Building the Billion-Mulmod PC

Bo-Yin Yang

Institute of Information Science
Academia Sinica
Taipei, Taiwan
by@crypto.tw

September 10, 2009

Acknowledgement

- This is a joint work with
  - Daniel J. Bernstein, University of Illinois at Chicago, USA
  - Chen-Mou (Doug) Cheng, National Taiwan University, Taiwan
  - Tanja Lange, Technische Universiteit Eindhoven, the Netherlands
  - Students
    - Hsueh-Chung Chen and Tien-Ren Chen (GPU)
    - Ming-Shing Chen and Chun-Hung Hsiao (x86 CPU)
    - Zong-Cing Lin (Cell)

Outline

- ECM and how it all Started
- Performance walls for single-threaded processors
- How (many-core) graphics cards come to rescue
- Computing scalar multiplication for ECM using many cores
- Performance results

Elliptic Curve Method of Integer Factorization

- ECM: an important factoring algorithm by itself
- Also critical to 1024+-bit NFS (finding ≈30-bit prime factors in lots of 100–300-bit numbers)
  - 5% sieve time in 1999 RSA-512 factorization
  - Steadily increasing for RSA-576, 640, and 663
  - 33% sieve time in RSA-768 (linear algebra not yet finished)
  - Estimated to account for 50% sieve time for RSA-1024
- Faster ECM will speed up NFS
- Can increase its workload share and *speed up NFS even more*

Pollard's $p-1$ Method

- Assume $n = pq$, where $p-1$ is B_1-powersmooth but $q-1$ is not
- Let $s = \mathrm{lcm}(2, 3, 4, \ldots, B_1)$; then $(p-1)|s$ but $(q-1) \not| s$
- Pick a random a and compute a^s
  - For every a, $a^s \equiv 1 \mod p$ by Fermat's little theorem
  - Conversely, we must have $a^s \not\equiv 1 \mod q$ for some a
  - In this case, $\gcd(a^s - 1, n) = p$

Stage 1 ECM

- Generalization of Pollard's $p - 1$ method
  - From $\mathbb{Z}_p^*$ to elliptic curve groups over $\mathbb{F}_p$
- If a non-torsion point P on elliptic curve $E = E(\mathbb{Q})$ has B_1-powersmooth order mod p but not mod q, then E factors n
  - Take rational function ϕ on E s.t. $\phi(O) = 0$
  - Compute $\gcd(\phi([s]P), n)$, where $s = \mathrm{lcm}(2, 3, 4, \ldots, B_1)$
- Can spend a bit more computation and execute "stage 2" to increase the probability of finding prime factors

CUDA EECM

- Use Edwards curves
- Scalar in non-adjacent form (NAF) with large signed window
- GPU: Multi-precision modular arithmetic coprocessor to CPU
  - In current range, schoolbook multiplication faster than Karatsuba due to efficient use of fused multiply-and-add instruction
  - Montgomery reduction (no trial divisions)

Edwards Curves

- $x^2 + y^2 = 1 + dx^2y^2$
  - $$(x_1, y_1) + (x_2, y_2) = \left(\frac{x_1y_2 + x_2y_1}{1 + dx_1x_2y_1y_2}, \frac{y_1y_2 - x_1x_2}{1 - dx_1x_2y_1y_2} \right)$$
  - Neutral point O is $(0, 1)$
  - $-(x, y) = (-x, y)$
- An elliptic curve in Weierstrass form is birationally equivalent to an Edwards curve if and only if it has an order-4 point
- Using homogeneous coordinates $(X : Y : Z) = (X/Z, Y/Z)$
  - Mixed addition: 9M+6a (Hisil et al., ASIACRYPT'08)
  - Doubling: 4S+3M+6a (Bernstein and Lange, ASIACRYPT'07)

Modular Multiplication

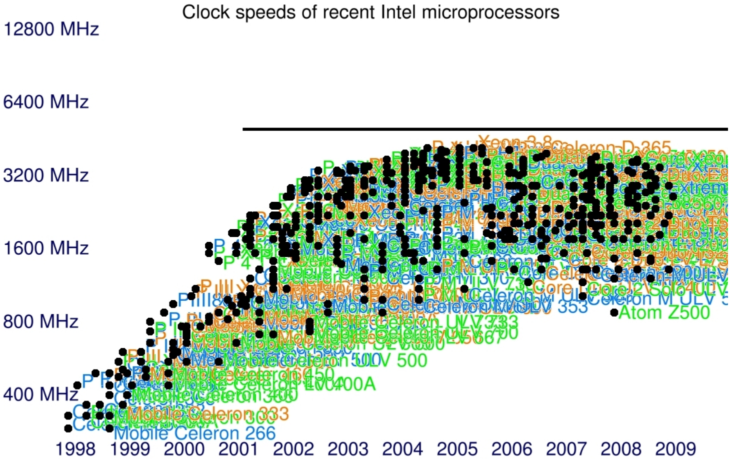
- Bottleneck operation of many cryptosystems
  - Special moduli: ECC over prime fields
  - General moduli: ECM, RSA
- Question: How fast can we do mulmods?
  - More interestingly, how many mulmods/CHF on a PC

