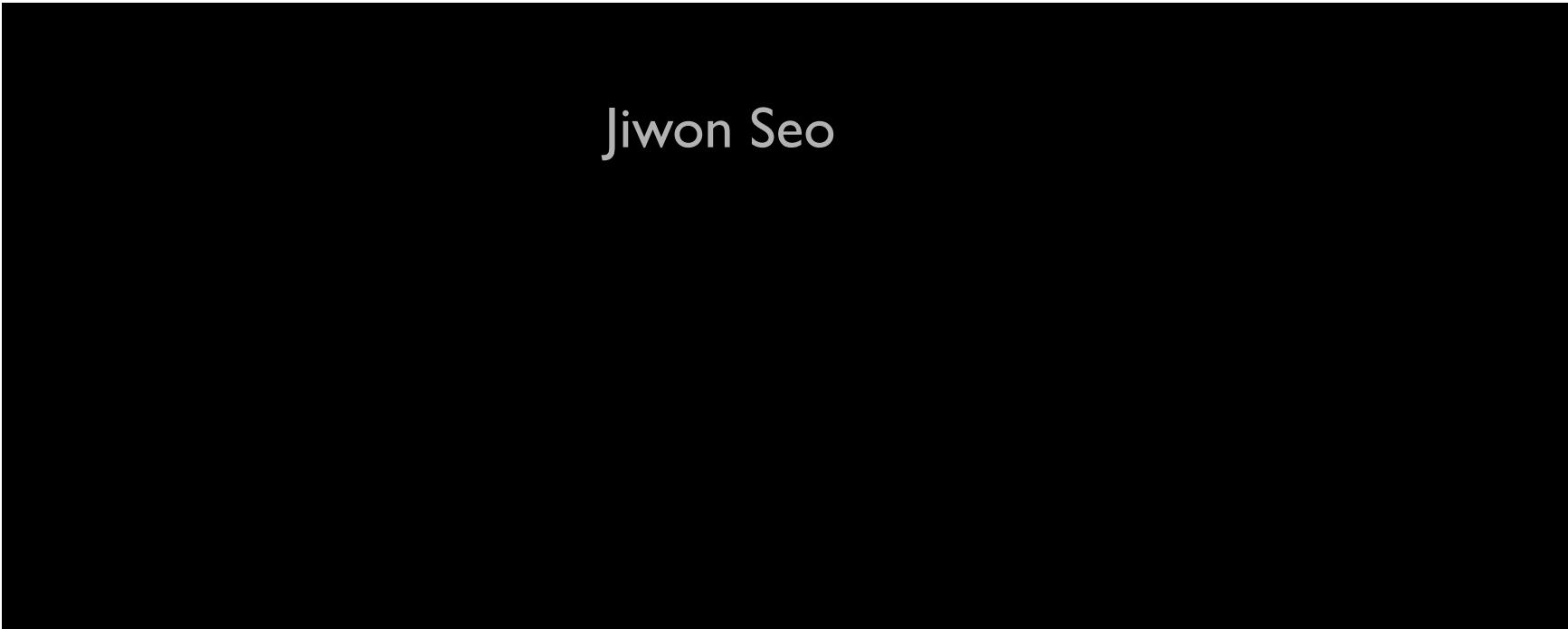




CUDA, Parallel Programming Language

CUDA Performance Consideration



Jiwon Seo

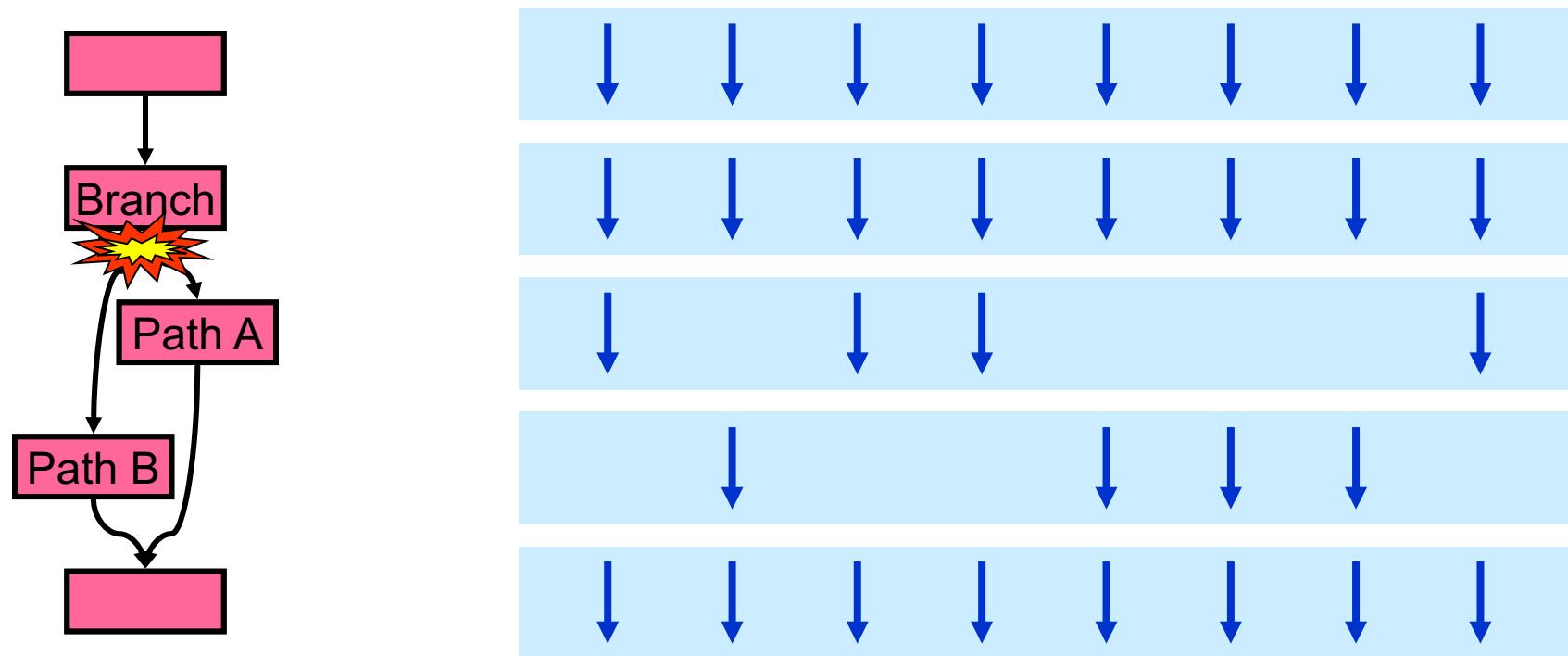
Topics

- Warp Divergence
- Occupancy
- Speedup with Shared Memory

Control Flow Divergence

```
if (foo(threadIdx.x) )  
{  
    do_A();  
}  
else  
{  
    do_B();  
}
```

