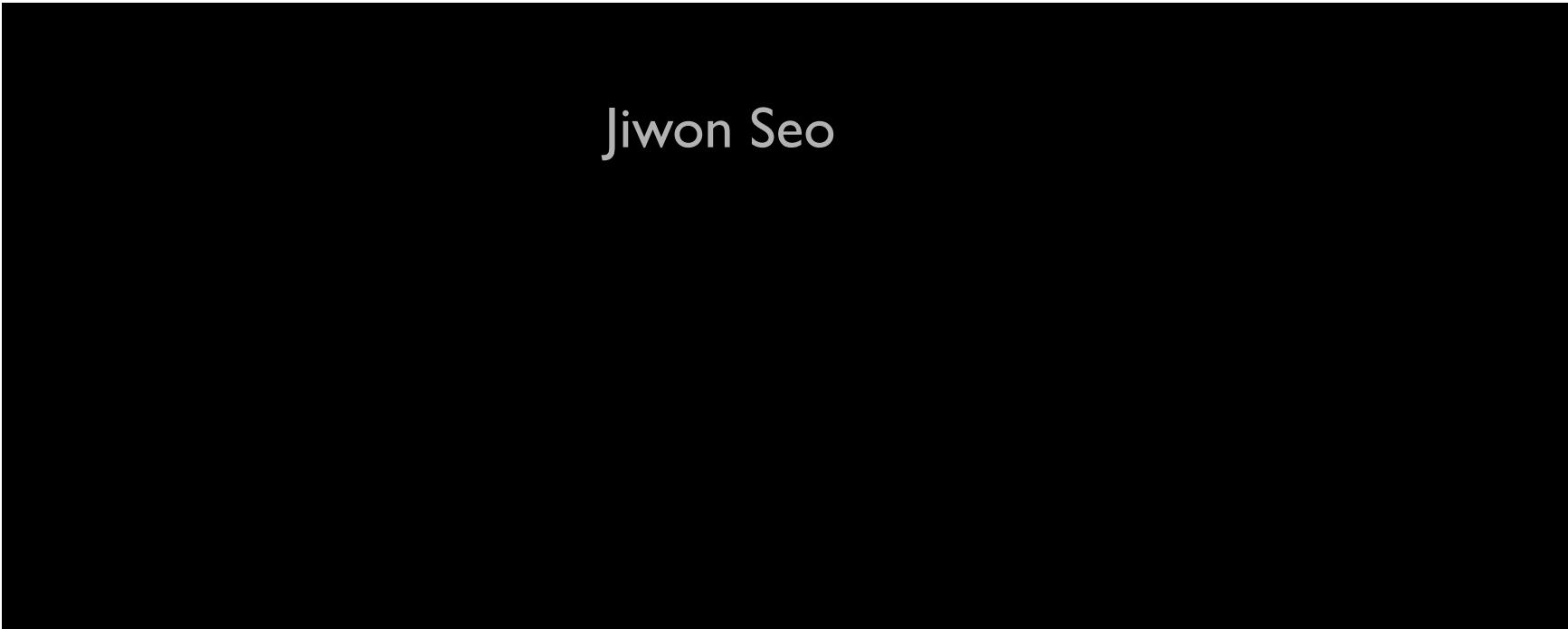




# CUDA, Parallel Programming Language

## CUDA Memories



Jiwon Seo

# Latency & Throughput

- **Latency** is the delay caused by the physical speed of the hardware
- CPU = low latency, low throughput
  - CPU clock = 3 GHz (3 clocks/ns)
  - CPU main memory latency: ~100+ ns
  - CPU arithmetic instruction latency: ~1+ ns
- GPU = high latency, high throughput
  - GPU clock = 1.5 GHz (1.5 clock/ns)
  - GPU main memory latency: ~300+ ns
  - GPU arithmetic instruction latency: ~10+ ns

# Compute & I/O Throughput

## GeForce GTX 1080 TI

Compute throughput	11.3 TFLOPS (single precision)
Global memory bandwidth	484 GB/s (121 Gfloat/s)

- GPU is very IO limited! IO is very often the throughput bottleneck, so its important to be smart about IO.
- If you want to get beyond ~900 GFLOPS, need to do multiple FLOPs per float data load.

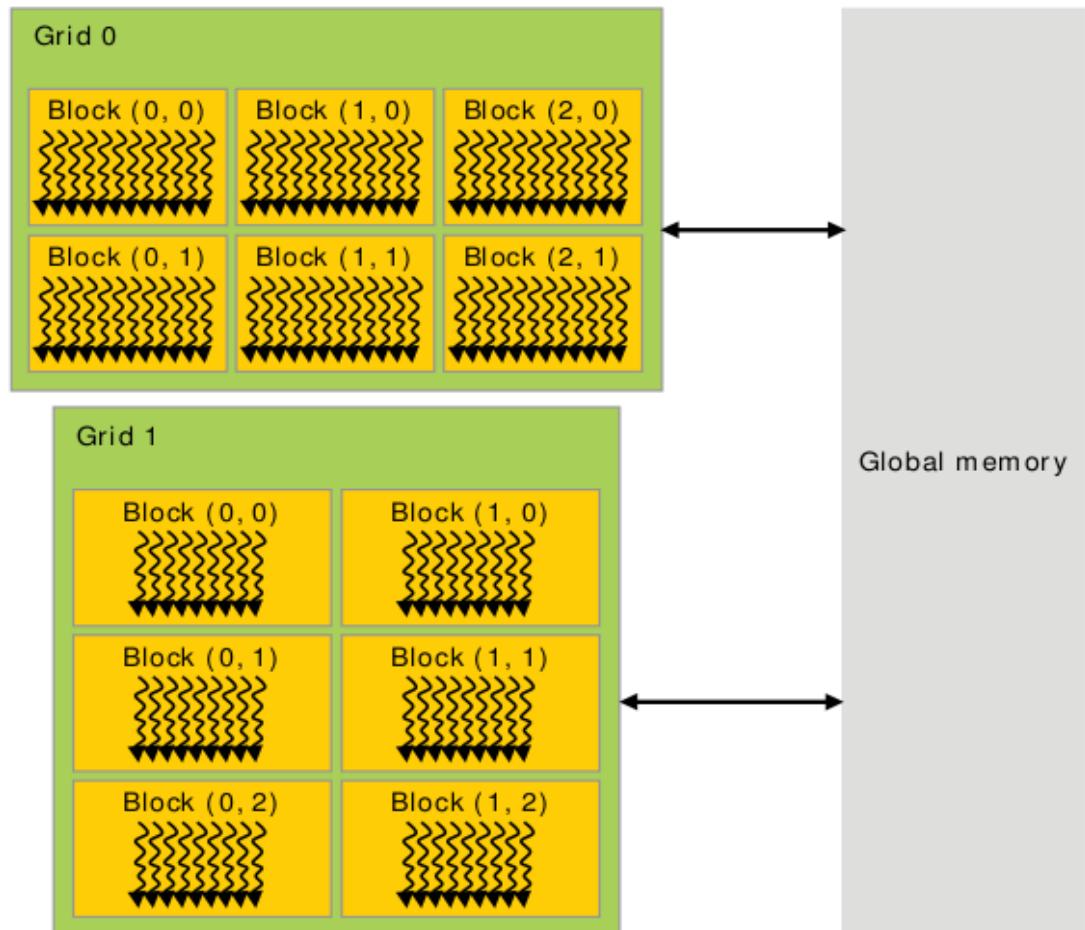
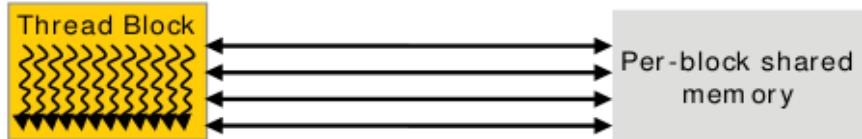
# GPU Memory Breakdown

