NYU
New York University

CSCI-UA.0480-003

Parallel Computing

Lecture 21: CUDA - IV

Mohamed Zahran (aka Z)
mzahran@cs.nyu.edu
http://www.mzahran.com

Some Refreshing Exercises!

- Suppose registers and shared memory capacities were not an issue. When is it still beneficial to put values fetched from memory into the shared memory?

Some Refreshing Exercises!

- Assume a kernel is launched with 1000 blocks. Each block has 512 threads.
  - If a variable is declared as local in the kernel, how many versions will be created throughout the lifetime of the kernel?
  - How about if the variable is created as shared?

Some Refreshing Exercises!

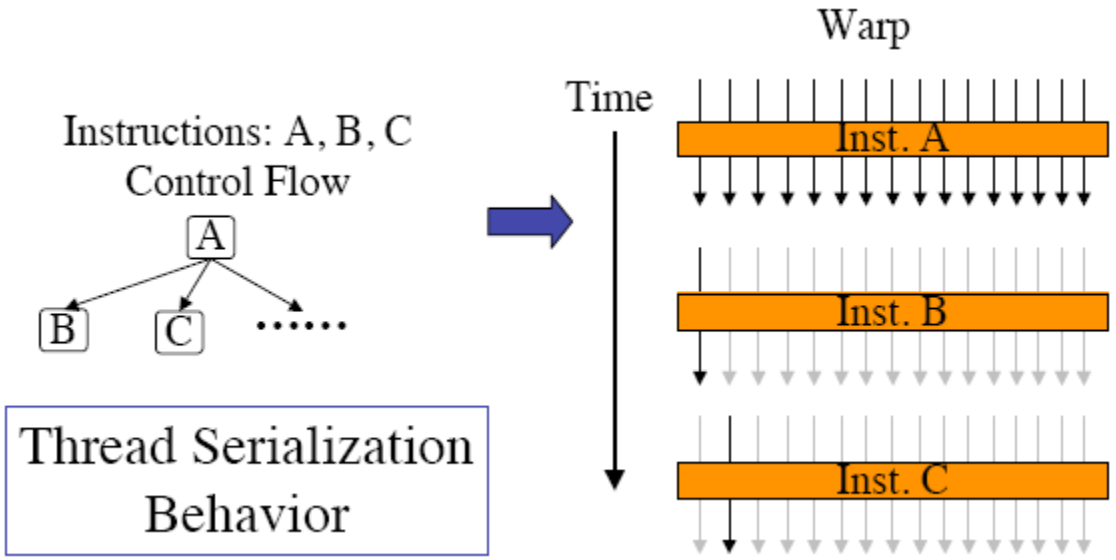
- A kernel contains 36 floating point operations and 7 32-bit word global memory accesses per thread. For each of the following device properties, indicate whether this kernel is compute- or memory-bound.
  - Peak FLOPS = 200 GFLOPS, Peak Memory Bandwidth = 100 GB/s
  - Peak FLOPS = 300 GFLOPS, Peak Memory Bandwidth = 250 GB/s

Performance Considerations

- There are many hardware constraints.
- Depending on the application, different constraints may dominate.
- We can improve performance of an application by trading one resource usage for another.

Performance Issue: Thread Diversion

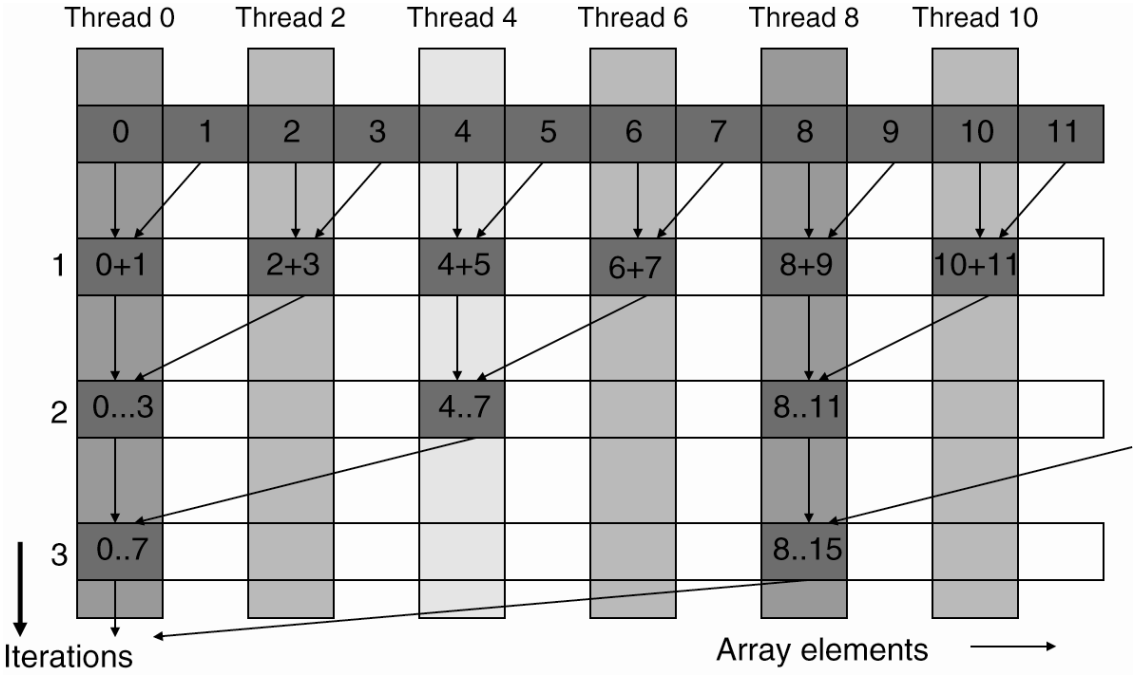
- Works well when all threads in a warp follow the same control-flow
- Performance loss due to thread diversion



