Accelerating SNP genotyping from whole genome sequencing data for bedside diagnostics

Chen Sun[1,†] and Paul Medvedev[1,2,3]

1 Department of Computer Science and Engineering, The Pennsylvania State University, USA

2 Department of Biochemistry and Molecular Biology, The Pennsylvania State University, USA

3 Center for Computational Biology and Bioinformatics, Genome Sciences Institute of the Huck, The Pennsylvania State University, USA

[†]to whom correspondence should be addressed: chensun@cse.psu.edu

Abstract

Motivation: Genotyping a set of variants from a database is an important step for identifying known genetic traits and disease related variants within an individual. The growing size of variant databases as well as the high depth of sequencing data pose an efficiency challenge. In clinical applications, where time is crucial, alignment-based methods are often not fast enough. To fill the gap, (Shajii et al. 2016) propose LAVA, an alignment-free genotyping method which is able to more quickly genotype SNPs; however, there remains large room for improvements in running time.

Results: We present the VarGeno method for SNP genotyping from Illumina whole genome sequencing data. Our method performs 2-8 times faster than LAVA with similar memory usage. VarGeno uses Bloom filters to achieve a 2x speedup without changing the accuracy, while a 8x speedup is achieved by using a quality value filtering method (VarGeno-QV) at the cost of only slight decrease (0.04%) in accuracy.

Availability: VarGeno is freely available at: https://github.com/medvedevgroup/vargeno.

1 Introduction

Given a set of target genetic variants, the problem of variant genotyping is to report which variants an individual possesses (Luikart et al. 2003; Shajii et al. 2016; Syvänen 2005). Single nucleotide polymorphism (SNP) genotyping has been widely used in human disease-related research such as genome wide association studies (Hirschhorn and Daly 2005). The approaches to SNP genotyping can be roughly divided into three categories: microarray methods, sequencing mapping-based methods, and alignment-free methods.

The first approach uses SNP arrays (Pastinen et al. 2000). SNP arrays are based on the hybridization of fragmented, single-stranded, target DNA, labelled with fluorescent dyes, to arrays containing immobilized allele-specific oligonucleotide probes (LaFramboise 2009). SNP arrays are fast and inexpensive; however, they can only hold a limited number of probes: the state-of-the-art Affymetrix genome-wide SNP array 6.0 has only 906,000 SNP probes, compared with 31,565,214 known common SNPs in dbSNP (build 150).

The second approach is based on high-throughput whole genome sequencing and read mapping. In a standard pipeline using this method, sequencing reads are first mapped to the reference genome. The mapping results are then used as input for genotyping tools such as SAMtools mpileup (Li et al. 2009), or Freebayes (Garrison and Marth 2012), or GATK HaplotypeCaller (Depristo et al. 2011; McKenna et al. 2010). The limitation of this direction is that it requires a lot of time in read mapping. This limitation becomes especially crucial in clinical applications, where bedside genotyping of disease-related SNPs may become common in the future.

The third approach is based on high-throughput whole genome sequencing followed by an alignment-free sequence comparison (Vinga and Almeida 2003). Read mapping generates large amounts of general information for downstream analysis. However, in SNP genotyping, not all the mapping information is required. Computing time and memory are wasted to generate unnecessary information; instead, alignment-free methods only generate necessary information with less computational resources. Recent alignment-free ideas that have made significant application improvements include pseudo-alignment (Bray et al. 2016), lightweight alignment (Patro et al. 2017), and quasi-mapping (Srivastava et al. 2016). An alignment-free approach has also been applied to SNP genotyping by (Shajii et al. 2016). They introduce a SNP genotyping tool named LAVA, which builds an index from known SNPs (e.g. dbSNP) and then uses approximate k-mer matching to genotype the donor from sequencing data. LAVA is reported to perform 4-7 times faster than a standard mapping-based genotyping pipeline, while achieving comparable accuracy.

A Bloom filter is a space efficient data structure for representing sets that occasionally provides false positive answers to membership queries (Bloom 1970; Broder and Mitzenmacher 2004). In applications where false positives are acceptable, Bloom filters can help to greatly improve scalability. They have been used in the context of indexing, compressing and searching whole genome datasets (Rozov, Shamir, and Halperin 2014), and large sequence databases (Solomon and Kingsford 2016, 2017; Sun et al. 2017).

