A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines

Lin Ma
Roger D. Chamberlain

Lin Ma and Roger D. Chamberlain, “A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines,” in *Proc. of 23rd IEEE International Conference on Application-specific Systems, Architectures and Processors* (ASAP), July 2012, pp. 24-31.

Dept. of Computer Science and Engineering
Washington University in St. Louis

2012 IEEE 23rd International Conference on Application-Specific Systems, Architectures and Processors

A Performance Model for Memory Bandwidth Constrained Applications on Graphics Engines

Lin Ma[1] and Roger D. Chamberlain[1,2]
[1]*Department of Computer Science and Engineering, Washington University in St. Louis*
[2]*BECS Technology, Inc., St. Louis, Missouri*
{*lin.ma, roger*}*@wustl.edu*

***Abstract*—Graphics engines are excellent execution platforms for high-throughput computations that exploit a large degree of available parallelism. The achieved performance is, however, highly dependent on the access patterns that the application imposes on the memory subsystem. Here, we propose an analytic model that helps improve the understanding of the performance of memory-limited kernels that employ random memory access schemes, especially as impacted by cache and various configuration parameters that can be used to tune kernel execution, such as the number of blocks and the number of threads per block. The analytic model is first explored through the use of a synthetic micro-benchmark, which is then followed by an empirical validation using a pair of production applications used in computational biology.**

***Keywords*-GPGPU, hashing, BLAST, DNA classification**

I. Introduction

With the introduction of general-purpose capabilities into graphics engines, many data parallel applications have been implemented that exploit the unique properties of these compute resources [10]. Both the CUDA and OpenCL development environments support application implementation using familiar languages, and the available debugging and performance monitoring tools provide substantial information about an application's correctness and execution speed.

An important issue that remains, however, is a comprehensive understanding of what algorithmic and/or architectural features have significant impact on the performance of particular applications. Getting a code running correctly is not very difficult, but getting it to perform well can still be quite a challenge. Ultimately, the achieved performance for an application is a complicated interaction between a variety of parameters, some set by the application developer, others imposed by the architecture of the particular graphics engine being used. The interactions between these parameters, as well as the impact of each on the application's performance, are often not well understood.

We can improve our understanding of the performance of applications on graphics engines through the use of analytic performance models, and several research groups have made good progress in developing such models. Ryoo et al. [11] summarize five categories of optimization mechanisms, using Pareto-optimal curves to prune the program optimization space by up to 98%. They do not, however, consider memory latency and multiple conflicting performance indicators. Hong et al. [7] propose a simple analytical model to capture an estimate of the cost of memory operations by counting the number of parallel memory requests in terms of memory-warp parallelism and computation-warp parallelism. But their assumption of no cache misses is not always realistic.

Liu et al. [8] describe a general performance model that predicts the performance of a biosequence database scanning application fairly precisely. Their model incorporates the relationship between problem size and performance, but does not look into microarchitecture-level parameters like how the number of threads and blocks influence the run time. Baghsorkhi et al. [2] measure performance factors in isolation and later combine them to model the overall performance via work flow graphs so that the interactive effects between different performance factors are modeled correctly. Their compiler-based approach still doesn't provide suggestions for run-time configuration of kernels. Govindaraju et al. [5] propose a cache model for efficiently implementing three memory intensive scientific applications with nested loops. It is helpful for applications with 2D-block representations while choosing an appropriate block size by estimating cache misses. He et al. [6] focus on the access patterns of gather and scatter operations, which can suffer from low memory bandwidth utilization, and design a probabilistic cache model to predict cache misses. Zhang et al. [13] present a quantitative performance model that characterizes an application's performance as being primarily bounded by one of three potential limits: instruction pipeline, shared memory accesses, and global memory accesses. In the present work, we focus on applications that are performance limited by the memory subsystem, in effect following the concept of [13], providing an analytic model for achievable processing throughput for a class of applications that typically have poor memory performance: hashing.

It is well understood that the memory bandwidth achievable by an application is not only impacted by which memory subsystem is being accessed (e.g., whether one is reading from off-chip global memory or on-chip shared memory), it is also impacted by the address access patterns themselves (i.e., whether or not global memory reads are

1063-6862/12 $26.00 © 2012 IEEE
DOI 10.1109/ASAP.2012.19

coalesced). Applications that employ hashing are particularly problematic from this perspective, since the size of the shared memory (which supports efficient random access) is limited and random accesses to global memory are difficult, if not impossible, to coalesce.

In our prior work [9], we presented an analytic model that described the performance of a Bloom filter [3] in terms of the parameters mentioned above. In that model, we constrained the number of requested blocks to evenly divide the work across the multiprocessors. In the present model, that restriction is lifted, characterizing application performance over a wider parameter space than the previous model. We also add support for modeling caches, incorporating the effect of cache misses into the performance expressions. In addition, this paper extends the model validation in two significant ways. First, we develop a micro-benchmark application that can be used to generate arbitrary memory access patterns. This allows us to separately investigate different performance factors and ultimately combine them into a coherent framework. Furthermore, this also provides us with the ability to perform a more comprehensive empirical validation of the analytic performance model. The validation effort uses an NVIDIA GTX480, based on the Fermi architecture. Second, we add a second real application from the literature [12] to further strengthen the empirical validation effort. Both our initial application (BLAST) and our second application (DNA classification) have Bloom filters as substantial components in their computation.

