# Introduction to GPU programming for deep learning applications

Instructor: Xin Liang (xliang@uky.edu)

Workshop on Deep Learning Systems in Advanced GPU Cyberinfrastructure

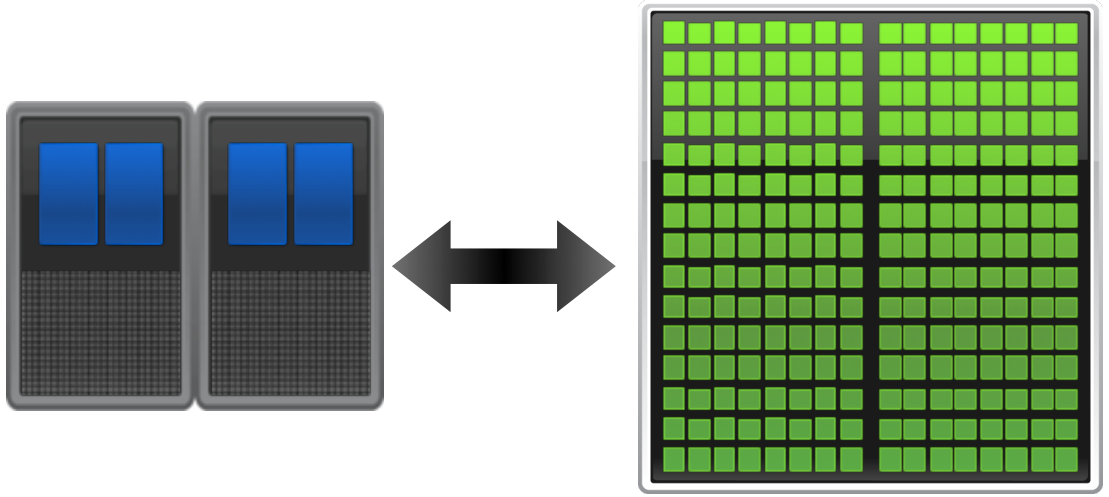
Some of the materials are from Internet. Copyrights may be reserved by NVIDIA, Paweł Luniak etc.

# Agenda

- Optimization of CUDA code
- Introduction to Tensor Core architecture and programming
- Implementing MLP using CUDA
- Advanced optimizations for DL applications with GPUs

# Optimization of CUDA code

# CPU vs GPU



**Latency Optimized CPU**

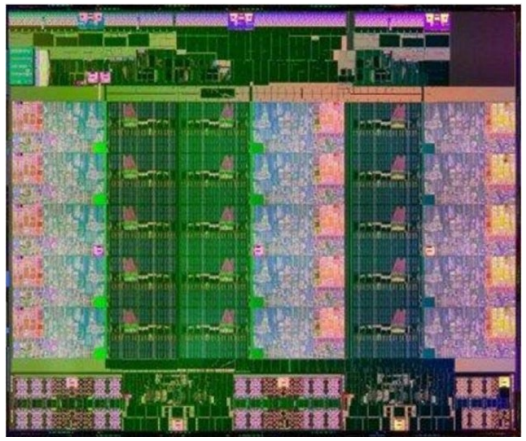
Fast Serial Processing

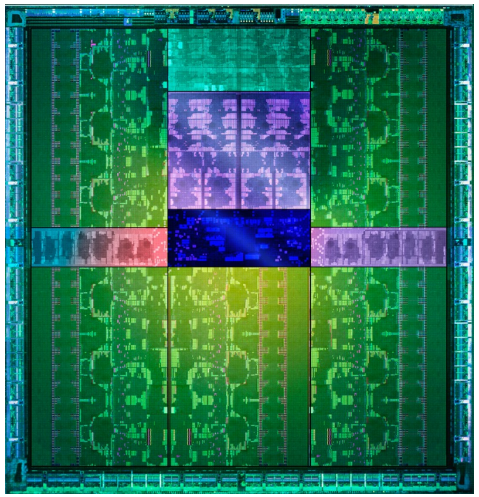
**Throughput Optimized GPU**

Scalable Parallel Processing

# CPU vs GPU

| Specifications | Ivy Bridge EX (Xeon E7-8890v2) | Kepler (Tesla K40) |
|---|---|---|
| Processing Elements | 15 cores, 2 issue, 8 way SIMD @**2.8** GHz | 15 SMs, 6 issue, 32 way SIMD @**745** MHz |
| Resident Strands/Threads (max) | 15 cores, 2 threads, 8 way SIMD: **240** strands | 15 SMs, 64 SIMD vectors, 32 way SIMD: **30720** threads |
| SP GFLOP/s | 672 | 4291 (6.3x) |
| Memory Bandwidth | 85 GB/s | 288 GB/s (3.4x) |
| Flops / Byte (Roofline) | 7.9 Flops/Byte | 15 Flops/Byte |
| Register File | xx kB (?) | 3.75 MB |
| Local Store/L1 Cache | 960 kB | 960 kB |
| L2 Cache | 3.75 MB | 1.5 MB |
| L3 Cache | 37.5 MB | - |





# CPU vs GPU

- Different goals produce different designs
  - Throughput cores: assume work load is highly parallel
  - Latency cores: assume workload is mostly sequential
- Latency goal: minimize latency experienced by 1 thread
  - lots of big on-chip caches
  - extremely sophisticated control, branch prediction
- Throughput goal: maximize throughput of all threads
  - lots of big ALUs
  - multithreading can hide latency … so skip the big caches
  - simpler control, cost amortized over ALUs via SIMD

# GPU programming paradigm

1. Setup inputs on the host (CPU-accessible memory)
2. Allocate memory for outputs on the host
3. Allocate memory for inputs on the GPU
4. Allocate memory for outputs on the GPU
5. Copy inputs from host to GPU
6. Start GPU kernel (function that executed on GPU)
7. Copy output from GPU to host

# An example

Add two arrays

- A[ ] + B[ ] -> C[ ]

On the CPU:

```
float *C = malloc(N * sizeof(float));
for (int i = 0; i < N; i++)
C[i] = A[i] + B[i];
return C;
```

How can parallel the code on GPUs?

# An example

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    //Decide an index somehow
    c[index] = a[index] + b[index];
}
```

# An example

```
__global__ void
cudaAddVectorsKernel(float * a, float * b, float * c) {
    unsigned int index = blockIdx.x * blockDim.x + threadIdx.x;
    c[index] = a[index] + b[index];
}
```

# CPU and GPU

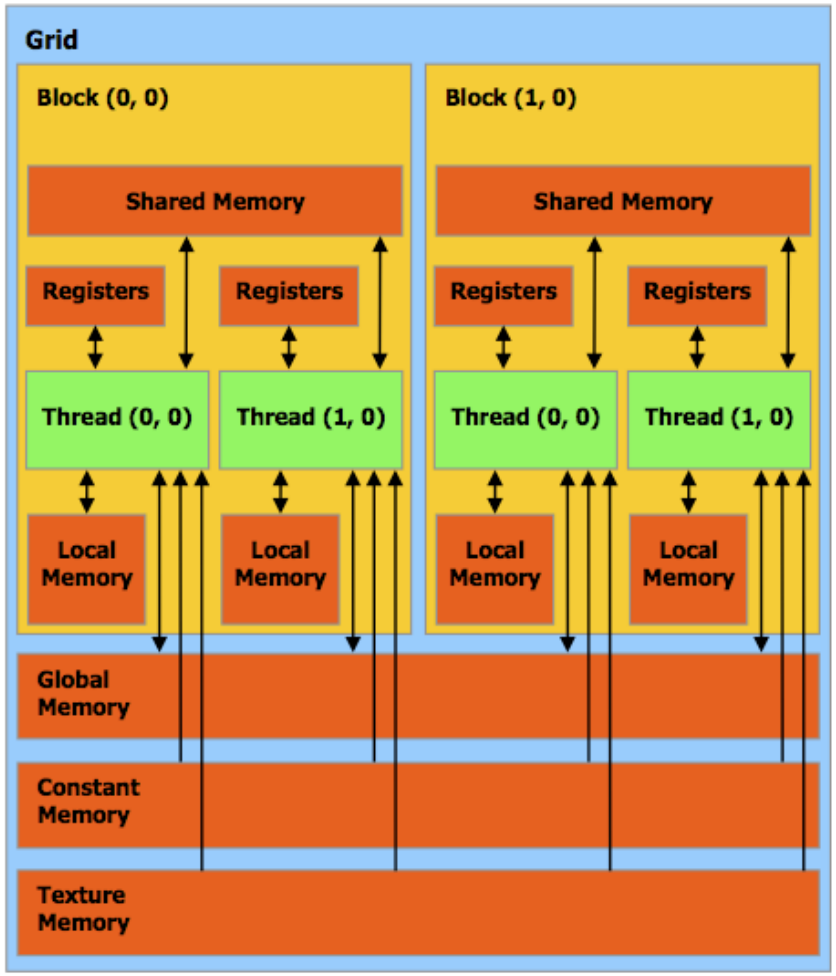
```
void cudaAddVectors(const float* a, const float* b, float* c, size){
    //For now, suppose a and b were created before calling this function

    // dev_a, dev_b (for inputs) and dev_c (for outputs) will be
    // arrays on the GPU.

    float * dev_a;
    float * dev_b;

    float * dev_c;

    // Allocate memory on the GPU for our inputs:
    cudaMalloc((void **) &dev_a, size*sizeof(float));
    cudaMemcpy(dev_a, a, size*sizeof(float), cudaMemcpyHostToDevice);

    cudaMalloc((void **) &dev_b, size*sizeof(float)); // and dev_b
    cudaMemcpy(dev_b, b, size*sizeof(float), cudaMemcpyHostToDevice);