Performance Issue: Thread Diversion

```
1. __shared__ float partialSum[]

2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = 1;
4.      stride < blockDim.x; stride *= 2)
5. {
6.    syncthreads();
7.  if (t % (2*stride) == 0)
8.   partialSum[t] += partialSum[t+stride];
9. }
```

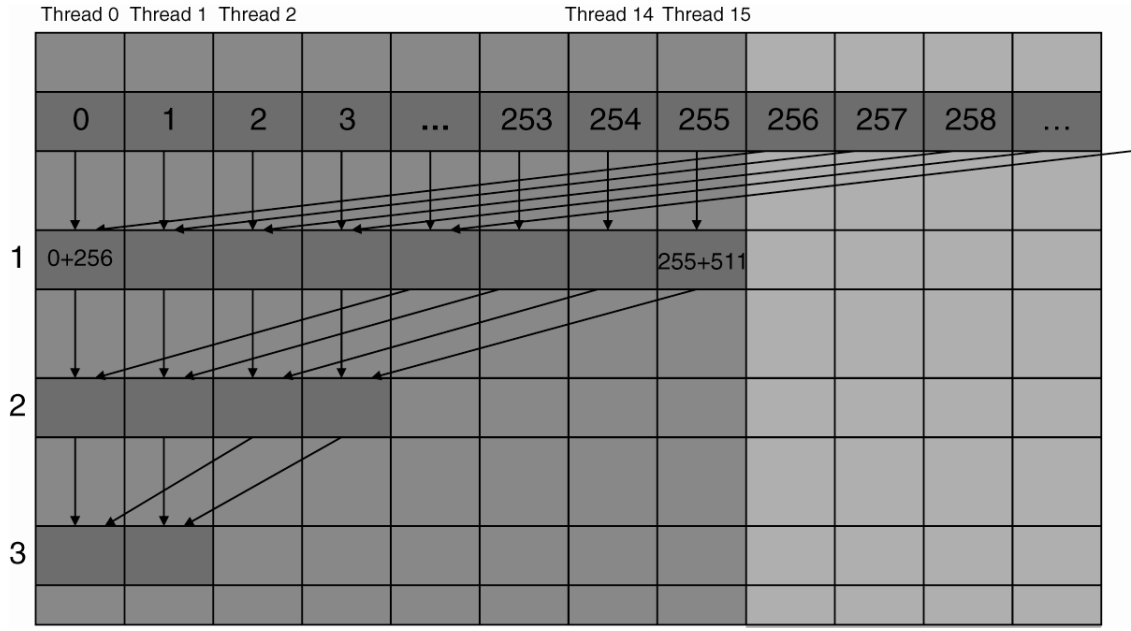


Example: Sum Reduction Kernel

Performance Issue: Thread Diversion

```
1. __shared__ float partialSum[];

2. unsigned int t = threadIdx.x;
3. for (unsigned int stride = blockDim.x>>1;
4.   stride > 0; stride >>= 1)
5. {
6.   __syncthreads();
7.   if (t < stride)
8.     partialSum[t] += partialSum[t+stride];
9. }
```



Why is this version better than the previous one?

Example: Sum Reduction Kernel

Performance Issue: Global Memory

- Typical application: process massive amount of data within short period of time
  - From global memory
  - large amount + short period = huge bandwidth requirement
- Two main challenges regarding global memory:
  - Long latency
  - Relatively limited bandwidth

Dealing With Global Memory: **TILING**

- We have seen this before
- Make use of shared memory available in SMs to reduce trips to global memory

Dealing With Global Memory: **Coalescing**

- To more effectively move data from global memory to shared memory and registers
- For best results: can be used with tiling
- Global memory:
  - DRAM
  - Reading a bit is slow
  - So memory is implemented to read several bits in parallel

Dealing With Global Memory: **Coalescing**

- If an application can make use of data from multiple consecutive locations, the DRAM can supply the data in much higher rate.
- Kernel must arrange its data access accordingly
- When all threads in a warp execute a load instruction:
  - The hardware detects whether the addresses are consecutive
  - The hardware combines (coalesces) all accesses in a consolidated access to consecutive DRAM locations

