Accelerating Minor Allele Frequency Computation with Graphics Processors

Mian Lu, Jiuxin Zhao, Qiong Luo
Hong Kong University of Science and Technology
{lumian, zhaojx, luo}@cse.ust.hk

Bingqiang Wang
BGI-Shenzhen, China
wangbingqiang@genomics.cn

ABSTRACT

The computation of minor allele frequency (MAF) is at the core of a Genome-Wide Association Study (GWAS). Due to the high computation intensity and high precision requirement, so far the scale of MAF computation analysis is up to hundreds of individuals. To enable the computation for thousands of individuals, we have developed GAMA, a high performance MAF computation program with GPU acceleration. Specifically, we design a parallel reduction algorithm that matches the GPU's data-parallel architecture. To implement the new algorithm efficiently on the GPU, we utilize the fast, on-chip local memory shared within each GPU multiprocessor effectively. To avoid user-level thread synchronization, we exploit the GPU thread-warp based scheduling. Furthermore, we address the floating point underflow issue through a logarithm transformation. As a result, GAMA enables MAF computation for up to a thousand individuals for the first time. On a server equipped with an NVIDIA Tesla C2070 GPU and two Intel Xeon E5520 2.27 GHz CPUs, GAMA outperforms a state-of-the-art single-threaded MAF computation tool and our optimized parallel implementation (16-threaded) on the CPU by around 47 and 3.5 times, respectively.

1. INTRODUCTION

With the rapid progress of DNA sequencing techniques, large-scale genome-wide association studies (GWAS) have become practical. These studies investigate DNA variations among a group of individuals to identify causes of complex traits, such as diseases. In these studies, the computation of minor allele frequency (MAF) is a fundamental data analysis task [2, 4, 8].

Due to the high intensity, the MAF computation usually takes an excessively long running time. For example, the state-of-the-art MAF computation tool *realSFS* [3] (single-threaded), is estimated to take around nine months to process the whole human genome for a data set of 1,024 individuals on a server with two Intel Xeon E5520 2.27 GHz CPUs.

BigMine'12, August 12, 2012 Beijing, China

Another complication is that, floating point underflow may occur when the number of individuals is large, e.g., 1,024.

In this work, we develop an efficient MAF computation tool named GAMA (GPU-Accelerated Minor Allele frequency computation), adopting the graphics processing unit (GPU) as a hardware accelerator. Specifically, we make the following contributions.

1. We propose a new parallel reduction algorithm for the site frequency spectrum (SFS) construction, which is the most expensive component in MAF computation. Compared with the original construction algorithm, our new algorithm can better utilize the GPU hardware resource.

2. We implement the new SFS construction algorithm on the GPU efficiently through two techniques. First, we implement it in a block nested-loop style utilizing the small on-chip GPU *shared memory*. Second, we avoid the user-level thread synchronization overhead by exploiting the GPU's thread-warp based scheduling.

3. We transform the computation to logarithm space to solve the floating point underflow issue.

On a server equipped with an NVIDIA Tesla C2070 and two Intel Xeon E5520 2.27GHz CPUs, GAMA outperforms *realSFS* [3] (single-threaded) by 47 times on a human genome data set of 1,024 individuals. Compared with our optimized parallel CPU implementation (16 threads), GAMA has a speedup of 3.5 times.

The remainder of the paper is organized as follows. In Section 2, we introduce the background. We present the GPU-based implementation and numerical optimization in Section 3 and 4, respectively. We evaluate GAMA in Section 5 and conclude the paper in Section 6.

2. BACKGROUND

2.1 Minor Allele Frequency and Site Frequency Spectrum

A chromosome contains a large number of *sites* (positions), and each site stores one of four base types (known as A, T, C, and G). For diploid creatures, e.g., human, there are two almost identical copies for each chromosome (known as *homologous chromosome*). However, a single nucleotide mutation (known as single-nucleotide polymorphism, or SNP) may happen on one of the paired homologous chromosomes. In most cases, we only consider two

alleles (two possible types of bases) for a site. For a given population with N individuals, there are $2N$ bases in total for each site. For a given site, the less common allele is called a *minor allele*, otherwise a *major allele*. The frequency of minor allele occurring in a given population is called *minor allele frequency* (MAF). MAF is important for GWAS, which has been studied in various genomics research projects [2, 4, 6, 8].

Given a population with N individuals, for each site, we define a vector h. $h[i]$ indicates the likelihood when the MAF is equal to i. Suppose for a given site, the major and minor allele are denoted as M and m, respectively. For one individual, there are three possible combinations of bases in homologous chromosomes, which are $\{MM\}$, $\{Mm\}$, and $\{mm\}$. Then for two individuals, there are five possible combinations of alleles, which are $\{MMMM\}$, $\{mMMM\}$, $\{mmMM\}$, $\{mmmM\}$, and $\{mmmm\}$. As a result, for N individuals, the length of the likelihood vector h is $(2N+1)$. The set of all h vectors for all sites is called the *site frequency spectrum (SFS)* for the population. Based on SFS, the useful information of MAF can be produced after postprocessing.

