The Billion-Mulmod-Per-Second PC

Daniel J. Bernstein[1], Hsueh-Chung Chen[2], Ming-Shing Chen[3], Chen-Mou Cheng[2], Chun-Hung Hsiao[3], Tanja Lange[4], Zong-Cing Lin[2], Bo-Yin Yang[3]

[1] University of Illinois at Chicago
[2] National Taiwan University
[3] Academia Sinica
[4] Technische Universiteit Eindhoven

Abstract. This paper sets new speed records for ECM, the elliptic-curve method of factorization, on several different hardware platforms: GPUs (specifically the NVIDIA GTX), x86 CPUs with SSE2 (specifically the Intel Core 2 and the AMD Phenom), and the Cell (specifically the PlayStation 3 and the PowerXCell 8i). In particular, this paper explains how to carry out more than one billion 192-bit modular multiplications per second on a $2000 personal computer.

1 Introduction

The paper "ECM on Graphics Cards" at Eurocrypt 2009 [6] reported a new implementation of ECM performing 41.88 million 280-bit mulmods per second on an NVIDIA GTX 295 GPU. Here "mulmods" are modular multiplications, and ECM is the elliptic-curve method of factorization, a critical subroutine inside NFS, the number-field sieve. See [6, Section 1] for discussion of the cryptanalytic importance of ECM and NFS.

For comparison, the standard GMP-ECM software package, running simultaneously on all four cores of an Intel Core 2 Quad Q9550 CPU, performs only 14.85 million mulmods per second with the same 280-bit modulus length. The same paper recommended building a $2226 computer with two GTX 295 GPUs and a Core 2 Quad Q6600 CPU, performing in total an astonishing 96.79 million 280-bit mulmods per second.

In this paper we show that GPUs are capable of much higher performance. For example, with 210-bit moduli, the same GTX 295 can carry out 481 million mulmods per second. This example uses a somewhat smaller modulus size than [6], but this change explains only a small part of the tenfold increase in speed.

This paper also sets new ECM speed records on several different CPUs: for example, with 192-bit moduli, a Cell-based IBM BladeCenter QS22 can carry out 334 million mulmods per second; an AMD Phenom II 940 can carry out 202 million mulmods per second (20% more on the same CPU than the ECM software being used in the ongoing RSA-768 factorization project); an Intel Core 2 Quad Q9550 can carry out 114 million mulmods per second; and a low-cost PlayStation 3 can carry out 102 million mulmods per second with slightly larger, 195-bit moduli. Our software is tuned in many platform-specific ways but in every case benefits from systematically exploiting the available parallelism.

We find that GPUs are faster than CPUs, but that the best price-performance ratio is achieved by computers that run CPUs and GPUs simultaneously, as in [6]. Specifically, a computer with one Phenom II 940 CPU and two GTX 295 GPUs costs only about $2000 and handles 1.1 billion 192-bit mulmods per second with our ECM software, several times faster than the best result of [6].

Our GPU software goes beyond the software of [6] not only in speed but also in generality: most importantly, within the range of modulus sizes that are of interest inside NFS-over-ECM, we handle several different sizes, while [6] handled only 280 bits. We expect the same techniques to be useful for other computations bottlenecked by modular multiplication. The traditional example (typically with 1024-bit moduli, larger than in this paper) is RSA, while a much more modern example (typically with similar modulus sizes to this paper) is evaluation of pairings to verify short signatures. Note that for efficiency one must feed many simultaneous computations to the hardware; this is not possible for a laptop carrying out an occasional cryptographic computation, but it is feasible for a busy server bottlenecked by cryptography, and it is very easily achieved inside cryptanalytic computations such as NFS-over-ECM.

Readers not familiar with ECM can find all relevant background in [32], [7], and [6].

2 Today's Computing Hardwares

2.1 X86 and Streaming SIMD Extensions

The 64-bit x86 instruction set (x86-64) is supported by all AMD CPUs since the K8 (Opteron and Athlon 64), some versions of the Intel Pentium 4, and all Intel Core 2 and i7 CPUs. In x86-64 there are sixteen 64-bit general-purpose integer registers (GPRs). Modern x86-64 processors decode the variable-length CISC instructions into RISC-like micro-operations for possibly out-of-order dispatching in 3 unified pipelines (Intel Core) or 3 integer plus 3 floating-point pipelines (AMD).

The GNU Multi-Precision (GMP) library uses the biggest multiplication available, the `MUL` instruction (unsigned $64 \times 64 = 128$-bit) to compute multi-precision integers in a straightforward manner, using 64-bit limbs with native `ADC` (add-with-carry) instructions.

AMD K8 and K10 CPUs sport an impressive integer multiplier that can in theory dispatch a $64 \times 64 = 128$-bit `MUL` once every two cycles with a latency of 4–5 cycles. In practice other bottlenecks — principally the forced use of registers `(RDX,RAX)` for the product and `RAX` for the multiplicand — makes it challenging to average one $64 \times 64 = 128$-bit `MUL` every 3 cycles.

Intel CPUs can only dispatch one $64 \times 64 = 128$-bit `MUL` every 4 cycles, with a latency of 7 cycles. Furthermore, `MUL` uses resources (32-bit multipliers) that conflict with other instructions. Therefore it becomes imperative to consider other approaches to big integer arithmetic than the 64-bit `MUL` instructions, such as the x86 vector instructions below.

SSE2 Instructions All Intel CPUs since the Pentium 4 and all AMD CPUs since the K8 supports the SSE2 (Streaming SIMD Extensions 2) instruction set, where SIMD in turn stands for Single Instruction Multiple Data (performing the same action on many operands). SSE2 instruction set operates on 16 architectural 128-bit registers, called `xmm` [0–15], as packed 8-, 16-, 32- or 64-bit `ints`. The instructions are arcane and highly non-orthogonal:

Load/Store: Between `xmm` and memory or lowest `xmm` unit zero-extended and GPR.
Reorganize Data: Multi-way 16- and 32-bit move called Shuffles (8-bit available in SSSE3 only), and Packing, Unpacking, or Conversion on vector data of different densities.
Logical: AND, OR, NOT, XOR; Shift (packed 16-, 32- or 64-bits) Left, Right Logical and Right Arithmetic; Shift entire `xmm` register right/left byte-wise only.

