WHAT THE PROFILER IS TELLING YOU: OPTIMIZING GPU KERNELS

Akshay Subramaniam
asubramaniam@nvidia.com

BEFORE YOU START

The five steps to enlightenment

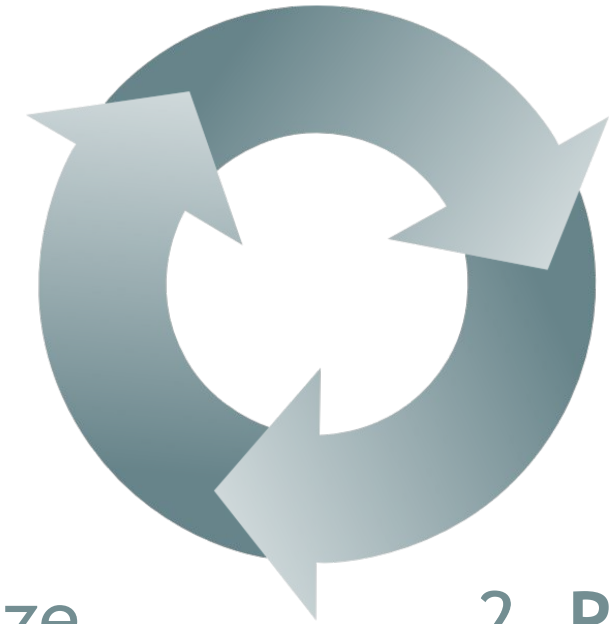
1. Know your hardware
   - What are the target machines, how many nodes? Machine-specific optimizations okay?
2. Know your tools
   - Strengths and weaknesses of each tool? Learn how to use them (and learn one well!)
3. Know your application
   - What does it compute? How is it parallelized? What final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

THE APOD CYCLE

4. Deploy and Test

1. Assess

- Identify Performance Limiter
- Analyze Profile
- Find Indicators



3. Optimize

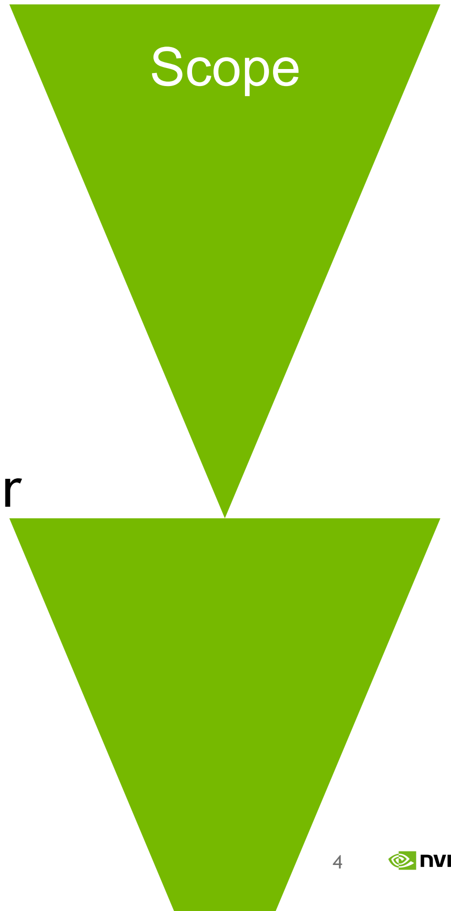
2. Parallelize

3b. Build Knowledge

GUIDING OPTIMIZATION EFFORT

“Drilling Down into the Metrics”

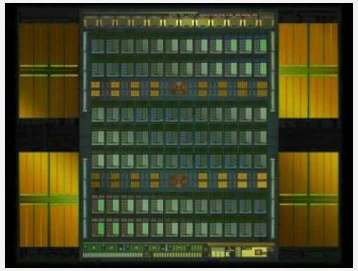
- Challenge: How to know where to start?
- Top-down Approach:
  - Find Hotspot Kernel
  - Identify Performance Limiter of the Hotspot
  - Find performance bottleneck indicators related to the limiter
  - Identify associated regions in the source code
  - Come up with strategy to fix and change the code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



KNOW YOUR HARDWARE: VOLTA ARCHITECTURE

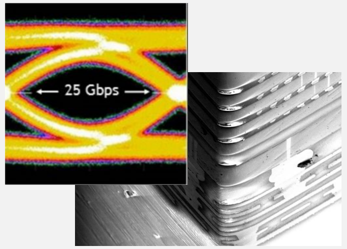
VOLTA V100 FEATURES

Volta Architecture



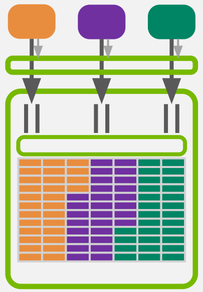
Most Productive GPU

Improved NVLink & HBM2



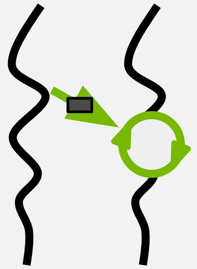
Efficient Bandwidth

Volta MPS



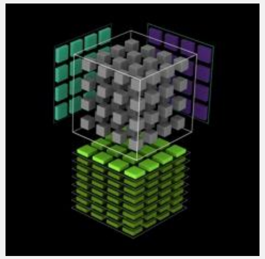
Inference Utilization

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

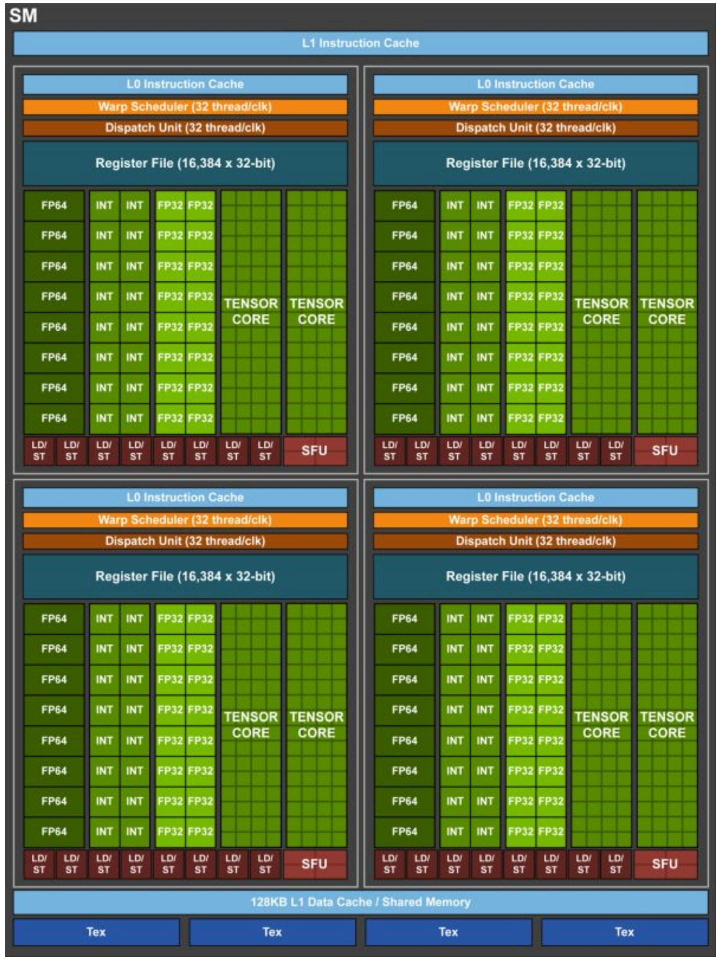
VOLTA SM

	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

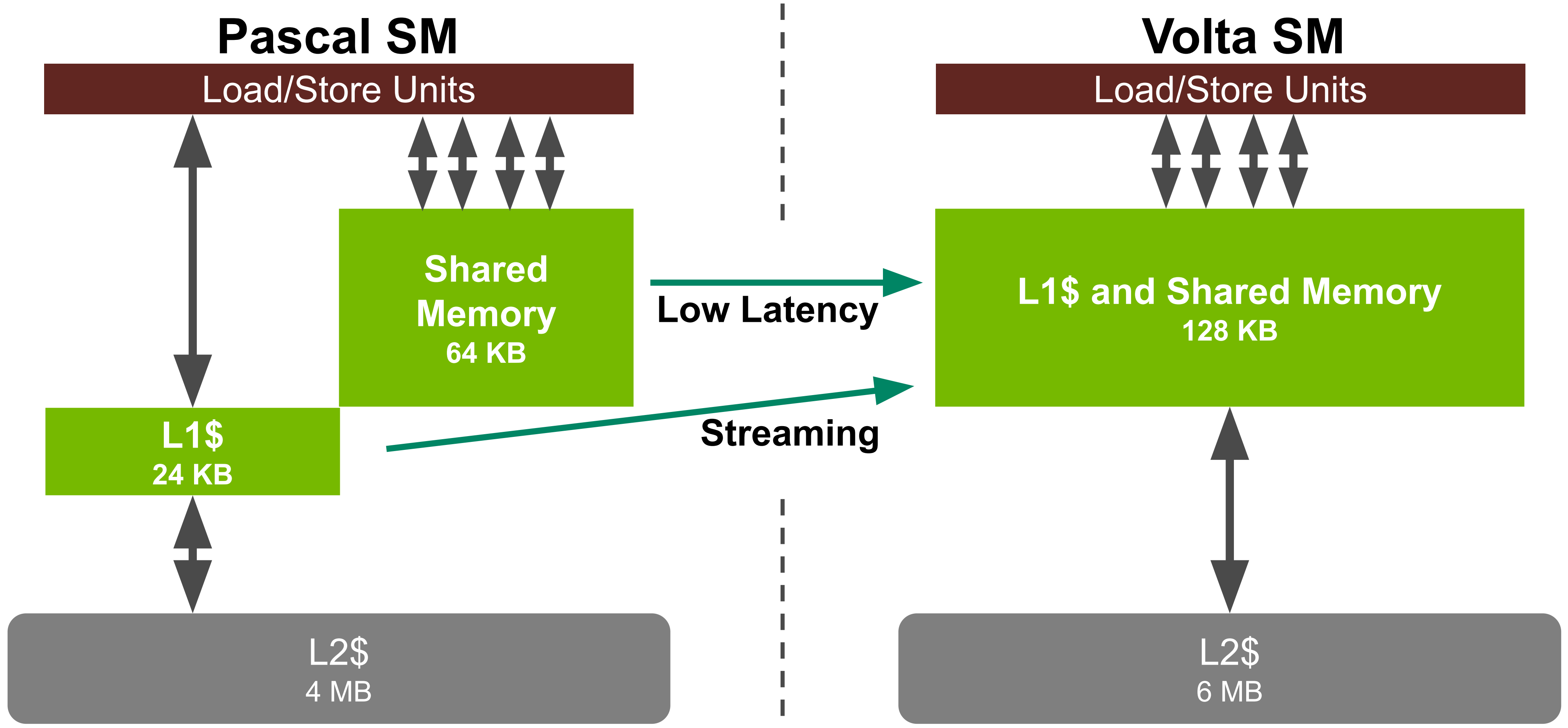
Same active threads/warps/blocks on SM

Same amount of registers

Expect similar occupancy, if not limited by shared mem.


SM
L1 Instruction Cache
L0 Instruction Cache
Warp Scheduler (32 thread/clk)
Dispatch Unit (32 thread/clk)
Register File (16,384 x 32-bit)
FP64
INT
FP32
TENSOR CORE
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex

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

$$\underset{\text{FP16 or FP32}}{D} = \underset{\text{FP16}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix}} + \underset{\text{FP16 or FP32}}{\begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}}$$

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

Volta, Turing, Ampere and future:

- NVIDIA Nsight Systems
- NVIDIA Nsight Compute

Older generations

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

Third Party

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

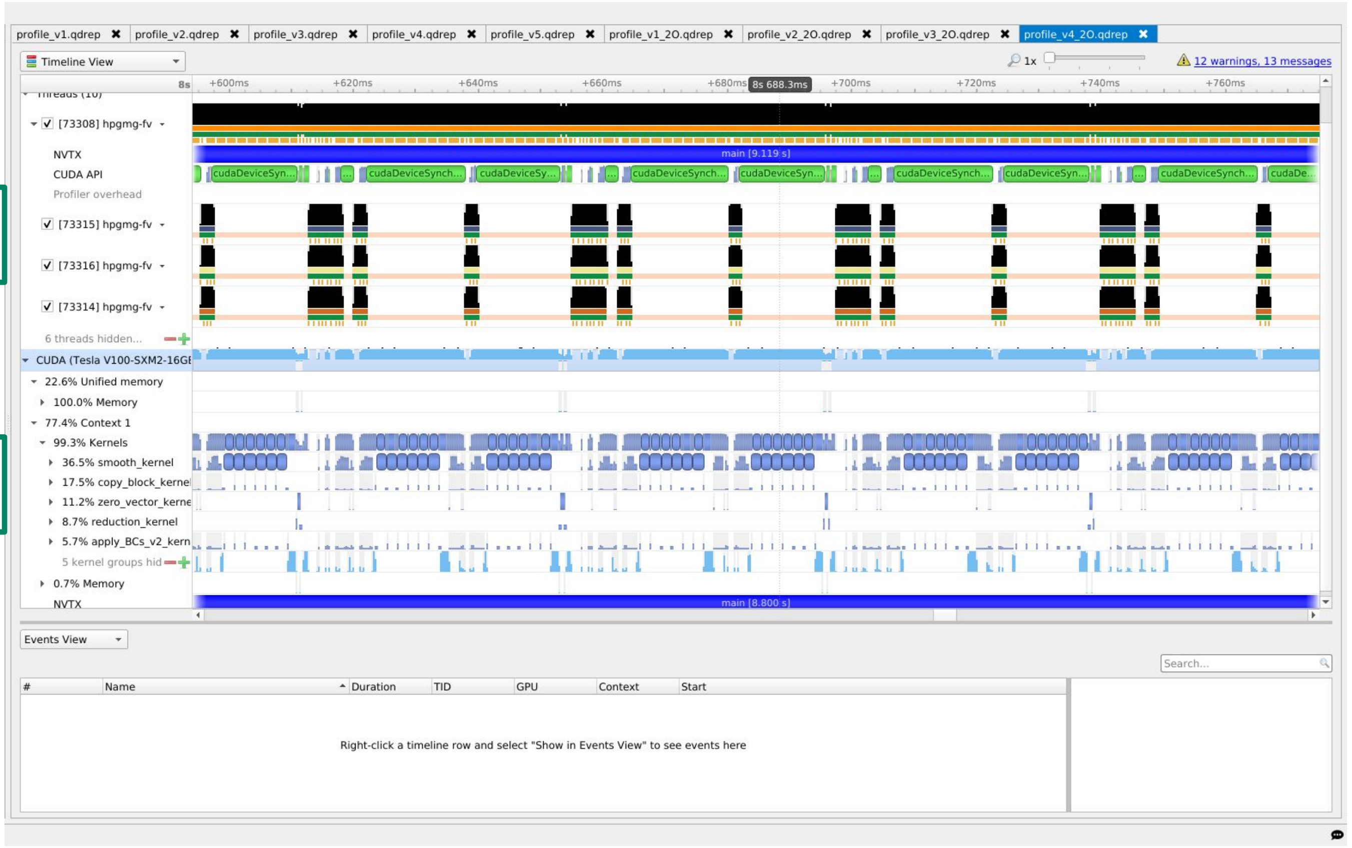
Without loss of generality, in this talk we will be showing Nsight systems and compute screenshots

Nsight Systems

System level analysis tool (think timeline)

