



CUDA Training

GPGPU Programming

Programming and Execution model for GPUs

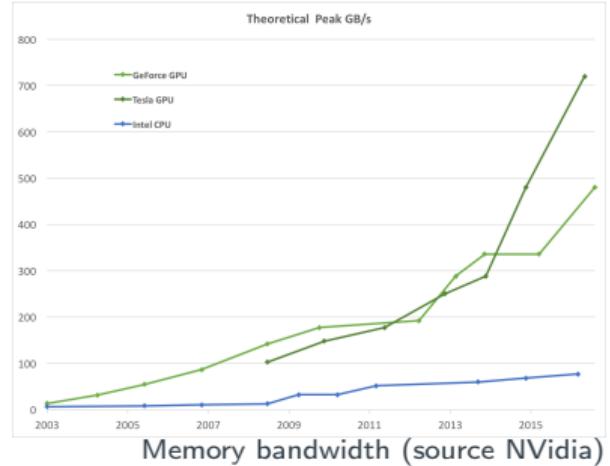
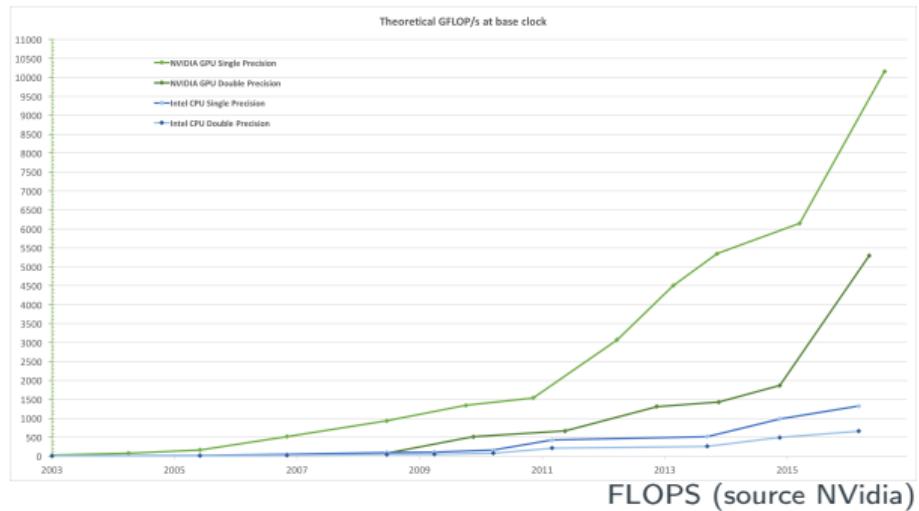
João Barbosa
jbarbosa@tacc.utexas.edu
September, 2018

Outline

- Why do we care about General Purpose GPUs?
- Brief history of the GPU
- Programming and Execution model
- More advanced concepts

Why do we care about General Purpose GPUs?

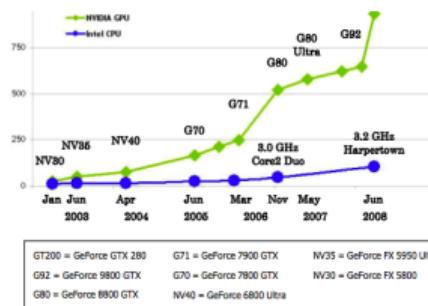
Performance Trends



Brief history of the GPU

Introduction to GPU Technology

- ▶ **Definition and Purpose:** GPUs (Graphics Processing Units) are specialized processors initially designed for rendering images and handling graphical computations.
- ▶ **Early Development:**
 - 1980s: Basic graphics hardware was used in gaming consoles and arcade systems.
 - 1990s: Introduction of 3D graphics in personal computers with basic GPUs.
- ▶ **Key Milestone:** The emergence of NVIDIA in 1993, which led to major advancements in GPU technology.



Evolution of GPUs for General Purpose Computing

► Graphics to General-Purpose Computing

- 2000s: GPUs evolved beyond just graphics, with capabilities supporting broader computational tasks.

► Parallel Processing:

Unlike CPUs, GPUs can handle thousands of tasks simultaneously, making them ideal for parallel processing.

► Rise in Popularity:

Researchers and developers began using GPUs for scientific computing, machine learning, and data analysis.



Introduction of CUDA

► Launch of CUDA:

- 2006: NVIDIA introduced CUDA, a groundbreaking platform allowing developers to use GPUs for general-purpose computing.

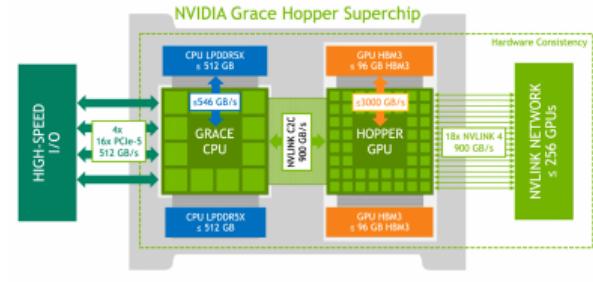
► What is CUDA? A parallel computing platform and API that gives direct access to the GPU's virtual instruction set and parallel computational elements.

► Impact of CUDA: Made GPUs more accessible for developers, leading to advancements in AI, deep learning, and large-scale computations.

GPU Computing Applications						
Libraries and Middleware						
cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSTL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica
Programming Languages						
C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)	
CUDA-Enabled NVIDIA GPUs						
Volta Architecture (compute capabilities 7.x)						Tesla V Series
Pascal Architecture (compute capabilities 6.x)			GeForce 1000 Series	Quadro P Series		Tesla P Series
Maxwell Architecture (compute capabilities 5.x)	Tegra X1		GeForce 900 Series	Quadro M Series		Tesla M Series
Kepler Architecture (compute capabilities 3.x)	Tegra K1		GeForce 700 Series GeForce 600 Series	Quadro K Series		Tesla K Series

GPU and CUDA Today

- ▶ **AI and Deep Learning:** GPUs are the backbone of deep learning due to their ability to handle extensive matrix and tensor operations.
- ▶ **CUDA Ecosystem:** Expansions like CUDA libraries, tools, and frameworks (e.g., cuDNN for deep learning).
- ▶ **Continued Evolution:** Ongoing innovations in GPU architectures (such as NVIDIA's Ampere and Hopper) continue to enhance GPU efficiency and scalability.
- ▶ **Conclusion:** GPU and CUDA have revolutionized computing, making high-performance processing accessible across industries.



