TECHNISCHE UNIVERSITÄT
MÜNCHEN

Lehrstuhl für Rechnertechnik und Rechnerorganisation /
Parallelrechnerarchitektur

Advanced Optimization Techniques for Sparse Grids on Modern Heterogeneous Systems

Alin Florindor Muraraşu

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Vorsitzender:	Univ.-Prof. Dr. M. Bichler
Prüfer der Dissertation:	
	1. Univ.-Prof. Dr. A. Bode
	2. Univ.-Prof. Dr. H.-J. Bungartz

Die Dissertation wurde am 25.03.2013 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 25.05.2013 angenommen.

Abstract

GPU accelerated systems are heterogeneous systems characterized by a peak performance in the order of TFlop/s obtained using a large number of cores and wide vector units. Moreover, GPUs provide an advantageous ratio between performance and power consumption. However, reaching high efficiency on GPUs is often a difficult task whose successful completion requires advanced optimization techniques. In order to fit an application to GPUs, redesigning data structures and algorithms is often necessary so that they consume less memory and become more vector friendly. For extra performance, an empirical optimization method is required in order to cope with non-trivial interactions between optimization parameters characteristic to GPU programs. On heterogeneous systems, another key problem is the distribution of the computational work among the different processors. This thesis proposes solutions to these challenges in the context of the sparse grid technique, a numerical technique that addresses the curse of dimensionality problem arising in high-dimensional settings such as computational steering. The performance results presented here validate a set of advanced optimization techniques that allow for efficiently porting sparse grid algorithms to GPUs, for improving the performance using an empirical optimization method, and for increasing the utilization of the system using load balancing.

Acknowledgements

Many people contributed to the success of this work. First of all, I would like to thank my Doktorvater, Prof. Dr. Arndt Bode, for his help, advice, and especially for offering me the chance to work in a very supportive research environment at the Lehrstuhl für Rechnertechnik and Rechnerorganisation (LRR). I would like to thank my mentor, Josef Weidendorfer, for his help not only on the technical side but on all research related aspects. I am thankful to all the members of LRR (too many to be named individually) who contributed to a four year experience that made me a better researcher and more importantly a better person.

I would like to thank my collaborators from Prof. Dr. Hans Bungartz's group, Gerrit Buse, Dirk Pflüger, Daniel Butnaru, Tobias Weinzierl, people on whose help I could always count. I learned a lot from them and for this I will always be grateful.

Last but not least, I am very thankful to my family, Georgiana, Lenuţa, Oana, and Florin, for providing me with the comfort necessary to complete this four year effort.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter presents the motivation for the use of Graphics Processing Unit (GPU) based heterogeneous systems for accelerating parallel applications. An overview of the advanced optimization techniques proposed in this thesis is provided, including an empirical optimization method and load balancing. The chapter introduces the application used for validating the optimizations, namely a computational steering application in which high-dimensional simulation data is efficiently handled using the sparse grid technique. At the end of the chapter, the contributions of the thesis are summarized and the structure of the thesis is presented.

1.1 Motivation

Heterogeneous systems are very popular nowadays because they allow for higher processing speeds to be achieved without sacrificing power efficiency. Their main disadvantage is that heterogeneous systems are difficult to program efficiently. Often, advanced optimization techniques are required in order to achieve a high level of performance. Such techniques are proposed in this thesis and are validated using routines critical for the performance of a computational steering application. Since this application deals with high-dimensional simulation data that needs to be visualized in real-time, the main requirement is to fully exploit the computational power of GPU based heterogeneous systems. The thesis has two main directions: The first refers to understanding and addressing some of the most important challenges of heterogeneous computing, while the second is more application specific as it refers to the computational steering application which has to be parallelized and optimized for modern hardware.

Faced with the power wall, processor architects adopted the multi-core solution in which the computational speed of the hardware increases by replicating the execution engines (cores) within a CPU. This allows for the power consumption to remain at an acceptable level. However, the trade-off is that programmers now have to parallelize their applications. Even more performance and a higher Flop/s (Floating Point Operations per Second) / Watt ratio can be obtained using general purpose accelerators, e.g. GPUs, which can generally be described as many-core processors containing a set of so-called wimpy cores, i.e. simple in-order execution engines that do not include speculative logic for branch prediction, cache prefetching, etc. A notable property of accelerators is that they cannot operate independently from CPUs whose main responsibility in this context is to offload tasks to accelerators. Often, the transistor budget of accelerators is allocated to wide vector units. Multithreading is a rather inexpensive technique which can provide accelerators with the means to hide the latency of the instruction pipeline by interleaving the execution of a large number of threads per core. Given their large number of cores and wide vector units, accelerators are not suitable for all classes of applications, especially for those that are not vectorizable or do not contain enough data parallelism. Nevertheless, there are many applications from computer graphics and scientific computing that harness the strengths of accelerators, thus explaining their success.

Nowadays, a typical heterogeneous system contains CPUs and GPUs. CPUs can be seen as latency oriented processors [1] in the sense that they are optimized for efficiently executing a serial stream of instructions. They incorporate large caches and implement techniques for branch prediction and data prefetching. Additionally, CPUs contain complex logic (out-of-order) for automatically extracting the Instruction Level Parallelism (ILP) from a sequence of instructions. In contrast to CPUs, GPUs are throughput oriented [1] or massively parallel processors, making them a perfect fit for data parallel computations. In terms of peak performance, GPUs are generally up to two orders of magnitude faster than one CPU core. GPUs typically contain more cores than CPUs, have wider vector units, and operate at a lower clock frequency. A GPU has its dedicated memory which has approximately one order of magnitude more bandwidth than the memory of the CPU but its size is rather small, e.g. up to 6 GB. The GPU memory is separated from the CPU memory by a PCI Express (PCIe) bus whose bandwidth is lower than the bandwidth of both the CPU memory and the GPU memory. In this respect, a GPU based heterogeneous system can be considered a distributed memory system.

Understanding the challenges of GPUs is of high importance when porting applications to GPUs. Most importantly, their vector nature imposes restrictions on applications. Vector based architectures are characterized by the fact that the same operation is executed simultaneously on multiple chunks of data, a processing model referred to as Single Instruction Multiple Data (SIMD). Such an execution puts constraints on the placement of the data in memory, i.e. the data must be aligned and the chunks must be consecutive words in memory. Moreover, the control flow is another crucial element determining the efficiency of a vector processor. In the presence of branches, undesired situations might occur in which different operations must be executed on different data chunks. However, this violates the SIMD requirements and cannot be executed at once by a vector unit, meaning that the different vector operations are serialized and during the execution of each operation, only a subset of the lanes of the vector unit is actually utilized. In such a scenario, the obtained performance is suboptimal. From here, one can imagine that complex data structures and recursive algorithms that are control flow dependent, cannot be paired efficiently with GPUs. Therefore, it is essential to analyze whether such data structures and algorithms can be replaced with more GPU friendly versions.

Memory consumption can also become a challenge on a GPU taking into account that the GPU memory is a scarce resource. Thus, it is important that the data structures copied to the GPU have a small memory footprint. Although the GPU memory is faster than the CPU memory, the PCIe bus acts as a severe performance bottleneck for applications characterized by a low ratio between GPU computation and PCIe communication.

Most GPU programs expose parameters whose values control the optimizations applied to the code or influence the performance behavior, e.g. parameters that control multithreading, locality, and parallelism granularity. The objective is to find the values for the optimization parameters that maximize the performance. However, complex interactions between parameters cannot be addressed effectively using theoretical methods, e.g. performance models. In this context, an empirical optimization method, or search based auto-tuning, tries to find the optimal values for the parameters by evaluating the performance for different combinations of values assigned to the optimization parameters. In the absence of auto-tuning, a programmer is often required to fully understand non-trivial characteristics of the GPU in order to determine the best values of the parameters. Furthermore, hardware details are not always available. The alternative is to use auto-tuning which employs search algorithms for exploring the optimization space in order to improve the performance.

Load balancing is another key problem on heterogeneous systems. It refers to finding the most efficient mapping between a parallel computation and the heterogeneous processors. In general, load balancing on heterogeneous systems is accomplished by (over)decomposing the computational work into many small tasks which are dynamically assigned to the CPU and the GPU for execution. The main requirement is that algorithms must have multiple implementations, i.e. a version for the CPU and one for the GPU. When a task is assigned to a processor, the version corresponding to that processor's type is invoked. Task based load balancing is affected by scheduling overheads and the grain (or task) size problem, i.e. determining the optimal grain size that generates sufficient parallelism without amplifying the effect of overheads, and makes the best use of the CPU and the GPU. Besides task based schemes, for computations that are data parallel, another load balancing scheme can be applied based on dividing the work into two chunks, one for the CPU and another one for the GPU. The condition is that the chunks must ensure that the CPU and the GPU finish processing at the same time. Such a scheme can be referred to as static load balancing. Its main benefit is that it does not suffer from the grain size problem. However, the disadvantage is that it is less adaptive, i.e. the initial work distribution cannot be modified. Adaptivity is important for input dependent programs and on non-dedicated systems where it is common that multiple running applications interfere with each other.

All the techniques mentioned above, i.e. redesigning data structures and algorithms, auto-tuning, and load balancing, are applied to the sparse grid technique [2], a numerical technique used for building numerical approximations of high-dimensional functions. Sparse grids address the curse of dimensionality problem, i.e. the number of grid points required by the representation of a function using a full grid depends exponentially on the number of dimensions. If a function is sufficiently smooth, then the sparse grid approximation offers an accuracy close to the one obtained using full grids but with considerably less points. Consequently, the sparse grid technique provides lossy compression functionality. In a computational steering application, sparse grids are used to compress high-dimensional simulation data. The compressed data is stored in a database and is later decompressed for visualization at interactive rates.

Both the compression and the decompression are obtained through recursive sparse grid algorithms. Moreover, complex key-value based data structures, e.g. hash tables and trees, are the typical solutions used for storing the sparse grid points and their corresponding values. Hence, in its initial form, the sparse grid technique is highly incompatible with GPUs. However, this thesis shows that by redesigning the data structures and the algorithms, an efficient implementation of sparse grids for GPU based heterogeneous systems can be achieved.

1.2 Scientific Contribution

There are three major contributions of this thesis:

- the porting of the sparse grid technique to GPUs (Chapter 4); the GPU implementation is based on a data structure characterized by minimal memory footprint (Chapter 3)

- an auto-tuning method for GPU programs (Chapter 5)

- load balancing schemes for data parallelism on heterogeneous systems (Chapter 6).

The first major contribution includes **a data structure for sparse grids based on a bijective mapping**. The data structure is at the core of **the first GPU implementation of the sparse grid technique**, developed as part of this thesis. The bijection eliminates the need to store the coordinates of grid points, meaning that the sparse grid is stored as a sequence of values ordered in a special way so that the index of every value can be transformed using the bijection into a multi-dimensional point of the grid and vice versa. The consequence is that the memory footprint of sparse grids is minimal. This property is especially beneficial in the context of GPUs because of their limited amount of memory, e.g. up to 6 GB. Using the bijection based data structure, bigger problems can be solved on the GPU. In addition to the data structure, **redesigned non-recursive sparse grid algorithms** are proposed together with a **comprehensive set of CPU and GPU optimizations** that improve locality and make better use of vector units. Furthermore, **input specialized algorithms for sparse grids** are proposed that exploit common patterns found in the input data. Although the bijection and the non-recursive algorithms especially address the constraints of GPUs, i.e. reduced memory and unsupported (or inefficient) recursion, they are also advantageous for CPUs.

Second, a **search based auto-tuning solution** is proposed for improving the GPU optimized implementations of the sparse grid algorithms. The cost of auto-tuning comes in the form of the time spent in searching for the optimal performance. Moreover, auto-tuning often deals with high-dimensional and highly unstructured search spaces which are difficult to explore efficiently. This thesis proposes a set of optimization parameters that are exported by the optimized GPU routines of the sparse grid technique. The parameters and their ranges are provided as input to a search engine whose responsibility is to find the values of the parameters that result in the best performance. In the context of auto-tuning, a contribution is represented by **search partitions**, i.e. a partition groups together interdependent optimization parameters and every partition is orthogonal to all the other partitions. In general, this results in a pruning of the search space and does not miss the optimum provided that correct assumptions are made regarding the dependent (or independent) parameters. In order to accelerate the auto-tuning process, an **input reduction technique** is proposed that reduces the size of the input data while ensuring that the global behavior for the initial data is captured. Auto-tuning is then used to optimize the GPU routines using the reduced data, thus executing in less time.

The third major contribution is a **load balancing solution** for sparse grids. A separation is made between sparse grid routines that allow for both the CPU and the GPU to be used simultaneously and other routines that do not. In the latter case, the scheduling problem is reduced to indicating the processor, i.e. CPU or GPU, that provides the best performance depending on the input parameters. Such situations are caused by complex data access patterns that generate excessive communication over PCIe in order to ensure data consistency when the CPU and the GPU are both used at the same time. Other routines engage all the heterogeneous processors in the computation. Two load balancing schemes, dynamic and static, are employed for addressing the data parallelism in those functions. As the dynamic approach is affected by the grain size problem, a **multi-grain load balancing algorithm** is proposed: It ensures on one hand that the fastest processor, e.g. GPU, is never idle (a situation encountered in the case of a single grain size). On the other hand, in a multi-grain approach, the tasks have sizes that allow for an efficient use of the CPU and of the GPU, e.g. the task size for the CPU allows for harnessing locality at cache level, while the task size for the GPU allows for the full utilization of cores, vector units, and multithreading. The proposed **static load balancing approach** divides the work into two chunks for the CPU and the GPU based on approximations of the execution time as a function of work for the CPU and the GPU.

Although the optimizations proposed in this thesis focus on the sparse grid technique, a subset is also applicable to other applications, e.g. dense and sparse linear algebra, stencil computation, direct n-body method. More precisely, the proposed techniques for auto-tuning and load balancing can be directly transferred to other GPU programs.

The software developed as part of this work has two main components:

- The sparse grid library, **fastsg**, is a full implementation of the sparse grid technique, including boundaries, dimensional truncation, and input specialized routines.
- The sparse grid benchmark, **sgbench**, is a simplified implementation of zero-boundary sparse grids. It captures the performance behavior of fully functional implementations.

The motivation behind the benchmark comes from the necessity to evaluate the fastest hardware for sparse grid applications without spending too much time in porting the code. Both the library and the benchmark incorporate optimized CPU and GPU versions of sparse grid algorithms, a search engine that contains the core functionality of auto-tuning, and a scheduler that distributes the computational work among all the processors in a heterogeneous system.

1.3 The Structure of the Thesis

Chapter 2 describes the current processor landscape shaped by the power wall, one of the main reasons behind the appearance of multi-core CPUs and of many-core accelerators such as GPUs. In Chapter 3, the theory behind the sparse grid technique is detailed, including the use of sparse grids for handling high-dimensional data in a computational steering application. Different data structures and algorithms for sparse grids are also covered. Among the data structures, emphasis is placed on a bijection based data structure. Chapter 4 presents the details behind the porting of the sparse grid technique to GPUs. Moreover, CPU and GPU optimizations are described that allow for an efficient utilization of the memory hierarchy and of the vector units on CPUs and GPUs. Chapter 5 describes an empirical optimization method that incorporates common GPU optimization parameters, a search method based on partitioning the optimization parameter space, and an input reduction technique that shortens the auto-tuning time. The central topic of Chapter 6 is load balancing whose main objective is to combine the computational speed of the CPU and of the GPU. Dynamic and static schemes for load balancing are explained. Finally, Chapter 7 contains the conclusion, shows the optimization progress for the sparse grid routines, and outlines directions for future work.

Chapter 2

A Complex Hardware and Software Landscape

Accelerators are special purpose processors characterized in general by a high GFlop/s rate and a high Flop/s per Watt ratio. This makes them especially well suited for addressing the power wall problem, i.e. nowadays, it is extremely difficult to increase the frequency of processors while keeping their power consumption at an acceptable level. This chapter presents the power wall and a solution to it in the form of accelerators. The differences between general purpose CPUs and accelerators are covered. More precisely, CPUs are described as latency oriented processors whereas accelerators are throughput oriented. The chapter provides an overview of accelerators. Among them, emphasis is placed on Nvidia GPUs. Besides the presentation of hardware aspects, the chapter also compares CPUs and GPUs in terms of programming models. The challenges of GPU computing are discussed at the end of the chapter.

2.1 The Power Wall and the ILP Wall

For more than 20 years, the transistor density in a processor has doubled every 2 years. This tendency was captured by an empirical law called Moore's Law. Increasing the density allowed for the performance to grow at the same rate, meaning that new generations of processors enabled many applications to execute two times faster than before almost for free. The 2x speedup is the result of two essential techniques: frequency scaling and microarchitecture optimizations.

Frequency scaling is explained by Dennard's scaling recipe [3] which states that if the area of a transistor is decreased by a factor of 2, then the frequency, f, of the processor is increased by 1.4x, the supply voltage, V, is decreased by 0.7x, and the capacitance, C, decreases by 0.7x [4].

Consider the equation for the active power of a processor:

$$\boxed{P = C \cdot V^2 \cdot f.} \tag{2.1}$$

Assuming that the number of transistors is the same before and after shrinking, the power consumption is reduced by a factor of 2. A notable aspect is that using frequency scaling, only 1.4x more performance can be achieved. In the absence of space constraints, a new processor incorporates two times more transistors. In order to reach a 2x speedup, the higher transistor budget is used to implement microarchitecture optimizations, e.g. instruction pipelining, out-of-order execution, branch prediction, data prefetching, and multithreading. According to an empirical rule discovered by Pollack [4], the performance resulting from microarchitecture optimizations is the square root of the density improvement, i.e. for 2x more density, the benefit is 1.4x.

In the recent years, because of the extremely small size of transistors, there have been diminished returns from Dennard's rule, i.e. an improvement of 1.4x from frequency scaling cannot be achieved anymore. The main cause is the leakage current whose influence on power consumption was insignificant for many years. However, by aggressively shrinking the area of the transistor, the negative effect of the leakage current is amplified significantly, thus leading to more wasted static power which can no longer be neglected compared to the active power of the processor. This problem is named the power wall. Besides this, microarchitecture optimizations targeting Instruction Level Parallelism (ILP) have also reached their limits. Major improvements are more difficult to obtain and are not expected to be close to the 1.4x speedup predicted by Pollack's rule. The ILP limitation is commonly called the ILP wall.

The power wall and the ILP wall do not mean that doubling the density of transistors in a processor is not feasible anymore. In fact, this trend still continues. However, the power wall does not allow for the frequency to increase at the same rate as before, i.e. 1.4x, between consecutive generations of processors. Nevertheless, a new processor contains more transistors, e.g. 2x, compared to one from the previous generation. Hence, the main problem refers to determining the optimal allocation of the transistor budget. In the presence of the ILP wall, the low benefit of microarchitecture optimizations does not justify the high cost in terms of transistors. Improved theoretical performance can still be obtained by placing more execution engines, the so-called cores, in a processor. The resulting multi-core design has serious implications on software development, i.e. applications have to be parallelized in order to run efficiently on a multi-core processor [5]. The advantage is that by doubling the number of cores per processor every 2 years, a maximum speedup of 2x is possible without increasing the frequency.

A notable characteristic of multi-core processors is that they can be more power efficient than single-core processors. Consider the scenario in which the same number of transistors can

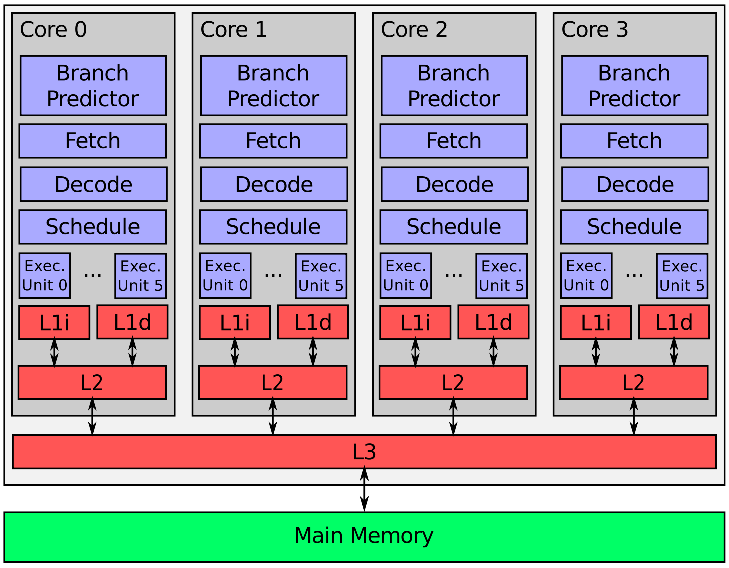


Figure 2.1: Simplified view on the architecture of a 4-core Nehalem CPU. The cache hierarchy has 3 levels: L1 (64 KB), L2 (256 KB), and L3 (8 MB). The branch predictor provides speculative execution. The out-of-order engine includes the schedule unit and 6 different execution units.

be used to build a single-core processor operating at a frequency f or an n-core processor ($n > 1$) at a frequency f/n obtained by reducing V by the right amount. The theoretical performance is the same in both cases. According to Eq. 2.1, the active power decreases more rapidly than the frequency depending on the voltage, meaning that a single-core processor is more power hungry than n cores. Nevertheless, n cores demand parallelism. Furthermore, because of the limited transistor budget per core, the cores in an n-core processor cannot implement the complex functionality of a single-core processor that allows it to efficiently execute serial programs.

In the rest of this thesis, CPUs refer to general purpose multi-core processors that implement the x86 Instruction Set Architecture (ISA). The simplified architecture of an Intel Nehalem quad-core CPU is provided in Fig. 2.1. CPUs are typically optimized with respect to latency [1], meaning that they contain a complex multi-level hierarchy of caches and advanced logic used for automatically extracting the parallelism from a sequential stream of instructions. Moreover, CPUs incorporate speculative techniques for branch prediction and data prefetching. In general, the number of cores per CPU is 1 - 4 while their frequency is 2 - 3 GHz.

2.2 Accelerators

This section provides an overview of the so-called general purpose accelerators, describes the architecture of Nvidia GPUs, and discusses important trends related to CPUs and GPUs.

2.2.1 General View

In contrast to CPUs, accelerators are special purpose processors. The motivation behind their emergence is that significant improvements in both execution time and power consumption can be achieved by customizing the hardware for specific applications. Special purpose accelerators already exist for: Extensible Markup Language (XML) processing, network processing, encryption, and video decoding. In general, accelerators do not run an operating system (OS) and accordingly are complementary to multi-core CPUs. Therefore, accelerator based computing requires the presence of a CPU whose responsibility is to offload tasks to the accelerator.

The accelerators of interest in this thesis are those that can be used to improve the performance of a wider range of programs, not only one. Examples of such accelerators include: Cell Broadband Engine (Cell BE) [6], Graphics Processing Units (GPUs) [7] produced by Nvidia and AMD, Intel Many Integrated Core Architecture (Intel MIC) [8], and Field Programmable Gate Arrays (FPGA) [9]. For a comparison of different accelerators, the reader is referred to [10]. Although these accelerators are typically presented by their manufacturers as general purpose, they can only be used efficiently by applications that match the strong points of accelerators and are less sensitive to their weak points.

A system that contains general purpose CPUs and accelerators is called a heterogeneous system. In order to understand the strengths and the weaknesses of accelerators, it is important to first look at the allocation of the transistor budget in accelerators. Instead of investing the transistors for implementing complex ILP techniques as in the case of CPUs, a strategy characteristic to accelerators is to use the transistors to create a rather large number of cores. The larger the number of cores, the simpler the cores. The so-called wimpy cores of an accelerator are in-order and do not incorporate speculative techniques. The number of cores of an accelerator often considerably exceeds the number of cores of a CPU.

Another characteristic common to most accelerators is that they place emphasis on data parallelism. Accordingly, they contain SIMD units that are wider than the short vector units (4 / 8 32-bit lanes for SSE / AVX) of CPUs. In order to compensate for the lack of out-of-order execution, some accelerators hide the latency of the instruction pipeline through interleaved multithreading which allows for a fast context switch at every clock cycle between different threads. The fast context switch is based on storing / restoring the state of each thread to / from registers. In order to support the concurrent execution of a large number of threads per core, accelerators contain proportionally large register files.

Regarding the memory access, accelerators have in general their dedicated memory which offers a higher bandwidth than the memory of CPUs. In contrast to CPUs which contain deep cache hierarchies, e.g. 2 levels or more, some accelerators replace the cache with an explicitly

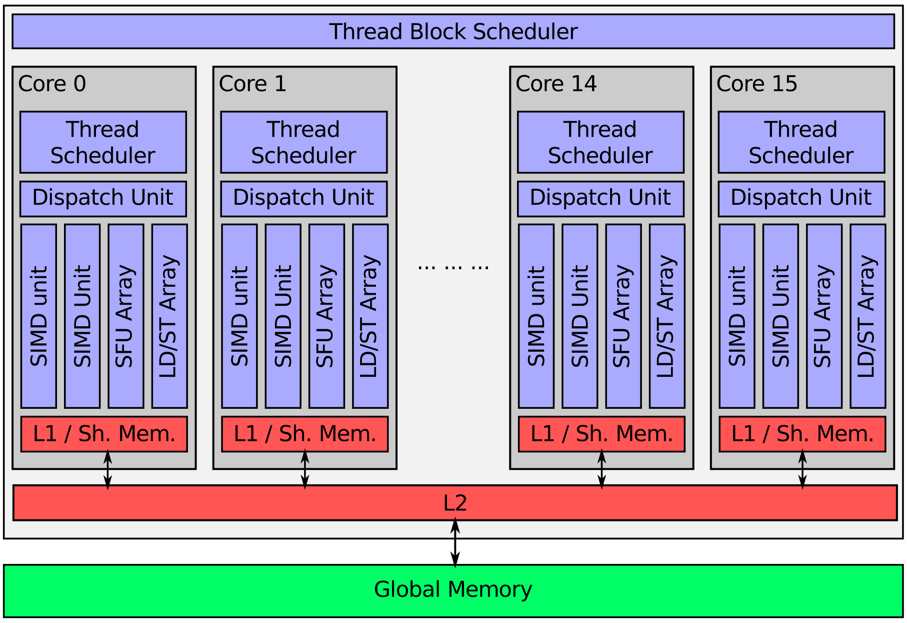


Figure 2.2: Simplified architecture of a 16-core Nvidia Fermi GPU. The GPU contains a fully coherent 2-level cache hierarchy: L1 (16 or 48 KB) and L2 (768 KB). The SIMD units contain 16 lanes each. The thread block scheduler assigns thread blocks, e.g. groups of 256 threads, to cores. The thread scheduler interleaves the execution of thread warps, i.e. groups of 32 threads.

controlled fast memory or scratchpad. Accelerators are often attached to a system via the PCI Express (PCIe) bus whose bandwidth is significantly lower than the bandwidth of the memory on accelerators, e.g. PCIe 2.0 provides a theoretical 8 GB/s unidirectional bandwidth while the memory bandwidth of the accelerator can be as high as 200 GB/s.

The large number of cores, the wide SIMD units, and multithreading are characteristics that make accelerators high throughput processors as opposed to multi-core CPUs which are optimized for low latency processing [1]. The high throughput nature of accelerators means that they require an abundance of parallelism in order to reach high performance.

2.2.2 A Closer Look at GPUs

GPUs are the most common type of accelerators nowadays. They are massively parallel processors capable of both graphics processing and general purpose computing. GPUs dedicate most of their transistors to increasing the number of cores and the width of their vector units. In order to allow for 16 to 30 cores, the GPU cores are simpler than the CPU cores. The following discussion concentrates on the Fermi generation of Nvidia GPUs [11].

The GPU shown in Fig. 2.2 contains up to 16 in-order cores. Each core can issue two instructions per cycle, thus feeding two 16-lanes (32 bits per lane) SIMD units. In reality, an instruction uses vectors of 32 × 32-bit values. The execution for the 32 values needs 2 cycles to complete. In order to cope with the latency of memory operations and stalls in the instruction pipeline, a core incorporates a rather large register file containing 32768 × 32-bit registers capable of storing the context or state, i.e. instruction pointer and private variables, for up to 1536 threads. The overhead resulting from switching between threads can be neglected. Each core also contains 4 transcendental units, called Special Function Units (SFUs), which compute the following functions: *sin*, *cosine*, *reciprocal*, and *square root*. The SFUs are decoupled from the SIMD units meaning that while the SFUs are busy, instructions can still be dispatched to the available execution units. A GPU core also incorporates 16 load / store units.

The hierarchy of memories on the GPU has evolved considerably from pre-Fermi GPUs to Fermi GPUs. A two level fully coherent cache is included in Fermi. Moreover, each core contains 64 KB of fast memory on-chip which can be configured in two ways: (1) in a 16 / 48 KB configuration with 16 KB allocated to a scratchpad memory and 48 KB to the L1 cache, and (2) in a 48 / 16 KB configuration for scratchpad / L1 cache. The scratchpad is memory explicitly managed by programmers. The 64 KB per core are partitioned in 32 banks so that consecutive 32-bit words correspond to successive banks. The maximum throughput is 32 × 32-bit values. This throughput can be obtained when the values used by a vector operation are distributed uniformly among the 32 banks. The L2 cache shared by all the cores has a size of 768 KB. The slowest memory on the GPU is called global memory and has a size of up to 6 GB. A typical memory interface has a width of 384 bits and a frequency of 3.7 GHz, thus resulting in a bandwidth of approximately 180 GB/s.

The GPU is connected to the system via a PCIe bus. A 16-lane PCIe 2.0 provides a theoretical bandwidth of 8 GB/s per direction. Transferring the data to and from the global memory of the GPU is done using Direct Memory Access (DMA).

2.2.3 Trends

Recently, there has been a visible trend for CPUs and accelerators to borrow characteristics from each other. The following discussion is built around Intel CPUs and Nvidia GPUs, notable representatives of CPUs and accelerators respectively.

First, GPUs typically have wider SIMD units than CPUs. Each core of pre-Fermi GPU is equipped with an 8-lane (32 bits per lane) SIMD unit while in the more recent Fermi GPUs, a core contains two SIMD units with a width of 16 lanes. For many years, Intel CPUs have included an extension of the instruction set called Streaming SIMD Extensions (SSE) which allows for the use of a 4-lane SIMD unit. Recent processors from Intel such as Sandy Bride contain vector units with 8 lanes exposed to software through a vector instruction set called

Advanced Vector Extensions (AVX). One can observe that the width of the SIMD unit of current CPUs was upgraded to the width of the vector units found in pre-Fermi GPUs.

Second, on pre-Fermi GPUs, there were only read-only caches, i.e. the constant cache optimized for latency and the texture cache optimized for bandwidth. In contrast, the caches of CPUs allow for both read and write operations and are organized in a deep hierarchical structure with 2 - 3 levels where each level provides different trade-offs between latency, bandwidth, and size. Starting with the Fermi generation, GPUs also contain a two level cache hierarchy that supports both load and store operations and is fully coherent.

The third point is Intel's MIC [8] which is an x86 based accelerator. Hence, it can be regarded as a combination between a CPU and a GPU. Similarly to GPUs, MIC is a high throughput processor and contains more than 50 cores. Emphasis is placed on data parallelism by equipping each core with a 16-lane vector unit. The cores are in-order with support for 4-way multithreading. In contrast, Fermi GPUs support 48-way multithreading. The memory hierarchy includes a 2-level fully coherent cache. Similarly to GPUs, a MIC is accessed from the CPU via PCIe.

2.3 Programming Models

This section compares OpenMP and CUDA, the two main programming models used for multi-core CPUs and GPUs respectively. The section also contains a presentation of OpenCL, a programming model that addresses both CPUs and GPUs.

2.3.1 The OpenMP Programming Model

In the multi-core era, emphasis is placed on Task Level Parallelism (TLP). In contrast to ILP which is typically handled by the compiler and the CPU, TLP is the responsibility of programmers, i.e. developers parallelize their applications using threads which provide a useful abstraction of concurrent execution. A thread is defined as the smallest sequence of instructions that can be scheduled by the operating system (OS) on a core of a CPU. Threads coexist within a process, i.e. an instance of a program in execution, and share its resources, e.g. they all use the same code and address space. However, each thread has a private program counter and stack.

A typical challenge in the case of thread based parallel programming is given by race conditions. In the presence of a race condition, the result of a threaded application is nondeterministic. This problem is addressed by synchronization methods, e.g. locks and semaphores, which help to avoid the corruption of the shared state through mutual exclusion. Incorrect usage of synchronization can lead to deadlocks in which two threads each request access to a resource blocked by the other, thus both of them wait indefinitely. Another common synchronization method on multi-core CPUs is a barrier used to ensure that no thread advances until all the threads reach a common point in the program marked by the barrier.

On multi-core CPUs, threads can be used to implement a variety of parallel computation models [12]: the data parallel model (tasks execute the same operations on different chunks of data), the task graph model (dependent tasks are used to built a dependency graph which is scheduled on worker threads), the work pool model (tasks are placed in a pool from where they are grabbed by worker threads), the master-slave model (a boss thread distributes tasks to worker threads), and the pipeline model (threads execute different functions and are arranged in a sequence that defines the direction of the data flow between threads). Sometimes, complex parallel applications require the use of more than one model, resulting in hybrid models.

OpenMP [13] is an established thread based programing model for shared memory machines, including multi-core CPUs. It provides compiler directives, i.e. pragmas, runtime functions, and environment variables, which allow a programmer to explicitly parallelize an application.

OpenMP is based on a fork-join model in which a master thread forks in a team of worker threads at points in a program explicitly specified by the programmer. Such a point corresponds to the beginning of a parallel region. A parallel region also has an end, i.e. the point where the threads join. In general, threads are scheduled for execution on different CPU cores and their number matches the number of cores or hardware threads.

A subset of the most important functions provided by OpenMP through pragmas is:

- creating a team of threads: *#pragma omp parallel*
- distributing the iteration space of a loop among threads: *#pragma omp for*
- synchronizing threads: *#pragma omp critical* (mutual exclusion), *#pragma omp atomic* (atomic operation), and *#pragma omp barrier* (barrier)
- specifying data attributes: *shared* (accessed by all the threads in a team) and *private* (accessed by only one thread).

The OpenMP 3.0 standard defines a task construct (*#pragma omp task*) which provides extra flexibility to the programming model by allowing it to be employed for parallelizing complex applications that contain pointer based data structures, e.g. trees, or recursive functions. In such scenarios, the previous work distribution methods are impractical. At runtime, a task is executed by the thread that encounters it or is assigned to another thread for execution.

The library part of OpenMP provides useful routines that complement the functionality of pragmas, e.g. *omp_get_thread_num* returns the identifier of a thread in a team and *omp_get_num_threads* returns the size of a team. Flexible lock based synchronization is also possible using a comprehensive set of OpenMP functions. Moreover, the environment variables defined by OpenMP are used to control different runtime parameters, e.g. the scheduling scheme of parallel for loops and the number of worker threads used in parallel regions.

Regarding memory consistency, OpenMP implements a relaxed consistency model in which, for efficiency reasons, threads are not required to maintain the exact view of shared data all the time. Instead of that, they can cache their data, thus resulting in less communication between threads. In order to enforce consistency, the shared variables have to be explicitly flushed.

2.3.2 The CUDA Programming Model

CUDA [14] is a programming model for Nvidia GPUs. It relies on library functions and on a C / C++ extension that allows for a convenient definition of GPU programs. An important aspect of CUDA is that the host (CPU) code is separated from the guest (GPU) code, referred to as a CUDA kernel. CUDA also contains routines for launching kernels to a GPU and managing the data transfers over PCIe, between the CPU and the GPU. CUDA is Single Program Multiple Data (SPMD) programming model in which all the cores, called Streaming Multiprocessors (SMs), execute the same kernel. The CUDA language extension includes keywords that allow to specify the device where a C / C++ function is executed (CPU, GPU, or both), the placement in the memory hierarchy of the variables used in a kernel, and the launch of a kernel.

The typical scenario in CUDA is *task offloading* to the GPU which is done in five steps:

1. The CPU allocates memory space for the input and the output data on the GPU.

2. The CPU copies the input data to the GPU memory.

3. The CPU launches the CUDA kernel.

4. The CPU waits for the GPU to compute the output data.

5. The CPU copies the output data from the GPU memory to the CPU memory.

Hence, the role of the CPU is to manage the data movement and the kernel launch.

CUDA includes a thread based abstraction of the execution on the GPU. This makes it to some extent similar to the parallel programming models for multi-core CPUs, e.g. OpenMP. However, CUDA provides only a limited subset of the functionality of OpenMP, especially for thread synchronization. The methods used to synchronize GPU threads in CUDA are barriers and atomic operations. Moreover, CUDA threads are more numerous than OpenMP threads and are typically more fine granular. Another major difference between the thread based programming models for CPUs and GPUs is that on GPUs, the threads are mainly used to implement a data parallel computation model and it is highly impractical, if not impossible, to obtain a realization of a more complex computation model, e.g. a task graph.

Current GPUs have up to 16 cores and each core can concurrently execute up to 1536 threads. Assuming that all the GPU cores are occupied by threads, if such a large number of concurrent threads communicate with each other, then the scalability is significantly reduced. Therefore, the philosophy in CUDA is to restrict the thread communication patterns for more performance. In order to achieve this, the threads are grouped in blocks which are scheduled each on a GPU core for execution. Each thread block has a unique identifier in a grid of blocks, i.e. *blockIdx*. Furthermore, each thread has an identifier relative to its block, i.e. *threadIdx*. Data sharing and synchronization via barriers are only possible within a block. Data exchange across different blocks can only be achieved in a safe way, i.e. free from race conditions, using atomic operations. In general, GPU programs expose a high degree of parallelism and there is almost no communication between blocks.

The variables used in CUDA kernels belong to one of following memory spaces:

- *global memory*: cached memory that contains the input and output arrays allocated by the CPU; it is persistent across kernel executions
- *local memory*: by default, any thread private variable declared in the kernel is stored in local memory; physically, the local memory is part of the global memory
- *shared memory*: shared variables can be accessed by all the threads in the same block
- *constant memory*: read-only cached memory optimized for low latency; its size is 64 KB
- *texture memory*: read-only cache memory optimized for high bandwidth.

The maximum number of concurrent thread blocks per GPU core is limited by the register consumption per thread and the shared memory consumption per block. Maximizing the *occupancy*, i.e. the number of concurrent (or active) threads per GPU is a common optimization goal for GPU programs. The explanation is given by the fact that if more threads are running on a core, then the positive impact of multithreading is more significant, thus improving the ability to reduce pipeline stalls by interleaving the execution of more threads.

Each GPU core contains two 16-lane SIMD units but in CUDA, these units are not addressed explicitly. In reality, a thread maps to an SIMD lane which can be seen as a Scalar Processor (SP). An SP contains a dispatch port through which it receives execution requests from an instruction dispatch unit, and two Arithmetic Logic Units (ALUs) for floating point operations and integer operations. More precisely, although CUDA groups the threads in blocks, at hardware level the threads in a block are executed in groups of 32 threads, called warps. A warp thus denotes 32 threads that all perform the same operation on 32 chunks of data. In CUDA, this execution model is referred to as Single Instruction Multiple Threads (SIMT).

Understanding the warp concept is essential for efficiently using the GPU. If the threads in a warp follow different control paths because of branches, then the performance of a CUDA kernel can be severely reduced, e.g. up to a factor of 32. Such a warp is called divergent warp. In this case, the SIMD unit is used inefficiently because the execution of the warp is serialized so that only the threads that execute the same operation can utilize the SIMD unit simultaneously. For instance, if 16 threads of a warp execute an operation while the other 16 execute another one, the time necessary for the SIMD unit to handle the entire warp is twice the time corresponding to the scenario in which all the threads execute the same operation.

Warps are also important with regard to the different memory types of the GPU. The shared memory (or scratchpad) on every core is divided into 32 banks. In order to use this memory efficiently, the threads in a warp are required to access different banks so that the accesses are uniformly distributed among banks. Another optimal scenario is when all the threads in a warp access the same value, meaning that it is broadcast to all the threads. If two or more threads access different addresses in the same bank, then the accesses are serialized, thus reducing the performance. With respect to global memory, it is used efficiently if all the threads in a warp access a contiguous and aligned memory region. This reduces the number of transactions to global memory and allows for a higher percentage of the data transferred from global memory to be actually used by threads as part of the computation.

2.3.3 The OpenCL Programming Model

OpenCL [15] is a framework for programming heterogeneous systems such as those containing CPUs and GPUs. OpenCL is developed by the Khronos standardization consortium and is adopted by Nvidia, AMD, Intel, and ARM. The main motivation behind OpenCL is the need for a language that can be used to program a wide variety of processors. In other words, OpenCL can be used to program both CPUs and GPUs.

OpenCL is both an Applications Programming Interface (API) and a language for expressing parallel cross-platform applications. The language is strongly influenced by CUDA. In essence, an OpenCL program describes the operations performed by one SIMD lane. However, in contrast to CUDA, which focuses on data parallelism, OpenCL also allows for task parallelism. Moreover, it can be used to program both fine-grained parallelism as in CUDA and more coarse parallelism like in OpenMP. Many of the OpenCL concepts map to CUDA concepts. A CUDA thread becomes a work item in OpenCL while a thread block is a work group. Data sharing and synchronization is only allowed between work items within the same work group. A CUDA kernel is called program in OpenCL which is executed by all the work items in an SPMD fashion. Again, a separation is made between host and devices. The devices execute OpenCL programs while the responsibility of the host is to manage the interaction with the devices, i.e. sending commands for data transfers to and from devices, and launching programs on devices.

OpenCL describes an abstract view of devices that hardware vendors have to implement in order to expose their processors through OpenCL. A device contains compute units and a compute unit consists of a set of processing elements. In Nvidia terminology, a compute unit is a Streaming Multiprocessor and a processing element is a lane of an SIMD unit, called Scalar Processor (SP) in CUDA. On multi-core CPUs, a compute unit is a core while a processing element is a lane of the SSE (or AVX) unit. The memory hierarchy specified in OpenCL includes the following levels: global memory and constant memory (physically placed off-chip) are available to all the work items, local memory (fast on-chip scratchpad memory) allows for the data to be shared among work items in the same work group, private memory (typically maps to register file) which contains the private variables per work item.

The OpenCL API is the interface that allows for discovering OpenCL devices and compiling code for them at runtime. At API level, concepts such as platform, context, and work queues are defined in order to properly manage a wide range of processors from different manufacturers: A platform groups all the devices from the same manufacturer (Intel, Nvidia, or AMD), a context is a handler for managing devices belonging to the same platform, whereas a work queue is associated with a device and is used by the host to submit OpenCL commands to the device.

Regarding vectorization, which is an important topic nowadays because of the emergence of vector processors, OpenCL offers two solutions referred to as explicit and implicit vectorization. In explicit vectorization, vector types enable a programmer to directly expose vector operations to the compiler. For CPUs, the implicit vectorization approach is based on compiler optimizations. On GPUs, implicit vectorization is achieved automatically in hardware as part of the SIMT execution model in which a warp is merged and split depending on branches so that an operation shared by a subset of the threads in a warp is executed in parallel.

