Lecture 7

CUDA Programming
A first application – Matrix Multiplication
Announcements

• Thursday’s office hours cancelled
• Lincoln accounts will be available next week
• Timing runs on Lilliput
  ♦ Use time slots starting in the wee hours of Sunday (12am to 4am)
• Setting up Triton accounts
  ♦ Generate a public key
  ♦ Send your key and your partner’s key to: RHAWKINS@SDSC.EDU
Memory interleaving

- Compensates for slow memory access times
- Assume we are accessing memory consecutively
- What happens if the stride = number of banks?

Ruye Wang,
fourier.eng.hmc.edu/e85/lectures/memory
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 (64 KB)
  - 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
Programming with CUDA

- Programming environment + C extensions
- Model: under control of the CPU, run a sequence of multi-threaded GPU kernels
- Extremely lightweight, virtualization
Thread execution model

- Spawn virtualized, hierarchically organized threads
  - Grid ⊇ Block ⊇ Thread
  - Specify number and geometry of threads in a block and similarly for blocks
  - $\Pi: \mathbb{Z}^n \rightarrow \mathbb{Z}^2 \times \mathbb{Z}^3$
  - $\Theta(\Pi \iota), \forall \Pi \iota \in \Pi$

- Thread Blocks
  - Cooperate, synchronize, with access fast on-chip shared memory
  - Threads in different blocks communicate only through slow global memory

- Threads assigned to SM in units of blocks

- Compiler re-arranges loads to hide latencies
CUDA language extensions

• Type qualifiers to declare device kernel functions
  __global__ void matrixMul( …)

• Kernel launch syntax
  matrixMul<< grid, threads >>>(…)

• Keywords
  blockIdx, threadIdx

• Runtime, e.g. storage allocation
  cudaMalloc, cudaFree, cudaMemcpy
Configuration variables

• Types to manage thread geometries
• `dim3 gridDim, blockDim`
  ♦ Dimensions of the grid in blocks
    (gridDim.z not used)
  ♦ 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(40, 30);  // 1200 thread blocks
dim3 DimBlock(4, 8, 16); // 512 threads per block
Kernel<<< DimGrid, DimBlock, >>>(...);
```
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;
}
```

```c
#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);

Rob Farber, Dr Dobb’s Journal
Managing memory

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

cudam_memcpy(a_d, a_h, sizeof(float)*N,
            cudam_memcpyHostToDevice);

int bSize = 4;
int nBlocks = N/bSize + (N%bSize == 0?0:1);
Transferring the Data

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);
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

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

Used 2 registers, 12+16 bytes smem
declared variables + system allocated data [params]
```
Dynamic behavior – resource utilization

- Each vector core (SM): 1024 thread slots and 8 block slots
- Hardware partitions slots into blocks at run time, accommodates different processing capacities
- Registers are split dynamically across all blocks assigned to the vector core
- A register is private to a single thread within a single block

Courtesy David Kirk/NVIDIA and Wen-mei Hwu/UIUC
Threads, blocks, grids and data

- All threads 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 and thread ID
- Programmer determines the mapping of thread IDs to global memory locations
2D thread blocks

```c
void addMatrix
    (float *a, float *b, float *c, int N)
{
    int i, j, idx;
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            idx = i + j*N;
            c[idx] = a[idx] + b[idx];
        }
    }
}

void main()
{
    ...
    addMatrix(a, b, c, N);
}
```

```c
__global__ void addMatrixG
    (float *a, float *b, float *c, int N)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    int j = blockIdx.y*blockDim.y + threadIdx.y;
    int idx = i + j*N;
    if (i < N && j < N)
        c[idx] = a[idx] + b[idx];
}

void main()
{
    dim3 dimBlock (blocksize, blocksize);
    dim3 dimGrid (N/dimBlock.x, N/dimBlock.y);
    addMatrixG<<<dimGrid, dimBlock>>>(a, b, c, N);
}
```

Figure 8. Serial C (a) and CUDA C (b) examples of programs that add arrays.
Warp scheduling

- Blocks are divided into warps of 32 (SIMD) threads which are ….
- Subdivided into schedulable units: 16 threads (→ 32 on Fermi)
- Warps are scheduled with zero overhead in hardware
- Scheduler finds an eligible warp: all operands are ready
  - scoreboard
  - Priorities
- Branches serialize execution within a warp
Constraints

• **SM**
  - Up to 8 blocks
  - Not more than 1024 threads
  - Up to 32 warps

• **Block** – 1, 2, or 3-dimensional
  - Registers subdivided over threads
  - Synchronization among all threads in the block

• **Warp**
  - All threads warp execute the same instruction: all branches followed
  - Divergence, serialization

• **Grid**: 1 or 2-dimensional (1:64k-1)
Consequences

• Registers
  ✷ Dynamically partitioned across each block in a Streaming Multiprocessor
  ✷ Bound to and accessible from their thread only, until the block finishes execution

• 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
Occupancy

• A minimum number of warps needed to hide memory latency
• Varying the block size
  – 8x8: 64 threads/block, max 8 blocks, only 512 threads/SM
  – 16x16: 3 blocks; can we hide the latency?
• Consider an application using 32 registers (4 bytes floats) per thread
• Each SM gets 512 registers, 64 per core, how many threads/SM?
Parallel Speedup

• How much of an improvement did our parallel implementation obtain over the conventional one?
• Speedup, $S$

Running time of the fastest program on conventional processors
Running time of the accelerated program

• Baseline: a multithreaded program
What limits speedup?

• **Serial sections**
  - Algorithms that present intrinsic barriers to scalability → alternatives
  - Cost of porting to the GPU

• **Data transfer costs between device and host**

• If we don’t use the parallelism on the device, we lose it
  - Inability to hide latency with device memory transfers
  - Frequent global memory accesses
  - Uncoalesced memory transfers
  - Utilization of floating point units
  - Branches