Two-Stage Modeling and Control of Concurrent Tasks in a Multi-Kernel GPGPU Environment

Hao Li, Yi-Cheng Tu
Department of Computer Science and Engineering
University of South Florida
Tampa, FL, U.S.A.
{haoli1,tuy}@mail.usf.edu

Bo Zeng
Department of Industrial Engineering
University of Pittsburgh
Pittsburgh, PA, U.S.A.
bzeng@pitt.edu

***Abstract*—The unrivaled computing capabilities of modern GPUs meet the demand of processing massive amounts of data seen in many application domains. While traditional HPC systems support applications as standalone entities that occupy entire GPUs, there are systems where multiple tasks are meant to be run at the same time in the same device. To that end, system-level resource management mechanisms are needed to fully unleash the computing power of GPUs in large data processing. In our previous work, we designed and implemented a push-based DBMS named G-SDMS that supports concurrent data processing on GPUs under NVidia's CUDA framework. This paper focuses on resource allocation of multiple GPU applications towards optimization of system throughput in the context of systems such as G-SDMS. Our approach is to control the launching parameters of multiple GPU kernels as provided by compile-time performance modeling as kernel-level optimization. Specifically, we construct a multi-dimensional knapsack model to maximize concurrency in a multi-kernel environment. We present an in-depth analysis of our model and develop an algorithm based on dynamic programming technique to solve the model. We prove the algorithm can find optimal solutions (in terms of thread concurrency) to the problem and bears pseudopolynomial complexity on both time and space. Such results are verified by extensive experiments running on our microbenchmark that consists of real-world CUDA kernels. Furthermore, solutions identified by our method also significantly reduce the total running time of the workload, as compared to sequential and random scheduling schemes. We also present a more general pre-processing model with batch-level control to enhance performance.**

***Index Terms*—push-based systems, GPU, GPGPU, CUDA, resource model**

I. Introduction

With the recent development of semiconductor technology, the number of processing units integrated on a chip increases rapidly, resulting in massively parallel computing capability. Many-core hardware systems such as Intel Xeon Phi co-processors and Graphics Processing Units (GPU) are becoming more and more popular. As shown in Fig. 1, the single precision peak performance of the latest NVidia GPU reaches 6144 GFLOPS and the latest AMD GPU has 5914 GFLOPS. On contrary, the CPU only provides 354 GFLOPS, and the Intel Phi reaches 1200 GFLOPS. Such unrivaled computing power has made GPUs an indispensable component in today's high-performance computing (HPC) systems and shown great value in many compute-intensive applications.

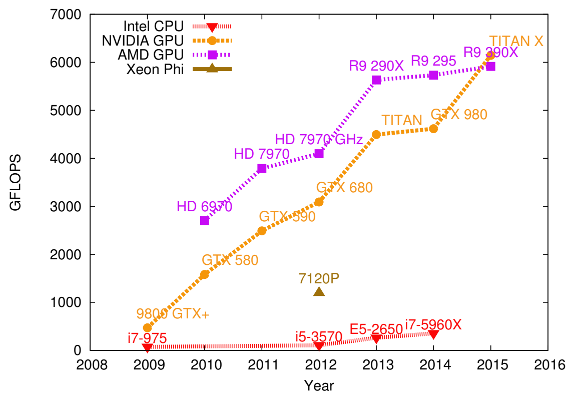

Fig. 1. Growth of computing capacity on CPU, Intel Phi, and NVidia/AMD GPUs. Data is extracted from www.techpowerup.com

The use of GPUs in application domains that are typically not heavy users of HPC resources is also explored. For example, novel database management system (DBMS) architectures have been proposed to meet the challenges of handling "big data." Among others, DBMSs with a *push-based* query engine design has gained much momentum [1]. Unlike traditional relational DBMSs, the core of a push-based DBMS follows a stream-based design in its data input mechanism. In particular, it creates a shared I/O stream to deliver data to all running queries simultaneously, while traditional DBMSs ("pull-based" system) retrieve the needed data from storage for each individual query. Due to single I/O stream and minimization of random I/Os, push-based DBMS successfully moves the performance bottleneck from I/O to computation. The high-speed data streams in such systems require large computing capabilities and parallel hardware such as GPUs fills this gap. General-purpose programing frameworks such as the compute unified device architecture (CUDA) and Open Computing Language (OpenCL) also made implementing push-based DBMSs for GPUs a feasible task. Our work reported here focuses on a CUDA environment. In our previous work [2], we proposed a GPGPU-based Scientific Data Management System (G-SDMS) that uses CUDA-supported GPUs as the platform for query processing in a push-based manner. G-SDMS can be viewed as a middleware that provides query processing/optimization and resource management functionalities on top of CUDA. As compared to most of today's research on GPUs with the assumption of processing computational kernels one at a time, our work on G-SDMS is system-oriented and imposes unique challenges.

A key challenge is to support concurrent execution of heterogeneous applications (i.e., queries). At runtime, data will be loaded into the memory chunk by chunk and all queries have to be processed against the *in situ* chunk before the next chunk is loaded. Such systems are generally optimized towards data processing throughput therefore maximizing resource utilization is essential. In a CPU-based environment, (main) memory and CPU cycles are often the only involved resources, and much work has been done in the context of data stream systems [3]. The GPUs, on the other hand, have a complex architecture (Section II-B) that provides abundant resources under more categories (e.g., registers, shared memory, blocks, threads, etc.). Such complexity brings opportunities for improved application performance, and also necessitates non-trivial modeling and algorithmic techniques in system design and implementation.

CUDA allows a kernel to run with a large number of threads and blocks. The limited total resource, however, means the threads will have to take turns to be executed on the hardware. To run a thread in a CUDA kernel, a certain amount of resource under different categories is required. In a multi-kernel environment such as G-SDMS, it is essential to *determine how many threads for each kernel should be launched simultaneously such that the overall performance is the best*. Being the main objective of our study, this problem is non-trivial due to the multiple types of resources involved. Let us illustrate this with a simple example (Fig. 2) with two kernels bearing different resource use patterns. If we schedule the kernels sequentially (as in a typical HPC resource scheduler), we can run 10 threads of kernel I or kernel II, as the concurrency is determined by the largest single-resource consumption (e.g., 10% of resource B for kernel I). If the latency of running such threads is T for both kernels, this gives a throughput of $10/T$. However, if we schedule both kernels concurrently, we could run 8 threads of both kernels I and II at the same time, leading to a throughput of roughly $16/T$. Obviously, by scheduling kernels with complementary resource utilization patterns together, we avoid hitting the limit of a bottlenecking resource quickly. The problem can be very complex by considering more general cases with more resources and kernels involved.

The CUDA framework achieves hardware-level kernel concurrency via a mechanism named *CUDA Stream* (not the input data stream mentioned earlier). Given sufficient resources, many CUDA streams can run simultaneously with each stream containing one or more kernels. In CUDA, all threads in a block are scheduled to run on the same resource pool (i.e., the multiprocessor) thus a block can be conceptually viewed as a basic unit for studying our problem. On the other hand, CUDA allows a kernel to be launched with user-specified parameters and such parameters determine the actually resource use of

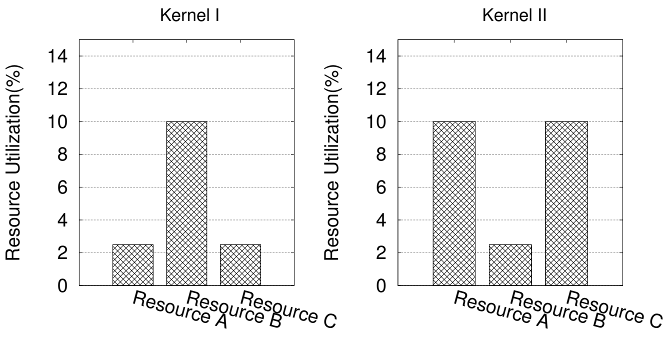

Fig. 2. Normalized resource use per thread of two different kernels

each block of threads at runtime. Therefore, our problem essentially becomes: *how to set the runtime parameters of kernels in different CUDA streams to achieve the best throughput*? To the best of our knowledge, optimization of multi-kernel parameters has not been studied before. As the first work on this topic, we aim at developing rigorous solutions by breaking this problem to two-stages. Specifically, we develop a kernel-level optimization model towards largest thread concurrency with the runtime parameters of all kernels as input, and a batch-level optimization model towards smallest batches to run all the kernels. We identify the kernel-level optimization problem as a variation of the multi-dimensional knapsack problem. Via thorough analysis of the model structure and features of CUDA runtime system and CUDA streams, we are able to reduce the number of dimensions of the constraints in the original model. We then develop an algorithm based on dynamic programming to solve the modified model. We prove the algorithm can find optimal solutions (in terms of thread concurrency) to the problem and bears pseudopolynomial complexity on both time and space. Moreover, the batch-level optimization problem can be identified as a variation of the multi-dimensional bin-packing problem. Via transforming the model into a cutting stock problem, we are able to develop an algorithm based on Gilmore and Gomory algorithm [4] [5]. Such results are verified by extensive experiments running on our microbenchmark that consists of real-world CUDA kernels. Furthermore, solutions identified by our method also significantly reduce the total running time of the workload, as compared to simple and random solutions.