2.4 Examples of GPU Applications

There are many applications that have already been ported to CUDA and validate the benefits of GPUs over CPUs for certain computations. Some of these applications are briefly described next. The chosen applications are only a subset of the existing GPU programs and were chosen here because of their high popularity. The applications are presented in the increasing order of their computational intensity, starting with applications that are computationally bound, i.e. high intensity, and finishing with memory bound applications, i.e. low intensity.

Matrix multiplication is a dense linear algebra operation that benefits significantly from the computational power of GPUs. The algorithm discussed here is the naïve matrix multiplication characterized by $\mathcal{O}(n^3)$ complexity where n is the number of rows and columns. Matrix multiplication is computationally bound meaning that its performance is influenced more by the GFlop/s rate rather than the memory bandwidth. Matrix multiplication is at the core of benchmarks such as LINPACK used to evaluate the fastest supercomputers in the world [16]. An efficient CUDA implementation of matrix multiplication is described in [17]. The optimized CUDA version of matrix multiplication proposed there reaches 60% of the peak performance of Nvidia GPUs (for single precision floating point numbers) and is included in CUBLAS 2.0, i.e. the library from Nvidia that implements the Basic Linear Algebra Subprograms (BLAS) interface. Compared to the GFlop/s rate obtained on an Intel Core2 Quad Q6850 operating at 3 GHz, the performance measured using a GTX280 GPU is up to 4.4x better.

N-body simulations are intensively used in astrophysics for simulating the formation of galaxies and in molecular dynamics for simulating the interaction between molecules. This is another example of a program that benefits from the high GFlop/s rate of GPUs. Efficient implementations for GPUs include [18, 19]. According to [18], an optimized CUDA version is 50x faster than a highly tuned serial CPU implementation. An important factor for obtaining this speedup is the set of transcendental functions hardwired in the GPU cores which allow for a fast computation of reciprocal and square root functions. In [19], the authors' implementation reaches a 100x speedup compared to the performance obtained on a single-core CPU.

The Fast Fourier Transformation (FFT) is used for solving Partial Differential Equations (PDEs), for image processing, and for digital signal processing. A 1-dimensional FFT runs in $\mathcal{O}(n \cdot \log(n))$ time where n is the number of points for which the Discrete Fourier Transform (DFT) is computed. CUDA implementations of FFT are detailed in [20, 21]. In [20], the authors show that depending on the input, a GTX280 GPU is between 2.7x and 16x faster than a quad-core AMD Phenom 9500 CPU. In [21], a GTX280 delivers a performance that is in the range from 8x and 40x better than an Intel Core2 Extreme QX9650, i.e. a quad-core processor.

Stencil computations typically result from the discretization of Partial Differential Equations (PDEs) using the Finite Difference Method (FDM). Stencils are also found in image processing applications. A stencil computation is based on a pattern, i.e. the stencil, used to define the neighbors of every cell in a discretized domain. All the values of the cells are updated according to an equation involving the values of the neighbor cells. Sometimes, the update of all the cells is part of an iterative process, i.e. the update is done multiple times. Most stencil computations are memory bound, meaning that they can make use of the high memory bandwidth of GPUs. Efficient implementations for GPUs are described in [22, 23]. In [22], a speedup of almost 7x is obtained on a GTX280 compared to a dual-socket system containing one AMD Barcelona 2356 quad-core CPU per socket. [23] describes a CUDA implementation that is roughly 10x faster on a Tesla 10-series GPU than the CPU implementation measured on a quad-core Intel Harpertown CPU.

Another application accelerated using GPUs is sparse matrix - vector (SpMV) multiplication, an operation commonly found in iterative methods for solving PDEs. SpMV is characterized by low computational intensity and irregular access to memory. Therefore, the GFlop/s rate of SpMV is highly influenced by memory performance. Efficient SpMV implementations for GPUs include [24, 25]. In [24], the authors' implementation delivers up to 16 GFlop/s on a GTX285 which is more than 10x the performance obtained on a quad-core Intel Clovertown system. The implementation described in [25] improves that GPU performance by up to 1.8x.

The different hardware setups used for obtaining the results presented above does not allow for a thorough analysis of the concrete benefits of GPUs compared to CPUs. Nevertheless, there are several observations that can be drawn. Computationally bound applications, e.g. matrix multiplication and n-body simulations, can reach speedups of almost 2 factors of magnitude on GPUs compared to a single-core CPU. At the opposite end, for memory bound applications, e.g. stencil computations and SpMV, the expected speedup of GPUs relative to single-core CPUs is roughly one factor of magnitude, i.e. based on the assumption that the GPU memory bandwidth is approximately 200 GB/s whereas the CPU memory has a bandwidth of 20 GB/s. For more complex applications, e.g. that are control bound and cannot use vector units efficiently, it is difficult to formulate realistic expectations from GPUs in terms of performance.

2.5 Challenges

GPU benefits do not come for free. There are many challenges that have to be addressed before an application obtains substantial performance on a GPU accelerated heterogeneous system.

Limited Amount of GPU Memory Nowadays, the bandwidth of the GPU memory is significantly higher than the bandwidth of the CPU memory, e.g. 10x. However, the disadvantage is that the GPU memory has a rather small size, e.g. 6 GB. This is an important limitation for applications that cannot fit their input and output data in the GPU memory.

Slow Data Transfers over PCIe The bandwidth of the PCIe bus, e.g. 8 GB/s, is rather low compared to the GPU memory bandwidth, e.g. 200 GB/s, and the CPU memory bandwidth, e.g. 20 GB/s. PCIe acts as a severe bottleneck especially for programs that are characterized by a low GPU computation to PCIe communication ratio. GPU computing requires enough computational work so that the PCIe transfer time is dominated by the GPU processing time.

Control Bound Codes Control bound codes may cause the GPU threads in a warp to follow different control paths. In the worst case scenario, control bound applications are affected by a slowdown of 32x assuming that all the 32 threads in a warp execute different operations.

Irregular Memory Access Patterns The optimal use of both global memory and shared memory is achieved when all the threads in a warp access a contiguous region of memory. At shared memory level, for maximum efficiency, the accesses generated by a warp must be uniformly distributed among the banks of the shared memory. At global memory level, the threads must access a 128 Byte segment aligned to a 128 Byte address boundary. In the presence of irregular access patterns, it is difficult to meet such requirements. Irregular accesses tend to generate bank conflicts in the shared memory and excessive traffic from the global memory to the cache, i.e. only a fraction of every cache line is actually used in the computation.

Incompatibility with Recursion Recursion is highly incompatible with GPUs because of the high number of concurrent threads, e.g. tens of thousands, that run at any time [26]. Providing stack space for all the threads would considerably increase the memory consumption. Furthermore, overheads associated with recursion, e.g. saving and restoring registers, copying parameter values on the stack, would significantly reduce the performance.

Incompatibility with Dynamic and Irregular Parallelism Dynamic parallelism in which tasks are created during execution is not supported by GPUs. Irregular parallelism is often affected by load imbalances, leading to undesired scenarios in which some threads in a warp have more work to do than others (the SIMD unit is used inefficiently), some warps in a thread are assigned longer tasks than others (multithreading is suboptimal), or some GPU cores are busy while others are idle. Current GPUs are not adaptive enough to cope with such problems.

Complex Interactions between Performance Parameters GPU applications often expose optimization parameters that allow for a fine control of the performance behavior. In general, the interactions between the parameters are complex and cannot be tackled using theoretical means, e.g. performance models. A typical example for GPUs is the trade-off between register consumption and concurrency, i.e. allocating more registers per thread results in an increase of the serial performance of a thread and a decrease of the number of threads running concurrently on each core, thus reducing the positive effect of multithreading.

Suboptimal Utilization of a Heterogeneous System Full utilization of a heterogeneous system implies using the CPU and the GPU simultaneously. The typical interaction between the CPU and the GPU in CUDA follows a task offloading model. However, this model focuses only on keeping the GPU busy, i.e. while the GPU computes, the CPU is idle. Load balancing aims at engaging both the CPU and the GPU in the computation in order to increase the performance. In practice, this can be achieved for instance by combining CUDA and OpenMP.

All these challenges are addressed for a set of performance critical routines extracted from a computational steering application that is based on the sparse grid technique [27]. The routines are characterized by different processing behaviors: integer bound, computationally bound, and memory bound. The thesis presents the porting of the sparse grid routines to GPU based heterogeneous systems, including topics such as empirical optimizations and load balancing.

2.6 Summary

While CPUs are latency oriented processors, GPUs are throughput oriented processors containing a large number of cores with wide SIMD units. GPUs belong to the class of many-core processors or accelerators. The main reason behind the emergence of GPUs is the power wall problem defined as the difficulty of increasing the frequency of processors while keeping the power consumption within acceptable limits. Two solutions are incorporated in GPUs in order to cope with the power wall problem: First, their transistor budget is mainly allocated to floating point units, and second, they are typically clocked at lower frequencies than CPUs.

In terms of peak performance, GPUs are factors of magnitude faster than CPUs. However, the theoretical speedup is rarely transferred to applications. In order to fit the strengths of GPUs, programs are expected to meet the following set of requirements: The type of parallelism characteristic to the application is data parallelism, the code is vector friendly, the memory consumption is low enough to fit the small amount of memory on the GPU (up to 6 GB), the code is free from non-recursive functions which are either not supported or execute inefficiently, and there is enough computational work so that the data transfers over PCIe are worthwhile.

Chapter 3

Computational Steering Using Sparse Grids

This chapter describes the computational steering application used as a benchmark in the next chapters. The computational steering approach is based here on the sparse grid technique, a numerical technique particularly useful when dealing with high-dimensional problems. The theory behind this technique is described together with its role in computational steering. For implementing the sparse grid technique, different data structure and algorithm variants can be employed. They are presented in this chapter in the context of heterogeneous systems. A data structure is proposed that has a minimum memory footprint and is GPU friendly. At the end of the chapter, the core components of a sparse grid benchmark are defined: sparse grid hierarchization, sparse grid interpolation, and sparse vector - matrix multiplication.

3.1 A Computational Steering Scenario

In order to get a better understanding of complex phenomena simulated on supercomputers, it is often necessary to apply slight variations to some parameters of the simulation which control for instance the initial conditions, the boundary conditions or even the geometries of the given problem. In a typical computational steering scenario, a user can update these parameters at any time and his action either changes the simulation on-the-fly or triggers a new simulation. This approach however is not always practical and may imply high costs as it requires a permanent connection with a supercomputer, at least for the duration of the computational steering process. Furthermore, depending on the simulation, the results may or may not be delivered to the user in real-time for visualization. In order to address these challenges, in this thesis, a different approach to computational steering is described. Instead of running a simulation

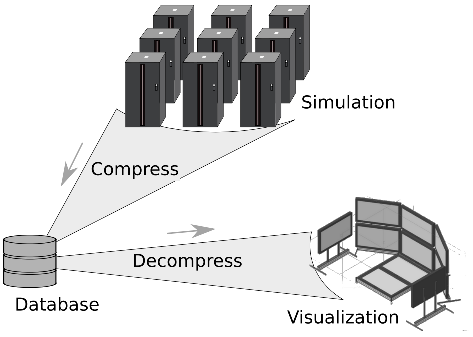


Figure 3.1: Main computational steering scenario. Compressed simulation data is stored in a database. Data is then decompressed in real-time for visualization.

for every parameter update, simulation results are obtained by interpolating precomputed simulation data stored in a database. The process is depicted in Fig. 3.1. What this actually accomplishes is a shift of weight between simulation time and storage space, meaning that data management techniques, e.g. *sampling*, *compression*, and *decompression*, gain more importance. As computational steering usually goes hand in hand with visualization, decompression has to be done as fast as possible in order to allow for a smooth interaction with the simulation data.

The chosen computational steering approach is not free of challenge. The parameters of the simulation often form a high-dimensional space. At database level, a combination of parameters identifies a simulation snapshot which is defined as the data returned by the simulation for the given parameters. It is in most cases not feasible both with respect to space and time to obtain a snapshot for too many combinations of parameters. Therefore, a *sampling* method must be used which reduces the number of snapshots stored in the database. Moreover, the parameters of the simulation together with the spatial dimensions (usually 3), result in a high-dimensional space, typically with up to 10 dimensions, which is a candidate for *data compression.* As an example, [28] describes the steering of the lid-driven cavity simulation (a fluid is trapped in a cavity whose upper wall moves horizontally). There, the physical domain is 3-dimensional (3d) while the Reynold's number and the time are the parameters of the simulation, thus resulting in a 5-dimensional (5d) problem.

A numerical technique that provides the functionality required by computational steering is the sparse grid technique which is used for solving high-dimensional problems arising in astrophysics, molecular dynamics, finance, and data mining [2, 29]. Its applicability to computational steering is presented in [28]. Sparse grids are employed for the numerical representation and treatment of high-dimensional functions which typically suffer from the so-called curse of dimensionality, the exponential dependency of the number of discretization points on the number of dimensions. This problem can be clearly seen in the case of the d-linear interpolation of a d-dimensional function. In a full grid discretization in which N points are allocated per coordinate direction, the total number of points spent for the entire d-dimensional domain is N^d. Hence, it is impossible to build a reasonable discretization in settings with more than 4 dimensions such as computational steering. In contrast to full grids, sparse grids need only $\mathcal{O}(N \cdot (\log(N))^{d-1})$ points. More importantly, the accuracy is only slightly deteriorated for sufficiently smooth functions [2]. Therefore, it can be said that the sparse grid technique offers an advantageous cost-benefit ratio, thus making it well suited as a lossy compression scheme.

The two central algorithms of the sparse grid technique are *hierarchization* and *interpolation*. If one sees the sparse grid technique as a compression scheme, data compression is achieved in two steps: First, a special discretization (sampling) of the domain is applied and second, hierarchization represents the discretized data using hierarchical basis functions. Sparse grid interpolation can be seen as decompression. In special circumstances of interest described later in this chapter, interpolation can be reduced to a linear algebra operation referred to as sparse vector - matrix multiplication.

3.2 Related Work

The use of sparse grids to efficiently handle high-dimensional data in computational steering is described in [30, 28, 31]. There, the authors study the applicability of sparse grids to certain Computational Fluid Dynamics (CFD) simulations, including also error analysis. In [28, 31], the authors store vectors in a sparse grid. In their approach, the dimensions of sparse grids are the non-spatial dimensions of the simulation whereas a vector is a 3d simulation snapshot.

Dimensionally truncated sparse grids are important in computational steering as they allow for the parameters of a simulation to be treated differently i.e. more or fewer discretization points per dimension, depending on their importance in the sparse grid approximation. The concept of dimensional truncation is derived from dimensional adaptivity described in [32, 33, 34] and implemented in *spinterp*, a Matlab framework for sparse grid interpolation [35, 36].

Reducing the memory footprint of sparse grids without sacrificing access time is an important objective in this chapter. The typical solutions for storing a sparse grid containing multi-dimensional points and associated values, are key-value data structures such as *hash-tables* and *trees* [37]. Their use for sparse grids is discussed in [38, 39, 40]. In general, hash-table approaches are preferred because of their lower memory requirements compared to pointer based approaches like trees. Among the data structures used for representing sparse grids in memory, a special class is that of data structures based on a bijective mapping (or perfect hash function) described in [41, 42, 43, 44]. Using the bijective mapping, each grid is given a unique index in a dense

1d array containing only the sparse grid values. Since the bijection does not further increase memory consumption, only the values of the sparse grid need to be stored, meaning that this data structure has *minimal memory consumption.*

Another point addressed in this chapter is accelerating sparse grid interpolation using input adaptation or specialization. Techniques for input adaptation are already at the foundation of *OSKI* [45] and *FFTW* [46]. *OSKI* is a library for sparse linear algebra that automatically selects the best data structure and algorithm given special characteristics of the input matrix, e.g. symmetry, block and diagonal structures. *FFTW* is a library for computing FFT transformations. There, a problem is recursively divided into subproblems using a dynamic programming algorithm. The smallest subproblems are solved by codelets, i.e. input specialized algorithms for computing FFT. Input adaptive approaches also exist for sorting [47, 48] and reduction [49]. Furthermore, general purpose frameworks have been developed that make use of machine learning techniques for mapping input data to multiple code versions [50, 51]. For a more comprehensive list of input adaptive solutions, the reader is referred to [50]. In the context of the sparse grid technique, [52] applies input adaptivity to sparse grid interpolation. Patterns in the set of interpolation points are used to create specialized versions of interpolation characterized by reduced complexity and lower memory requirements.

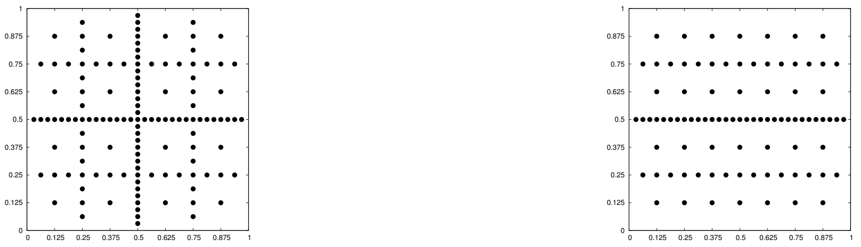
3.3 The Sparse Grid Technique

This section presents the sparse grid specific discretization of a d-dimensional domain and describes the basis functions used for the numerical representation of a generic d-dimensional function. Furthermore, two algorithms are covered: *hierarchization* which computes the coefficients of basis functions and *interpolation* which evaluates the sparse grid at a given point.

3.3.1 The Construction of Sparse Grids

For simplification, consider the case of a generic multi-dimensional function $f : \Omega \to \mathbb{R}$, where $\Omega := [0,1]^d$. An approximation for f is realized by discretizing the definition domain of f and representing f as a sum of weighted (or scaled) basis functions.

Let $\underline{l} := (l_1, \ldots, l_d)$ and $\underline{i} := (i_1, \ldots, i_d)$ be vectors from $\mathbb{N}^d$ and let $\underline{x_{\underline{l},\underline{i}}} := (x_{l_1,i_1}, \ldots, x_{l_d,i_d})$ denote a vector from Ω for which $x_{l_t,i_t} := i_t \cdot 2^{-l_t}$, $i_t \in \{1, \ldots, 2^{l_t} - 1\}$, and i_t odd, $\forall t \in \{1, \ldots, d\}$. Based on this definition, both (l, i) and $x_{l,i}$ are equivalent identifiers of a grid point. The relation between $\underline{l}$ and $\underline{i}$ is preserved for all the discretizations of Ω that are presented next.



(a) Regular sparse grid, 5 refinement levels, 129 points.

(b) Truncated sparse grid, 5 refinement levels, constraint vector (5, 3), 89 points.

Figure 3.2: 2d zero boundary sparse grids.

A first discretization that can be defined on Ω is the full grid, $\Omega_f \subset \Omega$,

$$\Omega_f := \{\underline{x}_{\underline{l},\underline{i}} : |\underline{l}|_\infty \leq n\}, \qquad |\underline{l}|_\infty := \max(l_1, \ldots, l_d), \tag{3.1}$$

where n is the refinement level of the grid. Since the number of points in Ω_f is $\mathcal{O}(2^{nd})$, full grids are impractical when tackling high-dimensional problems.

A significant reduction of the number of discretization points can be achieved through a regular sparse grid, $\Omega_r \subseteq \Omega_f$:

$$\Omega_r := \{\underline{x}_{\underline{l},\underline{i}} : |\underline{l}|_1 \leq n + d - 1\}, \qquad |\underline{l}|_1 := \sum_{t=1}^{d} l_t. \tag{3.2}$$

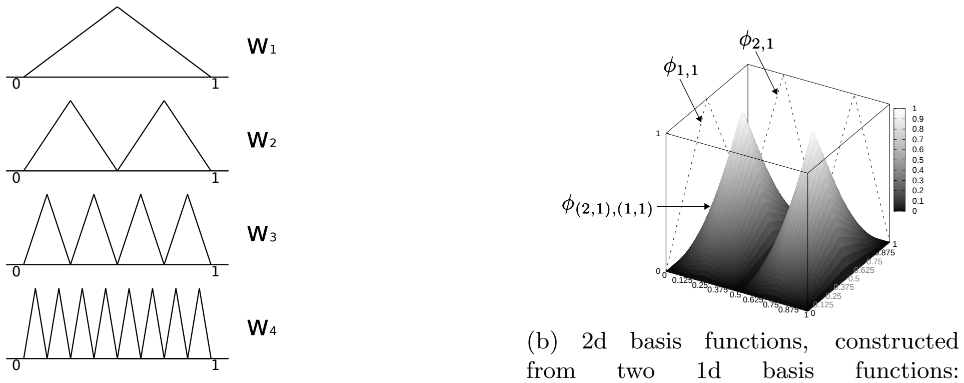
A regular sparse grid is obtained by replacing $|\underline{l}|_\infty \leq n$ in the definition of the full grid with the more restrictive $|\underline{l}|_1 \leq n + d - 1$. The resulting cardinality of Ω_r decreases considerably to $\mathcal{O}(2^n \cdot n^{d-1})$. A 2d regular sparse grid is depicted in Fig. 3.2a.

A dimensionally truncated sparse grid, $\Omega_c \subseteq \Omega_r$, for a given constraint vector $\underline{c}$ (containing an upper limit for all the components of $\underline{\mathrm{l}}$) is defined as:

$$\Omega_c := \{\underline{x}_{\underline{l},\underline{i}} : |\underline{l}|_1 \leq n + d - 1, l_t \leq c_t, t \in \{1, \ldots, d\}\}. \tag{3.3}$$

Ω_c is obtained by filtering Ω_r using $\underline{c}$, a controllable parameter which can further decrease the cardinality of a regular sparse grid. For $c_t = n, \forall 1 \leq t \leq d$, Ω_c becomes Ω_r. Fig. 3.2b is an example of a 2d truncated sparse grid. Here, $\underline{c} = (5, 3)$ reduces the number of points from 129 to 89. For comparison, a 2d full grid with the same refinement level contains 961 points.

Approximations of the function f can be built using any of these grids. Assuming that the



(a) 1d basis functions up to level 4. $V_4 = W_1 \bigoplus \cdots \bigoplus W_4$.

(b) 2d basis functions, constructed from two 1d basis functions: $\phi_{(2,1),(1,1)}(x, y) = \phi_{2,1}(x) \cdot \phi_{1,1}(y)$.

Figure 3.3: Hierarchical basis functions.

discretization is realized through truncated sparse grids, the interpolant $f_c : \Omega_c \to \mathbb{R}$ is

$$f_c := \sum_{\underline{x}_{\underline{l},\underline{i}} \in \Omega_c} \alpha_{\underline{l},\underline{i}} \cdot \phi_{\underline{l},\underline{i}}, \tag{3.4}$$

where $\alpha_{\underline{l},\underline{i}}$ is the weight (or *hierarchical coefficient*) and $\phi_{\underline{l},\underline{i}}$ is the *hierarchical basis function* centered at the grid point $\underline{x}_{\underline{l},\underline{i}}$ stemming from the discretization. $\phi_{\underline{l},\underline{i}}$ is obtained from the one-dimensional function $\phi_{l,i}(x) := \mathrm{h}(2^l x - i)$ by means of a tensor product, where h denotes the standard hat function $\mathrm{h}(x) := \max(1 - |x|, 0)$. Hence, the definition of $\phi_{\underline{l},\underline{i}}$ is

$$\phi_{\underline{l},\underline{i}}(\underline{x}) := \prod_{t=1}^{d} \phi_{l_t,i_t}(x_t). \tag{3.5}$$

The restriction that f is zero on the boundary of Ω is used to simplify the descriptions. Non-zero boundary values are addressed by adding two more basis functions $\phi_{0,0}$ and $\phi_{0,1}$ on level 0.

Fig. 3.3a depicts a hierarchy of 1d basis functions grouped according to l while Fig. 3.3b shows the construction of 2d basis functions from 1d ones. All the basis functions with the same $\underline{l}$ belong to a regular grid, have pairwise disjoint, equally sized supports, and cover the entire domain. They form a basis that spans a hierarchical subspace $W_{\underline{l}}$. The sparse grid space of functions, depicted in Fig. 3.3a for the 1d case, is a sum of hierarchical subspaces:

$$V_c = \bigoplus_{\underline{l}} W_{\underline{l}}, \qquad |\underline{l}|_1 \leq n + d - 1, \quad l_t \leq c_t, t \in \{1, \ldots, d\}. \tag{3.6}$$

Listing 3.1 Hierarchization for 1d sparse grids. This is the building block of the d-dimensional hierarchization algorithm.

```
function hierarchize1d(gp, leftParVal, rightParVal, l, n)
  if l < n then
    hierarchize1d(gp.leftChild, leftParVal, gp.val, l + 1, n)
    hierarchize1d(gp.rightChild, gp.val, rightParVal, l + 1, n)
  gp.val = gp.val - (leftParVal + rightParVal) / 2
```

Computing the α coefficients of the sparse grid approximation is referred to as *hierarchization*. The α coefficients are later used to determine the value of the approximation at any point inside the $[0,1]^d$ domain. This is called *interpolation* of the sparse grid at a given point.

3.3.2 Computing the Hierarchical Coefficients

Sparse grid algorithms for hierarchization and interpolation both have a recursive nature. Listing 3.1 shows the algorithm for hierarchization for 1d sparse grids. The function is called using a grid point *gp* located at the middle of the interval, e.g. 0.5 for the $[0,1]$ domain. For zero boundary sparse grids, the initial values for *leftParVal* and *rightParVal* are 0. Otherwise, the values are taken from the corresponding dependencies or parents on the boundary. From the initial grid point, the algorithm descends recursively to the left and the right child. Consider the notation (ll, li) for the level and the index of the left parent for the grid point identified by (l, i). The relation between a child and its left parent is given by the equation:

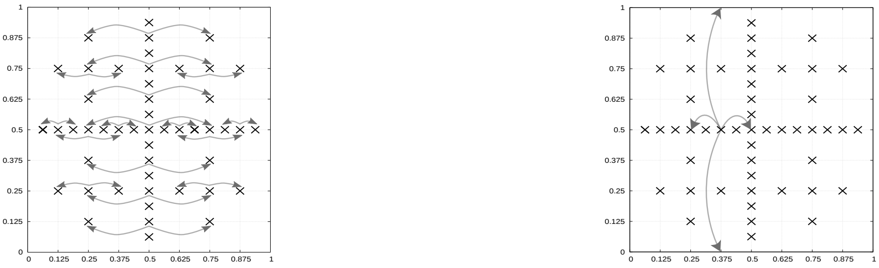
$$li \cdot 2^{-ll} = (i-1) \cdot 2^{-l}. \tag{3.7}$$

Similarly, in the case of the the right parent represented by (rl, ri), the equation becomes:

$$ri \cdot 2^{-rl} = (i+1) \cdot 2^{-l}. \tag{3.8}$$

In the recursive algorithm for hierarchization, the child grid points are updated before the hierarchical parents. The stop condition is given by the refinement level. When the maximum refinement level n is reached, the algorithm stops and returns from recursion. In effect, what this algorithm does is a depth-first traversal of the subspaces depicted in Fig. 3.3a. The final result is a sparse grid containing the hierarchical coefficients α.

Although it addresses the 1d case, the sparse grid algorithm from Listing 3.1 is in fact the building block of d-dimensional hierarchization. For d dimensions, the algorithm loops over each dimension and in each iteration t, only the grid points that satisfy the condition $l_t = 1$



(a) Recursive horizontal traversal of a 2d sparse grid.

(b) Hierarchical parents (or dependencies) of a hierarchical coefficient.

Figure 3.4: Traversal and data dependencies in 2d sparse grid hierarchization.

and $i_t = 1$ are selected and are passed as the *gp* parameter to *hierarchize1d*. Each grid point is identified by a pair $\underline{l}$ and $\underline{i}$. Thus, the depth of the recursion depends on the initial grid point since the recursive 1d hierarchization increases the t-th component of $\underline{l}$, l_t, while satisfying the sparse grid constraints, i.e. there is an upper limit for the L^1-norm of $\underline{l}$ and a constraint vector that puts an upper bound on every component of $\underline{l}$. The transition from 1d to multi-dimensional hierarchization is depicted in Fig. 3.4a for the 2d case. In the figure, t first indicates the horizontal dimension. The grid points on the vertical axis, for which $l_t = 1$ and $i_t = 1$, are selected as the initial grid points passed as parameters when invoking *hierarchize1d*. The arrows show the direction of the recursive traversal of the sparse grid, i.e. parallel to the horizontal axis. After the grid is updated, t is set to the vertical dimension and hierarchization traverses the sparse grid vertically. Fig. 3.4b shows the dependencies for a grid point, emphasizing that the algorithm jumps over grid points in order to reach the data dependencies. This indicates that improving locality is difficult to achieve for sparse grid hierarchization.

3.3.3 Evaluating the Sparse Grid Approximation

The algorithm for 1d sparse grid interpolation is shown in Listing 3.2. In general, sparse grid interpolation takes a point from $[0, 1]^d$ and sums up the scaled (or weighted) basis functions evaluated at the given point. The scaling is done using the hierarchical coefficients stored in the sparse grid. The *interpolate1d* algorithm performs exactly these operations. In line 2, it evaluates the current basis function at the given point and scales the result using a hierarchical coefficient, and in lines 5 and 6 recursively moves to the next basis function.

The algorithm already includes an optimization. One can see in Fig. 3.3a that the basis functions corresponding to the same subspace, e.g. W_4, have non-intersecting or disjoint supports, meaning that at most one basis function for each subspace has a non-zero contribution to the interpolation. This optimization is realized in line 4, i.e. the traversal of the sparse grid

Listing 3.2 Interpolation of a 1d sparse grid at point x.

```
function interpolate1d(x, gp, l, n)
  res = gp.val * basis1d(gp, x)
  if l < n then
    if x < gp.coord then
      res = res + interpolate1d(x, gp.leftChild, l + 1, n)
    else
      res = res + interpolate1d(x, gp.rightChild, l + 1, n)
  return res
```

is steered towards those grid points whose associated hierarhical basis functions are non-zero at the interpolation point (or their support includes the point). This significantly reduces the number of basis function evaluations. As an example, in Fig. 3.3a only 4 basis functions are actually evaluated, independent from the interpolation point.

For the general d-dimensional case, slight modifications have to be made to *interpolate1d*: (1) generate recursively the $\underline{l}$ identifier of a subspace and the grid point *gp* whose associated basis function actually contributes to the interpolation, and (2) replace *basis1d* with the d-dimensional basis function (the product of d values returned by *basis1d* for all the dimensions).

3.4 Traditional Data Structures for Sparse Grids

For implementing sparse grid algorithms, key-value based data structures are typically used. Examples includes hash-tables and tree maps. The key is the coordinate vector of a grid point or the equivalent pair of levels and indices. The main disadvantage in this case is that the key space often occupies more memory than the value space. More precisely, in the context of sparse grids there is an $\mathcal{O}(d)$ ratio between a key and a value in terms of memory use.

An important component of the *hash-table* approach is the hash function. Here, it maps a grid point to an index used to access an array of buckets. Ideally, the hash function must ensure a uniform distribution of the keys to buckets. Moreover, the smaller the number of collisions per bucket, the faster the access to the data stored in the hash-table. More formally, searching in a hash-table has a cost of $\mathcal{O}(1+N/k)$, where N is the number of pairs added to the structure and k is the number of buckets. Increasing the number of buckets often results in a decrease of the number of collisions but this can create empty buckets, resulting in inefficient memory use.

In a standard *tree* based implementation, an ordering relation must be established for the grid points. Then, the access to any value is done in $\mathcal{O}(\log(N))$. A more memory efficient approach is a prefix tree or a *trie* which compresses the key space by storing the prefix shared by multiple grid point coordinates only once. The concept is depicted in Fig. 3.5. The theoretical

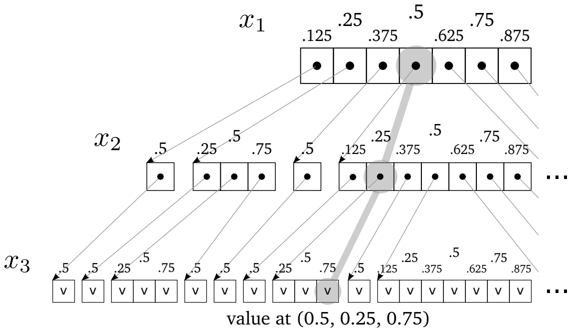


Figure 3.5: The trie data structure for a 3d sparse grid of level 3. The grid points are represented using coordinates. The arrays are linearized binary trees. Each level of the tree corresponds to one dimension. The access to the grid point given by $\underline{l} = (1, 2, 2)$ and $\underline{i} = (1, 1, 3)$ or equivalent coordinates $(0.5, 0.25, 0.75)$ is highlighted.

access time is here $\mathcal{O}(d)$. Hence, it does not depend on N as before.

The trie structure can exploit the sparse grid constraints, $|\underline{l}|_1 \leq n + d - 1$ and $l_t \leq c_t, \forall t \in \{1, \ldots, d\}$, to reduce the memory footprint. The arrays in Fig. 3.5 are linear representations of binary trees and contain pointers, except the leafs which contain the actual hierarchical coefficients. The arrays on level t in the trie correspond to dimension t. Accessing the coefficient associated to a grid point implies a top-down traversal of the arrays of pointers. Within each array, a one-to-one correspondence between a grid point coordinate and a pointer is used to indicate the next array on the path. A pointer borrows the refinement level of its respective grid point coordinate. Moreover, all the pointers on the path to a coefficient are subject to the same constraints as the sparse grid. In other words, the refinement level of a pointer limits the size of all the arrays that belong to the subtree starting from it. Consequently, the size of the arrays varies from top to bottom, resulting in a further reduction of memory consumption.

The example trie from Fig. 3.5 require 3 steps to access any leaf data. Assume that one wants to access a value stored at the point (0.5, 0.75, 0.25) in a sparse grid. Each component of the point triggers a jump to a child node in the tree. First, a pointer corresponding to 0.5 is selected from the array of pointers found at the root of the tree. Second, a jump is made to the address contained in the pointer and the procedure is repeated by choosing a second pointer for 0.25. A final jump is made from that pointer to an array containing the value for 0.75.

3.5 A Memory Efficient Data Structure

This section describes a special data structure for dimensionally truncated sparse grids based on a bijective mapping which minimizes memory footprint. As shown next, its foundation is built on top of two dynamic programming algorithms covered at the end of the section.

3.5.1 The *gp2idx* Bijective Mapping

Data structures such as hash-tables and trees store a truncated sparse grid as a set of point-value pairs. This is in general far from being a memory efficient solution as the key space needs $\mathcal{O}(d)$ more memory than the value space. Since GPUs are equipped with a rather small amount of memory, a large memory footprint does not allow for an efficient usage of GPUs. Therefore, the objective is to develop a data structure that minimizes the memory footprint without sacrificing the access time to the sparse grid data. This can be achieved using a bijection based data structure in which only the values of the sparse grid are stored in memory, without explicit information on the grid points associated with the values. The goal is to store the values in a specially ordered and dense 1d array. A bijective mapping can then calculate the index in the array for any given multi-dimensional grid point, and vice versa, given an index, its corresponding point is returned.

The key concept behind a bijective mapping is the decomposition of a truncated sparse grid into simple dense structures easy to linearize. Fig. 3.6 depicts such a layout for the 2d truncated sparse grid from Fig. 3.2b. In this figure as in the rest of the chapter, the levels are counted starting from 0 instead of 1, meaning that $\underline{l} \in \{0, \dots, n-1\}^d$ and $|\underline{l}|_1 \leq n-1$. There is no modification applied to the refinement level n and the constraint vector $\underline{c}$.

In this data layout, the coefficients of the sparse grid are stored as a sequence of dense *blocks*. A *block* contains the values corresponding to all the grid points that share the same $\underline{l}$. Accordingly, the size of a *block* is $2^{|\underline{l}|_1}$. From an implementation point of view, a *block* is a multi-dimensional array, i.e. $b[2^{l_1}]\dots[2^{l_d}]$. Each *block* can be uniquely identified using $\underline{l}$ which can be referred to as *block* identifier. In sparse grid terms, a *block* maps to a hierarchical subspace.

Another dense structure built on top of *blocks* is the *group* defined as the set of *blocks* for which their $\underline{l}$ vectors have the same L^1-norm. Based on this definition, the unique identifier of a *group* is the scalar $|\underline{l}|_1$, i.e. if $|\underline{l}|_1 = j$ then the *group*'s identifier is j.

The bijective mapping requires that the *groups* and the *blocks* are ordered. First, the groups are ordered ascendingly according to their identifiers. Second, within a *group* identified by j, an ascending order is based on the following comparison rule between the identifiers $\underline{u}$ and $\underline{v}$ ($|\underline{u}|_1 = |\underline{v}|_1 = j$) of any two *blocks*:

$$\underline{u} < \underline{v} \leftrightarrow \exists k : u_k < v_k \text{ and } \forall t > k, u_t = v_t. \tag{3.9}$$

The order allows for the definition of the bijective mapping *gp2idx* (and its inverse *idx2gp*). The *gp2idx* function takes a sparse grid point represented using the pair $(\underline{l}, \underline{i})$ and returns its corresponding position in the sparse grid's linear representation, built using a sequence of *groups*

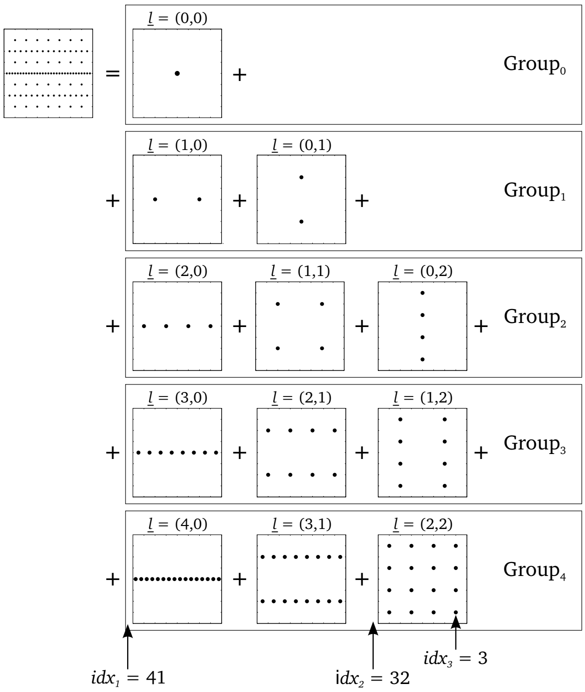


Figure 3.6: Decomposition of a 2d truncated sparse grid (Fig. 3.2b) into *block* structures. The point (0.875, 0.125) corresponding to $\underline{l} = (2, 2)$ and $\underline{i} = (7, 1)$ maps to index 76 $(= 41 + 32 + 3)$ in the linear representation of the sparse grid.

in which each *group* contains a sorted set of dense multi-dimensional *blocks*. First, *gp2idx* finds the *group* that contains the value for a given sparse grid point. Then, it finds the *block* within the *group*. Finally, the position of the searched value within its *block* is determined. Consequently, the index returned by *gp2idx* for $(\underline{l}, \underline{i})$ is a sum of 3 indices:

- idx_1 is the number of values in *groups* whose identifiers are smaller than $|\underline{l}|_1$
- idx_2 is the number of values stored in *blocks* whose identifiers are smaller than $\underline{l}$ according to the comparison rule from Eq. 3.9
- idx_3 is the position for $\underline{i}$ within the *block* identified by $\underline{l}$.

Using the notation $a(d, j)$ for the number of *blocks* in the *group* identified by j, the total number of values in the *group* is $a(d, j) \cdot 2^j$, since the size of a *block* is 2^j. As idx_1 counts all the values stored in *groups* whose identifiers are smaller than $|\underline{l}|_1$, its equation is:

$$idx_1 = \sum_{k=0}^{|\underline{l}|_1 - 1} a(d, k) \cdot 2^k. \tag{3.10}$$

The next subsection shows that $a(d, k)$ executes in $\mathcal{O}(d \cdot n)$ time and can be obtained through dynamic programming algorithms. Since parameters d and n normally have small values, e.g.

Listing 3.3 Dynamic programming solution for problem 1.

```
for j = 0 to n do
  a[1][j] = 1
for i = 1 to d do
  a[i][0] = 1

for i = 2 to d do
  for j = 1 to min(n, c[i]) do
    a[i][j] = a[i][j - 1] + a[i - 1][j]
  for j = min(n, c[i]) + 1 to n do
    a[i][j] = a[i][j - 1] + a[i - 1][j] - a[i - 1][j - c[i]]
```

$d \leq 10$ and $n \leq 10$, a can be memorized in a lookup table in order to save processing cycles.

Consider $pos(\underline{l})$ a function that returns the position of a *block* identified by $\underline{l}$ within its *group*. The number of values in any *block* from the *group* is $2^{|\underline{l}|_1}$. As idx_2 counts the number of values preceding the *block* with identifier $\underline{l}$ in its *group*, its value is returned by the equation:

$$idx_2 = \mathrm{pos}(\underline{l}) \cdot 2^{|\underline{l}|_1}. \tag{3.11}$$

As presented in the next subsection, $pos(\underline{l})$ has a complexity of $\mathcal{O}(d+n)$ provided that $a(d,j)$ is stored in a lookup table so that the access to it is performed in $\mathcal{O}(1)$ time.

Let $\tilde{i}_t$ be defined as $\tilde{i}_t := (i_t - 1)/2, \forall t \in \{1, \ldots, d\}$. Since idx_3 results from the linearization of a dense multi-dimensional array, it is calculated using the following equation:

$$idx_3 = (\ldots(\tilde{i}_1 \cdot 2^{l_2} + \tilde{i}_2) \cdot 2^{l_3} + \cdots + \tilde{i}_{d-1}) \cdot 2^{l_d} + \tilde{i}_d. \tag{3.12}$$

3.5.2 Dynamic Programming Algorithms

The bijective mapping *gp2idx* includes two functions, a and *pos*, which are based on dynamic programming algorithms. The first algorithm finds the number of vectors $\underline{v} := (v_1, \ldots, v_d)$ that contain only positive integers, subject to $d+1$ constraints specified via a scalar n and a constraint vector $\underline{c} := (c_1, \ldots, c_d)$. Given a comparison function that orders any pair of two vectors (resulting in a sequence), the second algorithm determines the index in the sequence for any given vector. The two algorithms are presented in detail, including a complete description of the problems that they solve and formal proofs for correctness.

Problem 1. Find the number of vectors $\underline{v}$ that contain positive integer components and satisfy the constraints: (1) $v_t < c_t, \forall t \in \{1, \ldots, d\}$ and (2) $|\underline{v}|_1 = n$, where $|\underline{v}|_1 := \sum_{t=1}^{d} v_t$.

Solution. $a[d][n]$ calculated in Listing. 3.3.

