Accelerating SAM to BAM Parsing on FPGA

Safa Messaoud
messaou2@illinois.edu

***Abstract*—The emergence of the Next Generation Sequencing (NGS) machines revolutionized the genetic research and medical practice. Personalized medicine, based on tailoring the clinical diagnosis and decision making to the patient's genetic make-up, is believed to be the future of healthcare. Fast and accurate Variant Calling to determine potential mutations, is one of the key enablers of personalized medicine. We identify the SAM to BAM file parsing to be a sequential bottleneck in the Variant Calling pipeline, run on IBM POWER8's processor. The report presents SAM2BAM, which is, to the best of our knowledge, the first FPGA-based accelerator for SAM to BAM parsing. SAM2BAM achieves up to 10X speedup compared to the software baseline.**

Keywords*—*Variant Calling, POWER8, FPGA, BAM, SAM.

I. Introduction

The emergence of the Next Generation Sequencing (NGS) machines [19], has significantly decreased the time and cost of sequencing the human genome [16]. This advent started the era of personalized medicine, based on tailoring the diagnosis and decision making process to the patient's genetic make-up. However, the current sequencing technology does not the have the capability of reading contiguous long DNA sequences. Instead, it generates short DNA fragments, called short reads, of lengths between 35 and 500 nucleotides[16]. The position of the reads within the sequenced genome is unknown. Different computational methods have been developed to extract genomic information from the short reads. Variant Calling (VC) is an algorithm that aims at identifying potential mutations from the sequencing data. Mutations are permanent changes in the DNA sequence. They can result from DNA copying mistakes during cell division, exposure to ionizing radiation or chemicals, or infection by viruses. Mutations could be potential risk indicator for certain diseases [7].

Variant Calling is a computational intensive algorithm. This is mainly due to the complexity of identifying variants from the sequencing data, as well as the huge size of files that need to be manipulated. The Genome Analysis Toolkit (GATK) Variant Calling pipeline [21], proposed by BROAD Institute at MIT and Harvard and advocated as best practice, has four major sequential stages: (1) alignment, (2) data compression (3) data filtering and (4) variant discovery. The run time of the GATK pipeline on a POWER8 machine with 24 SMT8 cores and 500 GB of memory is 25 hours. The execution time breakdown over the different stages is shown in Figure 1. The Variant Calling pipeline performs computation on huge files. The FASTQ file [8], consisting of the short reads generated by the sequencing machine for the whole genome of one patient with 30x coverage, is of the order of 180 GB. The SAM file generated by the first stage of the pipeline, and containing the mapping positions of the short reads to the reference sequence, can reach up to few Tera Bytes size per person. In order to reduce the storage requirements and transmission bandwidth, SAM is compressed to a BAM file [9], using SAMTools [11] software. The resulting compression ratio for SAM to BAM files is around 4 [9]. The SAM to BAM conversion stage represents around 16% of the overall execution time on POWER8.

High speed Variant Calling is a key enabler of personalized medicine. By leveraging supercomputer and cloud resources, it was possible to run the Variant Calling algorithm in less than two hours [24]. Hardware accelerators, however, represent a more cost and power efficient option. As sequencing data analysis is a relatively recent field, previous attempts for hardware acceleration of these analytics only covered the alignment and variant calling algorithms. Besides, these algorithms are common across different NGS data analysis workflows.

In this work, we focus on accelerating the second stage of the GATK pipeline, namely the SAM to BAM conversion, on a POWER8 processor. In future work, we will also address the data cleaning stage. SAM to BAM conversion involves two steps: (1) parsing and (2) compression. In the parsing stage, every record in the SAM file is translated from the ASCII format to a binary one. The generated BAM records are then compressed using the BGZF[15] library, implemented on top of the standard gzip algorithm. Our profiling of the SAM to BAM conversion algorithm resulted in identifying the parsing stage as being a serial bottleneck. Parsing one record consists of sequentially performing multiple different operations, each on a small chunk of data. Hence, the desired computing platform must be able to support fine grained parallelism and efficient execution of sequential tasks. As a result, designing custom hardware is a natural choice for accelerating the parsing algorithm. In this work, we present SAM2BAM, an FPGA [18]-based accelerator for SAM to BAM parsing. SAM2BAM achieves up to 10X speedup compared to the software baseline. We implemented SAM2BAM on an Altera Statix V FPGA platform and tested it with real SAM files obtained by converting to SAM the G15512.HCC1954.1 BAM file benchmark [3], obtained from UCSC, using SAMTools.