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 kernel-optimization model and the analysis, simplification of the model, and a more general batch-optimization model; in Section IV-A, we present the dynamic programming algorithm of the kernel-optimization model as well as the column generation algorithm of the batch-optimization model; Section V describes experimental validation of our solutions; we compare our study with related work in Section VI and conclude the paper in Section VII.

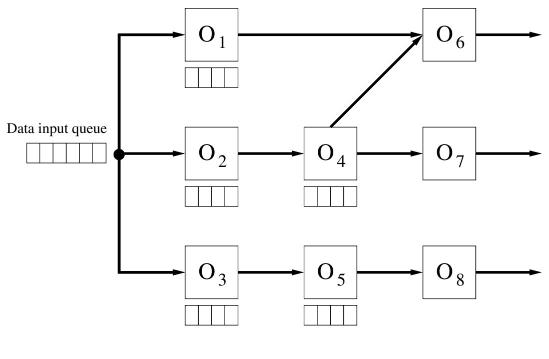

Fig. 3. An example of G-SDMS runtime query network [7]

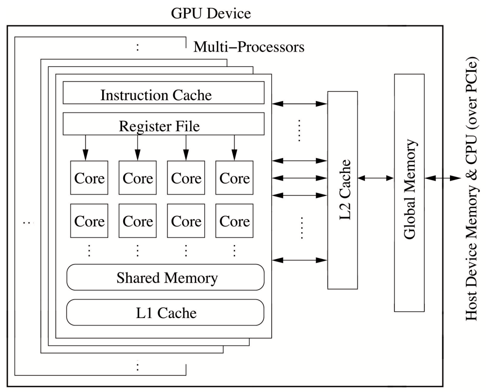

Fig. 4. Architecture of a typical NVidia GPU [7]

II. Background

A. The G-SDMS System

G-SDMS is a push-based DBMS that embraces the computing power of GPUs. At runtime, the query engine of G-SDMS creates a network of operators and pushes data into this network. Such a network is much like the relational algebraic query plans supported in traditional DBMSs, the difference is that multiple query pipelines can be combined to form a complex workflow. The output of an upstream operator is the input of the next one(s) in the workflow (Fig. 5). Each operator has a memory buffer (queue) to hold the intermediate results. Operators supported by the system include relational operators, aggregates, and user-defined functions. G-SDMS retrieves data via a scan-based I/O framework, combining data processing load into the scanning process to thinly spread cost to many queries. Since random I/O is minimized, G-SDMS can receive input data at a high rate (e.g., up to 4.8 GB/s with dual FibreChannel interfaces [6]). This is also the main motivation of using GPUs to process the queries. Since all kernels share the same input data stream, kernel concurrency is an important requirement in the implementation of G-SDMS. In practice, data is loaded in chunks into global memory, and all queries have to finish their execution with the current chunk before the next chunk of data can be loaded. An obvious system design goal is to maximize throughput of data processing.

B. Typical GPU Architecture

A modern GPU is a special hardware that encapsulates many processing units together to provide high parallel computing capability. As shown in Fig. 4, main components of a GPU includes: (1) A number of *Multi-Processors* (MP) that each groups tens of processor cores together. The cores execute threads in a Single-Instruction-Multiple-Data (SIMD) fashion; (2) *Multi-level memory*. Of largest amount (e.g., 12GB for the Titan X Pascal) is the *global memory*, which can be accessed in parallel by cores in different MPs. The bandwidth of global memory can be as high as 480GB/s [8]. GPU also offers high-speed on-chip cache called *shared memory* (SM) similar to L1 cache, and each MP has its own SM with a size up to 96KB [9]. SM is user programmable in GPU code and is not visible to the CPU code. Within each MP, there are also other memory: the read-only data cache 24 KB [9] and the nonprogrammable L2 cache with a certain size (3 MB) and a bandwidth smaller than that of SM [10].

In the CUDA programming framework, a function to be executed in a parallel way is called a CUDA *kernel*. A kernel can be spawned with a large number of computational *threads*. Threads for a kernel is called a *grid* and the grid is divided into *blocks* that each contains the same number of threads to be executed on a single MP. On the other hand, multiple blocks can be run on the same MP, and one MP can process up to 32 blocks. It is device driver's responsibility to schedule the blocks to use the different MPs. Threads are scheduled as groups of 32 threads called *warps*. The entire global memory can be accessed by any thread in any MP, shared memory and registers of each MP can only be accessed by the thread of the same MP. CUDA provides a mechanism 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. However, NVidia does not reveal much detail about the internal mechanism for kernel scheduling in CUDA streams. Our previous work [7] studied kernel scheduling policies of CUDA streams, the findings of that work form the foundation of this paper.

III. Multi-kernel Optimization

A GPU contains different types of resources including physical hardware units and software constraints. In our previous work [7], we have identified three types of resources / constraints that affect the performance in a single-kernel setup: *registers*, *shared memory*, and *maximum warps allowed in an MP*. In a multi-kernel environment, there is one additional constraint we have to consider: *total blocks of all the kernels allowed to run simultaneously in an MP*.

As long as all the resources are sufficient, multiple kernels can be executed at the same time. For any kernel, its resource consumption can be controlled at runtime by changing the launching parameters in the host (CPU) code. CUDA allows three parameters in launching a kernel: *total number of blocks*, *block size* (i.e., number of threads in a block), and *shared memory consumption* as an optional parameter. Note that the product of the first two is actually the total number of threads. The third parameter is generally not specified, as programmers often hardcode the total shared memory use to match the size of a chunk of input data. Therefore, in this paper, we only consider *total number of blocks* and *block size* as the controls we apply to affect resource consumption. Note that, in CUDA, each thread gets its own set of registers while the shared memory is shared by all threads in the block. Therefore, by changing the block size, we can control the register use per block and shared memory use among all blocks of a kernel. Needless to say, the block size itself directly determines the number of warps per block.

Before we start developing our optimization model, it is worth mentioning that the problem of optimizing single-kernel performance was solved in our previous work [7]. In particular, we build a model to quantify the total number of threads that can be executed simultaneously (i.e., *occupancy* in CUDA terminology) as an indication of kernel performance. Based on this model, we can accurately predict kernel performance under any block size and then pick the one with highest performance to run. Although some ideas can be borrowed, the same problem under a multi-kernel environment is much more complicated. First, the modeling method based on a series of discrete functions for the single-kernel situation will only yield models that are too complicated to handle; Second, kernel scheduling rules among different CUDA streams are not revealed by NVidia – such information is vital for the development of our optimization model; Finally, with multiple kernels, the solution space of the optimization problem increases exponentially. This places stringent requirements on the efficiency of the algorithm(s) for solving the optimization.

Fortunately, our previous work [7] built a solid foundation for multi-kernel modeling by identifying basic rules of CUDA stream scheduling. Here we briefly present one scheduling rule that is most relevant to our modeling. The rule says: *CUDA scheduler always takes as many MPs as possible in scheduling the different blocks of a kernel.* For example, if we have two kernels A and B, both of them have 14 blocks, and there are 14 MPs. Each MP can run two blocks of A, or one block of B, or one block of A and one block of B at the same time. Based on the rule above, CUDA scheduler will schedule one block of A to each MP then schedule one block of B to each MP, now there are one block of A and one block of B running on each MP. Without the rule, scheduler will put as many as blocks of A into MP, which makes each of seven MPs has two blocks of A, and each of the rest seven MPs has one block of B, rest of seven blocks of B need to wait for the next round to be scheduled.

