

CS 677: Parallel Programming for Many-core Processors

Lecture 4

Instructor: Philippos Mordohai

Webpage: www.cs.stevens.edu/~mordohai

E-mail: Philippos.Mordohai@stevens.edu

Logistics

- **NEW** Midterm: March 27
- Project proposal presentations: March 13
 - Have to be approved by me by March 8

Project Proposal

- Problem description
 - What is the computation and why is it important?
 - Abstraction of computation: equations, graphic or pseudo-code, no more than 1 page
- Suitability for GPU acceleration
 - Amdahl's Law: describe the inherent parallelism. Argue that it is close to 100% of computation.
 - Synchronization and Communication: Discuss what data structures may need to be protected by synchronization, or communication through host.
 - Copy Overhead: Discuss the data footprint and anticipated cost of copying to/from host memory.
- Intellectual Challenges
 - Generally, what makes this computation worthy of a project?
 - Point to any difficulties you anticipate at present in achieving high speedup

Amdahl's Law

- “The speedup of a program using multiple processors in parallel computing is limited by the time needed for the sequential fraction of the program.”
- Example
 - 95% of original execution time can be sped up by 100x on GPU
 - Speed up for entire application:

$$\frac{1}{(5\% + \frac{95\%}{100})} = \frac{1}{5\% + 0.95\%} = \frac{1}{5.95\%} = 17x$$

Overview

- More Performance Considerations
 - Memory Coalescing
 - Occupancy
 - Kernel Launch Overhead
 - Instruction Performance
- Summary of Performance Considerations
 - Lectures 3 and 4
- Timers

Memory Coalescing (Part 2)

slides by

Jared Hoberock and David Tarjan
(Stanford CS 193G)

Consider the stride of your accesses

```
__global__ void foo(int* input,
                   float3* input2)
{
    int i = blockDim.x * blockIdx.x
          + threadIdx.x;
    // Stride 1
    int a = input[i];
    // Stride 2, half the bandwidth is wasted
    int b = input[2*i];
    // Stride 3, 2/3 of the bandwidth wasted
    float c = input2[i].x;
}
```

Example: Array of Structures (AoS)

```
struct record
{
    int key;
    int value;
    int flag;
};
```

```
record    *d_records;
cudaMalloc((void**) &d_records,
    ...);
```


Example: Structure of Arrays (SoA)

```
struct SoA
{
    int * keys;
    int * values;
    int * flags;
};
```

```
SoA d_SoA_data;
```

```
cudaMalloc((void**) &d_SoA_data.keys, ...);
```

```
cudaMalloc((void**) &d_SoA_data.values, ...);
```

```
cudaMalloc((void**) &d_SoA_data.flags, ...);
```

Example: SoA vs. AoS

```
__global__ void bar(record
*AoS_data, SoA SoA_data)
{
    int i = blockDim.x * blockIdx.x
          + threadIdx.x;
    // AoS wastes bandwidth
    int key = AoS_data[i].key;
    // SoA efficient use of bandwidth
    int key_better = SoA_data.keys[i];
}
```

Memory Coalescing

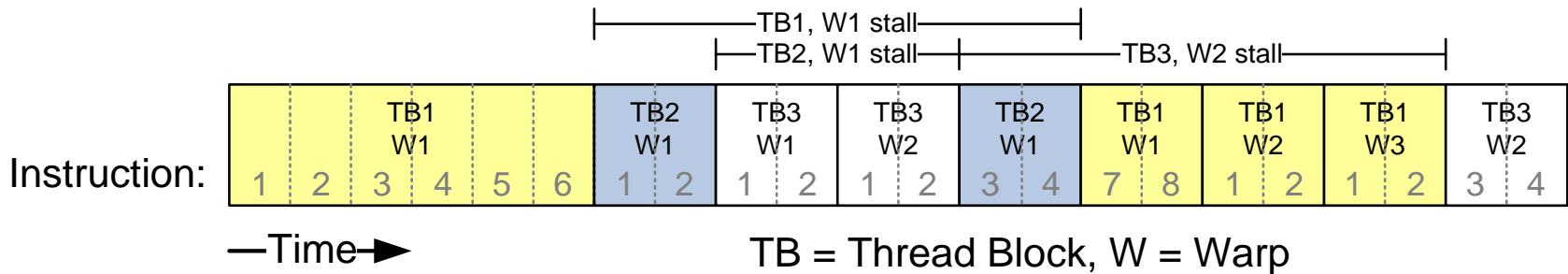
- Structure of arrays is often better than array of structures
 - Very clear win on regular, stride 1 access patterns
 - Unpredictable or irregular access patterns are case-by-case

Occupancy

slides (mostly) by
Jared Hoberock and David Tarjan
(Stanford CS 193G)
and Joseph T. Kider Jr. (UPenn)

Reminder: Thread Scheduling

- SM implements zero-overhead warp scheduling
 - At any time, only one of the warps is executed by SM
 - Warps whose next instruction has its inputs ready for consumption are eligible for execution
 - Eligible Warps are selected for execution on a prioritized scheduling policy
 - All threads in a warp execute the same instruction when selected



