VSCSE summer school - short course

Introduction to CUDA

Lecture 6
Practical Performance Tuning

Joshua A. Anderson

Objective

- Putting the CUDA performance knowledge to work
  – Plausible strategies may or may not lead to performance enhancement
  – Different constraints dominate in different application situations
  – Case studies help to establish intuition, idioms and ideas
- Algorithm patterns that can result in both better efficiency as well as better HW utilization

This lecture covers useful strategies for tuning CUDA application performance on many-core processors.

How thread blocks are partitioned

- Thread blocks are partitioned into warps
  - Thread IDs within a warp are consecutive and increasing
  - Warp 0 starts with Thread ID 0

- Partitioning is always the same
  - Thus you can use this knowledge in control flow
  - However, the exact size of warps may change from generation to generation
  - (Covered next)

- **However, DO NOT rely on any ordering between warps**
  - If there are any dependencies between threads, you must __syncthreads() to get correct results

Control Flow Instructions

- Main performance concern with branching is divergence
  - Threads within a single warp take different paths
  - Different execution paths are serialized in G80
    - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- A common case: avoid divergence when branch condition is a function of thread ID
  - Example with divergence:
    - `If (threadIdx.x > 2) { }`
    - This creates two different control paths for threads in a block
    - Branch granularity < warp size; threads 0, 1 and 2 follow different path than the rest of the threads in the first warp
  - Example without divergence:
    - `If (threadIdx.x / WARP_SIZE > 2) { }`
    - Also creates two different control paths for threads in a block
    - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path

Parallel Reduction

- Given an array of values, “reduce” them to a single value in parallel
- Examples
  - sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array
- Typically parallel implementation:
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

A Vector Reduction Example

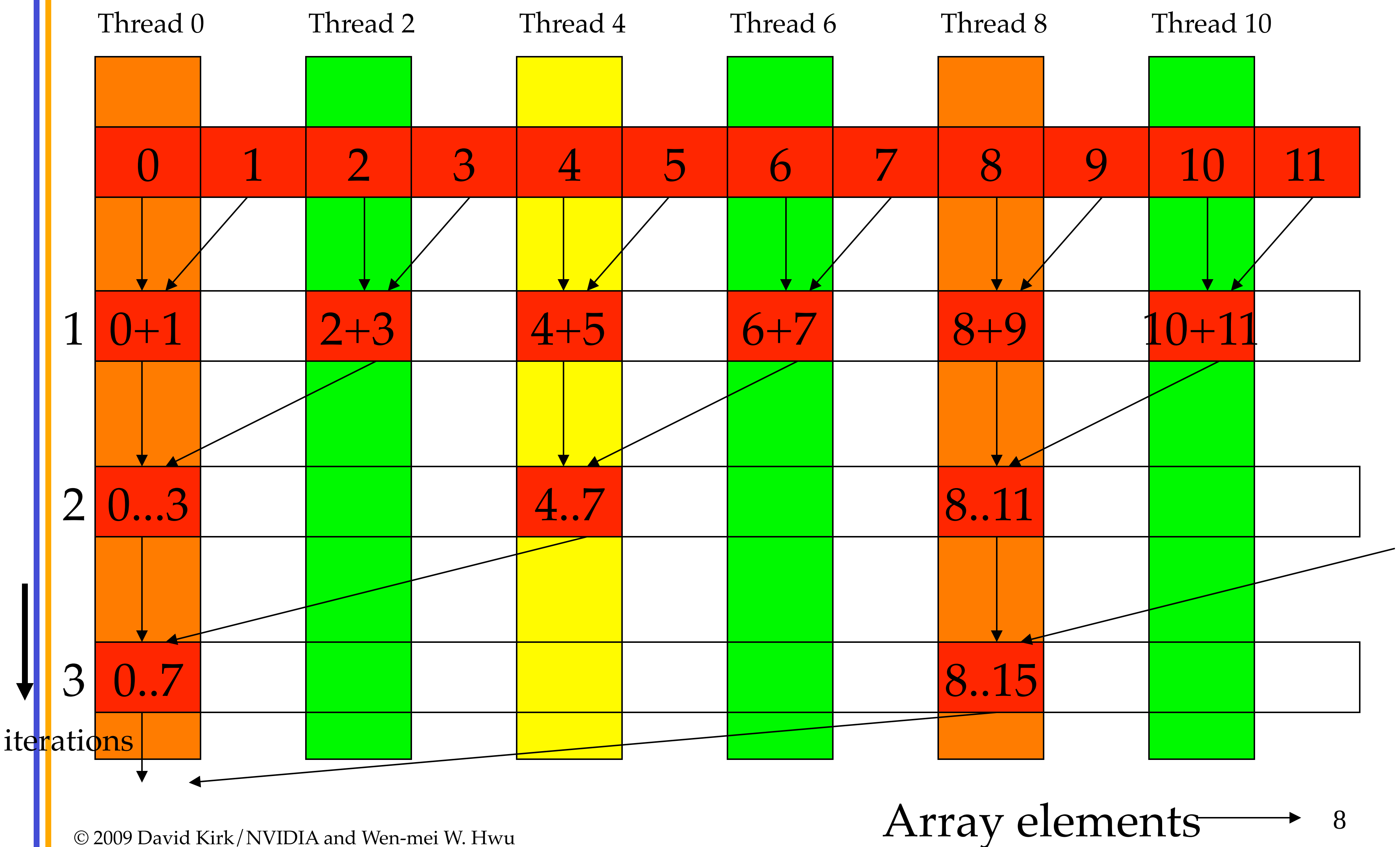
- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

A simple implementation

- Assume we have already loaded array into
  `__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];
}
```

Vector Reduction with Branch Divergence



Some Observations

- In each iterations, two control flow paths will be sequentially traversed for each warp
  - Threads that perform addition and threads that do not
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence

- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5^{th} iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations ($512/32=16= 2^4$), where each iteration only has one thread activated until all warps retire

Shortcomings of the implementation

- Assume we have already loaded array into

