2011 International Conference on Parallel Processing

GSNP: A DNA Single-Nucleotide Polymorphism Detection System with GPU Acceleration

Mian Lu*, Jiuxin Zhao*, Qiong Luo*, Bingqiang Wang†, Shaohua Fu†, Zhe Lin†
*Hong Kong University of Science and Technology
Email: {lumian, zhaojx, luo}@cse.ust.hk
†Beijing Genomics Institute, Shenzhen
Email: {wangbingqiang, fushaohua, linzh}@genomics.org.cn

***Abstract*—We have developed GSNP, a software package with GPU acceleration, for single-nucleotide polymorphism detection on DNA sequences generated from second-generation sequencing equipment. Compared with SOAPsnp, a popular, high-performance CPU-based SNP detection tool, GSNP has several distinguishing features: First, we design a sparse data representation format to reduce memory access as well as branch divergence. Second, we develop a multipass sorting network to efficiently sort a large number of small arrays on the GPU. Third, we compute a table of frequently used scores once to avoid repeated, expensive computation and to reduce random memory access. Fourth, we apply customized compression schemes to the output data to improve the I/O performance. As a result, on a server equipped with an Intel Xeon E5630 2.53 GHZ CPU and an NVIDIA Tesla M2050 GPU, it took GSNP about two hours to analyze a whole human genome dataset whereas the CPU-based, single-threaded SOAPsnp took three days for the same task on the same machine.**

I. Introduction

Single-nucleotide polymorphism (SNP) detection is one of the most fundamental genomics applications, which is used to find DNA sequence variation for a single nucleotide. The SNP detection usually takes an excessively long running time, e.g., several days, due to the large amount of data to be processed and the high intensity of computation. As graphics processing units (GPUs) successfully accelerate various scientific applications, we propose to adopt the GPU to improve the SNP detection performance.

We focus on SNP detection on second generation DNA sequencing data, which are very short DNA fragments and may contain errors. We adopt the algorithm employing a Bayesian model, which has shown high accuracy in practice [1]. Our GPU-accelerated SNP detection system, GSNP, provides the same functionality as the popular CPU-based SNP detection tool SOAPsnp [2]. GSNP achieves a high performance through both algorithmic optimization and GPU acceleration. In particular, we make the following technical contributions.

1) We propose a sparse data representation format for a matrix central to the computation. This representation reduces memory access as well as matches the GPU's processing feature.
2) We develop a multipass sorting algorithm on the GPU to sort a large number of small arrays efficiently.
3) To avoid repeated computation and reduce random memory access, we compute a table of frequently used scores once and store them in the memory for access throughout the entire process.
4) To improve I/O performance, we apply customized compression schemes for the output data on the GPU. They provide a higher compression ratio and faster performance than general data compression algorithms for SNP results.

We evaluate our GSNP on a server equipped with an NVIDIA Tesla M2050 GPU and an Intel Xeon E5630 2.53 GHz CPU at the Beijing Genomics Institute (BGI) in Shenzhen, China, using the operational genomic data sets there. Compared with CPU-based SOAPsnp, GSNP achieved a performance speedup of around 40X. GSNP is being integrated to the production pipeline by BGI. The source code of GSNP can be downloaded from *http://www.cse.ust.hk/~lumian/gsnp.html*

The remainder of this paper is organized as follows. In Section 2, we introduce the background and related work. We overview our implementation in Section 3. Section 4 and 5 focus on optimization techniques for likelihood computation and data compression, respectively. The experimental results are reported in Section 6. We conclude the paper in Section 7.

II. Background and Related Work

A. Single-Nucleotide Polymorphism Detection

Second generation DNA sequencing produces a large number of short DNA fragments (called *reads*) at a high throughput, with an error rate of around 2%. Each read is a string of characters A, T, C, and G. Each of these four characters represents a nucleotide base (or base for short). Base A complements T, and C does G. Two bases connected via hydrogen bonds in two reverse complementary DNA strands are called a base pair (*bp*). As each read is sampled from one of the two reverse complementary DNA strands, we can calculate the corresponding reverse complementary read. The length of a read is measured in number of base pairs, or essentially the number of bases in the read. The sequencing *depth* is calculated as the total length of all short reads divided by the length of the original sequence. Since reads are randomly sampled, the original sequence may not be completely covered. The *coverage ratio* is the percentage of sampled base pairs in the sequence. Due to the dramatically reduced time and economic cost compared with the first generation sequencing, short reads DNA sequencing has been widely used since 2007.

Single-nucleotide polymorphism (SNP, pronounced [*snip*]) detection is one of the most important genome applications. It finds DNA sequence variations for a single nucleotide between different members of a species. Specifically, for each site (position) in multiple sequences, it finds whether the position holding different bases in the sequences is a SNP. For example, given two corresponding DNA fragments ATCGAG and ACCGAG from two individuals, the second site may be a SNP since T and C are two different nucleotides. T and C are called two *alleles*. For human being, there are around ten million known SNPs. SNPs contribute to different phenotypes and respond to drugs and environment. A well-known example is that the SNP of the genetic disorder Haemophilia has been identified in the X chromosome.

We focus on detecting SNPs on data generated from the second-generation sequencing equipment. The adopted algorithm is based on a Bayesian model [2], which has shown a high accuracy in practice [1].

B. Graphics Processing Unit

We use a GPU to improve SNP detection performance utilizing its massive thread parallelism and high memory bandwidth. The GPU architecture consists of hundreds of cores and can run thousands of concurrent threads in the SIMD style. The GPU *global memory* is several gigabytes. The access latency of the global memory is high. However, threads within a multiprocessor can employ *coalesced* access to group accesses on consecutive memory addresses to utilize the high GPU memory bandwidth. Additionally, there is a small but fast on-chip *shared memory* within each multiprocessor, and tens of kilobytes of *constant memory* that is cached.

C. Related Work

Most SNP detection tools are based on a Bayesian model. On the CPU, SOAPsnp [2] is one of the most successful and widely used SNP detection tools for short reads [1]. In the current release, SOAPsnp is single-threaded. SAMtools [3] is an integrated package manipulating short reads, including sorting, alignment, and SNP detection. Crossbow [4] implements the same SNP detection algorithm as SOAPsnp, but runs on Amazon EC2 to achieve a high performance. Compared to the cloud-based implementation, GSNP utilizes the GPU to improve the speed and has several key algorithmic optimizations, including memory layout, pre-computation, and compression.

