Stream Join Processing on Heterogeneous Processors

Tomas Karnagel, Benjamin Schlegel, Dirk Habich, and Wolfgang Lehner

University of Technology Dresden
Department of Computer Science
Database Technology Group
01062 Dresden
{tomas.karnagel; benjamin.schlegel; dirk.habich; wolfgang.lehner}@tu-dresden.de

Abstract:
The window-based stream join is an important operator in all data streaming systems. It has often high resource requirements so that many efficient sequential as well as parallel versions of it were proposed in the literature. The parallel stream join operators recently gain increasing interest because hardware is getting more and more parallel. Most of these operators, however, are only optimized for processors with homogeneous execution units (e.g., multi-core processors). Newly available processors with heterogeneous execution units cannot be exploited whereas such processors provide typically a very high peak performance. In this paper, we propose an initial variant of a window-based stream join operator that is optimized for processors with heterogeneous execution units. We provide an efficient load balancing approach to utilize all available execution units of a processor and further provide highly-optimized kernels that run on them. On our test machine with a 4-core CPU and an integrated graphics processor, our operator achieves a speedup of 69.2x compared to our single-threaded implementation.

1 Introduction

The window-based stream join is a fundamental operator in all data streaming systems. It joins tuples that fulfill certain predicates from two or more windows, which move continuously over input data streams. Depending on the size of the windows, the operator has high performance requirements so that many optimized versions have been proposed for it. This includes sequential stream join operators [VNB03, GO03], which exploit various index data structures (e.g., hash tables) for a faster processing as well as parallel stream join operators [GYB07, TM11], taking advantage of different types of parallelism. The parallel operators are gaining increasing interest because hardware nowadays is getting more and more parallel.

Besides increasing the number of parallel execution units (e.g., cores, vector instruction width) in modern multi-core processors, one interesting trend is the development of heterogeneous processors where circuits for different purposes are placed on one chip. On the one hand, each new processor generation provides new instruction sets like Streaming SIMD Extensions (SSE) or Advanced Vector Extensions (AVX) that provide specialized

instructions for video encoding, string processing, or data encryption. On the other hand, also different types of execution units are placed on a single chip. Examples for such processors with heterogeneous execution units are the IBM Cell processor, AMD processors of the Fusion brand, and Intel's Sandy Bridge or Ivy Bridge processors. The AMD and Intel processors, for example, host CPU and GPU cores on a single chip. Although heterogeneous processors typically provide a better peak performance than traditional multi-core processors, it is more demanding to develop algorithms that exploit this performance.

In this paper, we propose a window-based stream join operator that is tailor-made for processors with heterogeneous execution units. Our stream join processes tuples from windows of two input streams using a band predicate [DNS91]; the operator creates join tasks (i.e., batches of tuples from both streams) of which each is processed by one of the available execution units. A dynamic load balancer is used to evenly distribute the load. We implement the operator using OpenCL and provide three optimized OpenCL-kernels. On our test machine, which comprises CPU and GPU cores, we achieve high speedups of up to 70x compared to a single-threaded implementation and 20x compared to a multi-threaded implementation that both do not exploit the heterogeneous cores of the system.

2 Preliminaries

Our aim is to speed up window-based stream joins by partitioning the work and running the partitions on multiple devices within a heterogeneous environment. In the following, we present the window-based stream join and a way to run the join in a parallel fashion. We also give an introduction to OpenCL as a framework to program device code (called kernel) that is able to run on different hardware devices without further modifications.

2.1 Stream Joins with Band Conditions

The stream join is a important operation to concatenate and filter data streams. Stream joins are performed on two or more streams in which tuples are continuously arriving. Instead of holding all the data, the continuous stream is partitioned in stream windows, for which the join is executed. There exists many variants and optimizations for this operator. Aggarwal [Agg06] provides an excellent overview about these variants and respective sequential optimizations.

