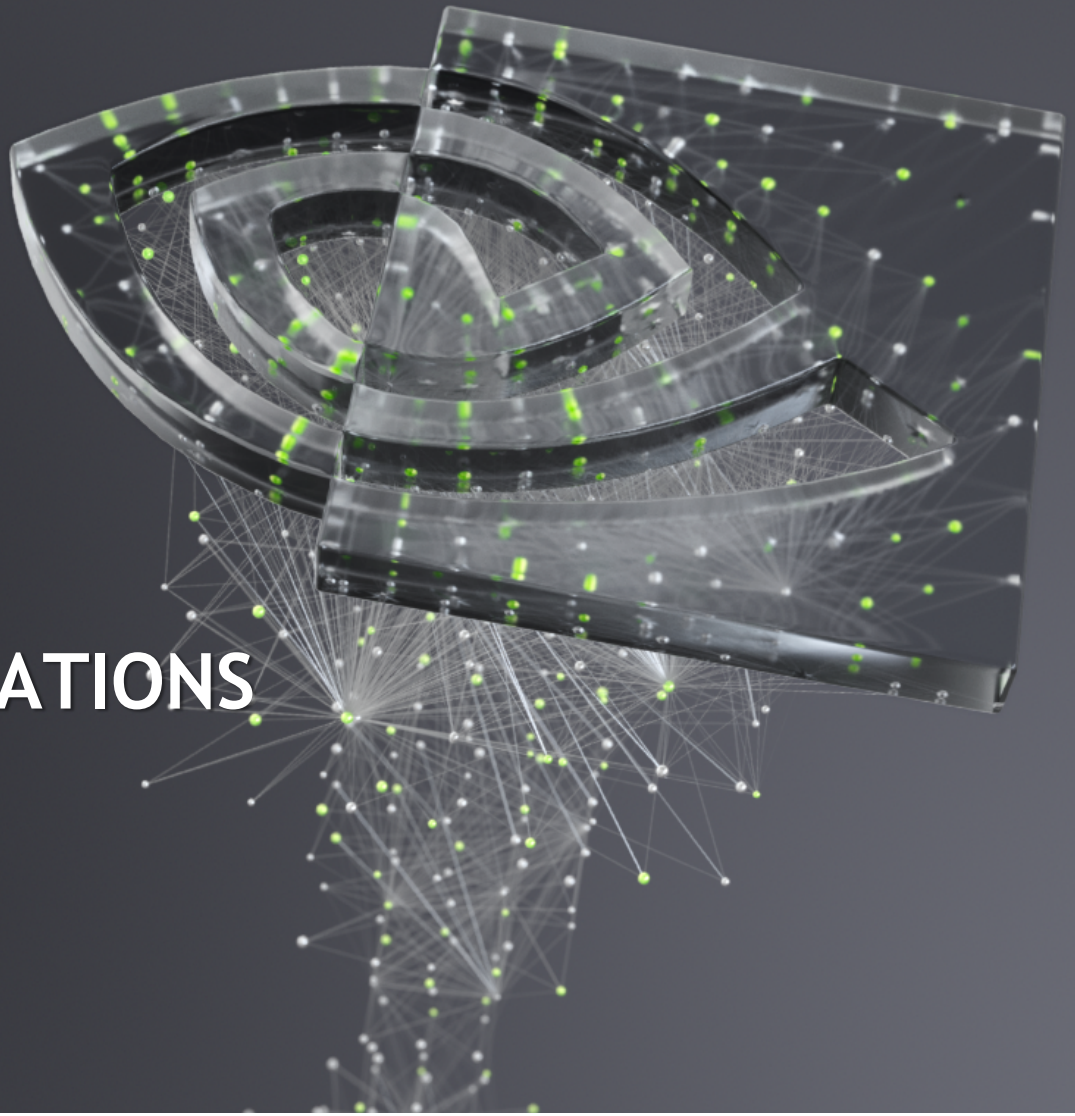
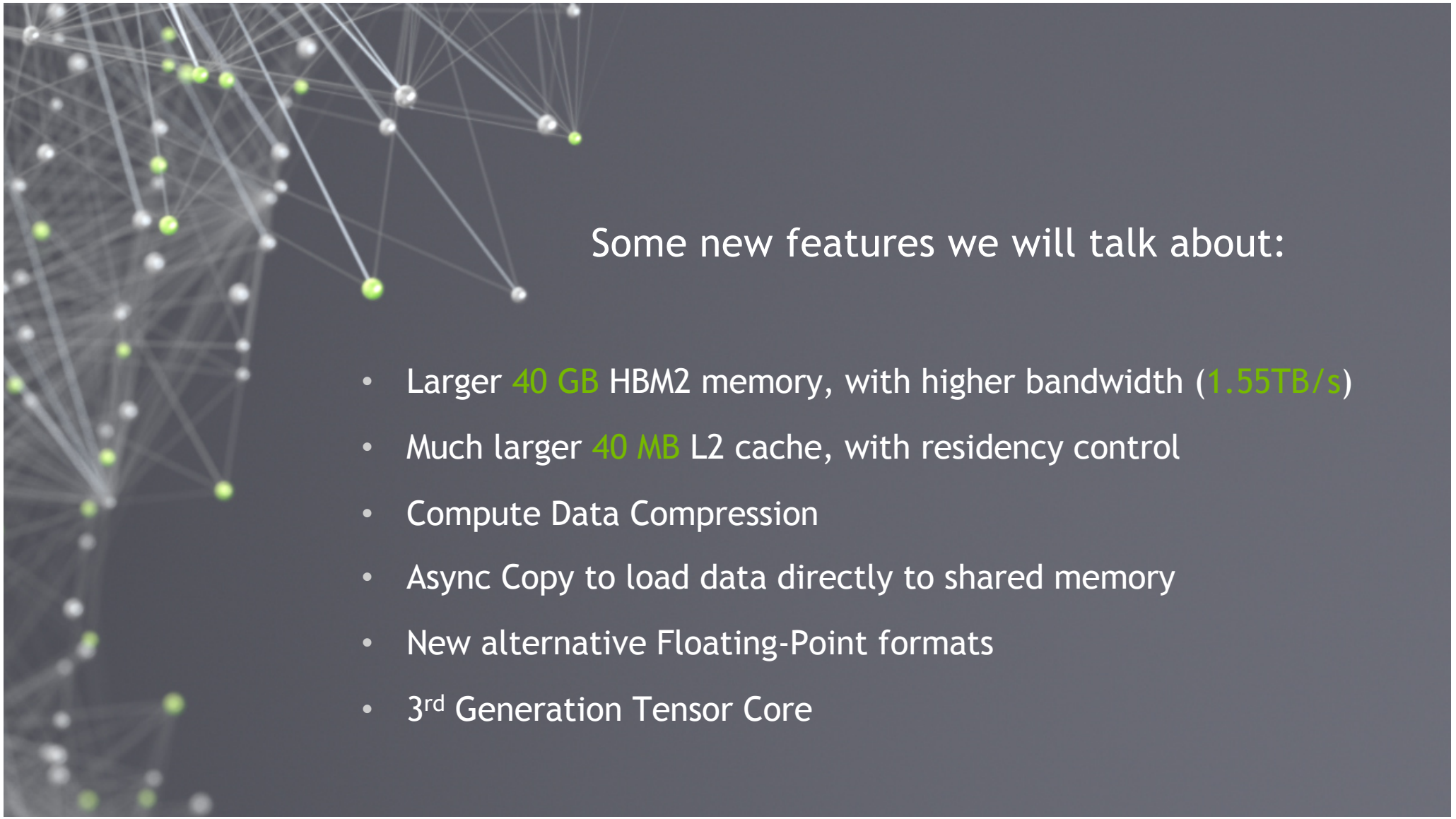


OPTIMIZING CUDA APPLICATIONS FOR NVIDIA A100 GPU

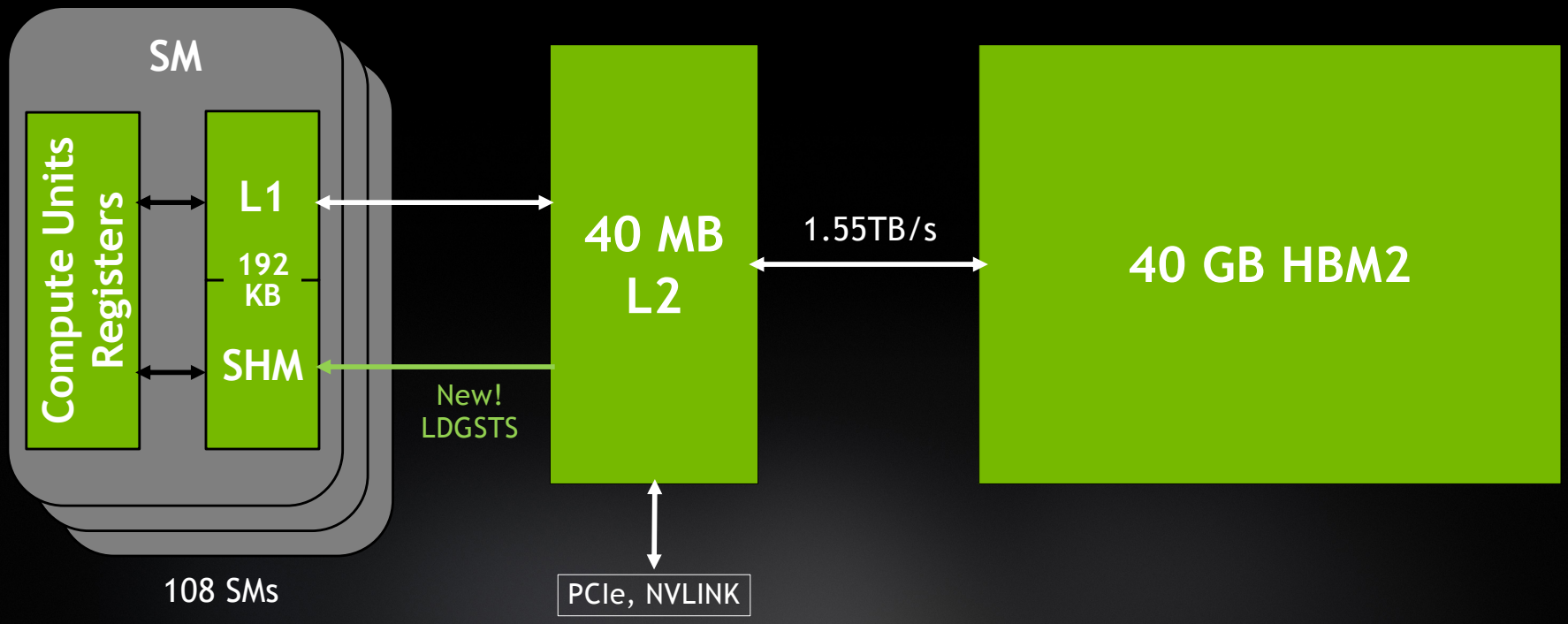
Guillaume Thomas-Collignon, Vishal Mehta

S21819 GTC 2020, 21st May 2020


Some new features we will talk about:
• Larger 40 GB HBM2 memory, with higher bandwidth (1.55TB/s)
• Much larger 40 MB L2 cache, with residency control
• Compute Data Compression
• Async Copy to load data directly to shared memory
• New alternative Floating-Point formats
• 3rd Generation Tensor Core

