# COMP 4901Q: High Performance Computing (HPC)

## Lecture 8: CUDA Programming on the GPU

Instructor: Shaohuai SHI (shaohuais@cse.ust.hk)
Teaching assistants: Mingkai TANG (mtangag@connect.ust.hk)
Yazhou XING (yxingag@connect.ust.hk)

Course website: https://course.cse.ust.hk/comp4901q/

# Outline

- CUDA Thread Organization
- Mapping Threads to Multidimensional Data
- Mapping Threads to Hardware
- Example: Matrix-Matrix Multiplication

- GPU Memories: Shared memory
- APOD Design Cycle

- CUDA Optimization Techniques
  - Kernel configurations: # of blocks, # of threads per block
  - Control flow
  - Global memory access
  - Shared memory access
  - Instruction optimization

# CUDA Thread Organization

- A grid of threads are organized into 1-D, or 2-D, or 3-D array of blocks, as specified by the execution configuration
  - The structure of the grid can be obtained by the built-in variable gridDim
    - gridDim.x, gridDim.y, gridDim.z
    - For compute capability below 3.0, x, y, z must be less than 65536
    - For compute capability 3.0 or above, x must be less than $2^{31}$, y and z must be less than 65536
  - Each block is identified by the built-in variable blockIdx
    - blockIdx.x, blockIdx.y, blockIdx.z
  - The maximum number of threads in a block[1]
    - 512 for compute capability 1.x
    - 1024 for compute capability 2.x or above

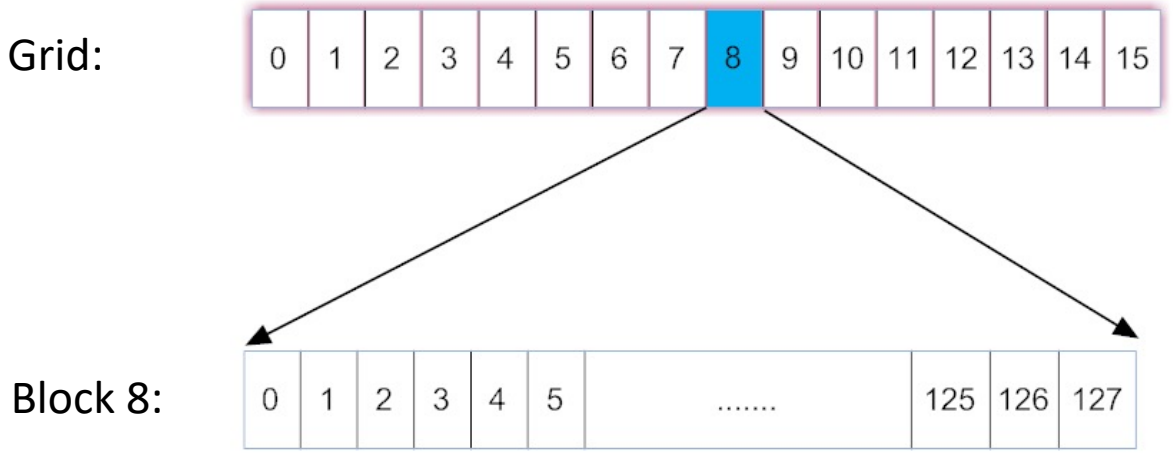
[1]Please check Appendix G of Ref. [2] for details about Compute Capability.

# CUDA Thread Organization (Cont.)

- A block of threads are further organized into 1-D, or 2-D, or 3-D array of threads, as specified by the execution configuration
  - The structure of the block can be obtained by the built-in variable blockDim
    - blockDim.x, blockDim.y, blockDim.z
    - blockDim.z must be less than or equal to 64
  - All threads in the same block share the same value of blockIdx
  - Each thread is identified by the built-in variable threadIdx
    - threadIdx.x, threadIdx.y, threadIdx.z

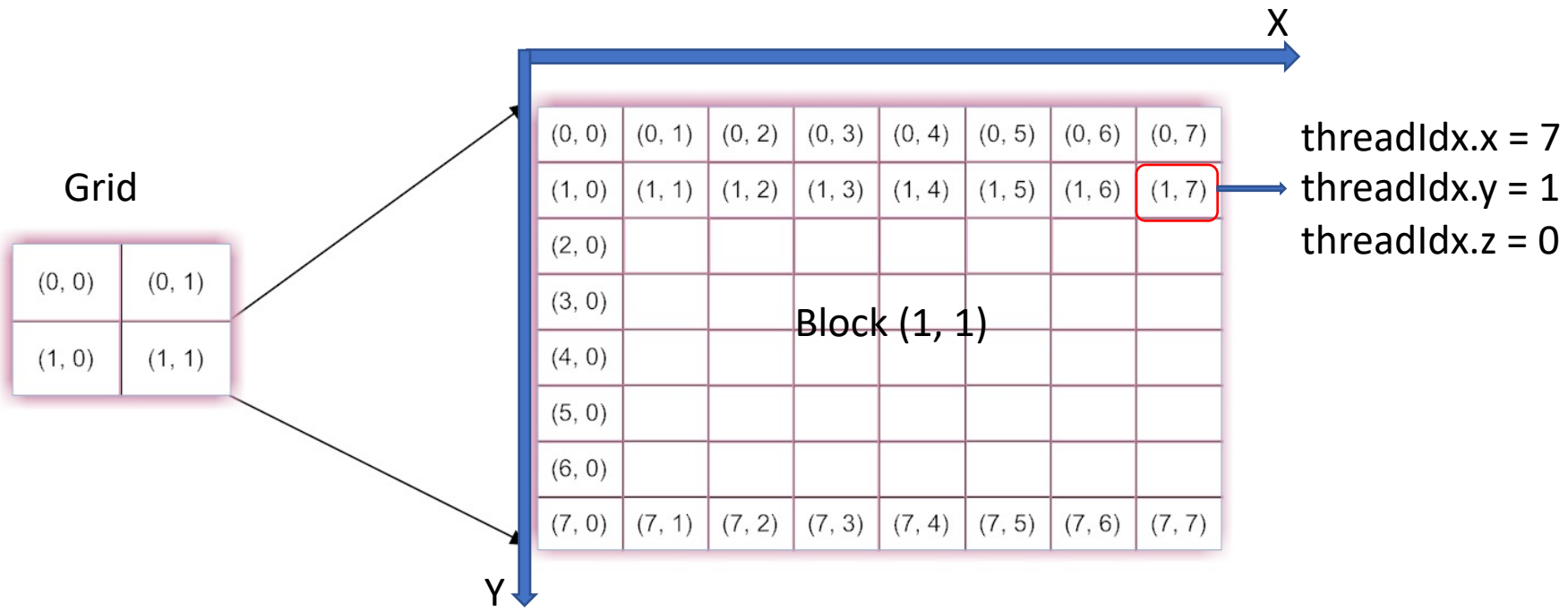
# 1D Example

- If you want to create a grid with 16 blocks in 1D, and each block has 128 threads in 1D:
  - dim3 dimBlock(128, 1, 1); /* set y and z to 1, you get 1D */
  - dim3 dimGrid(16, 1, 1); /* set y and z to 1, you get 1D */
  - vecAddkernel<<<dimGrid, dimBlock>>>(…);



# 2D Example

- If you want to create a grid with 2x2 blocks in 2D, and each block has 8x8 threads in 2D:
  - dim3 dimBlock(8, 8, 1); /* set z to 1, you get 2D */
  - dim3 dimGrid(2, 2, 1); /* set z to 1, you get 2D */
  - vecAddkernel<<<dimGrid, dimBlock>>>(…);



# Mapping Threads to Multidimensional Data

- The choice of 1D, 2D, or 3D thread organizations is usually based on the nature of the data.

- E.g., images are 2D array of pixels.

  - Convenient to use a 2D grid that consists of 2D blocks

  - Each thread processes one pixel: the location of the pixel can be easily calculated by blockDim, blockIdx, and threadIdx

# Example of a 76x62 Image

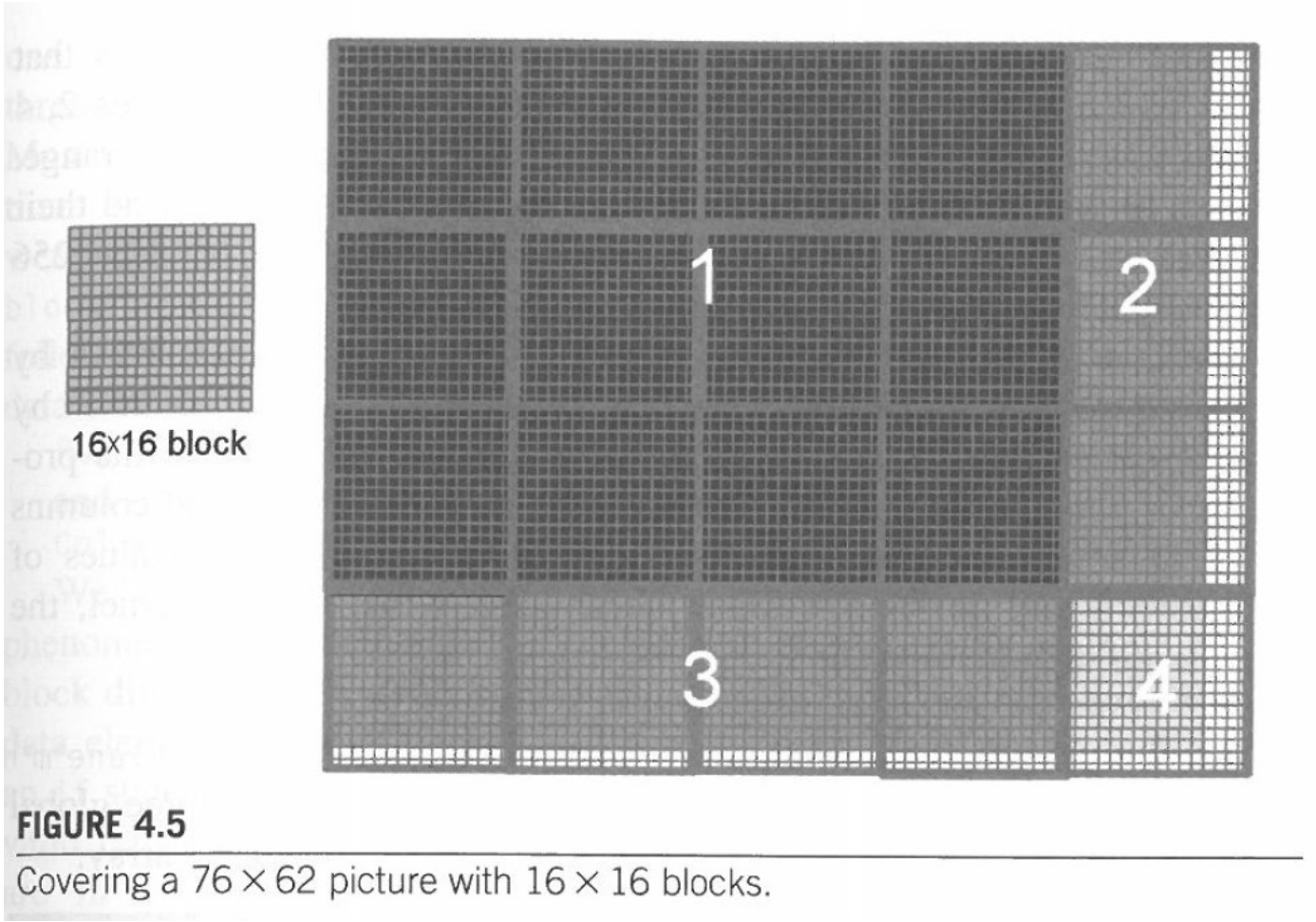



**FIGURE 4.5**

Covering a 76 × 62 picture with 16 × 16 blocks.

Figure 4.5 of Ref. 1.

Image size:
76 x 62

Block size:
16 x 16

We need 5 x 4 blocks!

80 x 64 threads will be generated, some of which are wasteful.

# Launching the Kernel

- Assume the image has a resolution of n x m

  - dim3 dimGrid(ceil(n/16.0), ceil(m/16.0), 1);
  - dim3 dimBlock(16, 16, 1);
  - pictureKernel<<<dimGrid, dimBlock>>>(…);

How would the threads be executed on GPU?

# Assigning Resources to Blocks

- Once a kernel is launched, CUDA generates the corresponding grid of threads.

- The threads, organized as many blocks, are assigned to execution resources (i.e., SMs) on a block-by-block basis.
  - Demand: many thread blocks to be executed, say N blocks in total
  - Resource: a limited number of SMs, say P SMs
  - On average, each SM will execute N/P blocks

- Active blocks (or "resident blocks")
  - An SM can handle multiple blocks at the same time. We call them "active blocks"
  - After the current active blocks finish, a new set of blocks will be assigned to the SM, and they become new active blocks

# Thread Scheduling

- Once a block is assigned to an SM, it is further divided into warps
  - Today, each ***warp*** includes 32 threads
  - Warp 0: threads 0 – 31
  - Warp 1: threads 32 – 63
  - Warp 2: threads 64 - 95
  - ……

- Warp is the unit of thread scheduling in SMs
  - The 32 threads of a warp follow the single instruction, multiple data (SIMD) model
  - An instruction is fetched, and executed for all 32 threads in the warp

# Benefit of Multi-warps: Latency Hiding

- Latency hiding by multi-warps
  - When an instruction executed by a warp needs to wait for some data, the warp will "sleep".
    - Remember that global memory has a long latency of several hundreds of GPU cycles
  - Another warp whose data is ready will be selected for execution.
    - In order to keep the execution hardware busy
  - Zero-overhead thread scheduling
    - Selecting a new active warp for execution causes little overheads

- As a general rule, we should arrange "enough" active warps for each SM.

# Hardware Limitations

- Due to limited hardware resources, the number of active blocks/warps/threads per SM is limited.

| Compute Capability | 1.x | 2.x | 3.x | 5.0 | 7.0 |
|---|---|---|---|---|---|
| Max. number of active blocks per SM | 8 | 8 | 16 | 32 | 32 |
| Max. number of active warps per SM | 24 → 32 | 48 | 64 | 64 | 64 |
| Max. number of active threads per SM | 768 → 1024 | 1536 | 2048 | 2048 | 2048 |
| Max. number of threads per block | 512 | 1024 | 1024 | 1024 | 1024 |

Remark: Other limiting factors include registers and shared memory, which will be discussed later.

# Example

- Assume compute capability of 3.0

- Case 1: block size is 8x8=64
  - How many active threads at most?

- Case 2: block size is 16x16=256
  - How many active threads at most?

- Case 3: block size is 32x 32=1024
  - How many active threads at most?

- Case 4: block size is 64x 64=4096
  - What will happen?

