Comparing GPU and CPU in OLAP Cubes Creation

Krzysztof Kaczmarski

Faculty of Mathematics and Information Science,
Warsaw University of Technology,
pl. Politechniki 1, 00-661 Warsaw, Poland
k.kaczmarski@mini.pw.edu.pl

Abstract. GPGPU (General Purpose Graphical Processing Unit) programming is receiving more attention recently because of enormous computations speed up offered by this technology. GPGPU is applied in many branches of science and industry not excluding databases, even if this is not the primary field of expected benefits.

In this paper a typical time consuming database algorithm, i.e. OLAP cube creation, implemented on GPU is compared to its CPU counterpart by analysis of performance, scalability, programming and optimisation ease. Results are discussed formulating roadmap for future GPGPU applications in databases.

Keywords: OLAP cube, GPGPU, CUDA, GPGPU optimization.

1 Introduction

Recently we may observe an increasing interest in GPGPU (General Purpose Graphical Processing Unit) systems and algorithms. This new paradigm in parallel programming, placed somewhere between PCs and supercomputers, promises enormous performance with low cost machines. For many algorithms a typical stock graphical device may generate speed up of tens or in some cases of even hundreds times. Numerical problems, computation centric algorithms with good abilities for parallel execution are the most effective. Even the additional programming effort and in many cases a need of reimplementation of whole applications is worth being done. Increasing volumes of data to be processed forces all branches of science and industry to look for new computational capabilities. This paper analyses a GPGPU application in databases on a typical algorithm, i.e. OLAP cube creation including performance, scalability, evolution, programming effort and optimisation ease. Further more, we shall compare it to a similar serialized CPU solution.

Section 2 presents the implementation of the experiment on CPU and GPU. Then, section 3 provide measured results and analysis of both solutions. Section 4 concludes.

I. Černá et al. (Eds.): SOFSEM 2011, LNCS 6543, pp. 308–319, 2011.

1.1 GPGPU Parallel Processing Capabilities

General Purpose Graphic Processing Unit (GPGPU) is a new parallel processing technology developed by manufacturers traditionally colligated with graphics. For example, NVIDIA company introduced CUDA (Compute Unified Device Architecture) [1] technology that can be utilized on many popular graphics cards to perform pure numerical computations. Similar technology is developed by ATI (ATI Stream) [2]. Established consortia work on a standard GPGPU programming language called OpenCL [3].

According to the Flynn's taxonomy [4] vector-based GPU parallel processing can be classified as single instruction, multiple data (SIMD). In other words, the very same operation is performed simultaneously on multiple data by multiple processing units. Since a GPU may have hundreds cores, and each of the cores is capable of running many threads simultaneously, the potential acceleration of processing is huge. Current GPGPU devices go one step further and allow single processors to choose different execution paths – single instruction multiple threads (SIMT) architecture.

Execution of a GPGPU program usually consists of two parts: host code running on CPU and device code (also called a kernel) running on GPU. CPU code may run many classical threads and many kernels asynchronously in the same time (also on multiple GPU devices). This capabilities resulted in variety of interesting applications such as fast sorting, face recognition, FFT implementation, real-time image analysis, robust simulations and many others ([5]). As shown in section 1.2, some attempts to harness the computational power of GPUs into databases, business intelligence and particularly in OLAP applications have also taken place.

1.2 GPGPU and Databases

Among many possible usages of GPGPU programming there were already many successful applications in databases. Due to limitations of this very specific hardware they were mostly focused on relational systems. In 2004, one of the first significant papers presented several algorithms performing fast database operations on GPUs [6]: relational query with predicates, range query, k-th largest number, etc. There is also a SQLite query engine ported to GPU with average speed up at x20 for various queries but with many other limitations [7]. Other authors build a relational join composed of several GPGPU primitives [8]. This approach is very flexible, open for further extensions and low level modifications. Authors observe x2-7 speed-up when comparing to a highly optimized CPU counterpart. Yet another work focuses on external sorting mechanisms which are important for large databases. [9].

