GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment

Anirban Nag
anirban@cs.utah.edu
University of Utah
Salt Lake City, Utah

C. N. Ramachandra
ramgowda@cs.utah.edu
University of Utah
Salt Lake City, Utah

Rajeev Balasubramonian
rajeev@cs.utah.edu
University of Utah
Salt Lake City, Utah

Ryan Stutsman
stutsman@cs.utah.edu
University of Utah
Salt Lake City, Utah

Edouard Giacomin
edouard.giacomin@utah.edu
University of Utah
Salt Lake City, Utah

Hari Kambalasubramanyam
hari.kambalasubramanyam@utah.edu
University of Utah
Salt Lake City, Utah

Pierre-Emmanuel Gaillardon
pierre-emmanuel.gaillardon@utah.edu
University of Utah
Salt Lake City, Utah

ABSTRACT

Precision Medicine will rely on frequent genomic analysis, especially for patients undergoing cancer treatments or suffering from rare diseases. Sequence alignment is invoked in multiple stages of the genomic analysis pipeline. Recent projects have introduced accelerators, GenAx and Darwin, for 2nd and 3rd generation sequencers respectively. In this work, we improve upon the GenAx design by increasing its parallelism and reducing its memory bandwidth demands. This is achieved with a combination of hardware and software innovations. We first integrate in-cache operators from prior work into the GenAx memory hierarchy; we then augment the in-cache peripheral circuit to support additional new operators. We then re-structure the sequence alignment algorithm to (i) leverage the many in-cache operators, (ii) exploit the common case in genomic datasets, (iii) use Bloom Filters to reduce futile accesses, and (iv) maximize data reuse within a re-organized memory hierarchy. While the baseline GenAx accelerator processes a batch of reads in 194 seconds while nearly saturating the 153.6 GB/s memory bandwidth, the proposed GenCache architecture processes the same batch of reads in 37 seconds at an improved energy efficiency of 8.6×, while demanding 20 GB/s average memory bandwidth. Our hardware and software techniques thus interact synergistically to target both memory and compute bottlenecks, while not affecting the outputs of the application. We show that the basic principles in GenCache can also be exploited by 3rd generation sequence aligners.

CCS CONCEPTS

• **Applied computing** → *Computational genomics*; • **Computer systems organization** → *Special purpose systems.*

KEYWORDS

Genomics, Hardware Acceleration, Sequence Alignment, In-Cache Operators, CISC Instructions, Cache Partitioning

ACM Reference Format:
Anirban Nag, C. N. Ramachandra, Rajeev Balasubramonian, Ryan Stutsman, Edouard Giacomin, Hari Kambalasubramanyam, and Pierre-Emmanuel Gaillardon. 2019. GenCache: Leveraging In-Cache Operators for Efficient Sequence Alignment . In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358308

1 INTRODUCTION

Precision Medicine promises to revolutionize healthcare in the near future, relying heavily on genomic information that has the key to both diagnosis and treatment [15, 21, 24, 25]. For example, genome analysis is useful in understanding cancer-causing mutations and crafting an optimal drug cocktail that targets those cancers with minimal side effects [10]. Thanks to the reducing cost of genome sequencing [36], we will have large enough genomic databases to enable high-confidence hypothesis testing and significant discoveries. Not only will every person and every newborn be sequenced, a single person will be sequenced multiple times to monitor natural progression of genetic health, growth of a tumor, and structural variations in different cells of the body. However, this introduces a significant computation and storage challenge. Infrastructures in hospitals and in the cloud will be fed with several genomic analysis queries per patient, for thousands of patients per day. Most analysis software packages consume several CPU hours to perform each of these computations [9].

MICRO-52, October 12–16, 2019, Columbus, OH, USA

ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358308

Genomic accelerators have the potential to dramatically bring down the execution time and energy for these tasks. In the past year, multiple such accelerators have emerged [9, 35, 37], and led to orders of magnitude improvement. In this work, we extend this family of accelerators by primarily augmenting the memory hierarchy and tailoring the algorithm for the new hardware. The basic ideas have the potential for impact beyond the domain of genomic analysis.

A generic sequence alignment pipeline involves the following steps: a genomic segment is partitioned into small sub-segments; these sub-segments index into hash tables to identify potential matching locations in a reference genome; filtration heuristics are used to narrow the set of candidate locations; each location is then assigned an alignment score with a dynamic programming step. The first three steps are critical because they require many parallel operations and memory fetches, while also determining the pressure on the compute-intensive dynamic programming step. Prior accelerators like GenAx [9] and Darwin [35] use tiling and dedicate a large fraction of chip area for scratchpad SRAM that reduce memory bandwidth overheads imposed by the first three steps.

The proposed *GenCache* architecture uses a combination of hardware and software innovations. We first employ the concept of in-cache operators to provide computational capabilities within the SRAM arrays; we introduce new operators that can be leveraged to perform many parallel filtering operations. We then design a new multi-phase algorithm that reduces redundant work, uses appropriate in-cache operators in each phase, and manages scratchpad allocation and tiling to maximize reuse and parallelism. This new algorithm is based on our characterization of the common case when aligning genomic reads and seeds. We also observe that by promoting higher parallelism and reuse for some data structures, the cacheability of the largest data structure is negatively impacted; we alleviate that effect with a Bloom Filter that reduces cache pollution and bandwidth overheads.

Algorithm modifications alone, or in-cache operators alone yield relatively low speedups. The combination of the two results in much more than additive speedup because the new algorithm shifts the bottleneck to compute, which is then accelerated by the new in-cache operators in GenCache.

The above hardware and software techniques help reduce data movement by more than an order of magnitude for a 2nd generation sequencing pipeline. Execution time is reduced by 5.3×, energy efficiency is improved by 8.6×, and the outputs of the genomic algorithm are unchanged. The new GenCache architecture imposes an area overhead of 16%, relative to GenAx. We also show that in-cache operators can be leveraged by 3rd generation pipelines, yielding a 1.8× improvement in 3rd generation filtering.

2 BACKGROUND

2.1 Second and Third Generation Reads

Sequencing cost has dropped significantly over the past decade, due to 2nd generation devices like Illumina HiSeq X and NovaSeq 6000. These devices typically produce small genomic segments called *reads* of size 100-200 bases at a throughput of 2300 Mbases/min. The machines introduce machine artifacts (errors) when generating reads, which range from 2-5%. Additionally, a human genome differs from other human genomes by about 0.1%. Since this difference is small, a reference-based reconstruction of the genome using these short reads is possible. In such a reconstruction, the reads are uniquely aligned against a reference genome by performing inexact matching which accommodates for errors like mismatches and insertions/deletions (indels).

Once the reads are aligned, further statistical downstream analysis is performed to correct machine artifacts. This requires that the machine produce the same base multiple times for sufficient statistical guarantees. Thus, the machine generates multiple reads such that each base in the genome has a redundancy of $30\times - 50\times$, adding to the computational demands. Given its low error rate, the 2nd generation data is especially useful in discovering small variants in the genome.

Some genomic variants, called *structural variants*, span across hundreds of base-pairs. Such variants are not easily detected with second generation reads. Third generation sequencing devices like the PacBio and Oxford Nanopore produce long reads of size 1 to 100 kilobases at a throughput of about 240 Mbases/min. Such long reads are more effective at capturing structural variants. However, these reads have a raw error rate of 10% to 30% (machine induced) and this leads to significant computational overhead in the alignment step. Because of this trade-off, it is expected that both 2nd and 3rd gen sequencing technologies will be used in tandem to identify a broad range of variants. Initial hybrid pipelines [11, 13] have been developed that, for example, use information from 2nd gen alignment to improve the 3rd gen alignment. Accelerators for sequence alignment can therefore be an integral part of 2nd gen, 3rd gen, and hybrid pipelines.

2.2 Second Generation Pipelines

GATK Pipeline. The Broad Institute has defined a standard second generation pipeline – GATK. The first stage in this pipeline is sequence alignment. Even with a reference genome, this stage can consume 4.5 hours on a modern CPU [28], thus providing a throughput of 12.5 Mbases/min, i.e., well below the throughput of the sequencing machine. The second stage cleans the output of alignment and the third stage, variant calling, distinguishes real variants (mutations) from machine artifacts.

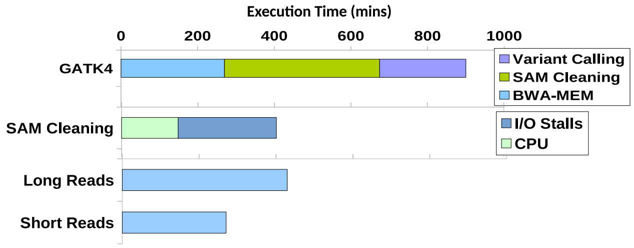
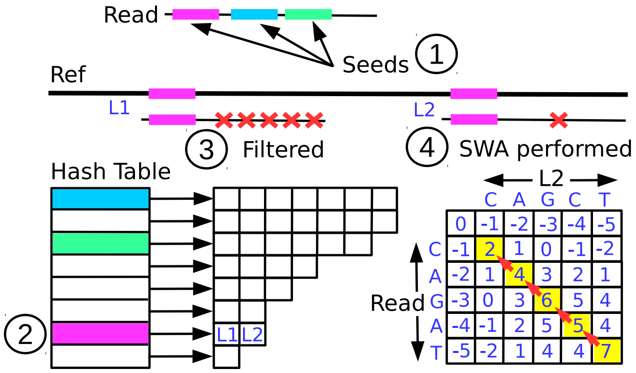


Figure 1: Execution time breakdown for stages of the GATK pipeline and comparison of time taken by long and short read alignments.

