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

- Midterm: March 22
- Project proposal presentations: March 8
  - Have to be approved by me by March 3

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

Overview

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

- Floating-Point Considerations

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

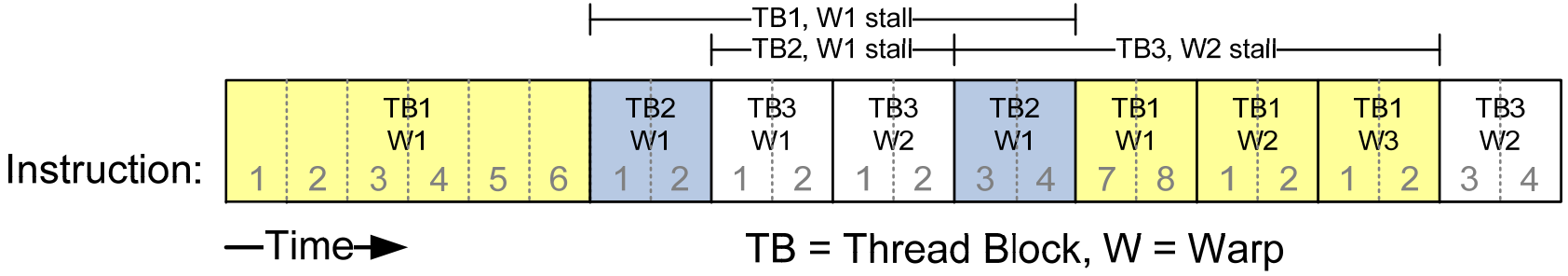
- Structure of arrays is often better than array of structures
  - Very clear win on regular, stride 1 access patterns
  - Unpredictable or irregular access patterns are case-by-case

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



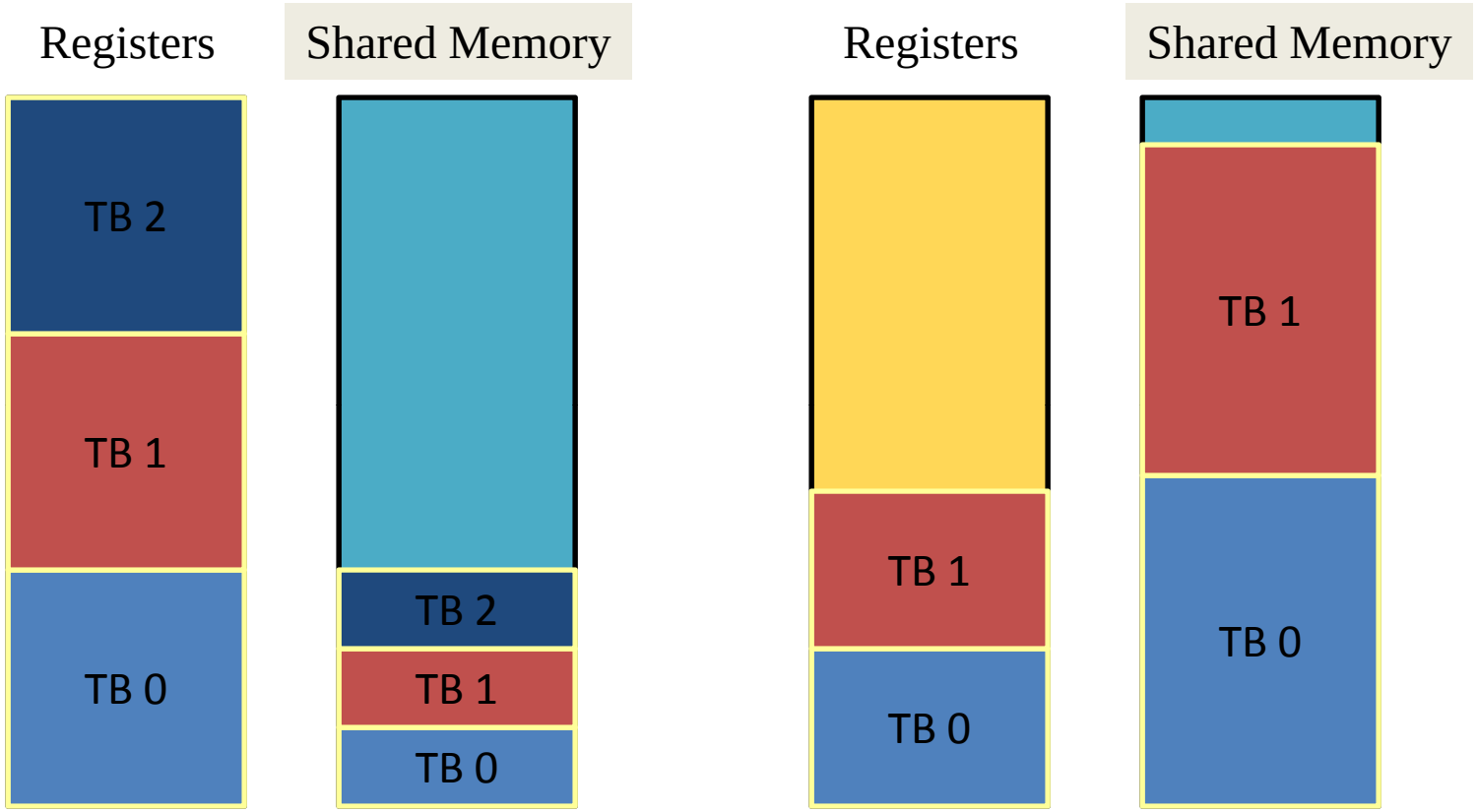
Thread Scheduling

- 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

CUDA_Occupancy_calculator.xlsm - Microsoft Excel

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:			
Active Threads per Multiprocessor	512	(Help)	
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

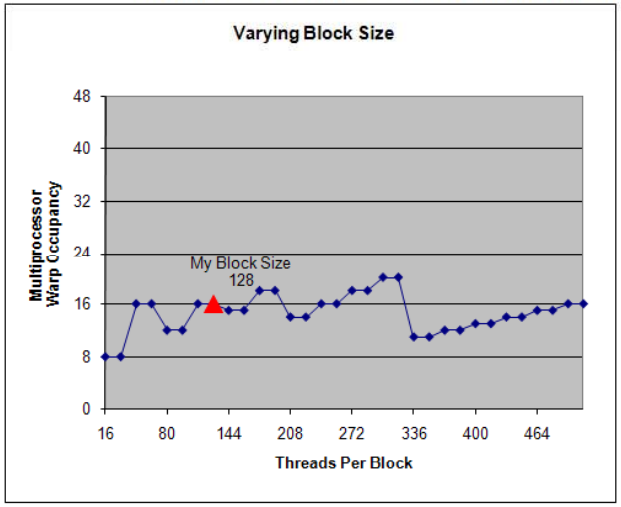
These data are used in computing the occupancy data in blue