Practical Side of Computing

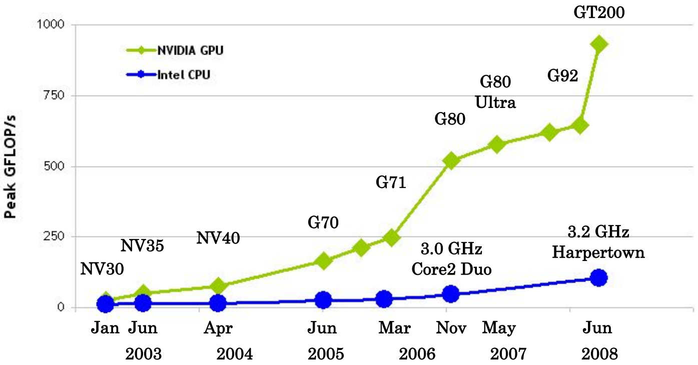
- Moore's law in semiconductor industry
  - Transistor budget doubles every 18–24 months
- In the past, it has been used to increase processor clock frequency

Year	Hi-End CPU	MHz
1979	Z80	2
1984	80286	10
1989	80486	40
1994	Pentium	100
1999	Athlon	750
2004	Pentium 4	3800
2009	Core i7	*3200*

- Runs into: power wall, ILP wall, memory walls


Clock speeds of recent Intel microprocessors
12800 MHz
6400 MHz
3200 MHz
1600 MHz
800 MHz
400 MHz
1998
1999
2000
2001
2002
2003
2004
2005
2006
2007
2008
2009
Atom Z500
Mobile Celeron 266
Mobile Celeron 333

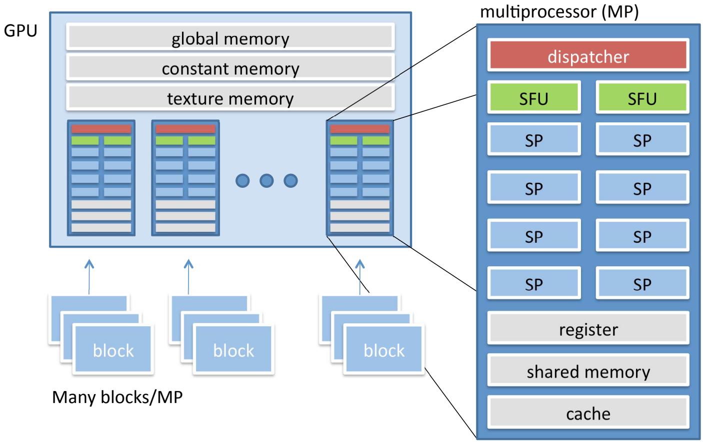
Let's Flog That Dead Horse Some More



Why Are GPUs So Fast?

- Massively parallel paradigm
  - "Many-core" processors
  - Better way of exploiting transistor budget afforded by Moore's law
- Example: NVIDIA's G200b
  - 240 "cores," $>$1.4 billion transistors
  - 470 mm^2, TDP: 204 watts (TSMC 55nm)
  - 1062.72 GFLOPS (single-precision), 159 GB/s memory bandwidth
    - ⋆ Compared with 106.56 GFLOPS, 25.6 GB/s of Intel i7 at 3.33 GHz
- Very, very cheap per GFLOPS, thanks to WoW gamers! :D

NVIDIA's GPU Hardware



Hardware Design

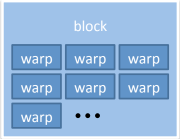
- NVIDIA advertises streaming processors (SPs) as "cores"
- However, an x86 "core" is closer to a streaming multiprocessor (MP)
  - SPs on the same MP share a single dispatcher (instruction decoder)
  - Hence must execute the same instruction (SIMD)
  - More like ALU (arithmetic-logic unit)
- MP: basic building block for NVIDIA's GPUs
  - Scalable design: simply put more MPs into higher-end GPUs
  - Examples: GTX 280: 30 MPs; GTX 260: 24 MPs

How to Program: CUDA and OpenCL

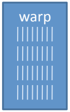
- GPU: data-parallel coprocessor to CPU
  - CPU-GPU communication and synchronization via device memory
- CUDA: Compute Unified Device Architecture
  - NVIDIA's proprietary technology
  - Runs on NVIDIA 8-series and later GPUs
  - Contains compiler, debugger/profiler, run-time environment
- OpenCL: Open Computing Language
  - Industry standardization and extension of GPGPU
  - First draft looks very similar to CUDA
  - Runs on GPUs as well as CPUs
    - NVIDIA released SDK for GPUs
    - AMD/ATI released SDK for CPUs; will release GPU SDK soon

CUDA Programming Model

- Hierarchical thread model
  - Only threads in a block can cooperate and synchronize
  - Each thread block must execute on same MP
- Barrier synchronization
- Explicit caching
  - Via shared memory



Up to 24 warps/block



32 threads/warp

Why So Many Threads?

