CUDA Profiling Tutorial

March 2nd, 2020

Profiling using the gpu1-project VM

- gpu1-project VM script

```
gcloud compute instances start gpu1-project
sudo update-alternatives --config java
```

- Mac OS or Linux:

```
gcloud compute ssh gpu1-project --ssh-flag="-Y"
```

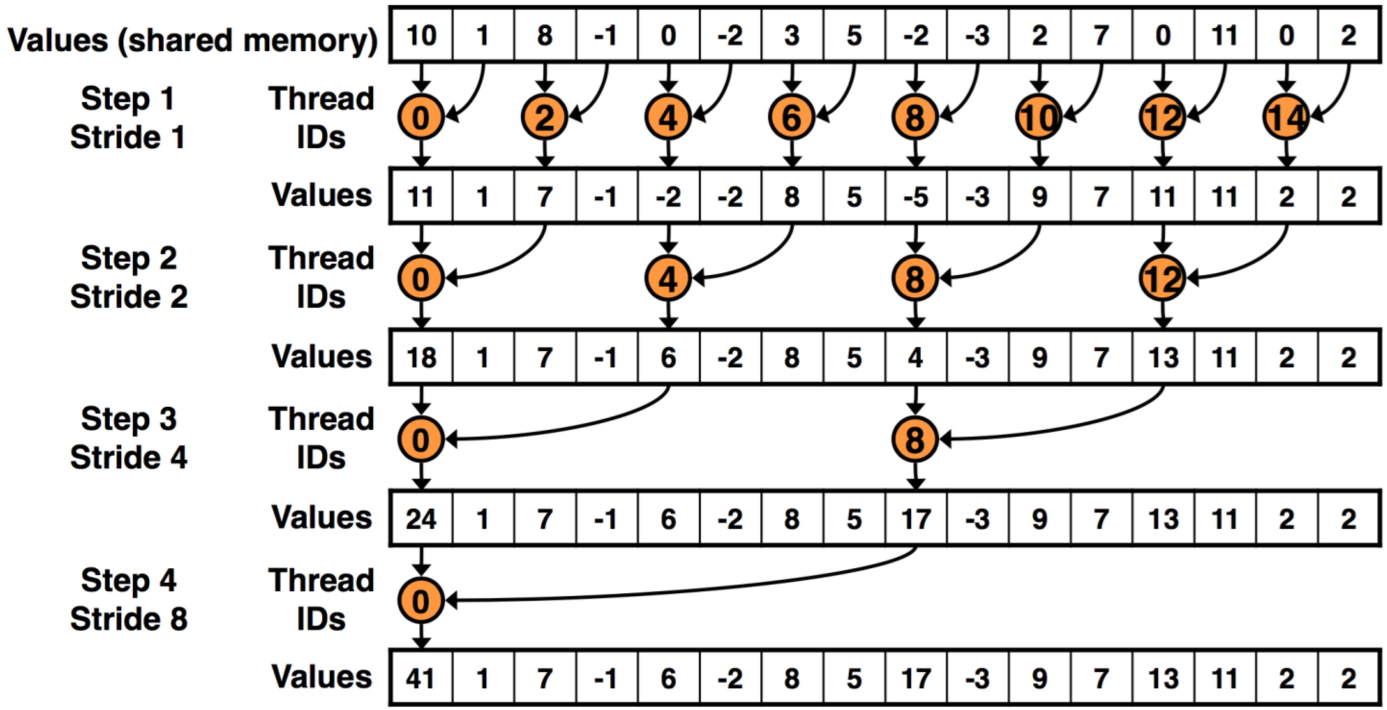
- Windows: use nvprof and NVVP on `rice.stanford.edu`

