2011 International Conference on Parallel Processing

Bloom Filter Performance on Graphics Engines

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

***Abstract*—Bloom filters are a probabilistic technique for large-scale set membership tests. They exhibit no false negative test results but are susceptible to false positive results. They are well-suited to both large sets and large numbers of membership tests. We implement the Bloom filters present in an accelerated version of BLAST, a genome biosequence alignment application, on NVIDIA GPUs and develop an analytic performance model that helps potential users of Bloom filters to quantify the inherent tradeoffs between throughput and false positive rates.**

***Keywords*-NVIDIA GPU, Bloom Filter, BLAST**

I. Introduction

A Bloom filter [2] is a probabilistic algorithm and data structure for performing set membership tests. With a manageable risk of producing false positives and no chance of false negatives, Bloom filters are widely used for large-scale data sets, including dictionaries [14], [16], databases [3], [9], [23], networking applications [4], data speculation and prefetching [18], and filtering of XML packets [8]. They are an effective, space-efficient approach for testing membership.

In the field of computational bioinformatics, sequence similarity search is a fundamental and crucial application for comparing and revealing the possibly biologically meaningful relationships between a given query sequence and a database of known sequences. Sequences identified as similar are hypothesized to be derived from the same ancestral sequence and therefore to share the same evolutionary origin and function. A typical search is to systematically compare a database of known sequences to an unknown query sequence, identifying those members of the database that exhibit a high degree of similarity to the query. Given the rapid rate at which new genomic sequence data is being produced, this search task has become progressively more expensive, motivating the use of heuristics that reduce the number of database sequences which must be compared in detail to the query. These heuristics in turn have proved amenable to acceleration on a variety of architectures.

The most widely used tool for similarity search is the Basic Local Alignment Search Tool (BLAST) [1], a program distributed by the National Center for Biological Information (NCBI). This application has seen a number of accelerated implementations, including TreeBLAST [10], RC-BLAST [7], and Mercury BLAST [5], among others. All of the above implementations use FPGAs as co-processors.

Figure 1 shows the computational pipeline for Mercury BLAST. The original stage 1 of NCBI BLAST is here further decomposed into two pipeline stages, stage 1a and stage 1b. Stage 1, as a whole, is responsible for identifying exact matches of substrings of length w (called w-mers) between the pre-loaded query and the streaming database. Mercury BLAST inserts a Bloom filter front end (stage 1a) that discards a large fraction of the database prior to the explicit table lookup used for match verification in stage 1b. The remainder of the Mercury BLAST pipeline is described in [5] and the implementation of the Bloom filters in stage 1a on an FPGA is described in [11].

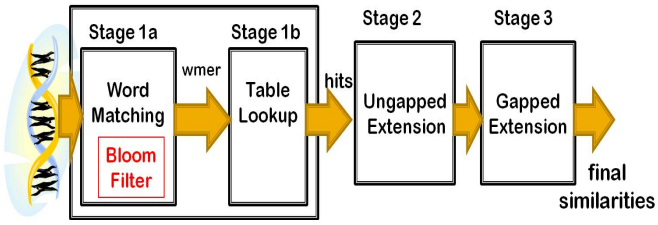

Figure 1. Mercury BLAST pipeline.

This work investigates the viability of using GPUs to implement the Bloom filters used by stage 1a of Mercury BLAST. We present an implementation of Bloom filters on a GPU, specifically targeting the BLAST application. To understand the performance achievable by such an implementation, we present an analytic performance model that quantifies a multidimensional performance vector, including sensitivity as well as throughput. We separate the parameters that inform this performance model into those that are application-specific (set by the implementer/user of the application) vs. architecture-specific (set by the manufacturer of the GPU, i.e. NVIDIA). This model is likely of general use for predicting GPU Bloom filter performance in applications other than BLAST (e.g., [22]).

Recently, several groups have implemented GPU-based Bloom filters for applications including IP routing [15] and error correction [13] [21]. These implementations have distinct problem sizes that to a great extent determine

0190-3918/11 $26.00 © 2011 IEEE
DOI 10.1109/ICPP.2011.27

how to map the Bloom vector and computation onto different memory spaces, how to appropriately choose runtime configurations, and how to program the kernel in an application-specific efficient way. Some use texture memory and/or constant memory for retaining the Bloom vector. Our implementation relies on the on-chip shared memory, which is significantly larger than the texture or constant caches. Costa et al. designed and open-sourced a BloomGPU library with flexiable APIs and automated tuning to offload Bloom filter support to the GPU for targeted applications' batch-oriented usage pattern [6]. However, the paper only exploited global memory rather than the much faster shared memory which might potentially achieve a better performance. Liu et al. [13] present DecGPU as the first parallel and distributed error correction algorithm for high-throughput short reads using a hybrid combination of CUDA and MPI. Extensive comparison has been conducted between DecGPU and other exsiting implementations. However, no detail about GPU performance optimization is explicitly stated.

In addition to reporting on our Bloom filter implementation, we provide an analytic performance model that quantifies the inherent tradeoffs that exist between two performance indicators, sensitivity and throughput. There exist several GPU performance models in the literature. Liu et al. [12] present a general GPU performance model classifying factors into three categories to establish the relation between problem sizes and performance factors and apply the model to a biosequence database scanning application. However, that model does not look into the fine-grained micro-architecture, including how choices of numbers of threads and blocks influence the timing, and how to map the computation onto a GPU architecture with given specifications. Ryoo et al. [20] focus more on micro-architecture and the kernel. They summarize five categories of optimization mechanisms to prune the GPU program optimization space by up to 98%. They do not, however, consider multiple, conflicting performance indicators.

II. Designing a Bloom Filter for Mercury BLAST Using a GPU

A. Parallel Bloom Filter Algorithm

Bloom filters test set membership by performing multiple hashes on a candidate element and checking a bit-vector, called the *Bloom vector*, to see if the addresses resulting from these hashes are all set to "true." Figure 2 illustrates this idea as applied to BLAST-style string matching. Fixed-length candidate substrings of length w, or *w-mers*, from the database are fed into k independent hash functions, and the resulting addresses are checked against one or more Bloom vectors loaded with portions of the query sequence.

