GPU implementation gave a speedup ranging from 1.1 to 27.0 with an average of 6.1. Note that both algorithms are using the same threshold values $(10^{-2}, 10^{-6})$ in the modularity optimization phase.

To investigate what is causing this speedup, we have measured how much time both algorithms are spending on the initial processing of each vertex in the first iteration of the modularity optimization. In this part both algorithms are hashing exactly $2|E|$ edges. The results show that the GPU code is on average 9 times faster than the code from [16]. There are several possible explanations for this. The OpenMP code uses locks in the preprocessing and the contraction phase, while the GPU code uses CAS and atomic operations. Moreover, the GPU code is doing most of the hashing in shared memory which is as fast as L1 cache. We also believe that the code from [16] could execute faster if it employed better storage strategies for the hashing.

The GPU code has been profiled to see how it is utilizing the hardware resources and how much parallelism is available. On UK-2002, on average 62.5% of the threads

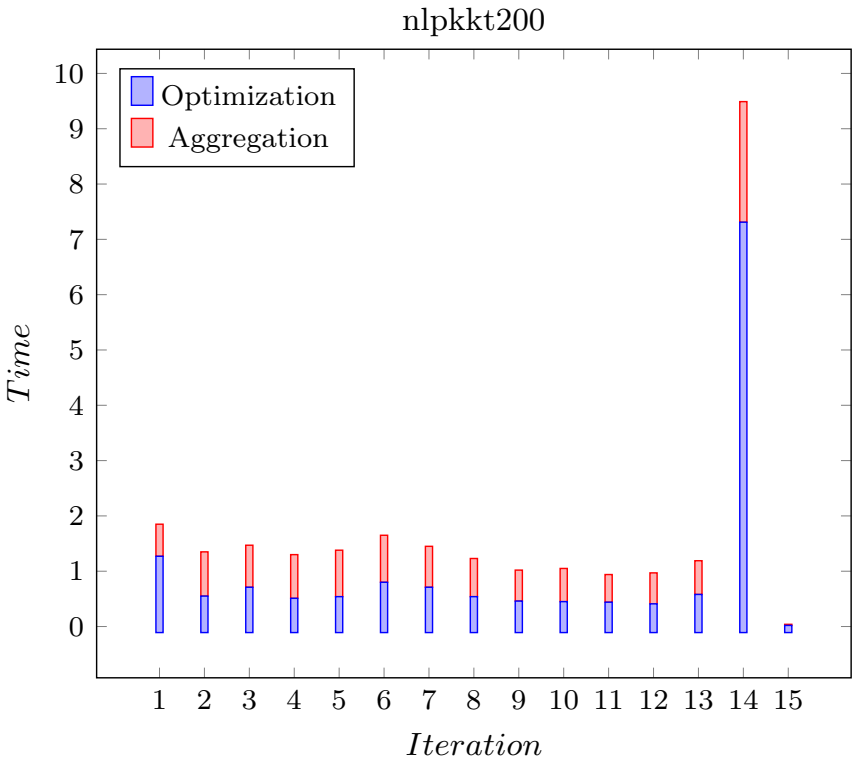

Figure 6: Time spent on the nlpkkt200 graph

in a warp are active whenever the warp is selected for execution. The four schedulers of each streaming multiprocessor has on average 3.4 eligible warps per cycle to choose from for execution. Thus, despite the divergence introduced by varying vertex degrees and memory latency, this indicate that we achieve sufficient parallelism to keep the device occupied.

Finally, we note that the implementation in [20] reported a maximum processing rate of 1.54 giga TEPS in the first modularity optimization phase when using a Blue Gene/Q supercomputer with 8192 nodes and 524,288 threads. Our largest TEPS rate was 0.225 giga TEPS obtained for the channel-500 graph. Thus the implementation on the Blue Gene/Q gave less than a factor of 7 higher TEPS rate than our one using a single GPU.

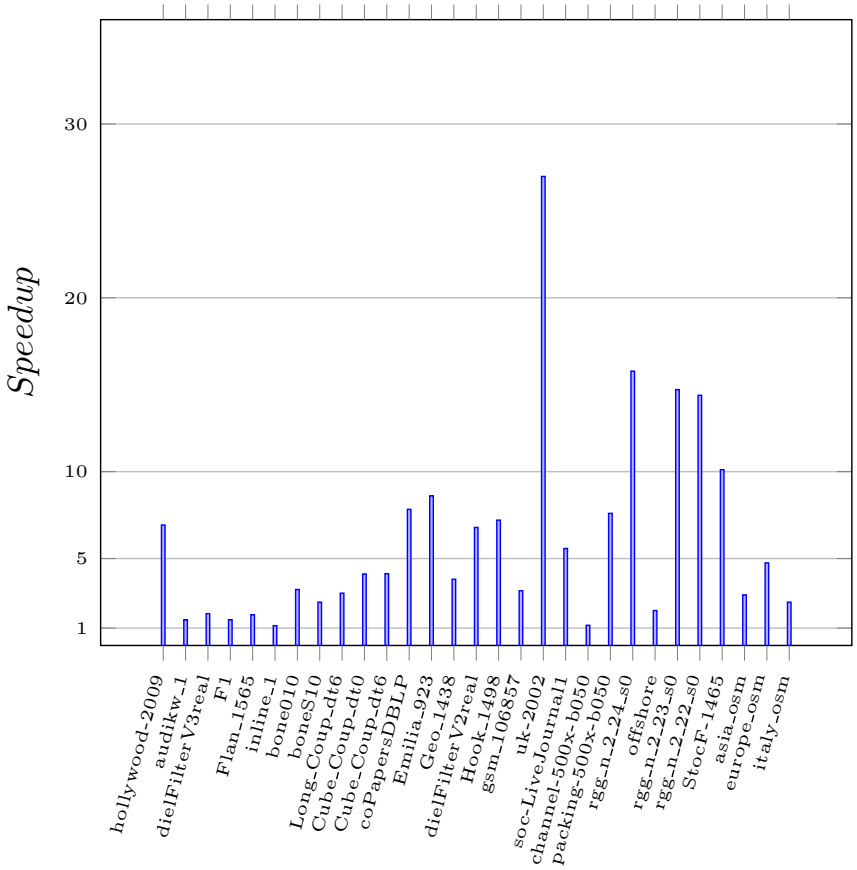

Figure 7: Speedup of the GPU implementation compared to [16]

6 Conclusion

We have developed and implemented the first truly scalable GPU version based on the Louvain method. This is also the first implementation that parallelizes and load balances the access to individual edges. In this way our implementation can efficiently handle nodes of highly varying degrees. Through a number of experiments we have shown that it obtains solutions with modularity on par with the sequential algorithm. In terms of performance it consistently outperforms other shared memory implementations and also compares favourably to a parallel Louvain method running on a supercomputer, especially when comparing the cost of the machines.

The algorithm achieved an even load balance by scaling the number of threads as-

signed to each vertex depending on its degree. We note that similar techniques have been used to load balance GPU algorithms for graph coloring [8] and for basic sparse linear algebra routines [14]. We believe that employing such techniques can have a broad impact on making GPUs more relevant for sparse graph and matrix computations, including other community detection algorithms.

We note that the size of the current GPU memory can restrict the problems that can be solved. Although the memory size of GPUs is expected to increase over time, this could be mitigated by the use of unified virtual addressing (UVA) to acquire memory that can be shared between multiple processing units. However, accessing such memory is expected to be slower than on-card memory. We believe that our algorithm can also be used as a building block in a distributed memory implementation of the Louvain method using multi-GPUs. This type of hardware is fairly common in large scale computers. Currently more than 58% of the 500 most powerful computers in the world have some kind of GPU co-processors from NVIDIA [22].

Using adaptive threshold values in the modularity optimization phase had a significant effect on the running time of both the sequential and our parallel algorithm. This idea could have been expanded further to include even more threshold values for varying sizes of graphs. It would also have been possible to fine tune the implementation of the aggregation phase to further speed up the processing. However, as this is currently not the most time consuming part, the effect of doing this would most likely have been limited.

Finally, we note that coarse grained approaches seem to consistently produce solutions of high modularity even when using an initial random vertex partitioning. This could be an indication that the tested graphs do not have a clearly defined community structure or that the algorithm fails to identify communities smaller than a network dependent parameter [11].

References

[1] Vincent D. Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of community hierarchies in large networks. *CoRR*, abs/0803.0476, 2008.

[2] Ulrik Brandes, Daniel Delling, Marco Gaertler, Robert Görke, Martin Hoefer, Zoran Nikoloski, and Dorothea Wagner. On modularity clustering. *IEEE Trans. Knowl. Data Eng.*, 20(2):172–188, 2008.

