CONCURRENCY AND COMPUTATION: PRACTICE AND EXPERIENCE
Concurrency Computat.: Pract. Exper. 2017; **00**:1–22
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/cpe

GPU processing of theta-joins

Christos Bellas and Anastasios Gounaris*

Department of Informatics, Aristotle University of Thessaloniki, 541 24, Hellas

SUMMARY

The GPGPU paradigm has been recently employed to accelerate the processing of big amounts of data through the utilization of the massive parallelism offered by modern GPUs. To date, several techniques have been proposed for the implementation of simple select, aggregate and equality join operations on GPUs. In this paper, we study the efficient implementation of theta-join queries between two relations using the CUDA framework. Theta-joins are notoriously slow and thus can benefit from massively parallel execution. However, their GPU-based implementation significantly differs from hash- and sort-based equality joins and needs to be carefully crafted. The implementation is driven by two main objectives. The first relates to the attainment of high efficiency in the parallelization through data reuse, which relates to the minimization of accesses to the slow global memory. The second is about the most efficient exploitation of the available memory given that, in general, it cannot hold the entire input and result. We propose a methodology for processing theta-joins on a GPU, which exploits the heterogeneous nature of GPGPU, while addressing memory limitations. Furthermore, we provide a series of implementation optimizations, which yield performance improvements of an order of magnitude. Copyright © 2017 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: gpgpu, cuda, parallel query processing, joins

1. INTRODUCTION

Big data analysis has been recognized as playing a key role in a growing number of scientific disciplines, such as genome analytics and earth sciences, and business operations, such as decision support, personalized recommendations and usage trend prediction. At a lower level, big data analysis very commonly employs queries with relational operators coupled with custom code and machine learning/data mining modules, e.g., as in Spark SQL [1].

One of the most persisting topics in query processing is efficient join processing. The most general but less studied form of joins is *theta (or inequality)* joins. A theta-join is an inner join operation, which combines tuples from different relations, if and only if a given theta (θ) condition is satisfied. It generalizes equality joins (or equi-joins) as it allows the use of inequality operators within the join condition. In a generic theta-join, all possible combinations (i.e., the cartesian product) may need to be evaluated, which leads to $O(n^2)$ time complexity. Additionally, the worst case space complexity is when $O(n^2)$ combinations of records satisfy the theta condition. In practice, a theta-join over a couple of hundred thousands of records may take hours in a modern DBMSs [2], whereas an equi-join may complete in a few seconds or less.

In this work, we deal with the problem of processing theta-joins on a graphics card efficiently with a view to benefiting from the massive parallelism that such cards can provide. Graphics cards are increasingly used in parallel applications [3]. In this work, we use the terms GPU and graphics card interchangeably. GPUs have been proposed for simpler relational operators, such as sorts, selects,

*Correspondence to: gounaria@csd.auth.gr

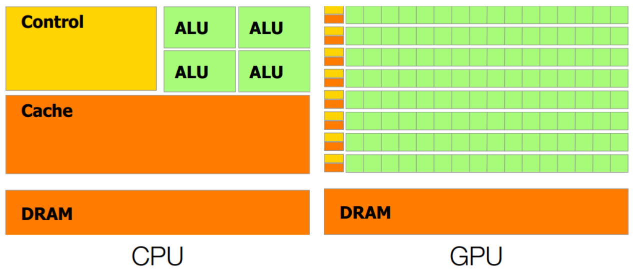

Figure 1. CPU vs GPU (from [9])

aggregates and equi-joins, e.g., in [4, 5, 6, 7], but to date, no solution tailored to theta-joins on GPUs has been proposed. Our proposal is the first one that explicitly targets theta-joins using a GPU. The closest work to us is the consideration of non-indexed nested loops in [4], which can cover theta-joins as well as equi-joins. Compared to [4], we elaborate on all practical details, including the handling of join output, whereas in [4], nested loops are discussed at a high level only.

Our solution follows the GPGPU (General-purpose computing on graphics processing units) paradigm, where the CPU, which is termed also as *host*, collaborates with the GPU, which is termed also as *device*, to evaluate a theta-join query. The CPU is responsible for splitting the workload to chunks of appropriate size and the GPU carries out the main theta-join processing. The implementation is in CUDA.

The key features of our solution are summarized as follows: (i) we make no assumptions about the θ condition and we do not rely on any type of pre-processing or knowledge about the data distribution and selectivity; this renders our proposal as generally applicable as possible; (ii) we manage to maintain a high processor utilization, despite the inherent presence of thread branching, i.e., threads follow a different execution path according to the result of the predicate evaluation, which typically returns both true and false values; and (iii) we efficiently address issues related to data reuse, minimization of accesses to the slow device memory, limited memory, exploitation of the memory hierarchies on a typical GPU and data layouts facilitating memory accesses. Overall, we provide a series of optimizations, and we experimentally show that these optimizations can yield speedups up to an order of magnitude when applied on a GPU of the Maxwell micro-architecture, which is the latest one from NVIDIA.[†]

The remainder of this article is structured as follows. The next section provides a concise overview of the CUDA architecture and programming framework. We also discuss a solution for theta-joins in a MapReduce environment, upon the parallelization of which we have built the part of our solution that refers to splitting the workload into chunks on the CPU side. Section 3 presents the generic framework of our solutions. The core technical contributions are in Sections 4 and 5, which deal with input- and output-bound theta-joins queries, respectively. We discuss the related work in Section 6, and we conclude in Section 7.

2. BACKGROUND

In order to make the text self-contained, we provide background details on CUDA architecture and programming, explaining the main concepts. Also, we briefly present a solution to theta-joins for MapReduce in a shared-nothing parallel environment [8], which, despite being significantly different from ours, can be leveraged to build a solution for GPUs, when the input is too large to fit in the device memory.

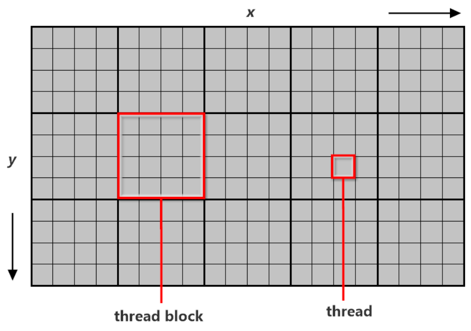

Figure 2. An example grid with 5X3=15 blocks, each consisting of 16 threads.

2.1. *Architecture of CUDA enabled GPUs*

GPGPU aims to take advantage of the different and complementary characteristics offered by CPUs and GPUs. As shown in Figure 1, a GPU has much more processing units, also referred to as cores, but less cache and control units compared to a CPU. GPU cores execute the same instruction on different data items, whereas CPU cores typically execute different instructions thus raising the need for more advanced program flow control. In general, modern CPUs are designed to minimize the latency of individual threads, whereas GPUs target throughput maximization [10].

CUDA is a GPU programming model invented by NVIDIA.[‡] Several other models exist, e.g., OpenCL. One of the main advantages of CUDA is that it renders GPU programming easier [11]. The implementation in this work is based on CUDA version 7.5.

CUDA-enabled GPUs divide the cores into groups called *Streaming Multiprocessors (SMs)*. Each SM follows the *Single Instruction Multiple Data (SIMD)* parallel processing paradigm. The hardware micro-architecture of CUDA-enabled GPUs has been evolving in the recent years. The most advanced micro-architecture at the time of writing is called Maxwell, and is the one assumed in this work. However, all our results will apply to the forthcoming (not yet released) generation, coined as Pascal, since the techniques and optimizations that we present in this work are orthogonal to the hardware improvements that are envisaged in the Pascal micro-architecture.[§]

2.2. *Thread Organization*

In CUDA, each function to be processed in parallel by the GPU is called a *kernel*. Each kernel is executed by multiple threads, where each thread is defined as a sequence of instructions. In our case, a thread is responsible for evaluating the theta condition over one or more pairs of records.

Threads are grouped in *thread blocks*. All threads in the same block are executed on the same SM and are allowed to inter-communicate. CUDA groups blocks thus forming a *grid* (see Figure 2). The coordinates in the figure are used as thread identifiers.

CUDA hardware can schedule multiple blocks to a SM according to the capacity of each SM in terms of concurrent blocks and active threads. Scheduled blocks are further partitioned in groups of 32 threads called *warps*. For example, assume that a thread block has $16 \times 8 = 128$ threads. This is internally split into $\frac{128}{32} = 4$ warps. Warps are currently the smallest unit of execution. For example, if a block has 140 threads, then this will correspond to 5 warps. The last warp will contain only 12 threads, which will result in 20 cores being idle during its execution.

