GPL: A GPU-based Pipelined Query Processing Engine

Johns Paul* Jiong He*
Nanyang Technological University, Singapore

Bingsheng He †
National University of Singapore

ABSTRACT

Graphics Processing Units (GPUs) have evolved as a powerful query co-processor for main memory On-Line Analytical Processing (OLAP) databases. However, existing GPU-based query processors adopt a *kernel*-based execution approach which optimizes individual kernels for resource utilization and executes the GPU kernels involved in the query plan one by one. Such a kernel-based approach cannot utilize all GPU resources efficiently due to the resource underutilization of individual kernels and memory ping-pong across kernel executions. In this paper, we propose *GPL*, a novel pipelined query execution engine to improve the resource utilization of query co-processing on the GPU. Different from the existing kernel-based execution, GPL takes advantage of hardware features of new-generation GPUs including concurrent kernel execution and efficient data communication *channel* between kernels. We further develop an analytical model to guide the generation of the optimal pipelined query plan. Thus, the *tile* size of the pipelined query execution can be adapted in a cost-based manner. We evaluate GPL with TPC-H queries on both AMD and NVIDIA GPUs. The experimental results show that 1) the analytical model is able to guide determining the suitable parameter values in pipelined query execution plan, and 2) GPL is able to significantly outperform the state-of-the-art kernel-based query processing approaches, with improvement up to 48%.

1. INTRODUCTION

Emerging hardware architectures have driven the database community to revisit and redesign database systems. As a powerful many-core accelerator, GPUs have recently been designed as a query co-processor and become an effective means to improve the performance of main memory databases for OLAP (e.g., [15, 16, 35, 37, 18]). However, existing GPU-based query processors adopt a *kernel*-based execution (KBE) approach which optimizes individual kernels for resource utilization and executes the GPU kernels involved in the query plan one by one. Note, a kernel is a program that runs in parallel on the GPU. The underlying rationale of existing query co-processing is, given that individual kernels are fully optimized, the overall query co-processing performance is optimized.

Despite the effectiveness of such kernel-based approaches on GPU-based query co-processing, we have identified a number of severe issues in performance and resource underutilization. First, executing one kernel at a time causes severe underutilization of GPU resources (e.g., memory bandwidth and computational power). Second, the inter-kernel communication is achieved through global memory or even host main memory in KBE. On the one hand, this explicitly materializes intermediate results across kernels, which increases GPU memory footprint. On the other hand, this causes significant overhead in memory ping-pong to global memory. Hence, KBE may incur serious performance degradations and low utilization on GPU resources (refer to Section 2.2). In this study, we investigate whether and how we can further improve the performance of GPU query co-processing by addressing the shortcomings of KBE.

Recently, some emerging hardware features have been introduced to new-generation GPUs. First, they support concurrent kernel executions. Early GPUs only support a single kernel running on the device. Recent GPUs from NVIDIA (Fermi or Kepler architectures) can support up to 16 kernels concurrently executed within the GPU. Second, a *channel* mechanism is supported for efficient data communication across kernels with proper tuning of configurations. These two hardware features change the assumptions of KBE, which opens new design space for query co-processing.

To take advantage of those emerging hardware features, we propose *GPL*, a novel pipelined query execution engine to improve the resource utilization of query co-processing on the GPU. Pipelined query execution is a widely adopted approach in existing database systems [41, 4]. However, it has never been developed for query co-processing. On new-generation GPUs, the support of concurrent kernel executions and channels enables the concurrent executions of operators in the pipeline and efficient intermediate result transfer among operators. Still, GPUs' special architectural characteristics and programming pattern make it challenging to implement an effective and efficient pipelined query execution engine on GPUs.

Firstly, it is challenging to fairly allocate GPU resources to concurrently running kernels. Imbalanced resource allo-

*Johns Paul and Jiong He contributed equally to this work.
†Work done while at Nanyang Technological University, Singapore.

SIGMOD'16, June 26-July 01, 2016, San Francisco, CA, USA
 ISBN 978-1-4503-3531-7/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2882903.2915224

cation can seriously impact the efficiency of pipelined execution. The scheduling of workloads on GPUs is transparent to users without a fixed mapping between kernels and GPU resources, making resource allocation more difficult. Existing optimizations in pipelined execution such as preemption [22] are hardly applicable to tune the workload distribution, since current GPUs do not support preemption.

Secondly, various tuning knobs (such as the *tile* size) affect the efficiency of the pipelined execution. A more platform-dependent approach is needed to find the optimal pipelined execution plan across GPUs from different vendors including AMD and NVIDIA. Manually tuning the configuration for each hardware platform is ineffective and error-prone.

We have proposed a series of designs and optimizations in GPL to address the above challenges. Firstly, we adapt the number of work-groups to control the amount of resources allocated to each kernel in the pipeline execution. We adopt the tiling technique to logically partition the input data into smaller data tiles as inputs to the pipeline [41, 4]. Thus, workload imbalance can be minimized by adapting the tile size. Secondly, we further develop fine-grained and lightweight operators that are efficient for pipelined executions, rather than simply adopting the operator implementation of existing GPU-based query co-processors [15, 40, 16, 37, 35, 18]. Finally, to find the optimal configuration for the parameters of pipelined execution, we propose an analytical model that has taken all tuning knobs as well as hardware specifications into consideration. Thus, GPL is portable across various platforms with little modification.

We have conducted experiments with TPC-H queries on both AMD and NVIDIA GPUs, in comparison with the state-of-the-art kernel-based query processing approaches [15, 16, 18]. The experimental results show that 1) the analytical model is able to guide the selection of suitable parameter values in the pipelined query execution plan, and 2) GPL is able to achieve a much better resource utilization and significantly outperform the state-of-the-art kernel-based query processing approaches, with a performance improvement up to 48% on the AMD GPU. Experimental results for NVIDIA GPUs can be found in Appendix A. To the best of our knowledge, GPL is the first pipelined query execution engine on GPUs.

The contributions of this paper can be summarized as follows: (1) we have demonstrated severe resource underutilization of existing GPU-based query co-processors; (2) we have developed a novel pipelined query execution engine to improve resource utilization of query co-processing on the GPU; (3) we have proposed an analytical model that can determine the optimal configuration for the parameters of pipelined execution.

The remainder of this paper is organized as follows. In Section 2, we introduce the background on GPU architectures and pitfalls of KBE on GPUs as the motivations for the design and development of GPL. In Section 3, we elaborate the design and implementation details of GPL, followed by an analytical model in Section 4. We present the experimental results in Section 5. We review the related work in Section 6 and conclude our paper in Section 7.

2. BACKGROUND AND MOTIVATION

In this section, we first introduce the background on GPUs. Next, we elaborate the performance issues of current GPU query co-processors.

Table 1: Hardware specification.

	AMD	NVIDIA
#CU	8	15
Core frequency (MHz)	720	875
Private memory/CU (KB)	vector:64, scalar:8	64
Local memory/CU (KB)	32	48
Global memory (GB)	32	12
Cache (MB)	4	1.5
Concurrent kernels	2	16
Programming API	OpenCL	CUDA

2.1 Graphics Processing Units

GPUs are originally designed as co-processors for CPUs to process graphics tasks. In recent years, they have evolved into a powerful accelerator for many applications such as High Performance Computing (HPC) [25] and data processing (such as MapReduce [12] and key-value stores [39]).

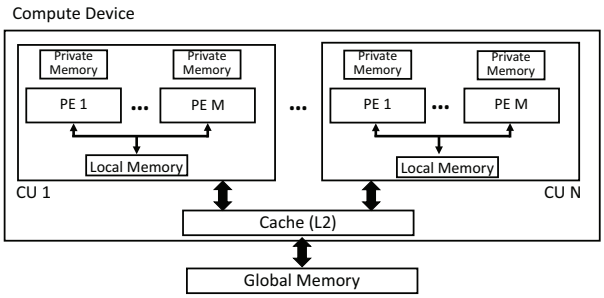


Figure 1: GPU Architecture.

Though the detailed components of GPUs from different vendors vary, their general architectural designs can be abstracted in the same model as shown in Figure 1. For illustration purposes, we use the terminology of AMD GPU and our design and implementation can be extended to NVIDIA GPUs with little modification. A GPU consists of multiple Compute Units (CUs), each of which is further composed of many vector Processing Elements (PEs). A program executed by the GPU is called a *Kernel*. Each PE executes its own instance of a kernel in a SIMD (Single Instruction Multiple Data) manner. Each PE has its own private memory (registers) and all PEs of a CU share a local memory, both of which are on-chip memory. Local memory is dedicated to one CU and invisible to all other CUs. Global memory is accessible to all CUs, and cache is placed between global memory and CUs to reduce the performance gap between PEs and global memory. The global memory for the AMD GPU is the main memory in the system, whereas the global memory for the NVIDIA GPU is within the GPU card.

The programmability of GPUs has been significantly improved by various programming frameworks like DirectCompute, CUDA and OpenCL. For illustration purposes, we use terms in OpenCL unless otherwise specified. A kernel execution consists of multiple work-groups, each of which further consists of multiple work-items. Each work-group is scheduled as the basic unit to one CU. Within a work-group, 64 work-items of that work-group are arranged together for actual execution on AMD GPU (denoted as a *wavefront*).

Recently, new-generation GPUs have been strengthened with various enhancements, including concurrent kernel execution and mechanisms for efficient communication between

kernels. Table 1 shows the features of AMD and NVIDIA GPUs used in our experiments.

Concurrent execution. Early GPUs only support a single kernel execution. That is why most existing GPGPU applications focus on the implementations and optimizations in a single-kernel-based pattern [9, 14, 13]. Recent GPUs from AMD (GCN-based GPUs) have been enabled with concurrent kernel execution capability so that multiple kernels can be executed on the same GPU simultaneously. For example, AMD GPU support concurrent kernel execution with Asynchronous Compute Engines (ACEs) that manages multiple kernels in an interleaved fashion.

Efficient data communication across kernels. Efficient data communication among concurrent kernel execution is supported via *channel*. Vendors can have different terminology and mechanisms for channel. On AMD GPU, pipe is introduced in OpenCL 2.0. The main function of data channel is to pass data between concurrently running kernels without explicit materialization in global memory so that these kernels can communicate with each other in a finer-grained manner.

There are three key parameters for channel, including packet size, number of channels and the total size of data to be passed. Packet size is the basic unit of channel for data transfer between two kernels. The number of channels defines multiple channels that are able to transfer data more efficiently between two kernels. With appropriate channel settings, computations and memory accesses can be overlapped across kernels.

