K-mer Counting Using Bloom Filters with an FPGA-Attached HMC

***Abstract*—As FPGAs are integrated into to the cloud, they become useful in a number of areas where they were not traditionally considered, such as processing genomics data. For many genomics applications, such as K-mer counting, the off-chip DRAM (and sometimes SRAM) memory subsystems provided by most FPGA-based boards for high capacity storage are not efficient. Recently new styles of memory have been developed, though their role in reconfigurable computing systems can be unclear. One of the challenges these memory systems present to FPGA designers is identifying how they can be used in current systems, and what new applications become possible. In this paper we describe how and why K-mer counting is one such use for an FPGA-attached Hybrid Memory Cube (HMC). The HMC's high random-access rate is ideal for large Bloom filters, an efficient structure for checking membership in a set, or even counting occurrences. Our HMC based counting Bloom filter, useful in a bioinformatics context, achieves a speedup of 13x over traditional FPGA-attached DRAM and 9.31x to 17.6x over multi-core, multi-threaded software on our host system.**

Keywords—FPGA; HMC; K-mer counter; Bloom filter; Bioinformatics accelerator

I. Introduction and Background

A. FPGA Cloud

Public cloud services are a large and growing industry, with the total cloud market estimated at $175 billion for 2015, $204 billion in 2016 and growth continuing at a similar rate [1]. One way that this growth is sustained is the movement of entire classes of computing from on-site or wholly owned servers to cloud based services, and one way to facilitate this shift is more heterogeneous clouds. As part of this trend, many cloud services offer GPU instances to their customers. More recently, the two largest providers of cloud infrastructure services, Amazon and Microsoft (who command a combined 42% of the market [2]), have announced products which will allow customers to use FPGAs as part of their cloud-based application [3][4].

For applications widely deployed on cloud-based FPGAs, the economic advantages of speedups that might otherwise be considered modest in the FPGA community can be significant. For example, during the early evaluation of their Catapult data-center FPGA platform, Microsoft found that for an increase in server total cost of ownership (TCO) of under 30% they could achieve a 95% improvement in per sever Bing search ranking throughput [5]. For a large problem, such as Bing, this could lead to tremendous cost savings, despite the lack of multiple orders of magnitude speedup. As demonstrated in work analyzing ASIC cloud deployments, the non-recurring engineering cost, which is often higher for FPGA-based applications than pure software, can quickly become irrelevant as problems approach the size of Bing or other large cloud-scale applications [6].

One area with many of these applications is genomics. During the 1990s the Human Genome Project (HGP) created the first draft sequence of the full human genome [7]. This reference sequence provided the first picture of the human genome, a string of ~3.2 billion nucleotides. The immature sequencing technology available at the time resulted in the first draft requiring nine years and almost $3 billion to complete. Modern devices, known as Next-Generation Sequencing (NGS) machines, were so successful that they reduced the cost of sequencing at a rate that far outstripped Moore's law [8]. NGS technology has been hugely influential across many fields, including personalized medicine, drug discovery and oncology.

Although processing NGS data seems like an ideal area for cloud computing, there has been a great deal of skepticism due to the specifics of genomic data, including privacy concerns, communication to computation ratio and difficulty parallelizing common bioinformatics algorithms [9]. Given the scale of the problem and the obvious economic incentives, there has been significant research into resolving these problems [10]. We believe that FPGA-based genomics accelerators such as [11] offer solutions to some of these problems. For example, in a field where large quantities of data may only have to be processed a single time, and shipping hard drives is a potentially cost effective means of transmitting data, the increased computation density of servers equipped with FPGAs can greatly reduce network traffic in the cloud provider's datacenter.

B. De Novo Assembly

One way that NGS machines are able to produce so much genomic data so quickly is through the use of short reads. This sequencing technique involves the high-bandwidth reading of huge numbers of small subsequences of the DNA in question, conducting multiple samplings of each region of a DNA strand. This is important, because each individual short read (roughly 100 base pairs) of the DNA is likely to include some errors due to the underlying chemical and electrical processing. Multiple reads of the same region help resolve these errors [12], but also greatly increase the problem size.

This work was supported in part by Micron Technology, Inc.

This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE-0718124.

In reference based assembly, the short reads are individually aligned to a known reference, such as the human genome discussed above. De novo assembly is a more computationally expensive problem [13]. In this case no reference is available, so the short reads must be assembled only in relation to each other. This can occur in many biologically important cases, such as a species which has not yet had its reference constructed (currently the vast majority of species on earth). It can also happen if the source of the reads is not known, for example in a metagenomic study of the life in seawater, dirt or the human gut. In these cases, the reads must be assembled into a coherent sequences in much the same way the first human reference genome was during the Human Genome Project.

C. K-mer Counting

As discussed previously, errors in short reads can be resolved through the use of many overlapping reads, where the number of reads of a given portion of the sequenced genome is called the coverage. Especially for de novo assembly, average coverage of at least 10x is important for high quality results, and due to biases in coverage a significantly higher average may be required to ensure that all regions have sufficient representation [14]. This proliferation of reads greatly increases the processing time required to complete the assembly. In addition, the inclusion of more reads introduces more unique erroneous short sequences, due to read errors. If these reads are all included in the assembly without any attempt at error correction, the accuracy of the final result suffers.

K-mer counting is a data reduction and error removal step included in many sequencing flows to address these issues. The short reads are broken into overlapping subsequences called K-mers, typically of length 20 to 70 bases. Then, the occurrence rates of each K-mer are counted across all reads in the dataset. Since scientists try to ensure sufficient coverage, the reads will generally include perhaps 15 or 30 copies of the same data. A K-mer that only occurs once or twice in the dataset is thus very likely to include an error, and can be discarded [15].

