

Profiling & Tuning Applications

CUDA Course

István Reguly



UNIVERSITY OF
OXFORD

OXFORD
e-Research
CENTRE

Introduction

- Why is my application running slow?
- Work it out on paper
- Instrument code
- Profile it
 - NVIDIA Visual Profiler
 - Works with CUDA, needs some tweaks to work with OpenCL
 - nvprof – command line tool, can be used with MPI applications

Identifying Performance Limiters

- CPU: Setup, data movement
- GPU: Bandwidth, compute or latency limited
- Number of instructions for every byte moved
 - $\sim 3.6 : 1$ on Fermi
 - $\sim 6.4 : 1$ on Kepler
- Algorithmic analysis gives a good estimate
- Actual code is likely different
 - Instructions for loop control, pointer math, etc.
 - Memory access patterns
 - How to find out?
 - Use the profiler (quick, but approximate)
 - Use source code modification (takes more work)

Analysis with Source Code Modification

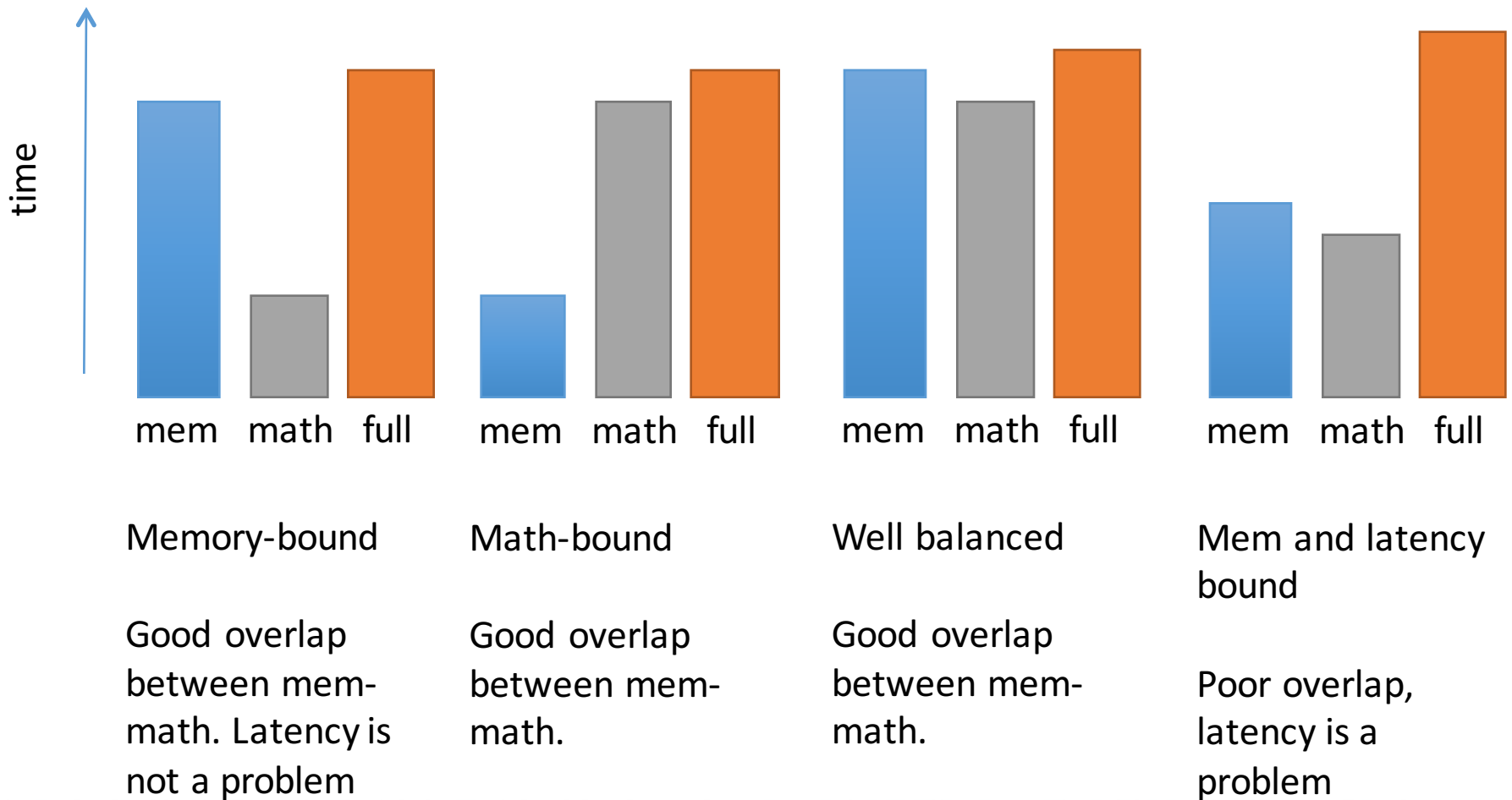
- Time memory-only and math-only versions
 - Not so easy for kernels with data-dependent control flow
 - Good to estimate time spent on accessing memory or executing instructions
- Shows whether kernel is memory or compute bound
- Put an “if” statement depending on kernel argument around math/mem instructions
 - Use dynamic shared memory to get the same occupancy

Analysis with Source Code Modification

```
__global__ void kernel(float *a) {  
    int idx = threadIdx.x + blockDim.x+blockIdx.x;  
    float my_a;  
    my_a = a[idx];  
    for (int i=0; i < 100; i++) my_a = sinf(my_a+i*3.14f);  
    a[idx] = my_a;  
}
```

```
__global__ void kernel(float *a, int prof) {  
    int idx = threadIdx.x + blockDim.x+blockIdx.x;  
    float my_a;  
    if (prof & 1) my_a = a[idx];  
    if (prof & 2)  
        for (int i=0; i < 100; i++) my_a =  
            sinf(my_a+i*3.14f);  
    if (prof & 1) a[idx] = my_a;  
}
```

Example scenarios



NVIDIA Visual Profiler

- Launch with “nvvp”
- Collects metrics and events during execution
 - Calls to the CUDA API
 - Overall application:
 - Memory transfers
 - Kernel launches
 - Kernels
 - Occupancy
 - Computation efficiency
 - Memory bandwidth efficiency
- Requires deterministic execution!

Meet the test setup

- 2D gaussian blur with a 5x5 stencil $\frac{1}{273}$
- 4096² grid

1	4	7	4	1
4	16	26	16	4
7	26	41	26	7
4	16	26	16	4
1	4	7	4	1

```
__global__ void stencil_v0(float *input, float *output,  
                           int sizex, int sizey) {
```

```
    const int x = blockIdx.x*blockDim.x + threadIdx.x + 2;  
    const int y = blockIdx.y*blockDim.y + threadIdx.y + 2;  
    if ((x >= sizex-2) || (y >= sizey-2)) return;  
    float accum = 0.0f;  
    for (int i = -2; i < 2; i++) {  
        for (int j = -2; j < 2; j++) {  
            accum += filter[i+2][j+2]*input[sizey*(y+j) +  
(x+i)];  
        }  
    }  
    output[sizey*y+x] = accum/273.0f;  
}
```


Meet the test setup

- NVIDIA K40
 - GK110B
 - SM 3.5
 - ECC on
 - Graphics clocks at 745MHz, Memory clocks at 3004MHz

- CUDA 7.0

```
nvcc profiling_lecture.cu -O2 -arch=sm_35 -I. -lineinfo -DIT=0
```

Interactive demo of tuning process

Launch a profiling session

Create New Session

Executable Properties
Set executable properties

Connection: Local Manage...

Toolkit: CUDA Toolkit 7.0 (/usr/local/cuda-7.0/bin/) Manage...

File: ./a.out Browse...

Working directory: Enter working directory [optional] Browse...

Arguments: Enter command-line arguments

Environment:

Name	Value
------	-------

Add
Delete

< Back Next > Cancel Finish

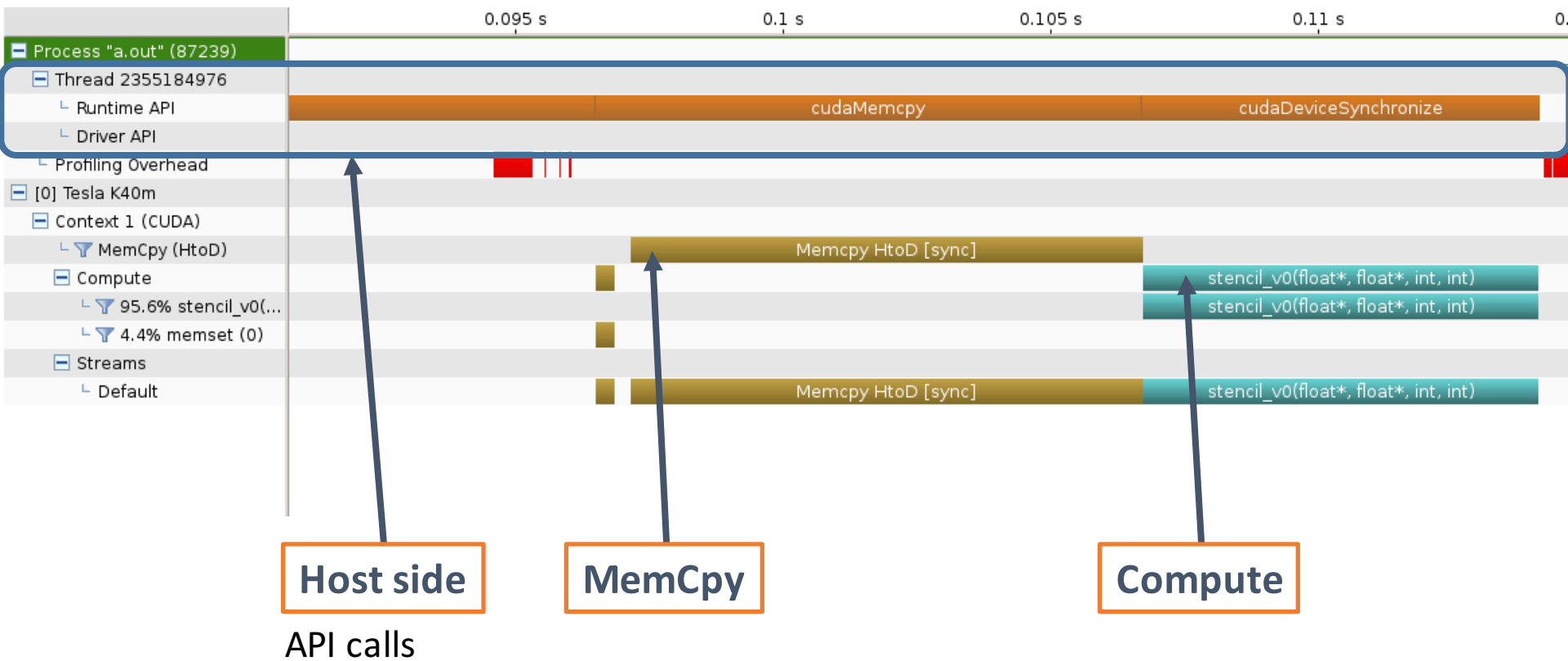
First look

The screenshot displays the NVIDIA Nsight Visual Studio Edition interface, which is used for profiling and analyzing GPU applications. The interface is divided into several panels:

- Timeline:** A central panel showing a detailed timeline of GPU activity. It includes a list of processes and threads on the left, a central timeline view with colored bars representing different operations (e.g., `cudaMemcpy`, `cudaDeviceSynchronize`, `MemCpy HtoD [sync]`, and `stencil_v0(float*, float*, int, int)`), and a right-hand panel showing properties for the selected process.
- Summary:** A panel on the right side of the timeline view, providing a summary of the selected process's performance, including duration and session information.
- Guide:** A panel on the left side of the interface, providing a guided analysis of the application. It includes sections for "1. CUDA Application Analysis" and "Examine GPU Usage", which describe the importance of understanding GPU usage and the steps involved in analyzing individual kernels.
- Analysis results:** A large panel on the right side of the interface, displaying the results of the analysis. It includes a "Results" section and a "Quick Access" bar.

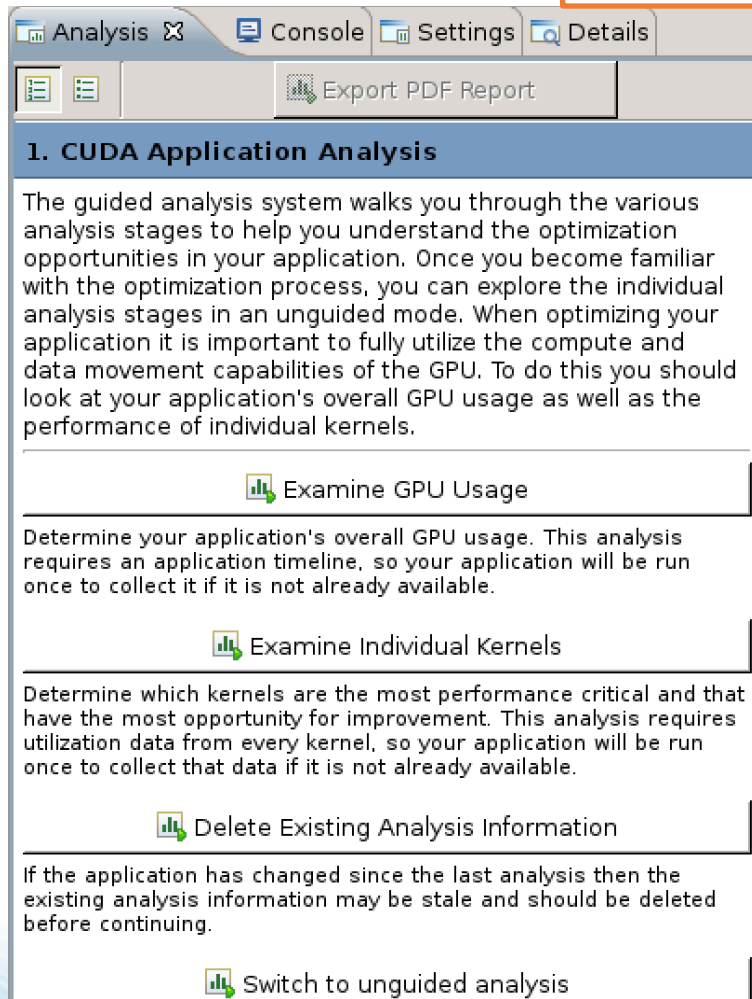
The interface also features a menu bar at the top with options like File, View, Run, Help, and Search, and a toolbar with various icons for navigation and analysis.

The Timeline



Analysis


Guided




The screenshot shows the 'Guided' analysis mode. The top navigation bar includes 'Analysis', 'Console', 'Settings', and 'Details'. Below this is a toolbar with 'Export PDF Report'. The main content area is titled '1. CUDA Application Analysis' and contains a detailed introductory paragraph about the guided analysis system. Below the text are five interactive buttons, each with a bar chart icon: 'Examine GPU Usage', 'Examine Individual Kernels', 'Delete Existing Analysis Information', and 'Switch to unguided analysis'. The 'Switch to unguided analysis' button is at the bottom of the list.

1. CUDA Application Analysis


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

 **Examine GPU Usage**


Determine your application's overall GPU usage. This analysis requires an application timeline, so your application will be run once to collect it if it is not already available.

 **Examine Individual Kernels**

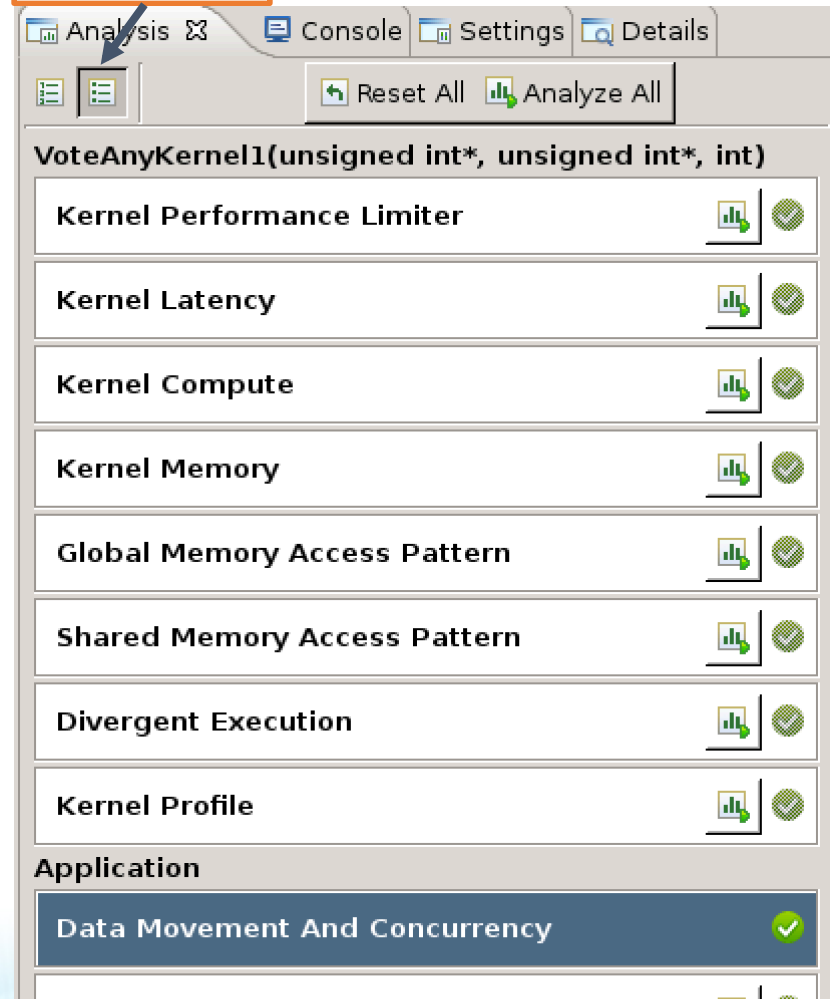
Determine which kernels are the most performance critical and that have the most opportunity for improvement. This analysis requires utilization data from every kernel, so your application will be run once to collect that data if it is not already available.

 **Delete Existing Analysis Information**

If the application has changed since the last analysis then the existing analysis information may be stale and should be deleted before continuing.



