Linear algebra over the field with two elements using GPUs

An attempt to add to M4RI a small core using the NVIDIA CUDA library

Jérémie Tharaud
ENSIMAG 2^{nd} year student
jeremie.tharaud@ensimag.imag.fr

Raphaël Laurent
ENSIMAG 2^{nd} year student
raphael.laurent@ensimag.imag.fr

ABSTRACT

Cryptography in general is mostly based upon algebraic computations. The efficiency of the algorithms that are used depends on how fast these computations can be made. That's why people try to gain as much speed as possible on this part. That's what the M4RI library is aimed at. It proposes a set of fast arithmetic functions on dense matrix over te field with two elements that are much and well optimized. It is a reference in the field. But M4RI only uses the CPU to do the computations, that's why we tried to see what could be gained from using the GPU to do the same tasks. In our work, we tried to understand what made the M4RI algorithms so efficient, and to see how we could adapt them to make a good use of the computational abilities of the GPU.

Keywords

Algebraic computations, $\mathbb{F}_2$, CPU, GPU, VRAM, CUDA, BLAS, CUBLAS, M4RI, row-major/column-major format

1. INTRODUCTION

Cryptography in general is mostly based upon algebraic computations. For instance, cryptographic attacks based on Groëbner bases [1] use intensively matrix-matrix multiplications. The efficiency of such algorithms is important because it determines how secure the cryptographic methods are. There exists good, well optimized implementations of matrix-matrix multiplications, such as the M4RI library (http://m4ri.sagemath.org) that contains a combination of the Strassen-Winograd and the M4RM ("Methods of the four Russians" Multiplication) algorithms, which is way faster than the well-known naive algorithm. But we thought it was possible to go even faster than that.

Indeed for the last few years, the GPUs (Graphical Processing Units) have been evolving, and now their use is not limited to computer graphics computations anymore. Now, the new GPGPU technique is born : General-Purpose computing on Graphics Processing Units, which means using GPUs to perform computation in applications traditionally handled by the CPU (Central Processing Unit).

What we have been working on is to see what could be gained from implementing the M4RI algorithms on GPUs. Such an attempt is legitimate insofar as GPUs are optimized for massive parallel computations, more than multicore CPUs are. Therefore, the implementation of matrix-matrix multiplication on GPUs consists mainly in parallelizing appropriately the existing algorithms. The aim of this article is to explain our work and to present a few comparisons in terms of performances (timing, especially) between the different GPU-based multiplication algorithms that we tried and implemented.

The paper is structured as follows. We proceed from the description of the "naive" algorithms (section 4 and 5) to the four russians method (section 7), illustrated by our results of benchmarking and comparisons (section 6 and 7.3). This is the heart of this article, but we thought that before that, it was appropriate to give some explanations about our data structures (section 2) and about programming on GPUs (section 3) first. There are then two more sections (section 8 and section 9) where we sum up the difficulties we encountered and conclude.

2. DATA REPRESENTATION

All our work only concerns algebraic computations over the field with two elements ($\mathbb{F}_2$). And there is a gain that can be made from working over $\mathbb{F}_2$: as each coefficient can only be 1 or 0, we can represent 64 coefficients with only *one* 64 bits machine word. Besides, the arithmetic operations in $\mathbb{F}_2$ are logical `&` and `XOR`, which are bitwise operations that can be done directly on 64-bits machine words. That's why we chose to use a flat row-major representation for our matrices. Thus we reduce one of the matrix dimensions by a 32 factor (a 64 bits machine is equal to 8 bytes whereas an 32 bits integer is 4 bytes).
For instance, we can store a 32768×32768 matrix in the VRAM since the compressed form needs $32768 \times 32768 \div 64 \times 8 \simeq 134MB$. Indeed, NVIDIA GPUs with CUDA capabilities have at least 256MB of dedicated memory.

