Entropy-based High Performance Computation of Boolean SNP-SNP Interactions Using GPUs

Carlos Riveros[1], Pablo Moscato[1], and Manuel Ujaldón[2]

[1]Centre for Bioinformatics, Biomarker Discovery and Information-based Medicine
University of Newcastle, Australia
[2]Computer Architecture Department, University of Malaga, Spain

Abstract. It is being increasingly accepted that traditional statistical single nucleotide polymorphism (SNP) analysis of genome-wide association studies (GWAS) reveals just a small part of the heritability in complex diseases. Study of interactions between SNPs has been suggested as a plausible approach to identify additional SNPs that contribute to disease but that do not reach genome-wide significance or exhibit only epistatic effects. We have introduced a methodology for genome-wide screening of epistatic interactions which is feasible to be handled by state-of-art high performance computing technology. Unlike standard software [1], our method computes all boolean binary interactions between SNPs across the whole genome without assuming a particular model of interaction. Our extensive search for epistasis comes at the expense of higher computational complexity, which we tackled using graphics processors (GPUs) to reduce the computational time from several months in a cluster of CPUs to 3-4 days on a multi-GPU platform [2]. Here, we contribute with a new entropy-based function to evaluate the interaction between SNPs which does not compromise findings about the most significant SNP interactions, but is more than 4000 times lighter in terms of computational time when running on GPUs and provides more than 100x faster code than a CPU of similar cost. We deploy a number of optimization techniques to tune the implementation of this function using CUDA and show the way to enhance scalability on larger data sets.

1 Introduction

GPUs have consolidated in parallel computing as low-cost platforms yet high performance alternatives whose scope of application spreads beyond graphical territory. On the programming side, CUDA [3] and OpenCL [4] have established the mechanisms for data intensive general purpose applications to exploit GPUs extraordinary power with remarkable scalability.

A natural scientific field to deploy this computational force is bioinformatics. The advent of the Human Genome Project has brought to the foreground of parallel computing a broad spectrum of data intensive biomedical applications where biology and computer science join as a happy alliance between demanding software and powerful hardware. Since then, the bioinformatics community generates computational solutions to support genomic and post genomic research [5] in many subfields such as gene structure prediction, phylogenetic trees, protein docking, and sequence alignment, just to mention a few.

The huge volumes of data produced by genotyping technology pose challenges in our capacity to process and understand data. In the field of genotyping experiments, ultra high density microarrays have jumped from 50.000 genes contained in a simple array to more than 5 million genetic markers. Furthermore, current clinical studies include hundreds of thousands of patients instead of thousands genetically fingerprinted a few years ago. The situation remains challenging, but when one has to choose a candidate to cope with this computational challenge, the GPU immediately comes to our minds.

The high performance computing capabilities of GPUs make now possible to tackle demanding applications on these huge volumes of data. Among these applications, the exhaustive search for associations of gene–gene interactions with disease (*epistasis*) is particularly challenging. The use of this interaction structure to reveal effects not readily seen in individual SNP-based analyses improves clinical analysis in many different ways [2], and constitute the cornerstone and major motivation for our biomedical application. In this work, we conduct a GPU implementation for a method based on a genome-wide screen of SNPs boolean function interactions using information entropy as evaluation function, and a subset from a real Genome-Wide Association Study as benchmark input data set (see section 7.1). The method is also readily extensible to the study of gene–environment interactions.

2 Related Work

Moore et al [6] used attribute interaction to select potential interacting SNPs and construct interaction graph. A major weakness of this method is that informs about the interaction between genetic data, but often cannot distinguish which two-locus model is proper for the interaction effects.

There are powerful methods to reduce dimensionality and to get a set of SNPs that can interpret the results best. Among them, we may cite the following three:

- S-statistics [7] within multilocus statistics methods.
- BOOST [8] and its GPU-accelerated version, GBOOST [9], are methods related to log-linear models which demonstrate its equivalence to logistic regression used by statistical methods, the most computationally expensive ones.
- Multifactor Dimensionality Reduction (MDR) [10] as data mining counterparts which does not distinguish a particular model. These techniques have already taken advantage of GPU performance with impressive results [11, 12].

BOOST and S-Statistics model SNP–SNP interactions through statistical modelling, and MDR classifies SNP variants combinations as low-risk or high-risk and measures the odds of such combinations.