Algorithm 1 gives pseudocode describing the Bloom filter string-matching computation. With multiple candidate elements, multiple sets, and multiple hash functions, this algorithm provides many opportunities to exploit parallelism.

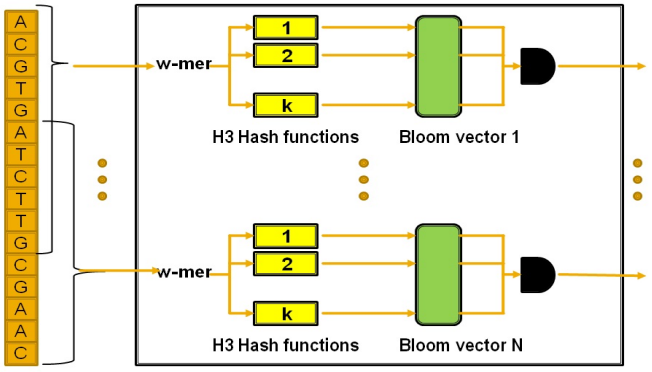

Figure 2. Parallel Bloom filters for detecting string matches of fixed length w between a query and a database.

In our design, long queries are split into multiple *sub-queries* of a given length n. Each sub-query is assigned an individual Bloom vector of size m bits, and each w-mer in the sub-query is considered to be an element of the set for that vector. Each w-mer in the database is simultaneously checked for set membership in each sub-query. This decomposition of a large set into multiple smaller sets (i.e., dividing a given query into a collection of sub-queries) is common practice with Bloom filters, as the false positive rate is a function of the number of elements in the set. A larger number of sub-queries, each with fewer elements, can lower the overall false positive rate.

Mercury BLAST, being FPGA-based, uses hash functions h_q from the H3 family [19], denoted as the set $\{h_q | q \in Q\}$, where Q represents the set of all possible $i \times j$ Boolean matrices. $q(k)$ is the bit string of the kth row of a given matrix $q \in Q$. Correspondingly, $x(k)$ is the kth bit of x, the element that needs to be hashed. The hashing function $h_q(x)$ is therefore defined as

$$h_q(x) = x(1) \cdot q(1) \oplus x(2) \cdot q(2) \oplus \cdots \oplus x(i) \cdot q(i) \quad (1)$$

where $\cdot$ denotes the bit by bit AND operation and $\oplus$ the exclusive OR operation. These bit-level linear transformations are well-suited to hardware implementation. In this paper, we do not investigate the use of alternative hash functions, leaving this for future work.

B. GPU Implementation

The GPU used for this study is the NVIDIA GTX 480, based on the Fermi architecture. It has 15 *streaming multiprocessors*, each of which has 32 *streaming processors* or processor cores (480 cores total) running at 1.4 GHz. The GTX 480 has about 1.5 GB of off-chip global memory, while each streaming multiprocessor has 48 KB of on-chip shared memory.

Kernel computations on the GPU are organized around thread blocks, which are independent from one another and are distributed across the multiprocessors for execution.

Algorithm 1 Parallel Bloom Filters for BLAST

```
1:  Input: query sequence
2:  Input: database sequence
3:  Output: stream of database w-mers
4:                                   ▷ Initialize Bloom vectors
5:  for all sub-queries do
6:      initialize all-zero bitVector of size m bits
7:      for each w-mer in sub-query (denoted x) do
8:          for each hash function h do
9:              bitVector[hash_h(x)] = 1
10:         end for
11:     end for
12: end for
13:                                  ▷ Perform membership tests
14: for all sub-queries do
15:     for each w-mer in database (denoted y) do
16:         for each hash function h do
17:             if bitVector[hash_h(y)] = 0 then
18:                 discard this w-mer
19:                 break
20:             end if
21:         end for
22:         if bitVector[hash_h(y)] = 1 for all h then
23:             output w-mer
24:         end if
25:     end for
26: end for
```

Each block consists of a number of threads, which are distributed across the processor cores within a multiprocessor. Threads are scheduled in groups of 32, called *warps*. The shared memory is shared across threads but is partitioned across blocks. The registers within each core are not shared but rather are partitioned across threads.

1) Problem Decomposition: We implement only the membership tests from Algorithm 1 (lines 13 to 26) on the GPU. As sub-queries are independent, they are distributed across the blocks, each with their associated Bloom vector. The vectors reside in shared memory to ensure fast access. This decomposition of the problem, assuming B blocks and k hash functions, is illustrated in Figure 3.

The database resides in the GPU's global memory. If, as is typically the case, the entire database is too large to fit in global memory, an additional outer loop (not shown) streams chunks of the database into global memory one at a time and serially executes *for* loop on line 14 for each chunk.

Individual threads partition the w-mers in the database. Each thread executes the inner loop of lines 15 to 25. A thread fetches a w-mer from the database (located in global memory), computes the k hash functions for its assigned Bloom vector, checks the resulting k addresses in the vector, and finally returns any hits to global memory.

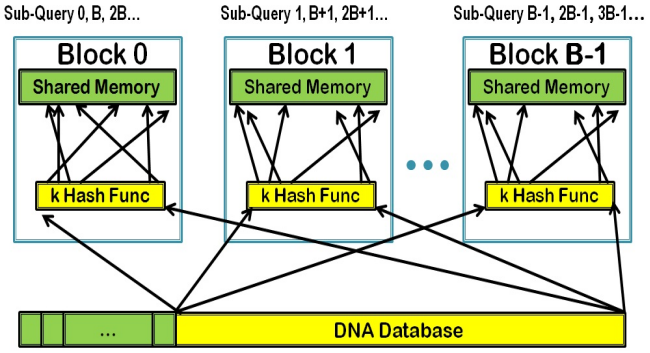

Figure 3. Implementation of parallel Bloom filter algorithm on GPU.

2) Operating Procedure: The overall sequence of operations consists of a number of steps, which are articulated below.

