NVIDIA

VOLTA / TURING OPTIMIZATION

G. Thomas-Collignon, NVIDIA, GTC 2019 S9234

AGENDA

Quick review of basic optimization guidelines

New features in Turing

Using FP16 (case study)

Profiling codes on Turing

BACKGROUND

Quick review of basic optimization guidelines

- Little's law – Need enough parallelism to saturate our resources
- Need enough occupancy and Instruction Level Parallelism
- Memory coalescing & access patterns
- Avoid intra-warp divergence
- Avoid shared memory bank conflicts
- Overlap of computation / communication (streams, CUDA Graphs, MPS)

👉 GTC'18
S81006
Volta Architecture and Performance Optimization

TURING

What's new in Turing?

Many new features, including:

- Tensor Cores, now for FP16 and Integer
- RT Core – Real-time Ray Tracing
- Full speed FP16 (like P100 / V100)
- Unified L1 cache (similar to Volta)

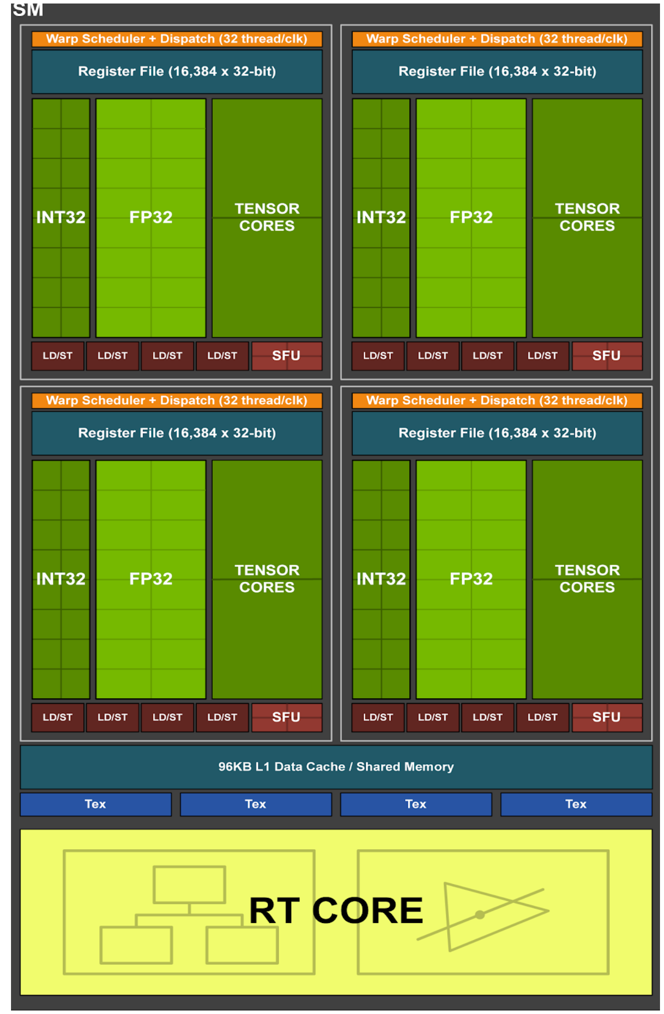
VOLTA / TURING SM

	V100	TU102
SMs	80	72
Compute Capability	70	75
FP64	32	2
INT32	64	64
FP32	64	64
Tensor Cores	8	8 (FP16 + Int)
RT Core	-	1
Register File	256 KB	256 KB
L1 and shmem	128 KB	96 KB
Max threads	2048	1024

Per SM

Volta binaries can run on Turing

Turing SM



RT CORES

New in Turing

👉 S9768
New features
in Optix 6.0

- Ray Tracing acceleration
- Exposed in NVIDIA Optix
- Easy interop with CUDA
- Used also for non-raytracing problems

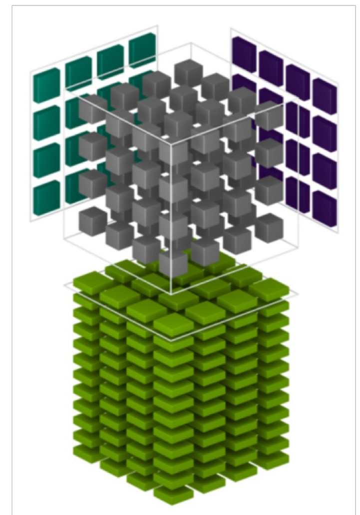
Docs and more: http://raytracing-docs.nvidia.com/optix/index.html

TENSOR CORES

New in Volta, Extended in Turing

S9926
Tensor Core Performance
The Ultimate Guide

GPU	SMs	Total	Peak FP16	Peak INT8	Peak INT4	Peak INT1
V100	80	640	125 TFlops	N.A.	N.A.	N.A.
TU102	72	576	130 TFlops	261 Tops	522 Tops	2088 Tops



half precision inputs → single precision or half precision accumulator

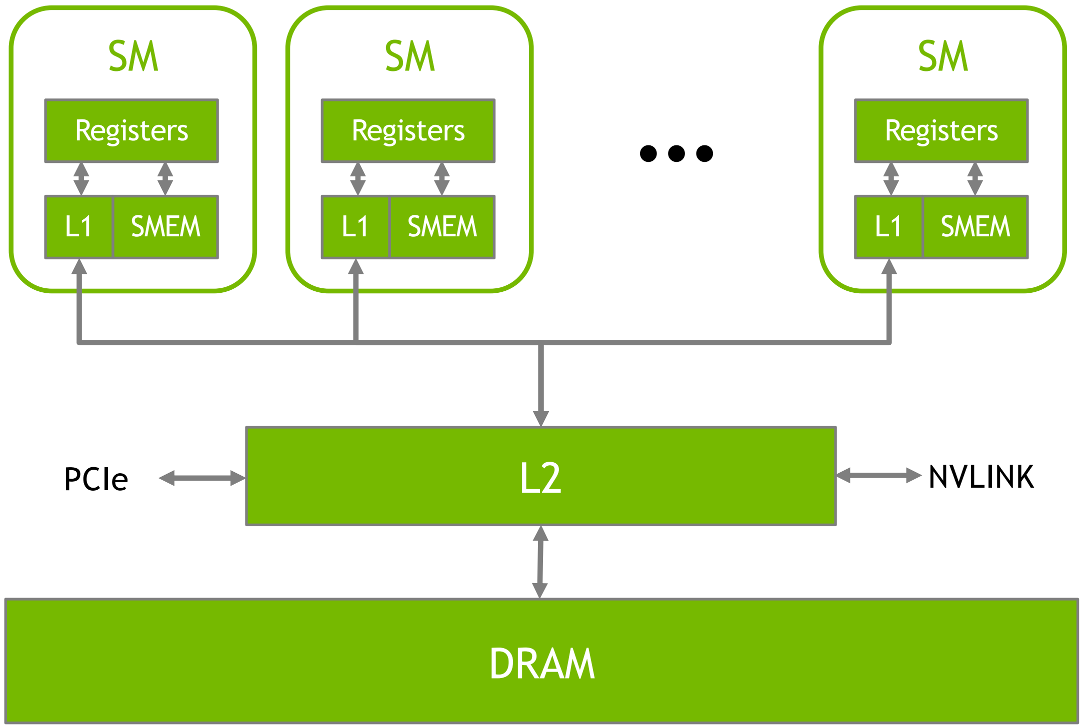
Turing $\left\{ \begin{array}{l} \text{8bit/4bit INT inputs → 32-bit INT accumulator} \\ \text{1bit Binary inputs → 32-bit INT accumulator (XOR + POPC)} \end{array} \right.$