In this paper, we propose an improvement to LAVA called VarGeno, which performs 2-8 times faster than LAVA with similar memory usage. VarGeno uses Bloom filters to achieve a 2x speedup without changing the accuracy. An 8x speedup is achieved by using a quality value filtering method (VarGeno-QV), at the cost of a slight decrease (0.04%) in accuracy. VarGeno-QV uses information from the base-quality scores (Cock et al. 2009).

2 Background

In this section, we introduce the method used by LAVA, since it forms the basis of VarGeno. Figure 1 illustrates the pipeline of LAVA, which contains two main modules: the dictionary generation module (i.e. the pre-processing module), and the main genotyping module. At a high-level, the pre-processing module takes as input a reference genome (e.g. hg19) and a list of known SNPs that need to be genotyped (e.g. dbSNP, or a disease related SNP database). Each SNP is a tuple of a position in the reference genome and an alternate allele. The pre-processing modules builds several dictionaries (i.e. indices), as described below. The genotyping module takes as input a collection of reads from a donor

that needs to be genotyped, as well as the indices. The genotype caller outputs, for each SNP in the SNP list, whether it exists in the donor and, if yes, its heterozygosity.

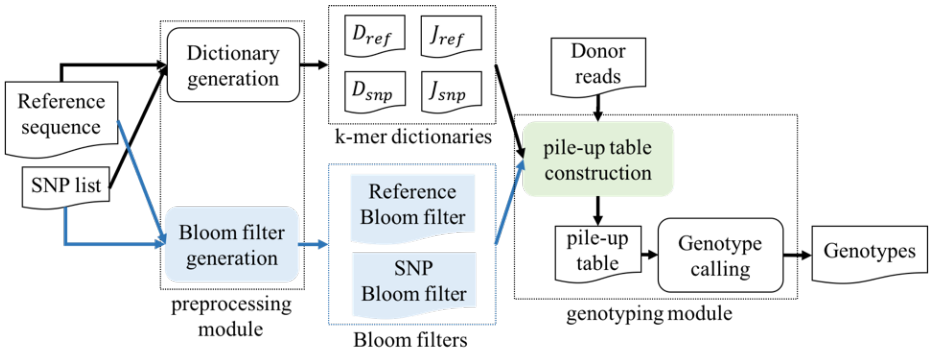

Figure 1. Comparison of LAVA and VarGeno pipelines. Modules and files in black are part of both pipelines and in blue are unique to the VarGeno pipeline. The green module is shared by both pipelines but with different implementations.

LAVA relies on a carefully chosen hard-coded value of $k = 32$ for its *k*-mer dictionaries, which we also adopt in this paper. A 32-mer can be directly encoded into 64-bits (which we denote as the *encoded k-mer*) and the theoretical probability that more than one error exists within a single 32-mer is acceptably low.

LAVA's dictionary generation module generates a reference dictionary D_{ref} and SNP dictionary D_{snp} by preprocessing the reference genome and the known SNP list. D_{ref} is an array of <encoded k-mer, genome position> tuples, sorted in increasing order of encoded k-mers. D_{ref} contains a tuple for every position of the reference genome, but k-mers with undefined bases (Ns) are not included. LAVA also constructs a secondary indexing hash table J_{ref}. J_{ref} maps a 32-bit unsigned integer u to the first location in D_{ref} at which there is an encoded k-mer whose upper 32 bits are u. To query an encoded *k*-mer, LAVA first queries J_{ref} to find the start and end indices of the block in D_{ref} with the same upper 32 bits as the query. Then LAVA does a binary search through this block to find the encoded k-mers whose lower 32 bits match the query (Figure 2). The reference positions where this k-mer occurs are then returned.

LAVA constructs a SNP dictionary D_{snp} and a secondary index J_{snp} in a similar fashion. D_{snp} is built from k-mers from positions that overlap some SNP from the SNP list, with the reference allele replaced by alternate allele. In addition to the encoded k-mer and genome position, D_{snp} also stores some annotations about each SNP. Finally, J_{ref} hashes the upper 24-bits, not 32-bits, since the size of D_{snp} is much smaller than D_{ref}.

