Index-supported Similarity Join on Graphics Processors

Christian Böhm
University of Munich
boehm@dbs.ifi.lmu.de

Robert Noll
University of Munich
nollr@cip.ifi.lmu.de

Claudia Plant
Technische Universität München
plant@lrz.tum.de

Andrew Zherdin
Technische Universität München
andrew.zherdin@lrz.tum.de

Abstract: The similarity join is an important building block for similarity search and data mining algorithms. In this paper, we propose an algorithm for similarity join on Graphics Processing Units (GPUs). As major advantages GPUs provide extremely high parallelism combined with a high bandwidth in data transfer to main memory. To exploit these advantages for similarity join, we propose an index structure designed for the specific environment of GPU. Experiments demonstrate massive performance gains of our method over conventional similarity join on CPU and significant further speed-up by index support.

1 Introduction

In recent years, *Graphic Processing Units* (GPUs) have evolved from simple chips controlling the display device into powerful coprocessors supporting the CPU in various ways. Graphics applications such as realistic 3D games are computationally demanding and require a large number of complex algebraic operations for each update of the display image. Therefore, today's graphics hardware contains a large number (up to some hundreds) of simple but programmable processors which are optimized to cope with this high workload of vector, matrix and other computations in a highly parallel way.

In terms of peak performance, the highly parallel graphics hardware has outperformed state-of the art multi-core CPUs by a large margin. Therefore, there is a great effort in many research communities such as life sciences [LSVMW07, MV08] or data mining [CSK08] to use the computational capabilities of GPUs even for purposes which are not graphics-related at all. The corresponding research area is called General Processing-Graphical Processing Unit (GP-GPU). Vendors of graphics hardware have anticipated this trend and developed libraries, precompilers and application programming interfaces for this kind of programming. Most prominently, NVIDIA's technology *Compute Unified Device Architecture* (CUDA) offers a C++ programming interface in which both the *host program* to be executed on CPU as well as the so-called *kernel functions* to be executed in a massively parallel fashion on GPU are assembled in a single program [cud07]. An analogous technique is also offered by ATI under the brand names Close-to-Metal, Stream SDK, and Brook-GP.

The *similarity join* is a basic operation of a database system designed for similarity search and data mining on feature vectors. In such applications, we are given a large set D of objects which are associated with a vector from a multidimensional space, the feature space. The similarity join determines pairs of objects which are similar to each other. The most widespread form is the ϵ-join which determines those pairs from $D \times D$ which have a Euclidean distance of no more than a user-defined radius ϵ:

Definition 1 (Similarity Join) *Let $D \subseteq \mathbb{R}^d$ be a set of feature vectors of a d-dimensional vector space and $\epsilon \in \mathbb{R}^+$ be a threshold. Then the similarity join is the following set of pairs:*

$$\text{SimJoin}(D, \epsilon) = \{(p, q) \in (D \times D) \quad | \quad \text{dist}(p, q) \leq \epsilon\},$$

where $\text{dist}(p, q) = \sqrt{(p - q)^{\mathbf{T}} \cdot (p - q)}$ is the Euclidean distance between p and q.

If p and q are elements of the same set, the join is a similarity self-join. Most algorithms including the method proposed in this paper can also be generalized to the more general case of non-self-joins in a straightforward way. Algorithms for a similarity join with nearest neighbor predicates have also been proposed. The similarity join is a powerful building block for similarity search and data mining. It has been shown that important data mining methods such as clustering and classification can be based on the similarity join. Using a similarity join instead of single similarity queries can accelerate data mining algorithms by a high factor [BBBK00].

