Revision of Relational Joins for Multi-Core and Many-Core Architectures*

Martin Kruliš and Jakub Yaghob

Department of Software Engineering
Faculty of Mathematics and Physics
Charles University in Prague
{krulis, yaghob}@ksi.mff.cuni.cz
http://www.ksi.mff.cuni.cz/

Abstract. Actual trend set by CPU manufacturers and recent developement in the field of graphical processing units (GPUs) offered us the computational power of multi-core and many-core architectures. Database applications can benefit greatly from parallelism; however, many algorithms need to be redesigned and many technical issues need to be solved. In this paper, we have focused on standard relational join problem from the perspective of current highly parallel architectures. We present comparison of different approaches and propose algorithm adaptations which can better utilize multiple computational cores.

Key words: relational, join, intersection, parallel, multi-core, many-core, GPGPU

1 Introduction

Joins in relational databases have been studied thoroughly since they are one of the most essential operations. Even though the algorithms are not likely to be improved anymore, the implementation details need to be revised every few years as they are sensitive to many aspects of hardware architectures, which are changing constantly.

In the past few years, parallel architectures become available for common users. Central processing units with up to 12 logical cores are occupying mainstream segment of the market and graphical cards containing tens or even hundreds of processing units are present in almost every PC. Furthermore, the field of general purpose computing have encountered several major changes in both software accessibility[1] and hardware design.

Unfortunately, sometimes the parallelism cannot be exploited as easily as hardware manufacturers suggest. There are many concerns that need to be taken into account. In case of multi-core CPUs, there are issues regarding memory

* This paper was supported by the grant SVV-2011-263312 and by the grant agency of Charles University (GAUK), project no. 277911.

[1] A release of OpenCL framework for instance

access. Namely the cache coherency maintenance on multiple cores, bottleneck created by multiple cores accessing memory via single bus, or different memory access latency caused by NUMA [1] factor. In case of many-core GPUs, the memory access problems become even more severe. First problem is that the GPU has its own memory which is connected to the rest of the system via external PCI-Express bus. Second problem is that the caches of a GPU are much smaller than caches of ordinary CPU, therefore access to the memory must be carefully optimized. In addition, the GPU architecture is different to the architecture of CPU as it exploits single instruction multiple data paradigm. In this paper, we will attend described issues and present some solutions.

For the sake of simplicity, we have reduced the problem of join operations, since they have many variations based on its purpose, datatypes, existence of indices, etc. Henceforth, we define join operation as a simple intersection of two key sets. Both sets have unique keys, which are numbers of fixed size (e.g. 32-bit integers). We also assume that the keys are distributed almost uniformly among their domain.

The paper is organized as follows. Section 2 reviews related work. Section 3 summarizes shortly standard serial algorithms for the join problem. Section 4 presents and compares possible parallelization techniques for serial algorithms. Section 5 addresses problems of many-core architectures (such as GPU cards) and revise parallel algorithms from their perspective. Finally, Section 6 presents experimental results and Section 7 concludes.

2 Related Work

Relational joins [2] are one of the most important database operations. There are many papers related to the subject of parallel join processing taking many different views. Liu et al. [3] focused on the pipelined parallelism of multi-join queries. In contrast, we are focusing on accelerating processing of single join query. Lu et al. [4] compared four hash-based join algorithms on a multiprocessor system. Schneider et al. [5] studied join algorithms in share-nothing system. Cieslewicz et al. [6] implemented highly parallel hash join on the Cray MTA-2 architecture. Recently, Changkyu et al. [7] reviewed hash and sort join algorithms from the perspective of modern multi-core CPUs.

New modern many-core architectures, such as GPU [8] become available for programers thanks to GPGPU languages and frameworks such as CUDA [9] and OpenCL [10]. The GPGPU techniques are summarized in survey of Owens et al. [11]. The GPU has already adopted many tasks used in query processing, such as sorting [12][13]. Bakkum et al. [14] implemented SQL query processor accelerated by GPU. Bingsheng et al. [15] studied several types of join algorithms using low level data primitives such as split or gather/scatter implemented on GPUs. We will reflect some of their findings in our work, but we use different approach to the problem that will exploit new hardware features and architectures of today.