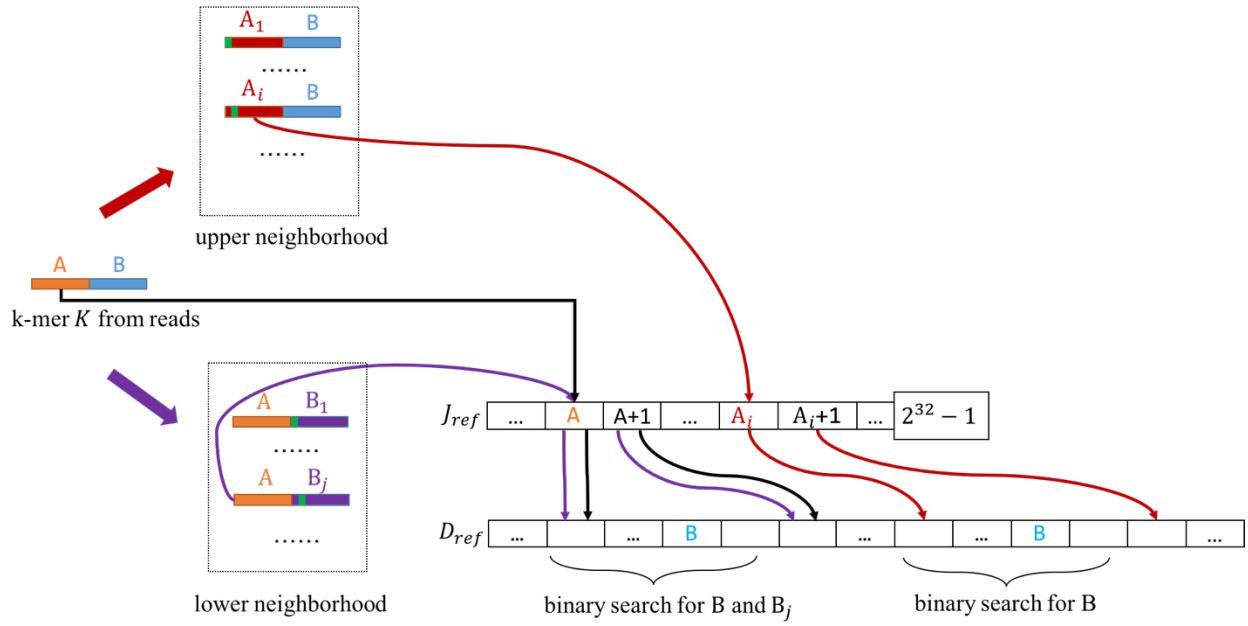

Figure 2. The structure of LAVA's reference dictionaries D_{ref} and D_{ref} and the process of querying. Let A be the upper 32 bits of an encoded k-mer K and let B be the lower 32 bits. The black arrows illustrate the process of querying K; the purple arrows illustrate the process of querying the lower neighborhood of K; the red arrows illustrate the process of querying the upper neighborhood of K.

To genotype the SNPs in a set of donor reads, LAVA first generates a pileup table. For each SNP in the SNP list, the table contains the number of reads supporting the reference and alternate alleles, respectively. To generate the pileup table, LAVA processes each read independently. Each read is first split into non-overlapping consecutive k-mers. The Hamming neighborhood of distance 1 for a k-mer K, denoted by $\mathrm{N(K)}$, is the set of all k-mers with a Hamming distance at most 1 to K. We refer to $\mathrm{N(K)}$ as the *neighborhood* of K, for short. Notice that $\mathrm{K} \in \mathrm{N(K)}$ and $|\mathrm{N(K)}| = 3k + 1$. For each of a read's generated k-mers as well as their reverse complements, LAVA queries D_{ref} and D_{snp} for every element of $\mathrm{N(K)}$. The neighborhood is used in order to account for the possibility of one erroneous nucleotide in K (recall that k is chosen so that the probability of more than one error is low). The genome positions of the matching k-mers are then used to identify the locus and allele for which the read has the strongest support, if any. After the pileup table is constructed, LAVA's genotyping module uses it to output a genotype for each SNP in the SNP list: homozygous reference, heterozygous, and homozygous alternate.

