A Portable GPU Framework for SNP Comparisons

Elliott Binder, Tze Meng Low
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA, USA
{ebinder, lowt}@cmu.edu

Doru Thom Popovici
Lawrence Berkeley National Laboratory
Berkeley, CA, USA
dtpopovici@lbl.gov

***Abstract*—With recent improvements in DNA sequencing technologies, the amount of genetic data available for analysis has grown rapidly. The increasing size of datasets has created a demand for high performance implementations capable of processing and analyzing data in a timely manner. In addition, rapid growth in genetic data has also led to the development of more accurate analysis techniques used in DNA forensics and law enforcement. At the heart of some analyses is the comparison of single nucleotide polymorphisms (SNPs) to detect the absence and/or presence of minor alleles, which has been shown to be similar to matrix-matrix multiplication from the domain of dense linear algebra. This similarity suggests that SNP comparison is embarrassingly parallel and may perform well on GPUs. In this paper, we present a portable GPU framework that allows us to leverage the CPU algorithm on GPUs to perform SNP comparisons. We demonstrate that with minor parameter changes to the framework, SNP comparison can be ported onto a variety of GPU platforms from both AMD and NVIDIA. In addition, we provide a model for defining the new parameters for a given GPU. Finally, we demonstrate performance portability across multiple GPU architectures where end-to-end (data transfer + computation) execution time is between 47% and 677% faster than a CPU implementation that is close to the theoretical peak of the CPU, and the kernel execution attains between 55% to 97% of the theoretical peak throughput of each specific GPU architecture.**

***Index Terms*—linkage disequilibrium; population genetics; dense linear algebra; matrix multiplication; GPUs; FastID**

I. Introduction

Advances in sequencing technology have made available a large and increasing amount of genetic data, which in turn has facilitated advancements in population genetic studies of human diseases identification [1] and more effective and personalized drug treatment [2] through genome-wide association studies. Beyond population genomics studies, law enforcement and forensics communities can also benefit from the improvement in sequencing technology through the use of an increasing number of single nucleotide polymorphisms (SNP) per forensic sample. Greater amounts of DNA data paired with increasing number of identities and forensic samples collected (on the order of tens of millions profiles) in multiple law enforcement databases can facilitate more detailed analysis with increased accuracy [3]–[7].

The essence of the computation for both computational biology and law enforcement is in the identification of the presence or absence of minor alleles at each SNP site. In addition, the expected increase in data availability suggest that high throughput implementations for performing SNPs comparisons are vital for efficient analysis.

Graphical processing units (GPUs), conventionally known to perform matrix-matrix multiplication very efficiently, possess a large number of special functional units that can perform the population count operation [8], [9], an important sub-operation for SNP comparison. In addition, linkage disequilibrium (LD) [10], a fundamental computation in population genomics that requires SNP comparisons, can be cast in term of specialized dense linear algebra operations where the specific entries of the vectors and matrices are 0s and 1s [11].

These observations have been exploited in current work that suggests using GPUs for SNP comparisons [12]. However, such work typically focuses on a specific application domain, and on the use of a specific GPU platform. They do not address the portability of their approach across different problem domains or different GPU platforms, especially across GPUs from different hardware vendors. In addition, GPU frameworks that support portability across different GPU platforms from the same vendor (e.g. CUBLAS [13] and CUTLASS [14]) are restricted to traditional dense linear algebra computation. Porting this platform to other application domains in order to perform computations such as SNP comparison is currently not supported. A third limitation is that existing high performance libraries for population-based analysis such as PLINK [15] do not support the use of GPUs.

In this work, we propose an OpenCL framework that facilitates the computation of SNP comparison on different GPU platforms. Specifically, we show that the CPU algorithm for SNP comparison is amenable for computation on GPUs. We specialized the OpenCL framework for higher performance by allowing users to configure the framework through the use of a configuration header file. Finally, we describe how the values for the configuration parameters can be obtained from the hardware features of the GPU platforms.

Contributions.

The specific contributions of this paper are as follows:

– Demonstration of the feasibility of performing SNP comparison using a CPU-based algorithm on general purpose GPU architectures, where we show that the end-to-end execution time (i.e. inclusive of data transfer) on the GPU is still lower than the execution time on the CPU.
– A portable framework implementing SNP comparisons across GPUs with different micro-architectures and be-

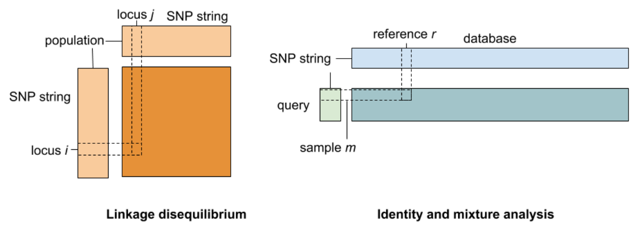

Fig. 1. Difference in matrix shapes and sizes. The two input matrices for LD are of the same size, whereas one matrix (the query) is often much smaller than the other (database) for FastID comparisons. Darker colors represent the output of the computation.

tween different vendors.
– A systematic approach identifying how software parameters characterizing the SNP comparison algorithm can be specialized for specific GPUs hardware features.

II. Algorithms Involving SNP Comparisons

In this section, we describe three algorithms, linkage disequilibrium, identity search and mixture analysis, all of which require SNP comparisons as a basis for their computation.

A. Linkage disequilibrium (LD)

Linkage disequilibrium, the non-random association of alleles at different loci, is a statistic of interest in genetic association and population studies, since alleles in a given population may change due to natural selection, demographic history, or other genetic factors. The degree at which alleles at different loci correlate may be a function of the underlying genetic processes that structure a population. Alleles at different loci are said to be in linkage equilibrium when the occurrence of one does not affect the occurrence of another, which is to say that the alleles are statistically independent and randomly associated. Linkage disequilibrium between alleles a and b, signified as D_{AB}, can be computed as the difference between the probability of both alleles occurring at a loci and the product of the probability of each allele occurring at that loci, or

$$D_{AB} = p_{AB} - p_A p_B$$

Alachiotis et. al. [11] showed that LD can be cast in terms of dense linear algebra, so the accumulated knowledge of the high performance computing (HPC) community can be leveraged to produce efficient implementations for computing LD. Where general matrix multiply (GEMM) is of the form $C = \alpha AB + \beta C$, and each output value γ can be implemented as a dot product in the form

$$\gamma = a^T b,$$

each element of the p_{AB} term of LD can be expressed as

$$\gamma = (a \,\&\, b)^T (a \,\&\, b) \tag{1}$$

B. FastID Identity Searching

Identity searching is the comparison of a suspect's DNA profile against a database of reference DNA profiles. FastID is a newly developed method for performing rapid and scalable analysis of forensic DNA samples through the use of SNP comparisons [16]. Specifically, the FastID method compares a small number of sample DNA sequences (known as the "query") against a known database of reference DNA profiles that have been collected. Typical DNA comparison against many DNA profiles is done in large datacenters, so providing fast and efficient implementations can reduce costs and facilitate investigations.

