# CS 314 Principles of Programming Languages

## Lecture 24: CUDA Programming

Prof. Zheng Zhang

*Rutgers University*

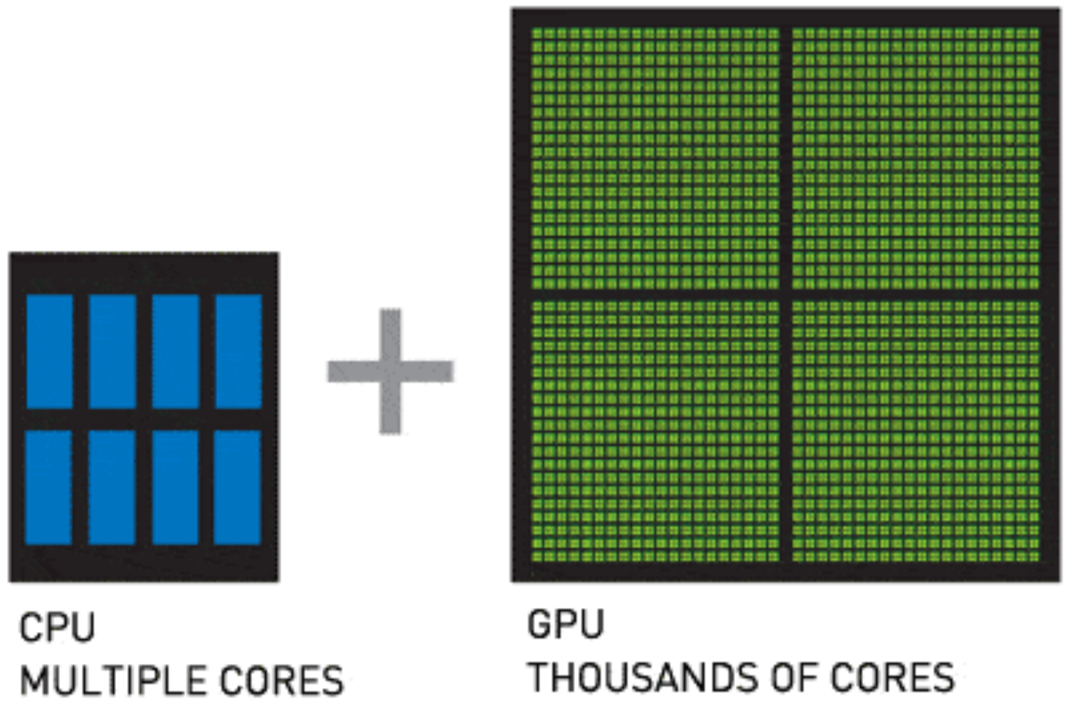
December 5, 2018

## Class Information

- Project 3 released: deadline 12/12.
- Homework 8 released: deadline 12/10.
- Final exam coverage:
  Lecture 1 - 21, HW 1-8, Recitation 1-12, Corresponding Book Chapters.
- Final exam: 12/19 4pm — 7pm.

## GPU Programming

- General Purpose Graph Processing Unit (GPU)
- Reflects the trend of multi-core scaling in post Moore's law era
- Application domain: machine learning, scientific simulation, weather prediction, computer vision, bioinformatics, and etc.



# Difference between CPU and GPU Programming

CPU Program

Add vector A and vector B to vector C.

```
void add_vector (float *A,
                 float *B,
                 float *C,
                 int N)
{
  for ( int i = 0; i < N; i++ )
    C[i] = A[i] + B[i];
}

void main ( ) { …
    add_vector (A, B, C, N);
…}
```

GPU Program

Add vector A and vector B to vector C.

```
_global_ void  add_vector (float *A,
                           float *B,
                           float *C,
                           int N)
{
    if ( tid < N )
      C[tid] = A[tid] + B[tid];
}

void main ( ) { …
    add_vector <<<dimGrid, dimBlock>>> (A, B, C, N);
…}
```

## Programming in CUDA

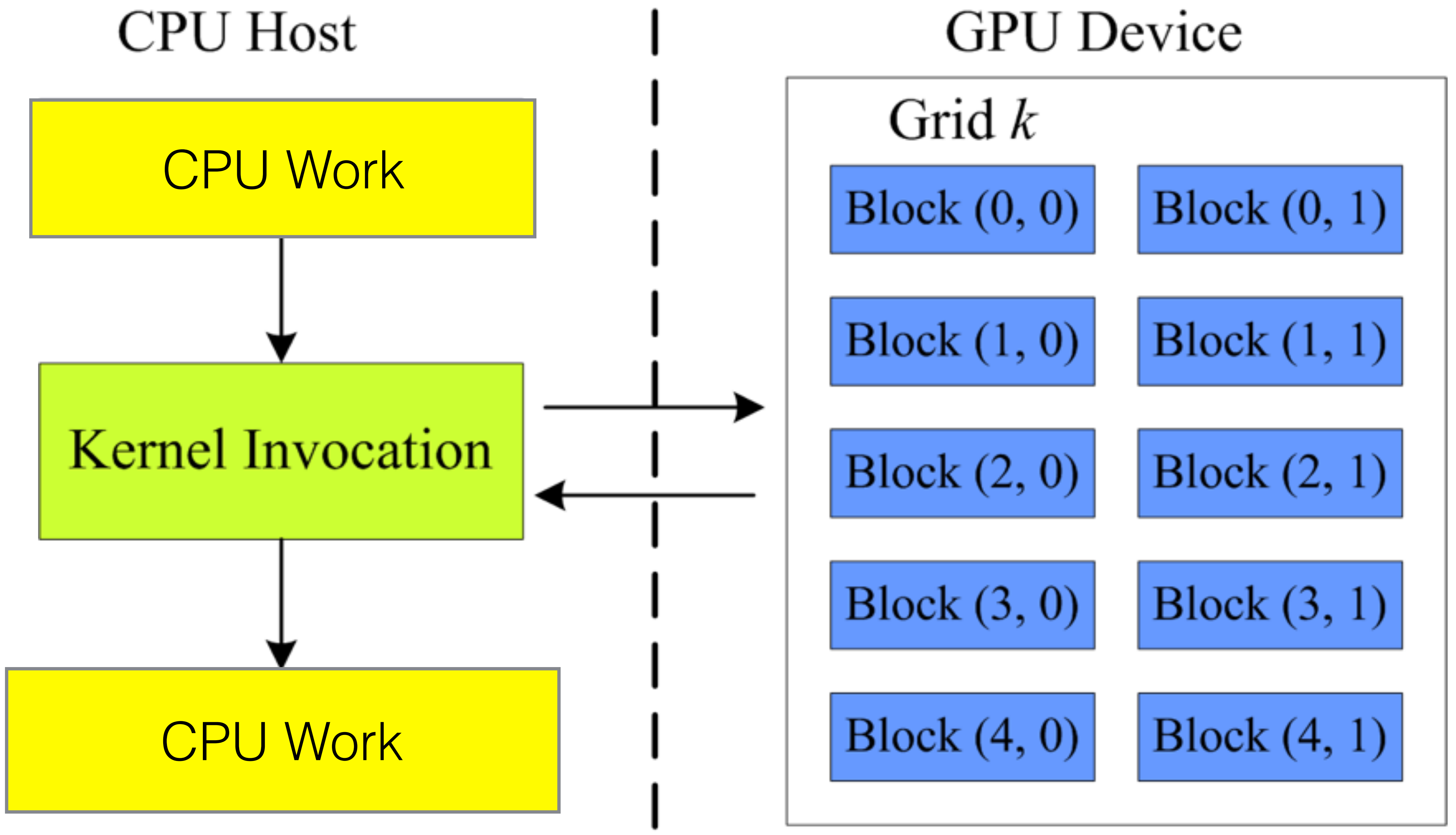
**Device Code (GPU):**

```
_global_ void  add_vector (float *A, float *B, float *C, int N){
                tid = blockDim.x * blockIdx.x + threadIdx.x;
              if (tid < N)
                C[tid] = A[tid] + B[tid];
}
```

**Host Code (CPU):**

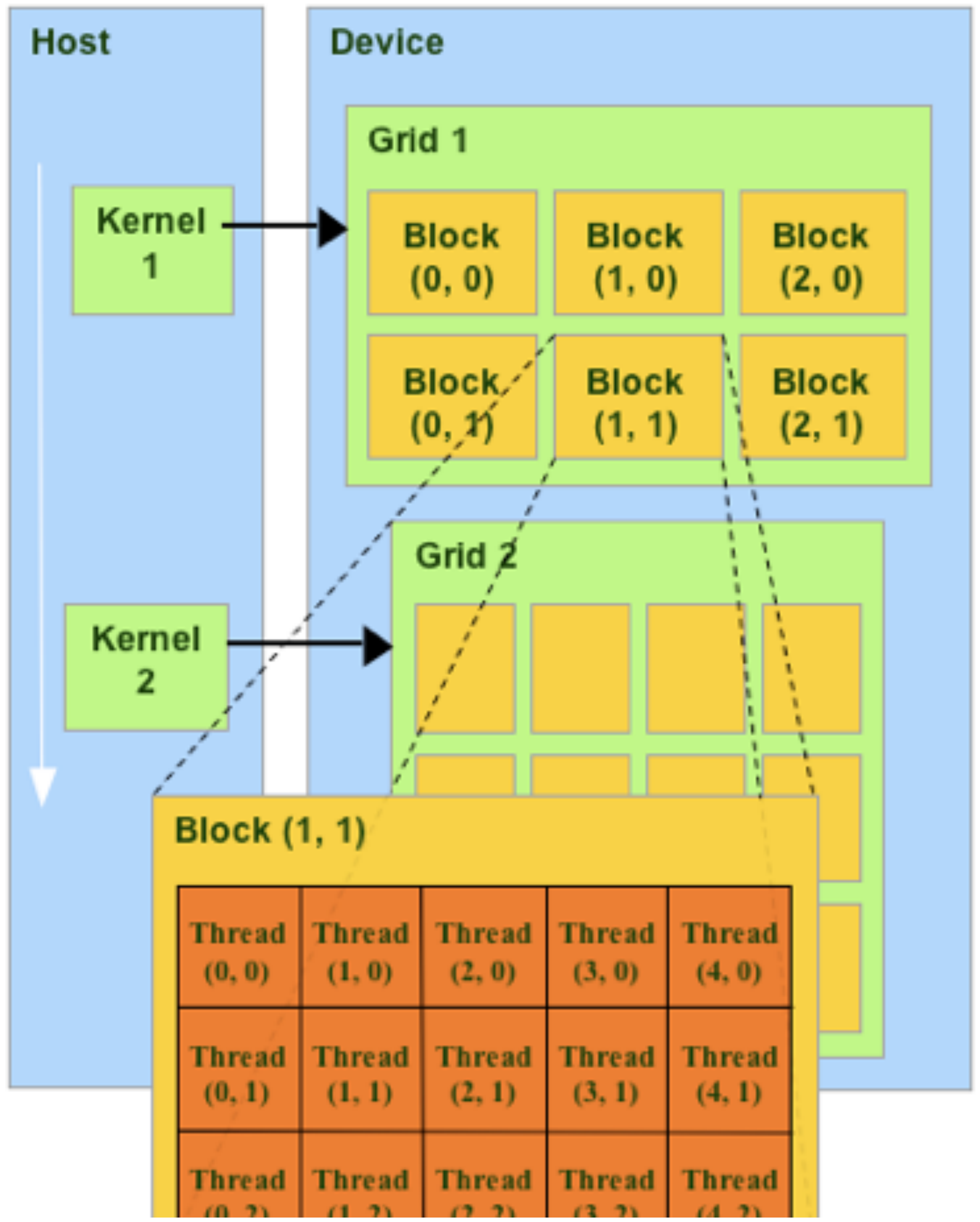
```
void main ( ) {
      … //allocate memory on GPU and initialization
     add_vector <<<dimGrid, dimBlock>>> (A, B, C, N);
      … // free memory on GPU
…}
```