In K-mer counting we break the data into K-mers, count the occurrence of each K-mer in the dataset, and eliminate all K-mers that appear fewer than some threshold count. For other K-mers, only one copy of the K-mer needs to be retained, generally along with the occurrence rate. This not only helps to remove errors, but also decreases the size of the data that must be processed by the assembler by removing redundant sections.

II. FPGA Based K-mer Counting

In this section we will discuss the details of our FPGA based K-mer counter and explain some of our design choices.

A. Structure

An overview of our K-mer counter can be seen in Fig. 1. First, the reads are partitioned into small blocks of K-mers that share a signature, as in [16]. This is a technique used when the available working memory for counting is small, such as on the FPGA. Next, the partitions of K-mers are transferred to the FPGA, one at a time, where they are counted. The incoming K-mers are processed and counts are written to FPGA attached memory. Here, an associative structure is required where the key is the K-mer and the value is a count which is incremented every time that K-mer is seen on the input stream. All of these blocks will be discussed in significantly more detail below.

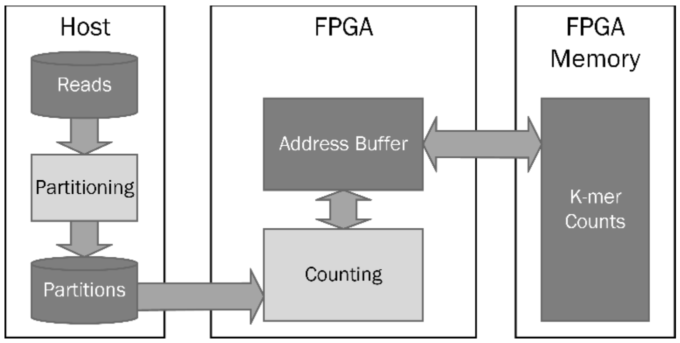

Fig. 1. Generic FPGA-accelerated K-mer counter block diagram.

The above design does not necessarily outperform software. There is some computation involved in unpacking the partitioned K-mers and identifying the correct memory address for each count, but our host system can saturate its DRAM bandwidth when using all CPUs for these tasks. In order for our accelerator to achieve any significant speedup its effective random-access bandwidth must be greater than the host's. Additionally, the representation must be compact since FPGA-attached memories are typically much smaller than what is available to the host. The following sections present tools for addressing these issues.

B. Bloom Filters

A basic Bloom filter is a space-efficient mechanism for answering the question, “Have I seen this data item before?” [17]. A naïve method for handling this question is to have a bit in memory for each possible data item, then simply set that bit when the corresponding value is seen. However, for many questions the number of possible items is astronomical. Instead, we could hash each data item for lookup in a smaller set of bits. However, collisions in the hash function become a problem, and storing the data items in this hash table to resolve collisions may also require a huge memory.

Bloom filters maintain a small number of bits in a hash table, and do not keep the original items, so collision detection is impossible. Instead, the Bloom filter applies several independent hash functions to the incoming data item, and uses the result of each of those as a lookup in the bit table. Thus, if there are 3 hash functions, the item will map to 3 separate bits in the table. If any of those bits are a 0, we are guaranteed to have never seen that value before, and a result of “not seen before” is returned; those 3 bits are also then set to 1. If during a lookup we find all the bits are true, we return the result of “has been seen before”. Note that this does mean there can be false positives: although a single different item is very unlikely to have set the same three bits to 1 previously since the hash functions are independent, it is possible that 3 previous items have each set one of the current value's bits to true. Because of this the number of hash functions, and the size of the Bloom filter, must be chosen to keep the false positive rate sufficiently low. The mathematical theory behind Bloom filters allows each parameter to be chosen correctly for a given application.

A counting Bloom filter, a simple extension of what we have just described, answers the question "How many times have I seen this data item before?" [18]. Instead of a single bit per location, a counting Bloom filter stores a small counter value, for example 4 bits if we wish to handle counts of 0..15. When we do the lookup, we increment all counter values associated with the hashes (saturating at the maximum value), and return the smallest value found. Note that it might be more correct to only increment the counters containing the smallest value, but this introduces a sequential dependency that can limit performance; in practice an error due to this over-incrementing is equivalent to the false positives found in standard Bloom filters, and is generally considered insignificant. As with a standard Bloom filter, the number returned will never be smaller than the true result, though the value returned may be too high, which is the equivalent of a false positive.

Since in subsequent sections we will discuss the implementation of a counting Bloom filter for a K-mer counter, it is important to consider the steps required to do a lookup in the data structure: (1) Hash the incoming data with N independent hashes; (2) Read the current value at those N locations; (3) Increment the value at each location, subject to saturation. These steps are illustrated in Fig. 2. Note that if we allow concurrent accesses to improve throughput, this read-modify-write operation must essentially be made atomic. If two lookups to the same counter are in flight simultaneously, the resulting count may be corrupted. Correctly operating Bloom filters sometimes produce results that are too high, but cannot report a number too low (other than from saturation).

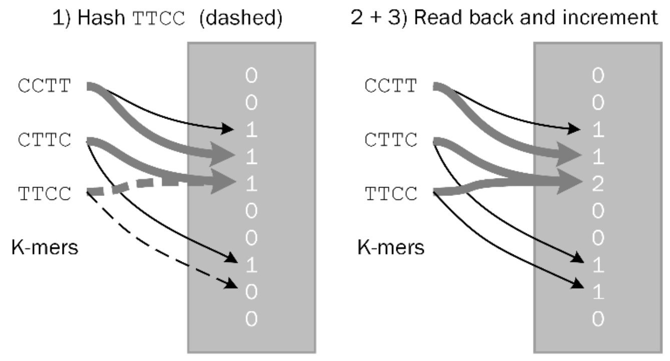

Fig. 2. Counting Bloom filter insert operation for a 2-hash filter, where CCTT and CTTC have already been seen. TTCC returns 1 total occurrence, including the current one.