[3] Ümit V. Çatalyürek, Cevdet Aykanat, and Bora Uçar. On two-dimensional sparse matrix partitioning: Models, methods, and a recipe. *SIAM J. Scientific Computing*, 32(2):656–683, 2010.

[4] Chun Yew Cheong, Huynh Phung Huynh, David Lo, and Rick Siow Mong Goh. Hierarchical parallel algorithm for modularity-based community detection using

GPUs. In *Euro-Par 2013 Parallel Processing - 19th International Conference, Aachen, Germany, August 26-30, 2013. Proceedings*, volume 8097 of *Lecture Notes in Computer Science*, pages 775–787. Springer, 2013.

[5] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. The MIT Press, third edition, 2009.

[6] D. Doyle D. Greene and P. Cunningham. Tracking the evolution of communities in dynamic social networks. In *Proceedings of the International Conference on Advances in Social Networks Analysis and Mining, (ASONAM)*, pages 176–183, 2010.

[7] T. A. Davis and Y. Hu. The University of Florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1–25, December 2011.

[8] Mehmet Deveci, Erik G. Boman, Karen D. Devine, and Sivasankaran Rajamanickam. Parallel graph coloring for manycore architectures. In *2016 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2016, Chicago, IL, USA, May 23-27, 2016*, pages 892–901, 2016.

[9] Richard Forster. Louvain community detection with parallel heuristics on gpus. In *Proceedings of the 20th Jubilee IEEEInternational Conference on Intelligent Engineering Systems*, pages 227–232, 2016.

[10] S. Fortunato. Community detection in graphs. *Physics Reports*, 486(3-5):75–174, 2010.

[11] Santo Fortunato and Marc Barthelemy. Resolution limit in community detection. *PNAS*, 104(1):36–41, 2007.

[12] konect network dataset - KONECT. http://konect.uni-koblenz.de, 2016.

[13] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. http://snap.stanford.edu/data, jun 2014.

[14] Weifeng Liu and Brian Vinter. A framework for general sparse matrix-matrix multiplication on gpus and heterogeneous processors. *Journal of Parallel and Distributed Computing*, 85:47–61, 2015.

[15] Yabing Liu, Krishna P. Gummadi, Balachander Krishnamurthy, and Alan Mislove. Analyzing facebook privacy settings: User expectations vs. reality. In *Proceedings of the 2011 ACM SIGCOMM Conference on Internet Measurement Conference*, IMC '11, pages 61–70, New York, NY, USA, 2011. ACM.

[16] Hao Lu, Mahantesh Halappanavar, and Ananth Kalyanaraman. Parallel heuristics for scalable community detection. *Parallel Computing*, 47:19–37, 2015.

[17] D. Meunier, R. Lambiotte, A. Fornito, K. D. Ersche, and E. T. Bullmore. Hierarchical modularity in human brain functional networks. *Frontiers in Neuroinformatics*, 37(3), 2010.

[18] M. E. J. Newman. Analysis of weighted networks. *Phys. Rev. E*, 70:056131, 2004.

[19] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69(2):026113, 2004.

[20] Xinyu Que, Fabio Checconi, Fabrizio Petrini, and John A. Gunnels. Scalable community detection with the louvain algorithm. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 28–37. IEEE Computer Society, 2015.

[21] Christian L. Staudt and Henning Meyerhenke. Engineering parallel algorithms for community detection in massive networks. *IEEE Trans. Parallel Distrib. Syst.*, 27(1):171–184, 2016.

[22] TOP500 Supercomputer Sites, list for November 2016. http://www.top500.org.

[23] Amanda L. Trauda, Peter J. Muchaa, and Mason A. Porter. Social structure of facebook networks. *Physica A: Statistical Mechanics and its Applications*, 391(16):4165–4180, 2012.

[24] Brendan Vastenhouw and Rob H. Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM Review*, 47(1):67–95, 2005.

[25] Matthew L. Wallace, Yves Gingras, and Russell Duhon. A new approach for detecting scientific specialties from raw cocitation networks. *J. Am. Soc. Inf. Sci. Technol.*, 60(2):240–246, 2009.

[26] Charith Wickramaarachchi, Marc Frîncu, Patrick Small, and Viktor K. Prasanna. Fast parallel algorithm for unfolding of communities in large graphs. In *IEEE High Performance Extreme Computing Conference, HPEC 2014, Waltham, MA, USA, September 9-11, 2014*, pages 1–6, 2014.

[27] Jianping Zeng and Hongfeng Yu. Parallel modularity-based community detection on large-scale graphs. In *2015 IEEE International Conference on Cluster Computing, CLUSTER 2015, Chicago, IL, USA, September 8-11, 2015*, pages 1–10, 2015.

[28] XN. Zuo, R. Ehmke, M. Mennes, D. Imperati, FX. Castellanos, O. Sporns, and MP. Milham. Network centrality in the human functional connectome. *Cerebral Cortex*, 22:1862–1875, 2012.

IV

On Stable Marriages and Greedy Matchings

Fredrik Manne*, Md. Naim*, Håkon Lerring*,
and Mahantesh Halappanavar†

Research on stable marriage problems has a long and mathematically rigorous history, while that of exploiting greedy matchings in combinatorial scientific computing is a younger and less developed research field. We consider the relationships between these two areas. In particular we show that several problems related to computing greedy matchings can be formulated as stable marriage problems and as a consequence several recently proposed algorithms for computing greedy matchings are in fact special cases of well known algorithms for the stable marriage problem.

However, in terms of implementations and practical scalable solutions on modern hardware, work on computing greedy matchings has made considerable progress. We show that due to this strong relationship many of these results are also applicable for solving stable marriage problems. This is further demonstrated by designing and testing efficient multicore as well as GPU algorithms for the stable marriage problem.

1 Introduction

In 1962 Gale and Shapley formally defined the stable marriage problem and gave their classical algorithm for its solution [5]. Since then this field has grown tremendously with numerous applications both in mathematics and in economics. For a recent overview see the book by Manlowe [17]. Graph matching is a related area where the object is also to find pairs of entities satisfying various optimality criteria. These problems find a large number of applications. For an overview motivated from combinatorial scientific computing see [22].

While research on stable marriage problems has mainly focused on theory and mathematical rigor, work on graph matching in scientific applications has a larger practical component concerned with implementing and testing code on various computer architectures with the intent of developing fast scalable algorithms.

In this paper we investigate the connection between one type of matching problems, namely those of computing greedy weighted matchings, and algorithms

*Department of Informatics, University of Bergen, N-5020 Bergen, Norway. Email: {fredrikm,naim}@ii.uib.no, hakon@lerring.no

†Pacific Northwest National Laboratory, 902 Battelle Boulevard, P.O.Box 999, MSIN J4-30, Richland, WA 99352, USA. Email: mahantesh.halappanavar@pnnl.gov

for solving stable marriage problems. Although there exist exact algorithms for solving various weighted matching problems these tend to have running times that typically involve the product of the number of vertices and the number of edges. As large graph instances can contain tens of millions of vertices and billions of edges it is clear that such algorithms can easily become infeasible. For this reason there has been a strong interest in developing fast approximation algorithms and also in parallelizing these, see [19] and the references therein. Although such algorithms typically only guarantee an approximation factor of 0.5 compared to the optimal one, practical experiments have shown that they are very often only within a few percent from optimal. One such algorithm is the classical greedy algorithm applied to an edge weighted graph. Here edges are considered by decreasing weight and an edge is included in the matching if it is not adjacent to an already included edge.

The main contributions of this paper are as follows. Initially we consider implementation issues when designing efficient algorithms for the stable marriage problem. Next, we show that several recently published algorithms for computing greedy matchings are in fact special cases of classical algorithms for stable marriage problems. This also includes a generalization of the matching problem known as b-matching where a vertex can be matched with several other vertices in the final solution. Due to the strong similarities between the stable marriage problem and greedy matching, we show that one can apply recent results on designing scalable greedy matching algorithms to the computation of stable marriage solutions. This is verified by presenting efficient parallel implementations of various types of Gale-Shapley type algorithms for both multithreaded computers as well as for GPUs.

The remainder of the paper is organized as follows. In Section 2 we review the Gale-Shapley algorithm and consider implementation issues related to this. Next, in Section 3 we show that the computation of a greedy matching can be reformulated as a stable marriage problem. In Section 4 we give parallel implementations of the Gale-Shapley and McVitie-Wilson algorithms and show their scalability, before concluding in Section 5.

