X-Device Query Processing by Bitwise Distribution

Holger Pirk	Thibault Sellam	Stefan Manegold	Martin Kersten
CWI, Amsterdam	CWI, Amsterdam	CWI, Amsterdam	CWI, Amsterdam
The Netherlands	The Netherlands	The Netherlands	The Netherlands
holger@cwi.nl	sellam@cwi.nl	manegold@cwi.nl	mk@cwi.nl

ABSTRACT

The diversity of hardware components within a single system calls for strategies for efficient cross-device data processing. For example, existing approaches to CPU/GPU co-processing distribute individual relational operators to the "most appropriate" device. While pleasantly simple, this strategy has a number of problems: it may leave the "inappropriate" devices idle while overloading the "appropriate" device and putting a high pressure on the PCI bus. To address these issues we distribute data among the devices by partially decomposing relations at the granularity of individual bits. Each of the resulting bit-partitions is stored and processed on one of the available devices. Using this strategy, we implemented a processor for spatial range queries that makes efficient use of all available devices. The performance gains achieved indicate that bitwise distribution makes a good cross-device processing strategy.

1. INTRODUCTION

Computer systems have ceased to be centralized systems in which a CPU controls dumb storage devices. Special purpose extension cards that can support the CPU, in particular General Purpose Graphics Processing Units (GPGPUs), are available at low prices. The design and use of these cards, however, is fundamentally different from the one of the CPU. Fast sequential execution based on behavior prediction (pipelining, prefetching, branch prediction, ...) is replaced by simple, yet massively parallel, execution. The internal memory of these devices is usually orders of magnitude faster but offers much smaller storage capacity than traditional memory and lacks the benefits of virtual addressing. While this can make their use difficult, it can also provide opportunities for significant gains in application performance. The high bandwidth and compute power of have aroused interest of data management researchers [2, 6, 14, 5, 15, 13, 6]. While computation-intensive applications are a natural fit for the massively parallelized architecture of GPGPUs, the, rarely computation intensive, processing of relational queries can not benefit from the available compute resources to the extend possible. However, data intensive applications like relational query processing can still benefit from the fast memory of GPUs if implemented appropriately.

Proceedings of the Eighth International Workshop on Data Management on New Hardware (DaMoN 2012), May 21, 2012, Scottsdale, AZ, USA.

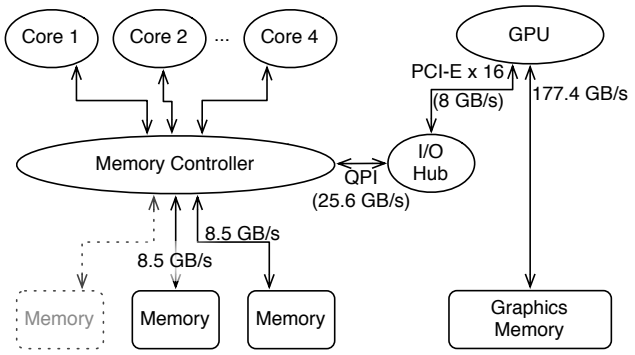
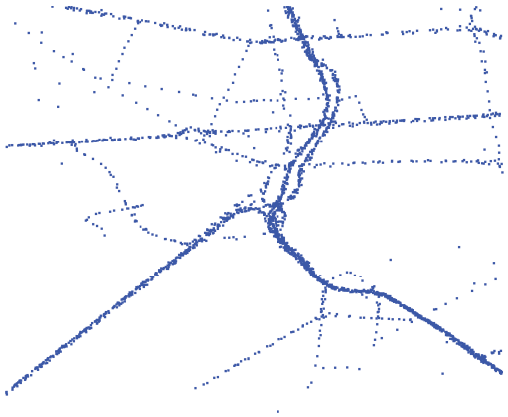


Figure 1: The Architecture of a typical GPU/CPU System

Unfortunately, the relatively small memory capacity and the lack of virtual memory management complicate the efficient management of large database on GPGPUs. The current state-of-the-art is to ship (parts of the) data to the GPU, process individual relational operations and transfer the results back to the main CPU [14]. As illustrated in Figure 1, the PCI-Bus has more than an order of magnitude lower throughput than the internal device memory. Continuous transfer of data through this bus can become a major bottleneck for CPU/GPU co-processing [11]. While the expensive cross-device transfer can be avoided in some cases, the costs for the necessary streaming of large datasets can negate the cost improvement achieved by the GPU processing. To mitigate this problem to some extend, data compression can help to reduce the data volume in the limited GPU memory [7]. However, lossless data compression may fail to achieve the necessary savings or hurt the performance due to the decompression costs. In particular the massively parallelized architecture of GPUs limits the arsenal of efficient compression methods. An alternative to reduce the data volume is the use of sampling techniques [19]. While samples are a good means to reduce the data size, they are merely useful to gain a rough overview of the available dataset. To gain precise results, sampling is not a suitable technique.

