Comparison of Modular Arithmetic Algorithms on GPUs

Pascal GIORGI [a], Thomas IZARD [a] and Arnaud TISSERAND [b]
[a] *LIRMM, CNRS – Univ. Montpellier 2, 161 rue Ada, F-34392 Montpellier, France*
[b] *IRISA, CNRS – INRIA Centre Rennes Bretagne Atlantique – Univ. Rennes 1, 6 rue Kérampont, F-22305 Lannion, France*

Abstract. We present below our first implementation results on a modular arithmetic library for cryptography on GPUs. Our library, in C++ for CUDA, provides modular arithmetic, finite field arithmetic and some ECC support. Several algorithms and memory coding styles have been compared: local, shared and register. For moderate sizes, we report up to 2.6 speedup compared to state-of-the-art library.

Keywords. Multiple precision arithmetic, mathematical library, CUDA.

Introduction

Modular operations on large integers are used in many applications such as cryptography, coding and computer algebra. Efficient algorithms and implementations are required for $a \pm b \bmod p$, $a \times b \bmod p$ where a, b and p are multiple precision integers and p is prime. Those operations are required in finite field arithmetic over $\mathbb{F}_p$ and in elliptic curve cryptography (ECC) where sizes are about 200–600 bits. Graphic processor units (GPUs) are used in high-performance computing systems thanks to their massively multithreaded architectures. But due to their specific architecture and programming style, porting libraries to GPUs is not simple even using high-level tools such as CUDA [8].

This paper presents our first implementation results on modular arithmetic for large integers, arithmetic over $\mathbb{F}_p$ and ECC scalar multiplication application on GPUs. This work is a part of a software library called PACE [5]. This library is aimed at providing a very large set of mathematical objects, functions and algorithms to facilitate the writing of arithmetic applications. Below, we only deal with the use of GPUs as accelerators for parallel computations with different sets of data. In asymmetric cryptography applications, this kind of parallelism level is required for servers on which cyphering and/or digital signatures are computed for parallel sessions. This paper presents sequential arithmetic operations (one thread per operation) but for massively parallel computations on independent data sets. We compare our code with $\mathtt{mp}\mathbb{F}_q$ library[3] and the work from [9].

1. Large Integers on GPU

We consider modular arithmetic for arbitrary values of the modulo (i.e. p) and size N of numbers in the range 160–384 bits. The operands a and b are integers in the range $[0, p-1]$ (reduced values). Standard modular arithmetic algorithms and some implementation

guidelines on standard processors may be found in good books such as [4]. But this is not the case for GPU implementation. The design and optimization of arithmetic operations is done using a complex trade-off between: *number representation* (width, radix), the *algorithm(s)* and some *architecture constraints* (type and number of functional units and memory latencies). Our goal is to define an efficient layer which can be used on a single thread, since our parallelism is only on the data. We will see that the memory mapping of the integers into various memories is a key element for this kind of GPU implementation.

1.1. Integers Representation and Memory

Large integers are usually stored into an array of words such as depicted below.

$$a = \boxed{a_{n-1}}\boxed{a_{n-2}}\boxed{\ldots}\boxed{a_0} = \sum_{i=0}^{n=-1} a_i \beta^i$$

The word size depends on the functional units characteristics. This size leads to various values for the radix β. The choice of β and the word's datatype is clearly related to GPUs capabilities. According to NVIDIA CUDA programming guide [8], the possible native datatype for a word can be either 32-bit integer, 32-bit or 64-bit floating-point (FP) numbers. Considering current lower[1] throughputs of the 64-bit FP numbers compare to 32-bit datatype on newest GPU (GT200 core), one can avoid for the time being the use of 64-bit FP number as word.

Considering 32-bit arithmetic units on GPU, one can issue one of the following operations in 4 clock cycles on a GPU multiprocessor: one exact multiply-and-add (MAD) operation on 10-bit operands with `float`, one exact multiplication on 16-bit operands with `int` and one exact addition with carry on 16-bit operands with `int`.

Looking at the school book multiplication[2] on N-bit integers this gives the following theoretical clock cycle counts:

	160-bit	192-bit	224-bit	256-bit	384-bit
# cycle with float, $\beta = 2^{10}$	1024	1600	2116	2704	6084
# cycle with int, $\beta = 2^{16}$	724	1252	1460	1924	4420

It seems clear from this comparison that 32-bit integers with $\beta = 2^{16}$ is a better choice than 32-bit FP numbers. Moreover, a smaller word number saves memory. Furthermore, bit manipulations remain easier with integer representation.