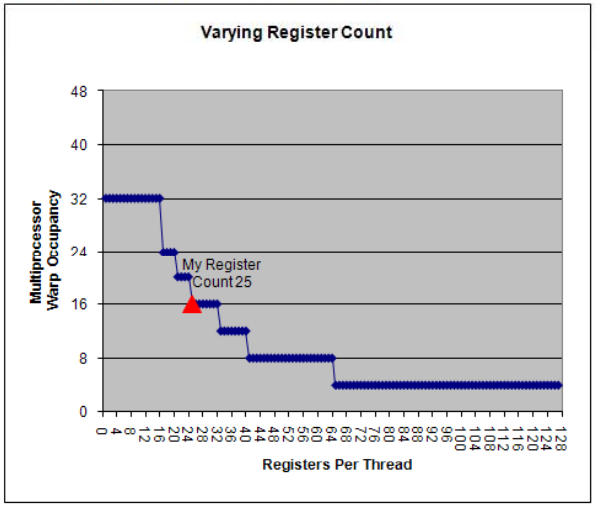
Maximum Thread Blocks Per Multiprocessor	Blocks
Limited by Max Warps / Blocks per Multiprocessor	8
Limited by Registers per Multiprocessor	4
Limited by Shared Memory per Multiprocessor	16

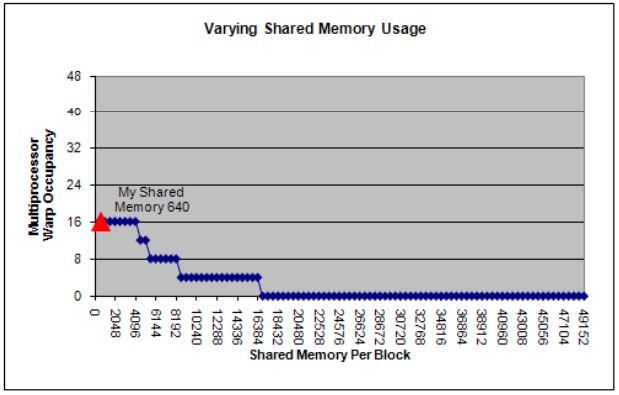
Thread Block Limit Per Multiprocessor highlighted RED

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.

