Fast Modular Arithmetic on the Kalray MPPA-256 Processor for an Energy-Efficient Implementation of ECM

Masahiro Ishii[1], Jérémie Detrey[2], Pierrick Gaudry[2], Atsuo Inomata[3], and Kazutoshi Fujikawa[3]

[1]Nara Institute of Science and Technology, Nara, Japan
masahiro-i@is.naist.jp
[2]LORIA (INRIA, CNRS and Université de Lorraine), Nancy, France
jeremie.detrey@loria.fr, pierrick.gaudry@loria.fr
[3]Information Initiative Center, Nara Institute of Science and Technology, Nara, Japan
atsuo@itc.naist.jp, fujikawa@itc.naist.jp

April 11, 2016

Abstract

The Kalray MPPA-256 processor is based on a recent low-energy manycore architecture. In this article, we investigate its performance in multiprecision arithmetic for number-theoretic applications. We have developed a library for modular arithmetic that takes full advantage of the particularities of this architecture. This is in turn used in an implementation of the ECM, an algorithm for integer factorization using elliptic curves. For parameters corresponding to a cryptanalytic context, our implementation compares well to state-of-the-art implementations on GPU, while using much less energy.

Keywords: Kalray MPPA-256 manycore processor; Multiprecision modular arithmetic; Integer factorization; Elliptic curve method

1 Introduction

Invented in 1985 by Lenstra [15], the elliptic curve method (ECM) is an integer factoring algorithm that is today considered the best one when one wants to extract prime factors of moderate size in a large number. It is therefore the method of choice when one wants to check if a number is smooth (*i.e.*, if all its prime factors are below a certain bound). It is also used as one of the steps in the factorization toolchain in general-purpose computer algebra systems such as Sage, GP/Pari, Magma or Maple. The widespread GMP-ECM [19] is a reference implementation in this context; more recent libraries like EECM-MPFQ [5] make use of the faster elliptic curve arithmetic provided by the so-called twisted Edwards curves, instead of the traditional Montgomery model.

As a smoothness test, ECM is also an important subroutine for more general algorithms. We focus here on ECM parameters that are relevant in the context of the number field sieve (NFS) for integer factorization or for computing discrete logarithms in large-characteristic finite fields [14]. In NFS, a large proportion of the time is spent looking for relations, which can be done by sieving or by ECM, and more generally with a combination of these two strategies. In NFS variants that yield the best asymptotical complexities, namely Coppersmith's multiple

polynomial NFS [8], or batch NFS [6], the role of ECM in the relation collection step is even more important. For a 768-bit integer handled with NFS, ECM is run on inputs that have typically around 200 bits, and the smoothness bound has about 35 bits.

Apart from the relation collection step, ECM is also important in the final step of NFS for discrete logarithms, called the individual logarithm step, where a descent phase is initialized using a smoothness test. Here, the input can have up to 500 bits, and the smoothness bound is also larger, but there are still too few published data on the topic to be precise. In a LogJam-type attack [2], assuming the large precomputation has been done, this smoothing step with ECM is the bottleneck.

In those two contexts related to NFS, the quantity of numbers to be tested for smoothness is huge, but this is a task that can be parallelized in a straightforward way. This is the reason why a lot of effort has been put in decreasing the cost of ECM for numbers of moderate sizes, in particular using non-general-purpose coprocessors. In [7], Bos and Kleinjung optimized ECM using twisted Edwards curves on GPU. This was further improved in [16] and provides the most efficient implementation so far for the NFS context, using algorithmic improvements to fit the memory constraints of a GPU environment.

In this paper, we want to explore the potential of the MPPA-256 processor developed by Kalray [1] as an ECM coprocessor. This is a recently designed, lightweight manycore processor, where each of the 256 cores is an independent 32-bit VLIW architecture. In the ECM algorithm, most of the time is spent in the elliptic curve group law, that must be performed modulo the integer that is being factored. Therefore, in the end, most of the time is spent doing multiprecision modular arithmetic, in particular modular multiplications, and this operation must be optimized as much as possible.

We propose a library for multiprecision arithmetic for numbers of fixed sizes corresponding to our target in the NFS context, where all critical parts are written in assembly, taking full advantage of the VLIW architecture to explicitly schedule the operations in all available pipelines. On top of it we implemented the ECM algorithm, following the algorithmic ideas of [16], that we slightly improved. The memory constraints of a GPU and of the MPPA-256 are rather different, but the same strategies behave pretty well.

The results are quite satisfactory: in terms of number of curves tried per second on the whole chip, the GPU is faster than the MPPA-256 by a factor around 3, but this must be put in a larger perspective since the peak power consumption of the MPPA-256 is only 16 W, while the GPU needs a bit less than 250 W. So, in terms of number of curves tried per joule, the count is in favor of the MPPA-256 by a factor ranging from 5 to 7, depending on the context.

The source code written for all our experiments is distributed under a free-software license and can be downloaded from `https://gforge.inria.fr/projects/kalray-ecm`. Although the ECM part is admittedly quite specialized, the multiprecision modular arithmetic library can be used in other contexts.

The paper is organized as follows. In the next section, we start with a description of the MPPA-256 processor, where we insist in particular on the architecture of the individual cores. Then, in Section 3, we explain our low-level implementation of the multiprecision modular arithmetic library. Finally, Section 4 contains details about the ECM applications, with benchmarks and a comparison with the literature.

2 The Kalray MPPA-256 manycore processor

2.1 Global overview

Launched in 2012, the Kalray MPPA-256 processor (codenamed Andey) is a single 28 nm CMOS chip, clocked at 400 MHz, which integrates a 4×4 array of 16-core *compute clusters* (CCs), along with 4 quad-core *I/O subsystems* located on the north, south, east and west ends of the chip, all connected by means of two toric networks-on-chip (NoCs), as depicted in Figure 1.

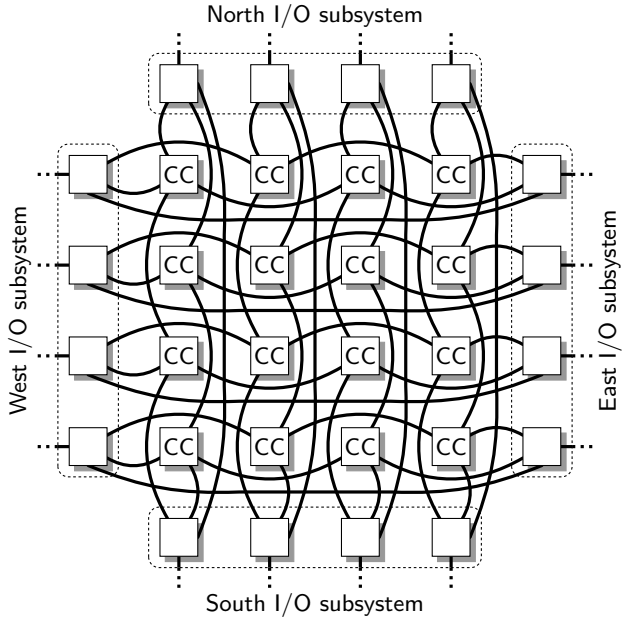


Figure 1: Global architecture of the Kalray MPPA-256 [10].

Each compute cluster is composed of 16 cores, or *processing engines* (PEs), along with an extra core, the *resource manager* (RM), reserved for system use, and a 2 MB memory bank, shared by the 17 cores. A schematic view of a compute cluster is given in Figure 2.

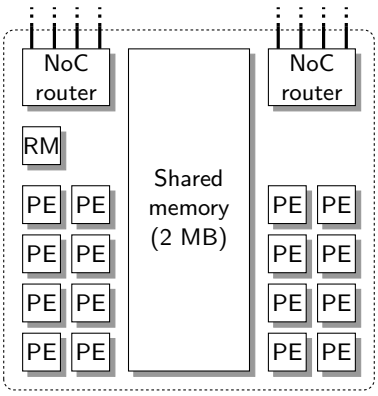


Figure 2: Details of a compute cluster [10].