NVidia - Volta GV100 GPU Stream Multiprocessor

Stream Multiprocessor

- ▶ 128 KB L1/Shared memory
- ▶ 32 Double precision units
- ▶ 64 Single precision units
- ▶ 64 Scalar cores
- ▶ 8 mixed-precision Tensor Cores

$$D = A \cdot B + C$$

where A,B, C and D are 4x4 Matrices



NVidia Volta (GV100) Stream Multiprocessor (source NVidia)

Difference between a CPU and GPU



Abstract CPU and GPU Architecture diagrams (source NVidia)

- ▶ **Minimize latency** of a single thread
 - Larger on chip cache
 - Complex logic (e.g. branch prediction)
- ▶ Complex cores → Lower core count
- ▶ **Maximize throughput** of all threads
 - Lots of resources
 - ▶ Simpler cores
 - ▶ Smaller caches
 - Control shared across multiple threads (warp)
 - ▶ Simpler cores → Higher core count

Programming and Execution model

Programming and Execution model

Caveat

- ▶ We are going to use the CUDA programming and execution model from NVidia.
- ▶ The other common portable programming and execution model is OpenCL, which is similar to CUDA minor some names and syntax changes.
- ▶ Microsoft has its own, called Direct Compute, but it is similar to CUDA and OpenCL
- ▶ OpenCL and Direct Compute can be used in all GPUs brands (CPUs also for OpenCL) not just NVidia.

CUDA - Compute Unified Device Architecture

Both an architecture and programming model

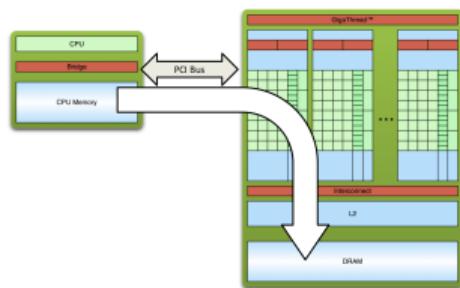
- ▶ Architecture and execution model
 - Introduced in NVIDIA in 2007
 - Get highest possible execution performance requires understanding of hardware architecture
- ▶ Programming model
 - Small set of extensions to C / C++
 - Enables GPUs to execute programs written in C / C++
 - Within C programs, call SIMD “kernel” routines that are executed on GPU.

CUDA - Programming and Execution model

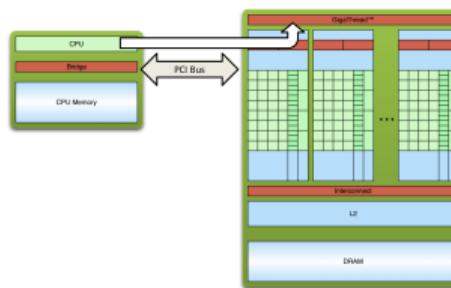
host The CPU and its memory (host memory)

device The GPU and its memory (device memory)

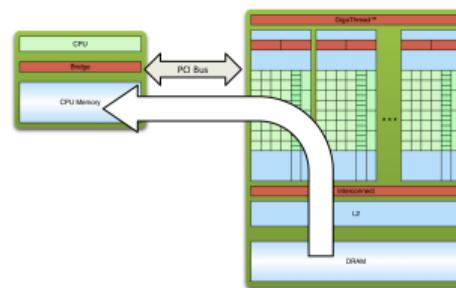
kernel The computational payload called from the host to be executed on the device



Copy data from the host to the Device



Invoke the kernel on the host



Copy data from the device to the host

CUDA - Thread Hierarchical Model

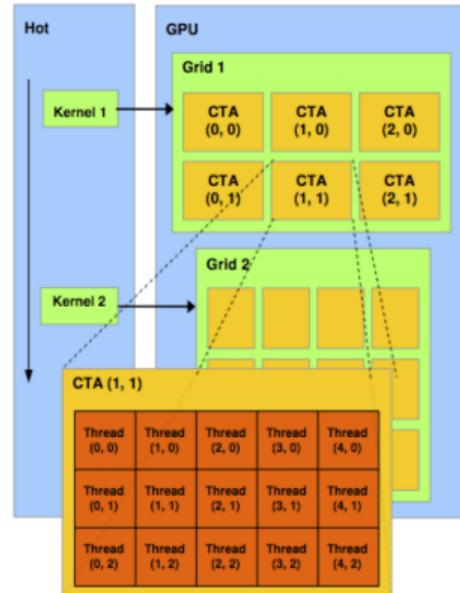
Thread – Unique ID

Block (3 dimensions)

- Threads within the same block can cooperate
- Threads within the same block execute the same code

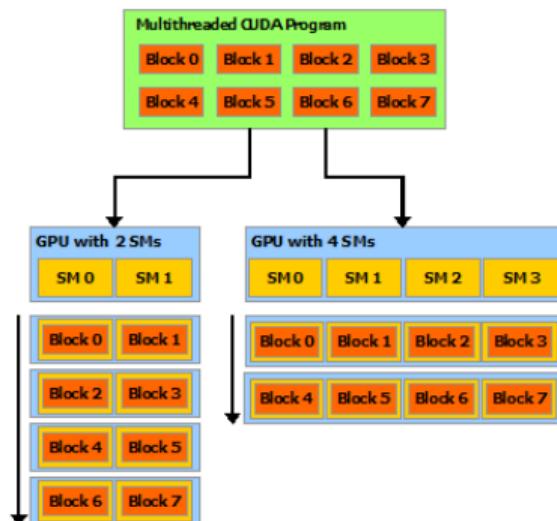
Grid (2 dimensions)

- Blocks are organized into a grid
- Blocks **cannot** cooperate



CUDA thread hierarchy (source NVidia)

Thread/Block Hierarchy transparency



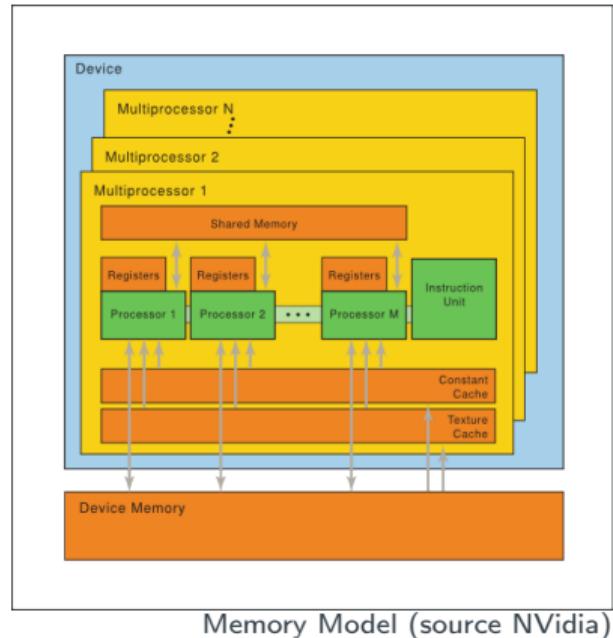
source: NVidia

- ▶ Blocks can be assigned arbitrarily to any processor
- ▶ Increase scalability to GPU architecture design

