Bachelor Thesis

In-memory OLAP aggregation on GPUs using CUDA Dynamic Parallelism

Jérôme Meinke

15 May 2015

University of Freiburg im Breisgau
Faculty of Engineering
Department of Computer Science

Reviewer

Prof. Dr. Hannah Bast

Supervisor

Dipl.-Inf. Steffen Wittmer

Abstract

Most queries involved with Online Analytical Processing (OLAP) depend on the functionality of aggregating data along the multidimensional hierarchies of an OLAP cube. In real-time OLAP, aggregated data for interactive operations e.g. roll-up and drill-down is computed on-the-fly. Fast response times are essential and can be accelerated significantly through data-parallel computation on graphics processing units (GPUs). In this thesis, an existing parallel algorithm is modified to use a technology called CUDA Dynamic Parallelism (CDP). Using this technology, GPU programs can be launched directly from within other GPU programs to extract more parallelism. Furthermore, we present a preaggregation method using the CUDA shuffle command to optimize both GPU implementations. For evaluation purposes, we additionally implement a sequential aggregation algorithm. Our experiments show that the single-threaded CPU implementation is outperformed by the GPU implementations by 16 to 218 times. The experiments further show that the CDP implementation reaches a speedup of 3.72 times over the non-CDP implementation when processing queries for an artificial OLAP cube. However, using CDP causes an average of 1.42x slowdown to the processing of queries in a typical OLAP scenario.

Zusammenfassung

Bei der Aggregation werden Werte aus einer großen Datenmenge, z.B. aus einem Datenlager, zu einem oder mehreren neuen Werten zusammengefasst. Dies kann beispielsweise durch die Berechnung einer Summe erfolgen. In dieser Bachelorarbeit geht es hauptsächlich um die Berechnung von Aggregationen zum Zeitpunkt der Abfrage unter Benutzung von Grafikprozessoren. Letztere kann durch die Verwendung von parallelen Algorithmen zu einer Beschleunigung der Berechnung führen. Seit 2013 gibt es eine neue Technologie namens CUDA Dynamic Parallelism (CDP), die es einem Programm für Grafikprozessoren ermöglicht weitere Programme aufzurufen. Es wird untersucht, ob und wie diese technische Neuerung für die Beschleunigung eines existierenden Berechnungsalgorithmus von Nutzen sein kann. Dafür werden nötige Änderungen am bestehenden Algorithmus beschrieben und implementiert. Für die Implementierung werden der zusätzlich nötige Speicherverbrauch durch die Verwendung von CDP und die Antwortzeiten für verschiedene Aggregations-Anfragen analysiert. Weiterhin wird eine schnellere Methode für die Preaggregation von Werten innerhalb kleinerer Ausführungseinheiten (Warps) auf Grafikprozessoren beschrieben. Zum Vergleich der Berechnungslaufzeiten der parallelen Programme mit einem sequenziellen Programm wurde außerdem die Berechnung der Aggregation für einen Hauptprozessor implementiert.

Declaration:

Hiermit erkläre ich, dass ich diese Abschlussarbeit selbständig verfasst habe, keine anderen als die angegebenen Quellen/Hilfsmittel verwendet habe und alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Schriften entnommen wurden, als solche kenntlich gemacht habe. Darüber hinaus erkläre ich, dass diese Abschlussarbeit nicht, auch nicht auszugsweise, bereits für eine andere Prüfung angefertigt wurde.

I hereby declare, that I am the sole author and composer of my Thesis and that no other sources or learning aids, other than those listed, have been used. Furthermore, I declare that I have acknowledged the work of others by providing detailed references of said work. I hereby also declare, that my Thesis has not been prepared for another examination or assignment, either wholly or excerpts thereof.

May 12, 2015, Freiburg im Breisgau

.....
Jérôme Meinke

Contents

1. Introduction

1.1. Motivation

***O**n-**L**ine **A**nalytical **P**rocessing* (OLAP) is an important tool for decision-makers to ensure commercial success. Efficiency and effectiveness are important characteristics for successful businesses in a highly competitive market. These characteristics can be optimized through the skillful management of financial and human resources. Analyzing business data using OLAP can help managers of those resources take actions and support them in the decision-making process. Although the analysis of business data is the main field of application, OLAP can also be used to analyze other data, e.g. social media data from Twitter.

One of the most important computational building blocks in OLAP is the aggregation along multidimensional hierarchies of a data cube. Processing queries, like *drill-up* and *drill-down*, requires the OLAP-software to be able to summarize values of a specified attribute (*measure*) along ranges or subsets of other attributes (*dimensions*). The SUM is the most frequently used aggregation operation, but other operations, such as AVG, MIN and MAX also exist. All queries must be computed fast, since OLAP is typically a highly interactive task expecting fast response times.

The requirements for fast query response times and the ability to collect, store and analyze growing amounts of business data can only be satisfied by the development of more powerful hardware and efficient algorithms. Moore's law [Wik15] states, that the number of transistors in a dense integrated circuit doubles approximately every two years. This has been largely the case until now. However, since 2004, manufacturers started to depart from raising the processor frequency of new products. The costs of increased power consumption and heat generation that come with the frequency-increase have outweighed the benefit of gaining more performance. As a reaction to this, manufacturers have moved away from frequency scaling and turned their focus to parallel scaling. This means multiple processor cores are integrated into a single Compute Processing Unit (CPU) circuit. Nowadays, multi-core CPUs and Graphics Processing Units (GPUs) are the state-of-the-art technology for solving compute-intensive problems cost-efficiently.

GPUs typically provide more processor cores in comparison to CPUs and as such allow for massive parallelization of computational problems. Although the use of GPUs needs specialized algorithms and software, the time and work invested for these purposes

is worth the effort. Many compute-intensive tasks from areas, such as bioinformatics, computational chemistry, machine learning, medical imaging, and weather and climate simulation see a large performance benefit from using GPUs.

Parallel algorithms for the real-time OLAP aggregation have been developed and enable fast computation times through the use of GPUs. This thesis evaluates whether a new technology called *CUDA Dynamic Parallelism* (CDP) can be used to further accelerate the aggregation on GPUs.

CDP allows a so-called *kernel*, the program executed on a GPU device, to be launched from within another kernel. Until CDP existed, a kernel could only be invoked from within the program running on the CPU. This new launch capability can be used to make the GPU device generate more work independently, without further interruption of and communication with the CPU. Furthermore, CDP allows for more parallelism to be extracted from a program. Unfortunately, applying the technology comes with significant overheads in both time and space requirements. The main challenge in using CDP is to employ it in such a way that the generated efficiencies outweighs the additional overhead.

Until now, no GPU aggregation algorithm using the CDP technology, introduced in 2013, was published. Here, for the first time, we provide and analyze such a CDP-enabled algorithm. We also compare its performance to a CPU-powered variant. However, since there was no readily available single-threaded CPU implementation, we implemented a program called Single-threaded OLAP Aggregation Engine (StOAP).

The following section discusses publications on related topics and explains how our work differs from others. Subsequently, we present the rest of the thesis.

1.2. Related work

The GPU powered computation of aggregated values is a well researched topic. In [Kac11], Kaczmarski compares the implementation of a parallel GPU algorithm for OLAP cube creation with its multi-core CPU counterpart. The GPU implementation performs about ten times faster than the CPU implementation. Kaczmarski deliberately did not include the time for transferring data from RAM to the GPU via the PCIe bus, which represents a major bottleneck between CPU and GPU. However, if this time is considered, the CPU implementation is faster. Thus, Kaczmarski concludes that in-GPU OLAP cube

creation only makes sense, if subsequent operations are computed on the data at the time, when it is already present in the GPUs memory. In this thesis, cube computations are only performed after all cube values have been transferred onto the GPUs memory in an initial step. After the initial data transfer, only values resulting from subsequent GPU computations must be retrieved via the PCIe bus. Therefore our test-timings solely include the query-execution times and the additional time needed to retrieve the results.

Multiple papers describe OLAP cube data structures designed for the incremental computation of all aggregations of the cube at the time when its values are inserted. However, in OLAP systems with frequent updates, especially in real-time Business Intelligence (BI), a great number of aggregations are invalid after each change. Therefore using above data structures or Kaczmarski's approach would require the re-computation of a big part of the cube's aggregations after every update. The aggregation algorithm presented in this work supports frequent updates of the cube because it computes only a subset of the values at the time they are requested. It operates on the GPU OLAP cube data structures described by Lauer et al. in [Lau+10]. These allow for fast insertions and deletions, thus enabling frequent changes of cube values.

In the same paper, Lauer et al. also present a target-based GPU aggregation algorithm[1]. Their evaluation shows that individual aggregations of a realistic, but still artificial OLAP scenario could be computed up to 42.6 times faster when using a GPU instead of a multi-core CPU. Their GPU implementation still reaches a speed of up to 12.9 over the multi-core CPU variant using data from a real-world company.

Another GPU aggregation algorithm is presented in [WL13]. According to the authors their source-based approach performs especially well for large and sparse cube areas. Eichel explains this source-based GPU aggregation algorithm in her master thesis [Eic13] and enhances the algorithms performance by introducing multiple optimizations. The main optimization presented in her thesis is the use of a hash table with different numbers of hash functions for the temporary storage of aggregation results. This reduces the amount of atomic operations needed for adding values to existing result cells. Although atomic operations on memory lead to the serialization of work, they are needed to prevent write-after-write hazards and guarantee the generation of correct values. Eichel points out that the optimal number of hash functions depends on various factors, such as the used GPU architecture, the number of kernel threads per block[2] and the amount of source- and target cells of the individual queries. A second optimization presented by

[1] The difference between target-based and source-based aggregation is explained in section 2.3.
[2] Details on the GPU programming model are explained in section 3.1.

Eichel is called warp-preaggregation. It allows to reduce the number of global atomic memory operations by preaggregating intra-warp values which are to be written to the same target cell. This second optimization can speed up the aggregation enormously, if only the values of a few target cells have to be computed. However, if the number of target cells per source cell varies widely, the warp-preaggregation method by Eichel might not work well. We explain this problem and present an alternative in section 4.3.4.

In [Bre+14], Bress et al. create a survey about the design of GPU accelerated Database Management Systems (GDBMS). They conclude that the use of a columnar data storage is best. This is what we are using here as well.

Several papers dealing with CUDA Dynamic Parallelism have been published. The paper [WY14] by Wang and Yalamanchili describes the characterization and evaluation of the implementation of CUDA Dynamic Parallelism in unstructured GPU applications. The authors particularly study the overhead of CDP support in GPUs built on the NVIDIA Kepler GK110, which is the same architecture we are using for our tests. The experiments in the paper show that the CDP implementation can generate a speedup of 1.13 - 2.73 times over non-CDP implementations. The disadvantage however is the non-trivial overhead of CDP, which slows down the overall performance by an average of 1.21. Wang and Yalamanchili also point out that they choose the most straightforward CDP implementation. This means they refrained from controlling the number of child kernels to possibly obtain better performance. Hence, it is assumed that more sophisticated implementations are possible. In this work we attempt to minimize the CDP kernel overhead by controlling the number of executed child kernels.

To our knowledge there exists no publication to date evaluating OLAP aggregation on GPUs under the aspect of using CDP. This thesis provides the first aggregation algorithm using this technology. Additionally, none of the presented algorithms were compared to a single-core CPU equivalent in terms of performance. We have implemented a naive approach of source-based aggregation as a single-threaded program, which we call StOAP. In this thesis we compare the GPU powered implementation with this single-threaded CPU version.

1.3. Thesis structure

The second chapter explains important background information about in-memory OLAP and the multidimensional data cube. Chapter 3 gives an overview about GPU computation, the NVIDIA GK110 microarchitecture, the CUDA programming model and CUDA Dynamic Parallelism. In the fourth chapter, we present both the single-threaded CPU and the parallel GPU variant of the source-based aggregation algorithm. Modifications which need to be applied to exploit Dynamic Parallelism are explained in chapter 4 as well. The results and the analysis of our work are described in the fifth chapter, which is followed by the conclusion and the proposal of future work in chapter 6.

2. OLAP

OLAP[1] has become an important constituent part of Business Intelligence (BI) software, as it enables users to conveniently analyze enterprise data on a query-and-report-basis.

Typically three preliminary steps are executed before OLAP can be used in an enterprise: In a first step business data is extracted from one or multiple data sources, e.g. a data warehouse system or operational systems. As such, OLAP services are separated from live transactional systems and inter-dependencies are eliminated. The second step consists of the transformation of the data to a form suitable for the analysis scenario. In the third and last step the data is loaded by an OLAP server. These three steps are called an ETL process. After these steps are completed, an OLAP server is ready to process user requests.

Commonly, user requests are generated by an OLAP client providing a Graphical User Interface (GUI) and functionality for querying values and creating different views on the data. For example, a user can request that data is analyzed to display a spreadsheet showing all of a company's products sold in Germany in the month of March, compare revenue figures with those for the same products in April, and then see a comparison of other product sales in Austria for the same time period. Furthermore, users might insert or update data at any time, thus requiring the ability of the server software to re-compute query results in real time.

2.1. Data storage and in-memory OLAP

The amount of data that can be managed by an OLAP system and the speed of the query computation depends on the technical implementation of the data storage. It can be implemented as a relational database (ROLAP), a multidimensional dataset (MOLAP), or a hybrid of both (HOLAP). Additionally, there are *in-memory* variants of all implementation models, where all data is kept in the Random-Access Memory (RAM) for fast access.

ROLAP software usually relies on a secondary Relational Database Management System (RDBMS) and communicates with it via Structured Query Language (SQL). Scalable

[1] The term was defined by the English mathematician and computer scientist Edgar F. Codd and his colleages in 1993. See CCS93, pp. 14–17.

RDBMSs enable the processing of large amounts of data. In contrast, there exists no scalable MOLAP implementation as yet and thus only a limited data volume can be managed in-memory[2]. On the other hand, using secondary software is connected with additional work and costs and makes ROLAP rely heavily on the RDBMS' performance. This work focuses on the in-memory MOLAP approach.

2.2. MOLAP data cube

In comparison to relational databases, where data is stored in tables, MOLAP stores data in cells of a data cube. A MOLAP data cube $C(\{D_1, D_2, \ldots, D_d; V_1, V_2, \ldots, V_n\})$ is a dataset consisting of *dimensions* D_i and *measures* V_j. Each dimension defines an axis of the cube and consists of a hierarchy of elements called *attributes*. A point on the space spanned by the cube's dimensions can be uniquely addressed by a spatial key of the form $(e_1 \in D_1, e_2 \in D_2, \ldots, e_d \in D_d)$. A cube *cell* represents a key-value pair.

