Performance Extraction and Suitability Analysis of Multi- and Many-core Architectures for Next Generation Sequencing Secondary Analysis

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

Tony C Pan
Georgia Institute of Technology
Atlanta, GA, USA
tony.pan@gatech.edu

Kanak Mahadik*
Adobe Systems
San Jose, CA, USA
mahadik@adobe.com

George Powley
Intel Corporation
Hudson, MA, USA
george.s.powley@intel.com

Priya N. Vaidya
Intel Corporation
Hudson, MA, USA
priya.n.vaidya@intel.com

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

Srinivas Aluru
Georgia Institute of Technology
Atlanta, GA, USA
aluru@cc.gatech.edu

ABSTRACT

High-throughput next generation sequencers (NGS) can rapidly read billions of short DNA fragments, called reads, at low cost. Moreover, their throughput is increasing and cost is decreasing at rates much faster than the Moore's law. This demands commensurate acceleration for NGS secondary analysis that process the reads to identify variations between genomes. Conventional architectural improvements can at best improve performance at the rate of Moore's law even if the software tools efficiently utilize the underlying architecture. Unfortunately, most of the dozens of software products developed for this purpose fail to exploit the underlying architecture well. Therefore, to match the pace of development of the sequencers, we will need architecture that is more tailored for the computational requirements of NGS secondary analysis as well as software that uses the architecture optimally.

To this end, in this work, we study the performance characteristics of NGS secondary analysis and investigate the suitability of modern Intel Xeon and Xeon Phi processors for the same. To keep the study manageable, we rely on recent studies that attribute a majority of the run-time to a few key kernels. We present detailed optimization efforts to accelerate these kernels on the latest Intel Xeon and Xeon Phi processors with the goal of extracting maximum performance. A comparison of our optimized implementations, along with published results on GPGPU implementations, shows that our optimized implementations on the Skylake processors yield highest performance. We also present an in-depth analysis of the key kernels and identify their performance characteristics and bottlenecks to inform future architectural designs.

KEYWORDS

Next generation sequencing, Genomics, Domain insights, Performance characterization, Architectural bottleneck analysis

ACM Reference Format:
Sanchit Misra, Tony C Pan, Kanak Mahadik, George Powley, Priya N. Vaidya, Md Vasimuddin, and Srinivas Aluru. 2018. Performance Extraction and Suitability Analysis of Multi- and Many-core Architectures for Next Generation Sequencing Secondary Analysis. In *International conference on Parallel Architectures and Compilation Techniques (PACT '18), November 1–4, 2018, Limassol, Cyprus.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3243176.3243197

1 INTRODUCTION

The high throughput Next Generation Sequencers (NGS) are producing data at an enormous rate. For example, a single Illumina [1] Hiseq X 10 system can sequence nearly 18000 human genomes per year, reading short DNA fragments (called reads) of length 150 basepairs at the rate of nearly 1.6 quadrillion basepairs per year, at the low cost of under $1000 per genome [2]. In addition, the throughput of NGS is increasing and the cost is decreasing at exponential rates faster than the Moore's law [3].

This has made short read sequencing the de facto standard for all modern genomic studies and enabled many applications including sequencing of individual genomes, genomes of new species, metagenomic samples of communities of microbial organisms, single cell sequencing, and sequencing RNA samples to gather digital gene expression data. All of these typically require mapping reads to one or more reference genomes, or assembling them *de novo* based on read overlaps in the absence of a suitable reference, and then identifying variants with respect to a reference or among the

*Kanak Mahadik was a research intern at Intel when she worked on this project.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PACT '18, November 1–4, 2018, Limassol, Cyprus
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-5986-3/18/11…$15.00
https://doi.org/10.1145/3243176.3243197

samples. The suite of computational methods and software that perform these tasks is known as NGS secondary analysis.

The rapid pace of improvement of the sequencers demands commensurate speedups for NGS secondary analysis. Conventional architectural improvements can at best improve performance at the rate of Moore's law even if the software tools are well optimized for the underlying architecture. Unfortunately, most of the dozens of software products developed for this purpose fail to exploit the underlying architecture well. Therefore, to match the pace of development of the sequencers, we will need architecture that is more tailored for the computational requirements of NGS secondary analysis as well as software that uses the architecture optimally.

To this end, our aim in this work is to 1) study the performance characteristics of NGS secondary analysis, 2) extract performance from modern high-end server processors for such analysis, and 3) investigate their suitability for typical NGS workloads as well as understand architectural bottlenecks. For high-end server, we chose Intel® Xeon® server processor (multi-core CPU), as it is the most widely available for NGS workloads, as well as Intel® Xeon Phi™ processor (many-core CPU) for its very different architectural features [1]. In each case, we choose the most recent top-of-the-line high performance computing products. A study of the entire gamut of NGS secondary analysis software is unwieldy. Fortunately, most of them employ the following few key algorithmic kernels that account for 64 – 99% of the run-time of the three secondary analysis tasks [4]: Smith Waterman alignment [5], Pairwise-Hidden-Markov-Model (PairHMM) Algorithm [6], FM-index [7] based sequence search, and *k*-mer counting [8]. Thus, we focus our study on these specific kernels resulting in the following beneficial outcomes.

- We first optimize these kernels to leverage current architectural features well. This is essential to properly identify true architectural bottlenecks. For each kernel, the performance reported in this paper for each architecture is equal or better than any existing in the literature for the same architecture and the corresponding implementations will be made available to the community as open source. For each kernel, we also compare our implementations with the latest published GPGPU implementations.
- Our results demonstrate that our optimized implementations on the Skylake processors yield highest performance.
- We acquire a deep understanding of the performance characteristics of the kernels and what constitutes important and unimportant architectural features for these kernels.
- The study also provides a deep understanding of the performance bottlenecks of Intel® Xeon® and Intel® Xeon Phi™ processors for supporting NGS secondary analysis.

To the best of our knowledge, this is the first ever study of this kind.

2 OVERVIEW OF THE KEY KERNELS

Let X be a DNA sequence, modeled as a string over the alphabet Σ = {A,C,G,T}. Let $|X|$ denote its length, $X[i]$ denote the base at position i, and $X[i,j]$ ($i \leq j$) denote the substring $X[i]X[i+1]X[i+2]\cdots X[j]$.

[1]Intel, Xeon and Intel Xeon Phi are trademarks of Intel Corporation or its subsidiaries in the U.S. and/or other countries. Other names and brands may be claimed as the property of others. ©Intel Corporation

2.1 Smith-Waterman Alignment

Smith Waterman [5] is a key kernel for NGS secondary analysis, and is particularly used in several sequence mapping tools [9–15], and GATK's Haplotype Caller [16, 17]. Our use case for SW alignment consists of a set of tasks, each given by a pair of DNA sequences on which SW should be performed. This is typical of NGS secondary analysis, with numerous alignments on short sequences with lengths rarely exceeding a few hundred bases.

Smith-Waterman (SW) algorithm computes the highest scoring local alignment between sequences X and Y (Figure 1). It computes matrix H whose ij^{th} element $H_{i,j}$ (typically referred to as *cell* in this domain) is the similarity score of aligning $X[0,i]$ and $Y[0,j]$, as given by the following recurrence:

$$\begin{aligned} H_{i,j} &= max\{0, E_{i,j}, F_{i,j}, H_{i-1,j-1} + f(X[i], Y[j])\} \\ E_{i,j} &= max\{H_{i,j-1} - \alpha, E_{i,j-1} - \beta\} \\ F_{i,j} &= max\{H_{i-1,j} - \alpha, F_{i-1,j} - \beta\} \end{aligned} \quad (1)$$

where $1 \leq i \leq |X|, 1 \leq j \leq |Y|$, α and β are gap open and gap extension penalties, and $f(a,b) = \gamma$ (match score) for a = b and δ (mismatch score) otherwise. The matrices H, E and F are initialized as follows:

$$H_{i,0} = E_{i,0} = H_{0,j} = F_{0,j} = 0,\ 0 \leq i \leq |X|, 0 \leq j \leq |Y| \quad (2)$$

The output of SW alignment is the best score S in H. The algorithm performs dynamic programming to compute matrices H, E and F, and has $O(|X| \cdot |Y|)$ run-time.

2.1.1 Related work. Acceleration of SW has been extensively studied [18–33]. Optimization efforts target both intra-task and inter-task parallelism. The scope for inter-task parallelism is obvious, as the pairwise alignments are independent of each other. As for intra-task parallelism, the computation of cell (i,j) depends on three neighboring cells – one to the immediate left $(i, j-1)$, one immediately above $(i-1, j)$, and the immediate diagonal neighbor above $(i-1, j-1)$. Note that this allows parallel computation of all the cells within an anti-diagonal. Available parallelism is limited due to short lengths of the sequences, and is uneven as the length of the anti-diagonals varies.

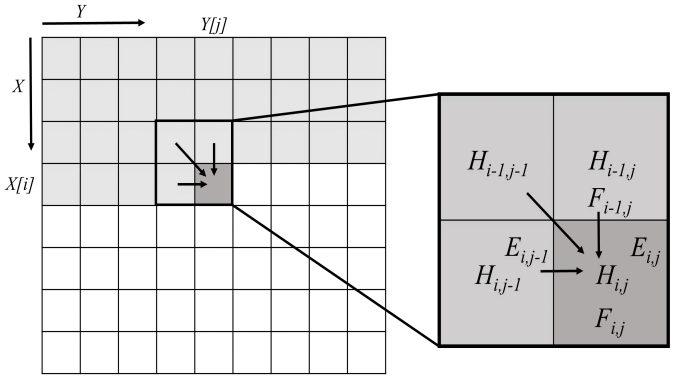


Figure 1: Smith Waterman alignment of sequences X and Y. The light gray cells have already been computed. The dark gray cell is currently being computed and its value depends on three neighboring cells (shown by the arrows). On the right, we have enlarged the current cell and the three neighbors to show the dependencies in detail.

2.2 PairHMM Algorithm

Pairwise Hidden Markov Model (PairHMM) [6] is a key component in NGS variant calling software(e.g. GATK's HaplotypeCaller [16], MuTect2 [34]) that are used to identify variations between genomes. Variant calling is executed after the reads have been mapped to the reference sequence. For regions in the genome that show signs of variation between the reads and the reference sequence, GATK variant callers stitch together overlapping reads in order to generate a set of DNA sequences, called candidate haplotypes. A pairwise alignment of each candidate haplotype and each read sequence in a region is performed using the PairHMM algorithm to obtain the likelihoods of the haplotypes given the read data. These likelihoods are used to identify the most likely haplotypes.