The suspect's DNA profile is compared against a reference sample by using the exclusive OR (XOR) operation, denoted as $\oplus$. Any differences between the samples will manifest themselves as set bits in the result. The number of set bits in the result is an indication of the likelihood that an input comes from the suspect. No set bits in the result signifies a positive match. Mathematically, identity search can be described as

$$\gamma = (a \oplus b)^T (a \oplus b), \tag{2}$$

where a and b are the binary vectors representing the input SNP profiles. In this case, greater γ values indicate that it is less likely that the individual is a match in the database.

C. FastID Mixture Analysis

Mixture analysis is the process of identifying the probability that an individual is one of the many individuals whose DNAs contribute to form the mixture of DNA profiles.

To perform this computation, the differences between the individual and the mixture profile are first identified with the exclusive OR (XOR) operation. The intermediate result is then intersected (using the logical AND operation) with the individual profile. The result represents minor alleles that are present in the individual profile but not in the mixture. This means that number of set bits in the result is inversely related to the likelihood that the individual is part of the mixture. Mathematically, we can express the mixture analysis as

$$\gamma = ((r \oplus m) \,\&\, r)^T ((r \oplus m) \,\&\, r) \tag{3}$$

where r is the reference profile and m is the mixture.

The above expression can further be simplified into the following expression:

$$((r \oplus m) \,\&\, r) \equiv r \,\&\, \neg m.$$

The importance of this simplification is that there exist instructions on certain CPU and GPU architectures, that can perform the negation of m as part of computing the logical AND operation. This reduces the number of instructions required, hence increasing the attainable throughput. Alternatively, on architectures that do not support the fused negation-logical AND operation, the mixture profile (m) can, in advance, be negated and stored in the database. In this second scenario, mixture analysis reduces down to the same computation as linkage disequilibrium.

$$
\begin{array}{cccc|c|cccccccc|l}
0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & \dots & 0 & 1 & 0 & 1 & \\
\vdots & & & & & & & & & & & & \vdots & \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & \dots & 1 & 1 & 0 & 0 & \\
\hline
0 & 1 & 0 & 0 & 1 & 0 & 1 & 0 & \dots & 1 & 1 & 0 & 1 & \text{Sample} \\
\hline
\vdots & & & & & & & & & & & & \vdots & \\
1 & 0 & 1 & 0 & 0 & 1 & 1 & 0 & \dots & 1 & 0 & 1 & 0 & \\
1 & 0 & 0 & 1 & 0 & 1 & 1 & 1 & \dots & 1 & 0 & 1 & 0 & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \Uparrow \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \text{padding} \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \Downarrow \\
 & & & & \text{SNP} & & & & & & & & &
\end{array}
$$

Fig. 2. Example SNP matrix of bitvectors representing the presence or absence of minor alleles where 1 represents the presence, and 0 represents the absence of the minor allele. Reproduced with permissions from [11].

D. *Comparisons of the algorithms*

While each algorithm relies on SNP comparison as its basic computation, the resulting matrix formulation of the problem will result in different input matrix shapes. Specifically, typical input matrices for linkage disequilibrium tend to be rectangular with a small inner dimension, while the dimensions of the output matrix are the large dimensions of the input matrices. However, the input matrices for identity search and mixture analysis are typically asymmetric in that one matrix (the database of DNA profiles) is typically significantly larger than the other (query) matrix. This difference in input shape is highlighted in Figure 1.

The shapes of the input matrices are important since it has been shown that different matrix-matrix multiplication algorithms are optimized for different input matrix sizes [17], [18]. The implication of the differences in input matrix shapes is that the proposed portable framework must handle both cases adequately.

III. High Performance LD on CPUs

High performance linkage disequilibrium on the CPU can be computed by casting the operation as a sequence of dense linear algebra operations, where the bulk of the computation is performed as part of a matrix-matrix multiplication [11]. The authors also established the theoretical peak of SNP comparison on a modern CPU by highlighting that the performance bottleneck is the throughput of the population count operation, a specific instruction that counts the number of set bits in a word size (64 bits on a CPU).

In order to achieve high performance, this matrix-matrix multiplication is specialized in the following manner:

- Input SNPs are converted into binary matrices that have been packed, padded and stored as bit vectors (unsigned long integers). Major alleles are encoded as 0s while minor alleles (mutations) are captured as 1s in the binary matrices. Each SNP is represented as consecutive bit vectors. An example of such a matrix is given in Figure 2.
- Computation is also simplified by replacing the multiplication and addition with a sequence of three operations, logical and (&), population count(POPC), and integer addition (+), i.e.

$$\gamma_{i,j} = \gamma_{i,j} + \sum_k \texttt{POPC}(\alpha_{i,k} \ \& \ \beta_{k,j}),$$

instead of the usual

$$\gamma_{i,j} = \gamma_{i,j} + \sum_k \alpha_{i,k} \times \beta_{k,j}.$$

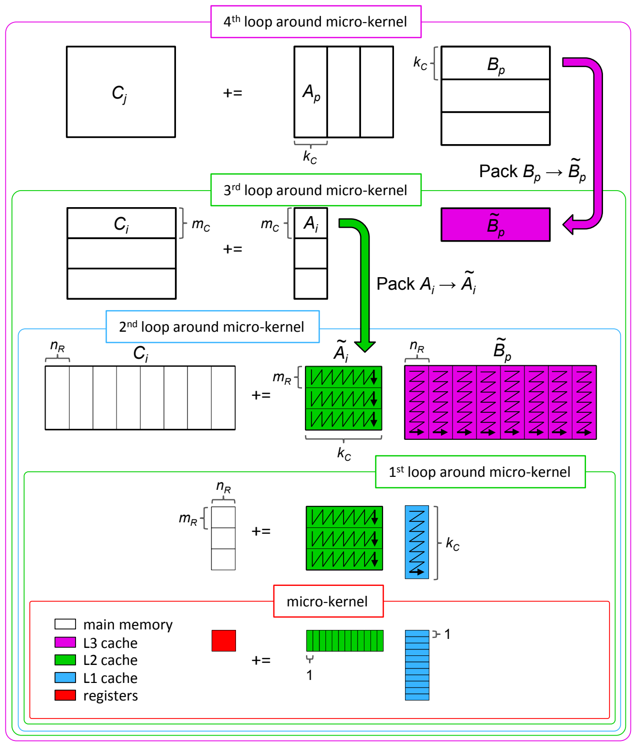
4th loop around micro-kernel
C_j
+=
A_p
k_C
B_p
Pack $B_p \rightarrow \tilde{B}_p$
3rd loop around micro-kernel
C_i
m_C
A_i
$\tilde{B}_p$
Pack $A_i \rightarrow \tilde{A}_i$
2nd loop around micro-kernel
n_R
$\tilde{A}_i$
m_R
1st loop around micro-kernel
micro-kernel
main memory
L3 cache
L2 cache
L1 cache
registers
1

Fig. 3. BLIS framework for instantiating high performance matrix matrix multiplication on traditional CPU architectures (Reproduced with permission from the authors [20]). Architecture-specific parameters are captured in a highly optimized micro-kernel and parameters for the loops around the micro-kernel.

