Christoph Weyerhaeuser, Tobias Mindnich, Franz Faerber, Wolfgang Lehner

Exploiting Graphic Card Processor Technology to Accelerate Data Mining Queries in SAP NetWeaver BIA

Erstveröffentlichung in / First published in:

IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 15-19.12.2008. IEEE, S. 506-515. ISBN 978-0-7695-3503-6

DOI: https://doi.org/10.1109/ICDMW.2008.61

Exploiting Graphic Card Processor Technology to Accelerate Data Mining Queries in SAP NetWeaver BIA

Christoph Weyerhaeuser, Tobias Mindnich,
Franz Faerber
SAP AG, PTU NetWeaver AS TREX
Dietmar-Hopp-Allee 16
69190 Walldorf, Germany
{christoph.weyerhaeuser, tobias.mindnich,
franz.faerber}@sap.com

Wolfgang Lehner
Dresden University of Technology
Database Technology Research Group
01062 Dresden, Germany
wolfgang.lehner@tu-dresden.de

Abstract

Within Business Intelligence contexts, the importance of data mining algorithms is continuously increasing, particularly from the perspective of applications and users that demand novel algorithms on the one hand and an efficient implementation exploiting novel system architectures on the other hand. Within this paper, we focus on the latter issue and report our experience with the exploitation of graphic card processor technology within the SAP NetWeaver Business Intelligence Accelerator (BIA). The BIA represents a highly distributed analytical engine that supports OLAP and data mining processing primitives. The system organizes data entities in column-wise fashion and its operation is completely main-memory-based. Since case studies have shown that classic data mining queries spend a large portion of their runtime on scanning and filtering the data as a necessary prerequisite to the actual mining step, our main goal was to speed up this expensive scanning and filtering process. In a first step, the paper outlines the basic data mining processing techniques within SAP NetWeaver BIA and illustrates the implementation of scans and filters. In a second step, we give insight into the main features of a hybrid system architecture design exploiting graphic card processor technology. Finally, we sketch the implementation and give details of our vast evaluations.

1 Introduction

Data mining is gaining more and more significance from two perspectives. On the one hand, the acquisition and storage of the digital footprints of individual entities (products, human beings, Web sessions) is now technically feasible and economically affordable. On the other hand, data mining mechanisms have gone mainstream, i.e., algorithms are more sophisticated and yet easier to use for non-expert users. Both aspects have a significant impact on the way data is processed. Within SAP NetWeaver BI, common data mining methods, such as association rule mining, time series, outlier analyses, and some special statistical algorithms, are seamlessly integrated into the backend. The implementation heavily exploits the SAP NetWeaver BI Accelerator (BIA) in order to provide efficient data mining capabilities to achieve short query times.

The SAP NetWeaver Business Intelligence (BI) Accelerator ([10]) is used to speed up OLAP and data mining queries usually stored in relational databases. Compared with most previous approaches, this tool offers an immense improvement in speed and flexibility of access to the data. This improvement is significant in business contexts where existing approaches incur quantifiable costs. By leveraging the falling prices of hardware relative to other factors, the new tool offers business benefits that more than balance its procurement cost.

Figure 1 shows the overall setup of a BIA system. In the presence of the acceleration tool, business data is replicated to the BIA and stored as highly optimized index structures within main memory. Since BIA follows the shared-nothing paradigm, a BIA landscape is able to scale out to a fairly large number of individual nodes. Analytical queries targeting data indexed within the BIA are transparently rerouted to the BIA through the BI analytical engine. The accelerator responds to queries and performs joins, aggregates, and data mining operations on the fly. The main principle is to keep data in one single place and to efficiently perform classic OLAP operations as well as highly specialized data mining tasks using the same data structures and access operations.

The BIA system is originally based on search engine technology and offers comparable performance for both fre-

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61

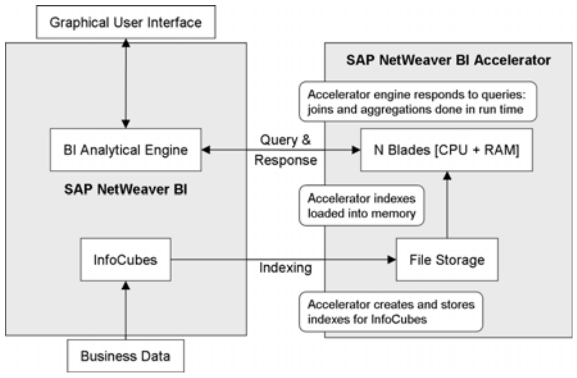

Graphical User Interface
SAP NetWeaver BI Accelerator
Accelerator engine responds to queries: joins and aggregations done in run time
BI Analytical Engine
Query & Response
N Blades [CPU + RAM]
SAP NetWeaver BI
Accelerator indexes loaded into memory
InfoCubes
Indexing
File Storage
Accelerator creates and stores indexes for InfoCubes
Business Data

Figure 1. Query Processing in Memory

quently and infrequently submitted queries. This enables companies to relax previous technically motivated restrictions on data access; since BIA is purely based on main memory technology, there is no need to explicitly build materialized views or to pre-compute certain aggregates. In practical terms, the result is that with the help of the accelerator, users can select and display aggregated data, slice it and dice it, drill down anywhere for details, and expect exact responses within seconds, however unusual their queries may be, even over data cubes containing billions of records and occupying terabyte volumes of raw data.

