Recitation 2:

GPU Programming with CUDA

15-418 Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2020
Goals for today

- Learn to use CUDA

1. Walk through example CUDA program
2. Optimize CUDA performance
3. Debugging & profiling tools

- Most of all,

ANSWER YOUR QUESTIONS!
Matrix multiplication

$$
C_{i,j} = \sum_k A_{i,k} B_{k,j}
$$
Matrix multiplication (matmul)

- Simple C++ implementation:

```cpp
/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

// Standard multiplication
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}
```
Benchmarking simple C++ matmul

- ./matrix -n 1024 -N 1024 -m simple

- Simple C++: 1300 ms, 1.6 Gflops
  - (45% faster than old GHC machines!)
Translating matmul to CUDA

- SPMD (single program, multiple data) parallelism
  - “Map this function to all of this data”: map($f, data$)
  - Similar to SIMD, but doesn’t require lockstep execution

- What this means: You write the “inner loop”, compiler + GPU execute it in parallel
Translating matmul to CUDA

- Simple CUDA implementation:

/* Find element based on row-major ordering */
#define RM(r, c, width) ((r) * (width) + (c))

// Standard multiplication
void multMatrixSimple(int N, float *matA, float *matB, float *matC) {
    for (int i = 0; i < N; i++)
        for (int j = 0; j < N; j++) {
            float sum = 0.0;
            for (int k = 0; k < N; k++)
                sum += matA[RM(i,k,N)] * matB[RM(k,j,N)];
            matC[RM(i,j,N)] = sum;
        }
}

1. Find the inner loop
Translating matmul to CUDA

- Simple CUDA implementation:

```c
__global__ void cudaSimpleOldKernel(int N, float* dmatA, float* dmatB, float * dmatC) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;
    if (i >= N || j >= N)
        return;

    float sum = 0.0;
    for (int k = 0; k < N; k++) {
        sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
    }

    dmatC[RM(i,j,N)] = sum;
}
```

2. Write it as a separate function
Translating matmul to CUDA

- Simple CUDA implementation:

```c
__global__ void cudaSimpleOldKernel(int N, float* dmatA, float* dmatB, float * dmatC) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i >= N || j >= N)
    return;
  float sum = 0.0;
  for (int k = 0; k < N; k++) {
    sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
  }
  dmatC[RM(i,j,N)] = sum;
}
```

3. Compute loop index + test bound (no outer loop)
Benchmarking simple CUDA matmul

- ./matrix -n 1024 -N 1024 -m cosimple

- Simple C++: 1300 ms, 1.6 GFlops
- Simple CUDA: 33 ms, 65 Gflops
  - (30% faster than old GHC GPU)

- ...actually, not very good yet! (stay tuned)
CUDA Terminology

CPU
Host

PCIe

GPU
Device
CUDA Programming Model

- Programmer writes *kernels* executed by each thread
- Blocks have fast shared memory between threads
- Blocks within a grid may execute in any order
CUDA Programming Model

Host (CPU) \[\xrightarrow{\text{PCIe}}\] Device (GPU)

Not all threads used
Invoking CUDA matmul

- Setup memory (from CPU to GPU)
- Invoke CUDA with special syntax
- Get results (from GPU to CPU)
Invoking CUDA matmul

- Setup memory (from CPU to GPU)
  - cudaMalloc((void **) &aDevData, N*N * sizeof(float));
  - cudaMalloc((void **) &bDevData, N*N * sizeof(float));
  - cudaMalloc((void **) &cDevData, N*N * sizeof(float));
  - cudaMemcpy(aDevData, aData, N*N * sizeof(float), cudaMemcpyHostToDevice);
  - cudaMemcpy(bDevData, bData, N*N * sizeof(float), cudaMemcpyHostToDevice);

- Invoke CUDA with special syntax
- Get results (from GPU to CPU)
invoking cuda matmul

- setup memory (from cpu to gpu)
- invoke cuda with special syntax

#define N 1024
#define LBLK 32
dim3 threadsPerBlock(LBLK, LBLK);
dim3 blocks(updiv(N, LBLK), updiv(N, LBLK)); // updiv() divides + rounds up
cudaSimpleKernelOld<<<blocks, threadsPerBlock>>>(N, aDevData, bDevData, cDevData);

