GWAS on GPUs: Streaming Data from HDD for Sustained Performance

Lucas Beyer and Paolo Bientinesi

RWTH Aachen University,
Aachen Institute for advanced study in Computational Engineering Science, Germany
{beyer,pauldj}@aices.rwth-aachen.de

Abstract. In the context of genome-wide association studies (GWAS), one has to solve long sequences of generalized least-squares problems; such a task has two limiting factors: execution time –often in the range of days or weeks– and data management –data sets in the order of Terabytes. We present an algorithm that obviates both issues. By pipelining the computation, and thanks to a sophisticated transfer mechanism, we stream data from hard disk to main memory to GPUs and achieve sustained performance; with respect to a highly-optimized CPU implementation, our algorithm shows a speedup of 2.6x. Moreover, the approach lends itself to multiple GPUs and attains almost perfect scalability. When using 4 GPUs, we observe speedups of 9x over the aforementioned CPU implementation, and 488x over ProbABEL, a widespread biology library.

Keywords: GWAS, generalized least-squares, computational biology, out-of-core computation, high-performance, multiple GPUs, data transfer, multibuffering, streaming, big data.

1 GWAS, Their Importance and Current Implementations

In a nutshell, the goal of a genome-wide association study (GWAS) is to find an association between genetic variants and a specific trait such as a disease [1]. Since there is a tremendous amount of such genetic variants, the computation involved in GWAS takes a long time, ranging from days to weeks and even months [2]. In this paper, we look at *OOC-HP-GWAS*, currently the fastest algorithm available, and show how it is possible to speed it up by exploiting the computational power offered by modern graphics accelerators.

The solution of GWAS boils down to a sequence of generalized least squares (GLS) problems involving huge amounts of data, in the order of Terabytes. The challenge lies in sustaining GPU's performance, avoiding idle time due to data transfers from hard disk (HDD) and main memory. Our solution, *cuGWAS*, combines three ideas: the computation is pipelined through GPU and CPU, the transfers are executed asynchronously, and the data is streamed from HDD to main memory to GPUs by means of a two-level buffering strategy. Combined, these mechanisms allow cuGWAS to attain almost perfect scalability with respect to the number of GPUs; when compared to OOC-HP-GWAS and another

F. Wolf, B. Mohr, and D. an Mey (Eds.): Euro-Par 2013, LNCS 8097, pp. 788–799, 2013.

widespread GWAS library, ProbABEL, our code is respectively 9 and 488 times faster.

In the first section of this paper, we introduce the reader to GWAS and the computations involved therein. We then give an overview of OOC-HP-GWAS, upon which we build cuGWAS, whose key techniques we explain in Section 3 and which we time in Section 4. We provide some closing remarks in Section 5.

1.1 Biological Introduction to GWAS

The segments of the DNA that contain information about protein synthesis are called *genes*. They encode so-called *traits*, which are features of physical appearance of the organism –like eye or hair color– as well as internal features of the organism –like blood type or resistances to diseases. The hereditary information of a species consists of all the genes in the DNA, and is called *genome*; this can be visualized as a book containing instructions for our body. Following this analogy, the letters in this book are called *nucleotides*, and determining their order is referred to as *sequencing* the genome. Even though the genome sequence of every individual is different, within one species most of it (99.9% for humans) stays the same. When a single nucleotide of the DNA differs between two individuals of the same species, this difference is called a single-nucleotide polymorphism (*SNP*, pronounced "snip") and the two variants of the SNP are referred to as its *alleles*.

Genome-wide association studies compare the DNA of two groups of individuals. All the individuals in the *case group* have a same trait, for example a specific disease, while all the individuals in the *control group* do not have this trait. The SNPs of the individuals in these groups are compared; if one variant of a SNP is more frequent in the case group than in the control group, it is said that the SNP is *associated* with the trait (disease). In contrast with other methods for linking traits to SNPs, such as inheritance studies or genetic association studies, GWAS consider the whole genome [1].