The output of PairHMM of a haplotype and read pair is a single floating-point probability score that represents the likelihood. Our use case inputs a batch of H candidate haplotypes and R read sequences, and calculates the PairHMM score for each of the $H \times R$ pairs. This use case exactly models the usage of PairHMM in GATK variant callers.

The PairHMM score of a pair is calculated using the following recurrence equations [35]:

$$\begin{aligned} M_{i,j} &= P_{i,j}(\theta M_{i-1,j-1} + \kappa I_{i-1,j-1} + \lambda D_{i-1,j-1}) \\ I_{i,j} &= \tau M_{i-1,j} + \epsilon I_{i-1,j} \\ D_{i,j} &= \zeta M_{i,j-1} + \eta D_{i,j-1} \end{aligned} \tag{3}$$

where M represents match, I represents insertion, and D represents deletion. $P_{i,j}$ is the prior probability of emitting two characters ($X[i]$, $Y[j]$). The constants θ, κ, λ, τ, ϵ, ζ and η are transition probabilities of the underlying HMM. The final output of PairHMM is the sum of the cells in the last row of the I and D matrices. Similar to SW, PairHMM uses dynamic programming; however, it uses floating-point operations to include read quality scores in the probability calculation.

2.2.1 Related work. Existing efforts to accelerate PairHMM include optimization for CPU, GPGPU, and FPGA [36–41]. Similar to SW, acceleration of PairHMM can exploit both intra-task parallelism and inter-task parallelism.

2.3 FM-index based Sequence Search

Full-text minute-space (FM) index [7] based DNA sequence search is the central kernel in many advanced and widely used sequence mapping tools (Bowtie [12], Bowtie2 [13], SOAP2 [11], BWA [14] and BWA-MEM [15]). These tools need to search for substrings of the reads in the reference DNA sequence, to seed identification of mapping locations. To facilitate fast searches, they create an FM-index of the reference sequence. FM-indices are generally used to perform three kinds of sequence search – 1) end-to-end exact search that finds the exact matches of the query sequence in the reference sequence, 2) end-to-end inexact search that allows a limited number of mismatches and gaps in the matches, and 2) super maximal exact match (SMEM) search that finds at each query position the longest exact match covering the position. To keep the study manageable, we focus on the first two types of searches as the compute and memory characteristics of all three types of searches are similar.

2.3.1 BWT and FM-Index data structures. The BWT, suffix array and FM-index of an example reference sequence R are shown in Figure 2. BWT is constructed by first obtaining all the rotations (Rotations(R)) of the reference sequence. All these rotations arranged in a lexicographical order constitute the BW-Matrix. The BWT is the last column of the BW-Matrix. The suffix array (SA) denotes the original positions of the first bases of the rotations in R. All occurrences of a query can be found as prefixes of the rotations in the BW-Matrix. Since the rotations are lexicographically sorted, these matches are located in contiguous rows of the BW-Matrix. Hence, all the matches of a query can be represented as a range of rows of the BW-Matrix, called the SA interval of the query. For example, in Figure 2, matches of sequence *"AG"* correspond to the SA interval $[1, 2]$.

The FM-index is used to facilitate fast search of BWT [42]. It comprises of D and Occ data structures. $D(x)$ is the number of bases in $R[0, L-1]$ that are lexicographically smaller than $x \in \Sigma$, where L is the length of R. $Occ(x, i)$ is the number of occurrences of x in $B[0, i]$, where B is the BWT of R.

2.3.2 Exact search using FM-index. Let Q denote a query sequence of length l. An occurrence of Q is a position p in R, such that

$$0 \leq p, p + l < L, \forall j \in [0, l), R[p + j] = Q[j] \tag{4}$$

Thus, an occurrence of Q is defined as the starting position of a substring of R that matches Q end-to-end with all the letters matching between Q and the substring of R. Exact search finds all such occurrences of Q in R.

Exact search based on FM-index is performed using backward search algorithm (Algorithm 1). The query is processed from the end to the beginning. For each value of i, the algorithm identifies the SA interval $[sp, ep]$ of $Q[i, l-1]$ in R. The width of this interval either shrinks or remains the same at each step. If the algorithm terminates before reaching the beginning of the query, Q does not occur in R. Otherwise, the range corresponds to the positions of exact occurrences of Q in R. The time complexity of the backward search algorithm is $O(l)$.

2.3.3 Irregular memory access in FM-index based exact search. For each base $Q[i]$, new values of *sp* and *ep* are computed by accessing two locations of the *Occ* structure. These locations depend on $Q[i]$ and the current values of *sp* and *ep*. There is no defined pattern

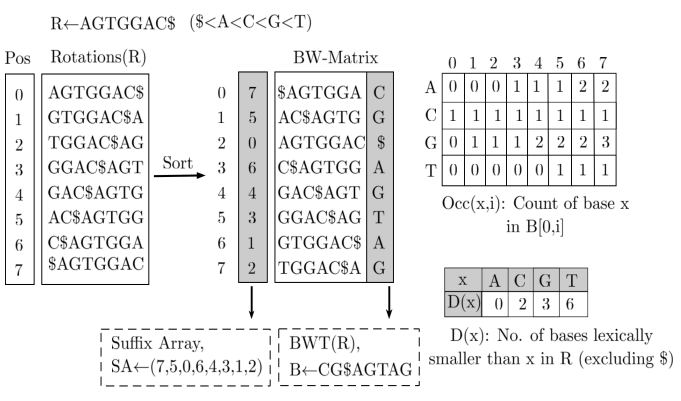


Figure 2: BWT, FM-Index and Suffix array(SA) of R. FM-Index consists of the D and Occ structures. $ denotes the end-of-sequence letter

Algorithm 1 Backward search algorithm for exact search

Input: FM-index of reference R, Query Q
Output: SA interval of Q in R, $[ep, sp]$

```
1: i ← l
2: (sp, ep) ← (1, L)
3: while sp ≤ ep AND i ≥ 1 do
4:     i ← i − 1
5:     c ← Q[i]
6:     sp ← D[c] + Occ(c, sp − 1) + 1
7:     ep ← D[c] + Occ(c, ep)
8: end while
   return [sp, ep]
```

between successive values of *sp* and *ep*. So, there is no locality between consecutive memory accesses to *Occ*. *Occ* is a huge data structure with a size of $L \times |\Sigma|$. Irregular accesses to such a data structure results in significantly high cache miss rates and average latency.

2.3.4 Compressed FM-index. The implementation of FM-index in real software tools differs from the above description. In order to reduce the memory footprint, *Occ* is divided into buckets of size h. The compressed data structure *COcc* has only L/h entries. In each entry i, two items are stored – 1) The count values of the bases up to bucket i; thus, for each base c, *COcc* will hold the counts $Occ(c, i \times h)$, and 2) The substring of BWT, $B[i \times h : (i+1) \times h]$. The *Occ* values of any position can be computed using the *COcc* count and BWT substring.

While this reduces memory footprint, there is also an additional benefit. Many accesses fall into the same *COcc* entry, improving data locality. In particular, as we move through the query from end to the beginning, the shrinking SA interval may mean that *sp* and *ep* locations in *Occ* fall in the same cache line. With compressed FM-index, this happens more frequently. The trade off is the increased runtime for processing a base from $O(1)$ to $O(h)$.

2.3.5 Inexact search using FM-index. Inexact search of a query allows for a bounded number of differences (mismatches or gaps), say z. The values of z are typically small. For example, Bowtie2 only allows a single mismatch. Inexact search utilizes the exact search algorithm, augmented by a combination of bounded traversal and backtracking over the search space. Since the algorithm has similar characteristics to the algorithm for exact search algorithm, we do not discuss it further for the sake of brevity. Interested reader can refer to [14] for more details.

2.3.6 Related work. There have been several algorithmic improvements to FM-index based algorithms and a few works considered architecture-aware acceleration of these problems on CPU, GPGPU and FPGA [43–48].

In particular, Chacon *et al.* [45, 46] developed a clever n-step FM-index strategy that allows processing n bases at a time, thereby reducing the number of memory accesses by a factor of n while increasing the number of executed instructions. They showed significant performance gains on CPU and GPGPU but their performance on CPU was still memory latency bound.

2.4 K-mer Counting

K-mer counting is an important step in many secondary analysis tasks, including NGS read error detection and correction [49–51], *de novo* assembly [52–55],sequencing coverage estimation [56], single nucleotide polymorphism (SNP) identification [49], and metagenomic sequence classification [57].

A k-mer is a length k substring of a sequence. K-mer counting refers to the task of computing the number of occurrences of each unique k-mer in a set of input sequences. As DNA sequences are double stranded, each k-mer $m = X[i]X[i+1]\cdots X[i+k-1]$ has the reverse complement $m' = c(X[(i+k-1)])\cdots c(X[i])$ where $c(\bullet)$ is the complement operation based on the base pairing $A \leftrightarrow T$ and $C \leftrightarrow G$. An application may consider a k-mer and its reverse complement as equivalent and aggregate their counts together.

2.4.1 Use case and related work. In NGS secondary analysis, k values typically range from less than 10 to just over 100, with common values falling between 25 and 55. We consider a k-mer and its reverse complement as equivalent, and define a *canonical* k-mer as the lexicographically smaller of a k-mer and its reverse complement. Given a set of reads, we study the use case of counting the number of unique canonical 31-mers.

K-mer counting has been extensively studied over the past decade [8, 58–65]. Counting is accomplished mainly through incremental updates to hash tables [8, 58, 64, 65], including hash based probabilistic data data structures [60–62] such as Bloom Filters [66] and Count-min Sketch [67], or through sorting and aggregation [59, 63].

The majority of these tools are designed for shared memory multi-core systems and adopt either inter-task or intra-task parallelism. Intra-task parallelism is achieved generally via concurrent updates of a shared data structure [8, 60], while inter-task parallelism via data partitioning followed by sequential computation for each partition. Partitioning minimizes subsequent synchronization and may occur on disk [58, 59, 63, 64], or in memory [59, 61, 63, 65]. Many-core accelerators, such as GPGPU, may also be employed [64] for compute intensive phases.

Kmerind [65] is designed for distributed memory environments, and adopts the partition-and-compute approach for inter-task parallelism. It has been shown to out-perform other k-mer counting tools in both distributed memory and shared memory environments.

3 ARCHITECTURES STUDIED

We conducted our study on the latest Xeon Scalable family of processors (Skylake) and the Xeon Phi x200 family of processors (Knights Landing). Knights Landing is the second generation of the Xeon Phi line, and the first generation to be available as a self-hosted processor. Most of the clusters, supercomputers and cloud deployments provide two sockets of Skylake processors or one socket of Knights Landing processor per node. Therefore, we compare two sockets of Skylake processors and with one of Knights Landing processor to provide node-level comparison. From here on, we will refer to the specific platforms used in this study (Table 1) as SKX and KNL, respectively.