Parallel processing of the data for OLAP cubes has been discussed by Raymond in [10], who used PC clusters. Approximately 50% speed-up was obtained and the algorithm was nearly linear where the number of records per processor exceeded 500 000. The other paper by Dehne [11] showed that the cube creation can be paralleled by using multiple CPUs, but the scalability was

not linear. These approaches focus on finding an optimal and equal load for all processors in a cluster. GPU parallelism assures automatically optimum load of all cores if only proper conditions when executing a parallel code are fulfilled. Our implementation uses all available cores at 100% of instruction throughput. An analysis of different execution configurations showed that this optimal load of a multiprocessor is done well at the hardware level.

GPU usage for accelerating performance of In-Memory OLAP servers has been proposed and developed by Jedox company [12]. The system is able of huge acceleration of OLAP analyses. This approach is similar to ours but no evidence on detailed algorithm and used data volume are available.

1.3 OLAP Cube Creation Problem

In order to evaluate performance of GPU and CPU we use a typical business intelligence problem: OLAP (On-Line Analytical Processing) cube creation, which is a decision making process important support. These multidimensional data structures contain aggregated data at different levels of aggregation (i.e. sales per years, months or days). Thanks to preprocessing, building reports and ad-hoc analyses can be very quick ("on-line"). However, creation of an OLAP cube may be time-consuming and therefore OLAP cubes often have to be restricted or limited in their size (i.e. scope, meaning that granularity of the cube has to be lower).

Typical OLAP cubes used in professional projects contain millions of aggregations. For example, an OLAP cube designed and deployed at a well-known Polish supermarket ALMA S.A. contains dimensions of hundreds of levels (e.g. SKU dimension). Although the underlying (detailed) data is not of exceptionally huge volume, the cube creation would have lasted 10 hours even on a powerful server ([13]). After optimization of the input data and reducing the number of intersections within the cube this time has been reduced to 5 minutes, but still it gives the idea of the nature of the problem.

The time of OLAP cube creation can be an issue on a heavily used configuration or when the cube has to be frequently updated (Multidimensional OLAP requires rebuilding to update). Even if this is not the case, the possibility of fast OLAP cubes generating can be useful for ad-hoc analyses and reports performed on local machines (even on laptops).

2 Implementation of OLAP Cube Creation

The data in the OLAP cubes is organized into dimensions and measures. Dimensions are categorical variables like year, month, product, region etc. The measures are numerical values e.g. sales amount, transactions count, customers count etc. The process of generating an OLAP cube involves:

1. finding all intersections of the dimensions
2. defining navigation paths according to the specified hierarchies
3. calculating the aggregates

The measures which the aggregates are calculated for may be any additive functions. Typically these are sum, average, maximum, minimum, median, count. OLAP cubes are usually stored as multidimensional tables in the file system. The actual file data structure contains the values of the aggregates for all selected intersections of dimensions. If an NWAY generation was chosen, then all possible combinations of dimensions values will be computed. The input data for an OLAP cube is typically stored as: a star schema; a snowflake schema; a constellation schema or single flat, denormalized data table (*detailed table*).

Since points 1 and 2 from the generation process may be precomputed and derived from database meta data and indexes information we focus on the most changing and time consuming point 3. To simplify memory storage we use only the simplest, yet efficient way to store data: denormalized data table.

2.1 Sequential Approach

The sequential algorithm processes the data table records one by one updating the values of aggregates at the dimensions' intersections. As said before, this process can be time consuming. In real-life situations generation times of several hours can pose some difficulties, especially when there is a need to recreate the cube frequently.

```
procedure calcCubeCPU(cube, data, data_size)
1   for k:=1 to data_size do
2      intersectionIdx := calcIntersectionIdx(data[k])
3      cube[intersectionIdx] := calcAggregate(cube[intersectionIdx], data[k])
```

Fig. 1. A sequential algorithm. k denotes indexes of input data set while `calcIntersectionIdx` is responsible for calculating appropriate cube intersection upon input data. `calcAggregate` calculates a new aggregate value from the previous value and new data (min, max, sum, etc.).

