

# CUDA Profiling Tutorial

May 24<sup>th</sup>, 2019

## Before we get started: Setting up NVVP with GCP VMs

- If you are on Mac OS or Linux: congrats! No extra work required!
  - `gcloud compute ssh reduction -ssh-flag="-Y"`
- Windows: Easiest way is to **enable password authentication**
  - Log into your VM with `gcloud compute ssh reduction`
  - Become root: `sudo -s`
  - Open `/etc/ssh/sshd_config`, change `PasswordAuthentication` to `yes`
  - `systemctl reload ssh`
  - `passwd <username>`
  - Get the VM external IP from Google Cloud Console
  - Test it using an X11-capable ssh client (e.g. MobaXTerm)
- Run `nvvp &`. If it works, you're ready!
- Of course, if you have NVVP locally, feel free to start a remote session

# Recall Lecture 11: Parallel Reduction Group Activity

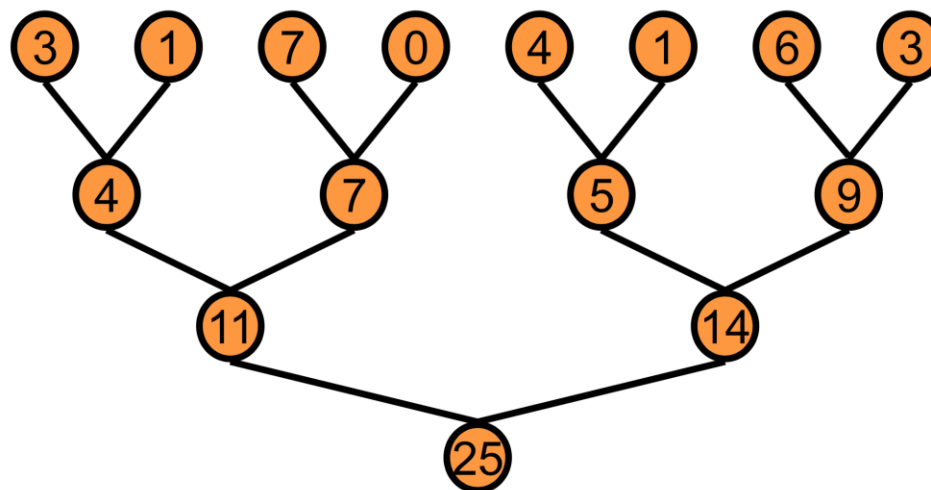


## What we're doing today

- Going over 3 reduction kernel implementations
- 1 using NVVP remotely
- 1 using nvprof and analyzing locally
- 1 on your own, in class
- Reduction code

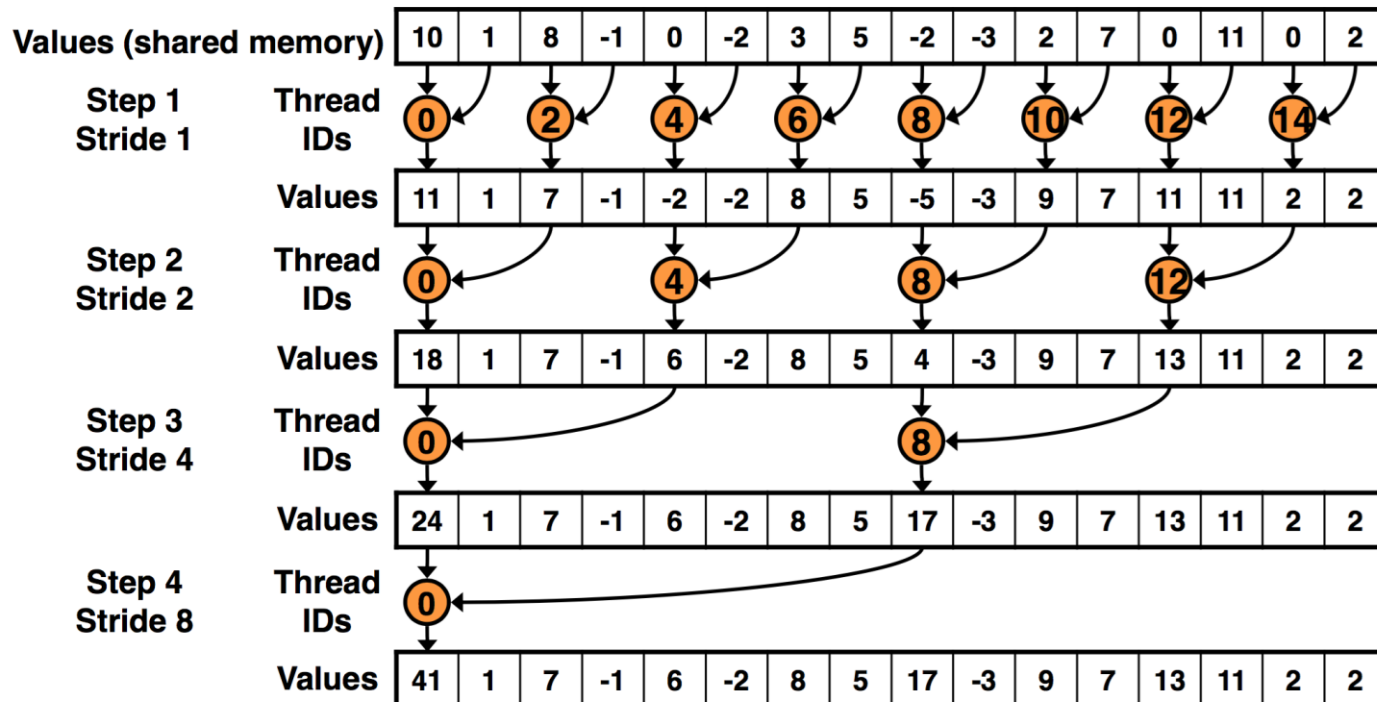
## Reduction Kernel 0

- Goal: efficiently sum up all elements in an array
- Idea: divide and conquer by splitting work into many blocks
- Each thread block does a reduction
- Merge results by invoking kernel multiple times with different strides