To overcome these problems, we search for interactions patterns by computing all possible interactions between any two SNPs, and then measuring interaction strength by *p-value* or by *class information entropy*. In [2] we developed

Table 1. The 16 boolean functions of two variables and its five unique binary functions, F_1 to F_5. Other functions are marked as either *trivial (T)* (always true or false) or *unary (U)* (only depend on a single variable).

Inputs		Boolean	Identifier
A	**0 0 1 1**		
B	**0 1 0 1**	function	and logic
O_1	0 0 0 0	$\mathbf{0}$	T
O_2	0 0 0 1	$\mathbf{A} \wedge \mathbf{B}$	F_1: AND
O_3	0 0 1 0	$\mathbf{A} \wedge \neg\mathbf{B}$	F_2: ANDN
O_4	0 0 1 1	$\mathbf{A}$	U
O_5	0 1 0 0	$\neg\mathbf{A} \wedge \mathbf{B}$	F_3: NAND
O_6	0 1 0 1	$\mathbf{B}$	U
O_7	0 1 1 0	$\mathbf{A} \otimes \mathbf{B}$	F_4: XOR
O_8	0 1 1 1	$\mathbf{A} \vee \mathbf{B}$	$\neg F_5$
O_9	1 0 0 0	$\neg(\mathbf{A} \vee \mathbf{B})$	F_5: NANDN
O_{10}	1 0 0 1	$\neg(\mathbf{A} \otimes \mathbf{B})$	$\neg F_4$
O_{11}	1 0 1 0	$\neg\mathbf{B}$	U
O_{12}	1 0 1 1	$\neg(\neg\mathbf{A} \wedge \mathbf{B})$	$\neg F_3$
O_{13}	1 1 0 0	$\neg\mathbf{A}$	U
O_{14}	1 1 0 1	$\neg(\mathbf{A} \wedge \neg\mathbf{B})$	$\neg F_2$
O_{15}	1 1 1 0	$\neg(\mathbf{A} \wedge \mathbf{B})$	$\neg F_1$
O_{16}	1 1 1 1	$\mathbf{1}$	T

the former and accelerated its computation on GPUs. Here, we focus on the latter, addressing an optimized CUDA implementation which is also compared with our former study using *p-value*.

3 SNPs and Boolean Functions

Common genotyping platforms measure SNPs as *biallelic* variations. The standard genotyping process can only assert if the individual has (or has not) one and/or the other allele variant at a particular locus. The individual is then labelled as *heterozygous* if both variants are detected, and *homozygous* in one or the other variant if only one was detected. Accordingly, we considered each SNP variable as 2 different independent binary variables, denoted as rsN_M, rsN_m for SNP rsN, where the subscripts M and m indicate the 2 variants for that SNP.

Any interaction between two binary SNP variables rsA_x and rsB_y can be expressed across the data set as a binary function operator relating the result value to the two interacting operands. For two operands, there are 16 different ways to combine binary values to obtain another binary value, we list those cases in Table 1. Two results are *trivial functions*: either always *true* (1) or always *false* (0), another four results are *unary functions*: either equal to the $\boldsymbol{A}$ input, or to $\boldsymbol{B}$, or their negated forms. The remaining ten functions can be expressed in terms of five unique binary functions and their negated forms.

4 Significance Measurement

From a statistical point of view, significance is attributed to variables based on the observed frequency of appearance of values in the populations under study ("cases", "controls"). Variables whose difference is above a certain user-defined confidence threshold based on the accumulated probability of observing such difference through chance alone are attributed association with the populations. Statistical methods assign a P-value to such difference, computed approximately from the χ^2 statistic, or exactly from the non-parametric Fisher Exact Test; for details we submit the interested reader to the classical references[13, 14]. Alternatively, a significance can also be attributed based on the class information entropy difference carried by the variable.

4.1 Entropy

The information entropy H of a discrete variable x assuming L different values over N samples, with n_i samples of value x_i is:

$$H = -\sum_{i=1}^{L} p_i \log p_i = -\sum_{i=1}^{L} \frac{n_i}{N} \log \frac{n_i}{N}.$$

where the *probability* p_i of observing a random sample with value x_i has been replaced by the *observed frequency* n_i/N. If the samples carry a *class label*, we can then define the *class entropy* of the variable x as the weighted sum of entropy for each label, that is

$$H_{\mathcal{P}} = \sum_{q=1}^{P} \left(\frac{C_q}{N}\right) H_q \quad = -\sum_{q=1}^{P} \frac{C_q}{N} \sum_{i=1}^{L} \frac{C_{q,i}}{C_q} \log \frac{C_{q,i}}{C_q} = -\sum_{q=1}^{P}\sum_{i=1}^{L} \frac{C_{q,i}}{N} \log \frac{C_{q,i}}{C_q} \quad (1)$$

where there are P possible labels, and $C_{q,i}$ samples carrying the label q have been observed with value x_i, $\sum_q C_{q,i} = n_i$, $\sum_i C_{q,i} = C_q$. The index $\mathcal{P}$ refers to the partition of the set of M samples induced by the P labels.