1) On the CPU:
   a) The database is encoded, using two bits per DNA base (character). Encoding need only be done once, after which the encoded database is stored in the file system.
   b) The H3 hash functions for each Bloom vector are generated, and the Bloom vectors are initialized (lines 5 to 12).
   c) The hash functions, Bloom vectors, and database are loaded into GPU memory.
2) On the GPU:
   a) The membership tests are performed.
   b) The database w-mers that hit in the Bloom filters are returned to CPU memory.

3) Optimization: We next describe various implementation decisions made in an attempt to optimize the performance of the kernel.

- *On-chip vs. off-chip memory allocation*. The off-chip global memory performs well when memory accesses within a warp can be coalesced. We accomplish this by having the 32 threads in a warp read 32 consecutive w-mers from the database. The on-chip shared memory more readily supports the random access pattern required by the Bloom vectors.
- *Thread-level parallelism (TLP) vs. Instruction-level parallelism (ILP).* There is a tradeoff between the work assigned to a single thread and the total number of threads. This corresponds to balancing TLP (more threads, less to do in each thread) and ILP (fewer threads, more to do in each thread). In the Bloom filter implementation, more TLP would be exploited if we had assigned each of the k hash functions to a distinct thread. However, the additional synchronization overhead implied by a one-thread-per-hash design makes it more efficient to compute all hash functions for one

w-mer in a single thread.

- *Unrolling loops.* As suggested in [20], we unrolled the loop (lines 16 to 24) that iterates through the k hash functions.

III. Performance Model

It is common for the performance of an application to be multidimensional. In the case of a Bloom filter, we have two primary performance indicators of interest: throughput and sensitivity. Throughput can be quantified for our BLAST application as the number of database w-mers processed per unit time, while sensitivity is quantified as the false positive rate realized during set membership tests. These indicators are influenced by a number of parameters, both application-specific and architecture-specific. Those parameters that are under control of the application developer are shown in Table I, while those that are fixed by the particular choice of GPU are shown in Table II. Additional variables used in the model are shown in Table III.

Table I
Application-Specific Parameters

Parameter	Description
DB	Database size (in w-mers)
Q	Query size (in w-mers)
n	Sub-query size (in w-mers)
m	Bloom vector size (in bits)
k	Number of hash functions
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-Specific Parameters

Parameter	Description
MP	Number of multiprocessors
S	Shared memory per multiprocessor (in bytes)
R	Number of 32-bit registers per multiprocessor
W	Warp size (in number of threads)
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

An analytic model that predicts throughput and sensitivity can be used for a number of purposes. Such a model can be used to tune the application so as to achieve good performance in a predictive way without empirically traversing a huge search space. In addition, multiobjective optimization techniques can be employed to explore the tradeoffs inherent in the performance vector. For example, knowledge of the achievable downstream throughput in the BLAST pipeline could be exploited to establish a throughput constraint and determine the best achievable sensitivity given this constraint. Alternatively, one could optimize a weighted combination of throughput and sensitivity. The analytic model we present will support any of the above options.

Table III
Model Variables

Variable	Description
SQ	number of sub-queries
FP	False positive count
TP	True positive count
FPR	False positive rate
B_a	Active number of blocks per multiprocessor
A	Peak performance indicator
B_{opt}	Set of optimal numbers of blocks (total)
T_{opt}	Set of optimal numbers of threads per block
Time	Execution time (in seconds)
Tput	Throughput (in w-mers per second)

A. Sensitivity

The sensitivity of a Bloom filter is quantified by the false positive rate, *FPR*, or fraction of set membership tests that return true when the element tested is not a member of the set. Lower false positive rates reflect better Bloom filter sensitivity.

Assuming element independence and good uniformity in the hash functions, the false positive rate for a Bloom filter is well modeled [4]. *FPR* is a function of the Bloom vector size m, the number k of hash functions, and the number n of elements hashed into the vector:

$$FPR = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k}. \tag{2}$$

According to the analytic model, *FPR* increases with n and decreases when m is increased. Increases in k can cause *FPR* to move in either direction, depending upon the value of the other two parameters.

In our usage of the Bloom filter within BLAST, both k and m are design parameters under direct control of the developer, while n is indirectly set by how the user decomposes the complete query into sub-queries.

To investigate whether biosequence data sets are sufficiently well-behaved so as to fit the theoretical expression for *FPR* above, we tested our implementation using real DNA sequences. Human chromosome 1 (250 Mbases) was used as our query sequence, while human chromosome 22 (50 Mbases) was used as the database. During execution of our GPU kernel, we counted false positives *FP*, false negatives *FN*, and true positives *TP*. The empirical *FPR* for a database of size *DB* is given by

$$FPR = \frac{FP}{DB - TP}. \tag{3}$$

We confirmed empirically that $FN = 0$, as required by any correct implementation. Figures 4 and 5 compare the theoretical and empirical *FPR* for a range of values of k, m, and n. In both figures, lines indicate the theoretical *FPR*, while mean measured *FPR* values over all sub-queries are shown as points with associated 95% confidence intervals. Figure 4 varies n for several values of m with a fixed $k = 6$, while Figure 5 varies n for several values of k for a fixed $m = 256$ Kbits. As expected, *FPR* grows with increasing n in all cases. For a given n, larger m leads to a smaller *FPR* and larger k can influence *FPR* either direction (depending on the value of n).

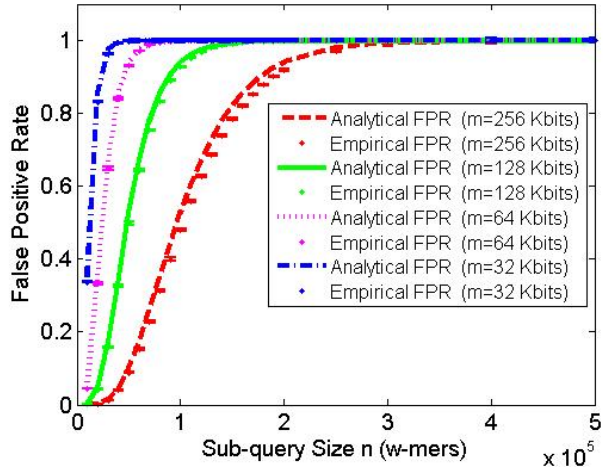

Figure 4. Theoretical and empirical *FPR* ($k = 6$).