In addition, Alachiotis et. al. showed that the BLAS-like instantiation framework (BLIS) [19] can be leveraged to easily incorporate these changes to obtain high performance LD implementations on the CPU. Specifically, the authors showed that these changes are restricted to the micro-kernel in the BLIS framework (Shown in Figure 3). By updating this micro-kernel, a parallel LD implementation that attains between 80-90% of the theoretical peak of the CPU can be obtained.

IV. SNP Comparisons on GPUs

Our approach to performing SNP comparison on the GPU is to map the BLIS framework onto the GPU. We start by first casting a model GPU hardware abstraction onto the CPU abstraction underlying BLIS. We then leverage the analytical models in Low et. al. [21] to determine software parameters that guide how the GPU kernel is to be written using the BLIS framework.

A. Model GPU architecture

In order for our framework to target different GPUs, it is first necessary to define a model GPU architecture that captures essential features of modern GPUs. Our assumptions about the model GPU are as follows:

- **Thread Group.** Computations on the GPU are performed using N_{grp} different groups of threads. Specifically, each thread group comprises of N_T threads that execute the same instruction at any given clock cycle. Thread groups are know as warps and wavefronts on NVIDIA and AMD GPUs, respectively.
- **Compute Cores.** A GPU is made up of N_C computational cores. Each core can perform its computation independent of other compute cores. These compute cores are also known as streaming multi-processors or compute units.
- **Compute Clusters.** Each compute core is organized in terms of N_{cl} clusters of computation. Each of these clusters can execute a group of threads independently, which implies that N_{cl} groups of threads can execute simultaneously on each core.
- **Arithmetic Units.** A compute cluster is comprised of multiple arithmetic units. For a given instruction, there exist N_{fn} arithmetic units that execute that specific instruction. It is assumed that each of these arithmetic units has a latency of L_{fn} cycles and can be effectively pipelined through the use of L_{fn} different thread groups. It is assumed that N_{fn} is different for different instructions. To differentiate between different numbers of arithmetic units, we use a superscript to denote the instructions. For example, N^{+}_{fn} is the number of functional units that can perform an addition. In addition, some of these arithmetic units may compute more complicated functions. We make the simplifying assumption that L_{fn} is the same for all instructions.
- **Shared Memory.** Each compute core is assume to have fast memory of N_{shared} bytes that is shared between all thread groups that execute on the same compute core. This fast memory is assume to be organized into N_b banks that can be accessed in parallel. However, simultaneous accesses to *different* elements in the same bank will cause a "bank conflict," resulting in a serialization of memory accesses and an associated drop in performance.
- **Load/Store Architecture.** It is assumed that data has to be loaded from memory before they can be computed on. In addition, each thread is assumed to be able to load and store N_{vec} elements at the same time.

A pictorial representation of our model GPU architecture is shown in Figure 4. It should be highlighted that an actual GPU architecture is more complicated and contains more hardware features than those captured by our model GPU architecture. However, additional features are not necessary to achieve high performance SNP comparison.

B. CPU / GPU abstractions

A commonly used abstraction for mapping GPUs to CPUs is the SIMD/SIMT abstraction [22]. This prevalent abstraction between CPU and GPU highlights the conceptual similarity between each thread group comprised of multiple threads that execute the same instruction (hence, single instruction multiple threads) and the single instruction multiple data (SIMD) instructions on the CPU, where each instruction performs the same operation on multiple pieces of data.

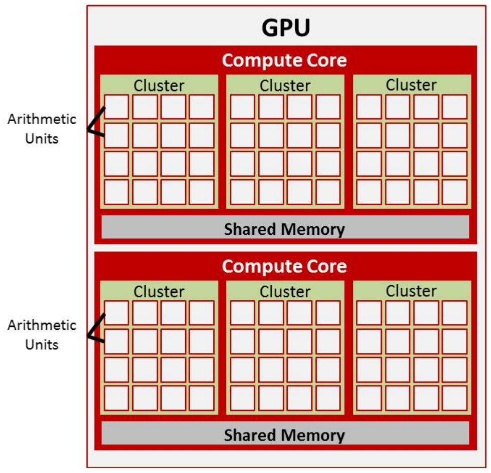

Fig. 4. Organization of the hardware features on the model GPU architecture. The GPU above can be characterized by the following parameters: $N_C = 2$, $N_{cl} = 3$, and $N_{fn} = 16$.

On a modern CPU, multiple SIMD instructions can be executed at the same time due to the presence of multiple SIMD functional units on the same CPU core. In addition, these SIMD functional units on the CPU achieve peak performance when they are pipelined. This means that sufficient number of independent SIMD instructions must be issued to fill the instruction pipelines to these functional units.

Since multiple thread groups can execute independently at the same time on different computational clusters within the same compute core, one could treat each computational cluster on the GPU as a SIMD functional unit on the CPU. This abstraction is enhanced when one considers pipelining of the SIMD functional unit. Recall that in order to hide the latency of the arithmetic units on the GPU, one has to rely on L_{fn} different thread groups. As thread groups are the equivalent of SIMD instructions, the equivalent of SIMD pipelining on the GPU can be achieved when the outputs of the L_{fn} different thread groups are independent.

The natural extension of the above abstractions is the abstraction that views the GPU compute core as the equivalent of the CPU core. Both CPU and GPU cores contain multiple independent SIMD functional units/compute clusters.

C. Mapping BLIS onto the GPU

Using the CPU/GPU abstraction presented previously, we now discuss how a GPU kernel for performing SNP comparison can be designed and implemented within the BLIS framework.

Recall that GPU cores are the conceptual equivalent of CPU cores. Within the BLIS framework, the cores partition both the second and third loops around the micro-kernel in a hierarchical fashion such that each core computes an independent $m_c \times n_r$ tile of C [23]. Similarly, we assign tiles of size $m_c \times n_r$ of C to each compute core on the GPU by parallelizing the second and third loops around the micro-kernel amongst the available GPU cores.