```
__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];
}
```

BAD: Divergence due to interleaved branch decisions

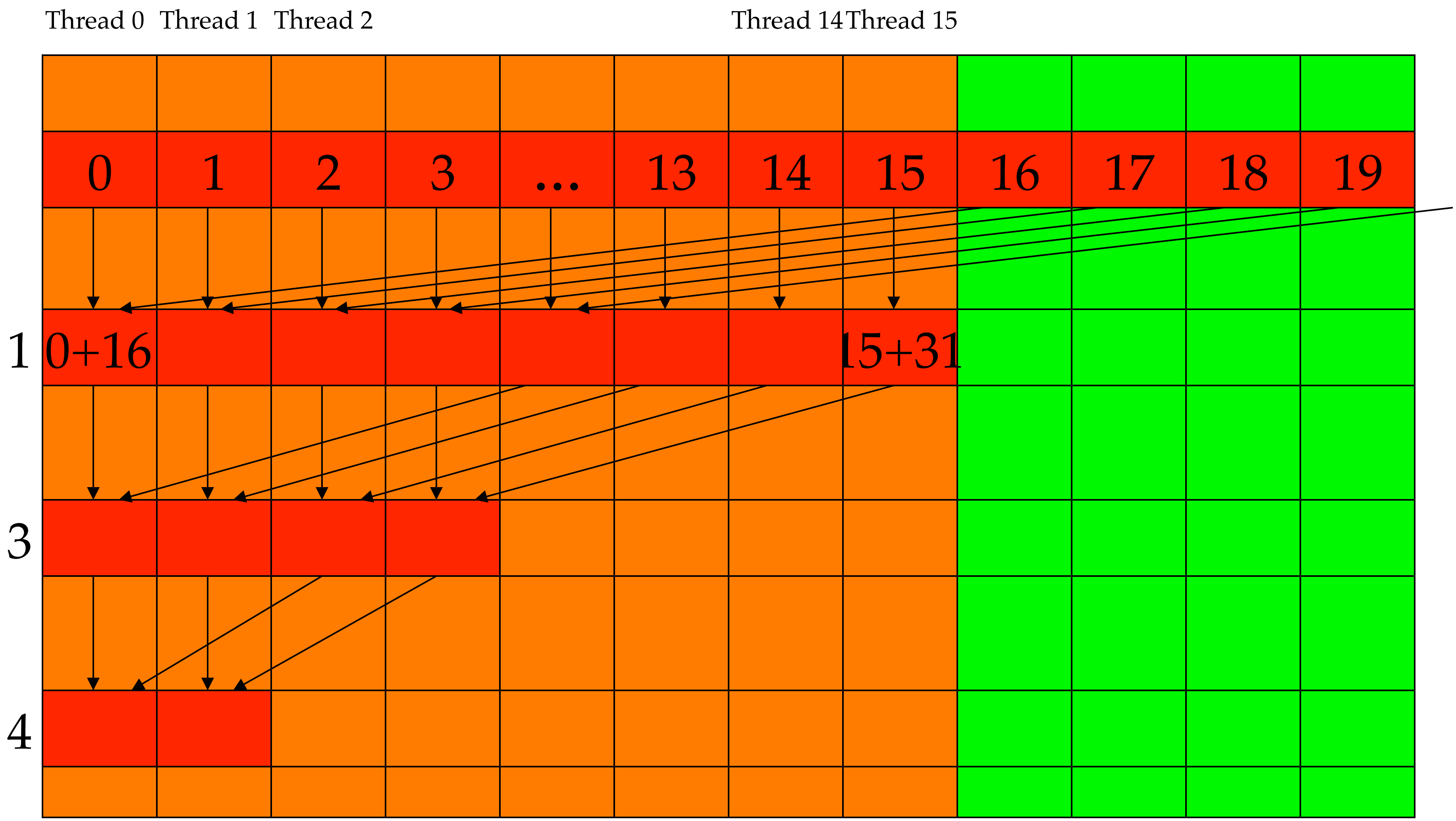
A better implementation

- Assume we have already loaded array into

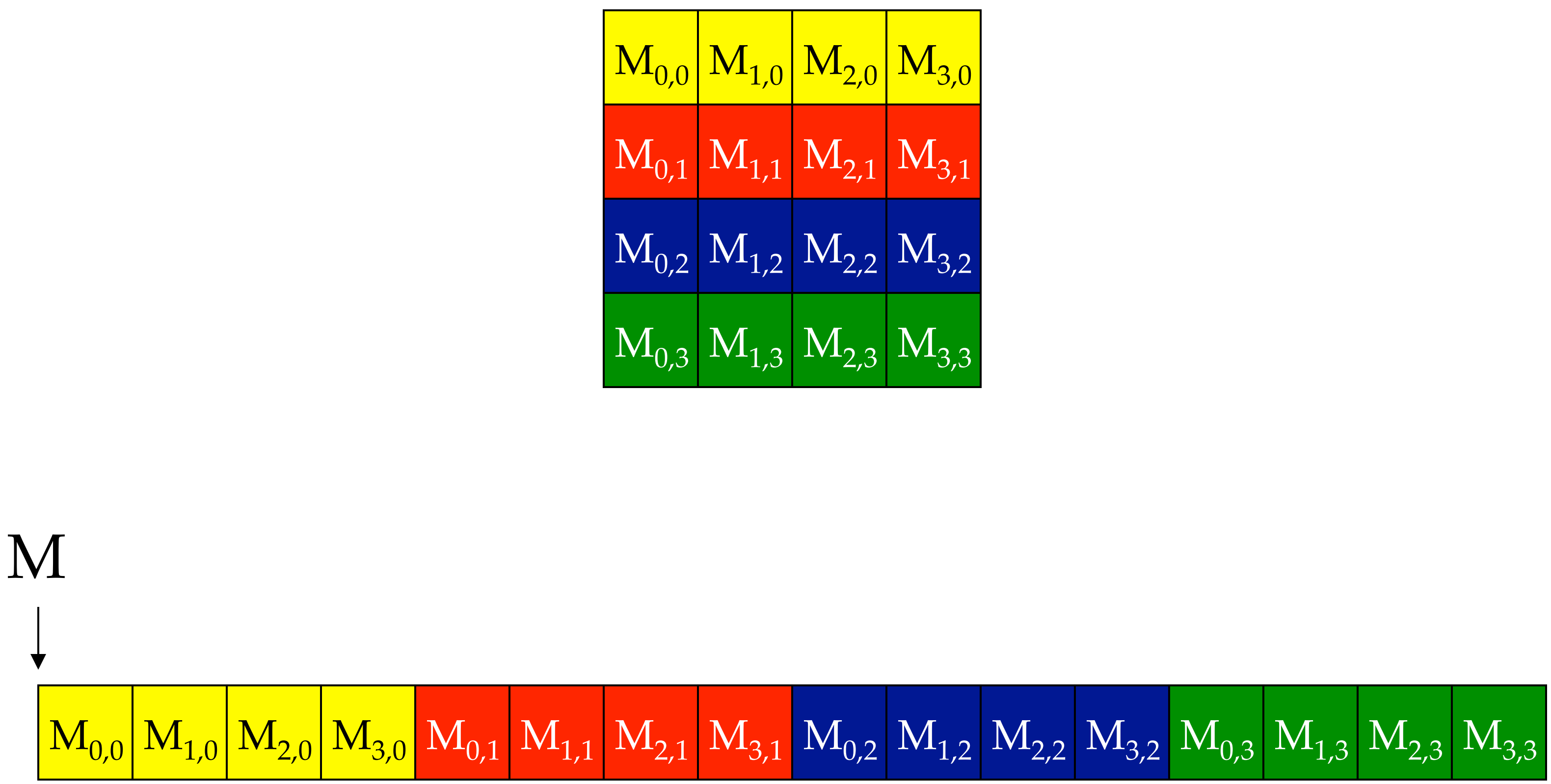
```
__shared__ float partialSum[]

