Stream Processing on High-Bandwidth Memory

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

ABSTRACT

High-Bandwidth Memory (HBM) provides lower latency on many concurrent memory accesses than regular DRAM. This is especially useful on GPUs, where thousands of lightweight threads access data on shared memory at the same time. On regular multicore CPUs, the degree of multithreaded parallelism is usually not high enough to improve performance noticeably by using HBM. However, with an increasingly rising core count inside CPUs, especially manycore processors like the Xeon Phi Knights Landing (KNL) from Intel, the properties of HBM are more and more interesting to exploit.

In this paper, we want to analyze the impact of HBM for data stream processing, notably multithreaded hash joins over several input data streams as well as tuple allocation and aggregation within this memory technology. Our results show improvements on the tuple processing rate by up to a magnitude when concurrently accessed data is stored in HBM on chip instead of DDR4, considering different HBM configurations of the KNL.

Keywords

HBM, MCDRAM, Stream Processing, Xeon Phi, KNL

1. INTRODUCTION

A widely known classification for performance analysis is the division into memory-bound or CPU-bound applications, allowing to tackle the right spots for optimization. In today's systems, parallelism is the key phrase to improve speedup for CPU-heavy software, which can be applied on different levels. Multithreading is a very common paradigm for executing independent computations in parallel, scheduled by the OS, reaching high degrees of concurrency through multicore CPUs and GPUs.

However, with an increased number of threads usually the number of concurrent memory accesses is raised at the same time. Regular memory controllers to main memory are capable of dealing with multiple memory requests of concurrent threads. They reach their limit on bandwidth very fast, though, especially with high numbers of threads, which is very common in GPUs and manycore CPU architectures.

That is a reason for HBM development, which provides a much higher available bandwidth for parallel memory requests, overcoming the memory wall [15]. While HBM is used regularly in GPUs, it is not very common in CPUs. With the latest release of a manycore processor in the Xeon Phi product line from Intel, namely Knights Landing (KNL), 16GB of Multi-Channel DRAM (MCDRAM) are added as HBM on chip. For comparison, today's DDR4 SRAM reaches around 90GB/s as available bandwidth, while the MCDRAM has a peak performance of up to 420GB/s with slightly worse latency [10]. This allows to fulfill much more memory requests at the same time, but the capacity is limited. Therefore it is not an option to just store everything in HBM, not to mention the increase of latency.

With this paper, we want to tackle the following questions:

- How good is the performance gain of HBM compared to regular DDR4 when used for processing of data streams?
- Which data structures benefit the most when stored in HBM, e.g., tuples, states, or hash tables?
- Where is the break-even point of multithreaded memory access, until which a performance gain is achieved compared to main memory DDR4 SRAM?

2. RELATED WORK

Since the MCDRAM was introduced by the Xeon Phi KNL processor in 2016, some research already has been done related to the KNL in other fields of science, like high performance computing or machine learning. Most of the papers try to determine the impact of a manycore CPU to their applications.

Smith et al. [11] used the KNL as a case study for tensor factorization on manycore CPUs. They placed different data structures on the MCDRAM as well as changing its configuration, exploring the influence of HBM on calculation efficiency compared to regular DDR4 SRAM. The results pointed out that the algorithm performs up to 30% better when the MCDRAM is manually addressed instead of being used as a low level cache (L3).

Barnes et al. [2] applied a huge set of workloads (called the NERSC workload) on the KNL. The algorithms performed best when data fully fits inside of MCDRAM when parallelization is possible.

30th GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 22.05.2018 - 25.05.2018, Wuppertal, Germany.
Copyright is held by the author/owner(s).

CEUR-WS.org/Vol-2126/paper6.pdf

Cheng et al. [3] investigated main memory hash join performance for traditional databases when executed on the KNL processor. Their results show that the MCDRAM can greatly contribute to hash join performance only if being manually addressed (flat mode), else it is underutilized.

However, joins on a DSMS differ fundamentally from join algorithms of a DBMS. A major difference is the streaming property where tuples arrive continuously, therefore, a join operator has to process tuples unblocked. This means that it cannot simply wait until all data has been read before producing results, hence certain stream join algorithms have been developed.