1.2 The Importance of GWAS

We gathered insightful statistics about all published GWAS [3]. Since the first GWAS started to appear in 2005 and 2006, the amount of yearly published studies has constantly increased, reaching more than 2300 studies in 2011. This trend is summarized in the left panel of Fig. 1, showing the median SNP-count of each year's studies along with error-bars for the first and second quartiles. One can observe that while GWA studies started out relatively small, since 2009 the amount of analyzed SNPs is growing tremendously. Besides the number of SNPs, the other parameter relevant to the implementation of an algorithm is the *sample size*, that is the total number of individuals of both the case and the control group. What can be seen in Fig. 1b is that while it has grown at first, in the past four years the median sample size seems to have settled around

10 000 individuals. It is apparent that, in contrast to the SNP count, the growth of the sample size is negligible. This data, as well as discussions with biologists, confirm the need for algorithms and software that can compute a GWAS with even more SNPs, and faster than currently possible.

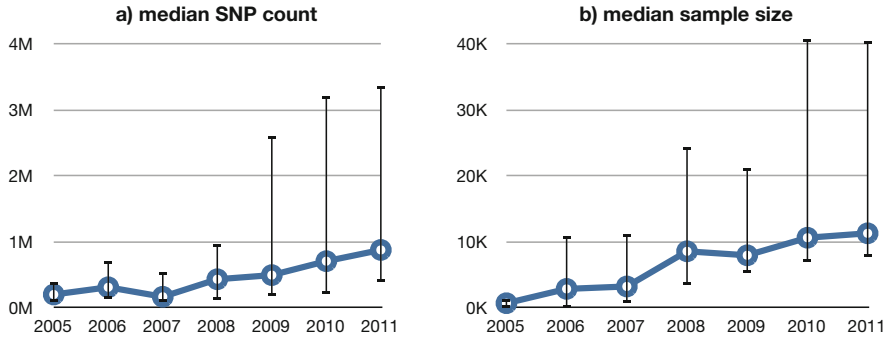


Fig. 1. The median, first and second quartile of a) the SNP-count and b) the sample size of the studies each year

1.3 The Mathematics of GWAS

The GWAS can be expressed as a variance component model [4] whose solution r_i can be formulated as

$$r_i = (X_i^T M^{-1} X_i)^{-1} X_i^T M^{-1} y, \quad i = 1 \ldots m \ , \tag{1}$$

where m is in the millions, and all variables on the right-hand side are known. This sequence of equations is used to compute in r_i the relations between variations in y (the *phenotype*[1]) and variations in X_i (the *genotype*). Each equation is responsible for one SNP, meaning that the number m of equations corresponds to the number of SNPs considered in the study.

Figure 2 captures the dimensions of the objects involved in one such equation. The height n of the matrices X_i and M and of the vector y corresponds to the number of samples, thus each row in the *design-matrix* $X_i \in \mathbb{R}^{n \times p}$ corresponds to a piece of each individual's genetic makeup (i.e. information about one SNP), and each entry in $y \in \mathbb{R}^n$ corresponds to an individual's phenotype.[2] $M \in \mathbb{R}^{n \times n}$ models the relations amongst the individuals, e.g. two individuals being in the same family. Finally, an important feature of the matrices X_i is that they can be partitioned as $(X_L | X_{R_i})$, where X_L contains fixed covariates such as age and sex

[1] A phenotype is the observed value of a certain trait of an individual. For example, if the studied trait was the hair color, the phenotype of an individual would be the one of "blonde", "brown", "black" or "red".

[2] In the example of the body height as a trait, the entries of y would then be the heights of the individuals.

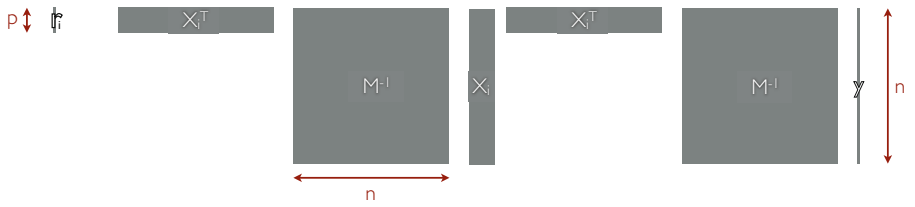


Fig. 2. The dimensions of a single instance of (1)

and thus stays the same for any i, while X_{R_i} is a single column vector containing the genotypes of the i-th SNP of all considered individuals.

