SPMD - CUDA GPU Programming

Joao Barbosa

Week 2, March 2021

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

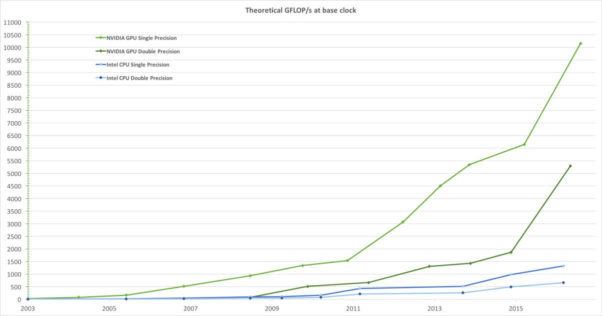


Figure: FLOPS (source NVidia)

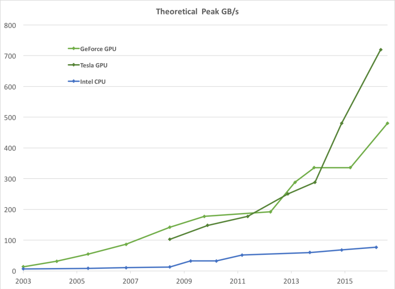


Figure: Memory bandwidth (source NVidia)

Brief history of the GPU

NVidia G70 - Pre-GPGPU Programable

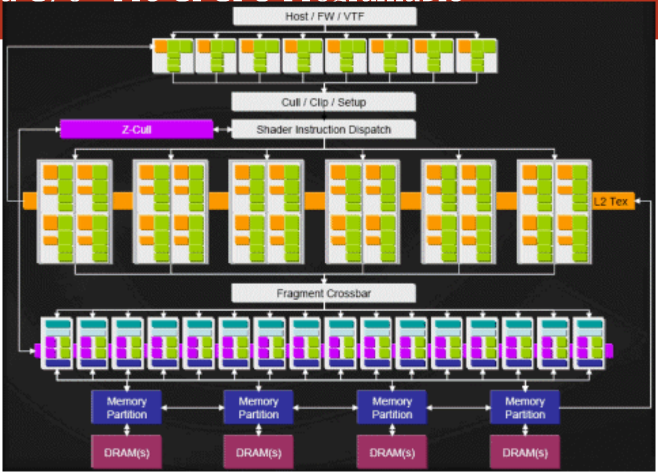


Figure: NVidia G70 Architectural diagram (source NVidia)

NVidia G80 - GPGPU Programable

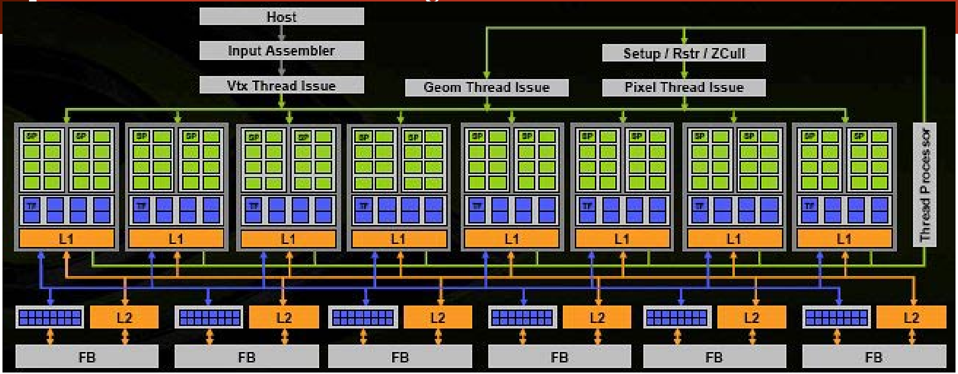


Figure: NVidia G80 Architectural diagram (source NVidia)

- November 2006 - NVidia launched the G80 Architecture
- June 2007 - CUDA - Compute Unified Device Architecture (NVidia)

