University of Bergen
Department of informatics

Maximum weighted matching on a GPU

Author: Nora Hobæk Hovland
Supervisor: Fredrik Manne

UNIVERSITETET I BERGEN
Det matematisk-naturvitenskapelige fakultet

October, 2020

Abstract

We consider the problem of designing fast algorithms for the maximum weighted matching problem. This is a variant of the maximum matching problem with several real world applications. There exists several approximation algorithms for this problem, both parallel and sequential. The parallel algorithms given so far only have an approximation ratio of $\frac{1}{2}$, except for one recent parallel version of the ROMA algorithm using OpenMP. ROMA is an approximation algorithm that solves the maximum weighted matching problem with an approximation ratio of $\frac{2}{3} - \epsilon$ and a running time of $O(m \log \frac{1}{\epsilon})$. In this thesis we give the first parallel GPU-implementation of the ROMA algorithm suited for complete graphs. Our algorithm achieves an average speedup of 207 on our input graphs when comparing it to the sequential algorithm, while still giving equally good matchings.

Acknowledgements

First of all I would like to thank my supervisor Fredrik Manne for all his help, support and useful advice.

I would also like to thank my family and friends for their support. A special thanks to my algorithms study mates for all the discussions, distractions and coffee breaks.

Lastly, I would like to thank Erlend for always encouraging me and for brightening up every day.

Nora Hobæk Hovland
15 October, 2020

Contents

Chapter 1

Introduction

Suppose you are a teacher and that you want to pair your students into groups of two for a project. Each student is willing to work with any other student, but you know that pairing some students will be better than pairing others. In fact you are a very experienced teacher and can estimate what grade any pair of students is likely to get. Being a utilitarian, you want to assign the groups in such a way that the total sum of grades becomes as high as possible.

This is an example of a maximum weighted matching problem. We can model this by letting the students be vertices of a graph, and taking pairs of students as edges. The estimated grade for each pair is then the weights of the edges of the graph. Furthermore, the graph is a complete graph since any two students can be put in the same group.

In addition to the simple example above, the maximum weighted matching problem has several real world applications for larger problems. Some examples are similarity computations of protein-protein interaction networks[26] and LU-factorization for Gaussian elimination [24] [26].

Many algorithms have been developed for solving the maximum weighted matching problem. The first such algorithm was given by Edmonds in 1965 [9], with a running time of $O(n^2m)$, where n if the number of vertices and m the number of edges of the graph. Today the fastest known algorithm has a running time of $O(n(m+\log n))$ [10]. On large graphs, or if the maximum matching has to be computed repeatedly, this running time can become excessively high.

Due to this many approximation algorithms have been designed, offering a better runtime in exchange for approximate solutions. These algorithms are often algorithmically simple,

and typically give at least $\frac{1}{2}$ of the optimal weight. There exists approximation algorithms promising higher performance as well, for instance the algorithms ROMA and RAMA with a $\frac{2}{3} - \epsilon$ guarantee.

To speed up the computation further, and also to be able to solve for large instances, one can write parallel programs. Parallelization has traditionally been done on CPUs, but in recent years it has become possible to also use GPUs for general computing, giving access to thousands of cores. Several $\frac{1}{2}$-approximation algorithms for the maximum weighted matching problem have been implemented on parallel computers, both on CPUs and GPUs [15]. The aforementioned ROMA algorithm has recently been parallelized on CPUs using OpenMP [3], but is yet to be run on a GPU.

In this thesis we look at different algorithms used to solve the maximum weighted matching problem, and develop a parallel version of the ROMA algorithm for the GPU that works for large, complete graphs. We use the programming platform CUDA to develop our algorithm.

Our parallel ROMA version is compared to the sequential version in terms of running time and solution quality.

The outline of the thesis is as follows: In Chapter 2 we first give an introduction to parallel programming, what it is and why one should do it. In this chapter we also introduce GPUs, the programming platform CUDA, and important concepts of CUDA programming. In Chapter 3 we give an introduction to the maximum weighted matching problem, and algorithms for solving it, with extra focus on the ROMA-algorithm. Then in Chapter 4 we give a description of our parallel algorithm. Chapter 5 contains experiments and results of running our parallel algorithm, as well as the sequential ROMA algorithm, on several graphs. Finally, in Chapter 6 we conclude and discuss possible ways to extend this work.

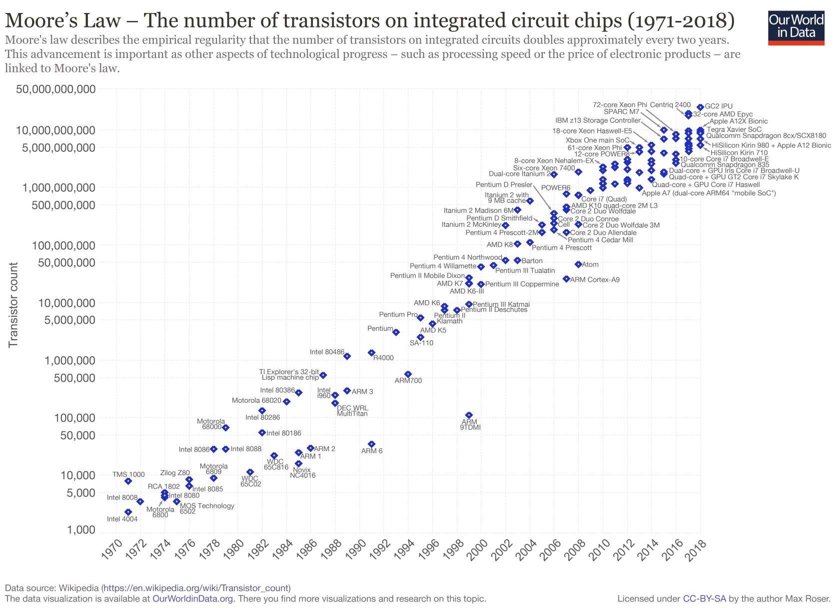
Chapter 2

Parallel Programming on GPUs using CUDA

2.1 Why parallel programming?

In 1965 Gordon Moore predicted that the number of transistors in a dense integrated circuit would double every year [17]. He moderated his prediction in 1975 to say that there would be a doubling every second year instead [18]. This historical trend has become known as *Moore's law* and it has been a very accurate prediction so far.

Figure 2.1: Moore's law [28]



The decrease in the size of transistors has led to increased speed of processors, and the clock rate of computers used to follow the trend of Moore's law. Software developers and users could simply wait for the next generation of microprocessors in order to obtain increased performance for a program, and larger and larger problems could be solved.

Since 2002 however, single-processor performance improvement has slowed down to about 20% per year [22]. As the speed of transistors increases, their power consumption does too. Most of the power is dissipated as heat, and when integrated circuits get too hot, they become unreliable. Thus the increase in speed of integrated circuits will eventually be limited by the transistor's ability to dissipate heat.

Today's situation is that the processor speed is no longer increasing as fast as we would like. The transistor size is still going down, but at some point it will not be physically possible to build smaller transistors. To benefit from the increase in transistor density one can put multiple processors inside the same computer. Instead of only focusing on building faster processors, several processors can be used at the same time and we can get faster computations in this way. The result is called multicore processors, and they are found in most computers today. For these multicore processors to be useful, in most cases the code running on them need to be parallel. As an example, say one wants to speed up a computer game. Then it does not help to only run many instances of the game. If one wants to benefit from the increased number of cores, the game must contain tasks that can be split into smaller tasks which can be done simultaneously.

Using parallel systems we get solutions faster and we have access to more memory. Thus we can solve larger and more complex problems. As the computational power increases, the number of problems we can consider solving does too. Some examples of such problems are climate modeling, protein folding, drug discovery, energy research and data analysis [22].

To use GPUs for general computing has become popular, as one has access to a much larger number of threads than on most CPUs. GPUs have for instance become specially important in deep learning, speeding up computations involving neural networks.

2.2 Parallel computing

Parallel computing can be defined as a form of computation where many calculations are done simultaneously. The idea is that large problems can often be divided into smaller ones, and these smaller problems can then be solved concurrently. One can

look at parallel computing from both the hardware and software aspect. The computer architecture (hardware aspect) focuses on supporting parallelism at an architectural level, while parallel programming (software aspect) focuses on solving the problem concurrently by using the computer architecture.

In a sequential program, the problem is divided into a series of calculations, where each calculation performs a specified task. Two tasks can either be dependent or independent of each other. If a program for instance takes input from the user before processing the input, the task of processing the input is depending on the task of receiving the input. The input must be given before one can do calculations on it, and the two tasks are dependent. If the input is two vectors that should be added together, pairs of numbers of these vectors can be added at the same time. The adding of the vectors is an example of a task that could easily be done in parallel because the sums of the different entries of the vectors are independent of each other.

Writing parallel programs one has to find the tasks that are dependent and independent of each other to be able to know which tasks can be done parallel and which tasks that must stay sequential. Some problems are *embarrassingly parallel*, meaning one can parallelize them with little to no effort, because there is little or no dependence. The vector addition is an example of such a problem.

Parallelism can be divided into two types: *task parallelism* and *data parallelism*. Task parallelism arises when many tasks or functions can be done independently in parallel. Then the different tasks are distributed across multiple cores/processors. In data parallelism many data items can be processed at the same time, and the data is distributed across multiple cores. The cores typically performs the same operations on each data element. The vector addition is an example of this. Two and two vector cells that should be added together can be taken care of by the different cores.

Computer Architecture

Computers can have different architectures and different ways to handle instructions and data flow through cores. One widely used classification scheme in parallel computing is *Flynn's taxonomy* [5] with the following classifications:

- *Single Instruction Single Data* (SISD) refers to the traditional case, a serial architecture. There is one core in the computer executing one instruction stream at the time, and operations are performed on one data stream.

- *Single Instruction Multiple Data* (SIMD) refers to a type of parallel architecture where multiple cores executes the same instruction, each on different data.

- *Multiple Instruction Single Data* (MISD) refers to a architecture where each core operate on the same data, but with different instructions.

- *Multiple Instruction Multiple Data* (MIMD) refers to parallel architectures where each core operate on separate data with separate instructions.

Computer architectures can also be further divided by how their memory is organized, generally classified into two types: *multi-node with distributed memory* and *multiprocessor with shared memory*.

Multiprocessor/multicore architectures with shared memory have several processors connected to the same memory, either physically or by sharing a low-latency link. Most of the processors today are of this type.

Multi-node systems with distributed memory, typically referred to as *clusters*, are constructed from many processors connected by a network. Each processor, usually a multicore processor, has its own local memory, and the different processors can communicate over the network.

2.3 What is a GPU?

Graphics processing units, GPUs, were initially specialized hardware for processing computer graphics. In later years it has become common to use GPUs for computations in many applications that traditionally were handled by the central processing unit, the CPU.

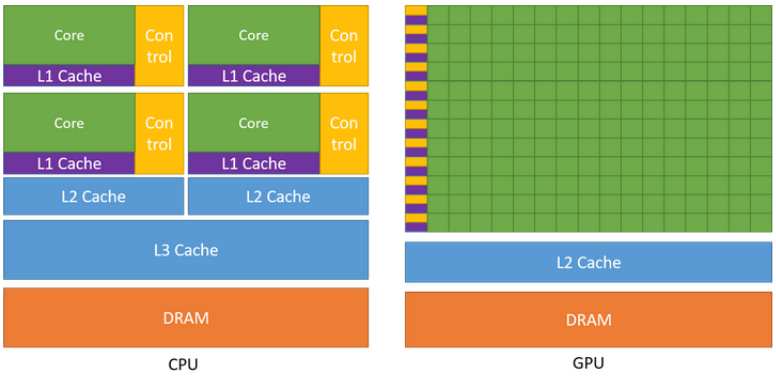


Figure 2.2: CPU vs GPU [20]

CPUs vs GPUs

CPUs are used for general purpose processing, with focus on flow control, caching and optimization. They are responsible for heavy tasks like for instance running operating systems such as Windows and Linux. A GPU on the other hand, is specialized and focuses on compute tasks instead of flow control. While a CPU typically has one thread on each CPU core, a GPU can have thousands of threads that can run simultaneously.

The threads of CPUs are heavy weight and threads on different cores can do different tasks (MIMD), while on a GPU the threads are light weight and works best on data parallel problems when they all do the same task (SIMD). Switching between threads on a CPU is slower than on a GPU. While CPU cores are designed to minimize latency of one or two threads at a time, GPU cores are designed to handle a large number of concurrent threads in order to maximize throughput [5]. Figure 2.2 illustrates the differences between a CPU and a GPU.

GPUs are available in almost all computers today, either integrated with the CPU or as a stand alone unit.

Heterogeneous Computing

Using one or multiple processors of the same architecture to execute an application is called *homogeneous computing*. An example of homogeneous computing is to use multiple CPUs.
In *heterogeneous computing* different architectures are used together to execute an application. Then each task can be handled by the architecture most suited for the job. An example of this is to use a CPU together with one or several GPUs.

In a CPU-GPU system a CPU is connected to a GPU through a PCI-Express bus. The CPU is called the *host*, and starts up the program. The host transfers data to the GPU, called the *device*, and then the GPU can do computations before transferring the data back to the CPU. This way of combining CPU and GPU computations is made possible with computing platforms such as *CUDA* and *OpenCL*. OpenCL is an open standard maintained by the Khronos Group [12], while CUDA is proprietary and designed by NVIDIA. We use CUDA in this thesis and do not go further into details about OpenCL.

2.4 CUDA

CUDA, Compute Unified Device Architecture, is a parallel computing platform designed by NVIDIA. It was launched in 2006 as the worlds first solution for general computing on GPUs and is supported on all modern NVIDIA GPUs [20]. CUDA can be used as an extension to for instance C or C++, allowing the user to write code for the GPU.

We first give a short introduction to the most important concepts of CUDA and CUDA programming, before describing how we implemented our algorithm using CUDA in Chapter 4.

2.4.1 CUDA memory model

A *thread* is the basic unit of operation in a GPU. The threads of the GPU are grouped into *blocks*. Threads have *local memory*, a unique identifier and execute the same program. Inside a block, the threads have *shared memory* and can synchronize execution. The shared memory inside each block can only be accessed by threads within this block.

Multiple thread blocks makes up a *grid*. Every block can access the *global memory*, and they need to be able to be executed in any order. The global memory is accessible by all threads of the GPU, as well as the host. While the local and shared memory are small (KB) and have fast access, the global memory is large (GB) and slow.

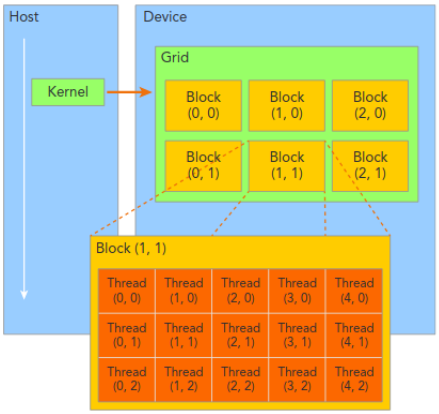


Figure 2.3: Grid of thread blocks [5]

2.4.2 Organizing threads

Thread blocks can be set to be one-, two, or three-dimensional, but from the hardware perspective all threads are one-dimensional. Each thread in a thread block has its own unique thread id, and in one dimension this is stored in `threadIdx.x`. In the same way each thread block has its own block id, `blockIdx.x`.

If a thread block has 1024 threads, the maximum thread id is 1023. Using more than one block and 1024 threads, and wanting a unique global id for each thread, one can calculate the global thread id as follows:

```
int tid = threadIdx.x + blockDim.x * blockIdx.x.
```

Here `blockDim.x` is the number of threads in each block.

2.4.3 GPU architecture

The GPU is built up of *Streaming Multiprocessors* (SMs). Each SM is designed to execute hundreds of threads concurrently, and a GPU typically has multiple SMs. This makes it possible to execute thousands of threads concurrently on a GPU.

When a kernel is executed, the thread blocks of the grid are distributed among available SMs for execution. Several thread blocks can be assigned to the same SM at once.

CUDA uses a *Single Instruction Multiple Thread* (SIMT) architecture, similar to the SIMD architecture. The threads are executed in groups of 32 called *warps*, where every thread within a warp executes the same instruction at the same time. Threads in a warp start together, but it is possible for individual threads to have different behavior. [5].

From the logical view of CUDA programming we have thread blocks, threads and grids, corresponding to the hardware components SMs, CUDA cores and devices respectively. All threads in a thread block run logically in parallel, while physically not all threads can execute at the same time. This means that different threads within a thread block can make progress at different paces. The same holds for thread blocks, since they need to be assigned to a streaming multiprocessor and can be processed in any order. Within the same warp on the other hand, all threads must wait for the other threads to finish.

2.4.4 Warps

Within thread blocks threads with consecutive thread id are grouped into warps. Each warp is made up of 32 threads that are executing the same instruction, each on their own private data. If threads within the same warp take different paths, we get a *warp divergence*.

Warp divergence

Conditional statements in the code can lead to warp divergence. Example:
Since the warps consists of threads with consecutive thread id, conditional statements involving the thread id can cause different execution paths. For instance if we would like to do something for only even numbered threads, half of the threads inside the warp would be doing something else. Then the threads of the warp diverge, and the warp needs to serially execute the two different paths, cutting the parallelism in half with only 16 threads executing at the same time.

Warp divergence can have a significantly negative effect on the performance of a program, increasing with the number of execution paths inside a warp.

2.4.5 Memory access patterns

Memory requests in CUDA are issued per warp. The 32 threads of the warp together performs a single memory access request consisting of each threads' requested address. This is done in one or more device memory transactions, depending on the distribution of the memory addresses within the warp. All memory accesses to global memory go through the L2 cache. Some can also go through the L1 cache, depending on the compile options and architecture of the GPU. If using only the L2 cache, a memory access is done by a 32-byte memory transaction, and if also using the L1 cache the number i 128 bytes.

An *aligned memory access* is when the first address of a device memory transaction is an even multiple of the cache line size (32 bytes for L2 cache or 128 bytes for L1 cache). Misaligned loads causes wasted bandwidth and should be avoided.

Coalesced memory accesses occurs when all threads of a warp ask for data within a contiguous chunk of memory. This is ideal, because otherwise more than one transaction could be necessary to get the data.

To achieve the best performance when reading and writing data, memory accesses should be both aligned and coalesced.

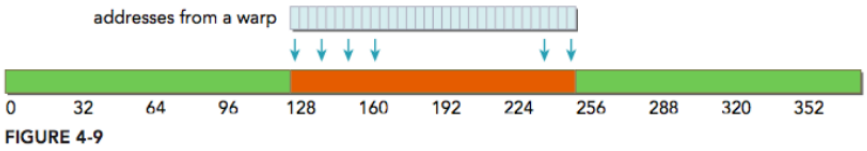


Figure 2.4: Aligned and coalesced memory access [5]

Figure 2.4 shows the ideal case, where a memory access is both aligned and coalesced. Only one transaction is needed and no data is unused.

The worst case scenario, where the requested memory addresses are scattered across global memory, is illustrated in Figure 2.5. The warp only asks for data from 32 addresses, but the addresses can fall across as much as 32 cache lines. This can in the worst case require 32 memory transactions.