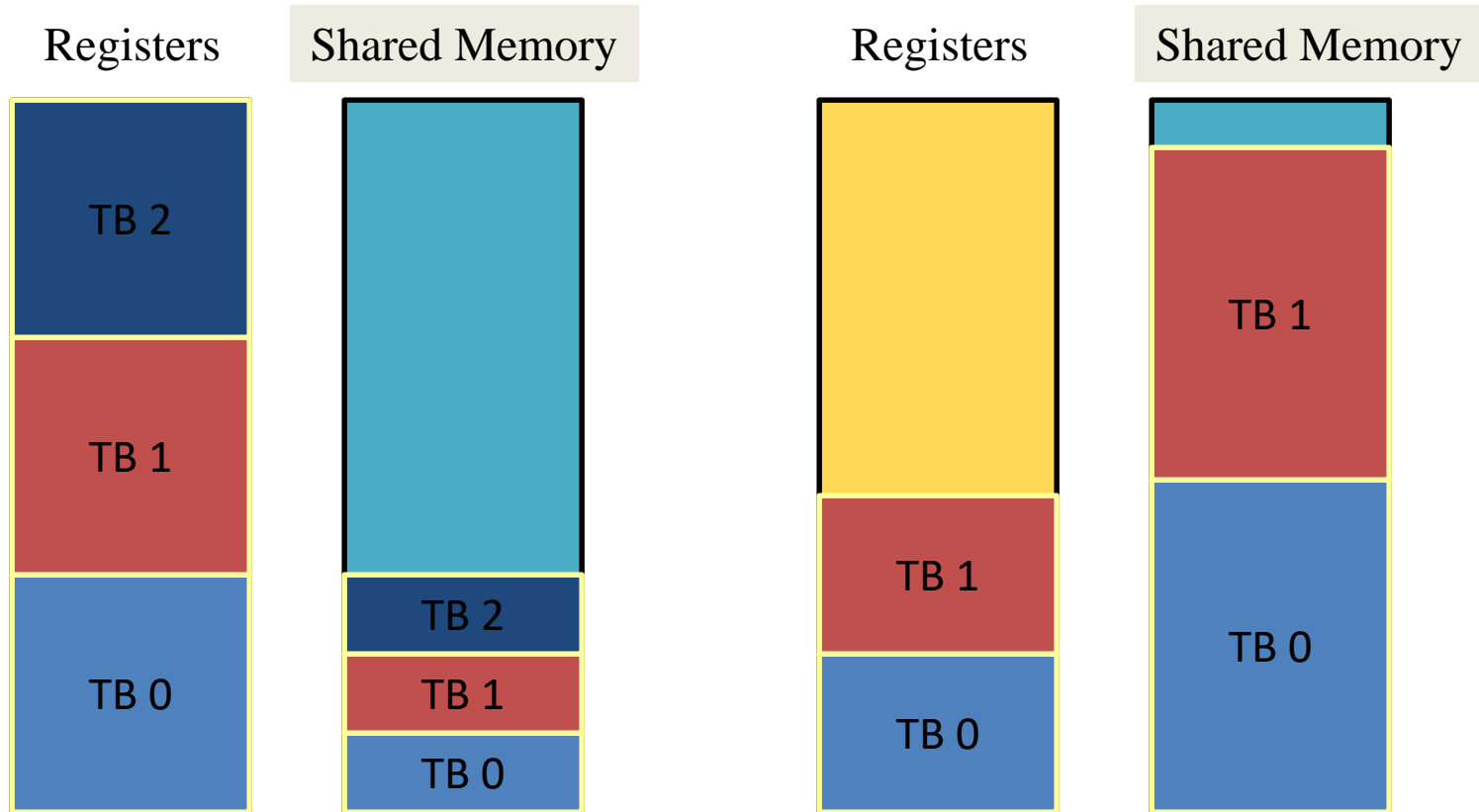
Thread Scheduling

- What happens if all warps are stalled?
 - No instruction issued → performance lost
- Most common reason for stalling?
 - Waiting on global memory
- If your code reads global memory every couple of instructions
 - You should try to maximize occupancy

Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep cores busy
- **Occupancy** = number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently
- Limited by resource usage:
 - Registers
 - Shared memory

Resource Limits (1)



- Pool of registers and shared memory per SM
 - Each thread block grabs registers & shared memory
 - If one or the other is fully utilized -> no more thread blocks

Resource Limits (2)

- Can only have N thread blocks per SM
 - If they're too small, can't fill up the SM
 - Need 128 threads / block (GT200), 192 threads / block (GF100)
- Higher occupancy has diminishing returns for hiding latency

Grid/Block Size Heuristics

- **# of blocks > # of multiprocessors**
 - So all multiprocessors have at least one block to execute
- **# of blocks / # of multiprocessors > 2**
 - Multiple blocks can run concurrently on a multiprocessor
 - Blocks not waiting at a `__syncthreads()` keep hardware busy
 - Subject to resource availability - registers, shared memory
- **# of blocks > 100 to scale to future devices**

Register Dependency

- Read-after-write register dependency
 - Instruction's result can be read approximately **24 cycles later**
- To completely hide latency:
 - Run at least 192 threads (6 warps) per multiprocessor
 - At least 25% occupancy for compute capability 1.0 and 1.1
 - Threads do not have to belong to the same block

Register Pressure

- Hide latency by using more threads per SM
- Limiting factors:
 - Number of registers per thread
 - 8k/16k/... per SM, partitioned among concurrent threads
 - Amount of shared memory
 - 16kB/... per SM, partitioned among concurrent blocks

How do you know what you're using?

- Use `nvcc -Xptxas -v` to get register and shared memory usage

```
nvcc -Xptxas -v acos.cu
ptxas info : Compiling entry function 'acos_main'
ptxas info : Used 4 registers, 60+56 bytes lmem, 44+40 bytes
             smem, 20 bytes cmem[1], 12 bytes cmem[14]
```

- The first number represents the total size of all the variables declared in that memory segment and the second number represents the amount of system allocated data.
 - Constant memory numbers include which memory banks have been used
- Plug those numbers into [CUDA Occupancy Calculator](#)

Home Insert Page Layout Formulas Data Review View

Cut Copy Paste Format Painter Clipboard

Arial 10 Font

General Number Alignment

Normal Bad Good Neutral Calculation Check Cell Styles

Insert Delete Format Cells

AutoSum Fill Clear Sort & Filter Find & Select Editing

Security Warning Macros have been disabled. Options...

MyRegCount 25

1.) Select Compute Capability (click): **1.3** (Help)

2.) Enter your resource usage:

Threads Per Block	128	(Help)
Registers Per Thread	25	
Shared Memory Per Block (bytes)	640	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs: (Help)

Active Threads per Multiprocessor	512
Active Warps per Multiprocessor	16
Active Thread Blocks per Multiprocessor	4
Occupancy of each Multiprocessor	50%

Physical Limits for GPU Compute Capability: 1.3

Threads per Warp	32
Warps per Multiprocessor	32
Threads per Multiprocessor	1024
Thread Blocks per Multiprocessor	8
Total # of 32-bit registers per Multiprocessor	16384
Register allocation unit size	512
Register allocation granularity	block
Shared Memory per Multiprocessor (bytes)	16384
Shared Memory Allocation unit size	512
Warp allocation granularity (for register allocation)	2