# Kernel Function

```
/* d_Pin points to the image data */

__global__ void PictureKernel(float *d_Pin, float *d_Pout, int n, int m)
{
   int Row = blockIdx.y * blockDim.y + threadIdx.y;
   int Col = blockIdx.x * blockDim.x + threadIdx.x;

   if( (Row < m) && (Col < n) )
      d_Pout[Row * n + Col] = 2 * d_Pin[Row * n + Col];
}
```

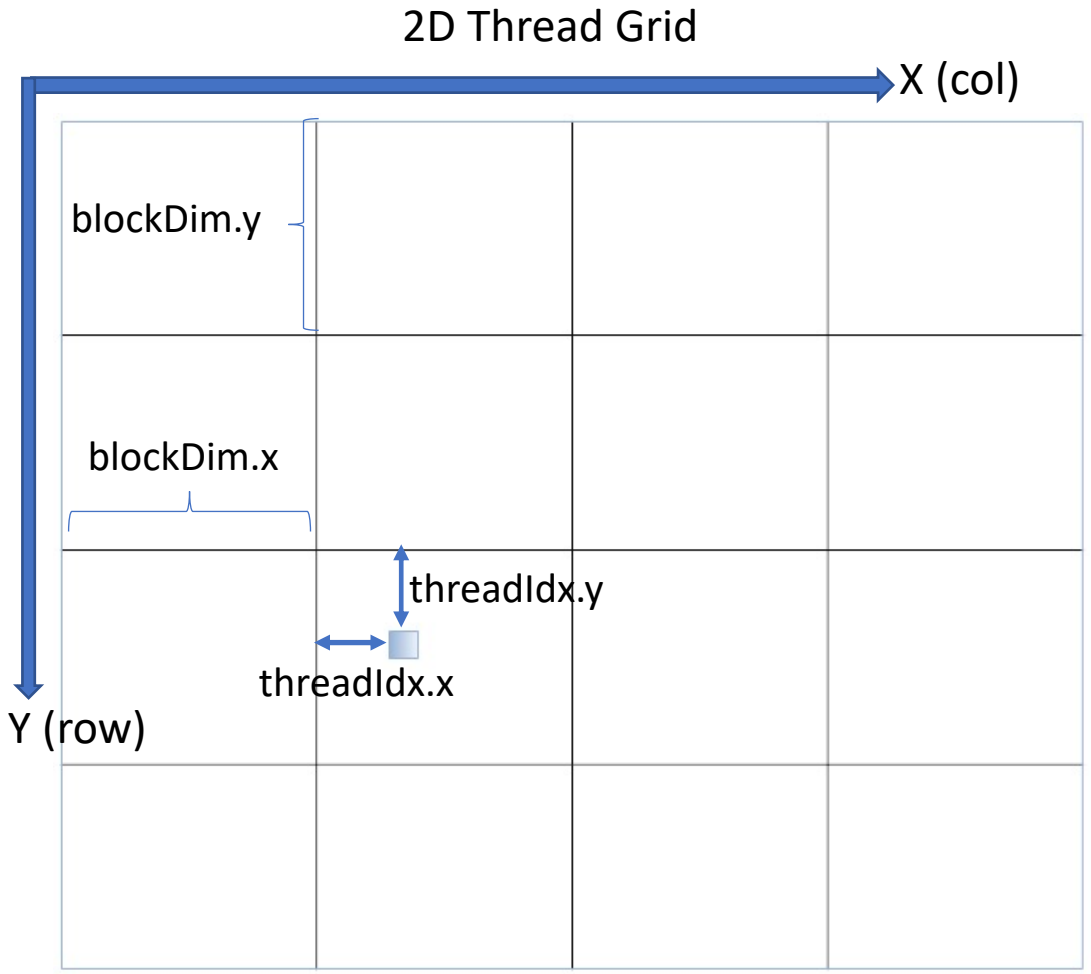
# Locate Your Data

- Consider the target thread:
  - There are blockIdx.y number of blocks on top of its own block.

    Row = blockIdx.y * blockDim.y + threadIdx.y

  - There are blockIdx.x number of blocks to the left of its own block.

    Col = blockIdx.x * blockDim.x + threadIdx.x



# Example: Matrix-Matrix Multiplication

- Problem: P = M x N

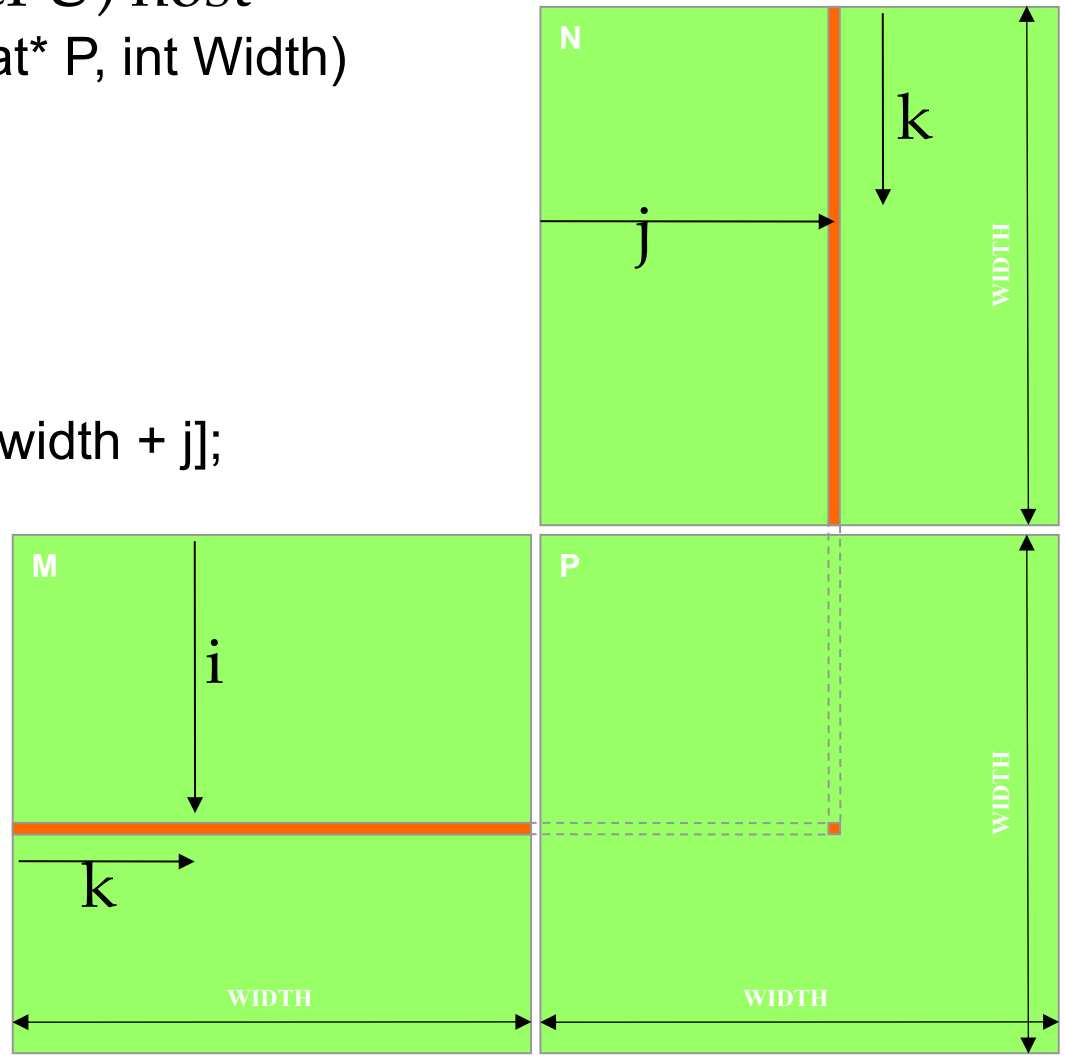
- M, N, P are square matrix: WIDTH x WIDTH

- Parallelization on GPU
  - The calculation of each item in *P* is done by a GPU thread
    - To perform a dot product of two vectors

  - Question: How to organize the threads into blocks?

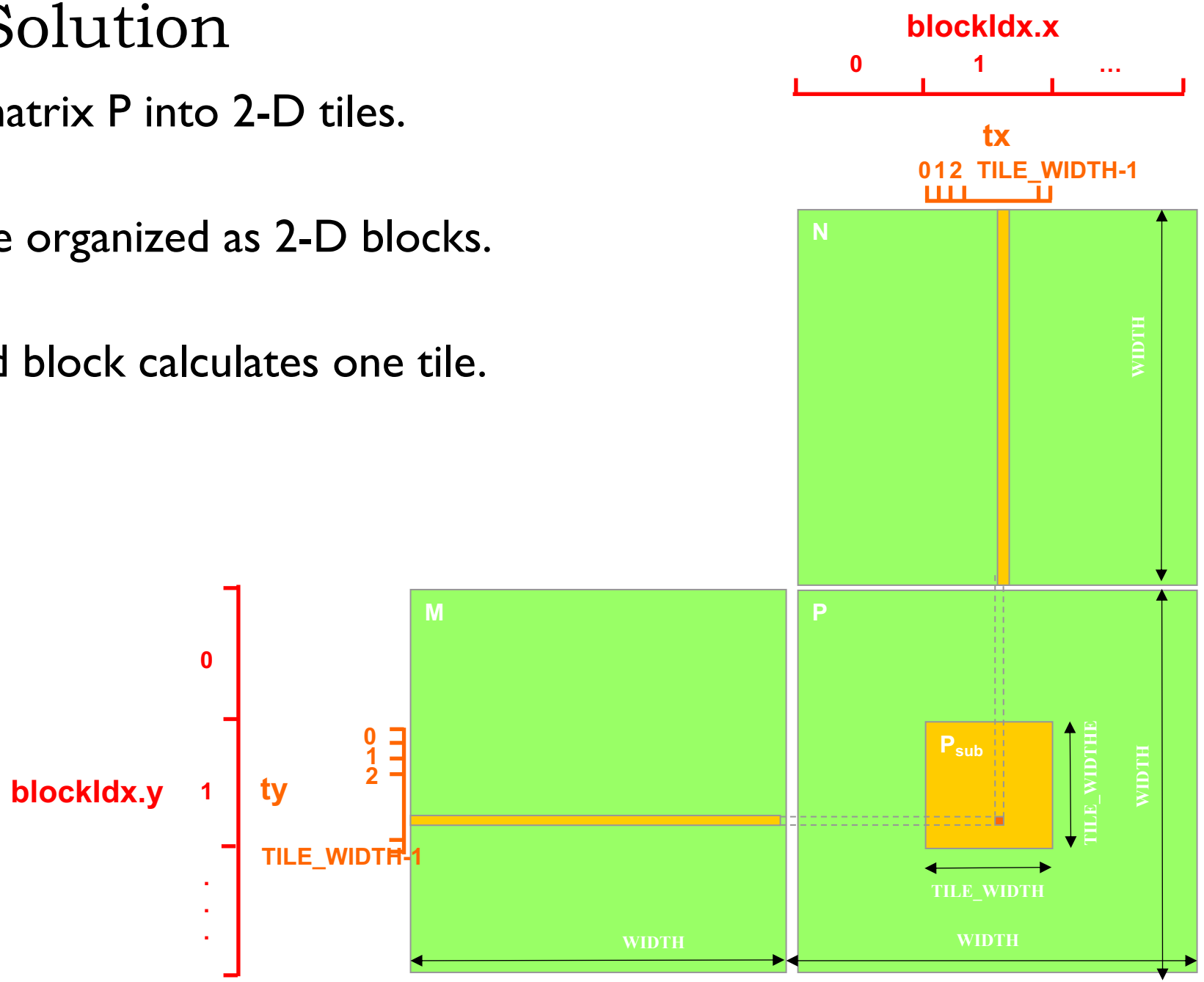
# Serial Solution

```
// Matrix multiplication on the (CPU) host
void MatMulOnHost(float* M, float* N, float* P, int Width)
{
    for (int i = 0; i < Width; i++)
        for (int j = 0; j < Width; j++) {
            float sum = 0.0;
            for (int k = 0; k < Width; k++) {
                sum += M[i * width + k] * N[k * width + j];
            }
            P[i * Width + j] = sum;
        }
}
```



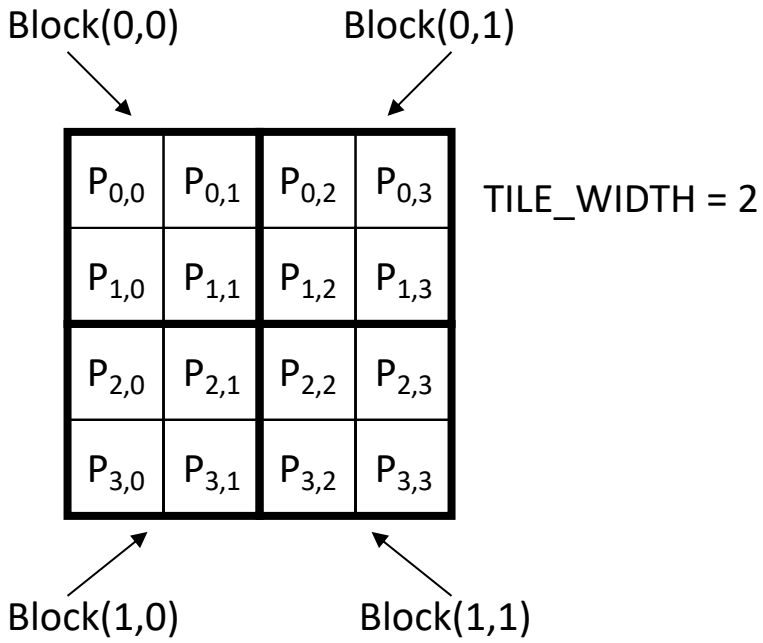
# A Tiled Solution

- Break-up matrix P into 2-D tiles.
- Threads are organized as 2-D blocks.
- Each thread block calculates one tile.

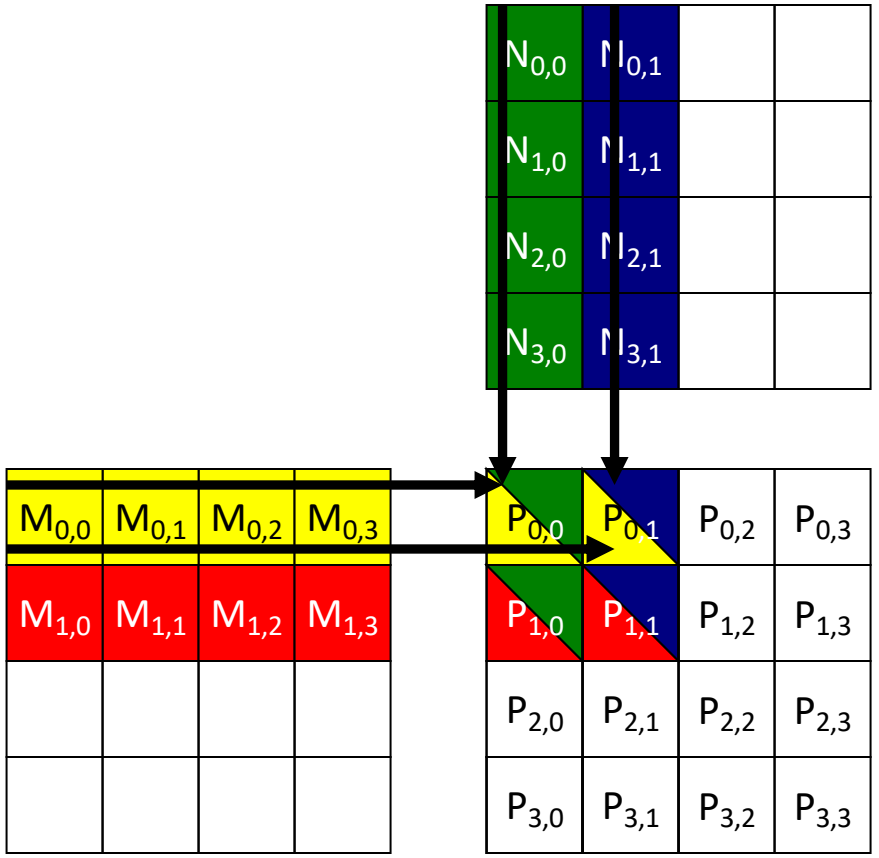


# A Small Example

- WIDTH = 4
- TILE_WIDTH = 2
- 2 x 2 = 4 blocks
- Each block has 2 x 2 = 4 threads



# A Small Example (Cont.)



# The Kernel Function

```
__global__ void MatMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
// Calculate the row index of the P element and M
  int Row = blockIdx.y*blockDim.y + threadIdx.y;
// Calculate the column index of the P element and N
  int Col = blockIdx.x*blockDim.x + threadIdx.x;


  if ( (Row < Width) && (Col < Width) ) {
    float Pvalue = 0.0;
// each thread computes one element of the block sub-matrix
    for (int k = 0; k < Width; ++k)
      Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
    Pd[Row*Width+Col] = Pvalue;
  }
}
```

# Launching the Kernel

```
#define TILE_WIDTH 16

// Setup the execution configuration
int NB = Width/TILE_WIDTH;
if (Width % TILE_WIDTH != 0) NB++;
dim3 dimGrid(NB, NB);
dim3 dimBlock(TILE_WIDTH, TILE_WIDTH);

// Launch the device computation threads
MatMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd, Width);
```

# Timing the Kernel by gettimeofday()

```
#include <sys/time.h>

double cpuTime() {
   struct timeval tp;
   gettimeofday(&tp, NULL);
   return ( (double)tp.tv_sec + (double)tp.tv_usec * 1.e-6 );
}

double t_start = cpuTime();
/* call your CUDA kernel */
your_kernel<<<grid, block>>>(…);
/* block the host until the GPU completed the kernel function */
cudaDeviceSynchronize();
double kernel_time = cpuTime() – t_start;

printf(“The kernel elapsed %f seconds.\n”, kernel_time);
```

# How Good Is It?

- Testbed: a desktop PC, Windows 7
  - CPU: Intel Core i7-3770 @3.9GHz
  - GPU: Nvidia GT640 with 384 cores @900MHz

- Running time comparison

| Matrix Size | N = 512 | N = 1024 | N = 1536 | N = 2048 |
|---|---|---|---|---|
| GPU time | 18.5ms | 141ms | 470.7ms | 1112ms* |
| CPU time | 181.6ms | 6817ms | 23863ms | 67797ms |
| GPU Flops | 14.5G | 15.2G | 15.4G | 15.4G |
| CPU Flops | 1.48G | 0.31G | 0.30G | 0.25G |
| **Speedup** | **9.8** | **48.3** | **50.7** | **61** |

*Remark: The running time on Nvidia GTX780 for N = 2048 is 112.7ms !