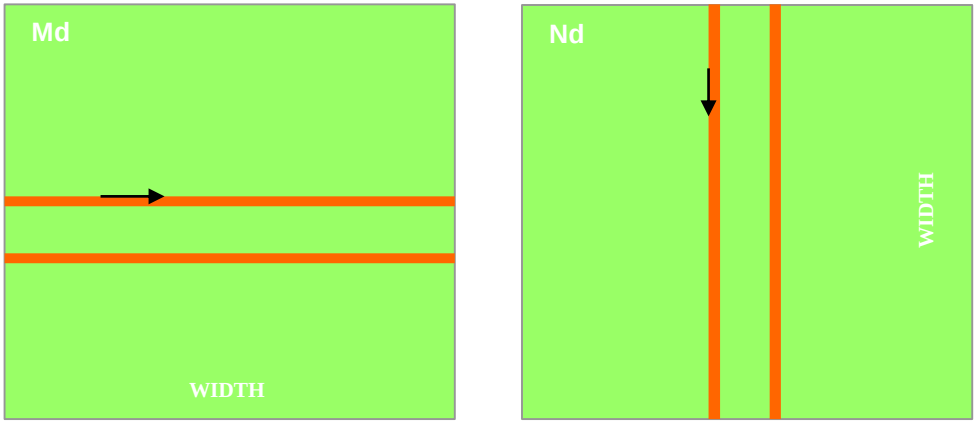
The device *coalesces* global memory loads and stores issued by threads of a warp into as few transactions as possible to minimize DRAM bandwidth

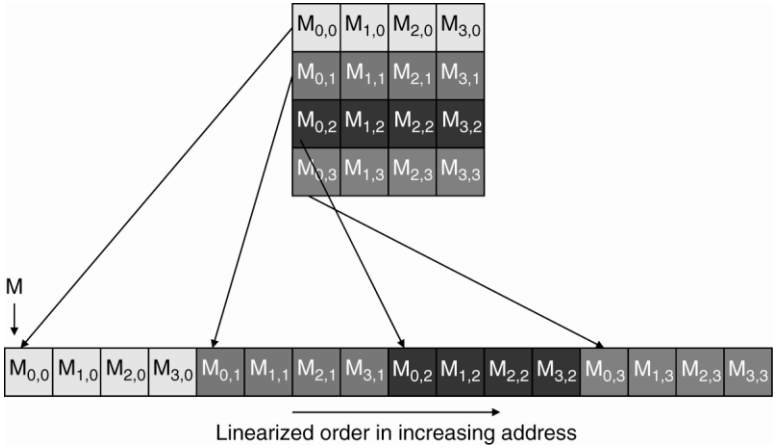
Dealing With Global Memory: **Coalescing**