Each $m_c \times n_r$ tiles on a GPU core is further subdivided into smaller tiles that are computed by each thread group assigned to the GPU core. Recall that each GPU core is made up of compute clusters, each of which executes multiple thread groups. The $m_c \times n_r$ tiles is split into $m_r \times (n_r/L_{fn})$ sub-tiles, where each sub-tile is computed by multiplying two input matrices of sizes $m_r \times k_c$ and $k_c \times (n_r/L_{fn})$ together. Thread groups "resident" to a compute cluster are assigned sub-tiles in the same row, while thread groups that can execute simultaneously are assigned sub-tiles from the same column.

The distribution of GPU cores between the second and third loop is left as a parameter since different problems may require different distribution of the GPU cores. This is because problem sizes offer different amounts of parallelism in each of the dimensions. For example, in mixture analysis, the dimension corresponding to the number of profiles in the database would have more parallelism than the dimension of the queries. This would imply a more skewed distribution of the cores to best utilize the available parallelism.

V. Portable Framework and GPU Specialization

Our SNP comparison framework is built on the OpenCL [24] framework that allows us to easily port the framework between GPUs. The use of OpenCL also allows us to standardize the creation and initialization of the various supported OpenCL devices. Standard operations such as writing data from host memory to device memory, compute kernels that operate on said data, and reading results from device memory to host memory are handled in a platform-independent manner within our framework using OpenCL.

Specifically, our GPU framework implements the third loop (and its content) around the micro-kernel of the BLIS framework. This is implemented in terms of a parameterized GPU kernel that first loads a tile of A into shared memory, then computes assuming that A resides in shared memory while B is read from main memory. Unlike BLIS which requires a custom hand-tuned micro-kernel for every architecture, our GPU kernel is parameterized via C macros which are captured in a header file. Performance is attained on the various GPUs by specializing this GPU kernel based on the available hardware resources and the problem being computed. Specifically, only 4 values are required for us to configure the framework for different GPUs. These values are

$$m_c,\ m_r,\ k_c \text{ and } n_r,$$

which are the corresponding values required by the BLIS framework.

A. Mapping software configuration to hardware parameters

Each thread group on the GPU computes a output matrix of $m_r \times (n_r/L_{fn})$, stored in its local registers, by multiplying the input matrix A that is $m_r \times k_c$ by the input matrix B that is $k_c \times (n_r/L_{fn})$. Each thread in a thread group can load/store N_{vec} elements at the same time. This means that m_r (or n_r/L_{fn}) should be a multiple of N_{vec}. To maximize the reuse of the output matrix stored in registers, it is important to maximize k_c. For that reason, we choose to set

$$m_r = N_{vec}. \tag{4}$$

Recall that the block of A packed into shared memory is of size $m_c \times k_c$. As all thread groups will require A, and access to the same element of A by different thread groups will result in a degradation in performance due to bank conflicts, N_{cl} compute clusters must simultaneously access different banks of shared memory in order for all compute clusters to be able to perform their computation. This implies that at most

$$m_c = \frac{N_b}{N_{cl}} \tag{5}$$

elements can be accessed by each of the N_{cl} compute clusters without causing a bank conflict. In addition, as shared memory is N_{shared} bytes, and each element is (by default) 4 bytes, k_c can be computed as follows:

$$k_c = \frac{N_{shared}}{4N_b}. \tag{6}$$

Given that m_r elements are accessed by a thread, and each cluster (and hence each thread group) has access to m_c elements of the matrix A, this means that each thread group can be subdivided into m_c/m_r smaller subgroups of size

$$\frac{N_T m_r}{m_c}.$$

The size of this subgroup is important because it is also the minimum number of elements of the B matrix that is required to ensure that each thread computes a unique value of the output matrix. As each thread can load N_{vec} elements, this means that a total of

$$\frac{N_T m_r}{m_c} N_{vec}$$

elements of B is accessed by a single thread group. Finally, L_{fn} thread groups have to be assigned to each compute cluster to ensure that there are sufficient thread groups per compute cluster to hide the latency of the arithmetic units. This implies that the parameter, n_r can be computed using the formula

$$n_r \geq \frac{N_T m_r}{m_c} N_{vec} L_{fn} \tag{7}$$

Note that unlike the previous software parameters, Equation 7 is an inequality because this is highly dependent on the compiler use of the available registers on the GPU. Ideally, this value should be as large as possible without resulting in register spilling. In practice, we set the upper bound of n_r as the number of registers divided by the total number of threads used in each CPU core. By reducing the number of thread groups, the amount of available resource is increased, and will also result in a reduction in resource contention that could cause an associated drop in performance. This is consistent with the observation by Volkov [25] that lower occupancy can lead to higher performance.

It is important to note that while we provide the analytical formulas for mapping the software parameters to the GPU hardware features, these formulas are captured as part of the configuration header file. Users of the framework are expected to only identify the hardware features of the GPU.

B. Determining hardware parameters

While some of the hardware parameters required for our model GPU architecture are readily available on GPU specification sheets, we found that we had to manually benchmark the GPUs to identify the throughput and latency (L_{fn}) of the different instructions that are required for our kernels [1].

C. Instruction latency

To measure the latency of a given instruction, we write a simple program that consists of a long chain of dependent operations using the instruction. An operation is *dependent* on a previous operation if its input depends on the output of the previous operation. By creating a dependent chain of instructions, we can ensure that one instruction must wait for its previous instruction to complete before it can be issued, thus exposing the full latency of said instruction. Dependency chains are necessary to prevent superscalar processors from issuing multiple instructions, thus overlapping the latency of multiple instructions and providing us with incorrect latency-per-instruction figures. In practice, when writing such a microbenchmarking program, it may be necessary to use some source value from memory (or to store results of the operation to memory) to persuade the compiler not to optimize the code in cases where the operations can be simplified statically (or remove code that accomplishes no work). A loop can be placed around the dependent chain to execute the chain many times and diminish any effects of launching the kernel or loading values from memory. Also, increasing the number of instructions in the loop body will diminish the effects of managing the loop (for example, instructions executed to increment a loop counter, comparison on an induction variable, or performing a conditional branch). For example, a dependent chain to measure the latency of population count would look like

```
unsigned int temp = Array[thread_index];
for (int i = 0; i < MANY_ITERATIONS; i++) {
    temp = popcount(temp);
    temp = popcount(temp);
    ...
}
Array[thread_index] = temp;
```

where each thread operates on its own value.

[1]NVIDIA release instruction throughput for their architectures [9] which we verified through microbenchmarking. We were unable to locate documentation for relevant instruction throughput on Vega 64, so we determined the theoretical peak solely through microbenchmarking.