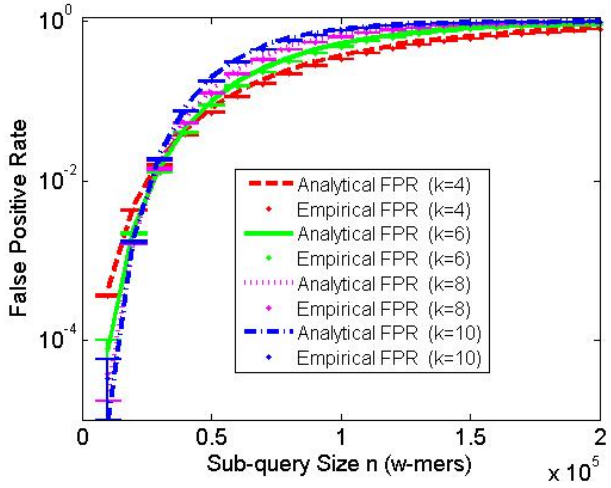

Figure 5. Theoretical and empirical *FPR* ($m = 256$ Kbits).

While the theoretical and empirical results are highly similar, the theoretical quantities frequently lie outside the confidence intervals of the empirical measurements. This is due to the fact that DNA bases are, generally, not independent of one another, but are in fact correlated. We explore the magnitude of this discrepancy by plotting a histogram of the relative error in Figure 6. While there are individual measurements with relative error greater than 10%, they are few, and the bulk of the errors are near zero.

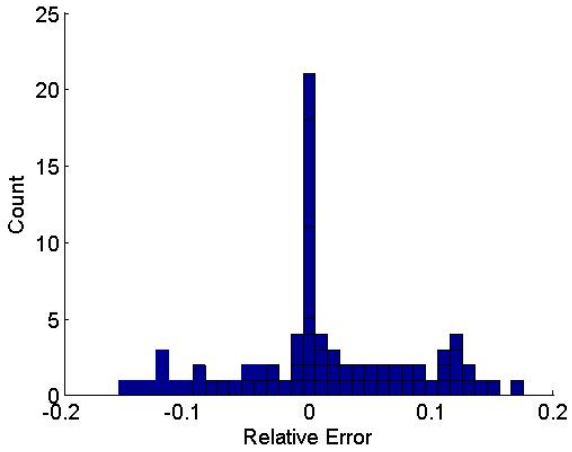

Figure 6. Histogram of relative error between theoretical predictions and empirical measurements for *FPR*.

B. Throughput

We focus initially on the throughput of the kernel, deferring the investigation of data movement to and from the GPU until later. As there is significant interaction among the various application-specific and architecture-specific parameters that influence throughput, we will construct the model piece by piece until all relevant parameters have been included. The first two parameters to be investigated, the number of hash functions k and the sub-query size n, influence both sensitivity and throughput and so partly control the tradeoff between them.

1) Number of Hash Functions: Kernel execution time should be linearly proportional to the number k of hash functions used. This is because each thread computes the k hash functions sequentially for each database w-mer assigned to it. Figure 7 shows empirical execution times vs. k for a variety of sub-query sizes n, along with a linear trend line fit to the data for each n. The tight fit of the trend lines to the data confirms our expectation that *Time* $\propto k$. Increasing k to improve sensitivity therefore negatively impacts throughput.

2) Sub-query Size: With a query of size Q that is divided into sub-queries of size n, the number of sub-queries that must be processed is $SQ = Q/n$. Because each sub-query is assigned to a block, and there are more sub-queries than blocks, multiple passes over the currently resident portion of the database are needed to process all sub-queries. Execution time is therefore proportional to total query size and inversely proportional to sub-query size.

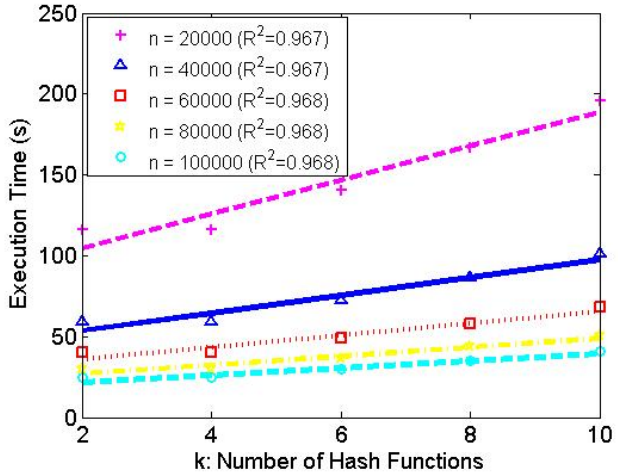

Figure 7. Execution time for different number of hash functions.

Figure 8 tests the above relation in four sets of experiments with different numbers k of hash functions. As above, the points represent empirical execution, while lines are fitted to the empirical data. The high goodness of fit confirms that *Time* $\propto$ *SQ*. Increasing n therefore improves throughput but negatively impacts sensitivity.

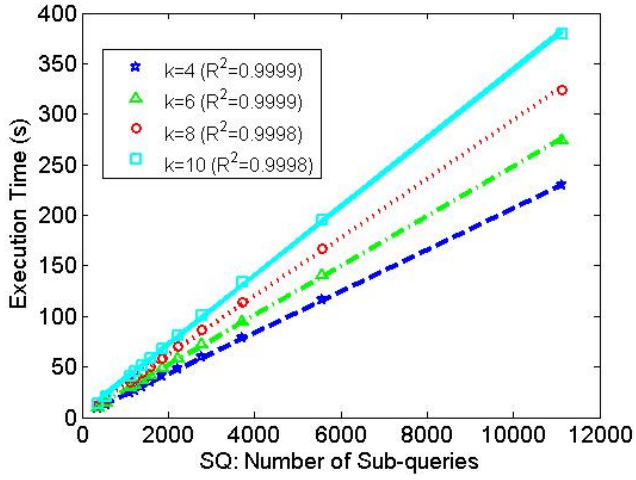

Figure 8. Execution time for different sub-query sizes.