This representation allows us to gain useful memory space, but we must be careful when processing computations based on these compressed matrices data. When we compute the product $\mathbf{A} \times \mathbf{B}$ of the matrices $\mathbf{A}$ and $\mathbf{B}$, we

need to read the lines of **A** and the colums of **B**. Our compressed representation is very convenient to read lines from the matrix **A**, but there is a problem when it comes to reading the columns of the matrix **B**. To bypass that issue, we chose to store in memory $\mathbf{B}^{\mathrm{T}}$ instead, in the flat row-major compressed format.

Although this idea may seem really naive, it brings a considerable improvement. We must not forget that any basic arithmetic operation on a GPU is roughly 4000 times cheaper than a read or write operation in the GPU global memory. In the NVIDIA CUDA programming guide, we can find the exact figures : the throughput of any bitwise operation is 8 operations per clock cycle, and the latency when accessing local or global memory may vary between 400 and 600 clock cycles. So, that's something that really must be kept in mind when programming on GPUs : the bottleneck is located at the memory reads/writes level.

3. PROGRAMMING ON GPUS WITH THE NVIDIA CUDA LIBRARY

Before getting more technical about how we implemented our algorithms, we think it is important to recall a few things about how GPUs work. The letters GPU stand for Graphical Processing Unit. Modern GPUs are very efficient at manipulating computer graphics, and their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. But to benefit from this highly parallel structure, problems must be approached in a quite different way, compared to CPU programming, and it is necessary to learn to program in a slightly different way from what one is usually used to. As getting more familiar with GPU programming has been an important part of our work, we want to try and summurize here what we learned about it, and hopefully it will be of some use to the readers of this article.

A GPU is composed of several multiprocessors that share a global memory space. Each multiprocessor is equal to 8 processors and can run from 768 parallel threads that share 16 KB of local memory, which is really faster than global memory. This memory is called shared memory, as it is a memory shared between the threads run by a multiprocessor. Plus, it is as fast as registers. The appropriate use of this fast local memory is crucial in order to improve the computations speed. Each multiprocessor also has 8 KB of local memory used to store constants. Each multiprocessor has 8192 registers shared between the threads of active blocks (8 active blocks at most). Knowing this figure is important to proportion the problem and use the capabilities of GPUs.

Whereas it is useful to know the exact hardware configuration to really deeply optimize algorithms, it is not really necessary to achieve a real gain compared to what is done by computing with the CPU. But here is what is really useful. The CUDA programmer must define *kernel functions* that will be executed in parallel by each thread, as opposed to more classic C functions that are run only once. Threads are gathered in thread blocks, that form a grid. Thus, the programmer has to specify for each call to a kernel, what the grid dimensions and the blocks dimensions are. That is to say, he or she has to specify how many thread blocks to use, and how many threads in each. Thread blocks are required to execute independently : it must be possible to execute them in any order, in parallel or in series. This independence requirement allows thread blocks to be scheduled in any order across any number of cores, enabling programmers to write code that scales with the number of cores. The number of thread blocks in a grid is typically dictated by the size of the data being processed rather than by the number of processors in the system, which it can greatly exceed. In the NVIDIA documentation, it is advised to use blocks of around 256 threads, because it offers a good compromise between memory latency and number of available registers for most kernels.

There is a programming pattern we used in our algorithms. It is a list of things we need to do each time. First, we allocate memory (`cudaMalloc`) in the device memory for the data upon which we want to do computations. At that point, it is usefull to check the status of the memory allocations (`cudaErrorMemoryAllocation`). Indeed, different devices may have different amounts of memory available, and it is useless to try to do computations when we don't have enough room available to store our data. Then we copy the data upon which we want to do the computations from the host to the device memory (`cudaMemcpy(destination, source, size, cudaMemcpyHostToDevice)`). After that, the dimensions of the blocks and of the grid are set, and a call to a kernel function is made. It is possible to do sequetially several calls to the same or to different kernel functions, depending of what treatment we want to do on our data. In that case, the dimensions of the blocks and of the grid need to be properly set for each call. Kernel functions take as paramaters pointers to memory zones where to read or write data, and other useful informations, such as matrices dimensions in our case. After the kernel functions calls comes the part were we retrieve the result from the device memory, and copy it to the host memory. Same as before, we call `cudaMemcpy`, but with `cudaMemcpyDeviceToHost` as last parameter this time. And finally we can free memory on the device using `cudaFree`.