To efficiently exploit the superior bandwidth of the GPU's internal memory we propose to re-evaluate the techniques that helped reducing the footprint of main memory resident databases: the *Decomposed Storage Model (DSM)* [4] reduced the footprint to the columns that are actively used for query evaluation, i.e., the *hot* columns. On top of that, lightweight compression of the values within a column proved effective [20]. However, to achieve the necessary compression, decomposing tuples into columns of scalar values may not be enough. To limit the data volume to the capac-

9(a) Traffic in Western Berlin



9(b) Drill-Down to Dreieck Funkturm (A Traffic Hot Zone)

Figure 2: Spatial Drill-Down

ity of the respective device memory we propose to take data decomposition to the next level: *Bitwise Decomposition*. Each of the resulting bit-partitions is stored (non-redundantly) and processed on the most appropriate device. The data volume on each device can be controlled by varying the number of bits that are stored in the device memory. Combined with lightweight compression this promises GPU-supported data processing without the need for expensive cross device data transfers.

To this end, we make the following contributions:

- We introduce the idea of Bitwise Distribution of relational data across processing devices.
- We develop a model to determine the optimal distribution strategy.
- We evaluate Bitwise Distribution in combination with Frame Of Reference/Delta compression for range queries on real-life spatial trajectory data.

The remainder of this paper is organized as follows. In Section 2 we present our use case, and provide an overview of the benefits and challenges that come with GPGPU programming. In Section 3 we describe the data distribution strategy, its implementation and the calculation of the necessary tuning parameters. The query processing on top of the distributed data is described in Section 4. In Section 5, we evaluate our approach and conclude in Section 6.

2. BACKGROUND

To introduce the problem, we provide a brief overview of established spatial data indexing techniques, and how our work differs from these. Following that, the boundary conditions of GPGPU processing as well as the resulting opportunities and challenges will be discussed.

2.1 Spatial Data Management

The use case that motivates this work is the need for quick retrieval of historical GPS data in a traffic analysis context. A database stores several years of vehicle movements on the European road network to support applications such as traffic forecast or infrastructure monitoring. Figure 2 illustrates the monitoring of traffic in the city of Berlin. Reporting is generally focused on certain hot zones of traffic. This results in spatial window (i.e., 2 dimensional range) selection queries. The results of these queries can either be displayed directly or post-processed by more sophisticated applications. These include data mining or decision support applications. Since the data as well as the queries are two-dimensional, classical index-structures do not support the application well.

There has been substantial work on spatial data management, especially on spatial indexing methods. The general idea is to cluster the data according to their geographical proximity in order to improve I/O performance. Many data structures have been proposed. Among those, the R-Tree [12] is ubiquitous. It encloses spatial objects in bounding boxes. More precisely, a node represents the smallest possible bounding box that can cover its child nodes. The major drawback of this data structure is that the entries might overlap and contain empty space, which degrades worse case lookups performance. Many improvements were proposed. For instance, R^+ trees [12] follow similar principles, but do not allow bounding boxes to overlap. This allows faster traversals, at the cost of higher construction and maintenance costs. Another family of methods relies on spatial grids to index the data. For instance, a uniform grid is proposed in [8]. However, such a grid is not well-suited to non uniform distributions. The quad-tree is a popular alternative. Each node represent one of four quadrants of its parent node [18]. This allows adaptive data storage. Finally, the data may be adapted to traditional single dimension data structures by applying dimensionality reduction techniques such as space filling curves. For instance, Z-ordering [9] relies on bit interleaving, which preserves some spatial locality.