The rest of the report is organized as follows: In section II, we summarize the related work. Section III introduces the variant calling pipeline. In Section IV, we present the SAM and BAM formats and analyze the SAM to BAM conversion algorithm. The accelerator design is explained in Section V. Finally, we draw conclusion and give an outlook for future work, in Section VI.

II. Related Work

In this section, we review the recent work on accelerating the GATK Variant Calling pipeline.

GATK recommends the Burrows Wheeler Aligner (BWA)[22] for the alignment stage. GPU-based accelerators

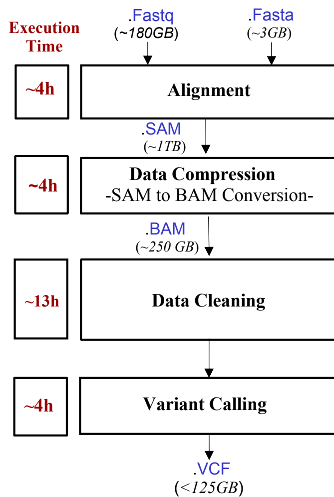

Fig. 1. Execution Time Breakdown of the Variant Calling Pipeline
Input File: G15512.HCC1954.1. Model 8247-22L@3026MHz. 24 cores SMT8, 512GB (Detailed description follows in Section III)

of the BWA algorithm include BarraCUDA [2], Arioc [17] and Swift [1]. BarraCUDA achieves up to 15X speedup over the CPU implementation. Arioc runs the first stage of BWA, namely the seed and extend stage, 10 times faster than the CPU version. Swift claims a 2X speed-up on the second stage of BWA, the Smith-Waterman alignment, compared to the multi-threaded CPU version. FPGA based aligners include BWA by Convey Computers and the German Cancer research center[14]. The proposed implementation runs 20 times faster than the corresponding CPU version. A more recent Smith-Waterman FPGA-based implementation, was reported by IBM [13]. It achieves 1.6X speed-up over the software-based multi-threaded version.

To the best of our knowledge, there has not been any HW acceleration aiming the BAM to SAM conversion, or data cleaning steps.

As for the Variant Calling stage, GATK recommends the Haplotype Caller (HC) [6] algorithm. Paired Hidden Markov Model (PairHMM) is identified as the major computational bottleneck of this stage. both an FPGA and a CUDA implementation achieving around 300X speed-up of PairHMM, were proposed by the BROAD institute [12].

III. Variant Calling

In this section, we explain the GATK Variant Calling pipeline. Variant Calling is the process of determining variations in the genome, that are associated with certain diseases. Variants are of three types: (1) Single Nucleotide Polymorphism (SNP), referring to replacing one nucleotide by another one, (2) Insertion/ Deletion (INDEL), occurring when one nucleotide is inserted/ deleted from the genome, and (3) Structural Variant (SV), that pertains to the insertion or deletion of larger than a sequence of 1k bases.

The GATK Variant Calling pipeline [21], shown in Figure 1 has 4 processing stages. It takes as input the sequencing data and a reference sequence, in form of FSTQ and FASTA files, respectively. It produces a VCF file containing the variants.

Alignment is the first stage in the pipeline. It aims at reconstructing the sequenced genome, by mapping the short reads to the reference sequence. GATK recommends the Burrows Wheeler Aligner (BWA) [22] for this task. BWA, first, finds the longest exact matches between a given short read (query) and the reference sequence (target). Then, it uses the Smith-Waterman[23] algorithm to align the short read to the reference sequence at the identified location. The alignment results are stored in a SAM file.

The next stage in the pipeline is *SAM to BAM Conversion*. SAM is an ASCII file. It is, hence, easy to be manipulated by biologists. However, SAM file size can reach few Tera Bytes per genome. In order to reduce storage requirements and transmission bandwidth, SAM is converted to a binary compressed file Format, called BAM. BAM consists of sorted and indexed records. Each record can be randomly accessed, loaded to the memory and decompressed, whenever needed in the following stages of the pipeline. The structures of the SAM and BAM files are explained in greater details in Section IV.