## Programming in CUDA



## Thread View

- **A kernel is executed as a grid of thread blocks**
- **A thread block is a batch of threads that can cooperate with each other by:**
  - Synchronizing their execution
  - Sharing data through a low latency scratch-pad memory (named “shared memory”)
- **Two threads from two different blocks cannot communicate through the scratch-pad memory**



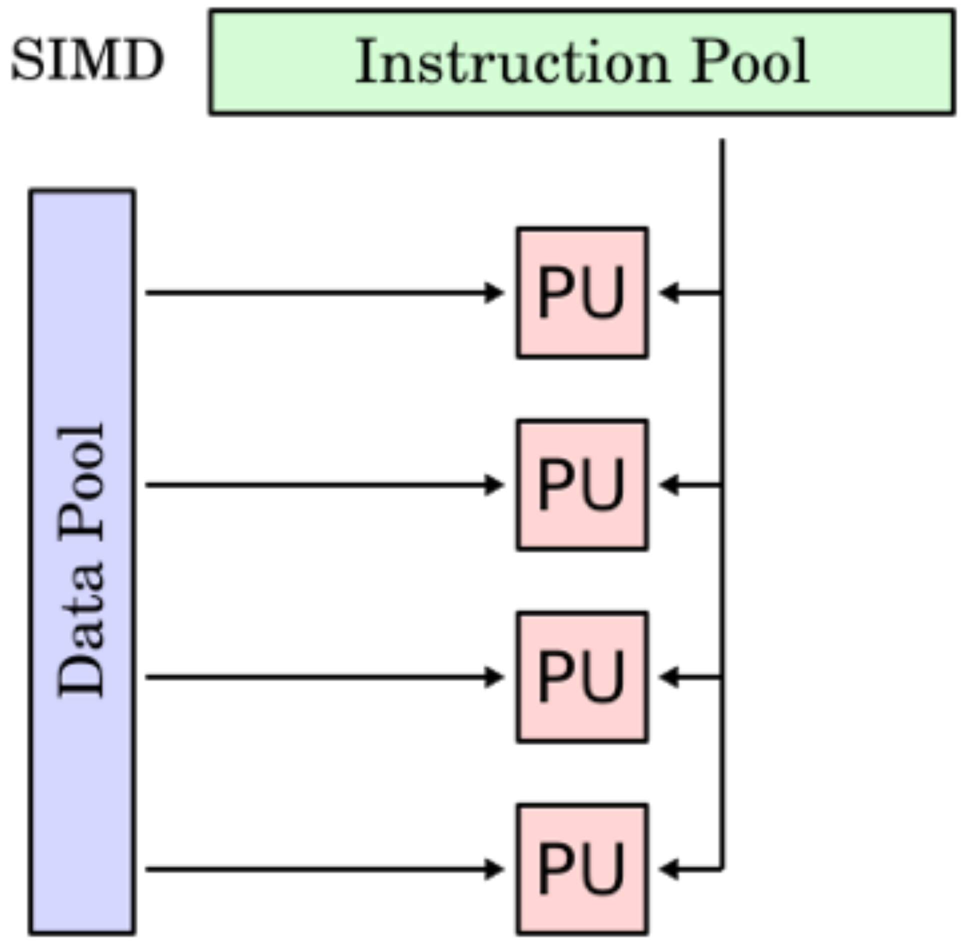
## Single Instruction Multiple Data (SIMD)

- **Flynn's taxonomy for modern processors**
  - Single Instruction Single Data (SISD)
  - Single Instruction Multiple Data (SIMD)
  - Multiple Instruction Single Data (MISD)
  - Multiple Instruction Multiple Data (MIMD)

*C[tid] = A[tid] + B[tid];*

*thread 0: C[0] = A[0] + B[0];*
*thread 1: C[1] = A[1] + B[1];*
....
*thread 15: C[15] = A[15] + B[15];*

Each PU is a processing unit



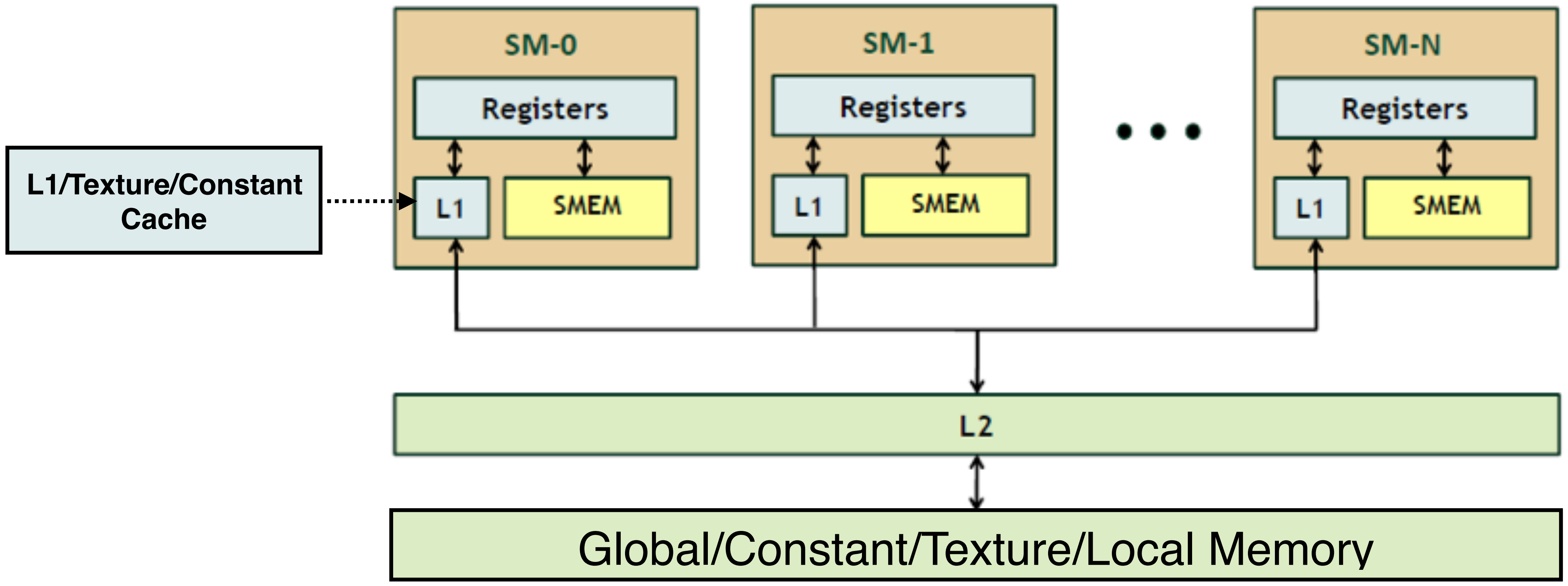
## Single Instruction Multiple Thread (SIMT)

- CUDA adopts Single Instruction Multiple Thread (SIMT) model
  - The machine model is organized into multiple SIMD units (warps)
    - Within each SIMD unit, the threads execute the same instruction
    - Across SIMD units, the threads execute different instructions
  - The software model hides the hardware complexity
    - SIMT allows specification of independent branching behavior, while SIMD does not
    - SIMD exposes the width to software writers, while SIMT does not

SIMT enables programmers to write thread-level parallel code for independent and scalar threads, even though the underlying hardware requires co-ordination between threads.

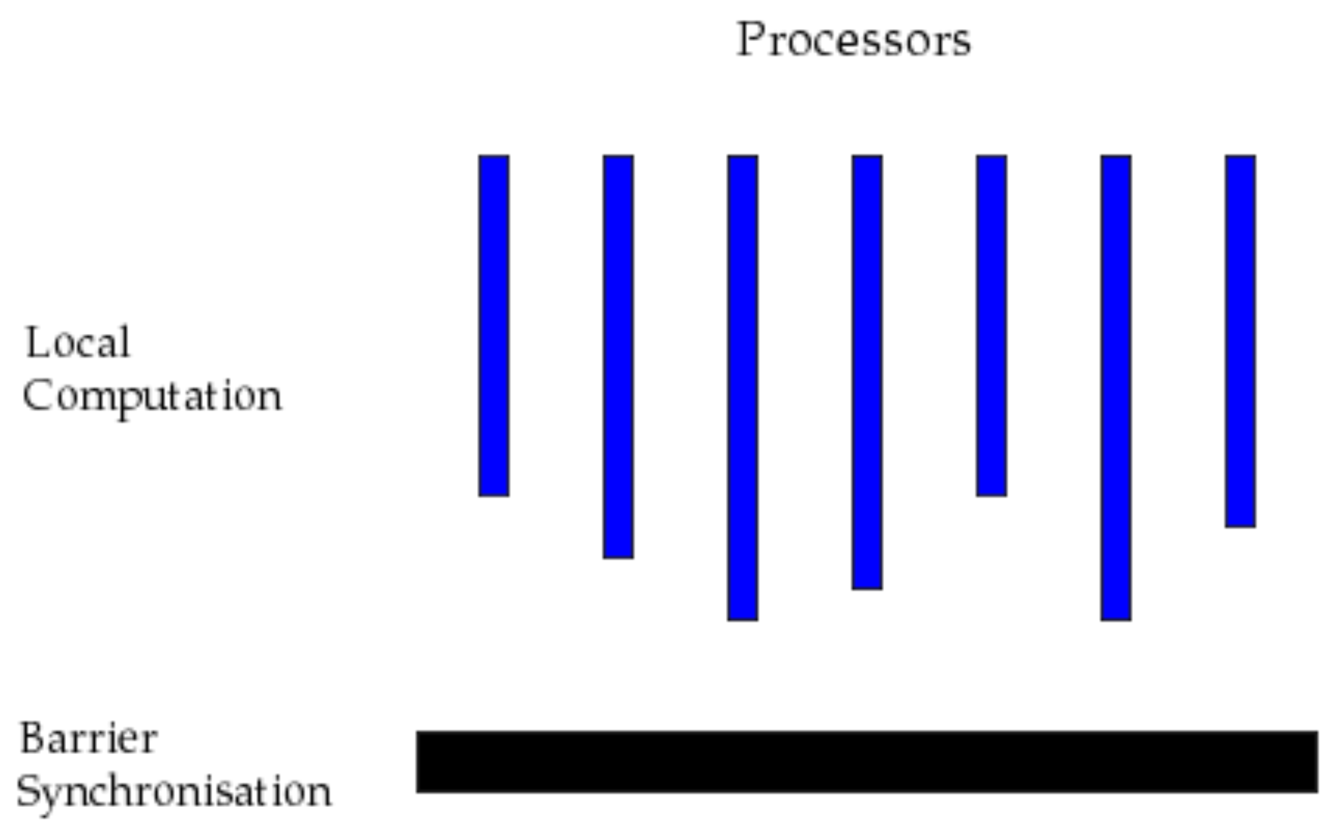
## CUDA Memory Model

- On Chip Memory
  - Registers and Shared Memory (SMEM)
  - Instruction Cache, Constant Cache, Texture/L1 Cache
    - Small latency, i.e., ~ 20 cycles or less
- Off Chip Memory
  - L2 (last level cache), ~ 400-1000 cycles latency
  - Global/Constant/Texture/Local Memory


L1/Texture/Constant Cache
SM-0
Registers
L1
SMEM
SM-1
Registers
L1
SMEM
SM-N
Registers
L1
SMEM
L2
Global/Constant/Texture/Local Memory

## CUDA Synchronization

- **Barrier Synchronization**
  - Within a thread block: __syncthreads( )
  - All computation before the barrier must complete before any computation after the barrier begins
  - Barriers divide computation into phases



## CUDA Synchronization

- **Compare-and-Swap (CAS)**
  - atomicCAS is provided
  - atomicAdd, atomicSub, atomicInc, and etc
  - built-in atomic functions can be used to implement other synchronization primitives such as locks, mutex, and monitors.

