Accelerating Foreign-Key Joins using Asymmetric Memory Channels

Holger Pirk
CWI
Amsterdam, The Netherlands
holger@cwi.nl

Stefan Manegold
CWI
Amsterdam, The Netherlands
manegold@cwi.nl

Martin Kersten
CWI
Amsterdam, The Netherlands
mk@cwi.nl

ABSTRACT

Indexed Foreign-Key Joins expose a very asymmetric access pattern: the Foreign-Key Index is sequentially scanned whilst the Primary-Key table is target of many quasi-random lookups which is the dominant cost factor. To reduce the costs of the random lookups the fact-table can be (re-) partitioned at runtime to increase access locality on the dimension table, and thus limit the random memory access to inside the CPU's cache. However, this is very hard to optimize and the performance impact on recent architectures is limited because the partitioning costs consume most of the achievable join improvement [3].

GPGPUs on the other hand have an architecture that is well suited for this operation: a relatively slow connection to the large system memory and a very fast connection to the smaller internal device memory. We show how to accelerate Foreign-Key Joins by executing the random table lookups on the GPU's VRAM while sequentially streaming the Foreign-Key-Index through the PCI-E Bus. We also experimentally study the memory access costs on GPU and CPU to provide estimations of the benefit of this technique.

Categories and Subject Descriptors

B.4.3 [**Input/Output and Data Communications**]: Interconnections (subsystems); H.2.4 [**Database Management**]: Systems—*Query Processing*

General Terms

GPGPU, Query Processing, I/O Bottleneck

1. INTRODUCTION

Making efficient use of the processing power of modern CPUs with the available bandwidth is a great challenge of database research. On the one hand, advances in processing speed, e.g., due to the rising number of cores, keep on outpacing the improvements in memory access bandwidth and latency. On the other hand, database workloads typically require less computation than, say, scientific simulations or image processing, making it a none-trivial task to exploit the excess of compute power effectively. This problem has been addressed from different angles.

In software, the utilization can be improved by, e.g., Decomposed Storage [4], on-the-fly (de-) compression of stored data [24, 9] or cache-conscious query processing [16, 13].

In hardware, vendors do their best to increase the available bandwidth through more (e.g., in Intel's Nehalem Architecture), wider (Burst Mode) and faster (DDR) memory channels [14]. Unfortunately, the conflict between cheap, large and fast memory forces vendors to compromise in order to efficiently support a variety of applications.

The memory of GPUs takes a slightly different approach: it is generally smaller yet around an order of magnitude faster in throughput than CPU memory. The peak bandwidth to the internal memory of current GPUs exceeds 100 GB/s. Unfortunately, the transfer from and to the CPUs' memory is costly: In practice, a throughput of around 4 GB/s can be achieved through the PCI-E×16 Bus. This makes the GPU memory unattractive as the sole operative memory and raises the question: How are the available resources used most efficiently, or, to be more concrete: how can the amount of data that is transfered through the slow PCI-channel be minimized while still exploiting the full speed of the fast internal memory channel? Rather than approaching this problem with sophisticated algorithms we carefully investigate the available hardware and use each available component to the best of its capabilities. We achieve this by distributing data amongst the memory structures according to the typical data access pattern.

To this end, our contributions include

- a theoretical and experimental study of the memory access characteristics of a GPU's internal and external memory
- an assessment of the impact of the *randomness/unclusteredness* of data on the memory access performance,
- busting the myth of always needing many cores to max out the GPU's memory,
- discouraging the feasibility of clustered joins on GPGPUs.

We also introduce ideas of distributed query processing to collaborative CPU/GPU processing in general and evaluated the benefits of a technique derived from Data Warehouse Striping in particular.

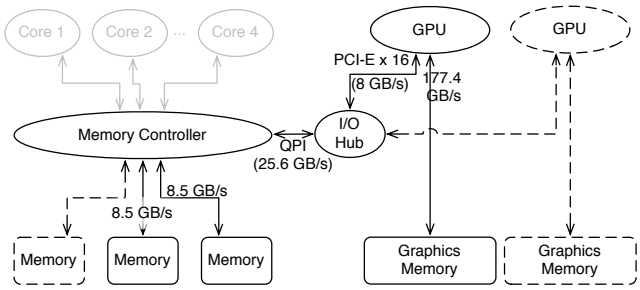


Figure 1: Architecture of our Test System (Dotted Components are optional and not installed)

Whilst the usage of GPUs for DBMS operations has been studied [13, 9, 11, 12], the common view on GPUs has, hitherto, been that of a coprocessor. As opposed to *real* coprocessors, however, (discrete) GPUs have the discussed PCI bottleneck which results in characteristics that are much closer to a distributed system than a processor/coprocessor architecture. This problem has been mentioned in previous work and is tackled through two techniques: compression [9] and reflection in the cost model/optimization [12]. To the best of our knowledge, a-priori cost modeling and appropriate data placement has not yet been used to tackle this problem. Generally, the data access characteristics of GPUs when accessing on- as well as off-board memory have not received significant attention. We believe that a careful analysis of the respective costs can improve performance of GPGPU query processing as well as simplify the algorithms needed to achieve good performance. Based on the results of this study, we propose a technique that we call *multi-channel data processing*: Transmitting data to the processing device (in our case the GPU) through multiple channels, using each available channel according to its unique properties.