2 The Stable Marriage Problem

In the following we review the stable marriage (SM) problem and how it can be solved using the Gale-Shapley algorithm and consider some implementation issues. Finally, we review some generalizations of the SM problem.

The SM problem is defined as follows. Let L and R be two equal sized sets $L = \{l_1, l_2, \ldots, l_n\}$ and $R = \{r_1, r_2, \ldots, r_n\}$. The entries in L are typically referred to as "men", while the entries in R are referred to as "women". Every man and woman has a total ranking of all the members of the opposite sex. These give the "desirability" for each participant to match with a member of the other set. The object is to find a complete matching M (i.e. a paring) between the entries in L

and R such that no two $l_i \in L$ and $r_j \in R$ both would obtain a higher ranked partner if they were to abandon their current partner in M and rematch with each other. Any solution satisfying this is *stable*.

Gale and Shapley [5] defined the stable marriage problem and also proposed the first algorithm for solving it. The algorithm operates in rounds as follows. In the first round each man in L proposes to his most preferred woman in R. Each woman will then reject all proposals except the one she has ranked highest. In subsequent rounds each man that was rejected in the previous round will again propose to the woman which he has ranked highest, but now disregarding any woman that he has already proposed to in previous rounds. Gale and Shapley showed that this process will terminate with each man in L being matched to a woman in R and that this solution is stable. Although an SM instance can have many stable solutions, the Gale-Shapley algorithm will always produce the same one.

An important variant of this problem is when each participant has only ranked a subset of the opposing participants. This is known as the stable marriage problem with incomplete lists (SMI). Any solution M to an SMI instance must then in addition to being stable, also consists of mutually ranked pairs (l_i, r_j). The SMI problem is solved by the Gale-Shapley algorithm, but the solution might not be complete leaving some participants unmarried [5]. There exists a number of variants of the SM problem, for two comprehensive surveys see the books [9, 17]. In the following we will only consider the classical SM and SMI problems.

The original Gale-Shapley algorithm is described as operating in rounds, where only the men who were rejected in round t will propose in round $t + 1$. It is not stated in which order the proposals in a round should be made or what kind of data structures to use. If one traverses the men in L in their original order in each round and lets each rejected man propose once it is discovered, then the men always propose in the same relative order in each round. The running time of such a scheme is $\Theta(n^2)$ even for an instance of SMI. If one is willing to forgo the requirement that the proposals in each round must be made in the same relative order then it is not hard to design an implementation of the Gale-Shapley algorithm with running time proportional to the number of actual proposals made. To do this one maintains a queue Q of men waiting to make their proposals. Initially $Q = L$ and in each step of the algorithm the man at the front of the queue gets to propose to his current best candidate $r_j \in R$, and any rejected l_i is inserted at the end of the queue. This will ensure that all men rejected in round t gets to propose before any man rejected in round $t + 1$, but the relative order among the men might not always be the same. The algorithm terminates when the queue is empty.

One simple enhancement of the Gale-Shapley algorithm is that no $l_i \in L$ should propose to an $r_j \in R$ who already has a proposal from someone whom r_j ranks higher than l_i, as such a proposal will be rejected. Thus each l_i should

propose to his most preferred r_j where l_i has not already been rejected and where r_j ranks l_i higher than her current best proposal (if any). This means that it is sufficient to only maintain the current best proposal for each r_j. When the algorithm terminates these proposals will make up the solution. We give our complete implementation of the Gale-Shapley algorithm in Algorithm 1.

In Algorithm 1 each r_j has a variable $suitor(r_j)$ initialized to $NULL$ that holds her current best proposal. Similarly, $ranking(r_j, l_i)$ returns r_j's ranking of l_i (as a number in the range 1 through n). We define $ranking(r_j, NULL) = n + 1$ to ensure that any proposal is better than no proposal. The function $nextCandidate(l_i)$ will initially return l_i's highest ranked woman and then for successive calls return the next highest ranked one following the last one retrieved.

For an SM instance it is straight forward to precompute the values of $ranking()$ in $O(n^2)$ time. However, for an SMI instance maintaining a complete $ranking()$ table would require $O(n^2)$ space and also proportional time to initialize it. In this case it is more efficient to store the value of $ranking(r_i, l_j)$ together with r_i in l_j's ranking list so that it can be fetched in $O(1)$ time when needed. These values can be precomputed in time proportional to the sum of the lengths of the ranking lists. To do this one first traverses the women's lists building up lists for each man l_j with the women that have ranked him along with in what position. Then using an array $position()$ of length n initially set to 0, the list of each man l_j is processed as follows. For each woman r_i that has ranked l_j we store the value l_j along with in what position r_i has ranked l_j in $position(r_i)$. We next traverse l_j's priority list and for each r_i in the list we look up $position(r_i)$ and see if it contains l_j. If so, we fetch r_i's ranking of l_j and store it together with l_j's ranking of r_i. At the same time any r_i that has not ranked l_j but which l_j has ranked can be purged from the priority list of l_j.

Algorithm 1 The Gale-Shapley algorithm using a queue

```
1: Q = L
2: while Q ≠ ∅ do
3:    u = Q.dequeue()
4:    partner = nextCandidate(u)
5:    while ranking(partner, u) >
      ranking(partner, suitor(partner)) do
6:       partner = nextCandidate(u)
7:    if suitor(partner) ≠ NULL then
8:       Q.enqueue(suitor(partner))
9:    suitor(partner) = u
```

McVitie and Wilson [4] gave a recursive implementation of the Gale-Shapley algorithm. This algorithm also iterates over the men, allowing each one to make a proposal to his most preferred woman. But if this proposal is rejected or if it

results in an existing suitor being rejected then the just rejected man recursively makes a new proposal to his best remaining candidate. The recursion continues until a proposal is made such that no man is rejected (because the last proposed to woman did not already have a suitor). At this point the algorithm will continue with the outer loop and process the next man. When all men have been processed the algorithm is finished. It is shown in [4] that the McVitie-Wilson algorithm gives the same solution as the Gale-Shapley algorithm. We note that similarly to the Gale-Shapley algorithm it is possible to avoid proposals that are destined to be rejected because the proposed to woman already has a better offer.

Comparing the two algorithms each man will consider exactly the same women before ending up with his final partner. The only difference is the order in which this is done. While the Gale-Shapley algorithm will maintain a list of men that needs to be matched, the McVitie-Wilson algorithm will always maintain a solution where each man considered so far is matched before including a new man in the solution. We note that one can implement a non-recursive version of the McVitie-Wilson algorithm simply by replacing the queue Q in Algorithm 1 by a stack and replacing the *dequeue*() and *enqueue*() operations with *pop*() and *push*() operations respectively. To see that this will result in the McVitie-Wilson algorithm it is sufficient to first note that the initial placement of L in Q is equivalent to an outer loop that processes each man once. Any rejected man will then be placed at the top of the stack and therefore be processed immediately, similarly to a recursive call in the original algorithm.

Wilson [27] showed that for any profile of womens preferences, if the men's preferences are random, then the expected sum of men's rankings of their mates as assigned by the Gale-Shapley algorithm is bounded above by $n(1+1/2+...+1/n)$. Knoblauch [16] showed that this is also an approximate lower bound in the sense that the ratio of the expected sum of men's rankings of their assigned mates and $(n+1)((1+1/2+...+1/n)-n)$ has limit 1 as n goes to ∞. Thus if the men's preferences are random then this sum is $\Theta(n \ln n)$ for large n. However, it is not hard to design instances where this sum is $\Theta(n^2)$. One such case is when the men have identical preferences.

2.1 Generalizations of SM We next review two generalizations of the SM problem. The stable roommates (SR) problem consists of a set of n persons, each one with a complete ranking of all the others persons. The objective is now to pair two and two persons together, such that there is no pair (x, y) of persons where x is either unmatched or prefers y to its current partner, while at the same time y is either unmatched or prefers x to its current partner. Just like for the SM problem, such a solution is stable. Unlike the SM problem there might not exist a solution for an SR instance. If some persons have only ranked a subset of the other participants we get the stable roommates problem with incomplete lists (SRI). Irving gave an algorithm for computing a stable solution to an SRI problem

or to determine that no such solution exists [11]. This algorithm operates in two stages, where the first one is similar to the Gale-Shapley algorithm where each person makes, accepts and rejects proposals. The second phase of the algorithm is slightly more involved but does not change the running time of $O(n^2)$. For more information on the SR and SRI problems see [9, 17].

