Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs

Johns Paul
National University of Singapore
idsjp@nus.edu.sg

Bingsheng He
National University of Singapore
hebs@nus.edu.sg

Shengliang Lu
National University of Singapore
lusl@nus.edu.sg

Chiew Tong Lau
Nanyang Technological University, Singapore
asctlau@ntu.edu.sg

ABSTRACT

In recent years, we have witnessed significant efforts to improve the performance of Online Analytical Processing (OLAP) on graphics processing units (GPUs). Most existing studies have focused on improving memory efficiency since memory stalls can play an essential role in query processing performance on GPUs. Motivated by the recent rise of just-in-time (JIT) compilation in query processing, we investigate whether and how we can further improve query processing performance on GPU. Specifically, we study the execution of state-of-the-art JIT compile-based query processing systems. We find that thanks to advanced techniques such as database compression and JIT compilation, memory stalls are *no longer* the most significant bottleneck. Instead, current JIT compile-based query processing encounters *severe under-utilization of GPU hardware* due to divergent execution and degraded parallelism arising from resource contention. To address these issues, we propose a JIT compile-based query engine named *Pyper* to improve GPU utilization during query execution. Specifically, Pyper has two new operators, *Shuffle* and *Segment*, for query plan transformation, which can be plugged into a physical query plan in order to reduce divergent execution and resolve resource contention, respectively. To determine the insertion points for these two operators, we present an analytical model that helps insert Shuffle and Segment operators into a query plan in a cost-based manner. Our experiments show that 1) the analytical analysis of divergent execution and resource contention helps to improve the accuracy of the cost model, 2) Pyper significantly outperforms other GPU query engines on TPC-H and SSB queries.

PVLDB Reference Format:
Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. Improving Execution Efficiency of Just-in-time Compilation based Query Processing on GPUs. PVLDB, 14(2): 202 - 214, 2021.
doi:10.14778/3425879.3425890

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/Xtra-Computing/Pyper.

Proceedings of the VLDB Endowment, Vol. 14, No. 2 ISSN 2150-8097.
doi:10.14778/3425879.3425890

1 INTRODUCTION

In recent years, we have witnessed a significant effort in improving the performance of Online Analytical Processing (OLAP) systems on graphics processing units (GPUs) [17, 21, 22, 32, 35, 38]. More recently, JIT compilation-based systems (hereafter referred to as *compiled systems*) [5–7, 11] have evolved as the state-of-the-art query execution systems on GPUs. JIT compilation has the major advantage to reduce unnecessary materialization of intermediate data at operator/kernel boundaries within query pipelines and to reduce the instruction count within GPU kernels. On the other hand, many efforts have focused on improving the memory efficiency since memory stalls play an important role in query processing performance on GPUs [13, 15–18, 36]. Given all these optimizations, it is time to revisit the query execution on GPUs and identify the opportunities to improve query processing performance on GPUs.

Taking a closer look at existing compiled systems [5–7, 11], we find that they work by first breaking down the query plan associated with each query into pipelines and then generating a single kernel for each pipeline of relational operators. We refer to this compilation strategy as a *monolithic* approach, since it generates a monolithic kernel for each pipeline. While this monolithic approach eliminates the amount of intermediate data materialization, we have identified two major performance issues.

First, the generated kernels in the monolithic approach often have a large number of nested branch instructions per kernel. Hence, the thread warps executing the kernels generated by the monolithic approach can have a high degree of divergence, resulting in poor utilization of a large number of GPU cores.

Second, the high resource requirements of the generated kernels lead to low parallelism on GPU. Overall, the monolithic approach generates kernels containing multiple relational operators, which require even more registers and shared memory resources. That means the monolithic approach exaggerates the problem of resource contention on the GPUs since threads can only be scheduled for execution on GPUs if there are enough free hardware resources available. Therefore, the monolithic approach also leads to degraded parallelism on GPUs and cannot fully utilize the large number of parallel cores on the GPU.

To demonstrate the impact of the two execution efficiency issues, we present the execution time break down of the code generated from the monolithic approach for TPC-H [1] (queries without subquery) and SSB [3] in Figure 1. We divide the execution time into three components: 1) estimated overhead of divergence, 2) estimated overhead of resource contention, and 3) others. The overhead

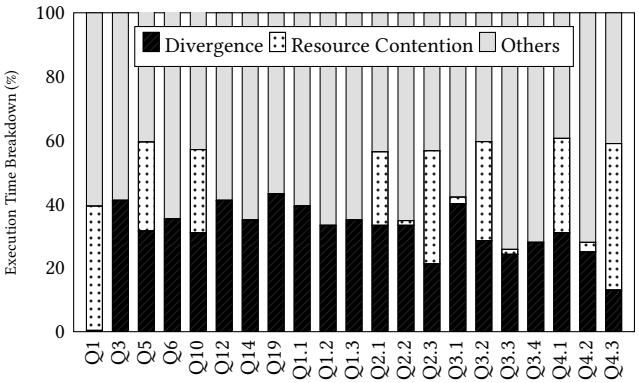


Figure 1: Execution time breakdown of TPC-H and SSB queries demonstrating the impact of divergent execution and resource contention among threads.

of divergence and resource contention were estimated based on the achieved execution efficiency of the warps and GPU hardware occupancy rate, both of which can be obtained using profiling tools from GPU vendors. The results of the breakdown clearly demonstrate that divergent execution and resource contention among the threads have a significant impact on the overall performance of both TPC-H and SSB queries. Overall these inefficiencies can contribute to more than 50% of the total execution time. Therefore, divergence and resource contention have now become the most significant bottleneck for query processing on GPUs.

In this paper, we propose *Pyper*, a JIT compilation-based query processing system to resolve GPUs' divergence and resource contention issues. Pyper introduces two new operators for query plan transformation, *Shuffle* and *Segment*, to improve GPU hardware utilization during query execution. Specifically, Shuffle is applied to a pipeline to reduce the execution divergence. Segment is used to split a pipeline that has overwhelming resource usage into several pipelines with lower resource usage but much higher thread parallelism. The benefits of the two operators come with overheads. Thus, we develop a cost model to guide the optimal use of Shuffle and Segment operators. The challenge is that we need to estimate the impact of divergence and resource contention, which have been overlooked in the previous studies [14, 15].

The major contributions of this paper are as follows.

- We study the efficiency of current JIT compilation-based query processing approaches and identify their severe underutilization of GPU hardware due to divergent execution and degraded parallelism arising from resource contention.
- We propose Pyper to address the execution efficiency issues of existing JIT compilation-based systems, by introducing two new operators (Shuffle and Segment) and an analytical model to determine the usage of the two operators in a cost-efficient manner.
- We conduct in-depth experiments demonstrating the benefit of using the Shuffle and Segment operators as well as the accuracy of our analytical model. Our experiments show that Pyper is able to improve the performance of TPC-H and SSB queries on average by 1.60x and 1.52x, compared to Hawk [5]. Pyper further achieves 5.01x and 2.55x performance improvement over Omnisci [2] for TPC-H and SSB queries respectively.

The rest of this paper is organized as follows. In Section 2.1, we present the GPU hardware's background and the related work on JIT query compilation. We then present the architecture design of Pyper in Section 3. We present the details of the analytical model and query optimizer in Sections 4 and 5, respectively. We present the experiments in Section 6 and conclude in Section 7.

2 BACKGROUND AND RELATED WORK

2.1 GPU Hardware & CUDA

A single GPU consists of multiple *streaming multiprocessors* (SMs), each of which consists of multiple CUDA cores. All CUDA cores in an SM share resources like the registers and shared memory. Due to this sharing, the workload scheduler in GPU only schedules new threads for execution on an SM, if there are enough registers and shared memory resources available. GPUs also have an L2 cache and a global memory that are shared among all the SMs. Each thread block is given exclusive access to shared memory, i.e., data in shared memory cannot be shared across different thread blocks.

Warp-at-a-time execution model: In the CUDA programming model, a program executed by the GPU is known as a kernel. A kernel is executed as a grid of thread blocks, which can further be divided into *warps* (groups of 32 threads). Each thread block is assigned to a single SM, and the CUDA cores inside each SM executes the threads in a SIMD fashion, at the granularity of a single warp. Hence, during any given execution cycle, an entire warp of threads can only execute a single instruction. That means, during each instruction issue cycle, only threads that can execute the same instruction will be activated, and the resources allocated to the other threads would be wasted. Thus, branch divergence can be a serious performance issue on GPU executions.

Resource constraints per thread: For the efficiency of GPU executions, registers, and shared memory are the key resources that constrain the number of parallel thread executions. Warps can only be scheduled for execution on GPUs if there are enough free hardware resources available.

2.2 JIT Query Compilation

JIT compilation-based systems (hereafter referred to as *compiled systems*) [5–7, 11] have been developed to further reduce unnecessary materialization of intermediate data at operator/kernel boundaries within query pipelines and to reduce the instruction count within GPU kernels. In the following, we review the related work of JIT-based query engines on GPUs (more related work can be found at our technical report [29]).

LLVM based database systems like Voodoo [31], HetExchange [6], HAPE [7], OmniSci (formerly known as MapD) and OpenCL based HorseQC [11] and Hawk [5] were proposed to generate optimized executable code for heterogeneous systems containing both CPUs and GPUs. Both Voodoo [31] and Hawk [5] attempt to generate code to execute on a variety of parallel architectures. HetExchange [6] developed abstraction for device-oblivious operators and exploited the heterogeneous parallelism of modern servers with multiple CPUs/GPUs. HorseQC [11] extends the operator-at-a-time approach by

fusing multiple operators together for better bandwidth utilization. An industrial startup OmniSci [27] developed its code generation following Hyper [25, 26].