All threads in a single warp are executed simultaneously in a SIMD manner. At any time, a SM chooses the warps that are ready for execution, so that long waiting times (i.e., waiting for other warps to fetch operands) are hidden [11]. In Maxwell GPUs, each SM has 128 cores and 4 warp schedulers; so, all processing units can be occupied simultaneously.

[†]Our code is publicly available at https://github.com/chribell/cuda-theta-join
[‡]http://www.nvidia.com/object/cuda_home_new.html
[§]Since Maxwell, SMs have been also referred to as SMMs.

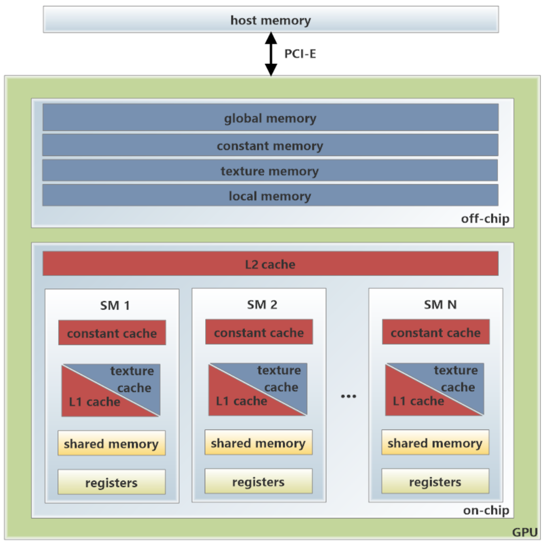

Figure 3. Memory hierarchy for Maxwell GPUs

Capacity restrictions play an important role in our theta-join processing techniques, as will be shown later. There are hardware constraints on the number of (A) resident (or else, scheduled or active) blocks per SM, (B) the number of resident warps per SM, (C) the number of scheduled threads per SM, and (D) the number of threads per block. As such, the number of active threads in a kernel cannot exceed the quantity $min\{(A) \times (D), 32 \times (B), (C)\}$ multiplied by the number of SMs on the GPU.

2.3. Memory Hierarchy

The memory structure on a GPU is hierarchical, as depicted in Figure 3. GPU has off-chip memory modules of several gigabytes that are connected to its host counterpart through a PCI-E link. In the future, faster link technologies, such as NVLink may be employed [10].

Off-chip memories include the *global, constant, texture* and *local* ones. The global memory is the largest but slowest memory, and it is the place where data are transferred from the host through the `cudaMemcpy()` command. The contents of the global memory can be cached in the L1 and L2 on-chip caches. The constant memory is much smaller (e.g., 64 KBs), but leads to significantly shorter access times. It is used for read-only data and is cached in a different place than L1 and L2 caches. The key reason for the short times is that a single thread can read once a value from constant memory and broadcast the value read to all the other threads in the same half-warp, thus reducing reads by 93.75%. The texture memory is also read-only, and is optimized for access patterns where subsequent accesses are spatially close. The local memory is part of the global memory and is used when the registers needed for a thread are full or cannot hold the required data. Each thread gets allocated its own part of the local memory.

The on-chip memory modules consist of the *shared* memory, the caches, and the registers. The latency of shared memory is two orders of magnitude lower than that of the global memory. Moreover, the contents of shared memory are accessible by all threads in the same block. This feature renders shared memory particularly useful for optimization of memory accesses and is leveraged in our techniques. The size of shared memory in Maxwell GPUs is 96 KBs. The L1 and L2 caches serve the global memory, and can spill data to the local memory. In the Maxwell micro-architecture, there is a single L1 per SM, and one shared L2 for all SMs. There are also caches for the constant and the texture memory. Finally, the registers are the fastest memory modules, similarly to CPU chips. The registers are visible to specific threads, and contain the instructions of a single thread and the local variables. If a thread requires more registers than a threshold, this leads to a reduction of the threads allowed to run concurrently, as discussed previously.

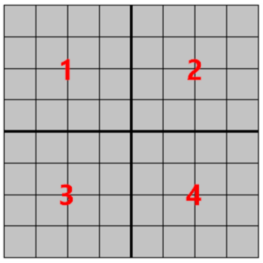

Figure 4. A simple 8X8 join matrix and its partitioning into 4 square regions

2.4. *Measuring Performance*

In this work, performance is measured in terms of the running time of the theta-join processing. Measuring the running time through synchronization of the kernel execution on the host side, e.g., with the help of the `cudaDeviceSynchronize()`, is not always appropriate, because it blocks the GPU pipelined execution. The main alternative is to use the CUDA `Event` API. More specifically, using the `cudaEventRecord()` function, we can create a timestamp for the event of starting the theta-join execution on the GPU. Using the `cudaEventSynchronize()`, we block the CPU processing until this event is recorded on the host. Then, using the `cudaEventElapsedTime()`, we can get the elapsed time between the events at the beginning and the end of theta-join execution.

2.5. *Theta-joins in MapReduce*

Apart from [4], the closest work to ours is the investigation of theta-joins in a different massively parallel setting, namely the one that is typically assumed in MapReduce programs [8].

In a MapReduce setting, multiple machines work in parallel and inter-communicate through the network. The main problem addressed in [8] is that each machine may have memory limitations, so that the goal is to allocate to each machine as small workload chunk in terms of memory requirements as possible. To this end, it is helpful to represent the work of a theta-join as a matrix, the *join matrix (JM)*. A join matrix has as many rows (resp. columns) as the number of records in the first (resp. second) relation to be joined. Figure 4 shows a matrix that represents the theta-join between two relations, each having eight records. Each cell in a JM represents a unit of work in the theta-join, which is equal to the evaluation of the theta predicate of a single pair of records.

The key contribution of the work in [8] is the proof that, given the number of the reducer machines available, the memory requirements are minimized, when each reducer machine gets an amount of the workload, which is visually represented as a square region in the JM. For example, in the figure, four reducers are assumed and the workload is split into four square regions. Note that in the generic case, JM is not square. The special case where one input relation is much smaller than the other is also considered.

The main difficulties in transferring the solution in [8] to our setting is that, if, on the one side, each thread becomes responsible for a single cell, then the amount of cells required may well exceed the GPU capacity. On the other side, if a thread becomes responsible for multiple JM cells, then load balancing becomes even more difficult. Finally, if the theta-join, and consequently the JM, is processed by multiple kernels, this aggravates the overhead of data shipping from the host to the device due to data replication. In the example in the figure, each record of each relation is transferred to two reducers. The smaller the reducer regions, the higher the replication factor. Consequently, there are significant trade-offs between the various workarounds. In the next section, we explain how we partially build on the results of [8] for allocating workload to the device. Then, we follow a technique that is novel and tailored to GPUs for the kernel execution.

3. OUR MAIN FRAMEWORK

In this section, we discuss our choices on how to divide the work between the host and the device. Then, in the following sections, we provide the implementation details for the local processing on the GPU.

3.1. Concept Mapping

In order to transfer results from the solution for the MapReduce paradigm discussed in Section 2.5 to the GPGPU paradigm, we first need to understand the association and the differences between the concepts of the two frameworks.

The map process in [8] is responsible for enforcing the desired data partitioning and does not contribute to actual theta-join processing. The latter is performed by the reducers in parallel using a single phase. In our GPGPU setting, the responsibility for data partitioning rests with the host. However, we may perform workload allocation to our parallel processors in multiple iterations. We further elaborate on this issue in Section 3.2.

A second difference is that, in the MapReduce paradigm, a reducer machine is a distinct processing unit with its own main memory and processing elements, to which part of the workload is allocated. All reducers run in parallel, whereas interaction between reduce-side processing of different key-value pairs on the same reducer machine is possible [12]. At the hardware level, a CUDA-enabled GPU has a number of mutually independent SMs. As such, SMs can be regarded as analogous to reducer machines. However, at the software level, the workload of a kernel is divided in thread blocks. Thread blocks are processed in parallel by the SMs and multiple blocks can be simultaneously resident on the same SM, as already explained. Consequently, the division of workload needs to be at a finer granularity than in MapReduce-based theta-joins, where it is adequate to divide the workload into as many pieces as the number of reducer machines.

3.2. Memory Limitations

Here we argue about our choice to consider multiple iterations of workload allocation. GPUs are more suitable for problems of fixed input and output size, so that the exact amount of required memory can be accurately estimated before execution and consequently, no extra logic to co-ordinate writes and handle conflicts is required. In theta-joins, we need to mainly focus on output because the memory requirements of the input are much lower. For example, consider a join between 50K $\times$ 25K tuples of 8 bytes each. The whole input can fit into less than 1MB, but the cartesian product requires approximately 20GBs. A 10-fold increase in each of the input relations incurs a 100-fold increase in the output, i.e., the input fits into 10MBs, while the output is exceeds 1TB. In generic theta-joins, there is no prior knowledge about the exact output size. So, we have to account for the worst case.