The *Data Cleaning* stage aims at compensating for the sequencing and alignment errors. It evolves two main steps, namely *Mark Duplicate* and *Base Re-calibrator*. The *Mark Duplicate* stage removes/ marks the read duplicates, which are identical short reads, mapped to the same position in the reference sequence. The *Base Re-calibrator* step uses the sequence context in order to provide more accurate quality scores for the bases.

The Haplotype Caller[6] algorithm is recommended algorithm for the *Variant Calling* stage. A haplotype is a set of DNA variations, that tend to be inherited together. SNPs and indels are called from haplotypes, assembled in regions of interest.

IV. SAM to BAM Conversion

In this section, we introduce the SAM/ BAM formats, present the SAM to BAM conversion algorithm and describe its profiling results on POWER8.

A. SAM Format

SAM [9] stands for Sequence Alignment/Map format. It is an ASCII tab-delimited file composed of a header and an alignment section. Each entry in the alignment section is a record corresponding to one short read's mapping information. Each SAM record (SAM_R) consists of 11 mandatory fields (See Table I) and variable number of optional fields, called TAGs. In the table below, RNEXT, PNEXT and TLEN describe pair-end reads, obtained by forward and reverse reading of a DNA fragment. This process aims at improving the alignment accuracy. Tags store additional information in the form *TAG:TYPE:VALUE*. TAG is a two-character string, that takes predefined values from a dictionary[9] and indicates the category of the additional information. TYPE defines the data type of VALUE.

B. BAM Format

BAM is the binary compressed version of SAM. The BAM fields are derived from the corresponding SAM ones, using a set of parsing functions, as shown in Table II. The parsing

TABLE I. SAM FILE FORMAT

SAM Field	Brief description
QNAME	Short read name
FLAG	Bitwise FLAG (Alignment and Sequencing details)
RNAME	Reference sequence name
POS	Mapping position in the reference sequence
MAPQ	Mapping Quality
CIGAR	CIGAR string
RNEXT	Reference name of the mate-read
PNEXT	Position of the mate-read
TLEN	Distance between the read and its mate
SEQ	Segment Sequence
QUAL	ASCII of sequencing quality +33

functions are described in II. As the input and out put types are describes in Table II, we omit this details in Table III.

TABLE II. BAM FILE FORMAT

BAM Field	Brief description	Size
block_size	*SizeDetermine*(SAM_R)	`int32_t`
refID	*Get_ref_ID*(RNAME)	`int32_t`
pos	*Str2l* (POS)−1	`int32_t`
bin_mq_nl	*Parse_bin_mq*(QNAME, POS, CIGAR, MAPQ)	`uint32_t`
flag_nc	*Parse_flag*(FLAG, CIGAR)	`uint32_t`
l_seq	*Length*(SEQ)	`int32_t`
next_refID	*Get_ref_ID*(RNEXT)	`int32_t`
next_pos	*Str2l* (PNEXT)−1	`int32_t`
tlen	*Str2l*(TLEN)	`int32_t`
read_name	QNAME '\0'	`char`
cigar	*Parse_cigar*(CIGAR)	`uint32_t` [*n_cigar_op*(CIGAR)]
seq	*Encode_to_4bits* (SEQ)	`uint8_t` [(*Length*(SEQ)+1)/2]
qual	*Phred_base*(QUAL)	`char`[*length*(SEQ)]
tag	*Parse_tag*(TAG)	`size_tag(TAG)`

C. BAM to SAM Conversion Algorithm

SAM to BAM conversion involves two steps, namely parsing and compression. Since the size and number of the fields in a SAM record are not pre-defined, the SAM file is parsed sequentially. When it reads a tab, it starts converting a new field in the corresponding BAM one, using a new function (See Table III). When it reaches a new line symbol, it stores the size of the previous BAM record and starts processing the new QNAME field. As functions operating on the different fields run in linear time, the overall parsing stage complexity is O(n), where n is the number of bytes in the SAM file. Each parsed SAM record is then compressed using the gzip algorithm, which also runs in linear time.

