Lecture 6

Performance
Computing with Graphical Processing Units
Announcements

• A2: GPU programming, released this evening

• Mac Mini lab (APM 2402)
  • Every Tues and Fri for 2 weeks
  • Starts Tues 10/23

• Run the incrArray code example
  • On a Mac Mini using instructions in What’s new section of main course web page
  • On Dirac

• Project proposals due on November 7

• See the project list page
Project Proposals

- **Due 11/7**
  - What are the goals of your project? Are they realistic?
  - What are your hypotheses?
  - What is your experimental method for proving or disproving your hypotheses?
  - What experimental result(s) do you need to demonstrate?
  - What would be the significance of those results?
  - What code will you need to implement? What software packages or previously written software will use?
  - A tentative division of labor among the team members
  - A preliminary list of milestones—with completion dates
Nehalem’s memory hierarchy

- Source: *Intel 64 and IA-32 Architectures Optimization Reference Manual*

### Table 2-20. Cache Parameters of Intel Core i7 Processors

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First Level Data</td>
<td>32 KB</td>
<td>8</td>
<td>64</td>
<td>4</td>
<td>1</td>
<td>Writeback</td>
</tr>
<tr>
<td>Instruction</td>
<td>32 KB</td>
<td>4</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Second Level</td>
<td>256 KB</td>
<td>8</td>
<td>64</td>
<td>10¹</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
<tr>
<td>Third Level (Shared L3)²</td>
<td>8 MB</td>
<td>16</td>
<td>64</td>
<td>35-40+²</td>
<td>Varies</td>
<td>Writeback</td>
</tr>
</tbody>
</table>

**NOTES:**
1. Software-visible latency will vary depending on access patterns and other factors.
2. Minimal L3 latency is 35 cycles if the frequency ratio between core and uncore is unity.
Intel Core memory hierarchy

- **Source:** *Intel 64 and IA-32 Architectures Optimization Reference Manual*

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>First Level</td>
<td>32 KB</td>
<td>8</td>
<td>64</td>
<td>3</td>
<td>1</td>
<td>Writeback</td>
</tr>
<tr>
<td>Instruction</td>
<td>32 KB</td>
<td>8</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
<td>N/A</td>
</tr>
<tr>
<td>Second Level (Shared L2)²</td>
<td>2, 4 MB</td>
<td>8 or 16</td>
<td>64</td>
<td>14²</td>
<td>2</td>
<td>Writeback</td>
</tr>
<tr>
<td>Second Level (Shared L2)³</td>
<td>3, 6 MB</td>
<td>12 or 24</td>
<td>64</td>
<td>15²</td>
<td>2</td>
<td>Writeback</td>
</tr>
<tr>
<td>Third Level</td>
<td>8, 12, 16 MB</td>
<td>16</td>
<td>64</td>
<td>~110</td>
<td>12</td>
<td>Writeback</td>
</tr>
</tbody>
</table>

©2012 Scott B. Baden /CSE 260/ Fall 2012
Comparing memory hierarchies

• Bang
  “E5345 Clovertown”
  Intel Core Architecture
Today’s lecture

• CUDA Programming
• Matrix Multiplication on the GPU
Fermi platforms in the class

CSEClass 01, 02: GeForce GTX 580 [2.0, GF100]
  15 Vector units @ 32 cores/unit (480 cores), 4 SFUs
  1.25 GB device memory
CSEClass 03-07: GeForce GTX 460 [2.1, GF104]
  7 Vector units @ 48 cores (384 total cores), 8 SFUs
  1.0 GB device memory
Dirac: Tesla C2050 [2.0, GF100]
  1 device per node
  14 Vector units @ 32 cores (448 total cores), 4 SFUs
  3 GB device memory + ECC (2.625GB usable)
  SP MAD: 1030.4 Gflops, DP FMA: 515.2

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

©2012 Scott B. Baden /CSE 260/ Fall 2012
CUDA

- Programming environment + C extensions
- Under control of the *host*, run a sequence of multi-threaded GPU kernels on the *device*
- Extremely lightweight virtualized threads
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
```

David Kirk/NVIDIA & Wen-mei Hwu/UIUC
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 language extensions

• Type qualifiers to declare device kernel functions
  __global__ void matrixMul( …)
• Kernel launch syntax
  matrixMul<<< grid, threads >>>(…)  
• Keywords: configuration parameters 
  blockIdx, threadIdx, blockDim, gridDim
• Runtime, e.g. storage allocation 
  cudaMalloc, cudaFree, cudaMemcpy
Coding example – Increment Array

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

#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);
Managing memory