3 Methods

The run time of genotyping is dominated by querying all the k-mers in $\mathrm{N(K)}$ for every K in every read. Our immediate goal in optimizing LAVA is to reduce the number of unnecessary queries. Here we present our improvements to reduce the number of unnecessary queries.

The time performance of the querying process depends on the k-mer being queried. The neighborhood $\mathrm{N(K)}$ can be partitioned into three subsets: 1) the original k-mer K, 2) the *upper neighborhood* of K, which is the set of k-mers whose encoding differs with K in the upper 32 (respectively, 24) bits when querying D_{ref} (respectively, D_{snp}), and 3) the *lower neighborhood* of K, which is the set of k-mers

whose encoding differs with K in the lower 32 (respectively, 40) bits when querying D_{ref} (respectively, D_{snp}). The querying process can also be divided into three sub-process, according to which k-mers in N(K) are queried (Figure 2). Each k-mer in the upper neighborhood of K will have a different upper 32 bits from K and will require a separate access to J_{ref} and one random access to D_{ref}. This will likely result in a cache miss for every k-mer in the upper neighborhood. On the other hand, all the k-mers in the lower neighborhood will share the same upper 32 bits and hence will all query the same block of D_{ref}. Each block typically fits in a cache line, hence the query of k-mers in the lower neighborhood is unlikely to generate any cache misses. The same logic applies to queries of D_{snp}.

3.1 Upper neighborhood queries

VarGeno applies a Bloom filter to improve the performance of querying the upper neighborhood. A Bloom filter for representing a set $\mathrm{S} = \{x_1, x_2, \dots, x_n\}$ of n elements contains a bitvector of size m, and p independent hash functions $h_1, h_2, \dots, h_p$. Each hash function maps each k-mer to a random integer uniformly between 0 and $\mathrm{m} - 1$. The bitvector is initialized to an array of zeros. For each element $x \in \mathrm{S}$, the bits $h_i(x)$ of the bitvector are set to 1 for $1 \leq i \leq p$. To check if an item y is in S, we check whether all $h_i(y)$ are set to 1. If not, then y is not a member of S. Otherwise, y is a member of S with a small false positive rate (Broder and Mitzenmacher 2004).

VarGeno's preprocessing module builds Bloom filters BF_{ref} (respectively, BF_{snp}) where the elements being represented are the lower 32 bits of all k-mers in D_{ref} (respectively, the lower 40 bits of all k-mers in D_{snp}). Before searching any k-mers in the upper neighborhood of k-mer K, VarGeno's genotyping module first checks if K's lower bits exist in BF_{ref}. If not, then the upper neighborhood k-mers are not queried, as they will not exist in D_{ref}. Since each upper neighborhood query endures a likely cache miss, avoiding such queries provides substantial running time improvements.

3.2 Lower neighborhood queries

VarGeno also changes how lower neighborhood queries are performed. The size of a lower neighborhood for a D_{ref} query is 48 k-mers. To improve the performance, we observe that doing that many binary searches within one small block is inefficient, and, instead, a linear scan of the block using a direct computation of the Hamming distance to the each lower neighborhood k-mer can be faster. Figure 3 illustrates the strategy. VarGeno uses a fast bitwise routine to determine whether two k-mers are within a Hamming distance of one, and, if so, where the differing position is. We provide its details in the Supplementary Information.

If the block size is too large, then the linear scan might be slower than the original binary search approach. VarGeno includes a size threshold—if the block size is smaller than the threshold, then a linear scan is used; otherwise, the original binary search based method is used. The following Observation shows that the number of large blocks is small:

Observation 1. Let n be the number of distinct k-mers stored in a dictionary D and let b be the number of blocks in D. Under the assumption that the encoded k-mers are independent from each other, the size of a block in D is at least t with probability of at most $\frac{n}{bt}$.

