GenomicsBench: A Benchmark Suite for Genomics*

Arun Subramaniyan  
University of Michigan  
Ann Arbor, MI, USA  
arunsub@umich.edu

Yufeng Gu  
University of Michigan  
Ann Arbor, MI, USA  
yufenggu@umich.edu

Timothy Dunn  
University of Michigan  
Ann Arbor, MI, USA  
timdunn@umich.edu

Somnath Paul  
Intel Corporation  
Hilsboro, OR, USA  
somnath.paul@intel.com

Md Vasimuddin  
Intel Corporation  
Bangalore, KA, India  
vasimuddin.md@intel.com

Sanchit Misra  
Intel Corporation  
Bangalore, KA, India  
sanchit.misra@intel.com

David Blaauw  
University of Michigan  
Ann Arbor, MI, USA  
blaauw@umich.edu

Satish Narayanasamy  
University of Michigan  
Ann Arbor, MI, USA  /nsatish@umich.edu

Reetuparna Das  
University of Michigan  
Ann Arbor, MI, USA  
reetudas@umich.edu

Abstract—Over the last decade, advances in high-throughput sequencing and the availability of portable sequencers have enabled fast and cheap access to genetic data. For a given sample, sequencers typically output fragments of the DNA in the sample. Depending on the sequencing technology, the fragments range from a length of 150-250 at high accuracy to lengths in few tens of thousands but at much lower accuracy. Sequencing data is now being produced at a rate that far outpaces Moore’s law and poses significant computational challenges on commodity hardware. To meet this demand, software tools have been extensively redesigned and new algorithms and custom hardware have been developed to deal with the diversity in sequencing data. However, a standard set of benchmarks that captures the diverse behaviors of these recent algorithms and can facilitate future architectural exploration is lacking. To that end, we present the GenomicsBench benchmark suite which contains 12 computationally intensive data-parallel kernels drawn from popular bioinformatics software tools. It covers the major steps in short and long-read genome sequence analysis pipelines such as basecalling, sequence mapping, de-novo assembly, variant calling and polishing. We observe that while these genomics kernels have abundant data level parallelism, it is often hard to exploit on commodity processors because of input-dependent irregularities. We also perform a detailed microarchitectural characterization of these kernels and identify their bottlenecks. GenomicsBench includes parallel versions of the source code with CPU and GPU implementations as applicable along with representative input datasets of two sizes - small and large.

Index Terms—Genomics, Bioinformatics, Benchmarking, Computer Architecture.

I. INTRODUCTION

Genomics is at the forefront of the precision medicine revolution. Genome sequencing can help in early cancer detection [1], developing targeted therapies to different tumor mutations [2], identifying the causes of complex genetic diseases [3], assessing risk factors, and developing new drugs. For example, 42% of the drugs approved by FDA in 2018 were based on precision medicine data obtained from genome sequencing [4]. With the advent of portable and cheap sequencers, it is now feasible to test and monitor the emergence of novel infectious diseases such as COVID-19 [5] among our population and take timely action to prevent their spread.

A genome is a long string of DNA bases or nucleotides (A, C, G and T). The human genome, for example, contains ~6 billion bases, ~3 billion bases per DNA strand. Genome sequencing refers to the process of determining the sequence of bases (i.e., A, C, G and T) in an individual’s DNA. Genome sequencers typically read DNA by fragmenting it into billions of short substrings (called “reads”).

Genome sequencing technology is far outpacing Moore’s law in computing. Over the last decade, they have become increasingly cheaper, faster, more portable, and produce longer reads. The cost to sequence a human genome has dropped from $10 million, a decade ago, to less than $1000 today. Sequencing providers like Illumina can sequence a human genome for $600 [6] and BGI/MGI [7] has further reduced the cost to $100. Apart from the dramatic reductions in cost, there has also been a corresponding increase in sequencing machine throughput. For example, MGI’s DNBESEQ-TX and Illumina’s Novaseq 6000 produce 20 Terabases [8] and 3.3 Terabases per day respectively [9]. In addition, sequencing no longer requires large bench-top instruments. Oxford Nanopore has introduced the portable MinION sequencer which can produce longer reads (few Kilobases to Megabases) in real-time, although with a higher error rate (5-15%). These portable sequencers also enable a kind of software-defined sequencing paradigm by exposing interfaces to control the length of DNA in real-time as it passes through the pore [10]. Taken together, all these developments have given rise to widespread usage of genome sequencing and ushered in the era of population genomics with several countries/organizations aiming to sequence the genomes of millions of humans [11]–[13]. However, computing solutions, hampered by challenges in scaling transistors, have not been keeping pace.

In this paper, we identify commonly used modern sequencing pipelines, characterize their performance, and
extract their compute-intensive kernels. The goal is to compile a standardized genome sequencing benchmark suite that highlights the growing compute need in genomics and helps shape future computing research in this space. Such an effort has been lacking for this important computational domain. Some notable prior works that perform detailed architecture characterization of important bioinformatics workloads such as BioPerf [14], BioBench [15] and MineBench [16] were carried out in the last decade when sequencing technologies were still nascent and not so diverse. Modern sequencing pipelines have vastly different bandwidths, latencies, portability requirements, algorithms, and pipelines than those used a decade ago. For instance, new kernels that leverage vectorized implementations for dynamic programming are now common. Machine learning algorithms are now widely used to process long but noisy reads. There is a wide variety of sequencers that vary in terms of throughput, read length, and accuracy, to meet different medical research and clinical needs. These have resulted in a plethora of bioinformatics tools and pipelines. Without a standardized benchmark suite that represents common computational kernels, it becomes increasingly difficult to design efficient computing system and processor architectures for this rapidly emerging domain. There is also growing interest in developing custom hardware solutions for sequencing [17]. These efforts can also greatly benefit from the availability of a genomics benchmark suite.