What we're doing today

- Reduction code

- Kernel 0 with NVVP
- Kernel 1 with nvprof

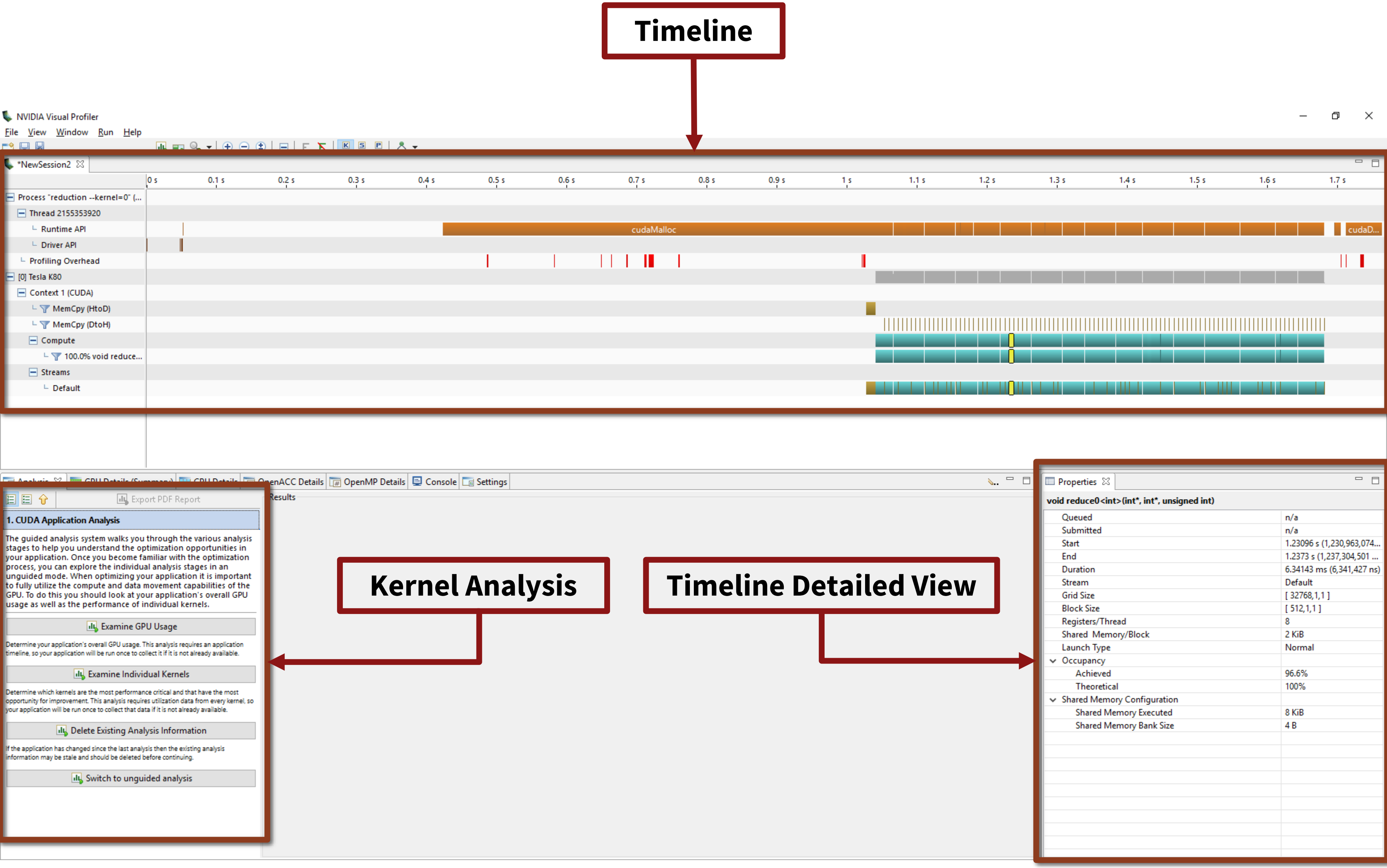
Reduction Kernel 0 Algorithm



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
Timeline Detailed View
NVIDIA Visual Profiler
File View Window Run Help
*NewSession2
Process "reduction --kernel=0" (...
Thread 2155353920
Runtime API
Driver API
Profiling Overhead
[0] Tesla K80
Context 1 (CUDA)
MemCpy (HtoD)
MemCpy (DtoH)
Compute
100.0% void reduce...
Streams
Default
cudaMalloc
Export PDF Report
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
Examine Individual Kernels
Delete Existing Analysis Information
Switch to unguided analysis
OpenMP Details
Console
Settings
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 = 10.7144 GB/s, Time = 0.00626 s, Size = 16777216 Elements, NumDevsUsed = 1, BlockSize = 512

