Parallel Computing with GPUs

Profiling

Dr Paul Richmond
http://paulrichmond.shef.ac.uk/teaching/COM4521/

Credits

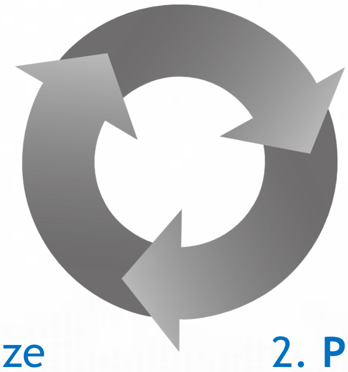
- ❑The code and much of the content from this lecture is based on the GTC2016 Talk by C. Angerer and J. Progsch (NVIDIA)
  - ❑S6112 – CUDA Optimisation with NVIDIA Nsight for Visual Studio
  - ❑Provided by NVIDIA with thanks to Joe Bungo
- ❑Content has been adapted to use Visual Profiler Guided Analysis where possible
- ❑Additional steps and analysis have been added

Learning Objectives

- ❑Understand the key performance metrics of GPU code.
- ❑Understand profiling metrics and relate this to approaches which they have already learnt to address limiting factors in their code.
- ❑Appreciate memory vs compute bound code and be able to recognise factors which contribute to this.

- ❑Profiling Introduction
- ❑The Problem
- ❑Visual Profiler Guided Analysis
  - ❑Iteration 1
  - ❑Iteration 2
  - ❑Iteration 3
  - ❑Iteration 4

The APOD Cycle



4. Deploy and Test

1. Assess
   - Identify Performance Limiter
   - Analyze Profile
   - Find Indicators

3. Optimize

3b. Build Knowledge

2. Parallelize

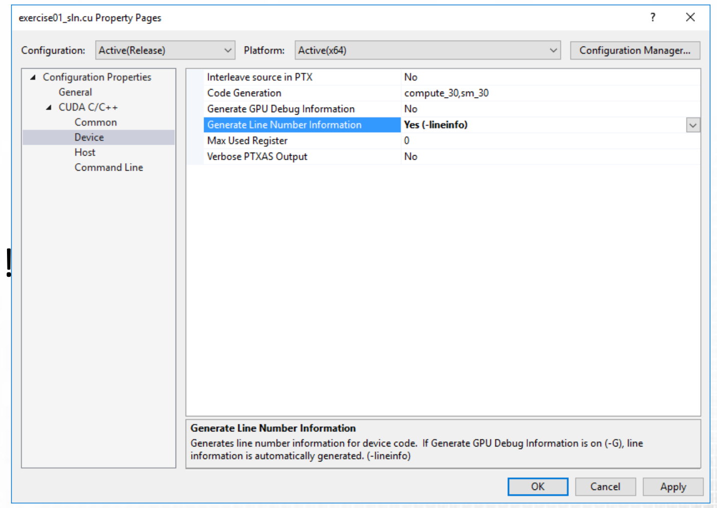
https://devblogs.nvidia.com/assess-parallelize-optimize-deploy/

CUDA Profiling Options

- ❑**Visual Profiler (1st choice)**
  - ❑Stand alone cross platform (java on Eclipse) program
  - ❑Guided performance analysis
  - ❑Links to CUDA best practice guide
  - ❑Limited support for the latest Turing/Ampere GPU architectures
- ❑NVProf
  - ❑Command line profiler
  - ❑Results can be visualised in Visual Profiler
- ❑Visual Studio Nsight Profiler
  - ❑Built into visual studio
  - ❑Detailed kernel and source level analysis (more than Visual Profiler)
  - ❑Unguided
- ❑Nsight Compute
  - ❑Modern CUDA profiler to replace visual profiler
  - ❑Standalone cross platform program
  - ❑Only supports Volta+ architectures (Uni machines are the previous Pascal arch)

Changes to your code

- ❑If you want to associate profile information with source line
  - ❑`--lineinfo` argument
  - ❑Works in release mode
- ❑Must flush GPU buffers
  - ❑`cudaDeviceReset()`
  - ❑At end of program
- ❑Only profile in Release mode!



Configuration: Active(Release) Platform: Active(x64)

Interleave source in PTX	No
Code Generation	compute_30,sm_30
Generate GPU Debug Information	No
Generate Line Number Information	**Yes (-lineinfo)**
Max Used Register	0
Verbose PTXAS Output	No

Generate Line Number Information

Generates line number information for device code. If Generate GPU Debug Information is on (-G), line information is automatically generated. (-lineinfo)

Conveyor belt model

- ❑Our GPU program is like a factory assembly line
  - ❑Data in and data out (in a new form)
  - ❑Skilled operators (multi processors) doing stuff with chunks of the data
  - ❑Both the belt and people have maximum operating speed
- ❑Ideal situation
  - ❑Conveyor belt runs at full speed
  - ❑Skilled operators always 100% busy

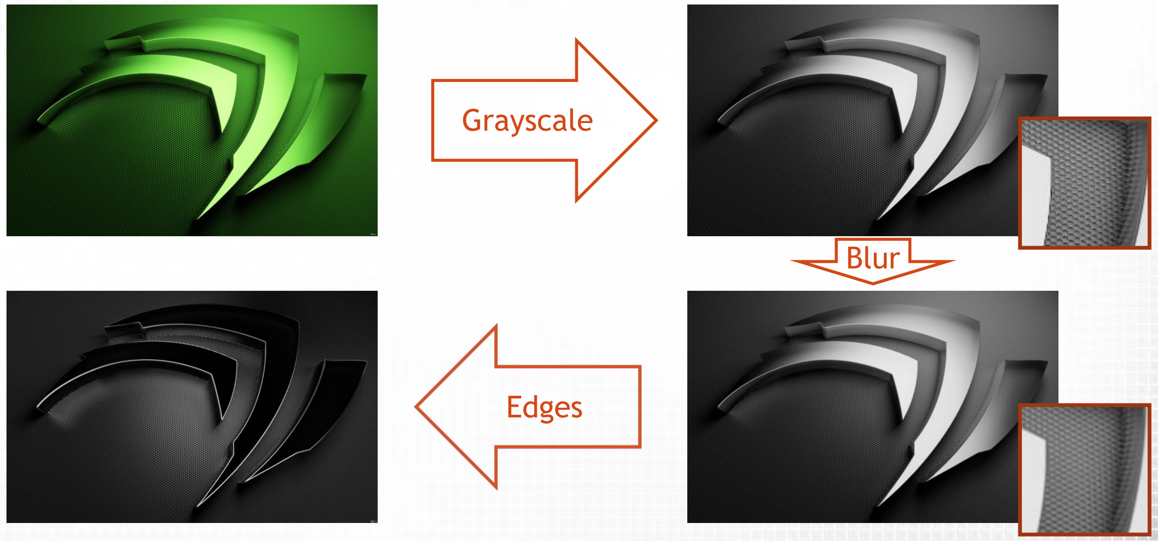
What is likely to affect this model?

Potential Performance Limiters

- ❑Memory
  - ❑Program limited by memory bandwidth
  - ❑Can't get data to the registers on the device fast enough
  - ❑*Are you using lots of global memory but not faster local memory caches?*
  - ❑*Have you exceed the amount of cache available?*
- ❑Compute
  - ❑Memory bandwidth well below peak
  - ❑GPU is too busy performing compute
  - ❑*Have you got high levels of divergence with low warp execution efficiency ?*
- ❑Latency
  - ❑Poor occupancy = not enough active threads
  - ❑Instruction execution stalls due to poor memory access patterns (sparse or poorly used data)
  - ❑*Is problem size or block size too small? Are you using the memory bandwidth effectively (cache line utilisation)?*

- ❑Profiling Introduction
- ❑The Problem
- ❑Visual Profiler Guided Analysis
  - ❑Iteration 1
  - ❑Iteration 2
  - ❑Iteration 3
  - ❑Iteration 4

Introducing the Application



Introducing the Application

- ❑Grayscale Conversion

```
// r, g, b: Red, green, blue components of the pixel p
foreach pixel p:
  p = 0.298839f*r + 0.586811f*g + 0.114350f*b;
```

Introducing the Application

- ❑Blur: 7x7 Gaussian Filter

1	6	15	20	15	6	1
6	36	90	120	90	36	6
15	90	225	300	225	90	15
20	120	300	400	300	120	20
15	90	225	300	225	90	15
6	36	90	120	90	36	6
1	6	15	20	15	6	1

```
foreach pixel p:
  G = weighted sum of p and its 48 neighbors
  p  = G/256
```

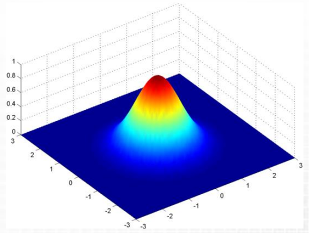
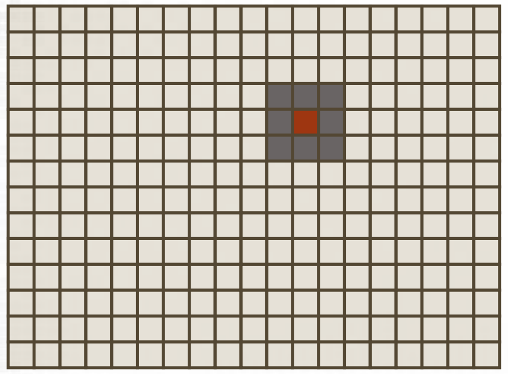


Image from Wikipedia

Introducing the Application

- ❑Edges: 3x3 Sobel Filters



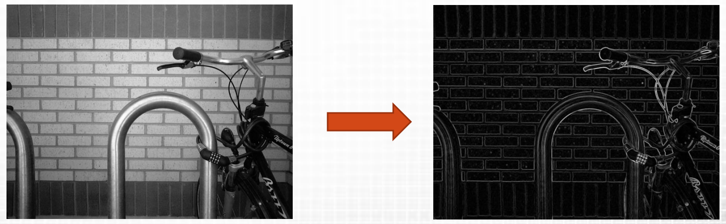
```
foreach pixel p:
  Gx = weighted sum of p and its 8 neighbors
  Gy = weighted sum of p and its 8 neighbors
  p  = sqrt(Gx + Gy)
```

Weights for Gx:

-1	0	1
-2	0	2
-1	0	1

Weights for Gy:

1	2	1
0	0	0
-1	-2	-1



The Starting Code

```
void gaussian_filter_7x7_v0(int w, int h, const uchar *src, uchar *dst)
{
  // Position of the thread in the image.
  const int x = blockIdx.x*blockDim.x + threadIdx.x;
  const int y = blockIdx.y*blockDim.y + threadIdx.y;

  // Early exit if the thread is not in the image.
  if( !in_img(x, y, w, h) )
    return;

  // Load the 48 neighbours and myself.
  int n[7][7];
  for( int j = -3 ; j <= 3 ; ++j )
    for( int i = -3 ; i <= 3 ; ++i )
      n[j+3][i+3] = in_img(x+i, y+j, w, h) ? (int) src[(y+j)*w + (x+i)] : 0;

  // Compute the convolution.
  int p = 0;
  for( int j = 0 ; j < 7 ; ++j )
    for( int i = 0 ; i < 7 ; ++i )
      p += gaussian_filter[j][i] * n[j][i];

  // Store the result.
  dst[y*w + x] = (uchar) (p / 256);
}
```

What is good and what is bad?

- ❑https://github.com/chmaruni/nsight-gtc

The Starting Code

```
void gaussian_filter_7x7_v0(int w, int h, const uchar *src, uchar *dst)
{
  // Position of the thread in the image.
  const int x = blockIdx.x*blockDim.x + threadIdx.x;
  const int y = blockIdx.y*blockDim.y + threadIdx.y;

  // Early exit if the thread is not in the image.
  if( !in_img(x, y, w, h) )
    return;

  // Load the 48 neighbours and myself.
  int n[7][7];
  for( int j = -3 ; j <= 3 ; ++j )
    for( int i = -3 ; i <= 3 ; ++i )
      n[j+3][i+3] = in_img(x+i, y+j, w, h) ? (int) src[(y+j)*w + (x+i)] : 0;

  // Compute the convolution.
  int p = 0;
  for( int j = 0 ; j < 7 ; ++j )
    for( int i = 0 ; i < 7 ; ++i )
      p += gaussian_filter[j][i] * n[j][i];

  // Store the result.
  dst[y*w + x] = (uchar) (p / 256);
}
```

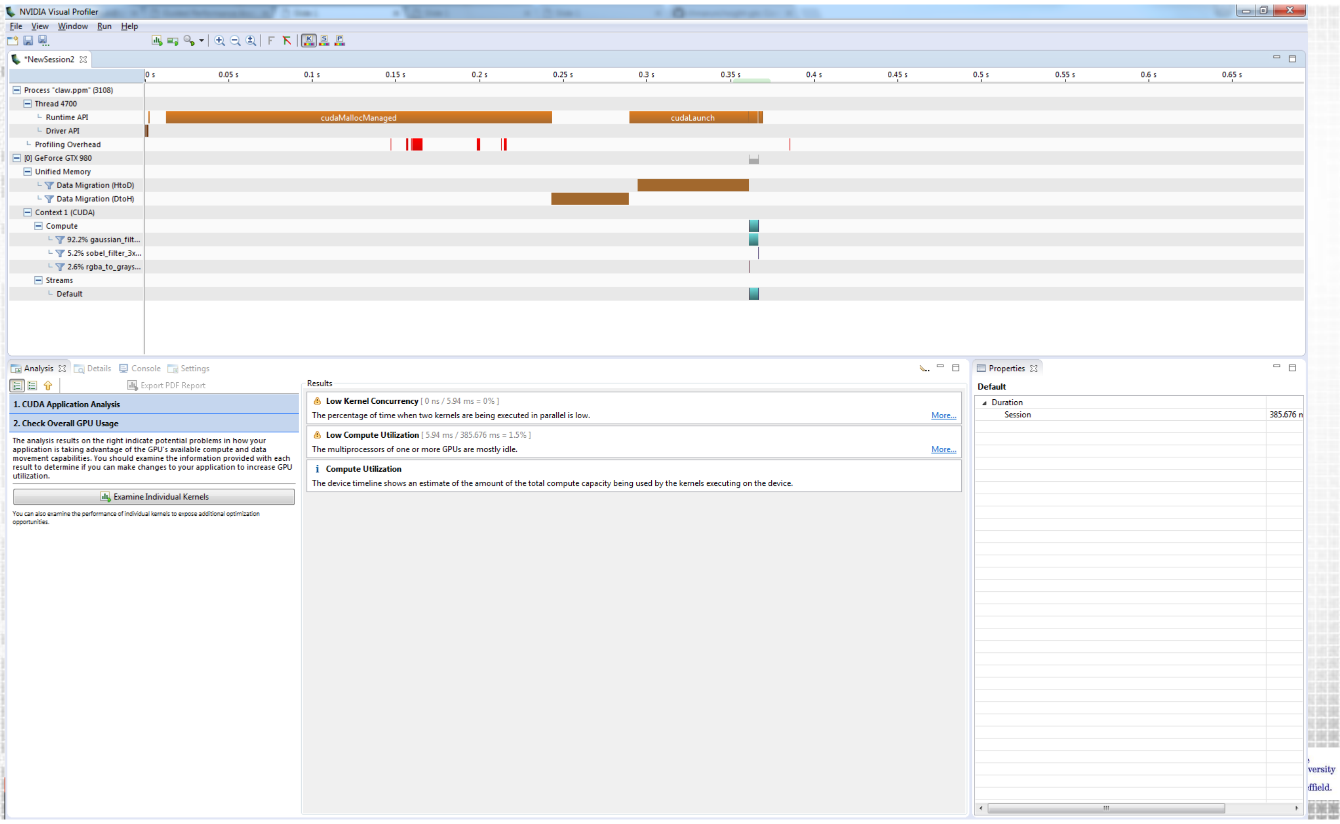
What is good and what is bad?

- ❑https://github.com/chmaruni/nsight-gtc

Profiling Machine

- ❑NVIDIA GeForce GTX980
  - ❑GM200
  - ❑Compute Capability SM5.2
- ❑CUDA 7.0
- ❑Windows 7
- ❑Visual Studio 2013
- ❑Nsight Visual Studio Edition 5.0

- ❑Profiling Introduction
- ❑The Problem
- ❑Visual Profiler Guided Analysis
  - ❑Iteration 1



Results

1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Examine Individual Kernels

You can also examine the performance of individual kernels to expose additional optimization opportunities.

⚠ **Low Kernel Concurrency** [0 ns / 5.94 ms = 0%]

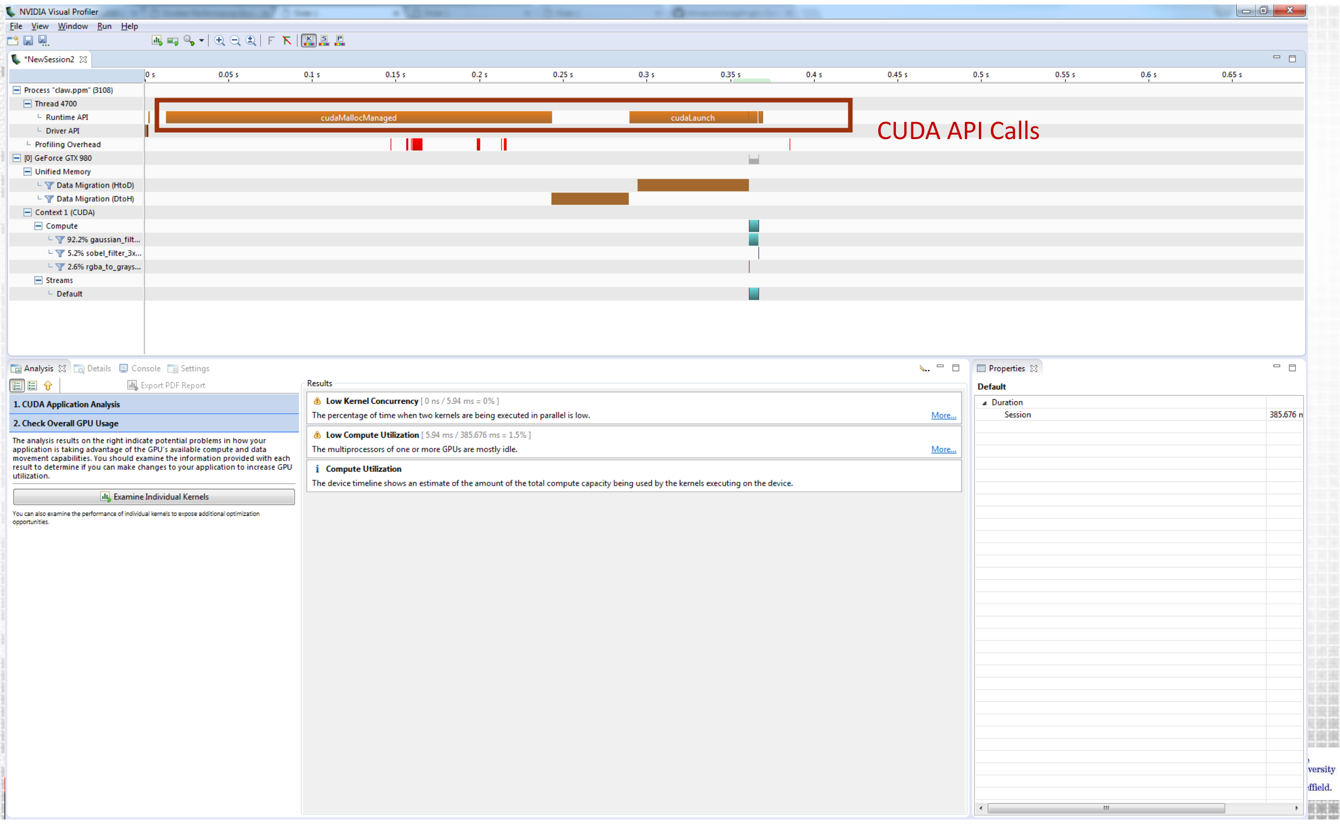
The percentage of time when two kernels are being executed in parallel is low. More...

⚠ **Low Compute Utilization** [5.94 ms / 385.676 ms = 1.5%]

The multiprocessors of one or more GPUs are mostly idle. More...

ℹ **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.



CUDA API Calls

Results

1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Examine Individual Kernels

You can also examine the performance of individual kernels to expose additional optimization opportunities.

⚠ **Low Kernel Concurrency** [0 ns / 5.94 ms = 0%]

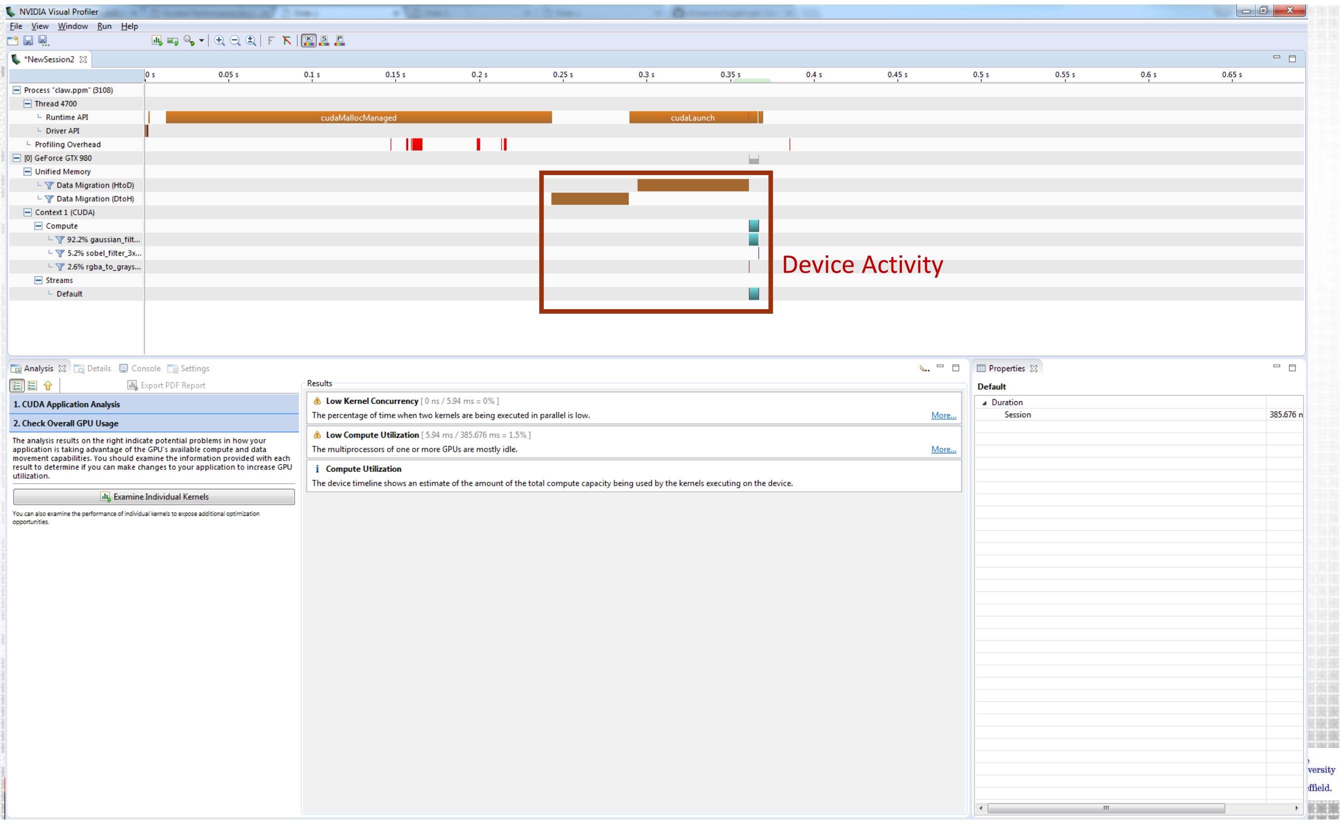
The percentage of time when two kernels are being executed in parallel is low. More...

