Accelerating Bloom Filter Operations Using GPUs and Hardware Accelerators

CSCI 338 Parallel Processing, Final Project (Prof. Kelly Shaw)

Josh Minwoo Kang and Andrew Thai

(Date: November 11, 2019)

***Abstract*—We accelerate the operations of instantiating a Bloom filter (BF) and performing membership queries on it for large text data, first using a Graphics Processing Unit (GPU) with CUDA programming, and then using a BF hardware accelerator. For our GPU acceleration, we explored different parallelization and optimization schemes, such as (1) varying the CUDA block dimensions and (2) optimizing the memory usage with GPU device shared memory and (3) pre-processing the inputs by sorting the elements. We further implemented a hardware accelerator that is integrated into a RISC-V System-on-a-Chip (SoC), and we measured its performance through a cycle-accurate simulator. We report up to $10\times$ speed-up from GPU and $8\times$ from hardware acceleration.**

I. Introduction

Bloom filters (BFs) are space-efficient probabilistic data structures that offer constant-time membership querying [1]. Due to their notable efficiencies, Bloom filters have long attracted the attention of research literature, including reports on multiple variations of BF designs [2] and applications of BFs to various topics pertaining to computer systems, such as networking [3], databases [4], and security [5].

A particular application worth highlighting is the use of BFs to support string matching on large text data. String-based information retrieval is a popular workload especially in web services, which nowadays require implementations of dictionaries that can perform efficiently within a limited critical response time, while also working with increasingly large sets of data [6]. Since the use of Bloom filters already offer significant space-savings, further accelerating the BF operations—such as mapping elements to the filter and testing to check if an element has been previously mapped—is a topic of research that is highly applicable to modern mobile and web-based systems.

In this paper, we explore the viability of utilizing programmable Graphics Processing Units (GPUs) and hardware accelerators as platforms for Bloom filter acceleration. GPUs are particularly suitable for our purpose, since BF operations are highly data-parallel and thus can be efficiently parallelized via Single Instruction Multiple Data (SIMD) hardware. While a similar parallelization can be achieved with shared-memory multiprocessors using the POSIX Threads library, we opt for GPUs in order to observe substantial speed-ups that arise from the sheer volume of Streaming Processors (SPs) that GPUs can offer. Furthermore, we also explore an implementation of a Bloom filter hardware accelerator which can potentially provide even greater improvements in performance. The goal of this paper is thus to demonstrate a straight-forward yet scalable model for implementing high-performance BF-based systems.

This paper is organized as follows. In Section 2, we provide an overview of the Bloom filter: what it is and why it might be used. Section 3 discusses our sequential implementation of this data structure; Section 4 describes the CUDA implementation and Section 5 discusses potential optimizations that can be applied when using CUDA programming. Section 6 demonstrate our implementation of a Bloom filter hardware accelerator. We further talk about our results and why they may have occurred (Section 7), and we finally conclude with comments on our findings and future work that stem from our results (Section 8).

II. Background

Given a set or dictionary of n elements, it is often the case that users want to check if an element is in that set, whether their objective is to add, read, search, or remove a particular piece of data. However, a naive implementation of such a data structure would require a number of memory accesses which are slow in general. For large values of n, it is further impossible to store the set in the cache, so the data must be stored in lower-level memory. Thus, systems supporting membership queries can potentially require a substantial number of expensive memory accesses. The solution to this problem is to use the Bloom filter.

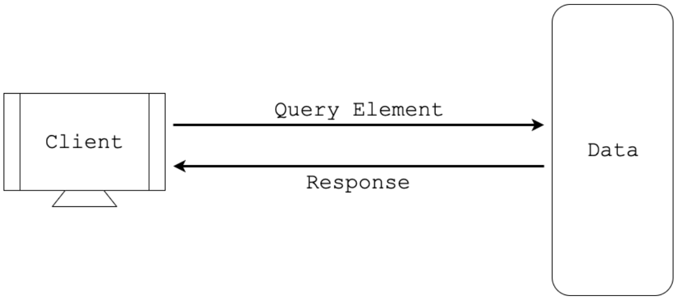


Fig. 1. Diagram representing dictionary-based querying.

A. Overview of the Bloom Filter

The main function of a Bloom filter is to test whether an element is part of a set in a space and time efficient manner. However, the BF itself does not store the data of each element (a separate data structure should be used to actually store the elements). Rather, a BF is a logical array of m bits initialized with 0's that keeps track of whether an element has been mapped to a position or not. That is, whenever an element is added, the BF will use its hash function(s) to place a 1 where the element would be mapped to. Then, the search function of the bloom filter checks whether an element is not in the set, or potentially in the set, by checking the index of where the element would map to. Note that the search function cannot guarantee that an element is in the set since a 1 only indicates that *some* element(s) was mapped to that index.

