Usable assembly language for GPUs: a success story

Daniel J. Bernstein, UIC
Hsieh-Chung Chen, Harvard
Chen-Mou Cheng, NTU
Tanja Lange, Eindhoven
Ruben Niederhagen, E+AS
Peter Schwabe, Academia Sinica
Bo-Yin Yang, Academia Sinica
OpenSSL crypto library (version 1.0.1, 2012.03) includes 15 different asm implementations of AES.

SHA-3-512 implementations benchmarked in eBASH:
17 for blake512,
25 for keccakc1024,
15 for groestl512,
7 for round3jh512,
12 for skein512512.

Widespread use of asm in the fastest implementations.
Why not just one portable implementation? What do compilers do wrong?

- Instruction selection: e.g., compiler doesn’t see how to use vector instructions.
- Instruction scheduling.
- Register allocation.

Can blame programming language for hiding critical information. Increasing gap between common languages and hardware.
NVIDIA GTX 295 graphics card:
Massively parallel—2 GPUs, 60 cores, 480 32-bit ALUs. (NVIDIA marketing terminology: 60 “MPs”; 480 “cores”.)
Massively vectorized—
1 slow instruction decoder/core.
Relatively little fast RAM—
16384 bytes “shared mem”/core. (Newer “Fermi” GPUs: better.)
Can C compilers use GPUs?
No! Intolerable slowdown.
NVIDIA solution: Change the programming language. Tweaked “CUDA” version of C. ( “OpenCL” variant of CUDA is also supported by AMD.)

CUDA programs explicitly state parallelization, vectorization. Eliminates biggest problem in instruction selection.

But the NVIDIA compilers still have big trouble with register allocation.
Case study: ECC2K-130—
“infeasible” ECDLP challenge
posed in 1997 by Certicom.

Optimized attack
(see paper for references):
\[ \approx 2^{60.9} \text{ iterations of} \]
\[(x, y) \mapsto (x', y'); \text{ many in parallel.} \]

\(x, y\) are in \(\mathbb{F}_{2^{131}}\);
\(x\) has even Hamming weight \(w\);
\(j = 3 + ((w/2) \mod 8)\);
\(\lambda = (y + y^{2^j})/(x + x^{2^j})\);
\(x' = \lambda^2 + \lambda + x + x^{2^j}\);
\(y' = \lambda(x + x') + x' + y\).
$\approx 70000$ bit ops/iteration with best techniques known;
$\approx 2^{77}$ bit ops overall.

Main cost (85% of bit ops):
5 poly mults/iteration,
$131 \times 131 \rightarrow 261$ bits.
\[ \approx 70000 \text{ bit ops/iteration} \]

with best techniques known;

\[ \approx 2^{77} \text{ bit ops overall.} \]

Main cost (85% of bit ops):

5 poly mults/iteration,

\[ 131 \times 131 \rightarrow 261 \text{ bits.} \]

Compare to theoretical capacity of one GTX 295: 60 cores,
each 256 bit ops/cycle,

\[ 1.242 \cdot 10^9 \text{ cycles/second} \]

\[ \Rightarrow 2^{70} \text{ bit ops in 2 years.} \]

64 dual-GTX-295 PCs:

\[ 2^{77} \text{ bit ops in 2 years.} \]
This comparison assumes that 100% of GPU time is spent on useful bit operations.

We try writing CUDA code, feed it to NVIDIA’s `nvcc`. Experiment extensively with “optimizations” to CUDA code.
This comparison assumes that 100% of GPU time is spent on useful bit operations.

We try writing CUDA code, feed it to NVIDIA’s nvcc. Experiment extensively with “optimizations” to CUDA code.

10× slower than theory!
What’s going wrong?
This comparison assumes that 100% of GPU time is spent on useful bit operations.

We try writing CUDA code, feed it to NVIDIA’s nvcc. Experiment extensively with “optimizations” to CUDA code.

10× slower than theory! What’s going wrong?

nvcc is constantly running out of registers, spilling to “local memory”; huge cost. (Less huge on Fermi.)
NVIDIA has ptxas assembler, documents “PTX” instruction set. (Recent NVIDIA nvcc releases support inline PTX in CUDA.)

Great! Rewrite code in PTX, paying attention to regs.
NVIDIA has ptxas assembler, documents “PTX” instruction set. (Recent NVIDIA nvcc releases support inline PTX in CUDA.)

Great! Rewrite code in PTX, paying attention to regs.

10× slower than theory! What’s going wrong?
NVIDIA has ptxas assembler, documents “PTX” instruction set. (Recent NVIDIA nvcc releases support inline PTX in CUDA.)

Great! Rewrite code in PTX, paying attention to regs.

10× slower than theory!
What’s going wrong?

PTX isn’t the machine language. ptxas is the actual compiler: converts to SSA, re-assigns regs, spills to expensive local memory.
2007 van der Laan reverse-engineered binaries, wrote decuda tool to print machine language in a readable format. (NVIDIA now supports this.)

Also cudasm to convert readable format back to machine language.

But this was “tedious” and hampered by cudasm bugs: “we must extract minimum region of binary code needed to be modified and keep remaining binary code unchanged . . . implementation of cudasm is not entirely complete, it is not a good idea to write whole assembly manually and rely on cudasm.”
But this was “tedious” and hampered by cudasm bugs: “we must extract minimum region of binary code needed to be modified and keep remaining binary code unchanged . . . implementation of cudasm is not entirely complete, it is not a good idea to write whole assembly manually and rely on cudasm.”