⚠ **Low Compute Utilization** [5.94 ms / 385.676 ms = 1.5%]

The multiprocessors of one or more GPUs are mostly idle. More...

ℹ **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.



Device Activity

1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Examine Individual Kernels

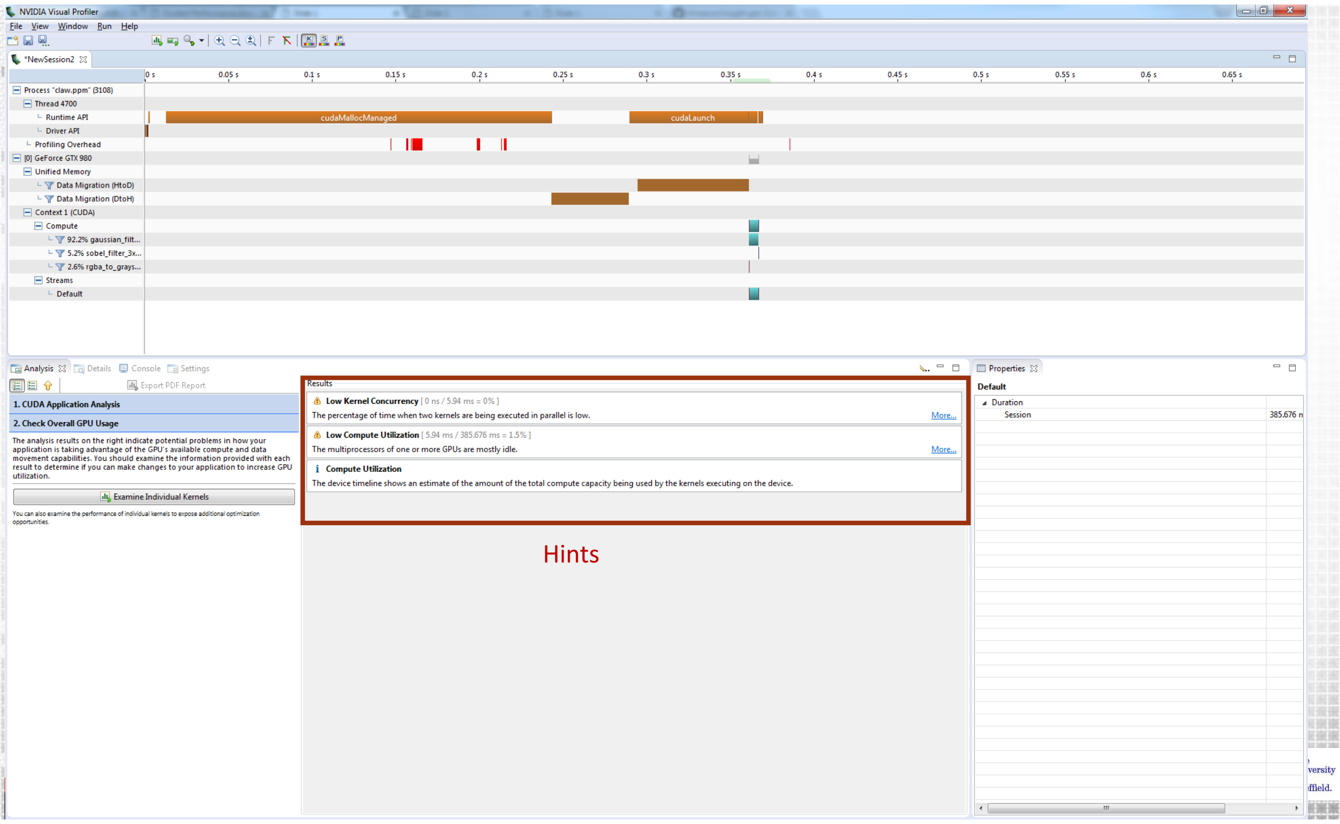
You can also examine the performance of individual kernels to expose additional optimization opportunities.

Results

Low Kernel Concurrency [0 ns / 5.94 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Compute Utilization [5.94 ms / 385.676 ms = 1.5%]
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.



Hints

Results

Low Kernel Concurrency [0 ns / 5.94 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

Low Compute Utilization [5.94 ms / 385.676 ms = 1.5%]
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.

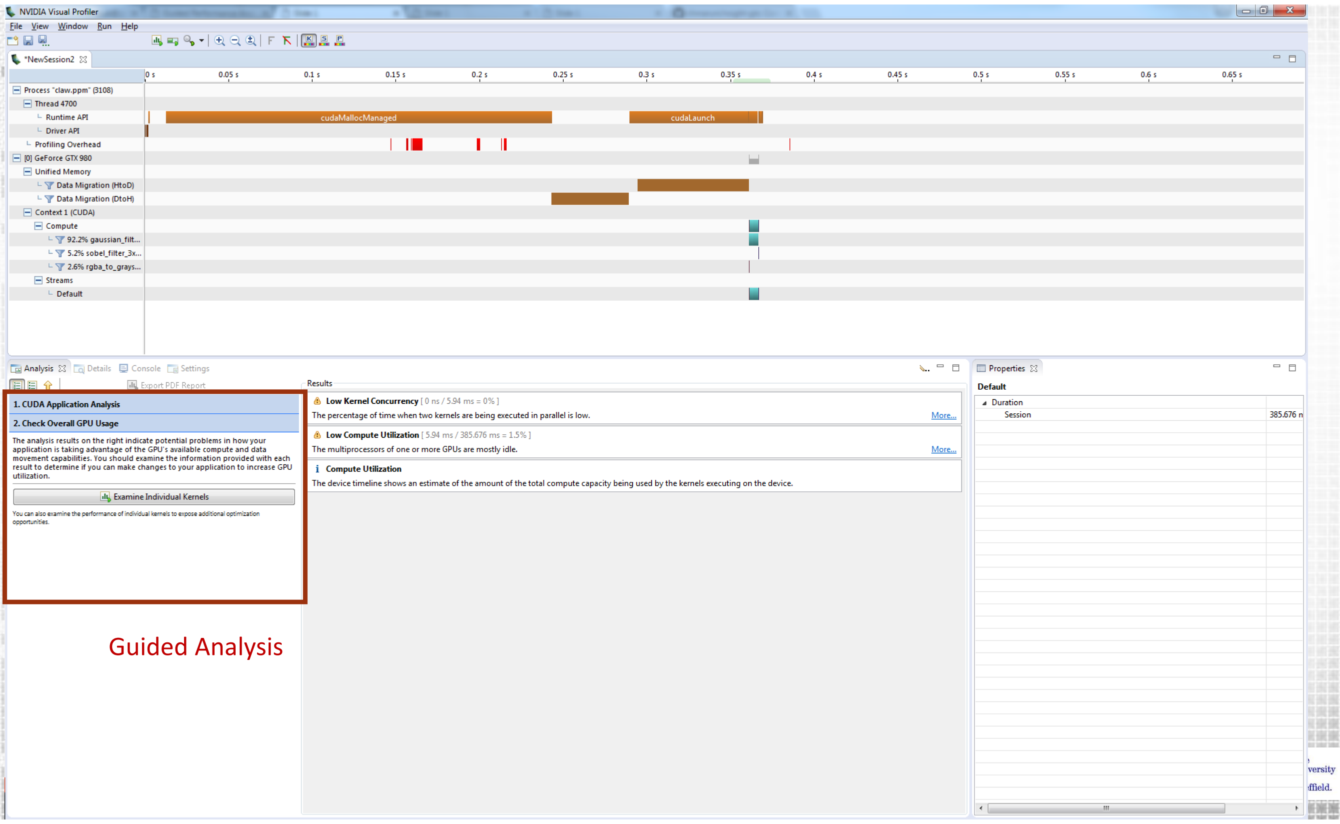
Results

Low Kernel Concurrency [0 ns / 5.94 ms = 0%]
The percentage of time when two kernels are being executed in parallel is low. More...

- ❑We are using only a single stream
- ❑Kernels have data dependencies so cant be executed in parallel

Low Compute Utilization [5.94 ms / 385.676 ms = 1.5%]
The multiprocessors of one or more GPUs are mostly idle. More...

- ❑This is a problem
- ❑The guided analysis will try and address this



Guided Analysis

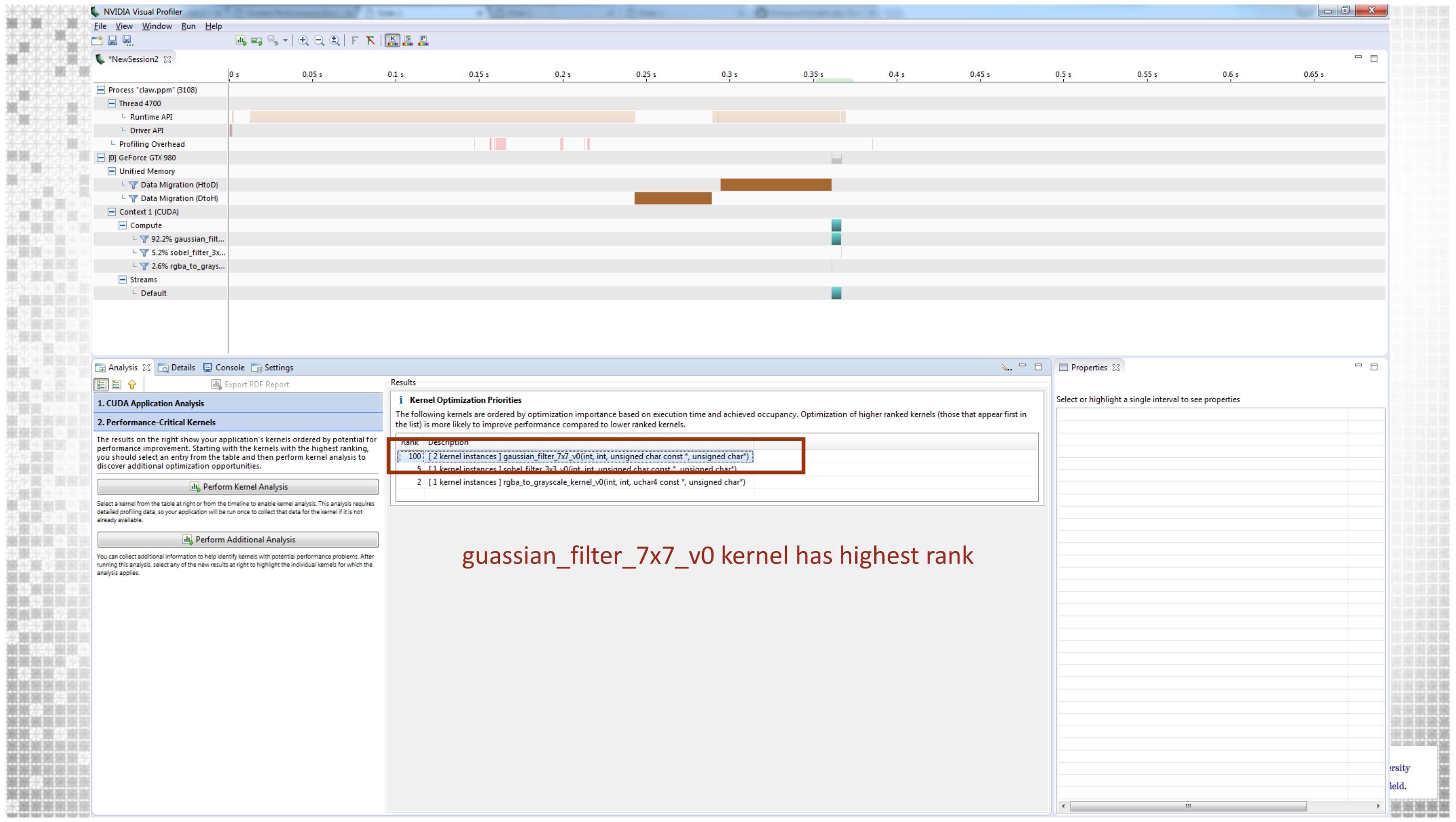
1. CUDA Application Analysis

2. Check Overall GPU Usage

The analysis results on the right indicate potential problems in how your application is taking advantage of the GPU's available compute and data movement capabilities. You should examine the information provided with each result to determine if you can make changes to your application to increase GPU utilization.

Examine Individual Kernels

You can also examine the performance of individual kernels to expose additional optimization opportunities.



1. CUDA Application Analysis

2. Performance-Critical Kernels

The results on the right show your application's kernels ordered by potential for performance improvement. Starting with the kernels with the highest ranking, you should select an entry from the table and then perform kernel analysis to discover additional optimization opportunities.

Perform Kernel Analysis

Select a kernel from the table at right or from the timeline to enable kernel analysis. This analysis requires detailed profiling data, so your application will be run once to collect that data for the kernel if it is not already available.

Perform Additional Analysis

You can collect additional information to help identify kernels with potential performance problems. After running this analysis, select any of the new results at right to highlight the individual kernels for which the analysis applies.

Results

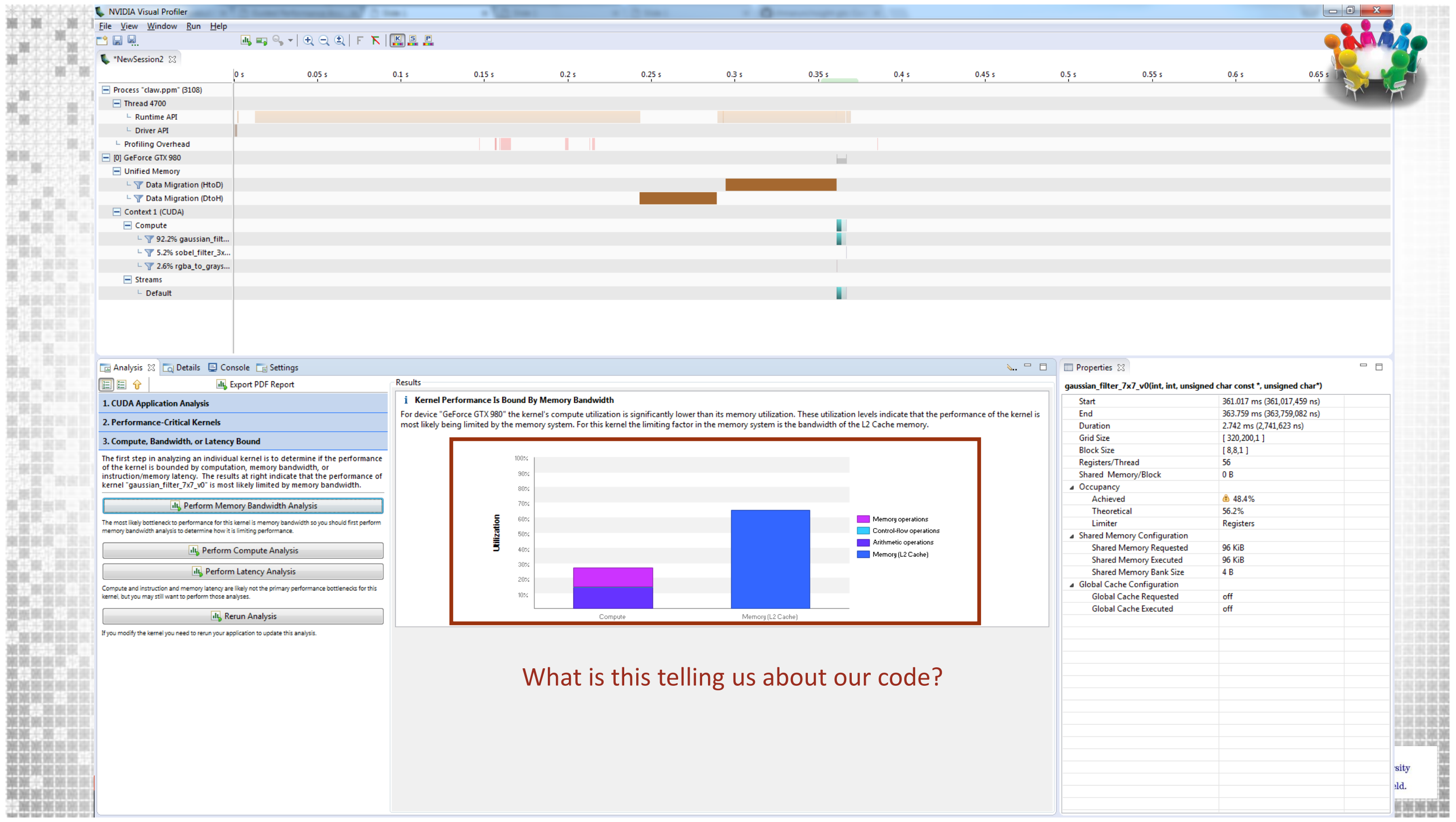
Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[2 kernel instances] gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)
5	[1 kernel instances] sobel_filter_3x3_v0(int, int, unsigned char const *, unsigned char*)
2	[1 kernel instances] rgba_to_grayscale_kernel_v0(int, int, uchar4 const *, unsigned char*)

Select or highlight a single interval to see properties

guassian_filter_7x7_v0 kernel has highest rank



1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "gaussian_filter_7x7_v0" is most likely limited by memory bandwidth.

Perform Memory Bandwidth Analysis

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Latency Analysis

Compute and instruction and memory latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Results

Kernel Performance Is Bound By Memory Bandwidth

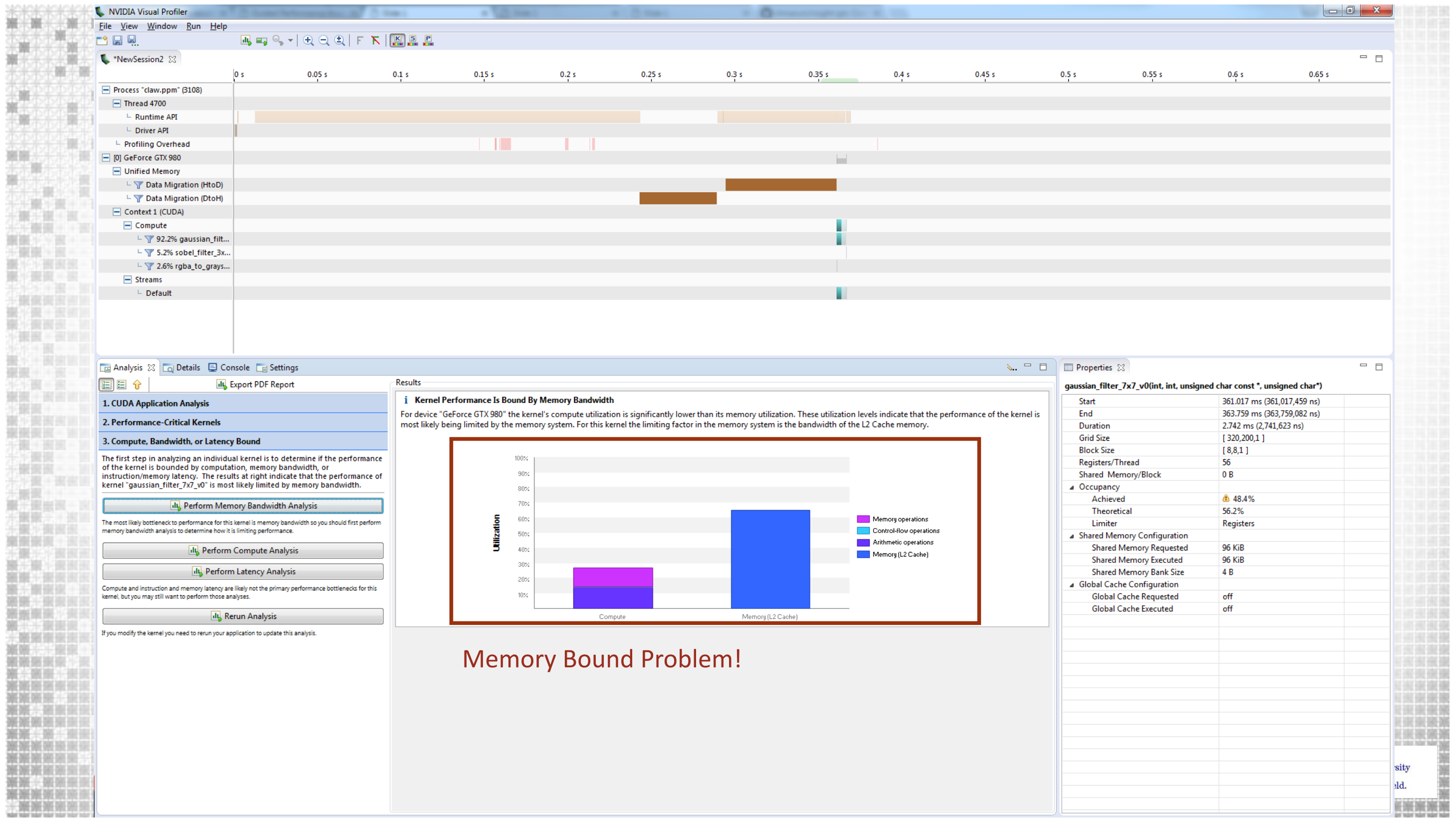
For device "GeForce GTX 980" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the L2 Cache memory.

Properties

gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)

Start	361.017 ms (361,017,459 ns)
End	363.759 ms (363,759,082 ns)
Duration	2.742 ms (2,741,623 ns)
Grid Size	[320,200,1]
Block Size	[8,8,1]
Registers/Thread	56
Shared Memory/Block	0 B
Occupancy	
Achieved	⚠ 48.4%
Theoretical	56.2%
Limiter	Registers
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

What is this telling us about our code?



1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "gaussian_filter_7x7_v0" is most likely limited by memory bandwidth.

Perform Memory Bandwidth Analysis

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Latency Analysis

Compute and instruction and memory latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Results

Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GTX 980" 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 L2 Cache memory.