According to the above rule of CUDA stream scheduling, our model can target one MP, the final result of each kernel is the number of MPs times optimized blocks. In particular, we divide the total threads of each kernel by the number of MPs, using the result as the total thread in our model. In this way, we make sure each MP has same amount and portion of

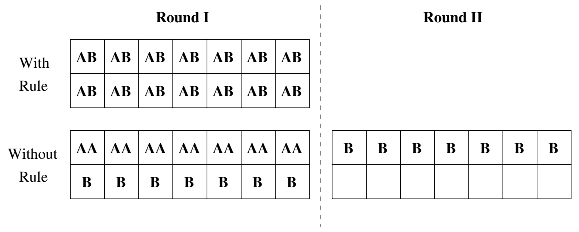

Fig. 5. An example of CUDA stream scheduling

kernels. We also assume that *there is at least one solution for all the kernels to fit in the MPs*. Otherwise, the left kernels need to wait for another round to run. If there is a situation that combined MPs can hold the total threads of all the kernels while a single MP cannot (i.e. the number of kernels exceed the maximum number of blocks in an MP), we group two MPs as a unit, which means we divide the total threads of each kernel by half number of MPs.

A. Kernel-level Optimization Model Development

The desirable optimization goal of the multi-kernel resource allocation problem is total running time of all kernels. However, it is difficult (if possible at all) to derive a model that maps the launching parameters of multiple kernels to running time. This is mainly due to the lack of low-level details of CUDA runtime environment. To the best of our knowledge, no one has done research on performance modeling in a multi-kernel GPU environment. In this paper, we set the optimization goal to be *maximizing concurrency*, which is defined as the total number of threads that can be scheduled to run at the same time. Such a goal is meaningful for two reasons: (1) it is a direct measurement of throughput; and (2) as shown in our previous research [7], concurrency has a strong (negative) relationship with kernel running time.

To achieve maximum concurrency on a GPU, we need to get the most threads (of different kernels) running in an MP (Eq. (1)). The problem can be formulated as the following integer programming statement:

$$\begin{aligned}
\textbf{Maximize} \quad & \textstyle\sum_i \sum_j 32 j x_{ij} b_i && (1)\\
\textbf{subject to} \quad & \textstyle\sum_i \sum_j 32 j x_{ij} b_i r_i \le R && (2)\\
& \textstyle\sum_i \sum_j j x_{ij} b_i \le W && (3)\\
& \textstyle\sum_i b_i s_i \le S && (4)\\
& \textstyle\sum_i b_i \le B, b_i \in \mathbb{Z}_+ && (5)\\
& \textstyle\sum_j x_{ij} = 1, \forall i && (6)\\
& \textstyle\sum_j 32 j x_{ij} b_i \ge c_i, \forall i && (7)\\
& x_{ij} \in \{0,1\}, \forall i, j && (8)
\end{aligned}$$

In the above statement, i is the index of a kernel, j is the index of all the possible choices of block size for a single kernel. Since CUDA schedules 32 threads (a warp) as a unit, we use warp instead of thread in this model, $32j$ stands for thread number of a block for a single kernel. To be specific, CUDA allows a block to have up to 32 warps in it therefore we have $t_{i,j} = j$ ($j \in [1, 32]$), x is a binary number to represent which block size is chosen in a solution (Eq. (6) and Eq. (8)). The quantities b_i and s_i stand for the block number and shared memory use for kernel i, respectively. r_i is the per-thread register number for the same kernel. The constants R, W, S, and B stand for the total number of registers, warps, shared memory and blocks of an MP in the GPU. The reason for having Eq. (7) is as follows: for most CUDA programs, the total number of threads c_i is fixed by the programmer to cater to the data size, changing total blocks and block size are the actually the same: when block size increases by a factor of f, total number of blocks will decrease by the same factor f, however, the data size of a kernel can hardly be the multiplier of 32, thus we use $\geq$ instead of $=$ in Eq. (7). For each kernel i, r_i, c_i, and s_i are constants thus the inputs to the optimization problem. On the other hand, the solution to the optimization contains quantities x_{ij} and b_i.

Remark:
(1) Note that the aforementioned formulation has an interesting feature: according to Eq. (7), any **feasible** solution to the formulation actually provides us a schedule with the maximal concurrency. However, due to a large number of $0-1$ variables (x_{ij} for all i and j) and the other 6 non-trivial constraints, it is an NP-hard problem to locate a feasible solution. To address such challenges, we discuss model simplification and transformation in Section III-B. Such transformation results in the development of a pseudo-polynomial algorithm in solving the problem (Section IV-A);
(2) Eq. (7) determines that solutions to the formulation do exist. In other words, we can find a set of launching parameters for every kernel such that they can all be processed by the GPU at the same time. In Section III-C, we will remove this assumption and introduce a batch-level model to solve the more general situation of the problem.

B. Kernel-optimization Model Analysis and Simplification

By studying the structure of the current model, we realize it is a flavor of the well-known *multidimensional knapsack problem* (MKP). An MKP is NP-hard even when the number of constraints is only one [11]. It is easy to see our model is equivalent to an MKP with $m = 4$ therefore it is also an NP-hard problem. Moreover, the original model involves binary variable x_{ij} as part of the solution and as many as seven constraints. Therefore, the original formulation is difficult to analyze or to compute. To remedy that, we aim to transform the model into a form that is easier to handle via considering the actual environment where our problem is defined. Specifically, we derive a reformulation with a much smaller number of variables and constraints.

Our first goal is to eliminate the binary integer x_{ij}. As mentioned before, CUDA schedules threads in groups of 32 (i.e., a warp). For example, if we launch a kernel with 240 threads, the CUDA runtime framework will actually launch 8 warps for this kernel (with the last warp containing empty threads in this case). Therefore, we use warp number w_i ($w_i = \frac{t_i}{32}$) to replace jx_{ij}, and the value of w_i ranges from 1 to 1024/32 = 32 (since the maximum block size is 1024). As a result, the total thread number of a kernel has a ceiling of the total threads in the assigned warps, Eqs. (3) and (7) become:

$$\sum_i w_i b_i \leq W \tag{9}$$
$$32 w_i b_i \geq c_i, \forall i \tag{10}$$

We then aim at removing some of the constraints. As we mentioned, any feasible solution to the original model is actually a solution that gives us the maximal concurrency. Hence, it suffices to develop a new model that aims at finding one feasible solution to the original model with fewer constraints. Note that based on Eq. (10), we can easily calculate the results of $\sum_i \sum_j 32 j x_{ij} b_i r_i$ given any problem inputs. Thus, the constraint about registers in Eq. (2) only serves the purpose of determining if there is a feasible solution, and we can remove it from the problem statement. Now we have the newly derived constraints shown in Eqs. (9) and (10) plus the remaining constraints Eqs. (4) and (5).

With the above constraints, we further reduce the level of difficulty in solving the problem via a technique that modifies the object function. This can be done by transforming a constraint into the object function. In particular, we can choose any of the remaining constraints as our new object function. In our problem, we pick Eq. (4) since it is the only one that has a unique coefficient s_i. Consequently, the new problem formulation becomes:

$$\textbf{Minimize} \quad \sum_i b_i s_i \tag{11}$$
$$\textbf{subject to} \quad \sum_i b_i \leq B \tag{12}$$
$$\sum_i w_i b_i \leq W \tag{13}$$
$$32 w_i b_i \geq c_i, \forall i \tag{14}$$
$$b_i \in \mathbb{Z}_+ \; \forall i. \tag{15}$$

Remark:
(1) *Equivalence*: if the optimal value of the new formulation is less than or equal to total shared memory S, the corresponding optimal solution is feasible to the original formulation, i.e., a schedule with maximal concurrency;
(2) *Simplicity*: although this reduced formulation deals with general integer variable b_i, we have way fewer discrete variables, along with only three non-trivial constraints, which indicates its computational burden might not be heavy in practice, if a well-designed algorithm can be developed. Note that the quantities b_i and w_i are the solutions and all other quantities are inputs to the model.

After a series of transformations without adding new assumptions, the problem becomes one to *minimize the total shared memory use of all kernels*. Intuitively, minimizing shared memory use of one kernel will also minimize its block number so that there are more space left for remaining kernels in dimension of B (see Eq. (4) in original model).