 **Switch to unguided analysis**



Unguided







The screenshot shows the 'Unguided' analysis mode. The top navigation bar is identical to the guided mode. The toolbar now includes 'Reset All' and 'Analyze All' buttons. The main content area is titled 'VoteAnyKernel1(unsigned int*, unsigned int*, int)' and displays a list of analysis stages. Each stage has a bar chart icon and a green checkmark, indicating that the analysis is complete. The stages are: 'Kernel Performance Limiter', 'Kernel Latency', 'Kernel Compute', 'Kernel Memory', 'Global Memory Access Pattern', 'Shared Memory Access Pattern', 'Divergent Execution', and 'Kernel Profile'. Below these is an 'Application' section with a 'Data Movement And Concurrency' item, which also has a green checkmark.



VoteAnyKernel1(unsigned int*, unsigned int*, int)



Kernel Performance Limiter  



Kernel Latency  



Kernel Compute  

Kernel Memory  


Global Memory Access Pattern  

Shared Memory Access Pattern  

Divergent Execution  

Kernel Profile  

Application

Data Movement And Concurrency 

Examine Individual Kernels

Results

i Kernel Optimization Priorities

The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher ranked kernel: the list) is more likely to improve performance compared to lower ranked kernels.

Rank	Description
100	[1 kernel instances] stencil_v0(float*, float*, int, int)

Lists all kernels sorted by total execution time: the higher the rank the higher the impact of optimisation on overall performance

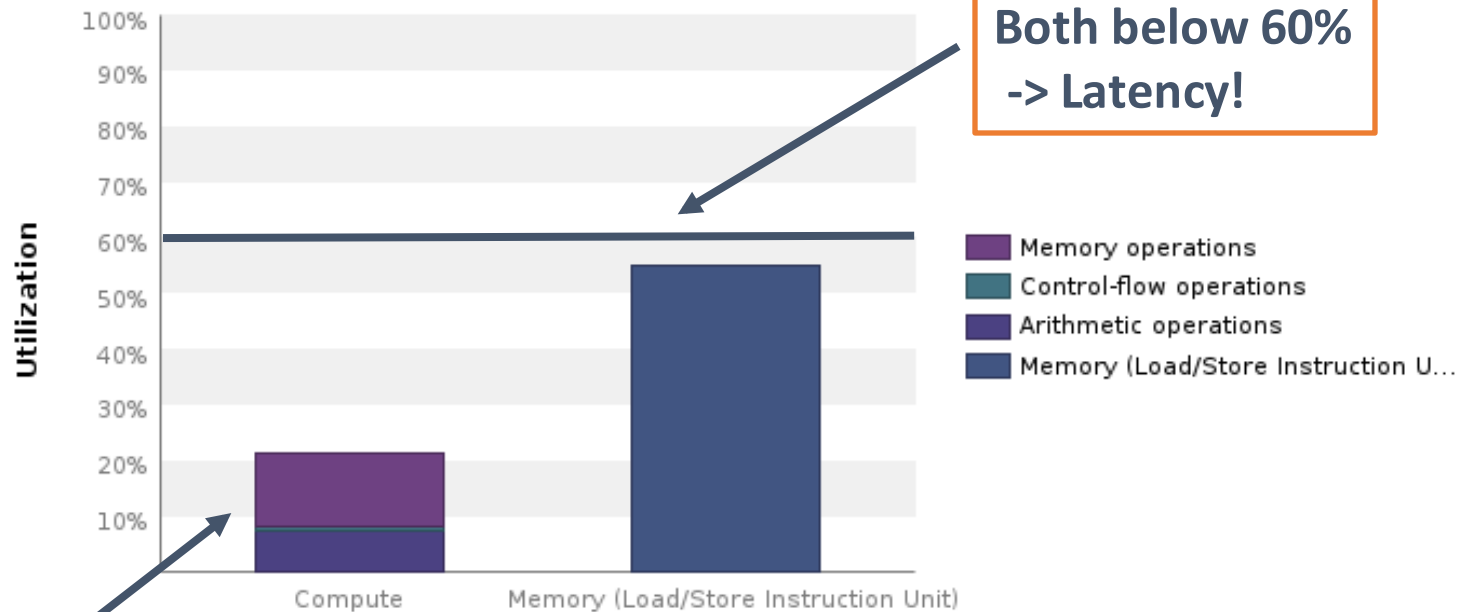
Initial unoptimised (v0)**8.122ms**

Utilisation

Results

i Kernel Performance Is Bound By Instruction And Memory Latency

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



Both below 60%
-> Latency!

Most of it is
memory ops

Let's investigate

Latency analysis

Analysis Details Console


Export PDF Report

1. CUDA Application Analysis


2. Performance-Critical Kernels


3. Compute, Band...or Latency Bound

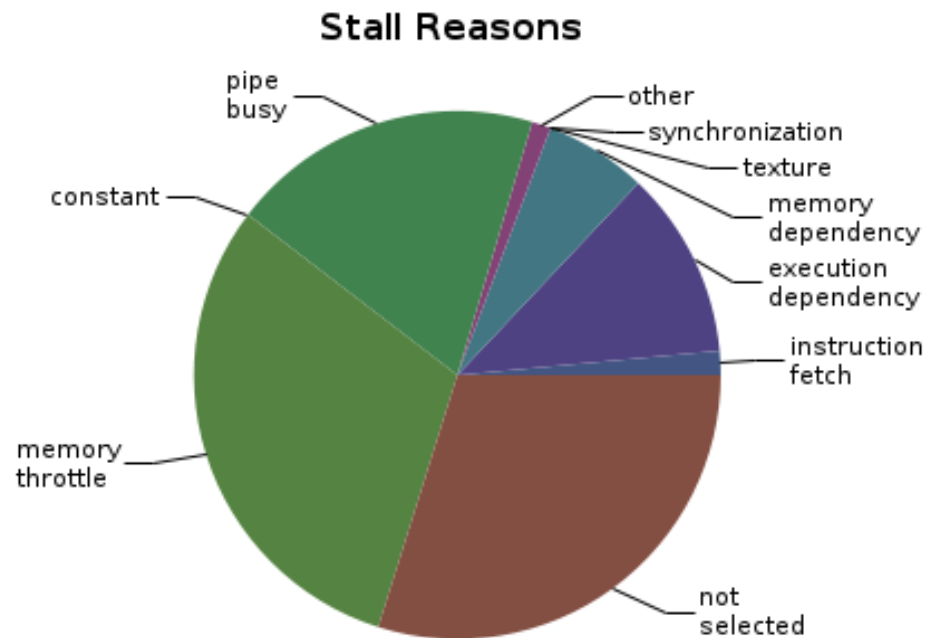
The first step in analyzing an individual kernel is to determine if the performance of the kernel is bounded by computation, memory bandwidth, or instruction/memory latency. The results at right indicate that the performance of kernel "stencil_v0" is most likely limited by instruction and memory latency.

 Perform Latency Analysis

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

 Perform Compute Analysis

 Perform Memory Bandwidth Analysis




Memory throttle -> perform BW analysis


Memory Bandwidth analysis


Results


i Memory Bandwidth And Utilization

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. [More...](#)

	Transactions	Bandwidth	Utilization
L1/Shared Memory			
Local Loads	0	0 B/s	
Local Stores	0	0 B/s	
Shared Loads	0	0 B/s	
Shared Stores	0	0 B/s	
Global Loads	40894464	248.782 GB/s	
Global Stores	2621440	16.585 GB/s	
Atomic	0	0 B/s	
L1/Shared Total	43515904	265.367 GB/s	

L2 Cache			
L1 Reads	62914560	248.782 GB/s	
L1 Writes	4194304	16.585 GB/s	
Texture Reads	0	0 B/s	
Atomic	0	0 B/s	
Noncoherent Reads	0	0 B/s	
Total	67108864	265.367 GB/s	

Texture Cache			
Reads	0	0 B/s	

Device Memory			
Reads	3756909	14.856 GB/s	
Writes	2904475	11.485 GB/s	
Total	6661384	26.341 GB/s	
ECC Overhead	2451525	9.694 GB/s	

L1 cache
not used...

Investigate further...

Unguided

The screenshot shows the NVIDIA Nsight Systems interface. On the left, a sidebar lists analysis categories for the kernel `stencil_v0(float*, float*, int, int)`. The 'Global Memory Access Pattern' category is selected and highlighted in blue. Other categories include Kernel Performance Limiter, Kernel Latency, Kernel Compute, Kernel Memory, Shared Memory Access Pattern, and Divergent Execution, all of which show a green checkmark indicating they are analyzed.

The main panel displays the 'Results' for the selected category. It includes a warning icon and the title 'Global Memory Alignment and Access Pattern'. The text explains that memory bandwidth is used most efficiently when global memory loads and stores have proper alignment and access patterns. It provides an optimization tip: 'Select each entry below to open the source code to a global load or store within the kernel with access pattern. For each load or store improve the alignment and access pattern of the memory access.'

