Lecture 6

Accelerators
Introduction to CUDA
Announcements

• Programming Lab #2 has been posted: GPU programming
• Use Lilliput and Lincoln
  ♦ Come to office hours and get a Lincoln account
  ♦ Device contention on Lilliput
• Triton workshop: November 3\textsuperscript{rd} (afternoon?)
• Midterm: Tuesday November 9\textsuperscript{th} – open book, open notes
Nehalem’s memory hierarchy

- Source: *Intel 64 and IA-32 Architectures Optimization Reference Manual*, Table 2.7
- All data caches are allocate-on-write
Performance programming: Aliev-Panfilov

- ./apf –n 1000 –t 2.5 –i 1000
  - 11.1s (OpenMP; NT=1) vs. 8.24s (w/o OpenMP)
  - Scaling (OpenMP): 11.1s (nt=1); 6.4s (nt=2); 3.9s (nt=4)
  - Triton.sdsc.edu+PGI: 7.78s, 3.93s, 2.06s, 1.30s [nt=8]; 7.7s [serial]

- Miss rates according to valgrind
  - L1d (284M (4.9%) = 214M rd (4.8%) + 70M wr (5.6%)
  - L2d (280M (4.9%) = 210M rd (4.7%) + 70M wr (5.5%)

```c
#pragma omp parallel
  for (j=1; j<=m+1; j++){
    DOUBLE *EJ = &E[j][0], *EP = &E_prev[j][0];
    for (i=1; i<=n+1; i++)
  }

for (j=1; j<=m+1; j++){
  for (i=1; i<=n+1; i++)
    E[j][i] = E_prev[j][i]+alpha*(E_prev[j][i+1]+E_prev[j][i-1]-4*E_prev[j][i]+E_prev[j+1][i]+E_prev[j-1][i]);
```

10/12/10  ©2010 Scott B. Baden /CSE 260/ Fall '10
Loop fusion

```c
for (j=1; j<=m+1; j++){
    DOUBLE *RR = &R[j][1], *EE = &E[j][1];
    for (i=1; i<=n+1; i++, EE++, RR++)
        EE[0] += -dt*(kk*EE[0]*(EE[0]-a)*(EE[0]-1)+EE[0]*RR[0]);
}

for (j=1; j<=m+1; j++){
    DOUBLE *RR = &R[j][1], *EE = &E[j][1];
    for (i=1; i<=n+1; i++, EE++, RR++)
        RR[0] += dt*(epsilon+M1*RR[0]/(EE[0]+M2))*(-RR[0]-kk*EE[0]*(EE[0]-b-1));
}
```
Fusing the PDE with the ODE loop

```c
for (j=1; j<=m+1; j++){
    DOUBLE *EJ = &E[j][0], *EP = &E_prev[j][0], *RR = &R[j][0];
    for (i=1; i<=n+1; i++)
        EJ[i] += -dt*(kk*EJ[i]*(EJ[i]-a)*(EJ[i]-1)+EJ[i]*RR[0]);
        RR[i] += dt*(epsilon+M1*RR[i]/(EJ[i]+M2))*(-RR[i]-kk*EJ[i]*(EJ[i]-b-1));
}
```
Vector processing
Streaming SIMD Extensions

- Each $p[i]$ is independent

  for $i = 0:N-1$ { $p[i] = a[i] \times b[i];$ }

- SSE (SSE4 on Intel Nehalem), Altivec
- Short vectors: 128 bits (256 bits in ‘11)

Jim Demmel

10/12/10

©2010 Scott B. Baden /CSE 260/ Fall '10
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

```
gcc -O3 -ftree-vectorizer-verbose=2 s35.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:8: note: LOOP VECTORIZED.
s35.c:4: note: vectorized 1 loops in function
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 (Triton) 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
  ```c
  for (i = 0; i < 1024; i+=4)
      a[i:i+3] = b[i:i+3] + c[i:i+3];
  ```

- Vector instructions
  ```c
  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;
Loop not vectorized: data dependency

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

  Loop not vectorized/parallelized: multiple exits

• This loop will vectorize
  
  for (i=0; i<n; i++) {
    a[i] = b[i] + c[i];
    maxval = (a[i] > maxval ? a[i] : maxval);
  }
Dealing with spurious dependencies

• Tell the compiler to ignore the dependence

212: for (i=1; i<nx1; i++)
213:   for (j=1; j<ny1; j++)
214:     #pragma ivdep
215:   for (k=1; k<nz1; k++)

\[
\text{Un}[i][j][k] = c \times (\text{Un}[i-1][j][k] + \text{Un}[i+1][j][k] + \text{Un}[i][j-1][k] + \\
\text{Un}[i][j+1][k] + \text{Un}[i][j][k-1] + \text{Un}[i][j][k+1] - \\
c2 \times \text{b}[i-1][j-1][k-1]);
\]

212, Loop not vectorized/parallelized: too deeply nested
215, Unrolled inner loop 4 times!

