GPU-Based Regression Analysis on Sparse Grids

Steffen Hirschmann

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart
hirschsn@studi.informatik.uni-stuttgart.de

Abstract: Prediction and forecasting has become very important in modern society. Regression analysis enables to predict easily based on given data. This paper focuses on regression analysis on sparse grids using the existing toolbox Sparse Grid ++ (SG^{++}). The core workload of the regression analysis will be implemented on graphics cards using NVIDIA's Compute Unified Device Architecture (CUDA). Therefore, we give guidance how to get high performance when dealing with this particular problem using CUDA enabled graphics cards. We also focus on problems where the datasets are larger than the available device memory. Finally, we present test results for real-world and artificial datasets.

1 Introduction and Related Work

Machine Learning is an important field in computer science. We rely on it every day mostly in predictions and forecasts. Applications reach from traditional weather forecasts up to shopping prediction on the internet or the classification of galaxies in astrophysics. The trend of collecting larger and larger datasets and processing them is called *Big Data*.

We want to employ a traditional technique to discover relationships amongst variables: regression analysis. However, since we want to be able to deal with huge datasets, traditional regression analysis is not applicable since it does not scale linearly in the amount of data points. Therefore, we use the approach of [Pfl10]: Discretize the feature space using sparse grids. Sparse grids [BG04] are a well-known technique to avoid the so-called *curse of dimensionality*, so we are able to deal with datasets that have more dimensions than classical methods allow for. In fact, this approach scales linearly in the amount of data points.

When dealing with huge amounts of data we also need to parallelize the workload. Using graphics cards is known to give high speedups if the application is well parallelizable. However, traditional sparse grids algorithms work recursively on the grid and thus cannot be applied here. Hence, we employ iterative sparse grid algorithms as shown in [HP11]. Recently, the concerning algorithms from the toolbox SG^{++}[sgp14, Pfl10] have been ported to several platforms like Open Computing Language (OpenCL) [HP13] or the Intel®Many Integrated Core Architecture [HKP^{+}12].

In this paper we want to show how to implement these iterative algorithms using NVIDIA's CUDA. We continue by showing how to speed up the algorithms making use of CUDA and other programming concepts. Also, we present techniques to handle datasets that are larger than the available memory of graphics cards, which is very important for dealing with *Big Data*.

After a brief introduction to the concepts of regression analysis in Section 2 we present a variational formulation and the resulting system of linear equations for doing regression analysis on sparse grids. In Section 3, we briefly introduce NVIDIA's CUDA [cud12] programming concept to the reader. Afterwards in Section 4, we show implementation details and concepts to speed up the basic algorithms. We conclude by presenting tests with real-world and artificial data in Section 5.

2 Regression Analysis

In statistics, regression analysis is a method to find relationships between variables; see for example [Syk93]. Given a dataset $S = \{(\vec{x}_i, y_i) | i = 1, \ldots, m\}$ of m d-dimensional vectors $\vec{x}_i$ and their respective function values y_i, the goal of a regression analysis is to determine a function f such that: $\forall i : y_i = f(\vec{x}_i)$. To ensure a good quality of the learned function, one usually divides the dataset S into two independent parts: The training dataset S_{train} where the function is learned from and the testing dataset S_{test} against which the learned function is validated. Typically, one chooses $\frac{|S_{train}|}{|S_{test}|} = 2$.

A simple binary classification can be achieved if the codomain of f is divided into two different classes. Depending on the value of the prediction $f(\vec{x})$ the respective class is chosen.

2.1 Regression Analysis on Sparse Grids

Sparse grids are a discretization technique for dealing with high-dimensional problems. For a general introduction we refer to [BG04]. To understand the presented algorithms, the reader needs to know that the basis functions are build from one dimensional functions via a tensor product approach. These one dimensional functions are linear hat functions, each associated to a certain level l and index i:

$$\phi_{l,i}(x) = \max\left(1 - \left|2^l \cdot x - i\right|, 0\right) \qquad (1)$$