CPU info

GPU info



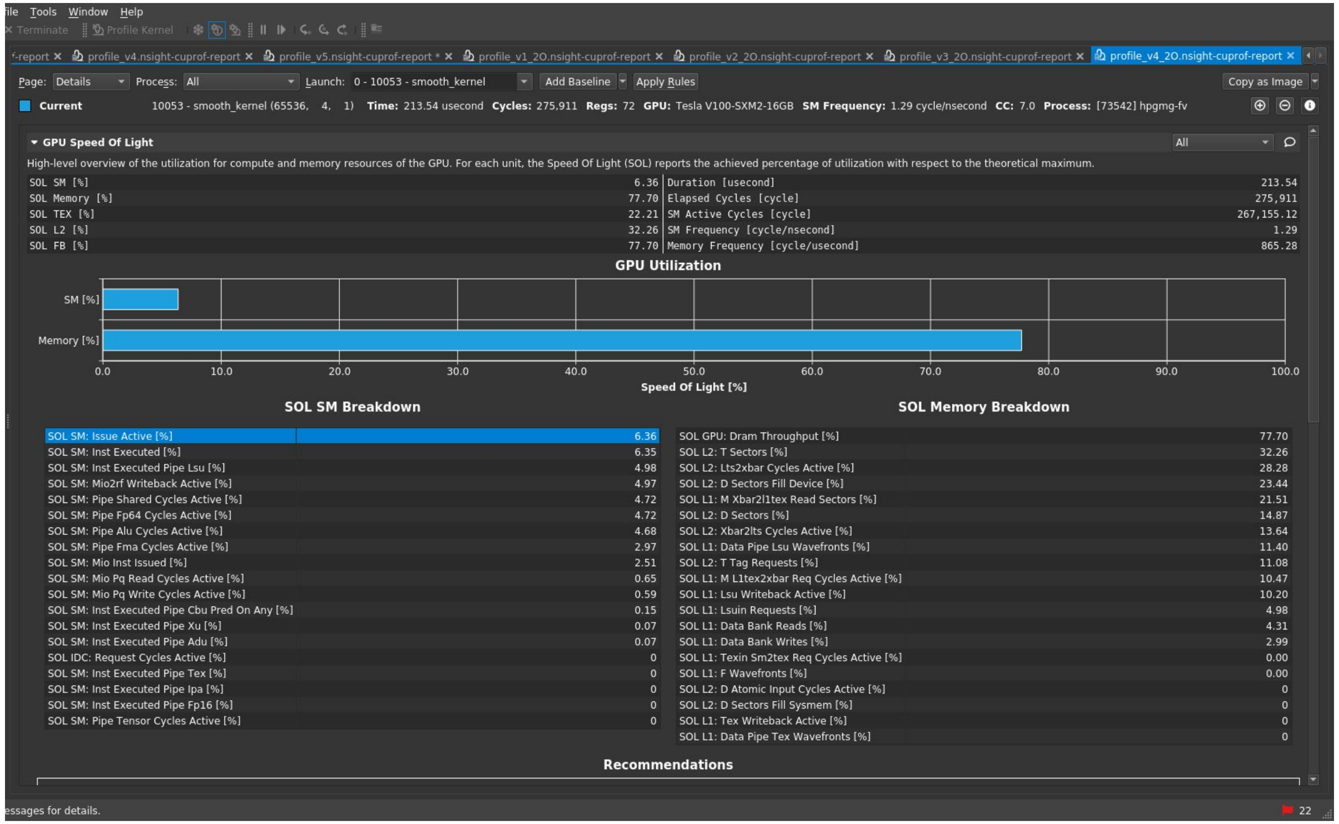
Right-click a timeline row and select "Show in Events View" to see events here

- **Nsight Systems:**

```
nsys profile -o profile_v4_20 ./build/bin/hpgmg-fv 7 8
```

Nsight Compute

Kernel analysis tool (think metrics)



Page: Details Process: All Launch: 0 - 10053 - smooth_kernel

Current 10053 - smooth_kernel (65536, 4, 1) **Time:** 213.54 usecond **Cycles:** 275,911 **Regs:** 72 **GPU:** Tesla V100-SXM2-16GB **SM Frequency:** 1.29 cycle/nsecond **CC:** 7.0 **Process:** [73542] hpgmg-fv

GPU Speed Of Light

High-level overview of the utilization for compute and memory resources of the GPU. For each unit, the Speed Of Light (SOL) reports the achieved percentage of utilization with respect to the theoretical maximum.

SOL SM [%]	6.36	Duration [usecond]	213.54
SOL Memory [%]	77.70	Elapsed Cycles [cycle]	275,911
SOL TEX [%]	22.21	SM Active Cycles [cycle]	267,155.12
SOL L2 [%]	32.26	SM Frequency [cycle/nsecond]	1.29
SOL FB [%]	77.70	Memory Frequency [cycle/usecond]	865.28

SOL SM Breakdown

SOL SM: Issue Active [%]	6.36
SOL SM: Inst Executed [%]	6.35
SOL SM: Inst Executed Pipe Lsu [%]	4.98
SOL SM: Mio2rf Writeback Active [%]	4.97
SOL SM: Pipe Shared Cycles Active [%]	4.72
SOL SM: Pipe Fp64 Cycles Active [%]	4.72
SOL SM: Pipe Alu Cycles Active [%]	4.68
SOL SM: Pipe Fma Cycles Active [%]	2.97
SOL SM: Mio Inst Issued [%]	2.51
SOL SM: Mio Pq Read Cycles Active [%]	0.65
SOL SM: Mio Pq Write Cycles Active [%]	0.59
SOL SM: Inst Executed Pipe Cbu Pred On Any [%]	0.15
SOL SM: Inst Executed Pipe Xu [%]	0.07
SOL SM: Inst Executed Pipe Adu [%]	0.07
SOL IDC: Request Cycles Active [%]	0
SOL SM: Inst Executed Pipe Tex [%]	0
SOL SM: Inst Executed Pipe Ipa [%]	0
SOL SM: Inst Executed Pipe Fp16 [%]	0
SOL SM: Pipe Tensor Cycles Active [%]	0

SOL Memory Breakdown

SOL GPU: Dram Throughput [%]	77.70
SOL L2: T Sectors [%]	32.26
SOL L2: Lts2xbar Cycles Active [%]	28.28
SOL L2: D Sectors Fill Device [%]	23.44
SOL L1: M Xbar2l1tex Read Sectors [%]	21.51
SOL L2: D Sectors [%]	14.87
SOL L2: Xbar2lts Cycles Active [%]	13.64
SOL L1: Data Pipe Lsu Wavefronts [%]	11.40
SOL L2: T Tag Requests [%]	11.08
SOL L1: M L1tex2xbar Req Cycles Active [%]	10.47
SOL L1: Lsu Writeback Active [%]	10.20
SOL L1: Lsuin Requests [%]	4.98
SOL L1: Data Bank Reads [%]	4.31
SOL L1: Data Bank Writes [%]	2.99
SOL L1: Texin Sm2tex Req Cycles Active [%]	0.00
SOL L1: F Wavefronts [%]	0.00
SOL L2: D Atomic Input Cycles Active [%]	0
SOL L2: D Sectors Fill Sysmem [%]	0
SOL L1: Tex Writeback Active [%]	0
SOL L1: Data Pipe Tex Wavefronts [%]	0

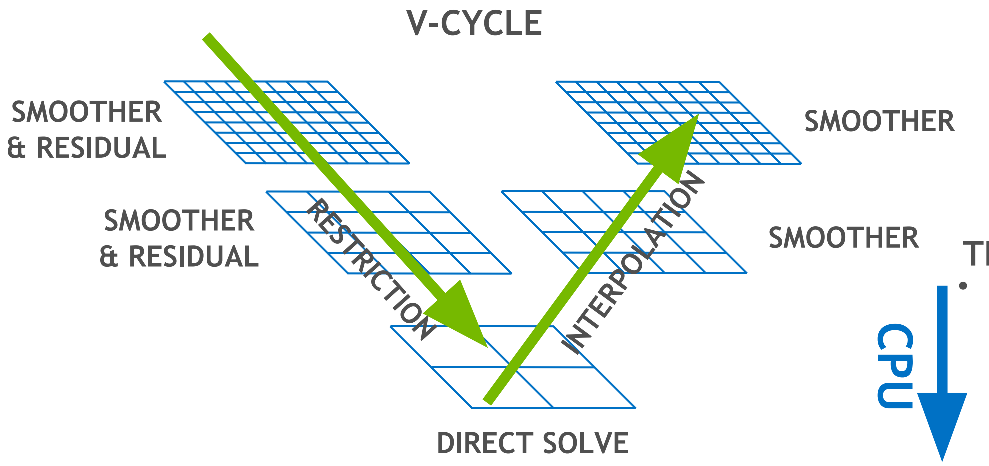
- **Nsight Compute:**

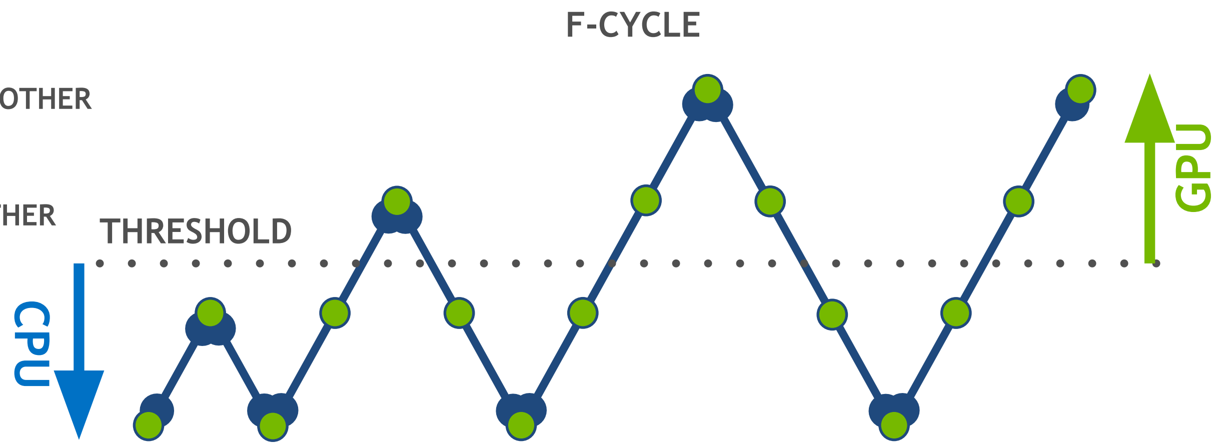
```
nv-nsight-cu-cli -o profile_v4_20 \
    --launch-count 1 ./build/bin/hpgmg-fv 7 8
```

KNOW YOUR APPLICATION: HPGMG

HPGMG

High-Performance Geometric Multi-Grid, Hybrid Implementation





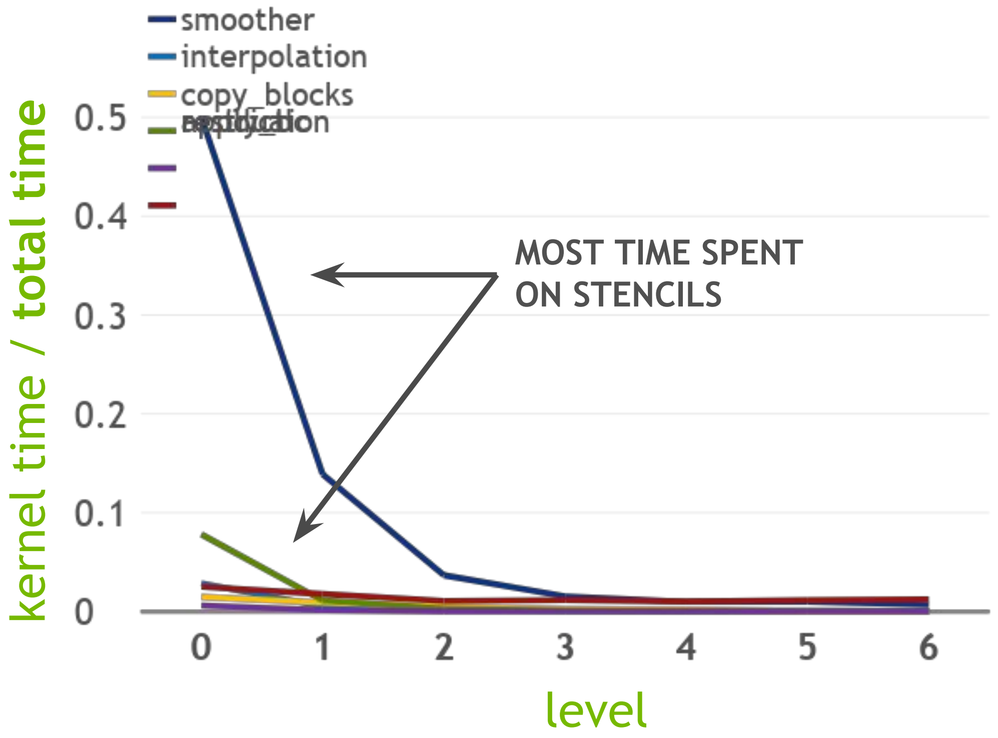
Fine levels are executed on throughput-optimized processors (GPU)

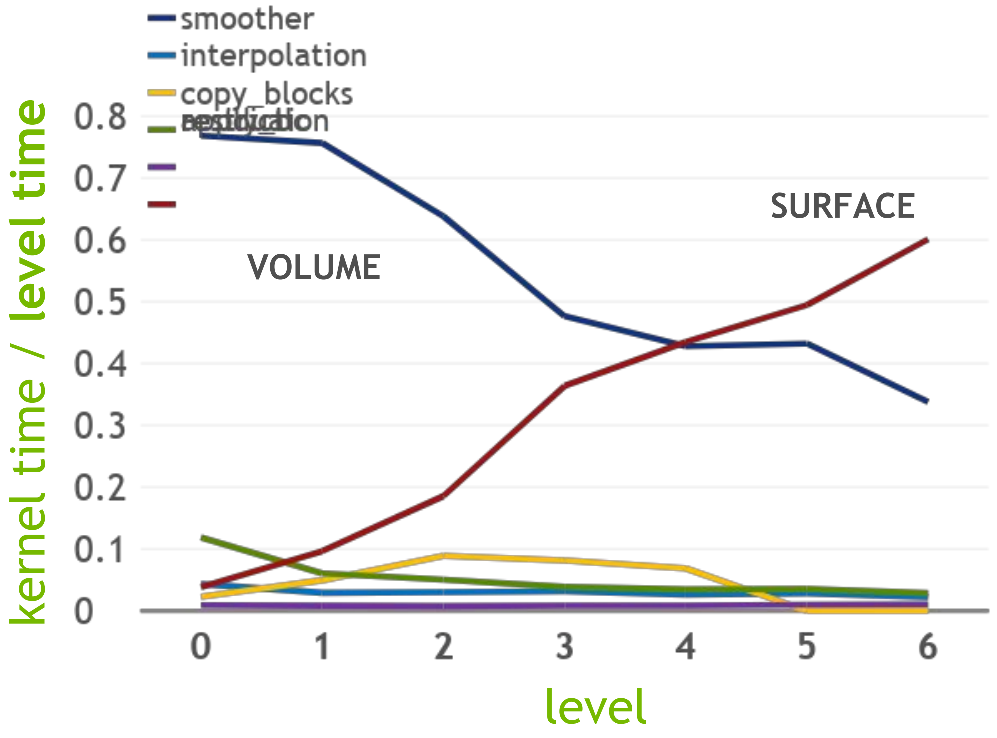
Coarse levels are executed on latency-optimized processors (CPU)