NVidia G80 - GPGPU Programable

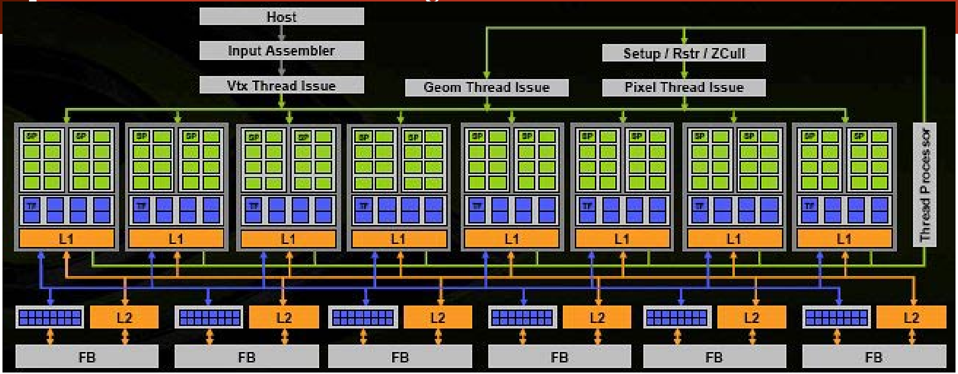


Figure: NVidia G80 Architectural diagram (source NVidia)

- August 2009 - OpenCL 1.0 - Open Computing Language (Apple)
- June 2010 - OpenCL 1.1 - Open Computing Language (Khronos Group)

Latest NVidia - Volta GV100 GPU

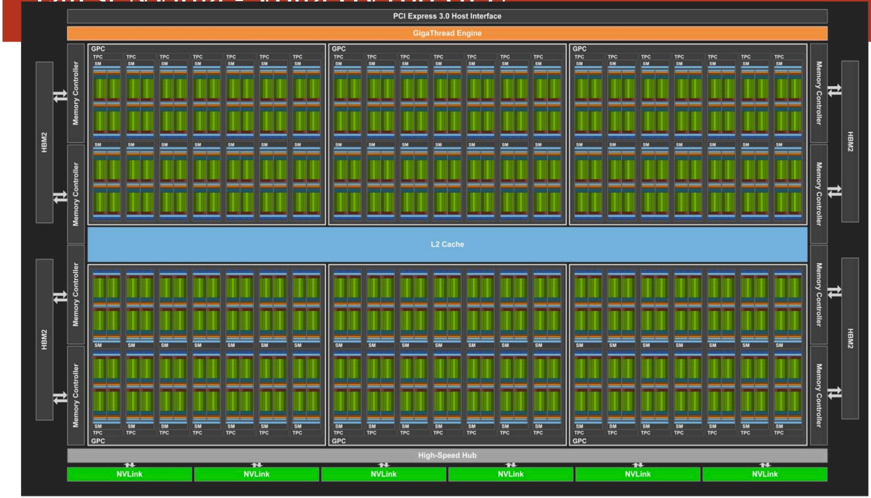


Figure: NVidia Volta (GV100) Architectural diagram [1]

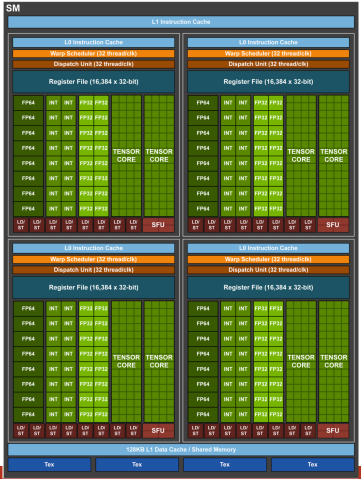
NVidia - Volta GV100 GPU Stream Multiprocessor

Stream Multiprocessor

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

  $$D = A.B + C$$

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


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

Difference between a CPU and GPU

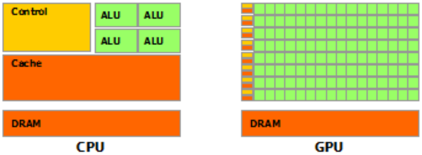


Figure: Abstract CPU and GPU Architecture diagrams (source NVidia)

- **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 as its own, called Direct Compute, but it is similar to CUDA and OpenCL
- OpenCL and Direct Computed 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