```c
float *a_h, *b_h;       // pointers to host memory
float *a_d;             // pointer to device memory

cudaMalloc((void **) &a_d, size);

for (i=0; i<N; i++) a_h[i] = (float)i;  // init host data

cudaMemcpy(a_d, a_h, sizeof(float)*N, cudaMemcpyHostToDevice);
```
Computing and returning result

int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
incrementOnDevice <<< nBlocks, bSize >>> (a_d, N);

// Retrieve result from device and store in b_h
cudaMemcpy(b_h, a_d, sizeof(float)*N, cudaMemcpyDeviceToHost);

// check results
for (i=0; i<N; i++) assert(a_h[i] == b_h[i]);

// cleanup
free(a_h); free(b_h);
cudaFree(a_d);
Experiments - increment benchmark

- Total time: timing taken from the host, includes copying data to the device
- Device only: time taken on device only

\[
\begin{array}{cccc}
\text{Reps} = 10 & 100 & 1000 & 10000 \\
8.5 & 83 & 830 & 8300 & \text{Device time} \\
29 & 100 & 850 & 8300 & \text{Kernel launch + data xfer} \\
77 & 770 & 7700 & & \text{Host} \\
16 & 103 & & & \text{a[i] = 1 + sin(a[i]) : Device) } \\
7000 & 23.6 \text{ sec} & & & \text{Sine function (Host)} \\
\end{array}
\]
Measuring performance

• Two ways
  ♥ Use an ordinary timer, e.g. gettimeofday()
  ♥ Use Cuda events/elapsed time (#ifdef CUDA_TIMER)
• See incrArray
• Note that kernel invocation is asynchronous

```c
cudaThreadSynchronize();
double t_device_compute = -getTime();
    incr<< nBlocks, bSize >>> (a_d, N);
cudaThreadSynchronize();
t_device_compute +=getTime();
```
CUDA Error Handling

_Cuda error: Can't run kernel: invalid device function._

- Cuda can silently fail, you can observe misleading performance
- E.g. if you specify an invalid grid / thread block dimensions
- Note: the last error can be cleared by successive kernel calls, so check frequently

```c
.cudaMalloc((void **) &a_d, size);
.checkCUDAError("Unable to allocate storage on the device");
```

- Consult _checkCUDAError() in utils.cu (incrArr)_
- What about asynchronous calls?
Getting information about the binary

- Compiler will report a kernel’s register usage along with that of local, shared and constant memory
  --ptxas-options=-v

incrementArrays (float *a, int N)

```c
int idx = blockIdx.x*blockDim.x + threadIdx.x;
if (idx<N) a[idx] = a[idx]+1.f;
```

ptxas info: Compiling entry function
'__Z22incrementArrayOnDevicePfii' for 'sm_13'
ptxas info: Used 4 registers, 16+16 bytes smem, 4 bytes cmem[1]
Today’s lecture

• CUDA Programming
• Matrix Multiplication on the GPU
Naïve Host Code

// “ijk” kernel
for i := 0 to n-1
  for j := 0 to n-1
    for k := 0 to n-1
      C[i,j] += A[i,k] * B[k,j]

for (unsigned int i = 0; i < N; i++)
  for (unsigned int j = 0; j < N; j++) {
    DOUBLE sum = 0;
    for (unsigned int k = 0; k < N; k++)
      sum += A[i * N + k] * B[k * N + j];
    C[i * N + j] = (DOUBLE) sum;
  }
Naïve kernel implementation

• Each thread computes one element of C
  - Loads a row of matrix A
  - Loads a column of matrix B
  - Computes a dot product

• Every value of A and B is loaded N times from global memory
Naïve Kernel

```c
__global__ void matMul(DOUBLE* C, DOUBLE* A, DOUBLE* B) {
    int I = blockIdx.x * blockDim.x + threadIdx.x;
    int J = blockIdx.y * blockDim.y + threadIdx.y;
    int N = blockDim.y * gridDim.y; // Assume a square matrix
    if ((I < N) && (J < N)){
        float _c = 0;
        for (unsigned int k = 0; k < N; k++) {
            float a = A[I * N + k];
            float b = B[k * N + J];
            _c += a * b;
        }
        C[I * N + J] = _c;
    }
}
```

```c
for (unsigned int i = 0; i < N; i++)
    for (unsigned int j = 0; j < N; j++) {
        DOUBLE sum = 0;
        for (unsigned int k = 0; k < N; k++)
            sum += A[i * N + k] * B[k * N + j];
        C[i * N + j] = (DOUBLE) sum;
    }
```
CUDA code on the host side

```c
unsigned int n2 = N*N*sizeof(DOUBLE);
DOUBLE *h_A = (DOUBLE*) malloc(n2);
DOUBLE *h_B = (DOUBLE*) malloc(n2);
// Check that allocations went OK
assert(h_A); assert(h_B);

genMatrix(h_A, N, N); genMatrix(h_B, N, N); // Initialize matrices

DOUBLE *d_A, *d_B, *d_C;
cudaMalloc((void**) &d_A, n2); ... &d_A ... &d_B
checkCUDAError("Error allocating device memory arrays");

