Exploring utilisation of GPU for database applications

Slawomir Walkowiak, Konrad Wawruch, Marita Nowotka, Lukasz Ligowski, Witold Rudnicki

Interdisciplinary Centre for Mathematical and Computational Modelling, University of Warsaw, Pawinskiego 5A, 02-106 Warsaw, Poland

Abstract

This study is devoted to exploring possible applications of GPU technology for acceleration of the database access. We use the n-gram based approximate text search engine as a test bed for GPU based acceleration algorithms. Two solutions - hybrid CPU/GPU and pure GPU algorithms for query processing are studied and compared with the baseline CPU algorithm as well as with the optimized versions of the CPU algorithm. The hybrid algorithm performs poorly on most queries and only modest acceleration is achievable for long queries with high error level. On the other hand speedups up to 18 times were achieved for pure GPU algorithm. Application of the GPU acceleration for more general data base problems is discussed.

Keywords: Data base, GPU, CUDA

1. Introduction

An exponential growth of the CPU performance has slowed down dramatically after 2004 due to several factors described in the seminal paper from Berkeley group [1] as a *memory wall*, a *power wall* and an *instruction level parallelism (ILP) wall*, creating together a *brick wall*. Two former *walls* are fundamental physical problems due to overheating and memory latency, whereas the *ILP wall* is an effect of diminishing returns from an increased CPU complexity. The Berkeley group calls for a radically different way of exploiting Moore's law, by developing massively parallel 'many-core' designs and an appropriate redesign of the development tools and programming models.

One can argue that modern GPUs along with a streaming computing approach are first practical implementations of this radically new computing paradigm. Modern GPUs contain hundreds of computing cores, devoting almost all transistors to computations. The approach adopted by GPU developers is most effective for problems which are regular, massively parallel and computationally intensive, such as rendering 3D scenes in modern video games. This description fits many scientific applications as well. Initially access to the computing power of GPUs for scientific computations was limited by the necessity of expressing algorithms in terms of graphic operations. Nevertheless several interesting results were achieved, showing a potential of GPU in high performance computing [2, 3, 4]. Introduction of CUDA, and recently OpenCL programming environments enabled much simpler utilisation of the computational capabilities of GPUs. As of December 2009 there are over 400 entires to the CUDA Zone with articles devoted to applications of CUDA technology in various branches of science and technology.

Among these entriese there are only few related to the database technology. Di Blas and Kaldeway [5] discuss general applications of GPU in database technology and in particular for acceleration of the search. Veronese et al. [6] discuss implementation of GPU for text mining and show application for multi-label text categorisation. Melchor et al. [7] show implementation of the GPU for private database access. Gosink et al. [8] show implementation of data-parallel binning indexing databases.

In the current article we explore possible applications of the GPU for database technology, using the hybrid GPU/CPU implementation of approximate search in large text database. We also discuss generalisations of this approach to more general database problems. The remaining part of the article is organised as follows. First a short introduction to GPU architecture is given. It is followed by brief description of the nsearch algorithm for approximate search in text databases and detailed description of GPU accelaration of the algorithm. Then the analysis of performance is given. In the final part the possible extensions of this work to more general database applications is discussed.

2. GPU architecture essentials

Two main features that differentiate GPUs from CPUs are massively parallel architecture and utilisation of multithreading to hide latency of memory access. For example a GT 200 GPU chip has 240 execution units (cores) which are clustered in 30 so-called multiprocessors. In the fully occupied GPU there are 240 threads executed simultaneously. Moreover, when a thread requests access to a variable located in a device memory it is not waiting idle until the variable is fetched. Instead it is put aside and other threads are executed until the requested variable is delivered. At least 192 threads executed on the same multiprocessor are required to hide the latency, therefore several thousands of concurrent threads are required to fully utilize computing power of such chip.

The GPU card is attached to the host computer by the PCI express card. It is accessed by a program executed on the main CPU, which handles IO operations, memory allocations and general program control. Host program utilizes GPU via computational kernels written in the CUDA language, which is an extension of the standard C.

