ECE408/CS483/CSE408 Spring 2018
Applied Parallel Programming

Lectures 5:
Locality and Tiled Matrix Multiplication

Objective

• To learn to evaluate the performance implications of global memory accesses
• To prepare for MP-3: tiled matrix multiplication

A Simple Matrix Multiplication Kernel
(Simplified Dimension and Syntax!)

__global__ void MatrixMulKernel(float* d_M, float* d_N, float* d_P, int Width)
{
    // Calculate the row index of the d_P element and d_M
    int Row = blockIdx.y*blockDim.y+threadIdx.y;
    // Calculate the column idenx of d_P and d_N
    int Col = blockIdx.x*blockDim.x+threadIdx.x;

    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        // each thread computes one element of the block sub-matrix
        for (int k = 0; k < Width; ++k){
            Pvalue += d_M[Row][k] * d_N[k][Col];
        }
        d_P[Row][Col] = Pvalue;
    }
}

// Setup the execution configuration
// TILE_WIDTH is a #define constant
dim3 dimGrid(ceil(Width/(TILE_WIDTH*1.0)),
            ceil(Width/(TILE_WIDTH*1.0)), 1);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH, 1);

// Launch the device computation threads!
MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);

You need to extend the code to handle rectangular matrix in MP-2!
How about performance on a device with 150 GB/s memory bandwidth?

- All threads access global memory for their input matrix elements.
  - Two memory accesses (8 bytes) per single-precision floating point multiply-add.
  - Two operands need to be fetched for each two floating-point operations (* and +).
  - Each floating-point operation needs 4 bytes of operand.
  - 150 GB/s limits the code at 37.5 (150/4) GFLOPS.
- The actual code runs at about 25 GFLOPS.
- Need to drastically cut down memory accesses to get closer to the peak of more than 1,000 GFLOPS.

Tiled Matrix-Matrix Multiplication using Shared Memory

A Common Programming Strategy

- Global memory is implemented with DRAM - slow.
- A profitable way of performing computation on the device is to tile the input data to take advantage of fast shared memory:
  - Partition data into subsets (tiles) that fit into the (smaller but faster) shared memory.
  - Handle each data subset with one thread block by:
    • Loading the subset from global memory to shared memory, using multiple threads to exploit memory-level parallelism.
    • Performing the computation on the subset from shared memory, each thread can efficiently access any data element.
    • Copying results from shared memory to global memory.
- Tiles are also called blocks in the literature.

Declaring Shared Memory Arrays

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width) {
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
```
Shared Memory Tiling Basic Idea

Thread 1

Thread 2

…

in Global Memory

Global Memory

in

On-chip Memory

Thread 1

Thread 2

…

Outline of Technique

• Identify a tile of global data that are accessed by multiple threads
• Load the tile from global memory into on-chip memory
• Have the multiple threads to access their data from the on-chip memory
• Move on to the next block/tile

Idea: Place global memory data into Shared Memory for reuse

• Each input element is used in calculating by WIDTH P elements.
• Load each element into Shared Memory and have several threads use the local version to reduce the memory bandwidth

Tiled Multiply

• Break up the execution of the kernel into phases so that the data accesses in each phase are focused on one subset (tile) of M and N
Loading a Tile

- All threads in a block participate
  - Each thread loads one M element and one N element in basic tiling code

- Assign the loaded element to each thread such that the accesses within each warp is coalesced (more later).
Loading an Input Tile 0

Tile 0 2D indexing for each thread:

\[ M[\text{Row}][tx] \]
\[ N[ty][\text{Col}] \]

Loading an Input Tile 1

Accessing tile 1 in 2D indexing:

\[ M[\text{Row}][1*TILE\_WIDTH+tx] \]
\[ N[1*TILE\_WIDTH+ty][\text{Col}] \]
Loading an Input Tile

However, recall that M and N are dynamically allocated and can only use 1D indexing:

\[
M[Row \cdot m \cdot \text{TILE_WIDTH} + \text{tx}] \\
N[\text{m} \cdot \text{TILE_WIDTH} + \text{ty}] \\
N[(\text{m} \cdot \text{TILE_WIDTH} + \text{ty}) \cdot \text{Width} + \text{Col}]
\]

Accessing a Tile

To perform the \( k^{th} \) step of the product within the tile:

\[
\text{subTileM}[\text{ty}]_k \\
\text{subTileN}[k][\text{tx}]
\]

Barrier Synchronization

- An API function call in CUDA
  - \text{__syncthreads()}

- All threads in the same block must reach the \text{__syncthreads()} before any can move on

- Best used to coordinate tiled algorithms
  - To ensure that all elements of a tile are loaded
  - To ensure that all elements of a tile are consumed
Tiled Matrix Multiplication Kernel

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    __shared__ float subTileM[TILE_WIDTH][TILE_WIDTH];
    __shared__ float subTileN[TILE_WIDTH][TILE_WIDTH];
    int bx = blockIdx.x; int by = blockIdx.y;
    int tx = threadIdx.x; int ty = threadIdx.y;

    // Identify the row and column of the P element to work on
    int Row = by * TILE_WIDTH + ty;
    int Col = bx * TILE_WIDTH + tx;
    float Pvalue = 0;

    // Loop over the M and N tiles required to compute the P element
    // The code assumes that the Width is a multiple of TILE_WIDTH!
    for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        subTileM[ty][tx] = M[Row*Width + m*TILE_WIDTH+tx];
        subTileN[ty][tx] = N[(m*TILE_WIDTH+ty)*Width+Col];
        __syncthreads();
        for (int k = 0; k < TILE_WIDTH; ++k)
            Pvalue += subTileM[ty][k] * subTileN[k][tx];
        __syncthreads();
    }
    P[Row*Width+Col] = Pvalue;
}
```