More recently, DogQC [12] proposed techniques to address divergence in query execution on GPUs. We highlight our technical contributions over DogQC. First, compared with DogQC, Pyper has a cost model to help determine the optimal insertions of the Shuffle and Segment operators. This is important because the unnecessary insertion of these operators can lead to severe performance degradation. Second, Pyper adopts more efficient implementations for the Shuffle and Segment operators. The Shuffle implementation in DogQC is only capable of re-distributing the workload within a single warp even though the re-distribution is done via shared memory. This is insufficient because a single straggler warp within a thread block can prevent the entire block from releasing the resources allocated to it, leading to unnecessary inefficiencies. By re-distributing the workload over an entire thread block, Pyper can better address this inefficiency. Further, the use of buffered writes also helps improve the efficiency of the Segment operator in Pyper when compared to DogQC.

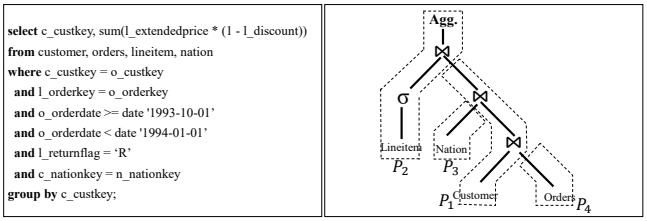
There have been previous studies beyond query processing, on workload divergence and resource contention. Some examples include financial modelling [37], sparse matrix computations [4, 9] and graph processing [19, 33]. In contrast, this paper focuses on query processing with JIT code generation where the execution pipeline is not specific to certain applications, rather being generated at run-time according to input queries. Further, our work develops a cost model to guide the optimizations of query pipelines.

2.3 Motivations

While those approaches have reduced the overhead of materializing intermediate data in the pipeline execution, such monolithic designs do not fully consider the GPU architecture features like 1) their warp-at-a-time execution model and 2) their severely resource-constrained hardware design. Hence, the code generated by these implementations often encounters severe under-utilization of GPU hardware due to divergent execution and resource contention among threads, as discussed in Section 1.

Example. In Figure 2, we use TPC-H query 10 with minor modifications (for simplicity of presentation) to demonstrate the working of the existing compiled systems. As shown in the figure, existing systems segment the query plan into different pipelines at predefined blocking/non-pipelineable operators (e.g., prefix sum, hash build, and aggregate) and then generate a single kernel for each one of these pipelines. Note, we will be using a pipeline similar to P_1 as a running example throughout the rest of this study to demonstrate the inefficiencies of existing systems and the benefits of the techniques proposed in this study.

The divergence is mainly caused by the multiple nested loops in the kernel, e.g., P_1 in Figure 2. Not every tuple will go through all the loops, and thus GPU threads executing P_1 may have different code paths. A group of 32 GPU threads is a warp, and the GPU cores sharing the same instruction issue unit can only execute the threads within the same warp during each instruction issue cycle (in a SIMD fashion). When the threads within a warp diverge or follow different code paths (e.g., due to branching), their execution



(a) SQL (b) Pipelines

$P_1(\ldots)$:
for every tuple c in Customer
for every match of c.custkey in hash table of Orders.custkey
for every match of c.nationkey in hash table of Nation.nationkey
materialize t in hash table of Orders.orderkey

$P_2(\ldots)$:
for every tuple l in Lineitem
if l.returnflag == 'R'
for every match of l.orderkey in hash table Orders.orderkey
group by custkey and materialize the output

$P_3(\ldots)$:
for every tuple n in Nation
materialize t in hash table of Nation.nationkey

$P_4(\ldots)$:
for every tuple o in Orders
materialize t in hash table of Orders.custkey

(c) Pseudocode in JIT generated kernels

Figure 2: Modified TPC-H Q10 query, execution plan and pseudo code of the generated procedures in the monolithic approach

gets serialized. In the worst case, the execution time can be the sum of the execution times of all executed code paths.

As for high resource consumption of the monolithic approach, the kernel generated corresponding to P_1 in Figure 2 needs to store hash tables or hash buckets headers corresponding to three different join operators in its shared memory and hence require significantly more shared memory resource when compared to fine-grained kernels containing a single join operator.

For simplicity purposes, we will use the pipeline P_1 as the example used for the rest of the paper to demonstrate our techniques.

3 DESIGN OF PYPER

To solve the performance issues of divergent execution and resource contention in the monolithic approach, we propose Pyper, a JIT compilation-based query processing system designed for GPUs. In the following, we first give a system overview of Pyper and then present the detailed design and implementation of Shuffle and Segment operators.

3.1 Overview

Figure 3 provides an overview of the architecture design of Pyper. As shown in the figure, Pyper is developed by augmenting a physical query plan generated from existing compiled systems [7, 20]. Specifically, we have the following featured components.

- Two plan transformation operators, Shuffle (Section 3.2) and Segment (Section 3.3). They can be inserted into a physical query plan, to help reduce the divergent execution and resource contention among threads. We designed these operators such that they can be inserted into any query pipeline, without modifying the implementations of the other physical, relational operators. Moreover, we develop efficient designs in order to reduce their runtime overheads.
- An analytical model (Section 4), which is capable of estimating the execution cost of a pipeline of relation operators. Our model has non-trivial extensions over the existing GPU analytical model [30] by developing a cost analysis of divergent executions and resource contention.
- An augmentation optimizer (Section 5), which uses the analytical model and a branch and bound based optimization

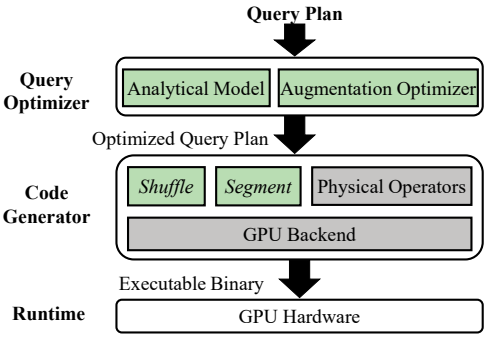


Figure 3: System Architecture of Pyper

algorithm to identify the optimal insertion points of Shuffle and Segment operators into the physical query plan.

Putting it together, Pyper works as follows. In this study, we use the pipeline physical query plan generated by the previous study named GPL [30]. Pyper further optimizes each of the pipelines by considering the insertions of Shuffle and Segment. This is guided by the augmentation optimizer, which takes the estimations of the analytical model into consideration. The outcome is an augmented (or optimized) query plan. Then, the code generator generates the code corresponding to each pipeline in the augmented query plan. Finally, the generated code is compiled into hardware-dependent machine code using GPU backend (e.g., *libnvvm* or *nvcc* from NVIDIA).

3.2 Shuffle Operator

The Shuffle operator is an operator that can be inserted between two relational operators (producer and consumer). It buffers the tuples from divergent warps from the producer operator and consolidates the input tuples to the consumer operator in the pipeline so that the execution of the consumer operator has the reduced divergence among the threads within the same warp. Thus, this paper focuses on the warp divergence brought by the divergent execution path of multiple operators in a pipeline. The divergence within an operator (mainly branch divergence) is not the main focus of this paper. Specifically, we focus on the divergence when different threads are assigned different amounts of work. For example, the number of tuples selected for processing in the next operator after a filter predicate can vary across threads, resulting in divergence among the threads of executing the operators after the filter. The same applies to a series of hash probe operations where different threads could end up generating different numbers of output tuples depending on the join selectivity.

How does Shuffle reduce divergent executions? To demonstrate the benefit of using the Shuffle operator, we present an example execution of the warps executing the pipeline P_1 (shown on the left of Figure 4) from the previous example query. We assume a warp size of 4 and the uniform selectivity of 50% per operator for this example for simplicity. We now look at the execution of the pipeline with and without inserting Shuffle operators.

Without Shuffle: Due to 50% selectivity of $\bowtie_{a=b}$ and $\bowtie_{c=d}$, only 50% and 25% threads in each warp will be allocated work in hash probes of $\bowtie_{c=d}$ and $\bowtie_{f=g}$ respectively. Such workload imbalance leads to the poor overall utilization of the GPU hardware since only a small subset of threads in each warp are allocated work.

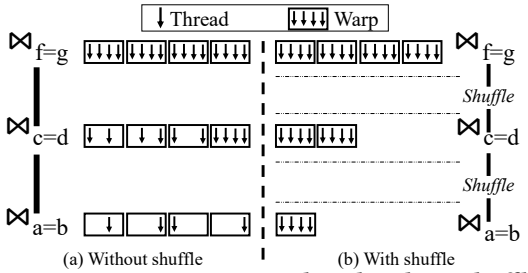


Figure 4: Warp executions with and without Shuffle.

With Shuffle: Inserting a Shuffle operator between each pair of relational operators in P_1 forces the workload to be concentrated over fewer warps executing at 100% efficiency, as shown in Figure 4. This helps make more efficient use of the GPU hardware since warps without any work can be quickly swapped out, allowing other warps to execute.