To this end, we present the GenomicsBench benchmark suite which covers the three key classes of sequencing-based analysis: reference-guided assembly, de-novo assembly and metagenomics. To select benchmarks, we identify the most popular and well-maintained software tools used for different steps in these applications. We then extracted the time-consuming kernels in these tools and analyzed their performance characteristics. We observe that many of these kernels have a high degree of data-parallelism. But they are irregular, making it challenging for GPUs to exploit them. This motivates the need for newer architectures to exploit irregular data-parallelism, or newer vectorization friendly algorithms for these computational tasks.

To summarize, this paper makes the following contributions:

- We present the GenomicsBench benchmark suite consisting of 12 representative kernels spanning the major steps in short and long-read sequence analysis pipelines such as basecalling, sequence alignment, de-novo assembly, variant calling, and polishing.
- We perform a detailed analysis of the available parallelism in these benchmarks and observe that while these benchmarks have abundant data-parallelism, it cannot be easily exploited on commodity hardware due to significant irregularity.
- We perform a detailed characterization of the microarchitectural performance bottlenecks, memory access characteristics and thread scaling behavior of these benchmarks.
- We will open-source both the benchmark suite and input datasets for the benefit of the broader research community.

II. BACKGROUND AND METHODOLOGY

A. Common Genomics Pipelines

In this section we describe some common genomics pipelines to analyze short and long read sequencing data (also illustrated in Figure 1). All three pipelines start with the raw sequencer output. Given a biological sample, typically, multiple copies of the contained genome sequence are extracted and then decomposed into smaller nucleotide fragments. A sequencer reads the sequence of nucleotides in
the fragments and generates raw signals based on what it reads. The first step in all the three pipelines prior to downstream analyses is the interpretation of these signals to derive reads, which are sequences of bases over the nucleotide alphabet \{A,C,G,T\}. This step is called basecalling. For Illumina sequencing machines, the signal data are fluorescence images which are converted into bases using a proprietary basecaller, Bustard [18]. For Oxford Nanopore (ONT) sequencers, raw signals are the current perturbations in the nanopore (e.g., in fast5 format). Guppy [19] is ONT’s proprietary basecaller software. We characterize the open-source research basecaller from ONT, Bonito [20] as part of the nn-base kernel (Section III), which demonstrates higher basecalling accuracy than Guppy [21].

Reference Guided Assembly: This pipeline reconstructs the sample genome by aligning reads from it to a reference genome and identifies differences in the sample (also called variants) compared to the reference genome. Typically, small differences, i.e., substitutions, short insertions and deletions (< 50 bases) are identified. Sufficient number of copies of the sample genome need to be sequenced to ensure random sequencing errors can be distinguished from true variations (each genome position is covered 30 – 50x on average). This is especially needed for long reads from PacBio and ONT which have 5 – 15% error rate per base [19], [22], resulting in input datasets of several hundreds of gigabytes. Subsequent analysis of this data can take several days on a modern multicore processor [23]. Figure 1 a. shows the two main time-consuming steps: (1) Read Alignment, which determines the best location for each read in the reference genome. (2) Variant Calling, which uses machine learning or statistics-based models to gather support for variants from aligned reads. BWA-MEM [24], [25] and GATK Haplotype Caller [26] are the most popular short-read software tools for these two steps recommended as part of GATK Best Practices [27]. These account for ∼30-40% and ∼40% time of the reference guided assembly pipeline respectively [25], [28]. We select the fmi, bsw, phmm, nn-variant and dbg kernels from this pipeline (Section III).

De Novo Assembly: This pipeline attempts to assemble the reads into a genome de novo based on read overlaps in the absence of a suitable reference. The availability of long reads for de novo assembly has greatly improved the quality of draft reference genomes. This is mainly because they can span large structural variations (e.g., > 50 bases insertions/deletions, large rearrangements) [29] and can help resolve mutations from maternal and paternal chromosomes [30]. Long read de novo assembly is typically done using the overlap-layout-consensus method as shown in Figure 1 b. In the overlap identification step, common seeds shared between read pairs are used to identify potential overlapping regions. In the layout step, these overlapping regions are extended into larger contiguous regions. Finally the consensus step corrects small errors in assembly. Large assembly errors are corrected in a later graph-based polishing step. For long-read assembly and polishing, Flye [31] and Racon [32] are popular software tools. Assembly of the human genome using Flye [31] and Racon [32] takes ∼4.5 days on a 64-thread server, each contributing ∼30% to the overall time [23]. Basecalling is performed using Guppy [19], ONT’s proprietary basecaller, and also accounts for ∼30% of overall time [23]. We select the chain, spoa, kmer-cnt kernels from this pipeline (Section III).