3 Serial Joins

3.1 Brief Overview

In this section we will review existing join algorithms shortly, so we can refer to them later. We also pinpoint their strengths and weaknesses in the perspective of current hardware properties.

There are two major approaches to solving join problem. First approach widely used was the merge join. Both joined sets are sorted first and then merged in single pass. Originally, it was designed for systems with small amount of internal memory since the sorting can be achieved using algorithms for external memory and the merge phase requires little additional memory. At present time, the merge join algorithm is used only in special cases. For instance, if the joined sets are (or can be) yielded by previous stages of query execution pipeline already sorted, the sorting phase may be omitted.

As the amount of internal random access memory in computers grew, database systems replaced merge join with hash join algorithm. Hash join stores one of the sets into a hash table and then looks up each item from the second set in the table. If the item is found there, it is included into the result. Hash join principles are used in the state of the art algorithms of today.

Special case of hash join algorithm uses bitmap as a hash table. In case the key domain is sufficiently small, we can create a bit field where each bit corresponds to one item from the domain. In our case, such field requires 2^{32} bits (512 MB). When a bit is set, the corresponding key is present in the hashed set and vice versa.

3.2 Considering Hardware Aspects

There are two most important aspects of current CPU architectures that affects join performance [7]:

- CPU caches,
- and virtual memory translation.

CPU uses two or three level caches to reduce memory access latency. The size of the cache is much smaller than size of main memory, thus accessing large memory areas randomly is not very efficient. Furthermore, the processor employs prefetch mechanisms which tries to identify memory that is likely to be needed soon and load it into the cache in advance. These mechanisms work on their best if the memory is accessed sequentially.

Second important issue is virtual memory translation. On IA32 architecture, the translation is performed through page tables [16]. Depending on virtual address space and some other details, the address is translated by looking up records in 3 or 4 tables. Therefore, each virtual memory access leads into 4 or 5 physical memory accesses. In order to reduce this overhead, modern CPUs implement TLB cache for translation. However, TLB cache is limited in its size, thus if we access larger amount of virtual pages randomly, the TLB misses are more frequent and memory latency rises.

In the light of current CPU architectures, we can redesign both merge and hash join algorithms. The sorting algorithm, which takes more than 90% of merge join time, can be optimized [17] and the hash join can compact its memory access pattern by introducing partitioning techniques [7] that we call bucketing. These optimization are well beyond the scope of this paper, but we will exploit bucketing for parallel reasons later.

4 Multi-Core Implementation

In this section, we will consider multi-core CPU architectures to increase performance of join algorithms. We will examine strengths and weaknesses of merge and join algorithms from the parallel point of view and then improve them with the bucket partitioning.

4.1 Direct Approach

Merge Join can be parallelized quite easily as the sorting algorithms are known to scale well [17]. The final merge can be parallelized as well by separating one of the sorted sets into fragments and processing each fragment concurrently. The only complication is that we need to find corresponding fragments in the second set so both fragments can be merged; however, these fragments are easily identified by simple application of binary search algorithm.

Hash join does not scale as well as merge join. We need to synchronize access to hash map (or the bitmap) by some kind of locking mechanism or atomic operations. The synchronization creates bottleneck of the algorithm, therefore the speedup is rather small as we can see in Section 6.

4.2 Bucketing Exploitation

The best way to solve any problem in parallel is to create independent tasks so that they can be processed by multiple threads without explicit synchronization. In case of join algorithms, we can employ the bucketing principle. We use hashing function to separate data into buckets and each bucket can be processed by a different thread. For the sake of simplicity, we always divide the set among 2^k buckets using upper k bits from the key. We can use more sophisticated hashing functions in case the key distribution is distorted in any way.

