RWTH AACHEN UNIVERSITY

Computational Engineering Sciences (CES)

Exploiting Graphics Accelerators for Computational Biology

Diploma Thesis

Author:
Lucas Beyer

Supervisor:
Prof. Paolo Bientinesi

July 23, 2012

Contents

Chapter 1

Acknowledgements

First of all, I would like to thank prof. Paolo Bientinesi for giving me the opportunity to work on an interesting problem involving GPUs, biology, high-performance and visualization. I also thank him for the trust he put in me by giving me the chance to present my work at an international conference at such an early stage of my academic education. Not only did I have a lot of fun, but I also learned quite a bit.

Next, I am thankful to Diego Fabregat-Traver for permanently letting me bug him with all my questions, for his openness and for all the work he has already done on this project. I also want to thank the rest of the AICES-HPAC group for the interesting discussions and especially for all the helpful feedback on my mock-talk. The access to the GPU cluster at Universitat Jaume I in Spain which I have been granted by Enrique S. Quintana-Ortì, as well as his comments have been very helpful.

Finally, I am grateful to my girlfriend Supinya for her great amount of patience and support (and food :p) as well as to my parents for all the support they gave me and to my brother for all the played games to relax me.

Chapter 2

Introduction

In this chapter we first give a succint introduction to genetics, explaining what Genome-Wide Association Studies (GWAS) are and then we describe the mathematical formulation and the computational aspects of GWAS. We conclude by presenting the objectives of this thesis along with an overview of related works.

In short, the goal of a GWAS is to find an association between genetic variants and a specific trait such as a disease. Because there is a tremendous amount of such genetic variants, the computations involved in GWAS take a long time, ranging from days to weeks or even months. In this thesis, we take the currently fastest available implementation and further speed it up by exploiting the computate offered by modern graphics accelerators, thus reducing the computation time to only hours.

2.1 Introduction to genetics

Cells are the building blocks of life; humans are made out of about 10 trillion (10^{13}) cells. A single cell is a microscopic (1–100 µm[1]) living organism which can grow, reproduce and synthesize proteins. In a human, many different types of proteins coexist. Each type is specialized in one function –like copying a cell, repairing damage, supporting muscle contraction, breaking down proteins contained in food– and can only carry out that specific function. Thus, the overall functionality of a cell is determined by the kind of proteins it produces, which, in turn is dictated by a part of the cell's DNA[2].

[1] A micrometre µm corresponds to a thousandth of a millimetre mm

[2] The remaining part of the DNA is called *noncoding DNA*; colloquially, this is referred to as *junk DNA*, because it has no or unknown functionality.

The segments of the DNA which 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 internal features of the organism –like blood type or resistances to diseases–. The fact that children look similar to their parents is due to the genes they inherit. During this process of inheritance, the parents' genes are copied and combined, but some of them incur random modifications, known as *mutations*. Mutations are the reason for differences between children and their parents, including undesirable ones such as diseases [22]. The analysis of these differences in the genes helps understanding, identifying and sometimes even preventing or curing such diseases.

The hereditary information of a species consists of all the genes in the DNA, and is called *genome*. (Exceptions are viruses, for many of which the genome is encoded in the RNA. [20]) To help us grasp the magnitude of the human genome, we can use an analogy with instructions written in a book: the genome is the book, which contains 23 chapters (the chromosomes); each chapter contains between 48 and 250 million letters out of an alphabet consisting of only the *nucleotides* A, C, G and T, and does not contain spaces; this whole book fits inside a cell nucleus of an average size of 6 µm and is present in almost every single cell out of the 10 trillion cells in the human body. A gene is a functional group of nucleotides. Determining the order of the nucleotides (letters) in a genome of a species (book) is a procedure called *sequencing the genome* of that species; the result is a genome sequence. The *human genome project* has successfully sequenced the human genome in 2004 [6], which makes it now possible to conduct *genome-wide* association studies.

Even though the genome sequence of every individual is different, the biggest part of it (99.9% for humans) stays the same within one species. 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"). As an example, if a fragment of the DNA of one individual was the gene AAGC**C**TA while the gene was AAGC**T**TA for another individual, there would be a SNP and the two *alleles*[3] would be **C** and **T**. SNPs are of particular interest because if most people with a specific trait (e.g. being red-haired) all have the same alleles for some SNPs (while most people who don't have this trait have other alleles for these SNPs), it is very likely that these SNPs play a role for said trait.

[3] An *allele* is one of two or more possible forms that a nucleotide (or gene, or group of genes) can have. When a nucleotide can either be `A` or `G`, those are the alleles; when a gene can either be `ACTA` or `ATCA`, those are the alleles.

2.2 Genome-Wide Association Studies

The predisposition and the way an individual develops and responds to a disease or a specific treatment is often affected by some SNPs or combinations thereof. This is why the study of SNPs and their correlation with traits is very important [5].

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* don't have this trait. The SNPs of the individuals in these groups are compared, if one allele (variant) of a SNP is more frequent in the case group than in the control group, the SNP is said to be *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 [14].

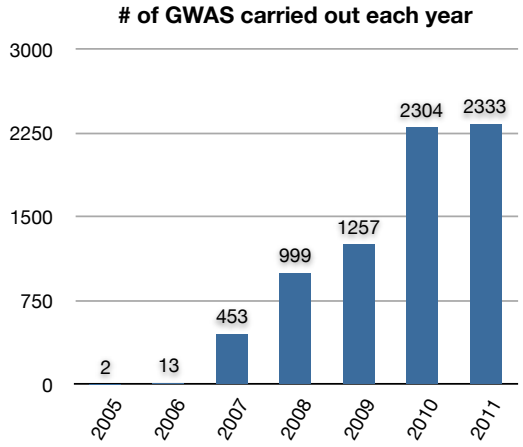


Figure 2.1: Amount of genome-wide association studies published each year.

Using the catalog of all published GWAS [12] recently compiled by the NIH's Office of Population Genomics, we gathered some statistics about GWAS. Fig 2.1 shows that the amount of published GWAS has increased tremendously in recent years, ending with more than six new studies per day in the last two years.

It is also interesting to look at how the scope of these studies has evolved over the past years. The evolution of the number of SNPs investigated in the studies is summarized in Fig. 2.2. The left panel (part a) shows the median[4] SNP-count of

[4]When all datapoints are ordered, the one in the middle is the median. The median has a similar interpretation as the average although it is less sensitive to outliers. The median can also be called the 50-percentile.

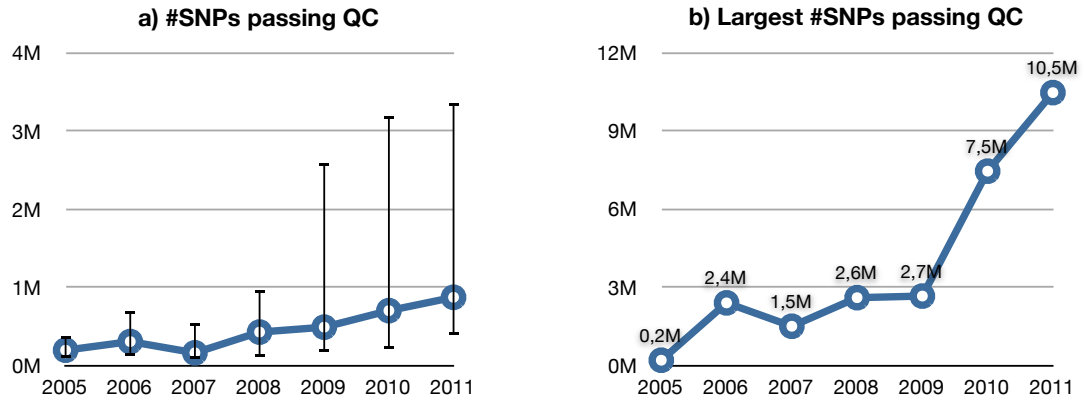


Figure 2.2: a) The median, first and second quartile and b) the largest SNP-count used for the studies each year.

each year's studies along with error-bars for the first and second quartiles[5]. The right panel (part b) displays the study with the largest SNP-count of each year. One can observe that while GWA studies started out relatively small, since 2009 the amount of analyzed SNPs is growing tremendously. Preliminary data for 2012 suggests that this trend keeps going with the largest study having analyzed about 16 million SNPs [11], which is 1.5 times the biggest one in any of the previous years. This data, as well as discussions with biologists, suggest that there is a need for algorithms and software that can compute a GWAS with even more SNPs, and faster than currently possible.

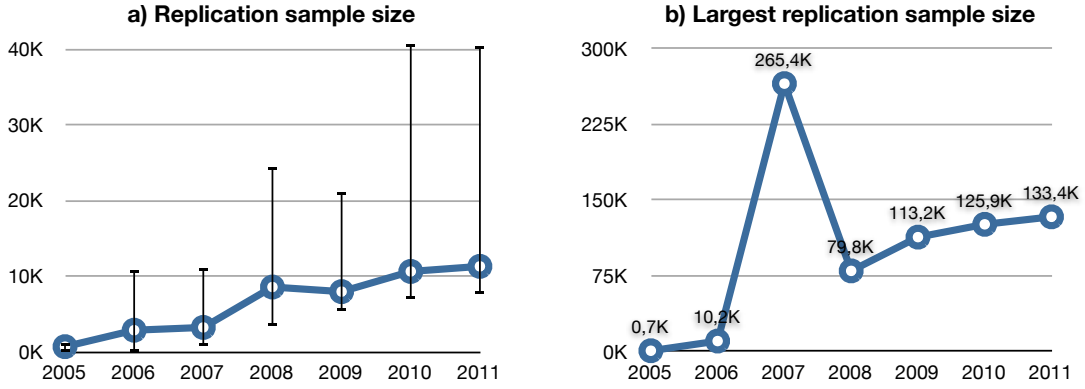


Figure 2.3: a) The median, first and second quartile and b) the largest sample size used for the replication of the studies each year.

[5]The first and second quartile are also called 25-percentile and 75-percentile respectively. They are defined analogously to the median, only that the 25%th and the 75%th ordered datapoint is picked instead of the 50%th. They give a feeling of the spread of the data when coupled with the median, similarly to the use of the standard deviation coupled with the average. This means that 50% of the datapoints lie within the black error-bars.

Besides the number of SNPs, another parameter relevant to the implementation of an algorithm is the *sample size*. The sample size is the total number of individuals of both the case and the control group. Almost always, a study is replicated (repeated) a second time in order to increase the confidence in the results. In almost all of the published GWAS, the sample size used during the replication is larger than the sample size of the initial study. This is why we will look at the replication sample size unless we specify otherwise. What can be seen in Fig. 2.3.a 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. The biggest general growth happened between 2007 and 2008, and since then the sample size is growing slowly. Even looking at the biggest sample size of each year (Fig 2.3.b), it is clear that besides one outlier in 2007[6], the growth of the sample size is negligible when compared to the growth of the SNP count. This is supported by the fact that a sample size of ten thousand individuals is usally more than enough to achieve statistically significant results.