Metagenomics Classification: The advent of portable sequencers like ONT MinION [33] has enabled several applications like real-time pathogen detection [34] and microbial abundance estimation [35] in the field. Abundance estimation involves aligning input microbial reads to a reference pan-genome (consisting of reference genomes of all bacteria, virus, fungi and humans) and later estimating the proportion of different microbes in the sample as shown in Figure 1 c. It is typically performed using software tools like Minimap2 [36] and Centrifuge [37].

III. GENOMICSBENCH BENCHMARK SUITE

FM-Index Search (fmi): The FM-index (Full-Text Index in Minute Space) is one of most common data-structures in aligners such as Bowtie2 [38], BWA-MEM [24], [25], SOAP3-dp [39] and metagenomics classification tools such as Centrifuge [37]. It is used to identify the locations of short matching substrings of the read (called seeds) in the reference genome. The FM-index is attractive because of its low memory footprint, ability to match substrings of any length and support for inexact matching (i.e., identifying seeds with a small number of edits with respect to the reference).

Figure 2 a. shows the FM-index constructed for a sample reference (R) and an example search query from the read. The FM-index consists of: (1) the suffix array (SA), which contains the locations of lexicographically sorted suffixes of the reference genome R, (2) the Burrows Wheeler Transform (BWT), computed as the last column of the sorted suffix array of the reference, (3) the count table (C) which stores the number of characters in R lexicographically smaller than a given character c and (4) the occurrence table (Occ) which stores the number of occurrences of a character up to a certain index in the BWT array.

The FM-index allows the backward search of a query of length (|Q|) in $O(|Q|)$ iterations, with at most 2 memory lookups per iteration (one each for computing the start and end (s, e) intervals of the match). It is characterized by irregular memory accesses to the large Occ table (blue arrows in Figure 2 a.) and is both memory-latency and memory-bandwidth bound. Since the memory access characteristics of FM-index search are similar across different tools, we choose the optimized super-maximal exact match (SME) search computation in BWA-MEM2 [25] in our benchmark suite. SME computation uses the FM-index to find the longest exact match spanning a given position in the read.

Input Datasets: We provide small and large datasets, which are a set of 1M and 10M human reads respectively, each 151 bases long, from sample SRR7733443 [25].
Banded Smith-Waterman (b-sw): The Smith-Waterman algorithm [40] is a dynamic-programming algorithm that estimates the pairwise similarity between pairs of sequences $X$ and $Y$ with lengths $m$ and $n$ respectively in $O(mn)$ time and space. It is commonly used in sequence alignment tools like BWA-MEM [24] and variant calling tools like GATK. The waterman algorithm.

Equation 1 shows the recurrence relation for the Smith-Waterman algorithm.

$$
H_{ij} = \max\{H_{i-1,j-1} + s(i,j), E_{ij}, F_{ij}\}
$$

$$
E_{i+1,j} = \max\{H_{ij} - q, E_{ij}\} - e
$$

$$
F_{i,j+1} = \max\{H_{ij} - q, F_{ij}\} - e
$$

Fig. 2. Selected benchmark kernels

**De-Bruijn Graph Construction (dbg):** Prior to calling variants using the reads aligned to a region of the reference genome (e.g., ~100-1000 bases), it is necessary to correct read alignment artifacts. Modern variant callers like GATK Haplotype Caller [26], [27] and Platypus [47] do this by re-assembling those reads into a De-Bruijn graph and later traversing this graph to generate strings that are likely to contain variants (called haplotypes). The graph is constructed from both the k-mers of the read and the reference as shown in Figure 2.c. Each node in the graph represents a unique k-mer and each edge links adjacent k-mers in the input. A hash table is used to track nodes that have already been inserted into the graph. If cycles are found in the graph, graph construction is repeated by increasing the k-mer size. Each input task to this kernel is a set of reads aligned to a reference region. The re-assembly tasks can be parallelized across different regions. We model the De-Bruijn graph construction implementation in the Platypus variant caller [47] for the benchmark suite that accounts for >60% of its runtime.
Input Datasets: We use BWA-MEM aligned records from the Platinum Genomes dataset [48]. Our small dataset uses a region of chromosome 22 (bases 16M-16.5M) while the large dataset uses the entire chromosome 22.

Pairwise Hidden Markov Model (pairHMM): Using the reads aligned to a region of the reference genome and the candidate haplotypes identified from De-Bruijn graph traversal, a pairwise alignment of each read to each candidate haplotype is performed to identify the most likely haplotypes supported by the reads. The total workload per region is $|R| \times |H|$ pairwise alignments, where $|R|$ and $|H|$ are the number of reads and haplotypes respectively. Pairwise alignment is performed using a Hidden Markov Model (HMM) and the likelihood score is computed using the following dynamic-programming recursion relations [49]:

$$
M_{ij} = (M_{i-1,j-1} + I_{i-1,j-1} + D_{i-1,j-1}) \cdot P_{ij}
$$

$$
I_{ij} = M_{i,j-1} + I_{i-1,j} + \epsilon
$$

$$
D_{ij} = M_{i-1,j} + D_{i,j-1} + \eta
$$

where: $M_{ij}$, $I_{ij}$ and $D_{ij}$ represent match, insertion and deletion probabilities for aligning read substring $R[i,j]$ to haplotype substring $H[i,j]$, where $0 \leq i \leq |R|$ and $0 \leq j \leq |H|$. These are weighted by different transition and emission parameters of the HMM: $\theta$, $\kappa$, $\lambda$, $\tau$, $\epsilon$, $\zeta$, $\eta$. $P_{ij}$ is the prior probability of emitting bases $\{R[i], H[j]\}$, computed using the floating point base-quality scores for the read $R$. Base-quality scores are typically provided by the basecaller and indicate the confidence of the basecaller in calling each base in the read. Low quality bases from the read contribute a smaller amount to likelihood score computed above. The computation in pairHMM differs from the Smith-Waterman kernel described earlier mainly in the use of floating-point computation. There exists abundant intra- and inter-task parallelism in this workload. Intra-task parallelism arises from data-parallel processing of cells along the wavefront as shown in Figure 2 d. Inter-task parallelism arises by parallel processing of different genome regions. We use the optimized SIMD implementation in GATK Haplotype Caller [26] as part of the benchmark suite and extend it to leverage inter-task parallelism using multiple CPU threads.

Input Datasets: We use the read-haplotype pair inputs to the calcLikelihoodScore function in GATK Haplotype Caller [26]. Our small dataset uses as input BWA-MEM aligned reads for region chromosome 22:16M-16.5M, while the large dataset uses reads aligned to the entire chromosome 22.

Chaining (chain): One of the most time-consuming steps in de novo assembly of long reads is overlap estimation between reads [31], [42]. We characterize the chaining implementation from Minimap2 [36] which is one of the most popular tools for estimating pairwise overlap between reads and extend it to support inter-task parallelism across different pairs of reads. Given a set of seeds (also called anchors) shared between a pair of reads, chaining aims to group together a set of co-linear seeds into a single overlapping region as shown in Figure 2 e. The chaining algorithm is a 1D dynamic programming based algorithm that compares each anchor with $N$ previous anchors (default = 25) to determine its best parent. The recurrence relation used to estimate the maximal chaining score of the $i^{th}$ anchor [36], [50] is:

$$
score(i) = \max \{ \text{score}(j) + \alpha(j, i) - \beta(j, i), w_i \}
$$

where $w_i$ is the length of anchor $i$, $\alpha(j, i)$ is the number of matching bases between anchors $i$ and $j$ after accounting for overlaps between them and $\beta(j, i)$ is a penalty that is set based on the relative distance between a pair of anchors on the two reads.

Input Datasets: Our input dataset uses the anchors for 1K and 10K reads from the Pachio sequence data for the C.elegans worm [50], [51] when computing overlaps with itself.

Partial-Order Alignment (poa): After assembling the reference genome of a new species, it is common to perform a polishing step to correct small errors in assembly using the aligned reads. Racon [52] is one of the most popular tools for long-read polishing. Given a set of reads aligned to the target genome, Racon first splits the reads into non-overlapping windows called chunks (which can be processed in parallel) and then incrementally constructs a partial-order graph [53] by aligning new sequences to it using a SIMD accelerated dynamic programming algorithm (see Figure 2 f). Later, the consensus sequence is generated from the graph using the heaviest bundle algorithm [54]. Each node in the partial-order graph represents a base of the input sequence and weighted edges represent support from different reads in the chunk. Since the nodes in multiple branches of the graph cannot be ordered relative to each other, the graph is said to be partially ordered. Aligning new sequences to the graph is the most time-consuming operation in Racon and has complexity $O((2n_p + 1)n|V|)$, where $n_p$ is the average number of incoming edges to nodes in the graph, $|V|$ is the number of nodes in the graph and $n$ is the length of the read chunk. Contrast this with Smith-Waterman which has complexity $O(nm)$, with regular data-dependencies. As used in Racon, our poa benchmark builds the consensus sequence for each chunk in a separate CPU thread.

Input Datasets: We use 1000 and 6000 consensus tasks for our small and large datasets respectively. These are obtained when polishing the Flye-assembled Staphylococcus aureus genome with Minimap2-2 aligned ONT long reads [19].

Adaptive Banded Signal to Event Alignment (abea): Comparing a time-series of raw nanopore signal data to a reference genome sequence is a common task in the polishing of long-read sequencing data and detection of methylated bases (i.e., non-standard nucleotides apart from A, C, G, T, which play an important role in controlling gene expression). After segmenting the signal data into different events based on sudden changes in signal current, each event is then compared against the k-mers of the reference genome using a computationally intensive dynamic programming algorithm called adaptive banded event alignment (ABEA) [55]. ABEA is the most time-consuming kernel when performing methylation calling using
the software tool Nanopolish [56]. Event alignment is more complex than banded sequence alignment since it requires an adaptive band [57] to capture long gaps in optimal alignments especially when dealing with long and error-prone Nanopore reads. These long gaps arise because k-mers are often over-represented (up to 2 × ) by multiple events as they are sampled by the nanopore. Furthermore, event alignment uses 32-bit floating point log-likelihood computation in its scoring function and is computationally more expensive than sequence alignment. We analyze the optimized GPU implementation of ABEA [55] as part of the benchmark suite. In this heavily optimized implementation, ABEA accounts for 24.5% of total runtime.

**Input Datasets:** For ABEA, our small and large datasets use 1,000 and 10,000 raw FAST5 reads from chromosome 22 of NA12878, and the GRCh38 reference genome. This data was obtained from the publicly available “Nanopore WGS Consortium” dataset [58], [59].