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

No Divergence until < 16 sub-sums



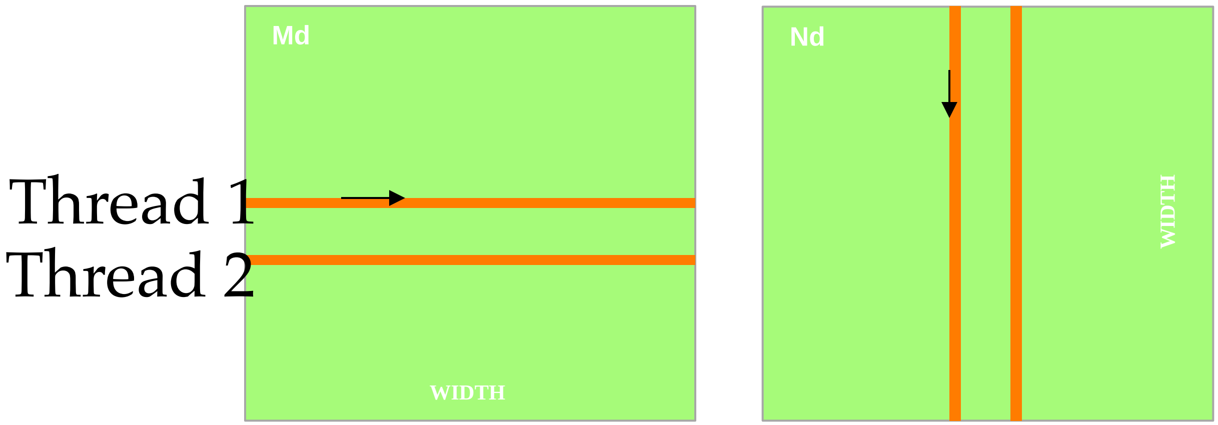
Memory Layout of a Matrix in C



Memory Coalescing

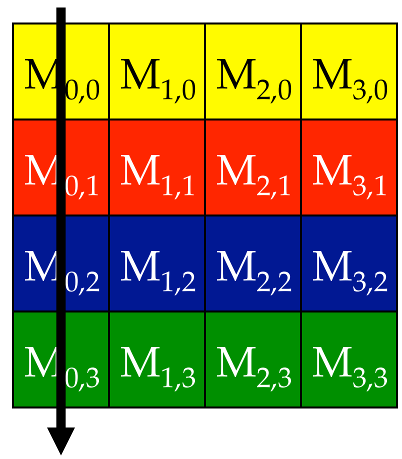
- When accessing global memory, peak performance utilization occurs when all threads in a Warp access continuous memory locations.

