A parallel algorithm for dot product over word-size finite field using floating-point arithmetic

Jérémy JEAN Stef GRAILLAT
UPMC Univ Paris 06 and CNRS, UMR 7606, LIP6, PEQUAN Team
4, place Jussieu
F-75252 Paris cedex 05 (France)
Email: Stef.Graillat@lip6.fr

***Abstract*—Recently, parallel computation has become necessary to take full advantage of the gains allowed by Moore's law. Many scientific and engineering applications exhibit data parallelism but might not make full use of it. Some ubiquitous operations such that the dot product can easily be parallelized and then make good use of available hardware, like multi-core or GPU. In this paper, we provide two slightly different algorithms to perform dot product calculations in a finite field using floating-point arithmetic and implement them on the GPU architecture. To do so, we pack input integers into floating-point numbers and exploit the computational capabilities of GPU to their full extent to get the result efficiently. Using error-free transformations, we show that it is possible to reach speedups between 10 or 40 with the parallel versions, with an algorithm using nearly no modular reduction.**

***Index Terms*—Finite field, floating-point arithmetic, error-free transformations, FMA, GPU, CUDA, GPGPU.**

I. Introduction

Let $p \geq 3$ be a prime number, and $(a_i), (b_i)$ two vectors of N scalars in $\mathbb{Z}/p\mathbb{Z}$. We want to compute the dot product of a and b in $\mathbb{Z}/p\mathbb{Z}$,

$$a \cdot b = \sum_{i=1}^{N} a_i \, b_i \pmod{p}.$$

The underlying issue of this calculation is the way we represent and manipulate numbers. In this paper, we choose the floating-point representation for numbers, and look for a way to perform operations exactly and in parallel.

A way of computing efficiently dot products in word-size fields has been presented in [1]. We extended this work in [2] to deal with greater prime number p. We now suggest a parallel version of it. N. Yamanaka *et al.* suggested in [3] a parallel version of an accurate dot product algorithm. Algorithms presented in this paper solve this problem in a finite field.

Our algorithms are less efficient that the ones of [1] for moderate size of prime p. The only advantage of our algorithms is that they can use greater prime number p. Moreover, we cannot compete with integer computations (RNS algorithms) if integer arithmetic units are available. Our algorithms are useful if only floating-point units are presents or if there are more floating-points units than integer units. This can be the case in embedded processors.

Outline. The paper is organized as follows. In Section II, we describe the GPU rising technology and its programming model known as CUDA. In Section III, we provide the basics of the floating-point arithmetic needed to understand the paper. More complete introduction on this subject can be found, for instance, in [4], [2], [5]. Section IV is dedicated to our new algorithms, where we explain how they work and how they can be parallelized. Final Section V exposes experimental results for all implementations we have made.

II. Overview of CUDA

The presentation of CUDA relies on [6].

A. Generalities

Today, parallel GPUs have begun making computational inroads against the CPU, and a subfield of research, dubbed GPGPU for General Purpose Computing on GPU[1], has found its way into many fields. There is increased pressure on GPU manufacturers like NVIDIA from GPGPU users to improve hardware design, usually focusing on adding more flexibility to the programming model. In this objective, NVIDIA introduced CUDA, a general purpose parallel computing architecture – with a new parallel programming model and instruction set architecture – that leverages the parallel compute engine in NVIDIA GPUs to solve many complex computational problems in a more efficient way than on a CPU. That is this technology that we will be using in the following. Getting new algorithms that fit GPU arichecture and make it possible to get high performance is a challenging problem. For exemple, research in that direction are projects such as the Magma project developped by Jack Dongarra (http://icl.cs.utk.edu/magma/). In this project, they design new approaches for linear algebra algorithms and frameworks.

B. Architecture

GPU extend computational parts – in particular ALU (Arithmetic Logic Unit) – of CPUs. CUDA's programming model allows us to take advantage of this heavy parallel architecture.

This work was done while Jérémy Jean was a member of the Pequan team at UPMC Univ Paris 06 and CNRS, UMR 7606, LIP6, 4 place Jussieu, F-75252, Paris cedex 05, France. Email: jean.jeremy@gmail.com

[1] see http://gpgpu.org/

C. Programming model

Kernels. C for CUDA extends C by allowing the programmer to define C functions, called *kernels*, that, when called, are executed n times in parallel by n different CUDA threads, as opposed to only once like regular C functions. Each of the threads that execute a kernel is given a unique thread ID that is accessible within the kernel and is used to access specific data needed in the parallelized sub-calculations.

Threads. The programming model imposes threads to be gathered in *blocks*, whose maximal size depends on the card. At the present time, size of blocks can not go further than 512 threads. Blocks of threads are as well grouped together in a *grid*, whose size depends on size of blocks and capabilities of the card.

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. 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. This needed independence of blocks goes along with a particular 3-level memory hierarchy.

Memory. CUDA threads may access data from multiple memory spaces during their execution. Each thread has a private local memory. Each thread block has a shared memory visible to all threads of the block and with the same lifetime as the block. Finally, all threads have access to the same global memory.

The second level of this particular hierarchy consists in the shared memory, shared by all threads of *one* block. This memory is expected to be much faster than global memory, so that global memory accesses should be avoided anytime they could be replaced by shared memory accesses.

Bank conflict. To achieve high memory bandwidth, shared memory is divided into equally-sized memory modules, called *banks*, which can be accessed simultaneously. So, any memory read or write request made of n addresses that fall in n distinct memory banks can be serviced simultaneously, yielding an effective bandwidth. However, if two addresses of a memory request fall in the same memory bank, there is a *bank conflict* and the access has to be serialized.