The element hierarchy of a dimension may contain *consolidated elements* or *base elements*. Consolidated elements are composed of references to other elements of the same dimension. Often *weights* can be assigned to references, defining the factor of how much a referenced element contributes to the consolidated parent element.

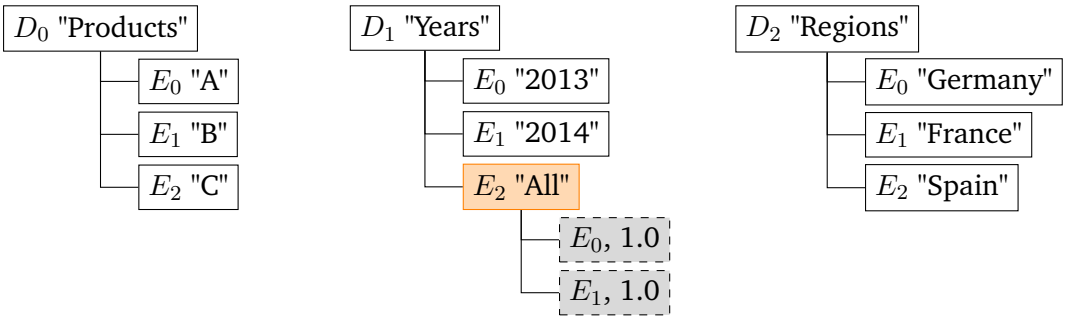


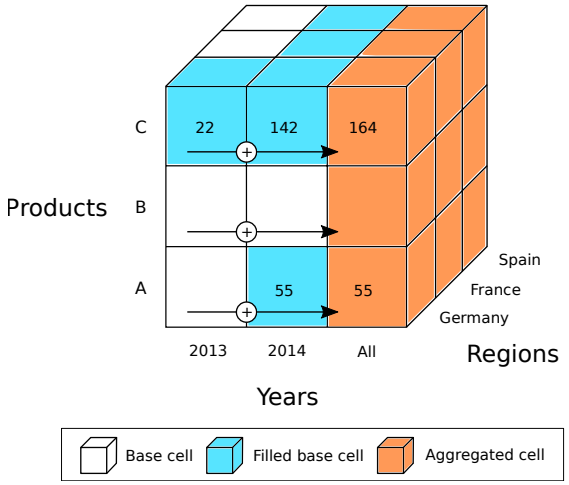
Figure 2.1.: Element hierarchy of dimensions D_1, D_2, D_3.

Figure 2.1 shows the element hierarchy of the three exemplary dimensions Products, Years, Regions. Together, they represent the structure of a cube. The dimensions and base elements in the figure are surrounded by a solid, black border. Consolidated elements are orange. Child elements are surrounded by a dashed border. Child elements consist of a base element reference and a weight.

[2]At the time of writing manufacturers like HP and Dell offer server mainboards, which can be equipped with a total 512GB of RAM.

If the key of a cell contains only base elements, we call this cell a *base cell*. Conversely, if the cell's key contains one or more consolidated elements, then the cell is an *aggregated cell*.

The values of base cells are stored in a so-called fact table, whereas the values of aggregated cells must always be computed from the values of the associated base cells. To reduce the physical space consumption of a cube, only those base cells are stored in the fact table, which actually contain a value. We call these *filled* base cells. In practice, OLAP cubes are very sparsely filled, often less than 1%.



a)

D_0	D_1	D_2	Value
0	1	0	55
2	0	0	22
2	1	0	142
2	1	1	97
2	1	2	23

b)

Figure 2.2.: Data cube and corresponding fact table

Our exemplary cube is illustrated in figure 2.2a. The cube contains five filled base cells, which are colored blue. The fact table, containing the corresponding key-value pairs, is shown in figure 2.2b. The first entry represents the cell with the key (A, 2014, Germany) and the value 55. Aggregated cells, whose spatial key involves the consolidated element *All* with index 2 from dimension D_2, are colored orange. The arrows with a plus-sign represent the aggregation of base cell values along the element hierarchy of the D_2 dimension. The following chapter describes this in detail.

2.3. Aggregation

The aggregation of cell values of a specified attribute (*measure*) along ranges or subsets of other attributes (*dimensions*) can be performed using the operators SUM, COUNT, AVG, MIN and MAX. As the SUM is the most frequently used operator, the aggregation is explained using summation as example.

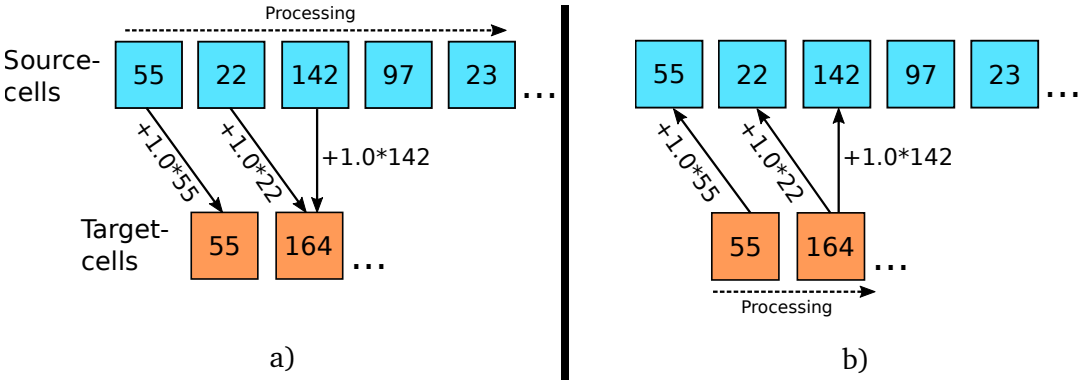


Figure 2.3.: Real-time aggregation approaches.

There are two different approaches for the real-time aggregation of cells.

The first, *source-based* approach iterates all cells of the fact table. For each of the source cells the associated target cells and element weights are looked up. In a next step, each source cell's value is multiplied by the associated weight and added to the value of all target cells.

The second, *target-based* approach iterates all requested target cells. For each of them, the contributing source cells are looked up in the fact table. Their values are retrieved and added to the target cell value, after being multiplied by the associated weight.

The source-based approach is illustrated in figure 2.3a as opposed to the target-based approach, which is illustrated in figure 2.3b.

Both approaches require the use of an *aggregation map*, which is constructed at the time when a query has been received, but before the aggregated cells are computed. The map contains the query meta data, like information about the requested target cells and the contributing source cells with the assigned weights[3].

[3] [Lau+10, p. 79] explains in detail how the coordinates of each requested aggregated cell are broken down into the corresponding ranges of base elements in each dimension along with their weights.

3. Parallel computation using GPUs

NVIDIA, ATI, Intel and other manufacturers have developed a new technology called General Purpose Graphic Processing Unit (GPGPU[1]) allowing for massive parallel processing. Two different programming models exist and can be employed to use this technology: NVIDIA's CUDA, which specifically targets NVIDIA GPUs, and OpenCL, which is an open standard and allows to create programs for CPUs, GPUs and FPGAs from many different manufacturers.

Parallel processing does not represent a universal remedy for the acceleration of programs. To be able to benefit from the use of GPUs, the program's task must be dividable into sub-problems, which can be computed in parallel on the various GPU cores. The results of the sub-problems are finally combined together to retrieve the result of the main problem. This is impossible if every computation step of a problem requires the result from a previous step to carry on. In this case the computation cannot be parallelized. Hence, the acceleration of a program depends on how many of its computational work can be parallelized.

The next section introduces NVIDIA's GPU architecture together with key concepts and features of CUDA. Subsequently, CUDA Dynamic Parallelism is explained in more detail.

3.1. NVIDIA's GPU architecture and the CUDA programming model

The Compute Unified Device Architecture (CUDA) programming model provides runtime libraries and extensions to high-level programming languages, including C and C++. These enable developers to access the computational resources, such as instruction set and memory of NVIDIA GPUs. Mostly, new GPU architectures support new CUDA features and are accompanied with the release of a greater CUDA version. The Compute Capability (CC) designates the general specifications and features of a compute device. At the time of writing, CUDA 6.5 is the last stable version and detailed documentation for it is provided in [NVI14b]. All information about NVIDIA's GPU micro-architecture will be presented

[1] We decided to solely use the term GPU in this thesis, as the majority of high-end GPUs currently available can be used for general purpose computing.

by the example of GK110, because it is the first architecture supporting CUDA Dynamic Parallelism.

In CUDA, a program consists of a set of parallel kernels. A single instance of a kernel is called a thread and is represented by a sequential program. The basic execution unit, a warp, consists of 32 threads. Up to 1024 threads[2] can be grouped together in a block. Furthermore, multiple blocks can be grouped together into a one-, two-, or three-dimensional grid. This hierarchical structure is sketched in figure 3.1.

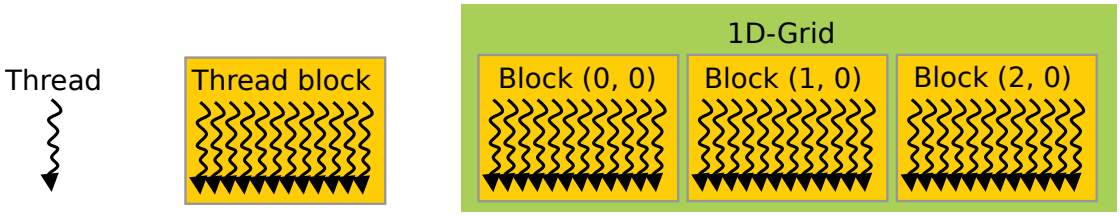


Figure 3.1.: CUDA thread hierarchy, similar as shown in [NVI14b]

Kernel code and launch parameters are usually sent to the GPU device from within a *host* program, which is executed on the CPU. The Grid Management Unit (GMU) of the GPU manages and prioritizes host-originated and CUDA-created work. The CUDA Work Distributor (CWD) unit receives grids from the GMU and forwards them to a maximum of 15 featured Streaming Multiprocessor (SMX) units for execution. Figure 3.2 illustrates a single SMX unit.

While threads are executed, they may access data from multiple memory spaces with different latencies. This in turn provides various possibilities of intra-thread communication and data exchange, which are important features for the functionality of parallel programs. The low-latency, local memory, which each thread might access can be either registers or L1 cache. It is typically exclusively accessible to every thread on its own, although threads may retrieve register values of other threads from the same warp through shuffle instructions. Intra-warp thread communication is also possible by reading and writing the low-latency shared memory. As all threads from a block are executed on the same SMX unit, this memory space also provides the most important way of intra-block thread communication. Communication and coordination of all threads, no matter in what structure and on which SMX unit they are executed, is only possible through the low-latency global memory, which provides space for up to 16GB of data.

[2]The maximum number of threads per block has been increased from 512 to 1024 since CC 2.x.

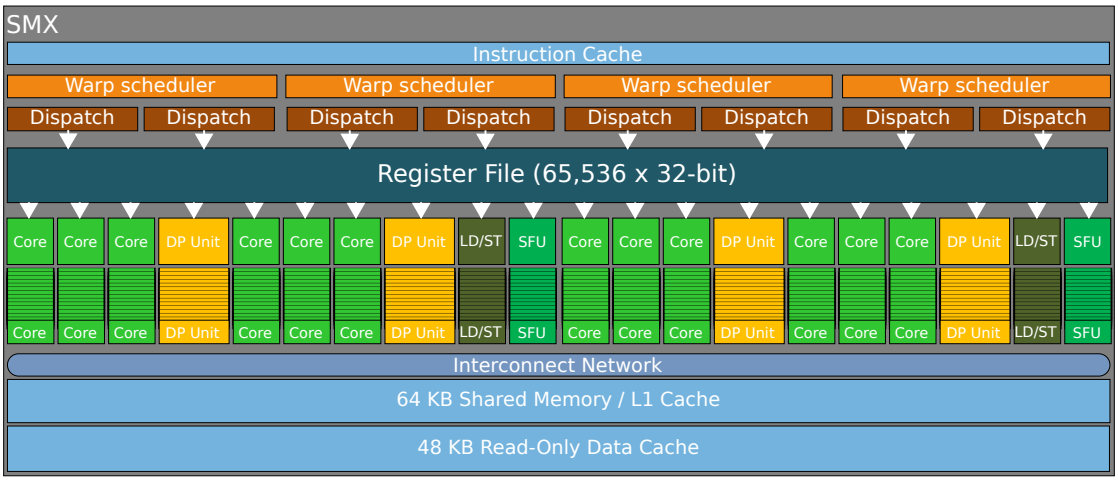


Figure 3.2.: GK110 SMX architecture from [NVI13]. Each SMX features 192 single-precision (Core) units , 64 double-precision (DP) units, 32 special function units (SFU) and 32 load/store-units (LD/ST).

Listing 1 shows the sample code for a kernel that performs N pair-wise subtractions on the elements of vectors A and B and stores the results into vector R. C Language extensions provided by CUDA allow developers to define a kernel using the `__global__` declaration specifier as shown on line 2. The `main` function of the host program prepares the data to be processed and copies it to the device before the kernel is called. Then, as shown in line 11, the `<<<...>>>` execution configuration syntax is used to specify the number of CUDA threads that execute the `vecSub` kernel. The host program can retrieve the resulting data from the device after the kernel was executed.

GPUs are based on *Single instruction, multiple data* (SIMD) processors[3], however they simulate multi-threading on divergent threads (SIMT) through the use of lockstep-execution and masking of active and inactive threads. In the SIMD multiprogramming context, multiple processing elements simultaneously perform the same operation on multiple data points. This requires the data to be stored as array structures in the device's memory. Consequently threads of a kernel can address the data point or range on which they should work on, by calculating an array position from their individual thread index as seen on lines 3 and 4 of listing 1.

[3]SIMD is a classification of computer architectures of Flynn's taxonomy presented in [Fly72].

```
1  // This is a CUDA kernel definition.
2  __global__ void vecSub(int* A, int* B, int* R) {
3    int index = threadIdx.x;
4    R[index] = A[index] - B[index];
5  }
6  // This function represents the host program.
7  void main() {
8    // initialize A and B and copy them to the devices global memory
9    ...
10   // CUDA kernel invocation with one block of N threads
11   vecSub<<<1, N>>>(A, B, R);
12   // retrieve R from the devices memory and process the results
13   ...
14 }
```

Listing 1: CUDA kernel definition and invocation

3.2. CUDA Dynamic Parallelism

CUDA Dynamic Parallelism (CDP) is the ability to launch a new GPU kernel from within another kernel without the need to communicate with the host. The feature was first introduced with the Kepler microarchitecture (CC[4] 3.5). The predecessor of Kepler, Fermi only allowed the launch of a new kernel from within host code executed by the CPU.

