Profiling a GPU Database Implementation

A Holistic View of GPU Resource Utilization on TPC-H Queries

Emily Furst
University of Washington
Box 352350
Seattle, Washington 98195-2350
eafurst@cs.washington.edu

Mark Oskin
University of Washington
Box 352350
Seattle, Washington 98195-2350
oskin@cs.washington.edu

Bill Howe
University of Washington
Box 352350
Seattle, Washington 98195-2350
billhowe@cs.washington.edu

ABSTRACT

General Purpose computing on Graphics Processing Units (GPGPU) has become an increasingly popular option for accelerating database queries. However, GPUs are not well-suited for all types of queries as data transfer costs can often dominate query execution. We develop a methodology for quantifying how well databases utilize GPU architectures using proprietary profiling tools. By aggregating various profiling metrics, we break down the different aspects that comprise occupancy on the GPU across the runtime of query execution. We show that for the Alenka GPU database, only a small minority of execution time, roughly 5% is spent on the GPU. We further show that even on queries with seemingly good performance, a large portion of the achieved occupancy can actually be attributed to stalls and scalar instructions.

CCS CONCEPTS

• **Information systems** → *Relational parallel and distributed DBMSs*; • **Computer systems organization** → *Single instruction, multiple data*;

KEYWORDS

GPGPU, Profiling, GPU Database

ACM Reference format:
Emily Furst, Mark Oskin, and Bill Howe. 2017. Profiling a GPU Database Implementation. In *Proceedings of DaMoN'17, Chicago, IL, USA, May 15, 2017,* 6 pages.
DOI: http://dx.doi.org/10.1145/3076113.3076119

1 INTRODUCTION

Databases are increasingly using GPUs to accelerate query processing [4]. Parallel databases have been an area of interest since the early 1990s, and because GPU architectures offer a large amount of parallelism, it is only natural that research would turn to GPU database implementations. While GPUs promise impressive performance speedups, it can be difficult to write GPGPU code well and fully utilize the parallelism of GPUs without extensive domain knowledge. With this in mind, how efficient are GPUs at query processing?

In this paper we present profiling results from the Alenka [3] database executing on a state of the art NVidia GPU. We've focused on full-query execution and examined a representative set of queries from the TPC-H benchmark [2] suite. We examine various profiling metrics in order to better understand what exactly happens over the course of a query execution. From this effort we derive the following conclusions:

- Communication costs to/from the GPU dominate GPU execution time. Our results show that on average, only about 5% of execution time is spent on the GPU. Database engines that can store in-memory data on the GPU across queries are preferred to those that must shuffle the relevant portions of the database to the GPU for each query.
- Depending on the type of query GPUs can be either *highly efficient* or *highly inefficient* at processing it. Queries comprised of operations that require a large amount of synchronization, such as aggregations, are not as well suited for GPU execution as queries recquiring less synchronization. Query optimization engines need to be judicious about using the GPU.
- Queries may appear to achieve high performance as they leverage the available parallelism in GPUs, but oftentimes they spend a large portion of execution time on stalls and executing scalar instructions. It is important that database engines understand how GPU architectures and memory hierarchies differ from CPUs.

In the next section we briefly summarize GPU architecture and how databases are utilizing them. In Section 3 we describe the methodology we use to explore GPU performance while executing database tasks. We then present profiling data in Section 4. Section 5 revisits the relevant related work, and Section 6 discusses common conclusions across our results and concludes the paper.

2 BACKGROUND

In this section we briefly describe GPU architectures and how databases accelerate query processing with them.