One of the first important algorithms for joining data streams was the Symmetric Hash Join (SHJ) [14], published around 1991, later further refined as XJoin [13], which presents a solution for cases where the hash tables do not fit in main memory. The adaptation of algorithms to new hardware further progressed in the last decade. Examples for this progression are the CellJoin [4], developed for the cell processor, the HandshakeJoin [12] as well as the ScaleJoin [5] for multicore CPUs, or the HELLS-Join for heterogeneous hardware environments like CPUs coupled with GPUs [6].

3. PROCESSOR AND MEMORY

Due to the fact that hardware and software improves by technological advance over time, applications try to adapt as effectively as possible for better performance or new possibilities. Two main categories regarding hardware are processors and memory. Both of them have a very heterogeneous landscape in terms of available variants. To name a few, there are CPUs, GPUs, coprocessors, FPGAs, DSPs as well as registers, caches, DDR4, HBM, NVRAM and many more. A recent trend goes to manycore CPUs with integrated HBM, discussed further in this section.

3.1 Manycore CPU

After hitting the CPU clock rate wall at around 4 GHz and developing processors to use multiple cores on a single chip, a direction goes to CPUs with more and more smaller cores, so called manycore architectures. While hundreds and thousands of (lightweight) cores are already common in GPUs, regular CPUs are far off from such numbers. For multithreading, performance improvements depend mainly on the possible degree of parallelism of applications.

Both GPU and CPU threads have their own advantages and disadvantages for parallelism, though. GPU threads are grouped together into warps. All threads inside a warp perform the same instructions simultaneously, ideally repeating them multiple times, leading to an intense amount of parallelism. However, if somehow a thread inside a warp has to change instructions, maybe because of branching code, its instructions get serialized, losing performance [8]. In addition, the GPU is not used for all instructions of an application, just for the parallel parts. Therefore, data needs to be transferred between the CPU (host) and the GPU for processing by its threads, increasing execution time by transfer delay.

On CPU side, a main challenge lies in efficient scaling of applications to hundreds of threads. This is a common problem of traditional databases when using manycore processors, for instance in terms of concurrency control [16]. If the trend to manycore processors continues, there has to be some serious redesign on databases if this hardware should be used efficiently.

A well-known example for manycore CPUs is the Xeon Phi product line from Intel. The first Xeon Phi, called Knights Ferry, was released 2010 as a prototype for research purposes and not commercially available. Knights Ferry is a coprocessor, that means, it needs a host system with a regular CPU, like GPUs also do.

The successor was released 2012, called Knights Corner (KNC), still as coprocessor only but commercially obtainable. For database usage, the main bottleneck emerges from the PCI connection to the host system, limiting data transfer through offloading to 15GB/s. As a result, the KNC is not often found in used hardware for database systems.

With the latest release, namely Knights Landing (KNL), available since 2016, Intel addressed this bottleneck by discarding the coprocessor design, although the KNL was later additionally released as coprocessor. With a high core count for CPUs (up to 72 cores on chip) as well as HBM for higher memory bandwidth, the KNL got an increased interest for researchers as well as owners of database systems.

3.2 Multi-Channel DRAM

As already stated earlier, there are different variants of memory, leading to the so-called memory hierarchy (see Figure 1). It is always a tradeoff between capacity and access latency (respective the price). The HBM in general has comparable latency to regular DDR4 SRAM but is more limited in size.

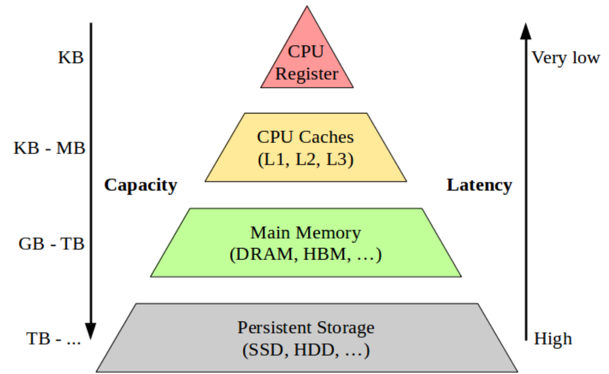

Figure 1: Memory Hierarchy

However, with multithreading purposes comes another requirement to memory, called memory bandwidth. Single-threaded applications mostly do not saturate the available bandwidth of main memory (for DDR4 around 90GB/s). Nevertheless, multiple threads accessing memory in parallel can reach this limit very quickly, especially on manycore architectures where hundreds of threads can be executed simultaneously. This is also a common problem on GPUs, therefore, the HBM was developed years ago.