We conduct calibration experiments to have an in-depth understanding on relationship between channel configurations and the throughput. In the experiment, we set up a simple chain of two kernels, *producer* and *consumer*. The producer kernel generates N integers and passes them to the consumer kernel. The channel packet size is set as 16 bytes, which achieves the best efficiency in most scenarios. The results are shown in Figure 2, when N is varied from 512K to 8 million on AMD GPU. We find that both parameters of data channel configurations can affect the throughput significantly. Also, the input data size (N) impacts the throughput. When the input data is small, the channel is not fully utilized. When the input data is too large, the working set size is larger than the data cache. Cache thrashing occurs and the throughput degrades. On the tested AMD GPU, $N = 1$ million is the suitable size setting. Thus, a feasible solution is that, if the input data is larger than the suitable size setting, the input data is chunked with the suitable size setting prior to transferring via the channel. For a general approach across different platforms, we use calibrations on varying the three key parameters, and obtain the suitable parameter values. Particularly, we experimentally obtain the relationship between the three key parameters and the throughput as the input to our cost model to determine the suitable channel configuration.

In the following of the paper, we mainly concentrate our presentation on AMD GPU, and leave the detailed results of NVIDIA GPU to Appendix A.

2.2 Pitfalls of Existing Query Co-Processing

Existing query co-processors [15, 40, 16, 37, 35, 18] mostly rely on kernel-based executions (KBE). A relational operator is implemented with multiple kernels. For example, a selection in GDB [13] can be implemented with 3 kernels

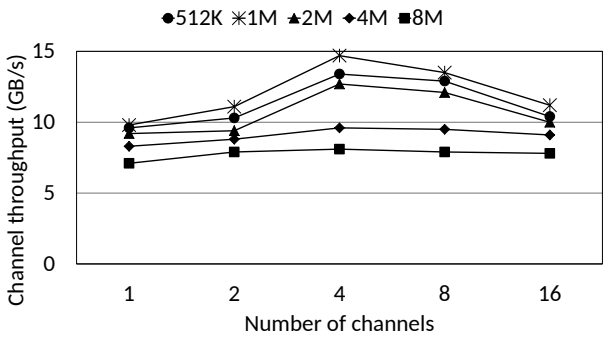


Figure 2: The relationship between channel configurations and throughput on AMD GPU for a packet size of 16 bytes.

including *map*, *prefix sum* and *scatter*. Existing query co-processors optimize the performance of individual kernels with various optimization techniques [14, 15] to exploit the GPU parallelism and to reduce the memory stalls of kernel execution. In KBE, each kernel is then executed on the GPU. Still, we observe serious performance pitfalls when evaluating an operator, even worse for a query. In the following, we summarize our observations, and present the experimental results from TPC-H queries with a scale factor of 10. More details on experimental setup can be found in Section 5.1.

To have a detailed understanding on the performance of KBE, we report various performance counters to comprehensively evaluate GPL on both platforms. Cache hit ratio measures the data locality exposed by data channels. Kernel occupancy is the ratio of in-flight wavefronts executed to the theoretical maximum wavefronts supported on each CU. It is a main indicator of the device utilization. We use the latest profiler from AMD (CodeXL) to obtain the above performance counters.

Observation 1: Explicit materialization of intermediate results across kernels not only generates a large amount of intermediate data, but also causes significant overhead in the communication between kernels. We have similar observations for all the tested queries and use TPC-H Q14 as an example for in-depth discussions. Figure 3 shows the normalized size of intermediate data generated by TPC-H Q14, with selections and joins. We normalize the intermediate data size to the input data size of the query. We also vary the selectivity and compare the results with the default. Specifically, we have changed the interval in the predicate of Q14 (*"l_shipdate >= date [DATE] and l_shipdate < date [DATE] + interval 1 month"*) to generate approximate selectivity ranging from 1% to 100% on $LINEITEM$ relation (16.4% by default). The size of the intermediate data, which mainly include the tuples generated from the selection, and the intermediate results generated from the join (l_partkey = p_partkey), increases with the selectivity. After 75%, the intermediate results even exceed the original data input. Large intermediate results would waste global memory and cause memory access latency.

Such large intermediate results impose significant overhead in the communication between kernels. Figure 4 presents the cost of memory accesses through global memory, where the memory stall (Mem_cost) is directly obtained from hardware profiler. The global memory accesses not only incur high access overhead, but also sacrifice the overlapping opportunity between query co-processing and data trans-

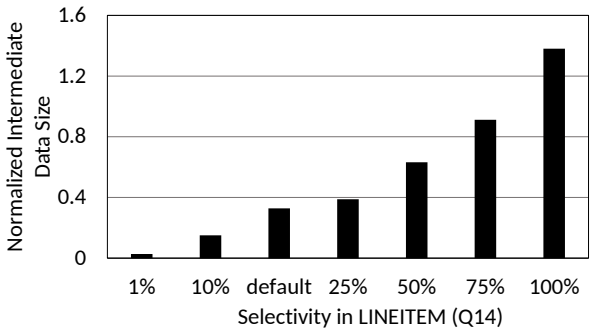


Figure 3: Size of intermediate results in KBE with varying selectivity (Q14).

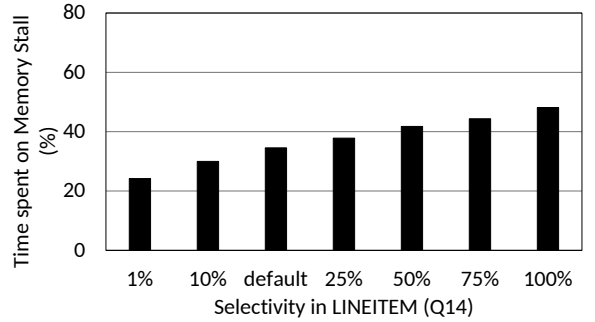


Figure 4: High communication cost in query execution with varying selectivity (Q14) on AMD GPU.

fers. Moreover, such bulky store/load operations are cache-unfriendly, generating more cache misses and degrading the query co-processing performance.

Observation 2: Processing one kernel at a time can lead to severe resource under-utilization of the GPU (e.g., memory bandwidth and computational power). We collect the resource utilization statistics of 5 TPC-H queries (Q5, Q7, Q8, Q9 and Q14) on AMD GPU in KBE approach and the average utilization results are presented in Figure 5. On AMD GPU, we use hardware counters *VALUBusy* and *MemUnitBusy* to evaluate the utilization of vector ALU and memory bandwidth. Also, we observe that the computation and memory utilization vary greatly among kernels. Within a query, some kernels may have high computation utilization but low memory utilization. Some other kernels favor more memory resources than computation resources. Thus, KBE barely can fully utilize both computation and memory resources of the GPU.

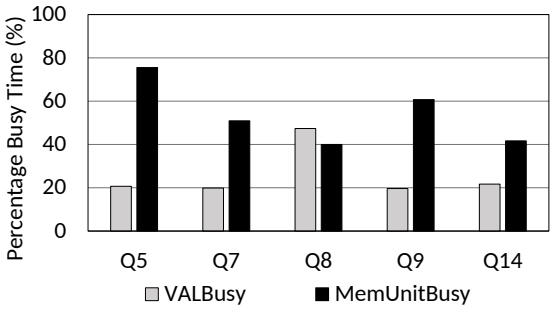


Figure 5: Low utilization of GPU resources in query execution on AMD GPU.

3. SYSTEM DESIGN AND IMPLEMENTATION

We develop a pipelined query processing engine named GPL to reduce inter-kernel communication overhead and improve resource utilization of query co-processing. GPL takes advantage of emerging GPU hardware features including channels and concurrent kernel execution to improve the resource utilization as well as to reduce memory overhead of the KBE approach. We note that, if we implement pipelined query processing on early GPUs with the support of only a single kernel, the intermediate results have to be materialized into the global memory and hence this system will suffer from similar performance bottleneck as KBE (as observed in Section 2).

In this section, we first show the overall architectural design of GPL. Then, we present the implementation of GPL operators. After that, we elaborate each key component of GPL with more details.

3.1 Architectural Design of GPL

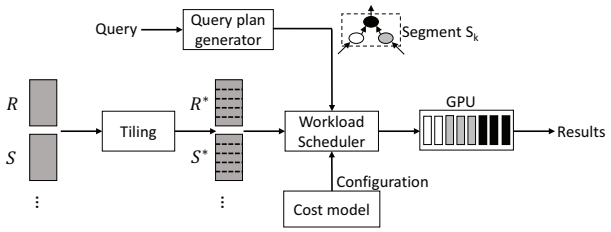


Figure 6: Architectural overview of GPL.

Figure 6 presents the main components of GPL. Like existing pipelined execution engines [41, 4], a query Q is firstly processed by the query plan generator to produce a segmented query plan tree. Specifically, it is processed according to its query plan tree T generated by Selinger-style optimizer [32]. Basically, T consists of a sequence of operators $(O(O_0, O_1, ..., O_i, ..., O_m))$. Such a sequence is produced by traversing T in post-order so that all child nodes precede before their parent nodes. We further denote the kernel sequence as $K(K_0, K_1, ..., K_j, ..., K_n)$, including blocking and non-blocking kernels. Because of the nature of blocking kernels, T is partitioned into a set of segments (S). Here, $S = \{S_0(K_0, ..., \hat{K}_p), ..., S_t(K_0, ..., \hat{K}_q)\}$ where S_i is a segment consisting of multiple kernels (K_0, K_1,...), and $\hat{K}_p$ and $\hat{K}_q$ are blocking kernels. We adopt an existing simple segment generation approach [23]. Each segment contains a sequence of non-blocking kernels, ending by a blocking kernel.

A segment is the basic unit for scheduling and execution in GPL. Each segment contains one or more kernels that are connected with channels, and simultaneously executed in pipelines. The input relations (e.g., R and S) are processed by tiling component to produce tiled relations (denoted as R^* and S^*), which are logically partitioned with nearly the same size. One tile is scheduled by workload scheduler as input to each segment. As we observed in Section 2.1, we need to determine the suitable tile size so that the channel and GPU resources can be fully utilized. Moreover, it also needs to consider the influences from data channel and concurrent kernel execution because they introduce additional cost such as pipelined execution delay. We propose a cost model that captures the characteristics of both queries and hardware.

The configuration produced from cost model can be used to guide the setting of parameter values of GPL. The amount of GPU resources allocated to each kernel is determined by the configuration to minimize the estimated execution time of individual segments. The workload scheduler component schedules the segments to GPU for execution one by one. Note, the output of a segment has to be materialized into the GPU global memory, which will be used as inputs for other segments.