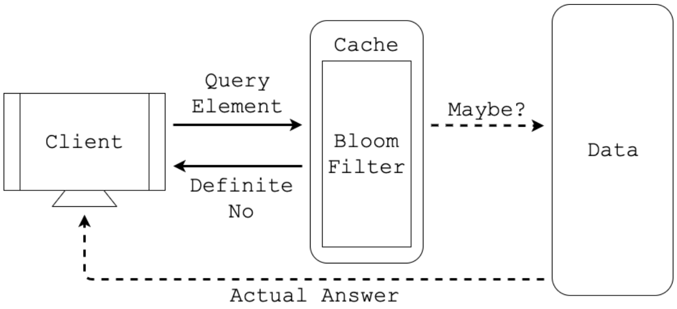


Fig. 2. Diagram representing a more space-efficient implementation of a dictionary using a Bloom filter that fits in the cache.

B. Bloom Filter Emulation

Our implementation will be reading in text from input files, first placing words into the set. Then, we will have another input file with words to search. Thus, the key areas to focus on are the mapping and searching functions.

We should note that we are only interested in the performance of the Bloom filter. Thus, this program will merely emulate the functionality of the Bloom filter–none of the data we read will actually be stored in a dictionary/set.

The algorithms necessary for the functionality of a Bloom filter will be further specified in the following discussion of the serial implementation.

III. Baseline Serial Implementation

Given a text file that we want to add to our Bloom filter, the natural way to proceed is to read each word one-by-one from a file stream. Then, for each `String`, we apply k hash functions to it–generating k indices that should be set in the Bloom filter. This constitutes what we will refer to as the `Map()` function.

Similarly, if we are given a text file with `Strings` to search in our Bloom filter, we will read each word individually from the file stream. Then, we apply the same k hash functions to these words to obtain k indices to check in the Bloom filter. If the bits at those locations are all set to one, we report that as a positive–the item is contained. This function will be referred to as `Test()`.

A. Representing the Bloom Filter in C

As discussed previously, the goal of our program is to emulate, not fully implement a Bloom filter. That is, we aren't storing the data, nor are we actually placing our Bloom filter in the cache. Instead, we are merely interested in how CUDA and hardware accelerators can be used to improve the mapping and testing algorithms of a Bloom filter.

The C Programming Language unfortunately does not have native support for a bit array. Thus, in order to implement a bit array in C, we would have to use an array of a type defined in C, such as `char`, and calculate an index whenever we wanted to edit the array. This introduces a few extra operations which, when scaled up to large inputs, could pose as a performance issue (although most likely very slight). Since the actual mapping and testing algorithms don't necessarily require a bit array, we chose to use an `unsigned char` array trading off space for a slight improvement in performance.

B. `Strings` and Pre-Processing

One of the major differences between C and other programming language is the implementation of what we know as strings. To simplify our code, we defined a `String` struct that holds a character array of a pre-defined size. This allows us to more easily initialize the text arrays.

We also opted to remove all instances of punctuation since some of our inputs involve sentences and stories. However, capitalization of characters along with numerical values are left unchanged.

C. Hashing: `HashString()` and `Hash()`

Our implementation of hashing first employs the use of Horner's rule for polynomial evaluation in what we will call our `HashString()` function. That is, we treat each character in a `String` as a part of the terms in a polynomial. Using the numerical value of the characters, we can then solve the polynomial to obtain a hash value for that word. We do this iteratively over every character using the equation that follows:

$$hash = (hash * 32) + hash + charAt(index)$$

In C, this is implemented as such:

```
hash += (hash << 5) + *(str++)
```

After obtaining the final value from `HashString()`, we use our `Hash()` function to then apply k more computations to the hash value to obtain the k hash values to be used in the Bloom filter. This is done by iteratively computing a displacement that will ensure that our k indices are dispersed. Lastly, these values are computed in modulo m (recall that m is the number of bits in the filter) so that every index is valid.

D. Adding and Checking Elements: `Map()` and `Test()`

Both the `Map()` and `Test()` functions are implemented in similar fashions. Both will call `Hash()` to obtain the k indices for a given word. `Map()` sets the bits at the k indices in the Bloom filter to 1 while `Test()` checks if each of those bits have been set to 1.

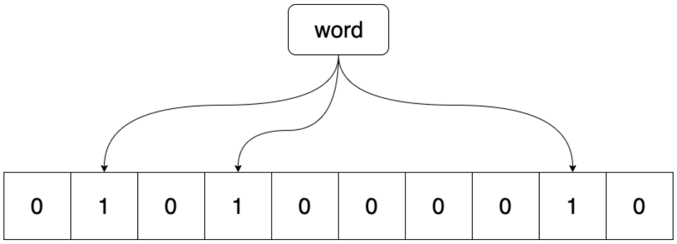


Fig. 3. Example Mapping with 3 Hash Functions

Note that there is no remove functionality in the original Bloom filter since we don't necessarily know how many elements a specific bit corresponds to. Setting a bit to 0 may potentially cause false negatives since that bit could have been set by a element that was not removed. Thus, more recent work has been done to implement the ability to remove elements from the filter, but this is not a focus in our project.