Since the MAF computation relies on a probability model, the result is more accurate when there are more individuals. The latest genomics studies report analysis results based on data sets of up to hundreds of individuals [2, 4, 8]. The goal of this work is to enable the processing of thousands of individuals efficiently. Such a data scale can produce much more accurate results, and our work will be a significant enabler for genomics research.

2.2 MAF Computation

We adopt the MAF computation model that is implemented in realSFS [3, 7], as well as adopted in genomics projects [2, 8]. Overall, the same algorithm is applied to every site independently. There are four steps to process a site, which are *data parsing*, *SFS construction*, *SFS normalization*, and *result post-processing*.

In this paper, we focus on the GPU acceleration for SFS construction (Section 3). Suppose the number of individuals is N. The computation complexity of SFS construction is $O(N^2)$, and those of the other three components are $O(N)$. In our experiments, SFS construction dominates the performance of MAF computation. To the best of our knowledge, there is no previous work accelerating the SFS construction algorithm using GPUs. We also address the floating point underflow issue that occurs in the component SFS normalization (Section 4).

3. SFS CONSTRUCTION ON THE GPU

3.1 Per-Thread-Per-Site SFS Construction

Algorithm 1 shows the SFS construction algorithm for one site. Given the likelihoods of three MAF for each individual ($P_{MM}[i]$, $P_{Mm}[i]$ and $P_{mm}[i]$ represent the likelihoods for the i-th individual), SFS (the array h) is constructed through an iterative-update approach.

The straightforward implementation of the SFS construction on the GPU is to make one thread handle one site. The major performance issue is the large number of memory accesses on h, as the small cached *shared memory* on the GPU cannot be utilized effectively. Suppose there are N individuals. The size of h for one site takes $\big(8 \times (2 \times N + 1)\big)$ bytes (h is double precision). The size of shared memory per GPU multiprocessor is up to 48 KB. Therefore, if we store h using the shared memory, when $N = 1,000$, the number of active threads on each multiprocessor is only three, out of 1,536 concurrent threads that are supported per GPU multiprocessor. Additionally, this algorithm is not suitable for buffering a sub-array of h in the shared memory at a time, as may introduce quite a few data transfers between the GPU global and shared memory.

Algorithm 1: The iterative-update SFS construction for one site: ***construct_sfs***(h, N, P_{MM}, P_{Mm}, P_{mm})

Input:
N: the number of individuals
P_{MM}, P_{Mm}, P_{mm}: three arrays, each of which has N double-precision numbers.
Output:
h: the array with $(2N + 1)$ doubles.

```
1  h ← 0
2  for i ← 1 to N do
3      p_MM = P_MM[i − 1]; p_Mm = P_Mm[i − 1];
       p_mm = P_mm[i − 1];
4      if i = 1 then
5          h[0] = p_MM; h[1] = p_Mm; h[2] = p_mm;
6      else
7          h[2i] = p_MM · h[2i − 2];
8          h[2i − 1] = p_MM · h[2i − 3] + p_Mm · h[2i − 2];
9          for j ← 2i − 2 to 2 do
10             h[j] = p_MM·h[j−2]+p_Mm·h[j−1]+p_mm·h[j];
11         h[1] = p_Mm · h[0] + p_mm · h[1];
12         h[0] = p_mm · h[0];
```

$$\begin{pmatrix}
p^i_{mm} & 0 & 0 & 0 & 0 & \dots & 0 \\
p^i_{Mm} & p^i_{mm} & 0 & 0 & 0 & \dots & 0 \\
p^i_{MM} & p^i_{Mm} & p^i_{mm} & 0 & 0 & \dots & 0 \\
0 & p^i_{MM} & p^i_{Mm} & p^i_{mm} & 0 & \dots & 0 \\
0 & 0 & p^i_{MM} & p^i_{Mm} & p^i_{mm} & \dots & 0 \\
\vdots & \vdots & \ddots & \ddots & \ddots & \ddots & 0 \\
0 & 0 & \dots & p^i_{MM} & p^i_{Mm} & p^i_{mm} & 0 \\
0 & 0 & 0 & \dots & p^i_{MM} & p^i_{Mm} & p^i_{mm}
\end{pmatrix}$$

Figure 1: The super-band matrix F_i for the i-th individual. The matrix size is $(2N + 1) \times (2N + 1)$.

In view of the drawbacks of the straightforward GPU implementation, we design a new SFS construction algorithm that matches the GPU architecture better. The basic idea is that we further parallelize the SFS construction algorithm for a single site. This way, we can make multiple threads handle a site with effective shared memory usage.

3.2 Parallel Reduction Algorithm for SFS Construction

In this section, we first show that the iterative-update on h (line 4 to 12 in Algorithm 1) is parallelizable through a representation of matrix multiplication. To remove the redundant computation in matrix multiplication, we further improve it as a reduction algorithm. Finally, we introduce the parallel implementation of the reduction.

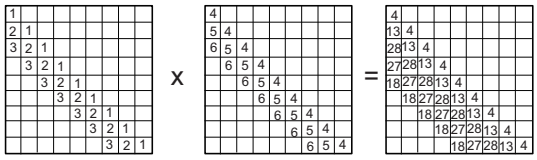


Figure 2: Multiplication of two super-band matrices when $N = 4$. Empty cells indicate zero values.