4. THE NAIVE ALGORITHMS

This section describes a few naive GPU-based algorithms of matrix-matrix multiplications (with an $O(2n^3)$ complexity). For all algorithms, we took inspiration from the naive implementations given by NVIDIA in the CUDA Software Development Kit [2] and programming guide [3]. The aim of this part of our work on simple algorithms was to get more familiar progressively with programming using CUDA, and to obtain a lot of data for later comparisons with the other algorithms.

4.1 The classical product

We will not introduce the very well-known algorithm for matrix-matrix multiplication and its implementation in C. For our experiment, we used the GPU-based version given by the CUDA SDK [2] and we added timing functions to benchmark the algorithm and compare it to the CPU-based version. The programming consists in allocating three matrices (**A**,**B**) : entry values, **C** : result) in RAM, copy them to the VRAM (video memory), call the kernel and copy the result from VRAM to RAM. The kernel computes multiplications between all sub-matrices of each entry (**A** and **B**).

We compute separately each sub-matrix of the result matrix **C**. The shared memory is used to store the sub-matrices from **A** and for **B** we need to compute that sub-matrix. Each thread computes one element of the block sub-matrix. The dimensions of the sub-matrices are 16×16, which requires 256 threads. The coefficient of the result matrix computed by each thread is converted modulo 2.

The reason why we implemented such a simple and inefficient algorithm was we wanted to have an idea of what the GPU performances were, what we could gain compared to CPU performances, even on stupid computations.

4.2 Using the BLAS library

Basic Linear Algebra Subprograms (BLAS, [4])) is an application programming interface standard for publishing libraries to perform basic linear algebra operations such as matrix-matrix multiplications. Heavily used in high performance computing, highly optimized implementations of the BLAS interface have been developed by hardware vendors such as by Intel as well as by other authors (e.g. ATLAS is a portable self-optimizing BLAS [6]). For the comparison between CPU and GPU, we used the LAPACK library for C ([5] and [7]). The computation is very easy : we call the matrix-matrix multiplication kernel with the command `cblas_sgemm` then we convert the result modulo 2. In the GPU version, it is the same : there is no direct kernel invocation, the computation is done with the command `cublasSgemm` from the CUBLAS library [8]. The matrices need to be in a row major format in order to have the true result, unlike the CPU command where you can choose the matrices representation.

The BLAS library is a very optimized one. We use it mainly for comparisons purposes. In the general case, it would be foolish of us to try and do better than what BLAS does. But as we work only on matrices with coefficients in $\mathbb{F}_2$, there actually is some hope to do better. Because we have a compressed data representation format we can manipulate with bitwise operations, because we have subtile algorithms approriate to work over $\mathbb{F}_2$ that wouldn't work in the general case.

5. IMPROVED DOT-PRODUCT

If we come back to the basics of matrix-matrix multiplications, we notice that it is based on dot products between row and column vectors. Since there are n^2 dot products, it could be useful to fasten their executions. To do that, we use the compressed format for the matrices (matrices of 64-bits machine words, as explained in section 2). Besides, as we do computations in $\mathbb{F}_2$, the product is a logical `&`, and the sum is a logical `XOR`, which are very low cost operations to do on 64 bits words. After that, there only remains to compute the parity of a 64-bits word to have the result. Here is how it works :
Given two vectors **A** and **B**
$\mathbf{A} = (a_i)_{1<=i<=n}$ and $\mathbf{B} = (b_i)_{1<=i<=n}$, the usual dot product is given by :

$$\mathbf{A}.\mathbf{B} = \sum_{i=1}^{n} a_i \times b_i.$$

Here, we compute instead :
$\mathbf{A}.\mathbf{B}$ =`parityof`($a_1 \& b_1$ `XOR` ... `XOR` $a_{\frac{n}{64}} \& b_{\frac{n}{64}}$).