We are interested in variables that decrease their class entropy when analyzed under the partition $\mathcal{P}$,

$$\Delta H_{0\to\mathcal{P}} = H_0 - H_{\mathcal{P}} \ \geq 0. \quad (2)$$

For a binary variable (taking only values 0 and 1) on a bipartition ("disease–control" study), the change in entropy is:

$$\begin{aligned}\Delta H = &-(\frac{n_{+1}}{N})\log(\frac{n_{+1}}{N}) - (\frac{N-n_{+1}}{N})\log(\frac{N-n_{+1}}{N})\\ &+(\frac{n_{01}}{N})\log(\frac{n_{01}}{n_{0+}}) + (\frac{n_{0+}-n_{01}}{N})\log(\frac{n_{0+}-n_{01}}{n_{0+}})\\ &+(\frac{n_{11}}{N})\log(\frac{n_{11}}{n_{1+}}) + (\frac{n_{1+}-n_{11}}{N})\log(\frac{n_{1+}-n_{11}}{n_{1+}}).\end{aligned}$$

5 CUDA

As a programming interface, CUDA consists of a set of C language library functions where the following elements meet:

Table 2. Data structures and parameters involved in our CUDA implementation.

Name	Size	Description
bitArray	$2 \times nS \times nI$ matrix of 64-bit integers.	A value for $2 \times nS$ binary SNPs (for each nI).
validArray	$nS \times nI$ matrix of bits (one per SNP value).	Validity of each SNP to account for missing values.
mask	Constant mask of bits.	For each individual, if belongs to *disease* or *control*.
values	$nF \times Q$ array of p-value or entropy values.	Q is chosen heuristically based on avail. resources.
counts	$2 \times nF \times Q$ array of counts.	For *true* bits in each class, for each function.
validCounts	$2 \times Q$ array of valid counts.	For *marginals*.
index	$nF \times Q$ array of indices.	Interaction between functions and binary SNPs.

Parameter	Meaning in our implementation	Value for our normalized data set (benchmark)
nS	Number of SNPs.	100 000
nI	Number of individuals.	1 000
nF	" boolean functions (see section 3).	5
nV	" independent variants per SNP (see section 3).	2
Q	The CUDA grid size (granularity of parallelism).	Number of threads per grid (see Table 7)

– A program is decomposed into **blocks** that run *logically* in parallel (physically only if there are resources available). Assembled by the developer, a block is a group of threads that is mapped to a single multiprocessor.
– All **threads** of concurrent blocks on a single multiprocessor divide the resources available equally amongst themselves. The data is also divided amongst all of the threads in a SIMD fashion with a decomposition explicitly managed by the developer.
– A **kernel** is the code to be executed by each thread. Conditional execution of operations can be achieved based on a unique thread ID.

In the CUDA model [3], all of the threads can access all of the GPU memory, but, as expected, there is a performance boost when threads access data resident in shared memory, which is explicitly managed.

6 An entropy-based CUDA implementation

To tackle a CUDA implementation of our methods, a preliminary observation for the five chosen boolean functions show that three are symmetric in their arguments (AND, NANDN, XOR) and the other two are mutually complementary. As a consequence, only half of the $(nS)^2$ pairs need to be computed. Two types of final results are of interest:

1. The list of N most significant interactions (where N is user defined). This requires a "running list" of topmost interactions, maintained by the CPU.
2. The aggregated sum of interactions significance, for each SNP. This requires an array of nS "running sums", also maintained by the host.

The data structures involved within our CUDA code are summarized in Table 2, along with the set of parameters which determine the size of the running benchmark. The parameter Q determines the *granularity* in which the problem is decomposed; it is chosen based on CUDA block size and available resources on the GPU. Ideally, each kernel invocation should work on a grid as large as possible. A comparison of relative per-thread time is given in Table 5.