    // Allocate memory on the GPu for our outputs:
    cudaMalloc((void **) &dev_c, size*sizeof(float));
```

# CPU and GPU

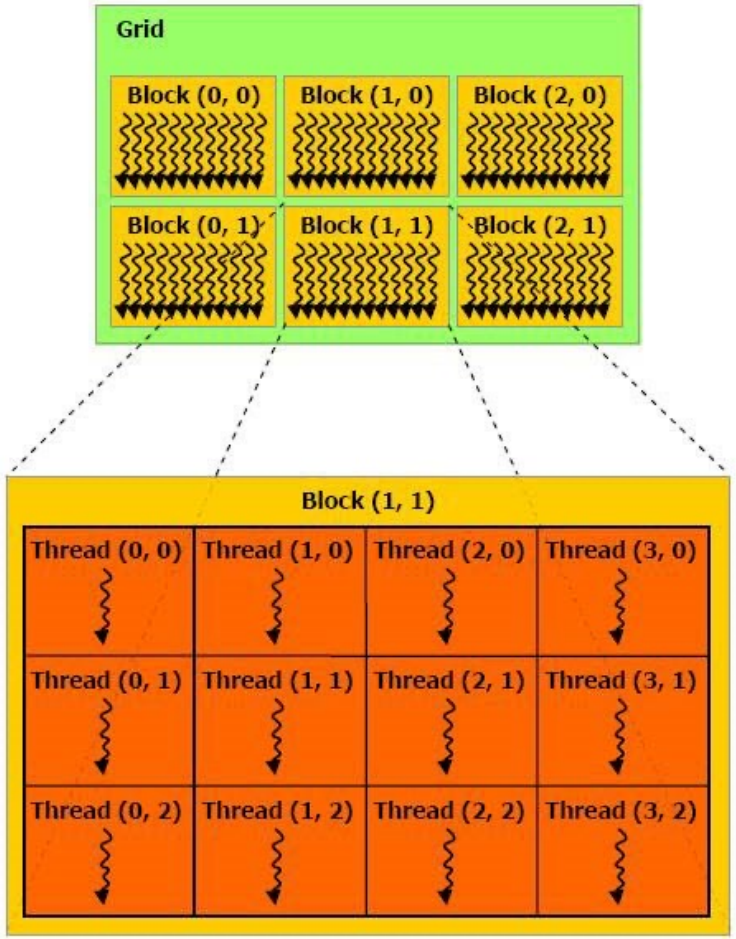
```
    //At lowest, should be 32
    //Limit of 512 (Tesla), 1024 (newer)
    const unsigned int threadsPerBlock = 512;

    //How many blocks we'll end up needing
    const unsigned int blocks = ceil(size/float(threadsPerBlock));

    //Call the kernel!
    cudaAddVectorsKernel<<<blocks, threadsPerBlock>>>
        (dev_a, dev_b, dev_c);

    //Copy output from device to host (assume here that host memory
    //for the output has been calculated)

    cudaMemcpy(c, dev_c, size*sizeof(float), cudaMemcpyDeviceToHost);