Speed evaluation of the above very simple sequential solution for flat denormalized table shows that it is very fast if all data is stored in RAM. Here, we do not discuss how data was transferred into RAM memory. For comparison with GPU, which is anyway dependent on RAM, it is not important what techniques, if any, were used, and what is the physical representation of the database. Since RAM is now cheap and RAM-only databases are more and more popular, for this experiment we can assume that we only deal with an in-memory database. Memory caching, internal processors' optimizations and optimal memory reads and writes make CPU implementation really hard to be beaten by a parallel procedure, which in most cases has to perform additional tasks. Consumed time measurements for the sequential algorithm compared to parallel ones are presented in fig. 5.

2.2 A Naive GPGPU Solution

A typical way of creating a GPGPU algorithm is to start from a naive solution and than optimize data structures, implementation or the algorithm itself.

A naive parallel implementation of a sequential procedure leads to a very simple, yet very inefficient algorithm. If we consider a single database entry to be processed by a single SIMD processor, all the data is computed (ideally) in the same time. In the first step each processor reads its input data, in the second step computes a cube intersection to be updated, and in the third stores appropriate value (see fig. 2). This algorithm behaves poorly (about x20 times slower than CPU) because of many conflicts between threads trying to update the same cube intersection in the same time. Collisions may be solved by atomic addition function (available in NVIDIA CUDA). However, atomic operations degrade memory bandwidth by serialization of parallel reads and writes. It is also impossible to calculate all kinds of aggregate functions by currently available hardware level atomic functions. One can try to implement software level exclusive writes [14] but this again limits parallelism.

```
procedure calcCubeNaiveGPU(cube, data, data_size)
1   forall 1<=k<=data_size in parallel do
2      intersectionIdx := calcIntersectionIdx(data[k])
3      cube[intersectionIdx] := calcAggregate(cube[intersectionIdx], data[k])
```

Fig. 2. A naive parallel algorithm with many write conflicts. k denotes indexes of input data set, processed concurrently, while `calcIntersectionIdx` is responsible for calculating appropriate cube point upon input data. `calcAggregate` calculates a new aggregate value from the previous value and new data.

2.3 Improved GPGPU Solution

Much better results can be obtained if only conflicts in accessing cube intersections could be removed. They appear, because many threads read the data influencing the same point in the cube. But if we could assure that a single thread gets all the data for the single point, and only for this point, then there will be no writing delays. All threads could do their jobs independently. At this point we should notice that if the data is sorted, then all inputs needed for a single cube intersection's calculation lay together. Then the only difficulty is to find these sequences (clusters) in the input data. This can be hard for a sequential process but it is easy for the parallel SIMD algorithm. A single thread reads two data records, say number n and $n-1$. Then it calculates cube intersections for them (c_i, c_j). If $c_i = c_j$ then it means that index n is in the middle of a sequence for given cube intersection, but if $c_i <> c_j$ then there is a border of sequences between $n-1$ and n. The rest of the computations is as follows. Having the beginning and end points of all the sequences for a given data set, we run another parallel computations with one thread responsible for processing the single cube intersection's aggregation (fig. 3).

Algorithm Optimization. This algorithm is GPGPU efficient in a sense that it uses the hardware according to its limitations. It is also theoretically efficient since its parallel computational complexity is obviously asymptotically not higher than the sequential one.

```
procedure calcCubeImprovedGPU(cube, data, data_size, cube_size)
1   forall 1<k<=data_size in parallel do
2      intersectionIdx1 := calcIntersectionIdx(data[k-1])
3      intersectionIdx2 := calcIntersectionIdx(data[k])
4      if (intersectionIdx1 <> intersectionIdx2)
5         begins[intersectionIdx2] = k
6         ends[intersectionIdx1] = k-1
7   begins[calcIntersectionIdx(data[0])] := 0
8   ends[calcIntersectionIdx(data[data_size]) := data_size-1
9   acc := 0
10  forall 1<=c<=cube_size in parallel do
11     for begins[c]<=i<=ends[c] do
12        acc := calcAggregate(acc, data[i])
13     cube[c] : = acc
```

Fig. 3. An improved parallel algorithm without write conflicts. Assuming sorted input data set. `k` denotes input data records. `c` denotes available cube intersections.

