Exploiting Manycore Architectures for Parallel Data Stream Processing

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

ABSTRACT

With the ever increasing complexity of applications, optimizations on hardware and software have to keep up with associated challenges. On data stream purposes, low latencies and high throughput are key requirements for being able to process continuous incoming data in real time. Achieving a high parallelization degree is an important factor for accelerating query execution time. By hardware side, the emerging manycore processor architectures give new opportunities for exploiting parallelism, like the Xeon Phi series from Intel.

In this work, we transfer our own stream processing engine PipeFabric on a Xeon Phi Knights Landing. We measure latencies for cache and memory as well as throughput of data tuples and compare them to results on a system with a modern multicore processor. Our observations lead to general optimization recommendations on hardware and software level for improved performance on processing continuous data when using a manycore processor. In addition to this, we show that a manycore CPU can surpass a modern multicore CPU easily on query runtime when adapting accordingly.

Keywords

Data Stream Processing, Many Integrated Core (MIC), Xeon Phi

1. INTRODUCTION

High amounts of data with intense arrival rates need to be processed by queries continuously in data stream processing applications. To achieve acceptable throughput rates with low response times, usually a combination of powerful hardware and careful stream processing engine (SPE) design is used.

On hardware side, there are two main ways of improving computational speed, distributed computing and parallel computing. The underlying hardware of distributed computing is usually a number of high-end multicore processors, connected to each other, sharing computational work done. By distributing tasks to many processors instead of just one, each of them can contribute his part to a faster processing speed. For parallel computing, the trend goes to processing units with simpler cores, but many of them inside a processor, called a manycore architecture. The main advantage compared to distributed hardware lies in shorter response time (latency) because of closer distances between processing units, negating any communication needed through network protocol. However, this comes with a cost in terms of lower clock speed, caused by an intense amount of waste heat on small chip area that is impossible to get rid of when clocking high. This results in bad runtimes when only a fraction of available cores is used.

The consequential question is how to utilize as many cores as possible efficiently on data stream processing purposes, to gain a speedup on queries compared against modern multicore processors. We focus on query processing on data streams in parallel, since real-time stream processing has become more and more a promising field of interest in the last years. For this paper, we evaluate stream processing on a manycore processor out of the Xeon Phi series from Intel, codename Knights Landing (KNL). We apply our own SPE PipeFabric both on KNL and a multicore CPU system to compare results and steps of optimization. With PipeFabric, we have full control on different stream operators, allowing to apply different levels of improvements.

The main goal is to investigate the potential and utilization of a manycore processor for the task of parallel data stream processing. To achieve this, we discuss the characteristics of Xeon Phi manycore CPU as well as our SPE, resulting in different possibilities of optimizations on hardware and software. We finally add some of these points on our preliminary experiments and show their impact on performance.

The rest of this paper is organized in the following way. The next Section 2 is about general information on data stream processing along with optimizations recommended to run an SPE in an efficient way on a system with many CPU cores. Section 3 handles important facts of the Xeon Phi manycore processors with regards to optimization opportunities of this architecture. Section 4 presents related work already done in this context with Xeon Phi. Section 5 shows measurements of latency and bandwidth from memory and caches of KNL and i7 multicore processor, as well as preliminary experiments and results on optimizing query processing for manycore CPUs. Finally, Section 6 tops off this work.

29^{th} GI-Workshop on Foundations of Databases (Grundlagen von Datenbanken), 30.05.2017 - 02.06.2017, Blankenburg/Harz, Germany.

2. STREAM PROCESSING ENGINES

Data stream processing consists of one (or more) data streams, providing a potentially endless amount of tuples representing data. Tuples can have different arrival rates and it is mostly impossible to store all of them, therefore a strategy like sliding windows must be applied to deal with outdated information. A query on streamed data applies different operators like joins, aggregations or simpler operations like selections or projections on them, resulting in challenges of speeding up runtime to handle high tuple arrival rates. This speedup can be achieved through partitioning of data or operators, parallelizing the query.

