Dieses Dokument ist eine Zweitveröffentlichung (Postprint) /

This is a self-archiving document (accepted version):

Tomas Karnagel, Dirk Habich, Benjamin Schlegel, Wolfgang Lehner

The HELLS-Join - A Heterogeneous Stream join for ExtremeLy Large windows

Erstveröffentlichung in / First published in:

SIGMOD/PODS'13: International Conference on Management of Data, New York 24.06.2013. ACM Digital Library, Art. Nr. 2. ISBN 978-1-4503-2196-9

DOI: https://doi.org/10.1145/2485278.2485280

Diese Version ist verfügbar / This version is available on:

https://nbn-resolving.org/urn:nbn:de:bsz:14-qucosa2-806668

SLUB
Wir führen Wissen.

TECHNISCHE UNIVERSITÄT DRESDEN

QUCOSA
Quality Content of Saxony

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

The HELLS-Join - A Heterogeneous Stream join for ExtremeLy Large windows

Tomas Karnagel, Dirk Habich, Benjamin Schlegel, Wolfgang Lehner

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

ABSTRACT

Upcoming processors are combining different computing units in a tightly-coupled approach using a unified shared memory hierarchy. This tightly-coupled combination leads to novel properties with regard to cooperation and interaction. This paper demonstrates the advantages of those processors for a stream-join operator as an important data-intensive example. In detail, we propose our HELLS-Join approach employing all heterogeneous devices by outsourcing parts of the algorithm on the appropriate device. Our HELLS-Join performs better than CPU stream joins, allowing wider time windows, higher stream frequencies, and more streams to be joined as before.

1. INTRODUCTION

Hardware trends are changing towards tightly-coupled heterogeneity that means different computing units are combined using a unified memory hierarchy. Fundamentally, heterogeneity is nothing new; co-processor architectures have already combined a common CPU with an accelerator like a GPU. However, these co-processor architectures are rather loosely-coupled due to the fact that each computing unit has its own isolated memory block. In this case, data has to be explicitly transferred between the different units. For many compute-intensive applications, these data transfers over the PCIe interface are sufficiently fast and the different accelerators are efficiently usable. However, the benefit for data-intensive applications in loosely-coupled environments is restricted. The performance gain from an accelerator is usually circumvented by the time needed for data shipping. Aside from data transfer as one bottleneck, a second bottleneck is the small size of the on-device memory of each computing unit. Both restrictions result in only a limited number of data-intensive applications where loosely-coupled hardware environments can be used efficiently. For data-intensive applications, a function shipping technique is more appropriate than a data shipping approach. In this case, an accelerator would have direct access to the data in a shared memory block, which avoids the previous mentioned bottlenecks. Upcoming processors like AMD and Intel mobile chips with integrated GPUs realize this shift. A large unified memory hierarchy is available and each accelerator has direct access. This tightly-coupled heterogeneous hardware approach opens a wide-range of optimization challenges for data-intensive applications [11]. A B+Tree search has been proposed as one of the first database algorithms for a tightly-coupled architecture, showing very promising results[2].

In this paper, we investigate the stream join operation as an example for a data-intensive application. A stream join is a fundamental operator in all data streaming systems [6]. 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. The performance of the system usually limits the size of the stream window, the number of streams to be joined, or the supported stream frequencies. Many stream join implementations have been proposed: sequential stream join operators [14, 4], which exploit various index data structures (e.g., hash tables) for a faster processing, as well as parallel stream join operators [3, 12, 13], taking advantage of different types of parallelism and accelerator hardware, like GPUs, FPGAs, and the Cell processor. Join operators without the streaming background have been proposed for the GPU [5, 7]. However, all of the proposed GPU implementations work batch-wise, joining two whole relations and they are not applicable with sliding stream windows. Additionally, all these approaches are only investigated in loosely-coupled hardware environments.