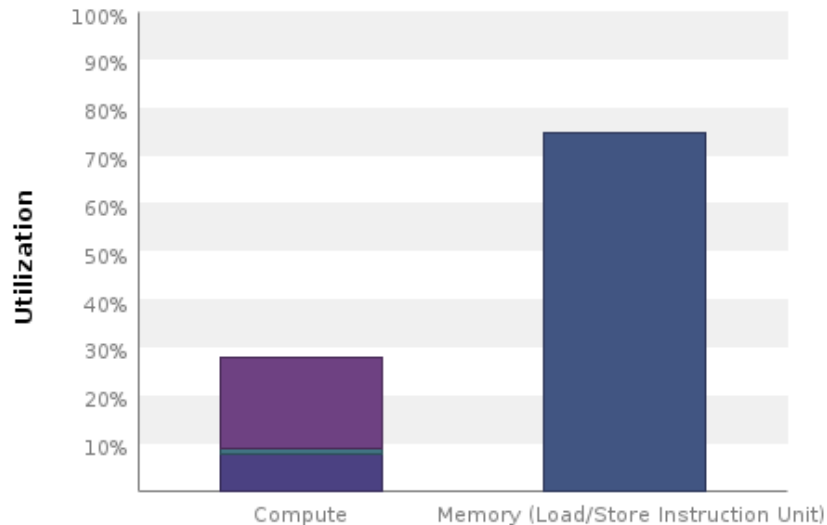
Below the text is a table of results. The table has two columns: 'Line / File' and a description of the memory access pattern. The file path is `profiling_lecture.cu - /home/mgiles/ireguly/cuda_course`. The table shows eight entries, all at line 25, representing global loads. Each entry reports the number of L2 transactions per access and the ideal number of transactions per access.

Line / File	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 6, Ideal Transactions/Access = 4 [3145728 L2 transacti
25	Global Load L2 Transactions/Access = 6, Ideal Transactions/Access = 4 [3145728 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transacti

6-8 transactions per access – something is wrong with how we access memory

Global memory load efficiency 53.3%
L2 hit rate 96.7%

Iteration 1 – turn on L1



Quick & easy step:
Turn on L1 cache by using
`-Xptxas -dlcm=ca`

Line / File	profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25	Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [10485760 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4 [9437184 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 20, Ideal Transactions/Access = 4 [10485760 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 18, Ideal Transactions/Access = 4 [9437184 L2 transactions for 524288 total ex

Memory unit is utilized, but Global Load efficiency became even worse: 20.5%

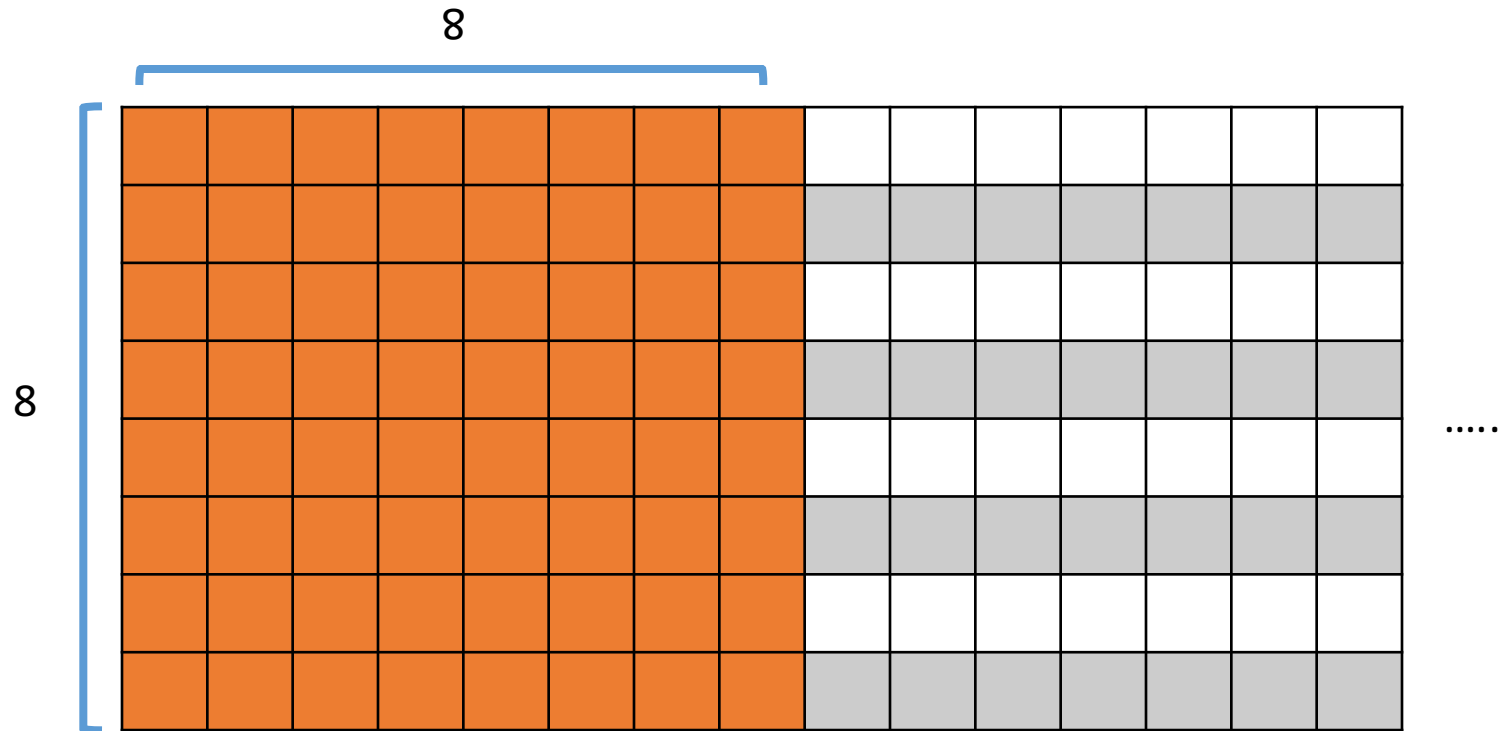
Initial unoptimised (v0)