Each core of the I/O subsystems runs under the RTEMS[1] real-time operating system, while the RM of each compute cluster runs under NodeOS, a specific operating system developed by

[1] *Real-Time Executive for Multiprocessor Systems*, `https://www.rtems.org/`.

Kalray. Both RTEMS and NodeOS implement POSIX-compatible APIs. MPPA-256 applications are then designed as POSIX-like processes deployed on the I/O subsystems and on the compute clusters, communicating together through the NoCs using network operations similar to `read`s and `write`s on UNIX sockets. Finally, a Pthreads-like interface allows one to run up to 16 threads in parallel on each compute cluster, thanks to their multi-core architecture.

2.2 Core architecture

The cores in the MPPA-256 are all based on the Kalray-1 (or K1) microarchitecture. It is an in-order, fully-pipelined, 32-bit, VLIW (*Very Long Instruction Word*) processor, which embeds five execution units: two *Arithmetic & Logic Units* (ALU_0 and ALU_1), a *Multiply–Accumulate Unit* (MAU), a *Load/Store Unit* (LSU), and a *Branch & Control Unit* (BCU). The MAU can also serve as *Floating-Point Unit* (FPU), and both the MAU and the LSU also support a subset of the ALU instruction set (referred to as ALU_{tiny}).

These execution units communicate by means of a shared register file (RF) of 64 32-bit general-purpose registers, which supports up to 11 read and 4 write accesses per cycle. In case of read-after-write dependencies, the register file can by bypassed, and the output of one unit directly used as the input of another one, so as to save one clock cycle between consecutive dependent instructions.

Finally, each K1 core has dedicated instruction and data caches of 8 kB each, along with a 64-byte write buffer.

The microarchitecture, along with a schematic representation of the pipeline stages, are depicted in Figure 3.

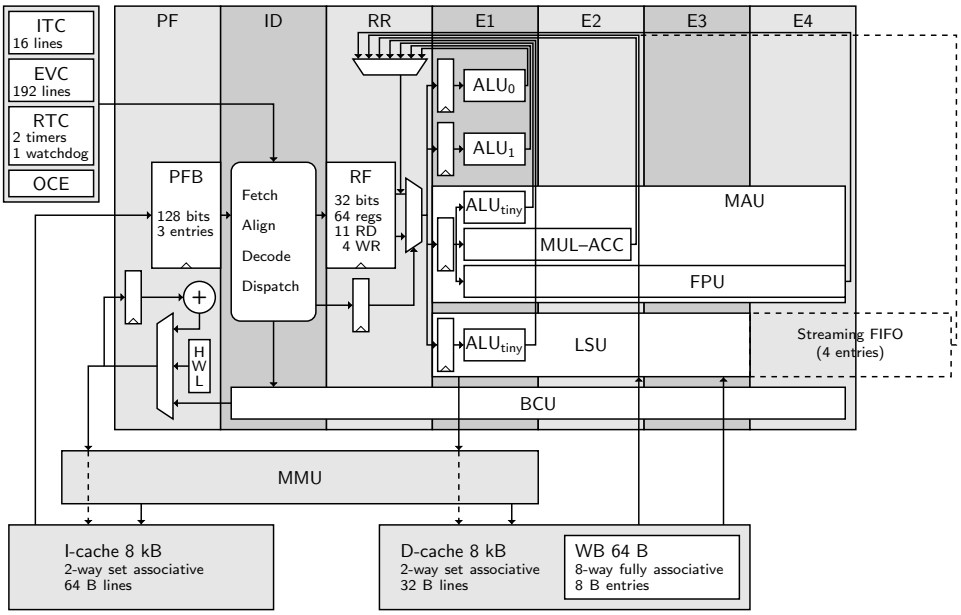


Figure 3: VLIW pipeline of the K1 architecture [9].

2.3 The Kalray-1 instruction set

The ALU_{tiny} instruction set, which is supported by both ALUs, along with the MAU and the LSU, covers most of the simple 32-bit integer operations, such as addition, subtraction and bitwise logic. The main ALUs also support a few extra integer instructions (such as shifts), and

can even be combined to support 64-bit instructions, operating on pairs of registers. All these ALU instructions have a 1-cycle latency.

The MAU supports a fully pipelined $32 \times 32 \to 64$-bit integer multiplication, with a 2-cycle latency and a 1-cycle inverse throughput. It is also possible to couple this multiplication with a 64-bit accumulation into a register pair at no additional cost.

The FPU, which shares its logic with the MAU, supports IEEE-754-compliant single-precision floating-point arithmetic, along with a few double-precision operations as well. However, we do not consider those in this work.

The LSU, in charge of all memory accesses, supports both 32- and 64-bit loads and stores. When the data is available in the cache, read instructions have a latency of only 2 cycles. A cache miss incurs a pipeline stall of approximately 10 cycles.

The BCU supports branches and function calls, which come at the cost of only a few cycles thanks to the low pipeline depth. The BCU also offers support for hardware loops, in which successive loop iterations are chained without any branching penalty.

Finally, since the Kalray-1 is a VLIW microarchitecture, it is possible to explicitly group instructions into *instruction bundles* which are to be issued at the same clock cycle and executed in parallel, as long as they are processed by different execution units. For instance, one can very well schedule in a single bundle a 64-bit addition (on the two ALUs), a 32-bit multiplication (on the MAU), a 64-bit load (on the LSU), and a conditional branch (on the BCU). Even if this puts higher pressure on the compiler to extract parallelism from the code, this allows one to finely tune and optimize critical parts of an application at the assembly level.

3 Multiprecision modular arithmetic

In this section, we present a flexible library for fast multiprecision modular arithmetic on the Kalray MPPA-256 processor. Even though C bindings are available for easy integration into larger projects, most of it is written in pure assembly code for efficiency purposes.

After detailing the data representation and algorithmic choices made in this library for the central operations, we present a few benchmark results in Section 3.7.

3.1 Representation

In the proposed library, integers are assumed to be unsigned (*i.e.*, non-negative), and are represented in radix 2^{32} using arrays of 32-bit words. For instance, the n_W-word array $(x_0, \ldots, x_{n_W-1})$ represents the $(32n_W)$-bit integer

$$X = \sum_{i=0}^{n_W-1} x_i \cdot 2^{32i}.$$

In the usual context of ECM, the size of the integers N we want to factor is known in advance. Consequently, for the sake of efficiency, the parameter n_W is fixed at compile time using a preprocessor macro. Supported values for n_W range from 2 to 16, inclusive, which corresponds to moduli N of size from 64 to 512 bits.

Note that, given the MPPA-256 two-level hierarchy of compute clusters and processing engines, it is perfectly possible to compile separate binaries with different values for n_W and have them run simultaneously on distinct compute clusters. This would allow an ECM implementation to schedule incoming numbers N on different clusters, according to their size, and even to dynamically reallocate compute resources to match the size distribution of these numbers. This is however not explored in this work.

3.2 Basic integer operations

Most of the basic arithmetic operations, such as integer addition, subtraction, comparison, assignment, and so on, were implemented in the proposed library. As can be expected, their latency $T_{\text{op}}(n_W)$ is linear in n_W, and most of our optimization efforts concentrated on minimizing the ratio $T_{\text{op}}(n_W)/n_W$. We illustrate this by detailing the case of the addition in the following paragraphs.