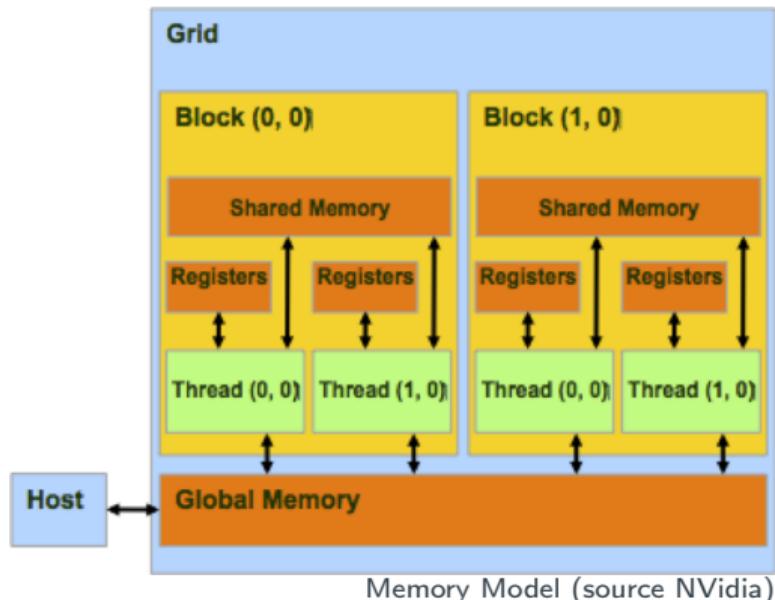
Blocks must be independent for this reason, to accommodate various GPU architectures

CUDA - Memory Model - Scope

- ▶ Thread
 - R/W Registers
 - *Local* memory
- ▶ Block
 - Shared memory
- ▶ Grid (Host also)
 - Device memory [RW]
 - Constant memory [RO]
 - Texture Memory [RO]



CUDA - Memory Model



► Device Memory

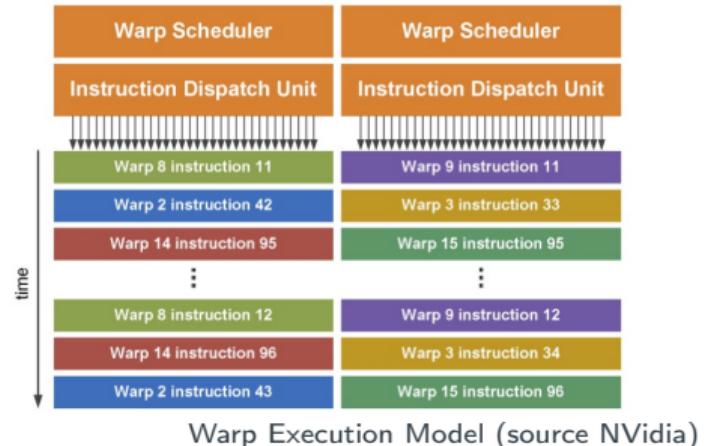
- Communication between host and device
- Content visible to all threads
- Long latency access

► Shared Memory

- Low latency access [L1 Cache Level]
- Used for intra-block / inter-thread cooperation

CUDA - Execution Model

- ▶ Thread Block
 - Schedule until completion on the same Steam Multiprocessor (SM)
 - Max 1024 threads (pre-Volta)
- ▶ Warp
 - 32 threads of the same warp
 - Shared the same program counter (pre-Volta)
 - Scheduled Interleaved in the SM



CPU - Vector Add

```
void vectoradd(float* a, float* b, float* c, int N) {
    for(int i = 0; i < N; i++) {
        c[i] = a[i] + b[1];
    }
}

int main(int argc, char* argv[]) {
    int N = 32;
    float* a = (float*)malloc(N*sizeof(float));
    float* b = (float*)malloc(N*sizeof(float));
    float* c = (float*)malloc(N*sizeof(float));

    vectoradd(a,b,c,N);

    // do something with c

    free(a); free(b), free(c);
}
```

GPU / CUDA - Vector Add

```
--global__ void vectoradd(float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x
    c[i] = a[i] + b[1];
}

int main(int argc, char* argv[]) {
    int N = 32;
    float* a, *b, *c;

    cudaMallocManaged(&a, N*sizeof(float));
    cudaMallocManaged(&b, N*sizeof(float));
    cudaMallocManaged(&c, N*sizeof(float));

    vectoradd<<<
        1, /* grid configuration */
        N /* block configuration */
    >>>(a,b,c,N);

    // do something with c

    cudaFree(a); cudaFree(b), cudaFree(c);
}
```



GPU / CUDA - Vector Add - Defining block size and number of blocks

```
--global__ void vectoradd(float* a, float* b, float* c, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x
    c[i] = a[i] + b[1];
}

int main(int argc, char* argv[]) {
    int N = 1 << 32;
    float* a, *b, *c;

    cudaMallocManaged(&a, N*sizeof(float));
    cudaMallocManaged(&b, N*sizeof(float));
    cudaMallocManaged(&c, N*sizeof(float));

    int thread_block_size = 1024;
    int number_of_blocks = (int)ceil(float(N) / thread_block_size) ;

    vectoradd<<<number_of_blocks,thread_block_size>>>(a,b,c,N);

    // do something with c

    cudaFree(a); cudaFree(b), cudaFree(c);
}
```



CUDA Training

GPGPU Programming

Programming and Execution model for GPUs

João Barbosa
jbarbosa@tacc.utexas.edu
September, 2018

More advanced concepts

Optimization and Advanced concepts

- ▶ Thread Divergence
- ▶ Memory Coalescing
- ▶ Shared Memory
- ▶ Shared Memory Bank Conflicts

