Performance Consideration 3

Prof. Seokin Hong

Agenda

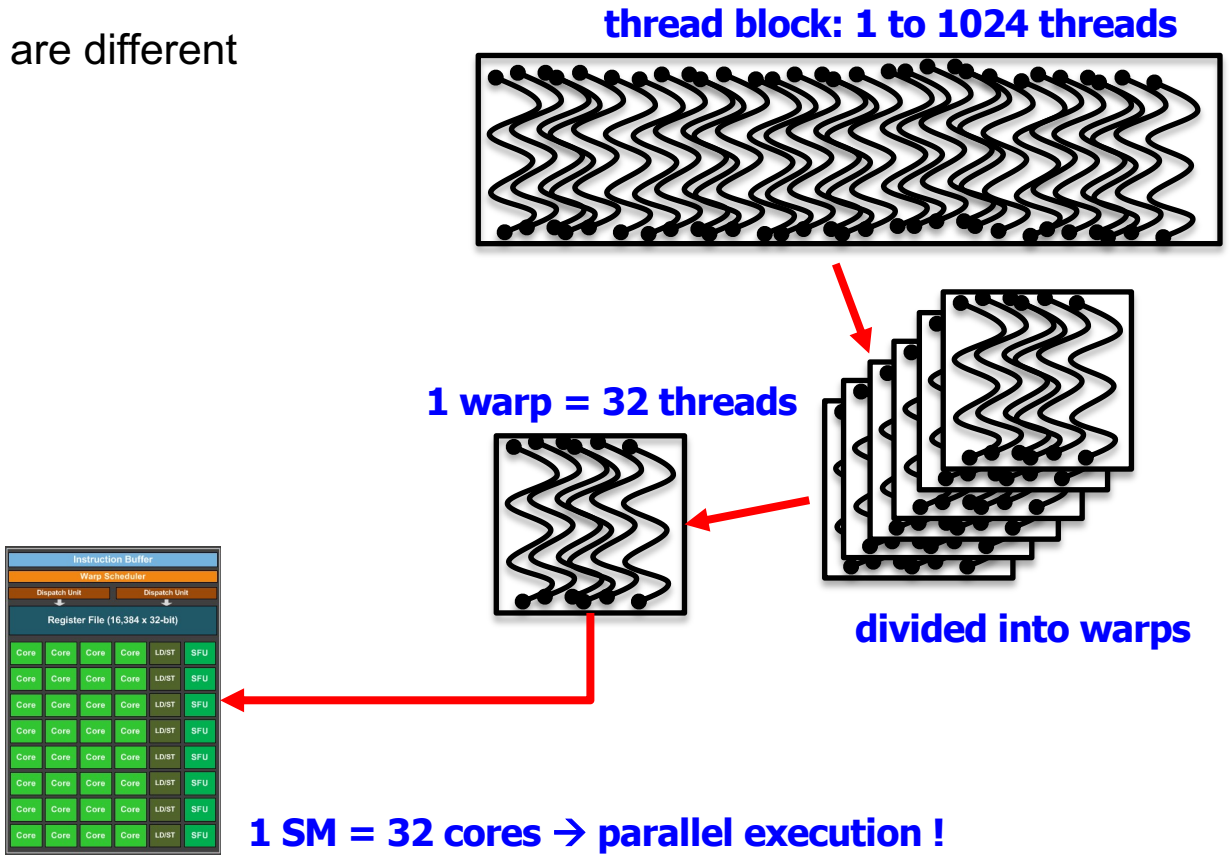
- **Memory Optimizations**
  - **More about Global Memory**
  - **Memory Coalescing** to fully utilize global memory bandwidth
    - **Memory Coalescing-aware Memory Allocation**
  - **Reducing Bank Conflict** to fully utilize shared memory bandwidth
- **Considering Control-Flow Divergence**
  - Warps and SIMD Hardware
- **Considering Occupancy**
  - Dynamic Partitioning of Resources
- **Considering Thread Granularity**

Considering Control-Flow Divergence

Slide Credit: Slides are modified from Prof Baek's slides

Review: Warps as Scheduling Units

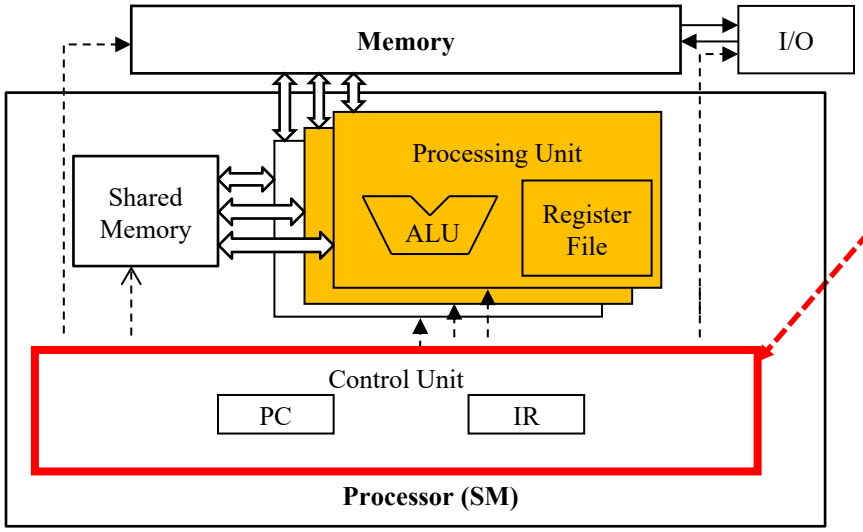
- Each block is divided into 32-thread warps
  - Warps are scheduling units in SM
  - Threads in a warp execute in **Single Instruction Multiple Data (SIMD)** manner
    - Ex) add r1, r2, r3
      - The r2 and r3 values are different in different core



Why SIMD?

- SM (Streaming Multiprocessor) has only one control unit that fetches and decodes instructions
  - cost-effective because the control units are quite complex

- So, 32 threads in a warp are controlled by a single CU (control unit)

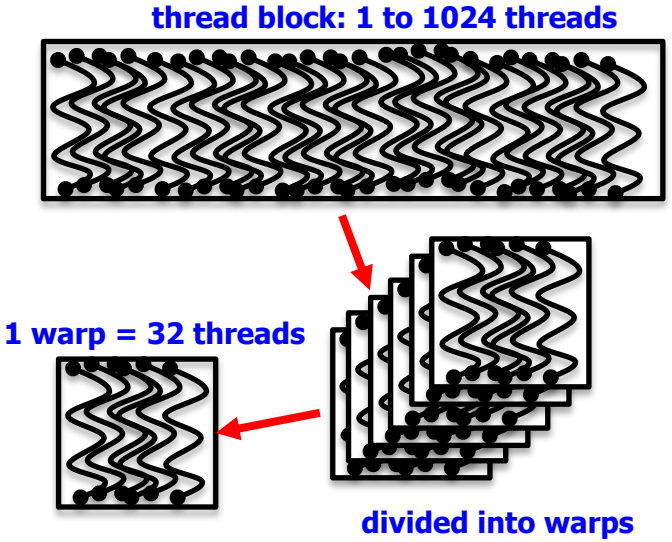


NVIDIA Pascal SM

only 1 control unit !