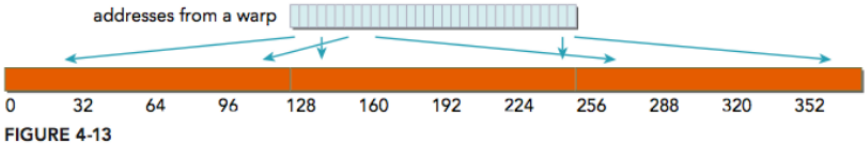


Figure 2.5: Worst case scenario memory access [5]

Since reading and writing to global memory is slow, as few transactions as possible should be used to get the best performance. Thus one should try to write programs with memory accesses that are as aligned and coalesced as possible.

2.4.6 Kernels

A typical CUDA program consists of the following steps:

- allocate GPU memory
- copy data from CPU memory to GPU memory
- invoke a CUDA kernel to do computations on the GPU
- copy back results from the GPU to the CPU

For allocating memory on the device and copying memory back and forth, CUDA has the special functions `cudaMalloc(..)` and `cudaMemcpy(..)`. These functions are similar to the regular `malloc(..)` and `memcpy(..)` functions from C.

The host has to allocate memory on the device and send data before the device can do any calculations.

To execute the parallel code on the GPU, the host calls on a *kernel function*. A kernel function can be written by adding the `__global__` qualifier to a function, telling the compiler that the function should be called from the CPU and executed on the GPU.

Listing 2.1: Kernel example

```
__global__ void helloWorld(){
    printf("Hello World!\n");
}
```

Calling the helloWorld kernel function from the host is done like this:

Listing 2.2: Kernel call

```
helloWorld<<<gridDim, blockDim>>>();
```

Here gridDim is the number of thread blocks and blockDim is the number of threads per block that we want the GPU to use.

The kernel function is where the magic happens, but also where things are most likely to go wrong. Running several threads and blocks at the same time can lead to non-determinism and wrong answers if one does not properly handle dependencies and *race conditions*.

2.4.7 Race conditions

A race condition occurs when the correctness of a program depends on the sequence or timing of a program's threads or processes. If the same value in a program can be changed by different threads, we can get a type of race condition called a *data race*. This happens when two or more threads are accessing the same memory location, where at least one of those accesses is modifying the value. There is no way of knowing which thread will win the race, resulting in non-deterministic behavior. Race conditions can occur in CUDA programs, as well as in other multithreaded programs.

Example

Assume that we have a shared variable of value 0, and that two threads are to increment the value by 1. We want the answer to be 2, but this does not need to be the case. If thread 1 reads 0, increments the value, and writes it back to shared memory before thread 2 reads the value, the answer will be 2. However, if both threads read the value 0 and increments by 1, they will both write 1 back to the shared variable and the answer will be 1. To deal with these kind of situations, one can use *atomic operations*.

2.4.8 Atomic operations

An atomic operation performs uninterruptible operations with no interference from other threads. When a thread has completed an atomic operation on a variable, it is guaranteed that the operation's change on the variable has been completed no matter how many threads are accessing the variable. Thus atomic operations make read-modify-write operations for data shared across threads safe, and can be used to prevent race conditions.

Most atomic functions are binary functions, performing basic mathematical operations on two operands. Atomic functions can be split into the following groups: arithmetic functions, bitwise functions and swap functions. The arithmetic functions are simple operations like addition, subtraction, maximum, minimum, increment and decrement.

If for instance one needs a value to be incremented by multiple threads (like in the example above), CUDA has a function, `atomicInc`, that does this in a safe way, making sure only one thread can read and change the value at the time.

An atomic swap function swaps the value of a memory location with another value, either conditionally or unconditionally. The function `atomicCAS` [20], atomic compare and swap, is one such function.

```
atomicCAS(int* address, int compare, int val)
```

It works by comparing the value of the memory location *address* to the value *compare*, swapping the value at *address* with the input value *val* if the two values compared are equal. The value that was on the *address* location before the conditional swap is returned regardless of whether the swap succeeds or not.

Atomic operations are often necessary for the correctness of a program, but they may also decrease the performance. This is because the use of atomic functions forces a sequential

execution of the given operations. Atomic operations should therefore only be used if absolutely needed.

2.4.9 Synchronization

Dependencies in a program, for instance when filling an array with values before processing the array further, creates a need for synchronization between threads. There are two basic approaches to synchronization in CUDA: *barriers* and *memory fences*. These are special instructions inserted into the code that affects all active threads.

To enable as high compiler optimization as possible, CUDA uses a *weakly ordered memory model* [5]. This means that the order in which a thread of the GPU writes data to different places in memory is not necessarily the same as the order in the source code. The order in which a thread's writes become visible to other threads may not follow the order in which the writes were performed. Similarly the order in which a thread reads data from memory does not have to match the order of the read instructions in the program if the instructions are independent of each other. To force a certain ordering one can insert barriers and memory fences to the code.

Barriers

A barrier makes sure that all calling threads in the block wait for each other to reach the barrier point before proceeding, and also ensures that all global and shared memory accesses made by these threads up to the point of the barrier are visible to every thread of the block. When using barriers one has to be sure that every thread does in fact reach the barrier, otherwise the execution can hang because the threads that reached the barrier are waiting for the rest to arrive. The barriers in CUDA are only possible within the thread blocks, and can be set by calling the function `__syncthreads()`.

Synchronization at grid level

Since synchronization using `__syncthreads()` is only allowed within the thread blocks and not across them, thread blocks can be executed in any order on any SM. This is what makes CUDA programming scalable across an arbitrary number of cores.

Global synchronization between blocks can in many cases be achieved by performing multiple consecutive kernel launches. This is because each kernel launch produces an

implicit global barrier in the sense that before launching a new kernel, the GPU is done computing and the control has been given back to the host.

Memory fence

A memory fence works by making the calling thread stop and wait until all of its modifications to memory up to this point are visible to other threads. One such memory fence is `__threadsfence()`, which works at grid level and stalls the calling thread until its writes to global memory are visible to all threads of the grid. A difference between memory fences and barriers is that a barrier needs to be called by every thread within a block, while memory fences can be called by individual threads.

Volatile qualifier

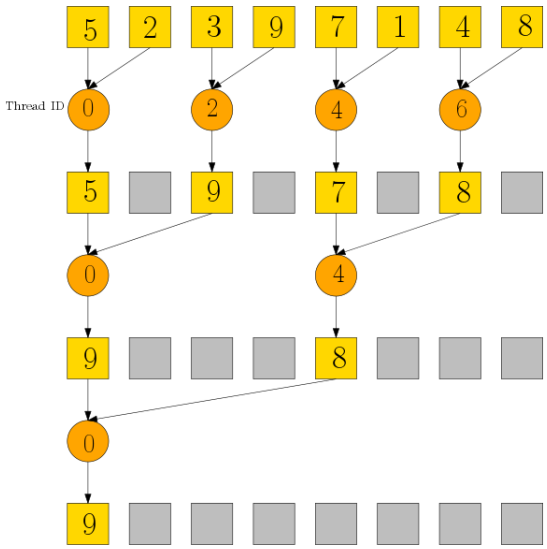
The compiler may optimize by temporally caching data in registers or local memory. If a global or shared variable can be changed or used at any time by any thread, this caching can cause problems when reading and writing to the value. This can be avoided by declaring the variable with the `volatile` qualifier. Then any reference to the variable is compiled to a global/shared memory read or write that skips the cache.

2.4.10 Parallel reductions

On a GPU, as well as on other types of multicore processors, parallel reductions can be used to solve some types of problems faster. As an example consider the problem of finding the maximum element in an array of size n. If solving this problem in a sequential manner one will need n steps in order search through the whole array. However, if we have $\frac{n}{2}$ threads working concurrently in a reduction, the maximum element can be found in $log_2(n)$ steps.

The max-reduction is illustrated in Figure 2.6

Figure 2.6: A reduction finding the largest element in an array



The number of elements in the array is cut in half in each step since each thread always chooses one out of two elements, resulting in $log_2(n)$ steps in total. Other reductions for doing addition, averaging etc can be done in a similar manner. Warp divergence may occur if one assigns pair of elements to only even numbered threads, like in the example of Figure 2.6. In the first step, only half of the threads are active, and in the next step only one fourth are active and so on. There are several strategies to make a reduction more efficient [5] and avoid warp divergence, but we will not go into them.

For reductions in CUDA there exists several libraries, and we will use a library called CUB [7] in this thesis. From CUB we make use of a function called `BlockReduce` to find the maximum element in an array. `BlockReduce` allows for the user to choose which binary function to be used in the reduction, and reduces a shared array of a block using the threads available inside the block. These binary functions can for instance be maximum/minimum-operations or addition.

Chapter 3

Maximum weighted matching

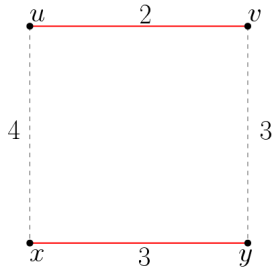
In this chapter we give an introduction to the maximum weighted matching problem. We present different algorithms that can be used for solving it, with extra focus on the ROMA algorithm that we are going to implement on a GPU.

3.1 Definitions

Given a simple undirected graph $G = (V, E)$, a subset of edges $M \subseteq E$ is a *matching* if no two members of M share an endpoint. In other words, M is a subset of pairwise non-adjacent edges. A vertex u is *matched* if there exists an edge $(u, v) \in M$, and this edge is called a *matching* edge. The other endpoint of the matching edge is the *mate* of u.
If the edges of G are weighted, $w : E \to \mathbb{R}^{\geq 0}$, the weight of a matching M, denoted $w(M)$, is the sum of the weights of all the edges in M.
M is said to be a *maximum weight matching* if there is no matching with larger weight. A *maximum-cardinality matching* on the other hand, is a matching with the largest possible number of edges. A matching that matches all vertices of a graph is a *perfect* or *complete* matching. Perfect matchings are always maximum-cardinality matchings, and can only be found in graphs with an even number of vertices.

A path or cycle P in a graph is *alternating* if every other edge is in M. An alternating cycle is always of even length.

Figure 3.1: An alternating cycle, matching edges are red



The symmetric difference of a path or cycle P and a matching M is defined as $M \oplus P = (M \backslash P) \cup (P \backslash M)$, that is, the union of M and P minus the intersection $M \cap P$. An alternating path or cycle P is an *augmentation* if the symmetric difference $M \oplus P$ is also a matching. Figure 3.1 is an augmentation because the symmetric difference, edges (u, x) and (v, y), is a valid matching.

The *gain* of an alternating path or cycle P is defined as $g(P) = w(P \backslash M) - w(P \cap M)$. Thus for the gain of an augmentation to be positive the sum of the weights of the unmatched edges in P must be greater than the sum of the weights of the matched edges. In Figure 3.1 the gain is (4+3)-(2+3)=2. If the gain of a path/cycle P is positive, P is a *weight-increasing path/cycle*.

An augmentation with at most k non-matching edges is called a *k-augmentation*. If a 2-augmentation P is *centered* at v, all edges of $P \backslash M$ are incident to either v or its mate. We denote a maximum-gain 2-augmentation centered at v by $aug(v)$.

Figure 3.2: 2-augmentation centered at vertex u and vertex v, matching edges are red

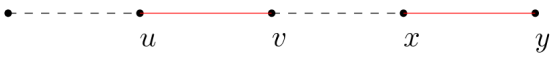


Figure 3.2 shows a 2-augmentation centered at vertex u or vertex v. While Figure 3.1 could be described as centered at any of its four vertices, this does not hold for Figure 3.2 since the edge (x, y) is not adjacent to the leftmost edge.

Now let $G := (V, E)$, with $n := |V|$ vertices and $m := |E|$ edges, be a *complete* graph. Then every pair of vertices is connected by an edge, so the number of edges is $m = \frac{n(n-1)}{2} = O(n^2)$. A maximum-cardinality matching M in a complete graph with an even number of vertices is always a perfect matching, and the size of the matching is $\frac{n}{2}$ since we pair together n vertices. In a complete graph, a maximum weight matching must be a perfect matching if the edge weights are positive and n is even. To find a

maximum-cardinality/perfect matching in a complete graph is trivial, since we can pair together any two vertices, but for the matching to also be the one of highest weight, we need a more elaborate approach.

3.2 Polynomial time algorithms for the maximum weighted matching problem

Several polynomial time algorithms exists for solving the maximum weighted matching problem. The first such algorithm was given by Edmonds [9] in 1965 and has a running time of $O(n^2m)$. This algorithm is called the *Blossom Algorithm*, and was initially an algorithm used to find the maximum matching in a graph. Assigning weights to the graph, the algorithm can be used to solve the maximum weight matching problem as well.

The Blossom Algorithm works by iteratively improving an initial empty matching along augmenting paths, where the key idea is to contract odd-length cycles in the graphs (blossoms) to a single vertex, continuing the search on the contracted graph.

Today the fastest exact algorithm we know for the maximum weighted matching problem has a running time of $O(n(m + n \log n))$ [10] for general graphs. For a complete graph, which is the graph type we focus on in this thesis, this would mean a runtime of $O(n^3)$.

If an algorithm requires $\Theta(n^3)$ operations it could take very long time to execute. For example if $n = 10^6$ and assuming a running time of n^3 would require $(10^6)^3$ operations. A sequential computer running at 5Ghz performs $5 \cdot 10^9$ operations per second, and could spend as much as $\frac{10^{18}}{5 \cdot 10^9} = 2 \cdot 10^8$ s $= 6.3$ years computing the solution.

As the worst case running time for large graphs on the maximum weighted matching problem can be this high, there has been an interest in applying other strategies to speed up the computation. One such strategy, as already described, is parallelization. Another strategy is to use an *approximation algorithm*.

3.3 Approximation algorithms

Approximation algorithms are used when one wants to speed up the running time and is willing to pay the price of getting approximate results. While an exact algorithm computes the optimal value of an optimization problem for all problem instances, an approximation algorithm computes a value that is within some factor α of the optimal value. An algorithm with approximation ratio α is called a α-approximation algorithm. Note that α is the worst case approximation ratio, and for many instances the solution can be much better.

Approximation algorithms are often used on *NP-hard* problems, but there exists many approximation algorithms for problems known to solvable in polynomial time as well. In the example from Section 3.2 with a computer running at 5Ghz one can see that even $O(n^3)$ - algorithms can be too slow on large inputs. This is why we sometimes want to use approximation algorithms on polynomial time problems. Approximation algorithms often have linear or close to linear running time, and in practice compute solutions that are nearly optimal [24].

In addition to speeding up the running time, approximation algorithms have other advantages as well. They are often conceptually simpler than exact algorithms, and easier to implement. Making the implementation easier is important for algorithms that are to be run on parallel computers.

There exists several approximation algorithms for the maximum matching problem. Many of these algorithms only have an approximation ratio of $\frac{1}{2}$, but in return they are often easier to implement and usually faster than the algorithms with better approximation ratio. Since we are going to use a GPU to compute a matching we want to pick an algorithm that is not too complicated to implement.

We give a short overview of some of the linear or close to linear time approximation algorithms for the weighted matching problem.

3.3.1 Greedy

The first approximation algorithm we present is very simple, but still gives reasonably good matchings. The greedy algorithm is a $\frac{1}{2}$-approximation algorithm, and has linearithmic running time [16]. It works by repeatedly adding the current heaviest non-matching edge with free endpoints to the matching, until there are no edges left.

Algorithm 1 Greedy algorithm for approximate weighted matching

Greedy$(G = (V, E), w : E \to \mathbb{R}^{\geq 0})$

1: $M := \emptyset$
2: **while** $E \neq \emptyset$ **do**
3: let e be the remaining edge in E of largest weight
4: add e to M
5: remove e and all edges adjacent to its endpoints from E
return M

The running time is $O(m + \text{sort}(m))$, where $\text{sort}(m)$ is the time it takes to sort m edges. Thus the running time is dominated by the time it takes to sort the edges. It is linear for integer edge weights [16], and $O(m \log(n))$ otherwise. We present the proof of the approximation ratio.

Let M be the matching found by the algorithm and $\hat{M}$ be the maximum weight matching. Each time an edge e is added to the matching M, at most two edges $e_1, e_2 \in \hat{M}$ are removed from the graph (at most one edge for each endpoint of e). Since the edge e is added to the matching we must have that $w(e) \geq w(e_1)$ and $w(e) \geq w(e_2)$. Thus $2w(M) \geq w(\hat{M})$ and it follows that the approximation ratio is $\frac{1}{2}$ [16].

It is also possible to compute a $\frac{1}{2}$-approximation in linear time, which we will see in the next algorithm.

3.3.2 Path Growing Algorithm

The Path Growing Algorithm by Drake and Hougardy[8], PGA, starts at an arbitrary vertex u, and grows a path from u by traversing the heaviest edge incident to u that leads to an unvisited vertex v. It then continues this process from u, and when it reaches a vertex where no edges lead to an unvisited vertex, it stops. This is repeated starting from every unvisited vertex.

For each of the resulting paths it then chooses the heavier set of the odd and even numbered edges to add to the matching. Since the search for a path is started only once for each vertex, and every edge is only considered twice (once from each endpoint), the running time is $O(n + m)$, which is the same running time as of the graph search algorithms DFS (Depth First Search) and BFS (Breath First Search). The approximation ratio is $\frac{1}{2}$ [8]. PGA' is an improved version where one instead of choosing between the odd and even numbered edges in a path, computes the optimal matching for each path.

This can be done by dynamic programming [16], but the improvement does not affect the asymptotic runtime or the approximation ratio.

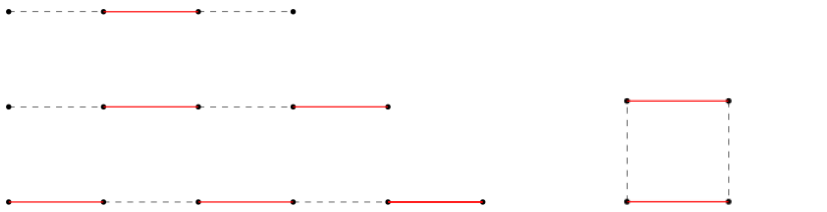
3.3.3 GPA

The Global Paths Algorithm, GPA, integrates the Greedy Algorithm and PGA′, and was given by Maue and Sanders [16]. It works by generating a maximal weight set of paths and even length cycles, and then calculating a maximum weight matching for each of them by dynamic programming. The paths and cycles are created by adding applicable edges in descending order of weight. An applicable edge is an edge that connects two endpoints of different paths or the two endpoints of an odd length path. GPA has the same running time as Greedy and also has an approximation ratio of $\frac{1}{2}$. It has shown to be well suited as a pre-processing algorithm, specially when followed by the algorithm ROMA presented in 3.3.5.

3.3.4 RAMA

The Random Augmentation Matching Algorithm, RAMA, by Pettie and Sanders [23] is a randomized approximation algorithm used to find an approximate maximum weight matching in a graph. It works by repeatedly choosing a random vertex and augmenting the current matching with the highest-gain 2-augmentation centered at that vertex. A graph can contain the following 2-augmentations:

Figure 3.3: 2-augmentations



(a) 2-augmenting paths (b) 2-augmenting cycle

The aug()-function finds the 2-augmentations centered at v, and if there is a 2-augmentation with positive gain, the matching is is updated by taking the symmetric difference of the old matching and the 2-augmentation with highest gain.

