Lecture 7
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Matrix multiplication
Using Shared Memory
Announcements

• Special (one time) office hours on Friday 2pm to 4pm
Today’s lecture

• Occupancy
• Using shared memory to increase memory locality in matrix multiply
Recapping from last time

- GPUs run 1000s of threads on 13 SMX units, each with 64 double precision cores, 192 single precisions cores, and 64 special function units
- Threads communicate slowly via global
- On chip memories are $O(100x)$ faster: shared memory/cache + registers
### The increment benchmark on Sorken

- Total time: timing taken from the host, includes copying data to the device and copying back the result
- Device only: time taken on device only
- Loop repeats the computation inside the kernel: 1 kernel launch + 1 set of data transfers in and out of device
- CUDA bandwidthTest reports 12.1 GB/s Host-Device for 32MB messages (in $PUB/Examples/CUDA/samples)

N = 8388480 (8M ints), block size = 128, times in **milliseonds**,

<table>
<thead>
<tr>
<th>Repetitions</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>$10^4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.46</td>
<td>12.1</td>
<td>119</td>
<td>1.19s</td>
<td></td>
</tr>
<tr>
<td>9.30</td>
<td>20.1</td>
<td>127</td>
<td>1.19s</td>
<td></td>
</tr>
<tr>
<td>44.0</td>
<td>358</td>
<td>3.53s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.31</td>
<td>24.1</td>
<td>233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3.10s</td>
<td>11.0s</td>
<td>89.5s</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- Device time
- Kernel launch+data xfer
- Host
- Device
- Host
Naïve implementation of matrix multiply

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

for (i = 0; i < N; i++)
  for (j = 0; j < N; j++) {
    DOUBLE sum = 0;
    for (k = 0; k < N; k++)
      sum += A[i * N + k] * B[k * N + j];
    C[i * N + j] = (DOUBLE) sum;
  }
```
Thread mapping

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

```c
unsigned int n2 = N*N*sizeof(DOUBLE);     // Generic types
DOUBLE *h_A = (DOUBLE*) malloc(n2);       // in types.h
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");
```
// 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 == cudaFree(d_A));
    assert(cudaSuccess == cudaFree(d_B));
    assert(cudaSuccess == cudaFree(d_C));
Configuration variables

- Types to manage thread geometries
- `dim3 gridDim, blockDim`
  - Dimensions of the grid in blocks
  - Dimensions of a thread block in threads
- `dim3 blockIdx, threadIdx`;
  - Block index within the grid
  - Thread index within the block

```c
__global__ void KernelFunc(...);