- Registers
- Local Memory
- Global Memory
- Shared Memory
- L1/L2/L3 Cache
- Constant Memory
- Texture Memory
- Read-only Cache

# GPU Memory Breakdown

- Registers
- Local Memory
- Global Memory
- Shared Memory
- L1/L2/L3 Cache
- Constant Memory
- Texture Memory
- Read-only Cache

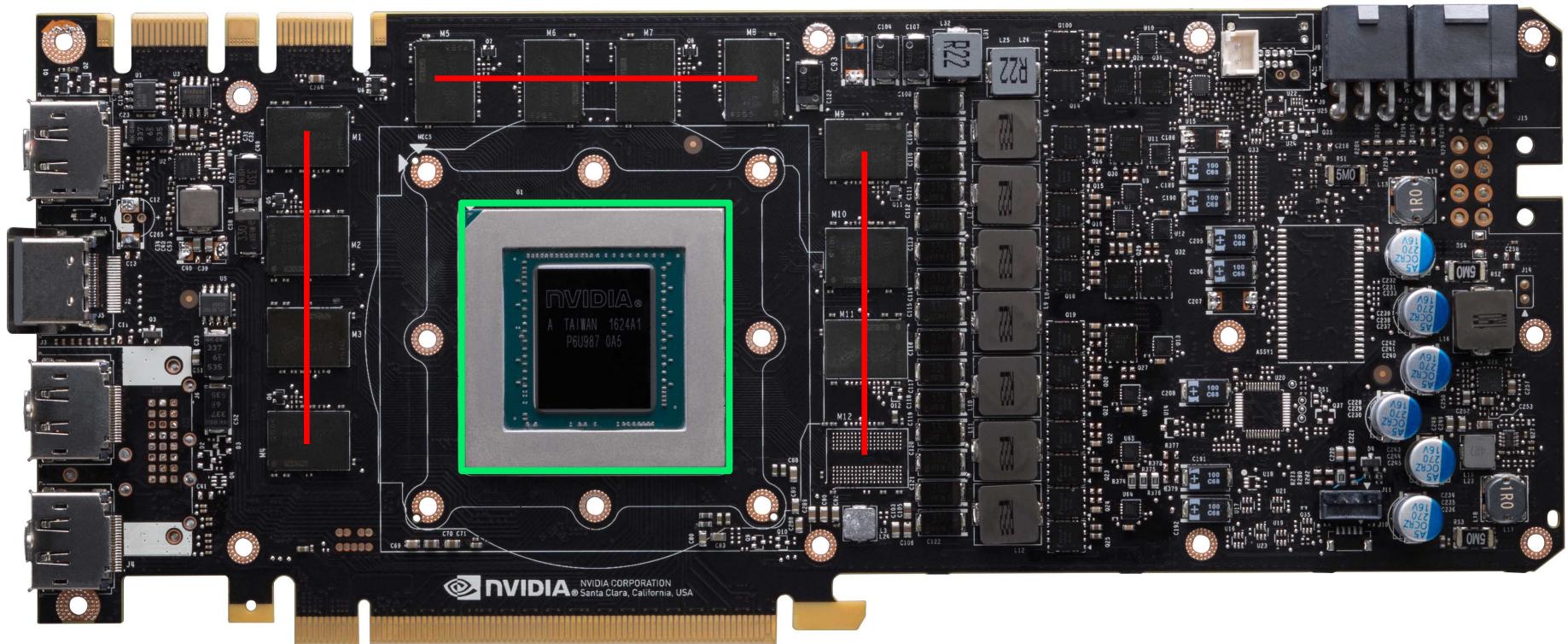
# Memory Scope



# Global Memory

- **Global memory** is separate hardware from GPU SMs
  - The vast majority of memory on a GPU is global memory
  - If data doesn't fit in global memory, process it in chunks that fit
  - GPUs have 1~32GB of global memory, with most having ~10GB
- Global memory latency is ~300ns on Kepler and ~600ns on Fermi