D. Performances

Since GPU does not execute code on the CPU, there is an extra cost in transferring data: the complexity of operations should justify the cost of moving data to the device. Code that transfers data for brief use by a small number of threads will see little or no performance lift. The ideal scenario is one in which many threads perform a substantial amount of work.

Performance benefits can be more readily achieved when the ratio of operations to elements transferred is important enough. For example, a matrix multiplication of $n \times n$ matrices requires n^3 operations (multiply-add), so the ratio of operations to element transferred is in $O(n)$, in which case the larger the matrix, the greater the performance benefit. Generally speaking, it is important to include transfers to and from the device in determining where operations should be performed.

In our particular case, we only compute a dot product on the GPU, that is a Level 1 BLAS. In that case, the ratio would advise us not to use GPU for that kind of computation, but rather do the calculation on the CPU to avoid time of transfers. However, Level 3 BLAS, which deals with matrix-matrix operations, relies heavily on dot product and with our parallel algorithms, we provide a means to perform Level 3 BLAS more efficiently. Consequently, the measured performances for our algorithms will not take transfer times into consideration since they are aimed to be used in higher level BLAS.

In the following of the article, we use floating-point representation for numbers. In the next section, we only present some basic results. See [2] for a more complete introduction.

III. Floating-point arithmetic

Let $p \geq 3$ be a prime number and consider the finite field $\mathbb{Z}/p\mathbb{Z}$. We use floating-point numbers in double precision to represent $\mathbb{Z}/p\mathbb{Z}$ integers. Denoting M the size of the double precision mantissa (53 bits according to the IEEE 754 standard), we limit p by

$$p - 1 < 2^{M-1}. \tag{1}$$

Any integer of the finite field could then be represented exactly by a floating-point number. The term $M-1$ is necessary rather than just M to be able to sum exactly at least two integers in the field without introducing a rounding error. In the sequel, we will assume the rounding mode to be directed *toward zero*. This is needed to ensure the error to be nonnegative in applications of error-free transformations (see III-A).

Notations. Throughout the paper, we assume to work with a floating point arithmetic adhering to IEEE 754 floating point standard in rounding toward zero [4]. We assume that no overflow nor underflow occur (this is always true since we only deal with integers that are less than 2^M). The set of floating point numbers is denoted by $\mathbb{F}$, We denote by $\mathbf{fl}(\cdot)$ the result of a floating point computation, where all operations inside parentheses are done in floating point working precision.

For $x \in \mathbb{F}$, $\mathbf{ufp}(x)$ will be the unit in the first place of x and $\mathbf{ulp}(x)$ the unit in the last place of x [7]. For $x \neq 0$, we have

$$\mathbf{ufp}(x) = 2^{\lfloor \log_2(x) \rfloor},$$

and $\mathbf{ulp}(x) = 2^{-M+1}\mathbf{ufp}(x)$. We will refer to machine precision as $\mathbf{u} = 2^{-M+1}$, because we chose rounding toward zero.

A. Error-free transformations

For $\circ \in \{+, -, \cdot, /\}$ an arithmetic operation, one can notice that $a \circ b \in \mathbb{R}$ and $\mathbf{fl}(a \circ b) \in \mathbb{F}$ but we usually do not have $a \circ b \in \mathbb{F}$. It is known that for the basic operations $+, -, \cdot$, the rounding error of a floating point operation, when performed in rounding to the nearest, is still a floating point number (see for example [8]):

$$\begin{array}{lcll} x = \mathbf{fl}(a \pm b) & \Rightarrow & a \pm b = x + y & \text{with } y \in \mathbb{F}, \\ x = \mathbf{fl}(a \cdot b) & \Rightarrow & a \cdot b = x + y & \text{with } y \in \mathbb{F}. \end{array} \tag{2}$$

These are *error-free* transformations of the pair (a, b) into the pair (x, y). Fortunately, the quantities x and y in (2) can be computed exactly in floating point arithmetic.

But this is no longer true in general when working in rounding toward zero (which is the case in our algorithms). For the multiplication, the rounding error is still a floating-point numbers (when no underflow) in rounding toward zero. Nevertheless, for addition, it is not true but when working only with non-negative numbers, it is true.

For computing the rounding error of a multiplication, we will use a Fused-Multiply-and-Add (`FMA`) operator [9]. Some computers have a *Fused-Multiply-and-Add* (`FMA`) operation that enables a floating point multiplication followed by an addition to be performed as a single floating point operation. The Intel IA-64 architecture, implemented in the Intel Itanium processor, has an `FMA` instruction as well as the IBM RS/6000 and the PowerPC before it and as the new Cell processor [10]. The Intel Haswell architecture, scheduled for release in 2012, will come with a FMA unit as well. Some recent GPU also have a FMA unit. On the Itanium processor, the `FMA` instruction enables a multiplication and an addition to be performed in the same number of cycles than one multiplication or one addition. As a result, it seems to be advantageous for speed as well as for accuracy.

The following algorithm applies when computing a product of two positive floating-point numbers. We make use of the FMA to evaluate *exactly* the round-off term of the floating-point product. This operation had been included in the IEEE 754 standard in 2008 and performs

$$\text{FMA}(a, b, c) = a \times b + c$$

with only one rounding.