# Reduction Kernel 0 Algorithm

- For each block, load data into shared memory
- Use shared memory to tree reduce within the block
- Store result back into global memory
- Repeat kernel until complete



# Reduction Kernel 0 Code

```
template <class T>
__global__ void reduce0(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    sdata[tid] = (i < n) ? g_idata[i] : 0;

    __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];

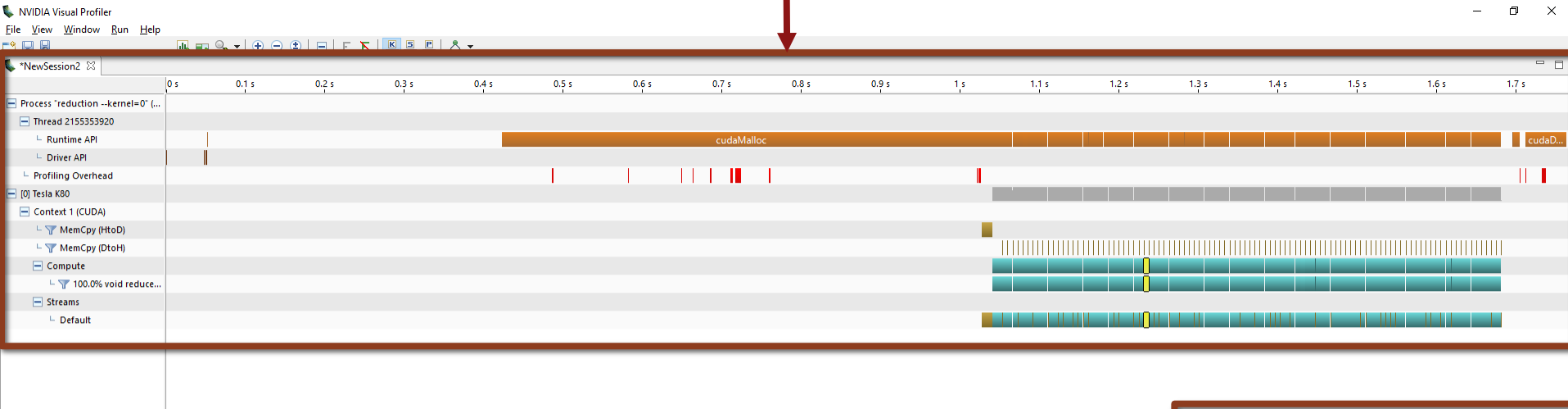
        __syncthreads();
    }

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

Let's see what NVVP says...



## Timeline



## Kernel Analysis

**1. CUDA Application Analysis**

The guided analysis system walks you through the various analysis stages to help you understand the optimization opportunities in your application. Once you become familiar with the optimization process, you can explore the individual analysis stages in an unguided mode. When optimizing your application it is important to fully utilize the compute and data movement capabilities of the GPU. To do this you should look at your application's overall GPU usage as well as the performance of individual kernels.

- Examine GPU Usage**  
Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.
- Examine Individual Kernels**  
Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.
- Delete Existing Analysis Information**  
If the application has changed since the last analysis then the existing analysis information may be stale and should be deleted before continuing.
- Switch to unguided analysis**

## Timeline Detailed View

**Properties**

**void reduce0<int>(int\*, int\*, unsigned int)**

Queued	n/a
Submitted	n/a
Start	1.23096 s (1,230,963,074...)
End	1.2373 s (1,237,304,501 ...)
Duration	6.34143 ms (6,341,427 ns)
Stream	Default
Grid Size	[ 32768,1,1 ]
Block Size	[ 512,1,1 ]
Registers/Thread	8
Shared Memory/Block	2 KiB
Launch Type	Normal
Occupancy	
Achieved	96.6%
Theoretical	100%
Shared Memory Configuration	
Shared Memory Executed	8 KiB
Shared Memory Bank Size	4 B

# Kernel Guided Analysis







- Let's see what the profiler has to say
- Click “Examine GPU Usage”

Reduction, Throughput = 12.1446 GB/s, Time = 0.00553 s, Size = 16777216 Elements, NumDevsUsed = 1, BlockSize = 512

GPU result = 2139353471

CPU result = 2139353471

TEST PASSED

Results	
 <b>Low Memcpy/Kernel Overlap</b> [ 0 ns / 13.26288 ms = 0% ]	<a href="#">More...</a>
The percentage of time when memcpys are being performed in parallel with kernel is low.	
 <b>Low Kernel Concurrency</b> [ 0 ns / 634.4838 ms = 0% ]	<a href="#">More...</a>