Arithmetic: Add/subtract on 8-, 16-, 32- and 64-bit integers (including "saturating" versions); multiply of 16-bits (high and low word returns, signed and unsigned, and *fused multiply-multiply-adds*) and 32-bits unsigned; max/min (signed 16-bit, unsigned 8-bit); unsigned averages (8-/16-bit); sum-of-differences for 8-bits. A regular set of arithmetic instructions are available on IEEE-754 single and double floats.

Experiments demonstrate that, on AMD CPUs, integer multiplication uses separate computational resources from the vector instructions. On Intel CPUs the resources conflict.

Both Intel Core and AMD K10 architectures can pipeline most vector instructions with a theoretical throughput of one instruction per cycle. One attractive possibility is vectorized floating-point arithmetic, specifically the `MULPD` (multiply 2 packed doubles — 53 bits of mantissa) instruction, with a radix of 2^{24}. Another attractive possibility is vectorized integer arithmetic, specifically the `PMULUDQ` (packed multiply unsigned doubleword to quadword) instruction, which can do two $32 \times 32 = 64$-bit products every cycle. Of course without intrinsic carry flags for SSE additions, for big integer arithmetic we still need to hand-carry *unsigned* limbs. Still, we are able to go as high as radix 2^{30}, which usually saves a limb, and carrying integral limbs uses shifts and bitmasks, which is no worse than the add-subtract in float limbs.

For completeness, we tested and optimized single-thread assembly code using `MUL`, `PMULUDQ`, and `MULPD` as the principal way to multiply for every x86-64 CPU we have. A simple model predicts, and experiments confirm, that `PMULUDQ` is fastest for Intel and using the 64-bit `MUL` is best for AMD.

Other vector instruction sets on x86 (SSE3, SSSE3, and SSE4) have no further instructions that help big integer arithmetic throughput. Note that the signed $32 \times 32 = 64$ multiply (`PMULDQ`) in SSSE3 cannot be used due to the lack of an arithmetic 64-bit right shift. We expect this to change only when AMD's SSE5 (with fused multiply-adds) come to market.

2.2 Graphics Cards from NVIDIA and CUDA

Today's graphics cards contain powerful graphics processing units (GPUs) to handle the increasing complexity and screen resolution in video games. GPUs have now developed into a powerful, highly parallel computing platform that finds more and more interest outside graphics-processing applications. In cryptography, there have been many attempts of exploiting the computational power of GPUs [10,11,26,31]. In particular, Bernstein *et al.* have explored the possibility of using graphics cards to speed up ECM computation [6]. The interested reader is referred to their paper for a more detailed description of the GPU computing platform and various NVIDIA graphics cards; here we only provide a brief summary of GPU programming. More importantly, we will compare our results with theirs in Section 4. Some of the information here is repeated from [6] to keep this paper self-contained.

Like Bernstein *et al.*, we use NVIDIA's GPUs because they provide a programmer-friendly parallel programming environment called CUDA. A GPU program (`*.cu`) is written in CUDA and compiled into intermediate virtual machine code (`*.ptx`). The NVIDIA driver then converts that into real machine code (`*.cubin`) and loads it to run with appropriate data.

A typical NVIDIA GPU contains several to several tens of streaming multiprocessors (MPs), as depicted in Figure 1. Each MP contains a scheduling and dispatching unit that

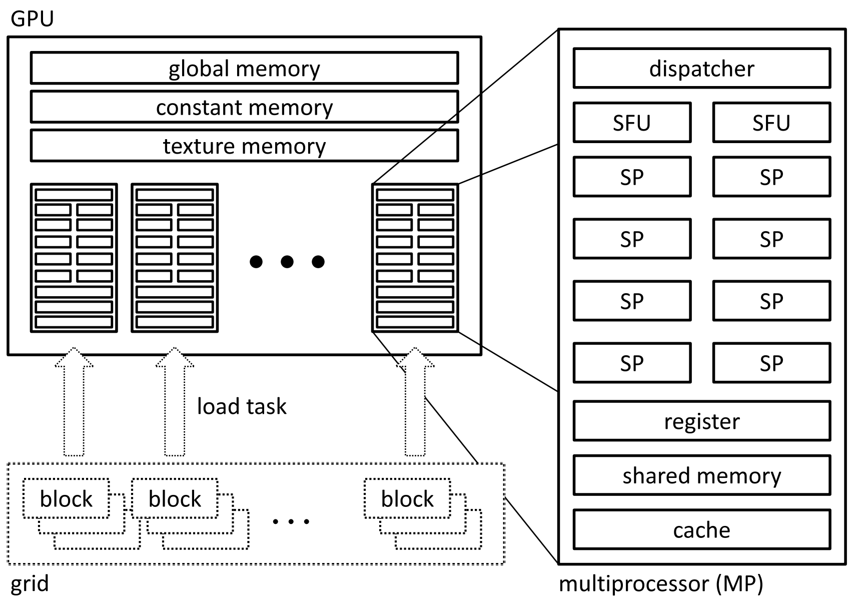

Fig. 1. NVIDIA's GPU Block Diagram

can handle many lightweight threads. The actual computation is done by eight streaming processors (SPs) and two super function units (SFUs) on each MP, and a GPU typically contains tens to hundreds of these SPs, which NVIDIA advertises as "cores."

Like in any other architecture with a memory hierarchy, the closer any memory is to the processor core, the faster it will be. So there are, as shown in Figure 1, a big register file, fast on-die shared memory, fast local read-only caches to device memory on the card, and uncached thread-local and global memories. Note that the NVIDIA platform only provides read-only caching. Programmers are on their own to manage the caching of read-write memories.