2.1 Inter- vs. Intra-Operator Parallelism

There are two strategies applicable on transaction, query, or operator level for parallelization. For transactions and queries, parallelism is mainly realised through DBMS systems that are not of further interest at this point. On operator level, Inter-Operator Parallelism can be achieved through parallel execution of different operators. However, this is often not completely possible for the simple reason of operator dependencies, e.g. a selection before an aggregation. On the other hand, Intra-Operator Parallelism is realised by creating multiple instances out of one operator, so the instances can share work between them (SIMD principle). To combine results of each instance, an additional merge step is needed. This shows Figure 1 exemplarily.

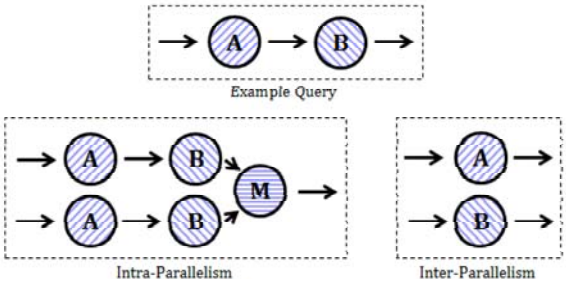

Figure 1: Inter-Operator and Intra-Operator Parallelism

2.2 Cluster vs. Single Node

When partitioning, it is possible to use multiple processors to distribute work. Therefore it is necessary to specify how to split the data and query. Cluster computing frameworks (e.g. Spark, Flink or Storm from Apache) can support this distribution, allowing a fine-grained tuning of the streaming application. However, they suffer from the same problem of all distributed approaches - high latencies, emerging from communication costs caused by sending messages through the network connecting the machines.

To avoid this and to reduce latency, a manycore processor can be used, supporting parallelism through hundreds of local threads. But to overcome the penalty of low clock speed some points have to be regarded. On data stream applications the question arises if it is even possible to gain such a speedup that a manycore CPU can be faster than a single multicore or a cluster of multicores, because of requirements on low latency. Finally, our work aims not only in recommendations concerning optimizations on Xeon Phi, but rather on answering this question.

2.3 PipeFabric SPE

PipeFabric is our own framework written in C++ for data stream processing. A query can be formulated using a DSL and consists of different stream processing operators forming a dataflow graph. It supports next to simple selection/projection operators aggregates, groupings, joins as well as complex event processing. The framework is optimized for low-latency mainly through efficient C++ template programming.

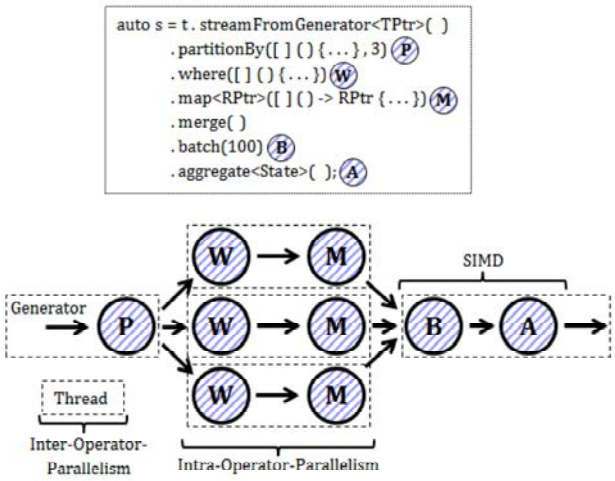

Figure 2: Query example

A simple example for a query shows Figure 2, splitted in C++ example code and the resulting dataflow graph. A generator produces tuples, which are then processed by three selections through partitioning, sorting out tuples that have an uneven number in their first attribute, in addition to three projections (map) afterwards, reducing their number of attributes. Results are merged together, followed by batching single tuples, allowing the aggregation step to process 100 tuples at once. Intra-Operator Parallelism (see Section 2.1) on data is ensured by three instances of the selection and projection operator, splitting up tuples according to their first attribute by partitioning operator. Inter-Operator Parallelism is achieved through multiple threads doing work in parallel with respect to data dependencies. With batching, the aggregation step can operate with SIMD support, aggregating on multiple tuples simultaneously.

As a source, the data stream can be constructed not only through a tuple generator or the query itself, but from file, through REST HTTP or via network ports. It is possible to vary the tuple arrival rates to simulate peaks, like transmission delays or different data distributions, e.g. data dependence from daytime in a social network. In addition to this, tuples can become outdated after a while, simulating a possibly endless stream of data.