Our aim is to support a large volume of simultaneous queries: we target bandwidth rather than latency. Therefore, our approach to retrieving spatial data is orthogonal. It relates mainly to work carried out on bulk data processing, illustrated by systems such as MonetDB [3]. There are two main differences with the previously presented data structures. First, we assume that current hardware provides sufficient throughput to scan a complete dataset efficiently, while the branches induced by traditional index structures are ill-suited for massively parallel processing. Second, we rely on a bit-level decomposition of the data. There is to our knowledge little to no work using it for multidimensional data storage and retrieval.

2.2 GPGPU Programming

The programming of a GPU is very different from the programming of a CPU. This is largely due to the fundamentally different architecture. To achieve high compute power at low costs, GPUs rely on a parallelism paradigm called *Single Instruction Multiple Threads* (SIMT).

Single Instruction Multiple Threads

The notion of SIMT is a peculiarity of GPU hardware and the source of a common misconception of GPU programming. Even though a GPU supports many parallel threads, these are not independent as they are on a CPU. All cores of a processor execute the same instruction at any given cycle. An ATI Evergreen-class GPU, e.g, has 16 SIMT-cores which execute every instruction for at least 4 cycles. Thus, every scheduled instruction is executed at least 64 times. Each core does, however, have its own registers and usually operates on different data items than the other cores. A set of threads coupled like that is called a *Work Group*. A work group of less than 64 items underutilizes the GPU's computation units. This also has a severe impact on branching: if one branch in a Work Group diverges from the others the branches are serialized and executed on all cores. The cores that execute a branch involuntarily simply do not write the results of their operations.

The Programming Model

Programming the high number of SIMT-cores of a GPU in an imperative language with explicit multithreading is a challenging task. To simplify GPU programming, a number of competing technologies based on the kernel programming model have been introduced. The most prominent ones are: DirectCompute, CUDA [17] and OpenCL [16]. While the earlier two are proprietary technologies, the later is an open standard that is supported by many hardware vendors on all major software platforms. The supported hardware does not just include GPUs, but CPUs as well: *Intel* and *AMD* provide implementations for their CPUs, *AMD* and *NVidia* for GPUs. *Apple*, one of the driving forces behind OpenCL, ships their current OS version with an OpenCL implementation for both GPUs and CPUs. The portability does, however, come at a price: to support a variety of devices, OpenCL resorts to the least common denominator, which radically limits the programming model.

The most important concept in OpenCL is the Kernel: a function that is defined by its (OpenCL) C-code, compiled at runtime on the Host, transferred in binary representation and executed on the device. The Kernel is then scheduled with a specified problem size (essentially the number of times the kernel is run) to operate on a number of data buffers.

A Priory Fixed Problem Size

This is done by dispatching the kernel (the compiled processing function) for execution on the device with the problem size as a parameter (e.g., n). The kernel is then executed exactly n times. Each invocation has access to an id that can be used to determine which piece of the work to do. If the problem size is not known apriori, a workaround is to use a single complex thread to do all the work. Naturally, reducing the degree of parallelism on a GPU has a negative impact on performance. Another solution is to specify an upper bound on the problem size and abandon the execution of some of the kernels at runtime. Since no new kernel can be scheduled to a core until the work group has finished, this may also lead to underuse of the available compute power.

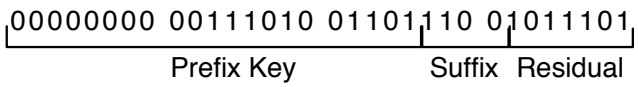


Figure 3: Bitwise Decomposition of Spatial Values

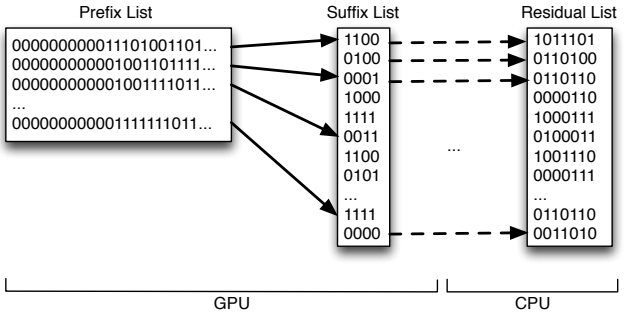


Figure 4: Bitwise Distribution of Spatial Values

Static Memory Allocation

The OpenCL programming model does not allow dynamic (re)allocation of memory. Similar to the problem size, input and output memory sizes have to be specified up front. While this is a problem for operations with a large upper bound on the output size (e.g., joins), for selection queries an overallocation of output and an overflow check at runtime mitigates this problem. Even though the lack of memory reallocation is a problem it also has a significant performance advantage. Memory can be addressed using physical addresses, which eliminates the need for costly translation from virtual to physical addresses and speeds up memory access.