We define a *super-band matrix*. N is the number of total individuals throughout this paper.

Definition 1: Super-band matrix. *Given an $(n \times n)$ lower triangular band matrix and an integer C, where $n \geq C \geq 1$, the matrix is a super-band matrix iff: (1). For each integer k, where $C \geq k > 0$, cells at positions $(i, i-k)$, where $N > i \geq k$, have the same value. (2). All other cells are zeros.*

The parameter C is called *cardinality*. In a super-band matrix with parameter C, the number of unique non-zero values is up to C, and all these unique non-zero values appear in each column (except the last $(C-1)$ columns).

For a given site, for the i-th individual (i starts from 1), suppose the likelihoods of three MAF for this individual are denoted as p^i_{mm}, p^i_{Mm}, and p^i_{MM}, we can define a super-band matrix F_i with the size of $(2N+1) \times (2N+1)$ and the cardinality of 3, which is illustrated in Figure 1.

For a given site, suppose the computation for the first $(k-1)$ individuals has been done. The iterative-update on h for the k-th individual (line 7 to 12 of Algorithm 1) can be equivalently represented as a matrix-vector multiplication (the vector size is $(2N+1)$):

$$F_k \times \begin{pmatrix} h_{k-1}[0] \\ h_{k-1}[1] \\ \vdots \\ h_{k-1}[2k-2] \\ 0 \\ 0 \\ \vdots \\ 0 \end{pmatrix}_{(2N+1)} = \begin{pmatrix} h_k[0] \\ h_k[1] \\ \vdots \\ h_k[2k-2] \\ h_k[2k-1] \\ h_k[2k] \\ \vdots \\ 0 \end{pmatrix}_{(2N+1)}$$

Note that, in Algorithm 1, the input and output arrays are the same one. However, we represent them separately as h_{k-1} and h_k in the matrix-vector multiplication. Now for N individuals, we have:

$$h_N = F_N \cdot h_{N-1} = F_N \cdot F_{N-1} \cdot h_{N-2} = F_N \cdot F_{N-1} \cdot F_{N-2} ... \cdot h_1$$

Note that as the matrix multiplication is associative, we have removed the original association among matrices (the original association is from right to left). Such a representation confirms that the original iterative-update SFS construction algorithm for a single site is parallelizable due to the matrix multiplication representation.

The matrix multiplication based SFS construction contains redundant computation. Figure 2 illustrates the process of multiplication on two super-band matrices when $N = 4$. This shows that the number of unique values in the result matrix is only five. To simplify the presentation, in the remaining of this section, the *matrix multiplication* specifically refers to the multiplication on two same size $(n \times n)$ super-band matrices with the same cardinality C $(n \geq 2C-1)$, unless otherwise specified.

$$\begin{pmatrix} a_{[0]}b_{[0]} \\ a_{[0]}b_{[1]} + a_{[1]}b_{[0]} \\ a_{[0]}b_{[2]} + a_{[1]}b_{[1]} + a_{[2]}b_{[0]} \\ \vdots \\ \sum_{i=0}^{C-2} a_{[i]}b_{[C-2-i]} \\ \sum_{i=0}^{C-1} a_{[i]}b_{[C-1-i]} \\ \sum_{i=1}^{C-1} a_{[i]}b_{[C-i]} \\ \sum_{i=2}^{C-1} a_{[i]}b_{[C+1-i]} \\ \vdots \\ a_{[C-3]}b_{[C-1]} + a_{[C-2]}b_{[C-2]} + a_{[C-1]}b_{[C-3]} \\ a_{[C-2]}b_{[C-1]} + a_{[C-1]}b_{[C-2]} \\ a_{[C-1]}b_{[C-1]} \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Figure 3: The first column of the result matrix for the multiplication on two super-band matrices A and B. The cardinality of input matrices is C. a and b represents the first columns of A and B, respectively.

We have two important observations for the matrix multiplication: First, the result matrix is a super-band matrix. Second, the cardinality of the result matrix is $(2C-1)$. The observations can be proved through directly generating the result matrix following the definition of matrix multiplication. Therefore, the proof is ignored here.

For a super-band matrix, the first column stores all unique values. Therefore, we focus on representing the first column of result matrix using the first columns from the two input matrices, in order to eliminate the computation and storage redundancy. Suppose a and b are the first columns of two input super-band matrices A and B respectively, then the first column of the result matrix can be represented in Figure 3. Note that, the vector of Figure 3 is obtained through replacing the element not in the first column of matrix A or B using the corresponding element with the same value in the first column of A or B. For example, the second element of Figure 3 in fact is $(A_{[1][0]}B_{[0][0]} + A_{[1][1]}B_{[1][0]})$. Since $A_{[1][1]} = A_{[0][0]}$, the result is represented as $(a_{[1]}b_{[0]} + a_{[0]}b_{[1]})$ using all elements from the first columns of A and B.