Not coalesced

coalesced

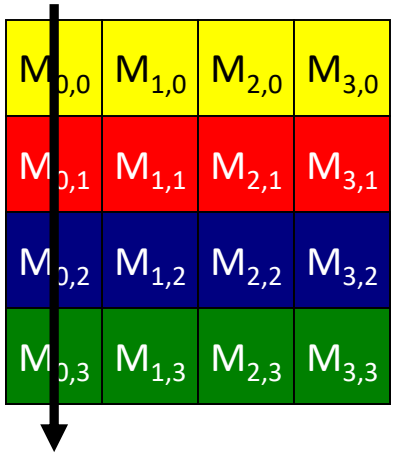
Same warp: Thread 1, Thread 2

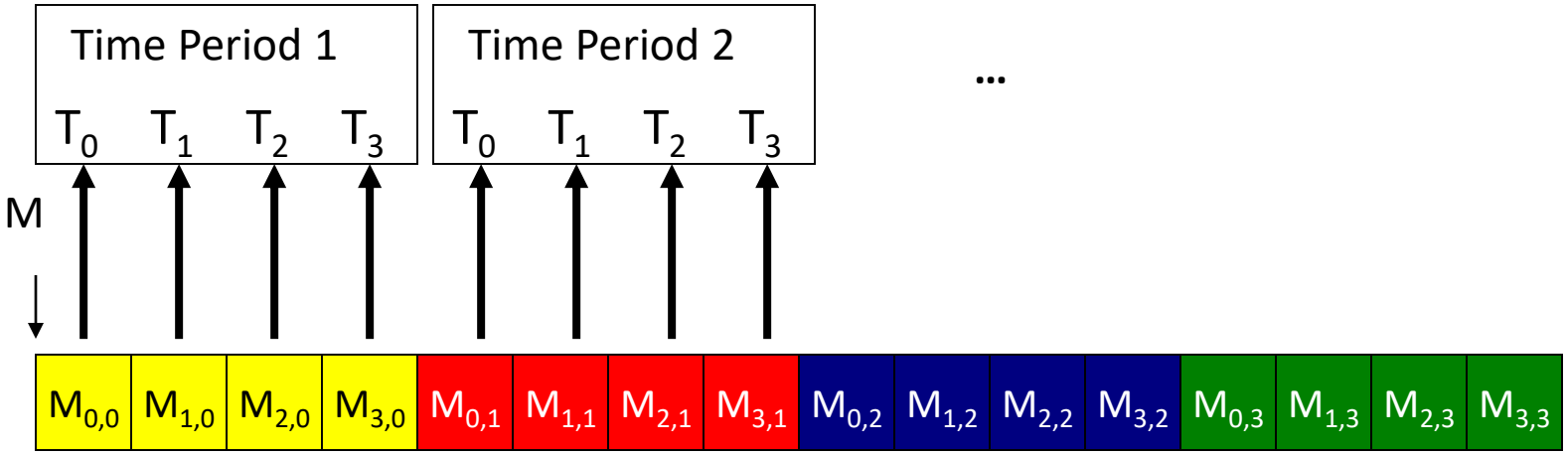




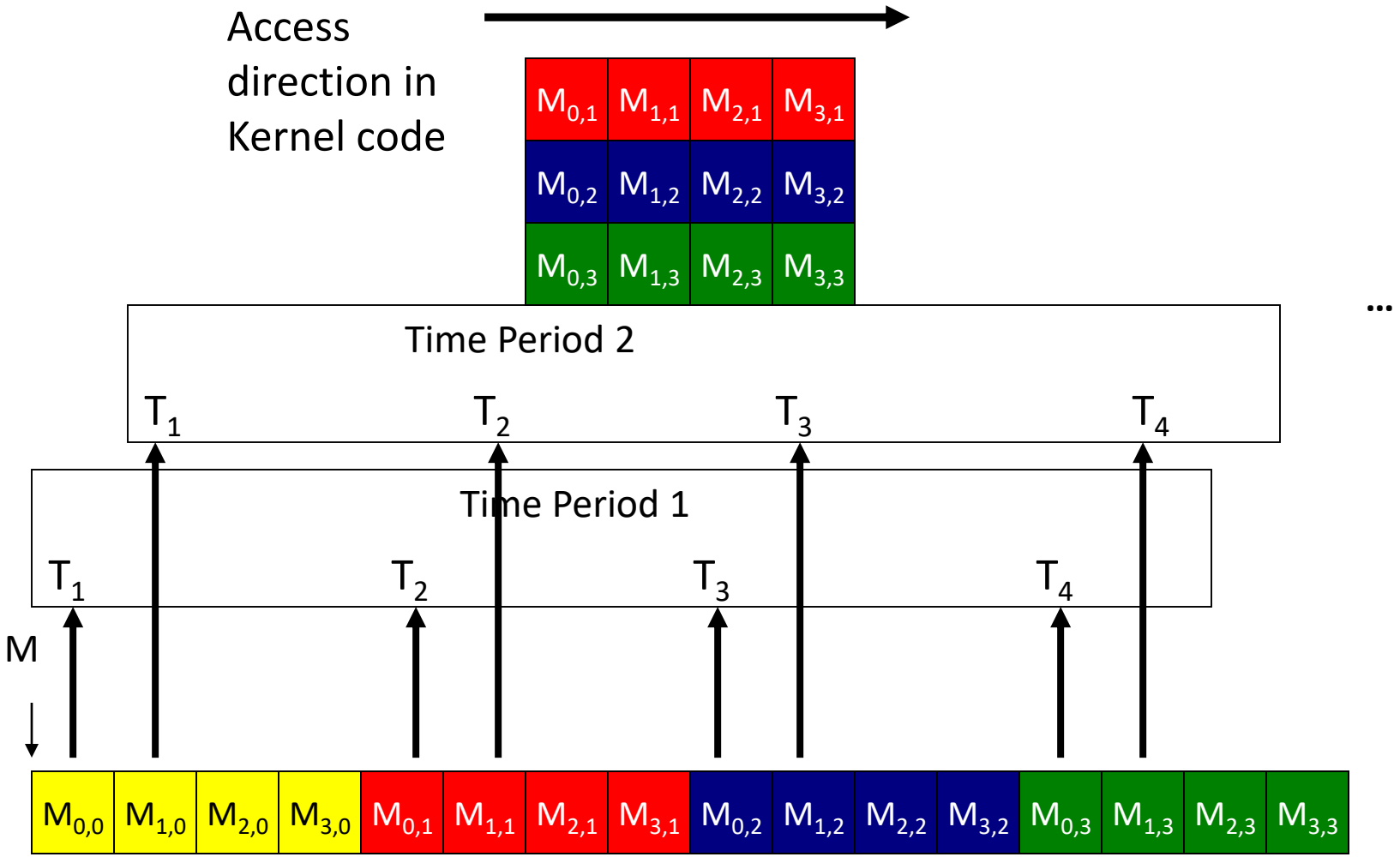
Dealing With Global Memory: **Coalescing**

Favorable access direction in Kernel code of one thread





Dealing With Global Memory: **Coalescing**



Dealing With Global Memory: **Coalescing**

- Device memory is accessed via 32-, 64-, or 128-byte memory transactions.
- These memory transactions must be naturally aligned, for example:
  - 32-byte transactions must have a starting address of: 0, 32, 64, 96, …
- Global memory instructions support reading or writing words of size equal to 1, 2, 4, 8, or 16 bytes.
  - Also the data must be aligned (i.e., its address is a multiple of that size).
  - The alignment requirement is automatically fulfilled for the built-in types.