This is iterated k times, and if we put $k := \frac{1}{\epsilon} n \log \frac{1}{\epsilon}$, the algorithm has expected runtime of $O(m \log \frac{1}{\epsilon})$ and an expected performance ratio of $\frac{2}{3} - \epsilon$. [23]

Algorithm 2 RAMA - Random Augmentation Matching Algorithm

RAMA($G = (V, E)$, $w : E \to \mathbb{R}^{\geq 0}$, int k)

1: $M = \emptyset$ (or initialize M with any matching)
2: **for** $i = 1$ **to** k **do**
3: randomly choose $v \in V$
4: $M = M \oplus aug(v)$
return M

3.3.5 ROMA

The Random Order Augmentation Matching Algorithm [16], ROMA, is a variant of RAMA. It has l phases, and in each phase the algorithm selects every vertex in random order, before augmenting in the same manner as RAMA. By setting the number of phases to $l = \frac{k}{n}$, where k is the number of iterations in RAMA, one obtains the same running time and performance ratio as given by RAMA [16].

Algorithm 3 ROMA-Random Order Augmentation Matching Algorithm

ROMA($G = (V, E)$, $w : E \to \mathbb{R}^{\geq 0}$, int l)

1: $M = \emptyset$ (or initialize M with any matching)
2: **for** $i = 1$ **to** l **do**
3: **for each** vertex $v \in V$ in random order **do**
4: $M = M \oplus aug(v)$
return M

ROMA and RAMA are very similar, but ROMA has some advantages. ROMA can be stopped after a phase where no vertices has been augmented. This is not possible in RAMA since we have no guarantee that every vertex has been processed after n iterations, meaning there can still be vertices that could be augmented. Note that when we say that we *augment* a vertex, we mean both finding the 2-augmentation of highest gain and taking the symmetric difference.

ROMA has shown to be more effective than RAMA in practice, as well as being a good algorithm for post-processing of matchings [16]). It computes a higher weight matching when initialized with a matching given by GPA.

3.4 Parallel algorithms for the weighted matching problem

Several parallel $\frac{1}{2}$-approximation algorithms for the maximum weighted matching problem exist, implemented on both CPUs and GPUs. Many of these algorithms are based on finding locally dominant edges. An edge is dominating if it is heavier than all of its incident edges. Manne and Bisseling gave one such algorithm [14] for a distributed memory computer in 2007, and later Halappanavar et al. ported it to shared memory computers [13]. In addition, Birn et al. has given parallel versions of the Localmax algorithm, for both distributed memory and GPUs [4]. The Suitor algorithm by Manne and Halapannavar from 2014 [15] gives the same matching as Greedy. It is well suited for parallelization, and scales better than previous multithreaded algorithms.

Why it is of interest to design a parallel ROMA algorithm for the GPU

We have so far listed some of the parallel $\frac{1}{2}$-approximation algorithms for the maximum weighted matching problem. There are many to choose from, both for execution on a CPU or a GPU. However, for approximation algorithms for the weighted matching problem with higher guarantee than $\frac{1}{2}$ of the optimal matching, none has so far been implemented on a GPU.

ROMA has recently been parallelized on CPUs using OpenMP [3], the results showing that it lends itself well to parallelization. On a GPU one has access to more threads. Thus it would be of interest to see how the ROMA algorithm performs on a GPU.

Out of ROMA and RAMA, two very similar algorithms, ROMA is the most natural choice for the following reasons:

- ROMA can be stopped at convergence.
- ROMA has shown to be better in practice, and works well when given an initial matching computed by for instance GPA.
- The fact that every vertex is processed in each phase of ROMA makes it easier to assign vertices to threads when parallelizing.
- ROMA has been parallelized before, so we know that it can be done.

Why complete graphs?

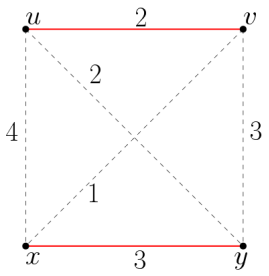
In [2] Azad et al. present a distributed memory algorithm for computing a heavy weight perfect matching on bipartite graphs. Their algorithm is based on ROMA and works by finding weight-increasing 4-cycles. This approach can be used on complete graphs as well. One can use *4-cycle augmentation*, which means one only has to find 4-cycles of the type of Figure 3.3b when looking for 2-augmentations. Since a complete graph always has a perfect matching, and all vertices are connected by an edge, every pair of matching edges participates in 4-cycles. The 4-cycle augmentation procedure is less complicated to implement than a general procedure that computes all 2-augmentations. This is an advantage when moving the problem over to the GPU. Complete weighted graphs are also very straight forward to represent on a GPU, as they can be given as adjacency matrices where the entries are the weights of the edges.

Next we explain how the augmentation can be done on complete graphs.

3.5 4-cycle augmentation for RAMA and ROMA on complete graphs

For complete graphs, if we start with a perfect matching, the augmentation step in ROMA and RAMA can be reduced to finding cycles of 4 vertices u, v, x and y, where the vertices makes up two matching edges, say (u, v) and (x, y), and then calculating the 2-augmentation resulting from *flipping* the edges. By *flip* we mean changing the two matching edges $\{(u, v), (x, y)\}$ to either $\{(u, x), (v, y)\}$ or $\{(u, y), (v, x)\}$. If starting with a non-perfect matching, any two unmatched vertices can be matched to construct a perfect matching since the graph is complete.

Figure 3.4: A 4-cycle example



In Figure 3.4 the weight of the current matching (u, v) and (x, y) is 5. If we change the matching to instead contain the edges (u, x) and (v, y) we get a matching of weight 7. Thus we get a positive gain of 2 if we flip the edges of the matching in this way. Doing this on a 4-cycle in a complete graph will not affect the matching of the rest of the graph in any way, since the vertices involved in the old matching of the 4-cycle are the same vertices as in the new matching. Thus changing which two edges to put in M still gives a perfect matching of the whole graph, with the same number of matching edges ($\frac{n}{2}$ if n is even).

Figure 3.5: Two different ways to flip Figure 3.4

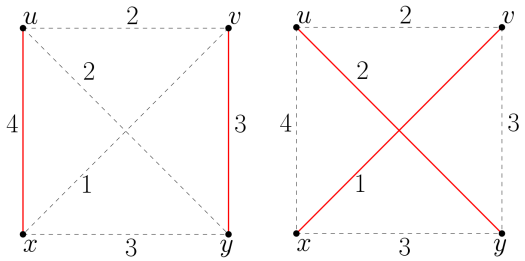


Figure 3.5 illustrates the two options for flipping the matched edges in Figure 3.4. Two

matched edges (u, v) and (x, y) always have two different ways to flip: a "vertical" flip resulting in $\{(u, x), (v, y)\}$, and a "diagonal" flip giving $\{(u, y), (v, x)\}$. In Figure 3.5 the first flip is vertical and the second is diagonal. The gains in this example are 4+3-(2+3)=2 for the horizontal flip, and 2+1-(2+3)=-2 for the vertical flip. Thus the best flip option is $\{(u, x), (v, y)\}$.

Working with complete graphs with positive weights, we know that the best matching we can get must have $\frac{n}{2}$ edges, meaning that every vertex is matched. The maximum gain 2-augmentation centered at a vertex u can therefore be found by looking at all alternating 4-cycles the matching edge $(u, mate(u))$ participates in. This makes up $(n-1)$ 4-cycles since each pair of matching edges have two different flip options (two different 4-cycles) and vertex u has $\frac{n-1}{2}$ neighboring matching edges.

The augmentation is done as follows on a complete graph:

- We start with an arbitrary complete start matching M.
- Let u be the vertex picked at random in an iteration of RAMA/ROMA, and v be the mate of u.
- To augment u we iterate through all matching edges $(x, mate(x))$, where $x \in V \backslash \{u, v\}$. We only process x if $x < mate(x)$ to avoid testing each edge twice.
- In each iteration we calculate the sum of the weights of edge (u, v) and $(x, mate(x))$, as well as the sum of the weights of the edges we get if we flip them. Consider both possible flips like in Figure 3.4.
- If flipping gives a positive gain, we can do a flip and improve the total matching. However, we want to flip with the best edge, that is, the one that gives highest gain, so we will save the temporary best edge, and after going through all the vertices/matching edges of the graph and calculating the gains, we know which edge to flip with, if any.

We represent the matching as an integer array M where for each vertex u, $mate(u) = M[u]$. To keep track of the best flip found so far we use a tuple (a, b), where a is u's new mate, b is $M[u]$'s new mate.

Algorithm 4 Flip procedure

flip(int array M, int tuple (a, b), int u)

1: $mateU = M[u]$
2: $M[u] = a$
3: $M[a] = u$
4: $M[mateU] = b$
5: $M[b] = mateU$
6: **return** M

Algorithm 5 4-cycle augmentation

aug($G = (V, E, w : E \to \mathbb{R}^{\geq 0})$, int array M, int u)

1: $gain = 0$
2: int tuple (a, b)
3: **for** $x := 0$ **to** V **do**
4: **if** $x \neq u$ **and** $x \neq M[u]$, **and** $x < M[x]$ **then**
5: $currentWeight = w[u, M[u]] + w[x, M[x]]$
6: $w_1 = w[u, x] + w[M[u], M[x]]$
7: $w_2 = w[u, M[x]] + w[M[u], x]$
8: **if** $w_1 > w_2$ **and** w_1 - $currentWeight > gain$ **then**
9: $gain = w_1$ - $currentWeight$
10: $(a, b) = (x, M[x])$
11: **else if** w_2 - $currentWeight > gain$ **then**
12: $gain = w_2$ - $currentWeight$
13: $(a, b) = (M[x], x)$
14: **if** $gain > 0$ **then**
15: $flip(M, (a, b), u)$
return M

In Algorithm 5, the input is a complete and weighted undirected graph, a matching array M, and a vertex u. For a matching edge $(x, M[x])$, we are only interested in processing it if it is a neighbor edge (no common endpoints with our current edge (u, v)), and from only *one* of its endpoints to avoid looking at it twice. Therefore we in line 4 check that each edge is within these requirements before processing it further. The scalar values w_1 and w_2 (lines 6 and 7) are the weights of the two different flip options, and if one of them is greater than the weight of our current matching edges (lines 8 and 11), we save this flip option as the current best one in (a, b).

If both w_1 and w_2 are greater than the current weight, the greatest of them is chosen. After iterating through all the matching neighboring edges of (u, v), we do the edge flip in line 15 if the gain of the best option is positive.

Note that in this augmentation procedure, the changing of the matching (the flip) happens

inside the procedure, instead of taking the symmetric difference of a returned augmentation like in Algorithm 3.3.5.

This simple way of augmenting vertices in a single for-loop will help us when we now are going to make the ROMA algorithm parallel.

Chapter 4

Parallel algorithms for weighted matching on complete graphs

4.1 Parallelizing ROMA

When parallelizing an algorithm, one first has to look at the different parts of the algorithm to figure out which of the operations that can and should be done in parallel. In some situations, for instance the before mentioned vector addition, everything can be done concurrently as there are no dependencies between the sums of the different entries. This is often not the case. To parallelize ROMA we need to single out the parts that can be done in parallel.

The outer for-loop going through the phases of ROMA is difficult to parallelize, because we want every vertex to be processed before starting a new phase. In addition, it will not be possible to stop at convergence if several phases run concurrently. We therefore let the outer for-loop stay sequential.

The for-loop that selects random vertices and augments them can be done parallel by removing the for-loop and doing the augmentations of many/all vertices at the same time. Here we need to be careful so that the matchings are valid, because we can get conflicts when the matching array is updated.

Another thing that could be done is for each vertex being augmented, that the augmentation procedure in itself is done in parallel by computing several 4-cycles at the same time.

4.1.1 Augmenting vertices in parallel

In our first parallel version of ROMA we remove the vertex-selecting for-loop by letting each vertex be taken care of by a thread of the GPU, and the threads do the augmentations concurrently. Only removing the for-loop in this way will most likely not result in a valid matching, because several threads can try to change the matching of the same vertex at the same time. To avoid invalid matchings, one needs to make sure that when one thread is doing changes on the matching array, no other thread are using the vertices whose matchings are being updated.

There are several ways to try and fix the problem of invalid matchings. One way could be to just let all threads do their best flips, and then try to change the edges that are making the matching invalid afterwards. This could maybe be an option on a sparse graph with few overlapping 2-augmentations, but for a complete graph there can be many collisions. Thus the matching can be invalid for many vertices, meaning the post-processing could spend unreasonably long time fixing the errors. How to fix an invalid matching is not obvious either, so we discard this approach.

Another way could be that each edge in M finds an edge to flip with, and then from this set of 4-cycles one chooses a subset that do not conflict. This would have to be done by some tie breaking method, for example gain or at random, but this method would probably need a good deal of synchronization and spend more time than desired.

A third way to avoid invalid matchings is to use *locks*. This can be done by making each thread lock the edges it wants to flip, and only allow a thread to flip if the two edges involved were locked by only this thread. Al-Herz and Pothen uses locks in a parallel augmentation-based algorithm for the maximum vertex-weighted matching problem [1], so this can be a good approach for our problem as well. We therefore choose to use a locking method in our parallel algorithm.

Now we look at how we parallelize the ROMA-phase for-loop. On a GPU we can calculate the unique global thread id of a thread in a grid, as seen in Section 2.4.2. Having unique ids for all the threads we can then assign vertices to threads by representing each vertex by a thread id. Vertex v_0 is taken care of by thread t_0 and so on. A phase of ROMA can then be done by calling a kernel with $n = |V(G)|$ threads, where the flip in the augmentation procedure involves a locking mechanism to make sure we have valid matchings. Since n in most cases is greater than the number of threads that can run simultaneously on the GPU, in practice a selection of the vertices will be augmented in parallel and not all of them at the same time.

Algorithm 6 A parallel phase of ROMA

ROMA-phase($G = (V, E, w : E \to \mathbb{R}^{\geq 0})$, int array M, int u, int array $locks$)

```
1: int u = threadIdx.x + blockDim.x · blockIdx.x
2: if u < V and u < M[u] then
3:     augLocks(G, M, u, M[u], locks)
   return M
```

Algorithm 6 is a parallel version of a ROMA-phase, and is run as a kernel on the GPU. The integer u is the vertex to be augmented. Only processing vertices where $u < M[u]$ ensures that every matching edge is augmented once per phase. In the original ROMA algorithm, Algorithm 3.3.5, each vertex is augmented in a phase. If we in our parallel version instead augment each matching edge, we avoid two threads augmenting the same edge from different endpoints at the same time, and we can avoid some collisions in this way.

The integer array $locks$, and u's mate $M[u]$ are given as input to the augment procedure, $augLocks$, which is a modified version of the 4-cycle augmentation from Algorithm 5. The other difference between $augLocks$ and Algorithm 5, besides the input parameters, is that the flip procedure has been changed. This is how we will ensure to get a valid matching. We present this new flip procedure in Algorithm 7.

Calling a kernel with $n = |V(G)|$ threads will make sure that every matching edge is processed, but only $\frac{n}{2}$ threads will be able to do augmentations per phase since there are only $\frac{n}{2}$ matching edges between n vertices. This way of assigning vertices/edges to the threads is simple, but can lead to many threads within a warp being idle, and can affect the efficiency of the program. We use another approach in a later version of the algorithm, Algorithm 8, that makes better use of the resources, and we do some profiling in Chapter 5 to see the effect.

Locks

To represent the locks we have an integer array of length n, where the value at index u tells if vertex u is locked. If the value is -1, this means that the corresponding vertex is unlocked. Any other value means that some thread has locked the vertex.

Locking a vertex u means looking at the value locks[u], and if the value is -1 change it to some other value to signal that it is locked, for instance the vertex id u. It is enough to lock an edge by only locking one of its endpoints, instead of locking both. This follow since we represent each edge by its smallest vertex id. When calling the 4-cycle augmentation

procedure, only vertices where $u < M[u]$ are being processed. In the same way inside the augment procedure, when looking through the neighbors to find the best candidate for flipping, we only process vertices that have smaller vertex id than their mate.

If a thread wants to flip the edge (u, v) with (x, y), it will need both vertex x and u to be unlocked (assuming $u < v$ and $x < y$), and then if the thread manages to lock both of the vertices, and none of the two matching edges has been changed in the meantime, it is safe to perform the flip.

For the locking process to be safe we use `atomicCAS` (Section2.4.8). By safe we mean that if two threads try to lock vertex u at the same time, only one of them will succeed. The return value of `atomicCAS` is -1 for the thread that managed to lock the vertex.

The order in which a thread t locks two vertices u and x can be important, because if we are not careful two edges that should flip with each other could end up not being flipped at all. This could happen if the vertices representing the two edges are locked by two different threads, where both threads are supposed to flip the two edges. To solve this problem we choose for each thread to always start by trying to lock $min(u, x)$. We will explain in more detail why the order is important and why locking the smallest edge works in the next section. We will also prove that as long as there are still vertices to be augmented, there will always be progress when using locks in this way.

If the value returned from `atomicCAS` is not -1, another thread has already locked the vertex, and the thread t should give up its attempt to augment the vertex. On the other hand, if the return value is -1, t succeeded, and will try to mark $max(u, x)$ as well.

If this also succeeds, t gets to do the edge flip, given that the mates of u and x have not been changed in the meantime. This can happen if another thread flips its edge with the edge of vertex u or vertex x after t has computed which egde to flip with but before it starts locking the vertices. To check if the mate of u is still the same, the original mate is given as input to the augment procedure. After flipping the edges, t changes its locks to -1 again so that the other threads are free to use these edges.
If locking the second vertex fails, t stops its attempt to augment and removes the first lock so that other threads can use this edge.

Algorithm 7 Flip procedure using locks

atomicFlip(volatile int array M, int tuple (a, b), int u, int $mateU$, int array $locks$)

```
1: int x = min(a, b)
2: int result1 = atomicCAS( &locks[min(u, x)], -1, u )
3: if result1 = -1 then
4:     result2 = atomicCAS( &locks[max(u, x)], -1, u )
5:     if result2 = -1 then
6:         if mateU = M[u] and M[x] = max(a, b) then
7:             flip(M, (a, b), u)
8:             __threadfence();
9:         locks[max(u, x)] = -1
10:    locks[min(u, x)] = -1
   return M
```

In Algorithm 7 the input is the matching array M, a tuple of integers (a, b), the vertex u and its mate $mateU$, where $u < mateU$, and the integer array $locks$. The tuple (a, b) is the matching edge to flip with, and keeps track of which vertices should become the new mates of u and $mateU$ respectively. The value at $locks[v]$ tells if vertex v is locked. The attempts of locking a vertex is done with `atomicCAS`, where the arguments are the pointer to the memory location of the vertex to be locked, the value -1 to compare with the value at this memory location, and the new value to swap to if the vertex is unlocked. If this new value is $min(u, x)$ or $max(u, x)$ does not matter, as the active thread will only be able to proceed if the return value of `atomicCAS` is -1, meaning the vertex was unlocked and that the thread succeeded in locking it. If locking both edges succeeds and the mates of u and x are unchanged, the flip procedure from Algorithm 4 is performed.

Since we are potentially using many different blocks on a GPU, for instance if we have one block for each vertex of the graph, there can be many updates on the global memory at the same time. To ensure that threads of different blocks have the latest values of the matching array M, we make the matching array volatile and set a threadfence (see Section 2.4.9) after changing the matching so that the updates are visible to all calling threads after the fence. If not using a threadfence after the flip, there could be situations where the locks are released before the changes on the global matching array are visible to other threads. This could then lead to some threads being able to lock their vertices, but still reading old values of the matching array when checking if the matching has been changed. Thus some threads could possibly perform illegal flips, and the resulting matching would not be valid. The threadfence fixes this issue by not allowing a thread to open its locks before its changes to the global matching array are done.

