Seminararbeit - Bachelor of Computer Science

Berkeley's Dwarfs on CUDA

Paul Springer
RWTH Aachen University
Contact: Paul.Springer@RWTH-Aachen.de

Supervised by

Prof. Dr. Martin Bücker	(Institute for Scientific Computing)
Sandra Wienke, M.Sc.	(HPC-Group - Center for Computing and Communication)

Contents

1 Introduction

Graphics processing units (GPUs) greatly improved their performance over the last ten years. The first graphics cards have been developed in the late 90's and were targeted for the mass market. These first cards were special purpose hardware, designed to accelerate graphic processing required in computer games. As the interest in computer games continued, GPU developers such as Nvidia and AMD/ATI continuously improved the performance and level of parallelism of their GPUs. To this end, it became desirable to exploit this special purpose hardware for general purpose computing. Therefore, parallel programming models such as CUDA and OpenCL are developed in order to utilize the hundreds of cores of modern GPUs which yield a peak performance of up to 1331 GFLOP/s[1] on a single GPU. These programming models greatly decreased the effort

[1]single precision floating point performance on Nvidia's Tesla M2090

necessary to program these architectures. At their beginning in 2007 (CUDA) and 2008 (OpenCL), they suffered from drawbacks such as low double precision performance, few debugging capabilities and the lack of atomic operations [37]. Today, the development of these models and the underlying architectures are by no means completed.
Nevertheless, the fact that more and more heterogeneous clusters within the TOP500 [3] consist of a mixture of CPUs and GPUs emphasizes the increasing importance of GPUs in the field of scientific computing. Hence, it is crucial to provide new benchmark suites which enable the evaluation of heterogeneous systems in order to evaluate and compare the performance of these systems.
Asanovic et al. [4] introduces 13 computational kernels, so called Dwarfs, which are believed to play an important role in future scientific computations. In the course of this report, I will focus on existing GPU implementations of some selected Dwarfs and introduce three benchmark suites which implement a subset of the 13 Dwarfs on the GPU. The performance across these benchmark suites can guide computer system researchers and indicate a first clue if an architecture is suited for future applications.
After all, one important question remains: Is it worthwhile to port an application to the GPU considering the effort needed to learn the new programming paradigm and the unique problems related to high performance GPU implementations. I will address this question throughout this report.

This report is organized as follows: Section 2 briefly describes the *CUDA programming model* and the underlying *GPU architecture* which is essential in order to develop high performance applications. Section 3 introduces the 13 Dwarfs and lists some references to existing GPU implementations. Section 4 discusses the performance and problems related to some selected GPU Dwarfs. Section 5 describes the effort needed to become familiar with the *Compute Unified Device Architecture* (CUDA) and carries out a case study where I implement a simple application using CUDA. Section 6 draws a conclusion.

2 Compute Unified Device Architecture

CUDA was released on 15 February 2007. It is restricted to NVIDA GPUs and is compatible with GPUs such as G80 and onwards. CUDA is being improved continuously, new features such as atomic operations, ECC support, double precision support, debugging capabilities and further tools and libraries emerge every year.

CUDA is designed such that a C/C++ programmer who is familiar with parallel programming should be able to port an application to the GPU with little effort [17]. As we will see in section 5.2.2, depending on the application, it can be easy to port an existing OpenMP application to CUDA. This initial GPU version however, does not necessarily achieve high performance (see section 5.2.3). Furthermore, various different optimization techniques make it difficult to find the optimal solution which achieves peak performance [43]. In order to optimize an application, fundamental knowledge about the underlying GPU architecture and optimization techniques is required.

The upcoming subsections briefly describe Nvidia's current GPU architecture called Fermi [41] and the very basic characteristics of the programming model, respectively.

See [37], [36], [12], [28], and [43] for further information.

2.1 Fermi Architecture

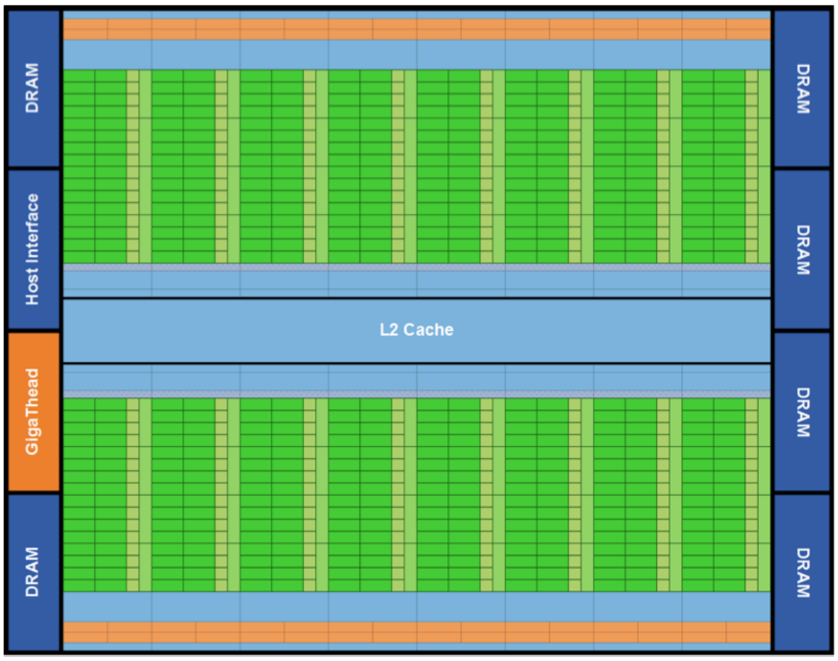

Figure 2.1: Fermi's 16 SM are positioned around a common L2 cache. Each SM is depicted as a vertical rectangular strip that contains an orange portion (scheduler and dispatcher), a green portion (execution units), a light green portion (SFUs) and light blue portions (register file and L1 cache). Taken from [34].

Modern Nvidia GPUs consist of up to 512 cores which make these architectures well suited for data-parallel applications. The cores are organized in multiple streaming multiprocessors (SMs) and perform floating point operations. A Fermi graphics card such as the TESLA M2090 consists of 16 SMs each having 32 cores and four special function units (SFUs)[2], an configurable shared memory of 48 KB/16 KB, a configurable L1 cache of 16 KB/48 KB , 32K registers á 4 Bytes and further units for scheduling and memory accesses (see Figure 2.1). Moreover, a Fermi GPU is capable of running 1536 threads per SM simultaneously which results in very little storage per thread. Fortunately, modern GPUs offer various solutions which can compensate this disadvantage such as prefetching, task level parallelism, data level parallelism, a rich variety of memories, etc. Prior to any computation the required data has to be transfered from the host to global memory which resides on the graphics card. These transfers limit the use of GPUs for applications which require very much data to carry out very few computations.

[2]Used to accelerate functions such as sin(), cos(), exp(), log(), etc. at the cost of numerical precision.

Global memory is accessible by all SMs and it comprises up to 6 GB of GDDR5 DRAM with a memory bandwidth of up to 177 GB/s. Even though the bandwidth is quite high compared to 24 GB/s of DDR3-1600, it is shared by all threads and becomes a limiting factor for many high performance applications. In order to hide the high latency and the limited memory-bandwidth of global memory, it is necessary to have a *high compute to global memory access ratio* (CGMAR) [28], [43].
Fermi is the first architecture which tackles this problem by introducing a L2 cache of size 768 KB which caches all accesses to global memory. The full cache hierarchy of Fermi is a further step towards general purpose computing. It can improve the performance of applications such as sparse linear algebra, sorting, ray tracing, fluid physics applications, etc. (see Figure 4.2).