Proof. Let $\mathrm{sub}(\underline{v}, g, h)$ be a notation for a vector with $h - g + 1$ components, that contains the components of $\underline{v}$ between position g and position h:

$$\mathrm{sub}(\underline{v}, g, h) := (v_g, v_{g+1} \dots, v_{h-1}, v_h). \tag{3.13}$$

Let $a(i, j)$ denote the number of solution vectors $\underline{v}$ with i components and $|\underline{v}|_1 = j$. It is obvious that $a(1, j) = 1$ and $a(i, 0) = 1$ for any $j \geq 0$ and $i \geq 1$ respectively. Moreover, consider that $i \geq 2$ and the i-th component of a generic solution vector $\underline{v}$ is fixed. The first constraint leads to $v_i \leq \min(j, c_i - 1)$. Based on the second constraint, one can write $|\mathrm{sub}(\underline{v}, 1, i-1)|_1 = j - v_i$. This means that the number of vectors $\underline{v}$ with the i-th component fixed is equal to $a(i - 1, j - v_i)$. Considering all the possibilities for the i-th component of $\underline{v}$, the following recursive equation is obtained:

$$a(i, j) = \sum_{t=0}^{m_i} a(i - 1, j - t), \quad m_i = \min(j, c_i - 1), \quad i \geq 2, j \geq 1. \tag{3.14}$$

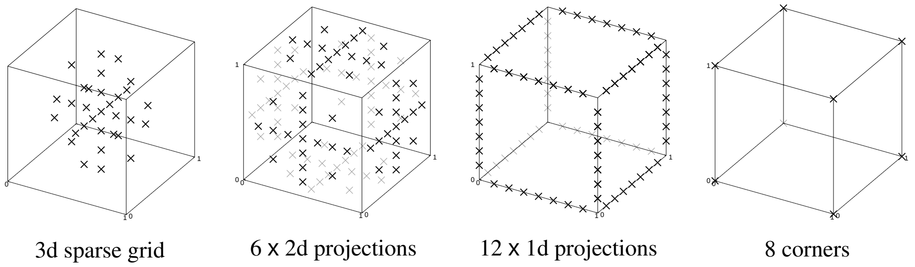
The combination between Eq. 3.14 and *memoization* leads to a dynamic programming algorithm. More precisely, the algorithm is obtained by storing $a(d, n)$ as a 2d array (d rows, $n + 1$ columns) and filling it row by row in ascending order of the row index. In this iterative form, the algorithm computes $a(d, n)$ and has $\mathcal{O}(d \cdot n^2)$ complexity. In this case, $\mathcal{O}(d \cdot n)$ results from traversing the whole 2d array and $\mathcal{O}(n)$ is the theoretical time spent in the innermost loop that calculates Eq. 3.14. As shown next, the $\mathcal{O}(d \cdot n^2)$ complexity can actually be reduced to $\mathcal{O}(d \cdot n)$.

Consider the situations $j < c_i$ and $j \geq c_i$. By expanding and subtracting one obtains:

$$a(i, j) - a(i, j - 1) = \begin{cases} a(i - 1, j) & \text{for } j < c_i \\ a(i - 1, j) - a(i - 1, j - c_i) & \text{for } j \geq c_i \end{cases} \tag{3.15}$$

This gives in fact a more efficient method than Eq. 3.14 for filling the 2d array a. More precisely, for computing $a(i, j)$, at most 3 previously calculated values are needed. Based on Eq. 3.15, one obtains Alg. 3.3 for determining $a(i, j)$, where $1 \leq i \leq d$ and $0 \leq j \leq n$. The complexity of this algorithm is $\mathcal{O}(d \cdot n)$. □

At this point, there is a solution for determining the number of vectors requested in the first problem. Using the comparison rule from Eq. 3.9, the set of solution vectors is converted into a sequence. The objective is now to find the position of a given vector in the sequence. Calculating the position has to be fast as it is executed in the innermost part of the sparse grid algorithms. This requirement is met through a dynamic programming algorithm that returns the position of any vector in the sequence in $\mathcal{O}(d + n)$ time.



3d sparse grid 6 x 2d projections 12 x 1d projections 8 corners

Figure 3.7: 3d sparse grid with non-zero boundary. The boundary of a 3d sparse grid is composed of lower-dimensional sparse grids with zero boundary.

Problem 2. Consider a sequence of d-dimensional vectors that have the same L^1-norm, n, satisfy the constraints (1) and (2), and are ordered according to Eq. 3.9. For a vector $\underline{v}$ from this sequence, determine its position, $pos(\underline{v})$, in the sequence.

Solution.

$$\mathrm{pos}(\underline{v}) = \sum_{t=2}^{d} \sum_{j=0}^{v_t-1} a(t-1, |\mathrm{sub}(\underline{v}, 1, t-1)|_1 + j) \tag{3.16}$$

Proof. The central idea is to count all the vectors that are smaller than $\underline{v}$ and have the L^1-norm equal to $|\underline{v}|_1$. Let $\underline{s} := (s_1, \ldots, s_d)$ be such a vector. According to Eq. 3.9, there must be an i so that $s_i < v_i$ and $\mathrm{sub}(\underline{s}, i+1, d) = \mathrm{sub}(\underline{v}, i+1, d)$, meaning that all the $\underline{s}$'s components after i are equal to the ones of $\underline{v}$. Hence, the sum of $\underline{s}$'s first $i-1$ components must satisfy $|\mathrm{sub}(\underline{v}, 1, i-1)|_1 \leq |\mathrm{sub}(\underline{s}, 1, i-1)|_1 < |\mathrm{sub}(\underline{v}, 1, i-1)|_1 + v_i$.

This is actually the link to the first problem. Taking into account all the possibilities for $|\mathrm{sub}(\underline{s}, 1, i-1)|_1$, one finds that the number of vectors smaller than $\underline{v}$ with i fixed is $\sum_{j=0}^{v_i-1} a(i-1, |\underline{v}|_1 - |\mathrm{sub}(\underline{v}, i, d)|_1 + j)$. Counting all the vectors smaller than $\underline{v}$ is equivalent to considering all the possible values for i, i.e. $i \in \{2, \ldots, d\}$. This results in Eq. 3.16. □

3.5.3 Extension for Non-zero Boundary Sparse Grids

One of the assumptions made to simplify the sparse grid theory is that the functions represented using sparse grids are zero-boundary functions. The bijective mapping can actually be extended to cover the non-zero boundary case based on the observation that the grid points on the boundary belong to a set of lower-dimensional zero boundary sparse grids. This is a direct result of the fact that the sparse grids on the boundary are projections of a non-zero boundary sparse grid on lower-dimensional hyperplanes that form the surface of the hypercube $[0,1]^d$. Fig. 3.7 shows this aspect through the decomposition of a non-zero boundary 3d sparse grid into a zero boundary 3d sparse grid followed by a sequence of zero boundary lower-dimensional sparse grids.

Given the layout exemplified in Fig. 3.7, the bijective mapping is applicable to any of the lower-dimensional sparse grids. However, before invoking the bijection, it is necessary to first indicate the sparse grid that contains a given grid point. This is done by grouping the sparse grids according to their number of dimensions. The size of a group of sparse grids corresponding to the number of dimensions j is $2^{d-j} \cdot C_d^j$, where C_n^k is the number of k-combinations from a set of n elements. The definition of an order for the sparse grids from the same group is necessary in order to find the sparse grid that contains the given point. Such an order can be built on top of the comparison rule provided in Eq. 3.9. Once the sparse grid is found, *gp2idx* can be used to return the index for the point in the 1d representation of the sparse grid.

3.5.4 The Case of Regular Sparse Grids

Regular sparse grids are a special type of dimensionally truncated sparse grids. Compared to regular sparse grids, truncated sparse grids are better fitted for anisotropic multi-dimensional functions where they may employ fewer points than regular sparse grids without sacrificing accuracy. As presented before, in terms of parameters, regular sparse grids lack the constraint vector which in the truncated case is used to tune the refinement level of the grid on a per dimension basis. Besides this aspect, in the case of regular sparse grids, determining idx_1 and idx_2 from the bijective mapping relies purely on combinatorics [43] as opposed to truncated sparse grids for which combinatorics is not applicable because of the constraint vector. Regarding execution time, the access to the data of a regular sparse grid is done in $\mathcal{O}(d)$ which is faster than the $\mathcal{O}(d+n)$ for the truncated case. Therefore, it is important that whenever all the components of the constraint vector are equal, resulting in a regular sparse grid, the implementation described in [43] is used.

3.6 Non-recursive Sparse Grid Algorithms

Non-recursive sparse grid algorithms are required by processor architectures such as GPUs where recursion is not possible or not supported efficiently. Moreover, non-recursive implementations are not affected by overheads caused by excessive operations involving the stack, e.g. saving and restoring registers, copying parameter values for every invocation of a recursive function. Furthermore, as shown in the next chapter, the non-recursive algorithms for multi-dimensional hierarchization and interpolation also have the benefit that they simplify parallelization.

Listing 3.4 Non-recursive multi-dimensional sparse grid hierarchization using the bijection. **Input**: d, *numGridPoints*, *sg1d*[*numGridPoints*]. **Output**: *sg1d*[*numGridPoints*].

```
1: for t = 0 to d - 1 do
2:   for j = numGridPoints - 1 downto 0 do
3:     gp = idx2gp(j)
4:     lp = leftParent(gp, t)
5:     rp = rightParent(gp, t)
6:     lv = sg1d[gp2idx(lp)]
7:     rv = sg1d[gp2idx(rp)]
8:     sg1d[j] = sg1d[j] - (lv + rv) / 2
```

3.6.1 Non-recursive Hierarchization

A non-recursive algorithm for hierarchization based on the *gp2idx* bijection is shown in Listing 3.4. The sparse grid values are stored in memory in a manner similar to the one depicted in Fig. 3.6. What the algorithm does is to traverse the sparse grid d times, each time updating every value based on the values of the hierarchical parents in the current dimension t. Therefore, the outer loop iterates over the dimensions whereas the inner loop iterates over the indices of the sparse grid points in the 1d representation of the sparse grid, i.e. the array *sg1d*. Notice that the inner loop starts with the highest possible index. This is done to preserve the semantics of the recursive algorithm in which child sparse grid points are updated before their parents.

In line 3, the index is transformed into the grid point *gp* represented using the $(\underline{l}, \underline{i})$ pair. Then, *gp* is used to compute its dependencies in the current dimension, i.e. the left and the right hierarchical parent, stored in the variables *lp* and *rp* respectively. Determining the left parent, *lp*, in dimension t is done by *leftParent(gp, t)*. Assume that *lp* is represented by the pair $(\underline{ll}, \underline{li})$. For dimension t, the relation between the current grid point and the left parent is:

$$\begin{gathered} li_t \cdot 2^{-ll_t} = (i_t - 1) \cdot 2^{-l_t}, \\ ll_k = l_k, li_k = i_k, \quad \forall k \in \{0, \dots, d-1\} \setminus \{t\}. \end{gathered} \tag{3.17}$$

This is actually the equation solved by *leftParent(gp, t)*. Similarly, the right parent *rp* identified through the pair $(\underline{rl}, \underline{ri})$ is calculated by *rightParent(gp, t)* using:

$$\begin{gathered} ri_t \cdot 2^{-rl_t} = (i_t + 1) \cdot 2^{-l_t}, \\ rl_k = l_k, ri_k = i_k, \quad \forall k \in \{0, \dots, d-1\} \setminus \{t\}. \end{gathered} \tag{3.18}$$

From this point on, the explanation of the algorithm is straightforward. In lines 6 and 7 the hierarchical parents are transformed into indices using the bijection *gp2idx* and then their values are obtained by indexing the *sg1d* array. Finally, in line 8, the value at the current position in

Listing 3.5 Non-recursive multi-dimensional sparse grid interpolation using the bijection. **Input**: d, n, $a[d][n]$, $sg1d[]$, m, $x[m][d]$. **Output**: $r[m]$.

```
1:  for j = 0 to m - 1 do
2:    r[j] = 0
3:    idx12 = 0
4:    for g = 0 to n - 1 do
5:      for b = 0 to a[d][g] - 1 do
6:        l = invPos(g, b)
7:        idx3 = 0
8:        p = 1
9:        for t = 0 to d - 1 do
10:         idx3 = idx3 * 2^l[t] + floor(2^l[t] * x[j][t])
11:         p = p * basis1d(l[t], x[j][t])
12:       r[j] = r[j] + sg1d[idx12 + idx3] * p
13:       idx12 = idx12 + 2^g
```

sg1d is updated using its old value and the values of the dependencies.

3.6.2 Non-recursive Interpolation

Listing 3.5 represents a non-recursive multi-dimensional algorithm for sparse grid interpolation. As seen in line 6, it uses *invPos*, i.e. the inverse of the *pos* function from Eq. 3.16. Here, interpolation is done for a set of m d-dimensional points from $[0,1]^d$. These points are contained in the 2d array $x[m][d]$. The interpolation results are stored in the $r[m]$ array. For every interpolation point, the algorithm traverses the sparse grid, *block* by *block*, and computes the contribution of each *block* to the result for the current interpolation point. The contribution of a *block* consists in the product between a hierarchical coefficient from the *block* and a d-dimensional basis function evaluated at the current interpolation point.

The j loop from line 1 of the algorithm iterates over the set of points where the sparse grid is interpolated. The g and b loops traverse *groups* and *blocks* respectively. In line 6, the identifier of the current *block* is determined. The *idx12* index points to the beginning of the current *block* in the array *sg1d*. For every interpolation point corresponding to a row of x, only one hierarchical coefficient from the *block* is used. As previously mentioned, the basis functions corresponding to a *block* or hierarchical subspace have pairwise disjoint supports that cover the entire domain, meaning that only one basis function is non-zero for any given interpolation point and accordingly, only its coefficient must be selected from the *block*. In the algorithm, that coefficient is found using *idx3*, calculated based on l and $x[j]$. After the t loop, *idx3* is used to indicate the right coefficient in a standard linearization of the multi-dimensional dense array used to represent the current *block*. Line 11 contains the evaluation of a linear basis function at component t of the current interpolation point $x[j]$. *basis1d* is defined here by:

$$\mathit{basis1d}(k, y) := 1 - |2^{k+1} \cdot y - \lfloor 2^k \cdot y \rfloor \cdot 2 - 1|, \quad k,\, y \text{ scalars.} \tag{3.19}$$

The results of all the evaluations of linear basis functions are multiplied and stored in p which represents the evaluation of the d-dimensional hierarchical basis function at the current point. In line 12, this result is further scaled by a hierarchical coefficient and then added to the other contributions in the output array r.

3.7 *fastsg*, a Lightweight Sparse Grid Library

fastsg is a collection of C++ routines for the sparse grid technique. It is based on the *gp2idx* bijective mapping so that applications built on top of it can benefit from the *minimal memory consumption* resulting from the bijection. The main goal behind developing *fastsg* is to provide a reduced set of functions for interpolating dimensionally truncated sparse grids. This functionality is of high importance in the context of the computational steering scenario described at the beginning of this chapter. Besides minimal memory consumption, another characteristic of the *fastsg* library is that its contained functions run efficiently on heterogeneous systems, a topic covered in depth in Chapter 4, Chapter 5, and Chapter 6.

Table 3.1 shows the interface to *fastsg*'s functionality. The library has a *triple layer design* with a clear separation between the data structure and the sparse grid algorithms, thus providing both flexibility and modularity. Hence, the first level (topmost in the table) is used for operating efficiently with the linear representation of the sparse grid. The second level provides access to the sparse grid functions, mainly *hierarchize* and *interpolate.* The third level contains special purpose routines, e.g. input specialized functions that address scenarios characteristic to computational steering (see Section 3.8 for more details).

In order to use the library, the following input parameters have to be provided: the refinement level of a sparse grid through the n parameter, the constraint array c, and a function f that can extract the data required by the sparse grid discretization from some multi-dimensional data. The *init* function applies the sparse grid discretization to the multi-dimensional data provided as input. After initialization, *hierarchize* is invoked in order to transform the discretized data into a representation using the hierarchical basis functions. From this point on, *interpolate* can be called at any time to evaluate the sparse grid approximation at any given set of points inside the $[0, 1]^d$ domain. For a given set of m points stored in x, the *error* routine returns the

Layer	Routine	Description
Data structure	init(d, n, $c[d]$, f)	Allocates a d-dimensional truncated sparse grid with refinement level n and constraint array c. Initialization is done using the function f.
	gp2idx($l[d]$, $i[d]$)	Converts the grid point represented using the pair of arrays (l, i) into the corresponding index in the linear representation.
	idx2gp(idx)	Converts the index in the linear representation into the corresponding multi-dimensional grid point, returned as a pair of two arrays (l, i).
	size()	Returns the number of points in the truncated sparse grid.
Sparse grid algorithms	hierarchize()	Based on f's values at the sparse grid points, computes the α coefficients of the approximation and stores them in the sparse grid.
	interpolate($x[m][d]$)	Returns the approximation's values at m d-dimensional points (stored in the matrix x) inside f's domain.
	error($x[m][d]$)	Returns the error of the sparse grid approximation, by comparing the results returned by the approximation against those returned by f.
Special purpose	interpolateSha($sel[sd]$, $x[m][d]$)	Special sparse grid interpolation for the *sha. pattern.*
	interpolateCar($xc[d]$, $size[d]$)	Special version of interpolation corresponding to the *car. pattern.*
	interpolateAuto($x[m][d]$)	Detects patterns in the set of interpolations points and calls the right function: interpolate, interpolateSha, or interpolateCar.
	initVec(d, n, $c[d]$, f, s)	Allocates a d-dimensional truncated sparse grid in which at each point a vector of size s is stored.

Table 3.1: Triple-layer interface exposed by *fastsg*. Only the most important routines are shown.

L^2 relative error of the sparse grid approximation based on the equation:

$$err := \frac{\sqrt{\sum\limits_{t=0}^{m-1} (f_c(x_t) - f(x_t))^2}}{\sqrt{\sum\limits_{t=0}^{m-1} f(x_t)^2}} \tag{3.20}$$

where f_c represents the sparse grid approximation and x_t is the t-th point in the set. One can imagine an iterative process in which sparse grids are created using different values for n and c until the approximation error is below a required limit.

Looking back at the computational steering application described at the beginning of this chapter, *fastsg* is employed there for compressing and decompressing high-dimensional simulation data provided to the library through the parameter f of *init*. More precisely, compression is achieved through a combination of *init* and *hierarchize*: The input data is filtered using *init* and is represented hierarchically using *hierarchize*. In sparse grid terminology, decompression is represented by *interpolation*. In the context of computational steering, the performance of *fastsg*, both in terms of memory consumption and execution time, has an essential role in ensuring a low response time for exploring and visualizing the data.

3.8 Special Features of fastsg

As the main requirements behind *fastsg* come from computational steering, the routines have versions that exploit certain special scenarios that are common in practice. Therefore, a special feature of *fastsg* is input specialized or input aware interpolation in which patterns in the input data are harnessed in order to reduce the number of executed operations and consequently the execution time. Furthermore, another feature addresses a scenario in which at every point in the sparse grid a vector of values is stored, instead of a single value as before. This typically results from isolating the spatial parameters from the other parameters of a simulation. The sparse grid discretization is then applied only to the space of non-spatial parameters.

3.8.1 Input Specialized Algorithms for Interpolation

Input specialization for interpolation is based on the existence of patterns in the input data, more precisely in the set of interpolation points. The patterns can be exploited in order to reduce the execution time of interpolation. For computational steering, specialization is based on two observations regarding characteristics commonly found in the set of interpolation points represented as a matrix x (a row of x is a d-dimensional interpolation point):

1. In order to visualize d-dimensional data, the interpolation points are actually 3d (the spatial dimensions are variable while the non-spatial dimensions are fixed), meaning that for each column in a set of $d-3$ columns from x, the column's components are all equal.

2. The interpolation points often result from a regular grid discretization, i.e. x is the result of the Cartesian product of d sets of values, each corresponding to a dimension.

From this point on, (1) is referred to as *sha pattern* (shared value) whereas (2) corresponds to the *car pattern* (Cartesian product). Each pattern is addressed by a specific interpolation algorithm. In Table 3.1, the corresponding routines are *interpolaSha* and *interpolateCar*. The *interpolateAuto* routine automatically detects the patterns. Moreover, if a pattern is found, then its corresponding routine is invoked, otherwise the default *interpolate* routine is called.

The sha Pattern

Listing 3.6 represents the specialized algorithm for *sha*. For simplicity, it contains only the core of sparse grid interpolation, i.e. it computes the contribution of one *block* resulting from the decomposition of the sparse grid, to all the interpolation results. In other words, it is the semantical equivalent of line 1 and lines 7 - 12 from Listing 3.5. The implementation of this algorithm can be explicitly invoked from *fastsg* or can be called automatically based on a pattern detection procedure. The detection of *sha* starts by traversing x and checking if there are columns containing one value across all rows. For those columns with this property, their index is saved in an array called *sel*. Let *sd* be the number of indices added to *sel*. Another array, *isel*, contains the indices of the remaining columns. Thus, the detection of *sha* executes in $\mathcal{O}(m \cdot d)$ time. This may seem expensive but it is actually worthwhile provided that the number of the *blocks* in the sparse grid is large enough.

The benefit of the *sha* specialized algorithm over the default interpolation algorithm from Listing 3.5 consists in a reduction of the number of executed operations. Since there are *sd* columns in x whose components share one value, a part of the product p, stored in the variable *sp*, is moved outside the innermost loop from Listing 3.5 to line 5 in Listing 3.6. This results in fewer floating point operations needing to be executed.

In order to obtain the $\mathcal{O}(m \cdot (d - sd))$ complexity seen in Listing 3.5 for the specialized algorithm, part of the *idx3* computation must also be moved outside the innermost loop from Listing 3.5. For this to happen, the array *psum* is introduced. It is used to store the prefix sums of the array l which identifies a *block*. Using the prefix sums, calculating *idx3* becomes a standard reduction. Moreover, the computation of *sidx3* corresponding to the *sd* shared dimensions is moved outside the innermost loop, thus reducing the number of iterations in that

Listing 3.6 Special interpolation core for the *sha pattern.*
Input: m, d, $x[m][d]$, $l[d]$, $b[2^{l[1]}]...[2^{l[d]}]$, sd, sel, $isel$. **Input / output**: $r[m]$.

```
psum[d] = 0
psum[d - 1] = l[d - 1]
for t = d - 2 downto 0 do
  psum[t] = psum[t + 1] + l[t]
sp = 1
sidx3 = 0
for t = 0 to sd - 1 do
  sidx3 = sidx3 + floor(2^l[sel[t]] * x[0][sel[t]]) * 2^psum[sel[t]]
  sp = sp * basis1d(l[set[t]], x[0][sel[t]])
for j = 0 to m - 1 do
  p = sp
  idx3 = sidx3
  for t = 0 to d - sd - 1 do
    idx3 = idx3 + floor(2^l[isel[t]] * x[j][isel[t]]) * 2^psum[isel[t] + 1]
    p = p * basis1d(l[isel[t]], x[j][isel[t]])
  r[j] = r[j] + sg1d[idx12 + idx3] * p
```

loop to $d - sd$ and the complexity to $\mathcal{O}(m \cdot (d - sd))$.

The car Pattern

Let X be a given set of d-dimensional interpolation points. If $X = C_1 \times C_2 \times \cdots \times C_d$, then the interpolation points correspond to the *car pattern.* As a simplification, assume that X does not contain duplicates. In order to detect this pattern, each column of the matrix x is traversed and the unique values per column are counted. This is done using a tree data structure in $\mathcal{O}(\log(m!))$ time per column. Subsequently, the resulting d counters, one for each column, are multiplied. If the product equals m, i.e. the number of rows of x, then x is the result of a Cartesian product. The complexity of *car*'s detection is $\mathcal{O}(d \cdot \log(m!))$. The same observation from *sha* applies also here, meaning that the detection may seem expensive but is amortized by the large number of *blocks* into which the sparse grid is decomposed.

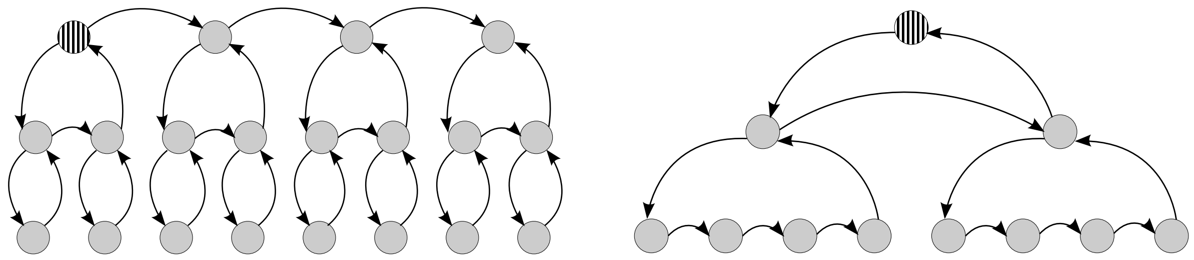
Listing 3.7 represents the specialized interpolation algorithm addressing the *car pattern.* This algorithm is a non-recursive Cartesian product generator with some additions described next. Here, at data structure level, the original matrix x is replaced with a more compact xc vector containing d vectors of different sizes. Each vector i of xc contains the unique values of column i of x. Assuming that the vectors in xc have the same size s, then the memory consumed by the interpolation points reduces from $d \cdot s^d$ to only $d \cdot s$ floating point numbers. Regarding complexity, Listing 3.7 which is based on the traversal of xc, executes in $\mathcal{O}(m)$ time.

Listing 3.7 Special interpolation core for the *car pattern.*
Input: d, $xc[d][maxs]$, $l[d]$, $b[2^{l[1]}]...[2^{l[d]}]$, $size[d]$. **Input / ouput**: $r[m]$.

```
j = 0
t = 0
stack[0] = 0
p[0] = 1
idx3[0] = 0
while t >= 0 do
  if t == d then
    r[j] = r[j] + sg1d[idx12 + idx3[t - 1]] * p[t - 1]
    j = j + 1
    t = t - 1
  else if stack[t] <= size[t] then
    p[t] = p[t - 1] * basis1d(l[t], xc[t][stack[t]])
    idx3[t] = idx3[t - 1] * 2^l[t] + floor(2^l[t] * xc[t][stack[t]])
    t = t + 1
    stack[t] = 0
  else
    t = t - 1
```

Listing 3.7 assumes a more general scenario in which the vectors of *xc* may contain each a different number of values. The sizes of the vectors are stored in the array *size*. In Listing 3.7, *xc* is represented through a $d \times maxs$ matrix, where *maxs* is the maximum of all the integer values stored in the *size* array. Listing 3.7 uses 3 stacks: *st*, *p*, and *idx3*. *st* and its respective index *t* generate the interpolation points using a Cartesian product. The meanings of *p* and *idx3* are the same as before (Listing 3.5) although here they are not scalars. In fact, using them as arrays is the algorithm's element that makes it possible to reuse the results of floating point and integer operations, thus reducing considerably the complexity.

By applying a permutation to the dimensions, the number of operations in Listing 3.7 can be further reduced. Such a permutation makes sense when the vectors in *xc* have different sizes. The central idea is to sort ascendingly the vectors of *xc* according to their sizes and to permute the dimensions according to the order. The advantage of the permutation is explained in Fig. 3.8. Generating the Cartesian product is similar to traversing a tree in a depth-first manner. The *i*-th level in the tree corresponds to the *i*-th vector in *xc*. The orientation of the edges shows how values are traversed in *xc*. The number of edges directly correlates with the number of operations executed. Fig. 3.8a and Fig. 3.8b handle the same computational work but first without permutation and then with the permutation of *xc*. In Fig. 3.8a, the vectors in *xc* have the lengths 4, 2, and 1. After permutation, they become 1, 2, 4. The number of operations (edges) reduces from 31 to 13.



(a) Operations without permutation.

(b) Operations with permutation.

Figure 3.8: The positive effect of permutation on computing the Cartesian product.

3.8.2 Sparse Grids of Vectors

Another special scenario addressed by *fastsg* refers to sparse grids that contain vectors instead of scalar values as before. Such a grid is created through *initVec* from Table 3.1. In computational steering, this results from the separation between spatial (x, y, and z) and non-spatial simulation parameters. There, the motivation comes from complex simulation geometries which cannot be properly discretized using sparse grids. In such cases, the simulation data corresponding to the same combination of non-spatial parameters is linearized and stored in a vector. Consequently, the sparse grid discretization is only used to handle the non-spatial dimensions.

All the algorithms presented in this chapter also apply to sparse grids of vectors with a slight modification: Scalar values are replaced with vector values. In this case, the linear representation of the sparse grid is an array of vectors. The *gp2idx* bijection points to a vector. The semantics of the vectors stored in the array, is application dependent. However, they are treated by *fastsg*'s functions as simple vector operands without a special meaning.

In computational steering scenarios, the number of points in the sparse grid tends to be dominated by the size of the vectors stored at the points. Intuitively, this results from the fact that obtaining simulation results for a large number of combinations of non-spatial parameters can be prohibitively expensive. In this particular case which is rather common in practice, the sparse grid algorithms become traditional linear algebra operations involving the multiplication and addition of matrices and vectors. In the case of hierarchization, it becomes a sequence of operations each involving: scaling and adding up two vectors, and subtracting their sum from a third vector. Since some vectors are reused across operations, this means that the locality can be improved. On the other hand, interpolation is more complex: Interpolating at one point is equivalent to sparse vector - matrix multiplication, referred to as *spvm* and shown in Listing 3.8.

In Listing 3.8, *ix* is an array of integers used to select the grid points that contribute to the interpolation. *x* contains the results from evaluating the basis functions at the given

Listing 3.8 Sparse vector - matrix multiplication.
Input: m, p, $ix[m]$, $x[m]$, $a[][p]$. **Output**: $y[p]$

```
for j = 0 to p - 1 do
  y[j] = 0

for i = 0 to m - 1 do
  for j = 0 to p - 1 do
    y[j] = y[j] + x[i] * a[ix[i]][j]
```

interpolation point. a is here the sparse grid represented as an array of vectors (or matrix) that contain hierarchical coefficients. y is the result of the interpolation. One can notice the same behavior of interpolation as in the scalar case: (1) only a subset of the grid points are actually used in interpolation selected through the sparse vector *ix*, (2) the hierarchical coefficients from a scale the results of basis function evaluations stored in x, and (3) the intermediary results are reduced (summed up) in y. Emphasis is placed here on the fact that in this computational steering scenario, the execution time is dominated by operations involving vectors and matrices.

3.9 Representative Computational Kernels

Although this chapter presents a multitude of sparse grid algorithms, three of them are sufficient to capture the main characteristics of sparse grids for computational steering: sparse grid hierarchization (*sghierarch*), sparse grid interpolation (*sginterp*), and sparse vector - matrix multiplication (*spvm*). These are the algorithms that are discussed in the next chapters in the context of multi-core CPUs and GPUs. They offer a mix of computational behaviors: *sghierarch* is integer bound, *sginterp* is computationally bound, and *spvm* is memory bound.

The input specialized algorithms for interpolation change only slightly the general behavior of interpolation. Therefore, optimizations developed for the most general type of interpolation apply also to the input specialized versions. With regard to hierarchization for sparse grids of vectors, it is memory bound and has a computational behavior similar to *spvm*. Therefore, most of the optimizations proposed to *spvm* are also applicable to it.

3.10 Summary

This chapter describes the computational steering application whose set of routines is used as a benchmark for evaluating the performance of heterogeneous computing. The computational steering approach is based on storing compressed high-dimensional simulation data, e.g. 4 - 10 dimensions, in a database and decompressing it later for visualization. Lossy compression functionality is achieved using a numerical technique called the sparse grid technique which is typically employed when dealing with high-dimensional problems.

A data structure is proposed for dimensionally truncated sparse grids, i.e. anisotropic sparse grids. It minimizes memory consumption through a bijection *gp2idx* which maps grid points to a set of consecutive integers. This helps to cope with the limited amount of memory on GPUs. Using the new data structure, larger problems can be solved on GPUs.

Non-recursive algorithms are built on top of the bijective mapping *gp2idx*. These algorithms are integrated in a library called *fastsg*. The features of *fastsg* include input specialized routines for interpolation and the ability to store dense vectors of values in a sparse grid, a functionality of high importance in computational steering scenarios that deal with complex geometries for which the sparse grid discretization is impractical. In this context, sparse grid interpolation reduces to a dense linear algebra operation, more precisely sparse vector - matrix multiplication.

A set of 3 computational kernels is extracted from *fastsg* for benchmarking GPU based heterogeneous systems. They have different performance behaviors: sparse grid hierarchization (*sghierarch*) is integer bound, sparse grid interpolation (*sginterp*) is computationally bound, and sparse vector - matrix multiplication (*spvm*) is memory bound.

Chapter 4

Sparse Grids on Heterogeneous Systems

The most popular heterogeneous systems nowadays are the ones containing multi-core CPUs and GPUs. This chapter describes optimizations for CPUs and GPUs that accelerate the three computational kernels described in Chapter 3: sparse grid hierarchization (*sghierarch*), sparse grid interpolation (*sginterp*), and sparse vector - matrix multiplication (*spvm*). On CPUs, emphasis is placed on optimizations that improve locality, reduce the number of integer operations, and efficiently use vector units. On the GPU side, a set of GPU specific optimizations is applied to the codes, focusing on improving the access to data stored in different memories on the GPU and ensuring an efficient utilization of the SIMD lanes at any time during execution. By optimizing both the CPU and the GPU versions of the kernels, a comparison is provided pointing to the best processor for each one of the three kernels.

4.1 Introduction

The most common accelerators these days, namely GPUs, are very different from general-purpose CPUs. Whereas CPUs incorporate large caches and complex logic for out-of-order execution, branch prediction, and speculation, in GPUs, most of the transistor budget is allocated to floating point units. GPUs have in-order cores that hide pipeline stalls through interleaved multithreading, e.g. allowing up to 1536 threads to reside and run concurrently on one core. CPUs are generally regarded as latency oriented processors [1] because of the complex techniques that they implement for extracting Instruction Level Parallelism (ILP) from a sequential stream of instructions. At the other end, GPUs are oriented on throughput [1] as they contain a large number of cores (e.g. 16) with wide SIMD units (e.g. 32 single precision lanes), making them ideal architectures for vectorizable codes.

Within a GPU core, a control unit creates, manages, and synchronously executes threads in groups of 32 referred to as warps. Every instruction is synchronously broadcast to all the threads in a warp. Nvidia refers to this execution model as Single Instruction Multiple Threads (SIMT). If the threads in the same warp execute different instructions because of branching, then the execution is serialized, meaning that two or more different instructions are executed sequentially. This undesired behavior is called warp diverge and can severely reduce the performance of a GPU program, up to a factor of 32.

The GPU's global memory, e.g. 6 GB, is much smaller than the memory of the CPU. However, its bandwidth is generally one order of magnitude higher. Transferring data between memories is done over PCIe, a bus that is in some cases a serious performance bottleneck. Flexibility and performance is provided by GPUs through a wide variety of memories: constant cache, texture cache, shared memory, and a coherent two-level cache hierarchy introduced in the Fermi generation of GPUs [11]. The properties of these memories vary in terms of latency, bandwidth, and usage. The constant cache is a read-only cache whose L1 has the lowest latency among the memories on the GPU [53]. The texture cache is also read-only and is used for optimizing the bandwidth rather than latency, i.e. its latency is comparable to the one of the global memory [53]. The shared memory is a low latency, read-write memory with 32 banks which can be accessed in parallel. It is private per core and is controlled explicitly.

In the absence of out-of-order execution, GPUs employ multithreading as the means to cope with instruction pipeline stalls, especially the latency caused by loads and stores to global memory. Multithreading is supported through a rather large register file per core, 32K registers, allowing for a large number of threads to run concurrently on every GPU core. An important aspect is that the context switch between the threads running on the same core has a low cost, meaning that the interleaved execution can be considered free from overheads.

Programming GPUs is inherently different from programming CPUs. In the case of multi-core CPUs, OpenMP is the standard thread based programming model. Nvidia GPUs are typically programmed using CUDA which is also based on threads, but there are some major differences compared to OpenMP. For synchronization, CUDA only provides barriers that can be used within a group of threads executing on the same GPU core, and atomic operations. Furthermore, a CUDA application has a CPU and a GPU part. The CPU part is responsible for: allocating memory on the GPU, transferring data to and from the GPU, and launching GPU programs. Threads are grouped in blocks which typically contain 4 or 8 warps [14]. This grouping is important as only the threads from the same block can synchronize via barriers (*__syncthreads*) and can share data stored in shared memory. A GPU program is in fact launched as a 3-dimensional (3d) grid of thread blocks. In order to identify a block within a grid, a 3d block identifier (*blockIdx*) is used. Similarly, a 3d thread identifier (*threadIdx*) is used to identify a thread within a block. Combining the block and the thread identifiers enables one to globally identify a thread and to assign work to it.

Optimizations on CPUs typically focus on cache and vector units, SSE or AVX. In order to improve cache reuse, loop interchange and loop tiling are standard optimizations applied to loop nests. Efficient vectorization requires that the layout of the data in memory has to be modified, e.g. it has to be aligned to a 16 / 32 byte boundary for SSE / AVX. More importantly, it is often necessary that data structures are transformed so that vector operands are stored in contiguous regions from memory. For many applications, this is usually achieved by converting an array of structures to a structure of arrays.

On the GPU side, a first optimization is to reduce the number of branches in order to minimize warp divergence. A second optimization is the efficient use of the memory hierarchy including the best mapping between data structures and memories. Coalescing accesses to global memory is another important objective. At shared memory level, data access patterns must be tuned so that they allow for a uniform distribution of read and write requests across all the 32 banks of the shared memory, or to one bank when all the threads in a warp access the same address. The third optimization target is multithreading. Choosing the right thread block size must be done so that it results in the maximization of the number of concurrent threads that execute at any time on the GPU.

All these optimizations are applied to the three computational kernels derived from the *fastsg* library. At the end of this chapter, performance results are provided showing the benefits of GPUs over CPUs in the context of the chosen kernels.

4.2 Related work

For a detailed description of standard optimizations for GPU codes, the reader is referred to the CUDA manual [14]. This chapter also explores the applicability to sparse grid algorithms of other tuning techniques validated for dense linear algebra and presented in [17, 54, 55], e.g. trading concurrency for improved register reuse (locality).

The vectorization and parallelization of sparse grid methods have typically been realized using the combination technique [56, 57]. An implementation for GPUs is presented in [58]. In the combination technique, the sparse grid approximation is obtained from a special superposition of smaller anisotropic regular grids. Compared to the bijection based algorithms covered in

Chapter 3, the combination technique has the disadvantage that grid points and their respective values have to be replicated across multiple regular grids [43], thus resulting in suboptimal memory consumption.

Optimized GPU implementations of sparse grid hierarchization and interpolation are described in [43] for regular sparse grids and in [59] for dimensionally truncated sparse grids. In [60], the authors present optimizations for the interpolation of adaptive sparse grids including: the conversion of a recursive interpolation algorithm to a non-recursive form, vectorization on CPUs, and porting to GPUs.

4.3 Optimizations for Multi-core CPUs

This section describes the main optimizations applied to *sghierarch*, *sginterp*, and *spvm*. The optimizations are formulated in terms of loop transformations [61], e.g. loop invariant code motion, loop interchange, loop tiling, and loop vectorization.

4.3.1 Sparse Grid Hierarchization on CPUs

The algorithm for *sghierarch* is shown in Listing 4.1. This algorithm is a slightly modified version of the algorithm from Listing 3.4 although the semantics and the complexity is the same as before. More precisely, the j loop iterating over the sparse grid points in Listing 3.4 is replaced in Listing 4.1 with a sequence of 3 loops:

- The g loop iterates over groups of sparse grid points.
- The b loop iterates over blocks in the current group g.
- The k loop iterates over the sparse grid points in the current block b.

One can verify that the semantics is preserved by looking at the decomposition of a sparse grid into groups and blocks shown in Fig. 3.6. The algorithm maps exactly to the traversal of that decomposed sparse grid in a bottom-up manner in order to avoid destroying data dependencies. Although this transformation may seem unnecessary, the new equivalent form of non-recursive multi-dimensional hierarchization contains the loop nest, given by the sequence of loops t, g, b, k, that can be transformed in order to improve the performance.

The main characteristic of this algorithms is that it is integer bound because of the expensive invocations to the *gp2idx* bijection and to its inverse *idx2gp* in the innermost loop. Both are based entirely on integer calculations. Using complexities to explain the nature of the algorithm, *gp2idx* has $\mathcal{O}(d + n)$ complexity whereas the floating point operations concentrated in line 11

Listing 4.1 Non-recursive multi-dimensional sparse grid hierarchization using the bijection. **Input**: d, *numGridPoints*, *sg1d*[*numGridPoints*]. **Output**: *sg1d*[*numGridPoints*].

```
1:  for t = 0 to d - 1 do
2:    j = size()
3:    for g = n - 1 downto 0 do
4:      for b = a(d, g) - 1 downto 0 do
5:        for k = 2^g - 1 downto 0 do
6:          (l, i) = idx2gp(j)
7:          (ll, li) = leftParent(l, i, t)
8:          (rl, ri) = rightParent(l, i, t)
9:          lv = sg1d[gp2idx(ll, li)]
10:         rv = sg1d[gp2idx(rl, ri)]
11:         sg1d[j] = sg1d[j] - (lv + rv) / 2
12:         j = j - 1
```

of the algorithm execute in $\mathcal{O}(1)$ time. Therefore, a first optimization direction focuses on decreasing the number of integer operations. As shown next, this is achieved mainly by moving the invariant code from the innermost loop, i.e. loop k. More precisely, the optimizations target the following points: (1) moving the computation of l shared by all the grid points within the same block outside the innermost loop, (2) reducing the access time to the hierarchical parents (dependencies) that reside in maximum $n-1$ blocks whose indices in *sg1d* (the linear representation of the sparse grid) can be determined before entering loop k, and (3) reducing the theoretical time for calculating the index vectors i, *li*, and *ri* from $\mathcal{O}(d)$ (originally) to $\mathcal{O}(1)$.

A first optimization, called *hopt1*, uses the fact that all the points in a sparse grid block have the same l. Thus, l can be computed once for an entire block which contains 2^g points (g is the L^1-norm of l). In terms of code modifications, *hopt1* introduces a call to *invPos(g, b)* outside loop k. By doing so, line 6 has to execute only the part of *idx2gp* that computes the index vector i in $\mathcal{O}(d)$ time, instead of invoking the whole *idx2gp* in $\mathcal{O}(d+n)$ time.

A second optimization, *hopt2*, reduces the number of integer operations performed for accessing the values of the hierarchical parents. Consider the left parent in dimension t represented by the pair of vectors *(ll, li)* and returned in line 7 by invoking *leftParent(l, i, t)*. The relation between *ll* and l in the innermost loop satisfies the equation:

$$ll_t < l_t \text{ and } ll_u = l_u, \forall u \in \{0, \ldots, d-1\} \setminus t. \tag{4.1}$$

Based on this equation, one can see that there can be only $l_t - 1$ blocks where the left hierarchical parents can reside for any point in the block identified using l. The same reasoning can also be applied to the right parent, leading to the same set of blocks where dependencies reside.

Consequently, it makes sense to move outside the innermost loop the computation of the indices in *sg1d* corresponding to the $l_t - 1$ blocks that contain the hierarchical parents. This is in fact *hopt2*. Its effect is a reduction of the number of integer operations executed in lines 9 and 10 by invoking the *gp2idx* bijection. After applying *hopt2*, what remains from *gp2idx* in line 9 is the part of it that transforms the vector *li* to a scalar index used to access the value of the hierarchical parent in the 1-dimensional (1d) representation of the block that contains it. This means that the complexity is reduced from $\mathcal{O}(d+n)$ to $\mathcal{O}(d)$ in lines 9 and 10.