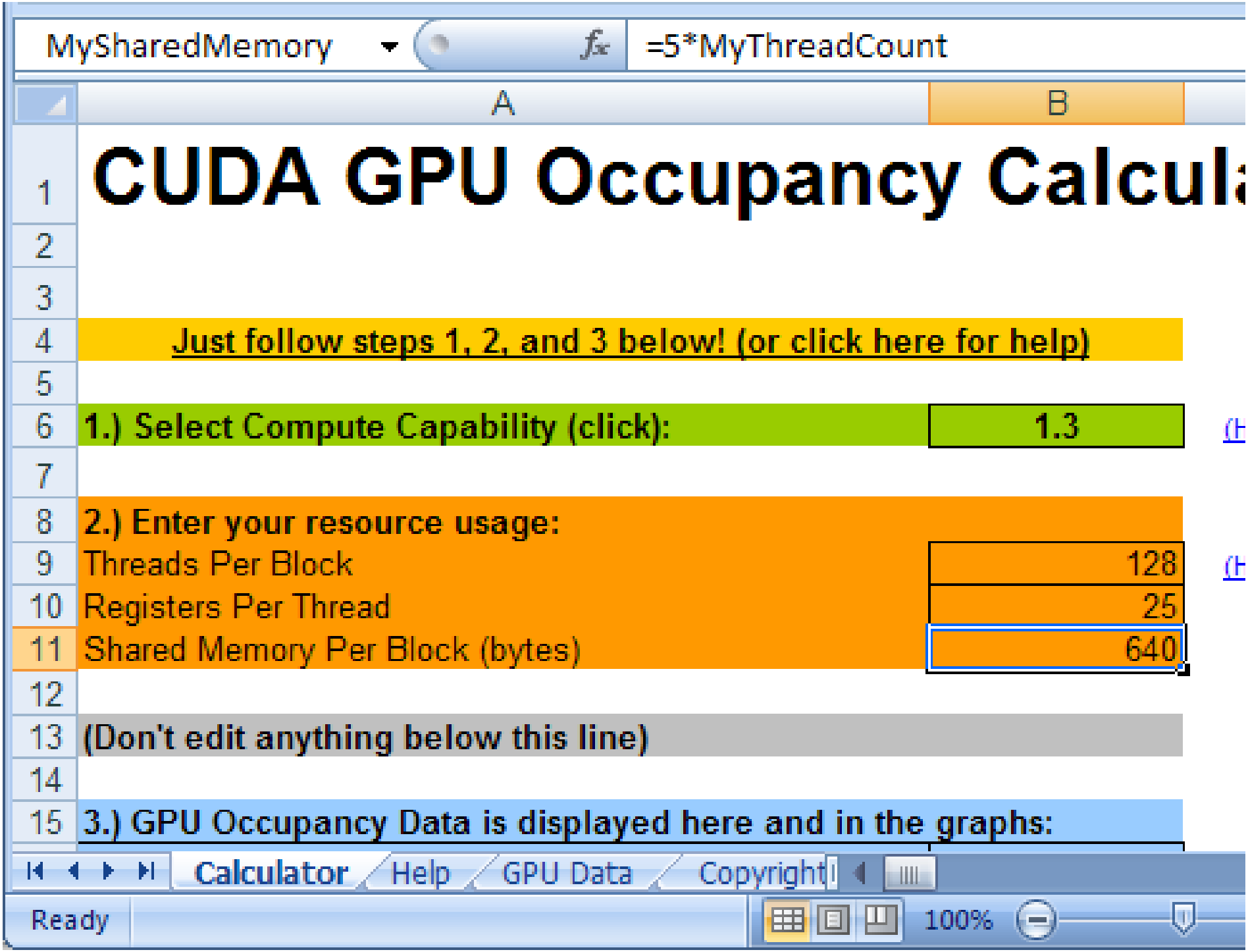


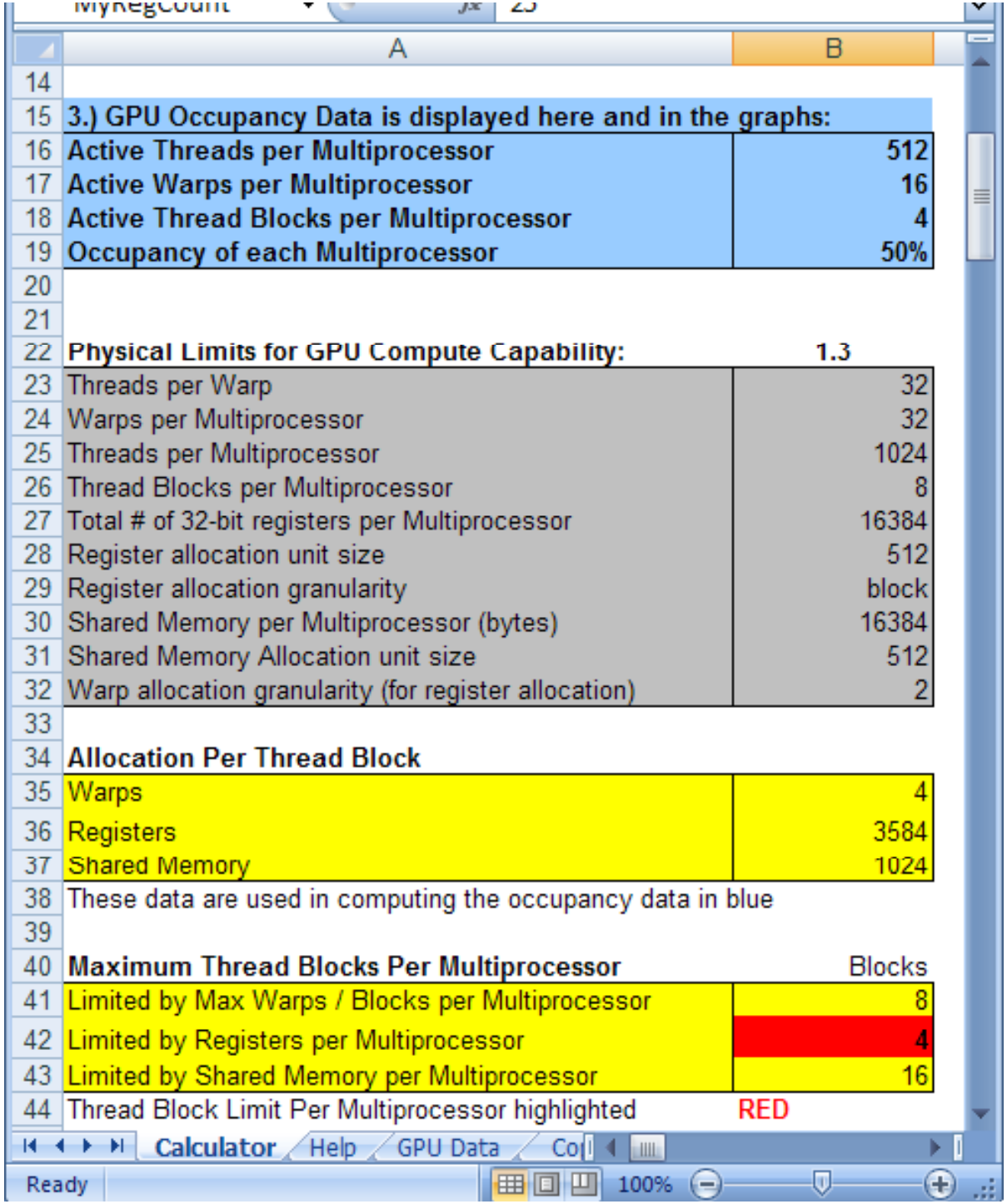




Sheets: Calculator | Help | GPU Data | Copyright & License

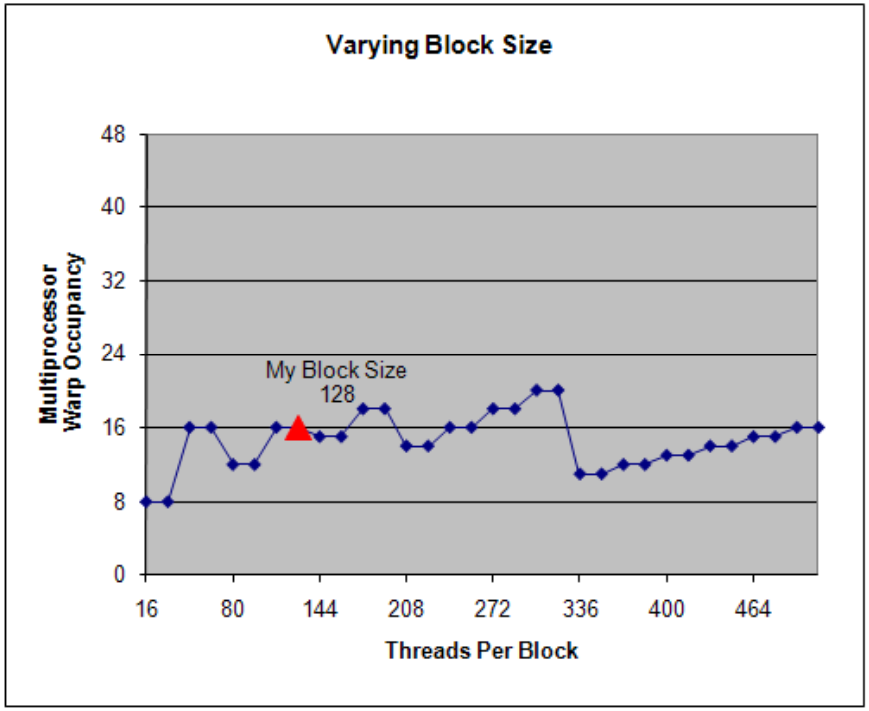
Ready 100%

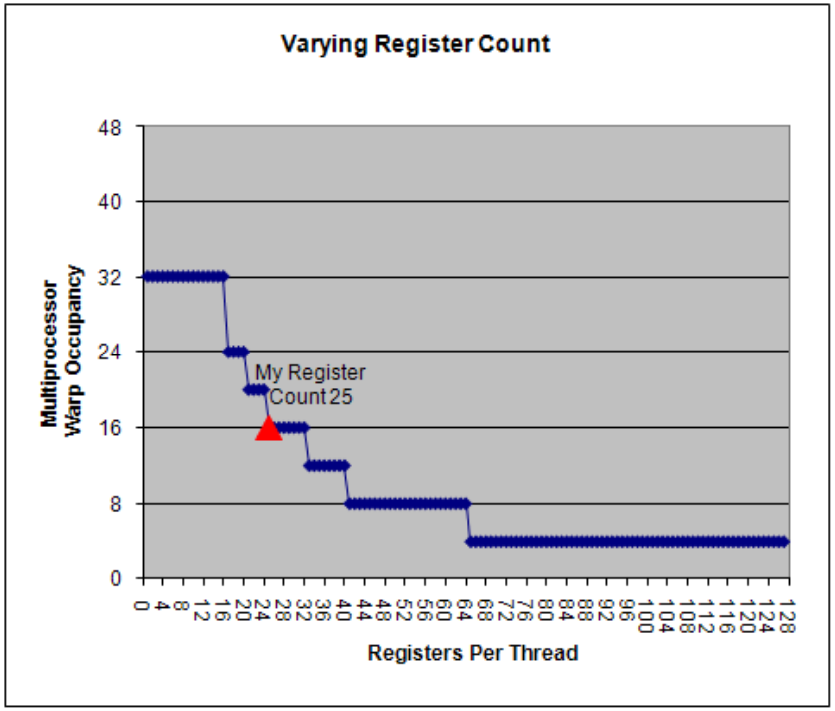

MySharedMemory
=5*MyThreadCount
A
B
CUDA GPU Occupancy Calcul
Just follow steps 1, 2, and 3 below! (or click here for help)
1.) Select Compute Capability (click):
1.3
2.) Enter your resource usage:
Threads Per Block
128
Registers Per Thread
25
Shared Memory Per Block (bytes)
640
(Don't edit anything below this line)
3.) GPU Occupancy Data is displayed here and in the graphs:
Calculator
Help
GPU Data
Copyright
Ready
100%