When doing a regression analysis, we want to find a function $f_N \in V_N$, where V_N is some function space. We discretize the feature space [Pfl10] using a sparse grid and use the basis functions from Equation (1) associated to the grid points as basis of V_N. A function $f_N \in V_N$ can be expressed as linear combination of the basis functions:$f_N(\vec{x}) = \sum_{i=1}^{N} \alpha_i \phi_i(\vec{x})$. To ensure well-posedness and a unique solution, we use a variational formulation according to [Pfl10] based on a least squares fit:

$$f_N \overset{!}{=} \underset{f_N \in V_N}{\operatorname{argmin}} \left(\frac{1}{m} \sum_{i=1}^{m} \underbrace{(y_i - f_N(\vec{x}_i))^2}_{\text{data term}} + \underbrace{\lambda \|\vec{\alpha}\|_2^2}_{\text{smoothness term}} \right)$$

Where the data term ensures closeness of f_N to the training data. The squared euclidean norm of the coefficient vector is a simple measure for the smoothness of a function and fast to compute. To minimize this functional, we differentiate with respect to the coefficients α_i which are to be determined and set the derivatives to zero. The resulting system of linear equations (SLE) is

$$\left(\frac{1}{m} BB^T + \lambda I \right) \vec{\alpha} = \frac{1}{m} B\vec{y}, \tag{2}$$

where I is the identity matrix which stems from the smoothness term. B stems from the data term and is defined as: $B_{i,j} = \phi_i(\vec{x}_j)$. So given $\vec{y}$ and V_N via a basis, one can solve the SLE from Equation (2) using iterative solvers as the conjugate gradient method (CG) which can handle high dimensional matrices. When applying CG to the SLE from Equation (2) we only have to provide multiplication with B and B^T. Since these two multiplications are by far the most expensive operations when solving the SLE we focus on them and parallelize them using graphics cards.

3 Compute Unified Device Architecture

The Compute Unified Device Architecture (CUDA) is a parallel computing architecture by NVIDIA, which allows to perform calculation on NVIDIA graphics cards. These calculations are done in parallel by all the cores of the graphics cards. However, if execution paths of the calculations diverge the program will be executed sequentially. This is classified as *Single Instruction Multiple Threads* (SIMT) as an extension to Flynn's taxonomy. Due to this highly parallel architecture, CUDA allows for huge speedups in several applications particularly highly parallelizable ones. Compare [cud12].

3.1 Concepts

CUDA is an extension to the C programming language. It adds three main abstractions: thread hierarchy, device memory and synchronization [cud12]. A computer hosting a graphics card is called a *host* and the card itself a *device*.

Thread Hierarchy Every CUDA thread executes a *CUDA kernel*. This is a special function which can be executed on the device and called from the host. All threads are gathered in blocks. A block can be up to three dimensional. Threads from the same block are scheduled together, reside on the same processor core and share the so-called *shared memory*. As of compute capability 2.x, a block may contain up to 1024 threads [cud12]. Since one block may not cover the whole problem, blocks are arranged in a so-called grid. Both, block size and grid size can be up to three dimensional. The value of these sizes must be specified at kernel launch time. In order to get the best performance one has to carefully choose block and grid sizes, depending on the actual calculations and the amount of memory that needs to be shared.

Memory Hierarchy A CUDA capable device provides different kinds of memory [cud12]. The global memory forms the highest level of the memory hierarchy and has the highest access times. It is shared throughout the whole device and can be accessed from the host. Copying to global memory can be done asynchronously, i.e. parallel to kernel executions. On the lowest level of the memory hierarchy there is the shared memory. It can be accessed very fast but is only shared throughout one block of threads and very limited.

4 Implementation

In order not having to implement basic things like the abstract learning process or sparse grid construction from scratch but to focus on the GPU-based algorithms, we used a sparse grid library called Sparse Grid ++.

4.1 Sparse Grid ++

Sparse Grid ++ (SG^{++}) [sgp14, Pfl10] is a toolbox that allows to use sparse grids without great expense. It is flexible and does not need the initial overhead that has to be spent while implementing sparse grids and corresponding algorithms.

SG^{++}implements regression analysis via different classes. The class responsible for the abstract learning process uses a conjugate gradient solver to solve the SLE (2). So-called operation classes implement the underlying numerical algorithms.