To abstract the computation of Figure 3, we first define an operator $\otimes$ in Algorithm 2. The defined operator is associative as well. Now we can introduce the reduction algorithm for a single site. For every individual, there is an initial vector (denoted as v_i for the i-th individual) holding three elements to store its p_{mm}, p_{Mm} and p_{MM}. Then we perform a vector-based reduction on all vectors:

$$h = v_1 \otimes v_2 \otimes ... \otimes v_N \quad (1)$$

, where h is the final SFS with $(2N+1)$ elements for a given site for N individuals. We can verify that the result of this formula is equivalent to the vector shown in Figure 3.

On the GPU, we parallelize this reduction algorithm. The parallel reduction consists of a few levels, and each level has several merges using the operator $\otimes$ on two successive vectors. The merges at the same level can be performed in parallel. Recall that the purpose of this algorithm is to par-

Algorithm 2: The definition of operator $\otimes$.

Input:
a: an array with the size of n.
b: an array with the size of m.
Output:
c: an array with the size of $(m+n-1)$

```
1 c ← 0;
2 for i ← 0 to n − 1 do
3 |   for j ← 0 to m − 1 do
4 |   |   c[i + j] += a[i] × b[j];
```

allelize the construction algorithm for a single site. Additionally, the parallelization among different sites is straightforward based on this algorithm.

3.3 Analysis and Optimizations of the Parallel Reduction

We have converted the original iterative-update SFS construction to a parallelizable reduction algorithm. Before introducing its GPU-based implementation, we analyze its performance and propose further optimizations.

Computation complexity. The complexity analysis is for the sequential implementations of both algorithms. We assume the number of individuals is a power of two. For an $\otimes$ operator, the computation complexity is $(n \times m)$ as it is a nested-loop computation. Therefore, for the reduction, the complexity $C_{reduction}$ is calculated as:

$$C_{reduction} = \sum_{i=1}^{log_2^N} [(2^i + 1)^2 \frac{N}{2^i}]$$

After the simplification, the result is:

$$C_{reduction} = 2N^2 + 2Nlog_2^N - N - 1$$

For the iterative-update (Algorithm 1), the complexity is calculated as the number of multiplications (such a measurement is consistent with the one of nested-loop computation). Therefore, the complexity $C_{iterative}$ is:

$$C_{iterative} = \sum_{i=1}^{N-1} (3 \times (2i + 1))$$

After the simplification, the result is:

$$C_{iterative} = 3N^2 - 3$$

Two algorithms have the same complexity upper bound $O(N^2)$. However, the concrete numbers are different. The workload of the reduction is around 67% of the iterative-update algorithm theoretically.

The nested-loop optimization. The nested-loop computation (Algorithm 2) has an additional advantage. That is, as long as an element in the outer input array ($a[i]$ in line 4 of Algorithm 2) is zero, the entire scan on the inner input array (line 3 to 4 of Algorithm 2) for that element is unnecessary. This is essentially similar to the computation on sparse data structures, which skips the computation on zero numbers. Though the sparse data representation is not adopted, the performance improvement of our technique should be similar to that with the sparse data format employed. Such an optimization is difficult to be employed by the iterative-update construction. In practice, this improvement can save the computation workload significantly due to the characteristic of real-world data sets. Therefore, before each $\otimes$ operation, we scan the two input arrays once, and make the one with more zeros as the outer input array.

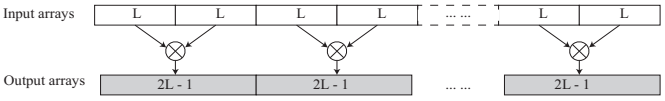


Figure 4: The primitive *binaryMerge*. The size of each input and output array are L and $(2L-1)$, respectively.

3.4 SFS Construction with the Reduction Algorithm on the GPU

We implement the optimized parallel reduction algorithm on the GPU. We first define a primitive *binaryMerge* to process multiple $\otimes$ operations in parallel. The primitive is illustrated in Figure 4, in which all input arrays have the same length L. Then each two successive input arrays are merged into an output array with the size of $(2L-1)$ using the operator $\otimes$. With such a primitive, it is straightforward to implement the SFS construction on the GPU. Note that, to fully utilize the hardware resource, multiple sites are processed in parallel, which also can be done through the primitive.

To implement the primitive *binaryMerge* on the GPU efficiently, overall, multiple threads are used to handle one $\otimes$ operator, and multiple $\otimes$ operators are processed concurrently as well. We focus on how to use multiple threads to handle an $\otimes$ operator. It is easy to parallel multiple $\otimes$ operators as they have the same operations.

The basic strategy to parallelize a $\otimes$ operator is that, each thread first holds an element from the outer array a, and then scans the entire inner array b sequentially to perform the computation. There are two major techniques adopted, which are shared memory utilization and the automatic thread-warp based synchronization.

Shared memory utilization. To utilize the small size shared memory effectively, we implement Algorithm 2 in a block nested-loop scheme. There are two data buffers in the shared memory, which are used to hold the inner input array b (denoted as buf_b) and output array c (denoted as buf_c) block by block. The inner array b is loaded into the shared memory block by block. For each block of b, there are several computation stages, and each stage will access a fixed number of elements in a to do the nested-loop computation. Within a stage, the output results are stored in buf_c. Suppose buf_b and buf_c can hold n_b and n_c elements respectively, then the number of elements accessed for a in each stage is $(n_c - n_b + 1)$. Note that for the nested-loop computation on a block of b, each element of a stored in the global memory is just accessed once by a thread, as a is the outer input array. The purpose of accessing the input array a through multiple stages is to utilize the small shared memory for buf_c to hold output elements.