SKX consists of 2 sockets each containing 28 cores, with 1 MB/core L2 cache, for a total of 56 cores running at the base frequency of 2.5 GHz and a total L2 cache size of 56 MB. KNL consists of 34 active tiles. Each tile contains two cores running at the base frequency

Table 1: Specific platforms studied

Intel® Xeon® Platinum 8180 Processor (SKX)	Intel® Xeon Phi™ Processor 7250 (KNL)
8.2 SP TFLOP/s	6.1 SP TFLOP/s

of 1.4 GHz and a 1 MB L2 cache shared between these cores for a total L2 cache size of 34 MB. Each core of SKX and KNL contains 64 KB L1 cache (divided into 32 KB each for instruction and data). In addition, SKX is also equipped with 77 MB of LLC cache. Each SKX core supports up to 2 hyperthreads, for a total of 112 logical cores. Each core of KNL can support up to 4 hyperthreads, giving a total of 272 logical cores.

Each processor is equipped with 2 AVX512 vector processing units (VPU) per core capable of processing multiple 32 and 64 bit integers and floating point numbers simultaneously. In addition, SKX also supports AVX512 vector processing of 8-bit and 16 bit integers thus capable of SIMD widths of 64 and 32, respectively. The lower precision integer operations can have significant performance implications on the integer-heavy NGS secondary analysis. For backward compatibility, all previous ISA extensions – SSE, AVX, and AVX2 – are supported on both SKX and KNL. For scalar operations, SKX has 4 scalar ALUs per core compared to 2 scalar ALUs per core on KNL. This, coupled with the other benefits of bigger core like better branch predictor, results in significantly better scalar performance of SKX.

SKX contains 192 GB DRAM with a peak memory bandwidth of 228 GB/s spread over two NUMA domains, one on each socket. KNL contains a 16 GB high-bandwidth MCDRAM with peak bandwidth of 350 GB/s in addition to 96 GB DRAM with 115 GB/s peak bandwidth. MCDRAM can work in three different modes, selected at boot time. The *Flat* mode treats this memory as a separate NUMA node. If the memory requirement of an application is less than 16 GB, a utility like *numactl* can be used to force all memory allocation on MCDRAM. Conversely, the programmer can allocate a particular data structure on MCDRAM using explicit function calls. In the *Cache* mode, MCDRAM acts as a large direct-mapped last level cache. This mode is useful in getting many benefits of HBM without needing code changes. In the *Hybrid* mode, part of the MCDRAM can work as cache and the remaining part as a NUMA node. While the best mode depends on the application, it is prescribed to use the *Flat* mode and allocate the critical data structures on MCDRAM and remaining on DRAM, thus effectively utilizing the combined bandwidth of MCDRAM and DRAM. For a more in-depth architectural description, the reader is referred to [68, 69].

4 OPTIMIZED IMPLEMENTATIONS

4.1 Smith Waterman Alignment

Our approach is based on the optimization efforts presented in [28] for SSE2 instruction (128 bit vectors) based architectures. Rather than parallelize a single SW alignment, we execute multiple alignments simultaneously while using the sequential approach within each. Within a task, cells of a matrix can be computed row by row. As the values in a row are only dependent on the same row and the one above it, only one row each of E, F and H needs to be maintained. We can keep overwriting these rows. This, coupled with the short sequence lengths, ensures that E, F and H stay in the lowest level cache. The only data to be read from the memory are the sequences. SW is a compute bound problem as runtime is quadratic while input size is linear.

4.1.1 Vectorization. We vectorize the computation by processing multiple pairs simultaneously, one in each vector lane. SW requires only integer computation. The maximum possible score is directly proportional to the lengths of the sequences, depending on which 8-, 16-, or 32-bit integer computation is used. As mentioned in Section 3, SKX supports AVX512 integer instructions in all three precisions, while KNL supports only 32-bit integer operations in AVX512 instruction set. We therefore prepared five implementations – AVX512 based int8 and int16 implementations for SKX; AVX2 based int8 and int16 implementations for KNL; and AVX512 based int32 implementation for SKX and KNL. Based on the lengths of the sequences, the appropriate implementation can be used. Using the minimum required precision allows for significant parallelism. For example, int8 implementation based on AVX512 can process 64 tasks in parallel. We manually vectorized the code using intrinsics. Using 12 vector operations - 6 max, 4 add, 1 cmpeq and 1 blend, we perform one cell update for all the tasks across vector lanes.

For each precision level, we pick the corresponding number of tasks at a time and process them to completion. We also converted the sequence data from AoS to SoA format to ensure we can read the sequence data for one cell update with just a vector load instruction. The lengths of pairs of sequences can vary a lot across tasks. Thus, some of the tasks being processed together across vector lanes can finish earlier than others, leaving the vector lanes idle. This has been dealt with in the past by either replacing the completed task with a new task in the vector lane [28] or by sorting the tasks according to the sequence lengths [24, 26, 29]. In our experiments we found that the overhead of replacing a completed task with a new one is much higher now with wide vector registers. To avoid this, we resort to sorting.

4.1.2 Sorting. Our goal is to ensure relatively similar lengths among tasks processed together across vector lanes. Hence, we partition tasks into batches, each of which is individually sorted by length using radix sort to take advantage of short length range. This ensures the intermediate data for radix sort fits in cache and a batch can be sorted with just one iteration of the bucketing step of radix sort. This also enables easy distribution of sorting work across threads.

4.1.3 Multithreading. We parallelize SW algorithm by dynamically distributing batches of tasks across threads using *OpenMP*. Each thread sorts the batches assigned, converts the sequence data from AoS to SoA format, and then performs SW.

4.2 PairHMM

PairHMM shares many performance characteristics with SW. It is also compute bound, requiring three matrices to be populated with similar data dependencies as SW. The key difference is that PairHMM uses floating point operations. For this study, we used our optimized implementation that is also a part of the Genomics Kernel Library (GKL) by Intel [37].

The PairHMM computation of one pair of sequences is called a task. Our implementation uses intra-task parallelism for vectorization by processing the matrices in the order of the anti-diagonals. It uses single precision computation first. If the result is below a certain threshold (implying a loss of precision), it recomputes using double precision. In most cases, single precision is good enough. The code is hand-vectorized using AVX512 vector intrinsics for floating point *multiply* and *add*; and integer *and*, *or* and *permute*. The work is multithreaded by dynamically distributing tasks across threads.

4.3 FM-index Based Sequence Search

Given lower memory foot print and better data locality, we build our implementation using compressed FM-index. We identify the following strategies to optimize the performance of exact and inexact search.

4.3.1 Software prefetching. Due to irregular access to *COcc*, hardware prefetching will be ineffective. However, note that the computation of each base of a query results in *sp* and *ep* values which decide the memory locations of next access to *COcc*. We can prefetch this location using *software prefetching*. However, software prefetching will only be useful if we can hide the latency of prefetching by computation. Hence we batch the reads together. We process the reads in a batch in a round robin fashion. Every time a read gets its turn, we process one base of the read to get the *sp* and *ep* values for the next access to *COcc* for the read. With these *sp* and *ep* values, we start the software prefetching of the corresponding locations of *COcc* so that it is available in cache by the time the read gets its turn again. Given that batch size is equal to our prefetch look ahead distance, we empirically find the best batch size such that we do not prefetch too early or too late. Moreover, we store each entry of *COcc* in a 64-byte cache aligned location. We also ensure that an entry of *COcc* fits in one cache line. Note that, if we were using the *Occ* data structure, we would still need to load one cache line for every *Occ* access. So, by ensuring that an entry of *COcc* fits in one cache line, we make sure that the amount of data to be read for each base is the same while achieving better data locality; and hence, lower memory bandwidth requirement.

4.3.2 Vectorization. We perform the computation of *Occ* count values from *COcc* count values and the corresponding BWT substring using vectorization. In a *COcc* entry, 16 bytes are consumed by the 4 unsigned integers that store the count values. That leaves a maximum of 48 bytes for the BWT substring. In order to get $Occ(c, i)$, the corresponding entry of *COcc* is present at index i/h. Since division operation is quite expensive, we use a power-of-2 value for h to perform division by shift operation. Thus, we can only use 32 bytes for the BWT substring. We could use 2 bits to represent each base, but extracting the count of a particular base from a 2-bit representation requires significant bit manipulation making it expensive. Therefore, we opt for a one byte representation of bases thus choosing the value of h as 32. We perform the byte level compare using AVX2 to get a 32-bit mask containing 1 for match and 0 for mismatch. Consequently, we use a 32-bit popcnt instruction on the mask to get the count.

4.3.3 Multithreading. We parallelize the search using *OpenMP* by distributing batches of queries equally among the threads.

4.3.4 Handling reads with no matches. For a query with no matches, the backward search algorithm could terminate before the entire query is processed. This effectively makes the corresponding batch smaller resulting in less effective software prefetching. To handle this, we replace any such queries with a new query in the batch. Thus, each thread starts with the first batch of queries out of its assigned quota and keeps replacing any query that is finished, until all the queries from its quota are over.

4.3.5 Positioning of data in memory. SKX has two NUMA domains, one for each socket. To prevent the extra latency of a socket accessing data in the NUMA domain of other socket, we create two copies of the FM-index and the query sets, one for each NUMA domain. On KNL, we use MCDRAM in *Flat* mode and explicitly allocate the FM-index on MCDRAM.

4.4 K-mer Counting

Our optimization efforts focus on improving the memory access and hashing performance of Kmerind. Kmerind's MPI based distributed memory k-mer counting algorithm partitions the hash table across MPI ranks. The algorithm proceeds in 4 steps.

Partition: On each MPI rank, the k-mers are assigned to remote ranks using a hash function, and then rearranged using radix sort based on the assigned rank. The hash function computation is compute bound while the radix sort is latency bound.
Alltoallv: The rearranged k-mers are sent to the corresponding remote ranks via MPI_Alltoallv collective communication.
Count: The received k-mers are inserted into the local hash table and counts of the unique k-mers is obtained by traversing the local hash table.
Return: The counts are communicated back to the source ranks.

The optimized hash functions apply to the *Partition* and *Count* steps, while the optimized hash table is used in the *Count* step. We note that Kmerind's partition-and-compute approach is inherently NUMA friendly and requires only coarse grain inter-task synchronization and data movement via MPI communication. MPI implementations commonly utilize memory-to-memory copy for intra-node communications, thus no network communication is incurred on shared memory systems.

