Joins in a Heterogeneous Memory Hierarchy: Exploiting High-Bandwidth Memory

Constantin Pohl
TU Ilmenau, Germany
constantin.pohl@tu-ilmenau.de

Kai-Uwe Sattler
TU Ilmenau, Germany
kus@tu-ilmenau.de

ABSTRACT

With High-Bandwidth Memory (HBM), an additional opportunity on hardware side for performance benefits is given. The large amount of available bandwidth compared to regular DRAM allows the execution of high numbers of threads in parallel masking penalties of concurrent memory accesses. This is especially interesting considering database join algorithms optimized for multicore CPUs, even more when running on a manycore processor like a Xeon Phi Knights Landing (KNL). The drawback of HBM, however, is its small size and given penalties in random memory access patterns.

In this paper, we analyze the impact of HBM on join processing exemplarily on the KNL manycore architecture. We run certain main memory hash join and sort-merge join algorithms of relational DBMS as well as data stream joins, comparing execution time in different HBM configurations. In addition, we consider data skew and output materialization for our measurements. Our results finally show performance gains up to 3x for joins when HBM is used. However, there is still a lot of room for improvements to fully utilize this kind of memory. Therefore, we give additional advices regarding HBM at the end of this paper.

KEYWORDS

Parallelism, Join, HBM, Manycore, Xeon Phi, KNL, MCDRAM

1 INTRODUCTION

Specialization is a key for improvements in all parts of life. This applies to database systems as well [21], in terms of software and used hardware. There are specialized DB variants for processing data streams, relational tables, graphs and many more, optimized mainly for query performance.

On hardware level, a heterogeneous landscape of processors and memory evolved over the last decades. Some new database systems are developed to efficiently support certain hardware sets, like SAP HANA [9] as an example for an in-memory database, thanks to highly increased main memory size, or OmniDB [11, 25] and CoGaDB [6], focused on using GPUs and Coprocessors to accelerate query processing speed. But not only full database systems adapted on new hardware trends. Many existing algorithms make their parameters conditional on the underlying hardware, e.g. regarding cache and TLB sizes for partitioned hash joins [3, 5, 16].

High-Bandwidth Memory (HBM) is another entry in the memory hierarchy [15]. However, HBM is rarely met in multicore CPU environments today. GPUs on the other hand can scale very well with HBM support because of their high lightweight thread count and many concurrent memory accesses. Since the release of the Xeon Phi Knights Landing (KNL) manycore CPU from Intel 2016, the HBM got an increased research interest. The KNL, supporting up to 288 threads on a single chip, comes with 16GB of HBM, the so called Multi-Channel DRAM (MCDRAM). Because of the huge amount of supported full-fledged threads, the memory controllers to regular main memory can be overwhelmed very quickly, depending on the application. This leads to stalling memory requests and idling threads, degrading performance in general. HBM tries to close this gap by providing high memory bandwidth to applications. For the MCDRAM, the bandwidth can exceed 400GB/s while regular DDR4 main memory reaches its limit at around 90GB/s. Furthermore, the KNL as well as the MCDRAM provide different hardware configuration possibilities. This allows additional tuning depending on the given applications, like dividing the cores into NUMA regions or running MCDRAM as a huge last level cache instead of addressing it explicitely.

Regarding memory bandwidth and join processing, the everlasting question arises shortly afterwards if a sort-merge join can surpass a careful tuned hash join in terms of performance. Since sort-merge joins benefit very well from wider SIMD registers and higher memory bandwidth [2], expectations arised that it will possibly outperform hash joins at some point. With AVX512 on the KNL as well as the MCDRAM described above, there is a lot of potential. However, our scope of this paper is not about reviewing this ongoing dispute in detail, but rather showing the impact of increased bandwidth on these join algorithms.

All in all, we explicitly address the usage of High-Bandwidth Memory on database join operators in this paper. We therefore use the KNL manycore processor in different configurations, deriving strategies and advices when using HBM for joining, based on our results. The questions being answered by this paper can be written as follows.

(1.) *How to exploit HBM for certain data structures and which considerations are necessary?*
(2.) *What is the impact of HBM on database joins compared to regular memory usage?*
(3.) *Can a sort-merge join surpass hash joins in performance when HBM is used?*
(4.) *Is there a difference in using HBM transparently like CPU caches or explicitly for certain data structures?*

DaMoN'18, June 11, 2018, Houston, TX, USA

ACM ISBN 978-1-4503-5853-8/18/06…$15.00
https://doi.org/10.1145/3211922.3211929

The next Section 2 gives an overview of related work done with HBM as well as database join processing. Section 3 is about HBM properties and a comparison to other memory variants. Then Section 4 briefly describes the evaluated join algorithms in DBMS and also in Data Stream Management Systems (DSMS). After that, Section 5 shows our experiments with HBM by defining the used hardware, workloads and test cases first, followed by interpreting results. Section 6 derives strategies and recommendations regarding HBM usage, concluded by Section 7 with a summary.

2 RELATED WORK

Joining two relations in a DBMS is one of the most common tasks in database history. Recent trends try to fit existing algorithms to new hardware opportunities or even create new operations to benefit from hardware progression.

A big step regarding hardware are main memory databases, where full relations are held in main memory, avoiding costly disk accesses. Schuh et al. [19] compared recent main memory join algorithms and implementations with each other. They state that hardware-conscious joins deliver better performance in general than hardware-oblivious approaches when partitioning strategies are used. In addition, they argue that highly skewed data leads to unequal load balancing in such a way that a simple no-partitioning strategy can beat any partitioning approach in performance numbers.

Additional research focuses on exploiting multicore CPU environments which are absolutely indispensable today. Different strategies are derived how to partition input data and intermediate results as well as distributing work efficiently to multiple threads. Hash joins are commonly found [3, 5, 14] while sort-merge joins only surpass hash joins in terms of performance when input relations are very large (GB) [2].

There are even publications already addressing the Xeon Phi (co-)processors in terms of joining in the past, mainly regarding the high core count and AVX512 instruction set [7, 8, 12]. Since the KNL is released as full-fledged processor, the memory bottleneck between host and coprocessor of previous Xeon Phi versions is finally removed. This bottleneck is still a common problem with GPUs as well [11]. Besides this, additional related work with HBM can be found mostly in the context of High-Performance Computing, like tensor factorization, matrix multiplication, or analytical workloads [4, 17, 20].