2.2 Programming Model

CUDA is a *Single Instruction Multiple Thread* (SIMT) architecture, it wraps the computation on the GPU in so called kernels. Each kernel is capable of running thousands of threads simultaneously. The threads are locally organized in blocks, which are further organized in warps. A warp is a logical unit of 32 consecutive threads, these threads execute their operations in lock-step.

Furthermore, global memory accesses are issued by a warp as well. The threads memory accesses of threads belonging to the same warp can be coalesced under certain circumstances. For instance, if all the threads within the same warp access consecutive 4 byte elements in global memory, CUDA is capable to coalesce these 32 accesses to one global memory access[3].

Moreover, all threads within a block run on the same SM which enables these threads to share data by using the on chip shared memory and further collaborate by synchronizing their execution by very light-weighted barriers.

3 Dwarfs

This section briefly describes the Dwarfs as introduced by Asanovic et al. [4]. I will focus on the GPU implementation of some selected Dwarfs in section 4.

According to [4] a Dwarf is an algorithmic method that captures a pattern of computations and communication. Many researchers from various domains such as circuit design, computer architecture, massively parallel computing, computer-aided design, embedded hardware and software, programming languages, compilers, scientific programming and numerical analysis meet over a period of two years to design a collection of 13 Dwarfs which are believed to be the kernels for many future applications. They are designed to guide the development of parallel architectures and novel programming models. A new architecture should have good performance across all Dwarfs in oder to be suited for a broad range of future applications, this is especially important since many applications

[3]Different GPU architectures have different coalescing requirements [37].

can comprise several Dwarfs (e.g. computer graphics involves *Sparse Linear Algebra, Graph Traversal, etc.*[4]).

Initially there have been only seven Dwarfs but further research has shown that at least six more Dwarfs are necessary to cover a broader range of applications (see Table 3.1). Some of these Dwarfs such as *Dense Linear Algebra* or *Monte Carlo* seem to be a perfect fit for multicore and manycore architectures whereas Dwarf 13 (*Finite State Machines*) typically involves very little parallelism. Therefore it might be a good idea to change the underlying algorithmic approach in order to achieve good performance across all 13 Dwarfs.

Table 3.1 lists all the Dwarfs proposed by Asanovic et al. and provides references for their GPU implementation, if available. Furthermore, I will discuss the GPU references of Dwarf 1, 2, 3, 4, 5, 7 and 9 in section 4 in more detail.

	Dwarf	GPU References
Original	1) Dense Linear Algebra	[31][19][16][35][44][47]
	2) Sparse Linear Algebra	[31][19][40][10][7][9][14]
	3) Spectral Methods	[21][25, p.629f][38]
	4) N-Body Methods	[33, p.677f][25, p.75f][28, p.173f][46][8]
	5) Structured Grids	[31][16][17][12][32]
	6) Unstructured Grids	[16][17]
	7) Monte Carlo	[42][25, p247f, p.813f][39][23]
Additional	8) Combinational Logic	[33, p.785f][17]
	9) Graph Traversal	[31][29][16][25, p.75f, p.439f][22]
	10) Dynamic Programming	[16][17][13][30]
	11) Back-track and Branch & Bound	
	12) Graphical Models	
	13) Finite State Machine	[15]

Table 3.1: 13 Dwarfs introduced by Asanovic et al. [4].

4 GPU Dwarfs

This section introduces three available GPU benchmark suites which implement many of the aforementioned Dwarfes, lists typical problems related to efficient GPU implementations and depicts the problems and performance of some selected Dwarfs.

4.1 Benchmark Suites

Benchmark suites provide a good mean to compare different architectures with each other and exhibit their strengths and weaknesses.
Current benchmark suites such as *SPEC CPU 2006* [2], *SPLASH-2* [50] or *Parsec* [11]

typically only support CPU architectures but neglect accelerators such as GPUs or the implemented kernels do not span across a wide range of applications (e.g. BioParallel [26] for biomedical applications, MediaBench [1] for multimedia applications).
In order to overcome these limitations heterogeneous benchmark suites such as Rodinia benchmark suite [16], Parboil benchmark suite [31] and Scalable Heterogeneous Computing Benchmark Suite (SHOC) [19] have being developed.

Rodinia is an open source benchmark suite which implements a wide range of applications with different computation and communication patterns. The implemented applications can be mapped to a subset of the 13 Dwarfs. Rodinia uses OpenMP and CUDA in order to compare multicore CPUs vs. manycore GPUs. Rodinia is still under development, therefore, new applications are being added which cover even more Dwarfs. As the development of Rodinia proceeds even OpenCL versions of certain applications are added. A drawback of this benchmark suite is that it is barely documented.

Parboil is an open source benchmark suite which implements a subset of the 13 Dwarfs on the GPU. The suite itself is not well documented. Nevertheless the description of some kernels refers to published papers which describe the GPU implementation in more detail. Moreover, most of these applications are only implemented in CUDA, however some are having a basic CPU implementation as well. Thus, this benchmark suite does not seem to be a good choice if one wants to compare the performance of a CPU-based system vs. a GPU-based system.

In comparison to Rodinia and Parboil the **SHOC** benchmark suite is not open source, but it is however freely available. SHOC is divided in three parts:

- **Level 0** measures low level characteristics of the GPUs such as memory bandwidth, peak FLOPS, etc.
- **Level 1** implements eight computational kernels such as FFT, MD, Reduction, Scan, SGEMM, SpMV, Sort, Stencil2D and Triad which can be mapped to a subset of the 13 Dwarfs.
- **Stability Tests** stress the system with continuous computations and data transfers to exhibit problems related to bad cooling, overclocked GPUs, etc.

In contrast to Rodinia and Parboil, this benchmark suite targets CUDA and OpenCL alike. Therefore, SHOC is well suited to compare the performance between CUDA and OpenCL but it is not suited for comparing CPU vs GPU since it lacks a CPU implementation. A major advantage of SHOC is that it uses MPI to distribute the computation among several nodes each having one or more graphics cards. Hence SHOC is able to scale from a single node to a large cluster. Moreover, it is very easy to setup and well documented. Figure 4.1 depicts the performance differences between CUDA and OpenCL with respect to some GPU Dwarfs running on NVIDA's C2050[4].

[4]With a peak performance of 1030 GFLOP/s and 515 GFLOP/s for single and double precision, respectively.

		Max Peak	SGEMM	SpMV	FFT
Single	CUDA	998	617	10	350
	OpenCL	1005	427	5	62
Double	CUDA	501	297	9	176
	OpenCL	505	171	4	27

Table 4.1: SHOC: Performance comparison between CUDA and OpenCL. All values are expressed in GFLOP/s.

4.2 Typical Problems

For brevity, this subsection only describes the most common problems involved with GPU implementations. I will focus on specific problems with respect to the GPU Dwarfs in the following sections.

A problem common to some GPU Dwarfs is the time needed to offload the data from the CPU to the GPU. Depending on the algorithm and its memory access pattern it can be a good idea to chop up the communication into smaller pieces which can be processed in parallel (this is only possible if there are no further data dependencies between these blocks). Thus, some algorithms require fundamental changes to ease the problem of CPU to GPU communication.