4.4.1 Vectorization. K-mer counting requires hashing a large number of relatively short, fixed-length k-mers. As k is chosen based on application needs, string rather than integer hash functions are more appropriate for k-mers. However, string hash functions such as Google Farm Hash and Murmur Hash, while producing well distributed hash values, are optimized for hashing single long strings.

We focused our efforts on two aspects: 1) optimize a proven hash function for a large number of short strings, and 2) create a fast hash function with minimal number of CPU instructions that behaves relatively well. For 1), we used hand-vectorized Murmur3's 32-bit hash function, due to its algorithmic simplicity, to process 8 k-mers at a time using AVX and AVX2 intrinsics. We further

Algorithm 2 Robin Hood Hashing: Insertion

```
1:  B : hash table size
2:  H() : hash function
3:  ⟨k, v⟩ : key-value pair to insert
4:  T : array [0..B − 1] of key-value pairs

5:  b ← H(k) modulo B
6:  p ← b
7:  while (p < B) AND T[p] is not empty do
8:      b′ ← H(T[p].k) modulo B
9:      if (p − b) > (p − b′) then
10:         swap(⟨k, v⟩, T[p])
11:         b ← b′
12:     else if (p−b) == (p−b′) AND (k == T[p].k) then return
13:     end if
14:     p ← p + 1
15: end while

16: T[p] ← ⟨k, v⟩
```

unrolled the loop 4 times to hide the latency of instructions such as *PMULLD*. The implementation is specialized for keys of lengths 1 to 64 bytes, corresponding to lengths of commonly used k-mers. For 2), we implemented Cyclic Redundancy Code (CRC32C) using CPU intrinsics. Empirically, CRC32C produced reasonably well-distributed hash values for use with hash tables.

We note that using the same hash function for both the *Partition* and *Count* steps results in high collision rate in a small number of hash table buckets. This can be prevented by choosing different hash functions, potentially from the same family by using different seeds. CRC32C does not form a family of hash functions without extensive bit mixing, however. We therefore use CRC32C during the *Count* step only, to maximize its performance benefit if hash table resize is required.

4.4.2 Cache Friendly Hash Table. Given a set of k-mers and a well behaved hash function, successive k-mers are unlikely to be mapped to nearby positions in the same cache line. However, hash table insertion requires searching within a bucket for the insertion position or a matching k-mer. Such searches in open addressing hash tables with linear reprobing [70] access successive memory locations, allowing for cache line reuse and reduced memory bandwidth requirements for a parallel application.

Linear reprobing remains suboptimal, however, as entries from different buckets are interleaved and must be inspected during search. Instead, we adapted Robin Hood Hashing [71], a modified linear reprobing strategy for open addressing hash tables. Rather than inserting at the first unoccupied position, an element is inserted immediately after the last element of the previous bucket, shifting subsequent elements forward as needed. This in effect groups together all elements of a bucket, minimizing the number of elements to inspect and thus bandwidth requirement. Algorithm 2 outlines the Robin Hood Hashing insertion algorithm.

Our implementation of Robin Hood Hashing include several modifications. Instead of recomputing hash values (Algorithm 2 Line 8), a separate byte array I stores the offset value for a bucket b at $I[b]$. The data elements in bucket b then reside in T between the inclusive range $b + I[b]$ and $b + I[b + 1]$. This range can be computed in constant time and the data elements accessed without additional search. We also replace multiple consecutive *swap* operations (Algorithm 2 Line 10) with a single `memmove` instruction.

4.4.3 Software Prefetching. Hash table access is typically irregular as hashing reduces spatial locality. Hardware prefetching is therefore ineffective. However, since k-mers are inserted in batch and the bucket offsets are known, the insertion positions of the k-mers can be computed prior to insertion. We utilize software prefetching in a similar fashion as for FM Index. We first prefetch the offset byte array using the hash value of a k-mer, and then prefetch the data element array entry using the offset value. The hash values for the prefetched k-mers are stored in a circular buffer for reuse during actual insertion. We use a similar approach as that for FM Index's batch size to determine an optimal prefetch look-ahead distance.

5 EXPERIMENTAL RESULTS

Our experiments used the system configurations described in Section 3. All the codes were compiled with icc version 17.0.5. For a complete picture, for each kernel, we also show a comparison with the best published GPGPU implementations. Since the reported results are for older GPGPU architectures, we obtained the source codes in each case and ran them on the latest Nvidia Pascal P100 GPUs with a peak single precision performance of 10.6 TFLOP/s and high bandwidth memory (HBM) with a bandwidth of 732 GB/s. The codes were compiled with CUDA compiler version 8.0. Unless explicitly stated otherwise, we report only GPGPU execution time for comparison, and exclude the time for data transfer between the host and the GPGPU. The NVBIO library [47] from Nvidia provides optimized software components for building new bioinformatics tools. We have reported a comparison with NVBIO (v1.1.00) for the kernels that are available.

All our results use a single dual socket Intel Xeon processor, a single Intel Xeon Phi processor and a single Nvidia Pascal P100. The host processor of GPGPU is Intel Xeon E5-2697 v4 (Broadwell, BDX) running at 2.30GHz. Each experiment was run thrice and the minimum time was used.

Note that none of the kernels apart from k-mer counting need any synchronization as the tasks being parallelized are independent of each other.

5.1 Smith Waterman algorithm

We study performance of the use case in which a large set of DNA sequence pairs have to be aligned (Section 4.1). We compare the

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more information go to www.intel.com/benchmarks.

Benchmark results were obtained prior to implementation of recent software patches and firmware updates intended to address exploits referred to as "Spectre" and "Meltdown". Implementation of these updates may make these results inapplicable to your device or system.

Table 2: Calculation of theoretical peak throughput of Smith Waterman for SKX and KNL. The number of cycles required is higher on KNL as the blend operation consumes 2 cycles on KNL but only 1 cycle on SKX.

	SKX			KNL		
# Cores (a_1)	56			68		
Base AVX freq (GHz) (a_2)	1.7			1.2		
AVX version	AVX512	AVX512	AVX512	AVX512	AVX2	AVX2
Precision	int32	int16	int8	int32	int16	int8
# Bits in AVX register (a_3)	512	512	512	512	256	256
# Bits used per cell of E, F or H (a_4)	32	16	8	32	16	8
# Cells updated simulateneously ($a_5 = \frac{a_3}{a_4}$)	16	32	64	16	16	32
Vector operations required for one cell update	6 max, 4 add, 1 cmpeq, 1 masked blend					
# Vector INT ALU ports (a_6)	2					
# Cycles required/port to execute above operations (a_7)	12	12	12	13	13	13
(# Cell updates)/cycle/core/port ($a_8 = \frac{a_5}{a_7}$)	1.33	2.67	5.33	1.23	1.23	2.46
(# Cell updates)/cycle/core ($a_9 = a_6 \times a_8$)	2.67	5.33	10.67	2.46	2.46	4.92
(# Cell updates)/cycle ($a_{10} = a_9 \times a_1$)	149.33	298.67	597.33	167.38	167.38	334.77
Giga cell updates / second ($a_{11} = a_{10} \times a_2$)	253.87	507.73	1015.47	200.86	200.86	401.72

performance of our optimized implementations at three precision levels: 8-bit integer (int8), 16-bit integer (int16) and 32-bit integer (int32), on multi-core and many-core processors.

We used the standard throughput metric of giga cell updates per second (GCUPS), where a cell update is defined as the computation required to update one cell of H and the corresponding cells of E and F.

5.1.1 Input datasets. We use both real and synthetic datasets. Our real datasets are created by running GATK's Haplotype Caller and extracting the input to the Smith Waterman module. For each precision level, we created the input by filtering out the pairs that need higher precision. We also prepared five synthetic datasets by sampling millions of pairs of DNA sequences from human genome chromosome 1, with sequence length of 100, 200, 300, 400, and 500, respectively. These synthetic datasets are deliberately designed to have identical length for all pairs in a set to prevent idle vector lanes and to demonstrate maximum achievable performance. We note that Illumina, the dominant sequencers in the market, mostly produce equal length reads.

5.1.2 Performance on SKX and KNL. Figure 3 depicts the performance obtained on SKX and KNL. To assess the performance of our implementations, we use the roofline model to compute the theoretical peak SW throughput of the underlying architectures. Since SW is a compute bound problem, we can ignore the bandwidth requirement for calculation of the theoretical peak throughput. Our calculation, as shown in Table 2, takes into account the number of vector lanes used, the operations required to perform one cell update in a vector lane, and the number of cores and frequency. We do not count any other operations like load and store operations, update of loop variables, branching for the loop, sorting, conversion from SoA to AoS, etc. in this calculation. Since SKX supports 8- and 16-bit integer operations in AVX512, its theoretical peak throughput is higher at lower precisions as the number of vector lanes that can be used are higher. KNL lacks such support, and its theoretical peak throughput correspondingly are lower.

Since turbo frequency of a processor is dynamic, it cannot reliably be used to estimate the efficiency of an implementation. Therefore, we use the base frequency for the calculation of the theoretical peak throughput and run the two processors with turbo OFF. We calculate the efficiency of each implementation as the achieved throughput with turbo OFF divided by the theoretical peak throughput. Figures 3a and 3c show the efficiencies achieved by our implementation. On SKX, efficiency falls as precision reduces

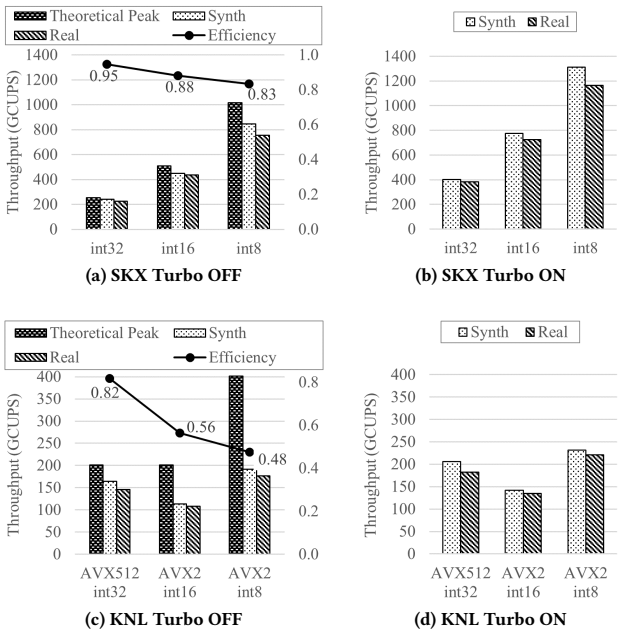


Figure 3: Throughput of SKX and KNL for Smith Waterman kernel. All experiments use the following scoring parameters: $\gamma = 1$ (match), $\delta = -4$ (mismatch), $\alpha = -6$ (gap open), and $\beta = -1$ (gap extend).