In this paper, we propose two algorithms for similarity join on the GPU. The first one is a parallelization of the Nested Loop Join (NLJ) with particular focus on the special demands of the graphics hardware. The second one is an indexed join algorithm which operates on an index structure which is particularly suited for the GPU. We demonstrate the superiority of our approach over the corresponding sequential join methods on CPU. The remainder of this paper is organized as follows: Section 2 quickly reviews the related work in similarity join processing and in GP-GPU processing in general. Section 3 explains the graphics hardware and the corresponding programming model. Section 4 and Section 5 are dedicated to the non-indexed and indexed NLJ on graphics hardware, respectively. Section 6 contains an extensive experimental evaluation of our technique, and Section 7 concludes this paper.

2 Related Work

For given multidimensional point data sets A and B similarity join operations can be classified depending on whether a multidimensional indexing structure exists on both data sets A and B [BKS93] or on neither [BBKK01, DS01]. There is a wide range of data structures that have been adopted for similarity join processing, quad-trees [DS01, KS00], R-trees [BKS93], ϵ-kbd trees [SSA97], hash functions [GIM99], and space-filling curves [DS01] to name a few. Two recent methods are predicated on the application of grids to multidimensional point data sets, namely the *Epsilon Grid Order* (EGO) [BBKK01] and the *Generic External Space Sweep* (GESS) method [DS01]. EGO is based on a particular

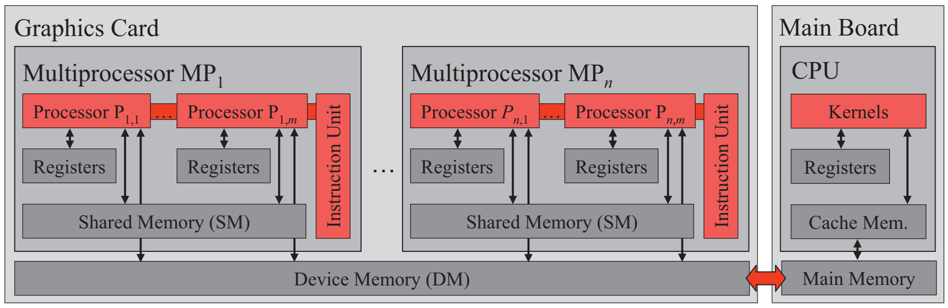

Figure 1: Architecture of a GPU.

sort order of the data points, which is obtained by laying an equidistant grid with cell length ϵ over the data space and comparing the grid cells lexicographically. By an external sorting algorithm and a particular scheduling strategy during the join phase, EGO avoids the problem of holding large portions of the data sets simultaneously in main memory. GESS constructs a hypercube with side length ϵ around each multidimensional input point. The similarity join problem is thus transformed to a spatial intersection [KS00] problem.

Some recent approaches, e.g. [GGKM06] and [GLW$^+$04] demonstrate that important building blocks for query processing in databases, e.g. sorting can be significantly speed up by the use of GPUs. In [HYF$^+$08] algorithms for relational join on GPU are presented. Probably the most related approach to ours is [LSS08], which is to the best of our knowledge the only paper on similarity join on GPU. The authors propose an algorithm based on the concept of space filling curves, e.g. the z-order, for pruning of the search space. The z-order of a set of objects can be determined very efficiently on GPU by highly parallelized sorting.

3 Architecture of the GPU

Graphic Processing Units (GPUs) of the newest generation are powerful coprocessors, not only designed for games and other graphic-intensive applications, but also for general-purpose computing (in this case, we call them GPGPUs). From the hardware perspective, a GPU consists of a number of multiprocessors, each of which consists of a set of simple processors which operate in a SIMD fashion, i.e. all processors of one multiprocessor execute in a synchronized way the same arithmetic or logic operation at the same time, potentially operating on different data. For instance, NVIDIA's GPU of the G80 series (e.g. the graphics card Geforce 8800 GTX) has 16 multiprocessors, each consisting of 8 processors, summarizing to a total amount of 128 processors inside one GPU. The architecture of the GPU is visualized in Figure 1: Apart from some memory units with special purpose in the context of graphics processing (e.g. texture memory), we have three important types of memory.