Control Flow Divergence

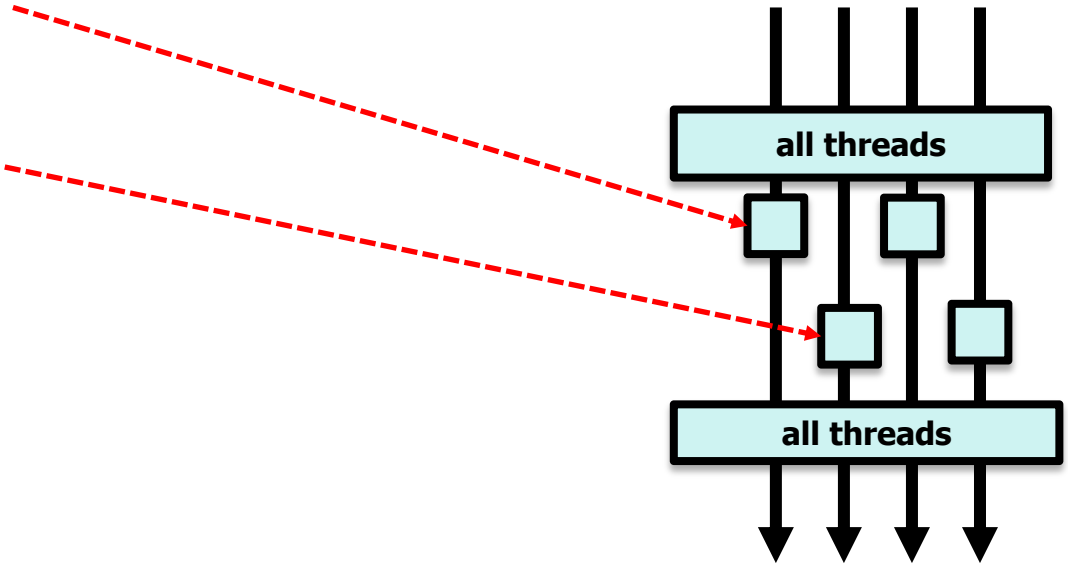
- Instructions are issued per 32 threads (warp)
- **Control Flow divergence** occurs when threads in a warp take different control flow paths by making different control decisions
  - Threads within a single warp can take different execution paths
    - **if-else**, ...
    - **for**
- **Different execution paths within a warp can be serialized**
- Different warps can execute different code with no impact on performance



Control Flow Divergence (cont'd)

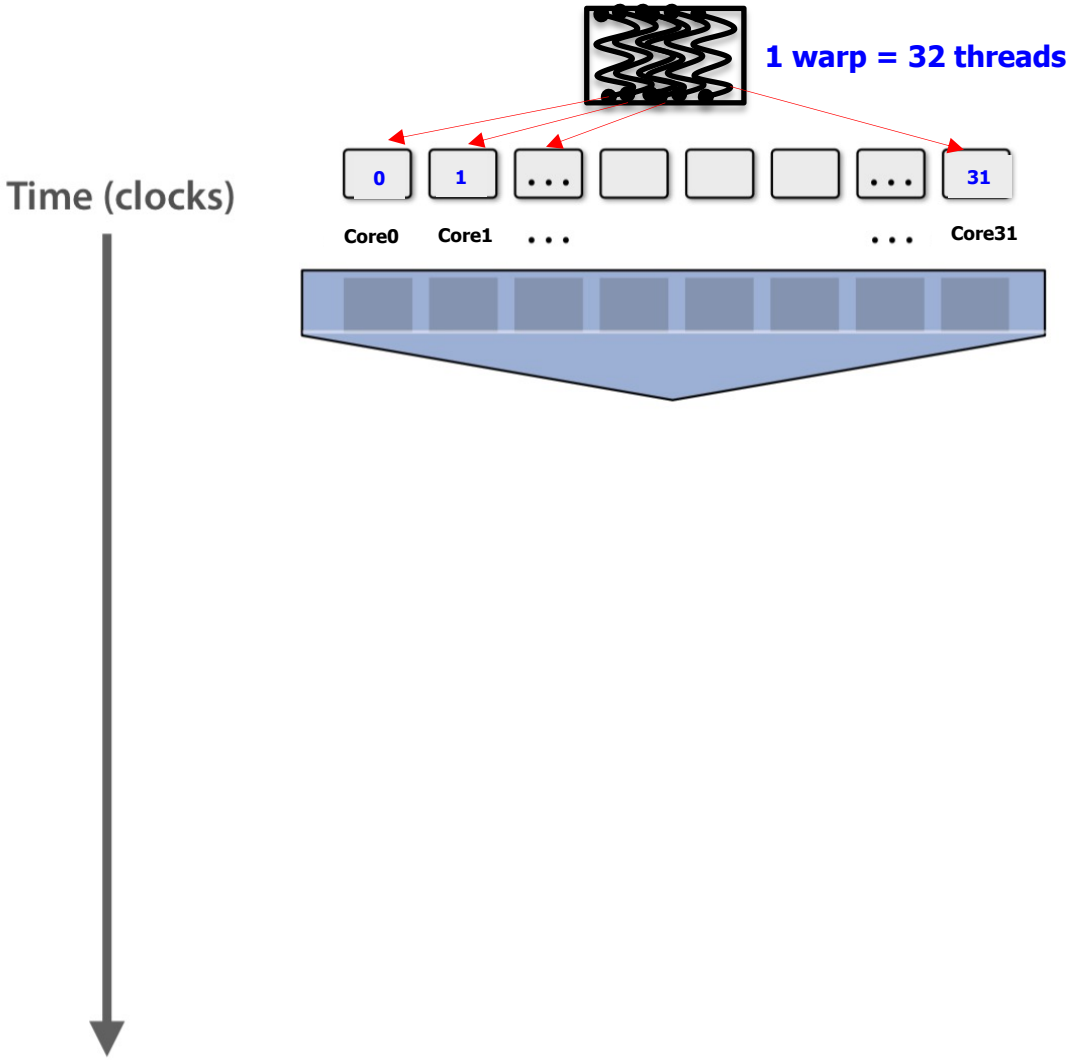
- Example with divergence:

```
if (threadIdx.x % 2 == 0) {
    ...
} else {
    ...
}
…
```



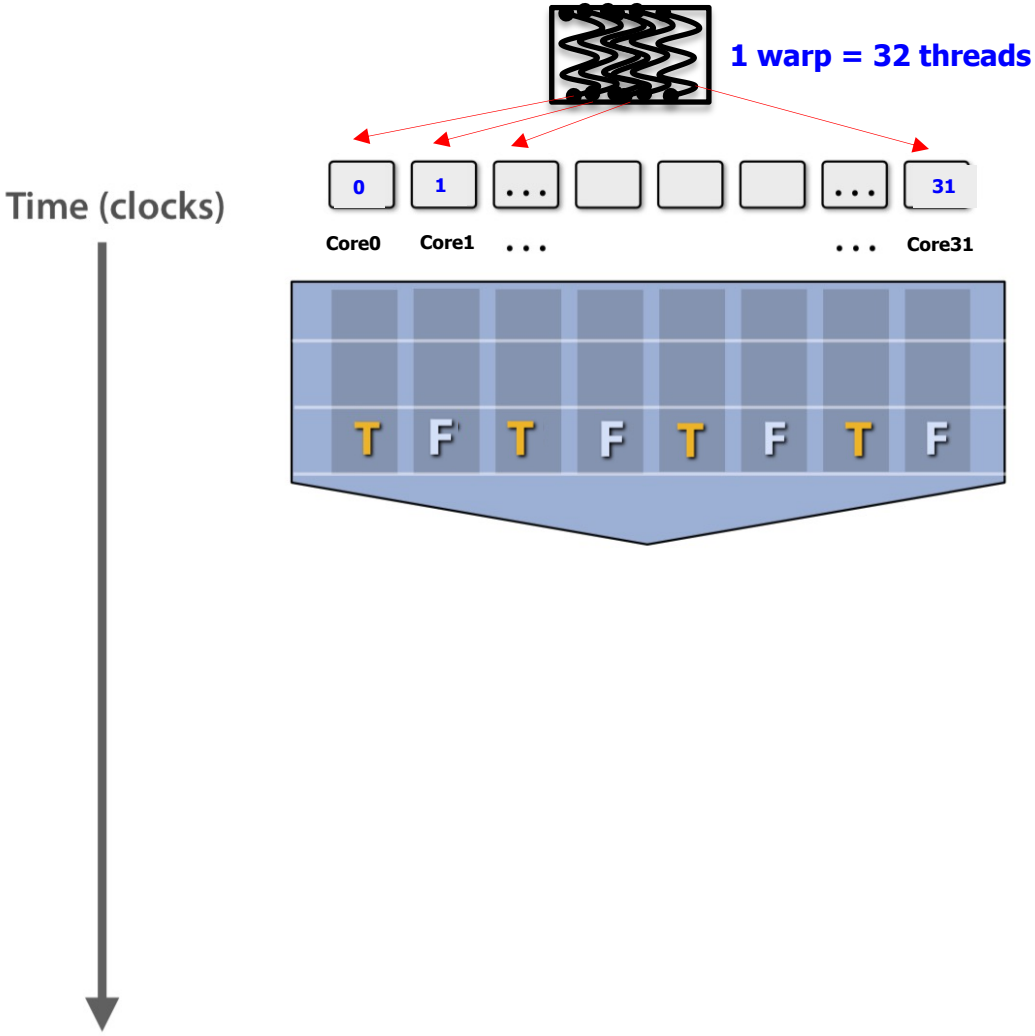
- If branch granularity < warp size,
  - **Different execution paths within a warp are serialized**