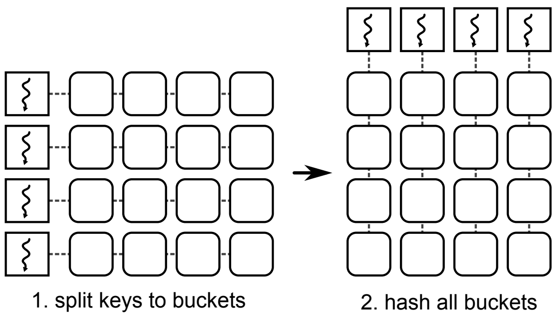
Divide And Conquer. One of possible solutions is based on divide and conquer programing paradigm. It requires that the hashing function can be incrementally refined. In our case, we can simply use more bits from the key for more fine grained bucketing. The algorithm is similar to QuickSort [18]. In the first pass it takes the array representing set of keys and reorder them so that keys with uppermost bit equal to 0 are on the left and keys starting with 1 are on the right. Then both parts of the array are processed recursively using two uppermost bits, etc. The recursion stops in predefined depth or when a bucket size decrease below predefined threshold. This approach has several advantages.

- The algorithm does not require any additional memory. We can perform splitting phases in place as the QuickSort does.
- Recursive calls can be processed by different threads since they work on disjoint ranges of the array.
- We can use different grain for different buckets, thus achieving more balanced workload even if the keys are not distributed homogeneously.
- Each splitting phase works sequentially so it benefits from caches.

After partitioning both sets into buckets, these buckets are processed concurrently. A simple hash join algorithm (using bitmap as a hash table) is used locally. Thanks to the partitioning, we require only 2^{32-k} bits for the bitmap.

Two-Pass Bucketing. Previous approach is quite effective, but it has two flaws. It takes a while before recursion reaches sufficient depth to fully exploit parallelism and we need to process each item $\mathcal{O}(k)$ times. We can achieve bucketing by one-pass algorithm if we put items directly into their buckets. The problem is that if we want to run such algorithm concurrently, access to these buckets needs to be synchronized, thus creating additional overhead. We propose lock-free adaptation, which requires two passes, but it can run concurrently on all available cores from the beginning.

We create a matrix of buckets $T \times b$, where T is number of threads available and b is number of buckets, so that each thread has its own set of buckets. We also expect that $T \leq b$, otherwise the following stages of the algorithm cannot fully utilize all the threads. For the sake of simplicity, we describe the algorithm for $T = b$.



In the first phase, each thread takes its equal share of input set and divide it into its local buckets. When the keys are scattered to the buckets, we prepare vector of b hash tables (in our case bitmaps). Each thread takes one column of the matrix (i.e. group of buckets containing keys with the same prefix of length k) and save them in the corresponding hash table. Threads can access hash tables without locking since each thread works on a separate bucket group and each group has its own table. Finally, the second key set is processed concurrently. Proper bucket is determined for each key in the set and the key is looked up in its hash table. No synchronization is required as the hash tables are used only for reading.

5 Adaptation for Many-Core Architectures

Before we start designing join algorithms for many-core GPUs, we recall the most important facts about the architecture first. These facts need to be considered carefully since they affect performance significantly.

5.1 GPU Architecture

Kernel Execution. There are two main concerns about GPU architecture. The first is rather specific program execution. Portions of code, which are to be executed on GPU, are called kernels. Kernel is a function that is invoked multiple times – once for each working thread. Each thread gets the same set of calling arguments and a unique identifier from 0 to $N - 1$ range where N is the number of invoked threads. The kernel uses thread identifier to select proper part of the parallel work. The thread identifiers may be also organized in two or three dimensional space for programmers convenience.

The thread managing and context switching capabilities of a GPU are very advanced. Therefore, it is usually better to create huge amount of threads even if they execute only a few instructions each. More threads are better for load balancing and fast context switching can be used to hide latency of accessing global memory.