Suppose then that we are given the address in memory of two n_W-word integers $X = (x_0, \ldots, x_{n_W-1})$ and $Y = (y_0, \ldots, y_{n_W-1})$, and that we want to compute their sum as the n_W-word integer $R = (r_0, \ldots, r_{n_W-1})$ along with the carry-out bit c:

$$X + Y = R + c \cdot 2^{32n_W}.$$

Since the K1 microarchitecture supports a 32-bit add-with-carry instruction (denoted by `addc` here) using a dedicated carry flag, a straightforward implementation would thus look something like the following pseudo-code (in which we denote by `X`, `Y`, and `R` the registers containing the memory addresses of the corresponding multiprecision integers):

```
addc 0, 0                          (Clear carry flag)
i ← 0                              (Initialize index)
repeat n_W times                   (Hardware loop)
    x ← load [X+4i]                (Load i-th word x_i)
    y ← load [Y+4i]                (Load i-th word y_i)
    r ← addc x, y                  (Add with carry)
    [R+4i] ← store r               (Store i-th word r_i)
    i ← add i, 1                   (Increment index)
c ← addc 0, 0                      (Save carry flag)
```

Assuming the operands are already in the L1 cache, each `load` has a latency of 2 cycles. However, the two `load`'s of each iteration can be pipelined and issued in two consecutive clock cycles. The add-with-carry, `store`, and increment instructions then require 1 cycle each, which gives a total of 6 cycles per iteration. Note that the use of a hardware loop allows us to completely avoid branching penalties after each iteration. We thus obtain a latency of $T_{\text{add}}(n_W) = 6n_W + O(1)$ cycles for the complete addition.

In fact, as mentioned earlier, the K1 instruction set includes 64-bit memory accesses, and the two main ALUs can be combined to support a 64-bit add-with-carry instruction. As these instructions have the same latency as their 32-bit counterparts, they can then be used to process the operands and compute the result two words at a time.

Furthermore, since the `store` and increment instructions are executed on different execution units (the LSU for the former, and one of the ALUs for the latter), both can be executed in parallel in the same clock cycle, thanks to the VLIW capabilities of the K1 microarchitecture, by explicitly writing these two instructions in the same *instruction bundle* at the assembly level.

These two improvements yield an addition having latency $T_{\text{add}}(n_W) = 5\lceil n_W/2 \rceil + O(1)$, as shown in the following pseudo-code (where the dotted horizontal lines delimitate the different instruction bundles and, for the sake of simplicity, restricted to the case where n_W is even):

```
addc 0, 0                                   (Clear carry flag)
i ← 0                                       (Initialize index)
repeat n_W/2 times                          (Hardware loop)
    x:x' ← load_64 [X+8i]                   (Load i-th dword)
    y:y' ← load_64 [Y+8i]                   (Load i-th dword)
    r:r' ← addc_64 x:x', y:y'               (Add with carry)
    [R+8i] ← store_64 r:r'                  (Store i-th dword)
    i ← add i, 1                            (Increment index)
c ← addc 0, 0                               (Save carry flag)
```

This is still not optimal, however: software pipelining techniques can be used to carefully rearrange and interleave the instructions of consecutive loop iterations, so as to maximize the instruction-level parallelism. For instance, one can schedule the addition-with-carry of the two $(i-1)$-st double-words (x_{2i-2}, x_{2i-1}) and (y_{2i-2}, y_{2i-1}) in parallel with the load of the next double-word (x_{2i}, x_{2i+1}):

```
x:x' ← load_64 [X]                          (Load first dword)
addc 0, 0                                   (Clear carry flag)
y:y' ← load_64 [Y]                          (Load first dword)
i ← 1                                       (Initialize load index)
j ← 0                                       (Initialize store index)
repeat n_W/2 times                          (Hardware loop)
    x:x' ← load_64 [X+8i]                   (Load i-th dword)
    r:r' ← addc_64 x:x', y:y'               (Add with carry)
    y:y' ← load_64 [Y+8i]                   (Load i-th dword)
    i ← add i, 1                            (Increment load index)
    [R+8j] ← store_64 r:r'                  (Store j-th dword)
    j ← add j, 1                            (Increment store index)
r:r' ← addc_64 x:x', y:y'                   (Add with carry)
[R+8j] ← store_64 r:r'                      (Store last dword)
c ← addc 0, 0                               (Save carry flag)
```

The resulting instruction scheduling on the various execution units for two consecutive iterations of the loop is given in the following table. Instructions corresponding to the same double-words of the operands and of the result are shown in the same color.

Cycle	LSU	ALU_0 & ALU_1
...	...	...
t	$x{:}x' \leftarrow \texttt{load}_{64}\ [X+8i]$	$r{:}r' \leftarrow \texttt{addc}_{64}\ x{:}x', y{:}y'$
$t+1$	$y{:}y' \leftarrow \texttt{load}_{64}\ [Y+8i]$	$i \leftarrow \texttt{add}\ i, 1$
$t+2$	$[R+8j] \leftarrow \texttt{store}_{64}\ r{:}r'$	$j \leftarrow \texttt{add}\ j, 1$
$t+3$	$x{:}x' \leftarrow \texttt{load}_{64}\ [X+8i]$	$r{:}r' \leftarrow \texttt{addc}_{64}\ x{:}x', y{:}y'$
$t+4$	$y{:}y' \leftarrow \texttt{load}_{64}\ [Y+8i]$	$i \leftarrow \texttt{add}\ i, 1$
$t+5$	$[R+8j] \leftarrow \texttt{store}_{64}\ r{:}r'$	$j \leftarrow \texttt{add}\ j, 1$
...	...	...

One can see from this scheduling that, even though the latency required to load, add, then store a pair of double-words is 6 clock cycles, each iteration now has a latency of only 3 cycles. Therefore, the total latency for this operation is $T_{\text{add}}(n_W) = 3\lceil n_W/2 \rceil + O(1)$ cycles.

This can be shown to be optimal, as the bottleneck for the addition lies in the *Load/Store Unit*, which has to load the $2\lceil n_W/2 \rceil$ double-words of the operands X and Y, and store the $\lceil n_W/2 \rceil$ double-words of the result R, thus requiring at least $3\lceil n_W/2 \rceil$ clock cycles.

Finally, note that, when n_W is small, a few cycles can be saved in the $O(1)$ part of the latency by fully unrolling the main loop. This avoids the constant-time overhead of the hardware loop, at the expense of an increase in code size, whose complexity jumps from $O(1)$ to $O(n_W)$.

3.3 Basic modular arithmetic

Basic modular operations such as negation, addition or subtraction directly rely on their integer counterparts on n_W-word operands described in the previous section. Operands are assumed to be already reduced with respect to the modulus N.

After the main operation, a final reduction step compares the result to the modulus N and conditionally subtracts or adds it (in the case of a modular addition or subtraction, respectively). This comparison is performed most-significant digits first, so as to return an answer as quickly as possible. Thus, it has an average latency of only a few cycles, even though its worst-case complexity (in the case of equality) is still linear in n_W.

3.4 Integer multiplication

Given two n_W-word multiprecision integers X and Y, their $2n_W$-word product $R = X \cdot Y$ is computed using a quadratic parallel–serial algorithm: the n_W words of the multiplicand X are first all loaded into registers, then, for i ranging from 0 to $n_W - 1$, each partial product $X \cdot y_i$ is computed, shifted left by i words, and accumulated into the partial result:

$R \leftarrow 0$
for $i \leftarrow 0$ **to** $n_W - 1$ **do**
    $R \leftarrow R + X \cdot y_i \cdot 2^{32i}$
return R

Note that each partial product $X \cdot y_i$ fits on $n_W + 1$ words, and that, before the i-th partial product is accumulated, the most-significant words r_{n_W+i} to r_{2n_W-1} of the partial result are all 0. Furthermore, because of the left shift by i words, this means that the accumulation into R will only modify words r_i to r_{n_W+i}, and the carry need not be propagated further. Also, after accumulating the i-th partial product, the i-th word r_i will have reached its final value, and may then be written back to memory. Consequently, at any point in the algorithm, only $n_W + 1$ words of the partial result (from r_i to r_{n_W+i}) need to be kept in the register file. Hence, the total number of registers required for the multiplication is $2n_W + O(1)$.