For *hopt2*, an array *parIdx* is used to memorize the indices in *sg1d* for a number of l_t blocks that contain the hierarchical parents for the current block identified through l. In order to simplify the final optimized algorithm, the l_t-th component of *parIdx* is set to the index in *sg1d* of the current block l. The components of *parIdx* are determined outside the k loop based on:

$$\begin{gathered} parIdx_u = \sum_{v=0}^{|l|_1 - l_t + u - 1} a(d,v) \cdot 2^v + pos(pl) \cdot 2^{|l|_1 - l_t + u}, \quad \forall u \in \{0, \ldots, l_t\} \\ pl_t = u \text{ and } pl_v = l_v, \forall v \in \{0, \ldots, d-1\} \setminus \{t\}, \end{gathered} \tag{4.2}$$

where a is from Eq. 3.15, *pos* is from Eq. 3.16, and *pl* is the level vector of a generic hierarchical parent. As an example, look at line 9. Using *hopt2*, the position in *sg1d* of the block containing the left hierarchical parent in dimension t is obtained by indexing *parIdx* using ll_t. By summing up the position of the block and the index within the block resulting from the linearization of the vector *li*, the index of the left parent in *sg1d* is found. The same procedure is then followed in line 10 for calculating the index in *sg1d* for the right hierarchical parent.

A third optimization, *hopt3*, exploits the fact that a generic hierarchical parent represented using *(pl, pi)* shares $d-1$ components of its child *(l, i)*. More precisely, only the t-th component of *pl* and *pi* is different compared to l and i respectively. By applying *hopt1* and *hopt2*, most of the integer computation involving l, *ll*, and *rl* move outside the innermost loop. *hopt3* complements them by targeting the indexing done for accessing the value of a hierarchical parent within the block that contains it. The objective is to find an $\mathcal{O}(1)$ method for calculating the index of the parent in its block based on the index of the child relative to the child's block. The index of the child as shown in Listing 4.1 is k. The relation between k and the vectors i and l is:

$$k = \sum_{u=0}^{d-2} i_u \cdot 2^{s_u} + i_{d-1} \tag{4.3}$$

where $s_u := \sum_{v=u+1}^{d-1} l_v, \forall u \in \{0, \ldots, d-2\}$. Let *lk* be the index of the left parent relative to its block. Taking into account that only the t-th component is different between l and *ll*, and between i and li, the equation used for calculating lk based on k is:

$$lk = (k/2^{l_t+s_t}) \cdot 2^{ll_t+s_t} + li_t \cdot 2^{s_t} + k\%2^{s_t}. \tag{4.4}$$

Using the notation rk to denote the index of the right hierarchical parent *(rl, ri)* relative to its block, its equation is obtained from the equation for lk by replacing all the occurences of ll_t and li_t with rl_t and ri_t respectively. Consequently, *hopt3* in combination with *hopt2* and *hopt1* reduce the complexity of *gp2idx* and *idx2gp* in the innermost loop of Listing 4.1 to $\mathcal{O}(1)$.

Finally, the fourth sequential optimization, *hopt4*, performs a loop interchange by transforming the sequence of loops from *(t, g, b, k)* to *(t, b, g, k)*. The benefit of such a permutation is two-fold: First, the reuse of the *parIdx* array created as part of *hopt2* increases, and second, locality at cache level is also improved. Without loop interchange, *parIdx* is not used at its full potential, meaning that it is created for a block in the group g, it is used by that block, and immediately discarded when moving to the next block in group g. The permutation optimization allows for *parIdx* to be used by all the hierarchical parents of every child grid point in group $n-1$. Cache efficiency can be explained following the same reasoning as for *parIdx*.

The optimized serial version of multi-dimensional hierarchization is shown in Listing 4.2. The mapping between optimizations and line numbers is the following:

- Line 3 results from the loop invariant code motion done in *hopt1*.
- *hopt2* introduces lines 4 - 8 where *parIdx* is computed. *parIdx* is then used in lines 21 and 28 to reduce the number of integer operations executed when accessing dependencies.
- Lines 9 - 11, 14, 18 - 21, and 25 - 28 are created as part of *hopt3*. The access to dependencies is further reduced, resulting at this point in $\mathcal{O}(1)$ complexity.
- *hopt4* swaps the loops b and g in Listing 4.1, leading to the sequence *(t, b, g, k)*.

The unoptimized hierarchization from Listing 3.4 does not provide possibilities for parallelization because of data dependencies which are not explicitly shown in that version. One has to avoid updating the hierarchical parents before their children. The situation changes radically in Listing 4.1 where parallelization is realized by distributing the interations of the b loop, e.g. across worker threads in OpenMP. A barrier is necessary after the execution of each iteration in the g loop in order to eliminate write-after-read hazards. Finally, the optimized hierarchization shown in Listing 4.2 is also parallelized based on decomposing the work in the b loop. This time, a barrier is used for synchronization after the execution of each iteration in the t loop resulting in less synchronization compared to the parallel unoptimized hierarchization based on Listing 4.1.

Listing 4.2 Non-recursive multi-dimensional sparse grid hierarchization using the bijection. **Input**: *d*, *numGridPoints*, *sg1d*[*numGridPoints*]. **Output**: *sg1d*[*numGridPoints*].

```
for t = 0 to d - 1 do
  for b = a(d, n - 1) - 1 downto 0 do
    l = invPos(n - 1, b)

    lt = l[t]
    for g = lt downto 0 do
      l[t] = g
      parIdx[g] = groupIdx[n - 1 - lt + g] + pos(l) * 2^(n - 1 - lt + g)
    l[t] = lt

    postfixSum = 0
    for t0 = t + 1 to d - 1 do
      postfixSum = postfixSum + l[t0]

    for g = l[t] downto 0 do
      for k = 2^(n - 1 - l[t] + g) - 1 downto 0 do
        it = (k / 2^postfixSum) % 2^g
        (llt, lit) = leftParent1d(g, it)
        (rlt, rit) = rightParent1d(g, it)

        if llt != 0 then
          lk = (k / 2^(g + postfixSum)) * 2^(llt + postfixSum) +
               lit * 2^postfixSum +
               k % 2^postfixSum
          lv = sg1d[parIdx[llt] + lk]
        else
          lv = 0

        if rlt != 0 then
          rk = (k / 2^(g + postfixSum)) * 2^(rlt + postfixSum) +
               rit * 2^postfixSum +
               k % 2^postfixSum
          rv = sg1d[parIdx[rlt] + rk]
       else
          rv = 0

       sg1d[parIdx[g] + k] = sg1d[parIdx[g] + k] - (lv + rv) / 2
```

Listing 4.3 Non-recursive multi-dimensional sparse grid interpolation using the bijection.
Input: d, n, $a[d][n]$, *sg1d*[], m, $x[m][d]$. **Output**: $r[m]$.

```
1:  for j = 0 to m - 1 do
2:    r[j] = 0

3:  idx12 = 0
4:  for g = 0 to n - 1 do
5:    for b = 0 to a[d][g] - 1 do
6:      l = invPos(g, b)
7       for j = 0 to m - 1 do
8:        idx3 = 0
9:        p = 1
10:       for t = 0 to d - 1 do
11:         idx3 = idx3 * 2^l[t] + floor(2^l[t] * x[j][t])
12:         p = p * basis1d(l[t], x[j][t])
13:       r[j] = r[j] + sg1d[idx12 + idx3] * p
14:     idx12 = idx12 + 2^g
```

4.3.2 Sparse Grid Interpolation on CPUs

The reference algorithm for the *sginterp* kernel is shown in Listing 3.5. *sginterp* is in general computationally bound. Its computational intensity is strongly influenced by the input parameter d which controls the number of iterations in the innermost loop t: The larger d, the more computationally bound the interpolation algorithm.

A first transformation, referred to as *iopt1* and applied to the reference interpolation algorithm is loop interchange. The resulting algorithm is shown in Listing 4.3. Loop interchange modifies the sequence of loops in the nest from *(j, g, b, t)* to *(g, b, j, t)*. There are two main advantages of this transformation. The first one is related to the number of the integer operations executed as part of the *invPos* invocation. Without loop permutation, *invPos* is invoked m times for every block. After applying the permutation, *invPos* is executed only once per block and its result, l, is reused m times. The second benefit of the permutation is about cache reuse. For a large number of interpolation points, the x matrix containing their coordinates may dominate in size a block. This means that there is a high probability that the value from a block needed by an interpolation point in line 13 is already loaded in the cache because of a previous interpolation point whose corresponding value from the block maps to the same cache line. This is not possible using the first algorithm from Listing 3.5 for which all the blocks are accessed for one interpolation point before moving to the next interpolation point. Therefore, the data loaded from the first blocks is evicted from the cache and cannot be reused for the next interpolation point.

Another optimization, *iopt2*, applied to Listing 4.3 is the vectorization of the j loop. The objective is for the SIMD unit of the CPU to operate simultaneously on 4 (SSE) or 8 (AVX) interpolations points. The theoretical speedup resulting from this optimization is 4 / 8 for SSE / AVX. Compilers generally fail to vectorize the j loop mainly because the access to the interpolation points is not unit-stride and the loop is not the innermost loop in the nest [62].

The unit-stride access requirement is solved by transforming the x matrix (stride-d access when vectorizing) to a more vector friendly $\tilde{x}$ based on the equation:

$$x[j][t] = \tilde{x}[(j/4) \cdot d + t][j\%4], \tag{4.5}$$

where the number 4 assumes SSE vectorization of single precision floating point operations. For AVX vectorization, 4 is replaced with 8. Using $\tilde{x}$, the access to the interpolation points in the innermost loop t becomes unit-stride. For more efficiency, $\tilde{x}$ is aligned to a 128-bit / 256-bit address boundary for SSE / AVX. Besides the conversion of x to $\tilde{x}$, vectorization also requires that the scalar variables p and *idx3* become arrays of size 4 / 8 for SSE / AVX.

The second requirement for vectorization (the vectorized loop must be innermost) can be addressed by manually applying the following sequence of loop transformations:

1. strip mining the j loop resulting in the loops $j0$ and $j1$
2. applying loop distribution to loop $j1$ resulting in loop $j10$ iterating over lines 8 - 9 from Listing 4.3, loop $j11$ for lines 10 - 12, and loop $j12$ for line 13
3. permuting the $j11$ and k loops so that loop $j11$ becomes innermost.

Vectorization can then be automatically applied to the innermost loops, especially $j10$ and $j11$. A stronger method to enforce vectorization independent from the compiler is based on intrinsics for SSE [63] and AVX [64]. However, the disadvantage is that the optimized code is considerably more complex and must be modified in order to address slightly different usage scenarios such as: using double precision instead of single precision floating point numbers and using 64-bit integers instead of 32-bit integers.

There are two sources of parallelism in Listing 4.3 resulting from distributing the interpolation points across worker threads or distributing the blocks into which the sparse grid is decomposed. In the first case, a subset of the interpolations points is assigned to each thread. The threads then interpolate independently, making this parallelization approach embarrassingly parallel. The disadvantage is exposed for a small number of interpolation points, e.g. 1, adding more cores does not help to accelerate the execution. This undesired situation is addressed by distributing the sparse grid blocks among worker threads, meaning that each thread computes the contribution of its assigned blocks to all the interpolation results (at all

Listing 4.4 Sparse vector - matrix multiplication.
Input: m, p, $ix[m]$, $x[m]$, $a[][p]$. **Output**: $y[p]$

```
for j = 0 to p - 1 do
  y[j] = 0

for j0 = 0 to p - 1, step bs do
  for i = 0 to m - 1 do
    for j1 = j0 to min(j0 + bs - 1, p - 1) do
      y[j1] = y[j1] + x[i] * a[ix[i]][j1]
```

the points). Reduction is necessary at the end of the computation in order to sum up the intermediary interpolation results from all the threads. Since in general, the number of interpolation points is reasonably large, the first approach is preferred because it does not require reduction.

4.3.3 Sparse Vector - Matrix Multiplication on CPUs

The *spvm* kernel from Listing 3.8 is memory bound because of its low computational intensity. More precisely, the algorithm performs one multiplication and one addition per two memory accesses, resulting in a computational intensity of 1 flop per memory reference.

Listing 4.4 shows the optimized version *spvm* including a loop tiling transformation, *sopt1*, applied to the j loop of the initial algorithm. In the reference *spvm*, the output vector y cannot be cached and reused across the rows of the a matrix assuming that the y vector is considerably larger than the cache. On the other hand, the central concept in Listing 4.4 is to tile a and y so that each tile of y fits in the cache and stays there during the traversal of all the rows of a. The benefit is improved locality since the data transferred from memory is decreased by up to a factor of 2, resulting in a computational intensity of 2 flop per memory reference. However, this does not necessarily lead to 2x more performance because of the indirect access to a via the vector of integers *ix*. The *ix* vector contains the indices of the rows of a that are multiplied with values from x and added to the output vector y. Although tiling improves the reuse of y, it also amplifies the negative effect on performance of the unpredictable jumps between rows of a using *ix*, i.e. the more tiles are used, the more jumps are made. In order to solve this trade-off, an empirical method is employed in which the performance for different tile sizes is measured and the tile size corresponding to the best performance is returned.

A second optimization, *sopt2*, for *spvm* is the vectorization of the $j1$ loop. In order to improve the performance resulting from vectorization, a and y are both aligned to 16 / 32 byte address boundaries for SSE / AVX. Moreover, padding is applied to all the rows of a in order to ensure that each row satisfies the alignment requirement for efficient vectorization.

Similarly to *sginterp*, there are two sources of parallelism in *spvm*:

1. The columns of y and the corresponding columns of a are distributed among worker threads. This approach is embarrassingly parallel.

2. The rows of a are distributed across threads and each thread computes its own copy of y. A reduction is necessary at the end in order to sum up the individual copies of y.

Assuming that the size of y, i.e. p, is large enough to provide work for all the processor cores in a system, the first approach is the best as it does not require reduction. However, if the parallelism is not enough, then the second decomposition scheme can be used to create more parallel work at the cost of more synchronization overhead.

Since *spvm* is memory bound, its performance depends strongly on the memory bandwidth of a system. Therefore, it makes sense to discuss *spvm* in the context of Non-Uniform Memory Access (NUMA) systems. A NUMA system is a shared memory system composed of multiple interconnected NUMA nodes, where a node contain at least one CPU and memory. A memory access is said to be local if both the CPU core initiating the access and the memory serving the request belong to the same NUMA node. A remote memory access refers to a data access initiated from a NUMA node to the memory of another node. Compared to a local access, the cost of a remote access is much higher. The main advantage of a NUMA system is that the theoretical memory bandwidth scales linearly with the number of NUMA nodes. However, this property is generally not directly transferred to applications unless they include optimizations that minimize the number of remote memory accesses.

Assuming a NUMA system with nt cores, an implementation of *spvm* optimized with respect to NUMA, is based on slicing the a matrix vertically in nt equal submatrices. Each submatrix is assigned to a worker thread. The thread then invokes the serial version of *spvm* using its submatrix. In order to ensure that a thread and its paired submatrix are placed on the same NUMA node, the following steps are executed:

1. Threads are pinned to CPU cores so that there is one thread per core.

2. Each thread allocates memory space for its submatrix and copies its corresponding slice from a to it. The first touch policy ensures that the submatrix is placed in the local memory of the NUMA node that contains the CPU core to which the thread is pinned.

3. Each thread computes a slice from y by invoking *spvm* on its local submatrix.

The performance of the NUMA aware *spvm* is affected by the initial overhead resulting from copying slices from a to the local memories of each NUMA node. However, in computational steering, the distribution of a across NUMA nodes is reused for multiple calls to *spvm* using different vectors x and *ix*. This means that the costs associated with changing the data layout are amortized, making the optimization worthwhile.

4.4 Optimizations for GPUs

This section describes the porting of *sghierarch*, *sginterp*, and *spvm* to GPUs. Aspects such as GPU specific parallelization and optimizations are covered in detail.

4.4.1 Sparse Grid Hierarchization on GPUs

The GPU version of *sghierarch* is based on Listing 4.2. Similarly to the parallelization for multi-core CPUs, the GPU parallelization is based on distributing the iterations of the b loop among thread blocks. The first thread from each thread block is designated as the master thread for its thread block. The master executes lines 3 - 8 where the arrays l and *parIdx* are computed. These arrays are shared by an entire thread block. The concurrent access to them is handled using a barrier inserted in the code using the *_syncthreads* function. The barrier ensures that the threads do not access sparse grid values until l and *parIdx* are computed by the master. A second barrier is necessary after each iteration of the g loop in order to eliminate write-after-read hazards in which the hierarchical parents (dependencies) are updated before their children. Each thread identified by *threadIdx.x* within a thread block of size *blockDim.x* updates only the values whose index relative to the sparse grid block containing them, *idx3*, satisfies the equation:

$$threadIdx.x = idx3\%blockDim.x. \tag{4.6}$$

By doing so, the read and write accesses to the *sg1d* array stored in global memory are coalesced.

An important aspect of the GPU implementation of *sghierarch* is the mapping between data structures and the memories of the GPU. As already mentioned, the linear representation of the sparse grid, *sg1d*, is stored in global memory. At the foundation of *invPos*, there is a matrix a of size $d \cdot n$ from Eq. 3.15. Since a is read-only and rather small, e.g. 100 integer values, it is placed in constant memory whose L1 cache has the lowest latency among all the memories on the GPU, except registers [53]. Taking into account that the arrays l and *parIdx* are accessed by all the threads in a thread block, they are placed in shared memory, a fast on-chip memory through which threads in the same thread block can communicate.

By storing only l and *parIdx* in shared memory for each thread block, the shared memory consumption is rather low. Therefore, it makes sense to configure the shared memory so that only 16 KB are used for the explicitly controlled memory where l and *parIdx* are stored, and 48 KB are used for the L1 cache. By increasing the size of the L1 cache, the number of conflict cache misses is reduced and the reuse of values corresponding to hierarchical parents is improved, thus resulting in a faster access to data dependencies.

4.4.2 Sparse Grid Interpolation on GPUs

The parallelization of *sginterp* on GPUs is based on distributing the set of interpolation points among threads, i.e. each GPU thread interpolates the sparse grid at one point. Thus, the code executed by every thread for its assigned interpolation point is based on Listing 4.3 in which m is replaced with 1, meaning that the j loop disappears. Similarly to *sghierarch* for GPUs, the first thread from every thread block is designated as the master thread responsible for computing the l vector for the entire thread block by invoking *invPos*. Synchronization is necessary to protect l since the concurrent access pattern for l corresponds to *one writer multiple readers*, i.e. the master writes to l, then all the threads in the thread block use l in the computation from lines 8 - 12. In order to address this concurrency related aspect, barriers are used so that the master thread waits at a barrier until all the threads no longer need the current l, and the threads wait at another barrier until the master updates l.

On GPUs, an important optimization direction is to use the different memories efficiently. As discussed in the context of *sghierarch*, *invPos* is based on accessing a matrix a of size $d \cdot n$. Since this matrix is read-only, it is stored in constant memory, a fast cached memory. The l vector shared by an entire thread block is placed in shared memory. Consider that the size of a thread block is *blockDim.x* which represents also the number of interpolation points handled by a thread block. There are two possibilities for storing the subset of interpolation points allocated per thread block: in shared memory or in registers.

First, the subset of interpolation points per thread block is placed in shared memory. The copy of the subset from global memory to shared memory is performed using coalesced load accesses. This is achieved by involving all the threads from the thread block in the copy operation and by ensuring that each thread accesses global memory using a stride of size *blockDim.x*. If a standard copy is done between global and shared memory (the same as *memcpy*), then the d-dimensional points are stored in shared memory consecutively. Using such a data layout, there is a high risk that the accesses to the subset of points from shared memory generate bank conflicts. In order to understand this problem, one needs to remember that the shared memory is divided into 32 banks and successive 32-bit words map to consecutive banks. For optimal performance, the shared memory accesses from a warp have to be distributed evenly among all the banks or have to refer to the same address from one bank. Whenever the size of an interpolation point, or d, is even, bank conflicts occur. Intuitively, this problem can be explained using an extreme example. For $d = 32$, the components at position $t \in \{0, \ldots, d-1\}$

of all the interpolation points for a thread block are stored in the same bank. Consequently, all the accesses to the points are serialized, thus severely reducing the performance. In fact, slowdown factors of different sizes can be seen whenever d is even.

A solution to the bank conflict problem is to pad each interpolation point so that the distance measured in banks between any two successive interpolation points is an odd number. More precisely, this is done by allocating $d+1$ values in shared memory for every interpolation point whenever d is even. Let wx be a 2d array with 32 rows and d columns representing the interpolation points that are assigned to a warp and are stored in shared memory. Without loss of generality, assume that the starting address of wx in shared memory corresponds to bank 0. The index of the bank where $wx[j][t]$ resides is then given by $(j \cdot d + t)\%32$, where 32 represents the number of shared memory banks. It is straightforward to prove that for any odd d, there cannot be two numbers, $u, v \in \{0, \ldots, 31\}$ and $u \neq v$, so that $(u \cdot d)\%32 = (v \cdot d)\%32$. This translates to the fact that an odd d eliminates the possibility that any two rows of wx start at the same bank. Furthermore, since all the threads in a warp access the columns of wx in the same order (the t loop from Listing 4.3), the distance d between values accessed by successive threads is preserved. Thus, there are no bank conflicts generated by any of the accesses to wx.

The main disadvantage of padding is that it increases shared memory consumption. In fact, the smaller an even d, the higher the shared memory requirements. For $d = 2$, approximatively 1.5 times more shared memory is used compared to the case without padding. On the other hand, for $d = 10$, padding accounts for approximatively 9% of shared memory consumption. For GPUs, the consumption of shared memory is inversely proportional to the number of concurrent threads that can be executed on one GPU core. A low number of concurrent threads per core limits the ability of multithreading to reduce the instruction pipeline latency, e.g. resulting from loads and stores to global memory.

There is another solution for eliminating bank conflicts that does not suffer from the limitations of padding, i.e. it does not increase shared memory consumption. Let bx be a 2d array stored in shared memory. It contains the interpolation points for a thread block. Let gx be the part from x paired with bx. gx is stored in global memory. The same data layout modification required by SSE / AVX is also applicable here and is based on the equation:

$$gx[j][t] = bx[(j/32) \cdot d + t][j\%32]. \tag{4.7}$$

By using bx, the access to a component t of an interpolation point is served by the bank identified by *off*+ *threadIdx.x*%32, where *off* is the bank where bx starts in shared memory and *threadIdx.x* is the identifier of a thread relative to its thread block. The result of *threadIdx.x*%32 is the identifier of a thread within its respective warp. Consequently, all the threads in a warp accessing the t-th component of their interpolation points are served by different banks.

The register file on each GPU core is rather large, i.e. it contains 32K 32-bit registers. It is obviously the fastest memory of the GPU and because of its size, it can even store more values than shared memory. A limitation of registers is that they cannot be used to share data among threads as opposed to shared memory. However, since an interpolation point is private per thread, it makes sense to consider a set of registers as a storage solution for an interpolation point. A brute force method to achieve this is by replacing all the components of an interpolation point with variables. The disadvantage is that one has to implement multiple versions of *sginterp* for different values of d, i.e. the size of an interpolation point. A better approach is to use an array to store the interpolation point per thread and to provide the compiler with information necessary for handling the array using registers. Let xs be an array of size d containing the interpolation point assigned to a thread. In order to enable the compiler to store / use xs to / from registers, the following requirements need to be met:

1. The size of xs must be known at compile time. This is done by declaring the array using a size represented by a constant positive integer.

2. All the loops where xs is indexed using the loop variable must be completely unrolled so that subsequently, xs's components are addressed using numerical constants. In CUDA, complete unrolling is achieved by inserting the directive *#pragma unroll* before a loop.

Provided that these conditions are fulfilled, multiple optimized GPU versions of *sginterp* using registers can be trivially generated by specializing for different values of d, e.g. for d from 1 to 10, using C++ templates for which d is a parameter.

4.4.3 Sparse Vector - Matrix Multiplication on GPUs

Porting *spvm* to GPUs is achieved with little effort by assigning to each thread the computation of one value from the output vector y. Since the *spvm* kernel has low computational intensity, an observation is necessary: In computational steering, the initial cost of transferring the matrix a from Listing 4.4 is amortized by a large number of invocations of the *spvm* version for GPUs. Therefore, a more important aspect is the transfer of the input vectors x and ix from the CPU to the GPU, and the output vector y from the GPU to the CPU. In order to accelerate the data copy, page pinned memory is used for allocating these vectors on the CPU side. In general, data is transferred from / to the GPU using DMA in two steps. For instance, in the case of a data copy from the CPU to the GPU, the data is first copied to a memory region where DMA can be used, then the data is copied from that region to GPU's memory. Using page pinned memory, the first step is no longer necessary, thus resulting in better performance.

4.5 Evaluation

This section presents the performance numbers resulting from the optimizations detailed in this chapter. The goal is to indicate the most suitable processor, CPU or GPU, in a given heterogeneous system for all the computational kernels, i.e. *sghierarch*, *sginterp*, and *spvm.*

4.5.1 Experimental Setup

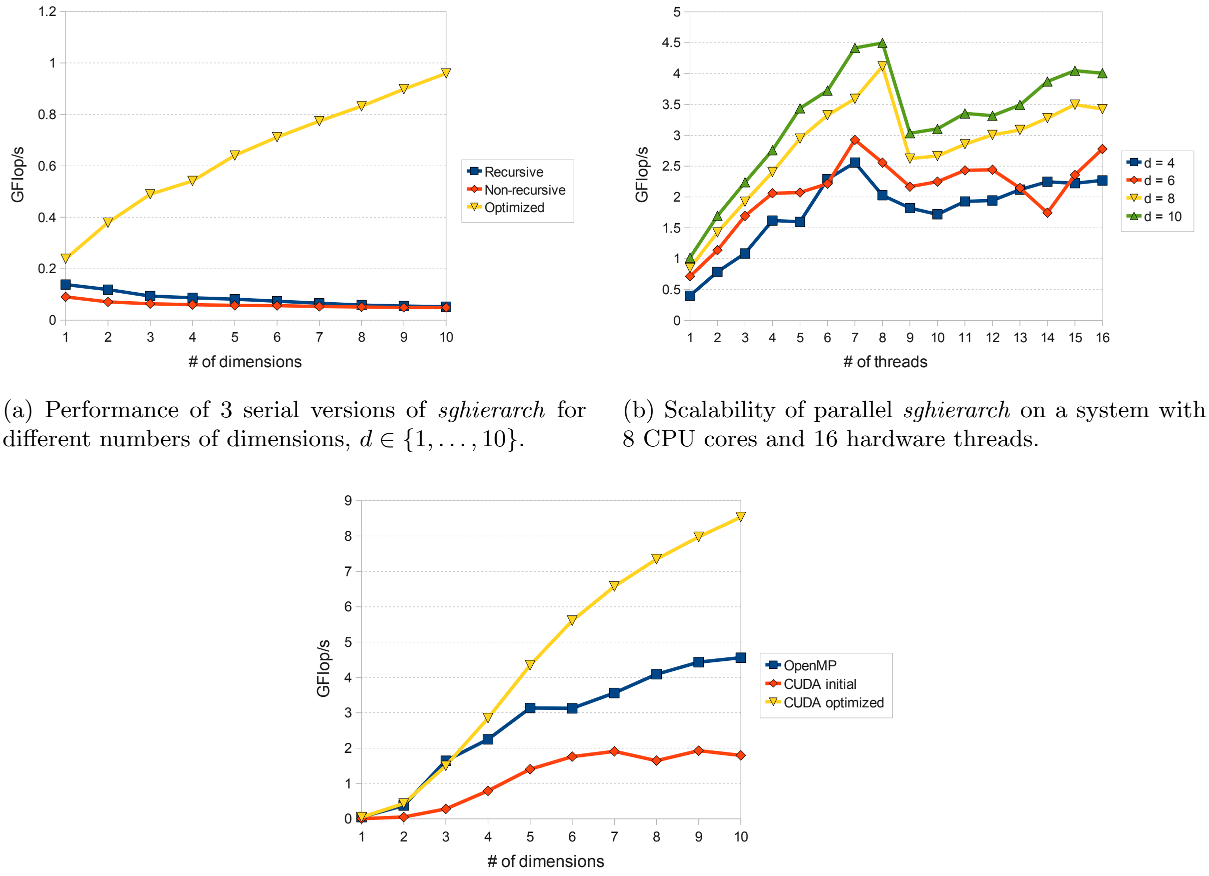
The measurement data corresponds to a dual-socket system with one Quad-core Intel Nehalem E5630 CPU per socket. Each CPU is clocked at 2.53 GHz and supports 8 hardware threads via Hyperthreading. The CPU part is complemented by an Nvidia Quadro 6000 GPU which operates at 1.15 GHz, has 16 cores with 32 SIMD lanes per core.

On the software side, the compilation is done using gcc 4.4.5. For the compilation of the GPU code, CUDA 4.2 is used. All the kernels operate with single precision floating point numbers. Page pinned memory is used on the CPU for allocating the data structures for all the computational kernels. The measurements include data transfers over PCIe according to the requirements of the computational steering application described in Chapter 3:

- The execution time for *sghierarch* includes the transfer of the sparse grid's linear representation, *sg1d* from the CPU to the GPU, and vice versa.
- The execution time for *sginterp* includes the transfer of the set of interpolation points x from the CPU to the GPU and of the interpolation results r from the GPU to the CPU.
- The execution time for *spvm* includes the transfer of the input vectors x and ix to the GPU and of the output vector y from the GPU.

4.5.2 Sparse Grid Hierarchization

Fig. 4.1a depicts the sequential performance of *sghierarch* on the CPU. In this section, the refinement level of all the sparse grids is 10. Three versions of *sghierarch* are displayed in the graph: the recursive version, the non-recursive version, and the optimized implementation. One can see than the recursive version and the non-recursive version are almost on par. For smaller values of d, the recursive *sghierarch* is faster than the non-recursive *sghierarch.* This happens because the recursive version requires only half of the number of read accesses to the sparse grid (and half of the invocations to the bijection *idx2gp*) compared to the non-recursive version. The smaller number of calls results from the use of a stack which allows for a top-down traversal of the sparse grid. During the traversal, the value of each hierarchical parent is retrieved only once and is propagated through the stack for updating all its children. A notable point is that



(a) Performance of 3 serial versions of *sghierarch* for different numbers of dimensions, $d \in \{1, \ldots, 10\}$.

(b) Scalability of parallel *sghierarch* on a system with 8 CPU cores and 16 hardware threads.

(c) GFlop/s rate of 2 CUDA versions of *sghierarch* compared to the OpenMP version (8 threads) for different number of dimensions, $d \in \{1, \ldots, 10\}$.

Figure 4.1: Performance of sparse grid hierarchization, *sghierarch*, on a heterogeneous system.

the recursive version is considerably more difficult to optimize. The optimized *sghierarch* is up to 18.4x faster than the non-recursive *sghierarch*. The graph also shows a performance gap between the optimized *sghierarch* and the other two versions which expands with the number of dimensions, meaning that the 18.4x speedup further increases for more than 10 dimensions.

Fig. 4.1b is the scalability graph for *sghierarch*, more precisely the GFlop/s rate depending on the number of threads. In the performance tests, the threads are mapped to the CPU cores of the system according to the next scheme: For a number of threads between 1 and 8, the threads are assigned one per core, whereas for more than 8 threads, Hyperthreading is evaluated by pinning 2 threads per core. The best speedup relative to the sequential version is 4.4x and is obtained for 8 hardware threads. Hyperthreading, i.e. using 16 threads, is not beneficial in this case, i.e. for $d = 10$, it reduces the speedup from 4.4x to 3.9x.

In Fig. 4.1c, the CUDA version of *sghierarch* is compared to the OpenMP version for different numbers of dimensions. *CUDA initial* corresponds to the non-recursive algorithm from Listing 4.1 while *CUDA optimized* is the CUDA implementation of Listing 4.2. The performance difference between the optimized CUDA version and the OpenMP version varies depending on the number of dimensions, i.e. for $d \leq 3$, the GPU is on par with the CPU and for d between 4 and 10, the speedup of the GPU relative to the CPU increases from 1.3x to 1.9x.

4.5.3 Sparse Grid Interpolation

Fig. 4.2a shows the sequential performance for 4 versions of *sginterp*: *recursive*, *non-recursive*, a version optimized for cache efficiency (*cache opt*), and a version vectorized using SSE intrinsics (*vector opt*). The number of interpolation points is here 10^4. The vectorized *sginterp* is up to 12.6x faster than the recursive implementation.

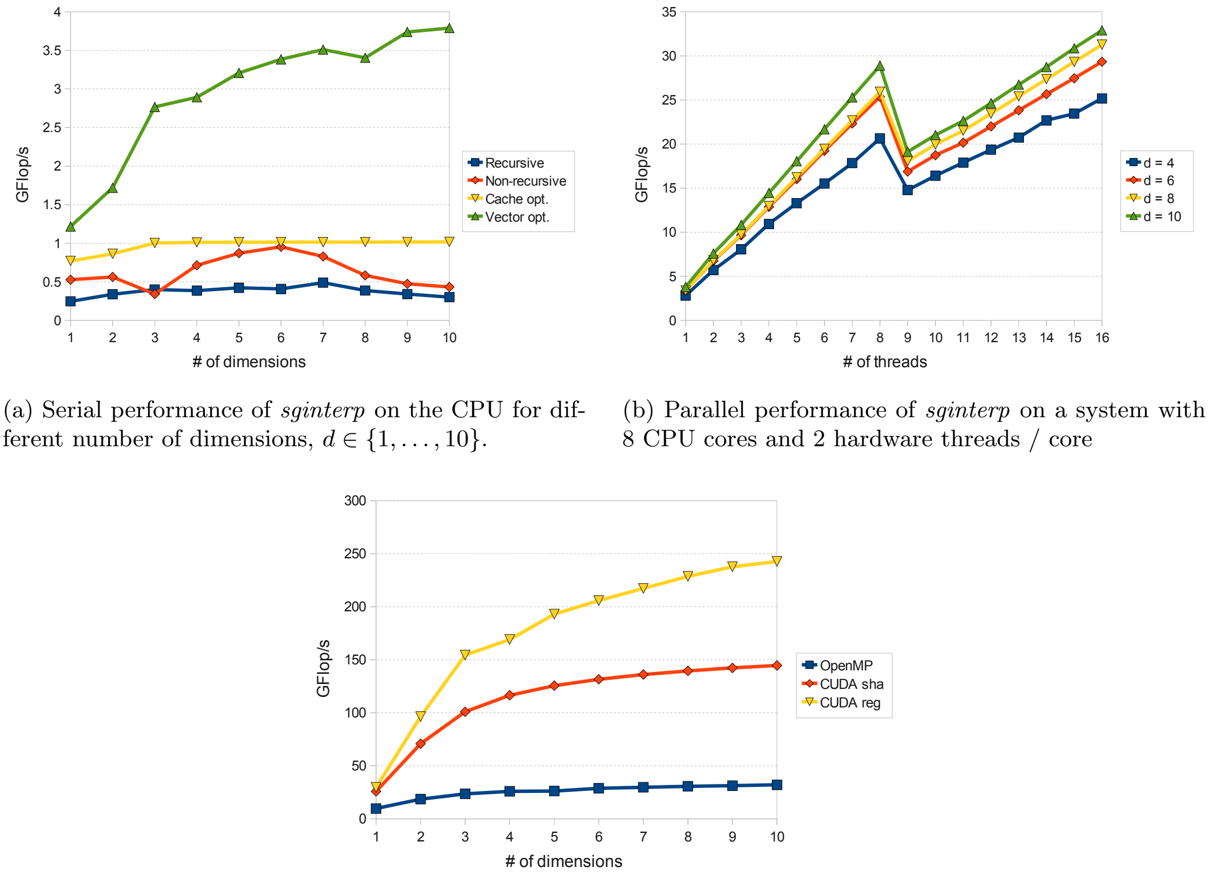
The scalability of the OpenMP version of *sginterp* is shown in Fig. 4.2b for $16 \cdot 10^4$ interpolation points. For $d = 10$, the speedup relative to the sequential version is 7.6x and 9.7x for 8 and 16 threads respectively. Consequently, in contrast to *sghierarch*, Hyperthreading has a positive effect on the GFlop/s rate of *sginterp*, meaning that there are pipeline stalls that cannot be addressed effectively by the compiler. As it can be seen in the graph, the benefit is consistent across all the tested numbers of dimensions, i.e. $d \in \{4, 6, 8, 10\}$.

Fig. 4.2c depicts the performance of *sginterp* on the GPU compared to the OpenMP version executed using 16 threads. The number of interpolation points is $2 \cdot 10^6$. In the *CUDA sha* version, the set of interpolation points is stored in shared memory whereas in *CUDA regs*, the interpolation points are placed in the register file. For $d = 10$, the speedup relative to the performance obtained on the CPU is 4.5x and 7.6x for *CUDA sha* and *CUDA regs* respectively. *CUDA regs* has the disadvantage that for $d \in \{1, \ldots, 10\}$, 10 versions of *sginterp* for the different values of d are created, e.g. at compile time based on C++ templates.

4.5.4 Sparse Vector - Matrix Multiplication

Fig. 4.3 shows the sequential performance of *spvm*. The input matrix a has 250 rows and 10^6 columns, thus encapsulating the behavior of a computational steering application in which the number of rows significantly smaller than the number of columns. In the figure, the vectorized version of *spvm* is up to 2.9x faster than the initial unoptimized *spvm*.

The scalability results for *spvm* are presented in Fig. 4.3b. *OpenMP initial* corresponds to the parallelization of *spvm* without the optimization for improving the locality at NUMA node level. *OpenMP numa opt* includes this optimization. The speedup of *OpenMP numa opt* relative to the sequential performance is 3.5x obtained for 8 and 16 threads, meaning that



(a) Serial performance of *sginterp* on the CPU for different number of dimensions, $d \in \{1, \dots, 10\}$.

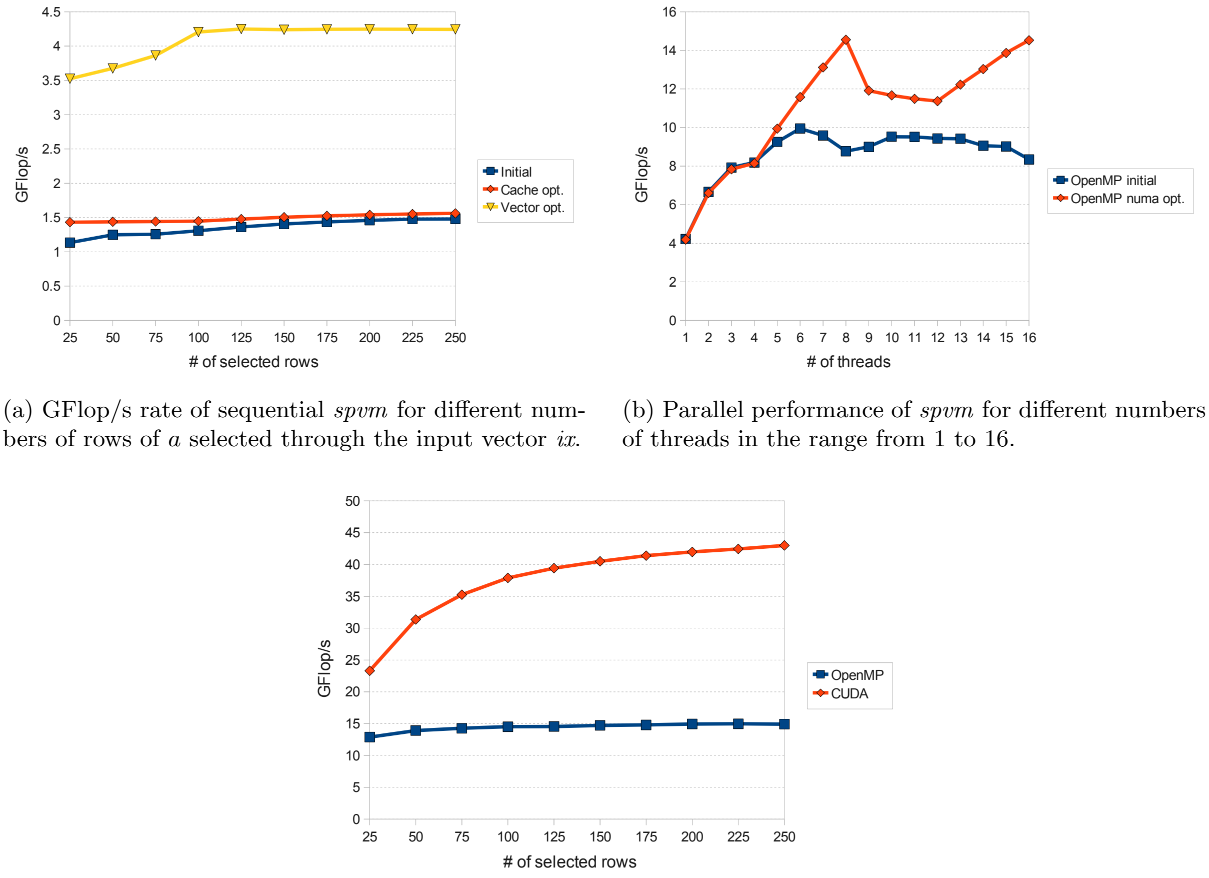
(b) Parallel performance of *sginterp* on a system with 8 CPU cores and 2 hardware threads / core

(c) GFlop/s rate of 2 CUDA versions of *sginterp* compared to the OpenMP version (16 threads) for different number of dimensions, $d \in \{1, \dots, 10\}$.

Figure 4.2: Performance of sparse grid interpolation, *sginterp*, on a heterogeneous system.

Hyperthreading has no effect here. In contrast, *OpenMP initial* is only 2.4x obtained for a non-intuitive number of threads, i.e. 6. For 8 threads, *OpenMP initial* provides a speedup of 2.1x, thus being slower than 6 threads. Notice that between 1 and 4 threads, the two OpenMP versions of *spvm* deliver the same performance. However, starting with 5 threads, the cores on the second NUMA node of the system are used and the difference in terms of performance between *OpenMP initial* and *OpenMP numa opt* is obvious. This happens because *OpenMP initial* generates considerably more remote memory accesses than *OpenMP numa opt*.

Fig. 4.3c depicts the GFlop/s rate of the CUDA version of *spvm* and the GFlop/s rate of the OpenMP version. For *spvm*, the GPU is between 1.8x and 2.9x faster than the CPU.



(a) GFlop/s rate of sequential *spvm* for different numbers of rows of a selected through the input vector ix.

(b) Parallel performance of *spvm* for different numbers of threads in the range from 1 to 16.

(c) GFlop/s of the CUDA version of *spvm* versus OpenMP (16 threads) for different numbers of selected rows of a (or non-zeros in the sparse input vector x).

Figure 4.3: Performance of *spvm* on a heterogeneous system.

4.6 Summary

This chapter describes optimizations for CPUs and GPUs applied to three computational kernels: sparse grid hierarchization (*sghierarch*), sparse grid interpolation (*sginterp*), and sparse vector - matrix multiplication (*spvm*). The kernels are extracted from the computational steering application described in Chapter 3. More precisely, they represent the performance hotspots of the application. Their processing behavior is diverse: *sghierarch* is integer bound, *sginterp* is computationally bound, whereas *spvm* is memory bound.