There are three main types of the memory accessible for a programmer - a host memory, a device memory and a small pool of shared memory located on chip. A bandwidth between host and device is one order of magnitude lower than that of the device memory and the latency gap is much higher. On the other hand using shared memory is as efficient as using registers, provided that a right access pattern is employed [9]. The highest performance of the device memory is achieved when subsequent threads access subsequent words. Then memory requests from subsequent threads can be coalesced into a single memory transaction. Random memory access can decrease bandwidth by one order of magnitude.

Threads in CUDA are hierarchically organized in warps (consisting of 32 threads) and blocks; a block size should be a multiple of and not less than 64. A computational kernel consists of multiple blocks, which can be organized into 1- and 2-dimensional grids. All threads within a block are executed on a single multiprocessor; they perform identical code. All threads in a warp execute identical instructions. Different blocks can perform different parts of the code. All blocks are executed independently.

Branching in a code should be avoided, but it is possible and does not lead to its serialisation. Instead instructions for relevant branches are executed.

This characteristics of the GPU leads to the following principles of the effective CUDA programming:

– avoid transferring data between main memory and GPU card,
– employ regular access patterns to the device global memory,
– use shared memory whenever possible, using regular access patterns,
– organize your data and computations in multiples of 64,
– use many threads - at least 192 threads per single multiprocessor.

3. Nsearch algorithm

Nsearch algorithm was developed in our laboratory for fast approximate searches in large text databases. It is the engine for bioinformatical and chemoinformatical services under development in our lab. It is used for searching the NLM Pubmed database, containing about 16 mln abstracts of scientific articles from biomedical sciences, in particular for names of chemical compounds of biological importance. These names are usually quite long and often are composed of multiple words. In most cases one can have various similar names for the same compound, one can also expect misspelling of the names both in the user query and in the original text.

A detailed algorithm description as well as an account of applications of the system will be presented elsewhere. In the current study we only give details which are relevant for the implementation of the algorithm on GPU.

The nsearch engine is based on an n-gram representation of the text. For each article all 3-grams present in the text are found and indexed. To increase specificity of the searches each article is split into 32 segments containing at least W words (currently $W = 9$), if the length of an article is lower than $32 \times W$ some of the segments are empty. An index entry for an n-gram for an article consists of the article's ID and its 32-bit map, showing distribution of the n-gram in the article. There are approximately 40 thousand unique 3-grams in the database. A number of index entries for each 3-gram follows the power-law distribution – there are few n-grams, which are present in almost all articles and there are plenty of n-grams which are present in very small number of articles.

The algorithm is based on a simple idea - a query result should contain all articles, which contain at least X% of the n-grams contained in the query, provided that these n-grams are located in one segment or at most in two adjacent segments. The original algorithm is very simple – one builds a segment-based list of articles containing sufficient number of the n-grams from a query. The search proceeds from most discriminative n-grams (small index sizes) towards least discriminative ones. First a gathering phase is performed. In this phase all articles containing the n-grams from the query are collected in a hit-list and each article receives a score equal to the highest number of the different n-grams in one segment. When the iteration count surpasses the maximal error count new articles are no longer included in the list and the algorithm switches to a pruning phase. In this phase each article in the list is checked against new index. If there is a hit in the matching segments the article's score is increased. Scores of all articles are compared with the current minimal score and articles scored lower are removed from the list (the minimal score is increased by one after processing each index).

The baseline algorithm described above is used as a reference for a comparison with a GPU-accelerated algorithm and also with an optimised version of the CPU algorithm. The optimisation of the CPU algorithm is achieved by partitioning the database into separate bins, each bin containing 256 articles, which are processed separately. In the pruning phase, if the article list for given bin is empty, the processing of this segment of database is skipped. The bottleneck of the algorithm is reading of the n-gram indexes, which is limited by a memory subsystem bandwidth. The binning approach allows for skipping unnecessary transfers from a memory.

The CPU version of the algorithm works best for queries when highly discriminative n-grams are processed when algorithm switches to the pruning phase. In such case the chance for random increase of a score is low and articles which are not similar to the query are quickly removed from the list. If the article list for a segment is empty, such segment is no longer processed, reducing overall memory access. On the other hand long queries with small number of highly discriminative n-grams and high error level results in a long article list which requires long processing and delays moment when dividend from the binning approach kicks in and cuts transfers from the memory.