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

MULTI-GRID BOTTLENECK

Cost of operations



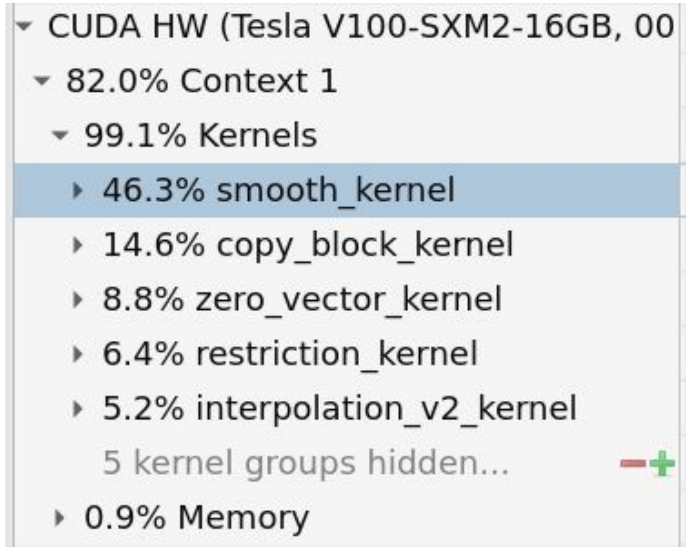


MAKE IT SO: ITERATION 1

2ND ORDER 7-POINT STENCIL

IDENTIFY HOTSPOT

Hotspot



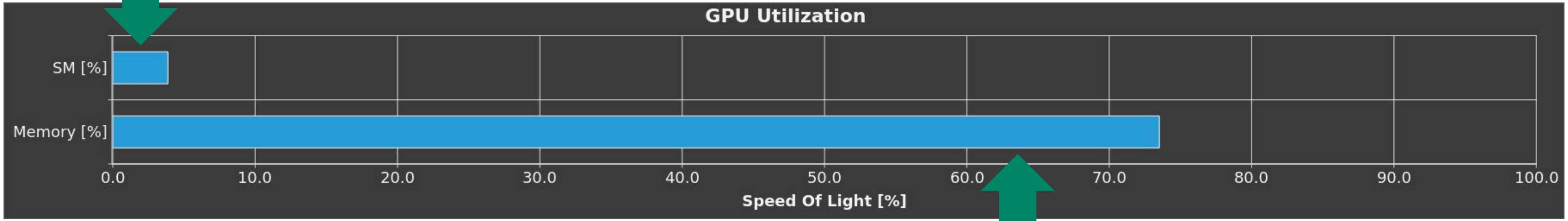
```
▾ CUDA HW (Tesla V100-SXM2-16GB, 00
  ▾ 82.0% Context 1
    ▾ 99.1% Kernels
      ▸ 46.3% smooth_kernel
      ▸ 14.6% copy_block_kernel
      ▸ 8.8% zero_vector_kernel
      ▸ 6.4% restriction_kernel
      ▸ 5.2% interpolation_v2_kernel
        5 kernel groups hidden...
    ▸ 0.9% Memory
```

Identify the hotspot: smooth_kernel()

Kernel	Time	Speedup
Original Version	2.079ms	1.00x

IDENTIFY PERFORMANCE LIMITER

Compute utilization

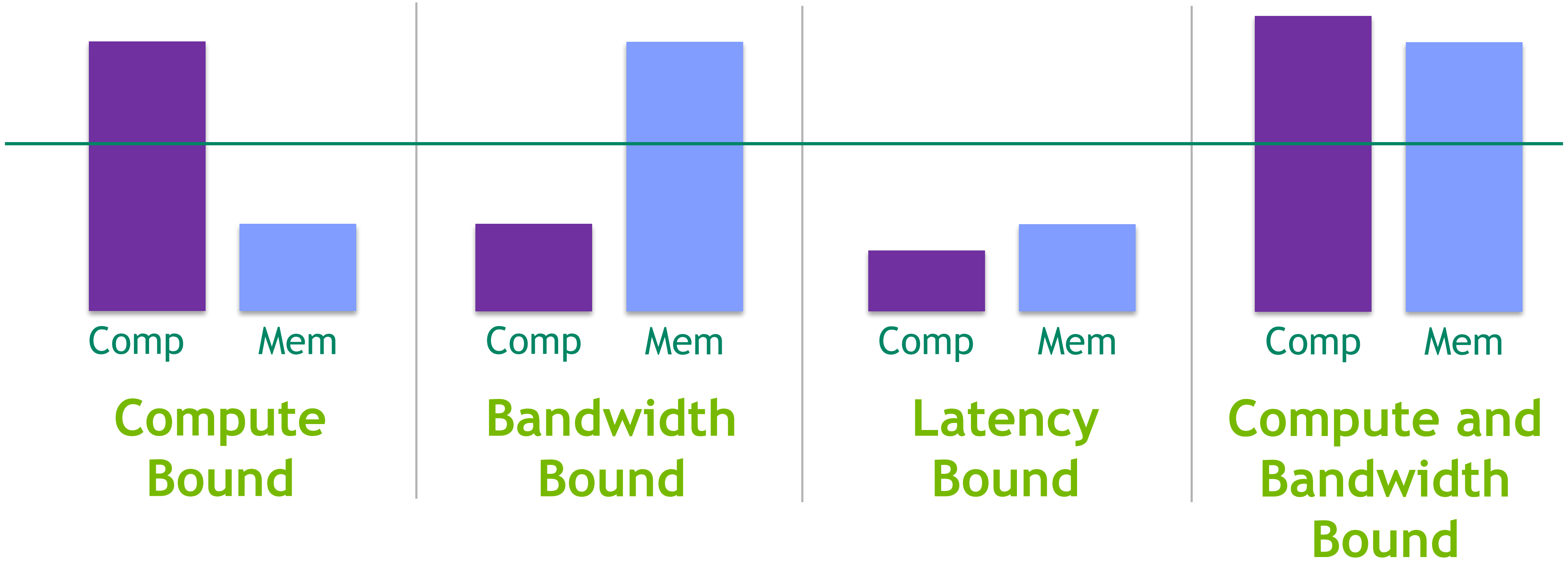


Memory utilization

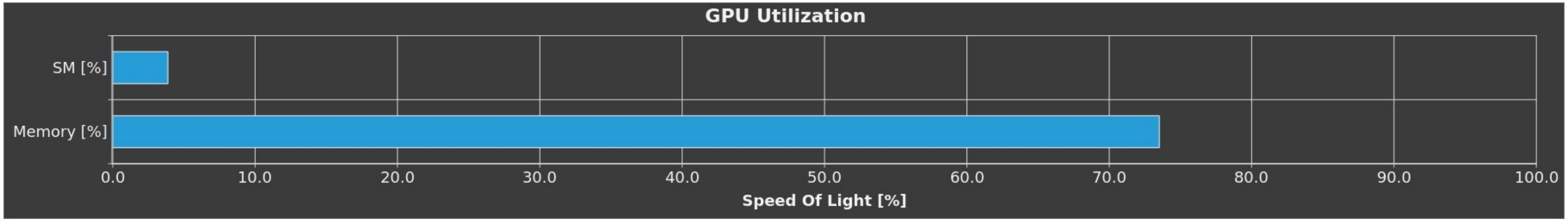
PERFORMANCE LIMITER CATEGORIES

Memory Utilization vs Compute Utilization

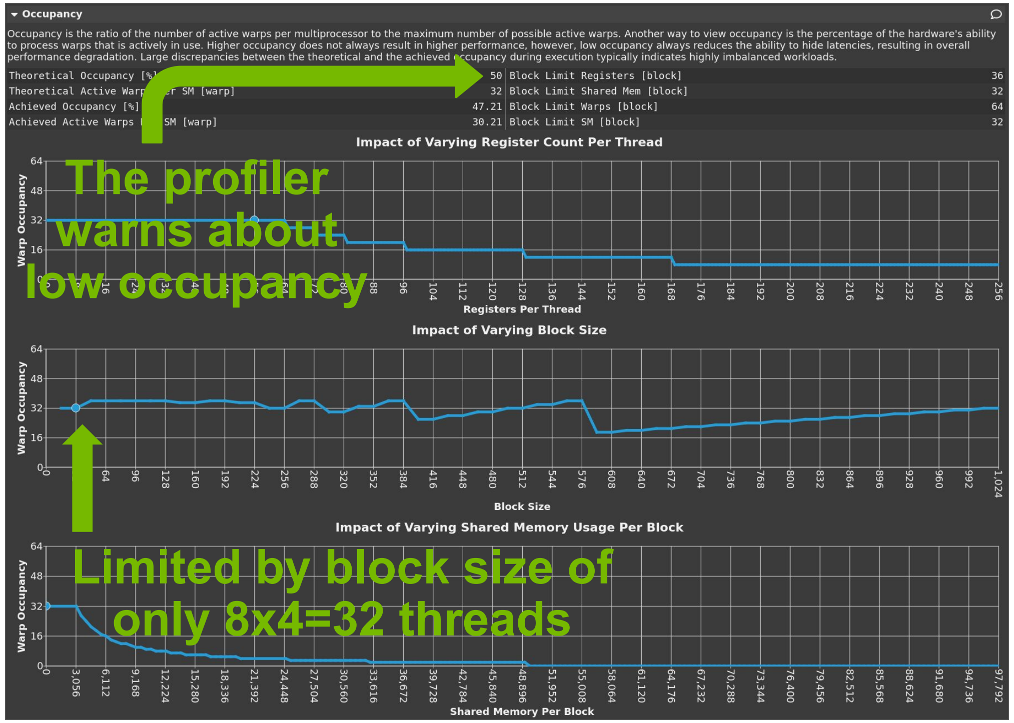
Four possible combinations:



BANDWIDTH BOUND ON V100



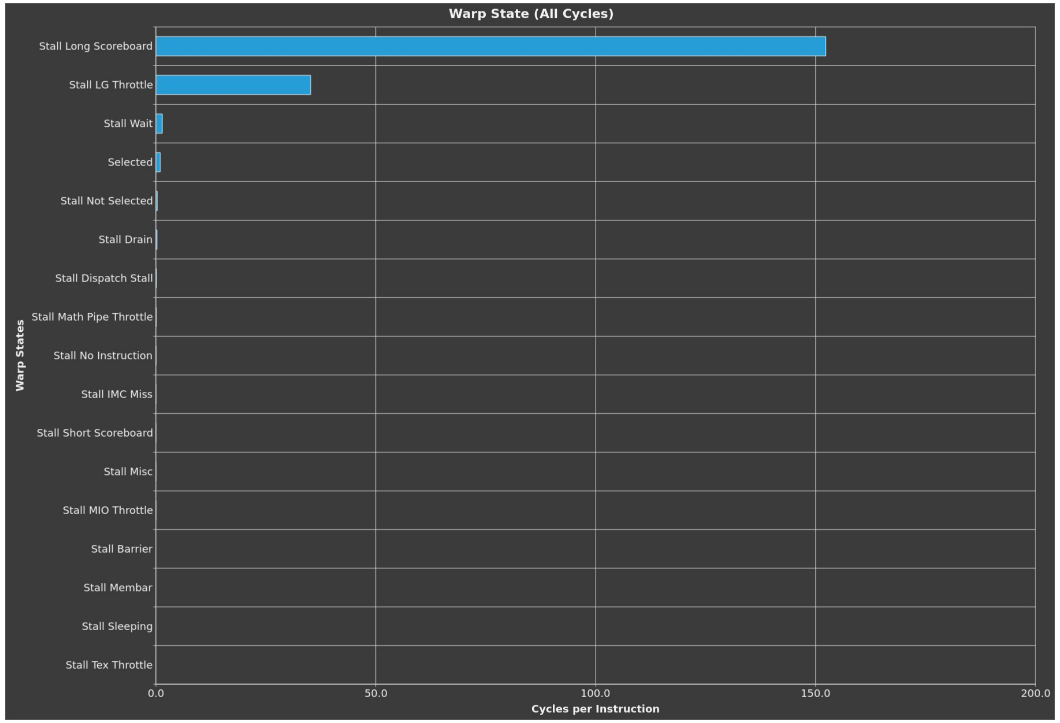
DRILLING DOWN: LATENCY ANALYSIS (V100)



Occupancy

Occupancy is the ratio of the number of active warps per multiprocessor to the maximum number of possible active warps. Another way to view occupancy is the percentage of the hardware's ability to process warps that is actively in use. Higher occupancy does not always result in higher performance, however, low occupancy always reduces the ability to hide latencies, resulting in overall performance degradation. Large discrepancies between the theoretical and the achieved occupancy during execution typically indicates highly imbalanced workloads.

Theoretical Occupancy [%]	50	Block Limit Registers [block]	36
Theoretical Active Warps per SM [warp]	32	Block Limit Shared Mem [block]	32
Achieved Occupancy [%]	47.21	Block Limit Warps [block]	64
Achieved Active Warps Per SM [warp]	30.21	Block Limit SM [block]	32



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

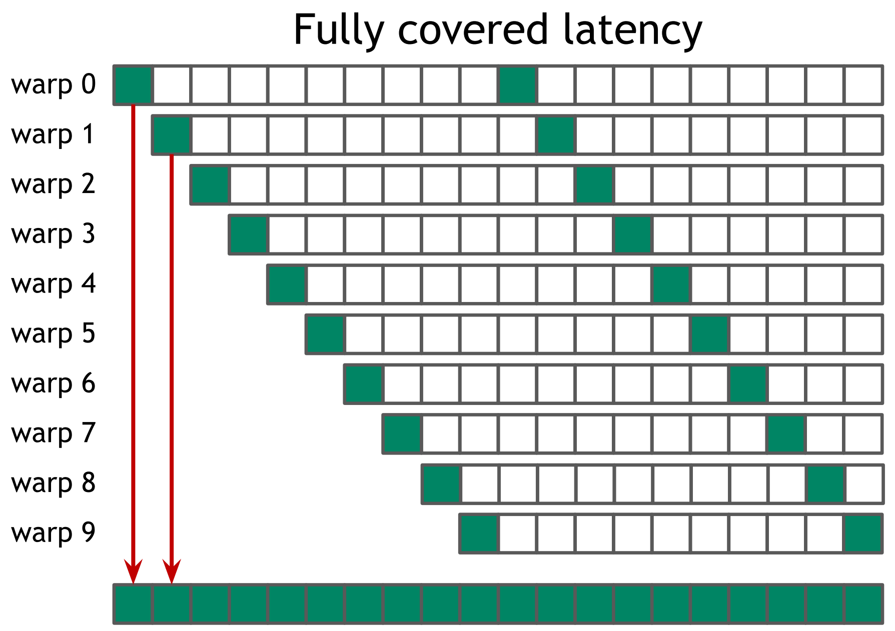
(*) Values vary with Compute Capability

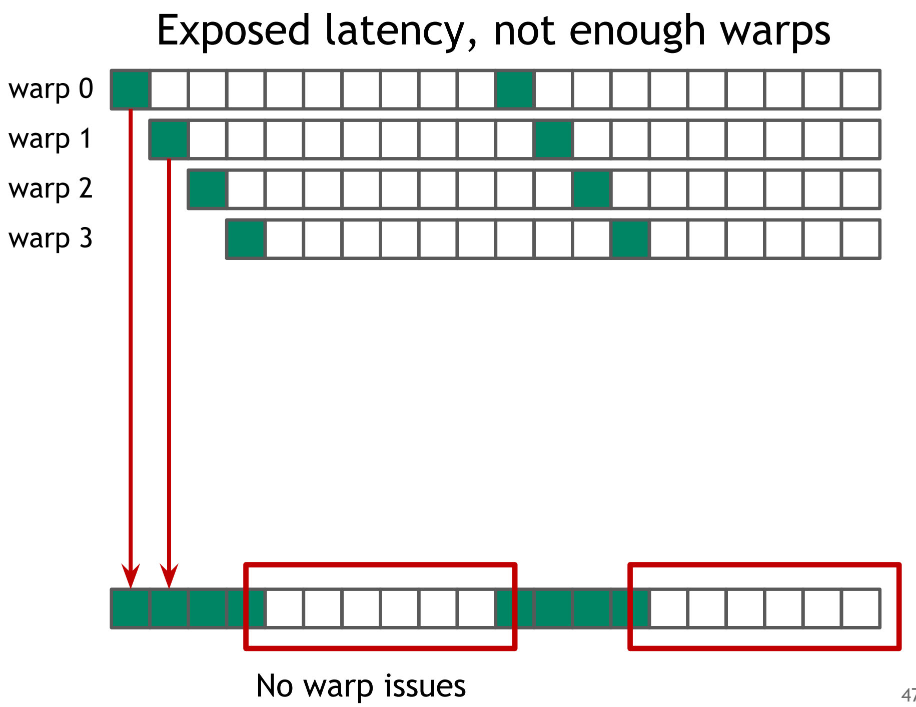
LATENCY

GPUs cover latencies by having a lot of work in flight

🟩 The warp issues

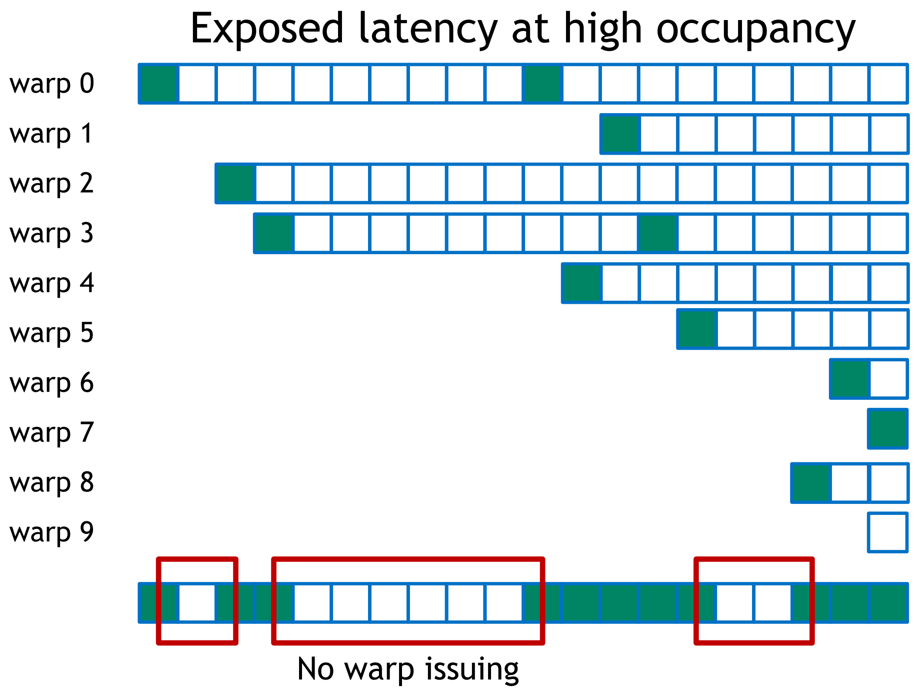
☐ The warp waits (latency)





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?