4.2 CUDA Operation Classes

The basic thing we had to do is to provide an operation class that multiplies a vector with B and B^T utilizing graphics cards via CUDA as best as possible. The operation class itself has to do some administrative tasks required by SG^{++}, like measuring execution times.

To evaluate SLE (2) we need the grid's level and indices as well as the dataset. The grid (of grid size g) is passed to the operation class as matrices storing levels $(l_{k,l}) \in \mathbb{R}^{g,d}$ and indices $(i_{k,l}) \in \mathbb{R}^{g,d}$ of the basis functions. Furthermore, the dataset (of dataset size m) $(x_{i,d}) \in \mathbb{R}^{m,d}$ is known to the operation class. These matrices as well as the source

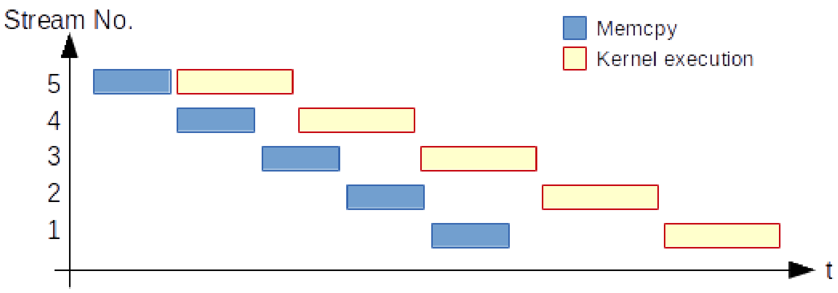

Figure 1: 5 streams each one operating on one of the 5 dataset chunks. Note, that kernel calls may not overlap. However, newer compute capabilities may allow for concurrent kernel execution.

vectors have to get uploaded into the global memory of the device. Therefore, we created a memory abstraction layer.

4.2.1 Memory Abstraction Layer

The memory abstraction layer is responsible for allocating storage on the device, uploading to the device and downloading from it. In between the memory management this layer has to call the actual CUDA kernels, which are implemented in the lowest layer of our architecture.

Streaming Since the global memory of graphics cards (up to 12 gigabytes) is very limited in contrast to host memory, datasets might be too big to fit entirely into the global memory of the device. Another motivation is that we want to reduce copying times, i.e. start the algorithms before every single data point is copied, in order to achieve good overall performance. Therefore, we have to incrementally copy small parts of the dataset onto the GPU, execute the kernels on these parts and then proceed with the next one. The applied algorithm might require for an accumulation of the partial results into one global; in some cases, synchronization might be unavoidable. This partial copy and execution is called *streaming*. There are two different approaches.

The first is to divide the dataset into N equally sized parts and immediately schedule copy and kernel execution on each one. This technique can only be used for datasets fitting onto the device since we cannot guarantee—without explicit synchronization—that only M of these N chunks reside in global memory simultaneously. This might occur if kernel execution takes a lot of time. Figure 1 shows this approach schematically.

The second approach is to divide the dataset into N parts again, however, only using M streams. These streams loop through the N parts and copy them to the card in order to afterwards execute the kernels on them. When a chunk is done, a stream proceeds with the next chunk. This technique ensures, that only M dataset chunks reside in global memory at the same time. The M streams allow to simultaneously copy a dataset chunk and execute

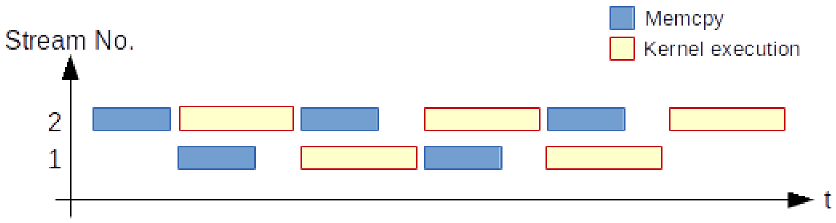

Figure 2: 2 streams iteratively processing 5 dataset chunks. Note that the kernel calls may not overlap. However, newer compute capabilities may allow for concurrent kernel executions.

a kernel on another. This approach is schematically shown in Figure 2 for $M = 2$.