Recently, parallel stream join operators gained much attention. This includes stream join implementations for FPGAs and NUMA systems [TM11] as well as for the cell processor [GYB07]. In the latter, chunks of tuples are transferred to the 8 co-processors of the cell processor and these cores perform a nested-loop like operation. The authors chose the nested-loop join because it has no intermediate state that needs to be updated with every new tuple. It is also the best choice for implementing band conditions. A band join between relation R and S is a join, where the values of the join attribute on R need to fall in a predefined band of the join attribute values in S to be joined together. An example could

be found in position tracking. Two position sensors stream data and the join on a time window is done to find cases, where both sensors are close to each other. The definition of *close* is done by the band conditions, e.g., a radius. There are approaches to do a efficient sort-merge join [LT95] or hash join [Sol93] with band conditions but the nested-loop join is a straightforward approach, which is also easy to parallelize.

In this paper, we focus on a stream join with band condition as an important derivation of the stream join. Based on the stream join running on the cell processor, we port the algorithm to work efficiently on the GPU (Graphics Processing Unit) as well as on the CPU. Furthermore, we exploit both computing devices for parallel execution using OpenCL. Previous work has been done for database join algorithms on multi-core CPUs [SD89] as well as join algorithms on the GPUs [HYF+08, KLMV12]. In the latter, the join data is copied to and from the GPU device through the PCI express interface and the join execution was done completely on the GPU.

2.2 Open Computing Language (OpenCL)

We chose the *Open Computing Language* (OpenCL)[Gro] to implement our join for heterogeneous hardware environments. OpenCL is standardized by the Khronos Group and implemented by hardware vendors for their specific hardware. It allows to run the same code on different hardware platforms including x86-CPUs and most of the modern GPUs. With a pre-installed driver, OpenCL programs can use all supported computing devices available in a system. These could be multiple CPUs, GPUs, FPGAs, or different kinds of accelerators. To use the OpenCL framework, a programmer has to write host code and device code. The host code is written in C or C++ (or other languages using wrappers). It is used to initialize the different hardware devices, to manage data transfers to and from the devices, and to enqueue device code (kernel) executions. For the execution of a kernel, the host has to specify the arguments for the kernel and the dimensions the kernel is executed in. The arguments could be single values or pointer to arrays of data, which need to be transfered before kernel execution. The dimensions define the logical positioning of work-items, e.g., in a two dimensional array. Work-items can query their position (index in each dimension) during execution. It is possible to group work-items into work-groups. Synchronization between work-items is only possible within work-groups.

OpenCL devices are physically partitioned in compute units, which again are partitioned in processing elements. Each processing element can execute one or more work-items. Kernels are written in a subset of C, which is extended by mathematic, relational, and vector functions. All started work-items execute the same kernel. Such kernels read data, do computation, and store a result. While all work-items execute the same kernel, the difference between them is that they load data and put the results to different positions in memory. If used on a GPU, the kernels should be highly parallel.

In OpenCL, the main memory of the device is called *global memory*. For most compute devices, global memory is the largest memory but also the memory, which takes the most time to access. Some devices also have *local memory* (shared memory), which is faster

but smaller. Local memory can be shared between work-items in one work-group. When sharing memory, the work-group synchronization features are needed to avoid inconsistent data. There is also *private memory*, which can only be accessed by a corresponding work-item. When supported in hardware, private memory is implemented with fast registers.

3 Parallel Stream Join Execution

For a fast stream join implementation on multiple devices, efficient load balancing is needed as well as optimized device friendly code. In the following, we present our approach to achieve dynamical load balancing and we further discuss the implementation details of our OpenCL kernels as well as two CPU variants as our basis of comparison.

3.1 Dynamic Load balancing

In OpenCL it is possible to run kernels on specific devices. If the system has multiple devices of the same or a different kind, then kernel and load distribution is needed. One way would be to distribute individual kernels, which are adapted to the device properties together with dedicated data chunks for this kernel. Another way is a unified kernel that is deployed to all devices and load balancing is done through partitioning the data.

In this paper, we build an unified kernel that works with partitioned data. Such a kernel does not need information about the devices at compile time and all OpenCL devices are supported. Hence, our focus is on a good data partitioning and load balancing. For efficient load balancing, we apply a job queue approach. We implemented a task queue where each task holds the attributes of a number of join tuples from two stream windows. There is also a placeholder for the results of the join. The data for filling the tasks could come from hypothetical endless streams, which keep filling the queue with tasks. Join windows are taken from the streams to create a task for the queue. The windows of one stream could be overlapping or distinct. We choose the distinct variant for this work. This is equal to a batch-wise stream join operation. A batch of new values for one stream is joined with a join window of another stream and vice versa. Only the data stored in a task is joined together. The OpenCL devices pick the tasks from the front of the queue and mark it as 'in-work'. Only tasks not 'in-work' and not 'done' are picked. A task is executed on a device and the result is written back to the result placeholder. The task is then marked as 'done' and can be further processed (e.g., streamed out to the next node). The device then selects the next free task to execute.