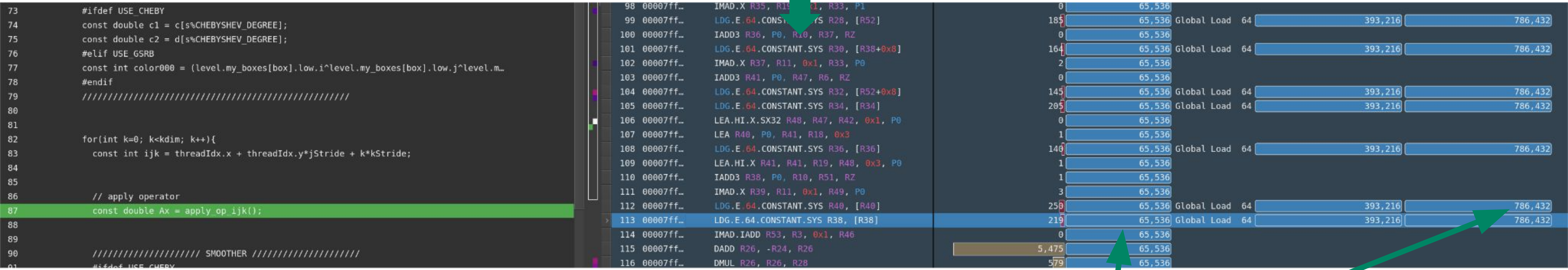
One V100:

BW = 4096 bit * 877Mhz * 2 / 8 = 898 GB/s ~ 1.23x of P100 (theoretical)

SM ratio: 80/56 = 1.43x of P100

LOOKING FOR MORE INDICATORS

Source Code Association



```
            #ifdef USE_CHEBY
            const double c1 = c[s%CHEBYSHEV_DEGREE];
            const double c2 = d[s%CHEBYSHEV_DEGREE];
            #elif USE_GSRB
            const int color000 = (level.my_boxes[box].low.i^level.my_boxes[box].low.j^level.m…
            #endif
            ////////////////////////////////////////////////////


            for(int k=0; k<kdim; k++){
              const int ijk = threadIdx.x + threadIdx.y*jStride + k*kStride;


              // apply operator
              const double Ax = apply_op_ijk();


              ///////////////////// SMOOTHER /////////////////////
              #ifdef USE_CHEBY
```

```
 98  00007ff…   IMAD.X R35, R19[illegible]1, R33, P1                        0  65,536
 99  00007ff…   LDG.E.64.CONS[illegible]YS R28, [R52]                     185  65,536  Global Load  64  393,216  786,432
100  00007ff…   IADD3 R36, P0, R10, R37, RZ                                 0  65,536
101  00007ff…   LDG.E.64.CONSTANT.SYS R30, [R38+0x8]                      164  65,536  Global Load  64  393,216  786,432
102  00007ff…   IMAD.X R37, R11, 0x1, R33, P0                               2  65,536
103  00007ff…   IADD3 R41, P0, R47, R6, RZ                                  0  65,536
104  00007ff…   LDG.E.64.CONSTANT.SYS R32, [R52+0x8]                      145  65,536  Global Load  64  393,216  786,432
105  00007ff…   LDG.E.64.CONSTANT.SYS R34, [R34]                          205  65,536  Global Load  64  393,216  786,432
106  00007ff…   LEA.HI.X.SX32 R48, R47, R42, 0x1, P0                        0  65,536
107  00007ff…   LEA R40, P0, R41, R18, 0x3                                  1  65,536
108  00007ff…   LDG.E.64.CONSTANT.SYS R36, [R36]                          140  65,536  Global Load  64  393,216  786,432
109  00007ff…   LEA.HI.X R41, R41, R19, R48, 0x3, P0                        1  65,536
110  00007ff…   IADD3 R38, P0, R10, R51, RZ                                 1  65,536
111  00007ff…   IMAD.X R39, R11, 0x1, R49, P0                               3  65,536
112  00007ff…   LDG.E.64.CONSTANT.SYS R40, [R40]                          250  65,536  Global Load  64  393,216  786,432
113  00007ff…   LDG.E.64.CONSTANT.SYS R38, [R38]                          219  65,536  Global Load  64  393,216  786,432
114  00007ff…   IMAD.IADD R53, R3, 0x1, R46                                 0  65,536
115  00007ff…   DADD R26, -R24, R26                                     5,475  65,536
116  00007ff…   DMUL R26, R26, R28                                        579  65,536
```

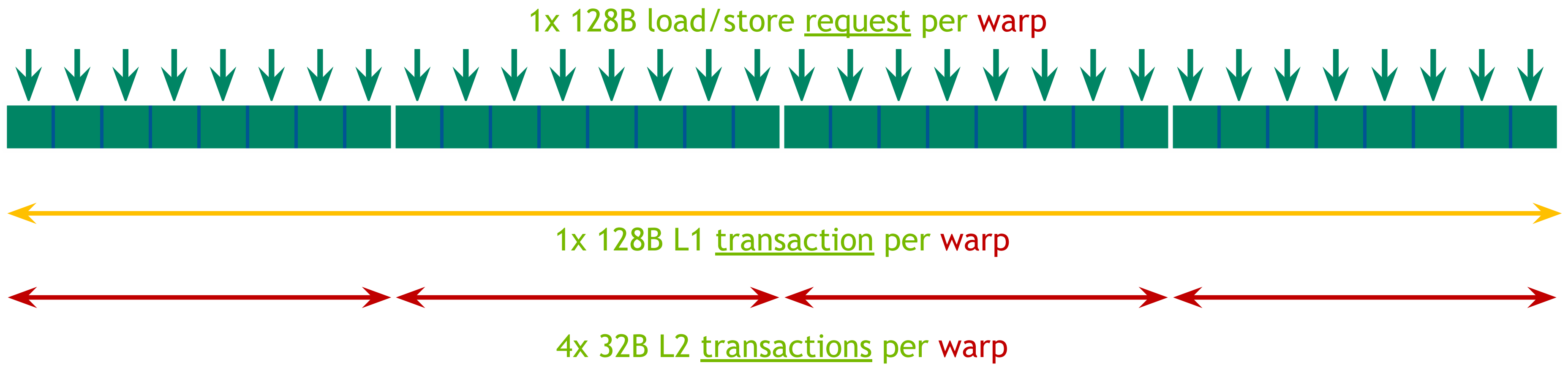
For line numbers use:
`nvcc -lineinfo`

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 128B load/store request per warp
1x 128B L1 transaction per warp
4x 32B L2 transactions per warp

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

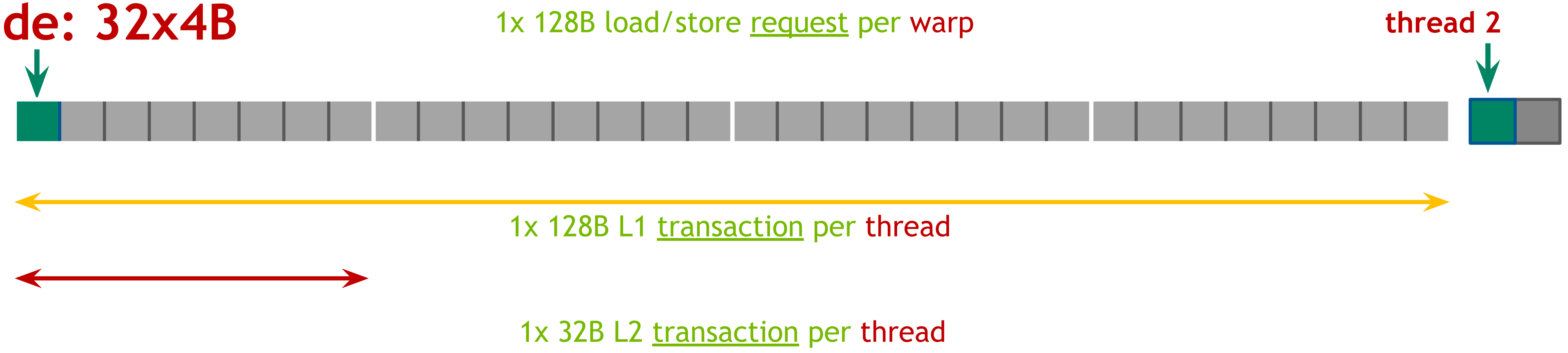
4x L2 transactions: 128B needed / 128B transferred

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



1x 128B load/store request per warp
thread 2
1x 128B L1 transaction per thread
1x 32B L2 transaction per thread

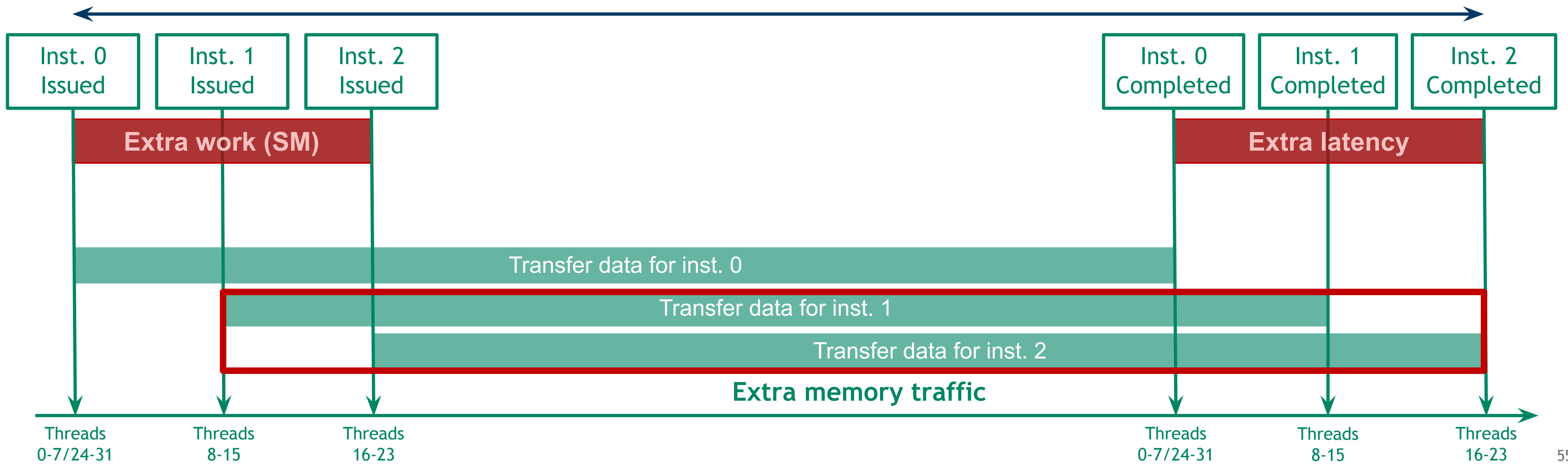
32x L1 transactions: 128B needed / 32x 128B transferred

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

TRANSACTIONS AND REPLAYS

With replays, requests take more time and use more resources

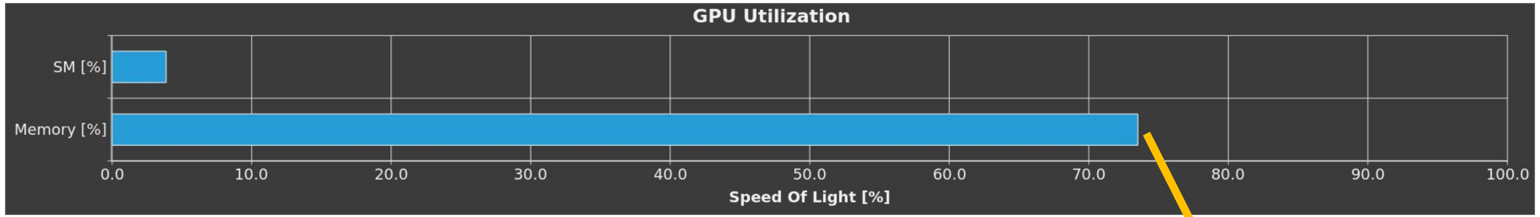
- More instructions issued
- More memory traffic
- Increased execution time



FIX: BETTER GPU TILING

Block Size Up from (8,4,1) to (32,4,1)

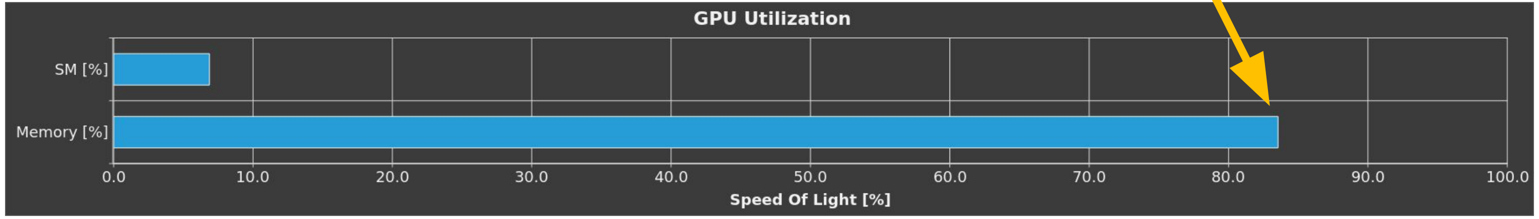
Before

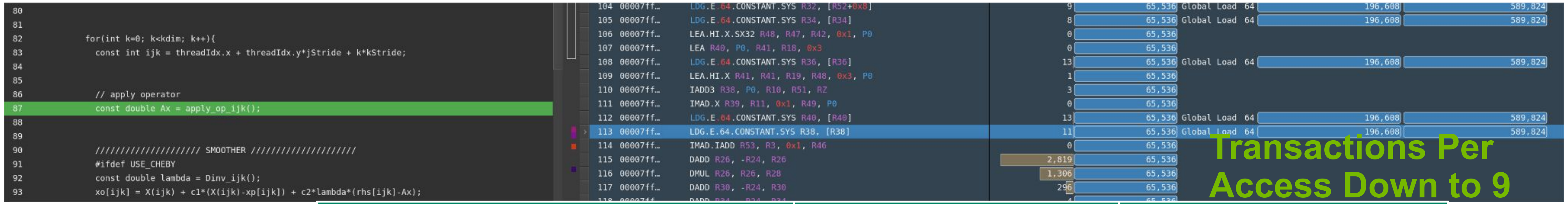


+10%

Memory Utilization Up

After





```
        for(int k=0; k<kdim; k++){
          const int ijk = threadIdx.x + threadIdx.y*jStride + k*kStride;


          // apply operator
          const double Ax = apply_op_ijk();


          ///////////////////// SMOOTHER /////////////////////
          #ifdef USE_CHEBY
          const double lambda = Dinv_ijk();
          xo[ijk] = X(ijk) + c1*(X(ijk)-xp[ijk]) + c2*lambda*(rhs[ijk]-Ax);
```