3) Architecture Factors: On an NVIDIA GPU, blocks are scheduled onto multiprocessors, and threads within a block have segregated registers and common access to on-chip shared memory. Threads are scheduled within blocks in warps of size W. The user specifies the number of blocks B_r and the number of threads T_r per block. An important consideration in specifying the execution conditions for GPU kernels is to ensure that the *occupancy*, i.e. the ratio of the number of active warps per multiprocessor to the maximum possible number of active warps, is high.

A specific instance of a GPU has particular values for the number of multiprocessors *MP*, shared memory size S, number of registers R, warp size W, maximum number of blocks $B_{\max}$, maximum threads per block T_{maxB}, and maximum threads per multiprocessor T_{maxMP}. A particular kernel uses a fixed number of registers R_T per thread and a quantity of shared memory S_B per block.

The number of concurrently executing blocks B_a per multiprocessor might be different than B_r/MP due to various resource constraints [17]. B_a is constrained by register usage, shared memory usage, and fixed device capability as follows:

$$B_a = \min\left(\left\lfloor \frac{S}{S_B} \right\rfloor, \left\lfloor \frac{R}{R_T \times T_r} \right\rfloor, \left\lfloor \frac{B_{\max}}{MP} \right\rfloor, \left\lfloor \frac{T_{\text{maxMP}}}{T_r} \right\rfloor\right). \quad (4)$$

If more blocks are requested than can execute concurrently, they are executed sequentially. The expression for B_a represents yet another potential interaction between throughput and false positive rate, as the shared memory used by the Bloom vector, m/8 bytes, lower-bounds the shared memory S_B per block.

The user must choose the number of blocks B_r and threads per block T_r so as to balance the number of blocks per multiprocessor, to ensure all multiprocessors are busy, and to maintain high occupancy.

In the performance model, we define an architecture factor $A(B_r, T_r)$, a function of the requested number of blocks and threads per block, that encodes the impact of block scheduling and thread occupancy on the execution time of the kernel. For optimal choices of B_r and T_r (i.e., $B_r \in B_{\text{opt}}$ and $T_r \in T_{\text{opt}}$), $A(\cdot) = 1$, indicating peak performance. We will not attempt to analytically quantify $A(\cdot)$ for other values of B_r and T_r.

$$A(B_r, T_r) = \begin{cases} 1 & \text{if } T_r \in T_{\text{opt}} \wedge B_r \in B_{\text{opt}} \\ \text{undefined} & \text{otherwise} \end{cases}$$

Elements of B_{opt} are integer multiples of the product of B_a and *MP*. This balances the number of blocks allocated to each multiprocessor:

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

In addition, the requested number of blocks must be within the hard limits set by the architecture:

$$B_r \leq B_{max}.$$

We next empirically investigate the elements of the set B_{opt}. For these experiments, $n = 50000$, $k = 6$, and the GPU is a GTX 480. In the initial experiment, $m = 256$ Kbits and B_r is varied from 1 to 90 for 4 distinct values of T_r. Figure 9 shows the throughput of the system under these conditions. Here, B_a is limited to 1 by the shared memory usage. Since the GTX 480 has $MP = 15$, $B_{\text{opt}} = \{15, 30, 45, ...\}$. Those

values for B_r are the peaks of the individual curves for different T_r. On this plot and those that immediately follow, we mark the peaks of the throughput curve with dark circles so they can be readily identified. The throughput achieved at these peaks is also shown with a horizontal dotted line.

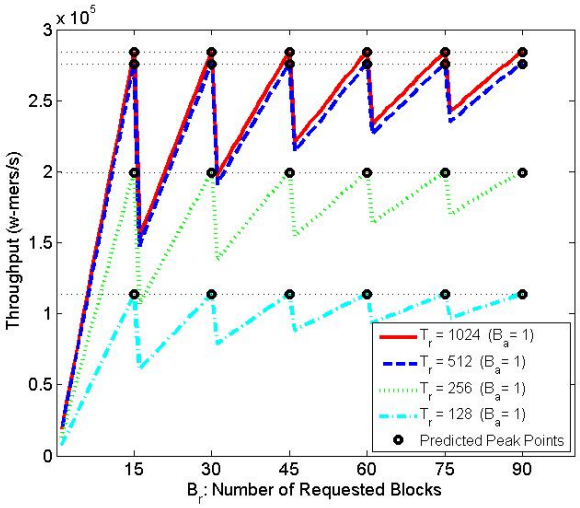

Figure 9. Throughput vs. B_r on GTX 480 ($n = 50000$, $k = 6$, $m = 256$ Kbits).

Reducing shared memory usage per block from $m = 256$ Kbits to 128 Kbits (see Figure 10), there may be up to $B_a = 2$ active blocks on a single multiprocessor. However, when T_r is large enough to deplete the register resources, the limiting factor changes from shared memory to registers. For example, in this case, when increasing T_r from 128 to 512, B_a stays at 2. As as result, throughput peaks every 30 blocks. However, when we further increase the threads per block to 1024, local register resources only allow one active block to launch per multiprocessor. B_a is reduced to 1, and peaks now occur every 15 blocks.

The same behavior is observed when $m = 64$ Kbits and $m = 32$ Kbits. In Figure 11, when using 128 threads, $B_a = 5$; hence, B_r reaches peak throughput at 75 and 150. 512 threads per block results in peaks being hit every 30 blocks.

Similarly, in Figure 12, the active block cycle for 128 threads per block is 120, given that $B_a = 8$. Hence, only one peak is reached within the range of block counts tested.

Performance is consistent with our model even on a Tesla C1060, which has more multiprocessors ($MP = 30$) than a GTX 480. Our results are shown in Figure 13 and Figure 14.

We next turn our attention to T_{opt}. [17] suggests that the requested threads per block T_r should always be a multiple of the warp size W, as the threads are scheduled in units of a warp. It is recommended that at least N_W warps should be used to hide register read-after-write latencies and to ensure that at least one warp is active while others are stalled on blocking or long-latency operations such as memory loads.