The queuing setup is illustrated in Figure 1. With the job queue, it is possible to have a dynamic load balancing because every device executes the tasks as fast as possible and without being idle, selects a new task from the queue. This way it would be easy to add more devices to the environment even if they have different processing capabilities.

One single task execution involves copying the needed data to the device, execution of the kernel with the given data, and copying the data from the device back to the result

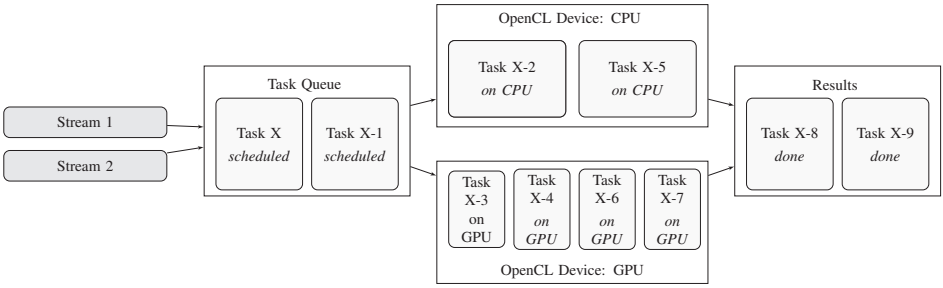

Figure 1: Queuing approach for a heterogeneous environment with a CPU and a GPU, both executing tasks from the same queue. The width of the task boxes within the devices symbolize their runtime.

placeholder. Taken a CPU with integrated Graphic Processor, for example, there are two devices, the CPU and GPU both on one processor chip. In OpenCL, copying the data to the CPU is nearly free because no data is copied physically. The data for the CPU is only referenced and the kernel is therefore working on the original data. Since the original data is not modified, there is no conflict with further processing steps. For the GPU, however, the data has to be physically copied. For the processors we aim for, the GPU shares the main memory with the CPU and thus the data copying takes a similar time to the memcpy instruction. If the GPU is connected through an external PCIe interface, then data copying is likely to be more time consuming.

3.2 Kernel Implementations

An OpenCL kernel is a code fragment, which can be run on all devices supporting OpenCL. To evaluate the performance of our stream join, we implemented 3 different kernels in OpenCL and 2 implementations not using OpenCL.

The first of our two non-OpenCL implementations is a **single-threaded** implementation, which processes the tasks sequentially and also performs the nested-loop comparisons within a task in sequential order. The single-threaded version runs only on the CPU. The second non-OpenCL implementation is a multi-threaded version for the CPU, implemented with **OpenMP**. This version also processes the tasks sequentially but parallelizes the nested-loop implementation within a task. We also tested running the tasks in parallel with OpenMP but we did not see a performance benefit.

The three OpenCL kernel work all in a similar way: each kernel is executed N times, where N is the number of rows in a join column. Each instance of the kernel has an unique ID. The ID is used by the kernel instance to determine the row in the first column it has to access. This row will then be compared to all entries in the second column. In this stage of our work, the kernel only counts the join partners found and does not return the join tuple. This way we know that the result is only one number per kernel instance and space for the result can be pre-allocated.

OpenCL device	AMD A10-5800K APU with Radeon HD Graphics CPU AMD A10-5800K	 GPU AMD Radeon HD 7660D
OpenCL compute units	4	6
Max parallel work-items	4 (cores)	384 (Shader)[1]
Global mem size	30,108 MB	1024 MB
Frequency	3.8 GHz	0.8 GHz

Table 1: Test platform with two compute devices and their properties.

The kernels differ in the way how they evaluate the join predicates and and increment the number of result tuples. **BranchedCL** works like explained above. During the kernel execution, the sum of join partners is incremented using a branch that evaluates whether the join predicate is true for two tuples.