4.2.2 Kernels

The kernels implement the actual multiplication with B and B^T as routines *mult* and *multTranspose*. We implemented the kernels iteratively, adapted from [HP11], rather than recursively. The high-level algorithm of the mult and multTranpose functions are shown in Algorithm 1. Note the basis function evaluation in the innermost loop according to Equation (1). The two algorithms are converted into CUDA kernels as follows: The outermost for loops are omitted and therefore m respectively g threads are launched. Each of these threads has an index corresponding to a loop indices of the former loop variable. So each thread is responsible for calculating one entry of the result vector, thereby iterating over the grid respectively dataset.

```
for $\gamma \leftarrow 0..g-1$ do
    $u_\gamma \leftarrow 0$
    for $\mu \leftarrow 0..m-1$ do
        $p \leftarrow v_\mu$
        for $\delta \leftarrow 0..d-1$ do
            $p \leftarrow p \cdot \phi_{l_{\gamma,\delta},i_{\gamma,\delta}}(x_{\mu,\delta})$
        end for
        $u_\gamma \leftarrow u_\gamma + p$
    end for
end for
```

```
for $\mu \leftarrow 0..m-1$ do
    $v_\mu \leftarrow 0$
    for $\gamma \leftarrow 0..g-1$ do
        $p \leftarrow \alpha_\mu$
        for $\delta \leftarrow 0..d-1$ do
            $p \leftarrow p \cdot \phi_{l_{\gamma,\delta},i_{\gamma,\delta}}(x_{\mu,\delta})$
        end for
        $v_\mu \leftarrow v_\mu + p$
    end for
end for
```

Algorithm 1: Pseudo code for function mult on the left. Calculates $u = Bv$. Pseudo code for function multTranspose on the right. Calculates $v = B^T\alpha$. See [HP11]. The grid has size g, the dataset m, there are d dimensions. A single basis function is described by its level l_γ and index i_γ which itself is a product of one-dimensional basis functions which have an additional index for the dimension. The variable p holds the support of a single basis function.

Using Template Metaprogramming A CUDA thread executing these kernels is most of the time busy incrementing the loop index of the innermost loop, comparing it to the number of dimensions and doing jumps from the loop end to its beginning. This has a huge impact on performance since due to the execution model graphics cards rely on, CUDA cores do not have a branch prediction like modern CPUs.

To tackle the problem of completely removing the innermost loop we employed template metaprogramming. This means, that the CUDA kernels are implemented as C++ templates with the number of dimensions as template parameter. Hence, the compiler can unroll the innermost loop because it has constant trip count. We added a wrapper routine for the kernels which unroll them for a dimensionality up to 100 by default. This bound might be increased arbitrarily. The compiler, then, creates 100 different implementations of a kernel with the inner loop unrolled. A disadvantage of this procedure is that compilation times increase tremendously—depending on the upper unroll bound. A benefit of template kernels is that a thread can keep his associated (multidimensional) grid or data point in registers. Since register count must be determined at compile time, this is only possible if the dimensionality is a compile time constant.

Utilizing Shared Memory The algorithms shown in Algorithm 1 require each thread to access the same data respectively grid point at the same instruction. To achieve better performance, we load this common data into shared memory for faster access. The threads with indices in $\{1, ..., d\}$ in a block will load the data or grid point into shared memory right before the instruction $p \leftarrow v_\mu$ respectively $p \leftarrow \alpha_\mu$. After the shared memory load the threads have to be synchronized in order not to load from undefined memory locations. Another synchronization is needed at the end of the loop to ensure every thread is done with this particular grid or dataset point.

This concept is independent of template kernels, since CUDA allows for specifying the amount of bytes of one single array, that resides in shared memory, at kernel launch time.