3. DATA LOADING

In this section we introduce how the GIS data is prepared for processing. We introduce two splits at the bit level in order to spread data between the devices. However, determining where to split is non trivial. A model of GPGPU memory consumption is presented, as well as our clustering algorithm.

3.1 Bitwise distribution

Thanks to their high bandwidth, GPUs can evaluate queries very efficiently. However, their limited memory capacity complicates their effective use on large datasets. To resolve this problem, the GPU can be used to generate an (over)approximate answer to the query. The CPU can be used, subsequently, for result set refinement and tuple reconstruction. Figure 3 illustrates a data partitioning scheme to support such GPU/CPU co-processing. The GPU holds the most selective components of the data while the main memory holds the bits necessary to restore the original values (*residuals*). In our use case, we choose the most significant bits of the coordinates in each dimension. The data in the GPU memory is essentially a reduced-resolution representation of the original data.

To minimize the number of false positives, the query dependent selectivity of the GPU resident data has to be maximized under the given GPU memory limitations. This is achieved through a simple form of prefix compression: the GPU-resident bits are split into a prefix and a suffix. The suffixes of all values with a common prefix are physically clustered and stored in a suffix list. The prefixes, together with the respective offset into the suffix list are stored in a prefix list (Figure 4). We expect that in most spatial datasets,

Denotation	Description
p	Prefix size (bits)
s	Suffix size (bits)
$i \in \{X,Y\}$	Dimension
D_i	Size of coordinate i (bits)
D_m	Smallest value of D_i (bits)
O	Size of a pointer to a cluster (bits)
H_i, L_i	Highest, lowest value on i
MAX	Shared memory of the GPU
N	Number of items
$P_i(p)$	Clusters in dimension i
$S(p,s)$	Space required for (p,s)

Figure 5: Parameters and variables of the space usage model

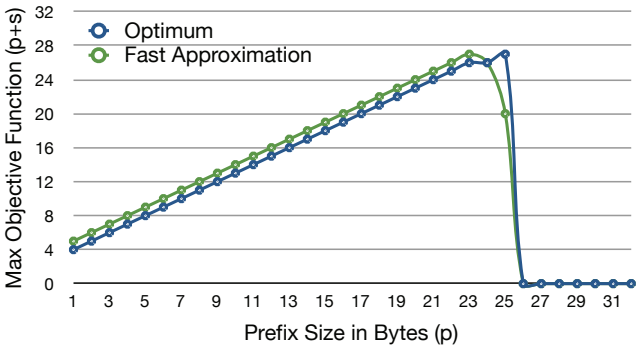


Figure 6: Optimal splits for varying k

the prefix bits of coordinates offer less variations than the suffixes. This allows efficient physical clustering and subsequent compression. For instance, the integer representation of a standard positive latitude with five decimals (up to $90x10^5$, the comma is implicit) always contains a null byte prefix.

3.2 Bitwise Decomposition

Finding a good split between *prefixes*, *suffixes* and *residuals* given a GPU memory capacity is a non-trivial problem. This section presents a model of the memory consumption for a given decomposition. We apply bounded search over the candidate solutions yielded my the model in order to find an optimal decomposition.

We apply our model to the two dimensions (longitude, latitude) of our use case, represented by $i \in \{X,Y\}$. Generalizing it to higher dimensionality is straight forward. p is the size in bits of the prefixes, s the size of the suffixes, D_i the number of bits necessary to represent one coordinate (D_i = 32 bits for an integer). For each dimension, H_i and L_i represent respectively the highest and the lowest values in the dataset. N is the total number of items. The memory capacity of the GPU is MAX. These parameters are summarized in Figure 5.

Figure 6 shows the maximal value of the objective function that can be achieved varying p with our use-case dataset and hardware. Increasing the prefix size allows better compression. Then, more bits can be stored on GPU, and less on the CPU. However, any value of p greater than 25 yields clusters that are too large to fit in GPU memory. This point is precisely what is targeted.