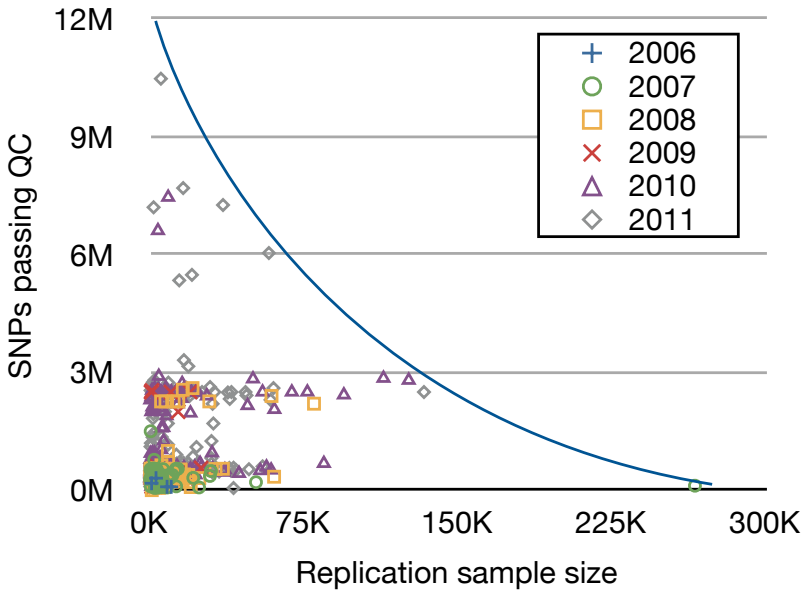


Figure 2.4: Every published GWAS's SNP and sample count.

Our intuition was that the studies analyzing a very large amount of SNPs are likely to have a small sample size and vice-versa. We have investigated this in Fig. 2.4, where every single published GWAS represents one datapoint whose horizontal and vertical coordinates are determined by its sample size and SNP count respectively. It is immediately noticeable that the whole upper right area is empty; this means there has never been a GWAS with both a big sample size

[6]The second largest sample size for that year is 51 535 individuals, which fits the curve on the graph nicely.

and SNP count, confirming our intuition.

2.3 The mathematics of GWAS

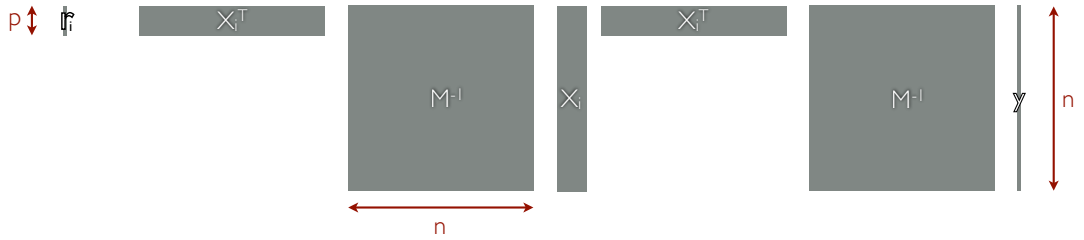


Figure 2.5: The dimensions of a single instance of Eq. (2.1).

Let n and m be the sample size and the number of SNPs considered, respectively. The GWAS can be expressed as a variance component model [9] whose solution r_i can be formulated as

$$r_i = (X_i^T M^{-1} X_i)^{-1} X_i^T M^{-1} y, \quad i = 1..m. \tag{2.1}$$

This equation is used to compute in r_i the relations between variations in the *phenotype*[7] y and variations in the *genotype* encoded in X_i. Fig. 2.5 captures the dimensions of the objects involved in one such equation. The height of the matrices X_i and M and of the vector y corresponds to the number of samples n, thus each line in the *design-matrix* $X_i \in \mathbb{R}^{n \times p}$ corresponds to an 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[8]. $M \in \mathbb{R}^{n \times n}$ 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 are partitioned as $(X_L|X_{R_i})$ where X_L contains fixed covariates such as age and sex and thus stays the same for any i, while X_{R_i} is a single column vector containing the genotypes of the ith SNP.

Even though Eq. (2.1) has to be computed for every single SNP, only the design-matrix X_i changes while M and y stay the same. Fig 2.6 reflects this fact by showing the variables proportionally sized for ten thousand individuals ($n = 10\,000$) and half a million SNPs ($m = 500\,000$), a problem which, as discussed in the previous section, can still be considered small.

[7] 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".

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

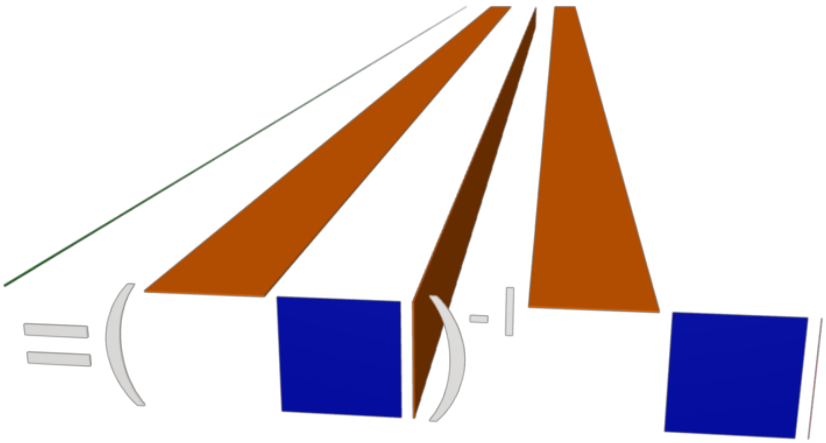


Figure 2.6: A proportionally correct depiction of the full Eq. (2.1) for $n = 10\,000$ and $m = 500\,000$.

2.3.1 The amount of data involved

At this point, we can already analyze the storage size requirements due to the data involved in GWAS. Biologists told us that a typical value for p is 20 and that $m = 10\,000$ is considered a big study. Our analysis in Section 2.2 supports this fact. We also know that, ideally, biologists would like to use all known SNPs in a study. As of June 2012, the SNP database *dbSNP* lists 187 852 828 known SNPs for humans [4], so we will consider $m = 190\,000\,000$. With these numbers, assuming that all data is stored as C's `double` type[9], we obtain the storage sizes reported in Table 2.1. While y and M fit into main memory (RAM) and even GPU memory, the output r comes close to the limit of current high-end systems' main memory and is already too big to fit on a GPU's 6 GB of memory. X is instead simply too big to fit into the memory of any system in the foreseeable future and will have to be streamed from disk.

Variable	Dimension	Approx. storage size
y	n	80 MB
M	$n \times n$	800 MB
r	$p \times m$	30 GB
X	$n \times p \times m$	300 TB

Table 2.1: Storage size necessary to hold all data.

[9]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 in Table 2.1 should be halved.

2.4 Related work

In the field of bioinformatics, the R project [19] is the most widely used software package for computations. GenABEL is a frequently used R library for genome-wide association analysis released in 2007 [2]. It is not comparable to this thesis' problem because it does not take M into account. Three years later, the authors of GenABEL released a new library called ProbABEL which, amongst other improvements, takes the relationships M between individuals into account. In their introductory paper [3], the authors report a runtime of almost 4 hours for a problem size of $p = 4$, $n = 1500$ and $m = 220\,833$ and estimate the runtime with $m = 2\,500\,000$ to be roughly 43 hours[10], which amounts to almost two days. While 2.5 mio SNPs can be consideered a big dataset when compared to the median SNP count in 2009 and 2010, a population size of only 1500 individuals is clearly much smaller than the average study in these years. 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, one can imagine computation times of weeks or even months. We will present further timings which support this claim in Section 3.3.

2.5 Fundamental HPC libraries and algorithms

In high-performance computing, just like in many other fields of programming, it is important to rely upon the efforts of others. Reimplementing everything from scratch, down to the lowest level is usually a bad idea, especially when existing solutions are known to perform very well. In this section, we present the HPC libraries relevant to this thesis.

2.5.1 Basic Linear Algebra Subprograms (BLAS)

The BLAS define a set of dense linear algebra *building block* operations, such as the matrix-vector and matrix-matrix products. The reference implementation, written in Fortran77, is far from attaining high performance. Various hardware vendors implement the BLAS and optimize them for their hardware platform. For example, Intel's version is part of the Intel Math Kernel Library (MKL) and AMD's implementation is part of the AMD Core Math Library (ACML). Good free open-source implementations of the BLAS are available too ([10] and [23]).

[10] We only consider what the authors called the *linear model* with the `-mmscore` option as this solves the exact problem tackled by this diploma thesis.

The BLAS are organized in three levels, corresponding to three classes of operations:

- level 1 BLAS consist of vector-vector subprograms,
- level 2 BLAS consist of matrix-vector subprograms, and
- level 3 BLAS consist of matrix-matrix subprograms.

Whenever designing an algorithm, it is best to aim at using as many level 3 BLAS as possible, as these are usually the only ones coming close to the hardware's peak performance.

In the following, we describe the subprograms which are used in this thesis; let α be a scalar, x and y vectors, A, B, C and X general matrices, and T a triangular matrix.

- `DOT` (BLAS-1): computes the dot-product $\alpha \leftarrow x^T y$;
- `GEMV` (BLAS-2): adds a matrix-vector product to a vector $y \leftarrow Ax + y$;
- `TRSV` (BLAS-2): solves a triangular system $Tx = y$;
- `SYRK` (BLAS-3): adds a square$d$ matrix to a matrix $C \leftarrow A^T A + C$;
- `GEMM` (BLAS-3): adds a matrix-matrix product to a matrix $C \leftarrow AB + C$;
- `TRSM` (BLAS-3): solves a triangular system with multiple right-hand sides $TX = B$.

2.5.2 Linear Algebra Package (LAPACK)

While LAPACK started as a Fortran77 library, nowadays highly optimized vendor implementations are available in the previously introduced MKL and ACML libraries. The difference between BLAS and the LAPACK is that the latter consists of higher level operations and relies upon the former internally. With more than 200 distinct routines for solving linear least squares, eigenvalue and singular value problems, matrix factorizations and many more operations, the scope of the LAPACK is also much broader than the BLAS.

We now introduce the subprograms which are used in this thesis; let x and y be vectors, and A and L be a s.p.d.[11] and a triangular matrix, respectively.

[11]The acronym s.p.d. stands for symmetric positive definite. A matrix is s.p.d. when it is symmetric and all its eigenvalues are positive.

- `POSV`: solves a s.p.d.[12] system of linear equations $Ax = y$;
- `POTRF`: computes the Cholesky factorization of a s.p.d. matrix $LL^T = A$.

2.6 Goals of the thesis

Having introduced the details of the problem at hand along with typical and desired use cases, we can fix the goals of this thesis[13].

(a) Accelerate the computation of GWAS by offloading the most expensive part of the computation onto the GPU, while *at the same time* computing all the other parts on the CPU.

(b) The GPU should be computing non-stop, it should never be waiting for data transfers to complete.

(c) Scale the algorithm to use multiple GPUs automatically.

(d) The algorithm should be efficient both on current and future hardware without the need to change a single line of code.

(e) The runtime should be at most linear in m: doubling the SNP count should at most double the computation time.

(f) Support for an arbitrarily large SNP count m.

(g) The sample size n should be limited only by hardware, with $n = 10\,000$ being the minimum on current hardware.

[12] A system of linear equations $Ax = b$ is s.p.d. when its system matrix A is s.p.d.
[13] Note that the order of the goals has no particular meaning.

Chapter 3

State of the art

In this chapter we describe the fastest algorithm [8] currently available for computing the solution r_i of a GWAS: HP-GWAS[1]. A good understanding of this algorithm is necessary since this thesis builds upon and extends HP-GWAS.

