Lecture 5

Performance programming for stencil methods
Vectorization
Computing with GPUs
Announcements

• Forge accounts: set up ssh public key, tcsh
• Turnin was enabled for Programming Lab #1: due at 9pm today, report by 9am tomorrow
• Don’t wait until the last minute to turnin: be sure you have all required files: teameval.txt, etc.

• A2: GPU programming
  • 2 parts, each has a turnin
Today’s lecture

• Performance programming for stencil methods
• Vectorization
• Computing with Graphical Processing Units (GPUs)
Motivating application

- Solve Laplace’s equation in 3-D with Dirichlet Boundary conditions
  \[ \Delta u = 0, \quad u = f \text{ on } \partial \Omega \]
- Building block: iterative solver using Jacobi’s method (7-point stencil)

```plaintext
for (i,j,k) in 1:N x 1:N x 1:N
    u'[i,j,k] = (u[i-1,j,k] + u[i+1,j,k] +
               u[i,j-1,k] + u[i,j+1,k] +
               u[i,j,k+1] + u[i,j,k-1]    ) / 6.0
```
Memory representation and access

Linear array space

3D grid

top  center  bottom

U

U'

Linear array space

$B(l \pm 1, J \pm 1)$

$U'$

$U$

$U'$

CACHE (array layout causes conflicts)
Processor Geometry
AMD Phenom’s memory hierarchy

**Dedicated L1**
- Locality keeps most critical data in the L1 cache
- Lowest latency
- 2 loads per cycle

**Dedicated L2**
- Sized to accommodate the majority of working sets today
- Dedicated to eliminate conflicts common in shared caches

**Shared L3 – NEW**
- Victim-cache architecture maximizes efficiency of cache hierarchy
- Fills from L3 leave likely shared lines in the L3
- Sharing-aware replacement policy

6MB L3 on cseclass 01 and 02

NUMA awareness

• When we allocate memory, a NUMA processor uses the *first-touch policy*, unless told otherwise

• Be sure to use *parallel for* when initializing data

```java
x = new double[n], y, z = new double[n];
#pragma omp parallel for
for (i=0; i<n; i++) {x[i]=0; y[i] = …; z[i] = …}
#pragma omp parallel for
for (i=0; i<n; i++) {x[i] = y[i] + z[i]; }
```

www.benjaminathawes.com/blog
Today’s lecture

• Performance programming for stencil methods
• Vectorization
• Computing with GPUs
Streaming SIMD Extensions

- Each operation is independent

```c
for i = 0:N-1 { p[i] = a[i] * b[i]; }
```

- SSE (SSE4 on Intel Nehalem), Altivec
- Short vectors: 128 bits (AVX: 256 bits)

```
\begin{align*}
1 & \\
4 & \\
2 & \\
6 & = \\
\end{align*}
\begin{align*}
1 & \\
2 & \\
2 & \\
3 & \\
\end{align*}
\begin{align*}
1 & \\
2 & \\
1 & \\
2 & \\
\end{align*}
```

4 floats
2 doubles
16 bytes

Jim Demmel
Fused Multiply/Add

\[ r[0:3] = c[0:3] + a[0:3] \times b[0:3] \]

Courtesy of Mercury Computer Systems, Inc.
How do we use the SSE instructions?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```bash
icc -O3 -ftree-vectorizer-verbose=2 -c s35.c
```

```c
int N;
float a[N], b[N], c[N] = ...;
for (int i=0; i<N; i++)
    a[i] = b[i] + c[i];
```

s35.c(17): (col. 5) remark: LOOP WAS VECTORIZED.
s35.c(20): (col. 20) remark: LOOP WAS VECTORIZED.
s35.c(7): (col. 5) remark: LOOP WAS VECTORIZED.
How does non-vectorized code compare?

- Low level: assembly language or libraries
- Higher level: a vectorizing compiler