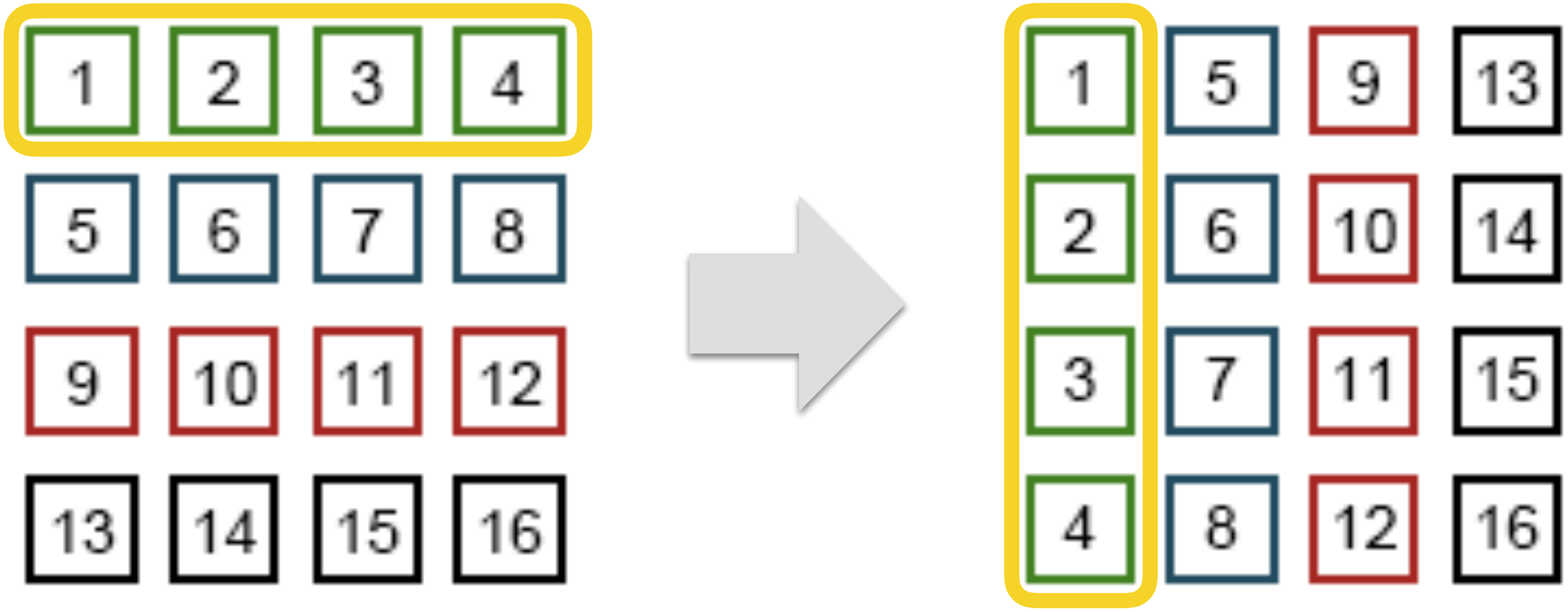
## Simple Implementation of Matrix Transpose

- **Each thread is in charge of one cell in the matrix**

```
__global__ void transpose_naive(float * outputData,
                                float * inputData,
                                int width,
                                int height)
{
  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

  if (xIndex < width && yIndex < height) {
     unsigned int index_in  = xIndex + width * yIndex;
     unsigned int index_out = yIndex + height * xIndex;
     outputData[ index_out ] = inputData[ index_in ];
  }
}
```

## Case Study I

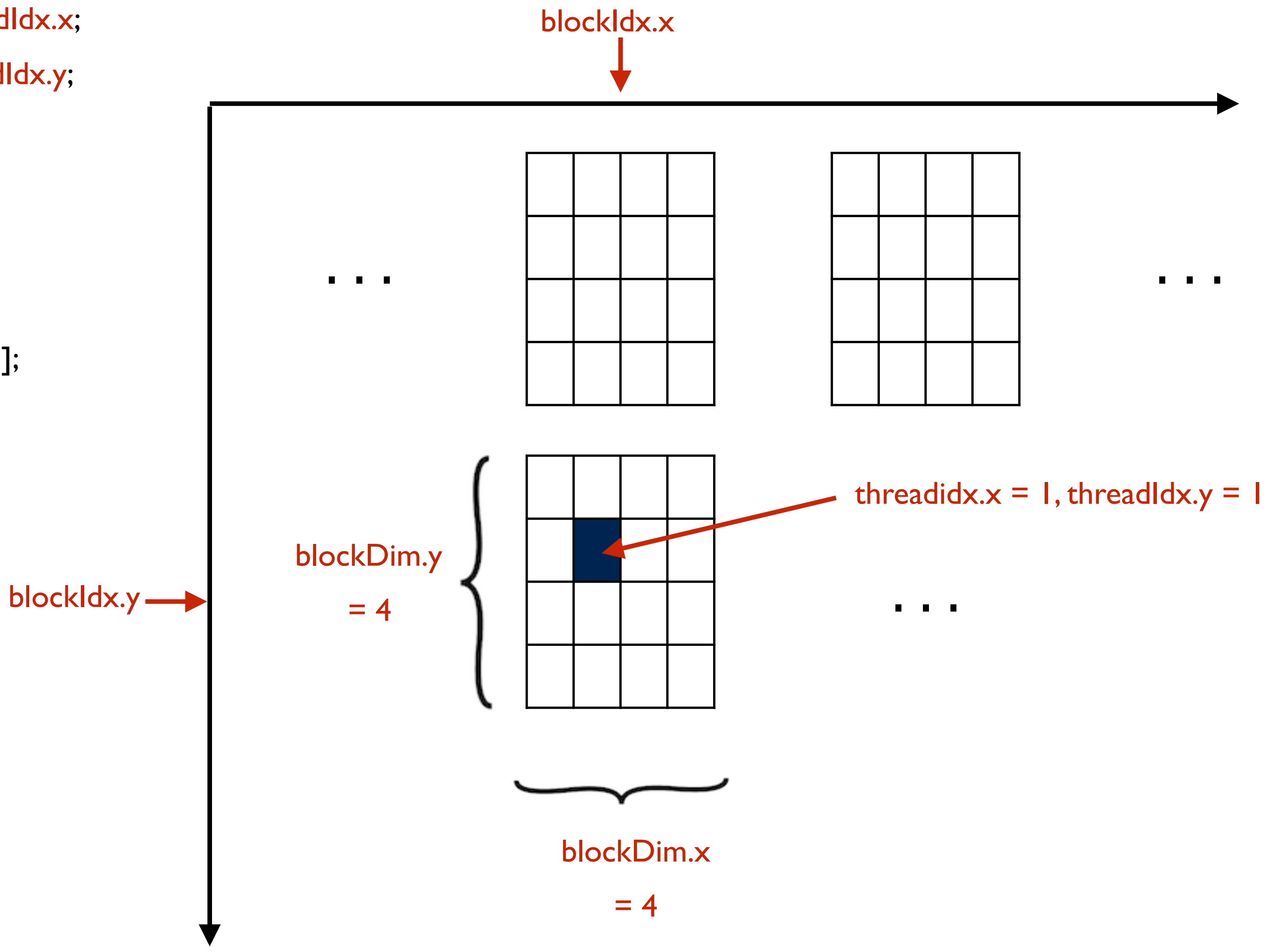
- **Matrix Transpose**



## Simple Implementation of Matrix Transpose

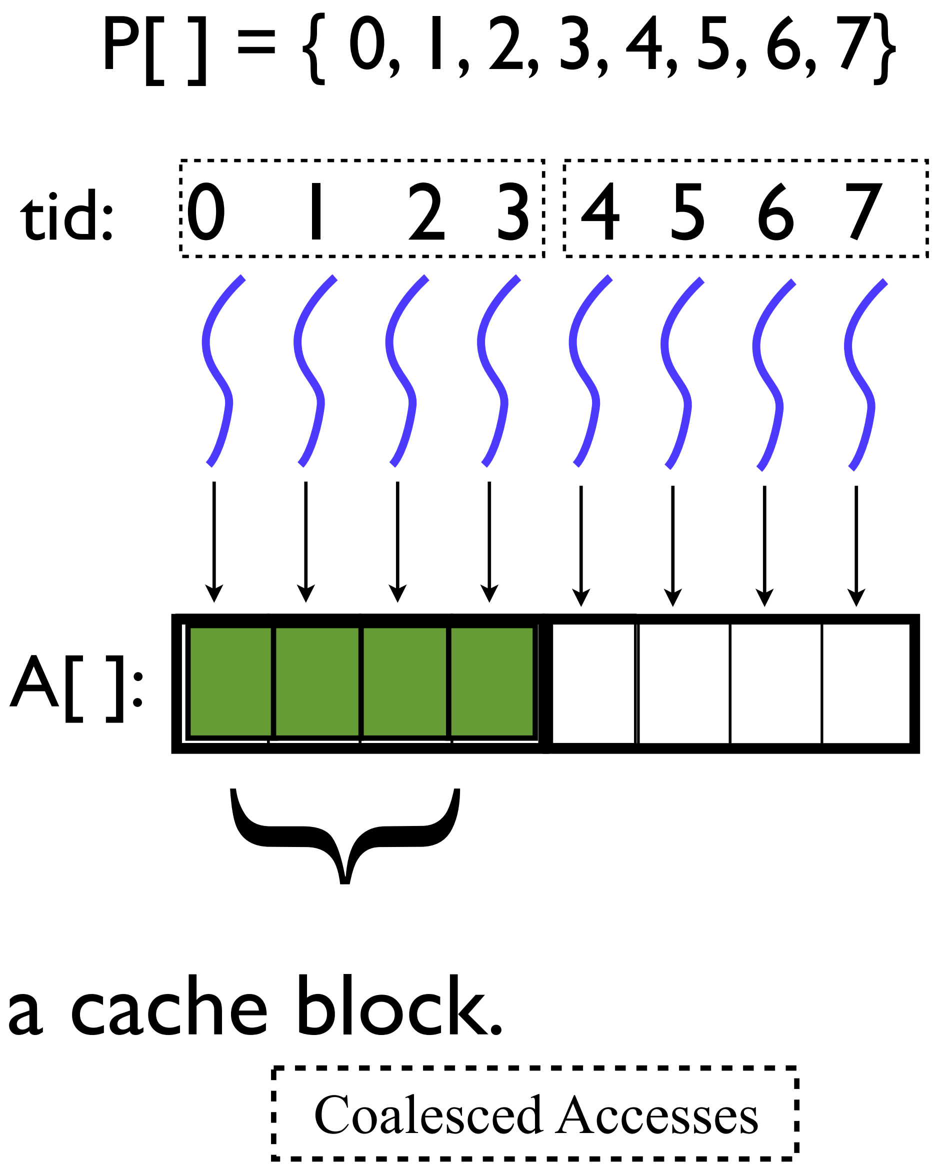
- **Each thread is in charge of one cell in the matrix**