E. A More Parallelizable Approach

The process of mapping and testing individual words from a file stream is not one that can be done in parallel as we only have access to the current `String` in our buffer. Thus, an alternative approach is to add every word from a file into a dynamic array that grows when it is at capacity. After the entire file is placed into the array, we can proceed with the corresponding `Map()` or `Test()` function.

Note that for the serial implementation, this means that each word is seen twice. That is, placing the words into an array is an unnecessary step since we could have placed the word into the Bloom filter as we read it in. Thus, when we compare the performance of the serial and parallel implementation, we will only measure the time it takes for the serial program to perform the `Map()` and `Test()` functions. We will also have a separate test that will measure the time it takes to complete the two functions when the words are processed directly from the files as this is the more natural and efficient method in a sequential implementation.

IV. CUDA Implementation

After serially reading the entire text files into arrays–one for mapping and the second for testing–we can proceed with a parallel implementation of `Map()` and `Test()`. Note that processing each word requires very little data and relatively few steps as we only need to calculate k hash values per word. Thus, we are more interested in improving the running time when we have a substantial number of `String`s to map and test. Since we are dealing with a large number of words that will all have the same hashing operations applied to them, we decided to proceed with CUDA for the parallelization.

A. Organization of Parallelism

The placement of the words in the files into one-dimensional arrays provides a convenient way to organize the data for our CUDA implementation. We will use one-dimensional blocks of varying sizes. Each thread within a block will be responsible for completely performing `Map()` or `Test()` functions on a single word in a given `String` array as seen in Figure 4.

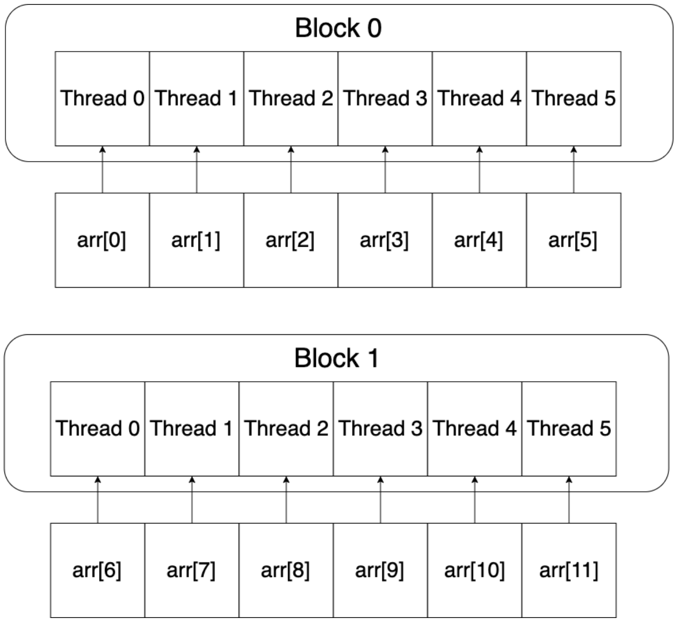


Fig. 4. CUDA Block Organization with Block Size 6

In our implementation, the block size manually defined to be some float, typically a power of 2. Then, the grid dimension–the number of blocks–is found by dividing the number of words in the array by the block size and rounding the result up. We can then find the index of the array using

```
blockIdx.x * blockDim.x + threadIdx.x
```

allowing each thread to process a unique word as seen in Figure 4.

B. Choice of Block Dimensions

We opted to only use one-dimensional blocks since the data we are processing is already organized in a 1-D array. In the case of a two-dimensional block, we would have to add additional computations to determine both the row and column indices. Since we also know that the computations required for each word are entirely independent, it is not necessary to bunch words together or have easy access to neighboring words (which was, on the contrary, completely necessary when we blurred images). However, we will explore the organization of the `String` array as a potential non-parallel optimization later in this paper.

V. CUDA Non-Parallel Optimizations: Shared Memory and Sorting

Our current CUDA implementation closely follows our serial implementation, leading to the question of whether there are modifications we can make in order to potentially improve performance. In this section, we attempt to decrease running time by using shared memory and sorting.

A. *Shared Memory*

We proceeded with two applications of shared memory in our program. The first is the use of a shared copy of the Bloom filter for use in the `Map()` kernel. That is, each block of threads will have a shared version of the Bloom filter such that, when all threads are done mapping, this local Bloom filter is copied into the actual Bloom filter on the device. The second application of shared memory we used was a shared miss counter for our `Test()` function. That is, each block had a shared counter that would be incremented whenever a thread detected a miss. Then, after all of the threads completed testing a word, we would add the count in this counter to the device counter.