Dealing With Global Memory: **Coalescing**

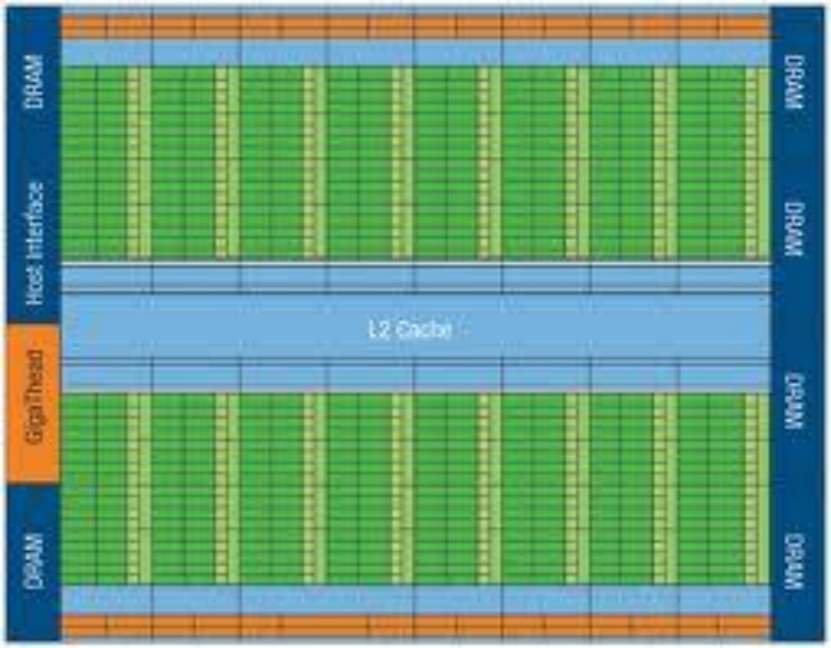
- For structures, the size and alignment requirements can be enforced by the compiler using the alignment specifiers __align__(8) or __align__(16)

Example:

```
struct __align__(16) {
        float x;
        float y;
        float z; };
```

Dealing With Global Memory: **Cache**

- Fermi (and up) has cache for global memory (L2 cache)
- Caches automatically coalesce most of kernel access patterns



Dealing With Global Memory: **Prefetching**

- Prefetch next data elements while consuming the current data elements.

Performance Issue: SM Resources

- Execution resources in SM include:
  - registers
  - block slots
  - thread slots
- There is an interaction among the resources that you must take into account.

Performance Issue: SM Resources

Example: Assume G80 executing the matrix multiplication with 16x16 thread blocks
(SM: 8 block slots, 768 thread slots, 8192 registers)

If a thread needs 10 registers then:

- A block needs 10x16x16 = 2560 registers
- 3 blocks -> 7680 registers (under the 8192 limit)
- We can't add another block (will make it 10240)
- 3 blocks x 256 threads/block = 768 (within limit)

By using 1 extra variable the program saw a 1/3 reduction in warp parallelism **-> performance cliff**

Assume the programmer declares one more auto var:

- 11 x 16 x 16 = 2816 registers per block
- 3 blocks -> 3 x 2816 = 8448 (above limit)
- SM reduces #blocks by 1 -> 5632 registers required
- This reduces the number of threads in SM to 2 x256 -> 512

Performance Issue: SM Resources

Example: Still with G80:

- An instruction takes 4 cycles
- Assume 4 independent instructions between global memory load and its use
- Global memory latency is 200 cycles

To keep execution units fully utilized:

We need to have 200/(4 x 4) = 14 warps

Trading thread-level parallelism with increased thread performance

Assume an extra register allows the programmer to use a transformation to increase independent instructions from 4 to 8, then:

- Now we need 200/(4 x 8) = 7 warps
- From previous slide, 1 block = 16x16 =256 threads = 8 warps
- Blocks reduced from 3 to 2 (look at prev. slide)-> warps reduced from 24 to 16
- Still we can fully utilize execution units

Performance Issue: Instruction Mix

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

Is the above code efficient?

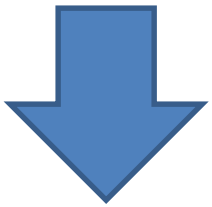
- Extra instructions to update loop counter
- Extra instructions for conditional branch at the end of each iteration
- Using k to access matrices incurs address arithmetic instructions.
- All of the above compete with the floating-point calculations for limited instruction processing bandwidth.

2 FP arithmetic
2 address arithmetic instructions
1 loop branch instructions
1 loop increment instructions

} **only 1/3 instructions are FP operations**

Performance Issue: Instruction Mix

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



```
Pvalue += Ms[ty][0] * Ns[0][tx] + …
          Ms[ty][15] * Ns[15][tx];
```