# Is it Good Enough?

- GTX780 has 2304 cores, and its theoretical single-precision computing power is close to 4TFlops
  - But our code only achieves 2x20483/112.7ms = 152.4GFlops
  - i.e., 3.8% of the GPU capacity!
- What is the problem?
  - Hints: GT640's memory bandwidth is 28.5GB/s, GTX780's memory bandwidth is 288GB/s
  - Memory accesses are the bottleneck!

# CGMA: Compute-to-Global-Memory-Access Ratio

```
/* key part of the kernel function */
for (int k = 0; k < Width; ++k)
    Pvalue += Md[Row*Width+k] * Nd[k*Width+Col];
```

- The above kernel function has 2*Width calculations, and 2*Width global memory access
  - Md[] and Nd[] are stored in GPU global memory
  - Pvalue is stored in GPU register, which is very fast
- The CGMA ratio is just 1

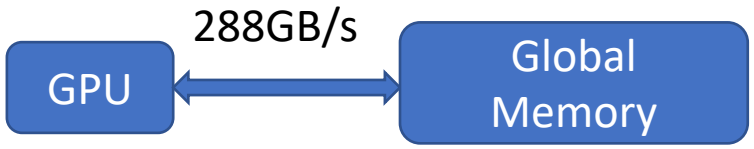
# GPU Global Memory

- GPU global memory has a large size
  - GTX780: 3GB
  - Tesla K40: 12GB

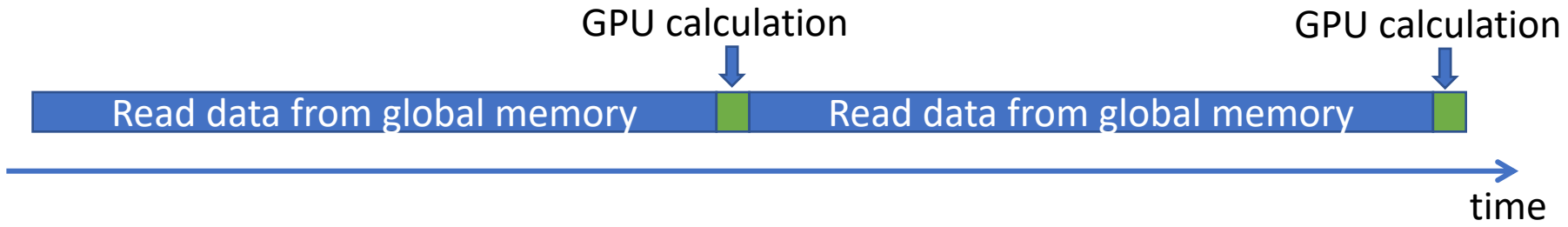
- Bandwidth: how fast can we move data from GPU memory to GPU?
  - GTX640: 28.5GB/s
  - GTX780: 288GB/s
  - Tesla K40: 288GB/s

- Latency:
  - 200-400 GPU cycles

# Memory Is the Bottleneck




- The previous kernel code has a CGMA of 1
- Every calculation requires 1 data access from global memory
- So the performance is limited by the memory bandwidth
  - Most of the time, GPU has nothing to do!



# How to Improve?

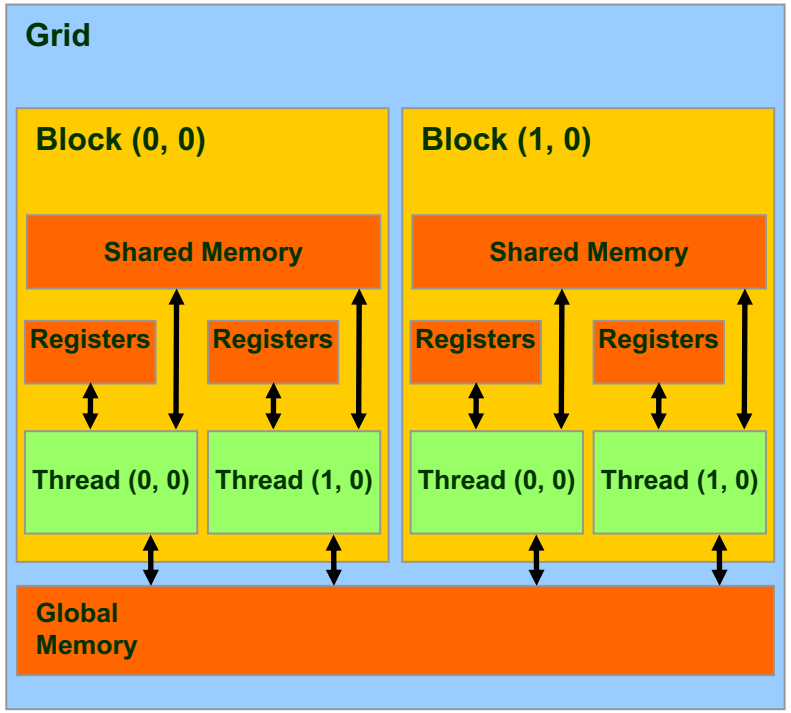
- To make the GPU busy, we need to "create" many calculations for each global memory access
  - I.e., we should achieve a high CGMA ratio

- CPUs use cache to alleviate the memory bottleneck

- GPUs use on-chip "shared memory"
  - Recent GPUs also use cache to improve data access performance
  - Caches are managed by hardware, while "shared memory" can be managed by programmer

# Shared Memory

- Each thread block has a shared memory
  - On-chip
  - very high bandwidth and very low latency
  - Limited size: at most 48KB for compute capability 2.x or above
  - Shared by all threads in the block



# Basic Strategy to Improve CGMA Ratio

- Preparation: Allocate shared memory
- Data loading: load data from global memory to shared memory
- Data processing: reuse the data in shared memory
  - The more calculations you have on each data, the higher CGMA ratio you can achieve
- Jump to Step 2 if not end

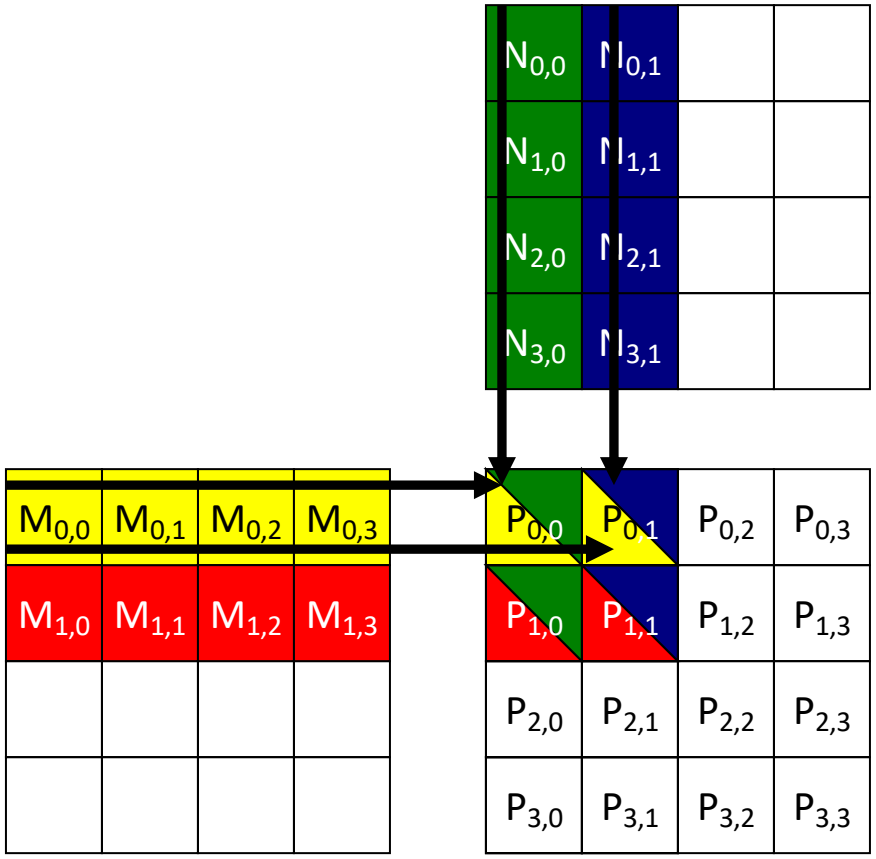
# Revisit Matrix-Matrix Multiplication

- Let's focus on block (0, 0), whose size is 2x2.

Observations:

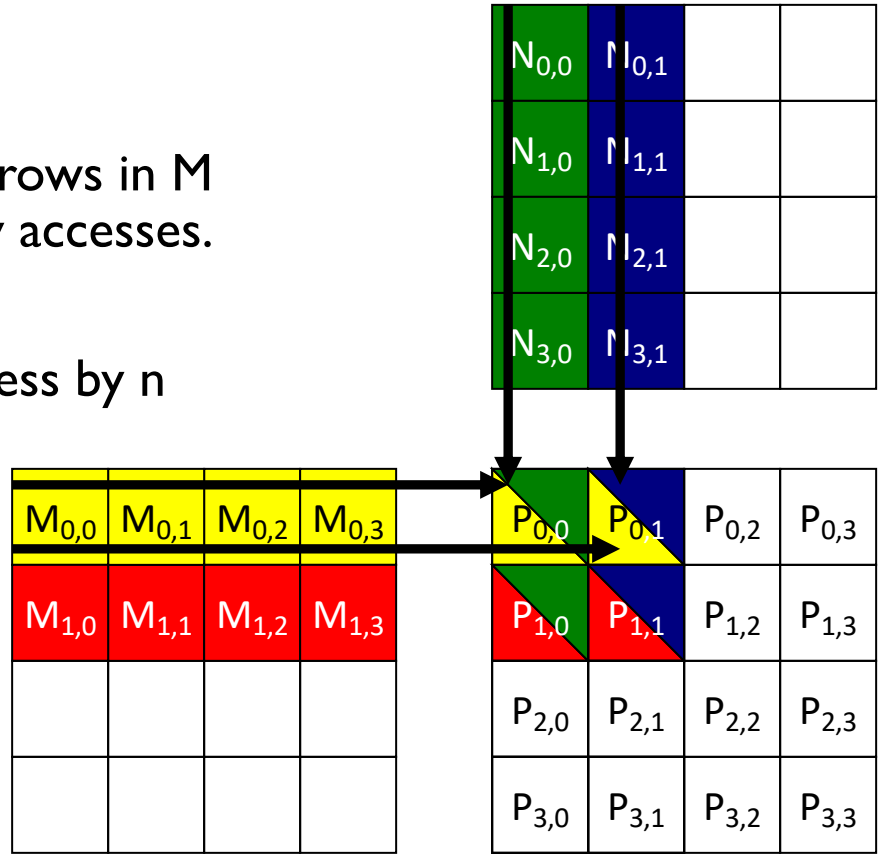
1. $P_{0,0}$ and $P_{0,1}$ both need to access the 1st row of M.
2. $P_{1,0}$ and $P_{1,1}$ both need to access the 2nd row of M.
3. $P_{0,0}$ and $P_{1,0}$ both need to access the 1st column of N.
4. $P_{0,1}$ and $P_{1,1}$ both need to access the 2nd column of N.

- Previous solution:
  - 32 global memory accesses for each block.
  - By using shared memory, we only need 16 global memory accesses for each block.



# Revisit Matrix-Matrix Multiplication

- Imagine the block size is nxn. Matrix size is NxN.
- Previous solution needs $2n^2N$ global memory accesses for each block.
- By using shared memory, we only need to load n rows in M and n columns in N, a total of 2nN global memory accesses.
- We can potentially reduce the global memory access by n times!



# New Challenge

- Shared memory has a limited size.

- It's not possible to load n rows in M and n columns in N to shared memory at once.

- Solution:
  - We divide the n rows and n columns into tiles, such that we can load a tile of M and a tile of N into shared memory
  - We calculate based on the two tiles in shared memory
  - We load another tile of M and another tile of N into shared memory, and repeat

# Load and Compute

| | Phase 1 | | | Phase 2 | | |
|---|---|---|---|---|---|---|
| $T_{0,0}$ | $\mathbf{M_{0,0}}$ ↓ $Ms_{0,0}$ | $\mathbf{N_{0,0}}$ ↓ $Ns_{0,0}$ | $PValue_{0,0}$ += $Ms_{0,0}*Ns_{0,0}$ + $Ms_{0,1}*Ns_{1,0}$ | $\mathbf{M_{0,2}}$ ↓ $Ms_{0,0}$ | $\mathbf{N_{2,0}}$ ↓ $Ns_{0,0}$ | $PValue_{0,0}$ += $Ms_{0,0}*Ns_{0,0}$ + $Ms_{0,1}*Ns_{1,0}$ |
| $T_{0,1}$ | $\mathbf{M_{0,1}}$ ↓ $Ms_{0,1}$ | $\mathbf{N_{0,1}}$ ↓ $Ns_{0,1}$ | $PValue_{0,1}$ += $Ms_{0,0}*Ns_{0,1}$ + $Ms_{0,1}*Ns_{1,1}$ | $\mathbf{M_{0,3}}$ ↓ $Ms_{0,1}$ | $\mathbf{N_{2,1}}$ ↓ $Ns_{0,1}$ | $PValue_{0,1}$ += $Ms_{0,0}*Ns_{0,1}$ + $Ms_{0,1}*Ns_{1,1}$ |
| $T_{1,0}$ | $\mathbf{M_{1,0}}$ ↓ $Ms_{1,0}$ | $\mathbf{N_{1,0}}$ ↓ $Ns_{1,0}$ | $PValue_{1,0}$ += $Ms_{1,0}*Ns_{0,0}$ + $Ms_{1,1}*Ns_{1,0}$ | $\mathbf{M_{1,2}}$ ↓ $Ms_{1,0}$ | $\mathbf{N_{3,0}}$ ↓ $Ns_{1,0}$ | $PValue_{1,0}$ += $Ms_{1,0}*Ns_{0,0}$ + $Ms_{1,1}*Ns_{1,0}$ |
| $T_{1,1}$ | $\mathbf{M_{1,1}}$ ↓ $Ms_{1,1}$ | $\mathbf{N_{1,1}}$ ↓ $Ns_{1,1}$ | $PValue_{1,1}$ += $Ms_{1,0}*Ns_{0,1}$ + $Ms_{1,1}*Ns_{1,1}$ | $\mathbf{M_{1,3}}$ ↓ $Ms_{1,1}$ | $\mathbf{N_{3,1}}$ ↓ $Ns_{1,1}$ | $PValue_{1,1}$ += $Ms_{1,0}*Ns_{0,1}$ + $Ms_{1,1}*Ns_{1,1}$ |

time

# New Kernel Function

```
#define TILE_WIDTH 16
__global__ void MatMulKernel(float* Md, float* Nd, float* Pd, int Width)
{
1.  __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2.  __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];

3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;

7.  float Pvalue = 0;
// Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        // Collaborative loading of Md and Nd tiles into shared memory
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.     Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();

12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
14.     __syncthreads();
    }
15. Pd[Row*Width + Col] = Pvalue;
}
```

# Breakdown of New Kernel Function (1)

```
1. __shared__ float Mds[TILE_WIDTH][TILE_WIDTH];
2. __shared__ float Nds[TILE_WIDTH][TILE_WIDTH];
```

- We use keyword __shared__ to allocate shared memory.
- The same amount of shared memory will be allocated for each thread block.
- One pair of Mds[][] and Nds[][] will be created for each thread block.
- All threads in a block can access the same Mds[][] and Nds[][].