The main difficulty with GPU is to fully benefit from high memory bandwidth within the global memory (the RAM of the GPU). In particular, one needs to design code such that many concurrent memory accesses from many concurrent threads of a multiprocessor can be coalesced into a single transaction. The basic idea is that the i-*th* thread of a multiprocessor needs to read/write the i-*th* data in a particular segment of memory. See [8, Chapter 5] for further details.

In our computational model, integers are not gathered between threads and using linear array to store integer's words leads to non-coalesced memory patterns. To fulfill this requirement, we provide functionality to load (resp. store) integer from (resp. to) global memory. These functions change the words order to ensure coalesced access. We define the following rule to reorder words of a same variable across all threads:

[1] Throughput of 64-bit float is barely equal to 1/8 of 32-bit float throughput.

[2] School book multiplication requires N^2 multiplications and $(N-1)^2$ additions or N^2 MAD

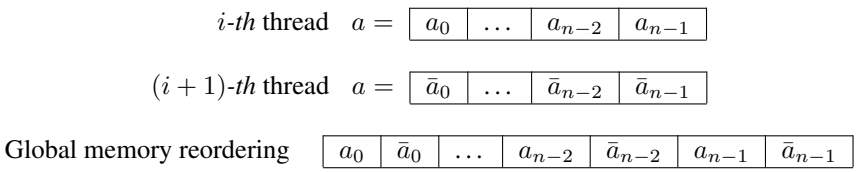

Figure 1. Global memory organization for coalesced integer access

To ensure memory alignment, two consecutive words of one integer of one thread are distant of i word addresses, where i is the minimal power of 16 greater or equal to the total number of threads launched on the GPU.

Global memory accesses are quite costly and must be avoided. Therefore, one would prefer to store integers in a non-coalesced form in another memory region (local memory or shared memory). Fetching all data in local or shared memory at the beginning of a GPU kernel function will ensure that all reads from global memory are coalesced.

1.2. Large Integer in GPUs Registers

For efficiency reason it would be interesting to map large integers directly into GPU registers. This should save at least the latency of memory load/store instructions. However, as all hardware registers, GPU's registers are not indexable. This means that no arrays can be mapped into registers. The only way to map large integers into registers is to design specific structures with one variable per integer word. This approach is feasible but not used for generic and portable code.

Fortunately, C++ and template metaprogramming [1] can help. In particular, one can design a generic structure which defines fixed number of variables with compile time indexable access.

```
template <uint N> struct IntegerReg : IntegerReg<N-1> {
  uint word;
  IntegerReg() : IntegerReg<N-1>() {word=0;}};
template<> struct IntegerReg<1> {
  uint word;
  IntegerReg()  {word=0;}};
#define AT(x,i)  x.IntegerReg<i+1>::word
```

Figure 2. A register-compliant structure for large integer with indexable access.

`IntegerReg` structure defined in Figure 2 uses C++ recursive inheritance to define a proper number of variables to store an N-word integer. One can access a particular variable from this structure by using the index given within the template parameter. The macro `AT` defines the proper accessor to the (i+1)-*th* word of the integer x. The only drawback of this structure is that all indices need to be known at compile time, meaning no runtime loop can be executed. Here again C++ templates come to rescue to define compile time loop as illustrated in Figure 3.

Compile time loop consists of completely unrolling loops by duplicating code with proper register values. This technique can be very efficient if the compiler is able to discover registers reusability. In case of GPU, `nvcc` compiler (the CUDA compiler)

```
template<uint beg, uint end>
struct IntegerLoop {
  template<uint N>
  static bool egal(const IntegerReg<N>& a, const IntegerReg<N>& b)
  {return  AT(a,beg)==AT(b,beg) && IntegerLoop<beg+1,end>::egal(a,b);}};
template<uint end>
struct IntegerLoop<end,end> {
  template<uint N>
  static bool egal (const IntegerReg<N>& a, const IntegerReg<N>& b)
  {return AT(a,end)==AT(b,end);}};
```

Figure 3. C++ compile time loop for register-compliant integers comparison.

seems to not be very friendly with such a technique. In particular, with quite simple code `nvcc` might ran out of registers when it tries to compile code for the GPU. This comes from GPUs code generation which uses intermediate PTX code in SSA form (Single Static Assignement) before to call `open64` compiler[3] to generate code for GPU. The SSA forms of compile time loops are usually quite large and discovering registers mapping from this form is not well handled by `nvcc`. To reduce SSA code and register usage, we change the base β value to 2^{32} whenever register-compliant integers are used.