The main architectural features of the BI accelerator are as follows:

- Distributed and parallel processing for scalability and robustness
- In-memory processing to eliminate the runtime cost of disk accesses
- Optimized data structures for reads and optimized structures for writes
- Data compression coding to minimize memory usage and I/O overhead.

Data Mining Using BIA

Over the last years, SAP's BIA system has been functionally enriched with regard to data mining capabilities. As of now, the system supports classic data mining operations like time series analyses, the detection of association rules or the computation of correlations within large item sets. Although data mining algorithms can be seamlessly integrated into the BIA engine, we experienced significant challenges arising from the specific patterns of data mining operations. For example, consider a data mining query to predict 20 values of the overall sales revenue on a weekly basis in a real SAP customer sales data cube with 1,093,470,000 fact entries (80 parts distributed over 10 blades with 80 cores in total). The variation in the range of the query shows an interesting fact: scanning the (in-memory) data and picking the qualifying entries for the succeeding time series analysis is demanding for a large portion of the overall query runtime. Query 1 considers only sales satisfying the predicate `country=ALABAMA` with a selectivity of 1.54%. Surprisingly, out of 150ms in total to complete the query, the system spends 40ms (and therefore almost one third of the overall query runtime) to find the appropriate sales entries. An increase in the selectivity, i.e., an increase in the number of satisfying entries, shows an even higher fraction of data scanning relative to the overall query runtime (50% of the overall runtime is spent on evaluating the predicate for the individual data cube entries for the predicate `country=CALIFORNIA` with selectivity of 23.07%). Obviously, there is a significant need to improve the scan times to lower the overall query runtime. As can be seen, if we perform a time series analysis for all sold articles (performed in 484ms for 1billion rows), the data access ratio is reduced to only 11%.

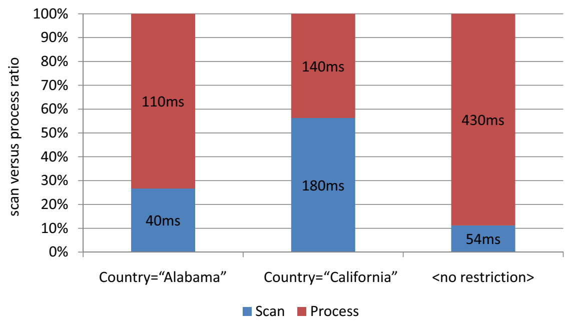


Figure 2. Example: Time Series Analysis

In order to tackle this issue of expensive scans and filters, we report on the implementation of scan operators for the main-memory-based BI acceleration engine from two perspectives. On the one hand, we outline the implementation of scan methods (point-wise access and scan mode). On the other hand, we describe our experience with an implementation of the scan exploiting graphic card processor technology and we discuss different configuration issues.

Related Work

Using graphic card processors to speed up computing-intensive applications has been around for quite some time (e.g., [1] for scan optimization) but has received special attention in the database ([7, 3]) and data mining ([4, 5]) community.

From a system-architectural perspective, recent work addresses the efficient processing of join operators (for example, [8, 9]) and–similar to our work–the optimization of the scan operation. For example, [6, 11] implemented segmented scan primitives and work-efficient scans also using Nvidia CUDA programming API. Another interesting work related to our approach is presented in [2], where the input vector is decomposed into small chunks (via a 2D matrix) fitting into the registers of their Cray Y-MP architecture. We use such a technique for our output data. However, all these efforts do not specifically focus on the decomposition and extraction of non-word-aligned values, which is the algorithmic center of our work.

Structure of the Paper

The rest of this paper is organized as follows. The next section outlines the SAP NetWeaver BI Accelerator in sufficient detail for the reader to understand the challenges for efficient data mining operations. Sections 3 and 4 represent the core of the paper and describe different methods to perform scans over indexed data sets. In a first step, we show the point-wise data access; in a second step, we focus on the scanning process to access compressed data. After the presentation of the conceptual work, we describe the details of our implementation on top of Nvidia GeForce 8800 GTX and present the results of our vast experiments. The paper closes with a conclusion summarizing our experiences with non-standard hardware to speed up data mining queries.

2 Basic Architecture

This section introduces the concept of the SAP NetWeaver BIA. First of all, we outline the general architecture and then we dive into the physical storage scheme used by BIA.

Scalable Multiserver Architecture

The BI Accelerator runs on commodity blade servers that can be procured and installed as required to handle increasing volumes. For example, a moderately large instance of the accelerator may run on $10 - 16$ blades in typical customer scenarios[1], each with two Intel Xeon quad-core processors and 16GBytes of main memory; thus, the accelerator typically has $160 - 256$GByte of RAM with $80 - 128$ CPU cores to handle user requests. The accelerator index structures are compressed, usually by a factor of well over ten relative to regular database tables from which they are generated (see the following discussion on compression), so the available main memory is enough to handle most of the BI scenarios running in commercial SAP NetWeaver BI environments.