GPU result = 2139353471
CPU result = 2139353471

TEST PASSED
```

Low Kernel Concurrency [0 ns / 682.91971 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Compute Utilization [682.91971 ms / 458.34311 s = 0.1%]
The multiprocessors of one or more GPUs are mostly idle. More...

Compute Utilization
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 Kernel Concurrency

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

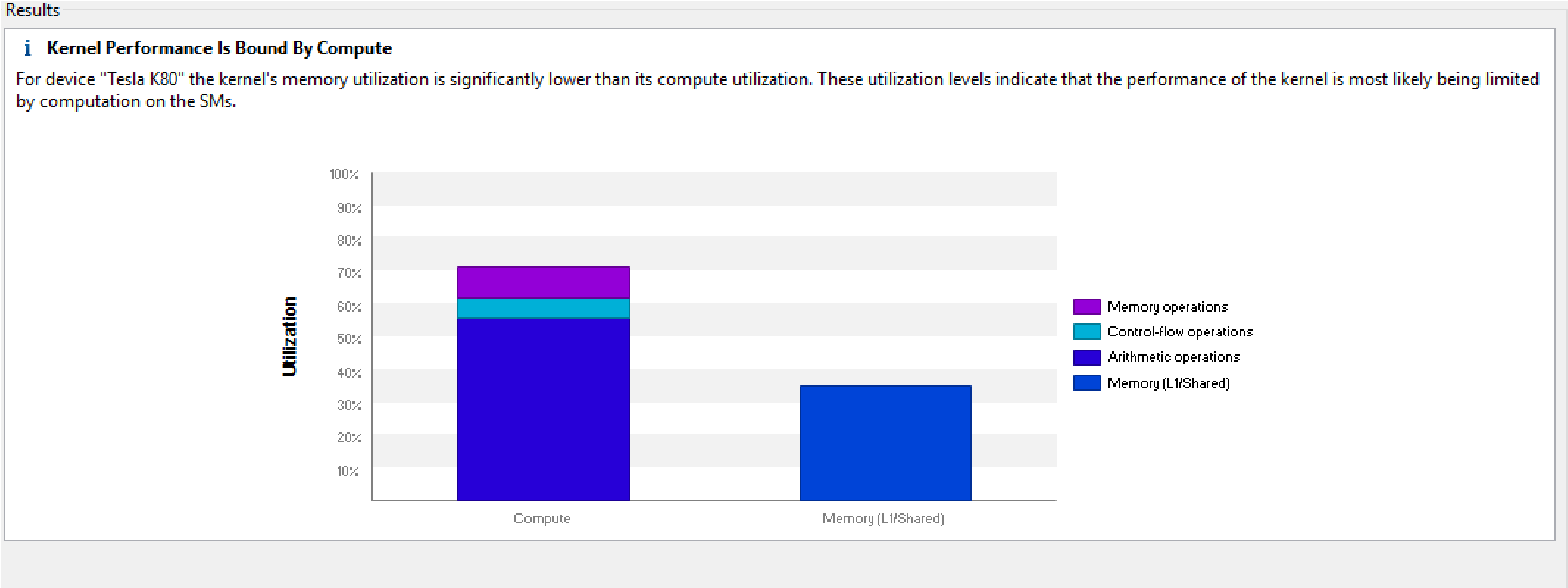
Low Compute Utilization

- 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

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 72.9% 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.

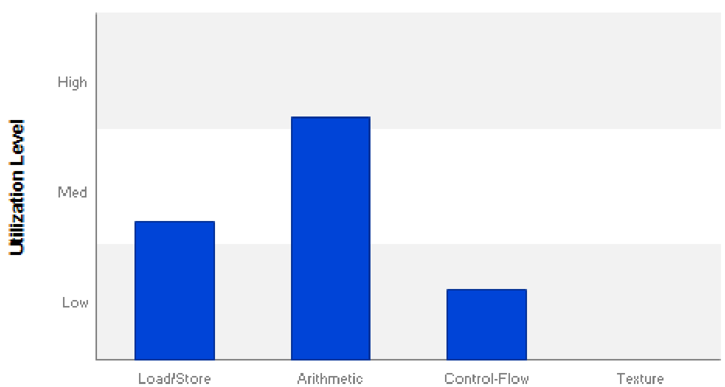
Optimization: Select each entry below to open the source code to a divergent branch within the kernel. For each branch reduce the amount of intra-warp divergence. More...