In the last generalization each person can be matched with more than one partner. More formally, we are looking for a stable solution to an SM, SMI, SR, or SRI instance where each person v_i is matched with at most $b(v_i)$ other persons, where $b(v_i) \geq 1$. Being stable again means that no two persons v_i and v_j would both obtain a better solution if they were to match with each other, either by dropping one of their current partners or if v_i has fewer than $b(v_i)$ partners or if v_j has fewer than $b(v_j)$ partners.

For the SM problem this gives us the many-to-many stable assignment problem (MMSA), where each "man" and "woman" can be matched with several participants of the opposite sex. This was solved by Baïou and Balinski [2] who presented a general algorithm based on modelling this as a graph searching problem. Applying the last generalization to an SR instance, gives the stable fixtures problem [12] for which Irving and Scott gave an $O(n^2)$ algorithm. Similarly to Irving's algorithm for SRI, this also consists of two stages, where the first stage is a natural extension of the Gale-Shapley algorithm to handle that each person can participate in multiple matches.

3 Matching Problems

We next explore the relationship between stable marriages and weighted matchings in graphs. A matching M on a graph $G(V, E)$ is a subset of the edges such that no two edges in M share a common end point. For an unweighted graph the object is to compute a matching of maximum cardinality. For an edge weighted graph a typical problem is to compute a matching M such that the sum of the weights of the edges in M is maximum over all matchings. Another variant could be to compute the maximum weight matching over all matchings of maximum cardinality.

We consider the Greedy algorithm for computing a matching of maximum weight in an edge weighted graph where all weights are positive. This algorithm considers edges by decreasing weight. In each step the heaviest remaining edge (u, v) is included in the matching before removing any edge incident on either u or v. If the weights of the edges in G are unique or if a consistent tie breaking scheme is used then it follows that the solution given by Greedy is also unique. In the following we will always assume that this is the case. It is well known that Greedy guarantees a solution of weight no worse than 0.5 times the weight of an optimal solution. We label the problem of computing a greedy matching in an edge weighted graph as the GM problem.

Given an instance G of the GM problem one can construct an equivalent

instance of the SRI problem by sorting the edges incident on each vertex u by decreasing weight, and letting this be the ranking of u's neighbors in the SRI instance. With this construction a solution to the GM problem is equivalent to a stable solution of the corresponding SRI problem. Consider the heaviest edge (u, v) in the graph. This is included in the GM solution and the corresponding vertex pair must also be part of any solution to the SRI instance, otherwise this solution would not be stable as both u and v would prefer to match with each other over any other partner. We can thus include (u, v) in the solutions to both instances and also remove u and v from further consideration. For the GM problem this means that any edges incident on either u or v are removed and for the SRI instance u and v are removed from all ranking lists. One can then repeat the argument using the heaviest remaining edge, and it follows by induction that the two solutions are identical. It is also clear that the corresponding SRI instance always has a unique stable solution.

The above construction implies that the solution given by Greedy is stable in the sense that there does not exist an edge $(u, v) \notin M$ such that the weight of M would increase if (u, v) was added to M while removing any edges incident on either u or v from M. This observation is often stated as that the solution given by Greedy does not contain any augmenting path containing three or fewer edges. An augmenting path of length k is a path containing k edges starting with an edge in M and then alternating between edges not in M and edges in M, such that if one was to replace all the edges on the path that belong to M with those that are not in M then the weight of the solution would increase.

We next show that the solution given by Greedy can also be obtained by solving an associated SMI (or SM) instance. To the best of our knowledge this result has not been shown previously.

Given an instance of the GM problem on an edge weighted graph $G(V, E)$. We define an SMI instance G' from G as follows. Let L and R be the sets of men and women respectively, both of size $n = |V|$. Any man l_i will include exactly those r_j in its ranking where there is an edge $(v_i, v_j) \in E$. As the edges in G are not directed, this also means that l_j will rank r_i. Similarly, any woman r_j will include exactly those l_i in her ranking where there is an edge $(v_i, v_j) \in E$. Both men and women order their lists by decreasing weight of the corresponding edges in G. Thus every $(v_i, v_j) \in E$ gives rise to four rankings in G'. We call the two pairs (l_i, r_j) and (l_j, r_i) for the corresponding pairs of (v_i, v_j).

Lemma 3.1. *Given a graph G with SMI instance G' as described above and let M be the greedy matching on G. Then the pairs in G' corresponding to the edges in M make up the unique solution to the SMI problem on G'.*

Proof. The proof is by induction on the edges of M considered by decreasing weight. Let (v_i, v_j) be the edge of maximum weight in G. Then $(v_i, v_j) \in M$ and it also follows from the construction of G' that l_i will rank r_j highest. Similarly,

l_i will also have the highest ranking among the men ranked by r_j. Thus (l_i, r_j) must be included in any stable solution of G'. A similar argument shows that the edge (l_j, r_i) will also be included in such a solution.

Assume now that the pairs in G' corresponding to the $k \geq 1$ heaviest edges in M must be included in any stable solution and consider the two pairs (l_s, r_t) and (l_t, r_s) corresponding to the $k+1$st heaviest edge (v_s, v_t) in M.

It is clear that any solution where l_s is matched to a woman that he has ranked after r_t while at the same time r_t is matched to a man that she has ranked after l_s, cannot be stable as both l_s and r_t would be better of if they were to match with each other. Thus if (l_s, r_t) is not included in a stable solution at least one of l_s and r_t must be matched to a partner which he or she has ranked higher than the other one of $\{l_s, r_t\}$. Assume therefore that l_s is matched to r_u and that l_s has ranked r_u higher than r_t, implying that the weight of (v_s, v_u) is greater than the weight of (v_s, v_t) in G. But since $(v_s, v_t) \in M$ it follows that $(v_s, v_u) \notin M$. Thus v_u must be matched to some other vertex v_z in M. And since $(v_s, v_u) \notin M$ the weight of (v_u, v_z) must be greater than that of (v_s, v_u). By the induction hypothesis the pairs in G' that correspond to the k heaviest edges in M must be included in any stable solution in G'. It therefore follows that l_z must be matched to r_u in any stable solution on G' contradicting that l_s is matched to r_u. A similar argument shows that r_t cannot be matched to any man in L to which she gives higher priority than she gives to l_s. Thus the pair (l_s, r_t) must be in any stable solution in G'. The argument for why (l_t, r_s) also must be included in a stable solution is analogous. It follows that any pair in G' corresponding to an edge in M must be part of a stable marriage in G'.

It only remains to show that once the pairs corresponding to the edges in M have been included in the solution M' to G', then it is not possible to match any other pairs in G'. If M' contains a pair (l_i, r_j) in addition to the pairs corresponding to the edges in M then $(v_i, v_j) \notin M$ and neither v_i nor v_j can be matched in M. But since l_i has ranked r_j (and vice versa) it follows that $(v_i, v_j) \in E$ and that M can be expanded with (v_i, v_j). This contradicts that M is maximal and the result follows.

We next consider the b-matching problem which is a generalization of the regular weighted matching problem similar to the many-to-many stable assignment problem and the stable fixtures problem. A b-matching on G is a subset of edges $M \subseteq E$ such that every vertex $v_i \in V$ has at most $b(v_i)$ edges in M incident on it. The objective is to compute the b-matching of maximum weight. A 0.5 approximation can again be computed using the greedy algorithm that selects edges by decreasing weight and whenever $b(v_i)$ edges incident on v_i have been selected, the remaining edges incident on v_i are removed [20]. Setting $b(v_i) = 1$ for all $v_i \in V$ gives a regular (one) matching.

It is straight forward to see that the stable fixtures problem is also a generalization of greedy b-matching. Given an instance of the greedy b-matching

problem, one can also construct an equivalent many-to-many stable assignment instance by setting the bounds $b(l_i)$ and $b(r_i)$ equal to $b(v_i)$. A proof similar to that of Lemma 3.1 shows that these two problems have equivalent solutions.

3.1 Algorithmic Similarities As a consequence of the fact that the solution given by GREEDY can be obtained by either solving a properly designed instance of SMI or SRI, any algorithm that solves either of these two problems can also be used to compute a greedy weighted matching. This process can be simplified as it might be possible to run an SMI or SRI algorithm directly on the original graph. Let G be an instance of GM and G' its corresponding SMI instance. Also let $\{r_{s_1}, r_{s_2}, \ldots, r_{s_f}\}$ and $\{l_{t_1}, l_{t_2}, \ldots, l_{t_g}\}$ be the ranked lists of l_i and r_i respectively. Then if follows from the construction of G' that $f = g$ and that $s_k = t_k$ for all k. Thus any proposal made to r_i could be handled directly by l_i as he has the same information as r_i. It follows that one can merge l_i and r_i into one node v_i that handles making, accepting, and rejecting proposals related to l_i and r_i. In this way both the Gale-Shapley and the McVitie-Wilson algorithm can be used directly on edge weighted general graphs to compute greedy matchings, but now using edge weights to rank potential partners. Irving's algorithm [11] for solving the SRI problem consists of two stages, of which the first is exactly this algorithm used on a general graph. If the rankings in an SRI instance are based on edge weights from a GM instance then the first phase will produce the greedy solution which is stable, thus making the second phase of the algorithm redundant.