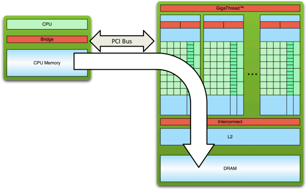
- Architecture and execution model
  - Introduced in NVIDIA in 2007
  - Get highest possible execution performance requires understanding of hardware architecture
- Programming model
  - Small set of extensions to C / C++
  - Enables GPUs to execute programs written in C / C++
  - Within C programs,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

CUDA - Programming and Execution model

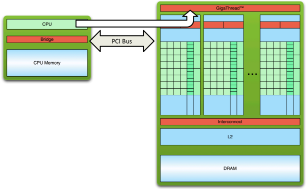
host The CPU and its memory (host memory)

device The GPU and its memory (device memory)

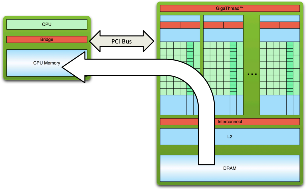
kernel The computational payload called from the host to be executed on the device



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

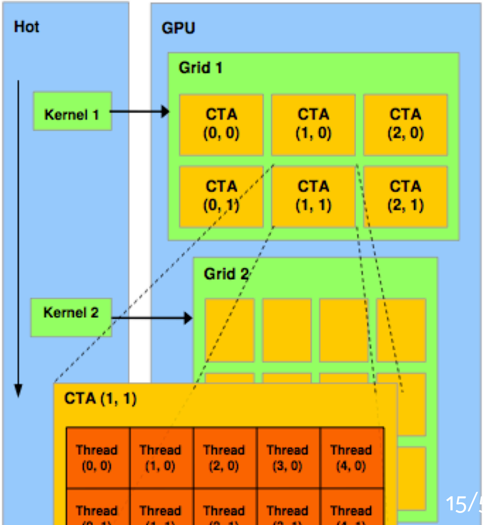
- Unique ID

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



Thread/Block Hierarchy transparency

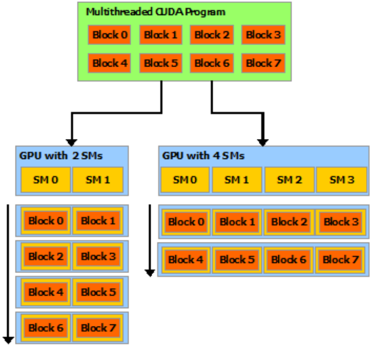


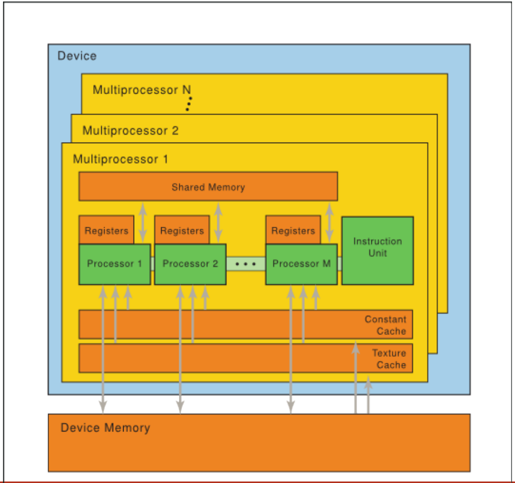
Figure: source: NVidia

- 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

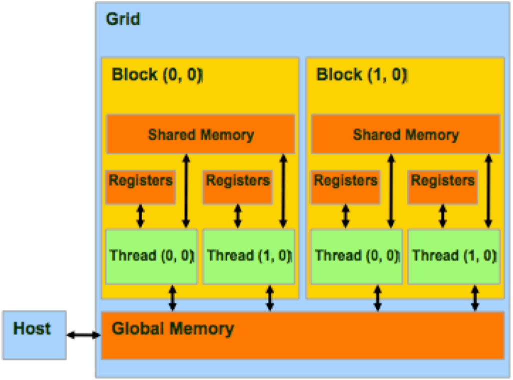


Figure: Memory Model (source NVidia)