Used via CUBLAS, CUDNN, CUTLASS, TensorRT
Exposed in CUDA 10 (4bit INT and 1bit binary are experimental)

Volta binaries using Tensor Cores should be recompiled for Turing to achieve full throughput

MEMORY SUBSYSTEM

Volta / Turing



Up to 80 Streaming Multiprocessors
256KB register file per SM

Unified Shared Mem / L1 Cache

Up to 6 MB L2 Cache

Global Memory

Volta: HBM2, 16, 32 GB
Turing: GDDR6 <= 48GB

TURING

L1 / Shared memory

Turing inherited the unified L1 introduced in Volta

	Volta	Turing
Total L1+Shared	128 KB	96 KB
Max shared	96 KB	64 KB
Possible splits	6	2
Throughput	128 B/cycle	64 B/cycle

Default max shared memory = **48 KB**.

Need to explicitly opt-in for > 48 KB on Volta and Turing

Volta binaries using more than 64 KB of shared memory won't run on Turing

L1/SHM

Variable split

By default, the driver is using the configuration that will maximize occupancy

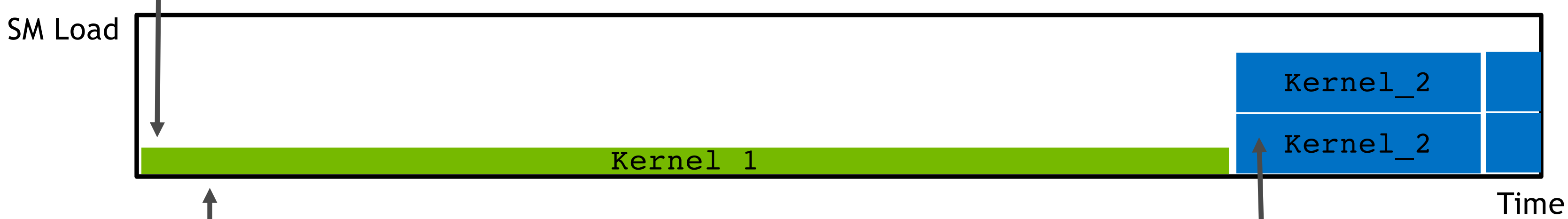
Shared / L1 splits	
Volta	**Turing**
96KB / 32KB 64KB / 64KB 32KB / 96KB 16KB / 112KB 8KB / 120KB 0KB / 128 KB	64 KB / 32 KB 32 KB / 64 KB

	Configuration used	
Examples	**Volta**	**Turing**
kernel_1 **0KB Shared Mem** Other resources: **up to 16 blocks/SM**	0 KB Shared 128 KB L1 16 blocks /SM	32KB Shared 64 KB L1 16 blocks/SM
kernel_2 **40 KB Shared Mem** Other resources: **up to 4 blocks/SM**	96 KB Shared 32 KB L1 2 blocks / SM	64 KB Shared 32 KB L1 1 block / SM

L1/SHM

When to change the default split

Already running **kernel_1** (no shared memory), light load 1 block / SM,
Volta : Full L1, no shared memory
Turing: Max L1, 32 KB shared memory



Launching kernel_2 concurrently (40 KB shared/ block)
Not enough shared memory with current configuration

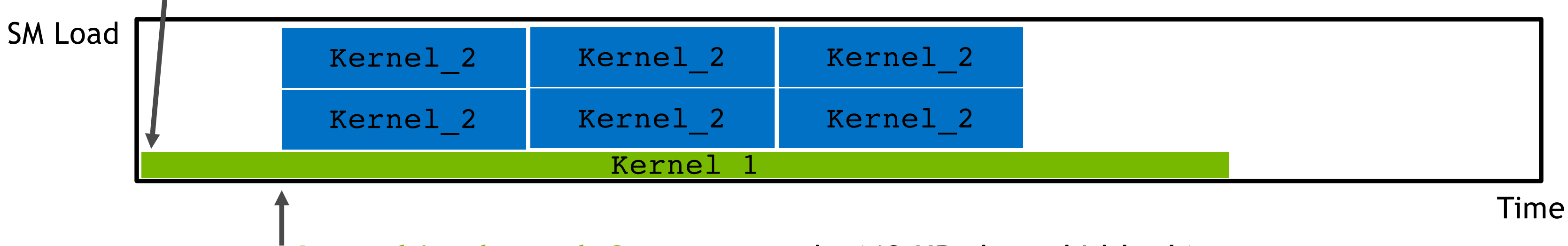
Kernel_2 runs after kernel_1 has completed

L1/SHM

When to change the default split

Forcing kernel_1 to run with max shared memory config:

```
cudaFuncSetAttribute (kernel_1,
                      cudaFuncAttributePreferredSharedMemoryCarveout,
                      cudaSharedmemCarveoutMaxShared);
kernel_1<<<blocks,threads,0,stream >>>()
```



Launching kernel_2 concurrently (40 KB shared/ block)
Kernel_2 can now run concurrently with kernel_1

