A GPU-based Multi-level Subspace Decomposition Scheme for Hierarchical Tensor Product Bases

Author: Ivan Chernov
1st examiner: Univ.-Prof. Dr. Hans-Joachim Bungartz
2nd examiner: Univ.-Prof. Dr. Michael Bader
Assistant advisor(s): Dipl.-Inf. Gerrit Buse
Thesis handed in on: December 5, 2013
I hereby declare that this thesis is entirely the result of my own work except where otherwise indicated. I have only used the resources given in the list of references.

December 5, 2013

Ivan Chernov
I would never have been able to finish my thesis without the guidance of my scientific advisor, help from friends and support from my family.

I would like to express my deepest gratitude to my advisor, Dipl.-Inf. Gerrit Buse, for his excellent guidance, patience, and providing me perfect background for my research, reasonable critics and valuable comments and suggestions to move forward.

Assistance provided by my TUM mentor Hans Pongratz was greatly appreciated.

I am particularly grateful for the assistance given by Christian Fuchs, who made an invaluable contribution to the process of CUDA setup.

I would also like to thank Martin Oswald and Frank Steinbruecker, for the CUDA course, which allowed me to implement my ideas in effective way.

And my special thanks are extended to my family members — they were always supporting me and encouraging me with their best wishes.
Abstract

The aim of this thesis is to implement a multi-level splitting of full grids on the GPU, which could be used in the incremental visualization of scientific data sets. The splitting is motivated by the approximation properties of the sparse grid technique. Looking towards large amounts of data, ideas of parallelization and data slicing are discussed and implemented. State-of-the-art implementations of the splitting algorithm are discussed and the highly parallelizable part is extracted. We compare against a highly tuned CPU version of the algorithm, and we aim to speed up the calculation as much as possible. We suppose that a higher degree of parallelism can lower the time to solution, so highly parallel GPUs sound promising as target platforms. We take general implementation ideas from the CPU version, transfer and map them to the GPU.

Although the performance results of this first implementation are promising, the parallel and vectorized CPU version is still a bit faster. Still, in terms of performance the GPU implementation comes close to the highly-tuned CPU version of the algorithm. Following the approach further might prove useful for the time-critical task of visualizing (reading, processing, drawing) of large amounts of data, which needs to be as fast as possible and requires access to both coarse and fine level representation of the data.
## Contents

**Acknowledgements**  
v

**Abstract**  
vii

1 **Introduction**  
1

2 **Multi-level Subspace Splitting**  
2.1 Piecewise linear interpolation on a hierarchical full grid  
2  
2.2 Piecewise linear interpolation on a sparse grid  
7  
2.3 Hierarchical decomposition and reconstruction  
9  
\hspace{2em} 2.3.1 Hierarchization in the 1-dimensional case  
9  
\hspace{2em} 2.3.2 Hierarchization in multiple dimensions  
12

3 **CUDA**  
17  
\hspace{2em} 3.1 GPU overview  
17  
\hspace{2em} 3.2 The CUDA platform  
18  
\hspace{3em} 3.2.1 GPU Hardware and programming model  
18  
\hspace{3em} 3.2.2 Execution model  
19  
\hspace{3em} 3.2.3 Managing memory  
20  
\hspace{2em} 3.3 The Fermi architecture  
22

4 **Problem Analysis and Solution Approaches**  
25  
\hspace{2em} 4.1 Potential for parallelization in the hierarchical transform  
25  
\hspace{2em} 4.2 Hardware setup  
25  
\hspace{3em} 4.3 Parallelization approaches and hardware constraints  
27

5 **Performance Results and Observations**  
31  
\hspace{2em} 5.1 Benchmarks  
31  
\hspace{2em} 5.2 Comparison with the CPU  
34

6 **Conclusions**  
37

**Bibliography**  
39
1 Introduction

Nowadays, algorithms for data processing and visualization have to handle increasing amounts of information. And, in many cases, information is not only growing in volume, but also in dimensionality. That is the point where most algorithms meet the so-called curse of dimensionality, a term coined by Bellman, 1961. It is the famous obstacle to the use of numerical treatment of high-dimensional problems. The phenomenon can be formulated as follows: the amount of data needed to maintain a certain level of accuracy grows exponentially with the dimensionality. That leads to an increase in analysis and processing effort as well as exponential growth of computation time of algorithms not specifically optimized for high dimensions.

One effective approach to overcome the curse of dimensionality is given by the sparse grids technique. Often, data is kept in form of a full grid. If in a given setting only a reasonable level of accuracy suffices, there is no need to keep and process the whole volume of data. In such cases, the full data grid can be transformed into a hierarchy of smaller grids. Being combined, even only the coarsest grids can help to reconstruct a reasonable representation of the original data set. It is proven in sparse grid theory that combination of grids reconstruct original data is optimal in terms of accuracy.

The process of conversion of the original full grid into set of hierarchical levels is called hierarchization, appears often in sparse grid settings. It is an additional step required before data processing and most of the time-restricted algorithms need the fastest possible implementation of this transformation.

We already have a highly tuned CPU version of hierarchization algorithm, which uses vectorization and parallelization. But big amounts of data to process and high potential of parallelization leads naturally to the idea of using GPU as highly parallel platform. In this work we implement hierarchization algorithm on the GPU and compare the results of this implementation with given highly-optimized reference CPU implementation.

In Chapter 2, we overview the theory of full and spatial grids. We observe full and spatial grids interpolation and show how a function can be represented as the sum of weighted hierarchical basis functions. Then we explain the hierarchical decomposition.
In Chapter 3, we describe the general GPU architecture and how it differs from the CPU. We give an overview of Nvidia’s CUDA platform for parallel computations and revise Nvidia Fermi architecture, which is later used for implementing and testing.

In Chapter 4, we propose several ways of effective parallelization of the hierarchization algorithm, considering possibilities and limitations of the Nvidia Fermi architecture and the CUDA platform.