In order to simplify the carry propagation when accumulating each partial product $X \cdot y_i$ into R, the words x_j of the multiplicand X are processed separately according to the parity of their index j: we write $X = X_0 + X_1 \cdot 2^{32}$, with

$$X_0 = \sum_{k=0}^{\lceil n_W/2 \rceil - 1} x_{2k} \cdot 2^{64k}, \qquad \text{and}$$
$$X_1 = \sum_{k=0}^{\lfloor n_W/2 \rfloor - 1} x_{2k+1} \cdot 2^{64k}.$$

This way, we first compute the sub-product $S_0^{(i)} = X_0 \cdot y_i$, whose individual products $x_{2k} \cdot y_i \cdot 2^{64k}$ are contiguous but do not overlap, and directly accumulate it into R. We then compute the second sub-product $S_1^{(i)} = X_1 \cdot y_i$, which is also contiguous and overlap-free, and finally accumulate it into R as well.

The *Multiply–Accumulate Unit* (MAU) of the K1 microarchitecture supports a $32 \times 32 \rightarrow 64$-bit integer multiplication, which has a latency of 2 cycles and an inverse throughput of 1 cycle,

meaning that one such instruction can be issued at every clock cycle. As this matches the inverse throughput of the 64-bit add-with-carry instructions, we can therefore efficiently pipeline each individual product of $S_0^{(i)}$, and then of $S_1^{(i)}$, with its accumulation into R, using only two extra 64-bit registers (denoted by $u{:}u'$ and $v{:}v'$) as buffers for the products.

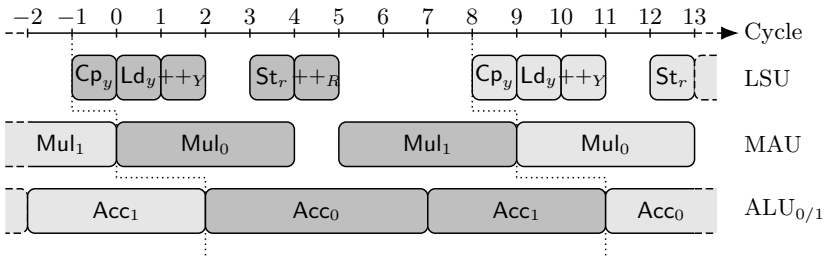
The following scheduling illustrates this for the computation and accumulation of $S_0^{(i)}$ then of $S_1^{(i)}$ into R, for $n_W = 8$, where we assume that the registers x_0 to x_{n_W-1} contain the n_W words of X, that y contains y_i, and that r_0 to r_{n_W} contain the $n_W + 1$ "active" words r_i to r_{n_W+i} of the partial result:

Cycle	MAU	ALU_0 & ALU_1
0	$u{:}u' \leftarrow \texttt{mul}\ x_0,\ y$	
1	$v{:}v' \leftarrow \texttt{mul}\ x_2,\ y$	
2	$u{:}u' \leftarrow \texttt{mul}\ x_4,\ y$	$r_0{:}r_1 \leftarrow \texttt{addci}_{64}\ r_0{:}r_1,\ u{:}u'$
3	$v{:}v' \leftarrow \texttt{mul}\ x_6,\ y$	$r_2{:}r_3 \leftarrow \texttt{addc}_{64}\ r_2{:}r_3,\ v{:}v'$
4		$r_4{:}r_5 \leftarrow \texttt{addc}_{64}\ r_4{:}r_5,\ u{:}u'$
5	$u{:}u' \leftarrow \texttt{mul}\ x_1,\ y$	$r_6{:}r_7 \leftarrow \texttt{addc}_{64}\ r_6{:}r_7,\ v{:}v'$
6	$v{:}v' \leftarrow \texttt{mul}\ x_3,\ y$	$r_8 \leftarrow \texttt{addc}\ 0,\ 0$
7	$u{:}u' \leftarrow \texttt{mul}\ x_5,\ y$	$r_0{:}r_1 \leftarrow \texttt{addci}_{64}\ r_1{:}r_2,\ u{:}u'$
8	$v{:}v' \leftarrow \texttt{mul}\ x_7,\ y$	$r_2{:}r_3 \leftarrow \texttt{addc}_{64}\ r_3{:}r_4,\ v{:}v'$
9		$r_4{:}r_5 \leftarrow \texttt{addc}_{64}\ r_5{:}r_6,\ u{:}u'$
10		$r_6{:}r_7 \leftarrow \texttt{addc}_{64}\ r_7{:}r_8,\ v{:}v'$

In the above scheduling, the $\texttt{addci}_{64}$ instructions clear the carry flag before performing an addition-with-carry. This avoids having to use an extra instruction to do so. Also note that the indices of the output registers of the second sequence of $\texttt{addci}_{64}/\texttt{addc}_{64}$'s are always one less than the indices of the corresponding input registers: this allows us to implement at no extra cost a sliding window for the $n_W + 1$ "active" words of R, so that this pattern can be repeated in a loop to iterate through the words of Y. As a direct consequence, the register r_0 gets overwritten at cycle 7: the contents of r_0 should therefore be stored back to memory as word r_i between cycles 3 and 7. Finally, one can verify that the final addition at cycle 10 will never generate an output carry.

We should also mention at this point that the K1 MAU supports a multiply-and-accumulate-with-carry instruction, which serves the same purpose as the combination of `mul` and $\texttt{addc}_{64}$ we use here, only with a latency of only 2 cycles instead of 3. However, this instruction has extra constraints regarding which pairs of 32-bit registers can be used as the accumulator: it turns out that these constraints are incompatible with the shift by one word that happens when accumulating $S_1^{(i)}$ into R (see cycles 7 to 10 in the previous scheduling). This is why we decided not to use this instruction.

Hence, using this method, each partial product $X \cdot y_i$ can be computed and accumulated into R in $n_W + 3$ clock cycles. However, when iterating through the partial products, we can slightly overlap consecutive iteration by 2 cycles, thus reducing the cost to $n_W + 1$ cycles per iteration, as depicted in the following "high-level" scheduling, for $n_W = 8$, in which one can see the iteration pattern repeating every 9 cycles.



In this scheduling, the tasks Mul_k and Acc_k represent the computation and the accumulation of $S_k^{(i)}$, respectively. At each iteration, the multiplier word y_i, which was preloaded into a buffer register $\boldsymbol{y}'$ by the task Ld_y in the previous iteration, is copied into the actual register $\boldsymbol{y}$ by task Cp_y. Once computed, the result word r_i, contained in register $\boldsymbol{r}_0$, is then stored back to memory by task St_r. Finally, tasks $++_Y$ and $++_R$ are in charge of incrementing the read pointer on Y and the write pointer on R, respectively.

One can show that this scheduling is optimal, as the two main ALUs have to accumulate and propagate carries through a total of $n_W + 1$ words at each iteration (this would be also the case if the multiply-and-accumulate-with-carry instruction were used).

Therefore, all in all, our implementation computes a product of two n_W-word integers in $T_{\text{mul}}(n_W) = n_W(n_W + 1) + O(1)$ clock cycles, which is only slightly more than 1 cycle per individual word-by-word product.

Finally, note that subquadratic algorithms such as Karatsuba might be more efficient for larger values of n_W, but this is not the case for the sizes considered in this work.

3.5 Montgomery reduction

Given an odd n_W-word modulus N along with the constant $R = 2^{32n_W}$, the *Montgomery reduction* [17] of a $2n_W$-word integer $X < N \cdot R$ with respect to N is defined as $\text{REDC}_N(X) = X \cdot R^{-1} \bmod N$. As $N < R$, using the *Montgomery representation* of integers modulo N, in which the elements $X \in \mathbb{Z}/N\mathbb{Z}$ are represented by $\widetilde{X} = X \cdot R \bmod N$, the product $Z = X \cdot Y \bmod N$ of two such residues X and $Y \in \mathbb{Z}/N\mathbb{Z}$ can then be computed as $\widetilde{Z} = X \cdot Y \cdot R \bmod N = \text{REDC}_N(\widetilde{X} \cdot \widetilde{Y})$.

Given the precomputed constant $\widetilde{R} = R^2 \bmod N$, conversions to and from this representation can be computed using only n_W-word integer multiplications and Montgomery reductions, as $\widetilde{X} = \text{REDC}_N(X \cdot \widetilde{R})$ and $X = \text{REDC}_N(\widetilde{X})$, respectively.