Even though (1) has to be computed for every single SNP, only the right part of the design-matrix X_{R_i} changes, while X_L, M and y stay the same.

1.4 The Amount of Data and Computation Involved

We analyze the storage size requirements for the data involved in GWAS. Typical values for p range between 4 and 20, but only one column varies with m. According to our analysis in Section 1.2, we consider $n = 10\,000$ as the size of a study. As of June 2012, the SNP database *dbSNP* lists 187 852 828 known SNPs for humans [5], so we consider $m = 190\,000\,000$. With these numbers, assuming that all data is stored as double precision floating point numbers, [3] the size of y and M is about 80 MB and 800 MB, respectively; both fit in main memory and in the GPU memory. The output r reaches 30 GB, close to the main memory of current high-end systems and too big to fit in a GPU's 6 GB of memory. Weighting in at 14 TB, X is too big to fit into the memory of any system in the foreseeable future and has to be streamed from disk.

In the field of bioinformatics, the ProbABEL [6] library is frequently used for genome-wide association studies. On a Sun Fire X4640 server with an Intel Xeon CPU 5160 (3.00 GHz), the authors report a runtime of almost 4 hours for a problem with $p = 4$, $n = 1500$ and $m = 220\,833$, and estimate the runtime with $m = 2\,500\,000$ to be roughly 43 hours[4] –almost two days. Compared to the current demand, $m = 2.5$ million is a reasonable amount of SNPs, but a population size of only $n = 1500$ individuals is clearly much smaller than the present median (Fig. 1). The authors state that the runtime grows more than linearly with n and, in fact, tripling up the sample size from 500 to 1500 increased their runtime by a factor of 14. Coupling this fact with the median sample size of about 10 000 individuals, the computation time is bound to reach weeks or even months.

[3] Which may or may not be the optimal storage type. More discussion with biologists and analysis of the operations is necessary in order to find out whether `float` is precise enough. If that was the case, the sizes should be halved.

[4] We only consider what the authors called the *linear model* with the `--mmscore` option as this solves the exact problem we tackle.

2 Prior Work: The OOC-HP-GWAS Algorithm

Presently, the fastest available algorithm for solving (1) is OOC-HP-GWAS [4]. Since our work builds upon this CPU-only algorithm, we describe its salient features. Other approaches to GWAS on GPU(s) include [10] and [11].

2.1 Algorithmic Features

OOC-HP-GWAS exploits the the symmetry and the positive definiteness of the matrix M, by decomposing it through a Cholesky factorization $LL^T = M$. Since M does not depend on i, this decomposition can be computed once as a preprocessing step and reused for every instance of (1). Substituting $LL^T = M$ into (1) and rearranging, we obtain

$$r_i = (\underbrace{(L^{-1}X_i)^T}_{\tilde{X}_i} \underbrace{L^{-1}X_i}_{\tilde{X}_i})^{-1} \underbrace{(L^{-1}X_i)^T}_{\tilde{X}_i} \underbrace{L^{-1}y}_{\tilde{y}} \quad \text{for } i = 1 \ldots m \ , \tag{2}$$

effectively replacing the inversion and multiplication of M with the solution of a triangular linear system (`trsv`).

The second problem-specific piece of knowledge that is exploited by OOC-HP-GWAS is the structure of $X = (X_L|X_R)$: X_L stays constant for any i, while X_R varies; plugging $X_i = (X_L|X_{R_i})$ into (2) and moving the constant parts out of the loop leads to an algorithm that takes advantage of the structure of the sequence of GLS shown in Listing 1.1. The acronyms correspond to BLAS calls. A more detailed derivation can be found in [4].