MEMORY HIERARCHY

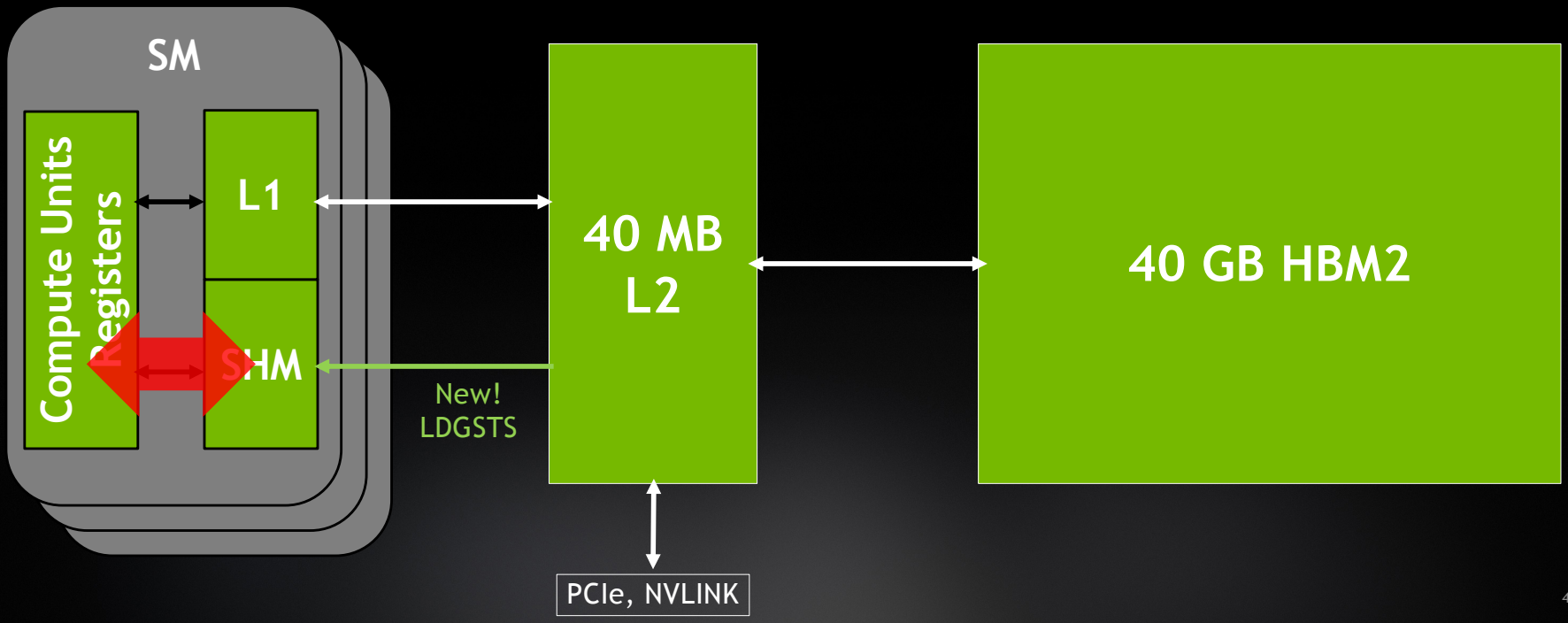
Understanding Memory and Caches



MEMORY HIERARCHY

Shared Memory

Shared memory traffic is always local to the SM

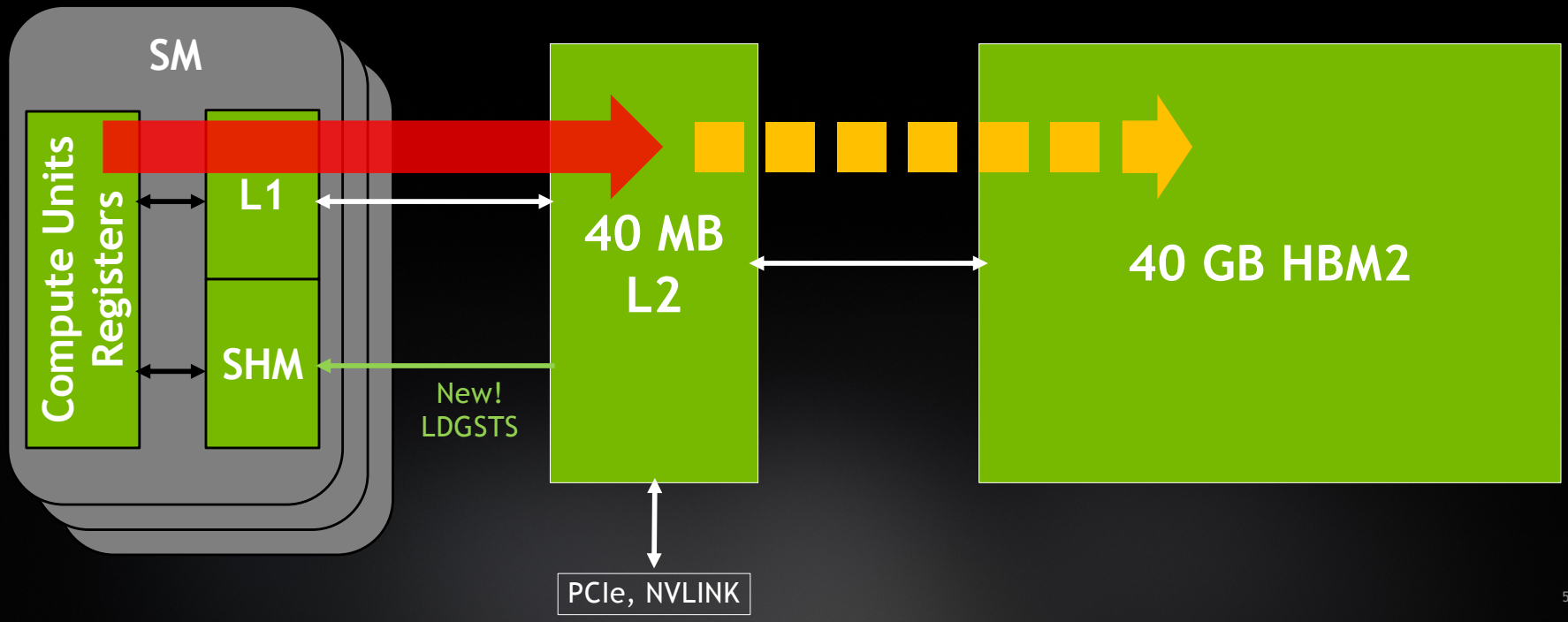


MEMORY HIERARCHY

Write Patterns

L1 is write-through, L2 is write-back

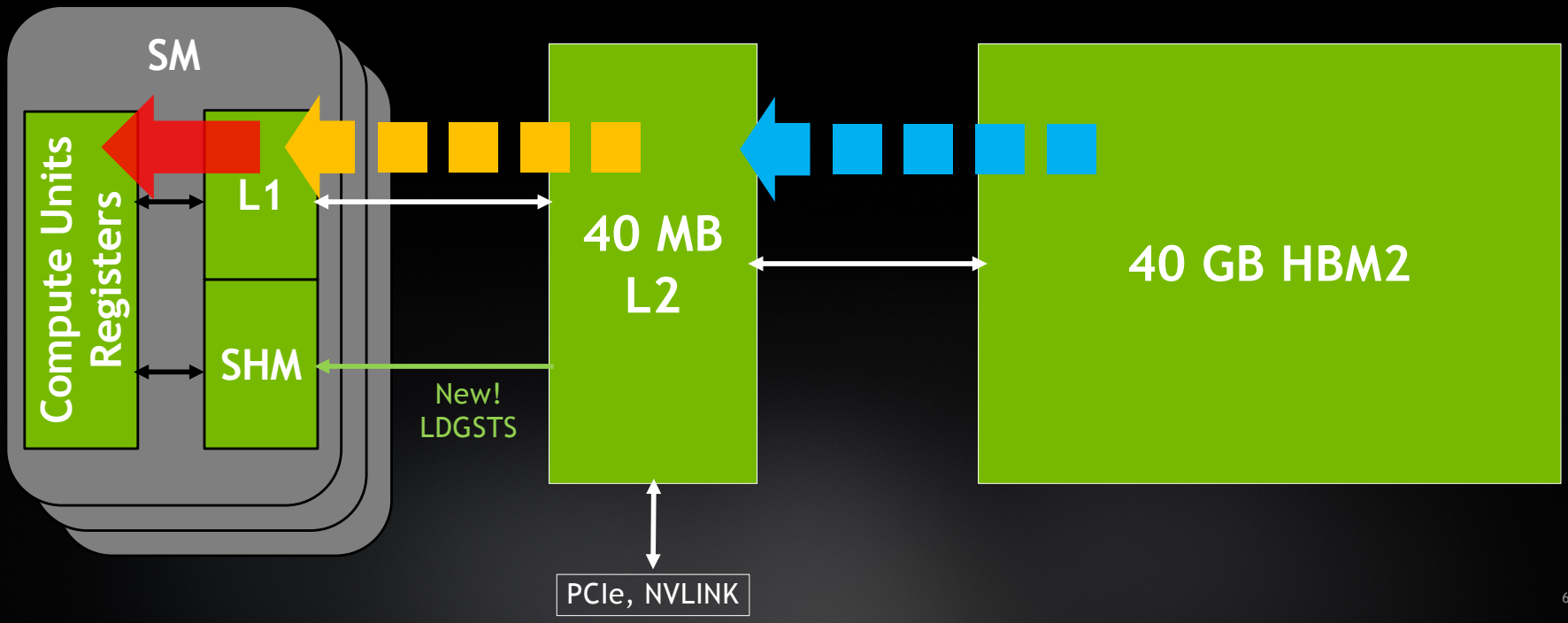
Writes will always reach at least L2, Read After Write can hit in L1 (e.g. register spills)



MEMORY HIERARCHY

Read Patterns

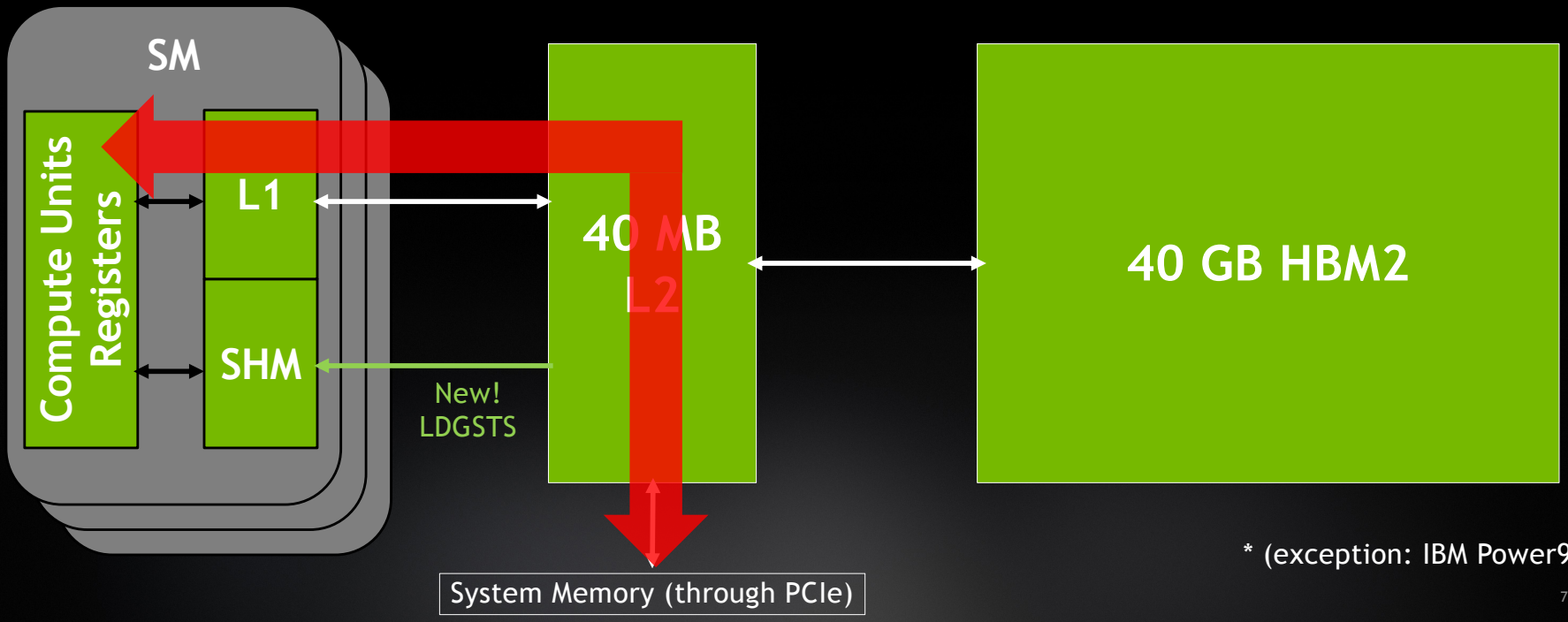
Reading from local / Global Mem can hit in L1 or L2



MEMORY HIERARCHY

System (Host) Memory

L2 does not cache System Memory*



* (exception: IBM Power9 with ATS)

MEMORY TRANSACTIONS

Cache lines and sectors

Cache line size **= 128 Bytes**

Minimum memory transaction unit = **1 sector = 32 Bytes**

For each warp: How many sectors are needed?

Since V100: default transaction size from DRAM -> L2 = 64 Bytes = 2 sectors

MEMORY TRANSACTIONS

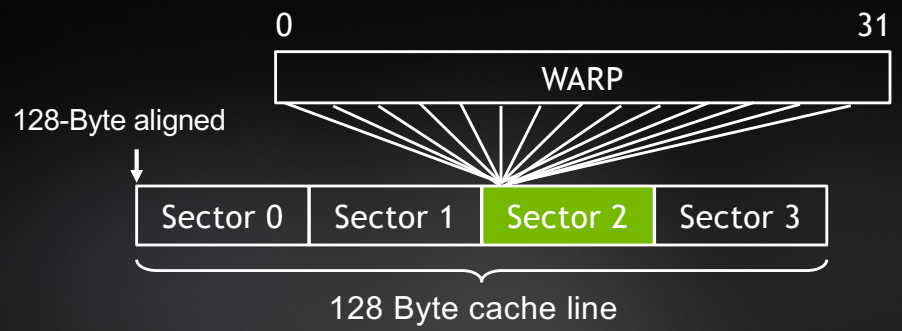
Cache lines and sectors

Cache line size **= 128 Bytes**

Minimum memory transaction unit = **1 sector = 32 Bytes**

For each warp: How many sectors are needed?

Since V100: default transaction size from DRAM -> L2 = 64 Bytes = 2 sectors



MEMORY TRANSACTIONS

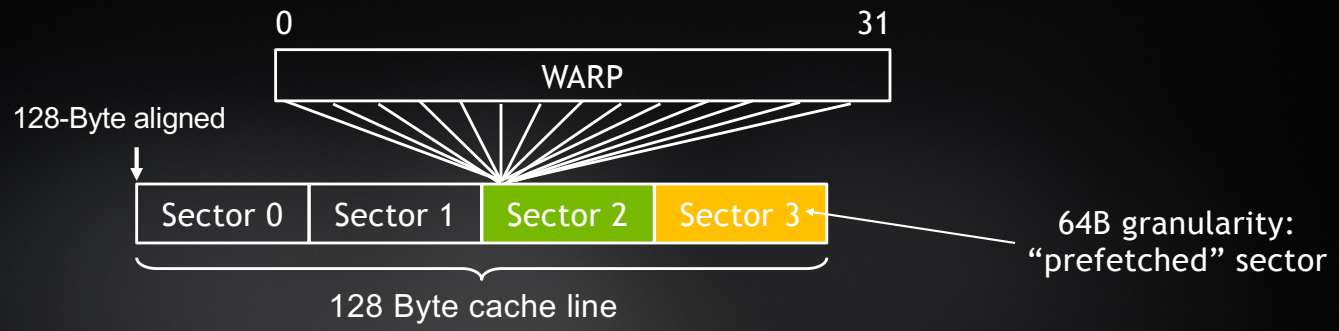
Cache lines and sectors

Cache line size **= 128 Bytes**

Minimum memory transaction unit = **1 sector = 32 Bytes**

For each warp: How many sectors are needed?

Since V100: default transaction size from DRAM -> L2 = 64 Bytes = 2 sectors



MEMORY TRANSACTIONS

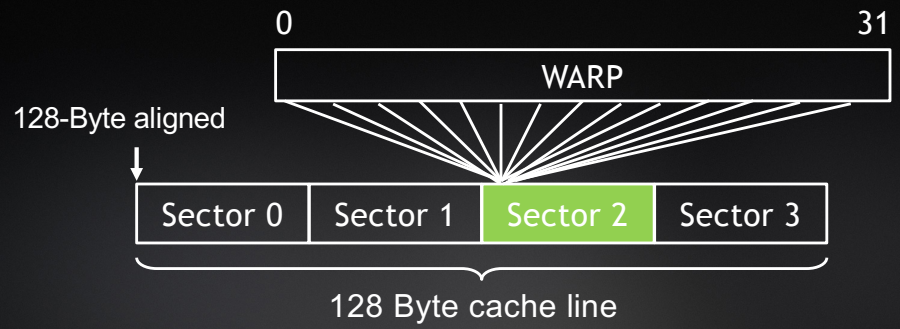
L2 granularity

On A100, the granularity can be set to 32, 64 or 128 Bytes

Random accesses might prefer smaller granularity (minimize overfetch)

Larger granularity can act as a prefetch

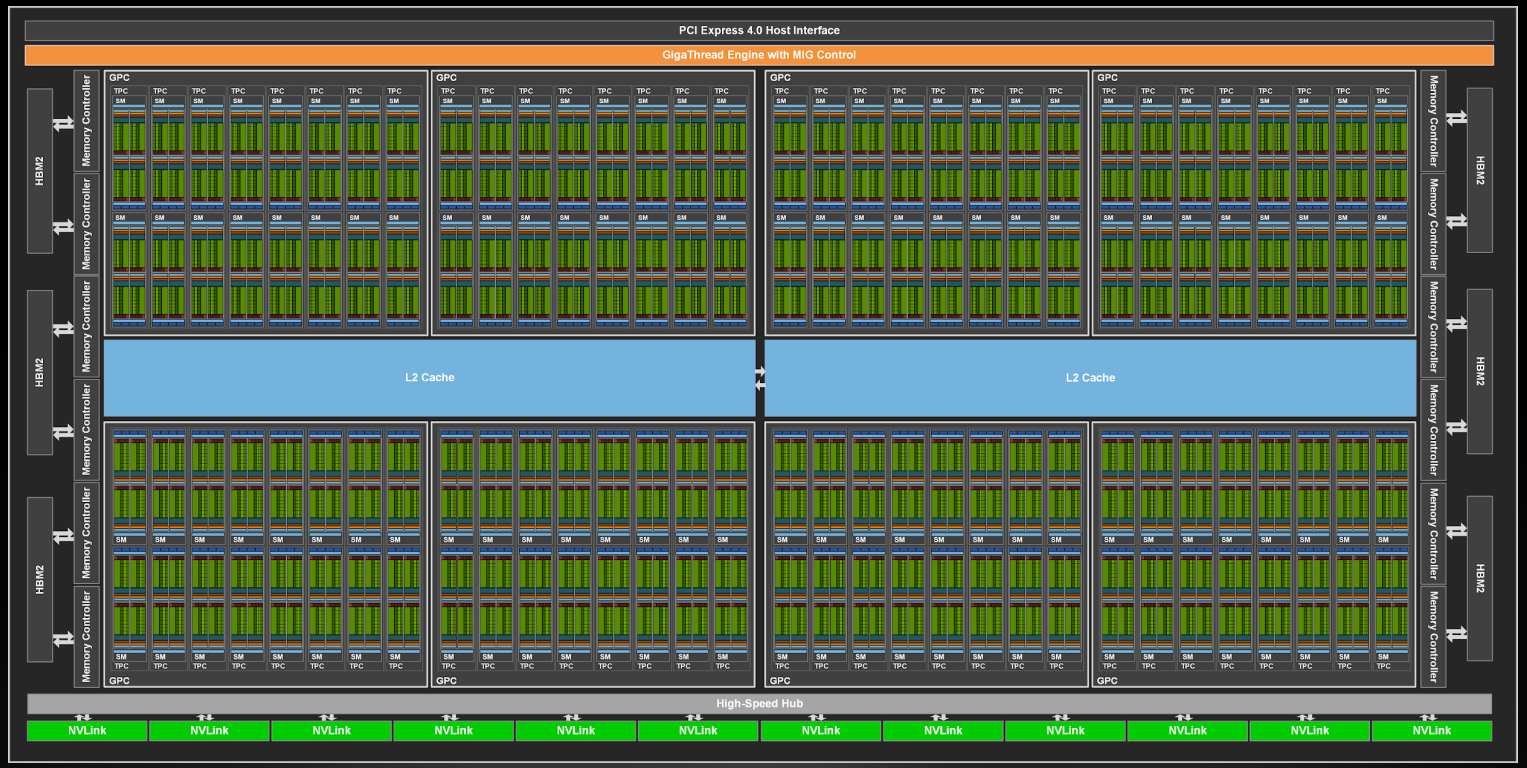
E.g. **cudaDeviceSetLimit (cudaLimitMaxL2FetchGranularity, 32)**