To overcome this limitation for intense multithreading on CPUs, the KNL utilizes its own HBM on chip, the so-called Multi-Channel DRAM (MCDRAM). The MCDRAM itself is a memory variant specialized for high bandwidth, allowing up to 420GB/s data transfer rates with slightly worse access latency [10] and a maximum capacity of around 16GB. Since

regular CPUs did not have any HBM by default in the past, it opens new possibilities for applications with high numbers of threads.

The utilization of the MCDRAM, however, is no trivial decision. Because of the generally higher accessing latencies, Intel provided three configurations (flat, cache, hybrid) to allow owners to decide where the MCDRAM can be preferably used.

In cache mode, the MCDRAM is not visible to applications. Instead, the operating system uses it as a huge L3 cache, with usual advantages and disadvantages. A disadvantage to remember is the increased latency on a cache miss, where the data has to be retrieved from main memory with detour of the MCDRAM. In flat mode, the MCDRAM can be addressed by the application itself, else it is not used. In this case, the programmer has to decide where the application benefits the most from HBM. The third configuration is a hybrid mode, where the MCDRAM is partly used as cache as well as addressable memory.

It is important to mention that processors of the next generation with more cores (like server CPUs) will very likely use this HBM in addition to regular DDR4 SRAM.

3.3 Summary

With simpler core design compared to current state of the art multicore processors as well as low clock frequencies of 1.5GHz, algorithms and implementations have to adapt to manycore CPU properties to gain any performance advantage. The parallel execution of code along with data partitioning is a key to achieve this goal. By increasing the thread count as well as fitting algorithms to manycore architectures efficiently, memory bandwidth typically becomes a major bottleneck very quickly. Therefore the MCDRAM as a version of HBM provides new possibilities to overcome this gap, supporting three different configurations as a tuning parameter. In this paper we want to analyze its impact and benefit on typical streaming operations and semantics, which are further explained in the following section.

4. DATA STREAM PROCESSING

While relational database systems are still the most common platforms for data storage and processing [7], more and more applications need to handle incoming data directly on the fly, such as IoT, social network or sensor data analysis. Instead of storing everything and processing the data later on, DSMS handle tuples usually with timestamps, allowing to remove outdated data from their system. It is necessary to be able to run queries for long times instead of only once, processing tuples directly after arrival.

To give an example, join algorithms that work well in relational databases (like sort-merge joins) are impossible to use directly in a DSMS because of their blocking property. That means, they can execute only if all the data is stored in main memory, for example to sort them for joining. On data streams possibly never ending, the joins must be nonblocking, in other words, producing results continuously instead of only once.

In this section, we want to give a short overview about our used stream processing engine *PipeFabric*[1] as well as a common join algorithm and window semantics in DSMS, relevant for our experimental analysis with HBM.

[1] https://github.com/dbis-ilm/pipefabric

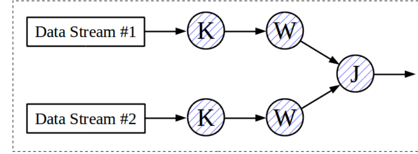

Figure 2: Query with windowed Join in *PipeFabric*

4.1 PipeFabric

PipeFabric is a stream processing engine developed at our research group of the TU Ilmenau. It is open source and fully functional, written in C++. Data streams can be constructed via different network protocols, like AMQP, MQTT or ZeroMQ, to get tuples from different servers like Apache Kafka or RabbitMQ. It is also possible to create finite streams from files or just construct tuples through a generator function.

For queries, common operators are provided, like selection and projection or joins over multiple streams. These operators are connected like a dataflow graph, where input tuples are forwarded between the query operators. This is exemplarily shown in Figure 2, where tuples from two data sources arrive. First, their key attribute (K) is specified, followed by window (W) operators, avoiding a memory overflow caused by infinite data streams as well as keeping only recent data for further computations. After that, the tuples are forwarded to a join (J) operator, which joins tuples according to their keys.