Our HELLS-Join approach for a tightly-coupled heterogeneous processor, consisting of a CPU and GPU, decomposes the stream join algorithm in several steps and each step is executed on the appropriate hardware. The compute-intensive parts are outsourced to the GPU while interpreting the results is conducted on the CPU. The necessary large number of comparisons can be done with a high degree of parallelism and are very suitable for the GPU. However, the GPU can not forward the result data efficiently, so this has to be done by the CPU. Our main contributions in this paper are:

- We propose an approach to decompose a stream join algorithm, employing all devices in a tightly-coupled

©2013 Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in DaMoN'13, June 24 2013, New York, NY, USA
https://doi.org/10.1145/2485278.2485280

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

hardware (CPU/GPU) system. Instead of executing an algorithm only on one hardware component, either CPU or GPU, we utilize all hardware components for several algorithm steps.

- We present an extensive evaluation showing the benefits of tightly-coupled hardware environments for our HELLS-Join. In particular, we demonstrate that our developed approach is scalable beyond on-device memory limitations as present in loosely-coupled environments.
- As we are going to show, our HELLS-Join performs better than CPU stream joins, allowing wider time windows, higher stream frequencies and more streams to join as before.

In the following section, we discuss two different CPU/GPU systems. Afterwards, we explain a naive stream join approach as foundation in Section 3. Our HELLS-Join approach is presented in Section 4, while Section 5 proposes different execution optimizations. An extensive evaluation is presented in Section 6. The paper closes with a conclusion and future work in Section 7.

2. CPU/GPU-SYSTEMS

This section gives a short overview of our underlying heterogeneous hardware infrastructures. The combination of CPU and GPU is available as loosely-coupled as well as tightly-coupled system. Therefore, we have chosen this combination to evaluate the differences of both systems for a data-intensive operation.

2.1 Loosely-Coupled Approach

The loosely-coupled combination of CPU and GPU is well-established, where a discrete GPU, denoted further by dGPU, can be used (i) for high performance image processing and (ii) as general purpose co-processor for compute-intensive applications. CPU and dGPU have their own separate memory hierarchies with different access bandwidths and latencies, and they are connected by a PCIe interface enabling necessary data exchanges. The dGPU consists of multiprocessors that are optimized for Single Instruction-Multiple Data (SIMD) executions.

2.2 Tightly-Coupled Approach

In tightly-coupled approaches, multiple different computing units are combined on one die with the advantage that bus and main memory systems are shared. In the past, this was only done in a homogeneous way by placing multiple CPU cores on one chip to improve parallelism. Due to smaller form factors, lower power consumption guidelines, and the demand for system on chips (SoC), hardware vendors integrate different processors and accelerators into one chip. The most common examples are integrated graphic processing units (denoted by iGPU). The iGPU is fully integrated and shares the main memory with the rest of the system. Examples of this kind of heterogeneous integrations are Intels® 2nd, 3rd, and 4th generation core processors for desktop and mobile systems and the AMD® Fusion processors (also called Accelerated Procession Unit (APU)). To illustrate the integration of the iGPU, Figure 1 shows the AMD® Trinity Architecture, the newest desktop Fusion processor from AMD. The iGPU and the CPU cores can access

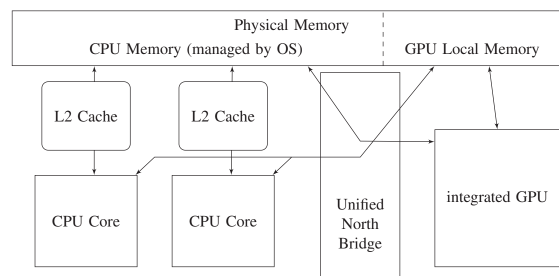

Figure 1: AMD Fusion, Trinity Architecture

the physical main memory. This memory is divided in an iGPU dedicated part and a part managed by the operating system. The partition sizes can be set on boot time. The iGPU and the CPU can access both parts of the memory using the unified north bridge; however, both devices have faster transfer rates when accessing their own dedicated memory. It is possible for the CPU and the iGPU to work on the same data stored in one of the memory parts.
At the moment, iGPUs are only integrated in desktop and mobile systems. However, recent announcements show the advances in the field. The Sony PlayStation 4 [9], for example, will have an 8 core CPU and an iGPU with 1152 processing elements, both accessing a fast GDDR5 Memory. There are also plans to equip micro server with heterogeneous CPU/iGPU processors. In the future, processors with similar properties are likely be used in large server environments, providing high performance for data-intensive applications.