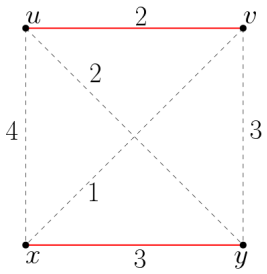
The resulting augmentation procedure, *augLocks*, is a modified version of the 4-cycle

augmentation (Algorithm 5) where the old flip-procedure is replaced with *atomicFlip* (Algorithm 7), and where *mateU* and the lock-array are included as input arguments. Because *augLocks* is otherwise identical to Algorithm 5, we do not include the code here.

Safeness of the locks

The idea behind locking an edge is to make sure only one thread is making changes on the matching of the edge at any given time. Both of the edges being flipped must be locked by the same thread for this to be safe, so we need to first try to lock one, and then if we succeed, lock the other. Which one of the edges we lock first may not seem important, but can become a problem if we are not careful.

As an example consider the 4-cycle example from Figure 3.4:



The edges (u, v) and (x, y) should flip and the new matched edges should become (u, x) and (v, y). Since several threads are computing which edges to flip at the same time, we could be in a situation where thread t_u is responsible for vertex u, where $u < v$, and thread t_x is responsible for vertex x, where $x < y$. Thread t_u does its computation and finds edge (x, y) as the best edge to flip with, so it should lock vertex x in addition to its own vertex. At the same time thread t_x is done with its computation, and wants to lock vertex u. If thread t_u locks vertex x, and before it gets the chance to lock its own vertex, u, thread t_x jumps in and locks vertex u instead, none of the threads managed to lock both edges. Hence none of them can perform a flip.

This problem can be avoided if a thread always starts with trying to lock the vertex of lowest id among its own vertex and the vertex representing the edge to flip with. In the situation above, if applying this rule, both thread t_u and thread t_x tries to lock $min(u, x)$, and only one of them succeeds. The thread not succeeding will give up, and the winning thread can now try to lock the other vertex, vertex $max(u, x)$. A success of locking the second vertex means that the thread can safely flip the edges without worrying about other threads interfering.

Let t_1 be the thread succeeding in locking the first vertex. If our small example graph is a subgraph of a larger, complete graph, a third thread could possibly come in and try to lock the second vertex, and spoil t_1's chance of performing the flip. This is hard to avoid, since all vertices are augmented in parallel and therefore if many threads wants to lock the same vertex at the same time, one cannot predict which one will succeed. If another thread locks the second vertex before t_1 manages to, t_1 simply has to give up its augmentation.

We will now show that in a parallel phase of ROMA (Algorithm 6) at least one thread will always be able to lock the vertices needed to augment the matching. Thus there will always be progress as long as there exists a 4-cycle with positive gain in the graph.

Lemma 4.1.1. *If there exists at least one 4-cycle of positive gain at the start of the ROMA phase in Algorithm 6, at least one thread will be able to flip, and the matching M will be updated.*

Proof. Assume that one thread t_k is assigned to each vertex v_k in the matching, $0 \leq k < |M|$, and that at least one thread has found another edge to flip with. For a thread to be able to perform a flip, it must have locked both of the matching edges involved. We will now show that there always exists at least one thread that manages to do this as long as there are vertices to be augmented in G.

If several threads tries to lock the same vertex at the same time, we know that `atomicCAS` will always succeed for one of the threads. To have progress we only need one of the succeeding threads to be able to lock their second vertex as well.

We look at a the set of all vertices that has been locked in a phase, and within this set we name the vertex with highest id v_x. The thread that locked v_x may have locked it as its first or second vertex. If it is as the second vertex we are done and can flip the two locked edges, given that no other flipping involving one of the edges has happened in the meantime. Suppose v_x is locked as the thread's first vertex instead. Then there must be another vertex v_y that this thread wants to flip with, and $v_y > v_x$. Since v_y is unlocked, the thread that locked v_x can also lock v_y. This is a contradiction since v_x was the vertex of highest id that was locked.

This means that as long there exists a 4-cycle of positive gain, at least one thread manages to lock the vertices needed to flip. If the matching has not changed during the thread's computation, the thread can safely flip and we have progress. If there has been a change in the matching array after the thread found which edge to flip with, but before it locked

the vertices involved, the thread's flip choice is not valid and it will not be able to change the matching. However, this means that there has already been another thread that managed to flip and thus there has been progress.

□

4.1.2 Parallel augmentation

Now that the for-loop that selects and augments the vertices has been parallelized, we discuss how the augmentation itself can be done in parallel. We would like to remove the for-loop that iterates over the matching edges to find the 4-cycle of highest gain.

On a GPU we have access to a large number of blocks, and within each block a set of threads. If we let each block represent a vertex to be augmented, we could use the threads within each block to find the best edge to flip with. Each block can be responsible for a vertex, augmenting them in parallel, while the augmentation inside each block is also done in parallel by the threads of the block. Using n blocks, where n is the number of vertices of the input graph, will make sure that all matching edges are augmented per phase.

Say block u augments the vertex u. To find 4-cycles in Algorithm 5, the for-loop iterates over all matching edges, first computing the 4-cycles of edge $(u, mateU)$ and another edge e_0, then the 4-cycles of edge $(u, mateU)$ and edge e_1 and so forth. To replace the for-loop inside each block, we can let each matching edge be taken care of by a thread. Then thread t_0 can compute the 4-cycles of edge $(u, mateU)$ and e_0, while thread t_1 finds the 4-cycles of edge $(u, mateU)$ and e_1.

For this idea to work independently of the number of threads per block and the number of matching edges, we do the following:
Each thread in a block calculates the gain of flipping the matching edge of the vertex with id equal to $blockIdx.x$ with the matching edge of the vertex with id equal to $threadIdx.x + k \cdot blockDim.x$, for some integer $k \geq 0$. Here $blockDim.x$ is the number of threads we use inside each block, and k is an integer such that the calculated vertex id does not exceed number of vertices, n. What the number of threads per block should be is discussed in Chapter 5.

If we have enough threads per block, we can let thread t_0 have matching edge e_0, thread t_1 have matching edge e_1 etc. Otherwise, if $n > blockDim.x$, we use a while loop that increments the vertex id with $blockDim.x$ each round so that every matching edge is eventually taken care of. For instance if $n = 8$ and we have 4 threads, thread t_1 will be assigned the first vertex v_1, and when done processing this it will process vertex v_5. If the id of a thread's assigned vertex is greater than its mate or equal to $blockIdx.x$ or $M[blockIdx.x]$, another thread is responsible for this matching edge, so $blockDim.x$ is added to the vertex id. The thread then starts a new round if the vertex id is still smaller than n. Making some threads start a new round like this will lead to some degree of warp

divergence, and we will look at the effect of this in Chapter 5.

Each thread with id smaller than its vertex' mate calculates the gain of at least one 4-cycle, and if the gain is positive, it is stored in a local variable of the thread. If a thread computes gain for several 4-cycles, like t_1 above, the highest gain of this thread is stored.

When all of the matching edges have been processed by the threads inside a block, the computed gain values are used to find the edge that gives the 4-cycle of maximum gain. The reason why we store all the gains locally for each thread instead of just updating a shared highest gain along the way is because we can get a race condition if several threads try to update the gain value at the same time. We could have used atomics to solve this problem, but this could take a long time, since if many threads try to change the value at the same time they need to do it in a sequential manner. To find the flip option that gives highest gain we use a reduction from the CUB library called `BlockReduce`[7]. The `BlockReduce` reduction uses the threads of the block to perform a reduction (Section 2.4.10), with the binary function of the users choice. We used a custom Max()-function to find the matching edge that gives highest gain.

When a block has found the edge to flip with, one of the threads in the block uses the *atomicFlip* procedure (Algorithm 7) to flip safely. There are several options for what to do if the flip procedure fails. For instance, the threads in the block could wait for the next phase to start, or they could try iterating over the neighbors one more time. We experimented with both options to find that the best option was to wait for the next phase. These experiments are presented in Chapter 5.

Since we use the same locking method as in our first parallel ROMA-phase, Algorithm 6, we still have a guarantee that the resulting matching is valid. Lemma 4.1.1, which states that in each phase at least one vertex will be augmented as long as there exists at least one weight-increasing 4-cycle, holds for our new parallel ROMA, Algorithm 8, as well.

In Algorithm 6, we only used half of the threads we asked for (the ones with vertex id smaller than their mate), while we now in theory use all threads within the active blocks. To only use half of the threads can affect the performance negatively, since warps with many idle threads still have to complete their computations if there are any active threads within the warp.

Calling for a kernel with n blocks, but only using half of the blocks (the ones where the corresponding vertices has id smaller than their mate), may seem like a waste of resources. The reason we do this is that asking for n thread blocks is the easiest way to assign all matching edges to the blocks, since each vertex participates in a matching edge. Even

though only half of the blocks will do actual work, the SMs will discard the blocks not in use and start processing new ones, so this will likely not be a problem. In Chapter 5 we do some profiling on both versions (using the two different ROMA-phases) of parallel ROMA, to see which version makes the best use of the resources.

Our new parallel phase of the ROMA algorithm is presented in Algorithm 8.

Algorithm 8 A phase of ROMA with parallelized augmentation

ROMA-phase($G = (V, E, w : E \to \mathbb{R}^{\geq 0})$, volatile int array M, int array $locks$)

```
 1: u = blockIdx.x
 2: mateU = M[u]
 3: if u < M[u] then
 4:     x = threadIdx.x
 5:     triple (a, b, maxGain)
 6:     while x < V do
 7:         if x = u or x = M[u] or x > M[x] then
 8:             x += blockDim.x
 9:         else
10:             currentWeight = w[u, M[u]] + w[x, M[x]]
11:             w_1 = w[u, x] + w[M[u], M[x]]
12:             w_2 = w[u, M[x]] + w[M[u], x]
13:             if w_1 > w_2 then
14:                 gain =w_1-currentWeight
15:                 if gain > maxGain then
16:                     (a, b, maxGain) = (x, M[x], gain)
17:             else
18:                 gain = w_2 - currentWeight
19:                 if gain > maxGain then
20:                     (a, b, maxGain) = (M[x], x, gain)
21:             x += blockDim.x
22:     __syncthreads()
23:     (a, b, maxGain) = BlockReduce((a, b, maxGain), Max())
24:     __syncthreads()
25:     if maxGain > 0 and threadIdx.x = 0 then
26:         atomicFlip(M, locks, (a, b), u, mateU)
    return M
```

Comments on the code: $(a, b, maxGain)$ is now a triple consisting of a and b describing how to flip, as well as the gain of flipping the edges, instead of a tuple like in Algorithm 6. The integers a and b are vertex u's new mate and $mateU$'s new mate respectively. To find maximum gain by reduction one only needs the gains, but to make it easier to find the corresponding edges afterwards we give the triples as input to `BlockReduce` in line 23. The binary function we use for the reduction is a custom max-function that takes in

two triples and finds the max by comparing their gains.

The reduction works for the entire block, taking in each thread's triples, and returns the one with the highest gain. Then, if this gain is higher than 0 we make one of the threads in the block try to do the flip in line 26.

We combine the host-side and device-side of our parallel ROMA algorithm:

Algorithm 9 Parallel ROMA

parallel-ROMA($G = (V, E, w : E \to \mathbb{R}^{\geq 0})$, int array M, int array $locks$, int k)

1: if M is non-complete, pair two and two vertices
2: allocate memory for G, M and $locks$ on GPU
3: send G, M and $locks$ to GPU
4: int $flips$
5: int $phases = 0$
6: **while** $flips \neq 0$ **and** $phases < k$ **do**
7: send $flips = 0$ to GPU
8: ROMA-phase<<<$blocks, threads$>>> $(G, M, locks, flips)$
9: get number of flips from GPU
10: increment $phases$
11: get M from GPU
12: **return** M

Algorithm 9 stops at convergence, or after k phases, for some fixed number $k \geq 0$. Convergence means that that there are no more weight-increasing 4-cycles, resulting in a phase with zero flips. If one only wants a constant number of phases, the while loop can be changed to a for loop with k iterations and the flips-counter can be omitted. The ROMA-phase kernel call in line 8 can be Algorithm 8 or Algorithm 6.

Note that the flips-counter given as input to the kernel function is a value counting each time a flip is performed, but we have not included the pseudo code for this in Algorithm 6 or Algorithm 8.

Chapter 5

Experiments and results

In this chapter we compare our parallel ROMA algorithm, Algorithm 9, with the sequential ROMA algorithm in terms of runtime and performance. Most experiments on the parallel algorithm are done using Algorithm 8 as the ROMA-phase kernel, but we also do some experiments with Algorithm 6 as the ROMA-phase.

5.1 Input graphs

Since we are dealing with complete graphs, we can easily generate symmetric n x n matrices filled with weights for different values of n. We run the algorithms on graphs with different weight distributions to see how this impacts the performance. The weight distributions we use are random weights, exponential weights and weights derived from geometric graphs. The random weighted graphs have a uniform weight distribution with weights between 0 and 1 and each edge has an expected weight value of 0.5. To simulate the behavior of a general graph, we also use exponential graphs where most edge weights are close to zero. Our geometric graphs can contain vertices with many incident edges of large weight. These vertices are more likely to participate in augmentations, so geometric graphs could have different behavior compared to the two other weight distributions. We now present how the different graphs are generated.

Random weight graphs: The weight of every edge in the graph is set to a real number drawn uniformly at random from the interval $[0, 1]$.

Exponential weight graphs: We use the exponential distribution from the default random engine generator in C++ [25]. This has probability density function $\lambda e^{-\lambda x}$ for $x \geq 0$. We

choose $\lambda = 3.5$, as this gives a max value of x that is close to 1. The weight distribution has a higher occurrence of values close to zero than values close to one, and is illustrated below:

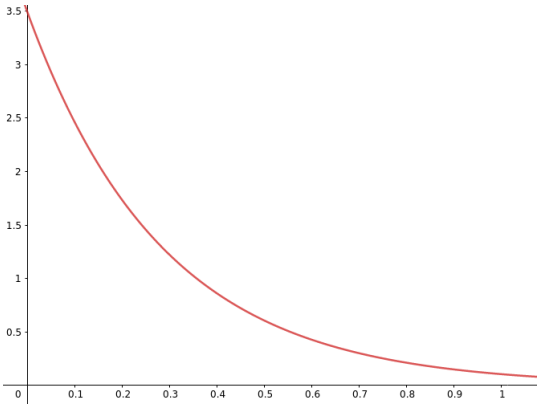


Figure 5.1: Exponential distribution with $\lambda = 3.5$

The mean value of this distribution is 0.29. A complete graph with exponential weight distribution is similar to a general graph in that many of the edges have so small weights that they can in practice be ignored.

The geometric graphs are generated as follows: n random points are generated in the plane within the unit square. These represents the vertices of the graph. The euclidean distance between pairs of points are then the edge weights of the graph. In this weight distribution points far from the center of the square will be incident on several edges of high weight.

We refer to the graphs of these three different weight distributions as random weight graph, exponential weight graph and geometric graph. The graphs generated are of vertex size $n = \{1024, 2048, 4096, 8192, 16384, 32768, 57344, 73728, 90112\}$, n always being a multiple of 1024. The reason for why we want n to be a multiple of 1024 is that this is the block size that we decided to use, and then it is practical if n is a multiple of this. The experiments where we find the best block size are presented in Section 5.5.1. Note that n does not need to be a multiple of 1024 for our algorithm to work, but it should ideally be a multiple of 32, which is the number of threads in a warp. The only actual requirement for n for the algorithm to work is that it should be an even number. If n is odd one can add a vertex where all incident edges have weight zero.

The largest graphs we generate have 90112 vertices, as larger graphs exceed the memory limit of the GPU that we use. 90112 vertices means an adjacency matrix of size $90112 \cdot 90112 \approx 8.12 \cdot 10^9$. The matrix contains floats of 4 bytes each, so it needs $8.12 \cdot 10^9 \cdot 4 \approx$

$3.25 \cdot 10^{10}$ bytes, which is close to 32 gigabytes. Note that since the matrix is symmetric, it would be enough to only store half of it, allowing for larger graphs. This would however slow down the lookup of the edges, and we therefore store the whole n x n matrix. The graphs with 90112 vertices has more than $4.06 \cdot 10^9$ edges.

5.2 Hardware

The GPU we use for our experiments is an NVIDIA Volta V100 from the Simula eX3 cluster [27]. It has 32 GB global memory, 80 streaming multiprocessors and a peak performance in double precision of 125 Teraflops. The compute capability is 7.0. A 64-bit 24-core x86 Xeon Platinum 8168 CPU is used together with the GPU. It operates at 3.7 GHz (turbo, 1 core) and has 768 GB of main memory. This CPU is also used in the experiments with sequential ROMA.

5.3 Method

Our sequential ROMA implementation is mainly written in C, but since we use the exponential distribution from C++ for generating some input graphs, we compile with g++ version 7.5.0. The parallel code is written in CUDA C/C++ and is compiled with nvcc V10.1.243 and the flag -arch=sm_70. We use the optimization flag -O3 when compiling both sequential and parallel ROMA.

Parallel ROMA 2, which is Algorithm 9 with Algorithm 8 as ROMA-phase, runs on n blocks of the GPU, where n is the number of vertices of the input graph. Each block uses 1024 threads, as this was found to be the best block size in our experiments. Parallel ROMA 1, Algorithm 9 with Algorithm 6 as ROMA-phase, uses $\frac{n}{1024}$ blocks, with 1024 threads per block.

When we tested the geometric graphs the number of flips in the last phases was very high, while the weight gain was close to zero. We therefore set the requirement for flipping to be that the gain should be at least 10^{-5} instead of gain > 0. This improved the running time of the geometric graphs, and did not change the solution quality noteworthy, as we will discuss in Section 5.5.4. The graphs with other weight distributions also had a small improvement in running time, while the solution quality was almost unaffected. We therefore use this requirement for all experiments that we present if not otherwise stated.

5.3.1 Runtime

We measure the runtimes of our parallel algorithms as well as the sequential ROMA algorithm by running 5 different graphs (different seeds) of each size. The average runtime of 5 runs per graph is computed. Then we average the runtimes of the 5 graphs of the same size. This is done for all three weight distributions.

When measuring the runtime of the sequential and parallel ROMA implementations, we only measure the time of the ROMA-phases/ROMA-kernels, and not the initialization of graphs, nor the time it takes to call cudaMalloc or cudaMemcpy for the graphs and matching arrays. The number of phases is set to be 8, but we allow the algortihms to stop earlier if there is no positive flips remaining.

We also measure the time and number of successful/unsuccessful flips per phase in the sequential and parallel ROMA algorithms for a chosen graph size of n=16384. Here we run one graph per weight distribution 5 times and take the average. The same is done to measure the weight increase in the matching for each phase.

5.3.2 Quality of the matchings

The quality of the matchings given by the sequential and parallel ROMA algorithms is measured by comparing their weights with the optimal solutions found by Edmonds' algorithm. Here we only look at smaller graphs (up to 32768 vertices) since the computation of the exact solution is very slow. The Edmonds' algorithm is only run once per graph since the exact solution does not need averaging of the weight results. For each weight distribution we only test for one graph per size because of the runtime. The parallel and sequential ROMA algorithms has 5 iterations of each graph, computing the average weight of each size. For the exact algorithm we used an implementation of Edmonds' blossom algorithm from the software package LEMON, Library for Efficient Modeling and Optimization in Networks [11].