Loop unrolling

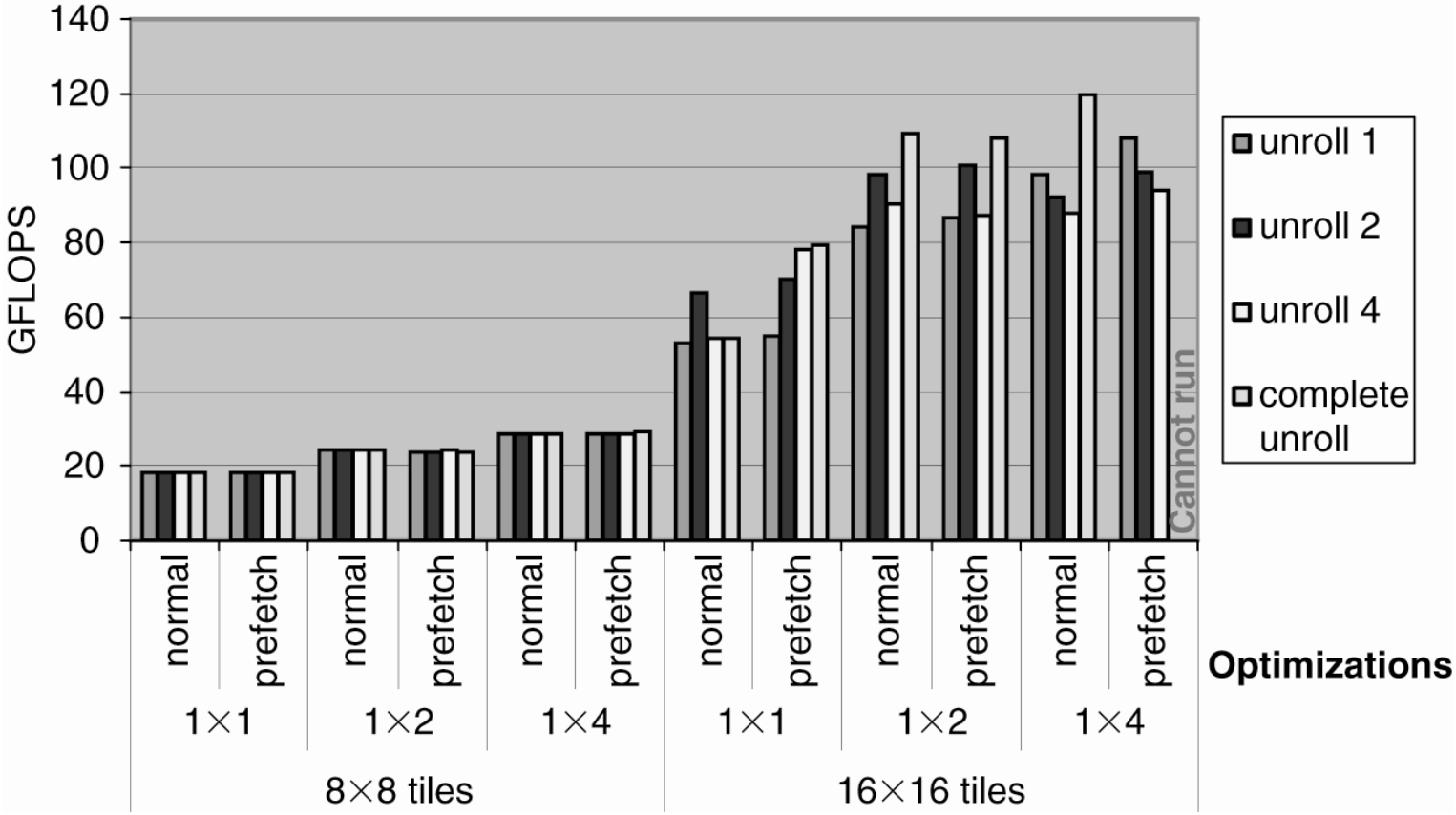
Performance Issue: Thread Granularity

- Algorithmic decision
- It is often advantageous to put more work into each thread and use fewer threads when redundant work exists among threads
  - Example: Let a thread compute 2 tiles