2.3 Access Bandwidth Evaluation

As a first proof of concept, we evaluated the memory access bandwidth of a loosely-coupled as well as of a tightly-coupled CPU/GPU processor.
In the first part of this evaluation, we placed a predefined data block in the GPU dedicated memory. To measure the transfer bandwidth from GPU to CPU, we called a *map* and an *unmapped* command on the CPU. The *map* command maps the specified data block into the main memory, where the CPU can modify the data. If the data is not residing in the CPU's main memory, mapping results in a copy operations to the main memory, while *unmap* releases the main memory and copies the data back. As expected and depicted in Table 1, the tightly-coupled approach has as better mapping bandwidth for its dedicated GPU memory, because data is not transferred via the PCIe bus as in the loosely-coupled approach. This is clearly beneficial for data-intensive operations, where lots of data has to be processed by different computing units. Additionally, when the memory already resides in the CPU main memory

	Loosely Env.	Tightly Env.	
	Local Mem	Local Mem	CPU Mem
Avail. Mem.	1 GB	2 GB	30 GB
Map	1.47 GB/s	5.18 GB/s	3092 GB/s
Unmap	2.53 GB/s	5.09 GB/s	240 GB/s

Table 1: Memory Transfer between GPU and CPU.[1]

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

part as it is possible in tightly-coupled systems, there is no copy operation needed and the memory bandwidth increases massively (on our system, we achieved a *theoretical* bandwidth of 3092GB/s for the *map* and 240GB/s for the *unmap* operation[1]). The second part of this evaluation focuses on bandwidth for read and write operations on the GPU, where the GPU reads and writes data to direct accessible memory blocks.

	Loosely Env. Local Mem	Tightly Env. Local Mem	Tightly Env. CPU Mem
Avail. Mem.	1 GB	2 GB	30 GB
Kernel Read	35.94 GB/s	25.29 GB/s	24.6 GB/s
Kernel Write	57.99 GB/s	21.76 GB/s	6.57 GB/s

Table 2: GPU Memory Access Bandwidth.[1]

In the loosely-coupled environment, the GPU can only access its local on-device memory, while in the tightly-coupled environment accessing the CPU's main memory is a second possibility. This time, the bandwidth of the loosely-coupled environment is much better than in the tightly-coupled environment as illustrated in Table 2. The reasons are specific optimizations for memory access and the usage of faster DRAM in our discrete GPU. However, the tightly-coupled environment shows two important properties:

1. There is no significant bandwidth difference in reading data from iGPU local memory or CPU memory.
2. There is a difference in write access times between dedicated and main memory. The relatively slow write access to CPU's main memory is caused by cache snooping to the CPU caches for coherent memory accesses.

Based on our bandwidth evaluation, we are able to conclude that dedicated memory accesses are slower for the tightly-coupled CPU/GPU than for the loosely-coupled variant. However, the data transfer times to the local memory are orders of magnitude faster if not obsolete at all. Looking at the memory sizes of our example setup, it should be clear that only the CPU's memory is sufficient to support data-intensive operations. These results motivate the investigation of the stream join algorithm in a tightly-coupled CPU/GPU environment and using the integrated GPU to provide scalability for extremely large stream window sizes.

3. STREAM JOIN OPERATOR

Stream joins are heavily used in stream and event based systems. Applications like position tracking or click stream evaluation are able to produce millions of events per second. Hence, an efficient stream join approach is required to provide online evaluation of these events.
Generally, the stream join is a regular join operator, in which the input data comes from two or more streams. In this paper, we focus on a stream join with band condition as illustrated in Figure 2. The join is performed on two streams in which tuples are continuously arriving. Instead of holding all data, the continuous stream is partitioned in sliding

[1]Measurements were taken with AMD OpenCL Sample BufferBandwidth using default parameters (0% cache-hit rate). Map/unmap function call times were used for calculation. If no data is copied, the bandwidth is theoretical.