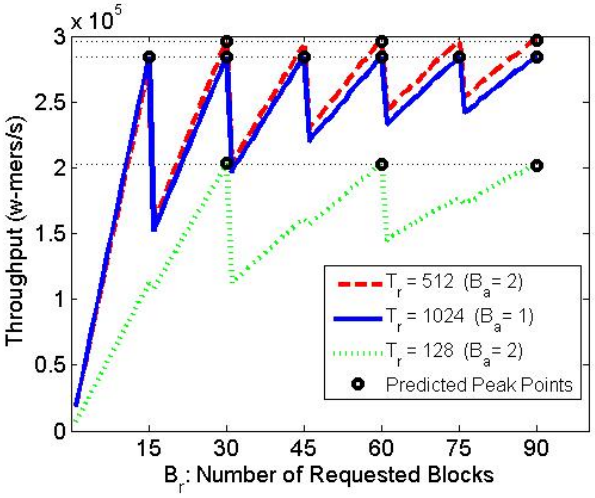

Figure 10. Throughput vs. B_r on GTX 480 ($n = 50000$, $k = 6$, $m = 128$ Kbits).

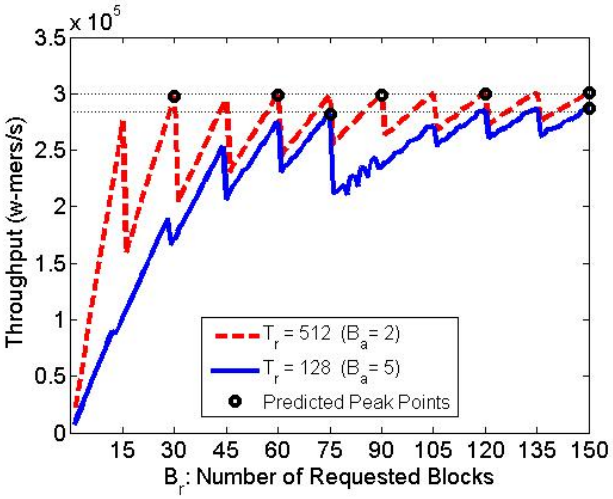

Figure 11. Throughput vs. B_r on GTX 480 ($n = 50000$, $k = 6$, $m = 64$ Kbits).

The requested threads per block must stay within the hard limits set by the architecture, and enough threads should be requested to ensure that all multiprocessors are utilized. These facts yield the following expression:

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

subject to the additional constraints

$$T_r \geq N_W \times W,$$

$$T_r \leq \min\left(T_{\text{maxB}}, \frac{T_{\text{maxMP}}}{B_a}\right),$$

$$T_r \geq \frac{\frac{R}{R_t}}{\frac{B_r}{MP} + 1}.$$

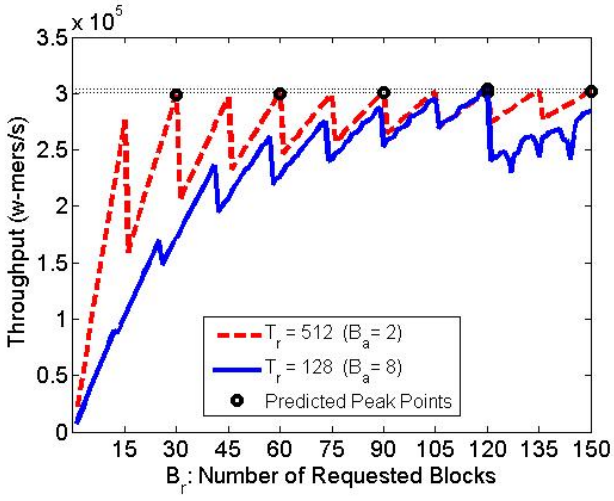

Figure 12. Throughput vs. B_r on GTX 480 ($n = 50000$, $k = 6$, $m = 32$ Kbits).

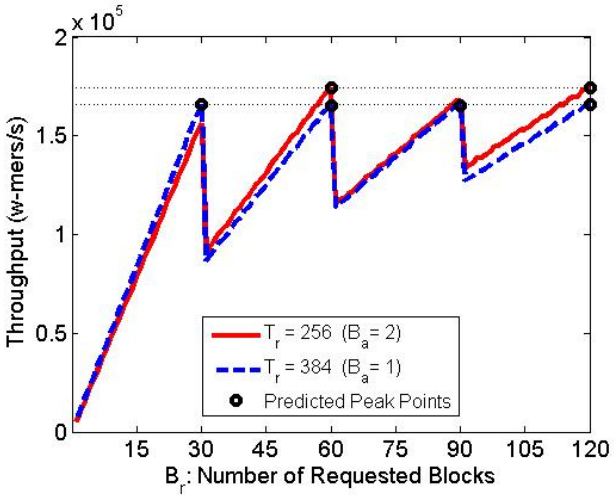

Figure 14. Throughput vs. B_r on Tesla C1060 ($n = 50000$, $k = 6$, $m = 32$ Kbits).

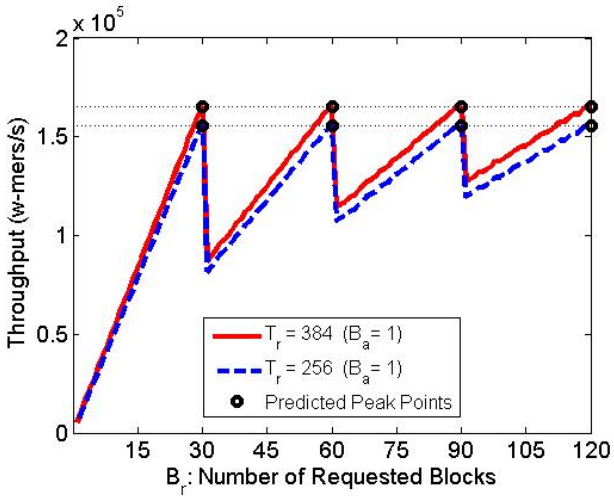

Figure 13. Throughput vs. B_r on Tesla C1060 ($n = 50000$, $k = 6$, $m = 64$ Kbits).