2.1 GPU architecture

GPUs have evolved from fixed-function accelerators (1990s) to those with limited flexibility in processing capabilities (2000s) to what we have now, which are fully programmable, highly parallel execution engines. At a high level AMD and NVidia GPUs use similar concepts, and so we'll describe GPU architectures using NVidia terminology.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
DaMoN'17, Chicago, IL, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM.
978-1-4503-5025-9/17/05. . . $15.00
DOI: http://dx.doi.org/10.1145/3076113.3076119

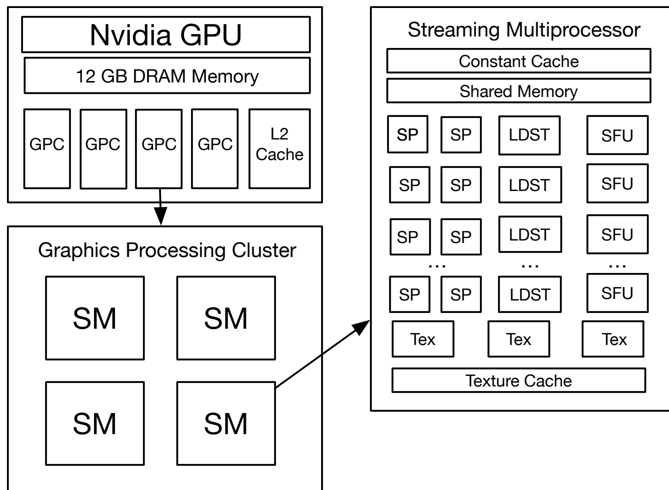


Figure 1: A basic block diagram breaking down the major parts of an Nvidia GPU. Warps are mapped to SMs, and kernels are executed on GPCs.

Programmers express GPU computations as a hierarchy of parallel threads. Threads are grouped into a warp. Warps are grouped into kernels. Threads within a warp can efficiently synchronize with each other, and can count on each other executing concurrently. In contrast, a thread in one warp cannot expect a thread in a different warp to be executing concurrently.

To write efficient code, programmers should aim to have each thread of a warp follow the same control flow. On current architectures, if control paths diverge then execution resources will be wasted. This inefficiency is because a warp is executed on a *streaming multiprocessor* (SM) that follows the single instruction multiple data (SIMD) execution model. In hardware, each thread of a warp essentially has the same program counter. Divergence in the control flow is handled via predication.

Threads have access to a hierarchy of memory resources. Each thread is provided a dedicated and custom (to the kernel) sized register file. Threads within the same warp have access to a private data-store (called *shared memory* on NVidia architectures). Key to performance, simultaneous access to the shared memory is provided for each thread within a warp. Threads also have simultaneous access to a constant-memory, although this structure is rarely used for general-purpose GPU code, and is generally intended for graphics rendering. Warps have access to a *global memory* through a cache. Bandwidth through this cache is extremely limited, however, and codes that consist of mostly random memory accesses will be largely idle on GPUs. Beyond this cache is either DRAM on the GPU itself or memory shared with the CPU, depending on the particular product used.

The architecture and programming model of GPUs has mostly been driven by the separation of address spaces. Thus, programs are typically written to explicitly move data from a CPU process space to the GPU and back again, as necessary. Recent advances in both NVidia and AMD hardware has provided a shared virtual address space for GPUs and CPUs [13], but the legacy style of programming remains pervasive in GPU code.

2.2 Databases on GPUs

More and more frequently, databases are trying to accelerate their operations with GPUs [4]. Such approaches use GPUs primarily for query processing, but there have been many approaches tackling GPU database processing at different levels of a database system [4].

The general conclusion is that GPUs are better at joins and worse at selections than CPUs due to expensive memory transfer costs [6, 8]. As a result, the most successful GPU database implementations are ones that avoid data transfers as much as possible. Bress et. all come to the conclusion that an optimal GPU DBMS would reside in-memory, use columnar storage, and implement one-operator-at-a-time bulk processing [4]. One-operator-at-a-time as opposed to tuple-at-a-time processing allows for more cache friendly memory accesses and makes more effective use of the memory hierarchy.

MapD is a data processing and visualization engine that can execute queries on either the CPU or GPU [10]. It tries to keep as much data in memory as possible and uses columnar storage. However, it does not assume that input data will fit in a GPU's RAM, so it applies a streaming mechanism for processing. MapD's basic processing model is operator-at-a-time, and its optimizer splits queries into chunks and assigns them to the most suitable processor.

GPUDB is a query execution engine that only supports query processing on the GPU [14]. It keeps the database in the main memory of the CPU and is able to optimize its performance by pinning the data in host memory. It utilizes column-based storage and uses a block-oriented processing model to overlap data transfers with computation.

