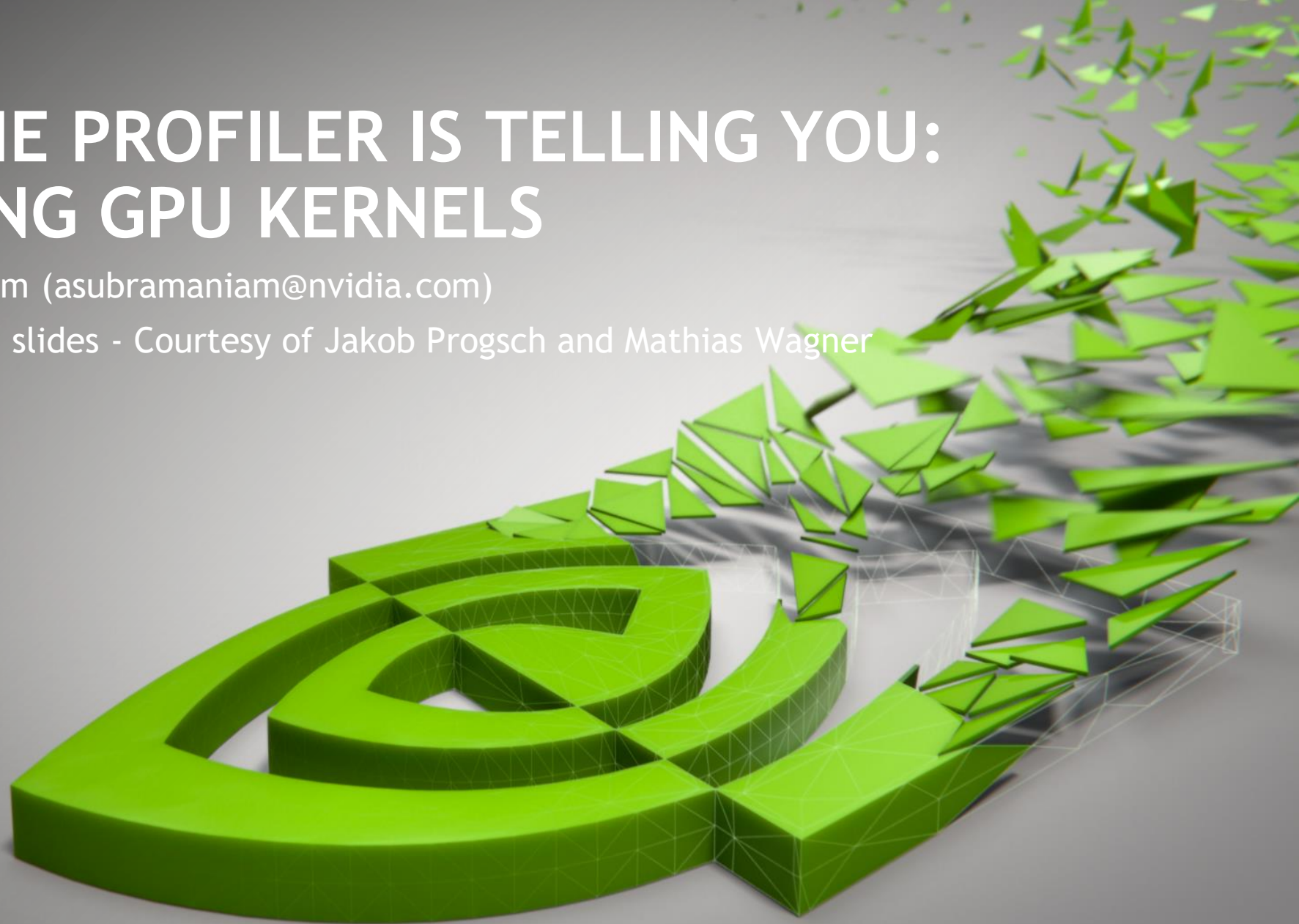


WHAT THE PROFILER IS TELLING YOU: OPTIMIZING GPU KERNELS

Akshay Subramaniam (asubramaniam@nvidia.com)

Including GTC 2018 slides - Courtesy of Jakob Progsch and Mathias Wagner

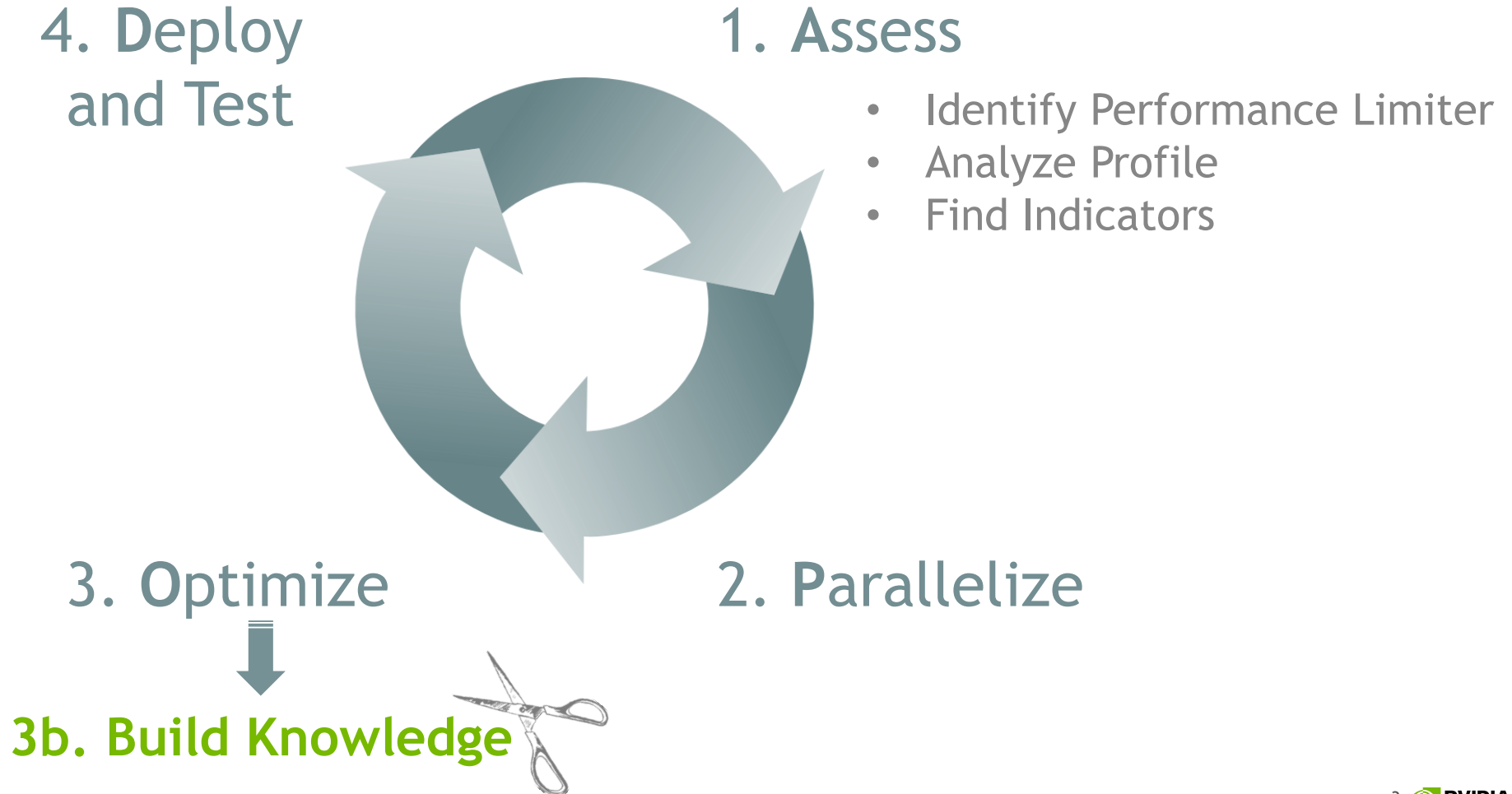


BEFORE YOU START

The five steps to enlightenment

1. Know your hardware
 - What are the target machines, how many nodes? Machine-specific optimizations okay?
2. Know your tools
 - Strengths and weaknesses of each tool? Learn how to use them (and learn one well!)
3. Know your application
 - What does it compute? How is it parallelized? What final performance is expected?
4. Know your process
 - Performance optimization is a constant learning process
5. Make it so!

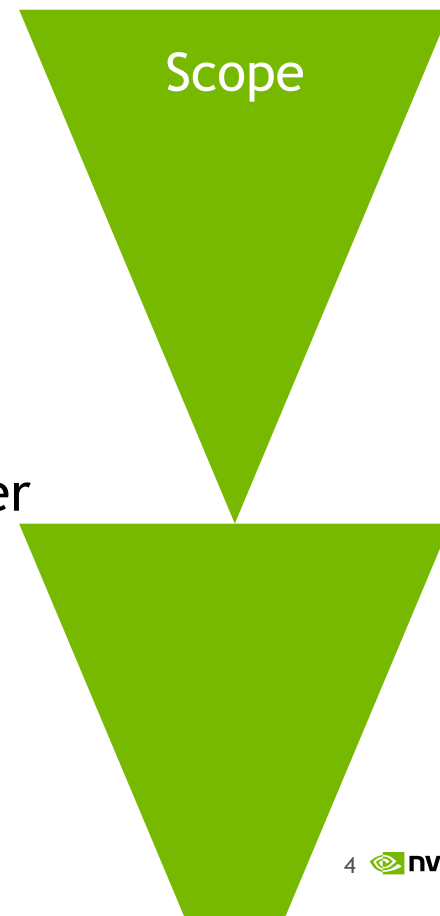
THE APOD CYCLE



GUIDING OPTIMIZATION EFFORT

“Drilling Down into the Metrics”

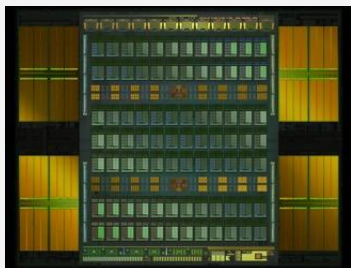
- Challenge: How to know where to start?
- Top-down Approach:
 - Find Hotspot Kernel
 - Identify Performance Limiter of the Hotspot
 - Find performance bottleneck indicators related to the limiter
 - Identify associated regions in the source code
 - Come up with strategy to fix and change the code
 - Start again



KNOW YOUR HARDWARE: VOLTA ARCHITECTURE

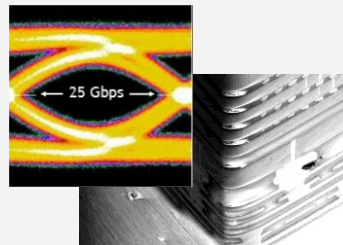
VOLTA V100 FEATURES

Volta Architecture



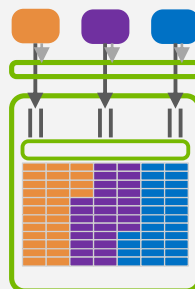
Most Productive GPU

Improved NVLink & HBM2



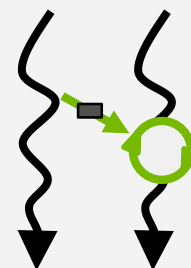
Efficient Bandwidth

Volta MPS



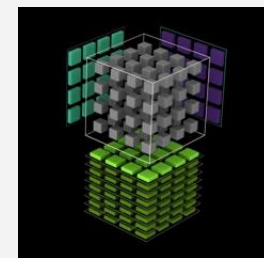
Inference Utilization

Improved SIMT Model



New Algorithms

Tensor Core



120 Programmable
TFLOPS Deep Learning

GPU COMPARISON

	P100 (SXM2)	V100 (SXM2)
Double/Single/Half TFlop/s	5.3/10.6/21.2	7.8/15.7/125 (TensorCores)
Memory Bandwidth (GB/s)	732	900
Memory Size	16GB	16GB
L2 Cache Size	4096 KB	6144 KB
Base/Boost Clock (Mhz)	1328/1480	1312/1530
TDP (Watts)	300	300

VOLTA SM

	GV100	GP100
FP32 Cores	64	64
INT32 Cores	64	0
FP64 Cores	32	32
Register File	256 KB	256 KB
Active Threads	2048	2048
Active Blocks	32	32

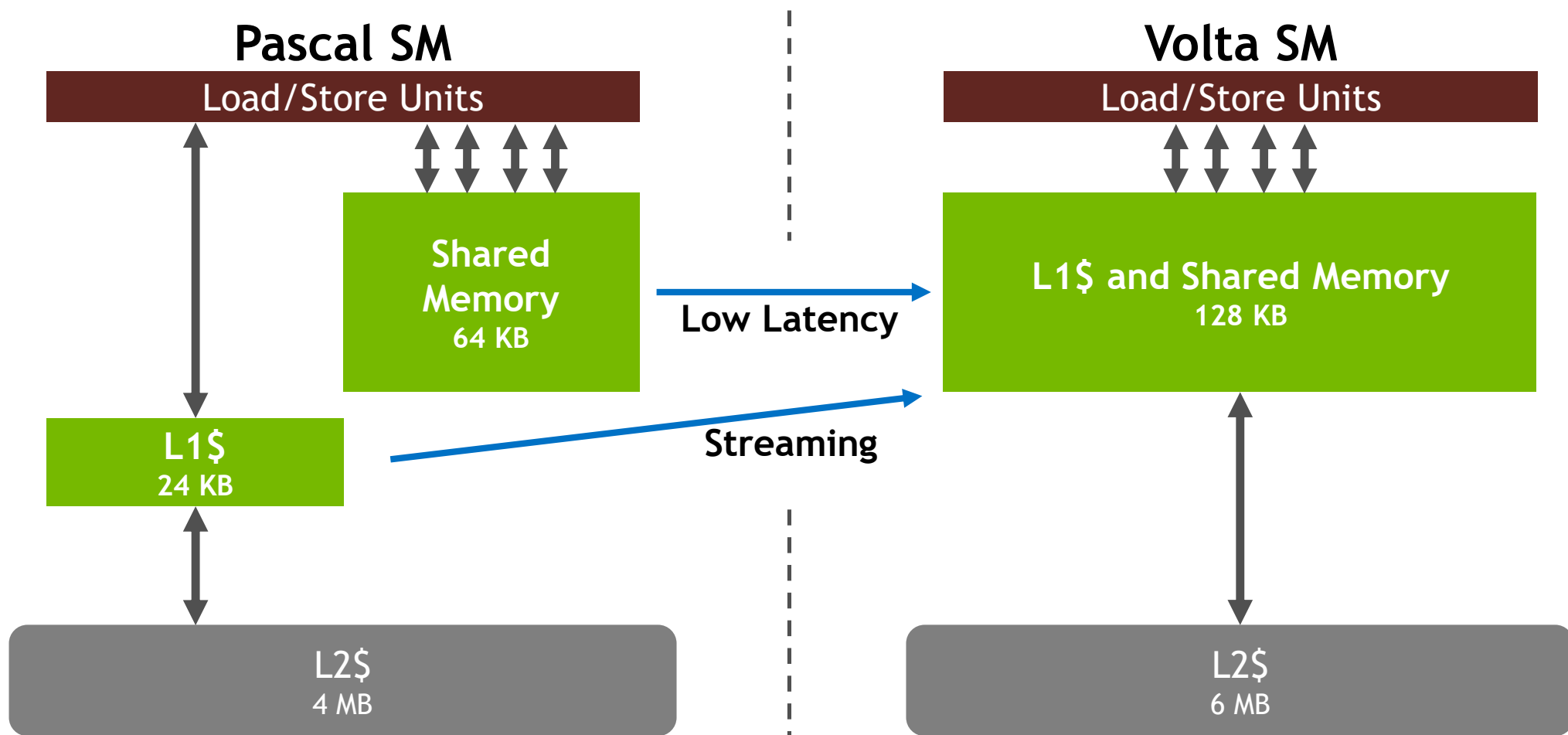
Same active threads/warps/blocks on SM

Same amount of registers

Expect similar occupancy, if not limited by shared mem.