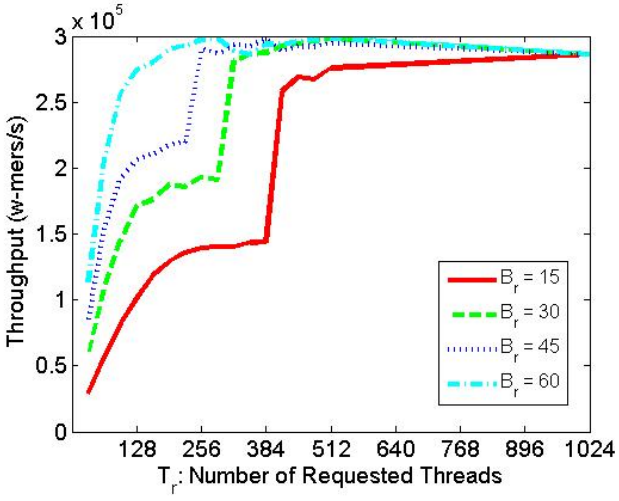

Figure 15. Throughput vs. T_r on GTX 480 ($n = 50000$, $k = 6$, $m = 64$ Kbits).

This model is illustrated in Figure 15, which shows empirical throughput varying with T_r for $n = 50000$, $k = 6$, $m = 64$ Kbits, and $B_r \in \{15, 30, 45, 60\} \subseteq B_{\text{opt}}$, executing on a GTX 480. For each of these curves, the performance is relatively flat once a sufficient number of threads is requested.

C. Data Movement

In Section III-B, our attention was limited to the execution time of the kernel. We next consider the overhead of data movement into and out of the GPU. Figure 16 stacks measured data movement times below kernel execution time for a range of values of the sub-query size n. The data movement times include loading the initial Bloom vector contents, loading the database, and returning the results. As can can be seen in the graph, the kernel dominates the overall time, and I/O is not a bottleneck for this application.

D. Overall Model

The overall performance model is therefore:

$$FPR = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^{k} \tag{5}$$

and

$$Time \propto \frac{k}{n} \cdot Q \cdot DB \cdot A(B_r, T_r) \tag{6}$$

or

$$Time = a_1 \cdot \frac{k}{n} \cdot Q \cdot DB \cdot A(B_r, T_r) + a_0 \tag{7}$$

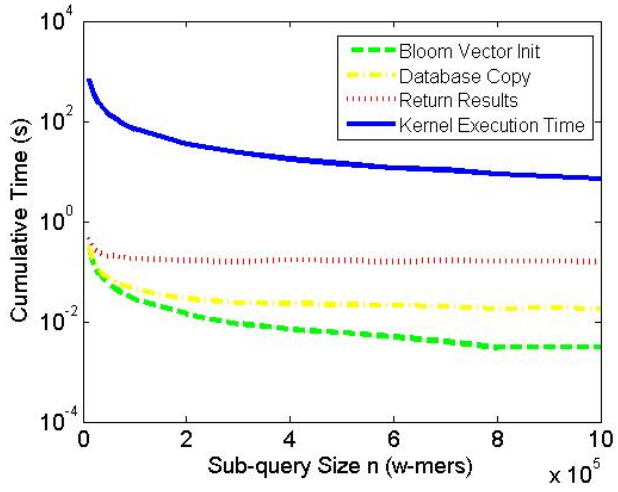

Figure 16. Cumulative execution time for data movement and kernel.

for constant coefficients a_1 and a_0. Also,

$$Tput = DB/Time.$$

Figure 17 establishes the constant coefficients for the GTX 480 by fitting a linear curve to measured data. The range of parameters for this plot include: $k \in \{4, 6, 8, 10\}$, $10000 \leq n \leq 300000$, $m \in \{64, 128, 256\}$ Kbits, $B_r \in B_{\text{opt}}$, and $T_r \in T_{\text{opt}}$.

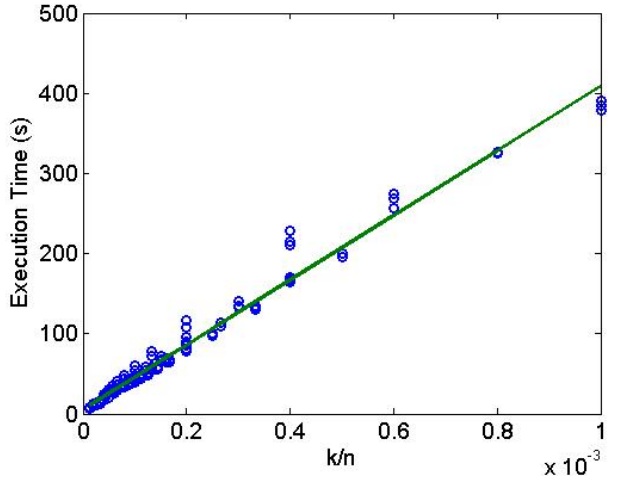

Figure 17. Modeled vs. measured execution time on GTX 480 ($a_1 = 4.01 \times 10^5$, $a_0 = 10$, $R^2 = 0.9909$).

The throughput predicted by the model, and confirmed via experiment, represents a speedup of approximately 35-fold over a 2.6 GHz, quad-core, AMD Opteron system executing the same algorithm (300,000 w-mers/s vs. 8,500 w-mers/s with $n = 50000$ and $k = 6$). On the Opteron, the code was compiled using `gcc` version 4.1.2 at optimization level `-O2` using OpenMP to express thread-level parallelism, but no attempt was made to exploit the SIMD instruction set.

E. Model Use to Evaluate Performance Tradeoffs

It is common practice when using Bloom filters to trade execution speed (throughput) for improved sensitivity (false positive rate) by partitioning the original set into subsets and performing set membership tests on each of the subsets. This is precisely what we are doing in BLAST when the original query sequence is decomposed into sub-queries. Figure 18 illustrates the quantified (via the model) tradeoff between execution time and false positive rate for $k = 6$ hash functions, $m = 256$ Kbits, $B_r \in B_{\text{opt}}$ blocks, and $T_r \in T_{\text{opt}}$ threads per block.