The range of values that can be covered by a prefix p in one dimension is 2^{D_i-p}. Therefore, the total number of prefixes necessary to represent all the values in a dimension is:

$$P_i(p) = \lfloor \frac{H_i}{2^{D_i-p}} \rfloor - \lfloor \frac{L_i}{2^{D_i-p}} \rfloor + 1$$

As a result, the total number of clusters in two dimensions given the size of the prefixes is $P_X(p)P_Y(p)$. For each cluster, the GPU stores the prefix p and a pointer to the suffix list. The size (in bits) of the pointer is O. Inside the clusters, each of the N points is represented with s bits for each dimension. Therefore, with two dimensions, the total storage footprint associated with a partition scheme (p,s) can be expressed as:

$$S(p,s) = 2(p+O)K_X(p)K_Y(p) + 2Ns$$

The objective is to store as much information about the data as possible on the GPU. Under these assumptions, the partitioning strategy research may be expressed as a constrained optimization problem:

$$\begin{aligned} \underset{p,s}{\text{Max}} \quad & p+s && (o) \\ s.t. \quad & S(p,s) \leq MAX && (c1) \\ & 0 \leq p+s \leq L_X && (c2) \\ & 0 \leq p+s \leq L_Y && (c3) \\ & p,s \in \mathbb{N} && (c4) \end{aligned}$$

A bounded search in the solution space is a simple and efficient way to reach an optimal solution. The algorithm in Figure 7 illustrates the procedure. The worse case complexity of the algorithm (i.e., size of the search space) is $O(\frac{D_m(D_m-1)}{2})$ with $D_m = \min_{i \in \{X,Y\}} D_i$. However, it seems reasonable to assume low minimum values (L_i) for most data sets. Also, during the exploration, if a candidate solution (p,s) violates $(c1)$ then any subsequent improvement of p or s can be discarded. This allows pruning of the search space.

An acceptable approximation of the optimum may quickly be obtained by assuming that constraint $(c1)$ is always met. This is applicable to any dataset wider than the GPU capacity. In this case, the optimization problem becomes:

$$\begin{aligned} \underset{p,s}{\text{Max}} \quad & p+s && (o') \\ s.t. \quad & (p+O)K_X(p)K_Y(p) + Ns = MAX/2 && (c1') \\ & 0 \leq p+s \leq \min_{i \in \{X,Y\}} D_i && (c2') \\ & p \in \mathbb{N}, s \in \mathbb{R} && (c3') \end{aligned}$$

Equation $(c1')$ yields an approximation of the optimal suffix size given a prefix:

$$\tilde{s}(p) = \lfloor \frac{MAX}{2N} - \frac{p+O}{N} K_X(k)K_Y(k) \rfloor$$

Thereby, the whole optimization system may be approximated as follows:

$$\begin{aligned} \underset{p \in \mathbb{N}}{Max} \quad & p + \tilde{s}(p) \\ s.t. \quad & p + \tilde{s}(p) \leq \min_{i \in \{X,Y\}} D_i \end{aligned}$$

Such result can be obtained in $O(\min_{i \in \{X,Y\}} D_i)$ by enumerating the possible values of p.

```
p_max ← 0
s_max ← 0
for p = 1 → D_m do
    for s = 1 → D_m − k do
        size ← S(p,s)
        if size > MAX then
            break
        end if
        if p + s > p_max + s_max then
            p_max ← p
            s_max ← s
        end if
    end for
end for
```

Figure 7: Optimal split lookup algorithm

3.3 Implementation

Once the optimal parameters are determined, the data can be decomposed accordingly. This is done in two phases. In the first phase, the data is scanned to build the prefix list (see Figure 3) according the number of prefix bits k. This step is similar to the determination of the size of the clusters of a radix clustering. The offsets into the suffix list are generated by prefix-summing the cluster sizes in the prefix list.

In the second phase, the data is scanned again to fill the suffix and residual list. This is equivalent to the clustering step of a radix clustering. After this phase, the data is in bitwise decomposed representation. The cluster and delta lists are transferred to the GPU memory and `free`d in the system's main memory, while the residual list is kept in the main memory. Therefore the data is distributed over the available devices.