- Why are there 32 threads in a warp?
  - Dispatcher runs at roughly 25% of the speed of SPs
  - Hence each MP should be regarded as a 32-way SIMD processor
  - The number may change in future GPUs
- Why not just run one single warp on each MP?
  - To exploit thread-level parallelism
  - Need 192 threads to completely hide arithmetic latency
  - Need more to hide device memory latency
- Need a lot of independent threads of computation
  - Elliptic curve method of integer factorization (ECM)
  - Pollard's rho method

Device Memory Latency Hiding Example

```
for (;;) {
    x = devmem[addr]; addr += offset;
    acc += x; x *= x; acc += x;
}
```

- Assume access latency of device memory is 200 cycles
- How many warps do we need to completely hide it?
  - Each SP sees 4 instructions per warp
  - Each memory access sees 4 compute instructions
  - Hence need $\lceil 200/16 \rceil = 13$ warps (416 threads)

GPU-based Modular Arithmetic Coprocessor

- Design goals
  - Fast
  - Flexible
  - Scalable
- Challenges
  - Easy-to-use yet powerful interface
  - Efficiency/processor utilization
  - Synchronization
  - Resource management
  - ...

Task Decomposition and Assignment, First Try

- The EUROCRYPT'09 way
  - k threads cooperate in computing a k-limb multiplication
  - Each thread must read operands, multiply, read accumulator, add the product to it, and write back to shared memory
  - Lots of headaches in solving race conditions
  - Lots of time wasted in memory I/O and synchronization
  - Need to hand-optimize thread organization for different k's

And Our Dear Friend TK. . .

- Informed us that his CPU code is better than our GPU code (Bernstein *et al.*, "ECM on graphics cards," EUROCRYPT 2009)

Date	Bits	Raw	@192	Comments
Sep., 2008		13	28	GMP-ECM (C2@2.4GHz)
	280	23	48	GPU-ECM (GTX 280)
Jan., 2009		42	89	GPU-ECM (GTX 295)
Feb., 2009	192		62	TK-ECM (C2@2.4GHz)
			164	TK-ECM (K10+@3GHz)

Back at the Drawing Board

- SIMD, RISC-like interface
  - Not suitable for scalar applications
  - Expose memory latency to programmer
- Task decomposition and assignment
  - Single-thread-does-it-all to minimize synchronization overhead
- Memory management
  - Operand compaction
  - Explicit caching
- Interface Design:
  - SIMD: Single Instruction Massive Data
    - Work on several thousands of integers simultaneously
    - Enough threads of computation to fully occupy large GPUs
  - RISC: Reduced Instruction Set Computer
    - All computation happens between "registers"
    - Explicit load/store operations for data movement
    - Expose memory latency to programmer

Example Code Fragment

Operation	Actual operation	Working set
D= z1 × y2	load(z1,z2);	z1,z2
E=A × B	load(A,B);	z1,z2,A,B
F=C × D	gpuMULT(D, z1,y2);	z1,D,A,B
G=E+F	load(C);	C,D,A,B
H=E-F	gpuMULT(E, A,B);	C,D,E,B
	gpuMULT(F, C,D);	F,D,E,B
	gpuADD(G, E,F);	F,G,E,B
	gpuSUB(H, E,F);	...

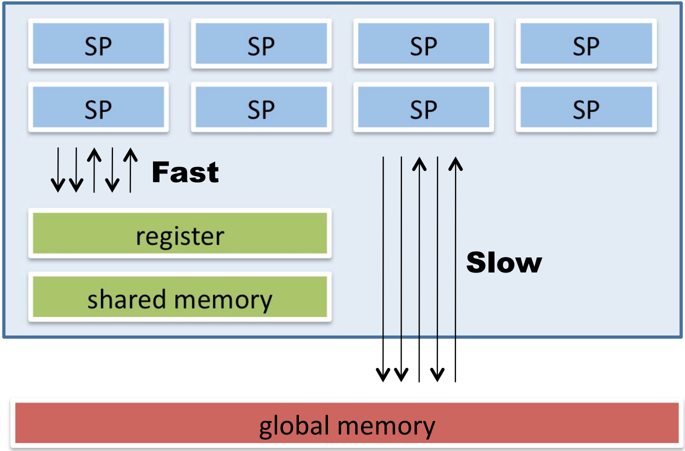
Task Decomposition and Assignment, Second Try