# Global Memory



Green box is 1080 TI, Red lines are global memory

NVIDIA GTX 1080 TI

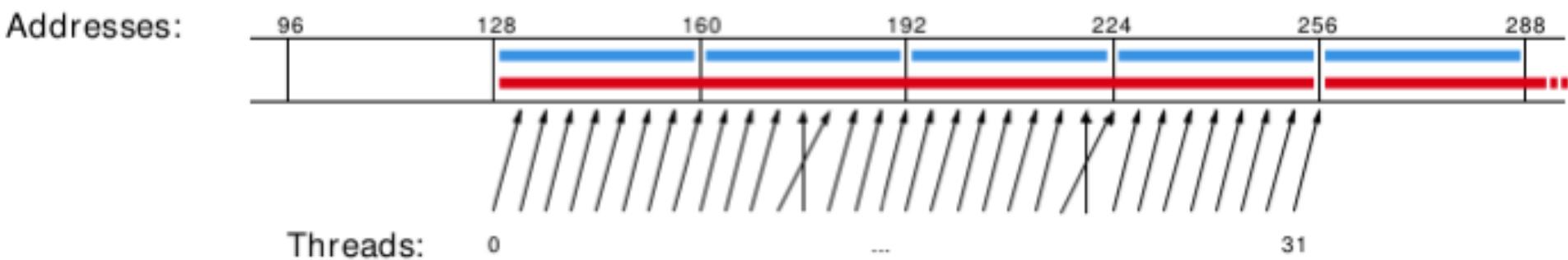
# Accessing Global Memory Efficiently

- Global memory IO is the slowest form of IO on GPU
- Because of this, we want to access global memory as little as possible
- Access patterns that play nicely with GPU hardware are called **coalesced memory accesses**.

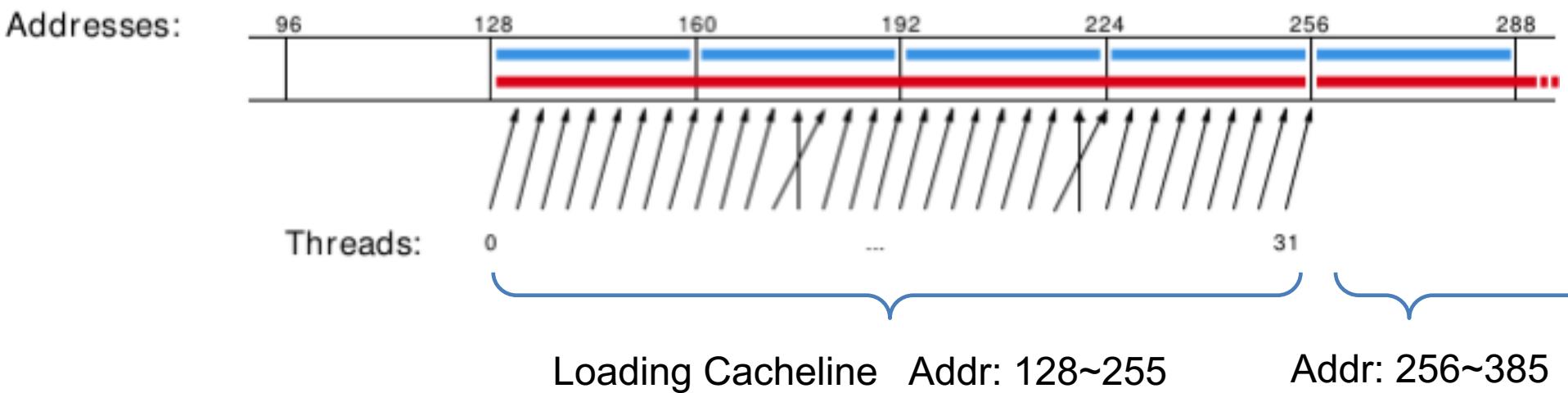
# Memory Coalescing

- Memory accesses done in large groups of **Memory Transactions**
  - Done per warp
  - Fully utilizes the way IO is setup at the hardware level
- Coalesced memory accesses minimize the number of cache lines read
  - GPU cache lines are 128 bytes and are aligned