We conclude that it is reasonable to accurately characterize the memory subsystem performance for hashing-dominated applications, and that under the proper circumstances graphics engines can be effective computational resources for executing applications even in the face of poor memory access patterns.

II. Performance Model

In this paper, we will utilize both throughput (number of input data elements processed per unit time) and execution time (time to process a fixed size data set) as the performance metrics of interest. Our focus is on applications whose performance is dominated by memory access bandwidth, either to the shared memory or to the global memory. We concentrate on the throughput of the kernel executing on the graphics engine itself, leaving the performance assessment of data transfers to and from the graphics engine for other work. (In the two applications used for empirical evaluation in this paper, the performance is dominated by kernel execution time.)

We characterize application performance in terms of a variety of parameters, with application parameters summarized in Table I and architecture parameters summarized in Table II. The exposition below first reviews the model from [9] and then extends that model over a wider range of parameters and set of applications.

Table I
Application Parameters

Parameter	Description
D	Data set size
k	Number of hashing functions
m	Working set size
n	Size of decomposed sub-problem
R_T	Number of registers per thread
S_B	Shared memory used per block (in bytes)
B_r	Requested number of blocks (total)
T_r	Requested number of threads per block

Table II
Architecture Parameters

Parameter	Description
MP	Number of multiprocessors
S	Shared memory per multiprocessor (in bytes)
R	Number of registers per multiprocessor
W	Warp size (in number of threads)
C	Cache size
N_W	Min number of warps
$B_{\max}$	Max number of blocks (total)
T_{maxB}	Max number of threads per block
T_{maxMP}	Max number of threads per multiprocessor

A. Base Model

When a kernel is launched, the application specifies a request for a number of blocks, B_r, and a number of threads per block, T_r. Together, these two parameters determine the occupancy of the multiprocessors in the graphics engine. First, we want to know how many active blocks can be launched on each multiprocessor, denoted B_a. This can be expressed in terms of the register usage, shared memory usage, and fixed device capability.

$$B_a = \min\left(\left\lfloor\frac{S}{S_B}\right\rfloor, \left\lfloor\frac{R}{R_T \times T_r}\right\rfloor, \left\lfloor\frac{B_{\max}}{\mathit{MP}}\right\rfloor, \left\lfloor\frac{T_{\text{maxMP}}}{T_r}\right\rfloor\right) \tag{1}$$

Second, throughput is maximized when the requested number of blocks B_r is an integer multiple of the product of B_a and *MP*, thereby balancing the number of blocks allocated to each multiprocessor.

$$B_{\text{opt}} = \{B_r = i \times B_a \times \mathit{MP} \mid i \in \mathbb{N}\} \tag{2}$$

Here, B_{opt} is the set of possible block request counts that is required to yield peak (optimal) performance.

Third, in a similar manner, the number of threads per block T_r should be an integer multiple of the warp size W, forming a set of possible requested threads per block necessary for peak (optimal) performance, T_{opt}.

$$T_{\text{opt}} = \{T_r = j \times W \mid j \in \mathbb{N}\} \tag{3}$$

Finally, we define a set of Boolean indicators that encode whether or not the requested number of blocks is in the optimal set and is feasible given the practical constraints of the engine (denoted by A_B), and whether or not the requested number of threads per block meets similar conditions (denoted by A_T).

$$\begin{aligned} A_B &= B_r \in B_{\text{opt}} \wedge \\ &\quad B_r \leq B_{\max} \end{aligned} \tag{4}$$

$$\begin{aligned} A_T &= T_r \in T_{\text{opt}} \wedge \\ &\quad T_r \geq N_W \times W \wedge \\ &\quad T_r \leq \min\left(T_{\text{maxB}}, \frac{T_{\text{maxMP}}}{B_a}\right) \wedge \\ &\quad T_r \geq \frac{\frac{R}{R_t}}{\frac{B_r}{\mathit{MP}} + 1} \end{aligned} \tag{5}$$

Beyond membership in B_{opt}, the only constraint on the number of blocks is that it is within the count allowed by the system. For the number of threads per block, constraints include a minimum number (to mask latencies and provide sufficient parallelism) as well as an upper bound based on resource limits. The derivation of these constraints are provided in [9].

B. Model Extension

Additional variables used in the models are listed in Table III.