    //Free GPU memory
    cudaFree(dev_a);
    cudaFree(dev_b);
    cudaFree(dev_c);
}
```

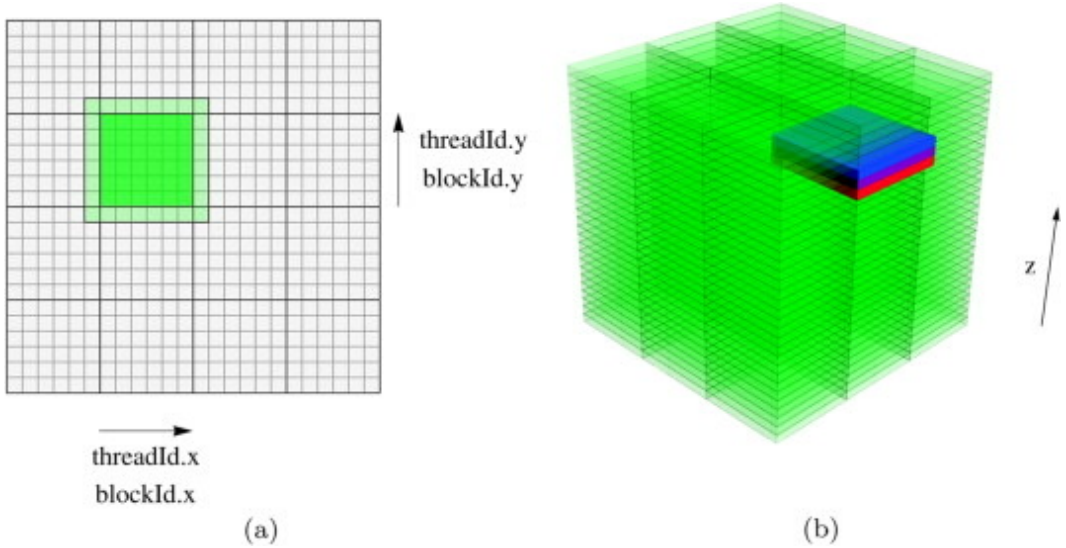
# Thread block organization

- Keywords you MUST know to code in CUDA:
  - **Thread** - Distributed by the CUDA runtime (threadIdx)
  - **Block/ThreadBlock** - A user defined group of 1 to ~512 threads (blockIdx)
  - **Grid** - A group of one or more blocks. A grid is created for each CUDA kernel function called.
- Imagine thread organization as an array of thread indices



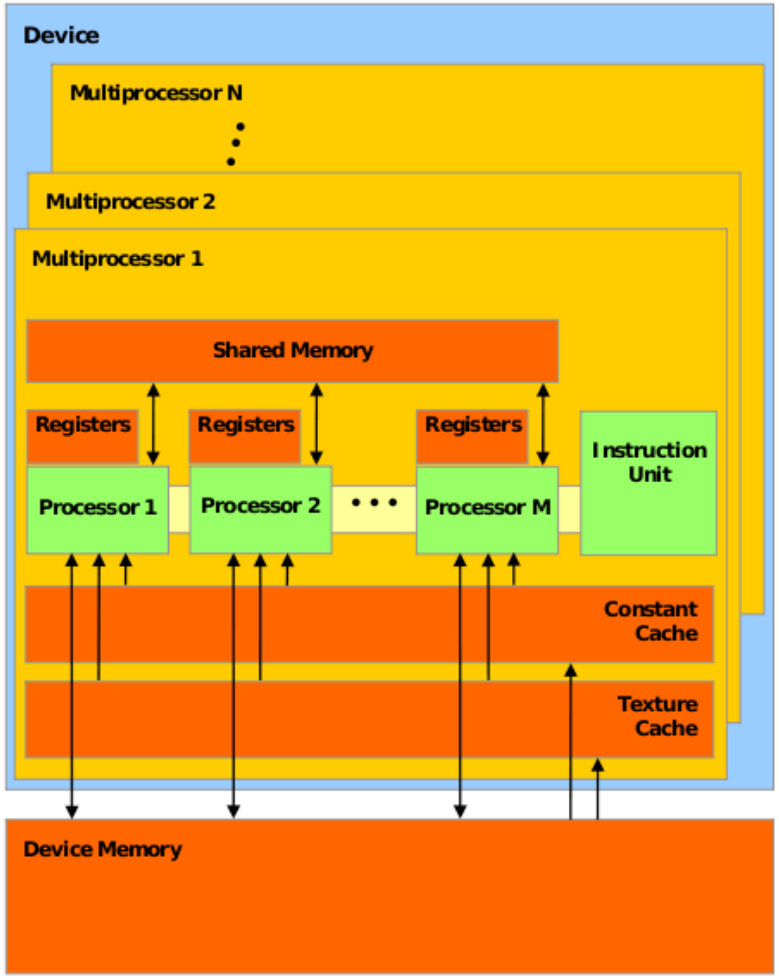
# Thread block organization



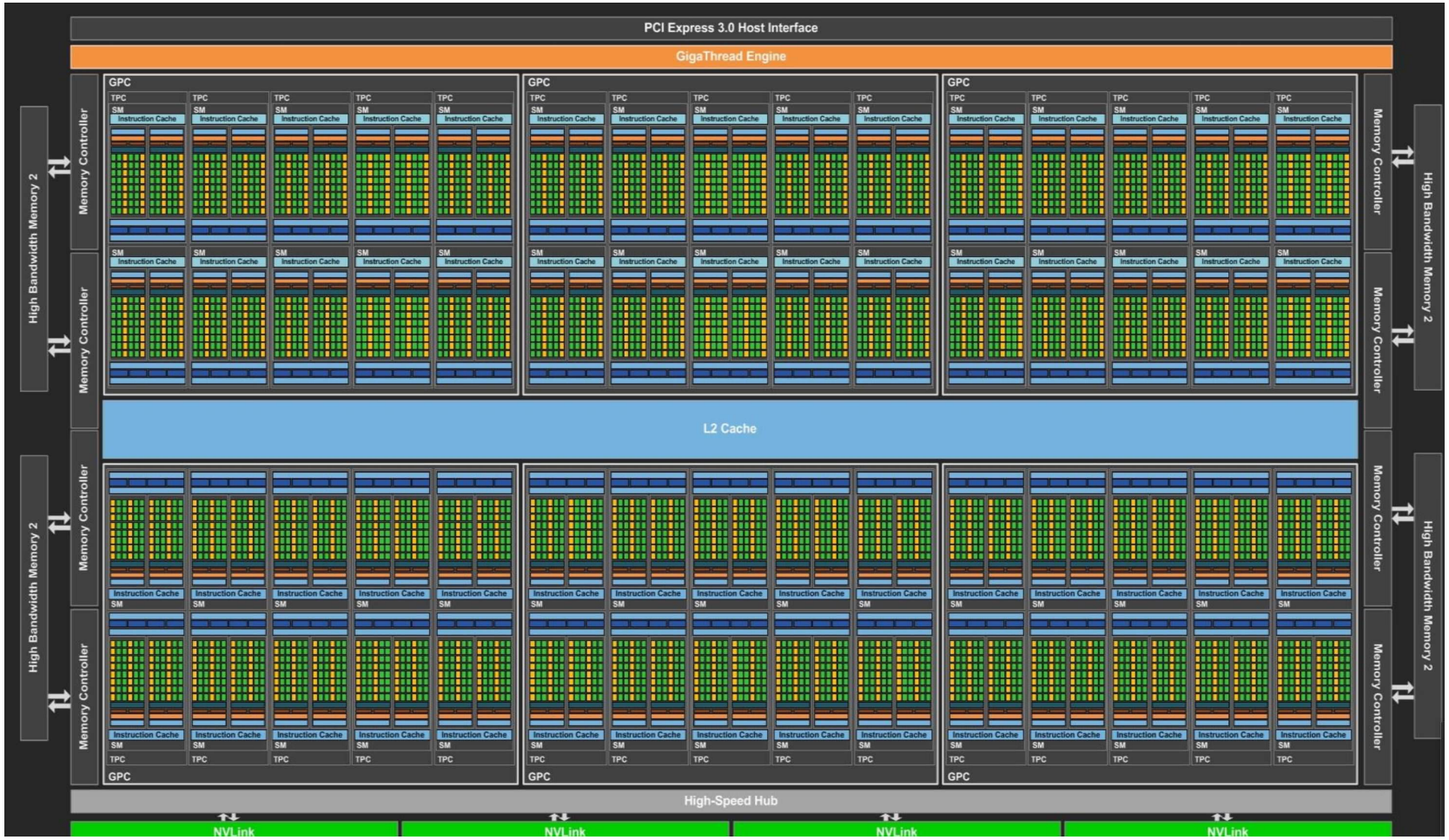




Anyone remember what does warp mean in GPUs?
How to represent warp in the figure?

# Abstraction of GPU architecture



# GPU architecture



# Latency vs throughput

**Latency** is the delay caused by the physical speed of the hardware

CPU = low latency, low throughput

- CPU clock = 3 GHz (3 clocks/ns)
- CPU main memory latency: ~100+ ns
- CPU arithmetic instruction latency: ~1+ ns

GPU = high latency, high throughput

- GPU clock = 1 GHz (1 clock/ns)
- GPU main memory latency: ~300+ ns
- GPU arithmetic instruction latency: ~10+ ns

Above numbers were for Kepler GPUs (e.g. GTX 700 series)

# Compute & IO throughput

- GPU is very IO limited! IO is very often the throughput bottleneck, so its important to be smart about IO.
- If you want to get beyond ~900 GFLOPS, need to do multiple FLOPs per shared memory load.

**NVIDIA A100 TENSOR CORE GPU SPECIFICATIONS (SXM4 AND PCIE FORM FACTORS)**

| | A100 40GB PCIe | A100 80GB PCIe | A100 40GB SXM | A100 80GB SXM |
|---|---|---|---|---|
| FP64 | 9.7 TFLOPS | | | |
| FP64 Tensor Core | 19.5 TFLOPS | | | |
| FP32 | 19.5 TFLOPS | | | |
| Tensor Float 32 (TF32) | 156 TFLOPS \| 312 TFLOPS* | | | |
| BFLOAT16 Tensor Core | 312 TFLOPS \| 624 TFLOPS* | | | |
| FP16 Tensor Core | 312 TFLOPS \| 624 TFLOPS* | | | |
| INT8 Tensor Core | 624 TOPS \| 1248 TOPS* | | | |
| GPU Memory | 40GB HBM2 | 80GB HBM2e | 40GB HBM2 | 80GB HBM2e |
| GPU Memory Bandwidth | 1,555GB/s | 1,935GB/s | 1,555GB/s | 2,039GB/s |

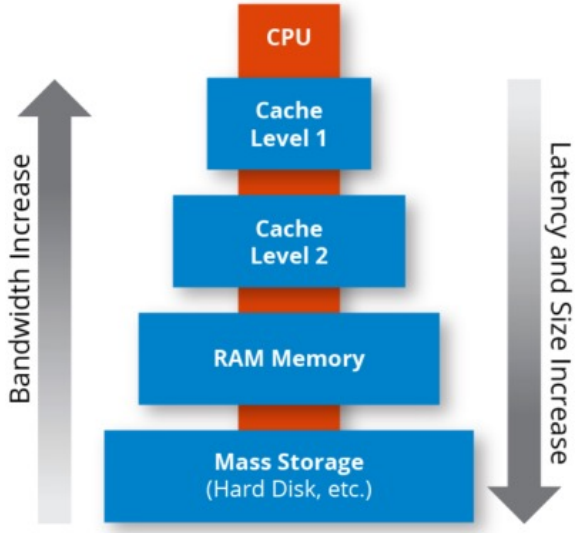
# Cache

A **cache** is a chunk of memory that sits in between a larger pool of memory and the processor

- Often times implemented at hardware level
- Has much faster access speed than the larger pool of memory

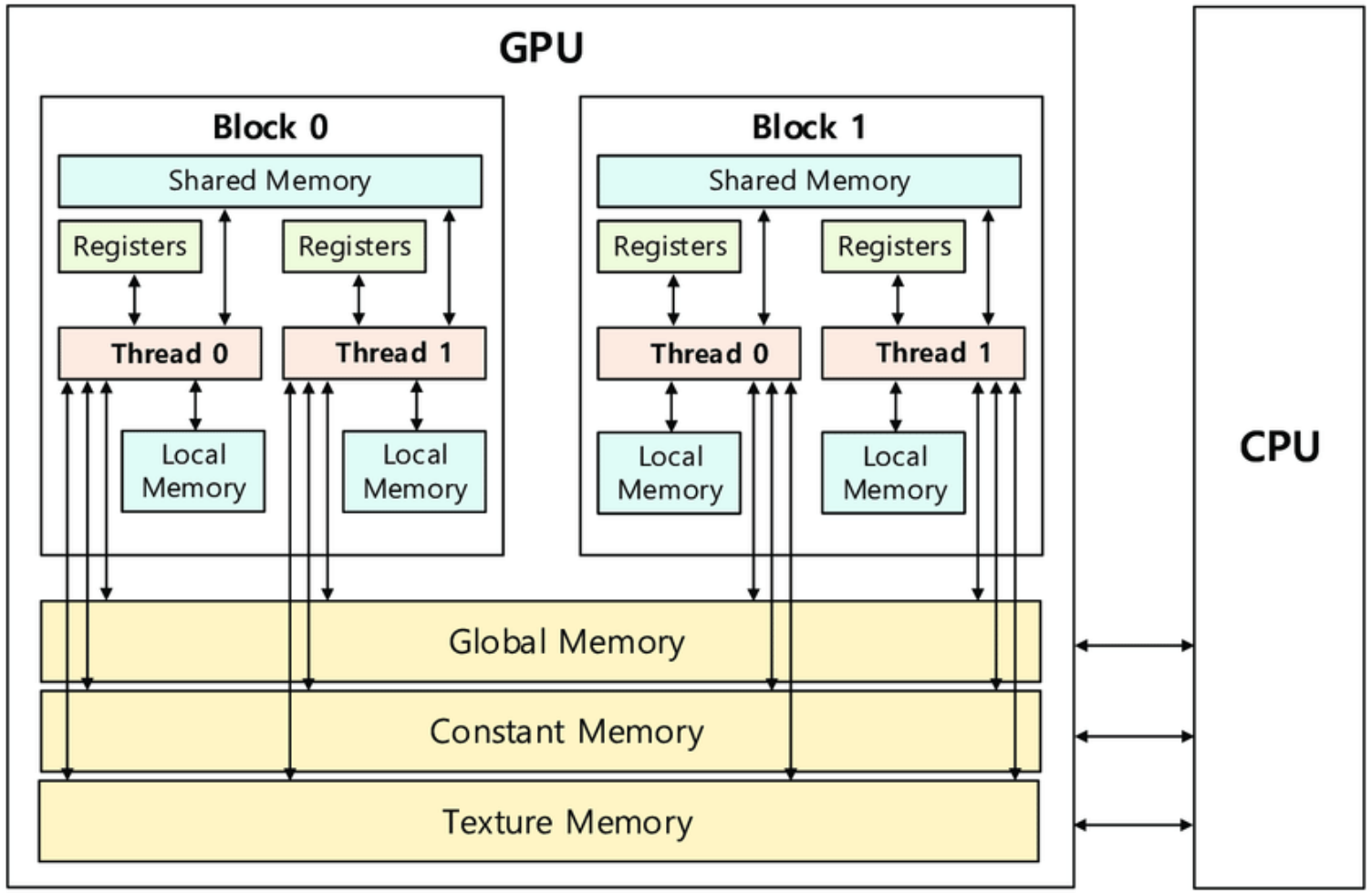
When memory is requested, extra memory near the requested memory is read into a cache

- Amount read is cache and memory pool specific
  - Regions of memory that will always be cached together are called **cache lines**
- This makes future accesses likely to be found in the cache
  - Such accesses are called **cache hits** and allow much faster access
  - If an access is not found in the cache, it's called a **cache miss** (and there is obviously no performance gain)



# GPU Memory

- Registers
- Shared memory
- Constant memory
- Texture memory
- Local memory
- Global memory
- L1/L2 cache
- Read-only cache (CC 3.5+)

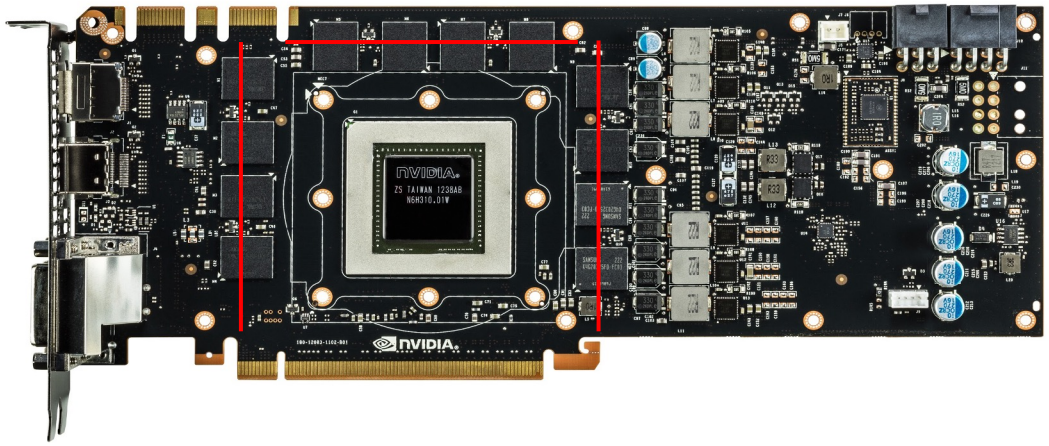


# Global Memory

**Global memory** is separate hardware from the GPU core (containing SM's, caches, etc).

- The vast majority of memory on a GPU is global memory
- If data doesn't fit into global memory, you are going to have process it in chunks that do fit in global memory.
- GPUs have .5 - 24GB of global memory, with most now having ~2GB.