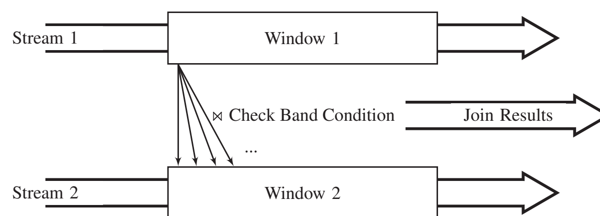

Figure 2: Stream join with band condition.

stream windows, for which the join is executed. For every arriving tuple, a join with all values in the window of the opposite stream is performed. For the example in Figure 2, a tuple arrives from Stream 1, the tuple is added to Window 1 and will be compared with all values in Window 2. For the comparisons, a band condition is applied if more than equality has to be evaluated. In this band condition case, the join tuples of Window 2 need to fall within a predefined band of the newly arrived tuple of Stream 1.
There are many variants and optimizations available for the stream join operator with band condition. Aggarwal [1] provides an excellent overview about these variants and respective sequential optimizations. Recently, parallel stream join operators gained much attention. This includes stream join approaches for FPGAs and NUMA systems [12] as well as for the Cell processor [3]. In the latter case, 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 a *nested-loop join* technique over the *hash join* or *sort-merge join* to implement the join with band conditions. There are approaches to do an efficient sort-merge join [8] or hash join [10] with band conditions but the nested-loop join is a straightforward approach for sliding stream windows. For sliding windows, there are periodically arriving and leaving tuples. For hash joins, this would mean changing the hash table on every arriving tuple. A tuple can be added to the table without a big effort but for leaving tuples the whole table needs to be scanned and timestamps need to be compared. Sort-merge joins would sort the window by the value that is compared. This would result in resorting for every arriving tuple. Also the window has to be scanned for leaving tuples (similar to hash table). The nested loop join does not keep an intermediate state like hash-tables or sorted data, so no additional update work is necessary.

4. HELLS-JOIN

In the previous sections, we discussed CPU/GPU systems and a basic stream join implementation. Having a heterogeneous hardware environment, the main challenge is to decompose the stream join algorithm in several components that can be scheduled on the appropriate device. We propose a three component division for the stream join: (i) a comparison step, (ii) an interpretation step and (iii) a step for adding tuples to a window. These steps are repeated for every arriving tuple. The several components are described as next.

4.1 Step 1: Comparison

Every arriving tuple has to be compared to all values from the opposite time window. In an empirical evaluation, we identified that the necessary tuple comparisons for large win-

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

dows are a performance bottleneck. Due to large time windows and independent comparisons, this task can be executed highly parallel, making it an ideal step for a GPU.
We hold the tuples in data structures of a predefined size. The size should be larger than the maximum amount of expected tuples in a time window (dependent on the time window and the stream frequency). Every tuple contains a timestamp, which has to be checked for every comparison. If a value falls within the time window, the band condition is checked. A join partner is found when the time window constraint and the band condition is fulfilled.
Using GPUs, we face certain limitations. Programs written in common GPU programming languages (e.g., OpenCL, or CUDA) have no possibility to output the joined data via network. A second challenge is the undefined number of join results which depend on various factors like band condition or data distribution. Furthermore, the number of the join results heavily varies over the stream join processing. That means, storing the direct join results leads to a large, often sparsely filled memory object. This straightforward approach introduces a large memory overhead and would limit our HELLS-join techniques to small windows sizes.
To overcome this limitation, we propose a method of *result compression*. Instead of storing the direct join result, we set a bit for every found join match. The position of the bit indicates the join partner for later processing. Because of memory access patterns of the GPU, only 32 bit results are written. This means, that each GPU-thread does 32 comparisons to write a 32 bit result bitmap. The bitmaps of all the work-items are placed in a data structure for later interpretation. The comparisons and intermediate result storage work as follows:

1. Start GPU-program with arriving tuple.
2. Comparisons within the GPU-program.
   (a) Compare timestamps and check time window constraint.
   (b) Compare value and check band condition.
3. Encode the join results as bitmap.