There are three classes of n-grams, which are used in different ways by the algorithm. The first group are the most discriminative n-grams, which are used in the gathering phase of the algorithm. The indexes for these n-grams are very small. The second group are intermediate n-grams, which are used in the pruning phase of the algorithm. The indexes of these n-grams are relatively large and are usually read entirely. The third group are least discriminative n-grams, which are present in most of the articles. The indexes for these n-grams are large, but thanks to the binning approach only small fragments of them are read by the algorithm. This distinction is not precise and depends on the query composition and on the parameters, nevertheless it was useful for designing hybrid CPU/GPU variant of the algorithm. GPU is used best, when processing large amounts of regularly organised data. On the other hand CPU is comparatively much better suited for processing small amounts of data with complicated algorithm. One should note, that CPU and GPU can be used in parallel.

3.1. Processing of queries with GPU

The GPU advantage over CPU for the n-search algorithm is a higher memory bandwidth. It can be achieved for jobs with very regular patterns of memory access. For example 256 threads in a block can work in parallel on a single segment of the database, with $i-th$ thread computing score for $i-th$ article of given segment. To perform such computation an index must be first unpacked – maps for all articles in the segment must be aligned, including empty maps for articles which are not listed in the index. Only then all articles can be processed.

We explore two methods of using GPU for processing the query. In the first one GPU replaces CPU entirely. Both the gathering and the pruning phases are performed on GPU. The GPU version of the algorithm is simpler than on CPU. The difference between both phases is very small on GPU, because no binning is applied. Algorithm scans all indexes and computes scores for all articles encountered. If a score for an article is too small the computations for this article are omitted. Nevertheless, the whole index must be read and algorithm checks scores for all articles. After processing all indexes, the complete list of hits is sent to CPU. The serious limitation of this version of algorithm is that it requires large memory on GPU. The total size of the indexes for the Medline database is 33 GBytes. Only about one tenth of this size can be stored in the memory of a single C1070 Tesla board. On the other hand a CPU version uses compressed indexes which fit in the 13 Gbytes of memory. Unfortunately compression method used on CPU is not suitable for GPU. In the simplified compression scheme, which has not been implemented but is suitable for GPU, the index size can be estimated to be slightly less than 17 GB - it could fit on five Tesla C1070 boards.

The alternative version of the algorithm is a hybrid CPU/GPU version.

In this version GPU is used for processing large n-gram indexes with many entries, where the overhead from processing unnecessary articles is small. There is a minimal size of the index, which should be processed on GPU, depending on the relative processing speed on GPU and CPU. On the other hand indexes processed with GPU should be discriminative - one would like to process these n-grams which are not likely to be present numerous times in nearly all article, so for example ' th', 'the', ' in' or 'in ' n-grams should be processed at the last stage of the algorithm.

In the parallel version of the algorithm processing of queries is split into five stages:

1. Gathering phase, which is executed on CPU or GPU (both versions were tested).
2. Merging scores results from CPU and GPU (CPU).
3. Initial pruning phase (CPU or GPU).
4. Final pruning phase (CPU).
5. Processing of the list of candidate articles (CPU).

The third and fourth stage become one stage if GPU is used in the gathering phase. The CPU algorithm is performed separately for each small segment of database consisting of 256 articles. If indexes processed in the gathering phase don't reference articles in given segment, then this segment is never processed in the pruning phase. Also if all articles from given segment are pruned the processing of this segment is finished - the algorithm doesn't need to process segments of database which cannot contain required information.

3.1.1. GPU algorithm

The pseudo-code of the part of the algorithm performed on GPU is displayed in Fig 1. In the first part the initialization of `bigmap` and `score` matrices is performed (lines 1-8). It is assumed here that both matrices are initialized with zeros. A map of each index entry is copied to the `bigmap` matrix at the article number and the score for this article is set to 1. One should note, that threads read consecutive elements of two matrices `map` and `index` and these operations can be coalesced. In most cases the values stored in the consecutive elements of the `index` matrix will not point to consecutive entries in the `bigmap` and `score` matrices. This leads to a decreased memory performance in comparison to the reading operation. Practically in most cases the time for initialisation `bigmap` and `score` matrices is proportional to the length of these matrices and not to the number of modified elements. Assuming index size I and database size D the time required for these operations is $O(I + 2D)$.