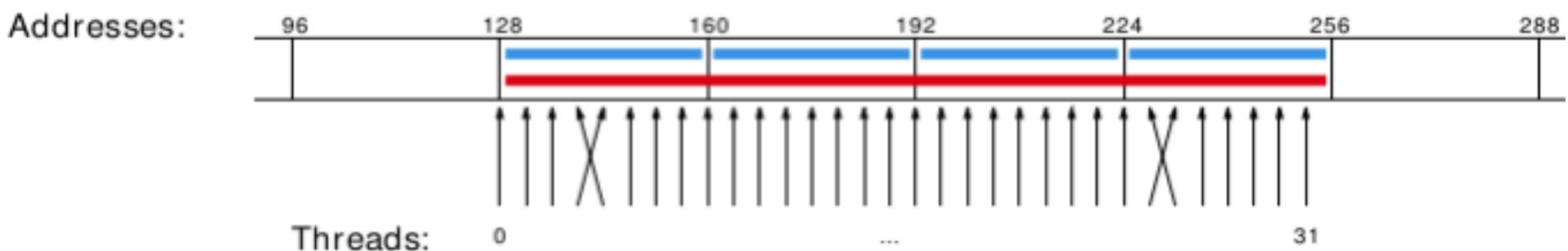
# Mis-aligned Accesses



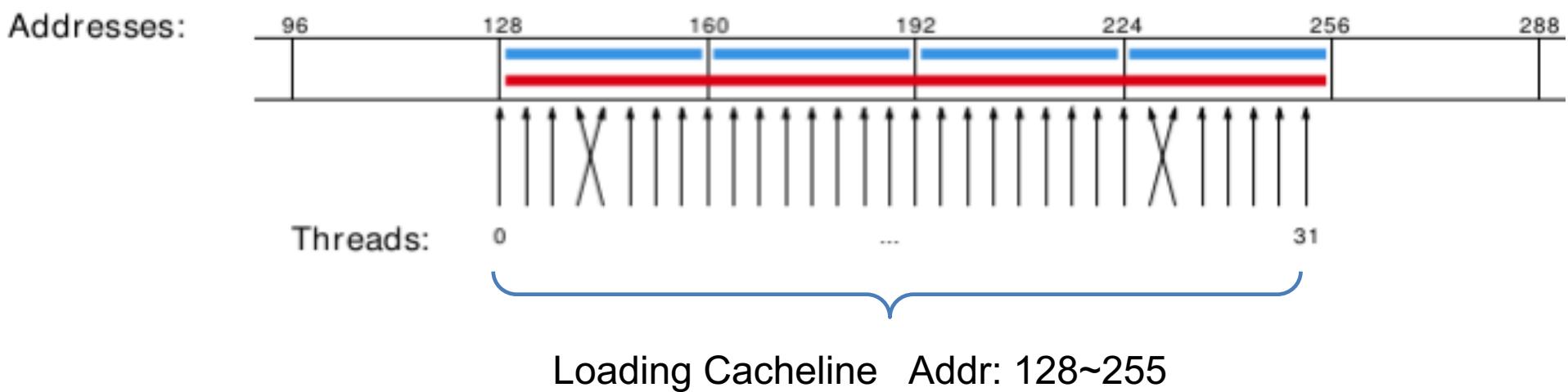
# Mis-aligned Accesses



# Aligned Accesses



# Aligned Accesses



# Shared Memory

- Fast memory located in the SM
- Same hardware as L1 cache      *programable cache*
- ~5ns of latency
- Maximum size of ~48KB (varies per GPU)
- Scope of shared memory == thread block

*thread block 단위로 작업 실행,, 끝나면 free*

# Shared Memory Syntax

- Can allocate shared memory statically or dynamically
- Static allocation:
  - `__shared__ float data[1024];`
  - Declared in the kernel, nothing in host code
- Dynamic allocation:
  - Host:
    - `kernel<<<grid_dim, block_dim, numBytesShMem>>>(args);`
  - Device (in kernel):
    - `extern __shared__ float s[];`

For reference: <https://devblogs.nvidia.com/using-shared-memory-cuda-cc/>

# CUDA Kernel w/ Shared Memory

Task: Compute histogram of color usages

Input: array of bytes of length N

Output: 256 element array of integers

**Naive**: build output in global memory, N global stores

**Smart**: build output in shared memory, copy to global memory at end, 256 global stores

# Histogram (Naïve)

```
// Determine frequency of colors in a picture
// Each thread looks at one pixel and increments
// a counter atomically

__global__ void histogram(int* color,
                         int* buckets) {

    int i = threadIdx.x
        + blockDim.x * blockIdx.x;

    int c = colors[i];
    atomicAdd(&buckets[c], 1);
}
```

# Example: Histogram

```
__global__ void histogram(int* color,
                         int* buckets) {
    __shared__ int hist[256];
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    int c = colors[i];
    atomicAdd(&hist[c], 1);
    __syncthreads();
    if (threadIdx.x < 256) {
        atomicAdd(&buckets[i], hist[i]);
    }
}
```

# Example: Histogram

```
__global __ void histogram(int* color, int* buckets) {
    __shared__ int hist[256];
    __shared__ int _color[blockDim.x];
    int i = threadIdx.x + blockDim.x * blockIdx.x;
    _color[threadIdx.x] = colors[i];
    __syncthreads(); 없어도 됨, 각각의 스레드가 다른 데이터를 fetch 하기때문에
    atomicAdd(&hist[_color[threadIdx.x]], 1);
    __syncthreads();
    if (threadIdx.x < 256) {
        atomicAdd(&buckets[i], hist[i]);
    }
}
```