Global memory latency is ~300ns on Kepler and ~600ns on Fermi

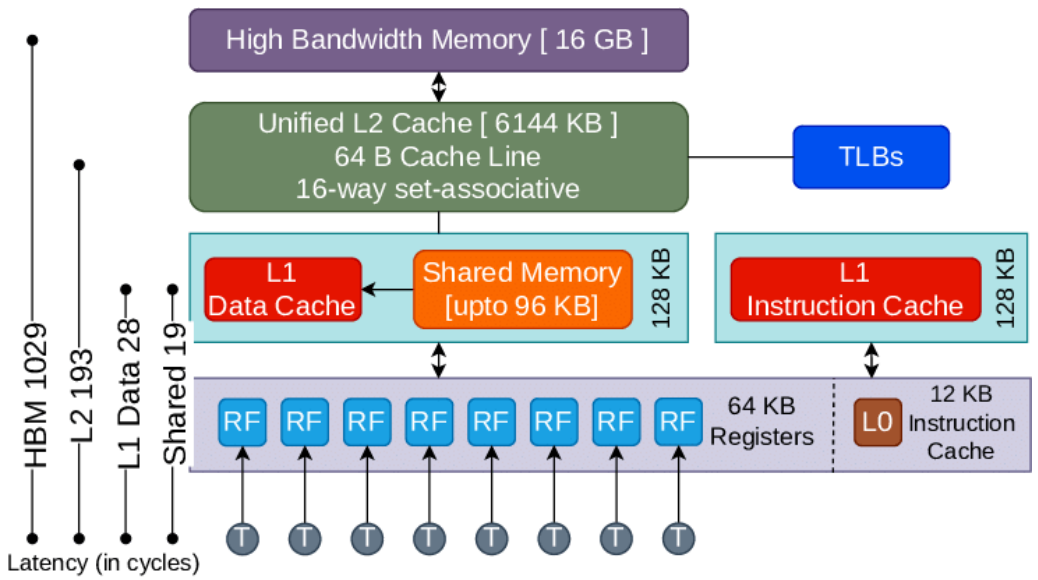


# Global Memory

Global memory IO is the slowest form of IO on GPU

- except for accessing host memory
- Because of this, we want to access global memory as little as possible

Access patterns that play nicely with GPU hardware are called **coalesced memory accesses.**




V100 GPU memory hierarchy and latency comparison [1]

[1] “Accelerating Polynomial Multiplication for Homomorphic Encryption on GPUs”.
K. Shivdikar et al.

# Memory Transactions and Coalescing

Access to global memory triggers transactions

- size of 32 or 128 bytes
- Transaction are aligned to its size
- Transaction are always R/W fully

Coalescing

- adjacent threads can share transactions

degree of coalescing = #bytes requested / #bytes read

more memory read than used => performance penalty

# Coalescing Details

## Details in CUDA Programming Guide

## L1-cached access

- 128-byte transactions
- Used local memory for CC 3.x

## L2-cached access

- 32-byte transactions
- default for CC 3.x global memory

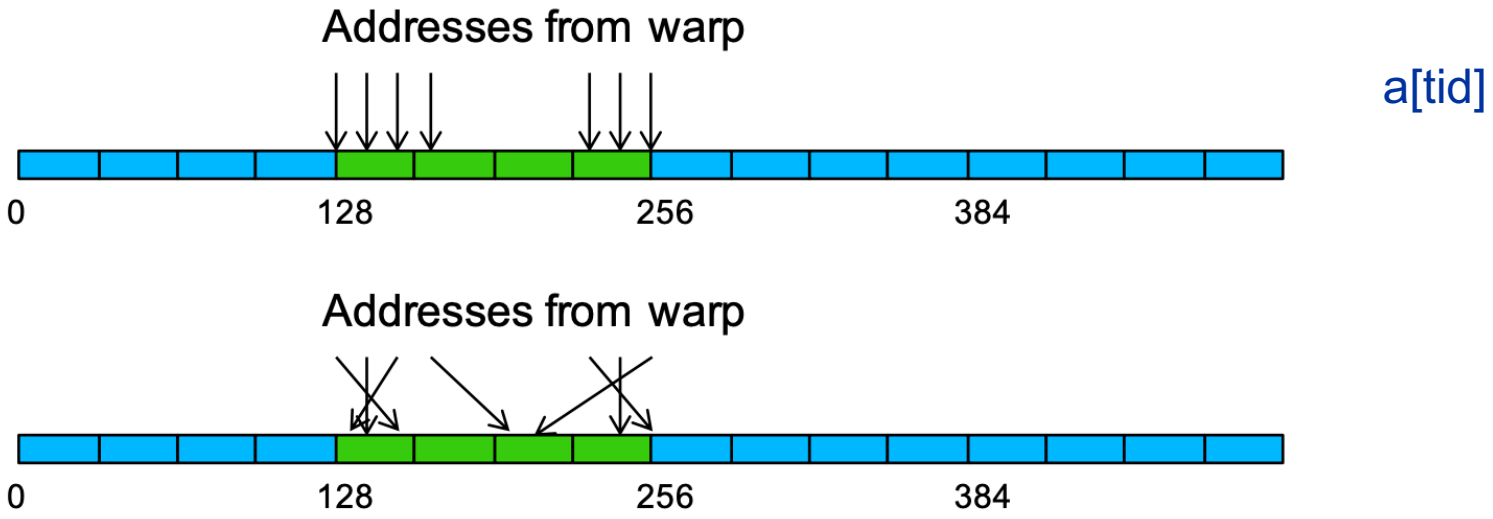
# Example: L2-Cached Thread Index Access

32 adjacent threads requesting 32 aligned/permuted 4-byte words

All addresses fall within 4 segments

Bus utilization: 100%

Transactions: 4




a[tid]

# Example: L2-Cached Shifted Access

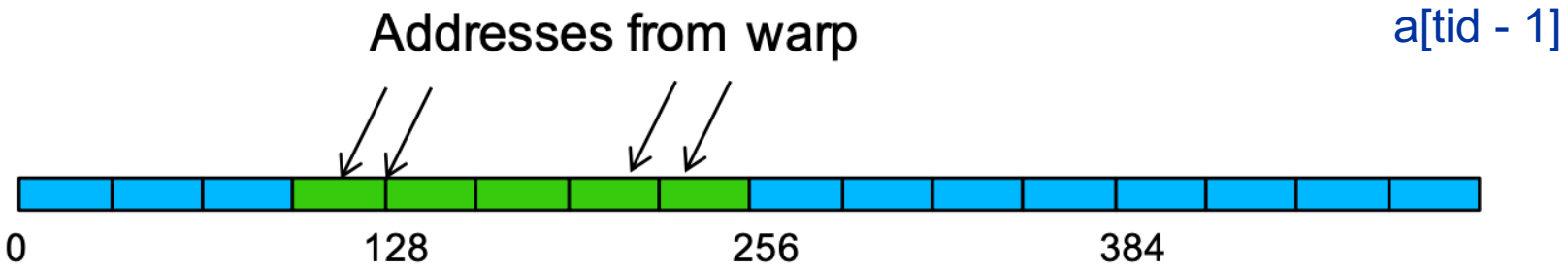
32 adjacent threads requesting 32 aligned 4-byte words

All addresses fall within 5 segments

160 bytes move across bus while 128 bytes are required

Bus utilization: 80%

Transactions: 5



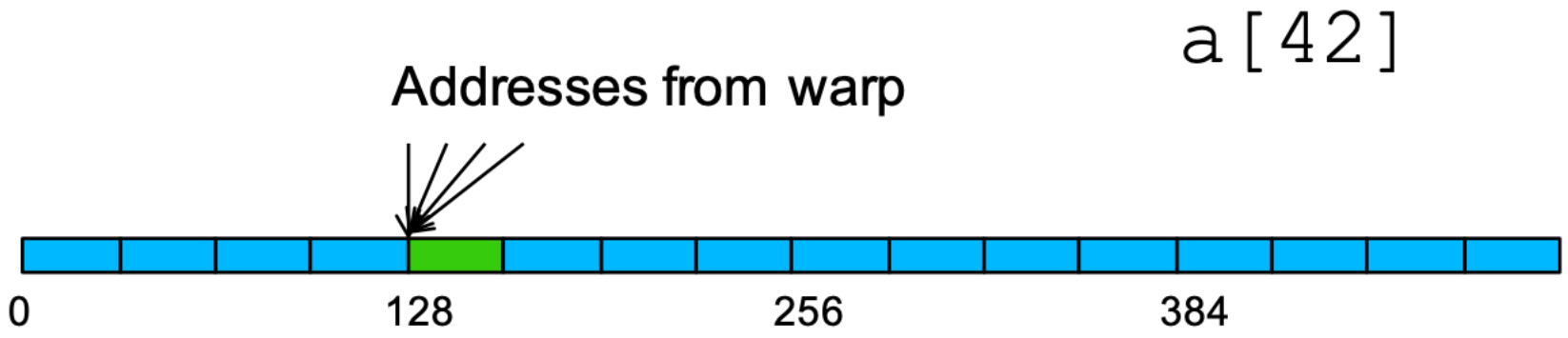
# Example: L2-Cached Single Access

32 adjacent threads requesting the same 4 byte word

Addresses fall in 1 segment

Warp requires 4 bytes but 32 bytes transferred

Bus utilization is only 12.5%



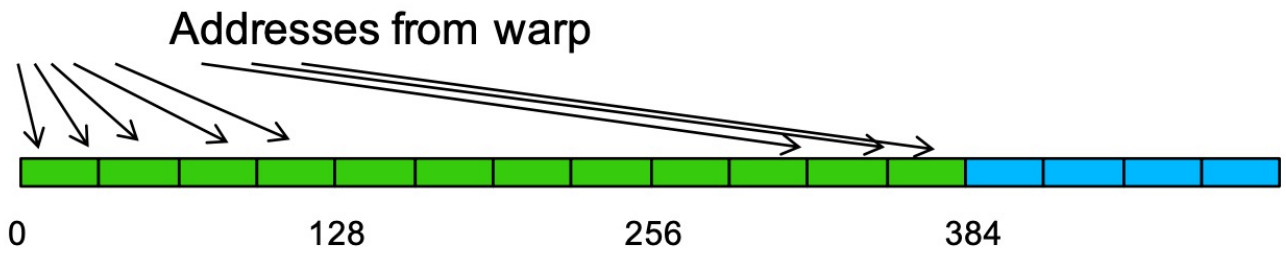
# Example: L2-Cached Strided Access

32 adjacent threads requesting 32 4-byte words with stride 3

All addresses fall in 12 segments

Bus utilization: 33%

Transactions: 12




**struct** {**float** x,y,z;} a; … a[tid].x

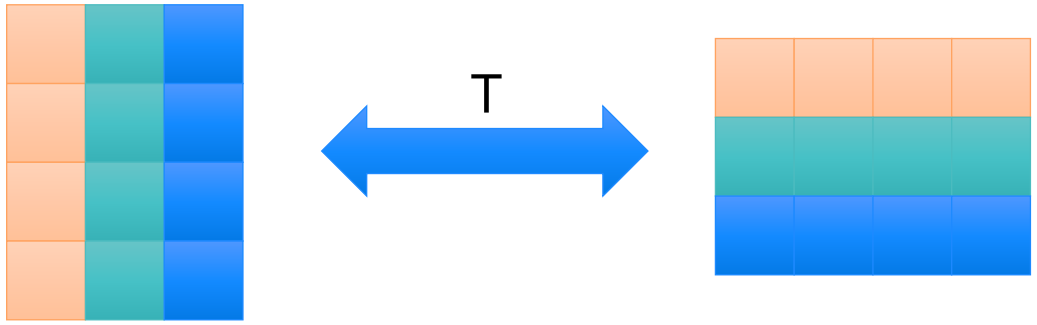
- use structure-of-arrays (SoA) instead of array-of-structures (AoS)

**float** a[M][N]; … a[tid][42]

- multi-dimensional arrays: pay attention to coalescing

# SoA vs AoS

Different data access patterns may also require transposing data structures




Array of Structs

Structure of Arrays

- The cost of a transpose on the data structure is often much less than the cost of uncoalesced memory accesses
- Use shared memory to handle block transposes

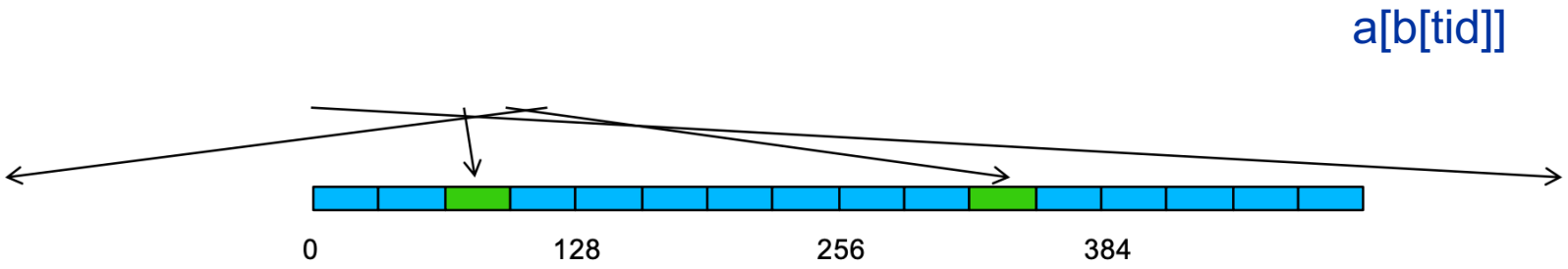
# Example: L2-Cached Fully Random Access

32 adjacent threads requesting 32 4-byte random words

All addresses fall in 32 segments

Bus utilization: only 12.5%

Transactions: 32




pointer chasing: lists, trees etc.

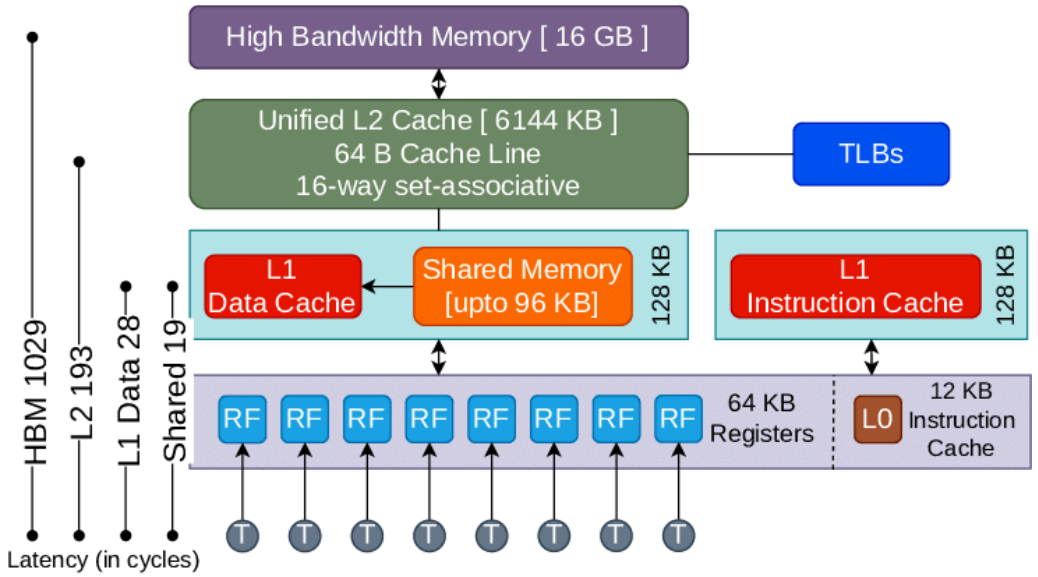
# Shared Memory

- Very fast memory located in the SM
- Same hardware as L1 cache
- Limited size
- Scope of shared memory is the block

Remember

SM = streaming multiprocessor

SM ≠ shared memory




V100 GPU memory hierarchy and latency comparison [1]

[1] “Accelerating Polynomial Multiplication for Homomorphic Encryption on GPUs”.
K. Shivdikar et al.

# Using per-block shared memory

- Access global memory

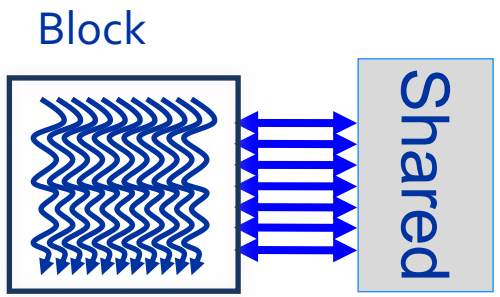
```
int *begin, *end;
```

- Using shared memory