When joining data streams directly on the fly, tuple per tuple, other non-blocking algorithms are necessary, though. Teubner et al. [22] developed the HandshakeJoin, optimized for multicore CPUs. Like soccer players shaking hands at the beginning of the match, tuples from two input sources are streamed through the available cores, performing a join locally. Another popular stream join regarding multicores is the ScaleJoin from Gulisano et al [10], providing a new data structure for improved load balancing and efficient scaling to high numbers of threads. Besides utilizing multicore CPUs, there are approaches to efficiently support GPUs or coprocessors as well. An example for this case is the HELLS-Join for heterogeneous hardware environments from Karnagel et al. [13]. It moves some algorithmic parts to devices different from CPU, increasing throughput and latency.

3 MEMORY HIERARCHY

Regarding the heterogeneous memory landscape, the variants can be classified into different levels in terms of access latency, capacity, price, or persistence, together forming a memory hierarchy (see Figure 1).

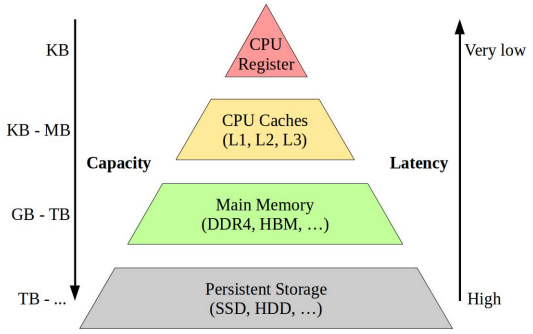

Figure 1: Memory Hierarchy

HBM usually has comparable latency for reads and writes to regular DDR4. Its strength lies in memory bandwidth, which determines performance for memory-bounded applications. Even for multithreaded executions of code with high numbers of threads, like on GPUs, HBM can enhance processing times by reducing memory stalls of threads, waiting for data.

However, the size of HBM is limited and ranges currently in 16 to 32GB for HBM2 technology. Examples for processors currently using HBM2 are Nvidia Tesla, Nvidia Quadro and AMD Vega on GPU side. For CPUs, to the best of our knowledge only the KNL with its MCDRAM as a HBM variant is currently released, although future CPUs with HBM support are announced. In addition, there is still research going for increasing size and bandwidth, e.g. for the announced HBM3.

As a consequence of the limited size, it is rarely possible to store entire workloads in HBM. Therefore the question arises, which data structures benefit the most from high bandwidth. The expectations lead to highly concurrent, sequentially accessed structures that are placed on HBM, eliminating the bandwidth bottleneck. To point an example, for join processing and the no-partitioning join algorithm [18], the shared hash table accessed by all available threads should be a good candidate for HBM ideally.

In this paper, we used the HBM from the Xeon Phi Knights Landing manycore CPU, the so-called MCDRAM. The next section will describe it in more detail, regarding its configuration and our hardware measurements.

3.1 Multi-Channel DRAM

For our experiments, we use a KNL 7210 with 64 cores, a main memory (DDR4) size of 96GB and a MCDRAM size of 16GB. The KNL is configured in SNC4 mode, that means, the core grid is divided into four NUMA regions with 16 cores each. This allows to use a NUMA scheduling policy as well as reducing worse case latency of memory access, because each core in a NUMA region has to access memory by using the closest memory controllers (DDR4 and MCDRAM).

Property	DDR4	MCDRAM
Size	96GB	16GB
Latency	130-145ns	160-180ns
Peak Bandwidth	71GB/s	431GB/s

Table 1: KNL Memory Properties (SNC4 mode)

We ran the Memory Latency Checker[1] tool from Intel to gather memory access latencies of DDR4 as well as MCDRAM. In addition, the well-known STREAM[2] benchmark (Triad) was used for measuring the bandwidth, compiled accordingly to Intels optimization recommendations. Results show that accessing DDR4 is slightly better in terms of latency, but the peak bandwidth is much more limited than the bandwidth of MCDRAM. Table 1 shortly summarizes the measured numbers.

Besides the numbers, the MCDRAM can be configured in three ways, in Cache, Flat or Hybrid mode.

Cache Mode. In Cache mode, the MCDRAM is not visible to the application, just like other CPU caches and registers. It is treated as a last level cache (LLC), in this case L3, because the KNL cores only use L1 and L2 caches. In this mode, no further code adaptation steps are necessary for utilization.

However, running the MCDRAM as cache does not necessarily improve overall performance of an application. If the same data is predictably or sequentially accessed in memory over and over again while the degree of parallelism is high, it can greatly benefit from the high bandwidth. But if the data is read only once or the application is mostly latency bound, the cache mode harms access time by costly cache misses where the request has to be redirected to main memory, all the way back.

Flat Mode. The opposite configuration, the Flat mode, uses the MCDRAM as one or more separate NUMA memory nodes, depending on the clustering mode of KNL. Without further code modification or memory assignment it is not used at all. Two common tools to manually address the MCDRAM are Numactl[3] and Memkind[4].

When starting an application, it is possible to assign certain NUMA nodes where memory should be allocated by using Numactl. When the MCDRAM NUMA nodes are explicitely selected without using regular DDR4 nodes, all memory allocations have to fit into 16GB or the application will fail (out of memory). Memkind on the other hand allows to address memory directly in the code by providing high-bandwidth allocators as well as fallback policies if there is not sufficient HBM available. The drawback of Memkind is that the programmer has to carefully select data structures that could benefit from higher bandwidth per hand, requiring a good understanding of application code.

If compared to Cache mode, there are no further cache misses than those on L1 and L2, but the programmer itself is responsible for its utilization, creating some kind of optimization problems where to use it.

[1] https://software.intel.com/en-us/articles/intelr-memory-latency-checker
[2] http://www.cs.virginia.edu/stream/
[3] https://www.systutorials.com/docs/linux/man/8-numactl/
[4] http://memkind.github.io/memkind/

Hybrid Mode. The third configuration, the Hybrid mode, is a tradeoff between both, using a fragment of memory as LLC and the rest as manually addressable memory. The ratio is changeable and can be configured in BIOS settings. This configuration is useful if 16GB LLC is not used at all or if there are multiple users, some addressing the MCDRAM explicitely and others do not. For our experiments, we just use Cache and Flat mode to compare DDR4 and HBM usage.