The rest of this paper is organized as follows: In Section 2 we introduce the memory architecture of GPUs and discuss its relevant characteristics for our case. In Section 3 we provide a simple cost model for multi-channel memory access systems that gives an idea of the expected performance gains. We evaluate the performance improvements of our approach in Section 4. In Section 5 we present past and future work that may be applied in combination with our approach and discuss its applicability to various applications. We conclude in Section 6.

2. BACKGROUND

Before exploring Multi-Channel Query Processing we introduce the base technology, i.e., the memory anatomy of CPU/GPGPU system. In this section, we discuss the options for the transfer of data from the host to the device and the access characteristics of the internal memory.

2.1 Host to Device Data Transfer

Exchanging data between host and a device is the fundamental operation of any system I/O. To get a general impression of the integration of a GPGPU device into the host system's memory structure, consider Figure 1. Our Nehalem-class test system incorporates two memory channels (high end systems may have three) that are connected to the memory controller [14]. The bandwidth of these depends on the external clock frequency of the CPU but is in the range of 8 to 10 GB/s. The Memory Controller is connected to the I/O Hub through the QPI-bus which has a peak bandwidth of 25.6 GB/s [15]. In practice it may be used for other purposes like, e.g., cache-coherency amongst CPUs as well, which may put additional load on the bus. The I/O Hub controls the PCI-E bus and may, in theory, transfer up to 8 GB/s (16 PCI-E transfer lanes with 500 MB/s each) to each GPU device, currently up to a limit of 18 GB/s (the Intel X58 IOH supports up to 36 PCI-E lanes).

The data access granularity between the I/O Hub and the Memory is determined by the burst size of the Memory which is 64 bytes for DDR3 RAM. The granularity on the PCI-bus is generally 64 bit but incurs a large overhead per word. The PCI controller, therefore, has the option of *PCI posting*: combining several PCI adjacent (write) requests into a single burst [23]. The maximum length of these bursts is implementation specific. In practice, random access to the host memory from the GPU should be avoided at all costs and is often unsupported by the hardware.

The transfer data between the device and the CPU can be performed in one of two ways: controlled by the Device (DMA) or controlled by the CPU itself (Memory Mapping). Depending on the need for preprocessing (e.g. preselection), either may have advantages. Unfortunately, some vendors only support a subset of the available techniques[1]. Since the host to device transfer is the limiting factor for data intensive applications (see Sections 3 and 4.2), however, it is crucial that both methods are implemented. We discuss the technical implementation of the transfer methods and their respective advantages in the following.

Mapping Device Memory

The ability to map device memory into the addressable memory of a user process is an integral part of the x86 architecture[2][14]. Historically, the *Northbridge* took care of all memory accesses on the "fast" connections, i.e., RAM, AGP and PCI-E devices.

Most current CPUs come with an integrated memory controller and only rely on the Northbridge for real I/O. This architecture will be our focus. When accessing a memory address that falls into the designated area for memory mapped devices, the CPU sends it to the *Northbridge (a.k.a. I/O Hub)* using QPI (or the AMD equivalent: HyperTransport). For PCI-E devices, the Northbridge takes care of wrapping the memory access into the appropriate Bus protocol and sending it to the device. PCI posting gives Memory Mapped Devices similar access characteristics as regular memory: data is accessed in blocks that are similar to cache lines for regular memory.

Direct Memory Access

Direct Memory Access (DMA) gives a device access to the system's main memory without involving the CPU and its internal buses for every single transfer. DMA is controlled by the device and can be initiated by the CPU or the device. In the earlier case, which is more interesting to us, the CPU prepares an area of the memory for DMA (this is sometimes called *pinning* of memory) and triggers the transfer by signaling the device. In the case of PCI, the

[1]E.g., the *ATI Stream SDK 2.4* supports neither Memory Mapping nor non-Bulk (Device-initiated) DMA on Linux

[2]and most other platforms as well

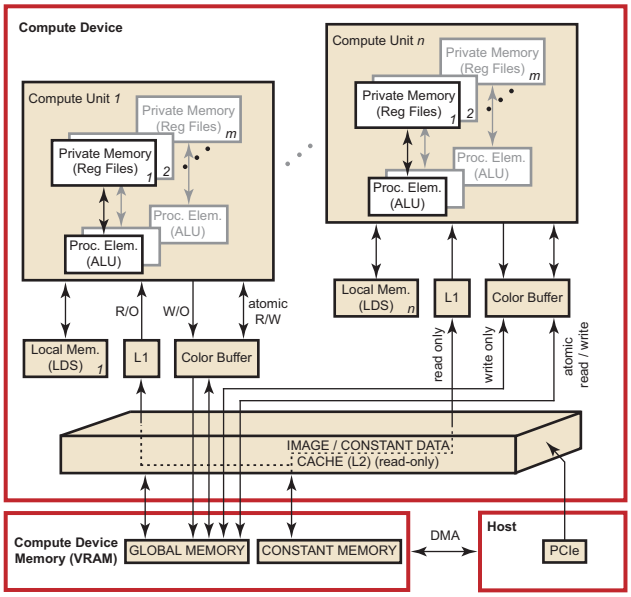


Figure 2: Memory Structure of an ATI (Evergreen) GPGPU card (taken from [1])