Although counting Bloom filters perform well for our purposes, they have since been superseded in many uses by more exotic data structures with similar properties. [19] compares a number of Bloom filters in its Table 3. We believe that our work could be easily extended to one of these structures, resulting in noticeable performance improvement. However, this is outside the scope of what we address here and the counting Bloom filter is sufficient to evaluate the HMC as a tool for FPGA genomics.

1) *Bloom Filters on FPGAs*

FPGA-based Bloom filters have been in use for networking applications since at least the early 2000s [20][21]. More recently other uses have emerged related to caching or otherwise filtering non-network traffic data [22][23]. There are also more exotic use cases, such as extremely low power systems [24]. However, all of these applications have one thing in common: The Bloom filters are typically small (100s of kB to MBs), and are stored on chip. In traditional DRAM, off chip Bloom filters are either not viable (too slow) or unnecessary (DRAMs are relatively large when considering typical non-HPC FPGA applications).

C. *The Hybrid Memory Cube*

To support the bandwidth requirements of genomics applications, such as K-mer counting, we consider the Hybrid Memory Cube (HMC) in this section. The HMC is made up of many modified DRAM memory dies stacked on top of a logic die using through-silicon vias [25]. The package provides multiple fast serial links to communicate with the memory, instead of the large parallel connections provided by traditional DRAM. Internally, the HMC is partitioned as an array of vertical slices, called vaults, as seen in Fig. 3. Each vault contains a DRAM memory controller called a vault controller (VC) on the bottom of the die stack. Each VC is connected to a column of memory banks. The VCs themselves are connected using a crossbar switch. On the other side of the switch is a collection of fast serial interfaces, called links, which can connect to the external processor (CPU, FPGA, etc.) or another HMC.

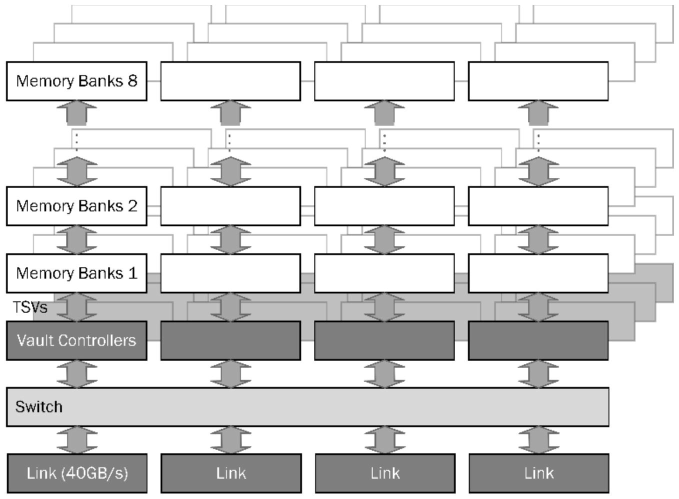

Fig. 3. HMC architecture block diagram.

The HMC gets its performance advantage over traditional DRAM by having many banks that can be accessed in parallel by the VCs. The controllers themselves are designed to handle a large number of transactions in flight, servicing them quickly as banks become available. This architecture allows for both very high bandwidth and improved random-access performance when compared to traditional DRAM.

There are some potential downsides to the architecture. First, if the HMC receives too many operations to the same small set of banks, it can slow down significantly since little parallelism is available to be exploited. Although internal buffering and reordering can mitigate this to some degree, a sufficiently pathological access pattern can greatly degrade HMC performance. The second issue is the reordering itself. Because of its more complex architecture, the HMC reorders operations at a number of points, including between the VCs and within each VC. The only ordering that is guaranteed to be maintained is that among operations to an individual bank. Finally, the HMC crossbar privileges a set of VCs for each link. Accessing memory only through the privileged links will result in double the performance of accessing non-privileged memory across all links simultaneously. However, this has the obvious downside of dividing the HMC into four separate address spaces, instead of appearing as a shared memory to all linked devices.

One use the HMC makes of its logic layer is supporting a variety of atomic operations. Although the 1.1 specification [26] provides the masked write required to treat each counter as independent of other counters for the read-modify-write increment operation, a single atomic operation to perform this update could reduce FPGA logic and increase performance by almost 50%. The latest HMC 2.2 specification provides additional atomic operations that should be considered when designing an FPGA-based system using the HMC.

D. *Bloom Filters for K-mer Counting*

Now that we have considered the major background concepts (K-mer counting, Bloom filters, HMCs), a logical question is why would someone want to do an FPGA/HMC based implementation of a Bloom filter for K-mer counting?

Bloom filters are a good fit for K-mer counting because of their space efficiency, independent of K. Alternatives, such as trees and hash tables can be prohibitive in size, requiring at least an order of magnitude more memory and growing with K. Also, Bloom filter false-positives, resulting in a potential over-counting of occurrence rates in a counting Bloom filter, is not overly problematic in this case. In K-mer counting we discard low occurrence K-mers because we know we do not need their data; however, if such a K-mer is retained, it merely represents a missed speedup since the data is larger than otherwise, not a degradation of the final results (see [27] Figure 4). However, if we throw away good data (false negatives) we are potentially damaging the final results.

