BACHELORTHESIS

Suitability Analysis of GPUs and CPUs for Graph Algorithms

Arbeitsbereich Wissenschaftliches Rechnen
Fachbereich Informatik
Fakultät für Mathematik, Informatik und Naturwissenschaften
Universität Hamburg

Vorgelegt von:	Kristina Tesch
E-Mail-Adresse:	3tesch@informatik.uni-hamburg.de
Matrikelnummer:	6525133
Studiengang:	Informatik
Erstgutachter:	Dr. Michael Kuhn
Zweitgutachter:	Prof. Dr. Thomas Ludwig
Betreuer:	Dr. Michael Kuhn

Hamburg, den 27.09.2016

Abstract

Throughout the last years, the trend in HPC is towards heterogeneous cluster architectures that make use of accelerators to speed up computations. For this purpose, many current HPC systems are equipped with Graphics Processing Units (GPUs). These deliver a high floating-point performance, which is important to accelerate compute-intensive applications. This thesis aims to analyze the suitability of CPUs and GPUs for graph algorithms, which can be classified as data-intensive applications. These types of applications perform fewer computations per data element and strongly rely on fast memory access.

The analysis is based on two multi-node implementations of the Graph500 benchmark, which execute a number of Breadth-first Searches (BFS) on a large-scale graph. To enable a fair comparison, the same parallel BFS algorithm has been implemented for the CPU and the GPU version. The final evaluation does not only include the performance results but the programming effort that was necessary to achieve the result as well as the cost and energy efficiency.

Comparable performance results have been found for both the versions of Graph500, but a significant difference in the programming effort has been detected. The main reason for the high programming effort of the GPU implementation is that complex optimizations are necessary to achieve an acceptable performance in the first place. These require detailed knowledge of the GPU hardware architecture. All in all, the results of this thesis lead to the conclusion that the higher energy efficiency and, depending on the point of view, cost efficiency of the GPUs do not outweigh the lower programming effort for the implementation of graph algorithms on CPUs.

Contents

1 Introduction

High Performance Computing (HPC) is essential to many application areas today such as climate and chemical research, biosciences as well as fluid dynamics and financial analysis. The increase in available computational power was the driving force behind the progress that has been made in those fields throughout the last decades. These days it is possible to predict weather catastrophes precisely as for example the position, at which a hurricane reaches the coastline as well as the path it will take, can be calculated [Cha14]. The human genome has been decoded and atom-to-atom interactions can be simulated for drug discovery [SCM14]. Solving increasingly complex problems seems to be in feasible reach, provided that the performance of supercomputers will rise even further. The computational strength of current supercomputer architectures arises from parallelism. In the 1990s a fundamental change took place with a shift away from specialized supercomputing hardware and towards cluster architectures that are built up from standard hardware connected through a network. Today, a compute node of a high performance cluster contains multiple Central Processing Units (CPUs), which contain multiple CPU cores themselves. Introducing parallelism to a single CPU's architecture was the solution to the problem of no longer increasing clock rates, which led to a steady performance increase of processors until then. The number of CPU cores per CPU is limited to a few dozen, while the number of compute nodes of a cluster is highly scalable. The era of high performance clusters began with a few compute nodes and reached a dimension of multiple ten thousand nodes by now[1]. Such a machine requires a lot of physical space and consumes a great amount of energy, of which a large portion is used for cooling. That way the performance of a supercomputer, which is usually measured through the number of Floating-point Operations Per Second (FLOPS), is not only limited by investment cost but by the immense operating costs related to energy consumption.

During the last couple of years, a new trend has emerged that led to the usage of accelerators, especially Graphics Processing Units (GPUs). These derive very high performance from a highly parallel hardware architecture. A GPU cluster is built up from CPU nodes that are equipped with at least one but more likely multiple GPUs. According to [Fel15], a third of all HPC systems make use of accelerators, of which 80% are GPUs. Cost and energy efficiency with regard to the delivered floating-point performance can be considered as the main reasons for this development. In contrast, the main obstacle preventing GPU usage from spreading even further in HPC is the necessity to port applications to GPU. Frequently, this process includes significant changes to the code through switching to a GPU-capable language as well as adapting algorithms to fit the GPU hardware architecture. A number of publications have shown notable performance improvements for compute-intensive applications containing a substantial portion of parallelism, e.g., [GLR+14][BKS15]. For this kind of applications the effort to enable GPU support seems to pay off, but unfortunately, not all HPC applications

[1] The Sunway TaihuLight supercomputer located in China, heading the TOP500 list from June 2016, is composed of 40,960 nodes that contain 10,649,600 cores in total.

can be described this way. While compute-intensive applications perform a high number of computations per data element, data-intensive applications process large amounts of data with much fewer computations per data element. As a result, the floating-point performance of a system is strongly related to the performance of compute-intensive applications, whereas other properties, as for example memory access time, have a greater impact on the performance of data-intensive applications as well. To take this into account, the Graph500[2] benchmark was introduced in 2010 to measure the performance of a supercomputer with regard to data-intensive applications.

The Graph500 benchmark performs a Breadth-first Search (BFS) on a large-scale graph. This means that the graph structure is traversed from one starting point until all nodes in the connected component are reached. A system's performance is measured through the number of Traversed Edges Per Second (TEPS). A BFS on a large-scale graph was chosen for the Graph500 benchmark with the objective that "the benchmark must reflect a class of algorithms that impact many applications" [MWBA10]. As the structure of a graph is capable of representing complex dependencies, graphs and thus graph algorithms are used in many application fields. These include national economy, logistics, social media as well as biology [FDB+14]. The performance of a system on running BFS is considered a meaningful indicator for the performance of a wide range of graph applications. On the one hand, this is deduced from the fact that BFS is a fundamental building block for many other graph applications and, on the other hand, many typical characteristics of graph algorithms are inherent in BFS. This includes unstructured memory access as well workloads that are difficult to balance. These characteristics, in particular, led to the assumption that graph algorithms, as well as other data-intensive applications, cannot benefit from GPU usage.

Nevertheless, a lot of research has been done on this topic recently with the hope to turn the high floating-point performance of modern GPUs into a high graph algorithm performance. Several implementations delivering great results for BFS on GPUs have been published. According to Luo, Wong and Hwu in [LWH10] the GPU-enabled Graph500 implementation even outperforms the available CPU implementations and Merrill, Garland and Grimshaw conclude in [MGG12] that "GPUs are well-suited for sparse graph traversal". However, it has to be remarked that in the great majority of cases the published results are for a single GPU or a multiple GPUs but single node implementation. This is true for both of the publications cited above. If considering the size of real-world graph data sets, which can contain billions of nodes and edges, it becomes visible that a single node or a single GPU implementation does not target real-world graph problems. Multi-node GPU implementations of Graph500 have been published as well, but their performance results can only be a partial answer to the question of the suitability of GPU clusters for graph algorithms. In [MGG12], the statement on the suitability of GPUs for BFS is based on a performance comparison only, which omits a number of other important factors. The performance will usually be the highest rated characteristic, but the effort that has to be made to achieve the performance results should be taken into account as well. This is a relevant point in HPC as code will not only be written by computer scientists but domain scientists as well. A hardware configuration that delivers a sufficient performance at low programming effort might appear more suitable for their research than a hardware configuration that has the potential of delivering an even better performance but featuring a highly complex programming

[2] `http://www.graph500.org`

model. The goal of this thesis is the evaluation of the suitability of a GPU cluster for graph algorithms in comparison with a CPU cluster, not only based on the performance but also the programming effort to achieve the performance results as well as energy and cost efficiency.

High GPU performance results are often found through the comparison with a CPU implementation that uses an obsolete algorithm or is not well optimized [LKC+10]. To enable a fair comparison, the CPU, as well as the GPU version of Graph500, have been implemented for this thesis with comparable optimization effort. Both implementations are based on the same algorithm structure, which is the key to allowing a conclusive comparison of the program's complexity together with the programming effort. The same algorithm structure means that the same data partitioning scheme, as well as inter-node communication pattern, is used. Local computations on the graph data have been optimized for the hardware architecture. The main contribution of this thesis is the analysis of the suitability based on two comparable implementations taking into account not only the resulting performance but other criteria, which will have a strong impact on the choice of a hardware configuration.

The remainder of this thesis is structured as follows. Chapter 2 provides an overview of the modern GPU architecture as well as GPU programming concepts. A number of GPU programming platforms are introduced, which will be evaluated in this thesis to find a preferably simple and suitable platform for the implementation of Graph500 on a GPU cluster. In addition, the key features of a modern CPU and the parallel programming model for CPUs are described. The third chapter describes the characteristics of graph algorithms and highlights the difficulties in utilizing the hardware of CPUs and GPUs for this kind of applications. A brief introduction of the Graph500 benchmark is included in this chapter as well as an overview on the current state of research. Chapter 4 explains the common algorithm structure of the implemented parallel BFS. It focuses on the overall structure and leaves out details on the hardware specific optimization for the next chapter. Chapter 5 includes these details and a comparison of three GPU programming platforms that aims to choose the most promising approach for further optimization. Multiple optimization steps are explained for the GPU implementation of Graph500 as well as for the CPU implementation. The 6th chapter introduces the test system and provides a detailed analysis of the performance results along with programming effort as well as the cost and energy efficiency. A final conclusion will be given in Chapter 7.

2 High Performance Computing with GPUs and CPUs

The following chapter provides an overview of CPU and GPU programming concepts in the context of High Performance Computing (HPC).

2.1 Modern GPU Architecture

The modern Graphics Processing Unit (GPU) has its origin in 3D graphics rendering. For this purpose, a highly parallel and throughput-oriented architecture is suitable because the pixels of an image can be processed independently and the computation time of a single pixel is not important as long as the computation of the whole image can be completed within a certain time frame. The modern GPUs still follow these design principles but their usage is not restricted to graphics processing anymore. Instead, the great computational power is used to accelerate HPC applications.

According to [Fel15], a survey by Intersect Research from October 2015 reveals Nvidia as the dominant vendor of accelerator hardware in HPC. In the summary, it is stated that 78% of all accelerator hardware used in HPC is manufactured by this company. For that reason and because Nvidia K80s are used for measurements in this thesis, Nvidia hardware will be used as an example during the next sections. From the information provided by the current Top500 list from June 2016[1], it can be inferred that the Nvidia Kepler architecture is one of the most commonly used GPU architectures in the HPC area today, most likely because the architecture is dedicated to accelerating HPC applications [NVI12]. A Nvidia K80 contains two separate GK210 graphics cards with also disjunct memories. These two GPUs are connected to a PCI switch inside the chassis. The K80 is then connected to the hosting CPU through a PCIe bus. This leads to a programming model employing the following steps: copying data to the GPU's global memory, doing computations on the GPU and copying the results back to the CPU.

2.1.1 Execution Model

The GPU utilizes an execution model called Single Instruction Multiple Threads (SIMT), which is deduced from the fact that one instruction is executed by a very large number

[1] `https://www.top500.org/lists/2016/06/`

of threads. The threads are running on processing units called “CUDA cores” in Nvidia’s terminology. Compute Unified Device Architecture (CUDA) refers to a GPU programming platform that is introduced later in this chapter. A GK210 contains 2496 CUDA cores, which are evenly distributed over 13 Streaming Multiprocessors (SMs). The 192 CUDA cores are colored green in the block diagram of a Kepler SM, depicted in Figure 2.1. The Kepler SM architecture is called SMX by Nvidia. A group of up to 1024 threads forms a block, which is assigned to one SM and subdivided into groups of 32 threads. These groups are named a warp. All threads in the same warp are scheduled together and execute the same instruction synchronously. If a conditional control construct leads threads of the same warp to take different paths, those paths are executed in sequence. Threads that follow a different path stall until their path is being evaluated. The orange line at the top of

Figure 2.1: Streaming multiprocessor (SMX) architecture [NVI12, p. 8]

the block diagram (Figure 2.1) shows four warp schedulers with two dispatch units each. Either of them selects a warp with threads ready to execute and dispatches up to two independent instructions to be executed by the threads of the warp per clock cycle [NVI12, p. 9].

The computational power of a GPU can only be utilized if there are always enough active warps. If the threads of a warp have to access data from the global memory, they have to wait many clock cycles for the data to arrive. In this case, the warp scheduler switches to another warp and dispatches its next instructions. As soon as the data has arrived the warp is set active again and the warp scheduler can select it to dispatch its next instructions. The switching between warps has no overhead as the execution context of each warp is kept

on-chip the entire time [NVI15, p. 81]. Therefore, enough active warps make it possible to hide memory latency completely. As the scheduling of warps in a block is independent of each other, no assumptions on the execution order of instructions executed by threads belonging to different warps can be made. To enforce synchronization between the threads of a block, the intrinsic function `__syncthreads()` can be called [NVI15, p. 12]. The same holds true for the blocks of threads that are scheduled to run on a SM: no assumption on the execution order can be made. To synchronize threads in different blocks the function that is executing on the GPU has to be split at the planned synchronization point. A function that is executed on the GPU is often called a kernel. The new kernels have to be invoked one after another from the hosting CPU.

2.1.2 Memory Model

The GPU's memory is usually disjoint from the CPU's memory, which enforces the typical workflow of any GPU program to start with copying data to the device's global memory [Bar15, p. 410]. A GK210 comes with 12GB on-board DRAM memory, whereof the most falls to global memory. This reveals a huge difference to high-end CPUs, which can be equipped with multiple times as much memory. It is common that compute nodes have 256 GB or even more RAM available. Additionally, the CPU memory access is accelerated through a cache hierarchy, which is transparent to the programmer in respect of functionality, whereas the GPU uses both a cache hierarchy and user-managed shared memory. Shared memory is located on-chip and within a streaming multiprocessor. In Figure 2.1, it is depicted through one of the light blue bars at the bottom. The memory available on every streaming multiprocessor is split into shared memory and L1 cache. A limited number of configurations is available[2] and a preference can be chosen through the call of a CUDA Runtime API function. As blocks of threads are assigned to streaming multiprocessors in an unpredictable way, only threads of the same block can use shared memory for communication purposes.

The GPU provides a very high theoretical bandwidth to global memory, 240GB/s for a GK210, but such a performance can only be seen if the data is accessed in a way such that the memory access is coalesced by the device. Memory coalescing, which leads to a reduction of the number of memory transactions, takes place if the threads of a warp access contiguous data in global memory at the same time [Har13a]. The access to shared memory is much faster than to global memory as the former takes 1-32 clock cycles and the latter takes 400-600 clock cycles [Bar15, p. 415]. Similarly to global memory, the variance in shared memory access time depends on the access pattern used. To minimize latency, bank conflicts have to be avoided when accessing shared memory. Banks are memory modules, which can be accessed simultaneously. Those banks are arranged in a way such that contiguous addresses map to different banks. One bank can serve 32-bit per clock cycle, but requests for addresses located in the same memory bank are processed in sequence. This is referred to as bank conflict and results in higher latency. An exception occurs in the case of the same address being requested by multiple threads because these requests can be served together [Har13b].

[2] For devices using the Nvidia Kepler architecture, it is possible to choose a 48KB/16KB or a 32KB/32KB split of the total 64KB [Bar15, p. 417].

2.2 GPU Programming Platforms

Throughout the last years, a great number of platforms has established to simplify GPU programming. The two most important ones and the basis of all other approaches are CUDA and OpenCL. Both follow a similar programming model but CUDA is limited to Nvidia Hardware, while OpenCL is an open standard. According to research of Fang, Varbanescu and Sips in [FVS11], similar performance can be achieved with either of them. Different programming platforms are introduced in this chapter, whose performance for graph algorithms will be compared in Chapter 5 to find the most promising approach with respect to performance and programming ease for the GPU implementation. CUDA has been chosen over OpenCL since the measurements for this thesis will be performed on Nvidia hardware and CUDA gives access to specific features of that hardware. The comparable reference code is written in CUDA too, which simplifies a comparison.