In the current SAMTools implementation, the parser is a single thread that processes a serial data stream from the SAM file. The compression stage is enhanced using multi-threading. Each parsed record is dispatched to a new thread.

D. Profiling

We Profiled the SAM to BAM conversion application on a POWER8 system with 512 GB memory and 12 cores, each supporting up to 8 threads simultaneously (SMT8). We used htop [5], which is an interactive process viewer, as profiling tool and ran the application on the whole genome cell line G15512.HCC1954.1 benchmark[3]. Because of SMT8, the system can support up to 192 threads. However, in average, our

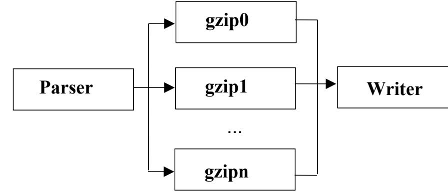

Fig. 2. SAM to BAM Conversion Software Architecture

TABLE III. SAM PARSING FUNCTIONS

Parsing Function	Brief Description
SizeDetermine	determines the size of the parsed BAM file
Strtol	transforms a variable length string into a 32-bit signed integer.
Length	determines the number of bytes in a string
get_ref_id	extracts the reference sequence ID from the header file and convert it to integer
Parse_bin_mq	determine the number of Bin, the record should be placed in
Parse_flag	converted to zeros
Parse_cigar	transforms each base length and its associated operator in the cigar string to a 32 bit unsigned integer. The four lower bits code for the operator, according to a lookup table ('MIDNSNSHP' →[0,8]). The remaining bits contain the binary representation of the base length.
n_cigar_op	determines the number of operations in a cigar string. The operations are mainly Match (M), Insert (I) and Delete (D). Exp: *n_cigar_op*(3M5I) returns 2.
Encode_to_4bits	maps each character in SEQ to an 8 bit unsigned integer between 0 and 15 ('=ACMGRSVTWYHKDBN'→[0,15])
Phred_base	subtracts 33 from each byte in Qual
Parse_tag	returns the parsed TAG field. TAG and TYPE do not change. The VALUE conversion follows different rules, given by TYPE [9]

profiling results showed that only 11 compression threads are running simultaneously. We deducted, that the parsing stage does not provide sufficient records to multiple compression threads. Hence, it represents a sequential bottleneck in the SAM to BAM conversion flow.

FPGA arises as a natural option to accelerate the parsing stage. On the one hand, the regularity of the memory access guarantees a high utilization of the FPGA. On the other hand, the FPGA implementation can leverage the fine grained parallelism of the parsing algorithm, as the different fields can be processed simultaneously. Since the amount of computation per thread is small, a multi-threaded or GPU-based implementation is not beneficial. This is due to the high communication and synchronization overhead between threads.

V. ACCELERATOR DESIGN

This section introduces the design of the SAM2BAM FPGA-based accelerator. In Section V-A, we perform a statistical analysis to estimate critical design parameters. Section V-B describes the accelerator architecture. Performance and rescouce utilization results are discussed in Section V-C.

An FPGA, attached to the POWER8 processor, can read 64 Bytes from its cache to its input buffer every cycle. Hence, our goal is to achieve a maximum throughput of 64 Byte/cycle.

A. Statistical Analysis for Design Parameters Estimation

The main design challenge consists of determining the start, length and type of each input field present in the FPGA

input buffer and dispatch it to the corresponding processing unit, while maintaining a throughput of 64 Byte/cycle. Since the sizes and number of fields per record are variable, this task can be highly complex. To alleviate this difficulties, we performed a statistical study, which aims at determining the upper-bound of some design critical parameters, namely the number of variable fields in a record, the number of different fields in the input buffer at a given time, and the size of FLAG, POS, PNEXT, MAPQ, TLEN and TAG_VALUE. For this purpose, we consider 34 records sampled from the Mutation Calling Benchmark 4 dataset, [3] provided by UCSC. The records were characterized by a low variance, hence we were able to derive tight confidence intervals for all the critical design parameters. Based on the results of this study, we assume the following: the number of optional fields (tags) in one record ranges between 1 and 11, at most one *newline* symbol and 11 fields are present in the input buffer, the maximum sizes of FLAG, POS, PNEXT, MAPQ, TLEN, TAG_VALUE are, respectively, 4, 9, 9, 3, 10 and 5 bytes. Besides, we assume that two consecutive *tab* or *newline* symbols do not occur in a SAM file. In case of violation of one of these assumptions, an error signal is sent to the application.