However, a closer look at the first part of the algorithm in fig 3 shows that each thread reads two subsequent input data records. When multiplied by all n threads we get all data records read twice, which is a waste of the global memory bandwidth. This can be avoided only by using a thread block's shared memory as a short-term cache for the global memory data reads. On-chip shared memory is about 100 times faster then GPU global memory. Although this technique required programming manual transfers between global and shared memory, a significant speed-up was observed. We should note there that limitations of fast on-chip memory forced major changes in the algorithm itself. Its architecture changed from straightforward data oriented to a cluttered memory-caching.

An important optimization for SIMD processing is coalesced memory reading and writing (non-coalesced read may degrade memory access even by x16 times). For NVIDIA devices, coalesced memory access is possible if threads within the so-called half warp (16 threads for current devices) read from the same memory segment (length from 32 to 256 bytes). If these circumstances are fulfilled, all data read or write for all 16 threads is done in single instruction, which is a great improvement for parallel algorithm, when compared to sequential one and CPU. In case of our GPGPU algorithm we perform a parallel access of global memory in lines 2, 3, 5, 6, 11, 12 and 13. In lines 2, 3 and 12 reading input data may be properly organized to assure coalescing. Also reading sequence indexes in line 11 may be coalesced. Accessing cube cells for writing is rather random or we could say data dependent. Coalescing for lines 5, 6 and 13 cannot not achieved due to random character of cube intersections writing. The most important problem with this optimization is that it is highly data-size dependent. If structure of records changes (even by 1 byte) all the potential of coalescing may be lost. It could be very hard, or even impossible to achieve a general and generic coalesced memory access for varying record sizes.

Another possible place for optimization of the algorithm is calculation of aggregation value for given intersection, which is done in lines 11 and 12. This

simple *for* loop construct performed by one thread during optimization was exchanged with a parallel efficient reduction [15,16] on a set of processors with logarithmic time complexity (observed a 2x speed-up).

All the mentioned optimizations of the parallel algorithm (especially shared memory usage) make it much longer, less readable and too complicated to be presented in a pseudo code version here. Its performance, which is overall about 10 times faster than CPU one is described in the section 3.

At this point we should have a look at the CPU algorithm again. Comparisons of CPU and GPU algorithms are often not fair for CPU using not optimized code and not efficient solutions [17]. We can notice that our CPU procedure is really not efficient from the current double core CPU's point of view. If we use all the cores in GPU running parallel threads we should do the same with CPU, especially if we consider that all currently available CPUs contain at least two, four or more cores. This conclusion heads us toward a parallel CPU procedure.

```
procedure calcCubeParallel(cube, cube_size, data, data_size, no_threads)
1   ExecutionPlan plan[no_threads]
2   defParallelPlan(plan, data, data_size, cube, cube_size, no_threads)
3   for 1<=i<=no_threads in parallel do
4     calcCubeCPU(plan[i].data, plan[i].cube, plan[i].data_size)
```

Fig. 4. A CPU parallel algorithm. `defParallelPlan` procedure takes input data and an array of `execPlan` and defines data ranges for independent threads. Then all threads with different execution plans are started.

2.4 Parallel CPU Algorithm

A CPU parallel algorithm with many threads is almost as simple as sequential one. If we have sorted input data then we can, in almost constant time (dependent only on number of threads which is constant), find independent partitions to assure that concurrent threads will not access the same cube intersection. This task is done by procedure `defParallelPlan` in the fig. 4. It returns execution plan for each CPU thread containing input data partition, output cube section, data ranges, etc.

According to our expectations the multiple threads CPU algorithm behaves properly when run on a dual core processor. Two threads increase the speed by about 40%. Also as it was expected, four threads do not change anything when run on thwo cores since two of four threads must wait for the processor's time all the time.

2.5 Parallel GPU Devices Algorithm

If we have successfully executed a parallel CPU procedure on double core processor, can we do the same with two graphical devices? Modern motherboards and devices with double GPU's (like GeForce 295GTX) allows up to eight GPUs in single PC box. This would give us a tremendous speed up.