2.2.1 CUDA

The Compute Unified Device Architecture (CUDA) is a parallel computing platform revealed by Nvidia in 2006, which supports the development of non-graphic applications utilizing the power of GPUs. It was the first platform fitting the needs of general purpose computations on GPUs (GPGPU) by eliminating the necessity to translate general computational problems into a graphic-related problem [Bar15, p. 393]. CUDA can be used in C/C++ and Fortran programs, but in this thesis, CUDA refers to the C/C++ language-extension. A number of syntactic elements and key words to express massive parallelism and its mapping to the GPU hardware is introduced as well as two APIs. To compile CUDA code, the Nvidia compiler *nvcc* is used. It distinguishes between host code, which is the code to be executed on the CPU, and device code that will be executed on a Nvidia GPU. The custom functions that will be executed on the GPU are often called a kernel.

The CUDA example code in Listing 2.1 clearly follows the typical structure of copying data from the CPU to the GPU (line 19-20), do computations on the GPU (line 23) and copy the results back (line 26). The last parameter of the `cudaMemcpy` function states the direction of the copy. In line 19 data is copied from `x` to `d_x`, which is named with respect to the naming convention of starting pointer variables to the GPU memory with a "d_". The `cudaMemcpy` function, as well as `cudaMalloc`, is provided by the CUDA Runtime API, which is built on top of the Driver API. The Driver API is considered to be closer to the hardware and includes access to more features than the Runtime API. Nonetheless, the Runtime API is used in the majority of cases because it provides the commonly used functions with an easier interface and negligible performance loss. The function qualifier `__global__` in line 1 is a keyword introduced by CUDA and denotes that the `saxpy` function in line 2-6 will be called from the CPU but executed on the GPU. `saxpy` (single precision a times x plus y) computes the arithmetic expression $a \cdot x + y$ with two vectors x and y and a factor a. Since it is possible to compute every entry in the result vector independently from any other, this is a common example used in parallel programming. On the GPU, every entry is computed by another thread. In order to achieve this, as many threads as there are entries in x and y have to be launched. The number of threads to launch on the GPU is specified through the launch configuration given after the kernel name. As can be seen in line 23 the launch configuration is framed with three angle brackets.

```
1  __global__
2  void saxpy(float a, float *x, float *y)
3  {
4      int i = blockIdx.x*blockDim.x + threadIdx.x;
5      y[i] = a*x[i] + y[i];
6  }
7
8  void main()
9  {
10     int N = 1<<20;
11     float *x, *y, *d_x, *d_y;
12     ...
13
14     // alloc memory on the GPU
15     cudaMalloc(&d_x, N*sizeof(float))
16     cudaMalloc(&d_y, N*sizeof(float))
17
18     // copy from CPU to GPU memory
19     cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
20     cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);
21
22     // call the GPU kernel
23     saxpy<<<2048,512>>>(2.0, d_x, d_y);
24
25     // copy result back to CPU memory
26     cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
27 }
```

Listing 2.1: CUDA SAXPY example[3]

CUDA organizes threads into 3-dimensional blocks. These blocks are arranged in a grid that can take up to three dimensions as well. The first parameter of the launch configuration in line 23 specifies the number of blocks in the grid and the second parameter specifies the number of threads to launch per block. In this example, both are one-dimensional, a call with `<<<dim3(2048,0,0), dim3(512,0,0)>>>` would be equivalent. Every thread has access to built-in variables that indicate the thread's position. In line 4 these variables are used to determine which entry has to be calculated by the executing thread. The kernel launch configuration in CUDA is directly related to the execution model described in Section 2.1.1. The blocks of threads, whose total number can be calculated from the first launch configuration parameter, are scheduled to run on a streaming multiprocessor. As the threads of a block are divided into warps, the total number of threads per block should always be a multiple of 32, so there are no underutilized warps. Furthermore, when choosing a launch configuration it should be considered that a block should contain many warps such that latency of global memory access can be hidden successfully. A reason not to choose the maximum number of threads per block, which is 1024 threads, could be that not enough resources are available, namely the amount of shared memory or the number of registers. A kernel call using too much shared memory per block will refuse to run and is, therefore, easy to detect. If more registers than available are used, their content will be stored in local memory, which is a fraction of the global memory only accessible by one thread. This will result in a poor performance and can be avoided by compiling with the `-ptxas-option=-v` option to get information on a kernel's memory usage [NVI15, p. 416].

[3]This example is taken from [Har12c] with slight modifications.

Earlier, kernels could only be launched from the CPU. This changed with the release of CUDA 5.0 as it introduces the Dynamic Parallelism (DP) feature that enables kernels to be launched from within a CUDA kernel. The same syntax that specifies the launch configuration enclosed with angle brackets is used in the CPU and GPU code. The launched child kernel will run asynchronously but always finishes its execution before the parent kernel returns [Adi14].

2.2.2 Thrust

Thrust is a template library for the C++ programming language and provides a selection of parallel algorithms. It follows the concept of the C++ Standard Template Library (STL) to provide generic implementations of algorithms that can be adapted to the user's needs easily. Since the concept is already known to C++ programmers there is no new language or programming paradigm to learn. Thrust was intended to be a high-level CUDA front-end in the first place but turned into a rather general library that features parallel programming in many different ways. This includes a CUDA back-end as well as an OpenMP or an Intel Building Blocks back-end. Therefore, the same code can be used to target different hardware architectures.

```
1  struct saxpy
2  {
3      float a;
4
5      saxpy(float factor_a): a(factor_a)
6      {}
7
8      __device__
9      int operator()(float &x, float &y)
10     {
11         return a * x + y;
12     }
13 };
14
15 void main()
16 {
17
18     int N = 1<<20;
19     thrust::host_vector x(N), y(N);
20     ...
21
22     // copy from CPU to GPU memory
23     thrust::device_vector d_x = x;
24     thrust::device_vector d_y = y;
25
26     // calculate saxpy on GPU
27     thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), saxpy(a));
28
29     // copy result back to CPU memory
30     y = d_y;
31 }
```

Listing 2.2: Thrust SAXPY example[4]

[4]This example is taken from [Hwu12] with slight modifications.

Thrust provides two containers of which one is using host memory, the `thrust::host_vector`, while the other, the `thrust::device_vector`, refers to memory in the GPU address space. To copy data from a `host_vector` to a `device_vector` a simple assign statement can be used, which can be seen in line 23 and line 24 of Listing 2.2. The strengths of Thrust are generic implementations of fundamental parallel algorithms such as sorting, reduction or transformation. These can be extended with user defined behavior in order to match specific requirements. As visible in line 1-13, the user defined behavior is specified through a `struct` that overloads the ()-operator. This construct is called a functor in Thrust's terminology. The `__device__` function qualifier is used, which is also available in CUDA and denotes the functor will be called from within a GPU kernel.

`thrust::transform`, called in line 27, takes two input vectors, executes a function with two parameters using the entries of both input vectors for every index as parameters and stores the result at the corresponding index of a result vector. The first input vector passed to the function is `d_x`. The starting point of the vector `d_x` as well its end is given to determine how many elements the transformation will be executed on. The vector `d_y` is used as second input vector as well as result vector, which leads to an in-place transformation. The most important difference to a CUDA implementation of SAXPY is that no launch configuration has to be specified. Thrust completely abstracts from the fact that the function will be executed in parallel and can be executed on a GPU. The exact same code can even be compiled using an OpenMP back-end or a CUDA back-end [Bar15, p. 527].

The high-level character of the Thrust library eases GPU programming significantly if an algorithm is to be implemented that fits any of the provided templates. In those situations Thrust provides good performance without the need to dive into the details of a CUDA implementation. On the downside, Thrust does not enable the user to do fine-grained optimization and does not give access to all CUDA features, for example, shared memory [Hwu12], which is important for some kind of algorithms. In these cases, it is very helpful that Thrust maintains interoperability with CUDA. A `thrust::device_vector` can be casted to a usual pointer to GPU memory with `thrust::raw_pointer_cast()`. Afterwards it can be passed to any CUDA kernel. Mixing Thrust and CUDA often seems to be a good approach to get the most performance out of GPU code.

2.2.3 OpenACC

OpenACC follows the well-established approach of OpenMP to provide compiler directives, which are used to automatically generate code that can be executed on a GPU or other accelerator cards. The current OpenACC 2.0 specification was released in June 2013 under the cooperation of many partners from industry and academia. The open standard aims for code portability between different hardware configurations including hardware from different vendors.

Only one line has to be added to a sequential SAXPY implementation to achieve off-loading the loop execution to a GPU with OpenACC. In the example, shown in Listing 2.3, this is line 3. A simple `#pragma acc parallel loop` causes a number of complex tasks to be performed. This includes the copying of all data, which is accessed in the following loop,

```
1  void saxpy(int n, float a, float *x, float *y)
2  {
3      #pragma acc parallel loop
4      for (int i = 0; i < n; ++i)
5              y[i] = a*x[i] + y[i];
6  }
7
8  void main()
9  {
10     int N = 1<<20;
11     float *x = new float[N];
12     float *y= new float[N];
13     ...
14
15     saxpy(1<<20, 2.0, x, y);
16 }
```

Listing 2.3: OpenACC SAXPY example[5]

to GPU memory as well as the generation of a GPU kernel that will be launched and the back-transfer of all computed results to CPU memory. As all the GPU related work is done by the compiler, OpenACC is a very comfortable and easy to use programming platform, which is especially true for programmers that are familiar with OpenMP. Unfortunately, it is likely that a situation occurs, in which the completely unassisted parallelization by the compiler fails with regard to performance. This is shown in [Har12a] with an OpenACC parallelization of the Jacobi method, which iteratively solves a linear equation. The key issue for the example is the code structure with nested loops, which is displayed in Listing 2.4. The outer loop controls the number of iterations and contains another loop that is parallelized with the `#pragma acc parallel loop` directive. The compiler parallelizes the inner loop correctly but enforces the data to be copied back and forth between the CPU and GPU for every iteration of the outer loop. The performance can be improved by adding a data directive, `#pragma acc data copy(...)`, surrounding the outer loop, which ensures the variables named in the brackets will be copied to the GPU before the outer loop begins and copied back afterwards. A data directive can also be used to hint the compiler a variable is pointing to GPU memory already, which obviates the need for copying it before executing code on the GPU. This enables interoperability with other programming techniques. Further tuning is possible with directives that influence the launch configuration of the generated kernel as it is shown in [Har12b]. It can be concluded that regardless of which programming platform used, it is necessary for the programmer to have deeper knowledge on the details of how code is executed on a GPU in order to exploit the possible performance.

```
1  while (iteration > 0)
2  {
3      #pragma acc parallel loop
4      for (int i = 0; i < n; ++i)
5      {
6          ...
7      }
8      --iteration;
9  }
```

Listing 2.4: OpenACC nested loop structure of the Jacobi method

[5] This example is taken from [Har12c] with slight modifications.

2.3 Modern CPU Architecture

The previous sections explained the GPU hardware architecture that aims to achieve high performance through massive parallelism. In contrast, the CPU hardware is designed for fast processing of a sequence of instructions. Since a CPU contained a single processing unit for decades, this focus is not surprising. As a result, the performance of a CPU could only be increased if the hardware was optimized to process instructions faster than before. A number of complex architectural optimizations have been introduced such as pipelining, which leads to a higher instruction throughput, the installation of multiple functional units and out-of-order execution, which tries to avoid idle times due to memory access latencies. To do this, instructions that turn up later in the instruction stream are executed if their parameters reside on-chip already [HW11, p. 8]. Additionally, a cache hierarchy was introduced with graded access times for different cache levels to solve the problem of a much faster processor than memory subsystem. Along with this a prefetching mechanism was established to avoid cache misses. All those optimizations improved the performance significantly, but the most important factor for the performance increase of CPUs was a higher clock speed. According to [Rau13, p. 10] the annual increase of the clock speed came to about 40% between 1987 and 2003. Unfortunately, the higher frequencies were accompanied with elevated power consumption and heat generation, which made a further increase impossible. For this reason, CPU clock frequencies are stable at a level of 2 to 3 GHz [HW11, p. 4]. Further improvements in the performance were only possible through placing multiple independent CPU cores inside a CPU. Such a CPU is called a multicore CPU. In addition to that, the instruction set of the x86 architecture has been extended with Single Instruction Multiple Data (SIMD) operations. These operate on a vector and apply the same instruction simultaneously to a number of data elements. In 1999, the Streaming SIMD Extensions (SSE) have been introduced that work on 128-bit registers. The Advanced Vector Extensions (AVX) provide 3-operand versions of the SSE instructions and enlarge the size of the registers up to 256-bit for AVX2 [Rau13, p. 125][Kus14, p. 327].

To fully utilize the capabilities of a multicore CPU, the workload of a program has to be split across a number of threads that execute on the different cores in parallel. The responsibility for this falls to the programmer. An established platform for parallel programming with threads is OpenMP, which is a portable standard that provides compiler directives along with several library functions and environment variables. An OpenMP implementation of SAXPY replaces line 3 of the OpenACC implementation in Listing 2.3 with `#pragma omp parallel for`. The code generated by the compiler follows the fork-join pattern [Ben15]. A program starts with one master-thread, that creates other threads at the beginning of the parallel region. These execute the code within the parallel region on different cores in parallel. The same memory can be accessed by all threads, which leads to the necessity to coordinate memory access. For this reason, OpenMP provides constructs such as critical sections (`#pragma omp critical`) and reductions (`#pragma omp reduction (variable: operator)`). At the end of the parallel region only the master-threads continues to exist. All details regarding the thread creation and the data partitioning over the threads are hidden from the programmer, who can focus on identifying parallelism in the code.

2.4 Message Passing Interface

The modern supercomputers are composed of a great number of compute nodes that are connected through a network. Typically every node has two or four sockets that are equipped with a multicore CPU. As the memory of different compute nodes is distinct from each other, the communication cannot rely on shared variables but has to be performed with messages transferred over the network. The de facto standard for distributed memory programming is the Message Passing Interface (MPI), which provides a set of functions for communication. Different implementations of the specification exist, for example, OpenMPI and MVAPICH.

A compute node of a GPU cluster is equipped with additional GPUs that are connected to the node through PCIe. As described in the earlier sections of this chapter, a GPU program always contains a CPU part that controls one or more GPUs. The exchange of data between GPUs connected to the same node can be performed without CPU assistance through technologies like GPUDirect. This allows the data of GPUs, which are connected to the same PCI switch, to be copied directly between the GPUs' global memories. Data exchange between two GPUs that are not connected to the same node requires the call of MPI functions. For GPU clusters CUDA-aware MPI is used, which enables passing pointers to GPU memory to the functions and obviates the need of copying the data to CPU memory first. The implementation of CUDA-aware MPI functions is still in progress, so far, most of the blocking functions have been implemented but not all asynchronous functions are available yet.

Summary *In this chapter, the parallel hardware architecture of the GPUs has been explained. A closer look at the execution model has revealed that applications require a substantial degree of parallelism to exploit the capabilities of a GPU. The CPUs follow a different approach that focuses on fast sequential execution and a comparably low degree of parallelism as the number of cores per CPU is quite limited.*

3 Parallel Graph Algorithms

In this chapter the characteristics of graph algorithms are identified and their impact on the choice of a suitable hardware configuration is explained. Additionally, the Graph500 benchmark is introduced and finally an overview of the relevant research that has previously been done on the subject will be given.

3.1 Characteristics of Parallel Graph Algorithms

Graph algorithms have always been an important topic in computer science, as "hundreds of interesting computational problems are couched in terms of graphs" [CLRS09, p. 587]. A graph G=(V,E) is a formal abstraction of objects and their relations among each other. These objects are represented by the vertices of the graph, a set V, and relations between vertices are specified through a set of edges E, which contains pairs of vertices that are connected to each other. Due to their capability of expressing complex dependencies, graphs are found in various application areas. A graphical illustration of a graph can be found in Figure 4.3 in Section 4.2.