Line / File	NA
NA	Divergence = 6.2% [32768 divergent executions out of 524288 total executions]

Function Unit Utilization

Different types of instructions are executed on different function units within each SM. Performance can be limited if a function unit is over-used by the instructions executed by the kernel. The following results show that the kernel's performance is not limited by overuse of any function unit.

Load/Store - Load and store instructions for local, shared, global, constant, etc. memory.
Arithmetic - All arithmetic instructions including integer and floating-point add and multiply, logical and binary operations, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.
Texture - Texture operations.



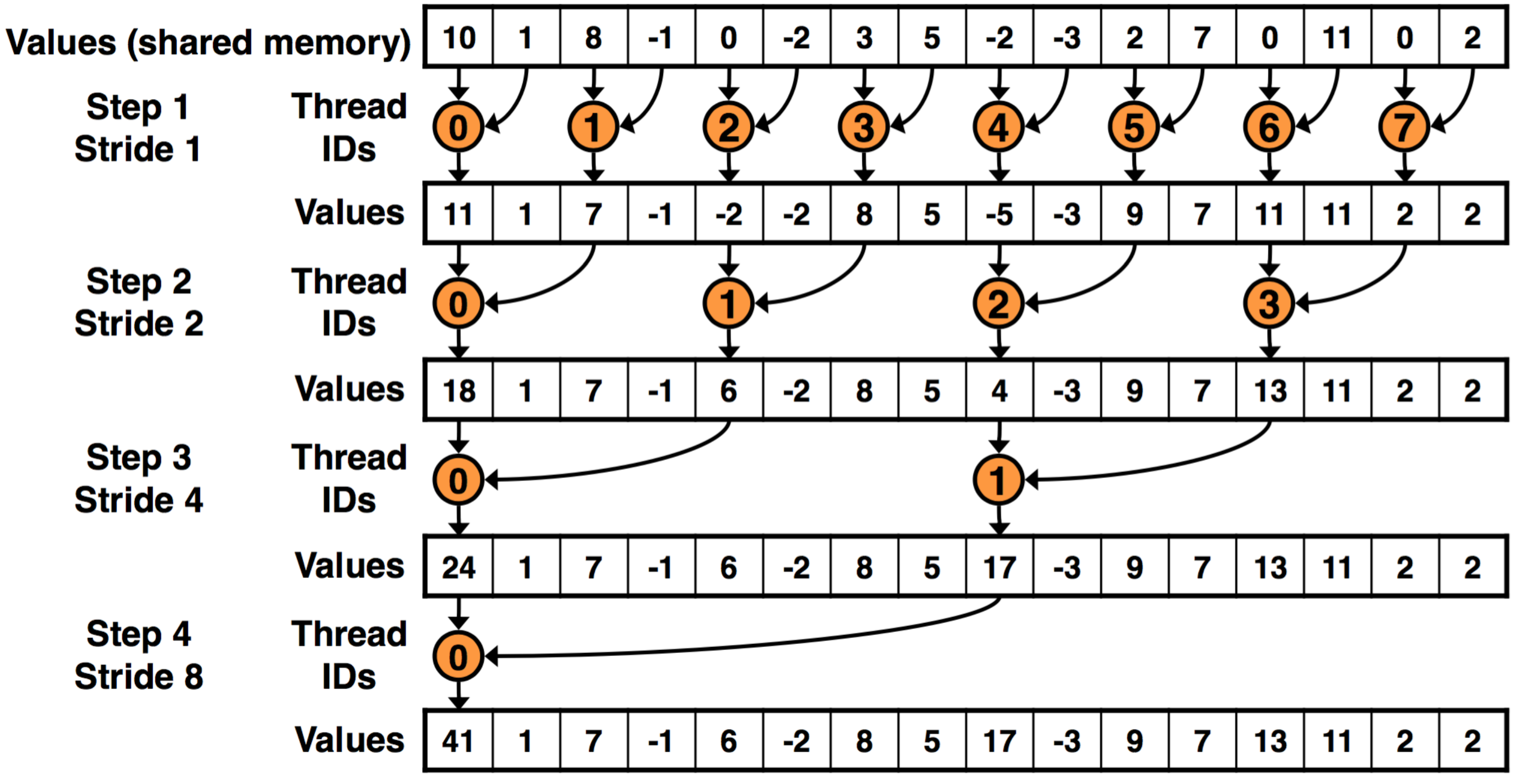
Let’s look at our kernel again…