Even though the global memory bandwidth of the Tesla C2050 is quite high (i.e. 144 GB/s) it is a limiting factor for many data-intensive applications. Therefore it is desirable to *coalesce* data transfers and reuse data as much as possible. Coalescing the memory accesses sometimes require non trivial reorganization of data structures (e.g. it is not preferable to access a row of a matrix in a row wise fashion per thread, since this would result in many uncoalesced data transfers). Another technique to improve the memory bandwidth is the efficient use of different memories (i.e. registers, shared memory, texture memory and constant memory) which then again requires further non trivial rearrangement of the algorithm.

There are even further limitations since registers and shared memory are scarce resources and extensive use of these resources could limit the number of threads/ blocks which can be executed on a SM simultaneously. This, on the other hand, would result in less thread level parallelism which could expose the high memory latencies of the global memory. A solution to this problem could be to subdivide a large kernel into few smaller ones.
But this is not a remedy to this problem as it poses two new problems. First, each kernel call involves a non negligible overhead. Second, shared memory is not persistent during kernel calls.

Moreover, some GPU Dwarfs (e.g. N-Body Methods and Graph Traversal) suffer from the lack of an inter-block synchronization mechanism [51]. A widely used solution to this problem is to stop the current kernel and relaunch it after a global synchronization by the host. Even though the Fermi architecture reduces the overhead of kernel launches by up to **10x** with respect to older GPUs [41], data within shared memory is still not persistent between kernel calls. This induces potentially unnecessary memory transfers.

Another problem common to some GPU Dwarfs is the problem of *thread divergence*, due to the SIMT nature of CUDA. Threads are said to diverge if they belong to the same warp and follow different execution paths. Thread divergence results in serialization of the diffrent execution paths. Hence, it can serverely impact the performane of an application.

4.3 Dense Linear Algebra

The first Dwarf the I will address in this report is dense linear algebra.

Due to its inherent parallelism DLA is one of the best analyzed GPU Dwarfs, it offers a variety of GPU implementations. In the course of this subsection, I will list some of these implementations and conclude with a CPU vs GPU comparison.

All benchmark suites mentioned above implement some DLA Dwarf(s). SHOC and Parboil both implement an ordinary GEMM, whereas Parboil's implementation uses the algorithm as described by V. Volkov et al. [47]. Rodinia on the other hand includes a LU decomposition and clustering algorithms such as *K-means* and *Stream Cluster*.

More important than the DLA Dwarfs of the benchmark suites are available libraries such as CUBLAS [35], MAGMA [44] and CULA [24]. CUBLAS is Nvidia's implementation of BLAS (Basic Linear Algebra Subprograms) based on CUDA which is currently limited to one GPU only.
MAGMA and CULA are heterogeneous libraries which implement a subset of LAPACK (Linear Algebra PACKage). Both rely on CUBLAS and CUDA to accelerate their functions. The main difference however is that CULA is a commercial library which covers a larger subset of LAPACK than MAGMA. MAGMA on the other hand is freely available and it also offers great speedups over pure CPU LAPACK implementations such as Intel's MKL (see Figure 4.1).
Both libraries offload computationally intensive parts which stress the functional units and at the same time require few memory accesses to the GPU, while computing small operations which do not offer sufficient parallelism (e.g. panel factorizations) on the CPU.

All of the libraries are constantly evolving, thus it is likely to see a CUBLAS with multi GPU support as well as more LAPACK functions implemented in both CULA and MAGMA.

4.4 Sparse Linear Algebra

Even though sparse linear algebra (SLA) is yet another important domain for scientific computing, there are not as many SLA libraries and algorithms available which use the GPU to accelerate their computation as we have seen for DLA. One reason for this is that SPA typically allows few arithmetic operations per memory access which is not favorable for a modern GPU. The two most notable SLA libraries for CUDA are CUSP [10] and CUSPARSE [40].

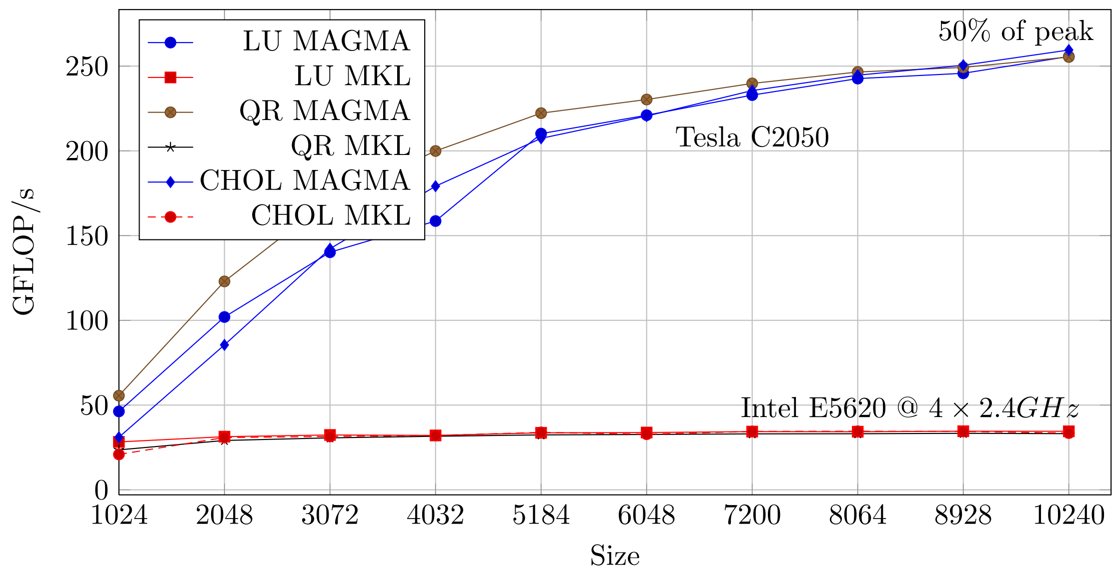

Figure 4.1: Performance of LU, Cholesky and QR decomposition of a square matrix using double precision.

CUSPARSE is Nvidia's implementation of the sparse BLAS routines (i.e. vector-vector, matrix-vector and matrix-matrix). Furthermore, it offers routines to convert different matrix formats to one another (e.g. dense to csr, dense to csc, crs to csc, etc.). CUSPARSE is designed for a single GPU and does not autoparallelize across multiple GPUs.

CUSP in comparison to CUSPARSE is an open source library which only implements a subset of the sparse BLAS routines but it offers iterative solvers based on Krylov subspace methods. Moreover, it offers I/O functions which can load an mtx-file directly into a sparse matrix container such as coo, csr, dia, ell or hyb.

The sparse matrix vector product SpMV (i.e. $A \times x = y$, where A is sparse matrix and x is dense vector) is one of the most important kernels within the SLA domain. It is the most compute intensive kernel in Krylov subspace methods which are used to solve linear equations of large systems which evolve from applications such as finite element methods, structural engineering, linear programming, etc. Henceforth I will discuss the problems involved with an efficient SpMV implementation as described by Baskaran et al. [7].

The greatest problem involved with an efficient SpMV implementation is that it is a inherently memory bound kernel. This has different reasons. (1) It has irregular and indirect memory accesses. (2) It does not exhibit much data reuse (i.e. only data reuse of x is possible). (3) It has a bad memory access to compute ratio. With these problems in mind Baskaran et al. [7] developed a SpMV algorithm that utilizes the different memories of a modern graphics card to hide memory latency. The efficient implementation involves padding of matrix rows to avoid non-aligned memory accesses, usage of the cached texture memory for vector x, avoidance of synchronizations, etc. Further applied optimizations are described in [7]. The benefit of using caches is depicted in Figure 4.2. Moreover, the performance of SHOC's SpMV implementation is shown in Table 4.1.