- get results (from gpu to cpu)

these addresses are only valid on gpu
Invoking CUDA matmul

- Setup memory (from CPU to GPU)
- Invoke CUDA with special syntax
- Get results (from GPU to CPU)

```c
    tHostData = (float *) calloc(N*N, sizeof(float));
    cudaMemcpy(tHostData, tDevData, N*N*sizeof(float), cudaMemcpyDeviceToHost);
    cudaFree(aDevData); cudaFree(bDevData); cudaFree(cDevData);
```

Need to move data manually (separate address spaces)
Compiling + running CUDA

- CUDA code is in separate *.cu file (cudaMatrix.cu)
  - Compiled like:
    
    nvcc cudaMatrix.cu -O3 -c -o cudaMatrix.o
  
  - (See assignment 2 for $PATH, etc)

- Linked with gcc + flags, e.g.:
  - g++ -O3 -L/path/to/cuda -lcudart -o matrix *.o

- Run like a normal program, e.g.:
  - ./matrix
Profiling performance: How well are we doing?

- CUDA 10 introduced “Nsight” profiling tools
  - Require root permissions; we are trying to resolve

- Deprecated:
  - Run “nvprof” to generate analysis data
    - nvprof --analysis-metrics -f -o cosimple.nvprof
      ./matrix -n 1024 -N 1024 -m cosimple
    - (nvprof has many other options)

- Visualize profile with nvvp cosimple.nvprof
  - You will want to run this locally so X-windows doesn’t lag
nvprof/nvvp Profiling Results

1. CUDA Application Analysis
   The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

2. Check Overall GPU Usage
   The device timeline shows an estimate of the amount of the total compute capacity being used.

- **Low Kernel Concurrency** [0 ns / 76.208 ms = 0%]
  The percentage of time when two kernels are being executed in parallel is low.

- **Low Compute Utilization** [76.208 ms / 7.705 s = 1%]
  The multiprocessors of one or more GPUs are mostly idle.

Examine Individual Kernels

You can also examine the performance of individual kernels to expose additional optimization opportunities.

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bound by compute, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance kernel "cudaSimpleKernel3D" is most likely limited by memory bandwidth.

- Perform Memory Bandwidth Analysis

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

- Perform Compute Analysis
- Perform Latency Analysis

For device "GeForce GTX 1080" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.
nvprof/nvvp Profiling Results

matmul is memory bound!
GPU microarchitecture

L2 Cache

“Global” memory, accessible across entire device
CUDA Programming Model

Grid

Block

SM
Streaming multiprocessor (SM) microarchitecture

Within an SM, thread blocks are broken into warps for execution.

“Shared” memory (only shared within SM/thread block)
Improving matmul memory usage

Why is matmul accessing memory so much?

```c
__global__ void
cudaSimpleOldKernel(int N, float* dmatA,
                     float* dmatB, float * dmatC) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j = blockIdx.y * blockDim.y + threadIdx.y;
  if (i >= N || j >= N)
    return;
  float sum = 0.0;
  for (int k = 0; k < N; k++) {
    sum += dmatA[RM(i,k,N)] * dmatB[RM(k,j,N)];
  }
  dmatC[RM(i,j,N)] = sum;
}
```
Improving matmul memory usage: Peeking under the hood

- Need to think about how threads within a warp access memory...
  - (This is bad – warps aren’t part of programming model)

- CUDA maps threads → warps
  row-major: Same y values, consecutive x values
  - Warp 0: (0,0) (1,0) (2,0) … (31,0)
Improving matmul memory usage: Warp memory access pattern

- What memory locations does warp 0 access?

  \[
  \text{int } i = \text{blockIdx}.x \times \text{blockDim}.x + \text{threadIdx}.x; \\
  \text{int } j = \text{blockIdx}.y \times \text{blockDim}.y + \text{threadIdx}.y;
  \]

- **Access:** \(\text{dmatA}[\text{RM}(i,k,N)]\), \(\text{dmatB}[\text{RM}(k,j,N)]\), \(\text{dmatC}[\text{RM}(i,j,N)]\) where \(\text{RM}(i,j,N) = i \times N + j\)

- Threads have same \(y\) + consecutive \(x\) ➔
- Threads accesses the same \(j\) + consecutive \(i\) ➔
- Threads access memory at stride of \(N\) floats ➔
- 1 reads + 1 writes per thread
Improving matmul memory usage: Better spatial locality