Lightweight Shuffle on the GPU. Now, re-balancing the workload in a lightweight manner is a non-trivial task. Consider a producer-consumer operator pair in the pipeline (denoting O_p and O_c as the producer and consumer operators, respectively). The Shuffle operator is inserted into this pipeline so that the output from O_p (which is sparsely distributed across the threads) is consolidated into a smaller number of warps, thus reducing the divergent execution in O_c. A potential implementation of the Shuffle operator is to simply gather the outputs from O_p. While this implementation helps eliminate divergent execution, it has severe efficiency issues. This is because the implementation is blocking and thus causes unnecessary intermediate result materialization of all the tuples between O_p and O_c.

To overcome these inefficiencies, we develop a lightweight approach that takes advantage of GPU hardware features to implement Shuffle. We have the following design rationales. First, our implementation completely avoids blocking across thread blocks, thus avoiding costly synchronizations using GPU global memory. Second, warps that do not encounter high levels of divergence are prevented from participating in the re-assignment of the workload, thus minimizing the synchronization within a thread block and global memory accesses. Third, we make use of GPU specific low overhead intrinsics to implement the Shuffle operator. Listing 1 illustrates the implementation of the Shuffle operator, which consists of the following two stages: *voting* and *shuffling*.

Voting Stage: During the voting stage, each thread votes using the __ballot instruction to reach a consensus on whether there is workload divergence within the warp (Lines 6 – 9). The GPU voting intrinsics are very efficient, each executing within a single cycle. __ballot(vote) in Listing 1 generates a 32-bit integer where the i^{th} bit is set to 1 if the i^{th} thread has a valid input tuple assigned to it (i.e., $vote = 1$). The __popc instruction is another intrinsic to compute the number of bits in this value that has been set to 1. In summary, __popc(__ballot(vote)) computes the number of threads that been assigned tuples for processing in each warp ($actThreadsW$). Finally, Lines 10 and 11 keeps track of the number of threads that are assigned with tuples within an entire block using a variable in shared memory ($actThreadsB$).

Shuffling Stage: During the shuffling stage (Lines 13 – 26), the workload is redistributed based on the consensus reached in the

voting stage. We can perform workload shuffling at different levels, which come with different benefits and overheads. Particularly, the shuffling across the entire kernel (named grid-level) achieves the most divergence reduction but has a high overhead. In contrast, the shuffling within a thread block (named block-level) achieves reasonable divergence reduction among the warps within a thread block but has relatively low overhead. We use grid-level shuffling (Lines 14 – 17) or block-level shuffling (Lines 19 – 26), based on the level of divergence encountered by the threads.

- Grid-level shuffling requires moving tuples across multiple thread blocks and is costly due to the use of GPU global memory. The threshold for Grid-level shuffling (*GridT*) depends on the GPU hardware as well the downstream operator. This is because the Grid-level shuffling is beneficial only when the additional cost of global memory use is more than compensated by the reduced divergence in the downstream operators. Hence, during system initialization, Pyper profiles the GPU hardware and determines the benefit of Grid-level shuffling for the supported relational operators. Finally, to avoid deadlocks and blocking execution across thread blocks, each thread block executing the *GridShuffle* function checks a global buffer (*global_shuffle_mem*) and either 1) picks up tuples from the buffer and redistributes them across its threads or 2) writes the tuples held by its threads into the buffer if there are not insufficient tuples already in the buffer for redistribution. The last set of thread blocks that will be scheduled for execution only picks up tuples from the buffer to ensure correctness.
- Block-level shuffling is more frequently used by Pyper and helps address more moderate levels of divergence (which is often encountered in common analytical workloads). Based on our experiments, we consider a warp as requiring block-level shuffling if a warp (32 threads) achieves less than 75% occupancy on the GPU hardware (i.e., $BlockT = 24$). Overall, if more than 2 warps are requiring block-level shuffling within a thread block (i.e., $numWarp > 2$ as shown in Line 22), then the Shuffle operator invokes the *BlockShuffle*. BlockShuffle is implemented as an efficient filter process described in the previous study [10]. Note, we use variables in shared memory and registers to invoke and perform the block-level shuffling, making it highly efficient. The output of BlockShuffle is the number of valid input tuples for this thread block. After BlockShuffle, all valid tuples are stored consecutively, thus avoiding any divergence in the next/downstream operator.

Even though we have a very efficient implementation for Shuffle, this operation still leads to an increase in the kernel resource requirements. Hence, the insertion of a Shuffle will be used only when it is beneficial for the performance. To further, minimize the overhead, the code corresponding to grid-level shuffling (Lines 10 – 17) is only generated as part of the Shuffle operator when extremely low levels of execution efficiencies are encountered. We leave these decisions to the cost model and augmentation optimizer.

Listing 1: Simplified code for Shuffle operator

```
1  Input tuple; // input generated by the previous operator
2  __shared__ Input shuffle_mem[ ]; // buffer for shuffling
3  __shared__ int numWarp = 0;
4  __shared__ int actThreadsB = 0;
5
6  /* Voting Stage */
7  int vote = 0
8  if (tuple != nil) vote = 1
9  int actThreadsW = __popc(__ballot(vote)));
10 if (tidInWarp == 0) //tidInWarp: threadID in warp.
11  atomicAdd(actThreadsB, actThreadsW)
12
13 /* Shuffling Stage */
14 if (actThreadsB <= GridT) {
15  //tidInBlock: threadID in block.
16  shuffle_mem[tidInBlock] = tuple;
17  tuple = GridShuffle(shuffle_mem, global_shuffle_mem)
18 }
19 else if (actThreadsW <= BlockT && tidInWarp==0) {
20  atomicAdd(numWarp, 1);
21 }
22 if (numWarp > 2) {
23  shuffle_mem[tidInBlock] = tuple;
24  numValid = BlockShuffle(shuffle_mem);
25  if (tidInBlock < numValid)
26   tuple = shuffle_mem[tidInBlock];
27 }
```

3.3 Segment Operator

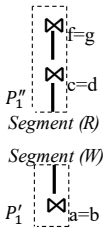
The Segment operator is to split a pipeline into two smaller pipelines. It can be inserted into a long resource-constrained pipeline, split into two smaller pipelines with lower resource requirements. The monolithic approach aggressively fuses relational operators in the same pipeline and compiles it into a single kernel on the GPU. Such a kernel usually has high resource requirements, resulting in low occupancy rate of GPU and degraded parallelism, which makes it difficult for modern GPU hardware to hide memory latency. We find that the Segment operator is desirable to split monolithic kernels and improve the GPU's utilization.

In this paper, we consider two major constrained resources on the GPU: registers and shared memory. Modern GPUs have limited sizes of shared memory and register files that need to be shared across a large number of threads. For example, the NVIDIA V100 GPU used in our experiments is configured with 64KB shared memory and 256 KB registers per SM, which needs to be shared among 64 active warps to ensure complete occupancy of each SM.

Consider P_1 in Figure 2, where the pipeline consists of two hash probe and one hash build operator. Assuming each operator needs to keep the hash table in the GPU shared memory, the thread block executing the kernel corresponding to the pipeline would require 33 KB of shared memory per thread block, as shown in Figure 5a. If this query plan runs NVIDIA V100 GPU, the pipeline can only achieve a maximum theoretical occupancy rate of 50%.

How does Segment improve the parallelism? To understand how the use of the Segment operator helps address the above inefficiency, we present Figure 5b, in which a Segment operator is inserted into the pipeline P_1 between two of its hash probe operators. The insertion of the Segment operator at this point splits P_1 into two smaller pipeline fragments: P_1' containing hash probe of $\bowtie_{a=b}$ and P_1'' containing hash probes of $\bowtie_{c=d}$ and $\bowtie_{f=g}$. Such a split means that the resulting pipeline fragments P_1' and P_1'' only require 16 KB and 17 KB of shared memory respectively, which enable P_1' and P_1'' to achieve a theoretical occupancy rate of 100% individually. Hence, by inserting the Segment operator into the pipeline, we can generate kernels that achieve significantly better occupancy rate and in turn, better utilization of GPU hardware. In the following, we present a lightweight design for Segment.

Lightweight design for Segment. Overall, the Segment operator is implemented as a *blocking* operator with a write (*Segment(W)*)



Pipeline	Operator	Shared Memory (KB)
P_1''	$\bowtie_{f=g}$	12
	$\bowtie_{c=d}$	5
	Segment (R)	0
Shared Memory		17

P_1'	*Segment (W)*	0
	$\bowtie_{a=b}$	16
Shared Memory		16

(b) After segmentation

Figure 5: Impact of Segment operators on query plan and resource requirements

and a read (*Segment(R)*) stage, as shown in the example of Figure 5b. When generating code for the pipeline fragments of a pipeline split using the Segment operator, the code for the write stage is generated in the kernel corresponding to the upstream pipeline fragment, and the code for the read stage is generated in the kernel corresponding to downstream pipeline fragment.

The write stage of the Segment operator includes the instructions to collect all the tuples generated by the downstream operator and writes this data into the GPU global memory; while The read stage includes the instructions to reads the data written by the write stage from the global memory and passes this data to the upstream operator in the query plan. We optimize write with *write buffer* [28], which writes the output in batches rather than per tuple. For read, we take advantage of coalesced memory accesses to improve bandwidth utilization.

The usage of the Segment operator has runtime overhead due to the materialization of the data into GPU global memory between the write and read stages of the Segment operator. Hence, the Segment operator should only be used when it can improve the overall performance. This will be determined by the cost model in the next section.