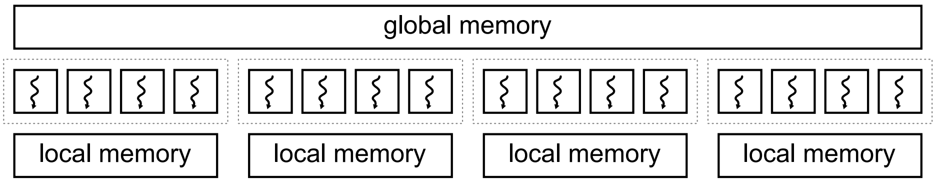


Fig. 1. Thread organization

Threads are aggregated into small bundles called groups. A group usually contains tens to hundreds threads which execute the kernel in SIMD[2] or virtual

[2] Single Instruction Multiple Data

SIMD fashion. Every thread executes the same instruction, but it has its own set of registers, thus working on different portion of data. SIMD suffers from branching problem – different threads in the group chose different branches of 'if' statements. To execute conditional code properly, all branches must be executed by all threads and each thread masks instruction execution according to local result the condition. On the other hand, SIMD approach eases synchronization within the group and threads can collaborate through shared local memory.

Finally, we have to point out that most of the graphic hardware is not capable of executing different kernels simultaneously, even if they do not occupy all the processing units. The only architecture currently capable of simultaneous kernel execution is the newest NVIDIA Fermi [8].

Memory Organization. We have four different memory address spaces to work with when programming GPUs:

- host memory,
- global memory,
- local memory,
- and private memory.

The *host memory* deserves extra attention, since it is not directly accessible from processing units. Input data needs to be transferred from host memory to graphic device memory and the results needs to be transferred back to host memory when the computation terminates. Furthermore, these transfer use PCI-Express bus, which is rather slow (in comparison with internal memory buses).

The *global memory* is directly accessible from GPU cores. Input data and computed results of a kernel are stored here. The global memory has high both latency and bandwidth. In order to access the global memory optimally, threads in one group are encouraged to use *coalesced loads*. Coalesced load is performed when all the threads in the group loads or stores continuous block of aligned memory, so that each thread processes one 4-byte word (see Figure 2).

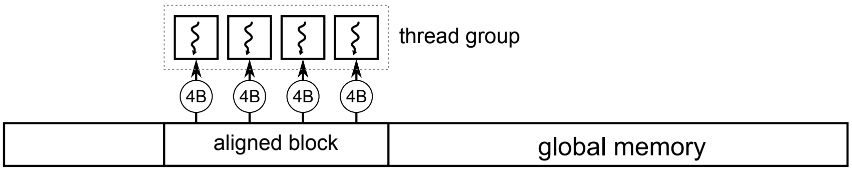


Fig. 2. Coalesced load

The *local memory* is shared among threads in one group. It is very small (in order of tens of kB) but almost as fast as GPU registers. Local memory can play role of program-managed cache for global memory, or the threads may store partial results in here while they cooperate on a task. The memory is separated into several (usually 16) banks. Following two 4-byte words are stored in following two banks (modulo number of banks). When two threads access the same bank (except if they read the same address), the memory operations are serialized.

Finally, the *private memory* is private to each thread and corresponds to processor registers. Private memory size is very limited (tens to hundreds of words), therefore it is suitable just for few local variables.

5.2 Algorithm Design

There are three different approaches to implementing join algorithm on GPU. First exploits the GPU power for sorting. Sorting algorithms for GPUs have been already thoroughly studied [12][13], but our preliminary results indicated that the sorting operation takes too much of precious time and the join algorithm can be implemented better. Second is to implement bucketing on GPU and create buckets small enough so they fit to local memory. This is also promising way; however, fine grained bucketing takes also quite large amount of time.