Bloom filters themselves are well suited to FPGA-based computation and for implementation using the HMC. Bloom filters generate large volumes of small, random accesses, which fit the HMC model well. Also, this system generates many memory operations that are almost always independent, allowing for a highly parallel implementation.

E. *Architecture*

Our system is built on a Micron SB-800 board featuring 4 Altera Stratix V FPGAs, and a 4GB HMC with a single high-speed serial link to each FPGA. In order to achieve the best performance, no link can accesses memory outside of its own four HMC vaults, so we decided to have each of the four FPGAs work independently on their own small Bloom filter. This approach fits well with K-mer counting where the large set of K-mers is already typically broken down into smaller bins on disk so that the counting structure can fit into system memory. Because of this approach, there is very little downside to using multiple smaller Bloom filters in parallel instead of one larger one across all of the FPGAs. The overall system architecture is shown in Fig. 4.

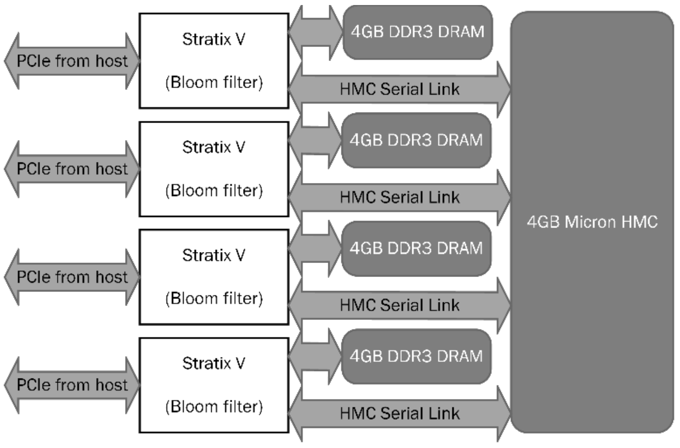

Fig. 4. HMC Bloom filter system architecture.

The Bloom filter on each FPGA is designed to maximize memory utilization, since the entire system should ultimately be memory performance limited. K-mers arrive at the Bloom filter uncompressed, potentially having first traveled through a decompression unit, as seen at the top of Fig. 5. Next the K-mers are hashed, with the K-mer retained in a bypass FIFO for output.

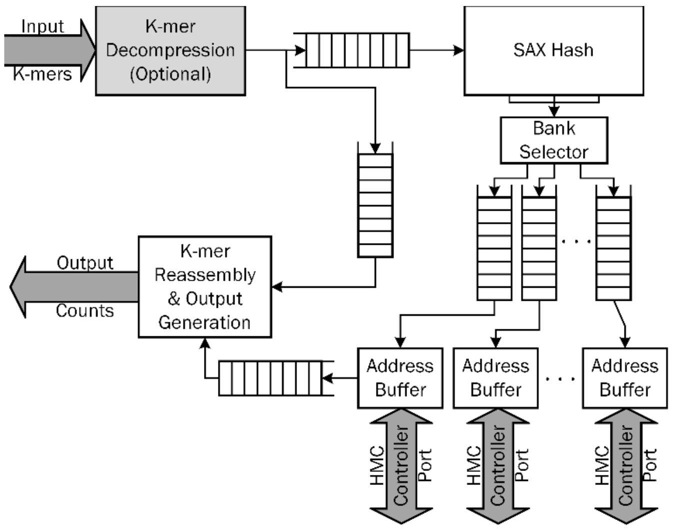

Fig. 5. FPGA-based Bloom filter block diagram.

After the K-mer has gone to two or more independent hash functions, the resulting hashed addresses are tagged to identify their origin K-mer. Tags are assigned sequentially, and are used to reintegrate data that flows down the independent datapaths in this design.

The Bloom filter must now look up each counter, increment it and find the minimum value. Doing a single lookup at a time would be simple, but serializing HMC access is totally infeasible due to latency. To saturate the HMC multiple operations must be in flight simultaneously, while maintaining atomic read-modify-write operation. This is particularly difficult because although the HMC will never reorder operations that access the same memory bank, the HMC memory controller located on the FPGA can perform just such a reordering. This controller, through which user logic communicates with the HMC, has 4 memory ports and may reorder operations arriving on different ports, even if they address the same bank.

To support efficient pipelining in the face of reordering, we assign banks to specific HMC controller ports, so that all accesses to a given bank use the same port, preventing accesses from being reordered. We also maintain a buffer of addresses in-flight, and will stall a subsequent access that targets the same Bloom filter counter to maintain atomicity. Because only data from the given counter is used, and writes are masked, reordering of operations that access different segments of the same memory word are not relevant.

When all the reads for a K-mer complete, the filter finds the minimum count, and issue memory writes to update the counters that have not already saturated. Subsequently, these addresses are removed from the buffer, so that stalled accesses can proceed. With full output enabled, the filter outputs the count along with the actual K-mer stored in the bypass FIFO.

1) Hash Functions

Selecting a good hash function is very important for Bloom filter performance, and a great deal of work has been done on hash functions appropriate for ASIC and FPGA use [28][29]. We selected a two round shift-add-xor (SAX) hash function [30]. This class of hash functions behaves very well for repetitive strings (like genomic sequences), avoids the use of multiplication and easily allows for very deep pipelining making it ideal for our purposes. The SAX function accepts 8 bits at a time. For length 63 K-mers, and 2-bit bases, we apply a 16 stage pipeline to compute the hash. Note that since our application is not latency sensitive, a deep hashing pipeline is both acceptable and necessary. A new K-mer can arrive each clock and requires multiple hashes. The hash pipeline is shown in Fig. 6. Note that the pipeline actually employs twice as many stages as discussed above, where the second hashing round improves result quality.