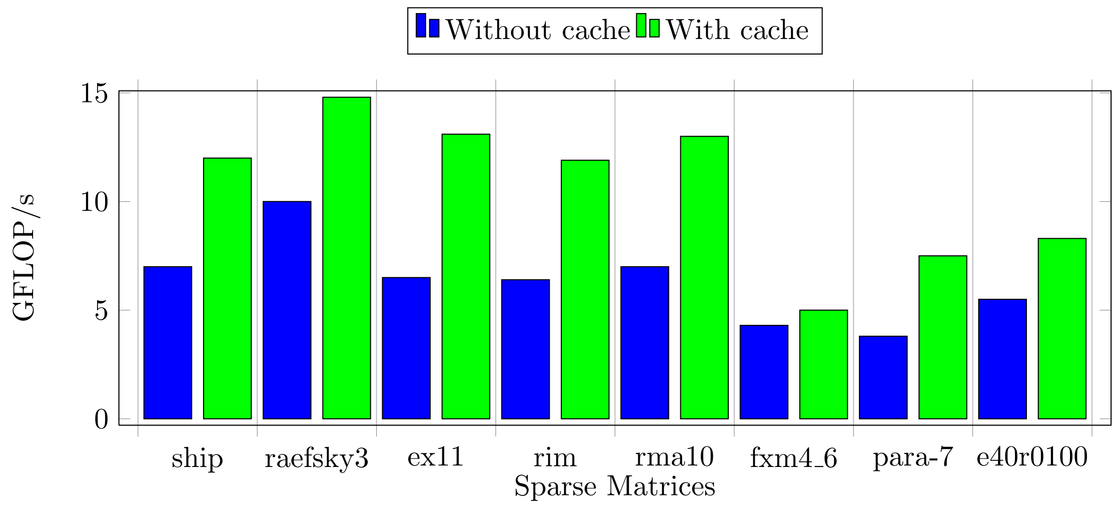

Figure 4.2: SpMV Performance shown by Baskaran et. al [7] running on a GeForce GTX 280.

4.5 Spectral Methods

Spectral methods are used in many different fields such as applied mathematics and scientific computing. They often involve the use of a fast fourier transform (FFT). The most notable library for accelerated FFTs on the GPU is Nvidia's CUFFT [38]. Furthermore, Parboil implements a FFT according to the description of Govindaraju et al. [21]. SHOC implements the FFT as described by V. Volkov et al. [48] (see Table 4.1).

According to Govindaraju et al. FFTs on the GPU face typical problems such as avoidance of bank conflicts, non coalesced memory accesses, efficient use of various memories, etc. Solutions to these problems involve padding of shared memory, reorganization of the algorithm, and so forth.
Moreover, Govindaraju et al. [21] empirically determined that it is beneficial to reduce the shared memory usage per block at the cost of more synchronizations in order to increase the amount of blocks which can run simultaneously on a GPU (i.e. increased thread level parallelism to hide memory latency). They also computed small FFTs entirely in shared memory and used the registers of the GPU extensively. With all these applied optimizations, they are able to achieve speedups of up to **61x** for large FFTs

running on a GTX 280 w.r.t. Intel's MKL version 10.2 running on an Intel QX9650. Nevertheless, MKL produced slightly lower errors than the GPU implementation. Detailed plots can be found in [21].

It is worth mentioning that all the implementations mentioned in this subsection are at least two years old and do not autoparallelize across multiple GPUs. A newer FFT implementation which utilizes multiple GPUs is described in [25, p.629f]. This version targets ,,large-scale" FFTs which do not fit into one GPU's global memory. It outperforms Intel's MKL by **7x** using single precision and **5x** using double precision.

4.6 N-Body Methods

N-Body methods arise in many scientific computations such as astrophysical simulations (e.g. movement of stars or galaxies), molecular dynamics and turbulent fluid simulations. The system of an N-Body method comprises N bodies which induce different forces f_{ij} (e.g. gravitational, electrical or magnetic) upon each other. A simple all-pair implementation has $\mathcal{O}(N^2)$ computational complexity which is typically impractical because of the large number of bodies.

There are various N-Body GPU implementations available such as floding@home [8], direct coulomb summation (DCS) [28, p.173f], and many more. During this subsection I will briefly introduce some of these applications, mention their performance on the GPU but refer the reader to the respective references for detailed information.

Folding@home [8] is a distributed application for biophysical processes such as protein folding. Its computing power is comprised of ordinary desktop clients, Playstation 3 gaming consoles and various GPUs whereas the GPUs contribute the most to its peak performance.

Nvidia describes a detailed implementation of a simple all-pair implementation of a astrophysical simulation in [33, p.677f]. They implement a blocked algorithm to utilize shared memory and data reuse. Moreover, they apply many other optimization techniques to achieve high performance (e.g. loop unrolling, coalescing, ...). With all optimizations applied, they achieve a peak performance of 204 GFLOP/s for a system with 16384 bodies on a 8800 GTX[5]. Bear in mind that the mentioned performance might not be a fair comparison as Nvidia as the author is biased. Furthermore, Nvidia neglects reciprocal forces[6] in order to achieve higher performance in terms of total GFLOP/s.

Kerk et al. [28, p.173f] describe the implementation of direct coulomb summation (DCS) in CUDA. They describe various optimization techniques and achieve a speedup of 44 w.r.t. an optimized single core CPU implementation. Moreover, Stone et al. [46] describe the implementation of the multilevel summation method (an approximation method for DCS) and compares it to optimized CPU and GPU DCS implementations.

Wen-Mei W. Hwu [25, p.75f] describes the most recent implementation of the *Tree-Based Barnes Hut N-Body Algorithm* which runs entirely on the GPU. This algorithm hierarchically subdivides the space into cells until a certain threshold is reached (e.g.

[5]peak performance of 345.6 GFLOP/s.

[6]$f_{ij} = -f_{ji}$

less than k bodies in a cell). This hierarchy is stored in an octree. Only particles of nearby cells interact with each other directly, particles in cells farther away are treated as one large particle centered at its center of mass. This approximation reduces the computational complexity $\mathcal{O}(N^2)$ of the precise all-pair interaction to $\mathcal{O}(NlogN)$. The implementation faces three main problems. (1) Building and traversing an irregular tree-based data structure. (2) Cope with many indirect memory accesses. (3) Avoiding recursions as they are not supported by today's GPUs. Wen-Mei W. Hwu employs old and new optimization techniques which overcome these problems. The description of these new optimization techniques is beyond the scope of this report, please refer to [25, p.75f] for further details.
Moreover, Wen-Mei W. Hwu implemented three different versions for better comparison. (1) A serial, barnes hut CPU version, (2) an all-pair GPU version and (3) a barnes hut GPU version. All versions are optimized and use single precision. The GPU barnes hut version achieves speedups of 5, 35, 66, 75 and 53 over the serial CPU version. Furthermore, even though the performance in terms of pure GFLOP/s of the all-pair GPU version is much higher than the GPU barnes hut version (304.9 GFLOP/s compared to 75.8 GFLOPs/s [25, p.91]) it trails the barnes hut version w.r.t. runtime (see Figure 4.3).