device becomes the *Bus-Master*, requests (parts of) the prepared region and issues an interrupt to the CPU once it is done [10]. Pinning the memory can be an expensive operation. Depending on the support by the hard- and software it might involve copying the data to a contiguous area in (kernel-)memory. Some current PCI-E devices support scatter/gather-lists that avoid this additional copy. Regardless of the necessity for this copy, the Operating System has to ensure consistency of the transferred data by flushing the caches and preventing the paging to disk. Setting up a DMA transfer is, thus, costly and should be done only for large (several megabytes at least) amounts of data.

If data resides readily in main memory, we expect DMA to perform better than Memory Mapping because it avoids an additional pass through the CPU's memory hierarchy. If, however, the data has to be modified (e.g., preselected or partitioned) it has to pass through the CPU in any case. In this case we expect Memory Mapping to outperform DMA because it avoids materializing the intermediates in RAM. Unfortunately, the implementation of host to device transfer, especially on "exotic" platforms (i.e., not Windows) is often not optimally exploiting the available hardware features, thus limiting the actual performance.

GPU Device Memory

In addition to the host memory the GPU has access to an internal memory (also called Video RAM, VRAM, Graphics or Device Memory). Figure 2 shows a memory architecture diagram of an ATI Evergreen class GPU. Prominent are two things: a) data can either be transferred into the VRAM or read directly from the host memory using PCI-E and b) the Level 1 and 2 Caches are read-only which obviously has some implications on its usage.

The VRAM has a bandwidth in the range of triple-digit GB/s but also a comparatively high latency of around 200 to 300 cycles as opposed to common CPU memory latency of 50 to 60 cycles. This conscious design decision is mitigated by the high degree of parallelism: When used correctly, the computation of one thread can hide the memory access latency of another. Correctly exploiting this parallelism is, however, not trivial. To simplify the programming of massively parallel computation devices, vendors rely on the *kernel programming model*. While supporting massive parallelism, this model comes with a number of limitations and pitfalls. In Appendix A be provide a brief overview of the kernel programming model and one of it's implementations, *OpenCL*, for the interested reader.

3. THEORETICAL MODEL

The discussed hardware properties inspired us to the following approach: for a sequential access and a concurrent random access, the operations are executed using different memory channels, taking the characteristics of the channels into account. We call a device that has multiple memory units attached using channels with different properties a *Multi-Channel (Processing) Device*. Query Processing that takes these properties into account will be referred to as *Multi-Channel Query Processing*.

In this section we analyze this approach theoretically by estimating the costs on either hardware using a simplified version of the Generic Cost Model for Hierarchical Memory Systems [17]: it models only a single layer of memory and therefore does not consider any caching effects. Even though this limits the general applicability of the model, the case we study generally involved tables that exceed the available cache by far. This makes caching largely irrelevant. For the purpose of this paper, our model aims at providing an intuitive insight into the rationale of our approach, rather than aiming at a most detailed and accurate prediction. The model, as well as our evaluation is targeted towards main memory DBMS using decomposed storage. We assume that all data is readily available in memory and neglect any disk I/O. Thus when mentioning to I/O in this paper, we refer to main-memory access costs, rather than disk I/O. We also use decomposed storage for all relations and intermediates.

The model depends on four input parameters:

$B_{mem/s}$ denotes the block size of the sequential memory channel and defines its data access granularity. This is the size of a memory burst/cache line.

$B_{mem/r}$ denotes the block size of the random memory channel and defines its data access granularity. This is the size of a memory burst/cache line.

B_{cpu} denotes the size of the processed data type on the sequential channel. It is mainly used to calculate how many data items are processed per cache line which determines the number of random misses per sequential miss.

$T_{mem/r}$ denotes the bandwidth/throughput of the channel that is used for random accesses.

$T_{mem/s}$ denotes the bandwidth/throughput of the channel that is used for sequential accesses.

The target measure T_{eff} gives an estimation of the expected throughput in terms of data processed from the sequentially channel. Note that our model does not take the cache capacity into account: we assume a cache miss for every lookup/processed data item. This is, of course, not

	CPU	GPU (ATI)	GPU (NVidia)
Type	Intel® Core™ i7 860	ATI Radeon™ HD 5850	NVidia GeForce GTX 480
Memory	8G	1G	1.5G
Internal Memory Bandwidth ($T_{mem/r}$)	17GB/s	128 GB/s	177.4 GB/s
External Memory Bandwidth ($T_{mem/s}$)	17GB/s	3.5 GB/s	4 GB/s
Access Granularity (B_{mem})	64B	128B	128B

Table 1: Evaluation Hardware Parameters

always the case, but becomes a problem for random accesses to large data structures. Since our focus is processing large dimension tables, we consider a model that takes caching into account out of scope. To feed the model, we use the parameters as seen in Table 1 that reflect the specifications of our target hardware.