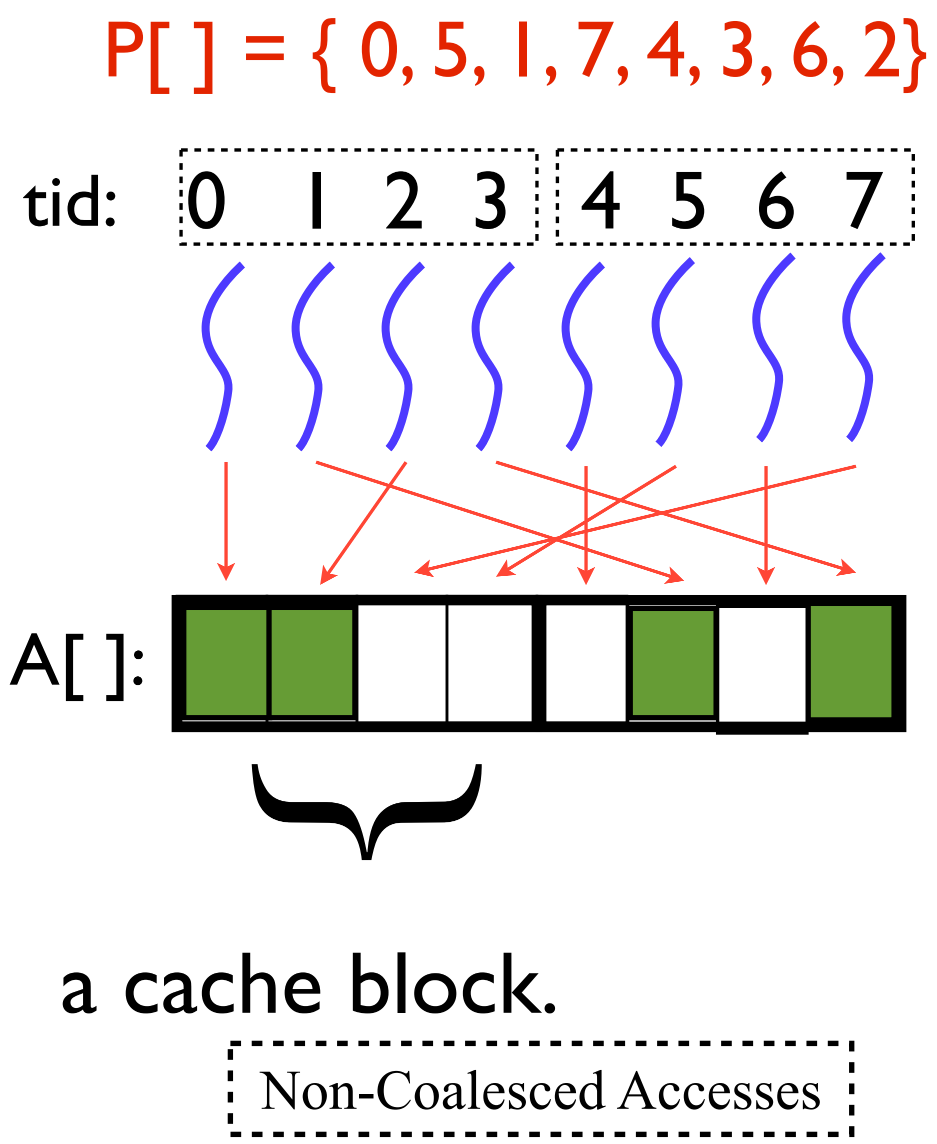
```
__global__ void transpose_naive(float * outputData,
                                float * inputData,
                                int width,
                                int height)
{
  unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
  unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

  if (xIndex < width && yIndex < height) {
     unsigned int index_in  = xIndex + width * yIndex;
     unsigned int index_out = yIndex + height * xIndex;
     outputData[ index_out ] = inputData[ index_in ];
  }
}
```



## Global Memory Coalescing

- **Data is fetched as a contiguous memory chunk**
  - A cache line or a cache block, typically 128 byte for GPUs
  - The purpose is to utilize wide DRAM burst for maximum memory level parallelism (MLP)s
- **A thread warp is ready when all its threads' operands are ready**





## Simple Implementation of Matrix Transpose

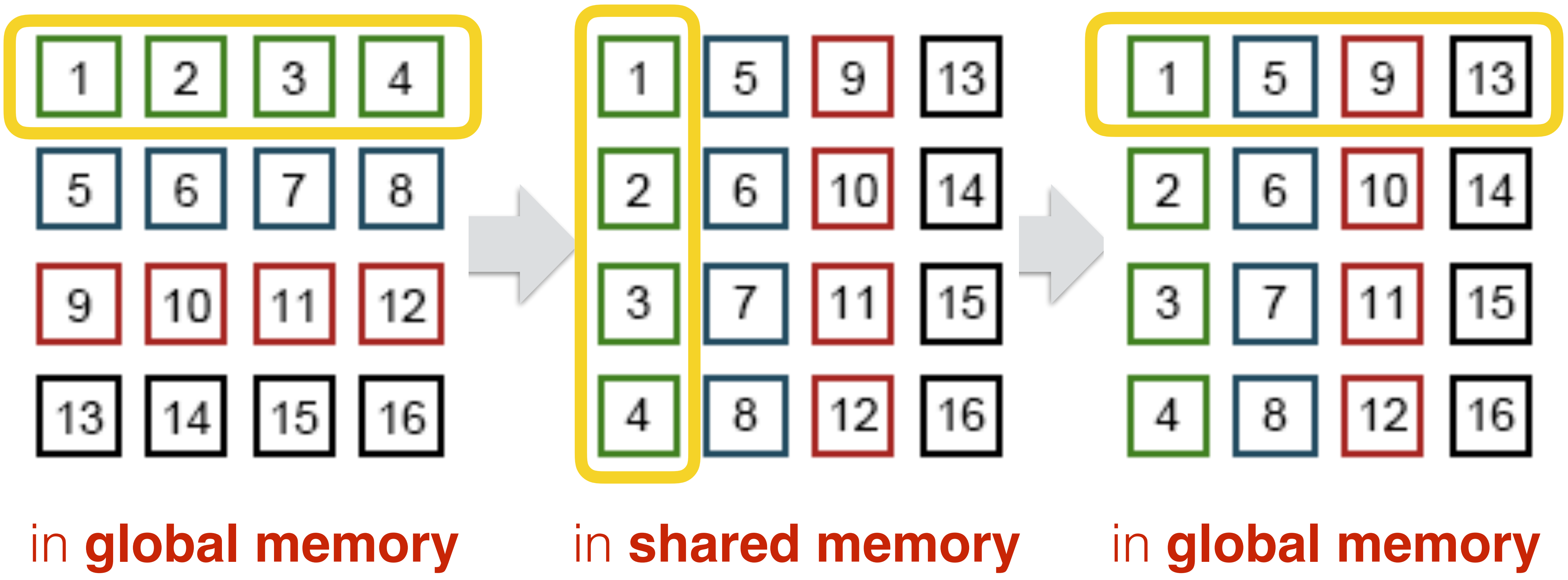
- **Each thread is in charge of one cell in the matrix**

```
__global__ void transpose_naive(float * outputData,
                                float * inputData,
                                int width,
                                int height)
{
   unsigned int xIndex = blockDim.x * blockIdx.x + threadIdx.x;
   unsigned int yIndex = blockDim.y * blockIdx.y + threadIdx.y;

   if (xIndex < width && yIndex < height) {
      unsigned int index_in  = xIndex + width * yIndex;
      unsigned int index_out = yIndex + height * xIndex;
      outputData[ index_out ] = inputData[ index_in ];
   }
}
```

coalesced or not?

## Matrix Transpose

- **Let's try something else!**



## Matrix Transpose

- **Improved Implementation**

```
__global__ void transpose(float *outputData, float *inputData, int width, int height){
        __shared__ float block[BLOCK_DIM][BLOCK_DIM + 1];
```

Input: 1 Million Matrix Elements

Naive Implementation
**Performance:** Throughput = 8.0226 GB/s, Time = 0.97381 ms