Allocation Per Thread Block

Warps	4
Registers	3584
Shared Memory	1024

These data are used in computing the occupancy data in blue

Maximum Thread Blocks Per Multiprocessor Blocks

Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

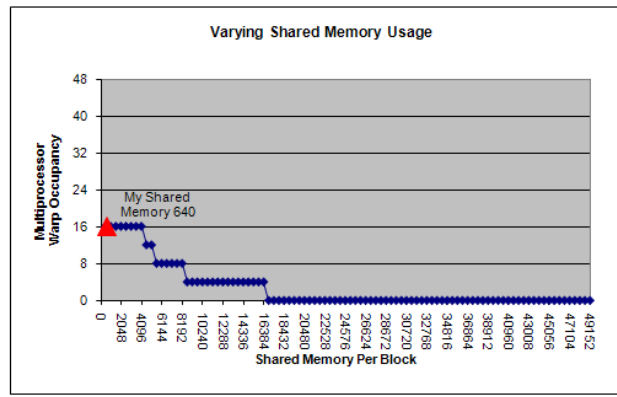
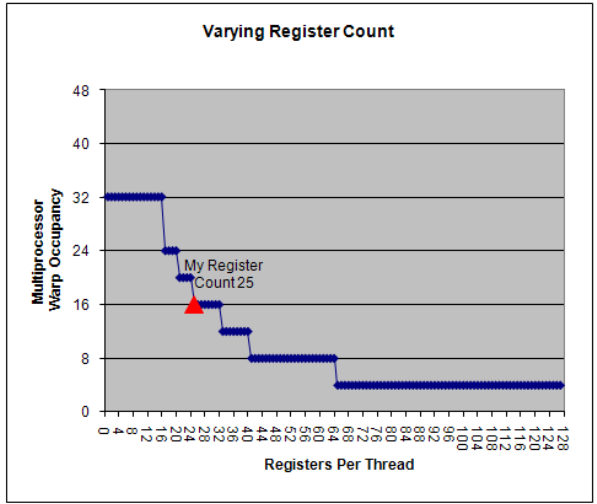
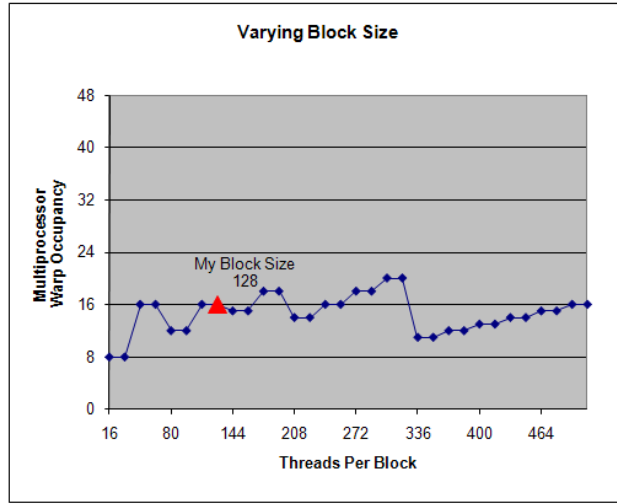
Thread Block Limit Per Multiprocessor highlighted RED

CUDA Occupancy Calculator

Version:	2.0
----------	-----

[Copyright and License](#)

The other data points represent the range of possible block sizes, register counts, and shared memory allocation.



A

B

CUDA GPU Occupancy Calcula

1

2

3

4

Just follow steps 1, 2, and 3 below! (or click here for help)

5

6

1.) Select Compute Capability (click):

1.3

7

8

2.) Enter your resource usage:

9

Threads Per Block

128

10

Registers Per Thread

25

11

Shared Memory Per Block (bytes)

640

12

13

(Don't edit anything below this line)

14

15

3.) GPU Occupancy Data is displayed here and in the graphs:



Calulator

Help

GPU Data

Copyright

Ready



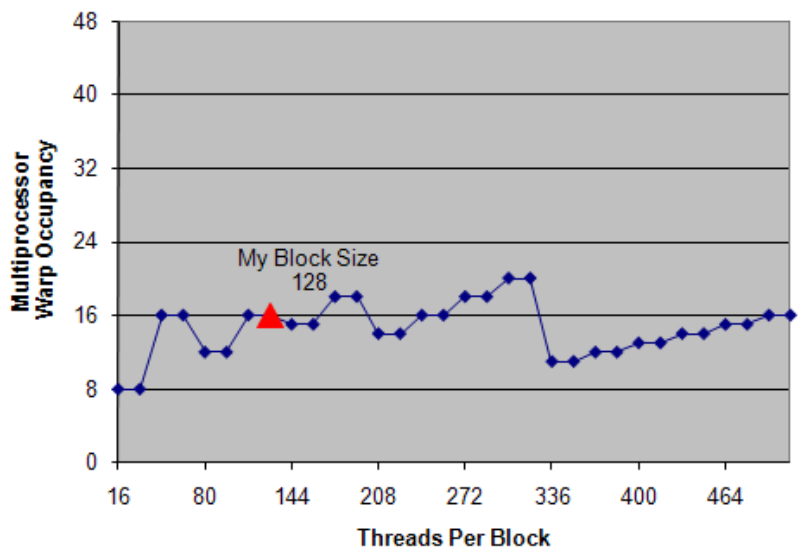
100%



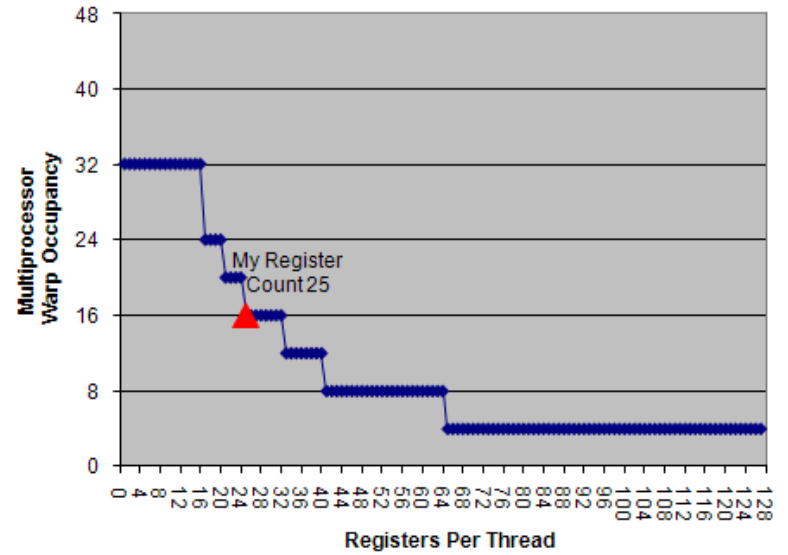
	A	B
14		
15	3.) GPU Occupancy Data is displayed here and in the graphs:	
16	Active Threads per Multiprocessor	512
17	Active Warps per Multiprocessor	16
18	Active Thread Blocks per Multiprocessor	4
19	Occupancy of each Multiprocessor	50%
20		
21		
22	Physical Limits for GPU Compute Capability:	1.3
23	Threads per Warp	32
24	Warps per Multiprocessor	32
25	Threads per Multiprocessor	1024
26	Thread Blocks per Multiprocessor	8
27	Total # of 32-bit registers per Multiprocessor	16384
28	Register allocation unit size	512
29	Register allocation granularity	block
30	Shared Memory per Multiprocessor (bytes)	16384
31	Shared Memory Allocation unit size	512
32	Warp allocation granularity (for register allocation)	2
33		
34	Allocation Per Thread Block	
35	Warps	4
36	Registers	3584
37	Shared Memory	1024
38	These data are used in computing the occupancy data in blue	
39		
40	Maximum Thread Blocks Per Multiprocessor	Blocks
41	Limited by Max Warps / Blocks per Multiprocessor	8
42	Limited by Registers per Multiprocessor	4
43	Limited by Shared Memory per Multiprocessor	16
44	Thread Block Limit Per Multiprocessor highlighted	RED

6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50

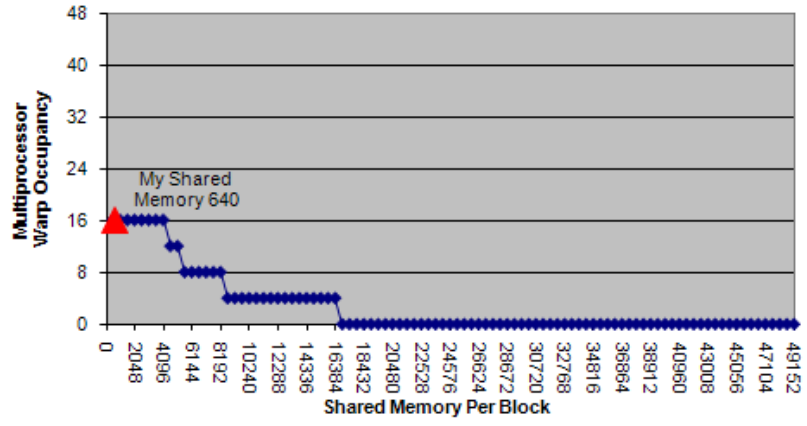
Varying Block Size



Varying Register Count



Varying Shared Memory Usage



How to influence how many registers you use

- Pass option `-maxrregcount=X` to `nvcc`
- This isn't magic, won't get occupancy for free
- Use this very carefully when you are right on the edge

Optimizing Threads per Block

- Choose threads per block as multiple of warp size
 - Avoid wasting computation on under-populated warps
- Run as many warps as possible per SM
 - Hide latency
- SMs can run up to N blocks at a time

Occupancy \neq Performance

- Increasing occupancy does not necessarily increase performance
- BUT...
- Low-occupancy SMs cannot adequately hide latency

Parameterize your Application

- Parameterization helps adaptation to different GPUs
- GPUs vary in many ways
 - # of SMs
 - Memory bandwidth
 - Shared memory size
 - Register file size
 - Max threads per block
- **Avoid local minima**
 - Try widely varying configurations

Kernel Launch Overhead

slides by

Jared Hoberock and David Tarjan
(Stanford CS 193G)

Kernel Launch Overhead

- Kernel launches aren't free
 - A null kernel launch will take non-trivial time
 - Actual time changes with HW generations and driver software...
- Independent kernel launches are cheaper than dependent kernel launches
 - Dependent launch: Some readback to the CPU
- Launching lots of small grids comes with substantial performance loss

Kernel Launch Overheads

- If you are reading back data to the CPU for control decisions, consider doing it on the GPU
- Even though the GPU is slow at serial tasks, it can do surprising amounts of work before you used up kernel launch overhead

Instruction Performance

slides by

Joseph T. Kider Jr. (Upenn)

Instruction Performance

- Instruction cycles (per warp) is the sum of
 - Operand read cycles
 - Instruction execution cycles
 - Result update cycles
- Therefore instruction throughput depends on
 - Nominal instruction throughput
 - Memory latency
 - Memory bandwidth
- Cycle refers to the multiprocessor clock rate

Maximizing Instruction Throughput

- Maximize use of high-bandwidth memory
 - Maximize use of shared memory
 - Minimize accesses to global memory
 - Maximize coalescing of global memory accesses
- Optimize performance by overlapping memory accesses with computation
 - High arithmetic intensity programs
 - Many concurrent threads

Arithmetic Instruction Throughput

- **int** and **float** add, shift, min, max and **float** mul, mad: 4 cycles per warp
 - int multiply is by default 32-bit
 - requires multiple cycles/warp
 - use `__mul24()` and `__umul24()` intrinsics for 4-cycle 24-bit int multiplication
- Integer division and modulo operations are costly
 - The compiler will convert literal power-of-2 divides to shifts
 - But it may miss
 - Be explicit in cases where the compiler cannot tell that the divisor is a power of 2
 - Trick: `foo % n == foo & (n-1)` if `n` is a power of 2