Finally, in Chapter 5, we compare our solution with the reference CPU implementation. We compare different performance parameters and analyze differences. In the Conclusions, we make some assumptions how GPU performance could be even increased.
2 Multi-level Subspace Splitting

At first, we briefly give an overview of the linear interpolation on a full grid. After that we review sparse grid interpolation, more useful and less expensive in terms of computation effort when dealing with high dimensional datasets. In Section 2.3, we introduce the hierarchical transform. In the same Section 2.3 we show that the hierarchical transform is effective for full grid reconstruction, transforming the function values to hierarchical surpluses and back.

The underlying idea of sparse grids was developed by the Russian mathematician S. Smolyak [14], who used it for numerical integration.

With the given 1D multilevel basis (\(H^1\)- and \(L^2\)-stable is taken into account), consisting of piecewise linear basis functions a class of functions with bounded second derivatives is considered. Each 1D function from this class can be expressed as a linear combination of basis functions, and it is proven in [1] that in such combinations corresponding coefficients decrease quickly with increasing level. Grid points, remaining after omission of non-significant ones, will form a sparse grid. It turns out that our calculations, needed for approximation of the original function using the sparse grids approach, are no more dependent on \(d\) exponentially — only \(O(N \cdot (\log N)^{d-1})\) degrees of freedom are involved, and approximation of given accuracy is reached with significantly less effort.

As was proven in [1], the cost complexity of the sparse grid approach has the order of only \(O(N \cdot (\log N)^{d-1})\) (significant reduction from \(O(N^d)\) in the full grid case). The accuracy, obtained with piecewise linear basis functions, with respect to the \(L_2\) norm, is \(O(N^{-2} \cdot (\log N)^{d-1})\) (just slightly deteriorates from \(O(N^{-2})\) in case of a full grid) — here \(N\) is the number of grid points in one coordinate direction at the boundary.

2.1 Piecewise linear interpolation on a hierarchical full grid

At first, we discuss the full grid interpolation approach. Consider a one-dimensional multi-level basis. We will work with the hierarchical basis based on so-called hat functions that are defined as follows:
\[ \phi(x) := \begin{cases} 1 - |x| & \text{if } x \in [-1, 1] \\ 0 & \text{otherwise} \end{cases} \]

Then a set of equidistant grids — one per level \( l \) — is considered on the unit interval \([-1, 1]\), with mesh size \( h_l = 2^{-l} \). Grid points \( x_{l,i} \):

\[ x_{l,i} := i \cdot h_l, \quad 0 \leq i \leq 2^l \]

The basis of hat functions \( \phi_{l,i} \), also called *nodal basis*, is given by

\[ \phi_{l,i} := \phi \left( \frac{x - i \cdot h_l}{h_l} \right) \]

Here, \( i \) is a locator of a hat function in a grid. From here we define function spaces \( V_n \) and so-called *hierarchical increments* \( W_l \):

\[ V_n := \text{span}\{ \phi_{n,i} : 1 \leq i \leq 2^n - 1 \} \]

\[ W_l := \text{span}\{ \phi_{l,i} : i \in I_l \}, \]

where \( I_l \) is the index set

\[ I_l := \{ i \in \mathbb{N} : 1 \leq i \leq 2^l - 1, \; i \text{ odd.} \} \]

The relation between \( V_n \) and \( W_l \) is defined as follows:

\[ V_n = \bigoplus_{l \leq n} W_l \]

The set of hat functions, corresponding to \( W_l \), is called *hierarchical basis*. Any function \( f \in V_n \) can be represented as the sum of \( \phi_{l,i} \) functions:

\[ f(x) = \sum_{1 < i < n} \sum_{i \in I_l} c_{l,i} \cdot \phi_{l,i}(x) \quad (\star) \]

Here, the \( c_{l,i} \) are called *hierarchical coefficients* or *hierarchical surpluses*. Evaluation equation \((\star)\) at point \( x \) corresponds to the classical interpolation on a full grid.

The 1D case can be extended to the \( d \)-dimensional case. For that we introduce \( d \)-dimensional indices \( \ell : \ell = (l_1, l_2, \ldots, l_d) \in \mathbb{N}^d \). The \( d \)-dimensional mesh size is set as follows:

\[ h_\ell := (h_{l_1}, h_{l_2}, \ldots, h_{l_d}) := (2^{-l_1}, 2^{-l_2}, \ldots, 2^{-l_d}) \]

The \( d \)-dimensional grid points are defined as follows:

\[ x_\ell := (x_{l_1,i_1}, \ldots, x_{l_d,i_d}) \]
2.1 Piecewise linear interpolation on a hierarchical full grid

![Diagram of piecewise linear interpolation](image)

Figure 2.1: One-dimensional hat functions $\phi_{l,i}$ and corresponding grid points $x_{l,i}$ up to level $n = 3$ are shown in the nodal basis.

![Diagram of piecewise linear interpolation](image)

Figure 2.2: One-dimensional hat functions $\phi_{l,i}$ and corresponding grid points $x_{l,i}$ up to level $n = 3$ are shown in the hierarchical basis.
2 Multi-level Subspace Splitting

Figure 2.3: This Figure illustrates Two-dimensional "pagoda" functions $\phi_{l,i}(x)$, as they appear in the hierarchical basis for $||l||_{\infty} \leq 3$. 
2.2 Piecewise linear interpolation on a sparse grid

where \( x_{lp, ip} := i_p \cdot h_{lp} = i_p \cdot 2^{-lp}, \) \( i_p = 0, \ldots, 2^{lp} \). The \( d \)-dimensional hat functions \( \phi_{lp}(x) \) are defined as product of 1D hat basis functions:

\[
\phi_{lp}(x) := \prod_{s=1}^{d} \phi_{ls, is}(x_s)
\]

The \( d \)-dimensional index sets \( I_l \) are defined in a similar way as the 1D ones:

\[
I_l := \{ i : 1 \leq i_p \leq 2^{lp} - 1, \ i_p \text{ odd}, \ 1 \leq p \leq d \}\n\]

The \( d \)-dimensional hierarchical increments \( W_l \) are defined as follows:

\[
W_l := \text{span}\{ \phi_{lp} : i \in I_l \}
\]