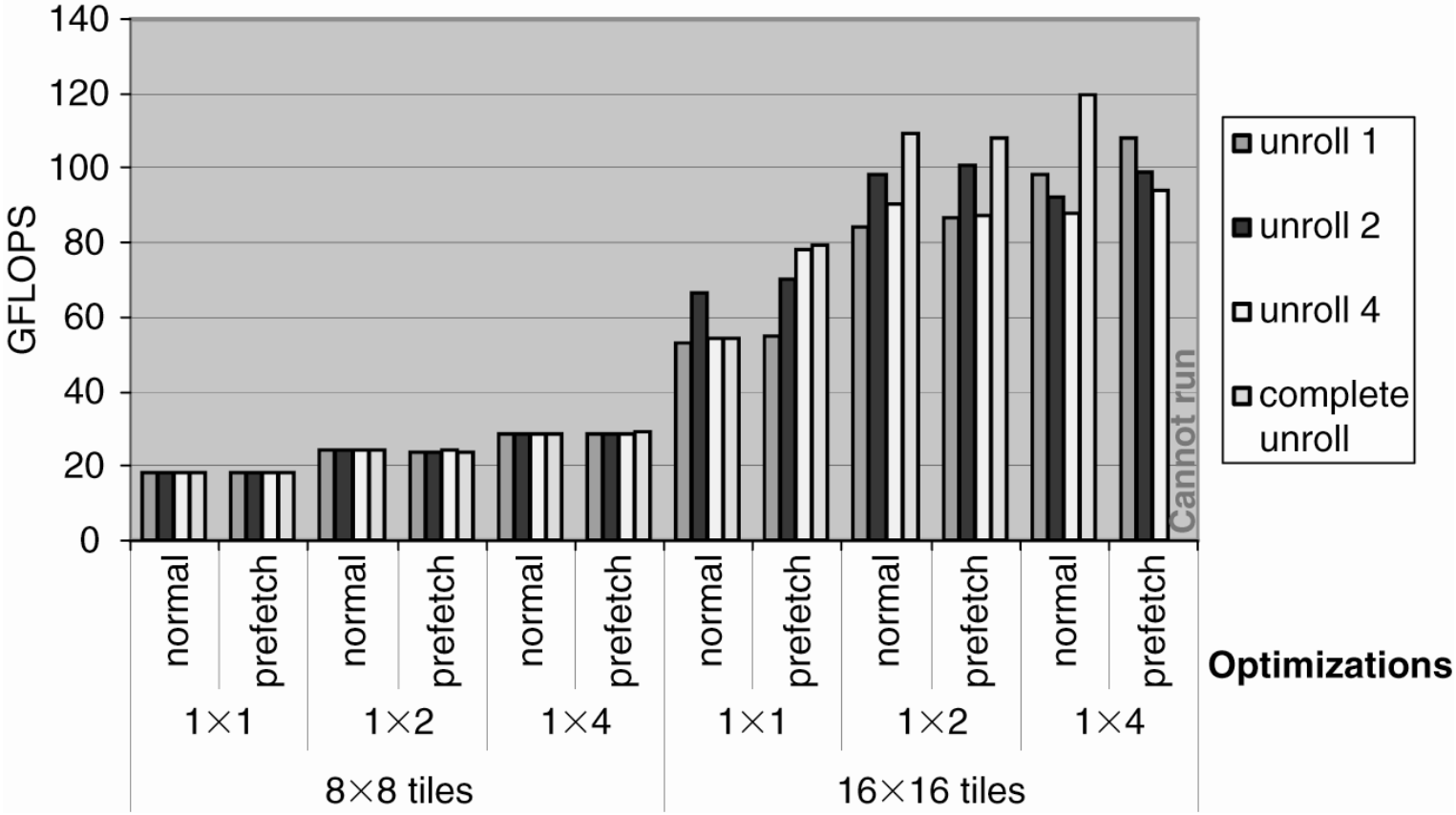
+ Less redundant work

- More resources requirements

Putting It All Together



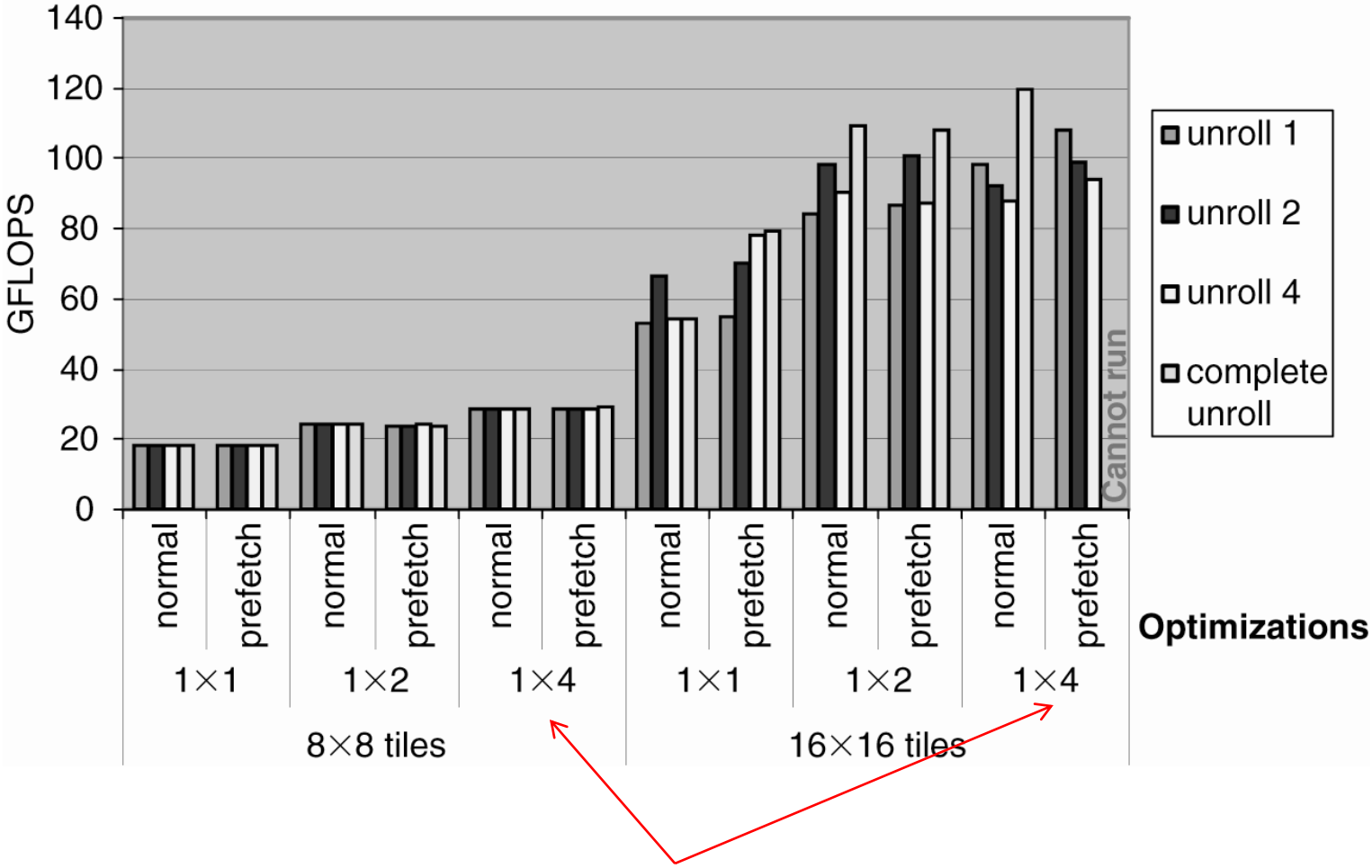
Putting It All Together



Until tile reaches 16x16 neither loop unrolling nor data prefetch helps.