B. Architectural Overview of the Accelerator

Figure 3 shows a top-level design of the accelerator. SAM2BAM is about 10 times faster than SAMtools' Software implementation of the parsing stage. The *Dispatcher* is the key component that enabled this speed-up. It exploits bit-level and instruction level parallelism to achieve a throughput of 64 Bytes/cycle. The *Dispatcher* determines the beginning and length of all SAM fields present in the FPGA input buffer and enables the *Processing Units* (PUs) corresponding to each of these fields. The PUs implement the functions presented in Table III. In case, a SAM field is not completely available in the current clock cycle, the *Dispatcher* notifies the PU by setting the *'done'* signal to 0. Then, the PU has to wait for the next clock cycle to obtain the remaining part of the field and continue processing. Once parsed, the fields are arranged by a *Data Aggregator* module in the output buffer and sent back to the FPGA cache or to the memory.

In the following, we describe the design of the SAM2BAM components. We particularly focus on the *Dispatcher*, as it is the enabler for the achieved performance.

1) Dispacher: The *Dispatcher* design is shown in Figure 4. As SAM is a tab-delimited file, a new field start always proceeds a *tab* or a *newline* symbol. Hence, the fist stage in the *Dispatcher* module compares the 64 bytes from the input buffer with the ASCII symbols of *tab* and *newline*. The comparison results are stored in two 64 bit vectors, namely t and n. In the next cycle, the *Dispacher* starts executing two pipelines in parallel. The first pipeline (PP1) determines the starting positions of every field in the input buffer. The second pipeline (PP2) identifies the number and type of these fields.

The pipeline PP1 determines the positions of the 11 fields that could be present in the input buffer. For this purpose, the n and t vectors are partitioned across 8 *PosDetermine8* blocks, that generate 4 local positions. For instance, the position of the fourth field *pos4* in the first *PosDetermine8* block is 7, in case a *newline* or *tab* are detected in bytes 1, 3, 5 and 7.

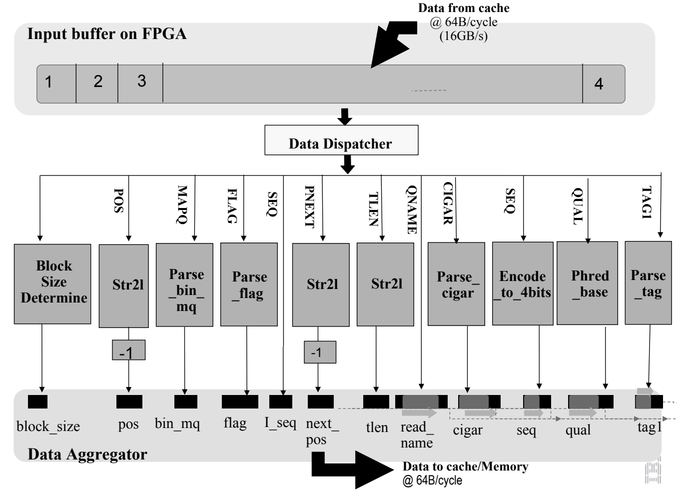

Fig. 3. SAM2BAM Accelerator Design

It is 8, if bytes 2, 4, 6 and 8 are *tabs*/*newlines*. The local positions are aggregated by the *PosDetermine64* module, to generate 11 global positions. The *PosTranslate* stage produces two versions of the global position vector. The first version contains the positions of the fields coming before a *newline*, and the second version consists of the ones coming after the *newline*, i.e. belonging to a new record. This information is useful for the *Router* stage.