# Breakdown of New Kernel Function (2)

```
3.  int bx = blockIdx.x;  int by = blockIdx.y;
4.  int tx = threadIdx.x; int ty = threadIdx.y;

// Identify the row and column of the Pd element to work on
5.  int Row = by * TILE_WIDTH + ty;
6.  int Col = bx * TILE_WIDTH + tx;
```

- The above statements are used to locate the data in matrix d_P for this thread
- Row is in correspondence with Y-axis
- Col is in correspondence with X-axis

# Breakdown of New Kernel Function (3)

```
7.  float Pvalue = 0;
    // Loop over the Md and Nd tiles required to compute the Pd element
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
      ……
    }
15. Pd[Row*Width + Col] = Pvalue;
```

- Line 7, Pvalue is a "local variable", and is usually stored in GPU register, the fastest memory in GPU.
  - Remark: the total number of registers per SM is limited, so each thread can only use a small number of registers.
  - Register spilling: if your thread has too many local variables, the registers will be used up, and global memory will be used to hold some local variables.
- Line 15, the result Pvalue is copied into global memory Pd[].

# Breakdown of New Kernel Function (4)

```
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
    // Collaborative loading of Md and Nd tiles into shared memory
9.      Mds[ty][tx] = Md[Row*Width + (m*TILE_WIDTH + tx)];
10.     Nds[ty][tx] = Nd[(m*TILE_WIDTH + ty)*Width + Col];
11.     __syncthreads();
        ……
     }
```

- The calculation of Pvalue is divided into Width/TILE_WIDTH phases
- In Lines 9 and 10, each thread loads data from global memory Md[] and Nd[] to shared memory Mds[][] and Nds[][]
  - Each thread block will completely fill up its shared memory Mds[][] and Nds[][]
- Line 11: __syncthreads() is a barrier function, which ensures that all threads in the block have finished loading the data.
  - A block has many threads that need to be scheduled at different time, so different threads may finish Line 10 at different time

# Breakdown of New Kernel Function (5)

```
8.  for (int m = 0; m < Width/TILE_WIDTH; ++m) {
        ……
12.     for (int k = 0; k < TILE_WIDTH; ++k)
13.         Pvalue += Mds[ty][k] * Nds[k][tx];
14.     __syncthreads();
    }
```

- Once the data are ready in shared memory, lines 12-13 carry out the calculations.
  - All operators are in shared memory, so the data accesses become more efficient than the old version.
- Line 14 ensures that all threads in the block finishes the calculation of the current phase.
  - So the shared memory can be reused for the next phase.

# Results of the New Kernel

- On GTX640:

| Matrix Size | N = 512 | N = 1024 | N = 1536 | N = 2048 |
|---|---|---|---|---|
| Without shared memory | 18.5ms | 141ms | 470.7ms | 1112ms |
| With shared memory | 7ms | 50.7ms | 172.3ms | 394.6ms |
| **Speedup** | **2.64** | **2.78** | **2.73** | **2.82** |

# Unrolling the For Loop

```
#pragma unroll
for (int k = 0; k < TILE_WIDTH; ++k)
    Pvalue += Mds[ty][k] * Nds[k][tx];
```

- Without unrolling, many additional instructions are used for each multiplication and addition

- By unrolling, the "for loop" is replaced by "TILE_WIDTH" number of statements

| Matrix Size | N = 512 | N = 1024 | N = 1536 | N = 2048 |
|---|---|---|---|---|
| Without unrolling | 7ms | 50.7ms | 172.3ms | 394.6ms |
| With unrolling | 5.5ms | 39.1ms | 131.9ms | 303.3ms |
| **Speedup** | **1.27** | **1.30** | **1.31** | **1.30** |

# Memory can Limit Parallelism

- Registers and shared memory are good, but limited in size.

- If each thread uses a lot of registers and/or shared memory, the number of "active threads" will be reduced, which may lead to poor performance.

- E.g., in Compute Capability 2.x, each SM has 32,768 registers and can support at most 1,536 active threads
  - If each thread uses 21 registers, you can achieve the maximum 1,536 active threads (1,536 x 21 < 32,768)
  - If each thread uses 22 registers, you can have at most 1,489 threads. If you block size is 512, then you can only have TWO active blocks, or 1024 active threads.

# Hardware Limitations

| Compute Capability | 1.1 | 1.2, 1.3 | 2.x | 3.0 | 3.5 | 5.0 | 7.0 |
|---|---|---|---|---|---|---|---|
| Number of 32-bit registers per SM | 8K | 16K | 32K | 64K | 64K | 64K | 64K |
| Max. number of 32-bit registers per thread | 128 | 128 | 63 | 63 | 255 | 255 | 255 |
| Max. amount of shared memory per SM | 16KB | 16KB | 48KB | 48KB | 48KB | 64KB | 96KB |
| Max. amount of shared memory per thread block | 16KB | 16KB | 48KB | 48KB | 48KB | 48KB | 48KB |

# APOD Motivation

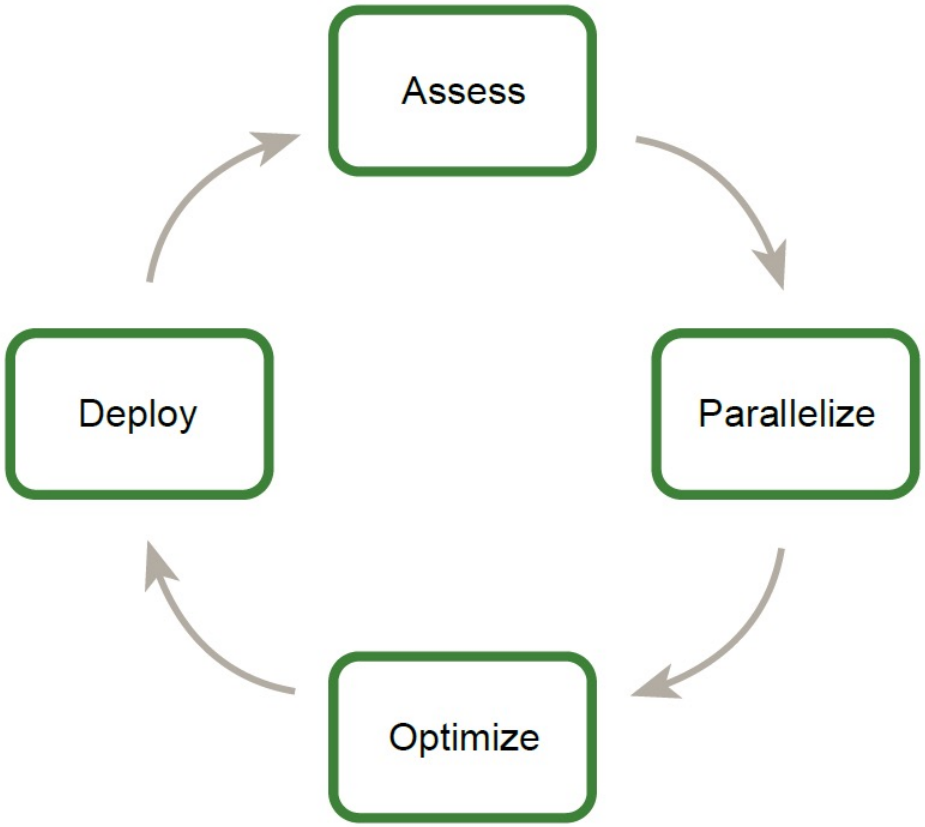
- GPU consists of many hardware resources, each with different capacities
  - Arithmetic units:
    - Throughput: how many calculations per unit time?
    - Latency: How long for each calculation?
  - Memory system:
    - Bandwidth/throughput: how many bypes per unit time?
    - Latency: How long for each data access?
- The speed of a CUDA kernel greatly depends on the resource constraint of the GPU device.

- We need to understand the major types of resource constraints in GPUs in order to develop highly efficient CUDA programs.
  - E.g., in Part II, "global memory" can be the bottleneck and we use "shared memory" to improve.

- APOD: Assess, Parallelize, Optimize, and Deploy

# APOD Design Cycle

- APOD: Assess, Parallelize, Optimize, and Deploy



# I. Assess

- The first step is to assess the application to locate the hotspots, i.e., parts of the code that are responsible for the bulk of the execution time.
  - Through theoretical analysis: time complexity analysis for each major step of your application
  - Through profiling: profilers are software tools that measure the time consumptions of function calls or instructions. E.g.,
    - GNU gprof for Linux
      - https://www.cs.utah.edu/dept/old/texinfo/as/gprof_toc.html
    - For CUDA development:
      - nvprof for Linux
      - Nvidia Visual Profiler for Windows
      - Ref: http://docs.nvidia.com/cuda/profiler-users-guide

$ **nvprof ./matrix**

```
==19300== NVPROF is profiling process 19300, command: ./matrix
CUDA initialized.
GPU done!
Elaspsed Time by event: 16.180511 ms
GPU (shared memory) done!
Elaspsed Time by event: 6.920672 ms
Elaspsed Time by CPU: 1458.828125 ms
==19300== Profiling application: ./matrix
==19300== Profiling result:
Time(%)     Time     Calls       Avg       Min       Max  Name
 59.04%  13.146ms         1  13.146ms  13.146ms  13.146ms  MatrixMulKernel(float*, float*, float*, int)
 20.27%  4.5129ms         1  4.5129ms  4.5129ms  4.5129ms  SharedMatrixMulKernel(float*, float*, float*, int)
 11.90%  2.6503ms         4  662.59us  659.75us  669.44us  [CUDA memcpy HtoD]
  8.79%  1.9560ms         2  978.02us  749.31us  1.2067ms  [CUDA memcpy DtoH]

==19300== API calls:
Time(%)     Time     Calls       Avg       Min       Max  Name
 77.83%  228.31ms         2  114.16ms  1.7350us  228.31ms  cudaEventCreate
 13.99%  41.047ms         1  41.047ms  41.047ms  41.047ms  cudaDeviceReset
  7.85%  23.016ms         6  3.8360ms  588.40us  14.740ms  cudaMemcpy
  0.10%  302.22us       166  1.8200us     164ns  61.213us  cuDeviceGetAttribute
  0.09%  257.85us         3  85.949us  63.839us  128.96us  cudaMalloc
  0.06%  186.60us         3  62.198us  46.782us  92.858us  cudaFree
  0.02%  52.845us         2  26.422us  26.248us  26.597us  cudaLaunch
  0.02%  46.251us         2  23.125us  22.843us  23.408us  cuDeviceTotalMem
  0.01%  34.460us         2  17.230us  16.290us  18.170us  cuDeviceGetName
  0.01%  33.645us         6  5.6070us  1.6740us  14.791us  cudaEventRecord
  ……
```

# II. Parallelize

- After identifying the hotspots, try to parallelize them
  - You can use existing parallel libraries, such as cuBLAS, cuFFT, etc.
  - Or, you can design parallel algorithms by yourself

# III. Optimize

- How to implement your parallel algorithm on a specific hardware to achieve its best performance?

- Program optimization is a challenging task
  - Fully understand your application
  - Fully understand your hardware
  - Go through the APOD cycle for many rounds
  - An optimized implementation could improve the performance by 10x

# CUDA Optimization Techniques

- Kernel optimizations
  - Kernel configurations: # of blocks, # of threads per block
  - Control flow
  - Global memory access
  - Shared memory access
  - Instruction optimization

- CPU-GPU interaction optimizations
  - Maximize PCI-e throughput
  - Overlapping kernel execution with memory copies

# Kernel Configuration

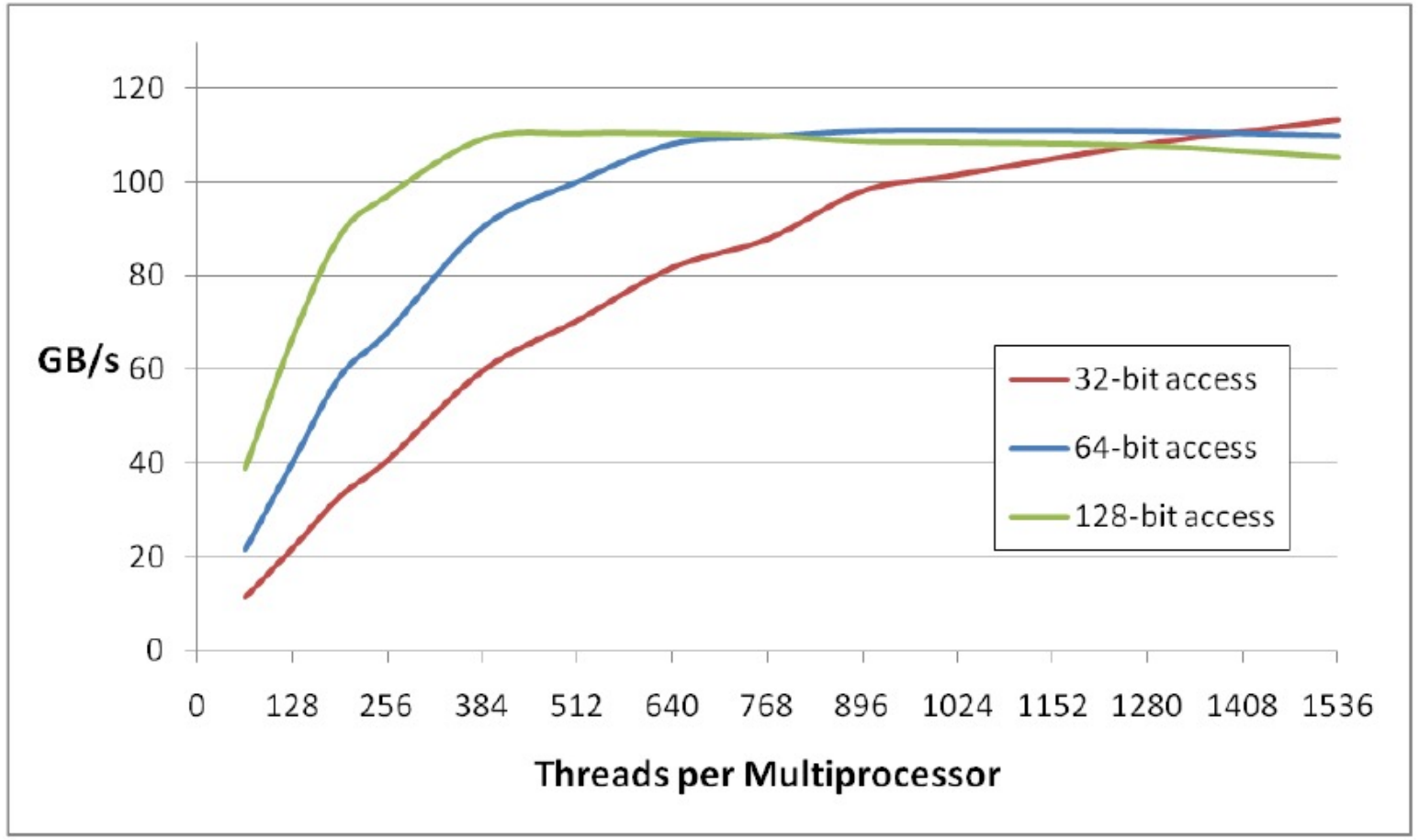
- How many blocks/threads to launch?