C. Batch-level Optimization Kernel for More a General Situation

As we mentioned, our kernel-optimization model is for a bunch of kernels that can all fit in an MP. However, there could be more general scenarios in which an MP cannot accommodate all kernels due to resource constraints. For such problems, our solution is to run all the kernels in different batches, each batch will fit in an MP. In each batch, we solve the below model to get a batch-level solution as a pre-precessing. Then the key problem becomes *how to determine the membership of each batch*. Specifically, the problem can be formulated as follows:

$$\textbf{Minimize} \quad G = \textstyle\sum_k y_k \tag{16}$$
$$\textbf{subject to} \quad G \geq 1 \tag{17}$$
$$\textstyle\sum_i w_i x_{ik} \leq W_{y_k} \tag{18}$$
$$\textstyle\sum_i r_i x_{ik} \leq R_{y_k} \tag{19}$$
$$\textstyle\sum_i s_i x_{ik} \leq S_{y_k} \tag{20}$$
$$x_{ik} \in \{0, 1\}, \forall i, k \tag{21}$$
$$y_k \in \{0, 1\}, \forall k \tag{22}$$

In this problem, we still target the maximum concurrency, i.e., we want to pack as many kernels as possible in a batch, thus the number of batches G is minimized, as shown in Eq. (16). Each G has the same maximum capacity, *i.e.,* total warp numbers W, register numbers R, and shared memory S. Here, k is the index of a batch, y_k is a binary variable where $y_k = 1$ if bin k is used, and i is the index of a kernel, x_{ik} is a binary variable setting to 1 if kernel i is put in batch k. Same as the model described earlier, the quantities r_i, w_i and s_i stand for the register number (per thread), warp number, and shared memory use for kernel i, respectively. For each kernel i, w_i, r_i, and s_i are constants thus the inputs to the optimization problem, the solution to the optimization problem contains quantities x_{ik}.

The above pre-processing model is a three-dimensional Bin Packing Problem (3D-BPP), which is strongly NP-hard [12]. Silvano *et al.* [13] proved that the lower bound of Bin Packing Problem is $\frac{1}{8}$, which is the asymptotic worst-case performance.

IV. Solving the Optimization Problem

In this section, we present algorithms to focus on solving the kernel-optimization model shown in Eq. (11) to Eq. (15) firstly, then presenting the algorithm of the batch-optimization model shown in Eq. (16) to Eq. (22).

A. Algorithm of Kernel-optimization model

In this section, we present algorithms to solve the simplified model shown in Eq. (11) to Eq. (15). Note that the new formulation is not a simple knapsack problem anymore. Indeed, because both w_i and b_i are variables, the formulation in Eq. (11)- Eq. (15) is a *quadratic general knapsack problem* (QGKP), which is also an NP-hard problem [14]. Hence, a brute-force algorithm would have to search through all $O\big(BW\big)$ possible combinations with respect to a total of n kernels, giving a total time complexity of $O\big((BW)^n\big)$, and this is clearly infeasible for practical instances.

However, the transformation of the original problem into QGKP enabled us to develop a (practically) efficient algorithm based on the *dynamic programming* approach. Dynamic Programming is a well-known divide-and-conquer technique to solve optimization problems. The idea is to transform a complex problem into relatively simple sub-problems. The algorithm examines previously solved sub-problems and combine the solution to give a best solution for a slightly larger sub-problem.

Applying dynamic programming to knapsack problem is to essentially trade time with space. We can use a table to record decisions made for sub-problems and recursively look up the table when involving previous decision. Following our discussions in Section III-B, we should use a three-dimensional table since there are three variables to be considered: the n kernels, total blocks ranging 0 to B, and total warps ranging 0 to W. The main task of the algorithm is to compute the value of a cell (i, b, w) in this table, where i is the kernel number, b is the block number of kernel i, and w is the warp number of kernel i, respectively. Here cell value (i, b, w) stands for the minimum total shared memory used of any subset of kernels 0 to i under targeted block number b and targeted warp number w. The key feature of the algorithm is that we only need to consider local choices in the table. In particular, the following result helps us drastically reduce the complexity of the table.

Theorem 1. *For a particular kernel i, if b_i is fixed, an optimal choice of w_i can be obtained as $w_i = \lceil \frac{c_i}{32 b_i} \rceil$.*

Proof. Note that to satisfy Eq. (14), w_i must be greater than or equal to $\lceil \frac{c_i}{32 b_i} \rceil$. Also, the smaller w_i, the smaller left-hand-side of Eq. (13). So, it would be optimal to set $w_i = \lceil \frac{c_i}{32 b_i} \rceil$. □

Hence, in the remainder of this paper, we simply set $w_i = \lceil \frac{c_i}{32 b_i} \rceil$ when b_i is available. Moreover, our dynamic programming algorithm can be simplified into a form similar to that for the general knapsack problem. Specifically, let $V[i, b, w]$ be the objective value considering up to i-th kernel with total b blocks and W warps. The Bellman equation is

$$V[i, b, w] = min_{b_i = 1, \ldots, B}\{V\big[i-1, b - b_i, w - w_i b_i\big] + s_i b_i\}$$

where s_i is shared memory usage per block of kernel i. Note that whenever b or w causes the solution infeasible, we will set the corresponding V to ∞.

Details of the algorithm to solve our problem can be seen as pseudocode in Algorithm 1. After we compute all the entries of V, $V[n, B, W]$ will contain the minimum shared memory use achieved by the solution. Meanwhile, another array P holds the solutions to the sub-problems and $P[n, B, W]$ is our solution. With the principle of optimality carried in the general knapsack problem, the correctness of the algorithm is shown as follows.

Algorithm 1: The Dynamic Programming Algorithm

```
1: for $b \leftarrow 0$ to $B$ do
2:     for $w \leftarrow 0$ to $W$ do
3:         $V[0, b, w] \leftarrow 0$
4:         $P[0, b, w] \leftarrow \phi$
5:     end for
6: end for
7: for $i \leftarrow 1$ to $n$ do
8:     $V[i, 0, 0] \leftarrow \infty$
9:     $P[i, 0, 0] \leftarrow \phi$
10: end for
11: for $i \leftarrow 1$ to $n$ do
12:     for $b \leftarrow 1$ to $B$ do
13:         $Q_b \leftarrow V\big[i-1, B-b, W-wb\big] + s_i b$
14:     end for
15:     $V[i, b, w] \leftarrow min_{b=1,\ldots,B}\{Q_b\}$,
        /* denote optimal $b$ as $b^*$ */
16:     $P[i, b, w] \leftarrow P\big[i-1, B-b^*, W-w_i b^*\big] \cup (i, b^*)$
17: end for
```

Theorem 2. *Algorithm 1 terminates with an optimal solution, i.e., the value of $V[n, B, W]$ is optimal.*

Proof. We prove the theorem via induction.
(1) When there is one kernel ($n = 1$), we have

$$\begin{aligned} V[1,1,w] &= min\{V[1,1-1,w], \\ &\quad V[0,B-1,W-w]+s_1\} \\ &= min\{\infty, 0+s_1\} = s_1 \end{aligned}$$

For $V[1,1,w]$, we get the optimal value s_1.

$$\begin{aligned} V[1,2,w] &= min\{V[1,2-1,w], \\ &\quad V[0,B-2,W-w]+2s_1\} \\ &= min\{s_1, 0+2s_1\} \end{aligned}$$

For $V[1,2,w]$, we can get the optimal value by comparing $V[1,1,w]$ and $2s_1$.

$$\begin{aligned} V[1,B,w] &= min\{V[1,B-1,w], \\ &\quad V[0,B-B,W-w]+s_1 B\} \end{aligned}$$

If we know the optimal value of $V[1, B-1, w]$, we can get the optimal value of $V[1, B, w]$ by comparing $V[1, B-1, w]$ and $0 + B \times s_1$. Deriving it one by one, we can get the optimal value of $V[1, 1, w]$, then value of $V[1, 2, w]$ based on $V[1, 1, w]$, $\cdots$, and value of $V[1, B, w]$ based on $V[1, B-1, w]$. Thus for each b from 1 to B, we get the optimal value.
(2) When there are two kernels ($n = 2$), we have

$$\begin{aligned} V[2,1,w] &= min\{V[2,1-1,w], \\ &\quad V[1,B-1,W-w]+s_2\times 1\} \\ &= min\{\infty, V[1,B-1,W-w]+s_2\} \\ &= V[1,B-1,W-w]+s_2 \end{aligned}$$