Table III
MODEL VARIABLES

Variable	Description
B_a	Active number of blocks per multiprocessor
A_B	Optimal block number indicator (Boolean)
A_T	Optimal thread number indicator (Boolean)
A_C	Working set fit in cache indicator (Boolean)
B_{opt}	Set of optimal numbers of blocks (total)
T_{opt}	Set of optimal numbers of threads per block
f_{app}	Application algorithmic complexity
f_{cache}	Cache factor
f_{sched}	Block scheduling factor
r_H	Cache hit rate
r_M	Cache miss rate
G	Ratio of cache to main memory throughput
$\overrightarrow{\text{algo}}$	Vector of algorithm parameters
$\overrightarrow{\text{inpt}}$	Vector of input size parameters
Time	Execution time (in seconds)
$\mathit{Time}_{\min}$	Shortest execution time (in seconds)
Tput	Throughput (in data elements per second)

What follows extends the model from [9]. Each application's peak performance is first described in terms of its algorithmic complexity. The algorithmic complexity is expressed via a function $f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}})$, defined in terms of an algorithm parameter vector $\overrightarrow{\text{algo}}$ and an input size vector $\overrightarrow{\text{inpt}}$. $\overrightarrow{\text{algo}}$ includes parameters from the algorithm design and implementation, e.g., number of hashing functions in a Bloom filter, size of a sub-block computation, etc. $\overrightarrow{\text{inpt}}$ takes the parameters relevant to the input problem size and working set size. The form of f_{app} is, of course, application specific. Specific examples will be provided in the sections below. f_{app} can be regarded as a general adapter of the model to different problem sizes, algorithms, and even to different implementations of each algorithm.

Similar to the indicators A_B and A_T (described above) that articulate whether or not a kernel's *configuration* (the combination of B_r and T_r) is optimal, an additional indicator A_C can be used to articulate whether or not the kernel's working set m fits in on-chip memory spaces, either shared memory or L1 cache. If the working set is allocated to and fits in shared memory, A_C is true because each reference to the working set is a hit. If shared memory isn't used, A_C is true when the working set fits in L1 cache. This can be expressed as:

$$A_C = \begin{cases} m < S & \text{if using shared memory} \\ m < C & \text{if using global memory.} \end{cases} \tag{6}$$

We are now in a position to articulate the peak performance of an application given a set of values for $\overrightarrow{\text{algo}}$ and $\overrightarrow{\text{inpt}}$.

$$\mathit{Time}_{\min} \propto f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) \quad \text{if } A_T \wedge A_B \wedge A_C \tag{7}$$

If and only if A_B, A_T and A_C are true does the kernel configuration provide peak performance.

Moving from peak performance, we next extend eqn. (7) to incorporate the effects of cache and of block scheduling.

$$\mathit{Time} \propto f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) \times f_{\text{cache}} \times f_{\text{sched}} \quad \text{if } A_T \tag{8}$$

Here, f_{cache} reflects the performance impact due to cache misses and f_{sched} reflects the performance impact due to the scheduling of blocks on multiprocessors. We consider each factor in turn.

Assuming that our memory access patterns are random (due to hashing), a simple model for cache hit rate is:

$$r_H = \min(1, \frac{C}{m}). \tag{9}$$

The above expression yields a hit rate of 1 when the working set size m is smaller than the cache size C. As the working set size exceeds the cache size, the hit rate is modeled as their ratio (reflecting the random access assumption). Given a hit rate r_H, the miss rate is straightforward to model:

$$r_M = 1 - r_H \tag{10}$$

We complete the cache performance model by expressing the cache factor as a linear combination of execution times that are blended by cache hit and miss rates:

$$f_{\text{cache}} = \begin{cases} 1 & \text{if } A_C \\ r_H + r_M \times G & \text{otherwise,} \end{cases} \tag{11}$$

where G reflects the multiplicative slowdown experienced with very low cache hit rates. In principle, one would like to express G in terms of the relative performance of the cache and the global memory. In practice, however, their relative performance difference is masked by the large number of threads supported. In this work, G is empirically determined.

In a similar manner to the cache effects, f_{sched} models the impact of block scheduling on the application performance, extending the overall model to predict execution time for numbers of requested blocks that are not only within the set B_{opt}, but also those outside of it.

$$f_{\text{sched}} = \frac{\lceil \frac{B_r}{B_a \times \mathit{MP}} \rceil \times B_a \times \mathit{MP}}{B_r} \tag{12}$$

The above expression reflects the block scheduling process on the multiprocessors, with the time determined by the multiprocessor with the largest number of blocks assigned to it (expressed via the ceiling function $\left\lceil \frac{B_r}{B_a \times \mathit{MP}} \right\rceil$). As we will see below, this yields a distinctive zigzag pattern in the throughput as the number of requested blocks is varied.

Given an expression for execution time, i.e. (8), we can describe processing throughput in terms of the data set size D and the execution time.

$$\mathit{Tput} = D/\mathit{Time} \tag{13}$$

III. Synthetic Micro-Benchmark

Given the stated goal of understanding the performance of applications with poor memory access patterns, we present a synthetic micro-benchmark that allows us to quantitatively explore the impact of random access patterns on application throughput. The computation is intentionally simple enough to ensure that memory accesses dominate its performance.

The choice of memory subsystem potentially has a significant performance impact, especially for randomly distributed accesses. In the present work, we focus on the shared memory and global memory subsystems, leaving the constant memory and texture memory for future work.