- Key to understanding:
  - Instructions are issued in order
  - A thread stalls when one of the operands isn't ready
  - Latency is hidden by switching threads
    - Global memory latency: several hundred cycles
    - Arithmetic latency: 18-22 cycles

- Conclusion:
  - Need enough active warps per SM to hide latency

# Global Memory Bandwidth

**Tesla C2050, ECC on, theoretical bandwidth: ~120 GB/s**

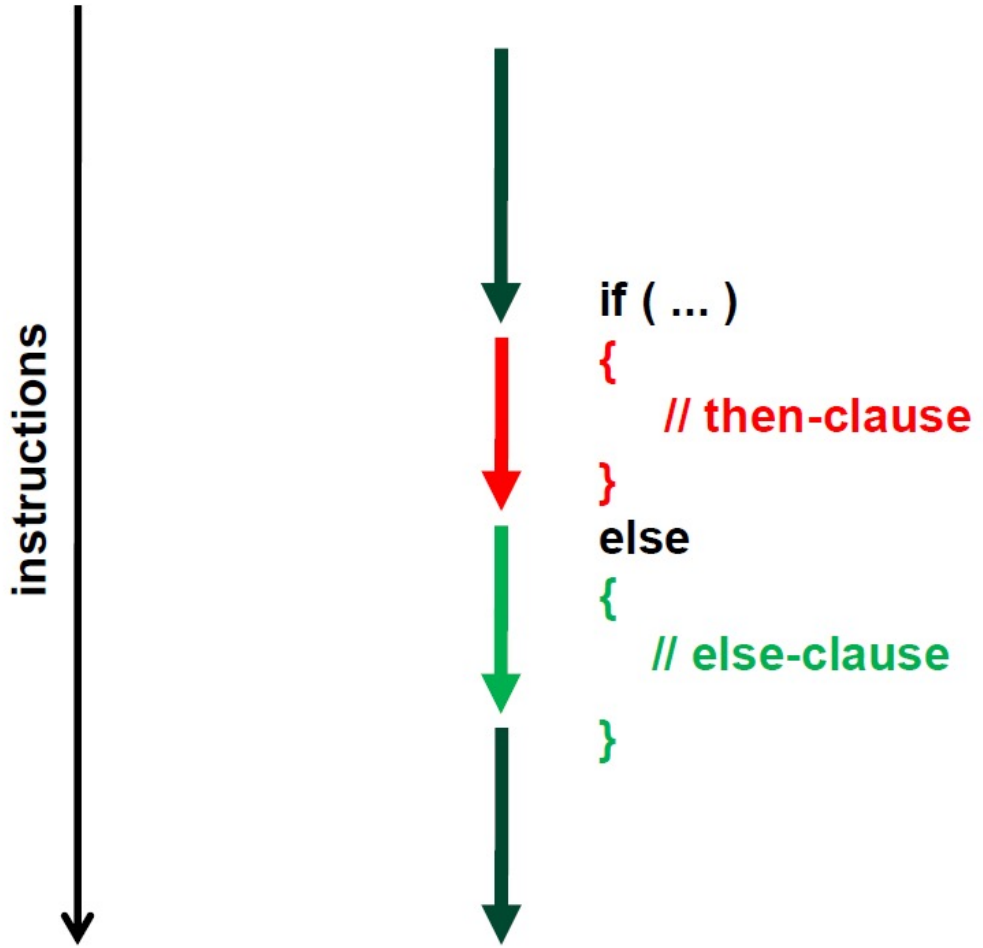


# Recommendations

- Need enough warps to keep GPU busy
  - # of blocks should be much larger than # of SMs
  - Typically, at least 16 active warps per SM

- Thread block configuration
  - Threads per block should be a multiple of warp size (32)
  - Usually 128-256 threads/block are good

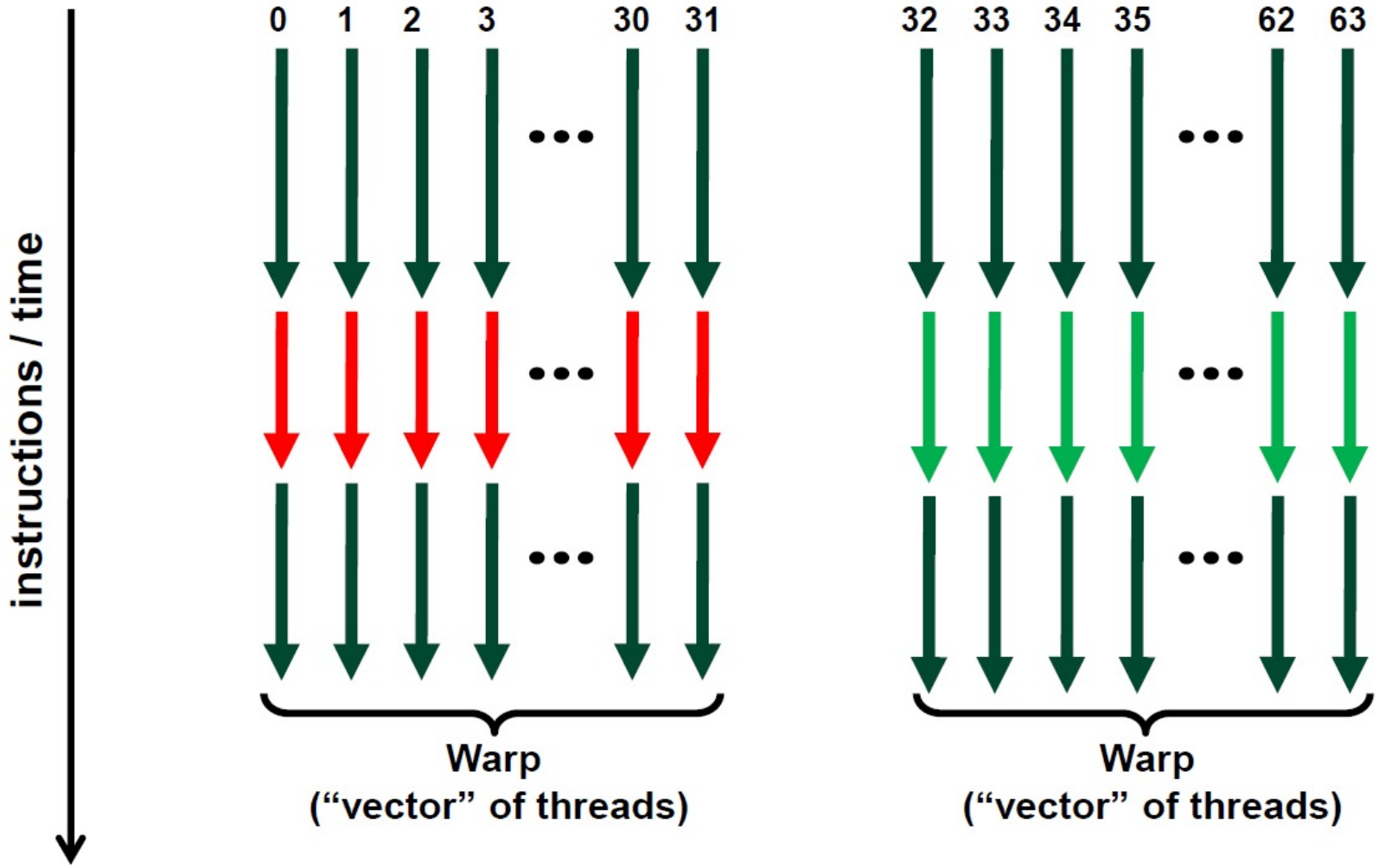
# Control Flow



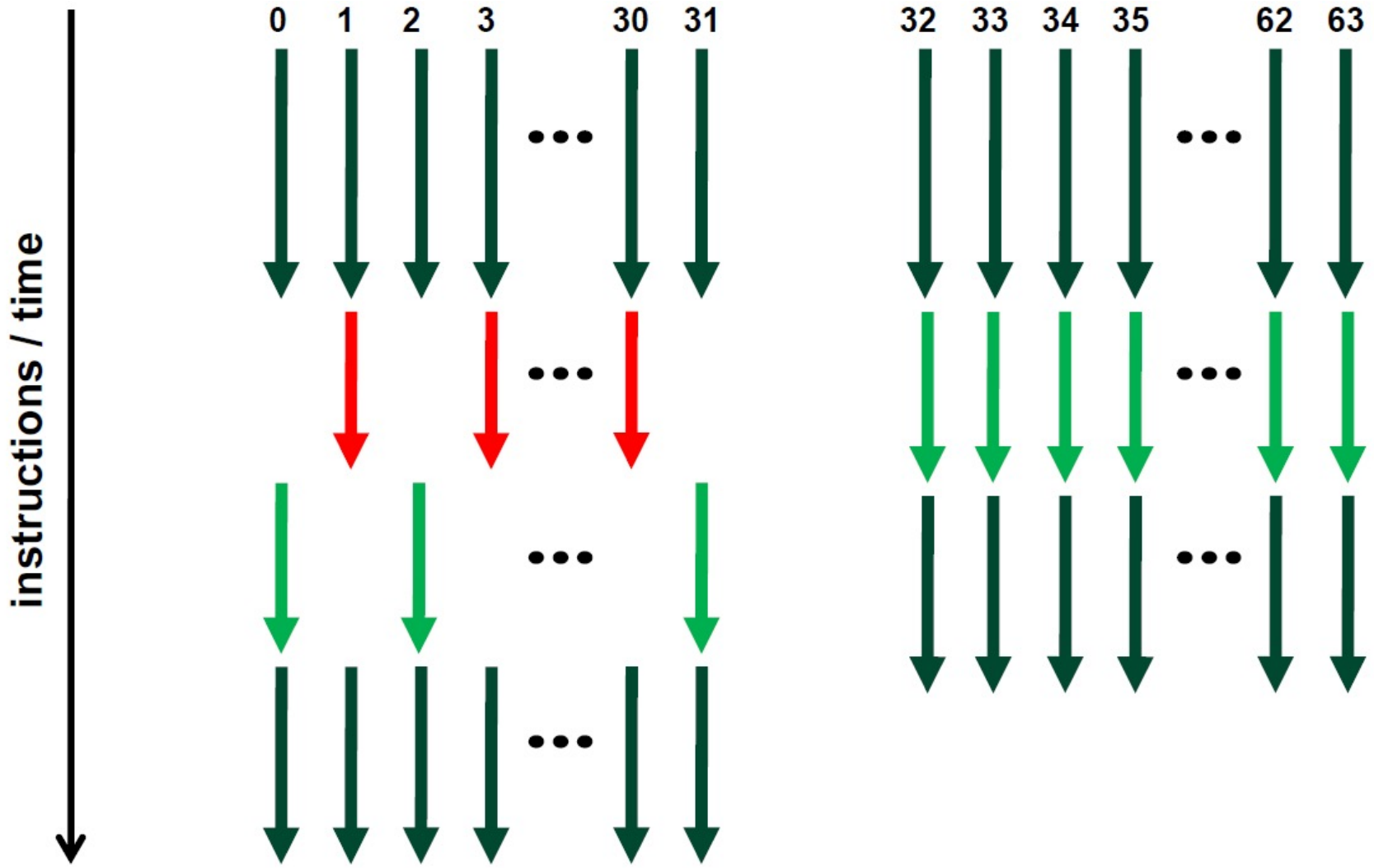
# Control Flow Instructions

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

# Execution within Warps is Coherent



# Execution Diverges within a Warp



# Parallel Reduction

- Given an array of values, "reduce" them to a single value in parallel

- Examples
  - Sum reduction: sum of all values in the array
  - Max reduction: maximum of all values in the array

- Typical parallel implementation:
  - Recursively halve # threads, add two values per thread
  - Takes log(n) steps for n elements, requires n/2 threads

# A Vector Reduction Example

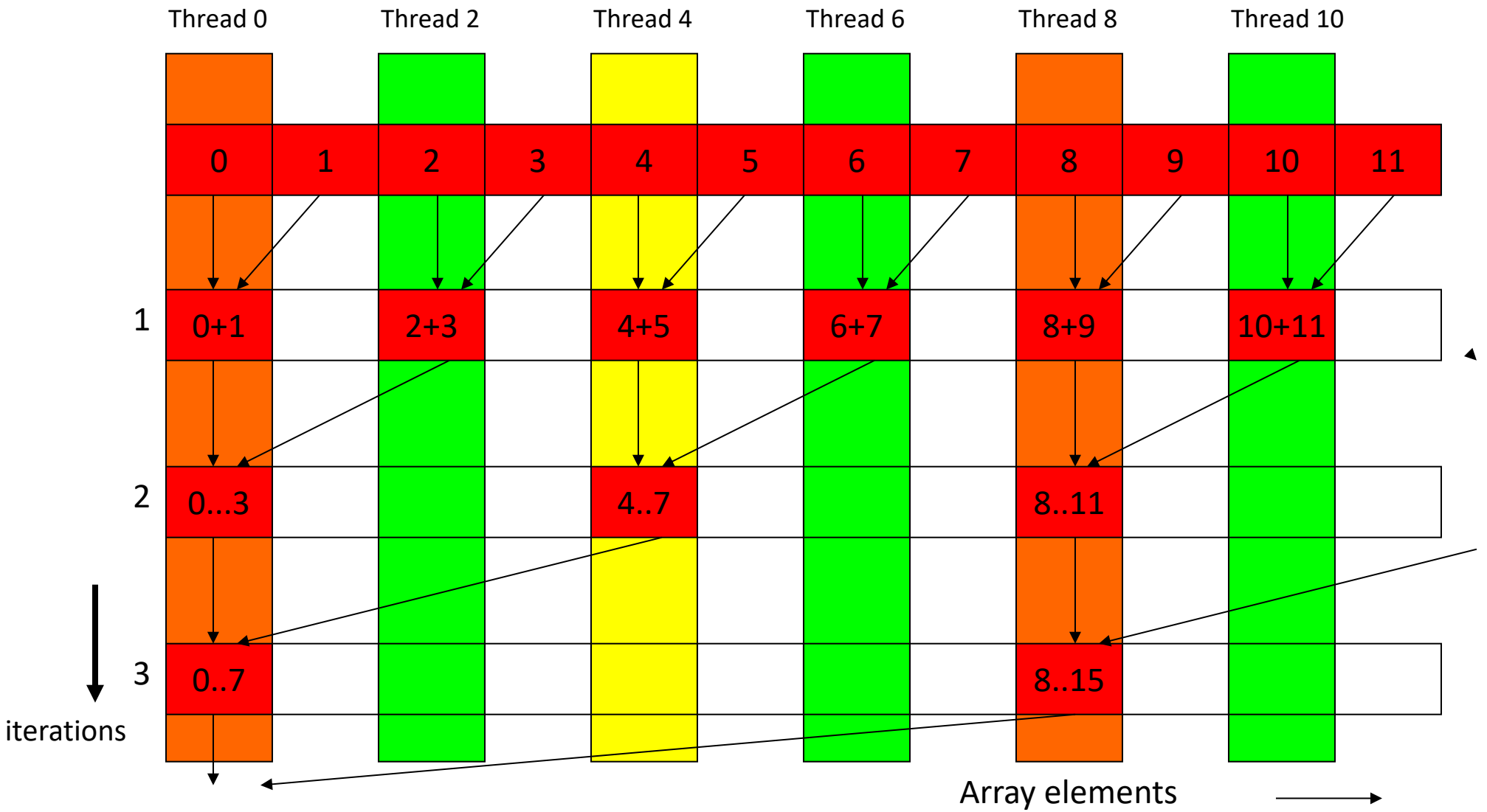
- Assume an in-place reduction using shared memory
  - The original vector is in device global memory
  - The shared memory is used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final solution will be in element 0

# A Simple Implementation

- Assume we have already loaded array into shared memory

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

# Vector Reduction with Branch Divergence



# Some Observations

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

- No more than half of threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations (512/32=16= 24), where each iteration only has one thread activated until all warps retire