Executing the kernel with one thread group is sufficient to measure instruction latency[2]. Latency can be calculated as

$$\frac{\text{clock frequency} \times \text{execution time}}{\text{\# of instructions}}$$

where the number of instructions is taken from the microbenchmarking program as the product of the number of instructions in the loop body and the number of iterations the loop is executed.

D. Instruction throughput

To measure throughput, we can use the same program as before, but change the number of thread groups that execute the program and compute throughput as

$$\frac{\text{\# of instructions} \times N_T \times N_{grp}}{\text{clock frequency} \times \text{execution time}}$$

For the architectures we targeted, we expect the execution time to remain nearly constant for $N_{grp} \leq N_{cl}$ since thread groups are assigned to a compute cluster and functional units in unoccupied compute clusters remain idle. From the analytical model, using $N_{grp} = N_{cl} \times L_{fn}$ is sufficient for achieving peak throughput. We expect additional thread groups will not improve throughput since the latency between dependent instructions is covered by other thread groups and the functional units are fully saturated.

Some operations may be difficult to benchmark alone due to compiler optimizations (for example, a sequence of ANDs on the same value can be reduced to one, or a sequence of ADDs can be replaced by a multiply). Microbenchmarking each kernel (LD, FastID) was sufficient to determine what peak throughput would be. It is also possible to combine instructions that are not so easily reducible (e.g. c = popcount(a & c); or c += a & c;). Combining different instructions can expose which instructions share functional unit pipelines. For example, we observed that population count is on a separate pipeline from integer math operations since execution time remained nearly constant when exclusively performing population count and when simultaneously performing population count with an equal number of arithmetic operations. Instructions that share a pipeline reduce the effective throughput of each instruction since the resource must be shared. For example, on the Vega 64 the addition and logical AND operations fall on the same pipeline which becomes the bottleneck. The peak throughput per functional unit can be determined by identify the bottleneck (i.e. the minimum throughput on all pipelines in use).

Instead of running any comparable program on a CPU, we report the calculated theoretical peak that would be achievable or use execution time reported in [11]. Even though the clock speed on CPUs is typically higher, fewer functional units return lower theoretical peak throughput for the entire device.

[2]According to the model GPU architecture, thread groups are pipelined on the functional units within a compute cluster. Executing the microbenchmark with more than enough thread groups to saturate the pipelines on all compute clusters will likely increase the execution time and produce misleading latency figures.

E. “Limitations” of OpenCL

It is generally acknowledged that OpenCL may sacrifice performance for portability. However, our experience showed that some of these limitations of OpenCL can be reduced with our approach.

OpenCL determines a maximum value of thread groups, given by the parameter `CL_KERNEL_WORK_GROUP_SIZE`, that can be instantiated for each kernel. However, this limit in the number of thread groups is not a limitation for our implementation. This is because we limit the number of thread groups necessary to reside on a core (i.e. occupancy) to the product of the number of compute clusters and the latency of an arithmetic operation. This number of thread group is significantly less than the maximum number of thread groups allowed by OpenCL [3].

We observed that NVIDIA's OpenCL implementation reserved several bytes of shared memory instead of making the entire shared memory available for use by the kernels. This means that we can store a smaller tile of the input matrix in shared memory. However, recall that a $m_c \times k_c$ tile of an input matrix is packed into shared memory where k_c is determined by the amount of memory available (See Equation 6. Since the value of k_c is in the order of 100s, the impact of not having access to all of shared memory is minimized since the reduced shared memory means reducing k_c by 1. We encountered no such limitation to shared memory on the Vega 64.

VI. Results

We evaluated our approach using three different GPUs. Specifically, the NVIDIA Titan V, NVIDIA GTX 980, and AMD Vega 64 GPUs were chosen to demonstrate the portability of our proposed framework in providing high performance across generations of GPU micro-architectures and between GPUs from multiple hardware vendors. The specific hardware parameters related to the model GPU architectures are captured in Table I.

To motivate the use of GPUs for SNP-based comparisons, we also compare against the 12-core Intel workstation used in [11]. Even though modern processors have been released with updated micro-architectures (such as Intel's 9th generation Core series), the throughput of population count instructions on these architectures are only improved by increased core count and clock frequency.

A. Experimental Setup

1) Profiling: For our evaluation, we used OpenCL's event profiling to measure the elapsed time of kernel executions. End-to-end execution time was measured using the CPU's realtime clock. To hide the latency overhead of transferring data to and from the GPU, we implemented double buffering for the input and output matrices of the GPU. This also allows us to enqueue data transfer commands to be processed during computation, as well as allowing the CPU to pack inputs into one buffer while reading from another.

[3]The thread groups that are to be scheduled onto a particular compute core are a part of the same workgroup.

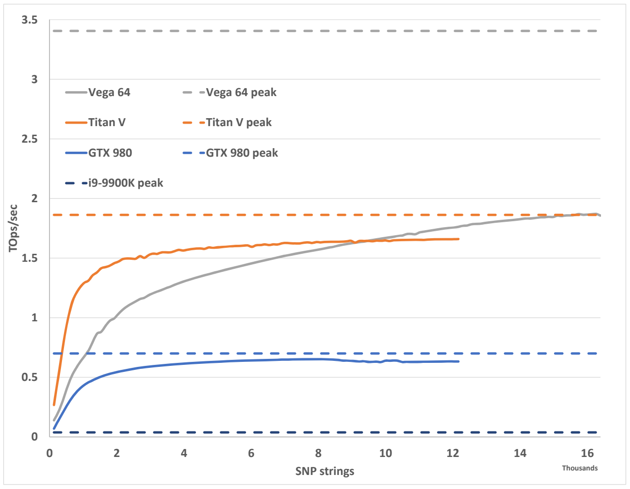

Fig. 5. LD kernel computation with near the maximum SNPs per device (Maxwell: 15,360; Volta: 25,600; Vega: 40,960) as the number of SNP strings increases to the device maximum (Maxwell/Volta: 12,256; Vega: 16,384).

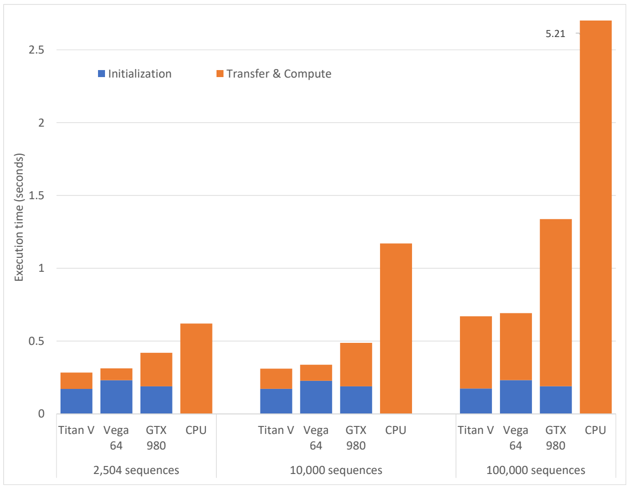

Fig. 6. End-to-end performance comparison of LD computation based on simulated datasets that consist of 10,000 SNPs. CPU execution time taken from [11], which uses a workstation with two Intel Xeon E5-2620 v2 (Ivy Bridge) 6-core processors running at 2.10 GHz.