CUDA documentation says [1] that for each graphical device one must execute single CPU thread which keeps the context and host code communicating with the dedicated device. So, for two devices it seems fine to have a dual core CPU. For four devices, a quad core system seems to be more sensible. A multiple GPU test was created exactly in the same way as parallel CPU code except for the line 4, where we call a GPU calculation algorithm. Limited space does not allow us to present this straightforward procedure here.

Surprisingly, the results are exactly opposite to what we could expect. The overall execution time of a cube calculation on two GPUs was longer than with one GPU! This problem is widely discussed by CUDA programmers and seems to be a limitation of current popular motherboards and CPUs. There are some anticipated changes in Intel's Nehalem i7 architecture but not yet evaluated widely by the community. Also cost of this most powerful chip together with appropriate GPU devices places all the system rather in advanced business market not an average customer which is against the assumption of bringing supercomputer power to the masses.

Why multiple GPU performance is so bad? The reason is in the specific task we execute. As many other database algorithms it is highly data intensive. Processing power is not so crucial. The most of the procedure is just reading or writing data. Also important part of the code is only copying data from RAM to the device's global memory. This is where our program meets the PCI Express (Peripheral Component Interconnect Express) standard bottleneck. Although version 2.0 of the standard works with x16 speed that is about 3GB/s. If we plug-in a double graphic card in a single slot it degrades to x8. Therefore, the resulting times must be much worse than for a single GPU. But database applications are almost always data intensive and scalability of GPGPU database solutions for multiple devices is right now questionable.

3 Results

3.1 Experiment Set Up

The test environment constituted a machine with Intell's 3Ghz CPU Core2 Duo, 4GB of RAM and single NVIDIA GTX 295 card. This GPU device is build of two multiprocessors (1.242 GHz, 240 cores and 896MB of DDR3 memory each). Sequential algorithms was executed on CPU while parallel algorithm on one or both GPU devices.

The data input used for testing was a flat data table (detailed table) with 25 millions records. Each record contained fields: *Year*, *Month*, *Day*, *Group*, *Product* and *Amount*. The data has thus 2 dimensions: time and product with 3 levels (year, month, day) in the time dimension and 2 levels in the product dimension (product group, product). The measure used was a sum of sales amount. The resulting OLAP cube with the highest granularity with all possible intersections contained 1 424 016 aggregations.

In the experiment, for CPU code pure C as the only programming language, while for GPU CUDA for C from SDK version 3.0 was used.

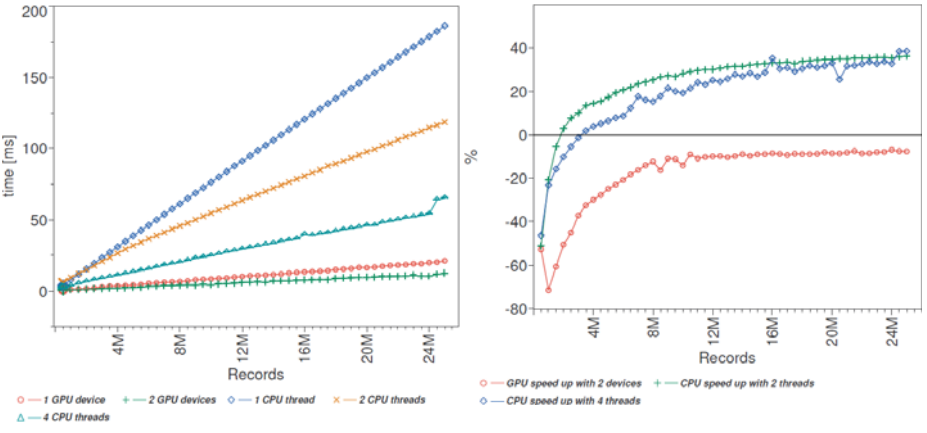

Fig. 5. (left) Running time against data sample size. **(right)** Performance improvement when adding more threads/GPU devices.

3.2 CPU and GPU Performance

The experiment was run on subsequent samples of the input data from 1 to 25 millions records. We divided performance comparison into two parts. The first one focuses only on pure algorithm running times, the second compares scalability of both algorithms and executing hardware.