The idea of the segment operator in Pyper is similar to previous studies [8, 23, 34], with the following major differences. First, the segment operator in Pyper is inserted to minimize resource contention in the generated code, whereas previous studies have different purposes, such as vectorization opportunities [8, 23]. Second, our segment implementation is designed specifically for GPUs and is more efficient for GPU hardware due to its use of local buffering and coalesced writes. Third, comparison to previous studies, this study has an efficient cost model to determine the optimal insertion point (with the awareness of the resource contention of the generated code) of the Segment operator.

4 ANALYTICAL MODEL

There have been several studies on building cost models to estimate the execution time of a query plan on the GPU [14, 15]. However, those studies fail to effectively account for the impact of divergent execution and resource contention. As we demonstrated in Section 1, the overhead of divergent execution and resource contention can contribute to over 50% of the total execution time in the monolithic approach. Thus, this paper extends the previous model to handle divergent execution and resource contention in JIT compilation-based query plans.

Table 1: Notations in the cost model

Name	Description	Sources
P	A pipeline in the query plan	Query Plan
$\|P\|$	The number of operators in the P	
O_i	i^{th} operator in P	
$\widehat{P}$	The execution cost of P	Model Output
$\widehat{O}_i$	The execution cost of O_i	
γ_i	Number of scheduled threads that have been allocated input in O_i	
$\widehat{C}_i(\gamma_i)$	Computation cost per tuple of O_i	
$\widehat{M}_i(\gamma_i)$	Memory access cost per tuple of O_i	
T_i	Avg. active threads per warp of O_i	Program & Selectivity Analysis
$\|E_i\|$	Number of *warp execution* of O_i	
T_{ij}	Number of threads that are allocated work in the j^{th} *warp execution* of O_i	
W_i	Number of warps of O_i that can execute simultaneously on the GPU	
W_i^{sm}	Number of warps of O_i that can execute simultaneously per SM	
$\|B_i^w\|$	Warps per thread block in O_i	
S_i	Shared mem. per thread block in O_i	
R_i	Registers per thread block in O_i	
$\|M_i^s\|$	Number of shared mem. accesses in O_i	
$\|M_i^g\|$	Number of global mem. accesses in O_i	
H_i	L2 cache hit rate in O_i	
$\widehat{M}_{ij}^s(\gamma_i)$	Cost of j^{th} shared mem. access in O_i	Profiling
$\widehat{M}_{ij}^{gdr}(\gamma_i)$	Cost of j^{th} global memory access in O_i	
$\widehat{M}_{ij}^{l2}(\gamma_i)$	Cost of j^{th} L2 cache access in O_i	
W_{max}	Max. simultaneous warps per SM	Platform Input
B_{max}	Max. simultaneous blocks per SM	
R	Registers available per SM	
S	Shared mem. available per SM	
$\#SM$	Number of *SMs* in the GPU	

First, instead of building an analytical model from scratch, we use the previous model [30] in estimating the cost of a pipeline execution, and then extend this model to estimate the impact of divergent execution and resource contention. To account for divergent execution, we need to model the number of active threads per warp in each operator, and then estimate the divergent execution for the entire pipeline. The major impact of resource contention is the degree of thread parallelism, which is a key factor for the performance in the execution unit and memory sub-system of the GPU.

Second, we treat the cost analysis for Shuffle and Segment operators the same as other relational operators. Hence, any query plan augmented by the insertions of Shuffle and Segment can be analyzed in uniformly.

4.1 Pipeline Execution Cost

The previous study GPL [30] proposed a model to compute the execution cost of a pipeline of relational operators (P). The model in GPL computes the cost of processing a single tuple by a pipeline of operators ($P = \{O_1, O_2, ..., O_{|P|-1}, O_{|P|}\}$) using Equation 1. Note, O_i can be Shuffle, Segment or relational operators. Here, $\widehat{C}_i$ and $\widehat{M}_i$ represent the computation and memory access cost of operator O_i for processing each tuple, respectively. λ_i represents the estimated number of input tuples for O_i from the query optimizer.

$$\widehat{P} = \sum_{i=1}^{|P|} (\widehat{C}_i + \widehat{M}_i) \times \lambda_i \tag{1}$$

GPL's model does not properly account for the divergent execution and resource contention in JIT compilation-based query plan. The total number of active threads issuing and executing instructions in parallel on the GPU (denoted as γ_i) can vary significantly based on the factors in divergence and resource usage. γ_i has a significant impact on query processing performance. Thus, we need to extend the estimation of $\widehat{C}_i$ and $\widehat{M}_i$ to account for the impact of γ_i. Specifically, we model the computation and memory access costs as functions of γ_i (denoted as $\widehat{C}_i(\gamma_i)$ and $\widehat{M}_i(\gamma_i)$, respectively). Thus, the estimated cost of P in our cost model is given in Equation 2.

$$\widehat{P} = \sum_{i=1}^{|P|} (\widehat{C}_i(\gamma_i) + \widehat{M_i}(\gamma_i)) \times \lambda_i \tag{2}$$

In the following, we detail how we estimate γ_i, $\widehat{C}_i(\gamma_i)$ and $\widehat{M}_i(\gamma_i)$.

4.2 Estimation of γ_i

Due to the warp-at-a-time execution model of GPUs, we estimate γ_i to be the number of active warps multiplied by the average number of active threads per warp. That is, γ_i can be computed using Equation 3, where T_i represents the average number of active threads per warp (in O_i) and W_i denotes the number of active warps of O_i that can execute simultaneously on the GPU.

$$\gamma_i = T_i \times W_i \tag{3}$$

Calculation of T_i. T_i is an effective measure of divergence among the threads executing O_i. Specifically, lower T_i represents a higher degree of divergence among the threads. T_i depends on selectivity of O_{i-1} in the pipeline. The idea is, if a tuple is eliminated by O_{i-1}, the thread in O_i and thereafter in the pipeline could have less work to handle. This effect has to account for all ancestor operators in the pipeline. Assuming the selectivity of O_i to be e_i. In this case, the ratio of the active threads per warp is $\prod_{j=1}^{i-1} e_j$. We estimate T_i as the number of threads in a warp multiplied by $\prod_{j=1}^{i-1} e_j$.

Calculation of W_i. Since GPU SM hardware design and resource distribution across the thread blocks of a kernel are uniform, the number of warps that can execute simultaneously on a given GPU hardware (W_i) can be computed as the product of W_i^{sm} and the number of SM in GPU hardware ($\#SM$), as shown in Equation 4.

$$W_i = \#SM \times W_i^{sm}. \tag{4}$$

For an operator O_i, the maximum number of warps per SM that can execute simultaneously on the GPU hardware (W_i^{sm}) depends on 1) the limit per SM set by the hardware and 2) the resource requirements of the kernel executing the operator.

- *Hardware Limit.* Due to the limited amount of resources available for managing thread contexts at the warp and thread block levels, NVIDIA CUDA sets hard limits on the number of warps and the number of thread blocks that can execute simultaneously on each SM of the GPU (denoted as W_{max} and B_{max}, respectively). Therefore, W_i^{sm} will be limited by Equation 5 where $|B_i^w|$ is the number of warps per thread block in O_i.

$$W_i^{sm} \leq min(W_{max}, B_{max} \times |B_i^w|) \tag{5}$$

- *Resource Limit.* Now, each thread block executing O_i requires certain amount of register and shared memory resources (denoted as R_i and S_i, respectively) and can only be scheduled for execution when sufficient free resources of all resource types are available in hardware. Hence, the number of thread blocks that can be scheduled for simultaneous execution on an SM is limited by the register (R) and shared memory resources (S) available at the hardware level in each SM. Hence, the value of W_i^{sm} is further limited by Equation 6.

$$W_i^{sm} \leq min(\lfloor \frac{R}{R_i} \rfloor, \lfloor \frac{S}{S_i} \rfloor) \times |B_i^w| \tag{6}$$

To summarize, the number of warps of O_i that can execute simultaneously on each SM (W_i^{sm}) is the largest integer satisfying Equations 5 and 6.

4.3 Estimation of Computation Cost ($\widehat{C}_i(\gamma_i)$)

Modern GPU hardware is designed to allow threads to issue and execute instructions in parallel. Hence, the computation cost of a relational operator ($\widehat{C}_i(\gamma_i)$) can be estimated as the execution cost of all the instructions that need to be executed in O_i (denoted as C_i) divided by the number of parallel active threads issuing instructions (γ_i), as shown by Equation 7. C_i can be computed by offline profiling of hardware and operator code as done in GPL [30]. Overall, Equation 7 clearly shows that the lower the workload divergence and resource contention among threads (i.e., higher γ_i), the lower the computation cost of the generated kernel is.

$$\widehat{C}_i(\gamma_i) = \frac{C_i}{\gamma_i} \tag{7}$$

4.4 Estimation of Memory Cost ($\widehat{M}_i(\gamma_i)$)

The cost of memory access on GPU can vary depending on the following factors: 1) the type of physical memory being accessed (shared or global memory), 2) the memory access pattern (sequential, random, stride) and 3) the number of threads issuing the memory access request in parallel. Taking into consideration the type of physical memory being accessed, memory access cost ($\widehat{M}_i(\gamma_i)$) per tuple can be computed as the sum of the memory access cost to the shared memory, L2 cache and global memory required to process a single input tuple (Equation 8). For an operator O_i, $|M_i^s|$ and $|M_i^g|$ represent the number of accesses to the shared memory and global memory (which could also be cached in the L2 cache), respectively.