To tackle this issue, we investigate two possible approaches. In the first one, we assume that the device processes a whole JM in a single execution. The most straightforward implementation is to split the device memory that is available (i.e., not occupied by the input) into as many buffers as the number of SMs. When a buffer becomes full, the execution is suspended until a flushing operation, which transfers the partial output to the host memory through the PCI-E bus, is completed. To perform this, we need to know the SM that executes a specific thread. CUDA API does not provide this kind of information directly, but we can retrieve the SM identifier by executing a PTX[¶] command. The main drawback of this approach is that it leads to poor performance because the parallel execution is suspended frequently, and also, the device code becomes more complex.

In the second approach, which is the one followed in this work, the host iteratively feeds the device with as many input data as the GPU memory can support (i.e., their cartesian product fits into the remaining global memory). In this way, we assign control flow activities to the host side and avoid complex memory handling operations on the device.

[¶]PTX is a pseudo-assembly language used for code translation in the CUDA framework

R, S	Relations to be joined
O	Output relation
r, s, o	Tuple sizes in bytes
$\|R\|, \|S\|, \|O\|$	Cardinalities of relations
$\|\|R\|\|, \|\|S\|\|, \|\|O\|\|$	Sizes of relations in bytes
$\|\|O\|\|_{max}$	Maximum space needed in bytes
$\|R'\|, \|S'\|$	Cardinalities of sub-relations in each iteration
M_h	Host memory
M_d	Device memory
T	Number of threads executed
$B = B.x \times B.y$	Thread block size and its dimensions
p	Chunk size

Table I. Notation

3.3. *High-level Approach*

Here, we present the high-level summary of our solution for $R \bowtie_\theta S$ joins. The main notation is shown in Table I.

3.3.1. *Host Role.*

In each iteration, the host sends to the device a single partition with as few data as possible in order to maximize the amount of memory available for the output, while balancing the workload across iterations. Each partition is processed in parallel by the GPU SMs, however different partitions are processed sequentially. More specifically, in each iteration, sub-relations $R' \subseteq R$ and $S' \subseteq S$ are sent to the device so that their cartesian product fits in the device memory M_d. The lower bound of number of partitions, which equals to the lower bound on iterations, is calculated as follows: $t = \lceil ||O||_{max}/M_d \rceil$, where $||O||_{max} = |R| \times |S| \times o$.

There are several ways to perform the partitioning. Leveraging the theorems presented in [8], which ensure that the memory requirements for holding the input are minimized, we distinguish between three cases: (i) both R and S are multiples of $\sqrt{|R||S|/t}$; (ii) one relation is much smaller than the other one, and more specifically $|S| < |R|/t$; and (iii) one relation is less or equal than the other one and $|R|/t \leq |S| \leq |R|$ is satisfied.

The first two cases are treated similarly to [8]. In the first case, the JM is partitioned into squares of side $\sqrt{|R||S|/t}$, i.e., $R' = S' = \sqrt{|R||S|/t}$. In the second case, i.e., when one relation is much smaller than the other one, the matrix is partitioned into rectangles of $|S| \times |R|/t$, i.e., $S' = S$ and $R' = |R|/t$ (here, without loss of generality, we assume that S is the smaller relation). In the third case, first, we create as many square partitions of size $\sqrt{|R||S|/t}$ as possible. Then to cover all the JM cells, we create smaller rectangle partitions. This case is treated differently than in [8] and involves a trade-off between aggravating the memory requirements and increasing the number of iterations. In our approach, we trade memory requirements at the expense of more iterations.

3.3.2. *Device Role.*

In each iteration, the device becomes responsible for a rectangle region of the JM, corresponding to a $R' \bowtie_\theta S'$ join. Since JM are two-dimensional structures, threads are grouped in blocks of two dimensions, which, in turn, are organized in a grid of two dimensions, exactly as shown in Figure 2. As such, the problem of efficient processing theta-joins on a GPU is essentially reduced to the lower-level problems of deciding on (i) the exact role of each thread and (ii) the thread organization.

A subtle detail has to do with the memory allocation on the host to store the intermediate results produced by the device in each iteration. If memory is allocated through the traditional `malloc()` C function, memory paging can take place. But if it is allocated through the `cudaHostAlloc()` function, provided by the CUDA API, then this memory is pinned and thus the system allows accessing through the memory physical address. Across iterations, and in cases where the host memory M_h is not sufficiently large to store the complete final output, we need to access the secondary storage. Therefore, the existence of a function, which transfers partial output results from the primary to the secondary host memory, is assumed to be in place. Nevertheless, the time overhead of this function is fully hidden by the processing on the device side. Finally, we assume that the host memory is not smaller than the device memory, i.e., $M_h \geq M_d$ always holds.

3.4. *Types of Queries Investigated*

We focus on two types of theta-join queries. The first one performs an aggregation on top of a theta-join. This type is characterized as input-bound, since the size of the output is very small. Consequently, its processing can typically be performed in a single iteration, i.e., $R' = R$ and $S' = S$ unless the input sizes are very large, which is rather unlikely in theta-joins. Larger inputs are not a big problem because we can apply the partitioning described previously. Dealing with theta-joins combined with an aggregation, apart from being interesting and useful in its own right, allows us to concentrate on implementation details that are orthogonal to the size of the output.

The second query type corresponds to generic theta-joins, where we need to shift our focus on writing the result, which typically does not fit into the device memory. I.e., the generic theta-joins are clearly output-bound.

In the next two sections, we analyze the approaches followed for the effective implementation of these two types of queries, respectively. In some parts, to facilitate understanding, we give exact numbers for our implementation decisions according to the characteristics of the GM204-200 or simply GTX 970 GPU (compute capability *5.2*). Our design decisions are largely driven by the technical specifications of GPUs. However, they are hardware- and framework-independent, in the sense that they can apply to any GPU after a straightforward parameter adjustment.

4. INPUT-BOUND THETA-JOINS

First we investigate the implementation of a theta-join followed by an aggregation. Through this simplification, we can focus on efficient thread specification and indexing without worrying whether the size of the produced results is larger than the device memory.

Without loss of generality, we assume that the relations to be joined consist of three simple integer attributes: a tuple identifier *id*, and two additional fields, *a*, *x*, which are used in the theta-join and aggregation conditions, respectively.

```
typedef struct {
  int id;
  int a;
  int x;
} Tuple;
```

Listing 1: Tuple representation (AoS)

The simplest format to store tuples in memory is referred in literature as Array of Structures (AoS). In AoS, the relations are stored as arrays, and the content of each array cell is a tuple as shown in Listing 1. However, this data layout is avoided and GPU data management solutions prefer an alternative, which is called Structures of Arrays (SoA) (shown in Listing 2). In SoA, there is a different array for each attribute, which is shared among all the tuples in the relation. For completeness we examine both data layouts, starting from the AoS one.

```
typedef struct {
  int id[N];
  int a[N];
  int x[N];
} Relation;
```

Listing 2: SoA data layout for a relation

The aggregate function used in the query calculates a *sum* on the $S.x$ field provided that the theta-join condition is satisfied as shown below. Consequently, the device's output is a single number.

```
SELECT SUM(S.x)
FROM R, S
```

```
WHERE R.a > S.a;
```

Listing 3: Aggregate theta-join query

4.1. *Partitioning the Join Matrix in Threads*

Initially, the workload allocated to each thread concerns the evaluation of the join condition on a single pair of tuples. As such, the size of the flattened grid is the size of the JM. Also, due to the 1-to-1 correspondence between the grid and the JM, tuple mapping is indirectly performed through the thread indexing and vice versa.

Then, the main issue is how to partition the grid in thread blocks. Since there are no memory constraints, one could be tempted to divide the JM in as many thread blocks as the number of SMs. However, this is practically impossible due to limit on the number of threads per block in modern GPUs. For example, in the GTX 970 graphics card, there are 13 SMs and the maximum number of threads per block is 1024. A very small theta-join over two relations of 2K tuples each, would require $\frac{2000\times2000}{13} > 307K$ threads per SM, which is two orders of magnitude higher than the current limit. So, driven by the GPU specifications, we need to consider larger grid dimensions for the kernel launch not to fail. The resulting grid has size of $\lceil\frac{|R'|}{B.x}\rceil \times \lceil\frac{|S'|}{B.y}\rceil$, where the product of the two thread block dimensions $B.x$ and $B.y$ needs to be at most 1024.

4.2. *Naive Implementation*

A naive implementation is as follows (see also Algorithm 1). First we copy the input relations to the device memory. As a preprocessing step on the host, unnecessary fields are filtered out before the copy. We also allocate memory for a single *sum* variable to hold the result of the aggregation. Every thread in which the theta condition is satisfied needs to increment the *sum* value by the corresponding $S.x$ value. Since threads run in parallel, in principle, multiple threads may try to update the *sum* variable simultaneously. Consequently, a simple arithmetic operation leads to information loss. To deal with such issues, CUDA provides a set of atomic functions. An atomic function guarantees that the corresponding operation will be performed by each thread without interference from other threads. In our implementation we use `atomicAdd`.