2.4 Software Optimizations

When transferred to manycore architecture, some important observations can be made, resulting in challenges to solve and opportunities to use. These points are discussed below.

2.4.1 Parallelization Overhead

The lower clock speed of cores in a manycore architecture (caused by reducing waste heat problems) leads to clearly higher run times initially than on multicore when running singlethreaded. Even when using a handful of threads the performance is notably lower compared to the parallelism of hundreds of possible threads on a manycore processor. To overcome this problem, it is necessary to use as many cores as possible simultaneous to exploit this advantage of manycore parallelism. When scaling out accordingly, typical problems like synchronization and load balance occur, addressed afterwards in this section.

2.4.2 Operator's Computational Cost

It is possible to run many queries at once, even with several data stream sources. But to gain speedup on a single query, the contained operators have to run in parallel through partitioning. For simple queries, e.g. by processing sensor data, filtering and aggregating residual tuples, it is difficult to gain any speedup. In operators with higher computational costs or longer sequences of operators, however, doing work in parallel finishes the query faster (obviously). Therefore it is necessary to take the minimal amount of work done by queries into account, to determine the ideal degree of parallelization.

2.4.3 Partitioning Degree and Aggregation Costs

The next point to regard is the number of threads needed for a given query to minimize runtime, compared to increased overhead gained through splitting data and merging results from different threads. The usage of statistics is a common solution for this problem, but it is already shown that the results can be far away from optimal when running on a dynamic workload as shown by Gedik et al. [1]. Regarding the complexity of a query is a first step, but for a dynamical adjustment more observations on the influence of runtime have to be made.

2.4.4 Thread Pool

The need of many threads when partitioning can become a major slowdown. By creating threads when needed, the overhead of creation can kill any speedup gains easily. The widely known solution for this is the usage of a thread pool. On initialization of the pool threads are created in the amount of support from underlying hardware (e.g. on a 60 core processor, supporting up to four threads per core, 240 threads). The query fetches its necessary amount of threads from the pool, possibly being able to add or remove threads from current state when having peaks in the data arrival rate. When efficiently scheduled, the delay and overhead for creating threads is reduced significantly, but it is no trivial task.

2.4.5 Thread Synchronization

Another bottleneck in parallelization of query operations lies in thread contention. When tuples are forwarded by publish-subscribe between threads, the exchange structure (like a queue) needs to guarantee that threads do not get in each other's way, e.g. when the publisher writes his result and the subscriber reads it. By scaling up to hundred and more threads on a manycore processor, the task of ensuring concurrency in an efficient way is not trivial anymore as shown by Yu et al. [2]. Until a new approach closes this gap, already existing patterns need to be optimized.

2.4.6 Core Scheduling

In addition to this, the scheduling from threads to cores is another optimization problem. Leis et al. [3] addressed this scheduling along with load balancing (see next Section 2.4.7) When threads are switched between cores, caching effects can be lost, resulting in higher runtime. An adverse scheduling for threads, e.g. when they have to communicate with each other, can become a bottleneck too, especially when the cores they run on have a long distance between them inside of the processor. The OpenMP Interface allows three possible assignment strategies for threads to cores, but it is also possible to implement own strategies, e.g. with the POSIX Thread Interface called *PThreads*.

2.4.7 Load Balancing

Next, a good load balancing of work to threads has to be achieved, shown by Fang et al. [4]. It is possible through bad partitioning of data (e.g. a suboptimal assignment strategy) that one out of many partitions gets most of input data. This results in many partitions idling and bad speedup. But not only the skew of data can be regarded here, the grade of partitioning is another factor. When a query does only use one or two cores instead of most of them (when not under competitive conditions) and the query is complex enough for parallelizing gains, a loss of speedup is the result. Especially when using a manycore architecture instead of a multicore, because of slower clock speed.

2.4.8 Latency Reduction

Finally, when tuples arrive on data stream, it is possible to use microbatching for reduced overhead between operators, as shown by Pinnecke et al. [5] on GPU's. Tuples are put into a batch and when the batch is full, it is forwarded to the next operator. This allows a fine-grained tuning of partitioning, when every operator in parallel works on a part of the batch, as well as using vectorization through processing tuples in a batch in parallel instead of every tuple on its own.