The \( d \)-dimensional \( V_n \) are defined as space of \( d \)-dimensional piecewise \( d \)-linear functions:

\[
V_n := \text{span}\{ \phi_{nl,i} | s = 1, \ldots, 2^{ls} - 1, s = 1, \ldots, d \} = \bigoplus_{|l| \leq n} W_l
\]

From here \( |l|_\infty \) and \( |l|_1 \) are defined as follows:

\[
|l|_\infty = \max(|l_1|, |l_2|, \ldots, |l_d|)
\]

and

\[
|l|_1 = \sum_{1 \leq i \leq d} |l_i|
\]

As in the 1D case, any function \( f \in V_n \) has a unique representation as the sum of corresponding \( \phi_{l,i} \) functions:

\[
f(x) = \sum_{|l| \leq n} \sum_{i \in I_l} c_{l,i} \cdot \phi_{l,i}(x)
\]

2.2 Piecewise linear interpolation on a sparse grid

Sparse grid spaces \( V_n \) are also defined as direct sums of hierarchical increments \( W_l \), but with another summation condition:

\[
V_n^{(1)} := \bigoplus_{|l| \leq n + d - 1} W_l
\]

In this sum those subspaces, that add the least significant support to the final sum, are omitted. Interpolants of the sparse grid are represented with the following sum:

\[
f^{(1)}(x) = \sum_{|l| \leq n + d - 1} \sum_{i \in I_l} c_{l,i} \cdot \phi_{l,i}(x)
\]

In Figure 2.4 one can see \( V_3 \) and \( V_3^{(1)} \) decompositions. For a more comprehensive introduction to sparse grids, see [1].
Figure 2.4: The image on the right shows the two-dimensional $W_l$ functions up to $l = 3$ in each of dimensions. The corresponding sparse grid of level $n = 3$ (left) for the sparse grid space $V_3^{(1)}$ consists of green, red and blue subspaces. For the full grid, corresponding to the full grid space $V_3$, both colored and gray ones have to be used.
2.3 Hierarchical decomposition and reconstruction

Often, huge data sets originating for instance from simulations are stored in the form of a full grid, where each cell is associated with a data value. When applications process or transfer such massive data blocks, data movement between main memory and a slow disk or a slow network connection often becomes a bottleneck, rather than CPU time. Thus it is important to consider I/O-efficient algorithms for processing big data grids.

The hierarchical transformation used in the idea illustrated in Figure 2.5 could be effectively used with such algorithms. The hierarchical decomposition of a grid into hierarchy of smaller grids of hierarchical surpluses is discussed in Section 2.1. In Figure 2.5 the process of sending decomposed grid over the network is shown. Smaller grids of surpluses — subspaces — are sent from coarsest level to finest. On the receiver’s side the original data grid can be reconstructed from coarse level information, even if fine level information is not received yet. Also the original grid, e.g. a big image, could be splitted into several smaller grids, so-called tiles, and each of them can be transformed and passed through the network simultaneously. In this Chapter we describe an algorithm to compute the hierarchical splitting — the hierarchization algorithm.

The hierarchization algorithm transforms function values, sampled at grid points, into hierarchical surpluses, using a bottom-up scheme (values in the grid are processed level by level, using $W_L$, starting from $||L||_1 = n$). After that, all original function values can be reconstructed using inverse operations. In the next two subsections we overview the hierarchization algorithm on a full grid in the one- and n-dimensional cases. We use hierarchization to transform a full grid into surpluses. This grid of surpluses, being cut at the diagonal, corresponds to a regular sparse grid (see colored part of the decomposed grid at Figure 2.4).

2.3.1 Hierarchization in the 1-dimensional case

Surpluses in 1D are computed using a bottom-up scheme, which is explained in this section. Level n of the grid can be thought of as a "correction" to the approximation, obtained using the previous n-1 levels. Considering level n of the grid, let $u_i = f(x_i)$ be the function value at the grid point $x_i$ ($x_i$ also on level n). Let $u_l$ and $u_r$ be the function values of $x_i$’s hierarchical neighboring grid points (grid points on levels from n-1 to 1) — left and right respectively. Surplus value $c_i$ is determined as $c_i = u_i - 0.5 \cdot (u_l + u_r)$. These calculations are applied to all grid points on all levels down to level 1, giving a valid hierarchical transform. Figure 2.6 illustrates the hierarchical transform for n = 3. The main difficulty in this algorithm is not the calculation of surpluses, but identifying and accessing each grid point’s hierarchical neighbors.
Figure 2.5: The image illustrates how the hierarchical transformation can be used in a distributed settings where the network connection is slow. A full grid is decomposed into color-coded levels of hierarchical subspaces. Starting with smaller coarse level subspaces, the data is transferred to the receiver (e.g. a render node), where the available information is used to approximate the original full grid data. The inverse of the hierarchical transformation provides a fast way to interpolate values that have not been received yet.
2.3 Hierarchical decomposition and reconstruction

Figure 2.6: These images illustrate hierarchical transformation in 1D. The top left image illustrates the function values to be hierarchized and hierarchical dependencies between them. Three other images represent three stages of the hierarchical transform, where the $\alpha_i$ denote surplus values.

The iterative bottom-up algorithm for hierarchization in 1D can be defined as follows:

**Algorithm 1** 1D hierarchical transform algorithm.

```
for level = n..1 do
    for all function values $u_j$ within level do
        find hierarchical neighbors $u_l$ and $u_r$ of $u_j$
        $c_j = u_j - 0.5 \cdot (u_l + u_r)$
    end for
end for
```

As it can be seen, all dependencies build a kind of hierarchy between coefficients, and all coefficients, connected with each other by such hierarchical dependencies, we denote as *hierarchical tree* (see Figure 2.7). Now it can be easily seen, that in the 1D case every subspace corresponds to one level of the hierarchical tree.
2 Multi-level Subspace Splitting

Figure 2.7: These three images represent 1D hierarchical decomposition and dependencies between hierarchical surpluses. Hierarchical surpluses are illustrated in the left image. Dependencies between hierarchical surpluses are illustrated in the middle image. The corresponding hierarchical tree is illustrated in the right image.