Illustrated using a reduction example

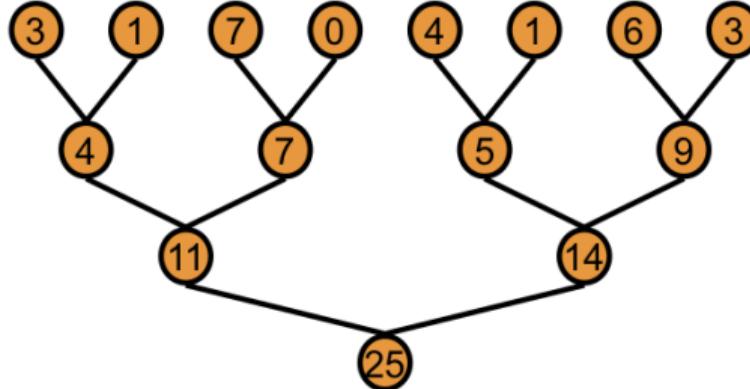
(based on original reduction example material by Mark Harris from NVidia)

Parallel reduction¹

- ▶ Important data parallel primitive
- ▶ Easy to Implement but hard to get right
- ▶ Hits almost all the major bottlenecks, we will implement incrementally

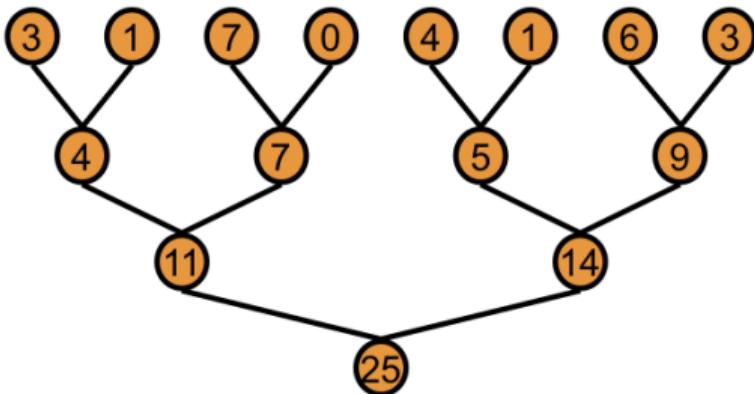


Parallel reduction¹



- ▶ Tree based computation inside each *thread block*
- ▶ Each *thread block* only does a partial reduction (MAX. threads per block 2048)
- ▶ Requires multiple *thread block*
 - to process large values of N
 - to increase occupancy of the GPU (maintain all SMs busy)

Parallel reduction¹



- ▶ Requires multiple *thread block*
- ▶ **Problem:** How do we communicate results between blocks?
 - *thread block* execution is independent
 - only *warps* can be synchronized inside a *thread block*

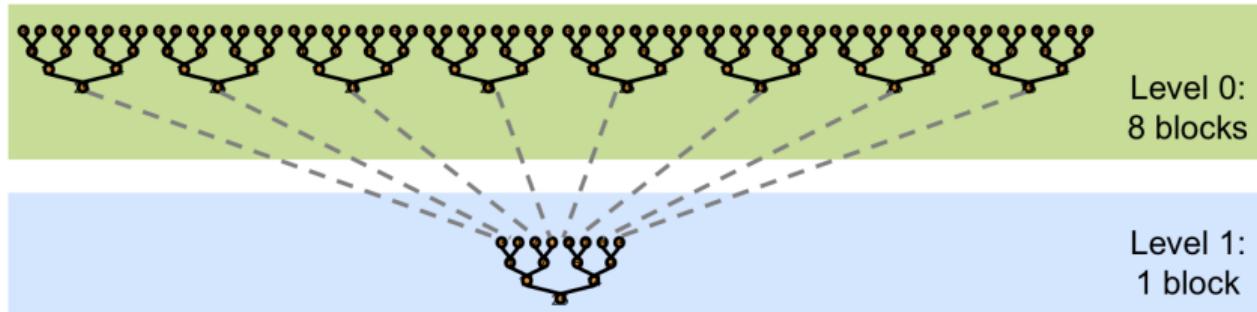
Parallel reduction¹

Problem: How do we communicate results between blocks?

- ▶ Up to the Volta architecture cuda does not have *global synchronize* primitive inside the kernel
 - After Volta it does have but we will see below (caveat: read white paper for details)
- ▶ With *global synchronize* the parallel pattern is easy to implement
 - *thread blocks* compute the partial result
 - once all *thread blocks* reach barrier, continue recursively
- ▶ CUDA (exception after Volta) has no global sync, why?
 - Expensive to build in GPU with high core count
 - Forces programmer to use less *thread 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*
 - kernel launch is cheap (low HW and SE overhead)



Parallel reduction¹



- ▶ **Problem:** How do we communicate results between blocks?
- ▶ **Solution:** decompose into multiple kernels
 - natural *global synchronization*
 - same kernel code at recursively lower values of N