After the initialisation the algorithm processes index after index. Each index is first read from a device memory (cost $O(I)$) and unpacked to the device memory(cost $O(D)$). Then the current scores are read from the global memory (cost $O(D)$). If the scores for the given article is higher than or equal to the minimum value at the current stage, then the main part of the algorithm is executed. First both a current map and a result map for a given article are read from the device memory (cost $O(2D)$). If there is a match between them (line 14) then the result map and the score are updated (cost $O(2D)$). The new result map is obtained from the bitwise operations performed on the result and current index map. This operation is either OR in the gathering phase (line 16) or AND in the pruning phase (line 18). Finally the score is increased by 1 (line 20). The total cost of processing single index is either $O(I + 2D)$ or $O(I + 6D)$, depending on the current score. If the score is too low for an article to be processed the cost is equal to the former value, otherwise it is equal to the latter one. A modification of the algorithm with a constant cost $O(I + 2D + D/N)$, where N is the length of query is possible. It can be achieved by unpacking all indexes to the device memory first (cost $O(I + D)$) and then processing database segment-wise using shared memory to for intermediate results. In such case only one additional operation on the device memory is required, namely reading the map for each index (cost $O(D)$).

```
// Unpack the first n-gram index and initialize score
1.  for B blocks
2.    for T threads
3.      load index[b*N+t]
4.      load map[b*N+t]
5.      bigmap[index]=map
6.      score[index]=1
7.    end
8.  end
// main loop
9.  for I-1 indexes
10.   Unpack i-th index to newmap // like above without score initialisation in line 6.
11.   for B blocks
12.     for T threads
13.       if score[current_article] >= minimum then
14.         if Match(bigmap[b*N+t],newmap[b*n+t]) then
15.           if score[current_article] > minimum then
16.             bigmap[b*N+t]=bigmap[b*N+t]| OR newmap[b*n+t]
17.           else
18.             bigmap[b*N+t]=bigmap[b*N+t]| AND newmap[b*n+t]
19.           fi
20.           score[b*N+t]+=1
21.         fi
22.       fi
23.     end
24.   end
25. end
```

Figure 1: The pseudo-code for the GPU part of the accelerated nsearch algorithm

3.2. Testing

Two separate test were performed. In the first test we compared the hybrid CPU/GPU version of the algorithm with the optimized CPU version, which is currently used as a main engine of our system. This test was performed to find out whether it is beneficial to replace the engine of the system with the new one, based on the hybrid CPU/GPU algorithm. The optimisation of the algorithm is achieved by segmentation of the database into small fragments containing 256 articles each. The algorithm is performed in each segment only if at least one article has chance to contain the query. The search in the segment is aborted once the number of hits in all articles falls below limit.

In another test the performance of the hybrid CPU/GPU algorithm was compared with that of the standard optimized CPU version. In this case tests were performed for 4 millions records (one fourth of total) from the original database using single Tesla C1070 card and single core of the host CPU. 2 699 queries with the lengths varying between 8 and 110 characters were randomly picked from the database. One should note that 12 missing 3-grams is equivalent for example to 4 independent substitutions or deletions of a single character, to removing one word consisting of 10 characters or to some combination of removing some characters in two or three contiguous

Error level	0	3	6	9	12
T(CPU) [ms]	80	99	140	191	245
T(CPU/GPU) [ms]	51	102	166	239	320
Average speedup	66.4%	91.8%	108.5%	114.5%	120.5%
Success rate	12.7%	35.1%	51.5%	59.2%	66.5%

Table 1: The average speedup, and percentage of the queries accelerated by hybrid algorithm as a function of allowed error.

blocks of characters. The number of allowed missing 3-grams varied between 0 (exact search) and 12.