IMPROVED L1 CACHE



INSTRUCTION LATENCY

Dependent instruction issue latency for core FMA operations:

Volta: 4 clock cycles

Pascal: 6 clock cycles

TENSOR CORE

Mixed precision multiplication and accumulation

Each Tensor Core performs 64 FMA mixed-precision operations per clock

$$\mathbf{D} = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 FP16 FP16 FP16 or FP32

TENSOR CORE

Example to use tensor core

cuBLAS/cuDNN: set TENSOR_OP_MATH

CUDA: nvcuda::wmma API

```
#include <mma.h>
using namespace nvcuda;
__global__ void wmma_ker(half *a, half *b, float *c) {
    wmma::fragment<wmma::matrix_a, 16, 16, 16, half, wmma::col_major> a_frag;
    wmma::fragment<wmma::matrix_b, 16, 16, 16, half, wmma::row_major> b_frag;
    wmma::fragment<wmma::accumulator, 16, 16, 16, float> c_frag;

    wmma::fill_fragment(c_frag, 0.0f);
    wmma::load_matrix_sync(a_frag, a, 16);
    wmma::load_matrix_sync(b_frag, b, 16);

    wmma::mma_sync(c_frag, a_frag, b_frag, c_frag);

    wmma::store_matrix_sync(c, c_frag, 16, wmma::mem_row_major);
}
```

KNOW YOUR TOOLS: PROFILERS

PROFILING TOOLS

Many Options!

From NVIDIA

- nvprof
- NVIDIA Visual Profiler (nvvp)
- Nsight Visual Studio Edition

Volta, Turing and future:

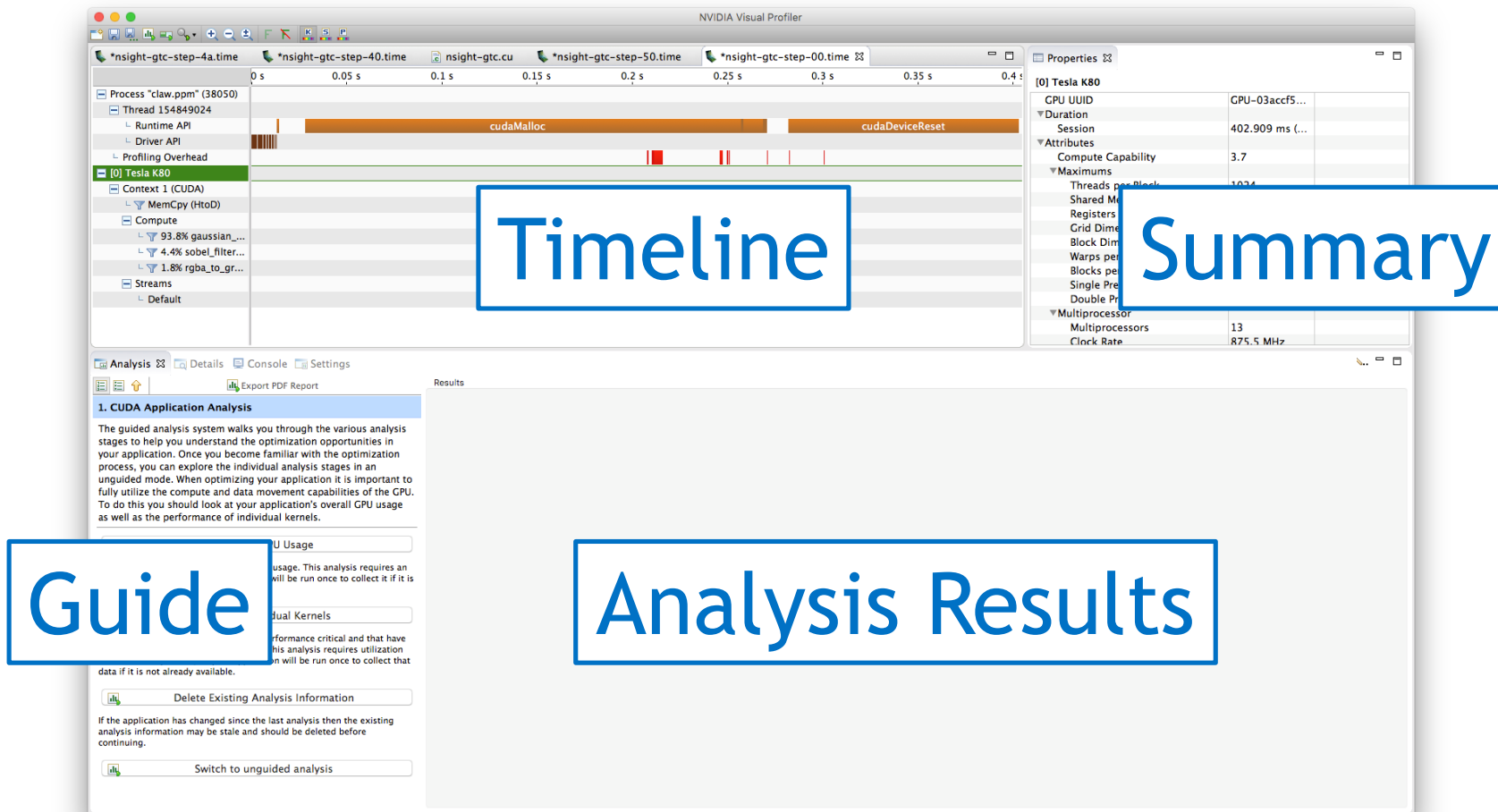
- NVIDIA Nsight Systems
- NVIDIA Nsight Compute

Third Party

- TAU Performance System
- VampirTrace
- PAPI CUDA component
- HPC Toolkit
- (Tools using CUPTI)

Without loss of generality, in this talk we will be showing nvvp screenshots

THE NVVP PROFILER WINDOW



PROFILING ON A REMOTE SYSTEM

Get timeline

```
dt01 asubramaniam ... > NVIDIA_CUDA-10.0_Samples > 0_Simple > matrixMul > nvprof -o matrixMul.timeline ./matrixMul
[Matrix Multiply Using CUDA] - Starting...
==10502== NVPROF is profiling process 10502, command: ./matrixMul
GPU Device 0: "Tesla K80" with compute capability 3.7

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
done
Performance= 228.00 GFlop/s, Time= 0.575 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
==10502== Generated result file: /home/asubramaniam/Codes/NVIDIA_CUDA-10.0_Samples/0_Simple/matrixMul/matrixMul.timeline
dt01 asubramaniam ... > NVIDIA_CUDA-10.0_Samples > 0_Simple > matrixMul
```


PROFILING ON A REMOTE SYSTEM

Get metrics

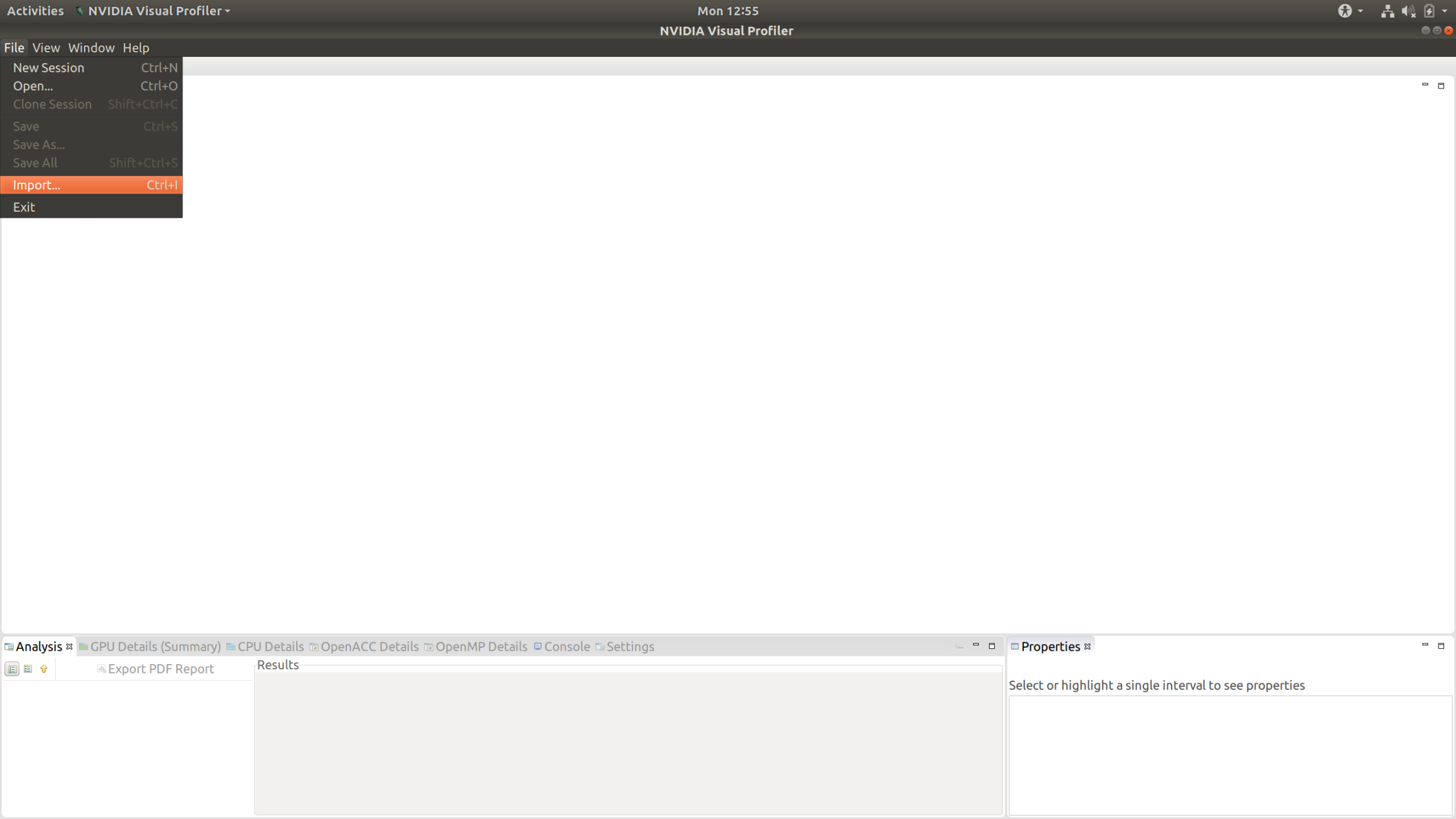
```
dt01 asubramaniam ... > NVIDIA_CUDA-10.0_Samples > 0_Simple > matrixMul > nvprof --analysis-metrics -o matrixMul.me
trics ./matrixMul
[Matrix Multiply Using CUDA] - Starting...
==10748== NVPROF is profiling process 10748, command: ./matrixMul
GPU Device 0: "Tesla K80" with compute capability 3.7

MatrixA(320,320), MatrixB(640,320)
Computing result using CUDA Kernel...
==10748== Some kernel(s) will be replayed on device 0 in order to collect all events/metrics.
==10748== Warning: PC Sampling is not supported on the underlying platform.
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
done    l2_subpl_read_sysmem_sector_queries
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Replaying kernel "void MatrixMulCUDA<int=32>(float*, float*, float*, int, int)" (done)
Performance= 0.09 GFlop/s, Time= 1497.165 msec, Size= 131072000 Ops, WorkgroupSize= 1024 threads/block
Checking computed result for correctness: Result = PASS

NOTE: The CUDA Samples are not meant for performance measurements. Results may vary when GPU Boost is enabled.
==10748== Generated result file: /home/asubramaniam/Codes/NVIDIA_CUDA-10.0_Samples/0_Simple/matrixMul/matrixMul.metric
s
dt01 asubramaniam ... > NVIDIA_CUDA-10.0_Samples > 0_Simple > matrixMul
```

PROFILING ON A REMOTE SYSTEM

Copy to local machine, import in nvvp





Import

Select

Import profile data generated by nvprof.