Listing 1.1. Solution of the GWAS-specific sequence of GLS (1)

```
1  L   ← potrf M                    (LL^T = M)
2  Xl  ← trsm L, Xl                 (X̃_L = L^{-1} X_L)
3  y   ← trsv L, y                  (ỹ = L^{-1} y)
4  rt  ← gemv Xl, y                 (r̃_T = X̃_L^T ỹ)
5  Stl ← syrk Xl                    (S_TL = X̃_L^T X̃_L)
6  for i in 1..m:
7      Xri ← trsv L, Xri            (X̃_{R_i} = L^{-1} X_{R_i})
8      Sbl ← dot Xri, Xl            (S_{BL_i} = X̃_{R_i}^T X̃_L)
9      Sbr ← syrk Xri               (S_{BR_i} = X̃_{R_i}^T X̃_{R_i})
10     rb  ← dot Xri, y             (r̃_{B_i} = X̃_{R_i}^T ỹ)
11     r   ← posv S, r              (r_i = S_i^{-1} r̃_i)
```

2.2 Implementation Features

Two implementation features allow OOC-HP-GWAS to attain near-perfect efficiency. First, by packing multiple vectors X_{R_i} into a matrix X_{R_b}, the slow

BLAS-2 routine to solve a triangular linear system (`trsv`) at Line 7 can be transformed into a fast BLAS-3 `trsm`. Second, Listing 1.1 is an *in-core* algorithm that cannot deal with an X_R which does not fit into main memory. This limitation is overcome by turning the algorithm into an *out-of-core* one, in this case using a double-buffering technique: While the CPU is busy computing the block b of X_R in a primary buffer, the next block $b+1$ can already be loaded into a secondary buffer through asynchronous I/O using the POSIX `libaio`. The full OOC-HP-GWAS algorithm is shown in Listing 1.2. This algorithm attains more than 90% efficiency.

Listing 1.2. The full OOC-HP-GWAS algorithm

```
1  L   ← potrf M                    (LL^T = M)
2  Xl  ← trsm L, Xl                 (X̃_L = L^{-1} X_L)
3  y   ← trsv L, y                  (ỹ = L^{-1} y)
4  rt  ← gemv Xl, y                 (r̃_T = X̃_L^T ỹ)
5  Stl ← syrk Xl                    (S_TL = X̃_L^T X̃_L)
6  aio_read Xr[1]
7  for b in 1..blockcount:
8      aio_read Xr[b+1]
9      aio_wait Xr[b]
10     Xrb ← trsm L, Xrb            (X̃_{R_b} = L^{-1} X_{R_b})
11     for Xri in Xr[b]:
12         Sbl ← gemm Xri, Xl       (S_{BL_i} = X̃_{R_i}^T X̃_L)
13         Sbr ← syrk Xri           (S_{BR_i} = X̃_{R_i}^T X̃_{R_i})
14         rb  ← gemv Xri, y        (r̃_{B_i} = X̃_{R_i}^T ỹ)
15         r   ← posv S, r                  (r_i = S_i^{-1} r̃_i)
16     aio_wait r[b-1]
17     aio_write r[b]
18 aio_wait r[blockcount]
```

3 Increasing Performance by Using GPUs

While the efficiency of the OOC-HP-GWAS algorithm is satisfactory, the computations can be sped up even more by leveraging multiple GPUs. With the help of a profiler, we determined (confirming the intuition), that the `trsm` at line 10 in Listing 1.2 is the bottleneck. Since cuBLAS provides a high-performance implementation of BLAS-3 routines, `trsm` is the best candidate to be executed on GPUs. In this section, we introduce cuGWAS, an algorithm for a single GPU, and then extend it to an arbitrary number of GPUs.

Before the `trsm` can be executed on a GPU, the algorithm has to transfer the necessary data. Since the size of L is around 800 MB, the matrix can be sent once during the preprocessing step and kept on the GPU throughout the entire computation. Unfortunately, the whole X_R matrix weights in at several TB, way

more than the 2 GB per buffer limit of a modern GPU. The same holds true for the result $\tilde{X}_{R_b}$ of the `trsm`, which needs to be sent back to main memory. Thus, there is no other choice than to send X_R in a block-by-block fashion, each block X_{R_b} weighting at most 2 GB minus the size of L.