4. QUERY PROCESSING

The decomposed distribution of the stored tuples largely determines the query processing strategy. The query is evaluated in phases with every device being responsible for one phase of the query evaluation. In each phase, the device does the best with the data it has available: narrowing down to the final result as much as possible and (partially) reconstructing the tuple values. In a (quite common) CPU/GPU co-processing setup, the processing is done in two phases: *GPU Preselection* and *CPU Refinement*. Since the result of each phase is a (potentially inaccurate) representation of tuples in the database, the two steps can be implemented like operators in a relational DBMS. Due to the high overhead when transferring data across devices, the Volcano-model [10] is not well suited to connect these operators. We, thus, implement them in the bulk processing model: in each phase the intermediates are materialized into the device's memory and copied once the processing phase has completed. When handling continuous query streams, the two evaluation phases of different queries can be interleaved, keeping all devices busy.

4.1 Phase 1: GPU Preselection

In the first phase, the GPU performs what can be considered a prefiltering of the dataset. Since the GPU memory only contains an approximate representation of the data (it misses the residuals), it cannot give an exact answer to the query. Instead it does a best-effort filtering of the tuples and returns partial results (see Figure 8). The partial result set is a superset of the exact answer to the query, but contains all the information for the CPU to narrow it down to the exact result set.

```
typedef struct {
        short queryID;
        int partialX, partialY;
        int tupleID;
} PartialResult;
```

Figure 8: Partially Reconstructed Tuple

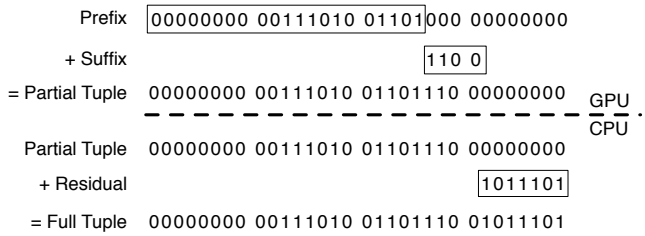


Figure 9: Cross-Device Tuple Reconstruction

Parallelization

The high degree of processing parallelism in GPUs calls for an equally high degree of parallelism in the query processor implementation. To achieve this, we parallelize the query evaluation in two dimensions: the queries and the data clusters. Every combination of query and cluster is assigned to one thread. While the number of queries is moderate (hundreds), the number of clusters (as determined by k) is usually high (in the range of tens of thousands). This results in a high degree of parallelism that can be used for efficient GPU data processing. However, the work items are grouped into work groups without optimization. Due to the SIMT processing model, the processing time spent on a cluster is determined by the size of the largest cluster in the work group. To somewhat mitigate this problem for highly skewed data, we split unusually large clusters into smaller chunks. Clustering work items into groups with similar cluster size is an optimization that we consider future work.

Runtime Pruning

When evaluating a relatively low number of queries (thousands) on the high number of clusters, it frequently happens that a cluster is hit by no query. Since the overlap of the cluster with the query can be checked by looking at it's prefix, skipping cluster scans is an easy optimization. This is a case of the workaround we discussed in Section 2.2: bounding the problem size and abandoning kernel execution at runtime.

4.2 Phase 2: CPU Refinement

In the second phase, the CPU copies the partial results from the GPU's device memory and joins them with the main memory resident residual list. Since this is an invisible/positional join [1] on the tupleID, it is cheap. The partial results are combined with the residuals to produce the final tuple values (see Figure 9). The query conditionals are evaluated again on the reconstructed values and the results, in case of a hit, copied to the output buffer.

5. EVALUATION

To evaluate the performance impact of our approach, we compare it to existing CPU-only and GPU-only approaches. As benchmark we use a set of range queries on a spatial trajectory database.

5.1 Setup

The experiments where run on a machine with two Intel(R) Xeon(R) CPU X5650 @ 2.67 GHz with 48 GB of main memory. The used GPU is a GeForce GTX 480 with 1.5 GB device memory.

The data is a real-life dataset consisting of around 240 Million 2D spatial datapoints. The data was collected by an industry partner by tracking Navigation Devices in North-Western Europe. The queries are generated by randomly selecting a point from the dataset and constructing a rectangle around it. The size of the rectangle is random but within a maximum. The generation of the queries is according to a workload description that we received from mentioned industry partner.

5.2 Loading

The data is stored on disk in binary format. Before the query evaluation is started it is loaded into main memory and decomposed/-compressed as necessary for the presented query evaluation technique. As discussed in Section 3.3, decomposition and compression involves radix clustering the data. This is likely to increase the loading costs. To give an impression of the decomposition costs, Figure 10 shows the base costs for loading and the added costs for the clustering step. As expected, the clustering doubles the costs since it involves a second pass through the data.