```
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

x Warp divergence

- Only consecutive threads run

✓ No divergence

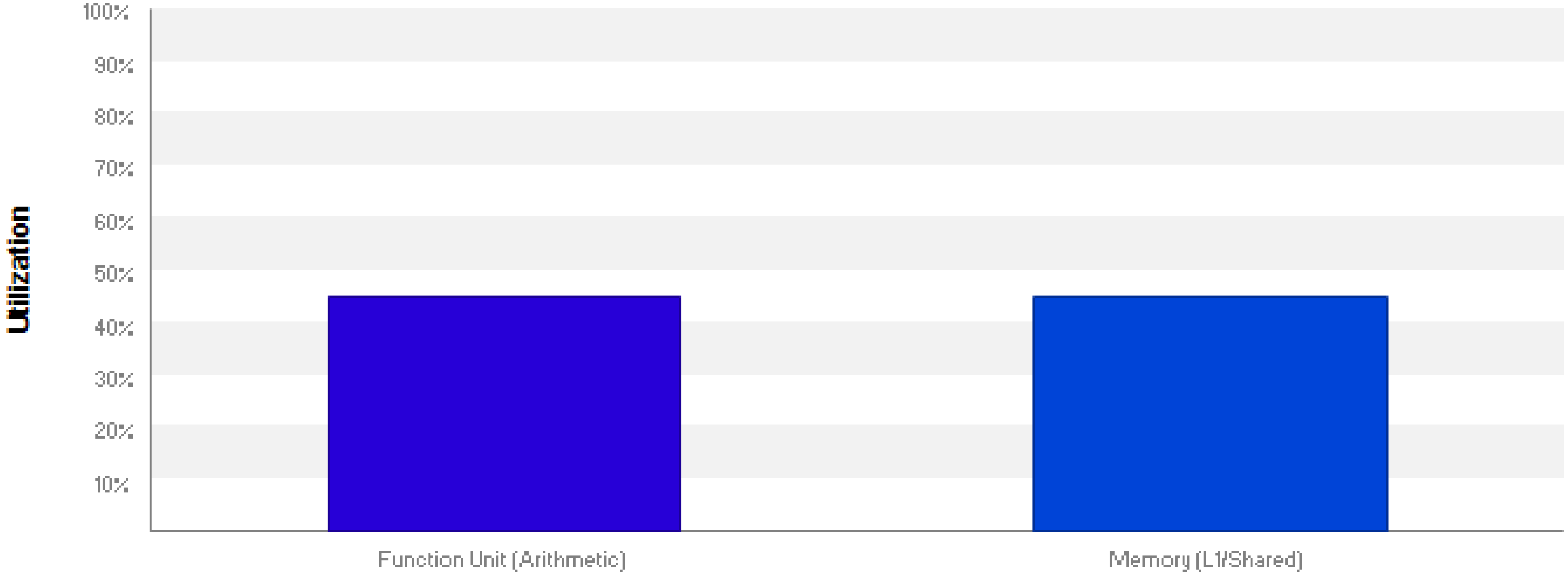
Reduction Kernel 1 Profiling with nvprof

```
sudo -E /usr/local/cuda/bin/nvprof --analysis-metrics \
       -o reduction.prof ./reduction --kernel=1
```

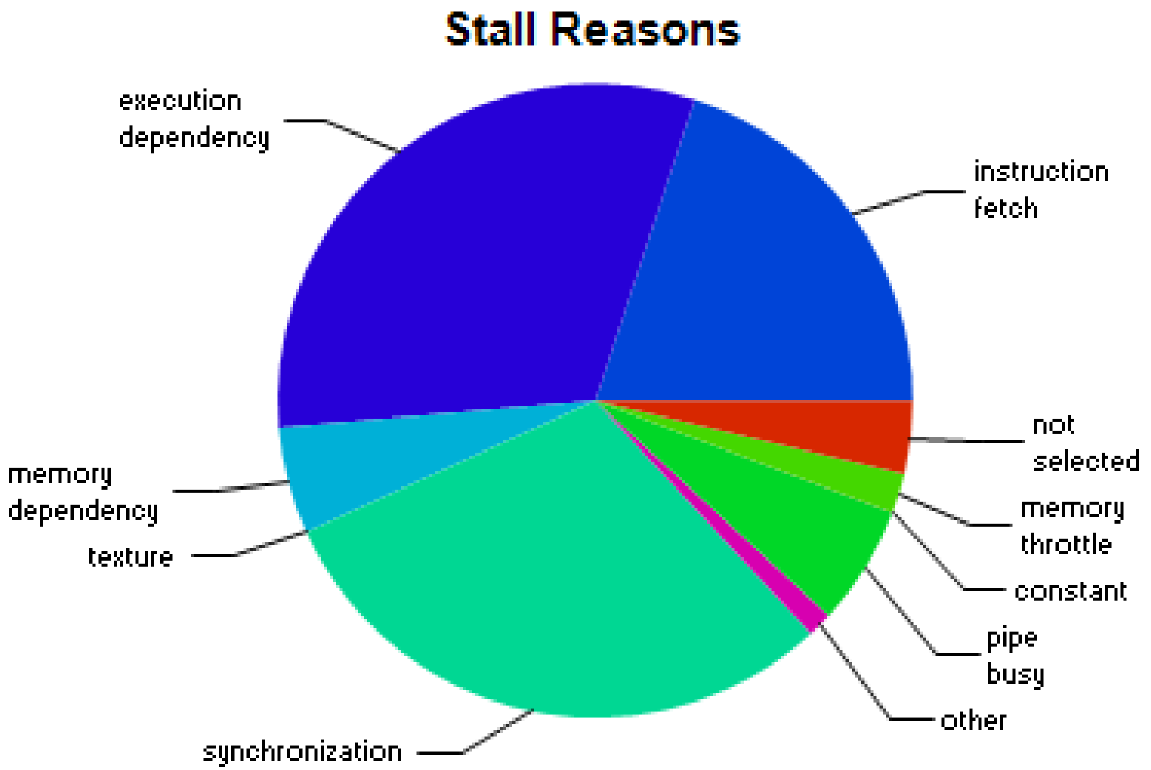
- `--analysis-metrics` collects everything needed for NVVP Guided Analysis
- Will take some time to run