2.5 Summary

Summarizing this up, even when given some opportunities by hardware (discussed in the next section), there are many points that have to be considered on software. They influence performance on multicore too, but even more on a manycore architecture, regarding the advantages and disadvantages of many cores in a single processor, e.g. reduced clock speed and more threads supporting.

3. MANYCORE PROCESSORS

As already shortly mentioned in Section 1, a manycore processor has some very important differences to a multicore chip. On manycore, the processing units (cores) are packed tightly together, to enable short communication paths between them and keeping the processor small. The cost, however, lies in lower clock speed because of cooling conditions. When clocking is high, a lot of heat loss is generated, which is simply impossible to get rid of when having 60+ high-end cores on a single chip. Another result out of this point is the switch from complex cores to simple cores (e.g. in terms of scheduling instructions), reducing energy consumption and possibly performance.

To gain a speedup on a manycore processor, it is both necessary to know the software and hardware possibilities.

The following recommendations are applicable to other use cases different from stream processing, too. The software regardments are already appointed in the previous Section 2.4, the following points deal with the Intel Xeon Phi manycore processors and opportunities given by them.

3.1 Xeon Phi

The Xeon Phi series from Intel consists of processors using manycore architecture. The first processors are just used for research purposes and as niche products, but with Knights Corner (KNC, released 2012) Intel's first commercial manycore processor was obtainable [6]. The next (and latest) one called Knights Landing was released end of 2016.

Since KNC was only available as coprocessor using a host system connected with a PCI bus, the major optimization focus was on efficient data transfer between them (called offload), in addition to manycore improvements. It is possible to transfer parts of a program to the manycore coprocessor through compiler pragmas (when using simple datatypes like integer or arrays) or using the Cilk programming model (when transferring complex objects). Running the full program on the coprocessor (called native mode) is limited by the small memory on chip, ranging around 6 to 16 GB only. Main bottleneck for database application usage of KNC is the PCI bus connection between host and coprocessor [7], allowing a data transfer rate (bandwidth) around 15GB/s. The KNL chip addresses this problem, being available as full-fledged processor, using memory directly without need of a host. In addition to this, it provides some important improvements compared to its predecessor KNC, like high bandwidth on-package memory or extended SIMD support through AVX-512 instructions. This leads to new opportunities that have to be regarded when optimizing applications to KNL. The following sections show the new possibilities given by KNL hardware.

3.1.1 MCDRAM

The KNL CPU brings along its own high-bandwidth memory, called Multi-Channel Dynamic Random Access Memory (MCDRAM). The size of this memory amounts to between 8 and 16 GB, allowing a bandwidth from around 320 GB per second (around four times higher than acessing DDR4-RAM). This memory can be used as a low-level cache (cached mode), as addressable memory for applications (flat mode) or as a combination of both (hybrid mode). When used in cache mode, the program can use it without any adaptation efforts, speeding up its run time. In flat mode, however, the memory is not accessed on its own. Therefore the numactl or memkind library can be used efficiently. It is advised to use the numactl library when all memory needs from the running application fits into MCDRAM, memkind otherwise [8]. In flat mode, the possible speedup is higher than in cached mode, but code has to be optimized for this carefully by the designer.

On data stream purposes in flat mode, the MCDRAM can be used for e.g. storing hash tables when joining tuples, utilizing the high bandwidth for speeding up accesses.

3.1.2 AVX-512 extensions

Another new feature supported by KNL are AVX-512 vector instructions, an addition to the AVX2 instruction set, used by intel processors in general. The register width is doubled compared to the 256-bit instruction set, allowing a better exploitation of SIMD operations (that means, execute an instruction in parallel on multiple data). Besides wider registers, exponential and reciprocal instructions, conflict detection and prefetch instructions are added, leading to new and more vectorization opportunities. The only thing needed is a compiler supporting these extensions, like the Intel C++ compiler or a newer version of the GNU compiler [9].

To exploit these SIMD effects, it is necessary to not process one tuple at a time. When each tuple is exchanged between operators, the processing cannot fully utilize the SIMD characteristics. A solution for this problem is batching. Tuples are gathered until batch size is reached and then, they are forwarded to the next operator at once, reducing communication efforts and allowing operators to use them simultaneously.