Select an import source:

type filter text

Command-line Profiler

Nvprof

< Back

Next >

Cancel

Finish



Select or highlight a single interval to see properties



Import Nvprof Data

Nvprof profile files

Import profile data for a single process or for multiple processes

- ☒ Single process
☐ Multiple processes

< Back

Next >

Cancel

Finish

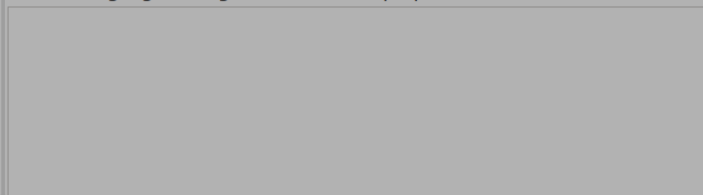
Analysis GPU Details (Summary) CPU Details Open



Export PDF Report

Results

Select or highlight a single interval to see properties



Import Nvprof Data

Import Profile Data for Single Process

Select one nvprof profile file containing timeline data and zero or more addition nvprof profile files containing event and metric values.

Profile Files

Timeline Options

Connection: Local Manage connections...

Timeline data file: /home/akshays/matrixMul.timeline Browse...

Event/Metric data files:

/home/akshays/matrixMul.metrics Browse... Remove

Kernel scopes:

Add the nvprof scope used for analysis in the format <context id/name>:<stream id/name>:<kernel name>:<invocation> Add

Remove

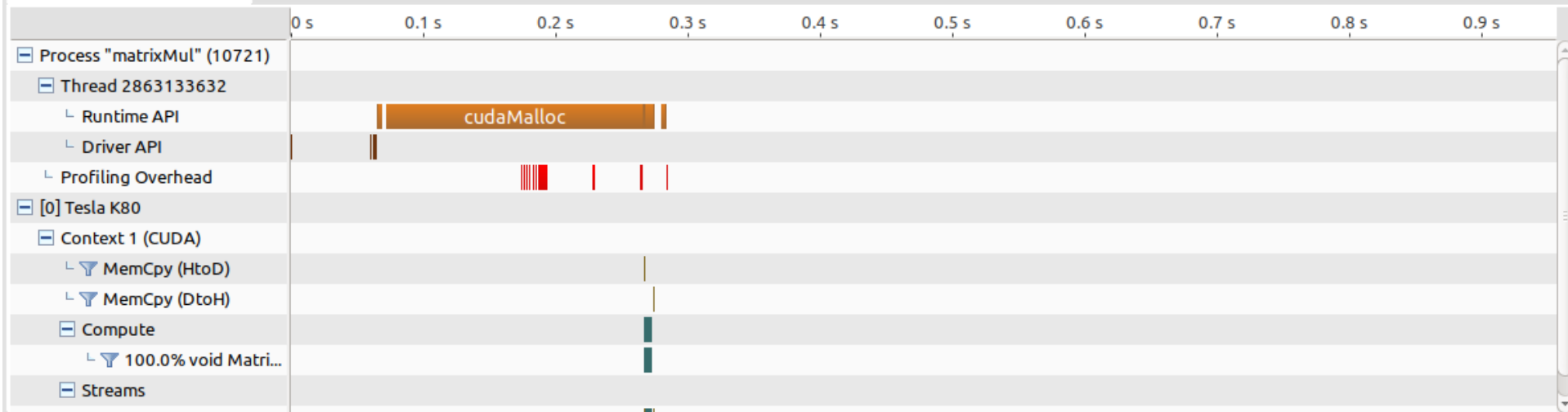
☒ Use fixed width segments for Unified memory timeline

Number of segments Specify the number of segments for unified memory timelines [default 100]

< Back Next > Cancel Finish



*matrixMul.timeline



Analysis GPU Details (Summary) CPU Details OpenACC Details OpenMP Details Console Settings

Export PDF Report

1. CUDA Application Analysis

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

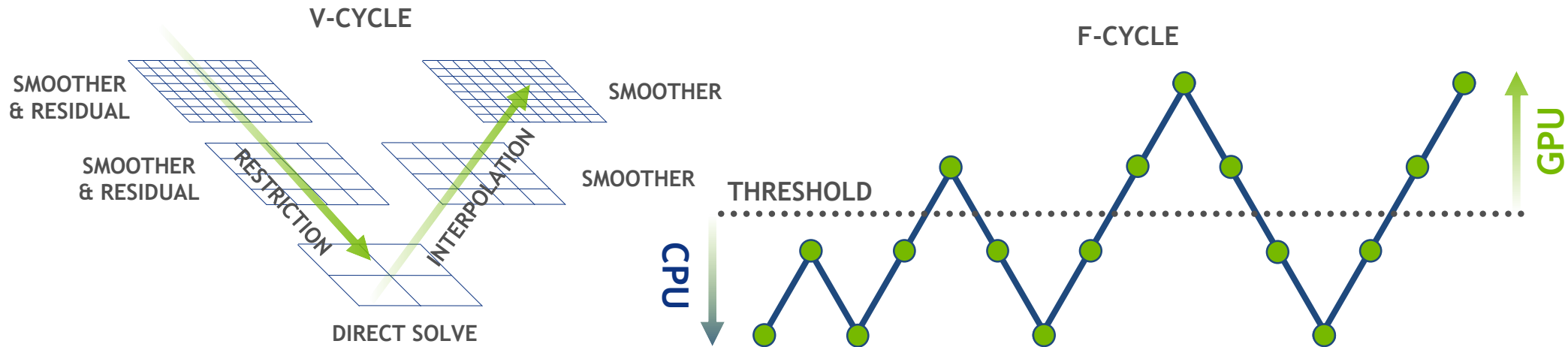
Results

Select or highlight a single interval to see properties

**KNOW YOUR APPLICATION:
HPGMG**

HPGMG

High-Performance Geometric Multi-Grid, Hybrid Implementation



Fine levels are executed on throughput-optimized processors (GPU)

Coarse levels are executed on latency-optimized processors (CPU)

<http://crd.lbl.gov/departments/computer-science/PAR/research/hpgmg/>

**MAKE IT SO:
ITERATION 1
2ND ORDER 7-POINT STENCIL**

IDENTIFY HOTSPOT

Hotspot



Results	
i Kernel Optimization Priorities	
The following kernels are ordered by optimization importance based on execution time and achieved occupancy. Optimization of higher r	
Rank	Description
100	[131 kernel instances] void smooth_kernel<int=8, int=8, int=4, int=8>(level_type, int, int, double, double, int, double*, double*)
70	[6 kernel instances] void copy_block_kernel<int=8, int=1>(level_type, int, communicator_type)
58	[3 kernel instances] zero_vector_kernel(level_type, int)
28	[263 kernel instances] void smooth_kernel<int=7, int=8, int=4, int=8>(level_type, int, int, double, double, int, double*, double*)
20	[1 kernel instances] color_vector_kernel(level_type, int, int, int, int, int)
20	[1 kernel instances] void reduction_kernel<int=1>(level_type, int, double*)

Identify the hotspot: smooth_kernel()

Kernel	Time	Speedup
Original Version	2.079ms	1.00x

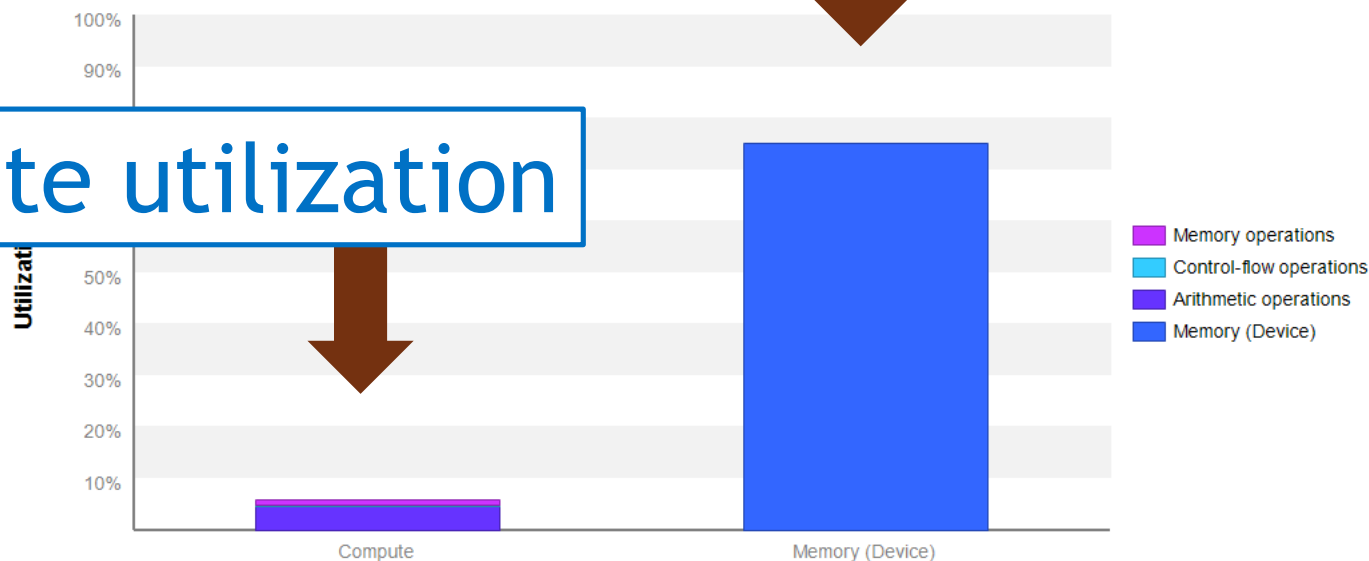
IDENTIFY PERFORMANCE LIMITER

i Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla V100-PCIE-16GB" the kernel's compute utilization is significantly low. The limiting factor in the memory system is the bandwidth of the Device memory.

Memory utilization

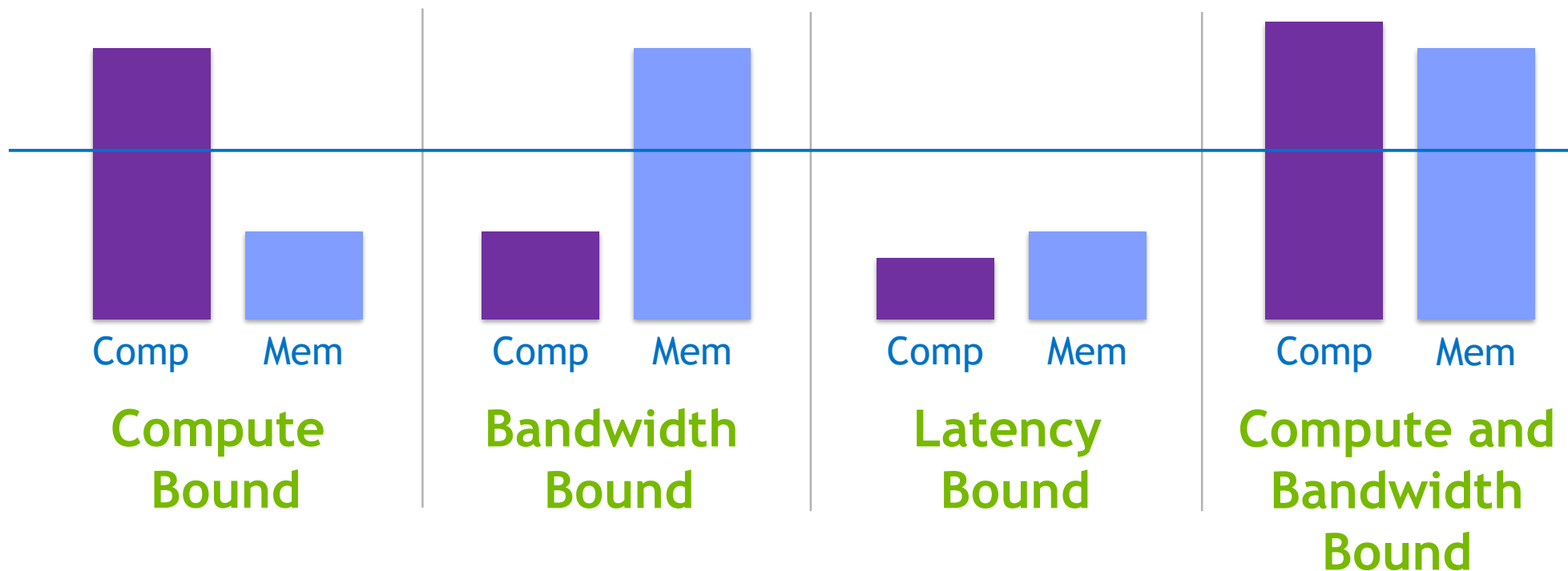
Compute utilization



PERFORMANCE LIMITER CATEGORIES

Memory Utilization vs Compute Utilization

Four possible combinations:

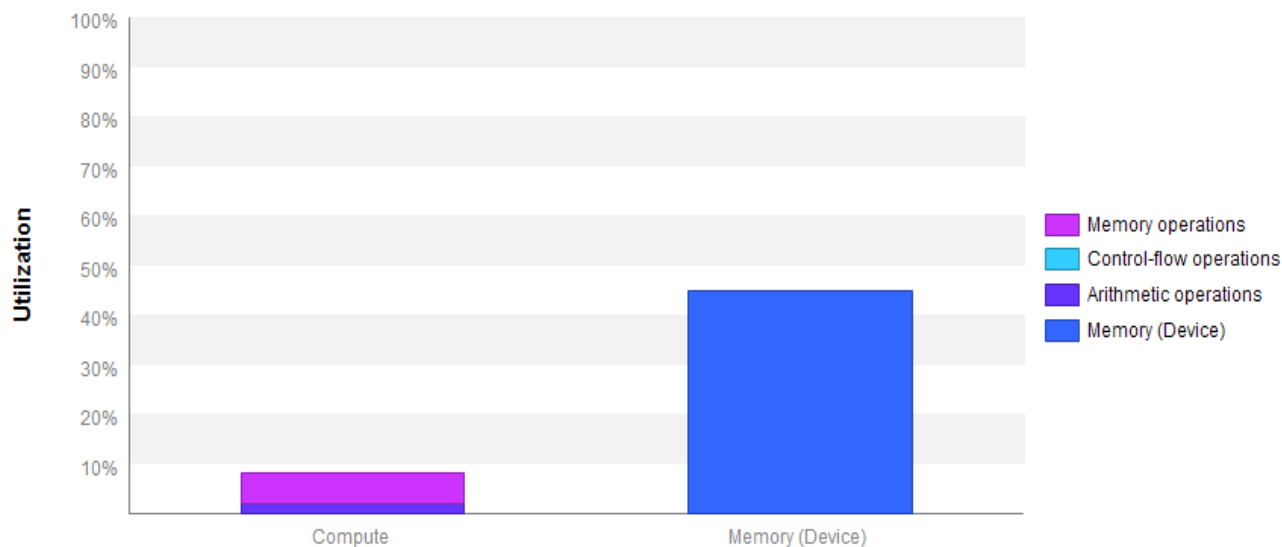


LATENCY BOUND ON P100

Results

i Kernel Performance Is Bound By Instruction And Memory Latency

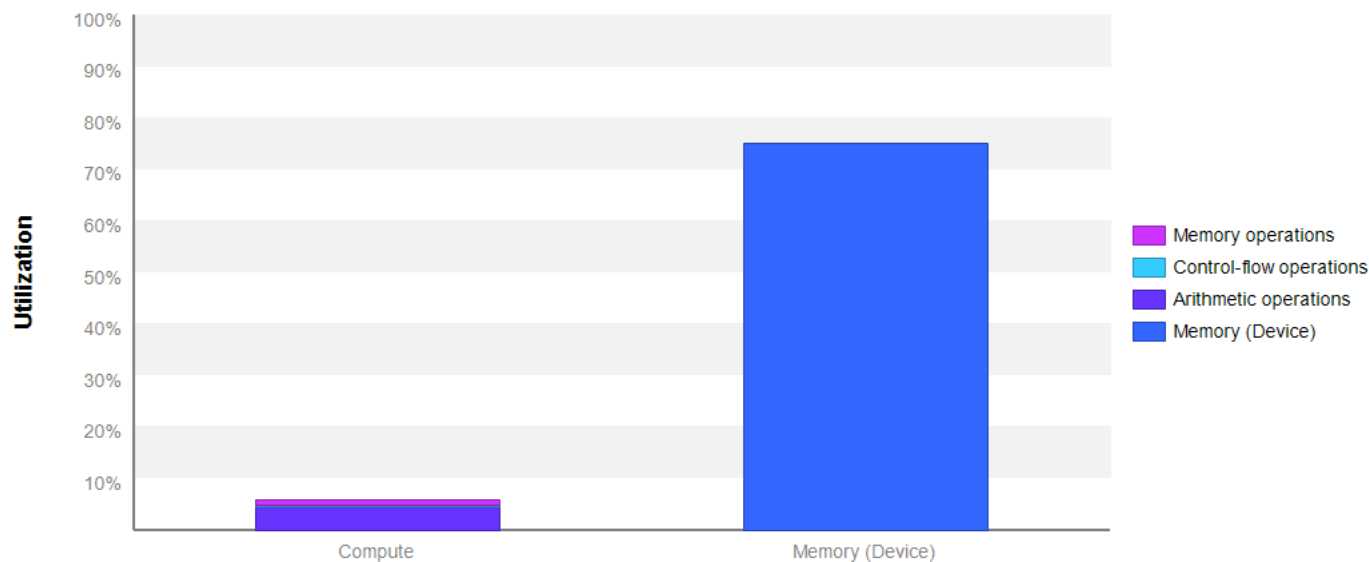
This kernel exhibits low compute throughput and memory bandwidth utilization relative to the peak performance of "Tesla P100-PCIE-16GB". These utilization levels indicate that the performance of the kernel is most likely limited by the latency of arithmetic or memory operations. Achieved compute throughput and/or memory bandwidth below 60% of peak typically indicates latency issues.



BANDWIDTH BOUND ON V100

i Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla V100-PCIE-16GB" 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 Device memory.

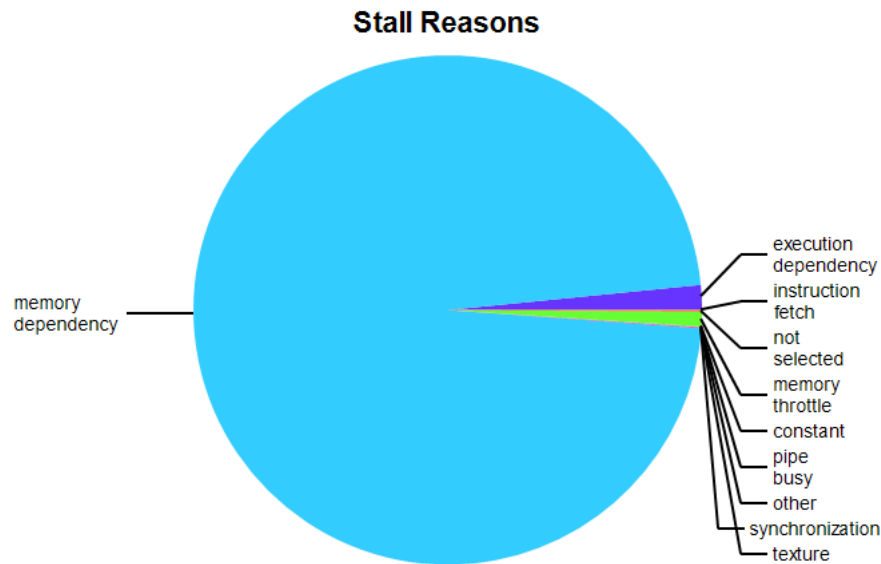


DRILLING DOWN: LATENCY ANALYSIS (V100)

The profiler warns about low occupancy

Grid Size	[65536,...
Block Size	[8,4,1]
Registers/Thread	48
Shared Memory/Block	0 B
Launch Type	Normal
Occupancy	
Achieved	⚠ 49.6%
Theoretical	50%
Limiter	Block Size

Limited by block size of only $8 \times 4 = 32$ threads



⚠ GPU Utilization May Be Limited By Block Size

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.

OCCUPANCY

GPU Utilization

Each SM has limited resources:

- max. 64K Registers (32 bit) distributed between threads
- max. 48KB of shared memory per block (96KB per SMM)
- max. 32 Active Blocks per SMM
- Full occupancy: 2048 threads per SM (64 warps)

When a resource is used up, occupancy is reduced

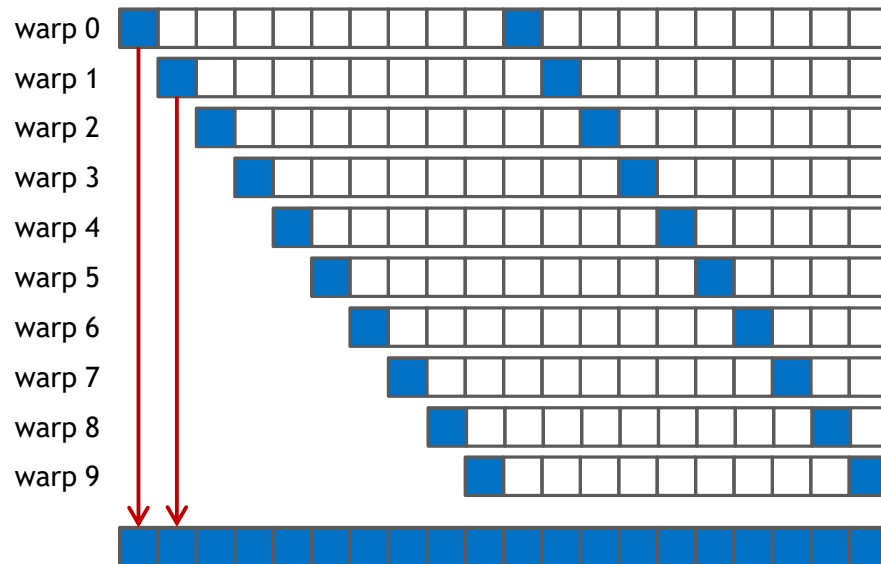
() Values vary with Compute Capability*

LATENCY

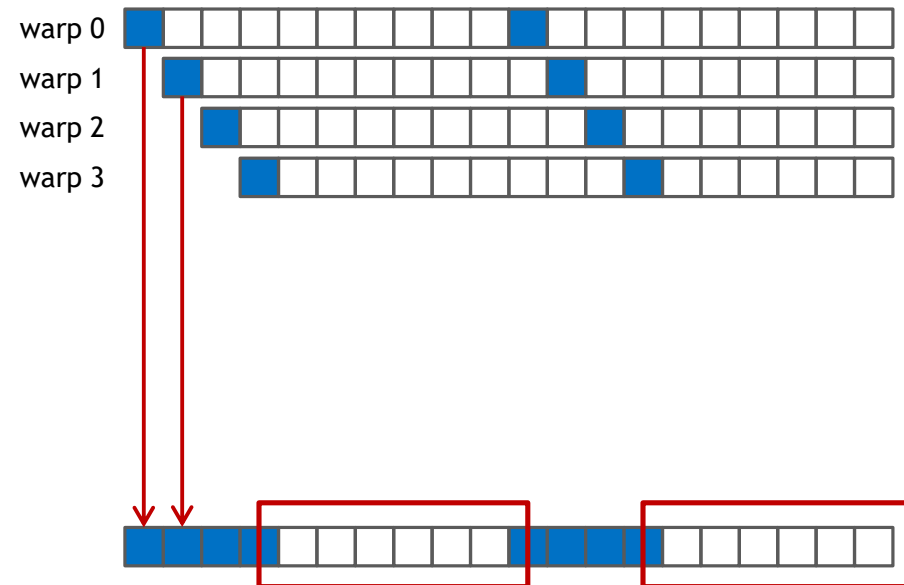
GPUs cover latencies by having a lot of work in flight

- The warp issues
- The warp waits (latency)

Fully covered latency



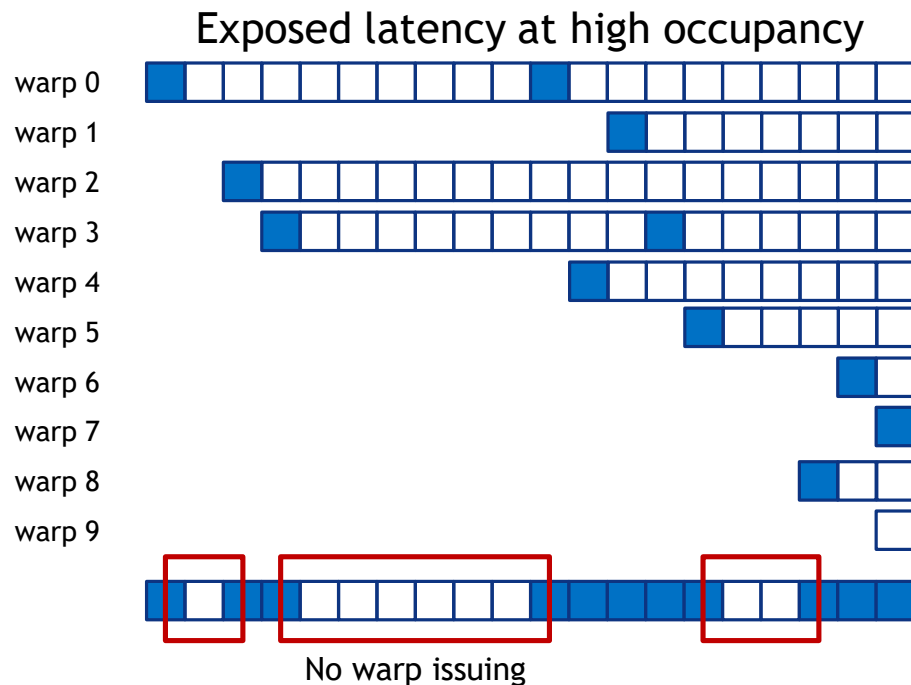
Exposed latency, not enough warps



No warp issues

LATENCY AT HIGH OCCUPANCY

Many active warps but with high latency instructions



GLOBAL MEMORY

HBM2 increase bandwidth from 732 GB/s to 900 GB/s

Basic optimization is the same: Coalescing, Alignment, SOA pattern.

Granularity is 32 bytes, i.e. 8 threads are accessing a continuous 32 byte space.













Latency: what is the occupancy we need to saturate global load/store?

One V100:

$BW = 4096 \text{ bit} * 877\text{Mhz} * 2 / 8 = 898 \text{ GB/s} \sim 1.23x \text{ of P100 (theoretical)}$

SM ratio: $80/56 = 1.43x \text{ of P100}$

LOOKING FOR MORE INDICATORS

Line	Global Access	File - /C:/Users/jprogsch/Documents/GTC18/stencils/smooth.bas	Global Access	Disassembly
87	 	const double Ax = apply_op_ijk0;	 	LDG.E.64.SYS R20, [R30];
88			 	LDG.E.64.SYS R24, [R30+-0x8];
89			 	LDG.E.64.SYS R22, [R26];
90		////////// SMOOTHER //////////	 	LDG.E.64.SYS R30, [R30+0x8];
91		#ifdef USE_CHEBY	 	LDG.E.64.SYS R26, [R26+0x8];
92		const double lambda = Dinv_ijk();		DADD R24, -R20, R24;
93		xo[ijk] = X(ijk) + c1*(X(ijk)-xp[ijk]) + c2*lambda*(rhs[ijk]-Ax);		DMUL R22, R24, R22;
94				IADD3 R25, P0, R33.reuse, R38, RZ;
95				DADD R28, -R20, R30;
96		#elif USE_JACOBI		LEA.HI.X.SX32 R42, R33, R35, 0x1, P0;
97		const double lambda = Dinv_ijk();		IMAD.U32 R24, R25, 0x8, RZ;
98		xo[ijk] = X(ijk) + (0.66666666666666666667)*lambda*(rhs[SHF.L.U64.HI R25, R25, 0x3, R42;
99				DFMA R22, R28, R26, R22;

Source Code Association

🔒 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. The analysis is per assembly instruction.