Proof. Let x_i be the size of block i in D. Under the independence assumption, x_i follows a Binomial distribution with n trials and success probability of $\frac{1}{b}$. The expected size of block i is therefore $\mathrm{E}(x_i) =$

$\frac{n}{b}$. Applying Markov's inequality (Buot 2006), the probability that x_i is at least t is $\mathrm{P}(x_i \geq \mathrm{t}) \leq \frac{E(x_i)}{t} = \frac{n}{bt}$.
■

The assumption that encoded k-mers are independent from each other is not true in our data, since many of the k-mers overlap. However, we argue that blocks of two encoded k-mers are much less dependent. Our bit encoding of k-mers is the natural one, representing the nucleotides with 2 bits each, in the same order as they appear in the k-mer. The encodings of two overlapping k-mers are shifted with respect to each other, which results in different higher-order bits and, hence, blocks.

Based on the size of D_{ref} for the datasets in (Shajii et al. 2016), we have that n is about 3 billion and b is 2^{32}. Using VarGeno's default size threshold of $t = 25$, the probability that a block is large is less than 0.028. Thus, the Observation estimates that VarGeno resorts to the binary search method for less than 3% of the blocks.

3.3 Quality score optimization

VarGeno further accelerates the genotyping process by utilizing the quality scores at each nucleotide to avoid generating the full neighborhood of each k-mer. The reason for generating all neighborhood k-mers is to account for all possible instances of a sequencing error that could be present. However, the donor reads have a Phred quality score associated with each nucleotide, which is an estimate of the probability of a nucleotide error. Nucleotides with high quality scores are unlikely to be erroneous.

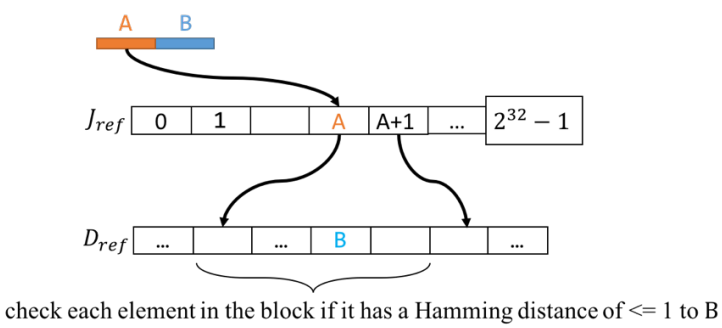

Figure 3. Using linear scanning to accelerate lower neighborhood querying.

Based on the above observation, VarGeno pipeline also provides an alternate option called VarGeno-QV. In VarGeno-QV, if the quality score for a certain position within a k-mer K is higher than a threshold, then the neighborhood k-mers which differ at this position are then skipped and not queried against the dictionaries during the genotyping stage. While we expect a time speed-up from this optimization, quality scores are not perfect estimates and we also expect the accuracy to decrease compared to VarGeno.

4 Results

We implemented VarGeno and VarGeno-QV in C++, building on the LAVA code base (Shajii et al. 2016) and code from (Sun et al. 2017). VarGeno and VarGeno-QV implementation is freely available at: https://github.com/medvedevgroup/vargeno. We compared accuracy, running time and memory usage against LAVA and LAVA-Lite (run in default parameters). LAVA-Lite is a lower-memory version of LAVA

that sacrifices some accuracy (Shajii et al. 2016). We also compared against a typical alignment-based discovery pipeline, denoted by BWA+mpileup, which was reported to have the highest number of SNP genotyping correct calls in (Shajii et al. 2016). This ran 'bwa-mem' (Li and Durbin 2010), followed by 'samtools mpileup' (Li et al. 2009), followed by 'bcftools call –gf' (Narasimhan et al. 2016). Default parameters were used. VarGeno fixes the number of hash function in the Bloom filter to be 1, to reduce the hashing time. All experiments were run on an Intel Xeon CPU with 512 GB of RAM and using single core (at 2.10 GHz).

4.1 Dataset

We used the same input datasets as in the LAVA paper. For the donor data, we used NA12878 reads from Phase 1 of the 1000 Genome Project (1000 Genomes Project Consortium et al. 2012). The dataset contains reads with length 101, and the depth of coverage is around 6X. For the SNPs list, we used all common SNPs from dbSNP (12,346,254 SNPs; build 142). We used GRCh37/hg19 as the reference sequence.