Modeling a Single Memory Channel

As a first intermediate, we calculate the number of bytes that have to be read from the random access channel to process one cache line from the sequential access channel $B_{eff/r}$ using Equation (1). $B_{eff/r}$ is simply the number of processed data items per sequential cache line multiplied with the size of a random cache line.

$$B_{eff/r} = \frac{B_{mem/s}}{B_{cpu}} \times B_{mem/r} \tag{1}$$

From that, we can estimate the effective throughput on a single-channel system using Equation 2: the number of processed sequential cache lines per second is the memory bandwidth divided by the data needed to process the cache line. The effective data throughput is the number of processed sequential cache lines per second multiplied with the size of a sequential cache line.

$$T_{eff} = \frac{T_{mem}}{B_{eff/r} + B_{mem/s}} \times B_{mem/s} \tag{2}$$

On a Nehalem System, e.g., the cache line size B_{mem} is 64 byte. When processing 4 byte integers ($B_{word} = 4$), we effectively transfer $B_{eff/r} = 16 \times 64 = 1024$ bytes on the random access channel per processed cache line on the sequential channel. $B_{eff/r} + B_{mem/s}$ is, thus, 1088. Since the available bandwidth is two times 8.5 GB/s we can process 16 million cache lines or roughly one GB per second on the sequential channel.

Modeling Multiple Memory Channels

Using a multi-channel system we are limited by the slower channel. Thus, Equation (3) can be used to estimate the costs: the throughput on the random channel is calculated similar to Equation 2 but without including the traffic for the sequentially accessed cache lines.

$$T_{eff} = \min\left(\frac{T_{mem/r}}{B_{eff/r}} \times B_{mem/r}, T_{mem/s}\right) \tag{3}$$

The ATI Radeon HD 5850 is specified with an internal memory bandwidth of 128 GB/s and a cache line size B_{mem} of 128 byte. From that we can calculate an effective throughput that is limited to 3.5 GB/s by the sequential traversal channel. This is more than three times the performance of the single-channel CPU-only processor. This is our performance target.

However, this only holds for applications with many capacitive cache misses. In the Section 5.4 we discuss potential applications that may fit this pattern.

I/O boundness on GPUs

The I/O boundness of Database Operations on CPUs is well researched [4, 24]. GPUs are, however, made out of many relatively slow processing units. This changes the balance between computation and I/O when using only a single core. Therefore, GPUs rely on massive parallelism to max out the memory bandwidth. Consequently, existing work (e.g., in [8]) focuses on maximizing the processing parallelism, oblivious of the fact that it might not be necessary due to I/O starvation. In the case of joins, the input data is usually radix-partitioned and every partition processed in one thread [8]. This allows efficient lock-, synchronization- and replication-free parallel processing of the partitions if the data is distributed close to uniformly. For skewed data, radix-partitioning might fail to achieve the aspired performance improvement due to suboptimal load balancing. It may even decrease performance due to the clustering overhead.

4. EXPERIMENTS

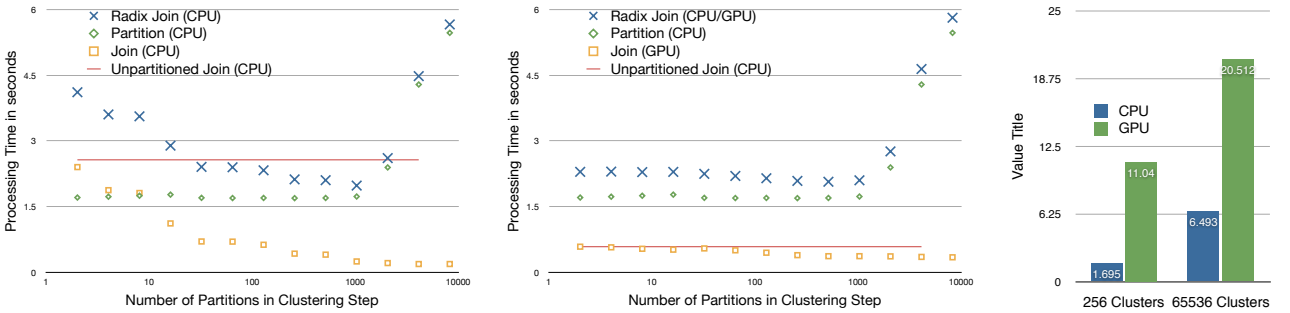
To gain detailed insight, we focus our experiments on the very core of a foreign-key join between a large table that is located in the host memory (or even on the host's (flash-) disk) and a smaller table that is located in the GPU's device memory. This closely resembles the case of a single dimension Star-Join between a large fact-table and a smaller dimension-table. We will therefore also refer to the smaller table as dimension-table. We consider the join attribute to be a 4 byte integer key, and assume that accesses to the smaller table are supported by a Foreign-Key index.