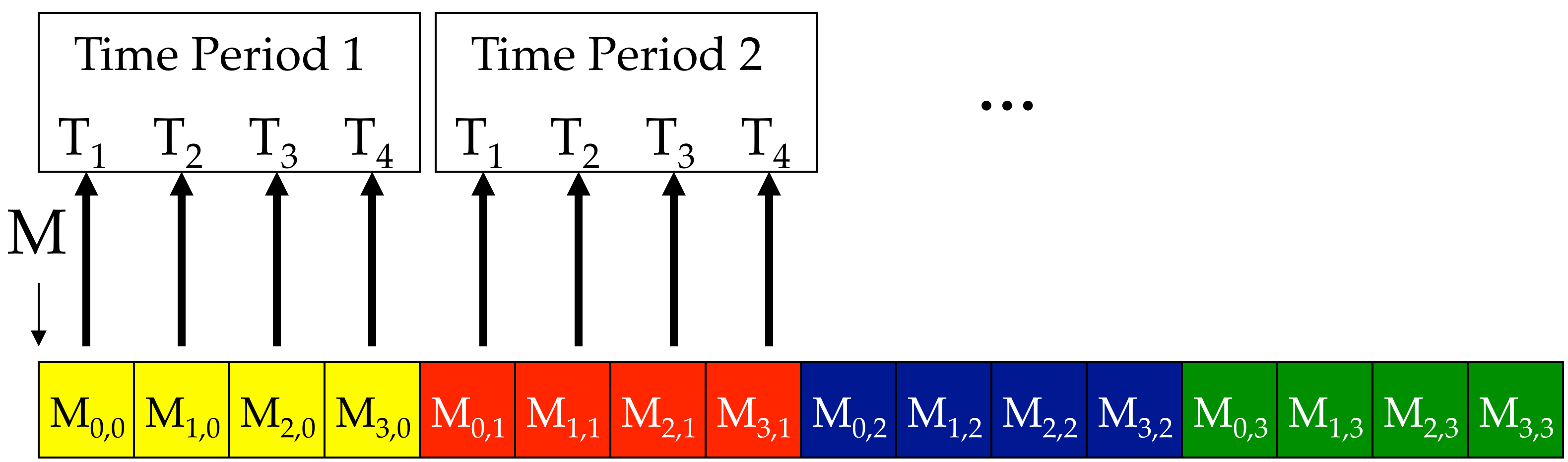
Not coalesced coalesced



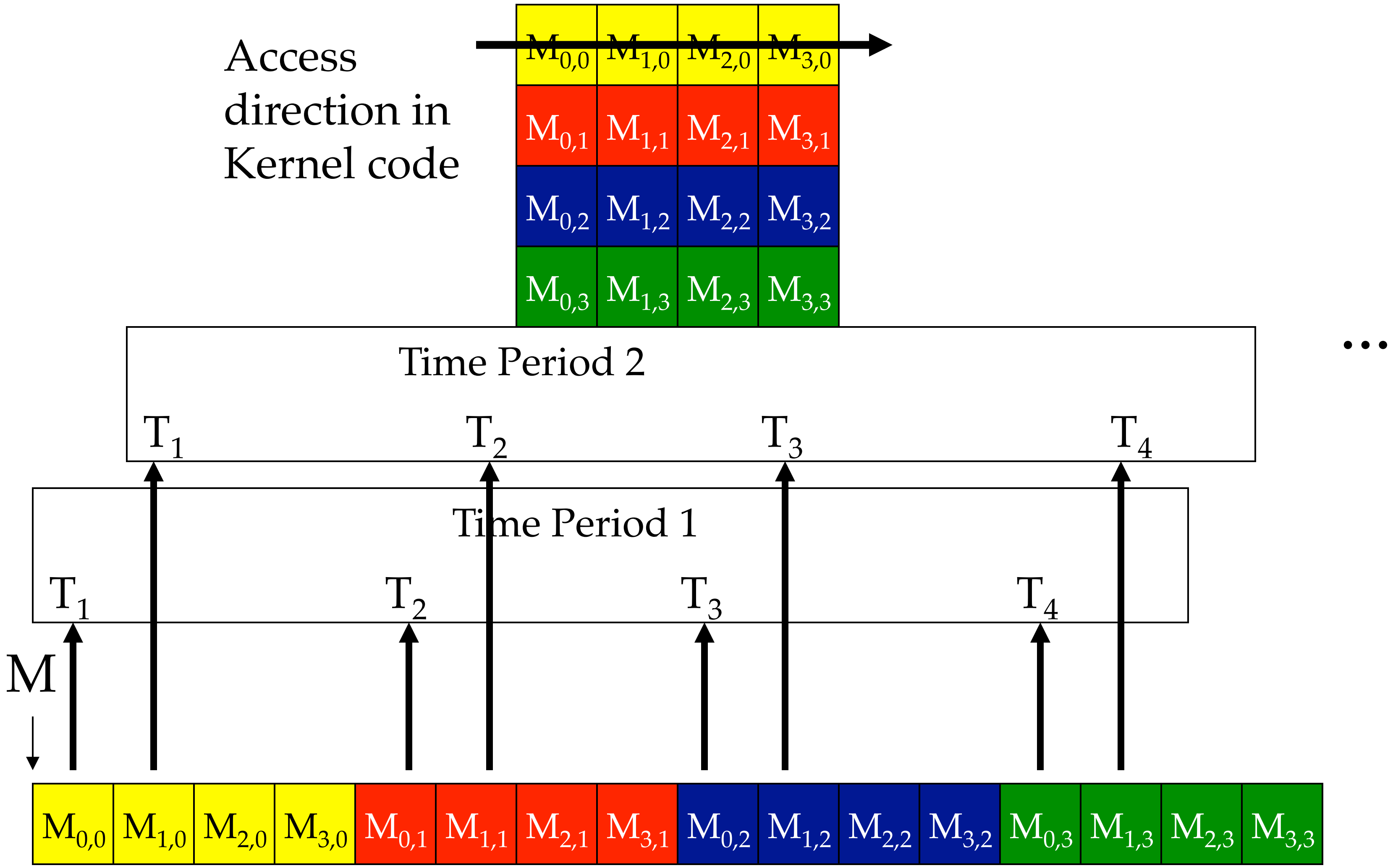
Memory Layout of a Matrix in C

Access direction in Kernel code

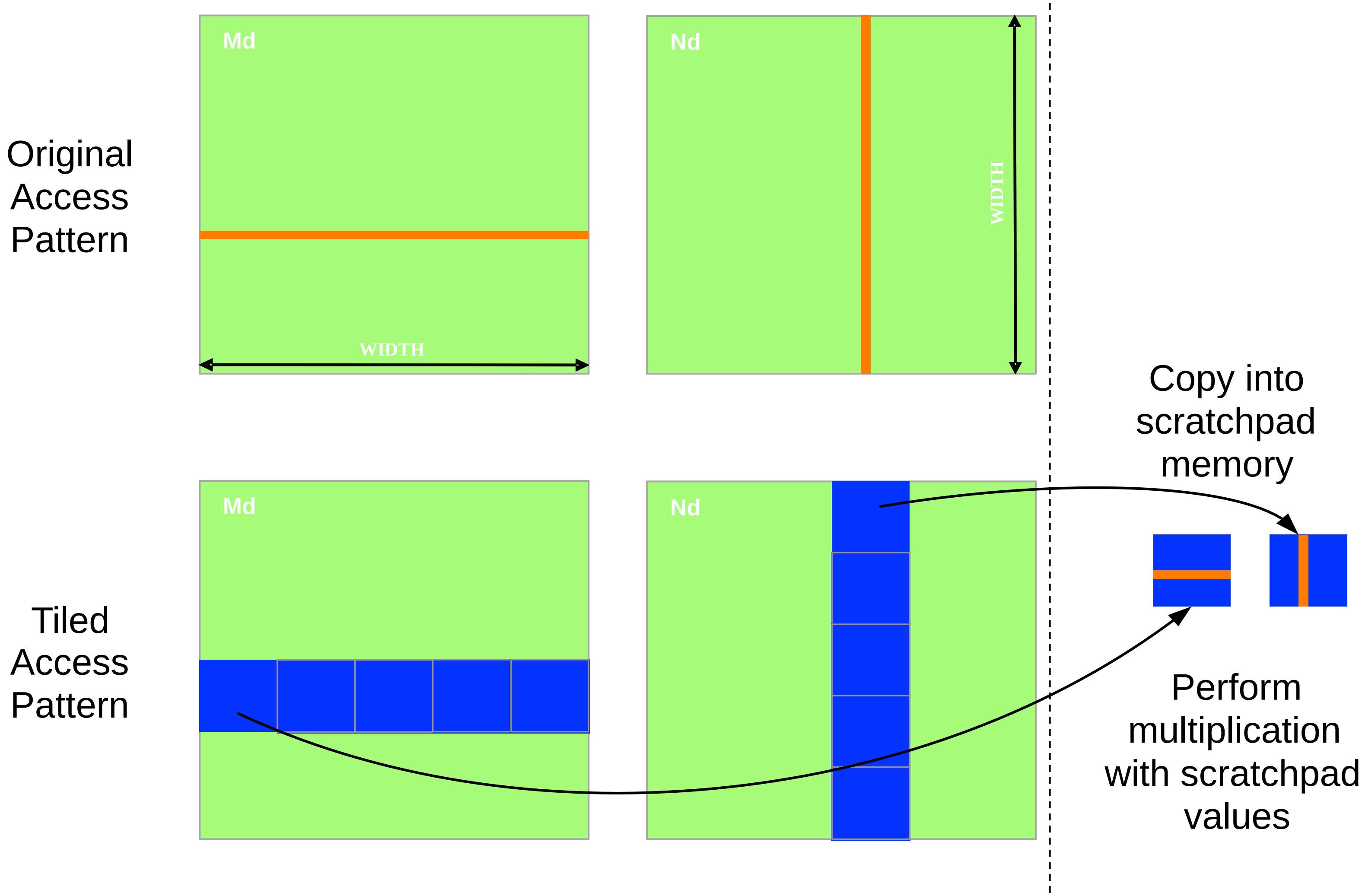




Memory Layout of a Matrix in C