When profiled, a naïve implementation of the algorithm displays a pattern (Fig. 3) typical for applications in which GPU-offloading is an after-thought: both GPU (green) and CPU (gray) need to wait for the data transfer (orange); furthermore, the CPU is idle while the GPU is busy and vice-versa.

Fig. 3. Profiled timings of the naïve implementation

Our first objective is to make use of the CPU while the GPU computes the `trsm`. Regrettably, all operations following the `trsm` (i.e. the for-loop at Lines 11–15 in Listing 1.2, which we will call the *S-loop*) are dependent on its result and thus cannot be executed in parallel. A way to break out of this dependency is to delay the S-loop by one block, in a pipeline fashion, so that the S-loop relative to the b-th block of X_R is executed on the CPU, while the GPU executes the `trsm` with the $(b+1)$-th block. Thanks to this pipelining, we have broken the dependency and introduced more parallelism, completely removing the gray part of Fig. 3.

3.1 Streaming Data from HDD to GPU

The second problem with the aforementioned naïve implementation is the time wasted due to data transfers. Modern GPUs are capable of *overlapping* data transfers with computation. If properly exploited, this feature allows us to eliminate any overhead, and thus attain sustained peak performance on the GPU.

The major obstacle is that the data is already being double-buffered from the hard-disk to the main memory. A quick analysis shows that when targeting two layers of double-buffering (one layer for disk $\leftrightarrow$ main memory transfers and another layer for main memory $\leftrightarrow$ GPU transfers), two buffers on each layer are not sufficient anymore. The idea here is to have two buffers on the GPU and three buffers on the CPU.

The double-triple buffering can be illustrated from two perspectives: the tasks executed and the buffers involved. The former is presented in Fig. 4; we refer the reader to [7] for a thorough description. Here we only discuss the technique in terms of buffers.

In this single-GPU scenario, the size of the blocks X_{R_b} used in the GPU's computation is equal to that on the CPU. When using multiple GPUs, this will not be the case anymore, as the CPU loads one large block and distributes portions of it to the GPUs.

The GPU's buffers are used in the same way as the CPU's buffers in the simple CPU-only algorithm: While one buffer α is used for the computation, the

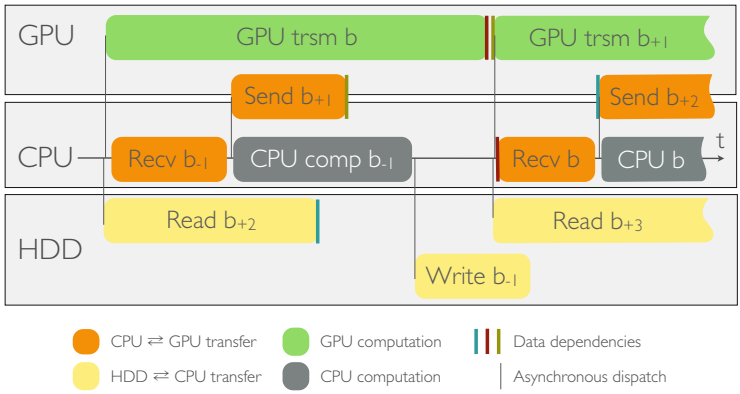


Fig. 4. A task-perspective of the algorithm. Sizes are unrelated to runtime.

data is transferred to and from the other buffer β. But at the CPU's level (i.e. in RAM), three buffers are now necessary. For the sake of simplicity, we avoid the explanation of the initial and final iterations and start with iteration b.

