Day3

- Discussion on the parallel reduction algorithm
- Implementation the parallel matrix-matrix multiplication algorithm on GPU

Dr. Tuan-Tu Tran
trant@uni-mainz.de

Part 1

Discussion on the parallel reduction algorithm

[1] Mark Harris, Optimizing Parallel Reduction in CUDA, http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf

Ver-1: the branch divergence problem

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

`N = 64, threadsPerBlock = 64 (blockDim.x = 64)`

"active" threads in each loop:

S = 1 => s*2 = 2:
threads: 0, 2, 4, … 30, 32, … 62
S = 2 => s*2 = 4:
Threads: 0, 4, 8 … 28, 32, … 60
…
S = 16 => s*2 = 32
Threads: 0, 32
S = 32 => s*2 = 64
Threads: 0

There is always the branch divergence problem

Ver-2: reduce the branch divergence problem

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

`N = 64, threadsPerBlock = 64 (blockDim.x = 64)`

"active" threads in each loop:

Loop1: s = 1 => s*2 = 2:
    tid = 0: => sData[0], sData[1]
    tid = 1: => sData[2], sData[3]
    ...
    tid = 16: => sData[32], sData[33]
    ...
    tid = 31: => sData[62], sData[63]
Loop2: s = 2 => s*2 = 4:
    tid = 0: => sData[0], sData[2]
    tid = 1: => sData[4], sData[6]
    ...
    tid = 16: => index = 64 => not active
…
Loop5: S = 16 => s*2 = 32
    tid = 0 => sData[0], sData[16]
    tid = 1 => sData[32], sData[48]
    tid = 2 => not active
Loop6: S = 32 => s*2 = 64
    tid = 0 => sData[0], sData[32]
    tid = 1 => not active

No branch divergence

The branch divergence problem in the last 5 steps (Loop2 to Loop6)

Ver-2: The bank conflict problem

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

`N = 64, threadsPerBlock = 64 (blockDim.x = 64)`

“active” threads in each loop:

Loop1: Bank conflict between:
tid0 and tid16
tid1 and tid17
...

Loop2: Bank conflict between:
tid0 and tid8
tid1 and tid9
...

Loop1: s = 1 => s*2 = 2:
tid = 0: => sData[0], sData[1]
tid = 1: => sData[2], sData[3]
...
tid = 16: => sData[32], sData[33]
...
tid = 31: => sData[62], sData[63]
Loop2: s = 2 => s*2 = 4:
tid = 0: => sData[0], sData[2]
tid = 1: => sData[4], sData[6]
tid = 8: => sData[32], sData[34]
...
tid = 16: => index = 64 => not active
…
Loop5: S = 16 => s*2 = 32
tid = 0 => sData[0], sData[16]
tid = 1 => sData[32], sData[48]
tid = 2 => not active
Loop6: S = 32 => s*2 = 64
tid = 0 => sData[0], sData[32]
tid = 1 => not active

Ver-3: Free of bank conflict problem

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

`N = 64, threadsPerBlock = 64 (blockDim.x = 64)`

"active" threads in each loop:

Loop1: s = 32:
- tid = 0: => sData[0], sData[32]
- tid = 1: => sData[1], sData[33]
- ...
- tid = 16: => sData[16], sData[48]
- ...
- tid = 31: => sData[31], sData[63]

Loop2: s = 16:
- tid = 0: => sData[0], sData[16]
- tid = 1: => sData[1], sData[17]
- ...
- tid = 15: => sData[15], sData[31]
- tid = 16: => not active

…

Loop5: S = 2:
- tid = 0 => sData[0], sData[2]
- tid = 1 => sData[1], sData[3]
- tid = 2 => not active

Loop6: S = 1 => s*2 = 64
- tid = 0 => sData[0], sData[1]
- tid = 1 => not active

Ver-1: the branch divergence problem

```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

`N = 64, threadsPerBlock = 64 (blockDim.x = 64)`

“active” threads in each loop:

S = 1 => s*2 = 2:
    threads: 0, 2, 4, … 30, 32, … 62
S = 2 => s*2 = 4:
    Threads: 0, 4, 8 … 28, 32, … 60
…
S = 16 => s*2 = 32
    Threads: 0, 32
S = 32 => s*2 = 64
    Threads: 0

Free of the bank conflict problem

Full application

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

- Solution: cudaReduction.cu

(thanks to Tassilo Kugelstadt and Denise Scherzinger)

- Discussion:
  - Padding to the nearest power of 2
  - Rercusive loop in the host
  - The output array in the GPU

Part 2

Implementation the parallel matrix-matrix multiplication algorithm on GPU

Parallel matrix-matrix multiplication algorithm on GPU

JOHANNES GUTENBERG UNIVERSITÄT MAINZ

Cmn = Amp x Bpn

- Using 2-Dimension Block and Grid
- In this practical we use **square matrices of integer, where m = n = p = N**. You can use the same skeleton of the reduction application:
  - Assume that A = B (read data only 1 time)
  - Read N^2 integer, instead of N
- N is the power of 2, and up to 2^12 = 4K (4096).
- **Basic Implementations** (either Case 1 or Case 2)
  - 1 block, small data (N = 8)
  - Directly access to the global memory
  - Using shared memory
- **Advanced Implementation** (either Case 1 or Case 2)
  - Multiple block, large data
  - Using shared memory and apply optimizations presented in [1], from page 27 to 68

[2]. **Hendrik Lensch and Robert Strzodka**, Massively Parallel Computing With CUDA: Memory Access, http://www.mpi-inf.mpg.de/~strzodka/lectures/ParCo08/slides/Par02-Memory.pdf

Question ?