The graph of running times, including only pure algorithm execution, without any additional set up operations, are given in the fig. 5 (left). We measured only the time spent by each thread inside the cube calculation procedure. Therefore we performed the experiment with five different execution set-ups: one, two and four CPU threads, one and two GPU devices. Since the algorithm is linear and can be well divided into independent partitions the running times as expected present a speed up of about 50% when run in multiple threads and multiple devices configurations, as expected.

The GPU implementation of the algorithm is about 10 times faster than the CPU one. Single GPU peek performance processes more than 1 million records per millisecond, while single CPU thread achieved only about one hundred thousands records per millisecond.

However, we must point out here that the GPU algorithm requires time consuming transfer of data between RAM and graphical device's global memory. This very time consuming operation performed over PCIE bus interface consumes much more time than the algorithm itself and for 25M records took about 340ms (17 times longer than the algorithm itself), which make the overall GPU execution pointless if the cube generation, or any other operation, is the only task to be done. This problem is mostly not mentioned by many authors who often show only "computation time". Utilization of GPGPU in databases may be sensible if data may be stored in the device's memory for longer time. Only device memory based data storage and execution engine working both on CPU and GPU sides could properly handle this mission.

The second analysis presents results of the algorithm and execution engine (CPU or GPU) scalability when switching from single to multiple processors environment. The fig. 5 (right) presents percentage improvement of overall procedure execution time (including set up operations, memory allocation, cleaning, data copying forth and back). As expected we can see good improvement around 40% for CPU, while for GPU we can observe degradation of overall speed. As it was said earlier this problem is caused by PCI-Express interface which slows down for concurrent data transfers. The final results for two graphical devices running in parallel are about 5% slower than one graphical device. The only benefit be could observe of using two devices is the doubled memory capacity, which would enable us to execute tasks of more than 64M records.

4 Conclusions

In this paper we analysed CPU and GPU implementation of a typical database computational algorithm. Both CPU and GPU code was highly optimised to assure peek performance.

When comparing just database records processing time, the GPU version proved to be significantly faster than its sequential CPU counterpart. In case of NVIDIA CUDA capable devices number of threads physically executed in the same clock cycle varies from 8 for the simplest mobile GPUs to hundreds for the most advanced ones (like GPUs from Fermi or Tesla lines). In this case, since most of the algorithm is concurrent, Amdahl's law promises great performance and it can be observed.

Our experiments proved that the practice of GPGPU programming is far from theoretical capabilities if special care on implementation is desisted. Procedures need to optimized at a very low level. The best results can be achieved only for full memory bandwidth and instruction throughput.

The CPU version was much easier to be codded. It took only about 2 hours to get it working with two threads, while GPU code took about 2 weeks of debugging, profiling and optimizing. Although, there are API interfaces for many programming languages still GPGPU programming is highly limited by hardware capabilities. An inexperienced programmer may face an unbreakable barrier of shared memory banks conflicts, memory bandwidth, number of registers per processor, templates meta programming, limitations in task synchronization or inter task communication.

Predicted advantages of utilization of GPU devices in databases:

- Enormous instruction throughput and data bandwidth. If there is a possibility of storing all the database in a GPU device memory (the most advanced cards may offer together up to 16GB) one may expect really spectacular results by minimizing time costly transfers between CPU RAM and device. However, there is no clear idea how a generic database storage can be organized to optimize GPU performance.
- Possible speed up of algorithm building blocks when using many well documented APIs with primitives like sorting, prefix-sums, array packing, etc.

– A GPU device may be often exchanged or multiplied up to 8 devices at low cost. Even an average PC can be equipped with a high performance GPU device achieving a low cost excellent improvement in performance.