Control Flow Divergence

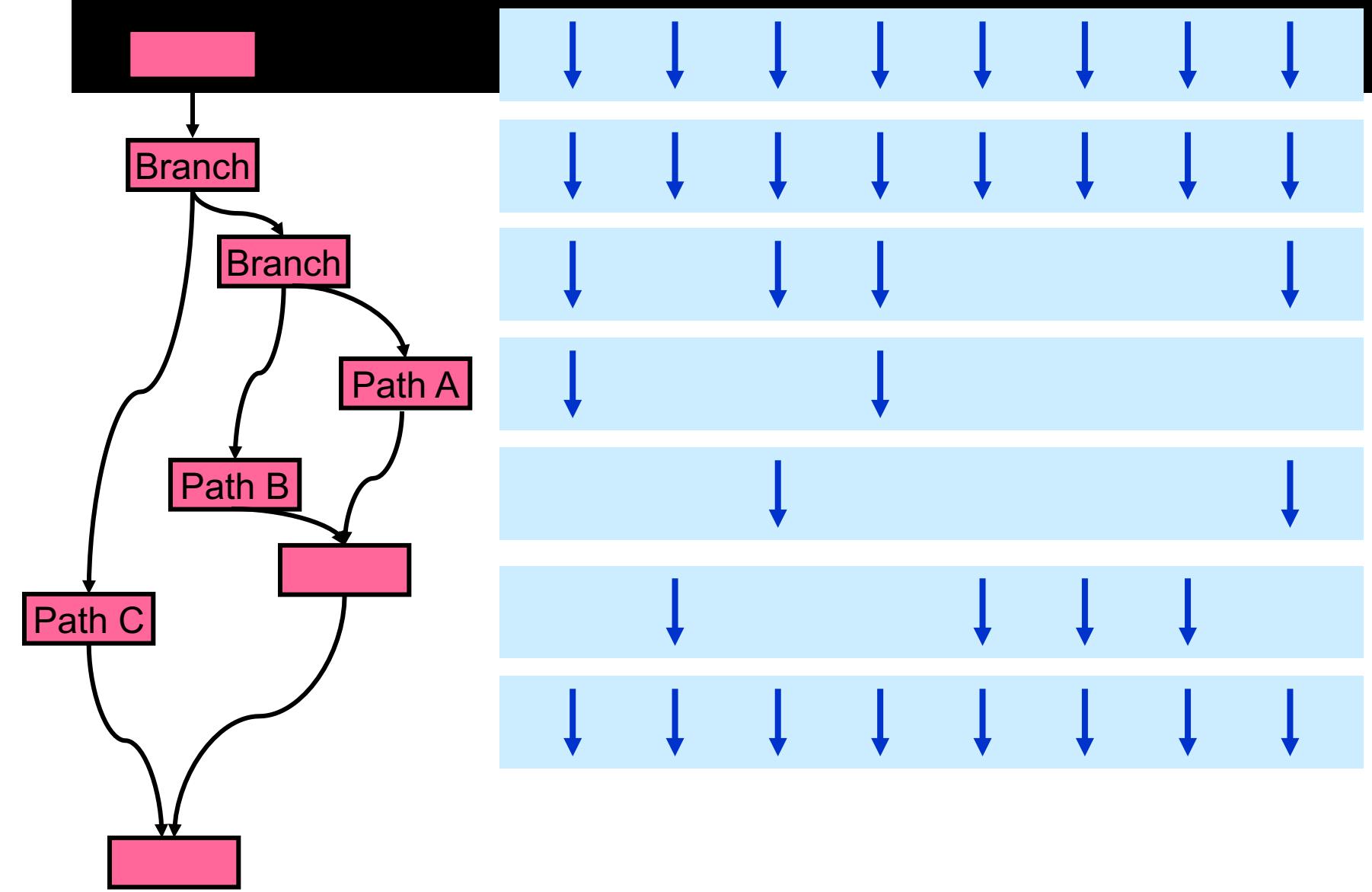


Control Flow Divergence

- Nested branches are handled as well

```
if(foo(threadIdx.x))  
{  
    if(bar(threadIdx.x))  
        do_A();  
    else  
        do_B();  
}  
else  
    do_C();
```

Control Flow Divergence



Control Flow Divergence

- You don't have to worry about divergence for correctness (*)
- You might have to think about it for performance
 - Depends on your branch conditions