The second test involved a comparison of three algorithms: the baseline CPU-based algorithm, the equivalent GPU-accelerated algorithm and the optimized version of the CPU-based algorithm, which is our standard engine. This is not a practical test for our system, because the size of the database is larger than the memory of Tesla cards which could be devoted to the running version of the system. Nevertheless it is interesting to see what can be a performance gain for a systems where GPU-accelerated solution is practical.

Due to lower compression of the data required for GPU version, the test was performed on the 1/10-th of the entire database (1.6 million records, indexes taking 3.3 GB), using a single board of the Tesla S1070 system. 1232 random queries with lengths varying between 7 to 75 characters were randomly picked from the database. The number of allowed missing 3-grams also varied between 0 (exact search) and 12.

Performance was tested on a server attached to the Tesla S1070 server and equipped with 4 Opteron 8216 CPUs with 1 MB cache, clocked at 2.4 GHz (only single core was used for testing). GPUs was attached to PCIe 1.0 16x slot.

4. Results and discussion

For comparison between the hybrid CPU/GPU and the optimised CPU algorithms we measured the speedup of these algorithms in comparison to the baseline algorithm measured as a function of the execution time of the baseline algorithm. The results of this comparison are displayed in Fig 2.

It can be seen that the hybrid algorithm in most cases does not outperform the standard CPU version and when it does, the performance gain is not high. The more detailed statistics for different levels of allowed error is given in the Table 1. It can be seen that the hybrid algorithm on average outperforms the standard algorithm for queries with higher error level. Nevertheless the performance gain from the hybrid algorithm is not substantial even for these queries – the average speedup reached only 120% for the largest error level.

There are two reasons for this disappointing result. One is the bottleneck resulting from transfering the results between GPU and CPU. In our test setup one needs to transfer about 32 MB of data to the GPU to initialize the algorithm. The transfer is carried with the maximal theoretical speed 2 GB/s, in practice about 1.2 GB is achieved. This adds about 30 ms to processing time for each query, which is non-negligible time for short queries. Another reason is that the CPU algorithm is very efficient due to the data base segmentation. In most cases the significant number of database segments are not processed any more when algorithm starts reading large indexes. Therefore the gain delivered by very fast processing of these large indexes is smaller than we

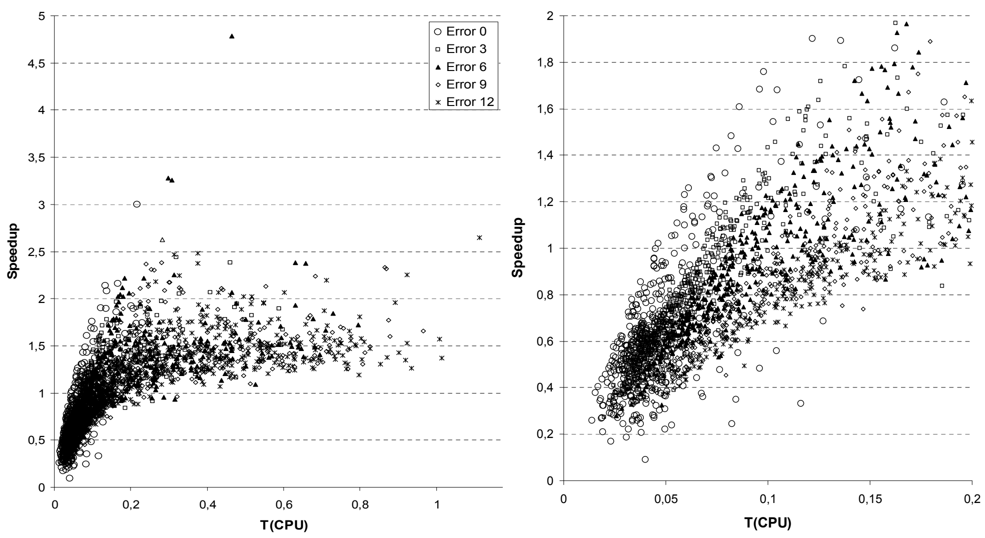

Figure 2: The speedup of the GPU accelerated algorithms versus execution time of the baseline algorithm is shown in a left panel. The close-up of the cluttered region close to origin is shown in the right panel. The queries with the different allowed error level are displayed with different symbols.

anticipated. When designing the hybrid algorithm we hoped that very fast processing of several medium-size indexes on GPU can substantially reduce the size of the article list and reduce the number of processed articles even further. The results of this experiment show that this did not happen.