In each BIA landscape, a name server maintains a list of active services, switches to backups where necessary, and balances load over active services (figure 3). Customers will implement the BI Accelerator on preconfigured hardware that can be plugged into their existing SAP landscape. In an adaptive landscape, accelerator instances will be replicated as required, by cloning services, and they will form groups with master and backup servers and additional cohort servers. The master servers will handle both indexing and search load; the cohorts will handle only query load. Such groups can be optimized for both high availability and good load balancing, with the overall goal being the requirement of zero administration.

[1] A lab installation with 140 nodes showed linear scaling behavior.

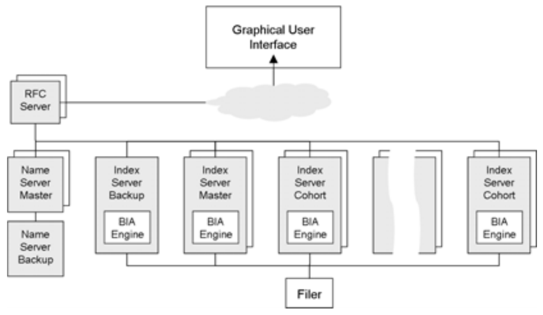


Figure 3. Multiserver Architecture for Scalability

The accelerator engine usually partitions large tables horizontally for parallel processing on multiple machines in distributed landscapes (figure 4). This enables it to handle very large data volumes and yet stay within the limits of installed memory. The engine splits such large volumes over multiple hosts, by a round-robin procedure to build up parts of equal size, so that they can be processed in parallel. A logical index server distributes join operations over partial indexes and merges the partial results returned from them. This scalability enables the accelerator to run on commodity computing infrastructures, such as blade servers over which load can be redistributed dynamically.

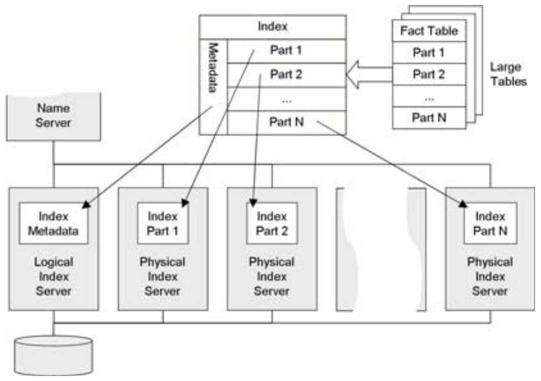


Figure 4. Horizontal Partitioning of Indexes

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61

Main Memory Data Structures

In a traditional database, all the data in a table is organized in complete rows. The BIA engine decomposes table data vertically into columns that are stored separately, thereby using memory space more efficiently than with row-based storage, since the engine needs to search only in relevant attributes. Column-wise storage has proven to be appropriate for data analytics scenarios, where most users want to see only a selection of data and attributes. The engine can also sort the columns individually to bring specific entries to the top.

Data for the accelerator is compressed using integer coding and dictionary lookup. Integers represent the text or other values in table cells, and the dictionaries are used to replace integers by their values during post-processing. In particular, each record in a table has a RowID, and each value of a characteristic or an attribute in a record has a ValueID (figure 5). Dictionary-based compression schemes greatly reduce the average volumes of processed and cached data, which allows more efficient numerical processing and smart caching strategies. Altogether, data volumes and flows are reduced in typical customer scenarios by an average factor of ten. The overall goal of this reduction is to improve the utilization of memory space and to reduce I/O within the accelerator. These techniques enable the accelerator to perform fast read and search operations on mass data and yet remain within the constraints imposed by installed memory.

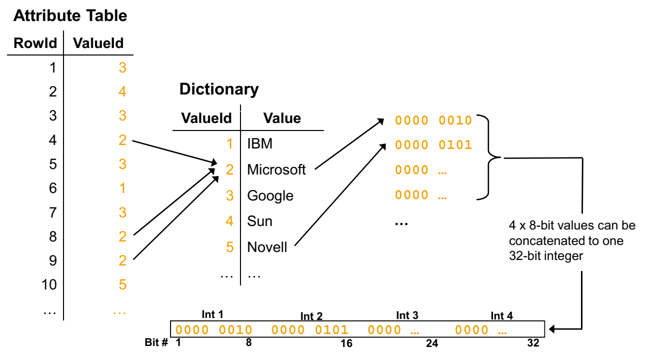


Figure 5. Data Compression Using Integers

To handle updates to the accelerator index structures, the accelerator uses a delta index mechanism that stores changes to an index structure in a delta index, which resides next to the main index in memory. The delta index structure is optimized for fast writes and it is small enough for fast searches. The engine performs searches within an index structure in parallel over both the main index and the delta index and then merges the results. From time to time, to prevent the delta index from growing too large, it is merged with the main index. The merge is performed as a background job that does not block searches or inserts, so response times are not degraded.

3 Data Access in BIA

As shown in our introduction, scanning data and retrieving items satisfying a given filter criterion are the most performance-critical operations. This section focuses on the basic mechanisms of accessing data for point-wise access and for scan-like operations.