TUNING FOR L2 CACHE

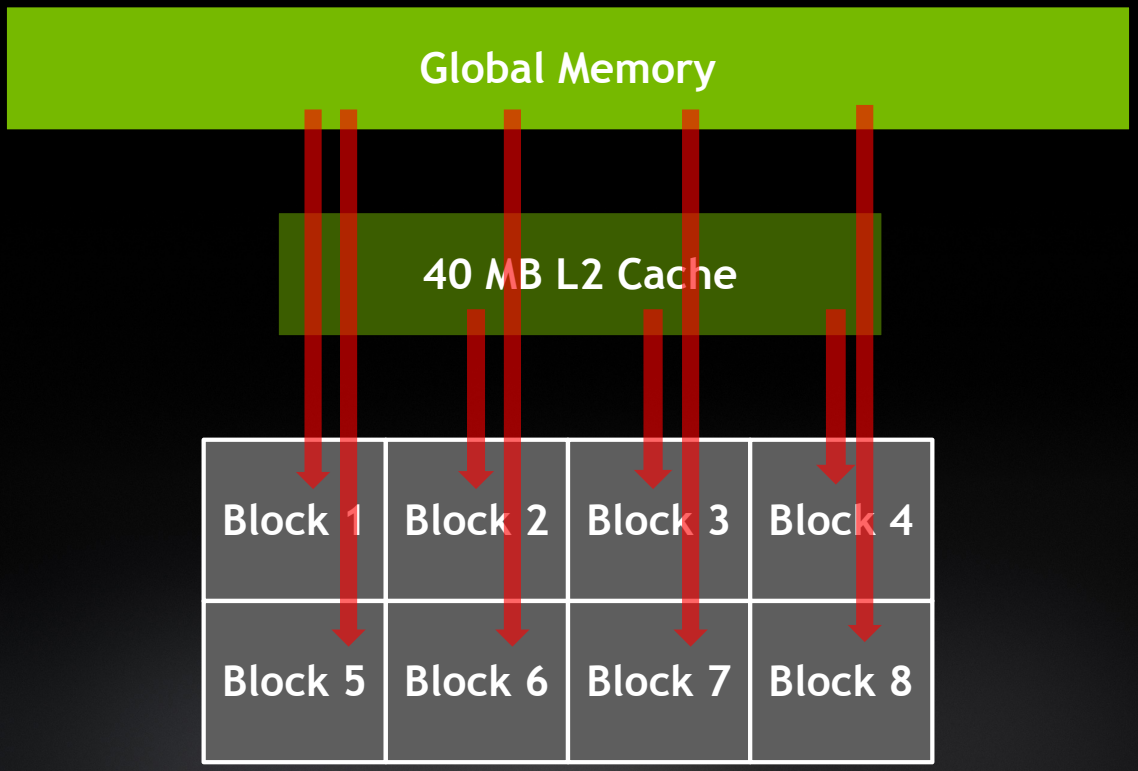
TUNING FOR L2 CACHE

Large L2 Cache with Residency Control



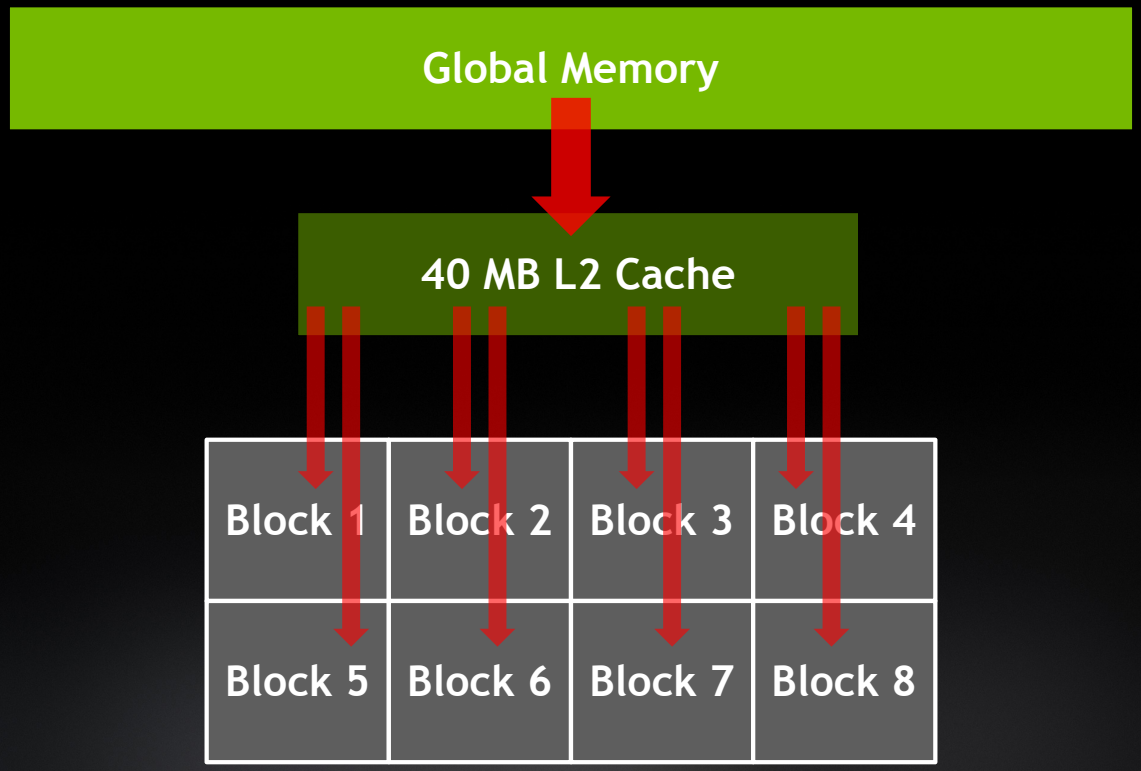
TUNING FOR L2 CACHE

L2 Cache reuse between CUDA thread blocks in a kernel



TUNING FOR L2 CACHE

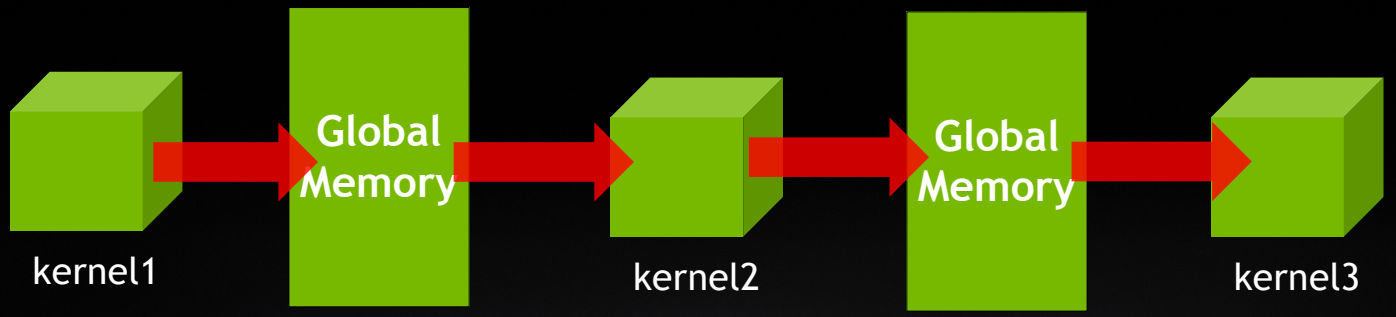
L2 Cache reuse between CUDA thread blocks in a kernel



TUNING FOR L2 CACHE

L2 Cache re use between kernel launches

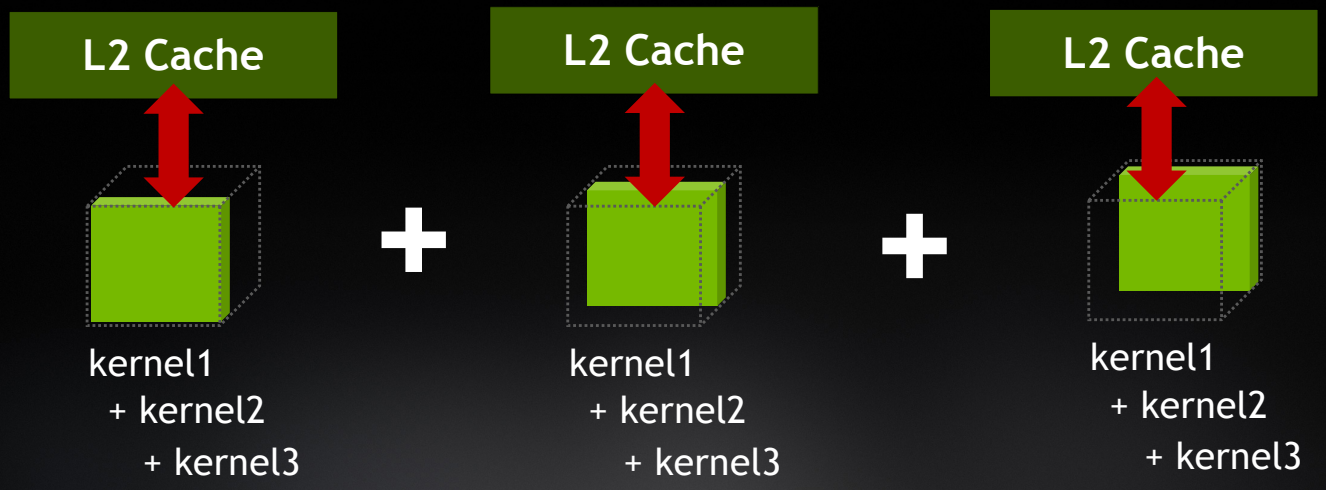
Typical case where Global memory is used as data staging buffer, between producer – consumer kernel launches



TUNING FOR L2 CACHE

L2 Cache re use between kernel lauches

The usual cache blocking techniques are now more effective on A100, especially when coupled with CUDA Graphs.

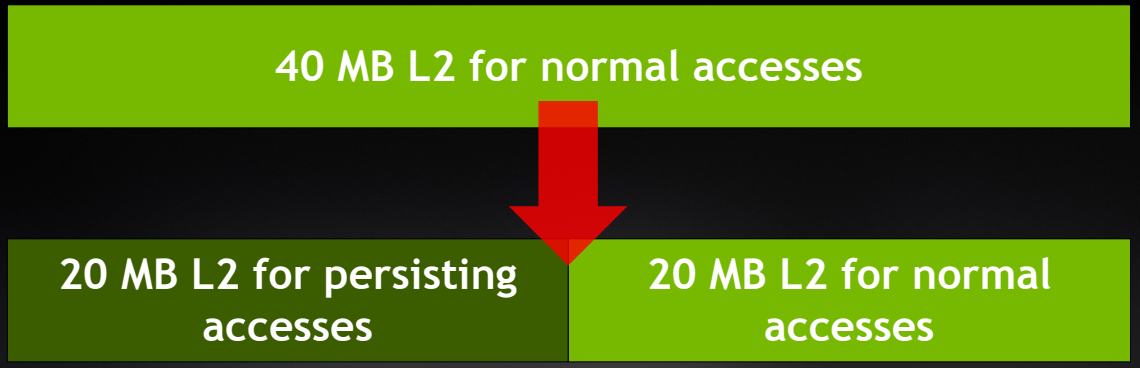


TUNING FOR L2 CACHE

L2 residency controls

- A part of L2 cache to be set-aside for persistent data accesses.
- Persistent accesses has higher residence priority in L2 cache over other data accesses.
- **Normal accesses can use the set-aside region of L2 when persisting accesses are not using it.**

```
cudaDeviceSetLimit(cudaLimitPersistingL2CacheSize, user_requested_size);
```



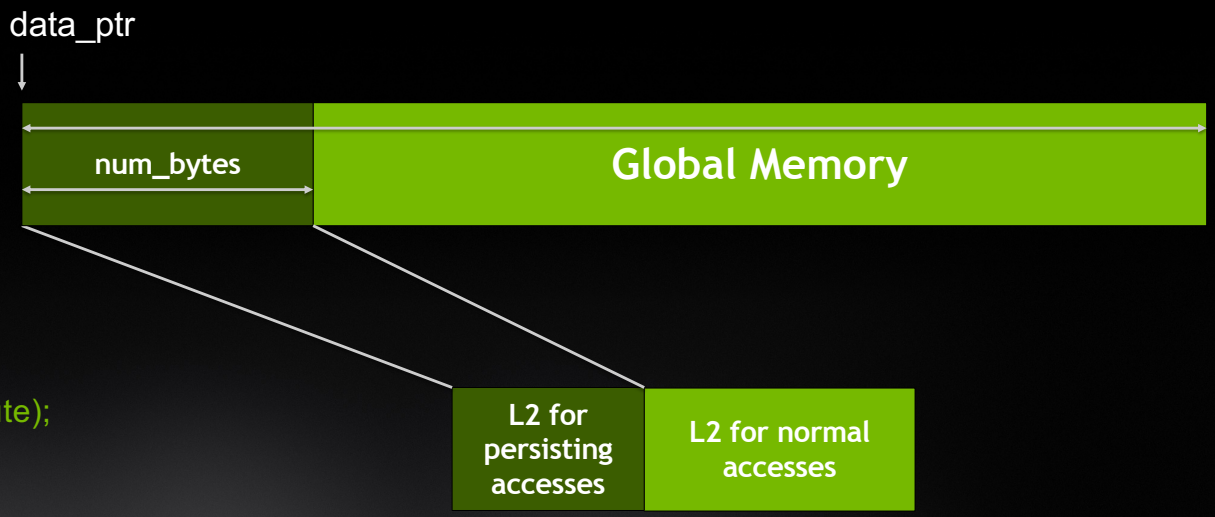
*MB = (1024 * 1024) bytes

TUNING FOR L2 CACHE

Setting Persistence on Global Memory Data Region

- Global memory region can be marked for persistence access using accessPolicyWindow
- Subsequent kernel launches in the stream or Cuda graph have persistence property on the marked data region.

```
cudaStreamAttrValue attribute;
auto &window = attribute.accessPolicyWindow;

window.base_ptr = data_ptr;
window.num_bytes = num_bytes;
window.hitRatio = 1.0;
window.hitProp = cudaAccessPropertyPersisting;
window.missProp = cudaAccessPropertyStreaming;

cudaStreamSetAttribute(stream,
cudaStreamAttributeAccessPolicyWindow, &attribute);
cuda_kernel<<<grid_size,block_size,0,stream>>>(data_ptr);
```