B. *`Map()` with Shared Memory*

To implement the shared Bloom filter in the `Map()` kernel, we first initialized the shared version of the Bloom filter by assigning a portion of a shared array of size m to each thread to zero out. Then, after all threads finished mapping their word to the shared filter. We had each thread update a different portion of the actual Bloom filter with the results in the shared filter.

There were several considerations that had gone into the implementation of this merging. The first was that we wanted to avoid setting any bits in the device filter to 0. Thus, we could not directly copy the bits in the shared array to the Bloom filter. A thread could potentially set a bit to 0 after another thread set it to 1. We also wanted to reduce the number of writes to the device array. If we were to copy the entire shared array over, we would do m writes (where some are to the same indices). But note that the implementation without the shared memory only writes when a bit is to be set, which most likely occurs less than m times since we don't expect every bit to be set in a Bloom filter.

Thus, we chose to proceed with branching. Using an `if` statement, we only update a bit in the device filter if that bit is set in the shared filter. After preliminary testing, we found that the drawbacks of writing outweighed the drawbacks of branching. Thus, we settled with branching in our final implementation.

C. *`Test()` with Shared Memory*

The implementation of `Test()` with shared memory begins with the initialization of a shared integer. Since it is a single value, we let thread 0 initialize its value to zero. Then, after all of the threads sync, each thread checks if its word is a hit or miss. A major issue to consider is the case where two threads attempt to increment the counter at the same time, resulting in an increase of 1, instead of 2, since they both read the same value before they added. To avoid race conditions, we increment the shared counter using `atomicAdd()`. Then, after every thread has completed testing their word, we have thread 0 add the result in the shared counter to the device counter, once again using `atomicAdd()`.

D. *Sorting Before Processing*

Consider `Hash()` and `HashString()`. These somewhat complex functions are repeatedly called and used on extremely large data. In the cases where we might expect similar, or even repeated data, we want to make use of the cache. We thus take advantage of technique known as a memoization.

Memoization is a process where we store the results of expensive function calls. Then, when we see input that we previously used, we can return the cached result instead of recomputing the data, allowing us to save space. This provides strong motivation for extremely large files such as books and novels where we would expect common words such as ”the” to appear a significant amount of times. Thus, since sorting the input arrays before processing them places all of the same words together, we expect memoization to play a large role in optimizing the performance of the Bloom filter.

VI. Bloom filter Hardware Accelerator

We further accelerate the Bloom filter operations using a hardware accelerator. The framework in which we develop our accelerator is based on the open-source Rocket Chip System-on-a-Chip (SoC) Generator [7] and the RISC-V Instruction Set Architecture (ISA) [8]. Alongside the Berkeley Rocket core, we attach our accelerator as a RoCC co-processor, which we implement using the Chisel paremeterizable hardware description language [9].

A. *SoC Architecture*

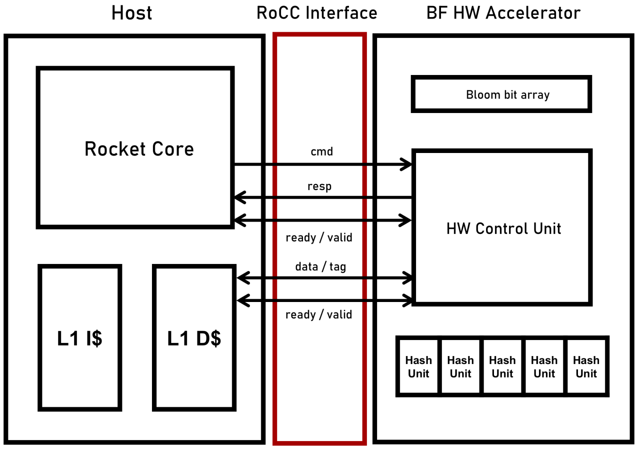
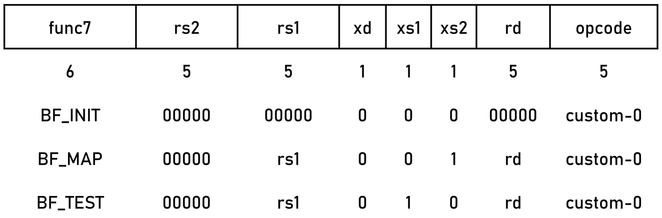


Fig. 5. High-level diagram of the SoC architecture

The high-level SoC architecture is described in Figure 5. The host and the accelerator are interfaced through the RoCC interface, which ensures the integrity of data transfers through handshaking signals (ready-valid). The host processor further invokes the accelerator and directs it to execute specific functionalities using custom instructions that extend the RISC-V ISA. For our BF accelerator, we implement three of such instructions: initializing and/or resetting the hardware Bloom filter (`BF_INIT`), mapping an input to the BF (`BF_MAP`), and finally, checking whether an element was previously mapped to the filter (BF_TEST). This set of instructions offers sufficient support for BF-based membership queries; further details on the instruction encodings are summarized in Figure 6.