The next section shortly describes the join algorithms that we use for our tests.

4 JOIN PROCESSING

For joining two or more data sources, the algorithms considerably differ between DBMS and DSMS joins. In data streams, tuples arrive sequentially and output tuples should be produced continually as long as input data is available. This leads to non-blocking join algorithms like the Symmetric Hash Join (SHJ) [23]. On the other hand, joins in DBMS are usually optimized to efficiently process huge relations at once. However, even in DSMS it depends on the application if batching up tuples or writing and reading to relational tables is allowed. In such cases, it is also possible to use DBMS joins to a certain degree.

Popular hash join algorithms that we consider for our experiments in addition to the SHJ are the No Partitioning Hash Join (NPJ) [14], the Parallel Radix Join [3] and the Partitioned Hash Join (PHJ) [5]. For sort-merge joins, we regard the M-Pass and M-Way implementations of Balkesen et al. [2] as well as the MPSM algorithm, originally described by Albutiu et al. [1]. Since [2] stated that their M-Way implementation is notably superior to other sort-merge joins, we stick to that algorithm for reasons of space.

The following paragraphs shortly describe the general ideas, differences and expectations when using HBM.

Symmetric Hash Join. The SHJ uses a hash table for each input data stream. When a tuple arrives from the previous data source, it is inserted into the corresponding hash table, probing afterwards with all other hash tables for results. Regarding tuplewise processing, results are produced continually as long as tuples arrive. This differs from DBMS joins where the input is read first and calculating results is started as soon as all input data is available. Without batching input data and sharing the hash tables between multiple threads, the SHJ is usually latency-bound, therefore it should not benefit that much from increased bandwidth.

No Partitioning Hash Join. The NPJ algorithm builds a single hash table for probing by processing the smaller relation with multiple threads in parallel, each thread being responsible for its own batch of input data. Because they access the same data structure when inserting, latches on buckets are necessary to prevent race conditions. However, it is expected that collisions are very uncommon due to the fact that millions of buckets are used. When the build phase is finished and the probing phase starts, threads process the bigger relation in parallel again, probing with the shared hash table and producing output tuples accordingly. Because of the good scaling without much contention regarding high thread counts, the expectations arise that the higher bandwidth should be directly visible in performance numbers.

Partitioned Hash Join. The PHJ first divides the smaller relation into multiple partitions in parallel. In the next step, partitions are assigned to threads, building a hash table for each partition. After that, the bigger relation is partitioned, probing the resulting partitions against all the previously created hash tables. When the partition size is small enough to fit into CPU caches while the partition number does not exceed the TLB size, a notable speedup can be achieved. On the other hand, the PHJ suffers from unequal load balancing on partitions (in cases of data skew) as well as latch contention on partitions in high thread numbers. However, HBM should improve performance like in the case of NPJ before, possibly masking the latch contention to a certain degree.

Parallel Radix Join. This join requires that both relations are stored in a contiguous memory region first. Threads scan over the region they are responsible for, building a histogram that holds the number of tuples per partition. The prefix sum calculated over the histogram points directly to the beginning of the different partitions afterwards. In a second scan, threads finally partition the input data according to the prefix sum, processing the partitions afterwards to build the hash tables in the same way the PHJ does. The partitioning of the second relation as well as probing the hash tables can be done similar. We expect that the HBM should benefit from cache configuration of the MCDRAM because of scanning the input twice.

We tested both Parallel Radix Join implementations for reasons of comparison, originally provided by Blanas et al. [5] as well as an optimized variant of Balkesen et al. [3], which greatly reduces the necessary instructions in terms of function calls.

M-Way Sort-Merge Join. The M-Way join of Balkesen et al. [2] consists out of four steps - partitioning the input, sorting locally, merging and finally joining.

The partitioning is necessary for avoiding synchronization efforts in later phases between threads, allowing them to work on their own partitions undisturbed. Sorting is done afterwards on each partition, efficiently supported by AVX instructions. The merging step uses multi-way merging for balancing bandwidth requirements and computational overhead, leading to a globally sorted relation. The same three steps are done for the second relation. At last, a multithreaded join using single-pass merge is performed.

Regarding HBM, expectations arised that a sort-merge join would ideally scale very well with increased bandwidth, especially for the merging step. The implementation of [2] even used multi-way merging to reduce pressure on memory controllers at the cost of increased computations and task switches. High bandwidth should allow out-of-cache merging to become more efficient, reducing this additional computation effort up to a certain degree.

5 EXPERIMENTAL ANALYSIS

In this section we address our questions arised from the introduction, especially showing the impact of HBM on join processing. In addition, we distinguish between transparent and explicite use of MCDRAM in Cache and Flat mode, described in Section 3.1.

First, we show the used hardware and software for our measurements, followed by a description of our workloads and test cases, finally concluded by results and discussion.

5.1 Setup

For the hardware, we use a Xeon Phi KNL 7210 with 64 cores and up to four threads each. It has 96GB DDR4 memory with additional 16GB as MCDRAM. The processor is organized in tiles, each tile contains 2 cores with a shared L2 cache of 1MB. Each core itself has its own L1 cache of 32kB. The CPU runs in SNC-4 clustering mode, that means, the core tiles are separated into four NUMA quadrants, allowing NUMA-aware code to run on dedicated nodes. The MCDRAM is configured in Flat and Cache mode correspondingly. Furthermore, the KNL fully supports AVX-512 instructions and runs with a peak clock frequency of 1.5GHz (Turbo). Threads are evenly distributed over the cores by using KMP_AFFINITY scattered setting of the OpenMP library.

On software side for the data stream results, we used our own open-source stream processing engine *PipeFabric*[5]. All code is compiled with AVX-512 enabled and Intel compiler version 17.0.6. The relational hash join implementations[6] are mostly from the paper of Blanas et al. [5], where the configurations are adapted to the KNL settings, compiled with the Intel compiler and also with AVX-512 enabled. In addition, we used the optimized parallel radix join and the M-Way sort-merge join[7] of Balkesen et al. [2, 3]. Since their code has a lot of intrinsic functions for AVX/AVX2, we did not further optimize for AVX512 which is beyond the scope of this paper. Further configurations like cache or TLB parameters are adapted accordingly, though. This allows us to pin down our results by directly comparing intermediate measurements.