Reduction Kernel 1 Profiling Results

	Throughput GB/s
Kernel 0	14.7
Kernel 1	18.1

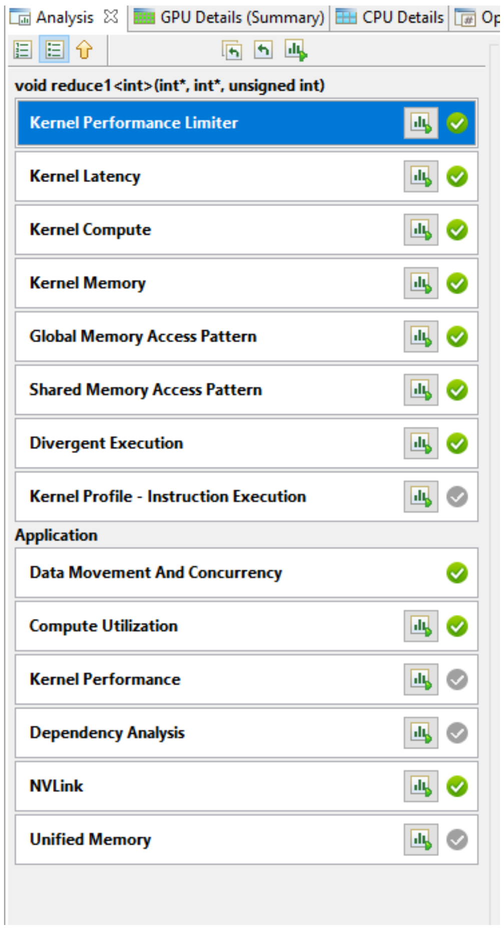


Reduction Kernel 1 Latency Analysis



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

```
107
108     __syncthreads();
109
110     // do reduction in shared mem
111     for(unsigned int s=1; s < blockDim.x; s *= 2) {
112       int index = 2 * s * tid;
113
114       if(index < blockDim.x) {
115         sdata[index] += sdata[index + s];
116       }
117
118       __syncthreads();
119     }
120
121     // write result for this block to global mem
122     if(tid == 0) {
123       g_odata[blockIdx.x] = sdata[0];
124     }
125   }
126
127   /*
128     This version uses sequential addressing -- no divergence or bank conflicts.
129   */
130   template <class T>
131   __global__ void
132   reduce2(T* g_idata, T* g_odata, unsigned int n) {
```

```
LD.E R5, [R8];
I2I.S32.S32 R8, R4;
SHR R9, R8, 0x1f;
MOV R8, R8;
MOV R9, R9;
MOV R10, R8;
MOV R11, R9;
MOV R10, R10;
MOV R11, R11;
SHF.L.U64 R11, R10, 0x2, R11;
SHL R10, R10, 0x2;
IADD R10.CC, R0, R10;
IADD.X R11, R3, R11;
MOV R8, R10;
MOV R9, R11;
MOV R8, R8;
MOV R9, R9;
LD.E R8, [R8];
IADD R5, R8, R5;
MOV R8, R10;
MOV R9, R11;
MOV R8, R8;
MOV R9, R9;
ST.E [R8], R5;
NOP.S;
```