**Genomic Relationship Matrix (grm):** All large-scale population genomics studies need to account for potential ancestral relationship between individuals in the study. This is done by computing a $N \times N$ matrix called Genomic Relationship Matrix (or GRM), where $N$ is the number of individuals in the study. Each element of the GRM $G_{ij}$ describes the average genetic similarity between individuals and is computed as follows:

$$G_{ij} = \frac{1}{S} \sum_{s=1}^{S} \frac{(x_{is} - 2p_s)(x_{js} - 2p_s)}{2p_s(1 - p_s)}$$

where $x_{is}$ and $x_{js}$ indicate the number of copies of the non-reference base at location $s$ for individuals $i$ and $j$ respectively and $p_s$ is expected frequency of a non-reference base at location $s$ in the population. $S$ is the total number of SNV (Single Nucleotide Variation) location markers in the reference genome. We extract the GRM kernel from the popular population genomics software PLINK2 [60]. The kernel performs dense matrix multiplication and can benefit from parallel computation of different output elements as shown in Figure 2 h.

**Input Datasets:** We compute the GRM on SNV data belonging to 2504 individuals from 1000 Genomes Project Phase 3 [60]. Our small dataset uses 194K variants from chromosome 22 and our large dataset uses 1.07M variants from chromosome 1.

**Neural Network-based Base Calling (nn-base):** When performing nanopore based genome sequencing, raw nanopore signal data must be correctly converted to a sequence of nucleotide bases through a process called basecalling discussed earlier. As DNA moves through a nanopore, it does so at a highly variable rate, and the resulting current is affected by multiple consecutive nucleotides occupying the pore (~5-10, depending on the pore chemistry). Due to the limited resolution of the ADC sampler and unavoidable background noise, there is considerable overlap between current levels measured for different 5-mers. Basecallers resolve this ambiguity in two stages. First, a deep recurrent or convolutional neural network aggregates contextual information to determine the most likely nucleotide observed at each time step. Using these probabilities, a connectionist temporal classification decoder [61] then determines the most likely sequence. The neural network is by far the most time-consuming basecalling stage. In order to make this computation regular and parallelizable, existing basecallers segment the signal and perform inference on many independent chunks, stitching the final sequence together as a post-processing step. Our benchmark includes the GPU-based CNN basecaller Bonito, which currently boasts the highest basecalling accuracy [20].

**Pileup Counting (pileup):** A common pre-processing step in long-read neural network variant callers such as Medaka [62] involves parsing of alignment data for all reads aligned to a region of the reference genome (called read pileup) and generating counts for different bases, insertions and deletions at these different pileup locations. These counts are later analyzed by the recurrent neural network to call variants. This pre-processing step is time consuming because it involves random access into the alignment record to extract and parse alignment information (represented as a CIGAR string [63]). Fortunately, the pre-processing step can leverage inter-task parallelism by distributing the processing of different 100 kilobase regions of the reference genome to different CPU threads. The benchmark suite includes the inter-task parallel version of pileup counting.

**Input Datasets:** We use the results from Minimap2 alignment of ONT reads. Our small dataset uses aligned reads to the *Staphylococcus aureus* genome [19], while the large dataset uses reads aligned to chromosome 20 of sample HG002 [64].

**Neural Network-based Variant Calling (nn-variant):** Long-read variants callers examine the read pileup for a particular genome reference position and call homozygous and heterozygous variants with respect to that reference. We chose to analyze the Clair variant caller because it outperforms competing tools in terms of both performance and accuracy for long reads [65]. As input, Clair accepts a size $33 \times 8 \times 4$ tensor. Given a particular reference position, this tensor is generated using pileup information for 16 bases flanking each side ($16 + 1 + 16 = 33$), and considering the pileup counts for...
each base (A,C,G,T) and strand (forward, reverse) individually (2 \times 4 = 8). Furthermore, 4 different encodings of the same information is used: (a) raw pileup counts, (b) support for insertions relative to (a), (c) support for deletions relative to (a), and (d) support for alternative variants or alleles relative to (a). Clair uses a series of recurrent neural networks with bidirectional long short-term memory (LSTM) units and fully-connected layers to predict a potential variant’s genotype, zygosity, and indel length of each haplotype. Refer to [65] for network details.

Input Datasets: For benchmarking Clair on long reads, we selected all raw FAST5 reads from the q13.12 region of chromosome 20 of NA12878 from the “Nanopore WGS Consortium” [58] dataset. These reads were basecalled using high-accuracy Guppy 3.6.0, and mapped using Minimap2. Our small dataset variant called the first 10,000 reference positions from this region, and our larger dataset used 500,000.

<table>
<thead>
<tr>
<th>CPU</th>
<th>Intel Xeon E3-1240 v5 3.5 GHz, AVX2, 1 socket, 8 threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>L1 &amp; L2 cache</td>
<td>4 x 32KB Inst; 4 x 32KB Data, 8-way</td>
</tr>
<tr>
<td>L1 cache</td>
<td>4 x 256KBx4, 8-way</td>
</tr>
<tr>
<td>L1 &amp; L2 cache</td>
<td>4 x 32KBx4, 16-way</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>31.79 GB/s</td>
</tr>
</tbody>
</table>

### TABLE I

Baseline System Configurations.