5.3 Query Processing

To evaluate the query processing techniques, we varied three different parameters: 1. The number of queries evaluated, 2. the processing device (GPU vs. CPU), 3. the data representation (bitwise decomposed (BWD) vs. attribute-wise decomposed).

Since we parallelize query processing with the number of queries on the GPU, we effectively turned the evaluation of many queries into a single, parallelized nested loop (theta) join. For reference, we also report the results of a similar evaluation technique on the CPU. All processing models that evaluate multiple queries in a single run through the queries are marked with QJ=QueryJoin.

Figure 11a shows the results of our main experiment. CPU is the processing of the queries in a query-at-a-time manner on the plain data. This is the baseline for our evaluation. The state of the art for GPU processing relies on streaming the plain data to the GPU and evaluating the queries in parallel. While the performance compared to CPU-based processing is worse for a single query, the GPU benefits from larger query sets.

The QueryJoin optimization on the CPU shows worse performance than the baseline: The more complex loops seem to hurt CPU efficiency. The same holds when combining bitwise decomposition with the QueryJoin optimization. The evaluation of 2048 queries was not complete within 30 minutes at which point we aborted the query evaluation.

Bitwise decomposed storage and processing on the CPU is the best evaluated solution when evaluating a single query. For larger query sets, GPU/CPU co-processing outperforms all other approaches significantly. For 2048 parallel queries, GPU/CPU co-processing is

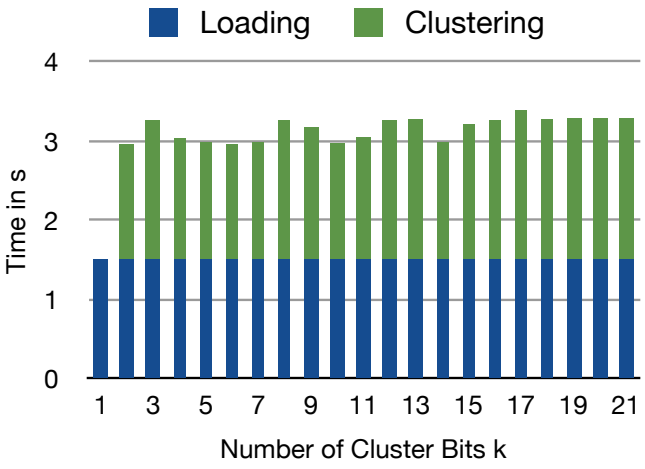


Figure 10: Data Loading+Clustering Time

about 6 times faster than bitwise decomposition on the CPU and more than two orders of magnitude faster than GPU processing on plain data. CPU processing on plain data is outperformed by more than three orders of magnitude.

In addition to the query evaluation performance, bitwise decomposition promises good load balancing over the available devices. To illustrate this, Figure 11b shows the time that is spent processing data on each device. It shows that single queries induce most of their load on the GPU, the load is almost perfectly distributed for larger query sets.

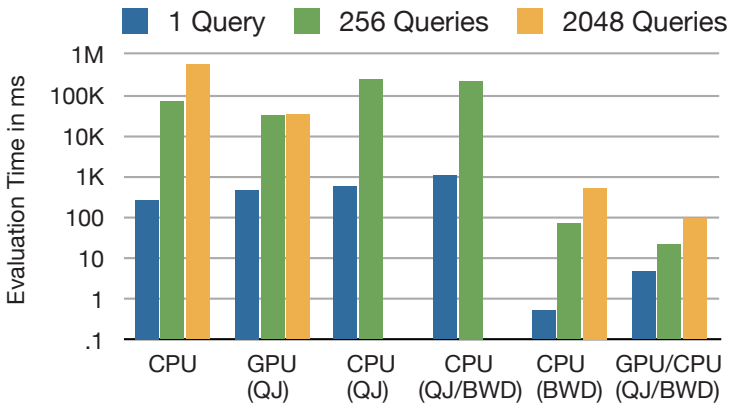
6. CONCLUSION AND FUTURE WORK

Efficient CPU/GPU processing is still an open research challenge. We presented a viable solution to this problem, tackling it by decomposing data into individual bits. Our approach outperforms current CPU/GPU co-processing strategies by more than two orders of magnitude for a spatial selection benchmark on real life data. This makes it a very attractive paradigm for relational cross device query processing.