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	smooth.base.h - \home\jprogsch\hpgmg_cuda\finite-volume\source\cuda\stencils
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 12, Ideal Transactions/Access = 8 [6291456 L2 transactions for 524288 total executions]

12 Global Load

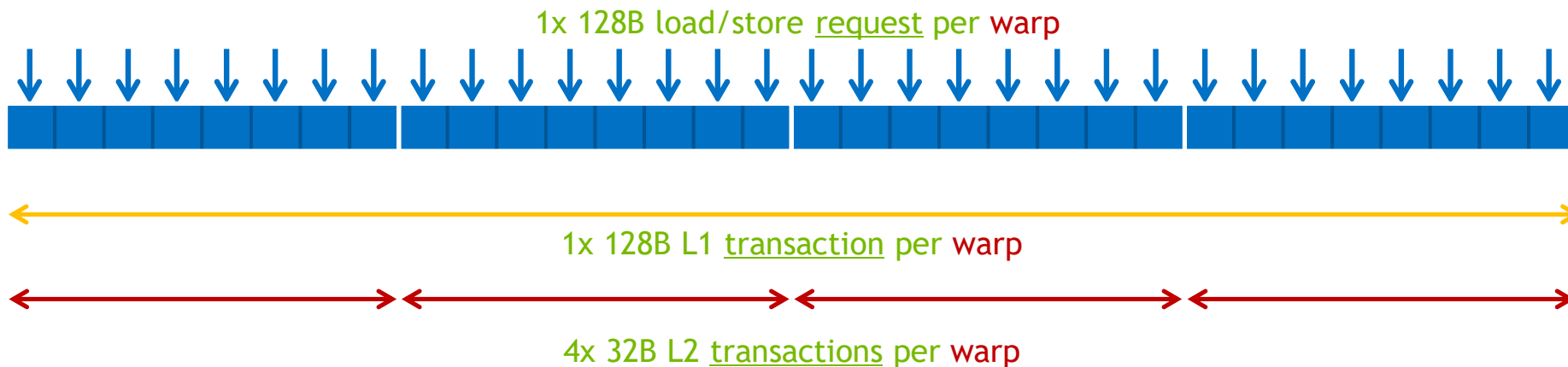
For line numbers use:
`nvcc -lineinfo`

12 Global Load Transactions per 1 Request

MEMORY TRANSACTIONS: BEST CASE

A warp issues 32x4B aligned and consecutive load/store request

Threads read different elements of the same 128B segment



1x L1 transaction: 128B needed / 128B transferred

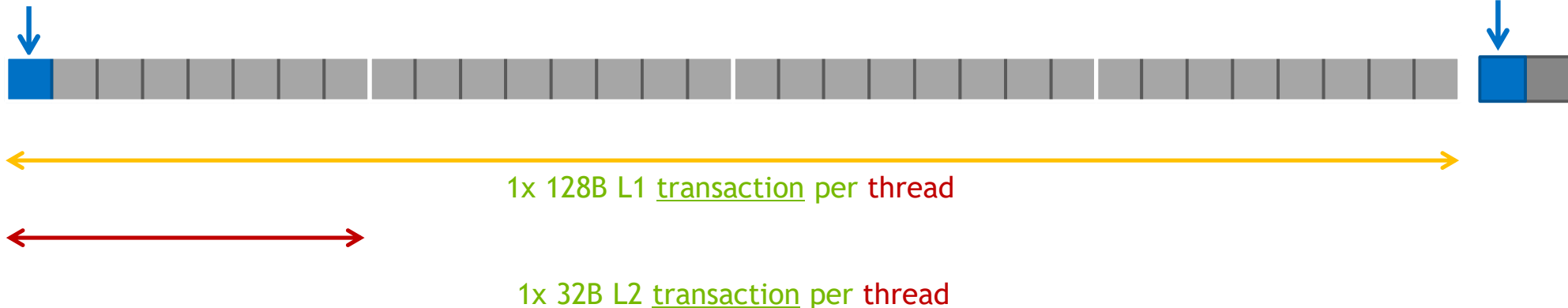
4x L2 transactions: 128B needed / 128B transferred

MEMORY TRANSACTIONS: WORST CASE

Threads in a warp read/write 4B words, 128B between words

Each thread reads the first 4B of a 128B segment

Stride: 32x4B



32x L1 transactions: 128B needed / **32x** 128B transferred

32x L2 transactions: 128B needed / **32x** 32B transferred

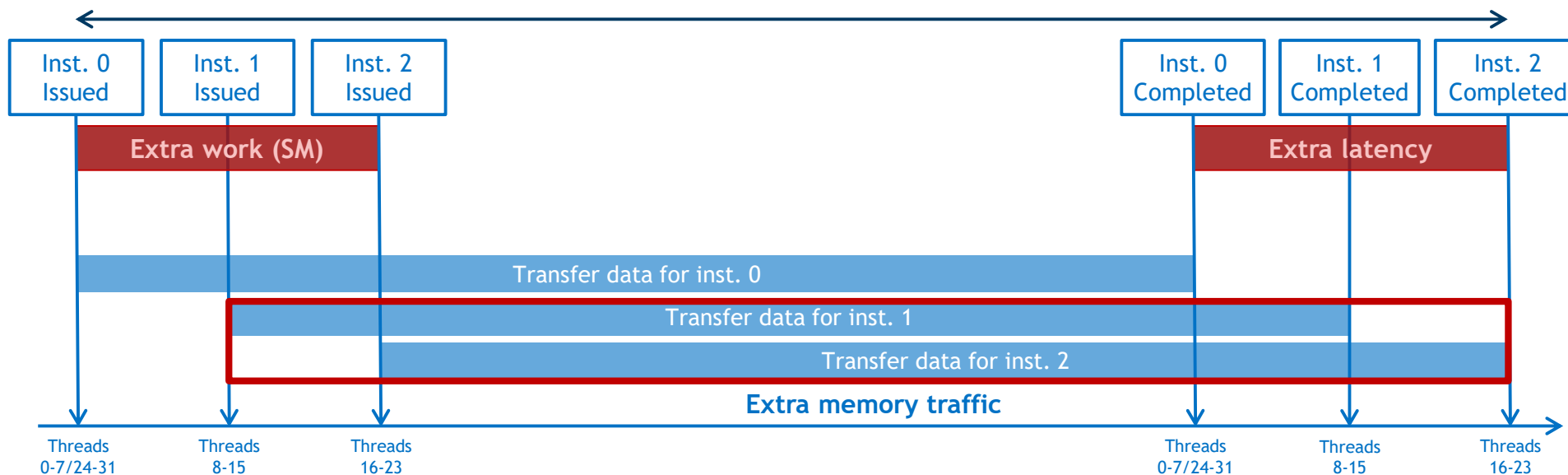
TRANSACTIONS AND REPLAYS

With replays, requests take more time and use more resources

More instructions issued

More memory traffic

Increased execution time



FIX: BETTER GPU TILING

Block Size Up

Before

Grid Size	[65536,1,1]
Block Size	[8,4,1]

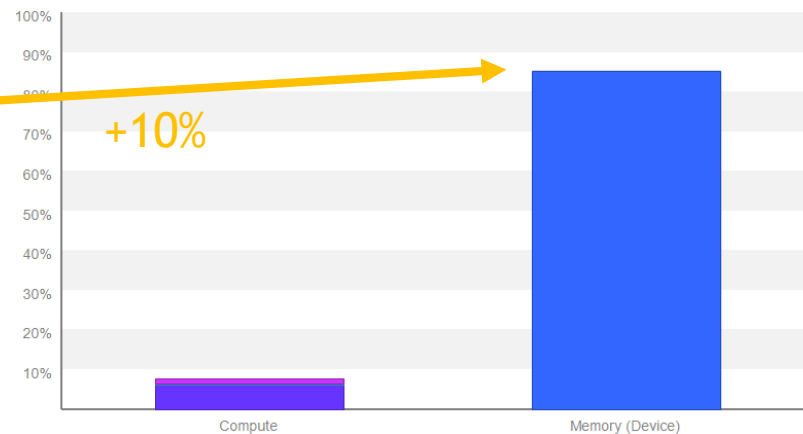
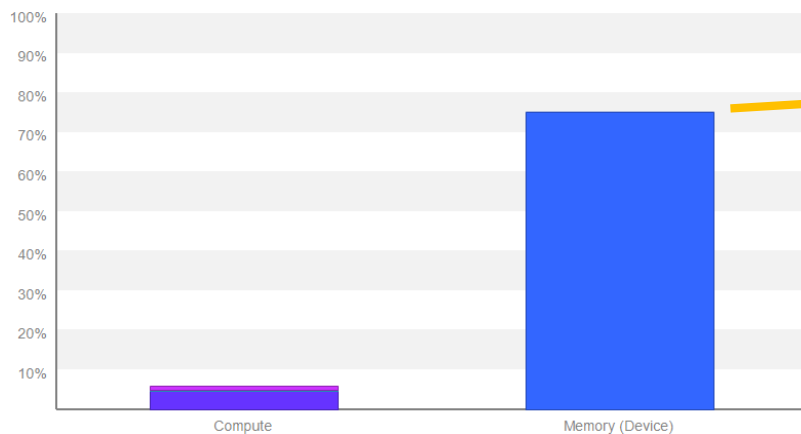
After

Grid Size	[16384,1,1]
Block Size	[32,4,1]

Transactions
Per Access
Down

87	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [4718592 L2 transactions for 524288 total executions]
87	Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [4718592 L2 transactions for 524288 total executions]

Memory
Utilization Up



Kernel	Time	Speedup
Original Version	2.079ms	1.00x
Better Memory Accesses	1.756ms	1.18x

PERF-OPT QUICK REFERENCE CARD

Category:	Latency Bound - Occupancy
Problem:	Latency is exposed due to low occupancy
Goal:	<u>Hide</u> latency behind more parallel work
Indicators:	Occupancy low (< 60%) Execution Dependency High
Strategy:	Increase occupancy by: <ul style="list-style-type: none">• Varying block size• Varying shared memory usage• Varying register count (use <code>__launch_bounds</code>)



PERF-OPT QUICK REFERENCE CARD

Category:	Latency Bound - Coalescing
Problem:	Memory is accessed inefficiently => high latency
Goal:	Reduce #transactions/request to reduce latency
Indicators:	Low global load/store efficiency, High #transactions/#request compared to ideal
Strategy:	Improve memory coalescing by: <ul style="list-style-type: none">• Cooperative loading inside a block• Change block layout• Aligning data• Changing data layout to improve locality



PERF-OPT QUICK REFERENCE CARD

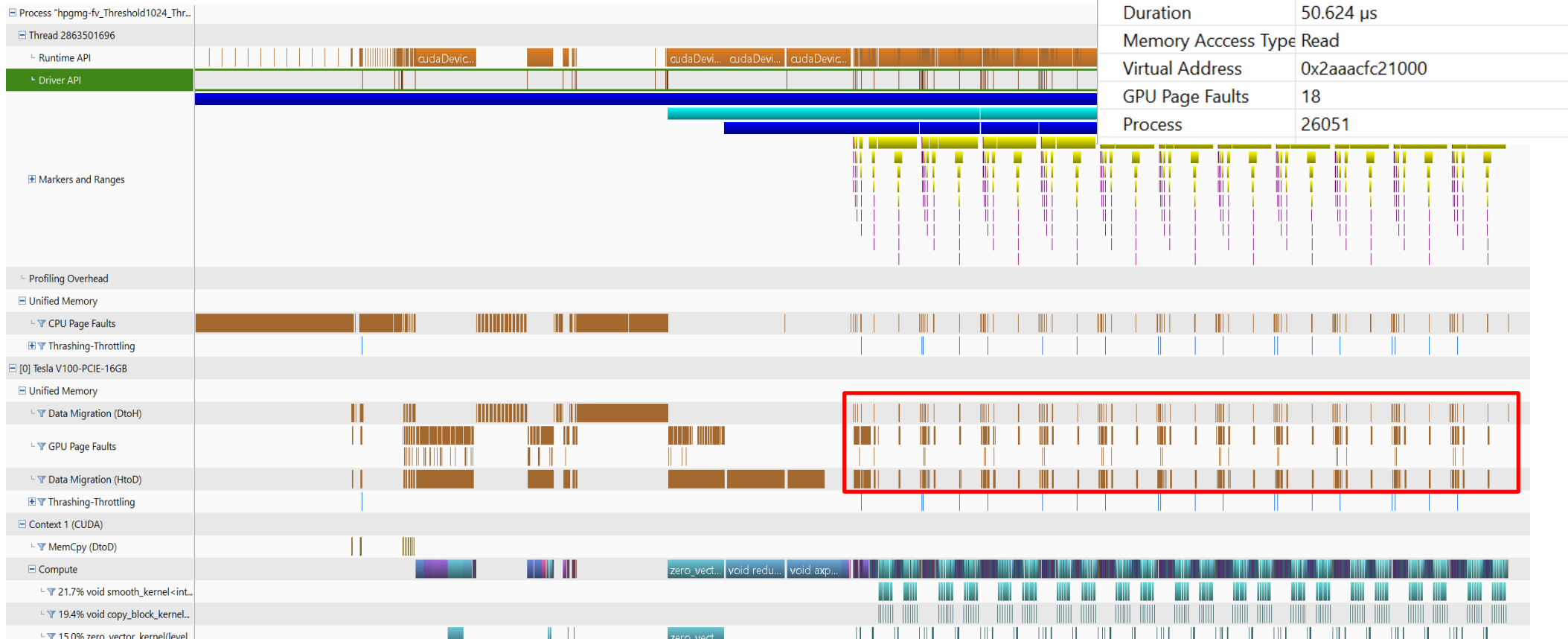
Category:	Bandwidth Bound - Coalescing
Problem:	Too much unused data clogging memory system
Goal:	Reduce traffic, move more <u>useful</u> data per request
Indicators:	Low global load/store efficiency, High #transactions/#request compared to ideal
Strategy:	Improve memory coalescing by: <ul style="list-style-type: none">• Cooperative loading inside a block• Change block layout• Aligning data• Changing data layout to improve locality