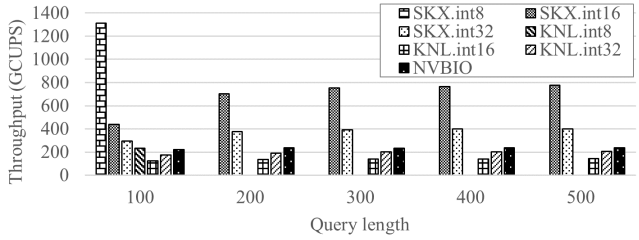


Figure 4: Throughput of various implementations for the Smith Waterman kernel. The results for int8 precision are provided only in cases where it is applicable.

since SW alignment with reduced precision runs faster, amplifying the overhead of sorting. Our efficiency on KNL with int32 precision is lower than SKX due to higher overhead of our scalar sorting implementation and scalar instructions like updating of loop variables. Our efficiency with AVX2 on KNL is lower because, while KNL supports AVX2, it is designed for AVX512 performance and not AVX2 performance.

Figures 3b and 3d show the performance with turbo ON. The gap between maximum turbo frequency with all the cores running and the base frequency is significantly narrower on KNL compared to SKX. This is evident in the results as the performance gap between SKX and KNL widens further when turbo is ON.

5.1.3 Comparison with GPGPU implementations. Figure 4 illustrates the comparison with the SW benchmark from the NVBIO library. This benchmark performs pairwise SW of one sequence with a set of sequences, while our implementations allow the more flexible input of a set of pairs of sequences. To ensure fair comparison, we use the same input as that of NVBIO by fixing one sequence in each pair presented to our implementation. This has no effect on performance. The best reported NVBIO throughput is 102.3 GCUPS on Nvidia K40 GPU with peak single precision throughput of 5 TFLOP/s. We observed a 2.3x increase in throughput to 237 GCUPS on P100 with 2.1× higher peak single precision throughput.

As can be seen in Figure 4, SKX performs the best out of the three architectures. This is due to the availability of lower precision operations on SKX and the high efficiency achieved by our implementation compared to the theoretical peak throughput. In NGS secondary analysis, the sequences on which SW is performed are typically small; hence, int8 or int16 precision is generally sufficient and the maximum performance is achievable.

5.2 PairHMM algorithm

Since PairHMM has very similar characteristics to SW, we do not delve deep into PairHMM performance. The throughput metric used is GCUPS where one cell update is defined as the computation to update one cell in all three matrices. Our PairHMM implementation that is part of GKL does not have support for KNL. Given the higher SKX throughput for SW, we did not evaluate KNL for PairHMM.

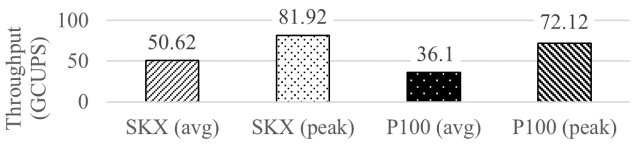


Figure 5: Throughput of various implementations for the PairHMM kernel.

A few GPU implementations are available. Among the most recent ones, Ren *et al.* [38] report throughput of 23.56 GCUPS on Nvidia K40 GPU. Wang *et al.* [39] report peak throughput of 34.8 GCUPS on Nvidia Titan X GPU. To our knowledge, Wang *et al.* [39] report the best throughput, even accounting for relative advantage of Titan X over K40. Thus we obtained their source code from them for evaluation on P100.

The input datasets for GPU vs. SKX comparison are obtained by running GATK's HaplotypeCaller and extracting the input to the PairHMM module. The input datasets are organized by batches. Each batch has a set of reads and haplotypes to align with each other using the PairHMM algorithm. Both implementations process the input in batches. Both implementations report the peak and average throughput across batches. Figure 5 shows that SKX performs slightly better than P100. Our GPGPU run shows proportionate increase in throughput to 72.12 GCUPS on P100 compared to the reported 34.8 GCUPS on Titan X.

5.3 FM-index based sequence search

We focus on two use cases for the FM-index based DNA sequence search: end-to-end matching of reads and end-to-end matching of fixed length seeds extracted from the reads. We present our results for both exact search and inexact search problems. We use the number of bases of queries processed per second as the throughput metric.

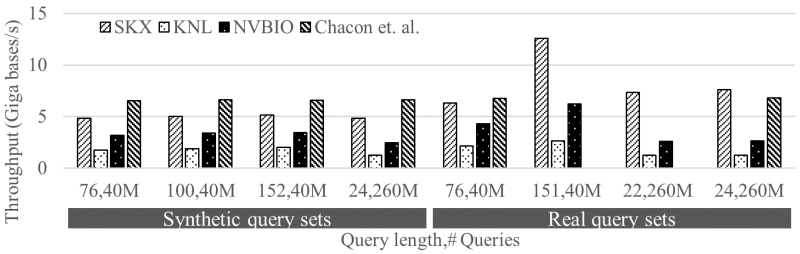
5.3.1 Input datasets. We perform our experiments on two real read datasets from NCBI's sequence read archive – human reads of length 76 (accession number SRS008504) and length 151 (accession number ERS474404). We also created three synthetic datasets by sampling reads of length 76, 101 and 151 from the entire human genome to mimic the lengths of reads found in real read datasets. The synthetic datasets ensure that each query has at least one match in the reference genome.

5.3.2 Performance of exact search. Performance counters for our implementations for SKX and KNL are presented in Table 3, while the performance of our implementations along with that of NVBIO and Chacon *et al.* on P100, are presented in Figure 6a. NVBIO provides a benchmark for exact search using FM-index (`fmmap`) that implements the backward search algorithm. The source code by Chacon *et al.* required that query lengths are a multiple of 4. To compare with this novel approach, we created three more synthetic read datasets by sampling reads of length 76, 100 and 152 (multiple of 4) from the human genome as reference. We tested performance with entire reads as well as seeds of length 22 and 24 extracted from the reads as queries, 22 being the default seed length of Bowtie2 and NVBIO. For each implementation, we consider the time consumed by the exact search kernel only, with the queries as input and suffix array intervals as output. Time for other operations like seed extraction are excluded.

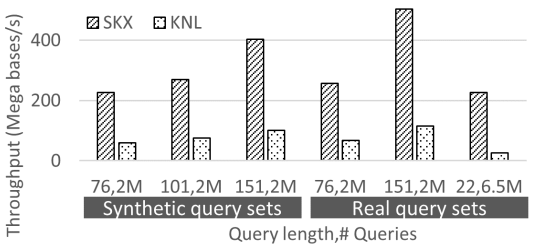
While SKX has lower available bandwidth (228 GB/s), the bigger L2 and L3 caches reduce some of the bandwidth pressure. The

Table 3: Performance counters on SKX for FM-index based exact search. The experiments were conducted using real reads. Both the original sequence and its reverse complement are searched as is the norm in sequence mapping software.

Architecture	SKX				KNL			
Query set	76,40M		151,40M		76,40M		151,40M	
# Cores	1	56	1	56	1	68	1	68
# Inst. (in billions)	198.37	199.01	324.06	324.06	211.72	225.62	341.22	356.71
# Cycles (in billions)	81.59	2.41	93.17	2.40	263.31	3.96	391.93	6.44
Time (s)	32.63	0.97	37.26	0.96	188.08	2.83	279.95	4.60
BW (GB/s)	7.30	204.60	6.50	199.60	14.00	109.70	16.20	68.90
IPC/core	2.43	1.47	3.48	2.41	0.80	0.84	0.87	0.81
BW/core (GB/s)	7.30	3.65	6.50	3.56	14.00	1.61	16.20	1.01
Speedup of 1 node over 1 core	33.81		38.77		66.55		60.86	
Scaling efficiency	0.60		0.69		0.98		0.89	



(a) Exact search. Reference sequence: human genome.



(b) Inexact search (up to 1 mismatch). Reference sequence: human genome.

Figure 6: Throughput of various implementations of the FM-index based exact and inexact search kernels. Both the original sequence and its reverse complement are searched as is the norm in sequence mapping software.

performance is still bandwidth bound on SKX. Our full node runs achieve nearly 200 GB/s at about 88% of the peak machine bandwidth. The bound on bandwidth is evident in the reduction in IPC per core and bandwidth per core from single core to entire node resulting in scaling efficiencies of only 0.6 and 0.69 on the two query sets. On KNL, our full node runs achieve bandwidths of 109.7 GB/s and 68.9 GB/s at less than 25% of the high bandwidth (440 GB/s) available on KNL. The IPC stays nearly the same between single core and full node runs achieving scaling efficiencies of 0.98 and 0.89 for the two query sets. The lower scaling efficiency of the second dataset is partly due to load imbalance. The bandwidth per core reduces probably due to the reuse of data from L2 caches of remote cores reducing bandwidth requirement. Moreover, nearly 99.9% of all load requests hit in L1 or L2 caches. Thus, KNL performance is not memory bound. Recall that byte level comparison necessitated AVX2 instructions. Since a majority of the instructions are scalar and AVX2 instructions also do not perform very well on KNL, the performance on KNL is instruction bound. This reflects in the low IPC per core. As a result, our implementation for SKX is 2.5-6× faster compared to that for KNL. Moreover, it is 1.3-2.2× faster compared to NVBIO on P100.

Chacon *et al.* [46] reported throughput of up to 2 Gigabases/sec on Nvidia GTX Titan GPU with a memory bandwidth of 288 GB/s and peak single precision throughput of 4.5 TFLOP/s using synthetic query sets that ensure that every query has at least one match. We see a 3.3× increase in the throughput at 6.6 Gigabases/sec on P100 with 2.5× higher memory bandwidth.

For synthetic queries, Chacon *et al.* performs the fastest. However, on an average, it is only 1.33× faster than our SKX implementation despite the 3.21× higher memory bandwidth on P100 and the additional algorithmic improvements used to reduce the number of memory accesses. For real queries, where some queries may not result in any match, our implementation on SKX has very similar throughput to that of Chacon *et al.* We expect that our implementations can also benefit by adopting the n-step FM-index technique of Chacon *et al.*

5.3.3 Performance of inexact search. Figure 6b shows the throughput of our inexact search implementation on SKX and KNL. SKX is 3.6-4.3× faster than KNL for full read sets and 8.7× faster than KNL for the set of seeds. NVBIO has a library function available for inexact search, but due to the lack of available benchmark code, we could not compare with it.