Uncached memories have relatively low throughput and long latency. For example, a GeForce 8800 GTX has 128 SPs running at 1.35 GHz; its controllers have a total throughput of only 86.4 GB/s to device memory. This translates to *one* 32-bit floating-point number per cycle per MP, not to mention the latency of 400–600 cycles.

These characteristics mean that GPU programming generally involves controlling a large number of *threads*. The benefits of using many threads are twofold. First, we will be able to exploit *thread-level parallelism* in the application via mapping these threads to the hundreds of SPs in GPU and having them run in parallel. Secondly, having multiple threads time-share a physical SP enables latency hiding, a well-known strategy in the computer architecture community. Only with enough threads can we eliminate wasted clock cycles (caused by dependent pipeline stalls) and fully utilize all computational units available on GPU. In particular, NVIDIA reports the theoretical maximal GFLOPS of their GPUs as if the user can always run enough threads filling up the 20+-stage pipelines of all SPs.

In the CUDA programming model, the threads of an application are organized in a two-tier hierarchy. At top level they form a *thread grid*, which just means that they run one single program called the *kernel*. A grid of threads are divided into *thread blocks*. Threads in the same block can cooperate, while different blocks of threads run independently. A block of threads must be executed by one single MP, which makes intra-block cooperation such as thread synchronization much easier. Thread blocks also make scaling easier: GPUs with more MPs can process more blocks in parallel without changing the program or the kernel configuration.

Even though the CUDA programming model is about the concept of threads, it is essential for the programmer to understand that the minimal scheduling entity is actually a *warp*. In CUDA, the number of threads in a warp depends on microarchitecture implementation; for all current GPUs it is 32. Each instruction is in fact decoded only once a warp. Hence, *each thread in a warp must run the same program (SPMD or same program multiple data)*, controlled by built-in variables (e.g., `ThreadID`). Thus we need at least $8 \times 24/32 = 6$ warps per MP since a float instruction has a latency of 20–24 cycles.

The GPU threads are lightweight hardware threads, which incur very fast context switches with little overhead. To support this, the physical registers are divided among all active threads. For example, on 8800 GTX there are 8192 registers per MP. If we were to use 256 threads, then each thread could use 32 registers — very few for complicated algorithms. *The register pressure can be even greater, as sometimes the conversion of the virtual machine code leaves something to be desired.* GPUs from the GT2xx family have twice as many registers, making things much easier.

To summarize, the massive parallelism in NVIDIA's GPU architecture makes GPU programming very different from sequential programming on traditional CPUs. In general, GPUs are most suitable for executing the data-parallel part of an algorithm. To get the most out of the theoretical arithmetic throughput, one must maximize the ratio of arithmetic operations to memory accesses.

2.3 Cell Processor

The Cell processor was jointly developed by Sony, Toshiba, and IBM in 2005. A Cell processor is constructed from one Power Processing Element (PPE) and eight Synergistic Processor Elements (SPEs) with a high-bandwidth Element Interconnection Bus (EIB), as shown in Figure 2. The processor cores work at clock rates up to 4 GHz, while the EIB works at half of

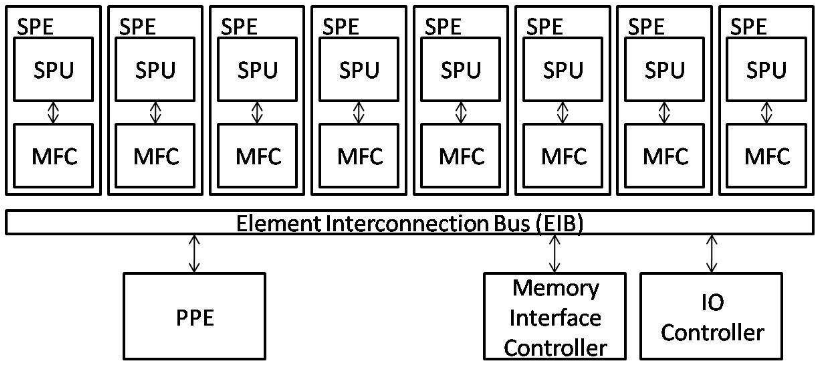
SPE
SPU
MFC
Element Interconnection Bus (EIB)
PPE
Memory Interface Controller
IO Controller

Fig. 2. The Cell Processor Block Diagram

the system clock rate. EIB does not only connects PPE and SPE but also memory and I/O controllers. EIB is composed of four rings, each offering a bandwidth of 16 bytes per cycle, and supports multiple simultaneous transfers per ring. The peak bandwidth of the entire EIB is 96 bytes per cycle. The PPE is more of a conventional PowerPC processor, which supports two-level on-chip caches with multi-threading capability and vector multimedia extensions. The PPE's main task is usually to run the operating system.

Each SPE is composed of a synergistic processor unit (SPU) and a memory flow controller (MFC). The SPU, acting like a RISC processor, is used mainly for computation. Each SPU has a SIMD unit that is capable of vector processing of integer and floating-point numbers of various lengths. The SIMD unit is an important feature for high-performance computing and will be described in more detail later in this section.

The SPU contains a 256-kilobyte local store for instructions and data needed for executing a program on it. Instructions and data must be explicitly moved to the local store by sending direct memory access (DMA) commands to the MFC. The MFC acts like a DMA engine and handles communication between the local SPE core and other cores, main memory, and the I/O controller. The use of DMA allows for efficient use of memory bandwidth and enables overlapping of computation and communication.