The *shared memory* (SM) is a memory unit with fast access (at the speed of register access, i.e. no delay). SM is shared among all processors of a multiprocessor. It can be used for local variables but also to exchange information between threads on different processors of the same multiprocessor. It cannot be used for information which is shared among threads on different multiprocessors. SM is fast but very limited in capacity (16 KBytes per multiprocessor). The second kind of memory is the so-called *device memory* (DM), which is the actual video RAM of the graphic card (also used for frame buffers etc.). DM is physically located on the graphics card (but not inside the GPU), is significantly larger than SM (typically up to some hundreds of MBytes), but also significantly slower. In particular, memory accesses to DM cause a typical latency delay of 400-600 clock cycles (G80), but the bandwidth for transferring data between DM and GPU (86 GB/s on G80) is higher than that of CPU and main memory (about 10 GB/s on current CPUs). DM can be used to share information between threads on different multiprocessors. If some threads schedule memory accesses from contiguous addresses, these accesses can be coalesced, i.e. taken together to improve the access speed. A typical cooperation pattern for DM and SM is to copy the required information from DM to SM simultaneously from different threads (if possible, considering coalesced accesses), then to let each thread compute the result on SM, and finally, to copy the result back to DM. The third kind of memory considered here is the *main memory* which is not part of the graphics card. The GPU has no access to the address space of the CPU. The CPU can only write to or read from DM using specialized API functions. The bottleneck of these operations is, of course, larger than DM accesses of the GPU.

The basis of the programming model of GPUs are threads. Threads are lightweight processes which are easy to create and to synchronize. In contrast to CPU processes, a context switch between different threads does not cause any considerable overhead either. In typical applications, thousands or even millions of threads are created, for instance one thread per pixel in gaming applications. The CUDA programming library [cud07] contains API functions to create a large number of threads on the GPU, each of which executes a function called *kernel function*. The kernel functions (which are executed in parallel on the GPU) as well as the host program (which is executed sequentially on the CPU) are defined in an extended C++ syntax. The kernel functions are restricted with respect to functionality (e.g. no recursion).

On GPUs the threads do not even have an individual instruction pointer. An instruction pointer is rather shared by several threads. For this purpose, threads are grouped into so-called *warps* (typically 32 threads per warp). One warp is processed simultaneously on the 8 processors of a single multiprocessor (SIMD) using 4-fold pipelining (totalling in 32 threads executed synchronously). To allow 4-fold pipelining, each processor contains also four arithmetic logic units (ALUs). If not all threads in a warp follow the same execution path, the different execution paths are executed in a serialized way.

Multiple warps are grouped into *thread groups* (TG). It is recommended [cud07] to use multiples of 64 threads per TG. The different warps in a TG (as well as different warps of different TGs) are executed independently. The threads in one thread group use the same shared memory and may thus communicate via the SM. The threads in one thread group can be synchronized (let all threads wait until all warps of the same group have

```
algorithm sequentialNLJ(data set $D$)
  for each $q \in D$ do        // outer loop
    for each $p \in D$ do      // inner loop: search all points $p$ which are similar to $q$
      if dist$(p, q) \leq \epsilon$ then
        report $(p, q)$ as a result pair or do some further processing on $(p, q)$
end
```

Figure 2: Sequential Algorithm for the Nested Loop Join.

reached that point). The latency delay of the DM can be hidden by scheduling other warps of the same or a different thread group whenever one warp waits for an access to DM. To allow switching between warps of different thread groups on a multiprocessor, it is recommended [cud07] that each thread uses only a small fraction of the shared memory and registers of the multiprocessor.

4 Similarity Join Without Index Support

The baseline technique to process any join operation with an arbitrary join predicate is the nested loop join (NLJ) which performs two nested loops, each enumerating all points of the data set. For each pair of points, the distance is calculated and compared to ϵ. The pseudocode of the sequential version of NLJ is given in Figure 2.

