Performance Modeling in CUDA Streams - A Means for High-Throughput Data Processing

Hao Li, Di Yu, Anand Kumar, and Yi-Cheng Tu

Department of Computer Science and Engineering, University of South Florida
4202 E. Fowler Ave., ENB118, Tampa, FL 33620, U.S.A.
{haoli1,diyu,akumar8}@mail.usf.edu, ytu@cse.usf.edu

***Abstract*—Push-based database management system (DBMS) is a new type of data processing software that streams large volume of data to concurrent query operators. The high data rate of such systems requires large computing power provided by the query engine. In our previous work, we built a push-based DBMS named G-SDMS to harness the unrivaled computational capabilities of modern GPUs. A major design goal of G-SDMS is to support concurrent processing of heterogenous query processing operations and enable resource allocation among such operations. Understanding the performance of operations as a result of resource consumption is thus a premise in the design of G-SDMS. With NVIDIA's CUDA framework as the system implementation platform, we present our recent work on performance modeling of CUDA kernels running concurrently under a runtime mechanism named *CUDA stream*. Specifically, we explore the connection between performance and resource occupancy of compute-bound kernels and develop a model that can predict the performance of such kernels. Furthermore, we provide an in-depth anatomy of the CUDA stream mechanism and summarize the main kernel scheduling disciplines in it. Our models and derived scheduling disciplines are verified by extensive experiments using synthetic and real-world CUDA kernels.**

***Keywords*-push-based systems, DBMS, GPU, GPGPU, CUDA, CUDA stream**

I. Introduction

Relational database system management system (RDBMS) has been the predominant database processing platform for several decades. The core of RDBMS is its query engine, which follows a pull-based design: the system creates a computational thread for each individual query to pull out the exactly needed data from storage to fulfill query processing. However, due to physical limitations of magnetic disks where data is stored, I/O bandwidth has been the main bottleneck that limits the performance of RDBMS. Push-based DBMS [4], [11], [23], [28] is another kind of database system where such limits are alleviated. It uses a streaming mechanism to deliver data to multiple queries, and all the queries share the same data streams simultaneously. By that, push-based DBMSs move the bottleneck from I/O to computation. The improvement of computing capabilities of modern hardware systems, especially multi-core processing units, coincides perfectly with the resource needs of push-based DBMSs. In this paper, we focus on the scenario of using Graphics Processing Units (GPU) as the platform for the implementation of such DBMSs. Traditionally used for graphics processing, GPUs are now widely used for general-purpose computing since they provide unrivaled parallel processing power, as well as general-purpose programing frameworks such as the compute unified device architecture (CUDA) and Open Computing Language (OpenCL). General-purpose computing on GPUs (GPGPU) has become an important high-performance computing (HPC) platform in many scientific and engineering fields.

The world has entered an age of big data where many applications have to deal with massive amount of data that can easily overwhelm a traditional DBMS. To that end, novel high-throughput data management system is a must. For reasons mentioned above, the database community sees push-based DBMSs as a promising approach to meet big data management challenges in many application domains. There are two types of push-based DBMS: 1) data stream management systems (DSMS) [3], [6] for monitoring and analyzing continuous real-time data; 2) scan-based DBMSs [4], [5], [11], [28] that process concurrent queries against streams of data obtained by scanning stored database tables. DSMS is widely used for real-time data processing in various environmental monitoring applications such as sensor networks and network management. Scan-based DBMSs are mainly used for read-intensive or read-only application domains such as online analytical processing (OLAP) and scientific data processing. In a push-based system, the rate of data input to the computing nodes can be large to an extent that the computational capacity in the query engine provided by CPU cores is overwhelmed. Comparing with CPUs, GPUs have much more computing power and lower energy consumption. In our previous work [26], we proposed a GPGPU-based Scientific Data Management System (G-SDMS) that aims at using CUDA-supported GPUs as the platform to run the query processing engine in a push-based DBMS. Unlike most high-performance computing (HPC) systems that focus on using GPU to process individual computation tasks, the main design goal of G-SDMS is to support heterogeneous operations running concurrently. This is due to the nature of push-based systems where all queries

978-1-4799-5666-1/14/$31.00 ©2014 IEEE

share the same data input streams.

In CUDA, kernel concurrency is achieved via a mechanism named *CUDA stream* [16]. One or more kernels can be assigned to different streams, multiple streams can run at the same time as long as the resources are sufficient. To achieve high efficiency of data processing in G-SDMS, the key problem is resource allocation among concurrent kernels. The G-SDMS query engine periodically re-launches kernels with different runtime parameters to reflect the resource allocation scheme determined by a runtime query optimizer. However, understanding kernel performance in response to resource consumption is a premise for such a design. Therefore, the objective of the study reported here is to model the performance of CUDA kernels running concurrently in CUDA streams. For that purpose, two important aspects should be considered: 1) the kernel scheduling scheme in CUDA when CUDA streams are launched, and 2) the performance of each single-kernel operation. Knowing the scheduling scheme of CUDA streams can help us effectively manage resource allocation to different concurrent query processing tasks, thus providing a handle for optimizing system performance. The performance model(s) of single kernels, on the other hand, serves as the foundation and basic building block of multi-kernel models. We start our work in this topic by modeling the performance of a CUDA kernel (assuming the absence of any other kernels) based on its resource consumption. Specifically, we identify the types of resources that are relevant in modeling kernel performance. Then, using the CUDA occupancy calculator provided by NVIDIA, we develop a model describing the relationship between performance and runtime parameters (e.g., number of threads, number of registers, and shared memory) for computation-bound kernels. Our model can predict rounds of computation, which is the major factor that determines the running time of such kernels. Our model is verified by extensive experiments using synthetic and real-world CUDA kernels. With the help of another NVIDIA tool, the Visual Profiler, we deduce the scheduling mechanism of CUDA streams from the execution behavior of a large number of testcases. Via such experiments, we summarize three basic disciplines that govern the kernel scheduling of CUDA streams. With these three rules, we can predict the concurrent kernel behavior given the kernel parameters. In summary, this paper makes the following two contributions:

1. we explore the connection between performance and resource occupancy of single compute-bound kernel and provide a model that can predict the performance of such kernels; and
2. we provide an in-depth anatomy of the CUDA stream mechanism and summarize the main kernel scheduling disciplines in it.