Parallelizing Both Outer Loops In most real world cases the grid size is much smaller than the dataset size. If the factor $\frac{m}{g}$ grows too large performance will suffer, since a thread executing a mult kernel, which is parallelized over the grid points, will iterate over all data points. To take care of this problem, the dataset is split into several parts. Every one of these parts can be calculated on different threads. There are two possibilities to achieve this. Firstly, to use a two dimensional block size (x, y) and let all threads associated with the same x component calculate the same result entry. However, we need accumulation of the result inside one single block. If we want this to take place in shared memory, we need runtime determined shared memory (which can only be a one dimensional) of size $x \cdot y$. That would require a huge amount of index calculations. Another problem is that mere a single array inside shared memory can have runtime determined size. So this approach is hardly compatible with the use of shared memory as described in the previous paragraph.

A different approach to overcome an imbalance between grid size and dataset size is to use streaming with a dataset chunk size roughly equally to the grid size. Then, every thread has to iterate over the same amount of elements (no matter which of the two kernels is executed) and hence the overall algorithm parallelizes better. However, we would have to explicitly synchronize this as mentioned in the previous subsection. This approach usually performs much better than using a two dimensional block size and it it simpler to implement.

5 Numerical Tests

First of all, we tested the kernels separately in order to ensure functionality and measure peak performance. Afterwards we ran tests on real-world and artificial datasets to measure real application performance and compare it to the performance of a CPU.

5.1 Benchmarks

The kernels were tested on several NVIDIA graphics cards, namely GeForce 560Ti, GeForce GTX680, Tesla K10 and Tesla K20. We used the first type of streaming presented in the previous chapter. Additionally, we used shared memory. Random numbers in $[0, 1)$ created by `rand()` from the glibc served as input for the data set and the grid's levels and indices. They do not make any sense when interpreted as levels or indices of sparse grids. However, in the benchmarks we wanted to find out the theoretical computing performance of the kernels. Note, that also the vast grid sizes used in the benchmarks are not realistic. Every shown result has been averaged over ten tests. Again, d is the number of dimensions, m the dataset size and g the grid size.

The timings include all memory transfers and the accumulation of the result if using multiple cards or GPUs. Table 1 shows the achieved effective floating point operation per second for single precision. The term "operations" refers to the abstract algorithms shown in Algorithm 1 and not to the operations the GPU really executes; hence index calculations, shared memory loads, result accumulation, etc. are not included.

Table 1: Single precision peak performance. Since we made use of 10 streams, the dataset size is ten times larger than the grid size. Besides, the dataset size is doubled in dual GPU tests since parallelization between different GPUs is based on splitting the dataset. The 560Ti has too few registers, so it was not able to calculate more than 16 dimensions using 640 threads per block. Reducing the number of threads per block would have solved this problem, but we did not want to vary them.

Configuration	m	g	d	eff. GFLOPS (sp)
560Ti	1.280.000	128.000	16	300
K10 (1 GPU)	1.280.000	128.000	24	420
K10 (2 GPUs)	2.560.000	128.000	20	800
GTX680	1.280.000	128.000	24	620
2x GTX680	2.560.000	128.000	20	1.200

To compare double precision performance to single precision performance, we ran a test varying the precision and leaving all other parameters the same. This is in particular interesting for the K20, since NVIDIA presents it as an accelerator card which is ought to be very well suited for double precision computations. Other cards—mostly consumer ones like the GTX680 we used—are known to have low double precision performance. The test results are shown in Table 2. Indeed our test shows that the K20 performs vastly better than all other tested cards when it comes to double precision. It turns out, that the quotient $\frac{T_{\text{dp}}}{T_{\text{sp}}}$ is closer to the one of a CPU than to the ones of the other GPUs.

Table 2: Single vs. double precision benchmark. Measured via the fraction $\frac{T_{\text{dp}}}{T_{\text{sp}}}$ (rounded to two decimal places). Parameters: $m = g = 102400$, $d = 10$. The K20 shows a very good double precision performance in contrast to other tested graphics cards.

Configuration	$\frac{T_{\text{dp}}}{T_{\text{sp}}}$
i7-2600	1.06
K20	1.57
GTX680	5.24

5.2 Dataset Tests

To ensure good performance in real data mining settings we tested the presented implementation on real datasets and compared them to a CPU based version. The dataset tests have been conducted in single precision only. As datasets we used:

Checker Board Dataset The checker board dataset has 30.000 instances and two dimensions. The function values' heights resemble the pattern of a checker board.