The percentage of time when two kernels are being executed in parallel is low.	
 <b>Low Memcpy Throughput</b> [ 1.942 MB/s avg, for memcpys accounting for 1.6% of all memcpy time ]	<a href="#">More...</a>
The memory copies are not fully using the available host to device bandwidth.	
 <b>Low Memcpy Overlap</b> [ 0 ns / 206.015 μs = 0% ]	<a href="#">More...</a>
The percentage of time when two memory copies are being performed in parallel is low.	
 <b>Low Compute Utilization</b> [ 634.4838 ms / 1.76324 s = 36% ]	<a href="#">More...</a>
The multiprocessors of one or more GPUs are mostly idle.	
 <b>Compute Utilization</b>	
The device timeline shows an estimate of the amount of the total compute capacity being used by the kernels executing on the device.	

## What do the issues mean?

### Low Memcpy/Kernel Overlap

- Means we're not copying data as we're running kernels
- Not a problem in our case

### Low Kernel Concurrency

- Means we aren't executing kernels in parallel
- Not a problem in our case

### Low Memcpy Overlap

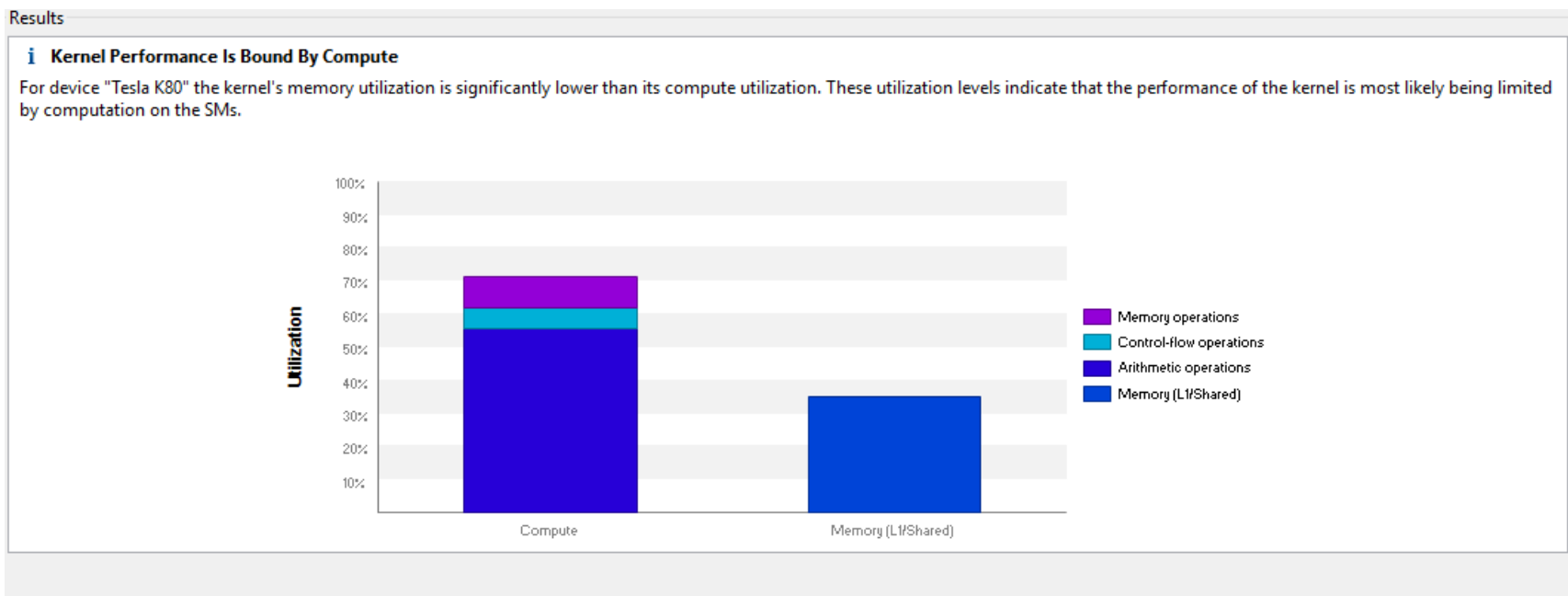
- Means we're not effectively copying data to/from GPU
- Not a problem in our case

### Low Compute Utilization

- Means the SMs aren't doing much work
- **This is a problem!**

# What does the profiler say?

- Hit “Examine Kernels” and select the top `reduce0` kernel
- Click “Perform Kernel Analysis”
- Spending a lot of time in compute!



## Tell us more, oh mighty profiler

- Hit “Perform Compute Analysis” since that’s our bottleneck
- Branch divergence is slowing us down

### Low Warp Execution Efficiency

Warp execution efficiency is the average percentage of active threads in each executed warp. Increasing warp execution efficiency will increase utilization of the GPU's compute resources. The warp execution efficiency for these kernels is 99.9% if predicated instructions are not taken into account. The kernel's not predicated off warp execution efficiency of 70.9% is less than 100% due to divergent branches and predicated instructions.

*Optimization: Reduce the amount of intra-warp divergence and predication in the kernel.*

[More...](#)

### Divergent Branches

Compute resource are used most efficiently when all threads in a warp have the same branching behavior. When this does not occur the branch is said to be divergent. Divergent branches lower warp execution efficiency which leads to inefficient use of the GPU's compute resources.

*Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence.*

[More...](#)

▼ Line / File | NA

NA	Divergence = 6.2% [ 32768 divergent executions out of 524288 total executions ]
----	---

## Let's look at our kernel again...