Other possible reason: To run at a lower occupancy, less blocks, larger L1

FP64, FP32, FP16

$$(-1)^{sign} \times 2^{exponent} \times \left(1 + \frac{mantissa}{2^{mantissa\_bits}}\right)$$

	FP64	FP32	FP16
Exponent bits	11	8	5
Mantissa bits	52	23	10
Largest number	$\approx 1.7 \times 10^{308}$	$\approx 3.4 \times 10^{38}$	65504.0
Smallest normal > 0	$\approx 2.2 \times 10^{-308}$	$\approx 1.2 \times 10^{-38}$	$\approx 6.1 \times 10^{-5}$
Smallest denormal > 0	$\approx 4.9 \times 10^{-324}$	$\approx 1.4 \times 10^{-45}$	$\approx 5.9 \times 10^{-8}$

CUDA FP16

- CUDA provides **half** and **half2** types and instrinsics in **cuda_fp16.h**
- Use **CUDA 10** for the best FP16 support:

  CUDA 8: `v1 = __hadd2 (v1, __hadd2 (v2, __hmul2 (v3, v3)));`

  CUDA 9.2: `v1 += v2 + (v3 * v3);`

  CUDA 10: Better support for half2, and atomics

- FP16 is available on Pascal and newer GPUs.

- **Host side:**

  CUDA provides functions to **assign / convert** values to FP16 on host.

HALF VS HALF2

half

Not used	v1

+

Not used	v2

=

Not used	v1+v2

32-bit registers

1 result per instruction
Same peak Flops as FP32
Generates **16-bit** loads & stores

half2

v1.y	v1.x

+

v2.y	v2.x

=

v1.y + v2.y	v1.x + v2.x

32-bit registers

2 results per instruction (SIMD)
2x the peak Flops of FP32
Generates **32-bit** loads & stores

Full compute throughput can only be achieved with **half2 type**.

Bandwidth-bound codes can still get ~2x speedup with **half type**

FP16

3 levels of peak performance

Instruction type	V100 Peak	Typical use
Tensor Cores	125 TFlops	Matrix products
half2	31 TFlops	Compute-bound kernels
half	15 TFlops	Bandwidth-bound kernels

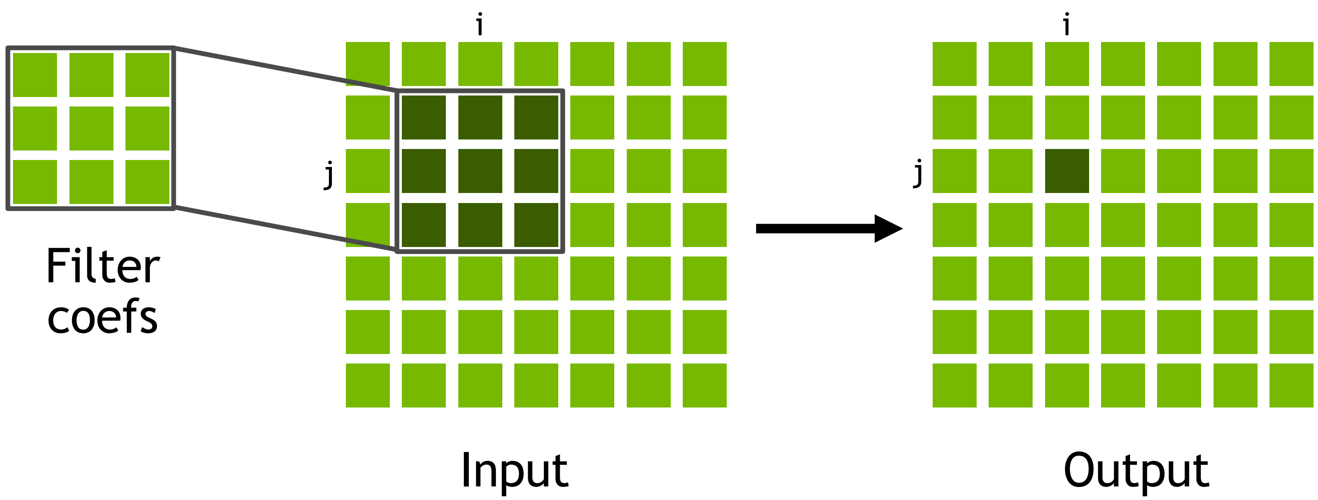
2D FILTER

Case study

2D non-separable filter of radius *r*:

$$Output[i,j] = \sum_{k=-r}^{r} \sum_{l=-r}^{r} coef[k,l] \times input[i+k, j+l]$$

Radius 1
3x3 Filter



ANALYSIS

Arithmetic intensity

For each point, a filter of **diameter N** on **FP32 data:**

Computation: N^2 mults + N^2 -1 adds = **$2 \times N^2 - 1$ Flops**

Memory: 1 read, 1 write = 8 bytes

Assuming the halos can be cached / amortized

Arithmetic intensity = $\dfrac{2 \times N^2 - 1}{8}$ **Flops / Byte**

ARITHMETIC INTENSITY

Expected behavior on Volta

Volta V100 FP32 = 15.6 Tflops/s, BW = 0.9 TB/s = **17 Flops / Byte**

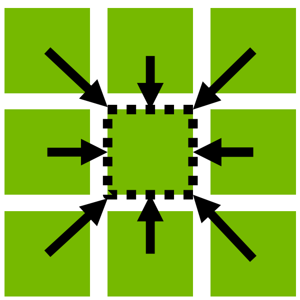
Filter Size	Flops	Flops/Byte
3x3	17	2.1
5x5	49	6.1
7x7	97	12.1
9x9	161	20.1
11x11	241	30.1
13x13	337	42.1