Most GPU-accelerated genomics applications are on sequence alignment [5]. The work most relevant to ours is a GPU-based χ^2 test algorithm for SNP detection [6]. However, that algorithm is not specifically designed for short reads and χ^2 test is rarely used in practice today. In contrast, our GSNP aims at accelerating a commonly used SNP detection system for real-world workloads.

III. Overview

A. SNP Detection Workflow

GSNP implements the same functionality as the CPU-based SOAPsnp. Figure 1 illustrates the workflow of SNP detection in SOAPsnp. There are seven function components. Input and output data for components are shown in the ellipses. At first, a global score matrix *p_matrix* is calculated, which is used to adjust sequencing quality scores in a later step called likelihood calculation. The next six components are processed through multiple passes. The component *read_site* loads a fixed number of *sites* (a *window*) from input files. Then per-site SNP detection is performed site by site for all sites in this window. Specifically, *counting* is to collect information for each site, and then the likelihood and posterior probability values are calculated. Results are output to a file. The *recycle* component is used to re-initialize memory buffers for the next window.

There are three input files for SNP detection, which are stored as plain text in specific formats. The first file is the main data file and contains a large amount (e.g., hundreds of gigabytes) of short read alignment results ordered by their matched positions in the reference sequence. Particularly, in the alignment result, each site in the reference sequence has none or a few corresponding bases, called *aligned bases*. Note that, due to sequencing errors and SNPs, aligned bases may not be identical to the base at the same site in the reference sequence. The second file contains the reference sequence. The third file contains the prior probability for known SNPs. The first file is obtained from sequence alignment software. The reference sequence and prior probability files can be obtained from public resources. The result of SNP detection is a table, in which each row records SNP related information for a site. The result will be output to a text file in a specific format.

TABLE I
TIME BREAKDOWN (SEC) BY COMPONENTS IN SOAPSNP.

	cal_p.	*read.*	*count.*	*likeli.*	*post.*	*output*	*recycle*	Total
Ch. 1	258	101	376	12267	113	550	8214	21879
Ch.21	31	12	55	1854	17	103	1603	3675

To identify the performance bottleneck in the CPU-based SOAPsnp, we evaluate it on a server at BGI. The hardware setting and data sets are described in Section VI. Table I shows the time breakdown of the seven components of SOAPsnp. Ch. 1 and Ch. 21 are two data sets used in our evaluations. Likelihood calculation is the most time-consuming component, which takes around 56% of the overall time. Memory recycle is the second most expensive component. Our further investigation reveals that, the performance of both the likelihood calculation and memory recycle are limited by the memory bandwidth, since a dense matrix representation is adopted, which introduces large memory access overhead. The third time-consuming component is the result output,

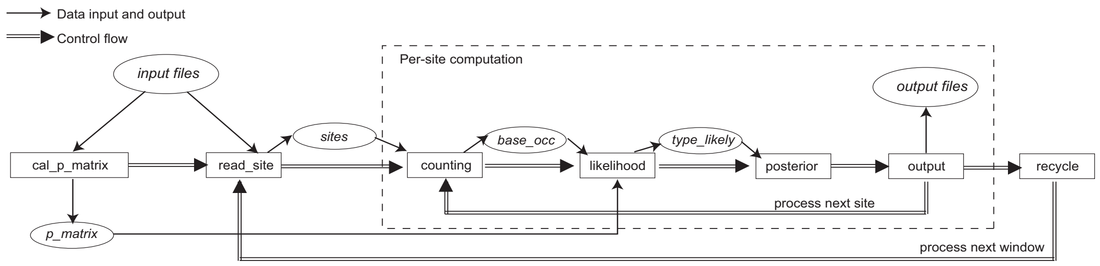


Fig. 1. Workflow of SNP detection in SOAPsnp.

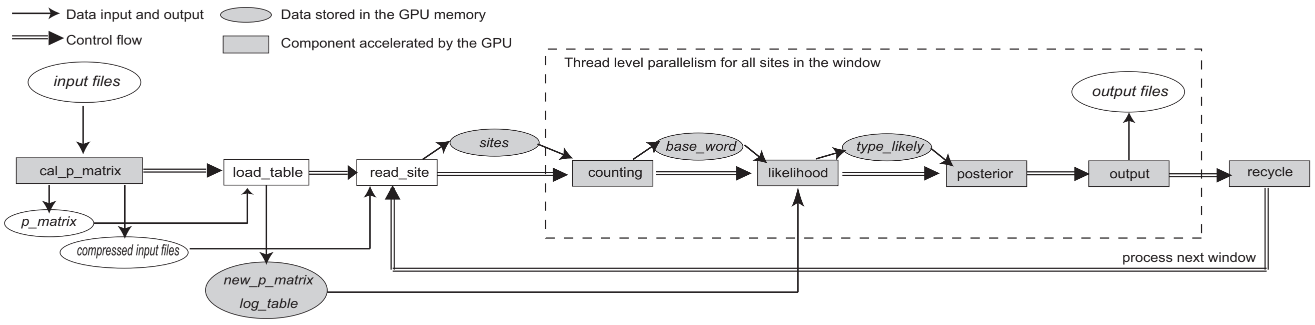


Fig. 2. Workflow of GSNP.

which is dominated by disk I/O. Outputing is more expensive than inputing due to the larger size (around 50% larger) and more complex data type parsing and conversion to plain text. The first component *cal_p_matrix* includes both I/O and computation with each taking around one half of the time. The second component *read_site* includes I/O and a number of flow control instructions. The I/O in *read_site* is more efficient than that in *cal_p_matrix* due to OS buffering.

B. GSNP Overview

Since per-site SNP detection is independent among sites, our baseline parallelization strategy on the GPU is to let one thread compute one site. Furthermore, we adopt various techniques to enhance the performance.

Figure 2 illustrates the workflow of GSNP. The GPU is used to accelerate most components, including *cal_p_matrix*, *counting*, *likelihood*, *posterior*, *output*, and *recycle*. The first component *cal_p_matrix* reads original input files once and compresses the input data on the GPU in addition to generating the *p_matrix*. An additional component in GSNP *load_table* is used to generate two new tables that are used in *likelihood* to improve the performance and to avoid any numerical inconsistency issues between the GPU and the CPU. *Counting* and *likelihood* are based on the improved data structure *base_word*, which adopts a sparse matrix representation method, while the *output* component adopts GPU-accelerated compression to improve the performance.

In the following, we present in detail our GPU-based implementation and optimization of likelihood computation, memory recycle, and result output. These three components are the most time-consuming in SOAPsnp.