func7	rs2	rs1	xd	xs1	xs2	rd	opcode
6	5	5	1	1	1	5	5
BF_INIT	00000	00000	0	0	0	00000	custom-0
BF_MAP	00000	rs1	0	0	1	rd	custom-0
BF_TEST	00000	rs1	0	1	0	rd	custom-0

Fig. 6. Custom RoCC instructions for BF hardware accelerator

Note that our implementation is fairly direct and simplified compared to previous designs of Bloom filter accelerators [10]. Most notably, this implementation does not consider the use of the shared L1 data cache, but instead the accelerator directly receives the input as data contained in one of the host processor's registers. Due to such limitation, our current implementation only processes a single word on each instruction.

However, our Bloom filter accelerator does consist of hardware units that directly support `Map()` and `Test()` functions, which provides the potential to speed up operations on large amounts of data. Recall that both functions require k hash functions to be applied to the results of `HashString()` on each input word. Our accelerator specifies hardware units that execute all k of such hash operations simultaneously and thus significantly reduces the number of cycles required to complete the hashing.

B. Simulation and Benchmarking

While the hardware architecture is specified using Chisel, we simulated our accelerator using Verilator, which is a cycle-accurate behavior model implemented in C++. As our benchmarks, we used `Map()` and `Test()` timing programs of the same style we used to benchmark the sequential and CUDA implementations. Difference arises in the specific implementations of the hash, map and test functions due to the fact that many of the GNU C library functions are not supported in the RISC-V cross-compiler toolchain. Instead of reading from files and processing into arrays of `String`s, we used constant arrays of words that were computed and organized beforehand. Since we look at speedups by comparing the software and hardware versions, this change does not affect the accuracy of our comparison.

Performance measurements were conducted using the RISC-V `RDCYCLE` instruction which reports how many cycles it took to proceed from line of code to another. The benchmark programs were written in C with functions invoking RV assembly instructions and were then compiled to Linux binaries using the FireMarshal RISC-V workload generation tool. Compiled binaries are compatible to be run using the Verilator-based simulations.

Further note that we ran our timing scripts on the simulation results "bare metal," which is convenient since we can disregard complications from predicting and supporting the interactions with the O/S. However, the fact that our implementation is in bare metal does indicate that the usage of the accelerator is limited to a single user. Context-switching is not supported and thus in order to build an accelerator that can be used in a more general system, the hardware implementation must be able to preserve its hardware context or establish appropriate communication with the host processor, such as through hardware interrupts [11], [12].

VII. Results

We used three text files of different sizes as our benchmarks: small (10,000 words), medium (466,551 words), and large (1,095,695 words). Note that both the small and medium text files are sorted dictionaries. The large file is a collection of books and thus contains repeated words in an unsorted ordering. Also note that we specifically only measured the time it takes for `Map()` and `Test()` to perform. All other pre-processing work, such as reading words into the arrays or initializing data, is not measured. We also chose to record the performance of the two functions separately so we have a better grasp of how changes in implementation impact performance. The data was gathered by taking the average running time of multiple tests.

A. Sequential Results

The first set of sequential results measure the time it takes to complete both `Map()` and `Test()` when the data was placed in to arrays. We have the following data in Table 1.

	Small	**Medium**	**Large**
Time to `Map()`	1.059 ms	40.966 ms	88.408 ms
Time to `Test()`	0.982 ms	37.513 ms	79.639 ms

Table 1. Sequential performance when using `String` arrays.

If we choose to map and test words as we read them from the files, we have the following results seen in Table 2 instead.

	Small	**Medium**	**Large**
Time to `Map()`	2.714 ms	104.95 ms	196.04 ms
Time to `Test()`	2.432 ms	98.877 ms	183.08 ms

Table 2. Sequential performance when reading data directly from files.

As expected, we can see that directly reading from a file input takes more time compared to having the words pre-read in an array. Thus, we should note that comparisons with all implementations of the Bloom filter will be done using the data obtained from mapping and testing the words already placed in the array since that is how the data will be presented to the parallel versions of the program. The results from the direct input should be interpreted as a realistic running time of a sequential implementation of a Bloom filter when we want to add or check n items.

B. CUDA Results

Without the use of shared memory or other potential CUDA optimizations, we obtained the following results as in Table 3.

	Small	Medium	Large
`Map()`			
Block Size 8	0.288 ms	14.46 ms	26.72 ms
Block Size 16	0.197 ms	9.674 ms	17.57 ms
Block Size 32	0.154 ms	7.441 ms	13.30 ms
Block Size 64	0.128 ms	6.185 ms	9.875 ms
Block Size 128	0.131 ms	6.264 ms	9.727 ms
`Test()`			
Block Size 8	0.316 ms	16.13 ms	31.17 ms
Block Size 16	0.208 ms	10.47 ms	19.83 ms
Block Size 32	0.157 ms	7.887 ms	13.88 ms
Block Size 64	0.143 ms	7.282 ms	12.13 ms
Block Size 128	0.149 ms	7.380 ms	12.45 ms

Table 3. CUDA performance without optimizations.