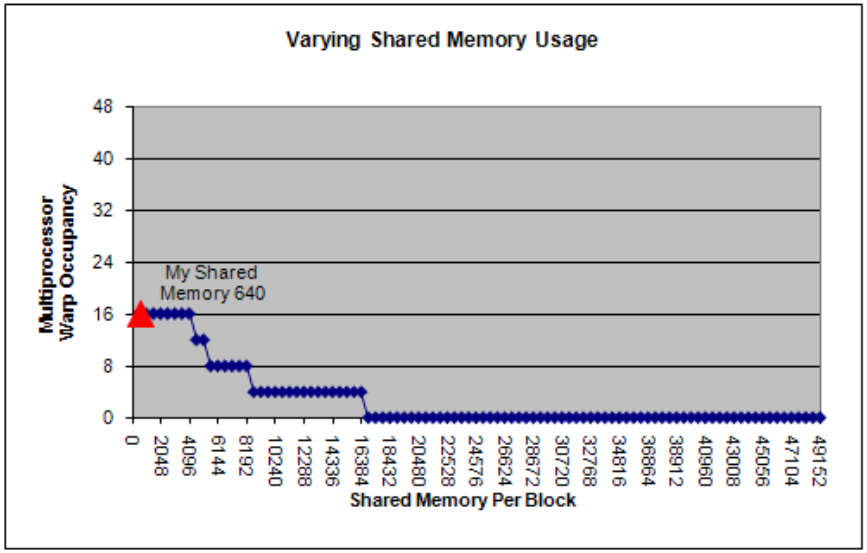


	A	B
14		
15	**3.) GPU Occupancy Data is displayed here and in the graphs:**	
16	**Active Threads per Multiprocessor**	**512**
17	**Active Warps per Multiprocessor**	**16**
18	**Active Thread Blocks per Multiprocessor**	**4**
19	**Occupancy of each Multiprocessor**	**50%**
20		
21		
22	**Physical Limits 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

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







Calculator | Help | GPU Data | Copyright & License

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

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads are a better choice
    - Usually, still enough registers to compile and invoke successfully
  - This all depends on computation

Occupancy != 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 mulitply 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

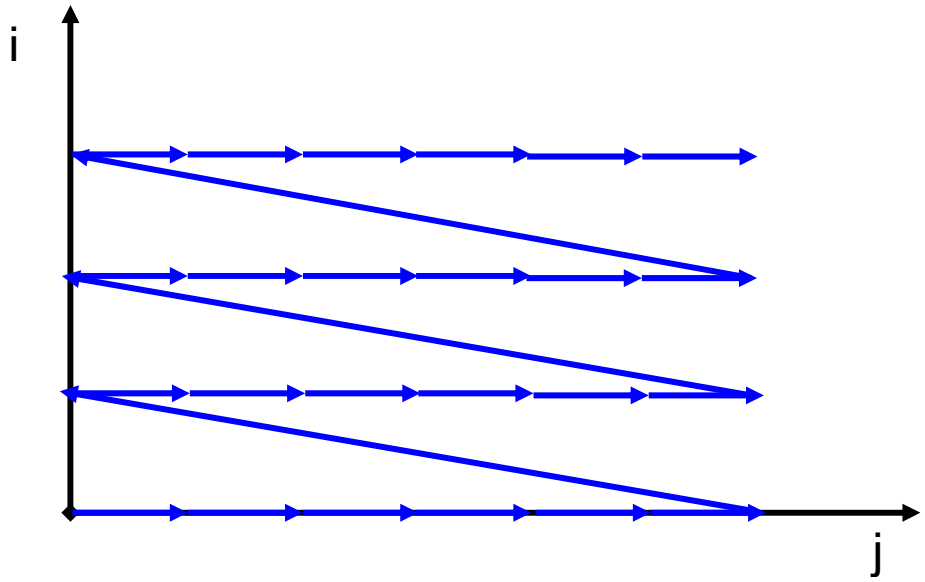
- Analyze reuse in computation
- Apply loop reordering transformations to improve locality based on reuse
- With any loop reordering transformation, always ask
  - **Safety?** (doesn't reverse dependences)
  - **Profitablity?** (improves locality)

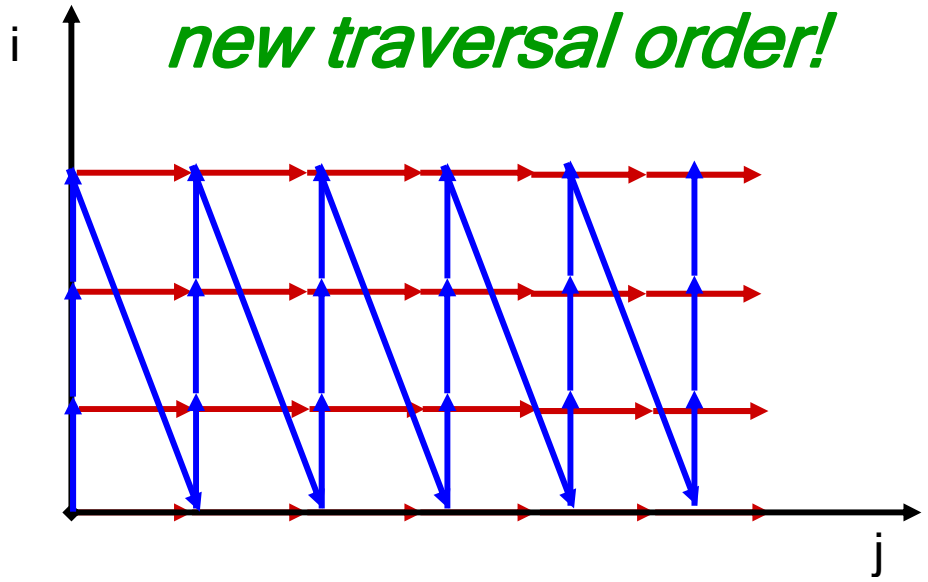
Loop Permutation: A Reordering Transformation

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.