Previous efforts at designing fast parallel greedy matching algorithms have been based on the notion of dominant edges. These are edges that are heavier than any of their neighboring edges. Preis showed that an algorithm based on repeatedly including dominant edges in the matching while removing any edges incident on these will result in the same solution as GREEDY [23]. Based on this observation Manne and Bisseling developed the pointer algorithm [18], which was further enhanced by Manne and Halappanavar in the SUITOR algorithm [19]. We note that the SUITOR algorithm is identical to the McVitie-Wilson algorithm applied to a general edge weighted graph, while the pointer algorithm has strong resemblances to the Gale-Shapley algorithm as outlined in Algorithm 1.

The same type of relationship also holds true between the greedy b-matching problem and the many-to-many stable assignment problem. The algorithm presented in [2] can be instantiated to solve the b-matching problem using a Gale-Shapley type algorithm where a vertex v will accept the $b(v)$ best offers at any given time. We note that this is the same algorithm as the one presented in [7] and also [14] for computing a greedy b-matching. In [13] the authors experiment with what they call *delayed* versus *eager* rematching of rejected suitors. The difference between these two variants is the same as that between a Gale-Shapley and a McVitie-Wilson style algorithm.

4 Experiments

As shown in Section 3 much of the theory for greedy matching algorithms are mainly restricted versions of previous results from the theory of stable marriages. However, the work on greedy matchings has to a large extent been driven by a need for developing scalable parallel algorithms for use in scientific applications. This has lead to the implementation of Gale-Shapley and McVitie-Wilson type matching algorithms on a large variety of architectures, including distributed memory machines [3, 18], multicore computers [10, 15, 19], and GPUs [1, 21].

There has been less emphasis on implementations and developing working code for the stable marriages problem. We believe that much of the work done on greedy matchings can easily carry over to developing efficient code for stable marriage problems. To show the feasibility of this we have developed shared memory implementations of both the Gale-Shapley and McVitie-Wilson algorithms. We used OpenMP to parallelize the Gale-Shapley algorithm and both OpenMP and CUDA for parallelizing the McVitie-Wilson algorithm.

In weighted matching both endpoints of an edge (u, v) evaluates the importance of the edge to the same number, i.e. the weight of the edge. Whereas in the stable marriage problem both u and v assign their own ranking of the other. Thus the main difference between greedy matching algorithms such as those presented in [18, 19] and the Gale-Shapley algorithm is that in the latter, a man who makes a proposal evaluates his chance of success based on the woman's ranking, instead of on a common value. Another difference is that it is typically not assumed in weighted matching problems that the neighbor list of a vertex is sorted by decreasing weight. It was shown in [19] that when this is the case, then it both simplifies the algorithm and also speeds up the execution considerably.

Our parallelization strategy for the McVitie-Wilson algorithm using OpenMP closely follows that of the Suitor algorithm as presented in [19], while our CUDA version of the same algorithm is a simplified version of the Suitor algorithm used in [21]. In both of our OpenMP algorithms the set of men is initially partitioned among the threads who then each run a local version of the corresponding algorithm until completion. A thread will first search the list of the current man to locate the woman he gives highest priority and where the woman also prefers him to her current suitor (if any). If such a woman is discovered the thread will use a compare-and-swap (CAS) operation to become the new suitor of the woman. In this way it is assured that no other thread has changed the suitor value. If the CAS operation succeeds the previous suitor (if any) is treated according to the current strategy and is inserted in a local stack (McVitie-Wilson) or a local queue (Gale-Shapley). If the CAS operation failed because some other thread had already changed the suitor value, then if the current man can still beat the new suitor then the thread will retry with a new CAS operation, otherwise it will continue searching for the next eligible woman.

There is a difference between the algorithms in how they can handle load

imbalance. For the parallel Gale-Shapley algorithm it is possible to synchronize the threads after each round of proposals and then redistribute the unmarried men to the threads before moving on to the next round. However, synchronization tends to be costly, and experiments done on greedy matching problems indicate that this is typically not worth the effort. For the McVitie-Wilson algorithm one can load balance the algorithm by using one of the dynamic load balancing strategies in OpenMP in the initial assignment of men to threads. This strategy was used successfully in experiments for the SUITOR matching algorithm on graphs with highly varying vertex degrees [19].

For the McVitie-Wilson CUDA algorithm we assign one thread to each man.Each thread then executes the algorithm similarly to the OpenMP version using a CAS operation to assign a man as the suitor of a particular woman. Using only one thread per man allows for a larger number of thread blocks which the runtime environment can balance across the device. But as the threads within one physical warp operate in SIMD, the run time of all threads in the same warp will be equal to the maximum execution time of any of the threads. Similarly, the threads within the same thread block will not release resources until all threads in the block have finished executing. It would have been possible to statically assign multiple men to each thread or to design a dynamic load balancing scheme with the aim of evening out the work load. But this would have resulted in a more complicated algorithm and as our main goal is proof of concept we did not pursue this.

Implementing the Gale-Shapley algorithm on the GPU presents additional challenges compared to the McVitie-Wilson algorithm. In a Gale-Shapley algorithm the threads would have to be grouped so that each thread group operates on one common queue, where the size of the group could be either a subset of threads in a warp or all the threads in one thread block. As the number of free men monotonically decreases between rounds, there should initially be more men than threads assigned to the same queue, something that would complicate the algorithm. Also, having several threads operate on one common queue would require synchronization which can be time consuming on the GPU. For this reason we chose not to implement the Gale-Shapley algorithm using CUDA.

As we are not aware of any sufficiently large publicly available data sets for the stable marriage problem, we have designed two different random data sets. The first set has been constructed to be relatively easy to solve, whereas the second set is intended to be more time consuming. We label these sets as *easy* and *hard* respectively. Each instance consists of n men and n women. In the *easy* data set each man is assigned a random number $\epsilon \in [0, 1]$ and then randomly picks and ranks $(1 + \epsilon) \ln n$ women. Each woman then ranks exactly the men that ranks her. With this configuration more than 98% of the participants were matched in every final solution and the total number of proposals is at most $2n \ln n$ with an

average of $n \ln n$. In the *hard* data set each man has an identical complete random ranking of all the women. Similarly, all woman share the same random ranking of all the men. Thus there will always exist a complete stable solution and the total number number of considered women will always be $n(n+1)/2$. Moreover, in the *hard* instances there will be contention among the men for obtaining the same set of women, and thus cause substantial synchronization requirements for the parallel algorithms. One obvious difference between the datasets is that the *easy* instances will require more memory access as each participant has an individual ranking list, while for the *hard* instances all rankings are stored in two shared vectors of length n. For each value of n we have generated 5 instances and for each of these we run each algorithm 3 times. For all timings we take the average of these 15 runs.

The OpenMP algorithms are run on a computer with two Intel Xeon E5-2699 processors and 252 Gbytes of memory. Each processor has 18 cores and runs at 2.30GHz. The GPU is a Tesla K40m with 12 GB of memory, 2880 cores running at 745 MHz and has CUDA compute capability version 3.5. For all parallel algorithms we measure their speedup against the fastest sequential algorithm run on the Intel Xeon machine.

In Figure 1 we present results from the *easy* instances when n varies from 5M up to 25M in steps of 5M. For the OpenMP algorithms the number of threads is set to 36. For most of these instances the running time stays well below one second. It is only for the n =25M instance that the GPU algorithm uses slightly more time than one second. This is also the largest *easy* instance that could be run on the GPU.

For smaller instances the GPU algorithm is the fastest one but as the problem size increases it is slowing down compared to the OpenMP algorithms. In general the OpenMP Gales-Shapley algorithm is faster than the OpenMP McVitie-Wilson algorithm with as much as 12%. For this setup one would expect that the graph displaying the time would resemble $n \ln n$ as the computing resources is the same for each instance. This is most true for the OpenMP algorithms where the time increases close to linearly with n, whereas the time grows faster than n for the GPU algorithm. This can be seen further in Figure 2 which shows the speedup of the OpenMP Gale-Shapley algorithm and the GPU McVitie-Wilson algorithm compared to the sequential Gale-Shapley algorithm. The OpenMP algorithm gives a constant speedup of about 9, while the speedup of the GPU algorithm starts out at about 17 but then drops sharply as the size of the instances increase. Thus this is most likely due to insufficient memory on the GPU. On these problems the sequential McVitie-Wilson algorithm was on average 27% slower than the sequential Gale-Shapley algorithm.