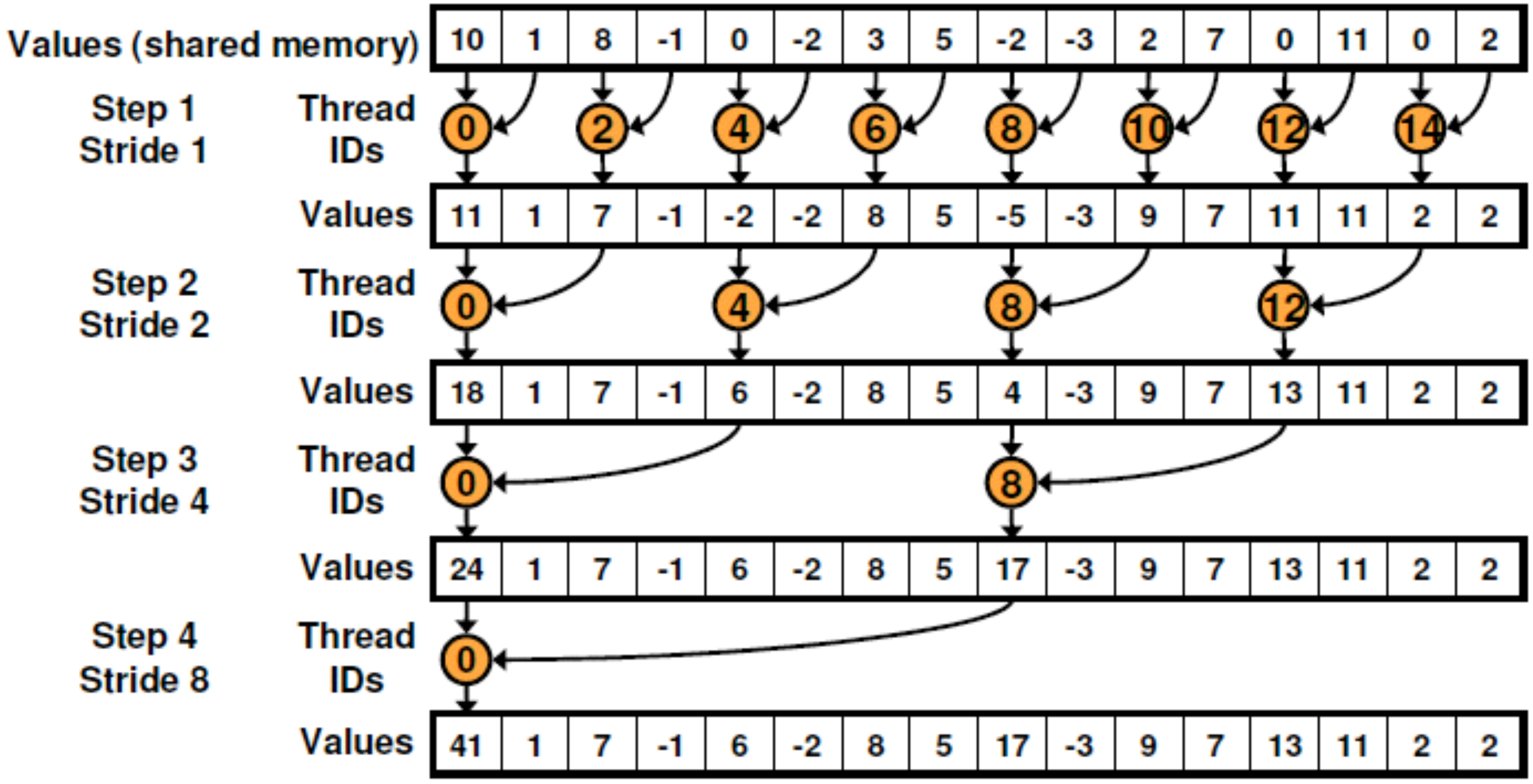
Improved implementation
**Performance:** Throughput = 11.5080 GB/s, Time = 0.67887 ms

**Code:** /ilab/users/zz124/cs515/samples/6_Advanced/transpose

```
        if( (xIndex < height) && (yIndex < width)) {
            unsigned int index_out = yIndex * height + xIndex;
            outputData[index_out] = block[threadIdx.x][threadIdx.y];
        }
}
```

## Case Study II

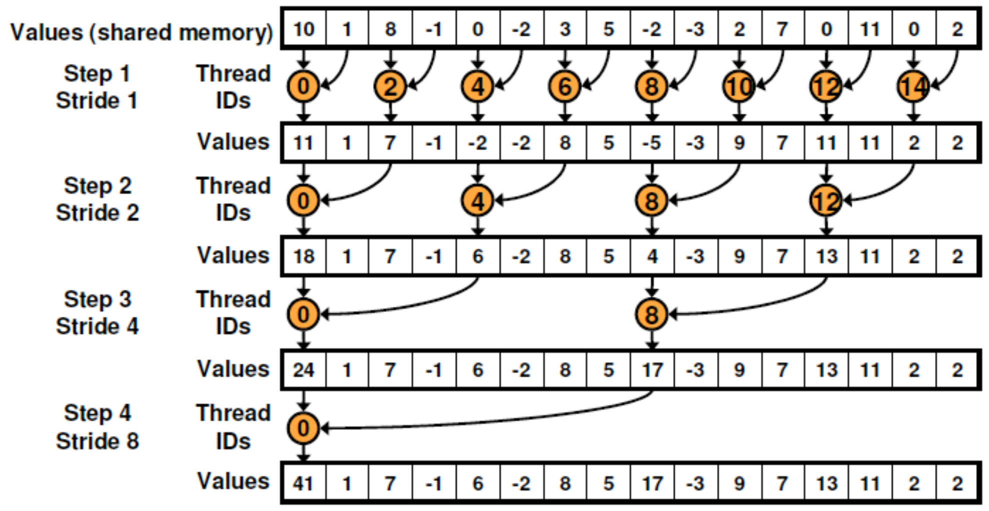
- **Parallel Reduction**



## Case Study II

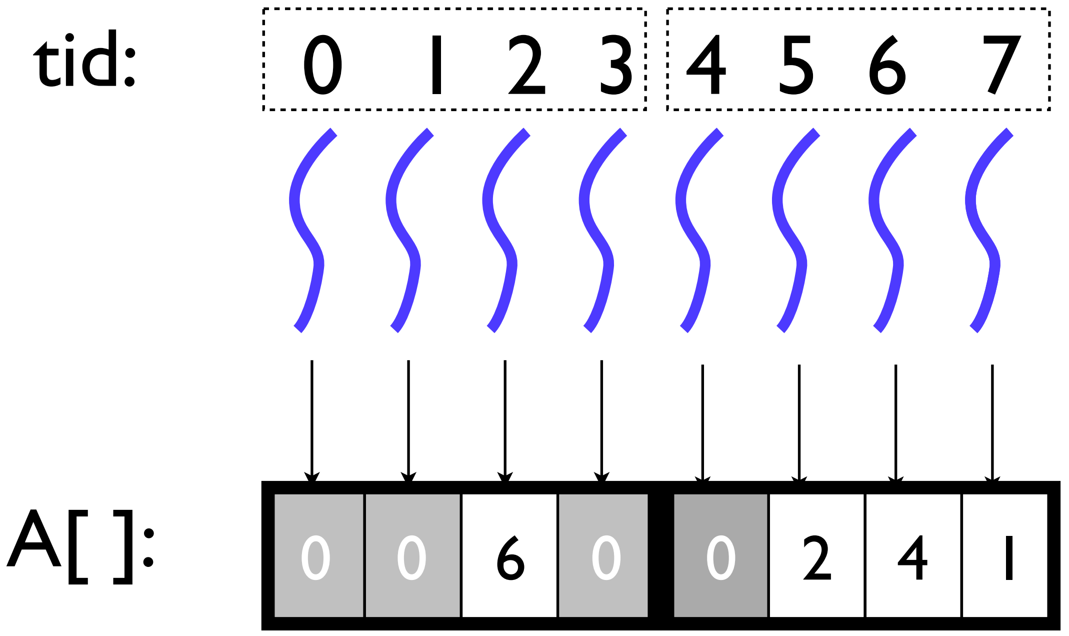
- **Parallel Reduction Code**

```
tid = threadIdx.x;

for ( s=1; s < blockDim.x; s *= 2) {
  if ( tid % (2*s) == 0 )  {
     sdata[ tid ] += sdata[ tid + s ];
  }
  __syncthreads();
}
```


Values (shared memory) 10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
Step 1 Stride 1
Thread IDs 0 2 4 6 8 10 12 14
Values 11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
Step 2 Stride 2
Thread IDs 0 4 8 12
Values 18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
Step 3 Stride 4
Thread IDs 0 8
Values 24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
Step 4 Stride 8
Thread IDs 0
Values 41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

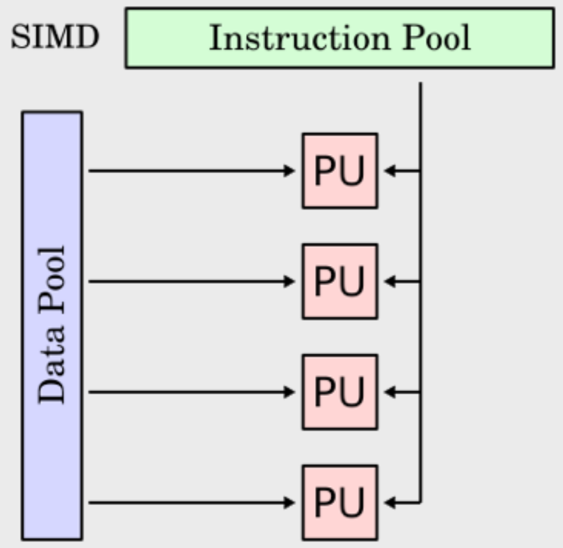
## Control Divergence

- Different runtime execution path lead to processor underutilization



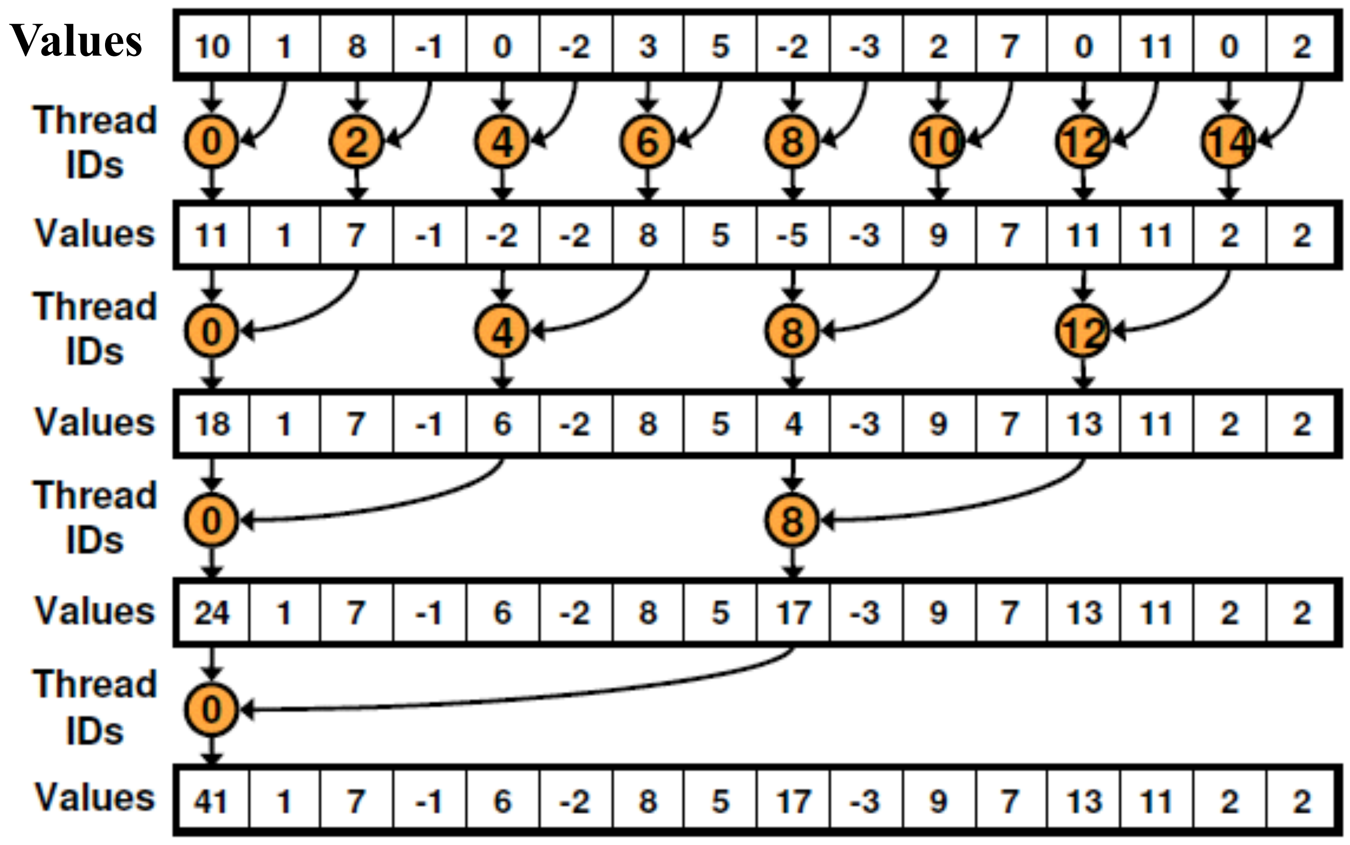
```
if (A[tid]) {
      do some work;
} else {
      do nothing;
}
```