Control Flow Divergence (cont'd)




```
<unconditional code>

float x = A[i];

if (threadIdx.x % 2 == 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

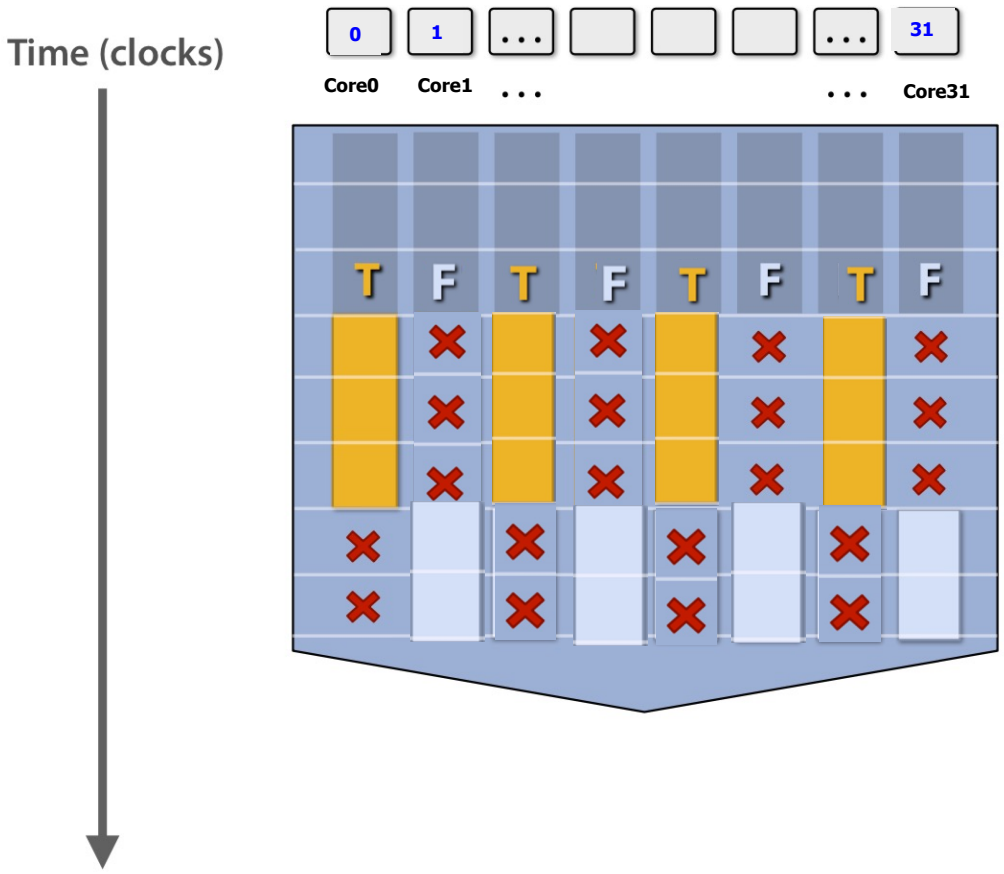
Control Flow Divergence (cont'd)




```
<unconditional code>

float x = A[i];

if (threadIdx.x % 2 == 0) {
   float tmp = exp(x,5.f);
   tmp *= kMyConst1;
   x = tmp + kMyConst2;
} else {
   float tmp = kMyConst1;
   x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Control Flow Divergence (cont'd)

- Not all Cores do useful work!

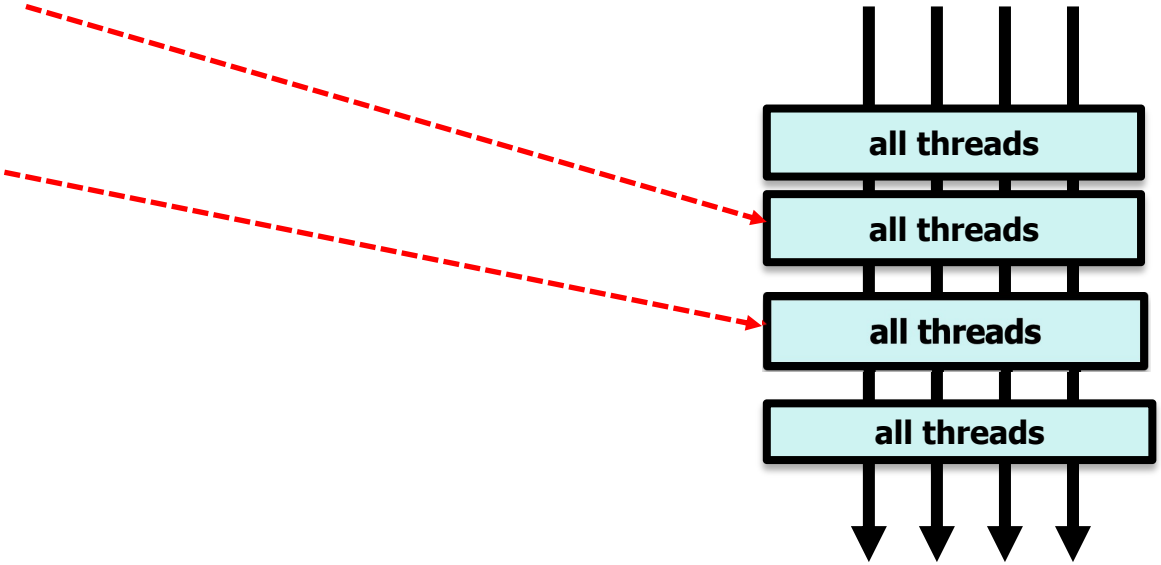



```
<unconditional code>

float x = A[i];

if (threadIdx.x % 2 == 0) {
    float tmp = exp(x,5.f);
    tmp *= kMyConst1;
    x = tmp + kMyConst2;
} else {
    float tmp = kMyConst1;
    x = 2.f * tmp;
}

<resume unconditional code>

result[i] = x;
```

Avoid diverging within a warp

- Example without divergence:

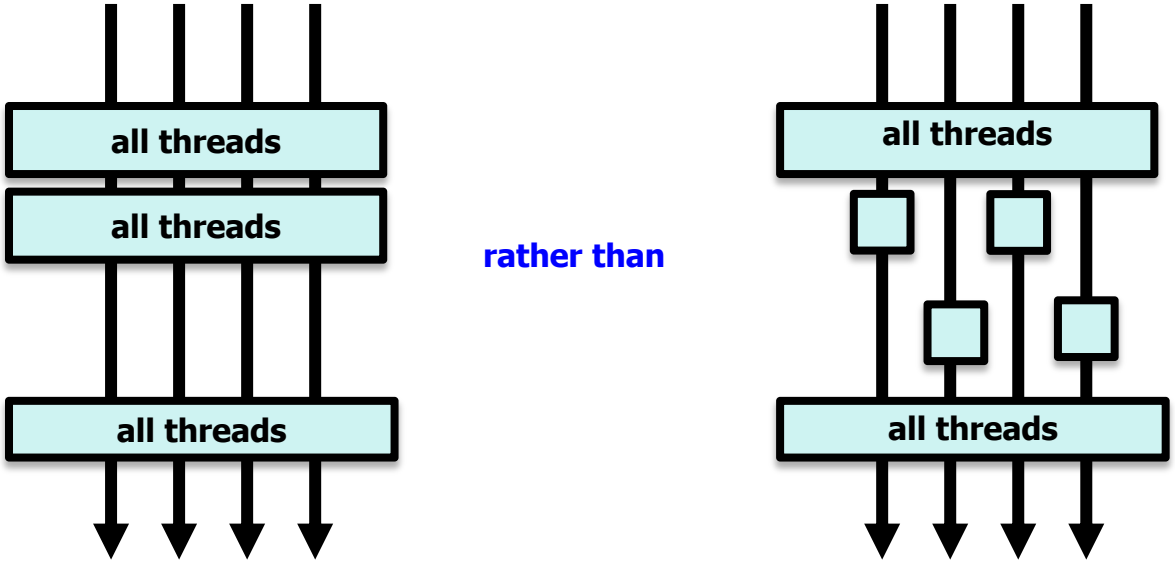
```
if (threadIdx.x / WARP_SIZE > 2) {
    ...
} else {
    ...
}
…
```



- Branch granularity is a multiple of warp size
  - All thread execute then-part only or else-part only

Avoid diverging within a warp

- how to avoid the divergent cases?
  - **algorithm-level optimization !**
  - if (threadIdx.x / WARP_SIZE > 2)



Example: Divergent Iteration

```
__global__ void per_thread_sum(int* number, float* data, float* sums) {
  ...
  // number of loop iterations is data dependent
  i = threadIdx.x;
  float sum = 0.0f;
  for (int j = 0; j < number[i]; ++j) {
    sum += data[j];
  }
  result[i] = sum;
}
```

for threadIdx.x = 0,
number[0] was 3

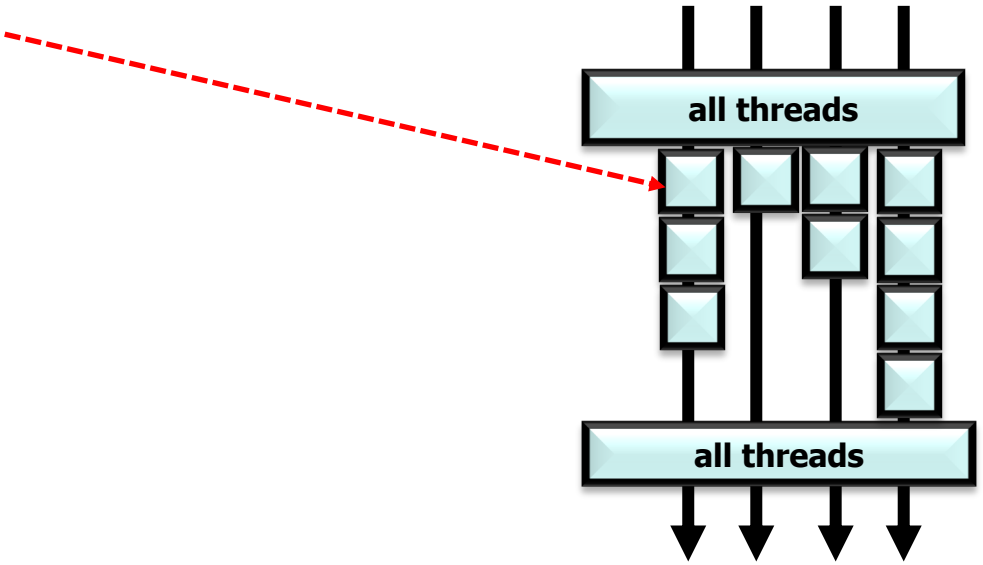
sum += data[0];
sum += data[0];
sum += data[0];

result[0] = sum;

Example: Divergent Iteration (Cont'd)

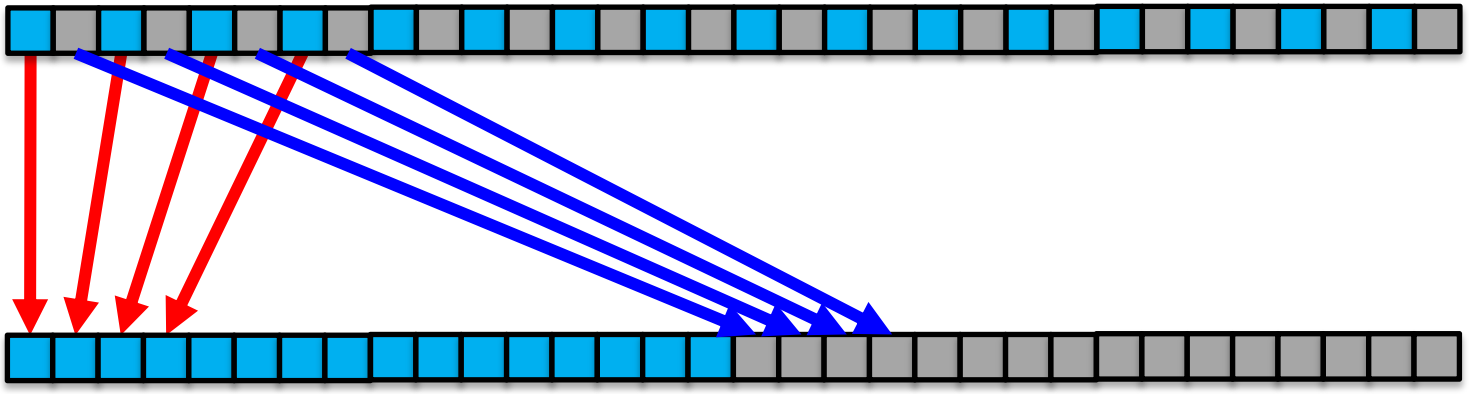
```
for (int j=0; j<number[i]; ++j) {
  sum += data[j];
}
```

```
for  threadIdx.x = 0,
number[0] was 3

sum += data[0];
sum += data[0];
sum += data[0];

result[0] = sum;
```



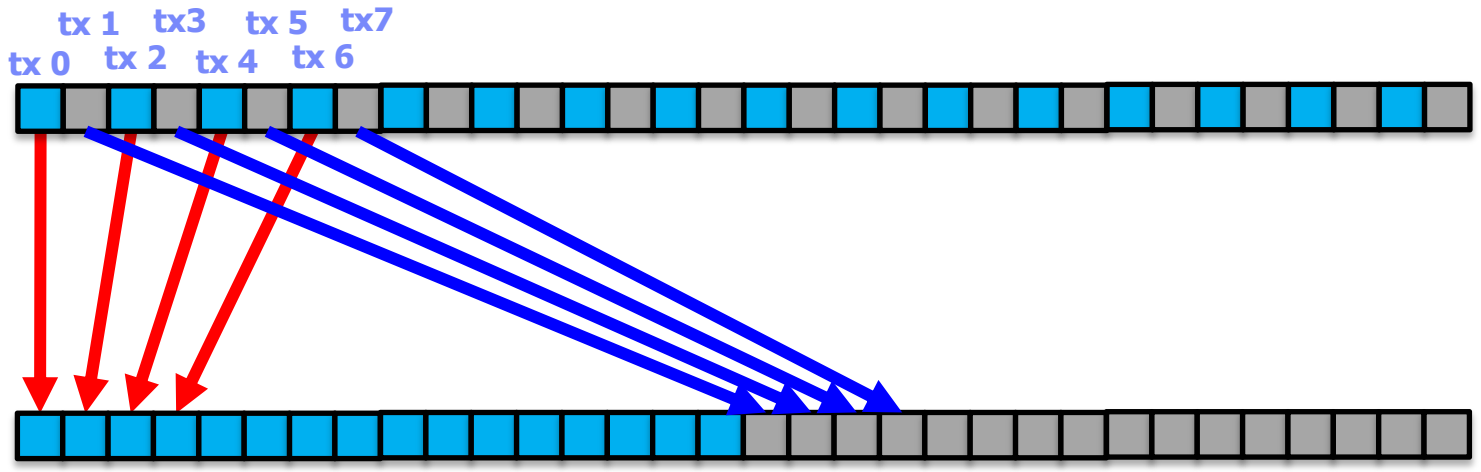
Example: Shuffling Problem

- even numbered items → left part
- odd numbered items → right part



Example: Shuffling Problem (cont'd)