IV. Likelihood Calculation

A. Likelihood Calculation Algorithm in SOAPsnp

Algorithm 1: Compute likelihood for a site in SOAPsnp.

```
1  initialize type_likely to all zeros
2  for $base \leftarrow 0$ to $3$ do
3      initialize $dep\_count$ to all zeros
4      for $score \leftarrow (q\_max - q\_min)$ to $0$ do
5          for $coord \leftarrow 0$ to $read\_len$ do
6              for $strand \leftarrow 0$ to $1$ do
7                  $occ$=$base\_occ[base{\ll}15|score{\ll}9|coord{\ll}1|strand]$
8                  for $k \leftarrow 0$ to $occ$ do
9                      $dep\_count[strand{\times}read\_len{+}coord]$++
10                     $q\_adjust$=adjust($score$, $dep\_count$)
11                     for $allele1 \leftarrow 0$ to $3$ do
12                         for $allele2 \leftarrow allele1$ to $3$ do
13                             $type\_likely[allele1{\ll}2|allele2]$
                               += likely_update($q\_adjusted$, $coord$, $base$, $allele1$, $allele2$)
```

Algorithm 2: **likely_update**($q\_adjusted$, $coord$, $base$, $allele1$, $allele2$)

```
1  $p_1 = q\_adjusted \ll 12 \mid coord \ll 4 \mid allele1 \ll 2 \mid base$
2  $p_2 = q\_adjusted \ll 12 \mid coord \ll 4 \mid allele2 \ll 2 \mid base$
3  return $\log_{10}(0.5 \times p\_matrix[p_1] + 0.5 \times p\_matrix[p_2])$
```

To simplify the presentation, we only describe the algorithm for one site. Algorithm 1 outlines the likelihood calculation for one site. Almost all existing CPU-based SNP detection tools for short reads adopt such an algorithm. The algorithm uses a matrix (denoted as *base_occ*) with four dimensions ($4 \times 64 \times 256 \times 2$, corresponding to $base \times score \times coord \times strand$) to store the number of occurrences (1 byte used) for each uniquely aligned base. The four dimensions correspond to four pieces of information about the aligned base: the base type, the sequencing quality score, the coordinates on the read, and the strand of the read. Accessing the information about all aligned bases in this matrix in a canonical order (*score-coord-strand*), the algorithm increments a counting array *dep_count*, adjusts the sequencing quality score (line 10) and iteratively updates the *type_likely* matrix. The *type_likely* matrix is finally output consisting of ten likelihood values, which correspond to the ten unique, unordered combinations of the two allele types (each allele type can be A, T, C, or G.).

B. Aligned Base Representation in GSNP

In SOAPsnp, the aligned base matrix *base_occ* uses a dense matrix representation. This representation allows sequential access during likelihood calculation following the canonical order (lines 4-7 in Algorithm 1) and direct updates to the number of occurrences in the *counting* component (the third component from left to right in Figure 1). However, this representation introduces a large number of memory accesses, and is the performance bottleneck according to our estimation as follows.

Since the size of *base_occ* for each site is fixed, the time for accessing *base_occ* can be estimated. Suppose the total number of sites is S, and the main memory bandwidth for sequential read is B_{cpu} bytes/sec. Then the time for accessing *base_occ* for the likelihood calculation (also *recycle*) is estimated as

$$\Big(\frac{S \times |base\_occ| \times 1}{B_{cpu}}\Big) sec \tag{1}$$

, where $|base\_occ|$ is the number of elements in *base_occ* per site ($4 \times 64 \times 256 \times 2 = 131,072$). We find that the estimated time is around 70% of the measured likelihood calculation time. Note that this estimation is simplistic in the hardware features, in particular, out-of-order CPU execution and prefetching, may improve or hide the memory access latency to some extent. Nevertheless, this simple estimation gives us a direct hint that the memory access time on *base_occ* is significant.

As shown in line 8 of Algorithm 1, if *occ* is zero, the computation (line 9 to 13) is not executed. Suppose the sequencing depth is X, the average non-zero percentage for *base_occ* can be estimated as

$$p_{nonzero} = \frac{X}{|base\_occ|} \times 100\% \tag{2}$$

A common sequencing depth is less than 100X, thus the non-zero percentage is up to around 0.08%.

Based on our estimation of the memory access time and the observation of the low non-zero element percentage, we adopt a sparse matrix representation in GSNP. We pack all non-zero elements into an array in the *counting* component. Since the

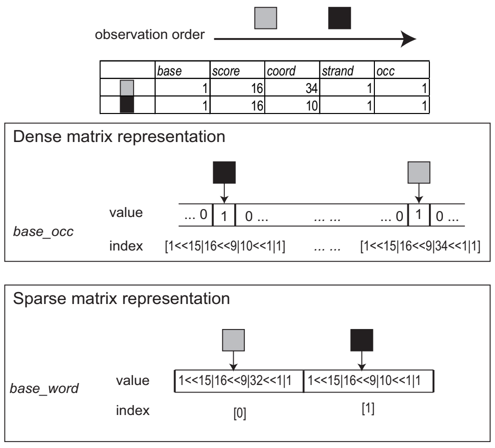


Fig. 3. Sparse and dense matrix representation of aligned bases for a site.

maximum values of base type, strand, quality scores and coordinate are 4, 2, 64, and 256, respectively, we use 32-bit word to encode all four items for an aligned base, and store them in the array denoted as *base_word*. We do not store the number of occurrences to avoid searching existing *base_word* elements to update the occurrence number in the *counting* component. Instead, each *base_word* element represents one occurrence. As the value for most non-zero elements in *base_occ* is one, this method seldom stores an element multiple times. However, the canonical order is not preserved since aligned bases for a site are unordered. Therefore, we perform a sort after counting all elements in *base_word*.

Figure 3 shows an example of the sparse representation *base_word* in comparison with the dense representation *base_occ*. According to the canonical order in Algorithm 1, the dark element in the matrix should be accessed before the grey element. The sequential access on *base_occ* can naturally keep such order. However, the grey element precedes the dark element in *base_word* since it comes earlier in the input.

There are two advantages of employing the sparse representation. First, it significantly reduces the number of memory accesses since the non-zero percentage of the dense representation is only around 0.08%. Second, since non-zero elements are packed together, threads can perform the same computation task concurrently on these elements, matching the GPU feature for SIMD style processing.