We further assume that all tables are stored in decomposed representation [6] and only the join-index attribute of the large table is sent to the GPU. Given that the join algorithm preserves the tuple-order of the large table in the join result, projection to more columns of the large-table can be done efficiently on the CPU. This is similar to semi-join solutions for join-processing in distributed database systems.

To evaluate the feasibility of our approach, we compare a state of the art radix-join on a modern CPU with our CPU/GPU hybrid implementation of a naïve implementation of an unpartitioned star-join. In this section we describe the system configuration and present our results.

4.1 Setup

The CPU implementations of our evaluation are run on a current mid-range Intel Nehalem Workstation. The GPU implementations are run on an mid-range ATI card as primary platform and on an upper mid-range NVidia card for reference. Detailed information about the hardware are dis-



(a) Partition+Join on CPU (b) Partition on CPU, Join on GPU (c) Partition CPU vs GPU

Fact-table: 1G, Dimension-table: 64M

Figure 3: Radix Join: CPU only vs. CPU+GPU

played in Table 1.

The best competing approach for star-joins is radix-joining which works best if the fact-data is uniformly distributed. We, therefore, only considered uniformly distributed data and expect even better results for skewed data.

The CPU star-join is implemented using a single pass out of place radix partitioning step. The join-phase is a single-threaded loop. In order to efficiently exploit memory bandwidth we explicitly unrolled the loop to achieve a consistently high number of outstanding cache misses. Since we are only interested in the actual join-performance, the results were directly aggregated while running the loop. We, thus, avoided any additional I/O so we can compare pure join-performance. In addition, we only consider Foreign-Key joins using pre-built hash-tables. Naturally, the existence of a Foreign-Key index on the fact-table makes any clustering of the Dimension-Table unnecessary.

4.2 Results

Using this setup, we conducted experiments to answer three questions: 1. Can our approach compete with a sophisticated CPU-only implementation? 2. How does it scale with an increasing dimension table size and how does it compare to a CPU-only implementation? 3. How much parallelism is needed to max out the GPUs internal memory bandwidth and do the gains justify the costs for partitioning?

Radix Joins on CPUs and GPUs

We evaluated the radix join for fixed Dimension- and Fact-table sizes. We varied the number of generated partitions and measured the execution time of the individual steps. Figure 3(a) shows the results which are consistent with the expectations and the recent literature [3]: The radix join is very sensitive to the optimal selection of the parameters and does not perform much better than the unpartitioned join. Even for the optimal number of partitions on our machine, the radix-partitioned hash-join performs only 30% better than the unpartitioned join because the expensive partitioning is not amortized by the improvement in the join costs. We observed different results in favor of partitioning on different machines and will report them separately.

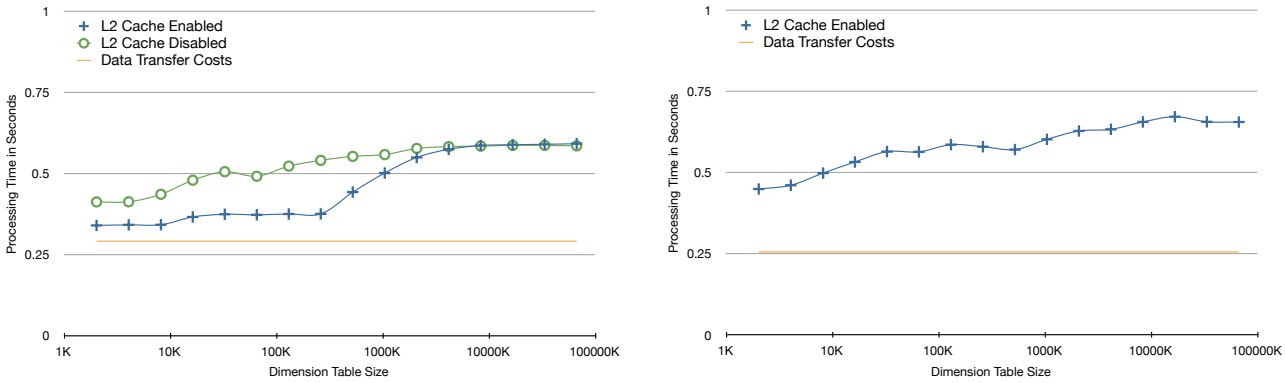
We also evaluated the join performance on the GPU and found that the partitioning step on the GPU is even more expensive than on the CPU (see Figure 3(c)).