3.2 Implementation of GPL Operators

GPL is based on an OpenCL-based query execution engine, OmniDB [40]. This section focuses on how GPL is developed from OmniDB. For the implementation details of individual operators/primitives, we refer readers to the previous studies [40, 15]. Even though GPL is based on OmniDB, our goal is to address the performance problem of kernel-based execution, which is adopted by most existing studies on GPU-based query co-processing, including OmniDB [40], Ocelot [18], and Red Fox [36]. We conjecture that the GPL design can improve the performance of all those systems. As a start, we have implemented GPL based on OmniDB, and leave the implementation on other systems as our future work.

OmniDB is not designed with pipelined execution capabilities. Our implementation specially maximizes the chances of non-blocking executions. Particularly, we reuse and modify the kernels and primitives in OmniDB to support pipelined execution if appropriate. In the following, we briefly describe the implementation details for selection, aggregation and hash joins. The implementation of other operators follows the same methodology.

Selection: We simplify the selection in OmniDB by removing prefix sum kernel. We only perform the map kernel on each tuple for evaluating the selection predicates, and send the satisfied tuple to the consumer kernel (if available) via channel.

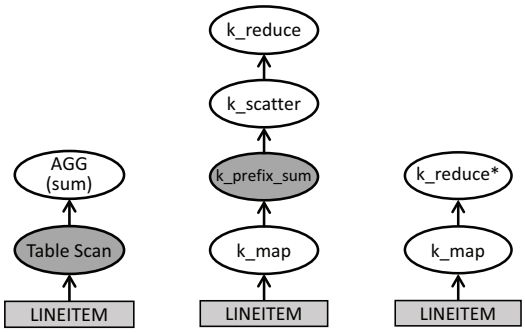
Aggregation: In OmniDB, prefix scan is used to calculate the prefix sum in parallel upon an entire input. It is also used to implement aggregations. In GPL, we use an non-blocking approach by directly performing executions on the intermediate results from the channel packet by packet. Take the sum operator as an example. Upon receiving a packet, the kernel updates the partial sum by adding the values in the packet.

Hash join: We implement the simple hash join with two phases, i.e., hash build and hash probe separately based on existing code base [15] as primitives. Hash build is to build the hash table, and hash probe is to probe the hash table for matches. Both primitives are non-blocking. Still, it requires a blocking barrier after hash build. Partitioned hash joins can be implemented similarly, where the partition phase also can be implemented in a non-blocking manner.

Despite the similarities, the execution model of OmniDB and GPL are inherently different. We compare the query execution plans of both KBE and GPL in Figure 7 for an example query (Listing 1).

```
SELECT SUM(l_extendedprice * (1 - l_discount) * (1 +
    l_tax)) AS sum_charge
FROM LINEITEM
WHERE l_shipdate <= DATE '1988-11-01'
```

Listing 1: An example query.



(a) Logical Plan (b) KBE (c) GPL

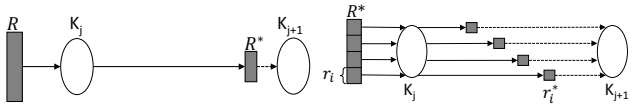
Figure 7: Comparison of query execution plan.

In the query plan, the shadowed ellipses represent the blocking operators and kernels. Blocking operators and kernels need to materialize the intermediate result in global memory. A blocking operator contains at least one blocking kernel. In conventional KBE implementations [15, 16, 13, 40], most operators are implemented in blocking fashion for intra-kernel parallelism. For instance, to filter tuples from $LINEITEM$ that satisfy the predicate in the example query, the prefix sum kernel (k_prefix_sum) is used in KBE. As shown in Figure 7c, all kernels in the query plan are non-blocking, and can be executed concurrently on the GPU. Thus, the concurrent kernel execution can be exploited, rather than executing one kernel at a time in KBE. In GPL, after the k_map function is evaluated on each tuple of the input tile, the satisfied tuples are stored as a packet and passed to k_reduce^* together with $LINEITEM$ via channel for further processing. The reduce kernel (k_reduce^*) directly performs the sum on each element in the packet.

3.3 Tiling

GPL adopts the tile-based pipelined execution model [41, 4]. Figure 8 compares the data storage in KBE (w/o tiling) and GPL, respectively. Without tiling, because KBE executes kernels one by one on the GPU, the GPU is entirely occupied by kernel K_j. Intermediate result R^* is produced and serves as the input to the next kernel K_{j+1}. With tiling in GPL, R is firstly partitioned into much smaller tiles. Tiles are processed one by one, immediately after the previous tile has been completely processed by K_j. The intermediate result r_i^* is passed to the next kernel via global memory (if K_{j+1} is blocking) or data channels (if K_{j+1} non-blocking).

The tile size is an important parameter for the efficiency of pipelined execution. Given a segment, the tile size determines the working set size of performing the pipelined execution as well as the amount of work on the GPU. Thus, on the one hand, small tile sizes cause resource underutilization of pipelined execution and dramatically degrade the data channel efficiency (as we observed in Section 2), and such an impact can be further exaggerated for deep pipelines. On the other hand, large tile sizes result in inefficient communications between kernels due to cache thrashing. Therefore, a carefully selected tile size is important in improving the data channel efficiency.



(a) KBE (b) GPL

Figure 8: Execution model of KBE and GPL.

3.4 Usage of Data Channels

With data channels, the data transfer in the pipeline is more efficient, further leading to fluent pipelined execution and higher GPU resource utilizations. In Figure 7, any arrows pointing to non-blocking nodes represent such data channels. That means, the output generated by the previous node directly passed via data channel to the next node, without materialization in the global memory.

Figure 9 presents the detailed mechanism of data passing between two kernels in the pipelined execution of GPL. For AMD data channel, the space is used on reservation basis by each work-group. Each time after one work-group in K_j has been finished, a synchronization operation is performed so that the next kernel K_{j+1} can fetch the results immediately. This light-weight synchronization guarantees data consistency as well as enables fine-grained coordination between connected kernels. In GPL, read and write operations are executed in the scope of a work-group. Each time a work-group of K_j that has finished its work is able to store its results into data channel, regardless of the progress of other work-groups. Once the corresponding work-group of K_{j+1} has received the data, it can start to execute without synchronization with other work-groups. In this way, GPL is able to establish a more fine-grained coordination between K_j and K_{j+1}. Moreover, data channel can expose more data locality. As the producer work-group stores its results into the channel, the consumer work-group can fetch them immediately. With a suitable tuning on the channel, it is very likely that the data still resides in cache.

In Section 2, we have observed that appropriate parameter settings are important for the throughput of data channels. Based on the cost model, we determine the optimal settings at various positions by capturing the dynamics among kernels (such as selectivity) and differences between hardware.

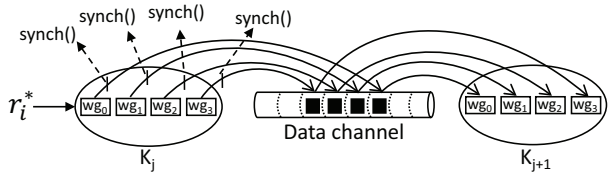
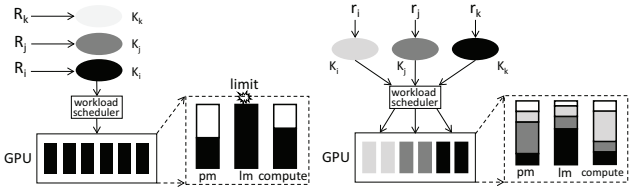


Figure 9: Fine-grained data passing between kernels w/ data channel.

3.5 Concurrent Kernel Execution

With concurrent kernel executions, the stages involved in the pipeline execution of a query segment are executed concurrently. Since stages can have different computation and memory characteristics, the GPU resources can be better utilized. We have designed a new execution paradigm combined with tiling technique to accommodate multiple kernels in a segment simultaneously. Figure 10 illustrates the workload distribution of three kernels K_i, K_j and K_k on the GPU. In KBE without concurrent execution, the workload scheduler distributes workload from one kernel each time



(a) w/o concurrent execution (b) w/ concurrent execution

Figure 10: Workload distribution w/o and w/ concurrent execution ("pm" and "lm" denote private memory and local memory, respectively).

onto all available GPU CUs. A single kernel cannot fully utilize all resources (e.g., private memory, local memory and compute units), as we have already observed in Section 2. If K_i is constrained by certain resource (e.g., local memory), the total active work-groups that can be scheduled for execution simultaneously is limited, leading to underutilization of all other resources (private memory and compute units). In contrast, in GPL the stages involved in the pipeline execution of a query segment are executed concurrently. Since stages can have different computation and memory characteristics, the GPU resources can be better utilized. In this way, each resource can be used to serve more kernels simultaneously, leading to a higher overall resource utilization.

We propose to control the resource allocation indirectly by adapting the work-group size as well as the total number of work-groups. The work-group size is fixed at 64 (wavefront size) to spawn more work-groups to gain scheduling flexibility. We leave choosing the optimal numbers of work-groups of each kernel according to the cost model.

4. ANALYTICAL MODEL

There are a number of parameters introduced in GPL. Inappropriate settings could cause significant performance loss. Without an automatic approach, it is impossible to evaluate and compare the performance of a large number of execution plans and to choose the most efficient execution plan. Therefore, we propose an analytical model to determine the optimal system configurations according to query and hardware information automatically.

4.1 Cost Model

There have been a number of studies on estimating the cost of individual database operators [13, 14, 15, 16]. We integrate existing methods into our cost model for estimations. However, additional factors related to tiling, data channel and GPU resource allocation need to be taken into consideration. The analytical model is designed to automatically determine these factors accurately without sacrificing performance.

We summarize all notations to be used in the cost model in Table 2. Notations are categorized into six groups: platform, profiling, program analysis, query optimizer, model output and calibration. *Platform input* notations are platform specific and can be obtained from its specification. Parameters from *Profiling input* are obtained by profiling tools such as CodeXL and Visual Profiler. Parameters from *program analysis* can be obtained from program analysis on the source code. Parameters from *query optimizer* are determined by query optimizer. Parameters from *calibration* are determined by calibration experiments. Parameters from *model output* are calculated as output in the cost model.

