Usable assembly language for GPUs

D. J. Bernstein
University of Illinois at Chicago

---

319 ms: `rpes/src/cuda`

183 ms: `rpes/src/qhasm` (new)

Measured on `behemoth`:
1.30GHz GTX 280 $\times 2$;
2.83GHz Core 2 Quad Q9550

1974 Knuth:
"There is no doubt that the 'grail' of efficiency leads to abuse. Programmers waste enormous amounts of time thinking about, or worrying about, the speed of noncritical parts of their programs, and these attempts at efficiency actually have a strong negative impact when debugging and maintenance are considered. We *should* forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil."

Computer isn't fast enough.
You've measured performance, identified the lines of software taking most of the CPU time.
Now what?

Computer isn't fast enough.
You've measured performance,
identified the lines of software
taking most of the CPU time.
Now what?

One traditional answer:
"Find a faster algorithm."

Computer isn't fast enough.
You've measured performance,
identified the lines of software
taking most of the CPU time.
Now what?

One traditional answer:
"Find a faster algorithm."

You've tried many algorithms.
Tried many software rewrites.
Computer is still too slow.
Now what?

Another traditional answer:

Rewrite critical lines in asm.

Another traditional answer:
Rewrite critical lines in asm.

Disadvantage of this answer:
increase in programming time;
asm is hard to use.

Another traditional answer:
Rewrite critical lines in asm.

Disadvantage of this answer:
increase in programming time;
asm is hard to use.

Advantage of this answer:
full control over CPU!
Programmer can control
details of memory layout,
instruction selection,
instruction scheduling, etc.

Compiler can be quite stupid:
often fails to exploit CPU,
even with programmer's help.

Yet another answer:
Move critical lines to a GPU.

Most common GPU architectures:
Evergreen, Northern Islands from AMD; Tesla, Fermi from NVIDIA.
In this talk I'll focus on
the Tesla GPU architecture.
Tesla GPUs are very easy to find:
GTX 280; GTX 295; AC;
Lincoln; Longhorn; etc.

Yet another answer:
Move critical lines to a GPU.

Most common GPU architectures:
Evergreen, Northern Islands from AMD; Tesla, Fermi from NVIDIA.
In this talk I'll focus on
the Tesla GPU architecture.
Tesla GPUs are very easy to find:
GTX 280; GTX 295; AC;
[redacted]; Longhorn; etc.

Yet another answer:
Move critical lines to a GPU.

Most common GPU architectures:
Evergreen, Northern Islands from AMD; Tesla, Fermi from NVIDIA.
In this talk I'll focus on
the Tesla GPU architecture.
Tesla GPUs are very easy to find:
GTX 280; GTX 295; AC;
~~Lincoln~~; Longhorn; etc.

Advantage of this answer:
GPU can do huge number of
floating-point operations/second.

GPU is tough optimization target.
Highly parallelized, vectorized:
30 cores ("multiprocessors")
running $\geq$ 3840 threads;
each instruction is applied
to vector of $\geq$ 32 floats.

GPU is tough optimization target.
Highly parallelized, vectorized:
30 cores ("multiprocessors")
running $\geq$ 3840 threads;
each instruction is applied
to vector of $\geq$ 32 floats.

Maybe NVIDIA makes up for this
with super-smart compilers
that fully exploit the GPU!

GPU is tough optimization target.
Highly parallelized, vectorized:
30 cores ("multiprocessors")
running $\geq$ 3840 threads;
each instruction is applied
to vector of $\geq$ 32 floats.

Maybe NVIDIA makes up for this
with super-smart compilers
that fully exploit the GPU!
Maybe not.

GPU is tough optimization target.
Highly parallelized, vectorized:
30 cores ("multiprocessors")
running $\geq$ 3840 threads;
each instruction is applied
to vector of $\geq$ 32 floats.

Maybe NVIDIA makes up for this
with super-smart compilers
that fully exploit the GPU!
Maybe not.

Move critical lines to a GPU
and write them in asm?
This is easier said than done.

2010 L.-S. Chien "Hand-tuned SGEMM on GT200 GPU":

Successfully gained speed using van der Laan's `decuda`, `cudasm` and manually rewriting a small section of `ptxas` output.