2.3.2 Hierarchization in multiple dimensions

The data structure we use is based on the decomposition of our approximation into several subspaces of regular shape, as detailed in [15].

In Figure 2.8 hierarchical surpluses and dependencies between them are shown. The rightmost part depicts the so-called subspace tableau, in which several instances of the unit square domain can be seen, each containing one subspace of our decomposition. Each dot actually denotes one grid point and the associated surplus value.

We combine all subspaces with the same $|\mathbf{u}|_1$ value into one subspace level. And the dependencies between surpluses can be seen — exact function values of current subspace’s points can be reconstructed only after all values of all subspaces from previous subspace levels will be obtained. The arrows in the right image of Figure 2.8 visualize these hierarchical relationships of the grid points with respect to the horizontal $x$- and vertical $y$-dimension. All function values $u_j$ in Figure 2.6 now correspond to vectors $\mathbf{u}_{x,j}$ and $\mathbf{u}_{y,j}$. The $\mathbf{u}_{x,j}$ values are the values along the projected one-dimensional sampling points $x_j$ (vertical red lines). $\mathbf{u}_{y,j}$ are the values along the projected one-dimensional sampling points $y_j$ (horizontal blue lines). Hierarchical dependencies between $\mathbf{u}_{x,j}$ are shown using gray arrows. Hierarchical dependencies between $\mathbf{u}_{y,j}$ are shown using pink arrows. It can be seen now, that each subspace is one level of the gray hierarchical tree with nodes $\mathbf{u}_{x,j}$, and, at the same time, one level of the pink hierarchical tree with nodes $\mathbf{u}_{y,j}$. Hierarchization in 2D case can be defined as follows:
2.3 Hierarchical decomposition and reconstruction

Figure 2.8: This Figure illustrates the hierarchical decomposition of the full grid in 2D and dependencies between hierarchical surpluses. The full grid \( V_3 \) is shown in the left image. Corresponding hierarchical surpluses are shown in the middle image. Dependencies between hierarchical surpluses in both x- (gray arrows) and y- (pink arrows) directions are shown in the right image.

1. Execute 1D hierarchization algorithm for \( x \)-direction, replacing \( u_j \) by \( \vec{u}_{x,j} \), and applying operations component-wise

2. Execute 1D hierarchization algorithm for \( y \)-direction, replacing \( u_j \) by \( \vec{u}_{y,j} \), applying operations component-wise

Figure 2.9 illustrates two subspace tableaus, in which several subspaces are shown, containing points of the original full grid. To indicate grid points we use integer indices, and their numbering is dictated by the data structure we use (see [10] for detailed description). Each subspace contains points with indices, dictated by the order, in which subspaces are stored in memory. Grid points of every subspace could be considered as a small \( d \)-dimensional full grid. The arrows in the left image visualize the hierarchical relationships of the grid points with respect to the horizontal \( x \)-dimension, in the right image — with respect to the vertical \( y \)-dimension. The coefficients in the full grid in the left part of the Figure 2.9 are stored in \( xy \) order, the \( x \) index being changed at first, and \( y \) index being changed after. On the right image indices are kept in \( yx \)-order.

The coefficients localization does not allow SIMD operations usage between vectors \( \vec{u}_{x,j} \):

1. Coefficients within one \( \vec{u}_{x,j} \) are not aligned in memory (e.g., coefficients 0, 3, 4).
Figure 2.9: These two images illustrate the memory layout for our data structure and data dependencies. Each image contains the subspace tableau with different storage order. In the left image coefficients are stored within 4 subspaces in $xy$-order. In the right image coefficients are stored within 4 subspaces in $yx$-order. On the sides of two tableaus the 1D piecewise linear basis is shown, illustrating how all dimensions are separately refined.
2.3 Hierarchical decomposition and reconstruction

2. Even being grouped in memory, the coefficients of the same full grid may not be aligned in vectors (e.g., coefficients 5, 7).

As it was shown in [2], the second issue is addressed by keeping the coefficients within subspaces. In the left part of Figure 2.9 is shown that hierarchization in $x$-direction could use SIMD operations if the coefficients in the full grids were stored in $yx$-order: 5, 7, 6, 8 instead of 5, 6, 7, 8. And hierarchization in $y$-direction could be improved with SIMD operations, if coefficients 5-8 were stored in $xy$-order (as shown in the right part of Figure 2.9). Coming up to higher dimensions we can use the same ideas.

During hierarchization the whole grid is processed in every dimension — one after another. As was just observed, $x$-dimension is processed in the best manner if subspaces would be stored in $yx$ order, and the $xy$-order is favored for processing $y$-dimension. Thus, after applying the bottom-up scheme the dimensional subspaces order is shifted clockwise. E.g. in 3D case after processing $z$-dimension the order will be changed from $xyz$ to $yzx$, after processing $x$-dimension the order will be shifted to $zxy$ and so on. This process of switching between different subspace order is denoted in [2] as *dimensional shift*. Noteworthy, dimensional shift, after being applied in every dimension, preserves the original subspace layout.

Finally we can remark that working with each dimension of the grid we follow the unidirectional principle — for us there is no difference between all the remaining dimensions and we can process them as one. For more details see [2].
2 Multi-level Subspace Splitting
3 CUDA

3.1 GPU overview

Originally, graphics processing units (GPU) were developed to accelerate graphics of personal computers and game consoles. While not being able to perform all functions of a CPU, GPUs are extremely efficient at performing mathematical computations and processing of huge amounts of data. Usual CPUs have only several cores, working in parallel. GPUs have hundreds to thousands of them and they can process thousands of mathematical operations in parallel. Such high degree of parallelism originally was only used for image processing, but nowadays it is also commonly used to perform scientific tasks. Modern CPUs can run only several high performance threads at a time. GPUs can run hundreds of lightweight threads simultaneously [9]. The architecture of GPUs will be considered in more detail in Section 3.3. Parallel (similar) processing of huge amounts of data on a GPU achieves positive speedups over high-end CPUs. Since Nvidia introduced the first C-based development environment for developers in 2006 [11]. GPU programming was made accessible not only to experts, but also to programmers familiar with C/C++. But at the same time, performance optimization for multi-core processors is still a challenge. The so-called GPGPU (general-purpose GPU) programming was a first approach to perform non-graphics processing on graphics-optimized architectures, typically by running specially modulated code on specially structured data [7].