`parityof` computes the parity of a 64-bit machine word (e.g. returns the number of bits that are set to 1 in the binary representation of 64-bit integer parameter).
To do that, we used an integer function ([3] C.2.3) :
`parityof(v)=__popcll(v) & 1` .

Now, about parallelizing, we use the same block partitionning as we did for the naive classical dot product. That is to say that each thread block computes a sub-matrix of the result matrix, and inside a thread block, each thread computes a coefficient. The only thing that changes from the classical matrix product is that the way to compute is a little different : we use our compressed data representation format and bitwise operations instead of integers and sums and products.

6. PERFORMANCES ANALYSIS

Our benchmarks were done on a computer which has sixteen 2.2 Ghz AMD64/Opteron and two NVIDIA GTX280, running on a 64-bit Debian/GNU Linux.
But the tests only use 1 CPU and 1 GPU. Let's remember that this GPU has 30 multiprocessors and 1GB of VRAM. The algorithms was compiled with GCC 4.3.3 and we used the option $-O2$.

Table 1: Naive product CPU versus GPU

Matrix Dimensions	CPU	GPU
1024x1024	26,06	0,048
2048x2048	288,7	0,275
4096x4096	-	1,857
8192x8192	-	13,059

We didn't try to launch computations on the CPU for matrices bigger than 2048×2048, because it would have taken too much time, and the result we have are sufficient to say that for this naive implementation, the GPU computing brings a huge gain (between 50 and 100 times faster). Yet, we can not launch computation for matrices bigger than 8192×8192 as we are limited by the size of VRAM (used to store the matrices).

Table 2: BLAS CPU versus GPU

Matrix Dimensions	CPU	GPU
1024x1024	0,356	0,049
2048x2048	2,752	0,205
4096x4096	21,79	0,985
8192x8192	170,8	5,415

Unsurprisingly, we see here too that computations made with the GPU are faster (between 7 and 30 times faster).

We also were able to do some tests on a GeForce 8600M GS, which is a more common and affordable Graphic card. It has 4 multiprocessors and 512 MB of dedicated VRAM. These tests give an idea of what can be achieved on an average machine.

Table 3: Tests on 8600m GS

Matrix Dimensions	Naive Product	BLAS
1024x1024	0,5797	0,1389
2048x2048	4.5739	1.0047
4096x4096	36.8049	7.7279
8192x8192	-	-

Same problem here : we couldn't compute products of matrices of size 8192×8192 because there is not enough memory available on the device to store the matrices. We see here that the more expensive card is 8 times faster for the bigger matices dimensions we could try, so it seems it is worth investing in good hardware components even for a GPGPU usage.

7. THE FOUR RUSSIANS METHOD

7.1 Idea of the algorithm

The "Method of the Four Russians" matrix multiplication algorithm can be derived from the original algorithm published by Arlazarov, Dinic, Kronrod, and Faradzev [11], but does not directly appear there. It has appeared in books including [10]. It is well discribed in this paper [12], which is focused on the numerous techniques employed for the special case of $\mathbb{F}_2$ in the M4RI library and the benefits so derived.