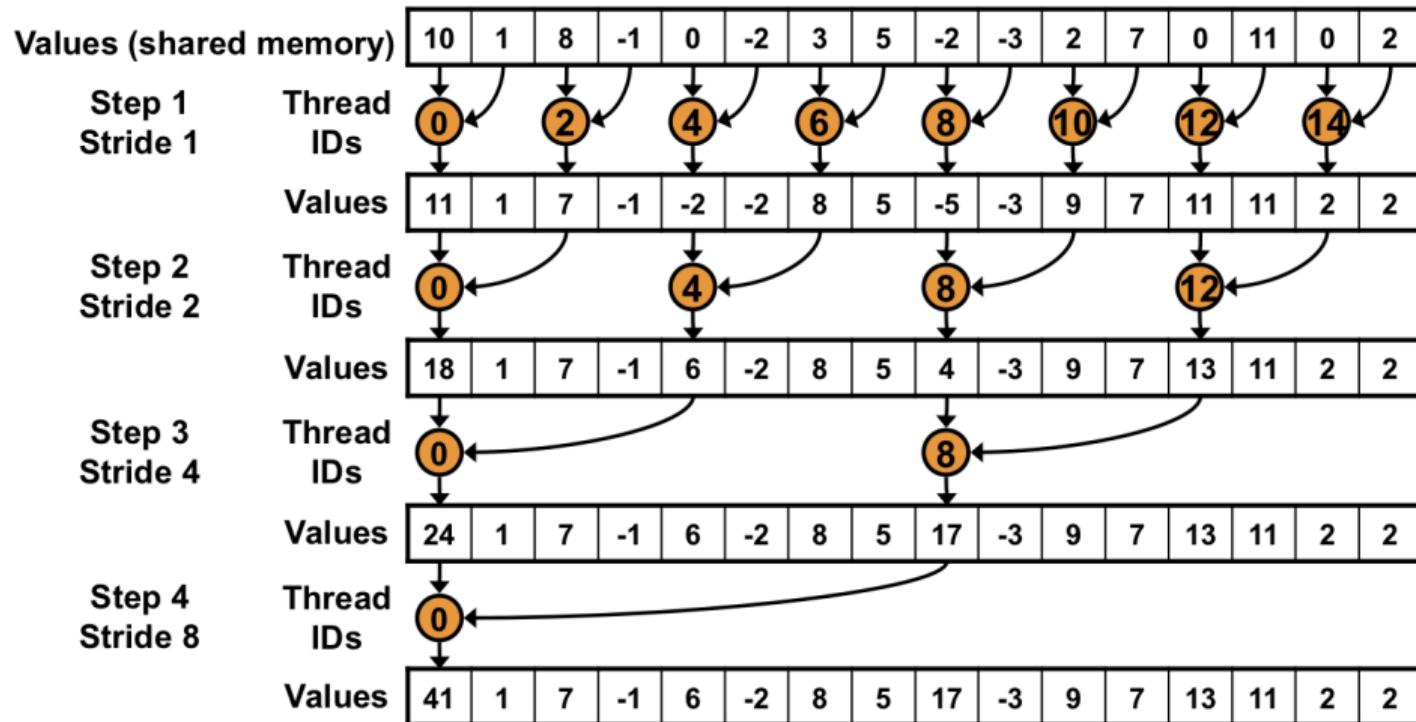
Parallel reduction ¹

Our goals should always be to:

- ▶ Achieve peak GPU performance
 - TFlops : *compute bound kernels*
 - GB/s : *bandwidth bound kernels*
- ▶ Parallel reduction is a *compute bound kernel*
 - Only one arithmetic operation per loaded element (in this case 2)
 - Goal peak bandwidth
- ▶ We are using maverick with a Tesla k40m
 - SP : 4.29 TFlops peak | DP : 1.42 TFlops peak
 - DRAM : 240 GB/s



Parallel reduction¹: Interleaved Addressing



Parallel reduction¹: Interleaved Addressing

```
--global__ void reduce0(int *idata, int *odata) {  
  
    // Shared memory reservation done on kernel call  
    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;  
  
    // Copy data to shared memory  
    sdata[tid] = idata[i];  
  
    // Make sure all threads (wraps) reach this point  
    __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) odata[blockIdx.x] = sdata[0];  
}
```



Parallel reduction¹: Interleaved Addressing

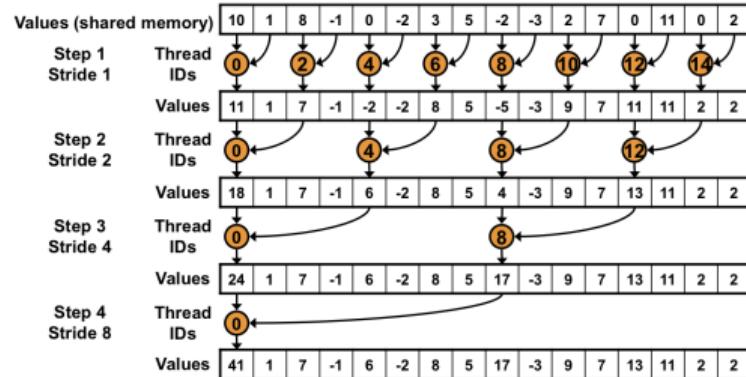
Reduction kernel implementation	Time (2^{22})	Bandwidth (2^{22})	Speedup
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A



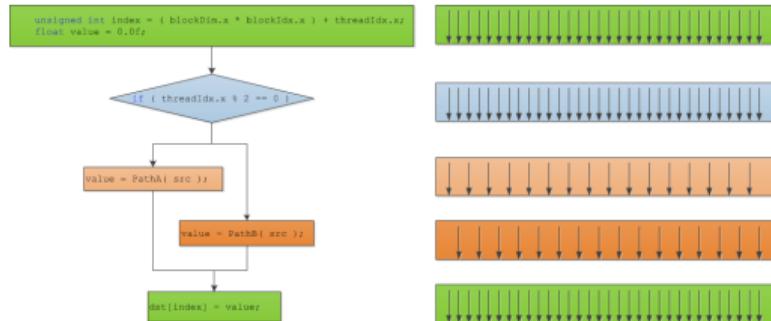
Parallel reduction¹: Interleaved Addressing

```
// 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();
}
```

- ▶ Highly divergent code
- ▶ Half of the threads don't do anything

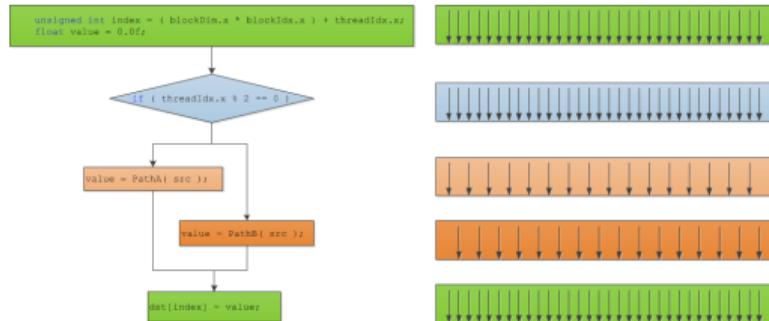


Parallel reduction¹: Thread divergence



What happens if the threads within the warp diverge?

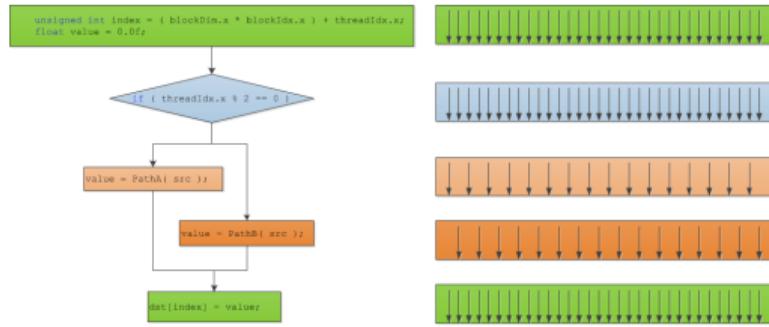
Parallel reduction¹: Thread divergence



What happens if the threads within the warp diverge?