## IV. Performance Characterization of Benchmarks

### A. Characterization Methodology

Several of the genomic analysis tools described earlier operate on large datasets and can run for several days. To keep the study manageable, we adopt the following methodology. We first profile all software tools with Intel VTune Profiler 2020 [66] as well as manual timing instrumentation to identify the most time-consuming kernels in both single and multi-thread settings. Later, we isolate these kernels and run representative input datasets of two sizes. Kernel executions with the small inputs finish in a few minutes, while the large inputs take 5–20 minutes on a single-thread. Both the small and large inputs capture the bottlenecks in the original application and exercise the kernel in similar ways (e.g., similar proportions of different dynamic instructions and memory accesses with different strides). We use the MICA pintool [67] to compute statistics on instruction distribution. Cache miss and memory stalls are obtained using performance counter events from the hardware event-based sampling collector [68]. All kernels and inputs/outputs are extracted as-is from the original software tools. The tools already support multithreading. For ease of benchmarking, we made the following modifications to the extracted benchmarks: (1) OpenMP parallelization with dynamic scheduling was used to reliably evaluate thread scaling of the benchmark after isolation from the software tool and (2) file I/O-related driver code was added for reading inputs and writing results. GPU benchmarks are characterized using Nvidia’s Visual Profiler [69] and mprof on the Nvidia Titan Xp GPU with 12GB GDDR5x memory. Table I details our experimental machine configuration. We present characterization results for all benchmarks except nn-variant which failed to complete successfully using nvprof on both a native run as well as within a Docker container.

### B. Parallelism

#### 1) CPU benchmarks

In this section, we present a detailed characterization of the sources of parallelism in our CPU benchmarks and the challenges in exploiting them.

**Overview:** Table II presents an overview of different benchmarks and their corresponding parallelism motifs based on the taxonomy provided in [45]. bsw, phmm, chain, spoa and abea are dynamic programming based but have important differences. The key ones are: (1) type of data dependency present (e.g., 1D or 2D), (2) amount of computation needed (e.g., banded/full matrix), (3) type of matrix traversal (e.g., wavefront/row-wise) and (4) type of input (e.g., sequence/graph). Also present in the benchmark suite are kernels that manipulate hash tables and perform graph construction (dbg, spoa).

Some of the GenomicsBench benchmarks like grm and kmer-cnt have **regular compute** patterns since their inputs come in regular, pre-determined sizes. In contrast, a majority of the GenomicsBench benchmarks work on inputs with varying sizes and characteristics and have **irregular compute** patterns. Table III shows the data-parallelism granularity for each of the irregular compute GenomicsBench benchmarks and the corresponding data-parallel computation performed. Note that it is possible to reduce the data-parallelism granularity further by vectorizing each of the data-parallel computations shown in the second column of Table III. However, this comes with significant additional complexity arising from the complex data dependencies present in the benchmarks (Figure 2). To overcome this, implementations often speculate on the absence of data dependencies to achieve high performance (e.g., [70]).

This complexity can often be traded-off for abundant parallelism to be exploited across two other dimensions: (1) read-level parallelism and (2) genome region-level parallelism as shown in Table III. Since each of the benchmarks process millions to billions of reads across millions of genome regions there exists abundant data-parallelism across both these dimensions. Several software tools have adopted this approach. For example, BWA-MEM2 [25] has demonstrated significant benefits by vectorizing inter-sequence computation instead of vectorizing the cell updates for bsw.

**Challenges in exploiting data parallelism:** In spite of abundant read-level / genome region-level parallelism in the GenomicsBench benchmarks, it is difficult to exploit them effectively in different software tools. To understand why, consider the following hypothetical scenario where each data-parallel computation entity discussed in Table III is assigned to a separate vector lane. Each vector is replaced with a new batch of tasks as soon as all of the ones currently
cell updates than the scalar implementation. Note that the vectorization challenges outlined above exist not only for CPU-based software tools but also for GPUs, which also employ SIMD units to increase compute density.

Similar observations can also be made for the other irregular CPU compute benchmarks. It can be seen from Figure 4 that there exists significant variation in the amount of data-parallel computation performed by different tasks in different benchmarks. For phmm, which computes the most likely haplotype given supporting reads, certain genome regions can have up to 1000× imbalance in the computation needed when compared to the average case (as can be seen from the mean (5.2M) and maximum (4.41G) cell update values across different regions). However, it must be noted that regions with such low or high computational demand are fewer (as indicated by the lightly shaded circles). Across different benchmarks we find that the ratios of maximum to average computation per task can vary from 4.1× to 8.3×.

2) GPU benchmarks

Whereas the CPU kernels selected for this benchmark suite were diverse and often encountered challenges in exploiting data parallelism, the GPU kernels we investigated had fairly assigned to it complete. For vectorization to be efficient, all the tasks assigned to each lane must perform a similar amount of computation. Any imbalances in the computation across vector lanes can severely reduce the efficiency of vector computation and lead to control divergence. For this reason, the inputs to the bsw kernel, for example, are sorted based on sequence lengths before being assigned to SIMD lanes. However, even if input sequence lengths have been accounted for, differences in input sequence content can greatly influence the computation performed in each SIMD lane. This is because matrix computation can also be aborted early when aligning highly dissimilar sequences of similar length using bsw. As a result we find that the AVX2 16-bit inter-sequence vectorized bsw implementation in GenomicsBench performs 2.2× more