Characteristics and Requirements

Since the BIA engine works purely within main memory, storage efficiency is one of the greatest challenges. Within the scope of this paper, this characteristic poses two implications:

- In contrast to classic database systems, the BIA engine does not have any secondary indexes. This implies that the evaluation of any filter expressions results in a full main memory scan of the associated column index.
- Compression schemes are not limited to byte boundaries. The system uses only as many bits per value as necessary to code the number of distinct values.

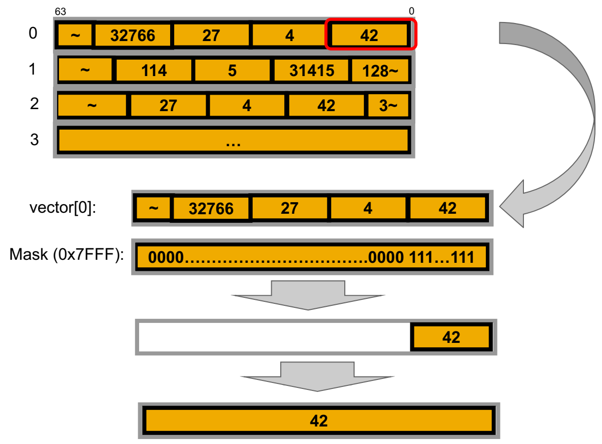
With regard to the second issue, the system takes advantage of the acceleration model: the initial number of distinct values can be retrieved from the primary database; if that number increases dramatically, the system is able to react by storing new data in the temporary index and by re-indexing the affected column with a new coding scheme after the re-index. The first issue mentioned above implies the need for highly efficient data access operations in order to extract values at a given position or to return all values satisfying a given predicate. The main challenge of these data access operations is therefore to extract values out of a compressed storage format–this will be the focus of the remainder of this paper.

Memory Layout

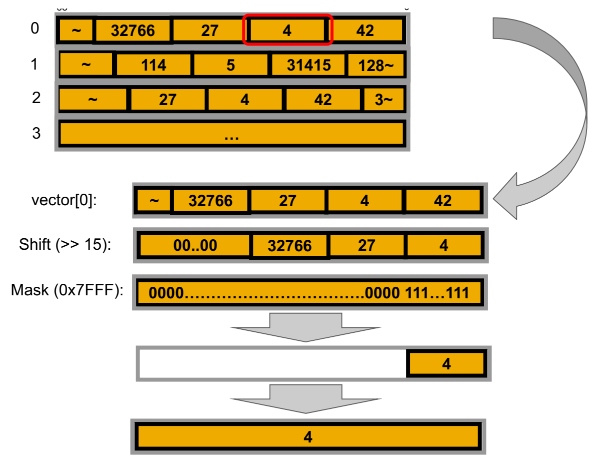
The BIA system supports coding schemes and extraction operations for all bit lengths between 1 and 32 to reflect indexes in the global dictionary. This implies the maximum number of 2^{32} distinct values to be represented within a single column. In real scenarios, most columns can be coded with 5 to 9 bits. For the scope of the following discussion, we refer to the 15-bit case, which is not word-aligned and therefore shows all possible access cases. All other cases either use the same algorithm or they are simpler because of byte alignment (e.g., 8 bit, 16 bit, 24 bit, and 32 bit).

Figure 6 shows the sample storage layout of individual values. As can be seen, the first four values are within the

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61



(a) position 1



(b) position 2

Figure 7. Extracting Values at Pos 1 and 2

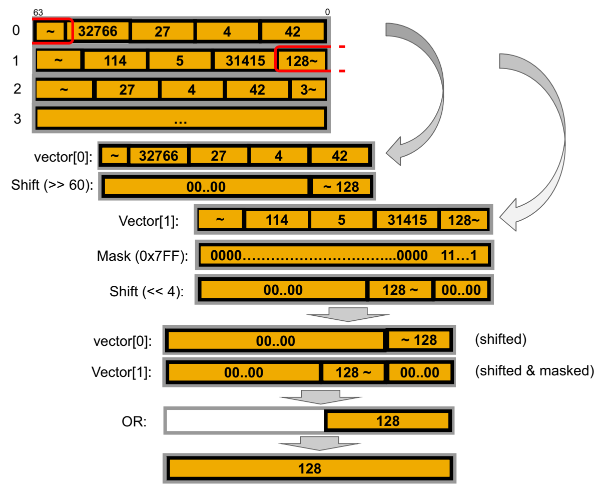


Figure 8. Extracting Value at Position 5

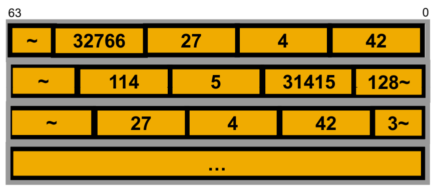


Figure 6. Example Setup for Scan Operations

first 64-bit vector. The 5th value, however, spans two vectors and has to be combined during the extraction process.

Point-Wise Data Access

The first way to access data in main memory retrieves the value stored at a given position within the index. The input consists of 64-bit vectors; the output is the value represented as a 32-bit integer used as an index into the dictionary.