1. Marks as active all threads on Path A
2. Executes warp

Parallel reduction¹: Thread divergence



What happens if the threads within the warp diverge?

1. Marks as active all threads on Path A
2. Executes warp
3. Marks as active all threads on Path B
4. Executes warp

Parallel reduction¹: Interleaved Addressing without divergence

Just replace divergent branch in inner loop:

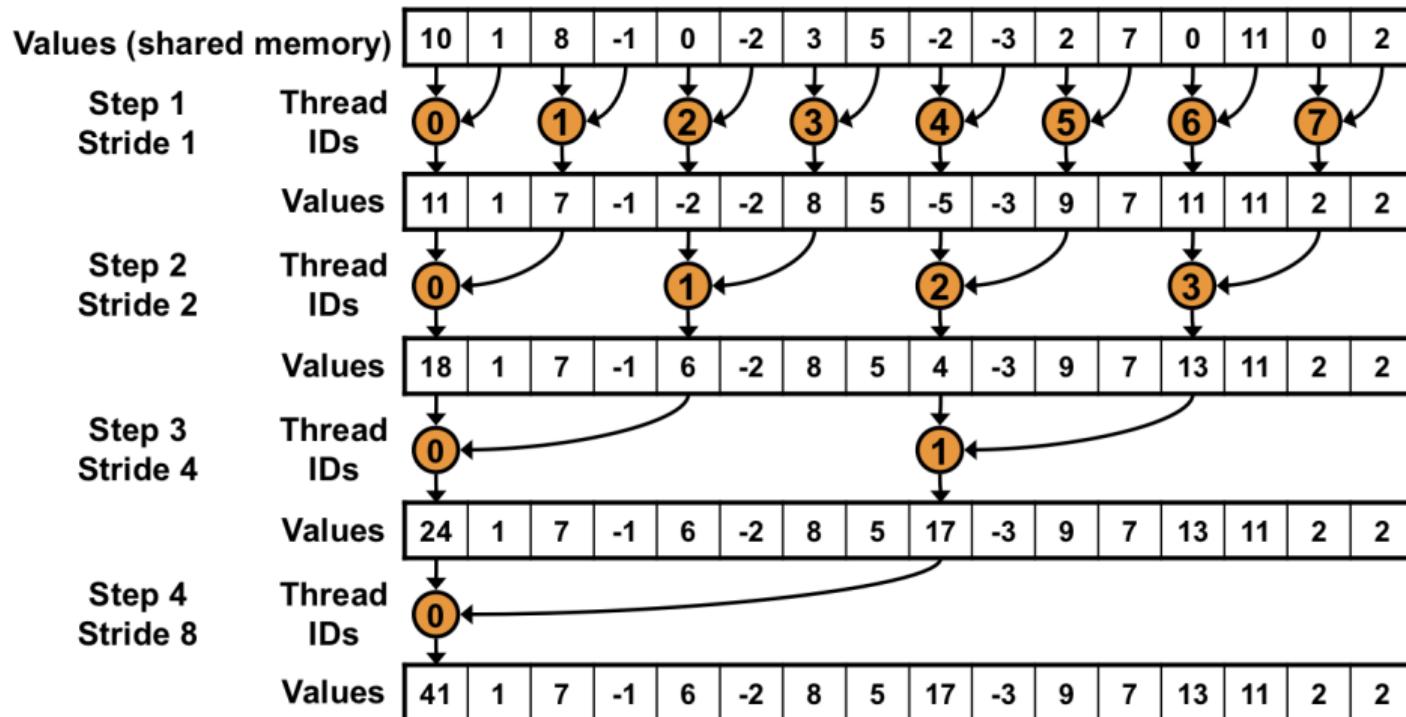
```
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
    if (tid % (2*s) == 0) {
        sdata[tid] += sdata[tid + s];
    }
    __syncthreads();
}
```