Before we directly compare the results of the serial and CUDA implementations, we should note the difference in running time between `Map()` and `Test()`. In the serial programs, the `Map()` operation took more time. However, in the CUDA version, `Map()` is faster. This is most likely due to the use of `atomicAdd()` to increment the miss counter in the CUDA kernel for `Test()`. If any threads attempt to add to the counter at the same time, then the adds will essentially be done sequentially, slightly increasing the running time of `Test()`.

We also find that, as block size increases, CUDA performance also improves until we reach the block size of 128 which is slightly slower than using a block size of 64. Potential causes for this is that less blocks can fit on each streaming multiprocessor and that there are not enough registers available for all of the threads to run concurrently. Depending on the number of threads we can fit on the streaming multiprocessor, it is also possible that a block size of 128 doesn't efficiently fit.

A peculiar data point can be seen for the `Map()` function using block size 128 on a large file. Note that there is decrease in running time from 64 blocks. This does not follow the trend of the other data points for a block size of 128. A potential cause for this is the fact that the large file is not a dictionary, but a collection of stories. Thus, we would expect it to have several instances of the same word. Having a larger block size means that more of these similar words are being processed within the same block. That is, threads that are operating on the same word in a block can make use of memoization, where the cache stores the results of relatively expensive operations. Thus, once one thread computes the hash values of a `String`, all other threads with the same word can complete their task quicker. This provides the motivation for a non-parallel optimization where we sort the `String` arrays before we process them.

Now we will discuss the difference between our serial and CUDA implementations. Comparing Tables 1 and 3, we find that the CUDA implementation is significantly faster than the serial program. We can observe that parallelizing the functions using GPUs offer at least $3\times$ and up to $10\times$ performance improvements, depending on the block size used in our CUDA program.

C. CUDA Optimization: Shared Memory Results

After implementing a shared Bloom filter for `Map()` and a shared miss counter for `Test()` we get the results seen in Table 4.

	Small	Medium	Large
`Map()`			
Block Size 8	297.8 ms	13894 ms	32573 ms
Block Size 16	75.26 ms	3508. ms	8204. ms
Block Size 32	19.38 ms	901.6 ms	2092. ms
Block Size 64	5.127 ms	238.0 ms	546.1 ms
Block Size 128	1.337 ms	62.14 ms	138.7 ms
`Test()`			
Block Size 8	0.363 ms	18.35 ms	36.27 ms
Block Size 16	0.257 ms	12.67 ms	24.88 ms
Block Size 32	0.160 ms	8.067 ms	14.24 ms
Block Size 64	0.144 ms	7.191 ms	12.02 ms
Block Size 128	0.148 ms	7.273 ms	12.24 ms

Table 4. CUDA performance when using shared memory.

The results for `Map()` present a stark increase in running time. That is, our shared memory implementation of the `Map()` operation is significantly slower than both the original CUDA and serial versions of the program. On the other hand, we find that using a shared counter for `Test()` improves performance for larger block sizes, specifically 64 and 128.

Since a block size of 64 was our fastest block configuration for our CUDA program without shared memory and shared memory improves performance with a block size of 64, we find that using shared memory for the `Test()` function is a successful optimization. We will now explore why these changes in performance occur.

To see why `Map()`'s performance degrades, consider the following pseudo-code for the CUDA implementation of `Map()` without shared memory and the implementation with shared memory. Note that the code that obtains the hash values that each word maps to is omitted as it is identical in both cases.

Algorithm 1 `Map()` without Shared Memory

```
$i \leftarrow$ threadID $+$ blockID $*$ blockDim
if $i$ is valid then
    for $j = 0 \ldots k - 1$ do
        filter[j] = $j^{th}$ hash value
    end for
end if
```

Algorithm 2 `Map()` with Shared Memory

```
$i \leftarrow$ threadID + blockID $*$ blockDim
initialize shared_filter
if $i$ is valid then
    for $j = 0 \ldots k-1$ do
        shared_filter[j] = $j^{th}$ hash value
    end for
end if
for every bit that was set in s_filter do
    set that bit in the actual filter
end for
```

Note that in both implementations, we will have to access the actual bloom filter in device memory p times, where p is the number of bits we had to set. Thus, using shared memory in this case does not help us decrease the number of accesses to device memory. In fact, the use of shared memory forces us to do two additional things: initialize the shared memory array and copy the results back to the device Bloom filter. Thus, we end up doing additional operations. Thus, `Map()` is less efficient with shared memory.