8.122ms

Enable L1

6.57ms

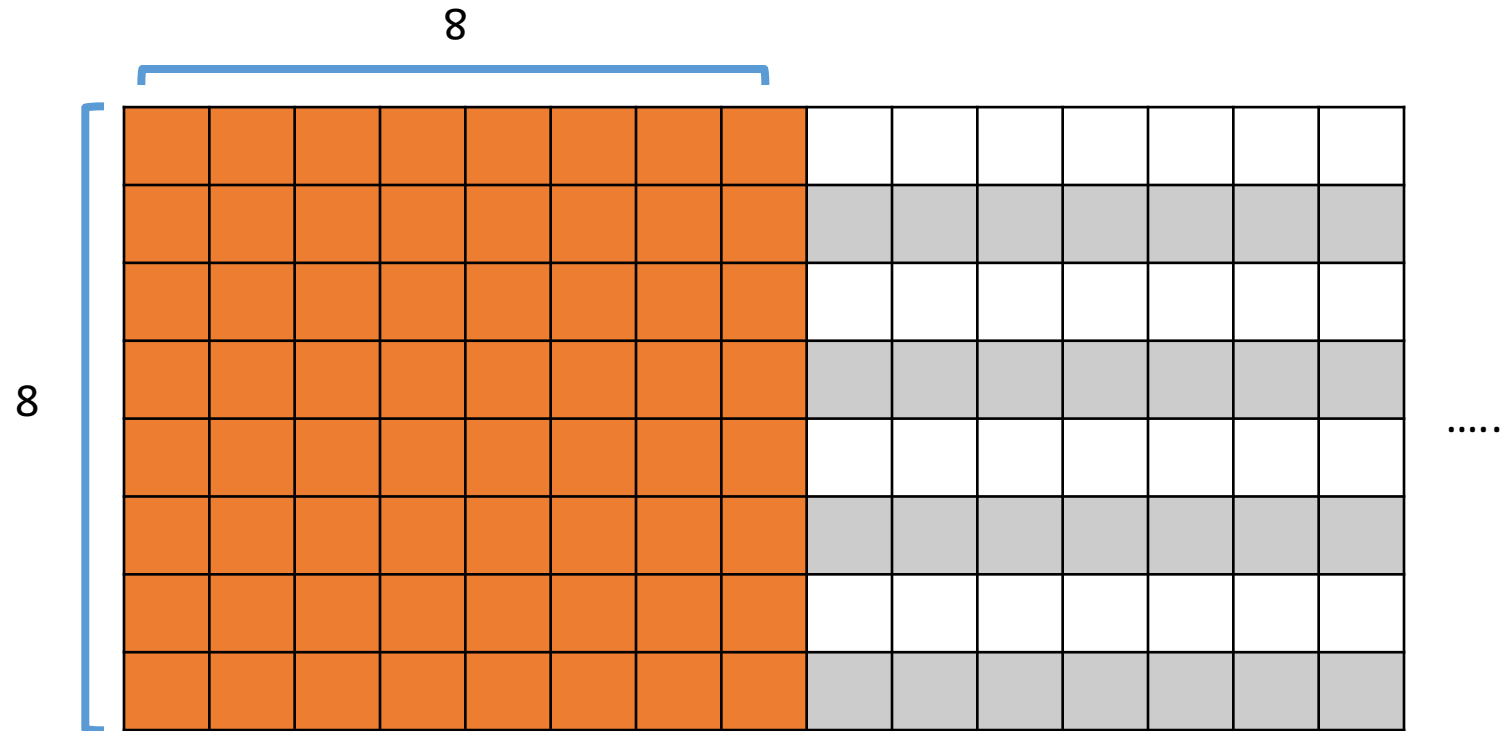
Cache line utilization



32 bytes (8 floats)
Unit of transaction

L1 cache disabled:
-> 32B transactions
Min 4, Max 8 transactions

Cache line utilization



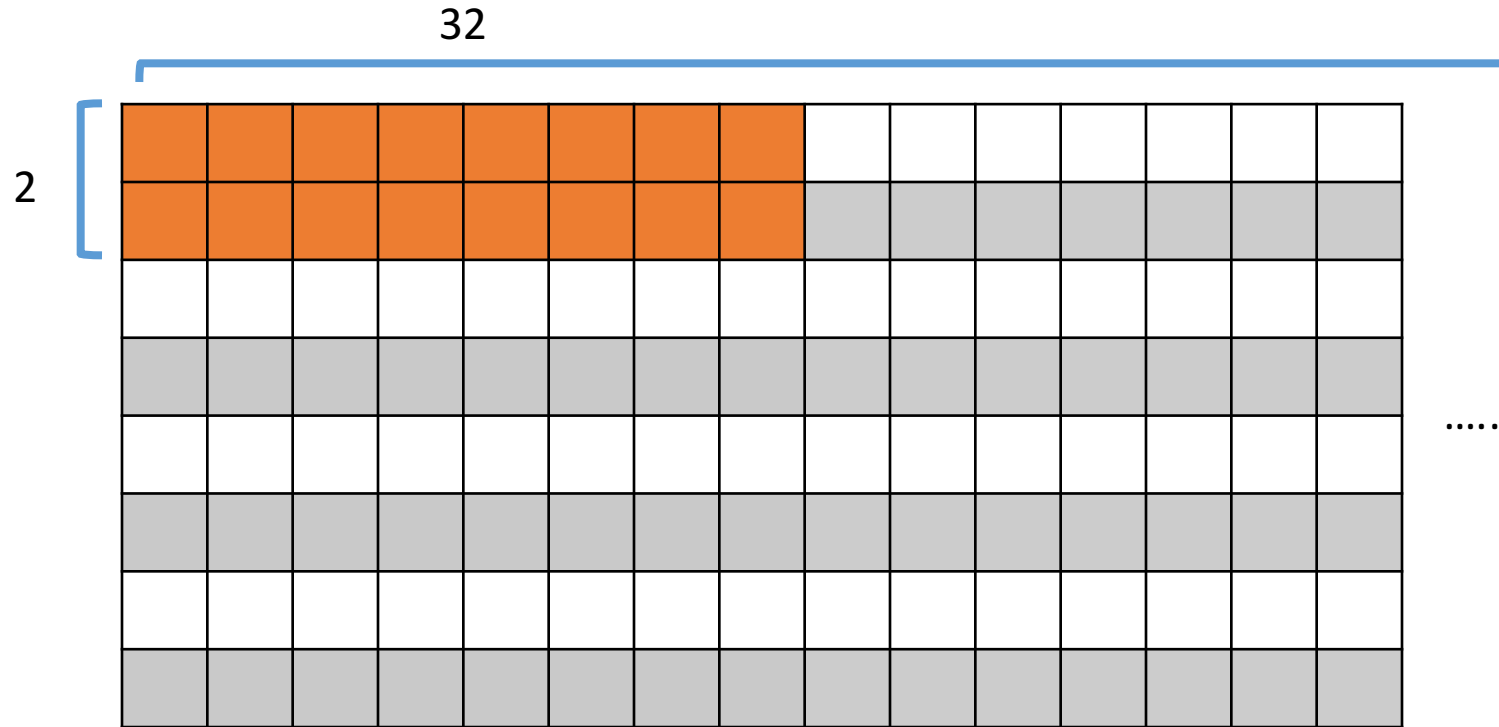
128 bytes (32 floats)
Unit of transaction

Each time a transaction requires more than 1 128B cache line: re-issue

L1 cache enabled:
-> 128B transactions
-> 4*32B to L2

Min 16, Max 32 transactions

Cache line utilization

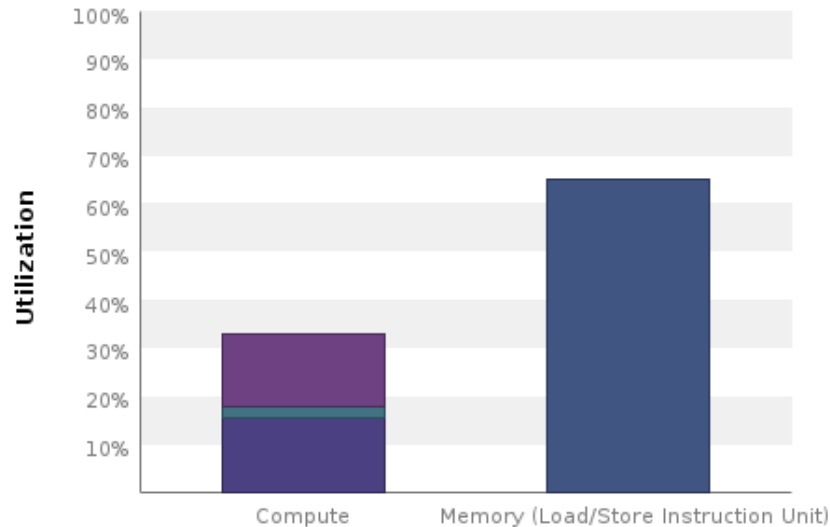


128 bytes (32 floats)
Unit of transaction

L1 cache enabled:
-> 128B transactions
-> 4*32B to L2

Min 4, Max 8 transactions

Iteration 2 – 32x2 blocks



Memory utilization decreased 10%
Performance almost doubles
Global Load Efficiency 50.8%

▼ Line / File	profiling_lecture.cu - /home/mgiles/ireguly/cuda_course
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transactions for 524288 total exec
25	Global Load L2 Transactions/Access = 7.5, Ideal Transactions/Access = 4 [3932160 L2 transactions for 524288 total ex
25	Global Load L2 Transactions/Access = 8, Ideal Transactions/Access = 4 [4194304 L2 transactions for 524288 total exec

Initial unoptimised (v0)	8.122ms
---------------------------------	----------------

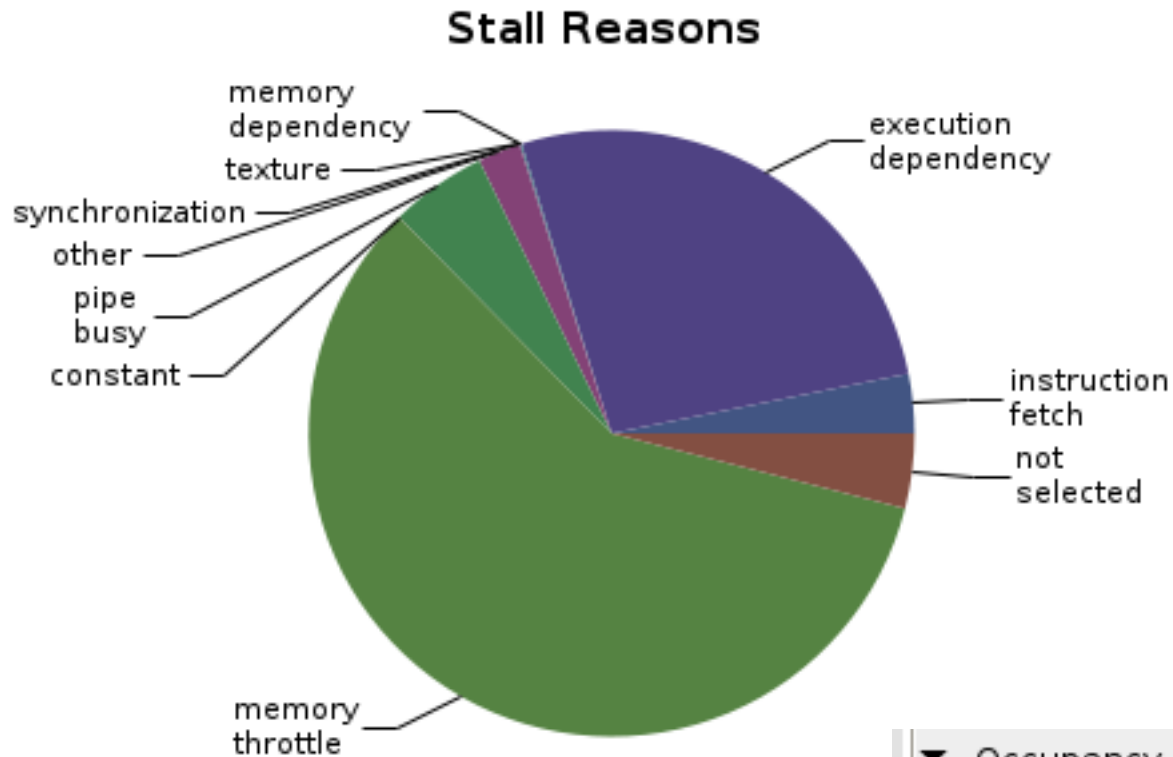
Enable L1	6.57ms
------------------	---------------

Blocksize	3.4ms
------------------	--------------

Key takeaway

- **Latency/Bandwidth bound**
- Inefficient use of memory system and bandwidth
- Symptoms:
 - Lots of transactions per request (low load efficiency)
- Goal:
 - Use the whole cache line
 - Improve memory access patterns (coalescing)
- What to do:
 - Align data, change block size, change data layout
 - Use shared memory/shuffles to load efficiently