From Step (1) we know $V[1, B-1, W-w]$ has an optimal value, so $V[2, 1, w]$ has the optimal value.

$$\begin{aligned} V[2,2,w] &= min\{V[2,2-1,w], \\ &\quad V[1,B-2,W-w]+s_2\times 2\} \\ &= min\{V[2,1,w], V[1,B-2,W-w]+2s_2\} \end{aligned}$$

Also from step (1) we know $V[1, B-2, W-w]$ has an optimal value, and $V[2, 1, w]$ has optimal value based on above proof. By comparing $V[2, 1, w]$ and $V[1, B-2, W-w]+s_2\times 2$ we can get the optimal value of $V[2, 2, w]$.

$$\begin{aligned} V[2,B,w] &= min\{V[2,B-1,w], \\ &\quad V[1,B-B,W-w]+s_2\times B\} \end{aligned}$$

Same as in step (1), we derive it one by one, we can get the optimal value of $V[2, 1, w]$, then value of $V[2, 2, w]$ based on $V[2, 1, w]$ and $V[1, B-1, W-w]$, $\cdots$, and value of $V[1, B, w]$ based on $V[2, B-1, w]$ and $V[1, B-B, W-w]$. Thus for each b from 1 to B, we can get the optimal value.
(3) The same approach shown in step (2) can be applied to cases $n = 3$ and beyond, and this concludes the proof. □

Time and space complexity: The complexity is clearly determined by the size of the dynamic programming table, which is $O(nBW)$. In practice, both n and B are small integers (i.e., $n \leq 32$ and $B \leq 16$ in the latest version of CUDA) thus this algorithm will have negligible cost. Similarly, the pre-processing stage takes $O(nBW)$.

B. Algorithm of Batch-optimization model

The 3D-BPP is a hardly NP-hard problem, to our knowledge no one has present an exact algorithm for it. We have developed a solution by applying classic Gilmore and Gomory algorithm [4] [5] after transforming this problem into a cutting stock problem. While the Gilmore and Gomory algorithm only deals with 1D bin-packing, we follow its philosophy of using column generation approach and decomposing our model into a master problem (cutting stock) and a sub-problem (pricing problem).

Our model (Eq. (16) to Eq. (22)) contains $k!$ symmetric solutions and there are many binary variables, that makes problem extremely hard. To make this problem simpler, we can transform it to a cutting stock problem: instead of focusing on which kernel is put in a particular part of a batch, we look at possible *patterns* used to put in a batch. The question is then changed to focus on how many times a particular pattern is used:

$$\textbf{Minimize} \quad Z = \textstyle\sum_j x_j \tag{23}$$
$$\textbf{subject to} \quad \textstyle\sum_j P_{ij} x_j \geq d_i, \forall i \tag{24}$$
$$x_j \geq 0 \tag{25}$$

In this model, i is the number of same kernels, j is the number of different types of patterns. x_j stands for number of jth pattern that has been used, P_{ij} stands for cutting pattern

Algorithm 2: The Column Generation Algorithm

```
1: Initialize patterns
2: repeat
3:     Substitute patterns into master problem Eq. (23), find $\pi$
4:     Solve sub-problem Eq. (26), get new pattern
5:     Add new pattern to master prbolem
6: until
7: $z_{sub} \geq 0$
```

that ith kernel used in jth pattern, d_i stands for demands of kernel i.

It is natural to consider Simplex Algorithm as the solution [15]. However, there are $2^i - 1$ patterns of the required i kernels [16]. Even if we had a way to generate all patterns, it is difficult to contain all variables into the algorithm. Thus for each iteration of in the Simplex Algorithm, we need to find the most negative column [5]. By defining a new sub-problem, we are able to find it.

$$\begin{aligned} \textbf{Minimize} \quad & z_{sub} = 1 - \textstyle\sum_i \pi_i P_{ij} && (26) \\ \textbf{subject to} \quad & \textstyle\sum_i P_{ij} w_i \leq W && (27) \\ \textbf{subject to} \quad & \textstyle\sum_i P_{ij} r_i \leq R && (28) \\ \textbf{subject to} \quad & \textstyle\sum_i P_{ij} s_i \leq S && (29) \\ & P_{ij} \in \mathbb{Z}^+ && (30) \end{aligned}$$

Here, π_i stands for the average demands of kernel i in this round of Simplex Algorithm. The sub-problem is a pricing problem as well as a three-dimensional knapsack problem, we can use dynamic algorithm similar to our algorithm in section IV-A and the complexity is $O(nWRS)$. Hence, the Column Generation Algorithm for solving our pre-processing model can be seen in above Algorithm 2.

V. Experimental Evaluation

A. Experimental Setup and Benchmark

We run all experiments in a workstation with an Intel E8400 CPU@3.0GHz, 8 GB of DDR3 1333MHz memory, one 300GB Seagate ST33204 hard disk, and one NVidia GeForce GTX TITAN X graphics card. The machine runs Ubuntu 14.04 OS and CUDA version 6.5.

In the experiments of simplified model, we compare the performance of our solution with two baselines:

(1) sequential execution of kernels, this simulates the behavior of a typical HPC resource schedulers where each application is treated as an independent process. In this setup, kernel parameters are set according to our previous work [7] to ensure best single-kernel performance;

(2) concurrent execution of kernels with randomly assigned kernel parameters. Since our work is the first one in dealing with multi-kernel resource management, this is best setup we can think of as a "simple" solution.

As to the benchmark, we build a kernel pool that consists of 13 real kernels, 11 of which are from NVidia CUDA libraries [17] and two are from our previous work [18]. We put each kernel to an individual CUDA stream. We generate workload in two different ways: (1) *fixed set*: we preorder all the 13 kernels and pick kernels following this order. Specifically, if there are m CUDA streams to fill, we put one kernel into each stream following the kernel order. After each kernel is put into a stream, we start from kernel 1 to fill the remaining streams till all streams are filled. (2) *random set*: we randomly pick a series of kernels to fill the CUDA streams. As compared to the fixed set, the random set simulates a more natural workload but may bring extra challenges due to higher uncertainty in workload composition. Each experiment runs 400 times, and a different combination of kernels is picked in each run. To be consistent to the assumptions made in our modeling process, we only run those combinations of kernels in which a feasible solution can be found. The total number of threads and the shared memory consumption are fixed for each kernel.

B. Experimental Results and Discussions of Simplified Model

Since we have proved (Section IV-A) that our algorithm will find the solutions with the largest number of active threads, discussions on experimental results will be focused on total running time of the workload. However, we want to first point out that in all experimental runs our solutions did reach the highest thread concurrency without an exception.

First, let us compare the performance of running kernels under workloads generated in the fixed set. In Fig. 6, we can see that, by applying our model, the minimum speedup against sequential execution ('sequential' hereafter) of 31 stream and 16 streams are 4.16x and 3.90x, respectively; the minimum speedup against concurrent execution with random kernel parameters ('random' hereafter) of 31 stream and 16 streams are 1.44x and 1.39x, respectively. Our model beats *sequential* in all cases while *random* some times finds solutions with better performance. Since our model targets maximum concurrency instead of actual total running time, it is not surprising that this has happened. While we admit this is a limitation of our approach, we also want to study the extent of such a limitation. Fig. 7 shows the distribution of the speedup against *random*: in 5.75% of all cases *random* finds better solutions under 31 streams, while this increases to 7.25% under 16 streams. Taking a closer look at such cases, we found that they also reached the highest kernel concurrency. Obviously, the lack of a deterministic relationship between concurrency and running time is the main reason for such results.