To understand the algorithm, we must see that, when computing the $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ product, each line of the matrix $\mathbf{C}$ is a linear combination of the lines from the matrix $\mathbf{B}$. It is the i^{th} line from matrix $\mathbf{A}$ that determines which lines from the matrix $\mathbf{B}$ appear in the i^{th} line of matrix $\mathbf{C}$. Hence the idea of precomputing the 2^k possible linear combinations of lines from matrix $\mathbf{B}$ for a given k, and store them in a table. When we have such a table, the multiplication algorithm is just to read successively each line from matrix $\mathbf{A}$ and to read in the table to know what is the coresponding line of matrix $\mathbf{C}$.

The idea to tabulate the possible values may seem surprising at first, but there are several arguments to support it. First, as we only do computations over $\mathbb{F}_2$, the coefficients can only be 0 or 1, so there is a sizeable probability that several lines of matrix $\mathbf{A}$ are the same, and then we spare time not doing redundant computations. Secondly, the precomputation of the table is not as costly as one could think. As a matter of fact, this precomputation can be done with an order that makes it really quick : the idea is to use a *gray code*. This way, from a linear combination we deduce the next one at almost no cost, simply by adding/substracting (arithmetic `XOR` in both cases) the corresponding line from matrix $\mathbf{B}$ to/from it.

7.2 Implementation using the GPU

There are here the same steps as in the other algorithms, so we won't detail the memory allocations and copies. What is more of interest is to detail the kernel function that we use. The partitioning we chose to use is the following : Each thread block computes the coefficients of a sub-matrix of the result matrix. The grid dimensions are the result matrix dimensions divided by the dimensions of a block. The dimensions (y,x) of a block must be chosen in such a way that there is enough room in the shared memory to store a sub-matrix of matrix $\mathbf{A}$, a sub-matrix of matrix $\mathbf{B}$, and the table which contains linear combinations of lines from the sub-matrix of matrix $\mathbf{B}$.

There are several important steps, that are separated by calls to `__syncthreads()` ; thus we are sure that each thread has finished earlier computations when a thread begins the next step. The first step is to load the sub-matrix of matrix $\mathbf{B}$ in shared memory. Each thread loads a coefficient. Next, we compute the table of linear combinations of lines from this sub-matrix. To do that, each thread will compute several lines of this table. It computes the first line by analysing the corresponding gray code one bit after another, and by adding the corresponding line each time a bit is 1. But after this first line is computed, for the following ones, the thread copies the previous line, it computes the next gray code, computes the bit that is different from the previous gray code, and then `XOR` only the corresponding line. The next step is the computation of the result itself. To do that, we need a loop to iterate on sub-matrices of matrix $\mathbf{A}$. In that loop, there are several steps. First, we load a sub-matricx from matrix $\mathbf{A}$ in shared memory. Same as usual, each thread loads one element. The next step is to calculate the indices corresponding to the bits from the lines of matrix $\mathbf{A}$ in the table where the linear combinations are stored. What we do here is that only a few threads are active because there are only a few indexes to compute. After this step, the indexes are in the shared memory so that any thread of the block can access to them. After that, there only remains to `XOR` with the corresponding line in the result matrix the relevant linear combination from the table. At this point all threads are active. After that, we go to the next block sub-matrix of matrix $\mathbf{A}$, and we keep iterating until the loop is over, i.e. all sub-matrices of matrix $\mathbf{A}$ have been taken into account.

7.3 Comparison and Performances analysis

We made some benchmarks of M4RI on our 64-bit Debian/GNU Linux :

This gives an idea of what we have to beat. But unfortunately, right now we don't have real results with our algorithms because their implementation is not stable and operative yet.

Table 4: 2.5Ghz Core 2 Duo, 8600m GS, GTX280

Matrix Dimensions	CPU	GPU1	GPU2
8192x8192	0,65	-	-
16384x16384	4,81	-	-
32768x32768	34,2	-	-

8. GPU PROGRAMMING TYPICAL DIFFICULTIES

The main difficulties we coped with were the programming of kernels. It is quite different as CPU programming because they are very optimised for parallelism. It changes the way to compute because we have several parameters to take into account (different type and size of memory, threads). Moreover, debugging is very difficult since it is impossible to call host functions inside the kernel (no printf, ...) and there is no warning when a allocation or execution failed. Fortunately a debugger exists and can help (CUDA GDB) although it is still in beta version. NVIDIA also released a library named CUtil which checks memories, executions,etc., but its use is far from being easy (there is no help).