It is very common for graph algorithms to compute information related to the underlying graph's structure. As the structure of a graph can only be detected by traversing it in a systematic way, this is the main part of many graph algorithms [LGHB07]. Breadth-first Search, Single-source Shortest Path (SSSP) and Connected Component Labeling (CCL) can serve as an example for this attribute. To traverse a graph, the neighborhood of the vertices is examined through following the out-going edges of a vertex to find all adjacent vertices. The order, in which vertices are chosen to be the starting point for further exploration of the graph structure, depends on the algorithm, but often a great number of vertices has to be processed in the same execution step. As these usually can be processed independently, many graph algorithms benefit from parallelization. This is important because graphs today contain billions of nodes and edges, which leads to the situation that large graphs can not be stored in one single computer's main memory and not be processed by a single processing unit [KKI14]. Therefore, graph data partitioning is the first challenge to be addressed in large-scale graph processing. As usually little knowledge about the graph structure is present at the time the graph is distributed over multiple processing units, an uneven distribution is very difficult to avoid. The problem of unbalanced workload is inherent to parallel graph algorithms as it does also appear during computation within a single processing unit. The reason for it is that "graph problems are typically unstructured and highly irregular" [LGHB07] and computations in graph algorithms directly depend on the graph data. This is also evident from the fact, that the effort to explore the neighborhood of a vertex is dependent on the number of out-going edges. The same algorithm will perform different computations on different input data as the computation steps usually depend on the result of previously processed graph data.

Following the edges of a graph to explore its structure corresponds to many memory access operations. This results in a dominance of these operations in comparison with computational operations [KKI14]. Due to the irregular structure graphs, the memory access patterns have low locality [LGHB07], which results in high memory access costs. The reason is that modern processors try to accelerate memory access through exploiting locality. This is done with a cache hierarchy on CPUs and coalesced memory access on GPUs.

3.2 Mapping to GPU and CPU Clusters

While the last section has presented the characteristics of parallel graph algorithms, this section aims to point out the potential and difficulties of choosing a CPU or GPU cluster to run graph algorithms. For graph algorithms, the distribution over multiple nodes is easily possible through a partitioning of the graph data. Splitting up the graph data is a necessity as the amount of data easily exceeds the capabilities of a single compute node. As described in Section 2.1.2 the amount of memory available on a GPU is far less than on a CPU. Consequently, the number of GPUs necessary to fit the graph data onto on-board memory will be much higher than the number of CPUs. A CPU contains multiple CPU cores, therefore, to fully utilize the CPU hardware computations on a single CPU should be parallelizable. As described before, this is possible for graph algorithms as vertices often can be processed in parallel. However, the presence of "enough" parallelism is even more important if a GPU is used. To exploit the capabilities of a GPU, a much higher degree of parallelism is required as it would for a CPU. If this constraint is met, a GPU is expected to outperform a CPU. Since the amount of parallelism depends on the input graph, no statement which hardware configuration will be favored can be made regarding this property.

Accelerating HPC applications with GPUs has been an emerging trend throughout the last years, "due to their high memory and computational throughput, low costs and power efficiency" [ZLL14]. In respect to memory bandwidth, the current GPUs exceed high-end CPUs by far: 240 GB/s bandwidth to off-chip memory on a GPU compare to 76.8 GB/s memory bandwidth for a recently released CPU from Intel[1]. In the previous section, it is revealed that memory access time is very critical to graph algorithm performance. Nevertheless, a profound evaluation which hardware configuration has more potential to provide good graph algorithm performance can not be reasoned with the theoretical maximum value as these stem from optimum conditions. Because memory access is unstructured, those values will never be seen when running graph algorithms. Computational throughput in the above quote refers to the number of Floating-point Operations Per Second (FLOPS). A theoretical peak performance of 2.91 TFLOPS of a Nvidia K80 [Gün14], which results in 1.455 TFLOPS for both contained GK210 GPUs, exceeds an Intel Broadwell CPU[1] with 604.8 GFLOPS[2]. As cost and energy efficiency are usually calculated with regard to the floating-point performance as FLOPS/Watt and FLOPS/€, it is not hard to imagine a GPU achieves better values than a CPU. As visible from the fact that the TOP500 list ranks supercomputers according

[1] Intel® Xeon® CPU ES-2695 v4 `http://ark.intel.com/de/products/91316/Intel-Xeon-Processor-E5-2695-v4-45M-Cache-2_10-GHz`

[2] 2.1 GHz × 18 CPU cores × 16 instructions per cycle = 604.8 GFLOPS

to their FLOPS performance in running a High Performance Linpack (HPL), this is a very common measure to quantify a system's performance. A reason for this is that the main part of many scientific applications are floating-point calculations [HW11, p. 3]. This does not apply to graph applications, which invalidates the FLOPS performance as a good measure to estimate how well a graph application will run on a system. For this reason, the cost and energy efficiency will be evaluated based on the achieved TEPS values in Section 6.3.

3.3 Graph500 Benchmark

The Graph500 list, which was introduced in 2010, ranks supercomputers according to their performance on running a BFS on a large-scale graph. It was established to provide a metric that gives useful information on the suitability of supercomputing systems for data-intensive applications [Gra]. A BFS starts at one source vertex and visits all vertices of the connected component. In every iteration, all direct, unvisited neighbors of the current set of active vertices are computed. They become the new set of active vertices, often denoted as the frontier, in the next iteration. The graph traversal stops if all reachable vertices are visited, which results in an empty frontier. Performance on running a BFS is measured in Traversed Edges Per Second (TEPS).

The Graph500 specification includes the following parts:

- Edge list generation
- Graph construction
- Search key selection
- Breadth-first Search
- Validation of results
- Performance statistics

The edge list consists of tuples of vertices that are connected by an edge. A Kronecker graph generator is used to generate an undirected graph that may contain self-loops and multiple edges between the same two vertices. Its degree distribution follows a power law, which means that there is a great fraction of vertices that are connected to many other vertices, while there are only a few vertices having a low degree. It is permitted to run the generator in parallel, but the generated data should not contain any locality. The overall number of vertices N is determined by an input parameter called SCALE with $N = 2^{SCALE}$. The number of edges in the edge list is given by $N \cdot edgefactor$, whereas 16 is a commonly used value for the edgefactor parameter.

During the graph construction phase, an arbitrary graph representation is built up from the edge list. This representation is used during 64 BFS runs. The source vertices for these runs

are sampled randomly from the available vertices with the condition that the source vertex has to be connected to at least one other vertex. All BFS runs are executed on the same graph data. The output of one BFS is a parent array, that denotes the predecessor of each vertex. The predecessor of the source vertex is the source vertex itself. Vertices that have not been visited during graph traversal, because they are not in the same connected component, are marked with a "-1". There is not one unique correct result because two vertices can discover the same vertex in the same BFS iteration and the parent vertex assignment is not further specified. Therefore, no check against a reference is employed, but a validation verifies the result by testing on a number of properties that have to be present if the result is valid. In the end statistics on the number of traversed edges, the time needed for BFS runs and reached TEPS are provided. Additional information about input parameter values and time measured for the graph construction phase are given as well.

Graph500 seems to be a good choice for an application to be implemented in the course of this thesis for two major reasons. On the one hand, this is the fact, that Breadth-first Search is the underlying graph algorithm. BFS is one of the most basic and simple graph algorithms, but "is perhaps the most challenging parallel graph problem because it has the least work per byte" as is stated in [FDB+14]. A low number of computations per vertex amplifies the importance of fast memory access to prevent the processing units from underutilization. Additionally, the other main characteristics of graph algorithms such as unbalanced workloads and a dependency on the input graph data are inherent in BFS. For this reason, it can be assumed, that results on BFS can be transferred to a wide range of other graph algorithms. On the other hand, Graph500 has the advantage of being a benchmark with a detailed specification. This enables a comparison not only between the implemented CPU and GPU version but also with published results.

3.4 Related Work

The specification of Graph500 [MWBA10] was published together with reference code. The current reference implementation (Version 2.1.4) provides a sequential, OpenMP, XMT and MPI version which can be found on the Graph500 website[3]. Checconi et al. have published an improved MPI version for CPU clusters in [CPW+12]. They use 2D partitioning of the graph data, which is explained further in the next chapter, and introduce a communication pattern following the rows and columns of the adjacency matrix. The multi-node GPU implementations of Ueno and Suzumura [US13] as well as the one by Fu, Dasari, Bebee, Berzins and Thompson [FDB+14] follow the same overall structure and focus on local GPU specific optimization. A greater number of publications targets a single GPU implementation of BFS such as [MGG12], [LWH10] and [ZLL14]. The first one by Merrill, Garland and Grimshaw focuses on load-balancing through warp cooperation, while the other ones propose fast GPU queue implementations.

Many papers on GPU implementations include a section with a comparison to a CPU implementation and speed-ups of 100-200x are stated. In their paper [LKC+10] Lee et al. from the Intel Corporation showed these values cannot be achieved in comparison against

[3] `http://www.graph500.org/referencecode`

a fully tuned CPU implementation. A similar finding is made by Jaros and Pospichal [JP12]. As these authors focus on a fair comparison of performance only, they do not include programming effort or cost and energy efficiency in their comparison. With regard to programming effort some recently developed frameworks for graph algorithms on GPUs have to be mentioned. Very promising are Gunrock [PWW+15] and MapGraph [FPT14] as these aim to provide a graph algorithm building kit, that eases the development of custom graph algorithms optimized for GPUs. The importance of the topic can also be seen from the fact, that Nvidia announced a new graph library called nvGRAPH[4] to be released with the next CUDA versions. Unfortunately, none of the existing frameworks targets multi-node configurations and for this reason, they have not been taken into account for the comparison of this thesis.

Summary *This chapter described the general graph algorithm characteristics that can also be found in BFS, which is the basis of the Graph500 benchmark. The contemplation of the CPU and GPU hardware architectures with regard to graph algorithm requirements reveals that it is difficult to draft an expectation regarding the finally delivered performance of the different hardware configurations.*

[4] `https://developer.nvidia.com/nvgraph`

4 Parallel Breadth-First Search

This chapter introduces the parallel BFS algorithm implemented for this thesis. It includes an explanation how the algorithm works with an example and a pseudo code. Information on the graph partitioning over the processes and representation is given in the first part of this chapter.

4.1 Graph Partitioning and Representation

To perform a parallel Breadth-first Search on a large-scale graph, the graph data has to be partitioned and distributed over the processes participating in the search. There are two commonly used strategies to assign parts of the graph data to the processes, which are named 1D partitioning and 2D partitioning. 1D partitioning assigns each vertex to a process. For every assigned vertex the owning process has information on all the vertices that are adjacent to it. It will also keep track of the states associated with that vertex. Depending on the algorithm used, this can include the information whether a vertex has been visited during the graph traversal, information regarding the predecessor of a vertex or the level in which a vertex was discovered. During graph traversal, a process that discovers a vertex, owned by another process, has to notify the owning process to change the vertex's state. Therefore, 1D partitioning leads to a communication structure that is determined by the graph structure and has a high overall communication cost.

2D partitioning follows another approach, which is based on a grid of processes that communicate in a regular way. This is achieved by assigning parts of the graph's adjacency matrix to the processes, which is illustrated in Figure 4.1. The example uses four processes P_0 to P_3 that are organized in a 2×2 grid. The emphasized grid lines indicate how the adjacency matrix is split across the processes. For example, the edge $(2, 0)$ is assigned to process P_0, whereas $(2, 4)$ is assigned to P_1. Distributing the edges and not the vertices leads to the situation that information regarding the same vertex is present on multiple processes and needs to be synchronized in every BFS iteration. With 2D partitioning, it is determined which processes need to synchronize through the way the processes are arranged. For the most part, communication is restricted to processes located in the same row or column. This will be explained further in the next section. Reducing the number of possible communication partners and forcing the communication pattern into a regular structure, reduces the complexity of the algorithm and is beneficial for a GPU implementation. This is, because the inter-node communication is issued from the host CPU and, therefore, a regular structure obviates the need for communication between GPU and CPU to determine which data that is located on the GPU has to be sent to other processes. 2D partitioning is used by the state of the art multi-node CPU and GPU implementations [CPW+12] [US13]. Therefore,

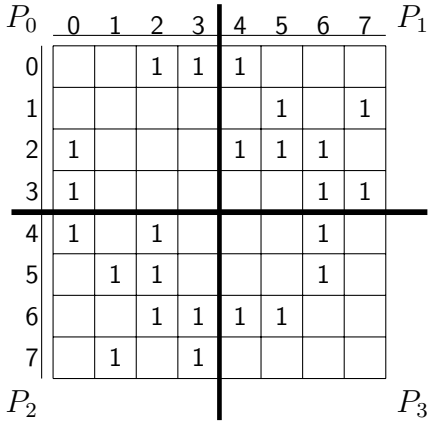

Figure 4.1: 2×2 grid of processes

the choice of this partitioning approach for both implementations does not privilege one over the other in the final comparison.

The decision of how to represent graph data is usually a decision between an adjacency list and adjacency matrix based approach. A representation with an adjacency matrix takes up a lot of memory, $O(|V|^2)$, which is especially adverse for sparse graphs. In return, such a representation gives fast access to the information whether a specific edge exists or not [CLRS09, p. 589]. As the BFS algorithm does not make use of this property and very large and sparse graphs are considered here, an adjacency list based representation is the better choice. This is because the memory requirements grow linearly with the number of edges in the graph and the adjacency list based approach enables fast iteration over all neighbors of a vertex, which is necessary during graph traversal. Allocating a separate list for every vertex will be very inefficient on the GPU as the addresses of the first list element will "always be aligned to at least 256 bytes" [NVI15, p. 80]. Due to padding a lot of memory would be left unused. For this reason a concatenated adjacency list together with an indices array will be used, equivalent to the representation in [HN07]. An illustration of the graph representation is given in Figure 4.2. It depicts the same graph as the adjacency matrix above. However, for the conduction of the parallel BFS algorithm that is explained in the next chapter, every process builds up a graph representation that only represents its assigned part of the adjacency matrix. The *indices* array contains a value for every vertex, which

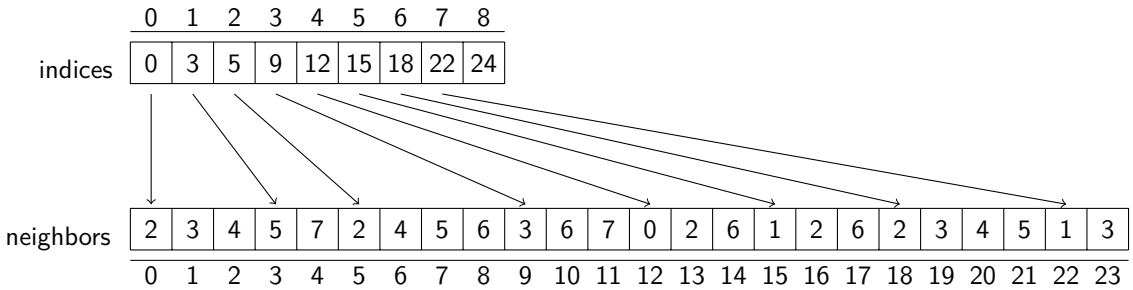

Figure 4.2: Graph representation

specifies the index of the start of the adjacency list in *neighbors*. Indices are used instead of pointers to enable copying the structure between CPU and GPU memory.

4.2 Algorithm Structure

Both, the GPU-enabled versions of Graph500 and the CPU-only version, use the same basic algorithm structure, which will be explained in this section. The behavior of the algorithm has been split into a number of functions regarding computations and communication. Listing 4.1 presents the layout of these functions in pseudo code. This section aims to provide an overview of the inter-process communication patterns and the computations that are performed. Therefore, no parameters are listed for any of the functions as those can be different for the GPU and CPU implementation and this section focuses on the functionality of the algorithm. The implementation details will be described in the next chapter. The overall structure of the algorithm implemented for this thesis mainly follows the approach described in [FDB+14] with a difference in the communication pattern. Through an additional communication step Fu et al. ensure that exactly one process will compute the predecessor of a vertex. As the additional communication has proven to be more overhead than some unnecessary computations for my implementation, this step is omitted.