Five Dimensional Astrophysical Dataset The five dimensional astrophysical dataset "Fifth Data Release" (DR5) [AMJ07] has 5 dimensions and 431908 data points. The DR5 data set is a real-world data set, that contains photometric data. Regression analysis allows to predict the spectroscopic red shift of galaxies.

Friedman1 Dataset The Friedman1 dataset [Fri91] is an artificial 10 dimensional dataset. It can feature arbitrarily many data points since it is created by a simple function:

$$y = f(x_1, \ldots, x_{10}) = 10\sin(\pi x_1 x_2) + 20(x_3 - 0.5)^2 + 10x_4 + 5x_5 + \epsilon$$

The ten variables $x_1, \ldots, x_{10}$ are all in $[0, 1]$. The latter five serve as noise only. The parameter ϵ serves as additional noise normally distributed with mean 0 and standard deviation 1. We created a training dataset with 50 million points and a testing dataset with 20 million points. The size of the training data is approximately 7.7 gigabytes and the testing data has about 3.1 gigabytes.

5.2.1 Configuration and Results

We basically used the same parameters than for the benchmarks. We did the regression analysis on a GTX680 and for comparison on an Intel®Core i7-2600. All datasets were split into two third training data and one third testing data. We used a sparse grid of level 6 for the checker board and the DR5 dataset and a sparse grid of level 4 for the Friedman1 dataset. The CG solver did 250 iterations. The parameter λ was set to 10^{-6}. We employed streaming for all of the three tests with a chunk size of approximately the grid size. The CPU implementation by Heinecke and Pflüger [HP13] is an optimized version vectorized with Intel's Advanced Vector Extensions (AVX) and parallelized using Open Multi-Processing (OpenMP). The results are shown in Table 3.

Table 3: CUDA and CPU timings of the conducted data set regressions in single precision. For a description of the data sets and the configuration we used see Section 5.2.

CUDA			**CPU**		
C. Board	DR5	Friedman1	C. Board	DR5	Friedman1
0.619667 s	77.9071 s	6 044.16 s	0.664094 s	468.269 s	48 247.5 s

The CUDA version barely pays off with the checker board dataset since it is very small. The little pay off comes from the fact that we used streaming to reduce the initial overhead of copying. Otherwise the CUDA version would perform worse. In the DR5 test the CUDA version out-speeds the CPU-based version by a factor of approximately six. This factor is enlarged once more in the Friedman1 test where the GPU performs approximaltely eight times better than the CPU.

To keep track of which optimization contributes how much to these timings we did timings of the different optimization techniques described in section 4.2.2. The results are shown in Table 4.

Table 4: Per optimization stage timings of the DR5 dataset. The abbreviations: TK stands for template kernels, ST for streaming with a dataset chunk size equaling the grid size and SM for the use of shared memory.

TK	ST	SM	Timing	eff. GFLOPS (sp)
✗	✗	✗	812.242 s	37.7886
✓	✗	✗	156.367 s	196.291
✓	✓	✗	94.2451 s	325.678
✓	✓	✓	77.9071 s	393.976

The Friedman1 test achieved a performance of about 500 effective GFLOPS (sp). This is already near the peak performance of 620 eff. GFLOPS (sp) (see section 5.1). Of course, one could get even nearer to this bound by increasing the grid size.

6 Discussion and Future Work

First of all, we presented how to derive a SLE from the variational formulation of finding a regression function. The abstract work is done by the SG^{++}toolkit, which allowed us to focus on the main work: parallelizing the matrix vector multiplication. We presented how to map the abstract algorithms to CUDA kernels and several techniques to improve the performance. The approach scales linearly in the amount of data, which enables us to handle datasets that are much bigger than we could handle using classical techniques. In order to allow for bigger datasets than the available memory of a graphics card and to have efficient memory management, streaming has to be used. We pointed out two different variants. As far as the author knows this is the first implementation of regression analysis using sparse grids where the dataset size is allowed to exceed the memory of graphics cards. Finally, we presented benchmarks and tests emphasizing the achieved performance.