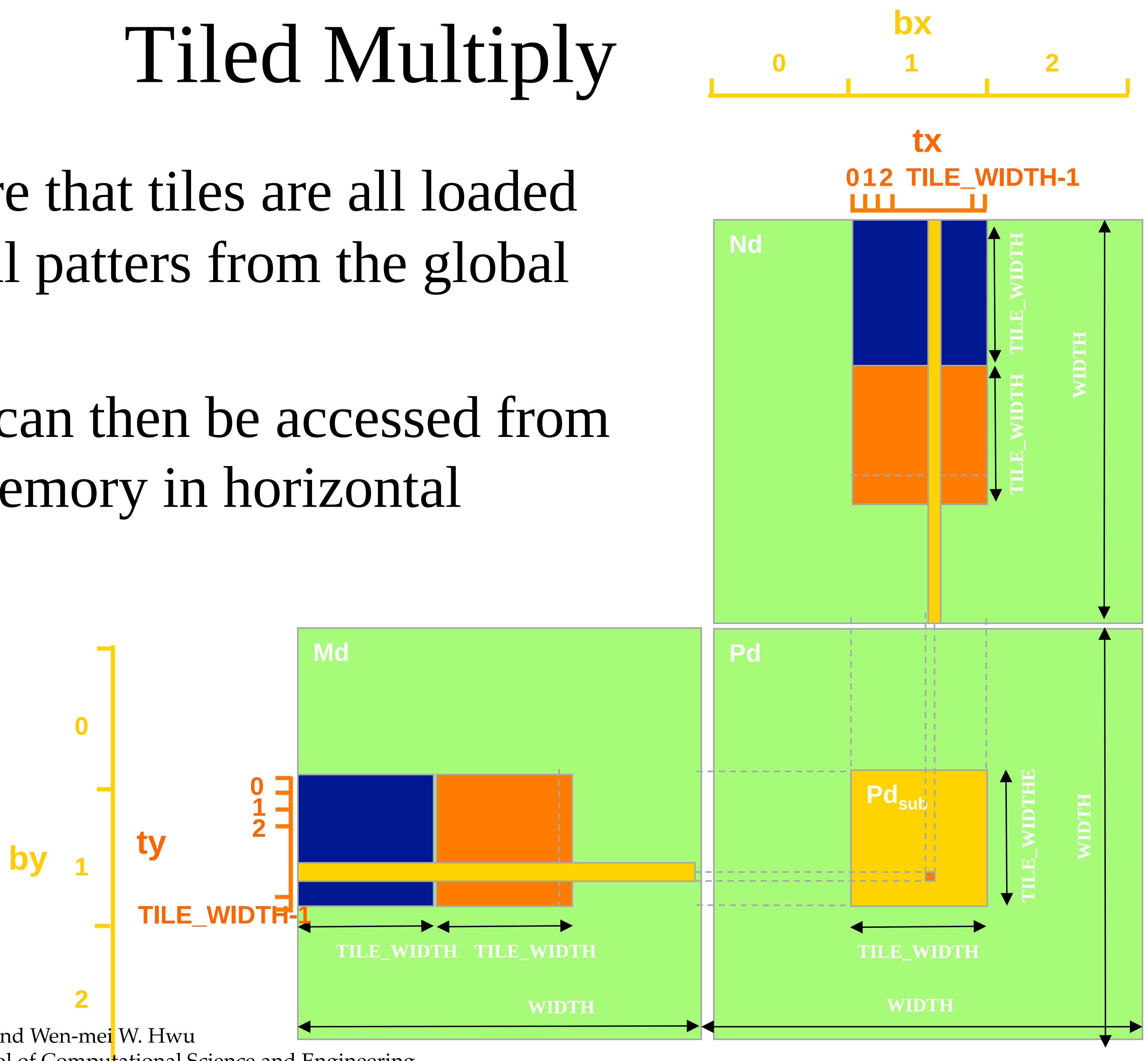


Memory Access Pattern



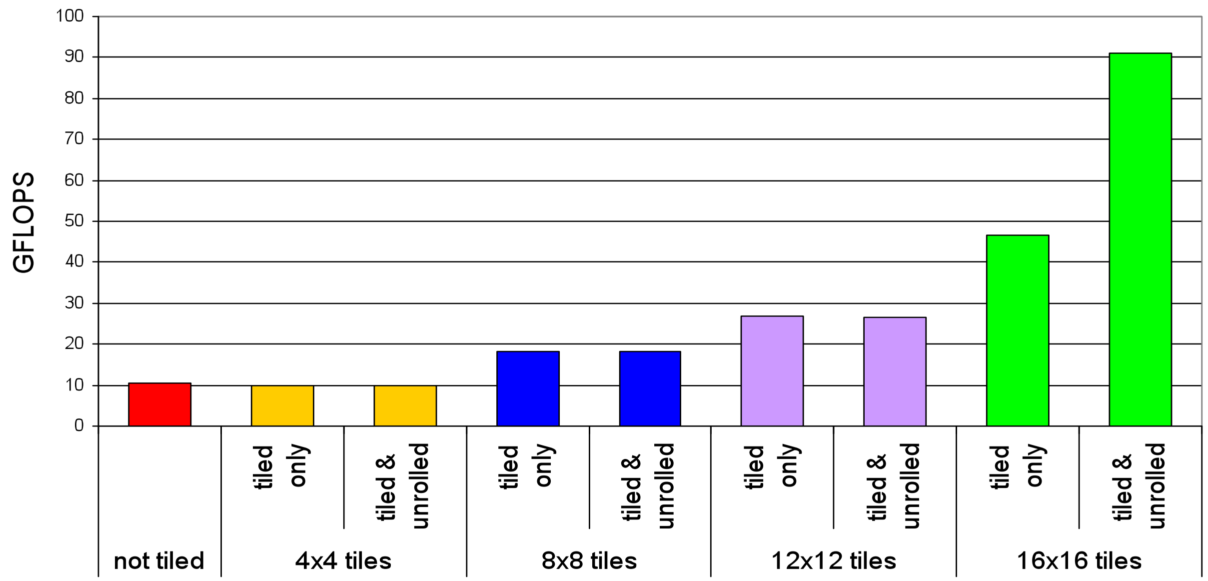
Tiled Multiply

- Make sure that tiles are all loaded in vertical patters from the global memory
- Md data can then be accessed from shared memory in horizontal direction