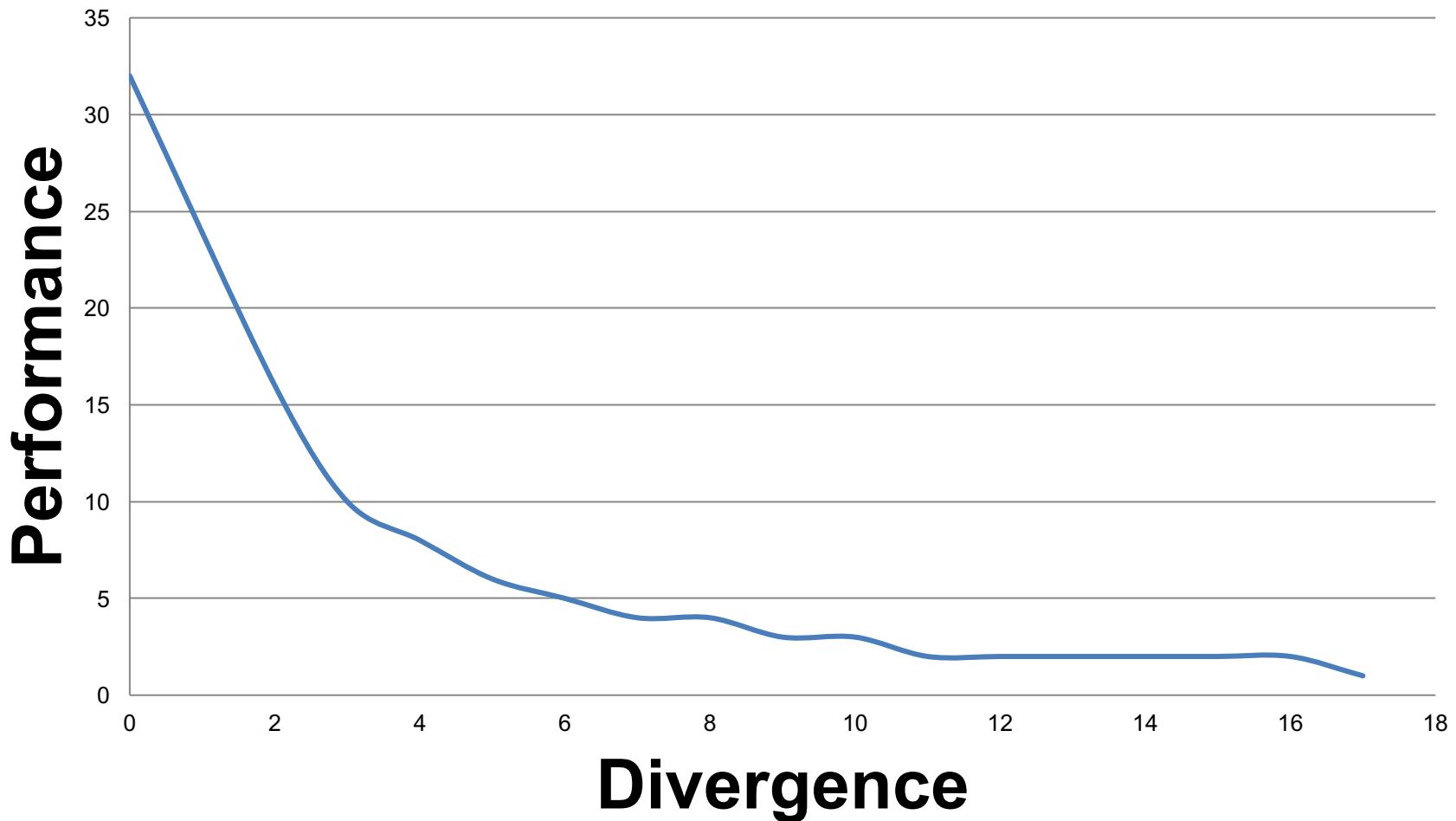
syncthread -> warp안 스레드 중 하나라도 reach되면 블락 안의 전체 스레드가 닿았다고 판단,,
if 등 닿지 않는 조건이 생기면 dead lock 발생하거나 의도와 다르게됨

Control Flow Divergence

- Performance drops off with the degree of divergence

```
switch (threadIdx.x % N)
{
    case 0:
        ...
    case 1:
        ...
}
```

Divergence & Performance



Occupancy

- **Occupancy = Active Warps / Maximum Active Warps**
- **Remember: resources are allocated for the entire block**
 - Resources are finite
 - Utilizing too many resources per thread may limit the occupancy
- **Potential occupancy limiters:**
 - Register usage
 - Shared memory usage
 - Block size

Occupancy Limiters: Registers

- Register usage: compile with `--ptxas-options=-v`
- Fermi has 32K registers per SM
- Example 1
 - Kernel uses 20 registers per thread (+1 implicit) 커널 usage
 - Active threads = $32K/21 = 1560$ threads
 - > 1536 thus an occupancy of 1
- Example 2
 - Kernel uses 63 registers per thread (+1 implicit)
 - Active threads = $32K/64 = 512$ threads
 - $512/1536 = .3333$ occupancy
- Can control register usage with the nvcc flag: `--maxrregcount`

Occupancy Limiters: Shared Memory

- **Shared memory usage: compile with --ptxas-options=-v**
 - Reports shared memory per block
- **Fermi has either 16K or 48K shared memory**
- **Example 1, 48K shared memory**
 - Kernel uses 32 bytes of shared memory per thread
 - $48K/32 = 1536$ threads
 - occupancy=1
- **Example 2, 16K shared memory**
 - Kernel uses 32 bytes of shared memory per thread
 - $16K/32 = 512$ threads
 - occupancy=.3333
- **Don't use too much shared memory**
- **Choose L1/Shared config appropriately.**