Not a serious obstacle! We fixed various bugs and now use cudasm to generate our GTX 295 code.
Everybody knows that writing in asm is painful.

Maybe the most painful part: have to manually assign live values to registers.

Our fix: qhasm-cudasm. **Usable** asm for GPUs.
Everybody knows that writing in asm is painful.

Maybe the most painful part: have to manually assign live values to registers.

Our fix: qhasm-cudasm. *Usable* asm for GPUs.

The old parts: cudasm; qhasm toolkit for parsing and smart register allocation.
Everybody knows that writing in asm is painful.

Maybe the most painful part: have to manually assign live values to registers.

Our fix: qhasm-cudasm. *Usable* asm for GPUs.

The old parts: cudasm; qhasm toolkit for parsing and smart register allocation.

New: usable syntax for the GPU instructions.
C/C++/CUDA:
z2 = x2 ^ y2;

PTX:
xor.b32 %r24, %r22, %r23;

cudasm:
xor.b32 $r2, $r3, $r2

qhasm-cudasm:
z2 = x2 ^ y2

See paper for many detailed examples.
low32 threadinfo
input threadinfo

enter Z9kerneladdPjPKjS_
low32 tid
low32 x
low32 y
low32 t
low32 tstart
low32 tend
low32 ttid
low32 tselected
low32 now
tselected = 0
low32 j
low32 twenty
cond testloop

tid = 65535 & threadinfo
cond tid12
tid12 tid - c[0]
low32 tid4
tid4 = tid << 2
offset tid4off
tid4off = tid << 2

low32 batchshift
batchshift = blockindex
batchshift int24* = 33536
x = parameters[0]
y = parameters[1]
t = parameters[2]

x += batchshift
y += batchshift

low32 0x0
low32 0x1
low32 0x2
low32 0x2
low32 0x3
low32 0x4
low32 0pos

syncthreads
new 0x4

0pos = tid4 + x
0x0 = g[0pos]
0pos += 512
0x1 = g[0pos]
0pos += 512
0x2 = g[0pos]
0pos += 512
0x3 = g[0pos]
0pos += 512
0x4 = g[0pos] if tid12 signed<

s[tid4off + 512] = 0x0
s[tid4off + 1024] = 0x1
s[tid4off + 1536] = 0x2
s[tid4off + 2048] = 0x3
s[tid4off + 2560] = 0x4 if tid12 signed

low32 1x0
low32 1x1
low32 1x2
low32 1x3
low32 1x4
low32 1pos

syncthreads

ew 1x4
1pos = tid4 + y
1x0 = g[1pos]
1pos += 512
1x1 = g[1pos]
1pos += 512
1x2 = g[1pos]
1pos += 512
1x3 = g[1pos]
1pos += 512
1x4 = g[1pos] if tid12 signed<

s[tid4off + 2608] = 1x0
s[tid4off + 3120] = 1x1
s[tid4off + 3632] = 1x2
s[tid4off + 4144] = 1x3

s[tid4off + 4656] = 1x4 if tid12 signed

syncthreads

j = 0
twenty = 20
syncthreads
tstart = halfclock
low32 2x0
low32 2x1
low32 2x2
low32 2x3
low32 2x4
low32 0y0
low32 0y1
low32 0y2
low32 0y3
low32 0y4

new 2x4
new 0y4

2x0 = s[tid4off + 512]
2x1 = s[tid4off + 1024]
2x2 = s[tid4off + 1536]
2x3 = s[tid4off + 2048]
2x4 = s[tid4off + 2560] if tid12 signed<

0y0 = s[tid4off + 2608]
0y1 = s[tid4off + 3120]
0y2 = s[tid4off + 3632]
0y3 = s[tid4off + 4144]
0y4 = s[tid4off + 4656] if tid12 signed

2x0 ^= 0y0

2x1 ^= 0y1

2x2 ^= 0y2

2x3 ^= 0y3

2x4 ^= 0y4

s[tid4off + 512] = 2x0

s[tid4off + 1024] = 2x1

s[tid4off + 1536] = 2x2

s[tid4off + 2048] = 2x3

s[tid4off + 2560] = 2x4 if tid12 signed

syncthreads

tend = halfclock
tend -= tstart

tend <<= 1

low32 3x0

low32 3x1

low32 3x2

low32 3x3

low32 3x4

low32 2pos

syncthreads

new 3x4

3x0 = s[tid4off + 512]

3x1 = s[tid4off + 1024]
\[ 3x2 = s[tid4off + 1536] \]
\[ 3x3 = s[tid4off + 2048] \]
\[ 3x4 = s[tid4off + 2560] \text{ if } \text{tid12 signed} < \]

\[ 2pos = \text{tid4} + x \]
\[ g[2pos] = 3x0 \]
\[ 2pos += 512 \]
\[ g[2pos] = 3x1 \]
\[ 2pos += 512 \]
\[ g[2pos] = 3x2 \]
\[ 2pos += 512 \]
\[ g[2pos] = 3x3 \]
\[ 2pos += 512 \]
\[ g[2pos] = 3x4 \text{ if } \text{tid12 signed} < \]
ttid = tid << 2

ttid += t

g[ttid] = tend

leave