It is easily possible to parallelize the NLJ, e.g. by creating an individual thread for each iteration of the outer loop. The kernel function then contains the inner loop, the distance calculation and the comparison. During the complete run of the kernel function, the current point of the outer loop is constant, and we call this point the *query point* q of the thread, because the thread operates like a similarity query, in which all database points with a distance of no more than ϵ from q are searched. The query point q is always held in a register of the processor.

Our GPU allows a truly parallel execution of a number p of incarnations of the outer loop, where p is the total number of ALUs of all multiprocessors (i.e. the warp size 32 times the number of multiprocessors). Moreover, all the different warps are processed in a quasi-parallel fashion, which allows to operate on one warp of threads (which is ready-to-run) while another warp is blocked due to the latency delay of a DM access of one of its threads.

The threads are grouped into thread groups, which share the SM. In our case, the SM is particularly used to physically store for each thread group the current point p of the inner loop. Therefore, a kernel function first copies the current point p from the DM into the SM, and then determines the distance of p to the query point q. The threads of the same warp are running perfectly simultaneously, i.e. if these threads are copying the same point from DM to SM, this needs to be done only once (but all threads of the warp have to wait until this relatively costly copy operation is performed). However, a thread group may (and should) consist of multiple warps. To ensure that the copy operation is only performed once per thread group, it is necessary to synchronize the threads of the thread group before and after the copy operation using the API function synchronize(). This API function blocks all threads in the same TG until all other threads (of other warps) have

```
algorithm GPUsimpleNLJ(data set $D$)      // host program executed on CPU
  deviceMem float $D'[][] := D[][]$;      // allocate memory in DM for the data set $D$
  #threads := $|D|$;      // number of points in $D$
  #threadsPerGroup := 64;
  startThreads (simpleNLJKernel, #threads, #threadsPerGroup);      // one thread per point
  waitForThreadsToFinish();
end.

kernel simpleNLJKernel (int threadid)
  register float $q[] := D'$[threadid][] ;      // copy the point from DM into the register
                                                 // and use it as query point $q$
                                                 // index is determined by the thread-id
  for $i := 0 ... n-1$ do      // this used to be the inner loop in Figure 2
    synchronizeThreadGroup();
    shared float $p[] := D'[i][]$;      // copy the current point $p$ from DM to SM
    synchronizeThreadGroup();      // Now all threads of the thread group can work with $p$
    if dist$(p, q) \le \epsilon$ then
      report $(p, q)$ as a result pair using synchronized writing
      or do some further processing on $(p, q)$ directly in kernel
end.
```

Figure 3: Parallel Algorithm for the Nested Loop Join on the GPU.

reached the same point of execution. The pseudocode for this algorithm is presented in Figure 3.

If the data set does not fit into DM, a simple partitioning strategy can be applied. It must be ensured that the potential join partners of an object are within the same partition as the object itself. Therefore, overlapping partitions of size $2 \cdot \epsilon$ can be created.

5 An Index Structure to Support the Similarity Join on GPU

The performance of the NLJ can be greatly improved if an index structure is available. On sequential processing architectures, the indexed NLJ leaves the outer loop unchanged. The inner loop is replaced by an index-based search retrieving candidates that may be join partners of the current object of the outer loop. The effort of finding these candidates and refining them is often orders of magnitude smaller compared to the non-indexed NLJ.

When parallelizing the indexed NLJ for the GPU, we follow the same paradigm as in the last section, to create an individual thread for each point of the outer loop. It is beneficial to the performance, if points having a small distance to each other are collected in the same warp and thread group, because for those points, similar paths in the index structure are relevant. We will see later how this grouping of similar points is easily achieved.