4.2 Validation

For validation, we used an up-to-date high-quality genotype annotation generated by the Genome in a Bottle Consortium (GIAB) (Zook et al. 2014). The GIAB gold standard contains validated variant genotype information for NA12878, from 14 sequencing datasets with five sequencing technologies, seven read mappers and three different variant callers. To measure accuracy, we counted SNPs from the SNP list database for which genotypes were present in the gold standard. Let X denote this set of SNPs. We defined the accuracy of a tool as the proportion of SNPs in X that were correctly genotyped by the tool. SNPs in X for which a tool did not produce genotype information were treated as incorrectly genotyped.

4.3 Effect of upper neighborhood query optimization

Table 1 shows the result of applying the Bloom-filter optimization of Section 3.1. It reduces the run time by 46% compared to LAVA, at the expense of only a 2% increase in memory usage. Note that this optimization has no effect on the output, and hence we do not compare the accuracy. We also measure the effect of varying the size of the Bloom filter (denoted by m). A larger size decreases the false positive rate and hence the number of unnecessary queries to the dictionaries; a smaller size decreases the memory usage. VarGeno's default setting is $m = 8n$, where n is the number of distinct values that are stored in a Bloom filter (these are pre-computed separately for BF_{ref} and BF_{snp}). This corresponds to a theoretical false positive rate of 0.118 (Broder and Mitzenmacher 2004). We also tried $m = 16n$, which corresponds to a theoretical false positive rate of 0.06. Our results indicate that there is not a significant change in running time or memory usage, relative to the totals.

	Running time (mins)	Memory usage (GB)
LAVA	315	59.7
Using Bloom filters ($m = 8n$)	171	60.9
Using Bloom filters ($m = 16n$)	170	62.1

Table 1. The effect of using Bloom filters to accelerate genotyping.

4.4 Effect of lower neighborhood query optimization

Next, we measured the effects of the optimizations proposed in Section 3.2. Using the linear scan optimization with default parameters resulted in an improvement of 38.5% to the run time. We also measure the effect of the block size cutoff. Figure 4 illustrates the performance as a function of different

block size thresholds. Performance drastically improves as long as threshold is at least ten. The performance is not substantially impacted by further increasing the threshold.

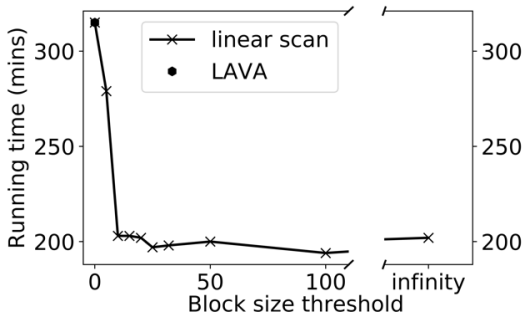

Figure 4. Speedup due to optimization of lower neighborhood queries, depending on the block size thresholds used. A block size threshold of zero is equivalent to what LAVA does. If threshold is infinity, a linear scan is applied to every block.

4.5 Overall VarGeno performance

The above optimizations are combined in VarGeno. Table 2 compares its performance, using default parameters, to LAVA, LAVA Lite and BWA+mpileup. VarGeno's run time is only 43% of LAVA, at only a cost of a 2% increase in memory. Note that the output is the same and hence the accuracy is not affected. Compared to BWA+mpileup, VarGeno is more than 12 times faster, but at the cost of using 20 times more memory. However, BWA+mpileup is not as accurate as VarGeno or LAVA, which is to be expected since it is designed for the more difficult task of discovery and not just genotyping.

	Running time (mins)	Memory usage (GB)	Accuracy (%)
BWA+mpileup	1800	3.2	72.16
LAVA	315	59.7	76.65
LAVA Lite	464	33.0	76.65
VarGeno	135	60.9	76.65
VarGeno-QV ($c = 39$)	39	60.9	76.61

Table 2. Performance of VarGeno and VarGeno-QV.