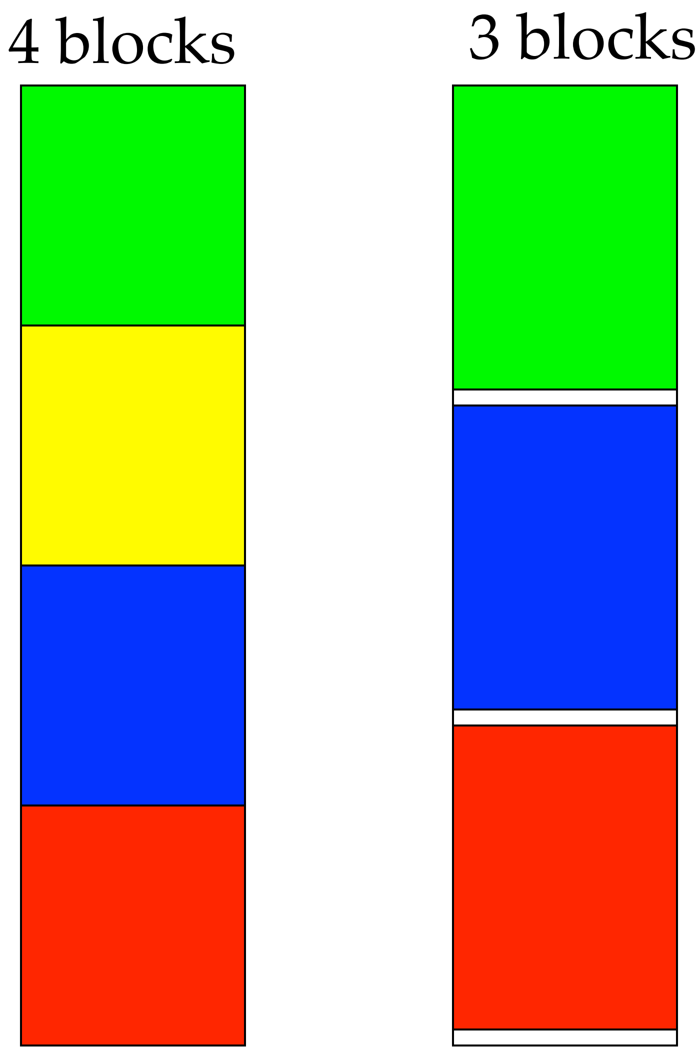
Tiling Size Effects

- For good bandwidth utilization, accesses should be aligned and consist of 16 contiguous words
- Tile size 16X16 minimal required to achieve full coalescing
  – Both reduction of global memory accesses and more efficient execution of the accesses



Programmer View of Register File

- There are 8192 registers in each SM in G80
  - This is an implementation decision, not part of CUDA
  - Registers are dynamically partitioned across all Blocks assigned to the SM
  - Once assigned to a Block, the register is NOT accessible by threads in other Blocks
  - Each thread in the same Block only access registers assigned to itself



Matrix Multiplication Example

- If each Block has 16X16 threads and each thread uses 10 registers, how many thread can run on each SM?
  - Each Block requires 10*256 = 2560 registers
  - 8192 = **3** * 2560 + change
  - So, three blocks can run on an SM as far as registers are concerned
- How about if each thread increases the use of registers by 1?
  - Each Block now requires 11*256 = 2816 registers
  - 8192 < 2816 *3
  - Only two Blocks can run on an SM, 1/3 reduction of thread-level parallelism (TLP)

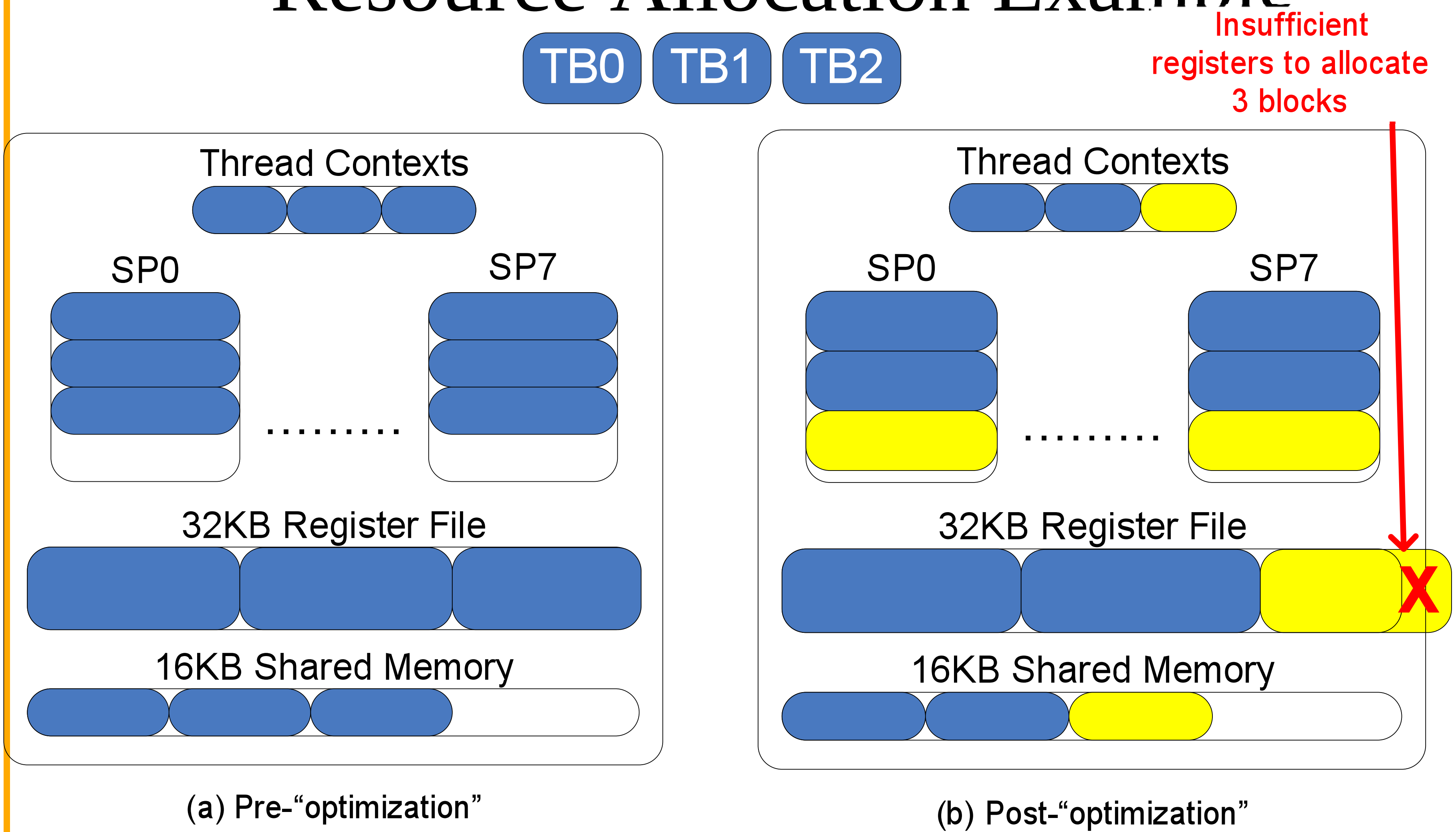
More on Dynamic Partitioning

- Dynamic partitioning of SM resources gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models.
  - The compiler can tradeoff between instruction-level parallelism and thread level parallelism

ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 10 registers, global loads have 200 cycles
  - 3 Blocks can run on each SM
- If a compiler can use one more register to change the dependence pattern so that 8 independent instructions exist for each global memory load
  - Only two can run on each SM
  - However, one only needs 200/(8*4) = 7 Warps to tolerate the memory latency
  - Two Blocks have 16 Warps. The performance can be actually higher!