2. Modular Arithmetic on Large Integers

We investigate GPU implementation of the 3 basic modular arithmetic operations on large integers: addition, subtraction and multiplication. We do not investigate division since it can be avoided in most of our target applications. Memory accesses within GPU can be very costly depending where data are localized. For instance one access to global memory costs around 400–600 cycles while one register or one shared memory access cost around 10 cycles. Minimizing data access and intermediate variables is thus critical.

For the modular addition/subtraction the basic algorithm can suffice since it requires only an integer addition (potentially with a carry) and a conditional subtraction. Moreover, this operation can be done inplace (`a += b mod p`) with only one extra register for the carry propagation. We use classical integer algorithm with carry propagation along limbs as described at [4, pp.30].

In the case of register version of integers, we have $\beta = 2^{32}$ and no instruction is available to get the carry of addition or subtraction of words. Thus, we need to calculate explicitly the carry using the following trick: the carry of $a + b$ is equal to the result of the test $a + b < a$. A similar trick is available for subtraction.

Separated operation and reduction become too costly for multiplication. The product $a \times b$ is $2n$ words long. Then the reduction to a single n words number is close to a division by p. In that case, the operation and the reduction are interleaved using a bit-serial scan of one operand (usually, the multiplier a). At each step, the partial product $a_{i,j} \times b$ (where $a_{i,j}$ is the jth bit of the word a_i) a is accumulated and reduced modulo p. We use the well known Montgomery's algorithm to avoid the use of multi-precision division [7]. In order to reduce memory usage we use an interleaved Montgomery's method which consists of doing multiplication and reduction at the same time. Our implementation is

[3] http://www.open64.net/

based on the Finely Integrated Operand Scanning (FIOS) method described in [6] which required only 3 extra words. Using $\beta = 2^{16}$ with 32-bit integers, we do not suffer from carry propagation along word additions.

We compare our arithmetic layer performance when data are localized in the different memory regions: local memory, shared memory and registers. In Table 1, we report computation times of modular operations for N in 160–384 bits. Operands are chosen randomly with full limb occupancy and the GPU kernel function consists in 1024 threads within 64 blocks. Each thread loads its two operands from the global memory to the chosen memory space (local, shared, register) and then compute a loop of 10 000 operations (using result in the loop as a new operand for the next operation). We perform our GPU computation on a Geforce 9800GX2 card.

	addition			multiplication		
N	**local**	**shared**	**register**	**local**	**shared**	**register**
160	9	2.3	0.7	88	40	22
192	11	2.3	0.7	125	51	33
224	23	5.0	1.1	172	107	55
256	26	3.1	1.5	214	80	81
384	38	7.4	3.9	673	221	261

Table 1. Computation times in ns for modular addition and multiplication.

One can see from this table that GPU data location in memory is important, and register usage can improve speed. One can also remark that multiplication for $N > 256$ registers version becomes less efficient than shared version. This comes from the CUDA compilation chain which is not able to handle code with large register usage. Note that each thread can use at most 128 registers of 32-bit integers.

In order to demonstrate benefits of GPU for multiprecision modular arithmetic, we compare our layer with the $\mathtt{mp}\mathbb{F}_q$ library [3] which is currently the best library to perform modular arithmetic on CPU with modulo of moderate size (i.e. less than 600 bits). In Table 2 we report the comparison of our best GPU modular arithmetic implementation (register version) with $\mathtt{mp}\mathbb{F}_q$ library. Benchmarks are identical as in previous section, except that $\mathtt{mp}\mathbb{F}_q$ handles sequentially the 1024 threads on a Core(TM)2 Duo E8400-3GHz processor. For N less than 384 bits GPU beats CPU calculation. However, speedup factors are moderate since CPU processor has higher frequency than GPU multiprocessor. Moreover, $\mathtt{mp}\mathbb{F}_q$ uses well tuned code with SIMD SSE-2 instructions, which allows to be more efficient implementation of large integers than our GPU implementation.

	addition mod p			multiplication mod p		
N	**our impl.**	**mpfq**	**speedup**	**our impl.**	**mpfq**	**speedup**
160	0.7	15	×21	22	64	×2.9
192	0.7	16	×22	33	70	×2.1
224	1.1	21	×19	55	105	×1.9
256	1.5	21	×14	81	109	×1.3
384	3.9	30	×7	261	210	×0.8

Table 2. Time comparison in ns of software library $\mathtt{mp}\mathbb{F}_q$ and our best GPU implementation.