These steps are illustrated as an example in Figure 3. A tuple with the timestamp $t = 130$ and a value $v = 456$ arrives. It is compared to all the tuples in the opposite window, meaning window sizes and the band conditions are checked. Only the tuple $(t = 120, v = 450)$ satisfies these constraints and is encoded as a set bit. The position of the set bit identifies the found joined tuple (here 5th from top).

4.2 Step 2: Interpretation

Naive CPU stream join implementations do not need a separate result interpretation because results are processed during the comparison phase. However, when executing the comparison step on the GPU, we have to introduce this algorithm component. The interpretation component is able to access the resulting bitmaps of the previous component when the processing is finished. The task of this component is to scan the intermediate data structures for the set bits. When a position is found, the interpretation component constructs the join result for further processing.
This interpretation is not as expensive as the comparison step. Checking for set bits is less computation than checking the time window constraint and the band condition. Furthermore, with the bitmap, it is easy to scan a group of bits if they are set (e.g., 32 bits at one time). Only if bits are set in such a group, the search for set bits becomes more fine grained. This is especially fast if none or only a few results need to be returned. This interpretation component is best suited for the CPU.

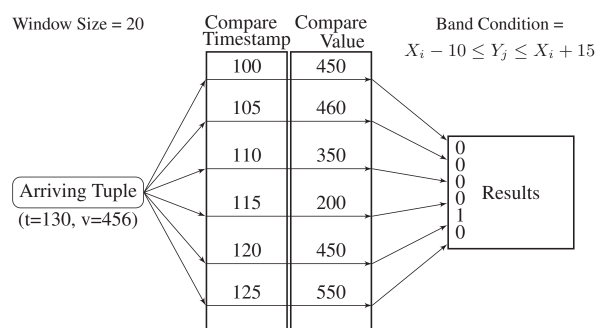

Figure 3: Parallel comparisons of one tuple to a window of tuples.

4.3 Step 3: Adding new Data

Despite the comparison, new tuples need to be added to a time window. In our fixed size data structure, this is done regarding the first-in-first-out (FIFO) concept. While this is an easy task for a CPU managed data structure, it is rather a problem for GPU managed memory. There are three ways to add a value to GPU managed memory:

1. Mapping the data structure to CPU memory and modifying the tuple within the structure.
2. Copying the tuple directly into the window data structure with language specific copy commands.
3. Starting an GPU program with the tuple as argument. It then writes the tuple on the specified location.

For a dGPU, the first approach would lead to expansive memory transfers and is not applicable. For iGPUs, the memory transfers are not as problematic but mapping would mean to stop the execution until the the memory is released (unmap). Even for a short operation like adding a value, this would stop the command pipeline and reveal the high call latency, which GPUs have in general. The second and third way of adding a value hides the latency because they can be queued between normal GPU-program executions. We found that the third way, launching a GPU-program with one thread to write the tuple, fits best into the command pipeline and therefore bringing best performance results.

5. EXECUTION OPTIMIZATION

In the previous section, we have proposed our novel HELLS-Join technique. In this section, we are applying hardware specific optimizations to enhance the performance. Our target platform is an iGPU of the Trinity architecture (as shown in Figure 1). However, for small stream windows, we want to use our implementation with dGPUs as well. In the following, we propose a way of optimal memory management for

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

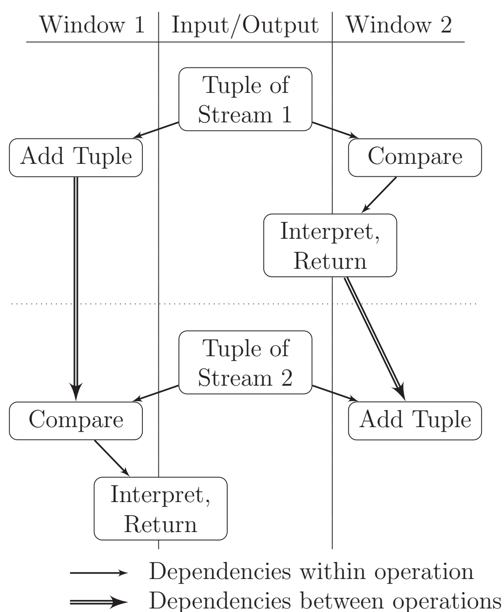

Figure 4: Execution pattern with a synchronous interpretation step.