Early GPGPU programming paradigms had several disadvantages. They required

![Figure 3.1: Schematic CPU and GPU architectures.](image)

...
the programmer to have good knowledge of graphics APIs and the GPU architecture. Many basic programming features were not supported, substantially restricting the programming model. All problems to be solved had to be expressed in unusual terms of textures and shader programs, increasing program complexity [8].

During the last decade several realizations of GPGPU, addressing these problems, were developed. Among them are OpenCL, C++ AMP, Direct Compute and some others. We will overview and use CUDA, the Compute Unified Device Architecture, developed by Nvidia. It is a GPGPU technology, allowing programmers to develop algorithms, using C and C++, and execute them on Nvidia’s GPUs, starting from the GeForce 8 series and newer. We will now discuss CUDA in more detail.

3.2 The CUDA platform

CUDA (Compute Unified Device Architecture) is a scalable parallel computing platform, programming model and a software environment for parallel computing. It was developed by Nvidia and implemented by the GPUs that Nvidia produce. CUDA gives developers direct access to the virtual instruction set and memory of the parallel computation elements in Nvidia GPUs. CUDA enables the GPU to be programmed using C and C++ programming languages [6].

3.2.1 GPU Hardware and programming model

The Nvidia GPU architecture is built around a scalable array of multithreaded Streaming Multiprocessors (SM). A multithreaded program is partitioned into several blocks of threads that execute independently, what means any GPU-optimized program highly scalable — a GPU with more multiprocessors will execute this program in less time than a GPU with fewer multiprocessors.

A multiprocessor is designed to execute hundreds of threads concurrently. To be able to manage large amount of threads, multiprocessor employs an architecture called Single-Instruction, Multiple-Thread (SIMT) (see "Hardware Multithreading”, [6]).

With CUDA, programmers can use a GPU for GPGPU programming. There are several possibilities for that, e.g.:

1. GPU-accelerated libraries for accelerated work with linear algebra, signal processing, image processing, FFT, etc.
2. Compiler directives of so-called OpenACC accelerator programming standard to parallelize loops in Fortran or C code [4].

3. Develop own parallel application, using C, C++ or Fortran extensions.

4. Use PGI Compiler, HMPP and other third parties techniques — list of them could be found in [3].

The third approach includes also the first two and allows programmers to combine them by writing two different parts of the program: host code running on the CPU, and functions that run on the GPU — kernels. In the following we use the expressions coined by Nvidia to identify the CPU and the GPU, respectively: host and device. Host code prepares data for device kernels. Host code can also use OpenACC directives and GPU-accelerated libraries. Kernels are parallel portions of an application, executed on the device.

The CUDA program calls kernels. Each CUDA kernel is executed by an array of threads. That means that each thread runs the same code, implementing the SIMT paradigm. To implement also the Single-Instruction Multiple-Data (SIMD) paradigm and to be able to work with different data in different threads, each thread has an ID, that can be used to compute memory addresses and make control decisions. Each thread within a block executes an instance of a kernel, and has private registers, private memory, per-thread inputs and output results, and a specific id (called thread ID) within its thread block.

The programmer or compiler organizes these threads in thread blocks and grids of thread blocks to allow thread cooperation, e.g. for efficient memory and results sharing. Within one block threads cooperate via shared memory. This per-block memory can be used for data and results sharing, and communication between threads in parallel algorithms. Cooperation between blocks is possible only using the global memory. Threads run in groups of 32, called warps, and 2-16 warps can be organized int 1D, 2D or 3D blocks, and blocks can be organized into 1D, 2D or 3D grids, and each thread and block has a unique 1-, 2- or 3-component identifier (with respect to dimensionality of the corresponding block/grid).

### 3.2.2 Execution model

CUDA uses a straightforward mapping onto hardware — see Figure 3.2:

- A kernel is launched as a grid of thread blocks. Only one (pre-Fermi architectures) or several (Fermi and later architectures) kernels can be executed at a time.
Thread blocks are executed on multiprocessors of the GPU, and several blocks can be executed on one multiprocessor (sharing the common amount of shared memory and registers, available on this multiprocessor).

Threads are executed by thread processors and can communicate within one thread block [13].

In other words: when a CUDA program on the host CPU invokes a kernel grid, the grid blocks are distributed to multiprocessors with sufficient capacity to execute them. All threads within one block execute concurrently on one multiprocessor, and several blocks can concurrently within one multiprocessor. As blocks terminate, new blocks are launched on the freed multiprocessors. All kernel launches are independent, and only within one block kernels can be synchronized with each other, using barrier synchronization instructions. There is one important limitation: kernel launches are limited to a run time of less than 5 seconds on a GPU with an attached display. Exceeding this limit causes a launch failure. GPUs without a display attached are not restricted to 5 seconds runtime. For us, this means that each kernel should not spend too much time processing huge amount of data, when a display is attached to our hardware setup. For more information see "Execution Environment and Memory Model" [6].