Figure 5 illustrates an example of the shared memory usage. In the first step, the elements $b[0]$, $b[1]$, $b[2]$ are buffered in the shared memory. Then for the first stage, elements $a[0]$, $a[1]$, $a[2]$ are accessed, and each of which accesses $b[0]$, $b[1]$, $b[2]$ that are stored in the shared memory to perform

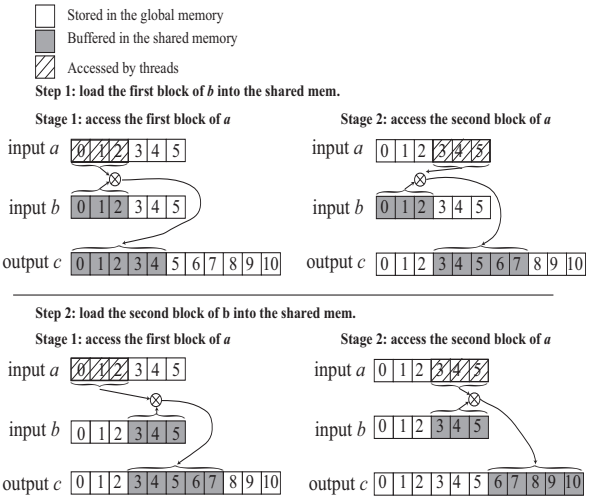


Figure 5: The shared memory usage for one $\otimes$ operation implemented on the GPU. Suppose the lengths of two input arrays are both 6, and the block size for the input array b and output array c are 3 and 5, respectively. Then the number of elements in a accessed for each stage is 3.

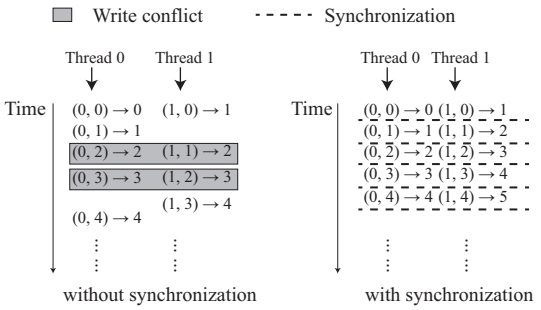


Figure 6: Write conflict without the thread synchronization and the conflict-free output with a synchronization before each write. The pair $(\mathbf{i,j}) \rightarrow i+j$ indicates the position i in the outer input array, and the position j in the inner input array. Then the output position for the pair of (i, j) is $(i + j)$.

the computation. The output position on c of this stage is from 0 to 4. Therefore, the elements $c[0]$, $c[1]$, ..., $c[4]$ are buffered in the shared memory. Next, in the second stage of step one, $a[3]$, $a[4]$, and $a[5]$ are accessed, and $c[3]$, $c[4]$, ..., $c[7]$ are buffered in the shared memory. The second step will perform the similar operations with the second block of b buffered in the shared memory. Note that, since there is overlap of the output elements buffered between different stages or steps, the data transfer between the global and shared memory for the output array is unnecessary for the overlapped elements.

Warp based implementation. Recall that we make each thread fetch one element of the outer array a, and then access all elements sequentially in the inner array b to perform the nested-loop computation. For the computation on i-th element in a and j-th element in b, the output position on c is $(i+j)$. There may be write conflict without synchronization. Figure 6 illustrates an example for two threads. Without the synchronization, two threads may write to the same position. With the synchronization, due to the sequential scan on the inner input array, it ensure the write positions among different threads to be different.

However, such a large number of synchronization will introduce considerable overhead. As threads in the same warp are automatically synchronized, we use one warp to handle one $\otimes$ operator. This is equivalent that there is a synchronization before each write. This way, the write conflict is solved without overhead.

Performance impact of the nested loop optimization on the GPU. As introduced in Section 3.3, an important technique for the reduction algorithm is the nested-loop optimization, which can greatly reduce the workload for real-world data sets. However, for the GPU-based implementation, such an optimization may introduce branch divergence. As long as a branch divergence occurs, different execution paths in a warp are serialized, which increases the total number of instructions. Therefore, the nested-loop optimization is a performance tradeoff on the GPU, which can reduce the computation workload as well as introduce overhead. However, through our evaluations, for the overall performance, the improvement from the nested loop optimization is more significant. Therefore, in our GPU-based implementation, we still adopt the nested loop optimization.

4. NUMERICAL OPTIMIZATIONS

4.1 Floating Point Underflow

The floating point underflow occurs in the SFS normalization step. Algorithm 3 shows the major steps of the SFS normalization for a site that are related to the floating point underflow problem, where t, g, and p are scalars storing intermediate results. Specifically, at line 3 of Algorithm 3, when the number of individuals is larger than around 400, t will be less than around -800. As a result, the result of $exp(t)$ will be smaller than 10^{-308}, which will be treated as 0 in computers (double precision) due to the finite representation of the exponent. Consequently, all elements in the array a ($a[i] = h[i] \times exp(t)$) may become 0 when t is sufficiently small. In such a case, the sum of all elements in the array a also becomes 0. If g is also 0 (it is quite possible), it makes b (line 5 of Algorithm 3) become 0. In such a case, the program will crash due to the *division by zero* error occurred in line 7 of Algorithm 3.