Figure 3 shows the block diagram of an SPE. Each SPU has a large 128-entry, 128-bit-

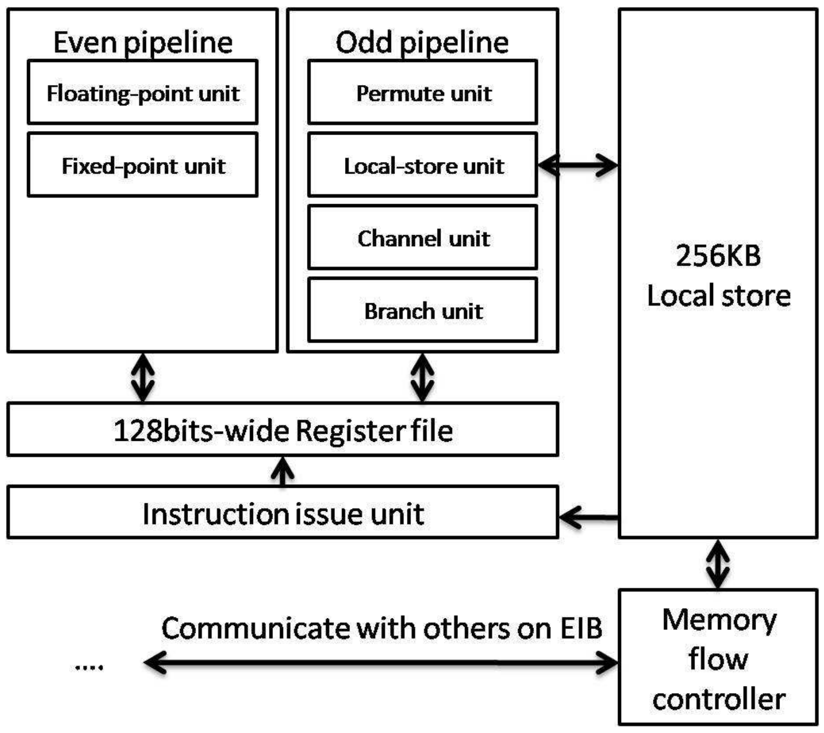

Fig. 3. SPE Block Diagram

wide register file. The flat architecture (all operand types stored in a single register file) of the register file allows for instruction latency hiding without speculation [1]. The SPU has two in-order issued pipelines, called the even and odd pipelines, which can issue and complete up to two instructions per cycle. The two pipelines handle different instruction types, as shown in Figure 3. Roughly, value computation will use the even pipeline while access to local store (including address calculation) will use the odd pipeline. The arithmetic logical units (ALUs) in the SPU are also designed to support 128-bit-wide SIMD operations, which can process up to eight short integer operations, four single-precision floating-point operations, or two double-precision floating-point operations concurrently every cycle.

In 2008, IBM introduced a new variant of the Cell processors, called the PowerXCell 8i. Compared with the previous Cell processors, PowerXCell 8i supports fully-pipelined double-precision floating-point operations. The double-precision peak throughput of a PowerXCell 8i processor is 102 GFLOPS using 8 SPEs, as opposed to 14 GFLOPS with the previous Cell processor. The Roadrunner, currently #1 on the list of Top 500 supercomputers [2], consists of 12960 PowerXCell 8i processors and offers a peak performance of more than 1700000 GFLOPS.

There has been a small amount of previous work in optimizing cryptographic algorithms on the Cell processor. Costigan and Scott published their experience porting SSL to the Cell processor [12]. They use multi-precision math library provided by IBM Cell's SDK on the SPE to implement the kernel operations in SSL, whereas we implement our own multi-precision arithmetic without using IBM's library. Recently, Costigan and Schwabe reported a fast implementation of elliptic curve cryptography based on Curve25519 for the Cell [13]. This curve is defined modulo $2^{255} - 19$ to allow extremely fast reduction. We handle general moduli as required for ECM.

3 Implementation

3.1 Elliptic-curve Arithmetic

We use the windowed double-and-add algorithm to compute the scalar multiples of a point on elliptic curves [6, 7], in which a scalar multiplication is transformed into a sequence of point doublings and additions. To avoid the expensive division operations, we use the projective coordinates to represent the point $Q = (X : Y : Z)$, whereas the starting point P is stored in its affine coordinates (x, y). We choose the standard Edwards coordinates and use the mixed addition law on Edwards curves [21]. The addition law is given by

$$\begin{aligned} X_3 &= Z_1(X_1Y_2 - Y_1X_2)(X_1Y_1 + Z_1^2X_2Y_2) \\ Y_3 &= Z_1(X_1X_2 + Y_1Y_2)(X_1Y_1 - Z_1^2X_2Y_2) \\ Z_3 &= Z_1^2(X_1X_2 + Y_1Y_2)(X_1Y_2 - Y_1X_2), \end{aligned}$$

whereas the doubling law is given by

$$\begin{aligned} X_3 &= ((X_1 + Y_1)^2 - (X_1^2 + Y_1^2))((X_1^2 + Y_1^2) - 2Z_1^2) \\ Y_3 &= (X_1^2 + Y_1^2)(X_1^2 - Y_1^2) \\ Z_3 &= (X_1^2 + Y_1^2)((X_1^2 + Y_1^2) - 2Z_1^2). \end{aligned}$$

Note that the extended Edwards coordinates presented by Hisil et al. [20] save 1 multiplication per addition but require extra storage and scheduling. On several platforms storage is a concern so we did not apply those formulas.

In the windowed double-and-add algorithm, the number of doublings will be equal to the number of bits in the scalar k, while the number of additions will depend on the window size. With a larger window size, a smaller number of additions will be executed during the computation of the scalar multiplication. However, this speed-up comes at a price of extra storage space, i.e., the pre-computed points need to be stored in memory. On modern x86 CPUs, this is not a problem since the on-die caches are usually large enough to hold many pre-computed points. On the Cell processor and GPU, the on-die fast memories are more limited, and we need to store the pre-computed points in off-chip device memories, accessing which involves a high latency. The Cell processor has an architectural design that helps relieve this problem. Namely, with the help of MFC, we are able to store pre-computed points in off-chip main memory rather than in on-die local store. Since the computation time of a point doubling on an elliptic curve is much longer than the transmission time for fetching the next pre-computed point to be used in the subsequent addition (if any), we can overlap computation and communication via the well-known double-buffer mechanism. As a result, our ECM implementation is able to support virtually an arbitrarily large window size. A similar latency-hiding strategy also works on NVIDIA GPU, except that we need to explicitly move the data, as opposed to the use of a DMA memory controller.