Figure 1 shows the breakdown of time taken in each stage of the pipeline when using 36 threads on an Intel Xeon E5 with 256 GB of memory. Roughly equal time is spent in sequence alignment (shown with the BWA-Mem algorithm), SAM cleaning, and variant calling. SAM cleaning is typically bottlenecked by I/O and is being addressed by other works using SSD and compression [4, 8]. In this work, we focus on the most compute-intensive task in the pipeline, sequence alignment, that impacts all of the first stage, and portions of the next two stages. Sequence alignment is also used in other genomic pipelines such as exome sequencing, RNA sequencing, etc. Figure 1 also shows that the computational overhead of alignment grows when we move from 2nd-gen short reads to 3rd-gen long reads.

Read Alignment Steps. As shown in Figure 2, a typical read alignment algorithm first partitions the read into small subsets called *seeds (step 1)*. The seed is used to index into a hash table to determine all the locations in the reference genome where this seed can be found *(step 2)*. These are the potential candidate locations where this read might align – L1 and L2 are the candidate locations in the example in Figure 2. An inexact match between the full read and these candidate locations is then performed. A quick inexact match is first performed to filter out a promising list of candidates *(step 3)*. In the example, only L2 advances to *step 4*, where a more compute-intensive inexact match is performed to score each candidate and identify the best alignment for the read.



		C	A	G	C	T
	0	-1	-2	-3	-4	-5
C	-1	2	1	0	-1	-2
A	-2	1	4	3	2	1
G	-3	0	3	6	5	4
A	-4	-1	2	5	5	4
T	-5	-2	1	4	4	7

Figure 2: A typical four step read alignment pipeline.

First Three Steps. A number of different heuristics are used to implement the first three steps, such as Hobbes, SMEM, Shifted Hamming Distance, and Myer's bit vector algorithm. We will shortly discuss each of these and the trade-offs they introduce. These steps require both significant memory bandwidth and compute operations. This work primarily improves these three steps by reducing its memory bandwidth requirments, supporting many parallel compute operations, and eliminating redundant locations that are forwarded to the fourth step.

Fourth Step. The compute-intensive inexact matching algorithm in the fourth step is usually a variant of Smith-Waterman local alignment (SWA) [33]. A read M of length m is matched against a reference fragment N of length n by filling up the cells of a matrix of size $m \times n$. The cells also store trace-back pointers such that the final alignment can be traced back from the highest scoring cell in the matrix. Many parallel implementations of this algorithm have been proposed using FPGAs and systolic arrays [12].

2.3 GenAx and Darwin

GenAx [9] and Darwin [35] are hardware architectures designed to accelerate second and third generation read alignment respectively. Both GenAx and Darwin use custom processing elements to accelerate the compute intensive SWA alignment step. GenAx serves as the baseline for much of our work. It uses the SMEM algorithm for the first three steps; some of the required operations are accelerated with CAM arrays. To reduce the memory bandwidth burden of the first three steps, GenAx uses a large scratchpad and tiling to store/reuse a subset of its hash table. Darwin also has a large scratchpad for storing counters used in the filtration technique called D-SOFT for long noisy reads. It also uses a tiling mechanism to manage the memory footprint of trace-back pointers. We later discuss more details of GenAx (Section 3) and Darwin (Section 6).

2.4 Bit-line or In-Cache Computing

Bit-line computing is a hardware technique for performing in-situ computations in an SRAM array [1]. When two wordlines are activated in parallel, the result sensed at bitline-bar (BLB) is the bitwise NOR and the result sensed at bitline (BL) is the bitwise AND. The major change is an extra row decoder for simultaneous activation and a reconfigured differential sense amplifier to sense BL and BLB separately. The bits at BL and BLB can be processed further using combinational circuits to produce results like XOR, OR, SUM, $CARRY$, etc. [14]. Since multi-row access can corrupt data, the wordline voltage is lowered, which slightly increases the access latency (1.5 ×). Such in-situ vector operations can unlock data-parallelism at sub-array level and can improve performance and energy due to reduced data movement.

2.5 Genomic Kernels

2.5.1 Seed Selection. The seed selection stage (step 1 in Figure 2) optimizes the number of candidate locations. The *Hobbes* algorithm fetches the frequency of all possible seeds of a read and then selects a batch of non-overlapping seeds with minimum aggregate frequency using a lightweight dynamic programming step. The *SMEM* algorithm finds the longest exact match at each position of the read (Super Maximal Extended Matches). Hobbes is good for reads with lesser errors but produces slightly more candidate locations. SMEM is good for reads with many errors but demands a higher memory bandwidth.

2.5.2 Filtering. Within the first three steps, filtration (step 3 in Figure 2) involves a significant amount of compute. Common filtration kernels perform bit-vector computations to estimate string similarity and are amenable to in-cache operations. We therefore provide an overview of the major filtration kernels used in our work, each targeting a different point in the trade-off space. Hamming Distance is not suitable when indels are present; Shifted Hamming Distance introduces a non-trivial false positive rate; Myer's is computationally more expensive. Note that some of these kernels can be combined to reduce compute and false positives; in our work, we find that a combination of SHD and Myer's works best in some phases.

Hamming Distance. The Hamming distance between two strings indicates the number of mismatches between the strings. The Hamming distance between the read and an equal length sequence at the candidate location in the reference indicates the number of mutations, while dis-regarding shifts because of insertions and deletions. Hamming distance of 0 indicates an exact match.

Shifted Hamming Distance (SHD). SHD is effective in identifying a small number of errors including insertions/deletions [38]. To check for *e* errors, the Hamming distance is computed after shifting the read by up to *e* places to the left and right, indicative of *e* insertions or *e* deletions (Step 11, 15 in Algorithm 1). After each shift, a Hamming mask is computed (Step 12, 16) – a vector of 0s and 1s that indicate if the bases match or not. These Hamming masks are then amended using certain heuristics (Step 13, 17) and combined using bit-wise operations (Step 14, 18) to estimate if the candidate location should be filtered or not (Step 7, 20). SHD can successfully filter reads with up to 5 errors without any false negatives, but with a 7% false positive rate.

Algorithm 1 Shifted hamming distance

```
 1: procedure SHD($ref[], read[], n, E$)
 2:     $HM \leftarrow ref \oplus read$                      ▷ Hamming Mask
 3:     $FM \leftarrow 1^n$                                 ▷ Final Mask
 4:     $AM \leftarrow Amend(HM)$                          ▷ Amended Mask
 5:     $FM \leftarrow FM \;\&\; AM$
 6:     $e \leftarrow$ Count 1s in HM
 7:     if $e < E$ then
 8:         return $pass$
 9:     else
10:         for all i ← 1 to E do                    ▷ Loop over $e$ errors
11:             $read\_tmp \leftarrow read \gg i$
12:             $HM \leftarrow read\_tmp \oplus ref$
13:             $AM \leftarrow Amend(HM)$
14:             $FM \leftarrow FM \;\&\; AM$
15:             $read\_tmp \leftarrow read \ll i$
16:             $HM \leftarrow read\_tmp \oplus ref$
17:             $AM \leftarrow Amend(HM)$
18:             $FM \leftarrow FM \;\&\; AM$
19:             $e \leftarrow$ Count 1s in FM
20:             if $e < E$ then
21:                 return $pass$
22:         return $fail$
23: procedure AMEND($Mask$)
24:     $M \leftarrow Mask$
25:     $M_{-1} \leftarrow Mask \ll 1$
26:     $M_{-2} \leftarrow Mask \ll 2$
27:     $M_{+1} \leftarrow Mask \gg 1$
28:     $M_{+1} \leftarrow Mask \gg 2$
29:     return $(M_{-1}\overline{M}M_{+1})|(M_{-2}\overline{M_{-1}M}M_{+1})|(M_{-1}\overline{MM_{+1}}M_{+2})$
```

Myer's Bit Vector Edit Distance. This algorithm is used to calculate the edit distance (including mismatches and indels) between two strings. This is calculated with dynamic programming, similar to SWA. Myer's algorithm is particularly amenable to in-cache operators as it uses bitwise operations. Algorithm 2 initially calculates the occurrence vector of each base (A/C/G/T) in the read string (Step 2). For example, for a read with two As in the beginning, the occurrence vector starts with two 1s. The algorithm then scans through each base in the reference string (Step 6) to calculate each column of the dynamic programming matrix. Essentially, the algorithm calculates the difference between two consecutive columns with the help of auxilliary vectors (Step 3, 4) and updates the number of edits. Apart from the addition operation in step 9, the rest are bitwise operations. In order to support Myer's algorithm in-cache or near-cache, addition and shift operations need to be supported. A minor transformation of Myer's algorithm is used to calculate banded edit distance to filter reads with more than *e* errors. This reduces computation by calculating only the cells of a banded diagonal.