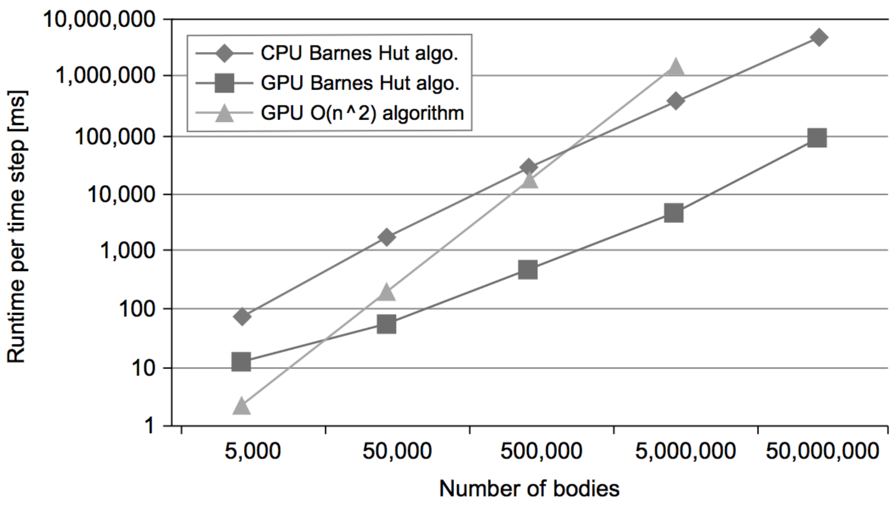

Figure 4.3: Runtime for a singe time step in milliseconds w.r.t. varying input sizes. Running on Nvidia Quadro FX 5800 and a 2.53GHz Intel Xeon E5540 respectively. [25, p.90]

4.7 Structured Grids

Structured grid algorithms appear in many scientific domains such as medical imaging, physics simulations or image processing. They subdivide the computational region into subspaces where updating an element within these subspaces depends on its neighboring elements. Hence, these applications offer much spatial locality which allows efficient use of the shared memory. Furthermore, due to their inherent parallelism and compute intense nature, structured grid applications are typically a good fit for the manycore architectures such as the GPU. Nevertheless, a major problem common to most structured grid algorithms comes from those elements at the boundary of neighboring subspaces. A trivial solution to this problem would be to terminate the kernel after one iteration and relaunch it after a global synchronization. This ,,solution" however, would not result in high performance because of the ramifications mentioned in section 4.2.

Meng et al. [32] addresses this problem and proposes a solution w.r.t. GPU implementations. They argue that it is beneficial to load redundant data (ghost data) in order to avoid synchronizations after each iteration at the cost of redundant computations. E.g. if a 4×4 block needs to be computed a 16×16 block[7] is loaded and the update for the inner 14×14 block is computed. The next iteration only computes valid updates for the inner 12×12 block and so on (see Figure 4.4). This approach allows the kernel to process 6 iterations before it writes the inner most 4×4 block back to global memory and synchronizes its execution. Furthermore, they state that it is advantageous to update the whole ghost zone even though it shrinks from one iteration to another. This choice is hardware dependent as it avoids thread divergence and further boundary checking and might not be the best choice for every underlying architecture.

In the remainder of this section I will briefly introduce three structured grid applications and related GPU references. Their performance is depicted in Figure 4.5.

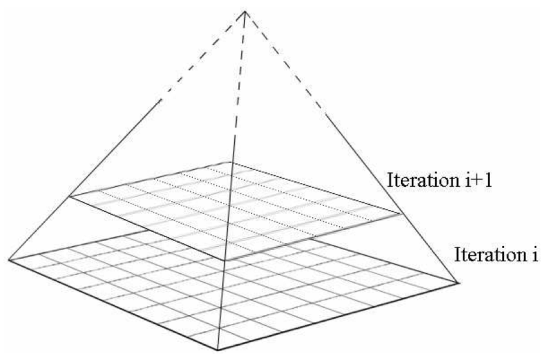

Figure 4.4: **Ghost Zone** Starting with an 8×8 block, it taks one iteration to compute the results of the inner 6×6 block [17].

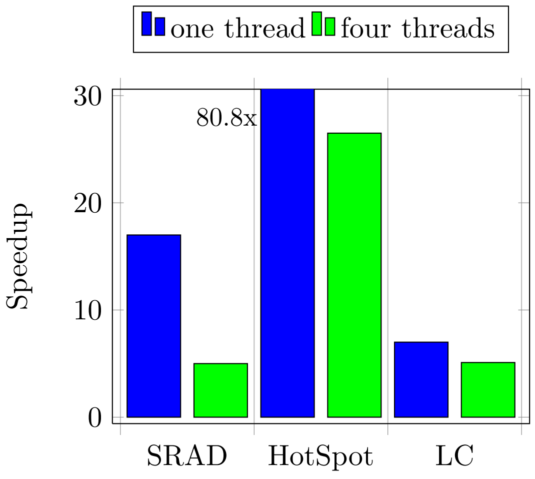

Figure 4.5: Speedup of SRAD, HotSpot and LC running on a GeForce GTX 280 over an optimized OpenMP implementation [16].

[7]This is a size should not be understood as an optimal choice as the optimal size depends on the application and the underlying architecture [32].

Speckle Reducing Anisotropic Diffusion (SRAD) is an image processing application for ultrasonic and radar images. It reduces the noise of an given image while maintaining its important features. Moreover, each element of the structured grid represents a pixel of the image. The implementation outlined by Che et al. [17] comprises three kernels. While each successive iteration over these three kernels produces an increasingly smoother image. Further GPU implementations are described in [16].

HotSpot thermal simulation which estimates the temperature of a processor based on its floor plan and simulated power measurements. Moreover, each grid cell represents the temperature of the corresponding area of the processor. GPU versions are discussed in [16], [17] and [32]. According to [17] and [32] *HotSpot* benefits from the use of ghost zones [32].

Leukocyte tracking (LC) detects and tracks leukocytes (white blood cells) in intravital microscopy videos [12]. A detailed GPU implementation including various optimization techniques is described by Boyer et al. [12] (also included in the Rodinia benchmark suite [16]). Boyer et al. applied standard optimizations such as utilization of the shared and texture memory but further improved the performance by more notable optimizations like the avoidance of *register spilling* (i.e. they are using a different algorithm for variance calculations [49]).

Further structured grid applications such as *heart wall, particle filter, cell, myocyte* and *lavaMD* can be found in the Rodinia benchmark suite [16].

4.8 Monte Carlo

Monte Carlo algorithms typically involve many parallel computations which can be easily mapped to separate threads running on the GPU. Applications such as *Monte Carlo Photo Transport* (MCPT) [25, p.247] or *Fast Simulation of Radiographic Images Using a Monte Carlo X-Ray Transport* (FSRIMC) [25, p.813], [5], [6] strongly rely on accurate *random number generators* which are already availabe for CUDA [39], [23].

The challenges of these algorithms however, lie in an efficient implementation which can cope with various problems common to both algorithms. The most notable problems are related to the fact that the execution path of each thread[8] depends on random events. Such problems are [25, p.247, p.813]:

- Thread divergence
- Uncoalesced memory accesses
- Bad cache utilization
- Warp incoherence, the execution time of the warp depends on the length of the longest photon path[9]

[8]A thread computes the path of one photon. Note, this is a simplification and does not hold for the actual implementation.

[9]A photon can be absorbed. Hence, the length of different paths can vary.

Further problems arise from the lack of atomic operations (i.e. only the Fermi architecture offers atomic operations for floating point variables. Moreover, atomic operations of previous architectures are less efficient.) and the need for new random number sequences for each photon.

Despite these problems [25, p.247] and [25, p.813] were able to achieve remarkable speedups as shown in Table 4.2.