The remainder of this paper is organized as follows. In Section II, we briefly introduce the technical background of this study; In Section III, we describe our performance modeling of single kernels; Section IV presents the experiments to explore kernel scheduling policies of CUDA streams; We compare our study with related work in Section V and conclude the paper in Section VI.

II. Background

A. The G-SDMS system

G-SDMS is a push-based DBMS that takes advantage of the large computational capabilities of GPUs. At runtime, the query processing engine executes pipelined query execution plans/trees that are similar to those in traditional RDMBS. Operators (e.g., representing either a relational operator or an analytical function) from all concurrent queries for a query operator network that share the same data stream as input, with the output of an upstream operator as the input to the next one(s) in the workflow (Fig. 1). Intermediate results can be hold by each operator's memory buffers (queues), new query plans can be added to the current network of operators at any time. Contrary to stream data management systems where data comes continuously from a remote streaming source (e.g., sensors, stock price updates), the G-SDMS data stream is the results of scanning data tables/files stored on disks. The scan-based I/O framework has the following benefits: 1) Random I/O is minimized to achieve extraordinarily high data throughput; and 2) Combining data processing load into the scanning process can thinly spread cost to many queries. With such an I/O framework, G-SDMS normally receives input data at a very high rate (e.g., up to 4.8 GB/s in a low-end storage server with dual FibreChannel interfaces [22]). This is also the main motivation of using GPUs to process the queries. In G-SDMS, the operators are mapped to CUDA kernels. Since all kernels share the same input data stream, kernel concurrency is an important requirement in the implementation of G-SDMS. Furthermore, query optimization in G-SDMS should naturally take the entire query network into consideration (comparing with treating each query individually in traditional DBMSs). Although different optimization goals can be targeted, we normally focus on maximizing input data throughput (i.e., processing as much input data as possible). For that purpose, the performance (i.e., throughput) of each operator/kernel needs to be controlled at runtime. This is accomplished by controlling the resource allocated to the kernels. Furthermore, the designed architecture of G-SDMS can serve not only for a DBMS but also for any task-parallelism system which has GPU clusters. The only problem is to figure out the resource schedule scheme of GPGPU, which is a black box to us, so that it can truly benefit the task-parallelism system as well as our G-SDMS. Thus, modeling kernel performance in a multi-kernel environment is the main objective of this study.

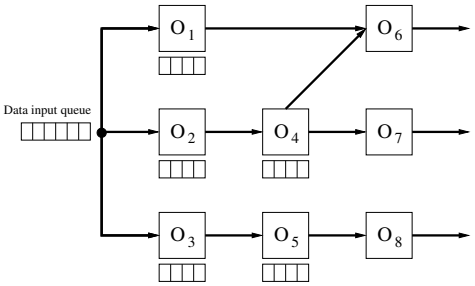

Figure 1. An example of G-SDMS runtime query network

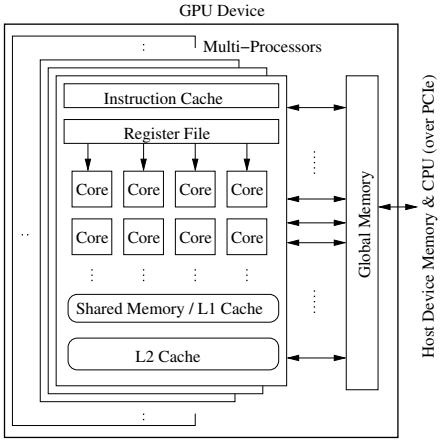

Figure 2. Architecture of a typical GPU

B. GPGPU and CUDA stream

The architectural features of GPUs from different vendors such as AMD and NVIDIA are almost the same. However, in this paper, we focus on the architecture of NVIDIA GPUs as well as the associated CUDA programming framework. We will also elaborate on *CUDA stream* - the main mechanism to enable concurrency in paralleled tasks.

1) Typical GPU Architecture: A GPU contains many processing units that are tightly coupled together by special hardware. A typical architectural view of the GPUs is shown in Fig. 2. Large number of processing cores are grouped together in different segments called multi-processors (MPs). A typical GPU contains multiple MPs. Each MP is an independent entity and controls all the cores it hosts. These cores on each MP process instructions in a *single-instruction-multiple-data* (SIMD) fashion. Data to be processed is generally placed in global memory (GM) of the GPU device. All MPs can access this memory to perform the computations. Global memory can be accessed in parallel by different cores. The access speed can be as high as 200GB/s [20], provided some access patterns are followed. Second level of memory offered by the GPU architecture is the high-speed L1 cache that belongs to each MP. Part (up to 48KB in each MP) of the L1 cache can be configured to a programmable section called the shared memory (SM). The SM is laid out in groups of 32 banks, each of 4 bytes. The banks can also be accessed in parallel. Within each MP, there are also nonprogrammable L2 cache with a certain size (512 KB [20]) and a bandwidth smaller than that of the L1 cache.

All of the data to be processed should be transferred to GM through the programming primitives in CUDA. Once, the data is in the device, the processing can begin by calling a special function, called *kernel*. This is when the multi-processors become active and the cores start using the registers to load data from GM and perform computations. The SM is programmed inside the kernel and is not visible to the CPU code. Once all the computations are done, the kernel returns and the device is free to spawn new kernels. In newer generations of CUDA, a kernel can launch other kernel threads at runtime via a mechanism called *dynamic parallelism*.