We have decomposed our implementation into four kernels (see Table 3), giving us ample space to tailor launch parameters to the different arithmetic intensity and resource usage of each step.

Table 3. CUDA kernels considered for our GPU implementation.

Kernel	Task
`init`	Initializes output data structs. and resets counters.
`count`	Computes Boolean functions and bit counting.
`entropy`	Computes entropy for the computed results.
`sort`	Indirect sorting of results prior to transfer to host.

Our `count` kernel has the highest register use of 46 for compute capability 2.0 (Fermi cards) and 33 for compute capability 1.3 (Tesla), and therefore only one block can run per Fermi multiprocessor. The maximum block size is in this case 640 threads, and the multiprocessor occupancy tops at 42%. The number of registers could not be reduced due to the intrinsic nature of the algorithm.

The sorting stage was implemented using the well-tuned radix sort from Merrill & Grimshaw, incorporated into the `thrust::` library [15]. The speedup obtained is at least 4x compared to the same algorithm on CPU, and makes a case for sorting on device since data is already there. Examination of register usage by the different kernels suggest that a marginal improvement might be realised by a different geometry for the entropy computation phase.

6.1 Exploiting the memory hierarchy

The most important issue to take care of when aspiring to develop an efficient GPU implementation for our GWAS is the fact that data have to be handled on the fly, avoiding as much as possible any temporary storage on file.

Shared memory The `count` kernel was the more appropriate kernel to benefit from a shared memory implementation. However, the alternative performed worse than any non-shared memory version, for all feasible block sizes and GPU platforms. Data reuse was to enough to amortize the cost of the extra copy in shared memory, but shared memory resources limit the number of blocks that were able to be computed in parallel. Therefore, the final balance was negative in terms of performance and we finally decided to discard this variant.

Coalescing An additional 2 to 3 msecs. per kernel invocation was obtained by rearranging input data layout such that all memory fetches for a given SNP are contiguous in memory, and each block works on a consecutive range of SNPs. On the other hand, output data coalescing was improved by arranging output data in contiguous memory for a block. All experimental timings presented are with the coalescence-improved versions of our code.

Table 4. Summary of hardware features for the GPUs used in this work.

Processor type	CPU	GPU #1	GPU #2	GPU #3
Processor model	Xeon E5645	Tesla C1060	Tesla C2050	GeForce GTX 480
Number of cores	6	240	448	480
Core speed	2.4 GHz	1.30 GHz	1.15 GHz	1.40 GHz
Processing power	14.4 GFLOPS	312 GFLOPS	515.2 GFLOPS	672 GFLOPS
Memory speed	2× 666 MHz	2× 800 MHz	2× 1.50 GHz	2× 1.85 GHz
” bus width	192 bits	512 bits	384 bits	384 bits
” bandwidth	32 GB/s	102 GB/s	144 GB/s	177 GB/s
” size (type)	48 GB (DDR3)	4 GB (GDDR3)	3 GB (GDDR5)	1.5 GB (GDDR5)

7 Experimental analysis

To demonstrate the effectiveness of our techniques, we have conducted a number of experiments on different architectures. See Table 4 for hardware features.

7.1 Computational challenges and input data set

For a GWAS, undertaking the exhaustive evaluation of all binary SNP interactions is a daunting task. Computation time is linear on number of samples and quadratic on number of SNPs. A well tuned distributed implementation of the interaction count procedure takes about one month of processing time in a cluster of 128 cores. Our reference GPU implementation is the method described using p-values as evaluation function [2], that takes about 3 days on a dual-socket quad-core CPU server endowed with 4 nVidia C2050 GPUs for a data set containing 1315 samples.

A typical GWAS has more than 10^5 SNPs, measured for thousands of samples. An order of magnitude of the number of binary interactions evaluated is $\mathsf{nF} \times \frac{1}{2}(\mathsf{nV} \times \mathsf{nS})^2 = 5 \times \frac{1}{2}(2 \cdot 5 \cdot 10^5)^2 \simeq 2.5{\cdot}10^{12}$, where nV is the number of independent variants per SNP (2 in our case), nF is the number of binary functions (5 in our case), and nS is the number of SNPs. For our tests, we have considered a reference data set consisting of 100000 SNPs and 1000 samples, with 762 samples as “controls” and 328 as “disease”. This data set was obtained through a normalization process, generated by randomly selecting SNPs and samples from a larger study, preserving control/disease ratio. In addition, the algorithms have been tested against complete GWAS and reported elsewhere (see [16, 2]).