Finally, as it is also compatible with addition, subtraction and negation modulo N, we can perform all the computations required for ECM in Montgomery representation in order to avoid conversions before and after each modular multiplication.

In [17], Montgomery gives an efficient algorithm requiring only multiplications for computing $\text{REDC}_N(X)$, provided that the 1-word constant $n' = (-N)^{-1} \bmod 2^{32}$ is known (thanks to a precomputation, for instance):

$T \leftarrow (x_0, \ldots, x_{n_W-1})$ *(i.e., $T \leftarrow X \bmod 2^{32n_W}$)*
for $i \leftarrow 0$ **to** $n_W - 1$ **do**
    $q \leftarrow t_0 \cdot n' \bmod 2^{32}$
    $T \leftarrow x_{n_W+i} \cdot 2^{32(n_W-1)} + (T + q \cdot N)/2^{32}$
if $T \geq N$ **then**
    $T \leftarrow T - N$
return T

The partial result T is first initialized with the n_W least significant words of X. Then, at each iteration, a multiple of N is added to it so as to make it divisible by 2^{32}. The value of T is then shifted right by one word, and the next word of X is loaded and added (with carry) to t_{n_W-1}. A single final subtraction of N might be necessary to keep the result below N.

At any point in the algorithm, T is an n_W-word integer along with a delayed carry bit, and thus occupies $n_W + 1$ registers denoted by $\boldsymbol{t}_0$ to $\boldsymbol{t}_{n_W}$. As the n_W-word modulus N is also kept in the register file ($\boldsymbol{n}_0$ to $\boldsymbol{n}_{n_W-1}$), the total number of registers required for this algorithm is then $2n_W + O(1)$.

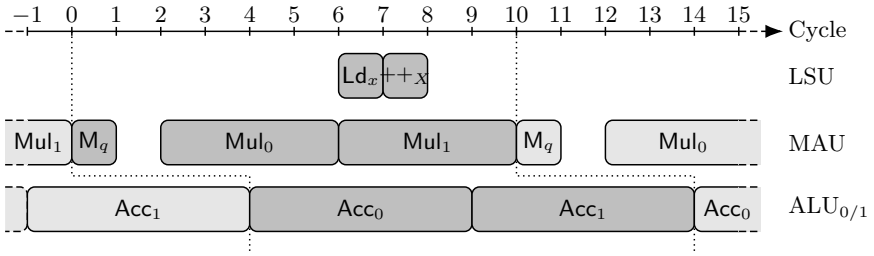
In fact, this algorithm is in many ways quite similar to that of the parallel–serial multiplication described in the previous section. In particular, by considering the odd- and even-indexed words of N and by writing $N = N_0 + N_1 \cdot 2^{32}$ as we did for X in the multiplication, we can also split the computation of the partial product $q \cdot N$ into two sub-products $S_0 = q \cdot N_0$ and $S_1 = q \cdot N_1$ and accumulate them separately into T. The only difference is that both accumulations into T might generate output carries.

The proposed scheduling, which resembles that of the multiplication, thus requires two extra cycles to compute the quotient q at the beginning of each iteration, and one extra cycle because of the longer carry chains. An example for $n_W = 8$ words is given below.

Cycle	MAU	ALU_0 & ALU_1
0	$q \leftarrow \texttt{mul}\ t_0, n'$	
1		
2	$u{:}u' \leftarrow \texttt{mul}\ q, n_0$	
3	$v{:}v' \leftarrow \texttt{mul}\ q, n_2$	
4	$u{:}u' \leftarrow \texttt{mul}\ q, n_4$	$0{:}t_0 \leftarrow \texttt{addci}_{64}\ t_0{:}t_1, u{:}u'$
5	$v{:}v' \leftarrow \texttt{mul}\ q, n_6$	$t_1{:}t_2 \leftarrow \texttt{addc}_{64}\ t_2{:}t_3, v{:}v'$
6	$u{:}u' \leftarrow \texttt{mul}\ q, n_1$	$t_3{:}t_4 \leftarrow \texttt{addc}_{64}\ t_4{:}t_5, u{:}u'$
7	$v{:}v' \leftarrow \texttt{mul}\ q, n_3$	$t_5{:}t_6 \leftarrow \texttt{addc}_{64}\ t_6{:}t_7, v{:}v'$
8	$w{:}w' \leftarrow \texttt{mul}\ q, n_5$	$t_7{:}t_8 \leftarrow \texttt{addc}_{64}\ t_8{:}0,\ x{:}0$
9	$u{:}u' \leftarrow \texttt{mul}\ q, n_7$	$t_0{:}t_1 \leftarrow \texttt{addci}_{64}\ t_0{:}t_1, u{:}u'$
10		$t_2{:}t_3 \leftarrow \texttt{addc}_{64}\ t_2{:}t_3, v{:}v'$
11		$t_4{:}t_5 \leftarrow \texttt{addc}_{64}\ t_4{:}t_5, w{:}w'$
12		$t_6{:}t_7 \leftarrow \texttt{addc}_{64}\ t_6{:}t_7, u{:}u'$
13		$t_8 \leftarrow \texttt{addc}\ t_8, 0$

In the above scheduling, we assume that the current word x_{n_W+i} of X was loaded into register x before cycle 8. Also note how the division of $T + q \cdot N$ by 2^{32} is handled transparently when accumulating S_0 into T (cycles 4 to 8).

Even though each iteration has a latency of $n_W + 6$ cycles, we can overlap consecutive iterations by 4 cycles, resulting in an actual cost of $n_W + 2$ cycles per iteration, as illustrated below in the case $n_W = 8$ (in which M_q represents the computation of q as $t_0 \cdot n'$, Ld_x and ${+}{+}_X$ the loading of x_{n_W+i} followed by incrementing the corresponding pointer, and Mul_k and Acc_k the computation and accumulation of S_k, respectively):



Therefore, the main loop of this algorithm requires $n_W(n_W + 2) + O(1)$ cycles, to which we need to add $\lceil n_W/2 \rceil$ cycles for loading N into the register file, and possibly another $\lceil n_W/2 \rceil$ cycles for subtracting N from T. The comparison between T and N is assumed to have a constant average cost of a few cycles only. All in all, this gives a total average cost of $T_{\text{REDC}}(n_W) = n_W(n_W + 3) + O(1)$ clock cycles for the Montgomery reduction, just slightly above the cost of the integer multiplication.

Finally, as mentioned at the beginning of this section, the REDC_N function can be used to efficiently reduce a $2n_W$-word product modulo N, and it is therefore called after each such

multiplication. Variants of this *Montgomery multiplication* have been proposed where the computations of the product and of the reduction are interleaved [11, 13]. However, it turns out that our implementation would not benefit from such variants: the number of carry propagations to perform would change only marginally and, more importantly, the higher number of registers required would rapidly exhaust the register file and limit us to smaller values of n_W.

3.6 GCD and modular inversion

Our library also supports a few higher-level functions, which are implemented in C, on top of the low-level arithmetic primitives described previously. This is the case for a multiprecision GCD and for a multiprecision modular inversion (in Montgomery representation), as they are required in ECM. Both were implemented using the extended binary GCD algorithm.

3.7 Benchmark results

We report in Table 1 the latency of several functions of our multiprecision library, as measured for different operand sizes on the target MPPA-256 processor. These benchmarks assume that all data is already present in the L1 cache, so that no spurious cache-miss occurs. Due to the in-order nature of the K1 microarchitecture, these timings are extremely stable.

Note that almost all timings are given for fully unrolled versions of the low-level arithmetic functions (*i.e.*, without hardware loops). The only low-level functions which were not unrolled are the integer multiplication and the Montgomery reduction for operand sizes above 256 bits ($n_W > 8$).

Timings for the modular functions (addition, Montgomery reduction and multiplication) are given as an interval, as the actual latency depends on whether final corrections (such as subtracting the modulus) have to be performed or not. However, these intervals do not include the worst-case latencies, which happen when the comparisons between the result and the modulus take linear time, as these occur only but rarely.