Pipeline PP2 is run in parallel to PP1. It determines the types of the fields (See Table I) in the input buffer. For this purpose, 8 *StateDetermine8* blocks are executed simultaneously. Each block counts the number of fields in an 8 Byte chunk of the input data. Each stage outputs two variable: (1) *SO*, a counter of the fields in the first record in the input buffer, and *SN*, a counter of the ones in the second record. In case only one record is present in the input buffer, SN is set to zero. The local counters are merged to a global counter in the *StateDetermine64* stage. The *StateEnabler* module keeps track of the dispatching state, which we define as the type of the first field in the input buffer. Hence, given the SO and SN counters, *StateEnabler* can exactly determine the type of the fields, currently present in the input buffer. Similarly to the *PosTranslate* stage, *StateEnabler* generates an enable vector for the fields belonging to the first records, and those in the second one.

The pipelines PP1 and PP2 meet in the *Router* stage. Given the current state, the field positions and the enable signals, the *Router* determines the length of each field and whether it is complete in the current cycle. These information, together with the starting positions and enable signals, are sent to the processing unites.

2) Processing Units: Although the *Processing Units* perform different operations, we use the same techniques to accelerate them. Each *Processing Unit* receives the start position and length of the field to be processed. It then compares this position to all numbers in the positions range, namely 1 to 64. In case of a match, the PU performs different operations depending on the length of the received field. For example, the *Strol* function converts a string to a long, by applying different

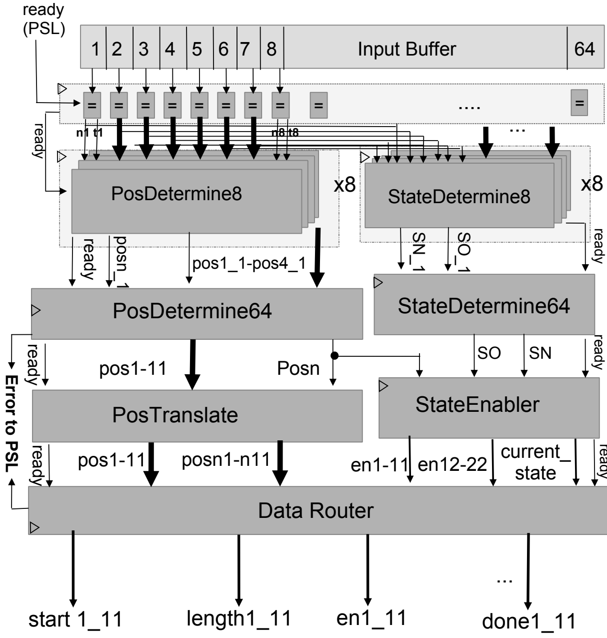

Fig. 4. Architecture

arithmetical rules depending on the number of characters in the input string. The output size of a processing unit corresponds to the maximal size of the field.

3) Aggregator: The *Aggregator* block received the parsed fields and their real lengths from the *Processing Units*. It also receives the current state and the number of fields in the input buffer from the *Dispatcher*. Based on this information, it allocates a place for the field in the 64 Byte output buffer.

C. Results

SAM2BAM was implemented on an Altera Stratix V FPGA, operated at 250MHz, and tested with the G15512.HCC1954.1 SAM file benchmark, obtained from UCSC[3]. Table IV summarizes the resource utilization and performance metrics of the design. BAM2BAM achieves significant speedup compared to the SAMtools parser run on POWER8, as it exploits bit-level parallelism in the *Dispacher*, instruction-level parallelism, by using a deep pipelined design, and data level parallelism, by processing the different fields in parallels in several PUs. The overall speed-up of the SAM to BAM conversion application is limited by the bandwidth of the PCIe bus that connects the FPGA to POWER8. Since PCIe only supports 8GB/s, the overall speedup is around 5 %. SAM2BAM, however, has to ensure a throughput of 16GB/s in order to guarantee full utilization of the PCIe bus every cycle. If SAM2BAM processes 8GB per second, some fields may not be complete in the current cock cycle. Hence the corresponding PU does not produce results until the following cycle. So, a throughput of 8GB/s can be sustained.

VI. Conclusion and Future Work

In this report, we present SAM2BAM, the first FPGA-based accelerator for parsing SAM to BAM file formats. Our design achieves a 10x speedup compared to the SAMTools software baseline. The *Dispacher* module is the key enabler of the achieved performance. The overall application speed-up is limited by the bandwidth of the bus connecting the FPGA to POWER8. In the future, we plan to accelerate the whole Variant Calling workflow on POWER8. Our analysis of the GATK pipeline resulted in identifying common computational kernels across the different stages [4]. We plan to rewrite the Variant Calling pipeline in terms of these kernels and accelerate them on hardware to achieve a higher speed-up.