ITERATION 2: DATA MIGRATION

PAGE FAULTS

Details



MEMORY MANAGEMENT

Using Unified Memory

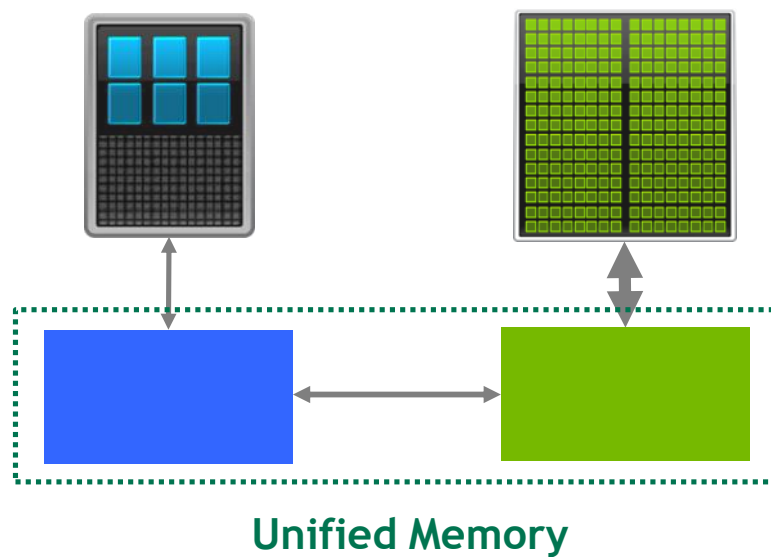
No changes to data structures

No explicit data movements

Single pointer for CPU and GPU data

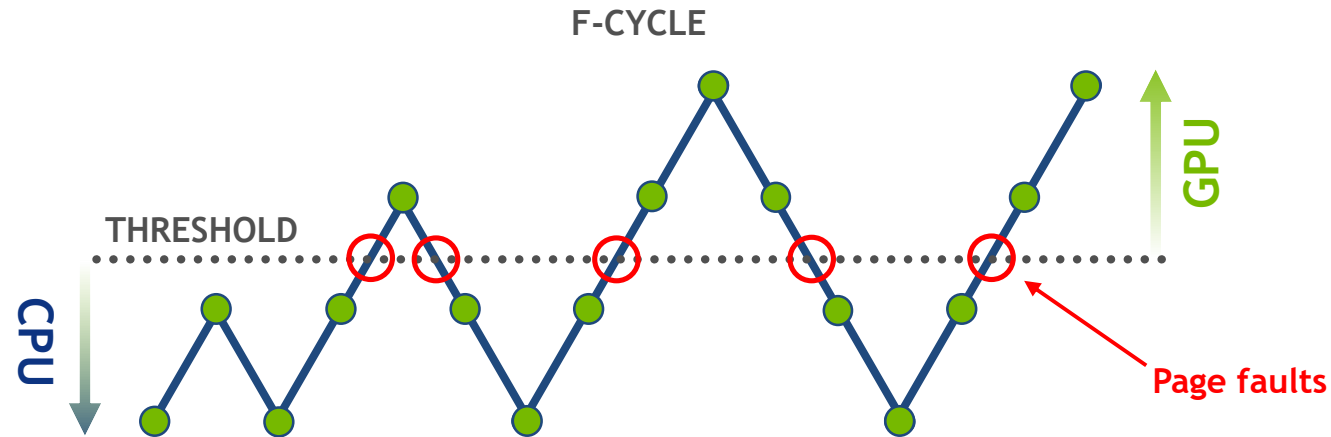
Use `cudaMallocManaged` for allocations

Developer View With Unified Memory



UNIFIED MEMORY

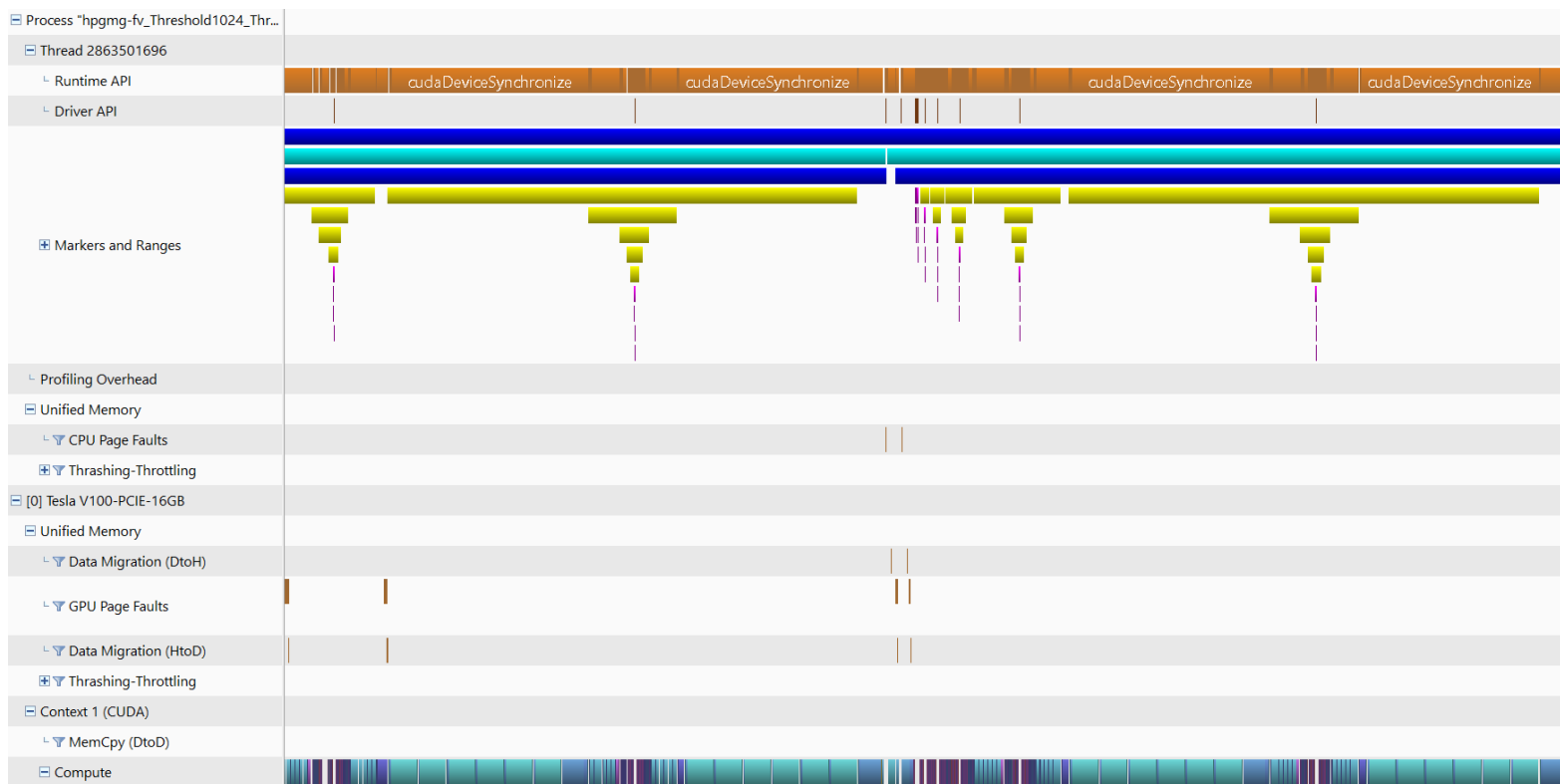
Eliminating page migrations and faults



Solution: allocate the first CPU level with `cudaMallocHost` (zero-copy memory)

PAGE FAULTS

Almost gone



PAGE FAULTS

Significant speedup for affected kernel

void interpolation_v2_kernel<int=4, int=1>(level_type, i

Queued	n/a
Submitted	n/a
Start	31.40143 s (31,401,428,910 ns)
End	31.40157 s (31,401,570,860 ns)
Duration	141.95 µs
Stream	Default
Grid Size	[1,1,8]
Block Size	[8,4,1]
Registers/Thread	56
Shared Memory/Block	0 B
Launch Type	Normal
▼ Occupancy	
Theoretical	50%
▼ Shared Memory Config	
Shared Memory Required	0 B
Shared Memory Exceeded	96 KiB
Shared Memory Barrier	4 B

void interpolation_v2_kernel<int=4, int=1>(level_type, i

Queued	n/a
Submitted	n/a
Start	31.16898 s (31,168,980,770 ns)
End	31.16901 s (31,169,011,713 ns)
Duration	30.943 µs
Stream	Default
Grid Size	[1,1,8]
Block Size	[8,4,1]
Registers/Thread	56
Shared Memory/Block	0 B
Launch Type	Normal
▼ Occupancy	
Theoretical	50%
▼ Shared Memory Config	
Shared Memory Required	0 B
Shared Memory Exceeded	96 KiB
Shared Memory Barrier	4 B

MEM ADVICE API

Not used here

`cudaMemPrefetchAsync(ptr, length, destDevice, stream)`

Migrate data to destDevice: overlap with compute

Update page table: much lower overhead than page fault in kernel

Async operation that follows CUDA stream semantics

`cudaMemAdvise(ptr, length, advice, device)`

Specifies allocation and usage policy for memory region

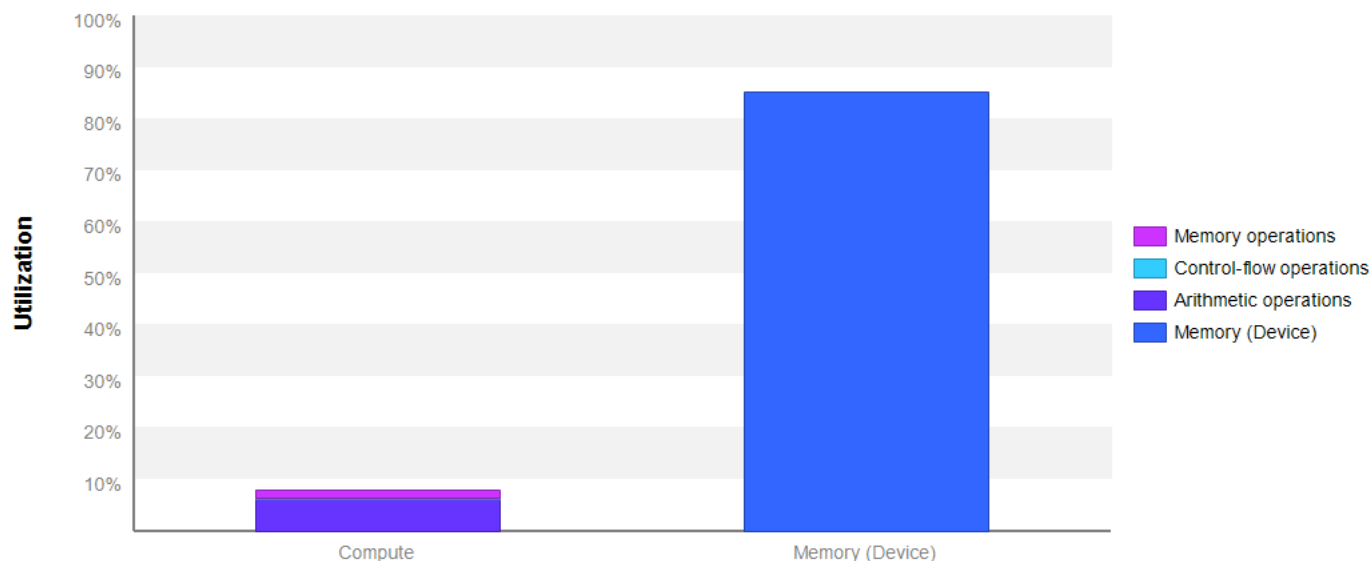
User can set and unset at any time

ITERATION 3: REGISTER OPTIMIZATION AND CACHING


LIMITER: STILL MEMORY BANDWIDTH

i Kernel Performance Is Bound By Memory Bandwidth

For device "Tesla V100-PCIe-16GB" 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 Device memory.

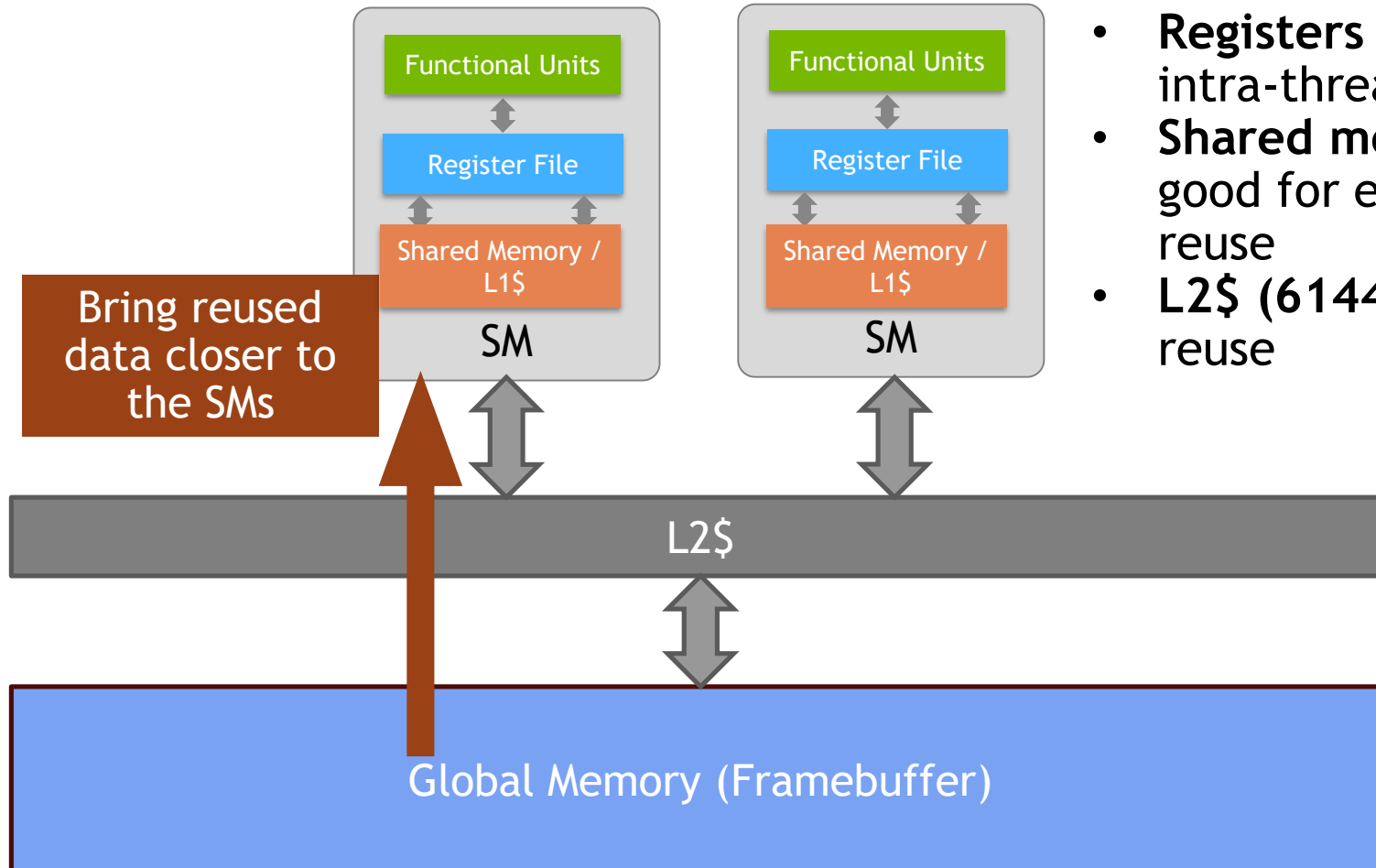


Device Memory

Reads	36453066	664.593 GB/s	
Writes	4573069	83.374 GB/s	
Total	41026135	747.966 GB/s	