```
__shared__ int scratch[BLOCKSIZE];
scratch[threadIdx.x] = begin[threadIdx.x];
// … compute on scratch values …
begin[threadIdx.x] = scratch[threadIdx.x];
```

- Communicating values between threads

```
scratch[threadIdx.x] = begin[threadIdx.x];
__syncthreads();
int left = scratch[threadIdx.x - 1];
```




Per-block shared memory is faster than L1 cache, slower than register file

It is relatively small: register file is 2-4x larger

# A shared memory application

Task: Compute byte frequency counts

Input: array of bytes of length n

Output: 256 element array of integers containing number of occurrences of each byte

# A shared memory application

Task: Compute byte frequency counts

Input: array of bytes of length n

Output: 256 element array of integers containing number of occurrences of each byte

Naive: build output in global memory, n global stores

Smart: build output in shared memory, copy to global memory at end, 256 global stores

# A common pattern in kernels

1. copy from global memory to shared memory
2. __syncthreads()
3. perform computation, incrementally storing output in shared memory, __syncthreads() as necessary
4. copy output from shared memory to output array in global memory

# Bank Conflicts

Shared memory is setup as 32 **banks**

- If you divide the shared memory into 4 byte-long elements, element `i lies` in bank `i % 32.`

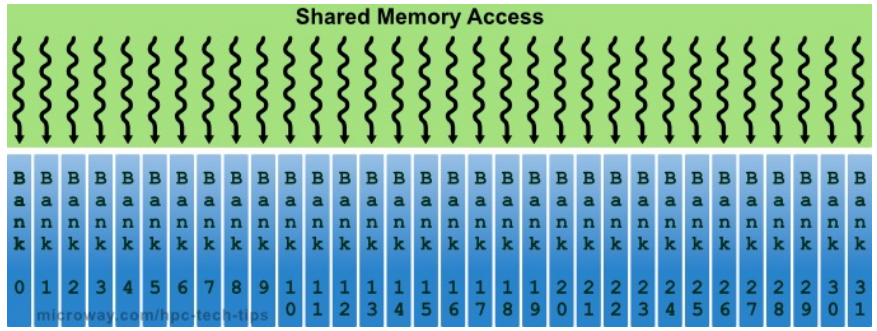
A **bank conflict** occurs when 2 threads in a warp access different elements in the same bank.

- Bank conflicts cause serial memory accesses rather than parallel
  - Serial *anything* in GPU programming = bad for performance

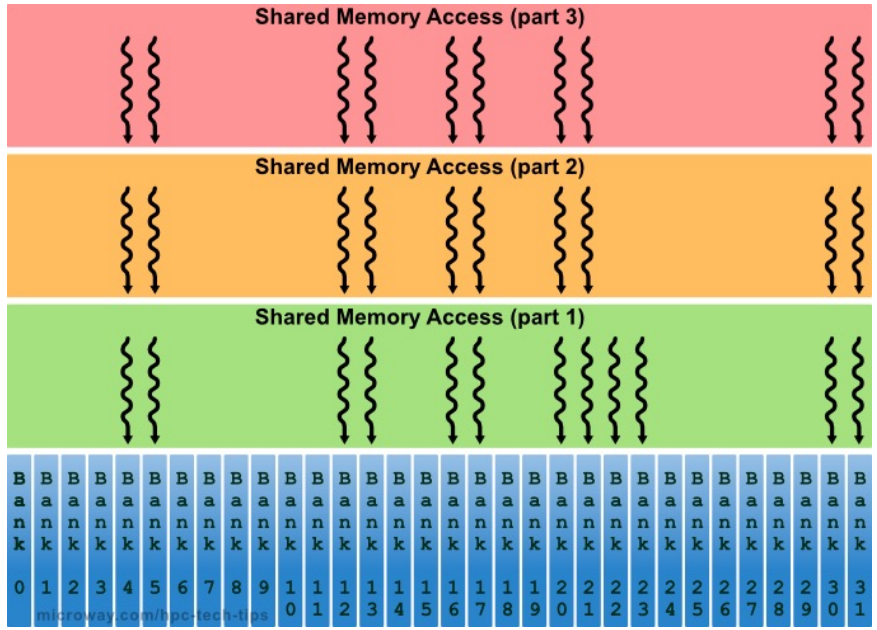
Note: accessing the same element leads to a broadcast instead of bank conflicts

# Bank Conflicts

Bank conflict free




Bank conflict



# Registers

A **Register** is a piece of memory used directly by the processor

- Fastest “memory” possible, about 10x faster than shared memory
- There are tens of thousands of registers in each SM
  - Generally works out to a maximum of 32 or 64 32-bit registers per thread

Most stack variables declared in kernels are stored in registers

- example: `float x; (duh...)`

Statically indexed arrays stored on the stack are sometimes put in registers

# Local Memory

**Local memory** is everything on the stack that can't fit in registers

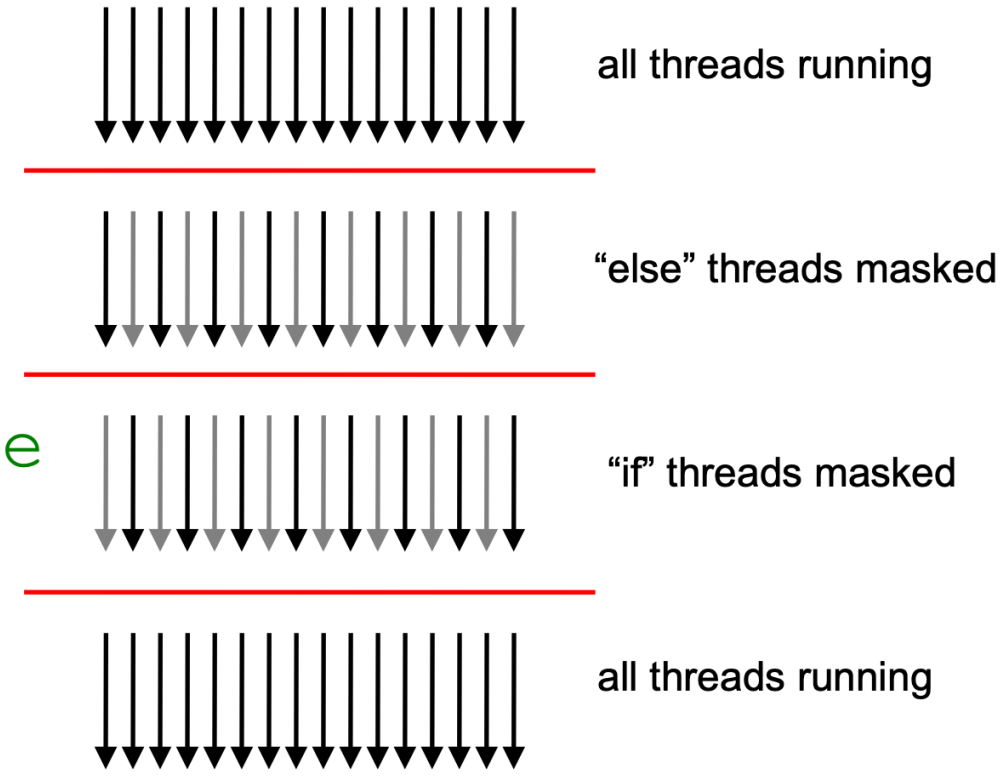
The scope of local memory is just the thread.

Local memory is stored in global memory

- much slower than registers

# Branch Divergence Within Warp