3.1 The HP-GWAS algorithm

The HP-GWAS algorithm achieves much higher performance than the algorithm used in ProbABEL by making efficient use of domain specific knowledge. The problem has two properties which offer optimizations opportunities. The first one is the symmetry and the positive definiteness of the matrix M, which can be exploited by a Cholesky factorization $LL^T = M$. Since M does not depend on i, we can compute its Cholesky decomposition once in a preprocessing step and keep the solution in memory. Inserting this decomposition into Eq. (2.1), we obtain

$$r_i = (X_i^T L^{-T} L^{-1} X_i)^{-1} X_i^T L^{-T} L^{-1} y \qquad \text{for } i = 1..m, \tag{3.1}$$

which can be rearranged as

$$r_i = (\underbrace{(L^{-1}X_i)}_{\tilde{X}_i}^T \underbrace{L^{-1}X_i}_{\tilde{X}_i})^{-1} \underbrace{(L^{-1}X_i)}_{\tilde{X}_i}^T \underbrace{L^{-1}y}_{\tilde{y}} \quad \text{for } i = 1..m, \tag{3.2}$$

[1]HP-GWAS is the name of the algorithm in the referenced paper. The same algorithm has been renamed to CLAK-CHOL by the author in later publications, these names are used interchangeably.

that is

$$r_i = (\underbrace{\tilde{X}_i^T \tilde{X}_i}_{S_i})^{-1} \underbrace{\tilde{X}_i^T \tilde{y}}_{\tilde{r}_i} \qquad \text{for } i = 1..m. \tag{3.3}$$

Listing 3.1: Solution of a sequence of GLS

```
1 L ← potrf(M)          (LL^T = M)
2 y ← trsv(L, y)        (ỹ = L^-1 y)
3 for i in 1..m:
4     Xi ← trsm(L, Xi)  (X̃_i = L^-1 X_i)
5     Si ← syrk(Xi)     (S_i = X̃_i^T X̃_i)
6     ri ← gemv(Xi, y)  (r̃_i = X̃_i^T ỹ)
7     ri ← posv(Si, ri) (r_i = S_i^-1 r̃_i)
```

This first optimization is expressed algorithmically in Listing 3.1. Although this algorithm already exploits one important property of Eq. (2.1), it is still far from optimal. The second problem-specific piece of knowledge we can exploit is the structure of $X = (X_L|X_R)$. We have in fact already noted that X_L stays the same for any i: $X_i = (X_L|X_{R_i})$. This partitioning can be inserted into the lines 4-6 in Listing 3.1. Line 4 becomes $(\tilde{X}_L|\tilde{X}_{R_i}) = L^{-1}(X_L|X_{R_i})$, line 5 becomes $\begin{pmatrix} S_{TL} & \bullet \\ S_{BL_i} & S_{BR_i} \end{pmatrix} = \begin{pmatrix} \tilde{X}_L^T\tilde{X}_L & \bullet \\ \tilde{X}_{R_i}^T\tilde{X}_L & \tilde{X}_{R_i}^T\tilde{X}_{R_i} \end{pmatrix}$ and line 6 becomes $\begin{pmatrix} \tilde{r}_T \\ \tilde{r}_{B_i} \end{pmatrix} = \begin{pmatrix} \tilde{X}_L^T \\ \tilde{X}_{R_i}^T \end{pmatrix}\tilde{y}$. Several of these parts do not depend on i and can thus be extracted and computed only once during the preprocessing step. As X_{R_i} is a single column-vector, this reduces the operations in lines 4-6 to vector operations. The resulting algorithm is shown in Listing 3.2.

Listing 3.2: Solution of the GWAS-specific sequence of GLS

```
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_Ri = L^-1 X_Ri)
8      Sbl ← dot Xri, Xl  (S_BLi = X̃_Ri^T X̃_L)
9      Sbr ← syrk Xri     (X_BRi = X̃_Ri^T X̃_Ri)
10     rb  ← dot Xri, y   (r̃_Bi = X̃_Ri^T ỹ)
11     r   ← posv S, r    (r_i = S_i^-1 r̃_i)
```

Further improvements are still possible. For instance, line 7 performs a `trsv`, a BLAS-2 operation known to have much lower efficiency than the BLAS-3 `trsm` operation. By taking multiple vectors X_{R_i} and packing them into a matrix X_{R_b}

as depicted in Fig. 3.1, we can replace multiple `trsv`s by a single more efficient `trsm`. This concept leads us to a final first version of the algorithm shown in Listing 3.3.

Figure 3.1: Packing vectors into a matrix in order to replace multiple `trsv`s by a single more efficient `trsm`.

Listing 3.3: Optimized solution of the GWAS-specific sequence of GLS

```
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 b in 1..blockcount:
7      Xrb ← trsm L, Xrb        (X̃_b = L^{-1} X_b)
8      for Xri in Xr[b]:
9          Sbl ← gemm Xri, Xl   (S_BL_i = X̃_R_i^T X̃_L)
10         Sbr ← syrk Xri       (X_BR_i = X̃_R_i^T X̃_R_i)
11         rb  ← gemv Xri, y    (r̃_B_i = X̃_R_i^T ỹ)
12         r   ← posv S, r      (r_i = S_i^{-1} r̃_i)
```

3.2 Handling huge datasets

The problem with the algorithm presented so far is that it is *in-core*, which means that it cannot deal with datasets bigger than the available memory. Algorithms which can handle data too large for memory are called *out-of-core*. One way to turn our algorithm in Listing 3.3 into an out-of-core algorith is to make use of a technique called double-buffering: while the CPU is busy computing the block b of X_R in a primary buffer, we can already load the next block $b+1$ into a secondary buffer. When the CPU is done computing the block b, and if the disk is fast enough, the next block will already be present in the secondary buffer. The CPU can then immediately start computing the block $b+1$ without having to wait for any data to arrive. The final algorithm is shown in Listing 3.4.

Listing 3.4: An out-of-core version of the algorithm from Listing 3.3

```
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_b = L^{-1} X_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          (X_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]
```

3.3 Performance

We have run the algorithm described in Listing 3.4 for a GWAS with $p = 4$, population size of $n = 10\,000$, and varying amount of SNPs m; we then compared the runtime to other currently available algorithms for computing GWAS. GWFGLS is the algorithm implemented in ProbABEL, which is one of the most widely used ones in practice. FLMM is a recently published algorithm [13] which focuses on a large population size rather than a large SNP count, and CLAK-Chol is the algorithm presented in Listing 3.4.

Fig. 3.2 shows the runtime for the various algorithms. Both axes have a logarithmic scale and the SNP count is given in millions. While FLMM already does a good job at reducing the computational time when compared to both GWFGLS and EMMAX, the CLAK-Chol algorithm does an even better job. Instead of taking months, the computation of a large GWAS can be executed in only several hours. The speedup of CLAK-Chol over FLMM, GWFGLS and EMMAX is 6.3, 56.8 and 112 respectively.

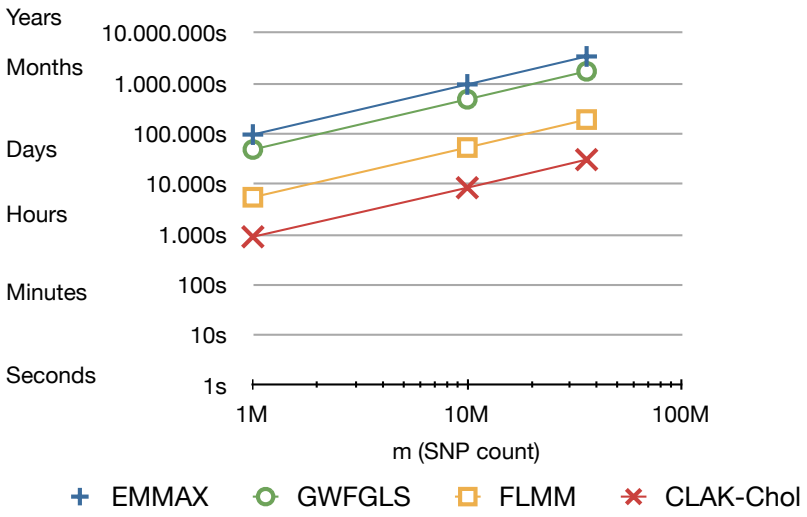


Figure 3.2: Comparison of the runtime of various algorithms with varying SNP counts. Notice the logarithmic scales.

Chapter 4

Graphics Processing Units (GPUs)

In this chapter we give an introduction to GPUs and describe why they are relevant to high-performance computing. A reader already familiar with how GPUs work and for what they can be used should jump directly to the next chapter (pg. 29).

4.1 History of GPUs

In the 80s, the first personal computers appeared, namely the IBM PC and the Commodore Amiga. In these computers, the *Graphics* or *Video Card* consisted of not much more than a Random Access Memory (RAM) and a RAM Digital-to-Analog Converter (RAMDAC). The RAM memory held a representation of the pixels which were shown on the screen. At that time, the CPU used to perform all the computations which are necessary for drawing and moving images on the screen, and then write the results into the graphics card's memory. From here, the memory was converted to a signal sent to the screen, resulting in an image being shown.

With the Professional Graphics Adapter (PGA), IBM was the first company to introduce a graphics card which incorporated its own processing unit, freeing the CPU from the graphic-related workload. As the name suggests, such a graphics card was not aimed at the personal computer market: the cost was around $5500. Due to the high price and the incompatibility with many programs and non-IBM systems, the PGA did not spread widely. Still, its separate processing unit set a landmark in the landscape of GPUs. At the end of the 80s and beginning of the 90s, the use of expensive graphics workstations for 3D graphics was very widespread in the professional CAD market. These workstations were mainly

built by SGI, the same company that also introduced the OpenGL Application Programming Interface (API) used for accessing both 2D and 3D functionality of the workstation's processing unit.

It is not before the mid-nineties that the 3D graphics cards broke into the commodity PCs thanks to the Voodoo card by *3dfx Interactive*, a company founded by former SGI employees. This card marked the end of expensive graphics workstations and the beginning of a new era, that of consumer-level 3D graphics cards. Despite its large success, the Voodoo suffered from one major drawback: it did not support the OpenGL API. Only a few years later, in the end of the nineties, both Nvidia and ATI introduced their consumer-level GPUs, the *RIVA TNT* and the *3D Rage* respectively. These graphics cards were all addressable with the OpenGL API and, coupled with tremendously successful 3D games such as *Quake*, they quickly eclipsed the Voodoo. 3dfx was later acquired by Nvidia.

In October 1999, Nvidia released the GeForce 256, which they marketed as "the world's first GPU, or graphics processing unit". They thereby coined the term GPU. It was indeed the first graphics accelerator which could compute both vertex transformation and lighting (T&L) in a single chip. This accelerator pioneered the generation of so-called *fixed-function pipeline* GPUs, of which the later released GeForce 2 and ATI's Radeon were part too. The fixed-function pipeline, shown in Fig. 4.1, is the sequence of operations the 3D polygons go through in order to be rendered as pixels on the screen. While this approach of a hardwired pipeline of algorithms leads to highly performant GPUs, it is not flexible enough to be used for any kind of computation unrelated to graphics.

Since GPUs' main application, games, requires the rendering of hundreds of thousands of polygons in only a fraction of a second (typically $\frac{1}{60}$ s) and the complexity of the scene grows with every new generation of games, GPUs need to deliver high computational performance in order to keep up with the game industry's demand. Already in the year 1999, the increase in performance of GPUs grew at a rate much faster than Moore's law.