The GPU kernel needs to be spawned with suitable number of computation threads to utilize the computation power of the device. A group of threads that are executed on a single MP is called a *block*. Large number of blocks can be processed by different MPs of the device. The device driver is responsible for scheduling the blocks on different MPs. MP schedules 32 threads as a unit, called *warp*. Each thread has access to the registers of its MP, entire global memory, and shared memory of its MP. MP will schedule eight registers as a unit for threads. The GM access is efficient if 32 threads on an MP access consecutive bytes at any given clock cycle. SM access is efficient if threads access different banks at a time. Therefore, it is the programmer's responsibility to carefully lay out and access the data for maximum utilization of the memory bandwidth [15].

2) CUDA stream: CUDA provides a programming model called *CUDA stream* with the ability to schedule multiple CUDA kernels simultaneously. One CUDA stream can encapsulate multiple kernels, and they have to be scheduled strictly following a particular order. However, kernels from multiple streams can be scheduled to run concurrently. Such a CUDA stream example is illustrated in Fig. 3. The kernels can only be executed in a serial order without CUDA stream, and they can be executed in a concurrent way with CUDA stream, the latter obviously can lead to much better performance.

The main purpose of using CUDA streams is to hide the memory latency: when kernel A is loading/writing data, kernel B can occupy the cores for computation. As a result, the cores in the MPs reach better utilization. For example, NVIDIA cards with the Fermi architecture allow up to 16-way concurrency and Kepler allows 32-way concurrency.

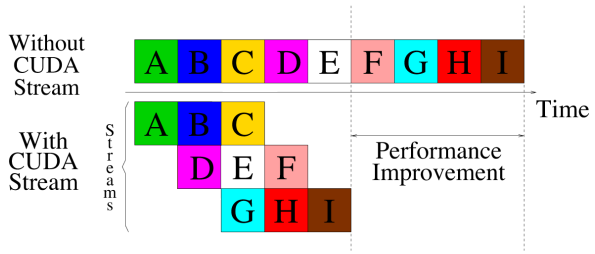

Figure 3. An example of three CUDA streams, each containing three kernels

For computing a tiled Dense General Matrix Multiplication (DGEMM) problem on a K20 GPU, the throughput is about 519 Gflop/s when operations are serial while it increases to 990 Gflop/s when a 3-way concurrency is achieved. [1] In Kepler with compute capability 3.5, there is a new feature called Hyper-Q in CUDA stream. It allows connection from CUDA streams and Message Passing Interface (MPI) processes therefore can ignore the false dependency in code and achieve higher concurrency.

NVIDIA does not reveal much detail about the internal mechanism for kernel scheduling in CUDA streams. However, such scheduling policies are critical in understanding and modeling the resource occupancy (and thus performance) of the kernels. This is one of the main research tasks of this study. NVIDIA provides a profiling tool named Visual Profiler with a graphical user interface to give developers feedback regarding their CUDA programs. We use it to explore the internal scheduling scheme for CUDA streams.

III. Single Kernel Modeling

In this section, we present our work on modeling the performance of a CUDA kernel assuming there are no other kernels running in the system. Specifically, our models describe the relationship between the resource consumption and performance of a kernel. The purpose of such work is two-fold: First, with the proper models we will be able to find the right configuration of kernel parameters that lead to optimized kernel performance. Note that this is an important problem itself in any GPGPU work, especially in many traditional HPC applications where there is only one (or a small number of) kernel. CUDA allows a kernel to be launched with a set of user-specified parameters. How to set such parameters has always been a puzzle to CUDA programmers. Second, single kernel models serve as the key component in our work towards performance modeling and optimization of multiple kernels encapsulated in multiple CUDA streams, which is always the scenario in our G-SDMS system. We will elaborate more on this in Section IV. In this paper, we focus on performance modeling of computation-bound kernels. Modeling the behavior of memory-bound kernels requires a significantly different approach and we leave that as future work.

Table I
Constant Variables and their values under CUDA Compute Capability 3.0

Symbol	Meaning	Value
M_{BMP}	Maximum number of blocks of an MP	16
M_{WMP}	Maximum number of warps of an MP	64
M_{TW}	Maximum number of threads of a warp	32
M_{RT}	Maximum number of registers of a thread	64
M_{RMP}	Maximum number of registers of an MP	65535
M_{SMP}	Maximum SM amount of an MP	49152
G_{RW}	Register allocation unit size for an MP	256
G_{WMP}	Warp allocation unit size for an MP	4
G_{SB}	SM allocation size for a block	256

A. Model development

As mentioned earlier, CUDA allows a kernel to be launched with large number of threads and blocks, giving users an impression that they are run simultaneously. Actually, the threads will have to take turns in using the hardware to make progress. The secret of GPU's high computing capability is parallelism. Generally, the level of parallelism in running a kernel determines a kernel's performance. To that end, we propose a model using *Running Rounds* to quantify the performance of a computation-bound single kernel.

$$RunningRounds = \left\lceil \frac{T \times B}{\alpha \times M_P} \right\rceil \quad (1)$$

To be more specific, running round is the number of rounds all the specified threads of a kernel are actually scheduled. Threads within the same round can be regarded as being processed concurrently. Here T is the number of threads in each block, and B is the block number of this kernel – those are two parameters specified by the programmer in launching the kernel. The numerator of Eq. (1) is actually the total number of threads for running the entire kernel. Furthermore, α is the number of threads that can be run in each round in a single MP, and M_P is the total number of MPs in a GPU. M_P is a constant depending on what GPU card is used. From now on, any such constants will be symbolized in bold font, and Table I shows the value of some relevant constants. Let us define β as the number of blocks that can be actually scheduled at the same time, we obviously have $\alpha = \beta \times T$.