	MCPT		FRSIMC	
	GTX 260	GTX 480	GTX 285	GTX 480
Speedup	90	415	15	34

Table 4.2: The reference CPU version for MCPT and FRSIMC is an optimized single threaded version running on an Intel Xeon processor and an optimized multi threaded version running on an Intel Q8200 processor, respectively. The data for this table is taken from [25, p.247, p.813].

4.9 Graph Traversal

Graph algorithms play a fundamental role in many scientific applications. Even though graph algorithms typically do not fit the GPU as good as other Dwarfs, there are many scientific papers available, which tackle exactly this problem [27], [29], [20], [22], [25, p.439], etc. Where [29] is probably the most notable one, as it introduces a hierarchical data structure to avoid costly synchronizations.
In the course of this section I will refer to existing GPU implementations of classical graph algorithms, show performance results, if available and comment on some problems common to most efficient graph algorithms on the GPU.

A graph G=(V,E) is typically represented as a set of adjacency lists. A mapping of this graph representation to the GPU is described by Harish et al. [22]. Moreover, most graph algorithms only provide poor spatial locality due to irregular memory accesses and data structures. Thus, coalescing and data-reuse is hard to achieve (i.e. Harish et al. did use shared memory for any of their graph algorithms). Another problem mentioned by Luo et al. [29] is the lack of global synchronizations. I like to point out that Xiao et al. [51] circumvent this problem as they introduce an inter-block barrier which runs entirely on the GPU. Moreover, as mentioned in section 4.6 it is necessary to cope with the fact that the GPU does not support recursion, as many graph-algorithm are naturally expressed recursively.

Breadth First Search (BFS) visits all nodes in a graph G=(V,E) which can be reached from a particular node in G. The most notable papers which outline a GPU implementation of BFS are [22], [20] and [29]. It is worth mentioning that the implementations of [22] and [20] increase the computational complexity of the best known serial algorithm whereas [29] does not do so. Thus, Table 4.3 shows the results achieved by Luo et al. [29] while leaving the results of [22] and [20] unnoticed.

Graph type	#Vertex	IIIT-BFS	CPU-BFS	UIUC-BFS	Speedup
Regular	10M	11019.8ms	5023.0ms	488.0ms	10.3
Real World	6,3M	2579.4ms	1323.0ms	236.6ms	5.6
Scale-free	10M	2252.8ms	506.9ms	483.6ms	1.05

Table 4.3: Runtime of three different BFS implementations. IIIT-BFS, CPU-BFS and UIUC-BFS denote the GPU implementation of Harish et al., an efficient CPU implementation by Cormen et al. [18] and the GPU implementation of Luo et al., respectively. Results taken from [29].

All Pairs Shortest Path (APSP) finds the shortest path in a weighted Graph G=(V,E) between all nodes. The GPU implementation of APAP is outlined by Katz et al. [27] and Harish et al. [22]. The implementation of Katz et al. runs up to **6.5x** faster that the implementation of Harish et al., this is mostly due to shared memory utilization. Please refer to [22] and [27] for further information.

Single Source Shortest Path (SSSP) finds the shortest path in a weighted Graph G=(V,E) between one particular node to all other nodes. An efficient GPU implementation for SSSP is outlined by Harish et al. [22]. They achieved speedups up to **70x**, please refer to [22] for more detailed information.

Maximum Flow (MF) finds the maximum flow in a flow network N=(V,E) from a source $s \in V$ to a sink $t \in V$. The push-relabel algorithm is one of the most efficient algorithms to compute the MF, its GPU implementation is outlined by Narayanan et al. [25, p.439f]. They stated that it maps well to the CUDA and it exhibits good data parallelism. Their GPU implementation of the push-relabel algorithm improved over a different CPU implementation[10] on average by **12x** across different inputs [25, p.439f].

5 Effort

This section describes the effort needed by an newcomer to CUDA to become familiar with the API, the fundamental principles of GPGPU and the time needed to port an existing CPU application to CUDA. This section is described from the view of the author.

5.1 The Learning Period

Since the CUDA API follows the style of C/C++, it is relatively easy to write basic GPU applications such as SAXPY, SDOT, etc.

I started out by reading [45] (approx. 300 pages) which took me around one week to work through it. By that time, I had written my first GPU applications but I have only gained very basic expirience with the underlying architecture and optimization techniques. Hence, I decided to read [28], which comprises another 250 pages. After

[10]The comparison is still valid as MF finds a global optimal solution. Hence, the results are the same.

reading the second book, I was familiar with further optimization techniques and parts of the underlying hardware. In order to reinforce my knowledge of CUDA and GPGPU, I opt to skim through [37] and [36]. This only took a very moderate amount of time since they mostly recapitulate [45] and [28] but still offer deeper insights into certain areas such as the different GPU architectures, memories, etc.

Unfortunately, none of the literature mentioned above covers topics such as debugging or libraries in detail. This requires even further learning effort. Nevertheless, Nvidia provides a rich amount of documentations and webinars which cover these topics. I want to stress that Nvidia's documentations and their webniars are easily comprehensible. Moreover, Nvidia offers many programming examples in the CUDA SDK which further eases the use of CUDA.

All in all, it is fair to say that CUDA is easy to learn since this whole process took me just a little more than two weeks of full-time labor.

5.2 Case Study

I carried out a case study to depict the time involved in order to port an existing CPU implementation to CUDA. Moreover, I started out by a naive GPU implementation and illustrate the importance of different optimization techniques and their impact on performance. I want to stress that this case study is not meant to be a fair comparison between GPU and CPU since I did not spent an equal amount of time on the optimization of the CPU and GPU implementation.

5.2.1 Problem Description

The problem can be described as follows:
Given a matrix $A \in \mathbb{R}^{n+k \times k}$, two vectors $b \in \mathbb{R}^{n+k}$, $x \in \mathbb{R}^k$ and a norm $p \in \mathbb{N}$ compute

$$||A \times x - b||_p^p$$

whereas $k \gg n$ and A and b have structures as depicted in Figure 5.1. Furthermore, all the entries of the „lower diagonal" of A are equal (i.e. $a_{i+n,i} = \lambda \in \mathbb{R}\ \forall i \in \{1,2,\ldots,k\}$). Moreover, n is typically very small (e.g. $n = 128$), whereas k can reach some hundred thousends.

The problem mainly involves three parts, a dense part, a sparse part and a reduction. Hence, it comprises Dwarf 1 and 2.

5.2.2 GPU Implementation

In order to show the tremendous impact of optimization techniques on the execution time I decided to implement several versions, where each succeeding version will improve over its predecessor. Since this report does not focus on optimization techniques I will just name the applied techniques and focus on the effort. The interested reader is referred to [28], [12] or [36] for further details. All GPU versions are run on a Nvidia Tesla C2050 graphics card. Depending on the version, they mainly consist of two to three kernels (i.e. dense, sparse and reduction).

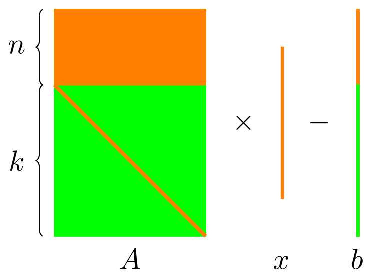

Figure 5.1: Structure of A, x and b. Green denotes zero entries, where orange denotes non-zero entries.

I also parallelized the existing CPU version using OpenMP. This version is compiled with gcc 4.5 with the optimization flag -O3 and it runs 4 threads on an Intel E5620 processor. I like to stress that the resulting execution times of the CPU version should not be understood as a fair GPU vs CPU comparison since I did not spent equally much time in optimizing the CPU and GPU version.

A brief overview of all GPU implementations is shown in Table 5.1.