3.2 Modular Arithmetic

The kernel operation of ECM is multi-precision integer modular arithmetic, in which an integer is represented using L limbs in radix 2^r with each limb ranging from -2^{r-1} to 2^{r-1} and stored in a single-precision variable. The single-precision arithmetic can be carried out by either fixed-point or floating-point arithmetic, depending on which arithmetic delivers higher throughput on the chosen hardware platform. Such a representation allows us to represent any integer between $-R/2$ and $R/2$, where $R = 2^{Lr}$.

On the x86 CPU, we take advantage of the wide arithmetic pipelines and use 64-bit integer arithmetic aided by XMM arithmetic. On NVIDIA GPU, we use 24-bit integer arithmetic, which in a single cycle can produce the lower 32 bits of the product of two 24-bit integers. We also use full 32-bit addition and subtraction, whose throughput is one every cycle. On the Cell processor, we use 16-bit integer arithmetic, which in a single cycle can produce a 32-bit product. The PowerXCell 8i processor has a better, fully-pipelined double-precision floating-point arithmetic implementation, which we take advantage of and implement the modular arithmetic with.

Stage-1 ECM requires additions, subtractions, and multiplications modulo N, where N is the number to be factored. We use Montgomery's method to avoid trial divisions in computing modular operations [25]. In this method, addition and subtraction modulo N are straightforward, as we can simply add and subtract, respectively, the two operands limb by limb, followed by a carry reduction to bring each limb to its normal range of -2^{r-1} to 2^{r-1}. We note that it is fairly safe to skip a small number of carry reduction steps after an addition or a subtraction because we have some headroom in the storage of the limbs if we do not need to multiply the result immediately.

The modular multiplication is more complicated, as it involves a multiplication step followed by a reduction step. Textbook description says that there are more advanced algorithms, e.g., the Karatsuba multiplication, that would outperform the plain schoolbook multiplication when the number of limbs is in the range that we are interested in. However, as the latter makes better use of the native fused multiply-and-add (MADD) instruction on the Cell processor and GPU, it turns out to be faster in this context and hence becomes the choice of multiplication method in our implementation. On the x86 CPU, since the number of limbs is small, we go to schoolbook directly.

The following step is the modular reduction. Suppose that the two original operands have L limbs with radix 2^r in multiplication step. Multiplication produces a partial product C with $2L$ limbs such that $C = \sum_{i=0}^{2L-1} c_i 2^{ri}$. In the multi-precision case of Montgomery multiplication, we will eliminate the lower half of C by adding a specific multiple of the modulus N sequentially, i.e., making $c'_0 = 0$, $c'_1 = 0, \ldots, c'_{L-1} = 0$ so that after this elimination step, the upper half $c'_{2L-1}2^{r(L-1)} + c'_{2L-2}2^{r(L-2)} + \ldots + c'_L$ will be the result of modular multiplication.

As we have mentioned in Section 3.1, some of the operands of the modular arithmetic operations are stored in off-chip device memories on the Cell processor and GPU. To load these operands incurs a long latency, which we hide via the well-known double-buffer strategy. To support this strategy, each modular arithmetic operation is further broken down into three sub-operations: load, execute, and store. This is similar to the design philosophy of the Reduced Instruction Set Computer (RISC), in which memory latency is exposed to the compiler designers and assembly-language programmers so that they can schedule the instructions properly to hide memory latency via instruction-level parallelism.

One reason why we are able to achieve such a tremendous speed-up over previous results is that we have a different thread organization. Recall that in [6], one modular multiplication is carried out by a group of 28 threads. This same amount of work can be done by fewer threads; in fact, there is a natural way to divide the work equally to be done by k threads as long as k divides the number of limbs n. It is easy to verify that in such a work decomposition, the total number of registers required for a fixed amount of computation roughly remains constant. If one uses less threads to compute a single multiplication, then each thread will use more resources, putting more pressure on, e.g., the fast on-die shared memory. The other hand, each thread will do more work, and hence we will have a higher compute-to-memory-access ratio. In [6], the authors used a design that is on one extreme of the spectrum, namely, n threads to compute an n-limb multiplication. In this paper, we try the other end, namely, one single thread to compute one n-limb multiplication. We are able to achieve a much improved compute-to-memory-access ratio, as well as eliminate inefficiencies due to synchronization overhead and such, resulting in a much improved performance.

3.3 Software Pipelining, Loop Unrolling, and Hyperthreading

ECM is embarrassingly parallelizable and can exploit SIMD by running many curves in parallel. This is always achievable in practice, since trying ECM on a single curve with a large parameter B_1 is not as effective as using the same amount of time to try many curves with a smaller B_1. This is also necessary for GPUs, since fewer threads would not run faster — we would see almost the same speed with many pipeline stalls. The reason of course is that compared to modern processors, the SPs in GPUs do very high latency operations. However, this phenomenon is not limited to GPUs.

We know that modern CPUs often have out-of-order execution and their dispatchers look very hard for ILP (instruction-level parallelism). But sometimes there is just not enough ILP, and all the pipelines would be mostly full of bubbles. In our preliminary implementations we observe some of these situations, especially in the reduction step of Montgomery modular multiplication. For example, on an SPU of the PowerXCell 8i processor, it takes about 900 cycles to complete two Montgomery multiplications simultaneously, using two-way SIMD on double-precision floats, of which 500 are actually wasted due to data-dependent stalls.

An analogous situation happened with an extreme case among modern CPUs, namely the Pentium 4, which has ALU running at a clock twice as fast as the rest of the chip, but with pipelines with 30+ stages. Even with out-of-order execution, it is extraordinarily difficult to fill a pipeline that is even deeper than the GPUs today. The difficulty to locate and use ILP is compounded because there are only six general-purpose registers.