GPU MEMORY HIERARCHY

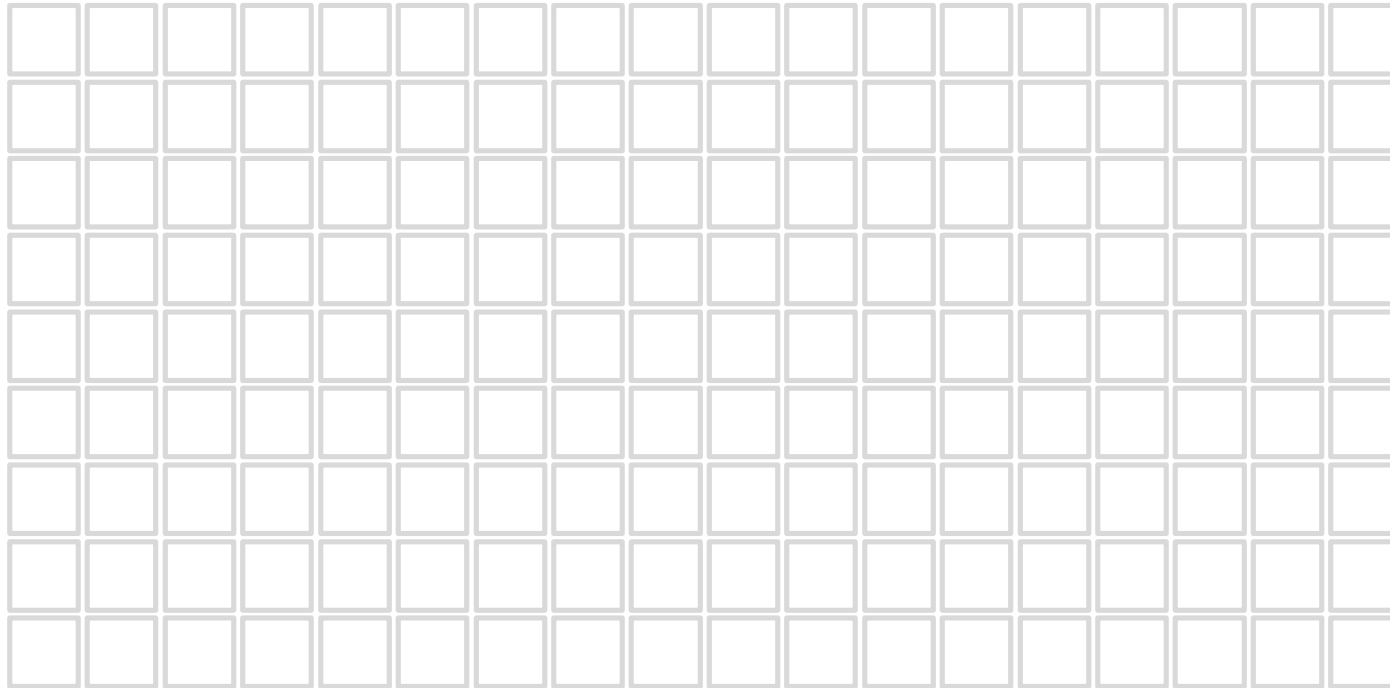
V100



- **Registers (256 KB/SM):** good for intra-thread data reuse
- **Shared mem / L1\$ (128 KB/SM):** good for explicit intra-block data reuse
- **L2\$ (6144 KB):** implicit data reuse

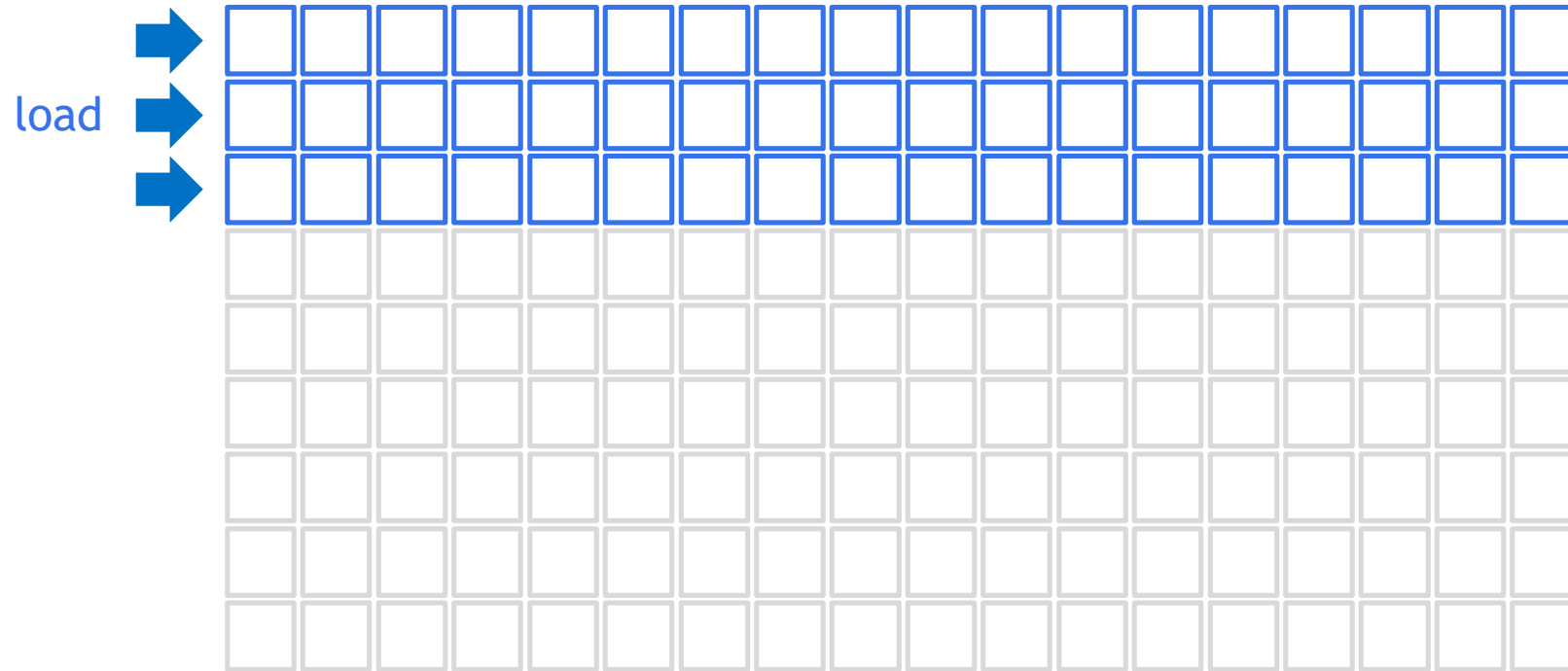
CACHING IN REGISTERS

No data loaded initially



CACHING IN REGISTERS

Load first set of data



CACHING IN REGISTERS

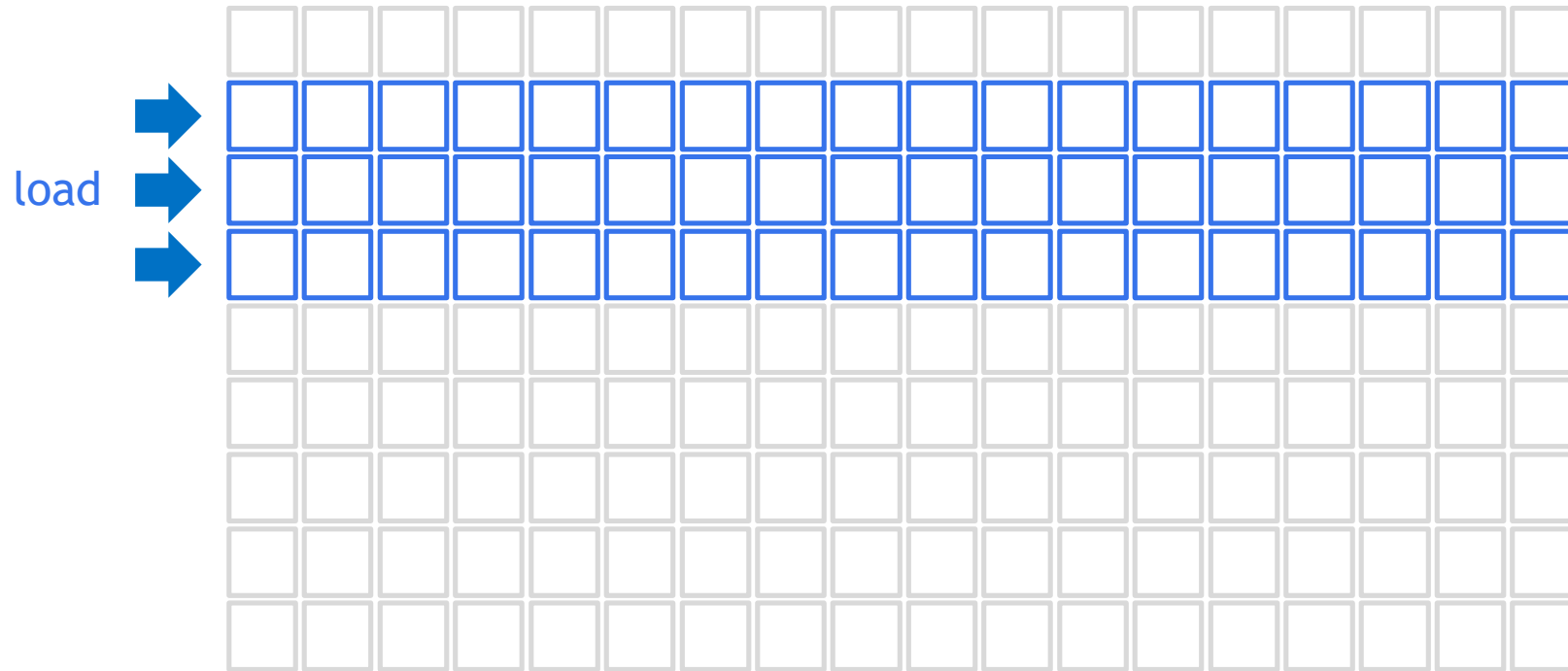
Perform calculation

Stencil



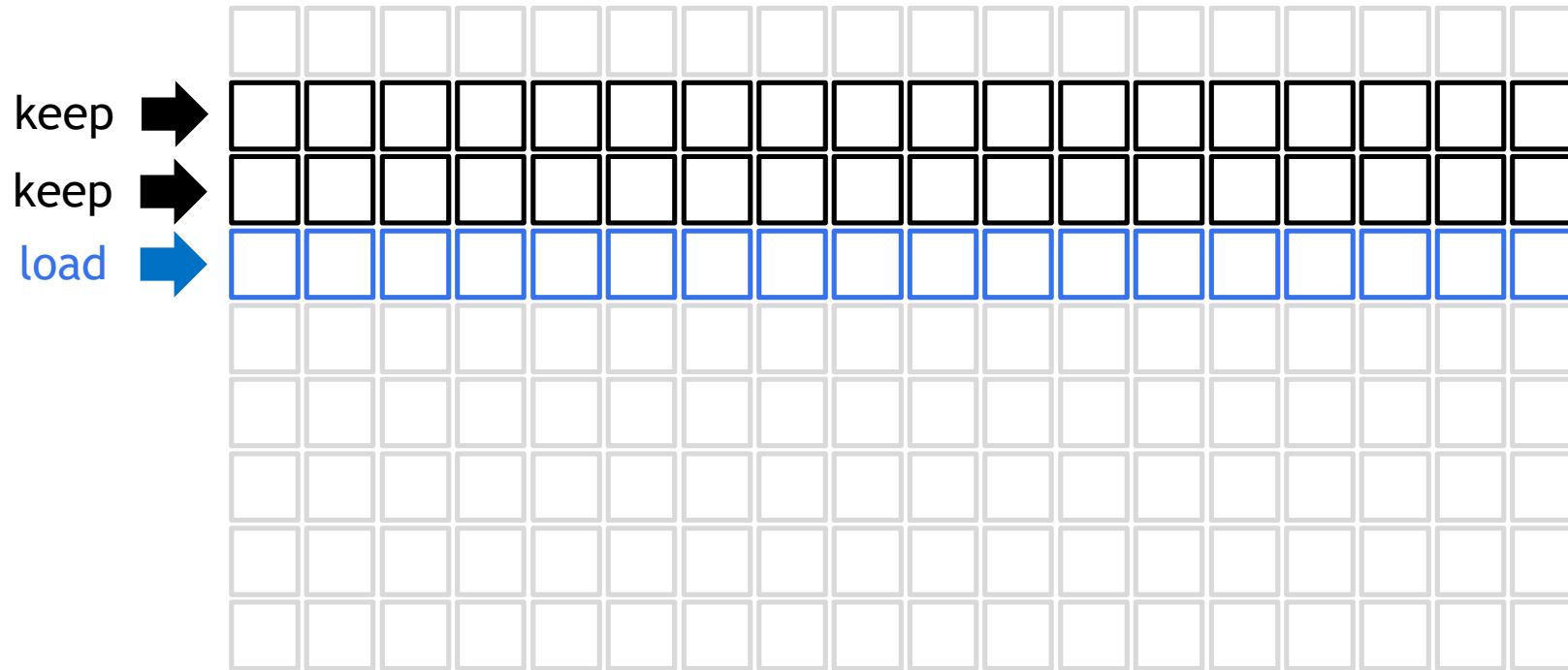
CACHING IN REGISTERS

Naively load next set of data?