In SOAPsnp, memory recycle is the second time-consuming component due to the large amount of data to initialize. With the sparse data representation, the data to be recycled is only around 0.08% of the dense representation (estimated as the non-zero percentage in Formula 2). Furthermore, with the high memory bandwidth on the GPU, the recycle time is negligible compared with other components in GSNP, as is shown in our evaluation.

C. Multipass Sorting Network

To maintain the canonical order in *base_word*, we need to sort this array for each site. There are several studies on GPU-accelerated sorting [7], [8], [9]. However, they are inefficient to sort a large number of *base_word* arrays because they are optimized to sort a single large array. In contrast, our task is to sort a large number of arrays (up to billions), and each array is small (tens of elements).

We first implement a batch sort primitive based on *bitonic sort* to sort multiple equi-sized small arrays in parallel on the GPU. We make each thread block in CUDA handle one or multiple small arrays. If the array can fit into the shared memory, the bitonic sort can be efficiently performed on the shared memory. Otherwise, we use heuristics to utilize the shared memory in multiple passes [9]. The primitive can achieve a high performance since the bitonic sort maps to the massive parallelism on the GPU very well.

However, the sizes of *base_word* arrays vary among different sites. Modifying the primitive to the sort arrays of different sizes cannot directly achieve a high performance since the workloads are imbalanced among the threads. To utilize the batch sort primitive, a straightforward method is to set the largest array size as the batch array size. However, as time may be wasted on sorting useless data and smaller arrays may not be processed efficiently. Therefore, we adopt a multipass method, in which for each pass, we only sort arrays of a similar size.

D. New Score Table

Most computation and memory accesses for likelihood calculations are in *likely_update* (Algorithm 2). In Algorithm 1, *likely_update* is performed ten times for each aligned base. For a human genome data set, the total number of invocations of *likely_update* is around one trillion.

Algorithm 3: **opt_likely_update**($q\_adjusted$, $coord$, $base$, i)

```
1 $idx = (q\_adjusted \ll 10 \mid coord \ll 2 \mid base) \times 10 + i$
2 return $new\_p\_matrix[idx]$
```

The *likely_update* function contains a logarithm function and two non-coalesced memory reads on *p_matrix*. *p_matrix* is allocated as a four-dimension array corresponding to the adjusted score, coordinate on the read, allele type, and base type. The matrix is 8 MB in size, and can be stored in neither shared memory nor contant memory. Additionally, although the new generation GPU has L1 and L2 caches for the global memory, which are up to 48 KB and 768 KB respectively, the L1/L2 caches may not improve the performance significantly due to their small sizes. Fortunately, the number of combinations of the two allele types from the two levels of loops at line 11 and 12 in Algorithm 1 is only ten.

In GSNP, we introduce a new data structure denoted as *new_p_matrix* storing values of *p_matrix* for all combinations of *allele1* and *allele2*. Specifically, for the ith element in *p_matrix*, we calculate ten values corresponding to ten combinations of the two alleles and store them at $[10 \times i, 10 \times i + 9]$ of *new_p_matrix*. Based on this optimization, Algorithm 2 is optimized to Algorithm 3, where i is the ith combination of *allele1* and *allele2*. The size of the new score table (*new_p_matrix*) is ten times larger (80 MB), which is still affordable for the GPU, but repeated logarithm function calls are avoided and two memory accesses on *p_matrix* (line 3 in Algorithm 2) are reduced to one on *new_p_matrix* (line 2 in Algorithm 3) for each function call. We compute *new_p_matrix* once and store them in a table in the GPU memory before all likelihood calculation.

E. Shared Memory Usage

We make the shared memory hold the likelihood result (*type_likely*) since it is frequently updated (line 13 in Algorithm 1). There are ten reads and ten writes on *type_likely* for each aligned base. At the end of the likelihood calculation, the data stored in the shared memory is transferred to the global memory through coalesced writes. This way, twenty memory accesses on *type_likely* for each aligned base are performed on the shared memory. Additionally, we leave the counting array *dep_count* on the global memory since it cannot fit into the shared memory and the number of accesses is only one-tenth of that of *type_likely*.

F. Summary of Likelihood Calculation in GSNP

With the sparse data representation, shared memory usage and the new score table, Algorithm 1 is optimized to Algorithm 4. Particularly, the function *opt_likely_update* is defined in Algorithm 3.

Algorithm 4: Optimized likelihood calculation for a site.

```
1  likelihood_sort($base\_word$)
2  likelihood_comp($base\_word$)
3  function likelihood_comp($base\_word$) {
4  initialize $dep\_count$ and $s\_type\_likely$ to all zeros
5  $last\_base = 0$
6  for $i \leftarrow 1$ to #non_zero do
7      extract($base\_word[i]$,$base$, $score$, $coord$, $strand$)
8      if $base > last\_base$ then
9          initialize $dep\_count$ to all zeros
10         $last\_base$=$base$
11     $dep\_count[strand \times read\_len + coord]$ += 1
12     $q\_adjust$=adjust($score$, $dep\_count$)
13     $n = 0$
14     for $allele1 \leftarrow 0$ to $3$ do
15         for $allele2 \leftarrow allele1$ to $3$ do
16             $s\_type\_likely[allele1 \ll 2 | allele2]$ +=
               opt_likely_update($q\_adjusted$, $coord$, $base$, $n$)
17             $n$+=1
18 copy $s\_type\_likely$ to the global memory
19 }
```

G. The Consistency of GPU and CPU Results

Finally, we discuss the numerical inconsistency issue between the CPU and the GPU based implementations, which affects the likelihood calculation result. This issue is important. Genomists at BGI suggest that it is critical to keep the results consistent for their research, especially for SOAPsnp, which has accumulated a large amount of experimental data.

Modern GPUs support IEEE-compliant floating point numbers as CPUs. Thus arithmetic operators on the GPU are guaranteed to produce the same result as on the CPU. However, several mathematical functions may not produce the same result on the GPU and CPU due to different implementation details [10]. In our experiments, we found that around 0.1% of the results were different between a SNP detection algorithm implemented on the CPU and on the GPU.

Fortunately, the only mathematical function in *adjust* is a base-10 logarithm on the sequencing scores and each score is an integer between 0 and 64. Thus we calculate all base-10 logarithm results of the 64 integers on the CPU once (*log_table*) and store them into the *constant memory* of the GPU. Similarly, for the other logarithm in the algorithm, which is used in *likely_update*, we also generate a new score table (*new_p_matrix*) on the CPU. Consequently, GSNP produces exactly the same result as that of SOAPsnp. The overhead of CPU-based computation for these two tables is negligible compared with the overall time.