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();
}
```



Parallel reduction¹: Interleaved Addressing

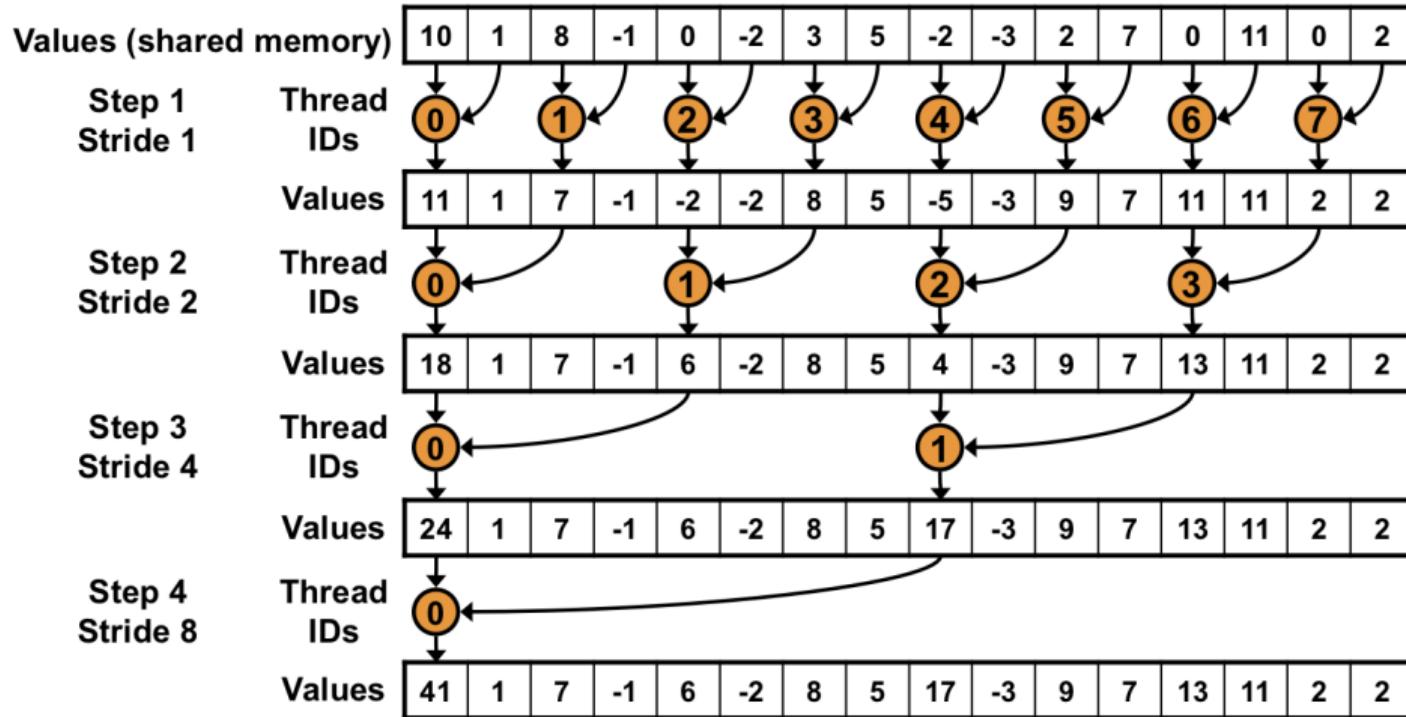


Parallel reduction¹: Interleaved Addressing

Reduction kernel implementation	Time (2^{22})	Bandwidth (2^{22})	Speedup	
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A	N/A
Interleaved Addressing without divergence	0.6215 ms	25.14 GB/s	1.60	1.60

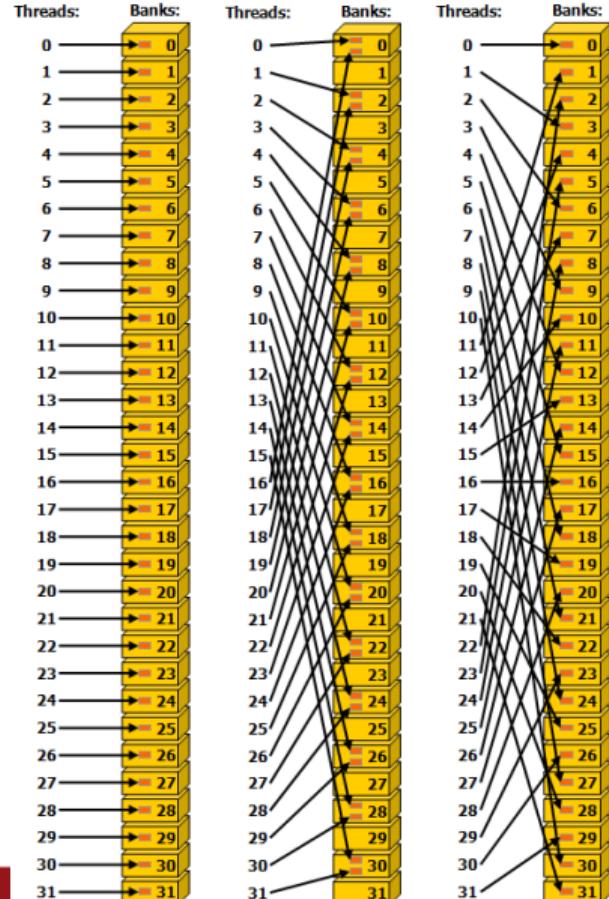


Parallel reduction¹: Interleaved Addressing



Problem: Shared memory bankconflict

Parallel reduction¹: Interleaved Addressing

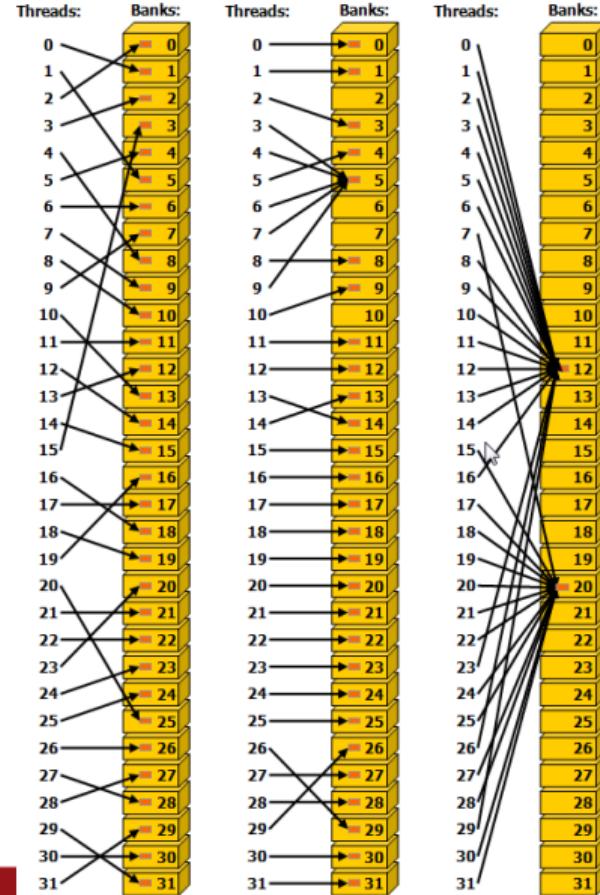


Since compute 3.0

- ▶ Shared memory organized into 32 banks
- ▶ No conflict as long as each thread in a warp writes to a single bank



Parallel reduction¹: Interleaved Addressing



Since compute 3.0

- ▶ Shared memory organized into 32 banks
- ▶ If more than one thread in a warp writes to the same bank

Parallel reduction¹: Sequential Addressing

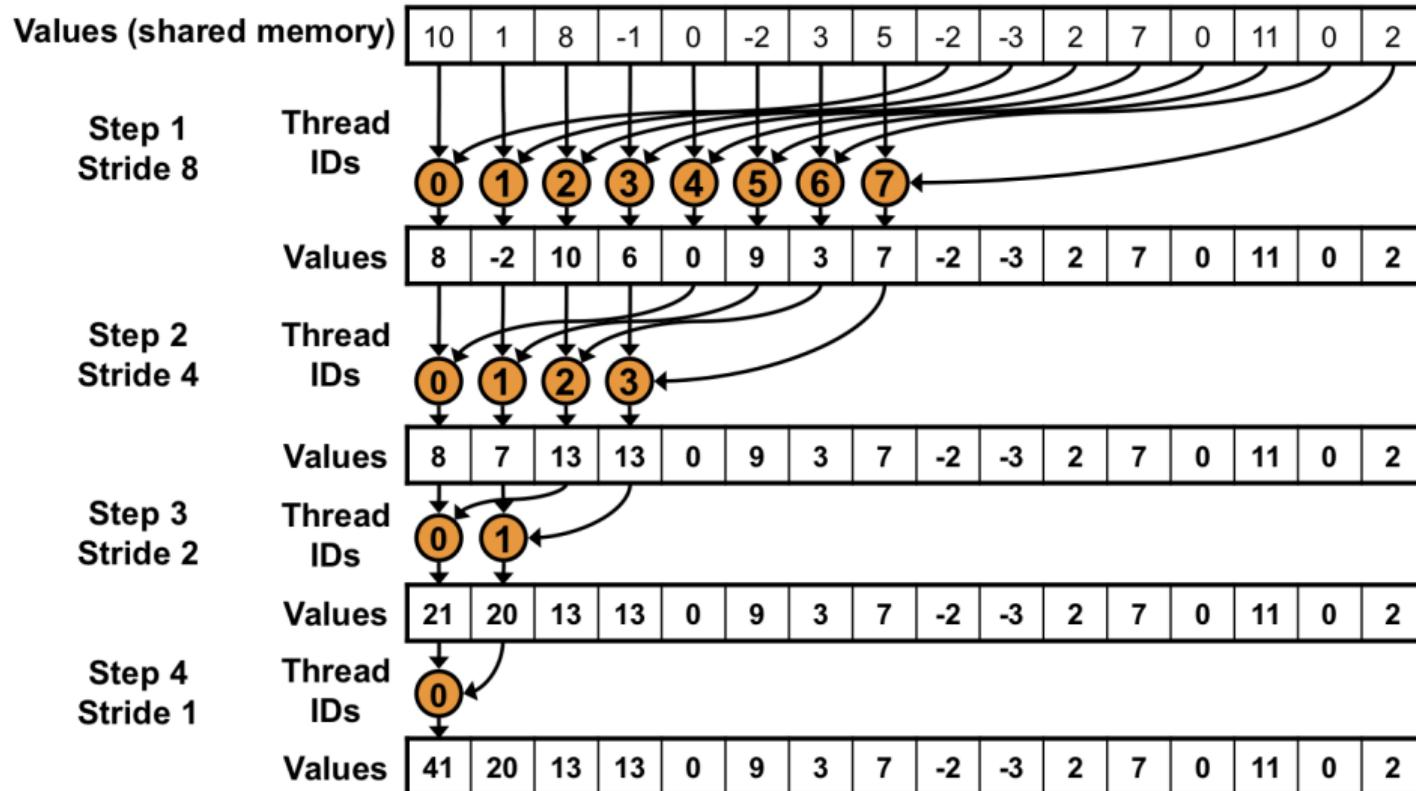
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {  
    int index = 2 * s * tid;  
    if (index < blockDim.x) {  
        sdata[index] += sdata[index + s];  
    }  
    __syncthreads();  
}
```