7.2 Optimal block and grid size

Using our input data set, we have conducted a number of experiments varying CUDA block sizes, grid sizes (the Q parameter) and GPU cards. For each GPU card, the times per thread remain virtually equal for block sizes of 64 (8x8), 128 (8x16), 256 (16x16), 320 (16x20) and 512 (16x32) threads (see Table 5). for

Table 5. Average processing time per grid execution (in milliseconds) and per thread (in nanoseconds) for our algorithm when using the GPU as co-processor, for three different grid sizes in Mthreads (see Table 7).

GPU (Nvidia model)	**Tesla C1060**			**Tesla C2050**			**GeForce GTX480**		
Grid size (# threads)	1 M	4 M	8 M	1 M	4 M	8 M	1 M	4 M	8 M
Thread time (ns.)									
`init`	1.27	1.26	1.22	0.81	0.80	0.80	0.60	0.59	0.58
`count`	19.94	19.72	19.30	3.68	3.63	3.58	2.79	2.75	2.71
`entropy`	1.17	1.18	1.03	4.02	3.97	3.93	3.07	3.03	3.00
Kernel time (ms.)									
`init`	1.33	5.26	10.17	0.85	3.34	6.63	0.63	2.46	4.87
`count`	20.81	82.65	160.61	3.84	15.20	29.81	2.91	11.51	22.57
`entropy`	1.23	4.94	8.58	4.19	16.63	32.68	3.20	12.69	24.94
Total time (ms.)	**23.37**	**92.85**	**179.36**	**8.88**	**35.17**	**69.12**	**6.74**	**26.67**	**52.37**

Table 6. Processing times (in milliseconds for each cycle on a grid, and in seconds overall) given grid sizes of 1M, 8M and 16M threads (10M for GeForce).

GPU (Nvidia model)	Tesla C1060			Tesla C2050			GeForce GTX480		
Grid size (in threads)	1 M	8 M	16 M	1 M	8 M	16 M	1 M	8 M	10 M
Number of cycles	19306	2485	1225	19306	2485	1225	19306	2485	1953
Cycle time (ms.)	83.6	651	1145	76.4	293.6	574.1	44.1	183.1	228.9
Kernel	23.4	179	367	8.9	69.1	139.2	6.7	52.4	66
Sort	43.7	395	624.5	28.6	142.5	279.8	8.6	60.2	75.2
Copy	11.1	76.9	153.3	13.8	77	150.2	10.1	70.4	87.6
Merge	28.0	27.6	27.5	28.9	29.3	29.7	25.3	24.1	22.3
Total (secs.)	**1622**	**1610**	**1403**	**1480**	**726**	**703.6**	**857**	**453**	**448**
Range of variability	± 66	± 350	± 15	± 4	± 10	± 4.5	± 71	± 7	± 2.8
% of variability	± 4%	± 21%	± 1%	± 0.3%	± 1.4%	± 0.6%	± 8.2%	± 1.5%	± 0.6%

results and Table 7 for threads geometry). As the GPU time per grid execution scales linearly with the size of the grid, the total execution time for the whole dataset is roughly independent of grid size and block size, provided the process becomes GPU-bound. The total computation time can then be accurately estimated as the product of: (grid size) $\times$ (block size) $\times$ (time per thread) $\times$ (number of grid launches).

Table 7. Block and grid sizes used when partitioning the problem using CUDA.

Block	Grid 1M		Grid 8M		Grid 10M		Grid 16M	
	Grid of 1 Mthreads		Grid of 8 Mthreads		Grid of 10 Mthreads		Grid of 16 Mthreads	
Threads per block	Blocks per grid	(threads per grid)	Blocks per grid	(threads per grid)	Blocks per grid	(threads per grid)	Blocks per grid	(threads per grid)
8×8	128×128	(1,048,576)	362×362	(8,386,816)	404×404	(10,445,824)	512×512	(16,777,216)
8×16	128×64	(1,048,576)	362×180	(8,340,480)	404×202	(10,445,824)	512×256	(16,777,216)
16×16	64×64	(1,048,576)	180×180	(8,294,400)	202×202	(10,445,824)	256×256	(16,777,216)
16×20	64×50	(1,024,000)	180×144	(8,294,400)	202×160	(10,342,400)	256×204	(16,711,680)
16×32	64×32	(1,048,576)	180×90	(8,294,400)	202×100	(10,342,400)	256×128	(16,777,216)

Total execution times and grid execution times for three different grid sizes are given in Table 6 (there appears to be an anomaly in the internal selection of parameters performed by the sort routine for the particular case of the 8×8, 8M grid and C1060 platform (see the 395 value), which increases the total execution time by more than 50%.