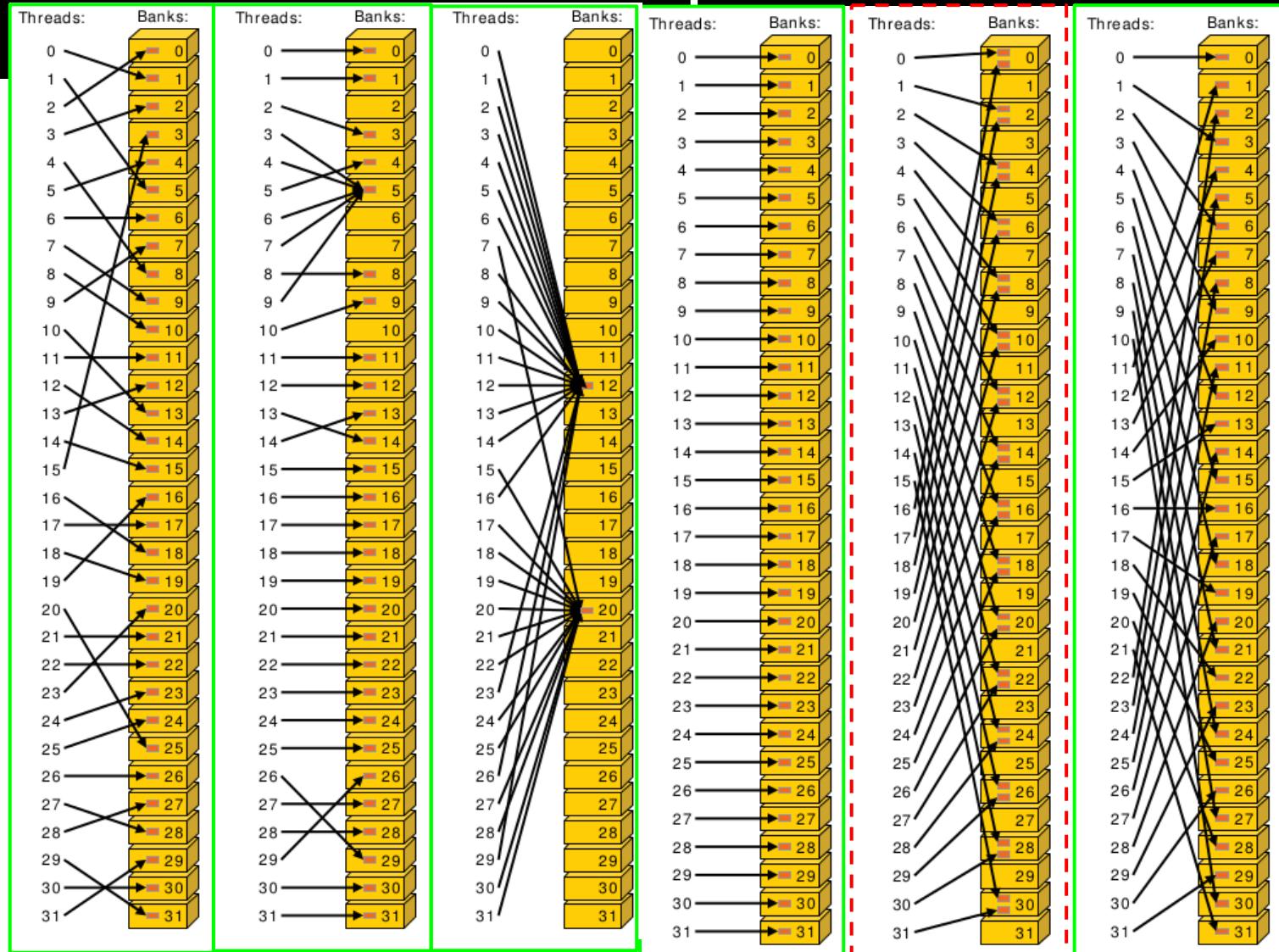
# A Common Pattern in Kernels

- (1) copy from global memory to shared memory
- (2) \_\_syncthreads()
- (3) perform computation, incrementally storing output in shared memory, \_\_syncthreads() as necessary
- (4) copy output from shared memory to output array in global memory

# Bank Conflict for Shared Memory

- Shared memory is setup as 32 **banks**
  - If an element is 4 byte-long,  $\text{element}[i]$  in bank  $i \% 32$
- A **bank conflict** occurs when 2 threads in a warp access different elements in the same bank.
  - Bank conflicts cause serial memory accesses rather than parallel
    - Serial *anything* in GPU programming = bad for performance

# Bank Conflict Example



# Bank Conflict and Strides

Stride 1  $\Rightarrow$  32 x 1-way “bank conflicts” (so conflict-free)

Stride 2  $\Rightarrow$  16 x 2-way bank conflicts

Stride 3  $\Rightarrow$  32 x 1-way “bank conflicts” (so conflict-free)

Stride 4  $\Rightarrow$  8 x 4-way bank conflicts

...

Stride 32  $\Rightarrow$  1 x 32-way bank conflict :(

# Padding to avoid Bank Conflict

To fix the stride 32 case, we'll waste a byte on padding and make the stride 33 :)

Don't store any data in slots 32, 65, 98, ....

Now we have

thread 0  $\Rightarrow$  index 0 (bank 0)

thread 1  $\Rightarrow$  index 33 (bank 1)

thread i  $\Rightarrow$  index  $33 * i$  (bank i)

쓸데없는 데이터 넣어서 처리? 같은 뱅크에 접근하지 않도록

forced sharing?

true sharing?

비슷한 솔루션 제공

# Registers

- Fastest “memory”, ~10x faster than shared memory
- There are tens of thousands of registers in each SM
  - Generally works out to a maximum of 32 or 64 32-bit registers per thread
- Most stack variables declared in kernels are stored in registers
  - example: `float x;` float는 레지스터에 저장, array size가 충분히 작으면 레지스터에 저장, 아니면 스택이나 로컬메모리저장
- Statically indexed arrays stored on the stack are sometimes put in registers

# Local Memory

- **Local memory** is everything on the stack that can't fit in registers
- The scope of local memory is just the thread.
- Local memory is stored in global memory
  - much slower than registers

# CUDA Variable Type Qualifiers

Variable declaration	Memory	Scope	Lifetime
<code>int var;</code>	register	thread	thread
<code>int array_var[10];</code>	local	thread	thread
<code>__shared__ int shared_var;</code>	shared	block	block
<code>__device__ int global_var;</code>	global	grid	application
<code>__constant__ int constant_var;</code>	constant	grid	application

- “automatic” scalar variables without qualifier reside in a register
  - compiler will spill to thread local memory
- “automatic” array variables without qualifier reside in thread-local memory

# CUDA Variable Type Performance

Variable declaration	Memory	Penalty
<code>int var;</code>	register	1x
<code>int array_var[10];</code>	local	100x
<code>__shared__ int shared_var;</code>	shared	1x
<code>__device__ int global_var;</code>	global	100x
<code>__constant__ int constant_var;</code>	constant	1x

- scalar variables reside in fast, on-chip registers
- shared variables reside in fast, on-chip memories
- thread-local arrays & global variables reside in uncached off-chip memory
- constant variables reside in cached off-chip memory

# CUDA Variable Type Scale

Variable declaration	Instances	Visibility
<code>int var;</code>	100,000s	1
<code>int array_var[10];</code>	100,000s	1
<code>__shared__ int shared_var;</code>	100s	100s
<code>__device__ int global_var;</code>	1	100,000s
<code>__constant__ int constant_var;</code>	1	100,000s

- 100Ks per-thread variables, R/W by 1 thread
- 100s shared variables, each R/W by 100s of threads
- 1 global variable is R/W by 100Ks threads
- 1 constant variable is readable by 100Ks threads

# GPU Memory Breakdown

- Registers
- Local Memory
- Global Memory
- Shared Memory
- L1/L2/L3 Cache
- Constant Memory
- Texture Memory
- Read-only Cache

# L1 Cache

- Caches local and/or global memory
- Same hardware as shared memory
- For some GPU architectures, configurable (16, 32, 48KB)
- Each SM has its own L1 cacheSM안에 있어서 크기 한계 존재

# L2 Cache

- Caches all global & local memory accesses
- ~1MB in size
- Shared by all SM's

# Constant Memory

Constant memory is global memory with a special cache

- Used for constants that cannot be compiled into program
- Constants must be set from host before running kernel.

