Lecture 6
CSE 260 – Parallel Computation
(Fall 2015)
Scott B. Baden

Computing with Graphical Processing Units
CUDA Programming
Matrix multiplication
Announcements

• A2 has been released: Matrix multiplication with CUDA
• You’ll optimize matrix multiply to at least 360 Gflops/sec
Today’s lecture

• Introduction to GPUs
  ◆ Architecture
  ◆ Programming (CUDA)
Recapping from last time

- GPUs process long vectors on thousands of specialized cores
- Executing 1000s of threads to hide data motion
- Some regularity involving memory accesses and control flow
Tesla Kepler K20m/K80 (GK110/210)

- Hierarchically organized clusters of streaming multiprocessors
  - 13 streaming processors @ 705 (745) MHz [x2 for K80]
  - Peak performance: 1.2 (2.4) Tflops/s
    Double Precision, fused multiply/add
- 5 (11¼) GB device memory (frame buffer)@208 (240) GB/s
- Stampede has device capability 3.5, Sorken 3.7
- 64K, 128K 32 bit registers
Overview of Kepler GK110
SMX Streaming processor

- AKA vector unit
- Stampede’s K20s (GK110 GPU) have 13 SMXs (2496 cores), so do K80s
- Each vector unit
  - 192 SP cores, 64 DP cores, 32 SFUs, 32 Load/Store units
  - Each scalar core: fused multiply adder, truncates intermediate result
  - 64KB (112K) on-chip memory configurable as scratchpad memory + L1 $L1$
  - 64K (128K) x 32-bit registers (256 (512) KB) up to 255/thread
  - 1 FMA /cycle = 2 flops / cyc / DP core * 64 DP/SMX * 13 SMX = 1664 flops/cyc
  - @0.7006 Ghz = 1.165 TFLOPS per processr (2.33 for K80)
Kepler’s Memory Hierarchy

- DRAM takes hundreds of cycles to access
- Can partition the on-chip Shared memory L,1$ cache
  \[ \left\{ \frac{3}{4} + \frac{1}{4} \right\} \]
  \[ \left\{ \frac{3}{4} + \frac{1}{4} \right\} \]
  \[ \left\{ \frac{1}{2} + \frac{1}{2} \right\} \]
- L2 Cache (1.5 MB)
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>>
```

```
KernelB<<4,8>>
```

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

- Kernel call spawns virtualized, hierarchically organized threads \[ \text{Grid} \supset \text{Block} \supset \text{Thread} \]
- Hardware dispatches blocks to cores, 0 overhead
- Compiler re-arranges loads to hide latencies
- Global synchronization: kernel invocation

Global Memory

KernelA\[<<2,3>,<3,5>>()]
Execution Configurations

- Grid ⊇ Block ⊇ Thread
- Expressed with configuration variables
- Programmer sets the thread block size, maps threads to memory locations
- Each thread uniquely specified by block & thread ID

```c
__global__ void Kernel (...);
dim3 DimGrid(40,30);  // 1200 thread blocks
dim3 DimBlock(3,5,10);  // 150 threads /block
Kernel<<<DimGrid, DimBlock, >>>>(...);
```
Thread execution

- Thread Blocks
  - Unit of workload assignment
  - Each thread has its own set of registers
  - All have access to a fast on-chip *shared memory*
  - Synchronization only among all threads in a block
  - Threads in different blocks communicate via slow global memory
  - Processor groups threads into *warps* of 32 threads
- SIMT parallelism: all threads in a warp execute the same instruction
  - All branches followed
  - Instructions disabled
  - Divergence, serialization

```
KernelA<<<2,3>,<3,5>>>()
```
Coding example – Increment Array

Serial Code

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

CUDA

```c
#include <cuda.h>
__global__ void incrementOnDevice(float *a, int N)
{
    // Each thread uniquely specified by block & thread ID
    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

- Data must be allocated on the device and host and moved between the two explicitly

```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

```c
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

<table>
<thead>
<tr>
<th>Reps</th>
<th>10</th>
<th>100</th>
<th>1000</th>
<th>$10^4$</th>
<th>$10^5$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time</td>
<td>3.3</td>
<td>36</td>
<td>358</td>
<td>3.58s</td>
<td>35.8s</td>
</tr>
<tr>
<td>Device time</td>
<td>71.</td>
<td>102</td>
<td>429</td>
<td>3.64s</td>
<td>35.9s</td>
</tr>
<tr>
<td>Kernel launch + data xfer</td>
<td>92.</td>
<td>730</td>
<td>7.06s</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Host</td>
<td>6.8</td>
<td>52</td>
<td>500</td>
<td>--</td>
<td>--</td>
</tr>
<tr>
<td>Sine function (Device)</td>
<td>6.4s</td>
<td>23.9s</td>
<td>200s</td>
<td>--</td>
<td>--</td>
</tr>
</tbody>
</table>

N = 8388480, block size = 128, times in **milliseconds**, cseclass02

Scott B. Baden / CSE 260, UCSD / Fall '15

18
What is the cost of moving the data and launching the kernel?

A. About 6.8 ms ((71-3)/10)
B. About 0.66 ms (102-36)/100
C. About 0.071 ms ((429-358)/1000)
D. About 68 ms (71-3)

N = 8 M block size = 128, times in milliseconds
Reps = 10   100   1000   10^4   10^5

3.3   36   358   3.58s   35.8s  Device time (increment)
71.   102   429   3.64s   35.9s  Kernel launch + data xfer
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

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

Scott B. Baden / CSE 260, UCSD / Fall '15
Naïve Host Implementation

// “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

```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; // Assumes a 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");
```

Scott B. Baden / CSE 260, UCSD / Fall '15
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 == 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, >>>>(...);
```
Thread mapping

```cpp
__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; // 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;
    }
}
```
Performance

• N=512, double precision
• Stampede, K20, 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>1 x 128</td>
<td>64</td>
</tr>
<tr>
<td>2 x 128</td>
<td>76</td>
</tr>
<tr>
<td>4 x 64</td>
<td>49</td>
</tr>
<tr>
<td>16 x 16</td>
<td>16</td>
</tr>
<tr>
<td>1 x 512</td>
<td>63</td>
</tr>
<tr>
<td>2 x 256</td>
<td>78</td>
</tr>
<tr>
<td>4 x 128</td>
<td>50</td>
</tr>
<tr>
<td>1 x 1024</td>
<td>62</td>
</tr>
<tr>
<td>2 x 512</td>
<td>79</td>
</tr>
<tr>
<td>4 x 256</td>
<td>51</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

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

• Consult checkCUDAError() in utils.cu (incrArr)
• What about asynchronous calls?
• cf CUDA Programming Guide, “Error Handling”
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)

  int idx = blockIdx.x*blockDim.x + threadIdx.x;

  if (idx<N) a[idx] = a[idx]+1.f;

  ptxas info : Compiling entry function '_Z6matMuliPdS_S_' for 'sm_37'
  ptxas info : Function properties for _Z6matMuliPdS_S_
      0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
  ptxas info : Used 26 registers, 352 bytes cmem[0]
Next Time

• How to use shared memory to improve significantly the performance of matrix multiply