OmniDB is a DBMS that optimizes for good code maintainability by creating a hardware-oblivious database kernel that accesses the hardware via adapters [15]. It is based on GPUQP [7] and uses column-oriented in-memory data storage model. Its processing model supports different granularities of work ranging from entire queries to chunks of tuples. It determines which processing device to execute on based on the current load of the device.

The common objective across recent GPU database implementations is combating the IO bottleneck by attempting to keep important data in-memory and by limiting data transfer. In order to see optimal performance, it is important that GPU database query processing engines include a cost model to decide whether it is better to execute a query on the GPU or CPU [4].

3 METHODOLOGY

Both AMD [12] and NVidia [1] provide similar mechanisms to profile execution on their GPUs. Like modern CPUs, GPU hardware provides a wealth of performance counters. But hardware cannot simultaneously profile all features. Profiling tools must be configured to track a handful of metrics (in our case one) at a time. Typical individual metrics include cycles spent accessing memory, or number of floating point operations executed. Unlike CPUs, there are tens to hundreds of streaming multiprocessors. Performance counters are thus aggregated (and averaged) across all SM units.

Most work analyzing the performance of GPU database implementations focuses mainly on query execution time and data transfer time. A detailed analysis of GPU performance is most often missing. Our work aims to explore in much more detail the time breakdown and performance of a GPU DBMS.

For our work, we are interested in understanding where the time goes while a GPU is executing a database query. We thus focus our profiling on partitioning GPU execution into several distinct buckets:

- **non-GPU time:** the percentage of execution time where the GPU is idle. This profiling bucket turns out to be the single largest.
- **predicated-instructions:** the percentage of instructions that are *not executed* because of divergent control flow.
- **memory-instructions:** the percentage of instructions the GPU executes that go to memory.
- **control-instructions:** the percentage of instructions that make control decisions for GPU threads.
- **int-instructions:** the percentage of instructions that do integer arithmetic on the GPU. This measure is distinct from that of floating point operations, but in the case of this database, floating point operations are very rare if at all present during execution.
- **other-instructions:** the percentage of instructions that are not integer based. This bucket includes synchronization as well as scalar instructions.
- **stall-cycles:** the percentage of cycles where the GPU is in use, but actually idle. Idleness comes from a variety of sources: not enough register space, waiting on the memory hierarchy, blocked on synchronization, etc.

We present our results in the form of a cumulative distribution function (CDF). On the left of the CDF is execution time where (potentially) the GPU is fully idle. On the far right of the CDF is execution time where the GPU is maximumly used. Note that this does not necessarily mean the GPU is *fully* used, only that this is the GPU kernel that used the GPU the most. Because so much of the execution time is not on the GPU, and so that other items are easily legible, we shift the x-axis on the graphs. Readers should pay particular attention to the labels along the x-axis. Each portion of the CDF between these two points is either non-kernel execution (far left) or grouped by individual GPU kernels, sorted by GPU occupancy. We use this display format because it clearly illustrates how much the GPU is used for the entire program execution, while at the same time facilitating the inspection and actual relevance of individual GPU kernels.

Alenka. Alenka is a column-based GPU database engine written to use vector based processing and high bandwidth of modern GPUs. We profile the Alenka database while running TPC-H queries 1-5 and 7 [2]. We generate TPC-H data with a scale factor of 10 and use the queries provided in the Alenka repository. We examine a selection of TPC-H queries that covers all major representative functionalities which include regular scan, index scan, and join.