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Parallelism Granularity</th>
<th>Data-Parallel Computation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fmi</td>
<td>Read</td>
<td># OCC Table Lookups</td>
</tr>
<tr>
<td>bsw</td>
<td>Seed</td>
<td># OCC Table Lookups</td>
</tr>
<tr>
<td>dbg</td>
<td>Genome Region</td>
<td># Cell Updates</td>
</tr>
<tr>
<td>phmm</td>
<td>Genome Region</td>
<td># Hash Table Lookups</td>
</tr>
<tr>
<td>chain</td>
<td>Read</td>
<td># Input Anchors</td>
</tr>
<tr>
<td>spoa</td>
<td>Read Chunk Window</td>
<td># Cell Updates</td>
</tr>
<tr>
<td>pipeline</td>
<td>Read</td>
<td># Read Lookups</td>
</tr>
</tbody>
</table>

### PARALLELISM GRANULARITY AND DATA-PARALLEL COMPUTATION FOR IRREGULAR CPU BENCHMARKS. OTHER REGULAR COMPUTE BENCHMARKS NOT SHOWN
regular control flow and compute patterns. Predictable control flow and data accesses are a prerequisite for efficient utilization of GPU computing resources, and abea and nn-base were likely implemented on the GPU for this reason.

<table>
<thead>
<tr>
<th></th>
<th>abea</th>
<th>nn-base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Branch efficiency</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Warp efficiency</td>
<td>75.09%</td>
<td>100%</td>
</tr>
<tr>
<td>Non-predicated warp efficiency</td>
<td>70.18%</td>
<td>94.43%</td>
</tr>
<tr>
<td>SM utilization</td>
<td>70.53%</td>
<td>99.83%</td>
</tr>
<tr>
<td>Occupancy</td>
<td>31.41%</td>
<td>88.47%</td>
</tr>
</tbody>
</table>

TABLE IV

GPU KERNEL CONTROL FLOW AND COMPUTE REGULARITY

The abea and nn-base kernels both avoid branch divergence entirely, and achieve relatively high warp efficiency. This is shown in Table IV. Warp efficiency is defined as the average fraction of active threads in a warp, and “non-predicated” efficiency restricts the definition of active to threads which are not executing predicated instructions. Neural network basecallers such as Bonito break sequences of raw nanopore signal into regular chunks of 4,000 consecutive measurements and feed that data into a fixed-size neural network. Since floating point matrix multiplication is computationally intensive and involves very little control flow, nn-base is able to achieve perfect warp efficiency and nearly-complete occupancy and SM utilization. The few predicated instructions reducing overall throughput are likely due to the fact that the neural network of nn-base does not operate using filters of sizes which are integer multiples of 32, the number of threads in a warp. On the other hand, the abea kernel performs a dynamic programming matrix computation instead of matrix multiplication, it is limited by the execution and memory dependencies inherent to the structure of the computation. Furthermore, abea requires frequent synchronization between warps. As a result, the SM utilization and warp efficiency are lower.

C. Instruction Diversity

Instruction diversity characterization helps determine the complexity of functional units needed for specialized hardware. Figure 5 shows the dynamic instruction breakdown for the different GPU benchmarks. The “Other” category includes string, system call, prefetching, and synchronization instructions.

![Breakdown of dynamic instructions in different benchmarks. grm is excluded because its multithreaded design to decompress inputs affects the accuracy of measurements from the MICA pintool.](image)

Among the benchmarks analyzed, phmm, bsw, and spoa benefit from SIMD vectorization and have a high proportion of vector computation instructions. It can be also be seen that phmm is the only CPU kernel that performs floating point computation, while the other kernels are dominated by scalar integer computation. phmm also uses single-precision floating point computation in most cases, and resorts to double-precision floating point only in rare cases when single-precision is insufficient to represent the result. bsw, phmm, and chain are compute-intensive and have a lower proportion of memory loads and stores when compared to memory-intensive benchmarks like fmi. We also looked at the common operations performed in vectorized benchmarks. For instance, bsw uses blend instructions for cell updates and band adjustment, and spoa extensively uses shift instructions to compare against cells present in a previous column or diagonal but which are part of a different SIMD vector.

D. Memory Access Characteristics

1) CPU benchmarks

In this section, we perform a detailed characterization of the memory access patterns of different GenomicsBench benchmarks.

Off-chip Data Requirements: Figure 6 shows the off-chip data requirements for different benchmarks. It can be seen that benchmarks like fmi and kmer-cnt have significantly higher off-chip data requirements, measured in DRAM bytes per kilo-instruction (BPKI) (66.8 BPKI and 484.1 BPKI respectively). For fmi and kmer-cnt the memory access bottlenecks are due to irregular memory accesses over large working sets, ~10 GB (FM-index) and ~8 GB (hash table) respectively, with little spatial or temporal locality. In kmer-cnt, there is low spatial locality because a 1-2 byte counter is updated for every 64 bytes (cache block) read from memory. Potential approaches to improve kmer-cnt performance include implementing cache-friendly hashing techniques like robin hood hashing [71], and improving temporal locality since the k-mers to be inserted into the hash table are known a priori.

In contrast, other benchmarks like spoa have modest off-chip data requirements (6.62 BPKI), while compute-intensive benchmarks like phmm have much lower data movement (0.02 BPKI) from off-chip memory.