Now let us study the performance of our model under workloads generated by picking a random subset of the 13 kernels (random set). Such results are presented in Figure 8. We can see that the maximum and average speedup is not much different from what we found in the fixed set workloads. The main problem, however, is that the percentage of cases in which the *random* solution finds better solutions increases: in 31 streams, this number is 14.5% and it further increases to 20.75% in 16 stream experiments. By plotting the distribution

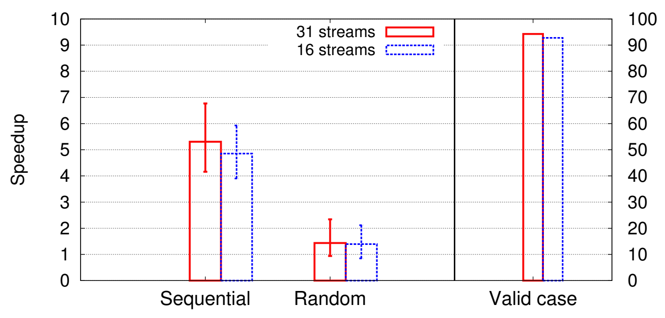

Fig. 6. Speedup over *sequential* and *random* solutions and the ratio of cases in which our solution shows better performance than *random*. Workloads are generated as a fixed set

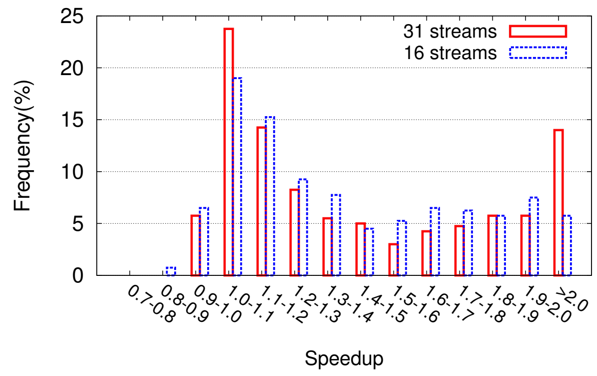

Fig. 7. Distribution of the speedup of our model against *random* in running workloads generated as a fixed set

of the speedups in Fig. 9 and comparing it with Fig. 7, it is clear that more cases fall into the low speedup range for the random set experiments. Notably, there are quite a few cases come with speedup of only 0.7 - 0.9, especially for the 16 stream case. To further explore the reason, we looked at the resource consumption patterns of the kernels in both the fixed set and random set experiments. We found out that the latter, by picking only a subset of the 13 kernels, renders more kernels with similar (or identical) demand on different

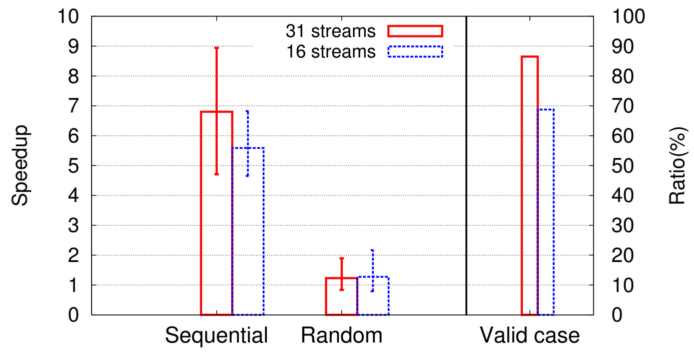

Fig. 8. Speedup over *sequential* and *random* solutions and the ratio of cases in which our solution shows better performance than *random*. Workloads are generated as a random set

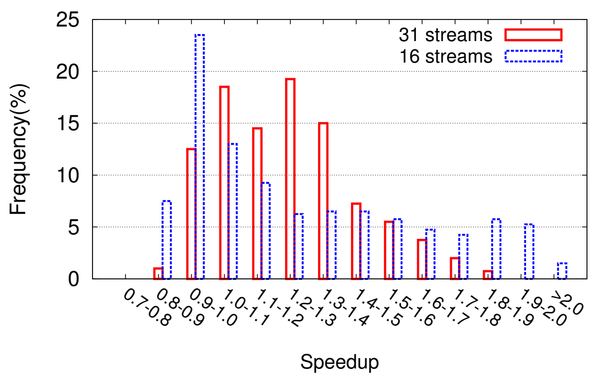

Fig. 9. Distribution of the speedup of our model against *random* in running workloads generated as a random set

TABLE I
SYNTHETIC KERNELS PARAMETERS

Kernels	I	II	III	IV	V
Register Number	8	16	24	32	40
Shared Memory	24576	12288	6144	4608	1536

resources (i.e., with bars of similar height in Fig. 2). As a result, the kernels all have similar resource use patterns and the random algorithm has a better chance in finding a good solution (i.e., Knapsack problem becomes easier with homogeneous items). When the number of CUDA streams is smaller, this becomes a more serious problem. In contrast to that, the fixed set experiments chose all kernels and show very different resource consumption patterns. This leads to much fewer optimal solutions (e.g., Knapsack problem becomes harder with heterogeneous items) and the advantage of our approach is shown in more cases. To verify such a hypothesis, we developed a workload that consists of five kernels with arbitrarily irregular resource consumption patterns (Table I). The kernel uses small number of registers has large amount shared memory usage, and vice versa. By this, we intentionally narrow down the subset of optimal/good solutions in the search space. The experimental results support our hypothesis (Fig. 11): among the 1,000 runs, our solution wins in 957 cases, and most of the other cases show a speedup within $(0.9, 1.0)$. The average speedup among all cases is 1.43x and the maximum speedup reaches 3.78x as shown in Fig. 10.

C. Experimental Results and Discussions of batch-optimization Model

In the experiments of batch-optimization model, we use four sets of different setups as in Table II shown. Set A and B are simply executed in sequential and concurrent order respectively without using any of our models, Set C is executed in concurrent order with just applying our kernel-optimized model, Set D is executed in concurrent oder with using both our batch-optimized model and kernel-optimized model. We compared Set D with three other sets. As to the benchmark, we use the same kernel pool and randomly pick 39, 52, 65 kernels

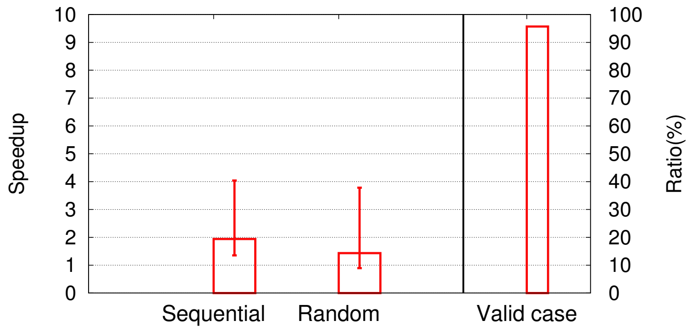

Fig. 10. Results of running workloads generated from synthetic kernels as a random set

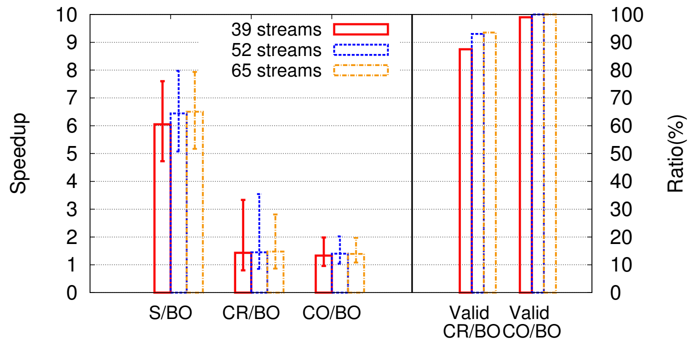

Fig. 12. Speedup over Set S, CR, and CO and the ratio of cases in which our solution (Set BO) shows better performance than CR and CO

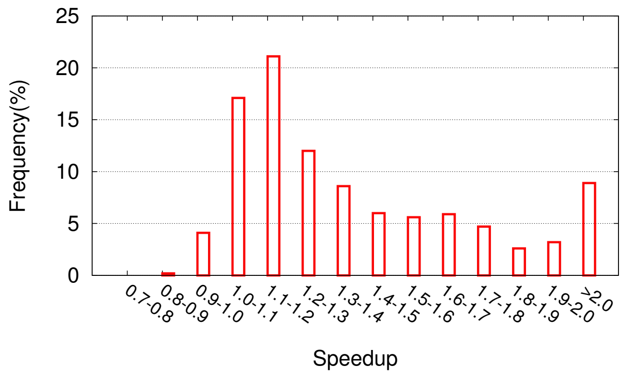

Fig. 11. Distribution of speedup over *random* in running workloads generated from synthetic kernels as a random set

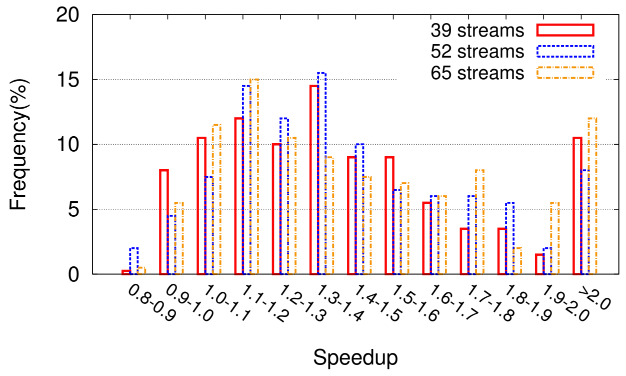

Fig. 13. Distribution of the speedup of our solution (Set BO) against Set CR

TABLE II
BATCH-OPTIMIZATION MODEL EXPERIMENT SETS

Experiment Set	Description
S	Sequential running with random batch assignment and random launching parameter
CR	Concurrent running with random batch assignment and random launching parameter
CO	Concurrent running with random batch assignment and optimized launching parameter
BO	Concurrent running with optimized batch assignment and optimized launching parameter

(the multipliers of our total kernel numbers), and each sets have run 200 times. To show our batch-optimization model can indeed decrease the number execution batches, we did synchronization after each batch finishes running.