The SAX hash produces a single 64-bit hash value, which could index 2^{64} different counters. However, since we only require 2^{30} different counters in our Bloom filter, this single 64-bit hash is broken into 2 32-bit hashes. Note that for designs that require more hash bits we replicate the SAX pipeline with a different seed. For example, using 4 hash functions for our Bloom filter, as in the evaluation below, required 2 copies of the entire SAX pipeline.

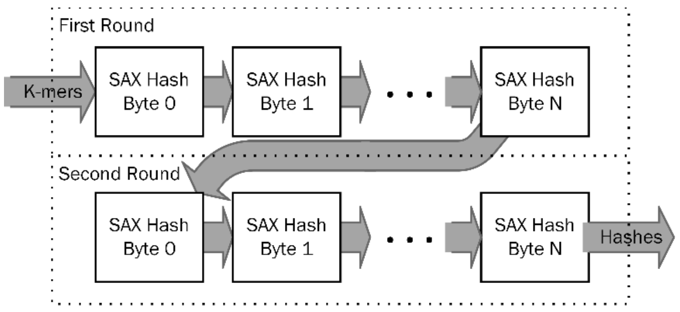

Fig. 6. SAX hash pipeline.

To demonstrate that SAX is an effective hash function for Bloom filters we compared the False Positive Probability (FPP) of a Bloom filter using SAX and SHA-256 to the expected FPP calculated using the approximation given in [31].

The results of this comparison can be seen in Fig. 7. On the horizontal axis is the input size. This data was generated by averaging the FPP from many runs with a relatively small Bloom filter, so as the input size approaches 8,000 the filter becomes very full and the FPP rises significantly. SAX hash performs almost as well as the much more expensive SHA-256 in all situations, and both track the expected FPP very closely.

The difference between SAX and SHA-256 is at most 1/100 of 1%. However, note that this is for a Bloom filter with a large load factor (the fullness of the filter), such that the FPP is much too high (around 2% for N=5 and 4% for N=3, where N is the number of hash functions). In the normal operating range where FPP is safely under 1%, SHA-256 and SAX perform extremely similarly. Given that SHA-256 is a very well behaved cryptographic hash function, this suggests that SAX is entirely sufficient for our Bloom filter.

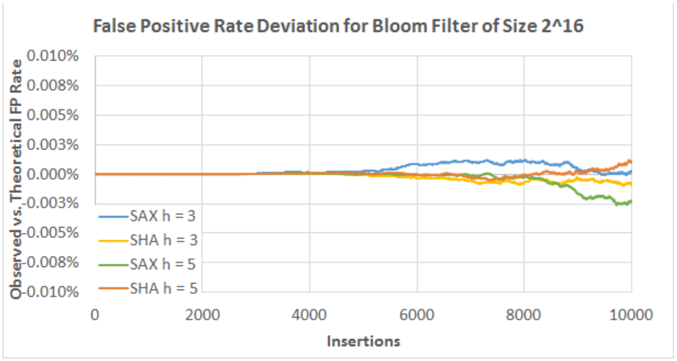

Fig. 7. Bloom filter hash function false positive probability (FPP) comparison. Positive differences represent higher than theoretical FPP.

2) Address Buffer

After being hashed, counter addresses from the K-mer enter a buffer (Fig. 8). This module serves two purposes. First, it remembers both the HMC tag (used to identify returning data from the HMC) and a K-mer identifying tag for each address for output generation. Second, the buffer provides collision detection. In particular, any K-mers that hash to one or more of the same counters will always be sent to the same address buffer because the bank selector distributes the hashes based on their hashed address. Therefore, collision detection at the address buffer is sufficient to avoid any races. In the case of a collision all addresses coming to that buffer are stalled until the aliasing K-mer's update is completed, at which point the entry is cleared and the new address can proceed. Stalling an entire set of banks, which can eventually put backpressure on incoming K-mers, is not the most efficient solution; however it allows for much simpler hardware, and collisions are extremely rare. The total impact on performance is much less than 0.1% in our tests.

The buffer also handles the reordering of read results returning from the HMC. This would not be necessary if there were a single buffer per bank, but there are many banks (for the HMC used here there are 4 vaults per link and 8 banks per vault for a total of 32 banks per link) and each buffer would be inefficiently small if there were truly one per bank. We instead have per-HMC controller port buffers, with reordering resolution logic.

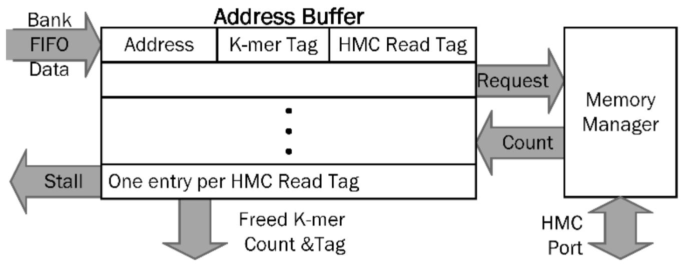

Fig. 8. Address buffer and memory communication.

When read data returns, the counter must be incremented, and that value must be written back using a bit-mask to avoid impacting any other counters that may have been updated after the read. This allows the buffer to stall only when there is an exact counter address collision, not every time two addresses share the same word in memory. Note that counts saturate at the full counter value. For example, once a count reaches 15 when using 4-bit counters that counter will no longer be written back. This saves a minor amount of memory bandwidth. Unsaturated counters are always incremented and written back as described.