The second kernel version is called **NoBranchCL**. In this kernel, we remove the branch using predication. The comparison result, which is either 0 or 1, is directly added to the sum with every comparison. This results in much more write operations but branching is avoided, which may beneficial for the CPU and GPU. The CPU could benefit from a better usage of the instruction pipeline. With branching the CPU has to guess which branch will be used next and wrong predictions could lead to performance loss. With a branch free algorithm this performance loss is avoided. The GPU can only execute in SIMD (Single Instruction Multiple Data) fashion. This means one instruction is run on all the data and then the next instruction is processed. When the execution is branched, the kernel instances not entering the branch are waiting idle for the kernel instances entering the branch. This idle time could be avoided with branch free algorithms.

Our final kernel is called **VectorizedCL**. Here, we use the branch free algorithm but instead of doing one compare operation at a time (in a kernel instance), we use OpenCL build-in SIMD instructions to compare 4 values at one time. This is done with vectors holding 4 values. Our first vector holds the search key on all positions. This vector is compared periodically with a second vector, which holds data from the second column. If the hardware supports SIMD instructions, the vector comparison can be done in the same amount of clock cycles as a normal comparison, resulting in an optimal speedup of 4x. Our test platform presented in Table 1 supports SIMD instructions on both devices.

4 Evaluation

We implemented the stream join algorithm in OpenCL with our dynamic load balancing approach. For a detailed evaluation, we use the 5 kernels as stated before. We are aware that in most scenarios with GPUs, the memory bandwidth of the PCIe bus is a limiting factor. But new hardware trends tend to include a graphic processor in the main processor. This could bring speedups in the data transfer rates because the GPU is accessing portions

[1]There are 6 SIMD units with each 16 thread processors with each 4 ALUs (Shader).

of the system main memory. Two current CPUs with integrated graphics are the Intel Ivy Bridge Processor and the AMD Trinity APU (Accelerated Processing Unit). We choose the AMD Trinity APU for our first tests, because the architecture seems to be more optimized for heterogeneous algorithms. The specifications of our test system are shown in Table 1. All tests presented were done on this system. We also tested our implementation on other systems with similar results.

In our implementation, we are able to select specific devices that access the job queue. We therefore tested our kernels on the CPU only, on the GPU only and both together. Having the tests limited to one specific device, it is possible to observe the performance differences of the kernels on these devices. For all our tests we used the same test parameters, which are shown in Table 2.

Property	Value
Initial No. of tasks	500
Task consists of	column1, column2, Spaceholder for result
Task size	300 KB (100 KB each)
No. of values per column	25,600
Comparisons per task	655.36 million
Started work-items in OpenCL	25,600 per task

Table 2: Overview on test parameters.

Our first test was done on the CPU and the results are illustrated in Figure 2. We have the single-threaded kernel (1 thread) and the OpenMP kernel on the left side of the figure. The OpenMP kernel achieves a speedup of 3.5x compared to the single-threaded approach, which is as expected for a 4-core CPU system. The right side of Figure 2 shows the OpenCL kernels run on the x86 CPU. The BranchedCL kernel runs similar to the OpenMP kernel. The overhead of OpenCL is responsible for the slightly worse performance. We can see, that the CPU benefits from avoiding branching and has a speedup of 1.9x compared to the kernel using branching. This is caused by a better instruction pipelining through avoided branch misses. Using SIMD with vectorized comparisons gives a further speedup of 3.6x, which is close to the expected optimal speedup for hardware supported SIMD instructions. The speedup is not 4x because of setup costs for the vectors. The vectorized OpenCL kernel on the CPU has a speedup of 21.3x compared to the single-threaded version.