Algorithm 2 Myer's Bit-vector algorithm for levenshtein distance

```
 1: procedure NUMBER_OF_EDITS($ref[], read[], n$)
 2:     $Peq[bp](i) \leftarrow (read_i == bp)$             ▷ Occurence vector
 3:     $Pv \leftarrow 1^n, Ph \leftarrow 0^n$               ▷ Auxilliary vectors
 4:     $Mv \leftarrow 0^n, Mh \leftarrow 0^n$               ▷ Auxilliary vectors
 5:     $Edits \leftarrow 0$
 6:     for all j ← 1 to n do                     ▷ Loop over $n$ bases
 7:         $Eq \leftarrow Peq[ref_j]$
 8:         $Xv \leftarrow Eq \;\&\; Mv$
 9:         $Xh \leftarrow (((Eq \;\&\; Pv) + Pv) \oplus Pv) \mid Eq$
10:         $Ph \leftarrow Mv \mid {\sim}Xh \mid Pv)$
11:         $Mh \leftarrow Pv \;\&\; Xh$
12:         if $Ph \;\&\; 10^{m-1}$ then
13:             $Edits \leftarrow Edits + 1$
14:         else if $Mh \;\&\; 10^{m-1}$ then
15:             $Edits \leftarrow Edits - 1$
16:         $Ph \leftarrow Ph \ll 1$
17:         $Mh \leftarrow Mh \ll 1$
18:         $Pv \leftarrow Mh \mid {\sim}Xv \mid Ph)$
19:         $Mv \leftarrow Ph \;\&\; Xv$
20:     return $Edits$
```

3 PROPOSAL

3.1 Architecture Overview

Without loss of generality, in this work, we model a baseline that resembles the GenAx architecture [9]. The proposed architecture, GenCache, is shown in Figure 3 with new components shaded blue. It operates as a co-processor (similar to a GPU card) and is equipped with significant memory bandwidth. The accelerator has large SRAM arrays; we add in-cache operators to these arrays. The input to the accelerator is a set of reads that must be aligned against the reference genome. These inputs are fed to a number of seeding lanes that execute the required seed selection and filtration heuristics (steps 1-3 in Figure 2). With help from a central controller unit, filtration commands are sent to in-cache operators in SRAM arrays, and responses are collected. The resulting candidate locations are then sent to an SWA engine where step 4 is performed to score each location and send final alignments back to the processor.

GenCache does not modify the SWA engine. It uses a new algorithm for steps 1-3 that requires fewer memory accesses and that

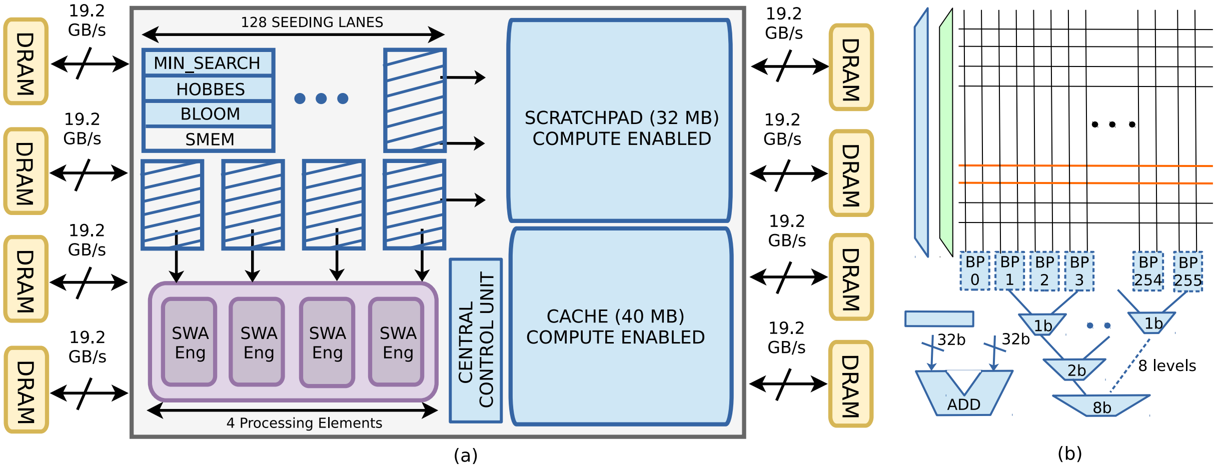


Figure 3: (a) Block Diagram of the GenCache Architecture. (b) Overview of bitlines and added logic in a subarray. Additional components over the GenAx architecture are colored in blue in both (a) and (b).

exposes more parallelism to the in-cache operators. The new algorithm also eliminates redundant reads that are sent to the SWA engine in GenAx. We are thus reducing the time taken in all steps and retaining a balanced pipeline. We preserve the same output quality as the baseline GenAx, i.e., we do not introduce any false negatives. While we will focus on 2nd-generation alignment here, similar approaches can be applied to 3rd-generation alignment as well (Section 6).

3.2 GenCache Operators

In this work, we propose adding compute capabilities to the large scratchpads in genomic accelerators – this has the potential to add a large amount of distributed compute with local wiring and local data movement. In addition, we observe that genomic workloads can benefit from new in-cache operators. This subsection describes these new operators and what role they play in genomics; the next subsection describes how the algorithms can be modified to leverage in-cache operators.

Supported Operators

Compute Caches [1] were based on the principle of bit-computing, where in-situ AND and NOR operations were performed on two wordline vectors. Subsequently, more compute elements were added within the bitline peripheral circuits of the SRAM arrays to perform operations like OR, XOR, and in-place COPY [1]. These works also add non-trivial full-adder circuits to a bit-line to iteratively perform addition and multiplication to support convolution operations in neural networks [7]. All of these operators are part of our design because they are useful for various kernels. In addition, we introduce five new RISC-like operators to efficiently implement SHD and Myer's algorithms. These operators are:

(1) Hamming Mask (HM): computes the hamming vector between two strings; required by exact match, and in steps 2, 12, 16 of SHD (Algorithm 1) and step 2 of Myer's (Algorithm 2).
(2) Hamming Distance (HD): adds the 1s in a hamming vector; required by exact match, and in steps 6, 19 of SHD (Algorithm 1) and steps 12, 14 of Myer's (Algorithm 2).
(3) Shift Left (SL): required in steps 15, 25, 26 of SHD and steps 16, 17 of Myer's.
(4) Shift Right (SR): required in steps 11, 27, 28 of SHD.
(5) 32-bit addition (ADD): required in step 9 of Myer's.

For some of the above operators, we create both 1-bit and 2-bit versions. In genomic workloads, the input operand is often a base-pair (ACGT), which is represented with 2 bits.

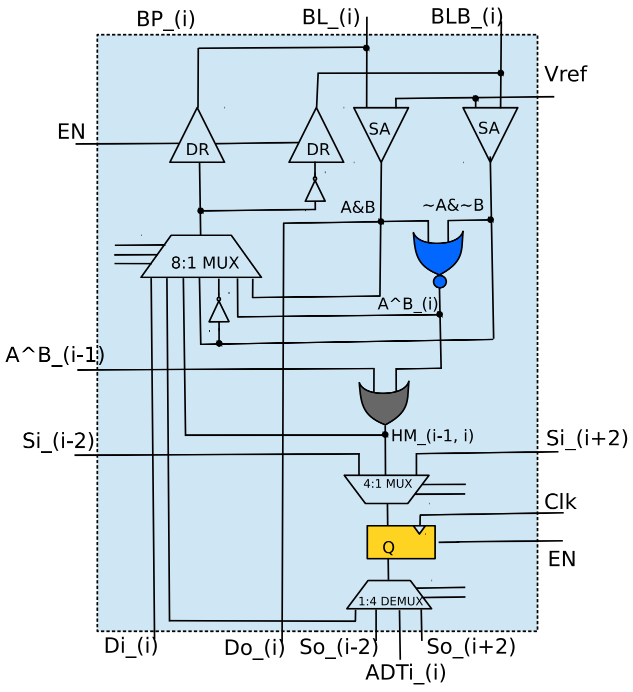


Figure 4: Bit-line Peripheral Circuit for GenCache

Additional Peripheral Logic

The above operators require extra logic that is implemented adjacent to the bitline sense-amps. The overhead of this extra logic is quantified later with a synthesized design. This bitline peripheral circuit for bitline i is shown in Figure 4. When two wordlines A and B are activated, the results emerging from the sense-amps are $A\&B$ and $\sim A\& \sim B$. These are fed to a NOR gate (colored BLUE) to produce an XOR result. Next, we introduce connections between adjacent bitlines. This is useful to not only perform a shift, but also to perform operations on 2-bit values. We introduce an OR gate (colored GRAY) for even-numbered bitlines that gathers the XOR result for two consecutive bitlines and decides if the 2-bit bases in rows A and B are the same or different (producing the signal HM or Hamming Mask). Next is a latch (colored YELLOW) that is used to perform 1-bit or 2-bit shift operations. Finally, the result of the bitline operation is sent to an adder tree used for aggregation; Hamming masks are produced at bitlines, while the hamming distance is produced at the adder tree. We implement a 128-bit adder tree (assuming inputs from odd-numbered bitlines in a 256-wide array, shown in Figure 3b) with 0.55 ns critical path latency at 28 nm technology.

In addition, prior work [7] has shown how word-granularity additions (say, addition of two 32-bit words) can be performed with bitline operators. Those implementations incur overheads to map data and iteratively perform addition over several cycles. Instead, we implement a dedicated 32-bit adder adjacent to the subarray, shown in Figure 3b. This adder receives input operands in consecutive cycles (the first operand is buffered in a register). This allows addition of two 128-bit rows (as required by step 9 of Myer's) in 4 cycles.