After any required write-backs are issued by the memory manager, the buffer entry is invalidated and dependent stalls are released. The count returned by the memory is sent to the output module along with the K-mer tag. The output module must collect the value of each counter that a K-mer hashed to and return the minimum of those values as the count for that K-mer.

III. Performance Evaluation

A. Methodology

1) K-mer Counting

The problem of K-mer counting is an important filtering step for many de novo sequence assembly algorithms as well as other areas of bioinformatics [32]. K-mer counting is also an appropriate evaluation for our filter because Bloom filters are well established in the area [27]. For K-mer counting, in the case of a false positive, a K-mer that should have been filtered will be processed, resulting in slightly slower assembly but not an incorrect result. Additionally, the saturation property of the counting Bloom filter is not a problem when counting K-mers because the greatest number of occurrences that are typically filtered are easily stored in 3 or 4 bits and no distinction is made between unfiltered K-mers based on their abundance.

2) Hiding Disk System Limitations

One challenge for K-mer counting on any platform is that there are a large number of K-mers in any given genome. This results in tables that are typically very large, on the order of 10s of GB [16]. These tables are too large to fit in memory for some desktop systems, let alone in most FPGA attached memory. This problem has been addressed by storing K-mers in bins on disk, based on some signature substring to ensure that the same K-mer always goes in the same bin [33][34].

3) Board Configuration and Comparison Targets

As discussed above, this research was performed on a Micron SB-800 board, an HMC board that plugs into a PCIe slot and uses 4 Altera FPGAs to support all of the external links of a single HMC device (smaller more recent boards, such as the AC-510 and AC-520, couple an HMC with a single Xilinx or Altera FPGA across multiple links). Thus, our system is composed of 1 HMC, 4 Altera FPGAs with 4 DDR3 DRAM memories, one per FPGA, and 1 host computer with 6 cores. In the results that follow, we will compare HMC, DRAM, and CPU implementations in various configurations.

The obvious question that arises is what resources form a "fair" comparison? If we view this paper as an evaluation of computation resources, then perhaps 1 FPGA vs. 1 CPU is the right comparison. Alternatively, if this is a paper on memory systems, then perhaps 1 HMC (requiring 4 FPGAs), 1 DRAM, and 1 Host Memory subsystem (which means all 6 cores) is the better target. Given that our Bloom filter logic consumes only roughly 15% of the logic resources on each FPGA, our entire design could fit on a single FPGA connected to all HMC ports on a board better suited to it. Our conclusion is that no one comparison is the "right" one. Thus, we will present the results of each of the different interesting configurations.

B. Results

First we will compare the HMC to DRAM, followed by our reference implementation of the same algorithm running on the host, and finally a fully optimized software K-mer counter.

1) HMC vs. DRAM Performance

The HMC's design gives it a significant advantage over traditional FPGA-attached DRAM for random access heavy workloads, such as a Bloom filter. This results in a significant speedup compared to the same design using DRAM on the FPGA, as seen in TABLE I. All tests were run on Illumina short read data from a Yoruba man and available as experiment SRX016231 [35]. This is a standard dataset, containing 93 Gbases of read data collected across 37 sequencer runs; it is frequently used for genomics performance evaluations.

The HMC-based Bloom filter achieves approximately a 13x speedup as compared to the DRAM based system. This might be surprising because, on our board, the aggregate HMC bandwidth is only about 4x greater than the DRAM bandwidth. However, DRAM is optimized for large block transfers, and the small Bloom filter accesses required fit this model poorly. In comparison, the HMC is able to support small accesses much more efficiently, outperforming its relative bandwidth advantage by 3.3x.

The value of K is irrelevant in these tests because the FPGA-based system simply utilizes more resources to calculate hashes in a longer pipeline. Note that this does increase latency slightly, but the impact is minor and is not important for this application. However, if host to FPGA bandwidth was a bottleneck then the K-mer length could become important.

Although the 4 FPGAs share the HMC, each only uses the memory in the 4 vaults local to its link. This means that there is almost no performance overhead to adding more FPGAs (for k = 47 four FPGAs are 3.94x faster than a single FPGA).

2) HMC vs. Host Performance

Our computation system actually has three memory systems that could be used for K-mer counting: HMC, FPGA-attached DRAM, and the host's memory system itself. Given that CPU

DDR3 memory controllers and architecture (wider busses, faster DRAM timings and dual-channel support) typically provide significantly more random-access bandwidth than FPGA-attached DRAM one might expect the host system to outperform the FPGA DRAM solution. In TABLE I. we compare the single FPGA implementations (DRAM and 1-FPGA HMC) to software based implementations, including a single-threaded implementation (1 CPU), and a fully parallel version using multiple threads on all the CPUs in the system (6 CPU). The number of hashes is not particularly relevant for this comparison because all three solutions are impacted identically.

TABLE I. HMC, DRAM AND HOST MEMORY BLOOM FILTER AND SYSTEM PERFORMANCE WITH 4 HASHES FOR VARIOUS K-MER LENGTHS (M K-MERS / SEC)

	K = 31	**K = 47**	**K = 63**
HMC 1 FPGA	55.8	57.4	57.5
HMC 4 FPGAs	221	226	227
DRAM 1 FPGA	4.44	4.49	4.45
HMC vs. DRAM 1 FPGA Speedup	12.6x	12.8x	12.9x
1 CPU 1 Thread	6.76	4.27	3.28
6 CPU Multiple Threads	41.2	28.8	22.8
1 FPGA HMC vs. 1 CPU Speedup	8.25x	13.4x	17.5x
1 FPGA HMC vs. 6 CPU Speedup	1.35x	1.99x	2.52x
4 FPGA HMC vs. 6 CPU Speedup	5.36x	7.85x	9.96x

The first thing to notice is that, unlike the FPGA-based versions, the CPU implementations are sensitive to K, the K-mer length. This makes sense since, while the FPGA can handle hashing and memory operations in parallel, the CPU must spend extra cycles on these tasks. Our CPU implementation is optimized, but doesn't make explicit use of SSE instructions or other potentially beneficial techniques.