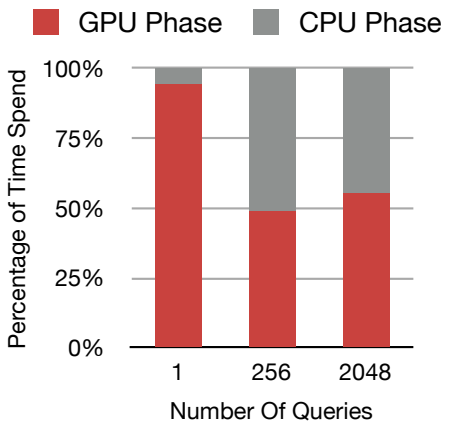
However, we believe that there is still room for further research. Focusing on the general strategy, we deliberately abstained from rigorous optimization to the available hardware (GPU and CPU alike). We believe that with more sophisticated optimization, focusing on, e.g., GPU memory access, we may improve the performance of our implementation even further. We also believe that co-processing setups other than GPU/CPU could benefit from the approach. SSD/HDD as well as physically distributed (Client-Server) architectures are likely candidates. In addition, we also believe that a study of the approach for other applications (joins, grouping, data mining, ...) is of value.

Acknowledgments

The work reported here has partly been funded by the EU- FP7-ICT project TELEIOS. This publication was supported by the Dutch national program COMMIT.



9(a) Processing Time by Evaluation Technique



9(b) Load Distribution for GPU/CPU

Figure 11: Query Evaluation Performance

7. REFERENCES

[1] D. Abadi, S. Madden, and N. Hachem. Column-stores vs. row-stores: How different are they really? In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 967–980. ACM, 2008.

[2] P. Bakkum and K. Skadron. Accelerating SQL database operations on a GPU with CUDA. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, pages 94–103. ACM, 2010.

[3] P. A. Boncz, M. L. Kersten, and S. Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, 2008.

[4] G. P. Copeland and S. N. Khoshafian. A decomposition storage model. In *Proceedings of the 1985 ACM SIGMOD international conference on Management of data*, SIGMOD '85, pages 268–279, New York, NY, USA, 1985. ACM.

[5] S. Ding, J. He, H. Yan, and T. Suel. Using graphics processors for high performance IR query processing. In *Proceedings of the 18th international conference on World wide web*, pages 421–430. ACM, 2009.

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

[7] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *Proceedings of the VLDB Endowment*, 3(1-2):670–680, 2010.

[8] W. R. Franklin. Adaptive grids for geometric operations. In *Sixth International Symposium on Automated Cartography (Auto-Carto Six)*, pages 230–239, 1983.

[9] M. G.M. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, Ottawa, Canada: IBM Ltd., 1966.

[10] G. Graefe. Volcano-an extensible and parallel query evaluation system. *Knowledge and Data Engineering, IEEE Transactions on*, 6(1):120–135, 1994.

[11] C. Gregg and K. Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on*, pages 134–144. IEEE, 2011.

[12] A. Guttman. R-trees: A dynamic index structure for spatial searching. In B. Yormark, editor, *SIGMOD'84, Proceedings of Annual Meeting, Boston, Massachusetts, June 18-21, 1984*, pages 47–57. ACM Press, 1984.

[13] B. He, W. Fang, Q. Luo, N. Govindaraju, and T. Wang. Mars: a MapReduce framework on graphics processors. In *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, pages 260–269. ACM, 2008.

[14] B. He, M. Lu, K. Yang, R. Fang, N. Govindaraju, Q. Luo, and P. Sander. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS)*, 34(4):21, 2009.

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

[16] A. Munshi. OpenCL specification 1.1. *Khronos OpenCL Working Group*, 2010.

[17] C. Nvidia. Compute Unified Device Architecture Programming Guide. *NVIDIA: Santa Clara, CA*, 83:129, 2007.

[18] H. Samet and R. E. Webber. Storing a collection of polygons using quadtrees. *ACM Trans. Graph.*, 4(3):182–222, 1985.

[19] L. Sidirourgos, M. Kersten, and P. Boncz. Sciborq: Scientific data management with bounds on runtime and quality. In *Proc. of the Int'l Conf. on Innovative Data Systems Research (CIDR)*, pages 296–301, 2011.

[20] M. Zukowski, S. Heman, N. Nes, and P. Boncz. Super-scalar ram-cpu cache compression. In *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*, pages 59–59. IEEE, 2006.