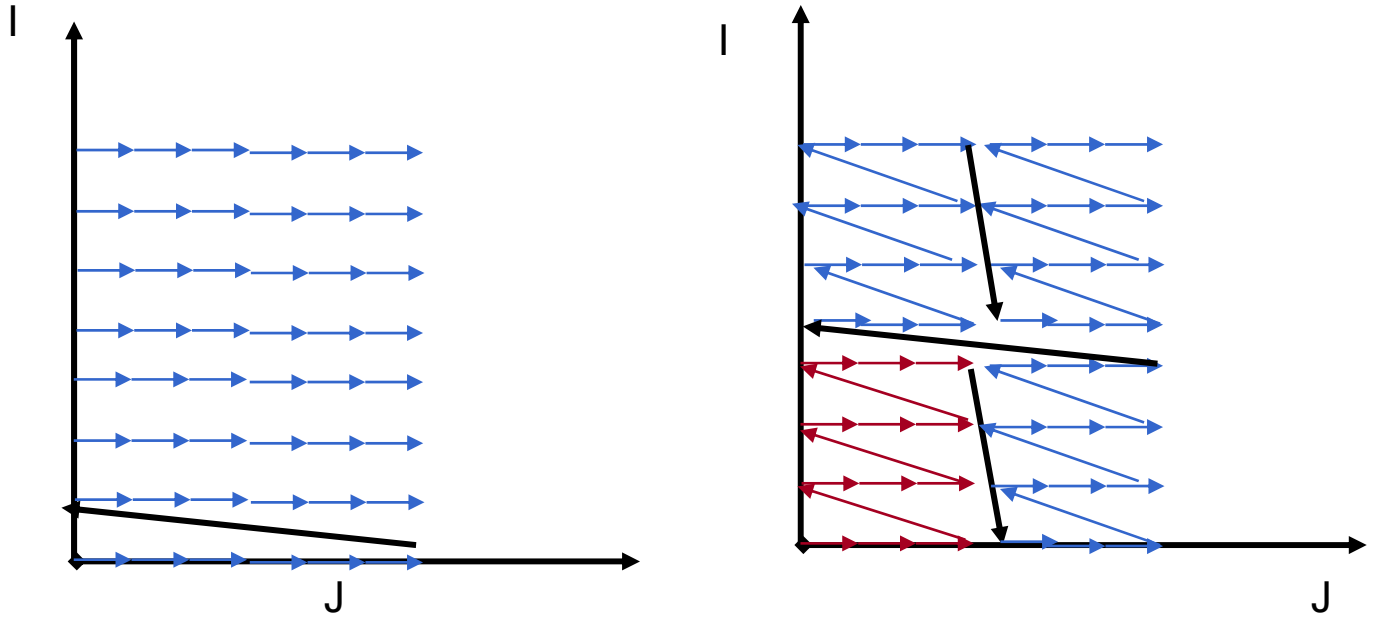
```
for (i= 0; i<3; i++)
   for (j=0; j<6; j++)
      A[i][j+1]=A[i][j]+B[j];
```

```
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)

Communication Through Memory

- Question:

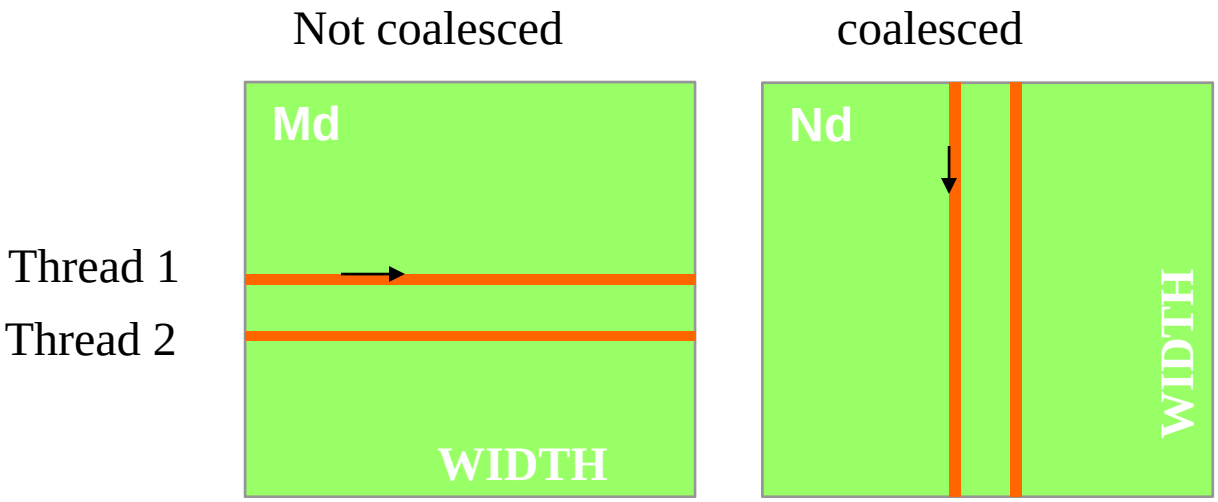
```
__global__ void race(void)
{
  __shared__ int my_shared_variable;
  my_shared_variable = threadIdx.x;