5.2 Workload and Test Cases

For the streaming section, we first measured the potential of HBM compared to regular DDR4 memory. We allocated tuples (consisting out of an integer, double and string value) from our data source directly in HBM with Memkind API and with standard C++ allocator for DDR4. After that, these tuples are streamed through a trivial selection operation with 50% selectivity. The next test case joins two data stream sources by using the SHJ algorithm, producing an average of 10 output tuples per input tuple. All results are fully materialized in the corresponding memory.

The workloads for relational joins followed the data generation of [5] with 16M tuples on build and 256M tuples on probe side. Each tuple is a <key, payload> pair, where the key and the payload both consist out of 8 byte. This leads to a total workload size of 256MB for R and 4GB for S. We generated the provided uniformly distributed data first. The skewed dataset follows a Zipf distribution with s=1.05 (low skew) and s=1.25 (high skew), generated with numpy package of Python. A separate test case is used for materialization efforts on the relational join algorithms of [5], otherwise output tuples are not materialized in memory. The joins, already shortly described in the previous Section 4, are run on KNL with Numactl library fully allocated on MCDRAM or regularly run on DDR4 memory with and without MCDRAM as a cache. In all cases, the input relations are generated and stored in main memory before the measurements are started, meaning that all threads have to initially read tuples from DDR4 or HBM for further processing.

[5] https://github.com/dbis-ilm/pipefabric
[6] http://web.cse.ohio-state.edu/~blanas.2/
[7] both found on https://www.systems.ethz.ch/projects/paralleljoins

5.3 Data Streaming

To point out the possible difference regarding performance of HBM and DDR4, we run a query with selection operator on a single input data stream. Because a stream can be simply endless, we calculate the average processing rate in necessary CPU cycles per tuple until the rate does not change anymore. The query is executed in parallel independently by an increasing number of threads. Figure 2 shows the results.

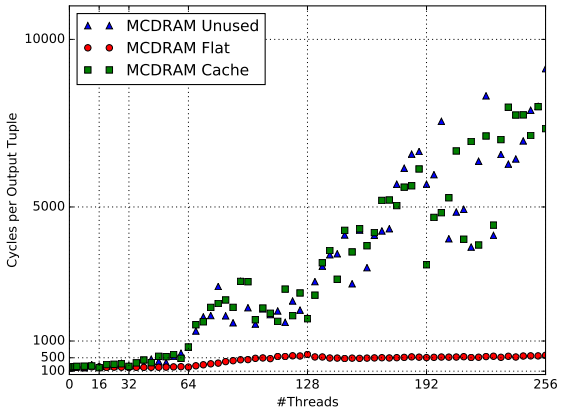

Figure 2: Tuple Allocation with Selection Query

Even with 256 threads executing the query with tuples allocated in HBM, the necessary cycles per tuple stays in low numbers of around 500 CPU cycles for each thread. When the MCDRAM is run as LLC or is not used at all and after running 64 threads in parallel, they begin to stall, waiting for the memory controllers to finish their requests. This problem gets just worse when more threads are added, increasing the average cycles per processed tuple up to more than 10 times. Since tuples are streamed from memory and are processed just once, the MCDRAM in Cache mode does not benefit from the higher bandwidth because the tuples are never reused again.

In the next step, we analyze the SHJ performance by joining two data streams. The join query is executed by an increasing number of threads independent from each other simultaneously, simulating Inter-Query parallelism. The tuples as well as the hash tables are fully allocated on MCDRAM. Because of general window semantics in data stream processing, discarding older tuples after time, 16GB of MCDRAM are enough for this task. Our measurements are shown in Figure 3.

For our implementation of the SHJ, the algorithm cannot benefit from higher bandwidth at all. There are randomized data accesses for the hash table as well as tuplewise processing, suppressing any notable advantage of HBM. Only for 64 threads running a join, the allocations in MCDRAM decrease the average cycles necessary for an output tuple by around 15%. This is the case where each thread runs on a single core with minimal cache thrashing on L2 and no cache thrashing on L1 by other threads.

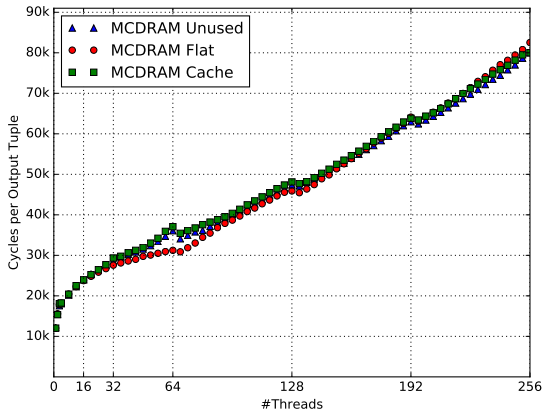

Figure 3: Symmetric Hash Join

5.4 Relational Hash Joins

As pointed out before, these joins can be applied in DSMS as well if blocking like batching or read/writes to tables are allowed. First, we run the NPJ with uniform dataset on the KNL processor by using the DDR4 memory and HBM by Numactl allocation. Results are shown in Figure 4.

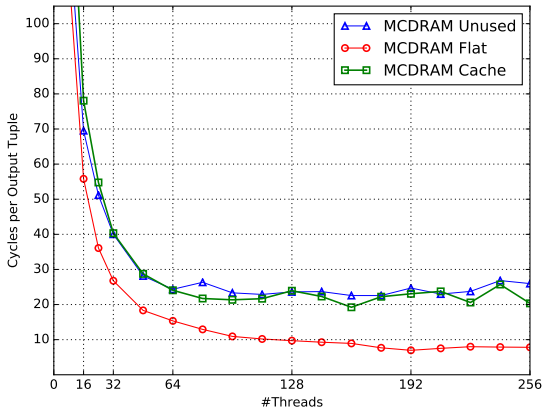

Figure 4: No Partitioning Join

The necessary cycles per output tuple decreases significantly by adding more and more threads. After 16 threads participate in processing the same join, the allocation in HBM allows them to actually benefit from the higher bandwidth. With less than 16 threads, the high bandwidth is not utilized at all and the slightly higher latency of MCDRAM access just worsens the overall performance. Between 16 and 64 threads, the neccessary cycles per tuple drop very fast, which can be explained simply by the KNL architecture, having 64 cores. We first schedule a thread to a single core, where it has no side effects from other threads, minimizing cache thrashing and context switching. Beyond the KNL properties, the NPJ scales very well in general even with hyperthreading enabled [5].