The single-threaded software performance is roughly comparable to the FPGA DRAM performance. Although the host DRAM has better support for random-access operations, the limited parallelism of the single threaded implementation dominates. However, the multi-threaded software version is significantly faster than the DRAM FPGA version.

Finally, the 1 FPGA HMC implementation is approximately twice as fast as the host CPU running a fully parallelized version of the software. While both the FPGA and the multi-threaded software implementations can overlap computation with memory accesses (the FPGA through spatial parallelism, the CPU via multi-threading), the random access support of the HMC results in a significant speedup. In fact, if we use the entire HMC capacity, requiring us to use all 4 of the FPGAs on the board, we achieve a 5.36x to 9.96x speedup compared to the fully parallelized host version, with speedup depending on K-mer length.

3) *HMC vs. Software K-mer Counter Performance*

In the previous section we compared Bloom filter based K-mer counters in both FPGA and CPU implementations. However, there are alternative strategies for performing K-mer counting. To give an idea of the current state of the art, we've compared our FPGA/HMC K-mer counter to KMC 2 [16] running on an Intel Core i7-5930K with 6 physical CPU cores and 32 GB of system memory. Note that this comparison is only K-mer counting, so time spent on binning and other processing is not included. In a Bloom filter the number of hashes impacts the false positive rate. Through tuning, we found that we could achieve the same quality as KMC 2 using a 2 hash Bloom filter and smaller partitions. Our comparison uses those settings.

TABLE II. HMC (WITH 2 HASHES) AND KMC 2 COMPARISON (M K-MERS / SEC)

	K = 31	**K = 47**	**K = 63**
HMC 1 FPGA	124	124	123
HMC 4 FPGAs	484	485	484
KMC 2 1 CPU (1 thread)	19.7	8.53	6.93
KMC 2 6 CPU (12 threads)	52.0	41.1	27.5
1 FPGA HMC vs. 1 CPU KMC 2 Speedup	6.29x	14.5x	17.7x
4 FPGA HMC vs 6 CPU KMC 2 Speedup	9.31x	11.8x	17.6x

As seen in TABLE II. the FPGA-based counter compares well to a highly optimized software based solution. Like the software version of our Bloom filter, KMC 2 also slows down as the K-mer length increases. Overall, our 1 FPGA system provides a 2.4x – 4.4x speedup when compared to a fully parallelized state-of-the-art K-mer counter, and a 9.31x – 17.6x speedup when using the entire HMC with all 4 FPGAs.

IV. CONCLUSIONS

Cloud-based FPGA instances have finally arrived. This creates exciting new opportunities in many areas, including bioinformatics. Genomics applications have not had as easy of a transition to the cloud as many other fields, but FPGAs can help overcome the hurdles they face. New memory technologies also offer new opportunities for FPGA-based accelerators, but they generally come with different strengths and weaknesses as compared to traditional SRAM and DRAM devices.

In this paper we developed a Hybrid Memory Cube based K-mer counter, which demonstrates very high performance. In many ways HMCs can be considered as a DRAM replacement, offering higher bandwidth and comparable capacity. HMCs also offer better random-access performance when compared to DRAM, especially for smaller transfer sizes. When we exploit the full bandwidth of an HMC, we can handle 484 M K-mers/second. In comparison a single DRAM based system can achieve only 9.3 M K-mers/second, and even using all four DRAMs the system would still be 13x slower. This 13x speedup represents an approximately 2x advantage in bandwidth, and a

3.3x advantage in efficiency for small random accesses, for the HMC compared to DRAM.

We also compared our FPGA & HMC implementation to a fully parallelized software version, which exploits multiple threads on each of 6 CPU cores in the host. The HMC based system was able to achieve a 5.36x – 9.96x speedup compared to the software version. When we compare our HMC implementation to a state-of-the-art, non-Bloom filter K-mer counter, our system is 9.31x – 17.6x faster.

Looking at these speedup numbers and the DRAM performance, we see that an FPGA K-mer counter using DRAM would perform roughly the same or slightly worse than software. This suggests that memory style is very important and we believe the same result would appear for many other genomics problems, including the remainder of the de novo assembly pipeline. Critically then, FPGAs in the cloud can be valuable for genomics, but being paired with nontraditional memories such as the HMC allow accelerators in a space that would not otherwise be possible.

References

[1] Gartner, Inc., "Forecast: Public Cloud Services, Worldwide, 2013-2019, 4Q15 Update," 2015. [Online]. Available: http://www.gartner.com/newsroom/id/3188817. [Accessed: 17-Jan-2016].

[2] M. Day, "How Seattle became 'Cloud City': Amazon and Microsoft are leading a tech revolution," *The Seattle Times*, p. A1, Dec. 9th, 2016.

[3] A. M. Caulfield *et al.*, "A cloud-scale acceleration architecture," in *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on*, 2016, pp. 1–13.

[4] Amazon Web Services, Inc., "Amazon EC2 F1 Instances (Preview)," 2016. [Online]. Available: https://aws.amazon.com/ec2/instance-types/f1/. [Accessed: 17-Jan-2017].

[5] A. Putnam *et al.*, "A reconfigurable fabric for accelerating large-scale datacenter services," in *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, 2014, pp. 13–24.