Figure 3 shows the same test as before but here the OpenCL kernels were executed on the GPU. The time measurement for the GPU kernel includes all data transfers needed. We see a much higher speedup for the initial BranchedCL kernel. This is caused by the higher core count of the GPU. More surprising is the worse performance of the NoBranchCL kernel. We suppose that the additional write costs have a bigger influence on performance than branching. Having no branching, every comparison results in a write operation to the sum variable (increasing the value by 0 or 1). Branching on the GPU results in some parts of the work-item waiting for the work-items that are branching and staying idle. We suspect that incrementing the sum variable is a relatively fast operation, so that the idle time for the not branching work-items is small. Also without branching, all work-items have to write

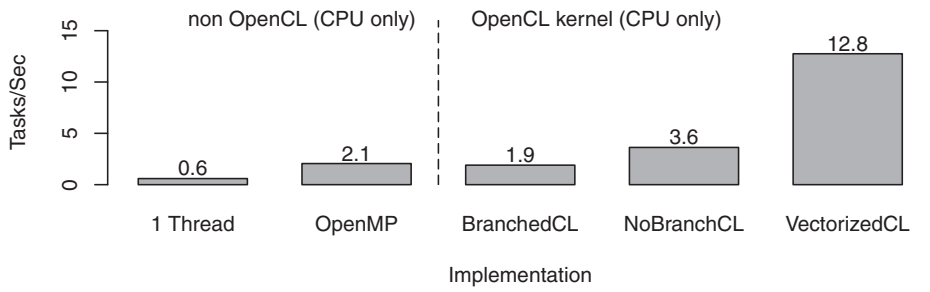

Figure 2: All implementations run on the CPU. The best performing OpenCL kernel has a speedup of 21.3x compared to single-threaded performance.

their sum variable while with branching some work-items would simply wait. Anyway the branch free algorithm is needed for the vectorization, which then brings a speedup of 2.4x compared to the not vectorized branch free algorithm and a final speedup of 47.5x compared to the single-threaded kernel.

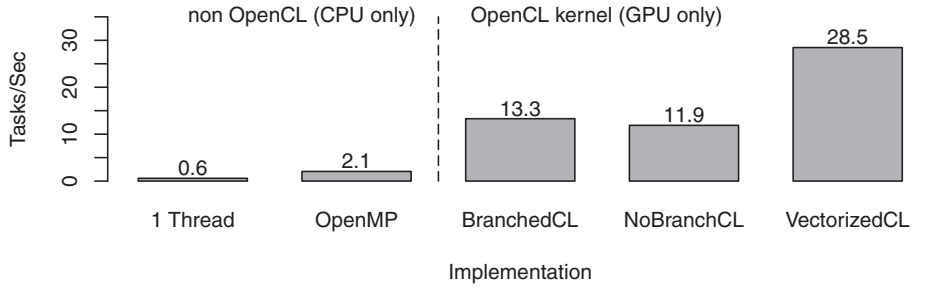

Figure 3: The first two implementations run on the CPU, the other three on the integrated GPU. The best performing OpenCL kernel has a speedup of 47.5x compared to single-threaded performance.

The final test was running our implementation using CPU and GPU for the OpenCL kernel. Here we only tested the vectorized version as the version with the most performance on both devices. Figure 4 illustrates the results. The first 4 bars are taken from previous tests and the last bar shows the performance of the VectorizedCL kernel running on both devices simultaneously. We see that the performance of both devices is added to a final processing throughput of 69.2 tasks per second (meaning 27 billion nested-loop comparisons per second, including transfer times). The final speedup to the single-threaded version is 69.2x and compared to the OpenMP variant the speedup is 19.8x.

5 Open Issues for Future Work

Having good performance results, still we left some issues open for discussion. We did not implement a join that returns result tuples but a join that returns the number of result tuples

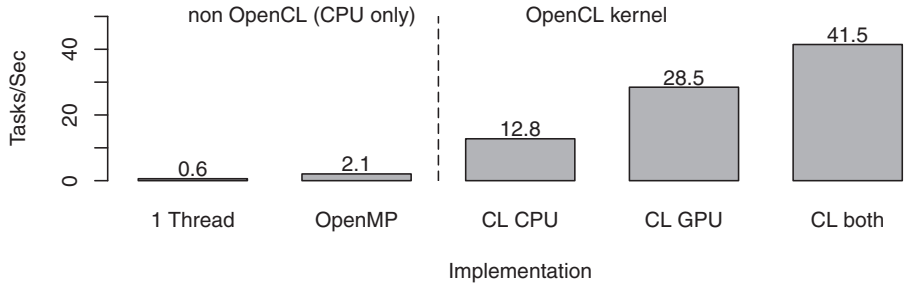

Figure 4: The first two implementations run on the CPU, while the OpenCL vectorized kernel runs either on the CPU, the GPU or on both devices. Running on both devices shows a speedup of 69.1x compared to single-threaded performance.