TABLE IV. Results

Metric	Value
Throughput	16 GB/s
Latency	80 ns
Achieved Speedup	10x
Resource Utilization	8%

References

[1] Rozanski, Andrei, Daniel T. Ohara, and Pedro AF Galante. ”GPU-Accelerated Pipeline For Next Generation Sequencing Data Simulation.” Zebrafish 1.21,186,406: 4-237.

[2] Klus, Petr, et al. ”BarraCUDA-a fast short read sequence aligner using graphics processing units.” BMC research notes 5.1 (2012): 27.

[3] TCGA Mutation/Variation Calling Benchmark 4 at CGHub. https://cghub.ucsc.edu/datasets/benchmarkdownload.html

[4] Athreya, Arjun. ”Core Computational Kernels in the CompGen Machine.” 2015

[5] htop an interactive process viewer for Linux. http://hisham.hm/htop/

[6] GATK Haplotype Caller. https://www.broadinstitute.org/gatk/guide/article?id=4148

[7] Disease-Related Variation Databases. http://www.humgen.nl/mutationDB.html

[8] Cock, Peter JA, et al. ”The Sanger FASTQ file format for sequences with quality scores, and the Solexa/Illumina FASTQ variants.” Nucleic acids research 38.6 (2010): 1767-1771.

[9] Sequence Alignment/Map Format Specification. https://samtools.github.io/hts-specs/SAMv1.pdf

[10] Stuecheli, Jeff. ”POWER8.” (2013)

[11] Li, Heng, et al. ”The sequence alignment/map format and SAMtools.” Bioinformatics 25.16 (2009): 2078-2079.

[12] Carneiro MO, Poplin R, Biagioli E, Thibault S. ”Enabling high throughput haplotype analysis through hardware acceleration”. https://github.com/MauricioCarneiro/PairHMM/blob/master/doc/pairhmm.tex

[13] Jaspers, Michael Johannes. ”Acceleration of read alignment with coherent attached FPGA coprocessors.” Master Thesis, Delft University of Technology. (2015).

[14] Convey Computer Corporation. ”Speeding Up the Next Generation Sequencing Pipeline.” (2012). https://samtools.github.io/hts-specs/SAMv1.pdf

[15] L. Peter Deutsch. ”GZIP File format specification version 4.3, RFC 1952.” https://samtools.github.io/hts-specs/SAMv1.pdf

[16] Illumina. ”HiSeq X Series of Sequencing Systems.” (2015). https://www.garvan.org.au/research/kinghorn-centre-for-clinical-genomics/clinical-genomics/datasheet-hiseq-x-ten.pdf

[17] Wilton, Richard, et al. ”Arioc: high-throughput read alignment with GPU-accelerated exploration of the seed-and-extend search space.” PeerJ 3 (2015): e808.

[18] History of FPGAs at the Wayback Machine. https://web.archive.org/web/20070412183416/http://filebox.vt.edu/users/tmagin/history.htm

[19] Koboldt, Daniel C., et al. ”The next-generation sequencing revolution and its impact on genomics.” Cell 155.1 (2013): 27-38.

[20] Hamburg, Margaret A., and Francis S. Collins. ”The path to personalized medicine.” New England Journal of Medicine 363.4 (2010).

[21] DePristo, Mark A., et al. ”A framework for variation discovery and genotyping using next-generation DNA sequencing data.” Nature genetics 43.5 (2011).

[22] Li, Heng, and Richard Durbin. ”Fast and accurate short read alignment with BurrowsWheeler transform.” Bioinformatics 25.14 (2009).

[23] Pearson, William R. ”Searching protein sequence libraries: comparison of the sensitivity and selectivity of the Smith-Waterman and FASTA algorithms.” Genomics 11.3 (1991).

[24] Kelly, Benjamin J., et al. ”Churchill: an ultra-fast, deterministic, highly scalable and balanced parallelization strategy for the discovery of human genetic variation in clinical and population-scale genomics.” Genome biology 16.1 (2015).

View publication stats