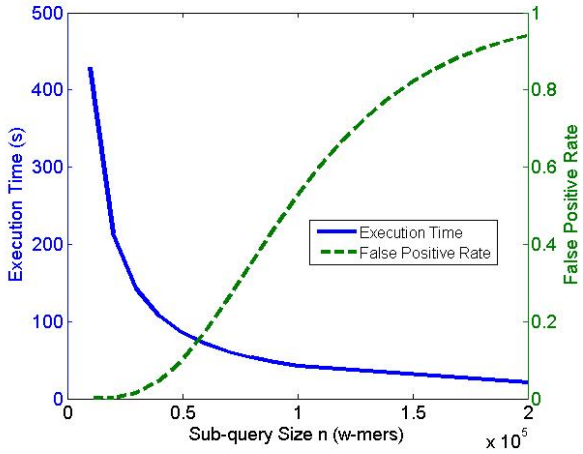

Figure 18. Tradeoff between false positive rate and execution time.

In this way, the user is capable of quantitatively assessing the tradeoff between throughput and sensitivity as controlled by the sub-query size, n.

IV. Conclusion

This paper has presented an implementation of Bloom filters on a GPU that is tailored for use with a pipelined version of BLAST, a commonly used biosequence alignment tool. The implementation makes efficient use of off-chip global memory by ensuring that accesses can be coalesced and makes efficient use of on-chip shared memory by using it for the scattered access to the Bloom vectors.

An analytic performance model is developed that quantitatively assesses the inherent tradeoffs between throughput and sensitivity for a Bloom filter on NVIDIA GPUs. Partitioning the set into subsets (in our case, partitioning the BLAST query into sub-queries) supports improved false positive rates for the Bloom filter operation, but requires greater execution time. The analytic performance model provides guidance not only for assessing the throughput/sensitivity tradeoffs, but also helps the user choose parameters of the

implementation that can have significant performance impact (e.g., number of blocks, number of threads per block, etc.).

As future work we plan to investigate alternative hash functions, as the H3 family of this implementation is well-suited to hardware implementation but there might be better alternatives for GPU implementation. In addition, we plan to marry this GPU implementation of Bloom filters with the existing Mercury BLAST pipeline to assess the utility of a GPU frontend to Mercury BLAST's FPGA.

ACKNOWLEDGMENT

This work was supported by NIH award R42 HG003225. 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] B. Bloom, "Space/time tradeoffs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, 1970.

[3] K. Bratbergsengen, "Hashing methods and relational algebra operations," in *Proc. of 10th Int'l Conf. on Very Large Databases*, 1984, pp. 323–333.

[4] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2004.

[5] 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.

[6] L. Costa, S. Al-Kiswany, and M. Ripeanu, "GPU support for batch oriented workloads," in *Proc. of 28th Int'l Performance Computing and Communications Conf.*, Dec. 2009, pp. 231–238.

[7] S. Datta, P. Beeraka, and R. Sass, "RC-BLASTn: Implementation and evaluation of the BLASTn scan function," in *Proc. of Symp. on Field Programmable Custom Computing Machines*, 2009, pp. 88–95.

[8] X. Gong, W. Qian, Y. Yan, and A. Zhou, "Bloom filter-based XML packets filtering for millions of path queries," in *21st Int'l Conf. on Data Engineering*, 2005, pp. 890–901.

[9] L. Gremillion, "Designing a Bloom filter for differential file access," *Communications of the ACM*, vol. 25, no. 9, pp. 600–604, September 1982.

[10] M. Herbordt, J. Model, B. Sukhwani, Y. Gu, and T. VanCourt, "Single pass streaming BLAST on FPGAs," *Parallel Computing*, vol. 33, pp. 741–756, 2007.

[11] P. Krishnamurthy, J. Buhler, R. Chamberlain, M. Franklin, K. Gyang, A. Jacob, and J. Lancaster, "Biosequence similarity search on the Mercury system," *J. VLSI Signal Processing*, vol. 49, pp. 101–121, October 2007.

[12] 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*, Sep. 2007.

[13] Y. Liu, B. Schmidt, and D. Maskell, "DecGPU: distributed error correction on massively parallel graphics processing units using CUDA and MPI," *BMC Bioinformatics*, vol. 12, no. 1, p. 85, 2011.

[14] M. McIlroy, "Development of a spelling list," *IEEE Trans. on Communications*, vol. 30, no. 1, pp. 91–99, 1982.

[15] S. Mu, X. Zhang, N. Zhang, J. Lu, Y. Deng, and S. Zhang, "IP routing processing with graphic processors," in *Proc. of Conf. on Design, Automation and Test in Europe*, 2010, pp. 93–98.

[16] J. Mullin and D. Margoliash, "A tale of three spelling checkers," *Software – Practice and Experience*, vol. 20, no. 6, pp. 625–630, 1990.

[17] "NVIDIA CUDA Programming Guide 3.0," Feb 2010.

[18] J.-K. Peir, S.-C. Lai, S.-L. Lu, J. Stark, and K. Lai, "Bloom filtering cache misses for accurate data speculation and prefetching," in *Proc. of 16th Int'l Conf. on Supercomputing*, 2002, pp. 189–198.

[19] M. Ramakrishna, E. Fu, and E. Bahcekapili, "Efficient hardware hashing functions for high performance computers," *IEEE Trans. on Computers*, vol. 46, no. 12, pp. 1378–1381, Dec. 1997.

[20] S. Ryoo, C. Rodrigues, S. Stone, S. Baghsorkhi, S.-Z. Ueng, J. Stratton, and W. 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.

[21] H. Shi, B. Schmidt, W. Liu, and W. Muller-Wittig, "Accelerating error correction in high-throughput short-read DNA sequencing data with CUDA," in *Proc. of Int'l Parallel and Distributed Processing Symp.*, 2009.

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

[23] P. Valdurez and G. Gardarin, "Join and semijoin algorithms for a multiprocessor database machine," *ACM Trans. on Database Systems*, vol. 9, no. 1, pp. 133–161, 1984.