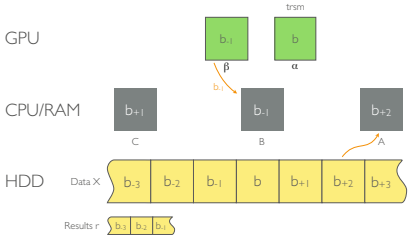
With reference to Fig. 5a, assume that the $(b–1)$-th, b-th and $(b+1)$-th blocks already reside in the GPU buffers β, α, and in the CPU buffer C, respectively. The block $b-1$ (i.e. buffer β) contains the solution of the `trsm` of block $b-1$. At this point, the algorithm proceeds by *dispatching* both the read of the second-next block $b+2$ from disk into buffer A and the computation of the `trsm` on the GPU on buffer α, and by receiving the result from buffer β into buffer B. The first two operations are *dispatched*, i.e. they are executed asynchronously by the memory system and the GPU, while the last one is executed synchronously because these results are needed immediately in the following step.

As soon as the synchronous transfer $\beta \to B$ completes, the transfer of the next block $b+1$ from CPU buffer C to GPU buffer β is *dispatched*, and the S-loop is executed on the CPU for the previous block $b-1$ in buffer B on the CPU (see Fig. 5b).

As soon as the CPU is done computing the S-loop, its results are written to disk (Fig. 5c). Finally, once all transfers are done, buffers are rotated (through pointer or index rotations, not copies) according to Fig. 5d, and the loop continues with $b \leftarrow b+1$.

3.2 Using Multiple GPUs

This multi-buffering technique achieves sustained performance on one GPU. Since boards with many GPUs are becoming more and more common in high-performance computing, we explain here how our algorithm is adapted to take advantage of all the available parallelism. The idea is to increase the size of the X_{R_b} blocks by a factor as big as the number of available GPUs, and then split the `trsm` among these GPUs. As long as solving a `trsm` on the GPU takes longer than loading a large enough block X_{R_b} from HDD to CPU, this parallelization



(a) Retrieve the previous result $b-1$ from GPU, and the second-next block $b+2$ of data from disk.

(b) Send the next block $b+1$ from RAM to the GPU, execute the S-loop on $b-1$ on the CPU.

(c) Write the results $b-1$ to disk.

(d) Switch buffers at both levels for the next iteration.

Fig. 5. The multi-buffering algorithm as seen from a buffer perspective

strategy holds up for any number of GPUs. Since in our systems loading the data from HDD was an order of magnitude faster than the computation of the `trsm`, the algorithm scales up to more GPUs than were available. Listing 1.3 shows the final version of cuGWAS.[5]

Listing 1.3. cuGWAS. The black bullet is a placeholder for "all GPUs".

```
1  L    ← potrf M                          (LL^T = M)
2  cublas_send L → L_gpu•
3  Xl   ← trsm L, Xl                       (X̃_L = L^{-1} X_L)
4  y    ← trsv L, y                        (ỹ = L^{-1} y)
5  rt   ← gemv Xl, y                       (r̃_T = X̃_L^T ỹ)
6  Stl  ← syrk Xl                          (S_TL = X̃_L^T X̃_L)
7  gpubs ← blocksize/ngpus
8  for b in -1..blockcount+1:
9      cu_trsm_wait α•                       (if b in 1..blockcount)
10     cu_send_wait C• → β•                  (if b in 2..blockcount+1)
11     α• ← cu_trsm_async L_gpu•, α•         (if b in 1..blockcount) ↙
                                          ↳ (X̃_{R_b} = L^{-1} X_{R_b})
12     aio_read Xr[b+2] → A                  (if b in -1..blockcount-2)
```

[5] The conditions for the first and last pair of iterations are provided in parentheses on the right.

```
13      for gpu in 0..ngpus-1:          (if b in 2..blockcount+1)
14          cu_recv B[gpu*gpubs..(gpu+1)*gpubs] ← β_gpu
15      aio_wait Xr[b+1] → C            (if b in 0..blockcount-1)
16      for gpu in 0..ngpus-1:          (if b in 0..blockcount-1)
17          cu_send_async C[gpu*gpubs..(gpu+1)*gpubs] → β_gpu
18      for Xri in B:                   (if b in 2..blockcount+1)
19          Sbl ← gemm Xri, Xl          (S_BL_i = X̃_R_i^T X̃_L)
20          Sbr ← syrk Xri              (S_BR_i = X̃_R_i^T X̃_R_i)
21          rb  ← gemv Xri, y           (r̃_B_i = X̃_R_i^T ỹ)
22          r   ← posv S, r             (r_i = S_i^-1 r̃_i)
23      aio_wait r[b-2]                 (if b in 1..blockcount+1)
24      aio_write r[b-1]                (if b in 1..blockcount+1)
25      swap_buffers
```