We later show that supporting these new operators results in a modest area overhead. In addition to the 5 new RISC-like operators introduced above, we introduce the following 4 CISC-like operators that perform the filtration kernels mentioned in Section 2.5.2: *SHD*, *SHD_C*, *MYERS*, and *MYERS_B*, where *SHD_C* performs SHD filtering for $e + 1$ errors using the result of *SHD_C* with e errors and *MYERS_B* performs banded Myer's algorithm. Each "CISC" instruction is made up of multiple RISC operators and is parameterized by the number of errors or the read length (Step 10 of SHD, Step 6 of MYERS). A ROM stores the sequence of 4b RISC instructions for each CISC instruction.

Not shown in Figure 4 is a local control circuit that receives commands from the central controller and implements the RISC/CISC API; in addition to receiving/buffering addresses, a 4-bit command is received to perform the operation and return the data response after a fixed number of cycles. The local controller "unrolls" a CISC instruction into a set of RISC intructions using the received parameter and the ROM sequence. Each RISC instruction is translated into 7b select signals for the MUX and DEMUX logic in the bit-line peripheral circuit. We have synthesized the local controller circuit and its area footprint is only 225 μm^2.

3.3 2nd-Gen Read Alignment

Having introduced various in-cache operators, we describe how the 2nd-gen read alignment algorithm can be modified to unlock the potential of GenCache.

SRAM Allocation: GenAx and GenCache

During the many steps of read alignment, the reference genome and the hash tables for seed positions are repeatedly accessed. Since these structures are too large to fit in cache, prior work GenAx uses the following tiling approach to alleviate the memory bottleneck. It partitions the reference genome into 2 MB slices and a hash table is constructed for each slice (see Figure 6). This hash table has a capacity of about 64 MB. GenAx thus allocates its on-chip storage for 2 MB of reference and 64 MB of hash table, and processes the entire batch of input reads before bringing in the next slice of reference and hash table. The storage is organized as two scratchpads.

The values in the hash table exhibit a low computation/byte ratio, while the values in the reference genome are involved in a majority of computations. If the reference genome occupies only 2 MB of on-chip storage, it can only leverage a small fraction of in-cache operators, thus limiting the potential speedup. Therefore, we first change our resource allocation. A much larger slice (24 to 48 MB) of the reference genome is placed in a scratchpad with compute capability. The remaining on-chip storage is then managed as a cache for the corresponding hash table. Given the large size of the hash table, the high miss rate in the cache is a potential bottleneck. We alleviate this bottleneck by re-structuring the read alignment software pipeline described next.

A New Error-Aware Algorithm

Figure 5 shows the number of errors encountered per read when aligning against the reference genome. We show results for two popular software packages, BWA-MEM [17] and SNAP [39]. We see that about 75-80% of the reads align with the reference with 0 errors (exact matches), 15% align with one error, 5% with 2-5 errors and the remaining with 6 or more errors. This distribution is consistent across most second generation read datasets since two human genomes typically differ by less than 0.1%. A key observation here is that higher efficiency can be extracted by treating each of these error cases differently.

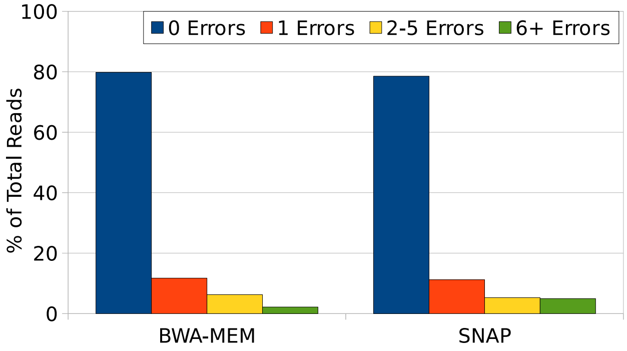
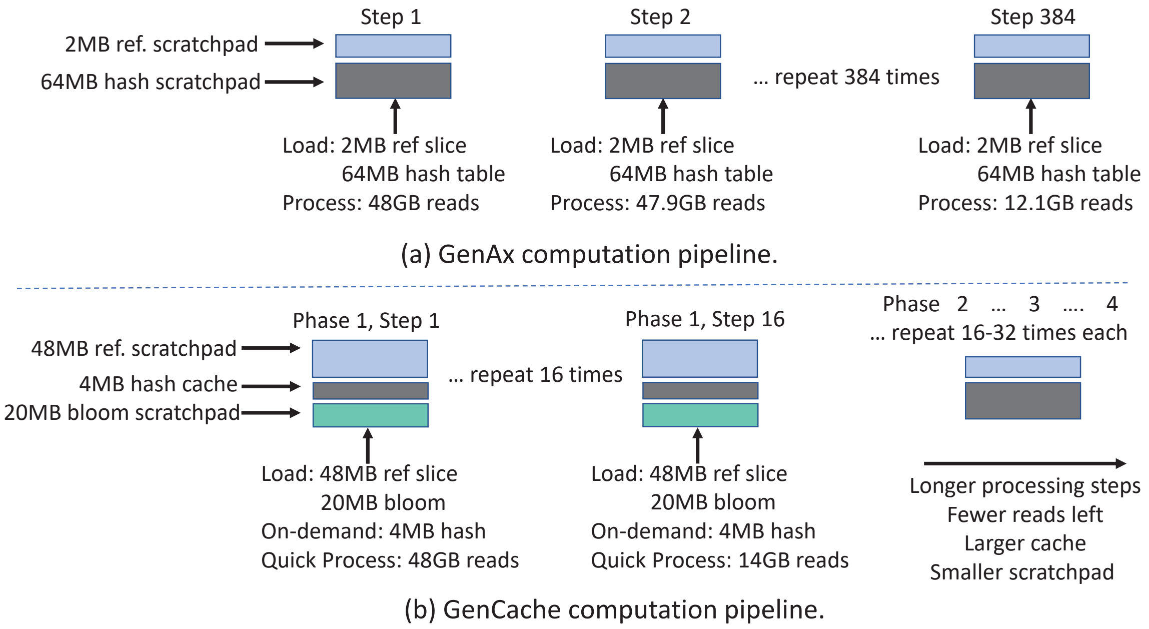


Figure 5: Read alignment profile for BWA-MEM and SNAP.

The seed-and-extend algorithm in GenAx divides a read into chunks of size 12 bp (seeds) and accesses the hash table to fetch locations where each of the seeds occurs in the reference genome. If neighboring seeds in a read return neighboring locations in the reference, the location is considered as a potential candidate. Potential candidates are then scored using the "extend" engine. It is



(a) GenAx computation pipeline.

(b) GenCache computation pipeline.

Figure 6: Comparing the GenAx and GenCache computational pipelines.

this step of fetching locations of all the seeds from the large hash table that creates a potential memory bottleneck.

For example, consider a read of length 100 bp that is divided into 8 seeds, each of size 12. Assume that the frequency of occurrence of each seed in the reference genome is given by the array [512, 256, 128, 64, 32, 16, 8, 4]. After fetching these 8 frequency counts and pointers to the corresponding list of locations, GenAx then fetches a total of 1020 locations from the scratchpad. If the intersection of the sets of locations per seed is non-null, GenAx declares a perfect match.

GenCache takes a different multi-phase approach in selecting seeds for a read. The pigeon-hole principle states that given a read with e errors, we need $e + 1$ seeds such that at least one of the seeds will be error free. When the read has zero errors, all seeds are error-free. We can therefore pick the least frequent seed and check for an alignment against locations for that seed. In the example read above, GenCache creates seeds, then fetches 8 hash table locations (the frequency counts above). It then focuses on the 4 locations for the last seed and performs an in-cache perfect match for the full read against those 4 locations – note that a large 32 MB slice of the reference genome has already been pre-loaded. We are thus fetching less than 10 memory locations (the frequency counts) in the common case and performing a handful of parallel in-cache matches, whereas GenAx reads 1000+ scratchpad locations, followed by a set intersection operation.

Similarly, in order to align a read with 1 error (mismatch or indel), we can narrow our search to the two most infrequently occurring seeds. We therefore employ a 4-phase algorithm, shown in Figure 7, where each phase deals with a different error scenario. The first phase deals with exact matches, the next phase deals with 1-error matches, the next with 2-5 errors, and the fourth phase handles 6+ errors. Each phase uses a different algorithm to identify an efficient set of seeds, a different set of operations to perform the matches, and different data structures. All of this is summarized in Figure 7; we walk through the details of each phase in Section 3.4. The bottomline is that our algorithmic change avoids a number of memory fetches in the common case (with few errors) and our hardware in-cache operators can perform highly parallel filtering operations.

Exploiting a Bloom Filter

One of the bottlenecks in our design is the large number of hash table look-ups and the potential for a high cache miss rate, especially when a large fraction of on-chip space is allocated for the reference genome. As we will describe in the next sub-section, futile or non-useful look-ups of the hash table in Phases 1 and 2 can be filtered out with a Bloom Filter.