Algorithm 1 — TwoProduct

```
Require: $a, b \in \mathbb{F}$ such that $a, b \geq 0$
Ensure: $x \in \mathbb{F}$ and $y \in \mathbb{F}$ such that $ab = x + y$
  $x \leftarrow \mathbf{fl}(ab)$
  $y \leftarrow \text{FMA}(a, b, -x)$
  return $(x, y)$
```

Theorem 1. *([2]) Let x and y be the result of* ***TwoProduct*** *applied to a and b.*

We have: $ab = x + y$, $x = \boldsymbol{fl}(ab)$, $0 \leq x \leq ab$, $0 \leq y < \boldsymbol{u}.\boldsymbol{ufp}(x)$ and $0 \leq y < \boldsymbol{u}x$.

We now present an other error-free transformation related to Euclidean division by a power of two. Suggested in [11], quoted in [5] by S. Rump and already discussed in [2], this algorithm splits a floating-point number into two non-overlapping others.

Theorem 2. *([2]) Let x and y be the result of* ***ExtractScalar*** *applied to $a \in \mathbb{N} \cap \mathbb{F}$ and $\sigma \in \mathbb{F}, \sigma = 2^k, k \geq M$. We have:*

$$a = x + y, \qquad 0 \leq y < \boldsymbol{u}\,\sigma, \quad 0 \leq x \leq a, \quad x \in \boldsymbol{u}\sigma\mathbb{N}.$$

The idea behind this splitting method is to use the rounding mechanism of the floating-point unit. Set to be toward zero, the rounding behaves the same way as a *truncation*. In terms of bits, the M-bit string a is divided in two strings s_1 and s_2 which do not overlap such that the concatenation $s_1 + s_2$ equals a. As subparts of a, both bit-strings s_1 and s_2 are in $\mathbb{F}$.

Algorithm 2 — ExtractScalar

```
Require: $a \in \mathbb{N} \cap \mathbb{F}$, and $\sigma = 2^k, k \in \mathbb{N}, \sigma \geq a$
Ensure: $x \in \mathbb{N} \cap \mathbb{F}$, $y \in \mathbb{N} \cap \mathbb{F}$ such that $a = x + y$
  $q \leftarrow \mathbf{fl}(\sigma + a)$
  $x \leftarrow \mathbf{fl}(q - \sigma)$
  $y \leftarrow \mathbf{fl}(a - x)$
  return $(x, y)$
```

IV. Dot product algorithms

In the following, we will assume the size of input vectors to be a power of two: $N = 2^k$. Since reduction algorithm is based on a binary tree concept, it is easier to describe in that case. For generalization to any N, we would just pad with zeros to reach the next power of two.

A. Naive algorithm

To present the basic concept of the dot product parallelized algorithm in CUDA, we start by describing the naive implementation of the algorithm.

The parallelization is achieved in two steps. First, the construction of a third vector c, which equals the vector-vector product of a and b. Second, the sum of all elements of this new vector. This sum modulo p is then the desired result.

$$\forall i \in [1, N],\; c_i = a_i\, b_i, \qquad a \cdot b = \sum_{i=1}^{N} c_i \pmod{p}.$$

In CUDA, this mechanism will be done with two kernels: one for the vector-vector product, an other kernel for the sum. The N products are distributed over the blocks of threads, constructing vector c. Sizes for the grid and blocks depends on the value of N. As mentioned before, thread blocks can be no greater 512, which means that one block can not have more than 512 threads working in parallel. Consequently, the total number of threads is $t = \min(n, 512)$ and these threads are split over $b = n/t$ blocks. With such a repartition, there will be $\dfrac{n}{b \times t}$ products done in each thread.

Once the vector-vector product c is known, we want to sum all its elements c_i to get the dot product $a \cdot b$. This is a common routine in parallel computing called a *reduction* (see [12], [13]). The first natural idea for this reduction consists in calculating partial sums of adjacent elements, striding across partial sums incrementally to finally get the final one (see Figure 1).

The problem of the summation in this order of calculation is bank conflicts. As detailed before, accesses to adjacent values by different threads of the same block will cause lots of bank conflicts. To prevent this to happen, we sum elements in a

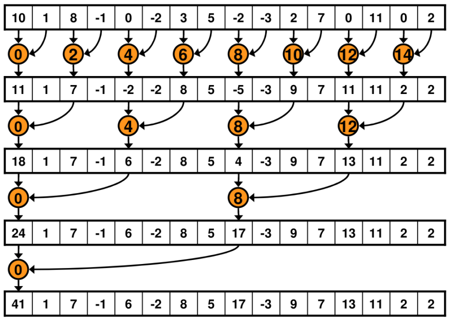


Fig. 1. Naive reduction leading to bank conflicts (indexes: 1 → 2 → 4 → 8).

different order, with strided indexing going down (see Figure 2).

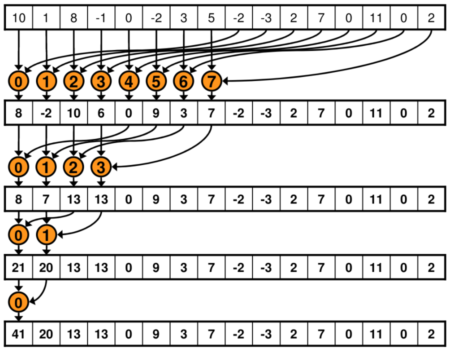


Fig. 2. Conflict-free reduction (indexes: 8 → 4 → 2 → 1).

Figures 1 and 2 are borrowed from [12].