For *sghierarch*, the CPU optimizations concentrate on reducing the number of integer operations and improving the locality whereas on the GPU, the tuning strategy is based on finding the best matching between data structures and the GPU's memories, i.e. shared memory, L1 cache, constant memory. On the CPU, *sginterp* is optimized with respect to cache reuse and vector units. An important point here is the use of a vector friendly layout for the set of interpolation points applicable to both the CPU and the GPU. On the GPU, the interpolation points can either be stored in shared memory or in registers. Similarly to the optimized *sginterp*, the tuned *spvm* for the CPU includes optimizations for cache reuse and vector units. The OpenMP version of *spvm* is NUMA aware and reduces the number of remote memory accesses.

On the CPU side, the speedup factors resulting from the sequential optimizations applied to *sghierarch*, *sginterp*, and *spvm* are 18.4x, 12.6x, and 2.9x respectively, compared to the initial unoptimized versions. All the OpenMP versions of the kernels scale on a NUMA system with 2 NUMA nodes and 4 cores (8 hardware threads) per node, i.e. the speedups relative to the serial optimized versions are 4.4x (*sghierarch*), 9.7x (*sginterp*), and 3.5x (*spvm*). The CUDA versions are 1.9x, 7.6x, and 2.9x faster than the OpenMP versions for the three kernels.

Chapter 5

An Empirical Optimization Method for GPUs

Given the diversity and complexity of current processors, considerable effort is invested in optimizing applications for each individual hardware. In order to achieve optimal performance, advanced knowledge about the hardware is needed but such information is often not available. Furthermore, tuning an application for a given hardware typically implies dealing with trade-offs difficult to analyze using purely theoretical methods. In order to cope with such challenges an empirical optimization method, also referred to as *search based auto-tuning*, aims at finding the optimal values for a set of optimization parameters usually exposed by a programmer for a given application. By doing so, auto-tuning shifts the weight from the programmer to an automatic search in a potentially high-dimensional space. This chapter describes optimization parameters for GPU programs. Additionally, reducing the auto-tuning time is another point tackled here. Although these aspects are studied in the context of the sparse grid technique, towards the end of the chapter their applicability to a wider range of applications is explained.

5.1 Introduction

Relying solely on compilers for optimizing programs rarely results in optimal performance. Even if an expert programmer manages to get the most from a specific processor, it often happens that the same percentage of the peak performance is not obtainable across different processors. This is called the performance portability problem. The empirical optimization of applications or search based auto-tuning, alleviates this problem to some extent. It implies generating and evaluating code variants corresponding to different values assigned to a set of optimization parameters exported by the application to a search engine. Optimization parameters may

Listing 5.1 View on typical central loop of auto-tuning.

```
Searcher searcher(searchSpace)
while searcher.notFinished() do
  nextPoint = searcher.next()
  metricValue = compileExecuteMeasure(nextPoint)
  searcher.update(nextPoint, metricValue)
return searcher.optimum()
```

control the data structure or algorithm to use, or may refer to loop transformations, e.g. loop tile factor, loop unroll factor, loop interchange, etc. The objective is to identify the variant that provides the best performance. In this way, less knowledge about the underlying hardware is needed. Trade-offs difficult to analyze statically can also be managed efficiently using auto-tuning. Moreover, auto-tuning can help to achieve performance portability, allowing for an easier fitting between software and hardware. Despite its name, auto-tuning is not in general completely automatic and requires the programmer to manually define optimization parameters that are worthwhile to be considered for the search. In most auto-tuning solutions, the main loop of auto-tuning is similar to the one displayed in Listing 5.1.

Auto-tuning is at the foundation of several high-performance numerical libraries including: *ATLAS* [65] for dense linear algebra, *OSKI* [66] for sparse linear algebra, *FFTW* [67] for Fast Fourier Transforms (FFT), and *SPIRAL* [68] for signal processing. Frameworks have also been developed in order to make auto-tuning more general-purpose. They aim at simplifying the method for parameterizing a given code. Since for numerical codes in particular, loop nests are the central point for improving the performance, some frameworks [69, 70, 71] aim at providing a fine control over loop transformations, e.g. they are flexible source-to-source compilers. On the other hand, other frameworks address the problem of searching the optimum in unstructured and potentially high-dimensional spaces resulting from the definitions of the optimization parameters. In this case, brute-force search is impractical or even impossible. For this reason, stochastic (e.g. Simulated Annealing) or heuristic search methods (e.g. Nelder-Mead) are typically employed [72]. In addition to this, pruning the search space can also simplify the search.

A characteristic of auto-tuning is that a considerable amount of time is often needed to measure the performance of a code variant. Consider a generic objective function f that executes in a few cycles (no memory references). On processors clocked at frequencies in the order of 1GHz, minimizing f across a space of 10^9 points is rather cheap in terms of time, e.g. 1s. In the auto-tuning context, let g be the execution time which depends on a set of optimization parameters. Evaluating g at one point generally implies compiling a code variant and executing it. Assuming the same cardinality of the search space as before, i.e. 10^9, and only 1s for compilation and execution, a brute-force search needs 11574 days to finish. More powerful search algorithms are of great help here but even then it is advantageous to reduce the time necessary to evaluate g.

In this chapter, auto-tuning of GPU programs is studied, including a set of optimization parameters that can be seen as common to a wide set of GPU applications and that provide the means to control: the *thread block size*, the *thread granularity*, and the *amount of parallelism*. A technique called *input reduction* is then used to tune these parameters while ensuring that the duration of auto-tuning also drops significantly. Relative to *input reduction*, search space pruning and search algorithms are complementary concepts which are also covered in this chapter.

5.2 Existing Approaches for Auto-tuning

ATLAS [65], *OSKI* [66], *FFTW* [67], and *SPIRAL* [68] are notable libraries that make use of auto-tuning for improving their performance across different CPUs. With respect to GPUs, examples that validate auto-tuning include: matrix - matrix multiplication [73], sparse matrix - vector multiplication [25], stencil computation [22], and 3d FFT [20]. Motivated by the success of application centered auto-tuning, general purpose frameworks have emerged, their main goal being to provide more flexibility and simplicity to auto-tuning. These frameworks typically fall in one of the following categories:

1. tools that simplify the definition of optimization parameters, e.g. ranges and constraints, and provide search methods that approximate the optimum in a high-dimensional space

2. tools that provide source-to-source compilation functionality, i.e. the programmer specifies high-level parametrized loop transformations and the tools transform the code accordingly.

In the first category, *Atune-IL* [74] and *ISAT* [75] are pragma based auto-tuning solutions. They allow for a simple specification of the search space and automatize the generation and evaluation of code variants. Both are tools that provide only core functionality for auto-tuning, lacking the flexibility necessary for expressing constraints among different optimization parameters. Without constraints, pruning the search space is more limited. *Active Harmony* [76] includes a language for specifying optimization parameters and constraints among them. A major strength of this tool comes from the parallel search method called *Parallel Rank Order* (POR) built around the Nelder-Mead search algorithm. *Orio* [71] provides multiple methods of search including Nelder-Mead and Simulated Annealing. The search strategy used in Orio involves two stages: In a first stage, a global optimization method is used (e.g. Nelder-Mead) whereas the second stage refines the neighborhood of the optimum returned by the first stage using a local optimization method (e.g. hill climbing). Other features of Orio include the possibility to specify constraints and to execute auto-tuning across different input parameters.

The second category tools address the compilation aspect of auto-tuning by providing the means to control parameters of loop transformations, resulting in most cases in source-to-source compilation. These tools complement the functionality of the frameworks in the first category. *xlanguage* [69] is a C preprocessor controlled through special pragmas placed in the source code for interfacing loop transformations, e.g. loop tiling, loop unrolling, loop permutation. In order to obtain a complete auto-tuning solution, a search engine has to be connected to *xlanguage*. The engine iteratively updates the loop transformation parameters. A similar source-to-source compiler is *POET* [70] which includes a scripting language for parameterizing loop transformations. The interaction with POET is achieved through scripts which define the loops transformed during the tuning process. An external search engine is again required for enabling auto-tuning. Another example of a source-to-source compilation tool is *CHiLL* [77] which is used by Active Harmony to transform loops. Compiler techniques can also be applied and controlled using the *ROSE* compiler infrastructure (a library and associated tools) [78] as shown in [79] where auto-tuning is studied in the context of a stencil computation.

A notable framework that incorporates both loop transformations and search methods is described in [80]. The authors propose a compiler integrated auto-tuning solution capable of performing automatically the entire sequence of operations required by auto-tuning: generation of parameterized loops, search space pruning, and program execution and measurement.

An extension of CHiLL for GPUs is presented in [81]. It automatically transforms annotated C code into CUDA code. In [82, 83] the authors describe auto-tuning of GPU programs taking into account the input dependency problem in which the optimization parameters depend on input parameters. The approach presented in [82] is based on pragmas which are used for specifying parameter ranges. In [83] the authors describe an auto-tuning solution for transforming code written in the language defined in [84] into optimized CUDA code.

The work described in this chapter differs from most auto-tuning approaches in that here the emphasis is on reducing the time spent measuring a code variant. Moreover, general optimization parameters for GPU programs are presented in combination with a method of pruning the resulting search space. These concepts are studied in a new context given by the sparse grid algorithms introduced previously in the thesis.

5.3 Auto-tuning Optimizations

5.3.1 Overview

Auto-tuning an application can be very time consuming since it often implies searching in a high-dimensional space for the point that optimizes a chosen metric, e.g. minimizes the execution time or maximizes the GFlop/s rate. A challenge comes from the fact that the search has exponential complexity relative to the number of optimization parameters. In order to accelerate auto-tuning, one can explore the following three areas:

1. *Advanced search algorithms* address the exponential complexity of auto-tuning. Commonly used algorithms are Simulated Annealing and Nelder-Mead.
2. *Less execution time* means decreasing the time needed for measuring the performance of the program for different values assigned to the optimization parameters.
3. *Search space pruning* reduces the set of candidate points to a manageable size. Here, a set of values for each optimization parameter is specified. Orthogonal sets of parameters, referred to as partitions, can further reduce the cardinality of the space since this allows to replace the Cartesian product with a union of smaller subspaces.

Although the contributions of this chapter touch all the three points, emphasis is placed on 2 and 3. For a list of general search algorithms for auto-tuning the reader is referred to [72].

5.3.2 Search Space Pruning Using Partitions

Space pruning can be achieved by restricting the sets of values for the optimization parameters and by specifying constraints among parameters. Besides this, this section introduces the concept of *partitions* through which one can split a high-dimensional search space into multiple lower-dimensional subspaces. Partitioning can significantly reduce the complexity of the search and this often leads to orders of magnitude fewer points in the search space.

Consider a 2d search space built from the optimization parameters $p_0 \in R_0$ and $p_1 \in R_1$. The minimization of the execution time using full search translates to the exploration of the Cartesian product, $S_{cart} = R_0 \times R_1$. In this situation, the cardinality of S_{cart} is $|S_{cart}| = |R_0| \cdot |R_1|$. Instead of forming the search space by means of a Cartesian product, one could consider the search space $S_{ortho} = (R_0 \times \{c_1\}) \cup (\{c_0\} \times R_1)$, where $c_0 \in R_0$ and $c_1 \in R_1$. The cardinality of S_{ortho} decreases to $|S_{ortho}| = |R_0| + |R_1|$. Performing full search on this second space is called *orthogonal search* [72]. Although orthogonal search may miss the optimum, examples in which the method is used include the ATLAS library [65]. In orthogonal search, the search for the optimum value of one optimization parameter is performed while keeping the others fixed.

Partitions allows for the creation of orthogonal sets of optimization parameters. Hence, one has the possibility to create *hierarchical algorithms*: orthogonal between partitions whereas any algorithm can be used inside a partition. Consider the parameters $p_0 \in R_0$, $p_1 \in R_1$, $p_2 \in R_2$, and $p_3 \in R_3$. By analyzing the correlations between the parameters, one can create the partitions: (p_0, p_1) and (p_2, p_3), e.g. meaning that there is a strong correlation between p_0 and p_1 (and between p_2 and p_3) whose tuning cannot be done independently. For example, p_0 and p_1 could be interdependent parameters that control the cache utilization. p_2 and p_3 could control the parallelism. In this example, p_0 and p_1 are minimized first and the search space is $R_0 \times R_1$ while p_2 and p_3 have fixed values. Second, the best values found for p_1 and p_2 are used when searching for the best p_2 and p_3 in the search space $R_2 \times R_3$. The main benefit lies in the reduced complexity of the final search space. Instead of the initial 4d space, partitions replace this space with two 2d search spaces.

Partitions have several requirements. First, there must be an order in which they are processed. Second, values for the fixed parameters have to be set. Third and most important, dependency relations between parameters have to be set from which the partitions are derived. In general, this can be achieved by using application specific knowledge. Auto-tuning solutions realize part of this functionality using constraints: In [76], optimization parameters are considered orthogonal by default. They become dependent if they appear in the same expression used as a constraint for pruning the search space. However, in this case, the order is not explicit and there is no definition for the fixed parameters. Consequently, partitions are complementary to constraints and provide more flexibility for fine tuning the search method.

Using partitions, one can build hybrid search methods placed between search methods that explore the Cartesian product with an exponential cardinality, and orthogonal search which traverses a space with a cardinality that depends linearly on the number of dimensions. Aggressively splitting a high-dimensional search space into multiple lower-dimensional search spaces can even make it feasible to perform full search on the resulting search subspaces. This is in general not possible on the initial space.

5.3.3 Representative Execution Sampling by Reducing Input Data

This section focuses on reducing the size of the input data so that the execution time for a code variant decreases. This technique can be referred to as *input reduction.* The goal is to extract a representative sample of a program's execution in order to estimate a characteristic of the entire execution, e.g. a performance metric such as time or GFlop/s rate. The benefit of input reduction for auto-tuning is that a smaller input data typically results in a faster execution of a code variant. The main requirement is that the execution of the program using the reduced input data must approximate the execution for the original data. Moreover, reducing the input data must avoid situations in which effects irrelevant for the global behavior gain more importance.

The problem of reducing the input data for auto-tuning is expressed in Eq. 5.1. Here, $\underline{b}$ and $\underline{s}$ are vectors containing the input parameters, $\underline{b}$ refers to the parameters generating the initial or the *big* size problem while $\underline{s}$ refers to the parameters corresponding to the reduced input data or the *small* size problem. T is the objective function or the performance metric that is optimized through auto-tuning. Without loss of generality, assume that T is the execution time. In order for the input data reduction to be effective, one has to determine a transformation from $\underline{b}$ into $\underline{s}$ so that $T_{\underline{s}}$ for the small data is less than $T_{\underline{b}}$ across all (or most) of the values assigned to the optimization parameters contained in $\underline{p}$. In general, this requirement is not difficult to achieve. In fact, the real challenge comes from the last line in the equation which requires that the optimization parameters that minimize $T_{\underline{s}}$, i.e. the small problem, provide also a good approximation for the minimum value of the initial $T_{\underline{b}}$ function.

$$\boxed{\begin{array}{c} T_{\underline{s}}(\underline{p}) < T_{\underline{b}}(\underline{p}),\ \forall\ \underline{p} \\ T_{\underline{b}}(\underset{\underline{p}}{argmin}(T_{\underline{s}})) \simeq min(T_{\underline{b}}) \end{array}} \tag{5.1}$$

5.4 Auto-tuning of GPU Programs

5.4.1 Input Reduction for GPU Programs

Challenges

There are four major risks that can affect the effectiveness of input data reduction on GPUs: (1) insufficient parallelism, (2) tail effects, (3) losing the computational character, and (4) measurement noises. These aspects are discussed next.

First, GPUs are massively parallel processors on which tens of thousands of threads run concurrently at any time. Therefore the small input data must ensure that a GPU is fully utilized, including both the Streaming Multiprocessors (SMs) and multithreading. This requirement comes from the observation that when a GPU program processes big data, most of the time the GPU is fully utilized and this is the behavior that must be captured by the reduced input data.

Second, GPU programs processing small input data can be affected by tail effects [85] which occur when the input data generates an amount of parallel work that is not a multiple of the maximum number of active (concurrent) threads, a number that depends highly on the CUDA kernel and the GPU. Note that this situation may occur even when the first requirement is satisfied. In this case, a tail effect can be observed when the last group of active threads are executed on the GPU. Sometimes the number of threads in this last group does not allow for a full utilization of the GPU's resources. Especially for small input data, a tail effect can consume a considerable amount of time which together with a low utilization of resources can have a significant negative impact on performance. In general, a GPU program that processes big data is executed as a sequence of waves of threads or iterations in which all except the last iteration utilize the entire GPU. Thus, a tail effect does not capture the global behavior of the program. In order for the input data reduction to be effective, tail effects have to be eliminated.

Third, the computational character of the program has to be preserved. In some cases, aggressively reducing the input data can result in less potential for data reuse and accordingly optimizations targeting locality lose their effectiveness. The size of the data must be large enough so that locality optimizations in particular can be properly explored. Note that this also implies that the reduction of the input data depends on the optimization parameters. Intuitively, this means that the reduced input data is not fixed but changes with the values assigned to the optimization parameters during the auto-tuning process.

Fourth, at a reduced scale, the effects of the optimizations can be hidden behind measurement noises, e.g. interference from other processes, which for the initial big data are not detectable. One advantage of GPUs in this context is that GPU programs are provided with a strong isolation, i.e. programs typically execute one at a time on a GPU, meaning that interference from other GPU programs is significantly reduced.

Full Utilization of the GPU

In order to use the input data reduction technique with GPU programs, it is necessary first to determine the minimal amount of work necessary to fully utilize the GPU. This work can be referred to as *active work*. The active work keeps busy an entire GPU and can be correlated to the maximum number of active threads. The relation between work and threads ensures both that the GPU's SMs are occupied and that multithreading is fully used to hide the pipeline latency. Simply ensuring the utilization of all the SMs without considering multithreading is often not sufficient for obtaining high performance on GPUs. Since GPUs are in-order processor architectures, multithreading is important as it provides the means to hide the instruction pipeline latency, e.g. arising from accesses to global memory. Multithreading on GPUs allows for inexpensive switching between thread warps at every clock cycle. Provided that the warps are independent, switching results in a better exploitation of the execution units within an SM.

The maximum number of active threads on the GPU depends on the register consumption, R_k, and the shared memory consumption, S_k, per thread block. Based on these two parameters, one can calculate the maximum number of active blocks, B_{gpu}, that can run on the GPU at any moment of time. This number can be obtained by traversing one by one the equations shown below (see [14] for a detailed explanation of the equations). The description of the parameters used in the equations is given in Table 5.1. B_{size} is the size of the thread block and is set by the programmer. Its value is typically 128 or 256. The parameters W_{size}, G_t, G_s, R_{sm}, S_{sm}, C_{block}, and M_{gpu} are properties of the GPU.

$$W_{block} = ceil^1\left(\frac{B_{size}}{W_{size}}, 1\right) \tag{5.2}$$

$$R_{block} = ceil\left(R_k \cdot W_{size}, G_t\right) \cdot W_{block} \tag{5.3}$$

$$S_{block} = ceil\left(S_k, G_s\right) \tag{5.4}$$

Both the register and the shared memory consumption, R_{block} and S_{block}, act as constraints for the maximum number of active blocks per SM, B_{sm}. The minimum computed in Eq. 5.5 ensures that the resulting B_{sm} blocks can get all the resources needed for execution.

$$B_{sm} = min\left(\frac{R_{sm}}{R_{block}}, \frac{S_{sm}}{S_{block}}, C_{block}\right) \tag{5.5}$$

The result of this sequence of equations is B_{gpu} (Eq. 5.6) which can be used to compute the active work and the size of the reduced input data. In order to avoid tail effects, the size of the reduced input data must be chosen so that the amount of work is a multiple of B_{gpu}.

$$\boxed{B_{gpu} = M_{gpu} \cdot B_{sm}} \tag{5.6}$$

The optimization parameters often affect the register and shared memory consumption per kernel. The implication is that the maximum number of active threads changes across different values for the optimization parameters. Since the input data is reduced according to the maximum number of active threads, the input data's size also varies across different values for the optimization parameters, which is an important observation.

Input Reduction Assumptions and a GPU Model

There are two central assumptions at the foundation of the input reduction technique technique: First, the threads are homogeneous, and second, all the threads that occupy the GPU at any time, start and finish simultaneously. Provided that these two requirements are met, a performance model for GPUs can be built.

Many GPU applications are characterized by a uniform parallel work which means that

[1] *ceil*(x, y) is x rounded-up to the closest multiple of y.

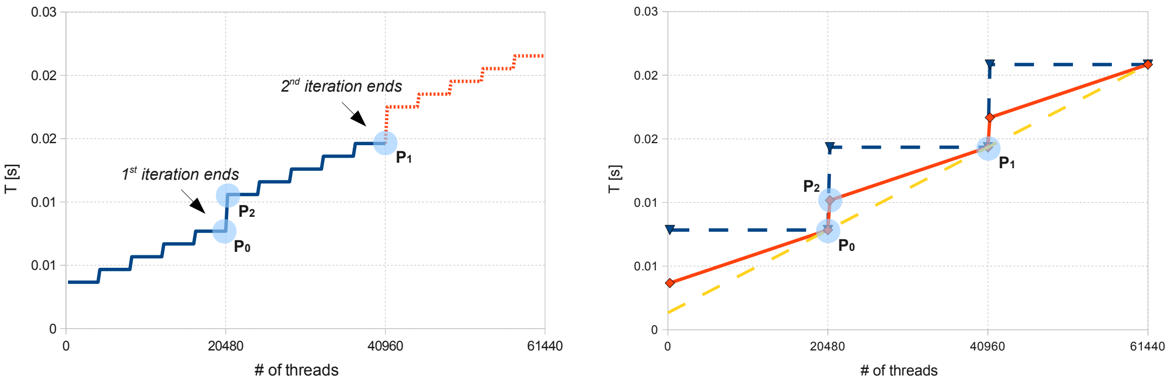
Parameter	Description	Values
W_{block}	Num. of warps per block	-
B_{size}	Num. of threads per block	-
W_{size}	Num. of threads per warp	32
R_{block}	Num. of registers per block	-
R_k	Num. of registers used by kernel	-
G_t	Register allocation granularity	64
S_{block}	Shared mem. per block	-
S_k	Shared mem. used by kernel	-
G_s	Shared mem. allocation granularity	128
B_{sm}	Num. of active blocks per SM	-
R_{sm}	Max. registers per SM	32 K
S_{sm}	Max. shared mem. per SM	16 KB / 48 KB
C_{block}	Constraint for num. of active blocks per SM	8
B_{gpu}	Num. of active blocks for entire GPU	-
M_{gpu}	Num. of SMs	15

Table 5.1: CUDA parameters with descriptions and default values for an Nvidia Fermi GPU.

the threads handle the same amount of work. These threads can thus be called *homogeneous threads*, a common type of threads encountered mostly in data parallel applications. From this point on, it is assumed that the thread scheduler per SM is round-robin and at every clock cycle it switches between thread warps even if they are from different thread blocks. Under these conditions, if the threads are homogeneous then it is expected that they execute in the same amount of time. More importantly, threads that execute in parallel (on different SMs) or concurrently (on the same SM) are also expected to finish executing at almost the same time.

In general, the threads composing a warp execute instructions in a lock-step fashion, i.e. an instruction is broadcast to the entire thread warp and every thread executes that instruction. Threads in a warp cannot execute different instruction at the same clock cycle but different warps are not affected by this limitation. Thread blocks running on different SMs can also execute different instructions. This actually makes GPUs a more flexible SIMD processor. Nevertheless, in the case in which the GPU threads are homogeneous, since threads process the same work, one can assume a lock-step model of execution that applies to the entire GPU, virtually making the GPU a huge SIMD processor.

If the lock-step model is extended to the entire GPU, then threads scheduled for execution on the GPU are executed in iterations, meaning that thread blocks are logically grouped together in waves and the waves are executed one after another, in different iterations. Moreover, during each iteration, a wave occupies the entire GPU. The number of threads required to fully occupy the GPU is B_{gpu}. Hence, the execution time of a kernel can be obtained by summing up



(a) Execution profile of *sginterp* on the GPU.

(b) Approximations of execution time on the GPU.

Figure 5.1: Execution profile of *sginterp* for different number of threads.

the execution time of all the iterations. Empirically, it can be shown that the first iteration consumes slightly more time than the rest which have approximately the same execution time.

In reality, the execution profile of a CUDA kernel is often similar to the one depicted in Fig. 5.1a. This graph is obtained by launching only one kernel using different number of threads starting with 256 in steps of 256, and measuring the execution time. The marks on the horizontal axis denote multiples of the number of threads that fully occupy the GPU. The first iteration finishes at 20480 which represents the number of threads that fully occupy the GPU in this particular situation. So far in the discussion, the GPU is seen as a huge SIMD processor but this is invalidated by the the small steps seen from 4096 to 20480 (threads). Based on the initial model, the execution time of one thread is the same as the execution time of 20480 threads, an intrinsic characteristic of SIMD processing. On the other hand, this behavior can only be observed from 256 to 4096. In this case, the execution of one thread block of size 256 takes the same time as the execution of 16 thread blocks since they execute in parallel on different SMs.

In Fig. 5.1a, the steps have different heights. Every time a new iteration is started, the steps are bigger than before. Within an iteration, adding more thread blocks per SM, e.g. moving from 4096 to 8192, helps multithreading to reduce the pipeline latency, resulting in smaller steps. Based on these observations, a GPU can be modeled as a huge *multithreaded* SIMD processor. In general, in order to obtain an accurate execution profile of a CUDA kernel, i.e. time as a function of the number of threads, both characteristics have to be modeled: SIMD and multithreading. However, depending on the situation, the requirements can be more relaxed, e.g. for a large number of threads, it is less important what happens within an iteration.

Another characteristic of the profile shown in Fig. 5.1a is that it has a periodic behavior: The profile of an iteration appears over and over again, in every subsequent iteration. This is important when one tries to accurately approximate tail effects. The exact behavior of one iteration is of less importance when the kernel is executed by a number of threads in the order of 10^6, which ensures that the GPU is fully utilized. In this case, tail effects have a limited impact as they occur after 40 iterations of full utilization of the GPU.

Fig. 5.1b depicts three methods for approximating the execution time of a GPU kernel as a function of the number of threads. The dashed blue line assumes that the GPU is a huge SIMD processor without multithreading. The continuous red line considers in addition the effect of multithreading which is modeled using a linear function, e.g. from 256 to 20480. Finally, for completeness, the dashed yellow line is the simplest approximation which would correspond to a very fast low latency processor that executes the threads one by one. The first approximation provides an upper bound for the execution time, the third provides a lower bound, whereas the second is the closest to the real time. In theory, given the previous assumptions, at most 3 points are required to build these approximations: P_0, P_1, and P_2.

It is important to notice that whenever the number of threads is a multiple of 20480, the approximations behave all the same. If it is not a multiple and the kernel executes in 10 iterations, then the difference between the approximations can be up to 10%. In order to explain this situation, imagine that only one thread executes in the last iteration. When the GPU is modeled as a huge SIMD processor, the execution is the same as if the GPU is fully occupied, i.e. the execution time for one thread is the same as for 20480 threads. However, when the GPU is modeled as a low latency processor, the execution time can be neglected for one thread compared to 20480 threads. Consequently, the difference between the two approximations can be up to 10% for 10 iterations and is inversely proportional with the number of iterations.

For input reduction, the benefit of using the approximation comes from the fact that instead of executing an initial work that translates for instance to 10^6 threads, a smaller work is processed, necessary for determining the execution time for P_0, P_1, and P_2. Looking again at Fig. 5.1b, the amount of work measured in threads required for calculating P_0, P_1, and P_2 is: $20480 + 20736 + 40960 = 82176$. 20736 corresponds to a number of threads that occupies the entire GPU in the first iteration while only one thread block (256 threads) is executed in the second iteration. Compared to 10^6 threads, this means that auto-tuning can be accelerated up to 10x since there is less work done for evaluating each code variant. It is important to note that this speedup can be further improved by using different methods for obtaining the execution profile. Up to this point only one has been discussed, i.e. executing the kernel multiple times for different number of threads. The time measurements are in this case done using CUDA events which offer a resolution of 0.5 µs.

A more efficient but more intrusive method of discovering Fig. 5.1a relies on instrumenting the kernel using the *clock* instruction which reads the number of cycles from a counter local for each SM. In this case, by knowing the start time and end time for each thread, the same graph from Fig. 5.1a can be built and this requires the kernel to be executed only once using 40960 threads (a 2x speedup compared to the previous method). In this approach, the graph from Fig. 5.1a is sampled by every thread block, i.e. in steps of 256 threads. Moreover, this allows for a more accurate approximation to be obtained which offers a better view on tail effects.

5.4.2 Optimization Parameters for GPU Programs

Parameter Description

The use of auto-tuning for exploring GPU optimizations is motivated by complex interactions and trade-offs between GPU optimization parameters which are difficult to treat theoretically. Besides optimization parameters that are application specific, there are parameters with a more general purpose character applicable to a wide range of GPU applications. Some of them are discussed next together with their interactions.

A common optimization parameter [14] is the **thread block size** (*bs*). It affects the *occupancy* metric which measures the GPU utilization in terms of number of threads. GPUs have a hardware limit for the number of concurrent threads, e.g. 1536 per SM, and the recommendation is to be as close as possible to that number so that multithreading can provide the best performance. The impact of *bs* goes beyond that. The parallelism and the scheduling of blocks on SMs is also influenced by the thread block size especially for rather small data which translates to a small number of threads. In this case, large thread blocks can result in the processing's taking place on a subset of the SMs, meaning the GPU is underutilized. In contrast, if the thread blocks are small then there are more blocks that can be placed by the GPU's scheduler on different SMs, resulting in better performance.

Another optimization parameter is the **thread granularity** or the *number of work units per thread* (*pt*). The importance of this parameter has already been stressed in [17, 54]. In order to understand thread granularity, one can imagine a *one-to-many relation* between a thread and multiple components in the output data computed on the GPU. Increasing the thread granularity is often done in order to improve the ILP and the locality at register or shared memory level. Therefore, in terms of resources it translates to more register and / or shared memory consumption per block. There is a trade-off between *pt* and *bs*, or in other words between locality and multithreading. More precisely, the thread block size is chosen so that the occupancy is maximized. Increasing *pt* often results in the occupancy's being reduced. For example, at 33% occupancy (512 threads from the limit of 1536 threads) a GPU can still make use of the maximum 63 registers allocatable per thread, leveraging locality at register level.

Moreover, with regard to parallelism, if the parallel work is insufficient for feeding the GPU, then the trade-off between *pt* and *bs* has to be explored again since more parallelism can be obtained by experimenting with both parameters.

An optimization parameter essential in scenarios that suffer from insufficient coarse-grained parallelism is the **work granularity** or the *size of a work unit* (*wu*). In this case the goal is to make the parallelism more fine-grain by tuning *wu*. This parameter is already exposed in some CUDA programs [18]. The *wu* parameter controls the *many-to-one relation* between threads and the component of the output computed on the GPU. The trade-off is that locality can be affected and an extra reduction step is necessary to combine the contribution from multiple threads. There are multiple options for reduction: by launching a reduction kernel, by using the shared memory, or by using atomic memory operations from CUDA. It is assumed next that the reduction is done using atomic operations. *wu* is complementary to the *pt* parameter, e.g. the former controls the parallelism while the latter controls the locality. They can even coexist resulting in a *many-to-many relation* between threads and output entries.

Partitioning the Search Space

There are two situations of interest when analyzing the dependencies between optimization parameters: First, there is enough coarse-grained parallel work to occupy the entire GPU and second, the initial work is insufficient to fill the GPU.

In the first case, there is no reason to generate more parallel work using *wu*. Therefore, *wu* is said to be inactive and is discarded from the search. Here, it is common to search for the best *pt* while keeping *bs* fixed, followed by the reversed situation in which *pt* is set to the previously found best value and the best *bs* is searched for. This often results in low occupancy, meaning that there are fewer threads per SM to hide instruction pipeline latencies using multithreading. Consequently, such an orthogonal search puts more emphasis on locality and ILP per thread. [17, 54] provide evidence for the benefit of this strategy in the context of dense linear algebra but there are no guarantees regarding its applicability to all GPU applications or to the next generations of GPUs. In order to fully explore the trade-off between multithreading and locality, *bs* and *pt* have to be considered interdependent.

In the second case, *wu* is active and is used to create more parallelism. *bs* and *pt* can also influence the parallelism to some extent. Compared to large thread blocks, small blocks can provide a better utilization of the SMs. Assume that a GPU has to process parallel work for 1024 threads. A thread block of size 1024 uses only one SM. 8 thread blocks of size 128 each can be scheduled to 8 SMs, resulting in the usage of more SMs. Regarding *pt*, it is inversely proportional to the number of threads. Hence, decreasing *pt* can also help to increase the

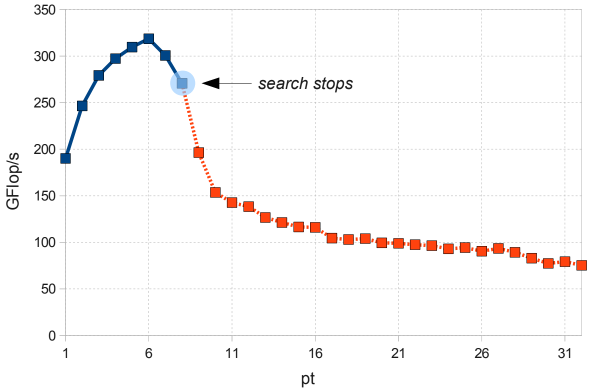


Figure 5.2: Search algorithm. It stops when two successive values for *pt* decrease the performance.

amount of parallelism. Consequently, there is a connection between these parameters given by the fact that increasing the parallelism can be achieved by tuning all of them. However, *wu* usually depends more strongly on *pt* since *pt* controls the number of threads (inversely proportional) and *wu* is activated when that number is not large enough to occupy the entire GPU. In terms of search partitions, this means that *pt* and *wu* should be in the same partition. Depending on the strategy followed, e.g. giving priority to locality instead of multithreading, *bs* can be placed in a different partition, splitting the original 3d search space into two subspaces with fewer dimensions, i.e. a 2d space built from *pt* and *wu*, and a 1d space for *bs*.

Searching for the Best Parameter Values

For searching in the optimization space composed of *bs*, *pt*, and *wu*, a variant of hill climbing is employed. The central concept is shown in Fig. 5.2 for determining the best value for the *pt* parameter. The search starts with $pt = 1$ and progresses towards points that improve the performance, here *GFlop/s*. The search for the best *pt* often translates to finding the value that makes the best use of the fast memory, e.g. register file or L1 cache. If a given *pt* leads to this capacity's being exceeded, then $pt+1$ and $pt+2$ should also result in performance degradation. Therefore, the stop condition of the algorithm is satisfied when one or more successive points do not improve the best GFlop/s already found. This is depicted in Fig. 5.2.

Listing 5.2 contains the search algorithm. The algorithm contains a loop nest with three loops which indicate a 3d search. It is worthwhile noting that the optimization parameters and the input reduction technique are in fact complementary to any search method. The search method discussed here is customized for the three optimization parameters presented in this chapter. With regard to partitions, in Listing 5.2 *bs*, *pt*, and *wu* are all placed in one partition,

Listing 5.2 Search method. *bs*, *pt*, and *wu* are dependent. The search space is 3d. *wu* is activated only when the parallel work is insufficient to occupy the GPU (line 6).

```
1: for bs = 32 to 1024, step 32 do
2:   pt = 1
3:   qpt.reset()
4:   while (pt <= m) do
5:     wu = n
6:     if (m / (pt * bs) < B_gpu(bs, pt)) then
7:       qwu.reset()
8:       while (wu >= 1) do
9:         g = measure_gflops(bs, pt, wu)
10:        res_map.add((bs, pt, wu), g)
11:        qwu.push(g)
12:        if (qwu.is_full()) and (qwu.first() == qwu.max()) then break
13:        wu = wu / 2
14:      qpt.push(qwu.max())
15:    else
16:      g = measure_gflops(bs, pt, wu)
17:      qpt.push(g)
18:      res_map.add((bs, pt, wu), g)
19:    if (qpt.is_full()) and (qpt.first() == qpt.max()) then break
20:    pt = pt + 1
21: return res_map.max()   // ((best_bs, best_pt, best_wu), gmax)
```

meaning that they are considered dependent. *qpt* and *qwu* are *circular buffers*. They contain the GFlop/s for consecutive *pt* / *wu* and are used to implement the stop condition in which the search for the best *pt* / *wu* ends if the maximum capacity of the buffer is reached and the oldest GFlop/s (stored in the first position) is not surpassed. The *first* method returns the oldest GFlop/s value in the buffer whereas *max* returns the maximum GFlops/s in the buffer. In practice, the maximum capacity of *qpt* and *qwu* is set to 2.

The integration of *wu* in the search algorithm follows an activation rule that takes into account the utilization of the GPU. This rule is shown in line 6 of Listing 5.2. It is not necessary to search for *wu* if there is enough parallelism. As previously discussed, the role of *wu* is to increase the parallelism so that the GPU is fully occupied. An observation is here worthwhile. In general, having more threads working on the same output component implies that reduction has to be executed on the GPU. When activated, the search for *wu* is the same as the one used for searching for *pt*, displayed in Fig. 5.2. This is explained by the fact that decreasing *wu* increases the parallelism up to the point where the overhead resulting from reduction does not allow for the performance to be improved. Furthermore, sometimes a small *wu* reduces the locality, i.e. when a piece of data is reused *wu* times. Whenever this situation occurs, increasing the parallelism can no longer improve the performance.

Listing 5.3 Search method using partitions. *pt* and *wu* form the first partition. *bs* is in the second partition. The initial search space is divided into: 2d (*pt* and *wu*) and 1d (*bs*).

```
1: bs = 512    // fixed
2: ... ... ...    // same as before
3: ((fixed_bs, best_pt, best_wu), g) = res_map.max()
4: for bs = 32 to 1024, step 32 do
5:   g = measure_gflops(bs, best_pt, best_wu)
6:   res_map.add((bs, best_pt, best_wu), g)
7: return res_map.max()    // ((best_bs, best_pt, best_wu), gmax)
```

A second search algorithm evaluated in this chapter is shown in Listing 5.3. Here, *bs* is placed in a separate partition from *pt* and *wu*. Hence, in a first stage (line 1) in which $bs = 512$, the best values for *pt* and *wu* are searched for. In line 2, the search method executes the same operations as in lines 2 - 19 from Listing 5.2. A second stage (line 4) uses the best values found for *pt* and *wu*, i.e. *best_pt* and *best_wu*, in order to find the best value for *bs*. A notable aspect in this case is that the original 3d search space is split into 2 lower-dimensional search spaces: a 2d one for *pt* and *wu*, and a 1d one for *bs*. This search method gives priority to locality rather than multithreading by searching first for the best *pt* and afterwards for the best *bs*.

5.5 Evaluation

This section focuses on evaluating the theory described before in the context of sparse grid routines. Two routines play a central role here: sparse grid interpolation (*sginterp*) and sparse vector - matrix multiplication (*spvm*). In contrast, the third sparse grid operation described in this thesis, sparse grid hierarchization (*sghierarch*) has less potential of being optimized using auto-tuning. On the other hand, as it is shown at the end of this chapter, from applying auto-tuning to *sginterp* and *spvm*, lessons can be derived that are useful in a wider context, e.g. dense linear algebra, stencil computation, and direct n-body method.

5.5.1 Experimental Setup

The hardware setup used in the experiments is the following:

- *sysA*: a system containing a Quad-core Intel Nehalem i7-920 (2.67 GHz, 8 hardware threads) and an Nvidia GTX480 (1.4 GHz, 15 cores, 32 SIMD lanes / core)

- *sysB*: a dual-socket system, one Quad-core Intel Nehalem E5630 per socket (2.53 GHz, 8 threads) and an Nvidia Quadro 6000 (1.15 GHz, 16 cores, 32 SIMD lanes / core).

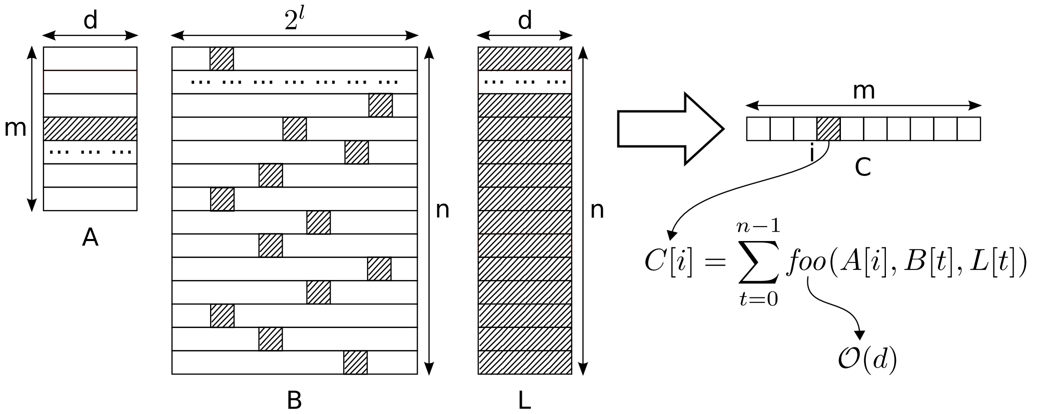


Figure 5.3: Simplification of *sginterp*. It shows how the output vector C results from the matrices A, B, and L. *foo* is $\mathcal{O}(d)$ and accesses one value from a B's row and one row from L.

Both GPUs are based on the Nvidia Fermi architecture.

For compilation, the *gcc* compiler is used, version 4.5.2 on sysA and 4.4.5 on sysB. The GPU part of the code uses Nvidia's CUDA version 4.2 on sysA and sysB. Both case studies from this chapter, *sginterp* and *spvm*, operate with single precision floating point numbers.

5.5.2 Sparse Grid Interpolation

Description

As shown before in the thesis, *sginterp* has an optimized GPU implementation which is part of a memory efficient numerical library, called *fastsg*, for interpolating high-dimensional functions. That implementation is used here as the reference for performance. The objective is to improve the performance by exploring the auto-tuning parameters previously described, i.e. *bs*, *pt*, and *wu*, and to accelerate the auto-tuning process by reducing the input data.

A simplified view on the computation performed by *sginterp* is provided in Fig. 5.3. More information on the central loop nest of interpolation is shown in Listing 5.4 which references the *foo* functions from Listing 5.5. m, d, n, and l are the input parameters. The input data is given by A, a matrix of size m (rows) $\times$ d (columns), and B, a matrix of size $n \times 2^l$. The output array is C, a row-vector of size m. In sparse grid terms, A contains the coordinates of the d-dimensional points where the compressed data is interpolated, B contains the compressed data, while L is an $n \times d$ matrix of integers in which every row is unique. C contains the results of the interpolation for each point (row) in A. Computing every component of C needs one row from A and exactly one value from each row in B. The *computational intensity*, i.e. the number of floating point operations per global memory reference, of *sginterp* is rather high since *foo* detailed in Listing 5.5 executes $6 \cdot d$ floating point operations and d is typically between 4 to 10,

Listing 5.4 Main loop nest of *sginterp*. Initially C is all 0.

```
for i = 0 to m - 1, step pt do           // A tiles
  for j = 0 to n - 1, step wu do          // B tiles
    for jj = j to min(j + wu, n) - 1 do    // inside a B tile
      for ii = i to min(i + pt, m) - 1 do   // inside an A tile
        C[ii] += foo(A[ii], B[j], L[j])      // $\mathcal{O}(d)$ ops, 1 global mem. reference
```

Listing 5.5 Details on *foo* function.

```
w = 1, x = 0                  // w is float, x is int
for k = 0 to d - 1 do
  f  = floor(2^rowL[k] * rowA[k])
  x  = 2^rowL[k] * x + f                                        // int ops.
  w *= max(1 - abs(2^(rowL[k] + 1) * rowA[k] - f * 2 - 1), 0)     // float ops.
return w * rowB[x]
```

e.g. in computational steering [28]. In the same computational steering application, m tends to be in the order of 10^6. In general, n is in the order of 10^4 while l is in the order of 10.