# Shortcomings of the Implementation

- Assume we have already loaded array into shared memory

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

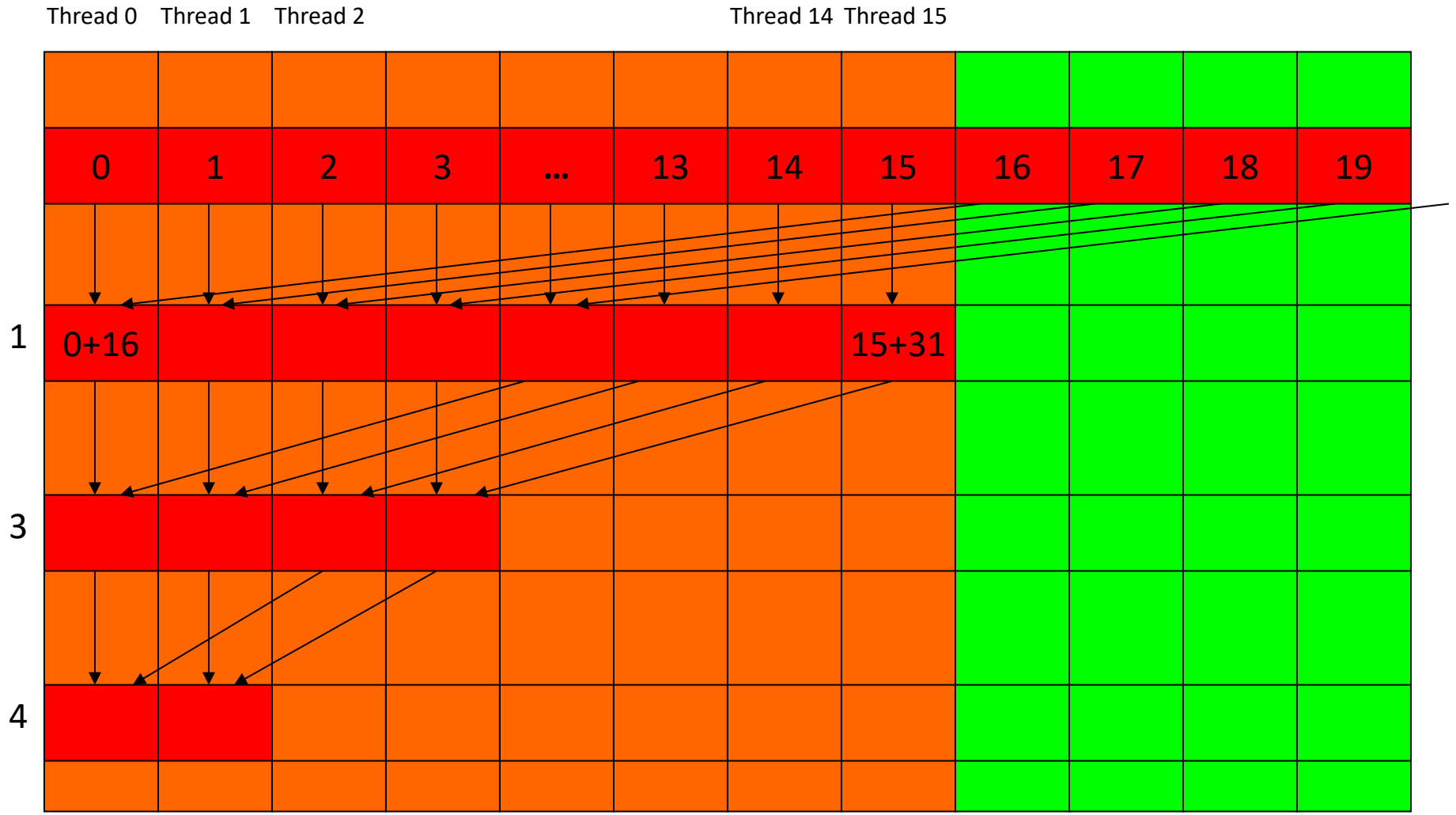
**BAD: Divergence due to interleaved branch decisions**

# A Better Implementation

- Assume we have already loaded array into shared memory

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

# No Divergence until < 16 Sub-sums



# CUDA Device Memory Space: Review

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

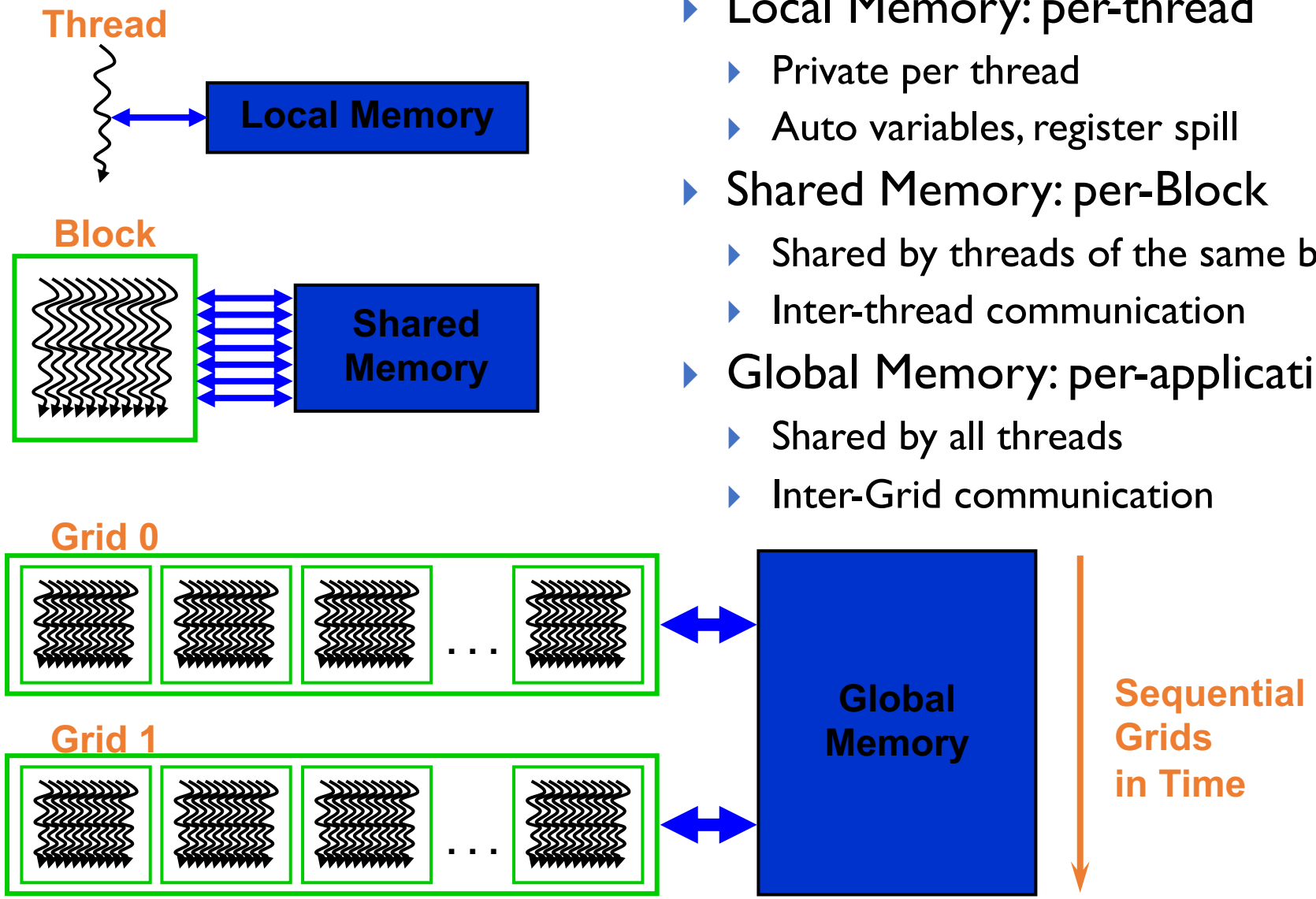
| Declaration | Memory |
|---|---|
| int v | Register |
| int vArray[10] | Local |
| __shared__ int sharedV | Shared |
| __device__ int globalV; or cudaMalloc() | Global |
| __constant__ int constantV[10]; and cudaMemcpyToSymbol(...); | Constant |
| cudaBindTexture2D structtextureReference | Texture |

| Memory | Location on/off chip | Cached | Access | Scope | Lifetime |
|---|---|---|---|---|---|
| Register | On | n/a | R/W | 1 thread | Thread |
| Local | Off | Yes†† | R/W | 1 thread | Thread |
| Shared | On | n/a | R/W | All threads in block | Block |
| Global | Off | † | R/W | All threads + host | Host allocation |
| Constant | Off | Yes | R | All threads + host | Host allocation |
| Texture | Off | Yes | R | All threads + host | Host allocation |

† Cached in L1 and L2 by default on devices of compute capability 6.0 and 7.x; cached only in L2 by default on devices of lower compute capabilities, though some allow opt-in to caching in L1 as well via compilation flags.

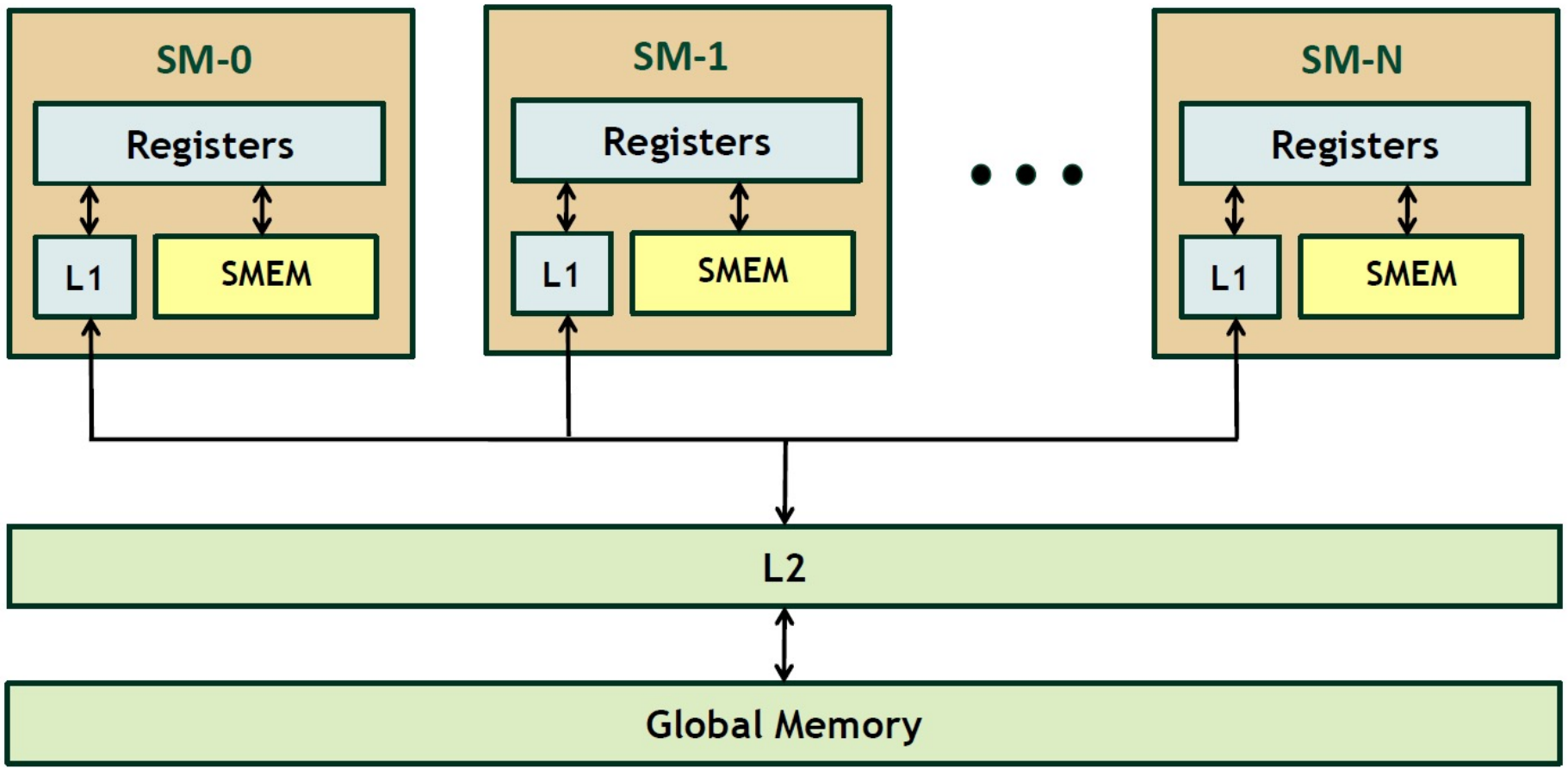
†† Cached in L1 and L2 by default except on devices of compute capability 5.x; devices of compute capability 5.x cache locals only in L2.

# Parallel Memory Sharing




- Local Memory: per-thread
  - Private per thread
  - Auto variables, register spill
- Shared Memory: per-Block
  - Shared by threads of the same block
  - Inter-thread communication
- Global Memory: per-application
  - Shared by all threads
  - Inter-Grid communication

# GPU Memory Hierarchy



# Global Memory Coalescing

- Memory operations are issued per warp (32 threads)
  - One instruction, 32 data accesses
  - How to satisfy these 32 data access requests?

- Coalescing:
  - Global memory loads and stores by threads of a warp (of a half warp for devices of compute capability 1.x) are coalesced into as few as one transaction when certain access requirements are met.

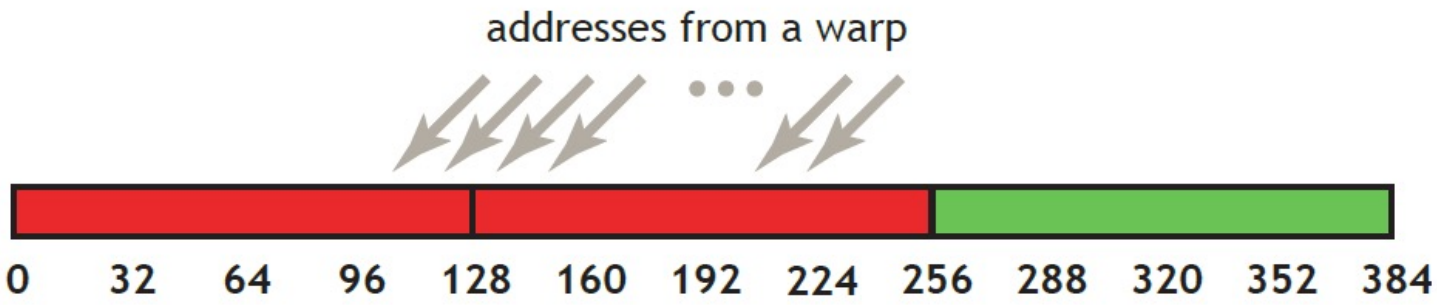
# Global Memory Coalescing

- Global memory accesses for devices of compute capability 2.x are cached in L1 (128-byte lines) by default
  - The basic global memory transaction is to read/write a continuous 128-byte segment
  - Each memory request from a warp is broken down into cache line requests

- Global memory accesses for devices of compute capability 3.x and 5.x are cached in L2 (32-byte segments)
  - The basic global memory transaction is to read/write a continuous 32-byte segment

# “BAD” Access Patterns

- Mis-aligned data accesses



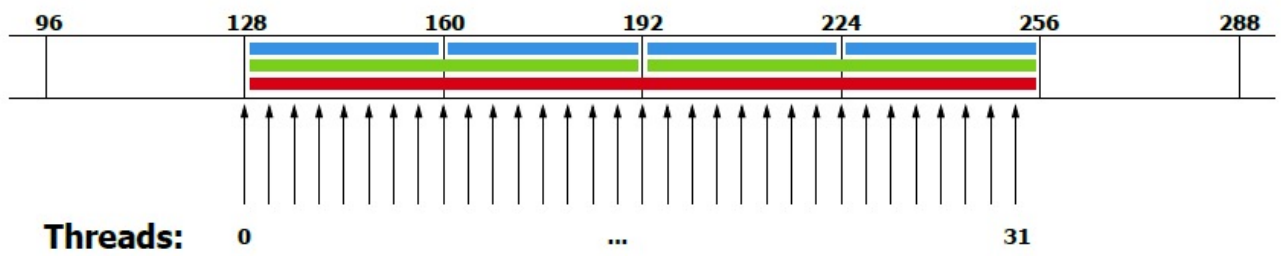

Mis-aligned sequential addresses that fit into two 128-byte L1-cache lines

From Ref. [3]