On the hardware side, the threads in a block will only be scheduled in one MP - we call this the *Block-to-MP binding*. Depending on the resource availability, multiple blocks can be run in the same round on an MP. In CUDA, the resources we have to consider include registers, shared memory, and also the number of warps. We will elaborate on warps in a moment. Each resource comes with a limited amount on

each MP, therefore each acts as a limiting factor to β. We have

$$\beta = \min(\beta_W, \beta_R, \beta_S) \tag{2}$$

where β_W is the number of concurrent blocks when only the warp number is considered as the limiting factor. And β_R and β_S are defined the same way for registers and shared memory, respectively. And we take the minimum value among the three quantities to get β.

Let us denote the actual use of SM in a block as ζ, we have

$$\beta_S = \begin{cases} 0 & \zeta > \mathbf{M_{SMP}} \\ \left\lfloor \frac{\mathbf{M_{SMP}}}{\zeta} \right\rfloor & \mathbf{M_{SMP}} \geq \zeta > 0 \\ \mathbf{M_{BMP}} & \zeta = 0 \end{cases} \tag{3}$$

which shows that, when ζ is larger than the total amount of SM in an MP, we cannot schedule any blocks. If it is zero, then SM is not a limiting factor at all and we set β_S to be the maximum number allowed by CUDA. If ζ is between zero and maximum SM of an MP, we floor the maximum SM per MP over ζ to get β_S.

Similarly, β_R is determined by the number of registers used in all the blocks. To calculate β_R we need to revisit the concept *warp*. As mentioned earlier, the basic unit of threads schduled by the CUDA scheduler is a warp (instead of a block). An MP can run multiple warps at the same time. We denote the number of warps in a block as γ, we have

$$\gamma = \left\lceil \frac{T}{\mathbf{M_{TW}}} \right\rceil \tag{4}$$

and

$$\beta_R = \begin{cases} 0 & R > \mathbf{M_{RT}} \\ \left\lfloor \frac{\delta}{\gamma} \right\rfloor & \mathbf{M_{RT}} \geq R > 0 \end{cases} \tag{5}$$

that says: if the number of registers used per thread R is beyond the total registers allowed in a thread M_{RT}, no threads can be scheduled; If the register per thread is less than M_{RT}, we can use limited warps due to register used to get β_R. Here δ is the number of warps that can run on an MP based on R. Particularly, we get it by

$$\delta = \left\lfloor \frac{\mathbf{M_{RMP}}}{\theta \times \mathbf{G_{WMP}}} \right\rfloor \times \mathbf{G_{WMP}} \tag{6}$$

and

$$\theta = \left\lceil \frac{R \times \mathbf{M_{TW}}}{\mathbf{G_{RW}}} \right\rceil \times \mathbf{G_{RW}} \tag{7}$$

In addition to registers, the number of threads can also affect the number of warps that can run simultaneously on an MP. CUDA sets further limits on how many warps can run at the same time. Specifically, we have

$$\beta_W = \min\left(\mathbf{M_{BMP}}, \left\lfloor \frac{\mathbf{M_{WMP}}}{\gamma} \right\rfloor\right) \tag{8}$$

As a side note, we often use the word *occupancy* to refer to the level of parallelism achieved in running CUDA kernels. We will also use this quantity in our following discussions. Thus, we formally define occupancy as

$$o = \frac{\alpha}{M_{TW} \times M_{WMP}} \tag{9}$$

B. Optimizing performance of a kernel

Note that the number of registers and amount of SM needed for a thread is fixed at compile time – their values are determined by the way CUDA kernel code is written. Meanwhile, the number of threads can be controlled by the kernel parameters T and B at runtime.[1] With the above modeling results, we can actually formulate the problem of *finding the optimal runtime parameters in launching a kernel* as the follows:

$$\boldsymbol{minimize} \quad \left\lceil \frac{T \times B}{\alpha \times M_P} \right\rceil \tag{10}$$

$$\boldsymbol{subject\ to} \quad \beta_W \geq \beta_S \tag{11}$$

$$\beta_W \geq \beta_R \tag{12}$$

$$T \times B = C \tag{13}$$

The first two constraints refer to the fact that β_S and β_R are static values given the code, so we need to make sure β_W overshadows them. The last constraint reflects the common situation that the total number of threads to run a kernel follows a fixed constant number C (e.g., assigning a thread to a single component of a natural system). The above problem can be solved by using any optimization package with integer programming capabilities. Luckily, as T can only be an integer smaller than the maximum number of threads allowed in a block (i.e., 2048 for latest version of CUDA), the above problem can be solved with a very small overhead.

C. Model Verification

If our performance model is correct, it can be used to predict the performance of a single kernel launched with any set of parameters. As a result, we can also solve the optimization problem mentioned in Section III-B before we run the kernel at all. Hence, we use kernels for solving real-world problems to validate our models. For each kernel, we run it with different values of T and B values (but with a fixed $C = T \times B$ value) and measure the actual running time of the kernels. We then compare the measured time with the running rounds predicted by our model for model verification. The experiments were run in a workstation with an Intel E8400 CPU@3.0GHz, 8 GB of DDR3 1333MHz

[1] Actually, CUDA allows the kernel code to be programmed with a certain amount of SM set as the 3rd runtime parameter. However, such practice will dramatically complicate code development and is not recommended therefore we do not discuss it in our modeling work.