4.2 Window-based Operations

Windows are a very common way to deal with infinite data streams. Usually data streams from various sources like sensor networks produce information continuously over long time periods. Since a few years the cost of memory dropped significantly, however, even if it would be theoretically possible to store all retrieved information, it would be no good idea. Finding correlations in terms of data mining is much more difficult in huge amounts of data, even if some information is already outdated, like sensor measurements weeks ago.

A window holds a certain number of tuples that are currently relevant for queries. There are different window algorithms, like sliding or tumbling windows, determining the data displacement strategy. A sliding window for example invalidates the oldest tuple when a new tuple arrives. The number of tuples a window holds can be fixed, e.g. one million, or time based, where the size changes dynamically.

To invalidate tuples, another common algorithm is the positive-negative approach [1]. When a tuple arrives at a window operator that already holds its maximum capacity, it forwards the new tuple as well as the (labeled) invalidated tuple to the following operator. Depending on the next operators, they can individually react according to their function. A sum over a certain attribute for example can just subtract the value of the invalidated tuple from its aggregation.

For our testing purposes with HBM we used a sliding window operator with fixed length for the input stream. Because of the sequential data access and processing of a window, we expect that it ideally benefits very well from the increased available bandwidth, leading to higher tuple processing rates.

4.3 Symmetric Hash Join

For our measurements in Section 5, we decided to show results based on hashing, like the common Symmetric Hash Join (SHJ). The SHJ [14] is one of the first published join algorithms for processing data streams unblocked. The main difference to hash joins in relational databases is that it produces results continuously for each incoming tuple. As a side note, it depends mainly on the individual scenario of data stream processing if micro-batching strategies are allowed, increasing overall throughput but delaying individual results.

Figure 3 shows the general idea of the join algorithm for two input streams.

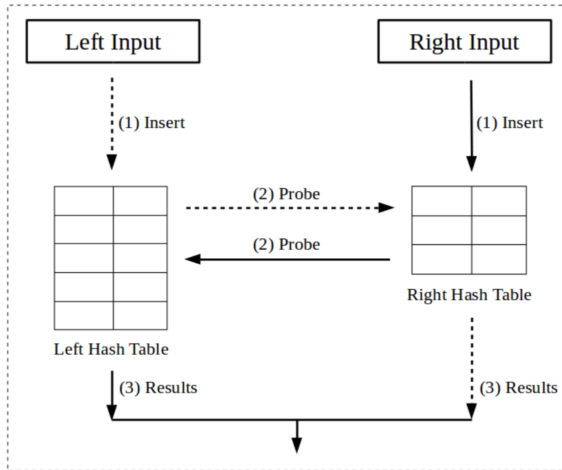

Figure 3: Symmetric Hash Join Algorithm

If a tuple arrives either on the left or the right input stream, it is first inserted into its corresponding hash table. After that, it is probed against the other hash table for matches. For all partners found, tuples are produced and returned to the following operator in a single output stream.

The sizes of the hash tables are dependent on the input tuple rate of the individual stream. If a stream delivers tuples faster than another stream or the same key value occurs more frequently over time, the corresponding hash table is reoptimized regarding its hash function to avoid too many collisions. When an outdated tuple from a predecessing window operator arrives, it already has an entry inside of the hash table. Therefore it is not joined, but removed from the table, leading to no more matches from future tuples of the other stream.

It is also possible to extend the SHJ to $n > 2$ input streams, resulting in n hash tables. Every input tuple is probed against all the other $n - 1$ hash tables after being inserted.

5. EXPERIMENTAL ANALYSIS

Peng et al. [9] pointed out that the memory access pattern has the most influence on benefits of HBM. While we have sequential access for data streams (tuple by tuple), hash joins use random access patterns to find matches in hash tables by probing. Therefore the expectations arised that the performance of our hash join operators will not improve noticeably. Instead, queries processing only a single input stream each, tuplewise or as microbatches, should deliver much better processing rates under high occurrence of threads and many parallel memory requests.