4.6 VarGeno-QV performance

VarGeno uses a quality value cutoff (denoted by *c*), so that it does not generate neighbors at positions with a Phred quality score more than *c*. We investigated the effect of *c* on VarGeno-QV, by trying all the Phred quality scores, which are integers in the range of $[0,42]$ (Figure 5). When $c = 42$, VarGeno-QV behaves exactly as VarGeno. At $c = 41$, it gives the most conservative improvement, only avoiding generating neighbors at the sites with the absolutely highest quality. At $c = 0$, the fastest run time is achieved (13 mins) but the accuracy decreases to 75.76. An appealing tradeoff is achieved at $c = 39$, with a run time of 39 mins and an accuracy of 76.61%, nearly identical to VarGeno (Table 2). With this setting, VarGeno-QV is more than 8 times faster than LAVA and more than 3 times faster than VarGeno.

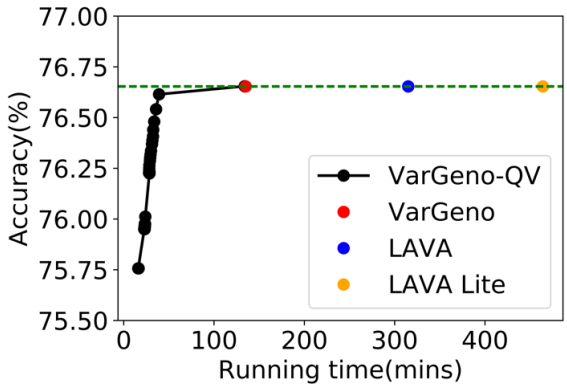

Figure 5: The performance of VarGeno-QV under different quality value cutoffs. The green line indicates the accuracy achieved by LAVA and VarGeno.

4.7 Pre-processing

In our experiments, the preprocessing time, which includes the time to generate dictionaries and Bloom filters by LAVA and VarGeno, was not counted. Since the pre-processing module is executed only initially and then only when the SNP list is updated, its performance is not as crucial. But, for the sake of completeness, Table 3 shows the preprocessing time of LAVA's dictionary generation and the time/memory required by VarGeno's additional Bloom filter generation step.

	Preprocessing time (mins)	Memory usage(GB)
Dictionary generation (LAVA)	52	70.6
Bloom filter generation (VarGeno)	13	6.2

Table 3. Preprocessing time and memory usage.

5 Conclusions

In this paper, we presented VarGeno and VarGeno-QV, an alignment-free SNP genotyping method. We demonstrated that it is 2x – 8x faster than LAVA, a state-of-the-art alignment-free variant genotyping method, with similar memory usage. Our method integrates three general techniques to accelerate alignment-free variant genotyping: (1) Bloom filters, (2) linear search with fast bitwise comparisons, and (3) base-quality score filtering. While (1) and (3) are already broadly applied, there are many *k*-mer based methods where (2) can still be harnessed to improve running times.

VarGeno is a streaming algorithm: it can process reads on-the-fly as they come off a sequencer. This is especially useful for variant genotyping scenarios where time is crucial, such as in clinical applications. For instance, in our experiment, VarGeno-QV can genotype 12 million variants from 6x whole genome sequencing data in 40 minutes. VarGeno can be applied more widely to portable medical devices, if the memory usage is further reduced. One possible way to achieve this, at the cost of running time, is to process the reference in separate chunks. Techniques to further reduce memory usage are a future research direction.

Acknowledgements

This work has been supported in part by NSF awards DBI-1356529, CCF1439057, IIS-1453527 to PM.

Conflict of Interest: none declared.

References

1000 Genomes Project Consortium, The 1000 Genomes Project et al. 2012. "An Integrated Map of Genetic Variation from 1,092 Human Genomes." *Nature* 491(7422):56–65.

Bloom, Burton H. 1970. "Space/time Trade-Offs in Hash Coding with Allowable Errors." *Communications of the ACM* 13(7):422–26.

Bray, Nicolas L., Harold Pimentel, Páll Melsted, and Lior Pachter. 2016. "Near-Optimal Probabilistic RNA-Seq Quantification." *Nature Biotechnology* 34(5):525–27.

Broder, Andrei and Michael Mitzenmacher. 2004. "Network Applications of Bloom Filters: A Survey." *Internet Mathematics* 1(4):485–509.

Buot, Max. 2006. *Probability and Computing: Randomized Algorithms and Probabilistic Analysis*.