- Even-Odd thread case
- in a warp,
  - even-numbered threads:
    - result[gx / 2] = input[gx];
  - odd-numbered threads:
    - result[HALF + gx / 2] = input[gx];



Even-Odd thread case

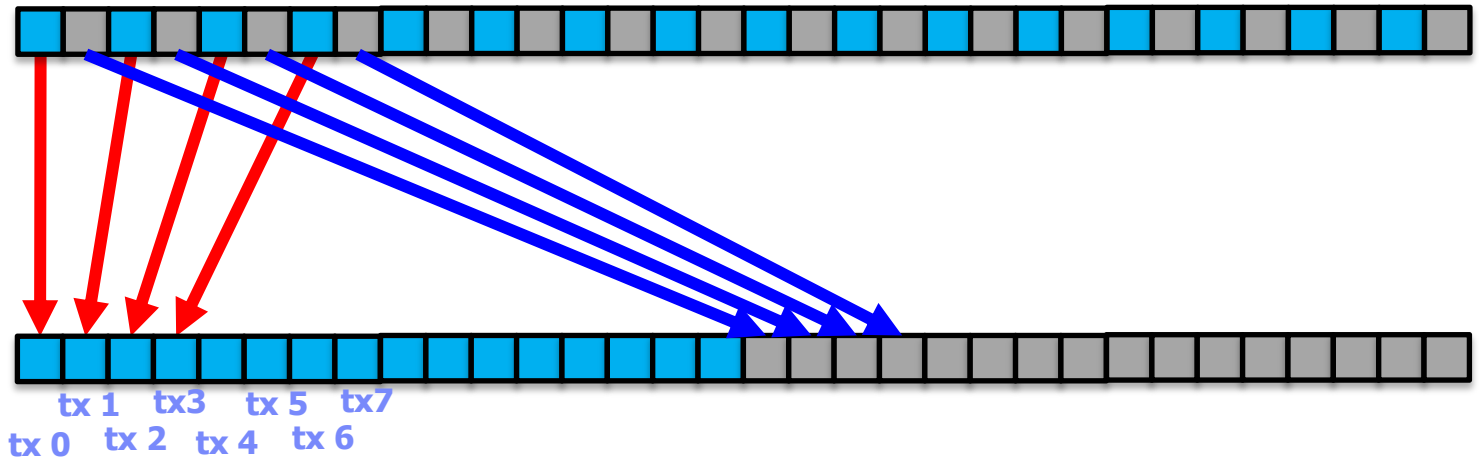
```
#include <stdio.h>
#include "common.h”

#define GRIDSIZE        (8 * 1024)
#define BLOCKSIZE       1024
#define TOTALSIZE       (GRIDSIZE * BLOCKSIZE)
#define HALF            (TOTALSIZE / 2)

//kernel
__global__ void evenodd(float* result, float* input) {
   register unsigned int gx = blockIdx.x * blockDim.x + threadIdx.x;
   if (threadIdx.x % 2 == 0) {
      result[gx / 2] = input[gx];
   } else {
      result[HALF + gx / 2] = input[gx];
   }
}
```

Even-Odd thread case (cont'd)

```
int main() {
                float* pSource = NULL;
                float* pResult = NULL;
                int i;
                // malloc memories on the host-side
                pSource = (float*)malloc(TOTALSIZE * sizeof(float));
                pResult = (float*)malloc(TOTALSIZE * sizeof(float));
                // generate source data
                genData(pSource, TOTALSIZE);
                // CUDA: allocate device memory
                float* pSourceDev = NULL;
                 float* pResultDev = NULL;
                CUDA_CHECK( cudaMalloc((void**)&pSourceDev, TOTALSIZE * sizeof(float)) );
                CUDA_CHECK( cudaMalloc((void**)&pResultDev, TOTALSIZE * sizeof(float)) );
                // CUDA: copy from host to device
                CUDA_CHECK( cudaMemcpy(pSourceDev, pSource, TOTALSIZE * sizeof(float), cudaMemcpyHostToDevice) );
                // CUDA: launch the kernel
                dim3 dimGrid(GRIDSIZE, 1, 1);
                dim3 dimBlock(BLOCKSIZE, 1, 1);
                evenodd<<< dimGrid, dimBlock>>>(pResultDev, pSourceDev);
                // CUDA: copy from device to host
                CUDA_CHECK( cudaMemcpy(pResult, pResultDev, TOTALSIZE * sizeof(float),
                                                                        cudaMemcpyDeviceToHost) );
                // print sample cases
                printf("SOURCE: [0] = %f, [1] = %f, [2] = %f, [3] = %f\n",pSource[0], pSource[1], pSource[2], pSource[3]);
                printf("RESULT: [0] = %f, [1] = %f, [HALF] = %f, [HALF+1] = %f\n",pResult[0], pResult[1], pResult[HALF], pResult[HALF + 1]);
}
```

Example: Shuffling Problem (cont'd)

- Half-by-half case
  - left-half warps:
    - result[gx] = input[gx * 2];
  - right-half warps:
    - result[gx] = input[(gx – HALF) * 2 + 1];



Half-by-half case

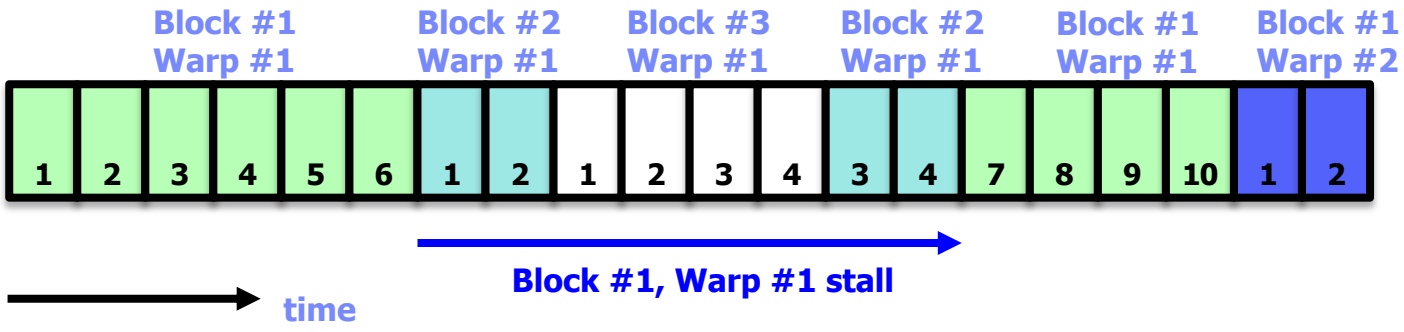
```
//kernel
__global__ void halfbyhalf(float* result, float* input) {
    register unsigned int tx = threadIdx.x;
    register unsigned int gx = blockIdx.x * blockDim.x + threadIdx.x;
    if (gx < HALF) {    // left half
        result[gx] = input[gx * 2];
    } else {      // right half
        result[gx] = input[(gx - HALF) * 2 + 1];
    }
}
```