It is not before 2001, marked by the release of Nvidia's GeForce 3, that the high-performance computation community became interested in GPUs. The GeForce 3 was the first chip to allow the programmer to run small custom programs in the vertex transform and pixel shading stages of the pipeline. These programs, called *shader programs*, would be run many times on different input data and at the same time in separate parts of the chip. For the first time, it was possible to make these powerful chips compute something completely unrelated to graphics. Nvidia's GeForce 8 GPUs in 2006 introduced the concept of *unified shaders*, meaning that a single type of processing unit was used to run any type of shader program, be it a vertex, fragment or the freshly introduced geometry shader. This profoundly changed the GPU architecture from very specialized processors into a collection

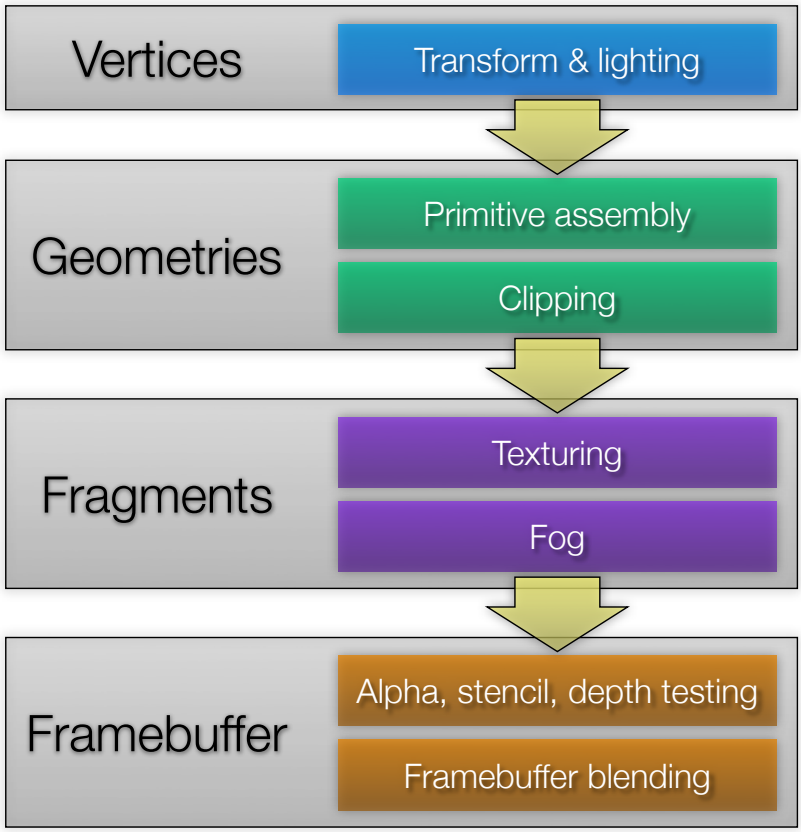


Figure 4.1: A typical representation of the graphics pipeline.

of more general-purpose processors called Streaming Multiprocessors[1] (SMs). At this point, it became clear even to the sceptics that GPUs can be utilized not only for games, but also for high-performance scientific computations, provided that the same operation has to be executed many times. In the following years, GPU manufacturers were able to lift more and more of the constraints of the shader programs and thus allow increasingly more complex programs to be run in parallel on the GPU.

Nowadays, there are mainly two programming APIs opposing each other in the field of high performance GPU computations: Nvidia's *CUDA* (Compute Unified Device Architecture) and OpenCL[2]. CUDA was made public for the first time early in 2007 and only works with Nvidia GPUs. The first OpenCL specification was published almost two years later as an effort of multiple vendors (Nvidia, AMD/ATI, Intel, ...) and users to come up with a consistent, cross-platform

[1]Note that this unified architecture allowed for the introduction of many new shader types and stages (e.g. the geometry and tesselation shaders) to the rendering pipeline.

[2]There are, in fact, a few more APIs such as the AMD Stream SDK but the current trend is to drop support for these libraries and invest into OpenCL.

API for high-performance parallel computations. The often quoted advantage of OpenCL is that one can write one single code and run it on any device, be it a GPU from Nvidia, an AMD/ATI GPU or even a CPU. While this is true, it is not practical for high performance because the code which performs best on one specific hardware architecture is bound to attain subobtimal preformance [7] on different architectures.

4.2 The architecture of a modern GPU

Understanding the architecture of a GPU is key for achieving high performance. This section describes Nvidia's Fermi GPU architecture [16] which is depicted in Fig. 4.2 and has been used throughough this thesis. While current CPUs have between two and eight very generic cores, a GPU is made of hundreds (512 for most Fermi cards) of specialized cores, represented by the small green squares in the figure. A Streaming Multiprocessor (SM) is a group of 32 of these cores along with four Special Function Units (SFUs) for computing transcendental functions, sixteen load/store units for accessing memory, 64KB of shared memory, two schedulers and a few more components. While these 16 SMs can run independently of eachother, the cores in one SM can not. Because a SM has two dispatch units, it can run two different programs in parallel on 16 cores each. These 16 cores execute instructions in lockstep in a Single-Instruction Multiple-Data (SIMD) fashion; they are said to run one *wrap*. Whenever a wrap has to wait for a data load or store, rather than to idle, the 16 cores will execute another wrap. Given enough wraps, this allows the Fermi GPU to hide expensive memory load instructions. A novelty of the Fermi architecture is the unified L2 cache, which enables much faster memory transfers for some access patterns. Nvidia has published a very detailed desctiption [18] of these memory access patterns and many more details one has to be aware of in order to achieve high performance.

4.3 Libraries for GPU computing

While it is relatively simple to write fast computational kernels[3] for a GPU, it is extremely tedious and time consuming to make it as fast as it can get.

Since the very beginning of CUDA, Nvidia has provided an implementation of the BLAS running on its GPUs called cuBLAS[15]. This implementation was notorious for being far from optimal –although implemented by Nvidia– until

[3]Although code which runs on the GPU was initially called a shader program, if the code is unrelated to the graphics pipeline it is called a *kernel*.

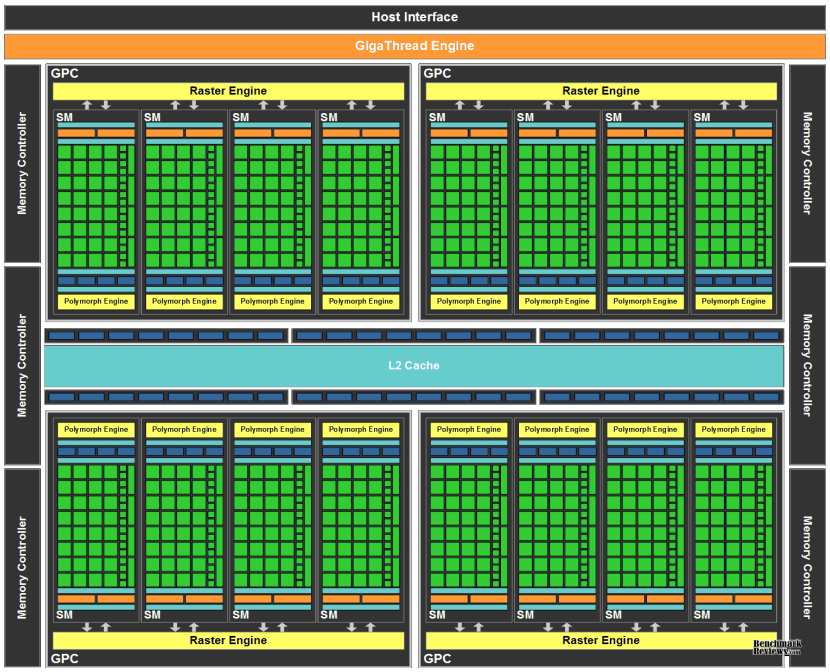


Figure 4.2: A diagram of Nvidia's Fermi GPU architecture.

V. Volkov thoroughly optimized the `gemm` and `trsm`[21] operations for Nvidia's G80 and Fermi architectures. Not only did he completely outperform Nvidia's own implementation by dismissing several of the guidelines given in the *CUDA C best practice guide*, but he also gives reasons for why his implementation is as fast as it can get [21], even though the `gemm` operation attains only 60% of the theoretical peak performance on Fermi architecture GPUs[4]. By now, his optimizations to those as well as other BLAS and LAPACK operations (including `potrf`) have been incorporated into cuBLAS, which is now regarded as a good high performance library.

Seeing that our algorithm for computing GWAS is expressed in terms of BLAS and LAPACK operations and cuBLAS achieves optimal performance, it would be a waste of time and effort to implement our own kernels and they would probably also be slower. Not only does using cuBLAS make our code attain optimal performance for the Fermi platform, but it also promises optimal performance for past and future Nvidia platform. For these reasons, we decided for using cuBLAS, as opposed to implementing our own kernels, thus already achieving goal (d).

[4] `gemm` performance on Nvidia's newest Kepler architecture is reported to attain up to 80% of the theoretical peak performance.

Chapter 5

Leveraging GPUs for GWAS

In this chapter, we describe how we use GPUs to accelerate genome-wide association studies, and which hurdles we had to overcome in order to achieve a highly efficient implementation.

5.1 Determining the current bottleneck

We start the discussion by inspecting the CPU implementation described in Chapter 3; we aim at identifying which part of the algorithm is the slowest; that will be the section we want to compute on the GPU.

Figure 5.1: Breakdown of the runtime of Listing 3.4.

Because of the complexity, the operation count, and personal experience with BLAS, one could guess that the `trsm` on line 10 in Listing 3.4 takes up most of the computation time. But the golden rule in code optimization is to never optimize without measuring first, even if one has a very strong guess as to which part of the code is the bottleneck. We thus measure the runtime of all operations of the current implementation of the HP-GWAS algorithm in order to determine the bottleneck. The breakdown of the time spent in every operation is shown in Fig. 5.1. As one can see, the initial guess was correct: the `trsm` is responsibe for most of the time, in fact, more time than all the other operations combined. As discussed in Section 4.3, a GPU high-performance implementation of this

operation is available through the cuBLAS library. We can offload this operation to the GPU in order to fully exploit it and thus accelerate the whole algorithm.

Listing 5.1: Moving the computation of the `trsm` to the GPU

```
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  aio_read Xr[1]
8  for b in 1..blockcount:
9      aio_read Xr[b+1]
10     aio_wait Xr[b]
11     cu_send Xr[b] → Xrb_gpu
12     Xrb_gpu ← cu_trsm L_gpu, Xrb_gpu    (X̃_b = L^{-1} X_b)
13     cu_recv Xr[b] ← Xrb_gpu
14     for Xri in Xr[b]:
15         Sbl ← gemm Xri, Xl              (S_BL_i = X̃_R_i^T X̃_L)
16         Sbr ← syrk Xri                  (X_BR_i = X̃_R_i^T X̃_R_i)
17         rb  ← gemv Xri, y               (r̃_B_i = X̃_R_i^T ỹ)
18         r   ← posv S, r                 (r_i = S_i^{-1} r̃_i)
19     aio_wait r[b-1]
20     aio_write r[b]
21 aio_wait r[blockcount]
```

5.1.1 Results

In order to compute the `trsm` on the GPU, the algorithm has to send the necessary data to the GPU first. While the L matrix can be sent once during the preprocessing step, unfortunately for every block b, the matrices X_{R_b} have to be sent to the GPU, and the resulting matrices $\tilde{X}_{R_b}$ have to be sent back to the CPU's main memory. Listing 5.1 shows the resulting algorithm, interleaved with the necessary memory movements. In this listing and in all of the following figures, the colors consistently encode the type and location of the operation: black (and gray in the figures) and green denote operations executed on the CPU and GPU, respectively, and orange and yellow denote CPU ↔ GPU and disk ↔ main memory transfers, respectively. A profiled breakdown of the runtime of the algorithm presented in Fig. 5.2 reflects the pattern seen in the code listing. The reason there is no yellow in the timings is because the disk ↔ main memory transfers are already completely hidden by the double-buffering technique.

The fact that the `trsm` is executed on the GPU already partly achieves our goal (a) to compute the most intensive part on the GPU. The size of L and thus

the number of samples n in the GWAS is limited by the largest possible buffer allocation of the GPU. This limit is not the same as the total memory available on the GPU. For instance, while current Fermi hardware has 6 GB of memory, it is not possible to allocate a buffer larger than 2 GB, which corresponds to $n = 16\,384$ and thus achieves our goal (g).