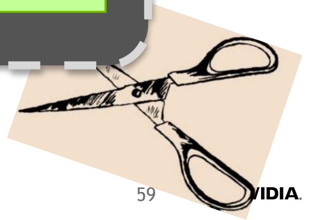
```
00007ff…   LDG.E.64.CONSTANT.SYS R32, [R52+0x8]        9   65,536  Global Load  64   196,608   589,824
00007ff…   LDG.E.64.CONSTANT.SYS R34, [R34]            8   65,536  Global Load  64   196,608   589,824
00007ff…   LEA.HI.X.SX32 R48, R47, R42, 0x1, P0        0   65,536
00007ff…   LEA R40, P0, R41, R18, 0x3                  0   65,536
00007ff…   LDG.E.64.CONSTANT.SYS R36, [R36]           13   65,536  Global Load  64   196,608   589,824
00007ff…   LEA.HI.X R41, R41, R19, R48, 0x3, P0        1   65,536
00007ff…   IADD3 R38, P0, R10, R51, RZ                 3   65,536
00007ff…   IMAD.X R39, R11, 0x1, R49, P0               0   65,536
00007ff…   LDG.E.64.CONSTANT.SYS R40, [R40]           13   65,536  Global Load  64   196,608   589,824
00007ff…   LDG.E.64.CONSTANT.SYS R38, [R38]           11   65,536  Global Load  64   196,608   589,824
00007ff…   IMAD.IADD R53, R3, 0x1, R46                 0   65,536
00007ff…   DADD R26, -R24, R26                     2,819   65,536
00007ff…   DMUL R26, R26, R28                      1,306   65,536
00007ff…   DADD R30, -R24, R30                       296   65,536
```

Transactions Per Access Down to 9

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

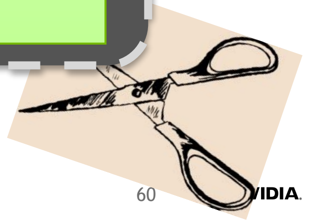
PERF-OPT QUICK REFERENCE CARD

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: • Varying block size • Varying shared memory usage • Varying register count (use __launch_bounds)



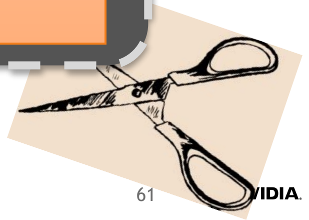
PERF-OPT QUICK REFERENCE CARD

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: • Cooperative loading inside a block • Change block layout • Aligning data • Changing data layout to improve locality



PERF-OPT QUICK REFERENCE CARD

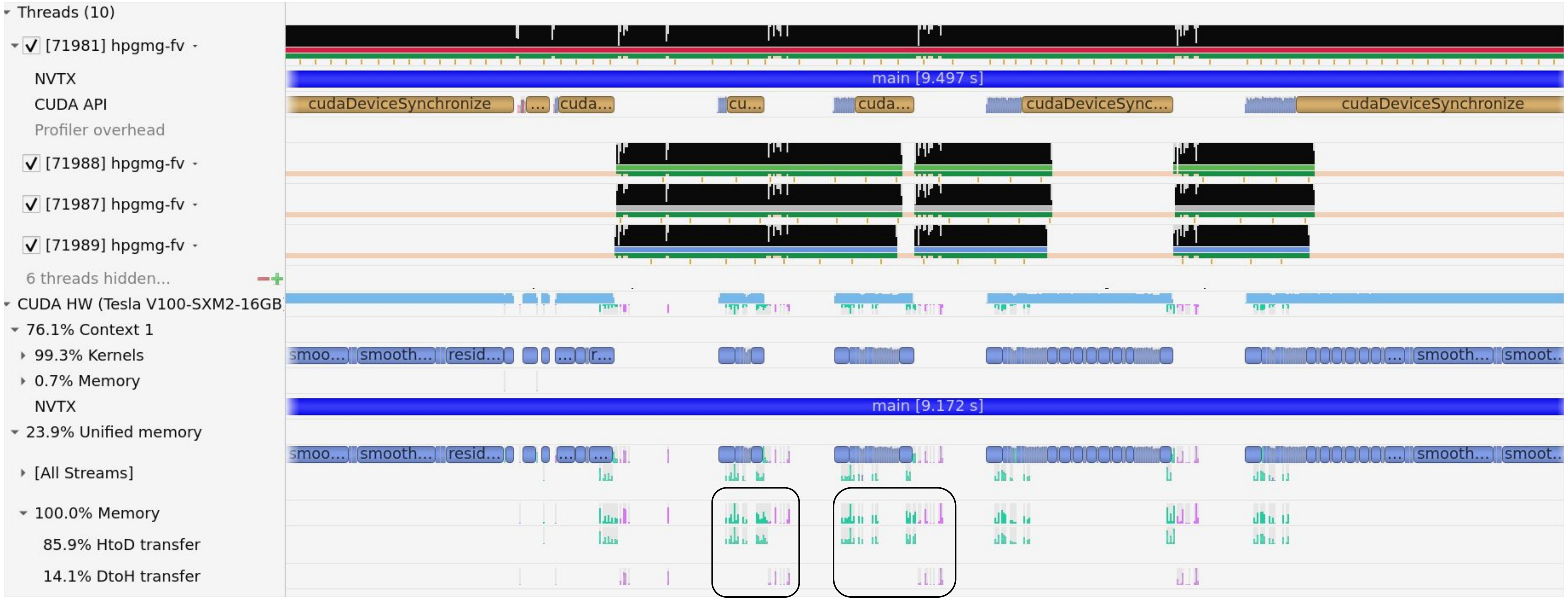
Category:	Bandwidth Bound - Coalescing
Problem:	Too much unused data clogging memory system
Goal:	Reduce traffic, move more <u>useful</u> data per request
Indicators:	Low global load/store efficiency, High #transactions/#request compared to ideal
Strategy:	Improve memory coalescing by: • Cooperative 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

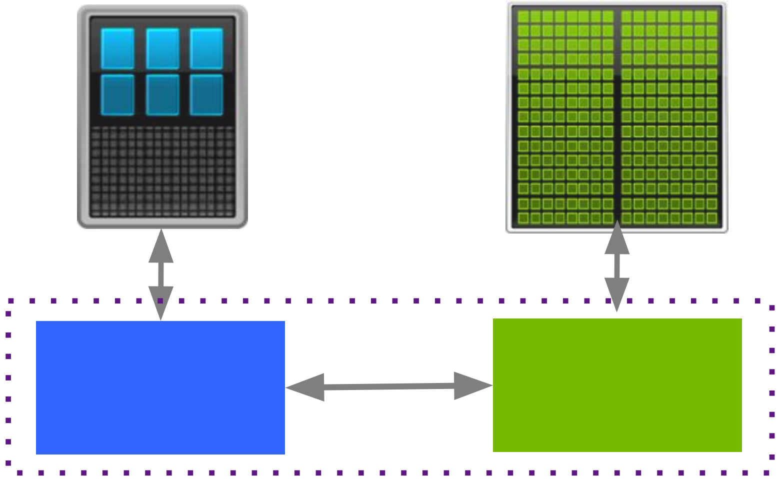
No changes to data structures

No explicit data movements

Single pointer for CPU and GPU data

Use **`cudaMallocManaged`** for allocations

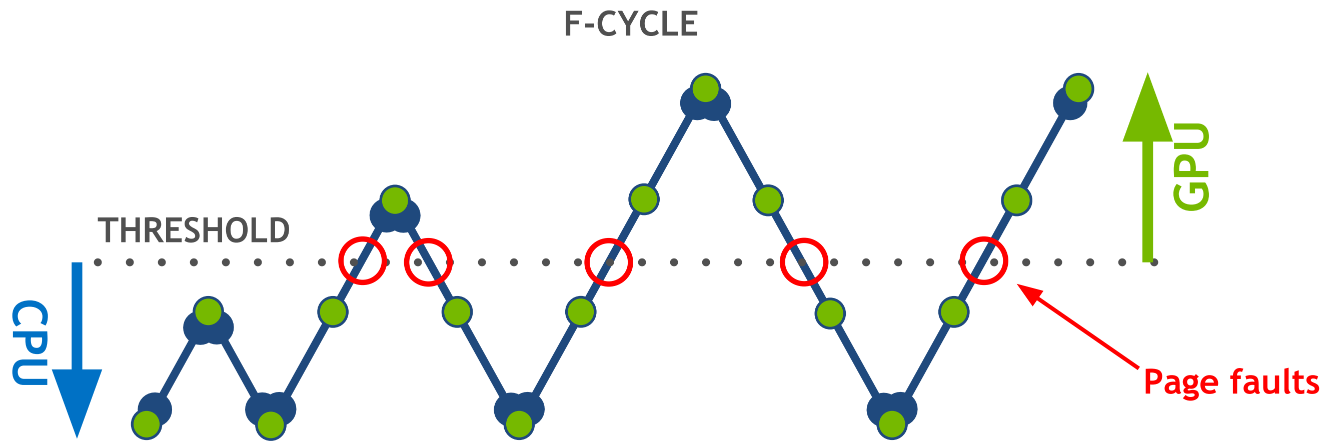
Developer View With Unified Memory



Unified Memory

UNIFIED MEMORY

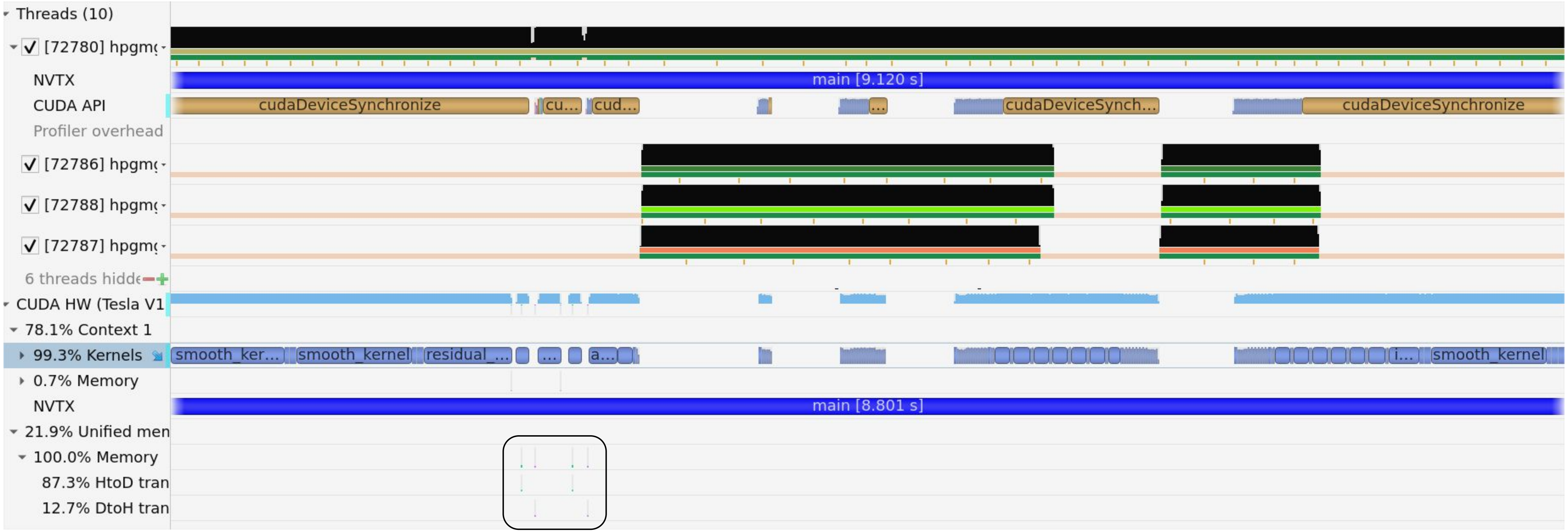
Eliminating page migrations and faults



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

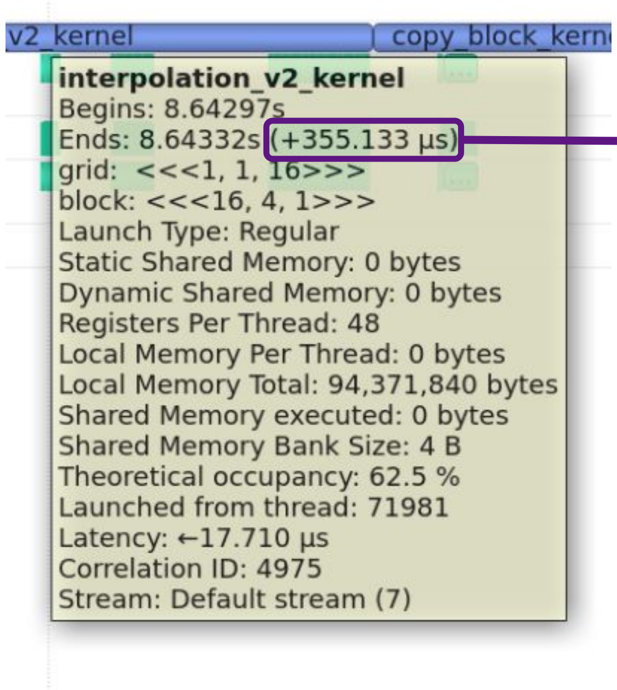
PAGE FAULTS

Almost gone

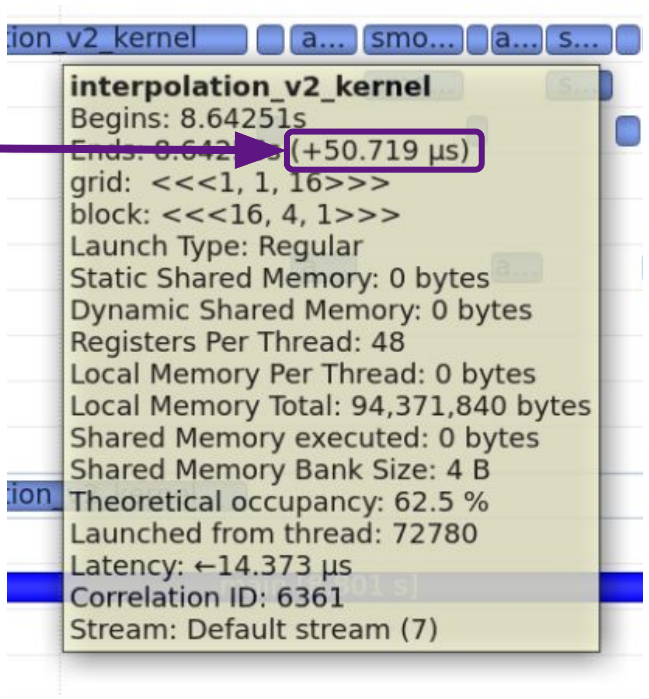


PAGE FAULTS

Significant speedup for affected kernel



interpolation_v2_kernel
Begins: 8.64297s
Ends: 8.64332s (+355.133 µs)
grid: <<<1, 1, 16>>>
block: <<<16, 4, 1>>>
Launch Type: Regular
Static Shared Memory: 0 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 48
Local Memory Per Thread: 0 bytes
Local Memory Total: 94,371,840 bytes
Shared Memory executed: 0 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 62.5 %
Launched from thread: 71981
Latency: ←17.710 µs
Correlation ID: 4975
Stream: Default stream (7)



interpolation_v2_kernel
Begins: 8.64251s
Ends: 8.642[illegible]s (+50.719 µs)
grid: <<<1, 1, 16>>>
block: <<<16, 4, 1>>>
Launch Type: Regular
Static Shared Memory: 0 bytes
Dynamic Shared Memory: 0 bytes
Registers Per Thread: 48
Local Memory Per Thread: 0 bytes
Local Memory Total: 94,371,840 bytes
Shared Memory executed: 0 bytes
Shared Memory Bank Size: 4 B
Theoretical occupancy: 62.5 %
Launched from thread: 72780
Latency: ←14.373 µs
Correlation ID: 6361
Stream: Default stream (7)

