CUDA Performance Optimization
GPU Programming with CUDA

April 25-27, 2016 | Jiri Kraus (NVIDIA) based on work by Andrew V. Adinetz
What you will learn:

- What is memory coalescing
- What is branch divergence

What you will not learn (in this session):

- Exposing enough parallelism
- Expressing data locality
Motivating Example: Matrix Transpose

CPU VERSION:
void transpose(
    Integer* a_trans,
    Integer* a,
    Integer n ) {
    for ( Integer row = 0; row < n; ++row ){
        for ( Integer col = 0; col < n; ++col )
        {
            a_trans[col][row] = a[row][col];
        }
    }
}

CUDA VERSION:
__global__ void transpose(
    Integer* a_trans,
    Integer* a,
    Integer n ) {
    Integer row = bIdx.y*bDim.y+tIdx.y;
    Integer col = bIdx.x*bDim.x+tIdx.x;
    if ( row < n && col < n )
    {
        a_trans[col][row] = a[row][col];
    }
}
Motivating Example: Matrix Transpose (demo)
Memory Transactions and Coalescing

- Access to global memory triggers transactions
  - size of 32 or 128 bytes
  - Transaction are aligned to its size
  - Transaction are always R/W fully

Coalescing

- adjacent threads can share transactions

\[
\text{degree of coalescing} = \frac{\text{#bytes requested}}{\text{#bytes read}}
\]

- more memory read than used => performance penalty
Coalescing Details

- Details in CUDA Programming Guide
- L1-cached access
  - 128-byte transactions
  - Used local memory for CC 3.x
- L2-cached access
  - 32-byte transactions
  - default for CC 3.x (Kepler) global memory

April 20-22, 2015
GPU Programming with CUDA
L2-Cached Thread Index Access

- 32 adjacent threads requesting 32 aligned or permuted 4 byte words
- All addresses fall within 4 segments
- Bus utilization: 100%
- Transactions: 4

\[ a[tid] \]
L2-Cached Shifted Access

- 32 adjacent threads requesting 32 aligned 4 byte words
- All addresses fall within 5 segments
- 160 bytes move across bus while 128 bytes are required
- Bus utilization: 80%
- Transactions: 5

\[ a[tid - 1] \]
L2-Cached Single Access

- 32 adjacent threads requesting the same 4 byte word
- Addresses fall in 1 segment
- Warp requires 4 bytes but 32 bytes transferred
- Bus utilization is only 12.5%
L2-Cached Strided Access

- 32 adjacent threads requesting 32 4-byte words with stride 3
- All addresses fall in 12 segments
- Bus utilization: 33%
- Transactions: 12

struct {float x,y,z;} a; ... a[tid].x

- use structure-of-arrays (SoA)
- float a[M][N]; ... a[tid][42]

- multi-dimensional arrays: pay attention to coalescing
L2-Cached Fully Random Access

- 32 adjacent threads requesting 32 4-byte random words
- All addresses fall in 32 segments
- Bus utilization: only 12.5%
- Transactions: 32

```latex
\text{a[b[tid]]}
```

- pointer chasing: lists, trees etc.
Matrix Transpose Access Pattern

Stride-1 column access (thread view) is actually stride-N access in memory

Source: GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero (NVIDIA)
Matrix Transpose using shared memory

- Row is loaded fully coalesced

Source: GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero (NVIDIA)
Matrix Transpose using shared memory

- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal

Source: GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero (NVIDIA)
Matrix Transpose using shared memory

- Row is loaded fully coalesced
- Indexing: location of block is flipped across the diagonal
- Column of shared is written to a row of the matrix
  - Transposes the Block
  - Coalesced Write

Source: GTC 2015 - Memory Bandwidth Bootcamp: Best Practices by Tony Scudiero (NVIDIA)
Using NVidia Visual Profiler

- Experiments:
  - memory access pattern
  - divergent execution

- Show in source code
  - `nvcc -lineinfo` ...
Task: Coalesced Matrix Transpose

- Use shared memory to coalesced writes to a_trans
  - Follow TODOs in
    
    CUDATranspose\exercises\tasks\transpose.cu

- Check solution with “Memory Access Pattern” experiment

- Solution is in
  
  CUDATranspose\exercises\solutions\transpose.cu

- Slides are in CUDATranspose\slides\CUDATranspose.pdf
Warps

- GPUs use the Single Instruction Multiple Threads (SIMT) execution
  - functionally transparent to the programmer
  - but has performance implications
- warp
  - group of synchronously executing threads
  - neighbor threads (mostly x dimension)
  - 32 threads (NVIDIA), 64 threads (AMD, =wavefronts)
  - basic unit of scheduling
Branch Divergence Within Warp

// ...  
if (condition) {
  // do smth
  // ...
} else {
  // do smth else
  // ...
}
// ...

divergence within warp => performance penalty

if (threadIdx.x % 2 == 0) ...

all threads running

"else" threads masked

"if" threads masked

all threads running
Branch Divergence Between Warps

// ...  
if(condition) {  
    // do smth  
    // ...  
} else {  
    // do smth else  
    // ...  
}
// ...

divergence between warps => no penalty
if(blockIdx.x % 2 == 0) ...
Conclusion

- To achieve coalesced global memory access
  - Try to use shared memory
  - Look for different way of storage or better algorithm
- Avoid divergent branches
- Use the tools