Compare with Basic MM Kernel

```c
__global__ void MatrixMulKernel(float* M, float* N, float* P, int Width)
{
    int Row = blockIdx.y * blockDim.y + threadIdx.y;
    int Col = blockIdx.x * blockDim.x + threadIdx.x;
    if ((Row < Width) && (Col < Width)) {
        float Pvalue = 0;
        for (int k = 0; k < Width; ++k)
            Pvalue += M[Row*Width+k] * N[k*Width+Col];
        P[Row*Width+Col] = Pvalue;
    }
}
```

Shared Memory and Threading

- Each SM in Maxwell has 64KB shared memory (48KB max per block)
  - Shared memory size is implementation dependent!
  - For TILE_WIDTH = 16, each thread block uses 2*256*4B = 2KB of shared memory.
    - Shared memory can potentially support up to 32 active blocks
    - The threads per SM constraint (2,048) will limit the number of blocks to 8
    - This allows up to 8*512 = 4,096 pending loads. (2 per thread, 256 threads per block)
  - TILE_WIDTH 32 would lead to 2*32*32*4B = 8KB shared memory usage per thread block,
    - Shared memory can potentially support up to 8 active blocks
    - The threads per SM constraint (2,048) will limit the number of blocks to 2
    - This allows up to 2*2,048 = 4,096 pending loads (2 per thread, 1,024 threads per block)

Memory Bandwidth Consumption

- Using 16x16 tiling, we reduce the global memory by a factor of 16
  - Each operand is now used by 16 floating-point operations
  - The 150GB/s bandwidth can now support (150/4)*16 = 600 GFLOPS!
- Using 32x32 tiling, we reduce the global memory accesses by a factor of 32
  - Each operand is now used by 32 floating-point operations
  - The 150 GB/s bandwidth can now support (150/4)*32 = 1,200 GFLOPS!
  - The memory bandwidth is no longer a limiting factor for performance!
Device Query

- Number of devices in the system
  ```c
  int dev_count;
  cudaGetDeviceCount( &dev_count);
  ```
- Capability of devices
  ```c
  cudaDeviceProp dev_prop;
  for (i = 0; i < dev_count; i++) {
    cudaGetDeviceProperties( &dev_prop, i);
    // decide if device has sufficient resources and capabilities
  }
  ```
- `cudaDeviceProp` is a built-in C structure type
  - `dev_prop.dev_prop.maxThreadsPerBlock`
  - `Dev_prop.sharedMemoryPerBlock`
  - ...

© David Kirk/NVIDIA and Wen-mei W. Hwu, 2007-2018 ECE408/CS483/
University of Illinois at Urbana-Champaign

ANY MORE QUESTIONS?
READ CHAPTER 4!