Occupancy Limiters: Thread Block Size

- **Each SM can have up to 8 active blocks**
- **A small block size will limit the total number of threads**

Block Size	Active Threads	Occupancy
32	256	.1666
64	512	.3333
128	1024	.6666
192	1536	1
256	2048 (1536)	1

- **Avoid small block sizes, generally 128-256 threads is sufficient**

Cuda Occupancy Calculator

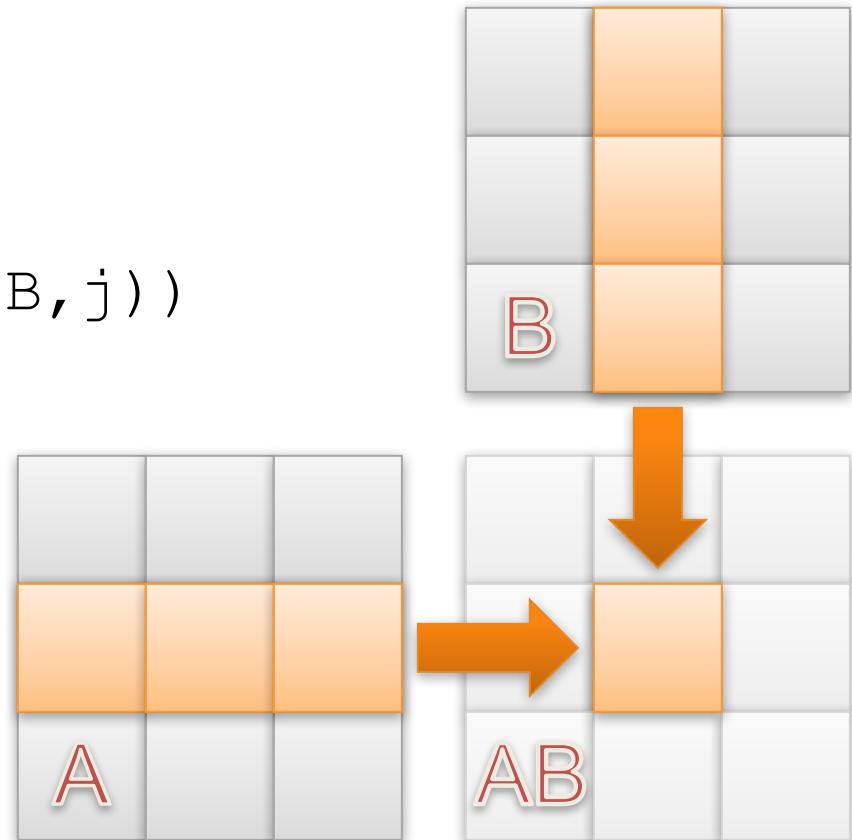
- A tool to help you investigate occupancy
- http://developer.download.nvidia.com/compute/cuda/4_0/sdk/docs/CUDA_Occupancy_Calculator.xls
- Demo: [CUDA Occupancy calculator.xls](#)

Occupancy Wrap-up

- In order to achieve peak global memory bandwidth we need to have enough transactions in flight to hide latency
- We can increase the number of transactions by
 - Increasing occupancy
 - Increasing instruction level parallelism
- Occupancy can be limited by
 - Register usage
 - Shared memory usage
 - Block size
- Use the cuda occupancy calculator and the visual profiler to investigate memory bandwidth/occupancy

Matrix Multiplication Example

- Generalize adjacent_difference example
- $AB = A * B$
 - Each element AB_{ij}
 - $= \text{dot}(\text{row}(A, i), \text{col}(B, j))$
- Parallelization strategy
 - Thread $\rightarrow AB_{ij}$
 - 2D kernel