The drawback of using atomic functions lies in the suspension of parallel execution as threads are scheduled and executed in a sequential manner. This happens because no other thread is allowed to access the variable's address until the operation is completed by the thread that is currently manipulating the variable. Apparently, the performance degradation depends on the number of theta conditions satisfied and the warp execution scheduling. In the worst case, the number of instructions forced to be executed sequentially is the maximum number of active threads supported by the GPU.

Algorithm 1 Work per thread in the naive implementation

```
1: load $R$ tuple
2: load $S$ tuple
3: if join condition is met then
4:     update sum in the global memory using atomicAdd
5: end if
```

4.3. *Improvements*

In this subsection, we analyze our main improvements to the naive implementation. The main motivation behind them has been to exploit on-chip memories, and increase data reuse and what is called *achieved occupancy* according to the NVIDIA terminology. Occupancy is defined as the ratio of active warps per SM to the maximum allowed active warps per SM. The theoretical occupancy is calculated prior to execution by the launch parameters and the register count per thread. The achieved occupancy is the coverage ratio of the theoretical occupancy. As mentioned before, GPU

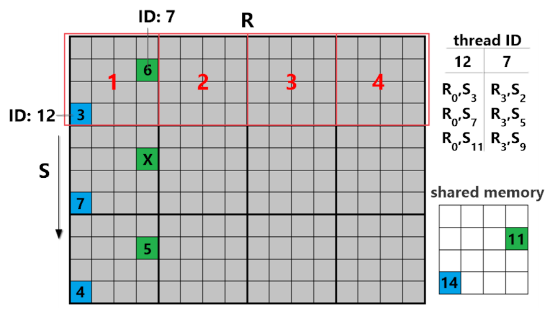

Figure 5. Illustration of theta-join processing according to the shared memory approach. Each thread reads a single record from the larger relation R and iterates over several records from S

hides its latency by massively executing threads. Occupancy is a metric that indicates the extent to which we keep the GPU busy. Higher occupancy does not guarantee better performance. However its maximization is a good heuristic approach.

4.3.1. Employing Partial Sums. The first improvement concerns the traffic reduction observed to access a single memory address by multiple threads. Our goal is to divide the sum calculation into smaller parts, i.e. we employ partial sums. This requires a post-processing step to produce the final sum.

More specifically, a partial sum is calculated for each thread block of the grid. The `atomicAdd` function is still used, but unlike the naive implementation, the kernel outputs a number of partial results. Hence, a secondary procedure is necessary to produce the final result. We can either pass the partial sums as input to a second kernel, which performs a parallel reduction [11], or copy the results back to host memory and calculate the final result with a simple iterative process on the CPU. The first choice is preferable since the intermediate results are already stored in the device memory and it takes negligible time.‖

4.3.2. Exploiting the Shared Memory. The second type of improvements concerns the usage of the shared memory. This memory allows threads to access data loaded from global memory by other threads within the same block, thus enabling data reuse.

Initially, we want to minimize the number of global memory transactions. If we launch a 2D grid of 2D thread blocks as before, and each block writes its partial sum in shared memory, then we can avoid the use of an atomic function to a large extent. An outline of this approach is as follows. Every thread in which the theta condition is satisfied loads the corresponding $S.x$ value to the shared memory and then, after a synchronization between the block's threads to avoid race conditions and ensure correct results, each block stores its partial sum in global memory. This approach improves the simple extension of the naive implementation as it avoids atomic function use, but incurs as many global memory write transactions as the number of thread blocks, which can still be very large. Further, it does not essentially exploit data reuse.

An alternative and more efficient approach, which addresses the issues mentioned above, is to assign more work to each thread. In other words, we allocate more pair comparisons to a single thread, and as such, we significantly depart from the initial naive approach. By launching less threads and assigning more work to each thread, we exploit data reuse through shared memory and further minimize global memory write transactions. Specifically, we launch a 1D grid of size equal to $\lceil\frac{|R|}{B.x}\rceil$, where R corresponds to the largest input relation.

In Figure 5, we provide a visual representation of how our shared memory approach manipulates the JM. In the figure, we assume that the block dimensions are $B.x = B.y = 4$. The size of R is

‖In order to perform parallel reduction efficiently, we decide to use a 32 × 32 block size. More details can be found at https://devblogs.nvidia.com/parallelforall/faster-parallel-reductions-kepler/

$|R| = 16$, which yields a 4×1 grid. In each thread block, $B.x$ records from R are read once. The red boxes in the figure show the comparisons for which each of the 16 threads in each of the 4 blocks is responsible at the initial stage. In general, a thread block is responsible for all S rows rather than only the first $B.y$ ones. When the initial comparisons finish, the red boxes slide by $B.y$, and this is repeated until all S rows are processed.

The figure shows the example executions of the threads with ids 7 and 12 in the first block (the first id is 0). The colored JM cells depict the cells examined by these two threads and the shown number is the $S.x$ value when the join condition is met. On the upper right part of the figure, all the combinations examined by these two threads are depicted; in the example, it is assumed that the pair of $R[3]$ and $S[5]$ (denotes as R_3S_5) does not satisfy the condition and the corresponding cell in the JM is marked by 'X'.

The choice of using the size of the largest relation for the grid size is important. In this way, we execute more threads and thus indirectly help the device to hide its latency. Also, each thread makes fewer comparisons, which also leads to fewer non-coalesced accesses to the global memory.

In addition, as shown in the lower right part of Figure 5, the shared memory holds the partial results for each thread in a single block. So, when all possible combinations have completed their evaluation, every block has a set of partial sums in its shared memory. To produce the corresponding partial sum for the complete block, we use the parallel reduction technique mentioned previously. After this stage, we store the intermediate result in global memory. Since we launch a 1D grid with fewer blocks, the number of global memory write transactions is decreased significantly compared to the previous approach, due to the fact that data reuse takes place.

An additional optimization concerns the use of the fastest memory on the device, namely the registers. We can store the participating fields of the R relation in the registers (i.e., $R.a$ in the example query), as they are reused during the execution of each thread. Hence, we further decrease the number of global memory read transactions. In general, registers must be used with caution because of their limited size. If their capacity is exceeded, register spilling occurs, i.e., register contents are spilled to local memory, which is off-chip. The following algorithm presents the work allocated to a single thread, which becomes responsible for a single R tuple.

Algorithm 2 Work per thread when exploiting the shared memory

```
1: load $R.a$ value to a register
2: load $S$ tuple with id equal to the thread $B.y$ coordinate
3: repeat
4:     if join condition is met then
5:         update corresponding shared memory cell
6:     end if
7:     load $S$ tuple with id larger by the $B.y$ size of the block
8: until no more $S$ tuples
```

4.3.3. Data Layout. Besides the improvements regarding exploiting unused memory, we have to investigate the data layout used. In our problem, a relatively high performance penalty is due to global memory accesses, while the computations are less expensive. According to the AoS data layout, the values of the same field in different records are not stored in consecutive memory addresses. To optimize memory access, consecutive threads should access consecutive memory addresses. This limitation is addressed by adopting the SoA data layout, which bears similarities to the column-oriented storage model in databases.

We analyze the AoS data layout a little further taking into consideration the hardware specifications and the way data flows through the memory hierarchy. Our tuple size is 12 bytes. We know that GTX 970 uses a memory bus width of 256 bits or 32 bytes. Given that the size of a L2 cache line is also 32 bytes, each memory access can be supported by L2 in full. Overall, in a L2 cache line, two tuples and the first two fields of a third tuple fit. If the missing field from the third tuple is requested, there is a cache miss. Thus a new memory access must be made. To avoid

Size	Naive	Partial Sums	Shared Memory			CPU		
			AoS	AoS+padding	SoA	6 threads	12 threads	PostgreSQL
500×50	0.02	0.021	0.012	0.012	0.011	**less than 0.001**		3
500×250	0.071	0.052	0.016	0.015	0.015	**less than 0.001**		25
500×450	0.121	0.087	0.018	0.017	0.018	**less than 0.001**		22
$5K \times 500$	1.271	1.056	0.086	0.084	**0.081(18.1X)**	1.467	1.847	260
$5K \times 2.5K$	6.303	5.1	0.293	**0.274(38.2X)**	0.277	12.804	10.472	1140
$5K \times 4.5K$	11.348	9.196	0.501	**0.465(46.6X)**	0.474	32.853	21.662	1950
$50K \times 5K$	108.815	94.018	4.797	**4.468(43.9X)**	4.504	301.681	196.268	20480
$50K \times 25K$	533.027	451.813	23.118	**21.318(49.3X)**	21.702	1804.519	1051.98	102360
$50K \times 45K$	957.757	809.986	41.433	**38.177(49.4X)**	38.838	2998.599	1884.701	183340
$500K \times 50K$	10610.993	9015	393.217	**363.63(57.6X)**	368.983	33445.87	20938.164	too high
$500K \times 250K$	53038.207	45101.3	2395.655	2563.121	**2340.031(45.9X)**	162775.844	107377.409	too high
$500K \times 450K$	95456.57	81126.835	4310.646	4576.681	**4341.435(43.8X)**	290216.809	190328.897	too high

Table II. Execution times for the query in Listing 3 with medium selectivity (in ms). The winning approach is in bold and the speedup compared to the best performing CPU time is inside the parentheses.