A *child kernel* can be launched from within a thread of another kernel, which we call the *parent kernel*.

The problem in using CDP to accelerate an existing algorithm are the costs of starting a new kernel. According to [WY14] the kernel overhead comprises of parameter parsing, calling `cudaGetParameterBuffer` and `cudaLaunchDevice`, as well as the process that device runtime manager setups, enqueues, manages and dispatches the child kernels. Other problems of using CDP are described in section 4.3.1.

In order to benefit from CDP the launching and execution time of the sum of child kernels must be smaller than the execution time of the code which does not make use of CDP. This can only be achieved if more parallelism is extracted and successfully exploited by child kernels. Thus, the new CDP ability must be used with care and we an algorithm must be found that benefits more from CDP than the associated overheads.

[4] See figure A.1 for a detailed table of features supported per CC.

4. Source-based aggregation

The first two sections in this chapter describe a single-threaded CPU and a parallel GPU algorithm for source-based aggregation. Modifications needed to use CDP and further optimizations follow in the third section. We have recognized a straight-forward opportunity to employ CDP within the source-based aggregation approach, hence this work focuses on that. Without loss of generality, we assume that the aggregation operator is summation. All implementations described in this thesis consider weighted summations; for reasons of simplicity we do not consider weights in this chapter.

4.1. Single-threaded CPU algorithm

The single-threaded aggregation on the CPU works in a *Single Instruction, Single Data* (SISD) context. As explained in section 2.3, the aggregation map is computed on the host before the actual computation process starts. Furthermore, a hash table for holding the results is allocated. Only then, the computation of target cells starts: One filled source cell after another is iterated. For each of them the value is added to all assigned target cells.

The above procedure is described by listing 2.

```
1  // iterate entries of the fact table
2  foreach(cell in cube) {
3    // find and iterate target cells
4    while(nextTargetExists(cell->path)) {
5      // build the target path
6      int[dimCount] targetPath;
7      for(dim in dimensions) {
8        targetPath[dim] = getTargetId(cell->path, dim);
9      }
10     // add the cells value to the target cell
11     aggregate(targetPath, cell->value);
12   }
13 }
```

Listing 2: Single-threaded, source-based aggregation

4.1.1. Run-time and space complexity

If there are n filled source cells in the cube's fact table and a query with m target cells must be processed, then in the worst case each of the n source cells contributes to all m target cells. Retrieving the relevant target cells requires the iteration over all d dimensions of the cell's key. Hence, assuming the fact table can be iterated in $O(n)$ time, the run-time complexity of the single-threaded algorithm is $O(mnd)$. Since the execution time depends on n and m, the algorithm is input- and output-sensitive.

Only filled target cells are stored in the hash map containing the aggregation results. Since the maximum of filled target cells is m, the space complexity is $O(m)$.

4.1.2. Single-threaded OLAP Aggregation Processor

The C++ implementation of the single-threaded aggregation in StOAP is based on a heavily stripped-down mix of open-source code from versions 3.1 and 5.1 of the OLAP server[1] by Jedox AG. All features not related to the loading and processing of cube data, such as user management, HTTP request handling, caching, and many more have been removed. We have implemented both a new command-line interface with commands for loading a cube and retrieving cell values and an interface for inter-process communication with a test-application using named pipes. The code for StOAP is licensed under the GPL[2] 2.0 and can be retrieved from the GitHub repository at `jmeinke/StOAP`[3] together with a `README` markdown documentation file.

Cube Data Structure

As all filled cube cells are iterated during the aggregation algorithm, an efficient data structure for the cube's in-memory fact table storage is a key requirement to obtain fast run-times. Hence, this data structure must provide various features:

- support for keys of arbitrary length, as a cube may consist of a large number of dimensions
- iteration over all stored keys and values

[1] Jedox' source code is available at: `http://sourceforge.net/p/palo/code/HEAD/tree/molap/`
[2] The GNU General Public License 2.0 can be found at `https://www.gnu.org/licenses/gpl-2.0.html`.
[3] `https://github.com/jmeinke/StOAP`

- support for insertion, update and lookup
- space-efficiency, because all data is kept in the limited RAM

Multiple publications, such as [AH13], [Böh+11] and [ZZN14], suggest using extensible multidimensional arrays, kD-, prefix-, B^+-trees and other sophisticated data structures for the implementation of a multidimensional storage. Despite this, we could not find a documented, open-source and ready-to-use implementation satisfying all requirements listed above. Choosing the right data structure and implementing it ourselves would take us too far off the subject of this thesis, hence we decided to go for a naive implementation using Google's fast *dense hash map*, which is part of [SHP12].

Each entry of the cube's fact table is represented by an entry in the hash map. In StOAP, cell keys are implemented as `vector<size_t>`[4], while cell values are implemented as `double`. We use the hash function of `boost::hash_combine`[5] to convert a cell key into a hash map key of type `size_t`[6].

4.2. Parallel GPU algorithm

In this section, we describe the various steps of the parallel algorithm presented in [WL13] and optimized in [Eic13]. Subsequently, the modifications needed to use CDP are explained.

The implementation of the algorithm assumes that all source cells are stored as sorted *pages* in the global memory of the GPU before an aggregation query is processed. Building the aggregation map is the first step when a query is received. This step is always executed on the host. Consequently, the map is transferred to the constant memory of the GPU to make the data accessible to the aggregation kernel. The read-only constant memory is well-fitted for this purpose, as low latency access by the threads is possible and no modifications to the map are performed during the kernel execution. The next step consists in reserving the space of a hash table for storing all target cell values in the GPU's global memory.

[4] `http://en.wikipedia.org/w/index.php?title=Sequence_container_(C++)&oldid=638485765#Vector`, 7th of April 2015
[5] `http://www.boost.org/doc/libs/1_57_0/doc/html/hash/combine.html`, 7th of April 2015
[6] `http://en.wikipedia.org/wiki/C_data_types&oldid=654945890#stddef.h`, 7th of April 2015

Typically, the next step is an optional preprocessing step called *prefiltering*. Usually only a subset of the source cells contribute to the target cells of an aggregation. Without prefiltering, all source cells must be processed by the aggregation kernel. If there are only a few relevant source cells, this results in a bad performance, because the GPU resources are not used efficiently. This phenomenon is explained in more detail at the beginning of section 4.3. Hence, first the pages and then the cells are filtered so that only relevant source cells serve as the input to the actual aggregation kernel.

Instead of iterating over the source cells sequentially, the aggregation kernel processes multiple source cells in parallel. As such, each cell is assigned to one thread of a moving kernel grid. Figure 4.1 illustrates the parallel processing of cells, which are colored blue. The page size corresponds to the grid size of the kernel launch configuration. This allows for coalesced global memory access[7].

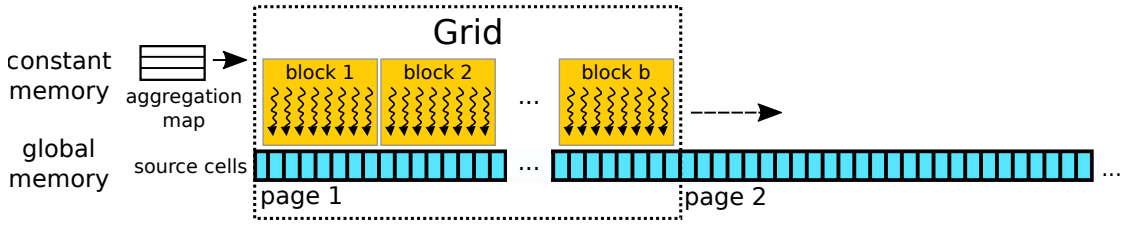


Figure 4.1.: Processing of source cells by a moving grid of threads, from [Lau+10]

When a thread with index i is executed, it first looks up the cell at index i in the current page from global memory. It decodes the coordinates of the cell's key and uses them to look up the assigned target cells in the aggregation map. If prefiltering is performed, there is always at least one target cell assigned. In the next step, each thread looks up the value of its source cell from global memory once and adds it to each of the assigned target cells. After that, the thread jumps ahead by the size of the grid to work on the next page.

Recall the smallest execution unit in CUDA is a warp. Hence, threads can only jump to the next page in groups of 32 threads. Figure 4.2 illustrates how a warp works on the source cells of one page and finally moves on to process the next pages.

As this is done in parallel for all threads in the grid, it is possible that multiple threads attempt to add their value to the same target cell at the same time. To prevent race

[7] Global memory accesses by the threads within a warp can be accelerated, if they are coalesced. Coalescing memory accesses is only possible if all threads within a warp access consecutive

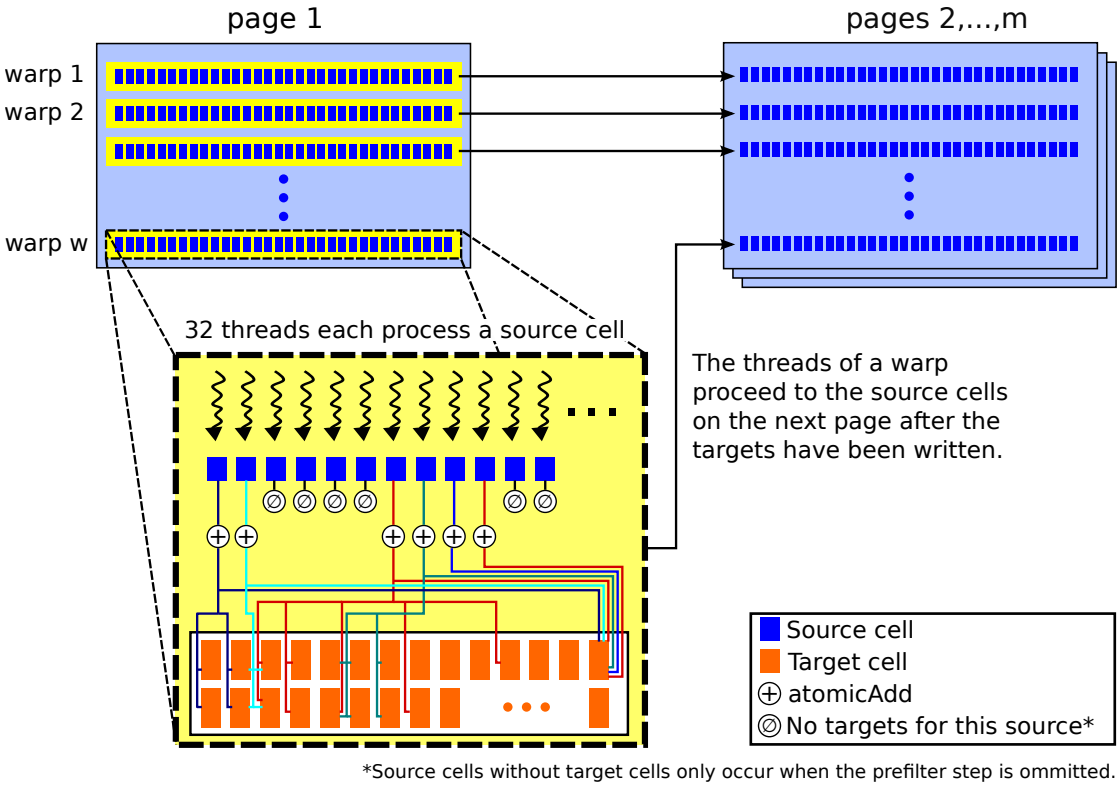


Figure 4.2.: Warp-wise iteration of pages during the aggregation process

conditions, leading to data loss and wrong results, the use of the `atomicAdd`[8] function is required. This way, the read and write requests required to add a value to an existing value are paired together. While a memory address is currently accessed by a thread using `atomicAdd`, no other thread has access to this address.

Unfortunately, using the `atomicAdd` function on global memory slows down the kernel execution. While multiple threads add their value to the hash table in parallel, some of them address the same target cell and need to wait for the other threads to complete their `atomicAdd` command. This serialization of memory requests can severely impede the amount of parallel work. The resources of the GPU, especially the available memory bandwidth, are used less efficiently and the kernel execution is slowed down.

[WL13] and [Eic13] describe methods to reduce the negative effect on the kernel execution time caused by the use of `atomicAdd` at large scale. One method is the warp preaggregation of target cells, another one is the use of multiple hash functions for the

[8] CUDA atomic functions are explained in [NVI14b, pp. 111].

same target cell. Both are described in the next two sections.

4.2.1. Warp preaggregation

Warp preaggregation attempts to reduce the number of accesses to the same hash table position among the threads of a warp. While one thread writes its n-th target cell, all other threads of the warp are either also writing their n-th target cell or idle, because the threads of a warp are executed in lock-step. Shared memory is used such that all warp threads have access to the key of the target cell from the first thread of the warp. Now, all warp threads compare their current target cell key to the one of the first warp thread. The results of the check are saved to a local variable of every thread, however they can be retrieved by other threads through the CUDA `ballot`[9] function. Hence, all source cell values from threads sharing the same target cell key with the first thread can be aggregated in the fast shared memory. Then, only the first thread adds the preaggregated target cell value to the hash table. Consequently, up to 31 concurrent `atomicAdd` operations on global memory can be omitted per warp. Threads not participating in the preaggregation add their value to the hash table on their own, which is not a problem as divergent threads are executed sequentially anyway.

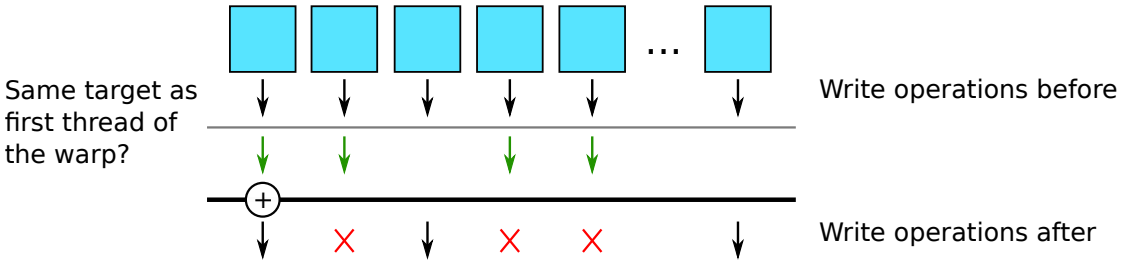


Figure 4.3.: Warp preaggregation, from [Eic13, p. 29]

4.2.2. Multiple hash functions for target cells

Even though `atomicAdd` guarantees that no race conditions occur, the serialization of thread operations lessens the amount of parallel work. The magnitude of serialization can be reduced by using multiple hash functions for the same target cell in the hash table. However, if more than one hash function is used for writing the same cell, duplicate

[9] Warp vote functions are explained in [NVI14b, pp. 115].

entries for this cell are created in the hash table. These duplicates represent intermediate results and must be processed in an additional, final step. This process consists in filtering and summarizing duplicates into a single target cell value using a parallel building block called *stream compaction*[10]. Finally, the result of the aggregation is ready to be transferred back to the host.