Latency analysis



▼ Occupancy	
Achieved	⚠ 41.7%
Theoretical	50%
Limiter	Block Size

Latency analysis

Optimization: Increase the number of threads in each block to increase the number of warps that can execute on each SM. [More...](#)

Variable	Achieved	Theoretical	Device Limit	Grid Size: [128,2048,1] (262144 blocks)Block Size: [3
----------	----------	-------------	--------------	--

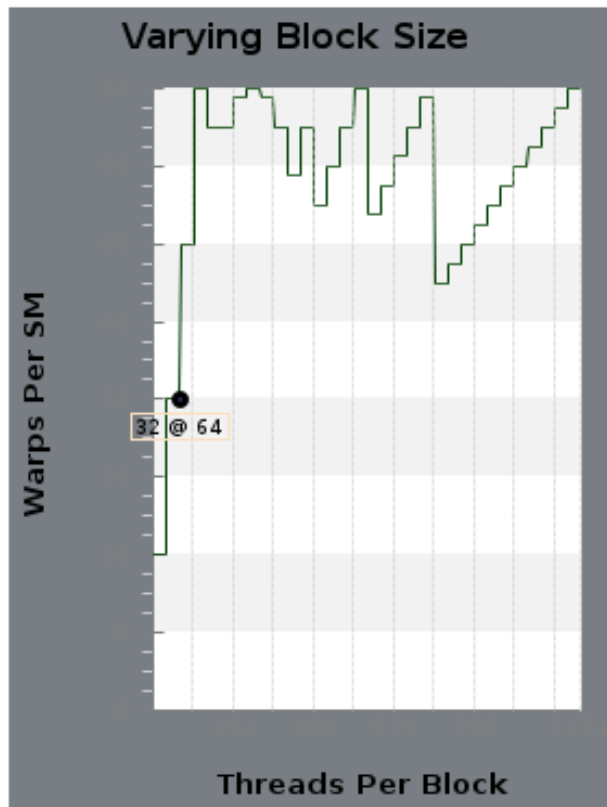
Occupancy Per SM

Active Blocks		16	16	
Active Warps	26.67	32	64	
Active Threads		1024	2048	
Occupancy	41.7%	50%	100%	

Warps

Threads/Block		64	1024	
Warps/Block		2	32	
Block Limit		32	16	

Latency analysis



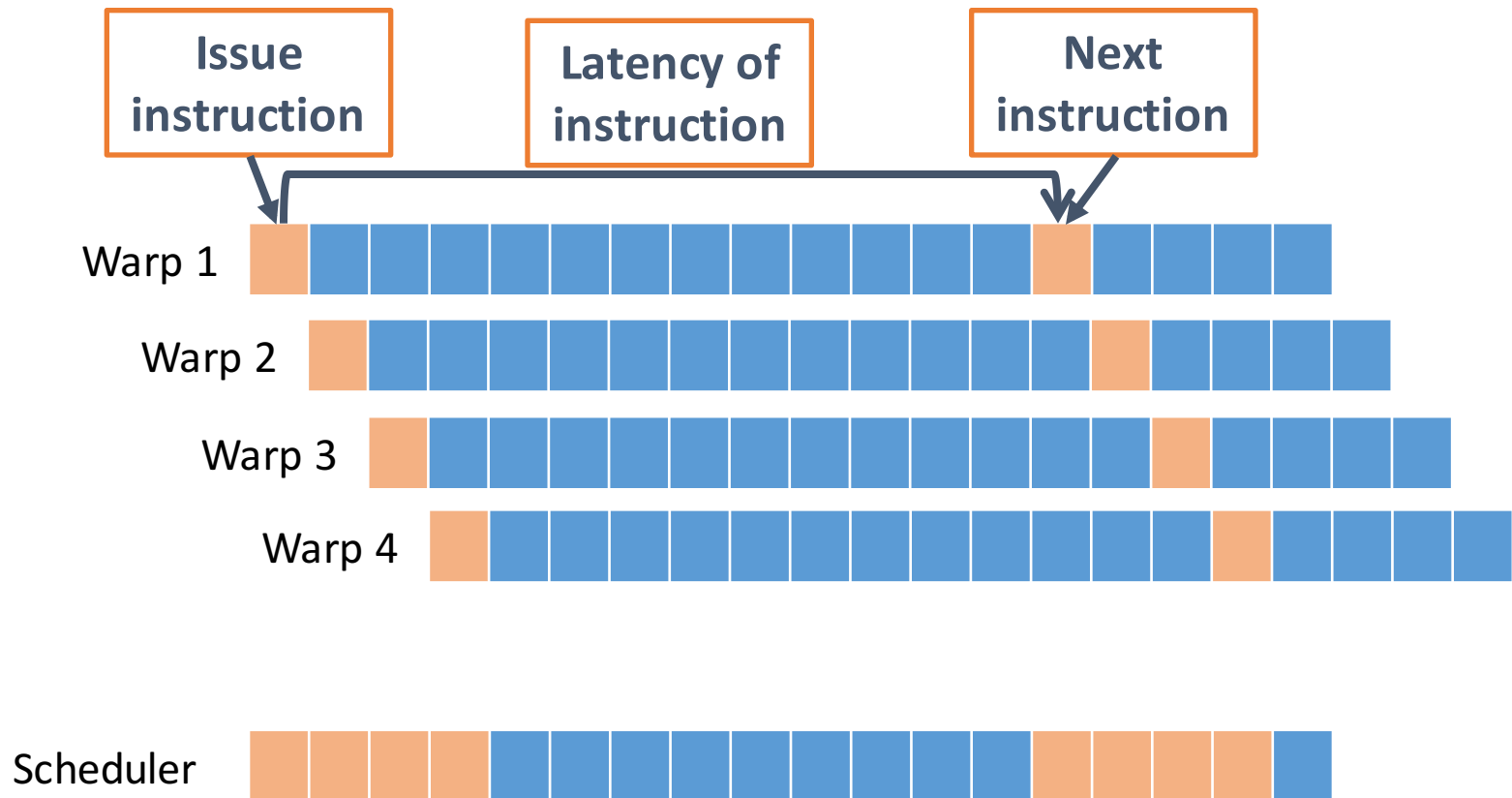
Increase the block size so more warps can be active at the same time.

Kepler:

Max 16 blocks per SM

Max 2048 threads per SM

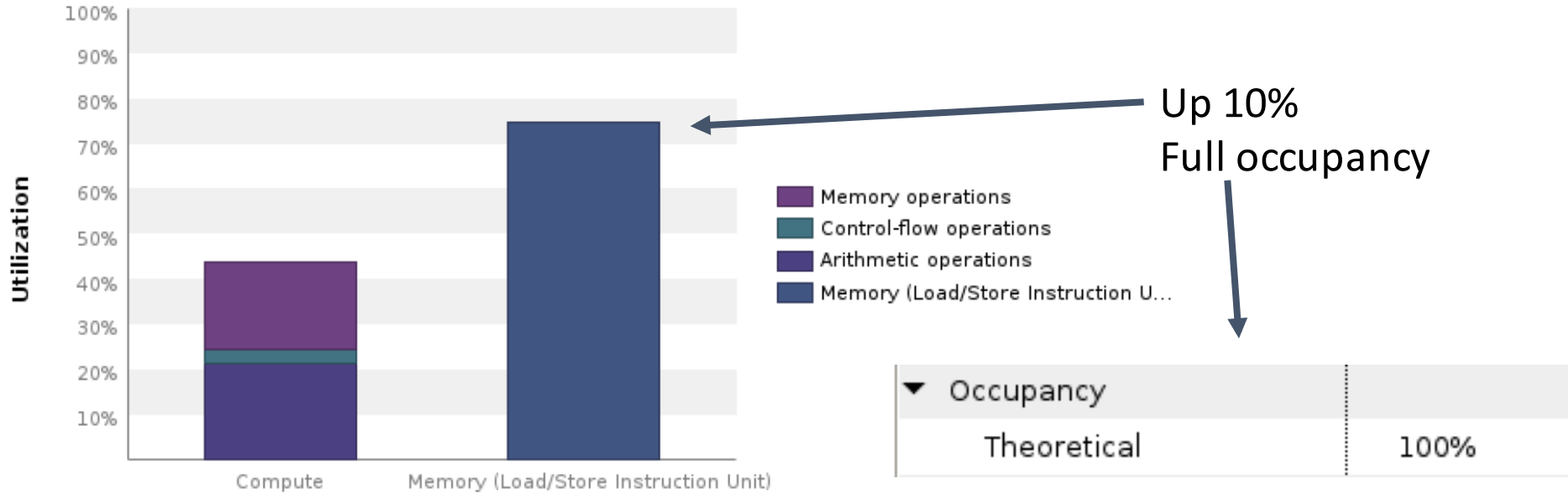
Occupancy – using all “slots”



Increase block size to 32x4

Illustrative only, reality is a bit more complex...