Figures 1 and 2 help us understand the operation of the micro-benchmark. Random numbers within a specified address range (i.e., working set size) are populated initially in the global memory. Then, as illustrated in Figure 1, each thread reads an individual data element, interprets that data element as a (random) pointer to a (synthetic) hash table in shared memory, and fetches the value from the table. Each block performs the same pattern of accesses, working with an assigned range of the pointers stored in global memory.

In a variation of the micro-benchmark, illustrated in Figure 2, the random pointers point to a (synthetic) hash table in global memory. In both cases, the size of the hash table (the working set size) is denoted by m.

We are now in a position to formulate a function f_{app} for the above micro-benchmark. Here, there are no parameters appropriate for $\overrightarrow{\text{algo}}$, so it is empty. The only input parameter is the data set size D, so $\overrightarrow{\text{inpt}} = (D)$ and thereby

$$f_{\text{app}}(D) = D \tag{14}$$

and

$$\mathit{Time} \propto D \times \frac{\lceil \frac{B_r}{B_a \times \mathit{MP}} \rceil \times B_a \times \mathit{MP}}{B_r} \quad \text{if } A_T \wedge A_C \ . \tag{15}$$

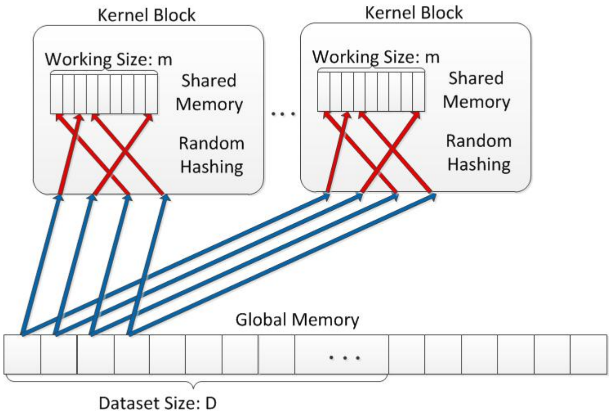

Figure 1. Shared memory random accesses.

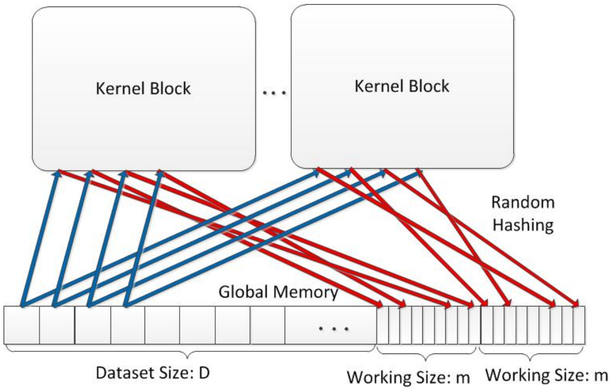

Figure 2. Global memory random accesses.

To investigate how the block scheduling factor f_{sched} influences the runtime, a small working set size is used to guarantee it fits on-chip, either in shared memory or L1 cache, so that A_C is true. The impact of the choice of memory subsystem accessed (whether shared or global) will be represented in the proportionality constant, which will be explored empirically.

The graphics engine used for this investigation is the NVIDIA GTX480, which has 15 streaming multiprocessors, $\mathit{MP} = 15$, each of which has 32 processor cores (480 cores total). The GTX480 has 1.5 GB off-chip global memory. Other architecture parameters are presented in Table IV.

We first empirically investigate how a range of choices for B_r influences the throughput for $T_r = \{960, 1024\} \subseteq T_{\text{opt}}$. The data set is 2^{25} random 4-byte words. In Figure 3, we use a working set size of 8 KB as the hashing range. In the experiments, B_r is varied from 1 to 90.

Our first observation from Figure 3 is that for both the shared memory subsystem and the global memory subsystem the empirically measured throughputs are closely aligned with the model predictions. Over the range of requested blocks explored in the graph, the values $B_r = \{15, 30, 45, 60, \ 75, 90\} \subseteq B_{\text{opt}}$ give peak throughput (min-

Table IV
ARCHITECTURE SPECIFICATION

Parameter	Specification
MP	15
S	16 KB or 48 KB (configurable)
C	48 KB or 16 KB (configurable)
R	32768
W	32
N_W	6 (NVIDIA recommended)
$B_{\max}$	120
T_{maxB}	1024
T_{maxMP}	1536