```
template <class T>
__global__ void reduce0(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    // load shared mem
    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x * blockDim.x + threadIdx.x;

    sdata[tid] = (i < n) ? g_idata[i] : 0;

    __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];

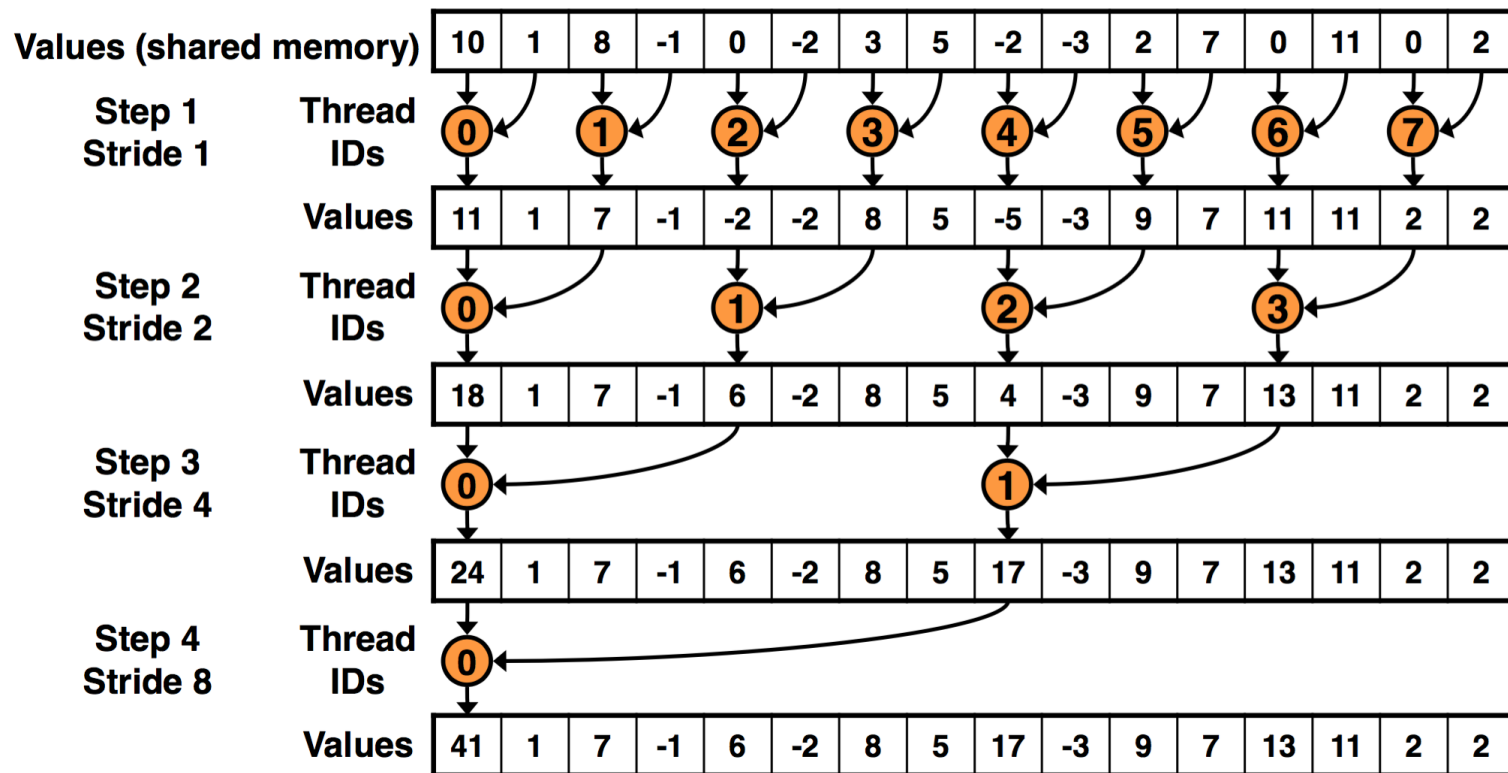
        __syncthreads();
    }

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

- Divergence is here
- Also, modulo is slow!

# Reduction Kernel 1


- Let's try to have only a few warps doing most of the work.
- This means less thread divergence.



# Remove divergence with strided index

```
// 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();
}

// do reduction in shared mem
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();
}
```



- If  $s = 1$ , threads 0, 2, 4, ... run
- If  $s = 4$ , threads 0, 8, 16 ... run
- Warp divergence
- Only consecutive threads run
- No divergence



## Reduction Kernel 1 Profiling with nvprof

- **Warning: Make sure you are calling the correct nvprof binary!**
- Use `/usr/local/cuda-10.0/bin/nvprof`
- Run:  

```
nvprof --analysis-metrics -o reduction.prof \  
./reduction --kernel=1
```
- `--analysis-metrics` collects everything needed for NVVP Guided Analysis