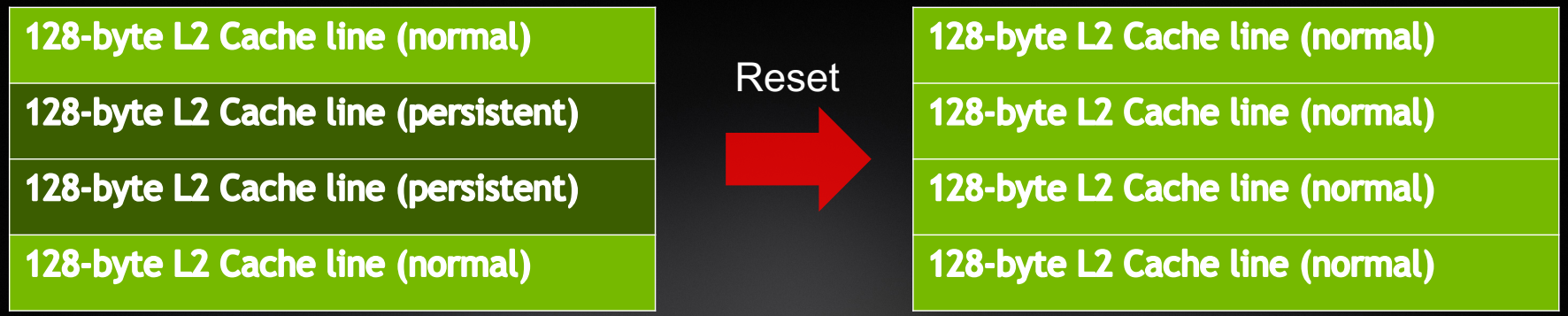
For more detailed API: S21170 (Carter Edwards)

TUNING FOR L2 CACHE

Resetting L2

- Reset does not evict but changes the persistent property of data in L2 cache to normal.
- Two reset techniques:
  1. Global reset: cudaCtxResetPersistingL2Cache()
  2. Reset using Access Window Hit property: Set cudaAccessPropertyPersisting to cudaAccessPropertyNormal

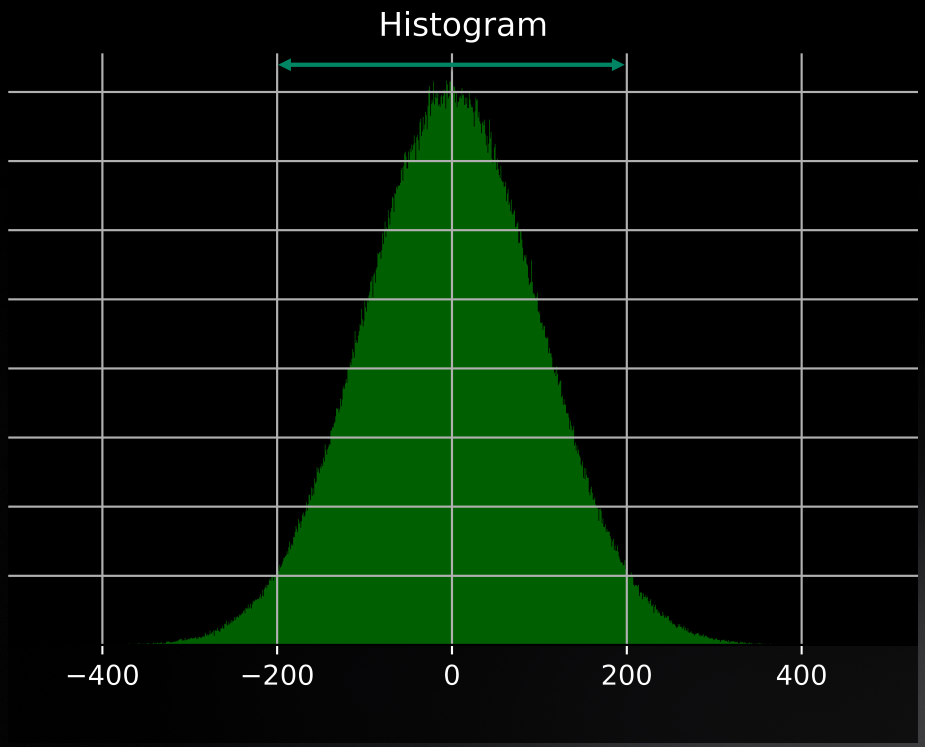
Note: If you enable L2 Persistence, don't forgot to reset it.



TUNING FOR L2 CACHE

Global Memory Histogram

- More frequently accessed histogram bins stay pinned in L2.
- Increases hit rate for global memory atomics

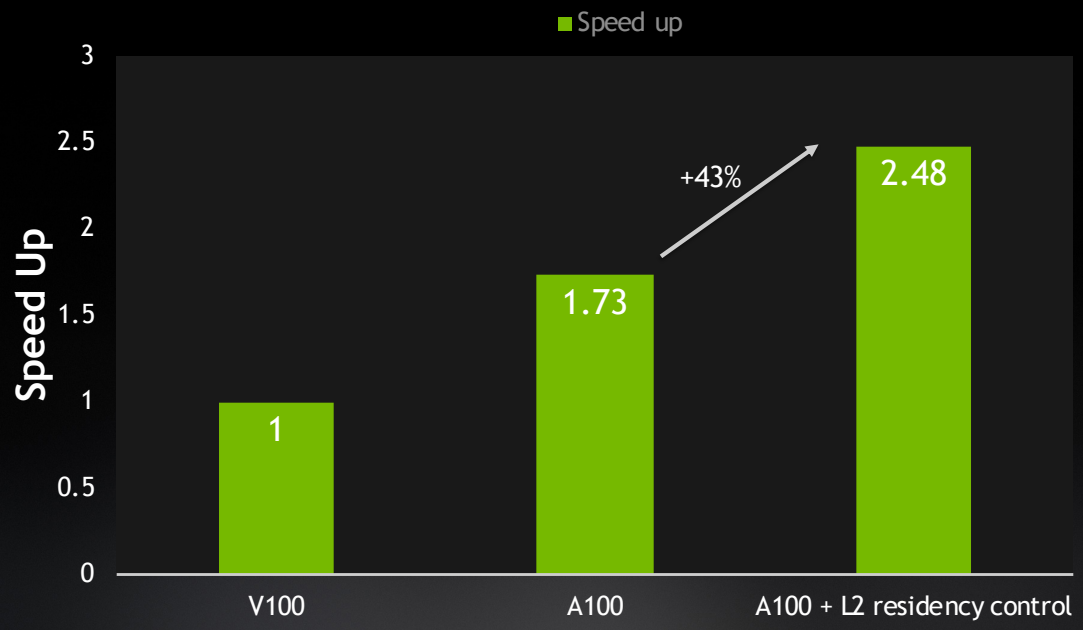



```
__global__ void histogram(int *hist, int *data, int nbins) {

    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    int bin_id = data[tid];
    // Performing atomics in global memory
    atomicAdd(hist + bin_id, 1);
}
```

TUNING FOR L2 CACHE

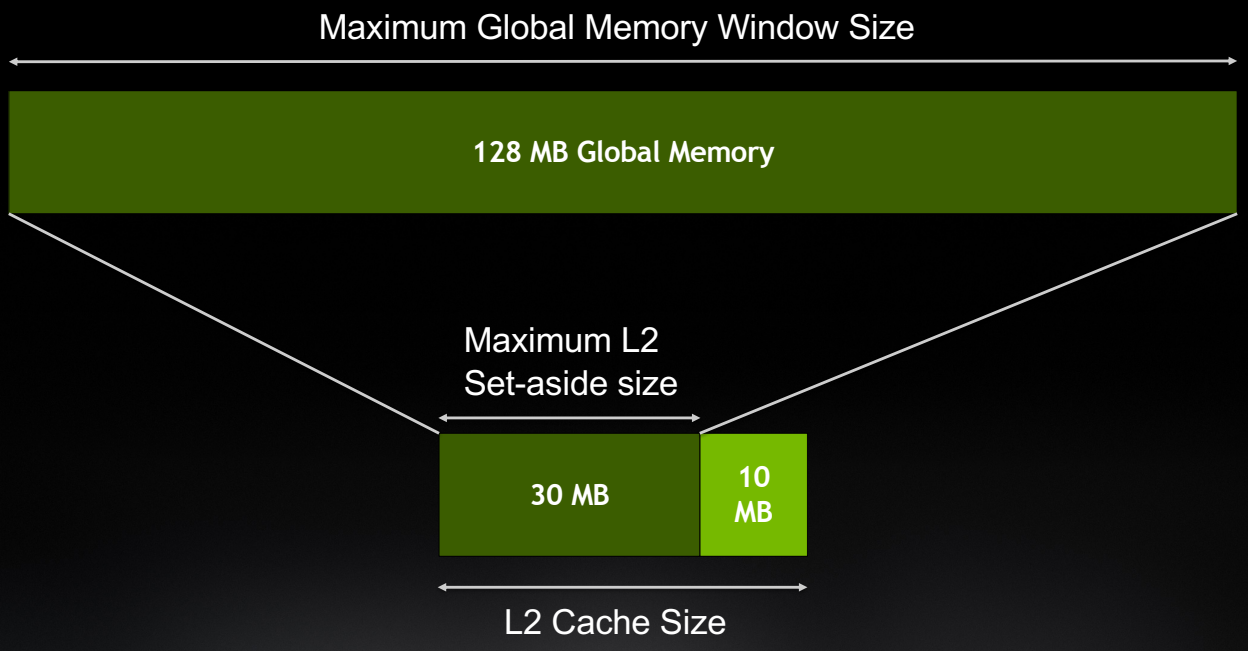
Global Memory Histogram

- Dataset Size = 1024 MB* (256 Million integers)
- Size of Persistent Histogram bins = 20 MB* (5 Million integer bins)



TUNING FOR L2 CACHE

Limits for NVIDIA A100 GPU

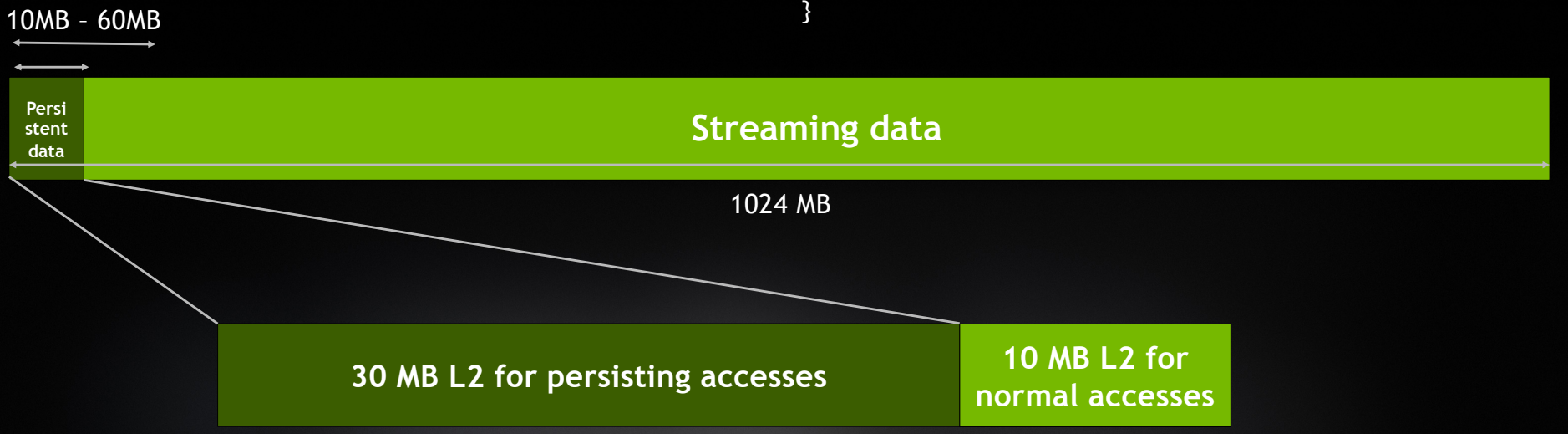


TUNING FOR L2 CACHE

Understanding Hit Ratio using Sliding window test

- Increase window size from 10MB to 60MB
- **Normal accesses can use set-aside L2, when available**
- Each thread reads and writes one element in both frequent access buffer as well as streaming buffer

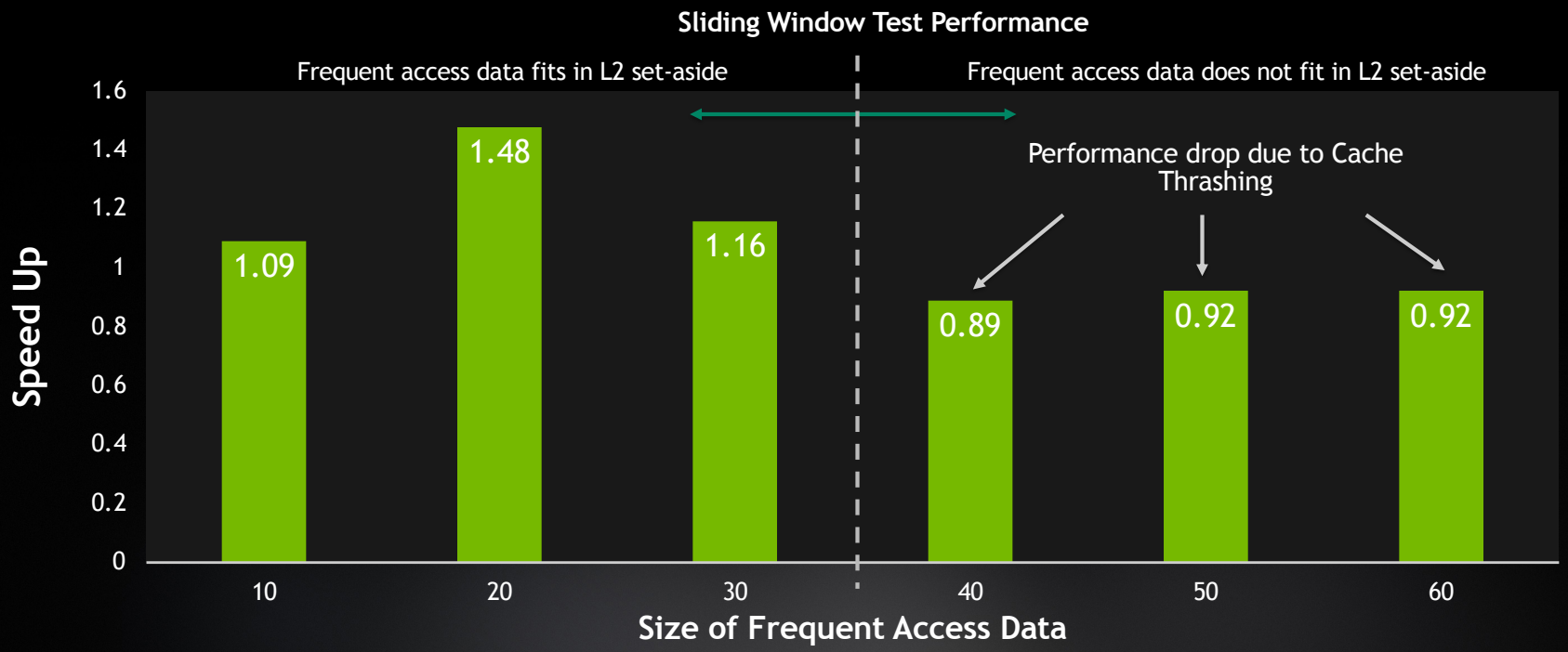
```
__global__ void kernel(int *data_persistent, int *data_streaming, int dataSize, int freqSize) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;

  data_persistent[tid % freqSize] = 2 * data_persistent[tid % freqSize];
  data_streaming[tid % dataSize] = 2 * data_streaming[tid % dataSize];
}
```