### 3.2.3 Managing memory

Each thread has a very limited amount of fast on-chip local memory and on-chip registers to work with. Within one block, all threads have access to the limited amount of shared memory, and also constant memory. The specifics of constant memory are that it is read-only and cached. As a result, on a cache miss a read from constant memory costs one memory read from slow device memory. In case of cache hit, it costs one read from the constant cache. In short, constant memory can be described as follows:

- Data is stored in the device global memory and read through multiprocessor constant cache
- There is only 64KB of constant memory per device and 8KB to work with it per multiprocessor
- Constant memory usage is optimal when the warp of threads read same location
- When the warp of threads reads in different locations — access is serialized and it is as slow as global memory in case of cache miss
3.2 The CUDA platform

Figure 3.2: The Figure illustrates Nvidia GPU from hardware and software perspectives. In the top image the Nvidia GPU architecture is shown: GPU has global memory and a set of Streaming Multiprocessors. Each of the Multiprocessors has several cores, set of registers and a bank of shared memory, usually no more than 48KB. In the bottom image a programmer’s perspective is shown: the CUDA kernel launches a grid of thread blocks. All threads are running the same code. Threads within a block cooperate via shared memory.
Table 3.1: Size and latency of different memory types on an Nvidia GeForce GTX580 (Fermi architecture) [6].

- Access latency can vary from one to hundreds clock cycles, depending on cache hit/miss.

An example of memory specifications (type, amount, access latency) of the Nvidia GTX580 GPU can be found in Table 3.1.

Host and device have separate memory spaces. Host code has also access to device global memory — it can allocate/free memory objects within the global memory of the device, and copy data to and from global memory of the device. Kernels can access only device memory, but, in contrast to the host, kernels can access all kinds of memory on the device, not only the global one. For effective use of the device capabilities, all data read/written during kernel execution more than once, should be copied to a faster memory (shared memory of block or thread registers).

Initially compatible only with Windows and Linux, now CUDA also works on Mac. All Nvidia GPUs starting from the G8x series, including GeForce, Quadro and the Tesla line are supported.

### 3.3 The Fermi architecture

The Nvidia Fermi GPU platform for which we develop, has several key features:

- Up to 16 Streaming Multiprocessors
- 32 CUDA cores per Streaming Multiprocessor
- 64 KB of fast on-chip memory with a configurable partitioning of shared memory and L1 cache
- Unified Address Space with full C++ support
• Up to 16 concurrent kernels
• Up to 6GB of global memory (RAM)

∗64KB of configurable shared memory and L1 cache at each SM allows the programmer to set either 48KB or 16KB of shared memory — the remaining part is configured as L1 cache. That means, that one thread block can use no more than 48KB of shared memory (equals \(3 \cdot 2^{12}\) float numbers).

Using the new primary scheduler, called "GigaThread", Fermi GPUs have the ability to run multiple, independent kernels on different thread groups simultaneously, allowing maximum utilization of GPU resources (see Figure 3.3) [5].

As any of Nvidia’s graphics cards, Fermi has several restrictions on the size of the blocks. The maximum number of threads per block is only \(2^{10}\). The maximum number of resident threads per streaming multiprocessor is 1536 threads. This means that if every block will occupy the whole multiprocessor, there will be no more than 16 blocks in such grid and no more than 1536 threads in each of these blocks [6].

Some general constraints also should be taken into account:

• Threads should be organized in groups of at least 32, with the total number of threads numbering in the thousands for best performance.

• CUDA-enabled GPUs are produced only by Nvidia (strict hardware limitation, unlike most of the other GPGPU techniques, like OpenCL).

The efficiency of the Fermi GPU usage in our concrete case is still a question to be answered. Due to Amdahl’s Law, efficiency of GPU usage is directly dependent on the amount of tasks which can be processed in parallel. In the next Chapter 4.1, we study the hierarchical transform in terms of how well it can be parallelized, and show which parts of the transformation algorithm can be effectively parallelized, considering the capabilities and limitations of the Fermi GPU architecture.
Figure 3.3: Schematic comparison of concurrent and serial kernel execution.
4 Problem Analysis and Solution Approaches

4.1 Potential for parallelization in the hierarchical transform

We conduct this analysis to identify the most effective way of parallelizing the hierarchization algorithm.

The hierarchical transform calculates hierarchical surpluses, based on the function values, using a bottom-up scheme. Within a full grid, according to the Algorithm 1, we process data in each dimension independently, one by one. Within each dimension the bottom-up scheme requires processing subspaces in a certain order — one dimension after another. At the same time, we know, that only hierarchical dependencies within the processed dimension need to be taken into account. That means that we can process each hierarchical tree separately, as long as their belong to a particular subspace doesn’t play any role during calculation of the hierarchical surpluses (see Figure 4.1). The specifics of Algorithm 1 make the transforms within one level independent — all of them can be done in parallel. Thus, we have three levels of parallel execution in the hierarchical transform:

- Process several grids (tiles) in parallel — corresponds to several grids on the GPU.
- Process several hierarchical trees in parallel — corresponds to several blocks within one grid.
- Process elements at each level of a hierarchical tree in parallel — corresponds to several threads within one block.

4.2 Hardware setup

To effectively parallelize the hierarchical transform we need to compute all surpluses within every subspace in parallel, as long as these calculations are independent. From Section 3.3 we know the limitations of the Fermi architecture. The first thing to pay attention to is the size of shared memory on each SM. We use a
Figure 4.1: This Figure illustrates the hierarchical transform in $x$-dimension. Concurrent surplus calculations order on a CPU is shown in the top image. Parallel surplus calculations order on a GPU is shown in the bottom image.
4.3 Parallelization approaches and hardware constraints

As Fermi GPUs can execute several kernels simultaneously, we can transform several grids at once, if they are small enough to fit on one GPU and be processed concurrently. Here, we assume that we are working with only one grid. As was

GeForce GTX580 Nvidia graphics card, whose specifics are described in Table 3.1. This card has Compute Capability 2.0, and, according to the specifications provided in [6] and official specifications of the card, we have the following hardware setup:

- 16 Streaming Multiprocessors
- 64 KB of fast on-chip memory with a configurable partitioning of shared memory and L1 cache (either 48KB given to shared memory and 16KB given to L1 cache, or other way around)
- Concurrent kernel execution (up to 16 concurrent kernels)
- 1.5GB of global memory (RAM)
- up to 1536 threads per SM

4.3 Parallelization approaches and hardware constraints

Figure 4.2: These images illustrate an example of two work items. The left one consists of one hierarchical tree and contains two subspaces. Work item on the right consisting of two hierarchical trees and also contains two subspaces.
discussed in Section 2.3, we transform the whole grid following the unidirectional principle, applying the same 1D scheme to every dimension. Like before, let the dimension in which we are currently applying our 1D scheme be denoted as work dimension.