Bandwidth bound

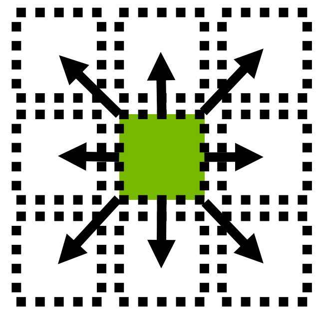
Compute bound

GPU IMPLEMENTATION

Gather vs Scatter approaches



3x3 Filter



Gather approach:
9 input values needed
to compute 1 output value

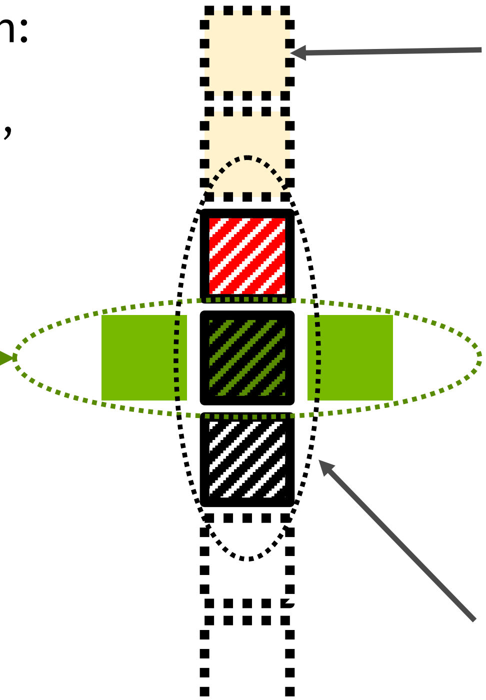
Typically implemented
with shared memory

Scatter approach:
1 input value contributes
to 9 output values

GPU IMPLEMENTATION

Each thread processes one column:

Each thread reads 3 input values,
contributing to 3 output values



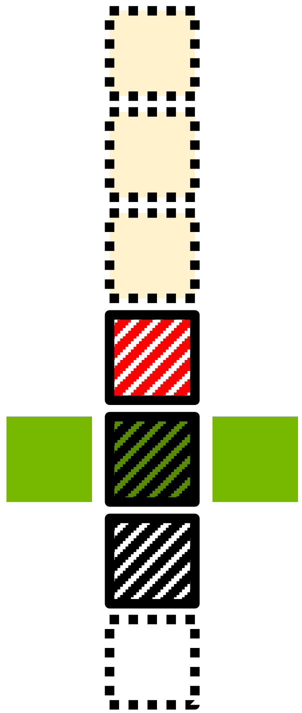
Previous results

3 new input values

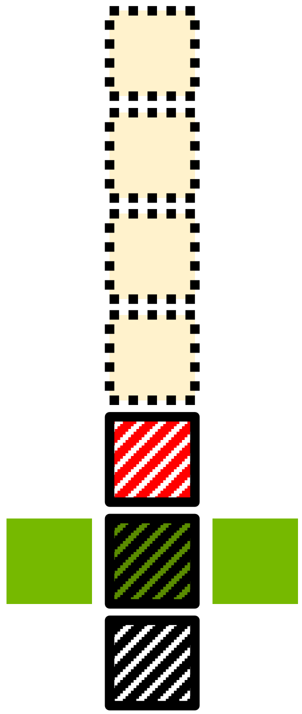
3 partial results
(sliding window)