When we compare the cycles at 192 threads, the join needs around 3.5 times more cycles on DDR4 than on HBM, greatly speeding up execution time. Running MCDRAM in Cache mode is not useful at all for the NPJ, suffering the same penalties in randomized access like the SHJ does.

In the next test case, we used the PHJ with uniform dataset as well. Compared to the NPJ, both input data is first partitioned into a fixed number of partitions by all threads, joined afterwards. Results are plotted in Figure 5.

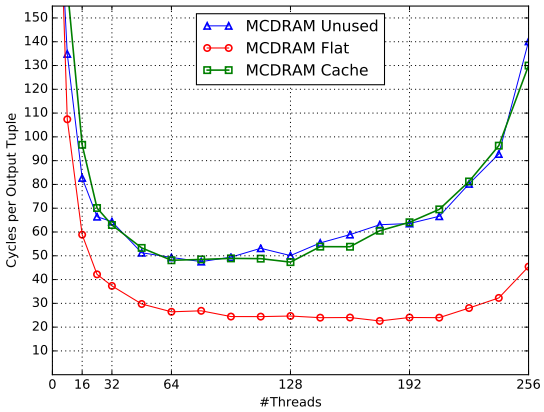

Figure 5: Parallel Hash Join

Interestingly, the performance gets worse after hitting a local minimum in cycles when adding more threads to the same work. This can be explained by examining the implementation and algorithm. In the partitioning phase, all threads can access all partitions, that means, the partitions have to be protected by latches. As fine-granular measurements show, after hitting around 80 threads, the time necessary to partition the input data begins to increase again, finally taking up to three times longer with 256 threads. Under high contention, locks and latches can close to nullify any throughput as already shown by Yu et al. [24]. With multiple queries and workloads running in parallel, it is not possible for a single query to utilize all 256 threads. Therefore, the degree of contention should stay in low numbers in practice.

While the high bandwidth of HBM can counter this progression to some extent in general, it still suffers from high contention at a high thread count. Like the NPJ, the PHJ can greatly benefit from HBM, up to 3x less cycles than DDR4 allocations. However, in absolute numbers the NPJ needs less cycles to finish after utilizing 16 threads at least.

Next, we analyzed the parallel radix join on KNL, again using the uniform dataset. By processing the input in multiple passes, cache and TLB misses are minimized. The results for the implementation of [5] are shown in Figure 6.

The measurements point out that performance only marginally improves after 32 to 64 threads are applied. With less threads it can beat the NPJ in performance, but after hyperthreading kicks

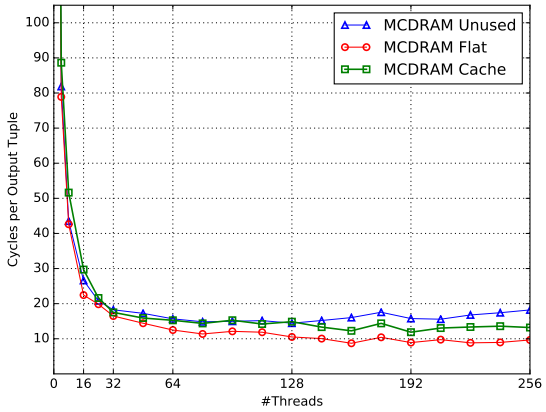

Figure 6: Radix Join

in, the NPJ needs slightly less cycles per tuple in average to finish. However, with MCDRAM as cache, a performance gain can be achieved. Because of the radix join reading the data multiple times for calculating the histogram and partitioning afterwards, it benefits from the data already copied into the cache. Allocating everything directly in MCDRAM is still a further improvement, because there are no cache misses in MCDRAM and no redirection to DDR4 again. Overall, the radix join cannot benefit that much from higher bandwidth like NPJ and PHJ do, but there is still a performance gain up to 1.9x over DDR4 possible.

Finally, we used the optimized parallel radix join implementation of [3]. They reworked the code of [5] in such a way that around 40% of the original function calls in the partitioning phase could be avoided, greatly speeding up the necessary execution time even in the join phase. Results of our measurements can be found in Figure 7, split into cycles per tuple and overall throughput.

The join uses 18 radix bits and is further adjusted to the KNL architecture in terms of cache and TLB parameters. Interestingly, the Flat mode of the KNL could not been verified in our measurements for this implementation, because any allocations per Numactl lead to extremely high cycles necessary per tuple. Nevertheless, the MCDRAM in Cache mode improves performance by a great amount, up to 3.9 times for 256 threads.

5.5 Relational Sort-Merge Joins

We use the M-Way sort-merge join of Balkesen et al. [2] as an example for our tests, since they stated that the join is superior to the other sort-merge algorithms of their paper in most of the cases. They provide some additional parameters to configure, like the partitioning fanout and the buffer size of M-Way merging. Since the maximum threads supported by our KNL is 256 and the L2 data TLB has a maximum of 256 pages, the partitioning fanout is equal to the amount of threads we use. Varying the buffer size for the merging FIFO queues delivers different results when using HBM or DDR4 (see Figure 8).