B. λ-algorithm

We assume now that there exists $\lambda \in \mathbb{N}$ such that $\lambda(p-1) < 2^{M-1}$. As detailed in [2], we get the following result:

Theorem 3. *([2]) Strengthening the hypothesis (1) on p and assuming there exists $\lambda \in \mathbb{N}$, $\lambda \geq 1$ such that:*

$$\lambda(p-1) < 2^{M-1}, \tag{3}$$

there exists an algorithm computing the dot product of two vectors of $\mathbb{Z}/p\mathbb{Z}$ of size N using only N/λ reductions in $\mathbb{Z}/p\mathbb{Z}$. We will refer to this algorithm as λ-algorithm.

As detailed in [2] for the serial algorithm, we need to split the product $a_i\, b_i$, so that the dot product $a \cdot b$ can be expressed as follows:

$$a \cdot b = \sum_{i=1}^{N} a_i\, b_i = 2^{l+1} \sum_{n_\alpha} (\alpha_i 2^{-(l+1)} - 2^l) + \\ 2^{l+1} \sum_{N-n_\alpha} \alpha_i 2^{-(l+1)} + \sum_{n_\beta} (\beta_i - 2^l) + \\ \sum_{N-n_\beta} \beta_i + \sum_{N} r_i + \underbrace{\left(2^{l+1} n_\alpha + n_\beta\right) 2^l}_{correction}.$$

For the definition of $\alpha_i, \beta_i, l, n_\alpha, n_\beta$, we refer to [2]. In the parallel algorithm then, the previous result c of the vector-vector product $a \times b$ (multiplications are performed componentwise) is then replaced by a four-vector result: $(\alpha, \beta, r, corr)$. Components of vectors α and β are either $\alpha_i 2^{-(l+1)}$, β_i, $\alpha_i 2^{-(l+1)} - 2^l$ or $\beta_i - 2^l$, so that finally, each vector only have field elements of $\mathbb{Z}/p\mathbb{Z}$ (details in the proof in [2]).

This first step done, we now need to sum elements together to get the dot product. Under the assumption $\lambda(p-1) < 2^{M-1}$, we sum each of four vectors α, β, r and $corr$ in parallel, by reducing modulo p anytime the partial sums accumulated may goes over λ values. By choosing N a power of two, this means that any time the number s of accumulated values in the partial sum is such that $2s > \lambda$, we need a reduction. If we do not, the next step of reduction would cause a rounding error and we would loose information.

We measured performances for this algorithm and present results below in Section V.

C. $(\alpha, \beta, \gamma, \delta)$-algorithm

1) General concept: This section presents a new algorithm, which almost leads to a reduction-free dot product algorithm. In the previous section, we added a hypothesis on p with the λ parameter. However, dot product over finite field takes two different parameters: the prime p and the size N of input vectors. We previously set the hypothesis on p. We now only assume (1) : $p - 1 < 2^{M-1}$, but we limit N to

$$N \leq 2^{s_1} \qquad \text{where } s_1 = \left\lfloor \frac{M}{2} \right\rfloor \quad \text{and} \quad s_2 = \left\lceil \frac{M}{2} \right\rceil. \tag{4}$$

This is done with a few more error-free transformations based on **ExtractScalar** and the basic idea of this method is splitting the product $a_i\, b_i$ into four pieces of at most $M/2$ bits each:

$$a_i\, b_i = \alpha_i + \beta_i + \gamma_i + \delta_i.$$

With the results of **ExtractScalar**, one can rewrite the same value:

$$a_i\, b_i = \alpha_i'\, 2^{3s} + \beta_i'\, 2^{2s} + \gamma_i'\, 2^{s} + \delta_i' \quad \text{with} \quad 0 \leq \alpha_i', \beta_i', \gamma_i', \delta_i' < 2^s.$$

With the condition (4) on the size N of the vectors, this means one can sum the whole vectors $\alpha, \beta, \gamma, \delta$ without rounding errors:

$$\forall v' \in \{\alpha', \beta', \gamma', \delta'\}, \quad \sum_{i=1}^{N} v_i' \leq \sum_{i=1}^{N} 2^s \leq N\, 2^s \leq 2^s 2^s \leq 2^M.$$

Implemented algorithm (3) is detailed in Appendix.

2) Proof of the sequential algorithm: Let s_3, s_4, s_5 be:

$$s_3 = 2s_1 + s_2 = M + s_1, \\ s_4 = 2(s_1 + s_2) = 2M, \\ s_5 = 2(s_1 + s_2) + s_1 = 2M + s_1.$$

Let $i \in \mathbb{N}$, such that $1 \leq i \leq N$. **TwoProduct** on a_i and b_i leads to $a_i\, b_i = h_i + r_i$, with:

$$h_i = \mathbf{fl}(a_i\, b_i), \quad 0 \leq h_i \leq a_i\, b_i, \quad 0 \leq r_i < \mathbf{u}.\mathbf{ufp}(h_i) \leq \mathbf{u} h_i.$$

The error-free transformation **ExtractScalar** on h_i with parameter $\sigma = 2^{s_5-1}$ gives α_i and β_i' as results, with:

$$\begin{array}{ll} h_i = \alpha_i + \beta_i', & 0 \le \beta_i' < 2^{M+s_1}, \\ 0 \le \alpha_i \le h_i, & \alpha_i \in 2^{M+s_1}\mathbb{N}. \end{array} \tag{5}$$

Case 1: $\alpha > 0$ (see Figure 3)