dim3 DimGrid(2,3);       // 6 thread blocks
dim3 DimBlock(3,5,1);    // 15 threads /block
Kernel<<< DimGrid, DimBlock, >>>>(...);
```
Performance

- N=512, double precision
- GPU: 1 Device of Sorken
- CPU Stampede, 2.0 GHz Intel Xeon E5-2680 0 @ 2.70GHz peak 21.6 GF / core

<table>
<thead>
<tr>
<th>Geometry</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 x 16</td>
<td>92</td>
</tr>
<tr>
<td>32 x32</td>
<td>116</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th># cores</th>
<th>GFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>2</td>
<td>43</td>
</tr>
<tr>
<td>4</td>
<td>84</td>
</tr>
<tr>
<td>8</td>
<td>160</td>
</tr>
</tbody>
</table>
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
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?
Warp Scheduling

- Threads assigned to an SMX in units of a thread block, multiple blocks
- Each block 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 SMX
- There are 4 warp schedulers/SMX
Today’s lecture

• Occupancy
• Using shared memory to increase memory locality in matrix multiply
Occupancy

- A minimum number of warps needed to hide memory latency
- **Occupancy:** \( \frac{\text{# active warps}}{\text{max \# warps supported by vector unit}} \)
- Limited by vector unit resources
  - Amount of shared memory
  - Number of registers
  - Maximum number of threads
- Consider a kernel (16x16 block size)
  - Shared memory/block = 2648 bytes
  - Reg/thread=38 [38*256 = 9728 < 16k]
  - # available registers is the limiting factor
- 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
- Maximizing the occupancy may not maximize performance
Occupancy Calculator

Determining occupancy

- Recall the definition for occupancy
  \[ \frac{\text{# active warps}}{\text{max # warps supported by vector unit}} \]

- NVIDIA provides an occupancy calculator

- Determine resource usage from `nvcc` for provided A2 code on Sorken (capability 3.7)
  ```
  nvcc --ptxas-options=-v
  ptxas info : Used 26 registers, 352 bytes cmem[0]
  ```
Occancy calculation with 16 x 16 threads

\[
\text{Occancy} = \frac{\# \text{ active warps per SM}}{\text{Maximum possible } \# \text{ active warps}}
\]

**Physical Limits for GPU Compute Capability:**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per Warp</td>
<td>32</td>
</tr>
<tr>
<td>Warps per Multiprocessor</td>
<td>64</td>
</tr>
<tr>
<td>Threads per Multiprocessor</td>
<td>2048</td>
</tr>
<tr>
<td>Thread Blocks per Multiprocessor</td>
<td>16</td>
</tr>
<tr>
<td>Total # of 32-bit registers per Multiprocessor</td>
<td>65536</td>
</tr>
<tr>
<td>Register allocation unit size</td>
<td>256</td>
</tr>
<tr>
<td>Register allocation granularity</td>
<td>warp</td>
</tr>
<tr>
<td>Registers per Thread</td>
<td>255</td>
</tr>
<tr>
<td>Shared Memory per Multiprocessor (bytes)</td>
<td>49152</td>
</tr>
<tr>
<td>Shared Memory Allocation unit size</td>
<td>256</td>
</tr>
<tr>
<td>Warp allocation granularity</td>
<td>4</td>
</tr>
<tr>
<td>Maximum Thread Block Size</td>
<td>1024</td>
</tr>
</tbody>
</table>

**CUDA GPU Occupancy Calculator**

1.) Select Compute Capability (click): 3.5
1.b) Select Shared Memory Size Config (bytes) 49152

2.) Enter your resource usage:
- Threads Per Block: 256
- Registers Per Thread: 26
- Shared Memory Per Block (bytes): 0

3.) GPU Occupancy Data is displayed here and in the graphs:
- Active Threads per Multiprocessor: 2048
- Active Warps per Multiprocessor: 64
- Active Thread Blocks per Multiprocessor: 8
- Occupancy of each Multiprocessor: 100%

**Allocated Resources**

<table>
<thead>
<tr>
<th>Resource</th>
<th>Per Block Limit</th>
<th>Per SM Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warps (Threads Per Block / Threads Per Warp)</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>Registers (Warp limit per SM due to per-warp reg count)</td>
<td>8</td>
<td>64</td>
</tr>
<tr>
<td>Shared Memory (Bytes)</td>
<td>0</td>
<td>49152</td>
</tr>
</tbody>
</table>

Note: SM is an abbreviation for (Streaming) Multiprocessor

**Maximum Thread Blocks Per Multiprocessor**

<table>
<thead>
<tr>
<th>Limitation</th>
<th>Blocks/SM</th>
<th>* Warps/Block = Warsps/SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Limited by Max Warps or Max Blocks per Multiprocessor</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Limited by Registers per Multiprocessor</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Limited by Shared Memory per Multiprocessor</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>Physical Max Warps/SM = 64</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Occupancy = 64 / 64 = 100%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Note: Occupancy limiter is shown in orange

Scott B. Baden / CSE 260, UCSD / Fall '15
Full occupancy
Today’s lecture

• Occupancy
• Using shared memory to increase memory locality in matrix multiply
Kepler’s Memory Hierarchy

- DRAM takes hundreds of cycles to access
- Each SMX has on-chip memory, can configure part as shared memory, part as L1 cache

\[ \frac{3}{4} + \frac{1}{4}, \quad \frac{1}{4} + \frac{3}{4}, \quad \frac{1}{2} + \frac{1}{2} \]

Set the mode using `cudaFuncSetCacheConfig()`

- All threads in a block share this memory, hence a set of warps making up the block
- L2 Cache shared by all SMXs (1.5 MB)
- Cache inclusion (L1 ⊂ L2) partially configurable on per-access basis with mem. ref. instruction modifiers
- 128 byte cache line size

B. Wilkinson
Recall Blocked Matrix Multiplication

N blocks, n×n global matrix, b=n/N
for i = 0 to N-1
  for j = 0 to N-1
    // load each block C[i,j] into cache, once :
    // b = n/N = block size
    for k = 0 to N-1
      // load each block A[i,k] and B[k,j] N^3 times
      // = 2N^3 × (n/N)^2 = 2Nn^2
      // write each block C[i,j] once :

Total: (2*N+2)*n^2
Improving locality in matrix multiply

• Naïve algorithm
  ♦ Each thread loads all the data it needs, independently loads a row and column of input
  ♦ Each input element loaded multiple times
  ♦ Each thread computes 1 MAD + 2 loads + 1 store

• Blocked algorithm with shared memory
  ♦ Threads cooperate to load a block of A&B into on-chip shared memory
  ♦ Each thread in the block performs the $ijk$ loop within shared memory
  ♦ Each thread: $b$ mpy-adds + 1 load + 1 store