Finally, timings for the GCD are given as the average for a hundred runs on random n_W-word inputs.

Table 1: Measured latencies (in clock cycles) of various functions for several operand sizes.

		Latency according to operand sizes			
Function	Complexity	192 bits ($n_W = 6$)	256 bits ($n_W = 8$)	384 bits ($n_W = 12$)	512 bits ($n_W = 16$)
Integer addition	$3n_W/2 + O(1)$	16	19	25	31
Integer multiplication	$n_W(n_W + 1) + O(1)$	51	81	171	287
Modular addition	$9n_W/4 + O(1)$	33–45	36–51	42–63	48–75
Montgomery reduction	$n_W(n_W + 3) + O(1)$	68–74	95–102	191–200	314–325
Montgomery multiplication	$2n_W(n_W + 2) + O(1)$	121–127	178–185	364–373	603–614
GCD	$O({n_W}^2)$	12070	17745	30920	47560
Point addition (ext. coordinates)	$\mathbf{A} = 8\mathbf{m} + 10\mathbf{a}$	1321	1823	3402	5405
Point doubling (ext. coordinates)	$\mathbf{D} = 4\mathbf{m} + 4\mathbf{s} + 6\mathbf{a}$	1184	1668	3212	5174

4 The Elliptic Curve Method

There are many good descriptions of ECM in the literature [19] and we will not recall it in details. The general idea is the following. Let N be an integer to be tested for smoothness, and let p be an (as-yet-unknown) prime factor of N. An elliptic curve E defined over $\mathbb{Q}$ is chosen,

together with a non-torsion point P on E. Hoping to create a point whose reduction modulo p is the point at infinity on the reduced elliptic curve, we multiply P by a large integer made of all prime factors up to a given bound. If the group order of the reduced curve modulo p is smooth, we will then indeed get the neutral element. All the computations can not be made modulo p since this factor is unknown, but by reducing all coordinates modulo N, we avoid coefficient expansion, while still having compatible operations. And in the end, if the group order was indeed smooth, then some non-invertible element modulo N will pop up, usually revealing p.

4.1 Curve arithmetic

Most of the time is spent in the elliptic curve group law, where coordinates are integers modulo N. Hence having fast modular arithmetic is crucial for efficiency. At a higher level, it is important to choose an appropriate coordinate system for the elliptic curve, reducing the number of operations in $\mathbb{Z}/N\mathbb{Z}$, and also appropriate addition chains to reduce the number of additions and doublings on the elliptic curve. Since the Kalray MMPA-256 processor has similar characteristics as GPUs—namely, a lot of computing power but limited or slow memory access—we followed the same strategy as the one used in the state-of-the-art implementations of ECM on GPUs [16, 7]. Therefore, we used the so-called extended coordinates on twisted Edwards curves with $a = -1$ [3, 12].

The costs in terms of modular multiplications ($\mathbf{m}$), squarings ($\mathbf{s}$) and additions / subtractions ($\mathbf{a}$) of point addition ($\mathbf{A}$) and doubling ($\mathbf{D}$) in this coordinate system are recalled in Table 1, along with average latencies benchmarked for various sizes of the modulus N. When only projective coordinates are required for the result, a modular multiplication can be saved in both operations (which are then denoted by $\mathbf{A}'$ and $\mathbf{D}'$, respectively).

Classically, we used a two-stage scalar multiplication, where stage 1 is performed using no-storage addition chains as developed in [7], while stage 2 relies on a baby-step/giant-step approach, again following [16]. In the following, we give a few more details on these two stages, since we slightly modified them compared to [7, 16].

4.2 Addition chains for stage 1

The textbook stage 1 of ECM consist in multiplying P by a scalar of the form $\prod_{\pi^e \leq B_1} \pi^e$ for a given bound B_1. The idea of [7] is to group primes π occurring in this product in batches having low Hamming-weight, so that a scalar multiplication by those batches involves less additions than we would have with the original scalar. Finding the best chains based on this idea would imply a fully exponential search; however, using a massive precomputation it is still possible to find very good chains with a simple greedy heuristic.

We have implemented the method presented in [7] and searched for no-storage addition chains, only with a slight modification of the initial ordering of the available addition chains s_i: instead of using the ratio $r(s_i) = \mathrm{dbl}(s_i)/\mathrm{add}(s_i)$ as in [7, Algorithm 1], we used the quantity

$$\kappa(s_i) = \frac{\log_2(s_i)}{\mathrm{dbl}(s_i) + (8/7) \cdot \mathrm{add}(s_i) - \log_2(s_i)},$$

where the constant $8/7$ comes from the approximate cost ratio between an addition in extended coordinates ($\mathbf{A} \approx 8\,\mathbf{m}$) and a doubling in projective coordinates ($\mathbf{D}' \approx 3\,\mathbf{m} + 4\,\mathbf{s}$).

We chose this metric as it better takes into account the number of bits actually contributed by each addition chain. For instance, while $r(1665) = r(863) = 10/3$, as both chains can be computed in 10 doublings and 3 additions, we have $\kappa(1665) \approx 3.92$ and $\kappa(863) \approx 2.65$, as the former is almost the double of the latter.

The addition chains we found matched the results of [7], only with a very minor improvement. Their costs are given in Table 2 for various values of B_1, along with the corresponding timings.

4.3 Stage 2 based on baby-step/giant-step

The idea of the stage 2 strategy is to test, for all primes π between B_1 and another bound B_2, whether π times the point Q coming out from stage 1 is the neutral element. This is achieved in a batch way, where the number of curve operations grows only like the square root of $B_2 - B_1$. Let w be the value taken for the giant-steps: we write all the primes $B_1 < \pi \leq B_2$ as $\pi = vw \pm u$, where

$$u \in U = \left\{ u \in \mathbb{Z} \;\middle|\; 1 \leq u \leq \frac{w}{2},\ \gcd(u, w) = 1 \right\}, \text{ and}$$
$$v \in V = \left\{ v \in \mathbb{Z} \;\middle|\; \left\lceil \frac{B_1}{w} - \frac{1}{2} \right\rceil \leq v \leq \left\lfloor \frac{B_2}{w} + \frac{1}{2} \right\rfloor \right\}.$$

The algorithm then computes all the points $[u]Q$ and $[vw]Q$ for $u \in U$ and $v \in V$. Finally, by constructing appropriate products of scalar based on the coordinates of these points, it is possible to test whether one among all the points is indeed the neutral element with only one GCD with N. This final construction is very similar to the one used in Montgomery's batch inversion, and we refer to [18, 16] for details.

In this setting, it is interesting to take for w a smooth number so that the set U has a small number of elements, which reduces the running time and the memory storage. Furthermore, this number should be around the square root of $B_2 - B_1$. In [16], they choose $B_2 = 16384$, and $w = 2 \cdot 3 \cdot 5 \cdot 7 = 210$. However, we found that, for this value of B_2, it is better to choose $w = 420$, yielding a total cost of 2538 multiplications in $\mathbb{Z}/N\mathbb{Z}$ instead of 2690 with $w = 210$. Similarly, when B_2 increases, it is better to choose larger multiples of 210 for w.

Choices of w for several values of B_1 and B_2 are given in Table 3, along with the corresponding costs and timings.

4.4 Benchmark results

In Tables 2 and 3, we report the number of operations and the measured latency for the two stages of ECM, for a few typical modulus sizes and B_1, B_2 parameters. These benchmarks were run on a single core of a single cluster, so that all the required data fit easily in memory.

For the stage 1, the measured latencies include the cost of a final GCD. This operation and the numerous additions account for the difference observed between the latency of the full stage 1 and the naive estimate obtained by multiplying the number of modular multiplications by the latency of a single modular multiplication as reported in Table 1. According to our measures there seems to be no other significant overhead for the stage 1.