Regarding the MCDRAM configurations, running MCDRAM as huge last level cache is the most trivial way of utilizing it, because there are no changes necessary inside an application to benefit from higher bandwidth. However, it worsens the latency on cache misses, because memory requests cannot go directly from L2 cache to main memory, instead they have to traverse MCDRAM before. The general advice[2] regarding MCDRAM in performance aspects is to choose carefully which data structures should be placed in MCDRAM and which not, using the flat mode. The Memkind API[3] provides HBM allocators for any data type to use HBM instead of regular DDR4 SRAM.

Another possibility addressing HBM is provided by Numactl[4]. Numactl allows applications to run fully on a certain memory device, e.g. on MCDRAM without using DDR4. Because of the goal of our work, analyzing effects of HBM on different individual data structures and operators, we did not use any test results with Numactl in this paper.

In summary, the test cases considered the following options:

- MCDRAM configuration (flat, cache)
- Tuple allocation of data streams
- Window content allocation
- Hash Table allocation of Hash Joins
- State allocation of aggregations

Allocations can be done in MCDRAM as well as in DDR4. All tests use data streams directly streamed from main memory without any I/O from disk, with one million tuples per stream (except for joins) for calculating the tuple processing rate (tp/s) of queries. We also tested higher tuple amounts per stream, but the processing rate (tp/s) did not change, pointing out that a million tuples is enough for calculating rates. With these results, we want to show the influence of HBM on performance regarding different data structures.

5.1 Setup

For our tests with HBM we used a KNL 7210 with 64 cores, 96GB DDR4 memory and 16GB MCDRAM. PipeFabric is compiled with the Intel compiler version 17.0.6 and the AVX512 instruction set support. Threads run in scattered setting, which means that each of the 64 cores gets first a single thread before a core gets a second one. The clustering mode of the KNL runs SNC-4, that means, the core grid is divided into four NUMA sections with 24 GB main memory and 4 GB MCDRAM each.

Tuples use an integer, a double as well as a string value as format. For stable results without too much randomness, the integer is simply counted up from zero modulo 10,000 and declared as key attribute for the join tests.

[2] `https://colfaxresearch.com/knl-mcdram/`
[3] `http://memkind.github.io/memkind/`
[4] `https://www.systutorials.com/docs/linux/man/8-numactl/`

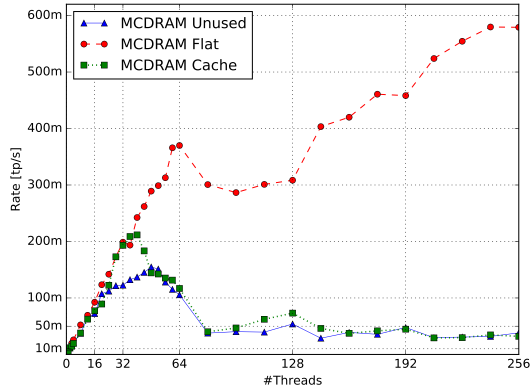

Figure 4: Tuple Allocation Performance

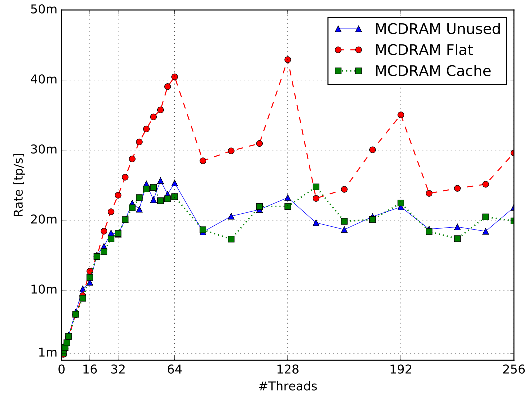

Figure 5: Window Performance

5.2 Tuple Allocation

Our first measurements address the allocation of data stream tuples. Three possibilities can be explored: (1) Storing them in main memory DDR4, (2) allocate them in HBM directly with memkind API, and (3) using MCDRAM as last level cache without further modifications. We tested all three variants regarding tuple throughput for a query with a selection operator (50% selectivity), streaming the tuples right through it in the absence of additional operators.