4.3. CDP-enabled GPU algorithm

The idea of using CDP within the source-based aggregation kernel is to occupy the GPU resources even more. In an aggregation scenario where adjacent source cells contribute to different numbers of target cells it is probable that some threads of a warp finish earlier than others. While some threads are still processing the target cells of their currently assigned source cell, the threads which have finished their work are ready to jump to the next cell. However, as threads are instructed warp, they can not move on and thus stay idle until all other threads of the warp are ready as well. We can transform the sequential writing of target cells into a parallel process by using CDP and thus reduce the occurrence of idle threads.

The implementation is realized by shifting the functionality of target cell writing from the existing aggregation kernel into a new kernel. Since this new kernel is launched from within the existing one using CDP, we will refer to the two kernels as *parent* and *child* from now on.

Multiple problems had to be solved in order to preserve the functionality of the aggregation algorithm while implementing the new concept. The following section describes the problems involved with the use of CDP and the solutions we have found.

4.3.1. Problems and solutions

To begin with, the structure of the aggregation map does not provide the number of target cells for a source cell. This information is required to communicate to children how many target cells they are supposed to write and how many threads need to run.

[10] Stream compaction is the primary method for transforming a heterogeneous vector, with elements of many types, into homogeneous vectors, in which each element has the same type. A detailed explanation can be found at `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html`.

We have solved this problem by counting the amount of target cells for every source cell in the parent before the child is launched.

Since the invocation of a CUDA kernel only makes sense for relatively big grid sizes[11], launching a child kernel from within each thread for writing its target cells is inefficient. This issue is solved by accumulating the target cell counts for all source cells of a block's page and subsequently launching the child only from within the block's first thread.

CDP does not allow parent and child kernels to access the shared and local memory of each other. Data can be passed on from parent to child only via global memory and via the launch parameters of the child kernel[12]. Since there is no way around this, we need to allocate enough global memory for saving the target cell counts of parent grid blocks before starting the aggregation on the GPU.

Coherent access to global memory with full consistency for parent and child is only guaranteed when explicit synchronization is requested. Without the latter, the execution order between child and parent is not deterministic. This means the parent should not modify parts of the global memory, which are supposed to be read by its child before a parent-child synchronization was performed. Hence, either points for explicit synchronization must be specified or the memory space reserved for information transfer from parent to child can not be reused. We decided to implement both methods.

Finally, the threads of the various child kernels each have their own index, which differs from the index of the parent threads. A child thread does not know what value it is supposed to contribute to which target cell, because it can not determine the source cell from its own index[13]. This problem is solved by computing the parent thread index using an upper bound search on the prefix-sum array containing the target cell counts. A detailed explanation on this follows in section 4.3.3.

4.3.2. Parent kernel implementation

Each thread $T_{i,p}$ of the CDP-enabled parent kernel, with i being the thread index, counts the number of target cells $N_{i,p}$ for the source cell at position i of page p. After that, $T_{i,p}$ writes $N_{i,p}$ to the corresponding index in a shared memory array S. Since this happens

[11] General guidelines for the grid configuration of a kernel in [Mic12, p. 24] suggest to use 1000 or more blocks with more than 128 threads each.

[12] The device runtime reserves global memory space to store the launching information, such as the parameters and configurations for launched, but pending kernels [WY14, p. 7].

[13] Thread and page indices are used to compute the assigned source cell.

block-wise, the size of S corresponds to the block size t. Subsequently, the inclusive prefix-sum of S is computed in place using a naive block-wise parallel scan[14]. After the scan is performed, the last element of S contains the number of target cells to be processed by the child in page p:

$$S_{b-1} = \sum_{i=0}^{b-1} N_{i,p} \tag{4.1}$$

In preparation of the child kernel launch, S is copied to the global memory array D. Finally, the first thread of each parent block calls a child kernel. For reasons of simplicity, an example for the above process is illustrated in figure 4.4 with a much smaller block size of 8 threads.

The launching thread uses S_{t-1} and a fixed block size t_c to calculate the grid size b_c:

$$b_c = \lceil \frac{S_{t-1}}{t_c} \rceil. \tag{4.2}$$

It is also possible to use a fixed grid size, which is evaluated by us as well. In this case, the threads of the child kernel repeat their work in a loop until all required target write operations are done.

Besides a pointer to the global memory array D, multiple launch parameters must be promoted to the child. The work of the parent kernel happens m times per block, with m being the number of pages. Hence, the block's current page index p and the index o of the block's first thread are required by the child threads to be able to calculate the index of the original source cells for their write operations.

Figure 4.4 additionally shows red arrows, which indicate synchronization points. The use of the CUDA `__syncthreads` function is necessary to ensure a consistent access to the values of the shared and global memory arrays by all the threads of a block. The function acts as a barrier at which all threads in the block must wait before any is allowed to proceed [NVI14b, p. 12]. The threads must be synchronized at point A, before the first step of the parallel scan is executed. Additionally, after every of the $\lceil log_2(t) \rceil$ steps of the parallel scan there must be a synchronization point as well. In the case of figure 4.4 these are three steps, each followed by either B_1, B_2 or B_3. Another synchronization point, C,

[14] A *scan* of an array generates a new array where each element j is the sum of all elements up to and including j [HSO07]. The result is called inclusive prefix-sum.

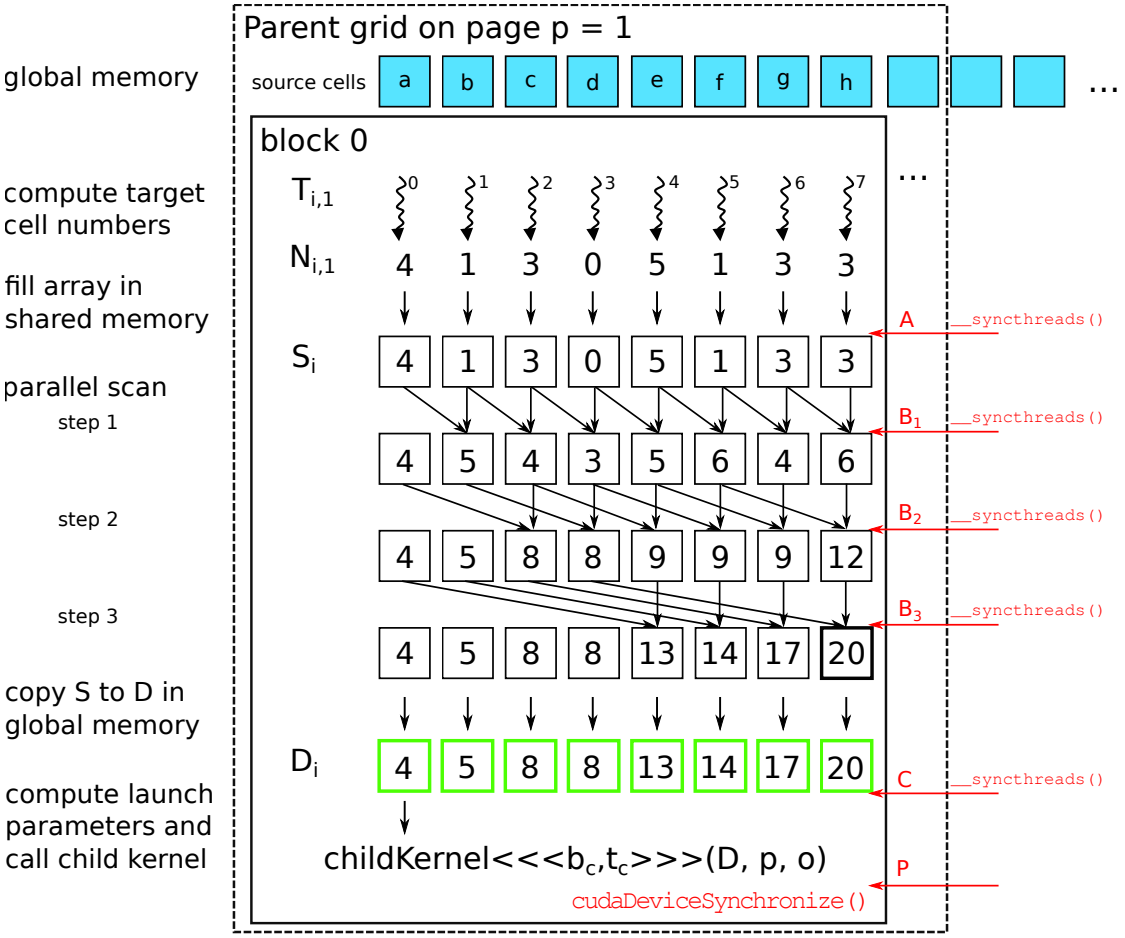


Figure 4.4.: Example of the work of a parent kernel block

is necessary to make sure all threads have written their value to the global memory array D before the child kernel is launched.

P is a point for the explicit synchronization of parent and child. The synchronization method for this purpose is `cudaDeviceSynchronize` and thus differs from the one used at the other points. If P is not implemented, it is possible that the launched child kernel is not executed immediately. Since we do not know when the child is executed and since it needs the information from D, the space used by D can not be reused by the parent. If explicit synchronization is implemented at P, we can overwrite and thus reuse the global memory space required by D after the child is executed. Although this allows us to save space, synchronization costs execution time. A trade-off can be chosen if synchronization at P is only carried out every k launches. This way, the space used to

store target counts can be reduced by the factor $\lceil \frac{m}{k} \rceil$. More information on the space consumption is given in section 5.5.2.

4.3.3. Child kernel implementation

The threads of a child block first copy the array D to a shared memory array S of the same size. Although this requires synchronization of the threads, it subsequently enables fast lookups of values in S. No further accesses to the global memory array D are required. After the synchronization, the threads calculate the index of the source cell they are responsible for. An upper bound search for the child's thread index i on S results in the index P_i of a thread from the parent block. P_i addresses the source cell which is assigned to the i-th target write operation of the block with offset o on page p. At this point each thread computes the index c of its assigned source cell using the parameter values p, o and the page size $g = b * t$:

$$c = p * g + o + P_i. \tag{4.3}$$

Now, every child thread calculates the coordinates of its individual target cell. This requires decoding the source cell coordinates first, then looking up and skipping over the targets in the aggregation map, which are written by other threads, until target $S_{P_i} - i$ is reached. The write operation for this source and target cell combination is exclusive for every child thread. Finally, the source cell value is retrieved and added to the target cell using the `atomicAdd` operation and multiple hash functions, as presented in the original aggregation kernel. Figure 4.5 illustrates the above steps, continuing the example shown in figure 4.4.

Figure 4.5 shows a child block of 20 threads which was the number of overall target cell write operations calculated by the parent block. One can see that threads T_5, T_6, T_7 each aggregate cell c to one of its target cells and thread T_8 aggregates cell e to its fifth target cell. No thread works on source cell d since it did not have any target cells.

We did not implement warp preaggregation in the child as described in section 4.2.1, because obviously for many scenarios there are only a few matching target cell coordinates between the first thread and other threads of a child warp. The preaggregation would only function well in aggregation scenarios with a few target cells. An alternative, more dynamic warp preaggregation is presented in the next section.

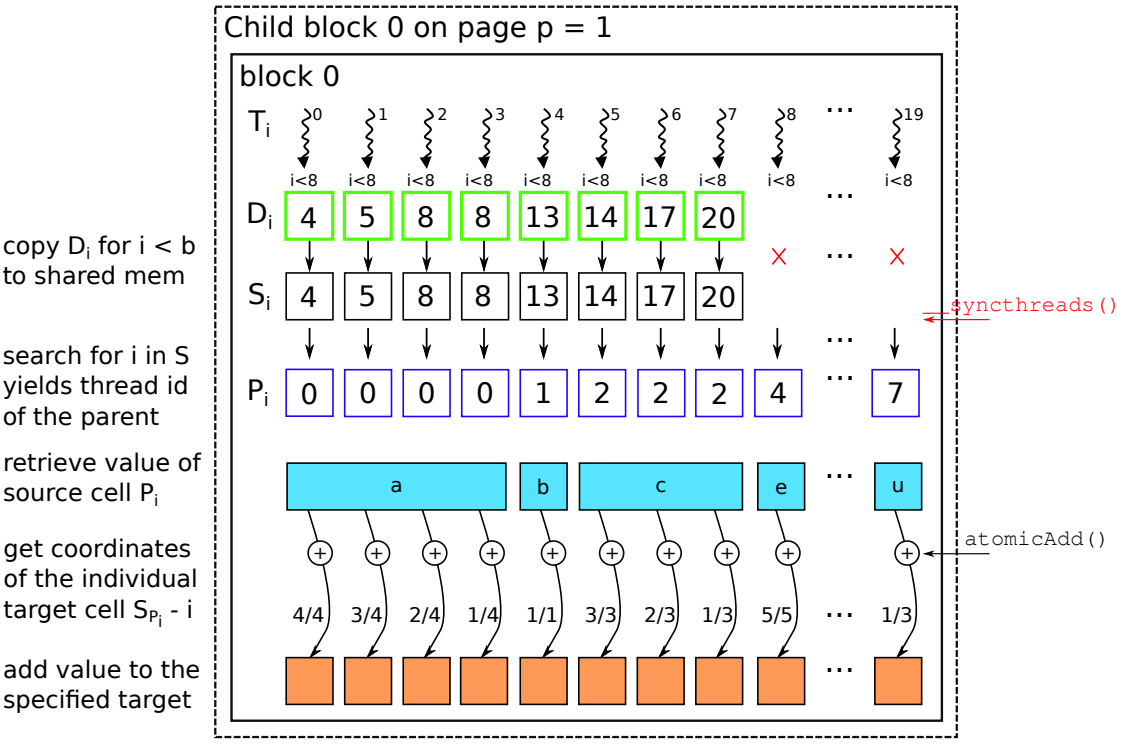


Figure 4.5.: Example of the work of a child kernel block with 20 threads

4.3.4. Warp preaggregation using CUDA shuffle

For reasons of simplicity, threads within a warp are referred to as *lanes* with indices from 0 to 31 in this section[15]. The thread index i of every first lane fulfills the condition $i \pmod{32} = 0$.