3.1.3 Clustering Modes

The next point to focus on are the new clustering modes. For efficient use of a manycore processor, the parallel part of a program has to spread out to as many cores as possible, to gain maximum speedup. However, memory accesses become more and more problematic this way. Every core has its own cache and synchronizing updates, reads and writes across 60 and more cores can eat up speedup gains easily. Latency delays caused by this are one of the bigger problems on the predecessor KNC. As solution, next to improved locality of cores, caches and connections on KNL, different clustering modes are integrated. Possible modes are all-to-all, quadrant/hemisphere and SNC-2/SNC-4 (Sub-NUMA-Cluster) [10]. The all-to-all mode uses uniformly distributed memory addresses, causing bad worst cases with long request times, therefore should not be used if possible. In quadrant mode the cores are divided into four regions (two regions on hemisphere mode), reducing communication between cores to their corresponding region, excluding long range exchanges. SNC-2/SNC-4 mode extends the hemisphere/quadrant mode through NUMA effects, regarding every region as one NUMA node. This mode delivers best result through short paths and should therefore be default.

3.1.4 Memory Page Size

Processors use the Translation Lookaside Buffer (TLB) to speedup the conversion from virtual memory addresses to physical memory addresses when accessing memory. It is possible to set different page sizes. A small page size wastes less memory compared to a huge page size, but increases the number of TLB entries, leading to longer time needed to find a specific entry. To speed up applications, the huge page setting is preferred, because TLB misses and TLB lookup time can easily become a bottleneck of processing on a manycore architecture (any core has to pass through the TLB when converting a virtual to physical memory address, more than on a multicore architecture).

4. RELATED WORK

There is much work done around using manycore processors, even for Xeon Phi KNL. They consider its hardware properties, various workloads, optimization of code and much more, mostly in context of high performance computing (HPC). For this paper, the focus is about parallel data stream processing. There are some publications about typical database operations on manycore architecture, like joins

or the MapReduce model, but not that much on data stream processing (especially on KNL) to the best of our knowledge. This section briefly summarizes some work done along with met problems.

The first paper from Cheng et al. [11] presents PhiDB, a query processor for OLAP with simultaneous multithreading capabilities. PhiDB is heavily optimized for Xeon Phi architecture, along with recommendations in terms of implementation details. The results are tested on a KNC coprocessor, showing the importance of careful adaptation of code and applications when porting from multicore to manycore architecture.

Another approach from Jha et al. [12] focuses on main memory hash joins on manycore architectures. They implemented state of the art hash join algorithms (separated into hardware-conscious and hardware-oblivious ones), running on a Xeon Phi coprocessor and on a modern CPU in comparison. Results show again the caution needed when optimizing for manycore processors, along with better efficiency for hardware-oblivious joins.

Lu et al. [13] added the aforementioned MapReduce model on a Xeon Phi coprocessor, named MRPhi. This framework is optimized for vectorization and SIMD instructions using the wider registers for speedup, in addition to pipelining the map and reduce phases.

The next attempt from Polychroniou et al. [14] looks onto vectorization of database operators in general, achieving speedups through SIMD possibilities. They tested implementations of Gather and Scatter, hash tables, selection scans, sorting, partitioning, joins and a bloom filtering. Results are tested through Xeon Phi coprocessor as well as with a modern CPU, showing the importance of SIMD usage when running on modern processors.

All these sources show the necessity of optimizing for possible settings on manycore processors as well as adjusting code to run on them. If optimized carefully, speedups are possible in different fields of processing data. The question still exists, if manycore processors can be used to speed up data stream processing in general.

5. PRELIMINARY EXPERIMENTS

Before porting our SPE to KNL, we run a bandwidth and latency analysis on both multicore and manycore CPU, to measure differences and derive further decisions. For this task, we use the Intel Memory Latency Checker (MLC) tool, which supports KNL since last update. Results are presented in next subsection.

After that, in addition to bringing PipeFabric on KNL, we apply first parallelization optimizations into our framework on a data generator for producing a data stream. Goal of these preliminary experiments on our SPE using KNL is to verify the general utilization of a manycore CPU for data streaming purposes, resulting ideally in a much better performance compared to a multicore processor by exploiting parallelism opportunities.