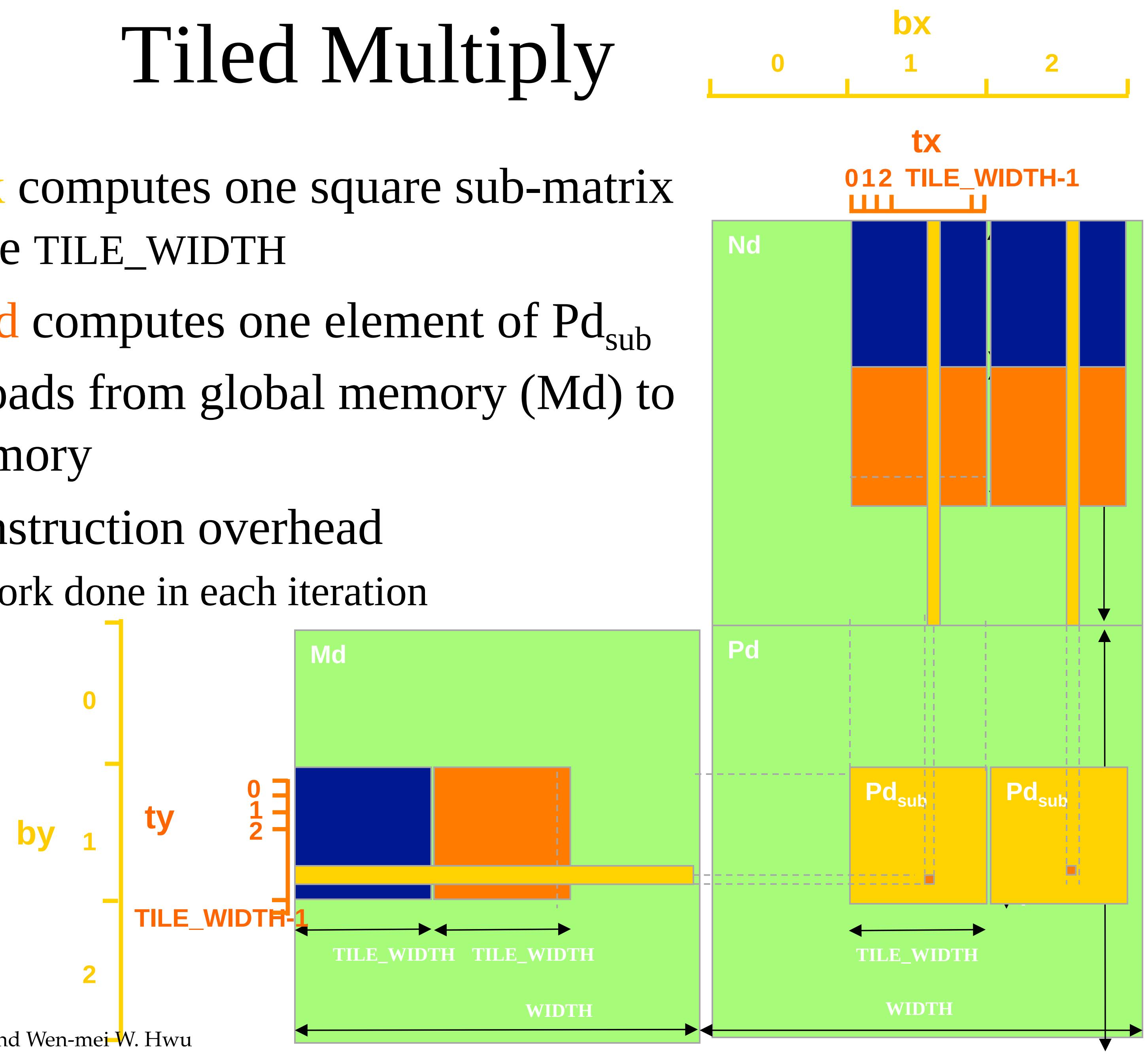
Resource Allocation Example



Increase in per-thread performance, but fewer threads: Lower overall performance in this case???

Tiled Multiply

- Each block computes one square sub-matrix Pd_{sub} of size TILE_WIDTH
- Each thread computes one element of Pd_{sub}
- Reduced loads from global memory (Md) to shared memory
- Reduced instruction overhead
  - More work done in each iteration



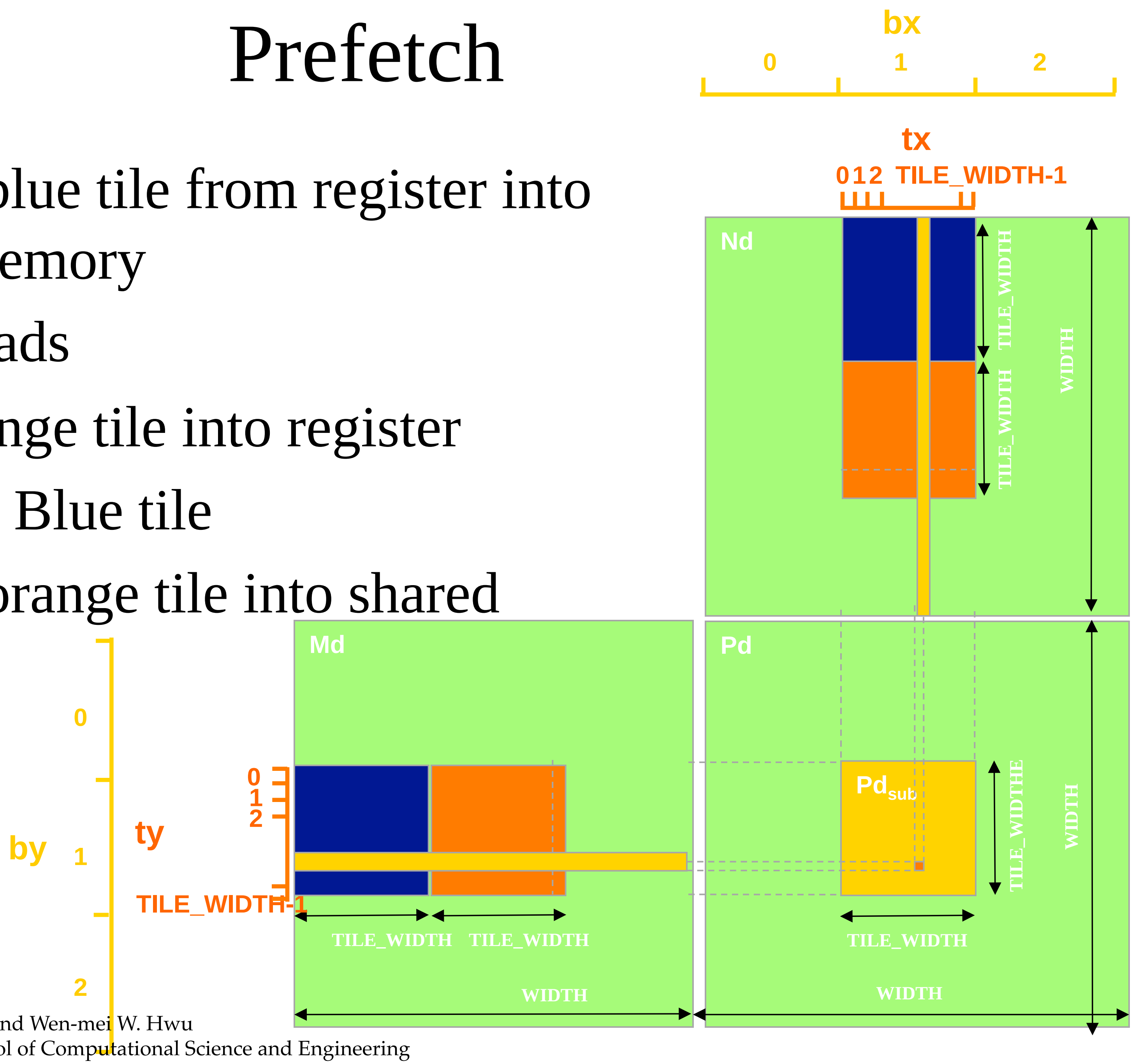
Prefetching

- One could double buffer the computation, getting better instruction mix within each thread
  - This is classic software pipelining in ILP compilers