Architectural Comparison

Architecturally, consider the contrast between GenAx and GenCache, summarized in Figure 6. The reference genome is a 3 billion base-pair structure that is encoded in 768 MB. GenAx partitions the reference into 384 2 MB slices and hash tables specific to each slice. A slice and hash table are brought into the GenAx scratchpads, and alignment is performed for the entire batch of reads for one human (800M reads). The reads that find perfect matches are set aside and are not matched against subsequent slices. As a result, the set of reads to be processed shrinks gradually as GenAx iterate through the slices. This pipeline increases reuse of the reference and hash table, but leads to high computational overhead and high bandwidth demand when fetching the reads. *It also results in early approximate matches being sent to the SWA engine; these are futile/redundant computations if a perfect match is discovered later.*

Instead, in GenCache, the reference genome is partitioned into 16 48 MB slices. All slices are first processed in Phase 1 that looks for exact matches, so that a vast majority of reads can be handled quickly with low computational overhead. *Also, unlike GenAx, these perfectly matched reads never send redundant work to the SWA* *engine.* The remaining 25% of reads then go through Phase 2, again iterating through 16 reference slices, performing the slightly more expensive 1-error match operations. More iterations are performed again in Phase 3 and Phase 4, with even more computations per read and an even smaller set of reads. The reference genome is therefore fetched from memory 4 times and we also incur more memory accesses because of large hash tables that do not fit in the hash table cache. Note that in GenAx, a read is fetched 1-384 times depending on if/when it finds a perfect match, with 20-25% of reads going through all 384 iterations because they never find a perfect match. On the other hand, in GenCache, a read is fetched 1-96 times with 75-80% of reads finding a perfect match in the 16 iterations of Phase 1.

We are thus trading higher memory bandwidth for reference and hash table, for lower memory bandwidth for input reads and a lower computational burden. We show in Section 5 that this is a worthwhile trade-off and results in fewer overall cycles for memory fetches and compute.

3.4 Details of the Four-Phase Algorithm

In this sub-section, we describe the design details (both algorithm and mapping to hardware) for the four phases described in Figure 7 for 2nd-gen read alignment.

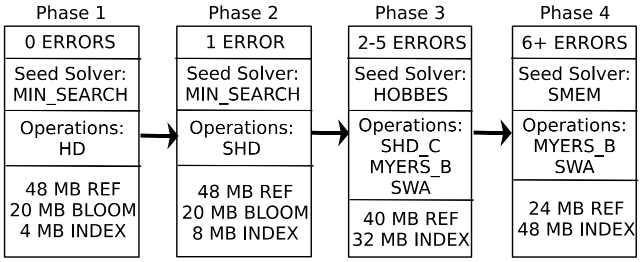


Figure 7: Four phases in the new alignment algorithm that exploits in-cache operators.

Seed Selection

Each phase starts with a seed solver that identifies a set of seeds that is most favorable for next steps in the algorithm. The first two phases are most concerned with identifying 1 or 2 seeds that occur least frequently. The *Min_Search* modules walk through various seeds and track their occurrence count with hash table look-ups to find seeds with minimum frequencies. Phase 3 uses the Hobbes algorithm, which performs a single (lightweight) dynamic programming operation per read. The SWA engine of GenAx is used for this. Phase 4 uses the SMEM algorithm, tailored for high-error scenarios and used in GenAx and BWA-Mem. Like GenAx, a TCAM array is used to perform the SMEM operation. Figure 3 shows hardware units that implement the control for each of the above seed solvers.

Phases 1 and 2

Most reads are handled by Phase 1 and 2 that leverage in-cache operators. In phase 1, we are looking for a perfect match against a few reference genome locations. We therefore write the read into specific cache subarrays, perfectly aligned with the candidate reference genome locations (in some cases, the read may be split across two subarrays). A Hamming Distance (HD) in-cache operation is performed for each of these locations. The central controller receives responses; when a perfect match is discovered, the read does not advance to subsequent phases and subsequent reference genome slices. Phase 2 follows a similar process, but must consider the hamming masks of the left- and right-shifted versions of the read (using RISC operations SL, SR, HM).

Phase 3

In Phase 3, after using the Hobbes unit to identify seeds and candidate locations in the reference genome, the following in-cache operations are used to filter out the worst locations. An SHD CISC operation is performed at each location to filter out locations with more than 5 errors. Since SHD lets through a small number of false positives (7%), a *MYERS_B* CISC operation (described in Section 3.2) is also performed to identify the true positives. The list of true positives is then sent to the SWA engine.

Phase 4

In Phase 4, after using the hardware SMEM unit (already included in GenAx) to identify the best candidate locations in the reference, a *MYERS_B* operation is performed at each location. Reads with edit distance less than 40 are advanced to the SWA engine.

3.5 Reducing Hash Table Accesses

SRAM Allocation to Data Structures

We observe that each phase places different demands on the various data structures; there is the potential for higher efficiency if the on-chip storage is re-allocated across the data structures at the start of each phase. For example, Phase 1 handles the most reads and benefits the most from the higher parallelism afforded by many subarrays, whereas Phase 4 performs more look-ups of seed locations for the SMEM seeding algorithm. In other words, Phase 1 needs a larger reference genome scratchpad (48 MB), while Phase 4 needs a larger hash table cache (48 MB). We therefore design the arrays so that 48 MB of SRAM support in-cache operators and leave it up to the central controller to configure/load the arrays at the start of every phase. The SRAM allocation of different data structures in the four different phases is shown in Figure 7.

Improving the Hash Table Cache

When a reference genome slice grows, its hash table size grows in proportion, while receiving an even smaller allocation for the hash table cache. We address this problem with the following insight: the hash table for a reference slice is highly skewed in its distribution (shown in Figure 8), with 70-80% of seeds having zero occurrences in the reference genome, 14-17% having a single occurrence and 6-13% having more than 1 occurrence. Seeds with 0 or 1 occurrence are especially useful during Phase 1 and 2. If we find a seed with 0 occurrences, the read cannot find a perfect match and does not need further processing in Phase 1. If we find a seed with 1 occurrence, a single HD operation on that location is enough to discover or give up on a perfect match in Phase 1. As described next, we leverage this observation to improve hash table cache efficiency.

Bloom Filter Details

We use a Bloom Filter to determine if a seed has more than 1 occurrence in the reference genome slice (a smaller set that is easier to track with a Bloom Filter). In Phase 1, we fetch Bloom Filter entries for seeds in a read until the Bloom Filter finds a seed with 0 or

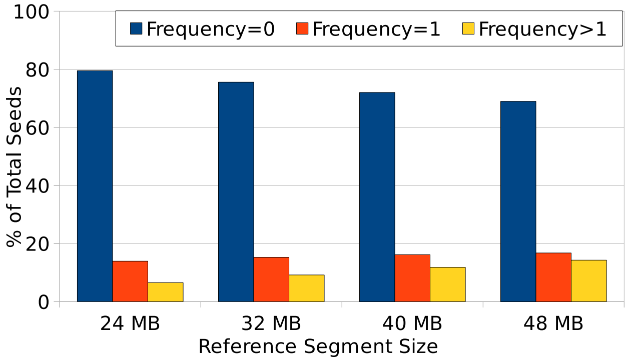


Figure 8: Distribution of seed occurrences for different reference genome slice sizes.

1 occurrence. That seed is then looked up in the hash table (hopefully in cache). If the hash table entry is empty (0 occurrences for that seed), the read will not have a perfect match and it is skipped over. If the hash table entry has a single occurrence, we perform an in-cache Hamming Distance operation to identify a perfect match. A similar Bloom Filter structure is also used for Phase 2.

The Bloom Filter implementation is a storage-efficient way to identify seeds with 0/1 frequency in a skewed hash table, which is crucial for Phases 1 and 2. This is a novel application of Bloom Filters, unlike prior caching policies [27] that use Bloom Filters for cache hit/miss prediction. Because the Bloom Filter is on chip, it is also a more efficient structure to identify candidate locations. Without this approach, hash table entries with zero or many occurrences put pressure on bandwidth, cache capacity, and in-cache operators.

4 METHODOLOGY

Software. For short read alignment, we use the single-end ERR194147_1.fastq read dataset with 787M reads of length 101 bp, as used in GenAx. We compare the alignment accuracy of our modified pipeline against that of GenAx and BWA-MEM [17]. GenCache accuracy matches that of GenAx because (i) the 4-phase algorithm only eliminates redundant SWA computations which would have had no impact on the GenAx output, (ii) the techniques used in the first 3 phases such as Hobbes, SHD, and MYERS, have no false negatives, and (iii) for complex alignments, GenCache falls back to the GenAx SMEM heuristic. Similar to GenAx, GenCache has a 0.0023% variance with the BWA-MEM output. This is because the GenAx SWA engine used in GenCache to make an apples-to-apples comparison uses a different tie-breaking mechanism than the software BWA-MEM algorithm.