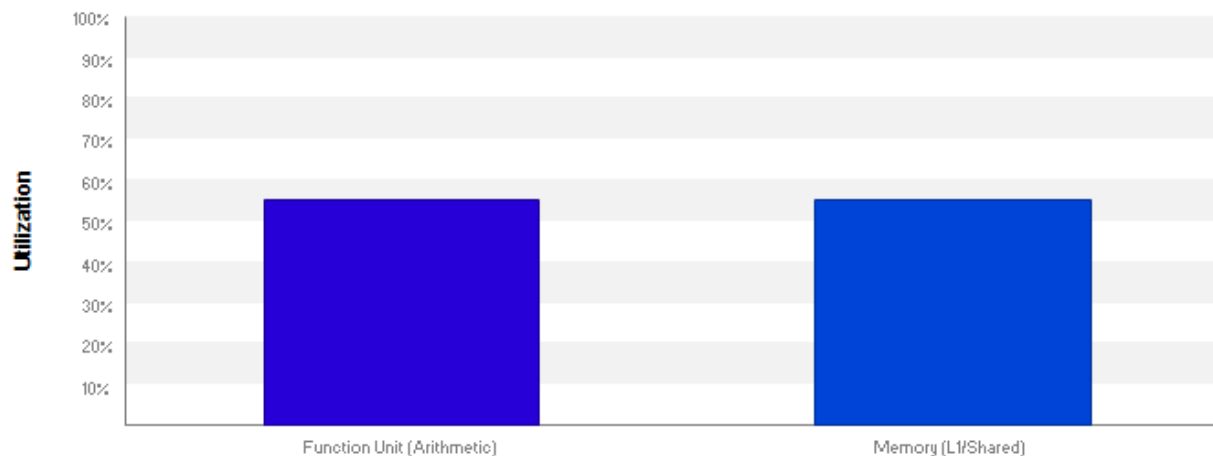
# Reduction Kernel 1 Profiling Results

- We seem to be bound by latency: things are taking too long!
- Let's run "Latency Analysis"

	Throughput GB/s
Kernel 0	16
Kernel 1	20

## i Kernel Performance Is Bound By Instruction And Memory Latency

This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla K80". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



# Reduction Kernel 1 Latency Analysis

## ⚠️ Instruction Latencies May Be Limiting Performance

Instruction stall reasons indicate the condition that prevents warps from executing on any given cycle. The following chart shows the break-down of stalls reasons averaged over the entire execution of the kernel. The kernel has good theoretical and achieved occupancy indicating that there are likely sufficient warps executing on each SM. Since occupancy is not an issue it is likely that performance is limited by the instruction stall reasons described below.

Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.

Synchronization - The warp is blocked at a `__syncthreads()` call.

Instruction Fetch - The next assembly instruction has not yet been fetched.

Memory Dependency - A load/store cannot be made because the required resources are not available or are fully utilized, or too many requests of a given type are outstanding. Data request stalls can potentially be reduced by optimizing memory alignment and access patterns.

Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.

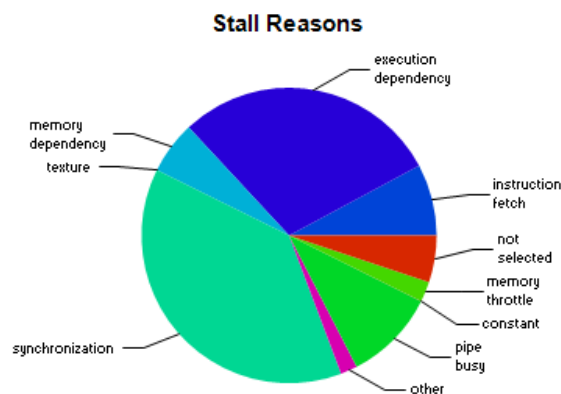
Pipeline Busy - The compute resource(s) required by the instruction is not yet available.

Texture - The texture sub-system is fully utilized or has too many outstanding requests.

Not Selected - Warp was ready to issue, but some other warp issued instead. You may be able to sacrifice occupancy without impacting latency hiding and doing so may help improve cache hit rates.

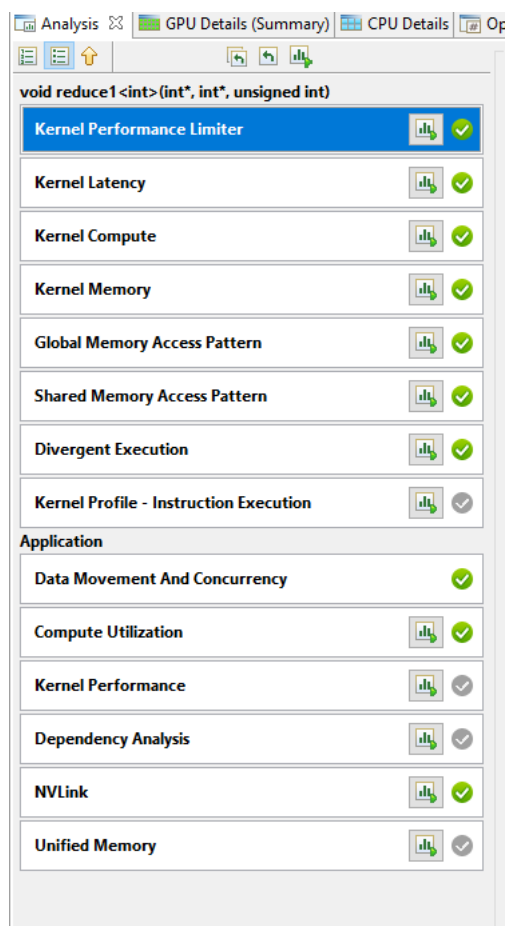
Constant - A constant load is blocked due to a miss in the constants cache.