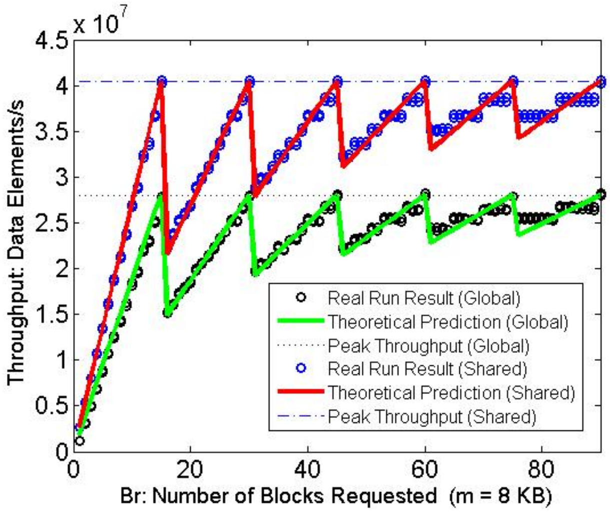

Figure 3. Throughput vs. B_r for random accesses to both shared and global memory subsystems ($D = 2^{25}$, $m = 8$ KB).

imum execution time) consistent with the prediction of eqn. (2), and when B_r is not in B_{opt} the zigzag pattern predicted by eqn. (8) is observed in the empirical data.

Our second observation is that there is a fairly significant difference in throughput between the shared memory and the global memory. This is consistent with our expectation that the shared memory is better suited to the random access patterns that drive the performance of the micro-benchmark.

To explore the impact of working set size on performance, we repeat the experiments above varying m from the initial value of 8 KB to 32 KB. The results of these experiments are shown in Figure 4 ($m = 32$ KB).

As the working set size gets larger, the throughput for the shared memory stays the same. This is consistent with the entire hash table fitting in shared memory for each working set size, so larger working sets do not provide any throughput disadvantages. In contrast to the shared memory result, the working set size has a dramatic impact on the performance of global memory accesses. While the zigzag pattern dependency on B_r is retained, the peak throughput is noticeably lower as the size of the working set increases. This is due to the smaller working set size being able to effectively exploit the on-chip caches that sit between the global memory and the multiprocessors on the GTX480.

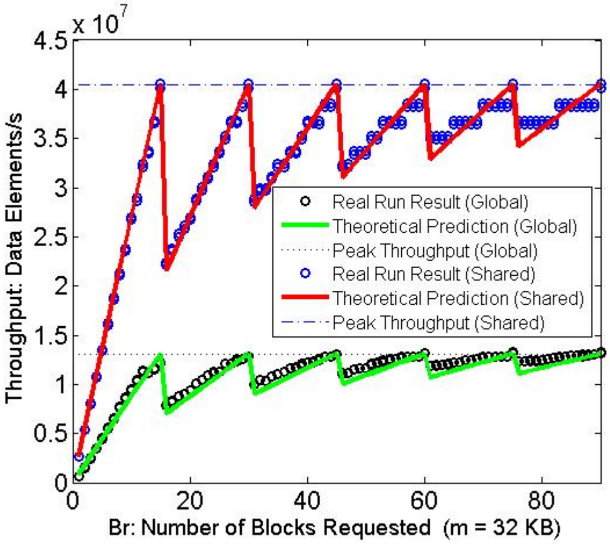

Figure 4. Throughput vs. B_r for random accesses to both shared and global memory subsystems ($D = 2^{25}$, $m = 32$ KB).

To quantify the effect on performance of cache, the cache model f_{cache} proposed in eqn. (11) is examined via the micro-benchmark, fixing f_{sched} to be optimal by ensuring that $B_r \in B_{\text{opt}}$. Cache hit rate r_H and cache miss rate r_M are explored by varying the working set size m. Different L1 cache sizes are also explored by setting it to 16 KB and 48 KB. Both the measured and model-predicted rates are shown in Figure 5. Dramatic increases in cache misses and decreases in cache hits are observed once a larger-than-cache working set size m is used. We see a nice correspondence between the modeled and measured results.

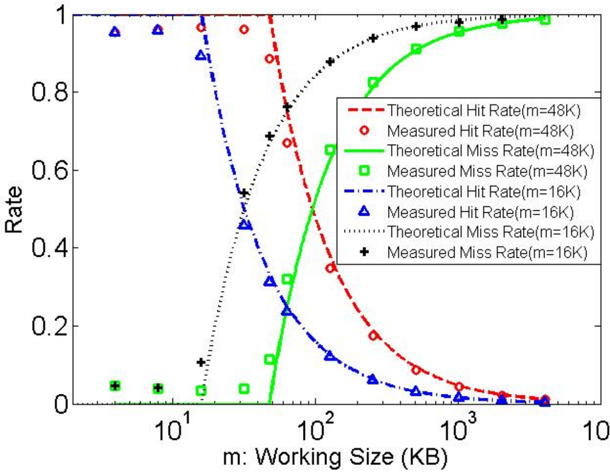

Figure 5. Cache hit and miss rates.

We next explore the cache model f_{cache}. Figure 6 compares measured and predicted execution times for the micro-benchmark as the working set size is varied over the same range as in Figure 5 (in this case, only for cache size 48 KB). For small working sets (that fit in cache) and for large working sets (that almost always miss cache) the execution time is flat. Eqn (11) does a reasonably good job of modeling the transition region between these two spaces.