For our running example of the 15-bit case, we can identify three different situations. In the first case, the position is aligned with a $15 * 64$-bit block. For example, the 1st value can be directly retrieved by masking out the unused bit positions (mask vector[0] with 0x7FFF in the 15-bit scenario) and by inflating the 15-bit value to a 32-bit length. Figure 7a shows this situation.

In the second case, the positions are still within a 64-bit word. As can be seen in figure 7b, the 2nd value has to be shifted by 15 bits, followed by the masking and inflation process equivalent to the first case (shift vector[0] by 15 and mask with 0x7FFF). Obviously, in order to extract the 3rd and the 4th values, the 64-bit vector has to be shifted by 30 or 45.

The most interesting case retrieves values at positions spanning two vectors. In our running example, the 5th value is comprised of 4 bits from the first vector and 11 bits coming from the second vector (figure 8). In a first phase, the first vector is shifted and masked in order to retrieve the first fragment of the final value (*shift vector[0] by 60* in the 15-bit example). Analogous to the first case, the second vector is masked in order to retrieve the second fragment of the final values (*mask vector[1] with 0x7FF*). After shifting the second vector by the number of bits coming from the first vector (*shift by 4*), both intermediate results are combined. An inflation step results in the final 32-bit integer value.

Depending on the situation of the required position, one of the three cases is applied in order to retrieve the values at a given position.

Data Access Using Scan

As already mentioned, scanning the data and returning the positions of values satisfying a given value range ((range-

From <= value < rangeTo)) are the most critical access patterns. The implementation of the scan obviously exploits the extraction rules illustrated above. A loop over all positions and a call of the appropriate extraction method for the current position have shown a significant performance degradation. We therefore introduced a special implementation with unrolled loops and pre-computed bit shifts and masks for every possible combination. Working on a $15 * 64$-bit block level, all matching entries are indicated by setting the appropriate bits in a 64-bit result vector.

Unrolling all loops and pre-computing all shift and mask operations show a significant performance advantage. Table 1 states performance figures for different data sizes searching for a single value with a selectivity of $\frac{1}{255}$. The factor of the performance difference for the 8-bit and the 15-bit case turns out to be around $3.5 - 4$, which is indeed significant.

Summary

When working completely within main memory, the need to utilize memory to the best possible extent is obvious. Catalog compression requires only the storage of index values and is therefore the building block for more sophisticated compression schemes. Since every column uses only as many bits as required to represent the number of distinct values, the extraction scheme requires significant attention. While this section has shown the basic extraction process, we can now concentrate on the implementation details of this extraction technique using graphic card processors.

4 Scan Using Graphics Card Processor

Modern graphic cards earmark the beginning of the *desktop-parallel computing age* by representing the first widespread parallel architecture on commodity hardware. The reasonable price in combination with performance optimized for arithmetic computation make it extremely interesting for high-speed data mining applications. As a first step to exploit GPU technology for OLAP and data mining queries, we investigated the cost/benefit ratio of using GPUs within the BIA by porting the scan operation introduced in the preceding section.

4.1 Architectural Approach

Within our experiments, we used a Nvidia GeForce 8800 GTX graphics board with the G80 GPU (1350 MHz). The G80 consists of 8 Thread Processing Clusters (SIMD Groups) with 16 ALUs, each providing the opportunity to execute $16 * 8 = 128$ threads in a truly parallel way. The graphic card was equipped with 768MByte GDDR3 RAM (memory bus width: 384bit). Communication with the regular main memory was performed using the PCIe x16 interface, resulting in a theoretic bandwidth of 4GByte/s (half-duplex). Since moving data from main memory to the local memory of the graphic card is an additional cost factor, we measured the real communication bandwidth in a first set of experiments. Table 2 shows the transfer cost for 512MByte. Compared to the $62,5$ms in theory, we experienced communication that was slower by a factor of $4 - 6$ depending on the transfer mode (PINNED does prohibit the memory from being swapped to disk, while PAGED does not have any restriction on the swapping mechanism). As can be seen, the PINNED mode reaches only 30% of the theoretic bandwidth (and is roughly 30% faster than the PAGED mode). Nevertheless, our experiments showed a reasonable communication bandwidth of $1.5-2.5$GByte/s (CPU RAM $\rightarrow$ GPU RAM) and $1.25 - 2$GByte/s (GPU RAM $\rightarrow$ CPU RAM).

Programming Model

Nvidia provides a programming framework to abstract from implementation details and to release the power of the SIMD architecture. We therefore also used the Nvidia Compute Unified Device Architecture (CUDA) consisting of a CUDA toolkit, CUDA SDK, and CUDA runtime driver. A CUDA application consists of a host-part (C/C++) and the device code usually called the kernel.

In order to exploit the SIMD flavor of the G80, the application has to exploit the extremely light-weight thread model. Up to 512 threads form a *thread block*; one or more thread blocks are executed sequentially on the same multiprocessor. All threads of one block share the 8192 registers of one multiprocessor and are scheduled by the GPU Thread Manager to fully occupy the hardware.