Figure 3 shows running times of the OpenMP algorithms using 36 threads as n increases up to 125M. It can be observed that the tendencies for the smaller instances still remain true for the larger ones. We note that the worst running

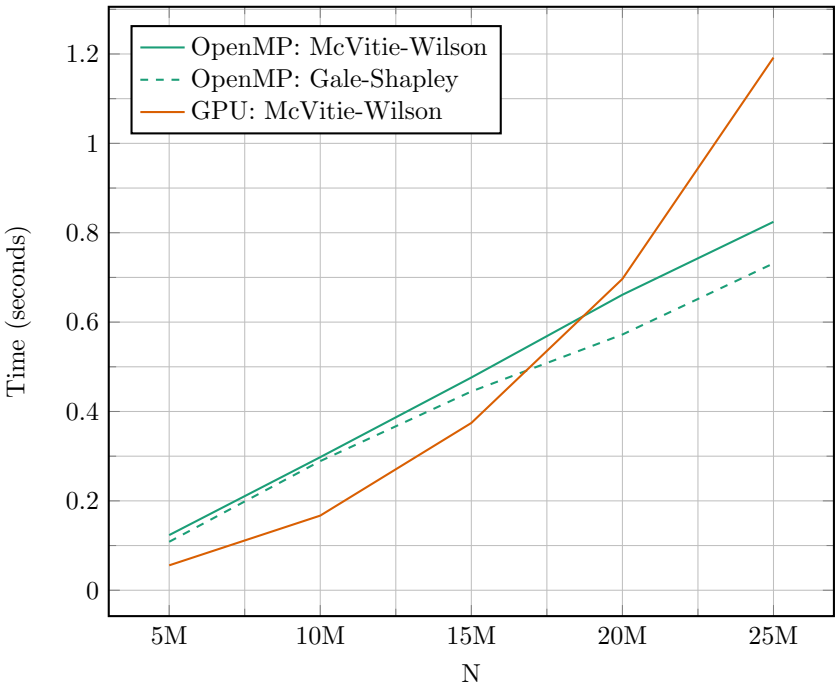

Figure 1: Running time on the *easy* dataset

time is only marginally larger than four seconds on the largest instance. Figure 4 shows the speedup of the OpenMP algorithms compared to the sequential Gale-Shapley algorithm for the three largest instances when using $t = 1, 9, 18, 27$ and 36 threads. The Gale-Shapley algorithm outperforms the McVitie-Wilson algorithm in almost all instances and reaches a speedup of almost 14 on the $n = 75$M instance.

Figure 5 shows the running time on *hard* instances where n increases from 100K up to 500K. The OpenMP codes are again run using 36 threads. As the dataset only consists of two vectors we can run the problems using all three

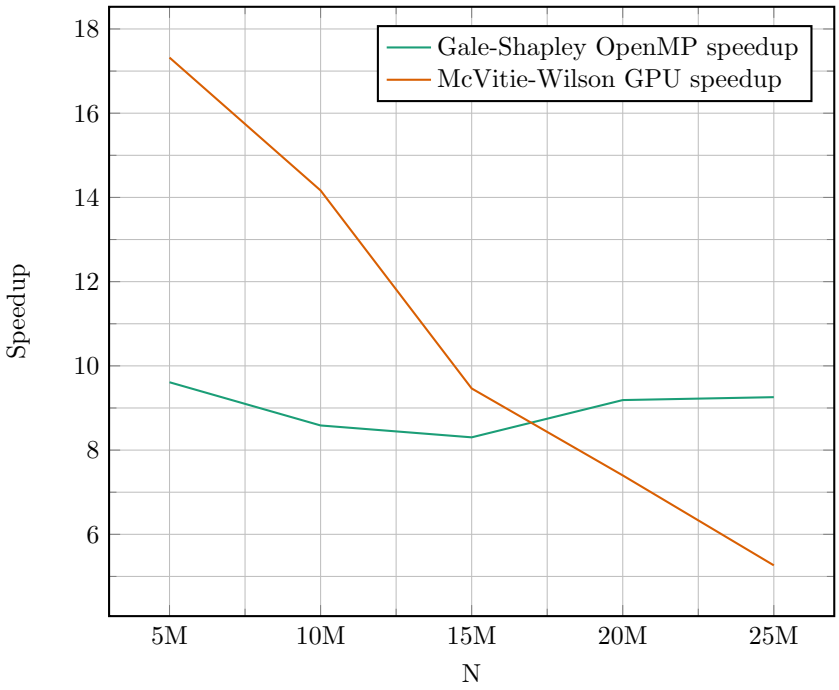

Figure 2: Speedup on the *easy* dataset

codes, the only limiting factor being time. Since the total amount of work grows as $\Theta(n^2)$ on these instances, it is to be expected that they will require more time than the *easy* ones. From the figure it can be observed that there is little difference in the running time between the Gale-Shapley OpenMP code and the McVitie-Wilson GPU code, which both take close to 250 seconds on the largest instance. However, the McVitie-Wilson OpenMP code performs considerably better, and is a factor of 5 times faster on the largest instance. This difference is also displayed in Figure 6 which gives the speedup of the same instances compared to the sequential McVitie-Wilson algorithm. While the McVitie-Wilson

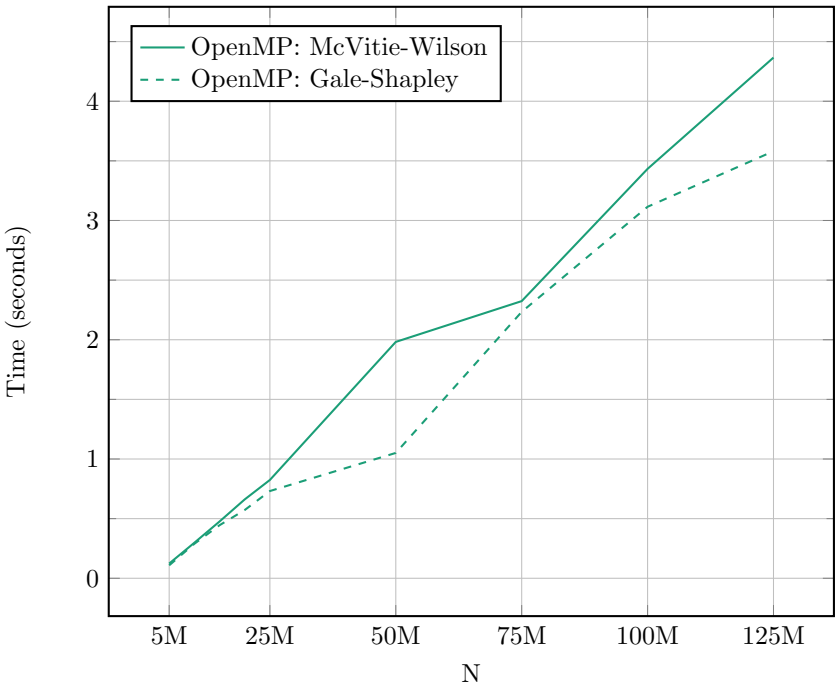

Figure 3: Running time of OpenMP algorithms on large *easy* instances

algorithm reaches a speedup of close to 22 when running on the 36 threads, the parallel Gale-Shapley algorithm is never more than a factor of 2.5 faster than the sequential McVitie-Wilson algorithm. For these instances the sequential Gale-Shapley algorithm was on average 109% slower than the sequential McVitie-Wilson algorithm.