Loop Transformations

Mary Hall
CS6963 University of Utah

Reordering Transformations

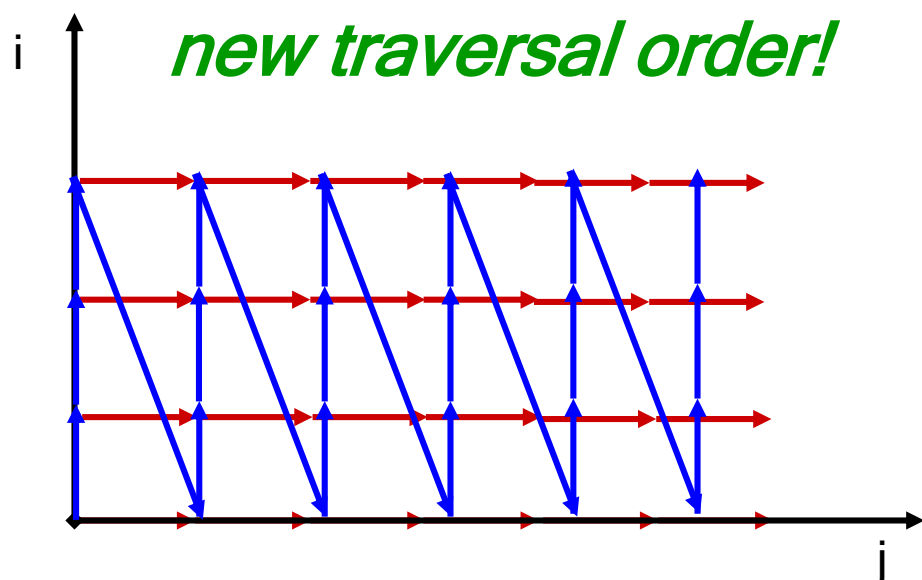
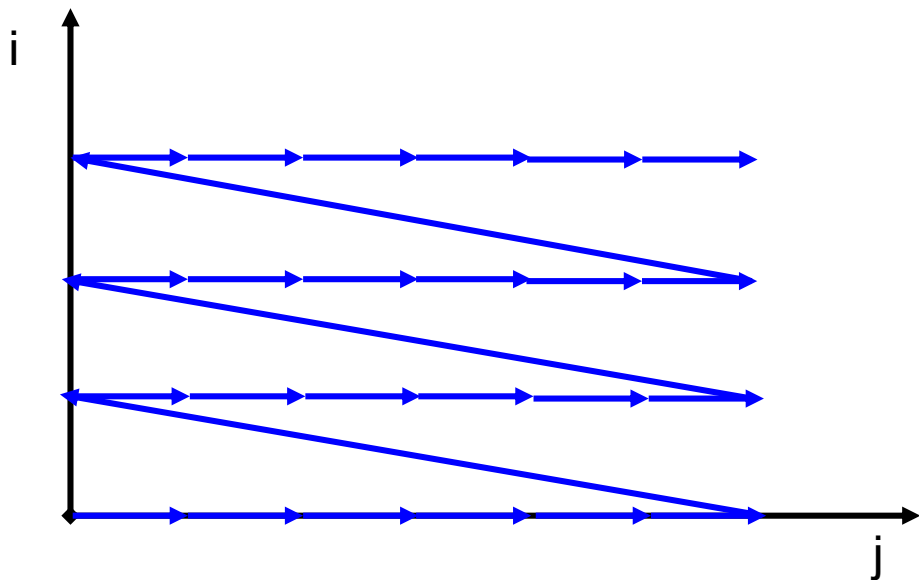
- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
 - **Safety?** (doesn't reverse dependences)
 - **Profitability?** (improves locality)

Loop Permutation: A Reordering Transformation

Permute the order of the loops to modify the traversal order

```
for (i= 0; i<3; i++)  
  for (j=0; j<6; j++)  
    A[i][j+1]=A[i][j]+B[j];
```

```
for (j=0; j<6; j++)  
  for (i= 0; i<3; i++)  
    A[i][j+1]=A[i][j]+B[j];
```



Which one is better for row-major storage?

Safety of Permutation

- **Intuition:** Cannot permute two loops i and j in a loop nest if doing so reverses the direction of any dependence.

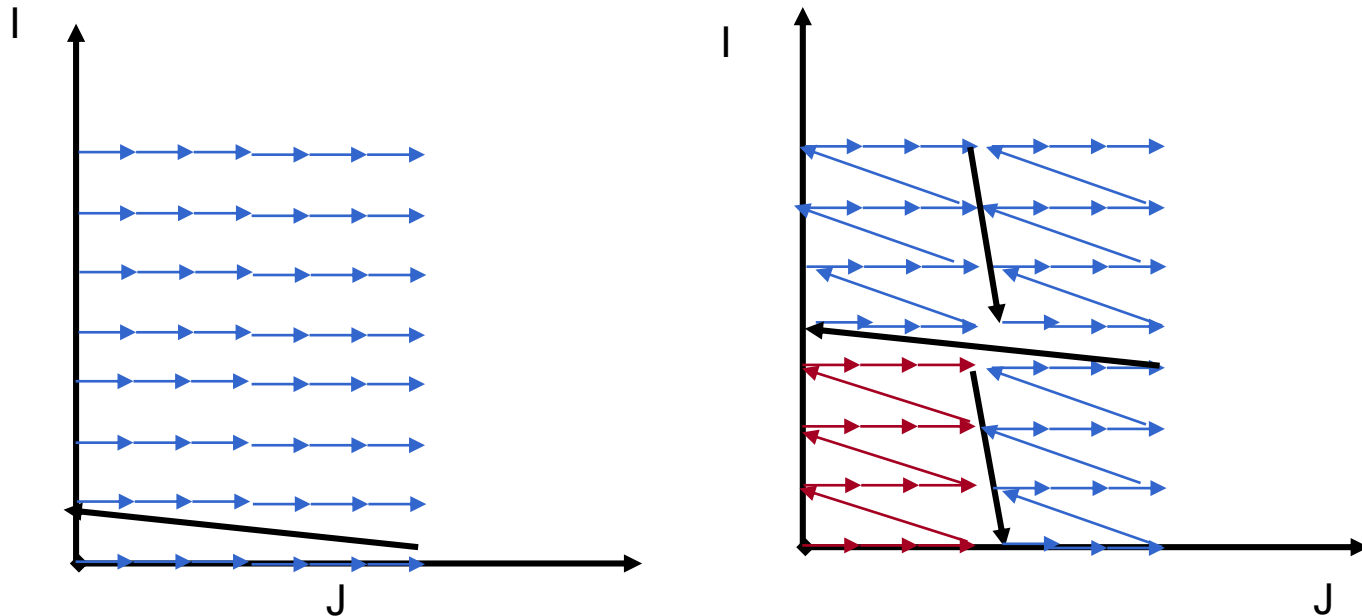
```
for (i= 0; i<3; i++)  
  for (j=0; j<6; j++)  
    A[i][j+1]=A[i][j]+B[j];
```

```
for (i= 0; i<3; i++)  
  for (j=1; j<6; j++)  
    A[i+1][j-1]=A[i][j]+B[j];
```

- Ok to permute?