5.4 K-mer counting

For the k-mer counting kernel, the use case consists of a set of reads and an integer k, and output is the count of all unique k-mers in the input. The number of k-mers processed per second is used as the throughput metric.

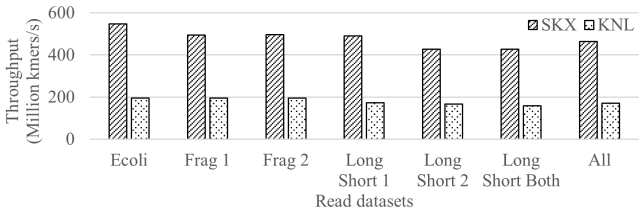


Figure 7: Throughput of k-mer counting on SKX and KNL for different data sets.

5.4.1 Input datasets. Experiments are run on real datasets, summarized in Table 4. The E. coli data is available [72] as a test dataset with the distribution of the assembler SPAdes [73]. The human chromosome 14 data (Chr14) is a standard benchmark data for genome assembly available at GAGE [74]. The Chr14 read dataset includes 6 different files. We create multiple datasets by picking subsets of the files.

5.4.2 Performance on SKX and KNL. Performance of our implementations on SKX and KNL for various datasets is shown in Figure 7. On an average, SKX is 2.7× faster than KNL, achieving a peak throughput of 550 million k-mers/sec. The algorithm contains predominantly scalar code for which KNL requires larger number of cycles to execute, resulting in its lower throughput. Specifically, while Robin Hood hashing scheme is good at keeping bandwidth requirement low, it is very challenging to vectorize due to the frequent branching, short loops, and irregular memory accesses. The step that inserts k-mers received by an MPI rank into its local hash table is bandwidth bound on SKX, but instruction bound on KNL. Moreover, the larger caches on SKX also help relieve some bandwidth pressure. As a result, despite having a higher memory bandwidth available, KNL performs worse than SKX. A different approach for KNL could potentially be more vectorization friendly, but irregular memory accesses and frequent branching are the key characteristics of most hashing techniques.

The hash function computation is also bandwidth bound on SKX but compute bound on KNL. This is because the ratio of amount of computation to be done for hash function to the amount of data to be read is lower than the compute to bandwidth ratio of SKX. But due to higher bandwidth availability on KNL, hash function is compute bound on KNL.

Table 4: Real datasets used for kmer counting kernels.

Dataset	Subset	# reads (10^6)	Read length	# kmers (10^6)
Ecoli	Frag	28.4	101	1948.5
Chr14	Frag 1	18.25	101	1292.4
	Frag 2	18.25	101	1292.4
	Long, Short 1	12.54	76-101	880.3
	Long, Short 2	12.54	76-101	878.7
	Long, Short all	25.07	76-101	1758.9
	All	61.58	76-101	4343.7

5.4.3 Comparison with GPGPU implementations. Many GPU based k-mer counters are embedded within assemblers, making it difficult to extract them in a form that allows fair comparison. To the best of our knowledge, Gerbil is the only standalone GPGPU based k-mer counter. Gerbil follows the partition-and-compute approach. First, it partitions the input dataset into multiple files such that, for each unique k-mer, all its occurrences are in the same file. These files can then be processed in parallel. Gerbil performs the first step entirely on the host CPU and uses both the host CPU and GPGPU for the second step. GPU performance of Gerbil therefore cannot be assessed directly. In addition, file I/O is an inherent part of the Gerbil algorithm. We measure the total time, including file I/O, to assess the benefit of GPGPUs and to compare Gerbil with our implementation.

Figure 8 compares the time to solution for different configurations. For each configuration of Gerbil, we ran it with different number of CPU threads to identify the best throughput and use that for the graph. We do not observe significant performance benefit in adding GPGPU for Gerbil. Moreover, our implementation on SKX was 1.15 − 3.84× faster than the best performance of Gerbil.

6 KEY INSIGHTS FROM OUR STUDY

This work provides a deep understanding of the performance characteristics of key kernels for NGS secondary analysis and their performance on the SKX and KNL architectures. Two of the four kernels – SW and PairHMM – are compute bound. The other two – FM-index based sequence search and k-mer counting – are bandwidth bound on SKX, but instruction bound on KNL due to excessive scalar operations. Integer operations at 8-bit, 16-bit and 32-bit level are quite useful with max, add, cmpeq, and, or, xor, permute, blend, shift, etc. being particularly useful. PairHMM is the only kernel that uses floating point operations and the operations are limited to only multiply and add. None of the expensive numerical floating point operations like division, square root, etc. are used. Double precision operations are used very rarely. Only PairHMM uses them in the rare scenario when single precision is insufficient.

FM-index based sequence search and hash table insertion for k-mer-counting do not benefit much from vectorization and also suffer from irregular memory accesses and poor data locality. While the problem of irregular memory access can be solved to some extent by using software prefetching, this results in the kernels being bandwidth bound. The fact that only part of the prefetched cache

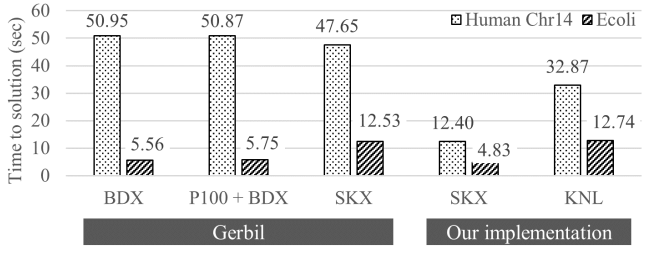


Figure 8: Time consumed by Gerbil and our optimized implementation on BDX, P100, SKX, and KNL.

line is utilized for k-mer counting further amplifies the bandwidth problem.

For FM-index based sequence search, the requirement to use byte level vector cmpeq instructions restricts the bucket size h to 32. If we had a vector cmpeq instruction that could compare at the 2-bit level, we could have 4 times larger buckets. This would reduce the size of *COcc* and result in more accesses to *COcc* hitting in the same *COcc* entry, thus reducing memory bandwidth requirement. So, for the same available bandwidth, we could get better performance. Sequence search trades computation for locality by using compressed FM-index. If we use *Occ* instead of its compressed variant *COcc*, we need less computation since we do not need to count the occurrences of a base from the BWT substring. But it has worse data locality and for every cache line prefetched for Occ, only 4 bytes are used.

7 POTENTIAL ARCHITECTURAL IMPROVEMENTS

Our analysis establishes that SKX is better suited than KNL for NGS secondary analysis. Therefore, with SKX as the platform of choice, we postulate multiple architectural improvements that can improve its performance for NGS secondary analysis tasks.

Higher peak performance will help both SW and PairHMM. This could simply be in the form of additional cores as both the kernels demonstrate good scaling to 56 cores. On the other hand, more bandwidth in the form of HBM will help FM-index based sequence search and k-mer counting. For example, with HBM on SKX, exact search performance should improve by $60 - 70\%$. For sequence search using FM-index, vector cmpeq operations at 2-bit level should help. PairHMM, hash function computation and k-mer reverse complement computation could benefit from single cycle vector permute operations. Hash function computation can also benefit from a lower latency vector integer multiplication instruction.

Modern HPC processors rely on vectorization to achieve performance. The difficulty of vectorizing workloads with irregular memory access and frequent branching can be reduced by well performing gather and scatter instructions and good masking support in vector instructions.

In addition, if there is a way to read from memory only the bytes of a cache line that we are going to use, that will help reduce the memory bandwidth requirement. This will need some kind of in-memory processing before the data reaches the memory controller or the bus. Especially for sequence search, the data to be read for processing each base will reduce from 64 Bytes to just 4 Bytes and could possibly improve the performance significantly. If this is not available, a scratchpad cache that is as fast as L1 or L2 and offers the programmer control over what data resides in it can reduce the amount of unnecessary data in the cache. With more efficient use of space, we can fit more useful data thus improving data locality. This is already present in GPGPUs in the form of shared memory.

8 CONCLUSIONS AND FUTURE WORK

In this paper, we presented an in-depth study of the performance characteristics of NGS secondary analysis workloads and assessed the suitability of modern Intel Xeon processor and Intel Xeon Phi processor for these workloads. For this purpose, we also conducted comprehensive optimizations of the key kernels underlying NGS secondary analysis for the two architectures.

This study has three beneficial outcomes. First, we developed a suite of well optimized implementations of the kernels for the two architectures. The suite will be open sourced to enable performance improvements to existing tools and to aid in developing new tools for NGS analytics. Second, the study provides a deep understanding of the performance characteristics of the kernels and what constitutes important and unimportant architectural features for these kernels. Additionally, we acquired a deep understanding of the performance bottlenecks of the two architectures for supporting NGS analytics. Based on our extensive study, we suggest several architectural improvements for improving the performance of NGS secondary analysis.

Future works can evaluate the benefits of some of the architectural ideas suggested here. We would also explore a completely different approach of designing the architecture from scratch starting with studying optimized FPGA implementations of the key kernels.

9 ACKNOWLEDGEMENTS

This research is supported in part by the National Science Foundation under IIS-1416259 and the grant for Intel Parallel Computing Center on Big Data in Biosciences and Public Health.