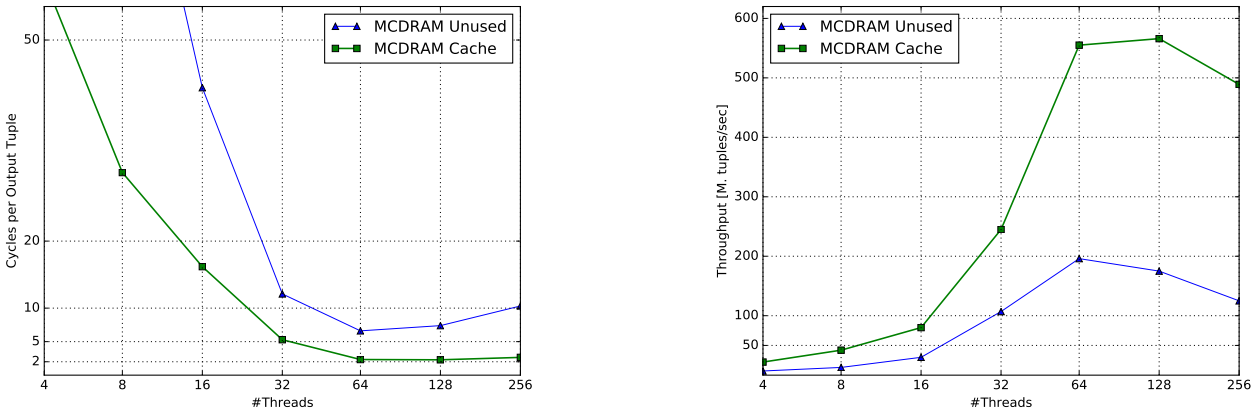

Figure 7: Performance of the Parallel Radix Join of [3]

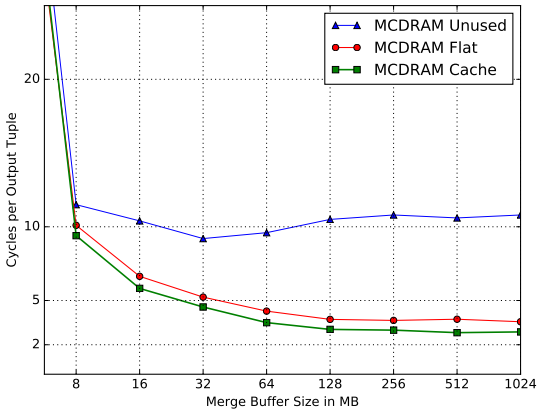

Figure 8: Variation of Sort-Merge Buffer Size

Please note that the x-axis is evenly distributed. If the size of the merge buffer (realized by FIFO queues) is too small, the amount of task switches between the merge tasks inside of a thread limits massively the overall performance. On the other hand, if the size gets too big, we get out of memory eventually and the demands on memory bandwidth increases more and more. For DDR4, there is a sweet spot by around 32MB buffer size, where computational efforts and bandwidth requirements get in balance. However, for HBM in Cache or Flat mode there is no best size in performance like with DDR4, but there is no notable improvement anymore after reaching 128MB size. This means that the full bandwidth of HBM is not utilized at all, else when hitting a certain threshold the limit on bandwidth would increase the necessary cycles per tuple (like DDR4 does). After the FIFO queue gets big enough there are no more task switches, which leads to no further improvement of performance when the size is continually increased.

Moreover, the Cache mode delivers slightly better numbers than in Flat mode. That is the case when some part of the implementation is latency bound and suffers from HBM access penalties when everything is fully allocated in MCDRAM without touching DDR4 anymore. To sum it up, the buffer size is set to 32MB for DDR4 allocations and 128MB for HBM usage as cache or in Flat mode. The results of the executed sort-merge join is shown in Figure 9.

Again, we show the results in terms of cycles per tuple and throughput. It achieves a good scaling until hyperthreading kicks in (after 128 threads). With the MCDRAM running as cache or being explicitely addressed improves performance up to 2.5 times on 32 threads. Both modes of the MCDRAM are relatively equal regarding results, even if the Cache mode is better on high thread counts because of the same reasons like from merge buffer results. A deeper look into the numbers of the different stages shows that the time spent for sorting is greatly reduced by running a huge amount of threads. However, the time necessary for merging is doubled between 128 and 256 threads, leading to worse results afterwards.

A short summary of findings is given in Table 2.

Algorithm	Findings
SHJ	- Latency bound without batching/tables - HBM allocations should be avoided in general
NPJ	- Good scaling of threads - HBM as cache should be avoided
PHJ	- Suffers from latch contention (many threads) - HBM as cache should be avoided
Radix	- Small benefit of HBM overall - Reading input twice improved by Cache mode
Opt. Radix	- Great performance of HBM, even as cache - More than one thread/core should be avoided
M-Way	- Great performance of HBM overall - Use HBM as cache with hyperthreading

Table 2: Findings

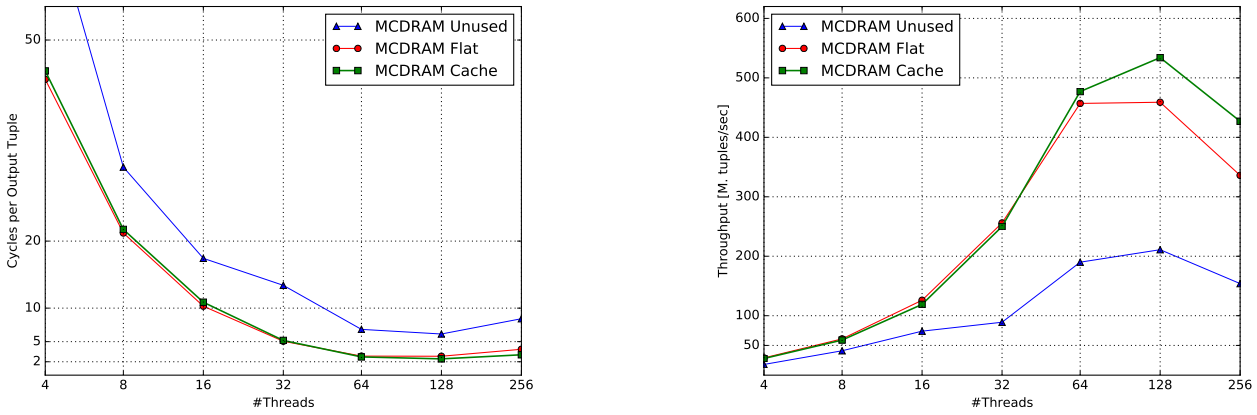

Figure 9: Performance of the Sort-Merge Join (M-Way) of [2]

5.6 Skewed Datasets

As already described, we use two skewed datasets with a Zipf distribution of s=1.05 and s=1.25. The NPJ actually benefits from skew because all threads build one big hash table in parallel without being influenced by uneven load partitioning, also mentioned by Blanas et al. [5]. Therefore we show the skewed datasets for the PHJ, where it leads to uneven partition sizes and unequal loads, using DDR4 only or HBM only. Measured results can be found in Figure 10.