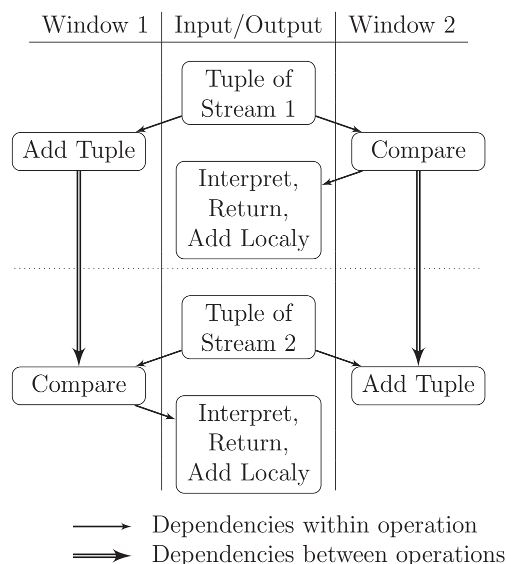

Figure 5: Execution pattern with a asynchronous interpretation step.

iGPUs and dGPUs and introduce a way to enhance parallel execution by reducing dependencies.

5.1 Memory Placement

For the memory placement, we have to differentiate between the dGPU and the iGPU. The dGPU has a fixed local memory with a high memory access bandwidth. Most dGPUs are not able to directly access CPU's main memory, so the local device memory must suffice. Here, we have to store both stream windows and the result data structures within the device memory. If both windows do not fit within the device memory, then a window (or a part of it) needs to be stored in main memory and copied if needed. These copy operations introduce an significant overhead to the normal execution and should be avoided.

Using an iGPU, we have two choices to store the data: the local memory or the OS managed memory. Even both memory locations are on the same physical memory, there are different access bandwidths for each of them. As shown in Table 2, reading from both parts of the main memory has a similar bandwidth, because a specialized GPU bus (*Radeon Memory Bus*) is used for the access. Also, the memory is allocated as uncached by the CPU, leading to these high GPU bandwidths. This means that the window data can be stored in the CPU managed main memory without bandwidth losses on read access, allowing significantly bigger stream windows to be stored and processed. For the iGPU, the result data structures are also stored into CPU memory. Due to the small size of the results, the effects from the low write bandwidth (Table 2) are not as significant as the high access bandwidth of the CPU (Table 1).

5.2 Parallel Execution

Given our algorithm with three steps, the aim is to run the steps in parallel on the GPU and the CPU. The three steps are illustrated in Figure 4 for two arriving tuples. First, a tuple of Stream 1 arrives, shown in the Input/Output section. Adding the tuple to Window 1 and comparing the tuple to Window 2 can be done in parallel because the two operations do not depend on each other. Interpreting the result is triggered after the comparison finished. Then, the resulting bitmap needs to be evaluated. Tuples are fetched from the data of Window 2 according to the bitmap, to combine the final join results. Therefore, Window 2 can not be modified until the interpretation is done. When a tuple from Stream 2 arrives, it can not be added to Window 2 until the interpretation step finished accessing the data. Reading the data of Window 2 in the interpretation step is stalling the execution until the interpretation is finished. Also there are multiple expansive copy operations when the dGPU is used. Therefore, we propose storing the window data twice. The first copy is reserved for the comparisons, while the second copy is used for the interpretation step. This way, the dependencies are reduced and the interpretation step can run totally asynchronous to the comparisons and tuple adding. This is illustrated in Figure 5. During the interpretation step, the tuple is added to a local window copy. To ensure correctness, the interpretation must be done in the same order as the comparisons. This way, we can ensure that the comparisons and the interpretation step work on the same data.