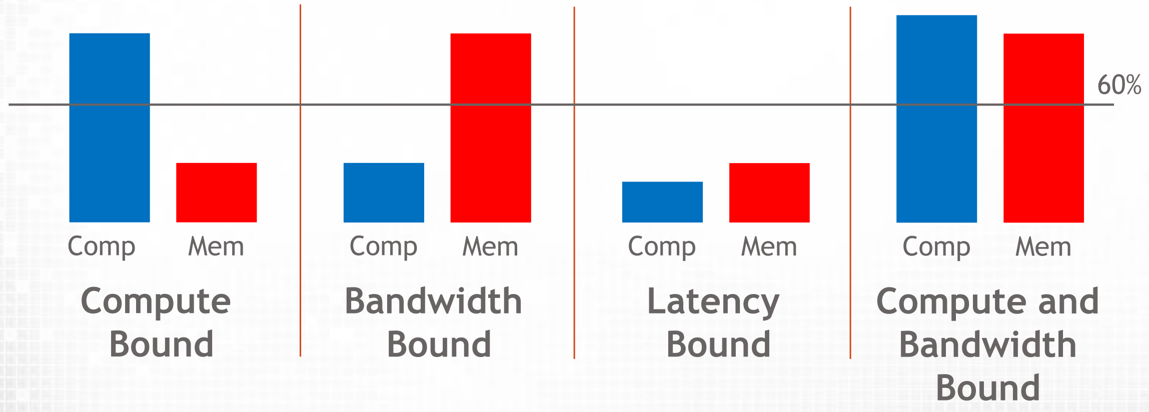
Properties

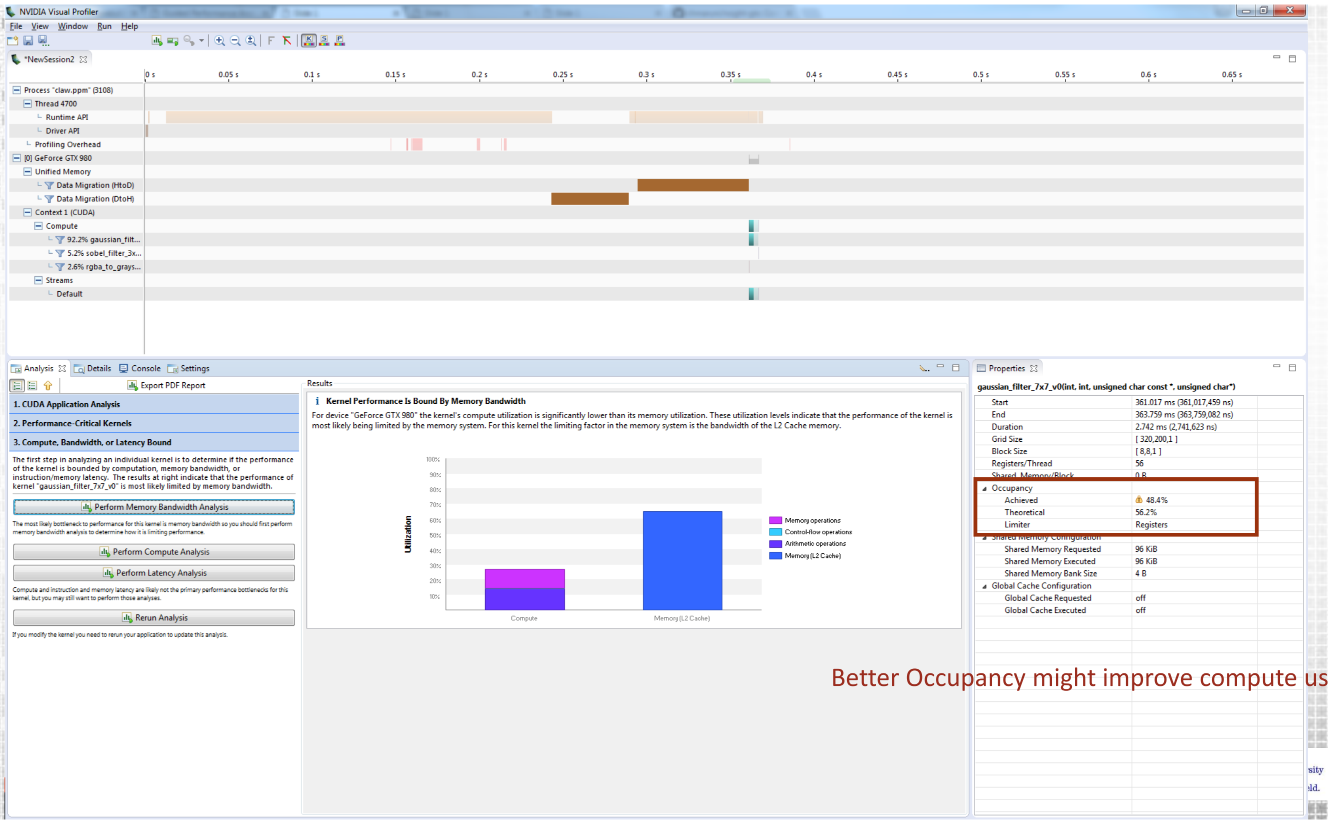
gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)

Start	361.017 ms (361,017,459 ns)
End	363.759 ms (363,759,082 ns)
Duration	2.742 ms (2,741,623 ns)
Grid Size	[320,200,1]
Block Size	[8,8,1]
Registers/Thread	56
Shared Memory/Block	0 B
Occupancy	
Achieved	⚠ 48.4%
Theoretical	56.2%
Limiter	Registers
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

Memory Bound Problem!

Memory vs Compute vs Latency





1. CUDA Application Analysis

2. Performance-Critical Kernels

3. Compute, Bandwidth, or Latency Bound

The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "gaussian_filter_7x7_v0" is most likely limited by memory bandwidth.

Perform Memory Bandwidth Analysis

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

Perform Compute Analysis

Perform Latency Analysis

Compute and instruction and memory latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

Rerun Analysis

If you modify the kernel you need to rerun your application to update this analysis.

Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GTX 980" 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 L2 Cache memory.

gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)

Start	361.017 ms (361,017,459 ns)
End	363.759 ms (363,759,082 ns)
Duration	2.742 ms (2,741,623 ns)
Grid Size	[320,200,1]
Block Size	[8,8,1]
Registers/Thread	56
Shared Memory/Block	0 B
Occupancy	
Achieved	⚠ 48.4%
Theoretical	56.2%
Limiter	Registers
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off

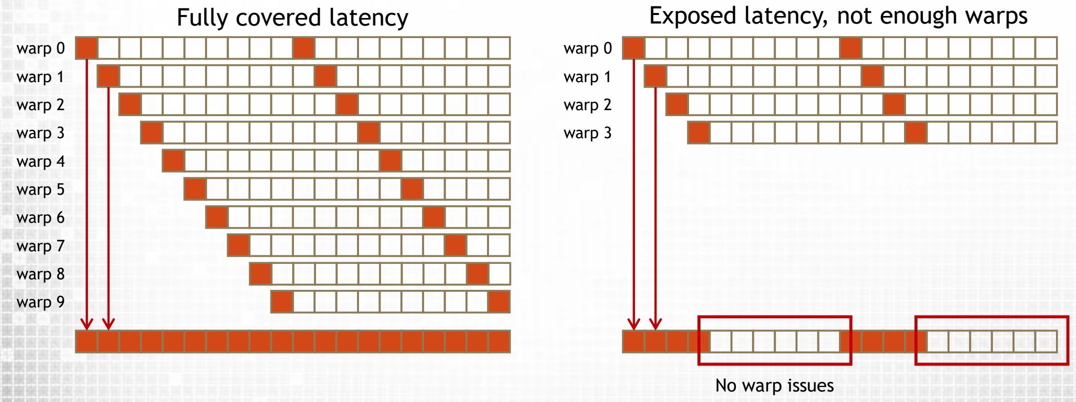
Better Occupancy might improve compute use

What about occupancy?

- ❑Occupancy: *"number of active warps over max warps supported"*
- ❑Increasing achieved occupancy can hide latency
  - ❑More warps available for execution = more to hide latency

🟥 The warp issues

□ The warp waits (latency)



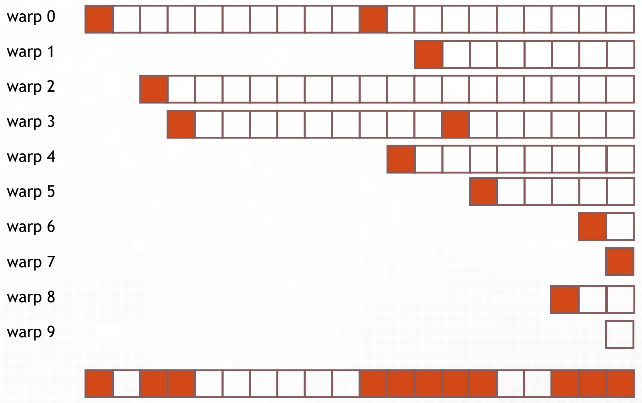
Occupancy

- ❑In our case we are not achieving theoretical occupancy (we have latency)

What is the problem here?

🟥 The warp issues

□ The warp waits (latency)

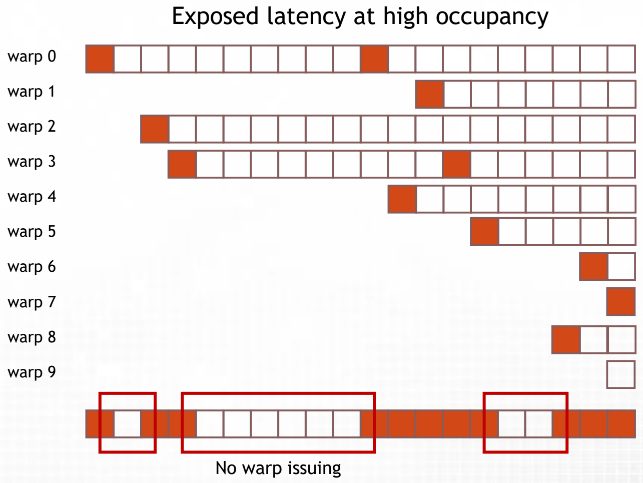


Occupancy

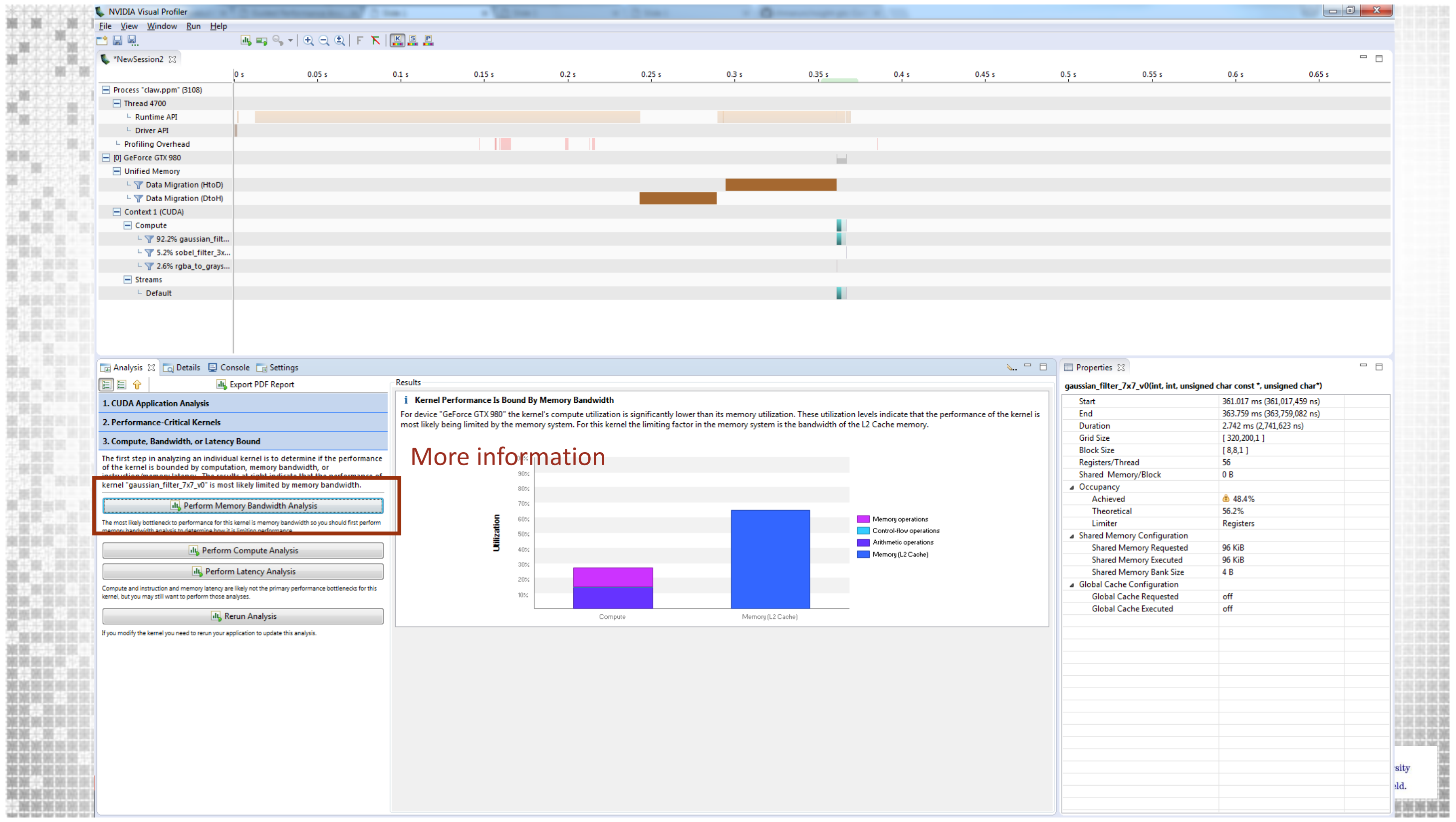
- ❑In our case we have good occupancy but still high latency
  - ❑Schedulers cant find eligible warps at every cycle

🟥 The warp issues

□ The warp waits (latency)



Warps are waiting for memory (transactions)



The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "gaussian_filter_7x7_v0" is most likely limited by memory bandwidth.

The most likely bottleneck to performance for this kernel is memory bandwidth so you should first perform memory bandwidth analysis to determine how it is limiting performance.

Compute and instruction and memory latency are likely not the primary performance bottlenecks for this kernel, but you may still want to perform those analyses.

If you modify the kernel you need to rerun your application to update this analysis.

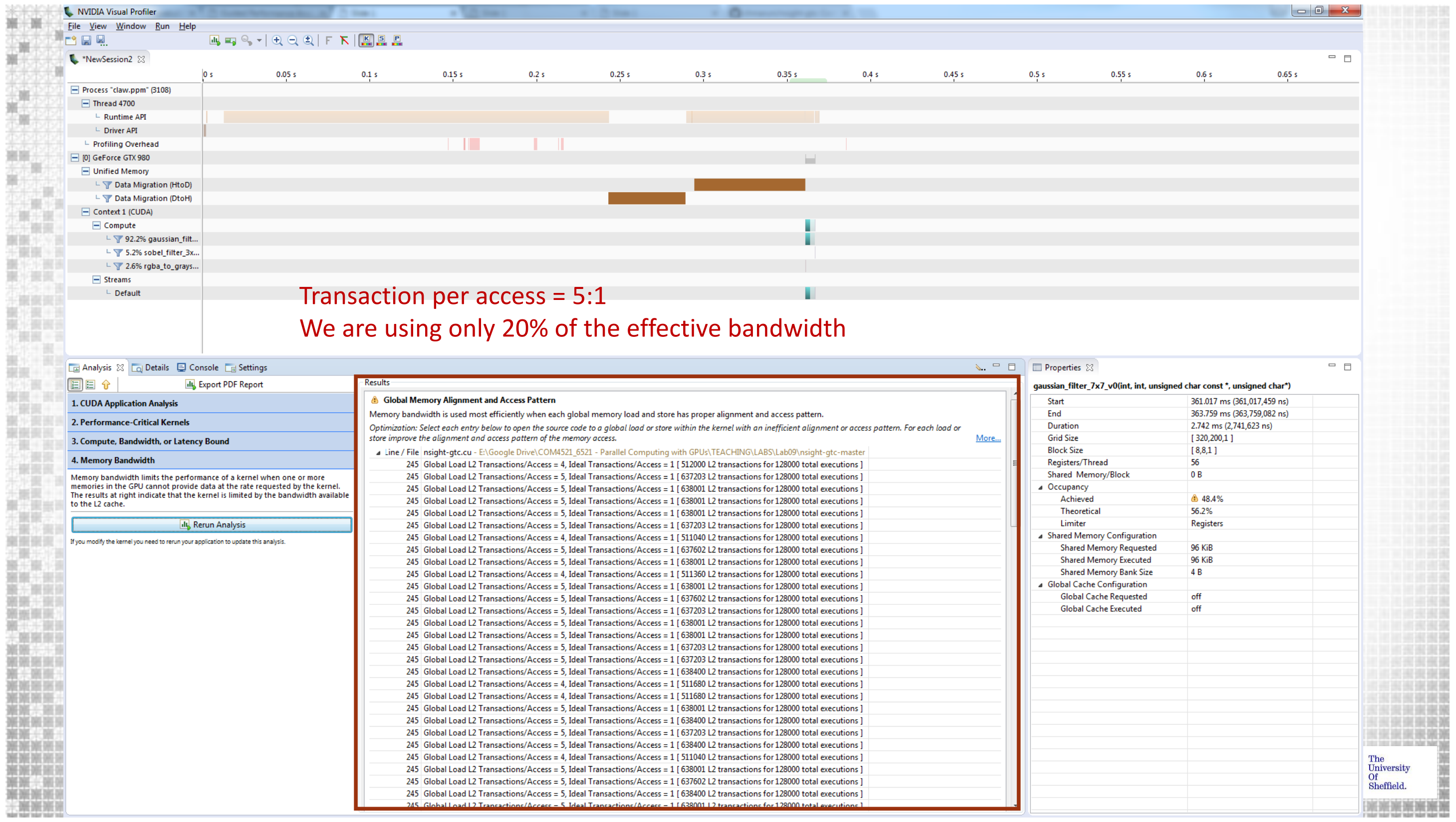
Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GTX 980" 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 L2 Cache memory.

More information

gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)

Start	361.017 ms (361,017,459 ns)
End	363.759 ms (363,759,082 ns)
Duration	2.742 ms (2,741,623 ns)
Grid Size	[320,200,1]
Block Size	[8,8,1]
Registers/Thread	56
Shared Memory/Block	0 B
Occupancy	
Achieved	⚠ 48.4%
Theoretical	56.2%
Limiter	Registers
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off



Transaction per access = 5:1

We are using only 20% of the effective bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results at right indicate that the kernel is limited by the bandwidth available to the L2 cache.

If you modify the kernel you need to rerun your application to update this analysis.

Global Memory Alignment and Access Pattern

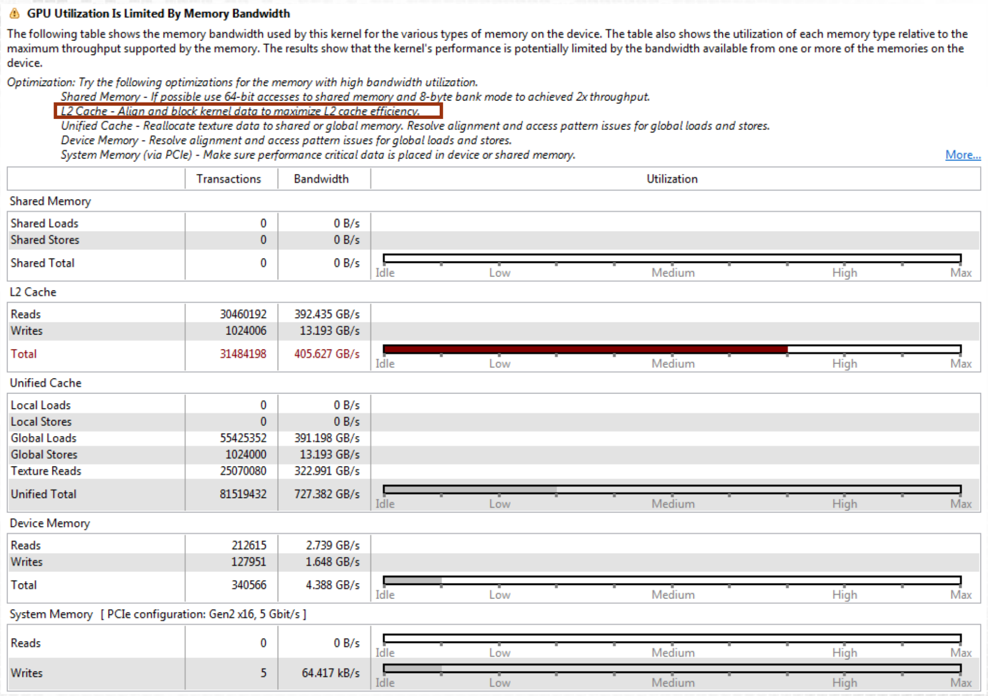
Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global 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 nsight-gtc.cu - E:\Google Drive\COM4521_6521 - Parallel Computing with GPUs\TEACHING\LABS\Lab09\nsight-gtc-master
245 Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 512000 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511040 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637602 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511360 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637602 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511680 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511680 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511040 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637602 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245 Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
```

gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)

Start	361.017 ms (361,017,459 ns)
End	363.759 ms (363,759,082 ns)
Duration	2.742 ms (2,741,623 ns)
Grid Size	[320,200,1]
Block Size	[8,8,1]
Registers/Thread	56
Shared Memory/Block	0 B
Occupancy	
Achieved	⚠ 48.4%
Theoretical	56.2%
Limiter	Registers
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off



⚠ 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.

Shared Memory - If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieved 2x throughput.

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

Unified Cache - Reallocate texture data to shared or global memory. Resolve alignment and access pattern issues for global loads and stores.

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.

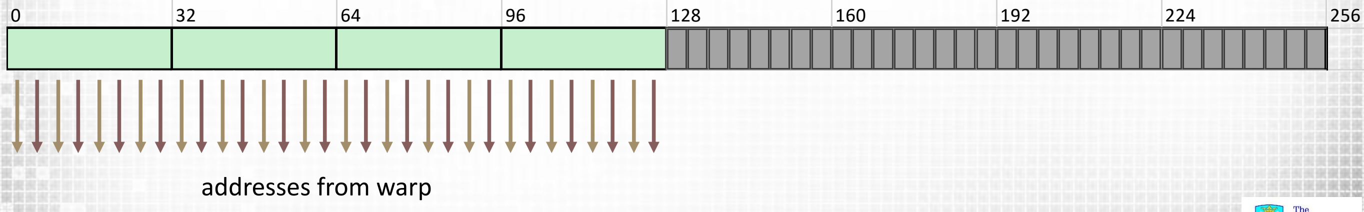
More...

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	Idle Low Medium High Max
L2 Cache			
Reads	30460192	392.435 GB/s	
Writes	1024006	13.193 GB/s	
Total	31484198	405.627 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	55425352	391.198 GB/s	
Global Stores	1024000	13.193 GB/s	
Texture Reads	25070080	322.991 GB/s	
Unified Total	81519432	727.382 GB/s	Idle Low Medium High Max
Device Memory			
Reads	212615	2.739 GB/s	
Writes	127951	1.648 GB/s	
Total	340566	4.388 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	64.417 kB/s	Idle Low Medium High Max

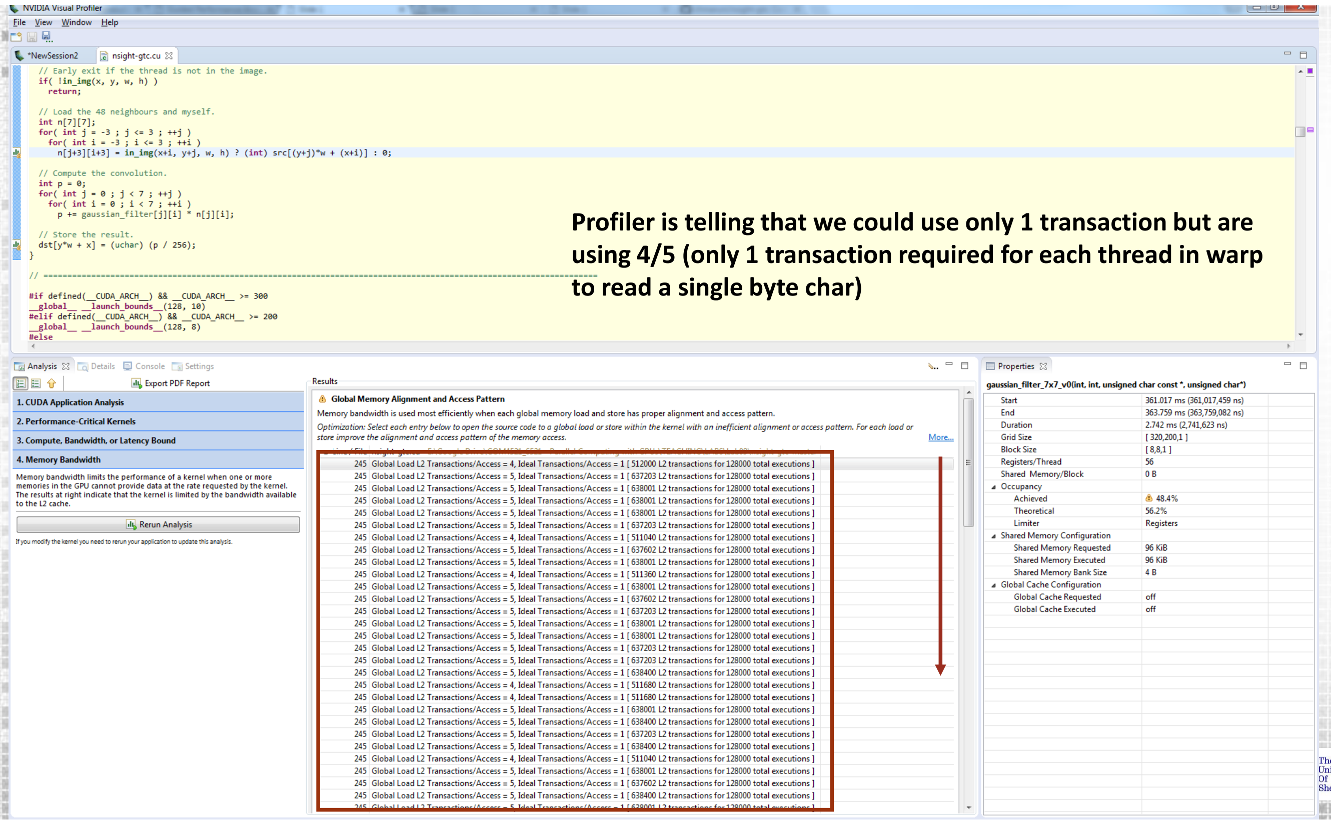
Transactions per access?

- ❑Think back to Lecture 11
  - ❑To get 100% efficiency **our threads need to access consecutive 4 byte values**
  - ❑32 Threads in warp accessing 4B each
    - ❑128B total via 4 L2 cache lines