- The new, improved way
  - One thread to run them all
  - Minimizes synchronization overhead
  - Maximizes compute-to-memory ratio
  - Automatic generation and tuning of code for various modulus sizes
- However, this is like dumping garbage into your neighbor's backyard
  - Now using k times more memory per thread
  - The challenge goes to memory management

Memory Management

- Storage compaction
- Double buffering
- Explicit cache management
  - Use shared memory as caches of device memory
  - Can store many pre-computed points
  - Allow use of a large window in NAF
- Cons: More programming complications

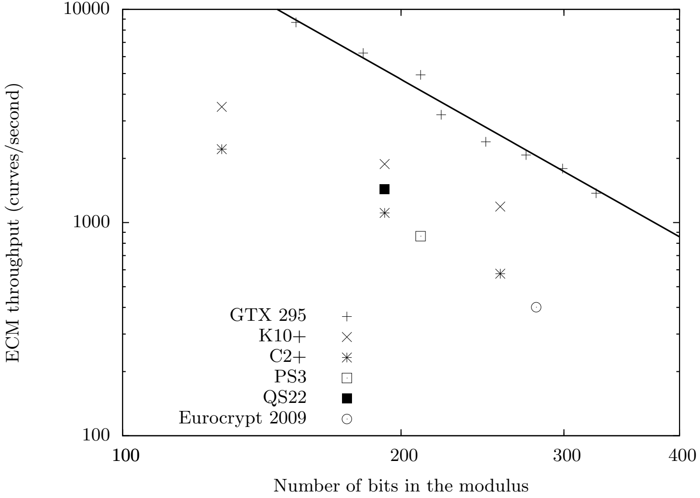
Explicit Cache Management



Performance Results

- Curves per second
- Million mulmods per second
- Price-performance ratio
- Performance per watt

Performance Comparison



Performance Comparison

	GTX 295	K10+	C2+	QS22	PS3	[1]
#cores	480	4	4	16	6	480
clock (MHz)	1242	3000	2830	3200	3200	1242
price (USD)	470	170	220	$$$	413	470
TDP (watts)	295	95	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

[1] Bernstein et al. “ECM on graphics cards.” EUROCRYPT 2009.

Back to our CPU performance

- It comes down to implementing 192-bit Montgomery mulmods
- We used Dan's `qhasm` and tricks and got 15% speed-up
- However, TK gets about 65 cycles/mulmod (21 `MULs`) on a K10+
- We can get 66–67 cycles/mulmod after multiple loop unrolling
- This is not acceptable
- We started looking for *any* improvement

Software Thread Integration

- Modern CPUs tend to be underutilized
  - Wide arithmetic pipelines
  - Vector instructions engines
  - Superscalar, i.e., can issue multiple independent instructions per cycle
- How to squeeze out the last bits of performance?
  - Run many "threads" of execution simultaneously
  - Exploit as much available circuitry as possible
  - Similar to why we must have so many threads on GPU
  - We hand-merged integer code and vector code
- Result: 22% speed-up on AMD CPUs
  - Compared with Kleinjung's code using 64-bit integer multiplications
  - Also works on Cell (240%)
  - Doesn't work so well for Intel CPUs ($< 10\%$)

Putting together a 10^9-mulmods/second PC

- Buy parts, say, from NewEgg.COM (8/24/2009 2PM)

ITEM	USD	Description	Notes
CPU	170	AMD Phenom II 945 (3.0 GHz)	retail K10+
MB	170	ASUS M4N82 Deluxe	ECC-capable
RAM	140	4x DDR2-800 ECC Kingston 2GB	
GPUs	940	2x PNY 1792MB NVIDIA GTX 295	
Case	360	Supermicro 4U with 865W PS	5x fans
HDD	100	2x Seagate SATA II 320GB	RAID 1

- Total: 1880 USD for 1.3 billion 192-bit mulmods per second

More is not necessarily better

- Trap: Don't think you can do three-four cards per PC easy.
- Problem 1: from reputable manufacturers, only more expensive Intel motherboards have enough PCIe x16 slots.
- Problem 2: (We learned the hard way during the Engineyard Challenge) you need spacing between cards.
- Kids who want to try 3 cards at home: buy this (ASUS P6T7 WS Supercomputer) and a 1200W power.

Concluding Remarks

- Many-core processors are becoming mainstream
  - Cheap, available off-the-shelf
  - Good for throughput-oriented computations
- Requires a different kind of thinking from sequential programming
- Watch out for memory management
  - "*All programming is an exercise in caching*"

Thanks for Listening!

- Questions or comments?