We also contrive a query with a significant amount of computation to explore the limitations of Alenka and the TPC-H queries. The query takes the following form:

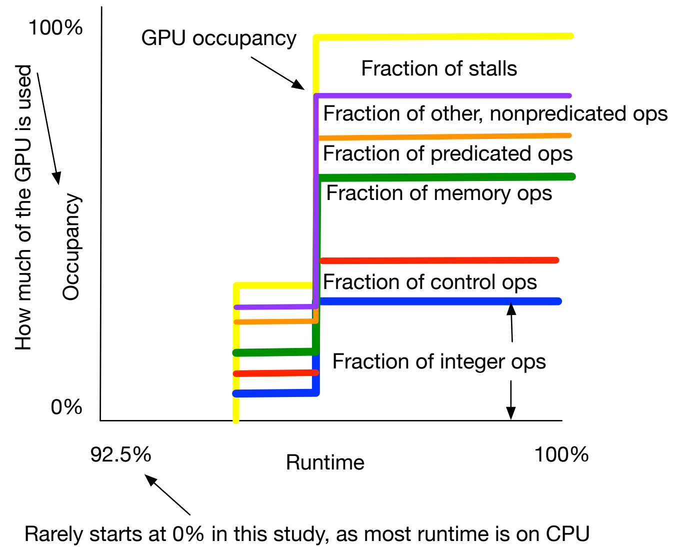


Figure 2: In this study we are interested in how much the GPU is utilized. Execution time is profiled on a GPU kernel by kernel basis and a CDF is generated based upon GPU occupancy. This simplified example depicts one such CDF. Actual results will partition GPU execution time into finer components.

B := FILTER lineitem;
D := SELECT $(...(((price + 1)^3 + 1)^3 + 1)^3 ... + 1)^3 + 1$ as a
FROM B;

Where the pattern of computation used to calculate *a* is demonstrated above. The length of the actual computation used is much longer to better utilize the GPU resources.

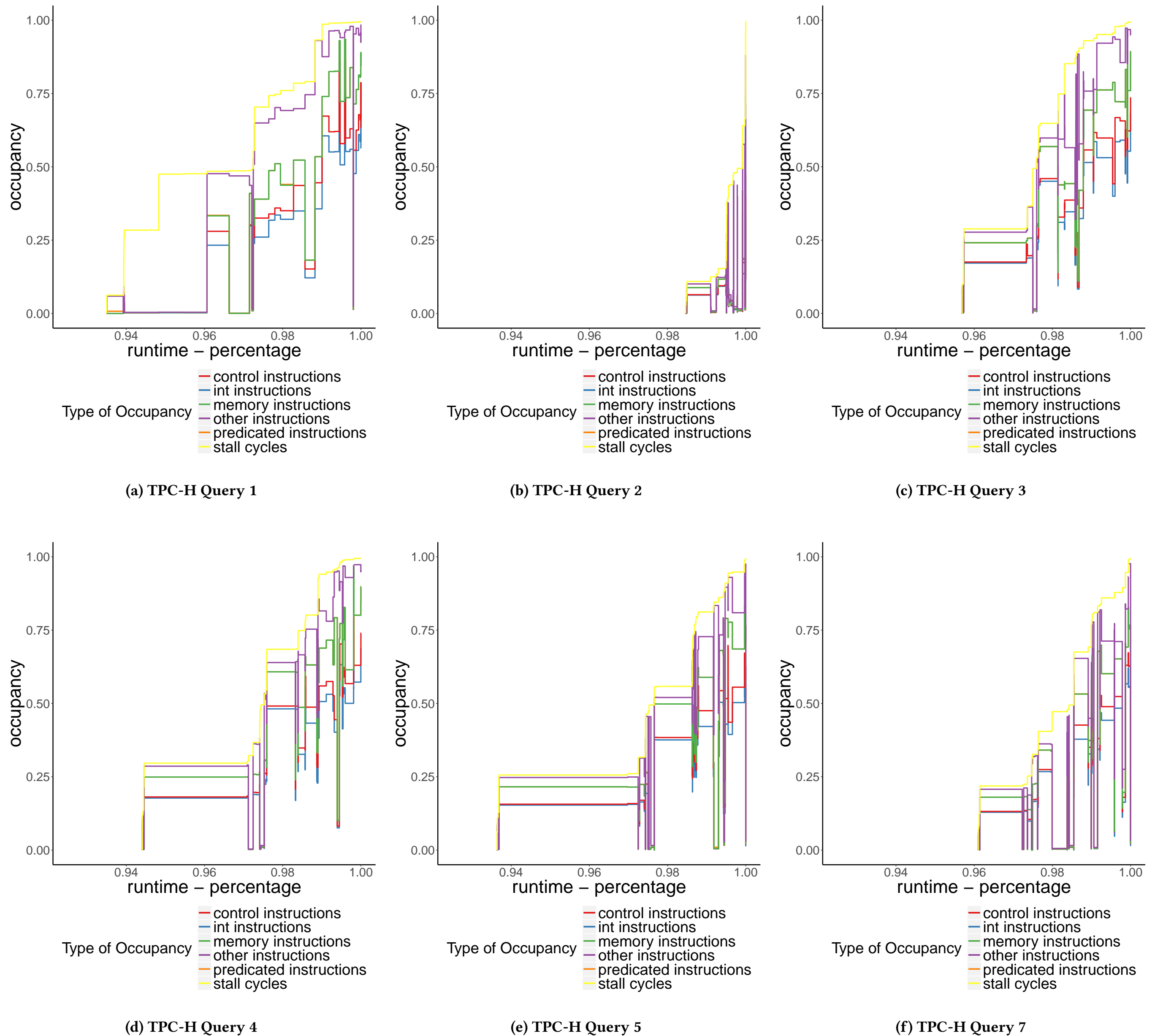
We profile Alenka on a machine with an Nvidia Titan X Pascal GPU and an Intel Core i7-4790K quadcore CPU running Ubuntu 16.04.1. The Titan X has 3584 CUDA cores, 12 GB of memory, and a maximum memory bandwidth of 480 GB/s. We use CUDA 8.0 and its toolkit for compilation of Alenka and collecting profiling results.