V. I/O and Data Compression

A. Data Input and Output

With the improved algorithm and the GPU hardware acceleration, the in-memory computation of the GSNP becomes very efficient. As a result, data input and output becomes the performance bottleneck due to slow disk I/O, which takes around 60% of the time when all the other components are accelerated using the GPU.

To improve the I/O performance in the GSNP, we consider the following constraints. First, since input files are stored in specific formats widely used by scientists, GSNP uses the same file format as SOAPsnp. Second, the disk access pattern for both read and write is already optimized to sequential in SOAPsnp. Third, the first two components of SOAPsnp read input files twice but they cannot be combined. Specifically, in Figure 1, the first component *cal_p_matrix* calculates a score matrix *p_matrix* on the input data. The input data is read again in the second component *read_site*. These two data inputs cannot be merged since the score matrix calculation in the first read requires all the data, and the second read is done window by window interleaved with the processing on each window. However, data read by the first component can be temporally stored with compression in the disk for the second component of *read_site* to read at a smaller size due to the compression. Finally, in the original *cal_p_matrix*, the I/O and computation time is roughly equal. We do not adopt the GPU to accelerate the computation due to quite a few branches and its inherent algorithmic sequentiality.

B. Output Compression Algorithms

In the GSNP, customized compression algorithms are developed for both temporary input files and output data. We describe the output compression in detail since it is more expensive. Similar algorithms are also applied to the input files.

To compress the data, GSNP does not adopt general compression algorithms, such as *gzip*, for two considerations. First, these algorithms are heavyweight and most of them are not suitable for GPU acceleration because of their inherent algorithmic sequentiality. Second, they may miss special characteristics of genome sequence data and cannot achieve a high compression ratio due to their general-purpose nature.

The output is a table containing 17 columns. Column-based compression is applied for each window. Most columns can be compressed using simple but effective algorithms. The first and second columns contain the name of the reference sequence and site ID. For sites in the same sequence, we only need to store the sequence name and the number of sites. For the three columns containing four base types, two bits are used to encode each type. Several columns related to SNPs are similar due to the low probability of SNPs. We only need to store differences for them. A certain number of columns related to the second allele are sparse. Then we only store non-zero elements for these columns.

The remaining six columns are related to sequencing quality, e.g, the average quality score. We have two observations. First, the number of distinct values is fewer than 100. Second, there are usually around tens of repeats for consecutive sites. The reason is that bases on a short read usually have the same sequencing quality. Based on these two observations, we apply two levels of compression, which is denoted as *RLE-DICT* compression. We first apply run-length encoding (RLE) to compress repeats, which produces two arrays storing the value and length for each run. Next, we use the dictionary-based encoding (DICT) to compress both run value and length arrays.

Our compression/decompression algorithms are lightweight and efficient on the CPU and GPU. Most algorithms only need a sequential scan of the data. We only implement RLE-DICT compression on the GPU for six quality related columns, which is more expensive than our other compression algorithms. RLE is implemented using the primitive *reduction* on the GPU. For DICT, we first use primitives *sort* and *unique* to build the dictionary. Then a binary search is performed for multiple elements in parallel to find their index in the dictionary. The dictionary is loaded into the constant memory if it fits. Next, we encode the index using least bits through a *map*.

Higher level applications based on the SNP detection result are to query sites satisfying certain conditions. A common operation is a sequential read on the SNP output data. The compressed output supports such operations efficiently since the compressed SNP result can be decompressed in memory by multiple passes. We also have developed decompression tools and APIs for GSNP output for further use.

VI. Evaluation

A. Experimental Setup

We evaluate GSNP using the platform at the Beijing Genomics Institute (BGI) in Shenzhen, including hardware and the operational genomic data sets. We first study the performance of CPU-based SOAPsnp. Then we investigate the performance impact of optimizations for GSNP, including sparse matrix representation, the new score table, and shared memory usage. We further study the effectiveness and efficiency of compression algorithms. Finally, we show the end-to-end performance comparison for all 24 sequences.

Hardware setup. We evaluate GSNP in a Dell PowerEdge M610x server equipped with an NVIDIA Tesla M2050 GPU and Intel Xeon E5630 2.53 GHz CPUs (8 cores, 16 threads in total). M2050 consists of 448 cores and has 3GB memory. The measured GPU memory bandwidths for coalesced and random accesses are 82GB/sec and 3.2GB/sec, respectively. The global memory of M2050 has L1 and L2 caches sized 48 KB and 768 KB, respectively. The server has 64GB main memory with a measured bandwidth 4.2GB/sec for sequential access. The sequential disk I/O is around 90MB/sec.

Implementation details. We develop GSNP using NVIDIA CUDA C 3.2 in 64-bit SUSE Linux Enterprise 11. SOAPsnp 1.03 (*http://soap.genomics.org.cn/soapsnp.html*) is adopted as our CPU counterpart, which is developed using a single thread. We have developed a multi-threaded version of SOAPsnp and it achieved a 3-4 times speedup using 16 threads on the CPU over the original single-threaded SOAPsnp. This limited speedup is mainly because the algorithm is bounded by memory bandwidth. In our evaluation, we still adopt the official single-thread SOAPsnp as the CPU-based counterpart. We also report the results of the optimized sequential CPU implementation (denoted as GSNP_CPU) in a few experiments, which adopts the same algorithm as GSNP but without GPU acceleration. We set the default window size for GSNP and GSNP_CPU as 256,000, and for SOAPsnp as 4,000. For such a setup, GSNP_CPU and SOAPsnp both consume around 2 GB main memory, and GSNP consumes around 1 GB main memory and 1.5 GB GPU memory. The performance is nearly unchanged if the windows become further larger for all three.

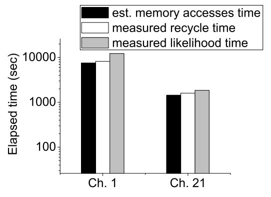
TABLE II
HUMAN CHROMOSOME 1 AND 21.