Optimization Parameters

The parallelization of the loop nest from Listing 5.4 using CUDA is done by distributing the iterations of the first 2 loops across GPU threads. It is worthwhile observing that whenever $wu < n$, multiple threads work on the same output component, meaning that reduction is necessary after each thread finishes its computation. The threads are grouped in thread blocks of size bs, where $\boxed{bs \in \{32, 64, 96, 128, \ldots, 1024\}}$ (32 values). Each row of L is stored in shared memory. Thus, each row of L is shared among all the threads in a block. Barriers are required to synchronize the access to L, i.e. threads need to wait until the the current row of L is copied from global to shared memory. Note that in practice L can be either be stored in global memory or can be computed on-the-fly using Eq. 3.16.

There are two aspects that influence the optimal bs. First, a small bs implies that there are more copies of a row of L since there is one row of L in shared memory per thread block. Thus, in oder to optimize shared memory consumption, a large bs is preferred. Second, a large bs means there are more threads waiting at the barriers used for synchronizing the access to L which can result in a reduced throughput of the instruction pipeline. In this case, smaller thread blocks are preferred. Consequently, it is not clear which option is the best, i.e. small or large thread block, and auto-tuning can be used to explore this trade-off.

The parallelization scheme applied to Listing 5.4 results in each thread's working on a tile of size $pt \times d$ from A to compute a number of pt values from C. Using the terminology introduced before, pt allows to control the *one-to-many* relation between threads and output components of C. The pt parameter has a double role. First, it allows for a better reuse of every row from L and of every row from B. Second, it can saturate the register file or the L1 cache if this is not achieved in the case in which one thread computes one component from C. A high consumption of registers per thread without generating register spilling can improve the locality and the ILP within a thread, resulting in better performance.

The range for pt is $\{1, \ldots, m\}$ (m values). In practice however, a better alternative is given by $\boxed{pt \in \left\{1, \ldots, \left\lfloor \frac{m}{M_{gpu} \cdot bs} \right\rfloor \right\}}$. This rejects those situations in which the locality achieved through a large pt, e.g. $pt = m$, reduces the amount of parallel work up to the point where it is insufficient to feed a number of M_{gpu} SMs. Based on common input parameter values found in computational steering scenarios, the set of values for pt has a cardinality in the order of 10^3.

From an implementation's point of view, pt and d are template parameters ensuring that the thread's tile from A and the corresponding tile from C are stored in registers whenever the value of $pt \cdot (d + 1)$ does not result in the register limit's per thread being exceeded. Otherwise, if the two tiles exceed the register limit then they are stored in global memory which is cached on GPUs which contain a two level cache hierarchy.

In general, the parameters bs and pt cannot be explored in isolation from each other without discarding the trade-off between multithreading and locality. Looking only at bs and ignoring pt, the best performance is usually found for the bs that maximizes occupancy, meaning that all the SMs and multithreading are fully utilized. However, this typically does not allow for locality to be harnessed to its full extent, e.g. high occupancy usually means that fewer registers are available per thread for improving the locality at register level. Consequently, a straightforward orthogonal search [72] implies that priority is given to either multithreading or locality. For instance, [17, 54] show that focusing on locality rather than multithreading provides the best performance for dense linear algebra.

The wu parameter is in the range $\boxed{wu \in \{1, \ldots, \frac{n}{4}, \frac{n}{2}, n\}}$ ($\log_2 n$ values). In those cases in which m is small and cannot fully occupy the GPU, i.e. $m/(bs \cdot pt) < B_{gpu}$, additional sources of parallelism have to be searched for. Looking only at Fig. 5.3, the solution to this problem is the following: (1) slicing the B matrix horizontally in tiles of shape $wu \times 2^l$, (2) assigning to each thread a tile from A and a tile from B, and (3) reducing in C the contributions of the threads that process the same tile from A. It is important to note that more parallelism can also be obtained by decreasing pt but at the cost of reduced locality and by decreasing bs at the cost of underutilized multithreading. In this regard, if reduction is fast, using wu for generating more parallelism is the best solution. It is worth reiterating that wu becomes activated, i.e. $wu < n$, only if the amount of parallel work is insufficient to occupy the entire GPU, i.e. $m/(bs \cdot pt) < B_{gpu}$.

Input Reduction

Based on typical values assigned to the input parameters, the cardinality of the search space to explore for *sginterp* is in the order of 10^5. In this situation, the auto-tuning problem is more complex since the input parameters, i.e. m, n, d, and l also influence the optimizations and the best values for the optimization parameters:

- m affects the amount of parallel work as discussed before,
- n affects the reuse of a tile from A, e.g. for a smaller n, the overhead of loading a tile from A into registers / L1 cache has a stronger negative impact on performance,
- d affects the computational intensity and the tiling of A in registers / L1 cache, e.g. a smaller d allows for more rows of A to be included in one tile (a larger pt),
- l affects the reuse of each row of B, e.g. a smaller l improves the cache hit ratio.

This means that in the context of *sginterp* the auto-tuning problem is in fact *7-dimensional.* In order for the performance to be portable across input parameters, auto-tuning has to be performed across different values for the input parameters. In this case, accelerating auto-tuning by reducing the input data gains even more importance.

In the context of *sginterp*, the input reduction technique means finding a vector $\underline{s} = (m_r, n_r, d_r, l_r)$ derived from the initial input parameters $\underline{b} = (m, n, d, l)$ so that Eq. 5.1 is satisfied. The transformation from $\underline{b}$ to $\underline{s}$ is built around an approximation of the execution time as a function of the number of threads which has been described previously in the chapter.

Assuming that $m \gg B_{gpu} \cdot bs \cdot pt$, then $wu = n$ and the number of launched threads is given by $m/(bs \cdot pt)$. According to Eq. 5.6, m can then be reduced to $m_0 = B_{gpu} \cdot bs \cdot pt$ (ensures full utilization of the GPU) and $m_1 = 2 \cdot B_{gpu} \cdot bs \cdot pt$. Both m_0 and m_1 are needed for the construction of the 2-point linear approximation depicted in Fig. 5.1. Therefore, $\boxed{m_r \in \{B_{gpu} \cdot bs \cdot pt, 2 \cdot B_{gpu} \cdot bs \cdot pt\}}$. For a more accurate approximation, at least 3 points are required. If the initial m is in the order of 10^6 (a typical value in computational steering scenarios), m_0 and m_1 can be factors of magnitude smaller. It is important to note that m_r depends on the optimization parameters, meaning it varies across different values for bs and pt.

In the case of insufficient parallelism, i.e. $m < B_{gpu} \cdot bs$, $wu < n$ and the number of launched threads is $m \cdot (n/wu)$. Let $p = m \cdot (n/wu)$. Similar to the other case, p can also be reduced to $\boxed{p_r \in \{B_{gpu} \cdot bs \cdot pt, 2 \cdot B_{gpu} \cdot bs \cdot pt\}}$.

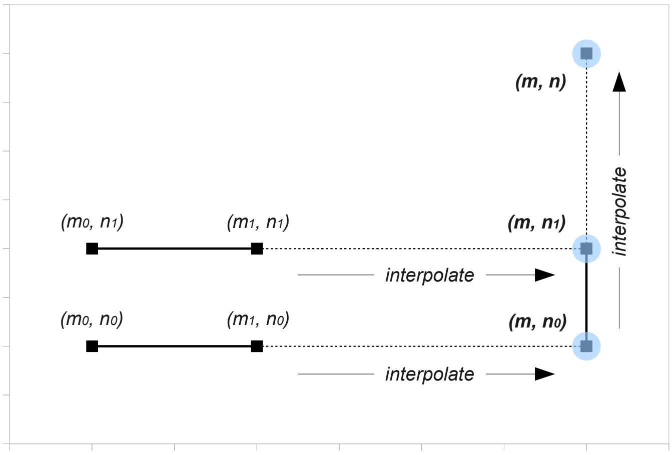
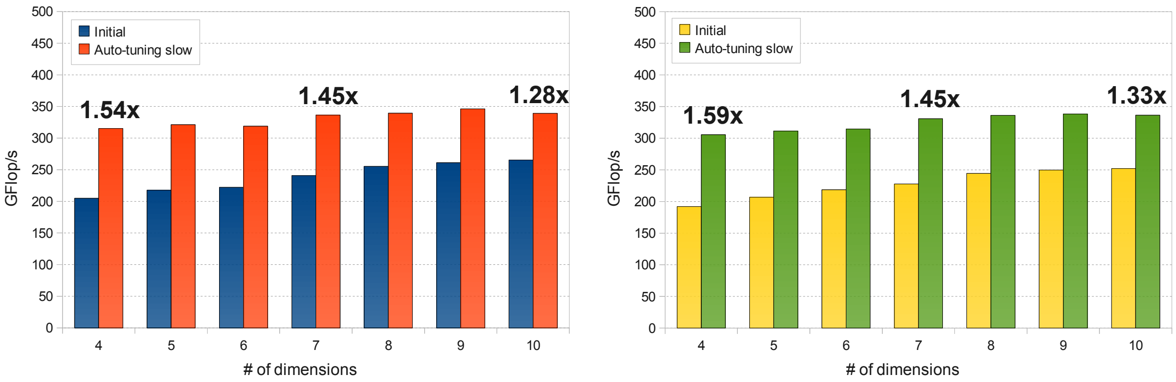


Figure 5.4: Bilinear interpolation at (m, n) using the values of the execution time at: (m_0, n_0), (m_0, n_1), (m_1, n_0), and (m_1, n_1).

The n parameter also provides the means to reduce the input data. Accordingly, wu is also scaled down to wu_r so that $n/wu_r = n_r/wu$, where n_r is the reduced n. In other words, the work per thread is reduced. The typical values for n are in the order of 10^4. The main challenge with reducing n is that in combination with reducing m, the chances to reach the resolution limit of the timing method increase significantly. Note that in general, for common data sets from computational steering, m is expected to be reduced by a factor of 10 while n can be reduced by a factor of 10^3. Overall, the execution time decreases proportionally by approximately 10^4 times, meaning that in some scenarios it is in the order of ms or even less. In general, the timing resolution is assumed to be close to 0.5 µs. In this context, reducing n must be done so that for the smallest n_r, the execution time does not fall below 0.1 ms, i.e. 100 times the timer's resolution.

Besides ensuring that n_r is large enough relative to the timing resolution, two values, n_0 and n_1, are actually needed in order to build a linear approximation of the execution time as a function of n. In other words, $n_r \in \{n_0, n_1\}$. Two values ensure that one-time effects, e.g. initializing data and writing results, do not scale when moving from n_r to n for estimating the execution time for n. The value of n_0 must be chosen so that it satisfies the timer resolution constraint. Then, $n_1 = 2 \cdot n_0$. If one combines the reduction of m with the reduction of n and uses linear approximations in both cases, then the execution time is approximated using bilinear interpolation. Hence, 4 points are needed: (m_0, n_0), (m_0, n_1), (m_1, n_0), and (m_1, n_1). The interpolation process is shown in Fig. 5.4. The arrows denote the order of the operations.

The remaining two parameters, d and l, are not reduced since they are in general in the order of 10 and more importantly they have a strong influence on the computational behavior



(a) GFlop/s rate of auto-tuned *sginterp* on sysA. (b) GFlop/s rate of auto-tuned *sginterp* on sysB.

Figure 5.5: Performance of *sginterp* after auto-tuning on sysA and sysB.

d	Best bs	Best pt	# of variants
4	64	7	184
5	64	6	174
6	64	5	161
7	128	5	140
8	128	4	128
9	128	4	118
10	128	4	116

Table 5.2: Best values for bs and pt resulting from auto-tuning. The best value found for wu is n. The same values are obtained on both sysA and sysB.

and on the optimizations, i.e. bs and pt depend highly on d and l.

Results

The size of the work done in *sginterp*, i.e. the number of rows of A, m (the number of interpolation points), is set to $2 \cdot 10^6$. l (the refinement level of the sparse grid), is 10. The number of columns of A, d, (the number of dimensions), is in the range $4 - 10$.

By applying the search method from Listing 5.2, the results shown in Fig. 5.5 are obtained. In the legend, *auto-tuning slow* refers to the version of auto-tuning without input reduction and partitions. *Initial* is the version of *sginterp* without auto-tuning. This version is used as the performance reference. The best values for the optimization parameters are shown in Table 5.2.

One point to mention here is that the best value for wu is n, meaning that wu is inactive and the amount of parallel work is in general enough to feed the entire GPU. An observation resulting from the experiments is that even for an m in the order of 10^5, wu is still n. wu is typically activated when m is below this threshold.

A second point worthwhile noting is that the input parameter d influences the best value for pt and bs. For instance, for $d = 4$, the best performance corresponds to $pt = 7$ while for $d = 10$, the best pt is 4. The trend is that pt decreases when d is increased.

A third point is that the best value for pt can be calculated accurately provided that the following information is available: the number of registers used by the kernel and the number of registers available on the GPU. If the number of registers used by the kernel for $pt = 1$ is known, then it becomes possible to estimate the number of registers used for any pt and subsequently to determine the value for pt that results in the best utilization of the register file. Since this static method matches the auto-tuning results, the conclusion is that it is not worthwhile on sysA and sysB to store the tiles from A in the L1 cache.

Table 5.2 also includes the number of code variants that are evaluated during the auto-tuning process. The numbers show that approximately 0.1% of the initial 3d search space (with a cardinality in the order of 10^5 points) is actually explored by the search method.

The input reduction technique has two forms: *auto-tuning red0* and *auto-tuning red1*. The first one reduces only m, whereas the second one also incorporates the reduction of n. An observation regarding the first form is that simply building the 2-point linear approximation of the execution time shown in Fig. 5.1 is not enough for obtaining a relative error below 1% compared to the execution time measured for the initial work. In order to address this issue, the implementation uses the *linear least squares method* in order to fit to a line a set containing more than two points. Consequently, *auto-tuning red0* exposes a parameter that controls the number of points used by the linear least squares method for building a linear approximation. On the tested systems, an empirically found value for this parameter that satisfies the requirement for an error below 1% is 5. *auto-tuning red1* includes *auto-tuning red0* and has two extra parameters, n_0 and n_1, for controlling the reduction of n. These two parameters are used as shown in Fig. 5.4. In the experiments, n_0 and n_1 are set to 1 and 10 respectively.

Fig. 5.6 shows the performance of the best code variant returned by different versions of the auto-tuning method. The performance is measured in GFlop/s and the measurements are done on sysA. *Auto-tuning ortho* refers to the implementation of Listing 5.3. It includes *auto-tuning red0* and *auto-tuning red1*. For $d = 9$, *auto-tuning ortho* returns a code variant that performs better than the one returned by *auto-tuning red1*. In theory, this situation should not occur since the points traversed by *auto-tuning ortho* are a subset of the points traversed by *auto-tuning red1*. In reality however, *auto-tuning red1* may return a false optimum predicted using the linear approximation described before. The false optimum is in most cases close in terms of GFlop/s to the real optimum obtained through *auto-tuning slow*. This situation is mainly

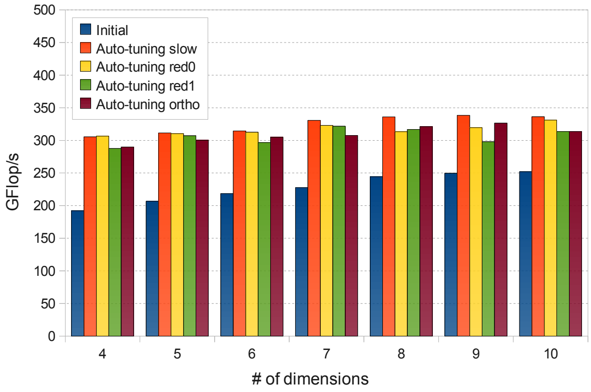


Figure 5.6: Impact of auto-tuning optimizations on the performance of *sginterp* on sysA.

System	Auto-tuning red0			Auto-tuning red1			Auto-tuning ortho		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
sysA	4x	17x	**10x**	84x	6800x	**2514x**	846x	42279x	**21426x**
sysB	6x	12x	**8x**	153x	1969x	**999x**	1594x	16777x	**10379x**

Table 5.3: Speedup of auto-tuning optimizations relative to auto-tuning time.

caused by a small execution time whose effect is explained next.

A small execution time in the order of 1 ms generates serious problems. An important challenge comes from the resolution of the timer. All the measurements included in this chapter rely on CUDA events which in theory have a resolution of 0.5 µs. In reality however, measuring the execution time of a kernel 100 times reveals on the tested systems that the standard deviation is approximately 0.02 ms. If the input data is reduced so that the execution time is smaller than 1 ms, the 0.02 ms can severely affect the comparison between different code variants generated during the optimized auto-tuning process. A more accurate but also more intrusive method for measuring the performance can be implemented using the *clock* instruction within a CUDA kernel. This gives access to a counter local to each SM which is incremented at every clock cycle. An implementation of this method is provided by [86].

Fig. 5.7 provides an aggregate view on the performance measured using optimizations, i.e. input reduction and search partitions, on sysA and sysB. The numbers are averaged for d in the range from 4 to 10. For every value of d auto-tuning is executed 5 times. The results show that there is only a slight difference of less than 5% between the GFlop/s rate for the best optimization parameters returned by *auto-tuning ortho* compared to *auto-tuning slow*. This suggests that for *sginterp*, auto-tuning can give priority to the improvement of the locality rather than to the exploration of the trade-off between locality and multithreading.

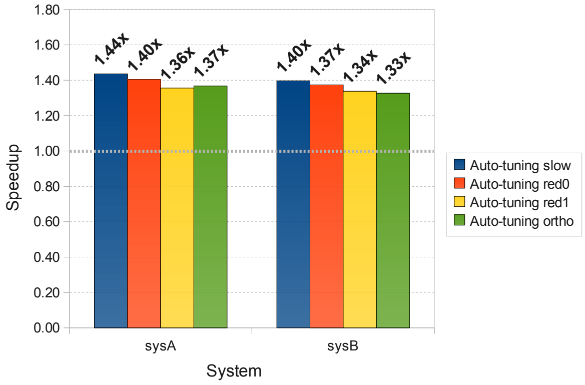


Figure 5.7: Impact of the optimizations on auto-tuning on sysA and sysB. The speedup is calculated relative to the GFlop/s rate measured for the initial *sginterp* without auto-tuning.

Table 5.3 shows the benefits of the optimizations relative to the auto-tuning time measured for *auto-tuning slow*. *Min* and *max* correspond to $d = 1$ and $d = 10$ respectively. This table proves that fast auto-tuning can be achieved at the cost of only a slight decrease in the GFlop/s rate shown in Fig. 5.7. As a concrete example, on sysA, for $d = 10$ *auto-tuning slow* needs 470 s to complete while the optimizations decrease this duration to 16 s for *auto-tuning red0*, 70 ms for *auto-tuning red1*, and 8 ms for *auto-tuning ortho*.

5.5.3 Sparse Vector - Matrix Multiplication

Description

The computation done in *spvm* is depicted in Fig. 5.8. In linear algebra terms, the result y is the inner product between an n-vector x with m non-zero components and A, an n-vector of vectors of size p. The vector of integers ix is used to select only those rows from A that correspond to the non-zero values stored in x. The selected rows are multiplied each with one value from x, and the results are subsequently summed up in y. This computational behavior is characterized by low computational intensity, making *spvm* a memory bound routine.

In computational steering, the A matrix typically contains 10^9 values and the size of each row, p, has approximately 10^6 or more elements. The number of rows of A, n, is in the order of 10^2. The number of selected rows from A is generally smaller than 10^2.

Optimization Parameters

In the parallelization scheme used for *spvm* on GPUs, each thread computes one component of the output vector y. Hence, each thread handles exactly one column from the matrix A.

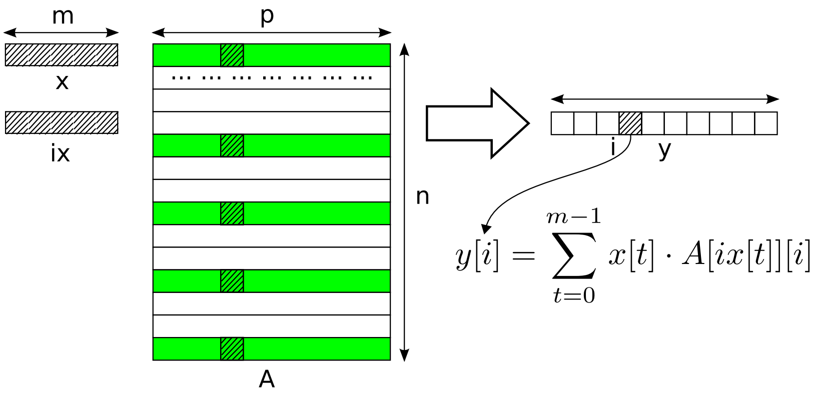


Figure 5.8: Computation done in *spvm*. It shows how the output vector y results from x and A. Only the green rows are used and are selected through the integer vector *ix*.

Listing 5.6 Main loop nest of *spvm*. Initially y is all 0.

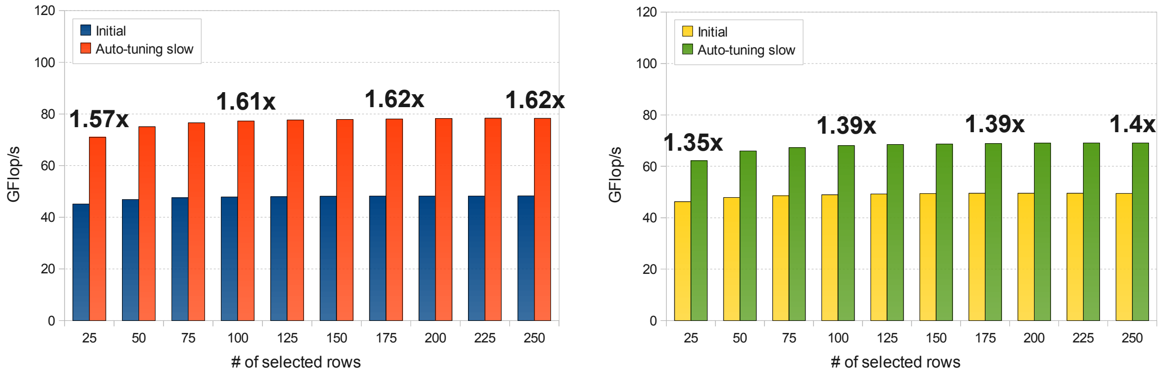
```
for i = 0 to p - 1, step pt do              // y tiles
  for j = 0 to m - 1, step wu do            // A tiles
    for jj = j to min(m + wu, m) - 1 do     // inside an A tile
      for ii = i to min(p + pt, p) - 1 do   // inside an y tile
        y[ii] += x[jj] * A[ix[jj]][ii]
```

In Listing 5.6, the optimization parameters for *spvm* are displayed. The *pt* parameter controls the locality, allowing a thread to compute *pt* components of the output vector. The *wu* parameter controls the horizontal slicing of the matrix A. More precisely, *wu* is inversely proportional to the number of slices. Each thread thus computes the contribution on y of *pt* columns in a slice. By doing so, more parallel work is created by choosing *wu* so that $wu < m$. The same observations from *sginterp* apply also here. First, increasing the amount of parallel work using *wu* requires reducing (summing up) the contributions from multiple threads that work on the same components of y. Second, *wu* should only be activated when the pair *bs* and *pt* does not allow for the GPU to be fully occupied, i.e. when $\boxed{p/(bs \cdot pt) < B_{gpu}(bs, pt)}$, where B_{gpu} comes from Eq. 5.6.

The ranges for the optimizations parameters are the following:

- $bs \in \{32, 64, 96, 128, \ldots, 1024\}$ (32 values)
- $pt \in \{1, \ldots, \lfloor p/(M_{gpu} \cdot bs) \rfloor\}$ (approximately 10^3 values)
- $wu \in \{1, \ldots, m/4, m/2, m\}$ ($\log_2 m$ values, approximately 10 in practice).

The methods used to explore the resulting 3d search space are the same as the methods described for *sginterp*, shown in Listing 5.2 and Listing 5.3. Listing 5.3 is based on the assumption that



(a) GFlop/s rate of auto-tuned *spvm* on sysA.

(b) GFlop/s rate of auto-tuned *spvm* on sysB.

Figure 5.9: Performance of auto-tuned version of *spvm*.

the search for the best bs is orthogonal to the search for the best pair pt and wu. Hence, pt and wu form together a search partition whereas bs is placed in another partition.

Input Reduction

Input reduction for *spvm* is applied to the input parameters p and n. Reducing p is equivalent to reducing the number of threads that execute on the GPU. The reduction of n affects the amount of work processed within a thread. Similarly to *sginterp*, the execution time is approximated as a function with two parameters: the number of components in the output vector y and the number of selected rows from A. The method is depicted in Fig. 5.4 is which for *spvm* the two optimization parameters in the figure are replaced by p and m.

Results

Fig. 5.9 shows the performance of *spvm* across different configurations of the sparse vector: from 25 non-zeros to 250 non-zeros using a step size of 25. The initial version denotes the version of *spvm* for which there is one component of y computed per thread. The auto-tuned version is 1.6x faster than the initial version on sysA, and 1.4x faster on sysB.

The best values for the optimization parameters shown in Table 5.4a is consistent across different values for m only on sysA. The best bs is 96 whereas the best pt is 8. On sysB, the best values for the optimization parameters shown in Table 5.4b depend on the values of the input parameters. On both systems, the best value found for wu is m and the discussion is the same as for *sginterp*. First, this result confirms that there is enough parallelism to keep the entire GPU busy. Second, the reduction done for $wu < m$ adds more overhead to the execution.

Size of x	Best bs	Best pt	# of variants
25	96	8	289
50	96	8	291
75	96	8	284
100	96	8	288
125	96	8	282
150	96	8	283
175	96	8	282
200	96	8	291
225	96	8	287
250	96	8	283

(a) The best values for bs and pt on sysA.

Size of x	Best bs	Best pt	# of variants
25	416	3	285
50	192	4	270
75	128	10	322
100	128	10	304
125	448	10	309
150	128	10	312
175	128	10	321
200	128	10	323
225	128	10	306
250	448	10	307

(b) The best values for bs and pt on sysB.

Table 5.4: The best values for bs and pt resulting from auto-tuning on sysA and sysB. The best value for wu is n on both systems.

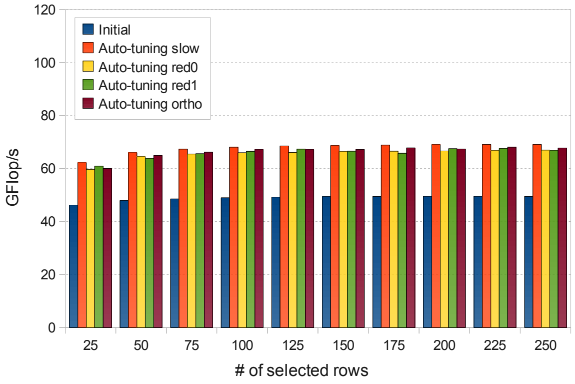


Figure 5.10: Impact of auto-tuning optimizations on the performance of *spvm* on sysA.

In Fig. 5.10 a comparison in terms of GFlop/s is shown between the best code variants returned by different versions of auto-tuning. *Auto-tuning slow* refers to the implementation of Listing 5.2 without input reduction. *Auto-tuning red0* includes the reduction of p whereas *auto-tuning red1* incorporates the reduction of both p and m. *Auto-tuning ortho* is the implementation of Listing 5.3 in combination with input reduction applied to p and m. One can see in Fig. 5.10 that the GFlop/s rate differs only slightly across auto-tuning methods. The fact that *auto-tuning ortho* is almost on par with *auto-tuning slow* validates the design choice made in Listing 5.3, i.e. higher priority is given to locality in comparison to multithreading. More precisely, the best pt is searched for prior to the search for the best bs.

Fig. 5.11 provides an aggregate view on the speedup obtained on sysA and sysB using auto-tuned versions of *spvm* resulting from different auto-tuning methods. Here, the speedup is relative to the GFlop/s rate and is calculated as an average across different input data. For

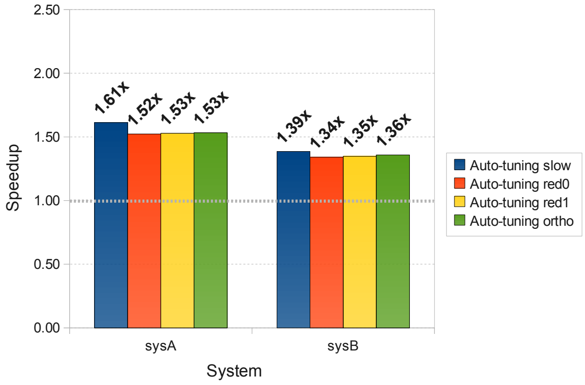


Figure 5.11: Impact of auto-tuning optimizations on performance. The speedup is relative to the GFlop/s rate.

System	Auto-tuning red0			Auto-tuning red1			Auto-tuning ortho		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
sysA	1.5x	4.5x	**2.2x**	1.7x	20.2x	**10.4x**	54.9x	1017.5x	**404.4x**
sysB	1.2x	1.4x	**1.3x**	1.5x	14.3x	**7.8x**	31.1x	404.8x	**208.4x**

Table 5.5: Impact of auto-tuning optimizations on auto-tuning time. Speedup is here relative to the duration of auto-tuning without optimizations.

each input data, auto-tuning is performed 5 times. The graph shows that there is a relative difference of no more than 5% between the performance of the best code variant found by *auto-tuning slow* and any other auto-tuning version. Looking at Table 5.5, one can see the benefits obtained in return for the 5% loss in performance. The *min* and *max* columns correspond to $m = 25$ and $m = 250$ respectively. In absolute numbers, for $m = 250$, the auto-tuning time is decreased as follows: Initially, *auto-tuning slow* needs 2.16 s to complete, *auto-tuning red0* executes in 500 ms, *auto-tuning red1* executes in 120 ms, and *auto-tuning ortho* returns in 3 ms.

5.6 Other Examples

There is a wide range of applications that can benefit from the lessons covered in this chapter, including: matrix - vector multiplication, stencil computation, and direct n-body method. These examples can all expose the optimization parameters previously discussed: *bs*, *pt*, and *wu*. In fact, *bs* is an inherent parameter of CUDA programs.

In matrix - vector multiplication, $A \cdot x = y$, the naïve parallelization is that one *thread block* processes one row of the matrix. In order to improve the locality, one tries to increase the reuse of x. For this, a thread block can work on *pt* rows from A. If the number of rows underutilizes the GPU, more parallelism can be created by slicing A vertically and having each thread block working on one slice. In this case, the slicing is controlled through *wu*.

Stencil computations can be done on 2d or 3d domains, can result from finite difference or from image processing, but in general, the parallelization scheme on GPUs is for each thread to traverse / update the matrix (in the case of a 2d domain) from top to bottom. This means that by default, each thread computes one column of the matrix. Similarly to *spvm*, the *pt* parameter controls the number of columns that are updated by each thread with the objective of increasing the locality. *wu* controls the horizontal slicing of the matrix as before.

In the direct n-body method, all the interactions, i.e. $\mathcal{O}(n^2)$, between the bodies of the system are computed at every iteration. In this context, *pt* allows one thread to update the positions of *pt* bodies, meaning that the thread computes the interactions between its bodies and all the rest. Using *wu*, the parallelism can be increased whenever the number of bodies is insufficient to fully occupy the GPU. More precisely, each thread computes in this case the interactions between its bodies and only a subset of *wu* bodies from the rest.

The input reduction technique can be applied to all these examples in order to shorten the auto-tuning time. Note that input reduction does not restrict one to the search space composed of *bs*, *pt*, and *wu*. In fact, the search space can also be built around other optimization parameters. Input reduction requires that the input parameters export their dependency on the number of threads and that the threads are homogeneous.

5.7 Summary

This chapter proposes techniques for accelerating the auto-tuning of GPU codes. The central point is an input reduction technique in which the input parameters are reduced so that the execution time is significantly decreased but the global behavior is preserved. The technique is used to accelerate the auto-tuning of sparse grid interpolation and sparse vector - matrix multiplication. Especially for sparse grid interpolation, the benefits in terms of auto-tuning time are considerable, e.g. 10^3x, while the loss in GFlop/s rate does not exceed 5% compared to the unoptimized auto-tuning method. In addition to input reduction, general optimization parameters for GPU codes are described in detail. Search partitions are also discussed in the context of search space pruning using correlations between optimization parameters.

Chapter 6

Load Balancing on Heterogeneous Systems

Obtaining high performance on heterogeneous systems implies that the CPU and the GPU are used to process work that matches their specific strong points. This translates in either executing a given routine on the CPU or on the GPU and often this decision depends on the nature of the input data. In other cases, both the CPU and the GPU can be used simultaneously for processing the work. In this context, load balancing is of high importance and one can employ a static or a dynamic scheme for avoiding idle computation resources. In order to achieve load balancing, multi-version routines optimized for the CPU and the GPU are a primary requirement. This chapter describes load balancing techniques that improve the performance of the computational steering application described in Chapter 3. More precisely, load balancing is applied to the central three routines of this applications: sparse grid hierarchization (*sghierarch*), sparse grid interpolation (*sginterp*), and sparse matrix vector multiplication (*spvm*).

6.1 Introduction

Heterogeneous systems containing multi-core CPUs and accelerators enables one to reach higher computational speeds while keeping power consumption at acceptable levels. The most common accelerators nowadays, GPUs are very different in both purpose and characteristics compared to CPUs. Whereas CPUs incorporate large caches and complex logic for out-of-order execution, branch prediction, and speculation, GPUs contain significantly more floating point units. They have in-order cores that hide pipeline stalls through multithreading. Thus, up to 1536 CUDA threads can run concurrently on one GPU core, called Streaming Multiprocessor (SM). According to [1], CPUs can be referred to as latency oriented processors with complex techniques used for extracting Instruction Level Parallelism (ILP) from a sequential stream of instructions. In contrast, GPUs are throughput oriented processors, containing a large number of cores (e.g. 16) with wide SIMD units (e.g. 32 lanes), making them ideal architectures for vectorizable codes. While any application can be executed on CPUs, not all programs can be ported to GPUs or deliver better performance on GPUs. This makes GPUs special purpose processors as opposed to CPUs which have a general purpose character.

Programming heterogeneous systems is a tedious task. One has to map the parts of an application for execution on a CPU and / or on a GPU. This mapping has to be dynamic so that it can adapt to different heterogeneous systems characterized by different ratios between the computational speed of the GPU and the one of the CPU. Offloading a routine for execution on the GPU can result in a situation in which the GPU is busy computing and during this time the CPU is idle. This makes sense if the CPU's performance for a given routine and system is not comparable with the one delivered by the GPU. Nevertheless, portability problems occur on other systems that are more balanced, e.g. contain more CPU cores. Here, the contribution of the CPU to the performance cannot be neglected. In order to achieve both performance and portability, heterogeneous programs must be able to adapt the mapping between routines and processors according to the underlying system.

There are two main requirements for writing programs for heterogeneous systems. First, in order to support a wide range of heterogeneous systems in a portable way, a developer should provide multiple versions of a function for each processor type available in the system. For multi-core CPUs, OpenMP is the de facto programming model. Nvidia GPUs on the other hand are best programmed using CUDA [14]. OpenCL [15] targets both CPUs and GPUs but even in this case, in order to achieve optimal performance, multiple versions are necessary. Second, adequate distribution of the work between the CPU and the GPU plays a crucial role in obtaining portable performance. This helps to avoid having idle processors.

In this chapter, load balancing is applied to the computational steering application described in Chapter 3. The goal is to accelerate the routines in this application, i.e. *sghierarch*, *sginterp*, and *spvm*, by exploiting the full computation power of heterogeneous systems. This can be achieved if the computation is distributed between the CPU and the GPU. A typical realization of load balancing relies on a dynamic approach in which the work is (over)decomposed at runtime into tasks of a fixed size (grain size). The tasks are grabbed for execution by the CPU and the GPU. Here, a challenge is represented by the scheduling's overhead which restricts the grain size. Moreover, the so-called grain size problem refers to finding a task size that is optimal with respect to both the CPU (e.g. a multiple of the tile size used for cache blocking) and the GPU (e.g. a multiple of the maximum number of concurrent threads). Dynamic load balancing is compared to a static approach in which the work is distributed at the beginning of the computation taking into account the computational speeds of the heterogeneous components. This approach lacks the problems of dynamic load balancing but it is less adaptive on a non-dedicated system shared by multiple running applications. More importantly, it cannot cope with situations in which the parallelism is irregular. Moreover, determining the speeds of the CPU and of the GPU poses further challenges as they generally vary across systems and applications, and even across different values assigned to the input parameters.

6.2 Existing Load Balancing Solutions

One of the notable examples of numerical libraries that include load balancing for heterogeneous systems is *MAGMA* [87]. In *MAGMA*, the goal is to provide routines for dense linear algebra that exploit the full computation power of modern systems that contain multi-core CPUs and GPUs. In this chapter, the work follows the same philosophy as *MAGMA* in the sense that load balancing is used to improve the performance of the *fastsg* library described in Chapter 3.

The frameworks described in [88, 89, 90, 91, 92, 93] are more general purpose and cope with heterogeneity by incorporating a runtime system that provides various options for load balancing by scheduling tasks. Most scheduling schemes are based on greedy algorithms in which scheduling decision are not revisited as opposed to more expensive approaches based on back-tracking. Besides load balancing, these solutions also aim at simplifying the management of data transferred to / from the GPU over PCIe. Since nowadays heterogeneous systems are typically distributed memory systems, coherence is software controlled. Although the terminology varies across frameworks, the principles are the same, borrowed from instruction scheduling in modern out-of-order processors: Tasks can be executed once their input dependencies are satisfied, not necessarily in the order in which they were submitted to the runtime system. Moreover, some frameworks also implement techniques such as speculation and register renaming. Therefore, the frameworks share in general the following properties:

- There are multiple implementations of a task for each type of processor.
- The arguments of a task have one of the attributes: *read*, *write*, or *read-write*.
- The arguments are logically (not physically) placed in a shared memory space.
- The tasks are launched asynchronously and their order together with their arguments implicitly form a Dependency Acyclic Graph (DAG).
- As soon as tasks in the DAG become ready for execution, they are either inserted in a central queue or in the individual queues of the processors.

- Data is moved automatically as required by the chosen schedule.

At the core of *StarPU* [88] and *StarSs* [91], there is the task concept. A feature of *StarPU* is that it uses a history based model [94] for estimating the execution time which allows for better scheduling decisions to be made as part of a Heterogeneous Earliest Finish Time (HEFT) scheme [95]. *Harmony* [89] is similar to *StarPU* and *StarSs* but in addition it allows for asynchronous control structures and speculation (branch prediction). *Anthill* [92] is a data-flow framework in which a program is written as a composition of filters that transform and pass data to each other. *Merge* [90] applies the map-reduce paradigm to heterogeneous computing, providing also load balancing across multiple types of processors.

The *Qilin* framework [93] differentiates itself from the rest of the frameworks presented here through the fact that it does not use tasks. By reducing its scope to data parallelism, load balancing is achieved in *Qilin* through data partitioning. Hence, the work is divided into two pieces of different sizes which are assigned to the CPU and the GPU and are inversely proportional to the predicted execution time for the CPU and the GPU.

In task based programming, choosing the best performing grain size is often challenging. In this chapter, two approaches are used in order to tackle this problem: (1) multi-grain dynamic load balancing in which there are two grain sizes, for the CPU and the GPU, which are adjusted according to the remaining unprocessed work, and (2) static load balancing which means in the chapter that the computation is divided into two work pieces whose sizes are proportional to the computational speeds of the CPU and the GPU for the given computation.

6.3 Considerations

6.3.1 Multi-versioning

Multi-versioning is an essential requirement for coping with heterogeneity. Although there are programming languages, e.g. OpenCL, which can be used to program both CPUs and GPUs, the two types of processors are radically different. For most routines, it is still necessary to optimize them individually for each type of processor. Therefore, the same function (algorithm) often has multiple implementations or versions. For instance, the CPU version is tuned for the best use of caches and vector units and may use OpenMP for parallelization. On the other hand, the GPU version implemented in CUDA typically includes optimizations that maximize the utilization of the GPU (occupancy, register file, etc.), coalesce the accesses to the global memory of the GPU, eliminate or reduce bank conflicts at shared memory level, minimize the number of branches that result in divergent warps, and make proper use of the flexible memory hierarchy found on the GPU (register file, L1 cache / shared memory, constant memory, texture memory, global memory).

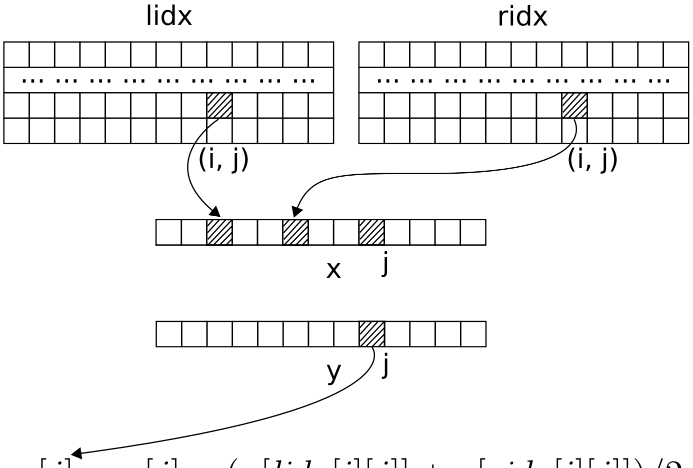
A first reason behind multi-versioning is portability. In general, when programming accelerators an offloading approach is employed in which the GPU is seen as a co-processor for specific tasks. In this context, a simple scheme is to create a mapping between each routine and the type of processor on which it performs the best, i.e. the execution time is minimized. Even if the GPU seems to be the best option for a given routine on a specific heterogeneous system, the situation may change on another system. It is thus important to provide a fallback solution to cover either the case in which there is no GPU on the second system or the case in which there is a different ratio between the computational speed of the GPU and the one of the CPU.