Circuit Models. We synthesize the bitline peripheral circuit, the 128-bit adder tree, the 32-bit adder, the new seed solver circuits, the Bloom Filter hashing circuit, and the central controller using Synopsys Design Compiler. We design 8:1 and 4:1 MUXes in our peripheral circuit using smaller gates from a standard cell library. The column circuit consists of two parts: the analog component (sense amps and write drivers) and the digital component (rest of the circuit). We pitch-match the analog component with every SRAM cell to reduce variability due to noise. The digital component of the bitline peripheral is used for every even-numbered bitline and can be pitch-matched across two bitlines (10 tracks), where our 8:1 or 4:1 mux can fit easily. However, the D Flip Flop occupies 20 tracks, which is why we use two rows of peripheral circuits below the SRAM array. Each row consists of the digital component of every fourth bitline, each pitch-matched to four bitlines (20 tracks). We validate the floorplanning through a place-and-route flow. The central controller includes logic for scratchpad allocation/indexing and a finite state machine to manage the responses of in-cache operations. The generated gate level netlists were converted to a SPICE netlist using FDSOI 28 nm technology node to model the delay, power, and energy values. We model GenCache by integrating these circuit estimates within Cacti's subarray and cache models [23]. In order to model a scratchpad, we use the modified version of Cacti within the Aladdin toolset [32]. For memory fetches, we assume energy of 51 pj/bit for DDR4 [29]. Table 1 lists the main components in the GenCache architecture and their power and area (all scaled to 32 nm). Table 2 summarizes performance and energy for each operator in GenCache. The bit-line peripheral circuit adds to the area overhead of the cache subsystem by 14.7%. To accommodate the Bloom Filter, we use 4 MB more SRAM than GenAx, an additional SRAM overhead of 5.9%. A conventional 256×256 SRAM array read takes 650 ps whereas a compute-enabled access to GenCache takes 1040 ps. The adder tree and 32-bit adder occupy negligible area. Overall, the GenCache chip has 16.4% higher area, 34.7% higher peak power, and 15% higher average power than GenAx.

Components	Area	Peak Power
Subarray		
Bit-line Peripheral (x128)	2842 μm^2	1.98 mW
Adder Tree	672 μm^2	0.369 mW
32-bit Adder	70 μm^2	0.126 mW
SRAM memory		
GenAx Scratchpad (68 MB)	189.2 mm^2	12 W
GenCache Scratchpad (32 MB)	102.1 mm^2	8.42 W
GenCache Cache (40 MB)	117.6 mm^2	10.33 W
Seed Solvers (x128)		
Min Finder	0.015 mm^2	5 mW
Hobbes Control	0.060 mm^2	169 mW
Bloom Hashes	0.073 mm^2	247 mW
SMEM	5.78 mm^2	4.4 W
Central Control Unit	2.59 mm^2	1.53 W
SWA Engine (x4)	7.43 mm^2	8.62 W
GenAx Chip Total	202.41 mm^2	25.02 W
GenCache Chip Total	235.68 mm^2	33.72 W

Table 1: Area and power of different components in GenAx (yellow/orange) and GenCache (white/orange) at 32 nm.

Architecture Model. Since GenAx already shows an order of magnitude improvement over a 56-thread CPU baseline and an NVIDIA Titan Xp GPU [9], we compare our architecture against GenAx only. For the GenCache architecture, we designed a cycle accurate simulator that models the latency and bandwidth of each component (cache HTree, cache subarray, queuing, instruction issue width of the seeding lanes, etc.). For the scratchpad, we use 16

Operations	Cycles $(@2.0GHz)$	Energy/Op (pj)	Throughput/Area $(MReads/s/mm^2)$
HD	7	46	12268
SHD (1e)	71	659	1136
SHD (5e)	328	3109	244
SHD_C	64	613	1252
MYERS_B (5e)	501	2079	160
MYERS_B (40e)	3413	14085	23.5
MYERS_B (90e)	7571	31237	10.6
GenAx SWA (40e)	456	491340	2.36

Table 2: Comparison of Latency, Energy and Peak Throughput of different GenCache Ops at 128 bp granularity for a 32 MB scratchpad (102.1 mm^2) with GenAx SWA Engine (1.86 mm^2).

slices of 2 MB each, where each slice has 64 banks and each bank is equipped with 512 I/O wires. This provides high bandwidth for GenCache to send the reads to the appropriate subarrays at the cost of leakage power. Each subarray is 256×256 bits. The bitline peripheral circuit is used for every even-numbered bitline, offering a parallelism of 128 bit-wise operations per subarray. Each seeding lane processes both forward and reverse read at a time. The seeding lanes also buffer and overlap processing for 4 reads to tolerate hash table access latencies. The CAM array used by the SMEM algorithm in phase 4 is used as a cache for each seeding lane in phase 1-3 to temporarily store reads and locations of seeds.

5 RESULTS

Figure 9 shows the throughput improvement as each of our key innovations is incrementally introduced. By adding in-cache operators to the 2 MB reference slice in GenAx, and no algorithmic changes, we see a marginal improvement of 5% due to the limited parallelism. If the reference slice is allocated 32 MB (with in-cache operators) to create a basic GenCache, we see a 1.62× speedup over GenAx; while the operators allow high parallelism, the system continues to be bottlenecked by memory accesses because the hash tables experience frequent cache misses. The fourth bar introduces the 4-phase algorithm (eliminating redundant computations) for the baseline GenAx architecture (without in-cache operators). This only yields a 1.36× speedup because of more iterations through the reads and limited parallelism for filtration. The same 4-phase approach over basic GenCache yields a 2.5× speedup due to (i) fewer fetches for reads (which accounts for a large fraction of memory bandwidth) and (ii) the high degree of parallelism offered by GenCache operators. *This is therefore a case of the whole being greater (4× speedup from in-cache operators and algorithm) than the sum of its parts (1.36× from algorithm alone and 1.62× from in-cache operators alone).* The addition of the bloom filter alleviates the remaining memory bottleneck from hash table misses in Phases 1 and 2 to yield a 5.26× speedup over baseline GenAx.

Figure 10 shows an energy comparison (in terms of reads per mJ) and memory fetches for the same six configurations. Most of the energy reduction is from a reduction in memory accesses. The in-cache operators by themselves do little to reduce memory accesses – in fact, they increase memory accesses because of a higher cache miss rate on hash table look-ups. A larger reference slice in

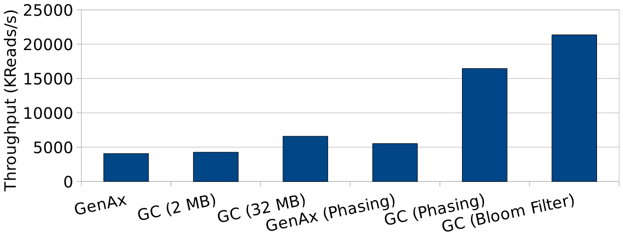


Figure 9: Throughput improvement of GenCache (Hardware & Software).

third, fifth, and sixth bars leads to fewer memory fetches per read. Energy improvement is the highest (8.6×) when using the 4-phase algorithm with bloom filter, due to reduced memory access and lower runtime.

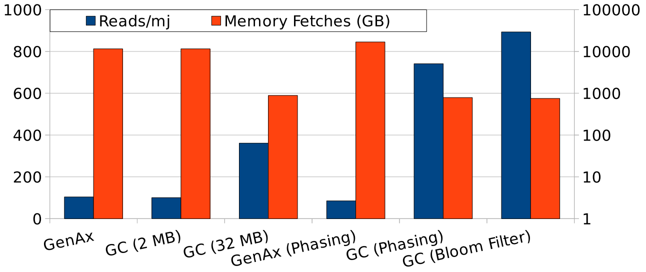


Figure 10: Energy improvement (in Reads/mJ) and number of memory fetches (log scale) of GenCache (Hardware & Software).

Figure 11a further breaks down the memory accesses across phases and data structures (note log scale). In the baseline GenAx, most memory accesses are for the reads, parts of which are fetched 384 times. In GenCache, most memory accesses are again for reads, but by eliminating redundancy, the total count is much lower. While the hash table accesses are fewer than the read fetches, the hash table accesses tend to be on the critical path, so reducing them has a large impact on performance. Figure 11b breaks down the time taken and energy consumed in each phase. Phase 1 and 2 take less time because of the high parallelism offered by in-cache operators HD and SHD. Phase 4 takes the most amount of time because it is bottlenecked by the CAM lookups for the SMEM based filtration. Phase 1 and Phase 4 consume high energy because of memory fetches and CAM lookups respectively.

Figure 12a shows the Bloom Filter size required for different false positive rates and reference genome slices. The SRAM allocation for phases 1 and 2, shown in Figure 7, is based on this data. Figure 12b shows the reduction in the miss rate for the hash table cache. Note that the hash table itself is 1.5 GB, of which only 4 MB is cached. We are using a 20 MB bloom filter to approximately track high-frequency entries in the 1.5 GB table. The bloom filter not only reduces futile memory bandwidth (captured in Figure 10), it also yields a higher cache hit rate by avoiding pollution. This is especially important given that most of the on-chip storage has been apportioned for storing other compute-friendly data structures.

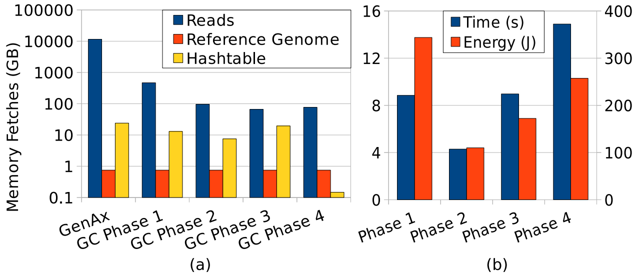


Figure 11: (a) Breakdown of time and energy by program phase. (b) Breakdown of memory accesses by program phase and data structure.