Figure 5.2: Profiled timings of the algorithm in Listing 5.1.

Figure 5.3: Summed runtime of the operations from Fig. 5.2.

5.2 Hiding the memory transfers and CPU computation

Clearly, this implementation is not optimal yet. As can be seen in Fig. 5.3, a bit more than a quarter (26.59% to be precise) of the whole algorithm's time is spent on CPU computation and data communication between the CPU and the GPU while only three quarters of the time is spent on GPU computations. This does yet not fulfill the goals we specified in Section 2.6. In order to get rid of this problem, we need to do two things: First, run the CPU computations for block b while the GPU computes the block $b + 1$, and second, transfer the data in the background while the computation takes place.

Luckily for us, modern GPUs[1] are able to perform computations at the same time as the memory transfers take place. This allows us to implement the same idea we used for hiding disk reads, namely double-buffering, in order to hide memory transfers to and from the GPU. On the downside, it turns out that when using two layers of double-buffering (one layer reads/writes from disk and another layer transfers to/from GPU memory), two buffers on each layer are not sufficient anymore.

[1]For nVidia, the G80 GPUs released in 2006 were the first to support this feature.

5.2.1 Two-layered double- and triple-buffering

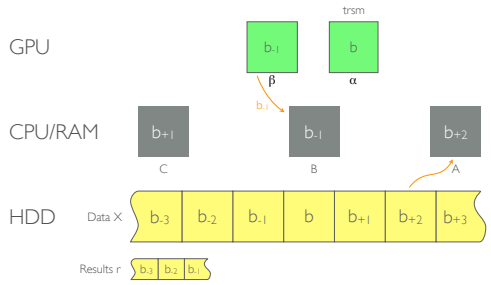
The idea here is to have two buffers on the GPU and three buffers on the CPU. The GPU buffers are used in the same way as the CPU buffers in the simple CPU-only algorithm: while one buffer α is used for computing, the data is transferred from and to the other buffer β. The three buffers on the CPU are now necessary because while a first buffer A is used for loading the second-next data block from disk, the previous result is retreived from the GPU into a second buffer B where the remaining CPU operations (lines 14-18 in Listing 5.1) take place and a third buffer C is used to send the next datablock to the GPU. A more detailed description of the algorithm is given in the following.

It is simpler to explain the algorithm by avoiding the beginning and the ending iterations; thus we jump right into an iteration of the algorithm, assuming that the $(b-1)$-th, b-th and $(b+1)$-th blocks already reside in the GPU buffers β, α, and the CPU buffer C respectively. Notice that block $b-1$ contains the solution of the previous `trsm` of b-1. As shown in Fig. 5.4a, the algorithm proceeds by *dispatching* the read of the second-next block $b+2$ from disk into buffer A, and by the computation of the `trsm` on the GPU on buffer α, and by receiving the result from buffer β into the buffer B. The two first operations are *dispatched*, i.e. they are executed asynchronously and don't block the CPU. The transfer of the results from the GPU back to the CPU is executed synchronously because these results are needed immediately in the next step.

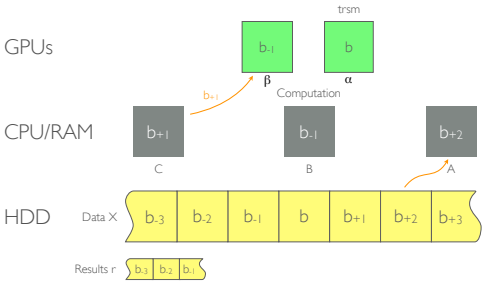
As soon as this synchronous transfer is done, the transfer of the next block $b+1$ from CPU buffer C to GPU buffer β can be dispatched, and the remaining operations (lines 14-18 in Listing 5.1) for the previous block $b-1$ can be started in buffer B. By doing this as shown in Fig. 5.4b, we now fully achieve our goal (a) of running the CPU computations in overlap with the GPU computations.

When the computation on the CPU is done, the results can be written to disk (Fig. 5.4c). Finally, once the write of $b-1$ to disk, the send of $b+1$ to GPU, the read of $b+2$ from disk, and the `trsm` of b are all done, buffers can be rotated (through pointer rotations, not copies) according to Fig. 5.4d, and the loop can continue with $b \leftarrow b+1$.

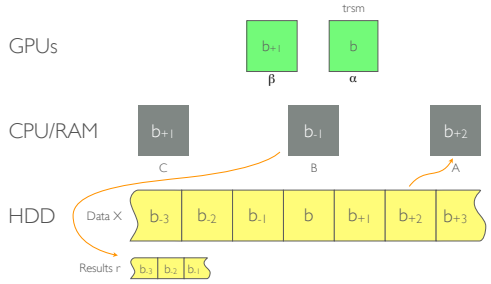
The previous description looks at the algorithm from the perspective of buffer management. Looking at it from the perspective of tasks while completely ignoring the buffers may also help understanding it better. Fig. 5.5 shows the timeline of one and a half iterations of the algorithm. The axis in the CPU section is the main line which depicts the flow of the program code, the other horizontal lines are dispatched from the main line and thus run asynchronously to it. Note that because this picture only intends to convey the idea, the sizes of the tasks are unrelated to their actual runtimes. The bars represent data dependencies;



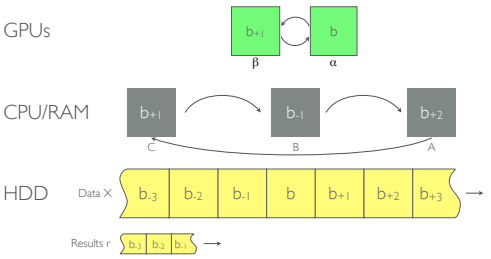
(a) Get the previous GPU result $b-1$ and the second-next block $b+2$ of data from disk.



(b) Send the next block $b+1$ to the GPU, execute the remaining computations on $b-1$ on the CPU.



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



(d) Switch buffers for the next iteration.

Figure 5.4: The double-triple-buffering algorithm as seen from a buffer perspective.

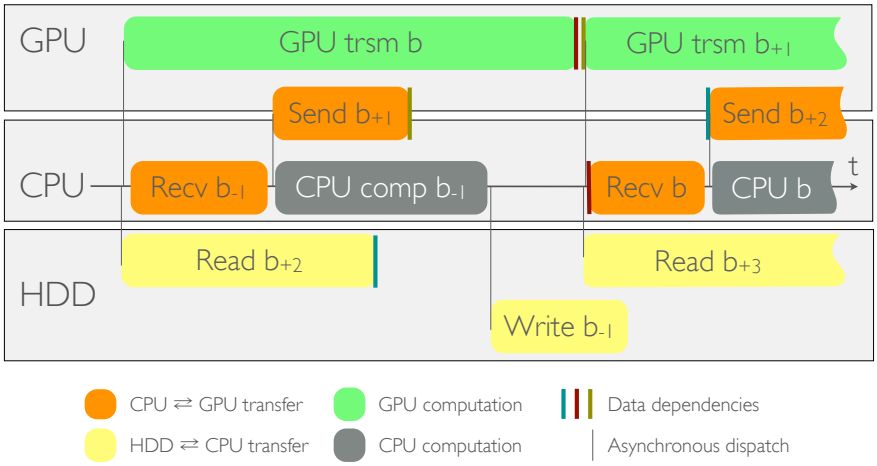


Figure 5.5: A timeline-perspective of the algorithm. Sizes are unrelated to runtime.

Figure 5.6: The time-profile of the final algorithm presented in this section shows that the goal of non-stop GPU computation (green) has been achieved.

for example, the task `Send b+2` has to wait for the task `Read b+2` to be done before it can start. These explicit data dependencies are only shown when tasks in different threads depend upon each other. Tasks as well as dispatches in the same thread always need the previous task to be done before they can start; these dependencies are implicit and thus not shown in the figure. The full pseudocode for this algorithm can be read in Listing 5.2.

One can see that if all CPU tasks end before the GPU `trsm` completes and all data dependencies are satisfied, the algorithm achieves goal (b) of non-stop computation on the GPU, which corresponds to the all-green profile seen in Fig. 5.6. While this sounds like a lot, all these conditions are usually satisfied: in our experiments, the GPU `trsm` still took much more time than all other operations and data transfers.

Listing 5.2: The parallelized algorithm described by Figs. 5.4 and 5.5

```
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  for b in -1..blockcount+1:
8      cu_trsm_wait α                              (if b in 1..blockcount)
9      cu_send_wait C → β                          (if b in 2..blockcount+1)
```

```
10      α ← cu_trsm_async L_gpu, α    (X̃_b = L^{-1} X_b)    (if b in 1..blockcount)
11      aio_read Xr[b+2] → A                                 (if b in -1..blockcount-2)
12      cu_recv B ← β                                        (if b in 2..blockcount+1)
13      aio_wait Xr[b+1] → C                                 (if b in 0..blockcount-1)
14      cu_send_async C → β                                  (if b in 0..blockcount-1)
15      for Xri in B:                                        (if b in 2..blockcount+1)
16          Sbl ← gemm Xri, Xl        (S_{BL_i} = X̃_{R_i}^T X̃_L)
17          Sbr ← syrk Xri            (X_{BR_i} = X̃_{R_i}^T X̃_{R_i})
18          rb  ← gemv Xri, y         (r̃_{B_i} = X̃_{R_i}^T ỹ)
19          r   ← posv S, r           (r_i = S_i^{-1} r̃_i)
20      aio_wait r[b-2]                                      (if b in 1..blockcount+1)
21      aio_write r[b-1]                                     (if b in 1..blockcount+1)
```

5.2.2 Results

The code has been tested on two different clusters nodes.

- *Quadro* is a cluster at the RWTH Aachen University with two nVidia Quadro 6000 GPUs and two Intel Xeon X5650 CPUs in each 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. This leads to a total GPU computational power of 1.03 TFlops. The CPUs, which have 6 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 $10 000 while the combined CPUs cost around $2000.
- *Tesla* is a cluster at the Universitat Jaume I in Spain with an nVidia Tesla S2050 system which contains four Fermi chips of the same model as *Quadro* but only 3 GB of RAM each. This amounts to a total of 2.06 TFlops. The CPU is an Intel Xeon E5440 delivering approximately 90 GFlops. We are grateful to Enrique S. Quintana-Ortì for granting us access to this system.

In all of the timings, the time to initialize the GPU as well as the initial computations (lines 1–6 in Listing 5.2) have not been measured. The GPU usually takes 5 s to fully initialize and the initial computations take a few seconds too, depending only on n and p. These have not been taken into account because a few seconds on startup are irrelevant for a computation which is intended to run for hours. In this section only the main computation loop is timed.