Using only the best processor type for a given routine is a scheme that in general limits the amount of parallelism in a heterogeneous system. The risk is in fact that some computation resources may be idle as each routine is executed by only one type of processor, e.g. the CPU is idle while the GPU executes a GPU friendly routine. The solution is to move from offloading to full distribution of the computation performed by a routine. In such a scheme the main challenge is realizing the cooperation between the CPU and the GPU for computing each routine. Since this cooperation implies that data is exchanged over PCIe between the memory of the CPU and the one of the GPU, simultaneously using the CPU and the GPU is not always feasible for every function of an application, e.g. when data access patterns are complex and suffer from lack of locality, thus they cannot be predicted at the time when the computation is distributed to the CPU and the GPU. This situation is described in this chapter for *sghierarch*. The other two kernels, *sginterp* and *spvm*, allow for the distribution of the computation and the data.

6.3.2 Assigning Work to Processors: *to-one* and *to-all*

The work performed by some routines cannot be distributed across a heterogeneous system, e.g. because of too much communication over PCIe which might be necessary in order to keep the data consistent. In this case, a heterogeneous system is often underutilized and the GPU acts as a coprocessor. In the rest of the chapter, this method of assigning work to a heterogeneous is referred to as *to-one* since the work is processed within one processor domain which is defined as the set of processors that share the same memory space. In this context, the PCIe bus can be seen as separator between the CPU domain and the GPU domain. As a more concrete example, *to-one* is explained for the *sghierarch* routine.

In the *sghierarch* routine, the sparse grid is stored in memory as a 1-dimensional (1d) dense array x. The sparse grid is characterized by a number of dimensions and a refinement level, d and l respectively. In a loop that iterates from 1 to d, each value of the sparse grid is updated using



$$y[j] = x[j] - (x[lidx[i][j]] + x[ridx[i][j]])/2$$

Figure 6.1: Simplification of *sghierarch*. It shows how the vector y is computed in iteration i, where i is from 1 to d. The marked components are used for the calculation of $y[j]$.

the current value and two dependencies from the same sparse grid. For simplicity and without loss of generality, consider that the indices of the dependencies are stored in two matrices with d rows, *lidx* and *ridx*. In reality however, these matrices are not stored in memory but their values are computed on-the-fly, i.e. *lidx* and *ridx* are in fact functions executed entirely on-chip without referencing memory (they execute approximately l operations, where l is typically in the order of 10). Thus, a simplified version of the update performed in *sghierarch* consists in the operation $y[j] = x[j] - (x[lidx[i][j]] + x[ridx[i][j]])/2$, where j iterates over all the values in y for every i in the range from 1 to d. y, x, *lidx* and *ridx* have the same number of columns, i.e. the number of points of the sparse grid. This computation is displayed in Fig. 6.1. For completeness, note that after every i-th iteration, y swaps with x. Hence, the final result is x.

The parallelization of the computation from Fig. 6.1 is achieved on a shared memory system by distributing the components of y across threads. On distributed memory systems such as a heterogeneous system containing CPUs and GPUs, there are two problems of high importance: (1) the indirection done through the arrays *lidx* and *ridx* and (2) the swap between y and x after each iteration i. Distributing the computation of y across the CPU and the GPU means that the entire x must be kept consistent in both memories, of the CPU and of the GPU, since it is not possible to transfer to the GPU only that part from x actually needed by the computation assigned to the GPU. This results from the fact that the exact components of x needed by the GPU are not known at the time when the distribution is performed but only during the actual computation. Simultaneously using the CPU and the GPU for *sghierarch* requires in fact that the entire y is exchanged over PCIe after each iteration i in order for the data needed in the next iteration to be consistent. More precisely, the GPU is sent the part of y previously updated by the CPU. Reversely, the CPU receives the updated part of y from the GPU. Performing such transfers over PCIe is inefficient. Note that the bandwidth of PCIe is typically one order of magnitude less than the bandwidth of the GPU memory. Therefore, for efficiency reasons, *sghierarch* cannot involve in its computation all the processors in a heterogeneous system. A better alternative is a *to-one* scheme of assigning the computation to only one type of processors that share a physical memory space.

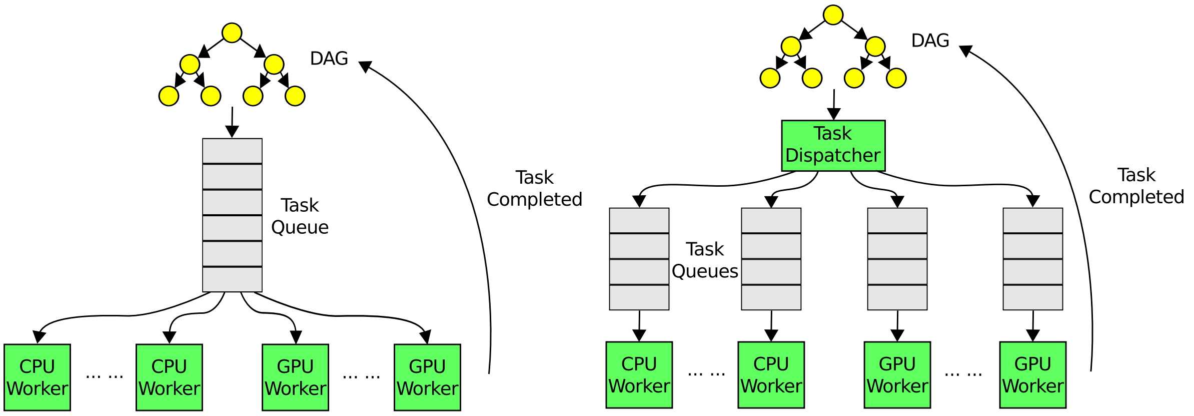
In contrast to the *to-one* work assignment, *to-all* means that the computation involves all the processors in a heterogeneous system. This applies to the *sginterp* and the *spvm* functions. For *sginterp*, interpolation points are distributed across processors and each processor returns the results for its assigned points. The sparse grid is copied entirely in the memory of the GPU. In the case of *spvm*, each processor computes a part of the output vector. More details are provided further in the chapter. For now, it is worthwhile mentioning that the memory access patterns in these two routines are regular and do not pose the difficulties of *sghierarch* for which an entire array has to be exchanged over PCIe to achieve the required consistency.

6.4 Task Based Scheduling for Heterogeneous Systems

6.4.1 Dynamic Task Based Scheduling

In dynamic task based scheduling schemes, the work is (over)decomposed into chunks whose number is ideally significantly larger, e.g. orders of magnitude, than the number of processors in the system. An abundance of work chunks allows the computation resources to be assigned an amount of work proportional to their processing speeds, i.e. the faster a processor, the more work it gets. In a dynamic scheme, scheduling is performed during execution, meaning that the tasks are scheduled no sooner than when their dependencies are satisfied. For this reason, dynamic scheduling targets especially the case of non-deterministic tasks defined as tasks whose duration is not known or cannot be estimated prior to execution time. This is the most general situation for scheduling and occurs for instance when the tasks are data sensitive or incorporate a complex control flow, i.e. if-statements whose conditions cannot be predicted before execution. Dynamic scheduling can also cope with the volatile conditions of a non-dedicated system.

The difficulty to estimate the duration of the tasks in the dependency graph, a directed acyclic graph (DAG), adds more complexity to the scheduling problem. General heuristics are often based on list based scheduling [96]. In list scheduling, the tasks are assigned priorities which provide an order relation for available tasks (ready for execution) which can be assigned to processors. As soon as the tasks become available, they are scheduled in the order of their priorities. The priority of a task can be given by its distance to a terminal task in the DAG or



(a) Using a global task queue.

(b) Using multiple queues and a dispatcher.

Figure 6.2: Dynamic load balancing schemes.

by its number of successors.

Dynamic scheduling has many forms: It can use a single queue or multiple queues, can be centralized or decentralized, can be triggered by an underloaded processor (receiver initiated) or by an overloaded processor (sender initiated). For a more comprehensive view of dynamic load balancing schemes, the reader is referred to [97].

Fig. 6.2 depicts dynamic scheduling using a single queue and multiple queues. In Fig. 6.2a, ready tasks whose dependencies are all satisfied, are pushed to the task queue. From the queue, a task is grabbed by a worker thread which executes the task either on the CPU or on the GPU. After the execution of the task completes, a signal is sent to the DAG in order to mark that the data needed by the successors of the task in the DAG is now available. This is done by incrementing a counter per task which initially has the value 0. The task becomes ready when the counter matches the number of dependencies, i.e. parent tasks. If all the dependencies of a child task in the DAG are satisfied then it is inserted at the back of the task queue.

From an implementation point of view, a task is defined as a collection of input data, output data, and a multi-version function. If a task is extracted from the queue by a CPU worker, then the CPU routine is invoked. Otherwise, the GPU worker calls the GPU routine. Since the CPU and the GPU have non-coherent memories, sometimes the input data needs to be transferred over PCIe in order to ensure consistency. In most task based frameworks, the data copy to / from the GPU is managed transparently by a runtime system.

The single queue design has the disadvantage that the worker threads access concurrently one task queue. The needed synchronization is a source of overhead especially for fine-grained tasks. Furthermore, the contention's negative impact on performance is proportional to the number of workers. In a design based on multiple queues, after a task becomes ready it is routed to one of the queues by a *dispatcher* component. A random scheme can be used for routing. A variation is to combine random with a computational speed proportionate selection of workers, i.e. the higher the speed of a processor, the higher the probability that its queue is selected as destination. In a work stealing [98] based scheduling scheme, if a worker executes a pop operation on an empty queue then it becomes a thief, chooses a victim worker from which it steals one or more tasks. The decentralized design allows in this case for improved scalability.

6.4.2 Scheduling Deterministic Tasks

Deterministic tasks are defined as tasks whose duration can be estimated accurately before execution [96]. In a heterogeneous system, this implies that for every task the execution time is known for both the CPU and the GPU routines associated with the task. Using the execution time and the data transfer time over PCIe, the DAG can be scheduled statically, prior to execution time. The advantage is that better schedules can be built but this requires a dedicated system. In contrast to dynamic scheduling, there is no queue contention in this case. Each worker processes the tasks assigned to it by a non-backtracking scheduling heuristic. A worker communicates the completion of a task to workers whose tasks are children of the finalized task.

In [88, 99] the authors use HEFT scheduling for assigning (deterministic) tasks to processors. They employ a history based approach [94] for estimating the duration of tasks. The history can be built during calibration runs and is used to feed a regression model that approximates the execution time of a task on different types of processors, e.g. CPU and GPU, as a function of the task size. The task size is a scalar value calculated through a linearization of the parameters of a task. The selection of a queue where a task is submitted is controlled by the equation:

$$i = \underset{k}{\texttt{argmin}}(\mathbf{tq}_k + \mathbf{te}_{j,k} + \mathbf{td}_{j,k}), \tag{6.1}$$

where j refers to the current task, k is the identifier of a queue (or worker) and is used to iterate over all the queues (workers), **tq** is the time when the queue is expected to become empty, **te** is the estimated duration of the task on processor k, and **td** is the time necessary to transfer the input data needed by the task to processor k. After the best destination queue i is chosen, tq_i is updated to $tq_i := tq_i + te_{j,i} + td_{j,i}$. It is worthwhile mentioning that $td_{j,k}$ is 0 if the input data already resides in the memory of processor k. The data copy over PCIe takes place only when the input data of the task is in the memory of the CPU and the chosen worker is a GPU worker, or vice versa. The data consistency across the PCIe border can be achieved through a software implementation of the Modified Shared Invalid (MSI) protocol as shown in [88].

6.4.3 The Grain Size Problem

Choosing the optimal grain (or task) size is a well-known problem in scheduling [96]. It revolves around the trade-off between parallelism and locality. A small grain size results in a larger number of tasks which allows for more processors to be used in the computation. However, the cost is increased overhead in the form of communication delay and scheduling time. In the case of heterogeneous systems, the problem is described in [99] for LU decomposition. There, the authors decrease the task size towards the end of the computation in order to allow for both the CPU and the GPU to process tasks simultaneously, i.e. the objective is to avoid having idle processors. In [100], the problem in further refined, focusing on the implications of the grain size on the optimizations applied to the CPU and the GPU implementations of a given algorithm. Consequently, there are three main aspects that have to be taken into account when dealing with the grain size problem on heterogeneous systems: (1) parallelism, (2) scheduling and communication overheads, and (3) constraints because of specific optimizations targeting the CPU and the GPU.

With regard to parallelism, the grain size must ensure that all the processors can take part in the computation. Hence, a lower limit for the number of tasks is given by the number of processors. The number of tasks needs to exceed considerably this limit, e.g. factors of magnitude, in order to provide the scheduler with the flexibility necessary to cope with heterogeneous tasks, heterogeneous processors, and the volatile conditions of a non-dedicated system. Provided that the number of tasks is set by the programmer, its value can then be used to estimate the *maximum grain size* per task type.

In order to maximize parallelism, the grain size must be minimized so that there are more tasks, but this typically creates unacceptable scheduling and communication delays. The reduction of these overheads puts a *lower limit on the grain size* defined as the minimal task size for which the execution time of the task dominates, e.g. more than 100x, the duration of queue operations used in the schemes shown in Fig. 6.2. Similarly, the execution time of the task must also dominate the PCIe communication time. This is often accomplished by avoiding that the data transfers become latency bound. In other words, reducing the scheduling and the communication delays requires that the grain size is increased. Note that a too large grain size reduces the parallelism and is more prone to situations in which the slow processor grabs the last task although the fastest processor is about to finish its current task and its processing speed is considerably higher. In this scenario, the fastest processor is idle towards the end of the computation. The more coarse granular the tasks, the more severe the negative impact on the performance.

Another point that needs consideration is the correlation between the grain size and the

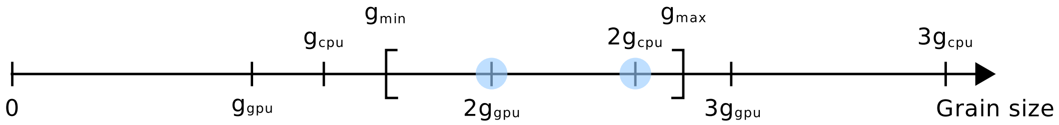


Figure 6.3: The constraints influencing the selection of the optimal grain size.

optimizations applied to the CPU and the GPU versions of a function executed by a task. As an example, a highly tuned CPU routine may include optimizations for caches, e.g. cache blocking. For optimal performance, the grain size preferred by the CPU should allow for the best exploitation of caches. This often translates to setting the grain size so that the size of the data processed by the task is a multiple of the optimal cache block (or considerably larger than the optimal cache block which reduces parallelism). The same observation applies also for the GPU routine, i.e. the grain size should ensure that the number of GPU threads is a multiple of the maximum number of active threads, B_{gpu} from Eq. 5.6.

Since GPUs are typically faster than CPUs, the effect of not fitting the grain size to the GPU optimizations is more severe. If the grain size is not a multiple of B_{gpu}, then there is a tail effect that can significantly reduce the performance of the GPU. Assume that the grain size is chosen so that the GPU routine is executed using $(n-1) \cdot B_{gpu} + r$ homogeneous threads, where $r \ll B_{gpu}$. The first $(n-1) \cdot B_{gpu}$ threads fully occupy the GPU. The tail effect results from the execution of the last r threads which cannot fill the entire GPU, thus reducing the speed of the GPU to fs, a fraction of s, e.g. $s/16$. Consequently, the average speed of the GPU is given by $s_{avg} = ((n-1) \cdot s + fs)/n$. The worst case scenario corresponds to $n = 1$. The increase of n, e.g. resulting from the increase of the grain size, limits the impact of fs, i.e. the tail effect. Another solution that completely eliminates the tail effect is to ensure that $r = 0$.

In some situations, the grain sizes preferred by the CPU and the GPU, g_{cpu} and g_{gpu} respectively, do not match. Moreover, most task based frameworks for heterogeneous programming do not efficiently support this scenario, meaning that the grain size is most of the time a constant scalar for all the tasks of the same type (the same multi-version function). Ideally, the grain size should be set to $SCM(g_{cpu}, g_{gpu})$, where SCM is the smallest common multiple. This can result in tasks that are too large and if the computation is not abundant, then the parallelism is reduced. Another approach is to prioritize by setting the grain size to either g_{cpu} or g_{gpu} depending on which type of processor is the fastest for the given task, i.e. the CPU or the GPU. The consequence is that the slower processor may no longer operate at full efficiency. This can have a significant negative impact on the performance of a heterogeneous system especially if the performance of the CPU is close to the performance of the GPU and the preferred grain sizes are radically different.

Fig. 6.3 shows all the restrictions resulting from parallelism, scheduling and communication overheads, and optimizations. The parallelism generates an upper limit for the grain size represented through g_{max}. The lower limit, g_{min}, results from the minimization of the scheduling and data transfer delays. Finally, g_{cpu} and g_{gpu} are the grain sizes preferred by the optimized CPU and GPU routines respectively. The two candidates for the best grain size are located in the interval $[g_{min}, g_{max}]$. If the GPU is faster than the CPU, then priority is given to $2 \cdot g_{gpu}$.

6.5 Scheduling Data Parallelism on GPU Based Systems

In a data parallel model, concurrent tasks execute similar or identical operations on data items that together form the initial data. In general, a data parallel computation contains a sequence of data parallel stages interleaved with synchronization phases [12]. Achieving load balancing for data parallelism on multi-core CPUs can be done efficiently using a uniform data partitioning in which each worker thread processes the same amount of data. This decomposition scheme cannot be employed on heterogeneous systems because of the different characteristics of GPUs compared to CPUs. In the presence of heterogeneity, data parallelism can still be managed using a task graph but this approach suffers from the grain size problem discussed before, thus leading to a suboptimal utilization of one type of processor. This section describes dynamic and static scheduling solutions for data parallelism that overcome the task size problem.

6.5.1 Heterogeneous Dynamic Scheduling for Data Parallelism

On homogeneous processors the benefit of dynamic scheduling of data parallelism is given by the ability to cope with non-deterministic tasks and to adapt to the variable conditions of a non-dedicated system. OpenMP provides dynamic scheduling for assigning to threads the independent iterations of a for loop using *#pragma omp for schedule(dynamic).* Reductions are also allowed. The iteration space of the loop is divided into chunks which are dynamically assigned to threads, i.e. when a thread finishes its chunk, it is dynamically assigned a new one. A straightforward scheme that provides this functionality is shown in Fig. 6.2a. The parameter exposed by dynamic scheduling is the chunk size, the same as the grain size.

On heterogeneous systems, a single grain size often cannot result in the best utilization of both the CPU and the GPU. Moreover, in the absence of an abundance of parallelism, a slow processor can grab a task near the end of the computation while the fast processor is about to finish its current task. In this case, the fast processor is idle, a situation that should be avoided in order to reach optimal performance. These two observations lead to a first requirement that has to be satisfied by the dynamic scheduling of a data parallel computation on heterogeneous

Listing 6.1 Dynamic scheduling. Scope: one CPU worker and one GPU worker, global queue.

```
$\alpha$ = t_cpu / t_gpu    // assume GPU faster than CPU
stop = false; stream = 0
while not stop do
  if cpu_worker() then    // CPU worker
    lock.acquire()
    if queue.size() >= g_cpu + ceil($\alpha$) * g_gpu then
      w_cpu = queue.pop(g_cpu)
    else
      w_cpu = queue.pop(min(ceil($\alpha^{-1}$ * g_cpu), g_cpu, queue.size()))
    lock.release()
    stop = (w_cpu == 0)    // true if empty queue
    if not stop then cpu_routine(w_cpu)
  else    // GPU worker
    lock.acquire()
    w_gpu = queue.pop(min(g_gpu, queue.size()))
    lock.release()
    stop = (w_gpu == 0)    // true if empty queue
    if not stop then
      sync(stream)
      gpu_routine(w_gpu, stream)
      stream = (stream + 1) % num_streams
```

systems: The CPU and the GPU routines must process their optimal or preferred grain sizes whenever the amount of remaining work permits it. The second requirement is that the fast processor should not wait for the slow processor to finish a task. This can be achieved by scaling down the grain size for the slow processor. These requirements are addressed in Listing 6.1.

Listing 6.1 depicts a grain size aware load balancing algorithm for data parallelism. For simplicity but without loss of generality, the case of one CPU core and one GPU is considered. Furthermore, it is assumed that the GPU is faster than the CPU for the given computation. Load balancing is based on a single queue scheme in which when a worker finishes its task, it immediately grabs a new task from the queue. The parameter of the *pop(n)* routine does not only extracts a task from the queue but also creates a task of size n.

Four main parameters control the behavior of load balancing: *t_cpu* (1), the time necessary to execute the grain size preferred by the CPU, *g_cpu* (2), and *t_gpu* (3), the time taken by the execution of *g_gpu* (4) on the GPU. *t_cpu* and *t_gpu* are obtained in a calibration step performed prior to execution and are used to scale down *g_cpu*. Besides them, *num_streams* controls the overlapping of GPU computation and PCIe communication. The assumption that the GPU is faster than the CPU does no imply that $t\_gpu < t\_cpu$ but rather that $g\_gpu/t\_gpu > g\_cpu/t\_cpu$ (computational speeds). The variables *w_cpu* and *w_gpu* represent the current task sizes which may differ from the preferred grain sizes depending on the status of the queue, i.e.

abundant work or almost empty. Provided that the remaining unprocessed work in the queue is abundant, $w_{cpu} = g_{cpu}$ and $w_{gpu} = g_{gpu}$.

The scaling down of the grain size of the slow processor, *w_cpu*, is triggered by the status of the queue, i.e. the remaining unprocessed work. The objective is to avoid that the fast processor, here the GPU, is idle at any time. This gives the condition of the if-statement from line 6. More precisely, line 6 checks if there is enough work so that the slow processor, the CPU, can process its preferred grain size, *g_cpu*, and during this time the GPU is guaranteed to be busy. While the CPU executes *g_cpu*, the GPU can execute approximately $\alpha \cdot g\_gpu$, where α is defined in line 1. Hence, the condition is that the remaining work in the queue is bigger than $g\_cpu + \lceil\alpha\rceil \cdot g\_gpu$. If the work in the queue is insufficient for this condition to hold, *g_cpu* is scaled down in line 9. The reason behind multiplying $\alpha^{-1} \cdot g\_cpu$ is to ensure that the duration of *w_cpu* is on par with *w_gpu*, i.e. the GPU does not wait for the CPU to finish.

The scheduling algorithm from Listing 6.1 ensures both that the preferred grain sizes are used whenever there is enough parallel work in the queue, and that both processors are in use at any time. The case of multiple CPU cores and GPUs is addressed by multiplying *g_cpu* and *g_gpu* with the number of CPU cores and the number of GPUs respectively.

An optimization applied to Listing 6.1 is the overlapping of PCIe communication and computation. This is shown in lines 19 - 21. Typically, *gpu_routine* copies input data to the GPU, launches a CUDA kernel, and copies output data to the CPU. Current Nvidia GPUs [11] support the parallel execution of a CUDA kernel, a data copy to the GPU, and a data copy from the GPU. This is depicted in Fig. 6.4. Hence, while a kernel processes the current input data, the next input data is simultaneously transferred to the GPU, and a previous output data is copied to the CPU. In CUDA, the overlapping of data transfers and computation is realized using streams which allow for operations submitted to different streams to execute concurrently.

In Fig. 6.4, overlapping enables the GPU to work almost continuously. In this theoretical example, the execution using 3 streams can be up to 3x faster than when 1 stream is used. In the case of load balancing, every time a GPU worker extracts a task from the queue, a stream is chosen in a round-robin manner, the worker waits for the operations (data copy to GPU, kernel execution, and data copy from GPU) of the previous task sent to that stream to finish, and subsequently submits the operations corresponding to the new task to the stream.

6.5.2 Heterogeneous Static Scheduling for Data Parallelism

Static scheduling for data parallelism is realized by decomposing the data into chunks whose number equals the number of worker threads. On homogeneous systems, workers execute the same operations on chunks of equal sizes. In OpenMP, static scheduling can be used to distribute

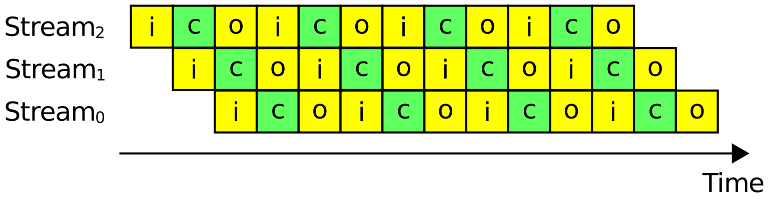


Figure 6.4: The benefits of using 3 streams in the GPU worker thread. *i* / *o* means data copy to / from the GPU. *c* refers to computation done by the GPU.

the iterations of a for-loop among threads. However, this simple form of scheduling is not possible on heterogeneous systems due to the different computational speeds of the processors. In the presence of heterogeneity, one expects that the faster a processor, the larger its chunk is.

On heterogeneous systems, the chunk sizes are directly proportional to the processing speeds (for a given routine) or inversely proportional to the execution time. Consider that the execution time of a generic work of size w_c on the CPU is $t_{cpu}(w_c)$ while the GPU executes the same amount of work in $t_{gpu}(w_c)$ time. Assuming that an arbitrary w_a is decomposed into two partitions, w_{cpu} for the CPU and w_{gpu} for the GPU, the execution time is given by $t_{het} = \max(t_{cpu}(w_{cpu}), t_{gpu}(w_{gpu}))$. For optimal performance, the goal is to find w_{cpu} and w_{gpu} that together minimize t_{het}. Since $w_{gpu} = w_a - w_{cpu}$, this translates to determining:

$$w_{cpu} = \underset{w}{\operatorname{argmin}} \max(t_{cpu}(w), t_{gpu}(w_a - w)) \tag{6.2}$$

Here, $t_{cpu}(w)$ and $t_{gpu}(w_a - w)$ are not actually known. They can still be approximated from $t_{cpu}(w_c)$ and $t_{gpu}(w_c)$. Therefore, $t_{cpu}(w) = (w/w_c) \cdot t_{cpu}(w_c)$ and $t_{gpu}(w) = (w/w_c) \cdot t_{gpu}(w_c)$. In these conditions, the equations for calculating w_{cpu} and w_{gpu} are:

$$\begin{aligned} w_{cpu} &= \frac{w_a}{1+\beta} \\ w_{gpu} &= \frac{w_a}{1+\beta^{-1}} \end{aligned} \tag{6.3}$$

where $\beta = t_{cpu}(w_c)/t_{gpu}(w_c)$. In order to determine the amount of work per CPU core, w_{cpu} is then divided to the number of cores, n_{cpu}. The same procedure is followed for deciding the work assigned to a GPU in a multi-GPU system containing n_{gpu} identical GPUs.

In general, choosing the right value for w_c is not an easy task as it must ensure that the approximation of the execution time must be accurate for all the values of w_a. The difficulty results from the fact that the time is not linear relative to work but is an increasing function containing steps, especially on the GPU because of the large number of cores and the wide

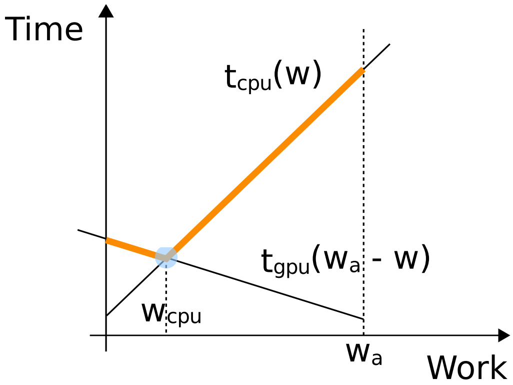


Figure 6.5: Static load balancing resulting from the intersection of two linear functions.

SIMD units (see Fig. 5.1 from Chapter 5). Determining w_c must take this aspect into account, e.g. by sampling at the end of a step. Otherwise, a tail effect may occur that limits the ability to capture the global behavior of the execution time depending on work.

In order to address tail effects, one can increase w_c based on the observation that the larger the value of w_c, the lower the impact of a tail effect. The disadvantage is that the duration of the calibration phase needed for obtaining the execution time for w_c, also increases. In this regard, a better approach relies on replacing w_c with two parameters, one for the CPU, w_{ccpu}, and one for the GPU, w_{cgpu}. The objective is to completely eliminate tail effects. As an example, the value of w_{ccpu} is chosen in accordance to the best cache utilization on the CPU while the value of w_{cgpu} ensures that the number of launched GPU threads is a multiple of the maximum number of active threads, B_{gpu} from Eq. 5.6.

The parameters w_{ccpu} and w_{cgpu} are used for approximating $t_{cpu}(w)$ and $t_{gpu}(w)$ using linear functions of work. Intuitively, these functions should pass through the point $(0, 0)$ signifying that when the work is 0, the execution time is also 0. However, this assumption is not always valid as it excludes overheads that occur only once, at the beginning and at the end of the computation, and that do not scale with work, e.g. data transfers. Therefore, in order to improve the quality of the linear approximations, at least two points for each type of processor must be used, more precisely the execution time measured for $\{w_{ccpu}, 2\,w_{ccpu}\}$ and $\{w_{cgpu}, 2\,w_{cgpu}\}$ for the CPU and the GPU respectively. The resulting linear functions are then given by:

$$\begin{aligned}
t_{cpu}(w) &= \overbrace{\frac{t_{cpu}(2\,w_{ccpu}) - t_{cpu}(w_{ccpu})}{w_{ccpu}}}^{\gamma_{cpu}} \cdot w + \overbrace{2\,t_{cpu}(w_{ccpu}) - t_{cpu}(2\,w_{ccpu})}^{\delta_{cpu}} \\
t_{gpu}(w) &= \underbrace{\frac{t_{gpu}(2\,w_{cgpu}) - t_{gpu}(w_{cgpu})}{w_{cgpu}}}_{\gamma_{gpu}} \cdot w + \underbrace{2\,t_{gpu}(w_{cgpu}) - t_{gpu}(2\,w_{cgpu})}_{\delta_{gpu}}
\end{aligned} \tag{6.4}$$

In this context, the solution of Eq. 6.2, w_{cpu}, is determined for a given work w_a by intersecting $t_{cpu}(w)$ with $t_{gpu}(w_a - w)$ as depicted in Fig. 6.5. The highlighted lines from Fig. 6.5 form one function whose minimization results in w_{cpu}. Hence, the equations for w_{cpu} and w_{gpu} are:

$$\begin{aligned}
w_{cpu} &= \frac{\delta_{gpu} - \delta_{cpu} + \gamma_{gpu} \cdot w_a}{\gamma_{cpu} + \gamma_{gpu}} \\
w_{gpu} &= \frac{\delta_{cpu} - \delta_{gpu} + \gamma_{cpu} \cdot w_a}{\gamma_{cpu} + \gamma_{gpu}}
\end{aligned} \tag{6.5}$$

In theory, if $w_{cpu} \leq 0$, then $w_{gpu} = w_a$, i.e. the GPU executes the entire work while the CPU is idle. A situation more common in practice corresponds to $w_{cpu} \geq w_a$, meaning that the CPU computes the whole work, i.e. $w_{cpu} = w_a$, while the GPU is idle. Such a scenario can be generated when there is a low computation to PCIe communication ratio. In order to calculate the work assigned to one CPU core and one GPU, w_{cpu} and w_{gpu} are divided by n_{cpu} (the number of CPU cores) and n_{gpu} (the number of GPUs) respectively.

6.5.3 Comparison

In dynamic scheduling of data parallelism on heterogeneous systems, choosing the right grain size is of high importance. In order to allow for the best utilization of both the CPU and the GPU, the grain size is replaced by two grain sizes, one for each type of processor, meaning that the CPU and the GPU have different grain sizes. In general, the grain sizes must result in optimal utilization of the cache of the CPU and of the massive parallelism (cores and multithreading) of the GPU. Moreover, if the grain sizes are too small, then the performance is drastically reduced by synchronization and data transfer overheads. If the grain sizes are too large, then load balancing cannot be achieved as the fast processors can be idle while waiting for the slow processors to finish. The input parameters of a program often influence the performance ratio between the CPU and the GPU, and the optimal grain sizes. Hence, dynamic scheduling needs to be input aware. A first advantage of dynamic scheduling is that it addresses the scenario of non-deterministic tasks which can only be assigned to processors at runtime. Another advantage is that dynamic scheduling adapts to the variable conditions of a non-dedicated system. Third, it allows for overlapping the PCIe transfers with the computation done on the GPU. Fourth, an MSI protocol can be used to automatically overcome the limited memory on the GPU, i.e. data transfers are managed transparently by a runtime system. Data on the GPU is evicted automatically whenever there is not enough free memory for copying the data needed for the execution of the current task.

In contrast to the dynamic approach for scheduling, heterogeneous static scheduling of data parallelism does no suffer from the grain size problem. The data is decomposed prior to execution into chunks, one per processor. The size of a chunk is proportional to the computational speed of the processor to which it is assigned. Regarding the dependency on input, the parameters of static scheduling need to be recalibrated whenever new input parameters are presented to the program. A primary disadvantage is that static scheduling cannot adapt to a non-dedicated heterogeneous system on which there can be multiple running programs. Moreover, overlapping PCIe communication with processing on the GPU is not a direct benefit and can be achieved by dividing the chunks into smaller pieces although this introduces the grain size problem. Similarly, coping with the small amount of memory on the GPU needs to be explicitly handled.

6.6 Evaluation

This section describes the performance results obtained using the load balancing techniques covered in this chapter. Three case studies are presented: *sghierarch*, *sginterp*, and *spvm*.

6.6.1 Experimental Setup

With regard to the experimental setup, the results are obtained on a dual-socket system containing 4 Intel Nehlaem cores (2.53 GHz) per socket. Hyperthreading is enabled. The CPUs are complemented by a Quadro 6000 GPU containing 16 cores operating at a frequency of 1.15 GHz. Each GPU core incorporates a 32-lane single precision SIMD unit.

On the software side, gcc 4.4.5 is used for compiling the host code while for the GPU part CUDA 4.2 is used. Similarly to the setup from Chapter 4 and Chapter 5, *sghierarch*, *sginterp*, and *spvm* operate with single-precision floating point numbers. StarPU 4.0 is used as the reference implementation of dynamic task based scheduling. From the multitude of scheduling algorithms, the one used in the tests is a HEFT algorithm that relies on history and performance modeling for approximating the duration of a task on the CPU and on the GPU.

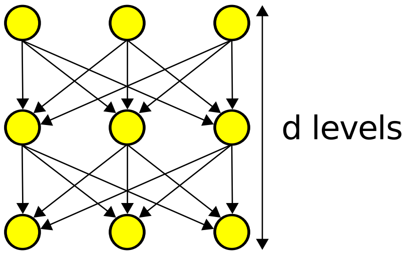


Figure 6.6: The dependency task graph for *sghierarch*.

6.6.2 Sparse Grid Hierarchization

In terms of tasks, *sghierarch*'s computation can be represented as shown in Fig. 6.6. Each task consists in updating a subset of values from the linear representation of a sparse grid. One can see there that the computation is done in d iterations. However, after each iteration, the modified data has to be made available to all the tasks in the next iteration. On a heterogeneous system, this implies that the entire data has to moved over PCIe: The CPU needs the data updated by the tasks assigned to the GPU whereas the GPU is sent the data modified by the CPU. Given the low computational intensity of *sghierarch*, such a data transfer severely decreases the performance of the routine. Consequently, the dependency graph together with the low computational intensity do not allow for a *to-all* mapping between the tasks of *sghierarch* and the heterogeneous processors, the CPU and the GPU.

For *sghierarch*, the question is whether the GPU is the best processor across all combinations of input parameters. Intuitively, the larger the d parameter, the smaller the impact of transferring the data over PCIe. More precisely, the data is first copied from CPU memory to GPU memory. On the GPU, *sghierarch* writes to each location d times and reads each value $2 \cdot d$ times. This makes the transfer over PCIe more worthwhile for a larger d.

In a *to-one* mapping between the computational load of *sghierarch* and the processors, the objective is to determine a map that specifies for different values assigned to the input parameters which version of *sghierarch*, for the CPU or for the GPU, delivers the best performance. For some applications, such a map can be determined by means of a performance model that includes the PCIe transfer costs and an approximation of the execution times on the CPU and on the GPU. Since *sghierarch* is characterized by a rather complex memory access pattern, it is difficult to create an accurate approximation of the execution time depending on the input parameters. Therefore, one can employ an empirical method in which *sghierarch* is executed on the CPU and on the GPU for different input parameters, and for each parameter the ratio between the performance of the GPU and the one of the CPU is saved in the map.

In the case of *sghierarch*, the two input parameters are the number of dimensions, d, and

	# of dimensions									
# of levels	1	2	3	4	5	6	7	8	9	10
1	1.18	1.16	1.21	1.24	1.12	1.17	1.24	1.10	1.21	1.07
2	1.42	1.23	1.00	1.07	0.98	1.11	0.93	0.82	0.88	0.71
3	1.25	1.17	1.20	1.15	0.99	0.94	0.90	0.82	0.92	0.74
4	0.96	1.29	1.23	1.25	1.05	1.00	0.84	0.81	0.69	0.77
5	1.26	1.31	1.11	1.07	0.89	0.87	0.91	0.82	0.68	0.65
6	1.02	1.05	1.11	0.98	1.02	0.96	0.88	0.82	0.75	0.68
7	1.24	1.23	0.97	0.96	0.99	0.98	0.97	0.92	0.90	1.01
8	1.31	0.89	0.95	1.04	1.23	1.30	1.15	1.27	1.45	1.30
9	0.91	0.81	0.87	1.07	1.10	1.46	1.76	1.83	1.70	1.56
10	0.85	0.89	0.95	1.32	1.73	2.09	1.85	1.76	1.82	1.88

Figure 6.7: The *to-one* map for *sghierarch*. It contains the ratio between the GFlop/s on GPU and on CPU. For the highlighted cells, the GPU is at least 0.95x slower than the CPU.

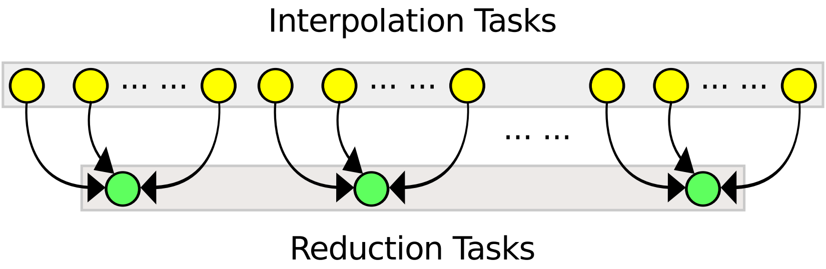


Figure 6.8: The task graph for *sginterp*. This decomposition is also applicable to *spvm*.

the refinement level, l. An example of a *to-one* map used for deciding which processor to use, is depicted in Fig. 6.7. In this figure, d and l are in the range from 1 to 10. For the highlighted entries, the CPU is faster than the GPU. In general, for *sghierarch*, the GPU outperforms the CPU but there are situations as seen in Fig. 6.7 in which the CPU is up to 1.53x faster than the GPU. The map can be built once at installation time or can be created on demand.

6.6.3 Sparse Grid Interpolation

The DAG for *sginterp* is depicted in Fig. 6.8. Each task refers to interpolating at some subset of points. The reduction tasks results from the division of the sparse grid in blocks (see Fig. 3.6). The division creates more fined-grained parallelism at the cost of the synchronization needed for reduction (sum). A task computes the contribution of a subset of the sparse grid blocks to the interpolation results for a subset of the interpolation points. Subsequently, all tasks interpolating at the same subset of points are reduced. The task graph for *sginterp* allows for a *to-all* distribution of the computation across all the processors in a heterogeneous system. For an abundance of interpolation points, a task computes the contribution of all the blocks for a subset of interpolation points, meaning that the reduction is no longer required. For now, consider that the coarse-grained parallelism is enough and reduction is not needed.

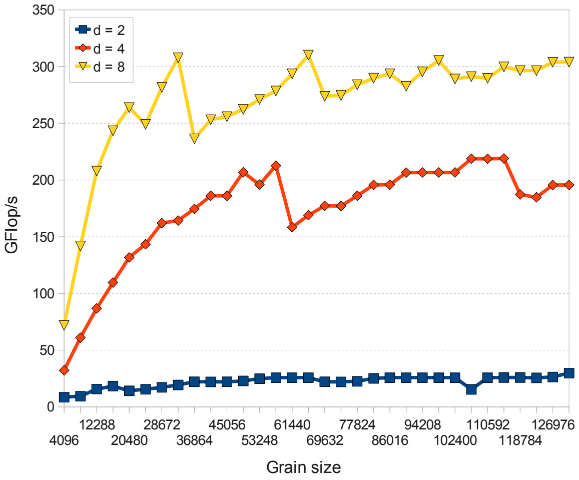
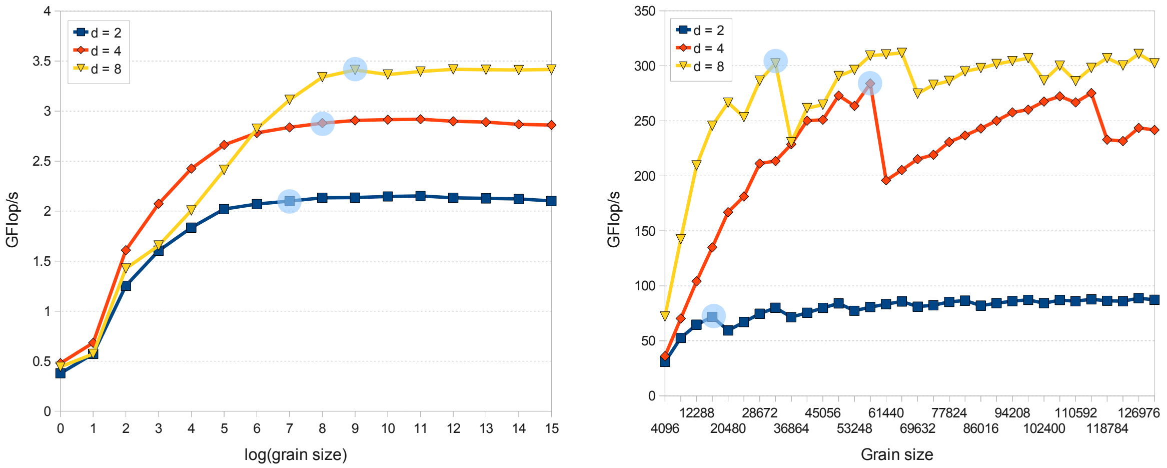


Figure 6.9: GFlop/s rate obained using StarPU as a function of grain size.

The performance of a task based implementation of *sginterp* is depicted in Fig. 6.9. The number of interpolation points is $2 \cdot 10^6$. StarPU is used for scheduling the work inside the heterogeneous system, across 8 CPU cores and 1 GPU. The StarPU implementation is a wrapper around the optimized version of *sginterp* for CPUs and the auto-tuned version of *sginterp* for GPUs. One can notice that the achieved GFlop/s rate depends highly on the grain size. This is in fact the grain size problem previously described. For $d = 8$, failing to determine the best grain size may result in a performance drop of up to 23%. The best performance is found for a grain size of 32768 and for multiples of this grain size. The larger the grain size, the less the impact of choosing a grain size. Nevertheless, one cannot ignore the benefits of a small grain size as this ensures more parallelism, thus a more efficient load balancing even for smaller problems. A notable aspect in Fig. 6.9 is that the best grain size is not invariant across different input parameters. This is explained by the interaction between the grain size and the optimization parameters for the two versions of *sginterp*, optimized for the CPU and for the GPU.