Tiling (Blocking): Another Loop Reordering Transformation

- Blocking reorders loop iterations to bring iterations that reuse data closer in time



Tiling Example

```
for (j=1; j<M; j++)  
  for (i=1; i<N; i++)  
    D[i] = D[i] + B[j][i];
```

Strip
mine

```
for (j=1; j<M; j++)  
  for (ii=1; ii<N; ii+=s)  
    for (i=ii; i<min(ii+s,N); i++)  
      D[i] = D[i] + B[j][i];
```

Permute

```
for (ii=1; ii<N; ii+=s)  
  for (j=1; j<M; j++)  
    for (i=ii; i<min(ii+s,N); i++)  
      D[i] = D[i] + B[j][i];
```

Legality of Tiling

- Tiling = strip-mine and permutation
 - Strip-mine does not reorder iterations
 - Permutation must be legal
- OR
- strip size less than dependence distance

A Few Words On Tiling

- Tiling can be used hierarchically to compute partial results on a block of data wherever there are capacity limitations
 - Between grids if total data exceeds global memory capacity
 - Across thread blocks if shared data exceeds shared memory capacity (also to partition computation across blocks and threads)
 - Within threads if data in constant cache exceeds cache capacity or data in registers exceeds register capacity or (as in example) data in shared memory for block still exceeds shared memory capacity

Summary of Performance Considerations

Summary of Performance Considerations

- Thread Execution and Divergence
- Communication Through Memory
- Instruction Level Parallelism and Thread Level Parallelism
- Memory Coalescing
- Shared Memory Bank Conflicts
- Parallel Reduction
- Prefetching
- Loop Unrolling and Transformations
- Occupancy
- Kernel Launch Overhead
- Instruction Performance

Thread Execution and Divergence

- Instructions are issued per 32 threads (warp)
- Divergent branches:
 - Threads within a single warp take different paths
 - if-else, ...
 - Different execution paths within a warp are serialized
- Different warps can execute different code with no impact on performance

An Example

```
// is this barrier divergent?  
for (int offset = blockDim.x / 2;  
     offset > 0;  
     offset >>= 1)  
{  
    ...  
    __syncthreads ();  
}
```


A Second Example

```
// what about this one?
__global__ void do_i_halt(int *input)
{
    int i = ...
    if(input[i])
    {
        ...
        __syncthreads() ; // a divergent barrier
        // hangs the machine
    }
}
```

Communication Through Memory

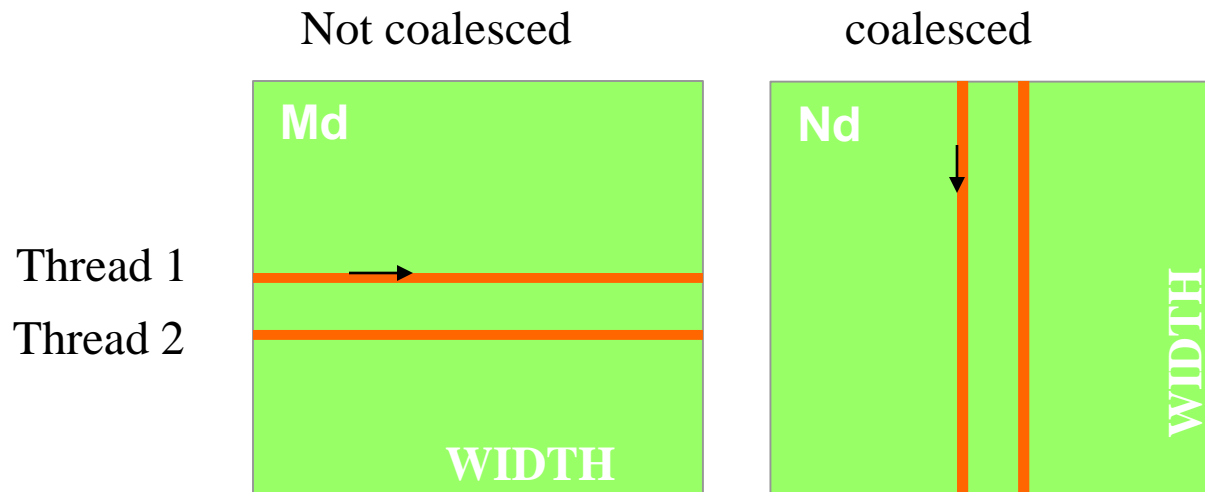
- Carefully partition data according to access patterns
- Read-only → constant memory (fast)
- R/W & shared within block → shared memory (fast)
- R/W within each thread → registers (fast)
- Indexed R/W within each thread → local memory (slow)
- R/W inputs/results → `cudaMalloc`'ed global memory (slow)

