Lecture 5: HW1 Discussion, Intro to GPUs

G63.2011.002/G22.2945.001 · October 5, 2010
Outline

Discuss HW1

Intro to GPU Computing
Outline

Discuss HW1

Intro to GPU Computing
Dense Matrix Multiply: Blocking vs Scalar

We provided a blocked example matrix multiplication code. Why is blocked matmul faster than un-blocked?

**Key:** Computational Intensity

*Definition:* Flops per FPN moved up the memory hierarchy

Large intensity: good for deep memory hierarchies
Computational Intensity for Scalar Matmul

Floating Point operations: $2N^3$

Assume: $\text{Size}(L1) \ll N^2$ FPNs

$N^2$ read each row of $A$ once
$+ N^3$ read each column of $B N$ times
$+ 2N^2$ read/write $C$

$N^3 + 3N^2$ FPN-size cache misses

(neglecting cache lines, etc.)

*Computational Intensity:* about 2

Discuss HW1 Intro to GPU Computing
Computational Intensity for Blocked Matmul

Floating Point operations: still $2N^3$

$b$: block size \hspace{1cm} $n$: \lceil N/b \rceil$

\[
\begin{align*}
&b^2n^3 \quad \text{read each block of } A \quad n^3 \text{ times} \\
&+ b^2n^3 \quad \text{same for } B \\
&+ 2N^2 \quad \text{read/write } C \\
\end{align*}
\]

\[
\frac{2b^2n^3 + 2N^2}{2b^2n^3 + 2N^2} \quad \text{FPN-size cache misses}
\]

Rewrite:

\[
b^2n^3 \approx b^2 \frac{N^3}{b^3} = \frac{N^3}{b}
\]

Computational Intensity:

\[
\frac{2N^3}{2N^3/b + 2N^2} \approx \frac{2N^3}{2N^3/b} = b
\]

→ incentive to choose $b \gg 2$
Computational Intensity for Blocked Matmul

Floating Point operations: still $2N^3$

- $b$: block size
- $n$: $\lceil N/b \rceil$

\[
b^2 n^3 \quad \text{read each block of } A \ n^3 \ \text{times} \\
+ b^2 n^3 \quad \text{same for } B \\
+ 2N^2 \quad \text{read/write } C \\
\frac{2b^2 n^3 + 2N^2}{2b^2 n^3 + 2N^2} \quad \text{FPN-size cache misses}
\]

Rewrite:

\[
b^2 n^3 \approx b^2 \frac{N^3}{b^3} = \frac{N^3}{b}
\]

Computational Intensity:

The power of assumptions:
Can we choose $b = N$?

→ incentive to choose $b \gg 2$
Consider each level of the memory hierarchy.

How do we exploit . . .

- \textbf{. . . L2:} Ignore—we’re nearly L2-local at most sizes.
  
  - \textbf{. . . L1:} 32 KiB = 4096 Floats.
    Key: memory layout.

  - \textbf{. . . registers:} 16 FP registers.
    Key: loop/operation ordering.
Optimizing for L1: Memory Layout

Memory layout of $A$: column-major.

Only use one entry of each cache line per fetch.

Better to store $A$ in row-major order.

Input is row-major. If memory available (not swap!), storing a transposed copy of $A$ can be a good idea. (Copy takes $O(N^2)$ time.)
Memory layout of $A$: column-major.

Only use one entry of each cache line per fetch.

Better to store $A$ in row-major order.

Input is row-major. If memory available (not swap!), storing a transposed copy of $A$ can be a good idea. (Copy takes $O(N^2)$ time.)
Optimizing for L1: Reuse Pattern, Block Size

Question
Blocking: good idea. Optimal $b_{L1}$?

Follow-up question:
How much needs to fit in L1?
Question

Blocking: good idea. Optimal $b_{L1}$?

Follow-up question:
How much needs to fit in L1?

One block of each of $A$, $B$, $C$. 
Optimizing for L1: Reuse Pattern, Block Size

Question
Blocking: good idea. Optimal $b_{L1}$?

Follow-up question:
How much needs to fit in L1?

One block of each of $A$, $B$, $C$. 

Discuss HW1 Intro to GPU Computing
Question

Blocking: good idea. Optimal $b_{L1}$?

Follow-up question:
How much needs to fit in L1?

One block of each of $A$, $B$, $C$.
All of $A$, plus one column of $B$ and $C$.
32 kiB: $8b_{L1}^2 + 2 \cdot 8b_{L1} \rightarrow b_{L1} \leq 60$
Further concerns:
- Cache line boundaries
- SIMD
- Cache set conflicts

All solved by small-block copy optimization.