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 ... it is not a good idea to write whole assembly manually and rely on `cudasm`."

2010 Bernstein–Chen–Cheng–Lange–Niederhagen–Schwabe–Yang "ECC2K-130 on NVIDIA GPUs"; focusing on GTX 295:

Extensive optimizations in CUDA for "ECC2K-130" computation: 26 million iterations/second.

Built new assembly language `qhasm-cudasm` for Tesla GPUs. Built 90000-instruction kernel entirely in assembly language; later reduced below 10000. 63 million iterations/second for the same computation.

My talk today: Another `qhasm-cudasm` case study.

2010.11 email from Kindratenko:

`rpes kernel in particular is of a very much interest to us because it is similar to some of the kernels Alex has implemented. ... We would be very much interested in understanding how this kernel can be re-implemented in the nvidia gpu assembly language that you have developed and what benefits this would give us.`

1953 Tom Lehrer “Lobachevsky”:

“I am never forget the day I am
given first original paper to write.
It was on
analytic and algebraic topology of
locally Euclidean parameterization
of infinitely differentiable
Riemannian manifold.
Bozhe moi!
This I know from nothing.
What I am going to do.”

Download `rpes` in `parboil1`.
Find three implementations
of the same computation:
`base`, `cuda_base`, `cuda`.
Note: no `rpes` in `parboil2`;
and TeraChem source isn't public.

`./parboil run rpes cuda`
`default -S`: 319 milliseconds
= 147 ms on one GPU
+ 94 ms on one CPU core
+ 78 ms copying data.

`cuda_base`: slower.
`base`: 63075 ms; no GPU.

Read code to understand it.

`base` has only 600 lines.

`CalcOnHost` in `base`:
46-line main computation
inside eight nested loops.

Main computation
loads data, does some arithmetic,
calls a few simple subroutines:
e.g., `H_dist2` computes
$(x_1-x_2)^2+(y_1-y_2)^2+(z_1-z_2)^2$.

Also one complicated subroutine,
`root1f`, 75 lines, computing
$\mathrm{erf}(\sqrt{x})/\sqrt{4x/\pi}$ given x.

Sample input used by `parboil`:

x, y, z coordinates for 30 atoms:
20 H atoms and 10 O atoms.

Each O atom has 17 "primitives"
(α, c) organized into 3 shells.
Same primitives, shells for each O:
3rd shell is always $(0.3023, 1)$;
2nd shell is 8 primitives starting
$(11720, -0.00031444 3412)$; etc.

H atom: 4 primitives in 2 shells.

Overall input data:
250 vectors (x, y, z, α, c)
organized into 70 shells.

Have $250^4 = 3906250000$
ways to choose four vectors
$v_1 = (x_1, y_1, z_1, \alpha_1, c_1)$,
$v_2 = (x_2, y_2, z_2, \alpha_2, c_2)$,
$v_3 = (x_3, y_3, z_3, \alpha_3, c_3)$,
$v_4 = (x_4, y_4, z_4, \alpha_4, c_4)$
out of this input.

46-line main computation uses
< 100 floating-point ops
to compute an "integral"
$P(v_1, v_2, v_3, v_4)$
given a choice of four vectors.

$70^4 = 24010000$ choices of
$w_1 = (x_1, y_1, z_1, \mathsf{shell}_1)$,
$w_2 = (x_2, y_2, z_2, \mathsf{shell}_2)$,
$w_3 = (x_3, y_3, z_3, \mathsf{shell}_3)$,
$w_4 = (x_4, y_4, z_4, \mathsf{shell}_4)$.
Define $S(w_1, w_2, w_3, w_4)$
as $\sum P(v_1, v_2, v_3, v_4)$.

Output of computation:
70^4 floats $S(w_1, w_2, w_3, w_4)$.

Actually, `rpes` computes only
$70 \cdot 71 \cdot 72 \cdot 73/24 = 1088430$ floats
from 187240905 integrals:
apparently there's a symmetry
between w_1, w_2, w_3, w_4.

GPU has 240 32-bit ALUs
(arithmetic-logic units;
mislabelled "cores" by NVIDIA).
Each ALU: one op per cycle;
$1.3 \cdot 10^9$ cycles per second.