Part of Intel's solution is to make the CPU run two hardware threads. The two threads each have their own set of architectural registers, switching whenever there is a stall. Intel calls this form of symmetric multithreading *hyperthreading*. While it of course can never get close to the 2× speedup from having another core, hyperthreading can achieve fairly impressive gains for certain classes of code.

If there are enough spare registers, both architectural and actual, then we can apply the following strategy to utilize these unused resources as well as the computational power wasted by the pipeline stalls. *We can run more "threads" of computation simultaneously* by interleaving instructions from several flows of independent computations into one single physical thread of instructions. By measuring the percentage of time spent in useful computation, we conclude that such a strategy of combining software pipelining and loop unrolling (SPLU) does work well on Cells.

On x86-64 CPUs, SPLU cannot work as above since they have too few GPRs architecturally. However, a different kind of SPLU is possible in the following sense: Modern x86-64 CPUs have multiple independent pipelines that execute multiple instructions in parallel. Their combined throughput is additive if there is no contention to shared resources such as arithmetic circuitry or external memory. When mixed properly, a sequence of instructions containing a stream of 64-bit integer multiplications (using `MUL` and GPRs) and another stream of 32-bit SIMD integer multiplications (using `PMULUDQ` and XMM registers) can theoretically achieve a throughput close to those combined from two threads executed separately on the latest AMD Phenom IIs. This we call *heterogenous software multi-threading*.

In practice, we are able to speed up our AMD code by 20%+. This agrees with conventional wisdom that the two types of multiplications share no circuitry on an AMD CPU.

Unfortunately this is not the case with Intel CPUs, and the throughput of the combined instruction stream is much lower than the sum of the throughputs had we executed two individual streams. *However, our heterogeneous software hyperthreaded ECM code used for the C2+ still gained more than native hyperthreading when we ran it on the Ci7.*

4 Experimental Results

We measure the performance of our implementations of stage-1 ECM for $B_1 = 8192$ on various hardware platforms and present the experimental results in this section. We have three different families of hardware platforms: GPU, x86 CPU, and Cell. For GPU, we perform our experiments on NVIDIA GTX 295. For x86 CPU, we have AMD Phenom II 940 at 3 GHz (K10+) and Intel Core 2 Quad Q9550 at 2.83 GHz (C2+). For Cell, we have Sony PlayStation 3 (PS3) and IBM BladeCenter QS22, and only the latter supports high-throughput double-precision floating-point arithmetic.

The performance of our latest implementations of stage-1 ECM for these hardware platforms is summarized in Figure 4 and Table 1. We note that in Table 1, since different

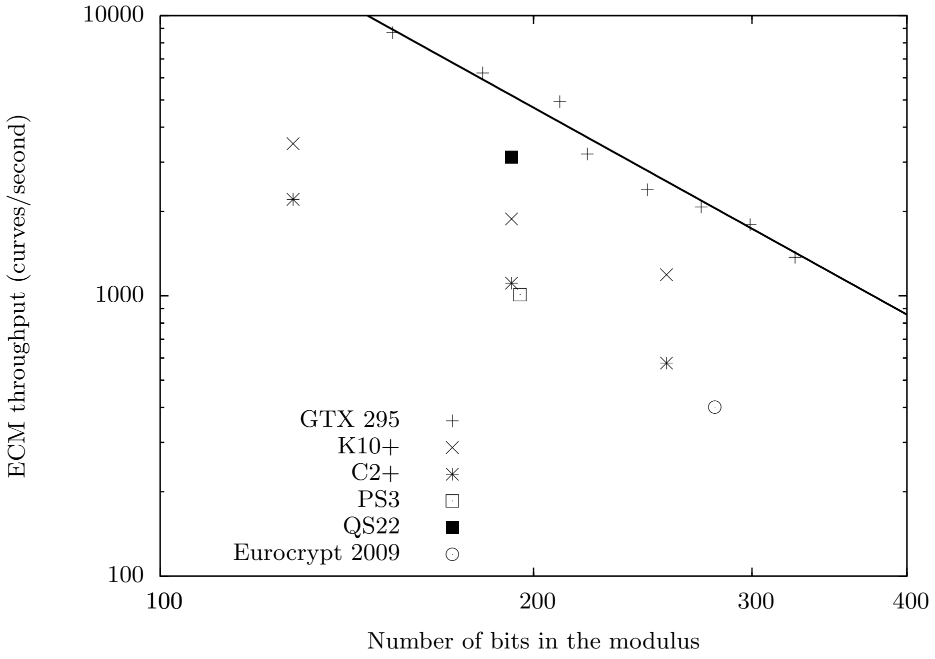

Fig. 4. Performance comparison of stage-1 ECM on various hardware platforms

implementations may use different bit lengths, we scale the results to 192 bits in order to compare their performance. As a rule of thumb, since the bottleneck of the computation is the (schoolbook) multi-precision integer multiplication, whose complexity grows quadratically as the number of bits in the multiplicand, we use the square of the length of the moduli in bits to scale the results. For example, the result of a 280-bit ECM would be scaled by $280^2/192^2$, or roughly a factor of two. Such a scaling ignores factors such as pressure on on-die memories, which can be significant for GPU implementations. As we can see from Figure 4, there are two dips on the GPU curve when we go above 210 bits and 299 bits. These are precisely when we have to reduce the number of thread blocks because we do not have enough fast memories to support as many thread blocks. As a result, the exponent of the linear regression line for GPU result on logarithmic scale is -2.46, showing that the performance actually drops faster than quadratically as we increase the number of bits in the modulus.

It is clear that from Figure 4, the GPUs have the best performance across all modulus lengths. The runner-up is AMD's K10+, whose price-performance ratio is also very compet-

Table 1. Performance results of stage-1 ECM on selected hardware platforms.