Version 1 is a naive GPU implementation which is quite similar to the CPU implementation except that it runs several hundred threads whereas each threat computes only one entry $y_i = |A_i * x - b(i)|^p$ where A_i denotes a row of A. After the computation the data is transfered back to the CPU which in turn is responsible for the sum reduction over all y_i.
The implementation was straight forward and took only some minutes. Nevertheless, as mentioned before, n is very small, thus this naive implementation does not utilize all the 14 SMs of the Telsa C2050 (i.e. with a blocksize of 64 threads only $128/64 = 2$ of 14 SMs are used, leaving 12 SMs idle).

Version 2 is a blocked version of dense kernel of *version 1.* Hence, it is able to utilize all the cores of the GPU. As it is shown in Table 5.2 it yields a major increase in performance.
Concerning the effort, this version involved a major redo of the algorithm. In my opinion the implementation using CUDA is no more difficult than it is with OpenMP.

Version 3 ports the reduction of the sparse part (i.e. $result += (\lambda * x(i+n))^p \; \forall i \in \{1,2,...,k\}$) to the GPU. The reduction was a little more complicated than it would be on a CPU since CUDA does not allow synchronization across different blocks. Moreover, one has to pay attention to so called thread divergence [28] in order to attain high performance.

Version 4 is another major step towards high performance as it introduces shared memory usage for the dense part of the computation. The availability of shared memory on the GPU requires different algorithms than those for a CPU. Even though one has be be familiar with the memory hierarchy of the GPU and the possibility of coalescing memory accesses to global memory, these concepts are not very difficult. Furthermore,

[36] depicts many good programming examples which make the implementations of own algorithms quite easy.

Version 5 is yet another improvement over *version 4*. *Version 5* pads the shared memory to avoid so called bank conflicts [28]. Once, one has understood how the GPU accesses shared memory this implementation is obvious.

Version 6 applies loop unrolling which results in more *task level parallelism* [28]. The underlying concept and implementation of loop unrolling is very easy.

Version 7 makes use of streams which allow simultaneous execution of the dense and sparse kernel. Moreover, it uses so called pinned memory [37] for asynchronous memory transfers. These concepts are also available in OpenMP. Thus, a programmer familiar with OpenMP will not have any difficulties using these concepts in CUDA.

With the good speedups achieved by *version 7* the reduction of the dense part takes up a large fraction of the execution time. Thus, it becomes necessary to perform the reduction in parallel on the GPU which results **version 8**. Since the data is already on the GPU no extra data transfer is required. The implementation was quite simple because the reduction is similar to the one of *version 3*.

Version 9 is not really a new version for itself since it merely tweaks the parameters such as blocksizes, loop unrolling level, etc. The parameter seemed to be chosen in a decent way already since *version 9* does not yield a significant speedup. Nevertheless tuning the parameter is an important part since suboptimal settings can drastically affect the performance of an application.

Version	Description
1	Dense and Sparse computation on GPU
2	Dense part is now blocked
3	Reduction of the sparse part on GPU
4	Uses shared memory and coalesced memory accesses
5	Padding of shared memory avoids bank conflicts
6	Loop unrolling increases task level parallelism
7	Streams allow simultaneous execution and memory transfers
8	Reduction of the dense part on GPU
9	Tweaking of the parameters

Table 5.1: Overview of the different optimization techniques of the GPU versions

5.2.3 Results

Table 5.2 shows the speedups achieved by the different GPU versions over the parallelized CPU implementation. The speedups shown in this table do not account for the initial memory transfer between host and device. This table clearly shows the importance of the optimization techniques as the performance of each successive version increases. All in all, the implementation of all versions took little less than two days, but it required a lot of preparation in advance to get to know all these optimization techniques as men-

tioned in section 5.1.
During the implementation phase, I spent a large amount of time debugging using Allinea's DDT. DDT is a graphical debugger which also incorporates cuda-memcheck which allows to find bad memory accesses. It was quite handy and I was able to find most of the bugs but it still seems to be little buggy itself as it sometimes shows wrong values of variables.
Finally, the speed up of version 2 over its predecessor requires a little explanation as it seems to be unusually high. This is due to the fact that most of the cores used by version 1 are idle, whereas version 2 utilizes all cores and further introduces *thread level parallelism*[28].

A further version which utilizes the constant memory for x and versions with double precision support are left as future work.

	n = 128 k = 32,000		n = 128 k = 3,200,000	
	Time [ms]	Speedup	Time [ms]	Speedup
CPU	5	-	320	-
GPU 1	32.02	0.16	4852.6	0.07
GPU 2	3.35	1.49	371.5	0.86
GPU 3	2.77	1.81	257.4	1.24
GPU 4	1.86	2.69	174.1	1.84
GPU 5	0.89	5.62	72.7	4.4
GPU 6	0.57	8.77	44	7.27
GPU 7	0.47	10.64	37.6	8.51
GPU 8	0.32	15.63	22.8	14.04
GPU 9	0.31	16.13	22.4	14.29

Table 5.2: Speedups achieved over the parallelized CPU implementation using single precision.

6 Conclusion

During the course of this report we have seen ...

- ... very promising speedups over traditional CPU implementations.
- ... that it remains challenging to overcome the problems related to an efficient GPU implementations.
- ... some optimization techniques and novel data structures
- ... available GPU benchmark suites.
- ... available GPU Dwarfs and related libraries.

Moreover, section 5 outlined the effort involved becoming familiar with CUDA and to port an existing CPU application to the GPU. Furthermore, despite the good speedups there is no clear answer whether it is beneficial to port an application to the GPU or ,,just" write an efficient parallelized CPU version. This is mostly due to the additional effort needed to write a GPU application and the fact that some applications are naturally a better fit for the GPU than others (e.g. dense linear algebra algorithms are often much easier to port than graph algorithms).

Finally, I think that the trend towards GPGPU is likely to continue. Hence, we will see new data structures and new optimization techniques which will allow to port even further applications to the GPU. Moreover, I think that the concepts of GPUs and CPUs are likely to converge. The Fermi architecture with its full memory hierarchy, ECC, improved double precision performance, etc. was a major step towards general purpose computing. CPUs on the other hand tend to improve their vector processing powers. More precisely, Intel announced its next architecture called *Haswell* which is expected to further improve its vector capabilities by supporting AVX 2.

References

[1] MediaBench. http://euler.slu.edu/ fritts/mediabench/, June 2011.

[2] Standard Performance Evaluation Corporation. http://www.spec.org/cpu2006/, June 2011.

[3] TOP500 Supercomputing site. http://www.top500.org/, June 2011.

[4] K. Asanovic, R. Bodik, B.C. Catanzaro, J.J. Gebis, P. Husbands, K. Keutzer, D.A. Patterson, W.L. Plishker, J. Shalf, S.W. Williams, et al. The Landscape of Parallel Computing Research: A View from Berkeley. Technical report, Citeseer, 2006.

[5] A. Badal and A. Badano. Accelerating Monte Carlo simulations of photon transport in a voxelized geometry using a massively parallel graphics processing unit. *Medical physics*, 36:4878, 2009.

[6] A. Badal and A. Badano. Monte Carlo simulation of X-ray imaging using a graphics processing unit. In *Nuclear Science Symposium Conference Record (NSS/MIC), 2009 IEEE*, pages 4081–4084. IEEE, 2009.

[7] M.M. Baskaran and R. Bordawekar. Optimizing sparse matrix-vector multiplication on GPUs. *IBM research report RC24704, IBM*, 2009.