Our index structure needs to be traversed in parallel for many search objects using the kernel function. Since kernel functions do not allow any recursion, and as they need to have small storage overhead by local variables etc., the index structure must be very simple as well. To achieve a good compromise between simplicity and selectivity of the index, we propose a data partitioning method with a constant number of directory levels. The first level partitions according to the first dimension of the data space, the second level according to the second dimension, and so on. Therefore, before performing the join,

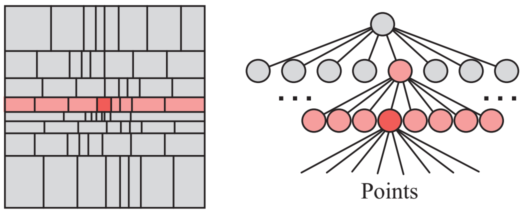

Figure 4: Index Structure for GPU.

some transformation technique should be applied which guarantees a high selectivity in the first dimensions (e.g. Principle Component Analysis, Fast Fourier Transform, Discrete Wavelet Transform, etc.). Figure 4 shows a simple, 2-dimensional example of a 2-level directory (plus the root node which is considered as level-0), similar to [KHT89, LEL97]. The fanout of each node is 8. In our experiments in Section 6, we used a three-level directory with fanout 16. Our algorithm now changes only a little bit. Before starting the actual join processing, our simple index structure must be constructed in a bottom-up way by fractionated sorting of the data: First, the data set is sorted according to the first dimension, and partitioned into the specified number of quantile partitions. Then, each of the partitions is sorted individually according to the second dimension, and so on. The boundaries are stored using simple arrays which can be easily accessed in the subsequent kernel functions. In principle, this index construction can already be done on the GPU, because efficient sorting methods for GPU have been proposed [GGKM06]. Since bottom up index construction is typically not very costly compared to the join algorithm, our method performs this preprocessing step on CPU. After index construction, we have not only a directory in which the points are organized in a way that facilitates search. Moreover, the points are now clustered in the array, i.e. points which have neighboring addresses are also likely to be close together in the data space (at least when projecting on the first few dimensions). Both effects are exploited by our join algorithm displayed in Figure 5.

Instead of performing an outer loop like in a sequential indexed NLJ, our algorithm now generates a large number of threads: One thread for each iteration of the outer loop (i.e. for each query point q). Since the points in the array are clustered, the corresponding query points are close to each other, and the join partners of all query points in a thread group are likely to reside in the same branches of the index as well. Our kernel method now iterates over three loops, each loop for one index level, and determines for each partition if the point is inside the partition or, at least no more distant to its boundary than ϵ. The corresponding subnode is accessed if the corresponding partition is able to contain join partners of the current point of the thread. When considering the warps which operate in a fully synchronized way, a node is accessed, whenever at least one of the query points of the warps is close enough to (or inside) the corresponding partition.

For both methods, indexed and non-indexed nested loop join on GPU, we need to address the question how the resulting pairs are processed. The most general technique is to report the results to the CPU where they can be further processed and permanently stored if

```
algorithm GPUindexedJoin(data set $D$)
  deviceMem index idx := makeIndexAndSortData($D$); // changes ordering of datapoints
  int #threads := $|D|$, #threadsPerGroup := 64;
  for $i = 1$ ... (#threads/#threadsPerGroup) do
    deviceMem float blockbounds[$i$][] := calcBlockBounds($D$, blockindex);
  deviceMem float $D'[][] := D[][]$;
  startThreads (indexedJoinKernel, #threads, #threadsPerGroup); // one thread per datapoint
  waitForThreadsToFinish $()$;
end.

algorithm indexedJoinKernel (int threadid, int blockid)
  register float $q[] := D'$[threadid][] ;        // copy the point from DM into the register
  shared float myblockbounds[] := blockbounds[blockid][];
  for $x_i := 0$ ... indexsize.$x$ do
    if IndexPageIntersectsBoundsDim1(idx,myblockbounds,$x_i$) then
      for $y_i := 0$ ... indexsize.$y$ do
        if IndexPageIntersectsBoundsDim2(idx,myblockbounds,$x_i, y_i$) then
          for $z_i := 0$ ... indexsize.$z$ do
            if IndexPageIntersectsBoundsDim3(idx,myblockbounds,$x_i, y_i, z_i$) then
              for $w := 0$ ... IndexPageSize do
                synchronizeThreadGroup();
                shared float $p[]$ :=GetPointFromIndexPage(idx,$D', x_i, y_i, z_i, w$);
                synchronizeThreadGroup();
                if dist$(p, q) \leq \epsilon$ then
                  report $(p, q)$ as a result pair using synchronized writing
end.
```