Table 2: Notations in the cost model

Notation	Definitions	Sources
$\#CU$	Number of CUs	platform input
w	Cycles to execute one instruction	
C	Concurrency degree	
mem_l	Global memory latency	
c_l	Data cache latency	
pm_{max}	Amount of available private memory per CU	
lm_{max}	Amount of available local memory per CU	
wg_{max}	Number of work-groups supported per CU	
$a\_wg\_K_i$	Number of active work-groups supported per CU for K_i	profiling input
$a\_CU\_K_i$	Number of active CUs for K_i	
$c\_inst\_K_i$	Number of compute instructions for K_i	
$m\_inst\_K_i$	Number of memory instructions for K_i	
cr_{K_i}	Data cache hit ratio for K_i	
pm_K_i	Private memory usage in each work-item for K_i	program analysis
lm_K_i	Local memory usage in each work-item for K_i	
r_{K_i}	Number of tiles processed by K_i	
set_l	Set of kernels that are leaf nodes in query plan tree	query optimizer
set_b	Set of kernels that have blocking kernels as child nodes	
wi_K_i	Work-group size in K_i	
λ_{K_i}	Ratio of intermediate data generated by K_i to Δ	
Γ	Relationship between data channel throughput and configuration	calibration
Δ	Tile size	model output
T_{K_i}	Execution time of kernel K_i per tile	
T_{S_k}	Execution time of segment S_k	
n	Number of data channels between two kernels	
p	Packet size of data channel (for AMD only)	
wg_K_i	Number of work-groups for K_i	
rep_K_i	Times for a CU to complete K_i	
c_K_i	Computation cycles of K_i	
m_K_i	Memory cycles of K_i	
$delay\_S_k$	Delayed cycles in segment S_k	

In the remainder of this section, we mainly present the model for the AMD GPU and we briefly describe the extensions needed for the NVIDIA GPU in Appendix A.

The data channels between running kernels pass data from one end to the other end in a fine-grained pattern so that the computation and data access latency can be largely overlapped. To quantitatively evaluate the performance of such data channels, we adopt a calibration-based approach to determine the relationship between the data channel throughput and three factors: data size, number of channels, size of packet (for AMD GPU). We find that for a given data size d, the channel throughput changes depending on the number of channels n and p. Thus, we conduct calibration with various data set sizes and data channel configurations to formulate the relationship between them as Γ in Eq. 1.

$$T = \Gamma(n, p, d) \tag{1}$$

Given a kernel K_i, the output data size can be calculated as $\Delta \times \lambda_{K_i}$ which is the amount of data K_i will send to K_{i+1}. λ_{K_i} is a kernel-specific parameter representing the data reduction ratio to the original tile size Δ and is obtained from the database query optimizer. For example, the output data size for a selection kernel depends on the selectivity. Given the output data size ($d = \Delta \times \lambda_{K_i}$), we search the data channel configuration that can maximize the data channel throughput from the relationship Γ. We denote the n_{max} and p_{max} to be the optimal values for n and p, respectively, for that channel setting.

To evaluate the cost of each segment, we categorize the cost into three parts: the computation cost (c_{K_i}), the memory cost (m_{K_i}) and the cost of delay ($delay_{S_k}$) between kernels of that segment due to imbalanced execution speed.

Computation cost: For each kernel executed on the GPU, the scheduling unit is work-group, which is further divided into execution unit wavefronts. As each work-item requests the same amount of resources (private memory and local memory) as well as the computation resources which is determined by the number of CUs that is assigned to that kernel, the total number of active work-groups ($a\_wg\_K_i$) as well as active CUs ($a\_CU\_K_i$) is constrained by the total available resources of each CU and the number of work-groups (wg_K_i) defined by programs. Given a segment S_k, Eq. 2 presents the resource limitations on private memory, local memory and CUs. The private memory (pm_K_i) and local memory (lm_K_i) required by each work-item of K_i can be obtained from analyzing the kernel code via program analysis. Particularly, the program analysis is performed in an off-line manner, with the source code of the kernel as input. The amount of private/local memory usage in each work item can be obtained from existing program analysis tools, such as AMD APP Profiler. This analysis is not restricted to certain query patterns.

$$\begin{cases} \sum^{S_k} pm\_K_i \times wi\_K_i \times wg\_K_i \le pm_{max} \times \#CU \\ \sum^{S_k} lm\_K_i \times wi\_K_i \times wg\_K_i \le lm_{max} \times \#CU \\ \sum^{S_k} wg\_K_i \le wg_{max} \times \#CU \end{cases} \tag{2}$$

Under the condition in Eq. 2, the times (req_K_i) to complete all work-groups of K_i on the GPU is obtained in Eq. 3.

$$req\_K_i = \frac{wg\_K_i}{a\_wg\_K_i \times a\_CU\_K_i} \tag{3}$$

Thus, the total computation cost for kernel K_i on each CU is calculated based on the profiled statistics on the number of computation and memory instructions in Eq. 4. w is a platform-specific parameter to represent the number of cycles to issue and execute one instruction. In our experiments, w is 4 for both AMD and NVIDIA GPU. Due to the resource limitations, K_i can be finished in req_K_i times as in Eq. 3.

$$c\_K_i = (c\_inst\_K_i + m\_inst\_K_i) \times w \times req\_K_i \tag{4}$$

Memory cost: The memory cost consists of global memory access cost and data channel access cost. For those kernels in set_l, they suffer from long access latency as they have to fetch data from data tiles initially stored in global memory. Besides, given a blocking kernel K_i, its parent kernel K_j is a member of set_b. All kernels in set_b also suffer from global memory latency as their directly preceding blocking kernel materializes the intermediate results in the global memory. Thus, the memory cost for kernels in set_l and set_b is derived in Eq. 5. It consists of two parts: the cost of global memory accesses and the cost of L2 cache accesses. The number of memory accesses are calculated by cache hit ratio

(cr_{K_i}) and the total number of memory instruction derived from profiler.

$$\begin{aligned} m\_K_i = & m\_inst\_K_i \times (1 - cr_{K_i}) \times mem\_l + \\ & m\_inst\_K_i \times cr_{K_i} \times c\_l, \forall K_i \in set\_l \cup set\_b \end{aligned} \quad (5)$$

For the remaining kernels in the kernel set, the memory cost is from data channel access overhead. As the data to be passed to K_i over data channel is known ($\Delta \times \lambda(K_i)$), the memory cost for these kernels can be obtained as Eq. 6 based on Eq. 1.

$$m\_K_i = \frac{\Delta \times \lambda_{K_i}}{\Gamma(n_{max}, p_{max}, \Delta \times \lambda_{K_i})}, \forall K_i \in (K - set\_l \cup set\_b) \quad (6)$$

Therefore, the expected execution time for K_i is derived in Eq. 7 as the sum of computation time and memory access time.

$$T_{K_i} = c\_K_i + m\_K_i \quad (7)$$

Delay cost: Delay may be incurred between kernels as they form producer-consumer chain to achieve pipelined execution. The delayed cycles of the pipelined execution need to be minimized to maximize the efficiency of each segment. Eq. 8 calculates the total delayed cycles incurred within a segment S_k by calculating the absolute value of the difference between two connected kernels K_i and K_j. The actual execution time of segment S_k is shown in Eq. 9. Because of concurrent execution, the execution time of a segment is adjusted by $\frac{1}{C}$ where C is the concurrency degree of given platform (2 for the AMD GPU used in our study).

$$delay\_S_k = \sum^{S_k} abs(T_{K_i} * r_{K_i} - T_{K_j} * r_{K_j}) \quad (8)$$

$$T_{S_k} = \frac{1}{C} \times \sum^{S_k} T_{K_i} + delay\_S_k \quad (9)$$

The objectives of our analytical model is to minimize the total execution time of all kernels within a segment S_k. To minimize T_{S_k}, we extensively explore the search space by adapting three parameters, Δ, n, p and wg_K_i.

We have tried all means to reduce the solution space. In our query optimization, the solution space is for individual query segments. Moreover, for finding the optimal configuration of each query segment, we have used many effective approaches to reduce the solution space without significantly scarifying the solution quality. For each parameter, we search the value within the feasible range. For example, we find that the throughput of data channels continues to drop when the number of channels is over 16. Thus, n can be selected between 1 and 16. The wg_K_i represents the number of work-groups for each kernel. As work-groups can be scheduled to any CUs, we set wg_K_i integral multiple of $\#CU$. In this way, the search space and the overhead associated with query optimization can be dramatically reduced. In our experiments, the elapsed time for query optimization is generally smaller than 5ms, which is ignorable compared with the query processing time. With each set of these parameters, c_{K_i}, m_{K_i}, T_{K_i} and T_{S_k} also can be obtained accordingly. We determine the smallest T_{S_k} as the optimal pipelined execution plan.

4.2 An Example

We use an example query (Figure 7) to show pipelined plan generation on AMD GPU with our model.

The analytical model is applied to evaluate on a query segment. Apparently, *k_map* and *k_reduce** are in the same segment S_k as both of them are non-blocking kernels.

Next, by program analysis we can obtain statistics about the requirements for each type of resources, e.g., the local memory and private memory usage. All these statistics and parameter values need to satisfy conditions in Eq. 2.

After that, both kernels are launched together and *k_map* works on the first tile of $LINEITEM$. The tile size here is a variable but initiated with a default value. Once the first tile has been processed, CodeXL is used to collect runtime information such as $active\_wg\_K_i$, number of computation and memory instruction and cache hit cr_{K_i}. With these information, theoretical computation and memory access time can be calculated according to Eq. 3 to 6, depending on the platform we use. On AMD GPU, the data channel cost is determined by the size of data to be passed as well as the data channel configurations. Thus, estimated execution time of each kernel is a function of the number of work-groups for each kernel, tile size and data channel configurations.

Another cost is delay cost between two connected kernels if there is a difference between the estimated execution time of them. As in Eq. 9, T_{S_k} is a function of wg_K_i, Δ and data channel configurations. As the feasible range of these parameters has been constrained, the optimal parameter settings can be found through searching the solution space.

5. EXPERIMENTAL EVALUATIONS

We experiment with TPC-H queries to evaluate the efficiency and effectiveness of GPL. Our experiments have been conducted on AMD and NVIDIA GPUs. Furthermore, to give more in-depth understanding on the performance improvement, we further investigate hardware performance counters on cache, memory units and device utilization.

5.1 Experimental Setup

Hardware. We evaluate GPL on both AMD and NVIDIA GPU. Overall, we have similar findings on both GPUs. Here, we focus our discussions on the AMD GPU, and the detailed results on the NVIDIA GPU is presented in Appendix A. The AMD platform is a coupled CPU-GPU architecture (AMD A10 APU) in which the CPU and the GPU are integrated on the same die without PCI-e bus. More details about the processor specification of the AMD and NVIDIA GPU can be found in Table 1.

Workloads. We evaluate GPL on TPC-H queries (Q5, Q7, Q8, Q9, Q14) with the scale factor (SF) of 10 (by default). Detailed description of these TPC-H queries is given in Appendix B. The total size of the tables generated by the standard dbgen utility is around 10GB. Q14 has a single join operation, whereas other queries have multiple joins. Q8 and Q9 involve rather complicated subqueries, and others do not have. Those queries also have other common operators such as group-by and order-by.