Usually warps of the child kernel attempt to write different targets in the same execution step. Thus, as opposed to warps of the original kernel, less threads would participate in the preaggregation. Figure 4.6 illustrates an example scenario where the legacy preaggregation method is used in the child kernel.

In the scenario each of the source cells $c_0, \ldots, c_{10}$ contributes to the target cells x, y and z. As indicated by the green arrows, lanes $0, 3, \ldots, 30$ participate in the preaggregation. Ten `atomicAdd` operations are saved like this, whereas 19 more could be saved.

The following paragraphs describe an alternative method for warp preaggregation in the child. In comparison to the legacy method the new preaggregation saves 29 `atomicAdd`

[15]Using the term *lane* for threads of a warp is introduced in [NVI14b, p. 116].

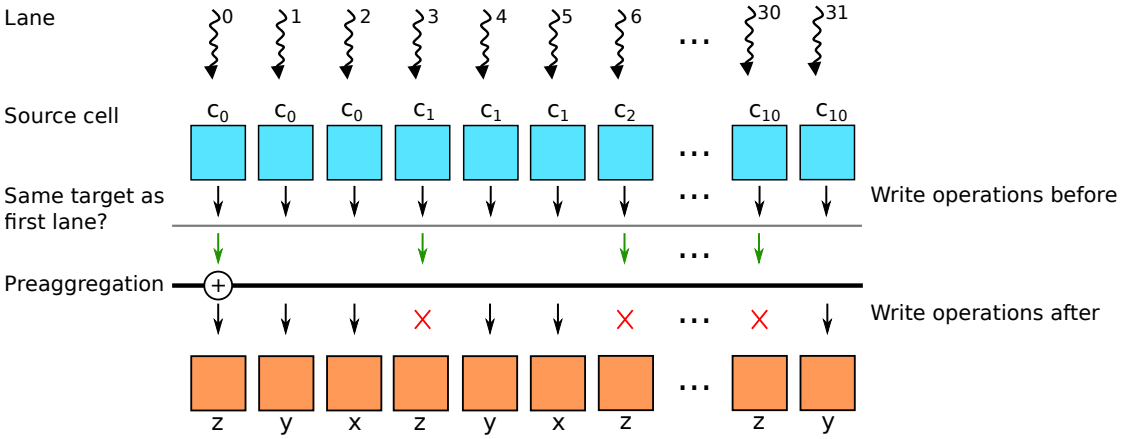


Figure 4.6.: Example of the legacy preaggregation in a child warp with three target cells for each source cell

operations in the above example scenario. This is achieved by using the target count N of lane 0 as a heuristic stride for the comparison of target cells.

The target count N_i for the source cell of thread with index i can be determined by the formula

$$N_i = \begin{cases} P_i - P_{i-1} & \text{if } i > 0 \\ P_0 & \text{if } i = 0 \end{cases} . \tag{4.4}$$

In the following paragraph, the thread index of lane 0 is called q. We use the warp shuffle function `__shfl` to broadcast N_q to all the warp sister threads of lane 0. This way all lanes have access to N_q in a local variable. Since a comparison stride greater than 31 overflows the warp, all lanes check if the precondition $N_q \leq 31$ holds. If the check was successful, they all proceed with the preaggregation. Otherwise, the preaggregation attempt is skipped for the whole warp because all 32 target addresses are different.

If the preaggregation attempt is not skipped, the next step is the calculation of the reference lane index I_{cmp} for the target comparison. I_{cmp} is given by the formula

$$I_{cmp} = l \pmod N, \tag{4.5}$$

where l is the lane index. Each lane then copies the target cell coordinates from the lane I_{cmp} to a local variable using `__shfl`. Afterwards, the lane's target cell coordinates are compared with the obtained ones. The result is saved to a boolean variable `merge`, which is given as an argument to the `__ballot` function. It returns a 32bit integer,

which has the l-th bit set, if the `merge` variable of lane l is true. The above process is illustrated in figure 4.7 using the same example scenario already presented in figure 4.6. Subsequently, the source cell values of the warp are aggregated by the first N_q lanes as described by lines 27-45 in listing 3.

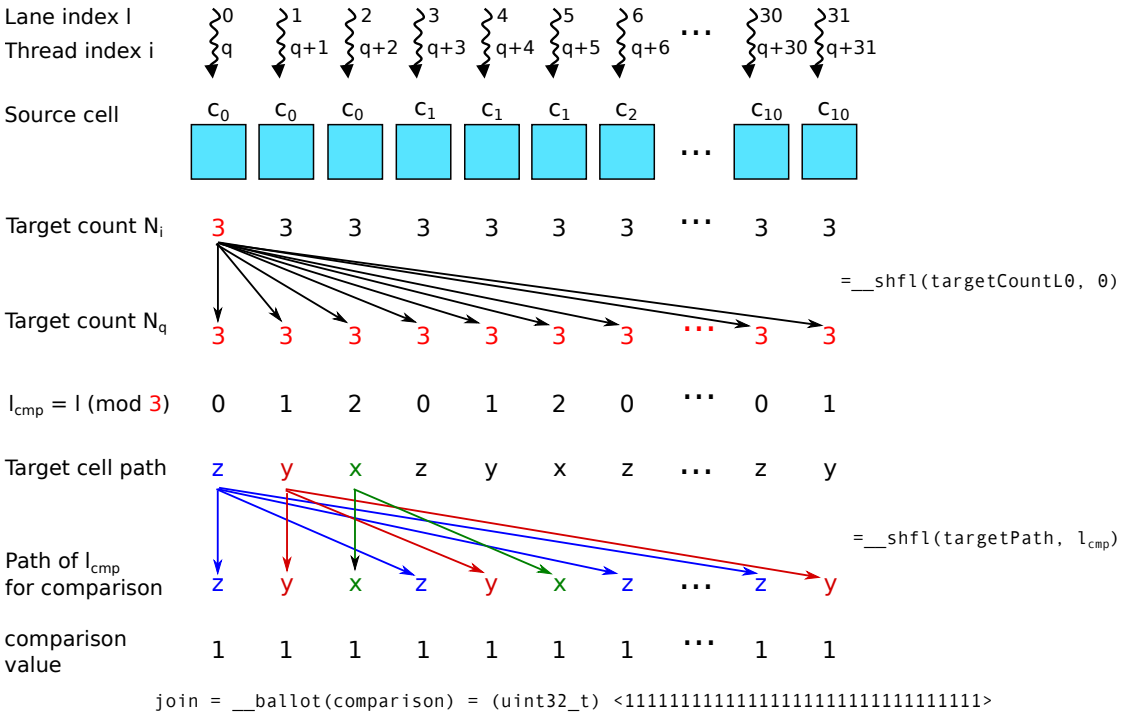


Figure 4.7.: Example of the strided target path comparison in a child warp with three target cells for each source cell

There are some restrictions to be considered when using shuffle. As stated in [NVI14b, p. 117], threads may only read data from another thread which is actively participating in the `__shfl` command. If the target thread is inactive, the retrieved value is undefined. Thus, if we want to retrieve values of other lanes for preaggregation, the relevant lanes must be active. This is possible, if all involved lanes take part in all preaggregations. Consequently, after one preaggregation step, the values of some lanes might be wrong and must be reset to their original value and resetting the value of lanes with an index greater than `targetCountL0` is required after every step. This can be seen in line 36 of the listing. Additionally, a value received by using `__shfl` should only be preaggregated, if the lane from which the value was retrieved is also participating in the preaggregation. This is checked in line 41 before the original value is replaced by the new value of the lane.

```
// variables for later use
gpuPath targetPathCmp;
double origValue = value;

// retrieve the targetCount of lane 0
int32_t targetCountL0 = __shfl(targetCount, 0);

bool writeMerged = false;
if (targetCountL0 <= 31) {
  // compLane is every Nth lane, used for comparison
  // e.g. for N = 3:
  // lanes 0, 3, 6, 9, ...
  // lanes 1, 4, 7, 10, ...
  // lanes 2, 5, 8, 11, ...
  int32_t compLane = laneId % targetCountL0;

  // distribute targetPath of the first m lanes
  targetPathCmp = __shfl(targetPath, compLane);

  // compare own path with the one of compLane
  bool merge = (targetPath == targetPathCmp);

  // since every mth thread compares with its own target path
  // joinInfo will always have at least one bit set
  uint32_t joinInfo = __ballot(merge);

  if (merge && __popc(joinInfo) > targetCountL0) {
    writeMerged = true;
    // shuffle only works for active threads
    // therefore we must loop and recover selectively
    for (int32_t m = 0; m < targetCountL0; ++m) {
      for (int32_t t = m + targetCountL0; t < 32; t += targetCountL0) {
        value += __shfl(value, t);
        // reset the value for all threads but the m-th one
        if (laneId != m) {
          value = origValue;
        } else {
          // check if bit t is set in joinInfo,
          // otherwise lane t is inactive and the
          // value returned by __shfl is undefined!
          if (joinInfo & (1 << t)) origValue = value;
          else value = origValue;
        }
      }
    }
  }

  // we can only omit the aggregation of a thread
  // if it has participated in the preaggregation
  if (laneId < targetCountL0 || !writeMerged) {
    aggregate(targetPath, origValue);
  }
}
```

Listing 3: Warp preaggregation using the warp `__ballot` and `__shfl` functions

5. Results and Analysis

In this chapter, we first describe our test queries and our test configuration. Afterwards, we compare the efficiency of both warp preaggregation methods. Beginning with section 5.4, we present, compare and analyze the query response times of StOAP and the different GPU implementations. Detailed information about the test environment are described in appendix B.

5.1. Test queries

We have chosen two data cubes with different properties as the basis for our test queries:

- The structure of the *Biker* cube was built by Jedox AG for demonstration purposes and reflects a realistic OLAP scenario of a fictitious company. It consists of 8 dimensions and a total number of approximately 6 trillion 680 billion cells. However, the cube actually contains only about 281 million filled cells, corresponding to a fill-ratio of circa 0.0042 percent.
- The *Machines* cube consists of 6 dimensions and about 226 billion cells. Only about 41.3 million cells of the cube are filled, corresponding to a fill-ratio of circa 0.01825 percent.

A cube's dimension order also defines the order of how filled source cell entries are stored in the fact table, since the fact table entries are sorted by cell keys when using the GPU implementation. As such, the dimension order of a cube can be beneficial for the performance of processing one type of aggregation query, while the same order negatively affects the performance of processing another type of query.

We have specially designed the last dimension of the Machines cube using a PHP script to simulate an unfavorable dimension order for our test queries relates to that cube. The dimension contains 64 base elements (*components*) and 2000 consolidated elements (*machines*). Every component contributes to one randomly chosen machine element. Additionally, every 7th component contributes to exactly 1000 machine elements. This design ensures that source cells which are being processed by adjacent GPU threads contribute to different counts of target cells. Dimensions where a few base elements are

consolidated in many elements occur in practical scenarios, however they are more irregularly distributed. Still, the Machine cube enables us to evaluate the impact of idle threads explained at the beginning of section 4.3 on the different GPU implementations.

Table 5.1.: Test-query properties

Query	Source cells	Target cells	
	Relevant	Filled	Overall
B_1	281057088	1	1
B_2	281057088	228	228
B_3	281057088	17460	17460
M_1	41294400	1	1
M_2	6216000	2519	2519
M_3	29534400	13971	874437

Table 5.1 describes the properties of our aggregation queries. For each query the number of relevant source cells and the properties of the target area are described. Relevant source cells contribute to any target cells and are the input to the aggregation kernel after the initial prefiltering step.

We have named queries related to the Biker cube B_1, B_2, B_3. Analogously, queries related to the Machines cube are named M_1, M_2, M_3. Since their target area is rather small, queries B_1 and M_1 build the group S. Queries B_2 and M_2 have more target cells and build the group M. Group L consists of queries B_3 and M_3. Our analysis focuses on the individual queries, rather than on groups, because the queries related to the Biker cube are representative for a practical OLAP scenario, whereas the queries related to the Machines cube represent a non-optimal OLAP scenario.

5.2. Test configuration

The performance of the source-based GPU implementation depends on various parameters, such as the kernel grid and block size and the number of parallel hash functions. In this thesis, we use a launch configuration of 30 blocks and 768 threads per block. Appendix C explains the validity of this configuration. Child kernels launched by the CDP implementation use a fixed block size of 1024 threads, since preceding empirical tests have shown this block size to result in the fastest processing times for all types of aggregation queries in comparison to using smaller block sizes.

According to [Eic13], the optimal number of parallel hash functions depends on the kernel launch configuration, the GPU model and the size of the queries' target area. The tests in [Eic13] were performed on a Tesla C2070. Furthermore, grid and block sizes of 84 and 256 respectively were employed for the launch of the aggregation kernel. Eichel concludes that using 512 parallel hash functions with warp preaggregation is best for queries with small target areas (1-3 filled target cells), while using 64 parallel hash functions without warp preaggregation is best for queries with medium to big target areas (56-22841 filled target cells).

We can not reuse the above results for the number of hash functions as we use other test cubes and queries, a different kernel launch configuration and because our tests are performed on a Tesla K40c. We must additionally assume that the optimal number of parallel hash functions depends on the individual aggregation kernel because the method how the CDP implementation aggregates target cells differs from the method used in [Eic13].

In order to fairly compare the non-CDP and CDP implementation, we attempt to find the best possible performance times for both of them. Thus, we first measure the response times of each query-kernel combination once for each hash function number in $\{4, 8, \ldots, 128, 196, 256, 512, 1024, 2048, \ldots, 12288\}$. Subsequently, a near-optimal number of parallel hash functions is identified by the fastest query response time. Finally, using this number, the average of 10 further measurements is calculated and used as the value for performance comparison.