- What if we flipped it around?

```c
int i = blockIdx.y * blockDim.y + threadIdx.y;
int j = blockIdx.x * blockDim.x + threadIdx.x;
```

- Threads have same $y$ + consecutive $x$ ➔
- Threads access the same $i$ + consecutive $j$ ➔
- Threads access memory at stride of 1 ➔
- GPU coalesces reads + writes to memory block ➔
- 1 read + 1 write per warp (if large memory blocks)
Benchmarking improved simple CUDA matmul

- ./matrix -n 1024 -N 1024 -m csimple

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops
  - (>2× faster than old GHC machines!)
Profiling improved simple CUDA matmul ***

- `nvprof --analysis-metrics -f -o csimple.nvprof ./matrix -n 1024 -N 1024 -m csimple`
- `nvvp csimple.nvprof`

- Doing better!

- ...Still memory bound, though

*** Using deprecated profiling tools
CUDA disassembly + its limits

- You can look at PTX assembly:
  cuobjdump --dump-ptx matrix

- ...But you will not see difference in this case
  (Coalescing done by hardware, not compiler)

```
.visible .entry __Z19cudaSimpleKernel0ldiPfS_S__(_
...
ld.global.f32 %f6, [%rd9];
ld.global.f32 %f7, [%rd7];
...
st.global.f32 [%rd12], %f9;
...
.visible .entry __Z19cudaSimpleKerneliPfS_S__(_
...
ld.global.f32 %f6, [%rd9];
ld.global.f32 %f7, [%rd7];
...
st.global.f32 [%rd12], %f9;
...
```
Blocked matmul: Even better memory usage

- Problem: Entire matrix doesn’t fit in local cache

Classic solution: Block into sub-matrices that do fit in cache, and then multiply and sum sub-matrices
  - (This is just a re-association of the original computation)
Blocked matmul: C++ version

```c
void multMatrixBlocked(int N, float *matA, float *matB, float *matC) {
    /* Zero out C */
    memset(matC, 0, N * N * sizeof(float));
    int i, j, k;
    for (i = 0; i <= N - SBLK; i+= SBLK) {
        for (j = 0; j <= N - SBLK; j+= SBLK) {
            for (k = 0; k <= N - SBLK; k+= SBLK) {
                for (int bi = 0; bi < SBLK; bi++)  {
                    for (int bj = 0; bj < SBLK; bj++) {
                        float sum = 0.0;
                        for (int bk =0; bk < SBLK; bk++)
                            sum += matA[RM(i+bi,k+bk,N)] * matB[RM(k+bk,j+bj,N)];
                        matC[RM(i+bi,j+bj,N)] += sum;
                    }   }   }   }   }
```

Outer loops iterate over submatrices in steps of SBLK

Inner bi, bj loops iterate over submatrix and accumulate into output matrix

Note: This code assumes SBLK evenly divides N; need extra loops for “leftovers” in general
Benchmarking blocked matmul in C++

- ./matrix -n 1024 -N 1024 -m block

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

- Block C++: 500 ms, 4.3 Gflops
  - (Only 20% faster than old GHC machines)
Blocked matmul: CUDA version

1. Find the inner loop

2. Write it as a separate function

3. Compute indices from block/thread id
__global__ void 
cudaBlockKernelCoarse(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.y * blockDim.y + threadIdx.y; \( i \) *= LBLK; \( i \) \textbf{Map threads across submatrices}
    int j = blockIdx.x * blockDim.x + threadIdx.x; \( j \) *= LBLK;

    for (int bi = 0; bi < LBLK; bi++)
        for (int bj = 0; bj < LBLK; bj++)
            dmatC[RM(i+bi,j+bi,N)] = 0;