Taking into account the results of both experiments we can conclude that, unfortunately, GPU is not particularly well suited for improving performance of the text search engine in our system, because big part of the work is performed for small indexes, which are better processed by CPU.

Nevertheless it is still interesting if a significant speedup can be obtained when entire algorithm is performed on GPU. The second test was designed to give an answer to this question. The speedups of the standard CPU algorithm and GPU algorithm with respect to the baseline algorithm are displayed in the Fig 3 as a function of the execution time of the baseline algorithm.

Both methods of acceleration lead to significant speedup in comparison with the baseline algorithm. We may measure the speedup just by averaging speedups obtained for individual queries, or by measuring the total time used by each algorithm to complete all queries. When using the first method the average speedup of the GPU algorithm is 18.2, whereas for CPU it is 7.0. Alternatively, the total execution time of the test suite is 260.2s for the baseline code, 63.1s for the optimized CPU version and 13.6s for the GPU version. Therefore speedups computed using the second method are respectively 4.1 for CPU and 19.0 for GPU algorithms. The discrepancy between these two results arises because the CPU optimisation works best for exact queries, which are executed faster. Therefore wem may conclude that the GPU-accelerated code is between 2.6 and 4.6 times faster than CPU-optimized code, depending on the method of averaging the individual results. One can also observe that GPU speedup is much more predictable

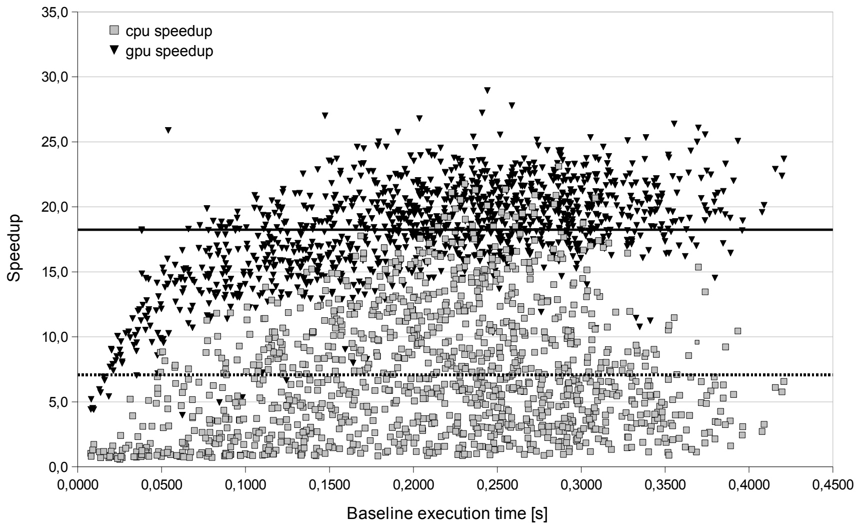

Figure 3: The speedup of two accelerated algorithms versus execution time of the baseline algorithm. The solid line denotes the average speedup of GPU, whereas the dashed one the average speedeup for CPU

- the GPU results are concentrated in the relatively narrow band in Fig 3, while results of the optimised CPU version are much more variable.

One should note that an optimisation of the GPU version of the algorithm is possible. The database segmentation can increase performance of the GPU version of the code in the same way it does it for CPU version. It has not been implemented in the GPU version yet, but the development along this lines is being carried out. The analysis of the algorithm shows that out of three operations on the global memory, namely reading of a compressed index, writing uncompressed maps to the global memory and reading scores from the global memory, two latter operations may be accelerated due to segmentation, whereas the reading of the index is constant part due to constraints of CUDA programming model. Reading of the index takes only about 1/16th of the time spent in the operations on the device memory in our test suite. The remaining 15/16th of time can be reduced using data base segmentation in the same manner as in the scalar code. Therefore it is theoretically possible to reduce the execution time of the GPU-accelerated algorithm to: $(1/16 + 2/9 \times 15/16 = 39/144)$ – almost four-fold (the 2/9 factor used here is the lower estimate of the speedup of the scalar version of algorithm). Therefore the relative speedup of the optimised version of the GPU algorithm against the optimised CPU version would be above 15 times and possibly close to 18. This algorithm was not implemented at the time of publication, nevertheless the results should not differ much from this theoretical estimate.