TUNING FOR L2 CACHE

Sliding window test, Fixed Hit Ratio of 1.0

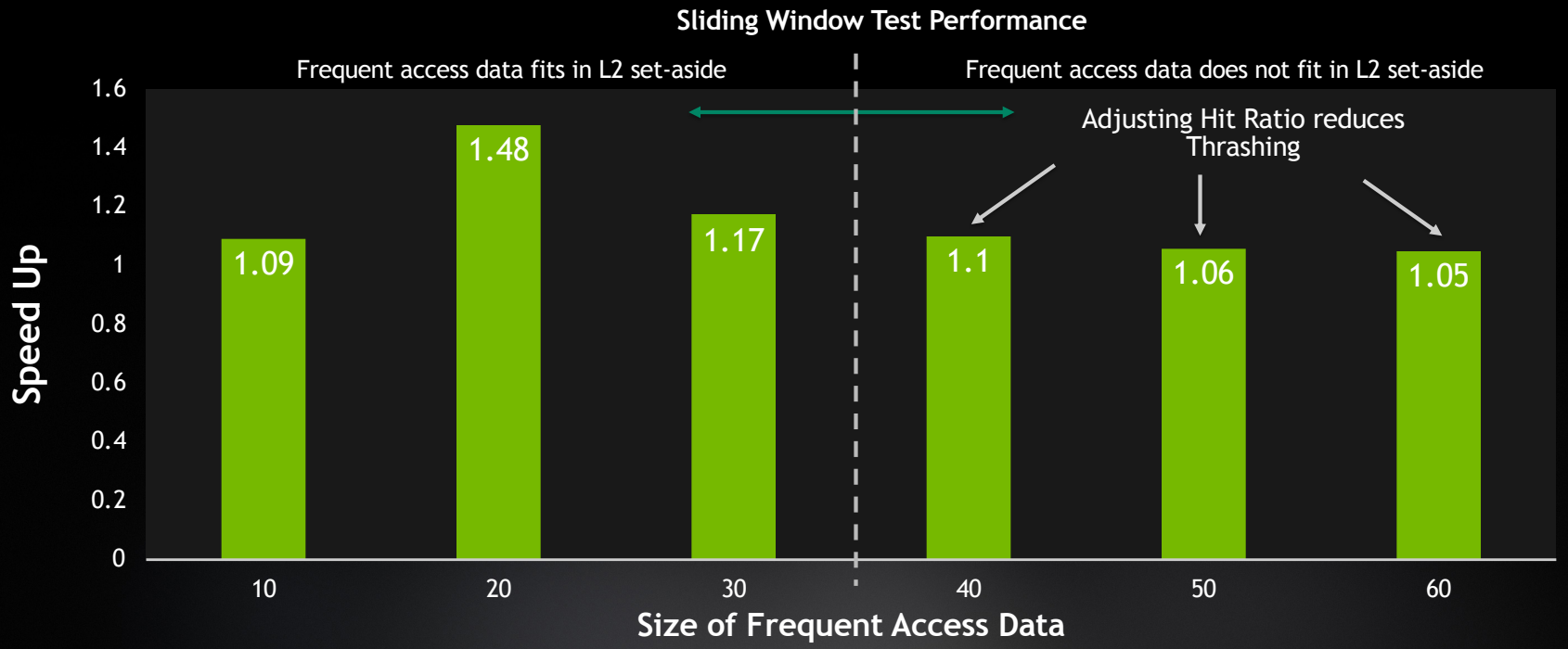
```
window.num_bytes = frequent_data_size;      // (10 - 60) MB
window.hitRatio = 1.0;                      // Always 1.0
```



TUNING FOR L2 CACHE

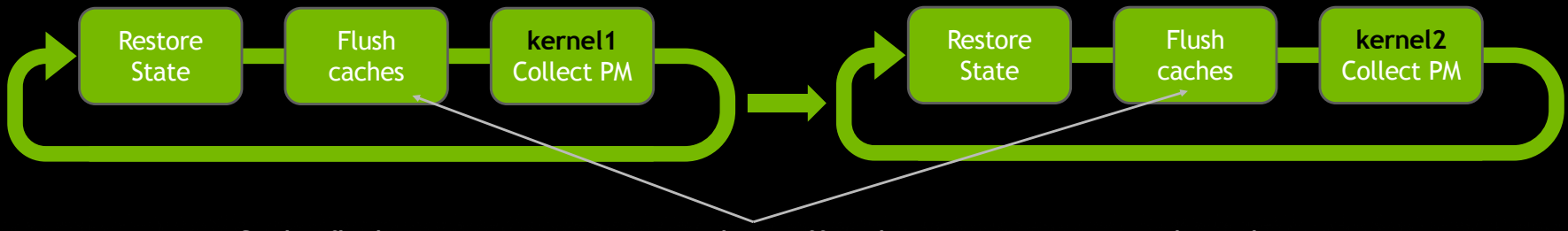
Sliding window test, Fixed Hit Ratio of 1.0

```
window.num_bytes = frequent_data_size;                          // (10-60) MB
data_in_cache = 20 * 1024 * 1024;                               // 20 MB
window.hitRatio = min ( 1.0 , data_in_cache / frequent_data_size );   // Fraction of frequent data
```



TUNING FOR L2 CACHE

Accurate profiling for L2 Cache between consecutive kernels



Cache flush prevents measuring caching effect between consecutive kernels

To measure caching between consecutive kernels:

- Turn off profiler cache control
- Run a dedicated experiment for L2 caching (no replays)

```
ncu –cache-control none –metrics lts__t_request_hit_rate.pct
```

*ncu is the command line version of Nsight Compute

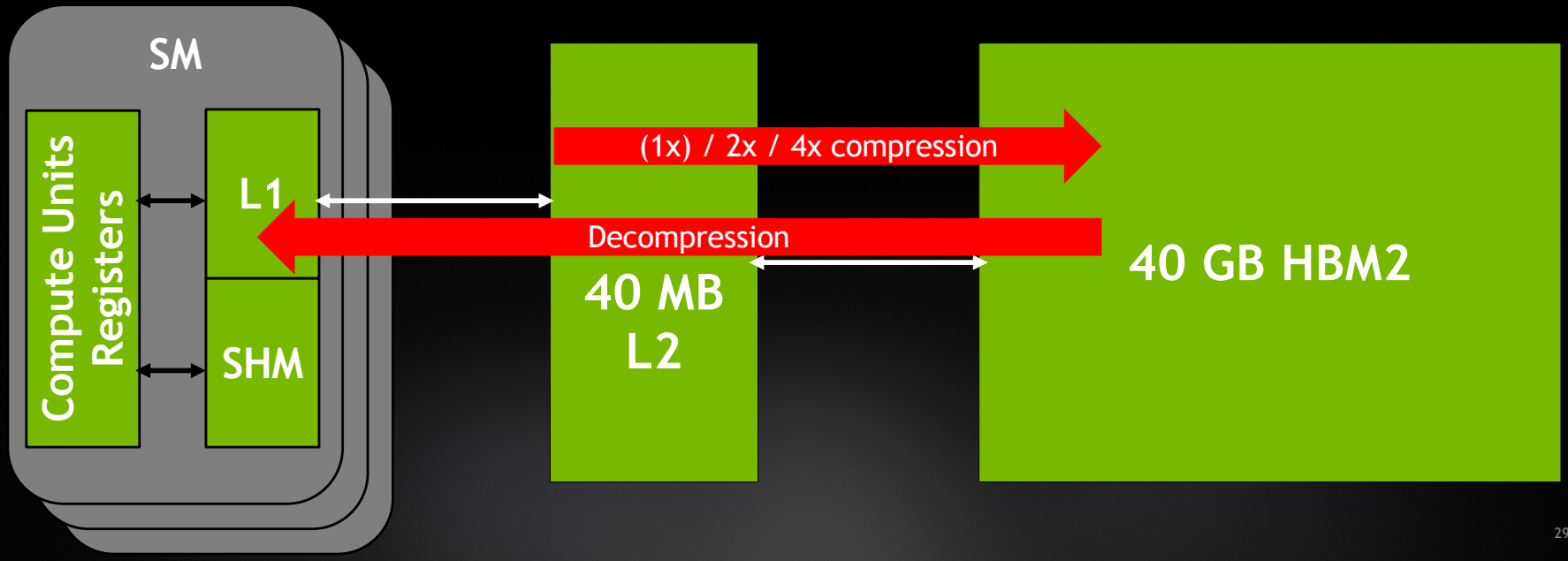

COMPUTE DATA COMPRESSION

COMPUTE DATA COMPRESSION

Hardware Memory Compression

NVIDIA A100 can compress your data in memory, with **ratios up to 4x!**

Saving bandwidth and L2 cache footprint



COMPUTE DATA COMPRESSION

How it works

- 2 consecutive cache lines (8 sectors) can be compressed 2x (4 sectors) or 4x (2 sectors)
- Data with **enough zero or similar bytes** will be compressed (lossless)
- Data must be allocated with **cuMemMap** driver API

  cuMemCreate + CU_MEM_ALLOCATION_COMP_GENERIC
- Compression does not reduce global memory footprint
- HW used for the compression is sensitive to access patterns
- Use **Nsight Compute** to check compression ratios and performance!

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

```
// Fixed number of thread blocks, loop until the end of the array
__global__ void saxpy_loop(float a, float4 *x, float4 *y, float4 *z, int64_t n)
{
  int64_t index = blockIdx.x * blockDim.x + threadIdx.x;
  for (int64_t i = index; i < n; i += blockDim.x * gridDim.x)
    z[i] = make_float4(a * x[i].x + y[i].x,
                       a * x[i].y + y[i].y,
                       a * x[i].z + y[i].z,
                       a * x[i].w + y[i].w);
}

// Each thread computes 1 element, launching as many blocks as needed
__global__ void saxpy_single(float a, float4 *x, float4 *y, float4 *z, int64_t n)
{
  int64_t i = blockIdx.x * blockDim.x + threadIdx.x;
  if (i >= n) return;
  z[i] = make_float4(a * x[i].x + y[i].x,
                     a * x[i].y + y[i].y,
                     a * x[i].z + y[i].z,
                     a * x[i].w + y[i].w);
}
```

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Running saxpy on 3 x 1.6 GB vectors, arrays initialized to 1.0, with 1024 threads / block

Visualizing the access patterns on these long vectors:

Running saxpy_loop with a number of blocks that can all reside in the GPU at the same time (1 wave)

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Running saxpy on 3 x 1.6 GB vectors, arrays initialized to 1.0, with 1024 threads / block

Visualizing the access patterns on these long vectors:

Running saxpy_loop with a number of blocks that can all reside in the GPU at the same time (1 wave)

Running saxpy_loop with more blocks than what can run at the same time (2+ waves)

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Running saxpy on 3 x 1.6 GB vectors, arrays initialized to 1.0, with 1024 threads / block

Visualizing the access patterns on these long vectors:

Running saxpy_loop with a number of blocks that can all reside in the GPU at the same time (1 wave)

Running saxpy_loop with more blocks than what can run at the same time (2+ waves)

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

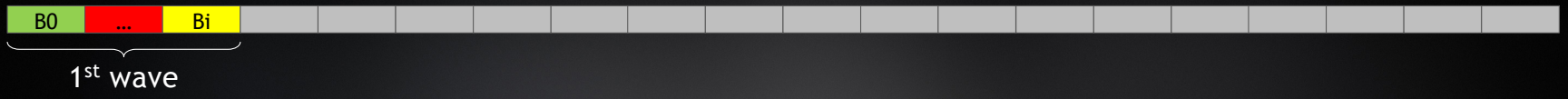
Running saxpy on 3 x 1.6 GB vectors, arrays initialized to 1.0, with 1024 threads / block

Visualizing the access patterns on these long vectors:

Running saxpy_loop with a number of blocks that can all reside in the GPU at the same time (1 wave)

Running saxpy_loop with more blocks than what can run at the same time (2+ waves)

Running saxpy_single, launching N/1024 thread blocks



COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

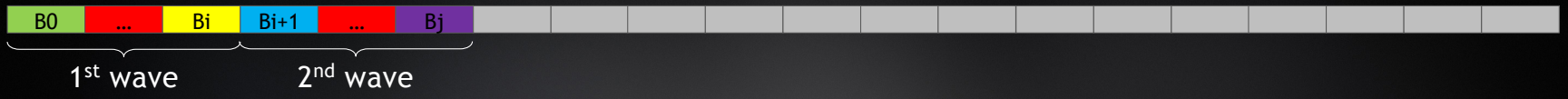
Running saxpy on 3 x 1.6 GB vectors, arrays initialized to 1.0, with 1024 threads / block

Visualizing the access patterns on these long vectors:

Running saxpy_loop with a number of blocks that can all reside in the GPU at the same time (1 wave)

Running saxpy_loop with more blocks than what can run at the same time (2+ waves)

Running saxpy_single, launching N/1024 thread blocks



COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Without compression:

	Time	Effective BW
saxpy_loop, 108 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 216 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 4000 blocks	3.6 ms	1.38 TB/s
saxpy_single, 102400 blocks	3.6 ms	1.38 TB/s

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Without compression:

	Time	Effective BW
saxpy_loop, 108 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 216 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 4000 blocks	3.6 ms	1.38 TB/s
saxpy_single, 102400 blocks	3.6 ms	1.38 TB/s

With compression turned on:

	Time	Effective BW
saxpy_loop, 108 blocks	1.96 ms	2.56 TB/s

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Without compression:

	Time	Effective BW
saxpy_loop, 108 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 216 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 4000 blocks	3.6 ms	1.38 TB/s
saxpy_single, 102400 blocks	3.6 ms	1.38 TB/s

With compression turned on:

	Time	Effective BW
saxpy_loop, 108 blocks	1.96 ms	2.56 TB/s
saxpy_loop, 216 blocks	3.13 ms	1.60 TB/s

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Without compression:

	Time	Effective BW
saxpy_loop, 108 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 216 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 4000 blocks	3.6 ms	1.38 TB/s
saxpy_single, 102400 blocks	3.6 ms	1.38 TB/s

With compression turned on:

	Time	Effective BW
saxpy_loop, 108 blocks	1.96 ms	2.56 TB/s
saxpy_loop, 216 blocks	3.13 ms	1.60 TB/s
saxpy_loop, 4000 blocks	37.6 ms	0.13 TB/s

10x slowdown!

COMPUTE DATA COMPRESSION

Access patterns : SAXPY test

Without compression:

	Time	Effective BW
saxpy_loop, 108 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 216 blocks	3.6 ms	1.38 TB/s
saxpy_loop, 4000 blocks	3.6 ms	1.38 TB/s
saxpy_single, 102400 blocks	3.6 ms	1.38 TB/s

With compression turned on:

	Time	Effective BW
saxpy_loop, 108 blocks	1.96 ms	2.56 TB/s
saxpy_loop, 216 blocks	3.13 ms	1.60 TB/s
saxpy_loop, 4000 blocks	37.6 ms	0.13 TB/s
saxpy_single, 102400 blocks	1.74 ms	2.89 TB/s

> 2x speedup

COMPUTE DATA COMPRESSION

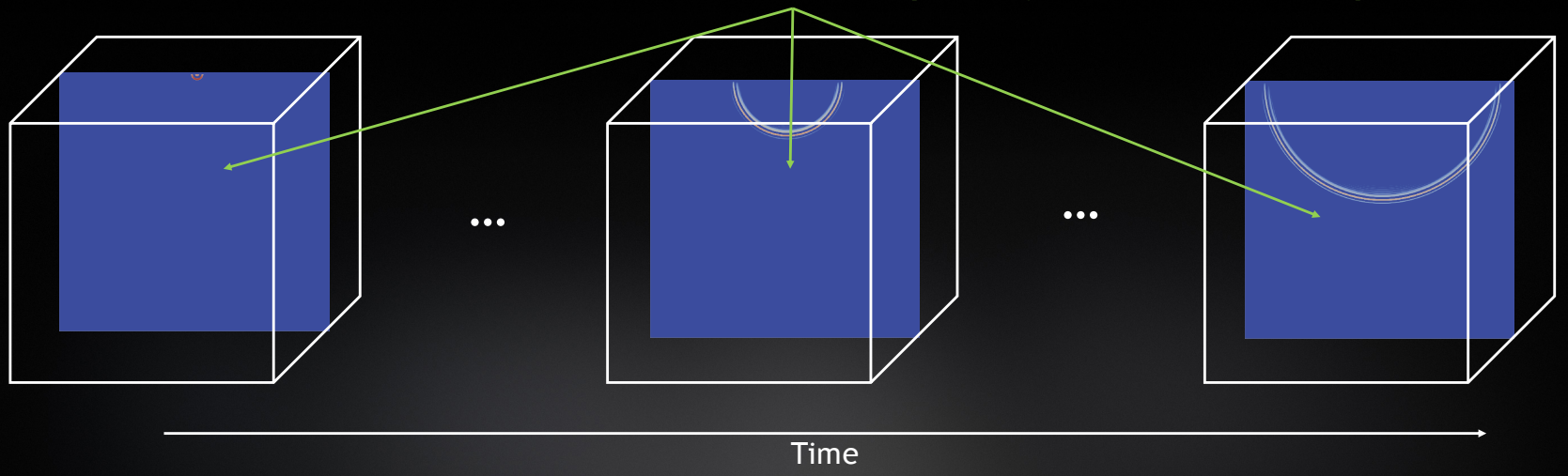
Experiment: Reverse Time Migration

Wave equation modeling in isotropic model

$$P_{t\ +\ dt}(x,y,z) = 2\ *P_t(x,y,z) - P_{t\ \_\ dt}(x,y,z) + \nabla P_t(x,y,z) * Velocity^2(x,y,z) * dt^2$$