  // what is the value of
  // my_shared_variable?
}
```

Instruction Level Parallelism and Thread Level Parallelism

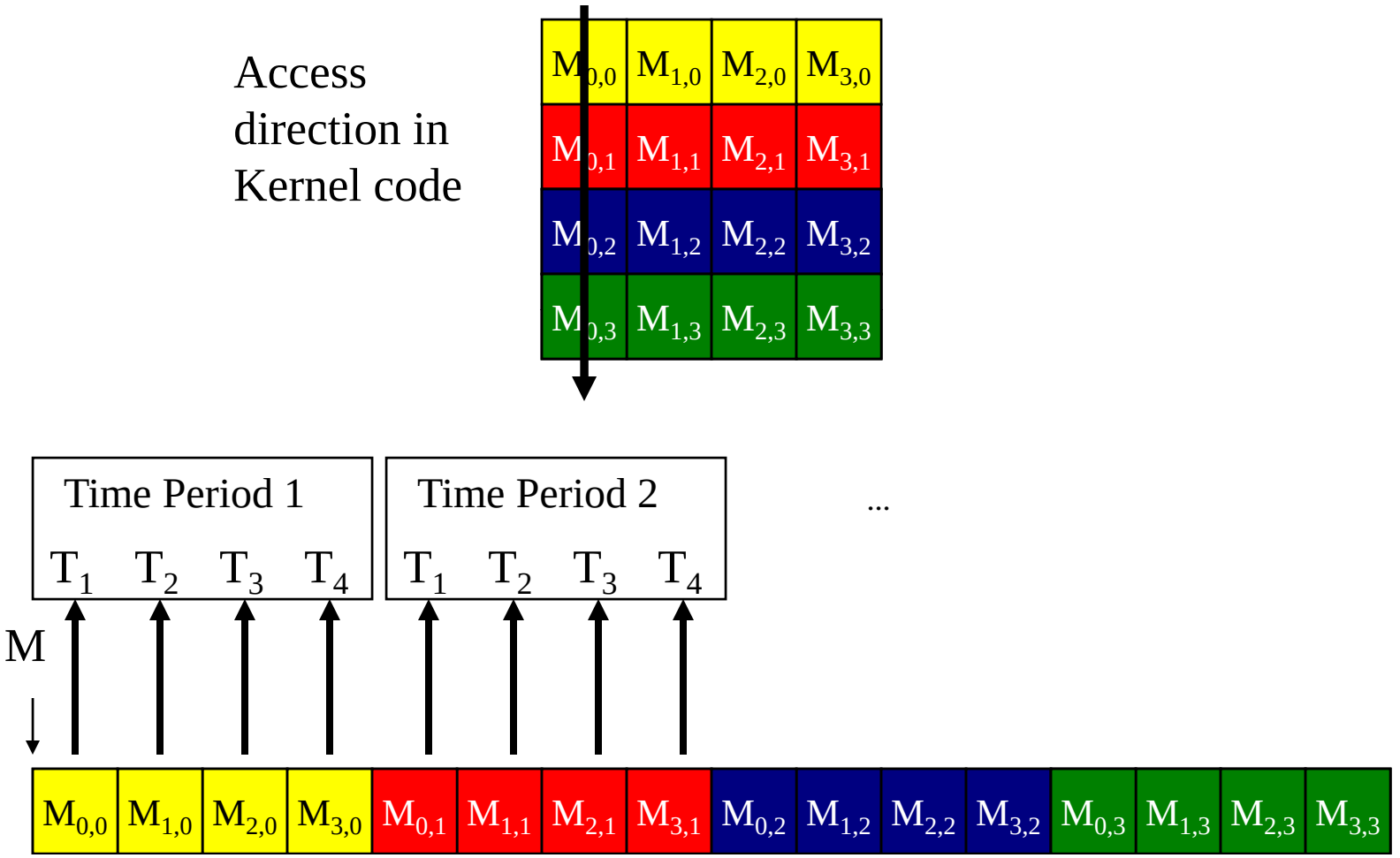
- Dynamic partitioning gives more flexibility to compilers/programmers
  - One can run a smaller number of threads that require many registers each or a large number of threads that require few registers each
    - This allows for finer grain threading than traditional CPU threading models
  - The compiler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

Memory Coalescing

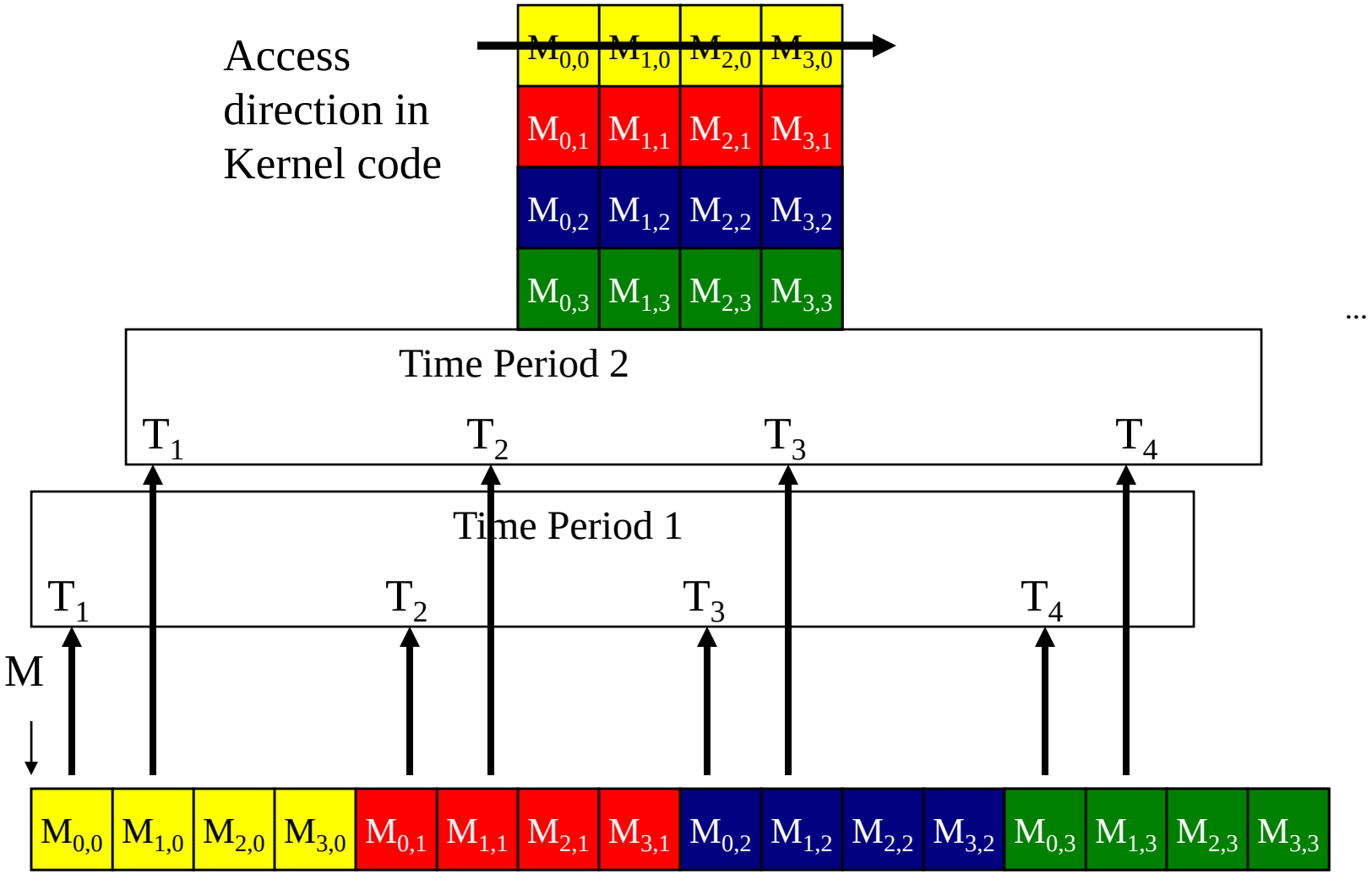
- When accessing global memory, peak performance utilization occurs when all threads in a half warp access continuous memory locations



Memory Layout of a Matrix in C



Memory Layout of a Matrix in C



Example: SoA vs. AoS

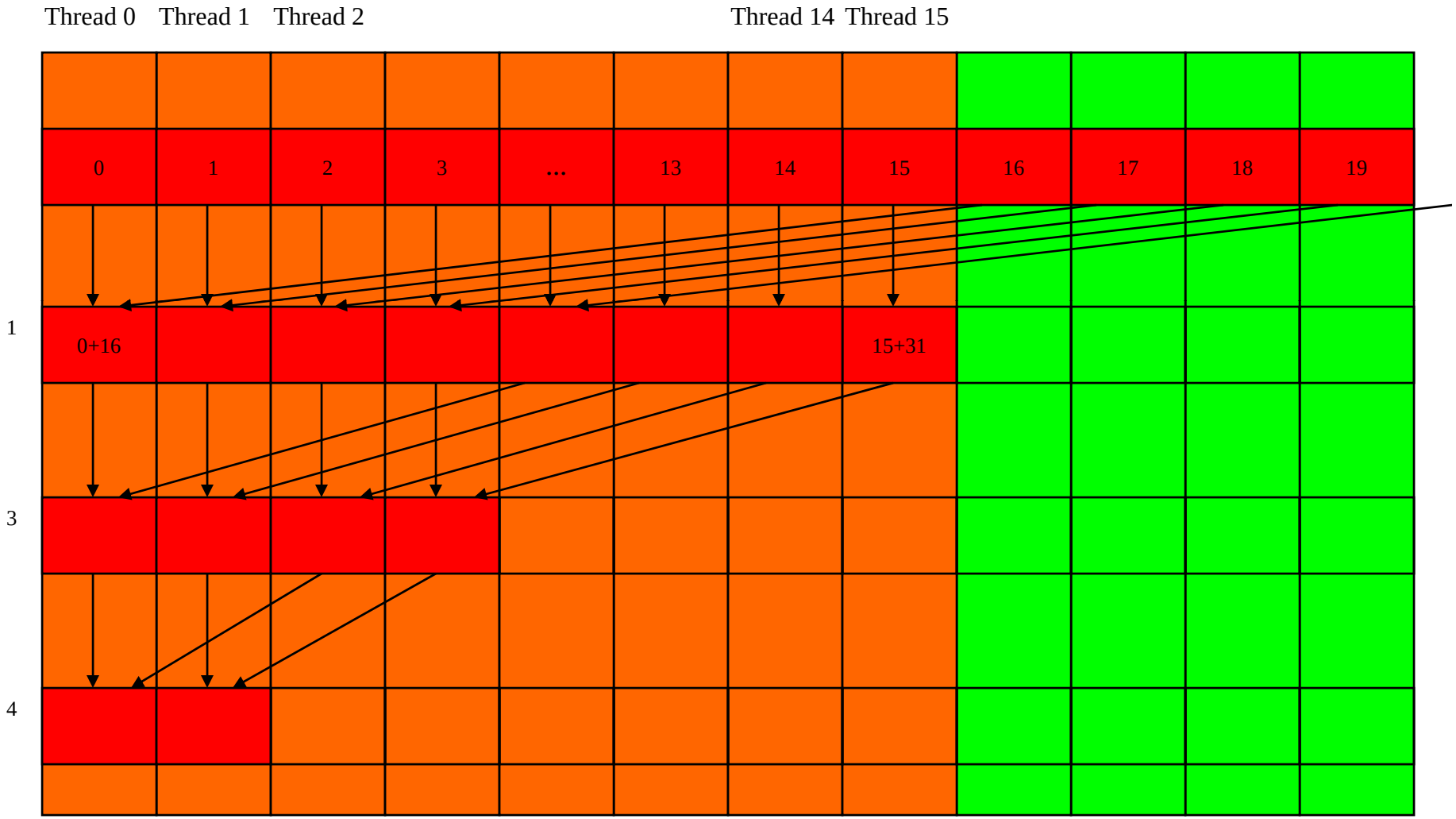
```
__global__ void bar(record
  *AoS_data, SoA SoA_data)
{
   int i = blockDim.x * blockIdx.x
         + threadIdx.x;
  // AoS wastes bandwidth
   int key = AoS_data[i].key;
  // SoA efficient use of bandwidth
  int key_better =
  SoA_data.keys[i];
}
```

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

- Heuristics
  - Minimum: 64 threads per block
    - Only if multiple concurrent blocks
  - 192 or 256 threads are a better choice
    - Usually, still enough registers to compile and invoke successfully
  - This all depends on computation

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

Floating-Point Considerations

Objective

- To understand the fundamentals of floating-point representation
- To know the IEEE-754 Floating Point Standard
- GeForce 8800 CUDA Floating-point speed, accuracy and precision
  - Deviations from IEEE-754
  - Accuracy of device runtime functions
  - -fastmath compiler option
  - Future performance considerations

What is IEEE floating-point format?

- A floating point binary number consists of three parts:
  - sign (S), exponent (E), and mantissa (M).
  - Each (S, E, M) pattern uniquely identifies a floating point number.

- For each bit pattern, its IEEE floating-point value is derived as:

  - value = $(-1)^S * M * \{2^E\}$, where $1.0 \leq M < 10.0_B$

- The interpretation of S is simple: S=0 results in a positive number and S=1 a negative number.

Normalized Representation

- Specifying that $1.0_B \leq M < 10.0_B$ makes the mantissa value for each floating point number unique.
  - For example, the only one mantissa value allowed for 0.5_D is M =1.0
    - $0.5_D = 1.0_B * 2^{-1}$
  - Neither $10.0_B * 2^{-2}$ nor $0.1_B * 2^0$ qualifies

- Because all mantissa values are of the form 1.XX…, one can omit the "1." part in the representation.
  - The mantissa value of 0.5_D in a 2-bit mantissa is 00, which is derived by omitting "1." from 1.00.

Note: Two's Complement

- The negative value of a number can be derived by:
  - Complementing every bit
  - Adding 1

- Example: -3
  - $3 = 011_B$
  - Complement every bit: 100
  - Add 1: 101

Exponent Representation

- In an n-bits exponent representation, $2^{n-1}-1$ is added to its 2's complement representation to form its excess representation.
  - See Table for a 3-bit exponent representation
- A simple unsigned integer comparator can be used to compare the magnitude of two FP numbers
- Symmetric range for +/- exponents (111 reserved)

2's complement	Actual decimal	Excess-3
000	0	011
001	1	100
010	2	101
011	3	110
100	**(reserved pattern)**	**111**