*Optimization: Resolve the primary stall issue; synchronization.*



# Digging into Execution Dependency Stalls

- We're at the end of guided analysis
- We need more information: switch to **unguided analysis**



Different analyses based on  
kernel or entire application

# Let's check our shared memory access pattern

- Turns out, we have bad shared memory accesses
- Check for **bank conflicts**

## ⚠ Shared Memory Alignment and Access Pattern

Memory bandwidth is used most efficiently when each shared memory load and store has proper alignment and access pattern.

*Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access.*

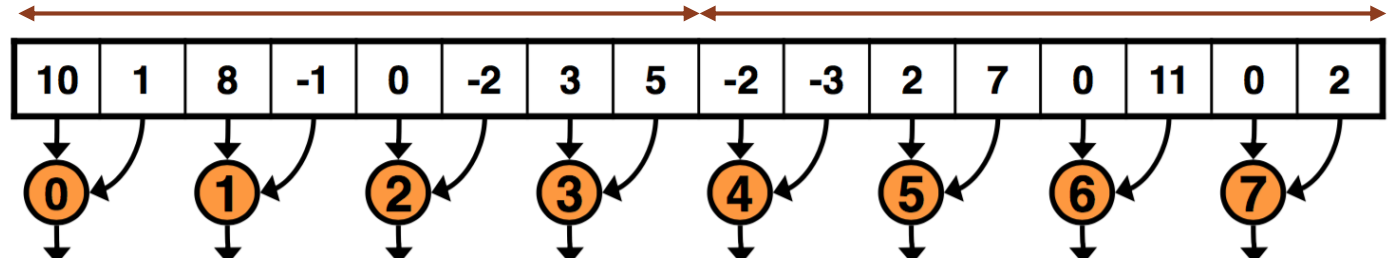
[More...](#)

Line / File	NA	
NA	Shared Load Transactions/Access = 2.8, Ideal Transactions/Access = 1 [ 1802240 transactions for 655360 total executions ]	
NA	Shared Store Transactions/Access = 2.8, Ideal Transactions/Access = 1 [ 1802240 transactions for 655360 total executions ]	
NA	Shared Load Transactions/Access = 2.8, Ideal Transactions/Access = 1 [ 1802240 transactions for 655360 total executions ]	