Unpartitioned Foreign-Key Joins for Varying Size

Next, we implemented our hybrid CPU/GPU non-partitioned Foreign Key Join using OpenCL [18] and ran it on the ATI card, the Nvidia card and the i7 CPU. We varied the size of the dimension table from 2K to 64M. On the ATI card we also controlled if the dimension table was cached in the private/shared caches of the GPU. This can be done by declaring the data structure read-only vs. writable: because the GPU's cache is read-only, only data structures that are explicitly declared as read-only are cached. Figure 4(a) illustrates the effect of caching on the GPU. With activated caching, we observe an overproportional increase of the costs whenever we exceed the size of a cache. This is similar to what has been observed on CPUs [17, 4] yet a lot less severe. Even when disregarding the costs for the data transfer from the host to the device, the performance only improves by a factor of roughly 2.5. Without caching, the costs start at a higher baseline but approach the same maximum. Figure 4(b) shows a similar behavior on the NVidia card. Figure 5 depicts our main result: the increase in the lookup costs on the CPU outgrows the additional costs for the host to device transfer as soon as the dimension table runs out of cache. On our machine, the multi-channel GPU approach outperforms the CPU for dimension tables larger than 8MB due to heavy cache and TLB thrashing on the CPU. It also shows that our model gives a reasonable prediction of the join performance for large dimension tables on the GPU and the CPU.

Foreign-Key Join with Varying Number of Threads

Figure 6 shows the impact of the level of parallelization on the query execution time. We observe a drop of the query costs when increasing the number of threads from one to two. This drop in query costs is expected because one thread can be used to partially hide the latency of the other. Beyond that, we don't observe any significant improvement in the query performance. Indeed, the query time actually increases for a high number of threads on the Nvidia card which suggests that the thread scheduling overhead on the Nvidia card is higher than on the ATI card. More importantly, however, this strongly discourages investing resources (in implementation as well as data structures) into the massive parallelization. Even though the precise impact of massive parallelization on other cases is an open question,



(a) ATI Radeon HD 5850

(b) NVidia GeForce GTX 480

Figure 4: Impact of Dimension Table Size on Foreign-Key Join Performance (Number of Threads: 64)

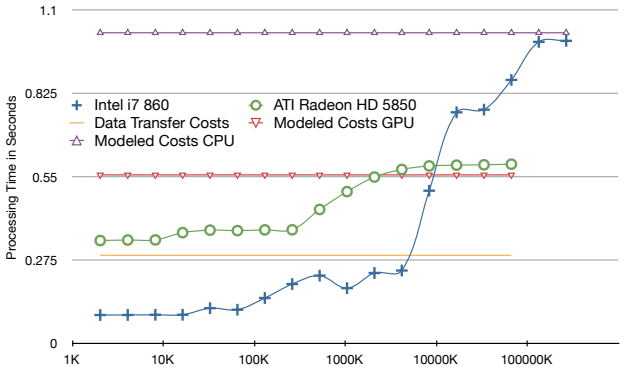


Figure 5: Intel i7 vs. ATI Radeon HD 5850

this strongly suggests that more research is needed. We also expect that a lower latency of the memory will arouse the need for higher parallelism.

5. RELATED AND FUTURE WORK

The memory architecture of a GPU/CPU system largely resembles that of a distributed system: two or more devices that have relatively fast internal memory are connected through a relatively slow channel cooperate to process a query. It may, thus, be beneficial to assess the applicability of techniques for distributed query processing at hand. In this section we provide a very brief overview of existing techniques of distributed query processing and suitable applications for the presented approach.

Distributed query processing faces two major questions: how to distribute data and how to distribute processing.

5.1 Data Distribution

Classical distributed query processing (see, e.g., [21] for an overview) usually assumes that the nodes have equal or at least similar resources regarding processing, storage as well as I/O performance. Under this assumption, the main challenge is to limit inter-node communication and even out data skew. A popular paradigm to achieve this is MapReduce [7]. However, it is mainly targeted at large homogeneous clusters where it provides a good means for load balancing. It is less well suited for our, very asymmetrical, case.

Classical Database distribution schemes with the goal of minimizing inter-node communication range from full denormalization [5], (clustered) partitioning [22] to (partial) replication [2]. Due to the asymmetric capabilities of the nodes in our case, however, we deem naïve splitting into equally sized partitions infeasible: we need a more sophisticated distribution strategy that takes the induced access pattern into account.

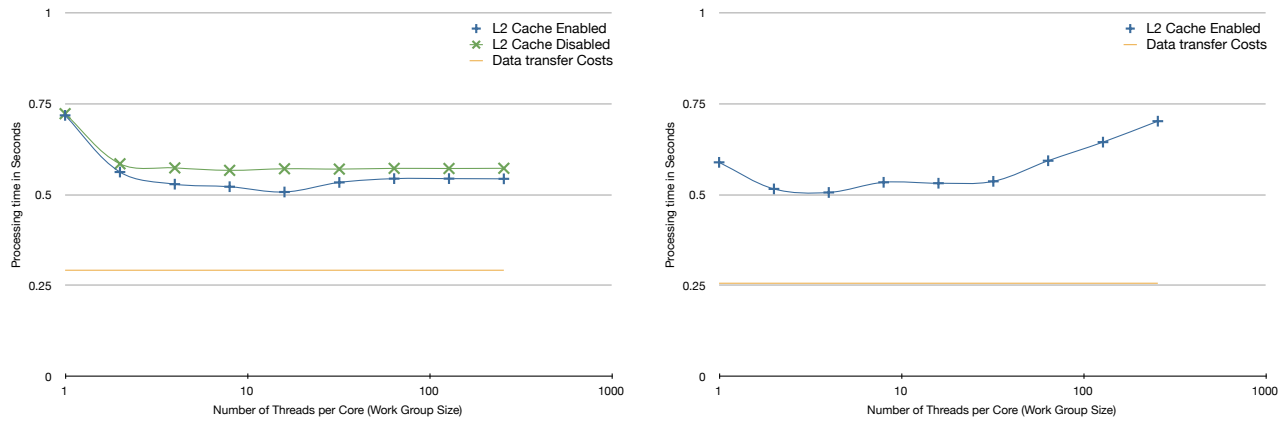
We see three possible approaches to this problem: a priory modeling/optimization, runtime monitoring/learning and manual placement. The first may suffer from suboptimal placements due to mispredictions and inaccuracies in the model, the second from unnecessary data transfers due to additional replacements. The third requires an experienced DBA with enough knowledge to manage the technology. Even though a good automatic Data Distribution scheme is necessary to help adoption of this technique, we considered this out of scope of this paper and leave it for future work. For now, we resorted to a heuristic that is close to Data Warehouse Striping [2].