To utilize the bandwidth and 256 supported threads of the KNL 7210 processor, we run the same query multiple times while increasing the number of OpenMP threads. Each OpenMP thread runs a single instance of the query (Inter-Query Parallelism), leading to a rising number of memory requests. Threads are created with the first *parallel* pragma of OpenMP, therefore we skip the first run for our measurements. Our results are shown in Figure 4 for all three options mentioned above. The abbreviation m expresses a million tuples per second while k stands for thousand tuples per second accordingly.

It can easily be seen that the sequential tuple access creates ideal conditions to saturate the bandwidth. Important to notice is the difference between MCDRAM as cache and being directly addressed. In the latter case there are no cache misses in MCDRAM, saving the detour back to main memory. The increased latency costs of MCDRAM access can be hidden after each core got two threads due to hyper-threading effects, improving performance of MCDRAM in flat mode even more.

5.3 Window Allocation

A window operator stores a sequence of tuples, defining a range in which tuples are relevant for further processing (see Section 4.2). The elements stored in a window can be allocated in main memory and HBM as well. We investigated the performance advantage in the same way like allocating tuples in the last Section.

The window operator uses a sliding window with a size of 100,000 tuples, preventing a possible memory overflow. This is especially useful for the limited space of the MCDRAM to 16GB. In addition, a selection operator with 50% selectivity processes the valid tuples afterwards. The results are shown in Figure 5.

After 20 threads requesting data from memory simultaneously, an advantage between HBM and regular DDR4 can be noticed, up to around 80% with 128 threads. Because of the window semantics, inserting new tuples and removing the oldest ones leads to a predictable memory access pattern. The cache mode cannot compensate misses and the generally higher access latencies with higher bandwidth, leading to no noticeable improvements overall. The MCDRAM seems to show best performance for 64, 128, 192 and 256 threads - where each core has one, two, three or four threads, with equal load on each core.

5.4 Hash Table Allocation

Another chance to use HBM for improved bandwidth is the allocation of hash tables, being used regularly by join operators. Two input streams deliver 100,000 tuples each for our test case. After key specification, the tuples are forwarded to the join operator. Because of the specified workload, one million tuples are produced by joining.

Like the tests before, we run this query in parallel with OpenMP threads to show Inter-Query Parallelism performance. In Figure 6 the results of the join operator are shown.

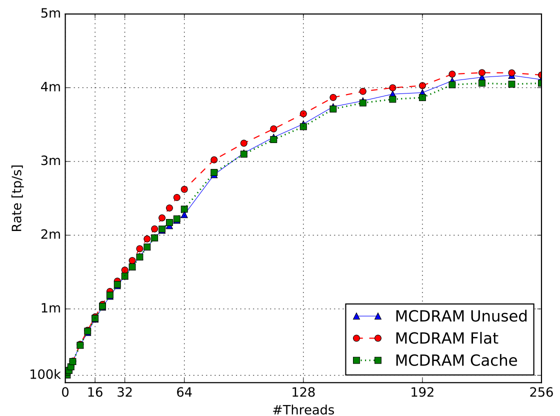

Figure 6: SHJ Performance

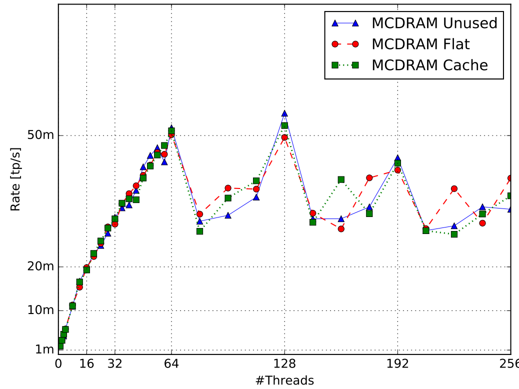

Figure 7: Aggregation Performance

The numbers are quite disappointing, although they met our earlier expectations. With MCDRAM as a cache the rate even gets worse, caused by random memory access and cache misses. But even with directly allocating and updating only hash tables in MCDRAM, the tuple processing rate increases by 15% at most while running 64 threads (one per core).

5.5 Aggregate State Allocation

Finally we want to improve the tuple processing rate of aggregation operators. Each operator holds a state where the aggregate is stored, e.g. in simple cases just a sum, average or current maximum. This state can also be allocated on HBM for increased bandwidth. Figure 7 shows the results of our measurements.