Iteration 3 – 32x4 blocks



Initial unoptimised (v0)	8.122ms
--------------------------	---------

Enable L1	6.57ms
-----------	--------

Blocksize	3.4ms
-----------	-------

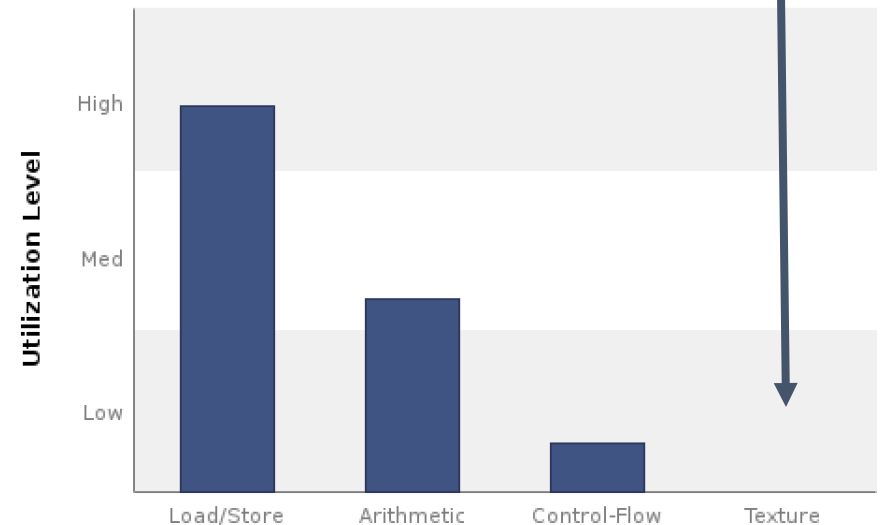
Blocksize 2	2.36ms
-------------	--------

Key takeaway

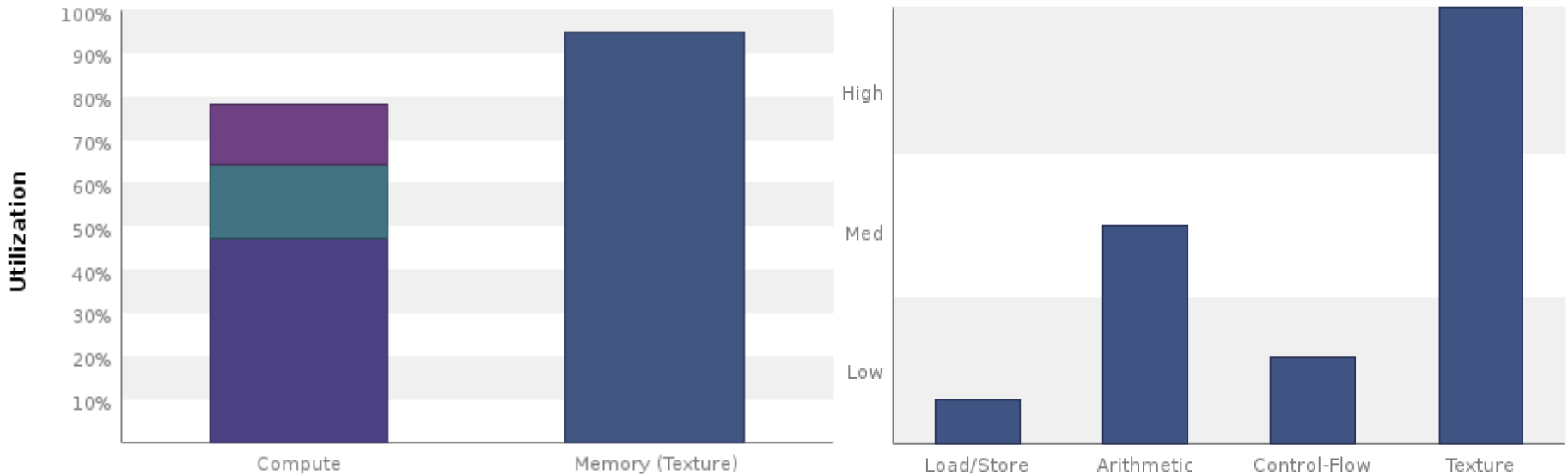
- **Latency bound – low occupancy**
- Unused cycles, exposed latency
- Symptoms:
 - High execution/memory dependency, low occupancy
- Goal:
 - Better utilise cycles by: having more warps
- What to do:
 - Determine occupancy limiter (registers, block size, shared memory) and vary it

Improving memory bandwidth

- L1 is fast, but a bit wasteful (128B loads)
 - 8 transactions on average (minimum would be 4)
- Load/Store pipe stressed
 - Any way to reduce the load?
- Texture cache
 - Dedicated pipeline
 - 32 byte loads
 - `const __restrict__ *`
 - `__ldg()`



Iteration 4 – texture cache



Texture Cache



Device Memory

Initial unoptimised (v0)	8.122ms
--------------------------	---------

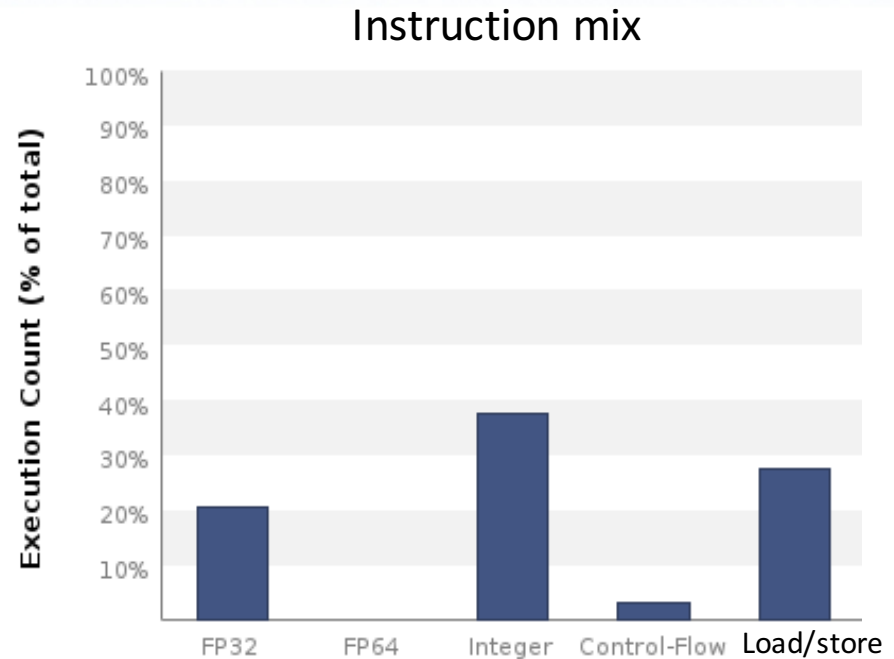
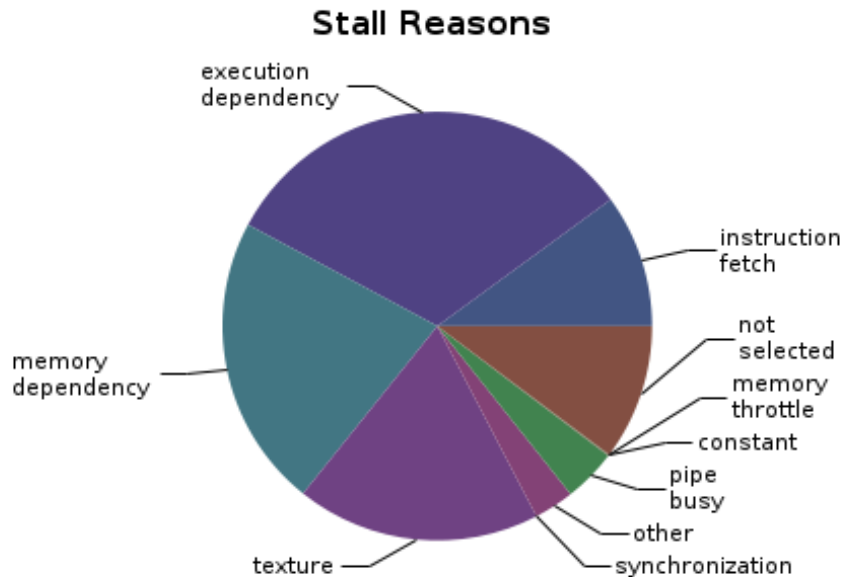
Blocksize 2	2.36ms
-------------	--------

Texture cache	1.53ms
---------------	--------

Key takeaway

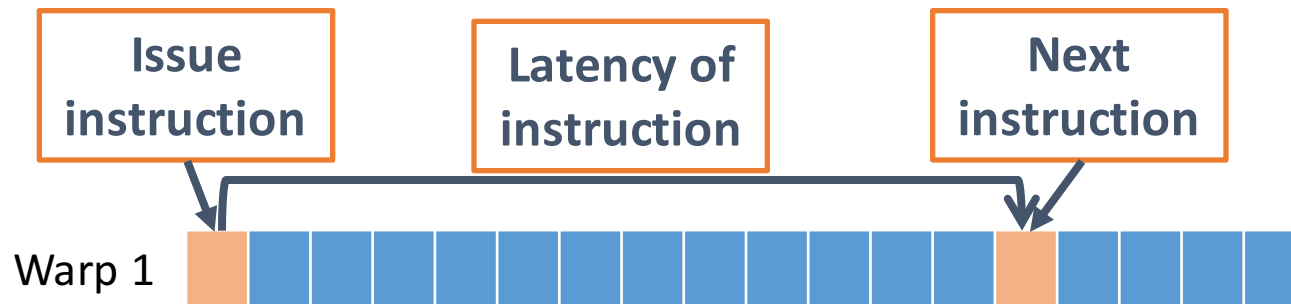
- **Bandwidth bound – Load/Store Unit**
- LSU overutilised
- Symptoms:
 - LSU pipe utilisation high, others low
- Goal:
 - Better spread the load between other pipes: use TEX
- What to do:
 - Read read-only data through the texture cache
 - `const __restrict__` or `__ldg()`