The most important message of this study is that very substantial acceleration of database applications may be achieved on GPU for suitable problems – speedups up to 18 times have been observed on the approximate test search problem when comparing similar algorithms.

In the current paper we described a very specific variant of the general problem, namely finding a subset of all items, which are described with given subset of labels, nevertheless the algorithms developed in the current project can be easily applied for the general problem.

GPU acceleration could be possibly used also for searches in the column oriented databases.

Column oriented databases constitute an interesting alternative to the usual row oriented databases [10, 11, 12]. Thanks to very high compression ratio and possibility of reading only these columns which are required they are very useful for decision support applications. One can increase the processing speed using GPU for example by processing highly accessed columns in GPU. The obvious condition is that they should fit in the GPU memory, but with 4GB RAM one can store one billion unique 32-bit keys on a single Tesla card. It is therefore possible to store on a single card for example database table consisting of 20 mln records with 50 columns using 32-bit keys corresponding to values. Using data parallel configuration of the database [13, 8] one can scale it to arbitrarily large tables (increasing the key size may be necessary). The developments in these directions are currently carried out in our laboratory.

Acknowledgements

This project was partially supported by ICM grant BW 1852/WR . The Tesla S1070 server was kindly donated by Nvidia as a part of the Professor Partnership Program.

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, K. A. Yelick, The landscape of parallel computing research: A view from berkeley, Tech. Rep. UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California Berkeley (2006).
URL `http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.html`

[2] R. Strzodka, M. Doggett, A. Kolb., Scientific computation for simulations on programmable graphics hardware., Sim. Model. Pract. Th. 13 (8) (2005) 667–680.

[3] S. Kupka, Molecular dynamics on graphics accelerators, in: CESCG, 2006.

[4] J. Yang, Y. Wang, Y. Chen, Gpu accelerated molecular dynamics simulation of thermal conductivities, J. Comp. Phys. 221 (2007) 799–804.

[5] A. D. Blas, T. Kaldewey, Data monster. why graphics processors will transform database processing, IEEE Spectrum 46 (9) (2009) 46–51.

[6] L. Veronese, A. F. D. Souza, C. Badue, E. Oliveira, P. M. Ciarelli, Implementation in c+cuda of multi-label text categorizers, in: GPU Technology Conference, 2009.

[7] C. A. Melchor, B. Crespin, P. Gaborit, V. Jolivet, P. Rousseau, High-speed private information retrieval computation on gpu, in: Emerging Security Information, Systems and Technologies, 2008. SECURWARE '08. Second International Conference on, 2008, pp. 263–272.

[8] L. Gosink, K. Wu, W. Bethel, J. D. Owens, K. Joy, Data parallel bin-based indexing for answering queries on multi-core architectures, in: 21st International Conference on Scientific and Statistical Database Management, Vol. 5566, Springer-Verlag, Berlin Heidelberg, 2009, pp. 110–129.

[9] NVIDIA CUDA Compute Unified Device Architecture, Programming Guide version 1.0, NVIDIA, 2007.

[10] M. Stonebraker, U. Cetintemel, One size fits all: An idea whose time has come and gone, in: Proceedings of the International Conference on Data Engineering (ICDE), 2005.

[11] P. Boncz, M. Zukowski, N. Nes, Monetdb/x100: Hyper-pipelining query execution, in: In Proceedings of the Conference on Innovative Database Research (CIDR), 2005.

[12] S. Harizopoulos, V. Liang, D. Abadi, S. Madden., Performance tradeoffs in read-optimized databases, in: In Proceedings of the 32nd Very Large Databases Conference (VLDB), 2006.

[13] D. DeWitt, J. Gray, Parallel database systems: the future of high performance database systems, Communications 36 (6) (1992) 1–26.