Instruction Level Parallelism and Thread Level Parallelism

- Dynamic partitioning 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**

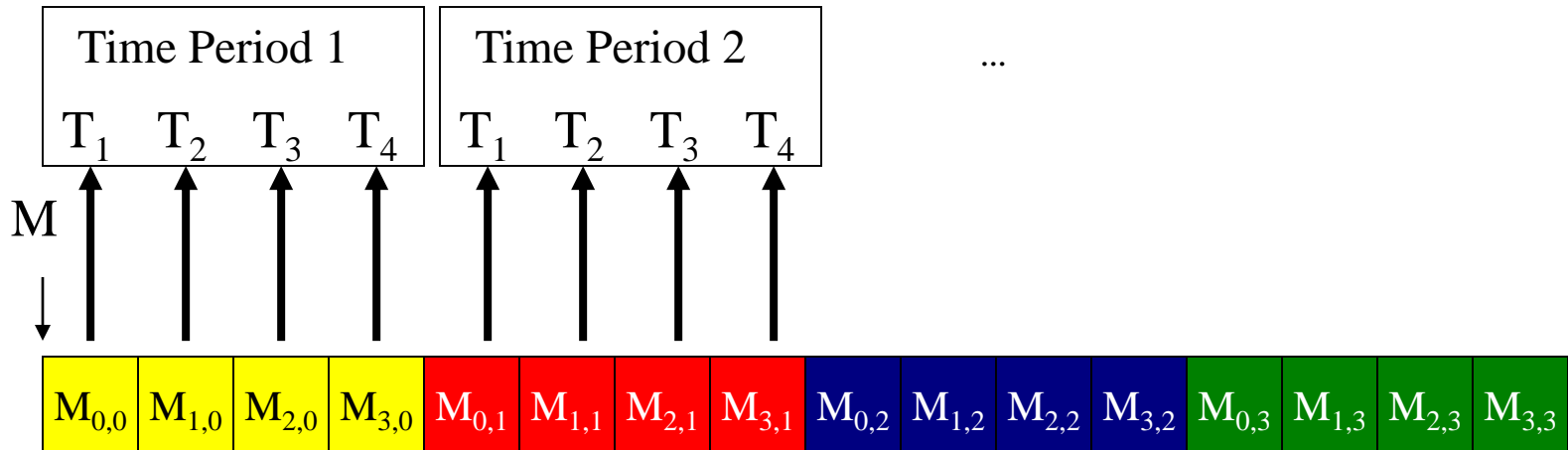
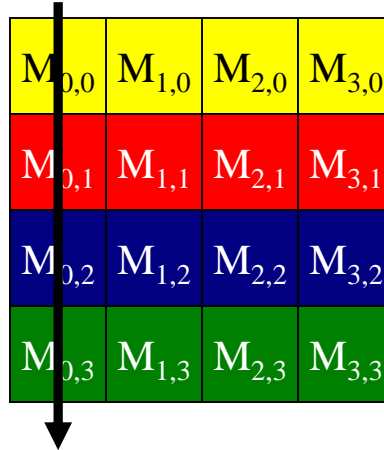
Memory Coalescing