Considering Occupancy

Slide Credit: Slides are modified from Prof Baek's slides

Review: 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



Review: Thread Scheduling

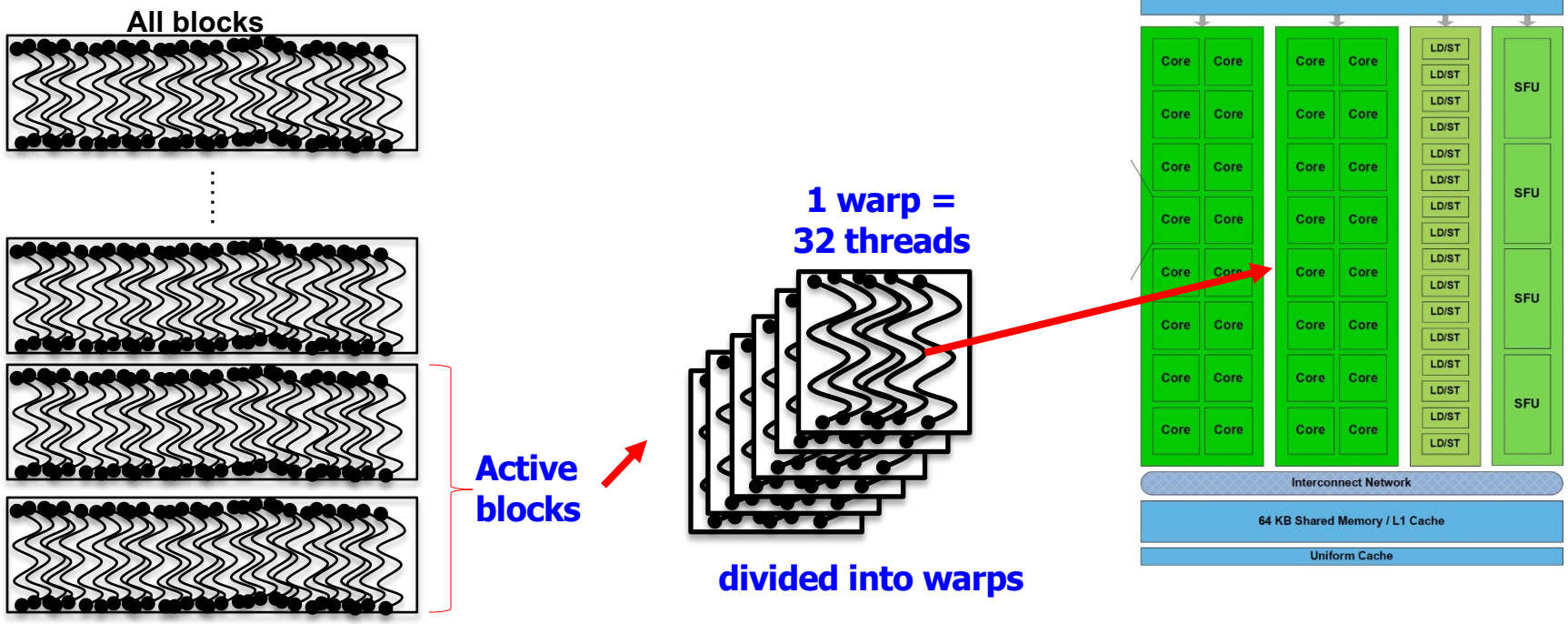
- What happens if all warps are stalled?
  - No instruction issued → performance lost

- Most common reason for stalling?
  - Waiting on global memory

- If your code frequently reads global memory,
  - **You should try to maximize (multiprocessor) occupancy to hide the global memory latency**

Occupancy

- **The (multiprocessor) occupancy**
  - the ratio of **active (aka resident) warps** to the **maximum number of warps** supported on a multiprocessor of the GPU
  - **Example**
    - Max. number of active warps per SM of a GPU = 64
    - The number of active warps per SM = 32
    - Occupancy= 32/64=0.5

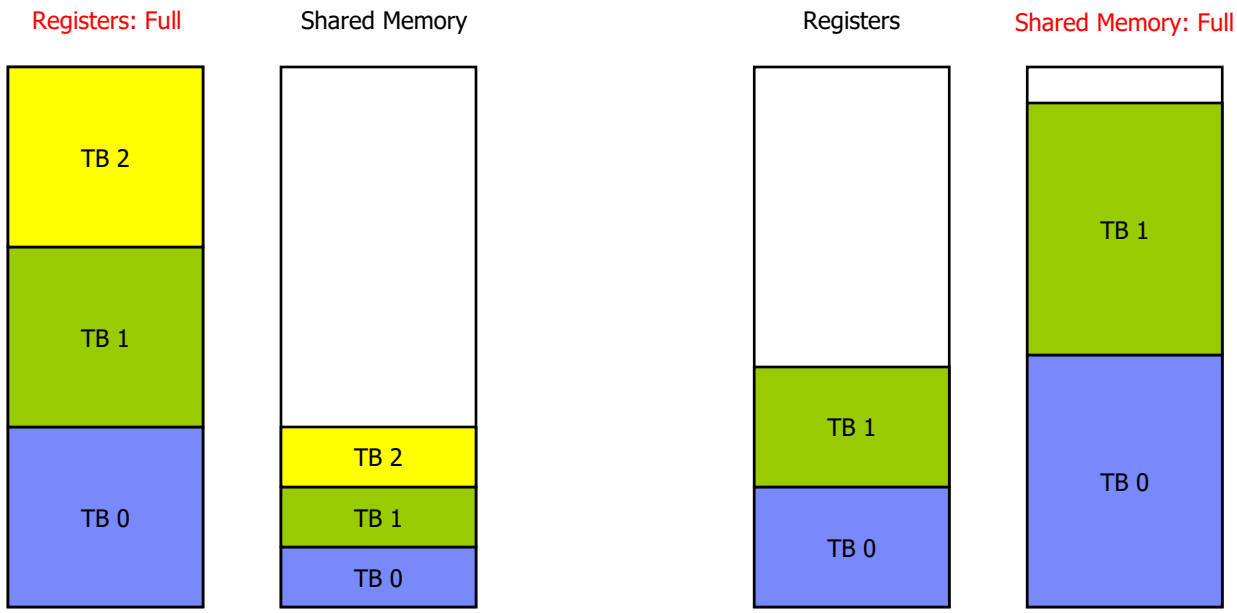


What determines occupancy?

- **Register usage** per thread
- **Shared memory** per thread block
- Block size

Resource Limits

- Pool of registers and shared memory per SM
  - Registers are allocated to each thread
  - Shared memory are allocated to each thread block
  - If registers or shared memory are **fully utilized → no more thread blocks**



Resource Limits (cont'd)

- up to 2048 threads per SM
- up to 1024 threads per Block
- up to 32 blocks per SM
- These numbers depend on the CC (compute capability) of GPU

Technical specifications	Compute capability (version)																		
	1.0	1.1	1.2	1.3	2.x	3.0	3.2	3.5	3.7	5.0	5.2	5.3	6.0	6.1	6.2	7.0	7.2	7.5	8.0	8.6
Maximum number of resident grids per device (concurrent kernel execution)	t.b.d.				16		4		32			16	128	32	16	128	16	128		
Maximum dimensionality of grid of thread blocks	2				3															
Maximum x-dimension of a grid of thread blocks	65535				$2^{31} - 1$															
Maximum y-, or z-dimension of a grid of thread blocks	65535																			
Maximum dimensionality of thread block	3																			
Maximum x- or y-dimension of a block	512				1024															
Maximum z-dimension of a block	64																			
Maximum number of threads per block	512				1024															
Warp size	32																			
Maximum number of resident blocks per multiprocessor	8					16				32								16	32	16
Maximum number of resident warps per multiprocessor	24		32		48					64								32	64	48
Maximum number of resident threads per multiprocessor	768		1024		1536					2048								1024	2048	1536
Number of 32-bit registers per multiprocessor	8 K		16 K		32 K		64 K		128 K			64 K								
Maximum number of 32-bit registers per thread block	N/A				32 K	64 K	32 K			64 K		32 K		64 K		32 K		64 K		
Maximum number of 32-bit registers per thread	124				63									255						

- to use 32 blocks, we need 2048 / 32 = **64 threads per block** !
  - If the number of threads in a block **(Block size)** is too small, occupancy will be low