Compute analysis



Compute utilization could be higher (~78%)
Lots of Integer & memory instructions, fewer FP
Integer ops have lower throughput than FP
Try to amortize the cost: increase compute per byte

Instruction Level Parallelism



- Remember, GPU is in-order:

$a=b+c$	$a=b+c$
\downarrow	
$d=a+e$	$d=e+f$

- Second instruction cannot be issued before first
 - But it can be issued before the first finishes – if there is no dependency
- Applies to memory instructions too – latency much higher (counts towards stall reasons)

Instruction Level Parallelism

```
for (j=0;j<2;j++)  
    acc+=filter[j]*input[x+j];
```

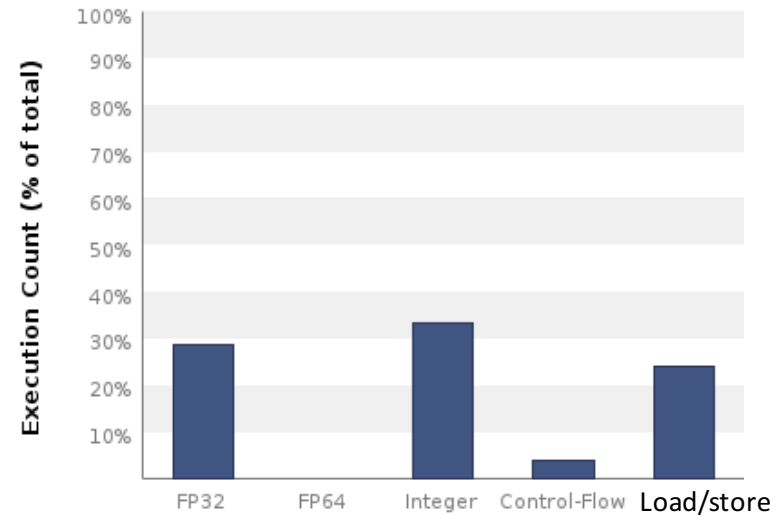
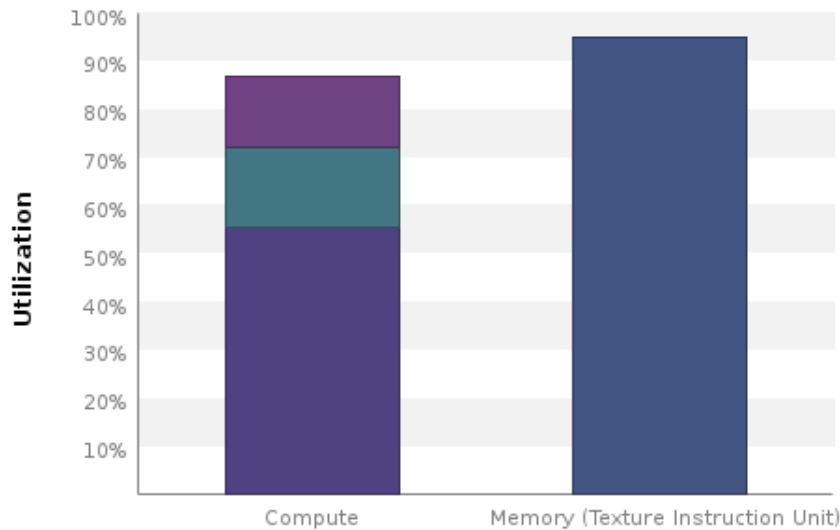
```
tmp=input[x+0]  
↓  
acc += filter[0]*tmp  
↓  
tmp=input[x+1]  
↓  
acc += filter[1]*tmp
```

#pragma unroll can help ILP
Create two accumulators
Or...

```
for (j=0;j<2;j++) {  
    acc0+=filter[j]*input[x+j];  
    acc1+=filter[j]*input[x+j+1];  
}  
tmp=input[x+0]  
↓  
acc0 += filter[0]*tmp  
↓  
tmp=input[x+1]  
↓  
acc0 += filter[1]*tmp  
↓  
acc1 += filter[1]*tmp  
↓  
tmp=input[x+0+1]  
↓  
acc1 += filter[0]*tmp  
↓  
tmp=input[x+1+1]  
↓  
acc1 += filter[1]*tmp
```

Process 2 points per thread
Bonus data re-use (register caching)

Iteration 5 – 2 points per thread



Initial unoptimised (v0)

8.122ms

Texture cache

1.53ms

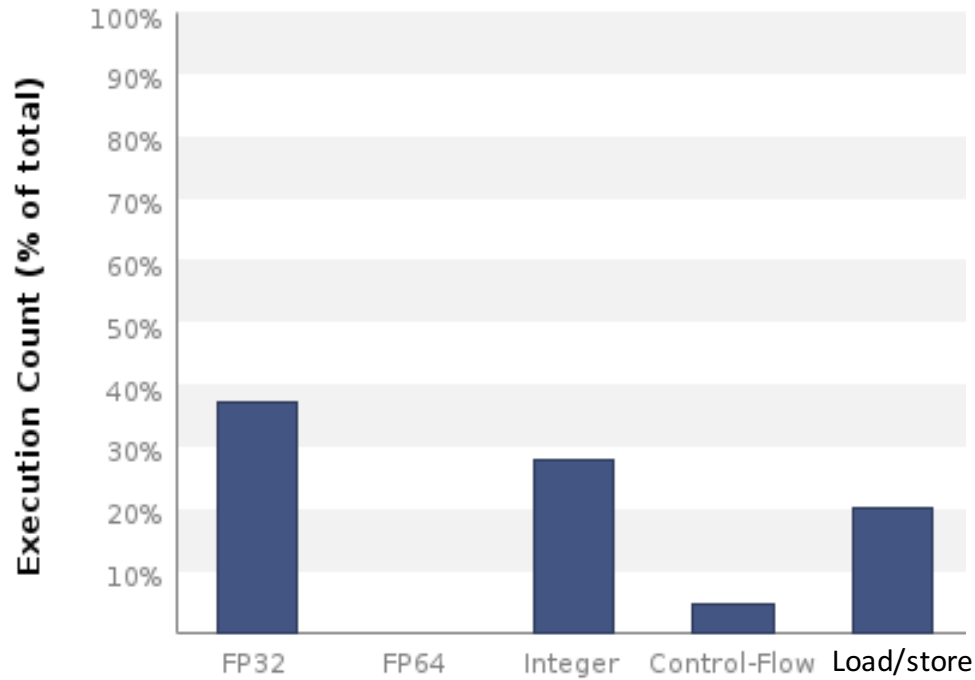
2 points

1.07ms

Key takeaway

- **Latency bound – low instruction level parallelism**
- Unused cycles, exposed latency
- Symptoms:
 - High execution dependency, one “pipe” saturated
- Goal:
 - Better utilise cycles by: increasing parallel work per thread
- What to do:
 - Increase ILP by having more independent work, e.g. more than 1 output value per thread
 - `#pragma unroll`

Iteration 6 – 4 points per thread



168 GB/s device BW

Initial unoptimised (v0)	8.122ms
2 points	1.07ms
4 points	0.95ms

Conclusions

- Iterative approach to improving a code's performance
 - Identify hotspot
 - Find performance limiter, understand why it's an issue
 - Improve your code
 - Repeat
- Managed to achieve a 8.5x speedup
- Shown how NVVP guides us and helps understand what the code does
- There is more it can show...

References: C. Angerer, J. Demouth, "CUDA Optimization with NVIDIA Nsight Eclipse Edition", GTC 2015

Metrics & Events

Device: Tesla K20c ▼

Metrics | **Events**

▼ ☐ Memory

- ☐ Requested Global Load Throughput
- ☐ Requested Global Store Throughput
- ☐ Device Memory Read Throughput
- ☐ Device Memory Write Throughput
- ☐ Global Store Throughput
- ☐ Global Load Throughput
- ☐ Shared Memory Efficiency
- ☐ Global Memory Load Efficiency
- ☐ Global Memory Store Efficiency
- ☐ Local Memory Overhead
- ☐ Requested Non-Coherent Global Load Throughput
- ☐ Local Memory Load Transactions Per Request
- ☐ Local Memory Store Transactions Per Request
- ☐ Shared Memory Load Transactions Per Request
- ☐ Shared Memory Store Transactions Per Request
- ☐ Global Load Transactions Per Request

Device: Tesla K20c ▼

Metrics | **Events**

▼ ☐ Instruction

- ☐ elapsed_cycles_sm
- ☐ warps launched
- ☐ threads launched
- ☐ Instructions executed
- ☐ Instructions issued 1
- ☐ Instructions issued 2
- ☐ thread inst executed
- ☐ active cycles
- ☐ active warps
- ☐ sm cta launched
- ☐ not_predicated_off_thread_inst_executed

▼ ☐ Memory

- ☐ fb subp0 read sectors
- ☐ fb subp1 read sectors
- ☐ fb subp0 write sectors