The KBE approach [15, 16] is the baseline to compare with GPL. To have an in-depth understanding of the performance of pipeline executions, we further compare the performance of a GPL variant, namely GPL (w/o CE), by disabling concurrent execution and data channel in GPL. Thus, each tile is executed by corresponding kernels and the intermediate results have to be stored back to the global memory.

To show that the performance of GPL is comparable to state-of-the-art systems, we further compare GPL against

Ocelot [18]. Ocelot is a hardware oblivious extension for MonetDB [19], which is an in-memory column store database system. It replaces MonetDB's relational operators with OpenCL based operators which can run on a wide variety of devices that supports OpenCL. Ocelot supports all major relational operators (selection, projection, grouping and aggregation). However, in the current code base [31] it does not support data-types greater than 4 bytes in size or operators like multi-column sort. It also cannot support non-trivial string operations like inequality and sorting etc. Following the previous study [18], we have to slightly modify the TPC-H queries (as in Appendix B).

5.2 Model Evaluation

To validate the effectiveness of our cost model, we show the relative error associated with estimating the GPL runtime by our analytical model. The relative error is defined in Eq. 10, where $T_{measured}$ is the measured execution time for the query and $T_{estimated}$ is the estimated execution time by our analytical model for the same query.

$$relative\_error = \frac{|T_{measured} - T_{estimated}|}{T_{measured}} \quad (10)$$

The relative errors associated with performance estimation of all tested queries with the optimal configuration on AMD GPU are shown in Figure 11. As the results show, our cost model can predict relatively accurate system configurations for GPL. Specifically, the cost model is able to produce suggested values for parameters including tile size and number of work-groups for each kernel. Note, our prediction model generally underestimates the execution time. One possible reason is that, when we consider the concurrency degree of given platform in Eq. 9, we assume ideal parallelism, potentially leading to under-estimation.

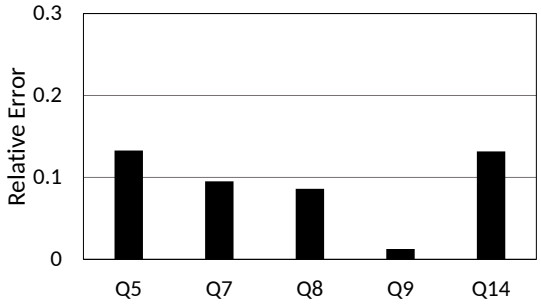


Figure 11: Relative error in estimating GPL runtime for TPC-H queries.

We take Q8 as an example to illustrate the effectiveness of our cost-based approach in determining the suitable parameter configuration. Similar results can be obtained from other queries. We study the detailed results on AMD GPU: 1) the relationship between tile sizes and overall query processing performance; 2) the relationship between resource allocations (number of work-groups assigned to each kernel) and delay cost.

Figure 12 presents the results of overall query processing performance when we vary the tile size from 256KB to 16MB while the other parameters are fixed to their default values. For better presentation, we normalize the execution time of all settings to that of 256KB. The star shows the optimal tile size estimated by the model. The results show that if the tile size is too small (resp. too large), the performance can be degraded due to inefficient resource utilization (resp. cache thrashing). Figure 13 shows that our cost model is able to capture this trend with very small relative errors and is able to estimate both the elapsed time and the optimal tile size accurately. The derived optimal tile size is 4MB, which is greatly different from the default size (1MB). The first query segment contains three kernels (2 *map* kernels K_0 and K_1, and 1 *hashbuild* kernel K_2). The optimal numbers of data channels between two *map* kernels and 1 *hashbuild* kernel are 4 and 2, respectively. Our cost model is able to capture the hardware characteristics as well as query statistics to give an accurate estimation.

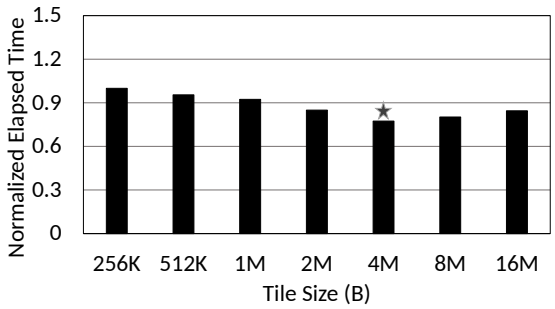


Figure 12: The overall query processing performance with varying tile sizes (Q8).

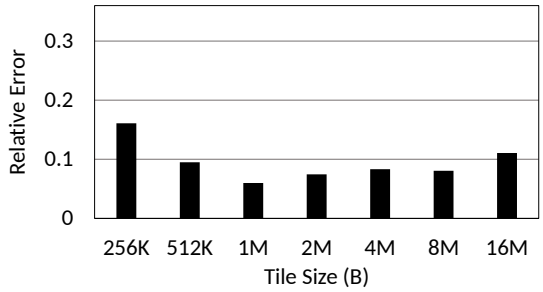


Figure 13: Relative error in estimating GPL runtime with varying tile sizes (Q8).

We further study the impact of the number of work-groups. We select a set of work-group settings (from S_1 to S_7). For each S_i (i $\in$ [1, 7]), where the number of work-groups assigned to each kernel is 2^{i-1} times of S_1. We set S_1 to be 2 for AMD GPU. Figure 14 shows the relative error in estimating the GPL runtime and Figure 15 shows the impacts of resource allocation imbalance on the overall query processing performance in terms of delay cost (normalized to the delay cost of S_1). The optimal settings derived by our cost model is S_4 (denoted by the star) for Q8 on AMD GPU, which is the setting with lowest delay overhead (Figure 15). The minimal delay overhead also corresponds to the lowest total query processing time. Note that the optimal setting varies for different queries. From the results, we find that our model is able to estimate the runtime of GPL with nominal error even with a varying number of work-groups. Also it can be seen that inappropriate resource allocations may result in serious delay overhead, especially in pipelined execution paradigm where any delay incurred along the pipeline can be exaggerated in the subsequent stages. Our model has captured this delay cost of pipelined execution. Thus, it is

able to produce the optimal resource allocation for a given query that incurs minimum delay cost.

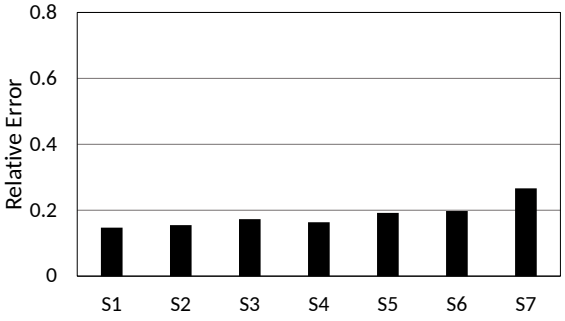


Figure 14: Relative error in estimating GPL runtime with varying number of work-groups (Q8).

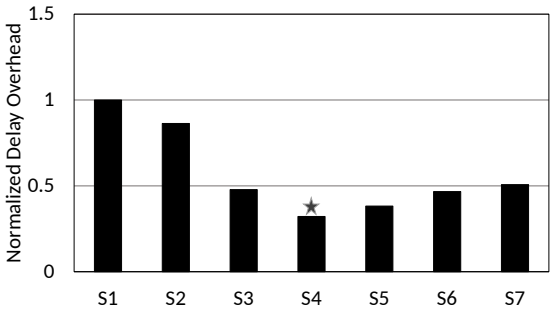


Figure 15: The delay cost with varying resource allocations (Q8).

5.3 Evaluation on GPL

5.3.1 Overall Comparison

Figure 16 shows the results for the TPC-H queries on AMD GPU. If tiling is applied without concurrent execution (GPL (w/o CE)), queries on AMD GPU suffer from performance degradation up to 31% on AMD. On the one hand, in the variant without concurrent execution, tiles are executed one by one as that in KBE. Thus, additional overhead is incurred by frequent kernel launches and large amount of intermediate data materialization. On the other hand, each time only one tile is processed by the GPU. Thus, insufficient data level parallelism would underutilize the GPU resources.

In GPL, concurrent execution as well as tiling are both applied together. By comparing query execution performance in KBE and GPL, we find that GPL outperforms KBE by over 48% on AMD GPU. Though applying only tiling would incur non-negligible overhead to query execution, combining tiling with data channels as well as concurrent execution can offset those overhead and even introduce more performance improvements over KBE approach. The main reasons include (1) the overhead of materialization of large intermediate results has been eliminated; (2) various types of kernels are executed concurrently. Thus, different types of GPU resources can be more efficiently utilized. It is even higher in NVIDIA GPU as it supports more concurrency.

5.3.2 Settlement of KBE Pitfalls

Pitfalls existing in KBE approach (Section 2.2) have negative impacts on the query execution performance. To understand the performance improvement as shown above, we

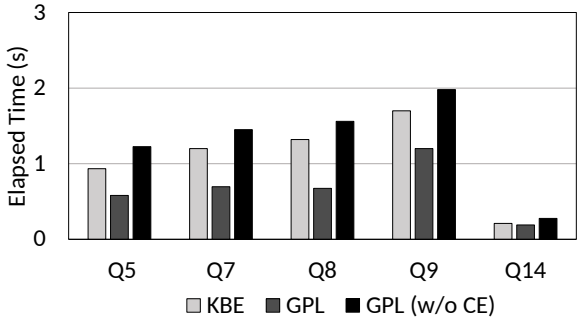


Figure 16: Comparison between KBE and GPL on AMD.

re-visit those pitfalls by examining them on both KBE and GPL designs.

Figure 17 shows the normalized size of intermediate results materialized in global memory in GPL. We normalize the results to the intermediate data size of KBE. In GPL, most intermediate results are passed through data channel in a fine-grained pattern so that the overhead can be overlapped with computation in concurrent kernel executions. Thus, intermediate results only need to be materialized in the global memory by blocking kernels like sort and hash build in GPL. However, all intermediate results in KBE need to be materialized in the global memory, resulting in high materialization overhead. The sizes of intermediate results materialized in the global memory in GPL are 15–33% of that of KBE.

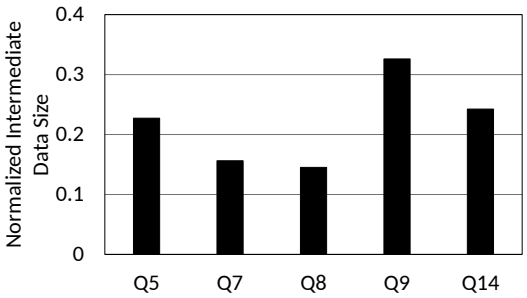


Figure 17: Reduced intermediate results materialized in the global memory (normalized to KBE).