The observation shows that it is just possible to store small states in L1 and L2 cache of each core. This results in no improvement of performance when utilizing HBM instead of DDR4. The peak rates at 64, 128 and 192 threads can again be explained by the scattered thread setting, where threads are distributed evenly to the cores. With 128 threads, each core maximized its parallelism because each core supports two hardware threads without hyperthreading side effects.

6. CONCLUSION

The questions of the introduction can be answered in the following way. First, the performance gain of HBM can increase up to a magnitude when a high count of threads is accessing memory sequentially. Instead of just running HBM as a cache without further modifications, it is absolutely necessary to carefully store suitable data structures in HBM to avoid expensive cache misses.

In addition to this, not all data structures benefit equally from the higher available bandwidth. Random access patterns that are commonly found in hash joins cannot exploit effectively the HBM properties with prefetching mechanisms. On the other hand, predictably removing and adding elements in a window experiences a notable boost in rate performance up to 80%. The ideal case, receiving tuples from a source and storing them directly for further processing in HBM can benefit the most from the higher bandwidth with up to a magnitude higher processing rates, which was quite surprising, though. However, it demonstrates the possible potential when algorithms get further enhanced and optimized for manycore CPUs with HBM support.

To summarize it up, the observations made with data stream processing on the KNL manycore architecture show that there is a huge gap in performance between operators with random and predictable memory access. Especially hash join operators show bad behavior and should ideally be replaced by join operators using more linear accessable data structures. Our future work will draw on these results, attending HBM characteristics for improved data structures and algorithms of database operations to maximize parallel performance.

7. REFERENCES

[1] A. Arasu, S. Babu, et al. The CQL Continuous Query Language: Semantic Foundations and Query Execution. *The VLDB Journal*, pages 121–142, 2006.

[2] T. Barnes, B. Cook, et al. Evaluating and Optimizing the NERSC Workload on Knights Landing. In *PMBS*, pages 43–53, 2016.

[3] X. Cheng, B. He, et al. A Study of Main-Memory Hash Joins on Many-core Processor: A Case with Intel Knights Landing Architecture. *CIKM*, 2017.

[4] B. Gedik, R. R. Bordawekar, et al. CellJoin: A Parallel Stream Join Operator for the Cell Processor. *The VLDB Journal*, 18(2):501–519, 2009.

[5] V. Gulisano, Y. Nikolakopoulos, et al. ScaleJoin: a Deterministic, Disjoint-Parallel and Skew-Resilient Stream Join. *IEEE TBD*, pages 1–1, 2016.

[6] T. Karnagel, D. Habich, et al. The HELLS-join: A Heterogeneous Stream Join for Extremely Large Windows. In *DaMoN*, pages 2:1–2:7, 2013.

[7] V. Leis. Query Processing and Optimization in Modern Database Systems. In *BTW*, pages 507–518, 2017.

[8] A. Meister, S. Breß, et al. Toward GPU-accelerated Database Optimization. *Datenbank-Spektrum*, 15(2):131–140, 2015.

[9] I. B. Peng, R. Gioiosa, et al. Exploring the Performance Benefit of Hybrid Memory System on HPC Environments. In *IPDPSW*, pages 683–692, 2017.

[10] C. Pohl. Exploiting Manycore Architectures for Parallel Data Stream Processing. In *GvD*, pages 66–71, 2017.

[11] S. Smith, J. Park, et al. Sparse Tensor Factorization on Many-Core Processors with High-Bandwidth Memory. In *IPDPS*, pages 1058–1067, 2017.

[12] J. Teubner and R. Mueller. How Soccer Players Would Do Stream Joins. In *SIGMOD*, pages 625–636, 2011.

[13] T. Urhan and M. J. Franklin. Dynamic Pipeline Scheduling for Improving Interactive Query Performance. VLDB, pages 501–510, 2001.

[14] A. Wilschut and P. Apers. *Dataflow Query Execution in a Parallel Main-Memory Environment*, pages 68–77. IEEE Computer Society, 1991.

[15] W. A. Wulf and S. A. McKee. Hitting the Memory Wall: Implications of the Obvious. *CAN*, 23(1):20–24, 1995.

[16] X. Yu, G. Bezerra, et al. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. pages 209–220, 2014.