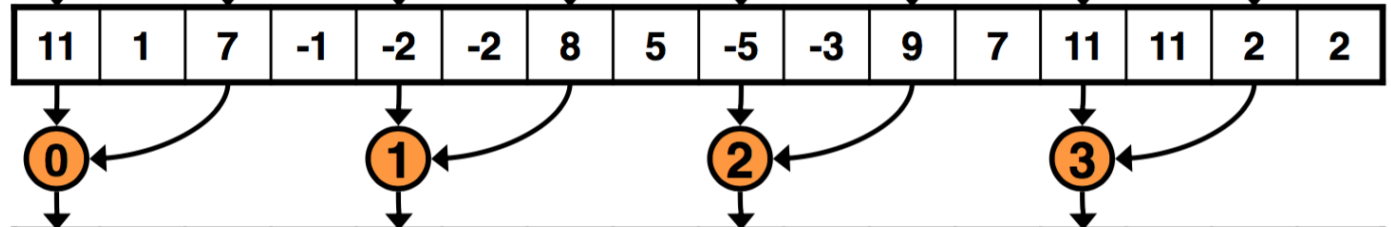
No. of banks = 8

No. of banks = 8

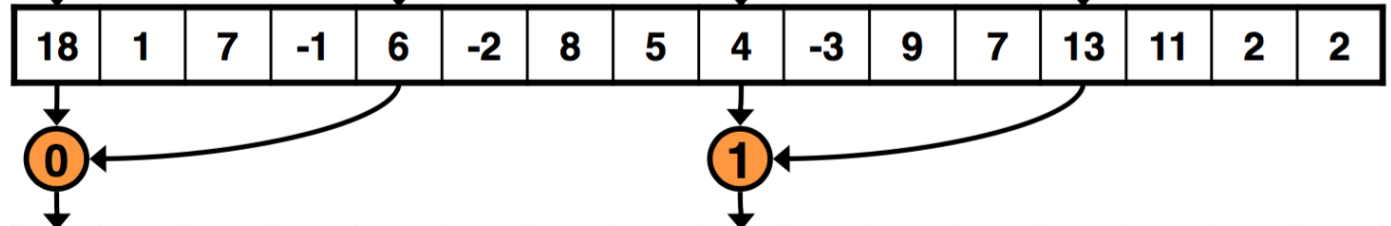
2 way conflict



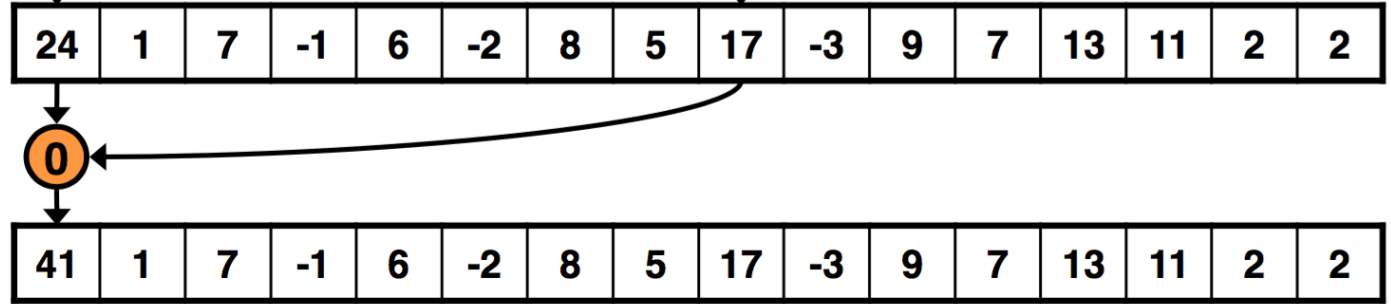
4 way conflict



8 way conflict

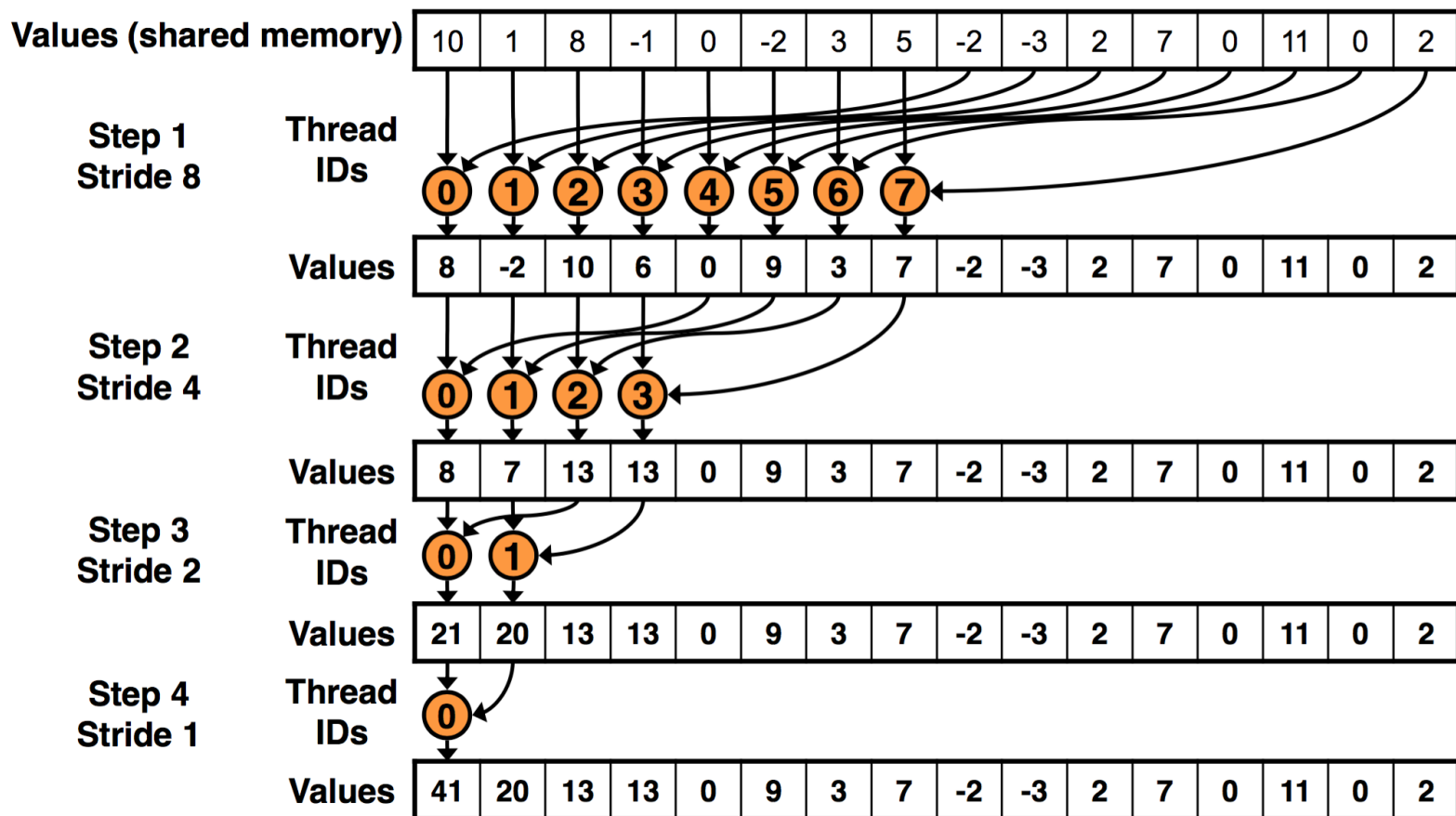


8 way conflict



## Reduction Algorithm, Kernel 2

- Remove bank conflicts through sequential accesses



# Reduction Algorithm, Kernel 2 Implementation

```
// do reduction in shared mem
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();
}
```