Further, let the set of subspaces, connected with each other by hierarchical dependencies, be denoted as work item (see Figure 4.2). In every dimension we transform coefficients by the bottom-up scheme. In terms of hierarchical trees that means that we process coefficients level by level, starting with coefficients at the leaf level and finishing with the root. Hierarchical trees do not have any dependencies with each other, that means that all trees can be processed concurrently. Within one tree level, elements are independent, and the corresponding surpluses can be calculated in parallel. Each tree can be processed on one block of threads, level by level, starting from the leaf level.

We choose the number of threads per block equal to the number of coefficients at the leaf level. Since in full grids all trees have the same structure, we can allocate all blocks with the same number of threads. The biggest amount of coefficients we have to process in parallel is given by the widest leaf level of the hierarchical tree.

Each SM in our GPU has up to 48KB of shared memory and $3 \times 2^9$ threads. We create blocks of $2^9$ threads each — to use all SM’s threads. Thus, each block can keep up to $2^{12}$ float coefficients and process $2^9$ of them at once. The biggest number of elements the block has to process at once is number of elements at the leaf level of the hierarchical tree. Since the leaf level of the hierarchical tree is a subspace within work item, we can formulate the following limitation: the maximum subspace level we can process is 9.

When working with multi-dimensional grids, we generally have more than one hierarchical tree. If the number of coefficients on the leaf level of every tree is less than 512 — we can process several trees within one block. Similar nodes of several trees can be considered as one vector node, and we can process several scalar trees as one tree of vectors.

The opposite situation is also possible: if the set of subspaces of one work item grows too large to fit in one block, we divide the work item in several equal slices (one or several hierarchical trees in each of them), and process each slice on a separate block. In the Figure 4.3 is shown schematic process of dividing work items and uploading them to the GPU.

We can formulate the limitations for the full grid we can process as follows:

- No more than $2^{12}$ float coefficients in the grid.
4.3 Parallelization approaches and hardware constraints

- No more than $2^9$ float coefficients in the largest subspace treated.

If the grid we have to transform exceeds these limitations, we can split it into several smaller grids and process them separately. If the grid we have to transform doesn’t reach these limits, we can process several grids of a kind concurrently, but in this case we have to keep in mind another limit: the maximum number of blocks in one GPU grid. This constraint can however be overcome. In our implementation this limit is 65536, so we can process no more than 65536 work items at once.

Finally, our parallelization approach is defined as follows:

**Algorithm 2** Multi-dimensional hierarchical transform on the GPU.

- Upload grid (or grids) to the global memory of the GPU.
- for each $d$ within dimensions do
  - Determine number $N$ of elements at leaf level of hierarchical trees.
  - If necessary, slice work items into smaller work items. Number of work items is $k$.
  - Prepare GPU, create grid with $k$ blocks of reasonable size.
  - Upload work items to the shared memory of corresponding block.
  - Execute 1D hierarchical transform and dimensional shift as kernel.
- end for
- Load back hierarchical surpluses.
Figure 4.3: This Figure illustrates the preparation to the hierarchical transform in x-direction on a GPU. One of the work items is divided into two parts. Every work item is loaded to one of the blocks. Colored blocks 1, 2 and 3 denote three stages of the hierarchical transform. Within each of the blocks all surpluses are computed in parallel.
5 Performance Results and Observations

In this chapter we compare results of the hierarchization transform, performed by the CPU and GPU. The following hardware setup was used:

- Intel Core i7-3770 CPU (Ivy Bridge)
  - 3.40GHz
  - Advanced Vector Extensions (AVX, described in details in [12])
  - 4 cores with Hyper-Threading technology
  - 16GB DDR3-1600 memory
  - L2 cache size: 8192 KB

- Nvidia GeForce GTX 580 (Fermi)
  - Up to 48 KB of shared memory per SM
  - CUDA Capability 2.0
  - Global memory: 1.5GB
  - Constant memory: 64KB

5.1 Benchmarks

We compare the two versions of the code, both transforming $N_t$ full grids (tiles) of identical size from the nodal basis to the hierarchical basis. We use single precision floating point data, as is common in visualization settings. For each dimension $N_{wi}$ denotes the total number of work items within all tiles.

Version A is optimized for parallel execution of the hierarchical transformation of several tiles on the CPU, using all available (4 in our case) hardware threads. Tiles are processed in parallel one by one. A tile is fully transformed using the bottom-up scheme in all dimensions by the same thread. During the transformation tile data is cached. As discussed in Section 2.3.2 and described in details in [2], a dimensional shift is used to allow SIMD operation usage, and AVX is used to vectorize the hierarchization operations.
5 Performance Results and Observations

Version B is optimized for parallel execution of the hierarchical transformation of several tiles on the GPU. Every dimension is processed separately. Tiles are uploaded to the GPU global memory, then processed in each dimension using the bottom-up scheme, and then the calculated surpluses are loaded back to the host memory. Each tile is pre-processed on the CPU (with possible slicing of work items which are too big). Then, on the GPU, we create a computational grid with the number of blocks equal to $N_{wi}$. The size of every block is computed during pre-processing of the tiles as the size of the biggest subspace within a tile. Each work item is loaded to the shared memory of one of the blocks and is transformed by its threads. The meta data of work items, such as addresses of the subspaces in global memory, is kept in constant memory.

We test 2-, 3- and 4-dimensional grids. As every configuration has to be run by both versions of the code, GPU limitations are taken into account. Tiles have not more than $2^9$ elements per dimension, and $N_{wi}$ does not exceed 65536 — the maximum size of CUDA’s 1D computational grid.

The following quantities are measured:

1. Time $t$, spent for processing all tiles
2. Time $t_t$, spent for processing one tile
3. Speedup, i.e. how the ratio between reference time $t$ of the 1-tile case and time $t_t$ changes with increasing number of tiles.