2) Clock speed: We calculate peak throughput using maximum clock frequencies reported by OpenCL.

B. Linkage Disequilibrium Performance

Figure 5 shows the performance of the LD kernel computation as the number of SNP strings increases with the length of SNP strings fixed near each device's maximum for one tile. This is consistent with expected behavior of a high performance matrix-matrix multiplication. As the number of SNP strings increases, there is greater data reuse on the accumulation of comparisons for each locus, thus reducing demand on the memory system and allowing the performance to achieve closer to the theoretical peak, shown by the corresponding dotted line (achieved throughput percentage of peak for GTX 980: 90.7%, Titan V: 97.1%, Vega 64: 54.9%).

Compute Capability	Parameter	Xeon E5-2620 v2	GTX 980	Titan V	Vega 64
Microarchitecture		Ivy Bridge	Maxwell	Volta	Vega (GCN5)
Freqency (GHz)		2.1	1.367	1.455	1.663
Thread Group Size	N_T	1	32	32	64
Max Thread Groups	N_{grp}	2	32	32	16
Compute Cores	N_C	2×6	16	80	64
Compute Clusters	N_{cl}	1	4	4	4
Arithmetic Units per Cluster					
32-bit addition [26]	N_{fn}^{+}	4	32	16	16
32-bit logical and	$N_{fn}^{\&}$	4	32	16	16
32-bit population count	$N_{fn}^{popcount}$	1	8	4	16
Instruction Latency	$L_{fn}^{popcount}$	3	6	4	4
Global Memory (GiB)			3.934	11.754	7.984
Max Allocation (GiB)			.983	2.939	6.786
Shared Memory (KiB)	N_{shared}		48	48	64
Shared Memory Banks	N_b		32	32	32
Registers per Core		16 logical	64K	64K	64K
Max Registers per Thread			255	255	256

TABLE I
MAPPING OF THE GPU FEATURES TO THE CORRESPONDING CPU ARCHITECTURE

Figure 6 shows the end-to-end runtime for LD using an increasing number of sequences, each with 10 thousand SNPs. Due to the overhead associated with initializing OpenCL (on the order of hundreds of milliseconds), initialization can dominate end-to-end execution time for small problem sizes. Large enough problems sizes provide enough computation to offset the cost of initialization and data transfer to and from the device. Since kernel functions (those programs that are built to run on the GPU) can be compiled before runtime, the kernel compilation time was excluded from our end-to-end timing. As problem sizes increase, GPUs will provide further improvements in execution time and achieve greater computational efficiency.

C. *Scalability of the algorithm*

We tested the scalability of the algorithm by increasing the number of compute cores that are used to compute the overall algorithm, and the results are shown in Figure 7. In general, the NVIDIA GPUs retain high performance (relative to a single compute core) as we increase the number of cores. However, the performance of the Vega 64 decreases more rapidly as the number of compute cores increases.

The scalability of the Titan V rises above 100% for fewer cores, which may be due to dynamic frequency changes during experimentation or due to the kernel launch cost amortizing across several cores. If launching a kernel is a relatively fixed cost, then performing more computation across more cores will diminish the cost of the kernel launch and result in greater operations per cycle. It is more useful to notice that the Titan V scales almost perfectly, losing virtually no performance as the problem size scales with an increased number of cores. The GTX 980 reaches about 90% efficiency when using all 16 cores.

Our framework performs less efficiently on the Vega 64's. As demonstrated in Figure 7, the performance per core of our framework drops drastically when using more than 8 compute cores on the Vega 64. This scalability issue may be related to memory system behaviors that we have not captured in our analytical model, and is an area of research that we will be pursuing in the near future .

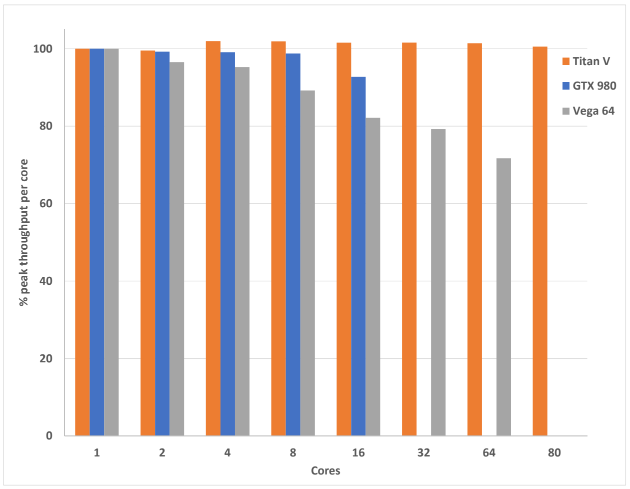

Fig. 7. Using the largest supported LD tile size, this chart shows the performance per core relative to the performance of 1 core. A decline in performance as the number of cores in use shows poor scalability. This illustrates that more work should be done to evaluate the scalability of GPUs, such as the effects of the memory hierarchy.

D. *FastID Performance*

Figure 8 shows end-to-end execution time of FastID for generating the resulting comparisons of 32 queries against a database containing more than 20 million entries [4]. Since performing forensic analysis can have strict time constraints, we opted to show performance for the smallest supported query size that would still make use of all of the available compute resources for our GPUs (in this case, 32 was determined by the number of shared memory banks). As was seen with LD, larger query (matrix) sizes can achieve better compute efficiency so FastID scales well with greater numbers of queries.

[4]The size of the database was chosen due to the size of the FBI NDIS database which as of December 2018 has around 18 million profiles [27].

GPU	GTX 980	Titan V	Vega 64
Linkage disequilibrium			
Core configuration	4×4	80×1	32×2
m_r	4	4	4
n_r	384	1024	1024
k_c	383	383	512
m_c	32	32	32
FastID			
Core configuration	1×16	1×80	1×64
m_r	4	4	4
n_r	768	1024	1024
k_c	383	383	512
m_c	32	32	32

TABLE II
SOFTWARE CONFIGURATION PARAMETERS FOR PERFORMING SNP COMPARISON ALGORITHMS. NOTICE THAT THE TILE COMPUTED BY EACH CORE REMAINS THE SAME WHILE THE CONFIGURATION OF THE CORES ARE DETERMINED BY THE PROBLEM THAT IS BEING COMPUTED.