3x3 Filter

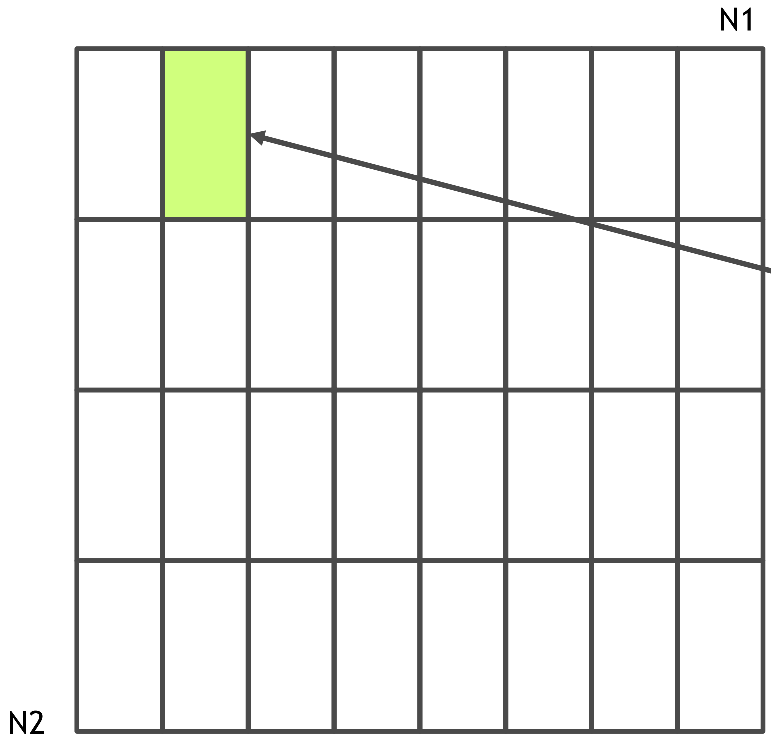
GPU IMPLEMENTATION



GPU IMPLEMENTATION



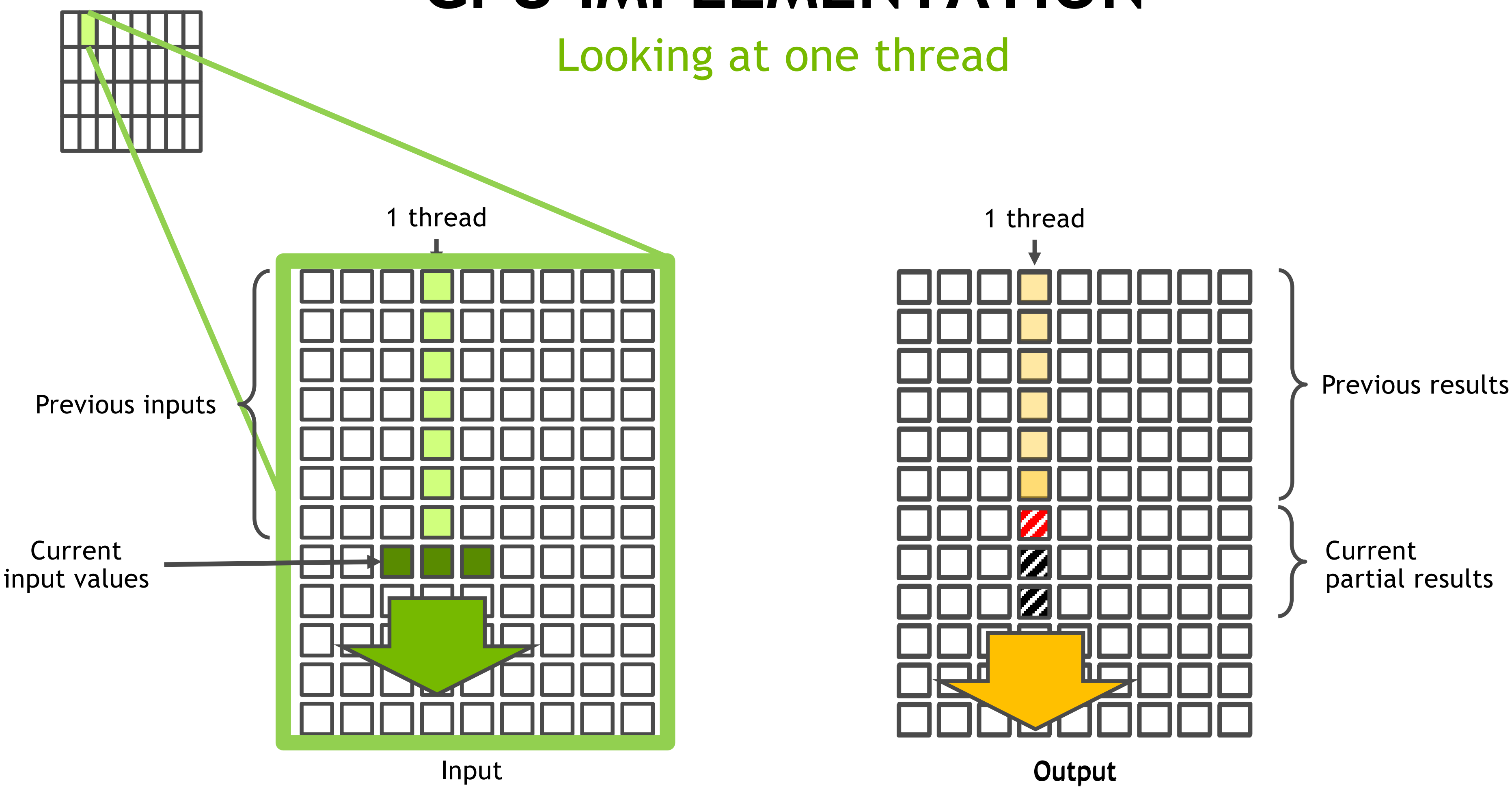
GPU IMPLEMENTATION



Each thread block will process a 2D tile

GPU IMPLEMENTATION

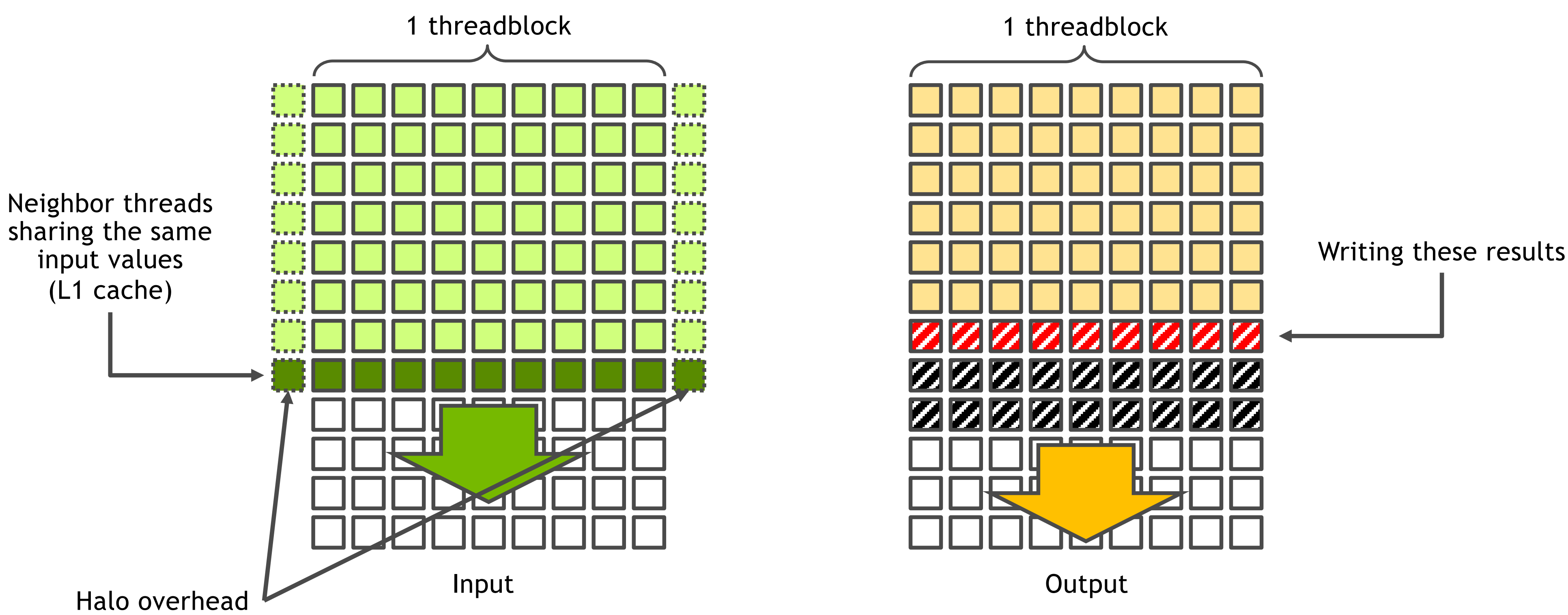
Looking at one thread



NVIDIA.