Copy all of $A$.
Copy $b_{L1}$-sized blocks of $A$, $B$, and $C$, operate on those, then copy output back.
L1 Block Copy: The Plan

Basic plan:

For each $i$:
  For each $j$:
    Load Block $C[i,j]$  
    For each $k$:
      Load Block $A[i,k]$  
      Load Block $B[k,j]$  
      $\lceil b_{L1}/b_r \rceil^3$ register kernels:
        $C+ = AB$
      Store Block $C[i,j]$
    (can be improved: many $A, B$ loads)

Aside: Also neatly deals with fringes.

So: how does this solve the problems above?

Can you define "alignment"?

Discuss HW1 Intro to GPU Computing
**L1 Block Copy: The Plan**

**Basic plan:**

For each $i$:
  For each $j$:
    Load Block $C[i,j]$
  For each $k$:
    Load Block $A[i,k]$
    Load Block $B[k,j]$
  $\ceil{b_{L1}/b_r}^3$ register kernels:
  $C += AB$

Aside: Also neatly deals with fringes.

So: how does this solve the problems above? Can you define “alignment”??
Alignment

A memory address \(a\) is \(n\)-byte *aligned* when \(n\) is a power of two and \(a\) is a multiple of \(n\) bytes. (see also IBM devWorks article)

```
#include <stdlib.h>

/* dynamic allocation */
double * __attribute__((aligned(64))) var;

int error = posix_memalign(
    (void **) &var, 64, array_size );

if ( error )
    abort();

/* static allocation */
double __attribute__((aligned(64))) ary2 [500];
```

Examples: Cache-line-aligned, SIMD-aligned.

Discuss HW1 Intro to GPU Computing
Alignment

A memory address $a$ is $n$-byte aligned when $n$ is a power of two and $a$ is a multiple of $n$ bytes. (see also IBM devWorks article)

```c
#include <stdlib.h>

/* dynamic allocation */
double * __attribute__((aligned(64))) var;

int error = posix_memalign((void **) &var, 64, array_size);

if (error)
    abort();
```

Examples: Cache-line-aligned, SIMD-aligned.

Code generation in the non-aligned case?
Register Kernel

Choose block size $b_r = 2^k$, with $b_{L1} \mod b_r = 0$.

```c
for (int j = 0; j < b_r; ++j)
    for (int k = 0; k < b_r; ++k)
        for (int i = 0; i < b_r; ++i)
            C[i+j*b_l1] += A[i+k*b_l1] * B[k+j*b_l1];
```

For each $Ab$ matvec:
   Perform $b_r$ scalar-vector updates.

- Vectorizable
- Pipeline-friendly (min. data dependencies)
- Access to $A$, $C$ unit-stride
- Access to $B$ is inner-loop invariant
- Unrolling, software pipelining: Compiler

Discuss HW1 Intro to GPU Computing
Psychoanalyzing the Compiler

Flags for Intel:
- O3 -fno-alias -funroll-loops
- std=c99 -D_XOPEN_SOURCE=500
- opt-streaming-stores auto -static
- fast -xHost

Flags for GCC:
- O3 -funroll-loops -march=native
- std=c99 -D_XOPEN_SOURCE=500
- ftree-vectorizer-verbose=2
- ffast-math

GCC 4.3 sometimes better than GCC 4.4.

Self-study material:
- Compiler Reference: Intel GNU
- C99 restrict keyword, Aliasing
**OProfile**: A sampling profiler. Uses Performance counters. Linux only, needs root.
Profiler

Many event types countable:

CPU_CLK_UNHALTED : Clock cycles when not halted
L2_RQSTS : number of L2 cache requests
LLC_MISSES : L2 cache demand requests from this core that missed the L2
FLOPS : number of FP computational micro-ops executed
IDLE_DURING_DIV : cycles divider is busy and all other execution units are idle.
L1D_ALL_REF : All references to the L1 data cache
L1D_PEND_MISS : Total number of outstanding L1 data cache misses at any cycle
IFU_MEM_STALL : cycles instruction fetch pipe is stalled
INST RETIRED : number of instructions retired
UOPs RETIRED : number of UOPs retired
MACHINE_NUKES_SMC : number of pipeline flushing events
RATSTALLS : Partial register stall cycles
BR_INST_DECORED : number of branch instructions decoded

Only, needs root.
**OProfile**: A sampling profiler. Uses Performance counters. Linux only, needs root.
Profiling