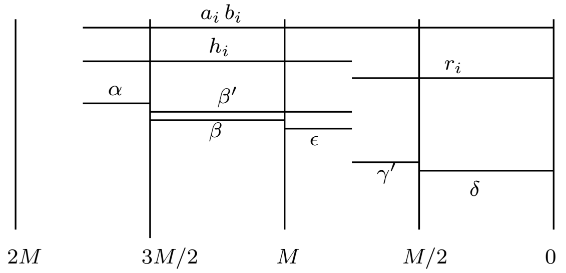


Fig. 3. Splitting of $a_i\,b_i$ in the general case ($\alpha > 0$)

In this case, we have $\textbf{ufp}(h_i) > 2^{s_3}$ so that $\textbf{ulp}(h_i) > 2^{s_1}$. Once we split h_i into two pieces $\alpha_i + \beta_i'$, one can split β_i' into two others. This is done by calling **ExtractScalar** on β_i' with parameter $\sigma = 2^{s_4-1}$. This results in:

$$\begin{array}{ll} \beta_i' = \beta_i + \epsilon_i, & 0 \le \epsilon_i < 2^{M}, \\ 0 \le \beta_i \le \beta_i', & \beta_i \in 2^{M}\mathbb{N}. \end{array} \tag{6}$$

Again, **ExtractScalar** on error term during the multiplication r_i with parameter $\sigma = 2^{s_3-1}$ gives the decomposition:

$$\begin{array}{ll} r_i = \gamma_i' + \delta_i, & 0 \le \delta_i < 2^{s_1}, \\ 0 \le \gamma_i' \le r_i, & \gamma_i' \in 2^{s_1}\mathbb{N} \end{array} \tag{7}$$

All in all, one has:

$$\begin{array}{rcl} a_i\,b_i & = & h_i + r_i \\ & = & \alpha_i + \beta_i' + r_i \\ & = & \alpha_i + \beta_i + \epsilon_i + r_i \\ & = & \alpha_i + \beta_i + \epsilon_i + \gamma_i' + \delta_i \end{array}$$

With (5), (6) and (7), we have $\textbf{ulp}(h_i) = \textbf{ulp}(\alpha_i + \beta_i + \epsilon_i) = \textbf{ulp}(\epsilon_i)$ because $\alpha_i > \beta_i > \epsilon_i$. Thus, $\textbf{ulp}(\epsilon_i) > 2^{s_1}$. Moreover, $\epsilon_i < 2^{s_2}$ so bits of ϵ_i are localized between 2^{s_1} and 2^{s_2}.

As for γ_i', it is the same. Either $\gamma_i' > 0$ in which case bits of γ_i' are localized in the same interval as ϵ_i, either $\gamma_i' = 0$ and there will have no problem in summation. Thus, one can say γ_i' and ϵ_i are in the same quarter, so one sums them together: $\gamma_i = \gamma_i' + \epsilon_i$. Finally:

$$a_i\,b_i = \alpha_i + \beta_i + \gamma_i + \delta_i,$$

and (5), (6), (7) give:

$$\alpha_i = A_i\,2^{M+s_1}, \quad \beta_i = B_i\,2^{M}, \quad \gamma_i = C_i\,2^{s_1},$$

for some $A_i, B_i, C_i \in [0, 2^{s_1}]$, so that:

$$\forall i \in [1, N], \quad a_i\,b_i = A_i\,2^{M+s_1} + B_i\,2^{M} + C_i\,2^{s_1} + \delta_i.$$

This means that the dot product $a \cdot b$ in $\mathbb{Z}/p\mathbb{Z}$ equals:

$$2^{M+s_1}\sum_{i=1}^{N} A_i + 2^{M}\sum_{i=1}^{N} B_i + 2^{s_1}\sum_{i=1}^{N} C_i + \sum_{i=1}^{N} \delta_i \pmod{p} \tag{8}$$

can be calculated by summing the four vectors and then reduced modulo p to get the final result. Each of the four sums can be done in floating-point arithmetic without any reductions modulo p by the consideration (4) on the size N.

Case 2.1: $\beta > 0$ (see Figure 4)

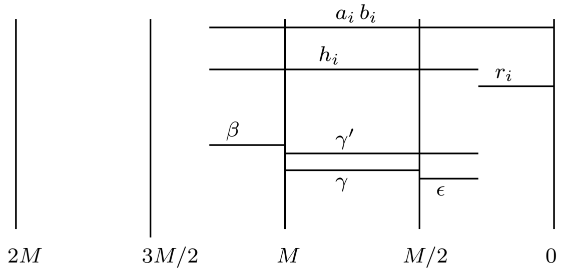


Fig. 4. Splitting of $a_i\,b_i$ when $\alpha = 0$ and $\beta > 0$.