Now consider `Test()`. Unlike `Map()`, if we compare the results in Table 4 to that of Table 3, we find that, for larger block sizes, performance does improve with the use of shared memory. However, smaller blocks did perform slightly worse.

A likely reason for this is how shared memory works in CUDA. Shared memory is only shared among threads in the same block. Thus, using larger blocks means that a larger number of threads can access and update the miss counter. That is, when we use larger blocks, we have less writes back to device memory since every thread updates the shared counter instead of the device counter. Then, only thread 0 writes the result the actual counter, resulting in a significant decrease in writes to global memory improving the performance of `Test()`.

D. CUDA Optimization: Sorting Results

Note that since the small and medium files are dictionaries, they are already sorted so that data is omitted in this section. Instead, we are interested in improvements on unsorted data, more specifically, the large file. We find the results seen in table 5, where the pre-sort data is from table 3. In these tests, we use the CUDA implementation without shared memory so we can compare our data to the original CUDA performance without any of our non-parallel optimizations.

As we can see, for every block size, sorting the `String` array before calling the CUDA kernels improved the performance of both `Map()` and `Test()`. We can see that the running time decreases by a few milliseconds for all block sizes. Thus, we can conclude that sorting the array improves the performance of the CUDA kernels. However, note that the time to sort the array serially outweighs the time saved from sorting the array, so this optimization does not necessarily improve the speed of the program as a whole. Since we are only interested in the performance of the `Map()` and `Test()` kernels, we consider this an optimization.

	Pre-Sort	**With Sort**
`Map()`		
Block Size 8	26.72 ms	19.72 ms
Block Size 16	17.57 ms	11.56 ms
Block Size 32	13.30 ms	8.143 ms
Block Size 64	9.875 ms	7.177 ms
Block Size 128	9.727 ms	7.484 ms
`Test()`		
Block Size 8	31.17 ms	24.59 ms
Block Size 16	19.83 ms	14.31 ms
Block Size 32	13.88 ms	9.380 ms
Block Size 64	12.13 ms	7.336 ms
Block Size 128	12.45 ms	7.569 ms

Table 5. CUDA performance changes on the large file after sorting the input.

E. Combining Successful CUDA Optimizations

Our previous results found that using shared memory for `Test()` and pre-sorting the array provides the best performance with a block size of 64. Thus, we found it necessary to include final results on how these three factors perform together to improve the performance of `Test()`. We get the following data in Table 6. Once again, we only list the results of the large file since sorting only effects unsorted texts. Refer to Table 4 to see the optimal `Test()` results on the small and medium files and Table 5 for the best results for `Map()`.

	Shared	**Sorted**	**Both**
`Test()`			
Block Size 64	12.02 ms	7.336 ms	7.315 ms

Table 6. CUDA performance of `Test()` on the big text file using successful optimizations.

Thus, we find a minor improvement over only using sorting as a means of optimization. This suggests that sorting the array provides a more significant impact compared to shared memory in terms of improving the performance of the `Test()` kernel.

F. Hardware Accelerator Results

Performance measurements of the Bloom filter hardware accelerator are summarized in Table 7. Data was collected from running RISC-V Linux binaries compiled using FireMarshal on our Verilator-generated simulation. We compare two contrasting versions, one which makes use of the hardware accelerator via custom assembly instructions and the other solely implemented in software.

Across different input data sizes and both for `Map()` and `Test()` functions, using the BF RoCC hardware accelerator returns significant improvements in performance. Results demonstrate that using the hardware accelerator offers performance improvements up to 8× fewer cycles than what is required to run the same program solely using software.

	Hardware	**Software**
`Map()`		
Input Size 11	646 cycles	5614 cycles
Input Size 30	1870 cycles	15346 cycles
Input Size 50	2907 cycles	25052 cycles
`Test()`		
Input Size 11	681 cycles	3467 cycles
Input Size 30	1872 cycles	11520 cycles
Input Size 50	3082 cycles	13894 cycles

Table 7. Hardware accelerator performance.

In fact, we notice that the speed-up increases along with the input data size. This is most likely due to that setting up the hardware accelerator costs a static overhead, while further utilizing the accelerator incurs a run-time directly proportional to the frequency of hardware invocations, thus linear with respect to the size of the input. From this, we infer that testing against larger data sets would potentially return even greater scalings in performance. Unfortunately, with our current methodology using Verilator-based simulation, testing with input sizes comparable to the texts we used for our sequential and CUDA implementations is infeasible because the simulation time simply becomes exceedingly long. However, once we map our hardware configuration onto a physical device, either taping out as an Application Specific Integrated Circuit (ASIC) or configuring on a Field Programmable Gate Array (FPGA), we will be able to overcome such limitations of using software simulations.

Furthermore, it is important to note that using hardware accelerators often not only enhances system performance, but also improves energy-efficiency. While measuring and comparing energy consumption of our BF implementations would require analysis based on physical hardware and would thus be outside the scope of this paper, a potentially important motivation for building BF hardware accelerators is to implement dictionaries on low-power embedded systems, as suggested by Lyons and Brooks [10].