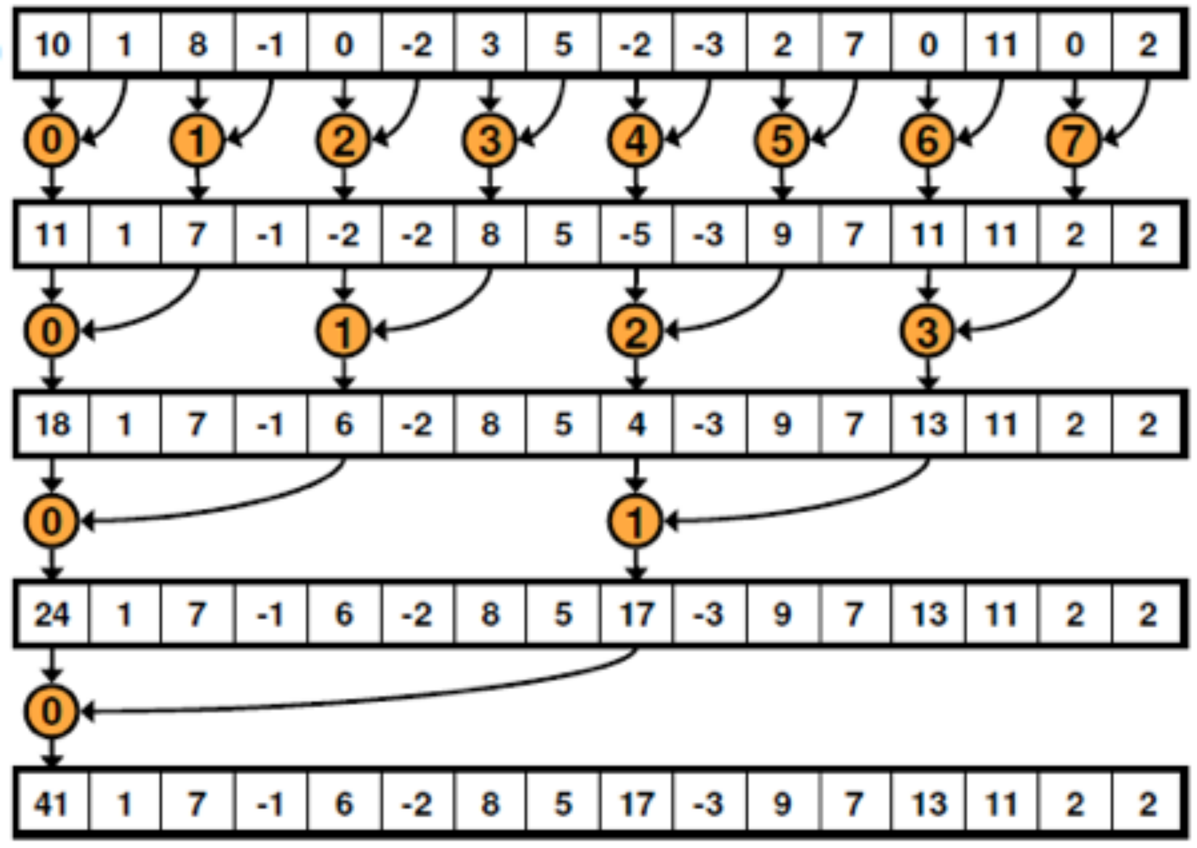
Recall that in SIMD model, only one instruction can be fetched and executed at one time. If any thread does not execute that instruction, the corresponding PU will be idle.



## Parallel Reduction

- **Optimize Control Divergence**





First Version → Second Version

## Parallel Reduction

- **Optimize Control Divergence**

tid

fo

i

}

}

Reduction on 16 Million Elements

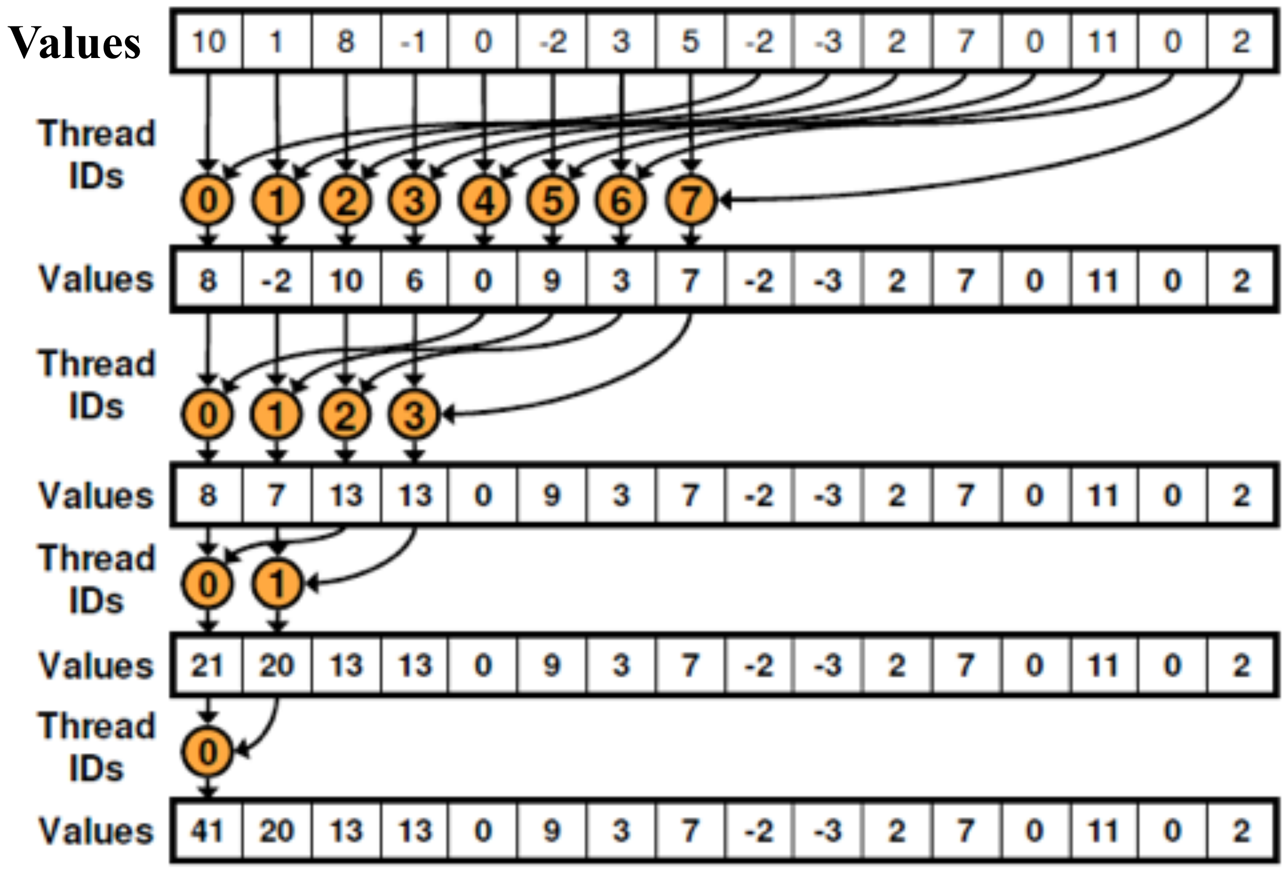
Implementation 1:
**Performance:** Throughput = 1.5325 GB/s, Time = 0.04379 s

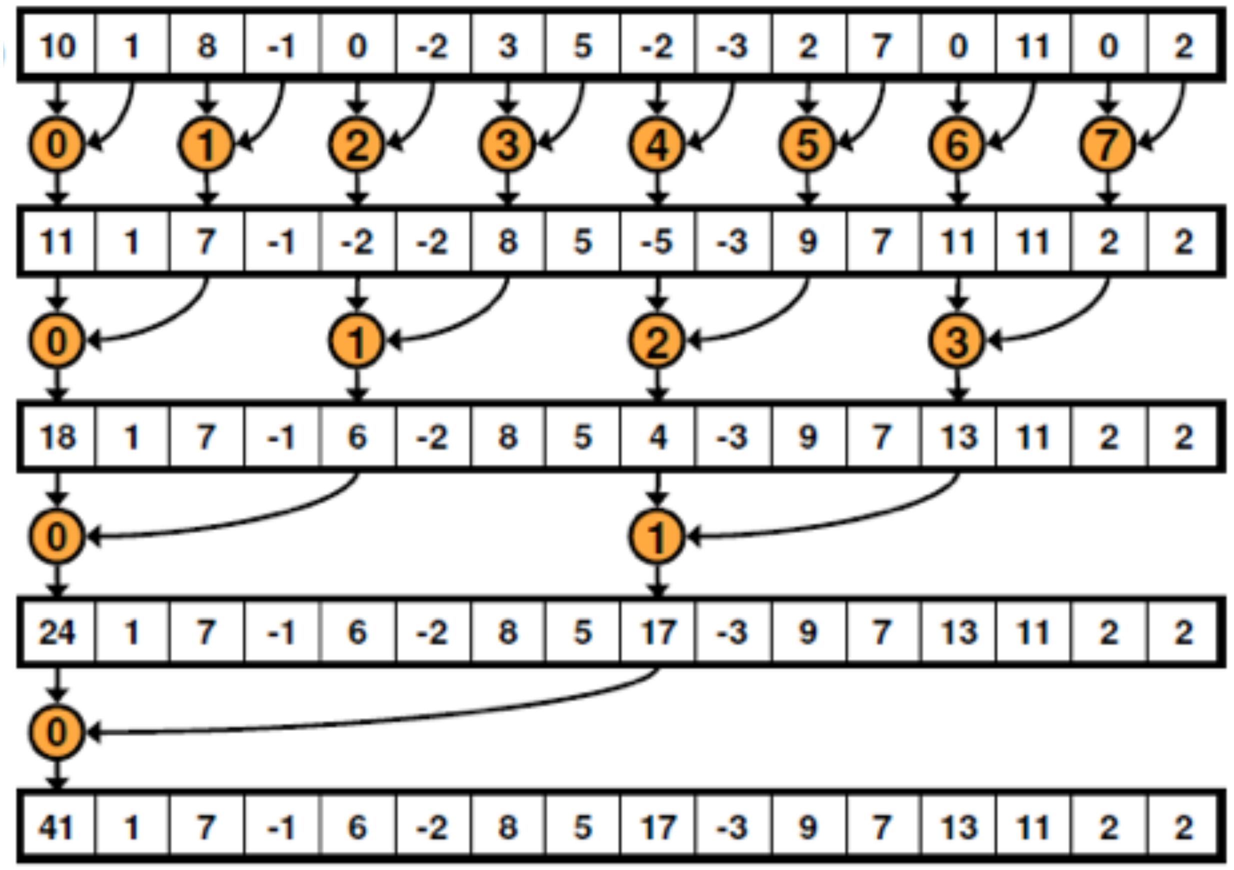
Implementation 2:
**Performance:** Throughput = 1.9286 GB/s, Time = 0.03480 s