In this case, $2^{s_2} < \textbf{ufp}(h_i) < s^{s_3}$, so we need to cut h_i with a new parameter σ, smaller than 2^{s_5-1}. Discarding the previous result of the splitting, one gets a new one with **ExtractScalar** on h_i with $\sigma = 2^{s_4-1}$ leading to:

$$\begin{array}{ll} h_i = \beta_i + \gamma_i', & 0 \le \gamma_i' < 2^{M} \\ 0 \le \beta_i \le h_i, & \beta_i \in 2^{M}\mathbb{N}. \end{array} \tag{9}$$

To get the full decomposition in four quarters, one needs to apply **ExtractScalar** on γ_i' with $\sigma = 2^{s_3-1}$, so we have:

$$\begin{array}{ll} \gamma_i' = \gamma_i + \epsilon_i, & 0 \le \epsilon_i < 2^{s_1}, \\ 0 \le \gamma_i \le h_i, & \gamma_i \in 2^{s_1}\mathbb{N}. \end{array} \tag{10}$$

This last equation ends the process of splitting and finally, with two applications of **ExtractScalar**, we have:

$$\begin{array}{rcl} a_i\,b_i & = & h_i + r_i \\ & = & \beta_i + \gamma_i' + r_i \\ & = & \beta_i + \gamma_i + \epsilon_i + r_i. \end{array}$$

Because in this case, we had $2^{s_2} < \textbf{ufp}(h_i) < s^{s_3}$, the remainder r_i is such that $r_i \le \mathbf{u}h_i < \mathbf{u}s^{s_3} = 2^{s_1}$. So that both ϵ_i and r_i are in $[0, 2^{s_1}]$. We define $\delta_i = \epsilon_i + r_i \in [0, 2^{s_1}]$, and then:

$$a_i\,b_i = \beta_i + \gamma_i + \delta_i. \tag{11}$$

This last statement (11) is similar as (8) in the general case, except that $A_i = 0$.

Case 2.2: $\beta = 0$ (see Figure 5).

Here, we even have $\textbf{ufp}(h_i) < 2^{s_2}$. This means that all significative bits of h_i are between 0 and M. The result of the multiplication $a_i\,b_i$ did not go over the mantissa, so that $r_i = 0$

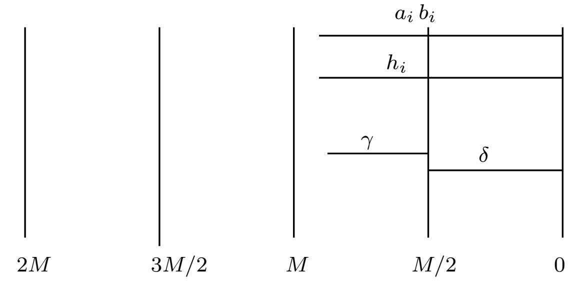


Fig. 5. Splitting of $a_i\, b_i$ when $\alpha = 0$ and $\beta = 0$.

and one just has to split h_i in two parts with **ExtractScalar** and $\sigma = 2^{s_3 - 1}$:

$$\begin{aligned} h_i &= \gamma_i + \delta_i, & 0 \le \delta_i &< 2^{s_1} \\ 0 \le \gamma_i &\le h_i, & \gamma_i &\in 2^{s_1}\mathbb{N}. \end{aligned} \qquad (12)$$

Finally in this case:

$$a_i\, b_i = h_i = \gamma_i + \delta_i$$

We have the same relation as (8) with both A_i and B_i equal to zero.

All in all, we get four column-vectors of N scalars with at most $M/2$ bits each. With the hypothesis (4) of the size of input vectors, one can sum those four vectors exactly (see Figure 6). Hence, each four sums is stored exactly in one floating-point number, which can then be reduced modulo p to get the final result.

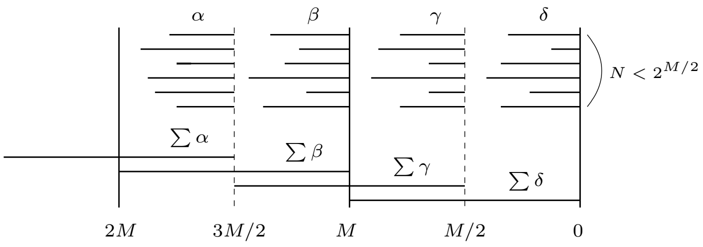


Fig. 6. One sums up the four vectors

3) Parallel version: In the sequential algorithm, we proceed incrementally on each couple of elements (a_i, b_i). Rather than considering the problem line by line, the parallel version performs operations on vectors – *i.e.* columns – in two steps: firstly, the decomposition of the vector-vector product $a \times b$ into the vector sum $\alpha + \beta + \gamma + \delta$ and secondly, the summation of the four vectors.