GPU IMPLEMENTATION

Looking at one threadblock



V100 RESULTS

16K x 16K input, FP32

		V100	
Filter Size	Time (ms)	TFlops	BW (GB/s)
3x3	2.9	1.6	730
5x5	3.0	4.3	704
7x7	3.3	**8.0**	658
9x9	3.6	12.1	599
11x11	4.8	13.4	444
13x13	6.5	13.8	328

~6x more Flops similar time

~80% peak bandwidth

~80% peak TFlops

V100 Peak = 15.6 FP32 Tflops, 900 GB/s

FP16 STRATEGIES

Float to Half Conversion

Very few code changes (`float -> half`)

Input data is converted to half

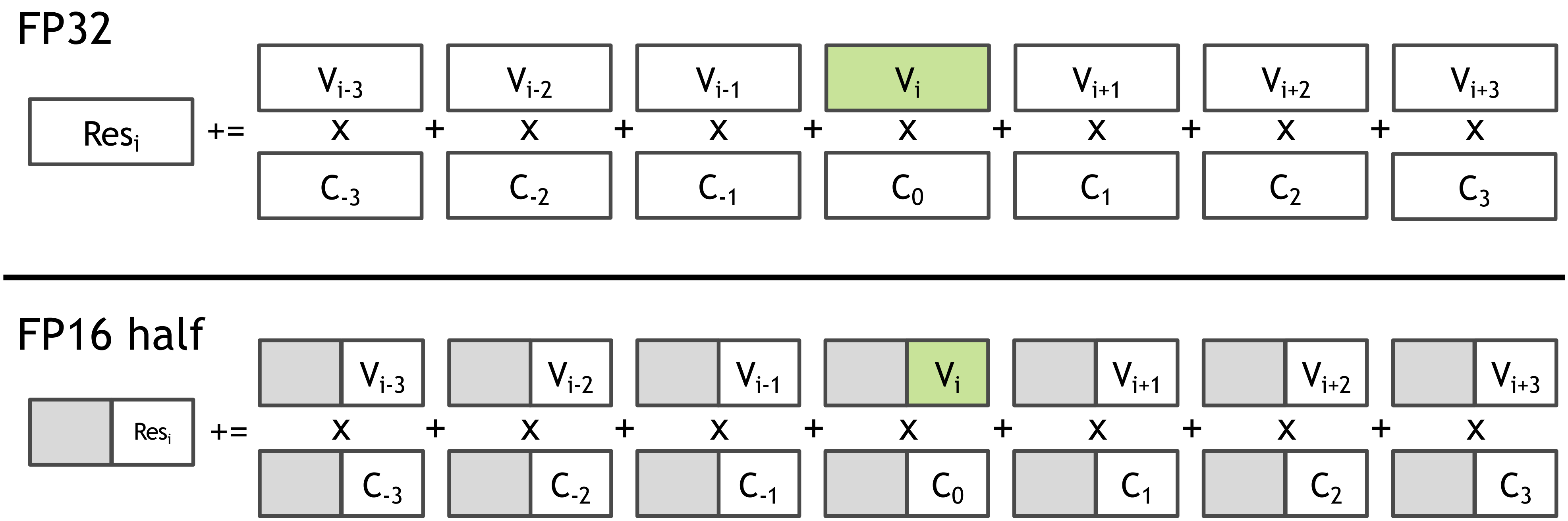
Filter coefficients in constant memory can be half or float

Expected results:

- Speed up ~2x for the bandwidth-bound kernels
- Similar time for the compute-bound kernels (same peak Flops performance)

FLOAT TO HALF

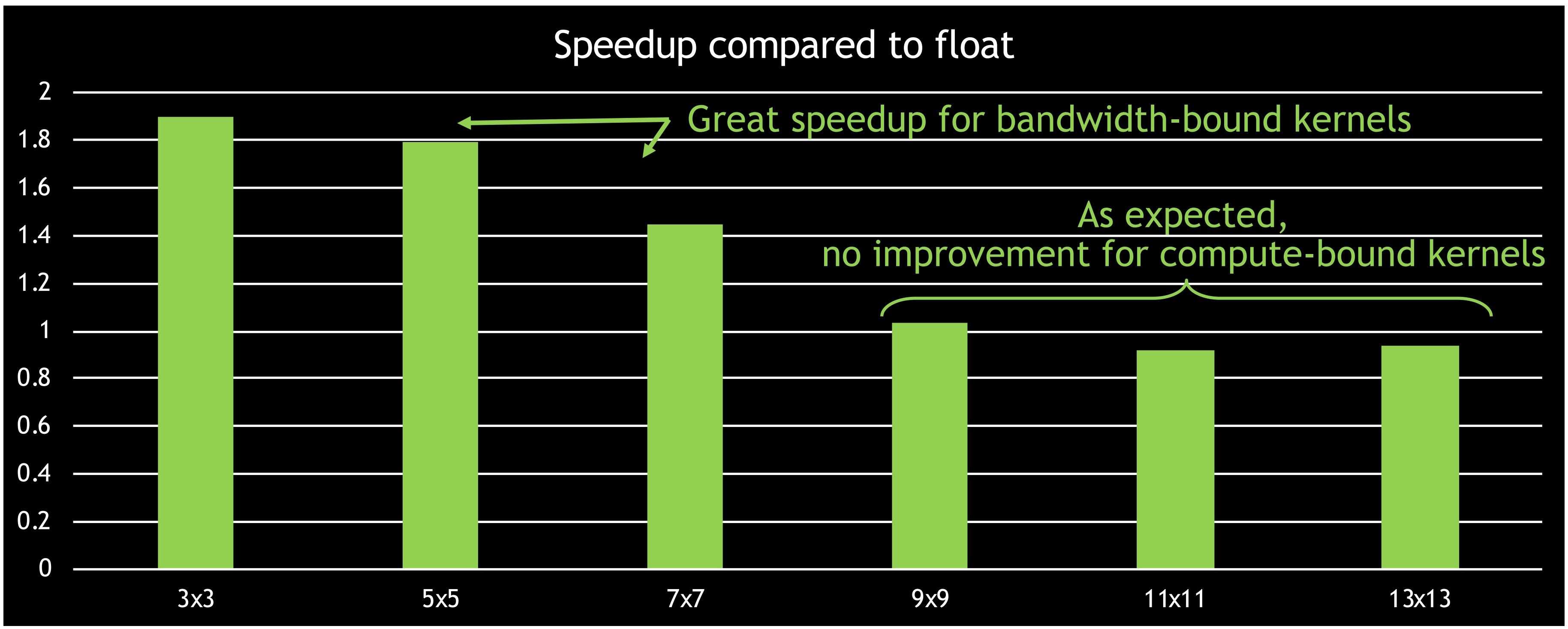
Updating one partial result



Transferring half the bytes to/from memory, same number of registers

V100 RESULTS

V100, 16K x 16K input, FP16 half



FP16 STRATEGIES

Float to Half2 Conversion

Running into typical "vectorization" issues.