Up to $65,535$ thread blocks can be combined to a *thread grid*. Thread blocks and thread grids are organized in a 3-dimensional structure used as an addressing scheme for the instantiation process of a CUDA kernel. This modular structure and 3-dimensional addressing scheme enables the application programmer to deploy the system with a well-designed parallel program layout. In total, the G80 provides 32 truly parallel threads in 4 clock cycles.

Scan Operation

As illustrated in the preceding section, a logical processing unit (data block) for a k-bit coding scheme consists of $k*32$ bit[2]. For our running example with $k = 15$ (figure 6), a data block consists of 480 bit. The obvious choice is to assign one data block to one kernel in order to perform the decoding process as described before. The individual kernels are grouped to blocks; the GPU thread manager schedules the

[2]As of now, Nvidia CUDA is only able to work with 32-bit data types on devices of computation capability 1.0

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61

	lines	**unrolled**	**loop**	**factor**
8bit	$1,000,000$	8.16ms	27.93ms	3.42
8bit	$100,000$	0.815ms	2.799ms	3.43
8bit	$10,000$	0.075ms	0.274ms	3.65
15bit	$1,000,000$	7.6ms	29.7ms	3.91
15bit	$100,000$	0.756ms	3.071ms	4.06
15bit	$10,000$	0.077ms	0.277ms	3.59

Table 1. Runtime Comparison: Loop versus Unrolled Scan Operation

direction	transfer mode	transfer time	resulting bandwidth
–	THEORY (8b/10b coded)	62.5 ms	4GByte/s
Host RAM → GC RAM	PINNED	196.3 ms	2.54GByte/s
Host RAM → GC RAM	PAGED	290.7 ms	1.71GByte/s
GC RAM → Host RAM	PINNED	246.8 ms	2.02GByte/s
GC RAM → Host RAM	PAGED	384.3 ms	1.3GByte/s

Table 2. Data Transfer Cost (512MByte, Half-Duplex)

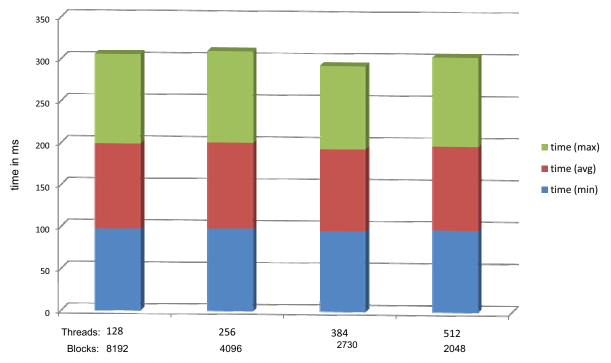


Figure 9. Number of Threads per Block

existing blocks and kernel programs. In order to gain maximum performance, we implemented separate code blocks for every bit combination.

The number of threads heavily depends on the kernel types: since kernels for non-aligned encoding schemes use multiple registers for their data manipulation, the maximum number of 512 concurrently running threads cannot be deployed. This deficit can be compensated by a variation of the number of threads per block. For the 15-bit case, experiments have shown an optional number of 192 threads per block. In contrast, for an 8-bit case, the maximum number of 512 threads per block can be generated and obviously showed the best performance.

Table 3 shows a comparison of runtimes for the 8-bit and the 15-bit scenario to learn the overhead of non-aligned kernel types. In order to retrieve the same number of values (and therefore perform the same number of value extraction operations), the memory size varies for the two situations. The selectivity of the point search was $\frac{1}{255}$ with uniformly distributed data sets.

As can be seen, the query runtimes increase proportionally with the data volume. Moreover, the 15-bit case requires between $19\% - 56\%$ more time by processing 187.5% data volume of the 8-bit case.

Scan Configuration

The system configuration opens up two directions of parameterization of the scan operation. On the one hand: the number of threads per block which alters the number of usable registers per thread; one the other hand: the size of data blocks (k*32 bit with k being the coding parameter) per thread. In the following, we will show our experience with regard to these parameters.

Figure 9 shows the query runtimes for an 8-bit coding scheme with varying number of threads per block for 512MByte raw data. Surprisingly, the resulting query runtimes comparing the extracted values with a single search value do not exhibit any correlation with the thread/block configuration.

The second parameter, the size of data blocks per thread, however, does show a significant impact on query runtimes. Figure 10 illustrates the behavior of the *thread factor*, determining the number of data blocks per thread. Within our experiments, we measured the runtimes with thread factor values 1, 8, 16, 32, 64, and 128 for different raw data sizes. As can be seen, the query runtimes get worse with increasing thread factor, i.e., the more threads (with smaller data sets), the better the query runtime. This behavior is surprising and can only be explained by the extremely light-weight thread model. A similar pattern was also experienced with the 15-bit case.