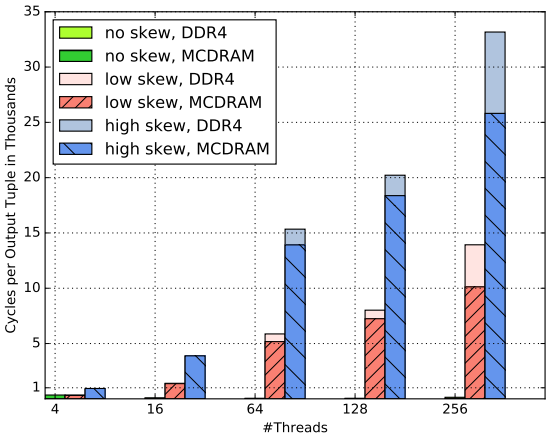

Figure 10: PHJ with skewed data

The bars of DDR4 and MCDRAM are above of each other to easily compare HBM advantage (showing cycles with and without using it). Higher skew increases overall cycles necessary to finish the join by an intense amount, as expected. HBM only helps at the beginning, when partitions are processed simultaneously by many threads. But after more and more threads finish their work, runtime is determined by the few large partitions left where threads are still processing tuples. At that point, only fractions of available bandwidth are used and the HBM advantage is fully lost.

5.7 Output Materialization

When output tuples are materialized in main memory and not discarded, additional costs occur. Blanas et al. [5] pointed out that materialization does not significantly impact the join performance at all. We tested the three relational join algorithms from [5] with and without materializing output tuples in memory, results are shown in Table 3.

The individual overhead in cycles is notably higher when only a single or few threads (less than 8) are used. This is somehow obvious because if cores are idling, it takes more CPU cycles for the running cores to write the same amount of output tuples into memory. If the output is materialized in HBM, it takes slightly more time than in DDR4. This can be explained through the lower execution times per tuple (around 10 CPU cycles). Adding an overhead of 5 to 10 cycles variies more in percentage than adding that overhead to 30 CPU cycles.

The important finding of this section is the fact that materializing output tuples on HBM or DDR4 is not very different in terms of CPU cycles as well as not determining the execution time that much. Because of almost equal read write latencies of both memory types, higher bandwidth does not improve output materialization at all. Therefore it depends on following operators if they can benefit from their input tuples being allocated in HBM or in regular DRAM.

	MCDRAM		
Join Algorithm	Flat	Cache	Unused
NPJ	32%	21%	20%
PHJ	14%	13%	14%
Radix	39%	35%	31%

Table 3: Overall materialization overhead

6 LESSONS LEARNED

(1.) Directly addressing HBM improves performance better than using it transparently like a cache. The obvious reason is that costly cache misses can be avoided. In addition, when data is read only once without reusing, caching effects cannot compensate the initial misses. But even with multiple scans in the radix join, performance is wasted compared to directly allocating data in HBM. Only in the situation of sort-merge joins where data is heavily reused and with hyperthreading enabled the cache can outperform allocations that are directly done in HBM.

(2.) Even if initially CPU bound, high amounts of threads can easily lead to memory bottlenecks. When algorithms are executed with multiple threads, processing data simultaneously and splitting work between them, memory requests at the same time are rising obviously. When CPUs with high thread counts available like the Xeon Phi series from Intel are used, this can limit the effective degree of parallelism. In such cases HBM can improve performance where it does not when the thread count for the same problem is low. Overall latency on the other hand does only decrease if memory bandwidth is saturated and threads start to idle, waiting the memory controllers to finish their requests.

(3.) Do not place everything in HBM. Compared to regular main memory, the capacity of HBM is much smaller. Workloads easily exceed 16GB in size, especially when intermediate results are stored for further processing. However, if the data ranges in size of a few MB, the L2 cache will effectively provide access already, leading to no notable improvements in performance. For stream processing with window semantics huge amounts of data can more easily be handled than in DBMS, because the amount of valid tuples considered for queries can be varied.

(4.) Random memory access patterns do not saturate bandwidth in general, being therefore not ideal for HBM. Sequential access patterns allow prefetchers and out-of-order cores to generate much more memory requests at a time. Random accesses on the other hand depend mainly on memory latency, where the MCDRAM is just a little worse than DDR4. This means that when joining two relations partially or fully stored in HBM the bandwidth can effectively being used, while probing hash tables in HBM only benefits when the thread count is very high. The same holds for data streaming purposes. Tuples allocated in HBM keep multi-threaded latency low, while tuplewise accessed hash tables do not gain improvements at all.

(5.) High bandwidth improves highly parallel hash joins and sort-merge joins more or less equally. Sort-merge joins can benefit very well from high bandwidth in their sorting and merging phase. However, the parallel radix join also benefits nicely from HBM in such a way that it still delivers better performance than the sort-merge join for our workload. Of course this depends on the individual implementation and optimizations, but for reasons of comparison, we used implementations from the same source for our tests.

(6.) Uneven load balancing by skew as well as latches on partitions are not noticeably influenced by HBM. While the high bandwidth can somehow mask the problem by allowing concurrent threads to finish their work faster, the general problem still is not solved. This leads to the advice that the NPJ is preferable in many-core environments simply because it does not suffer data skew at all, which is very common in real systems. This is somehow contrary to the findings of [7], however, they only tested with uniform datasets and without explicitly addressing the MCDRAM beyond cache mode. Nevertheless, if load balancing is handled correctly, it depends on the algorithm if it benefits from HBM, as we showed in our experiments.

7 CONCLUSION

HBM is another layer inside of the already heterogeneous memory hierarchy, which provides higher bandwidth with close to equal latency compared to regular main memory. The limited size, however, allows to store only fractions of data inside of it. With high thread counts in GPU and manycore CPUs, main memory controllers can fastly being overburdened with massive simultaneous memory requests even on regular database operations. The high bandwidth of HBM allows to overcome this bottleneck easily, but requires careful tuning which data structures can benefit the most.