Cache Miss Rates: Figure 8 shows the L1 and L2 cache miss rates and percentage of CPU cycles spent stalling for data. Notably in fmi and kmer-cnt, 41.5% and 69.2% of CPU cycles are spent waiting for data. While fmi uses all the bytes in a cache block when performing OCC table lookups, kmer-cnt only updates a 1-2 byte counter per LLC miss
and has poor spatial locality. Apart from these two, other benchmarks spend <20% of CPU cycles waiting for data.

2) GPU benchmarks

When accessing global memory, abea and nn-base kernels were unable to achieve peak memory bandwidth due to strided or irregular data accesses. This is shown in Table V.

<table>
<thead>
<tr>
<th></th>
<th>abea</th>
<th>nn-base</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global Load Efficiency</td>
<td>25.5 %</td>
<td>70.3 %</td>
</tr>
<tr>
<td>Global Store Efficiency</td>
<td>68.5 %</td>
<td>100 %</td>
</tr>
</tbody>
</table>

Table V: Useful proportion of GPU global memory bandwidth used.

The extent of irregularity of memory accesses in both GPU kernels is a direct artifact of the type of computation performed. For nn-base, neural network model weights and inputs can be loaded in several large accesses at the start of computation. Since Bonito’s convolutional neural network is comprised of many layers of separable convolutions, these matrix vector multiplications are not too large and can be performed in shared memory. At the end, results are written to global memory in contiguous transactions. For the abea kernel, however, there are dependencies between consecutive diagonal bands of the dynamic programming matrix which are computed. In order to calculate the matrix efficiently, the previous three rows (which the following band computation is dependent on) are stored in Shared Memory. This leaves no room to cache the reference’s k-mer current model and other frequently accessed data in Shared Memory. The resulting accesses to global memory are performed with sub-optimal efficiency due to the decreased spatial locality of data accesses.

E. Thread Scaling

Figure 7 shows the thread scaling behavior of the multi-threaded versions of the irregular CPU benchmarks. All inputs to these benchmarks are grouped into independent tasks with each task dynamically scheduled on a CPU thread using OpenMP. Almost all GenomicsBench benchmarks benefit from coarse-grained task-level parallelism. It can be seen that most of the benchmarks achieve perfect scaling (bsw, dbg, phmm and spoa), while fmi and chain achieve near-perfect scaling. kmer-cnt uses close to the peak random access memory-bandwidth on our system and does not scale well with increasing number of threads, whereas pileup suffers from random memory accesses.

F. Microarchitectural Bottleneck Analysis

Figure 9 shows the results of top-down analysis [72] of performance bottlenecks. It can be seen that memory-bound benchmarks like fmi and kmer-cnt spend 44.4% and 86.6% of their pipeline slots waiting for data. For fmi, >80% of OCC table accesses lead to opening of a new DRAM page making the accesses highly irregular. There is also little spatial or temporal locality in k-mer counting. Each update to the k-mer count table results in a last-level cache miss leading to significant memory-latency related stalls. Some of these stalls could potentially be mitigated by implementing software prefetching [71], since the k-mers to be looked up are known in advance. Compute-intensive benchmarks like bsw, chain and phmm spend >50% of their pipeline slots retiring instructions. They are bottlenecked by backend core resources because of limited number of available ports for scheduling vector and floating point instructions. grm performs CPU-friendly dense matrix multiplication and makes best use of available CPU pipeline slots (87.70% retiring). The memory-related stalls in spoa and pileup result from cache misses during incremental update of the partial-order alignment graph and random accesses to the read alignment records respectively.
V. RELATED WORK

The BioPerf benchmark suite [14] evaluates several DNA and protein sequence analysis benchmarks such as BLAST [73] and HMMER [74] and provides pre-compiled Alpha binaries with Simpoints to facilitate architectural simulation. Some of these benchmarks are further characterized in BioBench [15] and have been shown to have a high ILP. Recent updates to these benchmarks have also been proposed in BioBench2. GenomicsBench improves coverage of these prior benchmark suites by including both vectorized dynamic programming kernels (bsw, phmm, spoa) and GPU-optimized dense neural network kernels (nn-base, nn-variant). Among the GenomicsBench benchmarks, only phmm and bsw share similarities with HMMER and BLAST in performing floating point matrix computation and local alignment with Smith-Waterman respectively. Other benchmarking efforts also focus on characterizing the BioBench benchmarks on different architectural platforms [75], [76] and increasing its usability [77]. GenomicsBench leverages some of the optimized implementations for different short-read benchmarks proposed in prior work [71], and expands its scope to cover a broader diversity of sequence analysis steps and includes long-read benchmarks. Prior work [78] identifies some of the key computation kernels in secondary analysis, but with a focus on short reads. Intel’s Genomics Kernel Library (GKL) [79] also includes reference vectorized implementations of a few short-read benchmarks like phmm.

VI. CONCLUSION

In this paper, we present the GenomicsBench benchmark suite, containing 12 computationally intensive genomics kernels drawn from popular bioinformatics software tools. We perform detailed instruction level and microarchitectural analysis on these kernels to expose their performance bottlenecks. We also observe that the irregular data-parallelism in these benchmarks cannot be easily exploited by commodity hardware. GenomicsBench will be open sourced to the broader research community.

ACKNOWLEDGMENT

We thank the anonymous reviewers for their valuable feedback and suggestions which helped improve this paper.

REFERENCES

[13] “India to launch its 1st human genome cataloguing project.”