3. ECC Application

In order to evaluate our library in a realistic application, we use it for scalar multiplications. This is the main operation required in ECC. For instance, servers performing many digital signatures have to compute parallel and independent scalar multiplications. Below, we consider an elliptic curve (EC) $E(\mathbb{F}_p)$ defined over the prime finite field $\mathbb{F}_p$ (a is the parameter of the curve). See [4] for background, details and notations. In order to avoid modular inversions, we use the Jacobian coordinates system ($P = (X, Y, Z)$ where X, Y and Z are in $\mathbb{F}_p$). Montgomery representation is used for fast multiplications. All threads share p and a parameters but points P and Q are different.

All operations presented in this section are used in the same way. Data are sent to the GPU global memory in coalesced form. The GPU kernel loads the selected memory (local, shared or registers) from the global memory, converts necessary values to the Montgomery form, performs the curve operation, and converts back to standard form. Then results are sent from the GPU to the CPU.

3.1. Point Addition and Doubling

Two basic operations are used for points P and Q on $E(\mathbb{F}_p)$: *point addition* $P + Q$ (for $P \neq Q$) and *point doubling* $2P$ (specific addition for $P = Q$). Those point operations are defined using several additions, multiplications over $\mathbb{F}_p$. Point addition and doubling require respectively 2 and 3 extra integers (and 6 for P and Q). These intermediate values are stored in same memory than the other data.

Timings are reported in Table 3 for the three considered memory schemes: local, shared and registers. The computation was launched on 1024 threads (64 blocks of 16 threads) and 100 operations for each thread. Due to the limited number of registers for one thread, the CUDA compiler was unable to build the code for N greater than 192 bits (*ror* is reported for compilation aborted due to "run out of register" compiler error).

	point addition			point doubling		
N	**local**	**shared**	**register**	**local**	**shared**	**register**
160	2.57	0.78	0.70	1.64	0.50	0.54
192	3.51	1.01	1.13	2.30	0.58	0.70
224	4.41	1.95	*ror*	2.73	1.01	*ror*
256	5.89	1.56	*ror*	3.71	1.09	*ror*
384	13.9	7.50	*ror*	13.3	2.42	*ror*

Table 3. Computation timings in μs for point addition and doubling.

For both operation, local memory is the worst due to its high latency. Except for point addition with $N = 160$, the shared memory implementation is the best. One drawback of shared memory is the fact it limits the number of threads per block to 16. In the shared memory version, the huge timing gap between $N = 256$ and $N = 384$ is due to the card occupancy which is twice for $N = 256$ (or less) than for $N = 384$. For point doubling, occupancy is the same for all N and the factor 2 is due to the computations. We notice that register version is slower than shared version due to the fact that the compiler was unable to use correctly all the registers it has and put data in local memory.

In Table 4, we compare our best implementation with the PACE library coupled to $\texttt{mp}\mathbb{F}_q$ running on the CPU for the same data. The shared memory version is at least twice faster than the CPU version for point doubling, but is no more than twice for addition. The difference comes from the number of Montgomery multiplications required in each operation: 9 for point doubling and 16 for point addition and the fact that the speedup factor for one Montgomery multiplication is small.

	point addition			point doubling		
N	**our impl.**	**mpfq+pace**	*speedup*	**our impl.**	**mpfq+pace**	*speedup*
160	0.78	1.52	*1.9*	0.50	1.99	*4.0*
192	1.01	1.91	*1.9*	0.58	1.99	*3.4*
224	1.95	2.65	*1.3*	1.01	2.69	*2.6*
256	1.56	2.65	*1.7*	1.09	2.65	*2.4*
384	7.50	5.11	*0.7*	2.42	5.01	*2.0*

Table 4. Time [μs] for our GPU implementation and (PACE+$\texttt{mp}\mathbb{F}_q$) on CPU.

3.2. Scalar Multiplication

Scalar multiplication $Q = [k]P = P + \cdots + P$ (with k additions of point P) where k is an N-bit integer and P a point of $E(\mathbb{F}_p)$, is the main operation in ECC protocols (see [4] for details and algorithms). We use a right-to-left double-and-add algorithm since it does not require pre-computations and additional storage. It uses point addition and point doubling for each step of a loop over the bits of k. Furthermore, it allows some threads to follow the same execution pattern as the base point is doubled at each step of the loop.