	#sites	Seq. dep	#reads	Coverage	Input	Output
Ch.1	247 M	11X	44 M	88%	12 GB	17 GB
Ch.21	47 M	9.6X	6 M	68%	2 GB	3 GB

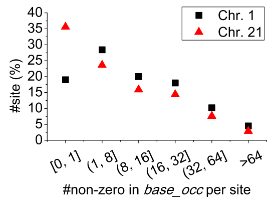
Data sets. We have a complete human genome data set stored in 24 separate files, each corresponding to one of the 24 DNA sequences. The total number of reads is around 500 million with a length of 100 base pairs each. The total size of input data is around 142 GB. We mainly use Chromosome 1 (Ch. 1) and 21 (Ch. 21) for performance study, which are the largest and smallest sequence, respectively. Table II shows the characteristics of the two data sets. The output size is for the file generated by SOAPsnp.

B. Performance Study of CPU-based SOAPsnp

Estimated memory access time on *base_occ*. The time for accessing the matrix in dense representation *base_occ* in likelihood calculation and memory recycle in SOAPsnp can be estimated using Formula 1 in Section IV-B. Figure 4(a) shows that from the estimation the majority of the likelihood calculation (65-70%) and memory recycle (89-92%) is spent on the memory access on *base_occ*. In other words, if the memory access on zeros can be eliminated, the speedup for the optimized CPU-based implementation is at least three and ten times for the likelihood calculation and memory recycle, respectively.



(a) Memory access time.



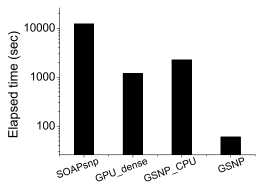
(b) Sparsity for *base_occ* matrix.

Fig. 4. (a) Comparison of the estimated memory access time on *base_occ* and the measured likelihood calculation and memory recycle time. (b) Percentage of sites with different numbers of non-zero elements in *base_occ* matrix.

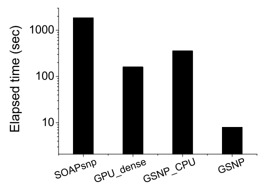
Sparsity of *base_occ*. Figure 4(b) shows the sparsity of *base_occ*. The vertical axis is the percentage of sites, and the horizontal axis is the number of non-zero elements in *base_occ* per site. This shows that most sites have only tens of non-zero elements. Since the total number of elements stored in *base_occ* is 131,072, the non-zero elements are up to around 0.08% in *base_occ* matrix for most sites, which is consistent with our estimation using Formula (2) in Section IV-B.

C. Performance of Likelihood Calculation in GSNP

In the CPU-based implementation SOAPsnp, the likelihood computation is the performance bottleneck, which takes around 56% of the overall time. We first study the performance impact of specific optimizations for the likelihood computation.



(a) Ch. 1



(b) Ch. 21

Fig. 5. Time of likelihood calculation: dense representation on the CPU (SOAPsnp), dense representation on the GPU (GPU_dense), sparse representation on the CPU (GSNP_CPU), sparse representation on the GPU (GSNP).

Sparse matrix representation. We first show the performance comparison of the likelihood calculation employing different data representation methods for aligned bases on both the GPU and CPU. Figure 5 shows that the GSNP_CPU outperforms SOAPsnp by around 4-5 times. With the GPU acceleration, GSNP is two orders of magnitude faster than SOAPsnp, and around 30X faster than GSNP_CPU. Moreover, the GPU-based implementation employing dense representation is around 14-17X slower than GSNP, which shows the efficiency of using sparse representation.

Time of *likelihood_sort* and *likelihood_comp*. Figure 6 shows the elapsed time of two steps (*likelihood_sort* and *likelihood_comp*) in likelihood calculation employing the sparse representation. It shows that the speedup for the sorting and computing on the GPU is around 22X and 40X, respectively. Bitonic sort has a higher complexity than quick sort adopted in GSNP_CPU, thus the speedup is less significant.

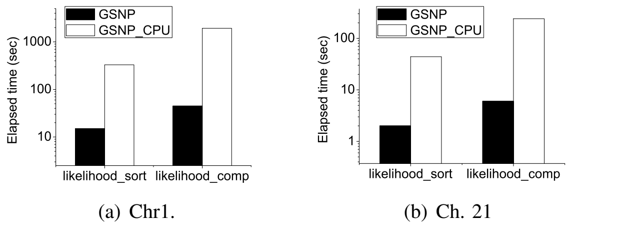


Fig. 6. Elapsed time of *likelihood_sort* and *likelihood_comp* for the likelihood calculation employing sparse representation on the GPU and CPU.

Batch sort primitive performance. We first measure the throughput (as defined in Formula 3) of the batch sort primitive through randomly generated data. We compare the performance of three implementations: (1) OpenMP based parallel CPU quick sort (16 threads), which uses one thread to sort one array. (2) Our batch sort primitive on the GPU. (3) GPU-based radix sort [11], which sorts multiple arrays sequentially. Figure 7(a) shows that the third one underutilizes GPU hardware resources and has very low throughput. Our GPU-based batch sort has around 1.5 timers higher throughput than the parallel CPU sort. Additionally, the throughput decreases when the batch array size becomes larger due to the higher sorting cost.

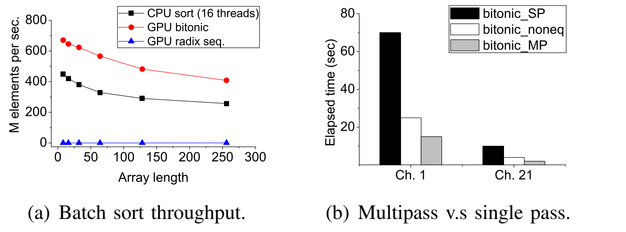


Fig. 7. Performance of *likelihood_sort* on the GPU and CPU

Multipass sorting. We study the performance of multipass sorting for *base_word*. The size of a *base_word* array is close to the number of non-zero elements in *base_occ*. The single pass uses the largest array size in the batch as the batch array size. The multipass adopts six passes, which are for array size [0, 1], (1, 8], (8, 16], (16, 32], (32, 64], and larger than 64. We also compare with the implementation sorting different size arrays directly using bitonic sort on the GPU. Figure 7(b) shows that the multipass bitonic sort (*bitonic_MP*) is around five times faster than the single pass (*bitonic_SP*). Through further investigation, we find that most arrays are sorted using size 128 and 256 in the single pass, which are larger than their real size. The total number of elements sorted for single pass is around four times larger than multipass through calculation. With a higher sorting throughput for smaller arrays, the overall speedup is around five times. The multipass is also more efficient than sorting different size arrays directly (*bitonic_noneq*) due to more balanced workloads.