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

    __syncthreads();
}
```

**Note different loop  
bounds & addition**

- 1<sup>st</sup> kernel call: we go 1/2 of block size to get next operand
- 2<sup>nd</sup> kernel call: we go 1/4 of block size to get next operand
- And so on

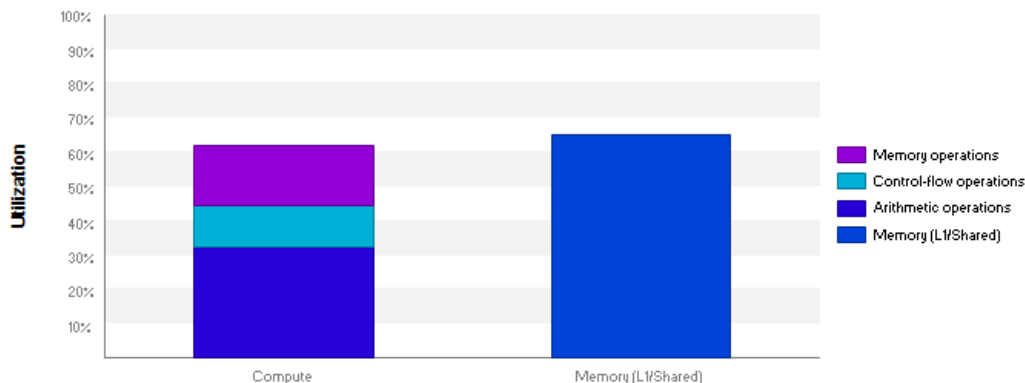


Your turn: Profile Kernel 2!

# Reduction Kernel 2, Results

## i Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla K80" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L1/Shared memory.



## ⚠ GPU Utilization Is Limited By Memory Bandwidth

The following table shows the memory bandwidth used by this kernel for the various types of memory on the device. The table also shows the utilization of each memory type relative to the maximum throughput supported by the memory. The results show that the kernel's performance is potentially limited by the bandwidth available from one or more of the memories on the device.

*Optimization: Try the following optimizations for the memory with high bandwidth utilization.*

*L1/Shared Memory - If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieved 2x throughput. Resolve alignment and access pattern issues for global loads and stores.*

*L2 Cache - Align and block kernel data to maximize L2 cache efficiency.*

*Texture Cache - Reallocate texture cache data to shared or global memory.*

*Device Memory - Resolve alignment and access pattern issues for global loads and stores.*

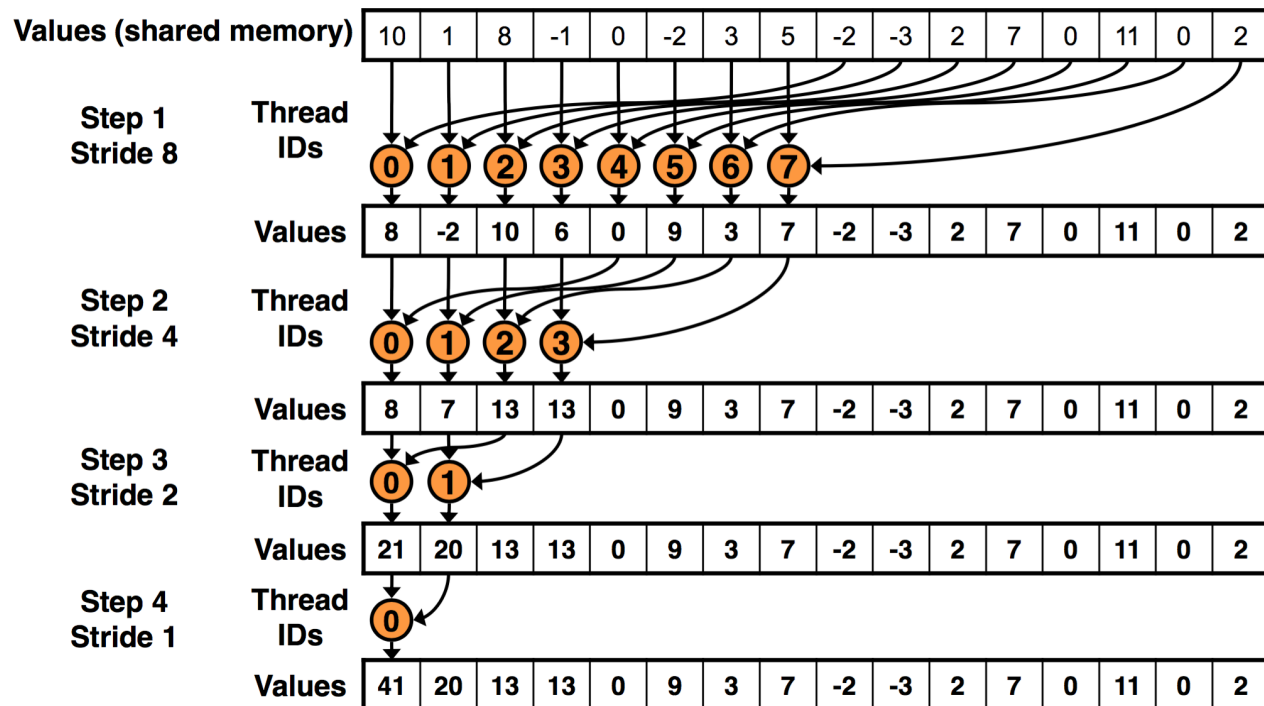
*System Memory (via PCIe) - Make sure performance critical data is placed in device or shared memory.*

[More...](#)

	Transactions	Bandwidth	Utilization
<b>L1/Shared Memory</b>			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	9469952	762.122 GB/s	
Shared Stores	5243469	421.983 GB/s	
Global Loads	524288	21.097 GB/s	
Global Stores	32768	329.637 MB/s	
Atomic	0	0 B/s	
<b>L1/Shared Total</b>	<b>15270477</b>	<b>1,205.531 GB/s</b>	
			<div> <div></div> <div>Idle</div> <div>Low</div> <div>Medium</div> <div>High</div> <div>Max</div> </div>

## Reduction Kernel 2, Results

- We're waiting for memory transactions
- Can we do more work?
- Half of threads are idle: wasteful



# Thank you – any questions?

See Lecture 11 for full optimization cycle