5.4 Sequential ROMA

We start our experiments on the sequential ROMA algorithm, before testing the parallel implementations.

5.4.1 Runtime

The runtime of the sequential ROMA algorithm is measured on graphs of different weight distributions and sizes. Setting the number of phases to be numbered from 0 to 7, but also allowing the algorithm to stop at convergence gives the runtimes presented in Figure 5.2. The average number of phases used by the different graphs is presented in Figure 5.3.

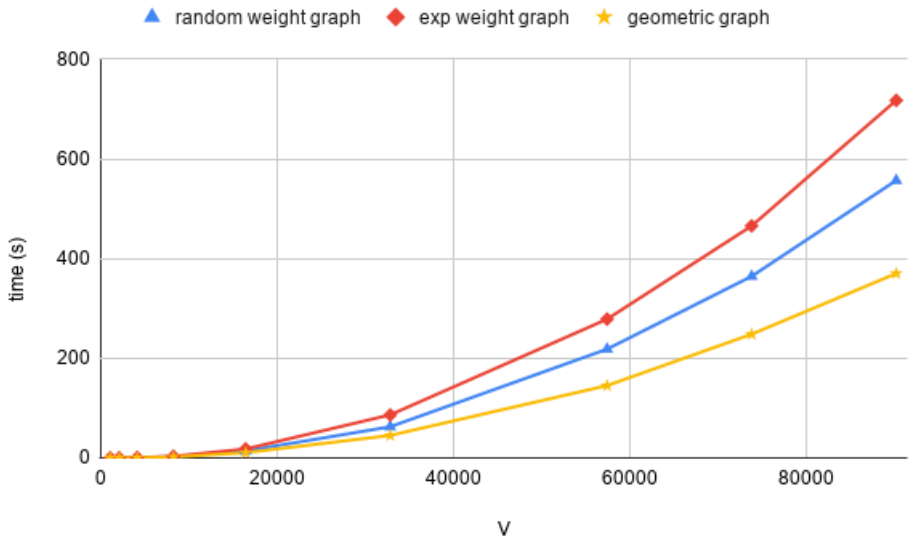


Figure 5.2: Sequential ROMA, average runtime on input graphs of size $n = 1024, 2048, 4096, 8192, 16384, 32768, 57344, 73728$ and 90112

We look closer at the running times for the smallest graphs of random weights where the number of vertices of a graph doubles from the number of the previous. For graph G_2 and graph G_1, where $|V(G_2)| = 2 \cdot |V(G_1)|$, we calculate the ratio $\frac{time_{G_2}}{time_{G_1}}$.

Table 5.1: Runtimes of random graphs, each graph having twice the number of vertices of the previous graph

V	runtime	ratio
1024	0.02996	
2048	0.12578	4.20
4096	0.63515	5.05
8192	2.80525	4.42
16384	14.08896	5.02
32768	62.57684	4.44

Similar tables with approximately the same ratios can be made for the exponential weight and geometric graphs, but we do not present them here. The runtime of ROMA is $O(l \cdot m)$, as discussed in Section 3.3.5, where l is the number of phases and m is the number of edges. Doubling the number of vertices from n to $2n$ results in a factor of 4 for the number of edges, since the number of edges goes from being $\frac{n^2-n}{2}$ to $\frac{(2n)^2-2n}{2} = \frac{4n^2-2n}{2}$. The runtime ratio of around 4 when doubling the number of vertices, as seen in Table 5.1, therefore seems reasonable.

The average number of phases used for each graph size are:

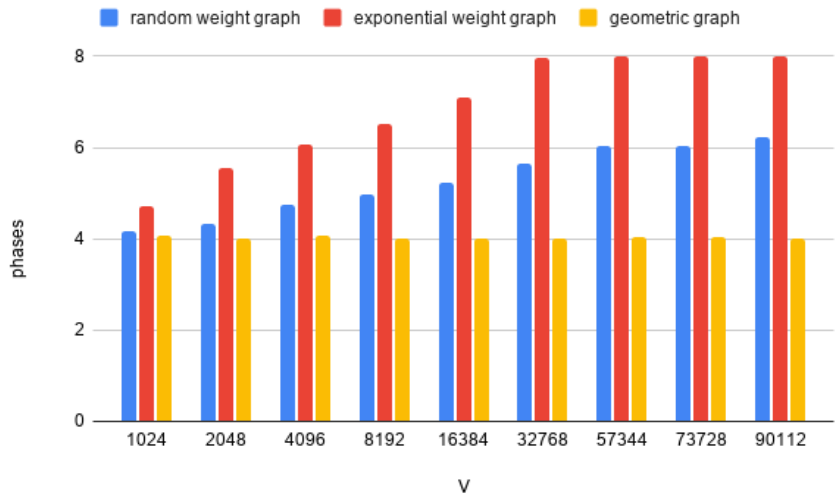


Figure 5.3: Sequential ROMA, average number of phases used on graphs of different sizes and weight distributions

The number of phases used for the different graphs did not vary much for each run of the same graph, but between the different weight distributions there were larger variations. The exponential graphs have the highest runtime. From Figure 5.3 one can see that the number of phases used is greater for these graphs, explaining why the runtime is higher.

The reason for why the exponential graphs need more phases could be because the weight of the random initial matching is low, since the majority of edges have small weights. We measure the weight increase of the different phases in the next section.

The graphs with a geometric weight distribution has a lower runtime than the other graphs, and the lowest number of phases. Before we set the requirement that the gain should be more than 10^{-5} to flip, the runtime of the random and geometric graphs were very similar. The reason for why the geometric graphs run faster with the new gain requirement is discussed in Section 5.5.4.

5.4.2 Flips and weight per phase

The number of flips and the weight increase per phase is measured on three graphs with different weight distributions and 16384 vertices.

Lines that stop before the last phase in the figures means that the solution converged and the algorithm stopped earlier.

Figure 5.4: Flips per phase, sequential ROMA

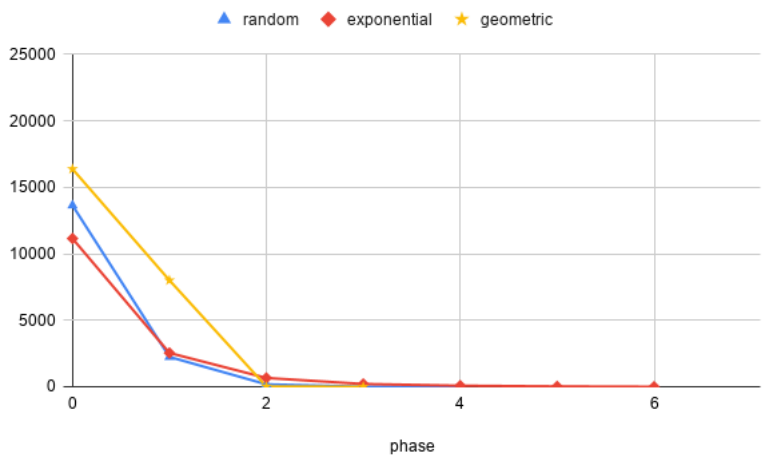


Table 5.2: Average number of flips per phase, sequential ROMA

phase	random	exponential	geometric
0	13647.2	11151.8	16384
1	2245.2	2544.2	8011
2	182	661.4	19
3	10,8	209	0.2
4	1.6	80	
5		28.2	
6		3.6	
7			
Total:	16086.8	14678.2	24414.2

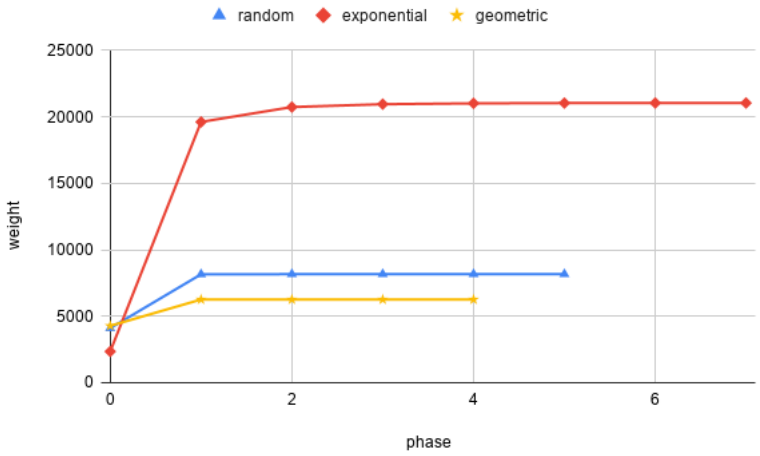
Figure 5.4 and Table 5.2 shows the average number of performed flips per phase of sequential ROMA on graphs of size n=16384. The number of phases used per run had small variations, so the last phase of each weight distributions has a number of flips greater than 0 because some runs of the algorithms used 1-2 extra phases. For example in the running of the random graphs, 4 runs stopped after 5 phases, while the 5th run used 7 phases. Since we only have one flip result for the 6th and 7th phase, we do not have enough data to average the number of flips for these phases. We therefore omit the results of such additional phases in Figure 5.4 and Table 5.2.

For the random weight graph 84.83% of the flips happen in the first phase. For the exponential and geometric graphs the numbers are 75.98% and 67.11% respectively.

While the number of flips decreases significantly from the first to the second phase for the random and exponential graphs, the number of flips in the second phase for the geometric graph is as much as half of the number from the first phase. Geometric graphs can contain some vertices with many incident edges of high weight, which are popular to flip with. A large amount of the vertices being augmented in a phase can therefore end up flipping with the same popular vertex. If many flips involving the same vertex are performed in a phase, all vertices that flipped with this vertex except the last one will have been given another mate than the one picked for them when the phase is done. Thus many vertices could need to perform a new flip in the next phase as well.

Figure 5.5 shows the start weight of each phase. The weight of phase 0 is thus the weight of the initial matching, the weight of phase 1 the weight of the matching given by phase 0 etc.

Figure 5.5: Weight at the start of each phase, sequential ROMA



The weight increase from the weight of the start matching of each phase to the weight of the end matching of the same phase is given as percent in Figure 5.3.

Table 5.3: Weight increase in percent, sequential ROMA

phase	random	exponential	geometric
0	98.67832	734.40572	46.31779
1	0.11628	5.70421	0.01040
2	0.00714	1.03053	0.00000
3	0.00047	0.30329	0.00000
4	0.00009	0.11192	
5		0.03877	
6		0.00347	

The weight of the returned matching of the graph with random edge weights is about the double of the initial start weight. This is expected since the weights have an expected value of 0.5, and picking edges at random, as done in the initial matching, should therefore give around half of the optimal weight. The exponentially weighted graph has the highest weight increase, the end weight being more than eight times larger than the start weight. Since most weights of this graph are small (close to zero), picking the right edges can increase the weight substantially.

For the geometric graph the increase in weight is not as high as for the other two graphs. This could be because the random initial matching for the geometric graph has higher weight compared to the optimal weight. Since geometric graphs can have vertices with many edges of large weight, picking an edge at random from each of these vertices gives higher possibility of picking an edge of large weight.

5.4.3 Time per phase

We measured the average runtime per phase on the same input graphs as for the flips and weight experiments above. Since each vertex is processed per phase of sequential ROMA, the phases should spend about the same amount of time for each phase. This was the case in some of the runs, but there were considerable variations in the runtime of one or more phases in other runs of the algorithm.

These variations did not seem to have any relation to the phase number or the number of flips performed during that phase, but appeared in random phases in some of the runs, increasing the average runtime of the given phase. The variations are probably due to traffic on the CPU that we use. Because of this, we do not present the results of these experiments here.

When measuring the runtime of the whole algorithm we did not experience these problems. We did not have any problems when measuring the time used by the GPU either.

5.4.4 Solution Quality

The solution quality is found by computing the average weight found by sequential ROMA on graphs using the different weight distributions, and comparing with the exact weight found by Edmonds' blossom algorithm. A common way to measure solution quality is to use the *gap to optimality, GTO*. This is computed as 1 - $\frac{weight_{ROMA}}{weight_{Edmonds}}$, and given as percent. A small GTO means good performance. As an example, an algorithm giving 99% of the optimal solution has a gap to optimality of 1%.

Table 5.4: Gap to optimality for sequential ROMA on input graphs of different weight distributions

V	GTO %		
	random	exponential	geometric
1024	0.95777	7.38184	0.00024
2048	0.70153	7.68937	0.00015
4096	0.50431	8.26674	0.00012
8192	0.36636	8.47281	0.00023
16384	0.26686	8.91158	0.00007
32768	0.18996	9.31349	

In Table 5.4, the largest geometric graph has no value because Edmonds' algorithm used too much time to compute the exact solution for this graph.

Combining the runtime results with the GTO, we see that sequential ROMA is both slower (Figure 5.2) and give worse matchings on graphs with exponential distribution, compared to the graphs with random weights between 0 and 1 and geometric graphs. As already suggested, this could be because the start matching is worse for exponential graphs when set at random. The solution quality is further discussed in Section 5.6.2.

5.5 Parallel ROMA

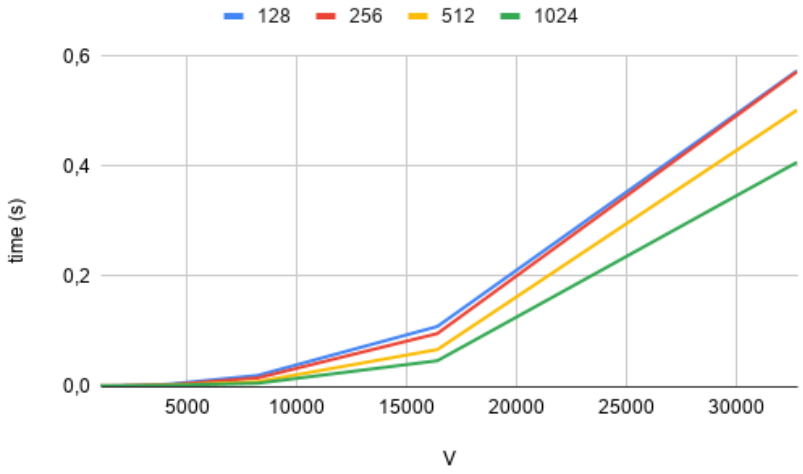
This section contains experiments for the parallel ROMA algorithms. The main focus is on the second version, parallel ROMA 2, but we perform some experiments using the first version, parallel ROMA 1, as well. The choice to focus on parallel ROMA 2 was made because this is the algorithm that was found to be the best out of the two, as the runtime experiments will show.

5.5.1 Choice of block size

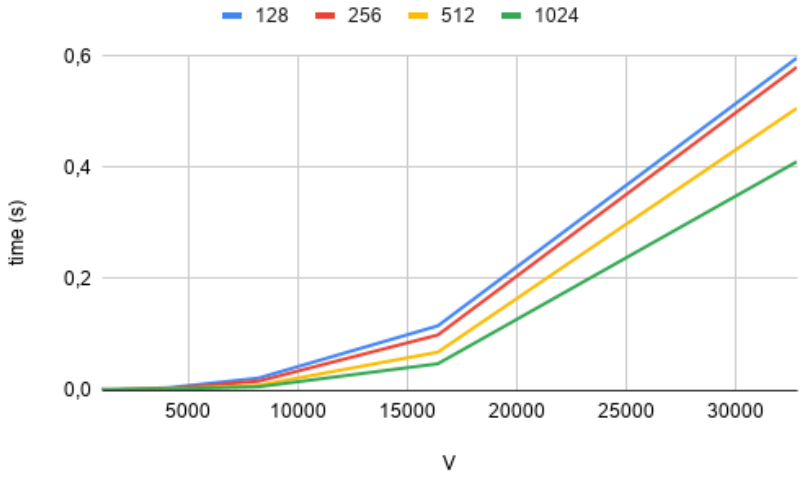
We need to set a block size before going on with the rest of the experiments. For this we run the algorithm parallel ROMA 2 with different block sizes to determine the optimal number of threads per block in terms of runtime and solution quality. The input graphs are of different weight distributions, sizes range between n=1024 and n=32768, and we average the results from 5 runs.

Figure 5.6: Runtimes, different block sizes on different input graphs

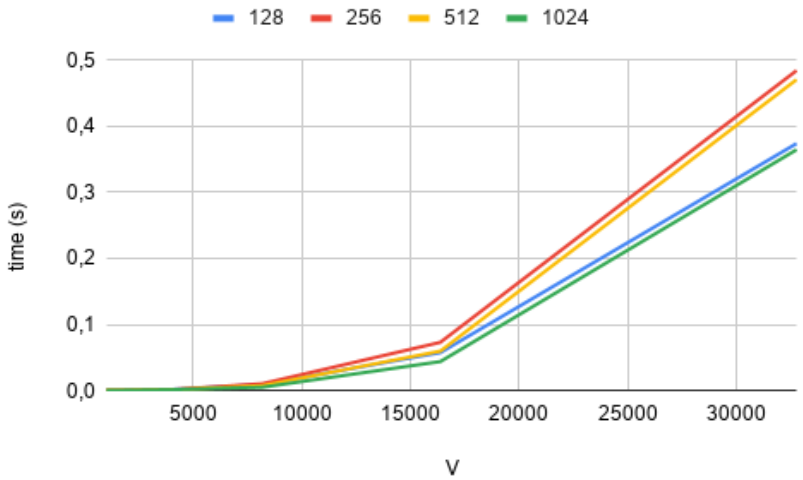
(a) random weight graphs



(b) exponential weight graphs



(c) geometric graphs



From Figure 5.6 one can see that the best block size in terms of runtime is 1024, which is the maximum number of threads per block that our GPU has to offer. When it comes to solution quality, there was not much difference between the block sizes for the random weighted graph. On the geometric and exponential graphs, the smaller block sizes gave lower weights. While a geometric graph of 1024 vertices had a GTO of 0.41% when using block size 1024, the number was 9.97% using 128 threads per block. For the exponential graphs of size 1024 the GTO was 10.61% using 128 threads per block, while a block size of 1024 gave a GTO of 7.00%

Since 1024 as block size showed to be best in terms of both runtime and solution quality for parallel ROMA 2, we use 1024 threads per block for the rest of our experiments.

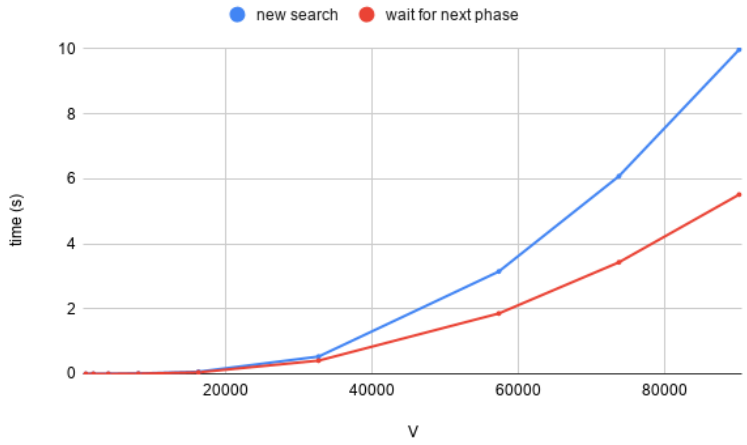
We also tested different block sizes for parallel ROMA 1. Here the best block sizes in terms of runtime were the opposite from Figure 5.6, with the fastest solution obtained using block size 128. However, once again the weights of the matching returned from the runs with smaller block sizes were worse than for the larger block sizes. A block size of 1024 gave the highest runtime, but also the best weights for all weight distributions. Because we are most interested in the results for parallel ROMA 2, we do not present the results from the testing of block sizes for parallel ROMA 1. Since the weights of the matching were highest with block size 1024, we choose this block size for the experiments of parallel ROMA 1 as well.