- 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

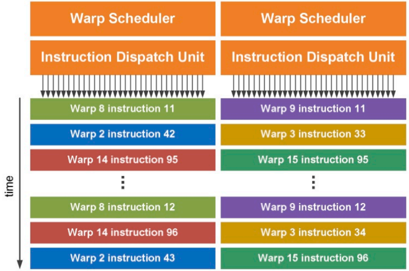


Figure: Warp Execution Model (source NVidia)

CPU - Vector Add

```c
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);
}
```

More advanced concepts

optimization and advanced concepts

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

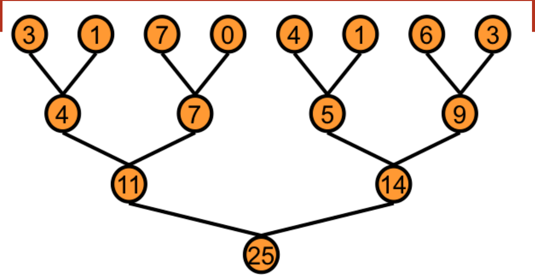
Ilustrated using an reduction example
(based of original reduction example material by Mark Harris from NVidia)

Parallel reduction [2]

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

[2] based of original reduction example material by Mark Harris from NVidia

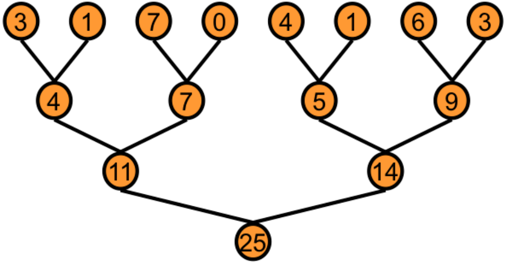
Parallel reduction [3]



- 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)

[3] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [4]



- 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*

[4] based on original reduction example material by Mark Harris from NVidia

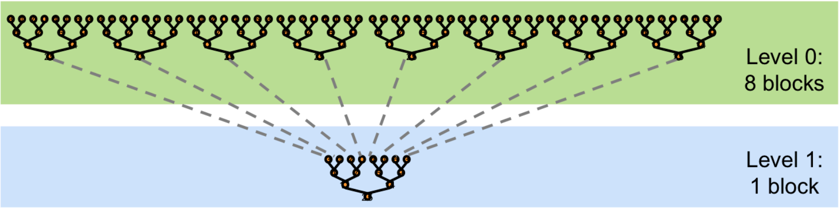
Parallel reduction [5]

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 bellow (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 *thred 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)

[5] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [6]



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

[6] based on original reduction example material by Mark Harris from NVidia

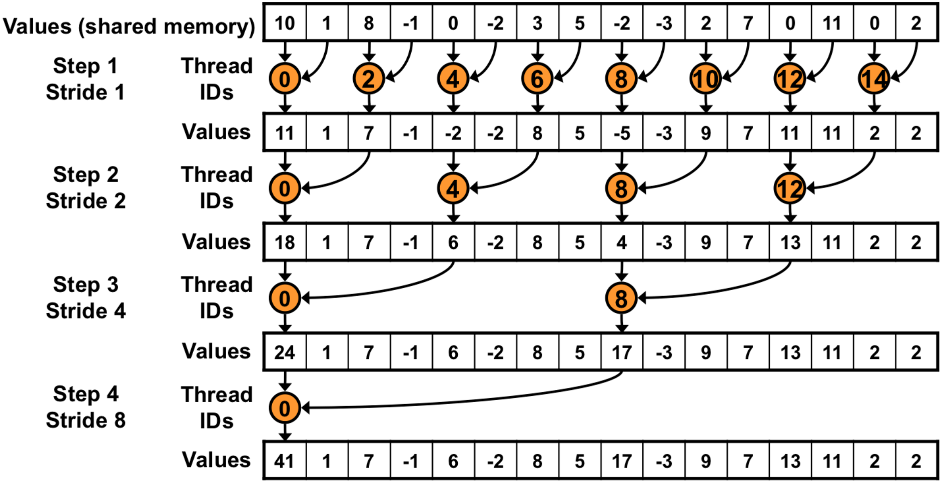
Parallel reduction [7]

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 bancdwith
- We are using maverick with a Tesla k40m
  - SP : 4.29 TFlops peak | DP : 1.42 TFlops peak
  - DRAM : 240 GB/s