    for (int k = 0; k <= N-LBLK; k+=LBLK) {
        for (int bi = 0; bi < LBLK; bi++)
            for (int bj = 0; bj < LBLK; bj++) {
                float sum = 0.0;
                for (int bk = 0; bk < LBLK; bk++) {
                    sum += dmatA[RM(i+bi,k+bk,N)] * dmatB[RM(k+bk,j+bj,N)];
                }
                dmatC[RM(i+bi,j+bj,N)] += sum;
            }
    }
}
Blocked matmul: Attempt #1 + Local memory

```c
__global__ void cudaBlockKernelCoarse(int N, float *dmatA, float *dmatB, float *dmatC) {
  int i = blockIdx.y * blockDim.y + threadIdx.y; i *= LBLK;
  int j = blockIdx.x * blockDim.x + threadIdx.x; j *= LBLK;
  float subA[LBLK * LBLK];
  float subB[LBLK * LBLK];
  float subC[LBLK * LBLK];

  for (int bi = 0; bi < LBLK; bi++) /* Zero out C */
    for (int bj = 0; bj < LBLK; bj++)
      subC[RM(bi,bj,LBLK)] = 0;

  for (int k = 0; k <= N - LBLK; k+=LBLK) {
    for (int bi = 0; bi < LBLK; bi++) {
      for (int bj = 0; bj < LBLK; bj++) {
        subA[RM(bi,bj,LBLK)] = dmatA[RM(i+bi,k+bj,N)];
        subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j+bj,N)];
      }   }
    for (int bi = 0; bi < LBLK; bi++) {
      for (int bj = 0; bj < LBLK; bj++) {
        float sum = 0.0;
        for (int bk = 0; bk < LBLK; bk++) {
          sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
        }
        subC[RM(bi,bj,LBLK)] += sum;
      }   }
  }    
  for (int bi = 0; bi < LBLK; bi++)
    for (int bj = 0; bj < LBLK; bj++)
      dmatC[RM(i+bi,j+bj,N)] = subC[RM(bi,bj,LBLK)];
}
```

CMU 15-418/15-618, Spring 2020
Benchmarking blocked matmul

- ./matrix -n 1024 -N 1024 -m block

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

- Block C++: 500 ms, 4.4 Gflops
- Block CUDA: 107 ms, 20 Gflops 😞
  - (0% speedup over old GHC machines)
Profiling blocked matmul ***

- `nvprof --analysis-metrics -f -o ccblock.nvprof ./matrix -n 1024 -N 1024 -m ccblock`
- `nvvp ccblock.nvprof`

- Huh...

*** - Using deprecated profiling tools
Blocked matmul: What went wrong?

- How much parallelism is there in our first attempt?
  - Each thread generates $32 \times 32$ output elements
  - Each thread block is $32 \times 32$ threads
  - There are $1024 \times 1024$ output elements

- We only spawn one thread block!
- Need to split loops across more threads
Blocked matmul: Attempt #2

- Original matmul had one thread for each output element: $1024 \times 1024$ threads
  - 1 thread for each $i, j$ loop iteration in C++ code

- Idea: Unroll the inner $bi$ & $bj$ loops in Attempt #1 across a threads in a block
  - Thread block shares a single copy of submatrix
Blocked matmul: Attempt #2

```c
__global__ void cudaBlockKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int bi = threadIdx.y;
    int bj = threadIdx.x;

    __shared__ float subA[LBLK * LBLK];
    __shared__ float subB[LBLK * LBLK];
    float sum = 0;

    for (int k = 0; k < N; k += LBLK) {
        subA[RM(bi,bj,LBLK)] = dmatA[RM(i,k+bj,N)];
        subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j,N)];

        for (int bk = 0; bk < LBLK; bk++) {
            sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
        }
    }
    dmatC[RM(i,j,N)] = sum;
}
```

Each thread responsible for one output element (like original CUDA code)

But now mapped within a LBLK × LBLK block

Keep a block-shared copy of submatrix

Explicitly read from global to shared memory

Only reference shared copy in loop

Explicitly write from local to global memory

Is this code correct?
Blocked matmul: Attempt #2

__global__ void cudaBlockKernel(int N, float *dmatA, float *dmatB, float *dmatC) {
    int i = blockIdx.y * blockDim.y + threadIdx.y;
    int j = blockIdx.x * blockDim.x + threadIdx.x;
    int bi = threadIdx.y;
    int bj = threadIdx.x;

    __shared__ float subA[LBLK * LBLK];
    __shared__ float subB[LBLK * LBLK];
    float sum = 0;

    for (int k = 0; k < N; k += LBLK) {
        subA[RM(bi,bj,LBLK)] = dmatA[RM(i,k+bj,N)];
        subB[RM(bi,bj,LBLK)] = dmatB[RM(k+bi,j,N)];