[8] A.L. Beberg, D.L. Ensign, G. Jayachandran, S. Khaliq, and V.S. Pande. Folding@ home: Lessons from eight years of volunteer distributed computing. 2009.

[9] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. In *Proc. ACM/IEEE Conf. Supercomputing (SC), Portland, OR, USA*, 2009.

[10] Nathan Bell and Michael Garland. Cusp: Generic Parallel Algorithms for Sparse Matrix and Graph Computations. http://cusp-library.googlecode.com, 2011. Version 0.2.0.

[11] C. Bienia, S. Kumar, J.P. Singh, and K. Li. The PARSEC benchmark suite: Characterization and architectural implications. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 72–81. ACM, 2008.

[12] M. Boyer, D. Tarjan, S.T. Acton, and K. Skadron. Accelerating leukocyte tracking using CUDA: A case study in leveraging manycore coprocessors. 2009.

[13] V. Boyer, D. El Baz, and M. Elkihel. Solving knapsack problems on gpu. *Computers & Operations Research*, 2011.

[14] L. Buatois, G. Caumon, and B. Lévy. Concurrent number cruncher: a GPU implementation of a general sparse linear solver. *International Journal of Parallel, Emergent and Distributed Systems*, 24(3):205–223, 2009.

[15] N. Cascarano, P. Rolando, F. Risso, and R. Sisto. iNFAnt: NFA pattern matching on GPGPU devices. *ACM SIGCOMM Computer Communication Review*, 40(5):20–26, 2010.

[16] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, S.H. Lee, and K. Skadron. Rodinia: A Benchmark Suite for Heterogeneous Computing. In *Workload Characterization, 2009. IISWC 2009. IEEE International Symposium on*, pages 44–54. IEEE, 2009.

[17] S. Che, M. Boyer, J. Meng, D. Tarjan, J.W. Sheaffer, and K. Skadron. A performance study of general-purpose applications on graphics processors using CUDA. *Journal of Parallel and Distributed Computing*, 68(10):1370–1380, 2008.

[18] T.H. Cormen. *Introduction to algorithms.* The MIT press, 2001.

[19] A. Danalis, G. Marin, C. McCurdy, J.S. Meredith, P.C. Roth, K. Spafford, V. Tipparaju, and J.S. Vetter. The Scalable Heterogeneous Computing (shoc) Benchmark Suite. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 63–74. ACM, 2010.

[20] Y.S. Deng, B.D. Wang, and S. Mu. Taming irregular EDA applications on GPUs. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 539–546. ACM, 2009.

[21] N.K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete Fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 2. IEEE Press, 2008.

[22] P. Harish and P. Narayanan. Accelerating large graph algorithms on the GPU using CUDA. *High Performance Computing–HiPC 2007*, pages 197–208, 2007.

[23] L. Howes and D. Thomas. Efficient random number generation and application using CUDA. *GPU gems*, 3:805–830, 2007.

[24] J. Humphrey, D. Price, K. Spagnoli, A. Paolini, and E. Kelmelis. CULA: Hybrid GPU accelerated Linear Algebra Routines (Proceedings Paper). 2010.

[25] Wen-Mei W. Hwu. *GPU Computing Gems (Applications of Gpu Computing)*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2011.

[26] A. Jaleel, M. Mattina, and B. Jacob. Last level cache (LLC) performance of data mining workloads on a CMP-a case study of parallel bioinformatics workloads. In *High-Performance Computer Architecture, 2006. The Twelfth International Symposium on*, pages 88–98. IEEE, 2006.

[27] G.J. Katz and J.T. Kider Jr. All-pairs shortest-paths for large graphs on the GPU. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55. Eurographics Association, 2008.

[28] D.B. Kirk and W.H. Wen-mei. *Programming massively parallel Processors: A Hands-on approach*. Morgan Kaufmann Publishers Inc. San Francisco, CA, USA, 2010.

[29] L. Luo, M. Wong, and W. Hwu. An effective GPU implementation of breadth-first search. In *Proceedings of the 47th Design Automation Conference*, pages 52–55. ACM, 2010.

[30] S. Manavski and G. Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC bioinformatics*, 9(Suppl 2):S10, 2008.

[31] Wen mei W. Hwu. Parboil Benchmark suite. http://impact.crhc.illinois.edu/parboil.php, June 2011.

[32] J. Meng and K. Skadron. Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs. In *Proceedings of the 23rd international conference on Supercomputing*, pages 256–265. ACM, 2009.

[33] NVIDIA. *GPU Gems3: Programming Techniques for High-Performance Graphics and General-Purpose Computation*. Addison-Wesley, 2007.

[34] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. *NVIDIA Whitepaper*, 2009.

[35] NVIDIA. CUBLAS Library User Guide. http://developer.download.nvidia.com/compute /DevZone/docs/html/CUDALibraries/doc/CUBLAS_Library.pdf, June 2011.

[36] NVIDIA. CUDA C Best Practices Guide, June 2011. Version 4.0.

[37] NVIDIA. CUDA C Programming Guide, June 2011. Version 4.0.

[38] NVIDIA. CUFFTLibrary User Guide. http://developer.download.nvidia.com/compute /DevZone/docs/html/CUDALibraries/doc/CUFFT_Library.pdf, June 2011.

[39] NVIDIA. CURAND Library User Guide. http://developer.download.nvidia.com/compute /DevZone/docs/html/CUDALibraries/doc/CURAND_Library.pdf, June 2011.

[40] NVIDIA. CUSPARSE Library User Guide. http://developer.download.nvidia.com/compute /DevZone/docs/html/CUDALibraries/doc/CUSPARSE_Library.pdf, June 2011.

[41] D. Patterson. The top 10 Innovations in the new Nvidia Fermi Architecture, and the top 3 next Challenges. *NVIDIA Whitepaper*, 2009.

[42] V. Podlozhnyuk and M. Harris. Monte carlo option pricing. *nVidia Corporation Tutorial*, 2008.

[43] S. Ryoo, C.I. Rodrigues, S.S. Baghsorkhi, S.S. Stone, D.B. Kirk, and W.W. Hwu. Optimization Principles and Application Performance Evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 73–82. ACM, 2008.

[44] P. Du S. Tomov, R. Nath. Magma: Matrix Algebra on GPU and Multicore Architectures. http://icl.cs.utk.edu/magma/index.html, June 2011. 1.0 RC5.

[45] J. Sanders and E. Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley, 2010.

[46] J.E. Stone, J.C. Phillips, P.L. Freddolino, D.J. Hardy, L.G. Trabuco, and K. Schulten. Accelerating molecular modeling applications with graphics processors. *Journal of Computational Chemistry*, 28(16):2618–2640, 2007.

[47] V. Volkov and J.W. Demmel. Benchmarking GPUs to Tune Dense Linear Algebra. In *High Performance Computing, Networking, Storage and Analysis, 2008. SC 2008.*, pages 1–11. IEEE, 2008.

[48] V. Volkov and B. Kazian. Fitting FFT onto the G80 architecture. *University of California, Berkeley*, 2008.

[49] BP Welford. Note on a Method for calculating corrected Sums of Squares and Products. *Technometrics*, 4(3):419–420, 1962.

[50] S.C. Woo, M. Ohara, E. Torrie, J.P. Singh, and A. Gupta. The SPLASH-2 programs: Characterization and methodological considerations. In *ACM SIGARCH Computer Architecture News*, volume 23, pages 24–36. ACM, 1995.

[51] S. Xiao and W. Feng. Inter-block GPU communication via fast barrier synchronization. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12. IEEE.