In our work, we examined the third possibility – applying hash join using bitmap as hash table on GPU. Unfortunately, the bitmap would require continuous memory block of 512 MB, which cannot be allocated on present GPU hardware due to some technical limitations. We have applied lightweight bucketing in order to reduce bitmap size so it would fit GPU memory. The algorithm is designed as follows:

```
for both input sets do
  calculate histogram (number of items in each bucket)
  perform prefix sum on the histogram (compute starting offsets)
  split keys into the buckets
end for
for each bucket do
  prepare empty bitmap
  for $\forall x$ of the first set in the bucket do
    set bit corresponding to $x$ to 1
  end for
  for $\forall y$ of the second set in the bucket do
    if bit corresponding to $y$ is 1 then
      include $y$ to the result
    end if
  end for
end for
```

The histogram is computed in highly parallel fashion using atomic increment operations. The splitting algorithm works similarly like the histogram computation, but it uses local memory as output cache, so that the data are written to global memory in larger blocks.

We have chosen total size of 16 buckets for the current hardware. This way the size of bitmap required for hashing is reduced to 32 MB which is feasible.

When the bitmap is being filled, a thread is created for each item x in the bucket. Data are written using atomic OR operation, thus all conflicts are avoided. After they complete, a thread is created for each key y. Threads in one group uses local memory as a cache for keys that has been included into the result. When the local cache is full, it is spiled into the global memory using atomic add operation to allocate position in output buffer.

6 Experimental Results

6.1 Methodology, Hardware, And Testing Data

Each test was performed 10 times and presented values are the average of measured times. All times were well within ±10% limit from the presented value.

Scalability tests were performed on Dell M910 server with four six-core Intel Xenon E7540 processors with hyper-threading (i.e. 48 logical cores) clocked at 2.0 GHz. Server was equipped with 128 GB of RAM organized as 4-node NUMA. GPU tests were conducted on a common PC with six-core Intel Core i7 870 CPU with hyper-threading clocked at 2.93 GHz. The machine was equipped with 16 GB of RAM and one NVIDIA GTX 580 GPU card with 512 CUDA cores and 1.5 GB of RAM. A RedHat Enterprise Linux (version 6) was used as operating system on both machines.

We use three data set for testing – pairs of $4M$, $8M$, $16M$ key sets. Each set was generated randomly with uniform distribution over the 32-bit key domain. We did not use larger data as they would not fit to the GPU memory.

6.2 Scalability on Multi-Core CPUs

First, we test scalability of CPU multi-core algorithms. Horizontal axis shows number of active threads and vertical axis represents time in ms. The *merge* denotes merge join algorithm, *hash* is hash join with bitmap hash table employing atomic operations for synchronization, *split* stands for divide and conquer recursive bucketing algorithm, and finally *bucket* represents two-pass bucketing algorithm.

Figure 3 shows several things. First, that the algorithms does not scale very well beyond 8 cores. This is most likely caused by relatively low memory bandwidth. As multiple cores share the same memory controllers and caches, they slow down each other. Second, the best algorithm is obviously bucketing algorithm as we expected [7]. Finally, even though the merge join algorithm does not perform very well on single core, its scalability makes him worthy adversary on eight and more cores.

Figures 4 and 5 shows how the algorithms perform on smaller datasets. The merge join is more suited for smaller sets as it can better utilize CPU caches and TLB. On the other hand, hash join performs worse on smaller data sets as it requires initialization of large bitmap which size is relative to the size of key domain, not the size of joined sets.