~64KB for user, ~64KB for compiler

- kernel arguments are passed through constant memory

# Constant Cache

- 8KB cache on each SM
- Special instruction (LDU, load uniform)
- Go to [http://www.nvidia.com/object/sc10\\_cuda\\_tutorial.html](http://www.nvidia.com/object/sc10_cuda_tutorial.html) for more details

# Constant Memory Syntax

- In global scope (outside of kernel, at top level of program):
  - `__constant__ int foo[1024];`
- In host code:
  - `cudaMemcpyToSymbol(foo, h_src, sizeof(int) * 1024);`

# Texture Memory

Complicated and not very useful for general purpose computation

Useful characteristics:

- 2D or 3D data locality for caching purposes through “CUDA arrays”. Goes into special texture cache.
- fast interpolation on 1D, 2D, or 3D array
- converting integers to “unitized” floating point numbers

# Read-Only Cache

- Many CUDA programs don't use textures, but we should take advantage of the texture cache hardware
- CC (compute capability)  $\geq 3.5$  makes it easier to use texture cache
  - Many `const` variables will automatically load through texture cache
  - Can also force loading through cache with `__ldg` intrinsic function
- Differs from constant memory because doesn't require static indexing

# Example – shared variables

```
// Adjacent Difference application:  
// compute result[i] = input[i] - input[i-1]  
__global__ void adj_diff_naive(int *result, int *input) {  
    // compute this thread's global index  
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if(i > 0)  
    {  
        // each thread loads two elements from global memory  
        int x_i = input[i];  
        int x_i_minus_one = input[i-1];  
  
        result[i] = x_i - x_i_minus_one;  
    }  
}
```

# Example – shared variables

```
// Adjacent Difference application:  
// compute result[i] = input[i] - input[i-1]  
__global__ void adj_diff_naive(int *result, int *input) {  
    // compute this thread's global index  
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;  
  
    if(i > 0)  {  
        // what are the bandwidth requirements of this kernel?  
        int x_i = input[i];  
        int x_i_minus_one = input[i-1];  
          
        result[i] = x_i - x_i_minus_one;  
    }  
}
```

Two loads

# Example – shared variables

```
// Adjacent Difference application:  
// compute result[i] = input[i] - input[i-1]  
__global__ void adj_diff_naive(int *result, int *input) {  
    // compute this thread's global index  
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if(i > 0)  {  
        // How many times does this kernel load input[i]?  
        int x_i = input[i]; // once by thread i  
        int x_i_minus_one = input[i-1]; // again by thread i+1  
  
        result[i] = x_i - x_i_minus_one;  
    }  
}
```

# Example – shared variables

```
// Adjacent Difference application:  
// compute result[i] = input[i] - input[i-1]  
__global__ void adj_diff_naive(int *result, int *input) {  
    // compute this thread's global index  
    unsigned int i = blockDim.x * blockIdx.x + threadIdx.x;  
    if(i > 0) {  
        // Idea: eliminate redundancy by sharing data  
        int x_i = input[i];  
        int x_i_minus_one = input[i-1];  
  
        result[i] = x_i - x_i_minus_one;  
    }  
}
```

# Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input) {
    int tx = threadIdx.x;
    // allocate a __shared__ array, one element per thread
    __shared__ int s_data[BLOCK_SIZE];
    // each thread reads one element to s_data
    unsigned int i = blockDim.x * blockIdx.x + tx;
    s_data[tx] = input[i];

    // avoid race condition: ensure all loads
    // complete before continuing
    __syncthreads();
    ...
}
```

# Example – shared variables

```
// optimized version of adjacent difference
__global__ void adj_diff(int *result, int *input) {
    ...
    if(tx > 0)
        result[i] = s_data[tx] - s_data[tx-1];
    else if(i > 0)
    {
        // handle thread block boundary
        result[i] = s_data[tx] - input[i-1];
    }
}
```

# Optimization Analysis

Implementation	Original	Improved
Global Loads	$2N$	$N + N/\text{BLOCK\_SIZE}$
Global Stores	$N$	$N$
Throughput	36.8 GB/s	57.5 GB/s
SLOCs	18	35
Relative Improvement	1x	1.57x
Improvement/SLOC	1x	0.81x

- Experiment performed on a GT200 chip
  - Improvement likely better on an older architecture
  - Improvement likely worse on a newer architecture
- Optimizations tend to come with a development cost

# About Pointers

- You can point at any memory space per se:

```
__device__ int my_global_variable;
__constant__ int my_constant_variable = 13;

__global__ void foo(void) {
    __shared__ int my_shared_variable;

    int *ptr_to_global = &my_global_variable;
    const int *ptr_to_constant = &my_constant_variable;
    int *ptr_to_shared = &my_shared_variable;
    ...
    *ptr_to_global = *ptr_to_shared;
}
```

# About Pointers

- Pointers aren't typed on memory space
  - `__shared__ int *ptr;`
  - Where does `ptr` point?
  - `ptr` is a `__shared__` pointer variable, not a pointer to a `__shared__` variable!

# Don't confuse the compiler!

```
__device__ int my_global_variable;
__global__ void foo(int *input)
{
    __shared__ int my_shared_variable;

    int *ptr = 0;
    if(input[threadIdx.x] % 2)
        ptr = &my_global_variable;
    else
        ptr = &my_shared_variable;
    // where does ptr point?
}
```

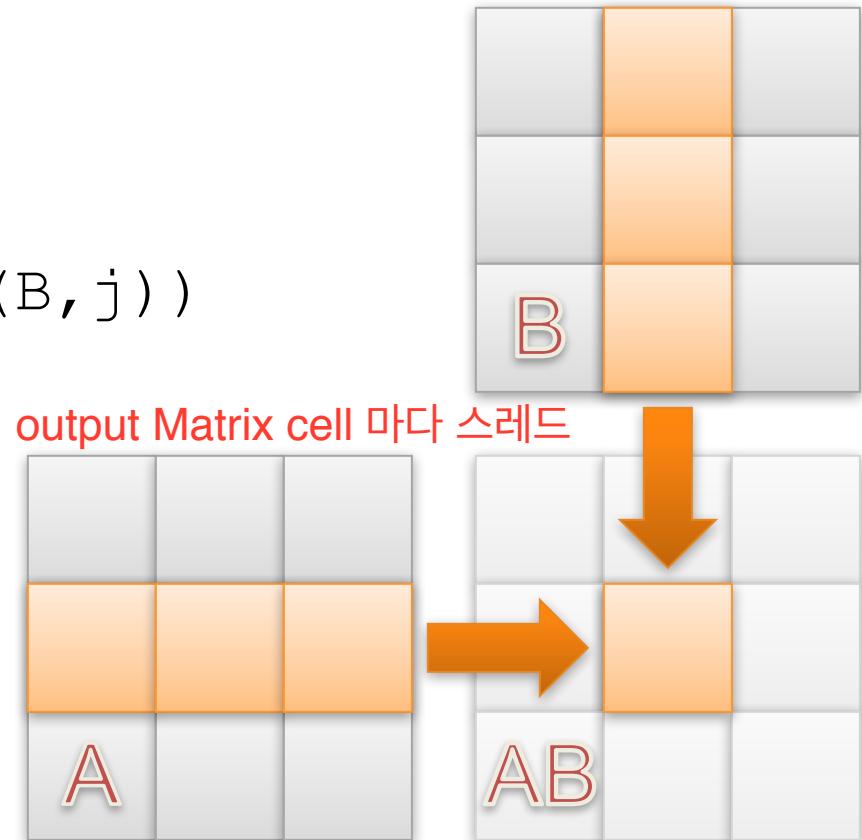
성능도 느리고 의미도 이상함

# Advice

- Prefer dereferencing pointers in simple, regular access patterns
- Avoid propagating pointers
- Avoid pointers to pointers
  - The GPU would rather not pointer chase
  - Linked lists will not perform well
- Pay attention to compiler warning messages
  - Warning: Cannot tell what pointer points to, assuming global memory space
  - Crash waiting to happen

# 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;
}
```