5.5.2 Options if flip fails

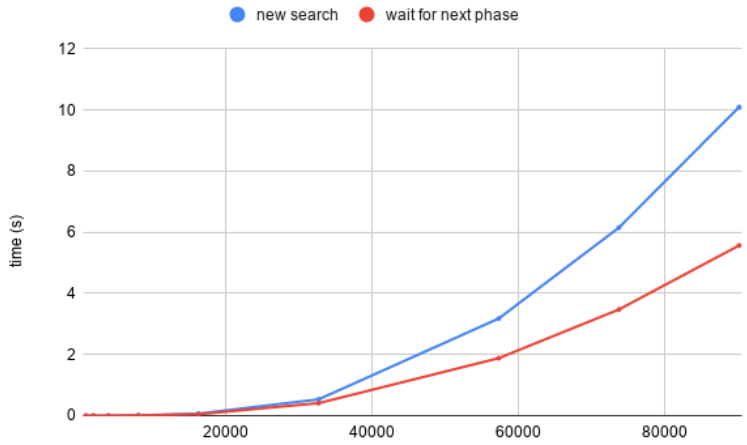
In parallel ROMA 2, if the flip procedure for a block fails, the threads of this block will wait for a new phase before trying to augment the vertex again. We discussed alternative ways to do this in Section 4.1.2, and now test the option of searching through the neighbors one more time, comparing it to waiting for a new phase. The input graphs are of size from n=1024 to n=90112 and have different weight distributions.

Figure 5.7: Runtimes parallel ROMA 2, waiting for new phase versus searching through neighbors one more time

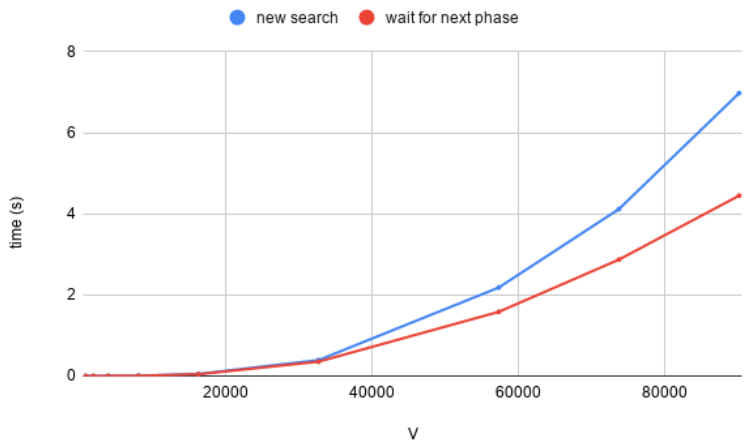
(a) random weight graphs



(b) exponential weight graphs



(c) geometric graphs



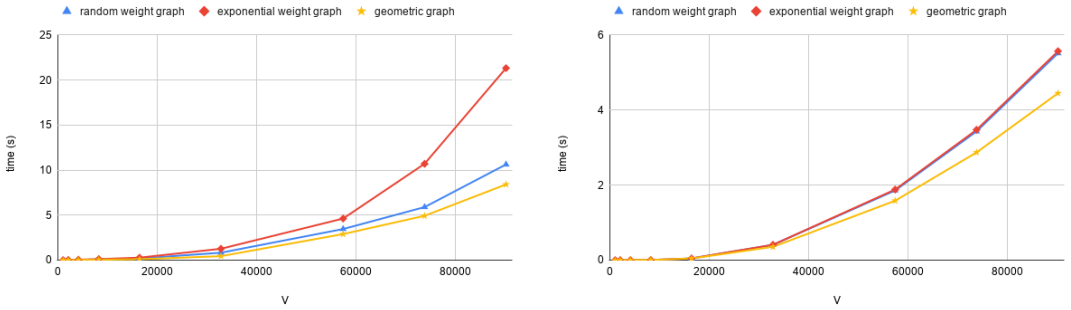
From Figure 5.7 one can see that the algorithm has better runtime when waiting for the next phase if the flip does not succeed, especially for the largest graphs. The average speedup for all weight distributions is 1.59 when waiting for the next phase.

If a set of vertices being augmented in different blocks fail to flip and need to start over again instead of waiting for the next phase, each of these blocks will need more execution time. Thus each kernel call will spend more time. If searching through the neighbors one more time meant that the number of needed phases/kernel calls decreased enough to compensate for the increased time per phase, this could still be a good approach. It turns out this is not the case. Even though the number of phases showed to be lower by one phase for some of the graphs when searching through the neighbors one more time, this is not enough. The reason is probably that the extra time used for each phase is so high that the total extra time of the phases added together is higher than the time saved by using one less phase.

5.5.3 Runtime

Running parallel ROMA 1 and 2 with 8 as the maximum number of phases gives us the following runtimes:

Figure 5.8: parallel ROMA 1 and 2, average runtime on input graphs of size $n = 1024, 2048, 4096, 8192, 16384, 32768, 57344, 73728$ and 90112



(a) parallel ROMA 1

(b) parallel ROMA 2

As one can see from Figure 5.8 parallel ROMA 2, is clearly faster on all inputs. We calculate the speedup $\frac{time\ \ parallel\ \ ROMA\ \ 1}{time\ \ parallel\ \ ROMA\ \ 2}$

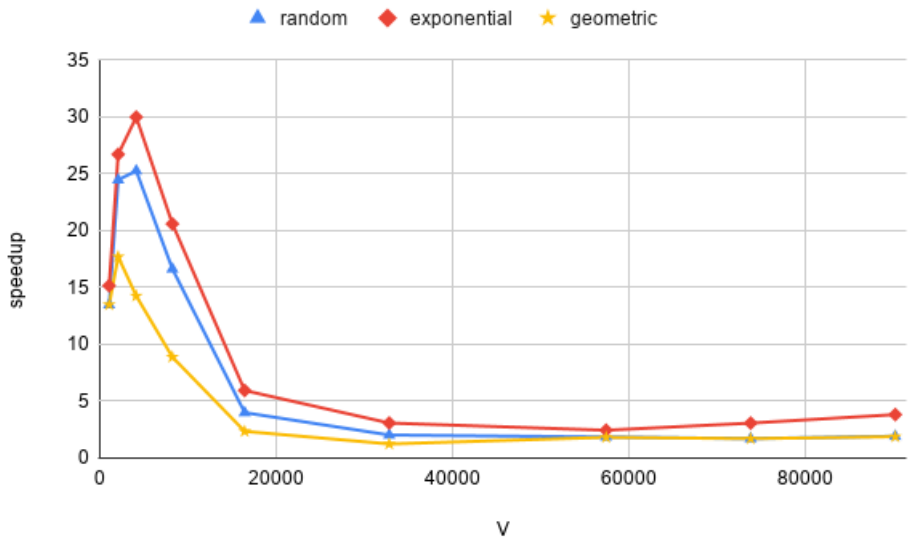


Figure 5.9: speedup parallel ROMA 1/parallel ROMA 2

We get the highest speedup for exponential weight graphs of size n=4096, where the speedup is 29.99. The highest speedup for the random weight graphs is also on the graph of this size, with a speedup of 25.27, while the geometric graphs has the highest speedup at 2048 vertices where it is 17.71. The average speedup for the random weight graphs is 10.16, for the exponential graphs it is 12.32, and for the geometric it is 7.04. The average speedup for all distributions is 9.84. In other words our second, improved version of parallel ROMA, parallel ROMA 2, is on average almost 10 times faster than the first version. This is why we mainly use this version in the experiments.

Figure 5.10 is a larger version of Figure 5.8b and shows the average runtime of parallel ROMA 2 on graphs with different weight distributions. Figure 5.11 shows the average number of phases used for each graph.

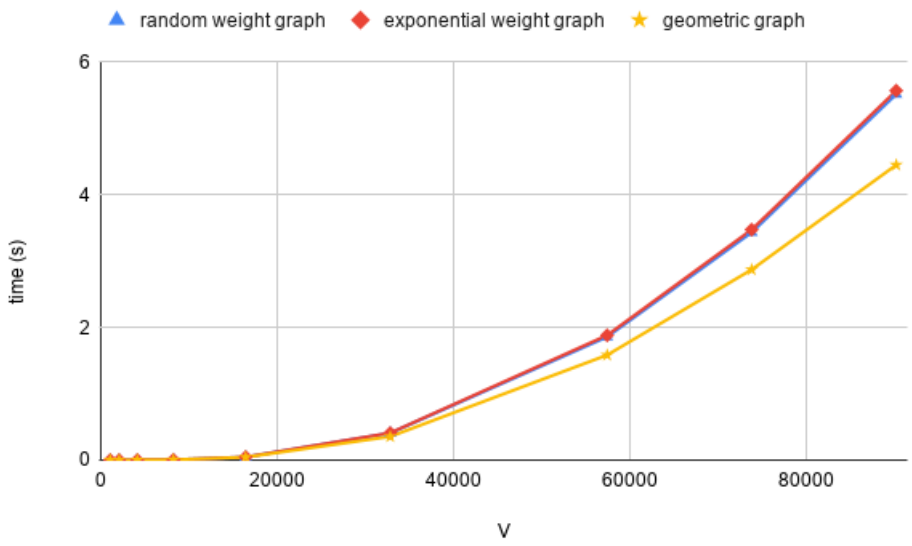


Figure 5.10: parallel ROMA 2, average runtime on input graphs of size $n = 1024, 2048, 4096, 8192, 16384, 32768, 57344, 73728$ and 90112

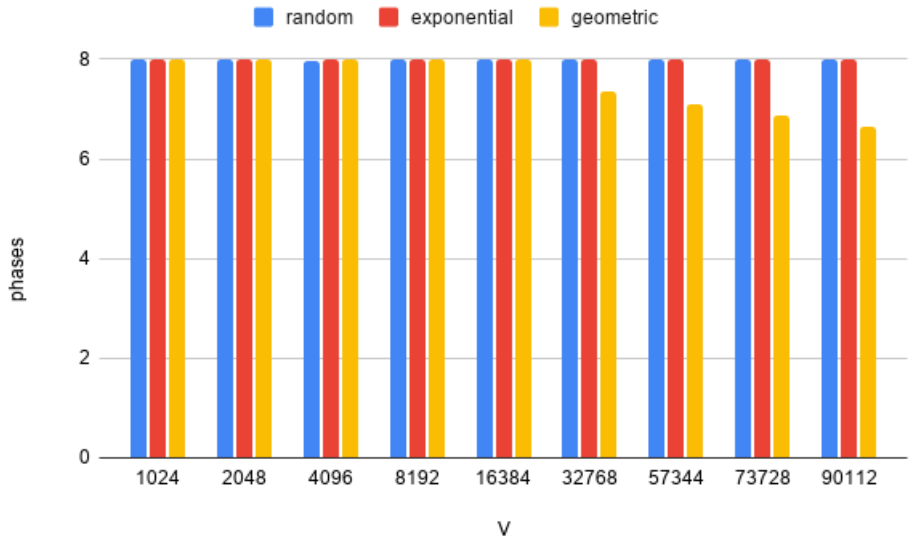


Figure 5.11: parallel ROMA 2, average number of phases

As shows in Figure 5.10, the different weight distributions does not seem to affect the runtime as much as it did for the sequential algorithm, see Figure 5.2. The reason is probably that all graphs were run for a maximum of 8 phases, but in the sequential version most graphs stopped earlier due to convergence. In the parallel version, both the random weighted and exponential weighted graphs run for 8 phases, while the geometric uses fewer phases as the graphs get larger, resulting in lower runtime. Why the geometric

graphs uses fewer phases has to to with the requirement that we should have a gain$> 10^{-5}$ to flip, and we discuss this further in the next section. Before we set the gain requirement, all three weight distributions had almost the same running time for each graph size.

The reason why parallel ROMA could need more phases to converge than sequential ROMA is that the maximum number of flips per phase is $\frac{n}{2}$ and not n like in the sequential version. This is because in parallel ROMA we augment the $\frac{n}{2}$ matching edges instead of the n vertices of the graph, resulting in fewer possible flips per phase. Even though the solution does not converge for parallel ROMA (except for the geometric graphs), this does not have to mean bad solution quality, as the last phases could have few and insignificant flips. If running more than 8 phases one could potentially get a better solution, but the difference in solution quality could be very small and not worth the extra time spent. We look at the solution quality in Section 5.5.6.

5.5.4 Flips per phase

We measure the number of successful flips per phase, compared to the number of potential flips. A successful flip is a flip that is performed, while an unsuccessful flip is when a call to the flip procedure does not result in a flip. The number of potential flips is the sum of successful and unsuccessful flips, in other words the number of calls to atomicFlip. The average number of successful flips per phase is shown in Figure 5.12 for parallel ROMA 2 on three graphs with different weight distribution of size n=16384.

Figure 5.12: Flips per phase, parallel ROMA 2

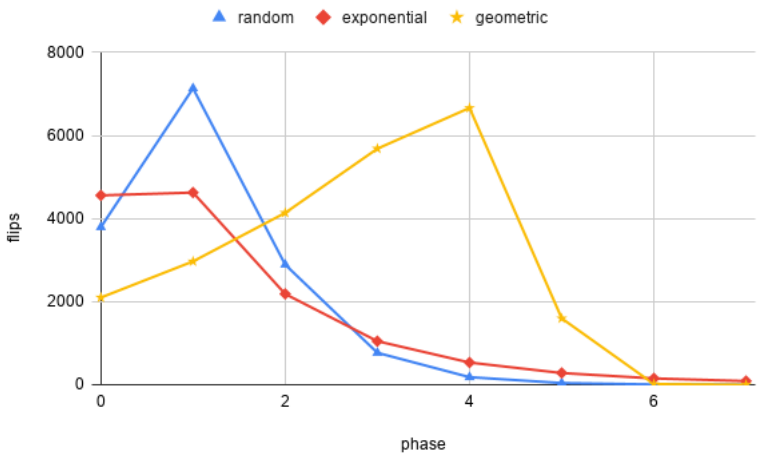
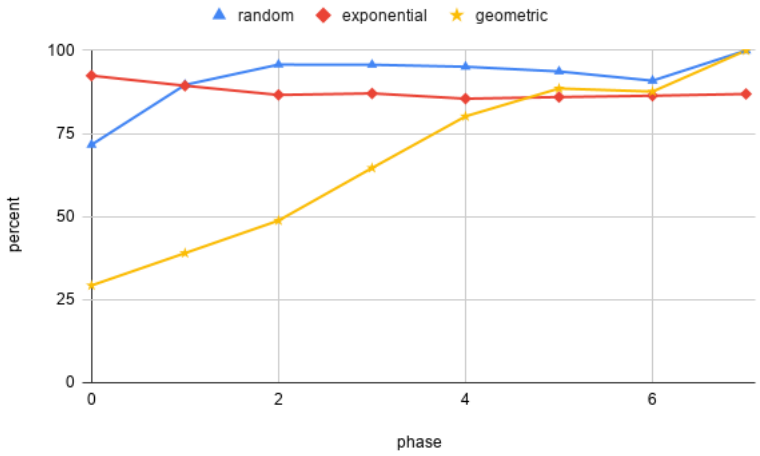


Table 5.5: Flips per phase, parallel ROMA 2

phase	random	exponential	geometric
0	3795.2	4559.6	2099.2
1	7136.4	4627.2	2970
2	2891.4	2185.4	4138.8
3	769.6	1048.8	5684.2
4	182.8	535	6664.8
5	41.6	282.2	1598.2
6	8	152.2	17
7	2	89	0.8
Total:	14827	13479.4	23173

We look at the percentage of successful flips per phase, which is the ratio $\frac{successful}{potential}$ flips.

Figure 5.13: Percent of successful flips per phase, parallel ROMA 2



From Figure 5.12 one can see that the most flips for the geometric graph happens in later phases than for the random and exponential graphs. Only 9.06% of the total flips for the geometric graphs happens in the first phase. In comparison the numbers for the random and exponential graphs are 25.60% and 33.83% respectively. Figure 5.13 shows that the geometric graph has a low percent of successful flips in the first phases. Geometric graphs can have many vertices with a high number of incident edges of large weight. Therefore is not surprising that fewer of the potential flips are performed in the beginning for geometric graphs, as several threads could compete for flipping the edges of the same popular vertices at the same time.

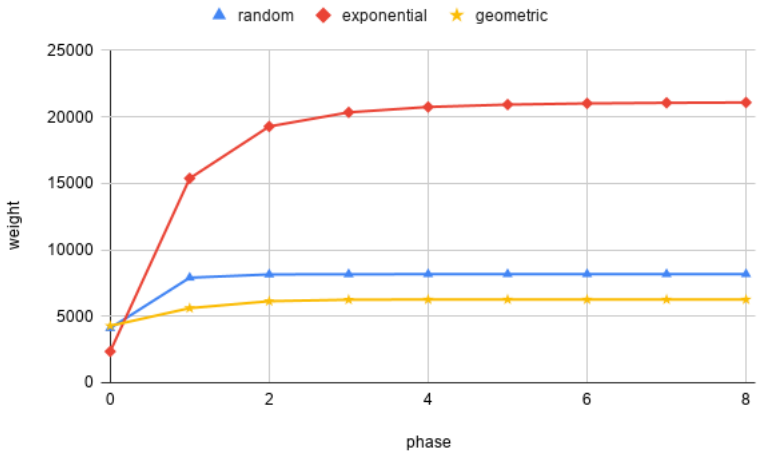
The graph with random weights has the highest number of flips in the second phase, where 48.13% of the total flips happen. This can have something to do with our initialization

of the start matching. In Section 5.5.5 we try another initialization to investigate this further.

The percent of successful flips on the graph with exponential weights is highest in the beginning, but does not change much during the phases. This is natural as most weights are close to zero, giving fewer weight-increasing 4-cycles and thus fewer collisions. The total number of flips for this weight distribution is lower than the others, but as we see in the next figure the exponential graphs has the highest weight increase.

Figure 5.14 shows the weights per phase. As done for Figure 5.5, the weight values are the weights of each phase's start matching in this figure as well. In other words, the weight of phase 0 is the weight of the initial matching, the weight of phase 1 is the weight of the matching returned from phase 0 etc.

Figure 5.14: Weight per phase, parallel ROMA 2



The weight increase from the weight of the input matching to the weight of the end matching for each phase is given as percent in Figure 5.6.

Table 5.6: Weight increase per phase in percent

phase	random	exponential	geometric
0	92.48325	553.73348	31.14509
1	3.09467	25.41469	9.26299
2	0.20120	5.50217	1.86412
3	0.03364	1.99432	0.23729
4	0.00736	0.85385	0.01491
5	0.00161	0.42883	0.00043
6	0.00030	0.21468	0.00000
7	0.00013	0.12394	0.00000

The largest increase in weight happens in the two first phases of parallel ROMA 2 for all weight distributions, similar to the results of sequential ROMA in Table 5.3. In parallel ROMA 2 the weight increase of the first phase (phase 0) is lower than for the first phase of sequential ROMA, while the weight increase of the second phase is much higher compared to sequential ROMA. The weight increase per phase decreases more gradually for parallel ROMA 2 compared to sequential ROMA. For example, while the weight increase in phase 3 for the exponential graph of sequential ROMA is only 0.30%, it is 1.99% for parallel ROMA 2.