We collect profiling results using the Nvidia command line profiling tool, Nvprof [1]. Nvprof allows the user to collect information about CUDA-related activities including events and metrics for CUDA kernels. The profiler supports various modes, but we use primarily the event/metric summary mode and default summary mode.

4 RESULTS

The first, and perhaps most important, result is that 95% of query time is spent with the GPU *idle*. The dominant cost is shuffling data from the CPU to the GPU, and not actually executing the query. Amdahl's Law suggests any future improvements lay not with GPGPU optimization but with optimizing the rest of the system.

Figure 3 shows the results of our profiling of TPC-H queries. The y-axis corresponds to the achieved occupancy where the total achieved occupancy is the ratio of the average active warps to the maximum number of warps supported. The x-axis corresponds



(a) TPC-H Query 1

(b) TPC-H Query 2

(c) TPC-H Query 3

(d) TPC-H Query 4

(e) TPC-H Query 5

(f) TPC-H Query 7

Figure 3: Profiling results of six TPC-H queries on Titan X GPU. Note x-axis labels to note percentage of runtime being shown.

to the runtime percentage. The various lines represent how the achieved occupancy is broken down into different types of activity ranging from integer instructions to stall cycles.

TPC-H query #1 is a selection query that returns a pricing summary report. The profiling results for this query are shown in Figure 3a. From this we can see that the majority of the query processing time, roughly 93%, was spent outside the GPU. This time includes data transfers and other coordination or processing done by the CPU. Overall, the average achieved occupancy for kernels is fairly high, with over 5% of execution time having achieved occupancy of roughly 50% or higher. We can further see that on average, only about half of the achieved occupancy was spent on integer, control, and memory instructions. The other half of achieved occupancy was spent on other unclassified instructions or stalls. While selection is fairly data parallel, there are quite a few sum and average aggregation operations in query one which could be the reason for the large number of stalled cycles and unclassified instructions. The line corresponding to predicated operations is not visible because the total number of predicated operations was