VIII. Conclusions and Future Work

Based on sequential and CUDA implementations of the Bloom filter program as best summarized in Table 6, we observe that GPU acceleration without any optimizations already provided up to 9× speed up with the optimal block size.

Once running the CUDA with successful optimizations, we found that the best performance we could get from our GPU acceleration passed the 10× running time reduction mark. That is, for the large file, our average `Map()` time decreased from 88 milliseconds to only 7.1 milliseconds. Similarly, our `Test()` running time improved from 79 milliseconds to just 7.315 milliseconds. Thus, while the sequential Bloom filter was already very fast with large inputs (about a million words), the CUDA program significantly improved the performance.

However, recall from our discussion in the results section that we omitted the time it takes to sort. Compared to `Map()` and `Test()`, sorting is a significantly slower algorithm. Thus, the sorting optimization is not one that is practical in an actual implementation. Despite this, note that even without sorting, our CUDA program is still several times faster than the serial version. Thus, CUDA is an effective way to improve the performance of a Bloom filter when we want to map and test large sets of data.

Also, the BF hardware accelerator can be further improved by adding additional hardware and logic to support parallel `Map()` and `Test()` processing, akin to what we accomplished using CUDA GPU programming. Instead of passing data between the host and accelerator using registers, we can also imagine an implementation that makes use of the shared L1 data cache. To further overcome the limitations posed by the serial memory interface, a future accelerator design may include a systolic pipelining scheme that can optimize the data flow between the host and the Bloom filter accelerator. In such implementations, we expect that the hardware accelerator will be able to return much more significant performance improvements than what we observed in this project, and the speedup may potentially be greater than what GPUs are capable of offering.

References

[1] L. Luo, D. Guo, R. T. B. Ma, O. Rottenstreich, and X. Luo, "Optimizing Bloom Filter: Challenges, Solutions, and Comparisons," *arXiv e-prints*, p. arXiv:1804.04777, Apr 2018.

[2] D. Li, H. Cui, Y. Hu, Y. Xia, and X. Wang, "Scalable data center multicast using multi-class bloom filter," in *2011 19th IEEE International Conference on Network Protocols*, pp. 266–275, Oct 2011.

[3] F. Angius, M. Gerla, and G. Pau, "Bloogo: Bloom filter based gossip algorithm for wireless ndn," in *Proceedings of the 1st ACM Workshop on Emerging Name-Oriented Mobile Networking Design - Architecture, Algorithms, and Applications*, NoM '12, (New York, NY, USA), pp. 25–30, ACM, 2012.

[4] G. Lu, Y. J. Nam, and D. H. C. Du, "Bloomstore: Bloom-filter based memory-efficient key-value store for indexing of data deduplication on flash," in *2012 IEEE 28th Symposium on Mass Storage Systems and Technologies (MSST)*, pp. 1–11, April 2012.

[5] E. A. Durham, M. Kantarcioglu, Y. Xue, C. Toth, M. Kuzu, and B. Malin, "Composite bloom filters for secure record linkage," *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, pp. 2956–2968, Dec 2014.

[6] A. Iacob, L. M. Itu, L. Sasu, F. Moldoveanu, and C. Suciu, "GPU accelerated information retrieval using Bloom filters," pp. 872–876, 10 2015.

[7] K. Asanović, R. Avizienis, J. Bachrach, S. Beamer, D. Biancolin, C. Celio, H. Cook, D. Dabbelt, J. Hauser, A. Izraelevitz, S. Karandikar, B. Keller, D. Kim, J. Koenig, Y. Lee, E. Love, M. Maas, A. Magyar, H. Mao, M. Moreto, A. Ou, D. A. Patterson, B. Richards, C. Schmidt, S. Twigg, H. Vo, and A. Waterman, "The Rocket Chip Generator," Tech. Rep. UCB/EECS-2016-17, EECS Department, University of California, Berkeley, Apr 2016.

[8] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The risc-v instruction set manual, volume i: User-level isa, version 2.0," Tech. Rep. UCB/EECS-2014-54, EECS Department, University of California, Berkeley, May 2014.

[9] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*, pp. 1212–1221, June 2012.

[10] M. J. Lyons and D. Brooks, “The Design of a Bloom Filter Hardware Accelerator for Ultra Low Power Systems,” in *Proceedings of the 2009 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED ’09, (New York, NY, USA), pp. 371–376, ACM, 2009.

[11] H. Mao, “Hardware acceleration for memory to memory copies,” Master’s thesis, EECS Department, University of California, Berkeley, Jan 2017.

[12] J. Koenig, “A hardware accelerator for computing an exact dot product,” Master’s thesis, EECS Department, University of California, Berkeley, May 2018.