Bandwidth-bound code

The wavefield contains lots of zeroes, especially the first time steps



COMPUTE DATA COMPRESSION

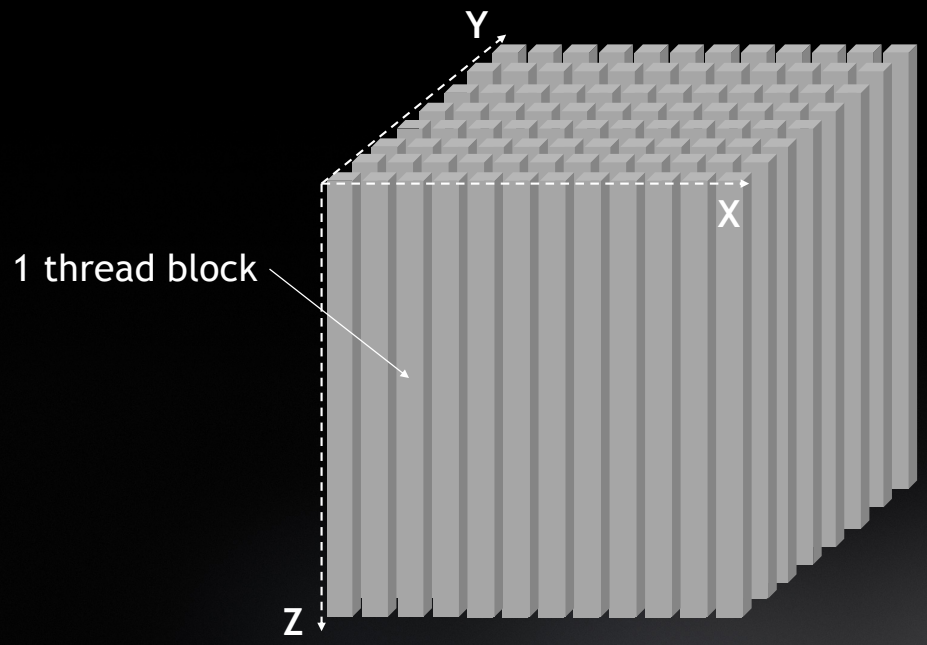
Experiment: Reverse Time Migration

Optimizing the RTM kernel to use compression

- Replaced cudaMalloc with cuMemMap+cuMemCreate (driver API)
- Trying to access more contiguous cache lines per warp
- Modified access pattern to get better locality between resident blocks in GPU

COMPUTE DATA COMPRESSION

Experiment: Reverse Time Migration



Original parallelization:

XY plan decomposed with 2D thread blocks

Using square block size (32 x 32) threads

Each thread loops on all the Z elements
(large stride between Z elements)

COMPUTE DATA COMPRESSION

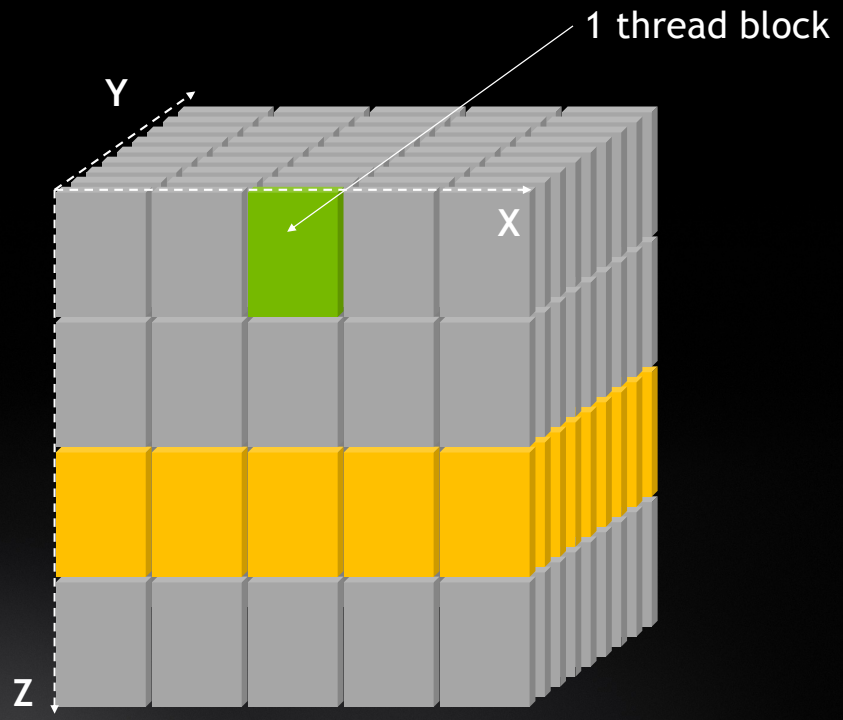
Experiment: Reverse Time Migration

Modifications

Block size (X,Y)
Changed from (32,32) to (128,8)

Adding blockIdx.z dimension
Each thread loops on fewer Z elements

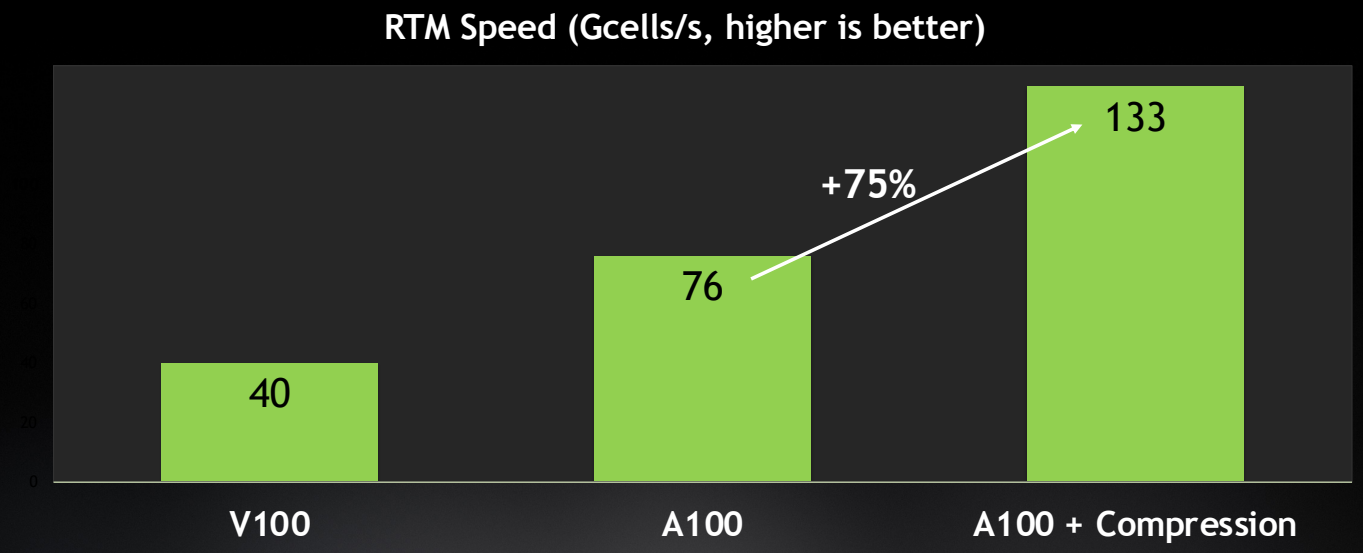
All the thread blocks with same blockIdx.z
are accessing a more localized region of memory



COMPUTE DATA COMPRESSION

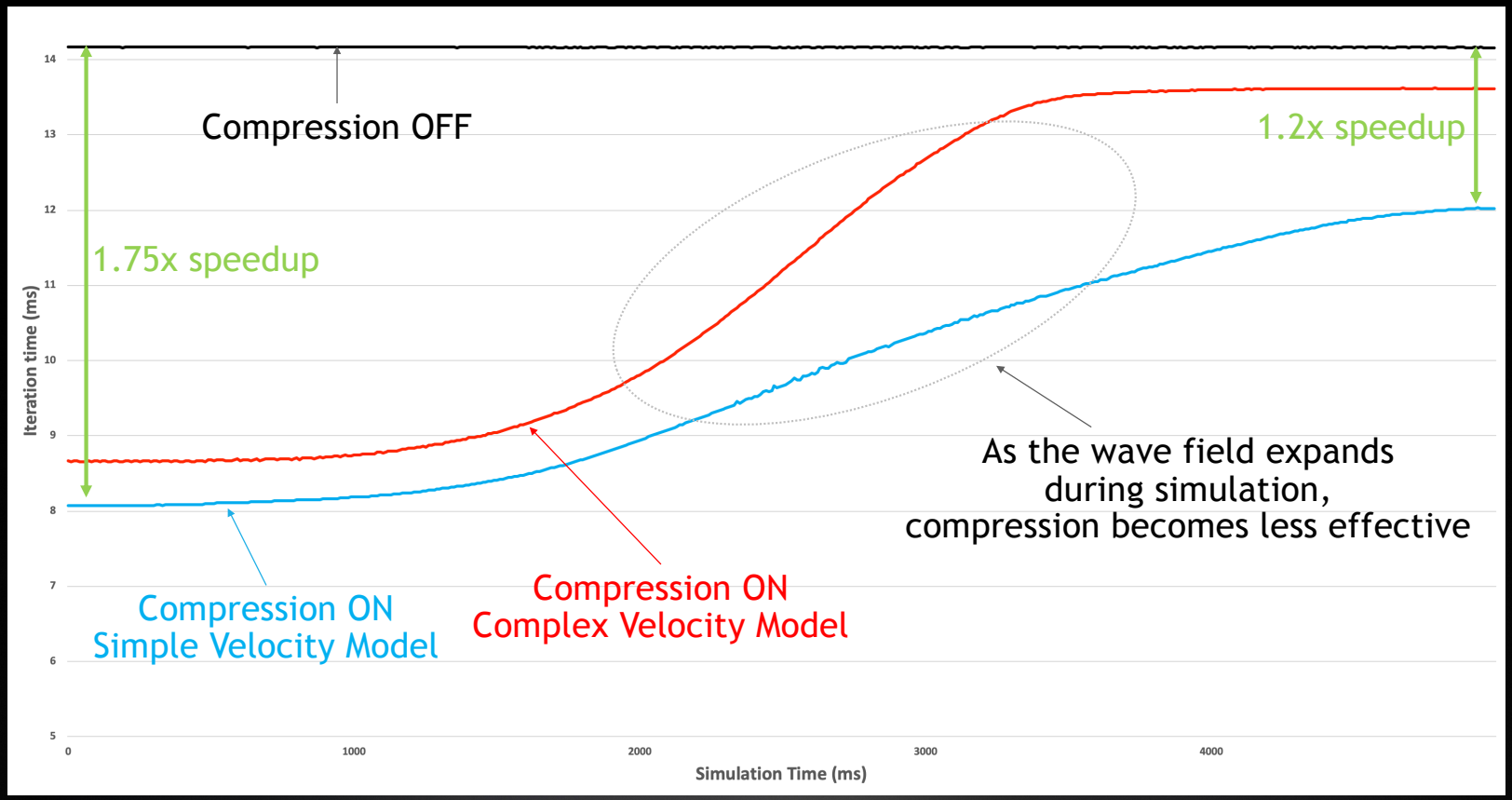
RTM Results

Comparing with best implementation without compression



COMPUTE DATA COMPRESSION

Time per iteration (lower is better) vs Simulation time, NVIDIA A100

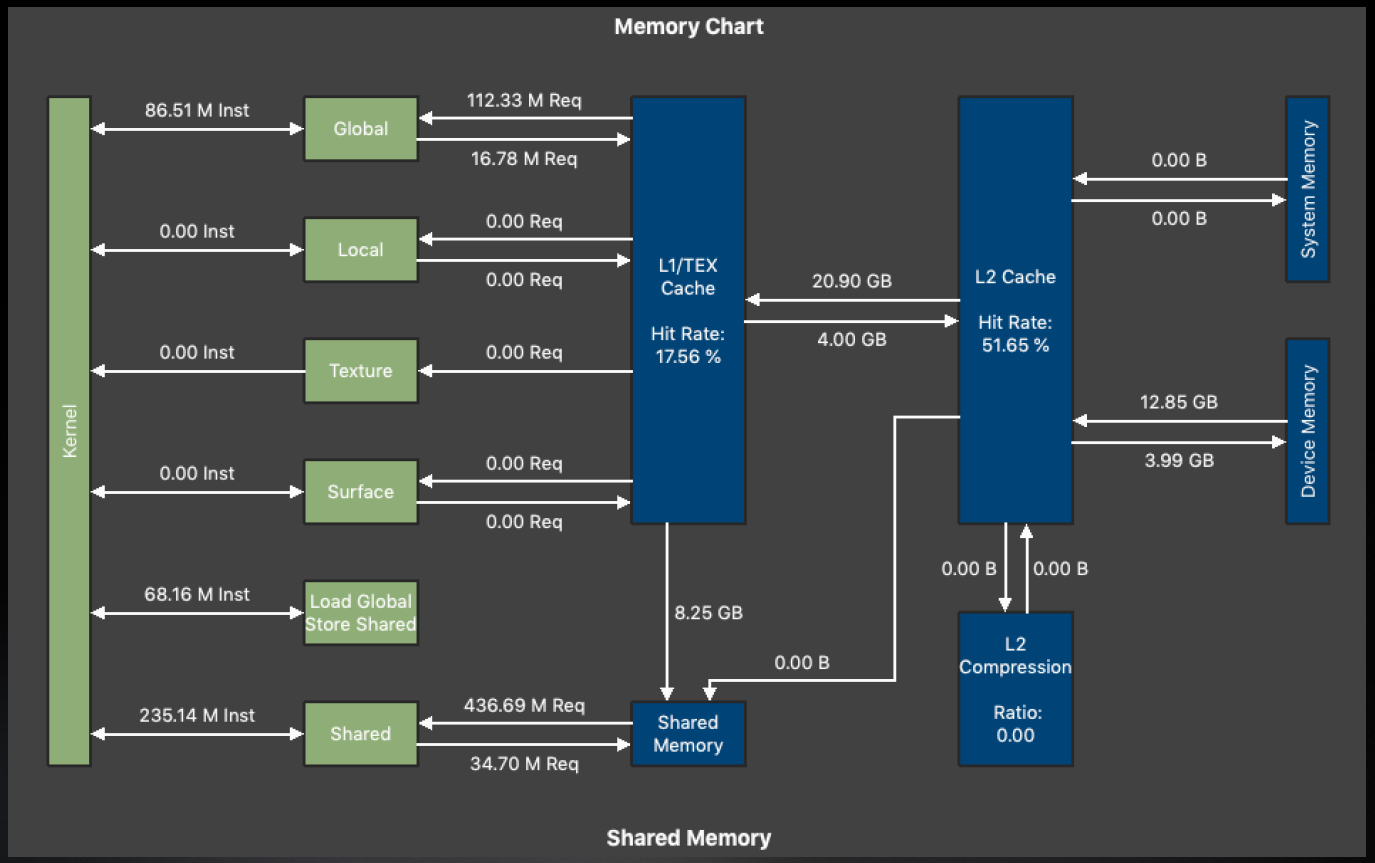


NVIDIA.