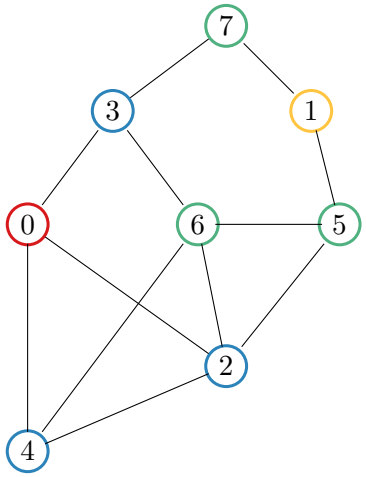

Figure 4.3: Graph

Figure 4.3 shows a small graph with eight nodes, that will be used as an example during this chapter. The nodes, numbered from 0 to 7, are colored depending on the iteration, in which they are discovered in a BFS starting at vertex 0. Vertices that are found in the same iteration are said to be found at the same BFS level. For example, all blue colored vertices are found at the second BFS level. The parallel BFS algorithm follows a level-based approach, which means that all processes have to be done with exploring the current level until any process proceeds with the next.

The overall work is split in a graph traversal part (lines 8-24 of Listing 4.1) and final result computation (lines 27-29). The result will be an array that contains information on the predecessor of every vertex. The algorithm is based on the idea that one row of processes

```
1  // set root in IN if root falls in range of process
2  // set root in VISITED if root falls in range of process
3
4  int level = 0;
5  int frontier = true;
6
7  //graph traversal
8  while(frontier)
9  {
10     update_levels_with_in();
11     compute_expansion();
12     update_assigned();
13
14     allreduce_out();              #row communication
15     update_visited();
16     update_levels_with_out();
17     frontier = vertices_in_out();
18
19     allreduce_frontier();         #global communication
20     broadcast_next_in();          #column communication
21
22     reset_out();
23     ++level;
24 }
25
26 //computation of predecessors
27 compute_predecessors();
28 allreduce_results();             #row communication
29 gather_result();                 #diagonal communication
```

Listing 4.1: Structure of the parallel Breadth-first Search algorithm

collectively discovers the predecessors of a set of vertices. For the example, provided in Figure 4.4, this means that P_0 and P_1 are responsible for detecting the predecessors of vertices 0-3 while P_2 and P_3 do the same for the vertices 4-7. These numbers correspond to the vertical numbering of the rows in the example. Each BFS iteration, which corresponds to one execution of the while loop in Listing 4.1, builds up a set of vertices, named *out* in the following, that includes the vertices, which were discovered during that iteration. This set is the starting point for further discovering in the next iteration and is then called *in*. All processes in one row build up one *out*, while *in* is shared by all processes in one column.

Figure 4.4 shows the initial state of the arrays *visited*, *assigned* and *levels* exemplary for process P_0. The array *levels* is as long as the number of matrix columns plus the number of rows that are assigned to a process. For the purpose of demonstrating the array has been split in Figure 4.4. The first part of the *levels* array of P_0 is depicted vertically on the left side, while the second part is depicted horizontally above the adjacency matrix. As can be identified from Listing 4.1, `update_levels()` is called twice in each iteration. The first call in line 9 updates the horizontal part and the second call in line 16 updates the vertical part with the current value of `level`. In the *visited* array of a process vertices are marked that have been discovered by another process of the same row or the process itself. In contrast, a vertex is marked in *assigned* only if it was discovered by the process itself. As mentioned in the previous section, information is duplicated across the processes. P_0 and P_1 maintain the information regarding the same vertices (0-3) in *visited* and the vertical part of *levels*. These arrays have to contain the same values on all processes located in the same row, while values of *assigned* are independent. Additionally, the horizontal *levels* array should contain

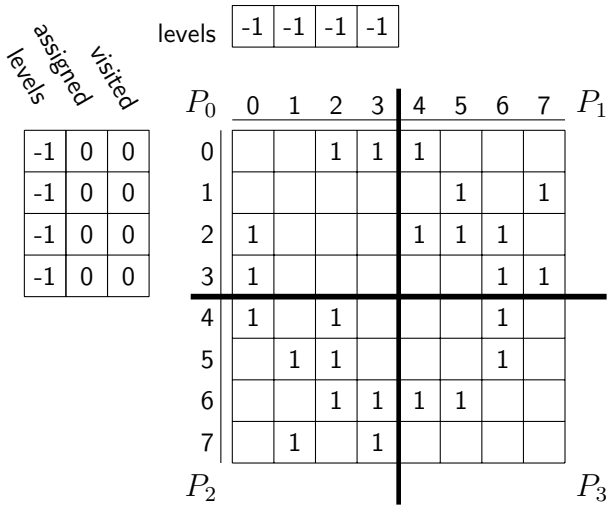

Figure 4.4: Before start of BFS

the same values for processes in the same column such as P_0 and P_2. During the predecessor computation phase, a process will compute a predecessor only for those vertices that have been set in *assigned*.

Before the graph traversal starts, the root vertex has to be set in the *in* array and gets marked as visited. This corresponds to the comments at the top of Listing 4.1 and is illustrated in Figure 4.5 for root vertex 0. The *in*-vertices are represented as red arrows pointing to one

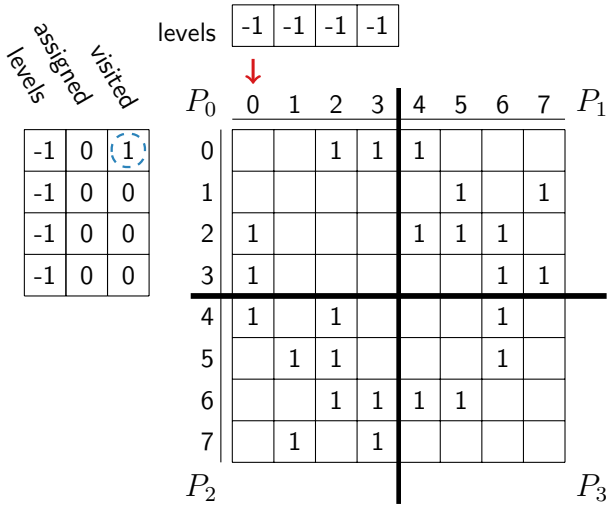

Figure 4.5: Set values for root vertex 0

column and are set for all processes in that column. The entry in *visited* that corresponds to the root vertex 0 has been circled. Not only P_0 will mark vertex 0 as visited but also P_1.

The first step in each iteration is to update *levels* for all vertices in *in*. In the first iteration *in* contains only the root vertex. The corresponding entry in the horizontal *levels* array has been marked with a circle again in Figure 4.6 . Additionally the results of the func-

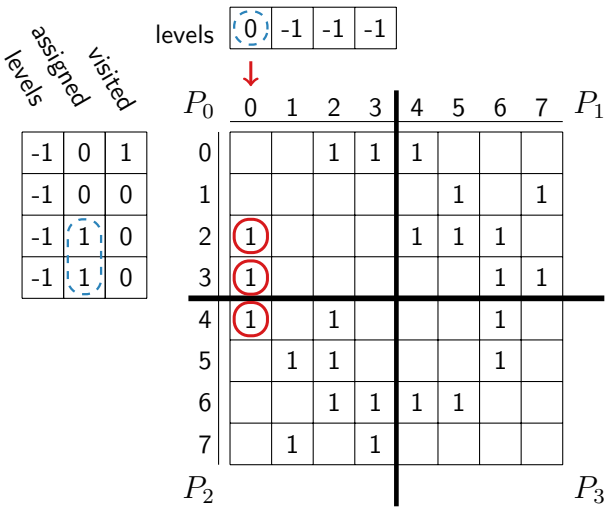

Figure 4.6: Results of `update_levels()`, `compute_expansion()` and `update_assigned()`

tions `compute_expansion()` and `update_assigned` are depicted. `compute_expansion()` aims to discover all unvisited vertices reachable from the vertices contained in *in*. The vertices that are found by `compute_expansion()`, are highlighted with a red circle. Those three vertices correspond to blue colored ones in the graph depicted in Figure 4.3. The vertices 2 and 3 will be set in the local *out* array of P_0 and vertex 4 will be set in local the *out* array of P_2. The call of `update_assigned` in line 12 of Listing 4.1 sets the corresponding entry in *assigned* if a vertex is set in the local *out* array. This is illustrated through the blue marking on the left side of Figure 4.6. After the graph traversal phase, an entry of *assigned* will be set if the corresponding vertex was discovered by the process in any of the BFS iterations. There is a possibility that the same vertex is set in the local *out* array of multiple processes in the same row. In this case, all of those processes mark the vertex as assigned and will calculate a predecessor. All those predecessors are valid as Graph500 does not specify which vertex to choose as predecessor in the case of concurrent discovery by multiple processes.

The next step is the concatenation of *out* for all processes in one row. This refers to the call of `allreduce_out()` in line 14 of Listing 4.1. All vertices contained in the concatenated *out* are represented by a green, horizontal arrow in Figure 4.7. The combined *out* of P_0 and P_1 contains two vertices, the *out* of P_2 and P_3 contains one vertex. In this iteration, only the processes P_0 and P_2 contribute to the concatenated out. The row-wide *out* is used to update *levels* and *visited* for all processes in the row. The result of these function calls are marked in the horizontal *levels* and *visited* array.

Every process checks if there is any vertex set in *out*. The result of this check is assigned to `frontier` in line 17. If no vertex is set in any of the row-wide concatenated *out* arrays there are no more reachable, unvisited vertices and the termination condition is met. A global `MPI_Allreduce()` on the loop control variable `frontier` ensures all vertices have the

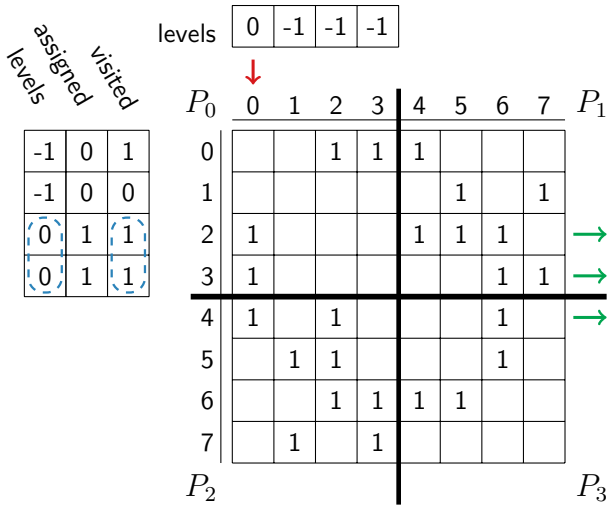

Figure 4.7: Results of `concat_row_out()`, `update_visited()` and `update_levels()`

correct value of `frontier`. The computed row-wide *out* arrays are used as *in* in the next iteration. Therefore, they are broadcasted to all processes in the column by the processes in the diagonal. This relationship becomes clearer if the green arrows in Figure 4.7 are compared to the red ones in Figure 4.8, which presents the setup for the second iteration. The *in* vertices correspond to the vertices that are directly reachable from the root vertex 0 as can be identified from the graph in Figure 4.3.

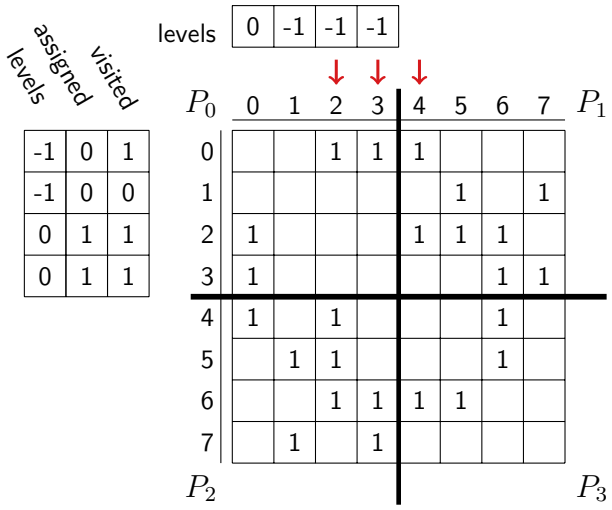

Figure 4.8: Setup for the second iteration

After the graph traversal, each process computes the predecessors for its assigned vertices. This information can be computed from the horizontal and vertical part of *levels*. The vertices 2 and 3 have been discovered in level 0, which can be deduced from the 0 entries in the

corresponding fields of the vertical part of *levels*. A vertex that was contained in the *in* array in the same iteration is a valid predecessor. As one can see in the horizontal part of the *levels* array in Figure 4.8, this is true for vertex 0. This vertex will be set as a predecessor for vertex 2 as well as 3 in `compute_predecessors()`. These results are merged for the processes in a row and then gathered to build up the global result.

Summary *The adjacency matrix is split across the processes using the 2D partitioning approach, which leads to a regular communication structure that follows the rows and columns. To show the functionality of the algorithm, it was gone through the first iteration of a BFS on a small graph.*

5 Implementation

This chapter introduces the implementation of Graph500 that is used for the measurements in the next chapter. The focus will be on the implementation of the parallel BFS algorithm that has been explained before. Optimizations that were undertaken for the GPU and the CPU version will be described.

5.1 Parallel BFS Implementation for GPUs

Section 4.2 explains the parallel BFS algorithm that has been implemented for this thesis. To give a better overview, the computational steps of the algorithm have been presented as function calls in Listing 4.1. Those functions can be divided into two groups: the ones that carry out communication between the processes and the ones that perform computations on the graph data and update the status values of the vertices. The former map to similar MPI function calls in the CPU and the GPU version, while the implementation of the latter differs to target the specific hardware. As described in Chapter 2, different GPU programming platforms exist that focus on a high level of abstraction or provide a low-level programming model, which offers many opportunities for optimization. As the programming effort will be taken into account for the final evaluation, those different approaches have been tested to find the GPU programming platform that has the potential of delivering the best performance at the lowest programming effort.