We want to prove our model works when dealing with large number of kernels that cannot fit in an MP, discussions on experimental results will also be focused on total running time of the workload.

We can see from Fig. 12 that, by applying both our batch-optimization and kernel-optimization models, the average speedup against sequential execution ('S' hereafter) of 39, 52, 65 streams are 6.05x, 6.44x, and 6.50x, respectively; the average speedup against random execution without applying any of our models ('CR' hereafter) of 39, 52, 65 streams are 1.43x, 1.45x, and 1.47x, respectively; the average speedup against random execution with only applying kernel-optimization model ('CO' hereafter) of 39, 52, 65 streams are 1.33x, 1.40x, and 1.39x, respectively. The minimum speedup over S of 39, 52, 65 streams can achieve 4.72x, 5.07x, and 5.17x, respectively. Our solution beats S in all cases dramatically, while CR and CO some times find solutions with better performance. By plotting the distribution of the speedups in Fig. 13 and Fig. 14, we can see there are only a few cases falling into the section "0.8-0.9" and "0.9-1.0". Note that our solution almost beats CO in all cases, the percentage of winning cases against CO is 99%, 100%, and 100% for 39, 52, 65 streams, respectively. This is reasonable because in our kernel pool, the execution time of kernels will change with different launching parameters chosen. Plus, our models do not directly target running time. With the help of NVidia's Visual Profiler [17], we can guarantee our model achieve maximum concurrency and run for minimum number of batches.

VI. RELATED WORK

Push-based Database Systems: In traditional DBMSs, the cost of I/O is expensive since its pull-based architecture needs to load data repeatedly. Sharing data among concurrent

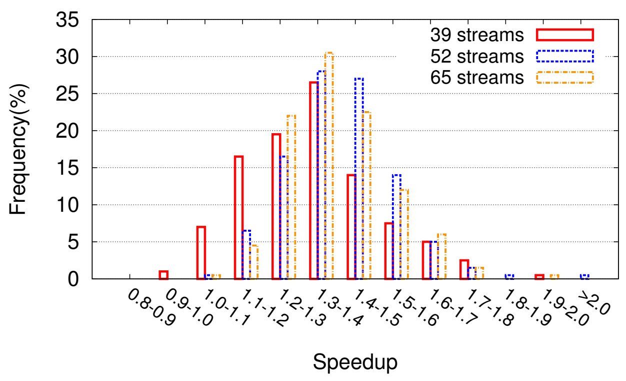

Fig. 14. Distribution of the speedup of our solution (Set BO) against Set CO

queries using a common I/O stream has become popular in database community. Harizopoulos *et al.* [19] enables dynamic operator sharing with an on-demand simultaneous pipelining I/O system (OSP). Ramen *et al.* [20] implemented a system called Blink that runs every query based on a table scan. Frey *et al.* [21] designed an efficient join algorithm called cyclo-join to process queries under a distributed environment through a ring-structured network. Unterbrunner *et al.* [22] proposed a distributed relational table design called Crescando that uses shared scan to process data stream on multi-core machines. Another sharing data approach was studied in [23], which is based on a data-sharing model in both record and column disk storage. More recently, Arumugam *et al.* [1] developed a truly push-based system called DataPath, in which queries are pushed to processors and all the operations 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 [24]. The advanced computing model such as CUDA [25] and OpenCL [26] accelerates its spread. According to [27], there are more than 60,000 technical papers published annually in the field of GPGPU. It is very clear that it becomes a popular computational platform in many application domains [28], [29]. The data management community has also done a lot of work on improving database performance using GPUs. GPU-based algorithms for computing major relational operators were developed by Govindaraju *et al.* [30], who reported dramatic performance improvement over a compiler-optimized SIMD implementation with up to 40 times speedup. Bakkum *et al.* [31] implemented a subset of command processors based on the open-source database named SQLite to achieve GPU acceleration. Sitaridi *et al.* [32] proposed a bank optimization solution for improving data access performance on GPU memory. It focused on resolving the conflict issues when using shared memory on GPUs in order to fully utilize the bandwidth of shared memory therefore enhancing performance. And there are some work about improving join algorithms on GPU. He *et al.* [33] implemented novel relation join GPU algorithms that obtained 2-7X better performance as compared to CPU-based algorithms. Kaldewey *et al.* [34] implemented some join processing algorithms on GPUs, and they got a 50% performance boost over CPU implementations of the same algorithms. Ran *et al.* [35] revisited He's algorithm after seven years under modern GPU architecture and achieved up to 20X speedup over the CPU-based join algorithms.

GPGPU and modeling: Besides database community, people who want to further explore potential of GPU begin to do modeling on it based on their different research domain, although there is only few of them. Xu *et al.* [36] proposed a GPU-accelerated simulation model for high-fidelity network systems. Baghsorkhi *et al.* [37] presented an analytical model to predict the performance of GPU applications with the help of an abstract interpretation method called work flow graph. Hong *et al.* [38] proposed an analytical model that estimates the execution time of programs running on GPUs and an improved version [39] later. The model estimates the number of parallel memory requests via analysis of program behavior and instructions. The same research group [40] also developed an empirical power model for GPUs. Kerr *et al.* [41] introduced a model based on Hong's analysis to predict relative performance of the same applications running on GPUs and CPUs. However, all above modeling efforts focused on single-kernel tasks on GPUs and single-kernel modeling efforts are not readily applicable to simultaneous multi-kernel scenarios. Moreover the modeling methods mentioned above either require extra effort to achieve accurate prediction or focus on a specific domain that is not applicable to our problem. This is also the motivation to conduct our research in this paper.

Other work also address multi-tasking in GPUs. With the help of Dynamic Parallelism in CUDA (a feature enables the developer to launch parallel work on a GPU), Krieder *et al.* [42] proposed an execution model and run-time system called GeMTC to decompose kernels into warp-level and integrated with Swift language. This proposal requires rewriting user kernels (i.e., decomposing into warp-level units) while kernels are treated as atomic units in our scheme based on CUDA streams. Wang *et al.* [43] adopted Kernel preemption (a technique that can swap the context of a kernel on one SM with the context of a new kernel) and developed a dynamic sharing mechanism named Simultaneous Multikernel (SMK) by improving resource utilization to boost performance. This technique is meant to be implemented in the GPGPU runtime system and only evaluated in a simulator while our strategy runs at the middleware level and is fully tested in a real system. It would be interesting to compare the performance if SMK is implemented in CUDA in the future.

VII. Conclusions

With very high parallel computing capacity, GPUs have become an integrated part of today's HPC systems and found applications in many scientific and computing domains. Management of large-scale scientific data has seen push-based query engine design as the main approach in dealing with the I/O bottleneck. By feeding shared data streams to multiple concurrent queries, such systems removed the bottleneck from I/O to computation, making GPUs a suitable platform for running the query engine. A key challenge in the implementation of such systems is to support concurrent tasks. Task parallelism feature (i.e., the CUDA stream) provided by CUDA can be leveraged to meet such challenges. The objective of this study is to allocate resources to concurrent CUDA kernels by configuring their runtime parameters for the purpose of maximizing system performance. We develop an integer programming kernel-optimization model to describe such a problem and a batch-optimization model to deal with a more general scenario that kernels are too many to execute in one batch. We design both algorithms of the two models for solving the optimization with proved correctness and high efficiency.