First Version → Second Version

## Parallel Reduction

- **Improve Coalescing**

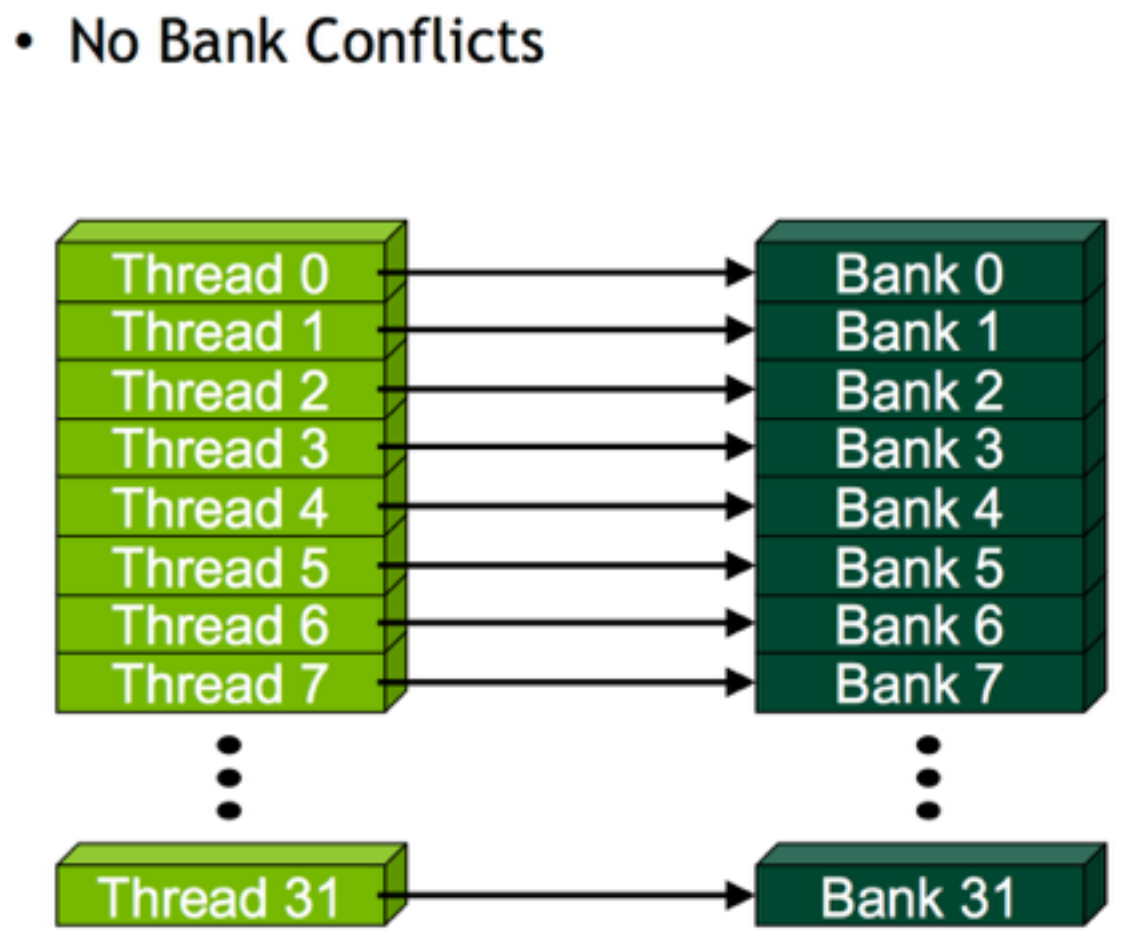


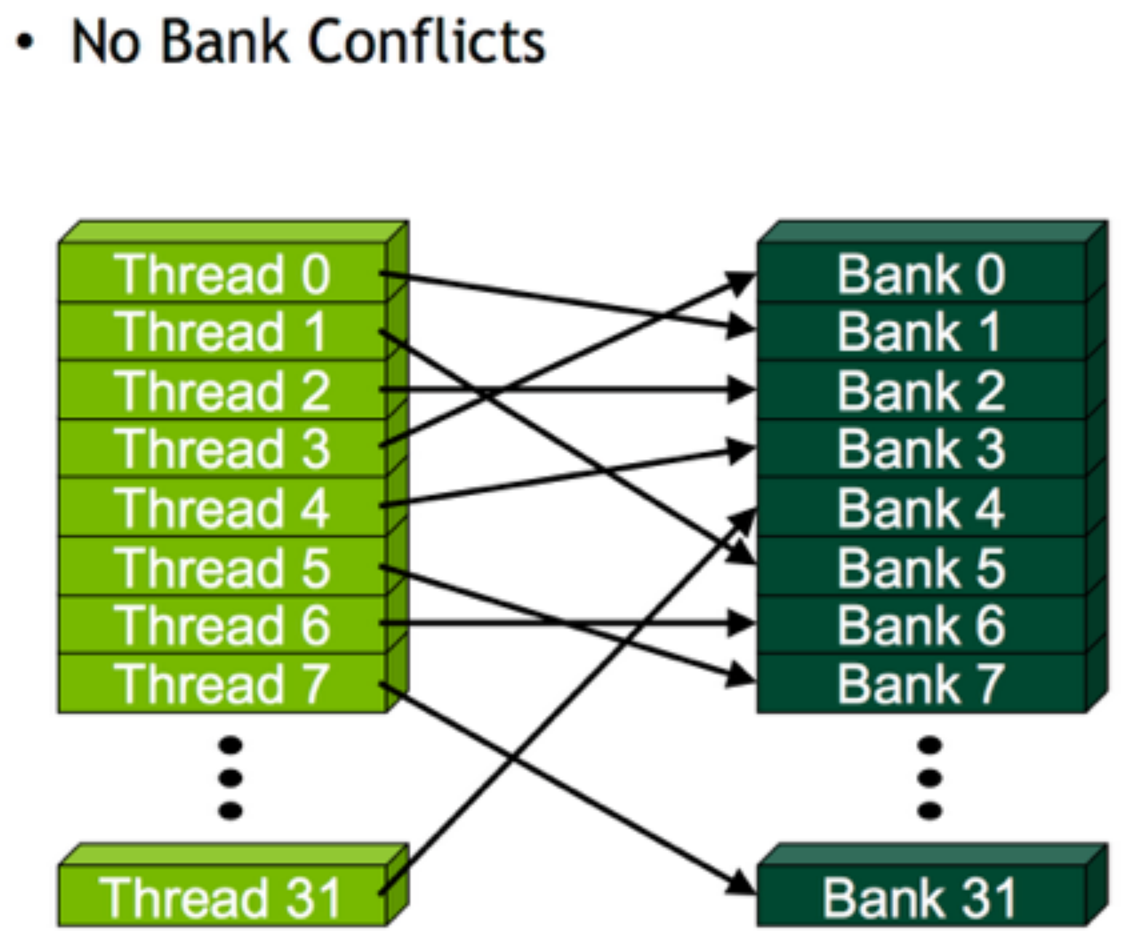


Second Version → Third Version

## Bank Conflicts

- **Shared Memory in CUDA Typically 16 or 32 banks**
  - Successive 32-bit words are assigned to successive banks
  - A fixed stride access may cause bank conflicts
  - Maximizing bank level parallelism is important