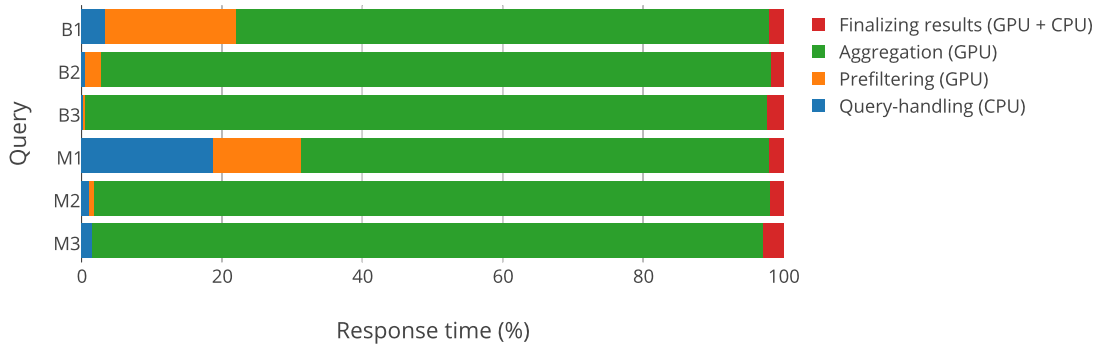


Figure 5.1.: Subtask durations in percentages of the query response time when using the non-CDP GPU implementation

Our performance tests focus on the whole query response time, rather than looking at the performance of individual subtasks involved with the query processing. Like this, the

tasks of handling the query, prefiltering relevant cells, aggregation, and finalizing results are considered in the evaluation. This matters, since it could happen that the execution time of the aggregation kernel is reduced, but in return the time for finalizing results increases[1]. Figure 5.1 shows the percentages of the overall response time consumed by the subtasks at the example of the non-CDP implementation. The figure shows that most of the processing time is consumed by the aggregation kernel for all queries. The time for prefiltering only plays a significant role for the queries of group S.

5.3. Warp preaggregation efficiency

Using warp preaggregation in the GPU implementation can significantly influence the query response time. In [Eic13, p. 30–31], the processing of a query with three target cells is accelerated by a factor of 1127.8 through using warp preaggregation. We compare the efficiency of the two warp preaggregation methods first because the results of this comparison are relevant for the analysis of response times later.

The comparison focuses on the amount of global memory `atomicAdd` operations performed for each query. In order to obtain the numbers described in table 5.2 we have replaced the variable holding the aggregation value by a hard-coded `1` in all three GPU implementations.

The preaggregation method presented in section 4.2.1 uses shared memory for the target path comparison and preaggregation of values, while the second method presented in section 4.7 achieves the same using register variables and the CUDA `__shfl` command. From now on the methods will be referred to as non-CDP and CDP preaggregation respectively. In warps of the non-CDP preaggregation all lanes compare their target cell with the warp's first lane. Warps of the CDP preaggregation use a stride for the comparison.

A warp preaggregation efficiency of 100% corresponds to an `atomicAdd` reduction factor of 32, as this is the maximum of target cells which can be preaggregated in a warp. As described in table 5.2, both methods achieve an efficiency of 100% for the queries of group S, since they reduce the number of atomic additions by a factor of 32.

[1] The process of finalizing results consists of the compaction of target cell duplicates and the transfer of results to the host.

Table 5.2.: Efficiency of warp preaggregation methods

Query	Preaggregation method				
	none	non-CDP		CDP	
	#atomicAdd	#atomicAdd	efficiency	#atomicAdd	efficiency
B1	281057088	8783034	100%	8783034	100%
B2	5021960992	324503612	48.36%	3418446050	4.59%
B3	4262699168	142364188	100%	2328197090	5.72%
M1	41294400	1290450	100%	1290450	100%
M2	1418491200	1348573800	0.16%	1418491200	0.00%
M3	4590532800	4568044950	0.02%	4581902770	0.01%

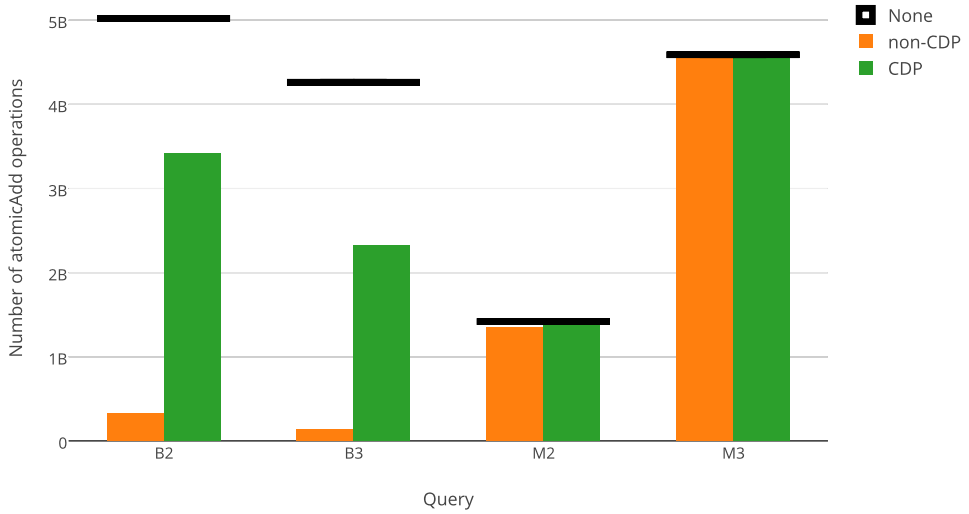


Figure 5.2.: Number of `atomicAdd` operations for group M and L queries using non-CDP or CDP warp preaggregation or no preaggregation at all

Figure 5.2 shows that queries $B2$ and $B3$ benefit much more from non-CDP preaggregation in comparison to the CDP one. The latter is less efficient because its strided target cell comparison works only for source cells contributing to less than 32 target cells. However, many source cells of queries $B2$ and $B3$ have more than 31 target cells. This results in the relatively low efficiency of CDP preaggregation for these queries. Despite the low efficiency, the number of atomic additions is reduced by circa 32% for query $B2$ and by circa 45% for query $B3$. $M2$ and $M3$ do not benefit at all from any of the methods because the queries feature a big variance of target cells per source cell, which largely prevents warp preaggregation.

5.4. StOAP and non-CDP GPU implementation

Table 5.3 shows the query response times of StOAP and the non-CDP GPU implementation. The optimal number of parallel hash functions (#PHF) chosen by us is provided with the response times for every query-kernel combination[2]. The star (*) in the table heading indicates that warp preaggregation was enabled. Bold-printed times indicate the smallest response time for the corresponding query.

Table 5.3.: Query response times of StOAP and the non-CDP implementation (*with warp preaggregation)

Query	StOAP	non-CDP		non-CDP*	
	Response time	Response time	#PHF	Response time	#PHF
B1	110,777 ms	2,133 ms	3072	**1,457 ms**	24
B2	487,705 ms	22,135 ms	3072	**10,394 ms**	4
B3	387,305 ms	42,032 ms	36	**22,734 ms**	28
M1	8,498 ms	283 ms	2048	**179 ms**	44
M2	793,631 ms	**3,634 ms**	4	5,484 ms	8
M3	256,578 ms	**58,648 ms**	4	60,564 ms	4

When looking at StOAP's response times, we notice that surprisingly the times of group L queries are smaller than the times of group M queries. StOAP's query response time seemingly increases with the amount of addition operations per query, which can be obtained from the second column of table 5.2. At least this seems to be the case for queries $B1$, $B2$ and $B3$.

The measurements further show that group S queries are processed most rapidly by the non-CDP implementation with 24 and 44 parallel hash functions using warp preaggregation. The optimal configuration for queries of groups M and L differs in terms of the usage of warp preaggregation. Queries $B2$ and $B3$ are processed faster with warp preaggregation, whereas queries $M2$ and $M3$ are processed faster without. This is related to the efficiency of the warp preaggregation described in table 5.2. Whereas queries $B2$ and $B3$ profit significantly from warp preaggregation, the number of `atomicAdd` operations for queries $M2$ and $M3$ is only reduced so sparsely that the overheads of using the preaggregation lower the kernel's performance.

[2]Figure E.1 from appendix E highlights the chosen number for the queries processed by the non-CDP kernel without warp preaggregation.

Before moving on to the evaluation of response times using the CDP implementation, we first evaluate the use of warp preaggregation using CUDA shuffle in the non-CDP implementation.

5.4.1. Preaggregation using CUDA shuffle

When a comparison stride of one is used by the CDP warp preaggregation, all warp lanes compare their target path to the one of the first lane, which is exactly the functionality of the non-CDP preaggregation. Hence, we can use the CDP preaggregation with a stride of one as a replacement for the existing method to evaluate the effect of using `__shfl` versus using shared memory in the non-CDP implementation.

Table 5.4.: Speedup of warp preaggregation using `__shfl` over using shared memory in the non-CDP implementation

Query	**non-CDP warp preaggregation using `__shfl`**		
	Response time	#PHF	Speedup
B1	1,367 ms	24	1.07
B2	8,559 ms	4	1.21
B3	15,209 ms	28	1.49
M1	163 ms	44	1.09
M2	4,392 ms	8	1.25
M3	60,909 ms	4	0.99

The speedup values in table 5.4 relate to the response times described in table 5.3. Speedups of up 49% can be reached by the non-CDP implementation while employing the preaggregation method using shuffle as opposed to using the method using shared memory. We attribute this effect to the omission of multiple thread divergency points through the use of shuffle.

As a last point, when comparing StOAP's performance with the fastest response times of the GPU implementation, the latter is clearly faster reaching speedups from 4 ($M3$) to 218 ($M2$) times over StOAP.

5.5. CDP implementation

In this section, we compare and evaluate the performance of various CDP implementations. Since we want to fairly compare both GPU implementations, we use the fastest times presented in tables 5.3 and 5.4 as a reference for the calculation of response time speedups.

The different CDP implementations originate from two configuration options for launching child kernels. The first option is related to the grid size of child kernels, which can be dynamic or fixed. If there are more than 1024 target cells for a page's parent block, the work can be assigned to multiple blocks of the child kernel. However, the grid size of child kernels can also be fixed to a single block. This can influence the scheduling of child kernels by the GPU's CWD unit and is evaluated in section 5.5.1. The second option is represented by the introduction of explicit parent-child synchronization, which allows to restrict the global memory consumption used by the parent-child information transfer. This is evaluated in section 5.5.2.

Table 5.5.: Speedup of the CDP implementation over the non-CDP variant (*with warp preaggregation)

Query	CDP implementation			CDP implementation*		
	Response time	#PHF	Speedup	Response time	#PHF	Speedup
B1	2,327 ms	6144	0.92	**2,083 ms**	36	0.66
B2	17,589 ms	108	1.26	**16,511 ms**	108	0.52
B3	34,804 ms	52	1.20	**27,458 ms**	48	0.55
M1	273 ms	5120	1.04	**239 ms**	88	0.68
M2	**4,315 ms**	8	0.84	4,509 ms	4	0.97
M3	**16,101 ms**	4	3.64	16,280 ms	4	3.72

We first test the CDP implementation using a dynamic child kernel grid size without explicit parent-child synchronization. Its response times are shown in table 5.5.

When comparing the best response times of both GPU implementations (non-CDP vs. CDP), we recognize that only query $M3$ benefits from the CDP implementation. This can be seen in figure 5.3.

Several causes explain why the non-CDP implementation processes all queries except for query $M3$ faster than the CDP variant. In the first place, the non-CDP kernel does not need any synchronization points as opposed to the CDP kernel, which synchronizes

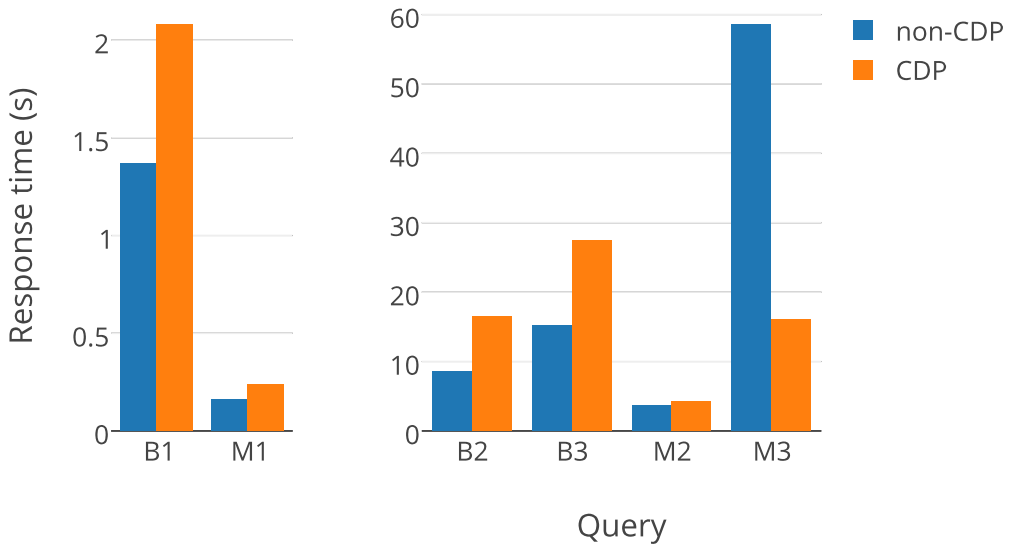


Figure 5.3.: Best response times using the non-CDP and CDP implementations

threads 12 times for each page processed by the parent kernel and once for each child kernel block. The synchronization using `__syncthreads` is needed to make sure all threads of a block have written or read global memory values required for parent-child information transfer. Synchronizations and global memory accesses are costly in terms of performance and might be more expensive than the launch of child kernels.

When processing query $M3$ using the CDP implementation there is a speedup of 3.64 over the non-CDP variant. We conclude the GPU resources are not efficiently used by the non-CDP kernel, since lots of warp threads are idle while one warp thread performs sequential contributions to 1000 target cells. All threads of the parent kernel still use a loop with possibly different iteration count as done by the non-CDP kernel for counting the target cells. However, only simple write operations are performed for the parent-child information transfer as opposed to directly writing target cells using the `atomicAdd` operation. In this case, the inefficient use of GPU resources caused by idle threads of the non-CDP implementation has a worse effect on the performance than the overheads of using CDP. Hence, query $M3$ is processed faster by the CDP implementation.

5.5.1. Optimization of child kernel scheduling

In contrast to launching child kernels with a dynamic grid size, the second CDP-enabled kernel launches all children with a fixed grid size of one block. As such, all the requested target cell aggregations of the parent's block are processed loop-wise by 1024 parallel threads of the responsible child kernel. Yet, idle warp threads are prevented because all child threads are assigned to target cells without gaps. The corresponding response times are shown in table 5.6.

Table 5.6.: Speedup of using a fixed grid size of one for children over dynamic grid sizes (*with warp preaggregation)

Query	CDP implementation			CDP implementation*		
	Response time	#PHF	Speedup	Response time	#PHF	Speedup
B1	2,430 ms	7168	0.96	**2,167 ms**	36	0.96
B2	15,603 ms	108	1.13	**14,026 ms**	108	1.18
B3	32,258 ms	36	1.09	**25,298 ms**	48	1.09
M1	285 ms	5120	0.96	**237 ms**	88	1.01
M2	3,685 ms	8	1.17	**3,521 ms**	4	1.28
M3	19,587 ms	4	0.82	**19,257 ms**	4	0.85

We have used the NVIDIA Visual Profiler to examine the time-line of the CDP implementation while query B3 is processed. Both, host-launched and device-launched kernels, and the parent-child relationship can be visually profiled as seen in figure 5.4.