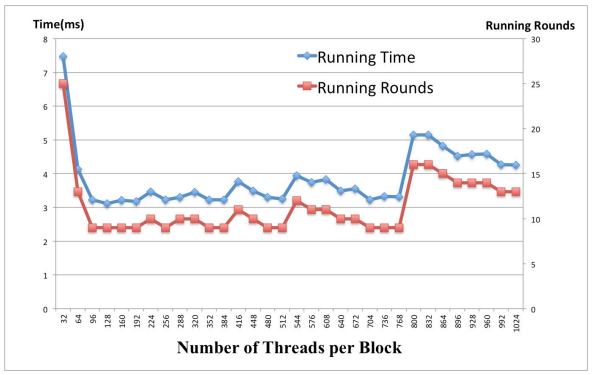

Figure 4. Running Time and Running Rounds under different number of threads per block in running Kernel I

Table II
CORRELATIONS OF MODEL VERIFICATION EXPERIMENTS USING 3 DIFFERENT KERNELS

Kernel No.	Pearson Correlation Coefficient
I	0.9974
II	0.9791
III	0.9606

memory, one 300GB Seagate ST33204 hard disk, and one NVIDIA GeForce GTX680 graphics card. The machine runs Ubuntu 3.5 OS and CUDA version 6.0. We chose three kernels for our experiments: kernel I is what we developed to compute the radial distribution function of molecular simulation data [7], kernel II is for computing histograms from large data [2], and kernel III is for non-indexed nested loop join from an in-memory database [12]. The results of kernel I can be seen in Fig. 4. We can see that the trends of both measured running time and predicted running rounds match almost perfectly. For such trend analysis, the Pearson Correlation is a good indicator. Table II shows the Pearson Correlation values for all three kernels we tested. They are all very close to 1.0, which indicates a (almost) perfect matching between the measured and predicted performance.

IV. MULTI-KERNEL STUDIES

In this section, we experimentally study the CUDA stream scheduling mechanism. The experimental setup is the same as that described in Section III-C except we design a series of kernels for our experiments in this section. The advantage of synthetic kernels is that we can use them to simulate tasks with any resource consumption pattern as we wish. In particular, we develop six different kernels in our experiments shown in Table III.

A. Level of Concurrency in a CUDA Stream

Since our goal is to study multi-kernel concurrency, knowing how many blocks from different kernels can run

Table III
KERNEL PARAMETERS FOR CONCURRENCY EXPERIMENTS

	Registers	Threads per block	Occupancy	Shared Memory(byte)
Kernel A	11	1024	100%	0
Kernel B	33	1024	50%	0
Kernel C	11	256	100%	0
Kernel D	44	256	63%	0
Kernel E	11	256	100%	24,576
Kernel F	44	256	63%	24,576

Table IV
KERNEL PARAMETERS FOR STREAM EXPERIMENTS

	Registers Needed	Threads per Block	Occupancy	Running Time(ms)
Kernel A	11	1024	100%	14
Kernel B	33	1024	50%	20

at the same time is still the key problem. From Eq. (2), we know that the three types of resources (i.e., warps, registers and SM) are the limiting factors for the number of active wraps a kernel can have at runtime. In the scenario of having multiple kernels / steams, we hypothesize that such factors play the same roles. We conducted three experiments to confirm our hypothesis. We use NVIDIA GTX 680 GPU as the experimental platform.

In the first experiment, we encapsulate one kernel A and one kernel B into two separate CUDA streams, respectively. More features of these two kernels can be found in Table IV. We found out that an MP can run one block of kernel A and one block of kernel B at the same time. As mentioned earlier, registers are allocated in units of 8. Each thread of kernel A needs 16 registers, each thread of kernel B needs 40 registers. Therefore, one block of kernel A has $1024/32 = 32$ active warps occupying $16 \times 1024 \times 1 = 16,384$ registers; one block of kernel B has $1024/32 = 32$ active warps using $40 \times 1024 \times 1 = 40,960$ registers. Neither kernels require any SM to run. In total, they have 64 active warps, 57,344 registers, and zero bytes of SM. Neither the total number of registers nor the amount of SM needed exceeds the limit on an MP, hence in this case, the number of active warps is the bottleneck.

In the second experiment, we use kernel C and kernel D, again, in two different CUDA streams, respectively. The result is that an MP can run up to one block of kernel C and five blocks of kernel D at the same time. One block of kernel C has $256/32 = 8$ active warps, uses $16 \times 256 \times 1 = 4,096$ registers, and zero bytes of SM; five blocks of kernel D have $256 \times 5/32 = 40$ active warps, use $48 \times 256 \times 5 = 61,440$ registers, and zero SM. In total, they have 48 active warps, 65536 registers, and zero bytes of SM. The total active warps and SM are both less than the limit set on an MP, therefore

in this case, number of registers is the constraint.

In the third experiment, we choose kernel E and kernel F in the same two-stream setup. An MP can run up to one block of kernel E and one block of kernel F at the same time. One block of kernel E needs $256/32 = 8$ active warps, uses $16 \times 256 \times 1 = 4,096$ registers, and 24,576 bytes of SM; one block of kernel F takes $256/32 = 8$ active warps, uses $48 \times 256 = 12,288$ registers, and 24,576 bytes of SM. In total, they take 16 active warps, 16,384 registers, and 49,512 bytes SM. The warps and registers are less than the limit of a MP, thus in this case, the amount of SM is the bounding factor.

Via the above experiments, we can clearly see that warps, registers and SM are also the limiting factors for scheduling multiple kernels in the CUDA streams. And the same resource bounding analysis we saw in Section III still holds true. It is possible to run multiple kernels if all limits are not exceeded. Furthermore, we can use the models developed in Section III to determine how many kernels can operate at the same time before running them. Such prediction is the basic information needed in deriving the scheduling policies in CUDA streams, as shown in the remainder of this section.