In our implementation, the operations on Window 1 and Window 2 are done on the GPU. For each window, we maintain a separate in-order command queue (using OpenCL). There, commands are queued and executed in order. Commands in different command queues can be executed in any order because they should not depend on each other. This gives the GPU the flexibility that is needed to hide latency and schedule the commands as good as possible.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

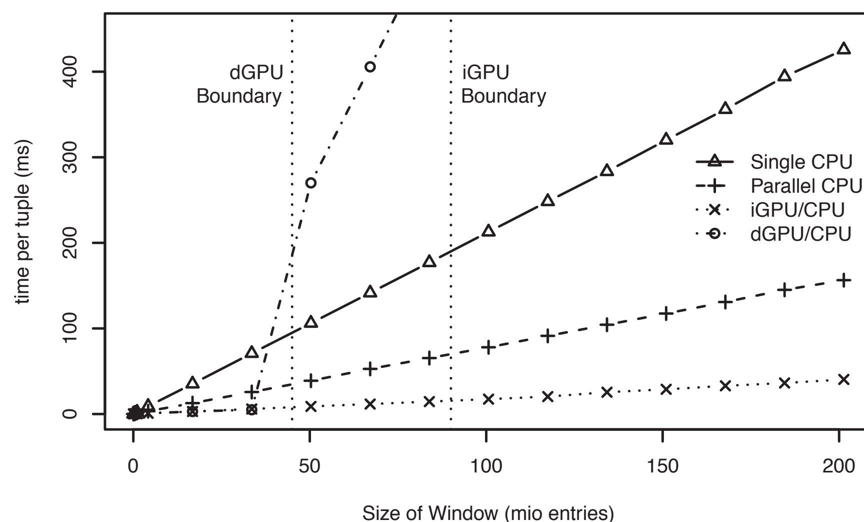

Figure 6: Scalability with window size.

6. EVALUATION

We implemented our HELLS-Join in C++ and OpenCL. It performs the three steps as described in Section 4. We applied our memory considerations and the dependency improvement for a better parallel execution. Our test environment consists of an AMD APU (A10-5800K) with a memory access pattern as shown in Figure 1 and a discrete NVIDIA GPU (Geforce GTX 295). The properties of the CPU and GPU are compared in Table 3. The Nvidia GPU consists of two chips on one board. For comparison and complexity reasons, we only use one of those chips. Comparing one NVIDIA GPU chip with the AMD iGPU shows, that the iGPU has only 69% of the compute performance as the dGPU. However, the transfer bandwidth is higher for the iGPU and there are more memory options to choose from (Table 1 and Table 2 represent these two GPUs).

We evaluated our implementations concerning the scalability with window sizes in Figure 6. All implementations have a linear scaling behavior for small window sizes. However, when the dGPU/CPU (loosely-coupled) implementation reaches the limits of the device memory (dGPU Boundary), further execution is limited by data transfers for swapping. These data transfers have an significant influence on execution times resulting in slower execution than single thread performance. Avoiding transfers, the iGPU version scales linearly even beyond its on-device memory (iGPU Boundary). The data is residing completely in the CPU managed memory space, so the device memory boundary has no impact. The speedup of the iGPU version is not significant for small windows, however, for large windows, the speedup reaches 12.3x compared to single threaded execution and 4.5x compared to multi threaded execution.

The test was done on random data with a selectivity around 0,004% per tuple. This means for each tuple, 0,004% of the tuples within the window matched and were joined. For a large window like e.g., 200 million entries, this would mean 8000 join results per arriving tuple. We believe an output of that size is realistic for common stream join applications like joining vehicle positions over a long time period. There, positions usually differ enough, resulting in only a small subset of the data being joined with the arriving tuple. However, we evaluated higher selectivities, where the interpretation step shows increasing significance on the execution time. We have seen a break even point with the CPU version around 40% selectivity and a performance drop of 0.5x for 100% selectivity. In these unlikely cases, the bottleneck shifts from the comparisons to the interpretation. In these unusual settings, a pure CPU version performs better.