The grid sizes are outlined in Table 7, where they correspond to approximately 1, 8 and 16 million threads per grid (except in the GTX480 GPU, where the largest grid that can be computed contains 10 million threads). The `merge` operation executes on the CPU concurrently with the kernel on the GPU.

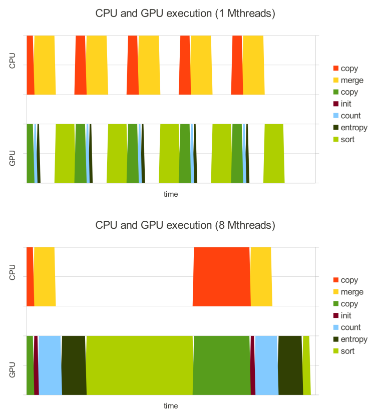
At the end, the computational time depends on which processor holds the bottleneck. In order to perform a deeper analysis of our execution, a profiling process was carried out. Figure 1 illustrates the time elapsed by those kernels/processes that CPU and GPU are responsible of. For a better understanding, we show time concurrently running on both sides, with processors synchronized when transfers (copies) are required. Results pose four different scenarios depending on large, larger, small and smaller grid sizes:

- **A large grid size leads to GPU-bound.** A higher degree of overlap between communication, GPU kernel execution and CPU execution (by using asynchronous transfers and multiple buffering) would reduce the total execution time and increase the size of the grid for which the process becomes GPU-bound. This is our winner scenario, and it is recommended to maximize the grid size to the extent permitted by video memory available once input data has been discounted (global memory in CUDA).
- **A small grid size leads to CPU-bound.** For grid sizes around 2 million threads and smaller, the merge time on CPU becomes larger than the kernel time on GPU. The process is then CPU-bound, and overall execution times are penalized. Our loser scenario, with smaller grid sizes of a million threads involving (slower) CPU processing sufficiently long to exceed the asynchronous simultaneous kernel processing.

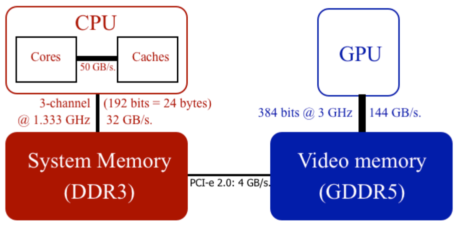
A more compact timeline could be achieved for smaller grid sizes if the time spent sorting would be used on the CPU for merging. Current `::thrust::sort()` implementation synchronizes CPU and GPU multiple times during execution, and to overcome this limitation we would require a multithreaded CPU implementation of our algorithm. Instead, our code relies more on massive parallelism on the GPU and adjusts itself to varying block and grid sizes, which allows us to employ simple heuristics in determining the value of the parameter Q. For example, in single card machines where the GPU is also the graphics processor, video memory can be the limiting factor. But in our experiments, where the GPU is seen as a co-processor and plays entirely the role of an accelerator, raw computational power can be more decisive. Next section confirms our hypothesis.

7.3 Influence of GPU resources: Tesla versus GeForce

An interesting survey on the GPU side concerns comparing different platforms where the same algorithm can be executed to determine which hardware re-



(a) Code profiling.



(b) Block diagram for processors and memories.

Fig. 1. (a) Timeline of CPU and GPU execution, for a CPU-bound case (1 Mthreads) and a GPU-bound case (8 Mthreads). (b) The way CPU/GPU interact with memory.

Table 8. Upper side: Elapsed time (in seconds) and speed-up when running a CPU single threaded version of our algorithm versus our GPU-assisted versions using CUDA. Lower side: Performance/cost analysis normalized to the CPU case for each GPU.