B. CUDA stream scheduling mechanism

We design several experiments to explore the CUDA stream scheduling mechanism. Two kernels shown in Table III are used for such experiments, and furthermore information of them is shown in Table IV. In our experiments, we set the thread numbers per block to be 1024 unless specified otherwise. According to our analysis shown earlier, each MP can at the same time execute up to two blocks of A, only one block of B, or a block of A plus a block of B. Note that two blocks of B require more registers than one MP can offer and therefore cannot concurrently run on one SM (in both GTX680 and Tesla K40).

First, we want to show that kernels in CUDA streams indeed enjoy concurrency via a mini-experiment. We launch 8 CUDA streams, each of which runs a copy of the same Kernel A picked from Table IV. Each kernel is set it to run on only four blocks and 1024 threads per block. each MP can run two blocks of Kernel A at the same time. Since the GTX680 has eight MPs, there should be four streams containing 16 blocks running at the same time. The result shown in NVIDIA Visual Profiler confirms this analysis (Fig. 5). The same results are obtained on the K40 GPU card.

The first scheduling rule we are interested in is: how are the different blocks of a kernel distributed to the different MPs? Two patterns are possible: 1) blocks of a kernel are put to as many MPs as possible; 2) the scheduler fills as many blocks as possible into one MP before putting anything in the next. This time, we use both kernels A and B for our experiments. Specifically, we set each kernel A to run with 8 blocks, and each kernel B with 5 blocks. We use three

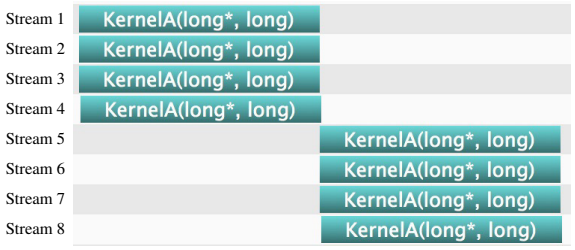

Figure 5. Visual Profiler view of eight streams, each containing a kernel A. Each green block represents the running period of a block

CUDA streams: Stream 1 contains a kernel A, Stream 2 contains a kernel B, and Stream 3 has a kernel B. We name each block of A in stream 1 as A1, each block of B in stream 2 as B2, and so on. If the scheduler acts as in pattern 1, five of the eight MPs will each run one A1 and one B2, the other three MPs will each run one A1 and one B3. For the second pattern, four MPs will each run two A1s, and the other four MPs will each run one B2, the remaining blocks of B need another round to be scheduled. In other words, all three streams should be concurrent for pattern 1, while only the first two streams will be for pattern 2. From the results shown in Fig. 6, we can deduce that it is pattern 1 that the scheduler follows. We also get the same results on K40. By this, we have the first discipline: ***CUDA scheduler always takes as many MPs as possible in scheduling the different blocks of a kernel.***

Further analyzing the result, we found that the running time for all A blocks is about 14 ms, for B kernels it is about 20 ms. Let us divide this process into three phases, every time when a stream is finished, we mark the beginning of a new phase (Fig. 6). We also use a grid to illustrate the running blocks of different MPs, with each cell in the grid standing for an MP. In phase I, there are five MPs each running one A1 and one B2, the other three MPs each runs one A1 and one B3. It takes 14 ms to finish phase I (as A1 dominates the running time). In phase II, since each MP can only take up one block of kernel B, now each of the five MPs is running one B2 and the other three MPs is each running a B3. It takes $20 - 14 = 6$ms time to finish phase II, while the first three B3s also finished. In phase III, the remaining two B3s are scheduled and it took them 20ms to finish.

In the next set of experiments, we want to know whether new blocks can be scheduled if MP is running some blocks yet some resources are released due to finished blocks. Again, we use 3 streams: Stream 1 contains a kernel A with 8 blocks, Stream 2 has a kernel B with 8 blocks, and Stream 3 a kernel A with 8 blocks. Here we can use the same phase analysis method mentioned above to draw conclusions. The scheduler can only follow one the

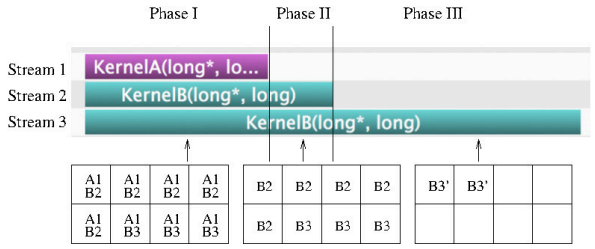

Figure 6. Three phases when running three streams containing kernels A, B, and B, respectively

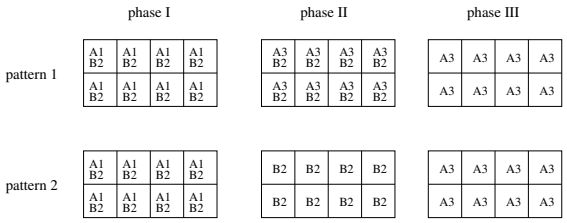

Figure 7. Expected phases in running 3 streams under two scheduling patterns.

two patterns: 1) it can schedule the next round of kernels whenever there are sufficient resources being released; and 2) all new blocks will be scheduled when all running blocks are finished. If it is pattern 1, in phase I each of eight MPs will take one A1 and one B2, and running time will be 14ms ; in phase II, all A1 blocks are finished, since each MP can run one kernel A and one kernel B, the MPs that just had A1s finished will immediately take one A3 and continue running the B2 for another 6 ms; in phase III, the eight MPs will each finish the A3 it had, and running time will be $14 - 6 = 8$ms. If it is pattern 2, phase I is the same; in phase II, A1 just finished, since an MP should be block-free to take new blocks, all MPs will only continue running B2, running time will be about 6 ms; in phase III, each of eight MPs will take one new A3, and the running time will be 14 ms. Fig. 7 shows the expected kernel assignment scenarios under these two patterns. For pattern 1, Stream 1 and Stream 2 should have concurrency in phase I, Stream 2 and Stream 3 should have concurrency in phase 2; For pattern 2, concurrency only exits in phase I between Stream 1 and Stream 2.