Algorithm 3: SFS normalization for a site.

```
1 ...
2 for i ← 0 to 2N do
3     a[i] = h[i] × exp(t)
4 h_sum = sum(a, 2N + 1);
5 b = p × h_sum + (1 − p) × g;
6 for i ← 0 to 2N do
7     h[i] = p×a[i] / b
8 ...
```

The key to avoid the floating point underflow is to represent small numbers correctly. A straightforward method is to adopt extended precision libraries (either on the CPU [1] or GPU [5]) to represent small numbers. We propose

another approach based on logarithm transformation to address this issue, which is more efficient than the extended precision solution.

4.2 SFS Normalization with Logarithm Transformation

The based idea of this solution is that we transform the computation into logarithm space. This way, very small numbers can be represented correctly in logarithm space.

Algorithm 4: The new normalization algorithm with logarithm transformation for a site.

```
1  ...
2  for i ← 0 to 2N do
3      h[i] = log(h[i])
4  for i ← 0 to 2N do
5      a[i] = h[i] + t
6  h_max = max(a);
7  h_sum = 0.0;
8  for i ← 0 to 2N do
9      h_sum += exp(a[i] − h_max);
10 h_sum = log(h_sum) + h_max;
11 b = addProtect(log(p) + h_sum, log(1 − p) + log(g));
12 for i ← 0 to 2N do
13     h[i] = exp(a[i] + log(p) − b)
14 ...
```

Algorithm 5: addProtect(a, b)

```
1  m = max(a, b);
2  s = exp(a − m) + exp(b − m);
3  return log(s) + m;
```

Algorithm 4 and 5 shows the new normalization algorithm with logarithm transformation. Note that the function *addProtect* (Algorithm 5) essentially is used to perform the following computation:

$$addProtect(log(x)+log(y), log(z)+log(w)) = log(xy+zw)$$

Moreover, the purpose of the *max* operation in line 6 of Algorithm 4 and line 1 of Algorithm 5 is to further avoid the floating point underflow.

5. EVALUATION

5.1 Experimental Setup

We first investigate the performance for the SFS construction and normalization, including the performance impact of various techniques. We then show an end-to-end performance comparison.

Hardware setup. We conduct experiments on a server equipped with an NVIDIA Tesla C2070 GPU and two Intel Xeon E5520 2.27 GHz quad-core CPUs (8 cores, 16 threads in total). The GPU has 448 cores and 6 GB global memory. The shared memory on each GPU multiprocessor is 48 KB. The server has 32 GB main memory.

Data sets. We use a real-world human genome data set provided by BGI-Shenzhen. It contains 2,674,654 sites, each of which has 1,024 individuals.

Implementation details. All CPU programs are developed using C++. The parallel CPU implementation is developed using OpenMP. The GPU implementation is developed using NVIDIA CUDA C 4.0. There are three implementations used for evaluations as summarized in Table 1. The realSFS program adopted in this study has been improved by us compared with the publicly released one [3]. The optimizations include numerical ones for the floating point underflow issue as well as those on memory layout and computation efficiency. As a result, the optimized realSFS is already around three times faster than the original one. The time of disk I/O is excluded in our evaluations, which takes around 10% of the overall time in realSFS.

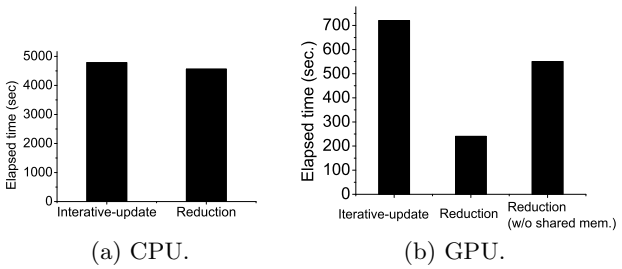
Table 1: Summary of different implementations.

	realSFS (improved)	**GAMA-CPU**	**GAMA**
CPU/GPU	CPU	CPU	GPU
Multi-threaded?	No	Yes	N/A
SFS construction	Iter.-update	Reduction	Reduction
Numerical opt.	Logarithm	Logarithm	Logarithm

5.2 Performance of the SFS Construction

We first examine the performance of the reduction-based SFS construction algorithm and its nested-loop optimization. The CPU implementations in this group of experiments are all single-threaded.

Performance impact of algorithms. As the iterative-update algorithm cannot take advantage from the nested-loop optimization, we first remove this technique from our implementations to investigate the performance impact from the reduction algorithm only. Figure 7(a) shows that on the CPU, the reduction based algorithm is slightly faster than the iterative-update algorithm. We consider this difference is mainly from the computation complexity difference. Instead, Figure 7(b) shows that, on the GPU, the implementation with the reduction construction algorithm outperforms the one with the iterative-update by around three times. The more significant improvement on the GPU mainly benefits from the effective GPU shared memory usage. To confirm this, Figure 7(b) further shows that if we remove the shared memory optimization, reduction is only 24% faster than iterative-update.