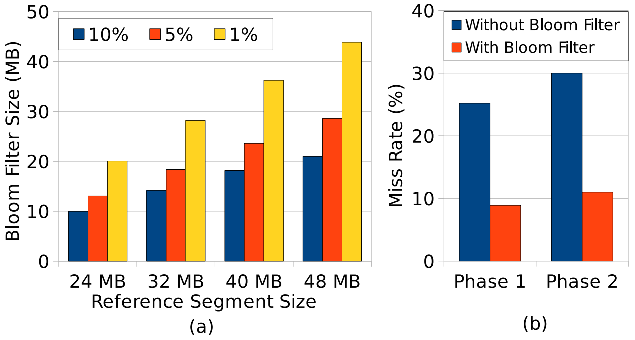


Figure 12: (a) Bloom Filter sizes for varying reference slices and false positive rates. (b) Reduction in miss rate for Phase 1 and 2 when using a bloom filter with 10% false positive rate.

Iso-area Analysis and Effect of HBM. As a sensitivity analysis, we reduced the on-chip SRAM allocation of GenCache to 62 MB for an iso-area comparison with GenAx. In spite of the reduced parallelism and cache space for hash tables, we observe a 4.3× speedup over GenAx. Using an HBM interface with 256 GB/s bandwidth instead of a Bloom Filter does little to improve the performance (8%) of the 4-phase algorithm because there aren't as many hardware threads to hide the latency of memory accesses.

6 APPLICATION TO 3RD GENERATION ALIGNMENT

GenCache operators are useful for other genomic operations such as Indel Realignment in the GATK pipeline. Here, we show its potential in the context of a 3rd generation kernel.

Baseline. We assume a Darwin-like baseline that uses D-SOFT filtration for long reads. D-SOFT finds seeds from a read that map to multiple non-overlapping bins of a reference genome. Depending on the number of bases that match in a bin, the bin is chosen as an *anchor*. The read is then iteratively aligned against the reference and extended (SWA step) on either side of the anchor using smaller tiles (512 bp).

In-Cache Filtration. We leverage the observation that D-SOFT produces 100-10,000 false hits per read, 97% of which can be easily pruned by using a thresholding operation on the score of the first SWA tile. *Instead of using an SWA operation, we propose to use a MYERS_B operation on the first tile as a filtration step, which is amenable to acceleration with GenCache.* This approach only advances 3% of D-SOFT hits to the Darwin SWA engine, while not impacting accuracy.

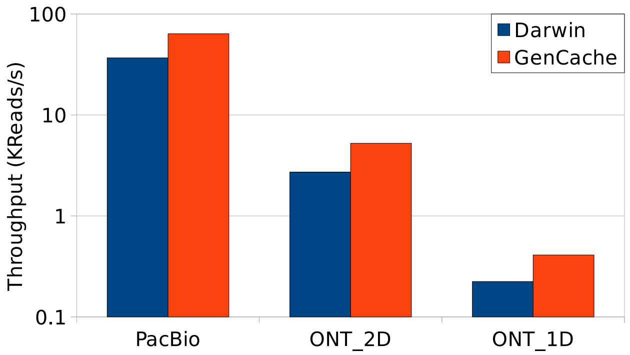


Figure 13: Throughput for 3rd gen workloads (log scale).

Methodology. We simulated three sets of reads of size 10 Kbp and 30× coverage using PBSIM [26], representing the various 3rd gen sequencing methods and their error profiles. PacBio mimics a 15% error rate of Pacific Bioscience devices. ONT_2D and ONT_1D mimic the error rates of Oxford Nanopore devices at 30% and 40% respectively. Darwin uses a 64 MB scratchpad for storing the counters needed for the D-SOFT algorithm for all the reference bins (for bin_size = 128). In our design, we allocate 16 MB to cache the hash table and prefetch the locations for neighboring seeds in a read; we allocate 16 MB for counters; the remaining 32 MB (with in-cache operators) is used to store a slice of the 768 MB reference genome. A 32 MB slice of the reference consists of 1M bins of size 256 bp, which needs 1 MB of counters; with a 16 MB allocation for counters, we process 16 reads in parallel.

Results. Figure 13 shows the improvement from using this in-cache filtration (log scale). GenCache improves the performance of 3rd gen alignment by 1.8×. This is because of two main reasons: (i) the 32 MB scratchpad provides an additional throughput of 1 BReads/s for the Myers Edit distance step along with the 21 MReads/s throughput provided by the Darwin SWA engine, (ii) the cache for the hash table and the fewer counters increases the throughput of the counter update in D-SOFT (thus creating alignment tasks at a faster rate for the SWA engine). In terms of chip area, the Darwin chip is estimated to be 264 mm^2, whereas the GenCache design is estimated to be 289 mm^2 at 32 nm.

7 RELATED WORK

There have been many efforts to improve sequence alignment with a variety of approaches, ranging from hardware acceleration to distributed bioinformatics runtimes. Accelerators like GenAx [9] and Darwin [35] have focused on sequence alignment, while efforts like GateKeeper [2] accelerates SHD-based filtering, Wu et al. [37] use FPGAs to accelerate INDEL realignment in the Cloud, and the Dragen system uses FPGAs for alignment and variant calling [22]. The FPGA approaches of accelerating filtering or alignment have the following downside in comparison to GenCache:

(i) they provide lower parallelism in comparison to in-cache operators, (ii) they incur additional bandwidth penalty to fetch reference segments, which remain in-place for GenCache, and (iii) they incur frequent cache misses due to smaller SRAM caches. Madhavan et al. [19] explore the use of race logic to perform SWA operations by encoding information as timing delays in the circuit. Distributed systems like Persona [5] and Adam [20] take a wholistic approach to combine different phases and build a cluster-scale, high-throughput bioinformatics framework.

A large body of recent work has exploited opportunities for near data processing [3], including some that embed operators into memory arrays. Other in-SRAM operation based accelerators include PROMISE [34] which uses a mix of analog and digital circuits to support matrix-multiplication operations for machine learning workloads. The NAND-Net architecture [16] focuses on in-SRAM and in-DRAM implementations of binary neural networks. Architectures such as DRISA [18] and Ambit [30] based on in-DRAM operations accelerate neural networks and database search applications respectively. The ISAAC [31] and PRIME [6] architectures implement analog dot-product operations within resistive crossbars to accelerate deep neural networks.

The Compute Cache [1] work proposes bit-line computing enabled caches, and evaluates a String-Match kernel, but has not been used to accelerate genomic workloads. GenCache is the first work to leverage such in-cache operators to provide significant speedups in sequence alignment.

8 CONCLUSIONS

In this work, we show that in-cache operators can be leveraged to provide significant speedups in sequence alignment. In particular, the filtration operations require access to large datasets and prior work has overcome this bottleneck with tiling. With new in-cache operators and a re-configured memory hierarchy, we show the potential for high parallelism. We analyze the sequence alignment workload to identify redundant work and create a new error-aware four-phase pipeline that is a better fit for the GenCache architecture. The algorithm alone, or the in-cache operators alone yield small speedups of 1.36× and 1.62×; their combination yields a more than additive speedup of 4×. We introduce a new Bloom Filter structure to reduce futile accesses to our largest data structure, thus offering more on-chip capacity for in-cache operations and boosting the speedup to 5.26×. The improvements are caused by higher parallelism, fewer memory fetches, and elimination of redundant work. The 15× reduction in memory accesses is especially important when future security/privacy measures will further penalize off-chip accesses. Our circuit analysis shows that the additional in-cache logic has a small area/power overhead. We also show significant benefits by exploiting in-cache operators in 3rd gen pipelines. We have thus helped alleviate the significant memory bottleneck noted by both recent genomic accelerators, GenAx and Darwin. The introduced operators can be exploited by other stages of the genomic pipeline for additional improvements and are left for future work: reference-based compression, Indel Realignment, Variant Calling, protein sequencing. The combination of in-cache operators, algorithm re-structuring, and the proposed SRAM allocation into cache/Bloom/scratchpad can also apply to other workloads that rely on bitwise operators and that must manage irregular memory accesses. This paper therefore provides further evidence that in-cache operators are useful for a broad class of applications.

ACKNOWLEDGEMENT

We thank the anonymous reviewers for many helpful suggestions. This work was supported in parts by NSF grant CNS-1718834 and NSF career award #1751064.

REFERENCES

[1] Shaizeen Aga, Supreet Jeloka, Arun Subramaniyan, Satish Narayanasamy, David Blaauw, and Reetuparna Das. 2017. Compute Caches. In *Proceedings of HPCA-23*.

[2] Mohammed Alser, Hasan Hassan, Hongyi Xin, Oğuz Ergin, Onur Mutlu, and Can Alkan. 2017. GateKeeper: A New Hardware Architecture for Accelerating Pre-Alignment in DNA Short Read Mapping. *Bioinformatics* 33 (2017).

[3] Rajeev Balasubramonian, Jichuan Chang, Troy Manning, Jaime H. Moreno, Richard Murphy, Ravi Nair, and Steven Swanson. 2014. Near-Data Processing: Insight from a Workshop at MICRO-46. In *IEEE Micro's Special Issue on Big Data*, Vol. 34.

[4] James K. Bonfield and Matthew V. Mahoney. 2013. Compression of FASTQ and SAM Format Sequencing Data. *PLoS One* 8 (2013).

[5] Stuart Byma, Sam Whitlock, Laura Flueratoru, Ethan Tseng, Christos Kozyrakis, Edouard Bugnion, and James Larus. 2017. Persona: A High-Performance Bioinformatics Framework. In *Proceedings of USENIX-26*.