101	-3	000
110	-2	001
111	-1	010

A simple, hypothetical 6-bit FP format

- Assume 1-bit S, 3-bit E, and 2-bit M
  - $0.5_D = 1.00_B * 2^{-1}$
  - 0.5_D = **0 010 00**, where S = 0, E = 010, and M = (1.)00

2's complement	Actual decimal	Excess-3
000	0	011
001	1	100
010	2	101
011	3	110
100	**(reserved pattern)**	**111**
101	-3	000
110	-2	001
111	-1	010

Representable Numbers

- The representable numbers of a given format is the set of all numbers that can be exactly represented in the format.
- See Table for representable numbers of an unsigned 3-bit integer format

000	0
001	1
010	2
011	3
100	4
101	5
110	6
111	7

Representable Numbers of a 5-bit Hypothetical IEEE Format

		No-zero	
E	M	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$
01	00	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern		

Represen of a 5-bit Hypot mat

Cannot represent Zero!

		No-zero	
E	M	S=0	S=1
00	00	2^{-1}	$-(2^{-1})$
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$
01	00	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern		

Representable Numbers of a 5-bit Hypothetical IEEE Format

- Exponent bits define major intervals of representable numbers
- Mantissa bits define the number of representable numbers in each interval
  - With N mantissa bits, 2^N representable numbers per interval
- Representable numbers come closer to each other in the neighborhood of 0
  - Desirable property
- There is a gap around 0
  - Significantly larger error between 0 and 0.5 than 0.5 and 1

Flush to Zero (Abrupt Underflow)

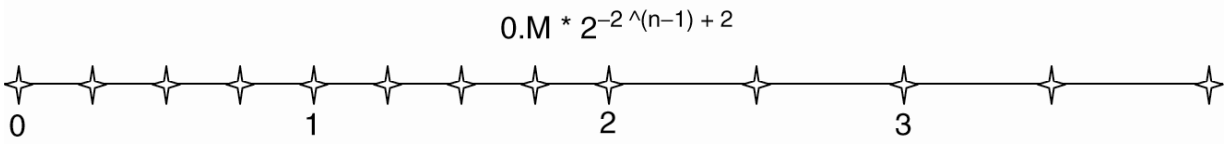
- Treat all bit patterns with E=0 as 0.0
  - This takes away several representable numbers near zero and lumps them all into 0.0
  - For a representation with large M, a large number of representable numbers will be removed

Flush to Zero

		No-zero		**Flush to Zero**	
E	M	S=0	S=1	**S=0**	**S=1**
00	00	2^{-1}	$-(2^{-1})$	**0**	**0**
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	**0**	**0**
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	**0**	**0**
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	**0**	**0**
01	00	2^{0}	$-(2^{0})$	$\mathbf{2^{0}}$	$\mathbf{-(2^{0})}$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$\mathbf{2^{0}+1*2^{-2}}$	$\mathbf{-(2^{0}+1*2^{-2})}$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$\mathbf{2^{0}+2*2^{-2}}$	$\mathbf{-(2^{0}+2*2^{-2})}$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$\mathbf{2^{0}+3*2^{-2}}$	$\mathbf{-(2^{0}+3*2^{-2})}$
10	00	2^{1}	$-(2^{1})$	$\mathbf{2^{1}}$	$\mathbf{-(2^{1})}$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$\mathbf{2^{1}+1*2^{-1}}$	$\mathbf{-(2^{1}+1*2^{-1})}$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$\mathbf{2^{1}+2*2^{-1}}$	$\mathbf{-(2^{1}+2*2^{-1})}$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$\mathbf{2^{1}+3*2^{-1}}$	$\mathbf{-(2^{1}+3*2^{-1})}$
11	Reserved pattern				

Denormalized Numbers

- The actual method adopted by the IEEE standard is called denormalized numbers or gradual underflow.
  - The method relaxes the normalization requirement for numbers very close to 0
  - whenever E=0, the mantissa is no longer assumed to be of the form 1.XX. Rather, it is assumed to be 0.XX. In general, if the n-bit exponent is 0, the value is
    - $0.M * 2^{-2\,\wedge(n-1)\,+\,2}$



Denormalization

		No-zero		Flush to Zero		**Denormalized**	
E	M	S=0	S=1	S=0	S=1	**S=0**	**S=1**
00	00	2^{-1}	$-(2^{-1})$	0	0	**0**	**0**