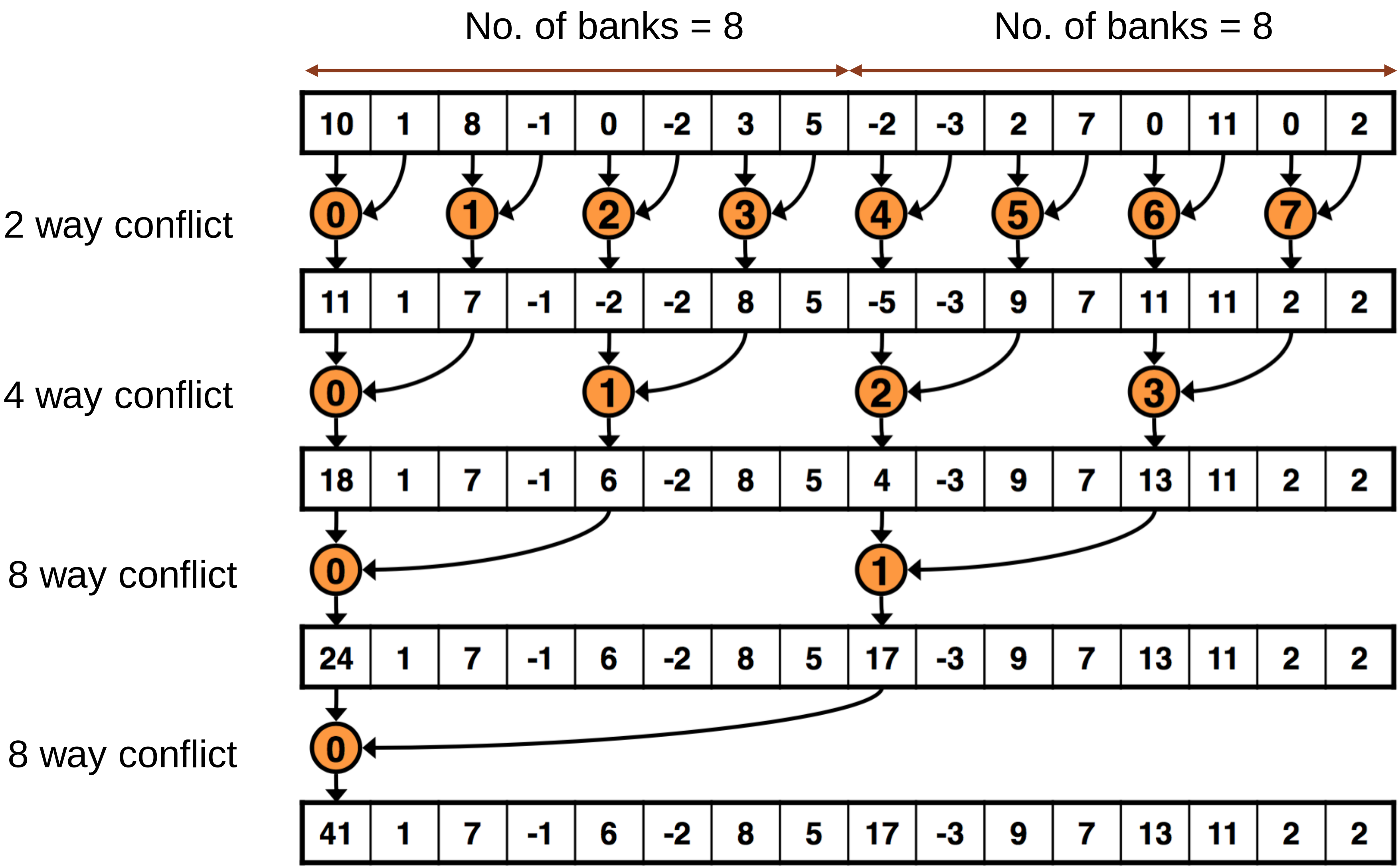
Results

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.