To get a first impression of the performance, a problem of the size described in Section 2.4, originally discussed by the developers of ProbABEL, on *Tesla* takes 2.88 s, compared to 4 hours using ProbABEL's GWFGLS algorithm. Even by

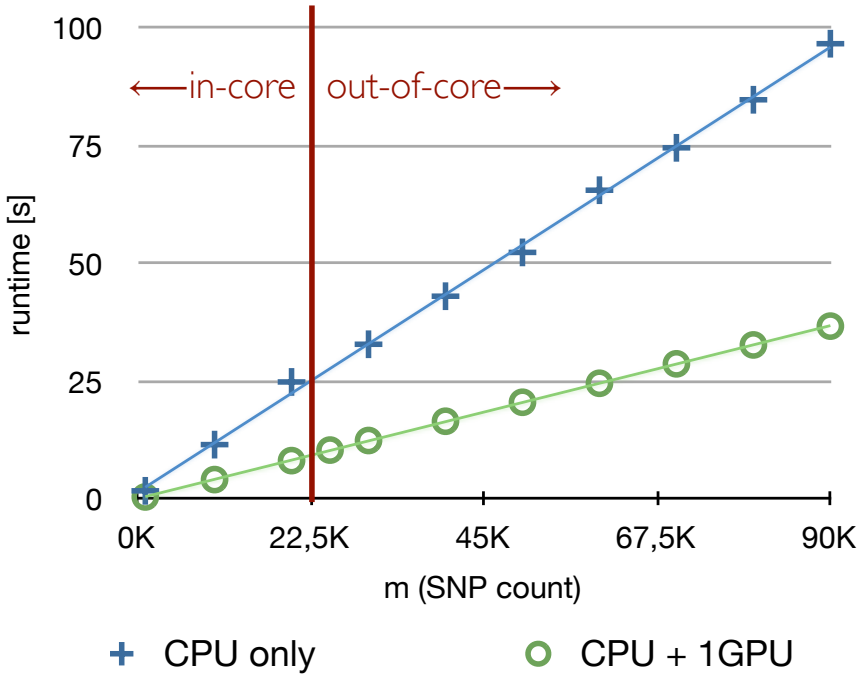


Figure 5.7: The timing of the CPU-only algorithm compared to the hybrid CPU-GPU algorithm.

adding about 6 s initialization time and accounting for Moore's Law (doubling the runtime as ProbABEL's timings are from 2010), the difference is still dramatic.

As a next performance comparison, Fig. 5.7 shows the runtimes of the original CLAK-Chol algorithm along with those of the hybrid CPU-GPU version of the algorithm, running on the *Quadro* cluster, and using one GPU. One can see that by leveraging the GPU for computing the `trsm` while executing everything else on the CPU at the same time, we achieved a 2.6x speedup over using the CPU only. As argued in Section 4.3, the implementation of `trsm` in cuBLAS attains about 60 % of the GPU's peak performance, i.e. about 309 GFlops. The peak performance of the CPU in this system amounts to 128 GFlops; comparing this to the aforementioned 309 GFlops shows us that the biggest speedup we can reach is 2.4x. Our speedup for the full algorithm is a little bigger than that because the time of the `trsm` on the GPU completely shadows the remaining operations on the CPU. This means that the performance of our implementation is perfectly in line with the theoretical peak.

In addition, the figure demonstrates that we achieved two more of our goals. First, we stated as goal (e) that the runtime of the algorithm should be linear in m. The sum of squared errors when fitting a line to the measured timings is exactly zero, which means that the measured timings are *perfectly* linear. Second, the

goal (f) of being able to cope with an arbitrary m dimension is also achieved. The red vertical line in the figure marks the largest value of m for which two blocks of X_R fit into GPU memory for $n = 10\,000$. Without the presented double-buffering technique, it would not be possible to compute GWAS with more than $m = 45\,000$ SNPs[2]. It is visible that the presented algorithm can compute GWAS with any given amount of SNPs.

5.3 Using more than one GPU

Even though we have shown that the implementation performs remarkably, we can still do better. It is becoming more and more usual to have more than one GPU in a computer. Even mid-budget notebooks recently include two GPUs[3]. Especially in the high-performance sector, boards with up to 4 chips such as the one in the *Tesla* cluster are already available. Our algorithm extends naturally to multiple GPUs by simply increasing the size of the X_{R_b} blocks by a factor as big as the number of available GPUs, and then splitting the `trsm` among these GPUs. This allows for easy parallelization to any number of GPUs. Listing 5.3 shows the final version of our parallel multi-GPU algorithm, which works for any number of GPUs.

5.3.1 Results and scalability

The exact same test as before (in Section 5.2.2) yields the timings shown in Fig. 5.8; results suggest that the runtime gets halved. In order to better evaluate the scalability with respect to the number of GPUs, we solved a GWAS with $p = 4$, $n = 10\,000$, and $m = 100\,000$ on the *Tesla* cluster varying the number of GPUs used. As it can be seen in Fig. 5.9, 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 0.54. Since a runtime reduction of 0.5 corresponds to perfect scalability (depicted by the green curve in the figure: doubling the number of GPUs cuts the runtime into half), it is evident that the attained scalability does satisfy our goal (c) of scaling to multiple GPUs.

Finally, in order to demonstrate the real-world benefits of this algorithm, we have run the algorithm on the problem presented in section 3.3 using the 4 GPUs of the *Tesla* cluster. The results are visible in Fig. 5.10, which uses a logarithmic scale

[2]Double the amount would be possible without doublebuffering because the whole memory could be invested in the single buffer.

[3]Although the reason here is to have a powerful one and an economic one, the powerful one only being turned on when needed.

on both axes. Unfortunately, we could not run tests larger than $m = 4\,000\,000$ because the available disk-space on the GPU cluster is not large enough. The other methods have been timed on CPU clusters which are connected to a very large storage facility. Because the runtime is almost perfectly linear in the number of SNPs, we are able to extrapolate the timings with high confidence, and thus get a full comparison. As it can be seen in the plot, the practical speedup is tremendous: by exploiting four GPUs, the computation time for analysing 36 million SNPs reduces from a full work-day to only slightly more than an hour and a half. When compared to one of the most widely algorithm currently used in ProbABEL (GWFGLS), the difference is even more dramatic: a reduction from 20 days to an hour and a half only.