We now calculate the weight increase from the initial start weight to the end weight after the two first phases. For parallel ROMA 2 we also calculate the weight increase after four phases.

Table 5.7: Weight increase in percent from initial start weight to the resulting weight after two phases for parallel and sequential ROMA, and after four phases for parallel ROMA

	random	exponential	geometric
sequential 2 phases	98.90933	782.00194	46.33300
parallel 2 phases	98.43998	719.87782	43.29305
parallel 4 phases	98.90611	782.23953	46.31055

Table 5.7 shows that when running only 2 phases, the total weight increase is higher for sequential ROMA. But when running parallel ROMA 2 for 4 phases we get a similar weight increase to running sequential ROMA for 2 phases. The GTO of the same phases are shown in Figure 5.8.

Table 5.8: GTO % of the resulting weight after two phases for parallel and sequential ROMA, and after four phases for parallel ROMA

	random	exponential	geometric
sequential 2 phases	0.27481	10.14628	0.00008
parallel 2 phases	0.51013	16.47516	2.07750
parallel 4 phases	0.27643	10.12207	0.01542

Since all vertices are augmented in a phase of sequential ROMA, while only half of them can be augmented in a phase of parallel ROMA, it is natural that the sequential version has the fastest weight increase. If one only runs sequential ROMA for 2 phases one should therefore run parallel ROMA for 4 phases to obtain the same weight increase. Table 5.8 shows that running parallel ROMA 2 for 4 phases while sequential ROMA is run for 2 phases, gives approximately the same GTO for the graphs of size n=16384.

The exception is the geometric graph, which has much better GTO for sequential ROMA. When running for 8 phases, sequential ROMA converges after 4-5 phases (except for the exponential graphs) as seen in Figure 5.3, while parallel ROMA 2 most of the time runs for all 8 phases (Figure 5.11). In Section 5.5.6 we find that running parallel ROMA 2 for 8 phases gives the same quality of the weights as sequential ROMA. The quality of the matchings from sequential and parallel ROMA is further discussed in Section 5.6.2.

Results for geometric graphs with and without the gain$> 10^{-5}$ requirement

Before we decided to use the requirement that gain should be greater than 10^{-5} for the flips to happen, we noticed that in the last phases of running the geometric graphs there were a lot of flips, but almost no increase in weight for the larger graph sizes.

Figure 5.15 shows the result of running a geometric graph of size n=90112 for both gain options. Figure 5.15a is the case where we only need the gain to be greater than zero to flip, and the number of gain values that are lower than 10^{-5} for each phase is shown.

Figure 5.15: Running a geometric graph of size n=90112 with and without the new gain requirement

```
phase number: 0, flips: 17518, time: 0.23938, gain < 0.00001: 0
phase number: 1, flips: 28221, time: 0.62065, gain < 0.00001: 0
phase number: 2, flips: 38012, time: 0.80551, gain < 0.00001: 0
phase number: 3, flips: 45058, time: 0.82118, gain < 0.00001: 10652
phase number: 4, flips: 46366, time: 0.83010, gain < 0.00001: 45883
phase number: 5, flips: 20182, time: 0.78781, gain < 0.00001: 20184
phase number: 6, flips: 2134, time: 0.76645, gain < 0.00001: 2135
phase number: 7, flips: 114, time: 0.76077, gain < 0.00001: 114
90112, start weight 23529.43985, end weight 34410.32655, time 5.63250
```

(a) flip when gain>0

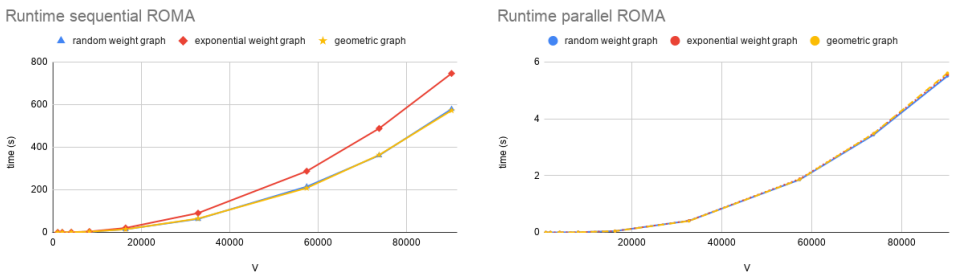
```
phase number: 0, flips: 17606, time: 0.23746
phase number: 1, flips: 28085, time: 0.62241
phase number: 2, flips: 37918, time: 0.80394
phase number: 3, flips: 36195, time: 0.80244
phase number: 4, flips: 3102, time: 0.75621
phase number: 5, flips: 0, time: 0.75526
90112, start weight 23529.43985, end weight 34410.27618, time 3.97812
```

(b) flip when gain$> 10^{-5}$

We used the largest graph here because this is the graph with most difference in runtime from the other weight distributions, as seen in the runtimes of both parallel ROMA (Figure 5.10) and sequential ROMA (Figure 5.2). One can see in the last phases in Figure 5.15a that the number of gain values lower than 10^{-5} and the number of performed flips are almost equal. Thus the last phases have thousands of flips that give almost no weight increase. In Figure 5.15b the number of phases is lower, because the last phases where the gain is small for each flip are removed. The total running time is thus lower, but

the solution weight is almost the same. Using this new gain requirement gives a lower running time for the large instances of the geometric graphs, because the last phases are not performed when the gain is very small. Before using this gain requirement, the running time for geometric graphs were close to the same as the running time for the random graphs in both sequential and parallel ROMA, as presented in Figure 5.16.

Figure 5.16: Runtimes of sequential and parallel ROMA without the new gain requirement

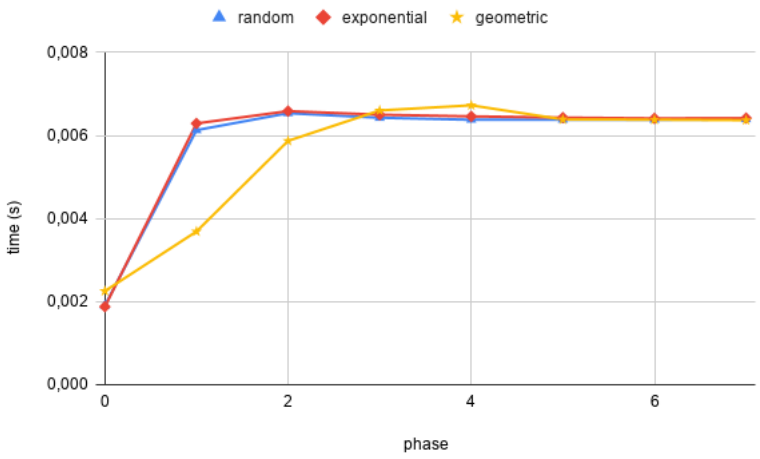


After changing to gain$> 10^{-5}$, the running time improved for the geometric graphs, while there were almost no change in the solution weights. We present the GTOs for both versions of parallel ROMA 2 in Section 5.5.6.

5.5.5 Time per phase

The average time per phase for parallel ROMA 2 on graphs with the different weight distributions and size n=16384 is:

Figure 5.17: Time per phase, parallel ROMA 2

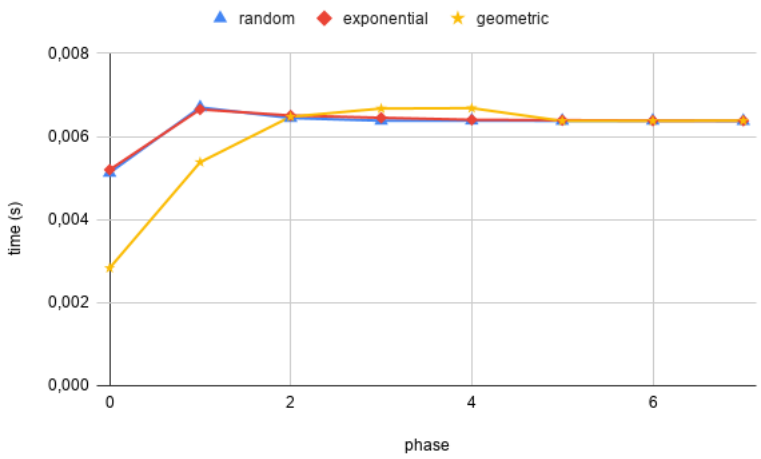


The time is lowest in the first phases of parallel ROMA 2, and stays the same after the 1-2 first phases for the random and exponential graphs. Considering the number of flips during the first phases for these graphs, this might seem strange. The most flips happens in the phases that uses the least time, and when the number of flips go down the time stays the same.

To explain this we need to consider the memory access patterns, as discussed in Section 2.4.5, of the different phases. In the first phase, the matching array is initialized such that vertex 0 is matched to vertex 1, vertex 2 is matched to vertex 3 and so on. The memory addresses to access from global memory are therefore close, compared to in the later phases where the matching array has been changed. Since we access global memory many more times than we flip edges, the time of the flipping does not affect the overall running time that much. This is probably why the last phases where the number of flips gradually goes down, have the same running time. Since the matching array has been rearranged a lot after the two first phases where the number of flips is high, the time used to access global memory has increased because the addresses asked for are more scattered across the matching array and graph array. In the geometric graph, the number of flips increases gradually during the phases, meaning there are less changes on the matching array in the beginning, and this can explain why it has lower runtime than the random and exponential graphs during the first phases.

To further test our theory, we tried another initialization for the start matching, where vertex 0 is matched to vertex $\frac{n}{2}$, vertex 1 to vertex $\frac{n}{2}+1$ etc. The resulting runtimes per phase for the same input graphs are given in Figure 5.18.

Figure 5.18: Time per phase, parallel ROMA with new initial matching

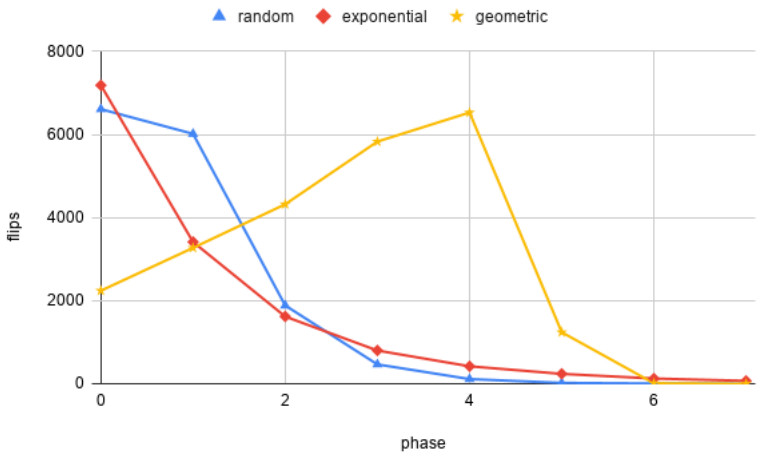


From Figure 5.18 one can see that the difference in runtime per phase is smaller for

the exponential and random graphs when using the new initial matching. This supports our theory, that the state of the matching array influences the runtime by how the threads access data from global memory. We also did some profiling to look at how the performance were influenced by memory accesses, and discuss this in Section 5.5.7.

The flips per phase for the new initialization is given in Figure 5.19.

Figure 5.19: Flips per phase, parallel ROMA with new initial matching



The flips per phase for the random weighted graph are highest in the first phase, instead of in the second phase like it was with our first initialization (Figure 5.12). Thus it is not only the time per phase that seems to be affected by the initial matching. How many times the matching is changed per phase seems to be affected by the start matching as well.

5.5.6 Solution quality

We measure the solution quality by the same means as for the sequential ROMA algorithm, using results from Edmonds' blossom algorithm to calculate the gap to optimality.

Table 5.9: Gap to optimality for parallel ROMA 2 on input graphs of different weight distributions

V	GTO %		
	random	exponential	geometric
1024	0.94992	7.00084	0.41312
2048	0.67850	7.66658	0.03664
4096	0.50521	7.99430	0.00041
8192	0.36627	8.52614	0.00023
16384	0.26805	8.78805	0.00008
32768	0.19007	9.35457	

Table 5.9 shows the GTO of parallel ROMA 2. The numbers are very close the the GTOs of sequential ROMA. We compare the quality of the solutions in Section 5.6.2.

Table 5.10: GTO, parallel ROMA 2 without new gain requirement

V	GTO %		
	random	exponential	geometric
1024	1.01325	7.60435	0.49820
2048	0.70295	8.12599	0.04109
4096	0.50015	7.94444	0.00035
8192	0.36893	8.55220	0.00010
16384	0.26556	8.72882	0.00007
32768	0.18653	9.36745	

Table 5.10 shows the GTO of a version of parallel ROMA 2 without the requirement that gain should be larger than 10^{-5}. The difference between the two versions of parallel ROMA 2 in terms of runtime was discussed in Section 5.5.4, and we now compare the solutions.

We calculate the ratios of the GTOs for the tables 5.9 and 5.10, and present them in Table 5.11 A number above 1 means that the parallel ROMA without the new gain requirement has the highest GTO, and numbers lower than 1 means that the version using the gain requirement $> 10^{-5}$ has the highest GTO.

Table 5.11: Ratio of GTOs from Tables 5.9 and 5.10

V	random	exponential	geometric
1024	1.06667	1.08621	1.20595
2048	1.03603	1.05992	1.12165
4096	0.98998	0.99376	0.85728
8192	1.00728	1.00306	0.42569
16384	0.99073	0.99326	0.96998
32768	0.98140	1.00138	

From the table one can see that both versions have the best GTO approximately half of the time. A similar GTO table could be given for the sequential ROMA algorithm, but we do not include it here. These numbers tell us that the change in gain requirement did not affect the solution quality notably.

5.5.7 Profiling

To compare parallel ROMA 1 with parallel ROMA 2 in terms of efficiency and utilization, we do some profiling using nvprof to see what has been improved in parallel ROMA 2. Like in the experiments where we measured flips and time per phase, we use a graph of size n=16384. We present results for the random weight distribution. The other weight distributions were tested as well, but gave almost identical numbers. The metrics used are [21]:

- achieved_occupancy: ratio of the average active warps per active cycle to the maximum number of warps supported on a multiprocessor
- branch_efficiency: ratio of non-divergent branches to total branches expressed as percentage
- warp_execution_efficiency: ratio of the average active threads per warp to the maximum number of threads per warp supported on a multiprocessor
- gld_efficiency: ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.
- gst_efficiency: ratio of requested global memory store throughput to required global memory store throughput expressed as percentage.

Figure 5.20: Profiling parallel ROMA 1

achieved_occupancy	branch_efficiency	warp_execution_efficiency	gld_efficiency	gst_efficiency
	%	%	%	%
0.485743	99.93	50.00	12.58	14.54
0.484667	99.93	49.99	12.56	13.57
0.484040	99.93	49.97	12.56	12.84
0.483021	99.93	50.26	12.53	12.15
0.480745	99.92	51.06	12.51	11.73
0.475245	99.92	53.51	12.51	11.76
0.467105	99.92	57.22	12.51	11.69
0.475153	99.91	62.24	12.49	11.44

Figure 5.21: Profiling parallel ROMA 2

achieved_occupancy	branch_efficiency	warp_execution_efficiency	gld_efficiency	gst_efficiency
	%	%	%	%
0.927098	76.84	87.30	33.35	12.43
0.940836	77.51	87.71	26.58	12.50
0.967685	77.46	88.09	24.65	12.45
0.985175	77.52	88.12	24.67	12.52
0.990375	77.60	88.17	24.67	12.44
0.992235	77.64	88.18	24.67	12.50
0.992330	77.65	88.18	24.68	12.50
0.992441	77.67	88.18	24.67	12.50

In Figure 5.20 and 5.21, the results per row are results for the different phases/kernel calls. Parallel ROMA 2 performs better than parallel ROMA 1 on all metrics except for the branch efficiency. The reason why the branch efficiency is lower for parallel ROMA 2 is probably because the threads can take two different execution paths when augmenting the vertices, depending on whether the vertex being augmented has higher id than its mate or not. In parallel ROMA 1, the threads of the vertices with higher id than their mates will simply stop executing, and this will not affect the branch efficiency.

The achieved occupancy for parallel ROMA 2 is almost twice as high as for parallel ROMA 1. Since only vertices with id lower than their mate are augmented in parallel ROMA 1, and each vertex is assigned its own thread, half of the threads will be inactive. An achieved occupancy of around 0.5 is therefore not surprising. In parallel ROMA 2, all threads within a block are given work, and instead half of the blocks are inactive. The achieved occupancy is therefore much higher. For the same reason, there is a large difference between the warp execution efficiency for the two versions of parallel ROMA.

The numbers does not change that much during the phases, but one thing to notice is the gld_efficiency from Figure 5.21. A higher efficiency in the beginning can explain why the runtimes per phase are higher during the later phases even though the number of flips decreases, as discussed in Section 5.5.5.

We do some more further profiling on metrics measuring global memory transactions to see if there is a difference between the two start matching initializations we tested in Section 5.5.5. One new initialization is tested as well.

The metrics we use are:

- gld_transactions: number of global memory load transactions
- gld_transactions_per_request: average number of global memory load transactions performed for each global memory load
- gld_requested_throughput: requested global memory load throughput
- gld_throughput: global memory load throughput
- gld_efficiency: ratio of requested global memory load throughput to required global memory load throughput expressed as percentage.

The original initialization matches vertex 0 with vertex 1, vertex 2 with vertex 3 etc. We refer to this initialization as neighbor matching. The next initialization matches vertex 0 with vertex $\frac{n}{2}$ and vertex 1 with vertex $\frac{n}{2}$+1, and we refer to it as $\frac{n}{2}$-matching. A third initial matching is constructed in the following way: the vertex array is divided in half. The last half is shuffled. Then vertex 0 is matched to the first vertex of the shuffled array, vertex 1 to the next vertex of the shuffled array etc. This matching is named shuffled matching.

Figure 5.22: Profiling parallel ROMA 2 on a random weighted graph og size n=16384 with different initial matchings

gld_transactions	gld_transactions_per_request	gld_requested_throughput GB/s	gld_throughput GB/s	gld_efficiency %
114308904	5.805426	589.816729	1766.241875	33.39
245589606	7.672186	314.651061	1194.351785	26.34
250538026	8.294406	283.617167	1154.791669	24.56
243433571	8.309972	278.653812	1135.449392	24.54
242314081	8.268442	275.660983	1124.093247	24.52
241908832	8.268698	275.909149	1125.290534	24.52
241871494	8.269425	274.523049	1119.738843	24.52
241855822	8.269883	275.361143	1123.221509	24.52

(a) Neighbor matching as start matching

gld_transactions	gld_transactions_per_request	gld_requested_throughput GB/s	gld_throughput GB/s	gld_efficiency %
230020337	8.340185	399.549105	1365.696369	29.26
260159186	8.337775	287.788961	1170.063759	24.60
247374757	8.277118	280.900083	1146.866496	24.49
242902483	8.275415	278.267535	1135.407443	24.51
242335919	8.307471	279.348461	1140.372004	24.50
242201145	8.305581	278.038293	1135.105904	24.49
242145626	8.304720	278.837968	1138.389403	24.49
242027706	8.303683	277.761892	1133.859061	24.50

(b) $\frac{n}{2}$-matching as start matching

gld_transactions	gld_transactions_per_request	gld_requested_throughput GB/s	gld_throughput GB/s	gld_efficiency %
267481175	8.430885	339.974267	1360.916728	24.98
260831105	8.360022	288.242422	1175.320552	24.52
246468694	8.341943	281.986137	1146.916487	24.59
242265115	8.371725	277.803733	1130.556187	24.57
241757516	8.380272	276.208758	1124.742022	24.56
241437471	8.381346	275.587625	1122.374202	24.55
241416774	8.381647	275.959222	1123.793062	24.56
241445996	8.382663	275.171343	1120.720219	24.55

(c) Shuffled matching as start matching

From Figure 5.22, if considering all initializations, one can see that the gld_efficiency is highest in the first phase using the neighbor matching. In this initialization the matched vertices lie closer in global memory. After the matching has been changed in the first phase, the gld_efficiency goes down. The number of gld_transactions_per_request is lower in the first phase, and when the matching has been changed the number increases. These numbers support our theory for why the first phase in Figure 5.17 in Section 5.5.5 has a lower runtime than the subsequent phases.