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
```

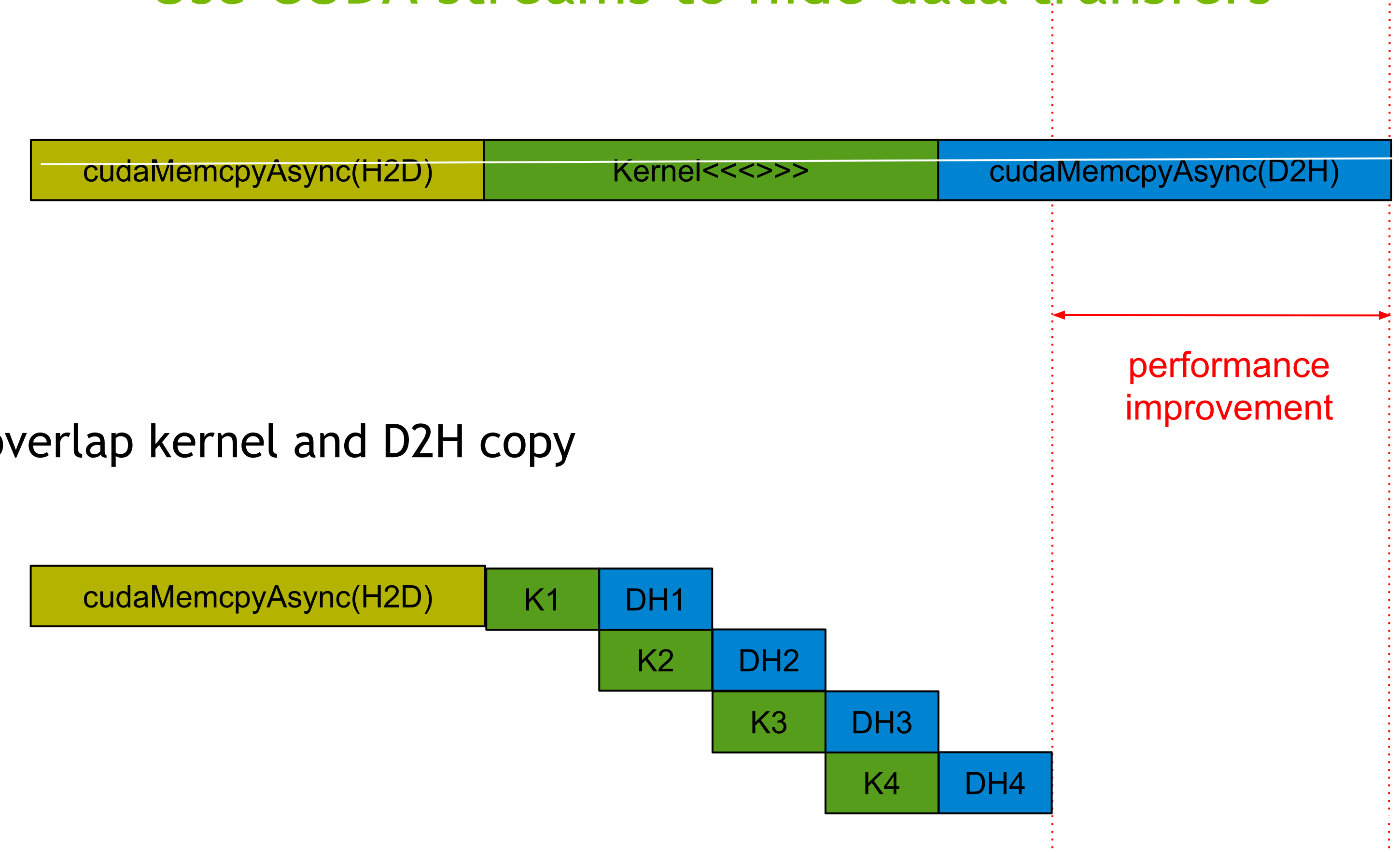
User can set and unset at any time

CONCURRENCY THROUGH PIPELINING

Use CUDA streams to hide data transfers

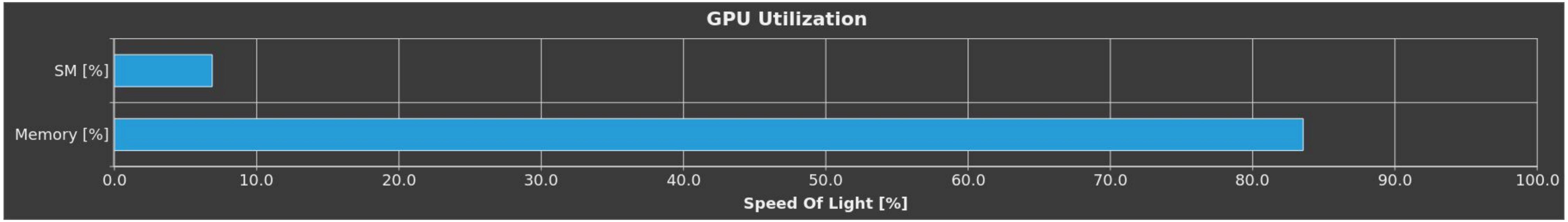
Serial

Concurrent– overlap kernel and D2H copy



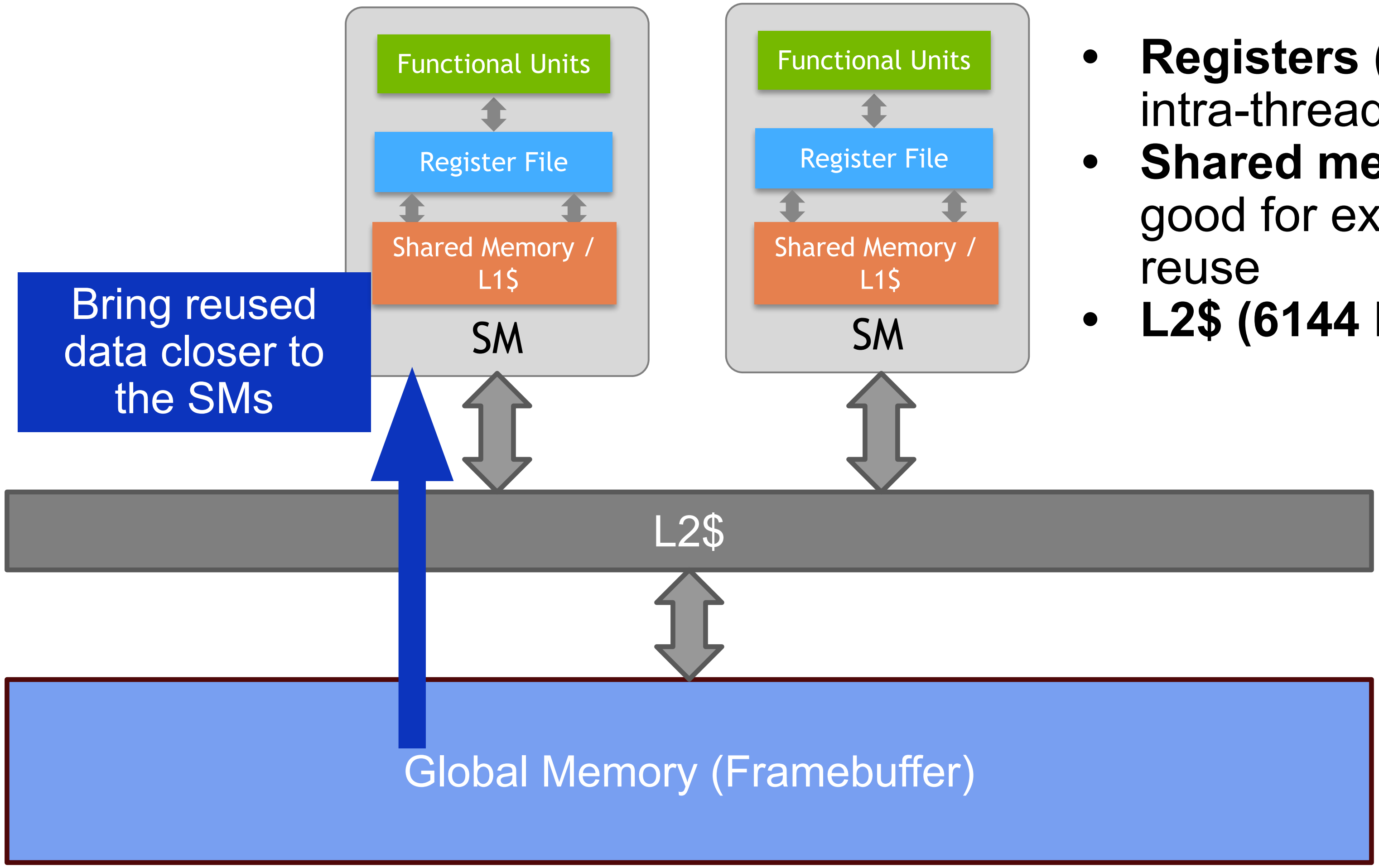
ITERATION 3:
REGISTER OPTIMIZATION AND CACHING

LIMITER: STILL MEMORY BANDWIDTH



GPU MEMORY HIERARCHY

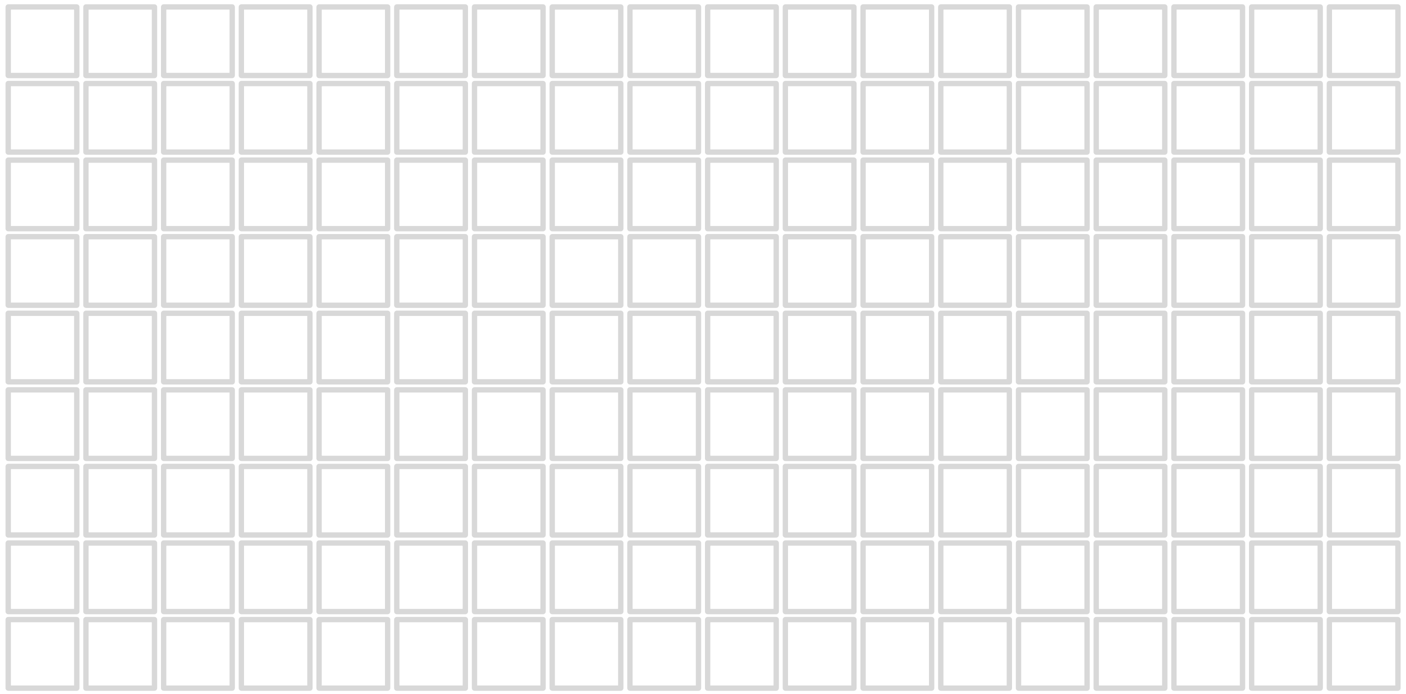
V100



- **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

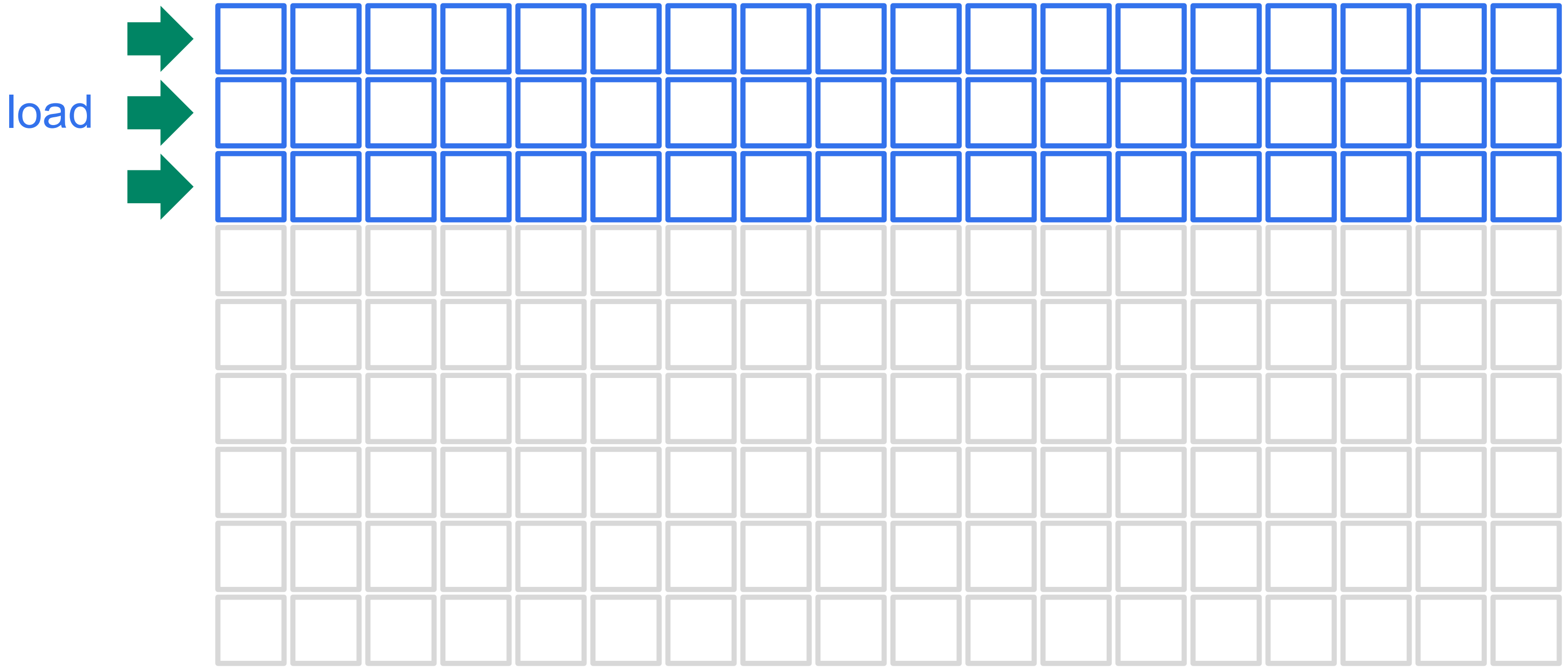
CACHING IN REGISTERS

No data loaded initially



CACHING IN REGISTERS

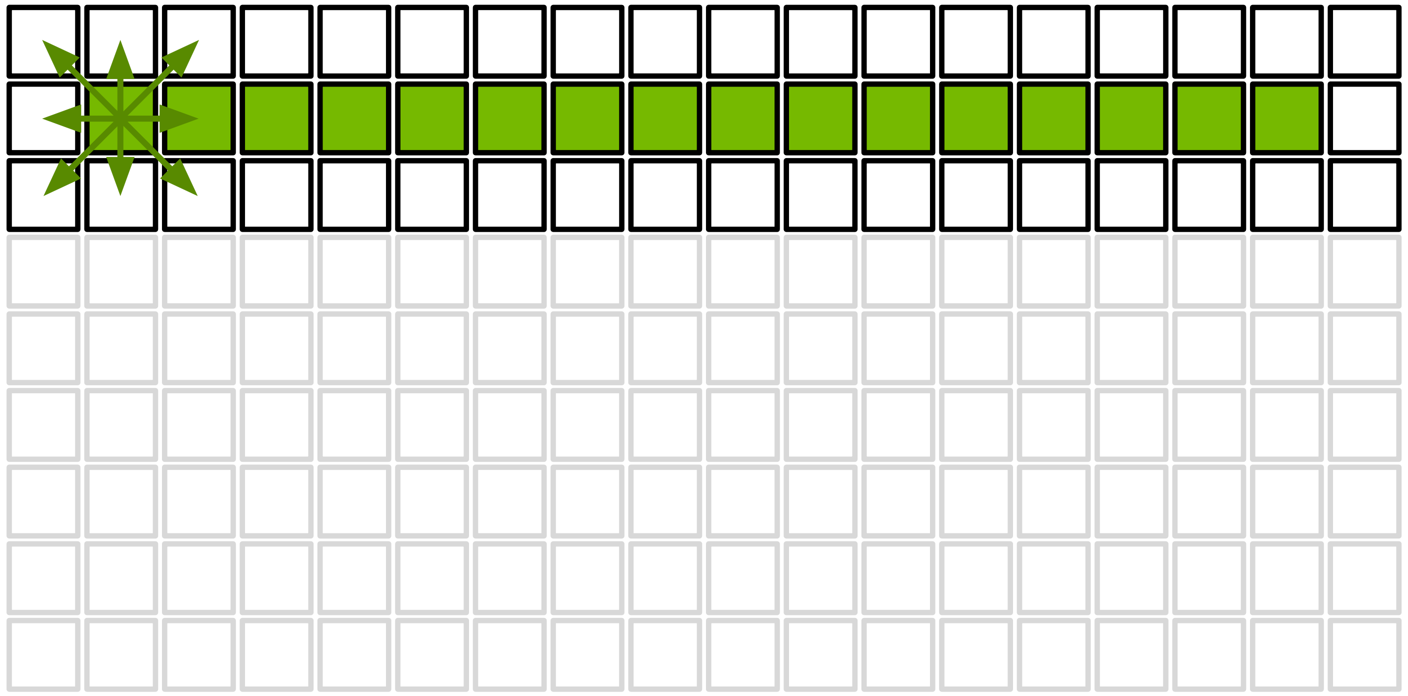
Load first set of data



CACHING IN REGISTERS

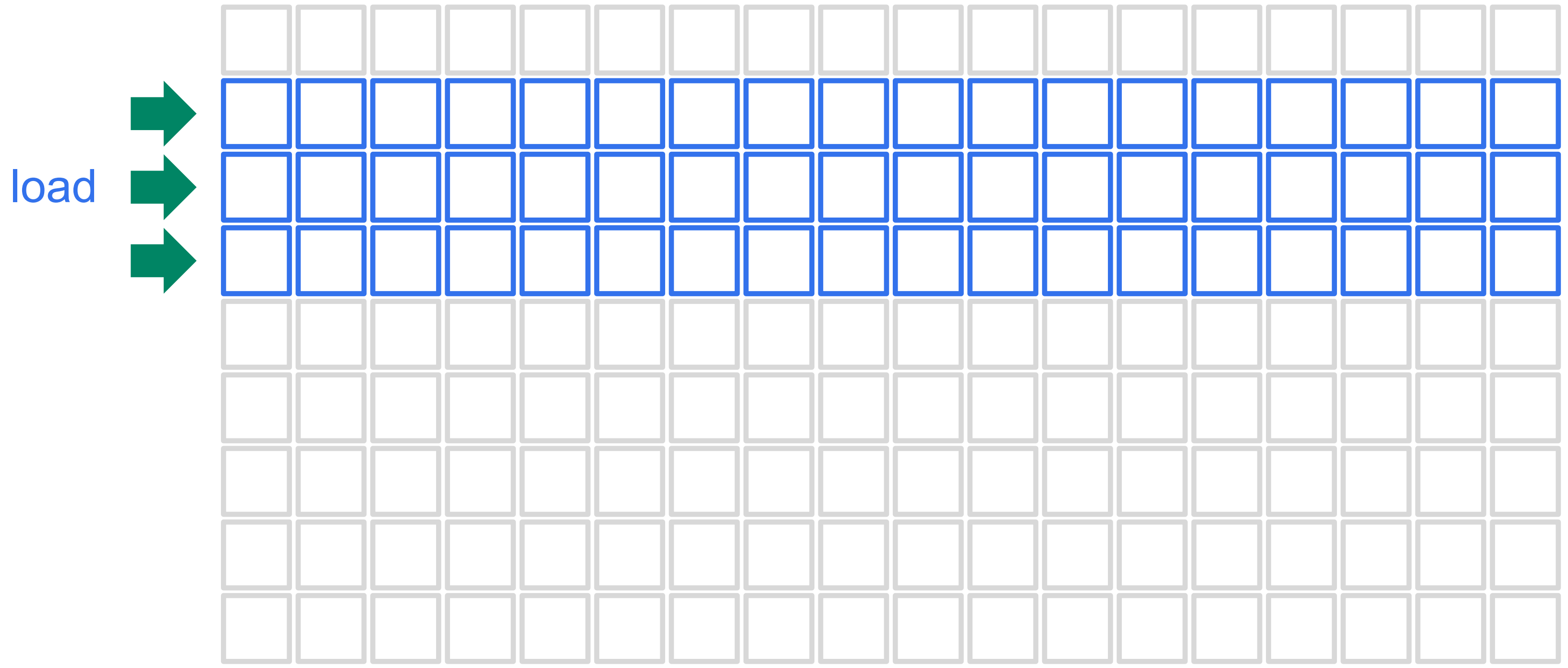
Perform calculation

Stencil



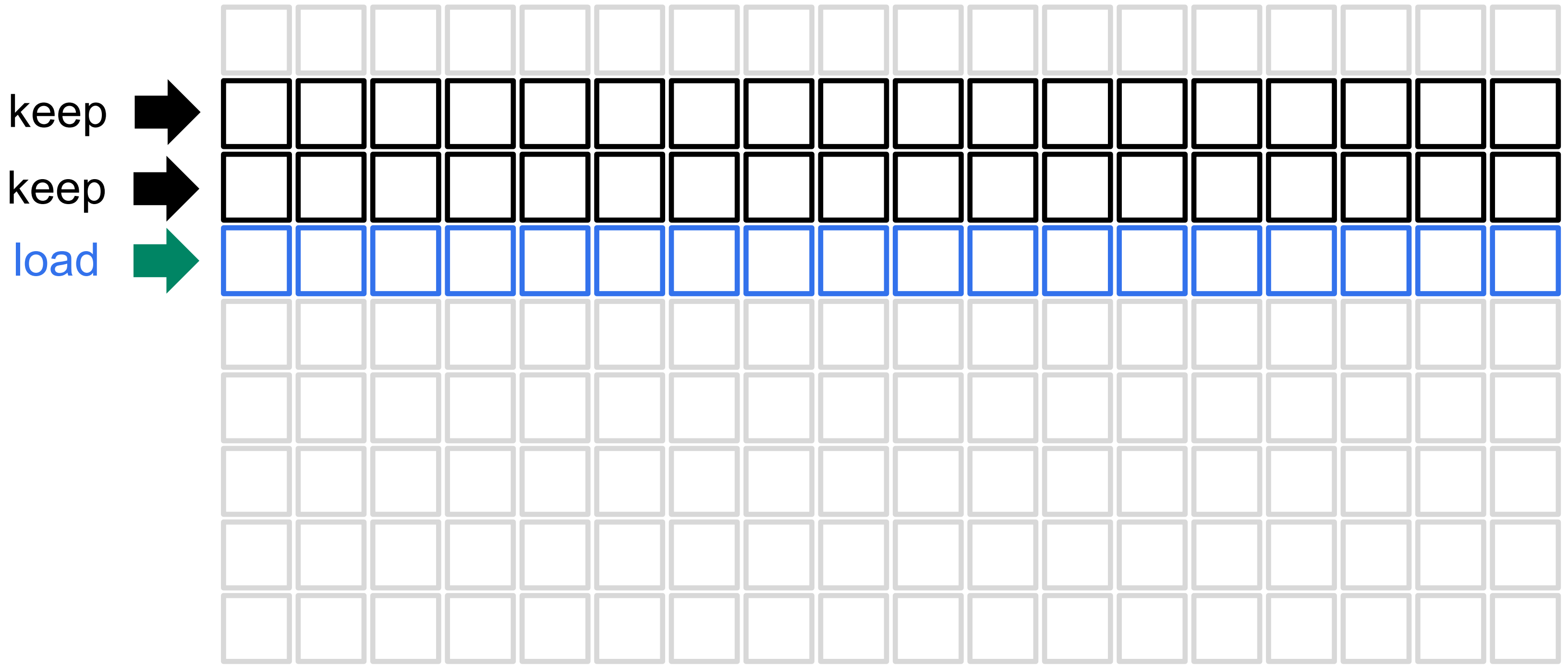
CACHING IN REGISTERS

Naively load next set of data?



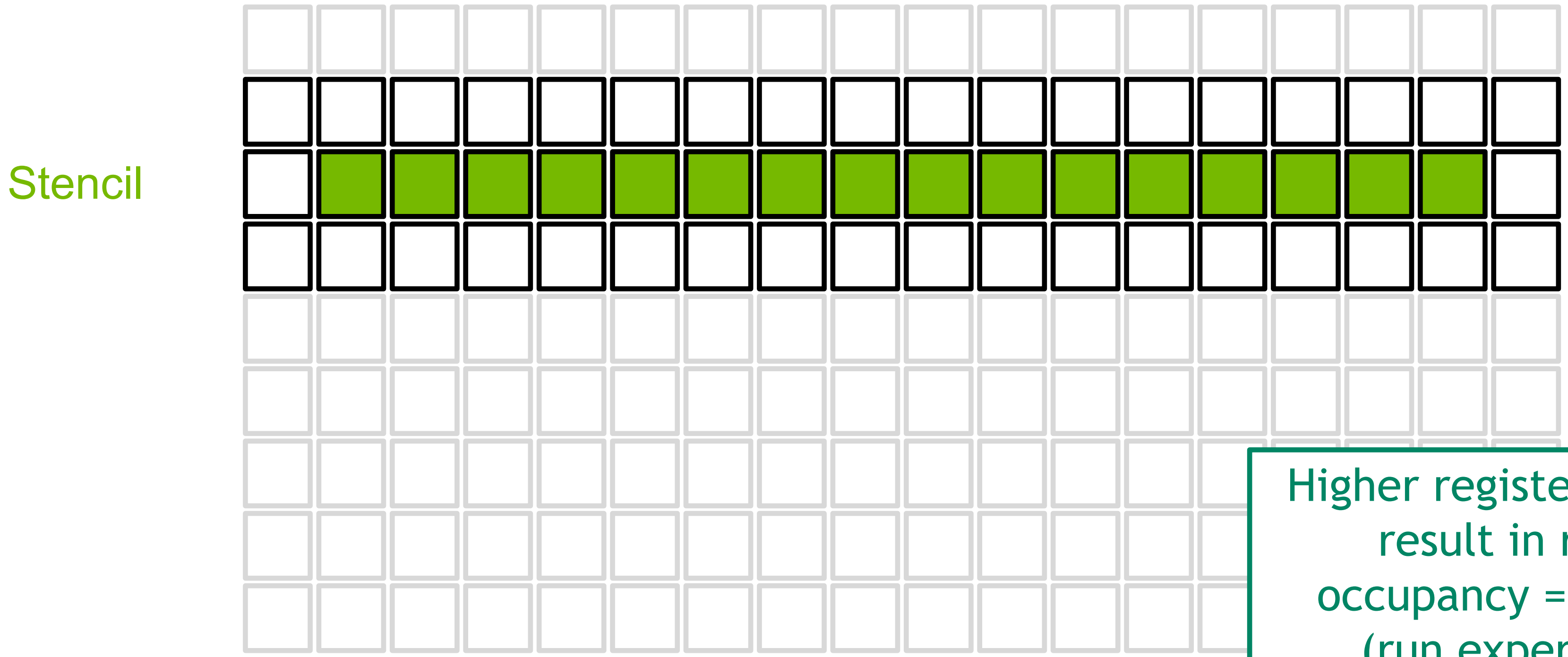
CACHING IN REGISTERS

Reusing already loaded data is better



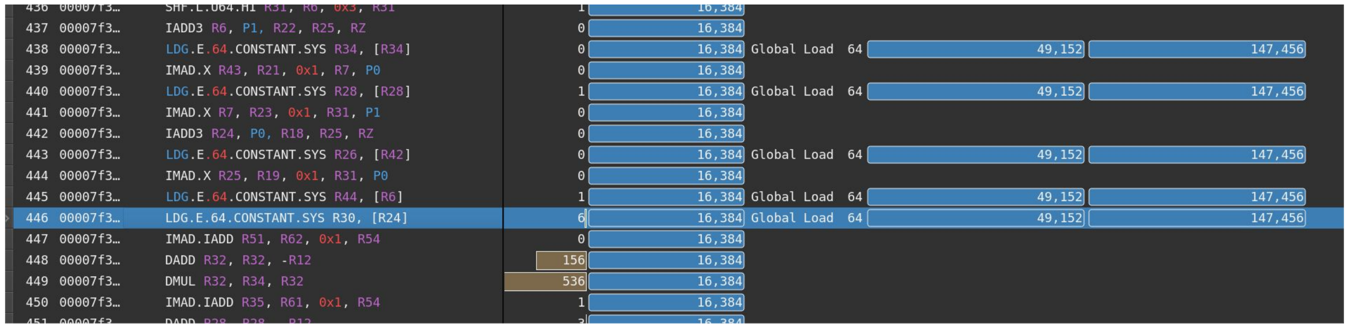
CACHING IN REGISTERS

Repeat



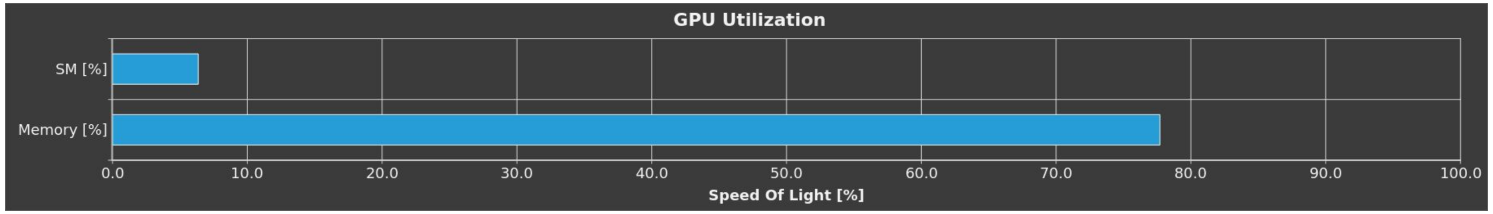
Higher register usage may result in reduced occupancy => trade off (run experiments!)

THE EFFECT OF REGISTER CACHING