First Implementation

```
__global__ void mat_mul(float *a, float *b,
                        float *ab, int width) {
    // calculate the row & col index of the element
    int row = blockIdx.y*blockDim.y + threadIdx.y;
    int col = blockIdx.x*blockDim.x + threadIdx.x;

    float result = 0;
    // do dot product between row of a and col of b
    for(int k = 0; k < width; ++k)
        result += a[row*width+k] * b[k*width+col];

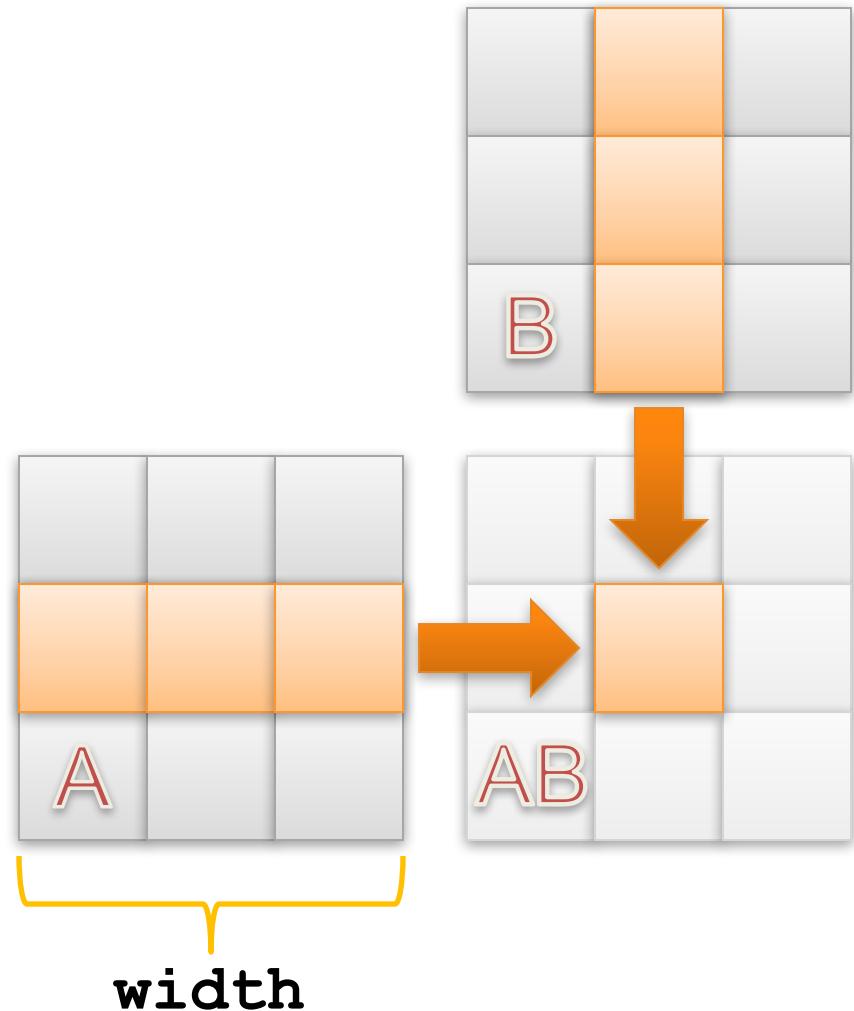
    ab[row*width+col] = result;
}
```

How will this perform?

How many loads per term of dot product?	2 (a & b) = 8 Bytes
How many floating point operations?	2 (multiply & addition)
Global memory access to flop ratio (GMAC)	8 Bytes / 2 ops = 4 B/op
What is the peak fp performance of GeForce GTX 260?	805 GFLOPS
Lower bound on bandwidth required to reach peak fp performance	GMAC * Peak FLOPS = 4 * 805 = 3.2 TB/s
What is the actual memory bandwidth of GeForce GTX 260?	112 GB/s
Then what is an upper bound on performance of our implementation?	Actual BW / GMAC = 112 / 4 = 28 GFLOPS