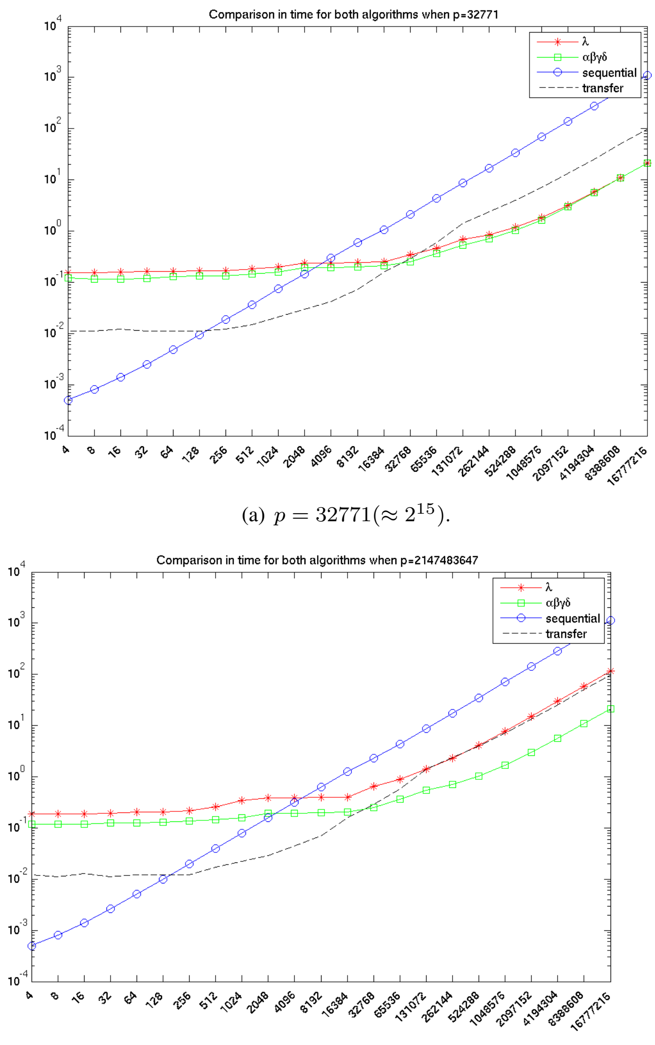
$$\begin{pmatrix} a_1 \\ \vdots \\ a_N \end{pmatrix} \times \begin{pmatrix} b_1 \\ \vdots \\ b_N \end{pmatrix} = \underbrace{\begin{pmatrix} \alpha_1 \\ \vdots \\ \alpha_N \end{pmatrix}}_{\sum \alpha_i} + \underbrace{\begin{pmatrix} \beta_1 \\ \vdots \\ \beta_N \end{pmatrix}}_{\sum \beta_i} + \underbrace{\begin{pmatrix} \gamma_1 \\ \vdots \\ \gamma_N \end{pmatrix}}_{\sum \gamma_i} + \underbrace{\begin{pmatrix} \delta_1 \\ \vdots \\ \delta_N \end{pmatrix}}_{\sum \delta_i}$$

The main interest in this algorithm lies in the absence of reductions in the summation of the four vectors. From that, the reduction algorithm to get the sums happens to be really efficient and provide the *exact* result.

At the end of the parallel calculation, the four floating-point sums will lead to the final result in $\mathbb{Z}/p\mathbb{Z}$ after 7 reductions and 3 sums, under the assumption $p - 1 < 2^{M-1}$.

V. Experimental results

The environment used to evaluate these algorithms is an Intel Core 2 Quad Processor Q8200 2.33GHz, which accesses a NVIDIA Tesla C1060 computing processor.



(a) $p = 32771 (\approx 2^{15})$.

(b) $p = 2147483647 (\approx 2^{31})$.

Fig. 7. Timing comparisons of λ- and $(\alpha, \beta, \gamma, \delta)$-algorithms for two different primes p.

On Figure 7, we represent independently transfer and purely computational timings. We plotted timings (in ms) for sizes $N = 2^k, k \in [2, 24]$ on a log-log scale. The serial multi-precision algorithm (blue) runs on CPU in linear time of N.

As for the λ-algorithm implementation on GPU (red), extra cost of the CUDA layer – thread synchronizations and block repartition for instance – makes the benefit of this implementation positive only for vectors greater than 4096 elements. Timing results for this implementation can be divided into two main parts, regarding the value of N: either N is smaller than 2^{14} and the time is almost constant, either N is greater than this threshold value, and the computation is done in linear time of N.

In the first half, constant time is due to CUDA threads: for our GPU architecture, there is enough device material to take care of all input elements at the same time so that there is no extra cost for different sizes up to 2^{14}. In the second half, a lack of resource introduces latency in computations and we reach the expected linear behavior of the dot product complexity. Asymptotically, all implementations behave the same way, with a speed-up bigger than 10 for the GPU one.

Finally, implementation for the $(\alpha, \beta, \gamma, \delta)$-algorithm provides timings of the same kind: almost constant for $N = 2^k$ up to 2^{14}, linear in N otherwise. However, there is a fundamental difference: timings for the $(\alpha, \beta, \gamma, \delta)$-algorithm do not depend on p. In the first case of the λ-algorithm, via the value of λ, we had to do more and more operations – namely, reductions – as p increased. With the splitting algorithm, by imposing a maximal size $N_{max} = 2^{M/2}$ for the input vectors, those reductions are useless (see Section IV-C).

Asymptotically, for $N > 10000$, we reach good speedups for both algorithms: 10 for λ-algorithm and more than 40 for $(\alpha, \beta, \gamma, \delta)$-algorithm.

VI. Conclusion and future work

In this paper, we suggested two parallel version of algorithms to compute the dot product in a finite field using floating-point arithmetic. Both algorithms have been designed to reduce the cost of modular reductions, which happen to be the slow operation of the process.

In the first algorithm, we generalize the idea introduced in [1] where it sums integers – packed into floating-point numbers – by packets. In our work, we use results on error-free transformations to extend the range of representable integers to make them fit into a double precision floating-point mantissa. The second method revealed in this paper describes a prior treatment of input vectors which enables summations to be done without any modular reductions. Based as well on error-free transformations, this particular way of summation results in good experimental performances.

Parallelized implementations of these algorithms behave almost the same way, depending mainly on the size of input vectors. Concerning the λ-algorithm, time of computation increases with p: for large finite fields, the value of λ is quite small, which leads to many reductions. This has a significant impact of timings. As for the $(\alpha, \beta, \gamma, \delta)$-algorithm, there is no dependence on p.

Consequently, for big input vectors, one can reach really good speedups: more than 40 for the splitting $(\alpha, \beta, \gamma, \delta)$-algorithm but still and all 10 for the λ one.