For the stage 2, the reported latencies also include a final GCD. For each B_1, the value of B_2 has been chosen as an integer multiple of 2^{14} such that the number of multiplications required is about the same as in the stage 1. This step is however more memory intensive. This becomes particularly visible in the last two lines of the table. For instance, in the case of $B_1 = 8192$ and $B_2 = 80 \cdot 2^{14}$, the arithmetic cost of the stage 2 is very similar to that of the stage 1 (around 90 k multiplications and as many additions for each stage). However, the measured latency of the stage 2 is about 10 % higher than that of the stage 1. We interpret this as the cost of the cache-misses that must be more frequent with such large values of B_2.

In Table 4, we finally provide benchmarks that are close to what we would have in an NFS context, during the cofactorization step of the relation collection, or during the initialization of a discrete logarithm descent (for the large modulus sizes and values of B_1 and B_2). The 256

Table 2: Cost and measured latencies (in clock cycles) for the stage 1 of ECM.

	Cost			Average latency according to size of modulus N			
B_1	Number of operations (curve ops. and mults.)	Total # of mults.	Difference with [7]	192 bits ($n_W = 6$)	256 bits ($n_W = 8$)	384 bits ($n_W = 12$)	512 bits ($n_W = 16$)
256	$361\,\mathbf{D}' + 38\,\mathbf{A} + 12\,\mathbf{m} = 2843\,\mathbf{m}$		$-1\,\mathbf{m}$	444 k	621 k	1.18 M	1.90 M
512	$739\,\mathbf{D}' + 74\,\mathbf{A} + 21\,\mathbf{m} = 5786\,\mathbf{m}$		$-20\,\mathbf{m}$	894 k	1.25 M	2.39 M	3.83 M
1024	$1473\,\mathbf{D}' + 140\,\mathbf{A} + 37\,\mathbf{m} = 11468\,\mathbf{m}$		$-40\,\mathbf{m}$	1.77 M	2.47 M	4.71 M	7.55 M
8192	$11774\,\mathbf{D}' + 1015\,\mathbf{A} + 192\,\mathbf{m} = 90730\,\mathbf{m}$		$-344\,\mathbf{m}$	13.9 M	19.4 M	37.1 M	59.6 M
32768	$47158\,\mathbf{D}' + 3899\,\mathbf{A} + 647\,\mathbf{m} = 361945\,\mathbf{m}$		——	55.3 M	77.5 M	148 M	237 M

Table 3: Cost and measured latencies (in clock cycles) for the stage 2 of ECM.

			Cost		Average latency according to size of modulus N			
B_1	B_2	w	Number of operations (curve ops. and mults.)	Total # of mults.	192 bits ($n_W = 6$)	256 bits ($n_W = 8$)	384 bits ($n_W = 12$)	512 bits ($n_W = 16$)
256	2^{14}	$2 \cdot 210$	$23\,\mathbf{D} + 69\,\mathbf{A} + 1802\,\mathbf{m} = 2538\,\mathbf{m}$		400 k	561 k	1.07 M	1.72 M
512	$3 \cdot 2^{14}$	$3 \cdot 210$	$43\,\mathbf{D} + 112\,\mathbf{A} + 4572\,\mathbf{m} = 5812\,\mathbf{m}$		913 k	1.28 M	2.44 M	3.93 M
1024	$7 \cdot 2^{14}$	$5 \cdot 210$	$58\,\mathbf{D} + 176\,\mathbf{A} + 9538\,\mathbf{m} = 11410\,\mathbf{m}$		1.80 M	2.52 M	4.79 M	7.71 M
8192	$80 \cdot 2^{14}$	$22 \cdot 210$	$147\,\mathbf{D} + 624\,\mathbf{A} + 84954\,\mathbf{m} = 91122\,\mathbf{m}$		15.4 M	21.1 M	40.1 M	64.1 M
32768	$360 \cdot 2^{14}$	$33 \cdot 210$	$430\,\mathbf{D} + 1148\,\mathbf{A} + 343716\,\mathbf{m} = 356340\,\mathbf{m}$		61.8 M	83.7 M	158 M	252 M

cores of the processor are working in parallel, each core working independently of the others on a particular modulus. The benchmark also includes the time for the data transfer between the I/O subsystems and the compute clusters. The costs for the initialization of the curve and the Montgomery constants for the given modulus are included as well. Not much effort has been put in optimizing these functionalities, and this explains the overhead of about 20 % for the smallest cases ($B_1 = 256$ for 192- and 256-bit moduli) compared to what we would expect by just taking the latencies of Tables 2 and 3 and deducing a lower bound for the throughput. For all the other cases, the overhead compared to the lower bound remains below 10 %. For the largest examples that require a lot of memory, the 16 cores of each cluster are divided into 8 pairs: in each pair, the first core only does stage-1's while the second one only does stage-2's. Since the parameters were chosen so that the two stages take about the same time, we can pipeline a modulus through the two cores of a pair while keeping the additional overhead due to thread synchronization quite low.

During these full benchmarks, the average power consumption reported by the monitoring tools of the MPPA-256 card was 16 W. The "throughput per joule" estimates given in Table 4 are based on this value.

Table 4: Measured throughput (in curves per second and curves per joule) for the full implementation of ECM.

		Average number of curves per second and per joule according to size of modulus N							
B_1	B_2	192 bits ($n_W = 6$)		256 bits ($n_W = 8$)		384 bits ($n_W = 12$)		512 bits ($n_W = 16$)	
256	2^{14}	105 k/s	6.56 k/J	76.6 k/s	4.79 k/J	41.4 k/s	2.59 k/J	25.9 k/s	1.62 k/J
512	$3 \cdot 2^{14}$	52.9 k/s	3.31 k/J	38.1 k/s	2.38 k/J	20.2 k/s	1.26 k/J	12.6 k/s	788 /J
1024	$7 \cdot 2^{14}$	27.6 k/s	1.73 k/J	19.9 k/s	1.24 k/J	10.5 k/s	656 /J	6.53 k/s	408 /J
8192	$80 \cdot 2^{14}$	3.49 k/s	218 /J	2.47 k/s	154 /J	1.22 k/s	76.3 /J	761 /s	47.6 /J
32768	$360 \cdot 2^{14}$	795 /s	49.7 /J	572 /s	35.8 /J	———		———	

4.5 Comparison with other ECM implementations

We have compared our implementation with the ones previously reported in the literature, using two criteria: the number of curves processed per second and the number of curves per joule. Since there is no official price for the MPPA-256 processor, comparisons based on curves per dollar, as done in some articles, were not possible. The results are given in Table 5. For comparing to general-purpose hardware, we used the EECM-MPFQ software which is an adaptation of GMP-ECM targetting specially the sizes considered in the present article. This experiment was run on a machine with two Intel E5-2650 processors, each having 8 cores, with an announced TDP of 95 W each. Thanks to hyperthreading, the best throughput was obtained by running 32 threads in parallel. Due to a different stage-2 strategy, it was not possible to obtain exactly the same value of B_2 as in our implementation, so we set the parameters to get a close enough value.

For GPU-based implementations, we did not run the experiments ourselves but copied the data given in [7, 16] which are the best published results so far for ECM on graphics cards. The implementation of [7] contains only a stage 1, so we extrapolated the throughput of our implementation for such a setting using the data of the line $B_1 = 1024$ and $B_1 = 8192$ of Table 2. This is not very precise but is anyway considered rather obsolete since a stage-2 implementation finds many more prime factors.

From the results in Table 5, it is clear that the general-purpose processors are not well suited: even in terms of pure throughput, modern Intel processors can hardly compete with the MPPA-256 processor, and if the power consumption is taken into account, they are far behind.

The comparison with the GPU implementation is more balanced: a single GPU chip can process 2 to 3 times as many curves per second, depending on the size of the modulus. On the other hand it requires a lot of energy and, in terms of curves per joule, the advantage is clearly on the MPPA-256 side. It must also be noted that our implementation is much more versatile: it is possible to handle much larger B_1, B_2 and sizes of moduli with only a moderate penalty.