Information from the Visual Profiler supports the pattern 1 result (Fig. 8) – we observed concurrency in both phase I and phase II, and phase II ran for only 8ms. We run a few other testcases and also in the K40 card, same results are obtained. Therefore, we have our second discipline: ***New blocks are scheduled immediately when resources are available due to the completion of previously running kernels.***

So far, each CUDA stream we tested only contains

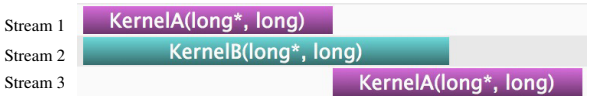

Figure 8. Visual Profiler view of three streams containing kernels B, A, and B

one kernel. Now we study the scheduling rules under the situation of having multiple kernels chained together in a CUDA stream. Specifically, we use four streams, each stream contains one kernel A with eight blocks and one kernel B with six blocks. The streams are ranked by its order of being defined in the source code, with Stream 1 ranked first, Stream 2 second, and so on. Again, we use our phase analysis method as shown in Fig. 9. In phase I, each MP takes an A1, it needs 14 ms to finish. Although each MP has space to run another block of A (e.g., A2, A3), the scheduler did not put any such A blocks for running. Instead, the A2s are scheduled in phase II after all A1s are finished. In phase II, six MPs each takes a B1 and an A2, the other two MPs each takes an A2, and it takes 14 ms to finish. In phase III, we can see the six MPs all keep running the old B1, the other two MPs each takes a B2, and it costs 6 ms to finish. This time, each MP is also available to take one more A3, but this did not happen – those A3s are scheduled after B1s are finished and all the B2s are scheduled in phase IV. The same situation happens in phase V. From the above results as well as those from a few other examples, we derive our third discipline: ***blocks of a CUDA stream can be scheduled only after all blocks in the streams of a higher rank are scheduled.*** This conclusion is very interesting in that programmers have to be careful in the order of defining streams in their code as such orders (rank) have profound effects on system performance.

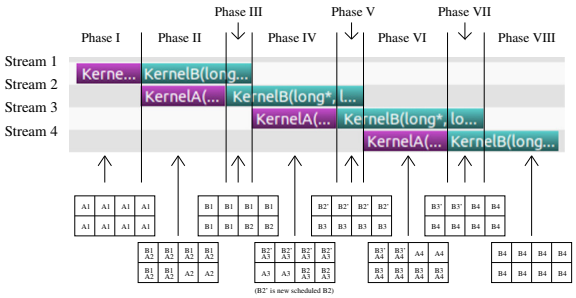

Figure 9. Eight phases of running four streams on GTX680

However, there are exceptions for the above rule, depending on the GPU architecture. NVIDIA introduced a new technique called Hyper-Q [21] for GPUs with compute capability 3.5 or higher. Hyper-Q allows a kernel block to be scheduled before all blocks of a higher-ranked stream are scheduled, thus achieving better concurrency. We use Tesla

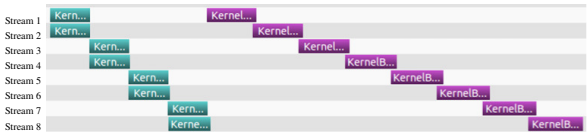

Figure 10. Visual Profiler view of running eight streams each containing one block of kernel A (blue) and one block of kernel B (purple) on K40

K40, which has compute capability 3.5, to run a similar experiment except we had eight streams this time. And the results (Fig. 10) clearly show the different behavior of Hyper-Q: the A blocks are scheduled before any of the B blocks are. Therefore, we modify the previous rule to the following: ***without Hyper-Q, blocks of a CUDA stream can be scheduled only after all blocks in the streams of a higher rank are scheduled.***

V. Related Work

Push-based Database Systems. The pull-based query processing mechanism in traditional DBMSs needs to load data repeatedly and mostly in a random order. Sharing data from a common I/O stream among concurrent queries has become an active topic in the database community. Harizopoulos *et al.* [11] introduced an on-demand simultaneous pipelining (OSP) to enable dynamic operator sharing by pipelining their outputs. Ramen *et al.* [23] presented a system called Blink that runs every query based on a table scan. Frey *et al.* [8] designed an efficient join algorithm called cyclo-join for ring-structured network. Unterbrunner *et al.* [27] implemented a distributed relation table design called Crescando. Crescando uses shared scan on multi-core machines for data-streaming processing. Data sharing in both record and column disk storage models was studied in [28]. Later, Arumugam *et al.* [4] developed a push-based system called DataPath, in which queries are pushed to processors and all the operations can share data. This kind of push-based DBMS becomes the new trend in developing data management systems.

GPGPU and databases. We focus on GPU as the platform because it provides much more computing power and lower energy consumption than modern CPUs. Actually, the GPGPU movement started a long time ago [18]. The advanced computing model such as CUDA [17] and OpenCL [10] accelerates its spread. According to [24], there are more than 30,000 technical papers published annually in the field of GPGPU. It has become very clear that it is a popular computational platform in many application domains [14], [19]. The data management community has also done a lot of work on improving database performance using GPUs. Govindaraju *et al.* [9] developed an algorithm for common database computing operations using GPU as a co-processor. The GPU-based algorithm got huge performance improvement than a compiler-optimized SIMD implementation, e.g. range query got nearly 40 times faster. Bakkum *et al.* [5] implements a subset of command processors called SQLite to accelerate database operations on GPU. Sitaridi *et al.* [25] proposed a solution to resolve bank and value conflict issue of SM on GPU. It can fully utilize the bandwidth of SM therefore enhancing performance. And there are many pieces of work focusing on improving the throughput of join algorithms. He *et al.* [12] implemented novel relation join algorithms on GPUs. Their algorithms obtained 2-7X better performance as compared to CPU-based algorithms. Kaldewey *et al.* [13] also implemented join processing algorithms on GPUs, although they only recorded a 50% performance boost over CPU implementations of the same algorithms.