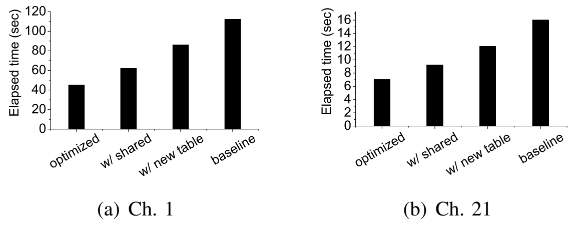


Fig. 8. Performance comparison of *likelihood_comp* on the GPU for the optimized implementation (*optimized*), only with the new score table (*w/ new table*), only with the shared memory used (*w/ shared*), and the baseline implementation without two optimizations (*baseline*).

New score table and shared memory usage. We study the performance impact of two optimizations for the likelihood computing step (*likelihood_comp*) in GSNP: (1) the new score table (*new_p_matrix*), and (2) shared memory usage. The performance numbers exclude the sorting step since the optimizations are not applicable to it. Figure 8 shows that the optimized GSNP is around 2.4 times faster than the baseline implementation. Using shared memory or new score table individually reduces the time of baseline implementation to around 55% and 78%, respectively. Shared memory improves the performance more since it directly eliminates twenty non-coalesced memory accesses on the global memory for each *base_word* (line 16 in Algorithm 4). The new score table reduces twenty non-coalesced memory reads on *p_matrix* to half, and eliminates ten logarithm functions for each *base_word*. Due to the high latency for non-coalesced access, the shared memory contributes more to improve the overall performance.

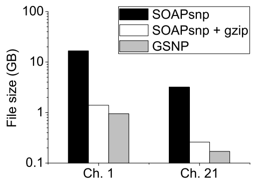
System information for likelihood calculation. To further understand the performance impact of the two optimizations, we investigate GPU system information (CUDA Visual Profiler [12]). Table III shows the number of instructions issued (*#inst. PW*), number of global memory loads (*#g_load*) and stores (*#g_store*), number of shared memory loads (*#s_load PW*) and stores (*#s_store PW*) for different implementations for Ch. 1 (Ch. 21 has similar conclusions). *PW* indicates that the counter is for a *warp* (32 threads) on a multiprocessor. This shows that using the shared memory reduces the number of global memory loads and stores to around 70% and 68% of the baseline, respectively. The number of memory accesses on the shared memory is close to the number of reduced memory accesses on the global memory. With the new score table, the number of instructions and global memory loads are reduced to around 73% and 64% of the baseline, respectively. Combining two techniques, the numbers of instructions and total number of global memory accesses are reduced to around 70% and 51%, respectively, for the optimized implementation.

TABLE III
HARDWARE COUNTERS FOR *likelihood_comp* WITH OPTIMIZATION TECHNIQUES ON THE GPU (CH. 1).

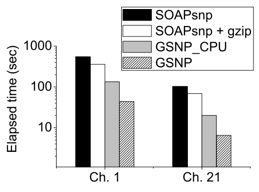
	baseline	**w/ shared**	**w/ new table**	**optimized**
#inst. PW	3.3×10^{10}	3.1×10^{10}	2.4×10^{10}	$2.3{\times}10^{10}$
#g_load	3.3×10^{8}	2.3×10^{8}	2.1×10^{8}	1.2×10^{8}
#g_store	3.7×10^{8}	2.5×10^{8}	3.6×10^{8}	2.4×10^{8}
#s_load PW	0	1.1×10^{8}	0	1.1×10^{8}
#s_store PW	0	1.1×10^{8}	0	1.1×10^{8}

D. Effectiveness and Efficiency of Data Compression

With the improved data structure and GPU acceleration for the likelihood computation and memory recycle, the disk I/O for result output becomes the performance bottleneck in GSNP, which takes around 60% time when all the other components are accelerated by the GPU. We study the compression ratio and speed to show the performance improvement from our customized compression techniques.



(a) Output size



(b) Output speed

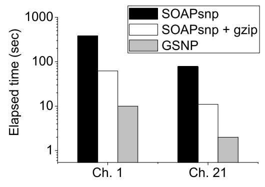
Fig. 9. The output file size and speed of SOAPsnp, SOAPsnp with *gzip* compression, and GSNP with customized compression algorithms.

Compression ratio. We study the compression ratio for the output. The *gzip* algorithm is utilized through its programming API *zlib* [13] (the same for following evaluations). Figure 9(a) shows the size of SOAPsnp output, SOAPsnp output with *gzip*, and GSNP. It shows that the SOAPsnp output and with *gzip* are around 14-16X and 1.5X larger than GSNP, respectively. This indicates that our algorithms take more advantage of the data characteristics.

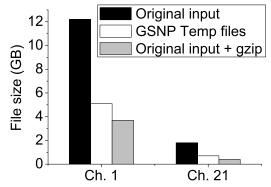
Output speed. The time for output speed includes compression (if any) and output. Figure 9(b) shows that *gzip* is around three times slower than GSNP_CPU due to more expensive computation. GSNP further accelerates the output by around three times using the GPU acceleration. Compared with SOAPsnp, the GSNP output is around 13-15 times faster.

Decompression speed. The measurement of decompression performance is to sequential read the original data once. For the compressed data, it is loaded from the disk and decompressed in-memory. Otherwise a sequential read is performed on the SOAPsnp output without compression. The GPU is not adopted since the decompression algorithms are simple and efficient on the CPU. Disk I/O dominates the decompression time. Figure 10(a) shows that reading compressed results of GSNP is around 40 times faster than SOAPsnp, and also outperforms *gzip* by around 6 times.

Compressed input data. We also study the size of the temporary file generated from *cal_p_matrix*, which is read by *read_site* to improve the data input. Figure 10(b) shows that the compressed data is around one-third size of the original input, and is comparable to *gzip*. Since the input data is more general than the output, *gzip* achieves a better compression ratio. There is overhead for *cal_p_matrix* generating temporary files, however, the overall performance of *cal_p_matrix* and *read_site* together are improved (summarized in Table IV).



(a) Decompression speed



(b) Temporary inpute file size.

Fig. 10. (a). Output decompression speed for SOAPsnp, SOAPsnp with *gzip*, and GSNP. (b). Data size of compressed input files generated by *cal_p_matrix*.