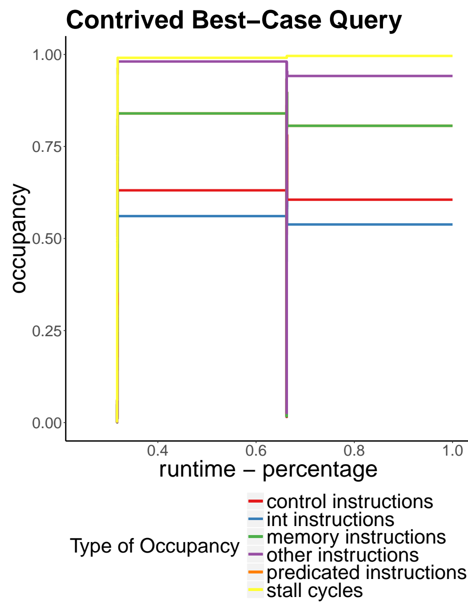


Figure 4: Profiling results of contrived query aimed at making better use of GPU resources. Even with this contrived query that performs an unrealistic amount of floating point arithmetic per record to increase occupancy, we still see only 70% of the runtime involving the GPU.

in general very small; the line actually lies just above the memory instruction line.

Query #2 is another selection query returning the minimum cost suppliers; however, it only returns a maximum of 100 items with the lowest costs. Again, figure 3b shows that only a small portion of execution time is spent on the GPU, in this case, only around 1.5% of the runtime. However, in that small portion of time, the total achieved occupancies are much lower than the ones seen in the results for query one. On average, the achieved occupancies for kernels in query two range between 15% and 20% with a few kernels reaching occupancies above 90% for very short periods of time. Of that achieved occupancy, a large portion of is actually spent on stalls and other scalar instructions, likely because significant coordination is required to produce the top 100 entries.

Figure 3c shows the profiling results for TPC-H query #3, a query that returns the top 10 orders by revenue. About 4% of execution time is spent on the GPU. Of this, roughly 2.5% of the time achieves an average occupancy of 50% or greater, with about 1% of execution time at above roughly 90%. Here though, about two thirds of the occupancy is attributed to integer, control, and memory instructions. This slightly better performance could be related to the larger number of inner joins performed in this query which as noted in section 2 have been shown to often perform better on GPUs than selection operations.

TPC-H query #4 does order priority checking. Figure 3d shows the profiling results for this query. Again, only around 5% of the execution time is spent on the GPU. Of this, 2.5% of execution achieves above 70% occupancy. Like in query 3, about two thirds of the occupancy can be attributed to integer, control, and memory instructions.

Figure 3e shows the results for query #5, a query that returns local supplier revenues. Here, only 93% of runtime is CPU time, and about 2.5% of execution time has an achieved occupancy of over 50%. About half of the achieved occupancy on average was spent on integer and control instructions. Compute_ld_st instructions account for on average about one sixth of the achieved occupancy. This query scans 6 tables and computes quite a few inner joins which could be the reason for such a large number of memory instructions.

Query #7 computes the shipping volume for various nations. The profiling results for this query can be seen in figure 3f. Approximately 4% of the execution time is spent on the GPU and most of this time has an achieved occupancy of at least 20%. The results for query 7 are a bit more interesting as quite a few of the kernels have very high percentages of inactive cycles. The kernels that are not predominantly spent on stalls or inactive cycles, have occupancies that are roughly two-thirds integer, control, and memory instructions. Query 7 is a nested query and this could have impacted the poor profiling results.

Finally, we present the results of our contrived query. By increasing the length of the computation in the contrived query, we were able to see the portion of query execution time where the GPU is active increase. The results shown in figure **??** show a query in which almost 70% of query execution time was spent on the GPU. Further, in general, the occupancy during GPU execution was much higher than seen in the TPC-H queries. These results lead us to believe that performance limitations are not due to the system profiled, Alenka, but in fact due to the nature of TPC-H queries.

5 RELATED WORK

While many database on GPU implementations have been proposed in recent years, few profiling studies focus on analyzing performance beyond execution time. Gregg and Hazelwood make a case for always including data transfer times in performance results in [6]. They find that data transfer to and from the GPU is often a huge limiting factor when comparing performance of CPU and GPU implementations on a variety of applications.

Coutinho et al. develop a tool for GPGPU profiling in [5]. However, their work breaks down performance on the warp level for individual kernels not on the entire program execution level.

Other approaches to profiling GPGPU workloads utilize modeling to estimate performance bottlenecks [9, 11, 16]. They then are able to provide some suggestions to the user that may improve performance.