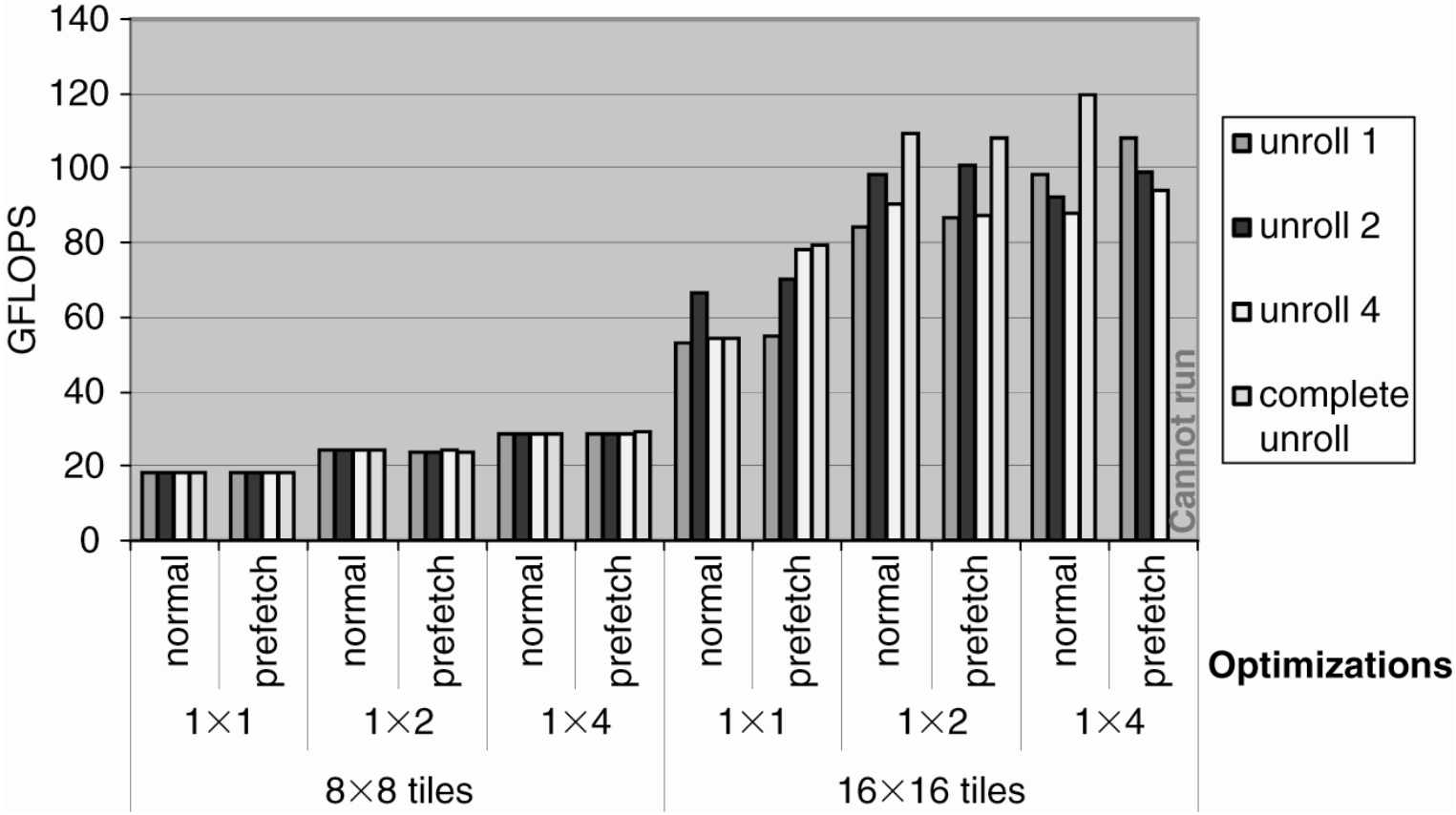
For small tile size, global memory bandwidth severely limits performance.

Putting It All Together



Granularity adjustment can reduce global memory access.

Putting It All Together



Data prefetching becomes less beneficial as thread granularity increases.

About Algorithms for GPUs

- When designing an algorithm for GPU, the two main characteristics that determine its performance are:
  1. How much data parallelism is available
  2. How much data must move through the memory hierarchy
- Then when you move from algorithm → code you need to take hardware constraints into account.

Conclusions

- As we program GPUs we need to pay attention to several performance bottlenecks:
  - Branch diversion
  - Global memory latency
  - Global memory bandwidth
  - Limited resources
- We have several techniques in our arsenal to enhance performance
  - Try to make threads in the same warp follow the same control flow
  - Tiling
  - Coalescing
  - Loop unrolling
  - Increase thread granularity
  - Trade one resource for another
- Pay attention to interaction among techniques