Lecture 4: GPU Optimization Part 1
Block Management
Thread Switching
Matrix Problem
Block Management
• Why do we need to manage blocks?
  • Recall from last time

We mapped every cell to a block and every thread at a point
• For 3 points – Only using 3 threads out of 1024 available
• We are wasting resources – Occupancy
  • Occupancy metric is the number of available threads divided by total threads per processor
  • Basically – Number of thread groups active at one time
  • Use GPU occupancy calculators for this
  • How do we improve our problem?

\[
\text{block } 0 \quad t_1 \quad t_2 \quad t_3
\]

Extend the blocks
• Extend the block stencil to encompass multiple cells

• How far do we go?
  • Try to get a Warp – Guarantees high Occupancy
Now we can use multiple thread dimensions
• Define a max size – BLOCKSIZE
• Some multiple of 32

```c
#define nBlocks 32
int main() {
    //...
    int cells_per_block = nBlocks / np;
    dim3 threads(np, cells_per_block);
    int blocks((ne / cells_per_block)+1);
    //...
    return 0;
}
```
Consider $n_{sp} = 3$, $n_c = 12$

$\text{cells} = \frac{32}{3} = 10.666667 = 10$

$\text{threads} = (3, 10), \text{blocks} = \left(\frac{12}{10} + 1\right) = 2$
Block Management

• Now we must properly manage the kernel

```c
__global__ void interpolate(...) {

    int j = threadIdx.x;  // Data point
    int ik = threadIdx.y;  // Cell per block

    // Cell
    int k = ik + blockIdx.x*blockDim.y;

    if (k < ne) {
        // Run calculations
    }
}
```

• Threads are now close to a warp – Need if statement
Block Management

• Increase number of cells per block
  • Increases computations per processor
  • Tries to fill in a warp
  • Conditional statement is cheap – Why?
• Coding more general
• No real rule yet for optimal speed
  • Sometimes 64, 128, 256, 512, ... is much better
  • Just try things out and see what is best
• Coding requires
  • Multiple blocks and thread definitions
  • Can really complicate programs
Thread Switching
Thread Switching

• Thread switching is very simple – but very powerful
  • Allows the code to switch threads mid kernel
  • Conditional statement control
  • Useful if you have multiple points

• Interpolate from 1 set of points to another
• Allows more operations per kernel
Thread Switching

• An example so you can see

```c
__global__ void interpolate() {

    int j = threadIdx.x; //Squares or Circles

    if (j < Squares) {
        //Operate on squares...all threads active
        //Save data to shared memory to share with circles
    }

    //Code needs to know we are done with Squares
    __syncthreads();

    if (j < Circles) {
        //Operate on circles now
        //Read data from square calculation from shared
    }
}
```
Thread Switching

• Benefits
  • Much work completed in one kernel
  • Would need multiple kernels without
  • Data does not have to be reloaded

• Problems
  • When switching – Threads become deactivated
  • Can lead to performance issues if many threads are “waiting”

• Will try to get to an example of this later
Matrix Problem
Matrix Problem

• We are going to write and optimize a code
• Consider matrix multiplication

\[ A_{ij} = B_{ik} C_{kj} \]

\[
\begin{bmatrix}
  a_{11} & a_{12} \\
  a_{21} & a_{22}
\end{bmatrix}
= \begin{bmatrix}
  b_{11} & b_{12} \\
  b_{21} & b_{22}
\end{bmatrix}
\times
\begin{bmatrix}
  c_{11} & c_{12} \\
  c_{21} & c_{22}
\end{bmatrix}
\]

\[
a_{11} = b_{11} c_{11} + b_{12} c_{21} \quad a_{12} = b_{11} c_{12} + b_{12} c_{22} \]

\[
a_{21} = b_{21} c_{11} + b_{22} c_{21} \quad a_{22} = b_{21} c_{12} + b_{22} c_{22}
\]
Matrix Problem

• CPU Code

```c
void MatMult(double *A, double *B, double *C, int n) {
    double sum;
    for (int i=0; i<n; i++)
        for (int j=0; j<n; j++) {
            sum = 0.0;
            for (int k=0; k<n; k++) {
                sum += B[k + i*n] * C[j + k*n];
            }
            A[j + i*n] = sum;
        }
}
```