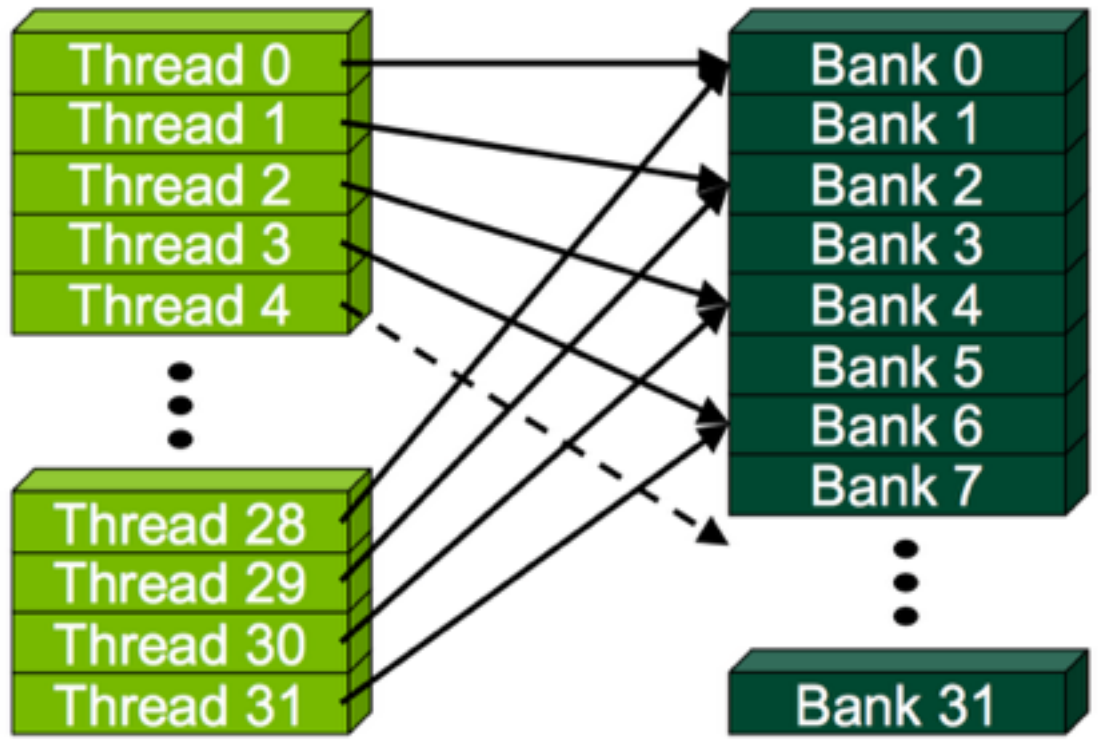




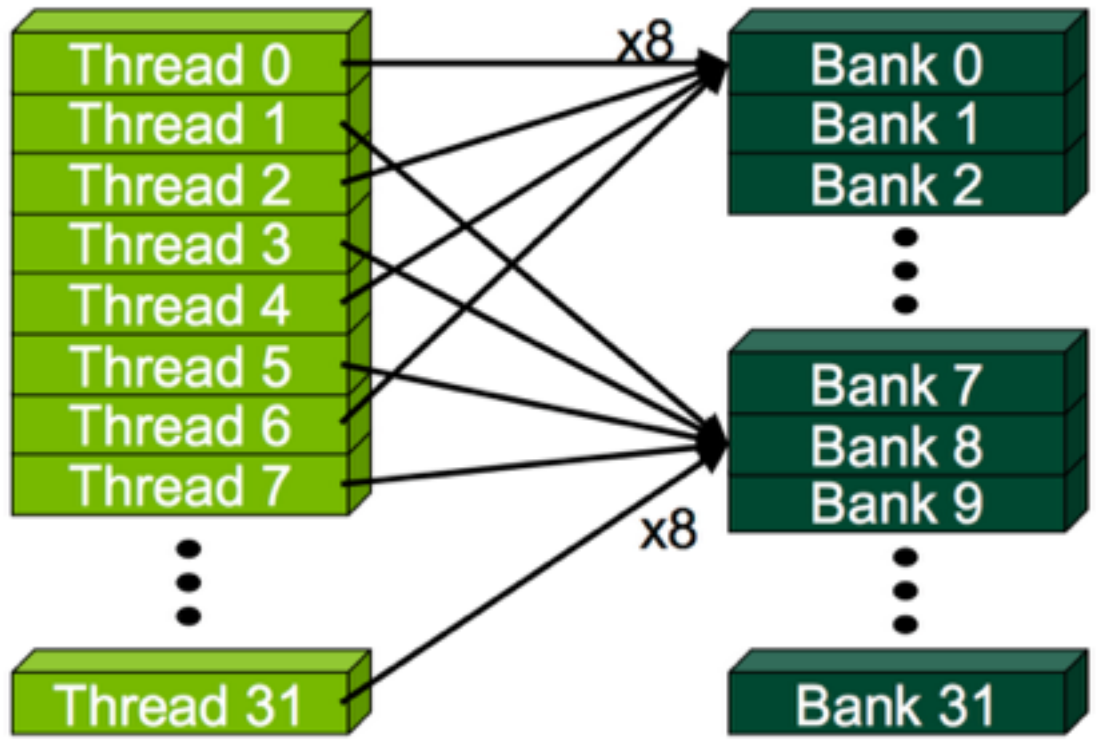
## Bank Conflicts

- **Shared Memory in CUDA Typically 16 or 32 banks**
  - Successive 32-bit words are assigned to successive banks
  - A fixed stride access may cause bank conflicts
  - Maximizing bank level parallelism is important

- 2-way Bank Conflicts

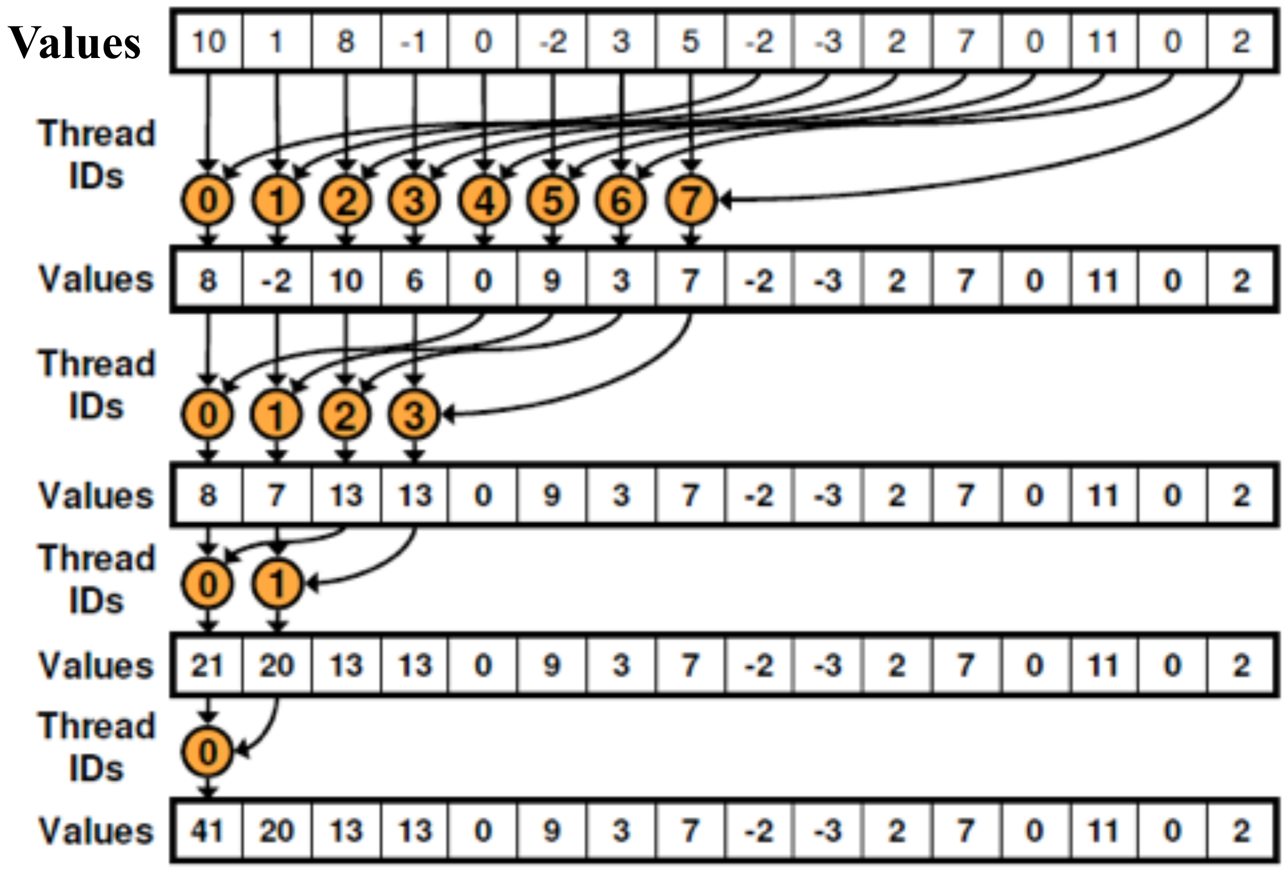


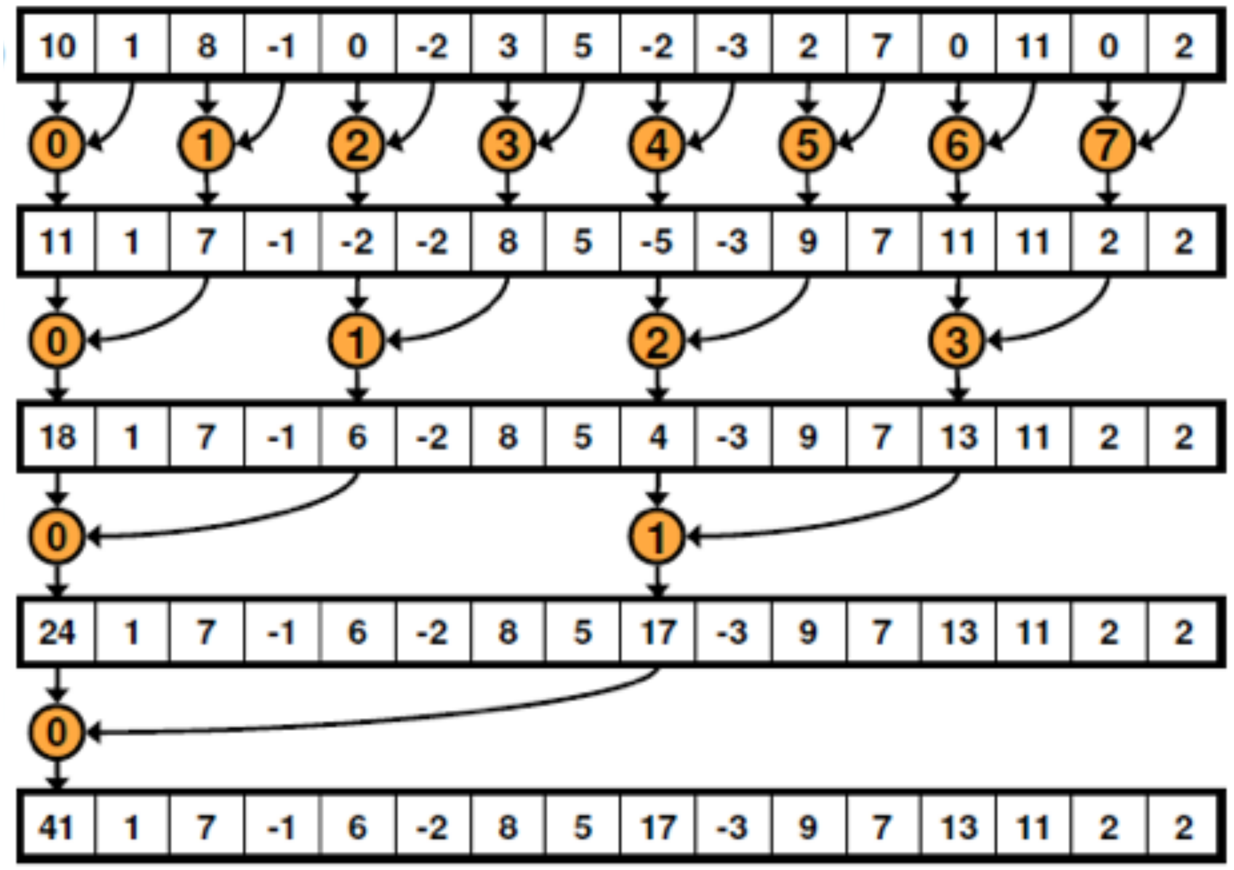
- 8-way Bank Conflicts



## Parallel Reduction

- **Improve Coalescing and Eliminates Bank Conflicts**





Second Version → Third Version

## GPU Resources at ilab

- **1 multi-GPU Machine: "atlas.cs.rutgers.edu"**
  - 1x Telsa K40:
    12 GB memory, 2880 CUDA cores, CUDA capability 3.5
  - 2x Quadro K4000:
    3 GB memory, 768 CUDA cores, CUDA capability 3.0
- **20 single-GPU Machines (listed on the right)**
  - Each has one GT 630
  - 2 GB memory, 192 CUDA cores, CUDA capability 3.0

Check http://report.cs.rutgers.edu/mrtg/systems/ilab.html for availability of different GPU machines

## Reading

- "Optimization Parallel Reduction in CUDA" by Mark Harris, NVIDIA http://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/reduction/doc/reduction.pdf
- "Parallel Prefix Sum (Scan) with CUDA", GPU Gem3, Chapter 39, by Mark Harris, Shubhabrata Sengupta, and John Owens. http://http.developer.nvidia.com/GPUGems3/gpugems3_ch39.html
- "Performance Optimization" by Paulius Micikevicius, NVIDIA http://www.nvidia.com/docs/IO/116711/sc11-perf-optimization.pdf
- CUDA Programming Guide https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html