COMPUTE DATA COMPRESSION

RTM Results

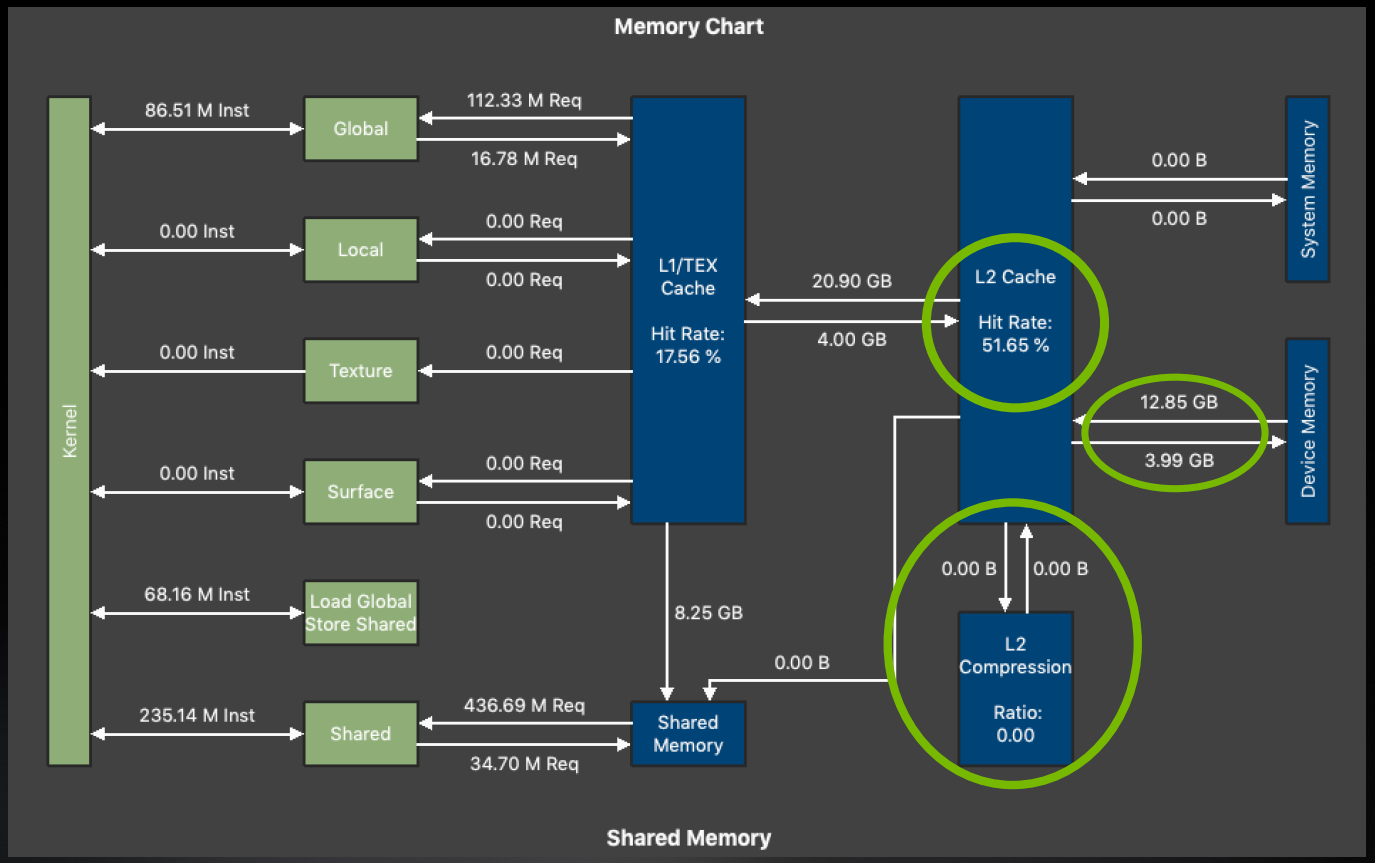
Compression disabled



COMPUTE DATA COMPRESSION

RTM Results

Compression disabled



COMPUTE DATA COMPRESSION

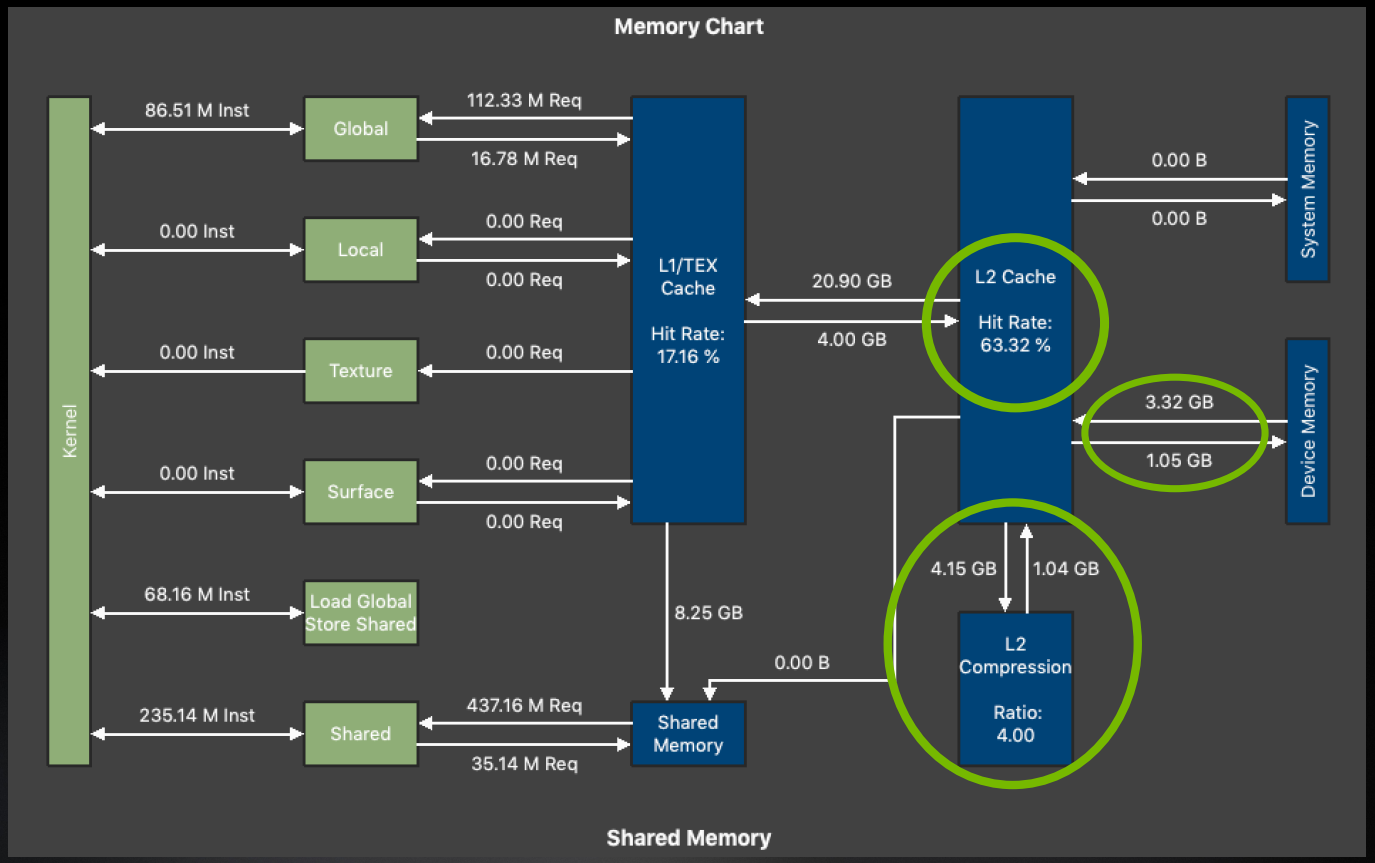
RTM Results

Compression enabled

Higher L2 hit rate

Reduced Mem BW

4x compression!



NVIDIA


ASYNC COPY

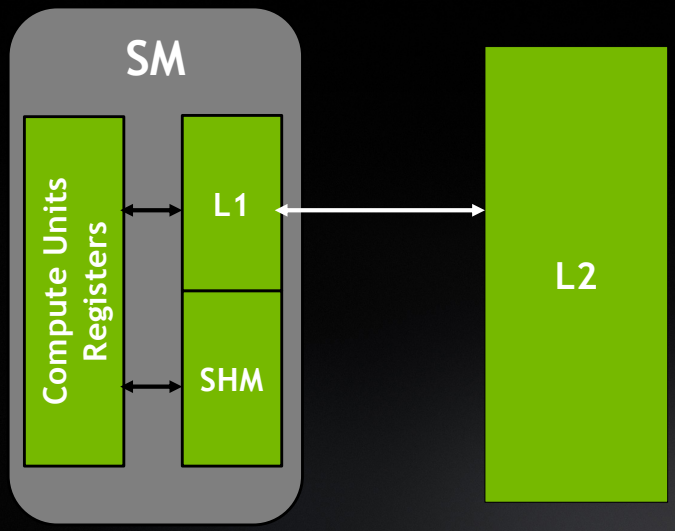
ASYNC COPY

Asynchronous load + store in shared Memory

Typical way of using shared memory:

```
__shared__ int smem[1024];

smem[threadIdx.x] = input[index];
```

```
LDG.E.SYS R0, [R2] ;
    * STALL *
STS [R5], R0 ;
```



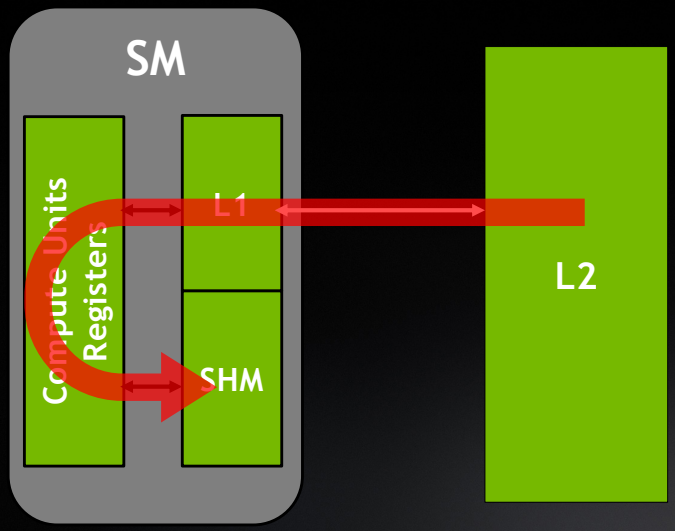
ASYNC COPY

Asynchronous load + store in shared Memory

Typical way of using shared memory:

```
__shared__ int smem[1024];

smem[threadIdx.x] = input[index];
```

```
LDG.E.SYS R0, [R2] ;
    * STALL *
STS [R5], R0 ;
```

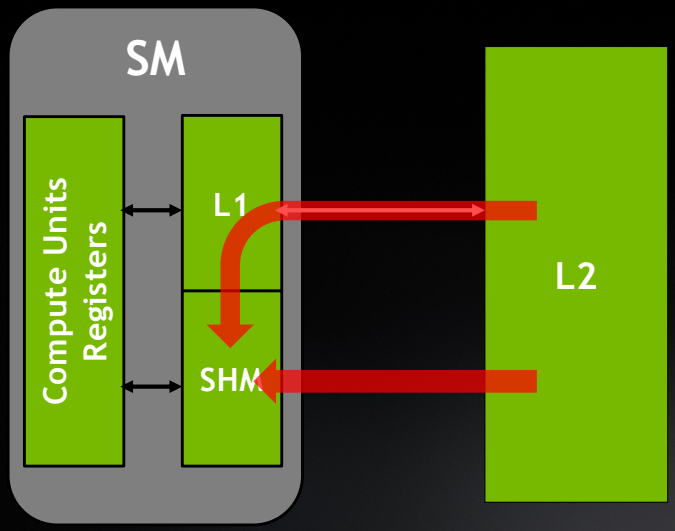


- Wasting registers
- Stalling while the data is loaded
- Wasting L1/SHM bandwidth

ASYNC COPY

Asynchronous load + store in shared Memory

```
__shared__ int smem[1024];
__pipeline_memcpy_async(&smem[threadIdx.x], &input[index], sizeof(int));
__pipeline_commit();
__pipeline_wait_prior(0);
```



Copies the data straight to shared memory asynchronously with 2 possible paths:

- L1 Access (Data gets Cached in L1)
- L1 Bypass (No L1 Caching, 16-Byte vector LDGSTS)

Very flexible scheduling (e.g. multi-stage)

For more details: S21170 (Carter Edwards)

ASYNC COPY

Using Async Copy in TTI Reverse Time Migration

TTI Radius 8 Reverse Time Migration (1-pass)

- Close to compute bound
- Couldn’t quite reach Speed Of Light
- High register pressure
- Low occupancy (1 block of 384 threads per SM)

```
Loop through Z dimension
  __syncthreads()
  Load data (+neighbor) into SHM
  __syncthreads()
  Compute Y and YY derivatives
  Compute Z derivatives
  Share Y and Z derivatives (SHM)
  __syncthreads()
  ... A lot more computation
  Write results
End loop
```

ASYNC COPY

Using Async Copy in TTI Reverse Time Migration

Using the data which was just loaded

Expensive load + sync (long wait, no other block in the SM)

Can’t easily prefetch the data for the next iteration

(even more registers)

```
Loop through Z dimension
  __syncthreads()
  Load data (+neighbor) into SHM
  __syncthreads()
  Compute Y and YY derivatives
  Compute Z derivatives
  Share Y and Z derivatives (SHM)
  __syncthreads()
  ... A lot more computation
  Write results
End loop
```

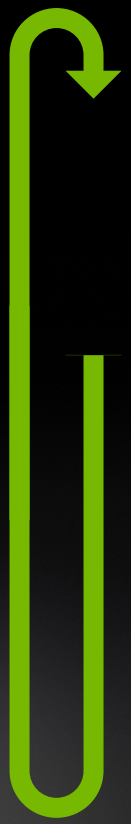
ASYNC COPY

Using Async Copy in TTI Reverse Time Migration

Using a single stage Async Copy pipeline

Just prefetching next iteration's data

Not using the L1 bypass



```
Loop through Z dimension
  Wait for Async Copy
  __syncthreads()
  Compute Y and YY derivatives
  __syncthreads()
  AsyncCopy Load data for next iter
  Compute Z derivatives
  Share Y and Z derivatives (SHM)
  __syncthreads()
  ... A lot more computation
  Write results
End loop
```