[7] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [8]: Interleaved Addressing



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

[8] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [9]: 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];
}
```

[9] based on original reduction example material by Mark Harris from Nvidia

Parallel reduction [10]: Interleaved Addressing

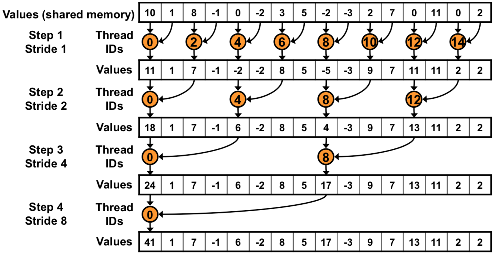
Reduction kernel implementation	**Time** (2^{22})	**Bandwith** (2^{22})	**Speedup**	
Interleaved Addressing with divergence	0.9941 ms	15.72 GB/s	N/A	N/A

[10] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [11]: 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();
}
```

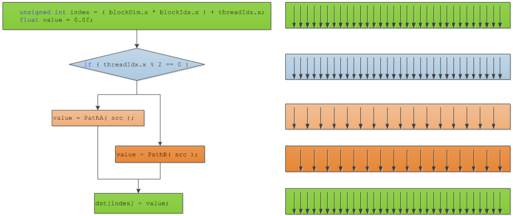
- Highly divergente code
- Half of the threads don't do anything



	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

[11] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [12]: Thread divergence



```
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;
```

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

```
value = PathA( src );
```

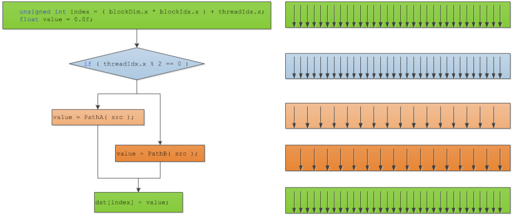
```
value = PathB( src );
```

```
dst[index] = value;
```

What happens if the threads within the warp diverge?

[12] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [12]: Thread divergence



```
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;
```

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

```
value = PathA( src );
```

```
value = PathB( src );
```

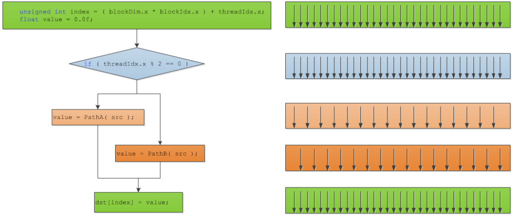
```
dst[index] = value;
```

What happens if the threads within the warp diverge?

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

[12] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [12]: Thread divergence



```
unsigned int index = ( blockDim.x * blockIdx.x ) + threadIdx.x;
float value = 0.0f;
```

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

```
value = PathA( src );
```

```
value = PathB( src );
```

```
dst[index] = value;
```

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

[12] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [13]: Interleaved Addressing without divergence

Just replace divergent branch in inner loop:

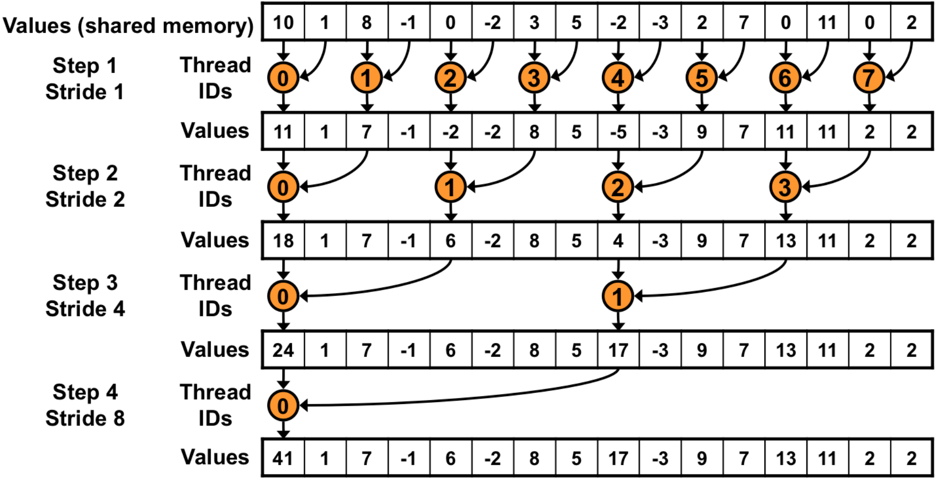
```
// do reduction in shared mem
for(unsigned int s=1; s < blockDim.x; s *= 2) {
  if (tid % (2*s) == 0) {
    sdata[tid] += sdata[tid + s];
  }
  __syncthreads();
}
```