To have a more accurate estimation, we consider the cost per memory instruction. This is possible because the instructions from each operator are known in JIT-based compilation system. Thus,

$\widehat{M}^s_{ij}(\gamma_i)$, $\widehat{M}^{l2}_{ij}(\gamma_i)$ and $\widehat{M}^{ddr}_{ij}(\gamma_i)$ represent the cost of shared memory, L2 cache and global memory accesses cost of the j^{th} memory access in O_i as a function of γ_i. H_i is the average L2 cache hit rate in O_i and is obtained following the same approach as GPL [30].

$$\widehat{M}_i(\gamma_i) = \sum_{j=1}^{|M^s_i|} \widehat{M}^s_{ij}(\gamma_i) + \sum_{j=1}^{|M^g_i|} (H_i \times \widehat{M}^{l2}_{ij}(\gamma_i) + (1-H_i) \times \widehat{M}^{ddr}_{ij}(\gamma_i)) \quad (8)$$

It is challenging and complicated to build analytical models for these metrics. We use a calibration-based approach for the calculation. For each memory instruction, each thread issues the memory access and those memory accesses form a certain access pattern (such as sequential, random and strided). Thus, we need to consider the factors that affect the cost of the memory access, including the size of the memory access, the access pattern, and the number of active threads. In JIT-based compilation systems, since the supported operators are known in advance, we perform an offline profiling during the model initialization to derive their functions with respect to the above-mentioned factors. Particularly, we perform profiling on the GPU memory hardware to measure the time per memory access for varying number of active threads on a given configuration of access pattern and memory access unit size. We study different configurations by considering all combinations of memory unit sizes (e.g., 2 bytes, 4 bytes and 8 bytes) and access patterns commonly used in databases (such as sequential, random and stride from 2 bytes, 4 bytes, ... to 32 bytes). Then, for each configuration, we then perform a regression analysis for the function which gives the memory cost by given the number of active threads as input. In our study, the entire calibration takes around 30 minutes when the system initializes (this calibration runs only once, and the calibration results can be reused afterwards for the same hardware). At runtime, we simply use these functions to quickly compute the cost of each memory access for each memory instruction.

As an example, Figure 6 shows the calibration data along with the fitted curve for sequential access to shared memory, L2 cache and the global memory of the V100 GPU (γ_i values from 40K to 160K). Their regression curves are also shown in the figure, with RMSE lower than 8%. For clarity of the figure, we scale up $\widehat{M}^s_i(\gamma_i)$ by a factor of 10. The performance trends of all the three types of memory (shared memory, cache and global memory) clearly show that the performance need to take γ_i into consideration. In experiments, the regression approach gives sufficient accuracy and contributes the accuracy of our cost model.

5 AUGMENTATION OPTIMIZER

With the analytical model in hand, we are able to estimate the cost of a query plan with arbitrary insertions of Shuffle and the Segment operators. However, we can potentially insert Shuffle and Segment operators after each operator into the pipeline. The solution space can be large for complex queries. Also, different insertions can have very different performance, as observed in our experiments. Thus, we develop a GPU aware optimizer in Pyper to efficiently find the augmented query plan with the lowest execution time.

Specifically, we make use of a Branch & Bound (B&B) based technique [24] to solve this problem. In the remainder of this section,

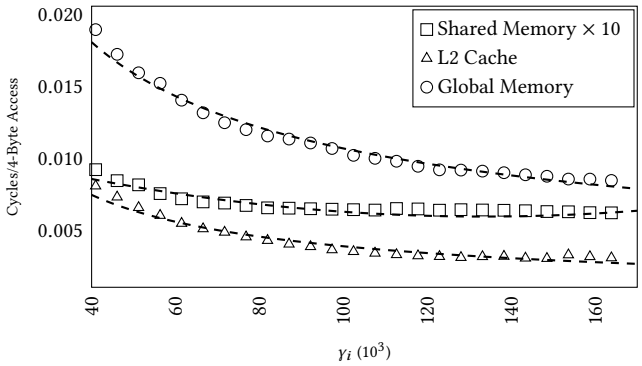


Figure 6: Variation of sequential memory access cost with γ_i for V100 GPU

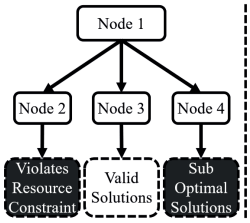
we detail how the B&B based technique is used to determine the optimal insertion point for the Shuffle and Segment operators.

Overview. The augmentation optimizer in Pyper applies the B&B based optimization algorithm to each pipeline in the query plan individually. For each pipeline, the algorithm starts with the original pipeline in the form of $P = \{O_1, ..., O_n\}$ as the root, where n is the number of operators in the pipeline, and O_i is the producer operator for O_{i+1} ($1 \leq i < n$). Then, the augmentation optimizer systematically enumerates a tree, based on a bounding function. Each node in the tree represents an augmented query plan for the input pipeline, which is generated through an insertion of: 1) a single Shuffle operator, 2) a single Segment operator or 3) a pair of Shuffle and Segment operators between any two relational operators in its parent node. When a pair of Shuffle and Segment operator is inserted, the Shuffle operator is always inserted before the Segment operator as the write stage of the Segment operator ensures that the data is written into consecutive memory locations (i.e., the warps in the next operators will not encounter any divergence). Thus, there are a total of 4^{n-1} possible nodes in the solution space.

Bounding Function. To limit the size of the solution space and to reduce the overhead of determining the optimal solution, the B&B based algorithm in Pyper makes use of a bounding function based on the cost of the kernels being generated by each node. Specifically, the augmentation optimizer keeps track of node with the current lowest execution cost. If any node generates a kernel with a single relational operator and the kernel has an execution cost greater than the current optimal node, then the node is marked as sub-optimal and its child nodes are not explored. This is because the overall execution cost of the pipeline represented by this node and its child nodes will be higher than the current optimal solution.

Pruning Criteria. We use the following pruning criteria or heuristics to reduce the number of nodes that needs to be explored.

- Due to the limited amount of resource available on the GPU, NVIDIA CUDA sets hard limits on the resources that can be utilized at the warp and thread block level (Equation 5). Hence, any node that requires the generation of a kernel that violates these resource constraints is marked as invalid and its child nodes are not explored. The reason is, if these resource constraints are violated, there is no choice but to segment the pipeline. Those solutions would be explored in another part of the tree, and thus we mark this node as invalid to avoid redundancy.



NODE 1	
Operators	$\{O_1, O_2, O_3\}$
Reg.	4.7 KB
Shared Mem.	45 KB
Cost	100 ms
Min Cost	100 ms

NODE 2	
Operators	$\{O_1, Shfl, O_2, O_3\}$
Reg.	5.2 KB
Shared Mem.	60 KB
Cost	100 ms
Min Cost	100 ms

NODE 3	
Operators	$\{O_1, Seg, O_2, O_3\}$
Reg.	4 KB
Shared Mem.	30 KB
Cost	90 ms
Min Cost	90 ms

NODE 4	
Operators	$\{O_1, Shfl, Seg, O_2, O_3\}$
Reg.	3.9 KB
Shared Mem.	30 KB
Cost	120 ms
Min Cost	90 ms

Figure 7: An example for augmentation optimization

• Further, we do not attempt to insert the Shuffle or Segment operators into nodes in which all the generated kernels can achieve 100% execution efficiency or theoretical occupancy rate. This is because the insertion of operators cannot further improve the execution efficiency or occupancy rate and just brings runtime overhead.

Example. We demonstrate a working example of our B&B based optimization algorithm in Figure 7. The example shows a simple pipeline consisting of three relational operators, O_1, O_2 and O_3. In initialization, the root node is represented by $P = \{O_1, O_2, O_3\}$. As branches, we generate 3 different child nodes by insertions of a single Shuffle, a single Segment and a pair of Shuffle and Segment (denoted Node 2, Node 3 and Node 4 in the figure, respectively) between operators O_1 and O_2. Their resource use is illustrated on the right. As shown in the table corresponding to Node 2, the insertion of the Shuffle operator leads to an increased resource requirements and thus the violation of the resource constraints. Node 4 is marked an sub-optimal along with all its nodes due to the high cost of materialization when a pair of Shuffle and Segment operators are inserted into the pipeline. Finally, the algorithm chooses Node 3 as the optimal solution for further exploration if it has child nodes.

Runtime overhead. The B&B procedure is very efficient in practice, thanks to the effective pruning and the efficient evaluation of the cost model. In our experiments, the augmentation optimizer is able to identify the optimal solution for each query with very low overhead (smaller than 1% of the query execution time).

6 EXPERIMENTS

6.1 Experimental Setup

Table 2: Hardware used for experiments.

	V100	P100	Titan Xp
Core Count	80 x 64	56 x 64	48 x 64
Clock Rate (GHz)	1.53	1.328	1.4
Memory Size (GB)	32	16	12
Memory Bandwidth (GB/s)	900	549	547.7
Memory Type	HBM2.0	HBM2.0	GDDR5
Shared Memory per SM	64KB	64 KB	64 KB
Register File per SM	256 KB	256 KB	256 KB

Hardware. To demonstrate the efficiency Pyper across different hardware generations, we use V100, P100, and Titan Xp GPUs from NVIDIA. All three GPUs are connected to the CPU via x16 PCIe 3.0 interface. The summary of the specification of all three GPUs is provided in Table 2. V100 and P100 have HBM memory; Titan Xp has GDDR5 memory. We mainly present the results on V100 that is a more recent GPU architecture and present the results on different GPU architectures in Section 6.5.