```
__global__ void copy(float *odata, float* idata)
    int xid = blockIdx.x * blockDim.x + threadIdx.x;
    odata[xid] = idata[xid];
}
```



addresses from warp



```
  // Early exit if the thread is not in the image.
  if( !in_img(x, y, w, h) )
    return;

  // Load the 48 neighbours and myself.
  int n[7][7];
  for( int j = -3 ; j <= 3 ; ++j )
    for( int i = -3 ; i <= 3 ; ++i )
      n[j+3][i+3] = in_img(x+i, y+j, w, h) ? (int) src[(y+j)*w + (x+i)] : 0;

  // Compute the convolution.
  int p = 0;
  for( int j = 0 ; j < 7 ; ++j )
    for( int i = 0 ; i < 7 ; ++i )
      p += gaussian_filter[j][i] * n[j][i];

  // Store the result.
  dst[y*w + x] = (uchar) (p / 256);
}

// ==========================================================================================

#if defined(__CUDA_ARCH__) && __CUDA_ARCH__ >= 300
__global__ __launch_bounds__(128, 10)
#elif defined(__CUDA_ARCH__) && __CUDA_ARCH__ >= 200
__global__ __launch_bounds__(128, 8)
#else
```

Profiler is telling that we could use only 1 transaction but are using 4/5 (only 1 transaction required for each thread in warp to read a single byte char)

1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Memory Bandwidth

Memory bandwidth limits the performance of a kernel when one or more memories in the GPU cannot provide data at the rate requested by the kernel. The results at right indicate that the kernel is limited by the bandwidth available to the L2 cache.

If you modify the kernel you need to rerun your application to update this analysis.

Results

Global Memory Alignment and Access Pattern

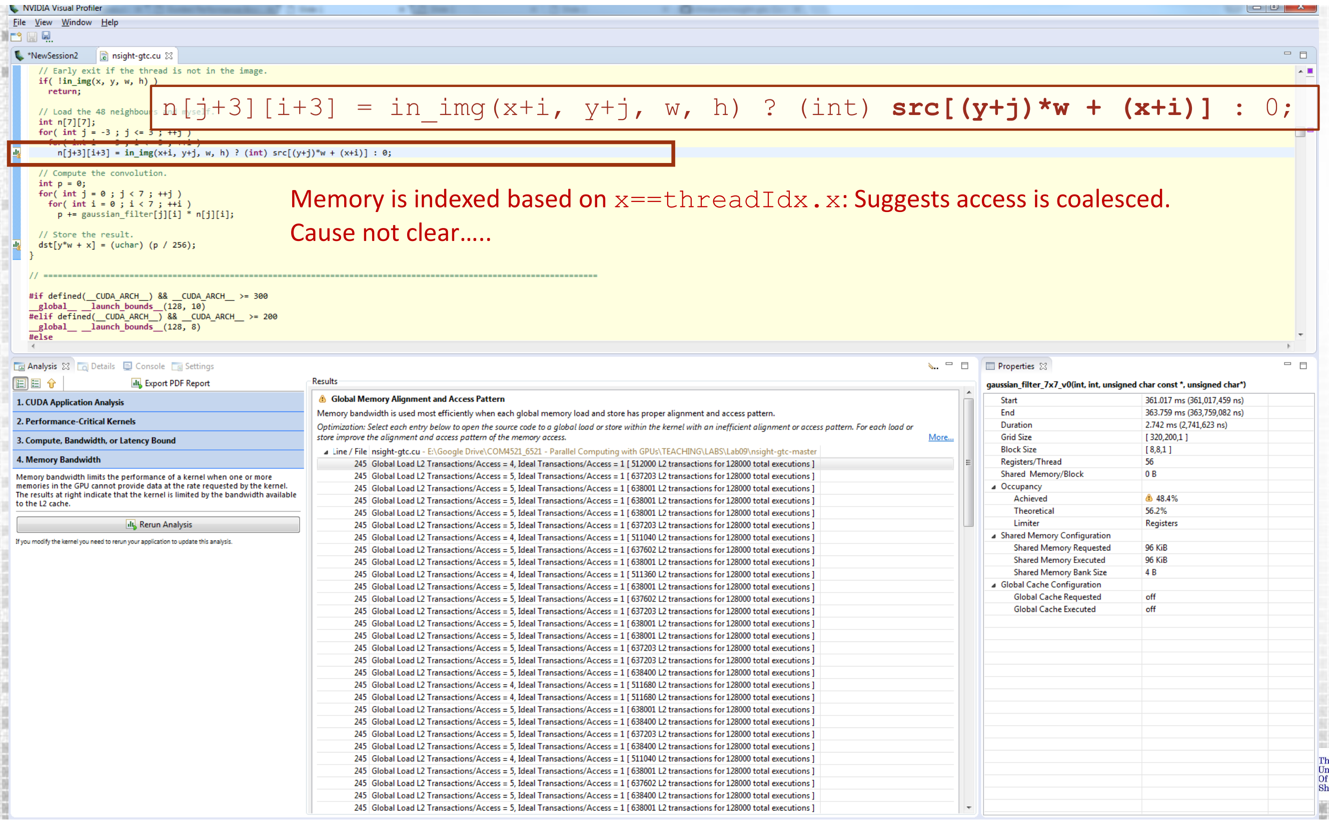
Memory bandwidth is used most efficiently when each global memory load and store has proper alignment and access pattern.

Optimization: Select each entry below to open the source code to a global 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...

```
245  Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 512000 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511040 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637602 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511360 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637602 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511680 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511680 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637203 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 4, Ideal Transactions/Access = 1 [ 511040 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 637602 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638400 L2 transactions for 128000 total executions ]
245  Global Load L2 Transactions/Access = 5, Ideal Transactions/Access = 1 [ 638001 L2 transactions for 128000 total executions ]
```

gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)

Start	361.017 ms (361,017,459 ns)
End	363.759 ms (363,759,082 ns)
Duration	2.742 ms (2,741,623 ns)
Grid Size	[320,200,1]
Block Size	[8,8,1]
Registers/Thread	56
Shared Memory/Block	0 B
Occupancy	
Achieved	48.4%
Theoretical	56.2%
Limiter	Registers
Shared Memory Configuration	
Shared Memory Requested	96 KiB
Shared Memory Executed	96 KiB
Shared Memory Bank Size	4 B
Global Cache Configuration	
Global Cache Requested	off
Global Cache Executed	off



`n[j+3][i+3] = in_img(x+i, y+j, w, h) ? (int) src[(y+j)*w + (x+i)] : 0;`

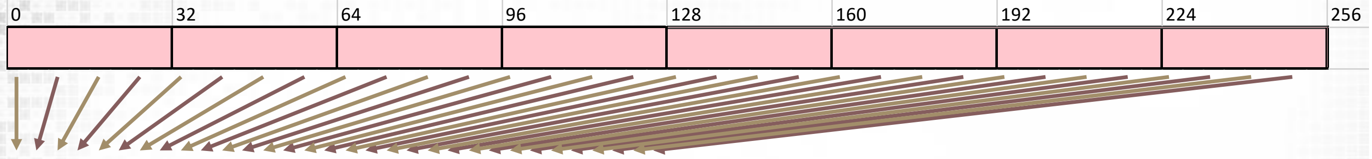
Memory is indexed based on `x==threadIdx.x`: Suggests access is coalesced. Cause not clear.....

Line / File nsight-gtc.cu - E:\Google Drive\COM4521_6521 - Parallel Computing with GPUs\TEACHING\LABS\Lab09\nsight-gtc-master
The same source code, analysis panel, Global Memory Alignment and Access Pattern results (lines 245, L2 Transactions/Access = 4 or 5, Ideal Transactions/Access = 1) and Properties for gaussian_filter_7x7_v0 as in the screenshot above.

Analysis

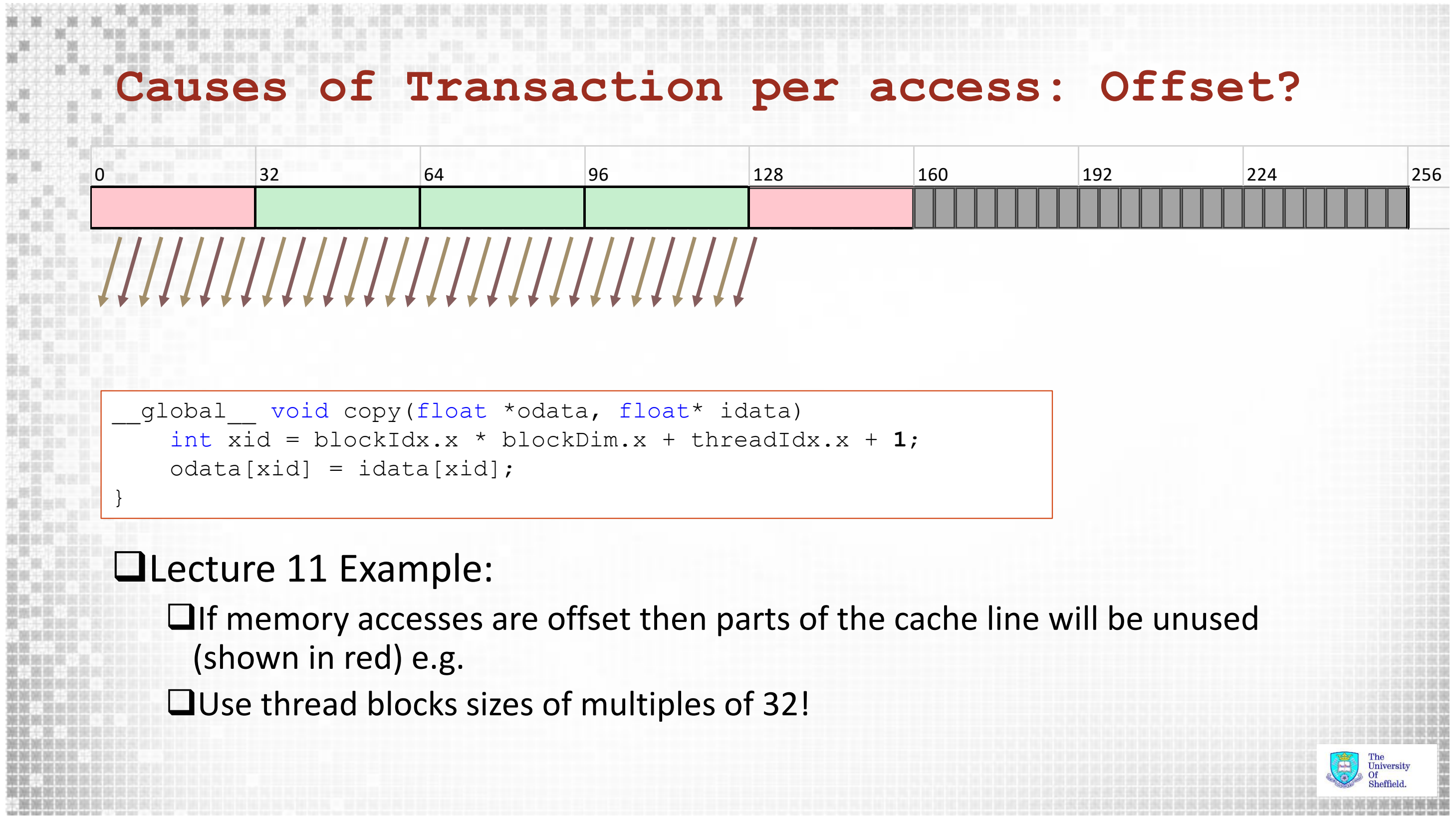
- ❑The limiting factor of our code is L2 Throughput
  - ❑There is nothing wrong with having high throughput
  - ❑Except: There is not enough compute to hide this
  - ❑We cant increase occupancy any further to hide this
- ❑Solution: We need to reduce the time it takes to get data to the device to do compute on it. Either by
  - ❑Moving data closer to the SMPs
  - ❑**Making our L2 reads/writes more efficient**
    - ❑Currently ~4-5 Transactions/Access
    - ❑Our L2 cache lines are being used ineffectively

Causes of Transaction per access: Striding?



```
__global__ void copy(float *odata, float* idata)
    int xid = (blockIdx.x * blockDim.x + threadIdx.x)* 2;
    odata[xid] = idata[xid];
}
```

- ❑Lecture 11 example
  - ❑Strides (like above) cause poor transactions per access
  - ❑In the above case 8 transactions where we could have used 4

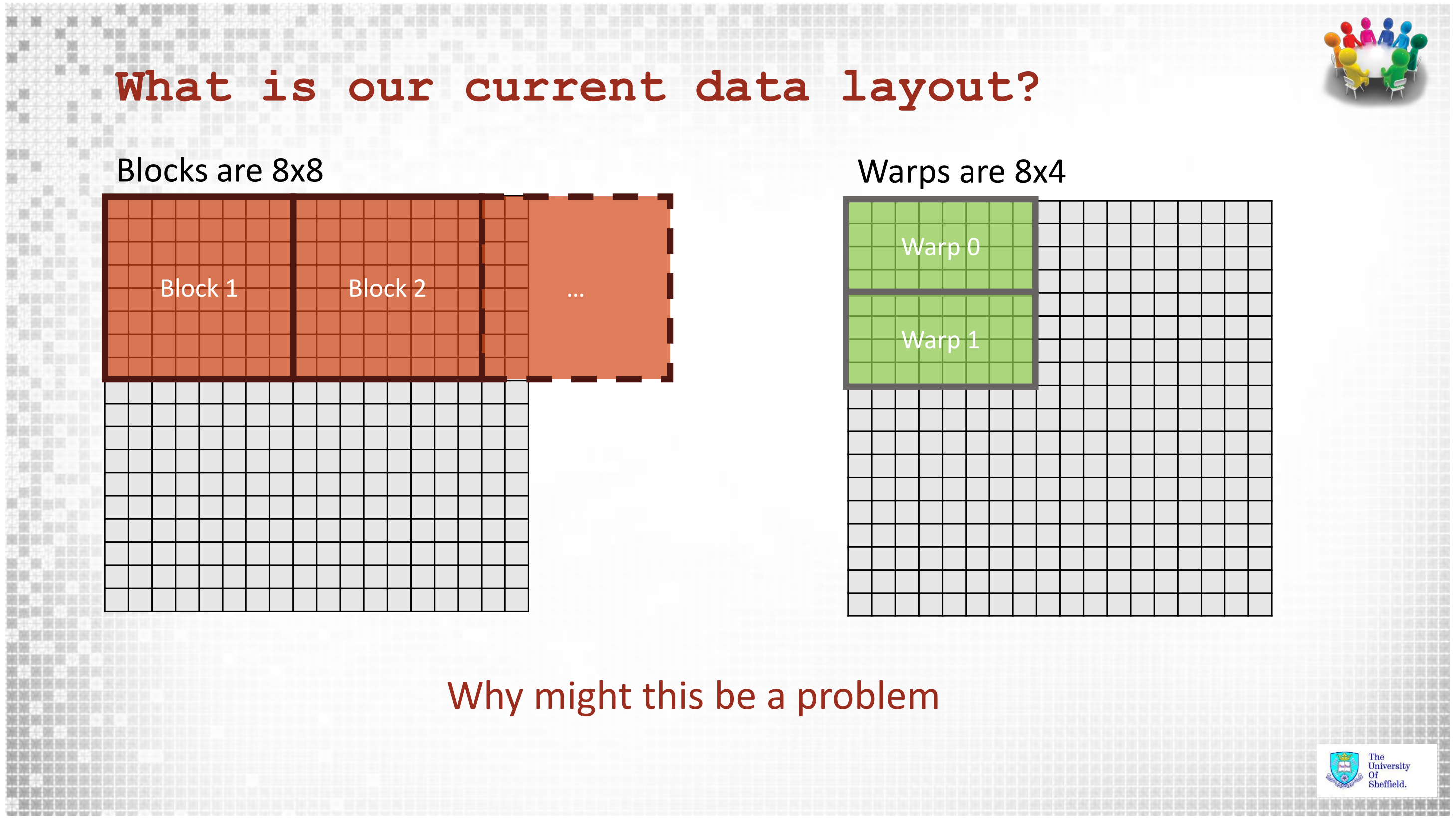


Causes of Transaction per access: Offset?