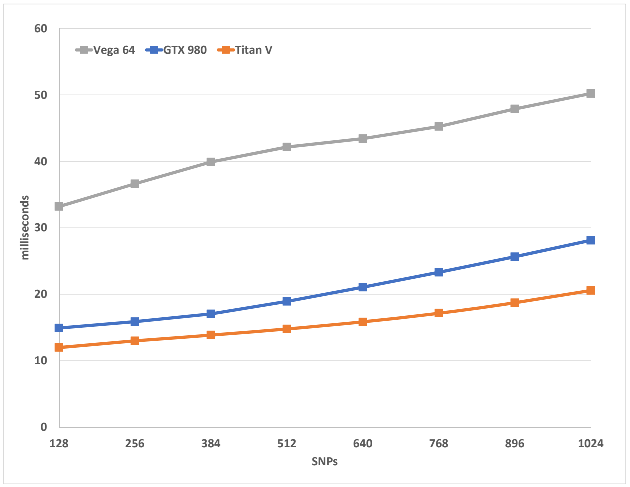

Fig. 8. End-to-end execution time of FastID 32 queries against a database of more than 20 million sequences for SNP counts from 128 to 1024.

E. Adapting for Microarchitectural Features

1) Arithmetic Units: Each of the GPUs used in evaluation implemented a different microarchitecture, which manifests in the model GPU as differing hardware parameters. The available arithmetic units per cluster becomes relevant when considering how to implement a microkernel efficiently for a given device. For FastID mixture analysis, we note that the kernel can be represented using the AND-NOT operation, or pre-negating one input matrix to avoid performing the additional negation upon execution. For the NVIDIA GPUs, since the ratio of arithmetic and logic operations (AND, NOT, ADD) to population count operations (POPC) per output element is less than the ratio of arithmetic and logic functional units to population count functional units, the population count remains the bottleneck. On the other hand, since there as many functional units for logic/arithmetic operations as there are for population count on the Vega 64, it is beneficial to reduce the number of operations that fall on the logic/arithmetic functional units and pre-negate an input matrix. Although we should theoretically pre-negate on the Vega 64,

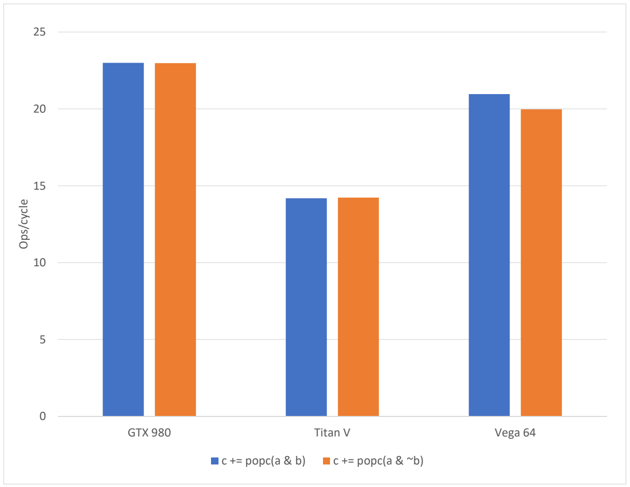

Fig. 9. On 1 core, NVIDIA cards achieve near identical performance when performing an AND or AND-NOT comparison. For Vega, since the negation (NOT) operation falls on the same pipe as ADD and AND, performing the NOT in the computational kernel reduces throughput.

scalability issues reduced throughput and there is little effect when including the negation in the kernel computation. If the scalability issues were to be resolved, we would expect to see lower performance when including the negation in the FastID mixture analysis kernel. Figure 9 illustrates that including the NOT in the computation has no noticeable affect on the NVIDIA cards, but throughput drops for the Vega 64. This was performed using 1 core to lessen the impact of scalability.

2) Global Memory Size: For GPUs that do not support matrices of the size required by the database or resulting output matrix (e.g. the GTX 980), the problem must be broken down into smaller tile sizes. This can be done naturally due to the tiling approach taken in our framework. Even for GPUs that can fit the entire database and output matrix in global memory (e.g. the Titan V), double buffering input and output tiles allows some of the data transfer to be overlapped with computation. The amount of data to be transferred at each step must be evenly balanced with the amount of computation that occurs each step to sufficiently overlap execution and data transfer. Including data transfer characteristics to the analytical model is a subject for future work.

VII. CONCLUSION & FUTURE DIRECTION

In this paper, we presented a portable framework for performing SNP comparisons on GPUs. Specifically, our framework builds on the work by Alachiotis et. al. by showing that the same algorithm on both CPUs and GPUs can be used for performing SNP comparison. In addition, we describe how different software parameters that characterizes the algorithm can be derived based on the available GPU hardware features. Finally, we demonstrate the portability of this approach on various GPU architectures from NVIDIA and AMD.

This approach represents SNP strings as dense bitvectors, but a typical DNA sample is expected to contain mostly major alleles. This suggests that sparse representations of the SNP strings may be beneficial. Extending the framework to sparse matrix-matrix multiplication operations is a goal for future work.

Notice that the results of the NVIDIA GPUs were relatively constant as the number of GPU cores scaled up, but the performance of the AMD GPU did not scale as well with more cores. One possibility is that the current GPU model is lacking in detail about the memory hierarchy of the GPU. A more detailed memory hierarchy model for the GPU may provide insights and better mapping of the software to the hardware features. We plan to investigate how algorithms presented in [28] as matrix multiplication for hierarchical memory architectures could further extract compute resources available on GPUs.

We believe that our framework can be extended to handle even larger problem sizes is to exploit multi-GPU systems such as the DGX-2. On such multi-GPU systems, the increased number of functional units (especially the population count instruction) and the collective memory on the GPUs would facilitate the storage of even larger datasets and computation with a high throughput. However, this comes at the cost of having to communicate between multi-GPUs, which would require an approach that is similar to distributed-memory computing. This is something we intend to explore in the near future.

VIII. Acknowledgement

This work was sponsored partly by the DARPA BRASS program under agreement FA8750-16-2-003, and NSF through award ACI 1550486. The content, views and conclusions presented in this document are those of the authors and do not necessarily reflect the position or the policy of the sponsoring agencies.

References

[1] D. E. Reich, M. Cargill, S. Bolk, J. Ireland, P. C. Sabeti, D. J. Richter, T. Lavery, R. Kouyoumjian, S. F. Farhadian, R. Ward *et al.*, "Linkage disequilibrium in the human genome," *Nature*, vol. 411, no. 6834, p. 199, 2001.