5 Conclusion

In this article we have shown how to implement a multiprecision modular arithmetic library for the Kalray MPPA-256 processor for moduli of up to 512 bits, where quadratic multiplication algorithms are well suited. The architecture of the processing engines (the cores) at the heart of this processor proved to be convenient for the task, since in our implementation, the pipelines of all the main execution units remain always busy: no obvious bottleneck could be found that would penalize the efficiency.

On top of this library, we have implemented the ECM algorithm for factoring integers with parameters that are useful for its application in the Number Field Sieve. In this setting, the latency is not an issue and the throughput is the main criterion for comparison. The results are quite satisfactory, with a throughput obtained with the Kalray MPPA-256 processor that is only slightly smaller than for a graphics card, but with a much lower power consumption. Also, the amount of fast memory available for each core is large enough to handle sizes that were not reachable in graphics cards.

References

[1] Kalray. URL `http://www.kalray.eu`

Table 5: Comparison with other ECM implementations for various parameters, in curves per second and curves per joule.

Stage 1 only Platform [ref]	B_1	Size of N	Curves per second	Ratio wrt. this work	Curves per joule	Ratio wrt. this work
GTX580 [7]	960	192 bits	171 k/s	2.96	702 /J	0.19
	8192	192 bits	19.9 k/s	2.70	81 /J	0.18

Stage 1 and stage 2 Platform [ref]	B_1	B_2	Size of N	Curves per second	Ratio wrt. this work	Curves per joule	Ratio wrt. this work
GTX580 [16]	256	2^{14}	192 bits	309 k/s	2.94	1.27 k/J	0.19
			256 bits	180 k/s	2.35	738 /J	0.15
			384 bits	86 k/s	2.08	352 /J	0.14
EECM-MPFQ [4] (dual Intel E5-2650)	256	$\approx 2^{14}$	192 bits	42.7 k/s	0.41	225 /J	0.034
			256 bits	27.8 k/s	0.36	146 /J	0.030
			384 bits	13.9 k/s	0.34	73 /J	0.028
			512 bits	8.65 k/s	0.33	46 /J	0.028
	1024	$\approx 7 \cdot 2^{14}$	192 bits	13.4 k/s	0.49	71 /J	0.041
			256 bits	8.63 k/s	0.43	45 /J	0.036
			384 bits	4.14 k/s	0.39	22 /J	0.034
			512 bits	2.58 k/s	0.40	14 /J	0.034
	8192	$\approx 80 \cdot 2^{14}$	192 bits	1.56 k/s	0.45	8.2 /J	0.038
			256 bits	993 /s	0.40	5.2 /J	0.034
			384 bits	464 /s	0.38	2.5 /J	0.033
			512 bits	288 /s	0.38	1.5 /J	0.031
	32768	$\approx 360 \cdot 2^{14}$	192 bits	372 /s	0.47	2.0 /J	0.040
			256 bits	240 /s	0.42	1.3 /J	0.036

[2] Adrian, D., Bhargavan, K., Durumeric, Z., Gaudry, P., Green, M., Halderman, J.A., Heninger, N., Springall, D., Thomé, E., Valenta, L., VanderSloot, B., Wustrow, E., Zanella-Béguelin, S., Zimmermann, P.: Imperfect forward secrecy: How Diffie-Hellman fails in practice. In: I. Ray, N. Li, C. Kruegel (eds.) CCS'15, pp. 5–17. ACM (2015). DOI 10.1145/2810103.2813707

[3] Bernstein, D.J., Birkner, P., Joye, M., Lange, T., Peters, C.: Twisted Edwards curves. In: S. Vaudenay (ed.) AFRICACRYPT'08, *LNCS*, vol. 5023, pp. 389–405. Springer (2008). DOI 10.1007/978-3-540-68164-9_26

[4] Bernstein, D.J., Birkner, P., Lange, T., Peters, C.: EECM: ECM using Edwards curves (2010). URL `http://eecm.cr.yp.to/`. Software.

[5] Bernstein, D.J., Birkner, P., Lange, T., Peters, C.: ECM using Edwards curves. Mathematics of Computation **82**(282), 1139–1179 (2013). DOI 10.1090/S0025-5718-2012-02633-0

[6] Bernstein, D.J., Lange, T.: Batch NFS. In: A. Joux, A. Youssef (eds.) SAC'14, *LNCS*, vol. 8781, pp. 38–58. Springer (2014). DOI 10.1007/978-3-319-13051-4_3

[7] Bos, J.W., Kleinjung, T.: ECM at work. In: X. Wang, K. Sako (eds.) ASIACRYPT'12, *LNCS*, vol. 7658, pp. 467–484. Springer (2012). DOI 10.1007/978-3-642-34961-4_29

[8] Coppersmith, D.: Modifications to the Number Field Sieve. Journal of Cryptology **6**(3), 169–180 (1993). DOI 10.1007/BF00198464

[9] Dupont de Dinechin, B., Ayrignac, R., Beaucamps, P.E., Couvert, P., Ganne, B., Guironnet de Massas, P., Jacquet, F., Jones, S., Morey Chaisemartin, N., Riss, F., Strudel, T.: A

clustered manycore processor architecture for embedded and accelerated applications. In: HPEC'13. IEEE (2013). DOI 10.1109/HPEC.2013.6670342

[10] Dupont de Dinechin, B., Guironnet de Massas, P., Lager, G., Léger, C., Orgogozo, B., Reybert, J., Strudel, T.: A distributed run-time environment for the Kalray MPPA-256 integrated manycore processor. In: V. Alexandrov, M. Lees, V. Krzhizhanovskaya, J. Dongarra, P.M.A. Sloot (eds.) ICCS'13, *Procedia Computer Science*, vol. 18, pp. 1654–1663. Elsevier (2013). DOI 10.1016/j.procs.2013.05.333

[11] Dussé, S.R., Kaliski Jr., B.S.: A cryptographic library for the Motorola DSP56000. In: I.B. Damgård (ed.) EUROCRYPT'90, *LNCS*, vol. 473, pp. 230–244. Springer (1991). DOI 10.1007/3-540-46877-3_21

[12] Hisil, H., Wong, K.K.H., Carter, G., Dawson, E.: Twisted Edwards curves revisited. In: J. Pieprzyk (ed.) ASIACRYPT'08, *LNCS*, vol. 5350, pp. 326–343. Springer (2008). DOI 10.1007/978-3-540-89255-7_20

[13] Koç, Ç.K., Acar, T., Kaliski Jr., B.S.: Analyzing and comparing Montgomery multiplication algorithms. IEEE Micro **16**(3), 26–33 (1996). DOI 10.1109/40.502403

[14] Lenstra, A.K., Lenstra Jr., H.W. (eds.): The development of the Number Field Sieve, *Lecture Notes in Mathematics*, vol. 1554. Springer (1993). DOI 10.1007/BFb0091534

[15] Lenstra Jr., H.W.: Factoring integers with elliptic curves. Annals of Mathematics **126**(3), 649–673 (1987). DOI 10.2307/1971363

[16] Miele, A., Bos, J.W., Kleinjung, T., Lenstra, A.K.: Cofactorization on graphics processing units. In: L. Batina, M. Robshaw (eds.) CHES'14, *LNCS*, vol. 8731, pp. 335–352. Springer (2014). DOI 10.1007/978-3-662-44709-3_19

[17] Montgomery, P.L.: Modular multiplication without trial division. Mathematics of Computation **44**(170), 519–521 (1985). DOI 10.1090/S0025-5718-1985-0777282-X

[18] Montgomery, P.L.: Speeding the Pollard and elliptic curve methods of factorization. Mathematics of Computation **48**(177), 243–264 (1987). DOI 10.1090/S0025-5718-1987-0866113-7

[19] Zimmermann, P., Dodson, B.: 20 years of ECM. In: F. Hess, S. Pauli, M. Pohst (eds.) ANTS VII, *LNCS*, vol. 4076, pp. 525–542. Springer (2006). DOI 10.1007/11792086_37