• How do we put this on GPU?
Matrix Problem

• Let each matrix be $(n \times n)$

\[ \vec{b} = \text{ceil} \left[ \frac{n}{W}, \frac{n}{W} \right] \quad \vec{t} = [W, W] \quad W = 32, 64, 128 \ldots \]

• If we have $n = 186$

\[ \vec{b} = \text{ceil} \left[ \frac{186}{32}, \frac{186}{32} \right] = [6, 6] \quad \vec{t} = [32, 32] \]

• Gives total thread grid of $(192, 192)$
  • Over 2000 threads doing nothing – See why matrices are bad?
Matrix Problem

• So the thread grid is awful if we generalize the problem
  • Regardless – still a good programming exercise
  • “Bare-Bones” GPU code

```c
__global__ void MatrixMult(double *A, double *B, double *C, int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x; //Column
    int j = threadIdx.y + blockIdx.y*blockDim.y; //Row

    for (int k=0; k<n; k++) {
        A[i + j*n] += B[k + i*n] * C[j + k*n];
    }
}
```

• Tell me all that is bad! (There are 3)
Matrix Problem

- Two are easy to find – The third one...
- Everything is limited by memory access
  - For your simple codes – you really didn’t care...but now...
  - Be careful how you store and grab data – Very important

```
__global__ void MatrixMult(double *A, double *B, double *C, int n)
{
    int i = threadIdx.x + blockIdx.x*blockDim.x; //Column
    int j = threadIdx.y + blockIdx.y*blockDim.y; //Row

    for (int k=0; k<n; k++) {
        A[i + j*n] += B[k + i*n] * C[j + k*n];
    }
}
```

THIS!!!
Matrix Problem

• You tell me how to fix these things
• How do we fix repeated writes to global memory?

```c
__global__ void MatrixMult(double *A, double *B, double *C, int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x; //Column
    int j = threadIdx.y + blockIdx.y*blockDim.y; //Row

    for (int k=0; k<n; k++) {
        A[i + j*n] += B[k + i*n] * C[j + k*n];
    }
}
```

• Quick and easy optimization!

Employ registers to run calculation
Matrix Problem

```c
__global__ void MatrixMult(double *A, double *B, double *C, int n) {
    int i = threadIdx.x + blockIdx.x*blockDim.x; // Collumn
    int j = threadIdx.y + blockIdx.y*blockDim.y; // Row

    double sum = 0.0;

    for (int k=0; k<n; k++) {
        sum += B[k + i*n] * C[j + k*n];
    }

    A[i + j*n] = sum;
}
```

• We now have fast register calculations
Matrix Problem

• Now try to fix:

```c
for (int k=0; k<n; k++) {
    sum += B[k + i*n] * C[j + k*n];
}
```

• Tell me what do I need to do?
  • Repeated access from 1 thread to multiple threads
  • Ring a bell?

• Shared memory is perfect – But wait...
What if the memory is outside my scope?

- Shared memory is local to the block
- Block 0 won’t see Block 2
- Have to be creative
- As a side note
  - Again, see how annoying this is getting?
  - We like to avoid matrices on GPUs if we can
Matrix Problem

• Solution is complicated
  • Do you see what we can do?
  • It is not trivial...
__global__ void MatrixMult(double *A, double *B, double *C, int n)
{
    int i  = threadIdx.x; //Column in block
    int j  = threadIdx.y; //Row in block
    int ii = i + blockIdx.x*blockDim.x;
    int jj = j + blockIdx.y*blockDim.y;

    __shared__ double B_s[W*W];
    __shared__ double C_s[W*W];
    double sum = 0.0;

    for (int m=0; m<n/W; m++) {
        B_s[i + j*W] = B[(m*W + blockIdx.y*W*n) + (i + j*n)];
        C_s[i + j*W] = C[(m*W*n + blockIdx.x*W) + (i + j*n)];
        __syncthreads();

        for (int k=0; k<W; k++) {
            sum += B_s[k + i*W] * C_s[j + k*W];
        }
    }
    __syncthreads();
    A[ii + jj*n] = sum;
}
Matrix Problem

• Was is worth it?
  • Shared memory implementation ~ 7x faster
  • So yes – definitely worth it
  • But what a pain!
Wrap Up

• Block management is important
  • Try to get a Warp to get best performance
  • Sometimes you just can’t

• Thread switching is powerful
  • Easy to use but watch out for inactive threads!

• Matrix problems
  • Can be a real pain – Try to avoid them
  • Shared memory is very powerful – Complicated to use