```
__global__ void copy(float *odata, float* idata)
    int xid = blockIdx.x * blockDim.x + threadIdx.x + 1;
    odata[xid] = idata[xid];
}
```

❑Lecture 11 Example:

- ❑If memory accesses are offset then parts of the cache line will be unused (shown in red) e.g.
- ❑Use thread blocks sizes of multiples of 32!

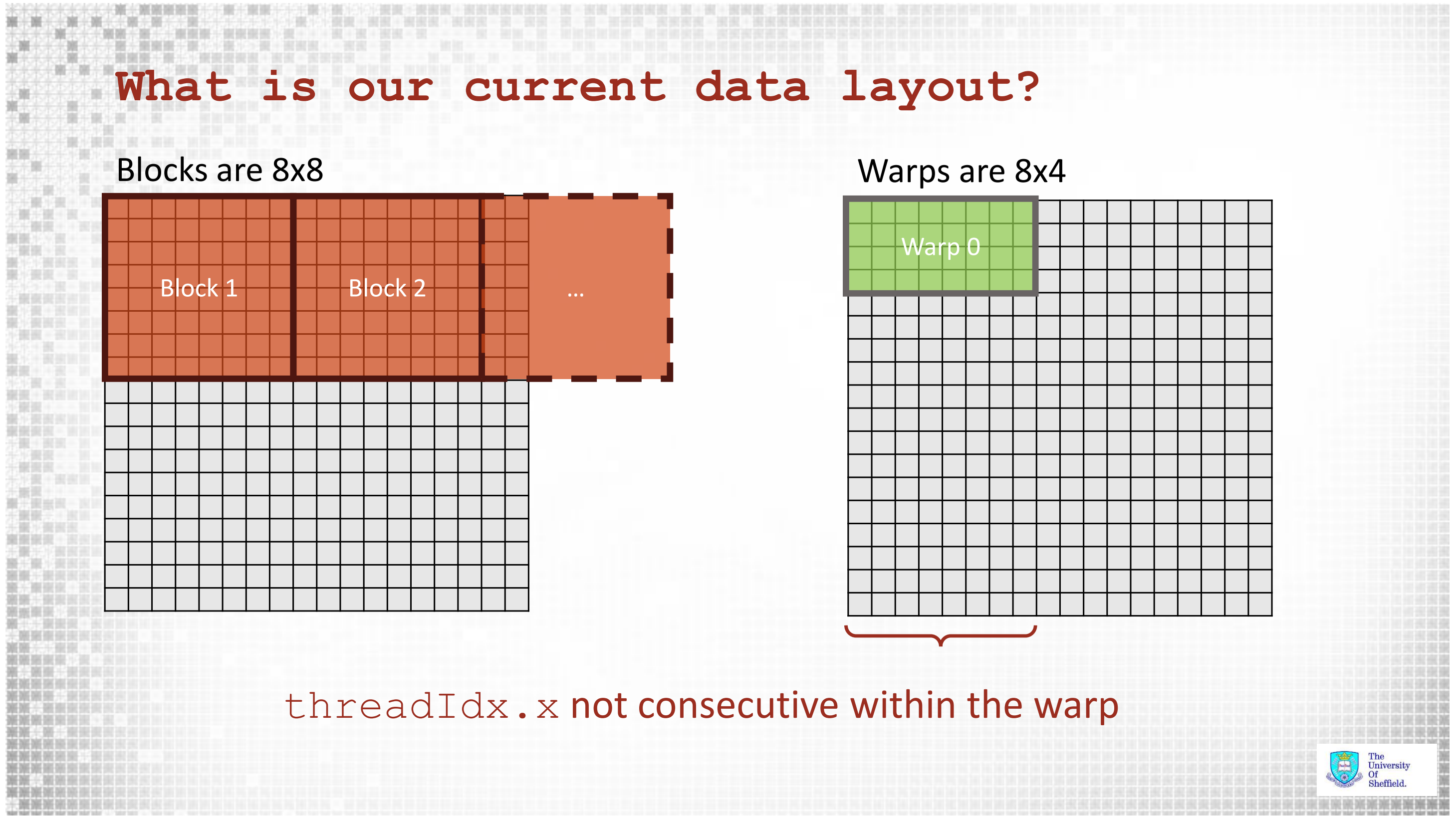


What is our current data layout?

Blocks are 8x8

Warps are 8x4

Why might this be a problem

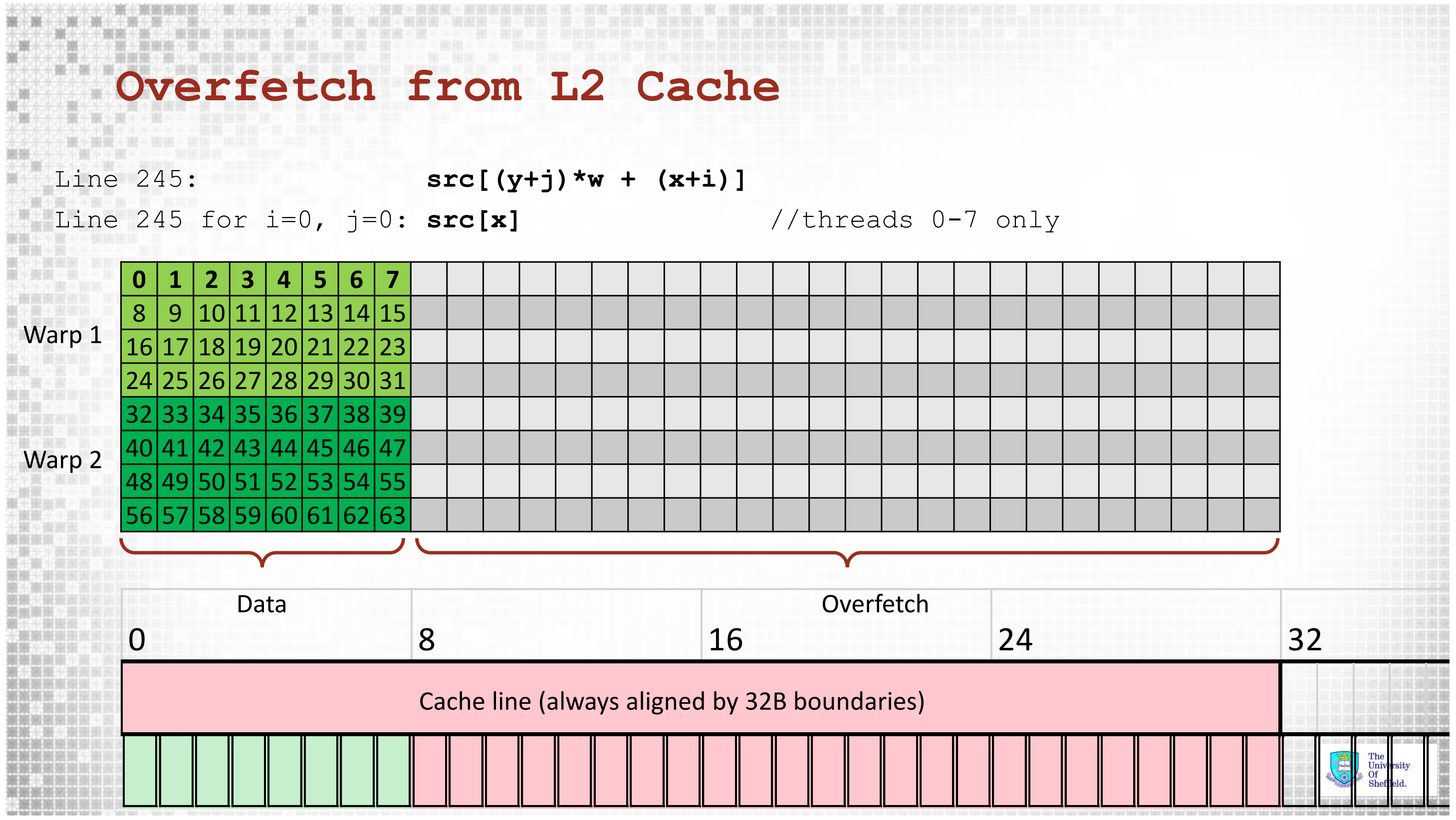


What is our current data layout?

Blocks are 8x8

Warps are 8x4

threadIdx.x not consecutive within the warp



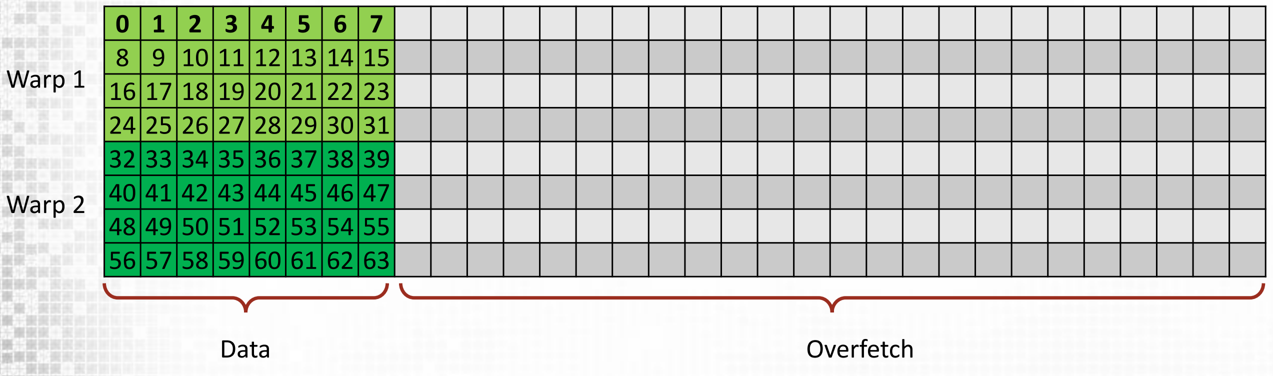
Overfetch from L2 Cache

```
Line 245:                 src[(y+j)*w + (x+i)]
Line 245 for i=0, j=0: src[x]              //threads 0-7 only
```

	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
Warp 1	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
	32	33	34	35	36	37	38	39
Warp 2	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

Overfetch with L1 Caching

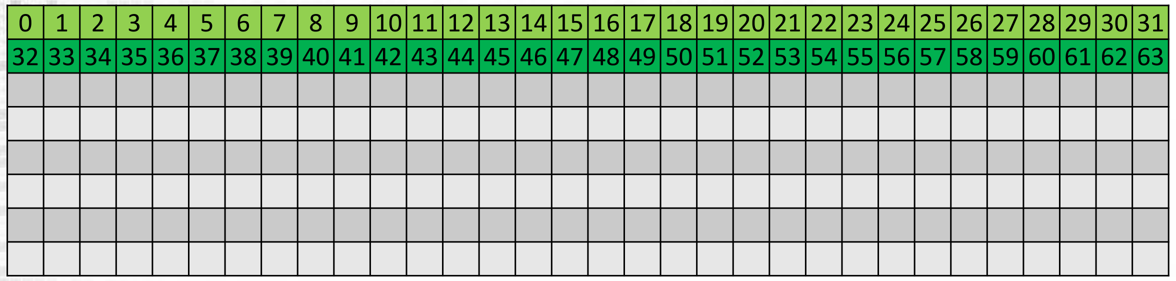
```
Line 245:              src[(y+j)*w + (x+i)]
Line 245 for i=0, j=0: src[y*w + x]
```



Warp 1	0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14	15
	16	17	18	19	20	21	22	23
	24	25	26	27	28	29	30	31
Warp 2	32	33	34	35	36	37	38	39
	40	41	42	43	44	45	46	47
	48	49	50	51	52	53	54	55
	56	57	58	59	60	61	62	63

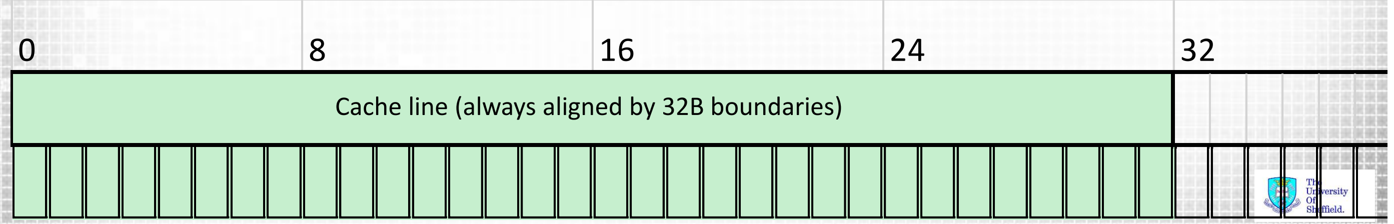
Any Ideas for improving this?

Optimisation: Improved Memory layout



0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

- ❑Minimum block width should be 32 (each thread requires only 1 byte)
- ❑Use Layout of 32x2



Deploy: Improved Memory layout

Kernel	Time (ms)	Speedup	Rel. Speedup
Gaussian_filter (Step 0)	5.49	1.00x	-
Gaussian_filter (Step 1a)	**1.00**	**5.49x**	**5.49x**

Break

- ❑What do we expect the analysis to look like next?
- ❑Any ideas for what else may be required?

Half time summary

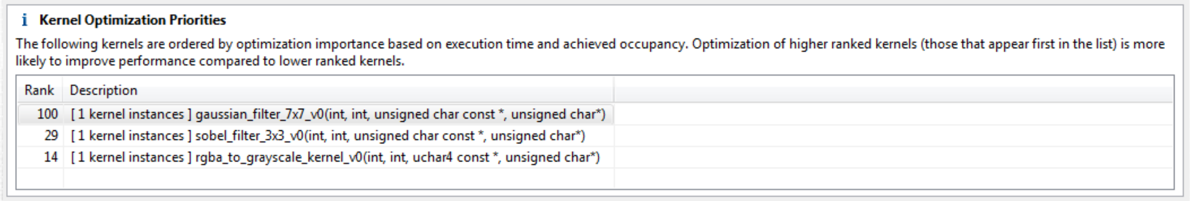
- ❑The guided profiler will help us optimise the right thing
- ❑Hotspot tells us the most appropriate place to optimise
- ❑Performance Limiter tells us what to focus on to improve
- ❑Code may be Memory, Compute or Latency Bound

- ❑Improvements so far
  - ❑Changed the access pattern (by changing block size)
  - ❑Reduced memory dependencies?

- ❑Profiling Introduction
- ❑The Problem
- ❑Visual Profiler Guided Analysis
  - ❑Iteration 1
  - ❑Iteration 2

Identify the hotspot

- ❑Examine GPU Usage in Visual Profiler
- ❑Examine Individual Kernels
  - ❑Gaussian filter kernel still the highest rank

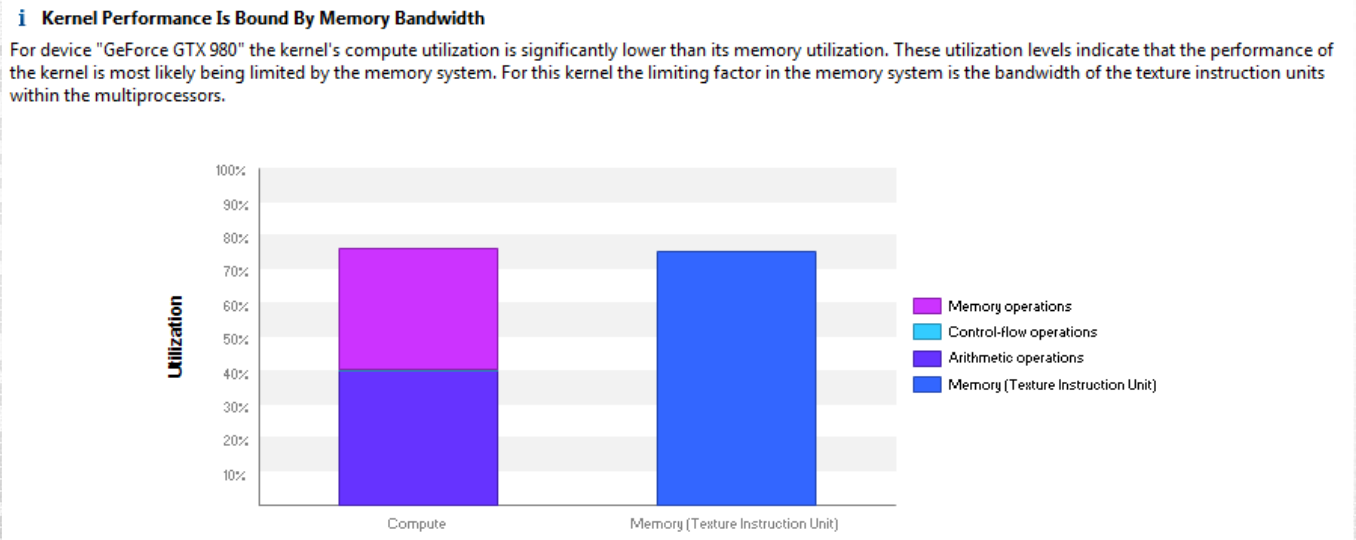


i Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] gaussian_filter_7x7_v0(int, int, unsigned char const *, unsigned char*)
29	[1 kernel instances] sobel_filter_3x3_v0(int, int, unsigned char const *, unsigned char*)
14	[1 kernel instances] rgba_to_grayscale_kernel_v0(int, int, uchar4 const *, unsigned char*)

Performance Limiter

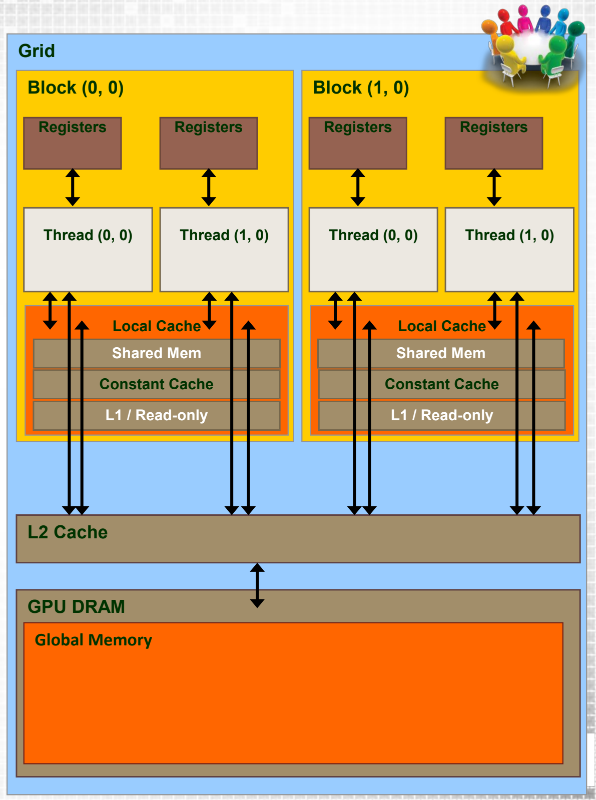


i Kernel Performance Is Bound By Memory Bandwidth

For device "GeForce GTX 980" 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 texture instruction units within the multiprocessors.

Tex Instruction Units?

❑What are texture instruction units and why might our code be using them?

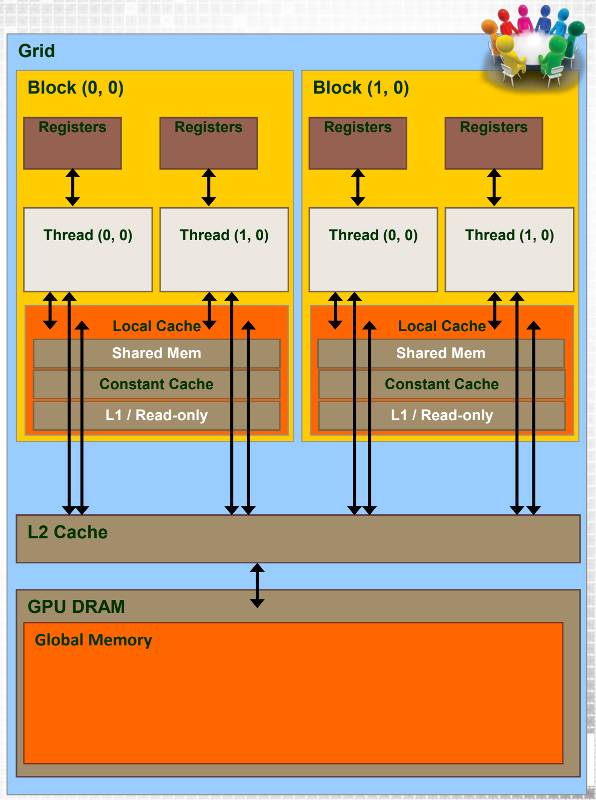


Tex Instruction Units?

❑What are texture instruction units and why might our code be using them?

❑Hint:

```
void gaussian_filter_7x7_v1(int w,
                            int h,
                            const uchar *src,
                            uchar *dst)
```

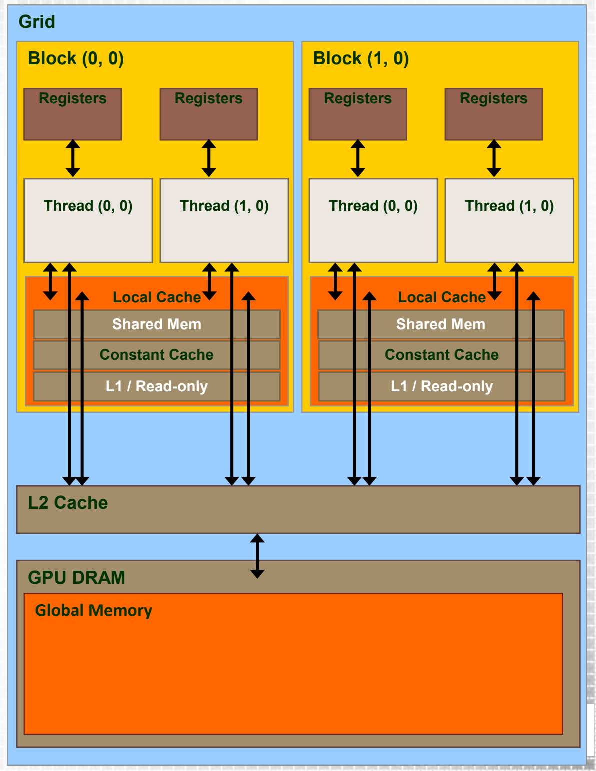


Tex Instruction Units?

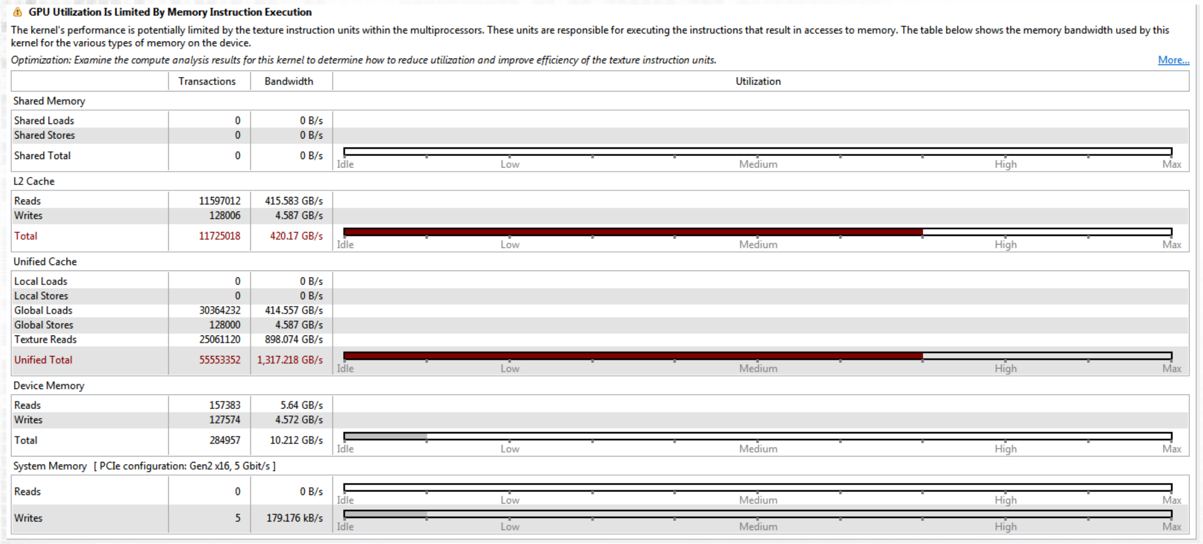
❑What are texture instruction units and why might out code be using them?