We take Q14 as an example to demonstrate the reduction in the size of intermediate data materialized in the global memory in GPL, with varying selectivity ranging from 1% to 100%. The results are presented in Figure 18. Because of blocking kernels, intermediate data materialized in the global memory cannot be completely eliminated. However, the total size has been significantly reduced by data channel. When the selectivity is increased to 100%, the size of materialized intermediate results comes down from 1.38 times that of original input data in KBE to just 0.22 times that of the original input data in GPL. Therefore, more memory space can be saved and the overhead of global memory accesses is also reduced.

Figure 19 demonstrates the average resource utilization for various TPC-H queries on AMD GPU. Compared with KBE, GPL sustains steady and high resource utilization for memory and computation, because of finer-grained workload scheduling. Besides, the reduced memory stalls increases the efficiency of both memory and computation resources.

To examine the communication overhead among kernels,

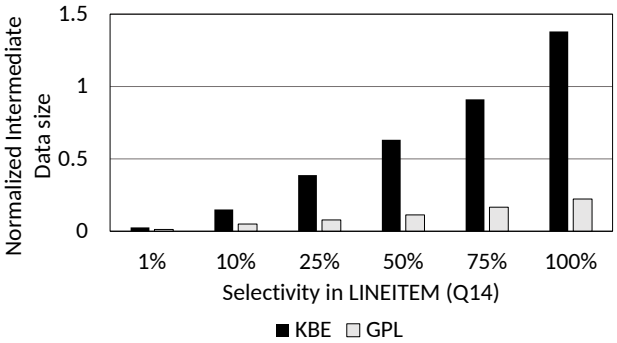


Figure 18: Size of intermediate results in GPL with varying selectivity (Q14).

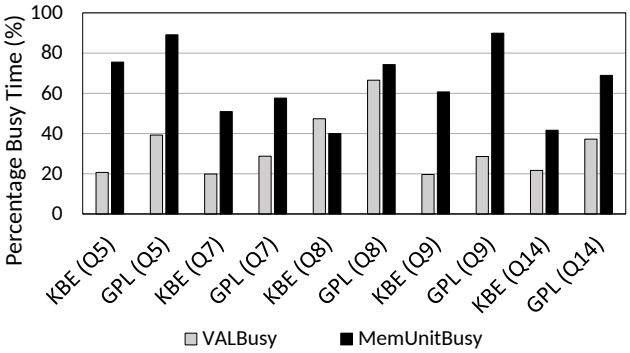


Figure 19: Improved GPU resource utilization on AMD GPU.

we show the query execution time breakdown with Q8 on AMD GPU in Figure 20. We have observed similar results for other queries. For GPL, two additional cost, the data channel access cost (*DC_cost*) and *Delay* cost existing in pipelines, are included into total query execution time. Thus, in GPL, the sum of memory cost, data channel cost and delay cost can be treated as the communication cost. The percentage of communication cost in GPL is up to 14% of the total execution time. However, it can be as high as 34% of the total execution time in KBE. On the other hand, GPL has better cache locality, resulting in an increase of 27% in cache hit ratio when compared to KBE (figures are omitted).

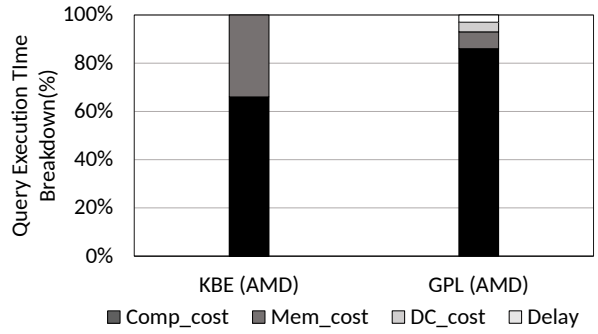


Figure 20: Query execution time breakdown (Q8) on AMD GPU.

5.4 Evaluation with Varying Data Sizes

We vary SF from 0.1 to 10, generating input data ranging from 100MB to 10GB. As the results show in Figure 21, when the data size increases, the performance improvement of GPL over KBE continues to increase on the AMD GPU. Moreover, the elapsed time increases of query execution in KBE are higher than those of GPL. As shown in Figure 10a, the KBE approach may be limited by a certain type of resource. Thus, simply increasing the data level parallelism cannot further improve the query execution performance in KBE. On the contrary, GPL leverages the increased data level parallelism and exploits the hardware strengths to overlap data accesses and computation for better performance.

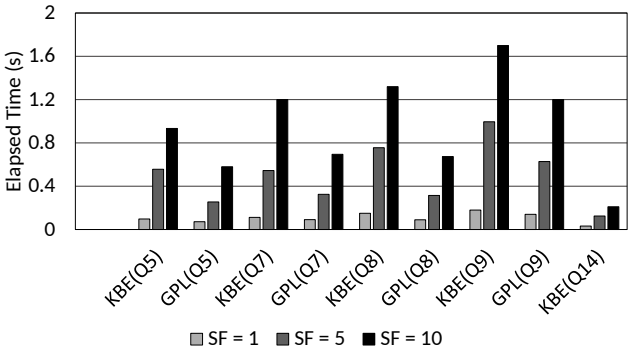


Figure 21: Query execution time on varying data size.

5.5 Performance Comparison with Ocelot

The comparison with Ocelot [18] was made for scale factors of 1, 5 and 10. Note, we cannot get the results for Q9 on Ocelot when the scale factor is 10. Figure 22 shows that, for most of the queries in our evaluation, GPL is comparable to or superior to Ocelot. For Q8 and Q9, GPL significantly outperforms Ocelot. GPL and Ocelot have comparable performance for other queries.

It would be a bit unfair to compare absolute execution time of Ocelot to that of GPL. On the one hand, since Ocelot is built on top of MonetDB, it can utilize optimizations like pre-fetching, data compression etc which are already implemented in MonetDB. Other major optimizations that are available in Ocelot include: 1) *Bitmaps*. The intermediate result of the selection operator is passed to the next operator as a bitmap in Ocelot. This helps Ocelot in reducing memory transactions when compared to GPL (the intermediate results are passed as integer arrays). 2) *Caching of hash tables*. Since generating hash tables is very costly, Ocelot's memroy manager keeps a cache of previously generated hash tables by saving the memory allocation time. So far, GPL does not have the above mentioned optimizations in Ocelot. On the other hand, GPL has more advanced pipelined query processing scheme, whereas Ocelot is still based on KBE. A more detailed study on how to extend our system with those advanced system features of Ocelot is our future work. Another future direction is to integrate GPL into Ocelot. Nevertheless, this comparison was made to show that the performance of GPL is comparable to, or even better than the state-of-the-art systems.

6. RELATED WORK

In-Memory Databases on CPUs. Larger main memory sizes drive database systems from disk-based design to in-memory design [34, 38, 19, 21]. As outlined in recent surveys [34, 38], in-memory databases have many challenges

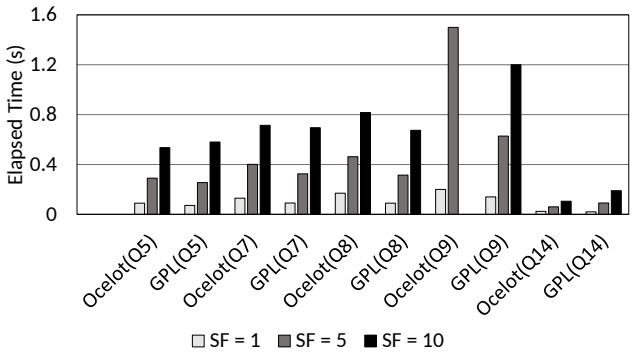


Figure 22: Query Execution Time for GPL and Ocelot on AMD GPU.

and opportunities for database research. We briefly review the related work relevant to this study, and refer the readers to those surveys for more details. Larger and faster caches reduce the performance gap between CPU and memory. A lot of cache-optimized algorithms (e.g., [1, 33, 24, 3, 7, 27]) have been developed to improve the cache locality. In our study, GPL takes advantage of channels to reduce the memory stalls. On the other hand, the emerging trend of multi-core and many-core processors enlarges the design space of in-memory databases with new optimization knobs such as data locality [8, 22] and SIMD [20, 29]. GPL is specifically designed for new-generation GPUs with the support of channels and concurrent kernel executions.

Pipelined Query Processing. Pipelined query processing has been extensively studied in relational database systems. Volcano [10] is a classic model to achieve pipelined execution paradigm in a tuple-at-a-time style. The simple execution pattern fails to utilize the CPU parallelism. Other later systems such as X100 [4] and Vectorwise [41] introduce vector processing capability to exploit the CPU parallelism. More recently, pipelined query processing has been explored on multi-core CPU [6] and NUMA architectures [22]. Particularly, Leis et al. proposes a morsel-driven query execution engine by dividing the input data into "morsels" as the basic scheduling unit and assigning them to each "worker" to exploit the thread parallelism. Our pipelined execution is similar to morsels and X100. Our work differs from theirs in two major aspects. First, the pipelined execution of GPL relies on kernels/primitives explicitly connected with channels, rather than a basic pipeline of operators. Second, GPL takes advantage of GPU hardware features, while their study does not. On the other hand, their study [22] targets at multi-core and NUMA systems and most techniques cannot be adopted into GPL. For example, preemption adopted in their study is not supported at current GPUs.

With the growing demands for concurrent query processing, identifying the opportunity of pipelining parallelism among many concurrent queries becomes an effective means to improve the query processing throughput. Harizopoulos et al. [11] designed a novel operator-centric relational query processing engine QPipe that can maximize the data and workload sharing among queries. Pandis et al. [26] designed DORA to decompose each transaction into smaller units and each unit is assigned to cores based on data locality. Both studies [11] and [26] try to achieve higher performance on CPUs by improving data sharing and data locality. With similar spirit on pipelined execution, GPL is specifically designed with the GPU hardware features in mind, to achieve higher resource utilization on GPUs. Giceva et al. [8] proposed a novel deployment algorithm to multi-core systems, by considering the behaviors of individual database operators and dataflow information. DataPath [2] introduces a novel push based data centric approach for CPUs. Different from previous studies, this paper advocates efficient pipelined execution on the GPU by taking advantage of emerging GPU features.