```
436  00007f3…   SHF.L.U64.HI R31, R6, 0x3, R31        1  16,384
437  00007f3…   IADD3 R6, P1, R22, R25, RZ            0  16,384
438  00007f3…   LDG.E.64.CONSTANT.SYS R34, [R34]      0  16,384 Global Load  64  49,152  147,456
439  00007f3…   IMAD.X R43, R21, 0x1, R7, P0          0  16,384
440  00007f3…   LDG.E.64.CONSTANT.SYS R28, [R28]      1  16,384 Global Load  64  49,152  147,456
441  00007f3…   IMAD.X R7, R23, 0x1, R31, P1          0  16,384
442  00007f3…   IADD3 R24, P0, R18, R25, RZ           0  16,384
443  00007f3…   LDG.E.64.CONSTANT.SYS R26, [R42]      0  16,384 Global Load  64  49,152  147,456
444  00007f3…   IMAD.X R25, R19, 0x1, R31, P0         0  16,384
445  00007f3…   LDG.E.64.CONSTANT.SYS R44, [R6]       1  16,384 Global Load  64  49,152  147,456
446  00007f3…   LDG.E.64.CONSTANT.SYS R30, [R24]      6  16,384 Global Load  64  49,152  147,456
447  00007f3…   IMAD.IADD R51, R62, 0x1, R54          0  16,384
448  00007f3…   DADD R32, R32, -R12                 156  16,384
449  00007f3…   DMUL R32, R34, R32                  536  16,384
450  00007f3…   IMAD.IADD R35, R61, 0x1, R54          1  16,384
451  00007f3…   DADD R28, R28, -R12                   3  16,384
```

Transactions for cached loads reduced by a factor of 8



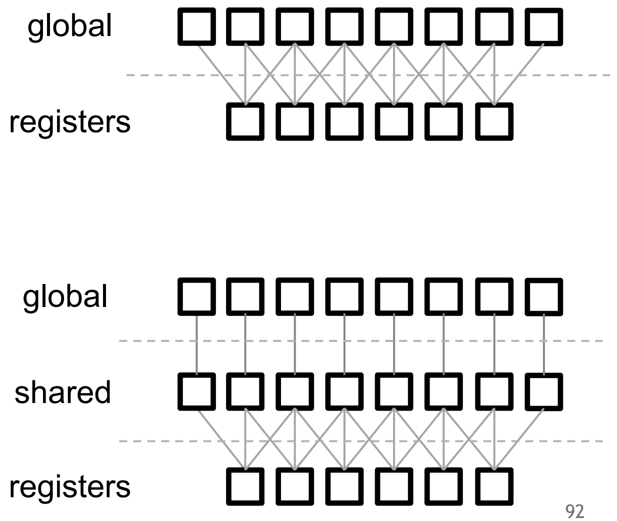
Memory utilization still high, but transferring less redundant data

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

SHARED MEMORY

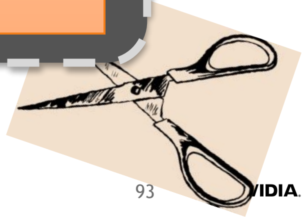
- Programmer-managed cache
- Great for caching data reused across threads in a CTA
- 128KB split between shared memory and L1 cache per SM
  - Each block can use at most 96KB shared memory on GV100
  - Search for `cudaFuncAttributePreferredSharedMemoryCarveout` in the docs