CACHING IN REGISTERS

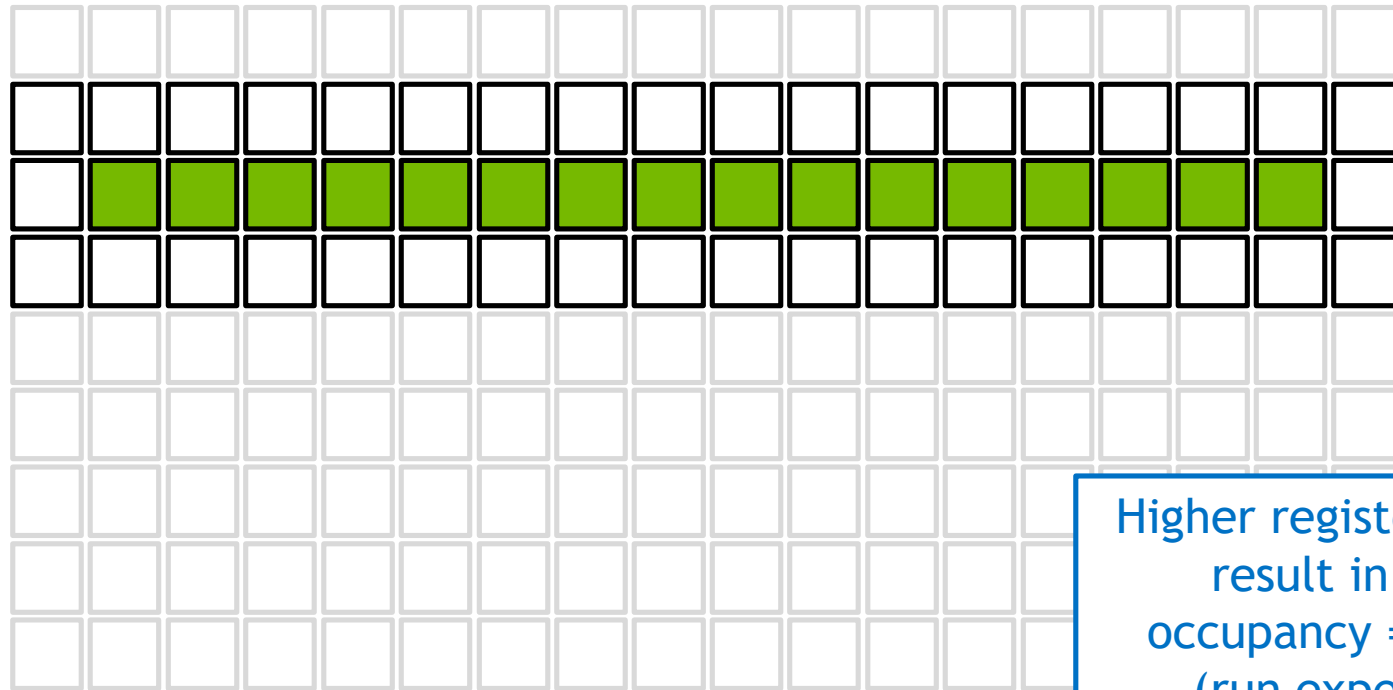
Reusing already loaded data is better



CACHING IN REGISTERS

Repeat

Stencil



Higher register usage may
result in reduced
occupancy => trade off
(run experiments!)

THE EFFECT OF REGISTER CACHING

```

Line / File smooth.reg.fv2.h - \home\cangerer\projects\GTC2017\hpgmg-cuda\finite-volume\source\cuda\st
85 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 589824 L2 transactions for
86 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 589824 L2 transactions for
88 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 589824 L2 transactions for
93 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4718592 L2 transactions for
94 Global Load L2 Transactions/Access = 9, Ideal Transactions/Access = 8 [ 4718592 L2 transactions for 524288 total executions ]
    
```

Transactions for cached loads reduced by a factor of 8

Memory utilization still high, but transferring less redundant data

Device Memory

Reads	29570665	636.46 GB/s	
Writes	4540978	97.737 GB/s	
Total	34111643	734.197 GB/s	

Kernel	Time	Speedup
Original Version	2.079ms	1.00x
Better Memory Accesses	1.756ms	1.18x
Register Caching	1.486ms	1.40x

SHARED MEMORY

- ▶ Programmer-managed cache
- ▶ Great for caching data reused across threads in a CTA
- ▶ 128KB split between shared memory and L1 cache per SM
 - ▶ Each block can use at most 96KB shared memory on GV100
 - ▶ Search for `cudaFuncAttributePreferredSharedMemoryCarveout` in the docs

```
__global__ void sharedMemExample(int *d) {  
    __shared__ float s[64];  
    int t = threadIdx.x;  
    s[t] = d[t];  
    __syncthreads();  
    if(t>0 && t<63)  
        stencil[t] = -2.0f*s[t] + s[t-1] + s[t+1];  
}
```



PERF-OPT QUICK REFERENCE CARD

Category:	Bandwidth Bound - Register Caching
Problem:	Data is reused within threads and memory bw utilization is high
Goal:	<u>Reduce</u> amount of data traffic to/from global mem
Indicators:	High device memory usage, latency exposed Data reuse within threads and small-ish working set Low arithmetic intensity of the kernel
Strategy:	<ul style="list-style-type: none">• Assign registers to cache data• Avoid storing and reloading data (possibly by assigning work to threads differently)• Avoid register spilling



PERF-OPT QUICK REFERENCE CARD

Category:	Latency Bound - Texture Cache
Problem:	Load/Store Unit becomes bottleneck
Goal:	Relieve Load/Store Unit from read-only data
Indicators:	High utilization of Load/Store Unit, pipe-busy stall reason, significant amount of read-only data
Strategy:	Load read-only data through Texture Units: <ul style="list-style-type: none">• Annotate read-only pointers with <code>const __restrict__</code>• Use <code>__ldg()</code> intrinsic



PERF-OPT QUICK REFERENCE CARD

Category:	Device Mem Bandwidth Bound - Shared Memory
Problem:	Too much data movement
Goal:	<u>Reduce</u> amount of data traffic to/from global mem
Indicators:	Higher than expected memory traffic to/from global memory Low arithmetic intensity of the kernel
Strategy:	(Cooperatively) move data closer to SM: <ul style="list-style-type: none">• Shared Memory• (or Registers)• (or Constant Memory)• (or Texture Cache)



PERF-OPT QUICK REFERENCE CARD

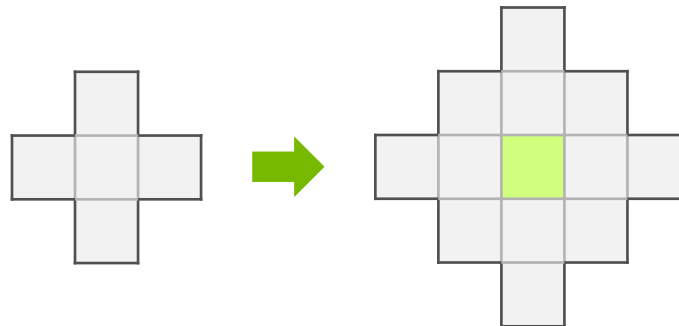
Category:	Shared Mem Bandwidth Bound - Shared Memory
Problem:	Shared memory bandwidth bottleneck
Goal:	<u>Reduce</u> amount of data traffic to/from global mem
Indicators:	Shared memory loads or stores saturate
Strategy:	Reduce Bank Conflicts (insert padding) Move data from shared memory into registers Change data layout in shared memory



**ITERATION 4:
KERNELS WITH INCREASED
ARITHMETIC INTENSITY**

OPERATIONAL INTENSITY

- Operational intensity = arithmetic operations/bytes written and read
- Our stencil kernels have very low operational intensity
- It might be beneficial to use a different algorithm with higher operational intensity.
- In this case this might be achieved by using higher order stencils




ILP VS OCCUPANCY

- Earlier we looked at how occupancy helps hide latency by providing independent threads of execution.
- When our code requires many registers the occupancy will be limited but we can still get instruction level parallelism inside the threads.
- Occupancy is helpful to achieving performance but not always required
- Some algorithms such as matrix multiplications allow increases in operational intensity by using more registers for local storage while simultaneously offering decent ILP. In these cases it might be beneficial to maximize ILP and operational intensity at the cost of occupancy.

Dependent instr.

```
a = b + c;  
d = a + f;
```




Independent instr.

```
a = b + c;  
d = e + f;
```

STALL REASONS: EXECUTION DEPENDENCY


```
a = b + c; // ADD
```

```
d = a + e; // ADD
```



```
a = b[i]; // LOAD
```

```
d = a + e; // ADD
```



Memory accesses may influence execution dependencies

Global accesses create longer dependencies than shared accesses

Read-only/texture dependencies are counted in Texture

Instruction level parallelism can reduce dependencies

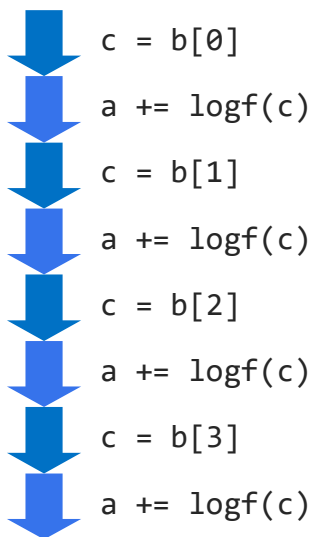
```
a = b + c; // Independent ADDs
```

```
d = e + f;
```

ILP AND MEMORY ACCESSSES

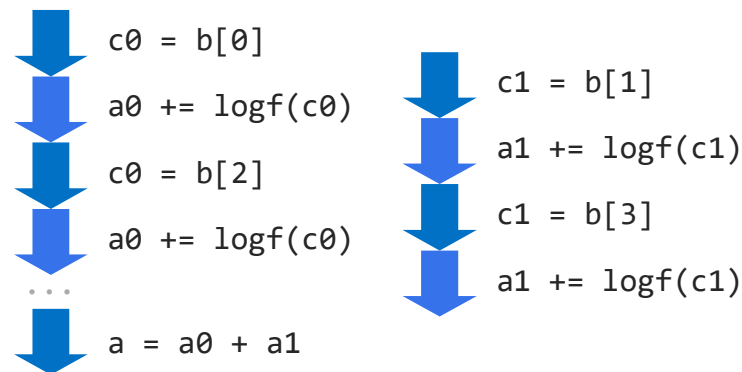
No ILP

```
float a = 0.0f;  
for( int i = 0 ; i < N ; ++i )  
    a += logf(b[i]);
```



2-way ILP (with loop unrolling)

```
float a, a0 = 0.0f, a1 = 0.0f;  
for( int i = 0 ; i < N ; i += 2 )  
{  
    a0 += logf(b[i]);  
    a1 += logf(b[i+1]);  
}  
a = a0 + a1
```



#pragma unroll is useful to extract ILP

Manually rewrite code if not a simple loop

PERF-OPT QUICK REFERENCE CARD

Category:	Latency Bound - Instruction Level Parallelism
Problem:	Not enough independent work per thread
Goal:	Do more parallel work inside single threads
Indicators:	High execution dependency, increasing occupancy has no/little positive effect, still registers available
Strategy:	<ul style="list-style-type: none">• Unroll loops (#pragma unroll)• Refactor threads to compute n output values at the same time (code duplication)



PERF-OPT QUICK REFERENCE CARD

Category:	Compute Bound - Algorithmic Changes
Problem:	GPU is computing as fast as possible
Goal:	Reduce computation if possible
Indicators:	Clearly compute bound problem, speedup only with less computation
Strategy:	<ul style="list-style-type: none">• Pre-compute or store (intermediate) results• Trade memory for compute time• Use a computationally less expensive algorithm• Possibly: run with low occupancy and high ILP

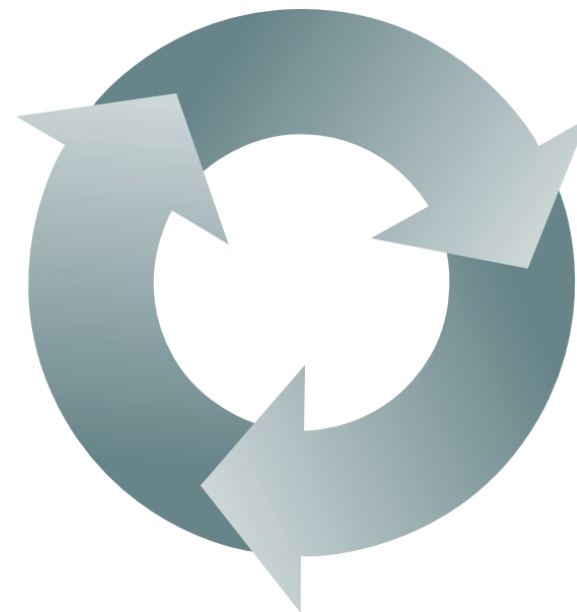


SUMMARY

SUMMARY

Performance Optimization is a Constant Learning Process

1. Know your application
2. Know your hardware
3. Know your tools
4. Know your process
 - Identify the Hotspot
 - Classify the Performance Limiter
 - Look for indicators
5. Make it so!



REFERENCES

CUDA Documentation

Best Practices: <http://docs.nvidia.com/cuda/cuda-c-best-practices-guide/>

Pascal Tuning Guide: <http://docs.nvidia.com/cuda/pascal-tuning-guide>

Volta Tuning Guide: <http://docs.nvidia.com/cuda/volta-tuning-guide/>

NVIDIA Developer Blog

<http://devblogs.nvidia.com/>

GTC 2018 Sessions:

<http://on-demand-gtc.gputechconf.com/gtcnew/on-demand-gtc.php>

THANK YOU