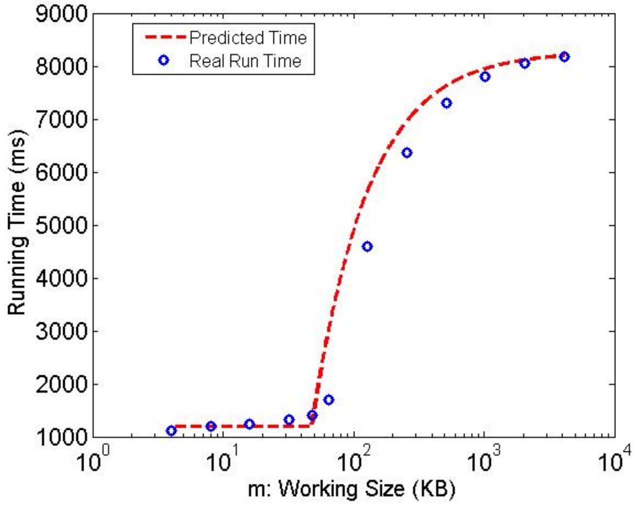

Figure 6. Impact of cache on execution time.

Given the close match between model predictions and empirical measurements for the micro-benchmark, we next consider a pair of real applications from the field of computational bioinformatics that are based either substantially or entirely on hashing.

IV. Bloom Filter in BLAST

BLAST is the most widely used tool for biosequence similarity search, which is a fundamental and crucial application for comparing and revealing the possibly biologically meaningful relationships between a given query sequence and an annotated database [1]. Given the rapid rate at which new genomic sequence data is being produced, BLAST searches have become progressively more and more expensive. In Buhler et al. [4], a Bloom filter [3], a probabilistic hashing algorithm and data structure for performing set membership tests with a manageable risk of producing false positives, is introduced at the front end of the traditional BLAST pipeline to discard a large fraction of the database prior to explicit table look-up and match verification.

A parallel Bloom filter algorithm for BLAST using a graphics engine is described in [9]. The algorithm deposits portions of the database in global memory, divides long queries into a set of sub-queries, and maps each sub-query to a specified Bloom-vector in shared memory for each kernel block. Multiple passes over the database are needed as the number of sub-queries are larger than the blocks the device can maximally support. Each thread reads a string of DNA characters (called a w-mer, since it is w characters in length) from the database in global memory, sequentially executes several hash functions in the kernel, and interrogates the values in shared memory pointed to by the hash results.

It is straightforward to develop an expression for f_{app} that reflects the Bloom filter implementation described above. The algorithmic parameters include the number of hashing functions k and sub-query size n, both are included in $\overrightarrow{\text{algo}}$. In terms of input problem size, we have the database sequence size D and the query sequence size Q. This results in

$$\overrightarrow{\text{algo}} = (k, n), \tag{16}$$

$$\overrightarrow{\text{inpt}} = (D, Q), \tag{17}$$

and

$$f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) = k \times \frac{Q}{n} \times D. \tag{18}$$

As the Bloom-vector is entirely held in shared memory, A_C is true. Substituting eqn. (18) into eqn. (8) yields an expression for the performance of BLAST's Bloom filter executing on a graphics engine.

$$\mathit{Time} \propto k \times \frac{Q}{n} \times D \times \frac{\lceil \frac{B_r}{B_a \times \boldsymbol{MP}} \rceil \times B_a \times \mathit{MP}}{B_r} \quad \text{if } A_T \tag{19}$$

The Bloom filter was executed while searching human chromosome 1 (250 MBases) against human chromosome 22 (50 MBases). The relations expressed in eqn. (18) were validated in [9]. Choosing $T_r = 1024 \in T_{\text{opt}}$, and drawing empirical measurements from [9], the predictive power of eqn. (19) is explored in Figure 7.

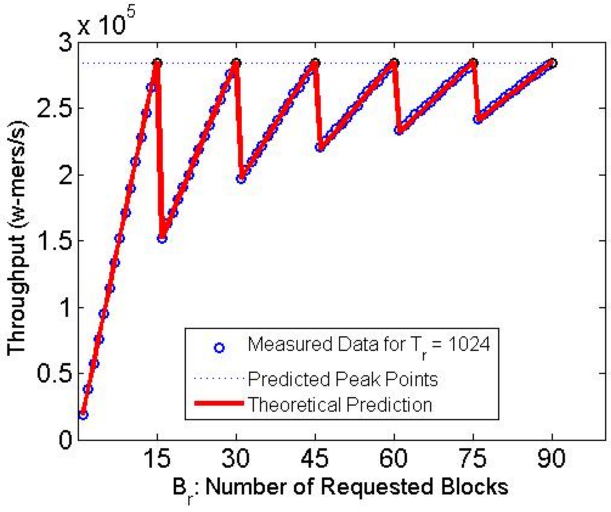

Figure 7. Throughput vs. B_r on GTX480 for Bloom filter of BLASTN.

As can be seen in the figure, there is an excellent correspondence between the empirical measurements and the model predictions. While the model of [9] was able to predict the repetitive nature of the performance peaks, it made no attempt to predict the throughput over the entire range of requested blocks.