	Low selectivity (0.1)		Medium selectivity (0.5)		High selectivity (0.9)	
Size	PS	SM-SoA	PS	SM-SoA	PS	SM-SoA
$50K \times 5K$	25.715	4.477	94.018	4.504	161.001	4.54
$50K \times 25K$	113.606	21.524	451.813	21.702	792.342	21.819
$50K \times 45K$	199.718	38.582	809.986	38.838	1423.813	39.092
$500K \times 50K$	2284.643	366.457	9015	368.983	15828.143	373.08
$500K \times 250K$	11430.76	2098.067	45101.3	2340.031	79131.898	2693.466
$500K \times 450K$	20566.394	4019.814	81126.835	4341.435	142412.156	5042.282

Table III. Testing with different selectivities (PS: Partial Sums, SM-SoA: Shared Memory(SoA))

Algorithm 3 Stand-alone CPU implementation using p threads

```
1: split $R$ in $p$ partitions
2: Work of each thread:
3: for all tuples in the corresponding $R$ partition do
4:     for all tuples in $S$ do
5:         evaluate join condition
6:     end for
7: end for
```

this issue, data alignment is necessary. To this end, we add 4 more bytes to our tuple as a dummy padding field as shown in Listing 4. As a result, the new tuple size is 16 bytes and exactly two tuples fit in a single cache line at the expense of some extra preprocessing to add the new field. By contrast, the use of the SoA data layout does not require any preprocessing.

```
typedef struct {
  int id;
  int a;
  int x;
  char padding[4];
} Tuple;
```

Listing 4: Tuple representation with padding (AoS)

4.4. *Evaluation*

For the evaluation of our implementations, we use several sizes for the input relations with different selectivities. Selectivity is formally defined as $sel = \frac{|R \bowtie_\theta S|}{|R||S|}$. The selectivities considered are (i) low with $sel = 0.1$, (ii) medium with $sel = 0.5$ and (iii) high with $sel = 0.9$.

In Table II, we present the kernel execution times for datasets with medium selectivity. Each measurement is the average of ten executions. The first two columns refer to the kernels that implement the approach, where we launch a 2D grid of 2D blocks. The next three columns refer to the kernels that implement our approach, where we use shared memory and we launch a 1D grid of 2D blocks. In all the cases, the blocks are of size 32×32 in order to maximize the achieved occupancy. The rightmost set of three columns provides the host time regarding (i) a stand-alone C

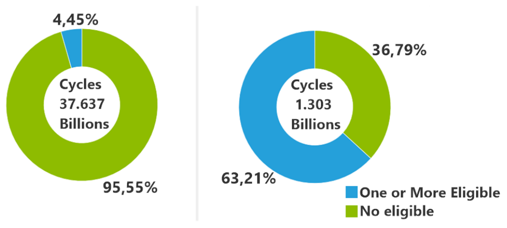

Figure 6. Warp issue efficiency: naive (left) vs. SM-SoA (right)

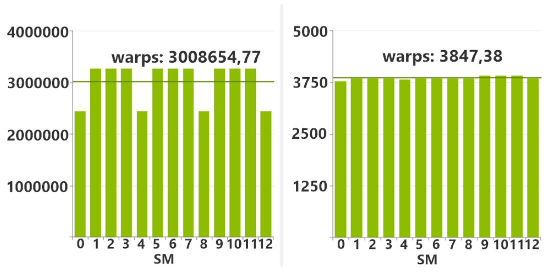

Figure 7. Warps launched: naive (left) vs.SM-SoA (right)

implementation using an Intel i7 5820k 3.3GHz CPU with, 16 GB DDR4 RAM at 2400MHz and 6 and 12 threads, and (ii) the corresponding times in PostgreSQL in line with the evaluation rationale in [2], which also uses the performance of PostgreSQL as a baseline. The specific type of CPU has 6 cores supporting 12 threads due to hyper-threading; as such, the penultimate column corresponds to full CPU utilization. The stand-alone implementation is free of any overheads that are involved through running a complete query processing engine; this advantage comes at the expense of not benefiting from all the sophisticated caching and indexing mechanisms that PostreSQL provides. It adheres to the rationale in Algorithm 3, where all data is in main memory, R is split into as many partitions as the number of threads, and each thread iterates over the complete S relation for each R tuple.[**]

In Table II, the winning approaches are in bold, and their speedup compared to the best performing CPU time is inside the parentheses. The execution times in the table show that exploiting the shared memory yields improvements of an order of magnitude (up to more than 20 times faster execution) over the naive implementations. By contrast, simply mitigating the impact of atomic operations yields improvements of only 10-20% lower times. Also, the comparison against the CPU times proves that allocating theta-join processing on GPUs is beneficial yielding lower times by two orders of magnitude when compared to our C stand-alone implementation unless the datasets are very small. The improvement is even more significant when compared against a state-of-the-art database management system, such as PostgreSQL, due to the inherent overheads from running a complete execution engine. Finally, there is no clear winner between the SoA and AoS+padding data layouts; however, the former is more efficient for larger datasets, and for the datasets that is not optimal, it is very close to the optimal.

To investigate the reason for the differences in the GPU execution times, we profile the naive implementation and the shared one using the SoA data layout for the dataset of size $50k \times 25k$ with medium selectivity. Figures 8-9 show lower-level metrics that explain why the shared memory approach is more efficient. On the left side of each figure, we demonstrate the profiling of the naive implementation and on the right that of the shared one. Both implementations have 100% theoretical occupancy. However, the naive one achieves only 67.8%, while the shared-memory based one achieves 99.76%. The main reason for this difference lies in the warp execution efficiency as shown in Figure 6. Despite the fact that we launch $800\times$ less warps in the shared-memory approach, as shown in Figure 7, a much higher proportion of them are eligible for execution per cycle and better

[**]The complete code is at `https://github.com/chribell/cuda-theta-join/blob/master/src/aggregate/cpu_aggregate.c`

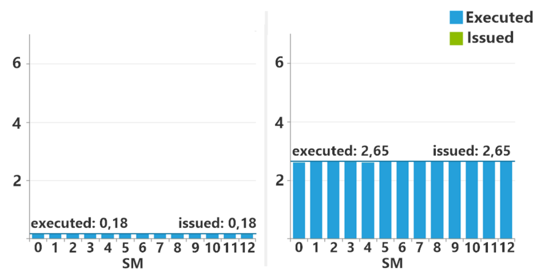

Figure 8. Instructions per Clock: naive (left) vs. SM-SoA (right)

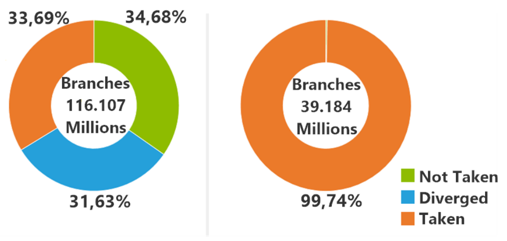

Figure 9. Branch condition: naive (left) vs. SM-SoA (right)

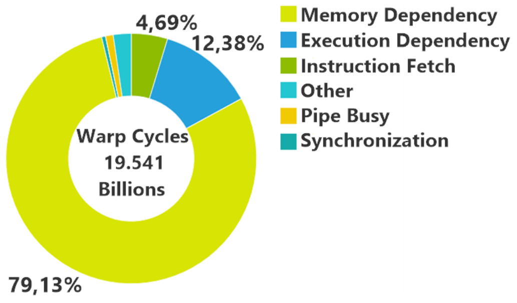

Figure 10. Memory dependency

load balance between the SMs is achieved. This leads to more instructions per clock, as shown in Figure 8. Branching is another issue that is tackled more efficiently in the shared-memory approach, where most threads have the same execution flow thus reducing divergence, as shown in Figure 9.