REFERENCES

[1] D. R. Bentley, S. Balasubramanian, H. P. Swerdlow, G. P. Smith, J. Milton, C. G. Brown, K. P. Hall, D. J. Evers, C. L. Barnes, H. R. Bignell, J. M. Boutell, J. Bryant, R. J. Carter, R. K. Cheetham, A. J. Cox, D. J. Ellis, M. R. Flatbush, N. A. Gormley, S. J. Humphray, L. J. Irving, M. S. Karbelashvili, S. M. Kirk, H. Li, X. Liu, K. S. Maisinger, L. J. Murray, B. Obradovic, T. Ost, M. L. Parkinson, M. R. Pratt, I. M. Rasolonjatovo, M. T. Reed, R. Rigatti, C. Rodighiero, M. T. Ross, A. Sabot, S. V. Sankar, A. Scally, G. P. Schroth, M. E. Smith, V. P. Smith, A. Spiridou, P. E. Torrance, S. S. Tzonev, E. H. Vermaas, K. Walter, X. Wu, L. Zhang, M. D. Alam, C. Anastasi, I. C. Aniebo, D. M. Bailey, I. R. Bancarz, S. Banerjee, S. G. Barbour, P. A. Baybayan, V. A. Benoit, K. F. Benson, C. Bevis, P. J. Black, A. Boodhun, J. S. Brennan, J. A. Bridgham, R. C. Brown, A. A. Brown, D. H. Buermann, A. A. Bundu, J. C. Burrows, N. P. Carter, N. Castillo, M. C. E. Catenazzi, S. Chang, R. N. Cooley, N. R. Crake, O. O. Dada, K. D. Diakoumakos, B. Dominguez-Fernandez, D. J. Earnshaw, U. C. Egbujor, D. W. Elmore, S. S. Etchin, M. R. Ewan, M. Fedurco, L. J. Fraser, K. V. F. Fajardo, W. S. Furey, D. George, K. J. Gietzen, C. P. Goddard, G. S. Golda, P. A. Granieri, D. E. Green, D. L. Gustafson, N. F. Hansen, K. Harnish, C. D. Haudenschild, N. I. Heyer, M. M. Hims, J. T. Ho, A. M. Horgan, K. Hoschler, S. Hurwitz, D. V. Ivanov, M. Q. Johnson, T. James, T. A. H. Jones, G. D. Kang, T. H. Kerelska, A. D. Kersey, I. Khrebtukova, A. P. Kindwall, Z. Kingsbury, P. I. Kokko-Gonzales, A. Kumar, M. A. Laurent, C. T. Lawley, S. E. Lee, X. Lee, A. K. Liao, J. A. Loch, M. Lok, S. Luo, R. M. Mammen, J. W. Martin, P. G. McCauley, P. McNitt, P. Mehta, K. W. Moon, J. W. Mullens, T. Newington, Z. Ning, B. L. Ng, S. M. Novo, M. J. O'Neill, M. A. Osborne, A. Osnowski, O. Ostadan, L. L. Paraschos, L. Pickering, A. C. Pike, A. C. Pike, D. C. Pinkard, D. P. Pliskin, J. Podhasky, V. J. Quijano, C. Raczy, V. H. Rae, S. R. Rawlings, A. C. Rodriguez, P. M. Roe, J. Rogers, M. C. R. Bacigalupo, N. Romanov, A. Romieu, R. K. Roth, N. J. Rourke, S. T. Ruediger, E. Rusman, R. M. Sanches-Kuiper, M. R. Schenker, J. M. Seoane, R. J. Shaw, M. K. Shiver, S. W. Short, N. L. Sizto, J. P. Sluis, M. A. Smith, J. E. S. Sohna, E. J. Spence, K. Stevens, N. Sutton, L. Szajkowski, C. L. Tregidgo, G. Turcatti, S. Vandevondele, Y. Verhovsky, S. M. Virk, S. Wakelin, G. C. Walcott, J. Wang, G. J. Worsley, J. Yan, L. Yau, M. Zuerlein, J. Rogers, J. C. Mullikin, M. E. Hurles, N. J. McCooke, J. S. West, F. L. Oaks, P. L. Lundberg, D. Klenerman, R. Durbin, and A. J. Smith. Accurate whole human genome sequencing using reversible terminator chemistry. *nature*, 456(7218): 53–59, 2008.

[2] Illumina. HiSeqX™ Series of Sequencing Systems. URL https://www.illumina.com/documents/products/datasheets/datasheet-hiseq-x-ten.pdf.

[3] Zachary D Stephens, Skylar Y Lee, Faraz Faghri, Roy H Campbell, Chengxiang Zhai, Miles J Efron, Ravishankar Iyer, Michael C Schatz, Saurabh Sinha, and Gene E Robinson. Big data: astronomical or genomical? *PLoS Biol*, 13(7):e1002195, 2015.

[4] Md Vasimuddin, Sanchit Misra, and Srinivas Aluru. Identification of significant computational building blocks through comprehensive investigation of NGS secondary analysis methods. *[Preprint] bioRXiv*, April 2018. doi: 10.1101/301903.

[5] T. Smith and M. Waterman. Identification of common molecular subsequences. *Journal of Molecular Biology*, 147(1):195–197, 1981.

[6] R. Durbin, S. Eddy, A. Krogh, and G. Mitchison. *Biological Sequence Analysis: Probabilistic Models of Proteins and Nucleic Acids*. Cambridge University Press, 1998.

[7] Paolo Ferragina and Giovanni Manzini. Opportunistic data structures with applications. In *Foundations of Computer Science, 2000. Proceedings. 41st Annual Symposium on*, pages 390–398. IEEE, 2000.

[8] Guillaume Marçais and Carl Kingsford. A fast, lock-free approach for efficient parallel counting of occurrences of k-mers. *Bioinformatics*, 27(6):764–770, 2011.

[9] Heng Li, Jue Ruan, and Richard Durbin. Mapping short DNA sequencing reads and calling variants using mapping quality scores. *Genome research*, 18(11): 1851–1858, 2008.

[10] Ruiqiang Li, Yingrui Li, Karsten Kristiansen, and Jun Wang. SOAP: short oligonucleotide alignment program. *Bioinformatics*, 24(5):713–714, 2008.

[11] Ruiqiang Li, Chang Yu, Yingrui Li, Tak-Wah Lam, Siu-Ming Yiu, Karsten Kristiansen, and Jun Wang. SOAP2: an improved ultrafast tool for short read alignment. *Bioinformatics*, 25(15):1966–1967, 2009. doi: 10.1093/bioinformatics/btp336.

[12] Ben Langmead, Cole Trapnell, Mihai Pop, and Steven L Salzberg. Ultrafast and memory-efficient alignment of short dna sequences to the human genome. *Genome biology*, 10(3):1, 2009.

[13] Ben Langmead and Steven L Salzberg. Fast gapped-read alignment with Bowtie 2. *Nature methods*, 9(4):357–359, 2012.

[14] Heng Li and Richard Durbin. Fast and accurate short read alignment with Burrows–Wheeler transform. *Bioinformatics*, 25(14):1754–1760, 2009.

[15] Heng Li. Aligning sequence reads, clone sequences and assembly contigs with bwa-mem. *arXiv:1303.3997v1 [q-bio.GN]*, 2009.

[16] M. A. DePristo, E. Banks, R. E. Poplin, K. V. Garimella, J. R. Maguire, C. Hartl, A. A. Philippakis, G. del Angel, M. A. Rivas, M. Hanna, A. McKenna, T. J. Fennell, A. M. Kernytsky, A. Y. Sivachenko, K. Cibulskis, S. B. Gabriel, D. Altshuler, and M. J. Daly. A framework for variation discovery and genotyping using next-generation DNA sequencing data. *Nat Genet*, 43(5):491–498, 2011.

[17] Broad Institute: GATK best practices. URL https://software.broadinstitute.org/gatk/best-practices.

[18] A. Wozniak. Using video-oriented instructions to speed up sequence comparison. *Bioinformatics*, 13(2):145–150, 1997. doi: 10.1093/bioinformatics/13.2.145. URL +http://dx.doi.org/10.1093/bioinformatics/13.2.145.

[19] Torbjørn Rognes and Erling Seeberg. Six-fold speed-up of Smith-Waterman sequence database searches using parallel processing on common microprocessors. *Bioinformatics*, 16:699–706, 09 2000.

[20] Michael Farrar. Striped Smith-Waterman speeds database searches six times over other simd implementations. *Bioinformatics*, 23(2):156–161, 2007. doi: 10.1093/bioinformatics/btl582. URL +http://dx.doi.org/10.1093/bioinformatics/btl582.

[21] Isaac TS Li, Warren Shum, and Kevin Truong. 160-fold acceleration of the Smith-Waterman algorithm using a field programmable gate array (fpga). *BMC Bioinformatics*, 8(1):185, Jun 2007. ISSN 1471-2105. doi: 10.1186/1471-2105-8-185. URL https://doi.org/10.1186/1471-2105-8-185.

[22] Adam Szalkowski, Christian Ledergerber, Philipp Krähenbühl, and Christophe Dessimoz. SWPS3 – fast multi-threaded vectorized Smith-Waterman for ibm cell/b.e. and x86/sse2. *BMC Research Notes*, 1(1):107, Oct 2008. ISSN 1756-0500. doi: 10.1186/1756-0500-1-107. URL https://doi.org/10.1186/1756-0500-1-107.

[23] Witold R Rudnicki, Aleksander Jankowski, Aleksander Modzelewski, Aleksander Piotrowski, and Adam Zadrożny. The new simd implementation of the Smith-Waterman algorithm on cell microprocessor. *Fundamenta Informaticae*, 96(1-2): 181–194, 2009.

[24] Yongchao Liu, Douglas L. Maskell, and Bertil Schmidt. CUDASW++: optimizing smith-waterman sequence database searches for CUDA-enabled graphics processing units. *BMC Research Notes*, 2(1):73, May 2009. ISSN 1756-0500. doi: 10.1186/1756-0500-2-73. URL https://doi.org/10.1186/1756-0500-2-73.

[25] Łukasz Ligowski and Witold Rudnicki. An efficient implementation of smith waterman algorithm on GPU using CUDA, for massively parallel scanning of sequence databases. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–8. IEEE, 2009.

[26] Yongchao Liu, Bertil Schmidt, and Douglas L. Maskell. CUDASW++2.0: enhanced Smith-Waterman protein database search on CUDA-enabled GPUs based on SIMT and virtualized SIMD abstractions. *BMC Research Notes*, 3(1):93, Apr 2010. ISSN 1756-0500. doi: 10.1186/1756-0500-3-93. URL https://doi.org/10.1186/1756-0500-3-93.

[27] Łukasz Ligowski, Witold R. Rudnicki, Yongchao Liu, and Bertil Schmidt. Chapter 11 - accurate scanning of sequence databases with the Smith-Waterman algorithm. In Wen-mei W. Hwu, editor, *{GPU} Computing Gems Emerald Edition*, Applications of GPU Computing Series, pages 155 – 171. Morgan Kaufmann, Boston, 2011. ISBN 978-0-12-384988-5. doi: https://doi.org/10.1016/B978-0-12-384988-5.00011-5. URL https://www.sciencedirect.com/science/article/pii/B9780123849885000115.

[28] Torbjørn Rognes. Faster Smith-Waterman database searches with inter-sequence simd parallelisation. *BMC Bioinformatics*, 12(1):221, 2011. doi: 10.1186/1471-2105-12-221.

[29] Yongchao Liu, Adrianto Wirawan, and Bertil Schmidt. CUDASW++ 3.0: accelerating Smith-Waterman protein database search by coupling CPU and GPU SIMD instructions. *BMC Bioinformatics*, 14(1):117, Apr 2013. ISSN 1471-2105. doi: 10.1186/1471-2105-14-117. URL https://doi.org/10.1186/1471-2105-14-117.