Using the $\frac{n}{2}$-matching as start matching, the gld_efficiency is lower in the first phase than for the neighbor matching, but still higher than for the rest of the phases. The gld_transactions_per_request number is the same in the first phase as for the rest.

Similarly for the shuffled matching the gld_transactions_per_request number is approximately the same for each phase. The gld_efficiency is almost equal in the first phase as for the next phases. Since this start matching is shuffled, the matched vertices lies more scattered across the global memory than for the two other initializations. This can explain why the gld_efficiency is worse for this initialization.

As Figure 5.22 tells us, the initialization of the start matching can affect the performance of parallel ROMA. In our implementation we did not take the memory access patterns into account. The memory accesses are therefore not particularly aligned or coalesced by the design of the algorithm, and it is the state of the matching arrays that decides how efficient memory will be accessed.

5.6 Sequential vs parallel ROMA

In this section we compare the results of sequential and parallel ROMA. We first present the speedup of the runtime, and then the quality of the solutions.

5.6.1 Speedup

The speedup is computed as $\frac{time_{sequential}}{time_{parallel}}$. For parallel ROMA 2 the speedup is given in Figure 5.23 and Table 5.12.

Figure 5.23: speedup sequential ROMA/parallel ROMA 2

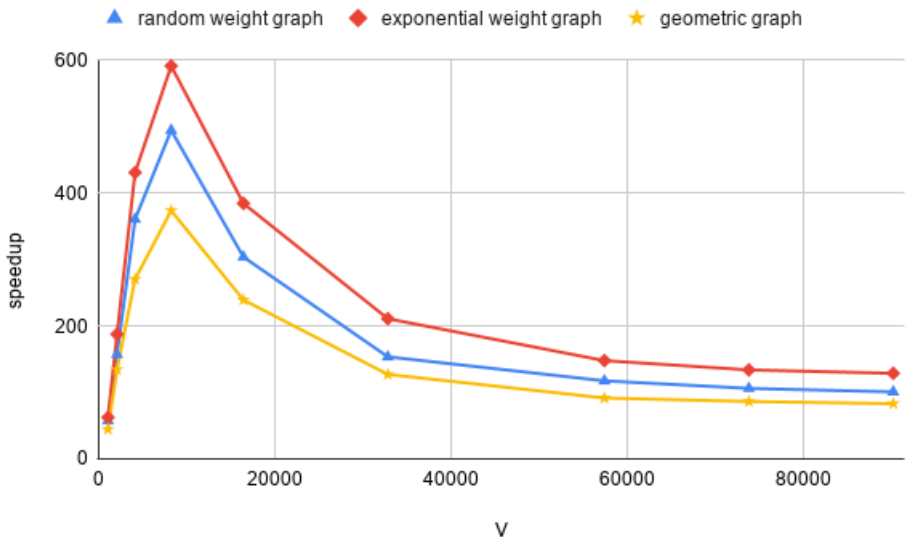


Table 5.12: Speedup sequential ROMA/parallel ROMA 2

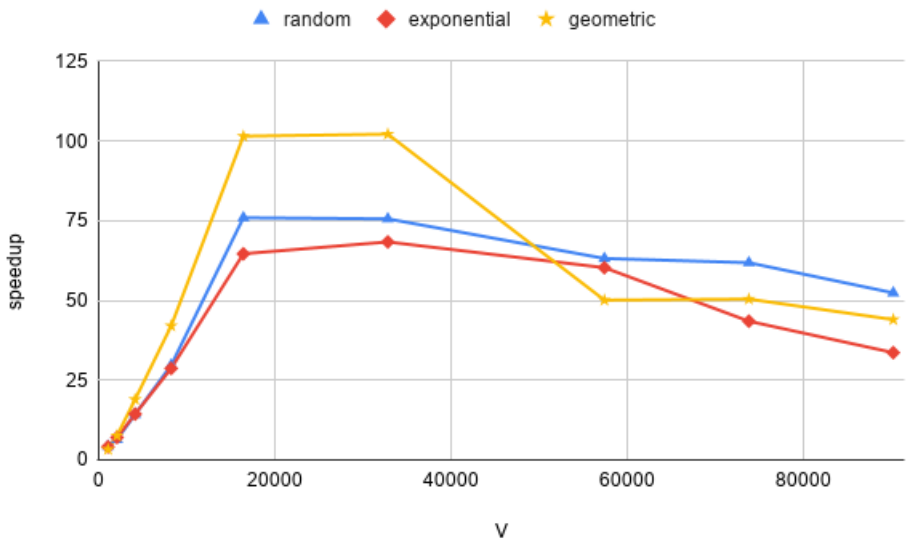
V	random	exponential	geometric	average
1024	57.62	62.46	44.68	54.92
2048	157.23	187.81	134.66	159.90
4096	360.88	431.09	270.52	354.16
8192	494.75	591.56	373.79	486.70
16384	304.03	384.64	239.39	309.35
32768	153.74	211.12	127.27	164.04
57344	117.56	147.99	91.68	119.08
73728	106.19	134.02	86.47	108.89
90112	100.96	128.85	83.19	104.33
average	205.88	253.28	161.29	206.82

The graphs of size n=8192 have the greatest speedups for all weight distributions, and an average speedup of 486.70. For this size the exponential graphs have the greatest speedup, with a speedup of 591.56. The reason for why the graphs with 8192 vertices have the highest speedup for all weight distributions could be that this size fits best with our hardware. The smallest graphs of size 1024 give the lowest speedup, with an average speedup of 54.92. Taking the average of the speedups of the different graph sizes for the different weight distributions, we see that the exponential graphs did best with an average speedup of 253.28, while the geometric graphs had the lowest speedup of 161.29. Averaging over both the graph sizes and the weight distributions gives a total average speedup of 206.82.

The reason for why the exponential graphs have the highest speedup could be that sequential ROMA uses more phases for these graphs, and thus more time. For the same reason the lowest speedup is for the geometric graphs where the number of phases of sequential ROMA is lowest.

The speedup of parallel ROMA 1 is presented in Figure 5.24.

Figure 5.24: speedup sequential ROMA/parallel ROMA 1



We discussed the speedup from parallel ROMA 1 to parallel ROMA 2 in Section 5.5.3, and presented it in Figure 5.9. Comparing the speedups in Figure 5.23 and Figure 5.24 we once again see that the second parallel algorithm, parallel ROMA 2 performs best on all input graphs in terms of runtime. Parallel ROMA 1 has an average speedup of 41.80, while parallel ROMA 2 has an average speedup of 206.82.

We now compare this speedup to the speedup obtained by previous parallel algorithms for the weighted matching problem. Since we have generated our own input graphs, we are not able to compare the speedups of specific graphs, and will instead look at the average and maximum speedups achieved.

In [15] the Suitor algorithm is run on two different machines. The first is a Dell computer with four 10-core 2.00 GHz Intel Xeon E7-4850 processors, and the second a Cray XMT-2 with 128 Threadstorm 500 MHz processors, where each processor can support up to 128 threads. A maximum speedup of 21,99 is reported for the Suitor algorithm on random generated graphs (not complete) when using OpenMP and 35 threads on the Dell computer. On the Cray XMT-2 the largest speedup achieved for OpenMP-Suitor

was around 100 (not easy to tell the exact speedup from the relevant figure).

In [19], the Suitor algorithm is implemented on an NVIDIA Kepler 40 GPU. This gave a maximum speedup of approximately 20 when comparing the GPU-Suitor to an OpenMP-Suitor using 2 threads. Only results of the OpenMP algorithm using 2 threads were reported, but using 1 thread instead the best one could hope for would be that the speedup increased by a factor of two.

The OpenMP-ROMA algorithm in [3] achieves an average speedup of 21.8 on input graphs from the *SuiteSparse Matrix Collection* when running 64 threads on a 2.1 GHz 16-core Intel Xeon Gold 6130 CPU. The maximum speedup achieved by OpenMP-ROMA is 44.0. In comparison, our algorithm has an average speedup of 206.82, and a maximum speedup of 591.56. Thus the average speedup of our algorithm is almost 10 times better than the average speedup of OpenMP-ROMA, and also better than the speedups of the Suitor implementations. However, it must be noted that the experiments are done on different hardware and with different input graphs, so it is hard to compare the speedups precisely. In addition, our algorithm only works for complete graphs, so it is expected that the performance would be worse for general graphs.

5.6.2 Solution quality

In Section 5.5.6 we gave the GTO of parallel ROMA 2. We now divide the GTOs of parallel ROMA 2 with the ones of sequential ROMA (Section 5.4.4) to compare them. Values below 1 means that parallel ROMA 2 has the lowest GTO, while values higher than 1 means that sequntial ROMA has the lowest GTO.

Table 5.13: Ratio of GTO parallel ROMA 2/ GTO sequential ROMA

V	random	exponential	geometric
1024	0.99181	0.94839	1688.71875
2048	0.96717	0.99703	242.85593
4096	1.00180	0.96704	3.49189
8192	0.99973	1.00629	0.99862
16384	1.00445	0.98614	1.01940
32768	1.00055	1.00441	

From Table 5.13 one can see that parallel ROMA 2 has the best GTO on about half of the graphs. On the geometric graphs the GTO of sequential ROMA were very close to zero for the smallest graph sizes, so the ratio between parallel and sequential ROMA

becomes quite high. Even though the ratio is high, the GTO of parallel ROMA is still very good on geometric graphs, as seen in Figure 5.9.

Note that we run the algorithms for 8 phases, and since sequential ROMA on most graphs converges after 4-5 phases, parallel ROMA running for 8 phases can use the double of this and achieve similar weight increase (as discussed in Section 5.5.4). Thus the main reason for why the GTO is equally good for sequential and parallel ROMA is because parallel ROMA runs for more phases.

The GTO for the exponential weight graphs are higher than for the others. As discussed earlier, this could be because the random start matching is worse for exponential graphs as fewer edges have high weight. To get a higher solution weight one could run another approximation algorithm to find a start matching before running ROMA. This could be for instance GPA or Greedy, but would require more time.

We now compare the GTO of our sequential and parallel ROMA algorithms to the GTO of some other ROMA implementations and Suitor. In [16] the reported GTO of ROMA lies between less than $\frac{1}{16}\%$ up to around 4% when run on various random instances. The best GTO is achieved on the complete geometric graphs, where it is close to zero (less than $\frac{1}{16}\%$) for the largest graphs, similar to our results. The other input graphs tested were not complete, so we focus on the geometric graphs. The complete geometric graphs used in [16] have from 2^7 to 2^{12} vertices, while our GTO results are for geometric graphs with 2^{10} to 2^{14} vertices. The gap to optimality of the complete geometric graphs in [16] starts at approximately 0.5% and decreases as the size of the graphs get larger. The decrease in GTO when increasing the graph size can also be observed in Table 5.9 and Table 5.4. Thus if comparing our results for the geometric graphs with the results reported from [16], the GTO values are similar, and both have this decreasing behavior of the geometric graphs' GTOs in common.

The OpenMP-ROMA algorithm in [3] achieves an average GTO of 2.32%, and a maximum GTO of 3.85% on the SuiteSparse Matrix Collection graphs. If averaging over all GTOs in Table 5.9, we get an average GTO of 2.93%. Thus we have very similar solution quality results if comparing the average GTOs. If comparing the maximum GTOs instead, our exponential graphs stand out with a maximum GTO of 9.35%. Again, since we do not compare results for the same graphs, it is hard to compare the quality of the solutions precisely, but we observe that the average GTOs are close to equal for GPU-ROMA and OpenMP-ROMA.

In [15] the Suitor algorithm has a GTO less than 0.4% for complete (geometric and random) graphs with 2^{10} to 2^{13} vertices, which is close to our results for geometric graphs.

For general random graphs the GTO of Suitor is 4.1%, and the maximum GTO reported is 9.2%. Our maximum GTO of 9.35% is thus very close to the maximum GTO of Suitor.

Total number of performed flips

If we compare the number of flips performed on the graphs from Section 5.5.4 and Section 5.4.2, we see that more flips are performed in sequential ROMA. Table 5.14 shows the total number of flips for these graphs.

Table 5.14: Total number of flips for sequential and parallel ROMA for graphs of size n=16384

	random	exponential	geometric
sequential	16086.8	14678.2	24414.2
parallel	14827	13479.4	23173

Even though sequential ROMA has a higher number of total performed flips for these graphs, the solution quality is approximately the same for sequential and parallel ROMA 2. This can be explained by the fact that sequential ROMA runs until convergence, and the last flips will not change the solution quality notably.

Chapter 6

Conclusion

6.1 Summary

The goal of this thesis was to implement the approximation algorithm ROMA on a GPU. We chose to focus on complete graphs, as this meant we only had to find 4-cycles when augmenting the vertices. Our two implementations, parallel ROMA 1 and parallel ROMA 2 both gave good results, with parallel ROMA 2 being the best with an average speedup of 206.82. The locking method that we used to avoid collisions worked well, and the parallel algorithms managed to perform a decent number of flips per phase when using it. If we allow the parallel algorithm to use twice as many phases as the sequential, the quality of the solutions given by parallel ROMA 2 are just as good as the quality of the solutions given by the sequential algorithm.

6.2 Future work

Even though we did some profiling of our algorithms, more analysis can be done. Our best implementation achieved a high speedup and the profiling gave good numbers for the achieved occupancy and warp execution efficiency, but there are probably still things that can be improved in order to speed up the running time further. For instance, if one manages to make the memory accesses more aligned and coalesced, the running time can probably be improved substantially.

Since we restricted ourselves to complete graphs, a natural next step could be to try to find a way that the ROMA algorithm could be implemented on a GPU for general

graphs as well. The challenge here is to find a safe and not too complicated way to do the augmentations when one has to consider all 2-augmentations instead of only 4-cycles. The OpenMP-ROMA algorithm in [3] uses a special locking method to perform safe 2-augmentations in parallel. This locking method is more involved than the one we use, but it could maybe be adapted to work on a GPU as well.

Another thing that we did not get the time to do is to write an OpenMP version of our algorithm.

It could also be interesting to see if one could implement the algorithm using multiple GPUs. Then one would have access to more memory and more threads, allowing us to solve for larger graphs.

Bibliography

[1] Ahmed Al-Herz and Alex Pothen. A parallel 2/3-approximation algorithm for vertex-weighted matching. In *2020 Proceedings of the SIAM Workshop on Combinatorial Scientific Computing*, pages 12–21. SIAM, 2020.

[2] Ariful Azad, Aydin Buluç, Xiaoye S Li, Xinliang Wang, and Johannes Langguth. A distributed-memory algorithm for computing a heavy-weight perfect matching on bipartite graphs. *SIAM Journal on Scientific Computing*, 42(4):C143–C168, 2020.

[3] André Berge. A parallel version of the random order augmentation matching algorithm. Master's thesis, University Of Bergen, June 2020.

[4] Marcel Birn, Vitaly Osipov, Peter Sanders, Christian Schulz, and Nodari Sitchinava. Efficient parallel and external matching. In *European Conference on Parallel Processing*, pages 659–670. Springer, 2013.

[5] John Cheng, Max Grossman, and Ty McKercher. *Professional CUDA c programming*. John Wiley & Sons, 2014.

[6] Shane Cook. *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes, 2012.

[7] NVIDIA corporation. *CUB documentation*.
URL: `https://nvlabs.github.io/cub/`.

[8] Doratha E Drake and Stefan Hougardy. A simple approximation algorithm for the weighted matching problem. *Information Processing Letters*, 85(4):211–213, 2003.

[9] Jack Edmonds. Maximum matching and a polyhedron with 0, 1-vertices. *Journal of research of the National Bureau of Standards B*, 69(125-130):55–56, 1965.

[10] Harold N Gabow. Data structures for weighted matching and nearest common ancestors with linking. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms*, pages 434–443, 1990.

[11] EGRES Group. Library for efficient modeling and optimization in networks, .
URL: https://lemon.cs.elte.hu.

[12] Khronos Group. *OpenCL*, .
URL: https://www.khronos.org/opencl/.

[13] Mahantesh Halappanavar, John Feo, Oreste Villa, Antonino Tumeo, and Alex Pothen. Approximate weighted matching on emerging manycore and multithreaded architectures. *The International Journal of High Performance Computing Applications*, 26(4):413–430, 2012.

[14] Fredrik Manne and Rob H Bisseling. A parallel approximation algorithm for the weighted maximum matching problem. In *International Conference on Parallel Processing and Applied Mathematics*, pages 708–717. Springer, 2007.

[15] Fredrik Manne and Mahantesh Halappanavar. New effective multithreaded matching algorithms. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 519–528. IEEE, 2014.

[16] Jens Maue and Peter Sanders. Engineering algorithms for approximate weighted matching. In *International Workshop on Experimental and Efficient Algorithms*, pages 242–255. Springer, 2007.

[17] Gordon E Moore et al. Cramming more components onto integrated circuits, 1965.

[18] Gordon E Moore et al. Progress in digital integrated electronics. In *Electron devices meeting*, volume 21, pages 11–13, 1975.

[19] Md Naim, Fredrik Manne, Mahantesh Halappanavar, Antonino Tumeo, and Johannes Langguth. Optimizing approximate weighted matching on nvidia kepler k40. In *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, pages 105–114. IEEE, 2015.

[20] NVIDIA. *CUDA C Programming guide*, .
URL: https://docs.nvidia.com/pdf/CUDA_C_Programming_Guide.pdf.

[21] NVIDIA. *CUDA documentation*, .
URL: https://docs.nvidia.com/cuda/.

[22] Peter Pacheco. *An introduction to parallel programming*. Elsevier, 2011.

[23] Seth Pettie and Peter Sanders. A simpler linear time 2/3- ε approximation for maximum weight matching. *Information processing letters*, 91(6):271–276, 2004.

[24] Alex Pothen, SM Ferdous, and Fredrik Manne. Approximation algorithms in combinatorial scientific computing. *Acta Numerica*, 28, 2019.

[25] The C++ resources network. *C++ exponential distribution.*
URL: `https://www.cplusplus.com/reference/random/exponential_distribution/`.

[26] Madan Sathe, Olaf Schenk, and Helmar Burkhart. An auction-based weighted matching implementation on massively parallel architectures. *Parallel Computing*, 38(12):595–614, 2012.

[27] Simula. ex3 cluster.
URL: `https://www.ex3.simula.no/`.

[28] Wikipedia. Moore's law picture.
URL: `https://en.wikipedia.org/wiki/Moore's_law`.