Occupancy Calculation

- **Assumption**
  - Max. warps per SM : 64
  - Number of registers per SM : 64K

- **Example1**
  - Kernel uses 20 registers per thread
  - Active threads = 64K/20 = 3276
  - Active warps = 103
  - Occupancy = 1

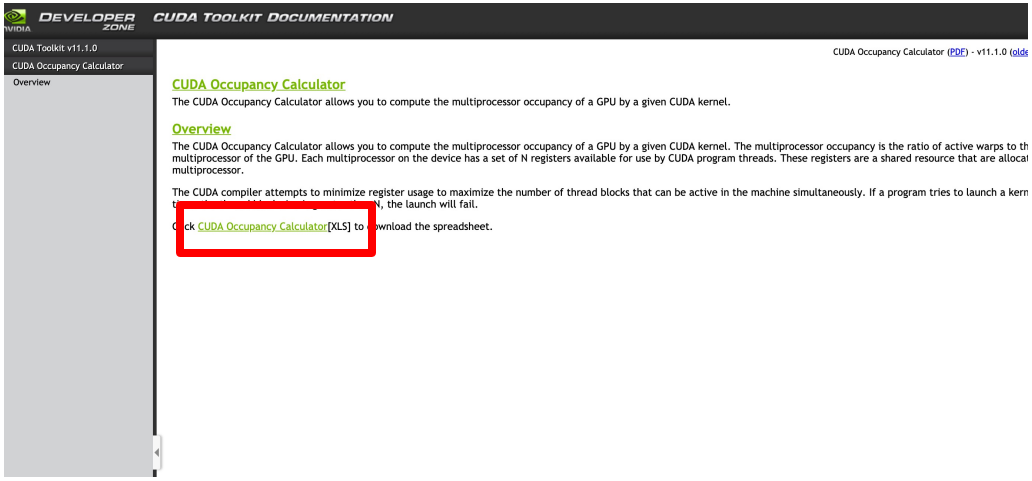
- **Example2**
  - Kernel uses 64 registers per thread
  - Active threads = 64K/64 = 1024
  - Active warps = 32
  - Occupancy = 32/64 = 0.5

Get the used resources

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

```
$ nvcc -Xptxas -v matmul-shared.cu
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z6matmulPfPKfS1_i' for 'sm_30'
ptxas info    : Function properties for _Z6matmulPfPKfS1_i
  0 bytes stack frame, 0 bytes spill stores, 0 bytes spill loads
ptxas info    : Used 32 registers, 8192 bytes smem, 380 bytes cmem[0]
```

- Plug those numbers into CUDA GPU Occupancy Calculator
  - https://docs.nvidia.com/cuda/cuda-occupancy-calculator/index.html



CUDA GPU Occupancy Calculator

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

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

1.) Select Compute Capability (click):	7.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	

2.) Enter your resource usage:		
Threads Per Block	1024	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	8192	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	2048	(Help)
Active Warps per Multiprocessor	64	
Active Thread Blocks per Multiprocessor	2	
Occupancy of each Multiprocessor	100%	

Physical Limits for GPU Compute Capability:	7.0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	32	64	2
Registers (Warp limit per SM due to per-warp reg count)	32	64	2
Shared Memory (Bytes)	8192	65536	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

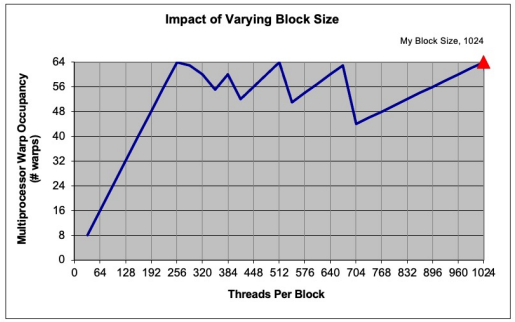
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	2	32	64
Limited by Registers per Multiprocessor	2	32	64
Limited by Shared Memory per Multiprocessor	8		

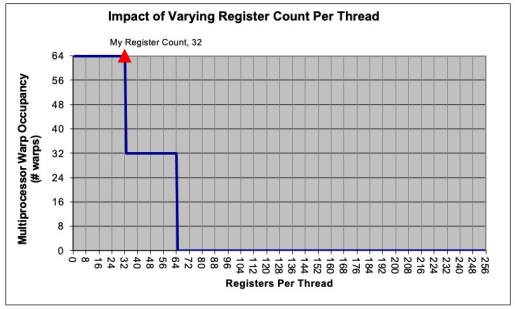
Note: Occupancy limiter is shown in orange

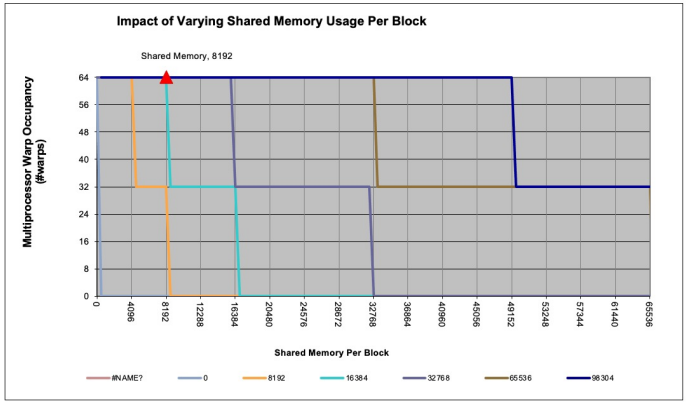
Physical Max Warps/SM = 64
Occupancy = 64 / 64 = 100%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







CUDA GPU Occupancy Calculator

CUDA Occupancy Calculator

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

1.) Select Compute Capability (click):	7.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	

2.) Enter your resource usage:		
Threads Per Block	8	(Help)
Registers Per Thread	32	
User Shared Memory Per Block (bytes)	8192	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	256	(Help)
Active Warps per Multiprocessor	8	
Active Thread Blocks per Multiprocessor	8	
Occupancy of each Multiprocessor	13%	

Physical Limits for GPU Compute Capability:	7.0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources		Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps	(Threads Per Block / Threads Per Warp)	1	64	32
Registers	(Warp limit per SM due to per-warp reg count)	1	64	64
Shared Memory (Bytes)		8192	65536	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	32		
Limited by Registers per Multiprocessor	64		
Limited by Shared Memory per Multiprocessor	**8**	**1**	**8**

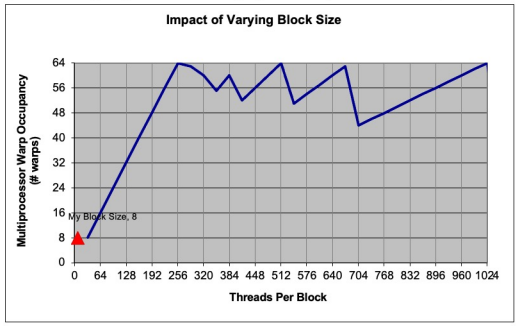
Note: Occupancy limiter is shown in orange

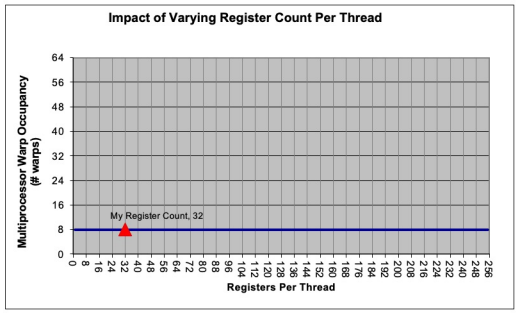
Physical Max Warps/SM = 64
Occupancy = 8 / 64 = 13%