Data Warehouse Striping [2] is a technique that is known to work well on Star/Snowflake like schemas: dividing the large fact table into interleaved partitions and replicating the small dimension tables to every node. Even though, we expect a more sophisticated distribution scheme to perform better for a broader range of applications, Data Warehouse Striping provides a good heuristic for our initial experiments.

This technique does, however, involve the replication of *all* dimension tables to all participating devices. For large dimension tables this may become an infeasible solution and more sophisticated techniques that take the available resources into account are called for.

5.2 Query Distribution

Since data is not exclusively stored but merely replicated on the GPU, query optimization has another degree of freedom. Some operators may be evaluated on either the GPU or the CPU (or even both). This degree of freedom, however, calls for query optimizers to take it into account. Since most DBMSs already come with a cost model and query opti-



(a) ATI Radeon HD 5850

(b) NVidia GeForce GTX 480

Figure 6: Impact of Level of Parallelization on Foreign-Key Join Performance

mizer, it is reasonable to leverage and extend the cost model to incorporate the additional factors: GPU processing and I/O costs as well as the costs for the transfer. Prior work [8] includes such a cost model which is also used to improve performance. However, it is only used to decide where to execute each operator in isolation and has does distinguish different kinds of memory. A strategic optimization of the query plan to, e.g., favor one device over another to avoid the transmission costs for the next operator is called for. The threshold for the size of the dimension-table at which an evaluation on the GPU is sensible could be determined using such a model.

5.3 Compiling Queries to Machine Instructions

GPUs are, in general, programmed using code that is compiled and transferred to the device at runtime. This gives the option of generating code that is custom made for the query at hand yet induces additional costs for compilation. The impact of compiling a query down to Machine Level Instructions on CPUs has received some attention lately [19]. Using Machine Level Instructions directly removes interpretation overhead and allows a very efficient exploitation of CPU as well as operator pipelining. Since all GPU programs are created and transferred to the device at runtime, we may see similar gains. However, the balance between increased costs for query compilation and and the performance gains at runtime have not been explored yet.

5.4 Applications

The idea of multiple I/O channels is, in general, applicable to most database operations. The asymmetric properties of the different channels, however, make some operations a better fit for the approach than others. Good candidates are all operations that involve concurrent random and sequential memory accesses on large datasets. We identified three major cases where an operation of that class occurs.

Dictionary Lookups

Somewhat similar is the application for dictionary compressed data. If the dictionary is larger than the cache the lookups cause a lot of thrashing. The access to the dictionary may, thus, create a bottleneck for the decompression. This could be alleviated by sacrificing fast sequential memory access on the compressed relation in favor of fast access to the dictionary.

Out of Order Tuple Reconstruction

A need for faster random access may also arise when reconstructing tuples in a column store. Whilst this operation causes sequential access for in-order reconstruction, the access pattern may change if executed after a join or group by. Especially in the case of a join (which may yield an increase of the data volume), the attribute that is to be reconstructed might fit in the GPU memory whilst the join product does not.

Multidimensional OLAP

Data Warehouses are usually organized in a Star or Snowflake Schema. These schemas exhibit a data distribution amongst the tables that fits the idea of Multi-Channel Query Processing very well: few, large, volatile fact tables that are normally accessed sequentially and many, small, quasi static dimension tables which are commonly accessed randomly using a Foreign-Key relation. Most multidimensional OLAP queries involve both kinds of tables which makes them a good candidate for the access through multiple channels. For now, we only consider single-way joins which limits the applicability to one-dimensional queries. A generalization for multi-way joins is target for future work.

6. CONCLUSION

I/O-Bound Applications like Database Management Systems have to scrape together bandwidth wherever they can. Especially repetitive random accesses to datasets that exceed the cache can put significant load on the memory bus due to heavy cache thrashing.

We investigated a promising technique to increase the effective processing bandwidth in order to improve query evaluation performance in throughput and latency. To address the challenges that are introduced with this new technique we have introduced techniques of distributed query process-

ing to the realm of hybrid GPU/CPU processing. We applied a technique based on Data Warehouse Striping and to the case of Foreign-Key Joins. The results show a significant decrease of the query evaluation time. This encourages us to continue research in this field which still holds a number of challenges in store. We identified a number of potential applications for our approach as well as related techniques that may help to increase the breadth of applications that profit from the approach.