For this paper, the results are tested on an Intel Xeon Phi KNL 7210 with 64 cores and 1.30GHz clock speed per core. MCDRAM is used in cache mode, clustering mode is SNC-4 and it uses huge page size for memory, as discribed earlier in Section 3.1. For comparison between manycore and multicore, an Intel Core i7-2600 is used, with 4 cores and 3.40GHz clock speed.

Processor	Memory type	Prefetching	No Prefetch
i7-2600	L1	1.1ns	1.1ns
	L2	3.4ns	3.4ns
	L3	3.9ns	14.9ns
	Memory	5.7ns	65.7ns
KNL 7210	L1	3.1ns	3.1ns
	L2	7.1ns	13.2ns
	MCDRAM	15.1ns	172.7ns
	Memory	14.2ns	146.3ns

Table 1: Idle latency for memory accesses

5.1 Bandwidth and Latency

The first point to address is general latency and bandwidth to caches and memory. The main requirement on real-time data stream processing is low latency, realised on software-side by our engine PipeFabric. But on hardware, the latency needs to be measured with respect to bandwidth. As a note, when bandwidth boundary is reached (e.g. 320GB/s for MCDRAM or 70GB/s for DDR4), latency on a core will increase due to longer time waiting for data.

An important fact to consider for measurements is the hardware prefetching mechanism. To speed up memory accesses, the hardware prefetcher predicts based on already done instructions the next needed elements in memory and loads them into cache. This leads to fast accesses when the prediction was right. When measuring latency, this can result in much lower values than in real use cases, where memory accesses can become more random and unpredictable. Table 1 shows idle latencies for both processors, that means the time needed for memory accesses without competition. Numbers are gained through MLC[1] tool from Intel, allowing not only to disable hardware prefetching, but also fine-grained control on test cases.

It can be seen that the latencies for KNL are worse than on i7 processor. This is affected by slower clock speed and internal manycore structure. Another observation is that MCDRAM latency is worse than regular DDR memory access. At first appearance this can be surprising, but that is because of measuring idling latency. When demand on memory increases, e.g. more cores try to access memory at the same time, MCDRAM latency will stay lower for more simultaneous accesses as DDR memory would do. This allows more cores to fetch data from MCDRAM on measured latency than on DDR, where the bandwidth limit is reached very quick.

As summary, the generally higher latencies for Xeon Phi should be kept in mind. The greater possible parallelization degree has to catch up this disadvantage for gaining speedup against multicore CPUs.

5.2 Insights for Stream Processing

Initially tests done on KNL as well as on i7 CPU show worse performance when using a manycore processor, as expected caused by slower clock speed and singlethreaded execution. This changes fast when increased parallelism opportunities of KNL come into play. To point an example, we picked our tuple generator from PipeFabric (see Section

[1]https://software.intel.com/en-us/articles/intelr-memory-latency-checker [Online; accessed 05-May-2017]

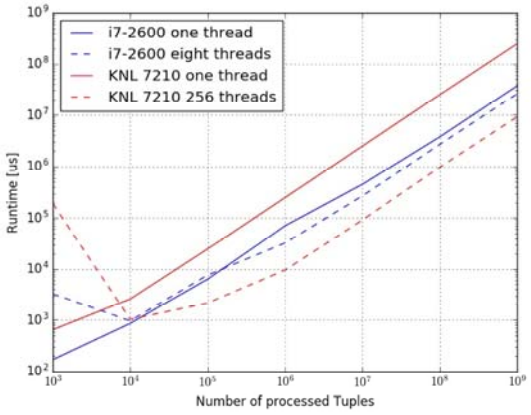

Figure 3: Speedup for generating tuples

2.3) as source for the query, analyzing the speed of generating tuples for upcoming operators.

We measure singlethreaded tuple generation first on i7 processor and on KNL manycore. As it can be seen in Figure 3 (please note the logarithmic scale of runtime at y-axis), the slow clock speed results in longer time needed to produce an equal amount of tuples. But with spreading out tuple generation to as many threads as available (through OpenMP) a notable speedup can be achieved.