Another aspect of the evaluation deals with the approaches' efficiency on datasets of the same size but with different selectivities. As shown in Table III, the kernels which implement our naive approach are more sensitive to the selectivity value. To the contrary, the execution times of the optimized kernels that leverage the shared memory are less affected by changes in the join selectivity.

Two final notes are as follows. Our problem is memory-bounded as shown in Figure 10, and this supports our approach to emphasize on efficient memory management. Through the shared memory approach, we better exploit the L1 and L2 caches, thus minimizing accessed to the global device memory. However, the global read/write efficiency reported by profiling remains relatively low. Employing other types of memory, such as texture memory, is not expected to yield tangible benefits. In general, further investigation is required to increase global memory efficiency.

Second, our improvements have been mostly driven by profiling analysis. We also heavily used the debug flag (-G) of the `nvcc` compiler during the development of the techniques. However, all the final experiments are without the debug flag. Note that the running times with the debug option are largely different than the ones presented in the tables and their analysis may lead to inaccurate conclusions if not done with care due to the associated overheads.

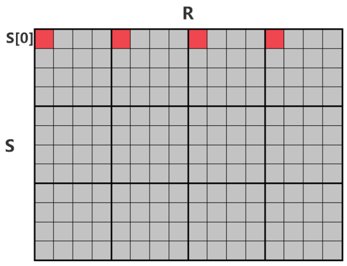

Figure 11. Illustration of same reads: tuple S[0] is read $|R|/B.x$ times in the worst case

5. OUTPUT-BOUND THETA-JOINS

Having established a general view about efficient thread indexing and JM partitioning, we proceed to the investigation of the output-bound query. The added complexity concerns the need to efficiently store output data. The query used includes the same theta condition. Output tuples consist of three fields, namely $R.a$, $S.a$, $S.x$, as shown below.

```
SELECT R.a, S.a, S.x
FROM R, S
WHERE R.a > S.a;
```

Listing 5: General theta-join query

5.1. *Naive Implementation*

As for the input-bound query, we first provide a naive implementation, where we allocate as many threads as the JM cells. Remember that, in each round, the host feeds the maximum amount of data so that their cartesian product can fit into the device memory. To store the result tuples, we initially use the AoS data layout as shown in Listing 6.

```
typedef struct {
  int Ra;
  int Sa;
  int Sx;
} Result;
```

Listing 6: Result tuple representation (AoS)

To reduce running times, it is important to manage to store output tuples in consecutive memory locations. For this to be achievable, and due to the fact that thread divergence existence is the norm, the usage of a global index counter is required. Every thread, in which the theta condition is satisfied, must store its corresponding output tuple to the address pointed by the index counter and increment the counter by one. To ensure correctness, the increment operation must be done using an atomic function. However, this leads to sequential thread execution, as already discussed for the atomic sum.

Note that in general, every input tuple is read multiple times, even from threads in the same block. The same S tuple is processed by all threads corresponding to the same row in the JM matrix. Similarly, R tuples are also reused. As such, storing the input records on the chip is beneficial. However, employing the shared memory to temporarily store input tuples plays a minor (if not negligible role) in this case. This is because we employ 32×32 thread blocks, which are decomposed in 32 warps. In each warp, the same S tuple is processed by multiple threads but all of them belong to the same warp. Consequently, each S tuple is actually read once from global memory and then cached in L1 to be reused by all threads in the same block and in L2, which is shared among SMs. In the worst case, an S tuple is not in L2 when subsequent blocks request it, and it is read $\frac{|R|}{B.x}$ times. An example of such a case is shown in Figure 11, where the first tuple from S can be read up to four times from the global memory.

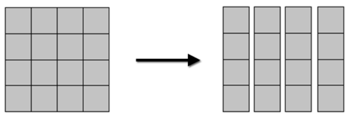

Figure 12. Assigning more work in each thread so that it processes a whole JM column (right) instead of a JM cell as in Figure 5 (left)

Algorithm 4 Work per thread when exploiting the shared memory for the output-bound case

```
1: load $R.a$ value to a register
2: load first $S$ tuple in shared memory
3: repeat
4:     if join condition is met then
5:         output result (main option: use shared memory, see Section 5.3)
6:     end if
7:     load next $S$ tuple in shared memory
8: until no more $S$ tuples (overall $p$ tuples are processed)
```

5.2. *A more efficient approach*

As shown in the input-bound query, the best implementation was when we launched a 1D grid. In the naive approach, the total number of threads is $T = \lceil \frac{|R'|}{B.x} \rceil \times \lceil \frac{|S'|}{B.y} \rceil \times B$, and each thread evaluates a single theta condition. In our optimized approach, we launch a 1D grid, and the number of threads is $T = \lceil \frac{|R'|}{B.x} \rceil \times B$. The main difference from the technique in the previous section, which aimed at the computation of a partial aggregation, is that the thread block size is one dimensional as well. That is, $B = B.x$, which implies that each thread becomes responsible for evaluating $|S'|$ theta conditions (see Figure 12) and as such, the code in Algorithm 2 is modified to iterate over all S tuples, as shown in Algorithm 4.

At first sight, the execution of a much lower number of threads comes at the expense of decreased achieved occupancy. However, as shown in [13] and other works, it is possible to achieve better performance at lower occupancy. A common misconception about CUDA programs is that, through only the execution of hundreds of thousands threads, the GPU can manage to hide its compute or memory access latency. However, exploitation of on-chip memories, efficient memory access and data reuse and sharing among threads are equally important.

We begin with the exploitation of on-chip memories. Since each thread processes exactly one tuple from the R relation, it can hold it in its registers throughout the thread lifetime.

Next, we exploit the shared memory for holding the contents from S. In the aggregation query, the thread block dimensions were fixed to 32×32 and, as such, the S records processed by the same threads in a block were cached in L1 and L2. By rendering the block size 1D, it is very likely to have different warps requiring the same S tuples, and there is no guarantee that these are cached. So, we explicitly transfer the S records in shared memory.

Due to the limited shared memory size, we partially iterate over S in chunks of size p; p is selected so that the chunks can fit in the shared memory. Further, since each thread retrieves at most one S tuple from global memory and stores it to the shared memory, the chunk size is directly linked with the block size B, and more formally $p \leq B$. Finally, to avoid further divergence as will shown below, the chunk size p needs to be a divisor of $|S|'$, i.e., $|S|' \ mod \ p = 0$.

To summarize, we launch a 1D grid of $\lceil \frac{|R'|}{B} \rceil$ blocks. As shown in Figure 13, the red rectangles represent the blocks launched while the blue rectangle the portion of S relation read. Upon the kernel launch, we move $R.a$ values to the registers of the corresponding threads. Then, an iterative process begins (the blue rectangle in the figure). In each iteration, we read p tuples from S, the participating fields of which, i.e., $S.a$ and $S.x$, are stored in the shared memory, so that they are reused by all threads in the same block. Next, through a second nested iterative process, each thread evaluates a

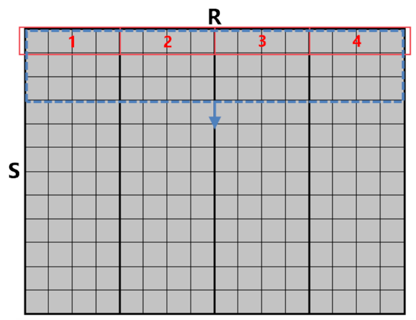

Figure 13. Our output-bound approach to process the JM on a GPU

sequence of p theta conditions and stores the corresponding output tuples. We discuss output storage later. The last operation is to increment the counter, which indicates the position for the next chunk. These three operations are repeated until the last portion of S is accessed. In between iterations, we synchronize threads at the block level with a view to ensuring correctness (and indirectly maintain the degree of parallelism). This technique can be also regarded as equivalent to applying a sliding window on the JM.

A few additional implementation details are as follows. If B is a divisor of $|R'|$ and multiple of 32, which is the warp size, we ensure that only $|R'|$ threads will be executed. This is the ideal case. For example, in a different case where $B = 32$ and $|R'| = 15000$, we execute 14976 threads in 468 warps, where all threads are active. To cover the remaining R tuples, we schedule an additional warp, where only 24 threads are active. Another issue that requires some care is not to allocate the reading of an S tuple to an inactive thread. Although this is required only in the warp processing the rightmost portion of the R relation, it involves the usage of an extra condition, which increases the requirements of registers per thread. This in turn may lead to further restrictions on the amount of threads that can launched in a single thread block, as discussed in Section 2, and calls for careful design of the kernel to avoid register spilling.

5.3. Output writing issues

In our implementations, we assume an array of size $|R'| \times |S'|$ as result of the join. With the use of atomic functions, we ensure that if X tuples are in the output, these will be stored in the first X addresses of the result array. This storage method is necessary because, when the execution is finished, it is more likely that we use less memory than the amount allocated, and if we do not write in consecutive memory addresses, then either a post-processing phase is required or we transfer more data to the host memory through the slow PCI-E link. But if results are stored consecutively, then there is no need for post-processing.