	GTX 295	K10+	C2+	QS22	PS3	GTX 295 [6]
#cores	480	4	4	16	6	480
clock (MHz)	1242	3000	2830	3200	3200	1242
price (USD)	500	190	270	$$$	413	500
TDP (watts)	295	125	95	200	<100	295
GFLOPS	1192	6+24	3+23	204	154	1192
#threads	46080	48+16	48+16	160	6	
#bits in moduli	210	192	192	192	195	280
#limbs	15	3+7	3+7	8	15	28
window size (bits)	u6	u6	u6	s5	s5	s4
mulmods (10^6/sec)	481	202	114	334	102	42
curves (1/sec)	4928	1881	1110	3120	1010	401
curves (1/sec, scaled)	5895	1881	1110	3120	1042	853

itive. Since the CPU results are obtained via SPLU, we list two numbers for GFLOPS and #threads, one for 64-bit integer (left) and the other for SIMD units (right). We note that such GFLOPS rating can be misleading since different platforms have different arithmetic pipeline widths, and the wider the pipeline is, the more useful a "FLOP" is, which is evident from the numbers of GPU vs. CPU as well as QS22 vs. PS3.

It is important to note that this gain in computational power does not come at a price of power consumption. The billion-mulmod PC that we recommend can be run on a 850W power supply, whereas we measured from the outlet a K10+ (Phenom II 940) running our code and got 170W. Since our box is more than five times as fast as the K10+, the billion-mulmod-per-second PC is more efficient per watt.

We show the effect of heterogeneous software hyperthreading on x86 CPUs in Figure 5, using 192-bit mulmods on AMD K10+ as an example. In Figure 5, each point represents a way to mix the XMM and integer instructions. We see that some ways of mixing actually result in worse performance than time sharing between XMM and integer units, although most combinations yield some improvements.

The Cell processor is also quite competitive in terms of price-performance ratio, as its price in Table 1 is that of a whole machine, unlike the other platforms for which only the component prices are listed. This is largely due to the fact that Sony is currently subsidizing its PS3 sales, making PS3 an attractive platform for ECM.

References

1. — (no editor), *SDK 3.1 Programming Tutorial*, IBM, 2008. Cited in §2.3.
2. TOP500 Supercomputing Sites, *32nd Top500 list* (2008). URL: `http://www.top500.org/lists/2008/11/`. Cited in §2.3.
3. — (no editor), *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Intel Corp., 2007. URL: `http://www.intel.com/design/processor/manuals/248966.pdf`.
4. — (no editor), *13th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2005), 17–20 April 2005, Napa, CA, USA*, IEEE Computer Society, 2005. ISBN 0-7695-2445-1. See [30].
5. A. O. L. Atkin, Francois Morain, *Finding suitable curves for the elliptic curve method of factorization*, Mathematics of Computation **60** (1993), 399–405. ISSN 0025-5718. MR 93k:11115. URL: `http://www.lix.polytechnique.fr/~morain/Articles/articles.english.html`.