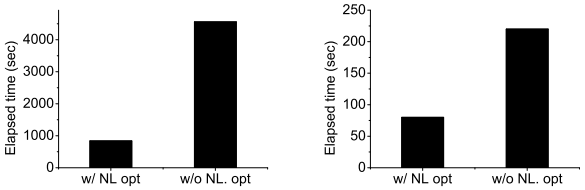


(a) CPU. (b) GPU.

Figure 7: Performance comparison between the iterative-update and reduction based (without the nested-loop optimization) SFS construction algorithms on the CPU and GPU.

Performance impact of the nested-loop optimiza-

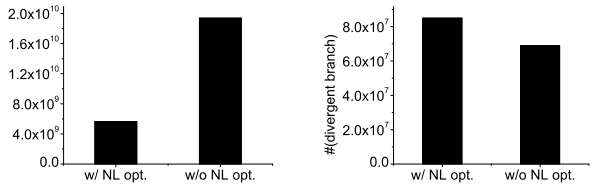
tion. Now we investigate the performance impact from the nested loop optimization for the reduction based SFS construction algorithm. Figure 8(a) shows that on the CPU, the implementation with the nested-loop optimization is around five times faster. However, the improvement is only around two times on the GPU as shown in Figure 8(b). Through our further study, the improvement on the CPU is consistent with the saved workload from this optimization. However, the less significant speedup on the GPU may be due to the overhead introduced by branch divergence, as discussed in the end of Section 3.4.



(a) CPU. (b) GPU.

Figure 8: Performance comparison for the reduction based SFS construction with and without the nested-loop optimization on the CPU and GPU.

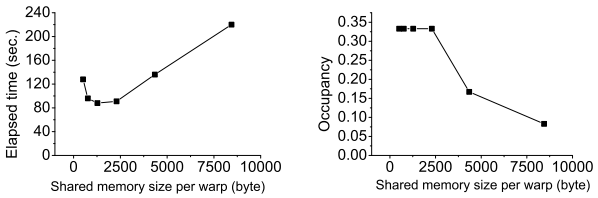
Hardware counters for the nested-loop optimization. To understand the less significant speedup on the GPU from the nested-loop optimization, we investigate the GPU hardware counters. Figure 9(a) shows that, with the nested loop optimization, the computation workload is reduced significantly in number of instructions. However, Figure 9(b) shows that this technique introduces additional divergent branches. This confirms that the performance improvement of the nested-loop optimization on the GPU is offset by branch divergence.



(a) The number of instructions. (b) The number of divergent branches (a warp).

Figure 9: The number of instructions and divergent branches for the reduction-based SFS construction on the GPU with and without the nested loop optimization.

The shared memory utilization. We demonstrate the performance with the shared memory size used per warp varied. Figure 10(a) shows that the best performance is achieved when the shared memory size per warp is around 2 KB. To understand this phenomenon, we further examine the GPU multiprocessor *occupancy* in Figure 10(b). Note that, when the shared memory per warp increases, the multiprocessor occupancy may decrease as there is limited shared memory resource, which may hurt the overall performance. Associating the two figures, when the shared memory size varies from 504 to 2296 bytes, the occupancy remains unchanged. Therefore, the performance improvement benefits from the larger shared memory buffer. When the shared memory used becomes larger than 2296 bytes, the occupancy starts to decrease. As a result, although the used shared memory per warp further increases, the overall performance is slowed down significantly.



(a) Elapsed time (b) Occupancy.

Figure 10: The performance with the size of shared memory used per warp varied for the reduction-based SFS construction on the GPU. (a.) Elapsed time. (b.) GPU multiprocessor occupancy.

The thread-warp based automatic synchronization. Figure 11 shows that with the warp-based synchronization, the performance is around 2.5 timers faster than the user-level explicit thread synchronization. Note that, without the warp-based synchronization, it cannot benefit from the nested-loop optimization. This is because the explicit synchronization should be in the inner loop. As soon as a synchronization occurs, no threads will be able to skip the inner loop. This explains why the performance with explicit thread synchronization is even worse than the implementation without the nested loop optimization.

Performance comparison among different SFS construction implementations. We summarize the performance comparison of the SFS construction using the iterative-update and reduction-based algorithms on the GPU and CPU in Figure 12. We have three observations: (1) The GPU-based implementations employing the iterative-update and reduction-based outperform their CPU counterparts by around 7 and 10 times, respectively. (2) The reduction implementations outperform the iterative-update algorithms on the CPU and GPU by around 5.7 and 9 times, respectively. (3) Compared with the original iterative-update construction on the CPU (shown in Figure 12(a)), our optimized GPU-accelerated reduction-based SFS construction (shown in Figure 12(b)) has a speedup of around 60 times.