With the recent CPU trend going to more and more cores on a single chip, HBM technology is likely to be focused in the next future, like the announced Stratix 10 MX FPGA with HBM support from Intel or the ACAP from Xilinx.

In this paper we investigated the influence of HBM on different database join algorithms and implementations. We distinguished between tuplewise stream processing and joining two relational tables. Mainly because of comparison reasons, we used different open source join implementations to analyze their behaviour under highly parallel execution and HBM support. The Xeon Phi KNL allows us to measure results without dealing with network delay between sockets and scaling up to 256 threads on a single chip. Besides that, with its cluster configuration we are able to run NUMA-aware code in different regions of the CPU.

The question if sort-merge joins could finally outperform hash joins when HBM is used can be answered with no, but in terms of performance, is not far away. However, since the parallel hash joins also include partitioning steps, they also benefit very well from higher bandwidth. Additionally, we regarded workloads with different skew levels as well as output tuple materialization. Results show that HBM has its use under skewed data, but is also far away from a full utilization, since unequal load leads to few long-running threads that cannot saturate bandwidth.

Finally, we summarized our lessons learned to top this paper off.

REFERENCES

[1] Martina-Cezara Albutiu, Alfons Kemper, and Thomas Neumann. 2012. Massively Parallel Sort-merge Joins in Main Memory Multi-core Database Systems. *Proc. VLDB Endow.* 5, 10 (2012), 1064–1075.

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-core, Main-memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (2013), 85–96.

[3] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-Memory Hash Joins on Multi-Core CPUs: Tuning to the Underlying Hardware. In *Proceedings of the 2013 IEEE International Conference on Data Engineering (ICDE '13)*. IEEE Computer Society, 362–373.

[4] Taylor Barnes, Brandon Cook, Jack Deslippe, Douglas Doerfler, Brian Friesen, Yun (Helen) He, Thorsten Kurth, Tuomas Koskela, Mathieu Lobet, Tareq Malas, Leonid Oliker, Andrey Ovsyannikov, Abhinav Sarje, Jean-Luc Vay, Henri Vincenti, Samuel Williams, Pierre Carrier, Nathan Wichmann, Marcus Wagner, Paul Kent, Christopher Kerr, and John Dennis. 2016. Evaluating and Optimizing the NERSC Workload on Knights Landing. In *Proceedings of the 7th International Workshop*

on Performance Modeling, Benchmarking and Simulation of High Performance Computing Systems (PMBS '16). IEEE Press, 43–53.

[5] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. 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 (SIGMOD '11)*. ACM, 37–48.

[6] Sebastian Breß. 2014. The Design and Implementation of CoGaDB: A Column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* 14, 3 (2014), 199–209.

[7] Xuntao Cheng, Bingsheng He, Xiaoli Du, and Chiew Tong Lau. 2017. A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, 657–666.

[8] Xuntao Cheng, Bingsheng He, Mian Lu, Chiew Tong Lau, Huynh Phung Huynh, and Rick Siow Mong Goh. 2016. Efficient Query Processing on Many-core Architectures: A Case Study with Intel Xeon Phi Processor. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 2081–2084.

[9] Franz Färber, Norman May, Wolfgang Lehner, Philipp Große, Ingo Müller, Hannes Rauhe, and Jonathan Dees. 2012. The SAP HANA Database – An Architecture Overview. *IEEE Data Eng. Bull.* 35 (2012), 28–33.

[10] Vincenzo Gulisano, Yiannis Nikolakopoulos, Marina Papatriantafilou, and Philippas Tsigas. 2015. ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join. In *2015 IEEE International Conference on Big Data (Big Data)*. 144–153.

[11] Jiong He, Mian Lu, and Bingsheng He. 2013. Revisiting Co-processing for Hash Joins on the Coupled CPU-GPU Architecture. *Proc. VLDB Endow.* 6, 10 (2013), 889–900.

[12] Saurabh Jha, Bingsheng He, Mian Lu, Xuntao Cheng, and Huynh Phung Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* 8, 6 (2015), 642–653.

[13] Tomas Karnagel, Dirk Habich, Benjamin Schlegel, and Wolfgang Lehner. 2013. The HELLS-join: A Heterogeneous Stream Join for Extremely Large Windows. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware (DaMoN '13)*. ACM, Article 2, 2:1–2:7 pages.

[14] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2015. Massively Parallel NUMA-Aware Hash Joins. In *In Memory Data Management and Analysis*. Springer International Publishing, 3–14.

[15] Gabriel H. Loh. 2008. 3D-Stacked Memory Architectures for Multi-core Processors. *SIGARCH Comput. Archit. News* 36, 3 (2008), 453–464.

[16] Stefan Manegold, Peter Boncz, and Martin Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. on Knowl. and Data Eng.* 14, 4 (2002), 709–730.

[17] Ivy Bo Peng, Roberto Gioiosa, Gokcen Kestor, Erwin Laure, and Stefano Markidis. 2017. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. *CoRR* (2017).

[18] Orestis Polychroniou, Arun Raghavan, and Kenneth A. Ross. 2015. Rethinking SIMD Vectorization for In-Memory Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, 1493–1508.

[19] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, 1961–1976.

[20] Shaden Smith, Jongsoo Park, and George Karypis. 2017. Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS '17)*. 1058–1067.

[21] Michael Stonebraker and Ugur Cetintemel. 2005. "One Size Fits All": An Idea Whose Time Has Come and Gone. In *Proceedings of the 21st International Conference on Data Engineering (ICDE '05)*. IEEE Computer Society, 2–11.

[22] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, 625–636.

[23] Annita N. Wilschut and Peter M. G. Apers. 1991. Dataflow Query Execution in a Parallel Main-Memory Environment. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems (PDIS '91)*. IEEE Computer Society Press, 68–77.

[24] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (2014), 209–220.

[25] Shuhao Zhang, Jiong He, Bingsheng He, and Mian Lu. 2013. OmniDB: Towards Portable and Efficient Query Processing on Parallel CPU/GPU Architectures. *Proc. VLDB Endow.* 6, 12 (2013), 1374–1377.