Cock, Peter J. A., Christopher J. Fields, Naohisa Goto, Michael L. Heuer, and Peter M. Rice. 2009. "The Sanger FASTQ File Format for Sequences with Quality Scores, and the Solexa/Illumina FASTQ Variants." *Nucleic Acids Research* 38(6):1767–71.

Depristo, Mark A. et al. 2011. "A Framework for Variation Discovery and Genotyping Using next-Generation DNA Sequencing Data." *Nature Genetics* 43(5):491–501.

Garrison, Erik and Gabor Marth. 2012. "Haplotype-Based Variant Detection from Short-Read Sequencing." *arXiv Preprint arXiv:1207.3907*.

Hirschhorn, Joel N. and Mark J. Daly. 2005. "Genome-Wide Association Studies for Common Diseases and Complex Traits." *Nature Reviews Genetics* 6(2):95–108.

LaFramboise, Thomas. 2009. "Single Nucleotide Polymorphism Arrays: A Decade of Biological, Computational and Technological Advances." *Nucleic Acids Research* 37(13):4181–93.

Li, Heng et al. 2009. "The Sequence Alignment/Map Format and SAMtools." *Bioinformatics* 25(16):2078–79.

Li, Heng and Richard Durbin. 2010. "Fast and Accurate Long-Read Alignment with Burrows-Wheeler Transform." *Bioinformatics* 26(5):589–95.

Luikart, Gordon, Phillip R. England, David Tallmon, Steve Jordan, and Pierre Taberlet. 2003. "The Power and Promise of Population Genomics: From Genotyping to Genome Typing." *Nature Reviews Genetics* 4(12):981–94.

McKenna, Aaron et al. 2010. "The Genome Analysis Toolkit: A MapReduce Framework for Analyzing next-Generation DNA Sequencing Data." *Genome Research* 20(9):1297–1303.

Narasimhan, Vagheesh et al. 2016. "BCFtools/RoH: A Hidden Markov Model Approach for Detecting Autozygosity from next-Generation Sequencing Data." *Bioinformatics* 32(11):1749–51.

Pastinen, Tomi et al. 2000. "A System for Specific, High-Throughput Genotyping by Allele-Specific Primer Extension on Microarrays." *Genome Research* 10(7):1031–42.

Patro, Rob, Geet Duggal, Michael I. Love, Rafael A. Irizarry, and Carl Kingsford. 2017. "Salmon Provides Fast and Bias-Aware Quantification of Transcript Expression." *Nature Methods*.

Rozov, Roye, Ron Shamir, and Eran Halperin. 2014. "Fast Lossless Compression via Cascading Bloom

Filters." *BMC Bioinformatics* 15.

Shajii, Ariya, Deniz Yorukoglu, Yun William Yu, and Bonnie Berger. 2016. "Fast Genotyping of Known SNPs through Approximate K-Mer Matching." *Bioinformatics* 32(17):i538--i544.

Solomon, Brad and Carl Kingsford. 2016. "Fast Search of Thousands of Short-Read Sequencing Experiments." *Nature Biotechnology* 34(3):300–302.

Solomon, Brad and Carl Kingsford. 2017. "Improved Search of Large Transcriptomic Sequencing Databases Using Split Sequence Bloom Trees." Pp. 257–71 in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10229 LNCS.

Srivastava, Avi, Hirak Sarkar, Nitish Gupta, and Rob Patro. 2016. "RapMap: A Rapid, Sensitive and Accurate Tool for Mapping RNA-Seq Reads to Transcriptomes." *Bioinformatics* 32(12):i192–200.

Sun, Chen, Robert S. Harris, Rayan Chikhi, and Paul Medvedev. 2017. "Allsome Sequence Bloom Trees." Pp. 272–86 in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10229 LNCS.

Syvänen, Ann-Christine. 2005. "Toward Genome-Wide SNP Genotyping." *Nature Genetics* 37:S5--S10.

Vinga, Susana and Jonas Almeida. 2003. "Alignment-Free Sequence Comparison-a Review." *Bioinformatics (Oxford, England)* 19(4):513–23.

Zook, Justin M. et al. 2014. "Integrating Human Sequence Data Sets Provides a Resource of Benchmark SNP and Indel Genotype Calls." *Nature Biotechnology* 32(3):246–51.