5.3 Performance of the SFS Normalization

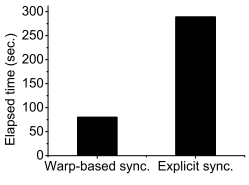
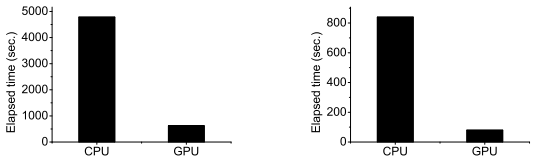


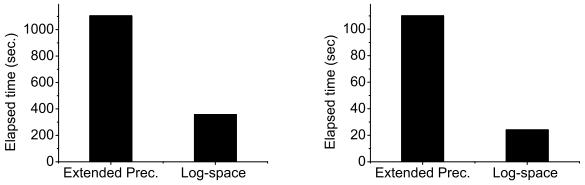
Figure 11: Performance comparison between the GPU implementations with and without the warp-based automatic thread synchronization.



(a) Iterative-update. (b) Reduction-based.

Figure 12: Performance comparison among the iterative-update and reduction-based SFS construction algorithms on the CPU and GPU.

We compare the performance of SFS normalization using the logarithm transformation with that using the CPU- [1] or GPU-based [5] extended precision libraries.



(a) CPU (b) GPU.

Figure 13: Performance comparison between the extended precision and logarithm transformation on the CPU and GPU for the SFS normalization.

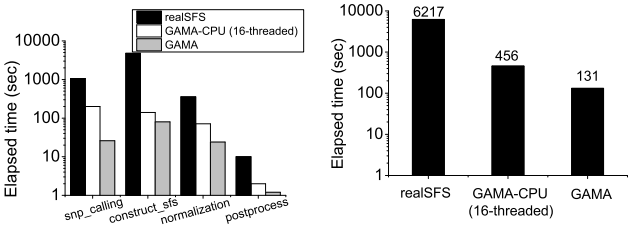
Figure 13 shows that the normalization based on the logarithm transformation is around 3 and 4.6 times faster than that using extended precision on the CPU and GPU, respectively. This indicates that the overhead introduced by the extended precision is much higher than that introduced by the logarithm transformation. Moreover, on the GPU, the computation with extended precision usually has a smaller speedup than that with native precision due to algorithmic complexity [5]. This also explains that the speedup of logarithm transformation over extended precision on the GPU is more significant than that on the CPU.

5.4 Overall Performance Comparison

Finally, we compare the end-to-end performance of MAF computation including all four components. Figure 14(a) shows that for different components, GAMA is around 15-60X faster than realSFS. Compared with the optimized 16-thread GAMA-CPU, the GPU-based implementation is around 2-7 times faster. For the overall computation time, Figure 14(b) shows that the speedup of GAMA over realSFS is around 47 times. Additionally, GAMA outperforms GAMA-CPU using 16 threads (around 7 times faster than the single-threaded GMAM-CPU) by around 3.5 times.

6. CONCLUSION

The state of the art in genomics research reports MAF results based on data sets of up to hundreds of individuals due to the high computation intensity and floating point underflow issue. We develop a fast GPU-accelerated MAF computation tool GAMA, which successfully performs the MAF computation for 1024 individuals for the first time.



(a) Components. (b) Overall performance.

Figure 14: Computation performance comparison among realSFS, GAMA-CPU and GAMA. (a.) Components. (b.) Overall performance comparison.

To achieve the high performance, we first design a new SFS construction algorithm to match the GPU architecture effectively. We implement the optimized algorithm on the GPU efficiently through effective shared memory utilization and warp based thread synchronization. Furthermore, we address the floating point underflow issue efficiently through logarithm transformation. As a result, on a server equipped with an NVIDIA Tesla C2070 GPU and two Intel Xeon E5520 2.27 GHz CPUs, compared with the optimized single-threaded MAF computation tool realSFS, GAMA accelerates the computation performance by around 47 times on a human genome data set of 1,024 individuals.

Acknowledgment

This work was supported by grants 617509 from the Hong Kong Research Grants Council and MRA11EG01 from Microsoft SQL Server China R&D.

7. REFERENCES

[1] ARPREC: Arbitrary Precision Package. http://crd.lbl.gov/ dhbailey/mpdist/.

[2] S. Y. Kim, K. E. Lohmueller, and et al. Estimation of allele frequency and association mapping using next-generation sequencing data. *BMC Bioinformatics*, 12:231, 2011.

[3] T. Korneliussen and R. Nielsen. realSFS: A program for estimating the site frequence spectrum: http://128.32.118.212/thorfinn/realSFS/. 2010.

[4] Y. Li, N. Vinckenbosch, and et al. Resequencing of 200 human exomes identifies an excess of low-frequency non-synonymous coding variants. *Nature Genetics*, 42(11):969–972, 2010.

[5] M. Lu, B. He, and Q. Luo. Supporting extended precision on graphics processors. In *Proceedings of the Sixth International Workshop on Data Management on New Hardware*, DaMoN '10. ACM, 2010.

[6] T. A. Manolio. Genomewide association studies and assessment of the risk of disease. *New England Journal of Medicine*, 363(2):166–176, July 2010.

[7] R. Nielsen and T. Korneliussen. Personal communication on realsfs. September 2011.

[8] X. Yi, Y. Liang, and et al. Sequencing of 50 Human Exomes Reveals Adaptation to High Altitude. *Science*, 329(5987):75–78, July 2010.