In order to obtain significant measurements, all quantities are measured in three independent runs and are averaged. To get reasonable speedup measurements we choose reasonably small tiles and vary their number $N_t$. The following configurations are tested:

1. Tiles of size 33x33, with $N_t$ varying from 1 to 7000. This leads to a maximum of 49000 work items being processed at once.
2. 1-2500 tiles of size 9x9x9, with $N_t$ varying from 1 to 2500. This leads to a maximum of 62500 work items being processed at once.
3. 1-600 tiles of size 9x9x5x5, $N_t$ varying from 1 to 600. This leads to a maximum of 60000 work items being processed at once.

The results are illustrated in Figures 5.1, 5.2 and 5.3. For each scenario we can see, that the behavior of CPU and GPU graphs is quite the same:

1. At first, with increasing $N_t$, we see the positive effect from parallelization (green areas):
5.1 Benchmarks

Figure 5.1: This graph presents benchmark results for 2D tiles of size 33x33.

Figure 5.2: This graph presents benchmark results for 3D tiles of size 9x9x9.

Figure 5.3: This graph presents benchmark results for 4D tiles of size 9x9x5x5.
5 Performance Results and Observations

- The increase of time $t$ spent in processing all tiles is not proportional to the increase of $N_t$ (see leftmost graphs). E.g. in the 2D case $t$ grows up to 17.5 times slower than number of tiles on the GPU, and up to 24.5 times slower on the CPU.

- The average time spent in processing one tile $t_t$ decreases (see middle graphs). E.g. in the 3D case on the GPU, $t_t$ decreases from 0.08ms to 0.005ms. On the CPU it decreases from 0.09ms to 0.004ms.

- The speedup also grows significantly (see rightmost graphs). E.g. in the 4D case on the GPU the speedup reaches 3.5x, and on the CPU 12.5x.

2. With $N_t$ coming closer to the orange area, the parallelization effect reaches its limit (see border between green and orange areas on the graphs):
   - $t$ gradually grows proportionally to the number of tiles
   - $t_t$ decreases slowly, gradually converging against a constant value
   - Speedup grows slowly, also converging against a constant value

3. When $N_t$ reaches the orange area, parallelization shows almost no further effect (orange areas):
   - $t$ grows proportionally to the number of tiles.
   - $t_t$ becomes constant, fluctuating within a minor error. E.g. in the 2D case it reaches 0.04ms for the GPU and 0.03ms for the CPU.
   - The speedup also becomes constant, insignificantly fluctuating. E.g. in the 2D case it reaches 17.5 for the GPU and 24.5 for the CPU.

The results also show that all parallel tests reach a point of saturation. When $N_t$ reaches this point the parallelization gives no more extra speedup. That means, that on both CPU and GPU the maximum parallelization effect could be seen when processing some reasonable number of tiles — in all our configurations it was $N_t > 250$.

5.2 Comparison with the CPU

As we can see, calculations take longer on the GPU when compared to the fully parallelized and vectorized CPU implementation. On the CPU, each tile is processed by one thread, respectively — the tile data is therefore cached, and the 1D bottom-up scheme is applied to all dimensions without additional data reloading.

The current GPU implementation processes the dimensions one after another, applying the 1D scheme to all tiles in one pass. For each dimension processed, there are two synchronization points on the GPU:
5.2 Comparison with the CPU

- After loading tile data to the shared memory of the block, there is a synchronization point, needed to prevent threads from reading not fully loaded data.

- After processing all tiles with respect to one dimension one more synchronization is needed, before the next dimension can be processed.

The synchronization for each dimension and inefficient resource usage are the possible bottlenecks of the GPU implementation. That is why with increasing number of dimensions the peak speedup on the GPU decreases so significantly, and that also explains why in the 2-dimensional case the GPU peak speedup (17x) is much closer to the CPU peak speedup (24x) than in the 4-dimensional case (3x vs 12x respectively). In the next Chapter we propose several options how to get around at least one of these problems.
5 Performance Results and Observations
6 Conclusions

Our goal in this work was to implement hierarchization on the GPU, using all available levels of parallelization and get at least comparable performance with existing CPU version. The given CPU version was already highly optimized and uses AVX vectorization as well as advanced caching strategies and OpenMP parallelization. As long as the Nvidia GPU architecture and parallelization techniques differs from CPU’s in many ways, the results of final comparison were not obvious.

Almost in all tests the CPU implementation is faster than the GPU one. Nevertheless, in terms of performance CPU and GPU versions behave similarly and the performance gap is rather small. Thus, any improvement in this first GPU implementation can lead to the superiority in speed and performance over the CPU implementation.

Another interesting observation is that there seems to be a specific problem size for which a saturation point is reached in processing units. Interestingly, for both CPU and GPU implementations, the saturation points are almost the same. That means, that the use of our parallelization strategies on the GPU is as effective as the complex techniques (e.g. AVX vectorization) used on the CPU are.

The slight superiority in speed of the CPU implementation can be explained by usage of AVX vectorization, processing tiles in a highly optimized manner while keeping data cached during the transformation. On the GPU, all values of the given tile are transformed independently, thus there must be a synchronization point between processing different dimensions of the problem. Besides, the possible chance to get rid of some synchronization overhead, the GPU version still has several possibilities of improvement.

Maximizing resource usage might be a promising approach. That could be achieved by playing with the size of CUDA blocks. At the moment one block transform one work item, and we choose the size of each block such that the biggest work item can be processed by any of blocks. A dynamic choice of the block size customized to the work item size, combined with division of work items into smaller ones might help here.

Usage of resources remains not optimal, because number of values we transform is halved each time we do a step to the upper level — level data becomes sparser.
That means that even if all threads would be busy processing the first level’s values, the last level will be processed by the only one thread — all others will be idle.

Besides that, GPU implementation could be extended to work with actual sparse grids. The decomposed grids are then cut at the subspace tableau’s diagonal, which leads to hierarchical trees of different sizes. Processing there is so far not supported
Bibliography


Bibliography