<table>
<thead>
<tr>
<th>FLOPS</th>
<th>L1D_PEND_MISS</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2.6e-04</td>
</tr>
<tr>
<td>187</td>
<td>0.0062</td>
</tr>
<tr>
<td>7</td>
<td>2.3e-04</td>
</tr>
<tr>
<td>470</td>
<td>0.0155</td>
</tr>
<tr>
<td>49</td>
<td>0.0016</td>
</tr>
<tr>
<td>2873</td>
<td>0.0950</td>
</tr>
<tr>
<td>434</td>
<td>0.0144</td>
</tr>
<tr>
<td>184312</td>
<td>6.0959</td>
</tr>
<tr>
<td>2022</td>
<td>0.0669</td>
</tr>
<tr>
<td>19</td>
<td>6.3e-04</td>
</tr>
<tr>
<td>5294</td>
<td>0.1751</td>
</tr>
<tr>
<td>31888</td>
<td>1.0547</td>
</tr>
<tr>
<td>66032</td>
<td>2.1839</td>
</tr>
<tr>
<td>114001</td>
<td>3.7704</td>
</tr>
<tr>
<td>1131</td>
<td>0.0374</td>
</tr>
<tr>
<td>11913</td>
<td>0.3940</td>
</tr>
<tr>
<td>94565</td>
<td>3.1276</td>
</tr>
<tr>
<td>108501</td>
<td>3.5885</td>
</tr>
<tr>
<td>4</td>
<td>1.3e-04</td>
</tr>
<tr>
<td>76622</td>
<td>2.5342</td>
</tr>
<tr>
<td>82075</td>
<td>2.7145</td>
</tr>
<tr>
<td>119036</td>
<td>3.9370</td>
</tr>
<tr>
<td>5</td>
<td>1.7e-04</td>
</tr>
<tr>
<td>2700</td>
<td>0.0893</td>
</tr>
<tr>
<td>14861</td>
<td>0.4915</td>
</tr>
</tbody>
</table>
**Profiling**

**OProfile**: A sampling profiler. Uses Performance counters. Linux only, needs root.
git clone ssh://git@forge.tiker.net:2234/hw1-solution.git
(Private, works if you signed up for an account.)
Great—but:
Most BLAS lose out to triple-loops for special-case matrices.

Want to see code of a “real” BLAS?
GotoBLAS2

git clone
ssh://git@forge.tiker.net:2234/hw1-solution.git
(Private, works if you signed up for an account.)
Key Messages of HW1

In HPC:

- Very simple things quickly become rather complex.
- Need: ideas, careful analysis.
- Flexibility $\leftrightarrow$ performance
- Run-time code generation can be useful.

This class helps by introducing

- known tricks
- helpful tools.
Key Messages of HW1

In HPC:
- Very simple things quickly become rather complex.
- Need: ideas, careful analysis.
- Flexibility ↔ performance
- Run-time code generation can be useful.

Matmul is a “microcosm” of single-proc optimization.

Do not worry if you did not figure out the tricks here on your own.
Questions?
Outline

Discuss HW1

Intro to GPU Computing
GPUs: System Context

- Processor
- Memory

Discuss HW1 Intro to GPU Computing
GPUs: System Context

Expansion Slots

PCI-Express (x4, x16, x1, x16) and regular PCI

PCle V2, x16 Bandwidth:
\[ \sim 6 \text{ GB/s} \]
GPUs: System Context

Expansion Slots
- PCI-Express (x4, x16, x1, x16)
- and regular PCI
- PCIe V2, x16 Bandwidth: ∼6 GB/s

GPU goes here

Discuss HW1: Intro to GPU Computing
GPUs: System Context

- Processor
- Memory
- Expansion Slots: PCIe V2, x16 Bandwidth: \( \sim 6 \text{ GB/s} \)

Discuss HW1 Intro to GPU Computing
GPU Computing?

- Design target for CPUs:
  - Make a single thread very fast
  - Take control away from programmer
- GPU Computing takes a different approach:
  - Throughput matters—single threads do not
  - Give explicit control to programmer
"CPU-style" Cores

Fetch/Decode
ALU (Execute)
Execution Context
Out-of-order control logic
Fancy branch predictor
Memory pre-fetcher
Data cache (A big one)

Credit: Kayvon Fatahalian (Stanford)
Idea #1:
Remove components that help a single instruction stream run fast

Credit: Kayvon Fatahalian (Stanford)
More Space: Double the Number of Cores

Credit: Kayvon Fatahalian (Stanford)
... again

Credit: Kayvon Fatahalian (Stanford)
... and again

16 cores = 16 simultaneous instruction streams

→ 16 independent instruction streams

Reality: instruction streams not actually very different/independent

Credit: Kayvon Fatahalian (Stanford)
Saving Yet More Space

Recall: simple processing core

Fetch/
Decode

ALU
(Execute)

Execution
Context

Idea #2
Amortize cost/complexity of
managing an instruction stream
→ SIMD

Credit: Kayvon Fatahalian (Stanford)
Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ SIMD
Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ SIMD

Credit: Kayvon Fatahalian (Stanford)
Idea #2
Amortize cost/complexity of managing an instruction stream across many ALUs
→ SIMD
Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams
Great if everybody in a group does the same thing.
But what if not?
What leads to divergent instruction streams?