```
__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];
}
```



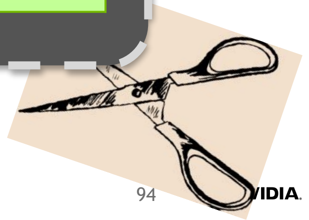
PERF-OPT QUICK REFERENCE CARD

Category:	Bandwidth Bound – Register Caching
Problem:	Data is reused within threads and memory bw utilization is high
Goal:	Reduce 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:	• Assign registers to cache data • Avoid storing and reloading data (possibly by assigning work to threads differently) • Avoid register spilling



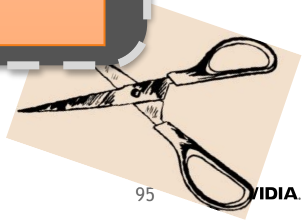
PERF-OPT QUICK REFERENCE CARD

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: • Annotate read-only pointers with const __restrict__ • Use __ldg() intrinsic



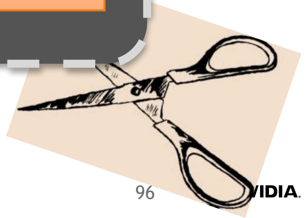
PERF-OPT QUICK REFERENCE CARD

Category:	Device Mem Bandwidth Bound – Shared Memory
Problem:	Too much data movement
Goal:	Reduce 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: • Shared Memory • (or Registers) • (or Constant Memory) • (or Texture Cache)



PERF-OPT QUICK REFERENCE CARD

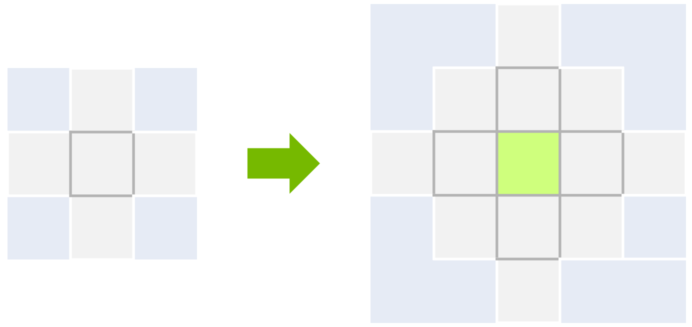
Category:	Shared Mem Bandwidth Bound – Shared Memory
Problem:	Shared memory bandwidth bottleneck
Goal:	Reduce amount of data traffic to/from global mem
Indicators:	Shared memory loads or stores saturate
Strategy:	Reduce Bank Conflicts (insert padding) Move data from shared memory into registers Change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

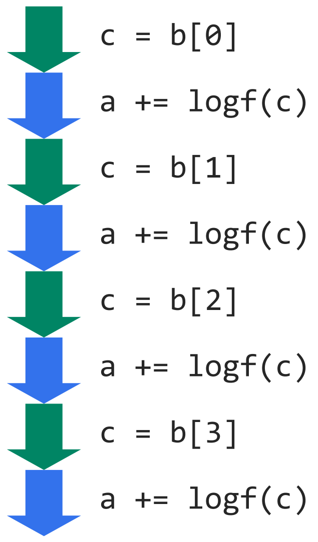
Instruction level parallelism can reduce dependencies

```
a = b + c;  // Independent ADDs
d = e + f;
```

ILP AND MEMORY ACCESSES

No ILP

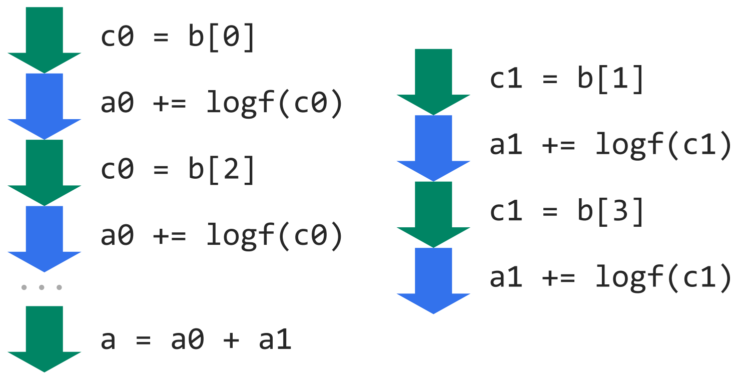
```
float a = 0.0f;
for( int i = 0 ; i < N ; ++i )
  a += logf(b[i]);
```



```
c = b[0]
a += logf(c)
c = b[1]
a += logf(c)
c = b[2]
a += logf(c)
c = b[3]
a += logf(c)
```

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
```



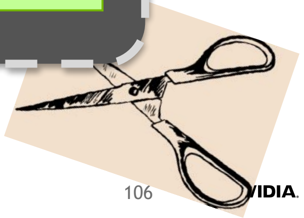
```
c0 = b[0]               c1 = b[1]
a0 += logf(c0)          a1 += logf(c1)
c0 = b[2]               c1 = b[3]
a0 += logf(c0)          a1 += logf(c1)
...
a = a0 + a1
```

#pragma unroll is useful to extract ILP

Manually rewrite code if not a simple loop

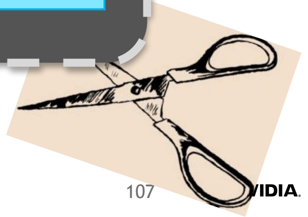
PERF-OPT QUICK REFERENCE CARD

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:	• 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:	• 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

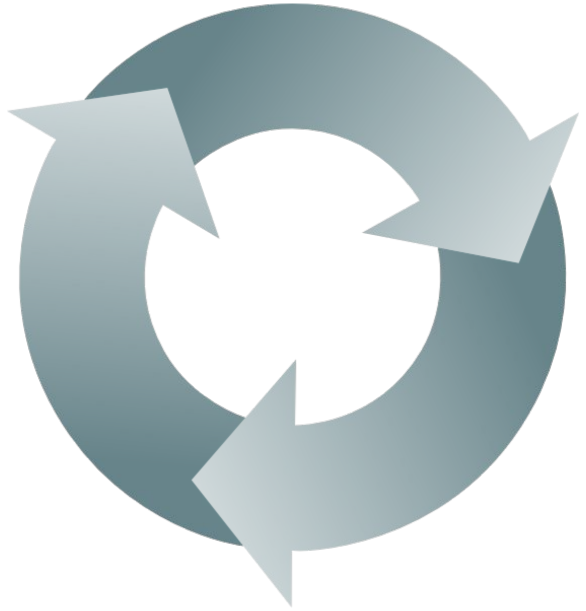


SUMMARY

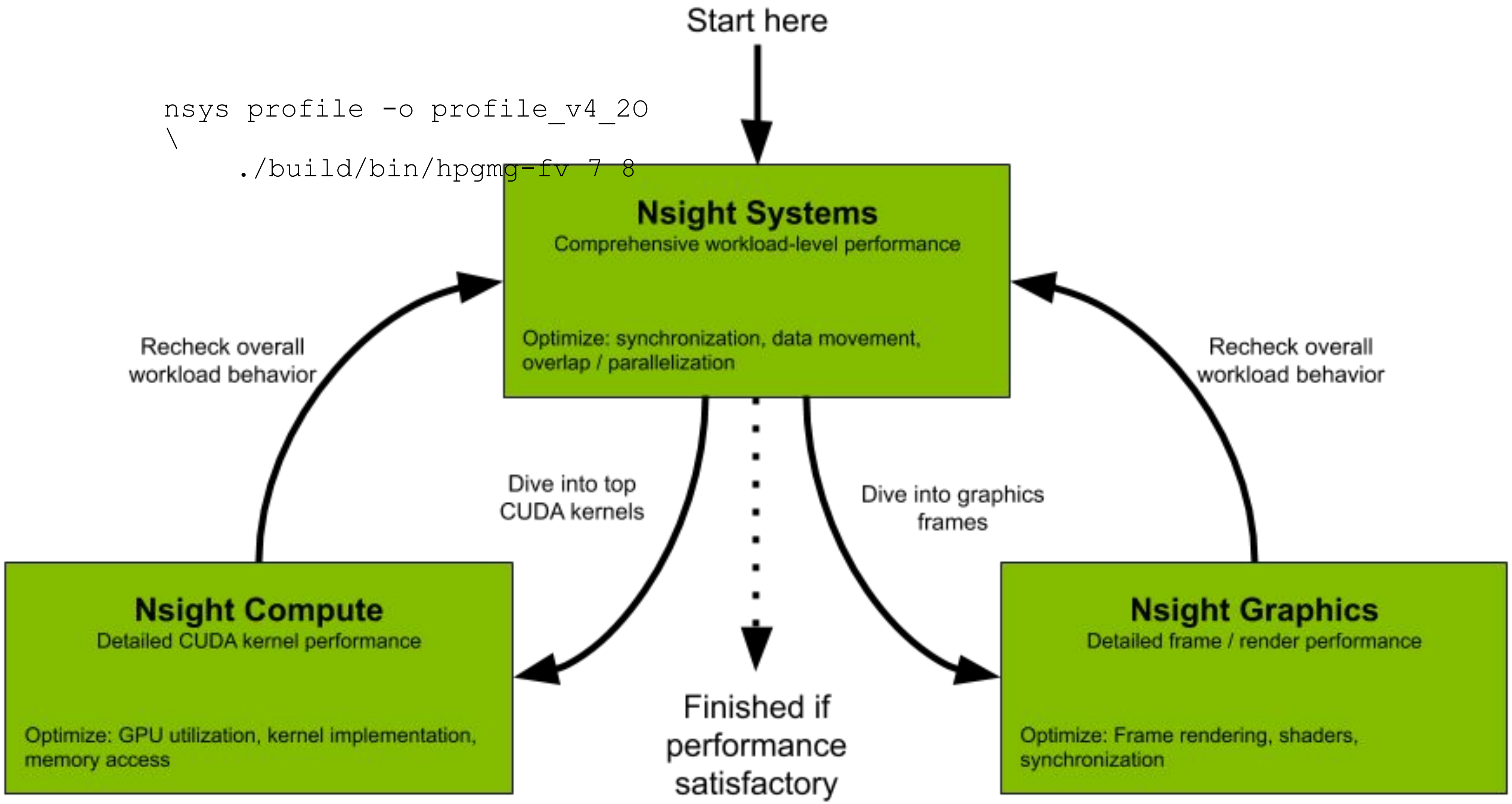
SUMMARY

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!



```
nsys profile -o profile_v4_20
\
     ./build/bin/hpgmg-fv 7 8
```


Start here
Nsight Systems
Comprehensive workload-level performance
Optimize: synchronization, data movement, overlap / parallelization
Recheck overall workload behavior
Recheck overall workload behavior
Dive into top CUDA kernels
Dive into graphics frames
Nsight Compute
Detailed CUDA kernel performance
Optimize: GPU utilization, kernel implementation, memory access
Finished if performance satisfactory
Nsight Graphics
Detailed frame / render performance
Optimize: Frame rendering, shaders, synchronization

```
nv-nsight-cu-cli -o profile_v4_20 \
     --kernel-regex ".*smooth_kernel*" \
     --launch-count 1 ./build/bin/hpgmg-fv 7
8
```

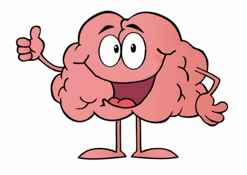
GUIDING OPTIMIZATION EFFORT

“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

$\}$ Nsight Systems

$\}$ Nsight Compute



REFERENCES

CUDA Documentation

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

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

Ampere Tuning Guide: https://docs.nvidia.com/cuda/ampere-tuning-guide/

NVIDIA Developer Blog on HPGMG

https://devblogs.nvidia.com/high-performance-geometric-multi-grid-gpu-acceleration/

Nsight Tools

https://devblogs.nvidia.com/migrating-nvidia-nsight-tools-nvvp-nvprof/
https://devblogs.nvidia.com/transitioning-nsight-systems-nvidia-visual-profiler-nvprof/
https://devblogs.nvidia.com/using-nsight-compute-to-inspect-your-kernels/

NVIDIA®