One important point is auto tuning. In the benchmarks and tests we tuned the parameters, e.g. number of streams, chunk sizes for streaming, number of threads per block and so on, by hand. There are a few general rules of thumb, but no sophisticated system which determines the best parameters. This could, for example, be done by testing different parameter combinations at compile time.

Our two approaches of using two dimensional block sizes and shared memory are hardly consistent. There might be ways to employ both techniques at once. This would be an important improvement since the grid size is almost always clearly smaller than the dataset size and using shared memory also has its advantages.

The foundations of this paper were laid in the beginning of 2013. At that time, CUDA 4.2 was the most recent release supporting compute capability 2.x. Newer CUDA releases might bring better performance and new features which may be useful in the context of this paper.

In general it can be assumed that good quality CUDA kernels run faster on NVIDIA graphics cards than, for example, OpenCL kernels (which are also able to run on graphics cards). However, other techniques—like OpenCL—have advantages of their own. For example, run-time compiled code. As a matter of fact the stake holders of a project should ponder which technique is better suited for their particular project. Since we relied on CUDA we could make use of several CUDA specific features:

CUDA streams We are able to copy and compute on different data simultaneously.

Pre-compiled kernels This can actually be a disadvantage or lead to more work—see the template kernel we had to use in order to unroll. However, if one eliminates this disadvantage there is no run-time overhead.

Explicit exploitation of the memory hierarchy We did make use of shared memory to efficiently share the same data among a block.

Explicit memory management Manually managing memory can bring better performance than using implicit memory management especially when it comes to using streams.

Atomic-operations They allow for easy merging of results of different threads or blocks. (Although they are not the best choice most of the time.)

Acknowledgments

I want to thank several people for supporting me with this paper. First of all, this paper is based on a student project the author conducted with Roberto Fontanarosa and Tobias Rupp. I also want to thank Patrick Diehl who supervised the project and enabled us to run benchmarks on a NVIDIA K20 of the University of Bonn. Therefore, my gratitude also goes to the University of Bonn. Finally, I want to thank Jun.-Prof. Dirk Pflüger who not only supervised the student project but also gave me counsel when writing this paper.

References

[AMJ07] Adelman-McCarthy, J. K. and Johnston, D. E. The Fifth Data Release of the Sloan Digital Sky Survey. *Astrophysical Journal Supplement Series*, 172:634–644, 2007.

[BG04] Bungartz, H.-J. and Griebel, M. Sparse grids. *Acta Numerica*, 13:1–123, 2004.

[cud12] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2012. Version 4.2.

[Fri91] Friedman, J. H. Multivariate Adaptive Regression Splines. *Annals of Statistics*, 19, 1991.

[HKP$^+$12] Heinecke, A., Klemm, M., Pflüger, D., Bode, A., and Bungartz, H.-J. Extending a Highly Parallel Data Mining Algorithm to the Intel(R) Many Integrated Core Architecture. In *Euro-Par 2011: Parallel Processing Workshops: Proceedings of the 4th Workshop on UnConventional High Performance Computing 2011 (UCHPC 2011)*, volume 7156/2012 of *Lecture Notes in Computer Science*, pages 375–384, Bordeaux, France, May 2012. Springer.

[HP11] Heinecke, A. and Pflüger, D. Multi- and Many-core Data Mining with Adaptive Sparse Grids. In *Proceedings of the 8th ACM International Conference on Computing Frontiers*, CF '11, pages 29:1–29:10, New York, NY, USA, 2011. ACM.

[HP13] Heinecke, A. and Pflüger, D. Emerging Architectures Enable to Boost Massively Parallel Data Mining Using Adaptive Sparse Grids. *International Journal of Parallel Programming*, 41(3):357–399, 2013.

[Pfl10] Pflüger, D. *Spatially Adaptive Sparse Grids for High-Dimensional Problems*. Verlag Dr. Hut, München, August 2010.

[sgp14] Technische Universität München. SG++ Documentation. website, April 2014. `http://www5.in.tum.de/SGpp/releases/main.html`.

[Syk93] Sykes, A. O. *An Introduction to Regression Analysis*. Coase lecture. Law School, University of Chicago, 1993.