[6] I. Magaki, M. Khazraee, L. V. Gutierrez, and M. B. Taylor, "ASIC clouds: specializing the datacenter," in *Proceedings of the 43rd International Symposium on Computer Architecture*, 2016, pp. 178–190.

[7] J. C. Venter *et al.*, "The sequence of the human genome," *Science*, vol. 291, no. 5507, pp. 1304–1351, 2001.

[8] K. Wetterstrand, "DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP)," 2014. [Online]. Available: http://www.genome.gov/sequencingcosts. [Accessed: 13-Oct-2014].

[9] M. C. Schatz, B. Langmead, and S. L. Salzberg, "Cloud computing and the DNA data race," *Nature Biotechnology*, vol. 28, no. 7, p. 691, 2010.

[10] J. G. Reid *et al.*, "Launching genomics into the cloud: deployment of Mercury, a next generation sequence analysis pipeline," *BMC Bioinformatics*, vol. 15, no. 1, p. 1, 2014.

[11] C. B. Olson *et al.*, "Hardware Acceleration of Short Read Mapping," in *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, 2012, pp. 161–168.

[12] M. A. Quail *et al.*, "A tale of three next generation sequencing platforms: comparison of Ion Torrent, Pacific Biosciences and Illumina MiSeq sequencers," *BMC Genomics*, vol. 13, no. 1, p. 1, 2012.

[13] D. R. Zerbino and E. Birney, "Velvet: algorithms for de novo short read assembly using de Bruijn graphs," *Genome Research*, vol. 18, no. 5, pp. 821–829, 2008.

[14] N. S. Movahedi, E. Forouzmand, and H. Chitsaz, "De novo co-assembly of bacterial genomes from multiple single cells," in *Bioinformatics and Biomedicine (BIBM), 2012 IEEE International Conference on*, 2012, pp. 1–5.

[15] R. Chikhi and P. Medvedev, "Informed and automated k-mer size selection for genome assembly," *Bioinformatics*, pp. 31–37, 2013.

[16] S. Deorowicz, M. Kokot, S. Grabowski, and A. Debudaj-Grabysz, "KMC 2: Fast and resource-frugal k-mer counting," *Bioinformatics*, vol. 31, no. 10, pp. 1569–1576, 2015.

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

[18] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM Transactions on Networking (TON)*, vol. 8, no. 3, pp. 281–293, 2000.

[19] B. Fan, D. G. Andersen, M. Kaminsky, and M. D. Mitzenmacher, "Cuckoo filter: Practically better than bloom," in *Proceedings of the 10th ACM International Conference on Emerging Networking Experiments and Technologies*, 2014, pp. 75–88.

[20] S. Dharmapurikar, P. Krishnamurthy, T. Sproull, and J. Lockwood, "Deep packet inspection using parallel bloom filters," in *High Performance Interconnects, 11th Symposium on*, 2003, pp. 44–51.

[21] J. W. Lockwood, J. Moscola, M. Kulig, D. Reddick, and T. Brooks, "Internet worm and virus protection in dynamically reconfigurable hardware," in *Proceedings of the Military and Aerospace Programmable Logic Device Conference*, 2003.

[22] A. Becher, D. Ziener, K. Meyer-Wegener, and J. Teich, "A co-design approach for accelerated SQL query processing via FPGA-based data filtering," in *Field Programmable Technology (FPT), 2015 International Conference on*, 2015, pp. 192–195.

[23] R. Dobai and J. Korenek, "Evolution of Non-Cryptographic Hash Function Pairs for FPGA-Based Network Applications," in *Computational Intelligence, 2015 IEEE Symposium Series on*, 2015, pp. 1214–1219.

[24] 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*, 2009, pp. 371–376.

[25] J. T. Pawlowski, "Hybrid memory cube (HMC)," in *Hot Chips*, 2011, vol. 23.

[26] Hybrid Memory Cube Consortium, "Hybrid Memory Cube Specification 1.1," 2014.

[27] P. Melsted and J. K. Pritchard, "Efficient counting of k-mers in DNA sequences using a bloom filter.," *BMC Bioinformatics*, vol. 12, p. 333, 2011.

[28] F. Yamaguchi and H. Nishi, "Hardware-based hash functions for network applications," in *Networks (ICON), 2013 19th IEEE International Conference on*, 2013, pp. 1–6.

[29] M. A. Gosselin-Lavigne, H. Gonzalez, N. Stakhanova, and A. A. Ghorbani, "A Performance Evaluation of Hash Functions for IP Reputation Lookup Using Bloom Filters," in *Availability, Reliability and Security (ARES), 2015 10th International Conference on*, 2015, pp. 516–521.

[30] M. Ramakrishna and J. Zobel, "Performance in Practice of String Hashing Functions.," in *DASFAA*, 1997, pp. 215–224.

[31] M. Mitzenmacher and E. Upfal, *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*. Cambridge University Press, 2005.

[32] R. Li *et al.*, "De novo assembly of human genomes with massively parallel short read sequencing.," *Genome Res.*, vol. 20, no. 2, pp. 265–72, 2010.

[33] S. Deorowicz, A. Debudaj-Grabysz, and S. Grabowski, "Disk-based k-mer counting on a PC.," *BMC Bioinformatics*, vol. 14, p. 160, 2013.

[34] G. Rizk, D. Lavenier, and R. Chikhi, "DSK: k-mer counting with very low memory usage," *Bioinformatics*, pp. 652–3, 2013.

[35] D. R. Bentley *et al.*, "Accurate whole human genome sequencing using reversible terminator chemistry," *Nature*, vol. 456, no. 7218, pp. 53–59, 2008.