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();
}
```

[13] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [14]: Interleaved Addressing



	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		1		2		3		4		5		6		7	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				1				2				3			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								1							
	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

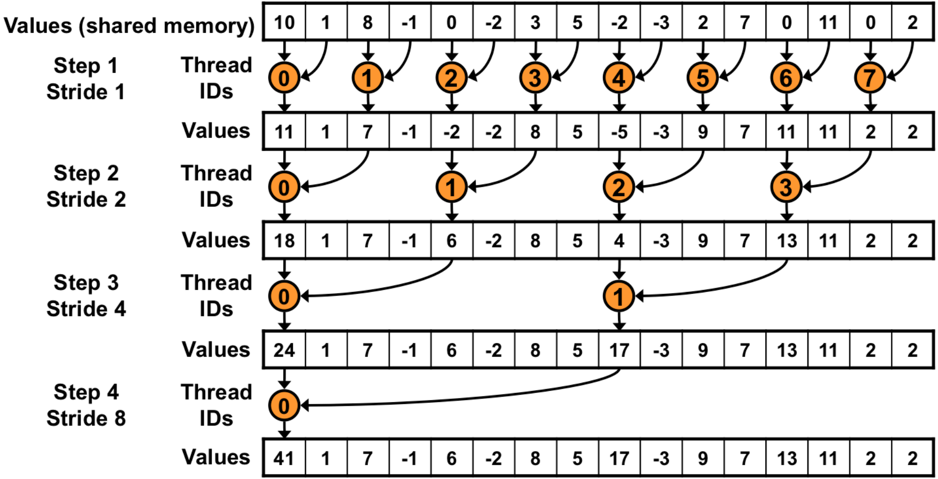
[14] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [15]: Interleaved Addressing

Reduction kernel implementation	**Time** (2^{22})	**Bandwith** (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

[15] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [16]: Interleaved Addressing

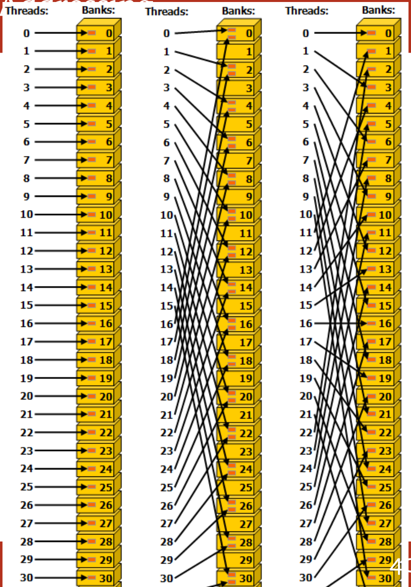


Values (shared memory)		10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Step 1 Stride 1	Thread IDs	0		1		2		3		4		5		6		7	
	Values	11	1	7	-1	-2	-2	8	5	-5	-3	9	7	11	11	2	2
Step 2 Stride 2	Thread IDs	0				1				2				3			
	Values	18	1	7	-1	6	-2	8	5	4	-3	9	7	13	11	2	2
Step 3 Stride 4	Thread IDs	0								1							
	Values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

Problem: Shared memory bankconflict

[16] based on original reduction example material by Mark Harris from Nvidia

Parallel reduction [17]: Interleaved Addressing



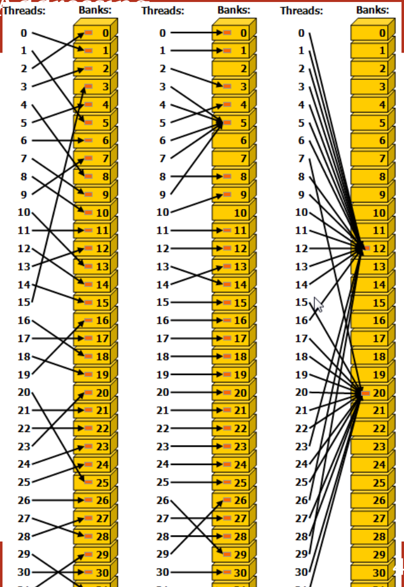
Since compute 3.0

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

Parallel reduction [18]: 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 reduction [19]: Sequencial Addressing

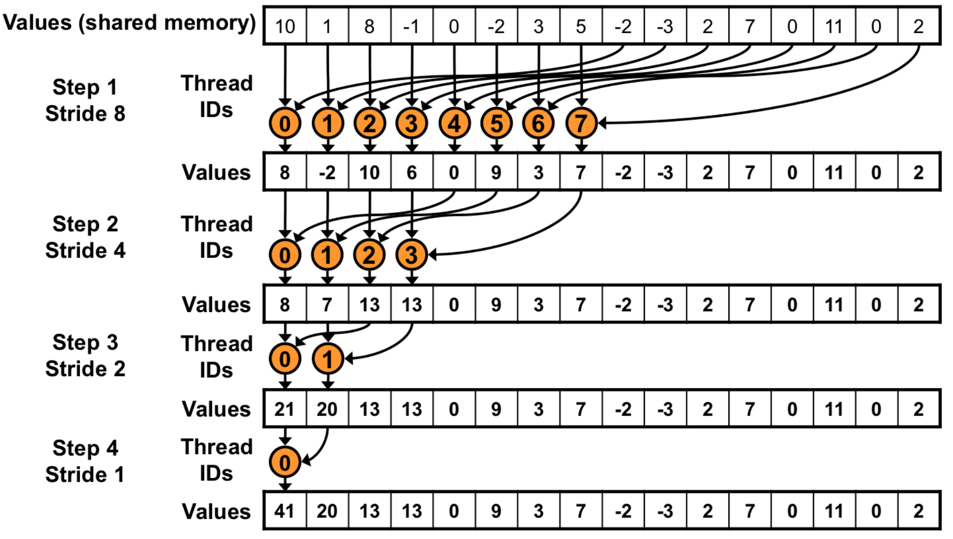
```
for (unsigned int s=1; s < blockDim.x; s *= 2) {
  int index = 2 * s * tid;
  if (index < blockDim.x) {
    sdata[index] += sdata[index + s];
  }
  __syncthreads();
}
```

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();
}
```