In a future work, it would interesting to investigage on RNS algorithms for GPU. A comparison between an integer RNS implementation and a floating-point RNS version could be very useful. Moreover, NVIDIA has launched a new graphic card called Fermi. Even if current GPU are said to be very efficient for floating-point computations, double precision is relatively slow compared to single precision. But Fermi GPU are believed to be 400% faster than previous NVIDIA GPU in double-precision floating-point operations. Integrating our algorithms in level 3 BLAS routines would make it possible to really compare the efficiency of our algorithms with other implementations.

We also plan to implement our algorithms in OpenCL and to compare them on different type of GPU (Nvidia, AMD-ATI).

References

[1] J.-G. Dumas, “Efficient dot product over word-size finite fields,” in *Proceedings of the 7th International Workshop on Computer Algebra in Scientific Computing, CASC’2004 (St. Petersburg, Russia, July 12-19, 2004)*, V. G. Ganzha, E. W. Mayr, and E. V. Vorozhtsov, Eds. Garching: Institut für Informatik, Technische Universität München, 2004, pp. 139–153.

[2] J. Jean and S. Graillat, “Fast dot product over finite field,” Research Report hal-00450888, 2010, available at http://hal.archives-ouvertes.fr/hal-00450888/en/.

[3] N. Yamanaka, T. Ogita, S. Rump, and S. Oishi, “A parallel algorithm for accurate dot product,” *Parallel Computing*, vol. 34, no. 6-8, pp. 392 – 410, 2008, parallel Matrix Algorithms and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/B6V12-4S33N13-1/2/7ce8ecd69522e2aab30d42cfc221d061

[4] “IEEE standard for floating-point arithmetic,” Tech. Rep., 2008. [Online]. Available: http://dx.doi.org/10.1109/IEEESTD.2008.4610935

[5] S. M. Rump, T. Ogita, and S. Oishi, “Accurate floating-point summation part I: Faithful rounding,” *SIAM J. Sci. Comput.*, vol. 31, no. 1, pp. 189–224, 2008.

[6] NVIDIA, *NVIDIA CUDA Programming Guide 2.0*, 2008.

[7] J.-M. Muller, “On the definition of ulp(x),” École normale supérieure de Lyon - Laboratoire de l’Informatique du Parallélisme, Tech. Rep., 2005.

[8] T. J. Dekker, “A floating-point technique for extending the available precision,” *Numer. Math.*, vol. 18, pp. 224–242, 1971.

[9] Y. Nievergelt, “Scalar fused multiply-add instructions produce floating-point matrix arithmetic provably accurate to the penultimate digit,” *ACM Trans. Math. Software*, vol. 29, no. 1, pp. 27–48, 2003.

[10] C. Jacobi, H.-J. Oh, K. D. Tran, S. R. Cottier, B. W. Michael, H. Nishikawa, Y. Totsuka, T. Namatame, and N. Yano, “The vector floating-point unit in a synergistic processor element of a Cell processor,” in *ARITH ’05: Proceedings of the 17th IEEE Symposium on Computer Arithmetic*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 59–67.

[11] C. Hecker, “Let’s get to the (floating) point,” *Game Developer Magazine*, 1996.

[12] M. Harris, “Optimizing parallel reduction in CUDA,” Nvidia, Tech. Rep., 2007, available at http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reduction.pdf.

[13] ——, “Parallel prefix sum (scan) with CUDA,” Nvidia, Tech. Rep., 2008, available at http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/scan/doc/scan.pdf.

Appendix

Algorithm 3 — Dot product computation without any reduction in the main loop

```
Require: $p \geq$ a prime, $a$ and $b$ two $\mathbb{Z}/p\mathbb{Z}$ vectors of size $N < 2^{s_1}$
Ensure: The dot product $a \cdot b$ of vectors $a$ and $b$ in $\mathbb{Z}/p\mathbb{Z}$.

  $A \leftarrow 0$
  $B \leftarrow 0$
  $C \leftarrow 0$
  $D \leftarrow 0$
  for $i = 1$ to $N$ do
    $[h, r] \leftarrow$ TwoProduct$(a_i, b_i)$
    $[\alpha, \beta] \leftarrow$ ExtractScalar$(2^{s_5-1}, h)$
    if $\alpha = 0$ then
      if $\beta = 0$ then
        $[\gamma, \delta] \leftarrow$ ExtractScalar$(2^{s_3-1}, h)$
      else
        $[\beta, \gamma] \leftarrow$ ExtractScalar$(2^{s_4-1}, h)$
        $[\gamma, \epsilon] \leftarrow$ ExtractScalar$(2^{s_3-1}, \gamma)$
        $\delta \leftarrow r + \epsilon$
      end if
    else
      $[\beta, \epsilon] \leftarrow$ ExtractScalar$(2^{s_4-1}, \beta)$
      $[\gamma, \delta] \leftarrow$ ExtractScalar$(2^{s_3-1}, r)$
      $\gamma \leftarrow \gamma + \epsilon$
    end if
    $A \leftarrow A + \alpha$
    $B \leftarrow B + \beta$
    $C \leftarrow C + \gamma$
    $D \leftarrow D + \delta$
  end for
  $A \leftarrow A \pmod p$
  $B \leftarrow B \pmod p$
  $C \leftarrow C \pmod p$
  $D \leftarrow D \pmod p$
  $A \leftarrow A + B$
  $C \leftarrow C + D$
  $A \leftarrow A \pmod p$
  $C \leftarrow C \pmod p$
  $A \leftarrow A + C$
  $res \leftarrow A \pmod p$
```