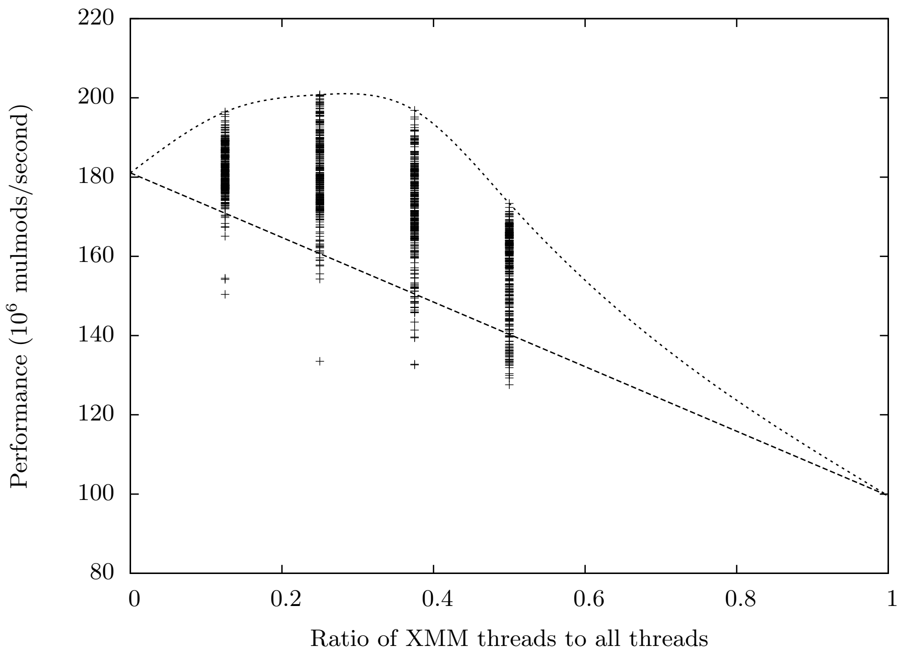

Fig. 5. Effect of Heterogeneous SSMT for 192-bit mulmods on K10+

6. Daniel J. Bernstein, Tien-Ren Chen, Chen-Mou Cheng, Tanja Lange, Bo-Yin Yang, *ECM on Graphics Cards*, in Eurocrypt [22] (2009), 483–501. URL: `http://eprint.iacr.org/2008/480/`. Cited in §1, §1, §1, §1, §1, §1, §1, §1, §2.2, §2.2, §3.2, §3.2, §1.
7. Daniel J. Bernstein, Peter Birkner, Tanja Lange, Christiane Peters, *ECM using Edwards curves* (2008). URL: `http://eprint.iacr.org/2008/016`. Cited in §1.
8. Daniel J. Bernstein, Tanja Lange, *Explicit-formulas database* (2008). URL: `http://hyperelliptic.org/EFD`.
9. Daniel J. Bernstein, Tanja Lange, *Faster addition and doubling on elliptic curves*, in Asiacrypt 2007 [23] (2007), 29–50. URL: `http://cr.yp.to/papers.html#newelliptic`.
10. Debra L. Cook, John Ioannidis, Angelos D. Keromytis, Jake Luck, *CryptoGraphics: Secret Key Cryptography Using Graphics Cards*, in CT-RSA 2005 [24] (2005), 334–350.
11. Debra L. Cook, Angelos D. Keromytis, *CryptoGraphics: Exploiting Graphics Cards For Security*, Advances in Information Security, 20, Springer, 2006. ISBN 978-0-387-29015-7.
12. Neil Costigan, Michael Scott, *Accelerating SSL using the vector processors in IBM's Cell Broadband Engine for Sony's Playstation 3* (2007). URL: `http://eprint.iacr.org/2007/061/`. Cited in §2.3.
13. Neil Costigan, Peter Schwabe, *Fast elliptic-curve cryptography on the Cell Broadband Engine* (2009). URL: `http://eprint.iacr.org/2009/016/`. Cited in §2.3.
14. Harold M. Edwards, *A normal form for elliptic curves*, Bulletin of the American Mathematical Society **44** (2007), 393–422. URL: `http://www.ams.org/bull/2007-44-03/S0273-0979-07-01153-6/home.html`.
15. Jens Franke, Thorsten Kleinjung, Christof Paar, Jan Pelzl, Christine Priplata, Colin Stahlke, *SHARK: A Realizable Special Hardware Sieving Device for Factoring 1024-Bit Integers*, in CHES 2005 [29] (2005), 119–130.
16. Kris Gaj, Soonhak Kwon, Patrick Baier, Paul Kohlbrenner, Hoang Le, Mohammed Khaleeluddin, Ramakrishna Bachimanchi, *Implementing the Elliptic Curve Method of Factoring in Reconfigurable Hardware*, in CHES 2006 [18] (2006), 119–133.
17. Steven D. Galbraith (editor), *Cryptography and Coding, 11th IMA International Conference, Cirencester, UK, December 18–20, 2007*, Lecture Notes in Computer Science, 4887, Springer, 2007. ISBN 978-3-540-77271-2. See [26].

18. Louis Goubin, Mitsuru Matsui (editors), *Cryptographic Hardware and Embedded Systems — CHES 2006, 8th International Workshop, Yokohama, Japan, October 10–13, 2006*, Lecture Notes in Computer Science, 4249, Springer, 2006. ISBN 3-540-46559-6. See [16].
19. Florian Hess, Sebastian Pauli, Michael E. Pohst (editors), *Algorithmic Number Theory, 7th International Symposium, ANTS-VII, Berlin, Germany, July 23–28, 2006*, Lecture Notes in Computer Science, 4076, Springer, Berlin, 2006. ISBN 3-540-36075-1. See [32].
20. Huseyin Hisil, Kenneth Koon-Ho Wong, Gary Carter, Ed Dawson, *Twisted Edwards Curves Revisited*, in Asiacrypt 2008 [28] (2008), 326–242. URL: `http://eprint.iacr.org/2008/552`. Cited in §3.1.
21. Huseyin Hisil, Kenneth Wong, Gary Carter, Ed Dawson, *Faster group operations on elliptic curves* (2007). URL: `http://eprint.iacr.org/2007/441`. Cited in §3.1.
22. Antoine Joux (editor), *Advances in Cryptology - Eurocrypt 2009 (28th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Cologne, Germany, April 26-30, 2009*, Lecture Notes in Computer Science, 5479, Springer, 200. See [6].
23. Kaoru Kurosawa (editor), *Advances in cryptology — ASIACRYPT 2007, 13th International Conference on the Theory and Application of Cryptology and Information Security, Kuching, Malaysia, December 2–6, 2007*, Lecture Notes in Computer Science, 4833, Springer, 2007. See [9].
24. Alfred J. Menezes (editor), *Topics in Cryptology — CT-RSA 2005, The Cryptographers' Track at the RSA Conference 2005, San Francisco, CA, USA, February 14–18, 2005*, Lecture Notes in Computer Science, 3376, Springer, 2005. ISBN 3-540-24399-2. See [10].
25. Peter L. Montgomery, *Modular multiplication without trial division*, Mathematics of Computation **44** (1985), 519–521. URL: `http://www.jstor.org/pss/2007970`. Cited in §3.2.
26. Andrew Moss, Dan Page, Nigel P. Smart, *Toward Acceleration of RSA Using 3D Graphics Hardware*, in Cryptography and Coding 2007 [17] (2007), 364–383.
27. Elisabeth Oswald, Pankaj Rohatgi (editors), *Cryptographic Hardware and Embedded Systems — CHES 2008, 10th International Workshop, Washington, D.C., USA, August 10–13, 2008*, Lecture Notes in Computer Science, 5154, Springer, 2008. ISBN 978-3-540-85052-6. See [31].
nference on the Theory and Application of Cryptology and Information Security, Melbourne, Australia, December 7-11, 2008.
28. Josef Pieprzyk (editor), *Advances in Cryptology - ASIACRYPT 2008, 14th International Co*, Lecture Notes in Computer Science, 5350, Springer, 2008. See [20].
29. Josyula R. Rao, Berk Sunar (editors), *Cryptographic Hardware and Embedded Systems — CHES 2005, 7th International Workshop, Edinburgh, UK, August 29 – September 1, 2005*, Lecture Notes in Computer Science, 3659, Springer, 2005. ISBN 3-540-28474-5. See [15].
30. Martin Šimka, Jan Pelzl, Thorsten Kleinjung, Jens Franke, Christine Priplata, Colin Stahlke, Miloš Drutarovský, Viktor Fischer, *Hardware Factorization Based on Elliptic Curve Method*, in FCCM 2005 [4] (2005), 107–116.
31. Robert Szerwinski, Tim Güneysu, *Exploiting the Power of GPUs for Asymmetric Cryptography*, in CHES 2008 [27] (2008), 79–99.
32. Paul Zimmermann, Bruce Dodson, *20 Years of ECM*, in ANTS 2006 [19] (2006), 525–542. Cited in §1.