VI. Conclusions and Future Work

GPGPU becomes an attractive option to build push-based DBMS in this age of big data for its high computing capability. Task parallelism enabled by a CUDA feature, the CUDA stream, makes CUDA the appropriate platform for implementing the push-based DBMS (named G-SDMS) under development in the authors' group. Understanding the performance of computational tasks under different resource consumption in the context of CUDA streams is the prerequisite of building the system. In this study, we explore the CUDA stream runtime resource scheduling scheme and develop a computation-bound single kernel performance model. Both of them are key components towards an optimized query engine in G-SDMS. A series of experiments validated our model. We also derived three basic disciplines that govern the CUDA stream scheduler.

One immediate task in refining G-SDMS is to dynamically schedule the resource for different operations (each one is achieved by one or multiple kernels) based on real-time feedbacks. For example, from the feedback if we find out that one kernel has low throughput due to little resource allocated, we can schedule more resources to this operation to increase its performance. In that way, we can keep balanced performance among kernels and achieve high system-level performance. The findings in this paper are the foundation to realize the resource control. The initial thought is to put a resource control module on CPU, it can separate from the database operations on GPU so that efficient computation is ensured. We can set a time period to get the feedback, once the feedback is analyzed on CPU, the control signal will be sent to each kernel to change their resources received. This will be an important step towards a full-fledged G-SDMS.

Acknowledgment

The reported work is supported by an award (IIS-1253980) from the National Science Foundation (NSF) of U.S.A.

REFERENCES

[1] CUDA STREAMS BEST PRACTICES AND COMMON PITFALLS. http://on-demand.gputechconf.com/gtc/2014/presentations/S4158-cuda-streams-best-practices-common-pitfalls.pdf.

[2] UIUC ECE408 Lab 6: Histogram. https://github.com/shuotian/ECE408/tree/master/lab6-histogram.

[3] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 12(2):120–139, Aug. 2003.

[4] 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 *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 519–530, New York, NY, USA, 2010. ACM.

[5] P. Bakkum and K. Skadron. Accelerating sql database operations on a gpu with cuda. In *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*, GPGPU '10, pages 94–103, New York, NY, USA, 2010. ACM.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, F. Reiss, and M. A. Shah. Telegraphcq: Continuous dataflow processing. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 668–668, New York, NY, USA, 2003. ACM.

[7] S. Chen, Y.-C. Tu, and Y. Xia. Performance analysis of a dual-tree algorithm for computing spatial distance histograms. *The VLDB Journal*, 20(4):471494, 2011.

[8] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner. Spinning relations: High-speed networks for distributed join processing. In *Proceedings of the Fifth International Workshop on Data Management on New Hardware*, DaMoN '09, pages 27–33, New York, NY, USA, 2009. ACM.

[9] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[10] T. K. Group. OpenCL. https://www.khronos.org/opencl/.

[11] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 383–394, New York, NY, USA, 2005. ACM.

[12] 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.

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

[14] H. Nguyen. *Gpu Gems 3*. Addison-Wesley Professional, first edition, 2007.

[15] NVIDIA. CUDA C Best Practices Guide, Version 5.0. https://developer.nvidia.com/category/zone/cuda-zone.

[16] NVIDIA. CUDA C/C++ Streams and Concurrency. http://on-demand.gputechconf.com/gtc-express/2011/presentations/StreamsAndConcurrencyWebinar.pdf.

[17] NVIDIA. CUDA home. http://www.nvidia.com/object/cuda_home_new.html.

[18] NVIDIA. GPGPU.org. http://gpgpu.org.

[19] NVIDIA. GPU Applications. http://www.nvidia.com/object/gpu-applications.html.

[20] NVIDIA. GTX 680 Whitepaper. http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf.

[21] NVIDIA. Kepler GK110 Whitepaper. http://www.nvidia.com/content/PDF/kepler/NVIDIA-Kepler-GK110-Architecture-Whitepaper.pdf.

[22] H. Paul. Fiber channel architecture, Dec. 27 2005. US Patent 6,981,078.

[23] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle. Constant-time query processing. In *Data Engineering, 2008. ICDE 2008. IEEE 24th International Conference on*, pages 60–69, April 2008.

[24] B. Sherbin. Live: Jen-Hsun Huang at GTC 2013. http://blogs.nvidia.com/blog/2013/03/19/live-jen-hsun-huang-at-gtc-2013/.

[25] E. A. Sitaridi and K. A. Ross. Ameliorating memory contention of olap operators on gpu processors. In *Proceedings of the Eighth International Workshop on Data Management on New Hardware*, DaMoN '12, pages 39–47, New York, NY, USA, 2012. ACM.

[26] Y.-C. Tu, A. Kumar, D. Yu, R. Rui, and R. Wheeler. Data management systems on gpus: Promises and challenges. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management*, SSDBM, pages 33:1–33:4, New York, NY, USA, 2013. ACM.

[27] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann. Predictable performance for unpredictable workloads. *Proc. VLDB Endow.*, 2(1):706–717, Aug. 2009.

[28] M. Zukowski, S. Héman, N. Nes, and P. Boncz. Cooperative scans: Dynamic bandwidth sharing in a dbms. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 723–734. VLDB Endowment, 2007.