Figure 5: Algorithm for Similarity Join on GPU with Index Support.

required. This is easily possible by a buffer in DM which can be copied to the CPU after the termination of all kernel threads. The result pairs must be written into this buffer in a synchronized way to avoid that two threads write simultaneously to the same buffer area. The CUDA API provides atomic operations (such as atomic increment of a buffer pointer) to guarantee this kind of synchronized writing. Buffer overflows are also handled by our similarity join methods. If the buffer is full, all threads terminate and the work is resumed after the buffer is emptied by the CPU.

6 Experimental Evaluation

To evaluate the performance of similarity join on GPU, we compare four different variants for performing similarity join: (1) nested loop join on CPU, (2) NLJ on CPU with index support, (3) NLJ on GPU and (4) NLJ on GPU with index support. All variants have been implemented in C++. The index structure described in Section 5 was used in variant (2) and (4). The experiments have been performed on a workstation with Intel Core 2 Duo CPU E4500 2.2 GHz and 2 GB RAM which is supplied with a NVIDIA 8500GT GPU (2 multiprocessors each consisting of 8 processors) with 512MB VRAM.

We generated three 8-dimensional synthetic data sets of various sizes (up to 10 million points) with different data distributions: Data set DS1 contains uniformly distributed data. DS2 consists of 5 Gaussian clusters which are randomly distributed in feature space. Similar to DS2, DS3 is also composed of 5 Gaussian clusters, but the clusters are correlated. The threshold ϵ was selected to obtain a join result where each point was combined with one or two join partners on average.

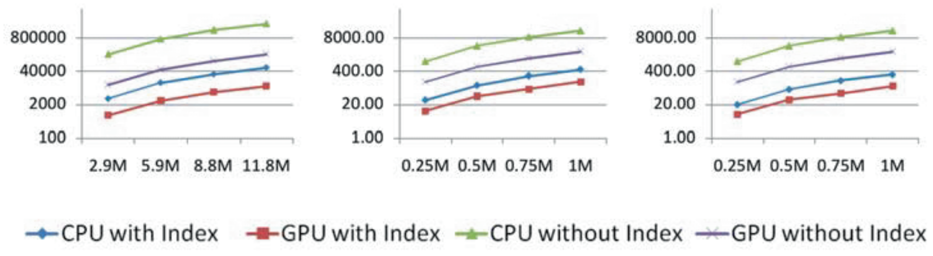

Figure 6: Experimental Results: Runtime in seconds on different data volumes ranging from 250,000 (0.25 M) to 11.8 (11.8 M) million points.

Figure 6 displays the runtime in seconds of all four variants for similarity join in logarithmic scale. The time needed for data transfer from CPU to GPU and back as well as the (negligible) index construction time has been included. NLJ on GPU with index support performs best in all experiments. Note that, due to massive parallelization, NLJ on GPU without index support outperforms CPU without index by a large factor (15.9 on 180 MByte of uniform data). The GPU algorithm with index support outperforms the corresponding CPU algorithm (with index) by a factor of 4.6. The overall improvement of the indexed GPU algorithm over the non-indexed CPU version is 246. This result demonstrates the potential of boosting performance of database operations with designing specialized index structures and algorithms for GPU. Further experiments (varying dimensionality, join selectivity (ϵ) and fanout and number of levels of the index) were conducted but must be left out due to space limitations. The speedup of the GPU versions over the corresponding CPU versions was always in the same order of magnitude like in the experiments shown here.