We believe that some the difference between the OpenMP algorithms can be explained by how the algorithms handle the large number of rejections. While the Gale-Shapley algorithm has to store each rejected man to memory and retrieve a new one, the McVitie-Wilson algorithm can continue working on the rejected

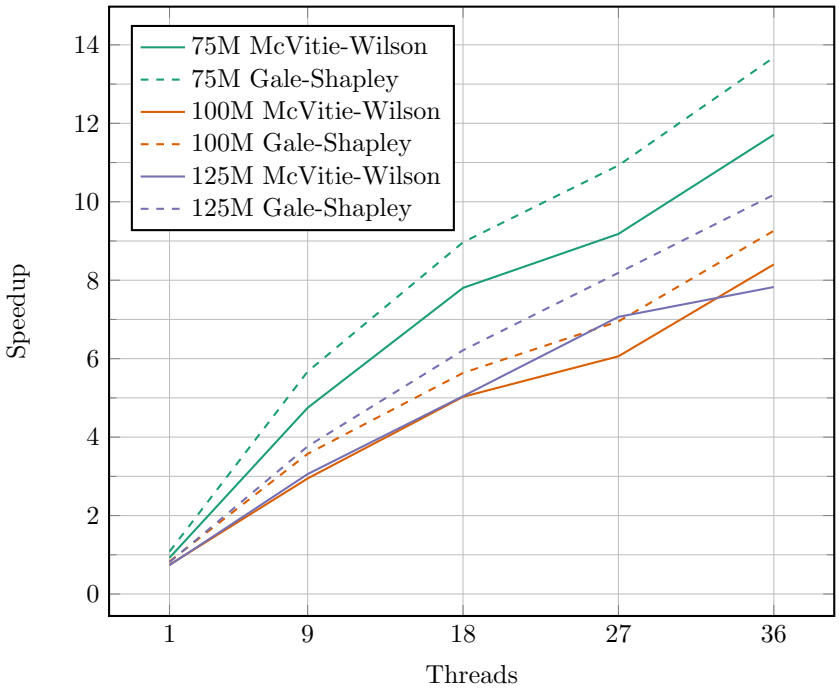

Figure 4: Speedup on large *easy* datasets

man without needing to access relatively slow memory. The poor performance of the McVitie-Wilson GPU algorithm compared to the OpenMP one is most likely due to how the machines handle contention for shared resources. The GPU algorithm utilizes several thousand concurrent threads that, at least initially, will be competing for matching their man with the same set of women. Synchronizing this will lead to a much larger strain on the system compared to that of the relatively low number of threads in the OpenMP algorithm.

Finally, figures 7 and 8 show the number of considered proposals per second for both *easy* and *hard* datasets on the OpenMP algorithms. For each instance this

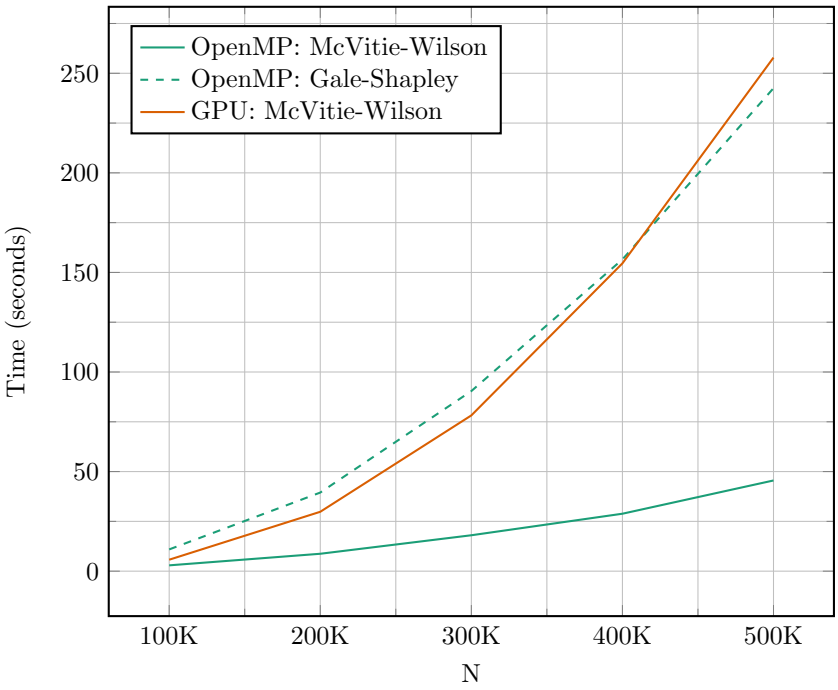

Figure 5: Running time on *hard* datasets

number is given as the sum over each man of his ranking of his final partner and then divided by the total time. In sparse graph algorithms this is often referred to as the number of traversed edges per second (TEPS) and is, among other things, used to rank the performance of computers in the Graph500 challenge [8].

For the *easy* instances the TEPS rate starts out at 10M for one thread and then increases to somewhere between 100M to 140M for 36 threads. Thus the efficiency when using 36 threads lies somewhere in the range of 30% to 40%. For the *hard* instances the TEPS rate for the McVitie-Wilson algorithm starts out at about 150M and increases up to 2.75 billion when using 36 threads for an

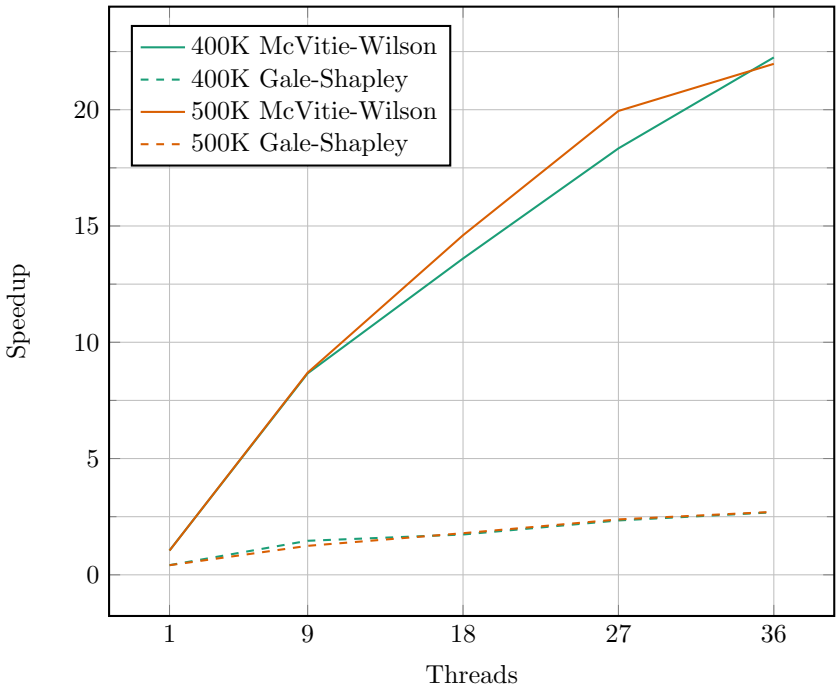

Figure 6: Speedup on *hard* datasets

efficiency rate of about 50%. As already noted the Gale-Shapley algorithm does not scale well on these instances. Comparing the TEPS rate between the *easy* and the *hard* instances when using the McVitie-Wilson algorithm on the same number of threads it can be observed that the maximum TEPS rate is more than a factor of 20 larger for the *hard* instances. This is most likely because the *hard* instances are not limited by access to memory as the whole dataset only consists of two vectors.

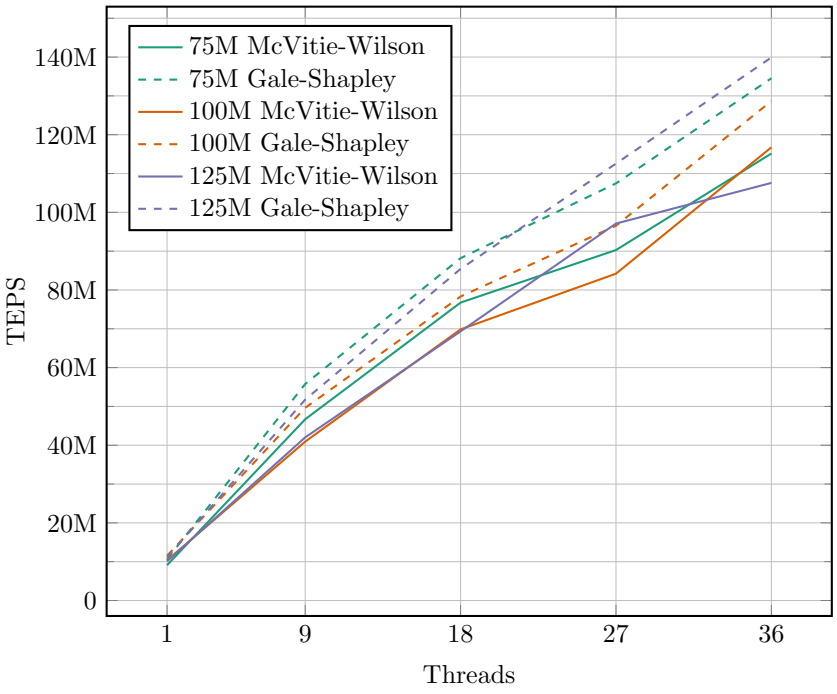

Figure 7: TEPS for *easy* datasets

5 Conclusion

In his book Manlove [17] lists some of the most noteworthy open problems related to SM. One of these is to determine if the SM problem is in the complexity class NC or not, that is, to determine whether the problem can be solved by an algorithm with polylogarithmic running time when using a polynomial number of processes. Efforts at designing such algorithms has mainly resulted in parallel algorithms requiring at least n^2 processes, and are thus mainly of theoretical interest [6, 26].