```
void gaussian_filter_7x7_v1(int w,
                            int h,
                            const uchar *src,
                            uchar *dst)
```

❑Compiler is reading `src` as read-only through Unified L1/Read-Only (texture cache)



Guided Bandwidth Analysis



⚠ **GPU Utilization Is Limited By Memory Instruction Execution**

The kernel's performance is potentially limited by the texture instruction units within the multiprocessors. These units are responsible for executing the instructions that result in accesses to memory. The table below shows the memory bandwidth used by this kernel for the various types of memory on the device.

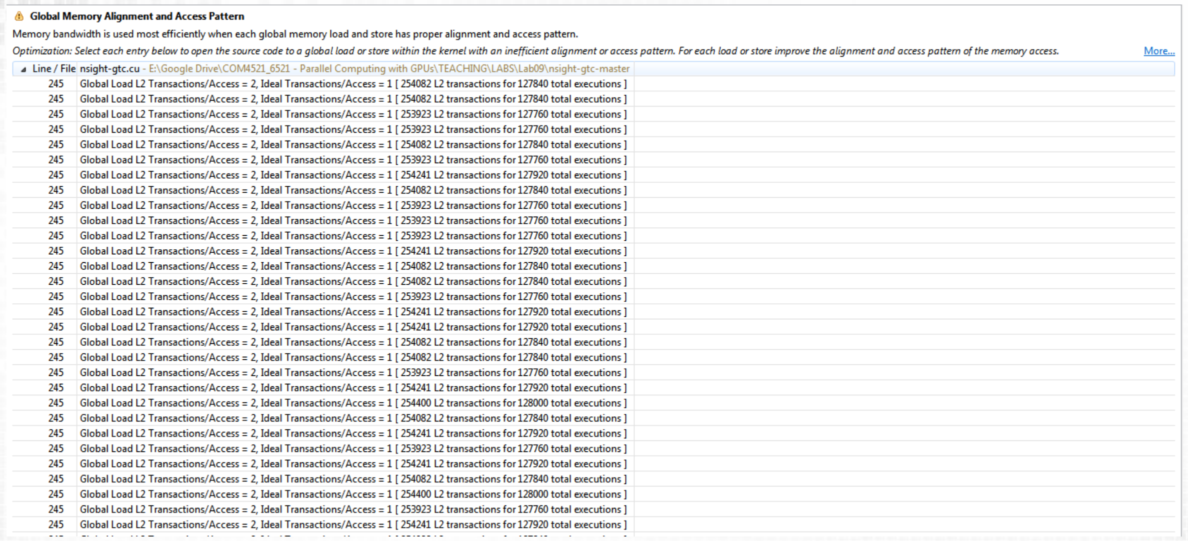
Optimization: Examine the compute analysis results for this kernel to determine how to reduce utilization and improve efficiency of the texture instruction units.

More...

	Transactions	Bandwidth	Utilization
Shared Memory			
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Shared Total	0	0 B/s	
L2 Cache			
Reads	11597012	415.583 GB/s	
Writes	128006	4.587 GB/s	
Total	11725018	420.17 GB/s	
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	30364232	414.557 GB/s	
Global Stores	128000	4.587 GB/s	
Texture Reads	25061120	898.074 GB/s	
Unified Total	55553352	1,317.218 GB/s	
Device Memory			
Reads	157383	5.64 GB/s	
Writes	127574	4.572 GB/s	
Total	284957	10.212 GB/s	
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	
Writes	5	179.176 kB/s	

❑We are doing lots of reading/writing through unified cache

Guided Bandwidth Analysis



Global Memory Alignment and Access Pattern

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

Optimization: Select each entry below to open the source code to a global 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	nsight-gtc.cu - E:\Google Drive\COM4521_6521 - Parallel Computing with GPUs\TEACHING\LABS\Lab09\nsight-gtc-master
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254400 L2 transactions for 128000 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254082 L2 transactions for 127840 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254400 L2 transactions for 128000 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [253923 L2 transactions for 127760 total executions]
245	Global Load L2 Transactions/Access = 2, Ideal Transactions/Access = 1 [254241 L2 transactions for 127920 total executions]

- ❑Still parts of the code reporting 2 transactions per access?

Transaction per request

```
Line 245:             src[(y+j)*w + (x+i)]
Line 245 for i=1, j=0: src[x+1]
```

What is wrong with this access pattern?

Transaction per request

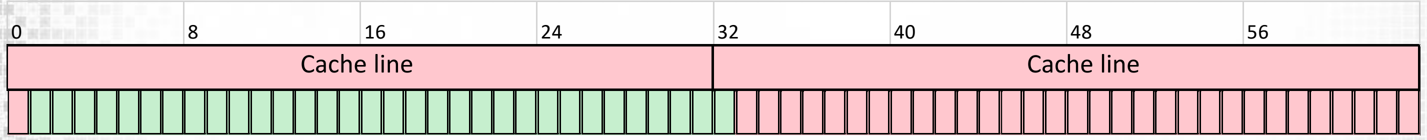
```
Line 245:             src[(y+j)*w + (x+i)]
Line 245 for i=1, j=0: src[x+1]
```

What is wrong with this access pattern?

Hint: Cache Lines are aligned by 32B boundaries

Transaction per request

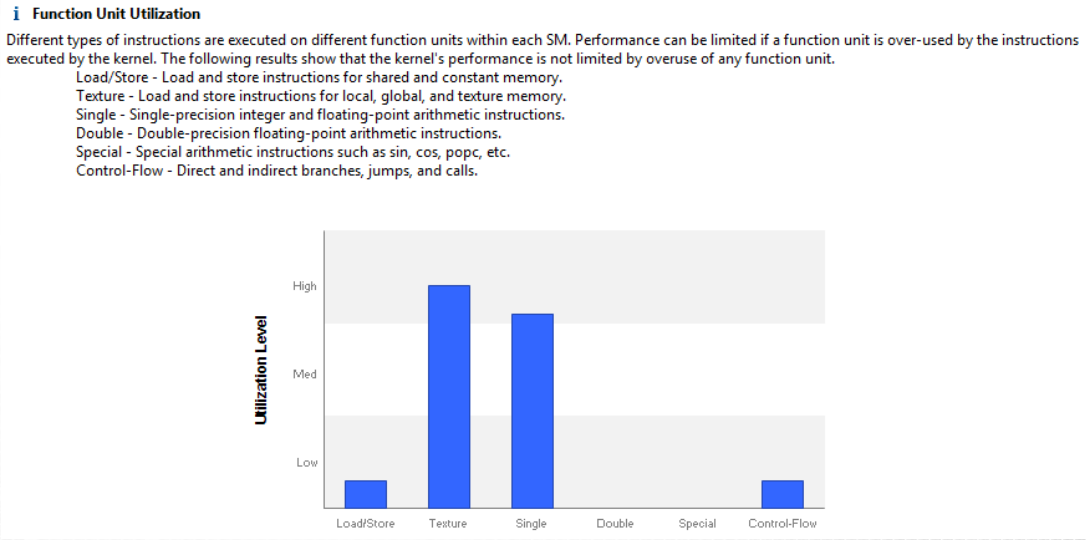
```
Line 245:             src[(y+j)*w + (x+i)]
Line 245 for i=1, j=0: src[x+1]
```



We have an offset access pattern

Guided Compute Analysis

- The guided analysis suggests that lots of our compute cycles are spent issuing texture load/stores

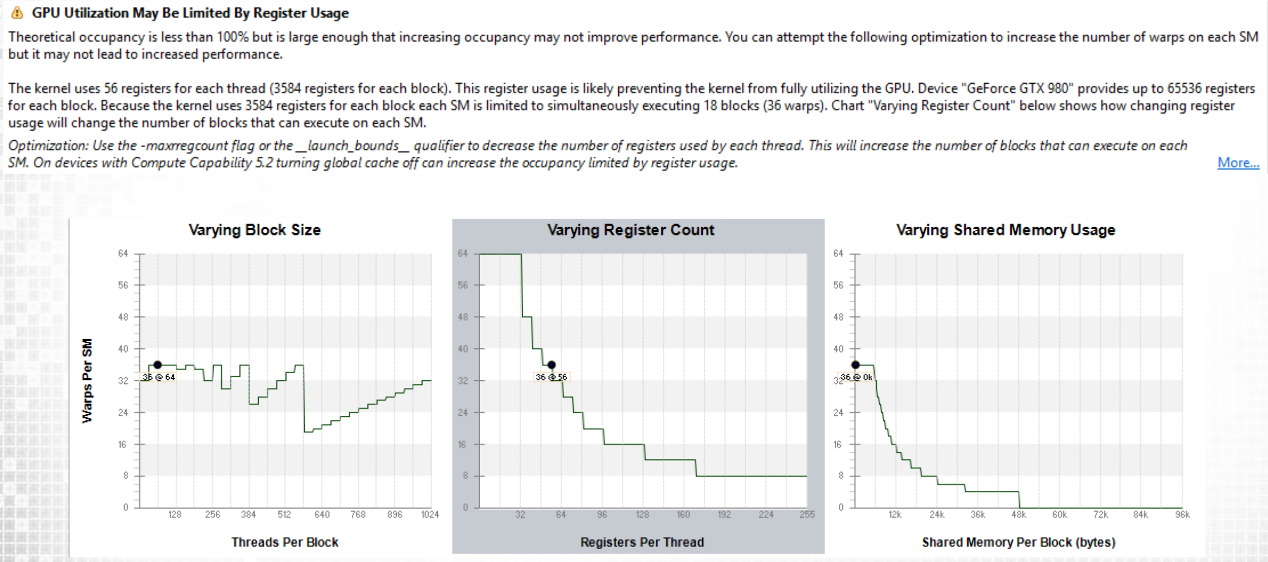


Function Unit Utilization

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 shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.
Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.

Guided Latency Analysis: Occupancy



GPU Utilization May Be Limited By Register Usage

Theoretical occupancy is less than 100% but is large enough that increasing occupancy may not improve performance. You can attempt the following optimization to increase the number of warps on each SM but it may not lead to increased performance.

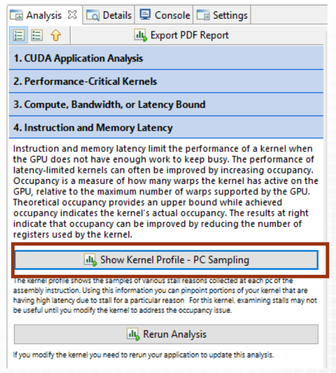
The kernel uses 56 registers for each thread (3584 registers for each block). This register usage is likely preventing the kernel from fully utilizing the GPU. Device "GeForce GTX 980" provides up to 65536 registers for each block. Because the kernel uses 3584 registers for each block each SM is limited to simultaneously executing 18 blocks (36 warps). Chart "Varying Register Count" below shows how changing register usage will change the number of blocks that can execute on each SM.

Optimization: Use the -maxrregcount flag or the __launch_bounds__ qualifier to decrease the number of registers used by each thread. This will increase the number of blocks that can execute on each SM. On devices with Compute Capability 5.2 turning global cache off can increase the occupancy limited by register usage. More...

Guided Latency Analysis: Occupancy

- Register usage is very high
- Occupancy currently limited by register usage

- Increasing occupancy might not help us however as we are dominated by texture load stores
  - More work per SMP will just mean even more texture load stores!
  - We can confirm this by looking at the unguided analysis: Kernel Latency



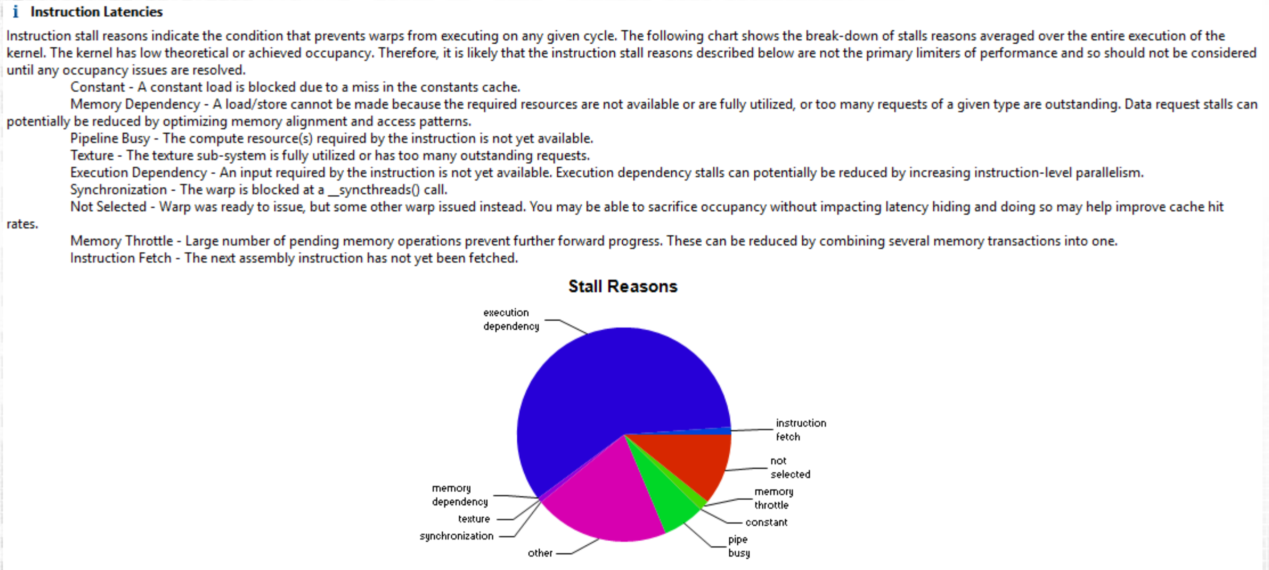
1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Instruction and Memory Latency

Instruction and memory latency limit the performance of a kernel when the GPU does not have enough work to keep busy. The performance of latency-limited kernels can often be improved by increasing occupancy. Occupancy is a measure of how many warps the kernel has active on the GPU, relative to the maximum number of warps supported by the GPU. Theoretical occupancy provides an upper bound while achieved occupancy indicates the kernel's actual occupancy. The results at right indicate that occupancy can be improved by reducing the number of registers used by the kernel.

The kernel profile shows the samples of various stall reasons collected at each pc of the assembly instruction. Using this information you can pinpoint portions of your kernel that are having high latency due to stall for a particular reason For this kernel, examining stalls may not be useful until you modify the kernel to address the occupancy issue.

If you modify the kernel you need to rerun your application to update this analysis.

PC Sampling



Instruction Latencies

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 low theoretical or achieved occupancy. Therefore, it is likely that the instruction stall reasons described below are not the primary limiters of performance and so should not be considered until any occupancy issues are resolved.

Constant - A constant load is blocked due to a miss in the constants cache.
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.
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.
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.
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.
Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.
Instruction Fetch - The next assembly instruction has not yet been fetched.

Execution/Memory Dependency

- ❑Rank these are best to worst
- ❑Which have instruction and memory dependencies?

```
int a = b + c;

int d = a + e;

//b, c and e are local ints
```

```
int a = b[i];

int d = a + e;

//b is global memory
//I and e are local ints
```

```
int a = b + c;

int d = e + f;

//b, c, e and f are local ints
```

Instruction/Memory Dependency

- ❑Rank these are best to worst
- ❑Which have instruction and memory dependencies?

🟨

```
int a = b + c;

int d = a + e;

//b, c and e are local ints
```

- ❑ Instruction Dependency
- ❑ Second add must wait for first

🟥

```
int a = b[i];

int d = a + e;

//b is global memory
//i and e are local ints
```

- ❑ Memory Dependency
- ❑ Second add must wait for memory request

🟩

```
int a = b + c;

int d = e + f;

//b, c, e and f are local ints
```

- ❑ No dependencies
- ❑ Independent Adds

Analysis

- ❑Our compute engine is dominated by load/store instructions for the texture cache
  - ❑Our texture bandwidth is good BUT
- ❑Our warps are stalling as instructions are waiting to issue texture fetch instructions
- ❑We still have poorly aligned access pattern within our inner loops

- ❑Solution: Reduce dependencies on texture loads
  - ❑Move data closer to the SMP
  - ❑Only read from global memory with nicely aligned cache lines
  - ❑**How?**

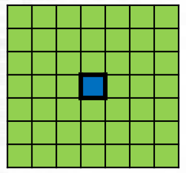
Analysis

- ❑Our compute engine is dominated by load/store instructions for the texture cache
  - ❑Our texture bandwidth is good BUT
- ❑Our warps are stalling as instructions are waiting to issue texture fetch instructions
- ❑We still have poorly aligned access pattern within our inner loops

- ❑Solution: Reduce dependencies on texture loads
  - ❑Move data closer to the SMP
  - ❑Only read from global memory with nicely aligned cache lines
  - ❑**Shared Memory**

Shared Memory

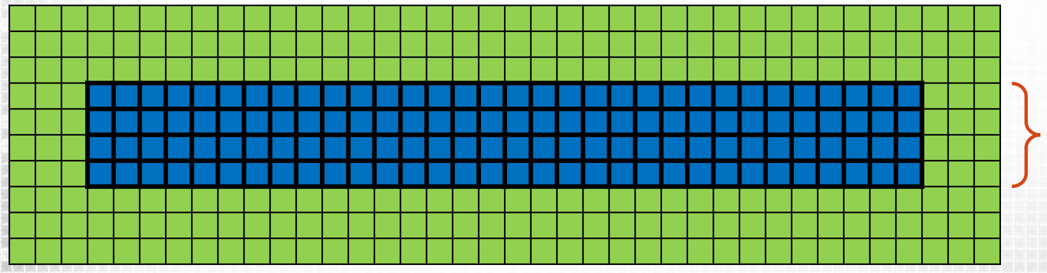
Single thread uses 7x7= 42 values



Use shared memory to store all pixels for the block

What important factor should we be considering?

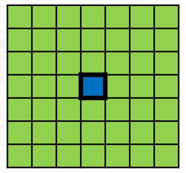
Single block (32x4) uses 38x10 = 680 values



Also increased Block size

Shared Memory

Single thread uses 7x7= 42 values

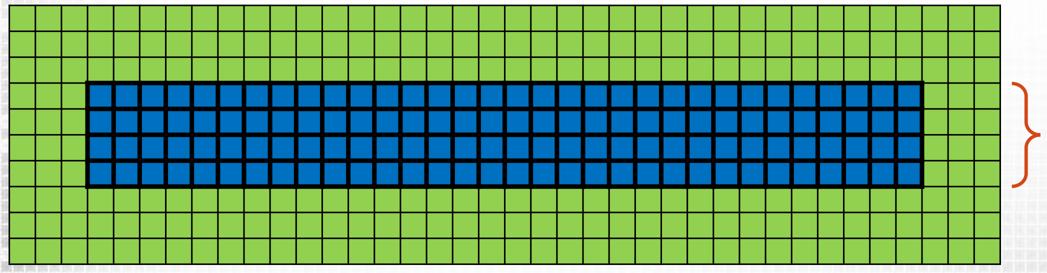


Use shared memory to store all pixels for the block