```c
#pragma novector
for (int i=0; i<N; i++) // N = 2048 * 1024
    a[i] = b[i] + c[i];
```

Single precision, running on a Nehalem processor using pgcc

- With vectorization: 7.693 sec.
- Without vectorization: 10.17 sec.

- Double precision

- With vectorization: 11.88 sec.
- Without vectorization: 11.85 sec.
How does the vectorizer work?

- Transformed code
  
  ```
  for (i = 0; i < 1024; i+=4)
      a[i:i+3] = b[i:i+3] + c[i:i+3];
  ```

- Vector instructions
  
  ```
  for (i = 0; i < 1024; i+=4){
      vB = vec_ld( &b[i] );
      vC = vec_ld( &c[i] );
      vA = vec_add( vB, vC );
      vec_st( vA, &a[i] );
  }
  ```
What prevents vectorization?

• Data dependencies
  for (int i = 1; i < N; i++)
    b[i] = b[i-1] + 2;
flow.c(10): warning #592: variable "b" is used before its value is set
  b[i] = b[i-1] + 2; /* data dependence cycle */

• Inner loops only
  for(int j=0; j< reps; j++)
    for (int i=0; i<N; i++)
      a[i] = b[i] + c[i];

\[
\begin{align*}
  b[1] &= b[0] + 2; \\
  b[2] &= b[1] + 2; \\
\end{align*}
\]
What prevents vectorization

• Interrupted flow out of the loop
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
    if (maxval > 1000.0) break;
  }

  t2.c(6): (col. 3) remark: loop was not vectorized:
  nonstandard loop is not a vectorization candidate.

• This loop will vectorize
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
  }
Today’s lecture

- Performance programming for stencil methods
- Vectorization
- Computing with GPUs
Heterogeneous processing
NVIDIA GeForce GTX 280

- Hierarchically organized clusters of streaming multiprocessors
  - 240 cores @ 1.296 GHz
  - Peak performance 933.12 Gflops/s
- SIMT parallelism
- 1 GB “device” memory (frame buffer)
- 512 bit memory interface @ 132 GB/s

GTX 280: 1.4B transistors
Intel Penryn: 410M (110mm²) (dual core)
Nehalem: 731M (263mm²)
Streaming processor cluster

- GTX-280 GPU
  10 clusters @ 3 streaming multiprocessors or vector cores
- Each vector core
  - 8 scalar cores: fused multiply adder + multiplier (32 bits), truncate intermediate rslt
  - Shared memory (16KB) and registers (16K × 32 bits = 64KB)
  - 1 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
  - 1 FMA + 1 multiply per cycle = 3 flops / cycle / core * 240 cores = 720 flops/cycl
    @1.296 Ghz: 933 GFLOPS
Streaming Multiprocessor

Streaming Multiprocessor (SM)

Instruction L1 Cache (read only)

Multithreaded Instruction Fetch and Issue Unit (MT Issue)
(Out-of-Order Thread Dispatch, up to 1024 Threads/32 Warp active)

16KB Shared Memory
(Texels, read/write)
(Not used explicitly for pixel shader programs)

Double Precision Unit
Super Function Unit (SFU)

Register File (RF) 0

Register File 1

Register File 2

Register File 3

Register File 4

Register File 5

Register File 6

Register File 7

Streaming Processor 0

Streaming Processor 1

Streaming Processor 2

Streaming Processor 3

Streaming Processor 4

Streaming Processor 5

Streaming Processor 6

Streaming Processor 7

Load Texture

Constant L1 Cache 8KB? (read only)

64KB (SM total)
16x84 x 32bits registers /SM

Load/Store Unit

Register File

L1 Fill

Store to Memory

Load from Memory

H. Goto

Memory Hierarchy

<table>
<thead>
<tr>
<th>Name</th>
<th>Latency (cycles)</th>
<th>Cached</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Local</td>
<td>DRAM – 100s</td>
<td>No</td>
</tr>
<tr>
<td>Constant</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Texture</td>
<td>1s – 10s – 100s</td>
<td>Yes</td>
</tr>
<tr>
<td>Shared</td>
<td>1</td>
<td>--</td>
</tr>
<tr>
<td>Register</td>
<td>1</td>
<td>--</td>
</tr>
</tbody>
</table>

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC
CUDA

- Programming environment with extensions to C
- Under control of the *host*, invoke sequences of multithreaded kernels on the *device* (GPU)
- Many lightweight threads
- CUDA: programming environment + C extensions