Workload. For our experiments, we use the TPC-H [1] and Star Schema Benchmark (SSB) [3] data sets with a scale factor of 50 (50 GB in size). SSB is a simplified version TPC-H benchmark and has been widely used in previous studies on OLAP. We use the entire set of SSB queries for evaluation. Since our system front-end is based on GPL, we do not currently support the processing of sub-queries and leave this for future work. Hence, we used all the eight queries in TPC-H which do not contain sub-queries (Q1, Q3, Q5, Q6, Q10, Q12, Q14 and Q19). The detailed SQL clauses can be found in their benchmark websites [1, 3].

All our experiments are based on the entire data that was already loaded on to the GPU global memory during initialization. With the rise in GPU memory capacity and the increased number of GPUs on a modern server(up to 16), we have witnessed hundreds of gigabytes of GPU memory on a single server, where many analytical workloads can be processed completely on GPUs.

Experimental Outline. First, we show the impact of our estimation on divergence and resource contention, as well as the effectiveness of augmentation optimizer. In order to demonstrate that impact, we have integrated our cost model and augmentation optimizer into a previous pipelined query processing engine on the GPU named GPL [30].

Second, we evaluate the overall performance benefit of Pyper with other implementations on the GPU. Here, we omit the comparison with the systems on the CPU such as Hyper [25], since the previous studies [5] have shown that their GPU-based systems are faster than Hyper. Specifically, we conduct a comprehensive comparison using the following GPU-based systems: Ocelot [17], GPL [30], Hawk [5], Omnisci [2] and DogQC [12]. Ocelot in a kernel based execution system that materializes the intermediate data between operators in the GPU global memory, whereas GPL is a pipelined query processing system that makes use of GPU L2 cache for moving the data between the operators. All other systems are compiled systems. To the best of our knowledge, Hawk [5] is the state-of-the-art opensource system that supports code generation for GPUs. However, even Hawk only supports earlier generations of GPUs and cannot run on the V100 GPU. Hence, we modified Hawk [5] for evaluation (denoted as Hawk-M) so that it can run on V100. Omnisci [2] is considered to be the state-of-the-art open-sourced commercial compiled system for GPUs.

Third, in Section 6.4, we evaluate the individual techniques in terms of efficiency and effectiveness of the Shuffle and Segment operators in reducing the divergence and resource contention among threads, respectively.

Fourth, in Section 6.5, we evaluate the generality of the Shuffle and the Segment operators along with the proposed cost model across different GPU hardware.

Additional experimental results can be found in our technical report [29].

6.2 Model Evaluation

Key Findings: Our analytical model is highly accurate, with less than 6.6% error on average for the tested queries. Without consideration of divergent execution and resource contention in compiled systems, existing models have a much higher error.

Estimation errors. To demonstrate our analytical model's accuracy, we compare the *relative error* associated with the estimation of the query execution cost by Pyper and GPL for the tested queries in Figure 8. We define the relative error as $\frac{|\widehat{Q}_m - \widehat{Q}_e|}{\widehat{Q}_m}$, where $\widehat{Q}_m$ and

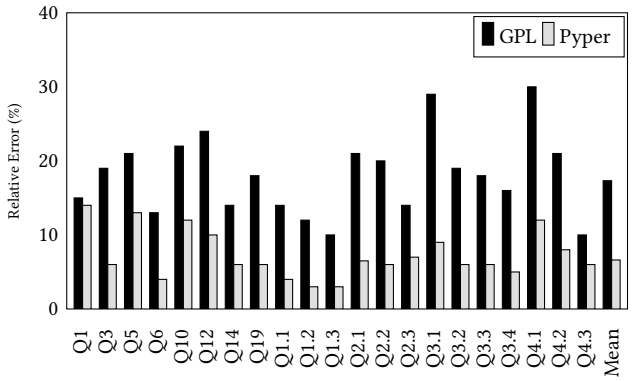


Figure 8: Relative error in the execution cost estimation by Pyper and GPL for TPC-H and SSB queries on V100

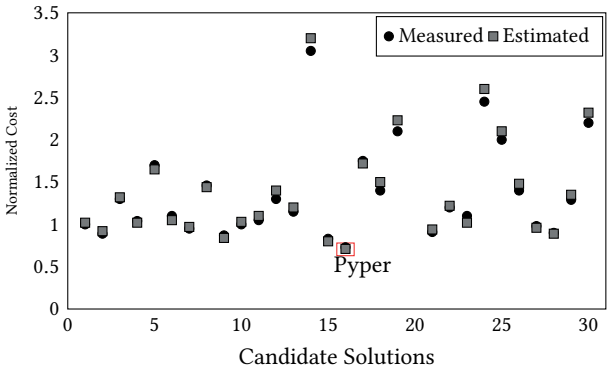


Figure 9: Normalized measured and estimated execution cost of different candidate solutions of Q6 on V100

$\widehat{Q_e}$ represent the total measured and estimated (by the analytical model) execution time of query Q. Note, the relative error values of Pyper presented in Figure 8 are from the query plan obtained from our augmentation optimizer. We also show the geometric mean of all the results as "Mean" in the figure. We have studied the relative error of other plans including those generated during the augmentation optimization, and observed similar results to Figure 8. We present one representative result later in Figure 9, and omit the results for other queries due to space limits.

The results in Figure 8 clearly show that Pyper is able to achieve significantly higher accuracy compared to GPL, for most queries. Taking a closer look, we can observe that Pyper can achieve much lower relative errors than GPL in some queries, or have similar relative errors to GPL in other queries. To understand this, we take TPC-H Q1 and Q6 as two representative queries [1]. 1) TPC-H Q6 represents the cases where Pyper achieves much lower relative errors than GPL. TPC-H Q6 is a query with a low-selectivity filter where many tuples are filtered out and the operators encounter a high degree of divergence. Pyper is able to address this divergent execution, whereas GPL does not. 2) TPC-H Q1 represents the cases where Pyper has similar relative errors to GPL. This is because TPC-H Q1 is an aggregation query with higher selectivity (most tuples are remained as results in the pipeline), i.e., the operators in Q1 encounter minimal divergence during execution. As we show later in Section 6.4, the improvement of the model accuracy can come from our improved estimation on resource contention.

We analyze the source of the relative error of our model. One source can be that the cost of shared memory and global memory instructions can vary significantly based on the number of memory bank conflicts and access patterns, making it difficult for Pyper to accurately estimate this cost. The other potential source of errors can be from multiple joins, where join intensive queries like Q5, Q10 and Q4.1 compared to the other queries. This is because the join queries lead to errors in estimating the cost of synchronization in accessing the hash table in parallel.

Effectiveness of query augmentation optimizer. We demonstrate the effectiveness of our query plan augmentation optimizer. We present the measured and estimated execution time of all candidate solutions (30 in total after pruning) explored by the augmentation optimizer for TPC-H Q6 in Figure 9. The execution time is normalized with respect to that of GPL. We observe similar results for other queries and omit the results.

The results in Figure 9 clearly show the following observations. First, there is significant performance difference between the different candidate solutions (the difference can be 2.75x for TPC-H Q6). Hence, it is very important to accurately identify the optimal insertion points of Shuffle and Segment operators into a query plan. In fact, an inaccurate estimation of the execution time leading to the selection of a non-optimal configuration could lead to severe degradation in query performance. Second, the augmentation optimizer in Pyper with the help of the analytical model is able to identify the optimal solution for Q6 (highlighted by the rectangle of Pyper in Figure 9), thus demonstrating its effectiveness. The augmentation optimization is quite lightweight due to the B&B approach, with the runtime overhead smaller than 1% of the query execution time.

6.3 Overall Comparison

Key Findings: For TPC-H queries, on average the code generated by Pyper outperforms Ocelot, Omnisci, GPL, Hawk-M and DogQC by 34.73x, 5.01x, 3.06x, 1.60x and 1.40x, respectively. Pyper on average achieves 6.59x, 2.55x, 2.35x, 1.52x performance improvement over Ocelot, Omnisci, GPL and Hawk-M for SSB queries.

We compare the overall performance of Pyper against other systems using TPC-H and SSB queries in Figures 10 and 11, respectively. Compiled systems are generally much faster than non-compiled systems.

Omnisci is a commercial system with a more complete set of functionalities than other systems in the test. It is still slower than other tested systems, due to its low execution efficiency of the GPU kernels. Further, Omnisci also fails to efficiently optimize complex join queries like Q3, Q5 and Q10, as demonstrated by the significantly worse performance of Omnisci for these queries. Overall, Pyper outperform Omnisci by 5.01x and 2.55x for TPC-H and SSB queries on average.

As mentioned before, efficient use of the Shuffle and Segment operators helps Pyper minimize the divergent execution and resource contention among threads. This combined with the availability of a cost model that avoids unnecessary insertions of the Shuffle and Segment operators help Pyper to outperform Hawk-M by 1.60x and 1.55x on average for TPC-H and SSB queries respectively.

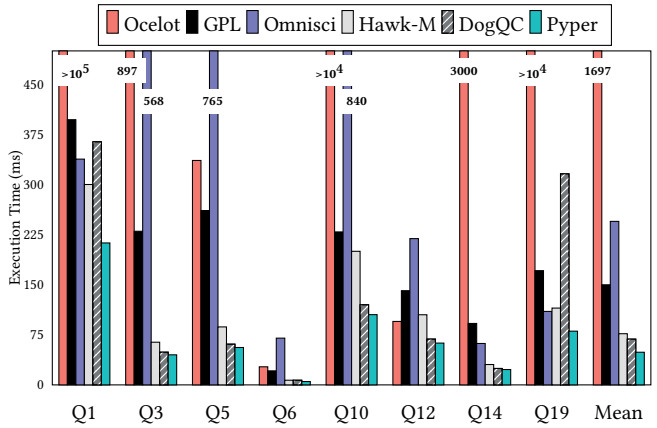


Figure 10: Overall performance evaluation of Pyper for TPC-H queries on the V100 GPU.