```
__shared__ unsigned char smem_pixels[10][64]
```

SM bank conflicts

Single block (32x4) uses 38x10 = 680 values

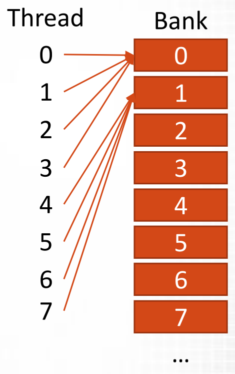


Also increased Block size

BUT WAIT!!!!!!!!!!!!!!

- ❑Wouldn’t aligned char access have 4 way bank conflicts?
  - ❑NOT for Compute Mode 2.0+…

> *“A shared memory request for a warp does not generate a bank conflict between two threads that access any address within the same 32-bit word (even though the two addresses fall in the same bank): In that case, for read accesses, the word is **broadcast** to the requesting threads (multiple words can be broadcast in a single transaction) …”*



I.e. A Stride of less than 1 (4B word) can be read conflict free if threads access aligned data

Improvement

- ❑Significant

Kernel	Time (ms)	Speedup	Rel. Speedup
Gaussian_filter (Step 0)	5.49	1.00x	-
Gaussian_filter (Step 1a)	1.00	5.49x	5.49x
Gaussian_filter (Step 40)	**0.49**	**11.20x**	**2.04x**

❑Profiling Introduction

❑The Problem

❑Visual Profiler Guided Analysis

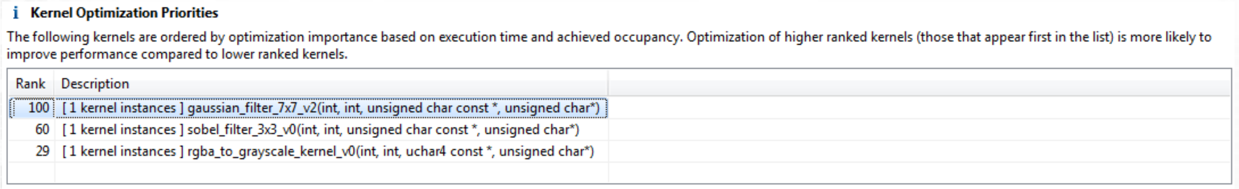
- ❑Iteration 1
- ❑Iteration 2
- ❑Iteration 3

Identify the hotspot

❑Examine GPU Usage in Visual Profiler

❑Examine Individual Kernels

- ❑Gaussian filter kernel still the highest rank
  - ❑Getting much closer though

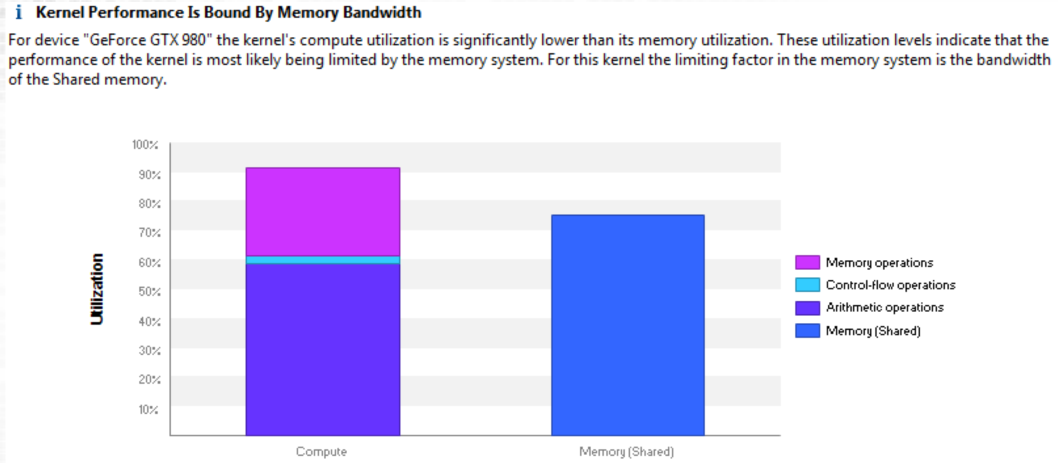


Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] gaussian_filter_7x7_v2(int, int, unsigned char const *, unsigned char*)
60	[1 kernel instances] sobel_filter_3x3_v0(int, int, unsigned char const *, unsigned char*)
29	[1 kernel instances] rgba_to_grayscale_kernel_v0(int, int, uchar4 const *, unsigned char*)

Performance Limiter



Kernel Performance Is Bound By Memory Bandwidth

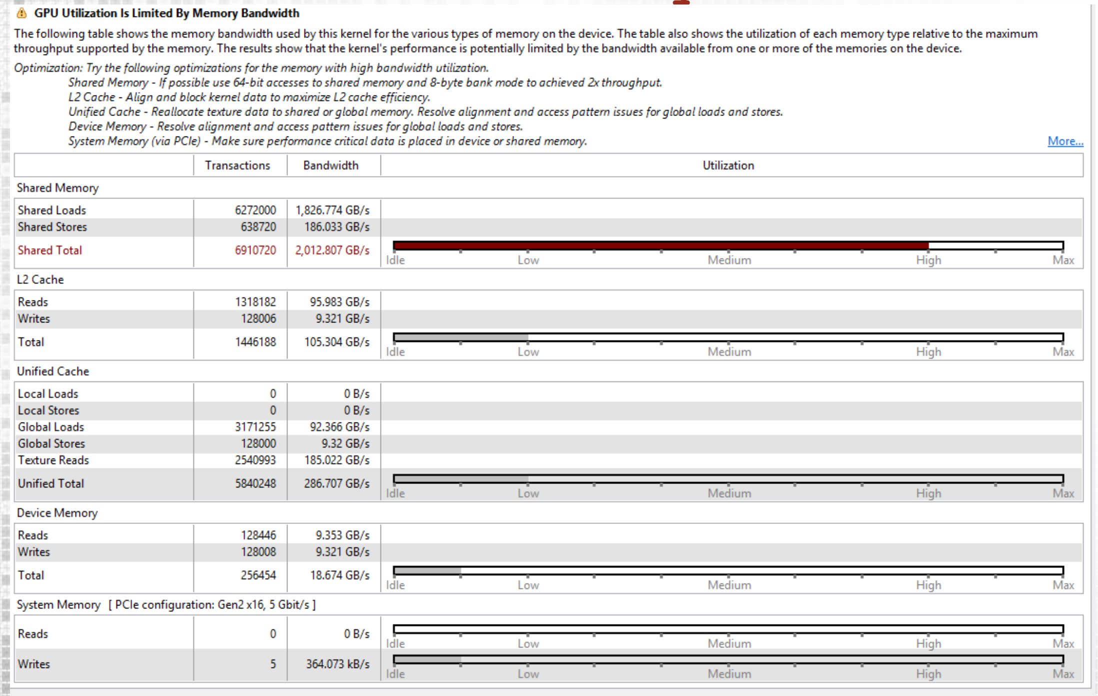
For device "GeForce GTX 980" the kernel's compute utilization is significantly lower than its memory utilization. These utilization levels indicate that the performance of the kernel is most likely being limited by the memory system. For this kernel the limiting factor in the memory system is the bandwidth of the Shared memory.

❑Actually very close to magical 60% of compute

❑Lets examine

- ❑1) The compute analysis
- ❑2) The latency analysis

Guided Bandwidth Analysis



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.

Shared Memory - If possible use 64-bit accesses to shared memory and 8-byte bank mode to achieved 2x throughput.
L2 Cache - Align and block kernel data to maximize L2 cache efficiency.
Unified Cache - Reallocate texture data to shared or global memory. Resolve alignment and access pattern issues for global loads and stores.
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
Shared Memory			
Shared Loads	6272000	1,826.774 GB/s	
Shared Stores	638720	186.033 GB/s	
Shared Total	6910720	2,012.807 GB/s	Idle Low Medium High Max
L2 Cache			
Reads	1318182	95.983 GB/s	
Writes	128006	9.321 GB/s	
Total	1446188	105.304 GB/s	Idle Low Medium High Max
Unified Cache			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Global Loads	3171255	92.366 GB/s	
Global Stores	128000	9.32 GB/s	
Texture Reads	2540993	185.022 GB/s	
Unified Total	5840248	286.707 GB/s	Idle Low Medium High Max
Device Memory			
Reads	128446	9.353 GB/s	
Writes	128008	9.321 GB/s	
Total	256454	18.674 GB/s	Idle Low Medium High Max
System Memory [PCIe configuration: Gen2 x16, 5 Gbit/s]			
Reads	0	0 B/s	Idle Low Medium High Max
Writes	5	364.073 kB/s	Idle Low Medium High Max

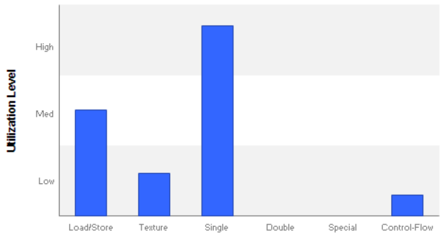
Compute Analysis

- ❑ We are simply doing lots of compute
- ❑ Additional floating point operations graph shows <u>no activity </u>i.e. all of our instructions are Integer

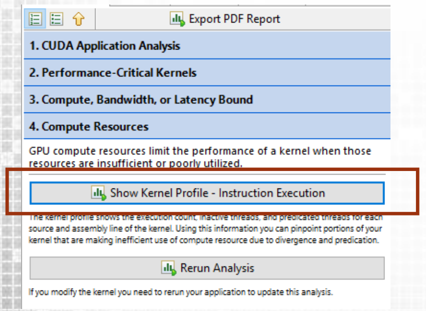
Function Unit Utilization

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 shared and constant memory.
Texture - Load and store instructions for local, global, and texture memory.
Single - Single-precision integer and floating-point arithmetic instructions.
Double - Double-precision floating-point arithmetic instructions.
Special - Special arithmetic instructions such as sin, cos, popc, etc.
Control-Flow - Direct and indirect branches, jumps, and calls.



What are all of these integer instructions?



1. CUDA Application Analysis
2. Performance-Critical Kernels
3. Compute, Bandwidth, or Latency Bound
4. Compute Resources

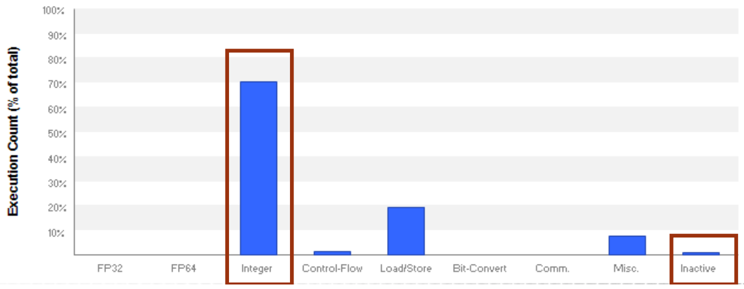
GPU compute resources limit the performance of a kernel when those resources are insufficient or poorly utilized.

The kernel profile shows the execution count, inactive threads, and predicated threads for each source and assembly line of the kernel. Using this information you can pinpoint portions of your kernel that are making inefficient use of compute resource due to divergence and predication.

If you modify the kernel you need to rerun your application to update this analysis.

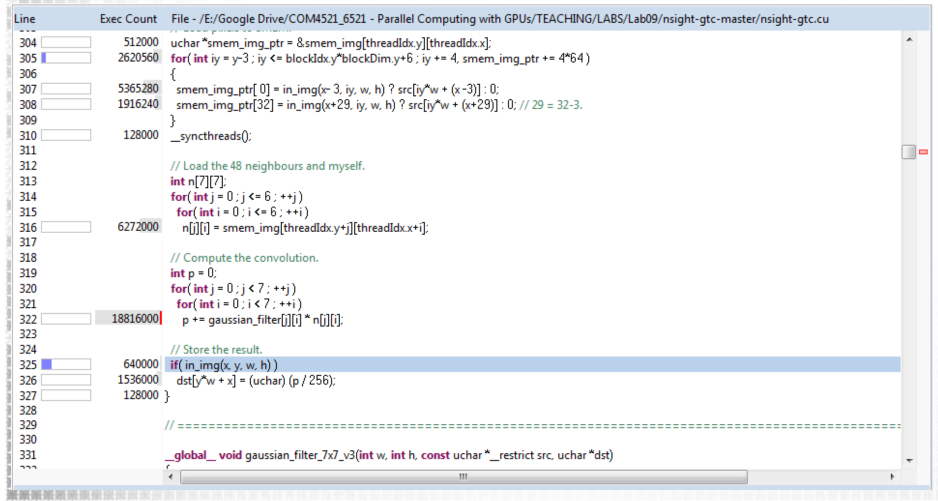
Instruction Execution Counts

The following chart shows the mix of instructions executed by the kernel. The instructions are grouped into classes and for each class the chart shows the percentage of thread execution cycles that were devoted to executing instructions in that class. The "Inactive" result shows the thread executions that did not execute any instruction because the thread was predicated or inactive due to divergence.



Compute Analysis by Line

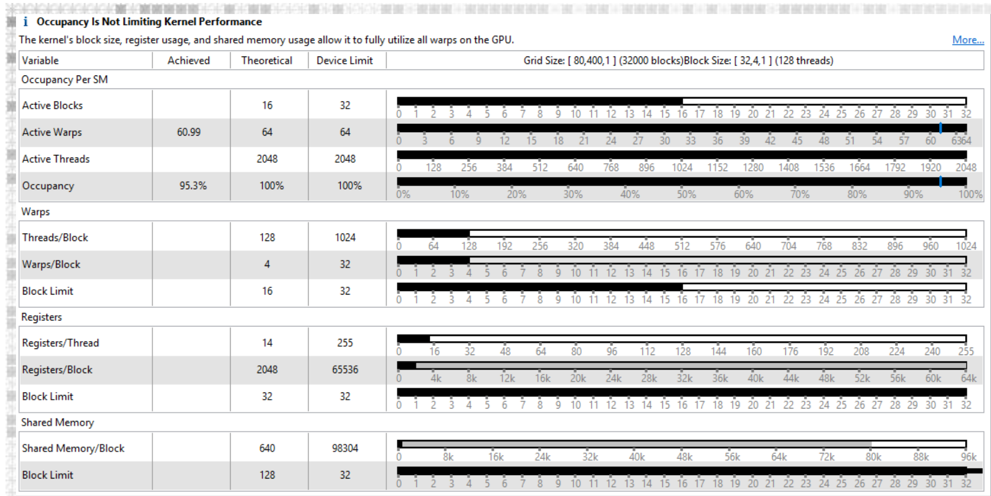
- ❑Selecting the CUDA function from compute analysis results allows a line by line breakdown
  - ❑This will switch to unguided analysis



```
Line     Exec Count   File - /E:/Google Drive/COM4521_6521 - Parallel Computing with GPUs/TEACHING/LABS/Lab09/nsight-gtc-master/nsight-gtc.cu
304        512000     uchar *smem_img_ptr = &smem_img[threadIdx.y][threadIdx.x];
305       2620560     for( int iy = y-3 ; iy <= blockIdx.y*blockDim.y+6 ; iy += 4, smem_img_ptr += 4*64 )
306                   {
307       5365280       smem_img_ptr[ 0] = in_img(x- 3, iy, w, h) ? src[iy*w + (x -3)] : 0;
308       1916240       smem_img_ptr[32] = in_img(x+29, iy, w, h) ? src[iy*w + (x+29)] : 0; // 29 = 32-3.
309                   }
310        128000     __syncthreads();
311
312                   // Load the 48 neighbours and myself.
313                   int n[7][7];
314                   for( int j = 0 ; j <= 6 ; ++j )
315                     for( int i = 0 ; i <= 6 ; ++i )
316       6272000         n[j][i] = smem_img[threadIdx.y+j][threadIdx.x+i];
317
318                   // Compute the convolution.
319                   int p = 0;
320                   for( int j = 0 ; j < 7 ; ++j )
321                     for( int i = 0 ; i < 7 ; ++i )
322      18816000         p += gaussian_filter[j][i] * n[j][i];
323
324                   // Store the result.
325        640000     if( in_img(x, y, w, h) )
326       1536000       dst[y*w + x] = (uchar) (p / 256);
327        128000     }
328
329                   // ==========================================================================================
330
331                   __global__ void gaussian_filter_7x7_v3(int w, int h, const uchar *__restrict src, uchar *dst)
```

Also PTX instruction breakdown provided

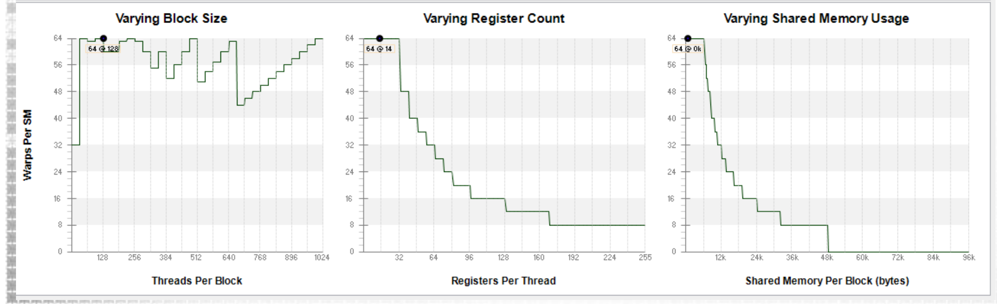
Guided Latency Analysis



Occupancy Is Not Limiting Kernel Performance

The kernel's block size, register usage, and shared memory usage allow it to fully utilize all warps on the GPU. More...

Variable	Achieved	Theoretical	Device Limit	Grid Size: [80,400,1] (32000 blocks)Block Size: [32,4,1] (128 threads)
Occupancy Per SM				
Active Blocks		16	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Active Warps	60.99	64	64	0 3 6 9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60 6364
Active Threads		2048	2048	0 128 256 384 512 640 768 896 1024 1152 1280 1408 1536 1664 1792 1920 2048
Occupancy	95.3%	100%	100%	0% 10% 20% 30% 40% 50% 60% 70% 80% 90% 100%
Warps				
Threads/Block		128	1024	0 64 128 192 256 320 384 448 512 576 640 704 768 832 896 960 1024
Warps/Block		4	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Block Limit		16	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Registers				
Registers/Thread		14	255	0 16 32 48 64 80 96 112 128 144 160 176 192 208 224 240 255
Registers/Block		2048	65536	0 4k 8k 12k 16k 20k 24k 28k 32k 36k 40k 44k 48k 52k 56k 60k 64k
Block Limit		32	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
Shared Memory				
Shared Memory/Block		640	98304	0 8k 16k 24k 32k 40k 48k 56k 64k 72k 80k 88k 96k
Block Limit		128	32	0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32



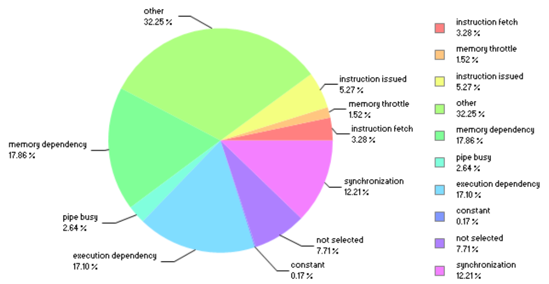
Would changing the block size, register usage or amount of shared memory per block improve occupancy?

Guided Latency Analysis

Source files :

file:/E:/Google%20Drive/COM4521_6521%20-%20Parallel%20Computing%20with%20GPUs/TEACHING/LABS/Lab09/nsight-gtc-master/nsight-gtc.cu

Sample distribution



🟥 instruction fetch	3.28%
🟧 memory throttle	1.52%
🟨 instruction issued	5.27%
🟩 other	32.25%
🟩 memory dependency	17.86%
🟩 pipe busy	2.64%
🟦 execution dependency	17.10%
🟦 constant	0.17%
🟪 not selected	7.71%
🟪 synchronization	12.21%

Line by Line Breakdown

The Kernel Profile - PC Sampling gives the number of samples for each source and assembly line with various stall reasons. Using this information you can pinpoint portions of your kernel that are introducing latencies and the reason for the latency. Samples are taken in round robin order for all active warps at a fixed number of cycles regardless of whether the warp is issuing an instruction or not.

Instruction Issued - Warp was issued
Instruction Fetch - The next assembly instruction has not yet been fetched.
Execution Dependency - An input required by the instruction is not yet available. Execution dependency stalls can potentially be reduced by increasing instruction-level parallelism.
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.
Texture - The texture sub-system is fully utilized or has too many outstanding requests.
Synchronization - The warp is blocked at a __syncthreads() call.
Constant - A constant load is blocked due to a miss in the constants cache.
Pipe Busy - The compute resource(s) required by the instruction is not yet available.
Memory Throttle - Large number of pending memory operations prevent further forward progress. These can be reduced by combining several memory transactions into one.
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.
Other - The warp is blocked for a uncommon reason.

More...

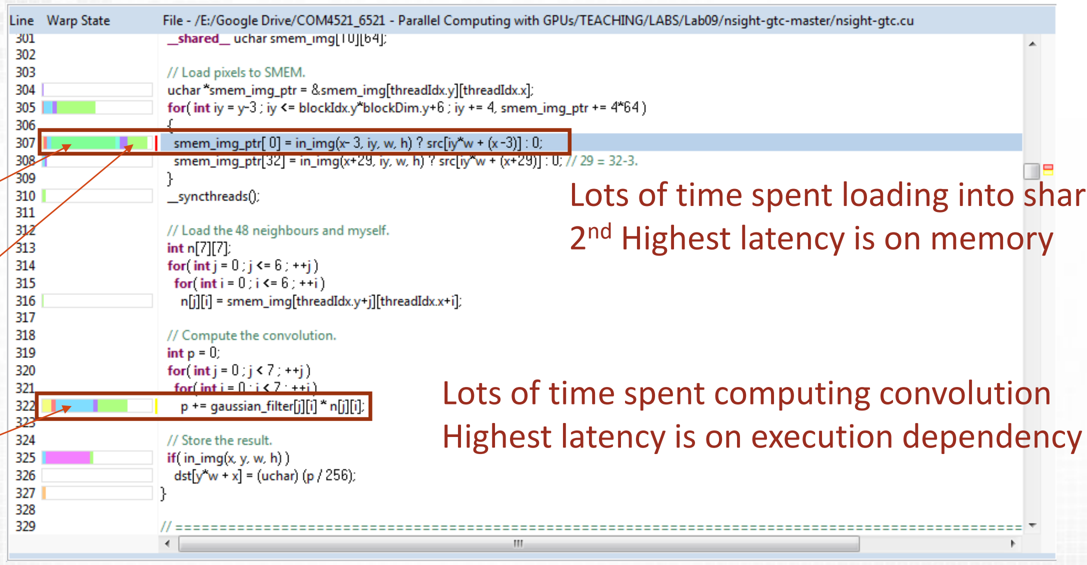
Latency Overview: Other 32.25%

❑Stall reason other generally means that there is no obvious action to improve performance

❑Other stall reasons may indicate either;

1. Execution unit is busy
   - ❑ Solution: Potentially reduce use of low throughput integer operations if possible
2. Register bank conflicts : a compiler issue that can sometimes be made worst by heavy use of vector data types
   - ❑ Solution: None
3. Too few warps per scheduler
   - ❑ Solution: Increase occupancy, decrease latency

Guided Latency Analysis: Line by Line



Line Warp State File - /E:/Google Drive/COM4521_6521 - Parallel Computing with GPUs/TEACHING/LABS/Lab09/nsight-gtc-master/nsight-gtc.cu

```
__shared__ uchar smem_img[10][64];

// Load pixels to SMEM.
uchar *smem_img_ptr = &smem_img[threadIdx.y][threadIdx.x];
for( int iy = y-3 ; iy <= blockIdx.y*blockDim.y+6 ; iy += 4, smem_img_ptr += 4*64 )
{
  smem_img_ptr[ 0] = in_img(x- 3, iy, w, h) ? src[iy*w + (x -3)] : 0;
  smem_img_ptr[32] = in_img(x+29, iy, w, h) ? src[iy*w + (x+29)] : 0; // 29 = 32-3.
}
__syncthreads();

// Load the 48 neighbours and myself.
int n[7][7];
for( int j = 0 ; j <= 6 ; ++j )
  for( int i = 0 ; i <= 6 ; ++i )
    n[j][i] = smem_img[threadIdx.y+j][threadIdx.x+i];

// Compute the convolution.
int p = 0;
for( int j = 0 ; j < 7 ; ++j )
  for( int i = 0 ; i < 7 ; ++i )
    p += gaussian_filter[j][i] * n[j][i];

// Store the result.
if( in_img(x, y, w, h) )
  dst[y*w + x] = (uchar) (p / 256);
}

// ==========================================================================
```

memory

other

execution

Lots of time spent loading into shared memory
2nd Highest latency is on memory

Lots of time spent computing convolution
Highest latency is on execution dependency

1st Analysis

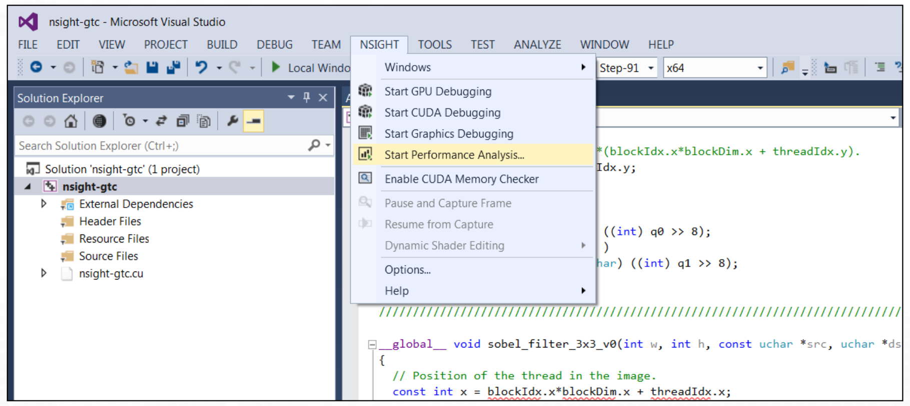
❑We have a reasonably well balanced use of the from Compute and Memory pipes.

❑There is some latency in loading data to and from shared memory

❑Our compute cycles are dominated by Integer operations

- ❑What operations are they?
- ❑We can either examine the code and PTX instructions (from Compute or Latency Analysis) or run additional analysis via Nsight within Visual Studio
  - ❑More detailed analysis
  - ❑Not guided like the visual profiler

Start profiling



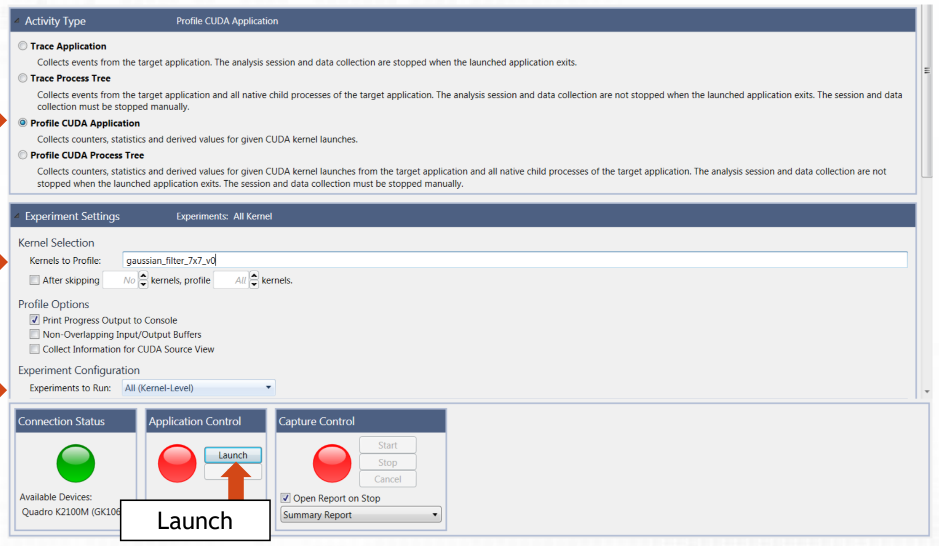
```
*(blockIdx.x*blockDim.x + threadIdx.y).
Idx.y;

((int) q0 >> 8);
)
har) ((int) q1 >> 8);