found. Returning a set of result tuples without prior knowledge of the result size, is not trivial with OpenCL. It is not possible to allocate memory dynamically within an OpenCL kernel. For our current implementation, the host code only allocates 4 bytes per started work-item to store the individual results. Besides the stream join, it would be possible to port other stream algorithms to our dynamic load balancing approach. We hope that our approach enables various algorithms to efficiently use heterogeneous hardware but we did not test that yet. Also it is worth thinking about dedicating specific tasks to the specific devices. While loosing the flexibility in load balancing, the performance could be better through more hardware specific optimizations. We also leave further testing for future work. We tested our approach on different AMD processors with integrated GPUs. We did not test Intel Core third generation chips (Ivy Bridge) or systems with distinct GPUs.

6 Conclusion

In this paper, we presented our novel approach to use a heterogeneous environment with all it's computing devices to speed up a stream join with band condition. The join is performed within windows of the stream and scheduled as a task within a queue. New values can be joined in batches with the window of the other stream. The compute devices select a task from the queue, execute the task, and return the results before selecting the next task. Executing a task involves copying the data if needed, the execution of the band join, and returning of the data. At the moment our implementation is limited to returning only the amount of join partners, not the joined tuples. With our heterogeneous approach we achieve a speedup of 69.2x compared to the single-threaded implementation on our test machine. In our final version, we use 3 levels of parallelization: parallel execution between devices, parallel execution on devices (between cores, respectively shaders) and using build-in SIMD instructions for execution of 4 comparisons at once. Also the heterogeneous execution is 3.2x faster than the OpenCL implementation only on the CPU (respectively 1.5x compared to GPU).

Seeing modern processors becoming more and more heterogeneous and proven by our

results, we believe using these heterogeneous environments effectively can have a big impact in computing and database acceleration.

7 Acknowledgments

This work was in part funded by the European Union and the state Saxony through the ESF young researcher group IMData 100098198.

References

[Agg06] Charu C. Aggarwal. *Data Streams: Models and Algorithms (Advances in Database Systems)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[DNS91] David J. DeWitt, Jeffrey F. Naughton, and Donovan A. Schneider. An Evaluation of Non-Equijoin Algorithms. In *Proceedings of the 17th VLDB '91*, pages 443–452, San Francisco, CA, USA, 1991.

[GO03] Lukasz Golab and M Tamer Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *Proceedings of the 29th VLDB '03*, pages 500–511, 2003.

[Gro] Khronos Group. http://www.khronos.org/opencl/.

[GYB07] Buğra Gedik, Philip S. Yu, and Rajesh R. Bordawekar. Executing stream joins on the cell processor. In *Proceedings of the 33rd VLDB '07*, pages 363–374. VLDB Endowment, 2007.

[HYF+08] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD '08*, pages 511–524, New York, NY, USA, 2008.

[KLMV12] Tim Kaldewey, Guy Lohman, Rene Mueller, and Peter Volk. GPU join processing revisited. In *Proceedings of the DaMoN '12*, pages 55–62, New York, NY, USA, 2012.

[LT95] Hongjun Lu and Kian-Lee Tan. On Sort-Merge Algorithm for Band Joins. *IEEE Trans. on Knowl. and Data Eng.*, 7(3):508–510, June 1995.

[SD89] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *Proceedings of the ACM SIGMOD '89*, pages 110–121, New York, NY, USA, 1989.

[Sol93] Valery Soloviev. A Truncating Hash Algorithm for Processing Band-Join Queries. In *ICDE*, pages 419–427. IEEE Computer Society, 1993.

[TM11] Jens Teubner and Rene Mueller. How soccer players would do stream joins. In *Proceedings of the ACM SIGMOD '11*, pages 625–636, New York, NY, USA, 2011.

[VNB03] Stratis D. Viglas, Jeffrey F. Naughton, and Josef Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *Proceedings of the 29th VLDB '03*, pages 285–296, 2003.