# Example 1

**4-Byte Word per Thread**

**Aligned and sequential**



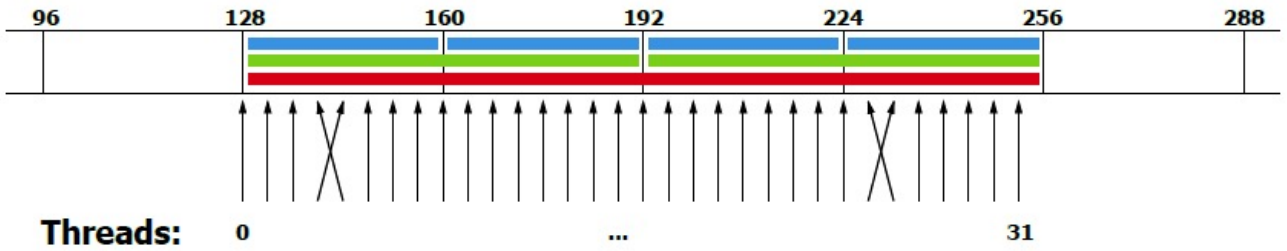

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.x and 3.x | |
|---|---|---|---|---|
| Memory transactions: | Uncached | | Uncached | Cached |
| | 1x 64B at 128<br>1x 64B at 192 | 1x 64B at 128<br>1x 64B at 192 | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224 | 1x 128B at 128 |

Cached: L1 (CC 2.x)
Uncached: no cache (CC 1.x), or L2 (CC 3.x or above)

From Ref. [2]

# Example 2

**Aligned and non-sequential**



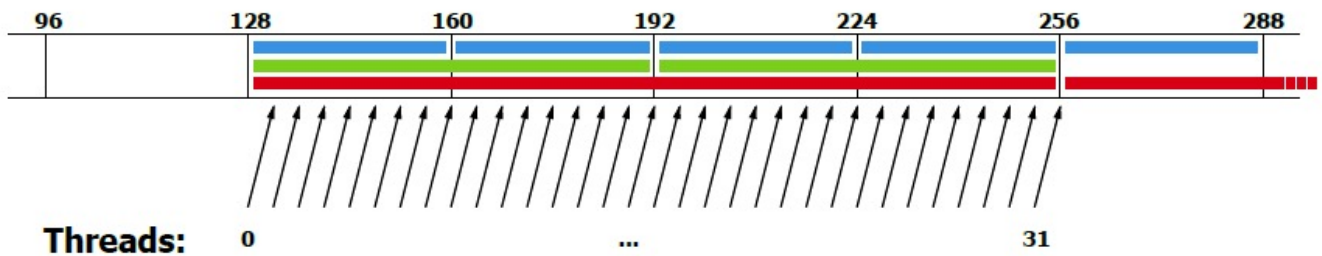

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.x and 3.x | |
|---|---|---|---|---|
| Memory transactions: | Uncached | | Uncached | Cached |
| | 8x 32B at 128<br>8x 32B at 160<br>8x 32B at 192<br>8x 32B at 224 | 1x 64B at 128<br>1x 64B at 192 | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224 | 1x 128B at 128 |

For CC 1.0 and 1.1, the performance of non-sequential data access is very poor.

From Ref. [2]

# Example 3

**Mis-aligned and sequential**



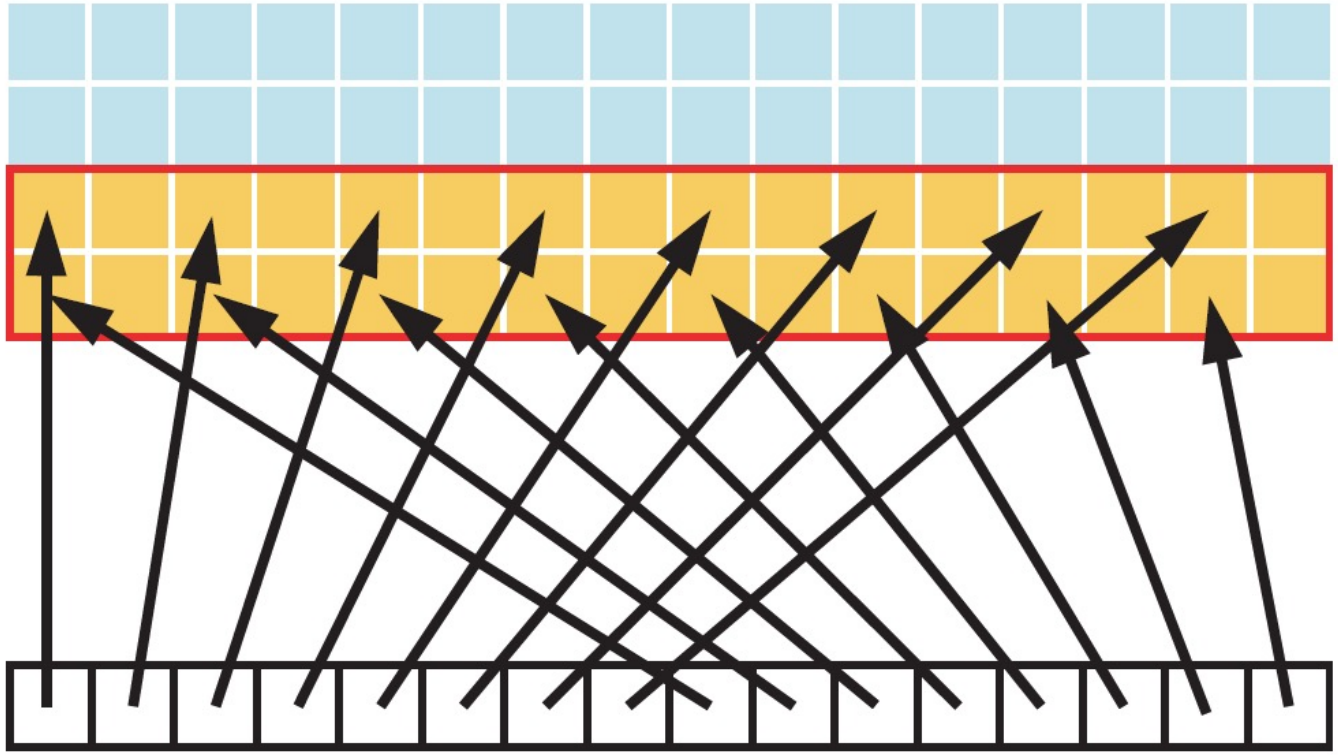

| Compute capability: | 1.0 and 1.1 | 1.2 and 1.3 | 2.x and 3.x | |
|---|---|---|---|---|
| Memory transactions: | Uncached | | Uncached | Cached |
| | 7x 32B at 128<br>8x 32B at 160<br>8x 32B at 192<br>8x 32B at 224<br>1x 32B at 256 | 1x 128B at 128<br>1x 64B at 192<br>1x 32B at 256 | 1x 32B at 128<br>1x 32B at 160<br>1x 32B at 192<br>1x 32B at 224<br>1x 32B at 256 | 1x 128B at 128<br>1x 128B at 256 |

Mis-aligned data access: the starting address is not a multiple of 32 or 128. Some bytes in the cache line are not useful.

From Ref. [2]

# Example 4: Strided Access

Adjacent threads accessing memory with a stride of 2



A stride of 2 results in a 50% of load/store efficiency since half the elements in the transaction are not used and represent wasted bandwidth.

From Ref. [3]

# Shared Memory

- Shared memory are on-chip
  - High bandwidth and low latency
  - Not as good as registers, but much better than global memory

- Uses:
  - Inter-thread communication within a block
  - Cache data to reduce redundant global memory accesses
  - Use it to improve global memory access patterns

- Organization:
  - Divided into equally sized memory modules, named banks
  - Successive 4-byte or 8-byte words belong to different banks

# Shared Memory Banks

- Compute Capability 1.x
  - 16 banks, 4-byte wide
- Compute Capability 2.x
  - 32 banks, 4-byte wide
- Compute capability 3.x
  - 32-bit mode: 32 banks, 4-byte wide
  - 64-bit mode: 32 banks, 8-byte wide
- Compute capability 5.x
  - 32 banks, 4-byte wide
- Compute capability 7.x
  - 32 banks, 4-byte wide

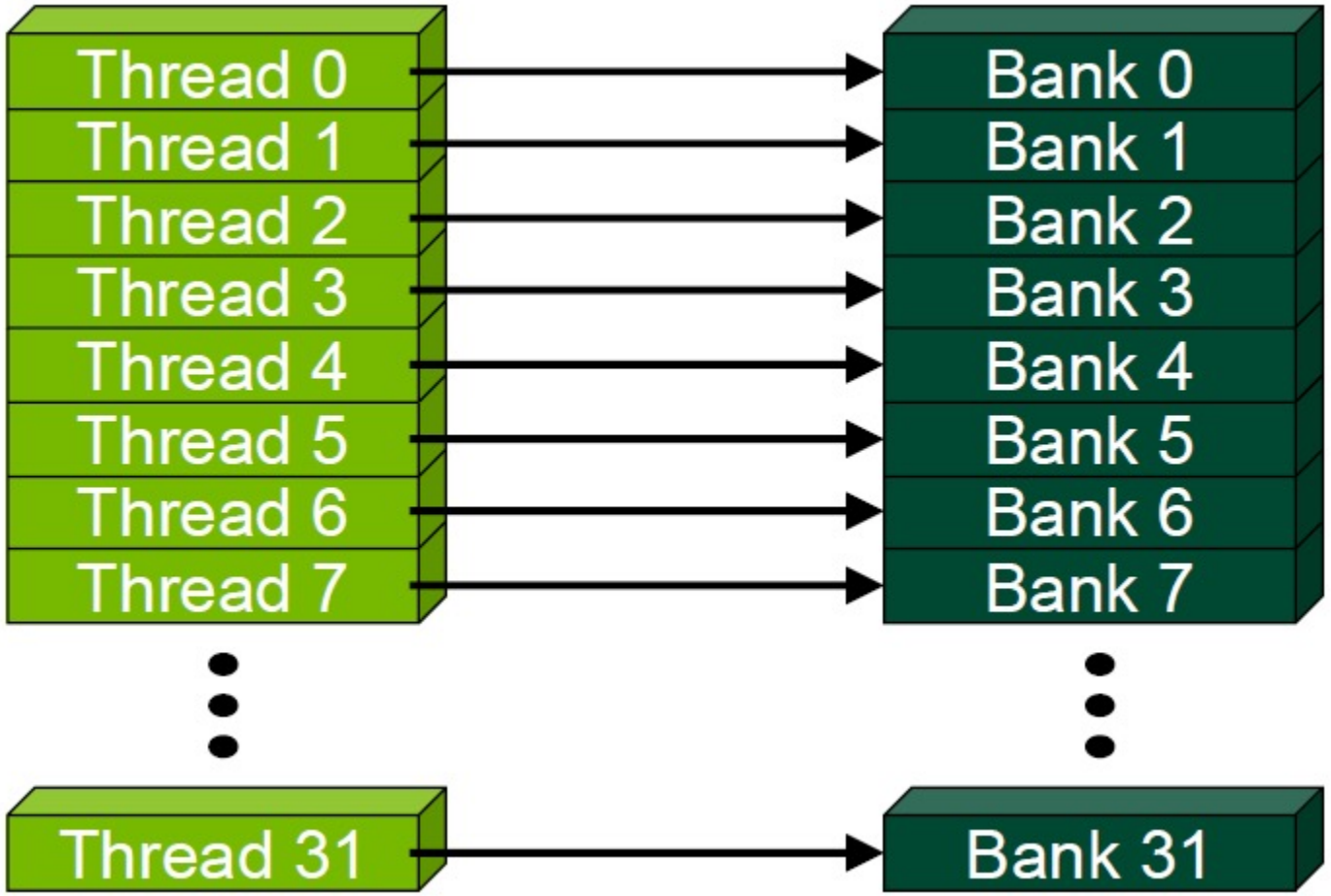
# Bank Conflict

- Shared memory accesses are issued per 32 threads (warp)

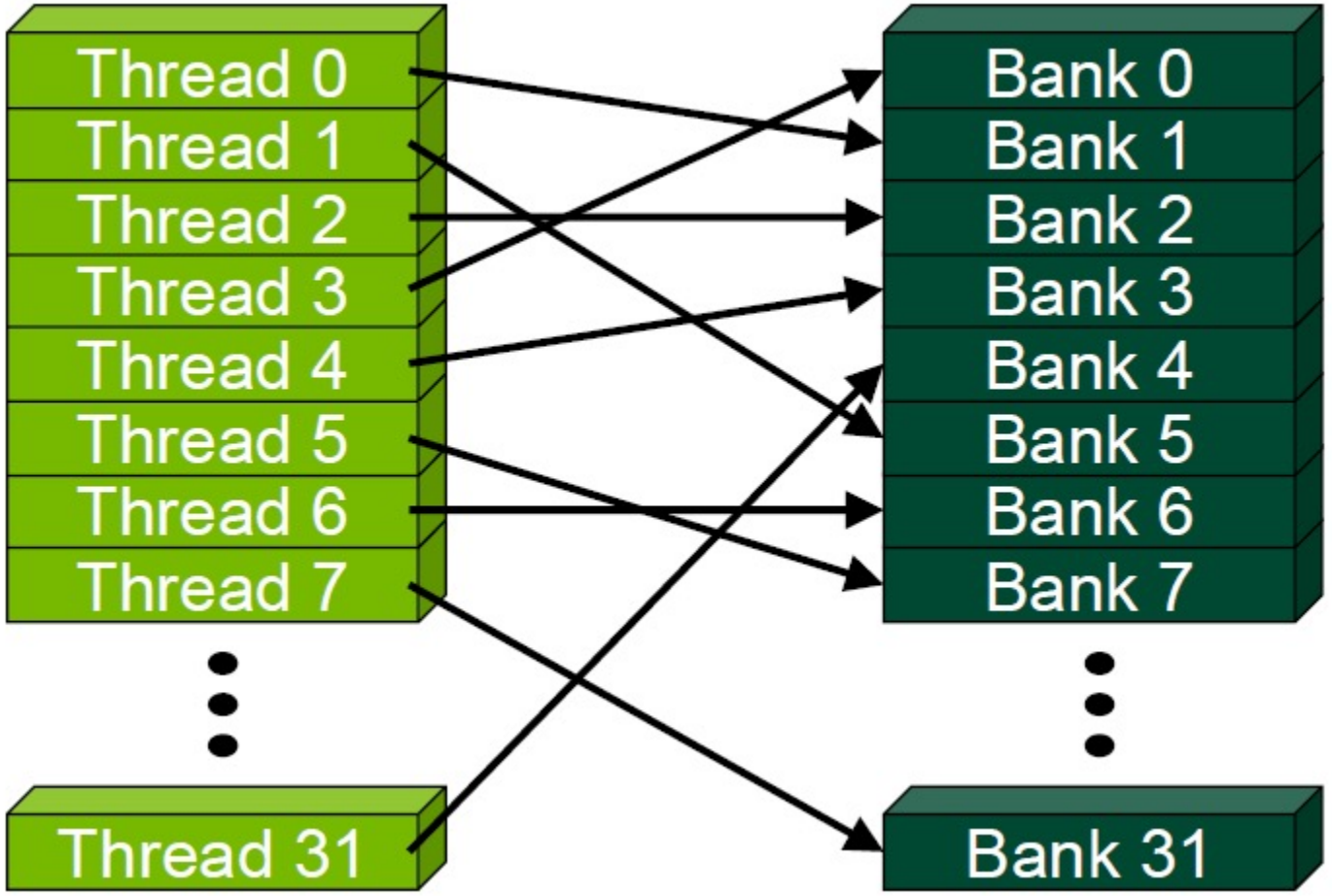
- Serialization: bank conflict
  - if n threads in a warp access different words in the same bank, n shared memory accesses are executed serially

  - We should avoid bank conflict as much as possible.