5.1.1 Selection of a GPU Programming Platform

The `compute_expansion()` function, described in the previous chapter, implements the graph traversal and can be seen as the core part of the BFS algorithm. The adjacency lists of all active vertices are traversed to find yet unvisited neighbors, whose status will be set to active for the next iteration. This procedure is reflected in the OpenACC implementation of the `compute_expansion()` function, provided in Listing 5.1. The data clause in line 3 states that `d_in`, `d_out`, `d_visited`, `d_indices` and `d_adjacencyarray` are pointers to the GPU's global memory. This obviates the need of copying the data before the OpenACC accelerated loop is executed. The loop in line 6 iterates over all vertices assigned to the executing process and checks if they are set in line 8. If this is the case, the starting index and the ending index of the adjacency list are calculated. The loop in line 12 iterates over all adjacent vertices and sets them in the `d_out` array if they are unvisited.

```
1  void compute_expansion(vertex_type *d_indices, vertex_type *d_adjacencyarray, char *
       d_visited, char *d_in, char *d_out, vertex_type matrix_size)
2  {
3      #pragma acc data deviceptr(d_in, d_out, d_visited, d_indices, d_adjacencyarray)
4      {
5          #pragma acc parallel loop
6          for(int i = 0; i < matrix_size; ++i)
7          {
8              if(d_in[i])
9              {
10                 const vertex_type neighbors_start = d_indices[i];
11                 const vertex_type neighbors_end = d_indices[i+1];
12                 for(int j = neighbors_start; j < neighbors_end; ++j)
13                 {
14                     vertex_type neighbor = d_adjacencyarray[j];
15                     if(!d_visited[neighbor])
16                     {
17                         d_out[neighbor] = true;
18                     }
19                 }
20             }
21         }
22     }
23 }
```

Listing 5.1: Implementation of `compute_expansion()` with OpenACC

As described in Section 2.2.2, Thrust provides generic implementations of parallel algorithms like transformations, reductions and for-each loops that can be customized with functors. The functor that implements the adjacency list traversal, `compute_expansion()`, takes the same parameters as the OpenACC `compute_expansion()` function. This is visible in line 7 and 8 of Listing 5.2. The implementation of the functor is not shown as it resembles the lines 8 to 20 of the OpenACC implementation. The functor is used to customize the for-each template. The call of `thrust::for_each()` is depicted in line 4 to 8 of Listing 5.2. A glance at the code reveals that Thrust cannot demonstrate its strength of enabling a high level of abstraction and providing readable code for this problem. The main reason for this is visible from the OpenACC implementation. In line 10 and 11, access to the outer for-loop's running index is required, which also applies to the functor implementation and cannot be avoided. As `thrust::for_each()` abstracts from the loop index as usual for-each loops, this information has to be added manually. In order to achieve this, every entry of `d_in` is mapped to its corresponding index with `thrust::make_tuple()` and `thrust::make_zip_iterator()` is called to enable iterating over these tuples.

```
1  thrust::counting_iterator<vertex_type> matrix_index_start(0);
2  thrust::counting_iterator<vertex_type> matrix_index_end = matrix_index_start +
       matrix_size;
3
4  thrust::for_each(
5      thrust::make_zip_iterator(thrust::make_tuple(matrix_index_start, d_in.begin())),
6      thrust::make_zip_iterator(thrust::make_tuple(matrix_index_end, d_in.end())),
7      compute_expansion(d_indices.data(), d_adjacencyarray.data(),
8              d_visited.data(), d_out.data()));
```

Listing 5.2: Implementation of `compute_expansion()` with Thrust

With regard to programming effort and code complexity, Thrust only seems to be a valid

option if the algorithm to be implemented exactly matches one of the generic algorithms. In that case, the implementation with Thrust is very comfortable. This can be seen from the implementation of the `vertices_in_out()` function in Listing 5.3, which calculates a boolean value that indicates if any vertex in `d_out` is set or not. The `thrust::reduce()` function is used to compute the bit-wise OR-conjunction of all values in `d_out` in just one line.

```
1 frontier = thrust::reduce(d_out.begin(), d_out.end(), 0, thrust::bit_or<char>());
```

Listing 5.3: Implementation of `vertices_in_out()` with Thrust

The third programming platform that has been taken into account for the GPU implementation is CUDA. The CUDA kernel that implements the `compute_expansion()` function is given in Listing 5.4. The main difference in comparison to the OpenACC implementation is that no for-loop is used to iterate over all vertices' entries in `d_in`. Instead, every entry of `d_in` is assigned to a separate thread.

```
1  __global__ void compute_expansion(vertex_type *indices, vertex_type array_length,
       vertex_type *adjacencyarray, unsigned int *in, unsigned int *out, unsigned int *
       visited)
2  {
3      unsigned int global_thread_id = get_global_thread_id();
4      unsigned int lane_id = global_thread_id % 32;
5      unsigned int in_chunk = in[global_thread_id/32];
6      bool vertex_active = is_bit_set(in_chunk, lane_id);
7
8      if(vertex_active)
9      {
10         for(int i = indices[global_thread_id]; i < indices[global_thread_id+1]; ++i)
11         {
12             vertex_type neighbor = adjacencyarray[i];
13             if(!is_neighbor_visited(neighbor, visited))
14             {
15                     set_bit(neighbor, out);
16             }
17         }
18     }
19
20 }
```

Listing 5.4: Implementation of `compute_expansion()` with CUDA

The first step for every thread is to check if its corresponding vertex is active or not. A function, `get_global_thread_id()`, is used to determine the unique id across all launched threads. The ids range from zero to the overall number of vertices assigned to the processes and indicate which vertex a thread is responsible for. The current states of the vertices in `d_in` are stored in 32-bit integers of which each bit denotes the state of a vertex. For this reason, 32 threads with continuous thread-ids request the same integer from `d_in`, which can be seen in line 5. Every thread out of the 32 tests another bit and stores the result in a boolean variable called `vertex_active`. Based on this value, each thread decides whether to perform the exploration of the adjacency list of its vertex. This is implemented in the same way as it is in the OpenACC version.

A comparison of the performance of the initial implementations has shown that CUDA and Thrust deliver better results than OpenACC, with Thrust being ahead of CUDA. For

the sake of a fair comparison it has to be mentioned, that the CUDA version already uses integer instead of char as the datatype of `d_in`. A single integer stores the status values of 32 vertices, while a char only stores one. Therefore, the use of integers reduces the memory requirements significantly, which is important to enable the BFS on large graphs since the global memory of the GPU is very limited. The code examples of the `compute_expansion()` function have shown that is far easier to express the graph traversal in OpenACC and CUDA than in Thrust. Finally, CUDA has been chosen as programming platform to implement the functions as it has more potential for further optimization. The reason for this is that CUDA gives access to shared memory and other special features of the GPUs. The only exception is the `vertices_in_out()` function, which is implemented with a `thrust::reduce()` call as an own CUDA implementation is not expected to give as good results as the optimized `thrust::reduce()` function.

5.1.2 Optimization

The initial CUDA implementation has been profiled with nvprof[1]. According to the profiling results, the execution of the `compute_expansion()` function takes up about 80% of the overall GPU execution time in the initial CUDA implementation. This also includes calls to cudaMemcpy. In contrast, the GPU implementations of `update_levels()`, `update_assigned()` and `update_visited()` to do not consume more than 1-2% of the runtime for the unoptimized GPU implementation. For this reason all optimizations explained in this section target the `compute_expansion()` function. The second largest amount of GPU time (about 14%) is spent on the `compute_predecessors()` function. As the structure of the `compute_predecessors()` kernel is similar to the one of `compute_expansion()`, it will also benefit from all optimizations introduced for the `compute_expansion()` function. As the initial implementation delivers a limited performance, further optimization is necessary. This section introduces the main optimizations that have been tested.

Warp Cooperation The main performance problem of the initial implementation of `compute_expansion()`, provided in Listing 5.4, is unbalanced work across the threads. On the one hand, threads will correspond to non-active vertices and not execute the rather expensive loop in line 10, but they will stall and block the resources of the GPU as long as other threads in the same warp have not finished the loop execution. On the other hand, the number of loop iterations is determined by the length of the adjacency list that is to be processed. Therefore, not all threads with active vertices will perform the same number of iterations. All threads of a warp will occupy the GPU resources until the last thread has finished the loop execution. This situation can be improved with an optimization introduced by Merill et al. [MGG12]. They suggest that all threads of a warp should cooperate to process the adjacency lists of the active vertices assigned to the warp. The adjacency list of an active vertex can be divided into blocks, which are collectively processed by all threads in the warp. One after another, the adjacency lists of all active vertices assigned to the warp are processed. Listing 5.5 shows the implementation of this optimization.

[1] `http://docs.nvidia.com/cuda/profiler-users-guide/`

```
__global__ void compute_expansion(vertex_type *indices, vertex_type array_length,
    vertex_type *adjacencyarray, unsigned int *in, unsigned int *out, unsigned int *
    visited)
{
    unsigned int global_thread_id = get_global_thread_id();
    int block_offset = get_block_id() * NUM_WARPS_PER_BLOCK;
    int thread_id = get_block_local_thread_id();
    unsigned int lane_id = thread_id % 32;
    int warp_id = thread_id / 32;

    volatile __shared__ int communication[NUM_WARPS_PER_BLOCK];
    //assumes the number of threads per block is a multiple of warpsize

    //32 threads in a warp request one word from global memory (32-bit integer)
    unsigned int in_chunk = in[block_offset+warp_id];
    //every thread checks if corresponding IN bit is set
    bool vertex_active = is_bit_set(in_chunk, lane_id);

    vertex_type n = indices[global_thread_id] * vertex_active;
    vertex_type n_end = indices[global_thread_id+1] * vertex_active;


    //whole warps assists in gathering neighbors and check on visited
    while(__any(n_end - n))
    {

        if(n_end - n)
        {
            communication[warp_id] = lane_id;
        }

        int n_gather;
        int n_gather_end;

        int winner = communication[warp_id];
        if(winner == lane_id)
        {
            n_gather = n;
            n_gather_end = n_end;
            n = n_end;
        }

        n_gather = __shfl(n_gather, winner) + lane_id;
        n_gather_end = __shfl(n_gather_end, winner);

        vertex_type neighbor;
        while(n_gather < n_gather_end)
        {
            neighbor = adjacencyarray[n_gather];

            //status lookup and possibly set vertex in out
            if(!is_neighbor_visited(neighbor, visited))
            {
                set_bit(neighbor, out);
            }
            n_gather += warpSize;
        }

    }

}
```

Listing 5.5: `compute_expansion()` optimized with warp cooperation

The first part of the `compute_expansion()` is similar to the first implemented version: all threads of a warp request the same integer from `d_in` and every thread checks if its corresponding bit is set. This value is assigned to the `vertex_active` variable, which can be observed in line 15 of Listing 5.5. Before that, a number of auxiliary variables have been defined with one major difference to the initial implementation. In line 9, an array, named `communication`, is allocated in shared memory. It has as many fields as there are warps in the thread-block and is used by the threads of a warp to communicate. The qualifier `volatile` is necessary to prevent the compiler from optimizing the memory location. Otherwise, it would be possible that the compiler locates the memory in the local registers of the threads. This would lead to the situation that writes to the fields of the array would not be visible to other threads. In lines 17 and 18, every thread calculates the starting and ending index of its vertex' adjacency list or sets them to 0 if the vertex is non-active. The loop in line 22 to 57 will be executed by all threads of the warp together as long as there are unprocessed adjacency lists. Every thread that still needs the adjacency list of its vertex to be processed, writes its `lane_id`, which is the position within the warp, in the warp's field of the `communication` array. This is visible in lines 25 to 28. As the `communication` array is located in shared memory, the writes to the same memory location lead to a bank conflict (explained in Section 2.1.2) and will be executed in sequence. The last write succeeds and determines whose thread's adjacency list is the next to be processed collectively by the threads of a warp. In line 33, every thread reads the entry of the shared memory array that states the "winner". Since all threads are scheduled to execute the same instruction synchronously, it is not possible that any thread reads the corresponding field of `communication` before all threads have completed the write operation in line 27. The winning thread communicates the start and end of its vertex' adjacency list to all other threads (lines 41 and 42). For this propose, the Kepler intrinsic shuffle instruction `__shfl` is used, which provides the functionality of a warp-wide broadcast. Every thread adjusts the starting index with its `lane_id`, which ensures all threads process a different vertex from the adjacency list. The processing is done in the same way as before, visible in line 47 to 53. Afterwards, every thread adds 32 to its index such that the threads will process the next block of 32 vertices from the adjacency list. This process is repeated until the ending index of the adjacency list is reached.

The described approach ensures that the work is split across the threads of a warp and reduces the waste of compute resources. A significant increase of the performance can be observed for this optimization. The effects of the optimizations described in this chapter are presented in detail in Section 6.4 of the next chapter.

Memory Access Pattern The second optimization addresses the access of values from `d_in` and tries to optimize the access pattern in a way such that coalesced memory access is possible. As described in Section 2.1.2, the memory access can be coalesced by the device if the threads of the warp access continuous addresses of the global memory at the same time. In the previous versions of `compute_expansion()` all threads of a warp access the same integer from the global memory. There are 32 times more threads than entries in `d_in`. This is changed for the optimization of the memory access pattern. The number of threads is reduced to match the number of entries in `d_in`. Every thread accesses the integer corresponding to its `global_thread_id`. As the threads are divided into warps according to their thread-id, the threads of a warp will access continuous addresses, which enables

memory coalescing. Afterwards, the thread with the `lane_id` 0 starts and broadcasts the value it obtained from `d_in` to all other threads in the warp. The integer is processed in the cooperative fashion that has been explained before. One after another, every thread of the warp broadcasts an integer value such that all values from `d_in` are processed. In comparison to the previously described optimization, the change of the access pattern makes only a small improvement on the performance.

Dynamic Parallelism The optimization that introduced the warp cooperation to the `compute_expansion()` function aims to reduce unbalanced workloads of threads in the same warp. Unfortunately, unbalanced work of warps is not handled through this optimization as the work is only redistributed between the threads within the same warp. Dynamic Parallelism is used to address this problem. As every thread has to calculate the start and end index of its corresponding vertex' adjacency list, the total number of adjacent vertices can be calculated with negligible overhead. If this number exceeds a threshold, the task of processing the adjacency list is handed over to a child kernel. The thread calls a `compute_expansion_device()` kernel with a launch configuration such that the number of threads in the child grid matches the number of vertices in the adjacency list. Every thread in the child grid processes one vertex of the adjacency list. This means that the status of the vertex is checked and it is added to `d_out` if it is still unvisited. After calling `compute_expansion_device()`, the thread in the parent grid continues to assist in processing the adjacency lists of other threads in its warp as before. The only difference is that it will not try to write its `lane_id` in the shared memory as the adjacency list of its vertex is asynchronously processed by the child kernel instead. This procedure does not lead to fully balanced workloads between the warps, but it reduces the possible spread of imbalance.

Unfortunately, this optimization attempt did not improve the performance due to a lot of overhead when launching a child kernel. If the chosen threshold is too small, it even leads to a great decrease in the performance.

5.2 Parallel BFS Implementation for CPUs

The functionality that has been implemented in GPU kernels for the GPU implementation is realized as OpenMP parallelized for-loops in the CPU implementation. The code implementing the equivalent functionality to the `compute_expansion()` kernel is shown in Listing 5.6. The code in line 4 to 16 is very similar to the lines 8 to 18 in the initial GPU kernel in Listing 5.4. The major difference is that `in`, `out` and `visited` are char arrays in the CPU version. This obviates the need for functions that read and write the correct bit and ensure those actions are atomic if necessary. Line 2 reflects the architectural difference between the CPU and the GPU hardware. The GPU version does not need a loop as it is unrolled and a separate thread is launched to execute one loop iteration. On the other hand, the CPU executes one iteration of the loop after another. With the OpenMP directive in line 1, it is possible to split the loop execution across multiple threads, but those will still run their assigned iterations in sequence. The optimizations that haven been undertaken for the CPU implementation will be described in the following part.

```
#pragma omp parallel for
for (int i = 0; i < matrix_size; ++i)
{
    if (in[i])
    {
        const vertex_type neighbors_start = indices[i];
        const vertex_type neighbors_end = indices[i + 1];
        for (int j = neighbors_start; j < neighbors_end; ++j)
        {
            vertex_type neighbor = adjacencyarray[j];
            if (!visited[neighbor])
            {
                out[neighbor] = true;
            }
        }
    }
}
```

Listing 5.6: Neighborhood exploration on the CPU

Loop Structures The first optimization applied to the CPU implementation, besides the usage of asynchronous MPI communication whenever possible, was a simplification of the structure. Functions of Listing 4.1 that are called one after another have been implemented as separate for-loops in the initial version. Those loops with the same number of iterations and often iterating over the same array, `in` or `out`, have been merged. This reduces the overhead for thread creation and splitting the work across the threads. Furthermore, the C++ `std::fill()` function calls, used for the array initialization, in the beginning, have been replaced with OpenMP parallelized loops.

Scheduling Strategy An equal distribution of the loop iterations of `compute_expansion()` across the threads does not necessarily lead to an equally distributed workload. This is because the amount of work a thread has to perform is related to the number of active vertices in its part of `in`. With a static assignment of the loop iterations to the threads, a waste of computational resources is likely to occur as some threads will finish earlier than others. OpenMP dynamic scheduling seems to be the solution for this problem. The scheduling clause determines the assignment of work to threads. The use of `#pragma omp parallel for schedule(dynamic, 10)` will assign chunks of ten loop iterations to the threads. As soon as they have finished the execution, they obtain the next chunk. This strategy reduces the imbalance between the threads at the cost of more scheduling overhead. The scheduling overhead cannot be ignored as it has a noticeable impact on the performance if the chunk-size is too small. Tests have shown that a good chunk-size for the `compute_expansion()` function and `compute_predecessors()` also is a fraction of about $\frac{1}{1024}$ of the total number of vertices in the graph. As this number changes with the scale factor, the scheduling strategy has been set to `schedule(runtime)` to enable an adaption to the scale factor through an environment variable.