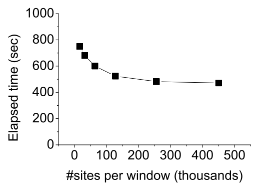
E. Overall Performance Comparison

Time of GSNP components. Table IV summarizes the elapsed time for different components in GSNP, and the corresponding speedup (the number in parentheses) compared with SOAPsnp (shown in Table I). This shows that for the two most time-consuming components *likelihood* and *recycle* on the CPU, GSNP accelerates them by two to three orders of magnitude. Due to the compression, the output performance is also improved by 13-15 times. The speedup of *counting* and *posterior* are less significant due to the data transfer overhead between the main and the GPU memory. The first component is slightly slowed down due to temporary files generated. However, the first and second components *cal_p_matrix* and *read_site* together save around 42 and 3 seconds for Ch. 1 and 21, respectively. *cal_p_matrix* here includes the time for the table *new_p_matrix* and *log_table* generation and loading to the GPU memory, which take around 2 seconds. The overall speedup is around 42-50X compared with SOAPsnp on the CPU. The speedup for Ch. 21 is more significant since a higher percentage of sites have no aligned bases (around 30%), and can benefit more from our algorithms.

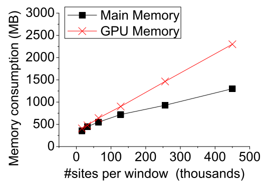
TABLE IV
TIME BREAKDOWN (SEC) OF GSNP AND THE SPEEDUP COMPARED WITH SOAPSNP ON THE CPU.

	cal_p.	*read.*	*count.*	*likeli.*	*post.*	*output*	*recycle*	Total
Ch.1	297	20(5)	87(4)	60(204)	16(7)	44(13)	3(2738)	527(42)
Ch.21	37	3(4)	14(4)	8(231)	3(6)	7(15)	1(1603)	73(50)

Performance impact of window size. We also show the time and memory efficiency with the window size varied. Figure 11(a) shows that the time slightly increases when the window size decreases from 450,000 to 128,000, and that it increases more dramatically when the window size is less than 128,000. Such a performance slowdown is from the overhead introduced by more windows and under-utilized hardware for small windows. Additionally, the time nearly



(a) Elapsed time



(b) Memory consumption

Fig. 11. Elapsed time and memory consumption with the number of sites per window varied in GSNP for Ch. 1.

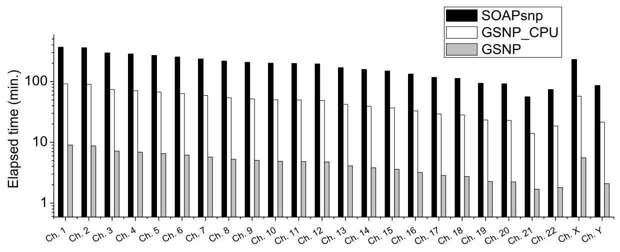


Fig. 12. Performance comparison of SOAPsnp, GSNP_CPU, and GSNP.

remains a constant if the window becomes larger than 256,000. Figure 11(b) shows the memory consumption with window size varied. Associating two figures, when the window size is set to 128,000, the speedup is still around 32 times, but both the GPU and CPU memory consumption are less than 1 GB, which are available for most hardware configurations today.

End-to-end performance comparison. Finally, we compare the performance for all human chromosomes. Figure 12 shows that GSNP has a speedup of at least 40 times compared with SOAPsnp. The total time to finish this whole human genome workload for SOAPsnp is around three days on the CPU. Our GSNP only takes around two hours.

VII. Conclusion

Since 2007, second generation DNA sequencing has become popular and also made the computation more challenging due to larger amounts of data to process. We have developed GSNP, an efficient GPU-accelerated SNP detection tool for second generation DNA sequencing. Our GSNP achieves a high performance through algorithm improvement and GPU-specific optimization. We adopt a sparse matrix representation to reduce memory access. We implement a multipass sorting algorithm on the GPU to sort a large number of small arrays efficiently. Random memory access on the GPU are reduced through shared memory and a new table storing frequently used values. Additionally, we have developed a set of high performance compression algorithms to improve the I/O performance. With the operational data sets and the hardware platform provided by genomists in BGI, our GSNP achieves a speedup of around 40 times compared with the popular SNP detection tool SOAPsnp [2] on the CPU. GSNP will be released soon as an updated version for SOAPsnp. We believe it will have a significant impact on genomics research.

Acknowledgment

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

References

[1] "YanHuang Project," http://yh.genomics.org.cn/.
[2] R. Li, Y. Li, X. Fang, H. Yang, J. Wang, K. Kristiansen, and J. Wang, "SNP detection for massively parallel whole-genome resequencing," *Genome Research*, vol. 19, no. 6, pp. 1124–1132, June 2009.
[3] H. Li, B. Handsaker, A. Wysoker, T. Fennell, J. Ruan, N. Homer, G. Marth, G. Abecasis, R. Durbin, and 1000 Genome Project Data Processing Subgroup, "The Sequence Alignment/Map format and SAMtools," *Bioinformatics*, vol. 25, no. 16, pp. 2078–2079, August 2009.
[4] B. Langmead, M. Schatz, J. Lin, M. Pop, and S. Salzberg, "Searching for SNPs with cloud computing," *Genome Biology*, vol. 10, no. 11, November 2009.
[5] A. Gharaibeh and M. Ripeanu, "Size Matters: Space/Time Tradeoffs to Improve GPGPU Applications Performance," in *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, November 2010, pp. 1–12.
[6] R. Jiang, F. Zeng, W. Zhang, X. Wu, and Z. Yu, "Accelerating Genome-Wide Association Studies Using CUDA Compatible Graphics Processing Units," in *Proceedings of the 2009 International Joint Conference on Bioinformatics, Systems Biology and Intelligent Computing*, August 2009, pp. 70–76.
[7] N. Govindaraju, J. Gray, R. Kumar, and D. Manocha, "GPUTeraSort: High Performance Graphics Co-processor Sorting for Large Database Management," in *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, 2006, pp. 325–336.
[8] N. Satish, C. Kim, J. Chhugani, A. D. Nguyen, V. W. Lee, D. Kim, and P. Dubey, "Fast Sort on CPUs and GPUs: A Case for Bandwidth Oblivious SIMD Sort," in *Proceedings of the 2010 International Conference on Management of Data*, 2010, pp. 351–362.
[9] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander, "Relational Joins on Graphics Processors," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 511–524.
[10] NVIDIA, "Appendix C. Mathematical Functions," *NVIDIA CUDA C Programming Guide*.
[11] "Thrust Project," http://code.google.com/p/thrust/.
[12] "NVIDIA Compute Unified Device Architecture," http://www.nvidia.com/cuda.
[13] "ZLIB Library," http://zlib.net/.