Loop {
 	
Load current tile to shared memory
 	
__syncthreads()
 	
Compute current tile
 	
__syncthreads()
}
	Load next tile from global memory
	
	Loop {
	Deposit current tile to shared memory
	__syncthreads()
	
	Load next tile from global memory
	
	Compute current tile
	
	__syncthreads()
	}

Prefetch

- Deposit blue tile from register into shared memory
- Syncthreads
- Load orange tile into register
- Compute Blue tile
- Deposit orange tile into shared memory
- ….



Instruction Mix Considerations

```
for (int k = 0; k < BLOCK_SIZE; ++k)
    Pvalue += Ms[ty][k] * Ns[k][tx];
```

There are very few mul/add between branches and address calculation.

Loop unrolling can help.

```
Pvalue += Ms[ty][k] * Ns[k][tx] + …
              Ms[ty][k+15] * Ns[k+15][tx];
```

Unrolling

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  // compute results for tile
  for (i = 0; i < 16; i++)
    {
      Ctemp += As[ty][i]
        * Bs[i][tx];
    }

  __syncthreads();
}
C[indexC] = Ctemp;
```

(b) Tiled Version

```
Ctemp = 0;
for (...) {
  __shared__ float As[16][16];
  __shared__ float Bs[16][16];

  // load input tile elements
  As[ty][tx] = A[indexA];
  Bs[ty][tx] = B[indexB];
  indexA += 16;
  indexB += 16 * widthB;
  __syncthreads();

  // compute results for tile
  Ctemp +=
     As[ty][0] * Bs[0][tx];
  ...
  Ctemp +=
     As[ty][15] * Bs[15][tx];

  __syncthreads();
}
C[indexC] = Ctemp;
```

(c) Unrolled Version

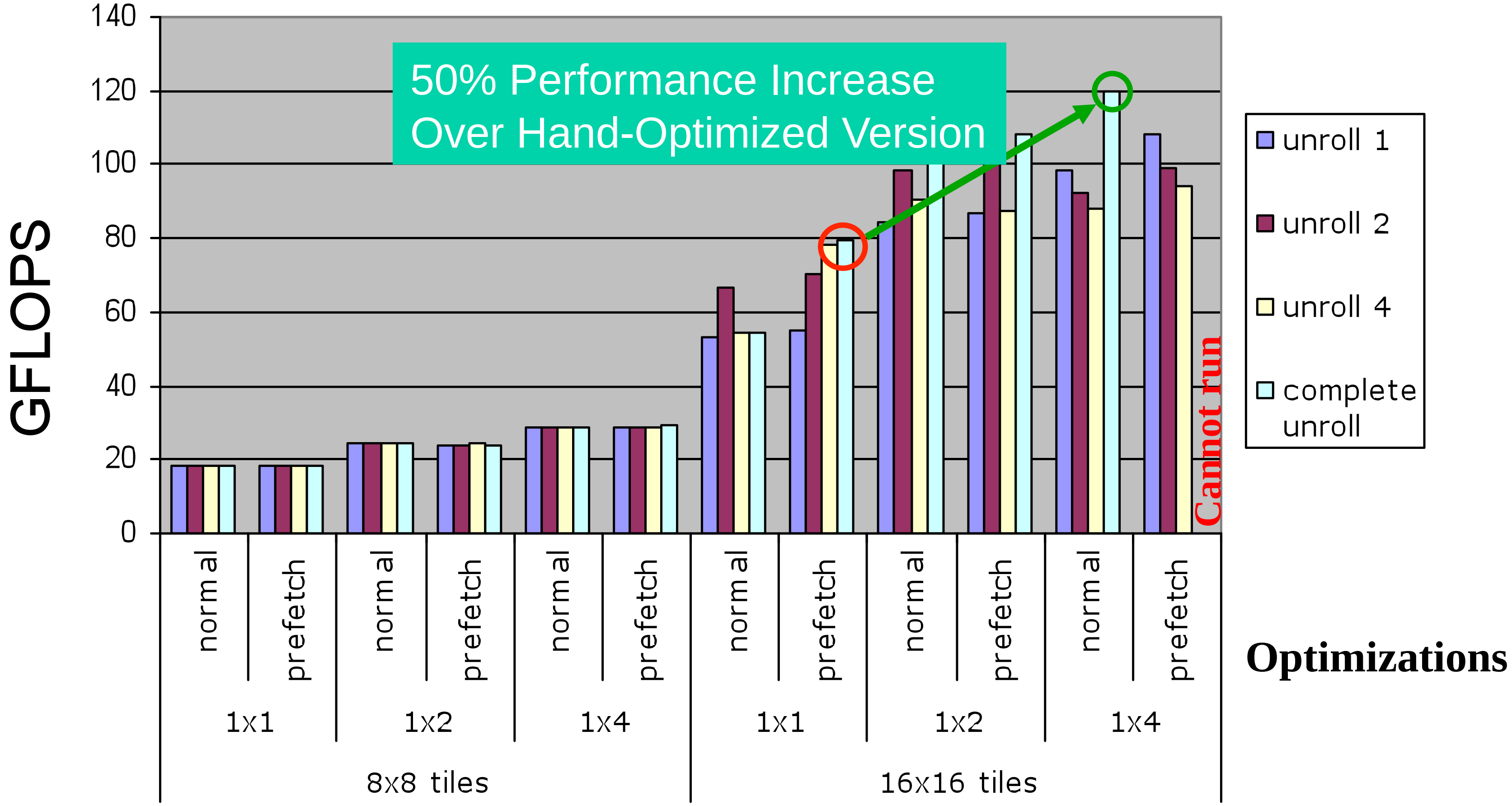
Does this use more registers?

Removal of branch instructions and address calculations

How Close Are We to Best Performance?

- Investigated applications with many optimizations
- Exhaustive optimization space search
  - Applied many different, controllable optimizations
  - Parameterized code by hand
- Hand-optimized code is deficient
  - Generally >15% from the best configuration
  - Trapped at local maxima
  - Often non-intuitive mix of optimizations

Matrix Multiplication Space



Major G80 Performance Detractors

- Long-latency operations
  - Avoid stalls by executing other threads
- Stalls and bubbles in the pipeline
  - Barrier synchronization
  - Branch divergence
- Shared resource saturation
  - Global memory bandwidth
  - Local memory capacity