4 Results

In order to show the speedups obtained with a single GPU, we compare the hybrid CPU-GPU algorithm presented in Listing 1.3 using one GPU with the CPU-only OOC-HP-GWAS. Then, to determine the scalability of cuGWAS, we compare its runtimes when leveraging 1, 2, 3 and 4 GPUs.

In all of the timings, the time to initialize the GPU and the preprocessing (Lines 1–7 in Listing 1.3), both in the order of seconds, have not been measured. The GPU usually takes 5 s to fully initialize, and the preprocessing takes a few seconds too, but depends only on n and p. This omission is thus irrelevant for computations that run for hours.

4.1 Single-GPU Results

The experiments with a single-GPU were performed on the *Quadro* cluster at the RWTH Aachen University; the cluster is equipped with two nVidia Quadro 6000 GPUs and two Intel Xeon X5650 CPUs per node. The GPUs, which are powered by Fermi chips, have 6 GB of RAM and a theoretical double-precision computational power of 515 GFlops each. In total, the cluster has a GPU peak of 1.03 TFlops. The CPUs, which have six cores each, amount to a total of 128 GFlops and are supported by 24 GB of RAM. The cost of the combined GPUs is estimated to about $10 000 while the combined CPUs cost around $2000.

Figure 6a shows the runtime of OOC-HP-GWAS along with that of cuGWAS, using one GPU. Thanks to our transfer-overlapping strategy, we can leverage the GPU's performance and achieve a 2.6x speedup over a highly-optimized CPU-only implementation. cuBLAS' `trsm` implementation attains about 60 % of the GPU's peak performance, i.e. about 309 GFlops [8]. The peak performance of the CPU in this system amounts to 128 GFlops; if the whole computation were performed on the GPU at `trsm`'s rate, the largest speedup possible would be 2.4. We achieve 2.6 because the computation is pipelined: the S-loop is executed

on the CPU, in perfect overlap with the GPU. This means that the performance of cuGWAS is perfectly in line with the theoretical peak.

In addition, the figure indicates that the algorithm (1) has linear runtime in m and (2) allows us to cope with an arbitrary m. The red vertical line in the figure marks the largest value of m for which two blocks of X_R fit into the GPU memory for $n = 10\,000$. Without the presented multi-buffering technique, it would not be possible to compute GWAS with more than $m = 22\,500$ SNPs, while cuGWAS allows the solution of GWAS with any given amount of SNPs.

4.2 Scalability with Multiple GPUs

To experiment with multiple GPUs, we used the *Tesla* cluster at the Universitat Jaume I in Spain, since it is equipped with an nVidia Tesla S2050 which contains four Fermi chips (same model as the *Quadro* system), for a combined GPU compute power of 2.06 TFlops, but with only 3 GB of RAM each. The host CPU is an Intel Xeon E5440 delivering approximately 90 GFlops.

In order to evaluate the scalability of cuGWAS, we solved a GWAS with $p = 4$, $n = 10\,000$, and $m = 100\,000$ on the *Tesla* cluster, varying the number of GPUs. As it can be seen in Fig. 6b, the scalability of the algorithm with respect to the number of GPUs is almost ideal: Doubling the amount of GPUs reduces the runtime by a factor of 1.9.