Another option is to use the shared memory for temporarily storing the results before writing them into the global memory. For this to be applicable, the corresponding amount of shared memory must be free, and this depends on the size of output tuples. In the naive approach, where we launch a 2D grid of 2D blocks, the space needed is defined by the block dimensions. For example, if we have a 32×32 block and output tuple size equal to 12 bytes, then the space needed would be $1024 \times 12 = 12288$ bytes. On the other hand, in the second approach, B and p define the space needed per iteration. If $B = 256$ and $p = 25$ then we would need $256 \times 25 \times 12 = 76800$ bytes of shared memory, which is already partially occupied by p tuples from S. In general, through the usage of shared memory, we alleviate the storage problem and manage to accelerate the whole execution.

However, the issue of writing tuples consecutively from the shared memory of SMs to the global memory still exists. To solve it, we can either (i) use a shared atomic function or (ii) resort to other techniques similar to prefix sum. Since a block's execution is independent from the execution of other blocks, grid level synchronization is required to ensure that no output tuple overwriting occurs. In our experiments, option (i) behaved better. Although the second option sounds reasonable, in our experiments, it has not yielded any benefits. More specifically, for option (ii), we investigated a 2-pass approach, where we first compute where each thread should write and then we proceeded in

Size	Medium selectivity (sel=0.5)			High selectivity (sel=0.9)		
	Execution	Transfer	Output	Execution	Transfer	Output
$50K \times 5K$	30.238	645.909	1.4GB	47.487	935.876	2.5GB
$30K \times 30K$	139.914	2042.585	5.03GB	199.796	3653.337	9.05GB
$50K \times 25K$	343.761	2736.725	6.98GB	441.718	5835.067	12.57GB

Table IV. Naive approach times (in ms)

Size	Medium selectivity (sel=0.5)			High selectivity (sel=0.9)		
	1 thread	6 threads	12 threads	1 thread	6 threads	12 threads
$50K \times 5K$	3151	611.81	490.096	3515	698.174	1009.288
$30K \times 30K$	12847	2304.92	1876.202	15048	3177.28	3315.225
$50K \times 25K$	17718	3335.919	2811.114	20922	4659.629	5386.257

Table V. CPU times (in ms)

the actual result writing. We employed the `exclusiveScan` operation from CUB[††] to access the results of the 1st pass and avoided any `atomicAdd` operations. However, this resulted in higher execution times by several factors up to an order of magnitude. The performance degradation is partially attributed to the fact that the pointers for each thread are stored in the global memory and the amotized cost of accessing the global memory is higher than an atomic operation.

Finally, we investigated compression, i.e., instead of forcing writes in consecutive memory addresses, to write results in predefined places. Since not each tuple pair is expected to produce a result, then we can employ compression to reduce the amount transferred to the host. We implemented such an approach with the help of the corresponding functionality of the `thrust` library, but, as shown in the evaluation section, no improvements were yielded. An additional implication of such an approach is that we need to split the device memory into two parts: one to hold the initial results and another for the compressed ones.

In summary, we consider efficient output writing to remain an open issue. Promising directions include a closer collaboration between CPUs and GPUs in theta-join processing. An early idea is the result of each tuple pair evaluation on the GPU side to be a bit indicating whether the theta predicate evaluates to true or false, and the actual result tuple construction to take place on the CPU. In such an approach, the challenge is not to transfer the bottleneck to the CPU. However, in the current work, we focused on techniques where the complete processing takes place on the GPU; co-operation between the two types of processors in producing the final results is left for future work. Finally, advances, such as NVlink, may alleviate any current bottlenecks regarding transfer times, and render GPU-side join processing even more attractive.

5.4. *Evaluation*

For the evaluation of the output bound query, we execute each kernel ten times and calculate the average execution time. We examine two kernels, one for each approach in Sections 5.1 and 5.2, respectively, with datasets of medium and high selectivity. In Table IV, we present the times for our naive approach. We distinguish between execution and transfer time. Table V presents the CPU times using the stand-alone implementation (also presenting single-thread executions). In Table VI, we present only the execution times for our shared memory-based approach for a range of p and B values. The transfer times through the PCI-E bus are in the same range as in Table IV and are omitted. The shared memory-based approach presented employs the AoS data layout with tuple padding. We choose the AoS data layout to store output tuples, so that threads write in consecutive memory addresses. Our experiments with the other data layout modifications that we have presented do not exhibit significant differences in execution times, as also shown in the evaluation of the input-bound query and are omitted.

[††] `https://nvlabs.github.io/cub/`

	$50K \times 5K$			$30K \times 30K$			$50K \times 25K$		
p	$B = 256$	$B = 512$	$B = 1024$	$B = 256$	$B = 512$	$B = 1024$	$B = 256$	$B = 512$	$B = 1024$
10	**20.791(23.6X)**	21.207	22.733	86.955	97.871	122.501	131.778	148.627	186.026
25	21.095	21.404	22.628	84.423	97.431	122.061	**126.014(22.3X)**	147.693	183.195
100	21.597	21.264	22.803	84.532	96.398	120.676	128.073	145.531	183.680
250	21.298	21.559	22.619	**84.289(22.3X)**	95.627	121.526	128.072	144.350	183.740
500	-	21.388	22.567	-	97.499	121.899	-	144.407	183.055
625	-	-	22.488	-	-	121.931	-	-	185.052

Table VI. Shared memory-based (AoS plus padding) execution times with medium selectivity (in ms). The best performing configuration is in bold and the speedup compared to the best performing CPU time without considering transfer times is inside the parentheses.

	Medium selectivity (sel=0.5)			High selectivity (sel=0.9)		
Size	Execution	Compression	Transfer	Execution	Compression	Transfer
$50K \times 5K$	30.321	76.313	585.583	45.733	96.401	1013.376
$30K \times 30K$	216.083	274.934	2117.466	271.012	347.681	3754.746
$50K \times 25K$	422.425	376.478	2979.389	496.492	479.365	5302.361

Table VII. Times when enabling compression (in ms)

As previously, to maximize the achieved occupancy, in the naive approach we launch a 2D grid of 2D blocks of size 32×32. In the shared approach, we launch a 1D grid of 1D blocks of size $B = B.x$ and choose a p to iterate the S relation. To avoid further divergence while iterating the last portion, p should be a divisor of $|S'|$. However, in real case scenarios and depending on the JM partitioning, this may not be feasible. Our implementations are not tailored to a specific data input size with a cost of a small amount of divergence, which is negligible thanks to automated `nvcc` compiler optimizations.

From the results in Tables IV, V and VI, three main conclusions can be drawn. First, the improvements on the naive approach are significant but of lower magnitude than in the previous query. For example, the more efficient shared-memory approach runs up to 2.7 times faster than the naive implementation and 23.6 times faster than a 12-thread CPU implementation, whereas for the input-bound query higher speedups have been observed. This is due to two factors: (i) there are too frequent global memory transactions, and (ii) the use of atomic functions to store results consecutively has a negative impact on performance. Second, the performance is higher for relatively low values of p and B, e.g., 25 and 256, respectively. Third, the transfer time seems to dominate. However, as reported in [10], in the close future, the transfer times are expected to drop by an order of magnitude due to the NVLink technology, which implies that they will become similar to, if not lower than the execution times. In our experiments, the transfer overhead is outweighed by the benefits from the parallel execution on the GPU for certain configurations, such as 6-threads on a CPU and medium selectivity.

Next, we measure the time required only for writing the output on the GPU without performing any meaningful processing. For example, for the $50K \times 5K$ times dataset with $sel = 0.5$, writing 125 million tuples takes 20.04 ms, which means that our shared-memory approach manages to drop the execution times close to their minimum possible.

Finally, we have investigated additional approaches to writing the output, as mentioned previously. We repeat in Table IV in order to apply compression, and the results are shown in Table VII. By comparing the two tables, we can observe that compression performs worse, i.e., the execution time is higher, there is a compression overhead and no benefits from reduced transfer time. This is due to the fact that the compression-based approach suffers from increased conflicts due to concurrent accesses to main memory and does not benefit from shared memory. On the other hand, the naive approach already produces the results in a compacted form, in the sense that results are written in a coalesced manner.

6. RELATED WORK

As already mentioned in the introduction, to the best of our knowledge, there have been no other works on theta-joins on GPUs. Existing proposals for relational query operators focus on sorts, selects, aggregates and hash- and sort-based equi-joins or the efficient handling of data transfer between the host and the device.