///////////////////////////////////////////////////////////////////////////

__global__ void sobel_filter_3x3_v0(int w, int h, const uchar *src, uchar *ds
{
  // Position of the thread in the image.
  const int x = blockIdx.x*blockDim.x + threadIdx.x;
```

Kernel Analysis



Activity Type — Profile CUDA Application

○ **Trace Application**
Collects events from the target application. The analysis session and data collection are stopped when the launched application exits.

○ **Trace Process Tree**
Collects events from the target application and all native child processes of the target application. The analysis session and data collection are not stopped when the launched application exits. The session and data collection must be stopped manually.

● **Profile CUDA Application**
Collects counters, statistics and derived values for given CUDA kernel launches.

○ **Profile CUDA Process Tree**
Collects counters, statistics and derived values for given CUDA kernel launches from the target application and all native child processes of the target application. The analysis session and data collection are not stopped when the launched application exits. The session and data collection must be stopped manually.

Experiment Settings — Experiments: All Kernel

Kernel Selection
Kernels to Profile: gaussian_filter_7x7_v0
☐ After skipping No kernels, profile All kernels.

Profile Options
☑ Print Progress Output to Console
☐ Non-Overlapping Input/Output Buffers
☐ Collect Information for CUDA Source View

Experiment Configuration
Experiments to Run: All (Kernel-Level)

☑ Open Report on Stop
Summary Report

Select Profile CUDA Application

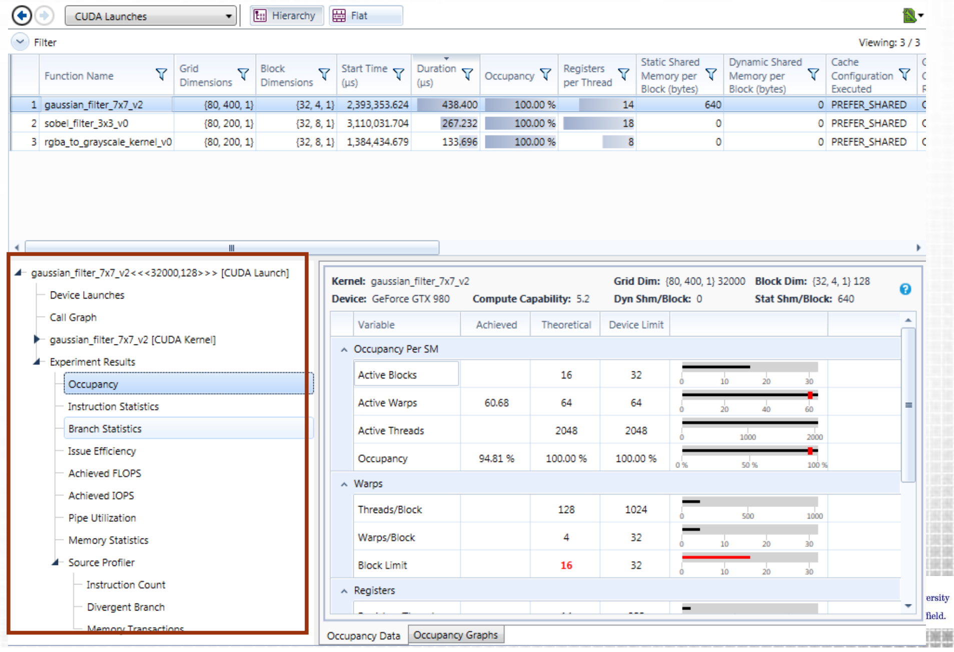
Select the Kernel (optional, will profile all kernels otherwise)

Select the Experiments (All)

Launch

CUDA Launches View

Performance Indicators



Viewing: 3 / 3

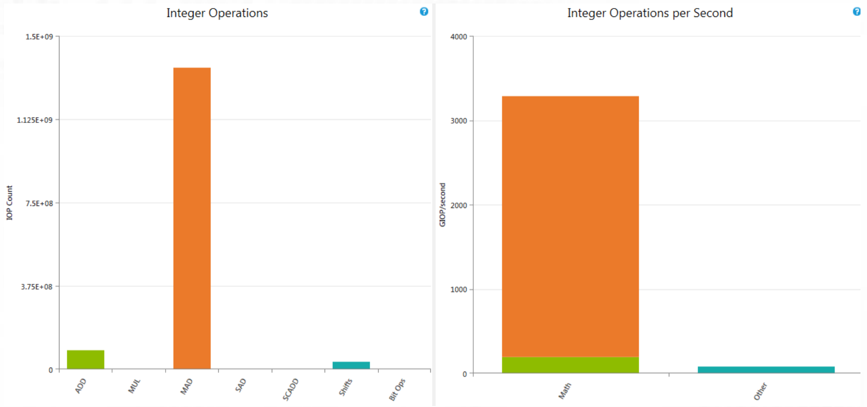
	Function Name	Grid Dimensions	Block Dimensions	Start Time (µs)	Duration (µs)	Occupancy	Registers per Thread	Static Shared Memory per Block (bytes)	Dynamic Shared Memory per Block (bytes)	Cache Configuration Executed
1	gaussian_filter_7x7_v2	{80, 400, 1}	{32, 4, 1}	2,393,353.624	438.400	100.00 %	14	640	0	PREFER_SHARED
2	sobel_filter_3x3_v0	{80, 200, 1}	{32, 8, 1}	3,110,031.704	267.232	100.00 %	18	0	0	PREFER_SHARED
3	rgba_to_grayscale_kernel_v0	{80, 200, 1}	{32, 8, 1}	1,384,434.679	133.696	100.00 %	8	0	0	PREFER_SHARED

- gaussian_filter_7x7_v2<<<32000,128>>> [CUDA Launch]
  - Device Launches
  - Call Graph
  - gaussian_filter_7x7_v2 [CUDA Kernel]
  - Experiment Results
    - Occupancy
    - Instruction Statistics
    - Branch Statistics
    - Issue Efficiency
    - Achieved FLOPS
    - Achieved IOPS
    - Pipe Utilization
    - Memory Statistics
    - Source Profiler
      - Instruction Count
      - Divergent Branch
      - Memory Transactions

Kernel: gaussian_filter_7x7_v2 **Grid Dim:** {80, 400, 1} 32000 **Block Dim:** {32, 4, 1} 128
Device: GeForce GTX 980 **Compute Capability:** 5.2 **Dyn Shm/Block:** 0 **Stat Shm/Block:** 640

Variable	Achieved	Theoretical	Device Limit
Occupancy Per SM			
Active Blocks		16	32
Active Warps	60.68	64	64
Active Threads		2048	2048
Occupancy	94.81 %	100.00 %	100.00 %
Warps			
Threads/Block		128	1024
Warps/Block		4	32
Block Limit		16	32
Registers			

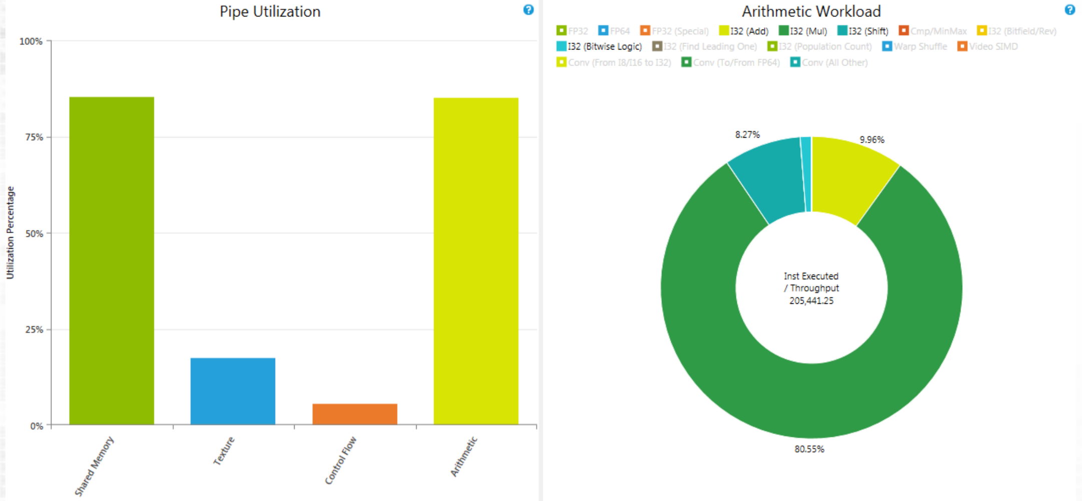
Achieved IOPS



- ❑No surprises…

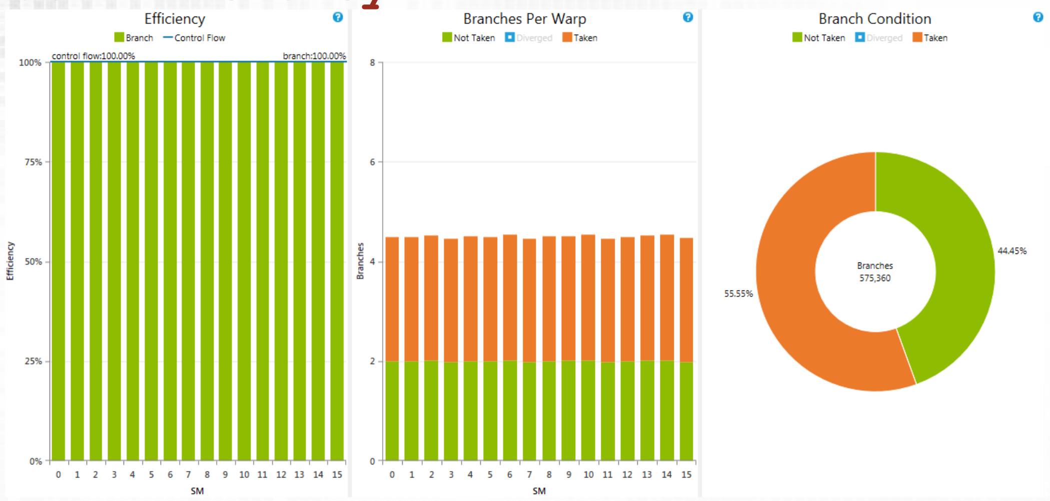
```
int p = 0;
 for( int j = 0 ; j < 7 ; ++j )
   for( int i = 0 ; i < 7 ; ++i )
     p += gaussian_filter[j][i] * n[j][i];
```

Pipe Utilisation



- ❑More detailed confirmation
  - ❑Integer operations dominate

Issue Efficiency



❑This is good

❑We have no divergent code

2nd Analysis

❑We have a reasonably well balanced use of the from Compute and Memory pipes.

❑**There is some latency in loading data to shared memory and on executions to read it back**

❑**Our compute cycles are dominated by Integer operations**

There is some latency in loading data to shared memory and on executions to read it back

❑Consider a simplified problem

❑Each thread needs to load an r, g, b, a value into shared memory

- ❑**Which has fewer shared memory load instructions?**

```
__shared__ char sm[TPB*4];

char r,g,b,a;

r = sm[threadidx.x];
g = sm[threadidx.x+1];
b = sm[threadidx.x+2];
a = sm[threadidx.x+3];
```

```
__shared__ char4 sm[TPB];

char r,g,b,a;
char4 rgba;
rgba = sm[threadidx.x];
r = rgba.r;
g = rgba.g;
b = rgba.b;
a = rgba.a;
```

There is some latency in loading data to shared memory and on executions to read it back

❑Consider a simplified problem

❑Each thread needs to load an r, g, b, a value into shared memory

- ❑**Which has fewer shared memory load instructions?**

```
__shared__ char sm[TPB*4];

char r,g,b,a;

r = sm[threadidx.x];
g = sm[threadidx.x+1];
b = sm[threadidx.x+2];
a = sm[threadidx.x+3];
```

```
__shared__ char4 sm[TPB];

char r,g,b,a;
char4 rgba;
rgba = sm[threadidx.x];
r = rgba.r;
g = rgba.g;
b = rgba.b;
a = rgba.a;
```

Our compute cycles are dominated by Integer operations

```
int p = 0;
 for( int j = 0 ; j < 7 ; ++j )
   for( int i = 0 ; i < 7 ; ++i )
     p += gaussian_filter[j][i] * n[j][i];
```

❑Which of the following is faster?

```
int a, b, c;
a = sm_a[i]; b = sm_b[i];

c += a * b;
```

```
float a, b, c;
a = sm_a[i]; b = sm_b[i];

c += a * b;
```

Our compute cycles are dominated by Integer operations

```
int p = 0;
 for( int j = 0 ; j < 7 ; ++j )
   for( int i = 0 ; i < 7 ; ++i )
     p += gaussian_filter[j][i] * n[j][i];
```

❑Which of the following is faster?

```
int a, b, c;
a = sm_a[i]; b = sm_b[i];

c += a * b;
```

```
float a, b, c;
a = sm_a[i]; b = sm_b[i];

c += a * b;
```

Integer multiply add is 16 cycles

Float combined multiply add is 4 cycles

Analysis

❑We have a reasonably well balanced use of the from Compute and Memory pipes.

❑There is some latency in loading data to shared memory and on executions to read it back

- ❑**Solution 1**: Reduce SM Load Stores dependencies by using wider requests. i.e. 4B values rather than 1B (chars)
- ❑I.e. Store shared memory values as 4B minimum

❑Our compute cycles are dominated by Integer operations

- ❑Almost all MAD operations
- ❑**Solution**: Change slower Integer MAD instructions to faster floating point FMAD instructions
- ❑I.e. Use floating point multiply and cast result to `uchar` at end

Improvement

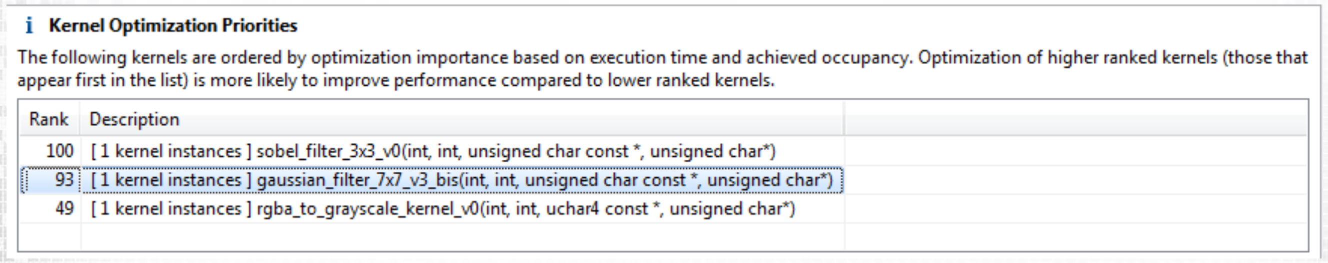
❑Significant

Kernel	Time (ms)	Speedup	Rel. Speedup
Gaussian_filter (Step 0)	5.49	1.00x	-
Gaussian_filter (Step 1a)	1.00	5.49x	5.49x
Gaussian_filter (Step 40)	0.49	11.20x	2.04x
Gaussian_filter (Step 5a)	**0.28**	**19.60x**	**1.75x**

- ❑Profiling Introduction
- ❑The Problem
- ❑Visual Profiler Guided Analysis
  - ❑Iteration 1
  - ❑Iteration 2
  - ❑Iteration 3
  - ❑Iteration 4

Identify the hotspot

- ❑Examine GPU Usage in Visual Profiler
- ❑What should be our next step?



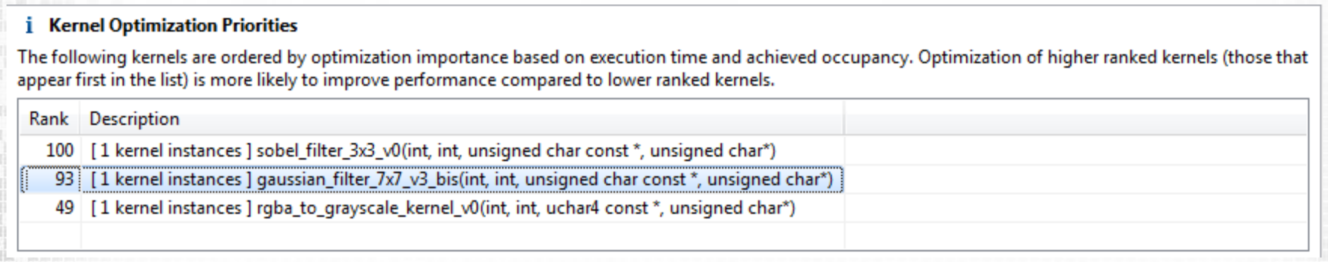
Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] sobel_filter_3x3_v0(int, int, unsigned char const *, unsigned char*)
93	[1 kernel instances] gaussian_filter_7x7_v3_bis(int, int, unsigned char const *, unsigned char*)
49	[1 kernel instances] rgba_to_grayscale_kernel_v0(int, int, uchar4 const *, unsigned char*)

Identify the hotspot

- ❑Examine GPU Usage in Visual Profiler
- ❑Examine Individual Kernels
  - ❑Gaussian filter kernel no longer highest rank!
  - ❑We can now optimise the sobel_filter kernel



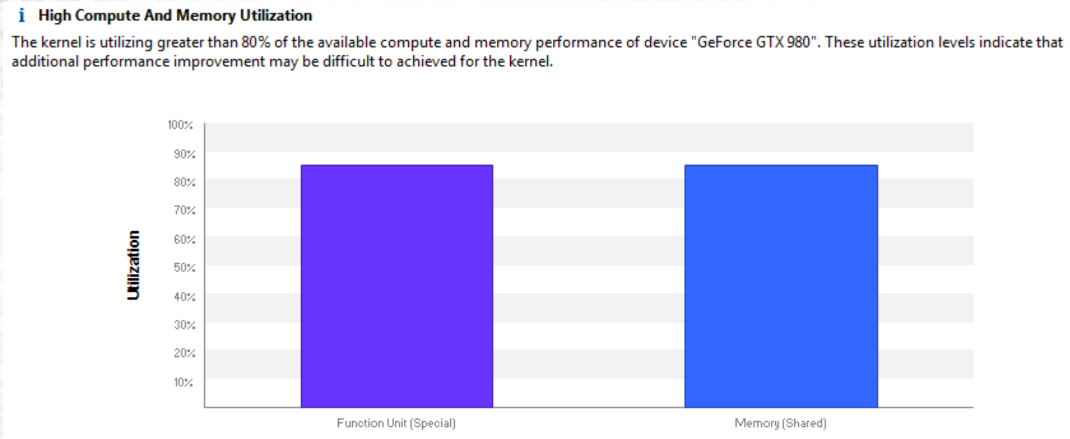
Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernels (those that appear first in the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] sobel_filter_3x3_v0(int, int, unsigned char const *, unsigned char*)
93	[1 kernel instances] gaussian_filter_7x7_v3_bis(int, int, unsigned char const *, unsigned char*)
49	[1 kernel instances] rgba_to_grayscale_kernel_v0(int, int, uchar4 const *, unsigned char*)

  - ❑Lets look at our Gaussian kernel anyway…

Performance Limiter

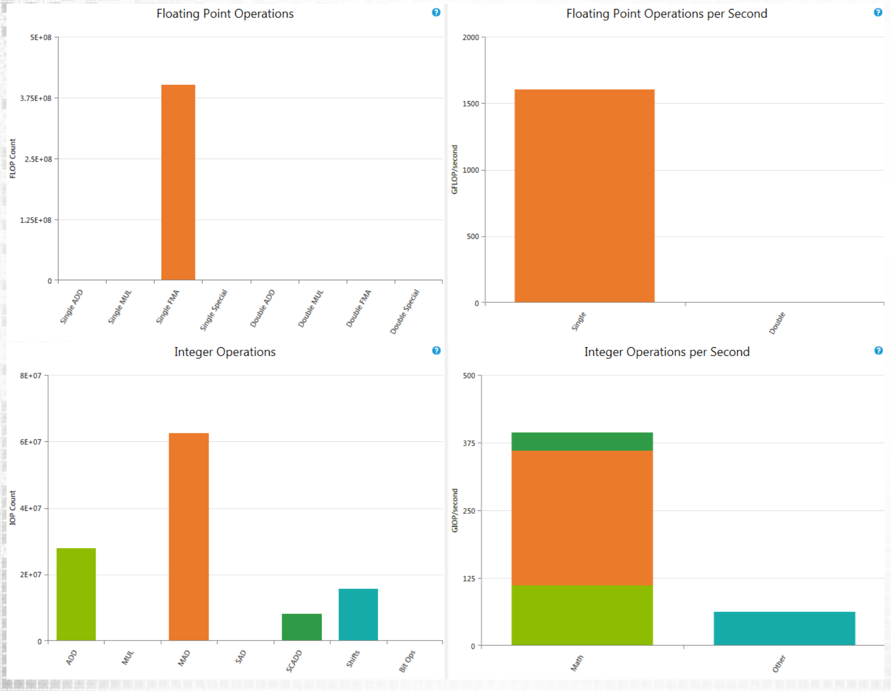


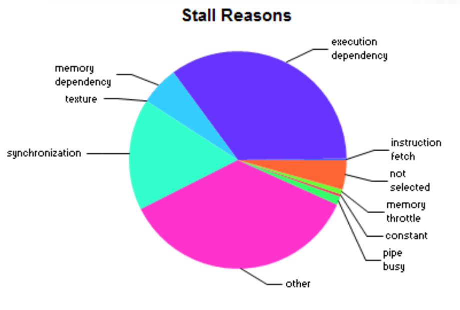
High Compute And Memory Utilization

The kernel is utilizing greater than 80% of the available compute and memory performance of device "GeForce GTX 980". These utilization levels indicate that additional performance improvement may be difficult to achieved for the kernel.

- ❑Looking good

VS NSight IOPS/ FLOPS Metrics





Analysis

- ❑Our algorithm is making good use of compute and memory
- ❑Further improvement will be difficult (but not impossible)

- ❑**Solution**: Optimise a different kernel
  - ❑sobel_filter_kernel to get the same treatment
- ❑**Solution**: Improve Gaussian kernel by changing the technique (parallelise differently)
  - ❑Separable Filter: Compute horizontal and vertical convolution separately then approximate by binominal coefficients
  - ❑Ensure we apply the same optimisations to separable filter version

Improvement

Kernel	Time (ms)	Speedup	Rel. Speedup
Gaussian_filter (Step 0)	5.49	1.00x	-
Gaussian_filter (Step 1a)	1.00	5.49x	5.49x
Gaussian_filter (Step 40)	0.49	11.20x	2.04x
Gaussian_filter (Step 5a)	0.28	19.60x	1.75x
Gaussian_filter (Step 9)	**0.22**	**24.95x**	**1.27x**

- ❑25x speedup on existing GPU code is pretty good

- ❑Companion Code: https://github.com/chmaruni/nsight-gtc

Summary

- ❑Profiling with the Visual Profiler will give you guided analysis of how to improve your performance
  - ❑Show you how to spot key metrics
- ❑**We are trying to achieve good overall utilisation of the hardware (compute and memory engines)**
  - ❑**Through an appreciation of memory and compute bounds**
- ❑Follow the APOD cycle
  - ❑Assess: What is the limiting factor, analyse and profile
  - ❑Parallelise and improve (apply the knowledge you have learnt over the course)
  - ❑Optimise
  - ❑Deploy and Test
- ❑If in doubt use the lab classes to seek guidence!