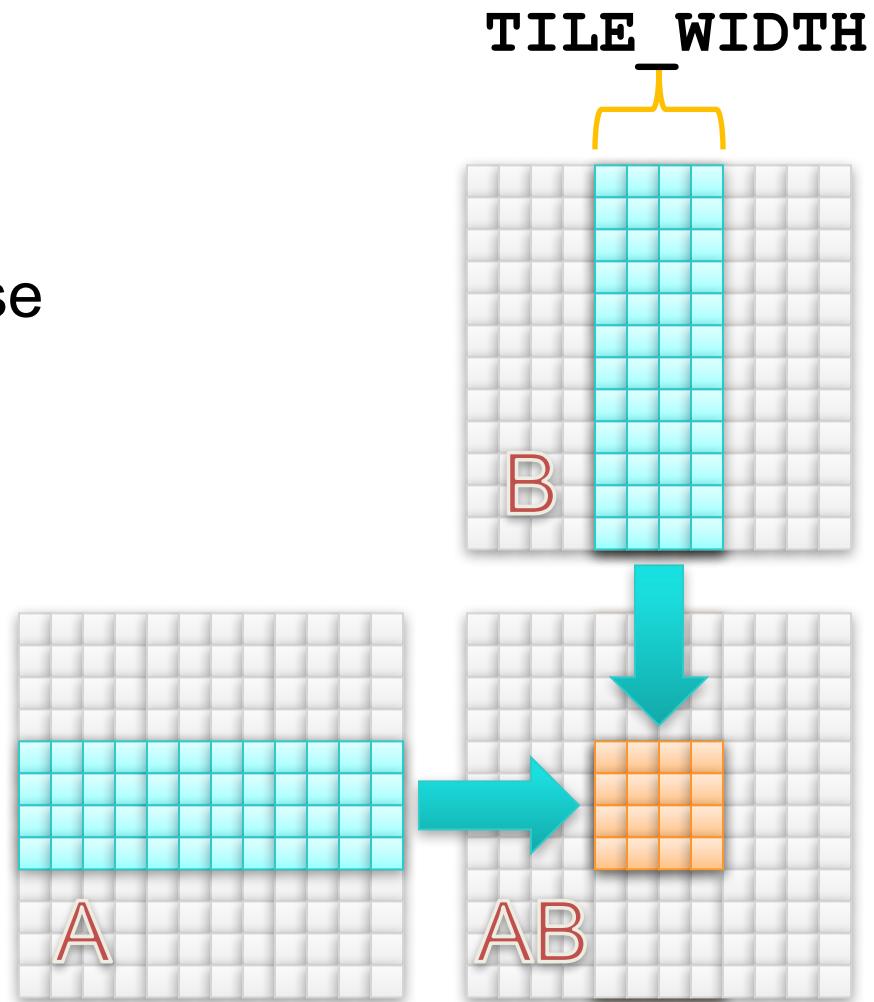
Idea: Use shared memory to reuse global data

- Each input element is read by `width` threads
- Load each element into shared memory and have several threads use the local version to reduce the memory bandwidth



Tiled Multiply

- Partition kernel loop into **phases**
- Load a tile of both matrices into shared each phase
- Each phase, each thread computes a **partial** result



Use of Barriers in mat_mul

- Two barriers per phase:
 - `__syncthreads` after all data is loaded into `__shared__` memory
 - `__syncthreads` after all data is read from `__shared__` memory
 - Note that second `__syncthreads` in phase p guards the load in phase $p+1$
- Use barriers to guard data
 - Guard against using uninitialized data
 - Guard against bashing live data