Many research works study the efficient memory coalescing and splitting of the workload into chunks. In [4], a set of primitives for join processing implemented on a GPU are presented. Their behavior on modern GPUs is also discussed in [14]. The work in [4] contains a chunk-based implementation of nested loops that, in principle, can process theta-joins; however, no implementation details, e.g., shared memory usage, are provided in order their proposal to be comparable against ours. In other words, we share the same high-level approach and our contribution can be deemed as a detailed investigation of the corresponding implementation issues.

In [6], the relations are split into chunks with the use of a tailored data structure that allows for efficient data transfer and usage on both sides. In [5], data rearrangement through clone creation is proposed to reduce overall memory contention and race conditions between executed threads. We address the same problems, but tailored to the specific and challenging case of theta-joins. Using the shared memory in equi-joins on GPUs is also discussed in [15].

Also, due to the limited size of the GPU memory, there is a need to exploit certain technologies, which allow the device to use host memory. The study presented in [16] investigates the use of the Unified Virtual Addressing (UVA). UVA provides a single virtual memory address space for all memory in the system, thus enabling pointers to be accessed from GPU no matter where in the system they reside.

Another problem reported in the literature relates to thread divergence. In contrast to a typical CPU, a GPU devotes a larger amount of transistors to data processing compared to data caching and flow control. This means that any kind of divergence during parallel execution incurs a performance penalty. Thread divergence is common in join queries. In [17], there is a study of conjunctive selection queries, but the proposed technique cannot generalize to theta-joins that we target. Additional works on GPU queries include topics, such as concurrent queries [18] and portable development [19]. These issues are orthogonal to our research. Also, specific forms of joins other than equi-joins, e.g., continuous intersection joins of moving objects [20], have been investigated.

With a view to exploiting both CPUs and GPUs, the study in [21] presents a relational query processing strategy, which first calculates quickly and approximately the query results based on compressed data within the device memory and produces the final results on the host side. Another example of closer collaboration between CPU and GPU regarding equi-joins has appeared in [22]. Investigation of techniques where the CPU and GPU cooperate more closely to process a theta-join is an interesting direction for future work, in line with the recent trend in GPGPU [10].

We close this section with a brief discussion of techniques for parallel theta-joins on CPUs. Regarding theta-joins in MapReduce, apart from the work in [8], [23] provides an adaptive solution, whereas [2] explores techniques for more efficient predicate evaluation when there are two predicates. [24] deals with the issue of preprocessing the JM, when selectivity information is known a-priori. [25] improves on [8] for a specific form of JMs, termed as monotonic. [26, 27, 28] investigate the case where multiple relations are joined in a single step. None of these proposals contains techniques that can be transferred to our setting. Finally, theta-joins are also related to similarity joins, e.g. [29]. However, the main effort in similarity joins in parallel settings is to prune non-relevant pairs as soon as possible, which corresponds to eliminating JM candidate cells rather than devising techniques to efficiently process all JM cells as in generic theta-joins.

7. CONCLUSIONS

This work deals with theta-joins on GPUs, an issue that has not been explored in depth to date. It considers two cases. In the first one, the theta-join is followed by an aggregate, while in the second, the complete output, which may be very large, needs to be stored. Our work emphasizes on issues such as exploitation of on-chip memories, data reuse, reduced accesses to the slow global memory, high achieved occupancy, and efficient data layout. It thoroughly discusses the implementation issues involved and proposes specific optimizations, which yield performance improvements up to an order of magnitude over naive implementations in the first case, and up to 2.7 times in the second case.

A main lesson learned through this activity is that paying attention to the details is of high importance and several factors are significant in an efficient implementation; these factors do not merely relate to a high number of threads to hide latencies, but include all the types of issues mentioned above. Moreover, the relative importance of each factor changes from one case study to another, which calls for specialized solutions for each case. Finally, the most challenging remaining issues is how to efficiently write huge amounts of data to the device memory as a result of data processing and transfer them to the host, and how CPUs and GPUs can co-operate more closely.

REFERENCES

1. Armbrust M, Xin RS, Lian C, Huai Y, Liu D, Bradley JK, Meng X, Kaftan T, Franklin MJ, Ghodsi A, *et al.*. Spark SQL: relational data processing in spark. *ACM SIGMOD International Conference on Management of Data*, 2015; 1383–1394.
2. Khayyat Z, Lucia W, Singh M, Ouzzani M, Papotti P, Quiané-Ruiz J, Tang N, Kalnis P. Lightning fast and space efficient inequality joins. *PVLDB* 2015; **8**(13):2074–2085.
3. Keckler SW, Dally WJ, Khailany B, Garland M, Glasco D. Gpus and the future of parallel computing. *IEEE Micro* 2011; **31**(5):7–17.
4. He B, Yang K, Fang R, Lu M, Govindaraju N, Luo Q, Sander P. Relational joins on graphics processors. *ACM SIGMOD international conference on Management of data*, ACM, 2008; 511–524.
5. Sitaridi EA, Ross KA. Ameliorating memory contention of olap operators on gpu processors. *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, ACM, 2012; 39–47.
6. Bakkum P, Chakradhar S. Efficient data management for gpu databases. *High Performance Computing on Graphics Processing Units* 2012; .
7. Zhou G, Chen H. Parallel cube computation on modern cpus and gpus. *The Journal of Supercomputing* 2012; **61**(3):394–417.
8. Okcan A, Riedewald M. Processing theta-joins using mapreduce. *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, ACM, 2011; 949–960.
9. NVIDIA. Cuda toolkit documentation. http://docs.nvidia.com/cuda.
10. Mittal S, Vetter JS. A survey of cpu-gpu heterogeneous computing techniques. *ACM Computing Surveys (CSUR)* 2015; **47**(4):69.
11. Kirk DB, Hwu WW. *Programming Massively Parallel Processors - A Hands-on Approach, 2nd Ed.* Morgan Kaufmann, 2013.
12. Lin J, Dyer C. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool Publishers, 2010.
13. Volkov V. Better performance at lower occupancy. *Proceedings of the GPU Technology Conference, GTC*, vol. 10, 2010; 16.
14. Rui R, Li H, Tu Y. Join algorithms on gpus: A revisit after seven years. *2015 IEEE International Conference on Big Data, Big Data*, 2015; 2541–2550.
15. Pietron M, Russek P, Wiatr K. Accelerating select where and select join queries on a GPU. *Computer Science (AGH)* 2013; **14**(2):243–252.
16. Kaldewey T, Lohman G, Mueller R, Volk P. Gpu join processing revisited. *Eighth Int. Workshop on Data Management on New Hardware*, ACM, 2012; 55–62.
17. Sitaridi EA, Ross KA. Optimizing select conditions on gpus. *Ninth Int. Workshop on Data Management on New Hardware*, ACM, 2013; 4.
18. Wang K, Zhang K, Yuan Y, Ma S, Lee R, Ding X, Zhang X. Concurrent analytical query processing with gpus. *PVLDB* 2014; **7**(11):1011–1022.
19. Zhang S, He J, He B, Lu M. Omnidb: Towards portable and efficient query processing on parallel CPU/GPU architectures. *PVLDB* 2013; **6**(12):1374–1377.
20. Ward PGD, He Z, Zhang R, Qi J. Real-time continuous intersection joins over large sets of moving objects using graphic processing units. *VLDB J.* 2014; **23**(6):965–985.
21. Pirk H, Manegold S, Kersten M. Waste not efficient co-processing of relational data. *30th Int. Conf. on Data Engineering (ICDE)*, IEEE, 2014; 508–519.
22. He J, Lu M, He B. Revisiting co-processing for hash joins on the coupled CPU-GPU architecture. *PVLDB* 2013; **6**(10):889–900.
23. Elseidy M, Elguindy A, Vitorovic A, Koch C. Scalable and adaptive online joins. *PVLDB* 2014; **7**(6):441–452.
24. Koumarelas I, Naskos A, Gounaris A. Binary theta-joins using mapreduce: Efficiency analysis and improvements. *Int. Workshop on Algorithms for MapReduce and Beyond (BMR) (in conjunction with EDBT/ICDT'2014)*, 2014.
25. Vitorovic A, Elseidy M, Koch C. Load balancing and skew resilience for parallel joins. *Proc. of ICDE*, 2016.
26. Beame P, Koutris P, Suciu D. Skew in parallel query processing. *PODS*, 2014; 212–223.
27. Zhang X, Chen L, Wang M. Efficient multi-way theta-join processing using mapreduce. *PVLDB* 2012; **5**(11):1184–1195.

28. Chu S, Balazinska M, Suciu D. From theory to practice: Efficient join query evaluation in a parallel database system. *ACM SIGMOD International Conference on Management of Data*, 2015; 63–78.
29. Sarma AD, He Y, Chaudhuri S. Clusterjoin: A similarity joins framework using map-reduce. *PVLDB* 2014; **7**(12):1059–1070.