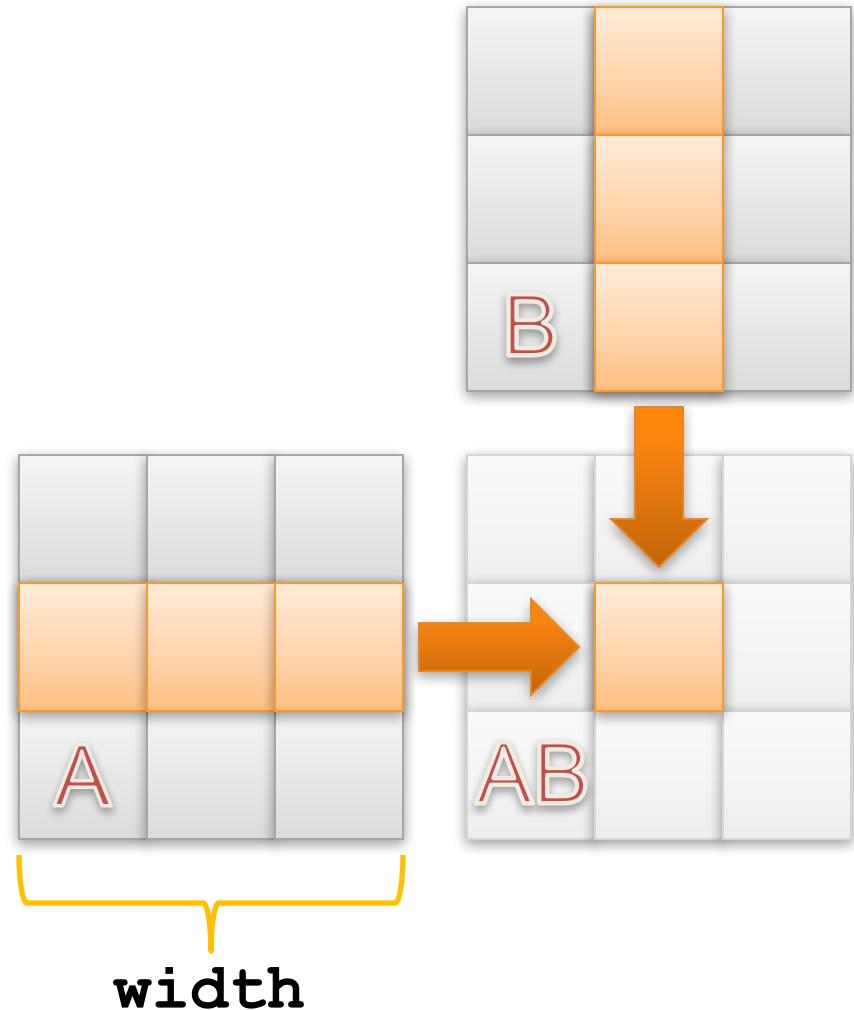
# How will this perform?

each dot product

How many loads per term of dot product?	2 (a & b) = 8 Bytes      cell 2개
How many floating point operations?	2 (multiply & addition) $a[i,j] * b[j,i]$ , sum + result
Global memory access to flop ratio (GMAC)	8 Bytes / 2 ops = 4 B/op      가져오고, 할당?
What is the peak fp performance of GeForce GTX 260?	805 GFLOPS throughput, 모든 연산 유닛을 사용했을때 최대치
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

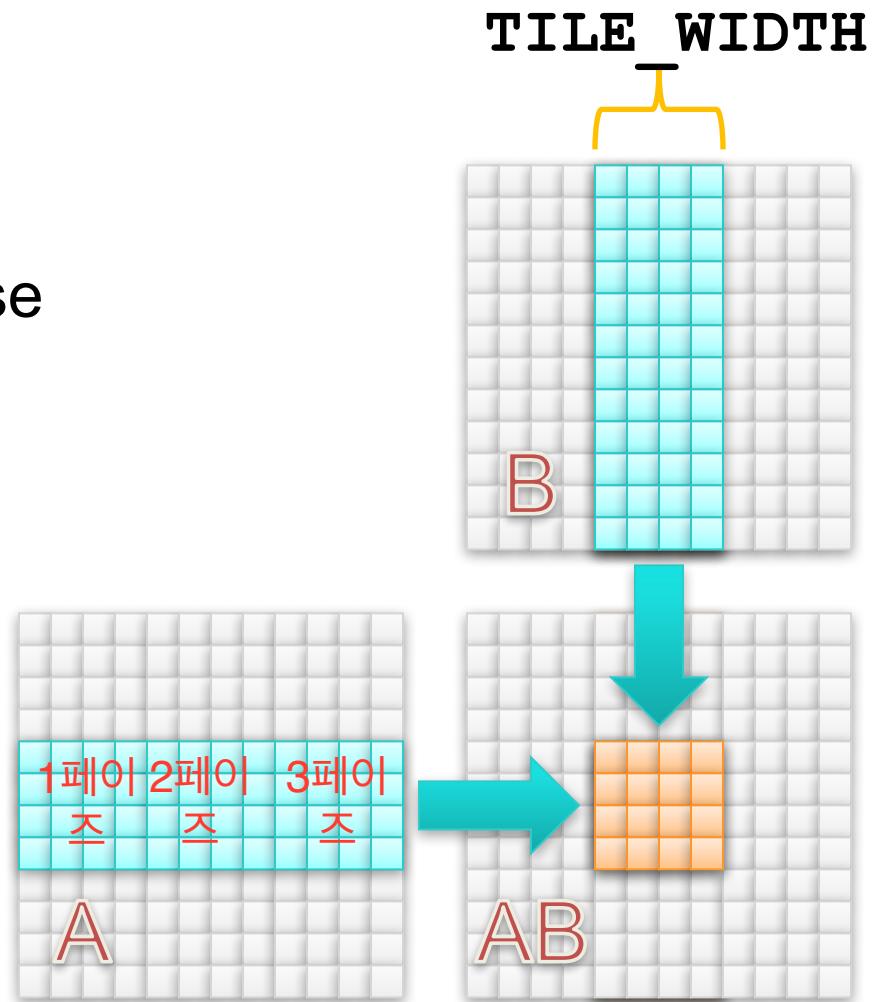
# Idea: Use shared memory to reuse global data