[2] M. T. Alam, D. K. de Souza, S. Vinayak, S. M. Griffing, A. C. Poe, N. O. Duah, A. Ghansah, K. Asamoa, L. Slutsker, M. D. Wilson *et al.*, "Selective sweeps and genetic lineages of plasmodium falciparum drug-resistant alleles in ghana," *Journal of Infectious Diseases*, vol. 203, no. 2, pp. 220–227, 2011.

[3] J. Isaacson, E. Schwoebel, A. Shcherbina, D. Ricke, J. Harper, M. Petrovick, J. Bobrow, T. Boettcher, B. Helfer, C. Zook, and E. Wack, "Robust detection of individual forensic profiles in dna mixtures," *Forensic Science International: Genetics*, vol. 14, pp. 31 – 37, 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1872497314001859

[4] A. Shcherbina, D. O. Ricke, E. Schwoebel, T. Boettcher, C. Zook, J. Bobrow, M. Petrovick, and E. Wack, "Kinlinks: Software toolkit for kinship analysis and pedigree generation from hts datasets," in *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, May 2016, pp. 1–6.

[5] R. Kosoy, R. Nassir, C. Tian, P. White, L. Butler, G. Silva, R. Kittles, M. Alarcon-Riquelme, P. Gregersen, J. Belmont, F. De La Vega, and M. Seldin, "Ancestry informative marker sets for determining continental origin and admixture proportions in common populations in america," *Human Mutation*, vol. 30, no. 1, pp. 69–78, 1 2009.

[6] W. Branicki, F. Liu, K. van Duijn, J. Draus-Barini, E. Popiech, S. Walsh, T. Kupiec, A. Wojas-Pelc, and M. Kayser, "Model-based prediction of human hair color using dna variants," in *Human Genetics*, 2010.

[7] M. Gymrek, A. L. McGuire, D. Golan, E. Halperin, and Y. Erlich, "Identifying personal genomes by surname inference," *Science*, vol. 339, no. 6117, pp. 321–324, 2013. [Online]. Available: http://science.sciencemag.org/content/339/6117/321

[8] AMD, "Vega instruction set architecture," 2017, page 146.

[9] NVIDIA, "CUDA C programming guide," 2018, 5.4.1. Arithmetic Instructions, Table 2.

[10] R. Lewontin and K.-i. Kojima, "The evolutionary dynamics of complex polymorphisms," *Evolution*, vol. 14, no. 4, pp. 458–472, 1960.

[11] N. Alachiotis, T. Popovici, and T. M. Low, "Efficient computation of linkage disequilibria as dense linear algebra operations," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 418–427.

[12] S. Samsi, B. S. Helfer, J. Kepner, A. Reuther, and D. O. Ricke, "A linear algebra approach to fast DNA mixture analysis using GPUs," in *2017 IEEE High Performance Extreme Computing Conference, HPEC 2017, Waltham, MA, USA, September 12-14, 2017*, 2017, pp. 1–6. [Online]. Available: https://doi.org/10.1109/HPEC.2017.8091027

[13] NVIDIA. cublas. [Online]. Available: https://developer.nvidia.com/cublas

[14] Nvidia, "Cutlass: CUDA templates for linear algebra subroutines," Dec 2018. [Online]. Available: https://github.com/NVIDIA/cutlass/releases/tag/v0.1.0

[15] C. C. Chang, C. C. Chow, L. C. Tellier, S. Vattikuti, S. M. Purcell, and J. J. Lee, "Second-generation plink: rising to the challenge of larger and richer datasets," *Gigascience*, vol. 4, no. 1, p. 7, 2015.

[16] D. O. Ricke, "FastID: Extremely fast forensic dna comparisons," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, Sep. 2017, pp. 1–4.

[17] J. A. Gunnels, G. Henry, and R. A. van de Geijn, "A family of high-performance matrix multiplication algorithms," in *Computational Science - ICCS 2001, International Conference, San Francisco, CA, USA, May 28-30, 2001. Proceedings, Part I*, 2001, pp. 51–60. [Online]. Available: https://doi.org/10.1007/3-540-45545-0_15

[18] J. Zhang, F. Franchetti, and T. M. Low, "High performance zero-memory overhead direct convolutions," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, ser. JMLR Workshop and Conference Proceedings, J. G. Dy and A. Krause, Eds., vol. 80. JMLR.org, 2018, pp. 5771–5780. [Online]. Available: http://proceedings.mlr.press/v80/zhang18d.html

[19] F. G. Van Zee and R. A. van de Geijn, "BLIS: A framework for rapidly instantiating BLAS functionality," *ACM Transactions on Mathematical Software*, vol. 41, no. 3, pp. 14:1–14:33, Jun. 2015. [Online]. Available: http://doi.acm.org/10.1145/2764454

[20] F. G. Van Zee and T. Smith, "Implementing high-performance complex matrix multiplication via the 3m and 4m methods," *ACM Transactions on Mathematical Software*, vol. 44, no. 1, pp. 7:1–7:36, Jul. 2017. [Online]. Available: http://doi.acm.org/10.1145/3086466

[21] T. M. Low, F. D. Igual, T. M. Smith, and E. S. Quintana-Ortí, "Analytical modeling is enough for high-performance BLIS," *ACM Transactions on Mathematical Software*, vol. 43, no. 2, pp. 12:1–12:18, Aug. 2016. [Online]. Available: http://doi.acm.org/10.1145/2925987

[22] D. Glasco, B. Khailany, M. Garland, S. W. Keckler, and W. J. Dally, "GPUs and the future of parallel computing," *IEEE Micro*, vol. 31, pp. 7–17, 10 2011. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MM.2011.89

[23] T. M. Smith, R. A. van de Geijn, M. Smelyanskiy, J. R. Hammond, and F. G. Van Zee, "Anatomy of high-performance many-threaded matrix multiplication," in *28th IEEE International Parallel & Distributed Processing Symposium (IPDPS 2014)*, 2014.

[24] J. E. Stone, D. Gohara, and G. Shi, "Opencl: A parallel programming standard for heterogeneous computing systems," *Computing in Science Engineering*, vol. 12, no. 3, pp. 66–73, May 2010.

[25] V. Volkov, "Better performance at lower occupancy." Presented at UC Berkeley, 2010.

[26] A. Fog *et al.*, "Instruction tables: Lists of instruction latencies, throughputs and micro-operation breakdowns for intel, amd and via cpus," *Copenhagen University College of Engineering*, p. 375, 2018.

[27] "Codis - ndis statistics," Jan 2019. [Online]. Available: https://www.fbi.gov/services/laboratory/biometric-analysis/codis/ndis-statistics

[28] T. M. Smith *et al.*, "Theory and practice of classical matrix-matrix multiplication for hierarchical memory architectures," Ph.D. dissertation, 2018.