[6] Ping Chi, Shuangchen Li, Ziyang Qi, Peng Gu, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. 2016. PRIME: A Novel Processing-In-Memory Architecture for Neural Network Computation in ReRAM-based Main Memory. In *Proceedings of ISCA-43*.

[7] Charles Eckert, Xiaowei Wang, Jingcheng Wang, Arun Subramaniyan, Ravi Iyer, Dennis Sylvester, David Blaauw, and Reetuparna Das. 2018. Neural Cache: Bit-Serial In-Cache Acceleration of Deep Neural Networks. In *Proceedings of ISCA-45*.

[8] Patrick Foley, Abirami Prabhakaran, Karthik Gururaj, Mishali Naik, Shiva Gopalan, Aleksandr Shargorodskiy, and Ernesto Brau. 2017. Accelerate Genomics Research with the Broad-Intel Genomics Stack. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/accelerate-genomics-research-with-the-broad-intel-genomics-stack-paper.pdf.

[9] Daichi Fujiki, Aran Subramaniyan, Tianjun Zhang, Yu Zeng, Reetuparna Das, David Blaauw, and Satish Narayanasamy. 2018. GenAx: A Genome Sequencing Accelerator. In *Proceedings of ISCA-45*.

[10] Genetics Science Learning Center. 2017. Your Doctor's New Genetic Tools. http://learn.genetics.utah.edu/content/precision/example/.

[11] Sara Goodwin, James Gurtowski, Scott Ethe-Sayers, Panchajanya Deshpande, Michael C Schatz, and W Richard McCombie. 2015. Oxford Nanopore Sequencing, Hybrid Error Correction, and De Novo Assembly of a Eukaryotic Genome. *Genome Research - 25* (2015).

[12] Ernst Joachim Houtgast, Vlad-Mihai Sima, Koen Bertels, and Zaid Al-Ars. 2015. An FPGA-Based Systolic Array to Accelerate the BWA-MEM Genomic Mapping Algorithm. In *International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS-15)*.

[13] Miten Jain, Sergey Koren, Karen H. Miga, Josh Quick, Arthur C. Rand, Thomas A. Sasani, John R. Tyson, Andrew D. Beggs, Alexander T. Dilthey, Ian T. Fiddes, et al. 2018. Nanopore Sequencing and Assembly of a Human Genome with Ultra-Long Reads. *Nature Biotechnology* 36 (2018).

[14] Supreet Jeloka, Naveen Bharathwaj Akesh, Dennis Sylvester, and David Blaauw. 2016. A 28 nm Configurable Memory (TCAM/BCAM/SRAM) Using Push-Rule 6T Bit Cell Enabling Logic-in-Memory. *IEEE Journal of Solid-State Circuits* 51 (2016).

[15] Muin J. Khoury. 2016. Cancer Precision Medicine: More Population Sciences Ahead! https://blogs.cdc.gov/genomics/2016/01/20/cancer-precision-ahead/.

[16] Hyeonuk Kim, Jaehyeong Sim, Yeongjae Choi, and Lee-Sup Kim. 2019. NAND-Net: Minimizing Computational Complexity of In-Memory Processing for Binary Neural Networks. In *Proceedings of HPCA-25*.

[17] Heng Li. 2013. Aligning Sequence Reads, Clone Sequences and Assembly Contigs with BWA-MEM. *arXiv preprint arXiv:1303.3997* (2013).

[18] Shuangchen Li, Dimin Niu, Krishna T. Malladi, Hongzhong Zheng, Bob Brennan, and Yuan Xie. 2017. DRISA: A DRAM-based Reconfigurable In-Situ Accelerator. In *Proceedings of MICRO-50*.

[19] Advait Madhavan, Timothy Sherwood, and Dmitri Strukov. 2014. Race Logic: A Hardware Acceleration for Dynamic Programming Algorithms. In *Proceedings of ISCA-41*.

[20] Matt Massie, Frank Nothaft, Christopher Hartl, Christos Kozanitis, Andre Schumacher, Anthony D. Joseph, and David A. Patterson. 2013. ADAM: Genomics Formats and Processing Patterns for Cloud Scale Computing. *University of California, Berkeley Technical Report, No. UCB/EECS-2013* 207 (2013).

[21] Matthew Might and Matt Wilsey. 2014. The Shifting Model in Clinical Diagnostics: How Next-Generation Sequencing and Families are Altering the Way Rare Diseases are Discovered, Studied, and Treated. *Genetics in Medicine* 16 (2014).

[22] Neil A. Miller, Emily G. Farrow, Margaret Gibson, Laurel K. Willig, Greyson Twist, Byunggil Yoo, Tyler Marrs, Shane Corder, Lisa Krivohlavek, Adam Walter, et al. 2015. A 26-Hour System of Highly Sensitive Whole Genome Sequencing for Emergency Management of Genetic Diseases. *Genome Medicine* 7 (2015).

[23] Naveen Muralimanohar et al. 2007. *CACTI 6.0: A Tool to Understand Large Caches.* Technical Report. University of Utah.

[24] National Cancer Institute. 2017. The Genetics of Cancer. https://www.cancer.gov/about-cancer/causes-prevention/genetics.

[25] Anna C. Need, Vandana Shashi, Yuki Hitomi, Kelly Schoch, Kevin V. Shianna, Marie T. McDonald, Miriam H. Meisler, and David B. Goldstein. 2012. Clinical Application of Exome Sequencing in Undiagnosed Genetic Conditions. *Journal of Medical Genetics* 49 (2012).

[26] Yukiteru Ono, Kiyoshi Asai, and Michiaki Hamada. 2013. PBSIM: PacBio Reads Simulator Toward Accurate Genome Assembly. *Bioinformatics* 29 (2013).

[27] Jih-Kwon Peir, Shih-Chang Lai, Shih-Lien Lu, Jared Stark, and Konrad Lai. 2002. Bloom Filtering Cache Misses for Accurate Data Speculation and Prefetching. In *Proceedings of International Conference on Supercomputing (ICS-16).*

[28] Abirami Prabhakaran, Beri Shifaw, Mishali Naik, Paolo Narvaez, Geraldine Van der Auwera, George Powley, Serge Osokin, and Ganapati Srinivasa. 2016. Infrastructure for Deploying GATK Best Practices Pipeline. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/deploying-gatk-best-practices-paper.pdf.

[29] Tamara Schmitz. 2015. The Rise of Serial Memory and the Future of DDR. Xilinx White Paper (456) https://www.xilinx.com/support/documentation/white_papers/wp456-DDR-serial-mem.pdf.

[30] Vivek Seshadri, Donghyuk Lee, Thomas Mullins, Hasan Hassan, Amirali Boroumand, Jeremie Kim, Michael A. Kozuch, Onur Mutlu, Phillip B. Gibbons, and Todd C. Mowry. 2017. Ambit: In-Memory Accelerator for Bulk Bitwise Operations Using Commodity DRAM Technology. In *Proceedings of MICRO-50.*

[31] A. Shafiee, A. Nag, N. Muralimanohar, R. Balasubramonian, J. Strachan, M. Hu, R.S. Williams, and V. Srikumar. 2016. ISAAC: A Convolutional Neural Network Accelerator with In-Situ Analog Arithmetic in Crossbars. In *Proceedings of ISCA.*

[32] Yakun Sophia Shao, Brandon Reagen, Gu-Yeon Wei, and David Brooks. 2014. Aladdin: A Pre-RTL, Power-performance Accelerator Simulator Enabling Large Design Space Exploration of Customized Architectures. In *Proceeding of ISCA-41.*

[33] Temple F. Smith, Michael S. Waterman, et al. 1981. Identification of Common Molecular Subsequences. *Journal of Molecular Biology* 147 (1981).

[34] Prakalp Srivastava, Mingu Kang, Sujan K. Gonugondla, Sungmin Lim, Jungwook Choi, Vikram Adve, Nam Sung Kim, and Naresh Shanbhag. 2018. PROMISE: An End-to-End Design of a Programmable Mixed-Signal Accelerator for Machine-Learning Algorithms. In *Proceedings of ISCA-45.*

[35] Yatish Turakhia, Kevin Jie Zheng, Gill Bejerano, and William J. Dally. 2018. Darwin: A Hardware-Acceleration Framework for Genomic Sequence Alignment. In *Proceedings of ASPLOS-23.*

[36] Kris A. Wetterstrand. 2017. DNA Sequencing Costs: Data from the NHGRI Genome Sequencing Program (GSP). http://www.genome.gov/sequencingcostsdata.

[37] Lisa Wu, David Bruns-Smith, Frank A Nothaft, Qijing Huang, Sagar Karandikar, Johnny Le, Andrew Lin, Howard Mao, Brendan Sweeney, Krste Asanović, et al. 2019. FPGA Accelerated INDEL Realignment in the Cloud. In *Proceedings of HPCA-25.*

[38] Hongyi Xin, John Greth, John Emmons, Gennady Pekhimenko, Carl Kingsford, Can Alkan, and Onur Mutlu. 2015. Shifted Hamming Distance: A Fast and Accurate SIMD-Friendly Filter to Accelerate Alignment Verification in Read Mapping. *Bioinformatics* 31 (2015).

[39] Matei Zaharia, William J. Bolosky, Kristal Curtis, Armando Fox, David Patterson, Scott Shenker, Ion Stoica, Richard M. Karp, and Taylor Sittler. 2011. Faster and More Accurate Sequence Alignment with SNAP. *arXiv preprint arXiv:1111.5572* (2011).