We are only aware of one previous attempt at implementing a parallel version

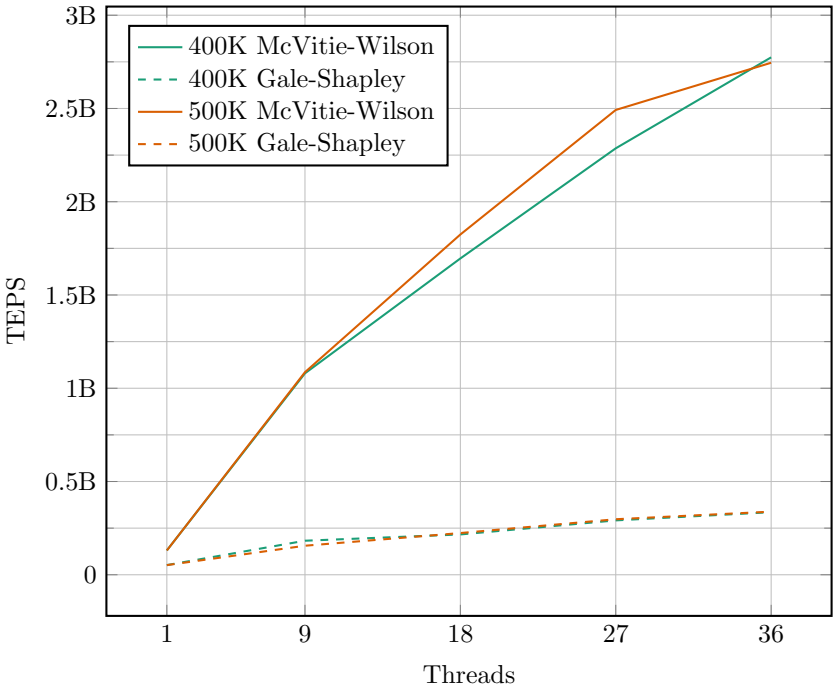

Figure 8: TEPS for *hard* datasets

of the Gale-Shapley algorithm and this did not result in any speedup [25]. Quinn [24] argues that one cannot expect a large speedup from a parallel Gale-Shapley style algorithm in practice as the algorithm cannot run faster than the maximal number of proposals made by any one man. We note that for a random instance the average number of proposals made by each man is in fact $O(\log n)$.

While the question of developing asymptotically faster parallel algorithms than those presented in Section 4 is of interest from a theoretical point of view, we believe that this is less relevant for a practitioner. To begin with the running time of the Gale-Shapley algorithm is linear in the instance size. Thus moderate

sized problem can already be solved rapidly. In addition, our current experiments on the SMI problem as well as previously experiments on GM problems shows that Gale-Shapley type algorithms scale well. One reason for this is that the size of the instance n is typically much larger than the number of threads used.

One notable difference between the formulations of the GM and the SMI problem is that for GM it is not assumed that the neighbor lists are initially sorted by decreasing weight in the same way as priority lists are ordered in SM. Thus work on developing parallel algorithms for the GM problem has focused on how one should search the neighbor lists. Suggested solutions include sorting the lists initially, searching through the list each time a new candidate is needed, or something in between. All of these strategies result in a running time that is superlinear in the input size. However, Preis's algorithm for GM has linear running time [23], but is more complicated and not suitable for parallel execution. We therefore ask if it is possible to design a linear time algorithm for the SM problem if the priority lists are not sorted, but instead given as real valued numbers such that $p_i(j)$ gives the value that person i assigns to person j of the opposite sex.

References

[1] B. O. Fagginger Auer and R. H. Bisseling. A GPU algorithm for greedy graph matching. In *Facing the multicore, Challenge II*, volume 7174, pages 108–119. LNCS, Springer, 2012.

[2] M. Baïou and M. Balinski. Many-to-many matching: stable polyandrous polygamy (or polygamous polyandry). *Discrete Applied Mathematics*, 101(1-3):1–12, 2000.

[3] Ü. V. Çatalyürek, F. Dobrian, A. H. Gebremedhin, M. Halappanavar, and A. Pothen. Distributed-memory parallel algorithms for matching and coloring. In *IPDPS Workshops*, pages 1971–1980, 2011.

[4] L. B. Wilson D. G. McVitie. The stable marriage problem. *Communications of the ACM*, 14(7):486–490, 1971.

[5] L. S. Shapley D. Gale. College admissions and the stability of marriage. *The American Mathematical Monthly*, 69(1):9–15, 1962.

[6] T. Feder, N. Megiddo, and S. A. Plotkin. A sublinear parallel algorithm for stable matching. *Theor. Comput. Sci.*, 233(1-2):297–308, 2000.

[7] G. Georgiadis and M. Papatriantafilou. Overlays with preferences: Distributed, adaptive approximation algorithms for matching with preference lists. *Algorithms*, 6(4):824–856, 2013.

[8] Graph 500. `http://www.graph500.org`.

[9] D. Gusfield and R. W. Irving. *The stable marriage problem.* The MIT press, 1989.

[10] M. Halappanavar, J. Feo, O. Villa, A. Tumeo, and A. Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *Int. J. High Perf. Comput. App.*, 26(4):413–430, 2012.

[11] R. W. Irving. An efficient algorithm for the "stable roommates" problem. *J. Algorithms*, 6(4):577–595, 1985.

[12] R. W. Irving and S. Scott. The stable fixtures problem - A many-to-many extension of stable roommates. *Discrete Applied Mathematics*, 155(16):2118–2129, 2007.

[13] A. Khan, A. Pothen, M. M. A. Patwary, M. Halappanavar, N. R. Satish, N. Sundaram, and P. Dubey. Designing scalable b-matching algorithms on distributed memory multiprocessors by approximation. In *Proceedings of Supercomputing'16*, pages 66:1–66:11, 2016.

[14] A. Khan, A. Pothen, M. M. A. Patwary, N. R. Satish, N. Sundaram, F. Manne, M. Halappanavar, and P. Dubey. Efficient approximation algorithms for weighted b-matching. *SIAM J. Sci. Comput.*, 38(5):593–619, 2016.

[15] A. M. Khan, D. F. Gleich, A. Pothen, and M. Halappanavar. A multithreaded algorithm for network alignment via approximate matching. In *SC*, page 64, 2012.

[16] V. Knoblauch. Marriage matching: A conjecture of Donald Knuth. Economics Working Papers, http://digitalcommons.uconn.edu/econ_wpapers/200715, 2007.

[17] D. Manlove. *Algorithmics of matching under preferences*. World Scientific, 2013.

[18] F. Manne and R. H. Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *PPAM'08*, volume 4967 of *LNCS*, pages 708–717. Springer, 2008.

[19] F. Manne and M. Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium (IPDPS'14)*, pages 519–528, 2014.

[20] J. Mestre. Greedy in approximation algorithms. In *Algorithms - ESA 2006, 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006, Proceedings*, volume 4168 of *Lecture Notes in Computer Science*, pages 528–539. Springer, 2006.

[21] Md. Naim, F. Manne, M. Halappanavar, A. Tumeo, and J. Langguth. Optimizing approximate weighted matching on nvidia kepler K40. In *22nd IEEE International Conference on High Performance Computing, HiPC 2015, Bengaluru, India, December 16-19, 2015*, pages 105–114, 2015.

[22] U. Naumann and O. Schenk. *Combinatorial Scientific Computing*. CRC Press, 2012.

[23] R. Preis. Linear time 1/2-approximation algorithm for maximum weighted matching in general graphs. In *STACS'99*, volume 1563, pages 259–269. LNCS, Springer, 1999.

[24] M. J. Quinn. *Designing efficient algorithms for parallel computers*. McGraw-Hill, 1987.

[25] J. L. Träff. A parallel approach to the stable marriage problem. In *Proceedings of the Nordic Operations Research Conference (NOAS '97)*, pages 277–287, 1997.

[26] C. White and E. Lu. An improved parallel iterative algorithm for stable matching. Extended Abstract, Companion of IEEE/ACM International Conference for High Performance Computing, Networking, Storage and Analysis (SuperComputing 2013).

[27] L. B. Wilson. An analysis of the marriage matching assignment algorithm. *BIT*, 12:569–575, 1972.