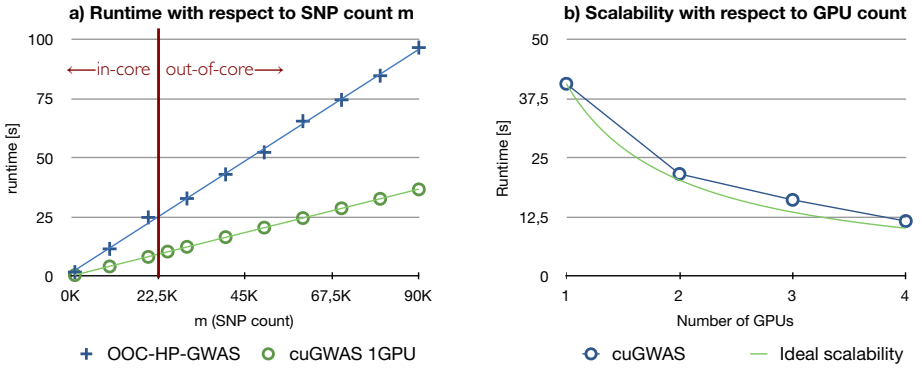


Fig. 6. The runtime of our cuGWAS algorithm a) using one GPU compared to OOC-HP-GWAS (CPU), b) using a varying amount of GPUs

5 Conclusion and Future Work

We have presented a strategy which makes it possible to sustain peak performance on a GPU not only when the data is too big for the GPU's memory, but also for main memory. In addition, we have shown how well this strategy scales to multiple GPUs.

As described by the developers of ProbABEL, the solution of a problem of the size described in Section 1.4 by the GWFGLS algorithm took 4 hours.

In contrast, with cuGWAS we solved the same problem in 2.88 s. Even accounting for about 6 seconds for the initialization and Moore's Law (doubling the runtime as ProbABEL's timings are from 2010), the difference is dramatic. We believe that the contribution of cuGWAS is an important step towards making GWAS practical.

Software. The code implementing the strategy explained in this paper is freely available at `http://github.com/lucasb-eyer/cuGWAS` and `http://lucas-b.eyer.be`.

Acknowledgements. Financial support from the Deutsche Forschungsgemeinschaft (German Research Association) through grant GSC 111 is gratefully acknowledged. The authors thank Diego Fabregat-Traver for providing us with the source-code of OOC-HP-GWAS, the Center for Computing and Communication at RWTH Aachen for the resources, Enrique S. Quintana-Ortí for granting us access to the *Tesla* system as well as Yurii S. Aulchenko for introducing us to the computational challenges of GWAS.

References

1. Genome-Wide Association Studies, `http://www.genome.gov/20019523`
2. Fabregat-Traver, D., Bientinesi, P.: Computing Petaflops over Terabytes of Data: The Case of Genome-Wide Association Studies (2012)
3. Catalog of Genome-Wide Association Studies, `http://www.genome.gov/gwastudies`
4. Fabregat-Traver, D., Aulchenko, Y.S., Bientinesi, P.: Solving Sequences of Generalized Least-Squares Problems on Multi-threaded Architectures (2012)
5. `http://www.ncbi.nlm.nih.gov/mailman/pipermail/dbsnp-announce/2012q2/000123.html`
6. Aulchenko, Y.S., Struchalin, M.V., Van Duijn, C.M.: ProbABEL package for genome-wide association analysis of imputed data. BMC Bioinformatics 11, 134 (2010)
7. Beyer, L.: Exploiting Graphics Accelerators for Computational Biology
8. Volkov, V., Demmel, J.W.: Benchmarking GPUs to tune dense linear algebra
9. Quintana-Ortí, G., Igual, F.D., Marqués, M., Quintana-Ortí, E.S., Van de Geijn, R.A.: A run-time system for programming out-of-core matrix algorithms-by-tiles on multithreaded architectures.
10. Yung, L.S., Yang, C., Wan, X., Yu, W.: GBOOST: a GPU-based tool for detecting gene–gene interactions in genome–wide case control studies
11. Lee, S., Kwon, M.-S., Huh, I.-S., Park, T.: CUDA-LR: CUDA-accelerated Logistic Regression Analysis Tool for Gene-Gene Interaction for Genome-Wide Association Study