	01	$2^{-1}+1*2^{-3}$	$-(2^{-1}+1*2^{-3})$	0	0	$\mathbf{1*2^{-2}}$	$\mathbf{-1*2^{-2}}$
	10	$2^{-1}+2*2^{-3}$	$-(2^{-1}+2*2^{-3})$	0	0	$\mathbf{2*2^{-2}}$	$\mathbf{-2*2^{-2}}$
	11	$2^{-1}+3*2^{-3}$	$-(2^{-1}+3*2^{-3})$	0	0	$\mathbf{3*2^{-2}}$	$\mathbf{-3*2^{-2}}$
01	00	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$	2^{0}	$-(2^{0})$
	01	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$	$2^{0}+1*2^{-2}$	$-(2^{0}+1*2^{-2})$
	10	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$	$2^{0}+2*2^{-2}$	$-(2^{0}+2*2^{-2})$
	11	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$	$2^{0}+3*2^{-2}$	$-(2^{0}+3*2^{-2})$
10	00	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$	2^{1}	$-(2^{1})$
	01	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$	$2^{1}+1*2^{-1}$	$-(2^{1}+1*2^{-1})$
	10	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$	$2^{1}+2*2^{-1}$	$-(2^{1}+2*2^{-1})$
	11	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$	$2^{1}+3*2^{-1}$	$-(2^{1}+3*2^{-1})$
11	Reserved pattern						

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() / __umul24() intrinsics for 4-cycle 24-bit int multiply

- Integer divide and modulo are expensive
  - Compiler will convert literal power-of-2 divides to shifts
  - Be explicit in cases where compiler can't tell that divisor is a power of 2!

Arithmetic Instruction Throughput

- Reciprocal, reciprocal square root, sin/cos, log, exp: 16 cycles per warp
  - These are the versions prefixed with "__"
  - Examples:__rcp(), __sin(), __exp()

- Other functions are combinations of the above
  - y / x == rcp(x) * y == 20 cycles per warp
  - sqrt(x) == rcp(rsqrt(x)) == 32 cycles per warp

Runtime Math Library

- There are two types of runtime math operations
  - __func(): direct mapping to hardware ISA
    - Fast but low accuracy (see prog. guide for details)
    - Examples: __sin(x), __exp(x), __pow(x,y)
  - func() : compile to multiple instructions
    - Slower but higher accuracy: error of 5 ulp or less
    - (ulp: units in the last place or units of least precision)
    - Examples: sin(x), exp(x), pow(x,y)

- The -use_fast_math compiler option forces every func() to compile to __func()

Make your program float-safe!

- Hardware now has double precision support
  - G80 is single-precision only
  - Double precision has additional performance cost
  - Careless use of double or undeclared types may run more slowly on G80+
- Important to be float-safe (be explicit whenever you want single precision) to avoid using double precision where it is not needed
  - Add 'f' specifier on float literals:
    - `foo = bar * 0.123;  // double assumed`
    - `foo = bar * 0.123f;  // float explicit`
  - Use float version of standard library functions
    - `foo = sin(bar);  // double assumed`
    - `foo = sinf(bar);  // single precision explicit`

Floating-Point Calculation Results Can Depend on Execution Order

Order 1

$1.00*2^{0} + 1.00*2^{0} + 1.00*2^{-2} + 1.00*2^{-2}$

$= 1.00*2^{1} + 1.00*2^{-2} + 1.00*2^{-2}$

$= 1.00*2^{1} + 1.00*2^{-2}$

$= 1.00*2^{1}$

Order 2

$(1.00*2^{0} + 1.00*2^{0}) + (1.00*2^{-2} + 1.00*2^{-2})$

$= 1.00*2^{1} + 1.00*2^{-1}$

$= 1.01*2^{1}$

Pre-sorting is often used to increase stability of floating point results.

Special Bit Patterns in the IEEE Standard Format

Exponent	Mantissa	Meaning
11 . . . 1	$\neq 0$	NaN
11 . . . 1	$= 0$	$(-1)^S \times \infty$
00 . . . 0	$\neq 0$	Denormalized
00 . . . 0	$= 0$	0