```
// ...
if(condition) {
   // do smth
   // ...
} else {
   // do smth else
   // ...
}
// ...
```

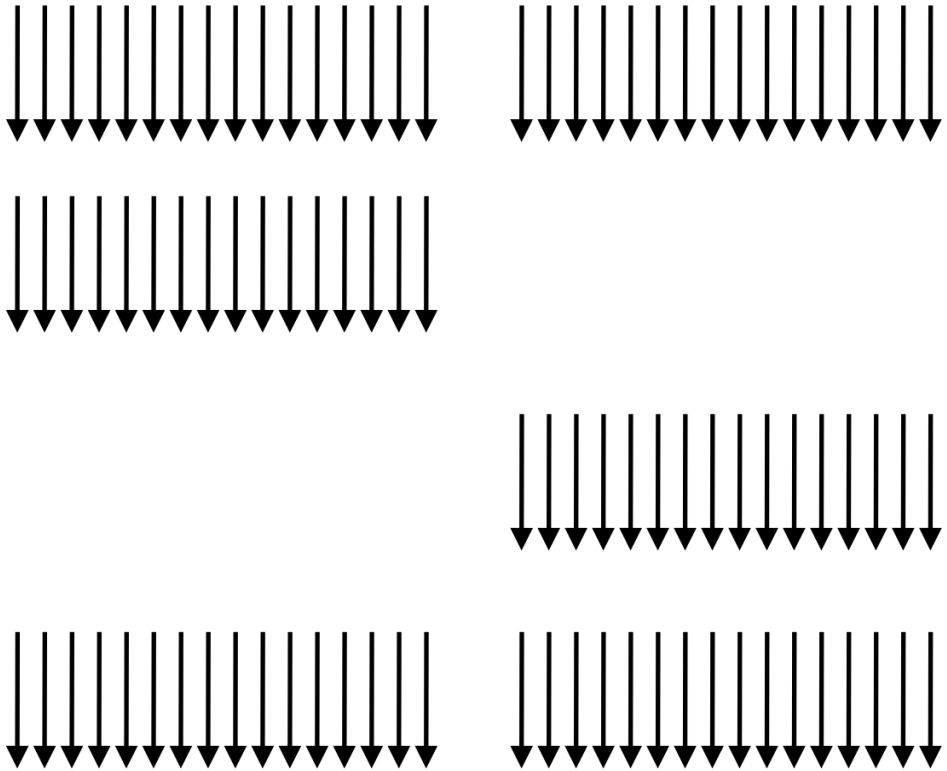



divergence *within* warp => performance penalty

```
if(threadIdx.x % 2 == 0) ...
```

# Branch Divergence Between Warp

```
// ...
if(condition) {
   // do smth
   // ...
} else {
   // do smth else
   // ...
}
// ...
```

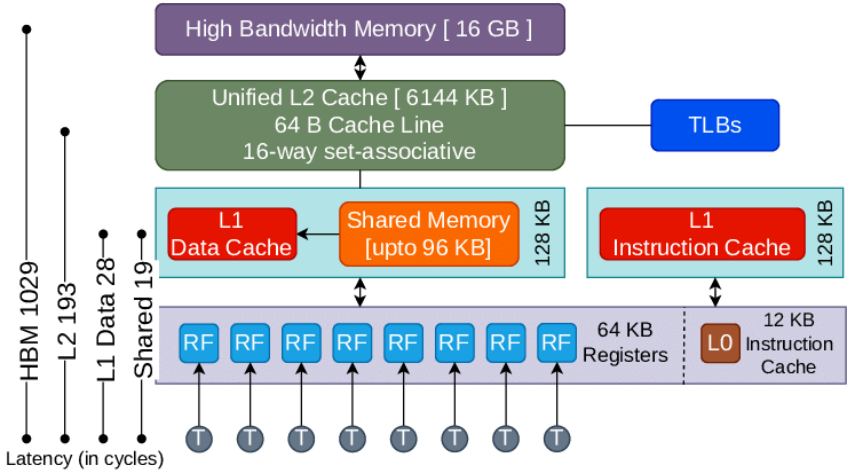


divergence *between* warps => no penalty