Both optimizations lead to a noticeable improvement of the average performance. As mentioned before, the optimization results will be further examined in Section 6.4.

Frontier Representation The frontier is the set of active vertices during one graph traversal iteration. In the previous versions of the parallel BFS implementation, the frontier is stored as a fixed-size array with one entry per vertex. The CPU implementation uses a char array for this purpose. Towards the beginning and end of a BFS run, the number of vertices in the frontier is small. This leads to many unset values in the `out` arrays and accordingly the `in` arrays of the next iteration. As a consequence, the amount of data that has to be communicated to concatenate the `out` arrays and to broadcast the new `in` values is very high in comparison to the contained information. The better option seems to communicate the vertex-ids of the discovered vertices instead of a status value for every vertex. This approach does also have the advantage that the `compute_expansion()` does not have to iterate over all vertices to check if they are active, but only over the vertices that are actually contained in the frontier. The reduced communication volume comes at the cost of an increased overhead for insertion into the list by multiple threads in parallel. Nevertheless, a positive impact on the overall runtime can be measured for the iterations in the beginning and end of the BFS run. However, the runtime of the BFS iterations that add many vertices to the frontier is significantly increased such that the overall performance decreases. The reason for this is that the global vertex-ids use 64-bit integers, which is required by the Graph500 specification. This leads to a much higher communication volume for these iterations than before. Even the calculation of local vertex-ids using 32-bit does not have a positive effect due to the insertion overhead for many vertices. To overcome these problems, a hybrid approach has been tested. Locally a fixed-size array is used for `out` and a compressed list representation is generated for communication purposes if it is worthwhile. The `in` representation depends on the choice of the frontier representation of `out` in the previous iteration. The communication time can be reduced with this strategy but the generation time of the compressed representation varies depending on the number of collisions at atomic operations. For this reason a, higher maximum performance can be achieved, but the average performance decreases.

5.3 Other Parts of Graph500

Section 3.3 introduces the specification of the Graph500 benchmark, which includes additional parts besides the BFS implementation. In this thesis, the focus was on the implementation of the parallel BFS. Therefore, some components have been taken from the reference implementation or the version of Graph500 that has been implemented for the Student Cluster Competition 2016[2]. This applies, in particular, to the Kronecker graph generator that creates the edge list as well as the validation. The authorship is disclosed in the beginning of the source files. For debugging purposes and to ensure the correctness of the results, tests have been implemented for the CPU and GPU version. These operate on a tree graph, which leads to an unambiguous result.

Summary *CUDA has been chosen as main programming platform for the GPU implementation as it is the only platform that provides the necessary optimization capabilities. The optimizations of the GPU version try to adapt the `compute_expansion()` function to the*

[2] `http://www.hpcadvisorycouncil.com/events/2016/isc16-student-cluster-competition/index.php`

GPU's hardware architecture and execution model. The optimizations of the CPU implementation require only small modifications, for example merging some for-loops and changing the OpenMP scheduling strategy.

6 Evaluation

This chapter analyses the CPU and GPU implementations with regard to a number of different criteria. At the beginning, an overview is given on the system that was used to conduct the measurements.

6.1 Test Environment

All measurements have been performed on the DKRZ's[1] supercomputer Mistral. In addition to the approximately 3000 compute nodes that contain two CPUs of Intel Haswell or Broadwell processors, Mistral includes twelve GPU nodes equipped with Nvidia K80s. The following lists show the configuration details of the CPU and GPU nodes. Up to 64 nodes have been used to measure the performance of the CPU implementation. The maximum number of GPU nodes per job was limited to four. Since each node contains four GPUs, the maximum number of GPUs that has been used for measurements is 16.

CPU Node

- Processors: 2x Intel® Xeon® acCPU E5-2680 v3 @ 2.50GHz (Haswell)
- RAM: 64 GB or 128 GB
- Operating System: Red Hat Enterprise Linux Server release 6.7 (Santiago)

GPU Node

- Processors: 2x Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz (Haswell)
- GPUs: 2x Nvidia K80 with two GK210 each
- RAM: 256 GB
- Operating System: Red Hat Enterprise Linux Server release 6.7 (Santiago)

[1] Deutsches Klimarechenzentrum `https://www.dkrz.de`

Network

- FDR-Infiniband

Software

- Compilers: gcc/g++ 5.1.0, nvcc, PGI 16.5
- CUDA 7.5
- MPI: OpenMPI v1.10.2 with CUDA support

6.2 Performance Analysis

This section aims to analyze the performance characteristics of Graph500 running on CPUs and GPUs. For this purpose, the two versions of the parallel BFS implementation that deliver the best average performance for 64 BFS runs have been chosen for further analysis. These make use of all the optimization steps that were explained before except the last ones, that make use of Dynamic Parallelism and introduce a hybrid frontier representation, as these do not increase or even lower the average performance. The 64 BFS runs are executed on the same graph data but with different starting vertices. The performance is measured in TEPS, which is the performance metric introduced by the Graph500 benchmark. A BFS traverses all edges in the connected component of the source vertex, which usually are about 99.9995% of the edges of a graph that has been created with the Graph500 graph generator. The exact number of traversed edges is obtained from the validation that is employed after every BFS run to ensure the result is correct. To calculate the TEPS performance, this count is divided by the execution time of the BFS run.

At first, the performance of both implementations for different size graph data sets is examined. Two parameters are used to determine the number of vertices and edges in the graph. The scale parameter affects the number of vertices, which is calculated as 2^{scale}. The number of edges depends on the graph's scale and the edgefactor parameter, which influences the average number of out-going edges of a vertex. The overall number of edges is calculated as $2^{scale} \cdot edgefactor$. Figure 6.1 depicts the average GTEPS performance of the GPU and CPU implementation for different scales and a constant edgefactor of 16. Because the TEPS measure is a rate, the harmonic mean is used to calculate the average performance as it is also required by the Graph500 specification for the final output statistics. The number of BFS iterations that is necessary to explore the entire connected component as well as the distribution of the workloads across the processes differ depending on the graph data and the chosen start vertex. Therefore, the average performance of multiple BFS runs with different start vertices is considered in order to enable a comparison of the performance results for graphs with a different scale. Both implementations use 16 MPI processes to run the benchmark. The GPU version utilizes 16 GPUs attached to four nodes, while the CPU implementation of Graph500 is executed on 16 nodes with two CPUs each. All 24

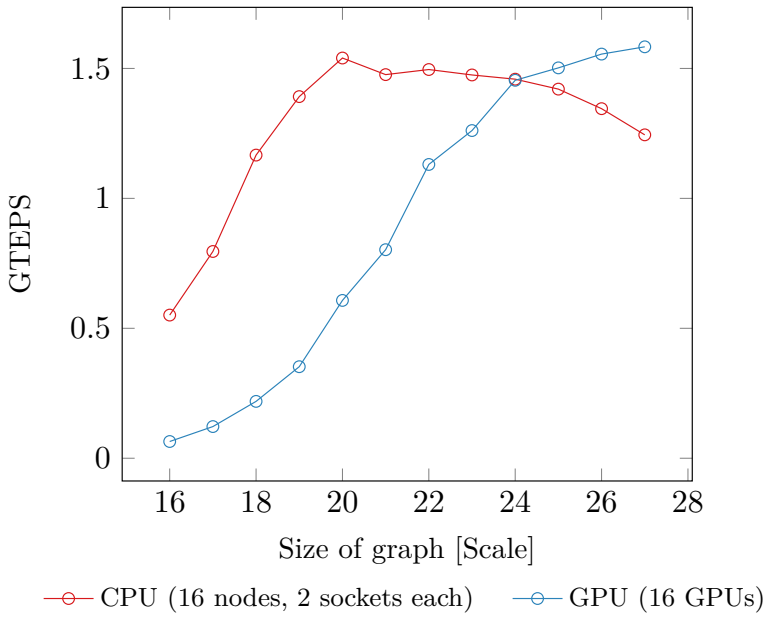

Figure 6.1: Average performance of 64 BFS runs with 16 MPI processes

available CPU cores are utilized, but no Hyper-Threading is applied as this has shown to lower the average performance. The red graph in Figure 6.1 represents the performance of the CPU implementation. A huge increase of the performance can be observed for the smaller scale values from 16 up to 20. In this range, the performance roughly triples. An increase of the scale factor by one leads to a graph with a doubled vertex count. For the smaller scales the CPU implementation benefits from an increased graph size. At scale 20, the maximum performance is achieved with 1.54 GTEPS. Subsequently, the performance remains nearly constant at a slightly lower value for the scale values of 21 to 24. For higher scales, a significant decrease of the performance can be observed with an increased graph size. The blue graph in Figure 6.1 shows the performance of the GPUs. It is visible that this implementation benefits from an increased number of vertices in the graph. Withal, the increase of performance is relatively small for scale 16 to 18, then intensifies for the scale values from 19 to 24 and levels off towards the highest tested scale values. The maximum performance is achieved on the highest scale of 27 with 1.58 GTEPS.

The differences in the courses of the graphs can be explained by the underlying hardware's characteristics. The CPU implementation achieves the best average performance for a graph with 2^{20} vertices. What makes this setting special is the amount of memory that is used. Adding up the amount of memory that is used to store the graph data set and to hold the status values of all vertices of a single process, it can be found that about 40 MB of memory are used. Therewith, it is the largest setting for which all data fits into the L3 caches of the two CPUs of a node. For the graphs that are generated with the Graph500 generator, the number of iterations of a single BFS run increases slowly as the number of vertices in the graph rises. For this reason, the number of communication steps remains relatively constant in spite of the increasing graph size. The communication effort mainly increases as a result of the higher amount of data that has to be transferred over the network. For the smaller graph sizes, this increase does not have a great impact, which leads to a better communication to

calculation ratio. This explains the rise of the performance until scale 20 is reached. From that point on, the effort of the calculations on the graph data and status values of the vertices increases disproportionately. Figure 6.2 depicts this development. The yellow bars represent the time that is spent in OpenMP accelerated regions accumulated for all threads and all 64 BFS runs. In contrast, the black line shows the calculation time that would be expected if the calculation effort increased proportionally with the graph size.

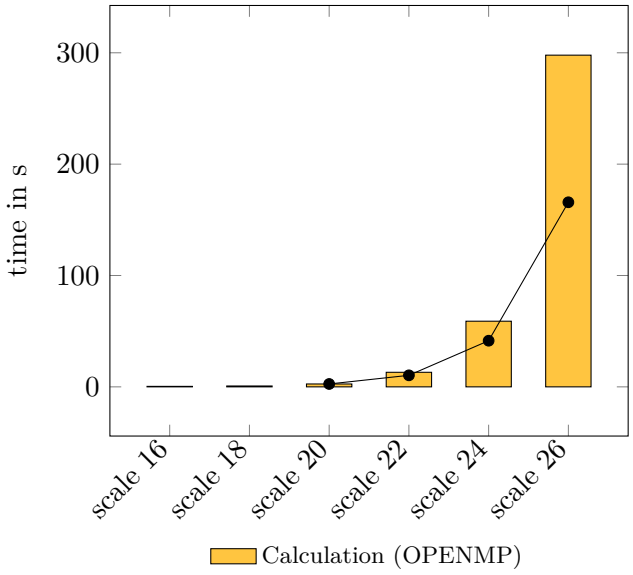

Figure 6.2: Accumulated time spent in OpenMP regions

As described before, the communication effort depends on the amount of data that has to be communicated. An increase of the graph's scale leads to a doubled amount of data that has to be communicated. The effect on the communication effort is noticeable for the lager graph scales, especially because the MPI communication has to be split into multiple function calls for scale values larger than 27 as the 2 GB data limit is exceeded. As a consequence, a further decrease of the performance can be expected for rising scale values. However, an advantage of executing the benchmark of CPUs is that scale values larger than 27 are possible, while this is the limit for the current GPU implementation. The reason for this is the limited amount of global memory of the GPUs. In Figure 6.1 it is visible that the GPUs exceed the performance of the CPUs only for the largest scales that have been tested. With the limitation through the available memory, it can be concluded that the usage of GPUs instead of CPUs is only advantageous for a small number of settings. The course of the graph that depicts the GPU performance can be explained by the degree of utilization of the hardware. The small graphs do not hold enough parallelism to successfully utilize the available hardware. As the number of vertices doubles for every increase of the scale value, a significant increase of the performance can be seen in the middle part. Different from the CPUs, the GPUs have very small caches that do have a noticeable impact on the performance. Instead, the GPUs can make use of the high bandwidth to global memory and can achieve a good performance on large graphs that hold a high enough degree of parallelism.

Doubling the edgefactor has the same effect on the edge list's size as an increase of the

scale by one. In either case, the amount of data doubles, but the parameters affect the graph's internal structure differently. The effects on the performance can be viewed in Figure 6.3. Both, the CPUs and the GPUs benefit from a higher average number of out-going vertices. This change in the graph structure means that the length of the adjacency list increases. Therefore, more contiguous memory addresses of the concatenated adjacency array are requested to process one vertex in the frontier. This leads to a better exploitation of the cache hierarchy on the CPUs and enables coalescing of memory accesses on the GPUs, resulting in the visible performance increase.

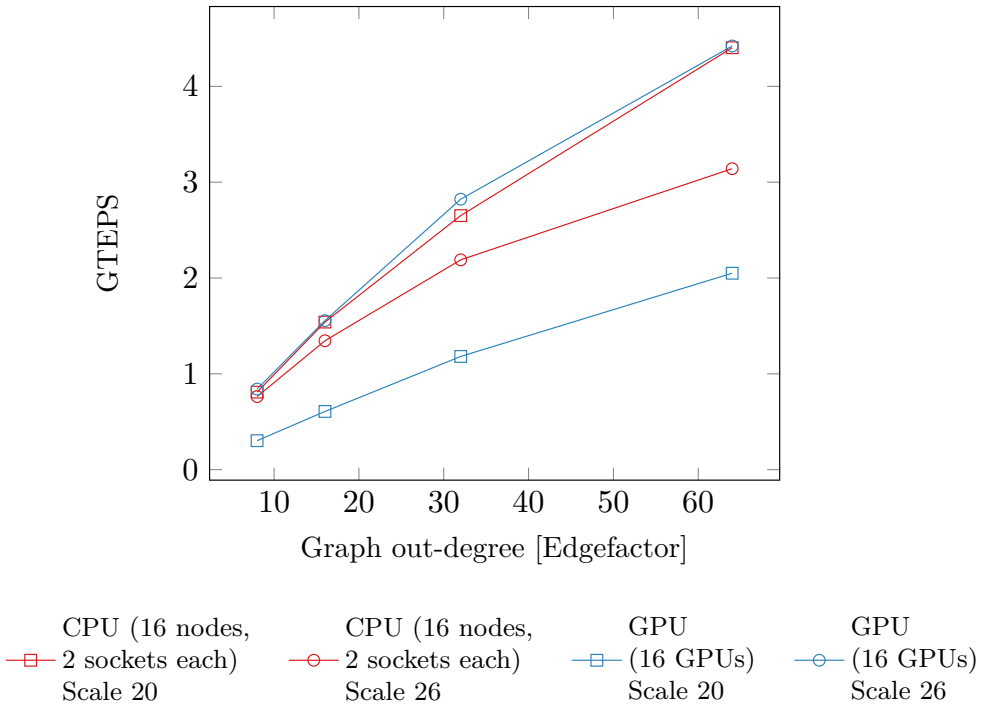

Figure 6.3: Average performance of 64 BFS runs with 16 MPI processes

Figure 6.4 presents the strong scaling behavior of the implementations. For this purpose, the execution time of a BFS is measured using a different amount of compute resources. The problem size, which corresponds to the graph size, stays fixed in this contemplation. TEPS values instead of BFS execution times are depicted in Figure 6.4 to enable a comparison with the previous results. This is possible because the TEPS values are derived through the division by the number of traversed edges, which does not change if the size of the graph stays fixed. To enable a simple partitioning of the adjacency matrix across the MPI processes and a simple calculation of launch configurations, only settings with $2^{2 \cdot x}$ MPI processes can be chosen (with $x > 0$). Runs with 4 and 16 MPI processes were possible for the GPU and the CPU implementation, while a run with 64 MPI processes was only possible for the CPU implementation due to the limited number of GPUs available. For the GPU implementation, the number MPI processes equals the number of utilized GPUs, whereas an MPI process of the CPU implementation is mapped to an entire CPU node.