• Without the pragma (What is causing this to happen?)

214, Loop not vectorized: data dependency
Loop not vectorized: data dependency
Alignment

- Unaligned data movement is expensive
- Accesses aligned on 16 byte boundaries go faster
- Intel compiler can “peel” and perform other optimizations

```c
double a[N], b[N];
for (int i = 1; i < N-1; i++)
    a[i+1] = b[i] * 3;

cchar *x = ..
for (int i = 0; i < 1024; i++)
    x[i] = 1;
```

```c
for (int i = 2; i < N-1; i++)
    a[i+1] = b[i] * 3
```

```c
peel = x & 0x0f;
if (peel != 0) {
    peel = 16 - peel;
    for (i = 0; i < peel; i++) x[i] = 1;
}
/* aligned access */
for (i = peel; i < 1024; i++) x[i] = 1;
```
Computing with Graphical Processing Units (GPUs)
Heterogeneous processing

- Two types of processors: general purpose + accelerator
- Accelerator can perform certain tasks more quickly subject to various overhead costs
- Accelerator amplifies relative cost of communication
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 (dual core)
Streaming processing 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
  - Share local memory and registers
  - 64-bit fused multiply-adder + 2 super function units (2 fused multiply-adders)
- 3 flops/core/cycle * 240 cores = 720 flops/cycle
- @ 1.296 Ghz: 933 GFLOPS
Streaming Multiprocessor

- Each vector processor has 8 cores (Streaming Processors)
  - Each core contains 1 fused multiply adder (single precision), truncates intermediate result
  - May complete 1FMA + 1 multiply per cycle = 3 flops / cycle
  - Share one 64-bit fused multiply adder
  - 2 Super Function Units (SFUs), each contains 2 fused multiply-adders
- 16 KB shared memory + 16K registers

DavidKirk/NVIDIA and Wen-mei Hwu/UIUC
Streaming Multiprocessor

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
(32 banks, read/write)
(Not used explicitly for pixel shader programs)

Double Precision Unit
Super Function Unit (SFU)

Streaming Processor (SP) 0
Register File (RF) 0

Streaming Processor 1
Register File 1

Streaming Processor 2
Register File 2

Streaming Processor 3
Register File 3

Register File 4

Register File 5

Register File 6

Register File 7

Constant L1 Cache 8KB? (read only)

Load Texture

Load/Store Unit

Store to Memory

Load from Memory

Register File 64KB (SM total)
16384 x 32bits registers / SM

L1 Fill

Instruction & Constant L1 Cache (read only)

H. Goto

10/12/10
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
- Model: Execution on the CPU, run a sequence of multi-threaded kernels on the “device” (GPU)
- Threads are extremely lightweight and virtualized
Hierarchical Thread Organization

- Grid ⊇ Block ⊇ Thread
- Thread Blocks
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory
  - Blocks within a grid are virtualized, too
  - May configure number of threads in a block and the number of blocks
- Threads assigned to SM in units of blocks, up to 8 for each SM
- Compiler re-arranges loads to hide latencies

KernelA<<<2,3>,<3,5>>>

DavidKirk/NVIDIA & Wen-mei Hwu/UIUC
Constraints

- An SM may be assigned up to 8 thread blocks comprising up to 32 warps
- Not more than 1024 threads
- Registers subdivided over all threads in a block
- All threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization
- Synchronization within block only
- Blocks are 3 dimensional
- Grids are 2 dimensional (1:64K-1)
Coding example – Increment Array

• Rob Farber, Dr Dobb’s Journal

Serial Code

```c
void incrementArrayOnHost(float *a, int N) {
    int i;
    for (i=0; i < N; i++) a[i] = a[i]+1.f;
}
```

Rob Farber, Dr Dobb’s Journal
#include <cuda.h>

__global__ void incrementOnDevice(float *a, int N) {
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx<N) a[idx] = a[idx]+1.f;
}

incrementOnDevice <<< nBlocks, blockSize >>> (a_d, N);