```
if(blockIdx.x % 2 == 0) ...
```

# Summary

- Global memory
- Local memory
- Shared memory
- Registers




- To achieve coalesced global memory access
  - Try to use shared memory
  - Look for different way of storage or better algorithm
- Avoid bank conflicts
- Avoid register spilling
- Avoid divergent branches

# Hands-on exercise

Parallel reduction:
Compute the reduction result (e.g., sum) of an array.

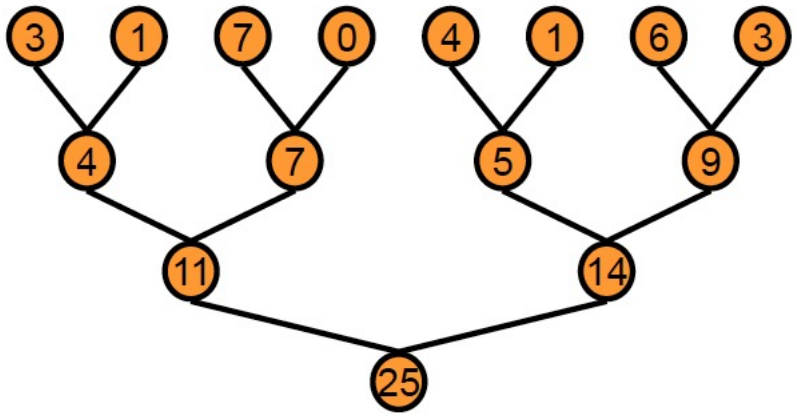
3 1 7 0 4 1 6 3 → 25

How will you implement the algorithm in CUDA?

Example and materials provided by Mark Harris from NVIDIA
https://developer.download.nvidia.com/assets/cuda/files/reduction.pdf

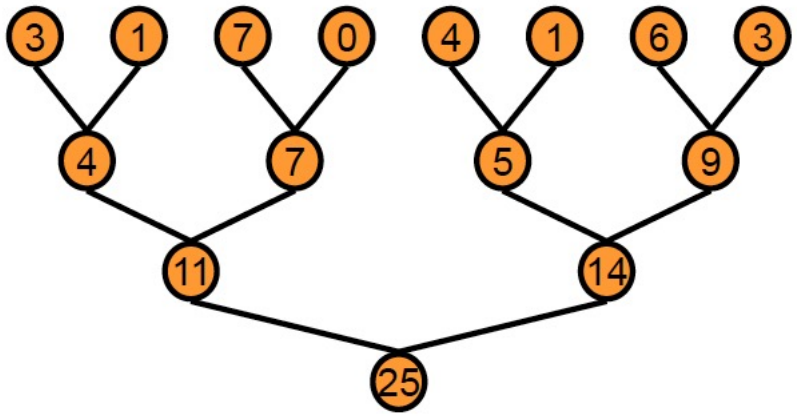
# Problem

Consider a tree-based approach. What are the problems in implementing it using CUDA?



# Problem

Consider a tree-based approach. What are the problems in implementing it using CUDA?




Need to be able to use multiple thread blocks

- To process very large arrays
- To keep all multiprocessors on the GPU busy
- Each thread block reduces a portion of the array

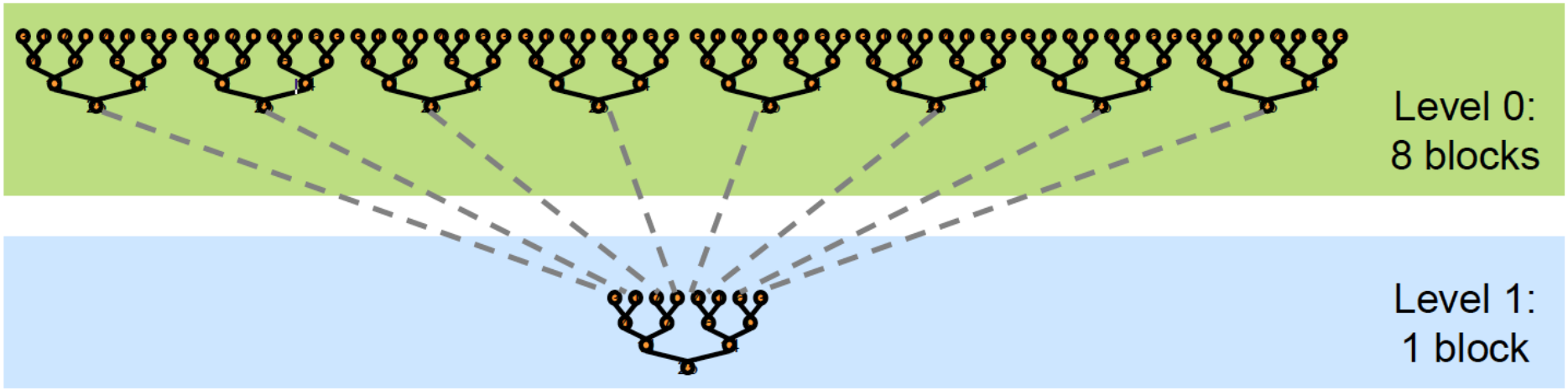
But how do we communicate partial results between thread blocks?

# Challenges and solutions

- If we could synchronize across all thread blocks, could easily reduce very large arrays, right?
  - Global sync after each block produces its result
  - Once all blocks reach sync, continue recursively
- But CUDA has no global synchronization. Why?
  - Expensive to build in hardware for GPUs with high processor count
  - Would force programmer to run fewer blocks (no more than # multiprocessors * # resident blocks / multiprocessor) to avoid deadlock, which may reduce overall efficiency
- Solution: decompose into multiple kernels
  - Kernel launch serves as a global synchronization point
  - Kernel launch has negligible HW overhead, low SW overhead

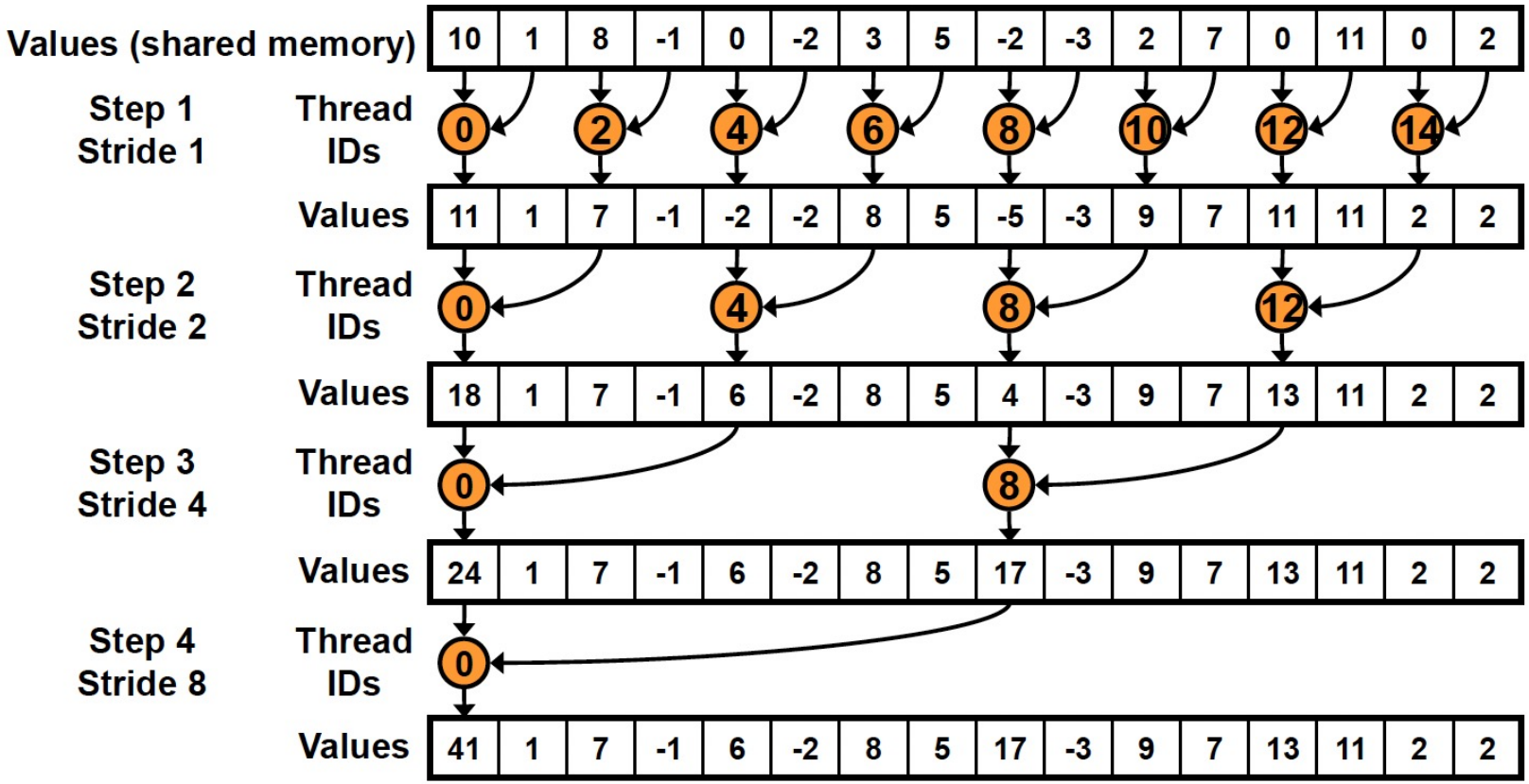
# Kernel decomposition

Avoid global sync by decomposing computation into multiple kernel invocations:




For reduction, the same code can be used for all the levels

# Interleaved addressing



# Interleaved addressing

```
__global__ void reduce0(int *g_idata, int *g_odata) {
    extern __shared__ int sdata[];

    // each thread loads one element from global to shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
    sdata[tid] = g_idata[i];
    __syncthreads();

    // do reduction in shared mem
    for(unsigned int s=1; s < blockDim.x; s *= 2) {
        if (tid % (2*s) == 0) {
            sdata[tid] += sdata[tid + s];
        }
        __syncthreads();
    }

    // write result for this block to global mem
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