```
KernelA<<4,8>>
KerneB<<4,8>>
KernelC<<4,8>>
```
Thread execution model

- Kernel call spawns virtualized, hierarchically organized threads
- Hardware handles dispatching, 0 overhead
- Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation
Hierarchical Thread Organization

- Thread organization
  - Grid ⊃ Block ⊃ Thread
  - Specify number and geometry of threads in a block and similarly for blocks
- Thread Blocks
  - Unit of workload assignment
  - Subdivide a global index domain
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory

KernelA<<<2,3>,<3,5>>>()  
Grid  Block

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC
Threads, blocks, grids and data

- Threads all execute same instruction (SIMT)
- A block may have a different number of dimensions (1d, 2d or 3d) than a grid (1d/2d)
- Each thread uniquely specified by block & thread ID
- Programmer determines the mapping of virtual thread IDs to global memory location
  \[ \Pi: \mathbb{Z}^n \rightarrow \mathbb{Z}^2 \times \mathbb{Z}^3 \]
  \[ \Theta(\Pi_i), \forall \Pi_i \in \Pi \]
Constraints

- **SM**
  - Up to 8 resident blocks
  - Not more than 1024 threads
  - Up to 32 warps
- **All threads in a warp execute the same instruction**
  - All branches followed
  - Instructions disabled
  - Divergence, serialization
- **Grid:** 1 or 2-dimensional (64k-1)
- **Block:** 1, 2, or 3-dimensional
  - ≤ 512 threads
  - Max dimensions: 512, 512, 64
  - Registers subdivided over threads
  - Synchronization only among all threads in the block
- **Synchronization only within a block**
Parallel Speedup

• How much did our GPU implementation improve over the traditional processor?
• Speedup, $S$

Running time of the fastest program on conventional processors

Running time of the accelerated program

• Baseline: a multithreaded program
What limits speedup?

- Avoid algorithms that present intrinsic barriers to utilizing the hardware
- Hide latency of host ↔ device memory transfers
- Reduce global memory accesses
  - Global memory accesses → fast on-chip accesses
  - Coalesced memory transfers
- Avoid costly branches, or render harmless
- Minimize serial sections
Performance issues

• Simplified processor design, but more user control over the hardware resources

• **Rethink the problem solving technique**
  - Hide latency of host $\leftrightarrow$ device memory transfers
  - Global memory accesses $\rightarrow$ fast on-chip accesses
  - Coalesced memory transfers
  - Avoid costly branches, or render them harmless

• If we don’t use the parallelism, we lose it
  - Amdahl’s law - serial sections
  - Von Neumann bottleneck – data transfer costs
  - Workload Imbalances
Fermi

- All the cseclass machines have Fermi processors
  - Private L1 Cache/Shared Memory (64KB/Vector unit), Shared L2 Cache (768 KB)
- CSEClass 01, 02: GeForce GTX 580 [Capability 2.0, GF100]
  - 15 Vector units @ 32 cores/unit (480 cores), 4 SFUs
  - 1.25 GB device memory
- CSEClass 03-07: GeForce GTX 460 Capability 2.1, GF104]
  - 7 Vector units @ 48 cores (384 total cores), 8 SFUs
  - 1.0 GB device memory

www.anandtech.com/show/3809/nvidias-geforce-gtx-460-the-200-king/2
Fin