[30] Yongchao Liu and Bertil Schmidt. SWAPHI: Smith-Waterman protein database search on xeon phi coprocessors. pages 25th IEEE International Conference on Application–specific Systems, Architectures and Processors (ASAP 2014), 04 2014.

[31] Haidong Lan, W. Liu, B. Schmidt, and B. Wang. Accelerating large-scale biological database search on xeon phi-based neo-heterogeneous architectures. In *2015 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, pages 503–510, Nov 2015. doi: 10.1109/BIBM.2015.7359735.

[32] Jeff Daily. Parasail: SIMD C library for global, semi-global, and local pairwise sequence alignments. *BMC Bioinformatics*, 17(1):81, Feb 2016. ISSN 1471-2105. doi: 10.1186/s12859-016-0930-z. URL https://doi.org/10.1186/s12859-016-0930-z.

[33] Enzo Rucci, Carlos Garcia, Guillermo Botella, Armando De Giusti, Marcelo Naiouf, and Manuel Prieto-Matias. First experiences optimizing Smith-Waterman on intel's knights landing processor. *arXiv:1702.07195 [cs.DC]*, 2016.

[34] Kristian Cibulskis, Michael S Lawrence, Scott L Carter, Andrey Sivachenko, David Jaffe, Carrie Sougnez, Stacey Gabriel, Matthew Meyerson, Eric S Lander, and Gad Getz. Sensitive detection of somatic point mutations in impure and heterogeneous cancer samples. *Nature biotechnology*, 31(3):213–219, 2013.

[35] Mauricio Carneiro, Tadeusz Jordan, P C Pratts, and George Vacek. Optimization of a haplotype PairHMM class for GPU processing. URL https://github.com/MauricioCarneiro/PairHMM.

[36] Chris Rauer, George S. Powley, Mir Ahsan, and Jr. Nicholas Finamore. White paper: Accelerating genomics research with opencl and fpgas. Technical report, Intel Corporation, March 2016. URL https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-accelerating-genomics-opencl-fpgas.pdf.

[37] Genomics kernel library (GKL), 2016. URL https://github.com/Intel-HLS/GKL.

[38] Shanshan Ren, Koen Bertel, and Zaid Al-Ars. Exploration of alternative gpu implementations of the pair-hmms forward algorithm. *2016 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*, 00:902–909, 2016. doi: doi.ieeecomputersociety.org/10.1109/BIBM.2016.7822645.

[39] Jie Wang, Xinfeng Xie, and Jason Cong. Communication optimization on gpu: A case study of sequence alignment algorithms. *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017. doi: 10.1109/IPDPS.2017.79.

[40] M. Ito and M. Ohara. A power-efficient fpga accelerator: Systolic array with cache-coherent interface for pair-hmm algorithm. In *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, pages 1–3, April 2016.

[41] Sitao Huang, Gowthami Jayashri Manikandan, Anand Ramachandran, Kyle Rupnow, Wen-mei W Hwu, and Deming Chen. Hardware acceleration of the Pair-HMM algorithm for DNA variant calling. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 275–284. ACM, 2017.

[42] Paolo Ferragina and Giovanni Manzini. An experimental study of an opportunistic index. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 269–278. Society for Industrial and Applied Mathematics, 2001.

[43] E. Fernandez, W. Najjar, and S. Lonardi. String matching in hardware using the fm-index. In *2011 IEEE 19th Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 218–225, May 2011. doi: 10.1109/FCCM.2011.55.

[44] Jing Zhang, Heshan Lin, Pavan Balaji, and Wu-chun Feng. Optimizing burrows-wheeler transform-based sequence alignment on multicore architectures. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 377–384. IEEE, 2013.

[45] Alejandro Chacón, Juan Carlos Moure, Antonio Espinosa, and Porfidio Hernández. n-step fm-index for faster pattern matching. *Procedia Computer Science*, 18: 70 – 79, 2013. ISSN 1877-0509. doi: https://doi.org/10.1016/j.procs.2013.05.170. URL http://www.sciencedirect.com/science/article/pii/S187705091300313X. 2013 International Conference on Computational Science.

[46] Alejandro Chacón, Santiago Marco-Sola, Antonio Espinosa, Paolo Ribeca, and Juan Carlos Moure. Boosting the FM-Index on the GPU: Effective techniques to mitigate random memory access. *IEEE/ACM Trans. Comput. Biol. Bioinformatics*, 12(5):1048–1059, September 2015. ISSN 1545-5963. doi: 10.1109/TCBB.2014.2377716. URL http://dx.doi.org/10.1109/TCBB.2014.2377716.

[47] J. Pantaleoni and N. Subtil. NVBIO: A library of reusable components designed by NVIDIA corporation to accelerate bioinformatics applications using CUDA. URL http://nvlabs.github.io/nvbio/.

[48] Szymon Grabowski, Marcin Raniszewski, and Sebastian Deorowicz. Fm-index for dummies. In Stanisław Kozielski, Dariusz Mrozek, Paweł Kasprowski, Bożena Małysiak-Mrozek, and Daniel Kostrzewa, editors, *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, pages 189–201, Cham, 2017. Springer International Publishing.

ISBN 978-3-319-58274-0.

[49] David R Kelley, Michael C Schatz, and Steven L Salzberg. Quake: quality-aware detection and correction of sequencing errors. *Genome Biology*, 11(11):1, 2010.

[50] Xiao Yang, Karin S Dorman, and Srinivas Aluru. Reptile: representative tiling for short read error correction. *Bioinformatics*, 26(20):2526–2533, 2010.

[51] Yongchao Liu, Jan Schröder, and Bertil Schmidt. Musket: a multistage k-mer spectrum-based error corrector for illumina sequence data. *Bioinformatics*, 29(3): 308–315, 2013.

[52] Serafim Batzoglou, David B Jaffe, Ken Stanley, Jonathan Butler, Sante Gnerre, Evan Mauceli, Bonnie Berger, Jill P Mesirov, and Eric S Lander. ARACHNE: a whole-genome shotgun assembler. *Genome Research*, 12(1):177–189, 2002.

[53] Daniel R Zerbino and Ewan Birney. Velvet: algorithms for de novo short read assembly using de bruijn graphs. *Genome Research*, 18(5):821–829, 2008.

[54] Jared T Simpson, Kim Wong, Shaun D Jackman, Jacqueline E Schein, Steven JM Jones, and Inanç Birol. ABySS: a parallel assembler for short read sequence data. *Genome Research*, 19(6):1117–1123, 2009.

[55] Evangelos Georganas, Aydın Buluç, Jarrod Chapman, Steven Hofmeyr, Chaitanya Aluru, Rob Egan, Leonid Oliker, Daniel Rokhsar, and Katherine Yelick. HipMer: an extreme-scale de novo genome assembler. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 14. ACM, 2015.

[56] Stefan Kurtz, Apurva Narechania, Joshua C Stein, and Doreen Ware. A new method to compute k-mer frequencies and its application to annotate large repetitive plant genomes. *BMC Genomics*, 9(1):517, 2008.

[57] Rachid Ounit, Steve Wanamaker, Timothy J. Close, and Stefano Lonardi. CLARK: fast and accurate classification of metagenomic and genomic sequences using discriminative k-mers. *BMC Genomics*, 16:236, 2015.

[58] Guillaume Rizk, Dominique Lavenier, and Rayan Chikhi. DSK: k-mer counting with very low memory usage. *Bioinformatics*, 29(5):652–653, 2013.

[59] Sebastian Deorowicz, Marek Kokot, Szymon Grabowski, and Agnieszka Debudaj-Grabysz. KMC 2: Fast and resource-frugal k-mer counting. *Bioinformatics*, 31 (10):1569–1576, 2015.

[60] Pall Melsted and Jonathan K Pritchard. Efficient counting of k-mers in DNA sequences using a bloom filter. *BMC Bioinformatics*, 12(1):1, 2011.

[61] Rajat Shuvro Roy, Debashish Bhattacharya, and Alexander Schliep. Turtle: Identifying frequent k-mers with cache-efficient algorithms. *Bioinformatics*, 30(14): 1950–1957, 2014.

[62] Qingpeng Zhang, Jason Pell, Rosangela Canino-Koning, Adina Chuang Howe, and C Titus Brown. These are not the k-mers you are looking for: efficient online k-mer counting using a probabilistic data structure. *PloS ONE*, 9(7):e101271, 2014.

[63] Marek Kokot, Maciej Długosz, and Sebastian Deorowicz. KMC 3: counting and manipulating k -mer statistics. *Bioinformatics*, 2017.

[64] Marius Erbert, Steffen Rechner, and Matthias MÃijller-Hannemann. Gerbil: a fast and memory-efficientk-mer counter with GPU-support. *Algorithms for molecular biology: AMB*, 12:9, 2017.

[65] T. Pan, P. Flick, C. Jain, Y. Liu, and S. Aluru. Kmerind: A Flexible Parallel Library for K-mer Indexing of Biological Sequences on Distributed Memory Systems. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, PP(99):1–1, 2017. ISSN 1545-5963. doi: 10.1109/TCBB.2017.2760829. 00003.

[66] Burton H Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, 1970.

[67] Graham Cormode and S Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms*, 55(1):58–75, 2005.

[68] Intel corporation: Intel® 64 and ia-32 architectures optimization reference manual. URL https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-optimization-manual.pdf.

[69] Avinash Sodani, Roger Gramunt, Jesus Corbal, Ho-Seop Kim, Krishna Vinod, Sundaram Chinthamani, Steven Hutsell, Rajat Agarwal, and Yen-Chen Liu. Knights landing: Second-generation intel xeon phi product. *IEEE Micro*, 36(2):34–46, 2016. ISSN 0272-1732. doi: doi.ieeecomputersociety.org/10.1109/MM.2016.25.

[70] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844, 9780262033848.

[71] Pedro Celis. *Robin Hood Hashing*. PhD thesis, Waterloo, ON, Canada, 1986.

[72] Ecoli reads datasets. URL http://spades.bioinf.spbau.ru/spades_test_datasets/ecoli_mc/.

[73] Anton Bankevich, Sergey Nurk, Dmitry Antipov, Alexey A. Gurevich, Mikhail Dvorkin, Alexander S. Kulikov, Valery M. Lesin, Sergey I. Nikolenko, Son Pham, Andrey D. Prjibelski, Alexey V. Pyshkin, Alexander V. Sirotkin, Nikolay Vyahhi, Glenn Tesler, Max A. Alekseyev, and Pavel A. Pevzner. SPAdes: A New Genome Assembly Algorithm and Its Applications to Single-Cell Sequencing. *Journal of Computational Biology*, 19, 2012.

[74] Genome assembly gold standard evaluations. URL http://gage.cbcb.umd.edu/data/index.html.