- 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` shared memory -> register memory , 최적화 가능
  - `__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

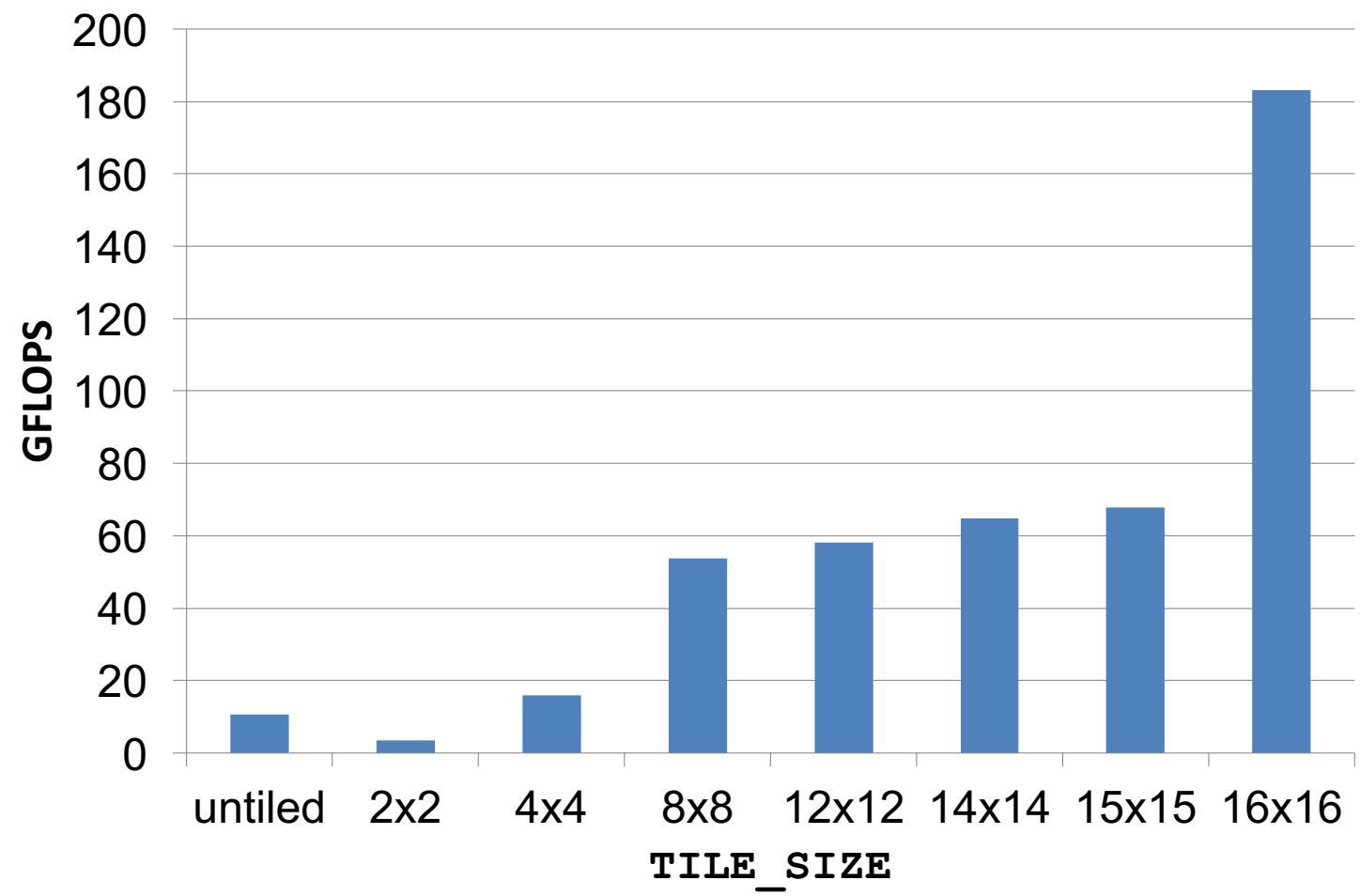
- Each **thread block** should have many threads
  - $\text{TILE\_WIDTH} = 16 \rightarrow 16 \times 16 = 256$  threads  
*same tile -> same block*
- There should be many thread blocks
  - $1024 \times 1024$  matrices  $\rightarrow 64 \times 64 = 4096$  thread blocks
  - $\text{TILE\_WIDTH} = 16 \rightarrow$  gives each SM 3 blocks, 768 threads
  - Full **occupancy**  
*fetch two data( A B),*
- Each thread block performs  $2 * 256 = 512$  32b loads for  $256 * (2 * 16) = 8,192$  fp ops
  - Memory bandwidth no longer limiting factor  
*레지스터로 가져오면 전송폭 문제는 병목구간 x*

# 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