Table 5 reports times and operation throughput (in $[k]P/s$) for our best implementation (i.e., the shared version) and $\texttt{mp}\mathbb{F}_q$+PACE implementations. Our tests use the same N for k and $\mathbb{F}_p$ and the Hamming weight of k equal to $N/2$. For $N = 384$, our scalar multiplication on GPU is slower than the CPU version (computed with $\texttt{mp}\mathbb{F}_q$+PACE). We noticed that the speedup factor decreases for larger numbers. This is mainly due to the small factor between the $\texttt{mp}\mathbb{F}_q$+PACE version and our shared memory version of the Montgomery multiplication (1.6 for $N = 160$ and 1.3 for $N = 256$).

	computation time in μs		operation throughput in [k]P/s		
N	**our impl.**	**mpfq+pace**	**our impl.**	**mpfq+pace**	*speedup*
160	179	464	5586	2155	*2.6*
192	304	550	3289	1818	*1.8*
224	507	878	1972	1138	*1.7*
256	617	1003	1620	997	*1.6*
384	4609	2941	216	340	*0.6*

Table 5. Scalar multiplication result for our GPU implementation and (PACE+$\texttt{mp}\mathbb{F}_q$) on CPU.

In [9], a seminal GPU implementations of $[k]P$ is provided with $N = 224$. Their throughput is about 1412.6 $[k]P/s$ using mixed affine-Jacobian coordinates and left-to-right double-and-add algorithm. Our implementation is a little better than this result.

3.3. w-NAF Implementation of Scalar Multiplication

Some signed-digit representations are used for recoding k in the $[k]P$. This is motivated by the fact that point subtraction on an EC is just as efficient as addition. Among these representations, w-digit windows non-adjacent forms (w-NAF) are frequently used in ECC [4]. Using w-NAF, $k = \sum_{i=0}^{l-1} k'_i 2^i$ where non-zero digits k'_i are odd, $|k'_i| < 2^{w-1}$ and at most one digit of any w-digit window is non-zero. Thus w-NAF recoding decreases the number of point additions/subtractions ($\approx 1/(w+1)$ w.r.t. 0.5 for binary). It is also used as a simple countermeasure against side channel attacks [2].

Several versions have been implemented: $w \in \{2, 3\}$ as well as basic and optimized storage. The basic storage uses a complete word for each w-NAF digit. The optimized storage uses the minimal number of bits for a w-NAF digit (i.e., 2 bits for $w = 2$ and 3 bits for $w = 3$) through a dedicated multiple precision storage. For $w = 3$, $3P$ is precomputed and stored. Addition of $-P$ and $-3P$ are obtained by subtracting respectively P and $3P$. Recoding from integer k into w-NAF is performed on the GPU.

Due to `nvcc` compiler limitations, it was not possible to compile more than $N = 256$ bits with $w = 2$ and $N = 160$ for $w = 3$. Using a $w = 3$ leads to 10% speed improvement compared to $w = 2$. The number of operations is reduced, but the additional internal value $3P$ puts a too high pressure on scheduling. Surprisingly, basic and optimized storage of the w-NAF recoding gives very close results.

Conclusion and Future Prospects

In this work we report our first implementation results on modular arithmetic for large integers on GPUs which achieves a speedup of 2.6 compared to state-of-the-art library. We show that porting modular arithmetic algorithms on GPUs is not direct. Our long term goal is to design a high-performance arithmetic library for cryptography.

Acknowledgments

The authors are grateful to the Nvidia donation program for its support with GPUs cards.

References

[1] Andrei Alexandrescu. *Modern C++ design: generic programming and design patterns applied*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[2] A. Byrne, N. Meloni, A. Tisserand, E. M. Popovici, and W. P. Marnane. Comparison of simple power analysis attack resistant algorithms for an ECC. *Journal of Computers*, 2(10):52–62, 2007.

[3] P. Gaudry and E. Thomé. The mpFq library and implementing curve-based key exchanges. In *Proc. Software Performance Enhancement for Encryption and Decryption Workshop*, pages 49–64, 2007.

[4] D. Hankerson, A. Menezes, and S. Vanstone. *Guide to Elliptic Curve Cryptography*. Springer, 2004.

[5] L. Imbert, A. Peirera, and A. Tisserand. A library for prototyping the computer arithmetic level in elliptic curve cryptography. In F. T. Luk, editor, *Proc. Advanced Signal Processing Algorithms, Architectures and Implementations XVII*, volume 6697, pages 1–9, San Diego, California, U.S.A., August 2007. SPIE.

[6] Cetin Kaya Koc, Tolga Acar, and Jr. Burton S. Kaliski. Analyzing and comparing montgomery multiplication algorithms. *IEEE Micro*, 16(3):26–33, 1996.

[7] P. L. Montgomery. Modular multiplication without trial division. *Mathematics of Computation*, 44(170):519–521, April 1985.

[8] NVIDIA. *NVIDIA CUDA Programming Guide 2.0*. 2008.

[9] R. Szerwinski and T. Güneysu. Exploiting the power of gpus for asymmetric cryptography. In Springer, editor, *Proc. Cryptographic Hardware and Embedded Systems*, volume 5154, pages 79–99, 2008.