The GPU results are depicted on the left side and the CPU results can be seen on the right

side of Figure 6.4. A different scaling of the y-axis has to be considered if the results are compared. For both implementations, it is visible that the performance characteristics, which have been found for the execution with 16 MPI processes, can also be recognized if more or fewer resources are used. It is remarkable that the increase of the performance cannot keep up with the increase of the compute resources. The resources quadruple, while the performance increase is in the range of a factor of about 1.5 to 2.5. This can be put down to the common algorithm structure of the parallel BFS as the communication effort rises with an increased grid size. By means of the CPU strong scaling results, it is once again evident that exceeding the cache capacity of the CPUs leads to an inflection of the performance. Distributing the adjacency matrix across more processes leads to a smaller amount of data per process and shifts this point to a larger scale value. For the GPU implementation, Figure 6.4 reveals that the saturation of the hardware starts later if more compute resource are added.

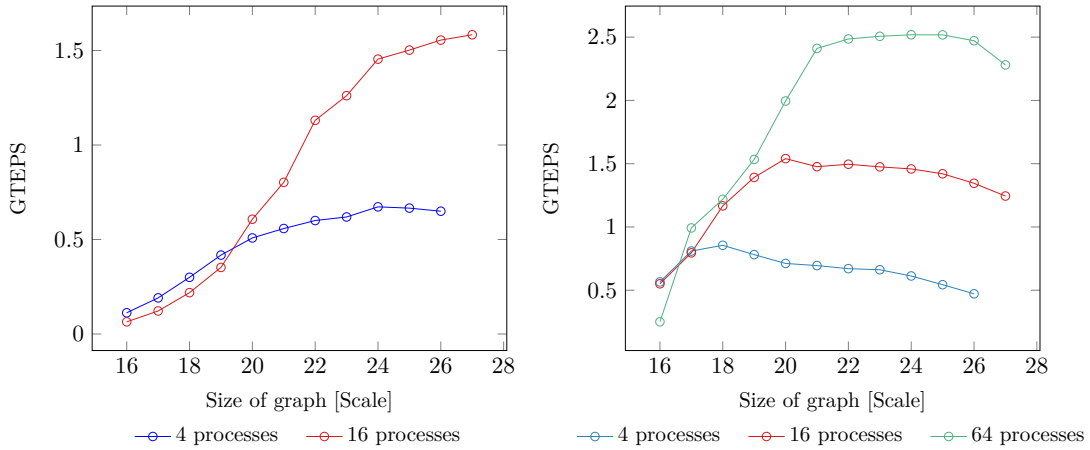

Figure 6.4: Average performance of 64 BFS runs on GPUs (left) and CPUs (right)

As Graph500 is an approved benchmark, other implementations have been published. In Figure 6.5, the performance of two additional implementations is depicted. 2D partitioning has been chosen for both implementations, that are described in [CPW+12] and [US13]. Additionally, both publications implement a parallel BFS algorithm similar to the one described in Chapter 4. The comparison of the performance holds a surprise. While the CPU implementation does not seem to deliver a good performance, the GPU version outperforms all other implementations by far. All tests have been conducted on the Mistral supercomputer. For the CPU reference implementation, 18 MPI processes were executed on 18 compute nodes as 16 MPI processes was not a valid choice for that implementation. A closer look at the GPU reference implementation reveals that a compression algorithm is used to reduce the amount of data that is communicated between the nodes. As this is the only major difference, it can be assumed that the higher performance is a result of this optimization for the most part. In contrast, the reference CPU implementation does not deliver a comparable performance to any of the other, but it has to be remarked that in [CPW+12] good results for a very large number of nodes are represented, starting the tests with 512 compute nodes. The implementation might not be optimized to run on such a small configuration.

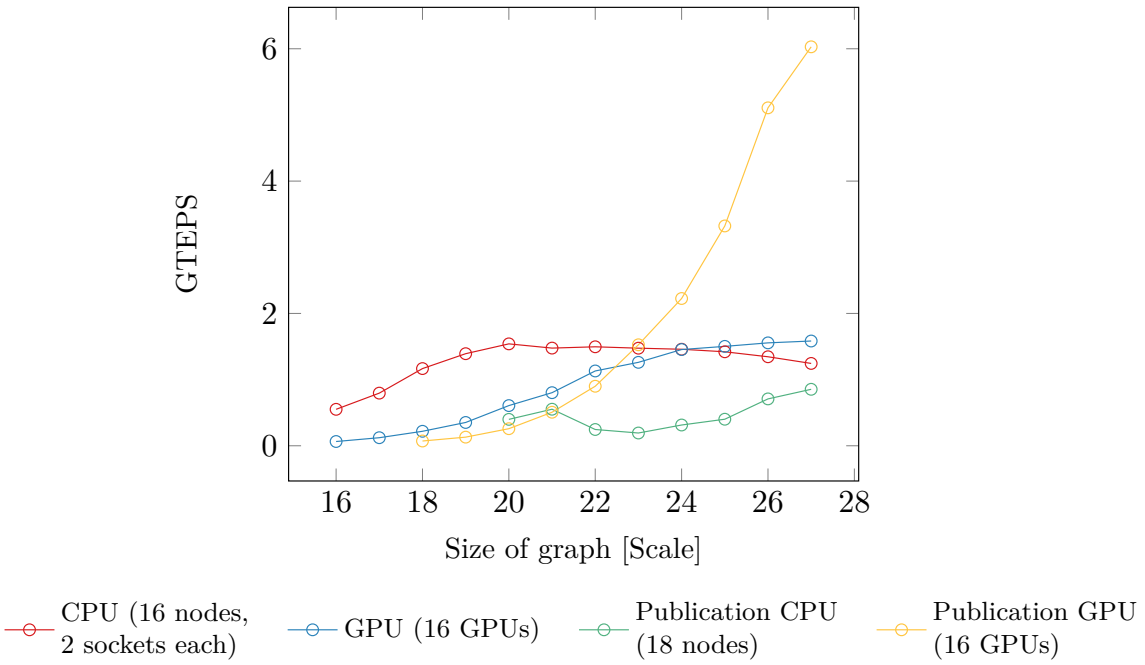

Figure 6.5: Average performance of 64 BFS of the reference implementations

6.3 Cost and Energy Efficiency

To evaluate the potential of using GPUs and CPUs for the execution of graph algorithms, it makes sense to compare the delivered performance with respect to the initial investment costs and the costs that arise from the consumed energy as these two make up a great part of the Total Cost of Ownership (TOC). Figure 6.6 provides a breakdown of the investment costs of the hardware that was used to measure the performance depicted in Figure 6.1. This only includes the core components, therefore, CPU chip costs and GPU costs.

Intel® Xeon® CPU E5-2680 v3 @ 2.50GHz (Haswell):		$1,747.50[2]	≈	1,555.97€
Nvidia Tesla K80:		$4,349.99[3]	≈	3,873.23€
CPU hardware:	16 x 2 x 1,555.97€	=	49,791.04€	
GPU hardware:	4 x 2 x 1,555.97€			
	+ 4 x 2 x 3,873.23€	=	43,433.6€	

Figure 6.6: Hardware costs of the GPU and CPU configuration

The costs that are calculated are for 16 CPU nodes with two CPUs each and for four GPU nodes that have 2 Nvidia K80s attached. As a K80 contains two GPUs, this makes up a total

[2] http://ark.intel.com/products/81908/Intel-Xeon-Processor-E5-2680-v3-30M-Cache-2_50-GHz
[3] http://www.acmemicro.com/Product/14303/NVIDIA-Tesla-K80-GPU-875-MHz-4992-cores-PCI-Express-3-0-x16-24GB-GDDR5-GPU-Accelerator?c_id=566

of 16 GPUs. The costs of the CPUs have been included as the computations on the GPUs are issued from a CPU process and the CPUs are necessary for the MPI communication. For this reason, a GPU cluster will always contain CPUs up to a certain ratio. The CPU hardware costs exceed the GPU hardware costs by about 6,350€. This is a sizable amount, especially as both configurations deliver a comparable maximum TEPS performance. Converted in TEPS/€, the CPU hardware delivers a maximum of 30.9 KTEPS/€, while the GPU hardware reaches 36.4 KTEPS/€. It seems that the statement that GPUs are cost efficient is valid. However, this only holds true if the TEPS/€ result for the maximum performance is considered. If an average of all performance results for graphs with scale 16 to 27 is compared, the CPU hardware delivers the higher value of 25.71 KTEPS/€ in comparison to 20.43 KTEPS/€ for the GPUs. This can be reasoned with the bad GPU performance on small graphs.

Energy efficiency can be measured as performance per watt, therefore, TEPS/W is the metric for graph algorithms. For each job on Mistral the consumed energy is measured in Joule and provided in the job statistics. This includes general job setup, the graph generation and source vertex sampling, the execution of a number of BFS runs and the validation. To calculate the energy efficiency, the average performance and energy consumption of a single BFS run is required. The validation has been turned off for the energy measurement, unfortunately this is not possible for the graph generation. In order to subtract out the graph generation and general overhead, the benchmark has been executed twice with a different number of BFS runs. As an example, a run with 12288 searches and a second one with 16384 BFS runs has been executed on a graph with 2^{20} vertices. In this case, the difference in the energy consumption of the program executions complies the energy consumption of 4096 BFS runs without additional overhead. The Graph500 benchmark requires the execution of only 64 BFS searches. This number has been raised for the energy measurements as considerable variations in the measured energy consumption have been found for short runs of the benchmark. For the graph with scale 20, 12288 and 16384 searches have been executed and 1024 and 1536 for a graph with scale 27. These numbers have been chosen as they restrict the relative standard deviation of the measured energy consumption for the same setting to a max of 2.5%. Based on the obtained energy consumption for a number of searches without overhead, the average consumption of a single BFS run can be calculated and converted in watts with the the average BFS execution time. Finally the average performance of a single BFS run has been divided by this value. The results are depicted in Table 6.1.

	Scale 20	Scale 27
CPU	0.487	0.371
GPU	0.371	0.749

Table 6.1: Energy efficiency in $\frac{MTEPS}{W}$

The energy efficiency has been calculated for the two scale values that deliver the best performance on CPUs and GPUs. The measurements have been conducted using 16 MPI processes and the same configuration as before when measuring the performance for different scale values. A comparison of the results, 0.749 MTEPS/W for the GPU and 0.487 MTEPS/W for the CPU implementation, reveals that a higher efficiency is achieved by the GPUs.

6.4 Productivity Analysis

This section aims to compare the programming effort that has to be made to implement a parallel BFS algorithm on CPUs or GPUs. Figure 6.7 compares the performance of the initial GPU and CPU implementations. A huge gap in the average performance of 64 BFS runs can be noticed as the CPUs deliver a maximum performance that exceeds the GPU results by a factor of about three. With reference to these results, it can be inferred that graph

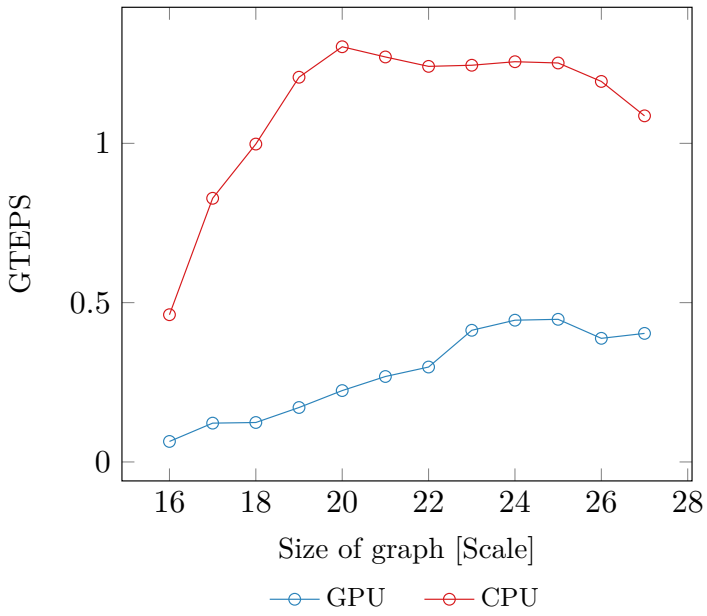

Figure 6.7: Performance results of the inital implementations

algorithms do not naturally match the GPU hardware architecture. Instead, complex optimizations have to be employed to reach a performance that is comparable to the initial CPU implementation. Table 6.2 lists the number of code lines for all non-communication functions of the parallel BFS algorithm. Only the initial implementation and the first optimization step are included. The total sum in the last line reveals that the initial GPU implementation does not only deliver a worse performance but also needs more lines of code. The Source Lines of Code (SLOC) metric does not allow reliable conclusions on the code complexity in general cases. Nevertheless, as the functionality of the compared entities is clearly defined and not too complex, it is acceptable to conclude that in the most cases more steps are required to implement the same functionality on the GPU than on the CPU. A comparison of the implementations shows that this overhead is introduced by the CUDA programming model. The additional lines in the initial GPU implementation arise from the need to determine the position of the thread in the grid, which is used for the data partitioning across the threads. An interesting development becomes visible if the overall lines of code of the initial versions are compared with the count for the first optimization in the next column. While the CPU optimization reduces the number of lines, the GPU optimization increases this number significantly. This does also apply to the other two GPU optimizations that were introduced in Section 5.1.2. In contrast, only one CPU optimization step increases the number of lines. This refers to the optimization that targets the frontier representation and does not improve

	CPU		GPU	
	initial	loop structure	initial	warp cooperation
update_levels_with_in	4	10	8	8
compute_expansion	9		20	34
update_assigned	3	3	4	4
update_visited	4	7	4	4
update_levels_with_out	4		8	8
vertices_in_out	3		4	4
compute_predecessors	11	11	13	39
reset_out	4	4	1	1
sum	42	35	62	100

Table 6.2: Source Lines of Code (SLOC)

the average performance. The scheduling optimization does not change the number of code lines as the scheduling clause is added to already existing OpenMP directives.

The effects of the different optimizations undertaken for the GPU implementation are shown in Figure 6.8 on the left side. Every optimization step that is depicted includes all previous optimizations. The order is as follows: initial, warp cooperation, memory access patter and finally dynamic parallelism. It is visible that only the fist optimization achieves a significant increase of the performance. In comparison, the memory access pattern optimization makes only a little improvement and the GPU implementation that uses Dynamic Parallelism does not increase the performance at all. The right side of Figure 6.8 plots the effects of the CPU optimizations. The optimization that merges OpenMP parallelized loops and the one that applies the OpenMP dynamic scheduling strategy, make only little changes to the code. These optimizations do not increase the complexity of the implementation but lead to noticeable performance improvements. Moreover, the optimizations do not require advanced knowledge beyond common parallel programming skills. This is contrary to the GPU case as all optimizations require specific knowledge of the hardware architecture and the execution model.