With only little to produce (around a thousand tuples), the parallelism through more threads generating tuples cannot catch up the delay of the overhead, leading to major slow-downs on i7 as well as on KNL. This changes very quickly, when more tuples are processed. On Xeon Phi, the speedup gain is much higher than on i7 caused by more threads generating tuples, exploiting the parallelization possibilities of a manycore architecture. Very soon the Xeon Phi catches up to i7 and finishes continuously faster than the multicore processor.

6. CONCLUSION AND FUTURE WORK

In this paper, we focus on improving data stream processing with modern hardware. Requirements of low latency and high throughput raise the question if a manycore processor can contribute to better performance through its intense parallelization degree. Low clock speed and at most average cache and memory latency of KNL compared to a multicore CPU (as shown in Table 1) needs to be compensated through extended parallelism.

Observations on our SPE PipeFabric lead to different optimization criteria needed when adapting to manycore processor architecture. The increased number of threads for exploiting parallelism efficiently intensifies delays through synchronization on data transfer, therefore a better mechanism than locks should result in much better performance. A thread pool is another important optimization, reducing overhead of thread creation significantly. Careful fine tuning is necessary when additional hardware opportunities come into play, like scheduling threads to certain cores or using high bandwidth memory. Finally, when utilizing all cores on KNL maximizing available parallelism, it can surpass performance of a multicore CPU easily even with its higher latencies to memory and cache.

Our future work will focus on more optimization criteria for Xeon Phi, exploring the influence on runtime through different approaches like using a threadpool or tuple exchange mechanisms of operators. When a manycore processor can contribute to better performance on stream processing, the next question raises, how much better performance can become when fully adapting on underlying hardware. Especially when compared to todays main solution for throughput - distributed computing on clusters of multicore processors.

7. REFERENCES

[1] B. Gedik, S. Schneider, M. Hirzel, and K.-L. Wu, "Elastic Scaling for Data Stream Processing," *IEEE TPDS*, pp. 1447–1463, 2014.

[2] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores," *VLDB*, pp. 209–220, 2014.

[3] V. Leis, P. Boncz, A. Kemper, and T. Neumann, "Morsel-driven Parallelism: A NUMA-aware Query Evaluation Framework for the Many-core Age," SIGMOD, pp. 743–754, 2014.

[4] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, and J. Zhu, "Parallel Stream Processing Against Workload Skewness and Variance," *CoRR*, 2016.

[5] M. Pinnecke, D. Broneske, and G. Saake, "Toward GPU Accelerated Data Stream Processing," *GVDB*, pp. 78–83, 2015.

[6] T. Halfhill, "Intel Shows MIC Progress." http://www.linleygroup.com/newsletters/newsletter_detail.php?num=4729, 2011. [Online; accessed 05-May-2017].

[7] C. Gregg and K. Hazelwood, "Where is the data? Why you cannot debate CPU vs. GPU performance without the answer," in *IEEE ISPASS*, pp. 134–144, 2011.

[8] Colfax, "MCDRAM as High-Bandwidth Memory (HBM) in Knights Landing Processors: Developer's Guide." https://colfaxresearch.com/knl-mcdram/, 2016. [Online; accessed 05-May-2017].

[9] Colfax, "Guide to Automatic Vectorization with Intel AVX-512 Instructions in Knights Landing Processors." https://colfaxresearch.com/knl-avx512/, 2016. [Online; accessed 05-May-2017].

[10] Colfax, "Clustering Modes in Knights Landing Processors." https://colfaxresearch.com/knl-numa/, 2016. [Online; accessed 05-May-2017].

[11] X. Cheng, B. He, M. Lu, C. T. Lau, H. P. Huynh, and R. S. M. Goh, "Efficient Query Processing on Many-core Architectures: A Case Study with Intel Xeon Phi Processor," SIGMOD, pp. 2081–2084, 2016.

[12] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh, "Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach," *VLDB*, pp. 642–653, 2015.

[13] M. Lu, L. Zhang, H. P. Huynh, Z. Ong, Y. Liang, B. He, and R. S. M. Goh, "Optimizing the MapReduce Framework on Intel Xeon Phi Coprocessor," in *IEEE Big Data*, pp. 125–130, 2013.

[14] O. Polychroniou, A. Raghavan, and K. A. Ross, "Rethinking SIMD Vectorization for In-Memory Databases," SIGMOD, pp. 1493–1508, 2015.