6 CONCLUSION

GPU implementations of databases are becoming increasingly popular. Previous work has analyzed which types of queries are best suited for GPU acceleration [4]. However, most of the previous work has only looked at the end-to-end performance, without breaking down the execution time.

This work provides a detailed analysis of the runtime performance of a column-based GPU database implementation, Alenka

when running TPC-H queries. We conclude that a major performance limiter is data transfer to and from the GPU. Further, we note that even on queries that appear to achieve good GPU occupancy, a large portion of time is spent on unclassified, scalar instructions or inactive stall cycles. This shows that although a large portion of execution may be spent on stalls, the abundance of parallelism on GPUs easily allows for high performance.

When examining the results across all queries, it seems that most of the query execution time not attributed to stalls or communication costs was spent on integer instructions or memory instructions. Surprisingly, we find that control flow is not a major limiter to performance. Because of this, we find that improvements in the memory hierarchy and arithmetic hardware may yield significant performance gains on database queries running on GPUs.

REFERENCES

[1] Nvprof, command line profiling tool. http://docs.nvidia.com/cuda/profiler-users-guide/.

[2] TPC-H, Benchmark Specification. https://tpc.org/tpch/.

[3] Alenka - A GPU Database Engine. https://github.com/antonmks/Alenka/, 2012–20017.

[4] Bress, S., Heimel, M., Siegmund, N., Bellatreche, L., and Saake, G. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data-and Knowledge-Centered Systems XV*. Springer, 2014, pp. 1–35.

[5] Coutinho, B. R., Teodoro, G. L. M., Oliveira, R. S., Neto, D. O. G., and Ferreira, R. A. C. Profiling general purpose gpu applications. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on* (2009), IEEE, pp. 11–18.

[6] Gregg, C., and Hazelwood, K. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Performance Analysis of Systems and Software (ISPASS), 2011 IEEE International Symposium on* (2011), IEEE, pp. 134–144.

[7] He, B., Lu, M., Yang, K., Fang, R., Govindaraju, N. K., Luo, Q., and Sander, P. V. Relational query coprocessing on graphics processors. *ACM Transactions on Database Systems (TODS) 34*, 4 (2009), 21.

[8] He, B., Yang, K., Fang, R., Lu, M., Govindaraju, N., Luo, Q., and Sander, P. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (2008), ACM, pp. 511–524.

[9] Hong, S., and Kim, H. An integrated gpu power and performance model. In *ACM SIGARCH Computer Architecture News* (2010), vol. 38, ACM, pp. 280–289.

[10] Mostak, T. An overview of mapd (massively parallel database). *White paper. Massachusetts Institute of Technology* (2013).

[11] Sim, J., Dasgupta, A., Kim, H., and Vuduc, R. A performance analysis framework for identifying potential benefits in gpgpu applications. In *ACM SIGPLAN Notices* (2012), vol. 47, ACM, pp. 11–22.

[12] Team, A. D. T. Codexl quick start guide. https://github.com/GPUOpen-Tools/CodeXL/releases/download/v2.0/CodeXL_Quick_Start_Guide.pdf.

[13] Vesely, J., Basu, A., Oskin, M., Loh, G. H., and Bhattacharjee, A. Observations and opportunities in architecting shared virtual memory for heterogeneous systems. In *Performance Analysis of Systems and Software (ISPASS), 2016 IEEE International Symposium on* (2016), IEEE, pp. 161–171.

[14] Yuan, Y., Lee, R., and Zhang, X. The yin and yang of processing data warehousing queries on gpu devices. *Proceedings of the VLDB Endowment 6*, 10 (2013), 817–828.

[15] Zhang, S., He, J., He, B., and Lu, M. Omnidb: Towards portable and efficient query processing on parallel cpu/gpu architectures. *Proceedings of the VLDB Endowment 6*, 12 (2013), 1374–1377.

[16] Zhang, Y., and Owens, J. D. A quantitative performance analysis model for gpu architectures. In *High Performance Computer Architecture (HPCA), 2011 IEEE 17th International Symposium on* (2011), IEEE, pp. 382–393.