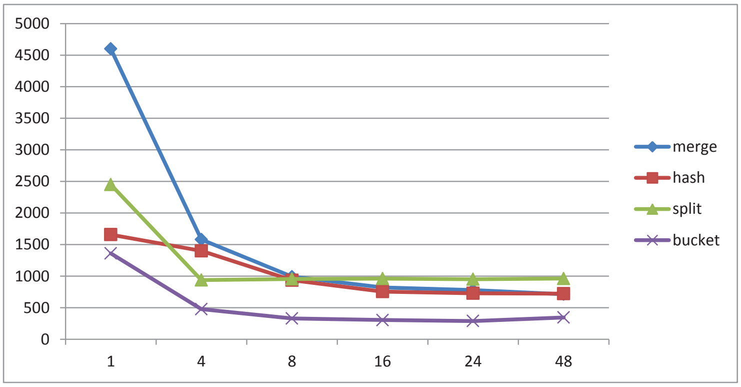


Fig. 3. Join of two sets, 16M keys each

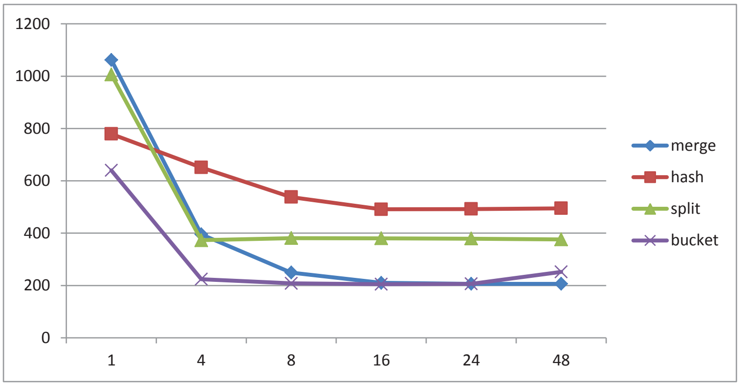


Fig. 4. Join of two sets, 4M keys each

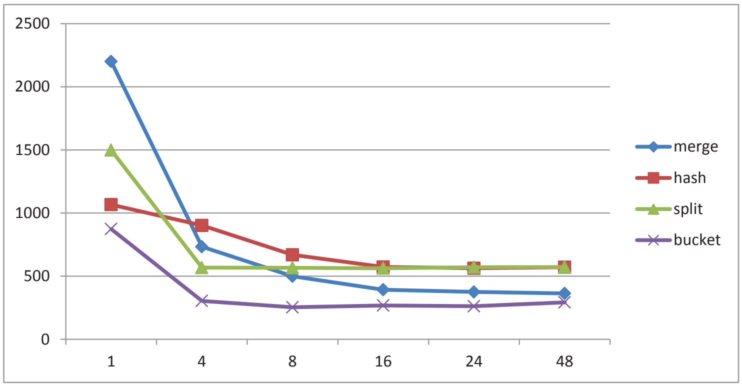


Fig. 5. Join of two sets, 8M keys each

6.3 Performance on GPU

We have compared our GPU algorithm to both serial and parallel algorithms on CPU. The vertical axis represent times in ms. The *hash(1T)* denotes hash join algorithm executed on single thread, *bucket(12T)* is two-pass bucketing algorithm executed on 12 cores, *GPU comp.* stands for GPU algorithm (only computation time), and finally *GPU total* represents times of GPU computation including data transfers between host and device memory. The *hash(1T)* and *bucket(12T)* were selected as best serial and parallel representatives on Core i7 CPU.

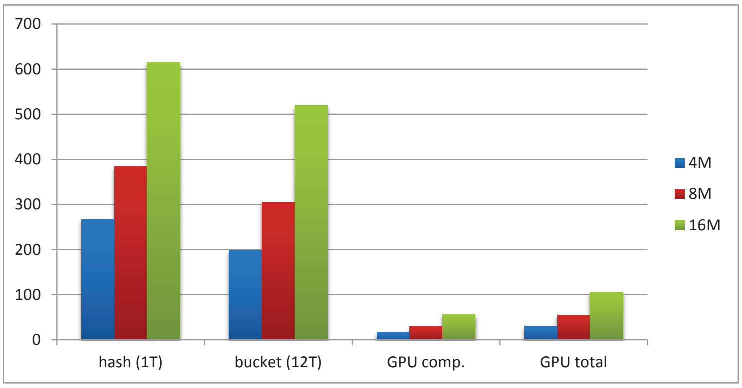


Fig. 6. Comparison of GPU and CPU algorithms

The GPU algorithm runs more than 10× faster than fully parallelized bucketing algorithm running on 12 cores, and still more than 5× faster if we take the time for data transfers into account. We can also observe that the transfer of data to the graphic card and back takes almost the same amount of time as the computation itself.