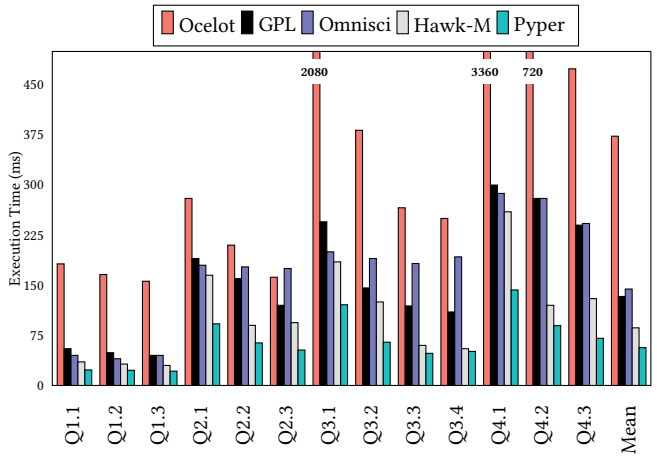


Figure 11: Overall performance evaluation of Pyper for SSB queries on the V100 GPU.

Even though DogQC segments its pipeline and makes use of techniques to minimize workload divergence and resource contention, Pyper still manages to outperform DogQC by up to 1.4x on average (up to 3.9x on TPC-H queries), due to the following reasons. First, DogQC lacks a cost model that can help determine the optimal insertions of Shuffle and Segment operators (the importance has been demonstrated by the results in Figure 9). Second, Pyper adopts more efficient implementations for the Shuffle and Segment operators. Note, we omit DogQC in Figure 11 since the open-source version of DogQC does not currently support SSB queries.

Time Breakdown. To get a deeper understanding of the reasons behind the performance improvement of Pyper, we present the execution time breakdown of Pyper along with Hawk-M in Figure 12. The breakdown involves the overhead of divergent execution, the overhead of resource contention, the cost of Shuffle and Segment (including data shuffling resulting from Shuffle and the cost of additional data materialization arising from Segment). For the clarify of figure, Qx(H) and Qx(P) represent the execution time breakdown of code generated corresponding to query Qx by Hawk-M and Pyper, respectively. We have the following observations.

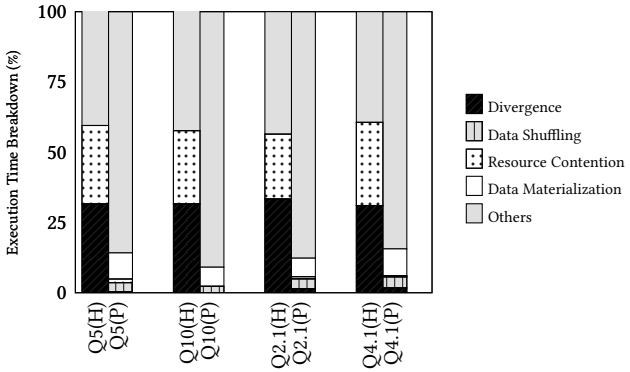


Figure 12: Time breakdown of Pyper and Hawk-M on V100

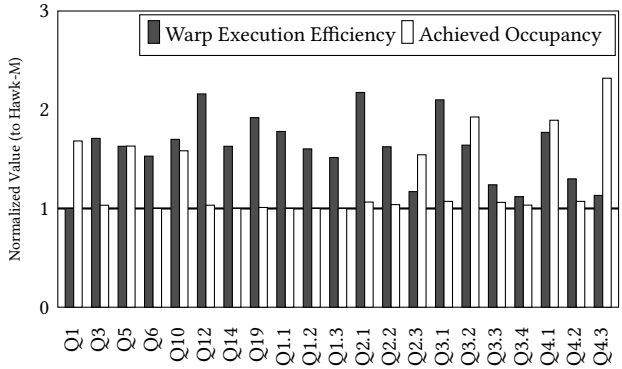


Figure 13: Normalized (to Hawk-M) warp execution and occupancy rate for TPC-H and SSB queries on V100

First, due to the use of the Shuffle and Segment operators, Pyper is able to generate code that almost eliminates the overhead due to divergent execution and resource contention among threads. Second, the use of the Shuffle and Segment operator adds very little runtime overhead associated with data shuffling and materialization (less than 12%). Thus, such small overhead is much smaller than the performance improvement by resolving divergent execution and resource contention among the threads.

6.4 Individual Operator Evaluation

Key Findings: The profiling results show that, with Shuffle and Segment operators, Pyper achieves up to 2.20x higher warp execution efficiency and 2.30x higher occupancy rate than Hawk-M.

For a more thorough comparison of individual operators, we present the execution time of Pyper (w/o Shuffle) and Pyper (w/o Segment), which are the same as Pyper except without using Shuffle and Segment, respectively. The results are shown in Figure 14. Both Shuffle and Segment individually helps improve the performance of query execution on GPUs. Further, the benefit of each operator individually varies depending on the query.

In the following, for each operator, we present the detailed profiling results to demonstrate that Shuffle and Segment operators are able to eliminate most of the overhead from divergent execution and resource contention.

We use the following metrics collected using the NVProfile tool from NVIDIA: 1) *warp execution efficiency* which represents the average number of active threads per warp and helps account for

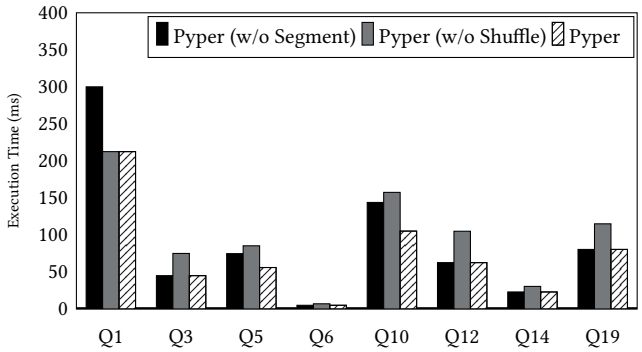


Figure 14: Performance benefit of Shuffle and Segment operators for TPC-H queries on V100.

the level of divergence among the GPU threads and 2) *achieved occupancy* which represents the maximum number of active warps per SM and helps account of the resource contention among the threads. Higher the warp execution efficiency and achieved occupancy indicate lower levels of divergence and resource contention among threads, respectively. We present the normalized warp execution efficiency and occupancy values of Pyper with respect to Hawk-M in Figure 13.

Impact of Shuffle Operator. As demonstrated by the normalized warp execution efficiency values in Figure 13, the use of the Shuffle operator helps improve the warp execution efficiency of almost all queries used in our study. Overall, the warps executing the code generated by Pyper achieves up to 2.20x higher warp execution efficiency compared to the Hawk-M implementation. Such significant improvement in the warp execution efficiency of the kernels means that threads executing the code generated by Pyper can make more efficient use of the GPU hardware by allowing a larger number of active threads in each warp.

Impact of Segment Operator. As shown in Figure 13, Pyper is able to achieve up significantly higher occupancy rate when compared to Hawk-M for queries Q1, Q5, Q10, Q2.3, Q3.2, Q4.1 and Q4.3. This is because one or more kernels generated by the Hawk-M implementation for these queries encounter severe resource contention on GPUs due to its high resource requirement. Pyper breaks down those large pipelines in these queries using the segment operator, thus reducing the resource requirements of the kernels and improving the parallelism.

We present the resource requirements of the most expensive kernel generated by the Hawk-M implementation and Pyper for queries Q1, Q5, Q10, Q2.3, Q3.2, Q4.1 and Q4.3 in Table 3. The shared memory and register requirements of the queries in Table 3 clearly show that the kernels generated by Pyper has significantly lower resource requirements compared to Hawk-M. More importantly, the resource requirements in Pyper are low enough to ensure up to 64 simultaneous active warps per SM on modern GPU hardware (with 64 KB shared memory and 256 KB registers per SM). Hence, the kernels generated by Pyper is able to achieve up to 2.3x better occupancy rate on GPUs (Figure 13).

6.5 Evaluation on Different GPU Architectures

We have conducted experiments on two older generation of GPUs including P100 and Titan Xp. The results of TPC-H queries are shown in Figure 15. Overall, the results demonstrate the efficiency of Pyper and the generality of both our operators and the cost model for different generations of GPU hardware. The slightly lower performance gain of Pyper is mainly due to the reduced performance benefit of the Segment operator.