Among properties of GPU devices we should enumerate sources of potential problems for database management systems:

– GPGPU hardware and its programming languages are still at the beginning of the evolution process. Programming is difficult and very low-level, highly bound to hardware capabilities and internal construction of a particular device.
– Implementations of memory intensive algorithms must be optimized for given, fixed input data. This is very unpleasant for database applications, which must cope with very different and changing data. Moreover, relational databases with well defined, fixed columns are not the only ones on the market. Unfortunately, due to the highly vector-like processing nature, GPUs are not yet ready for unstructured data. So, there is no well known API which could help to organize a general database storage and assure peek GPU performance by for example memory coalesced reads and writes.
– There is no clear way how stored procedures can be implemented at the GPU side. The same problem arises for object databases when objects' behaviour is to act concurrently for many objects in the same time.
– There are so far no evidences that indexes as structures, which must be often randomly accessed and modified by many threads in the same time, can be currently efficiently implemented at GPGPU side.
– Although a single thread in NVIDIA CUDA is executed independently from other threads, a decrease of efficiency may be observed for highly branching algorithms: when a single thread computes its branch, other threads at given streaming processor must wait.
– Scalability of multiple devices systems suffer from PCI-E interface limitations and may slow down the overall application.
– ACID properties need independent host threads running on single or many devices in the same tine. To be evaluated if possible with current GPU devices at proper level.

Last but not least, we believe that GPGPU processing will be important for future DBMSs and we expect effort of both GPU and database communities to achieve a significant cooperation of both technologies.

References

1. NVIDIA Corporation, CUDA programming guide (2009), `www.nvidia.com/cuda`
2. ATI Corporation, ATI stream sdk v2.2 documentation, `http://developer.amd.com/gpu/ATIStreamSDK`
3. Khronos Group, OpenCL - the open standard for parallel programming of heterogeneous systems, `http://www.khronos.org/opencl/`

4. Flynn, M.J.: Some computer organizations and their effectiveness. IEEE Transactions on Computers C-21, 948–960 (1972)
5. NVIDIA Corp., CUDA C posters, `www.nvidia.com/object/SC09posters.html`
6. Govindaraju, N.K., Lloyd, B., Wang, W., Lin, M.C., Manocha, D.: Fast computation of database operations using graphics processors. In: SIGMOD Conference, pp. 215–226. ACM, New York (2004)
7. Bakkum, P., Skadron, K.: Accelerating sql database operations on a gpu with cuda. In: Kaeli, D.R., Leeser, M. (eds.) GPGPU. ACM International Conference Proceeding Series, vol. 425, pp. 94–103. ACM, New York (2010)
8. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N.K., Luo, Q., Sander, P.V.: Relational Joins on Graphics Processors. In: Wang, J.T.-L. (ed.) SIGMOD Conference, pp. 511–524. ACM, New York (2008)
9. Govindaraju, N.K., Gray, J., Kumar, R., Manocha, D.: GPUTeraSort: high performance graphics coprocessor sorting for large database management. In: SIGMOD, pp. 325–336 (2006)
10. Raymond, T., Wagner, A., Yin, Y.: Iceberg-cube computation with pc clusters. In: Proc. ACM SIGMOD Conf. (2001)
11. Dehne, S.H.F., Eavis, T., Chaplin, A.: Parallelizing the data cube. In: Proc. Eighth Int'l Conf. Database Theory (January 2001)
12. Lauer, T., Datta, A., Khadikov, Z.: A CUDA-powered in memory OLAP server. NVIDIA Research Summit (2009)
13. Koral, K.: Benefits from BI at ALMA. In: SAS Business Forum, Poland (2007)
14. Shams, R., Kennedy, R.A.: Efficient histogram algorithms for NVIDIA CUDA compatible devices. In: Proc. Int. Conf. on Signal Processing and Communications Systems (ICSPCS), Gold Coast, Australia, pp. 418–422 (December 2007)
15. Blelloch, G.E.: Prefix sums and their applications. In: Sythesis of parallel algorithms, pp. 35–60. Morgan Kaufmann, San Francisco (1990)
16. Harris, M.: Optimizing parallel reduction in CUDA (2008)
17. Lee, V.W., Kim, C., Chhugani, J., Deisher, M., Kim, D., Nguyen, A.D., Satish, N., Smelyanskiy, M., Chennupaty, S., Hammarlund, P., Singhal, R., Dubey, P.: Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. SIGARCH Comput. Archit. News 38(3), 451–460 (2010)