It can be concluded that the performance that can be achieved with the initial CPU implementation is quite close to the final result that is depicted in Figure 6.1. Not much effort has to be made to improve the performance with optimizations that include only little changes to the code. On the other hand, the GPU implementation only reaches a comparable performance if very complex and specific optimizations are applied.

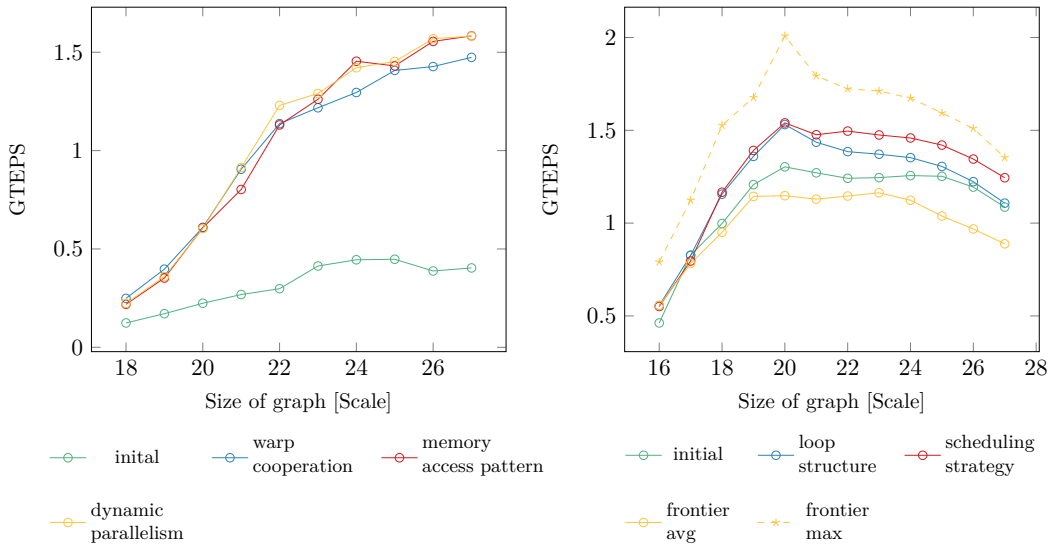

Figure 6.8: Effect of optimizations on the performance on GPUs (left) and GPUs (right)

Summary *A comparison of the performance of the GPU and GPU implementation reveals that a comparable maximum performance is achieved. However, the size of the graph that delivers the best performance is different. It appears that the GPUs are more energy efficient when running BFS and achieves a higher cost efficiency for the maximum performance. For the average performance on different size graphs, the picture is different. The productivity section points out that optimizations are necessarily required for the GPU implementation to reach an acceptable performance. Additionally, it is found that the GPU implementation requires more lines of code than the CPU version.*

7 Conclusion

This thesis analyzes the suitability of GPUs and CPUs for graph algorithms based on a number of different criteria. The first attribute that was considered in the last chapter is the delivered performance. The results of the performance analysis show that the implemented GPU and CPU versions of Graph500 achieve a comparable maximum performance. Since real-world graph applications deal with graph data sets of an enormous size, it can be considered positive that the GPU version reaches the maximum performance when operating on the largest possible graph data set. Apparently, the GPUs can exploit the higher bandwidth to their memories and find "enough" parallelism in large graph problems to utilize the capacity of their parallel hardware architecture. On the other hand, the GPUs' capabilities are limited by the quite small amount of available memory. Especially with regard to the size of real world graph data sets, this restriction weighs heavy. In contrast, the CPUs can be equipped with a great amount of memory and are easily capable of processing graphs with a lot more edges and vertices. However, the performance that can be achieved for large graphs is restricted due to slow access to main memory in the frequently occurring cases of a cache miss. Based on the performance results, it is difficult to become convinced that one hardware configuration is generally more suitable to run parallel graph algorithms than the other.

In Section 6.3, the GPU have been found to be more energy efficient than the CPUs. Regarding the cost efficiency, the picture varies depending on the which performance result is used for the calculation. Based on the maximum performance, the GPUs appear to be more cost efficient. On the other hand, if the average performance on different size graphs is used, the CPUs achieve the better value. The analysis of the programming effort confirms the generally accepted opinion that GPUs are harder to program than CPUs. This finding is deduced from the fact that more lines of code are necessary to implement the same functionality, which points out the more complex programming model. Another reason for the higher programming effort is that complex optimizations are required to achieve an acceptable performance. The optimizations that were undertaken for the GPU implementation can be considered as complex since they depend on detailed knowledge of the hardware architecture and execution model. This becomes also apparent through the fact that none of the examined high-level GPU programming platforms make the required optimizations even possible. The development time, which is a metric that provides definite information on the programming effort, has not been measured during the implementation phase. This is an improvement that should have been made to supplement the picture. However, the subjective impression is that programming for GPUs takes more time than for CPUs.

For the evaluation of the suitability of a hardware configuration, the performance that can be delivered is a key criterion in HPC. As expressed earlier, the result of the performance analysis of the two implemented versions of Graph500 does not allow an unquestionable conclusion on this point. However, in Section 6.2 it has also been observed that a reference

implementation running on GPUs exceeds all other tested implementations' performance. Based on this, one could argue that GPUs seem to be more suitable to run graph algorithms. However, one reason for the higher performance is an optimization that introduces the use of a compression algorithm to reduce the data that has to be communicated. It is reasonable to assume this optimization would also improve a CPU implementation's performance. This reveals the importance of taking the programming effort, which has been made to achieve the result, into account if the suitability of a hardware configuration is to be evaluated based on the performance. Usually, the performance that is found in research publications is the result of a long process of optimization and the development of new approaches. These results can serve as guide values that show what is possible. Nevertheless, estimating the suitability of a hardware configuration based on these values and not including the programming effort might lead into the wrong direction. Especially taking the overall costs into account, the more relevant question seems to be, which hardware configuration will more likely deliver the better performance for an adequate amount of effort put into it. It is reasonable to assume that this question will gain importance as more different products enter the HPC market.

From the analyses of this thesis, it can be concluded that even though the GPUs are more energy efficient and, depending on the point of view, more cost efficient, the programming effort bears no relation to the achieved performance. Since the energy and cost efficiency is calculated based on the performance, a considerably greater programming effort is included in the better efficiency results as well. Overall, it seems that the GPU efficiency results exceed the CPU results not far enough to compensate the higher programming effort that was required to achieve the result.

Bibliography

[Adi14] Andrew Adinetz. CUDA Dynamic Parallelism API and Principles. `https://devblogs.nvidia.com/parallelforall/cuda-dynamic-parallelism-api-principles/`, May 2014. Last accessed 18 September 2016.

[Bar15] Gerassimos Barlas. *Multicore and GPU Programming: An Integrated Approach*. Morgan Kaufmann, Waltham, MA, 2015.

[Ben15] Günther Bengel. *Masterkurs Parallele und Verteilte Systeme Grundlagen und Programmierung von Multicore-Prozessoren, Multiprozessoren, Cluster, Grid und Cloud*. Springer Vieweg, Wiesbaden, 2015.

[BKS15] Piotr Białas, Jakub Kowal, and Adam Strzelecki. GPU-Accelerated and CPU SIMD Optimized Monte Carlo Simulation of $\phi 4$ Model. *COMPUTING AND INFORMATICS*, 33(5), 2015.

[Cha14] Maike Chabowski. It's Raining HPC - How Supercomputing is Shaping Weather Forecasts. `http://insidehpc.com/2014/07/raining-hpc-next-gen-supercomputing-will-shape-weather-forecasts/`, July 2014. Last accessed 20 September 2016.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[CPW+12] Fabio Checconi, Fabrizio Petrini, Jeremiah Willcock, Andrew Lumsdaine, Anamitra Roy Choudhury, and Yogish Sabharwal. Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-memory Machines. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 13:1–13:12, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.

[FDB+14] Z. Fu, H. K. Dasari, B. Bebee, M. Berzins, and B. Thompson. Parallel Breadth First Search on GPU clusters. In *Big Data (Big Data), 2014 IEEE International Conference on*, pages 110–118, Oct 2014.

[Fel15] Michael Feldman. Accelerated Computing: A Tipping Point for HPC (Executive Summary). `http://www.intersect360.com/industry/downloadsummary.php?id=132`, November 2015. Last accessed 18 September 2016.

[FPT14] Zhisong Fu, Michael Personick, and Bryan Thompson. MapGraph: A High Level

API for Fast Development of High Performance Graph Analytics on GPUs. In *Proceedings of Workshop on GRAph Data Management Experiences and Systems*, GRADES'14, pages 2:1–2:6, New York, NY, USA, 2014. ACM.

[FVS11] Jianbin Fang, Ana Lucia Varbanescu, and Henk Sips. A Comprehensive Performance Comparison of CUDA and OpenCL. In *Proceedings of the 2011 International Conference on Parallel Processing*, ICPP '11, pages 216–225, Washington, DC, USA, 2011. IEEE Computer Society.

[GLR+14] Mike Giles, Endre László, István Reguly, Jeremy Appleyard, and Julien Demouth. GPU Implementation of Finite Difference Solvers. In *Proceedings of the 7th Workshop on High Performance Computational Finance*, WHPCF '14, pages 1–8, Piscataway, NJ, USA, 2014. IEEE Press.

[Gra] Graph500 Committee. The Graph 500 List. `http://www.graph500.org`. Last accessed 18 September 2016.

[Gün14] Micheal Günsch. Dual-Kepler mit bis zu 8,7 TFLOPS für Superrechner. `https://www.computerbase.de/2014-11/nvidia-tesla-k80-rechnet-mit-bis-zu-8.7-teraflops/`, November 2014. Last accessed 18 September 2016.

[Har12a] Mark Harris. An OpenACC Example (Part 1). `https://devblogs.nvidia.com/parallelforall/openacc-example-part-1/`, March 2012. Last accessed 18 September 2016.

[Har12b] Mark Harris. An OpenACC Example (Part 2). `https://devblogs.nvidia.com/parallelforall/openacc-example-part-2/`, March 2012. Last accessed 18 September 2016.

[Har12c] Mark Harris. Six ways to SAXPY. `https://devblogs.nvidia.com/parallelforall/six-ways-saxpy/`, July 2012. Last accessed 18 September 2016.

[Har13a] Mark Harris. How to Access Global Memory Efficiently in CUDA C/C++ Kernels. `https://devblogs.nvidia.com/parallelforall/how-access-global-memory-efficiently-cuda-c-kernels/`, January 2013. Last accessed 18 September 2016.

[Har13b] Mark Harris. Using Shared Memory in CUDA C/C++. `https://devblogs.nvidia.com/parallelforall/using-shared-memory-cuda-cc/`, January 2013. Last accessed 18 September 2016.

[HN07] Pawan Harish and P. J. Narayanan. Accelerating Large Graph Algorithms on the GPU Using CUDA. In *Proceedings of the 14th International Conference on High Performance Computing*, HiPC'07, pages 197–208, Berlin, Heidelberg, 2007. Springer-Verlag.

[HW11] Georg Hager and Gerhard Wellein. *Introduction to High Performance Computing for Scientists and Engineers*. CRC Press, Boca Raton, FL, 2011.

[Hwu12] W. Hwu. *GPU Computing Gems Jade Edition (Applications of GPU Computing Series)*. Morgan Kaufmann, Waltham, MA, 2012.

[JP12] Jiri Jaros and Petr Pospichal. A Fair Comparison of Modern CPUs and GPUs Running the Genetic Algorithm Under the Knapsack Benchmark. In *Proceedings of the 2012T European Conference on Applications of Evolutionary Computation*, EvoApplications'12, pages 426–435, Berlin, Heidelberg, 2012. Springer-Verlag.

[KDK+11] Stephen W. Keckler, William J. Dally, Brucek Khailany, Michael Garland, and David Glasco. GPUs and the Future of Parallel Computing. *IEEE Micro*, 31(5):7–17, 2011.

[KKI14] Tomasz Kajdanowicz, Przemyslaw Kazienko, and Wojciech Indyk. Parallel Processing of Large Graphs. *Future Gener. Comput. Syst.*, 32:324–337, March 2014.

[Kus14] Daniel Kusswurm. *Modern X86 Assembly Language Programming: 32-bit, 64-bit, SSE, and AVX*. Apress, New York, 2014.

[LGHB07] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. Challenges in Parallel Graph Processing. *Parallel Processing Letters*, 17(01):5–20, 2007.

[LKC+10] Victor W. Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D. Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, Ronak Singhal, and Pradeep Dubey. Debunking the 100X GPU vs. CPU Myth: An Evaluation of Throughput Computing on CPU and GPU. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, pages 451–460, New York, NY, USA, 2010. ACM.

[LWH10] Lijuan Luo, Martin Wong, and Wen-mei Hwu. An Effective GPU Implementation of Breadth-first Search. In *Proceedings of the 47th Design Automation Conference*, DAC '10, pages 52–55, New York, NY, USA, 2010. ACM.

[MGG12] Duane Merrill, Michael Garland, and Andrew Grimshaw. Scalable GPU Graph Traversal. *SIGPLAN Not.*, 47(8):117–128, February 2012.

[MWBA10] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the Graph 500. *Cray User's Group (CUG)*, 2010.

[NVI12] NVIDIA Corporation. NVIDIA's Next Generation CUDA Compute Architecture: Kepler GK220. `http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf`, 2012. Last accessed 18 September 2016.

[NVI15] NVIDIA Corporation. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`, September 2015. Last accessed 18 September 2016.

[PWW+15] Yuechao Pan, Yangzihao Wang, Yuduo Wu, Carl Yang, and John D. Owens. Multi-GPU Graph Analytics. *CoRR*, April 2015.

[Rau13] Thomas Rauber. *Parallel Programming for Multicore and Cluster Systems*. Springer-Verlag, Heidelberg, 2013.

[SCM14] Horacio Emilio Pérez Sánchez, José M. Cecilia, and Ivan Merelli. The role of High Performance Computing in Bioinformatics. In Ignacio Rojas and Francisco M. Ortuño Guzman, editors, *International Work-Conference on Bioinformatics and Biomedical Engineering, IWBBIO 2014, Granada, Spain, April 7-9, 2014.*, pages 494–506. Copicentro Editorial, 2014.

[US13] K. Ueno and T. Suzumura. Parallel distributed breadth first search on GPU. In *20th Annual International Conference on High Performance Computing*, pages 314–323, Dec 2013.

[ZLL14] Zhe Zhu, Jianjun Li, and Guohui Li. *Load-Balanced Breadth-First Search on GPUs*, pages 435–447. Springer International Publishing, Cham, 2014.

Appendices

A. Source Code

The source code is handed in on the CD in the printed versions. The `master` branch of the repository includes the final versions that were used for the performance and energy efficiency measurements. The code of the different optimization steps can be found on different branches. This is further explained in the included `README.md`, which also includes a setup guide.

List of Acronyms

API Application Programming Interface

AVX Advanced Vector Extensions

BFS Breadth-first Search

CCL Connected Component Labeling

CPU Central Processing Unit

CUDA Compute Unified Device Architecture

DP Dynamic Parallelism

DRAM Dynamic Random-access Memory

FLOPS Floating-point Operations Per Second

GPU Graphics Processing Unit

HPC High Performance Computing

HPL High Performance Linpack

ILP Instruction-level Parallelism

MPI Message Passing Interface

PCI Peripheral Component Interconnect

PCIe Peripheral Component Interconnect Express

RAM Random-access Memory

SIMD Single Instruction Multiple Data

SIMT Single Instruction Multiple Threads

SLOC Source Lines of Code

SM Streaming Multiprocessor

SMX Streaming Multiprocessor (Kepler)

SSE Streaming SIMD Extensions

SSSP Single-source Shortest Path

STL Standard Template Library

TEPS Traversed Edges Per Second

TOC Total Cost of Ownership

List of Figures

Listings

List of Tables

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ich bin damit einverstanden, dass meine Abschlussarbeit in den Bestand der Fachbereichsbibliothek eingestellt wird.

Ort, Datum	Unterschrift