// copy host memory to device
cudaMemcpy(d_A, h_A, n2, cudaMemcpyHostToDevice);
checkCUDAError("Error copying data to device");
cudaMemcpy(d_B, h_B, n2, cudaMemcpyHostToDevice);
checkCUDAError("Error copying data to device");
```
Host code - continued

// setup execution configurations
dim3 threads(ntx, nty,1); // ntx & nty are user input
dim3 grid(N / threads.x, N / threads.y);

// launch the kernel
matMul<<< grid, threads >>>(d_C, d_A, d_B);

// retrieve result
cudaMemcpy(h_C, d_C, n2, cudaMemcpyDeviceToHost);
checkCUDAError("Unable to retrieve result from device");

// Free device storage
assert(cudaSuccess == cudaMemcpy(d_A));
assert(cudaSuccess == cudaMemcpy(d_B));
assert(cudaSuccess == cudaMemcpy(d_C));
Configuration variables

- Types to manage thread geometries
- \texttt{dim3 gridDim, blockDim}
  - Dimensions of the grid in blocks
    \texttt{(gridDim.z \text{ not used})}
  - Dimensions of a thread block in threads
- \texttt{dim3 blockIdx, threadIdx;}
  - Block index within the grid
  - Thread index within the block

```c
__global__ void KernelFunc(...);
dim3 DimGrid(40, 30);   // 1200 thread blocks
dim3 DimBlock(4, 8, 16); // 512 threads per block
Kernel<<< DimGrid, DimBlock, >>>(...);
```
Execution Configurations

- Grid $\supset$ Block $\supset$ Thread

```
__global__ void Kernel (...);

dim3 DimGrid(2,3);       // 6 thread blocks

dim3 DimBlock(3,5,1);    // 15 threads /block

Kernel<<< DimGrid, DimBlock, >>>(...);
```
### Performance

- **Baseline** \([N=512, \text{double precision}]
  - Lilliput, C1060, 2.0 GHz Intel Xeon E5504, 4MB L3, peak 8.0 GF / core
  - Forge, M2070 14×32 cores
  - 21 GF on 4 CPU cores (MPI), 25 Gflops for \(N=2K\)

<table>
<thead>
<tr>
<th>Gflops dp</th>
<th>9.8</th>
<th>8.5</th>
<th>7.4</th>
<th>5.9</th>
<th>5.3</th>
<th>5.1</th>
<th>3.0</th>
<th>2.7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lilliput</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometry</td>
<td>2×256</td>
<td>2×128</td>
<td>2×64</td>
<td>4×128</td>
<td>4×64</td>
<td>4×32</td>
<td>8×64</td>
<td>8×32</td>
</tr>
<tr>
<td>Gflops sp</td>
<td>8.6</td>
<td>7.7</td>
<td>6.2</td>
<td>4.6</td>
<td>3.9</td>
<td>3.5</td>
<td>2.0</td>
<td>1.8</td>
</tr>
<tr>
<td>Lilliput</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometry</td>
<td>2×256</td>
<td>2×128</td>
<td>2×32</td>
<td>2×64</td>
<td>4×128</td>
<td>4×64</td>
<td>4×32</td>
<td>8×64</td>
</tr>
<tr>
<td>Gflops sp</td>
<td>65</td>
<td>48</td>
<td>56</td>
<td>39</td>
<td>52</td>
<td>29</td>
<td>46</td>
<td>29</td>
</tr>
<tr>
<td>Forge</td>
<td>64</td>
<td>46</td>
<td>50</td>
<td>28</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Geometry</td>
<td>2×128</td>
<td>2×256</td>
<td>2×64</td>
<td>4×128</td>
<td>4×64</td>
<td>4×32</td>
<td>2×32</td>
<td>8×64</td>
</tr>
</tbody>
</table>

©2012 Scott B. Baden /CSE 260/ Fall 2012
Warp Scheduling (Fermi)

- Threads assigned to an SM in units of a thread block, multiple blocks
- Each block is divided into warps of 32 (SIMD) threads, a schedulable unit
  - A warp becomes eligible for execution when all its operands are available
  - Dynamic instruction reordering: eligible warps selected for execution using a prioritized scheduling policy
  - All threads in a Warp execute the same instruction, branches serialize execution
- Multiple warps simultaneously active, hiding data transfer delays
- All registers in all the warps are available, 0 overhead scheduling
- Hardware is free to assign blocks to any SM
- How does scheduling work?
Summary - Programming issues

• Branches serialize execution within a warp
• Tradeoff: more blocks with fewer threads or more threads with fewer blocks
  ♦ Locality: want small blocks of data (and hence more plentiful warps) that fit into fast memory
  ♦ Register consumption
  ♦ Scheduling: hide latency
• Shared memory and registers do not persist across kernel invocations
• Next time: using shared memory
Fin