V. DNA Classification

Another application that exploits hashing is DNA classification with Bloom filters. As DNA sequencing technologies provide ever more data to be analyzed, frequently biologists are interested in identifying only the novel sequence in a given data set. Stranneheim et al. [12] describe an algorithm, called FACS, which uses Bloom filters to classify sequences as belonging to one of many reference sequences V.S. being novel. Their perl-based implementation is evaluated using synthetic meta-genomic data sets and compared to conventional methods such as BLAT and SSAHA2. Stranneheim et al. observed a 21-fold speedup when FACS was executed on a 2.8 GHz Intel Xeon processor.

We ported FACS to the NVIDIA graphics engine to explore the potential for even greater performance gains. There are numerous opportunities for parallel execution, making it potentially well suited for the graphics engine; however, its reliance on hashing as a basic operation poses some question as to its ultimate suitability. Let us first assess the opportunities for parallelism. Within each short query sequence (typically less than 390 characters long), hashing the w-mers (substrings of length w) are independent. In our implementation, each w-mer within a query is assigned to a thread, which is responsible for computing all of the k hashes to implement the Bloom filter.

Second, the queries themselves (totaling approximately 10^5 sequences) are also independent and can be analyzed in parallel. We assign queries to thread blocks. Further, multiple kernel invocations are used to process groups of queries, and CUDA streams are used to provide overlapping kernel execution and memory copy to/from the graphics engine. Figure 8 illustrates the organization of the FACS implementation on the graphics engine.

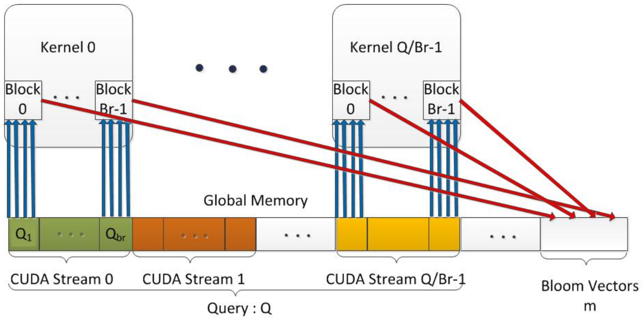

Figure 8. Implementation of FACS DNA classification application.

In the empirical investigation that follows, Bloom filters were created based on reference sequences from [12] with a measured false positive rate of 0.014% (lower than that in [12]), with one Bloom-vector per reference sequence. Due to the much larger size of the Bloom-vectors than on-chip shared memory, they were allocated to global memory.

The performance of this implementation can also be predicted by our model. The number of hashing functions k and the number of CUDA streams (which is inversely proportional to B_r) are the key parameters to be included in $\overrightarrow{\text{algo}}$. In terms of problem size, D is the number of sequences to be classified. This results in

$$\overrightarrow{\text{algo}} = (k, B_r) \tag{20}$$

and

$$\overrightarrow{\text{inpt}} = (D). \tag{21}$$

Since B_r might be larger than the number of active blocks that the multiprocessors can support, i.e., $B_a \times MP$, multiple passes are needed. So we have

$$f_{\text{app}}(\overrightarrow{\text{algo}}, \overrightarrow{\text{inpt}}) = k \times \frac{D}{B_r} \times \frac{B_r}{B_a \times MP} = k \times \frac{D}{B_a \times MP}. \tag{22}$$

A_C is false for this application due to the much larger Bloom-vector as working set than caches. So f_{cache} is greater than one, and substituting eqns. (22) and (11) into eqn. (8), we obtain the runtime expression for DNA classification as

$$\begin{aligned} \mathit{Time} \quad \propto \quad & k \times \frac{D}{B_a \times MP} \times \qquad (23) \\ & \left(\min\left(1, \frac{C}{m}\right) + \left(1 - \min\left(1, \frac{C}{m}\right)\right) \times G\right) \times \\ & \frac{\lceil \frac{B_r}{B_a \times MP} \rceil \times B_a \times MP}{B_r} \quad \text{if } A_T \ . \end{aligned}$$

The model is experimentally assessed on the graphics engine with the same synthetic meta-genome data set as [12], including 10^5 short query sequences. In our implementation, sequences are evenly distributed across a set of streaming kernels, and B_r blocks are requested on each kernel. This division restricts the number of sequences to be processed per kernel, therefore value options for B_r are limited, thereby preventing B_r from being a multiple of $B_a \times MP$.

The predictions of the model are presented in Figure 9. Because of the implementation restrictions described above, there are fewer empirical values for B_r relative to the previous application. Nonetheless, we see an excellent alignment between the model's predictions and the measured throughput achieved by our implementation.

For this problem, the working set size m (512 KB) is much larger than the cache, independent of the choice of cache size (16 KB or 48 KB). As a result, we do not predict a change in performance when the cache size is changed, and this was confirmed experimentally as well.

Quantifying the throughput in terms of hashes processed per second, our implementation executing on the GTX480 is 20 times faster than the perl version implementation executing on an Intel Core 2 Duo CPU running at 2 GHz.