	CPU: Xeon E5520	Tesla C1060 (grid of 16 Mthr.)		Tesla C2050 (grid of 16 Mthr.)		GeForce GTX480 (grid of 8 Mthr.)	
Computation	45234.2	1195.0	37.9x	506.9	89.2x	270.2	167.4x
CPU-GPU communication	None	187.5		183.4		170.3	
Total execution time	45234.2	1384.9	**32.7x**	698.8	**64.7x**	441.2	**102.5x**
Estimated cost (release date)	$400	$1600		$1600		$400	
Scalability	Reference	Good		Better		Best	
Performance/cost ratio	1	**8.17**		**16.17**		**102.50**	

sources are more critical on a successful acceleration. In general, Teslas are endowed with a smaller number of cores and a lower clock rate, which essentially means less GFLOPS (Giga FLoating-point Operations Per Second), whereas GeForces have less memory capacity. According to Table 4, our Tesla C2050 doubles the video memory size, but suffers from 24% fewer GFLOPS with respect to the GeForce GTX480.

As it can be observed comparing total execution times in Table 6, once the process becomes GPU-bound the time remains approximately constant, and differences between platforms reflect the above mentioned hardware differences. In this case, and provided the binary SNP data matrix fits in memory, the processing can be split in a succession of sequential computations over non-overlapping grids, and the two most important factors become total number of cores and core frequency, which are responsible of the raw computational power (GFLOPS). GeForce GTX480 outshines Tesla C2050 by 55 to 60% less time; the error-correcting memory (ECC), which the C2050 has but the GTX480 does not, adds an additional delay in memory access of around 6-8% on top of the GFLOP difference. And hardware investment is around five times smaller on the GeForce, which also plays in favor of being selected as our favorite architecture.

7.4 Acceleration against CPU counterparts

Another interesting comparison goes to put side by side our GPU running times versus those obtained on our high-end Intel CPU Xeon E5520. Table 8 contributes with these numbers for our input data set. GPU gains exceed two orders of magnitude, even after considering the overhead of CPU-GPU communications. This is mainly due to the higher degree of data parallelism exploited through a fine grain strategy on the GPU. In general, a coarse grain via multiple threads (task parallelism) is not a scalable alternative for our SNP-SNP interaction problem, and we need to switch into CUDA and GPUs to pursue significant acceleration factors.

7.5 Cost/performance ratio on each platform

The lower half of Table 8 incorporates a study where the cost and release date are included for each hardware platform. Our reference unit is the performance/cost on CPU, and from that departure point, speed-ups are reduced a factor of four on the two Teslas because the processor cost is four times higher. Gains are maintained for GeForces because its price is similar to that of the Xeon CPU.

We can also see the good scalability exhibited by the GPU architecture: The older the model, the worst performance for a similar cost (comparing the two Teslas). And yet there is another interesting conclusion comparing GPUs of similar age in the last two columns: High performance computing on the GPU is achievable with a modest budget, as gamers make mid-end and low-end products increasingly popular but at the same time computationally demanding.

7.6 Further improvements

Additional refinements may lead to even better execution times. For example, making use of CPU cores via a multithreaded implementation and GPU capabilities to execute and transfer simultaneously on Fermi platforms. This way, the copy to host and merge operation on the CPU could be executed concurrently with the sequence `init-count-entropy-sort` for an additional factor of at least 2x speedup in the total processing time. At the same time, this may draw the benefit of a dual or even quad core CPU depending on the number of running threads that programmer can deploy at a given time.

7.7 Comparing evaluation functions: Entropy versus p-value

As we explained in section 4, every single SNP-SNP interaction is evaluated through a significance function whose computational complexity is critical for determining the entire workload and therefore the execution time.

A primary goal of this work was to reduce the computational complexity of such function without compromising the identification of the most significant SNP-SNP interactions. Our final assessment goes to quantify to what extent our entropy-based method has contributed to that reduction with respect to the former statistical methods relying on P-values. Using the same Tesla C2050 GPU and identical parameters, the evaluation of the P-values with the Fisher exact test takes 67.14 seconds for running what we were able to replicate here in just 16.4 milliseconds. This represents an improvement factor higher than 4000, which goes far beyond our initial expectations.

8 Summary and conclusions

We have developed GPU-based epistasis models and an entropy-based method to explore gene-gene interaction in SNPs aimed at discovering their potential genetic meaning. Experimental results on real data are shown to demonstrate that this method is effective in detecting gene-gene interaction which may contribute to the understanding of genetic mechanisms of common diseases. Our approach constitutes an alternative way to previous studies based on p-value which are more demanding computationally but similar in expression of interest.