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61

	raw data size		query runtime	
no. of extracted values	**8-bit case**	**15-bit case**	**8-bit case**	**15-bit case**
$67,108,864$	64 MB	120 MB	17.16	20.53
$134,217,728$	128 MB	240 MB	31.77	39.89
$268,435,456$	256 MB	480 MB	50.44	78.95

Table 3. Query Runtime for 8- and 15-bit Case

	G80	**Xeon**
per input integer	˜0.92	˜14.0
per output integer	˜0.23	˜3.5
processor speed	1.35 GHz	2.67 GHz
factor	1.0	15.21

Table 5. Number of Processor Cycles

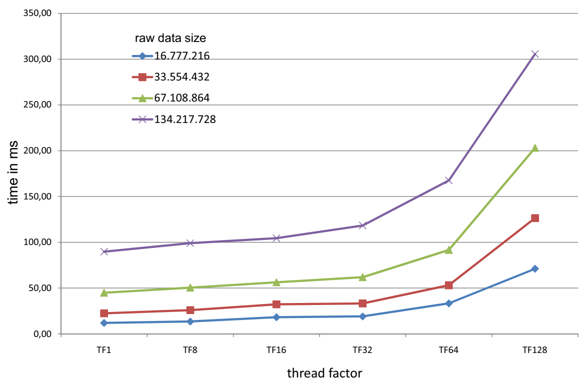


Figure 10. Number of Data Blocks per Thread

Cost per Filter Criterion

In another setup, we evaluated the cost of an additional filter criterion. We tackled this question from two perspectives. First, we were interested in the cost of adding another value as a filter criterion, i.e., the overhead to test against another value. In a second step, we looked into the overhead of different hit ratios. We therefore ran multiple tests with different hit/value ratios. As can be seen in figure 11 for the 8-bit case, a first series considers zero, one, and five hits for five different filter values; a second series distinguishes the situation of a hit/no hit for a point query (one single test value as filter criterion).

Overhead per Filter Criterion In order to compute the overhead of an additional filter criterion, we refer to the difference of the $\frac{0}{5}$- and $\frac{0}{1}$-query runtimes resulting in 1.5msec for $16,777,216$ entries. Therefore, an additional filter criterion increases the overall query runtime by 9% for the 8-bit case. For our running example of the 15-bit case, we experienced a similar overhead, which, however, amounts only to a relative increase of 6% due to the higher query costs in general.

Overhead per Hit As can be easily figured out in figure 11, the overhead of a hit amounts to 0.5msec in the $\frac{0}{5}$ and $\frac{1}{5}$ case normalized to the smallest set of $8,388,608$ different input values; this compares to 7% of the overall query runtime. In general, we experienced a $7\% - 24\%$ overhead for the 8-bit scenario, and $3\% - 13\%$ overhead for the 15-bit case. The minor overhead in the 15-bit case results from the higher shifting and masking overhead in those un-aligned bit cases.

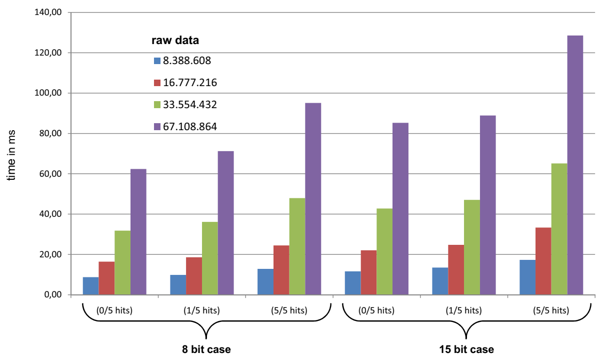
GPU versus CPU

As a final test, we measured query runtimes of the GPU-based implementation of the scan with the (from an algorithmical perspective) identical implementation on a regular CPU (Intel Xeon X5355 with 2.67GHz and 1333 MHz Front Side Bus) used in productive SAP NetWeaver BIA environments. We used a uniformly distributed test set with 255 distinct values and ran a point query on $33,554,432$ Bytes comprising $134,217,728$ different values (8-bit coding scheme). Surprisingly, we experienced a significant performance boost by a factor of 5 and more for the GPU-based implementation.

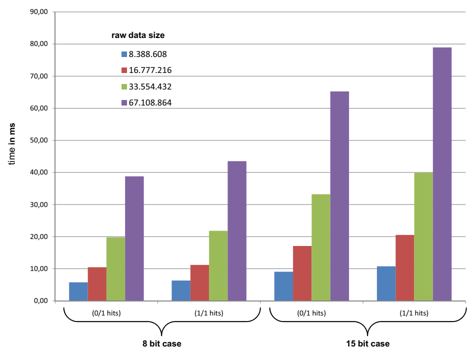
In order to be as fair as possible, we ran the test environment using all 4 cores of the Xeon X5355 CPU as much as possible by dividing the data set into 4 partitions and we ran 4 parallel scan operations. As can be seen, using all cores brings the CPU-based implementation close to the GPU version; the speed-up of the GPU-based implementation degrades to 1.29. It is also worthwhile to mention in this context that a further step to a dual-xenon system with 2x4 cores does not yield any noticeable performance gain because of higher communication costs between the CPUs.