[19] based on original reduction example material by Mark Harris from NVidia

Parallel reduction[20]: Sequential Addressing



Values (shared memory)	10	1	8	-1	0	-2	3	5	-2	-3	2	7	0	11	0	2
Values	8	-2	10	6	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	8	7	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	21	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2
Values	41	20	13	13	0	9	3	7	-2	-3	2	7	0	11	0	2

Problem: Shared memory bankconflict

Parallel reduction [21]: Sequential Addressing

Reduction kernel implementation	**Time** (2^{22})	**Bandwith** (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

[21] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [22]: Sequencial 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

[22] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [23]: 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();
```

[23] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [24]: First Add During Load

Reduction kernel implementation	**Time** (2^{22})	**Bandwith** (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

[24] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [25]: So far

We optimized bandwith 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

[25] based of original reduction example material by Mark Harris from NVidia

Parallel reduction [26]: 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

[26] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [27]: 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);
```

[27] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [28]: Loop Unroll

Reduction kernel implementation	Time (2^{22})	Bandwith (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

[28] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [29]: 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

[29] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [30]: 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);
  (...)
}
```

[30] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [31]: 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;
}
```

[31] based on original reduction example material by Mark Harris from NVidia

Parallel reduction[32]: Complete Unroll / Static Unroll

Reduction kernel implementation	Time (2^{22})	Bandwith (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

[32] based on original reduction example material by Mark Harris from NVidia

Parallel reduction [33]: Complete Unroll / Static Unroll

[33] based of original reduction example material by Mark Harris from NVidia

Thank you