Query Co-Processing on GPUs. Query co-processing on GPUs has attracted much research attentions in recent years. He et al. [14, 13] conduct systematic studies on the implementations and optimizations of a query co-processing engine on GPUs. The same group further extends the study to coupled CPU-GPU architectures such as AMD Fusion [15, 16]. Yuan et al. [37] conducted a comprehensive study of complex database queries with different software optimizations and hardware configurations. A follow-up paper from the authors [35] proposed a prototype system to share GPU resources among multiple queries to improve system throughput. Recently, Pirk et al. [28] proposed a novel strategy to exploit GPUs in query co-processing. In their proposal, approximate results are first calculated on the GPU, and then the intermediate results are further refined on the CPU. Heimel et al. [18] developed a database engine based on a set of hardware-oblivious operators written in OpenCL to achieve cross-platform capability with minimal performance loss. Ocelot is built on top of MonetDB [19], which is an in-memory column based database. However, it does not support pipelined executions on the GPU. Jason et al. [30] studied a set of database primitives on the integrated GPU of a coupled architecture and show that transparent access to CPU virtual addresses and very low overhead of computation offloading are important for query co-processing performance. Heimel et al. [17] proposed a accelerated solution to quickly and accurately estimate the selectivity of multi-dimensional predicates. All the previous studies use the kernel-based execution which has performance pitfalls, as shown in Section 2. To the best of our knowledge, GPL is the first pipelined query execution engine on GPUs.

7. CONCLUSIONS

In this paper, we propose a novel GPU-based pipelined query execution engine (GPL) on GPUs to address low device utilization problem of existing query processing engines on GPUs. Specifically, we leverage emerging capabilities of GPUs, including channels and concurrent kernel execution, for the efficiency of pipelined query execution. The data channel is placed between two neighbouring kernels to reduce the data transfer overhead and enables a finer-grained concurrent kernel executions. We further propose an analytical model to determine the optimal system configurations such as tile size and resource allocation. We have conducted the experiments with TPC-H queries on both AMD and NVIDIA GPUs. The results show that GPL can significantly improve the query processing performance over the state-of-the-art kernel-based approaches, with improvement up to 48% on the AMD GPU.

Acknowledgement

This work is in part supported by Singapore Ministry of Education Academic Research Fund Tier 2 under Grant MOE2012-T2-2-067.

8. REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbmss on a modern processor: Where does time go? In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 266–277, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[2] S. Arumugam, A. Dobra, C. M. Jermaine, N. Pansare, and L. Perez. The datapath system: A data-centric analytic processing engine for large data warehouses. In *SIGMOD*, 2010.

[3] C. Balkesen, J. Teubner, G. Alonso, and M. T. ÃŰzsu. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 362–373, April 2013.

[4] P. A. Boncz, M. Zukowski, and N. Nes. Monetdb/x100: Hyper-pipelining query execution. *Conference on Innovative Data Systems Research (CIDR)*, 2005.

[5] Z. Chen, J. Xu, J. Tang, K. Kwiat, and C. Kamhoua. G-storm: GPU-enabled high-throughput online data processing in storm. In *Big Data (Big Data), 2015 IEEE International Conference on*, pages 307–312, Oct 2015.

[6] Y. Cheng and F. Rusu. Parallel in-situ data processing with speculative loading. In *SIGMOD*. ACM, 2014.

[7] J. Cieslewicz, W. Mee, and K. A. Ross. Cache-conscious buffering for database operators with state. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, New York, NY, USA, 2009.

[8] J. Giceva, G. Alonso, T. Roscoe, and T. Harris. Deployment of query plans on multicores. *Proc. VLDB Endow.*, 8(3):233–244, Nov. 2014.

[9] N. K. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli. High performance discrete fourier transforms on graphics processors. In *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, SC '08, Piscataway, NJ, USA, 2008.

[10] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 1994.

[11] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. QPipe: A simultaneously pipelined relational query engine. In *SIGMOD*, 2005.

[12] B. He, W. Fang, Q. Luo, N. K. Govindaraju, and T. Wang. Mars: A mapreduce framework on graphics processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[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] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[15] 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.

[16] 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.

[17] 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, New York, NY, USA, 2015.

[18] 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.

[19] S. Idreos, F. Groffen, N. Nes, S. Manegold, K. S. Mullender, and M. L. Kersten. Monetdb: Two decades of research in column-oriented database architectures. *IEEE Data Engineering Bulletin*, 35(1), 2012.

[20] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. Improving main memory hash joins on intel xeon phi processors: An experimental approach. *Proc. VLDB Endow.*, 8(6):642–653, Feb. 2015.

[21] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi. H-Store: A high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.*, 1(2), Aug. 2008.

[22] V. Leis, P. Boncz, A. Kemper, and T. Neumann. Morsel-driven parallelism: A numa-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 743–754, New York, NY, USA, 2014. ACM.

[23] G. Luo, J. F. Naughton, C. J. Ellmann, and M. W. Watzke. Toward a progress indicator for database queries. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, New York, NY, USA, 2004. ACM.

[24] S. Manegold, P. A. Boncz, and M. L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3), Dec. 2000.

[25] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. KrÃijger, A. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 2007.

[26] I. Pandis, R. Johnson, N. Hardavellas, and A. Ailamaki. Data-oriented transaction execution. *Proc. VLDB Endow.*, 3(1-2), Sept. 2010.

[27] H. Pirk, F. Funke, M. Grund, T. Neumann, U. Leser, S. Manegold, A. Kemper, and M. Kersten. CPU and cache efficient management of memory-resident databases. In *Data Engineering (ICDE), 2013 IEEE 29th International Conference on*, pages 14–25, April 2013.

[28] H. Pirk, S. Manegold, and M. Kersten. Waste not... efficient co-processing of relational data. In *2014 IEEE 30th International Conference on Data Engineering*, March 2014.

[29] O. Polychroniou, A. Raghavan, and K. A. Ross. Rethinking simd vectorization for in-memory databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, pages 1493–1508, New York, NY, USA, 2015. ACM.

[30] J. Power, Y. Li, M. D. Hill, J. M. Patel, and D. A. Wood. Toward GPUs being mainstream in analytic processing: An initial argument using simple scan-aggregate queries. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, 2015.

[31] M. Saecker. Ocelot: A Hardware-Oblivious Database Engine. https://bitbucket.org/msaecker/monetdb-opencl.

[32] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*, SIGMOD '79, 1979.

[33] A. Shatdal, C. Kant, and J. F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[34] K.-L. Tan, Q. Cai, B. C. Ooi, W.-F. Wong, C. Yao, and H. Zhang. In-memory databases: Challenges and opportunities from software and hardware perspectives. *SIGMOD Rec.*, Aug. 2015.

[35] K. Wang, K. Zhang, Y. Yuan, S. Ma, R. Lee, X. Ding, and X. Zhang. Concurrent analytical query processing with GPUs. *Proc. VLDB Endow.*, July 2014.

[36] 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.

[37] 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.

[38] H. Zhang, G. Chen, B. C. Ooi, K. L. Tan, and M. Zhang. In-memory big data management and processing: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 27(7):1920–1948, July 2015.

[39] K. Zhang, K. Wang, Y. Yuan, L. Guo, R. Lee, and X. Zhang. Mega-kv: A case for GPUs to maximize the throughput of in-memory key-value stores. *Proc. VLDB Endow.*, 8(11):1226–1237, July 2015.

[40] S. Zhang, J. He, B. He, and M. Lu. OmniDB: Towards portable and efficient query processing on parallel CPU/GPU architectures. *Proc. VLDB Endow.*, Aug. 2013.

[41] M. Zukowski, M. van de Wiel, and P. Boncz. Vectorwise: A vectorized analytical dbms. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 1349–1350, April 2012.

APPENDIX

A. GPL ON NVIDIA GPU

As in case of the AMD GPU, the latest GPUs from NVIDIA (Fermi or Kepler architectures) have also been enabled with concurrent kernel execution capability so that multiple kernels can be executed on the same GPU simultaneously. They also have the ability to pass data directely from one kernel to another via Direct Data Transfer (DDT) [5]. All the techiques used by GPL on AMD GPU are applicable to NVIDIA GPU as well. To demonstrate that GPL works on platforms other than the AMD GPU mentioned in Section 5.1 we evaluate the performance of GPL on a NVIDIA GPU (Tesla K40) which is connected to the CPU with PCI-e 3.0 and has 12GB device memory. The data is initially loaded to the GPU and the cost of PCI-e data transfer overhead is omitted. The experimental results show that GPL is able to outperform existing kernel-based query processing approaches with performance improvement up to 47% on the NVIDIA GPU.

In Section A.1, we describe the relationship between channel configuration and throughput on NVIDIA GPU. Section A.2 describes how our analytical model (Section 4) can be adapted to NVIDIA GPU. Finally, we present our experimental results on NVIDIA GPU in Section A.3.

A.1 Kernel Communication on NVIDIA GPU

As mentioned in Section 2.1, calibration experiments were conducted to have an in-depth understanding on relationship between channel configurations and the throughput on the Telsa K40 GPU. Unlike the AMD GPU, the NVIDIA GPU do not need users to set the packet size. The results are shown in Figure 23, where N is varied from 512K to 8 million on the NVIDIA GPU. As seen in Section 2.1, we find that both parameters of kernel communication configurations can affect the throughput significantly.

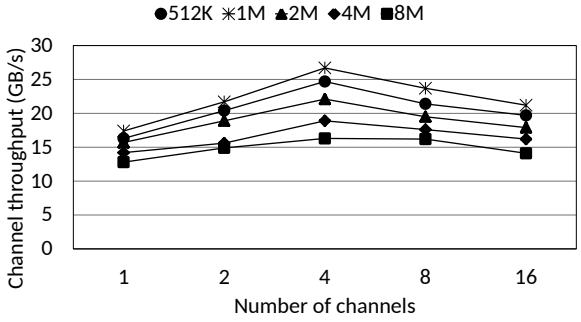


Figure 23: The relationship between kernel communication configurations and throughput on K40 GPU for a packet size of 16 bytes.

A.2 Model for NVIDIA GPU

The model described in Section 4.1 can be extended to NVIDIA GPU easily. The amount of private/local memory usage of each work item in NVIDIA GPU is obtained using the NVIDIA CUDA Occupancy Calculator. To adapt the cost model to NVIDIA GPU, only the data channel-related equations need to be modified accordingly. Specifically, the only parameters that can affect the throughput in of NVIDIA GPU are: (1) number of channels, (2) data size

to be passed. Therefore as seen in Section 4.1 the relationship between them can be formulated as Eq. 11.

$$T = \Gamma(n, d) \tag{11}$$

The global memory access overhead in Eq. 6 is also adapted accordingly based on the adapted channel throughput.

$$m\_K_i = \frac{\Delta \times \lambda_{K_i}}{\Gamma(n_{max}, \Delta \times \lambda_{K_i})}, \forall K_i \in (K - set\_l \cup set\_b) \tag{12}$$

A.3 Experimental Evaluation

We experiment with TPC-H queries to evaluate the efficiency and effectiveness of GPL and our analytical model. We use the NVIDIA Visual Profiler to obtain values of the hardware counters on the NVIDIA GPU.