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

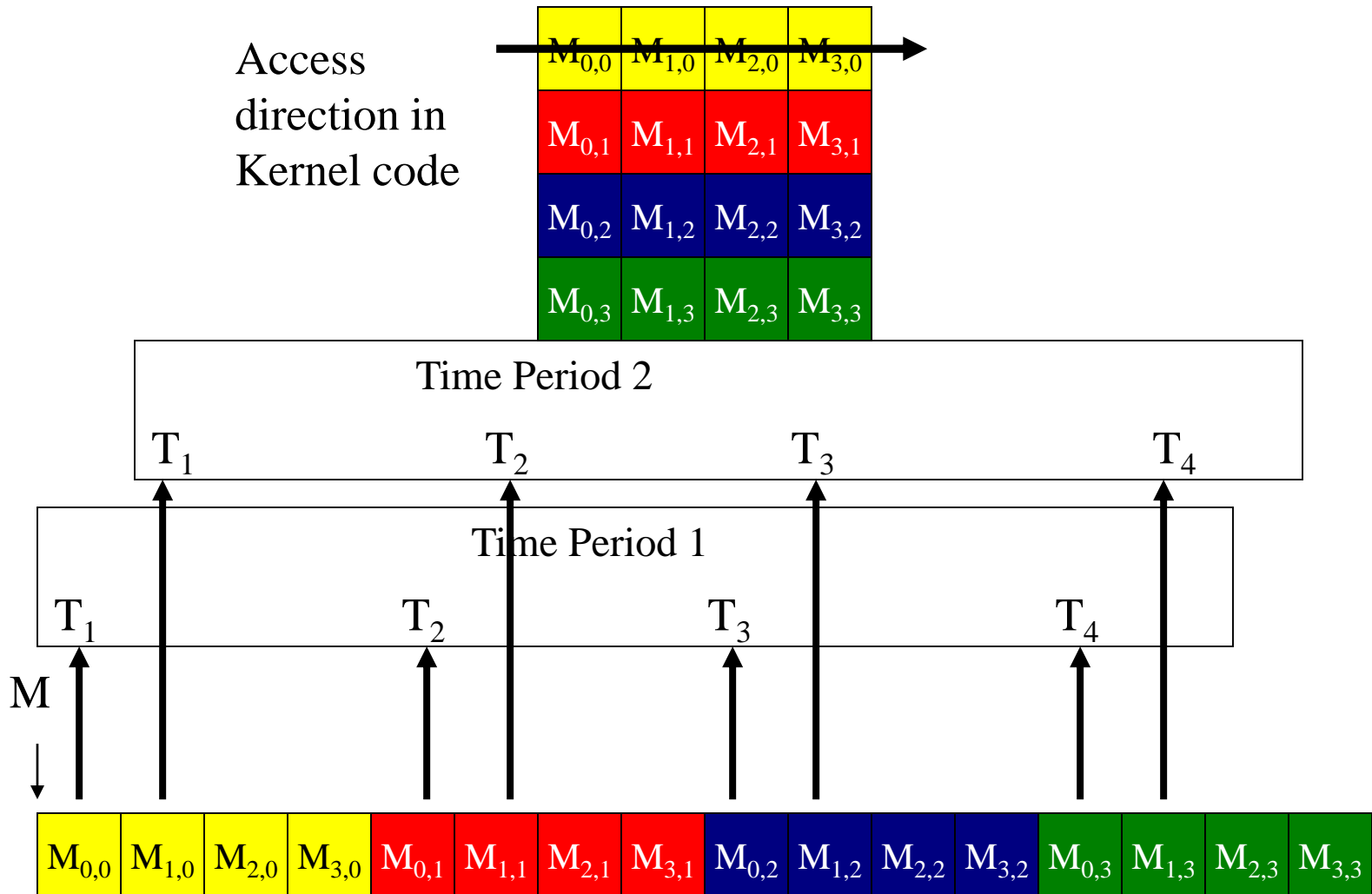


Memory Layout of a Matrix in C

Access
direction in
Kernel code



Memory Layout of a Matrix in C



Example: SoA vs. AoS

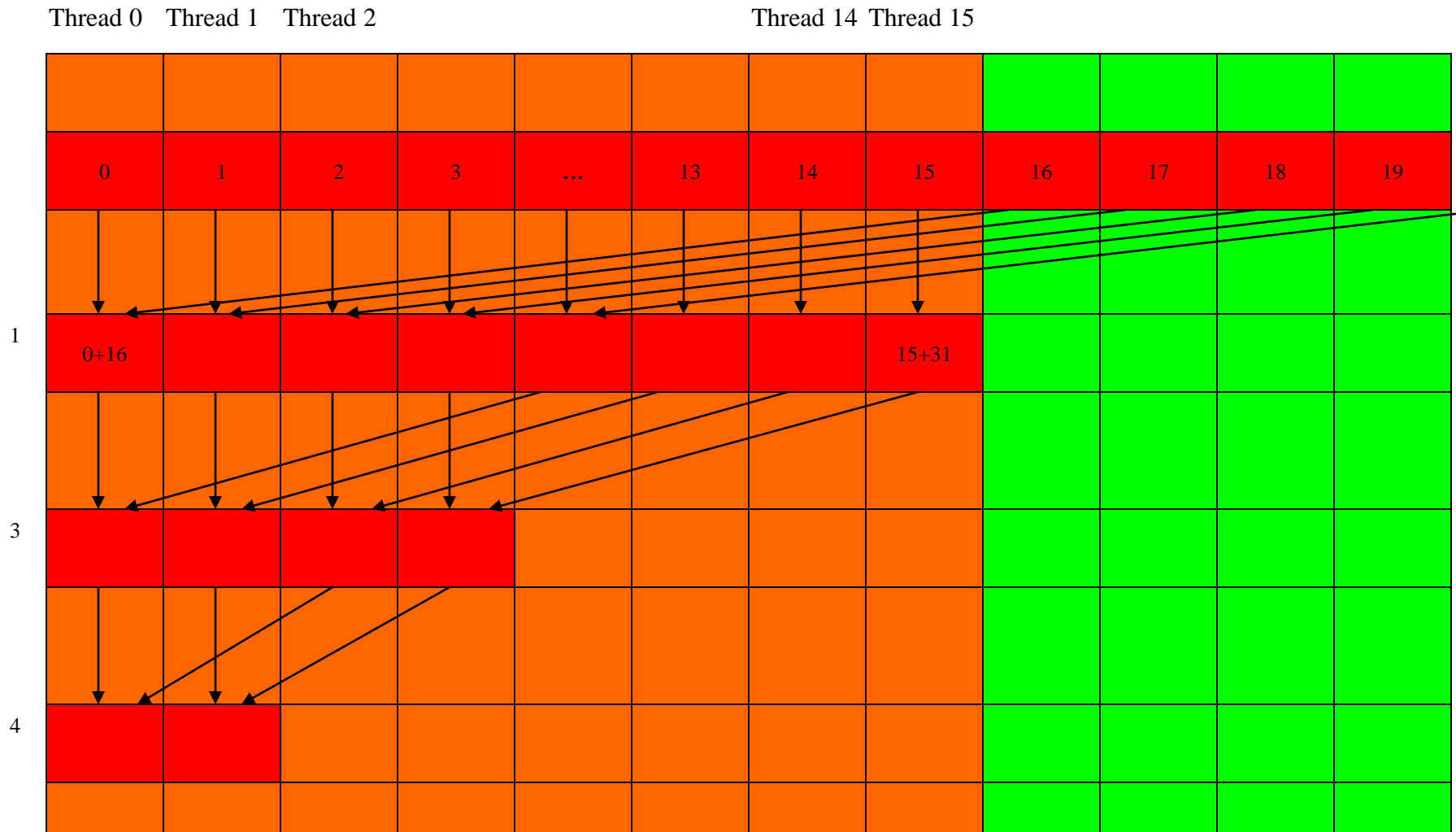
```
__global__ void bar(record
*AoS_data, SoA SoA_data)
{
    int i = blockDim.x * blockIdx.x
          + threadIdx.x;
    // AoS wastes bandwidth
    int key = AoS_data[i].key;
    // SoA efficient use of bandwidth
    int key_better =
    SoA_data.keys[i];
}
```

Shared Memory Bank Conflicts

- Shared memory is as fast as registers **if there are no bank conflicts**
- Bank conflicts are less of an issue in newer versions of CUDA

Parallel Reduction:

No Divergence until ≤ 16 sub-sums



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()  
}
```

Instruction Mix Considerations: Loop Unrolling

```
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. (Beware that any local arrays used after unrolling will be dumped into Local Memory)

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

Occupancy

- Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep memory busy
- **Occupancy** = number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently
- Limited by resource usage:
 - Registers
 - Shared memory

Optimizing Threads per Block

- Choose threads per block as multiple of warp size
 - Avoid wasting computation on under-populated warps
- Run as many warps as possible per SM
 - Hide latency
- SMs can run up to N blocks at a time

Kernel Launch Overhead

- Kernel launches aren't free
 - A null kernel launch will take non-trivial time
 - Actual time changes with HW generations and driver software...
- Independent kernel launches are cheaper than dependent kernel launches
 - Dependent launch: Some readback to the cpu
- Launching lots of small grids comes with substantial performance loss

Compute Capabilities

- Reminder: do not take various constants, such as size of shared memory etc., for granted since they continuously change
- Check CUDA programming guide for the features of the compute capability of your GPU

Timers

- Any timer can be used
 - Check resolution
- **Important:** many CUDA API functions are asynchronous
 - They return control back to the calling CPU thread prior to completing their work
 - All kernel launches are asynchronous
 - So are all memory copy functions with the `Async` suffix on the name

Synchronization

- Synchronize the CPU thread with the GPU by calling `cudaThreadSynchronize()` immediately before starting and stopping the CPU timer
- `cudaThreadSynchronize()` blocks the calling CPU thread until all CUDA calls previously issued by the thread are completed

Synchronization

- `cudaEventSynchronize()` blocks until a given event in a particular stream has been recorded by the GPU
 - Safe only in the default (0) stream
 - Fine for our purposes

CUDA Timer

```
cudaEvent_t start, stop;
float time;
cudaEventCreate(&start);
cudaEventCreate(&stop);
cudaEventRecord( start, 0 );

kernel<<<grid,threads>>> ( d_odata, d_idata,
    size_x, size_y, NUM_REPS);

cudaEventRecord( stop, 0 );
cudaEventSynchronize( stop ); // after cudaEventRecord
cudaEventElapsedTime( &time, start, stop );
cudaEventDestroy( start );
cudaEventDestroy( stop );
```

Output

- `time` is in milliseconds
- Its resolution of approximately half a microsecond
- The timings are measured on the GPU clock
 - Operating system-independent