# Interleaved addressing

```
__global__ void reduce1(int *g_idata, int *g_odata) {
   extern __shared__ int sdata[];

   // each thread loads one element from global to shared mem
   unsigned int tid = threadIdx.x;
   unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
   sdata[tid] = g_idata[i];
   __syncthreads();

   // do reduction in shared mem
   for (unsigned int s=1; s < blockDim.x; s *= 2) {
      if (tid % (2*s) == 0) {
         sdata[tid] += sdata[tid + s];
      }
      __syncthreads();
   }

   // write result for this block to global mem
   if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```

**Problem: highly divergent warps are very inefficient, and % operator is very slow**

# Replace divergent branch

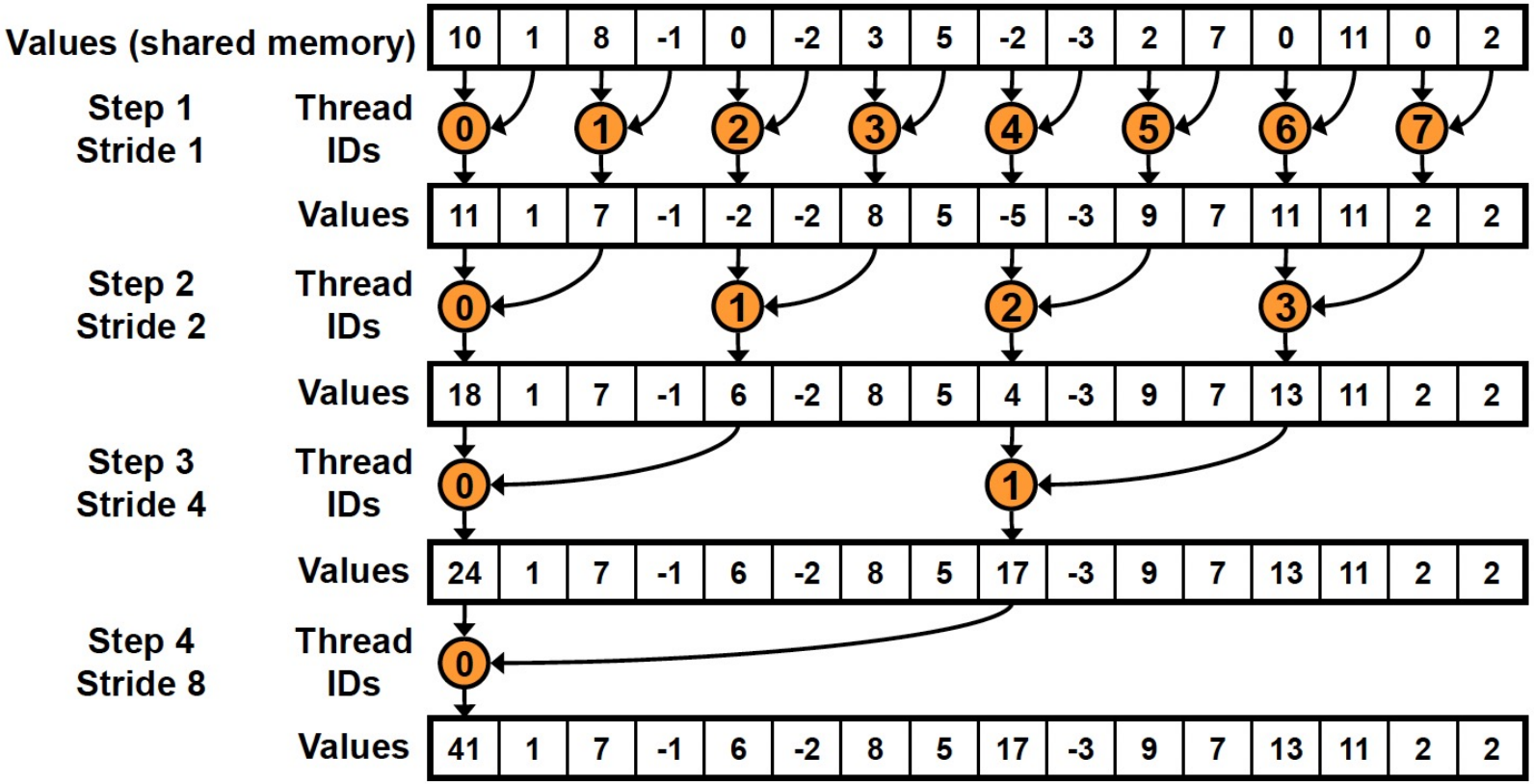
**Just replace divergent branch in inner loop:**

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

**With strided index and non-divergent branch:**

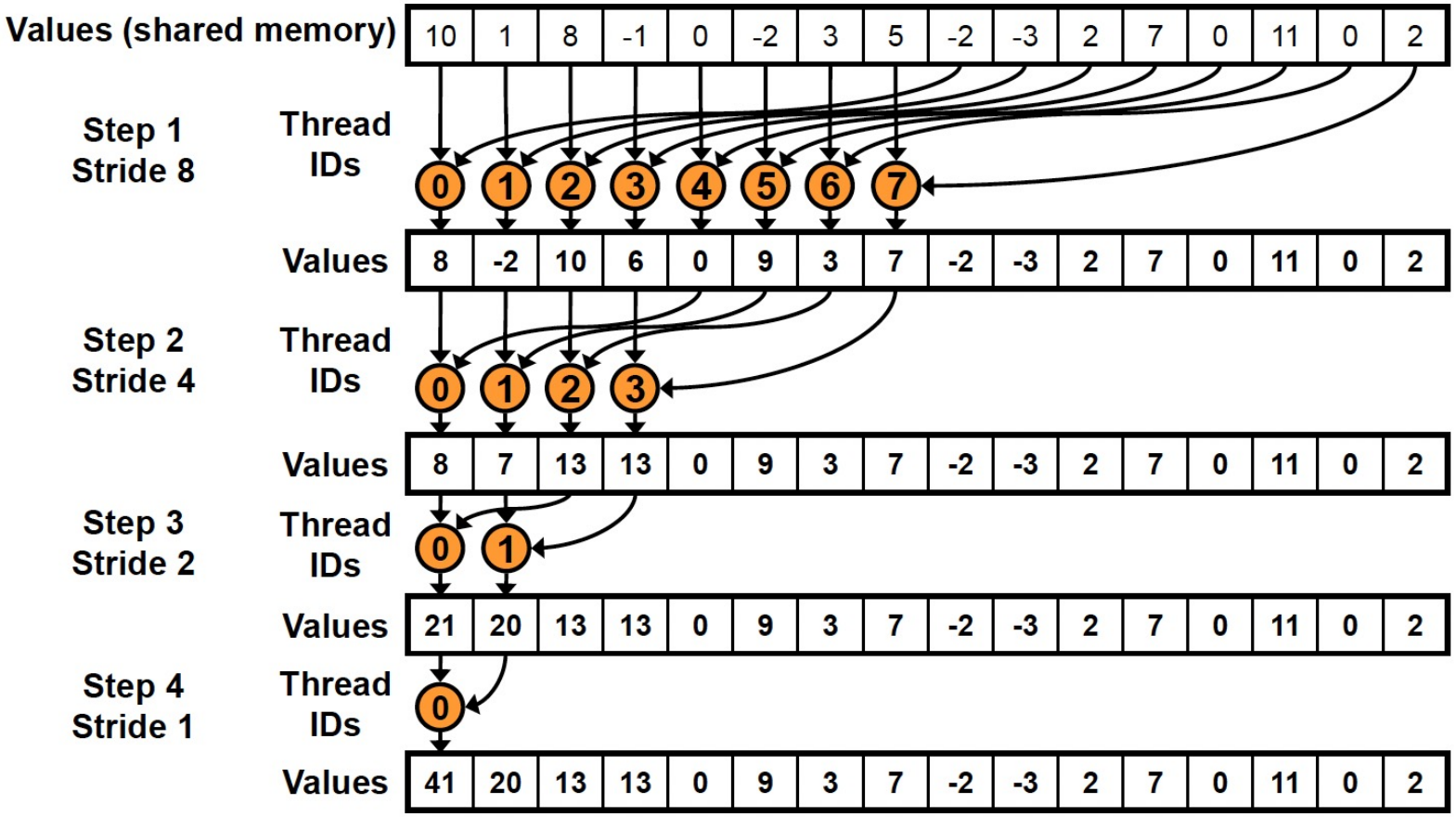
```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

# Replace divergent branch




**New Problem: Shared Memory Bank Conflicts**

# Sequential addressing




**Sequential addressing is conflict free**

# Sequential addressing

**Just replace strided indexing in inner loop:**

```
for (unsigned int s=1; s < blockDim.x; s *= 2)  {
    int index = 2 * s * tid;

    if (index < blockDim.x) {
        sdata[index] += sdata[index + s];
    }
    __syncthreads();
}
```

**With reversed loop and threadID-based indexing:**

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

# Problem

**Problem:**

```
for (unsigned int s=blockDim.x/2; s>0; s>>=1) {
    if (tid < s) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

**Half of the threads are idle on first loop iteration!**

**This is wasteful…**

# First add during reading

**Halve the number of blocks, and replace single load:**

```
// each thread loads one element from global to shared mem
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();
```

**With two loads and first add of the reduction:**

```
// perform first level of reduction,
// reading from global memory, writing to shared memory
unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*(blockDim.x*2) + threadIdx.x;
sdata[tid] = g_idata[i] + g_idata[i+blockDim.x];
__syncthreads();
```

# Loop unrolling

- As reduction proceeds, # “active” threads decreases
  - When s <= 32, we have only one warp left
- Instructions are SIMD synchronous within a warp
- That means when s <= 32:
  - We don’t need to __syncthreads()
  - We don’t need “if (tid < s)” because it doesn’t save any work
- Let’s unroll the last 6 iterations of the inner loop

# Loop unrolling

```
__device__ void warpReduce(volatile int* sdata, int tid) {
      sdata[tid] += sdata[tid + 32];
      sdata[tid] += sdata[tid + 16];
      sdata[tid] += sdata[tid +  8];
      sdata[tid] += sdata[tid +  4];
      sdata[tid] += sdata[tid +  2];
      sdata[tid] += sdata[tid +  1];
 }
```

IMPORTANT:
For this to be correct, we must use the "volatile" keyword!

```
// later…
 for (unsigned int s=blockDim.x/2; s>32; s>>=1) {
      if (tid < s)
         sdata[tid] += sdata[tid + s];
      __syncthreads();
 }

 if (tid < 32) warpReduce(sdata, tid);
```

**Note: This saves useless work in *all* warps, not just the last one!**
Without unrolling, all warps execute every iteration of the for loop and if statement

# Complete unrolling

- If we knew the number of iterations at compile time, we could completely unroll the reduction
  - Luckily, the block size is limited by the GPU to 512 threads
  - Also, we are sticking to power-of-2 block sizes
- So we can easily unroll for a fixed block size
  - But we need to be generic – how can we unroll for block sizes that we don't know at compile time?
- Templates to the rescue!
  - CUDA supports C++ template parameters on device and host functions

```
template <unsigned int blockSize>
__global__ void reduce5(int *g_idata, int *g_odata)
```

# Complete unrolling

```
Template <unsigned int blockSize>
__device__ void warpReduce(volatile int* sdata, int tid) {
      if (blockSize >= 64) sdata[tid] += sdata[tid + 32];
      if (blockSize >= 32) sdata[tid] += sdata[tid + 16];
      if (blockSize >= 16) sdata[tid] += sdata[tid +  8];
      if (blockSize >=   8) sdata[tid] += sdata[tid +  4];
      if (blockSize >=   4) sdata[tid] += sdata[tid +  2];
      if (blockSize >=   2) sdata[tid] += sdata[tid +  1];
 }
```

```
   if (blockSize >= 512) {
      if (tid < 256) { sdata[tid] += sdata[tid + 256]; } __syncthreads(); }
   if (blockSize >= 256) {
      if (tid < 128) { sdata[tid] += sdata[tid + 128]; } __syncthreads(); }
   if (blockSize >= 128) {
      if (tid <  64)  { sdata[tid] += sdata[tid +   64]; } __syncthreads(); }

   if (tid < 32) warpReduce<blockSize>(sdata, tid);
```

**Note: all code in RED will be evaluated at compile time.**

Results in a very efficient inner loop!

# Complete unrolling

- **Don't we still need block size at compile time?**
  - **Nope, just a switch statement for 10 possible block sizes**

```
switch (threads)
    {
    case 512:
      reduce5<512><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 256:
      reduce5<256><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 128:
      reduce5<128><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 64:
      reduce5<  64><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 32:
      reduce5<  32><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case 16:
      reduce5<  16><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  8:
      reduce5<   8><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  4:
      reduce5<   4><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  2:
      reduce5<   2><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    case  1:
      reduce5<   1><<< dimGrid, dimBlock, smemSize >>>(d_idata, d_odata); break;
    }
```

# Efficiency

- Log($N$) parallel steps, each step $S$ does $N/2^S$ independent ops
  - Step Complexity is O(log $N$)
- For $N=2^D$, performs $\sum_{S\in[1..D]} 2^{D-S} = N-1$ operations
  - Work Complexity is O($N$) – It is work-efficient
  - i.e. does not perform more operations than a sequential algorithm
- With $P$ threads physically in parallel ($P$ processors), time complexity is O($N/P$ + log $N$)
  - Compare to O($N$) for sequential reduction
  - In a thread block, N=P, so O(log N)

# Cost

- *Cost* of a parallel algorithm is processors time complexity
  - Allocate threads instead of processors: O(*N*) threads
  - Time complexity is O(log *N*), so *cost* is O(*N* log *N*) : not cost efficient!
- Brent's theorem suggests O(*N*/log *N*) threads
  - Each thread does O(log *N*) sequential work
  - Then all O(*N*/log *N*) threads cooperate for O(log *N*) steps
  - Cost = O((*N*/log *N*) * log *N*) = O(*N*) → cost efficient
- Sometimes called *algorithm cascading*
  - Can lead to significant speedups in practice

# Practice

Test your code or the provided implementation in our hands-on exercise repo. Check how the execution time differs with:

- Different reduction implementations
- Different input sizes
- Different number of threads per block

# Any questions so far?