Acknowledgments

The reported work is supported by an award (IIS-1253980) from the National Science Foundation (NSF) of U.S.A. Equipments used in the experiments are partially supported by a grant (CNS-1513126) from NSF.

References

[1] 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 *ACM SIGMOD International Conference on Management of Data*, New York, NY, USA, 2010, pp. 519–530.

[2] Y.-C. Tu, A. Kumar, D. Yu, R. Rui, and R. Wheeler, "Data Management Systems on GPUs: Promises and Challenges," in *25th International Conference on Scientific and Statistical Database Management*, ser. SSDBM. New York, NY, USA: ACM, 2013, pp. 33:1–33:4.

[3] L. Golab and M. T. Zsu, *Data Stream Management*. Morgan & Claypool Publishers, 2010.

[4] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting-stock problem," *Operations research*, vol. 9, no. 6, pp. 849–859, 1961.

[5] P. C. Gilmore and R. E. Gomory, "A linear programming approach to the cutting stock problem-part ii," *Operations research*, vol. 11, no. 6, pp. 863–888, 1963.

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

[7] H. Li, D. Yu, A. Kumar, and Y. Tu, "Performance modeling in CUDA streams - A means for high-throughput data processing," in *Big Data (Big Data), 2014 IEEE International Conference on*, Oct 2014, pp. 301–310.

[8] TechPowerUp, "NVIDIA GeForce GTX TITAN X Pascal," https://www.techpowerup.com/gpudb/2863/titan-x-pascal.html.

[9] NVidia, "CUDA C Programming Guide," https://docs.nvidia.com/cuda/cuda-c-programming-guide/.

[10] tom's hardware, "Update: Nvidia Titan X Pascal 12GB Review," http://www.tomshardware.com/reviews/nvidia-titan-x-12gb,4700.html.

[11] S. Martello and P. Toth, *Knapsack problems: algorithms and computer implementations*. John Wiley & Sons, Inc., 1990.

[12] T. Crainic, G. Perboli, and R. Tadei, *Recent advances in multi-dimensional packing problems*. Available from: http://www. intechopen. com/books/new-technologies-trends-innovations-and-research/recent-advances-in-multi-dimensional-packing-problems, 2012.

[13] S. Martello, D. Pisinger, and D. Vigo, "The three-dimensional bin packing problem," *Operations Research*, vol. 48, no. 2, pp. 256–267, 2000.

[14] H. Kellerer, U. Pferschy, and D. Pisinger, *Introduction to NP-Completeness of knapsack problems*. Springer, 2004.

[15] J. A. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[16] A. Arbel, "Large-scale optimization methods applied to the cutting stock problem of irregular shapes," *THE INTERNATIONAL JOURNAL OF PRODUCTION RESEARCH*, vol. 31, no. 2, pp. 483–500, 1993.

[17] NVidia, "CUDA Toolkit," https://developer.nvidia.com/cuda-toolkit.

[18] A. Kumar, V. Grupcev, Y. Yuan, Y. Tu, and G. Shen, "Distance histogram computation based on spatiotemporal uniformity in scientific data," in *15th EDBT*, 2012, pp. 288–299.

[19] S. Harizopoulos, V. Shkapenyuk, and A. Ailamaki, "QPipe: A Simultaneously Pipelined Relational Query Engine," in *ACM International Conference on Management of Data*, ser. SIGMOD. New York, NY, USA: ACM, 2005, pp. 383–394.

[20] V. Raman, G. Swart, L. Qiao, F. Reiss, V. Dialani, D. Kossmann, I. Narang, and R. Sidle, "Constant-Time Query Processing," in *Data Engineering, IEEE 24th International Conference on*, April 2008, pp. 60–69.

[21] P. W. Frey, R. Goncalves, M. Kersten, and J. Teubner, "Spinning Relations: High-speed Networks for Distributed Join Processing," in *5th International Workshop on Data Management on New Hardware*, ser. DaMoN. New York, NY, USA: ACM, 2009, pp. 27–33.

[22] P. Unterbrunner, G. Giannikis, G. Alonso, D. Fauser, and D. Kossmann, "Predictable Performance for Unpredictable Workloads," *VLDB Endow.*, vol. 2, no. 1, pp. 706–717, Aug. 2009.

[23] M. Zukowski, S. Héman, N. Nes, and P. Boncz, "Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS," in *33rd International Conference on Very Large Data Bases*, ser. VLDB '07. VLDB Endowment, 2007, pp. 723–734.

[24] NVidia, "GPGPU.org," http://gpgpu.org.

[25] NVidia, "CUDA home," http://www.nvidia.com/object/cuda_home_new.html.

[26] T. K. Group, "OpenCL," https://www.khronos.org/opencl/.

[27] J.-H. Huang, "LEAPS IN VISUAL COMPUTING," http://on-demand.gputechconf.com/gtc/2015/presentation/S5715-Keynote-Jen-Hsun-Huang.pdf.

[28] H. Nguyen, *GPU Gems 3*, 1st ed. Addison-Wesley Professional, 2007.

[29] NVidia, "GPU Applications," http://www.nvidia.com/object/gpu-applications.html.

[30] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha, "Fast Computation of Database Operations Using Graphics Processors," in *ACM International Conference on Management of Data*, ser. SIGMOD. New York, NY, USA: ACM, 2004, pp. 215–226.

[31] P. Bakkum and K. Skadron, "Accelerating SQL Database Operations on a GPU with CUDA," in *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU, New York, NY, USA, 2010, pp. 94–103.

[32] E. A. Sitaridi and K. A. Ross, "Ameliorating Memory Contention of OLAP Operators on GPU Processors," in *8th International Workshop on Data Management on New Hardware*, ser. DaMoN, New York, NY, USA, 2012, pp. 39–47.

[33] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational Joins on Graphics Processors," in *ACM International Conference on Management of Data*, New York, NY, USA, 2008, pp. 511–524.

[34] T. Kaldewey, G. Lohman, R. Mueller, and P. Volk, "GPU Join Processing Revisited," in *8th International Workshop on Data Management on New Hardware*, ser. DaMoN, New York, NY, USA, 2012, pp. 55–62.

[35] R. Rui, H. Li, and Y.-C. Tu, "Join algorithms on GPUs: A revisit after seven years," in *Big Data (Big Data), 2015 IEEE International Conference on*, Oct 2015, pp. 2541–2550.

[36] Z. Xu and R. Bagrodia, "GPU-Accelerated Evaluation Platform for High Fidelity Network Modeling," in *21st International Workshop on Principles of Advanced and Distributed Simulation*, ser. PADS '07, Washington, DC, USA, 2007, pp. 131–140.

[37] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-m. W. Hwu, "An Adaptive Performance Modeling Tool for GPU Architectures," *SIGPLAN Not.*, vol. 45, no. 5, pp. 105–114, Jan. 2010.

[38] S. Hong and H. Kim, "An Analytical Model for a GPU Architecture with Memory-level and Thread-level Parallelism Awareness," *SIGARCH Comput. Archit. News*, vol. 37, no. 3, pp. 152–163, Jun. 2009.

[39] J. Sim, A. Dasgupta, H. Kim, and R. Vuduc, "A Performance Analysis Framework for Identifying Potential Benefits in GPGPU Applications," in *17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '12. New York, NY, USA: ACM, 2012, pp. 11–22.

[40] S. Hong and H. Kim, "An Integrated GPU Power and Performance Model," *SIGARCH Comput. Archit. News*, vol. 38, no. 3, pp. 280–289, Jun. 2010.

[41] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling GPU-CPU Workloads and Systems," in *3rd Workshop on General-Purpose Computation on Graphics Processing Units*, ser. GPGPU-3. New York, NY, USA: ACM, 2010, pp. 31–42.

[42] S. J. Krieder, J. M. Wozniak, T. Armstrong, M. Wilde, D. S. Katz, B. Grimmer, I. T. Foster, and I. Raicu, "Design and Evaluation of the Gemtc Framework for GPU-enabled Many-task Computing," in *23rd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '14. New York, NY, USA: ACM, 2014, pp. 153–164.

[43] Z. Wang, J. Yang, R. Melhem, B. Childers, Y. Zhang, and M. Guo, "Simultaneous Multikernel: Fine-grained Sharing of GPGPUs," *IEEE Computer Architecture Letters*, vol. PP, no. 99, pp. 1–1, 2015.