        __syncthreads();

        for (int bk = 0; bk < LBLK; bk++) {
            sum += subA[RM(bi,bk,LBLK)] * subB[RM(bk,bj,LBLK)];
        }

        __syncthreads();
    }

    dmatC[RM(i,j,N)] = sum;
}
Benchmarking improved blocked matmul

- ./matrix -n 1024 -N 1024 -m cblock

- Simple C++: 1300 ms, 1.6 Gflops
- Simple CUDA: 33 ms, 65 Gflops
- Simple++ CUDA: 2.4 ms, 900 Gflops

- Block C++: 500 ms, 4.4 Gflops
- Block CUDA: 100 ms, 20 Gflops
- Block++ CUDA: 1.9ms, 1130 Gflops
  - (7% speedup over old GHC machines)
Benchmarking at 2048 × 2048
(8 × more work)

- `./matrix -n 2048 -N 2048 -m ...`

- Simple C++: 44000 ms, 0.4 Gflops
- Simple CUDA: 208 ms, 82 Gflops
- Simple++ CUDA: 18 ms, 940 Gflops

- Block C++: 5500 ms, 3.2 Gflops
- Block CUDA: 206 ms, 83 Gflops
- Block++ CUDA: 15 ms, 1180 Gflops

Big drop-off—data falls out of L3 cache

Big improvement—increased parallelism
Debugging tips and pitfalls

- `printf()` is available, but will reorder or lose output
  - So be cautious using `printf()` for debugging!

- Check your error codes

```c
#define CHK(ans) gpuAssert((ans), __FILE__, __LINE__);

void gpuAssert(CUDAError_t code, const char *file, int line){
    if (code != CUDASuccess)
        fprintf(stderr, "GPUassert: %s %s %s\n",
                CUDAGetErrorString(code), file, line);
}

#define POSTKERNEL CHK(CUDAPeekAtLastError())
```
Debugging tips and pitfalls

- Write reference version on host in C++

- Watch out for out-of-bounds memory errors (all kinds of crazy stuff will happen)

- Don’t assume stuff about N (e.g., that it’s a multiple of LBLK)

- cuda-gdb lets you step through + inspect code
Debugging tips and pitfalls

▪ What will happen here?

```c
for (int k = 0; k < N; k+= LBLK) {
   if (i >= N || j >= N) continue;
   // Some computation
   __syncthreads();
   // Some more computation
   __syncthreads();
}
```
Optimization advice

- Get the high-level abstraction + implementation first
  - Don’t start with low-level optimizations

- Use nvprof to figure out where your bottleneck is
  - Low utilization of compute + memory ⇝ no parallelism
  - Low utilization of compute ⇝ memory bound
  - Low utilization of memory ⇝ compute bound

- Memory is often key
  - E.g., when to use local/shared/global memory
CUDA syntax

- \_\_shared\_/\_\_global\_: Place variable in block-/device-shared memory
- cudaMalloc/cudaMemcpy/cudaFree: Manage device memory (flag sets to/from device)
- \_\_syncthreads\(): Barrier within a thread block
- kernel\(<<\text{blocks,threadsPerBlock}>>()\): Invoke kernel on device
- blockIdx/threadIdx: current block/thread idx
- blockDim/gridDim: Num threads per block/blocks per grid
CUDA as a vector processor

- NVIDIA has abused architecture terminology badly

<table>
<thead>
<tr>
<th>CUDA/GPU Terminology</th>
<th>Classic vector terminology</th>
</tr>
</thead>
<tbody>
<tr>
<td>Grid</td>
<td>Vectorizable loop</td>
</tr>
<tr>
<td>Thread</td>
<td>Loop iteration</td>
</tr>
<tr>
<td>Block</td>
<td>??</td>
</tr>
<tr>
<td>Warp</td>
<td>Thread</td>
</tr>
<tr>
<td>GPU/Device</td>
<td>Vector multicore</td>
</tr>
<tr>
<td>SM (streaming multiprocessor)</td>
<td>Core</td>
</tr>
<tr>
<td>Core</td>
<td>(Vector) Lane</td>
</tr>
<tr>
<td>Global memory</td>
<td>Memory</td>
</tr>
<tr>
<td>Shared memory</td>
<td>Local memory</td>
</tr>
<tr>
<td>Local memory</td>
<td>Registers</td>
</tr>
</tbody>
</table>