As a final comparison, we show the number of CPU/GPU cycles to process the $134,217,728$ different values with a selectivity of $\frac{1}{255}$. Table 5 shows the result: the massive parallel system architecture of the GPU with 128 multiprocessors is far superior (by a factor of 15) to the Intel Xeon CPU with only 4 cores in this particular setup. However, we also have to note that the current work on

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61



(a) 5 possible hits



(b) 1 possible hit

Figure 11. Overhead per Value and Hit

processor	time in msec	factor
Intel Xeon X5355 (1 core)	452	4.96
Intel Xeon X5355 (4 cores)	118	1.29
Nvidia G80	91	1.0

Table 4. Runtime Comparison: GPU versus CPU

a full-scale exploitation of the SSE4.1 feature of the Intel CPU Xeon E5430 shows extremely promising results, which reverses the gathered results in certain situations. Unfortunately, an SSE implementation requires a completely different algorithm design to take full advantage of the vectorization capabilities and is therefore not directly comparable to the GPU-based implementation.

Summary

The Nvidia CUDA SDK provides an abstract programming interface that makes it extremely productive to develop GPU-based applications. We conducted a vast set of experiments using a port of the SAP NetWeaver BIA scan operation. Within our experiments, we investigated the layout of threads per block and the total number of threads. We are deeply impressed by the extremely light-weight thread model, which allowed us to schedule individual threads for small kernels extracting values out of the smallest possible data blocks. Our experiments showed significant performance gains compared to a classic CPU-based implementation; however, using OpenMP to exploit all cores of the CPU, the performance gain degraded to 18% and–with the help of the SSE technique of Intel and a different algorithmic approach–it might even vanish into thin air.

5 Conclusion

Within this paper, we reported our experience in exploiting graphic card processor technology to speed up data mining queries within the SAP NetWeaver BIA. The BIA engine is a highly scalable shared-nothing column store with two main characteristics. First, all the data is stored in a compressed form completely within main memory. Second, a large spectrum of analytical applications ranging from reporting via interactive data exploration to data mining use the same data structures without any explicit or implicit redundancy. In order to cope with these requirements, the scan and filter operators, i.e., *the* more performance-critical operators, have to be awarded with special attention with regard to an efficient implementation.

We therefore focused on the extraction procedure of bit-compressed index values referring to the user data dictionary. We showed the main characteristics of point-wise data access and of the scan operator to retrieve values satisfying a certain predicate. We also gave details of our implementation based on a G80 processor found on a Nvidia GeForce 8800 GTX. Our extensive testing showed some surprising results. First, we were absolutely fascinated by the extremely light-weight thread scheduling–a necessary prerequisite to exploit the full potential of the 128 multiprocessors of the G80. Second, we experienced a speed-up of the factor 5 compared to one core of a Xeon X5355, which is used in the SAP BIA appliance product.

Although the experiment exploiting graphic card processor technology to speed up data mining queries by improving the scan and filter operations showed positive results, we decided against the general use of graphic cards for the BIA product. The main reasons are:

- data transfer from main memory to the GPU memory is not for free and adds to the overall query runtimes;

Final edited form was published in "IEEE International Conference on Data Mining Workshops (ICDM Workshops). Pisa, 2008". S. 506-515.
ISBN 978-0-7695-3503-6
https://doi.org/10.1109/ICDMW.2008.61

- for data-intensive applications like the BIA case, the graphic card does not have enough memory (even the current Nvidia Tesla product line only provides 1.5GBytes of local RAM);
- we also experienced a significant energy consumption of $120-170$ Watt in combination with additional cooling overhead.

Although we do not pursue the GPU-based implementation for SAP's Netweaver BIA for product development purposes in the near future, we are convinced that the massive parallel architecture with the extremely light-weight thread scheduling mechanism reflects a very powerful and easy-to-develop hardware platform for many non-standard database and data mining applications.

Acknowledgements

We would like to express our gratitude to our colleagues Daniel Schneiss, Michael Faber, and Thomas Legler for their extensive support and invaluable input.

References

[1] S. Chatterjee, G. E. Blelloch, and M. Zagha. Scan primitives for vector computers. In *Proceedings of the 1990 ACM/IEEE conference on Supercomputing*, pages 666–675, 1990.

[2] Y. Dotsenko, N. K. Govindaraju, P.-P. Sloan, C. Boyd, and J. Manferdelli. Fast scan algorithms on graphics processors. In *Proceedings of the 22nd annual international conference on Supercomputing*, pages 205–213, 2008.

[3] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1061–1063, 2007.

[4] N. K. Govindaraju, N. Raghuvanshi, and D. Manocha. Fast and approximate stream mining of quantiles and frequencies using graphics processors. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, pages 611–622, 2005.

[5] H. Guo, B. He, Y. He, Q. Luo, B. Peng, and X. Xiao. Frequent pattern mining on graphics processors, 2008.

[6] M. Harris, S. Sengupta, and J. D. Owens. Parallel prefix sum (scan) with cuda. In H. Nguyen, editor, *GPU Gems 3*. Addison Wesley, 2007.

[7] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *IEEE International Conference on Parallel Architecture and Compilation Techniques*, pages 1111–1120, 2008.

[8] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 511–524, 2008.

[9] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *Proceedings of the 24th International Conference on Data Engineering*, pages 1111–1120, 2008.

[10] J. A. Ross. *SAP NetWeaver BI Accelerator*. Galileo Press, 2008.

[11] S. Sengupta, M. Harris, Y. Zhang, and J. D. Owens. Scan primitives for gpu computing. In *Proceedings of the 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 97–106, 2007.