The comparison of the time-lines indicates that child kernels with a fixed grid size benefit more from the GPU's scheduling policy than the child kernels with a dynamic grid size. In contrast to time-line 5.4 a) less time elapses between the execution of child kernels in time-line b), corresponding to a higher launch density. Furthermore, more child blocks are running concurrently. Clearly, using a fixed grid size for the children can accelerate the processing of queries using the CDP implementation.

Nonetheless, the processing of group S queries is slightly slowed down and processing query $M3$ is noticeably slower. We think this is caused by the overheads of looping over the target cells in the child. The method of processing query S has not changed by fixing the child grid size. This can be looked up in the last column of table 5.7. The table shows the number of average child kernels with the number of average child blocks launched by each of the 30 parent blocks per query. Additionally, the overall sum of launched child kernels per query and the number of average blocks per child kernel are described.

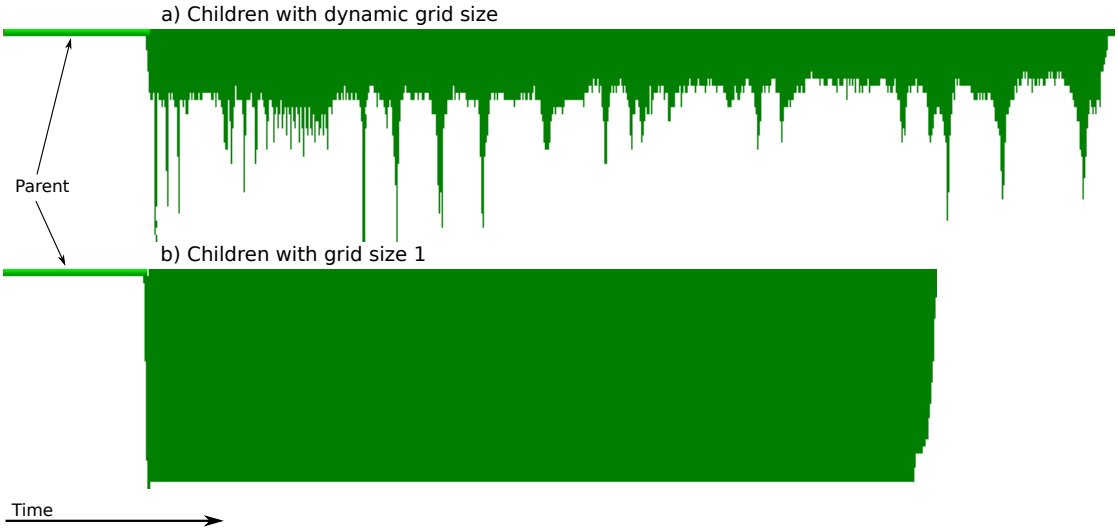


Figure 5.4.: Time-lines of the CDP implementation while processing query B3. Time-line a) features child kernels with dynamic grid sizes, while in time-line b) they are fixed to one block. The x-axis represents the number of concurrently executed kernels (at maximum 30).

The latter also corresponds to the number of loop iterations performed by a child kernel which has a fixed grid size of one block.

Table 5.7.: Statistics related to the launched child kernels per query

Query	Child kernels per parent block	Child blocks per parent block	Sum of child kernels	Blocks / loops per child
B1	12199	12199	365960	1
B2	12199	167160	365960	13.7
B3	12199	140687	365960	11.5
M1	1792	1792	53769	1
M2	270	46404	8094	172
M3	1282	149918	38457	117

According to the response times shown table 5.6 it seems advantageous to use a fixed block size of one for relatively small numbers of child kernels, even if one child block does the work which is otherwise done by 172 blocks. If many child kernels are executed as done for the Biker queries and query M3, the launch configuration favored by the GPU's scheduling policy seems to depend on the amount of blocks per child. If the child kernel is supposed to process a high number of target cells ($117 * 1024$ for query M3), it

is better to use a dynamic grid size over using a grid size of one. Otherwise, for a lower number of target cells, it is better to use the fixed grid size.

The profiling additionally indicates that no child kernels are executed while the parent has not completed its work. This happens because the GPU is fully occupied by the parent grid and no resources are left to execute child kernels. Hence, all child kernel launches are saved to the kernel launch queue until the parent has run. In the next section, we evaluate the use of explicit parent-child synchronization.

5.5.2. Usage of explicit parent-child synchronization

The function `cudaDeviceSynchronize` allows for explicit parent-child synchronization, which causes a parent block to wait for the completion of all child kernel grids it has launched. In our case the synchronization is primarily useful to control the consumption of global memory space while a query is processed.

Recall that the CDP implementation uses additional global memory space to transfer the target counts from a parent block to the child grid. We chose `uint32_t`, a 4 byte fixed-width integer, as data type for the target count of one source cell. Since b blocks of t threads each process the source cells of m pages, a global memory space of $4mbt$ bytes is required for the overall parent-child information transfer. E.g. for query B3 on a Tesla K40c, m corresponds to 12199 pages, b to 30 blocks and t to 768 threads resulting in roughly 1.124 GB of space. These are 9.37% of the overall global memory space available on the GPU (12 GB).

Moreover, the employment of CDP requires global memory space to buffer the configuration and parameters of every launched, but pending child kernel. We know from [NVI14b] that the device runtime reserves enough memory to host a fixed-size pool of 2048 pending kernel launches by default. However, in our CDP implementation the launch queue consists of a maximum of $m * b$ child kernels, a number which can grow much larger than 2048. If this happens, kernel launches are tracked in a lower performance virtualized buffer, leading to a severe execution slowdown. We can prevent this by computing the maximum number of child kernels in advance and extending the reserved memory of the fixed-size pool before processing a query[3]. To find out how much memory is used by the kernel launch pool we have repeated the experiment from

[3]The related configuration option `cudaLimitDevRuntimePendingLaunchCount` can be set by the host program using the `cudaDeviceSetLimit` API. See [NVI14b] for more details.

[WY14] on the Tesla K40c. The reserved memory size is measured by calling the runtime API `cudaMemGetInfo` before and after `cudaDeviceSetLimit`. Figure 5.5 shows the amount of memory reserved for different sizes of the launch pool at a synchronization depth of 2.

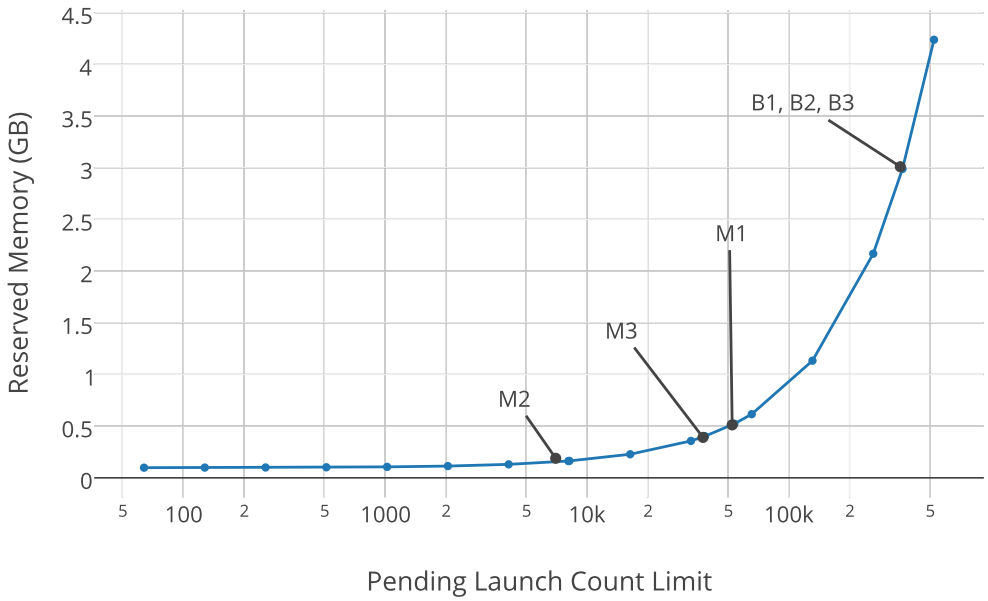


Figure 5.5.: Global memory reserved for kernel launches at synchronization depth 2 using CUDA 6.5

In figure 5.5 we can see that about 3 GB are reserved for the launch pool to process query $B3$. The overall sum of space needed by CDP to process the query is about 4.1 GB. These are more than 34% of the overall global memory space.

To evaluate the performance of a synchronized CDP implementation, we chose to synchronize each parent block after it has launched 2048 children. Consequently, the launch count can be fixed to $b * 2048 + 1$ independent of which query is processed. For $b = 30$ this results in a reserved space of 597 MB.

The memory used for the parent-child information transfer can be reduced to

$$\frac{4mbt}{\lceil \frac{m}{2048} \rceil} \text{bytes.} \tag{5.1}$$

In case of query $B3$ these are 187 MB. Like this, instead of requiring a CDP-specific space of 4.112 GB to process the query, only 784 MB (6.5%) of global memory are used.

Table 5.8 shows the response times of the synchronized CDP implementation. The speedup values have been calculated using the response times from table 5.5.

Table 5.8.: CDP implementation using explicit parent-child synchronization after every 2048 child launches (*with warp preaggregation)

Query	CDP implementation			CDP implementation*		
	Response time	#PHF	Speedup	Response time	#PHF	Speedup
B1	2,060 ms	7168	1.13	**1,945 ms**	36	1.07
B2	17,550 ms	104	1.00	**17,941 ms**	108	0.92
B3	54,219 ms	72	0.64	**42,732 ms**	48	0.64
M1	276 ms	3072	0.99	**242 ms**	88	0.99
M2	4,347 ms	4	0.99	**4,679 ms**	4	0.96
M3	16,067 ms	4	1.00	**18,749 ms**	4	0.87

Surprisingly, the synchronization has a positive influence on the processing of query $B1$. We guess this effect is related to a more optimal scheduling of child kernels in relation to the efficiency of the GPU's L2 cache. However, this was not verified. A rather negative effect of synchronization on the processing performance can be observed for the queries of group L.

6. Conclusion and Future Work

In this thesis, we present and implement an in-GPU-memory MOLAP aggregation algorithm which uses CUDA Dynamic Parallelism. Furthermore, we present and implement a new warp preaggregation method using the CUDA shuffle command. We additionally describe a sequential aggregation algorithm for MOLAP and publish StOAP, which represents the algorithm's implementation. We compare the performance of StOAP, the non-CDP, source-based GPU implementation and the CDP implementation using queries with target areas of different size. Furthermore, we evaluate the impact of using different child kernel grid sizes on the performance of the CDP implementation. Finally, we evaluate the use of explicit parent-child synchronization to minimize the CDP overheads regarding the consumption of global memory.

The experiments show that the single-threaded CPU implementation is outperformed by the GPU implementations by 16 to 218 times. It is further shown that the preaggregation method using CUDA shuffle accelerates query-processing by the non-CDP implementation by up to 49%.

Moreover, we conclude from the tests that queries with medium and big target areas related to a typical OLAP cube are processed faster by an average of 22% compared to the non-CDP implementation. This is the case if both GPU implementations do not use preaggregation. Otherwise, the non-CDP implementation is faster by an average of 42%. However, if strongly varying counts of target cells for adjacent source cells occur, the CDP implementation accelerates the query-processing by 372% if preaggregation is used and by 364% otherwise.

The experiments additionally show that the employment of a fixed child kernel block size is advantageous for queries with a medium-sized target area. It accelerates their processing through the CDP implementation by up to 28%. Furthermore, we find that the memory overheads connected with CDP can be reduced successfully through explicit parent-child synchronization. It even leads to a speedup of up to 13% for queries with a small target area. However, synchronization slows down the processing of queries with medium to big target areas.

After all, the performance of the CDP implementation looks promising and it is highly probable that a method can be found to dynamically decide in which situation the use of one or the other kernel is more advantageous.

A future project could be the implementation of both GPU aggregation approaches in combination. The average count of target cells for the source cells of a block can be used to calculate the variance of target cells per source cell. If the variance is high and the average count is low, the parent kernel would launch the child kernel which writes a single target cell per thread (see section 4.3.3). If the variance is low, the parent block could either launch a child kernel working like the non-CDP kernel (see section 4.2) or perform the non-CDP aggregation by itself. Like this, we could provide the algorithm with the advantages of both, the non-CDP and the CDP approach. Furthermore, if multiple successive pages of a parent block require the use of the same child kernel, multiple kernels might be merged to reduce the number of kernel launches.

An alternative to the above idea is to develop and implement a switch for deciding which algorithm should be used during run-time. The decision mainly depends on the query and the meta-data of the cube, more specifically the hierarchy of the cube's last dimension. This would enable us to get the best performance out of both implementations.

We have found the number of parallel hash functions has a greater impact on the performance of the CDP implementation compared to the non-CDP variant. No matter which one of the source-based approaches is used, the hash function count is a critical parameter for the kernel's performance. A method for predicting the optimal number of parallel hash functions should be researched. The latter depends on many parameters, such as the GPU model, the aggregation kernel, the queries' target area and the cube's meta-data. If suitable metrics for the dependent parameters can be found and after crafting multiple different cubes and test queries, machine learning could be applied to predict the best hash function number for every situation.

Furthermore, it could be investigated why the processing time of StOAP for query $M2$ is greater than the one for $M3$. Eventually, the performance of StOAP could be optimized by developing and implementing a time- and space-efficient data structure for the fact table. The basis of the data structure could be a prefix tree (trie) for binary keys, which are mapped to double-precision numerical values.

In this thesis, we evaluate the performance the GPU implementations using CUDA 6.5. Since new CUDA releases often introduce improvements in terms of performance, it would be interesting to know how well the described GPU implementations perform using CUDA 7.0, which was released during the time of writing. Additionally, the performance using multiple GPUs could be evaluated.