7 Conclusions

This paper presents comparison of join algorithms and their modifications in the perspective of multi-core CPUs. Partitioning of joined sets into buckets improves performance significantly and it can be easily adopted for lock-free concurrent processing. We have also presented an adaptation of join algorithm that works well on GPU many-core hardware and achieve more than 10× speedup to the CPU parallel algorithms.

There is much work to be done still. We are going to compare our GPU algorithm with other possible implementations and make more thorough analysis of them. We are also planing to compare our join algorithm with standard in-memory database systems. In our future work, we will focus on exploiting many-core GPU hardware to accelerate common database operations and index searching.

References

1. NUMA: Non-Uniform Memory Architecture. http://en.wikipedia.org/wiki/Non-Uniform_Memory_Access
2. Mishra, P., Eich, M.: Join processing in relational databases. ACM Computing Surveys (CSUR) **24**(1) (1992) 63–113
3. Liu, B., Rundensteiner, E.: Revisiting pipelined parallelism in multi-join query processing. In: Proceedings of the 31st international conference on Very large data bases, VLDB Endowment (2005) 829–840
4. Lu, H., Tan, K., Shan, M.: Hash-based join algorithms for multiprocessor computers. In: Proceedings of the 16th International Conference on Very Large Data Bases, Morgan Kaufmann Publishers Inc. (1990) 198–209
5. Schneider, D., DeWitt, D.: A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. ACM SIGMOD Record **18**(2) (1989) 110–121
6. Cieslewicz, J., Berry, J., Hendrickson, B., Ross, K.: Realizing parallelism in database operations: insights from a massively multithreaded architecture. In: Proceedings of the 2nd international workshop on Data management on new hardware, ACM (2006) 4
7. Kim, C., Kaldewey, T., Lee, V., Sedlar, E., Nguyen, A., Satish, N., Chhugani, J., Di Blas, A., Dubey, P.: Sort vs. Hash revisited: fast join implementation on modern multi-core CPUs. Proceedings of the VLDB Endowment **2**(2) (2009) 1378–1389
8. NVIDIA: Fermi Architecture. http://www.nvidia.com/object/fermi_architecture.html
9. NVIDIA: CUDA. http://www.nvidia.com/object/cuda_home_new.html
10. Khronos: OpenCL – The open standard for parallel programming of heterogeneous systems. http://www.khronos.org/opencl/
11. Owens, J., Luebke, D., Govindaraju, N., Harris, M., Kruger, J., Lefohn, A., Purcell, T.: A Survey of General-Purpose Computation on Graphics Hardware. In: Computer graphics forum. Volume 26., Wiley Online Library (2007) 80–113
12. Sintorn, E., Assarsson, U.: Fast parallel GPU-sorting using a hybrid algorithm. Journal of Parallel and Distributed Computing **68**(10) (2008) 1381–1388
13. Greß, A., Zachmann, G.: GPU-ABiSort: Optimal parallel sorting on stream architectures. In: Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium, Citeseer (2006) 45
14. Bakkum, P., Skadron, K.: Accelerating SQL database operations on a GPU with CUDA. In: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units, ACM (2010) 94–103
15. He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., Sander, P.: Relational joins on graphics processors. In: Proceedings of the 2008 ACM SIGMOD international conference on Management of data, ACM (2008) 511–524
16. Page Tables. http://en.wikipedia.org/wiki/Page_table
17. Chhugani, J., Nguyen, A., Lee, V., Macy, W., Hagog, M., Chen, Y., Baransi, A., Kumar, S., Dubey, P.: Efficient implementation of sorting on multi-core SIMD CPU architecture. Proceedings of the VLDB Endowment **1**(2) (2008) 1313–1324
18. Hoare, C.: Quicksort. The Computer Journal **5**(1) (1962) 10