Discuss HW1 Intro to GPU Computing
Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams

Credit: Kayvon Fatahalian (Stanford)
Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams

Great if everybody in a group does the same thing.
But what if not?
What leads to divergent instruction streams?
But what about branches?

```c
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

Credit: Kayvon Fatahalian (Stanford)
But what about branches?

\[
\text{ALU 1} \quad \text{ALU 2} \quad \ldots \quad \ldots \quad \text{ALU 8}
\]

\[
\begin{array}{cccccccc}
1 & 2 & \ldots & & & & 8 \\
\hline
T & T & F & T & F & F & F & F
\end{array}
\]

\[
\text{if } (x > 0) \{ \\
\quad y = \text{pow}(x, \ exp); \\
\quad y *= Ks; \\
\quad \text{refl} = y + Ka; \\
\} \text{ else } \\
\quad x = 0; \\
\quad \text{refl} = Ka; \\
\}
\]

Credit: Kayvon Fatahalian (Stanford)
But what about branches?

```
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

Not all ALUs do useful work! Worst case: 1/8 performance

Credit: Kayvon Fatahalian (Stanford)
But what about branches?

```cpp
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```

Credit: Kayvon Fatahalian (Stanford)
Remaining Problem: Slow Memory

Problem
Memory still has very high latency... but we’ve removed most of the hardware that helps us deal with that.

We’ve removed
- caches
- branch prediction
- out-of-order execution

So what now?

Idea #3
Even more parallelism
+ Some extra memory
= A solution!

Discuss HW1 Intro to GPU Computing
Remaining Problem: Slow Memory

Problem
Memory still has very high latency... but we’ve removed most of the hardware that helps us deal with that.

We’ve removed
- caches
- branch prediction
- out-of-order execution

So what now?

Idea #3
Even more parallelism
+ Some extra memory
= A solution!

Discuss HW1 | Intro to GPU Computing
Remaining Problem: Slow Memory

Problem

Memory still has very high latency.

...but we've removed most of the hardware that helps us deal with that.

We've removed:
- caches
- branch prediction
- out-of-order execution

So what now?

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/

Idea #3

Even more parallelism

+ Some extra memory

= A solution!
Remaining Problem: Slow Memory

Memory still has very high latency. . .
. . . but we've removed most of the hardware that helps us deal with that.

- caches
- branch prediction
- out-of-order execution

So what now?

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/

Hiding shader stalls

Time (clocks)

Fetch/Decode

ALU

ALU

ALU

ALU

ALU

ALU

ALU

ALU

Frag 1 … 8  Frag 9… 16  Frag 17 … 24  Frag 25 … 32

3

4

Idea #3

Even more parallelism

+ Some extra memory

= A solution!
Hiding Memory Latency

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Time (clocks)

Runnable

Stall

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Credit: Kayvon Fatahalian (Stanford)

Discuss HW1 Intro to GPU Computing
Hiding Memory Latency

Time (clocks)

Frag 1 … 8
Frag 9… 16
Frag 17 … 24
Frag 25 … 32

Stall

Runnable

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Increase run time of one group
To maximum throughput of many groups

Credit: Kayvon Fatahalian (Stanford)
GPU Architecture Summary

Core Ideas:

1. Many slimmed down cores → lots of parallelism

2. More ALUs, Fewer Control Units

3. Avoid memory stalls by interleaving execution of SIMD groups

Credit: Kayvon Fatahalian (Stanford)
**GPU-CPU Bird’s Eye Comparison**

**Floorplan: VIA Isaiah (2008)**
65 nm, 4 SP ops at a time, 1 MiB L2.

**Floorplan: AMD RV770 (2008)**
55 nm, 800 SP ops at a time.
Nvidia GTX200

- 4× Fetch/Decode
- ALU
- ALU
- ALU
- ALU
- DP ALU
- 32 kiB Ctx Private
- 16 kiB Ctx Shared

150 GB/s Off-chip Memory

Discuss HW1 Intro to GPU Computing
GPU Architecture (e.g. Nvidia GT200)

- 1 GPU = 30 SIMD cores
- 1 SIMD core: $32 \times 32$ PCs, HW Sched + 1 ID (1/4 clock) + 8 SP + 1 DP + 16 KiB Shared + 32 KiB Reg
- Device ↔ RAM: **140 GB/s**
- Device ↔ Host: **6 GB/s**
- User manages memory hierarchy
What is OpenCL?

OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- Comes with RTCG

Defines:
- Host-side programming interface (library)
- Device-side programming language (!)
Questions?
Discuss HW1 Intro to GPU Computing