A. CUDA

<table>
<tr><th>Feature Support</th><th colspan="6">Compute Capability</th></tr>
<tr><th>(Unlisted features are supported for all compute capabilities)</th><th>1.1</th><th>1.2</th><th>1.3</th><th>2.x</th><th>3.0</th><th>3.5, 5.x</th></tr>
<tr><td>Atomic functions operating on 32-bit integer values in global memory (Atomic Functions)</td><td colspan="6" rowspan="2">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in global memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 32-bit integer values in shared memory (Atomic Functions)</td><td rowspan="4">No</td><td colspan="5" rowspan="4">Yes</td></tr>
<tr><td>atomicExch() operating on 32-bit floating point values in shared memory (atomicExch())</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in global memory (Atomic Functions)</td></tr>
<tr><td>Warp vote functions (Warp Vote Functions)</td></tr>
<tr><td>Double-precision floating-point numbers</td><td colspan="2">No</td><td colspan="4">Yes</td></tr>
<tr><td>Atomic functions operating on 64-bit integer values in shared memory (Atomic Functions)</td><td colspan="3" rowspan="6">No</td><td colspan="3" rowspan="6">Yes</td></tr>
<tr><td>Atomic addition operating on 32-bit floating point values in global and shared memory (atomicAdd())</td></tr>
<tr><td>__ballot() (Warp Vote Functions)</td></tr>
<tr><td>__threadfence_system() (Memory Fence Functions)</td></tr>
<tr><td>__syncthreads_count(),
__syncthreads_and(),
__syncthreads_or() (Synchronization Functions)</td></tr>
<tr><td>Surface functions (Surface Functions)</td></tr>
<tr><td>3D grid of thread blocks</td><td colspan="4">No</td><td colspan="2">Yes</td></tr>
<tr><td>Unified Memory Programming</td><td colspan="5">No</td><td>Yes</td></tr>
<tr><td>Funnel shift (see reference manual)</td><td colspan="5" rowspan="2">No</td><td>Yes</td></tr>
<tr><td>Dynamic Parallelism</td><td></td></tr>
</table>

Figure A.1.: Feature support per Compute Capability taken from [NVI14b, pp. 181-182]

B. Test environment

All tests were performed on a server running CentOS 6.6 64bit with GNU/Linux kernel version 2.6.32. The test programs were compiled using GCC version 4.4.7. The hardware platform was composed of two Intel Xeon E5-2643 Quad-Core CPUs at a clock speed of 3.30GHz with 256GB of DDR3 SDRAM. A NVIDIA Tesla K40c GPU was used for testing. The server's CPU frequency scaling was set to performance mode. GPU persistence mode was enabled during all tests. This ensures that the GPU driver stays loaded and prevents the GPU from reverting back to idle clocks. The GPU clock has been set to a fixed speed of 875Mhz. StOAP has been assigned to one processor core to prevent context switching during the tests. The response times for StOAP were generated by a script using StOAP's named pipes as query input and result output interfaces. Another script using GNU `Wget` was used to send queries to the GPU-accelerated OLAP server. All query results computed by StOAP and by the GPU kernels have been compared at least once with the results of Jedox' multi-threaded OLAP server using a Perl script to verify the precision of the results.

C. Optimal page size

The page size p is determined by the formula

$$p = b * t, \tag{C.1}$$

where b is the number of blocks per grid and t is the number of threads per block. Those two parameters are used for launching the aggregation kernel on a 1-dimensional grid and can be freely chosen with certain limits[1]. However for efficiency, they should be optimized (e.g. by empirical tests) as described in [NVI14a, pp. 43-48].

The optimal size of t depends on the kernel code and the used GPU. Executing too many threads per block can result in register spilling, causing the threads to make use of high-latency GPU memory, whereas executing too few threads per block might result in a low GPU occupancy. Unpublished empirical tests by Lauer and Strohm suggest that $t = 768$ results in the best average performance of the aggregation kernel on a Tesla K20c.

Warps can be in a paused or stalled state. The grid size b must be chosen in a way to ensure there are always enough warps in the work queue for the warp schedulers to choose from. If no other warps are executed the GPU hardware is not fully occupied. To maximize the total number of active warps, Lauer and Strohm decided to add two blocks per available SMX unit to the grid.

Hence, the grid size b is defined by

$$b = 2 * P, \tag{C.2}$$

where P is the number of SMX units on the GPU. Substituting b in C.1 with the formula from C.2 leads to

$$p = 2 * P * t. \tag{C.3}$$

Thus, the optimal page size on a Tesla K40c with 15 SMX units is $p = 23040$.

[1] t can be a number between 0 and 1024 on GPUs supporting CC $\geq$ 2.0.

D. Upper bound search implementation

```
int32_t first = 0;
int32_t count = GPU_THREAD_COUNT;  // corresponds to block size b
int32_t it = first;
int32_t step = 0;
while (count > 0) {
  it = first;
  step = count / 2;
  it += step;
  if (!(targetId < s_targetCount[it])) {  // prefix-sum array of the counts
    first = ++it;
    count -= step+1;
  }
  else {
    count = step;
  }
}
```

Listing 4: C code implementation ported from a code example for `std::upper_bound` found at `http://www.cplusplus.com/reference/algorithm/upper_bound/`.

E. Testing different parallel hash function counts

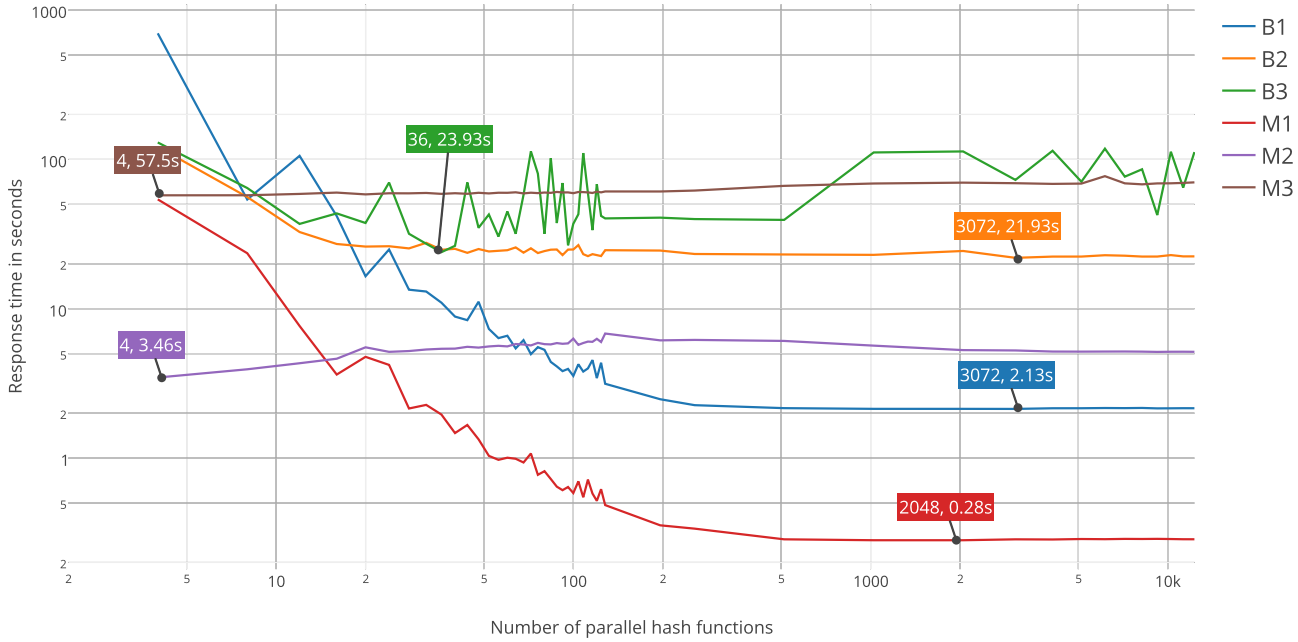


Figure E.1.: This is the plot for the non-CDP kernel without warp preaggregation. Response times of queries $B1$-$B3$ and $M1$-$M3$ are plotted in relation to the number of hash functions. The highlighted spots describe the amount of hash functions and the fastest response time (separated by a comma).

Danksagung

An dieser Stelle möchte ich mich besonders bei meinem Betreuer und Arbeitskollegen Steffen Wittmer, aber auch bei Peter Strohm und Alexander Haberstroh von der Firma Jedox AG bedanken. Sie haben sich mit mir zusammengesetzt um Themenvorschläge für meine Bachelorarbeit zu finden. Außerdem standen sie mir beim Einstieg in die umfangreichen Themen MOLAP und CUDA mit Rat zur Seite.

Weiterhin möchte ich mich bei der Firma Jedox AG bedanken, die mir über die Zeit des Studiums einen Arbeitsplatz bot und mir auch für die Dauer der Bearbeitungszeit einen Arbeitsplatz und die nötige Hardware zum Testen zur Verfügung stellte.

Mein Dank geht auch an die Personen, die sich Zeit fürs Korrekturlesen genommen haben: Steffen, Holger, Alex, und Guido. Thank you very much!

Am allermeisten danke ich aber meiner Familie und meiner Freundin Selly. Sie alle haben mir in der Zeit des Studiums den Rücken gestärkt und mich immer wieder motiviert.

Celia, meine Schwester, und ihr Freund Patrick wurden am 22. März Eltern des kleinen Levi. Vielen Dank für den Kleinen, der unsere Familie sehr bereichert und bei mir viel Motivation hervorruft. Ich liebe euch!

Bibliography

[AH13] Sk. Masudul Ahsan and K. Hasan. “An Efficient Encoding Scheme to Handle the Address Space Overflow for Large Multidimensional Arrays”. In: *Journal of Computers* 8.5 (2013). DOI: 10.4304/jcp.8.5.1136-1144. URL: http://ojs.academypublisher.com/index.php/jcp/article/view/jcp080511361144 (cit. on p. 16).

[Böh+11] Matthias Böhm, Benjamin Schlegel, Peter Benjamin Volk, Ulrike Fischer, Dirk Habich, and Wolfgang Lehner. “Efficient In-Memory Indexing with Generalized Prefix Trees”. In: *Datenbanksysteme für Business, Technologie und Web (BTW), 14. Fachtagung des GI-Fachbereichs "Datenbanken und Informationssysteme" (DBIS), 2.-4.3.2011 in Kaiserslautern, Germany*. 2011, pp. 227–246 (cit. on p. 16).

[Bre+14] Sebastian Bress, Max Heimel, Norbert Siegmund, Ladjel Bellatreche, and Gunter Saake. “GPU-accelerated Database Systems: Survey and Open Challenges”. In: *Transactions on Large-Scale Data and Knowledge-Centered Systems (TLDKS)* 8800 (2014). To appear. (cit. on p. 4).

[CCS93] Edgar Frank Codd, S.B. Codd, and C.T. Salley. “Providing OLAP to User-Analysts: An IT Mandate”. In: (1993). URL: http://www.minet.uni-jena.de/dbis/lehre/ss2005/sem_dwh/lit/Cod93.pdf (cit. on p. 6).

[Eic13] Susanne Eichel. “Parallele Berechnung großer spärlich besetzter aggregierter Bereiche mit Hilfe von Grafikprozessoren”. MA thesis. Albert-Ludwigs-Universität Freiburg im Breisgau, Sept. 2013 (cit. on pp. 3, 16, 18, 19, 31, 32).

[Fly72] M. Flynn. “Some Computer Organizations and Their Effectiveness”. In: *Computers, IEEE Transactions on* C-21.9 (Sept. 1972), pp. 948–960. DOI: 10.1109/TC.1972.5009071 (cit. on p. 12).

[HSO07] M. Harris, S. Sengupta, and J.D. Owens. “Parallel prefix sum (scan) with CUDA”. In: *GPU Gems* 3.39 (2007), pp. 851–876. URL: http://developer.download.nvidia.com/compute/cuda/2_2/sdk/website/projects/scan/doc/scan.pdf (cit. on p. 22).

[Kac11] Krzysztof Kaczmarski. “Comparing GPU and CPU in OLAP Cubes Creation”. In: *SOFSEM 2011: Theory and Practice of Computer Science* 6543/2011 (2011), pp. 308–319. DOI: 10.1007/978-3-642-18381-2_26 (cit. on p. 2).

[Lau+10] Tobias Lauer, Amitava Datta, Zurab Khadikov, and Christoffer Anselm. “Exploring Graphics Processing Units as Parallel Coprocessors for Online Aggregation”. In: *Proceedings of the ACM 13th International Workshop on Data Warehousing and OLAP*. DOLAP ’10. 2010, pp. 77–84. DOI: 10.1145/1871940.1871958 (cit. on pp. 3, 9, 17).

[Mic12] Paulius Micikevicius. *GPU Performance Analysis and Optimization*. Presentation at the GPU Technology Conference. June 2012. URL: http://on-demand.gputechconf.com/gtc/2012/presentations/S0514-GTC2012-GPU-Performance-Analysis.pdf (cit. on p. 21).

[NVI13] NVIDIA. *NVIDIA’s Next Generation CUDA Compute Architecture: Kepler GK110*. Whitepaper. Jan. 2013. URL: http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf (cit. on p. 12).

[NVI14a] NVIDIA. *CUDA C Best Practices Guide*. Version 6.5. NVIDIA Corporation, Aug. 2014. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf (cit. on p. 47).

[NVI14b] NVIDIA. *CUDA C Programming Guide*. PG-02829-001. Version 6.5. NVIDIA Corporation, Aug. 2014. URL: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf (cit. on pp. 10, 11, 18, 19, 22, 25, 27, 40, 45).

[SHP12] Craig Silverstein, Donovan Hide, and Geoff Pike. *Google SparseHash: An extremely memory-efficient hash_map implementation*. Feb. 2012. URL: https://code.google.com/p/sparsehash/ (cit. on p. 16).

[Wik15] Wikipedia. *Moore’s law — Wikipedia, The Free Encyclopedia*. [Online; accessed 20-February-2015]. Feb. 2015. URL: http://en.wikipedia.org/w/index.php?title=Moore’s_law&oldid=648043129 (cit. on p. 1).

[WL13] Steffen Wittmer and Tobias Lauer. *Computation of Large Sparse Aggregated Areas for Analytic Database Queries*. Presentation at the GPU Technology Conference 2013. 2013. URL: http://on-demand.gputechconf.com/gtc/2013/presentations/S3088-Computation-Large-Sparse-Aggregated-Areas.pdf (cit. on pp. 3, 16, 18).

[WY14] Jin Wang and Sudhakar Yalamanchili. “Characterization and Analysis of Dynamic Parallelism in Unstructured GPU Applications”. In: *2014 IEEE International Symposium on Workload Characterization*. Oct. 2014 (cit. on pp. 4, 13, 21, 41).

[ZZN14] Tilmann Zäschke, Christoph Zimmerli, and Moira C. Norrie. “The PH-tree: A Space-efficient Storage Structure and Multi-dimensional Index”. In: *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’14. Snowbird, Utah, USA: ACM, 2014, pp. 397–408. DOI: 10.1145/2588555.2588564. URL: http://doi.acm.org/10.1145/2588555.2588564 (cit. on p. 16).