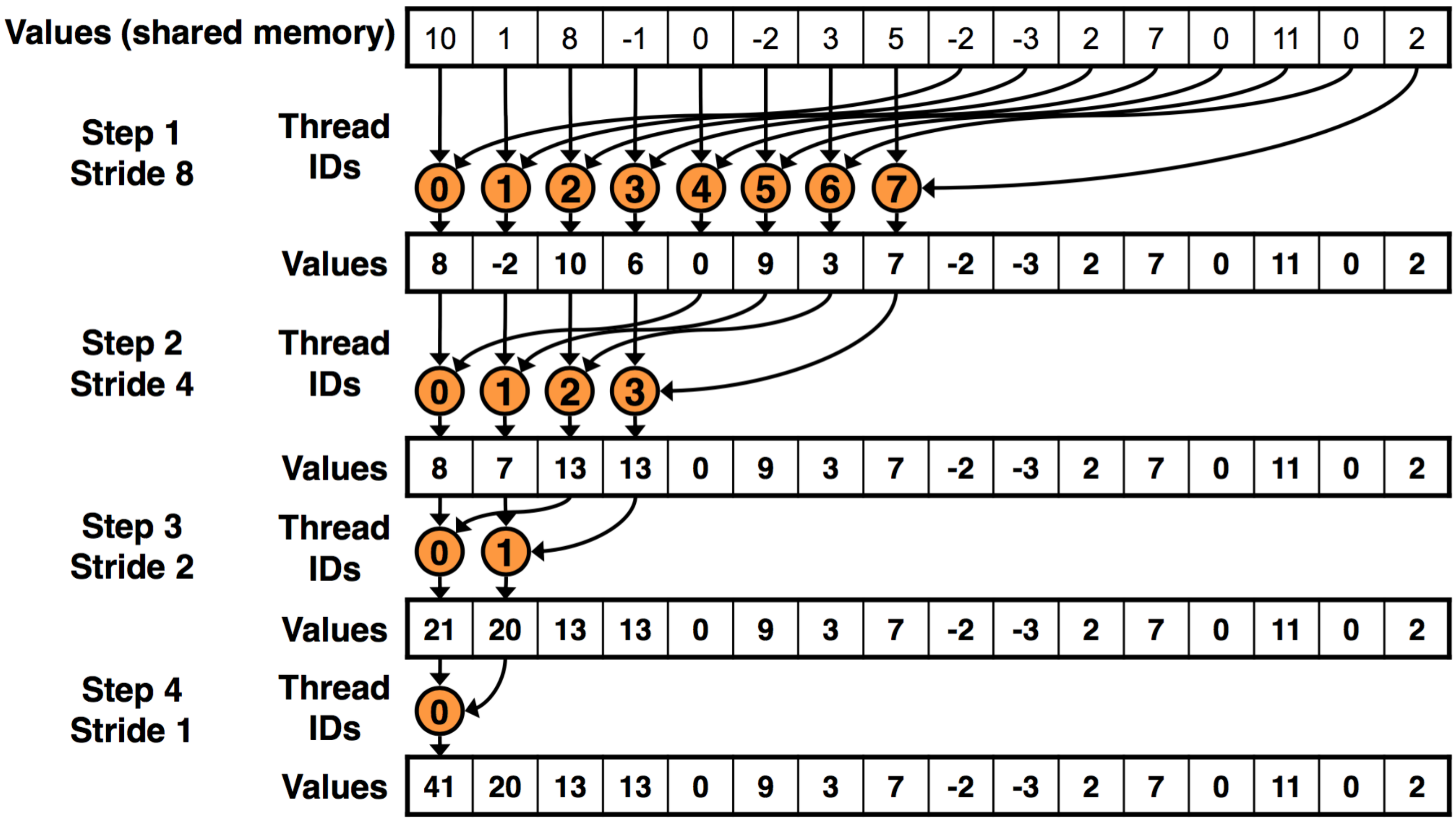
Optimization: Select each entry below to open the source code to a shared load or store within the kernel with an inefficient alignment or access pattern. For each load or store improve the alignment and access pattern of the memory access. More...

Line / File	reduction_kernel.cu - \home\wjen\reduction
115	Shared Load Transactions/Access = 2.8, Ideal Transactions/Access = 1 [1802240 transactions for 655360 total executions]
115	Shared Load Transactions/Access = 2.8, Ideal Transactions/Access = 1 [1802240 transactions for 655360 total executions]
115	Shared Store Transactions/Access = 2.8, Ideal Transactions/Access = 1 [1802240 transactions for 655360 total executions]


No. of banks = 8
No. of banks = 8
10 1 8 -1 0 -2 3 5 -2 -3 2 7 0 11 0 2
2 way conflict
0 1 2 3 4 5 6 7
11 1 7 -1 -2 -2 8 5 -5 -3 9 7 11 11 2 2
4 way conflict
0 1 2 3
18 1 7 -1 6 -2 8 5 4 -3 9 7 13 11 2 2
8 way conflict
0 1
24 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2
8 way conflict
0
41 1 7 -1 6 -2 8 5 17 -3 9 7 13 11 2 2

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

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

On your own: profile kernel 2

Thank you – any questions?