9. CONCLUSION

The actual power of high-end graphic cards shows that a simple naive algorithm is executed faster with a GPU than with a CPU. The capabilities of GPU only wait to be exploited. Now the current version of CUDA (2.2) is very stable and high-end GPU are always cheap so everyone can develop and contribute to the CUDA community, assuming to have some basic knowledge in C/C++. During this project, we developed a little core which can still be enhanced since GPUs offer more possibilities than CPU programming. To go further in the project, the dot product by parity can be improved by computing the parity of 64 words in one go with a tree (resulting in a better complexity) instead of one by one. The idea would consist in working with 1×64 submatrices and compute the parity of 64-bit machine words, obtained by having `AND`ed and `XOR`ed 64 times 64-bit machine words of the compressed forms. Furthermore, NVIDIA introduced enough tools to manipulate the device. Indeed, a possibility of improvement could be the computation on multi GPU (it was possible on our test machine). As we saw several times, parallelism is crucial to obtain the best performances.

In its current state, the project is unfortunately not as advanced as it could be. It is too bad that the two most interesting algorithms we worked on don't work at this point. We think we are very close to get them working, but there is a mistake somewhere and we couldn't figure out which untill now. Anyway, we think that we have succeeded at least in showing that there was much to be gained from using GPUs to do algebraic computations over $\mathbb{F}_2$. Which is a reason sufficient to continue to work on our algorithms to make them work and to further optimize them, and later to adapt other algorithms such as the Strassen-Winograd product to use the GPU. There will be a lot of questions to be considered when our algorithms will be operative. If our algorithms are indeed faster (what preliminary tests seem to indicate, but as tey were done on algortithms that don't compute the right results, we couldn't put them here) we will have to see where exactly are the memory limits different graphic cards, and to search for appropriate cutoffs in cascade algorithms.

10. REFERENCES

[1] Groëbner Basis. Available at +http://en.wikipedia.org/wiki/Groebner_basis, 2009

[2] NVIDIA Cuda SDK v2.2. Available at http://www.nvidia.com/object/cuda_get.html

[3] NVIDIA Programming Guide v2.2. Available at http://developer.download.nvidia.com/compute/cuda/2_2/toolkit/docs/NVIDIA_CUDA_Programming_Guide_2.2.pdf, 2009

[4] BLAS definition. Available at http://en.wikipedia.org/BLAS

[5] LAPACK definition. Available at http://en.wikipedia.org/LAPACK

[6] Automatically Tuned Linear Algebra Software (ATLAS). Available at http://math-atlas.sourceforge.net

[7] The University of Tennessee. LAPACK – Linear Algebra PACKage v3.2.1. Available at http://www.netlib.org/lapack, 2008

[8] NVIDIA Cublas Library v2.0. Available at http://developer.download.nvidia.com/compute/cuda/2_0/docs/CUBLAS_Library_2.0.pdf, 2008

[9] Sean Eron Anderson. Bit Twiddling Hacks. Available at http://graphics.stanford.edu/~seander/bithacks.html

[10] Gregory V. Bard. Accelerating Cryptanalysis with the Method of Four Russians. Cryptology ePrint Archive, Report 2006/251, 2006. Available at http://eprint.iacr.org/2006/251.pdf

[11] Martin Albrecht, Gregory Bard, William Hart. Efficient Multiplication of Dense Matrices over $GF(2)$. Available at http://arxiv.org/abs/0811.1714

[12] Gregory Bard. M4RI - Linear Algebra over $\mathbb{F}_2$. Available at http://m4ri.sagemath.org