In Fig. 6.10a, the GFlop/s rate of the CPU version is shown as a function of the grain size. Only one CPU core is used. The grain size interacts in this case with an optimization targeted at better cache usage. The grain size directly affects a loop interchange transformation applied to the CPU version of *sginterp*. A grain size of 512 is sufficient for the performance of the CPU version to get close to the maximum GFlop/s rate achieved for a much larger grain size.

The dependency of the GPU's GFlop/s rate on the grain size is depicted in Fig. 6.10b. As shown in Chapter 5, the GPU optimization parameters also depend on the input parameters, especially on the d parameter. The consequence is that the minimal work necessary to fully occupy the GPU varies, especially for different values of the d parameter. Therefore, if this



(a) GFlop/s rate of the CPU version of *sginterp* for different grain sizes.

(b) GFlop/s rate of the CUDA version of *sginterp* depending on the grain size.

Figure 6.10: GFlop/s rate as a function of grain size.

aspect is not taken into account when choosing the optimal grain size, then a tail effect may occur, meaning that every time the GPU worker executes a task, either not all the SMs are engaged in the computation or multithreading is not fully harnessed. This is the main cause for the performance variation seen for different grain sizes. One can notice that the GFlop/s rate as a function of grain size obtained using StarPU is almost identical to the GFlop/s depicted for the GPU. This happens because for the given system and input parameters, the largest contribution to the performance comes from the GPU.

Regarding load balancing, the fact that the grain size preferred by the CPU is 512, hence rather small, is an advantage for load balancing since it eliminates conflicts with the grain size preferred by the GPU, which is up to two orders of magnitude larger. The optimal grain sizes for the GPU are shown in Table 6.1 for different values of d. In other words, setting the grain size for load balancing to the grain size preferred by the GPU does not sacrifice the performance of the CPU version of *sginterp*. The method for finding the optimal grain size for the GPU is based on determining the minimal number of threads that fully occupy the GPU, B_{gpu} in Eq. 5.6, and multiplying it by the number of interpolation points processed per GPU thread. Since this method does not involve benchmarking, it has minimal overhead.

It is important to observe based on Table 6.1 that for $d = 4$ and for $2 \cdot 10^6$ interpolation points, only 35 tasks are created. Intuitively, this does not look like overdecomposition on a system with 16 CPU workers (16 hardware threads) and 1 GPU worker. Moreover, in such a scenario, the GPU, in the role of the fastest processor, may have to wait for the CPU to

d	Best grain size
1	16384
2	16384
3	16384
4	57344
5	49152
6	40960
7	40960
8	32768
9	32768
10	32768

Table 6.1: The best grain sizes for the GPU version of *sginterp* for different values of d.

finish its tasks at the end of the computation. For optimal performance, having the GPU in an idle state should be avoided. There are three solutions to this problem: (1) to create more parallelism at the cost of more synchronization (from reduction tasks), (2) to employ the multi-grain algorithm from Listing 6.1, and (3) to use static load balancing in which there is no grain size problem. Point 3 requires linear approximations of the execution time as functions of the number of interpolation points as shown in Eq. 6.2. Regarding static load balancing for *sginterp*, the dependency of the GPU optimization parameters on the input parameter d requires that the linear approximation is also a function of d. Hence, the approximation has to be rebuilt whenever *sginterp* is invoked using a new d.

Fig. 6.11 depicts the performance of *sginterp* on the benchmarked system. The GFlop/s rate is measured for different number of dimensions, d. The number of interpolation points used in the performance tests is $2 \cdot 10^6$. Three major load balancing schemes are tested:

- *StarPU heft*: StarPU's HEFT scheduling which uses execution history and includes a model of the PCIe bus which is used to estimate the duration of data transfers.

- *dyn, dyn str, dyn str multi*: dynamic load balancing implemented using OpenMP. The *str* attribute means that 3 CUDA streams are used for overlapping the GPU computation with PCIe communication. *multi* means that there are 2 different grain sizes, i.e. one preferred by the CPU and one preferred by the GPU, controlled as shown in Listing 6.1.

- *sta*: static load balancing implemented using OpenMP based on Eq. 6.2.

The *max* line refers to the sum between the GFlop/s rate measured when the CPU handles the entire work by itself and the GFlop/s rate measured for the GPU only case. No tasks are used for *max*, only the optimized versions of *sginterp* which execute the work in one iteration.

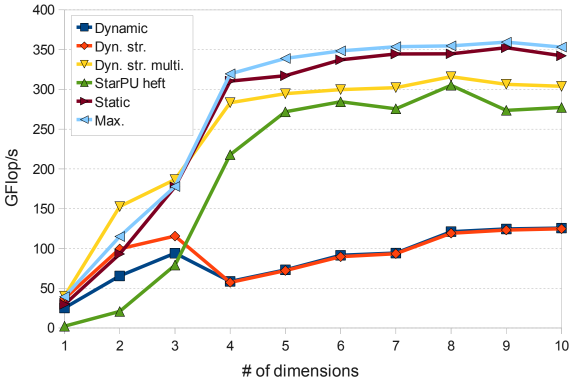


Figure 6.11: Load balancing of *sginterp*.

The most inefficient load balancing schemes are here *dyn* and *dyn str*. There is no notable difference between the two schemes for $d > 3$. However, for $d \leq 3$, the stream based load balancing scheme, *dyn str*, outperforms the more basic *dyn*. This is explained by the fact that for small values for d, *sginterp* tends to be more sensitive to memory performance, thus the PCIe transfer has more impact on the execution. The *str* optimization focuses exactly on hiding the PCIe overhead. Both these schemes are affected by the problem of an idle GPU. Since the number of tasks rarely exceeds 100 (only for d from 1 to 3) and the tasks are rather large since they have to fully occupy the GPU, there is a high probability that at some point during the computation the GPU worker is idle while the CPU workers are busy executing their assigned tasks. This undesired situation does not occur with *dyn str multi* and StarPU. The graph shows that for *sginterp*'s work, the *dyn str multi* scheme performs the best from the dynamic schemes, even better than StarPU. However, it is still rather far from the *max* performance.

The *sta* scheme is the closest to *max*. More precisely, its performance reaches approximatively 98% of the maximum performance obtained using *max*. Static load balancing is in fact the only scheme that beats the performance of the auto-tuned GPU version of *sginterp* for $d > 3$.

6.6.4 Sparse Vector - Matrix Multiplication

The same dependency graph from *sginterp* shown in Fig. 6.8 can also be used to explain the computation done in *spvm*. A task operates in this case on a tile from the output vector. For more parallelism, the matrix is divided horizontally into tiles. In this way, a task gets a vector tile from the output vector and a matrix tile from the input matrix. The task computes the contribution of the matrix tile on its assigned part from the output vector. In this decomposition scheme, more than one tasks may operate on the same tile of the output vector. Therefore, their individual contributions have to be reduced (summed up) by a reduction task.

In the tests presented next, the size of the output vector is 10^6 while the input matrix contains 250 rows, each of size 10^6. The sparse input vector contains a number of non-zeros in the range from 25 to 250 with a step of 25. It is assumed that the matrix can be transformed and distributed across memories, on different NUMA nodes or on the GPU, before the computation. This is in accordance with the conditions in a computational steering application in which the same matrix is used to compute the output vector for different sparse vectors. The load balancing schemes applied to *sginterp* are also applied to *spvm*.

The *spvm* kernel is memory bound. Since the benchmarked system is a NUMA system, improving the memory locality at a NUMA node level plays an important role. Moreover, hiding the overheads of data transfers over PCIe by overlapping computation and communication can also improve the performance considerably. In order to increase the efficiency of the transfers over PCIe, the experiments use page locked memory on the CPU side in order to decrease the number of memory copies. Without this optimization, the data is first copied to a page locked memory region and from there is transferred via DMA to the memory of the GPU.

When distributing the work in *spvm* using a dynamic load balancing scheme, *spvm* is decomposed into tasks. In order to execute efficiently on the CPU, experiments show that the grain size should be a multiple of 8192 which ensures the best cache utilization because of a loop blocking transformation. On the GPU side, the preferred grain size is 143360 which is the minimal work that ensures full utilization of both the SMs on the GPU and of multithreading.

Assuming that the input matrix is not tiled horizontally in order to create more parallelism, there are only 7 tasks to distribute, i.e. a rather small number. If all the tasks are assigned to the CPU workers, then the execution is inefficient since the GPU is idle. Creating more parallelism comes at the cost of synchronization and since the size of the sparse vector is small, e.g. up to 250, the synchronization can severely decrease the performance. Moreover, a dynamic scheme poses additional NUMA related challenges. In order to achieve the best performance with respect to memory bandwidth, the input matrix must be local to each of the CPU workers. This can be accomplished by having multiple copies of the input matrix, one per NUMA node. Moreover, another complete copy must be stored on the GPU.

In contrast to dynamic load balancing, a static scheme uses the memory more efficiently from a memory consumption point of view. In this case, for each worker, CPU or GPU, it is known prior to execution the exact tile of the output vector assigned to it. Since the computation of a tile needs only the corresponding part of the input matrix, the matrix is sliced vertically which ensures that a worker thread and its associated slice are placed on the same NUMA node's memory. Using a combination of thread pinning and first touch policy, the input matrix can be divided across NUMA nodes and the GPU so that no extra copies of the matrix are necessary.

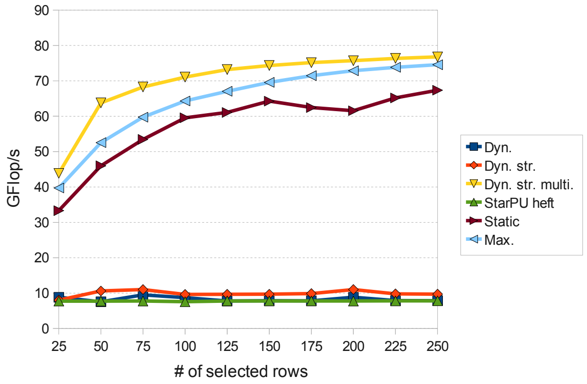


Figure 6.12: Load balancing of *spvm*.

Fig. 6.12 depicts the performance for different load balancing schemes. The small number of tasks causes *dyn*, *dyn str*, and *StarPU* to perform the worst. This is caused by the fact that the work is assigned to CPU workers and during the execution, the GPU is idle.

As seen in the graph, the *dyn str multi* scheme based on Listing 6.1 provides a better performance than the *sta* scheme. The multi-grain load balancing is in fact even faster than the expected *max* performance, computed by summing up the GFlop/s rate obtained for the CPU only execution and the GFlop/s rate for GPU only. This result is a direct consequence of the 2 CUDA streams used for overlapping the PCIe communication with computation on the GPU. Note that for determining *max*, the auto-tuned CUDA implementation of *spvm* does not include overlapping of GPU computation and PCIe communication.

6.7 Other Examples

The load balancing schemes described in this chapter can be applied to a wider range of applications, especially to those that contain data parallelism. Examples include: matrix multiplication, stencil computation, and direct n-body method. In order to achieve the best performance, the grain size problem has to be addressed. However, the employed scheme depends highly on the given problem and the input parameters. In a rather big problem, i.e. with numerous large tasks, the impact of the grain size is less significant.

For matrix multiplication, $A \times B = C$, both dynamic and static scheduling are suitable solutions on a heterogeneous system. If tasks are created by splitting the matrices both horizontally and vertically, then it becomes more difficult to implement Listing 6.1. In a multi-grain dynamic scheme in which workers grab tasks from a central queue, the tiling of the matrices and the creation of tasks is done during the actual execution. Two different types of tasks are generated, for the CPU and the GPU, operating with tiles of different sizes. The sizes must be chosen in such a way that both the CPU and the GPU are used efficiently during the execution of a task. On the CPU, the task size must allow for the best cache utilization, an important optimization in matrix multiplication. On the GPU, the goal is to fully occupy the GPU, SMs and multithreading, during the execution of every task. Static load balancing is more straightforward: A is partitioned horizontally according to Eq. 6.2, the entire B is copied to GPU memory, C is divided horizontally according to the splitting of A. The main advantage here is the elimination of the grain size problem. The disadvantages are more memory consumption due to the mirroring of B and no overlapping of communication and computation.

Stencil computation is in general memory bound. It usually involves multiple iterations during which the data is updated. This has a high implication on the load balancing scheme that should be used. Tasks are created by dividing the initial matrix into smaller submatrices structured in such a way that the boundaries are distinct vectors separated from the core. The idea is that only the boundaries have to be transferred over PCIe between iterations. If StarPU's HEFT scheduler is used, then the computation involves only the CPU since the PCIe transfer costs are too high in comparison to the execution time on the GPU and the scheduler does not look in the future to estimate the reuse of the data copied to the GPU and the transfer of the boundaries, not the core (except the last iteration). A solution to this is to manually copy the matrix to the GPU before the first iteration in the stencil computation so that the initial transfer does not affect later scheduling decisions. In fact, all the dynamic schemes previously described are impractical for addressing stencil computation. In this context, the best choice in this context is static load balancing in which the matrix is partitioned horizontally. The transfers over PCIe are in this case minimized, an important aspect for stencil computation.

The direct n-body method is computationally bound which means that all the load balancing schemes described in the chapter are applicable and should be efficient. It is assumed that the computation has more than one iterations, i.e. the bodies move multiple times. A task computes the interaction between two subsets of bodies. A reduction is necessary to sum up the contributions of the tasks that have a common subset of bodies. From this point, the discussion is similar to the one for *sginterp*. The primary grain size used in a StarPU implementation is given by the grain size preferred by the fastest processor, in general the GPU. A multi-grain dynamic load balancing approach also considers the grain size for the CPU which is set according to the cache optimization applied to the direct n-body method. For static load balancing, the bodies are divided into two subsets, one for the CPU and one for the GPU, according to the time measured on each of the two processors. The positions of the bodies must be consistent across PCIe. In order to achieve this, after each iteration of the n-body method (the new positions of all the bodies are updated), the positions of the bodies are exchanged over PCIe, the CPU sends its data to the GPU and vice versa.

6.8 Summary

The central point of this chapter is load balancing on heterogeneous systems composed of CPUs and GPUs. The main problem comes from the difficulty of reaching a high level of performance in the presence of heterogeneity using task based frameworks such as StarPU. First, the grain size or task size has a significant impact on performance. In order to choose the right grain size, one has to carefully study the interaction between the grain size and synchronization overheads, parallelism, and optimizations. This chapter describes an algorithm for load balancing that allows for the use of multiple grain sizes (Listing 6.1), one grain size that matches the CPU optimizations and one that is set according to the GPU optimizations. For data parallelism, a static load balancing scheme is proposed which eliminates the grain size problem.

Choosing between dynamic and static load balancing depends on the system and on the application. If the tasks are non-deterministic or the system is non-dedicated, then a dynamic approach based on tasks is a suitable fit as there are no guarantees about the load in the system, e.g. if the GPU becomes busy because of an external application, the load balancing scheme should adapt. If the system is dedicated and the tasks are deterministic, then dynamic load balancing can still be a better solution than the static one because of the overlapping between PCIe communication and GPU computation as shown for *spvm*. However, this rule is not general as it is invalidated in the case of *sginterp* for which the best performance, 98% of the maximum, is achieved by a static approach. The overlapping between communication and computation can also be achieved for the static approach although it introduces the grain size problem characteristic to task based load balancing.

The use of a *to-all* distribution of the work in which all the processors are engaged in the computation, is not always feasible. This is the case for *sghierarch* for which the low computational intensity and the communication requirements eliminate the possibility of using the CPU and the GPU simultaneously. Therefore, a *to-one* mapping is used in which depending on the input, the fastest processor is chosen for handling the entire work.

Chapter 7

Conclusion and Future Work

7.1 Sparse Grids on Heterogeneous Systems

This thesis tackles the problem of how to best exploit modern heterogeneous system such as those containing multi-core CPUs and GPUs. The sparse grid technique is a numerical technique that serves as a test application for exploring different optimization methods.

The sparse grid algorithms are the core of a computational steering application in which high-dimensional simulation data is stored in a database using a form of lossy compression and is afterwards decompressed for visualization. There are three main components of high importance for this application: sparse grid hierarchization (*sghierarch*), sparse grid interpolation (*sginterp*), and sparse vector - matrix multiplication (*spvm*). These components together with different variations created for more flexibility and performance, are incorporated in a numerical library, *fastsg*, and a benchmark, *sgbench*, which are proposed in this thesis.

One of the focus points of this thesis is how to best address the challenges of GPUs. GPUs are multithreaded many-core processors (16 cores) with wide SIMD units, e.g. up to 32 single-precision lanes. They have a limited amount of memory, e.g. less than 6 GB. Consequently, from the performance point of view, GPUs are nearly incompatible with control dependent codes and complex (key-value based) data structures such as hash-tables and trees which generally consume a lot of memory and often do not allow for an efficient use of SIMD units by violating the requirements for compact and aligned memory accesses.

7.2 Optimization Strategy

In their initial form, *sghierarch* and *sginterp* make use of hash-tables or trees, and are recursive. Therefore, in order to efficiently port *sghierarch* and *sginterp* to GPUs, a redesign of the data structure and algorithms is necessary. As a response to this requirement, a *bijective map is proposed* which minimizes the memory footprint, thus allowing for larger problems to be handled by GPUs. Moreover, a contribution of this thesis is a *set of non-recursive sparse grid algorithms* that are more efficient and are compatible with GPUs which do not support recursion efficiently.

The thesis describes a search based auto-tuning method and *optimization parameters common for GPU codes.* It also presents the interactions between them which are exploited by *search methods based on partitions*, a concept proposed in this thesis. The optimization parameters control *the size of a thread block*, *the amount of work per thread*, and *the parallelism granularity.* Emphasis is placed on decreasing the auto-tuning time by employing an *input reduction technique* in which an initial computational work is reduced without sacrificing too much the performance associated to the found optimization parameters. Within this technique, special attention is given to tail effects which are generated by insufficient parallelism towards the end of the computation on the GPU. Another important aspect of input reduction consists in a model of the execution on the GPU for homogeneous threads. As part of this model, GPU threads are grouped in waves which fully occupy a GPU. All the threads in a wave have a synchronized start and stop time. Therefore, the execution time of a GPU program can be approximated based on the number of waves and the duration of a wave.

A common question on heterogeneous systems is whether the CPU and the GPU can be simultaneously used for handling a given computational work. Different static and dynamic load balancing schemes are described in the context of data parallelism which is characteristic to the three sparse grid kernels. Dynamic task based load balancing which is included in heterogeneous programming frameworks such as *StarPU*, is common these days as it provides several important advantages: automatic transfer of data over PCIe, overlapping of data transfers with execution on GPU, and a software controlled memory consistency protocol which automatically manages data allocation on the GPU, data copy to / from the GPU, and data eviction from the GPU, thus allowing to overcome to some extent the limited amount of memory on the GPU.

An important problem of task based load balancing is how to choose the best task size (or grain size). A too small of a grain size may generate too much overhead and may not allow for the most efficient utilization of the CPU and the GPU. A too large of a grain size may not allow for the optimal distribution of the work among the processors. Moreover, the CPU and the GPU may have different preferred grain sizes that match their specific optimizations. In order to cope with this problem, a *multi-grain dynamic load balancing algorithm* is proposed which aims at keeping the fastest processor, e.g. the GPU, busy at any moment of time. Moreover, the algorithm allows for the overlapping of PCIe communication with computation on the GPU. Another load balancing option is a *static scheme* in which the work is divided into two chunks, for the CPU and for the GPU. The sizes of the chunks are calculated based on approximations of the execution time for the CPU and the GPU.

7.3 Summary of Performance Results

Fig. 7.1a, Fig. 7.1b, and Fig. 7.1c provide a concentrated view on the performance numbers presented in Chapter 4, Chapter 5, and Chapter 6. Emphasis is placed here on the impact of the optimizations described in this thesis for *sghierarch*, *sginterp*, and *spvm*. For CPUs, one can see that speedups of 86.4x, 106.3x, and 11.1x can be achieved. Multithreading is nowadays a characteristic of both CPUs and GPUs. The graphs show that only for *sginterp*, Hyperthreading is actually beneficial. For all the computational kernels, the GPU is faster than 8 CPU cores, providing speedups of 1.89x (*sghierarch*), 7.57x (*sginterp*), and 2.61x (*spvm*) compared to the corresponding vectorized and parallelized versions for the CPU. The GPU speedups relative to one CPU core correlate with other highly optimized GPU codes for matrix multiplication, n-body simulation, sparse matrix - vector multiplication, e.g. roughly one order of magnitude speedup for memory bound programs (*spvm*) assuming a 10x higher memory bandwidth of GPUs compared to CPUs, and up to two orders of magnitude for computationally bound programs (*sginterp*) assuming a 100x higher theoretical performance of GPUs.

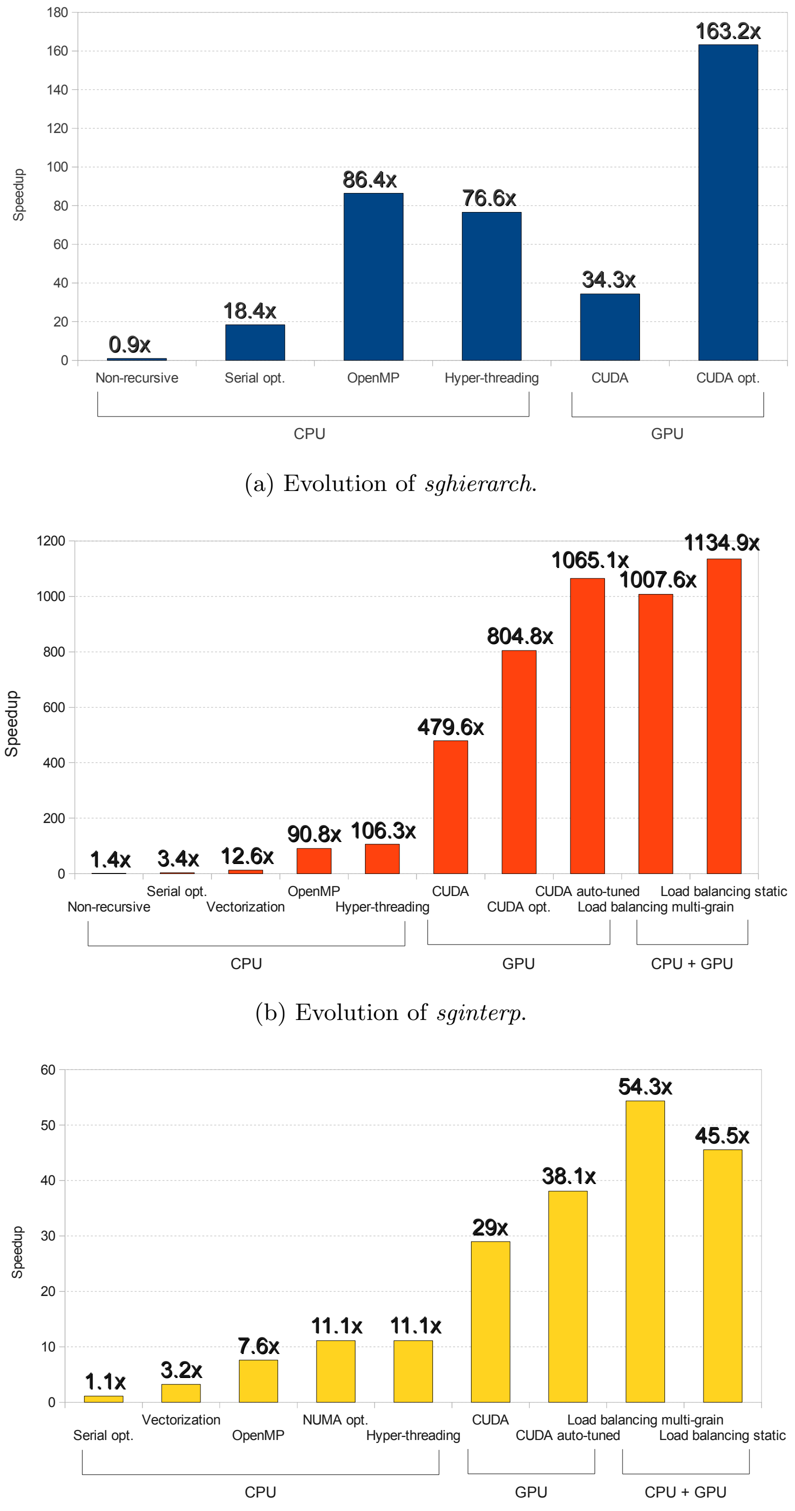
By using auto-tuning for the GPU implementations of *sginterp* and *spvm*, the performance is further improved as depicted in the graphs, i.e. the auto-tuned versions are 1.3x faster than the optimized versions without auto-tuning. The results presented in Chapter 5 show that input reduction can decrease the auto-tuning time by up to a factor of 10^3 for *sginterp* and up to 10 for *spvm* while the performance is on average 98% and 96% of the performance corresponding to the optimization parameters found without input reduction.

The best load balancing scheme for *sginterp* is the static one while for *spvm* the best performance is returned by the dynamic multi-grain approach. In the case of *sghierarch*, the necessity to transfer the data back and forth over PCIe does not allow for the simultaneous use of the CPU and the GPU. Therefore, a *to-one* mapping is used in which only the best processor is chosen for execution based on its performance measured for the given input parameters.

7.4 Future Work

7.4.1 Optimization Techniques

The empirical optimization method proposed in this thesis, i.e. GPU optimization parameters, partition based search method, input reduction technique, is expected to be applicable to a



(a) Evolution of *sghierarch*.

(b) Evolution of *sginterp*.

(c) Evolution of *spvm*.

Figure 7.1: Impact of optimizations on the performance of the sparse grid kernels. The speedup numbers are computed relative to the initial serial versions of the kernels. Tested system: dual-socket, 4 cores / socket, multithreaded Intel Nehalem (CPU), Nvidia Quadro 6000 (GPU).

wider set of applications. Incorporating the auto-tuning method in a framework would help to apply it more easily to other programs and to test its validity in a wider scope.

The proposed load balancing schemes are also applicable to other data parallel applications. By extracting the load balancing component from *fastsg*, a lightweight library could be created for testing to which extent the dynamic and static scheduling schemes described in this thesis can help other programs to improve their utilization of a heterogeneous system.

7.4.2 Enhanced Sparse Grid Functionality

Regarding the sparse grid technique, an important question is whether the data structures and algorithms proposed in this thesis can help to improve the performance of fully adaptive sparse grids characterized by more challenging sparsity patterns. Addressing those patterns using the data structure and algorithms from this thesis would imply storing zeros and performing unnecessary computations. The challenge is to determine in which conditions such an approach is more efficient in terms of space and time than using hash-tables and recursive functions which are common components of fully adaptive sparse grid implementations.

The future work includes implementing MPI versions of the sparse grid routines in the *fastsg* library. This would allow for the execution on a cluster of GPU accelerated heterogeneous systems. The objective would be achieving scalability while preserving the easy-to-use interface of *fastsg*. Nevertheless, algorithms characterized by complex data access patterns such as sparse grid hierarchization pose serious challenges for the porting of the library to clusters.

Bibliography

[1] M. Garland and D. B. Kirk. Understanding Throughput-oriented Architectures. *Commun. ACM*, 53:58–66, November 2010.

[2] H. J. Bungartz and M. Griebel. Sparse Grids. *Acta Numerica*, 13:147–269, 2004.

[3] Robert H. Dennard, Fritz H. Gaensslen, Hwa-Nien Yu, V. Leo Rideovt, Ernest Bassous, and Andre R. Leblanc. Design of ion-implanted MOSFET's with very small physical dimensions. *Solid-State Circuits Newsletter, IEEE*, 12(1):38 –50, winter 2007.

[4] S. Borkar and A.A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[5] Herb Sutter. The free lunch is over: A fundamental turn toward concurrency in software. *Dr. Dobbs Journal*, 30(3):202–210, 2005.

[6] Thomas Chen, Ram Raghavan, Jason N Dale, and Eiji Iwata. Cell broadband engine architecture and its first implementationa performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[7] John Owens. GPU architecture overview. In *ACM SIGGRAPH*, volume 1, pages 5–9, 2007.

[8] Intel. *Intel Many Integrated Core (Intel MIC) Architecture*, 2011. ISC'11 Demos and Performance Description.

[9] Ian Kuon, Russell Tessier, and Jonathan Rose. Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253, 2008.

[10] Andre R Brodtkorb, Christopher Dyken, Trond R Hagen, Jon M Hjelmervik, and Olaf O Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[11] NVIDIA. NVIDIA's Next Generation CUDA Compute Architecture: Fermi. White paper, 2009.

[12] A. Grama. *Introduction to parallel computing.* Addison Wesley, 2003.

[13] *OpenMP Application Program Interface Version 3.0*, 2008.

[14] NVIDIA. *CUDA Programming Guide 4.0*, 2011.

[15] OpenCL 1.1 Specification, 2011. The Khronos Group.

[16] The Linpack Benchmark — TOP500 Supercomputer Sites. `http://www.top500.org/project/linpack`.

[17] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *SC*, page 31, 2008.

[18] Lars Nyland and Mark Harris. Chapter 31 Fast N-Body Simulation with CUDA. 2007.

[19] R.G. Belleman, J. Bédorf, and S.F. Portegies Zwart. High performance direct gravitational N-body simulations on graphics processing units: An implementation in CUDA. *New Astronomy*, 13(2):103–112, 2008.

[20] Akira Nukada and Satoshi Matsuoka. Auto-tuning 3-D FFT library for CUDA GPUs. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 30:1–30:10, New York, NY, USA, 2009. ACM.

[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] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 4:1–4:12, Piscataway, NJ, USA, 2008. IEEE Press.

[23] P. Micikevicius. 3D finite difference computation on GPUs using CUDA. In *Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*, pages 79–84. ACM, 2009.

[24] N. Bell and M. Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[25] Jee W. Choi, Amik Singh, and Richard W. Vuduc. Model-driven autotuning of sparse matrix-vector multiply on GPUs. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, pages 115–126, New York, NY, USA, 2010. ACM.

[26] J. Nickolls, I. Buck, M. Garland, and K. Skadron. Scalable Parallel Programming with CUDA. *ACM Queue*, 6(2):40–53, 2008.

[27] H. J. Bungartz and M. Griebel. Sparse Grids. *Acta Numerica*, 13:147–269, 2004.

[28] D. Butnaru, D. Pflüger, and H. J. Bungartz. Towards High-Dimensional Computational Steering of Precomputed Simulation Data using Sparse Grids. *Procedia CS*, 4:56–65, 2011.

[29] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems.* Verlag Dr. Hut, München, August 2010.

[30] M. Salaterski. Storing Results of Parameterized CFD Simulations Using Sparse-Grid Techniques, May 2008.

[31] Daniel Butnaru, Benjamin Peherstorfer, Hans-Joachim Bungartz, and Dirk Pflüger. Fast Insight into High-Dimensional Parametrized Simulation Data. In *ICMLA (2)*, pages 265–270, 2012.

[32] J. Garcke. A dimension adaptive sparse grid combination technique for machine learning. In *Proceedings of the 13th Biennial Computational Techniques and Applications Conference, CTAC-2006*, volume 48, pages C725–C740, 2007.

[33] T. Gerstner and M. Griebel. Dimension-Adaptive Tensor-Product Quadrature. *Computing*, 71:2003, 2003.

[34] M. Hegland. Adaptive sparse grids. In *CTAC-2001*, volume 44, pages C335–C353, April 2003.

[35] A. Klimke and B. Wohlmuth. Algorithm 847: spinterp: Piecewise Multilinear Hierarchical Sparse Grid Interpolation in MATLAB. *ACM Transactions on Mathematical Software*, 31(4), 2005.

[36] A. Klimke. Sparse Grid Interpolation Toolbox – User's Guide. Technical Report IANS report 2007/017, University of Stuttgart, 2007.

[37] Hewlett-Packard Company. *Standard Template Library Programmer's Guide*, 1994.

[38] M. May and T. Schiekofer. An Abstract Data Type for Parallel Simulations based on Sparse Grids. In A. Bode, J. Dongarra, T. Ludwig, and V. Sunderam, editors, *Proceedings of the Third European PVM Conference*, volume 1156 of *Lecture Notes in Computer Science*, pages 59–67. Springer Verlag, 1997.

[39] M. Griebel. Adaptive sparse grid multilevel methods for elliptic PDEs based on finite differences. *Computing*, 61(2):151–179, 1998.

[40] D. Pflüger. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Dissertation, Institut für Informatik, Technische Universität München, February 2010.

[41] C. Feuersänger. Dünngitterverfahren für hochdimensionale elliptische partielle Differentialgleichungen, 2005.

[42] Chr. Feuersänger. *Sparse Grid Methods for Higher Dimensional Approximation*. Dissertation, Universität Bonn, 2010.

[43] A. F. Murarasu, J. Weidendorfer, G. Buse, D. Butnaru, and D. Pflüger. Compact Data Structure and Scalable Algorithms for the Sparse Grid Technique. In *PPoPP*, pages 25–34, 2011.

[44] A. Murarasu, G. Buse, D. Pflüger, J. Weidendorfer, and A. Bode. fastsg: A Fast Routines Library for Sparse Grids. In *ICCS*, 2012.

[45] R. Vuduc, J. W. Demmel, and K. A. Yelick. OSKI: A Library of Automatically Tuned Sparse Matrix Kernels. In *Institute of Physics Publishing*, 2005.

[46] M. Frigo and S. G. Johnson. *FFTW: An Adaptive Software Architecture for the FFT*, volume 3, pages 1381–1384. IEEE, 1998.

[47] X. Li, M. J. Garzaran, and D. Padua. Optimizing Sorting with Genetic Algorithms. In *CGO '05*, pages 99–110, Washington, DC, USA, 2005. IEEE Computer Society.

[48] M. Olszewski and M. Voss. Install-Time System for Automatic Generation of Optimized Parallel Sorting Algorithms. In *PDPTA*, pages 17–23, 2004.

[49] H. Yu and L. Rauchwerger. Adaptive Reduction Parallelization Techniques. In *ICS*, pages 66–75. ACM Press, 2000.

[50] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. A Framework for Adaptive Algorithm Selection in STAPL. In *PPOPP*, pages 277–288, 2005.

[51] E. A. Brewer. High-level Optimization via Automated Statistical Modeling. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 80–91, New York, NY, USA, 1995. ACM.

[52] A. F. Murarasu and J. Weidendorfer. Building Input Adaptive Parallel Applications: A Case Study of Sparse Grid Interpolation. In *Proceedings of the 15th IEEE International Conference on Computational Science and Engineering (ICCSE)*, 2012. accepted for publication.

[53] H. Wong, M. M. Papadopoulou, M. Sadooghi-Alvandi, and A. Moshovos. Demystifying GPU microarchitecture through microbenchmarking. In *ISPASS*, pages 235–246, 2010.

[54] Vasily Volkov. Better Performance at Lower Occupancy. Nvidia GTC 2010.

[55] Ahmad Abdelfattah, David Keyes, Jack Dongarra, and Hatem Ltaief. Optimizing Memory-Bound Numerical Kernels on GPU Hardware Accelerators. In *VECPAR*, 2012.

[56] M. Griebel. A parallelizable and vectorizable multi-level algorithm on sparse grids. In *Parallel algorithms for partial differential equations*, Notes Numer. Fluid Mech. 31, pages 94–100. Vieweg, Wiesbaden, 1991.

[57] M. Griebel, M. Schneider, and C. Zenger. A combination technique for the solution of sparse grid problems. In *Iterative Methods in Linear Algebra*, pages 263–281, 1992.

[58] A. Gaikwad and I. M. Toke. GPU based sparse grid technique for solving multidimensional options pricing PDEs. In *WHPCF '09: Proceedings of the 2nd Workshop on High Performance Computational Finance*, pages 1–9, New York, NY, USA, 2009. ACM.

[59] A. Deftu and A. Murarasu. Optimization Techniques for Dimensionally Truncated Sparse Grids on Heterogeneous Systems. In *Euromicro PDP*, 2013.

[60] A. Heinecke and D. Pflüger. Multi- and Many-Core Data Mining with Adaptive Sparse Grids. In *CF*, May 2011.

[61] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26:345–420, December 1994.

[62] A Guide to Auto-vectorization for Intel C++ Compilers, July 2011. Intel.

[63] Intel C++ Intrinsics Reference, 2007. Intel.

[64] Intel Advanced Vector Extensions Programming Reference, June 2011. Intel.

[65] R. Clinton Whaley, Antoine Petitet, and Jack Dongarra. Automated empirical optimizations of software and the ATLAS project. *Parallel Computing*, 27(1-2):3–35, 2001.

[66] Richard Vuduc, James W. Demmel, and Katherine A. Yelick. OSKI: A library of automatically tuned sparse matrix kernels. In *Proc. SciDAC, J. Physics: Conf. Ser.*, volume 16, pages 521–530, 2005.

[67] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. In *Proc. IEEE, Special Issue on Program Generation, Optimization, and Adaptation*, 2005.

[68] Markus Püschel, José M. F. Moura, Bryan Singer, Jianxin Xiong, Jeremy R. Johnson, David A. Padua, Manuela M. Veloso, and Robert W. Johnson. Spiral: A Generator for Platform-Adapted Libraries of Signal Processing Alogorithms. *IJHPCA*, 18(1):21–45, 2004.

[69] Sébastien Donadio, James C. Brodman, Thomas Roeder, Kamen Yotov, Denis Barthou, Albert Cohen, María Jesús Garzarán, David A. Padua, and Keshav Pingali. A Language for the Compact Representation of Multiple Program Versions. In *LCPC*, pages 136–151, 2005.

[70] Qing Yi, Keith Seymour, Haihang You, Richard W. Vuduc, and Daniel J. Quinlan. POET: Parameterized Optimizations for Empirical Tuning. In *IPDPS*, pages 1–8, 2007.

[71] Albert Hartono, Boyana Norris, and Ponnuswamy Sadayappan. Annotation-based empirical performance tuning using Orio. In *IPDPS*, pages 1–11, 2009.

[72] K. Seymour, H. You, and J. J. Dongarra. A Comparison of Search Heuristics for Empirical Code Optimization. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*, Third international Workshop on Automatic Performance Tuning (iWAPT 2008), pages 421–429, Tsukuba International Congress Center, EPOCHAL TSUKUBA, Japan, September 2008. IEEE.

[73] Yinan Li, Jack Dongarra, and Stanimire Tomov. A Note on Auto-tuning GEMM for GPUs. In *Proceedings of the 9th International Conference on Computational Science: Part I*, ICCS '09, pages 884–892, Berlin, Heidelberg, 2009. Springer-Verlag.

[74] Christoph A. Schaefer, Victor Pankratius, and Walter F. Tichy. Atune-IL: An Instrumentation Language for Auto-tuning Parallel Applications. In *Euro-Par*, pages 9–20, 2009.

[75] ISAT. http://software.intel.com/en-us/articles/intel-software-autotuning-tool/.

[76] Ananta Tiwari, Chun Chen, Jacqueline Chame, Mary W. Hall, and Jeffrey K. Hollingsworth. A scalable auto-tuning framework for compiler optimization. In *IPDPS*, pages 1–12, 2009.

[77] Chun Chen, Jacqueline Chame, and Mary Hall. CHiLL: A framework for composing high-level loop transformations. Technical report, 2008.

[78] ROSE. http://rosecompiler.org/.

[79] Chunhua Liao and Dan Quinlan. *A ROSE-Based End-to-End Empirical Tuning System for Whole Applications*, 2012. Tutorial.

[80] Chun Chen, Jacqueline Chame, Mary W. Hall, and Kristina Lerman. A Systematic Approach to Model-Guided Empirical Search for Memory Hierarchy Optimization. In *LCPC*, pages 433–440, 2005.

[81] Gabe Rudy, Malik Murtaza Khan, Mary W. Hall, Chun Chen, and Jacqueline Chame. A Programming Language Interface to Describe Transformations and Code Generation. In *LCPC*, pages 136–150, 2010.

[82] Yixun Liu, Eddy Z. Zhang, and Xipeng Shen. A cross-input adaptive framework for GPU program optimizations. In *IPDPS*, pages 1–10, 2009.

[83] Mehrzad Samadi, Amir Hormati, Mojtaba Mehrara, Janghaeng Lee, and Scott Mahlke. Adaptive input-aware compilation for graphics engines. In *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, pages 13–22, New York, NY, USA, 2012. ACM.

[84] William Thies, Michal Karczmarek, and Saman Amarasinghe. StreamIt: A language for streaming applications. In *Compiler Construction*, pages 49–84. Springer, 2002.

[85] Paulius Micikevicius. GPU Performance Analysis and Optimization. Nvidia GTC 2012.

[86] Using clock to measure performance of CUDA kernels. http://developer.download.nvidia.com/compute/DevZone/C/Projects/x64/clock.zip.

[87] MAGMA, Matrix Algebra on GPU and Multicore Architectures. `http://icl.cs.utk.edu/magma/index.html`.

[88] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. In *Euro-Par*, pages 863–874, 2009.

[89] G.F. Diamos and S. Yalamanchili. Harmony: an execution model and runtime for heterogeneous many core systems. In *Proceedings of the 17th international symposium on High performance distributed computing*, pages 197–200. ACM, 2008.

[90] M.D. Linderman, J.D. Collins, H. Wang, and T.H. Meng. Merge: a programming model for heterogeneous multi-core systems. In *ACM SIGOPS Operating Systems Review*, volume 42, pages 287–296. ACM, 2008.

[91] E. Ayguadé, R. Badia, F. Igual, J. Labarta, R. Mayo, and E. Quintana-Ortí. An extension of the StarSs programming model for platforms with multiple GPUs. *Euro-Par 2009 Parallel Processing*, pages 851–862, 2009.

[92] G. Teodoro, R. Sachetto, O. Sertel, M.N. Gurcan, W. Meira, U. Catalyurek, and R. Ferreira. Coordinating the use of GPU and CPU for improving performance of compute intensive applications. In *Cluster Computing and Workshops, 2009. CLUSTER'09. IEEE International Conference on*, pages 1–10. IEEE, 2009.

[93] C.K. Luk, S. Hong, and H. Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009.

[94] C. Augonnet, S. Thibault, and R. Namyst. Automatic calibration of performance models on heterogeneous multicore architectures. In *Euro-Par 2009–Parallel Processing Workshops*, pages 56–65. Springer, 2010.

[95] H. Topcuoglu, S. Hariri, and M. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3):260–274, 2002.

[96] H. El-Rewini and M. Abd-El-Barr. *Advanced computer architecture and parallel processing*, volume 30. Wiley-interscience, 2005.

[97] A. Osman and H. Ammar. Dynamic Load Balancing Strategies for Parallel Computers. In *ISPDC*, Romania, July 2002.

[98] M. Frigo, C.E. Leiserson, and K.H. Randall. The implementation of the Cilk-5 multithreaded language. *ACM Sigplan Notices*, 33(5):212–223, 1998.

[99] C. Augonnet, S. Thibault, R. Namyst, et al. Starpu: a runtime system for scheduling tasks over accelerator-based multicore machines. 2010.

[100] Alin Florindor Murarasu, Josef Weidendorfer, and Arndt Bode. Workload Balancing on Heterogeneous Systems: A Case Study of Sparse Grid Interpolation. In *Euro-Par Workshops (2)*, pages 345–354, 2011.