	AMD CPU	A10-5800K HD 7660D	NVIDIA GTX 295
Compute Units	4	6	2 x 30
Cores	4	384	2 x 240
Frequency (MHz)	3800	800	1242
Perf. (GFLOPS)	121.6	614.4	2 x 894.2
Max Power Cons.	100 W		300 W

Table 3: AMD APU and Nvidia GPU

7. CONCLUSIONS AND FUTURE WORK

In this paper we propose the HELLS-Join, a heterogeneous stream join on loosely and tightly-coupled hardware environments (CPU/GPU) for extremely large stream windows. We decompose the stream join algorithms, schedule the components on CPU and GPU devices, and introduce optimizations that enhance memory placement and parallel execution. Furthermore, we use an integrated GPU to enable scalability beyond device memory reaching speedups of over 12x compared to single threaded performance. With our implementation, we evaluate integrated GPUs for highly parallel and data intensive tasks. We show that integrated GPUs have reached a performance level that is useful for compute intensive algorithms combined with data sharing options for data intensive algorithms. Integrated GPUs essentially solve data storage and transfer problems of discrete GPUs, making GPUs ready for data intensive applications.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden

Final edited form was published in "SIGMOD/PODS'13: International Conference on Management of Data. New York 2013", Art. Nr. 2, ISBN 978-1-4503-2196-9
https://doi.org/10.1145/2485278.2485280

Our stream join implementation is highly optimized for the AMD Trinity architecture and shows very promising results. It is not foreseeable if these results are similar on coming architectures or hardware from different vendors. For portability, future work could include a general approach for deciding memory placement and execution devices on runtime to adapt to different hardware properties.

8. ACKNOWLEDGMENTS

This work is partly funded by the German Research Foundation (DFG) within the Cluster of Excellence "Center for Advancing Electronics Dresden", by the European Union together with the Free State of Saxony through the ESF young researcher group "IMData" 100098198, and by the iBIT project, as part of the Leading-Edge Cluster "Cool Silicon" 100067363.

9. REFERENCES

[1] C. C. Aggarwal. *Data Streams: Models and Algorithms (Advances in Database Systems)*. 2006.

[2] M. Daga and M. Nutter. Exploiting coarse-grained parallelism in b+ tree searches on an apu. In *Proceedings of the 2012 SC Companion: High Performance Computing, Networking Storage and Analysis*, SCC '12, pages 240–247, Washington, DC, USA, 2012. IEEE Computer Society.

[3] B. Gedik, P. S. Yu, and R. R. Bordawekar. Executing stream joins on the cell processor. In *VLDB '07*, pages 363–374, 2007.

[4] L. Golab and M. T. Özsu. Processing sliding window multi-joins in continuous queries over data streams. In *VLDB '03*, pages 500–511, 2003.

[5] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *SIGMOD '08*, pages 511–524, 2008.

[6] J. Kang, J. Naughton, and S. Viglas. Evaluating window joins over unbounded streams. In *ICDE'03*, pages 341–352, 2003.

[7] M. D. Lieberman, J. Sankaranarayanan, and H. Samet. A fast similarity join algorithm using graphics processing units. In *ICDE '08*, pages 1111–1120, 2008.

[8] H. Lu and K.-L. Tan. On sort-merge algorithm for band joins. *IEEE Trans. on Knowl. and Data Eng.*, 7(3):508–510, June 1995.

[9] S. E. playstation.com. Playstation4 specification.

[10] V. Soloviev. A truncating hash algorithm for processing band-join queries. In *ICDE*, pages 419–427, 1993.

[11] K. L. Spafford, J. S. Meredith, S. Lee, D. Li, P. C. Roth, and J. S. Vetter. The tradeoffs of fused memory hierarchies in heterogeneous computing architectures. In *CF'12*, pages 103–112, 2012.

[12] J. Teubner and R. Mueller. How soccer players would do stream joins. In *SIGMOD '11*, pages 625–636, 2011.

[13] K. Tsakalozos, M. Tsangaris, and A. Delis. Using the graphics processor unit to realize data streaming operations. In *MDS '09*, pages 3:1–3:6, 2009.

[14] S. D. Viglas, J. F. Naughton, and J. Burger. Maximizing the output rate of multi-way join queries over streaming information sources. In *VLDB '03*, pages 285–296, 2003.

Provided by Sächsische Landesbibliothek, Staats- und Universitätsbibliothek Dresden