We focus on CUDA for exploiting fine-grained parallelism, raw processing power and the memory hierarchy of the GPU, leading to speed-up factors exceeding two orders of magnitude versus counterpart methods implemented on high-end Intel Xeon CPUs. This is extraordinarily valuable in our case study where large-scale data sets and time constraints meet for a very demanding computation that is often unfeasible on a sequential computer. Our implementation also becomes competitive against other coarse grain parallelism alternatives such as distributed-memory and shared-memory multicomputers, and we have proven that it is a scalable solution evolving towards a more competitive performance/cost ratio as times goes by.

Acknowledgments. This work was supported by the Hunter Medical Research Institute Bioinformatics Fellowship, Australia, and the Ministry of Education of Spain under Project TIN2006-01078 and mobility grant PR2011-0144. We also thank NVIDIA for hardware donation under the following awards: Professor Partnership 2008-10, CUDA Teaching Center 2011-13, CUDA Research Center 2012-13 and, finally, CUDA Fellow 2012-13.

References

1. Purcell, S., Neale, B., Todd-Brown, K., Thomas, L., Ferreira, M.A., Bender, D., Maller, J., Sklar, P., de Bakker, P.I., Daly, M.J., Sham, P.C.: PLINK: a tool set for Whole-Genome association and Population-Based linkage analyses. The American Journal of Human Genetics **81**(3) (2007) 559–575
2. Riveros, C., Vimieiro, R., Holliday, E.G., Oldmeadow, C., Wang, J.J., Mitchell, P., Attia, J., Scott, R.J., Moscato, P.: Identification of genome-wide SNP-SNP and SNP-clinical boolean interactions in age-related macular degeneration. In: Methods and Technologies for Biomarker Analysis (tentative). [In Review]. Wiley-VCH (2013)

3. Nvidia Corporation: CUDA home page (2010) Nvidia Developer Zone, https://developer.nvidia.com/category/zone/cuda-zone.
4. The Khronos Group: OpenCL - the open standard for parallel programming of heterogeneous systems (2013) version 2.0, http://www.khronos.org/opencl/.
5. RoseIndia: A web site dedicated to bioinformatics tools, links, resources and tutorials (2009) http://www.roseindia.net/bioinformatics.
6. Moore, J., Gilberta, J., Tsai, C.: A flexible computational framework for detecting, characterizing, and interpreting statistical patterns of epistasis in genetic studies of human disease susceptibility. J. **241** (2006) 252–261
7. Hoh, J., Wille, A., Ott, J.: Trimming, weighting and grouping snps in human case-control association studies. Genome Res (11) (2001) 2115–2119
8. Wan, X., Yang, C., Yang, Q., Xue, H., Fan, X., Tang, N.L., Yu, W.: BOOST: a fast approach to detecting Gene-Gene interactions in genome-wide Case-Control studies. The American Journal of Human Genetics **87**(3) (2010) 325–340
9. 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. Bioinformatics **27**(9) (2011) 1309–1310
10. Hahn, L.W., Ritchie, M.D., Moore, J.H.: Multifactor dimensionality reduction software for detecting gene-gene and gene-environment interactions. Bioinformatics **19**(3) (2003) 376–382
11. Sinnott-Armstrong, N., Greene, C., Cancare, F., Moore, J.: Accelerating epistasis analysis in human genetics with consumer graphics hardware. BMC Research Notes **2**(1) (2009) 149
12. Hu, X., Liu, Q., Zhang, Z., Li, Z., Wang, S., He, L., Shi, Y.: SHEsisEpi, a GPU-enhanced genome-wide SNP-SNP interaction scanning algorithm, efficiently reveals the risk genetic epistasis in bipolar disorder. Cell Res **20**(7) (2010) 854–857
13. Fisher, R.A.: Statistical methods for research workers. Genesis Publishing Pvt Ltd (1932)
14. Agresti, A.: A survey of exact inference for contingency tables. Statistical Science **7**(1) (1992) 131–153 ArticleType: research-article / Full publication date: Feb., 1992 / Copyright © 1992 Institute of Mathematical Statistics.
15. Merrill, D., Grimshaw, A.: High performance and scalable radix sorting: A case study of implementing dynamic parallelism for GPU computing. Parallel Processing Letters **21**(02) (2011) 245–272
16. Riveros, C., Vimieiro, R., Moscato, P.: Cross-association of epistatic interactions in age-related macular degeneration. In: BDC 2012 Biomarker Discovery Conference @ Shoal Bay, Shoal Bay, Australia (2012)