Table 3: Resource use of Hawk-M and Pyper on V100

Query	Shared Mem. / Warp (KB)		Register / Warp (KB)		Occupancy (%)	
	Hawk-M	Pyper	Hawk-M	Pyper	Hawk-M	Pyper
Q1	0.1	0.1	4.5	3.8	55.2	93.1
Q5	1.3	1.0	4.4	3.1	58.1	93.2
Q10	1.3	0.9	3.3	3.1	56.8	90.0
Q2.3	1.1	0.6	3.8	3.5	60.2	92.5
Q3.2	1.5	1.0	4.3	3.7	49.0	88.3
Q4.1	1.1	0.8	4.0	3.8	47.3	89.1
Q4.3	1.3	1.0	4.5	3.2	40.1	93.0

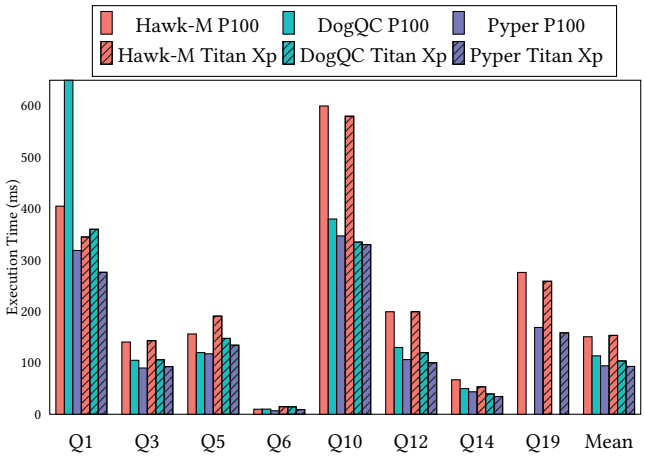


Figure 15: Performance evaluation on P100 and Titan Xp

7 CONCLUSION

JIT compilation-based code generation of queries has evolved as a powerful paradigm for query processing on GPU. In this study, we find that, thanks to advanced techniques such as database compression and JIT compilation, memory stalls are no longer the most significant bottleneck. Instead, existing compiled systems encounter severe overhead in divergent execution and resource contention because they adopt a monolithic approach for generating the code of query plan on the GPU. To improve the execution efficiency, we present *Pyper*, a compiled system designed for GPUs. Pyper introduces two new query plan transformation operators, *Shuffle* and *Segment*, which can be inserted into any query pipeline of the physical plan to reduce the divergent execution and resource contention. Further, Pyper makes use of an analytical cost model and an effective optimizer to determine the optimal use of the Shuffle and Segment operators. Our experiments show that Pyper generates code that on average is able to improve the performance of TPC-H queries by 34.73x, 5.01x, 3.06x, 1.60x and 1.40x when compared to Ocelot, Omnisci, GPL, Hawk and DogQC, respectively. Pyper on average achieves 6.59x, 2.55x, 2.35x, 1.52x performance improvement over Ocelot, Omnisci, GPL and Hawk for SSB queries.

8 ACKNOWLEDGEMENT

This work is in part supported by a MoE AcRF Tier 1 grant (T1 251RES1824) and Tier 2 grant (MOE2017-T2-1-122) in Singapore.

REFERENCES

[1] Tpc-h. http://www.tpc.org/tpch/, 1999.

[2] Omnisci. accelerated analytics platform. https://www.omnisci.com/, 2009.

[3] Star schema bechmark. https://www.cs.umb.edu/ poneil/StarSchemaB.PDF, 2009.

[4] H. Anzt, T. Cojean, C. Yen-Chen, J. Dongarra, G. Flegar, P. Nayak, S. Tomov, Y. M. Tsai, and W. Wang. Load-balancing sparse matrix vector product kernels on gpus. *ACM Trans. Parallel Comput.*, 7(1), Mar. 2020.

[5] S. BreB, B. Kocher, H. Funke, S. Zeuch, T. Rabl, and V. Markl. Generating custom code for efficient query execution on heterogeneous processors. *The VLDB Journal*, 27(6):797–822, Dec. 2018.

[6] P. Chrysogelos, M. Karpathiotakis, R. Appuswamy, and A. Ailamaki. Hetexchange: Encapsulating heterogeneous cpu-gpu parallelism in jit compiled engines. *Proceedings of the VLDB Endowment*, page 13, 2019.

[7] P. Chrysogelos, P. Sioulas, and A. Ailamaki. Hardware-conscious query processing in gpu-accelerated analytical engines. *Proceesings of the 9th Biennial Conference on Innovative Data Systems Research*, page 9, 2019.

[8] A. Crotty, A. Galakatos, and T. Kraska. Getting swole: Generating access-aware code with predicate pullups. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pages 1273–1284, 2020.

[9] Y. Djenouri, D. Djenouri, A. Belhadi, and A. Cano. Exploiting gpu and cluster parallelism in single scan frequent itemset mining. *Information Sciences*, 496:363 – 377, 2019.

[10] R. Fang, B. He, M. Lu, K. Yang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Gpuqp: Query co-processing using graphics processors. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, SIGMOD '07, pages 1061–1063, New York, NY, USA, 2007. ACM.

[11] H. Funke, S. Breß, S. Noll, V. Markl, and J. Teubner. Pipelined query processing in coprocessor environments. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, pages 1603–1618, New York, NY, USA, 2018. ACM.

[12] H. Funke and J. Teubner. Data-parallel query processing on non-uniform data. *Proc. VLDB Endow.*, 13(6):884–897, Feb. 2020.

[13] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, Dec. 2009.

[14] J. He, M. Lu, and B. He. Revisiting co-processing for hash joins on the coupled cpu-gpu architecture. *Proc. VLDB Endow.*, 6(10):889–900, Aug. 2013.

[15] J. He, S. Zhang, and B. He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4):329–340, Dec. 2014.

[16] M. Heimel, M. Kiefer, and V. Markl. Self-tuning, gpu-accelerated kernel density models for multidimensional selectivity estimation. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1477–1492, New York, NY, USA, 2015. ACM.

[17] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[18] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk. Gpu join processing revisited. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 55–62, New York, NY, USA, 2012. ACM.

[19] F. Khorasani, K. Vora, R. Gupta, and L. N. Bhuyan. Cusha: Vertex-centric graph processing on gpus. In *Proceedings of the 23rd International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '14, page 239–252, New York, NY, USA, 2014. Association for Computing Machinery.

[20] A. Kohn, V. Leis, and T. Neumann. Making compiling query engines practical. *IEEE Transactions on Knowledge and Data Engineering*, pages 1–1, 2019.

[21] A. Li, S. Song, J. Chen, J. Li, X. Liu, N. Tallent, and K. J. Barker. Evaluating Modern GPU Interconnect: PCIe, NVLink, NV-SLI, NVSwitch and GPUDirect. *IEEE Transactions on Parallel and Distributed Systems*, pages 1–1, 2019.

[22] C. Lutz, S. Breß, S. Zeuch, T. Rabl, and V. Markl. Pump up the volume: Processing large data on gpus with fast interconnects. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, page 1633–1649, New York, NY, USA, 2020. Association for Computing Machinery.

[23] P. Menon, T. C. Mowry, and A. Pavlo. Relaxed operator fusion for in-memory databases: Making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow.*, 11(1):1–13, Sept. 2017.

[24] D. R. Morrison, S. H. Jacobson, J. J. Sauppe, and E. C. Sewell. Branch-and-bound algorithms: A survey of recent advances in searching, branching, and pruning. *Discrete Optimization*, 19:79 – 102, 2016.

[25] T. Neumann. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow.*, 4(9):539–550, June 2011.

[26] T. Neumann and V. Leis. Compiling database queries into machine code. *IEEE Data Eng. Bull.*, 2014.

[27] OmniSci. https://www.omnisci.com/. 2019.

[28] J. Paul, B. He, S. Lu, and C. T. Lau. Revisiting hash join on graphics processors: A decade later. In *2019 IEEE 35th International Conference on Data Engineering Workshops (ICDEW)*, pages 294–299, April 2019.

[29] J. Paul, B. He, S. Lu, and C. T. Lau. Improving execution efficiency of just-in-time compilation based query processing on gpus (complete version). In *https://github.com/Xtra-Computing/Pyper*, 2020.

[30] J. Paul, J. He, and B. He. Gpl: A gpu-based pipelined query processing engine. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1935–1950, New York, NY, USA, 2016. ACM.

[31] H. Pirk, O. Moll, M. Zaharia, and S. Madden. Voodoo - a vector algebra for portable database performance on modern hardware. *Proc. VLDB Endow.*, 9(14):1707–1718, Oct. 2016.

[32] A. Shanbhag, S. Madden, and X. Yu. A study of the fundamental performance characteristics of gpus and cpus for database analytics (extended version). *arXiv preprint arXiv:2003.01178*, 2020.

[33] S. Singh and R. Nasre. Optimizing graph processing on gpus using approximate computing: Poster. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, page 395–396, New York, NY, USA, 2019. Association for Computing Machinery.

[34] E. A. Sitaridi and K. A. Ross. Optimizing select conditions on gpus. In *Proceedings of the Ninth International Workshop on Data Management on New Hardware*, DaMoN '13, New York, NY, USA, 2013. Association for Computing Machinery.

[35] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, New York, NY, USA, 2014. ACM.

[36] Y. Yuan, R. Lee, and X. Zhang. The yin and yang of processing data warehousing queries on gpu devices. *Proc. VLDB Endow.*, 6(10):817–828, Aug. 2013.

[37] E. Z. Zhang, Y. Jiang, Z. Guo, and X. Shen. Streamlining gpu applications on the fly: Thread divergence elimination through runtime thread-data remapping. In *Proceedings of the 24th ACM International Conference on Supercomputing*, ICS '10, page 115–126, New York, NY, USA, 2010. Association for Computing Machinery.

[38] Y. Zhang, Y. Zhang, J. Lu, S. Wang, Z. Liu, and R. Han. One size does not fit all: accelerating olap workloads with gpus. *Distributed and Parallel Databases*, pages 1–43, 2020.