7 Conclusions

In this paper, we have addressed the question, how similarity join algorithms can be efficiently executed on Graphics Processing Units (GPU). We have proposed an index structure which is particularly suited to be held in the Device Memory of the GPU and which can be accessed by kernel functions of the GPU in an efficient way. Since graphics processors correspond to a highly parallel architecture consisting of up to some hundreds of single programmable processors, we have also shown how similarity join algorithms can be parallelized. We have proposed two efficient similarity join algorithms, one based on the nested loop join and the other based on the indexed nested loop join. Both algorithms are particularly dedicated for GPU processing and outperform the sequential algorithms by a large factor.

References

[BBBK00] Christian Böhm, Bernhard Braunmüller, Markus M. Breunig, and Hans-Peter Kriegel. High Performance Clustering Based on the Similarity Join. In *CIKM*, pages 298–305, 2000.

[BBKK01] Christian Böhm, Bernhard Braunmüller, Florian Krebs, and Hans-Peter Kriegel. Epsilon Grid Order: An Algorithm for the Similarity Join on Massive High-Dimensional Data. In *SIGMOD*, pages 379–388, 2001.

[BKS93] Thomas Brinkhoff, Hans-Peter Kriegel, and Bernhard Seeger. Efficient Processing of Spatial Joins Using R-Trees. In *SIGMOD*, pages 237–246, 1993.

[CSK08] Bryan Catanzaro, Narayan Sundaram, and Kurt Keutzer. Fast Support Vector Machine Training and Classification on Graphics Processors. In *ICML 2008*, pages 104–111, 2008.

[cud07] *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[DS01] Jens-Peter Dittrich and Bernhard Seeger. GESS: a scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In *KDD*, pages 47–56, 2001.

[GGKM06] Naga K. Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. GPUTeraSort: high performance graphics co-processor sorting for large database management. In *SIGMOD*, pages 325–336, 2006.

[GIM99] Aristides Gionis, Piotr Indyk, and Rajeev Motwani. Similarity Search in High Dimensions via Hashing. In *VLDB*, pages 518–529, 1999.

[GLW$^+$04] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming C. Lin, and Dinesh Manocha. Fast Computation of Database Operations using Graphics Processors. In *SIGMOD*, pages 215–226, 2004.

[HYF$^+$08] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational joins on graphics processors. In *SIGMOD*, pages 511–524, 2008.

[KHT89] Masaru Kitsuregawa, Lilian Harada, and Mikio Takagi. Join Strategies on KD-Tree Indexed Relations. In *ICDE*, pages 85–93, 1989.

[KS00] Nick Koudas and Kenneth C. Sevcik. High Dimensional Similarity Joins: Algorithms and Performance Evaluation. *IEEE Trans. Knowl. Data Eng.*, 12(1):3–18, 2000.

[LEL97] Scott T. Leutenegger, J. M. Edgington, and Mario A. Lopez. STR: A Simple and Efficient Algorithm for R-Tree Packing. In *ICDE*, pages 497–506, 1997.

[LSS08] Michael D. Lieberman, Jagan Sankaranarayanan, and Hanan Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *ICDE*, pages 1111–1120, 2008.

[LSVMW07] Weiguo Liu, Bertil Schmidt, Gerrit Voss, and Wolfgang Müller-Wittig. Molecular Dynamics Simulations on Commodity GPUs with CUDA. In *HiPC*, pages 185–196, 2007.

[MV08] Svetlin Manavski and Giorgio Valle. CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment. *BMC Bioinformatics*, 9(Suppl 2), 2008.

[SSA97] Kyuseok Shim, Ramakrishnan Srikant, and Rakesh Agrawal. High-Dimensional Similarity Joins. In *ICDE*, pages 301–311, 1997.