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();  
}
```



Parallel reduction¹: Sequential Addressing



Parallel reduction¹: Sequential Addressing

Reduction kernel implementation	Time (2^{22})	Bandwidth (2^{22})	Speedup	
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A	N/A
Interleaved Addressing without divergence	0.6215 ms	25.14 GB/s	1.60	1.60
Sequential Addressing	0.2588 ms	60.38 GB/s	1.00	1.61

Parallel reduction¹: Sequential Addressing

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

Problem: Loosing half of the threads in a warp



Parallel reduction¹: First Add During Load

Solution: 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();
```

Reversed loop and threadID-based indexing

```
// 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();
```

Parallel reduction¹: First Add During Load

Reduction kernel implementation	Time (2^{22})	Bandwidth (2^{22})	Speedup	
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A	N/A
Interleaved Addressing without divergence	0.6215 ms	25.14 GB/s	1.60	1.60
Sequential Addressing	0.2588 ms	60.38 GB/s	1.00	1.61
First Add During Load	0.1915 ms	81.59 GB/s	2.39	3.84

Parallel reduction¹: So far

We optimized bandwidth to X GB/s

- ▶ Optimization
 - Removed thread Divergence
 - Removed shared memory bank conflict
- ▶ What are we missing?
 - We know reduction has low arithmetic intensity.
 - How about address arithmetic and loop overhead?
- ▶ **Strategy:** unroll loops



Parallel reduction¹: So far

- ▶ As reduction proceeds, # “active” threads decreases
 - When $s \leq 32$, we have only one warp left
 - Instructions are SIMT synchronous within a warp
- ▶ That means when $s \leq 32$:
 - We don't need to `syncthreads()`
 - We don't need *if*($tid < s$) because it doesn't save any work



Parallel reduction¹: Loop unroll

```
__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];
}
```

Reversed loop and threadID-based indexing

```
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);
```



Parallel reduction¹: Loop Unroll

Reduction kernel implementation	Time (2^{22})	Bandwidth (2^{22})	Speedup	
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A	N/A
Interleaved Addressing without divergence	0.6215 ms	25.14 GB/s	1.60	1.60
Sequential Addressing	0.2588 ms	60.38 GB/s	1.00	1.61
First Add During Load	0.1915 ms	81.59 GB/s	2.39	3.84
Loop Unroll	0.1915 ms	81.59 GB/s	1.35	5.19



Parallel reduction¹: So far

- ▶ If we knew the number of iterations at compile time, we could completely unroll the reduction
 - Set the GPU thread block size 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



Parallel reduction¹: Complete Unroll / Static Unroll

```
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];
}

template <unsigned int blockSize> __global__ void reduce5(int *idata, int *odata) {
    ...
    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);
    ...
}
```



Parallel reduction¹: Complete Unroll / Static Unroll

```
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;
}
```



Parallel reduction ¹: Complete Unroll / Static Unroll

Reduction kernel implementation	Time (2^{22})	Bandwidth (2^{22})	Speedup	
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A	N/A
Interleaved Addressing without divergence	0.6215 ms	25.14 GB/s	1.60	1.60
Sequential Addressing	0.2588 ms	60.38 GB/s	1.00	1.61
First Add During Load	0.1915 ms	81.59 GB/s	2.39	3.84
Loop Unroll	0.1915 ms	81.59 GB/s	1.35	5.19
Static Unroll	0.1753 ms	89.15 GB/s	1.09	5.67



Parallel reduction¹: Implement

```
template <unsigned int blockSize>
__device__ void warpReduce(volatile int *sdata, unsigned 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];
}

template <unsigned int blockSize>
__global__ void reduce(int *idata, int *odata, unsigned int n) {
    extern __shared__ int sdata[];
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*(blockSize*2) + tid;
    unsigned int gridSize = blockSize*2*gridDim.x;
    sdata[tid] = 0;
    while (i < n) { sdata[tid] += idata[i] + idata[i+blockSize]; i += gridSize; }
    __syncthreads();
    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(sdata, tid);
    if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}
```