Listing 5.3: The algorithm from Listing 5.2 using multiple GPUs. 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•   (X̃_b = L^{-1} X_b)  (if b in 1..blockcount)
12     aio_read Xr[b+2] → A                             (if b in -1..blockcount-2)
13     for gpu in 0..ngpus:                             (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:                             (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              (X_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)
```

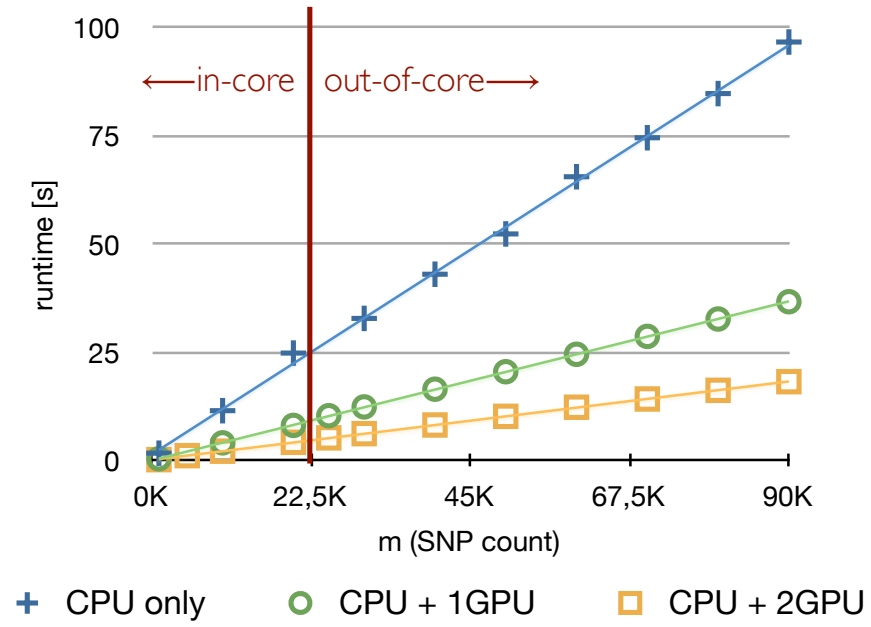


Figure 5.8: The timing of the CPU-only algorithm compared to the hybrid CPU-2GPUs algorithm.

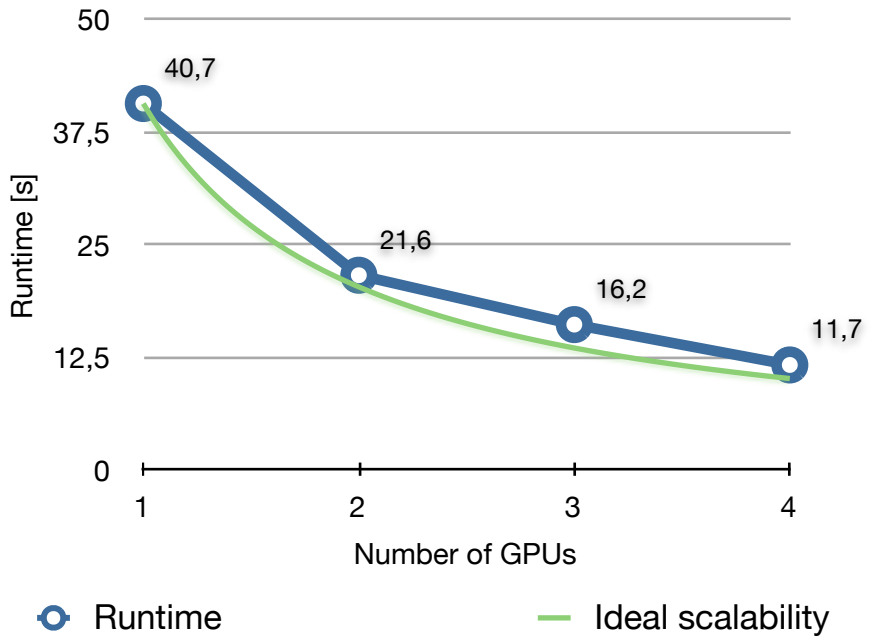


Figure 5.9: Runtime of the algorithm using a varying number of GPUs.

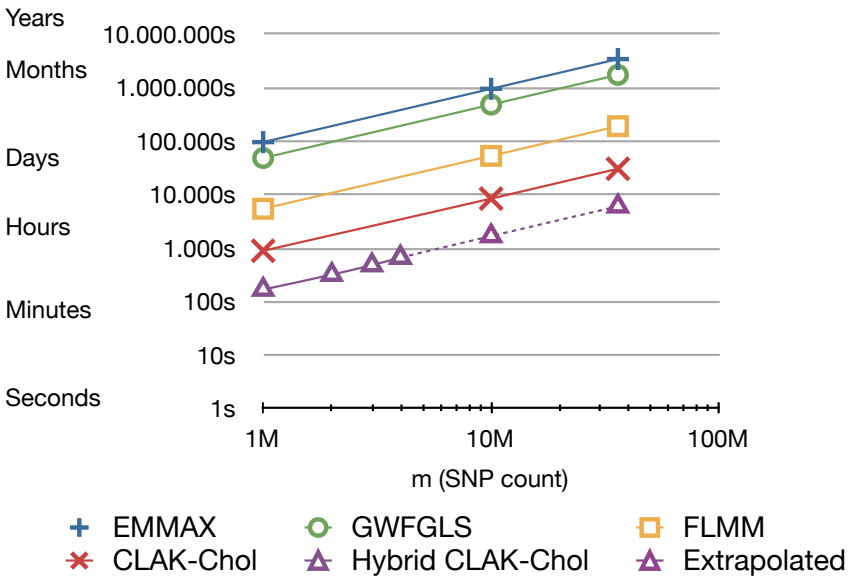


Figure 5.10: Comparison of the runtime of various algorithms with varying SNP counts.

Chapter 6

Realtime Visualization

We think that visualization is an important part of scientific work. Often times, the right visual representation can lead to key insights into data and results. But even if the focus of a work does not lie on the analysis of data –as is the case with this thesis– a good visualization conveys the mental image the author has of a problem. This helps to create a common basis during discussions and to explain the key concepts to people unfamiliar with them.

In this thesis, we visualize the problem being computed in the same way as depicted in Fig. 2.6 on Page 13. We render the problem in its actual dimensions, while it is being computed (i.e. in realtime), and highlight the parts which are currently being accessed and computed. Fig. 6.6 on Page 47 shows a photograph of the final visualization.

In the remainder of this chapter, we discuss the tools we used for creating the visualization as well as some implementation detail.

6.1 The hardware infrastructure

For our visualization, we used a virtual-reality system called *Powerwall*. It consists of a 4 m wide and 2 m high screen built into a wall of the room, two `Sony4K` projectors, a pair of glasses with markers, and several `ARTTrack` infrared cameras to track them. This infrastructure is controlled by ten computers: 8 identical ones called *powerwallclient01-08*, one *powerwallserver* and one *trackserver*. This setup is represented in Fig. 6.1.

The screen is divided into four quadrants. In each quadrant, the X11 screen of *two* powerwallclients is projected; when wearing the glasses, the left eye can only

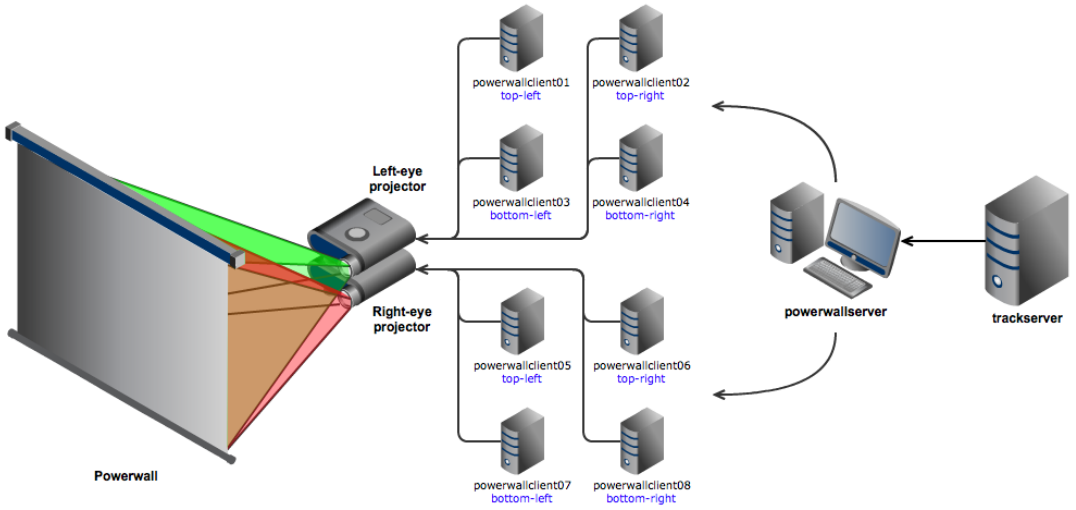


Figure 6.1: The hardware infrastructure of the powerwall.

see one of these screens while the right eye sees the other one. This makes it possible to trick the brain into seeing the scene as *real* 3D, by displaying the scene from a slightly different perspective for each eye. But for this to work, all eight displays need to be perfectly synchronized. NVidia's Quadro G-Sync addition cards connect all eight powerwallclients and the powerwallserver with eachother and provides a synchronized framebuffer swap[1] and frame counter.

6.2 The software ecosystem

Three separate applications are necessary for the visualization: the *renderer*, the *master* and the *gwascomp*. Before describing what each of them is responsible for, we need to explain the communication model.

6.2.1 The communication model

We use the ØMQ (pron.: zero-m-queue) library for all communications between the applications. ØMQ is often referred to as "sockets on steroids" and described by the diagram in Fig. 6.2. The reason for this appellation is that while ØMQ comes with a simple and minimalistic interface very similar to the familiar BSD

[1]The graphics cards usually employ a double-buffering technique too: the frontbuffer is shown on the screen while the next frame is being rendered into a backbuffer. When the scene rendering is done, the buffers are swapped out, resulting in the new scene being displayed on the screen.

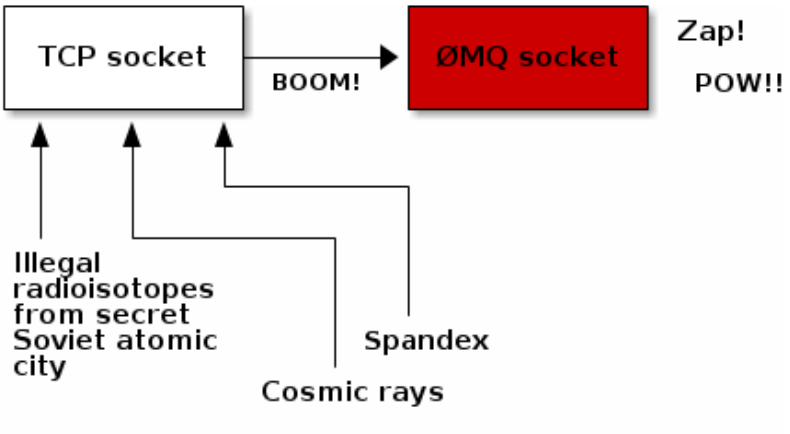


Figure 6.2: Quoting the official ØMQ documentation: "a ØMQ socket is what you get when you take a normal TCP socket, inject it with a mix of radioactive isotopes stolen from a secret Soviet atomic research project, bombard it with 1950-era cosmic rays, and put it into the hands of a drug-addled comic book author with a badly-disguised fetish for bulging muscles clad in spandex."

socket interface, behind the scenes it provides a tremendous amount of functionality and flexibility. ØMQ supports many well-known software communication patterns, including request-reply, push-pull, fan-out and pub-sub. The latter one is the communication pattern we use in all of our communications: whenever the *publisher* publishes a message, all of its *subscribers* receive this message, as shown in Fig. 6.3. (As opposed to the push/pull pattern, where only the first free subscriber receives the message.)

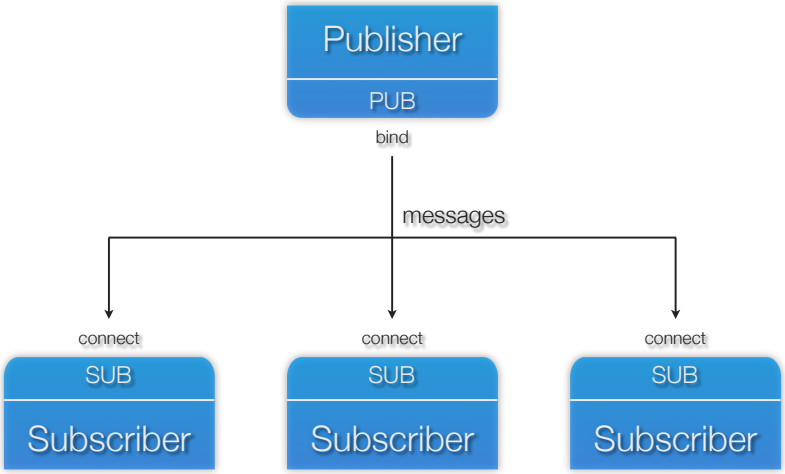


Figure 6.3: The publish-subscribe networking pattern.

6.2.2 The three applications

Now that ØMQ and pub-sub have been introduced, we can discuss the roles of the three applications.

The *gwascomp* application runs the GWAS computation on the powerwallserver. It publishes the current state of the computation along with the problem dimensions before the initialization, at the beginning of every iteration and at the end of the computation. That is, before the first line, between lines 8 and 9, and after the last line in Listing 5.3 (Page 38), respectively.

The drawing of the scene happens in the *renderer* application which runs on all eight powerwallclients. Each renderer needs to know the problem dimensions and the current state of the computation and thus subscribes to gwascomp. But in order to correctly render the scene from the viewer's perspective, the renderer needs to know the viewer's position and orientation at any given time. This is why the renderer subscribes to the *master*, which gets this information from the `ARTTrack` API once per frame and publishes the data. Fig. 6.4 gives an overview of all these interactions.

6.2.3 Changes to the transformation matrices

This section briefly explains the technical details necessary to create the illusion of "real" 3D as opposed to 3D on a flat surface. Explaining the mathematical foundations of 3D graphics is outside of the scope of this document and it is thus assumed that the reader is familiar with the basic concepts. A good introduction, written by Song Ho Ahn, is available at `http://www.songho.ca/opengl`.

In order to render 3D graphics onto a 2D screen, the 3D coordinate of every vertex[2] needs to go through several transformations. At first, the point v is brought from its *model* coordinate system into the *world* coordinate system using the *model-matrix* M: $v_{eye} = Mv$. This can be thought of as placing a 3D model into the world. Then, the point is transformed into the *eye*-space using the *view-matrix*: $v_{eye} = Vv_{world}$. The eye-space coordinates of the vertex represent its position in the world relative to the viewer. This transformation can be seen as placing a camera into the world and looking through it. Finally, the point is projected onto the screen and its coordinates are transformed to 2D coordinates through the *projection-matrix*: $v_{proj} = Pv_{view}$. This whole chain of transformations[3] is called the *model-view-projection* (MVP) transformation and

[2] A vertex is a point of a mesh in 3D space.

[3] Again, Song Ho Ahn explains these transformations in a much more detailed way on his webpage: `http://www.songho.ca/opengl/gl_transform.html`.

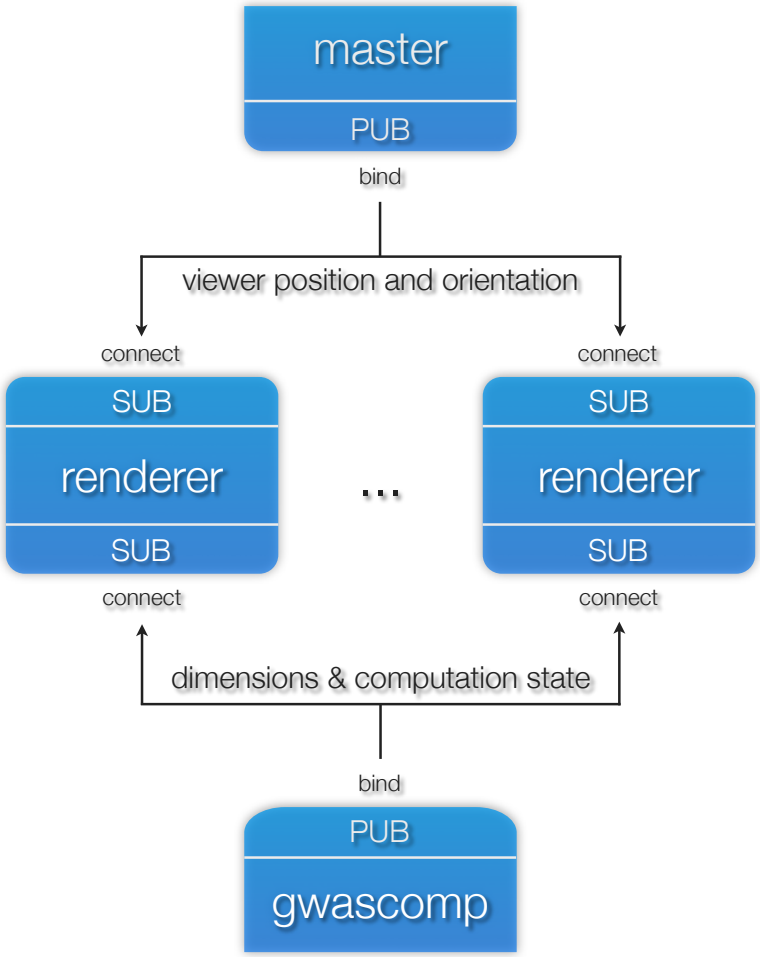


Figure 6.4: The interaction between the three applications necessary for the visualization.