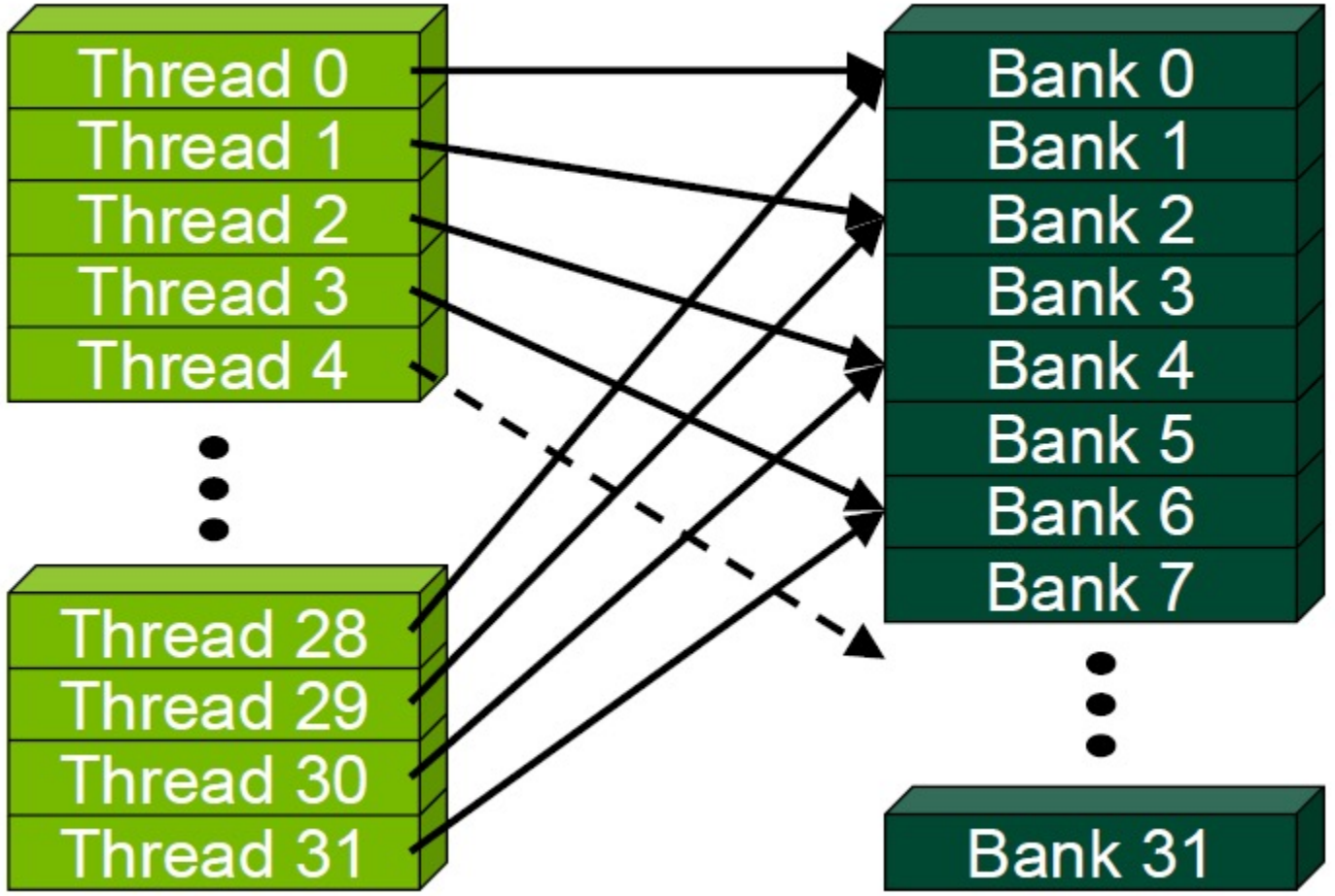
# Example 1: No Bank Conflict



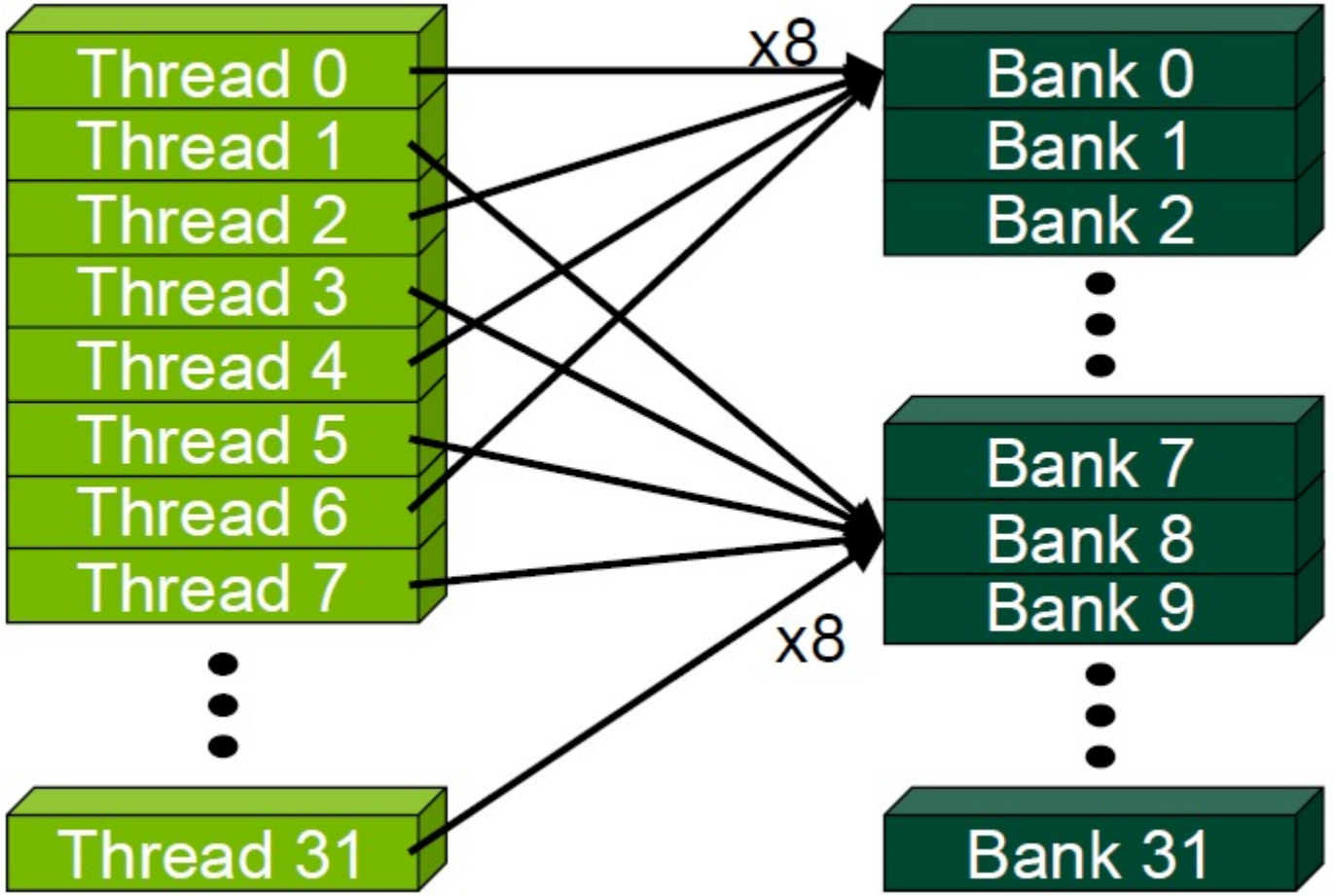
# Example 2: No Bank Conflict



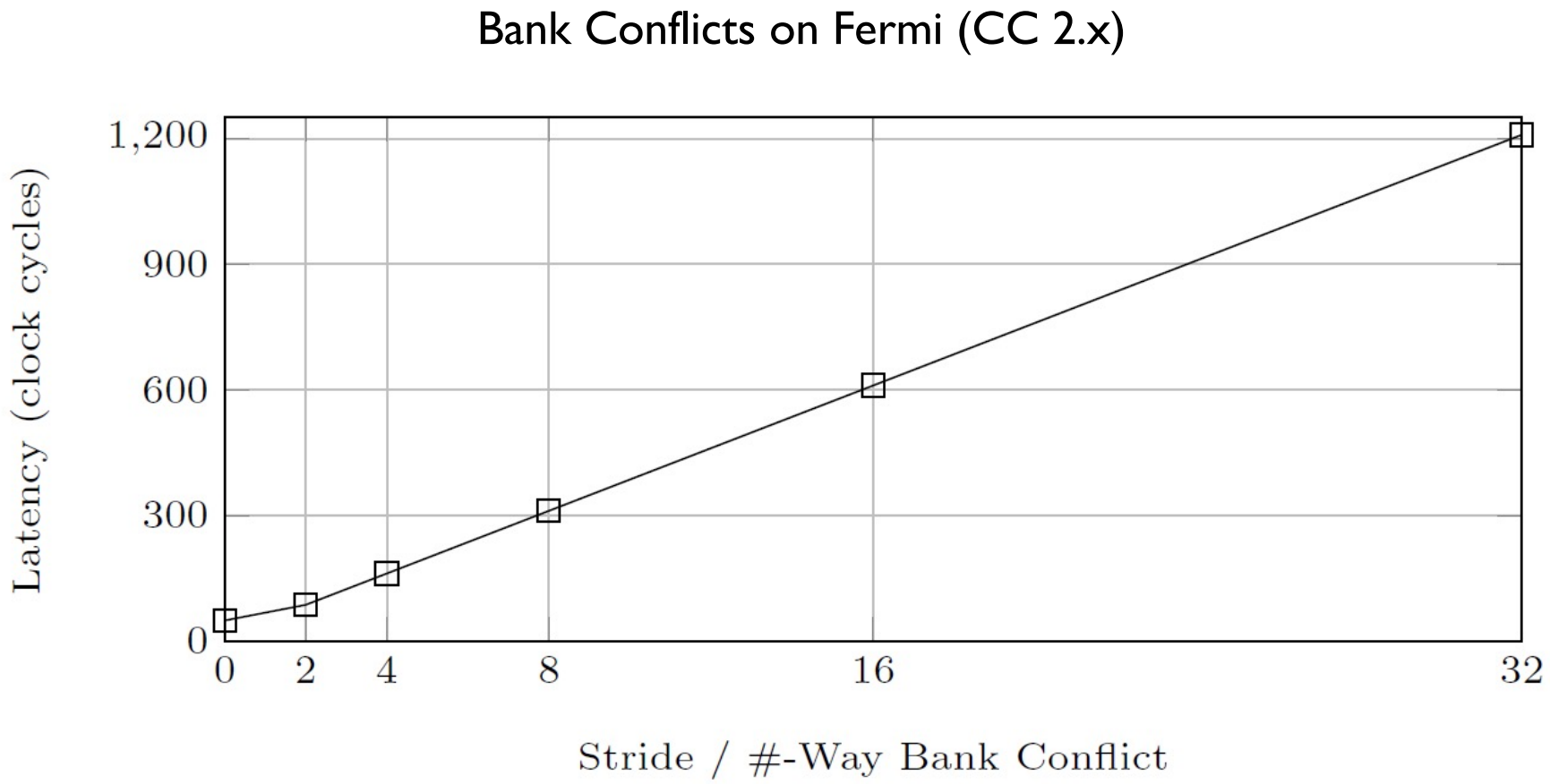
# Example 3: 2-way Bank Conflict



# Example 4: 8-way Bank Conflict



# Latency of Bank Conflicts

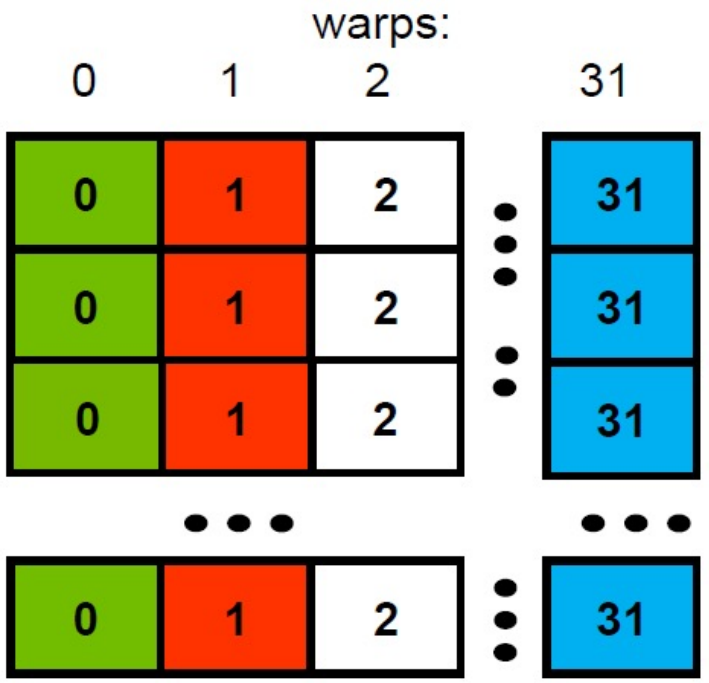



From Ref. [4]

# Avoiding Bank Conflicts

- 32x32 shared memory array

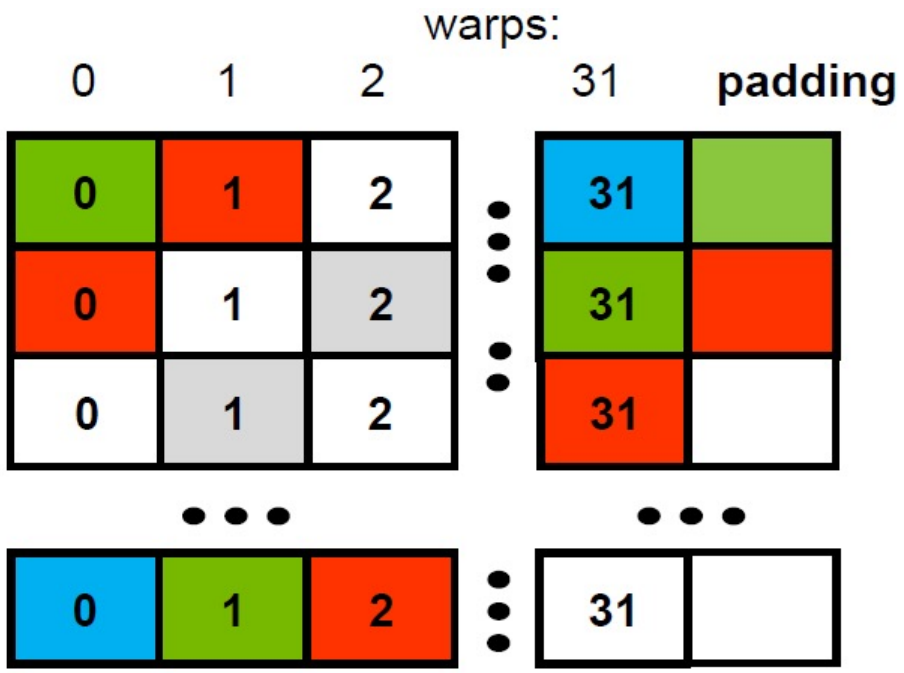
- If each warp accesses a column
  - 32-way bank conflicts

# Avoiding Bank Conflicts

- Add a column for padding
  - 32 x 33 shared memory array
  - No bank conflict now!

Bank 0
Bank 1
…
Bank 31



# References


1. David B. Kirk and Wen-mei W. Hwu, Programming Massively Parallel Processors, 2nd Edition, Morgan Kaufmann, 2013.

2. CUDA C Programming Guide, Nvidia. http://docs.nvidia.com/cuda/cuda-c-programming-guide/

3. CUDA C BEST PRACTICES GUIDE, Nvidia. http://docs.nvidia.com/cuda/cuda-c-best-practices-guide

4. X. Mei and X.-W. Chu, "Dissecting GPU memory hierarchy through microbenchmarking," IEEE TPDS 2017.