A.3.1 Model Evaluation

Figure 24 shows the *relative error* (defined in Section 5.2) made by the model for all tested queries with the optimal configuration. The relative error in the execution time estimation done by the model is very small for NVIDIA GPU as well. Taking Q8 as an example, we present the results for 1) the relationship between tile sizes and overall query processing performance; 2) the relationship between resource allocations (number of work-groups assigned to each kernel) and delay cost, on NVIDIA GPU. Figure 25 shows the relative error in estimating the optimal tile size and Figure 26 shows the change in runtime when we vary the tile size from 256KB to 16MB while the other parameters are fixed to their default values. These results show that our cost model is able to estimate the optimal tile size accurately (as indicated by the star) with very small relative error (Figure 26).

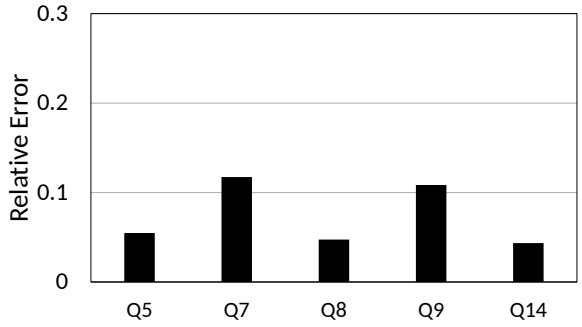


Figure 24: Relative error in GPL runtime estimation.

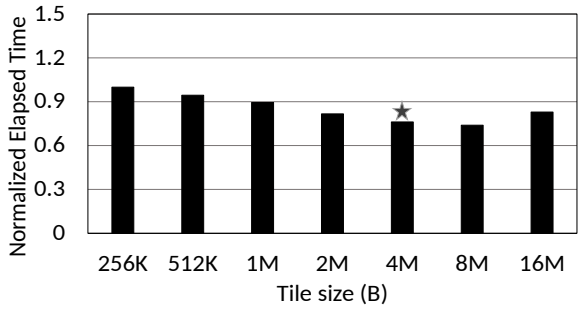


Figure 25: The overall query processing performance with varying tile sizes (Q8) on NVIDIA GPU.

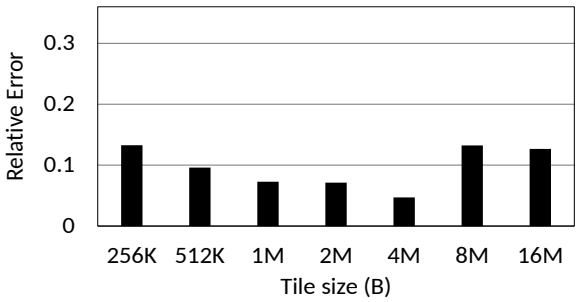


Figure 26: The overall query processing performance with varying tile sizes (Q8) on NVIDIA GPU.

We also study the impact of the number of work-groups on NVIDIA GPU. The results are simliar to those on the AMD GPU.

A.3.2 Evaluation of GPL

Runtime Comparison. Figure 27 show the runtime results for the TPC-H queries on NVIDIA GPU. It shows the runtime of GPL with concurrent execution enabled and GPL without concurrent execution. For better presentation, we normalize the execution time of every query with runtime of the KBE version of the same query. If tiling is applied without concurrent kernel execution then the queries show up to 1.15 times performance degradation on NVIDIA GPU. This is due to both insufficient data parallelism as well as due to the overhead incurred by frequent kernel launches which in turn is a result of input data tiling without using concurrent kernel execution. In GPL, both concurrent kernel execution and input data tiling are both applied together. The results show that GPL outperforms KBE by 50% on NVIDIA GPU.

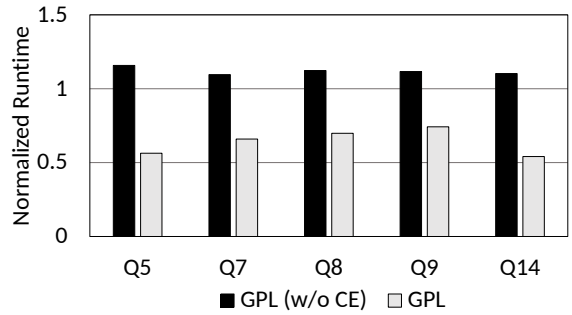


Figure 27: GPL execution time normalized to KBE on the NVIDIA GPU.

Resource Utilization. We take Q8 as an example to demonstrate higher resource utilization achieved by GPL on NVIDIA GPU. Figure 28 shows the average resource utilization achieved by both KBE and GPL for Q8 on the NVIDIA GPU. These results clearly shows that GPL is able to achieve higher average utilization of both memory and computational units when compared to KBE, on NVIDIA GPU.

Communication overhead. To examine the communication overhead among kernels on NVIDIA GPU, we show the breakdown of query execution time for Q8 in Figure 29. For GPL, the data channel access cost (DC_cost) and *Delay* cost existing in pipelines are included into total query execution time as mentioned in Section 5.3.1. In GPL communication

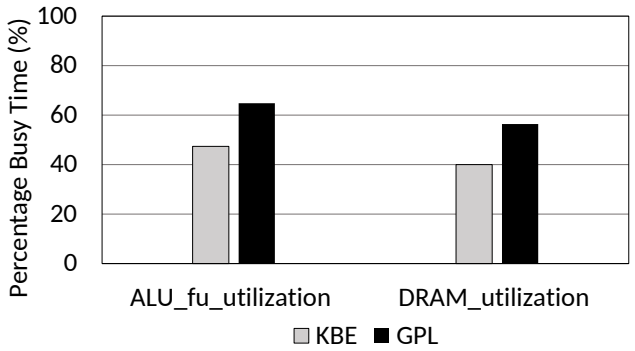


Figure 28: Improved GPU resource utilization (Q8) on the NVIDIA GPU.

cost is only 18% of the total execution time while in KBE it can go up to 32%.

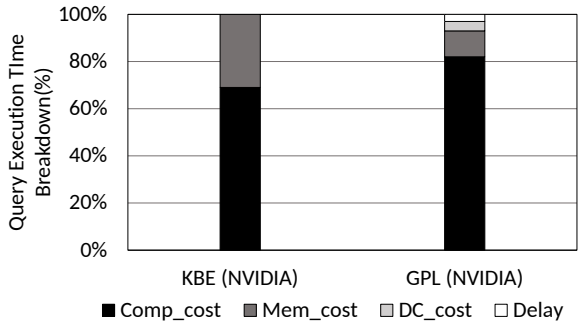


Figure 29: Improved GPU resource utilization (Q8) on the NVIDIA GPU.

Varying data sizes. GPL was also tested against KBE for different data sizes by varying the scale factor from 0.1 to 10 (Figures are omitted). The results are similar to those on the AMD GPU. When data size is small (100MB) GPL is only slightly faster than KBE. This is because an input which is too small cannot provide sufficient data level parallelism. Still, as data size increases, the performance improvement of GPL over KBE also increases.

B. TPC-H QUERIES

Since Ocelot does not support non-trivial string operations and multi-column sort, we used slightly modified versions of Q7, and Q9 to compare with Ocelot. Other queries are the same as those in the previous study [18]. The TPC-H queries used for comparison are given in Listings 2–6.

```
SELECT n_name,
    sum(l_extendedprice * (1 - l_discount)) as
        revenue
from customer, orders, lineitem, supplier, nation,
    region
where c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= date '1994-01-01'
    and o_orderdate < date '1995-01-01'
group by n_name
order by revenue desc
```

Listing 2: Query 5

```
select supp_nation, cust_nation, l_year,
    sum(volume) as revenue
from ( select n1.n_name as supp_nation,
    n2.n_name as cust_nation,
    extract(year from l_shipdate) as l_year,
    l_extendedprice * (1 - l_discount) as volume
from supplier, lineitem, orders, customer, nation n1,
    nation n2
where s_suppkey = l_suppkey
    and o_orderkey = l_orderkey
    and c_custkey = o_custkey
    and s_nationkey = n1.n_nationkey
    and c_nationkey = n2.n_nationkey
    and ((n1.n_name='FRANCE' and n2.n_name='GERMANY')
    or (n1.n_name='GERMANY' and n2.n_name='FRANCE'))
    and l_shipdate between date '1995-01-01'
    and date '1996-12-31') as shipping
group by supp_nation, cust_nation, l_year
order by l_year
```

Listing 3: Query 7

```
select o_year, sum(case when nation = 'BRAZIL' then
    volume else 0 end) / sum(volume) as mkt_share
from (
    select extract(year from o_orderdate) as o_year,
        l_extendedprice * (1 - l_discount) as volume,
        n2.n_name as nation
    from part, supplier, lineitem, orders, customer,
        nation n1, nation n2, region
    where p_partkey = l_partkey
        and s_suppkey = l_suppkey
        and l_orderkey = o_orderkey
        and o_custkey = c_custkey
        and c_nationkey = n1.n_nationkey
        and n1.n_regionkey = r_regionkey
        and r_name = 'AMERICA'
        and s_nationkey = n2.n_nationkey
        and o_orderdate between date '1995-01-01'
        and date '1996-12-31'
        and p_type = 'ECONOMY␣ANODIZED␣STEEL'
) as all_nations
group by o_year
order by o_year
```

Listing 4: Query 8

```
select nation, o_year, sum(amount) as sum_profit
from (
    select n_name as nation, extract(year from
        o_orderdate) as o_year, l_extendedprice * (1
        - l_discount) - ps_supplycost * l_quantity as
        amount
    from part, supplier, lineitem, partsupp, orders,
        nation
    where s_suppkey = l_suppkey
        and ps_suppkey = l_suppkey
        and ps_partkey = l_partkey
        and p_partkey = l_partkey
        and o_orderkey = l_orderkey
        and s_nationkey = n_nationkey
        and p_partKey < 1000
) as profit
group by nation, o_year
order by o_year desc
```

Listing 5: Query 9

```
select 100.00 * sum(case when p_partKey then
    l_extendedprice * (1 - l_discount) else 0 end) /
    sum(l_extendedprice * (1 - l_discount)) as
    promo_revenue
from lineitem, part
where l_partkey = p_partkey
    and l_shipdate >= date '1995-09-01'
    and l_shipdate < date '1995-10-01'
```

Listing 6: Query 14