often combined into a single step:

$$v_{proj} = MVPv. \tag{6.1}$$

In order to trick the brain into seeing the scene as real 3D, we need a different projection matrix for the left and right eyes. This idea is made clear by Fig. 6.5 which depicts a 3D scene both behind (left panel) and in front of (right panel) the projection surface (screen). The viewer's eyes are represented by the two dots on the left. For each eye, the 3D scene is projected onto the screen, resulting in the four points on the projection surface. One can see that the image on the screen needs to be different for each eye. For this to work, the projection matrices' projection plane needs to be at some point x between the front and the back

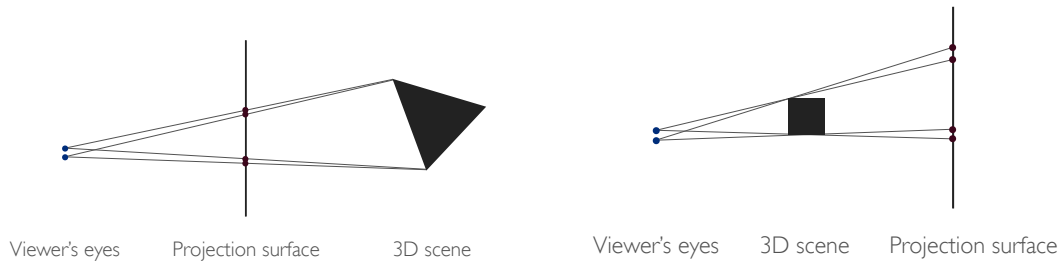


Figure 6.5: The 3D scene needs to be projected differently onto the screen for each eye. The left panel shows the projection of an object supposed to be behind the wall while the right panel shows the projection of an object supposed to be in front of ("come out of") the wall.

plane of the *viewing frustum*[4]. By following the derivation of the perspective projection matrix [1] with an arbitrary projection plane at $x \in (-1, 1)$, we come up with the following general perspective projection matrix, where r, l, t, b, f, and w correspond to the right, left, top, bottom, far and wall planes respectively:

$$P = \begin{pmatrix} \frac{2w}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2w}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & -\frac{f+w-2xw}{f+w} & -\frac{2xfw}{f+w} \\ 0 & 0 & -1 & 0 \end{pmatrix}.$$

By evaluating this projection matrix for each eye with the values for r, l, t, b, f and w computed relative to the eye position, it is possible to create the illusion of objects coming out of or moving into the wall.

[4]A frustum is the shape of a pyramid where the "head" has been cut off.

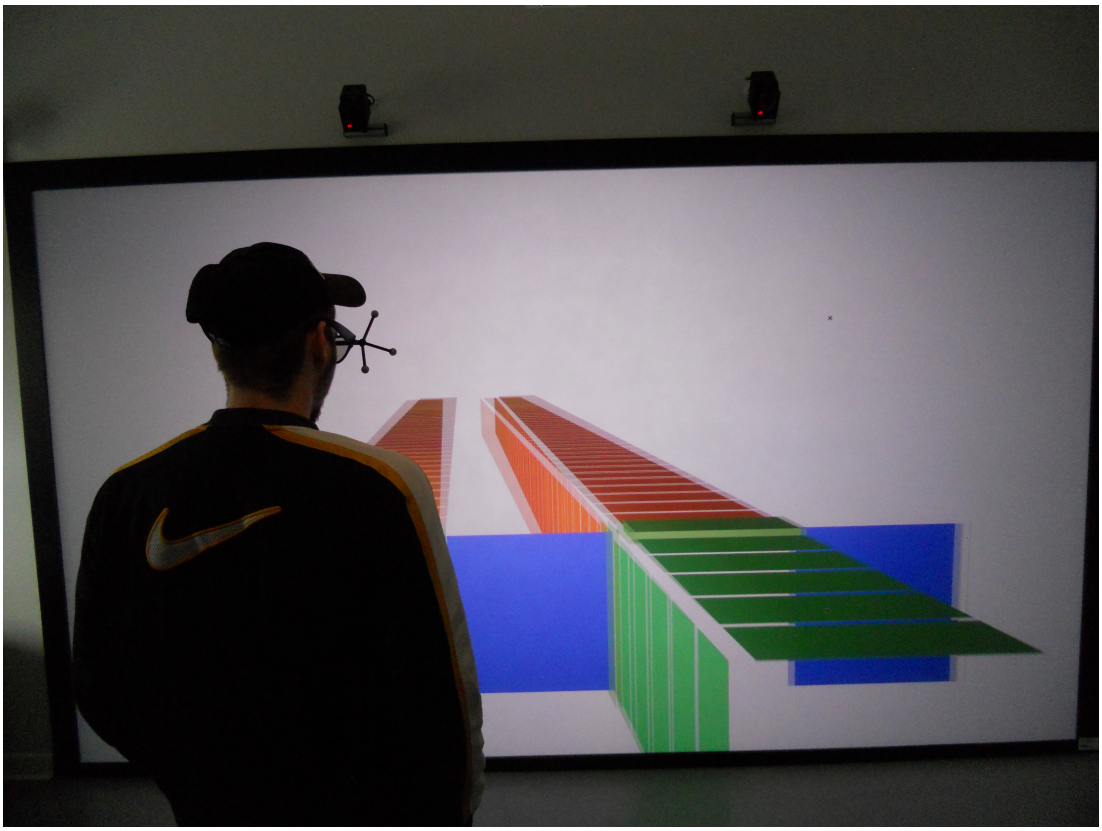


Figure 6.6: A photograph of the realtime visualization in our virtual-reality room.

Chapter 7

Conclusions

Genome-wide association studies have recently become a very popular and useful tool for linking genetic variants to traits and, more specifically, to major diseases. While the scope of these studies –especially the number of SNPs m– keeps growing, it is still far from practical to investigate all the 190 million known human SNPs due to the high computational cost: the currently most used methods take months for analyzing 36 million SNPs. While the recently published FLMM algorithm reduces the computational time to days, we are able to reduce it even further, to hours.

We achieved this speedup by making use of domain-specific knowledge, fast recent hardware (GPUs), as well as a layered multi-buffering technique. Our algorithm fulfills all the goals we listed in section 2.6: by using multiple buffers both on the CPU and on the GPU, we are able to keep the GPU computing without ever having to wait for data (goal b) or for the CPU (goal a). As Fig. 5.9 on page 39 clearly shows, the algorithm scales almost perfectly to multiple GPUs (goal c). Due to the nature of the algorithm, the sample size n is only limited by hardware memory, the current limit being around 16 000 (goal g). Finally, Fig. 5.8 on page 5.8 displays both a perfectly linear runtime with respect to m (goal e) and sustained high performance for an arbitrarily large SNP count m (goal f). Our goal d was to be future-proof and indeed, the future looks bright: Nvidia's very recently released GeForce GTX 680 has a theoretical peak performance of 3090.4 GFlops along with a `dgemm` efficiency of 80%[1] [17]. Because we chose to use cuBLAS instead of writing our own kernels, we expect our algorithm to greatly profit from this new architecture without us needing to change a single line of code.

While the completition of the human genome project in 2004 has made GWAS possible, our work makes them more practical.

[1] And consequently a more efficient `trsm`, as it is based mainly on `gemm`.

Bibliography

[1] S. H. Ahn. Opengl projection matrix.

[2] Y. S. Aulchenko, S. Ripke, A. Isaacs, and C. M. van Duijn. Genabel: an r library for genome-wide association analysis. *Bioinformatics*, 23(10):1294–1296, 2007.

[3] Y. S. Aulchenko, M. V. Struchalin, and C. M. van Duijn. Probabel package for genome-wide association analysis of imputed data. *BMC Bioinformatics*, 11:134, 2010.

[4] N. L. o. M. Bethesda (MD): National Center for Biotechnology Information. Announcement corrections: Ncbi dbsnp build 137 for human.

[5] B. Carlson. Snps — a shortcut to personalized medicine. *Genetic Engineering & Biotechnology News*, 28(12), 2008.

[6] I. H. G. S. Consortium. Finishing the euchromatic sequence of the human genome. *Nature*, 431:931–945, 2004 Oct 21 2004.

[7] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra. From cuda to opencl: Towards a performance-portable solution for multi-platform gpu programming. *Parallel Computing*, 38(8):391 – 407, 2012.

[8] D. Fabregat-Traver, Y. S. Aulchenko, and P. Bientinesi. Fast and scalable algorithms for genome studies. Technical report, Aachen Institute for Advanced Study in Computational Engineering Science, 2012. Available at http://www.aices.rwth-aachen.de:8080/aices/preprint/documents/AICES-2012-05-01.pdf.

[9] D. Fabregat-Traver, Y. S. Aulchenko, and P. Bientinesi. High-throughput genome-wide association analysis for single and multiple phenotypes - supplementary notes. 2012. Available at http://www.aices.rwth-aachen.de:8080/aices/preprint/documents/AICES-2012-07-01.pdf.

[10] K. Goto and R. Van De Geijn. High-performance implementation of the level-3 blas. *ACM Trans. Math. Softw.*, 35(1):4:1–4:14, July 2008.

[11] J. Gudmundsson, P. Sulem, D. F. Gudbjartsson, J. G. Jonasson, G. Masson, H. He, A. Jonasdottir, A. Sigurdsson, S. N. Stacey, H. Johannsdottir, H. Th Helgadottir, W. Li, R. Nagy, M. D. Ringel, R. T. Kloos, M. C. H. de Visser, T. S. Plantinga, M. den Heijer, E. Aguillo, A. Panadero, E. Prats, A. Garcia-Castano, A. De Juan, F. Rivera, G. B. Walters, H. Bjarnason, L. Tryggvadottir, G. I. Eyjolfsson, U. S. Bjornsdottir, H. Holm, I. Olafsson, K. Kristjansson, H. Kristvinsson, O. T Magnusson, G. Thorleifsson, J. R. Gulcher, A. Kong, L. A. Kiemeney, T. Jonsson, H. Hjartarson, J. I. Mayordomo, R. T. Netea-Maier, A. de la Chapelle, J. Hrafnkelsson, U. Thorsteinsdottir, T. Rafnar, and K. Stefansson. 44(3):319–322, 2012.

[12] H. LA, M. J. E. B. Institute), W. A, J. HA, H. PN, K. AK, and M. TA. A catalog of published genome-wide association studies. Accessed July 2, 2012.

[13] E. S. Lander, J. Listgarten, Y. Liu, C. M. Kadie, R. I. Davidson, and D. Heckerman. Fast linear mixed models for genome-wide association studies. *Nature Methods*, 8(10):833–835, 2011 Sep 04 2011.

[14] NHGRI. Genome-wide association studies. Internet: http://www.genome.gov/20019523, 2012. Retrieved July 2, 2012, from http://www.genome.gov.

[15] Nvidia. *CUDA CUBLAS Library.*

[16] Nvidia. *Fermi Compute Architecture Whitepaper.*

[17] Nvidia. *Kepler Compute Architecture Whitepaper.*

[18] Nvidia. *CUDA C Best Practices Guide*, Jan. 2012.

[19] R Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2012. ISBN 3-900051-07-0.

[20] M. Ridley. *Genome: The Autobiography of a Species in 23 Chapters.* P. S. Series. HarperCollins, 2006.

[21] V. Volkov and J. W. Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 31:1–31:11, Piscataway, NJ, USA, 2008. IEEE Press.

[22] F. O. Walker. Huntington's disease. *The Lancet*, 369(9557):218 – 228, 2007.

[23] R. C. Whaley and J. Dongarra. Automatically Tuned Linear Algebra Software. Technical Report UT-CS-97-366, University of Tennessee, December 1997. URL : `http://www.netlib.org/lapack/lawns/lawn131.ps`.

List of Figures

List of Listings

List of Tables