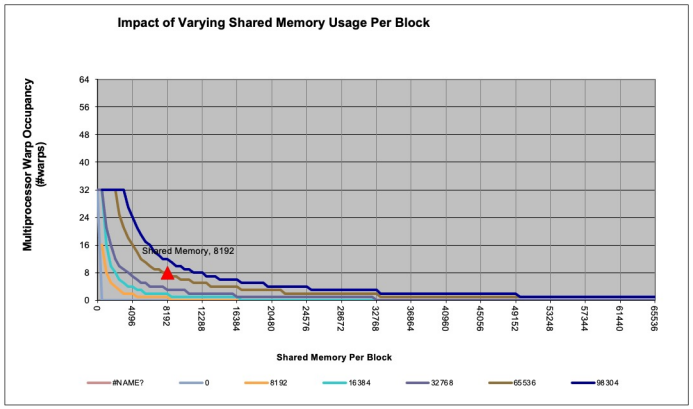
CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







CUDA GPU Occupancy Calculator

CUDA Occupancy Calculator

Click Here for detailed instructions on how to use this occupancy calculator.
For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

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

1.) Select Compute Capability (click):	7.0	(Help)
1.b) Select Shared Memory Size Config (bytes)	65536	

2.) Enter your resource usage:		
Threads Per Block	1024	(Help)
Registers Per Thread	64	
User Shared Memory Per Block (bytes)	8192	

(Don't edit anything below this line)

3.) GPU Occupancy Data is displayed here and in the graphs:		
Active Threads per Multiprocessor	1024	(Help)
Active Warps per Multiprocessor	32	
Active Thread Blocks per Multiprocessor	1	
Occupancy of each Multiprocessor	50%	

Physical Limits for GPU Compute Capability:	7.0
Threads per Warp	32
Max Warps per Multiprocessor	64
Max Thread Blocks per Multiprocessor	32
Max Threads per Multiprocessor	2048
Maximum Thread Block Size	1024
Registers per Multiprocessor	65536
Max Registers per Thread Block	65536
Max Registers per Thread	255
Shared Memory per Multiprocessor (bytes)	65536
Max Shared Memory per Block	65536
Register allocation unit size	256
Register allocation granularity	warp
Shared Memory allocation unit size	256
Warp allocation granularity	4
Shared Memory Per Block (bytes) (CUDA runtime use)	0

Allocated Resources	Per Block	Limit Per SM	= Allocatable Blocks Per SM
Warps (Threads Per Block / Threads Per Warp)	32	64	2
Registers (Warp limit per SM due to per-warp reg count)	32	32	1
Shared Memory (Bytes)	8192	65536	8

Note: SM is an abbreviation for (Streaming) Multiprocessor

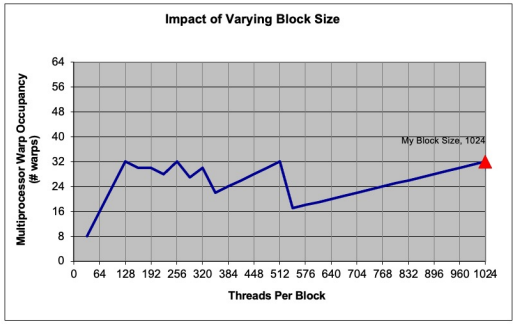
Maximum Thread Blocks Per Multiprocessor	Blocks/SM	* Warps/Block	= Warps/SM
Limited by Max Warps or Max Blocks per Multiprocessor	2		
Limited by Registers per Multiprocessor	**1**	**32**	**32**
Limited by Shared Memory per Multiprocessor	8		

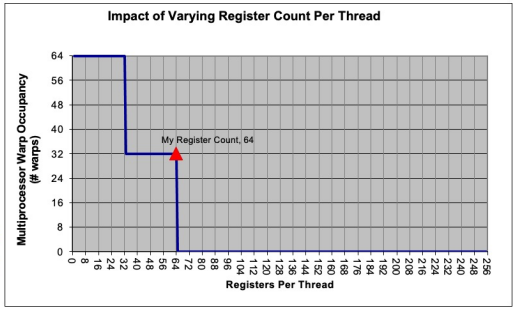
Note: Occupancy limiter is shown in orange

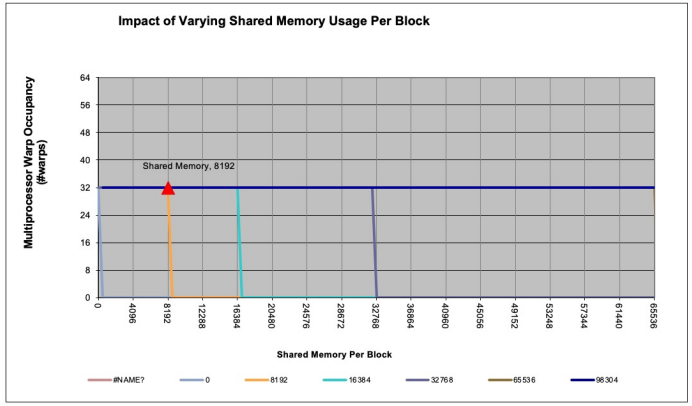
Physical Max Warps/SM = 64
Occupancy = 32 / 64 = 50%

CUDA Occupancy Calculator	
Version:	11.1
Copyright and License	

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.







Set the number of registers per thread

- Pass option `-maxrregcount=X` to nvcc
  - force to use only less than or equal to X number of registers.

- $ nvcc -Xptxas -v -maxrregcount=16 matmul-shared.cu

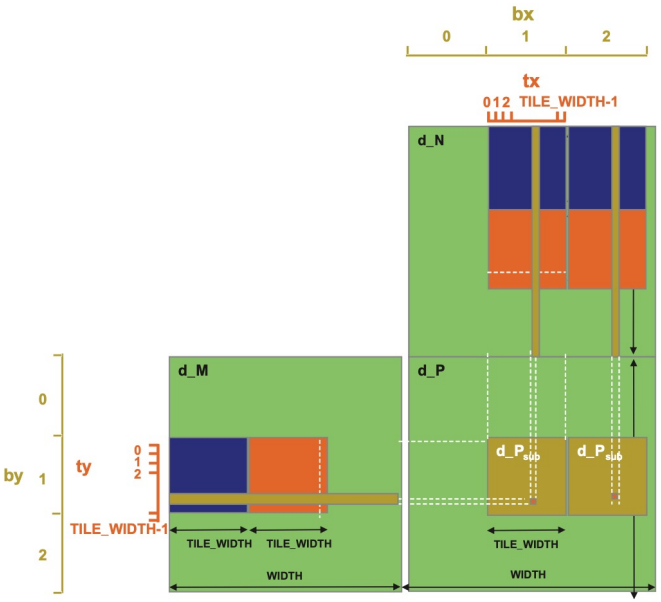
```
ptxas info    : 0 bytes gmem
ptxas info    : Compiling entry function '_Z6matmulPfPKfS1_i' for 'sm_30'
ptxas info    : Function properties for _Z6matmulPfPKfS1_i
    8 bytes stack frame, 8 bytes spill stores, 8 bytes spill loads
ptxas info    : Used 16 registers, 512 bytes smem, 348 bytes cmem[0]
```

Considering Thread Granularity

Selecting the optimal granularity of threads

- Using more active threads will increase occupancy
- But, it is sometimes advantageous to **put more work into each thread** and **use fewer threads**
  - Especially, **when some redundant work exists between threads**
    - e.g., floating point calculation, loading data from global memory
  - **Eliminating redundant work** and **using smaller number of threads** can improve the overall execution speed of the kernel

- **Example**
  - The tiled matrix multiplication algorithm uses one thread to compute one element of the output P matrix
  - The calculation of two P elements in adjacent tiles uses the same M row
  - The same M row is redundantly loaded by the two blocks assigned to generate these two P tiles
  - **This redundancy can be eliminated by merging the two thread blocks into one**
  - Each thread in the new thread block calculates two P elements
    - **This will reduce the global memory access**



Agenda

- **Memory Optimizations**
  - More about Global Memory
  - Memory Coalescing to fully utilize global memory bandwidth
    - Memory Coalescing-aware Memory Allocation
  - Reducing Bank Conflict to fully utilize shared memory bandwidth

- **Considering Control-Flow Divergence**
  - Warps and SIMD Hardware

- **Considering Occupancy**
  - Dynamic Partitioning of Resources

- **Considering Thread Granularity**