Input data is converted to half2

Filter coefficients converted to half2

Expected results:

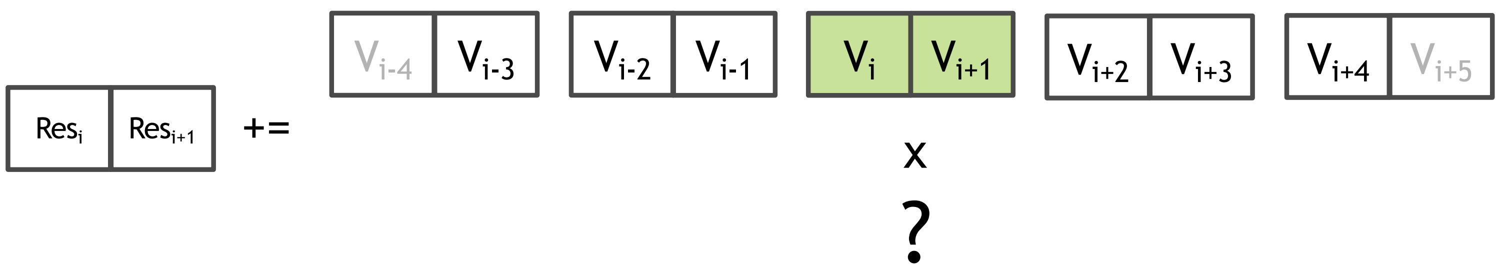
- Speed up ~2x for the bandwidth-bound kernels
- Speed up ~2x for the compute-bound kernels

FP16 STRATEGIES

Float to Half2: Vectorization issues

How can we compute the partial result, with the inputs packed in half2?

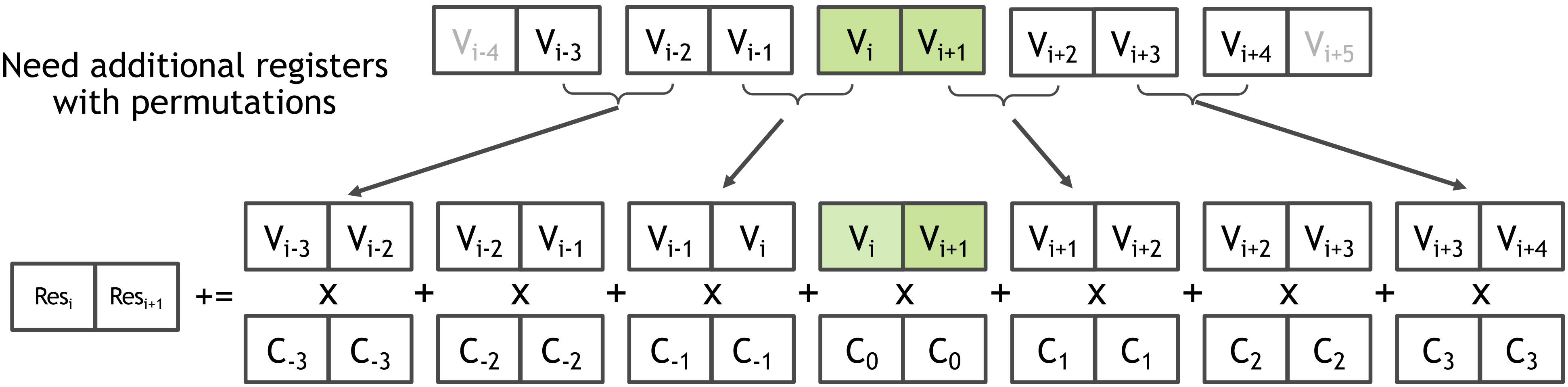
Need to write the filter for 2-way SIMD



FP16 STRATEGIES

Float to Half2: SIMD version

Need additional registers with permutations

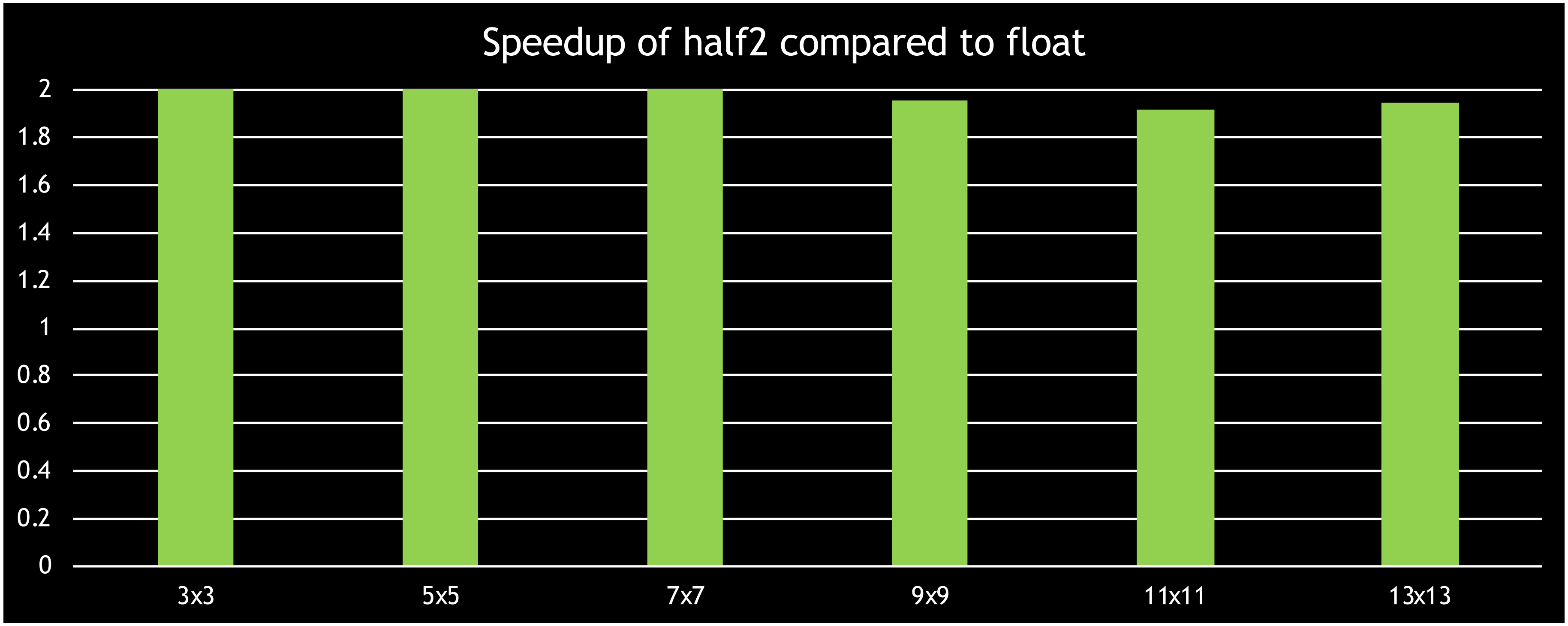


Coefficients are duplicated in both halves of the half2

Low impact on register count and extra instructions.

