CUDA

Kernels – Reduction and Memory Coalescence
Warps

- Thread blocks partitioned into warps (group of threads) based on thread block indices.
- (Warp is a term used in weaving. Warp threads are interwoven with weft threads.)
- **32 threads per warp** (Compute capability 1.0 – 5.2)
Reduction - CPU

float data[];
float sum = 0.0;
...
for(int i = 0; i < N; i++)
    sum += data[i];
Reduction – version 1

__global__ void kernel_func ()
{
    ...
    unsigned int t = threadIdx.x;
    sum += data[t];  // Race condition!
}
Reduction – version 1

```
data  0  1  2  3  4  5  6  7  8  9  10  11

sum   ...
```

...
Atomics

- Atomic functions – performed without interference from other threads
- \( T \) \texttt{atomic} \( OP(T * \text{address}, T \text{value}); \)
  - \( T \) is subset of \texttt{int}, \texttt{unsigned int}, \texttt{unsigned long long int}, \texttt{float}

  \begin{verbatim}
  old = *address;  // read
  t = old \texttt{OP value};  // modify
  *address = t;  // write
  return old;
  \end{verbatim}

- \( \texttt{OP} = \texttt{Add}, \texttt{Sub}, \texttt{Inc}, \texttt{Dec}, \texttt{And}, \texttt{Or}, \texttt{Xor}, \texttt{Exch}, \texttt{Min}, \texttt{Max}, \texttt{CAS} \)
- See CUDA C Programming Guide (Appendix on Atomics) for details
Reduction – version 1.5

__global__ void kernel_func ()
{
...
    unsigned int t = threadIdx.x;
    atomicAdd(&sum, data[t]);
    // Correct but slow
}
Reduction – version 2

__shared__ float partialSum[]
unsigned int t = threadIdx.x;
for(unsigned int stride = 1;
    stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if (t % (2 * stride) == 0)
        partialSum[t] += partialSum[t+stride];
}
CUDA and SIMT

- CUDA is SIMT – Single Instruction Multiple Thread execution
- Instructions issued for single warp
- Decisions - If all threads in a single warp take the same path through a decision there is no performance penalty
- Different warps can take different paths with no performance penalty
- Threads in a warp taking different paths is known as thread divergence and severely impacts performance
Thread Divergence

- Reduction version 2 suffers from severe thread divergence
- Trick is to reorder memory accesses so that all threads in a warp take the same execution path as much as possible.
Reduction – version 3

blockDim.x = 512

Iteration 1

Iteration 2

...
Reduction – version 3

__shared__ float partialSum[]
unsigned int t = threadIdx.x;
for(unsigned int stride = blockDim.x >> 1;
    stride > 0; stride >>= 1)
{
    __syncthreads();
    if (t < stride)
        partialSum[t] += partialSum[t+stride];
}
Memory coalescence

- CUDA hardware does memory coalescence
  - Simultaneous thread requests for adjacent memory are combined (coalesced) into a single memory operation.
Memory coalescence

- CPU
  
  ```c
  for(int row; row < M; row++)
    for(int col; col < N; col++)
      process data[row][col];
  ```

- This version preferable (in C/C++) to version with swapped `for`s because of cache coherence.

- This wisdom thrown out by CUDA (!)
Tiled Matrix Multiplication

\[ P = M \times N \]

Focus on single thread misleading

\[ M \quad \text{m}x\text{k} \]

Single thread, contiguous memory, row elements.
Good? (No)

\[ P \quad \text{m}x\text{n} \]

Single thread, non-contiguous memory, column elements.
Bad? (No)

\[ N \quad \text{k}x\text{n} \]

\[ p_{i,j} = \sum_{k} m_{i,k} n_{k,j} \]

TILE_WIDTH

Single thread misleading
Tiled Matrix Multiplication

__global__ void MatrixMulKernel(float * Md, float * Nd, float * Pd, int Width) {
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;

    for(int k = 0; k < Width; k++)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    Pd[Row*Width+Col] = Pvalue;
}
Tiled Matrix Multiplication
Memory Coalescence

\[ P = M \times N \]

Focus on multiple threads reveals memory coalescence

\[ M \quad \text{mxk} \]

\[ P \quad \text{mxn} \]

\[ N \quad \text{kxn} \]

\[ p_{i,j} = \sum_k m_{i,k} n_{k,j} \]

Multi thread, single iteration, coalesced memory. Good

Multi thread, single iteration, non-coalesced memory. Bad
Tiled Matrix Multiplication

```c
__global__ void MatrixMulKernel(float * Md,
                                  float * Nd, float * Pd, int Width)
{
    int Row = blockIdx.y * TILE_WIDTH + threadIdx.y;
    int Col = blockIdx.x * TILE_WIDTH + threadIdx.x;

    float Pvalue = 0;

    for(int k = 0; k < Width; k++)
        Pvalue += Md[Row * Width + k] * Nd[k * Width + Col];

    (Single thread)          contiguous                            non-contiguous
    (Multi thread)           non-coalesced                               coalesced

    Pd[Row*Width+Col] = Pvalue;
}
```
Memory Coalescence Solutions

• Copy global memory to shared memory
  – Non-coalescence not (as much of) an issue in shared memory
  – This is effective only for multiple memory accesses

• Reorder memory accesses
  – Not always possible