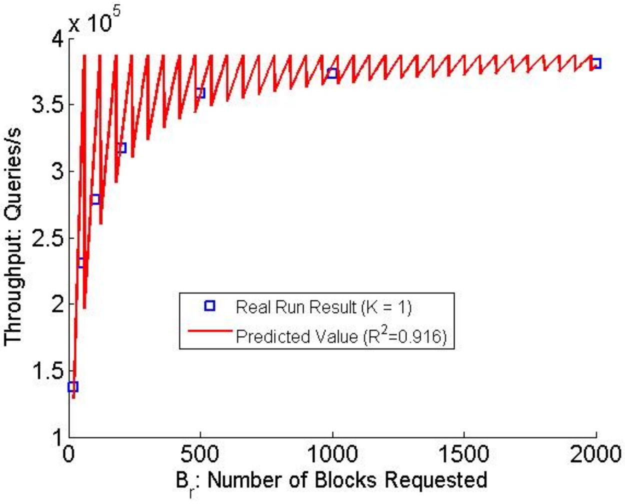

Figure 9. FACS throughput for different numbers of requested blocks.

VI. Conclusions

The paper has presented an analytical performance model that is well suited for memory-limited kernels. The model is validated using a synthetic micro-benchmark which allows us to quantitatively explore the impact of random memory access patterns in terms of cache effects and varied kernel configuration options. Two real applications from computational bioinformatics are also used to further examine the model's effectiveness. Given the excellent match between model predictions and empirical measurements, we conclude that the model can be effectively used not only to understand the performance of existing applications, but it can also be used to help configure the tuning parameters that must be set when executing any graphics engine kernel. The model confirms that, in general, shared memory is better suited to handling random memory access patterns. However, given sufficient parallelism and a small enough working set, even random access to global memory can be effective.

As future work, we intend to investigate the impact of various access patterns (e.g., sequential, strided, random) on other memory subsystems, such as constant memory and texture memory. A comprehensive understanding of the diversified memory spaces within graphics engines, each with distinct features, will be helpful for effectively implementing complex kernels.

Acknowledgments

This work was supported by NIH award R42 HG003225, NSF grants CNS-0751212, CNS-0905368, and CNS-0931693, and Exegy, Inc. R.D. Chamberlain is a principal in BECS Technology, Inc.

References

[1] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman, "Basic local alignment search tool," *Journal of Molecular Biology*, vol. 215, no. 3, pp. 403–410, Oct. 1990.

[2] S. S. Baghsorkhi, M. Delahaye, S. J. Patel, W. D. Gropp, and W.-M. Hwu, "An adaptive performance modeling tool for GPU architectures," in *Proc. of Symp. on Principles and Practice of Parallel Programming*, 2010, pp. 105–114.

[3] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[4] J. Buhler, J. Lancaster, A. Jacob, and R. Chamberlain, "Mercury BLASTN: Faster DNA sequence comparison using a streaming hardware architecture," in *Proc. of Reconfigurable Systems Summer Institute*, June 2007.

[5] N. K. Govindaraju, S. Larsen, J. Gray, and D. Manocha, "A memory model for scientific algorithms on graphics processors," in *Proc. of ACM/IEEE Supercomputing Conf.*, 2006.

[6] B. He, N. K. Govindaraju, Q. Luo, and B. Smith, "Efficient gather and scatter operations on graphics processors," in *Proc. of ACM/IEEE Supercomputing Conf.*, 2007.

[7] S. Hong and H. Kim, "An analytical model for a GPU architecture with memory-level and thread-level parallelism awareness," in *Proc. of 36th Int'l Symp. on Computer Architecture*, 2009, pp. 152–163.

[8] W. Liu, W. Muller-Wittig, and B. Schmidt, "Performance predictions for general-purpose computation on GPUs," in *Proc. of Int'l Conf. on Parallel Processing*, 2007.

[9] L. Ma, R. D. Chamberlain, J. D. Buhler, and M. A. Franklin, "Bloom filter performance on graphics engines," in *Proc. of Int'l Conf. on Parallel Processing*, 2011, pp. 522–531.

[10] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. J. Purcell, "A survey of general-purpose computation on graphics hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, 2007.

[11] S. Ryoo, C. I. Rodrigues, S. S. Stone, S. S. Baghsorkhi, S.-Z. Ueng, J. A. Stratton, and W.-M. Hwu, "Program optimization space pruning for a multithreaded GPU," in *Proc. of 6th IEEE/ACM Int'l Symp. on Code Generation and Optimization*, 2008, pp. 195–204.

[12] H. Stranneheim, M. Käller, T. Allander, B. Andersson, L. Arvestad, and J. Lundeberg, "Classification of DNA sequences using Bloom filters," *Bioinformatics*, vol. 26, pp. 1595–1600, July 2010.

[13] Y. Zhang and J. Owens, "A quantitative performance analysis model for GPU architectures," in *Proc. of Int'l Symp. on High Performance Computer Architecture*, Feb. 2011, pp. 382–393.