Acknowledgement

The work reported here has partly been funded by the EU-FP7-ICT project TELEIOS.

7. REFERENCES

[1] AMD. *AMD Accelerated Parallel Processing OpenCL Programming Guide*, 1.3c edition, June 2011.

[2] J. Bernardino and H. Madeira. Data warehousing and OLAP: improving query performance using distributed computing. In *CAiSE* 00 Conference on Advanced Information Systems Engineering*. Citeseer, 2000.

[3] S. Blanas, Y. Li, and J. Patel. Design and evaluation of main memory hash join algorithms for multi-core cpus. In *Proceedings of the 2011 ACM SIGMOD international conference on Management of data*, 2011.

[4] P. Boncz, S. Manegold, and M. Kersten. Database architecture optimized for the new bottleneck: Memory access. In *In Proceedings of VLDB Conference*, pages 54–65, 1999.

[5] S. Chen. Cheetah: a high performance, custom data warehouse on top of MapReduce. *Proceedings of the VLDB Endowment*, 3(1-2):1459–1468, 2010.

[6] 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.

[7] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[8] 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.

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

[10] A. F. Harvey and D. A. D. Staff. *DMA Fundamentals on Various PC Platforms*. National Instruments, April 1991.

[11] B. He, N. Govindaraju, Q. Luo, and B. Smith. Efficient gather and scatter operations on graphics processors. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, page 46. ACM, 2007.

[12] 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.

[13] 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.

[14] Intel Corporation. *Intel® 64 and IA-32 Architectures Software Developers Manual*, June 2009.

[15] Intel Corporation. *An Introduction to the Intel® QuickPath Interconnect*, January 2009.

[16] S. Manegold, P. Boncz, and M. Kersten. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Engineering*, pages 709–730, 2002.

[17] S. Manegold, P. Boncz, and M. L. Kersten. Generic database cost models for hierarchical memory systems. In *VLDB '02: Proceedings of the 28th international conference on Very Large Data Bases*, pages 191–202. VLDB Endowment, 2002.

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

[19] T. Neumann. Efficiently compiling efficient query plans for modern hardware. In *Proceedings of the VLDB Endowment*. VLDB, VLDB Endowment, 2011.

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

[21] M. Özsu and P. Valduriez. *Principles of distributed database systems*, volume 3. Springer New York, 2011.

[22] R. Ramakrishnan and J. Gehrke. *Database management systems*. McGraw-Hill, 2003.

[23] P. Specification. Revision 2.2. *PCI Special Interest Group*, 12, 1998.

[24] M. Zukowski, P. Boncz, N. Nes, and S. Héman. MonetDB/X100-a DBMS in the CPU cache. *Data Engineering*, 1001:17, 2005.

APPENDIX

A. GPGPU PROGRAMMING

Programming the high number of 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 [20] and OpenCL [18]. Whilst 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. In addition, the performance characteristics of the various implementations vary greatly. In this section, we discuss basic concepts of OpenCL and the relevant limitations of the programming model.

The Kernel 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.

Static Memory Allocation

The static memory allocation becomes a problem for operations for which the size of the output cannot be determined a priory. For selections the problem is somewhat manageable because the output is always smaller than the input, which gives a reasonable upper bound for the output size. The problem is harder for joins, where the upper bound, i.e., the product of the joined relations, can be large yet is rarely met. He et al. [13] propose to execute a join twice: once to estimate the size of the output and a second time to produce the output. Should the size of the output exceed the available storage, the authors propose to evaluate the joins in multiple passes which, obviously, increases the computational effort.

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. In particular, this speeds up random memory accesses significantly. We will evaluate the random memory access performance in Section 4.

A Priory Fixed Problem Size

Similar to input and output memory size, the problem size has to be specified up front. 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. 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: firstly, only a single core of a single processor can be active which may not be enough to max out the memory bandwidth and secondly there is no opportunity to hide stalls in a core by scheduling a different task. We will evaluate the performance impact of the degree of parallelism in Section 4.2.

Maximum Allocation/Mapping Size

Given the maximum size of the internal GPU memory, most GPUs use 32-bit (or even smaller) addresses for internal as well as external memory. To take it into account, the OpenCL standard defines a maximum size for a single allocation. The size is implementation specific and at least 128 MB. Whilst this ensures compatibility to lower-end cards, it poses challenges when using the GPU for larger datasets: the data has to be sliced up into several buffers. Even though this is not a fundamental problem, it complicates the implementation and may induce buffer management overhead at runtime.

Single Instruction Multiple Threads

Whilst not strictly a problem of OpenCL, a GPU hardware peculiarity is the notion of SIMT (Single Instruction Multiple Threads). SIMT is 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 (SIMD-)cores which execute every instruction for at least 4 cycles. Thus, every scheduled instruction is executed at least 64 times (usually on different data items). 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.