V100 RESULTS

V100, 16K x 16K input, FP16 half2



V100 RESULTS

16K x 16K input, FP16 half2

		V100		
Filter Size	Time (ms)	TFlops	BW (GB/s)	**Speedup vs FP32**
3x3	1.5	3.0	729	2.0x
5x5	1.5	8.6	704	2.0x
7x7	1.6	16.0	660	2.0x
9x9	1.8	23.6	588	1.96x
11x11	2.5	25.6	426	1.92x
13x13	3.4	27.0	320	1.95x

V100 Peak = 31.2 FP16 Tflops, 900 GB/s

FP16

Takeaways

- **Use half2** (or **Tensor Cores)** for compute-bound codes
- **(scalar) half** can be good enough for bandwidth-bound kernels
- **Speedups of ~2x** on compute and data transfers
- **Memory footprint** reduced by 2x
- Now available on **many GPUs**

How much precision does your problem require?

PROFILING

Profiling Tools for Turing

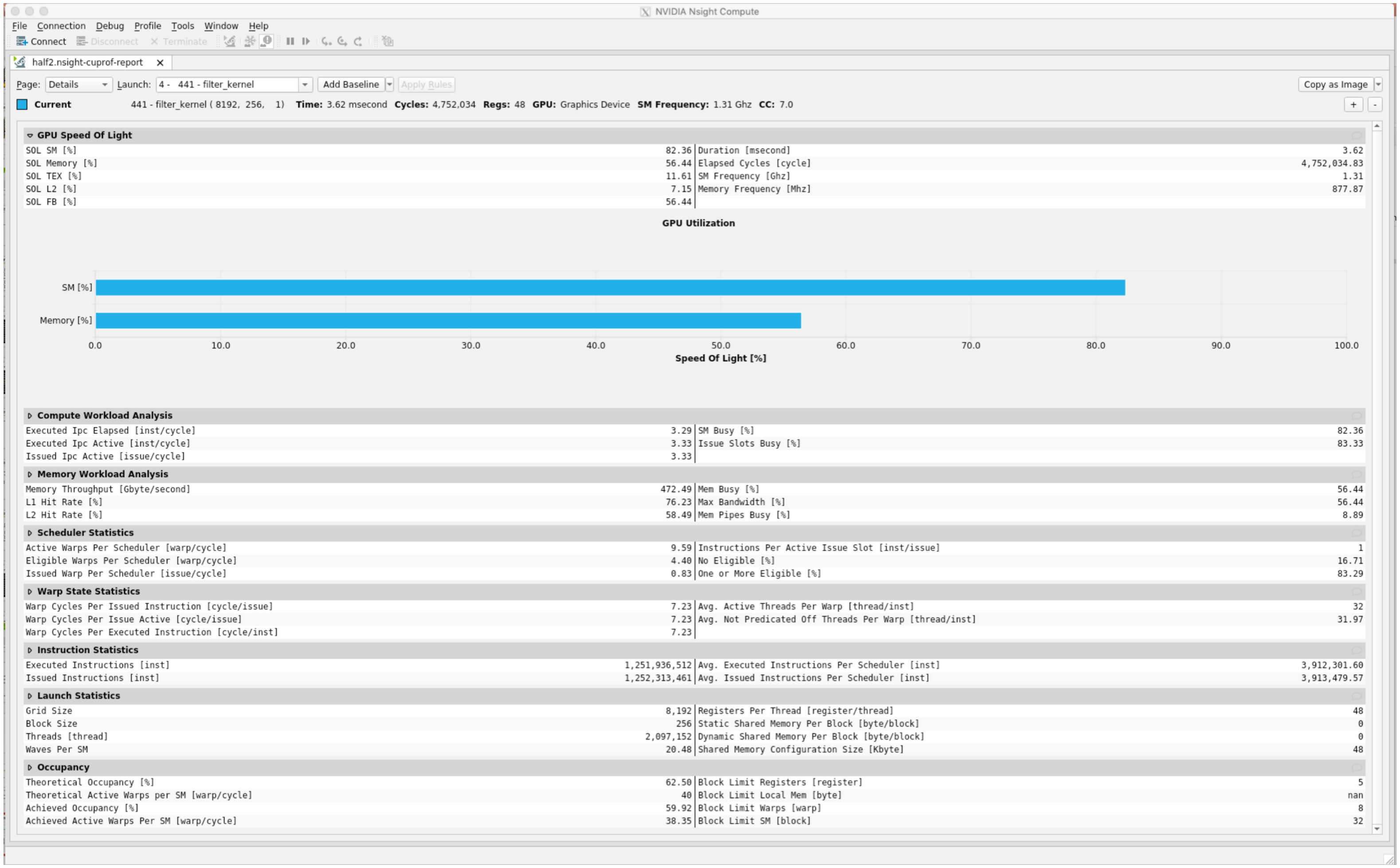
👉 S9345
CUDA Kernel Profiling Using
NVIDIA Nsight Compute

CUDA 10+ supports Turing

	Pascal	Volta	Turing
nvvp / nvprof	Full support	Full support	Tracing only (timeline)
Nsight Compute	Limited	Full support	**Full support**

Nsight Compute CLI: /usr/local/cuda-10.1/NsightCompute-2019.1/nv-nsight-cu-cli
Nsight Compute GUI: /usr/local/cuda-10.1/NsightCompute-2019.1/nv-nsight-cu

NSIGHT COMPUTE



TURING NEW FEATURES SUMMARY

- Binary compatible with Volta
- Unified L1
- Up to 64 KB Shared Memory per threadblock
- Full speed FP16
- Tensor Cores for FP16, Int8, Int4, Int1
- RT Cores (Optix)

Q & A


NVIDIA