In `cuda`'s 319 ms:
GPU can do $10.0 \cdot 10^{10}$ ops,
as complicated as multiply-add.
In 146 ms: $4.3 \cdot 10^{10}$ ops.

GPU is actually computing
187240905 integrals,
each < 100 ops:
total $< 1.9 \cdot 10^{10}$ ops.
ALUs are sitting mostly idle!

So I wrote a new `rpes`
using `qhasm-cudasm`.

Integrated into `parboil1`,
put online for you to try:

```
wget http://cr.yp.to/qhasm/
    parboilrpes.tar.gz
tar -xzf parboilrpes.tar.gz
cd parboilrpes
(x=‘pwd‘; cd common/src;
    make PARBOIL_ROOT=$x)
./parboil run rpes cuda
    default -S
./parboil run rpes qhasm
    default
```

Typical code in cudasm:

```
add.rn.f32 $r1, $r20, -$r21
mul.rn.f32 $r6, $r1, $r1
add.rn.f32 $r1, $r24, -$r25
mad.rn.f32 $r6, $r1, $r1, $r6
add.rn.f32 $r1, $r28, -$r29
mad.rn.f32 $r6, $r1, $r1, $r6
```

These instructions work without any of our cudasm bug fixes.

Same code in C/C++/CUDA:

```
dx12 = x1 - x2;
dy12 = y1 - y2;
dz12 = z1 - z2;
dist12 = dx12 * dx12
  + dy12 * dy12 + dz12 * dz12;
```

Compiler selects instructions
(e.g., `mad` for `*+`);
schedules instructions;
assigns registers.

Same in `qhasm-cudasm`:

```
dx12 = approx x1 - x2
dy12 = approx y1 - y2
dz12 = approx z1 - z2
dist12 = approx dx12 * dx12
approx dist12 += dy12 * dy12
approx dist12 += dz12 * dz12
```

Each line is an instruction.
Programmer *can* assign
some or all registers,
but `qhasm` includes a
state-of-the-art allocator.

CUDA:

```
w = 31.00627668 * rsqrtf(X);
```

qhasm-cudasm:

```
w = approx 1 / sqrt X
w = approx w * 31.00627668
```

cudasm:

```
rsqrt.f32 $r7, $r7
mul.rn.f32 $r7, $r7, 0x41f80cdb
```

Start 7680 threads on GPU:
30 blocks of 256 threads;
i.e., 256 threads on each core.

Split the 1088430 outputs
across these threads:
thread t computes outputs
t, $t + 7680$, $t + 15360$, etc.

Start 7680 threads on GPU:
30 blocks of 256 threads;
i.e., 256 threads on each core.

Split the 1088430 outputs
across these threads:
thread t computes outputs
t, $t + 7680$, $t + 15360$, etc.

Oops, imbalance: slowest thread
computes 50341 integrals;
average computes < 25000.
GPU is 50% idle!

Start 7680 threads on GPU:
30 blocks of 256 threads;
i.e., 256 threads on each core.

Split the 1088430 outputs
across these threads:
thread t computes outputs
t, $t + 7680$, $t + 15360$, etc.

Oops, imbalance: slowest thread
computes 50341 integrals;
average computes < 25000.
GPU is 50% idle!

Easy fix, not implemented yet:
sort shells by # primitives.
Reduces penalty to $\approx 10\%$.

Each GPU core has SRAM:
16384 32-bit registers
split between threads;
16384 bytes “shared memory”
accessible by all threads.

CPU copies atom data from
CPU DRAM to GPU DRAM.
GPU DRAM is very slow,
so threads begin by copying
atom data to shared memory.

Threads also initialize shared
`erfseries[X][i]` as
$\sum_j (-1)^j \binom{j}{i} (X/16)^{j-i} / j!(2j+1)$
so that

$(\sqrt{\pi}/2) \operatorname{erf} \sqrt{x+\epsilon} / \sqrt{x+\epsilon}$
$= \sum_i \texttt{erfseries}[16x][i] \epsilon^i.$

(Tweak: $2\pi^{2.5}$ scaling.)

$i \leq 7$ is adequate
for full `float` precision.
Maybe even overkill
for the application.