First Order Size Considerations

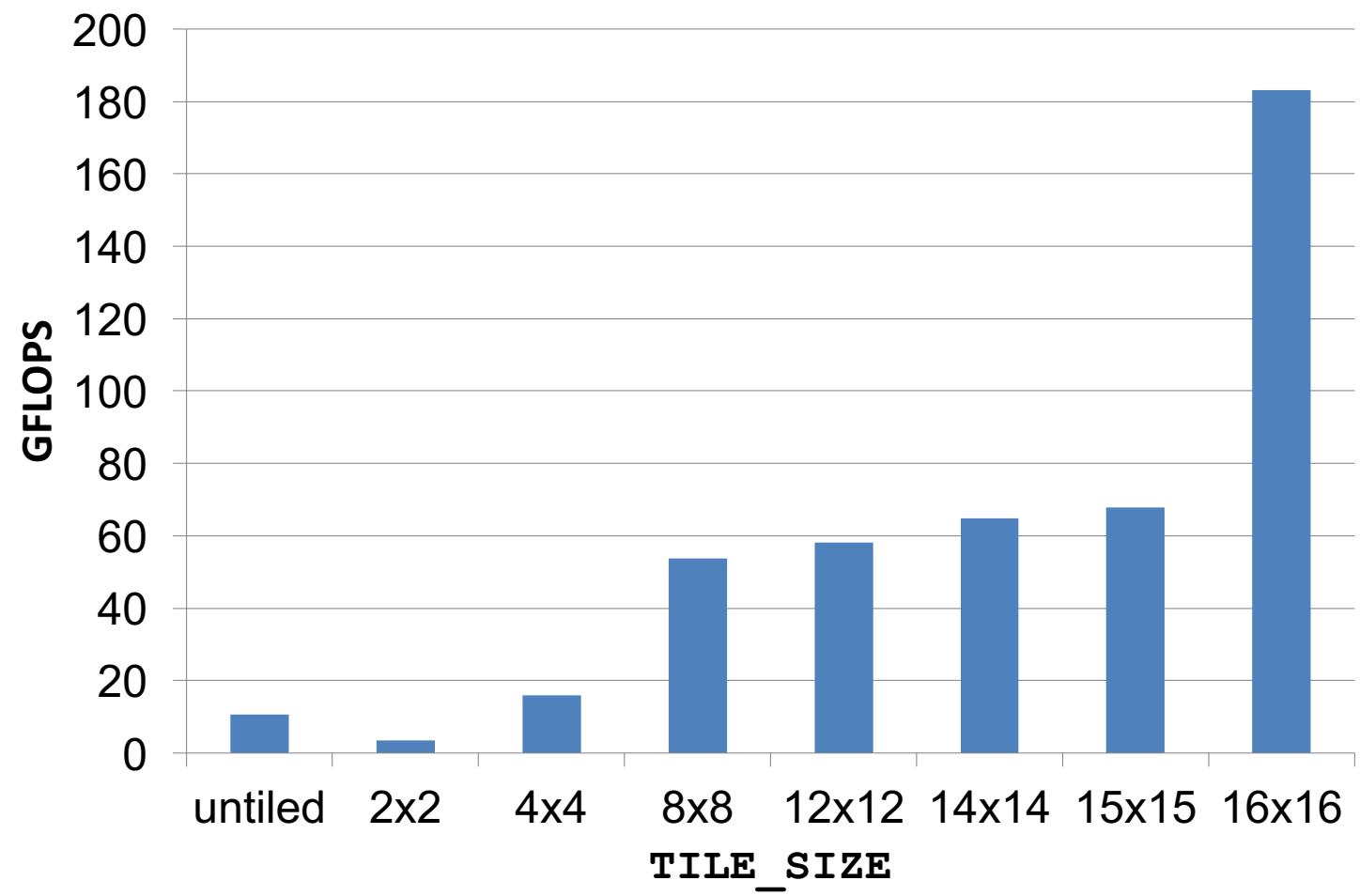
- Each **thread block** should have many threads
 - $\text{TILE_WIDTH} = 16 \rightarrow 16 \times 16 = 256$ threads
- There should be many thread blocks
 - 1024×1024 matrices $\rightarrow 64 \times 64 = 4096$ thread blocks
 - $\text{TILE_WIDTH} = 16 \rightarrow$ gives each SM 3 blocks, 768 threads
 - Full **occupancy**
- Each thread block performs $2 * 256 = 512$ 32b loads for $256 * (2 * 16) = 8,192$ fp ops
 - Memory bandwidth no longer limiting factor

Optimization Analysis

Implementation	Original	Improved
Global Loads	$2N^3$	$2N^2 * (N/TILE_WIDTH)$
Throughput	10.7 GFLOPS	183.9 GFLOPS
SLOCs	20	44
Relative Improvement	1x	17.2x
Improvement/SLOC	1x	7.8x

- Experiment performed on a GT200
- This optimization was clearly worth the effort
- Better performance still possible in theory

TILE_SIZE Effects



Memory Resources as Limit to Parallelism

Resource	Per GT200 SM	Full Occupancy on GT200
Registers	16384	<= 16384 / 768 threads = 21 per thread
<u>__shared__</u> Memory	16KB	<= 16KB / 8 blocks = 2KB per block

- Effective use of different memory resources reduces the number of accesses to global memory
- These resources are **finite!**
- The more memory locations each thread requires → the fewer threads an SM can accommodate

Final Thoughts

- Effective use of CUDA memory hierarchy decreases bandwidth consumption to increase throughput
- Use `__shared__` memory to eliminate redundant loads from global memory
 - Use `__syncthreads` barriers to protect `__shared__` data
 - Use atomics if access patterns are sparse or unpredictable
- Optimization comes with a development cost
- Memory resources ultimately limit parallelism