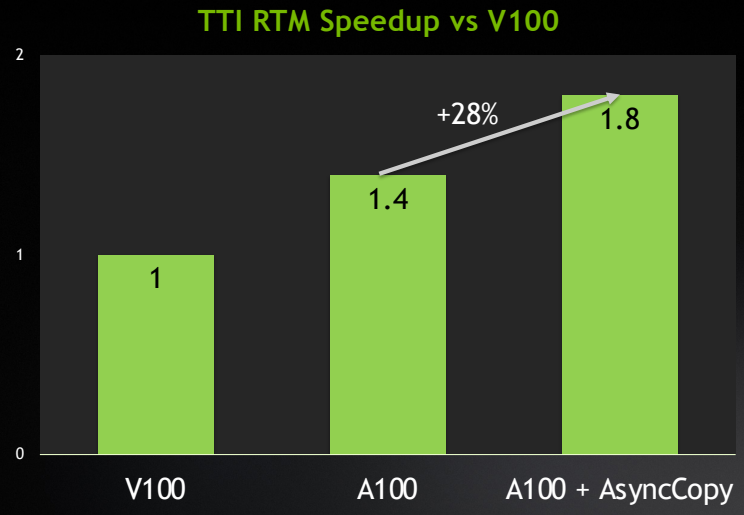
ASYNC COPY

Using Async Copy in TTI Reverse Time Migration

Using a single stage Async Copy pipeline

Just prefetching next iteration's data

Not using the L1 bypass

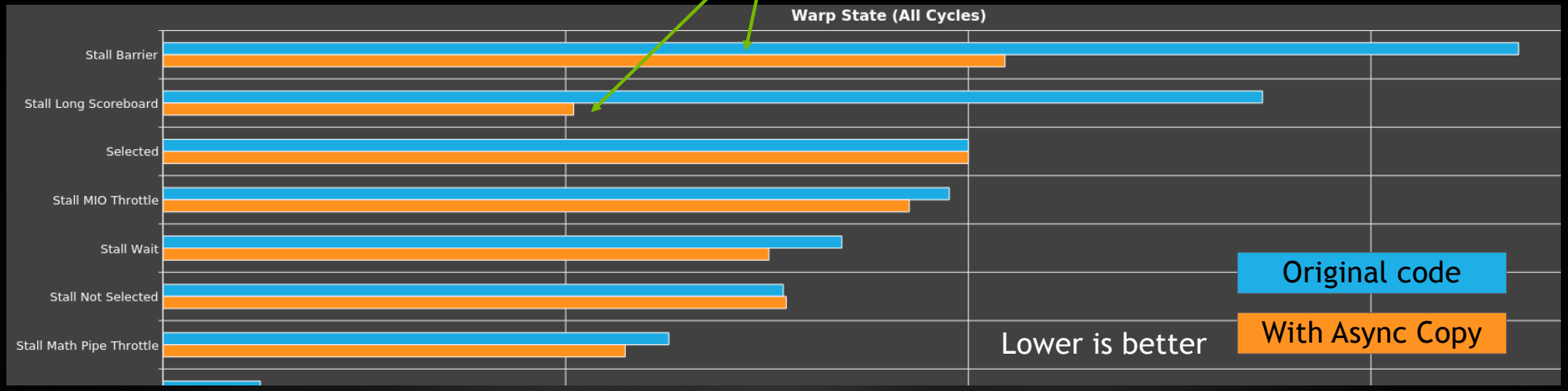


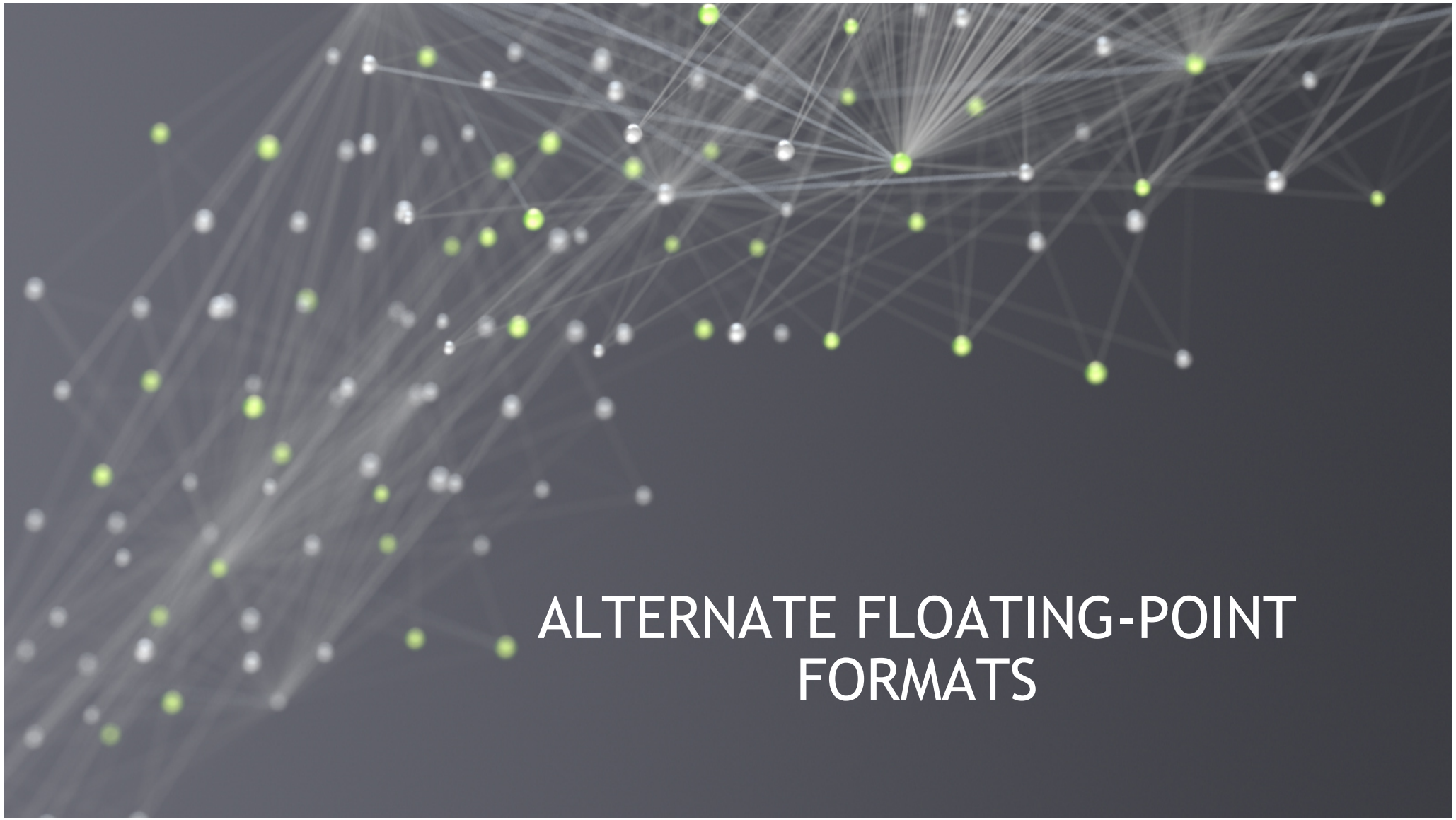

```
Loop through Z dimension
  Wait for Async Copy
  __syncthreads()
  Compute Y and YY derivatives
  __syncthreads()
  AsyncCopy Load data for next iter
  Compute Z derivatives
  Share Y and Z derivatives (SHM)
  __syncthreads()
  ... A lot more computation
  Write results
End loop
```

ASYNC COPY

TTI RTM: What Nsight Compute says

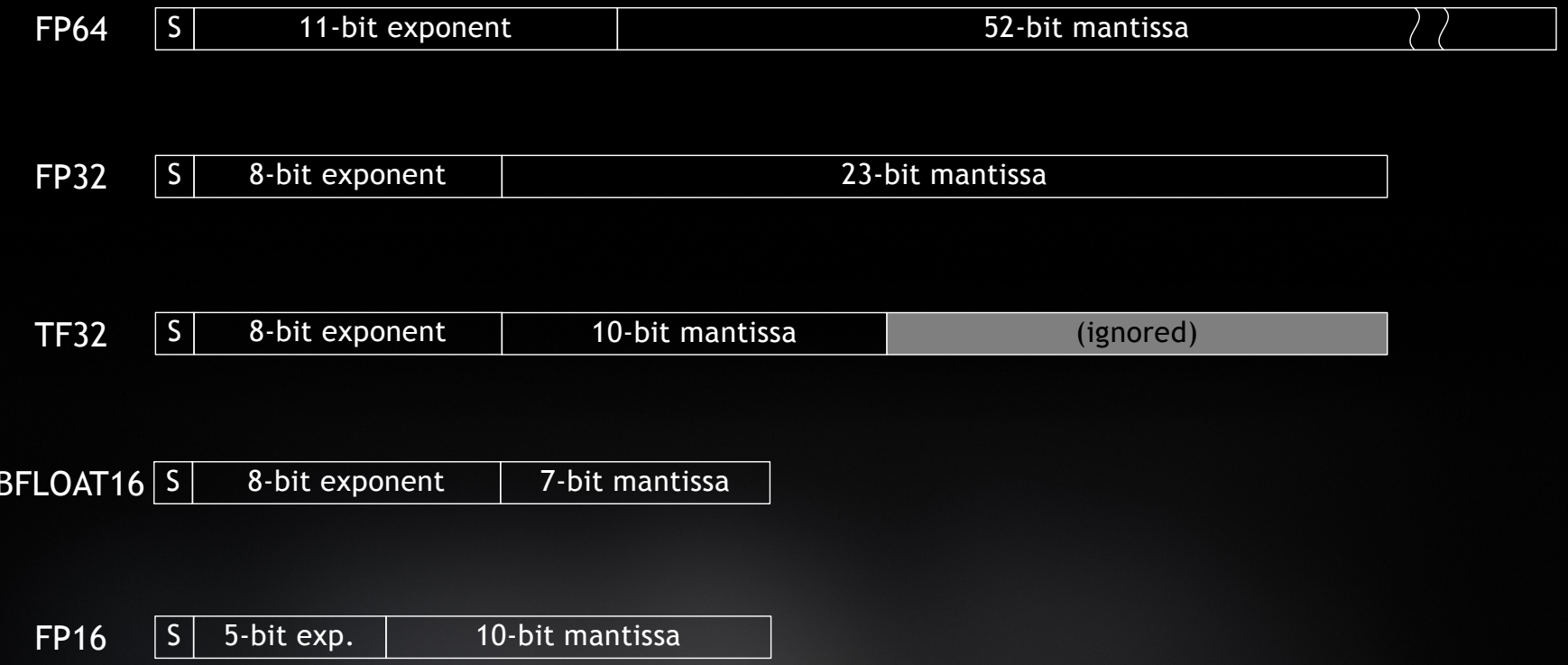
Great improvement for the 2 major stall reasons, syncthreads and memory loads




ALTERNATE FLOATING-POINT
FORMATS

FLOATING-POINT FORMATS

Native FP formats in A100



FLOATING-POINT FORMATS

Reduced precision benefits

- Reduce memory footprint
- Reduce memory bandwidth
- More FLOPS/ byte
- Compute units that have higher peak FLOPS capabilities

FP FORMATS

A100 Capabilities

A100	Scalar TFlops	Vector TFlops	TensorCore TFlops	Max val	Smallest normal > 0	Smallest inc. to 1.0
FP64	9.7	9.7	19.5	$\approx 1.8 \times 10^{308}$	$\approx 2.2 \times 10^{-308}$	$\approx 2.2 \times 10^{-16}$
FP32	19.5	19.5	TF32 156 (312)*	$\approx 3.4 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 1.2 \times 10^{-7}$
FP16	19.5	78	312 (624)*	65504	$\approx 6.1 \times 10^{-5}$	$\approx 9.8 \times 10^{-4}$
BFLOAT16	19.5	39	312 (624)*	$\approx 3.3 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 7.8 \times 10^{-3}$

* With sparsity feature

FP FORMATS

A100 Capabilities

A100	Scalar TFlops	Vector TFlops	TensorCore TFlops	Max val	Smallest normal > 0	Smallest inc. to 1.0
FP64	9.7	9.7	19.5	$\approx 1.8 \times 10^{308}$	$\approx 2.2 \times 10^{-308}$	$\approx 2.2 \times 10^{-16}$
FP32	19.5	19.5	TF32 156 (312)*	$\approx 3.4 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 1.2 \times 10^{-7}$
FP16	19.5	78	312 (624)*	65504	$\approx 6.1 \times 10^{-5}$	$\approx 9.8 \times 10^{-4}$
BFLOAT16	19.5	39	312 (624)*	$\approx 3.3 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 7.8 \times 10^{-3}$

Vector Flops using __half2 / __nv_bfloat162

* With sparsity feature

FP FORMATS

A100 Capabilities

A100	Scalar TFlops	Vector TFlops	TensorCore TFlops	Max val	Smallest normal > 0	Smallest inc. to 1.0
FP64	9.7	9.7	19.5	$\approx 1.8 \times 10^{308}$	$\approx 2.2 \times 10^{-308}$	$\approx 2.2 \times 10^{-16}$
FP32	19.5	19.5	TF32 156 (312)*	$\approx 3.4 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 1.2 \times 10^{-7}$
FP16	19.5	78	312 (624)*	65504	$\approx 6.1 \times 10^{-5}$	$\approx 9.8 \times 10^{-4}$
BFLOAT16	19.5	39	312 (624)*	$\approx 3.3 \times 10^{38}$	$\approx 1.2 \times 10^{-38}$	$\approx 7.8 \times 10^{-3}$

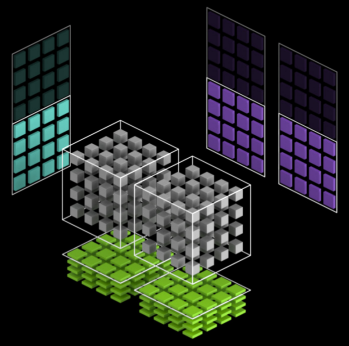
* With sparsity feature


THIRD GENERATION TENSOR CORES

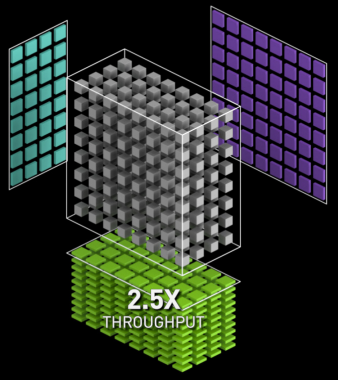
THIRD GENERATION TENSOR CORES

Nvidia V100 vs Nvidia A100

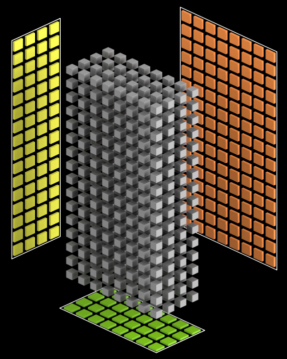
NVIDIA V100 Tensor Core FP16



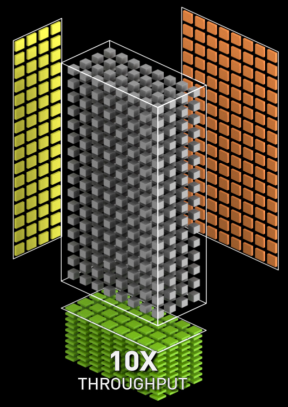
NVIDIA A100 Tensor Core FP16



NVIDIA V100 INT8



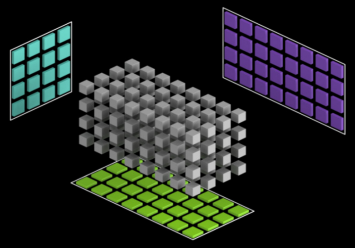
NVIDIA A100 Tensor Core INT8



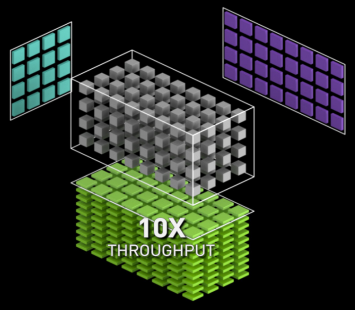
THIRD GENERATION TENSOR CORES

NVIDIA V100 vs NVIDIA A100

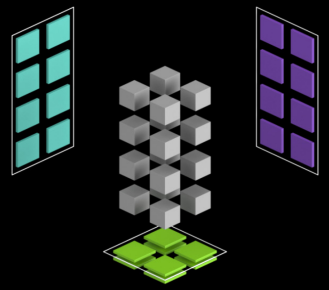
NVIDIA V100 FP32



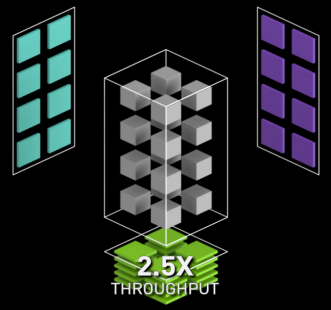
NVIDIA A100 Tensor Core TF32



NVIDIA V100 FP64

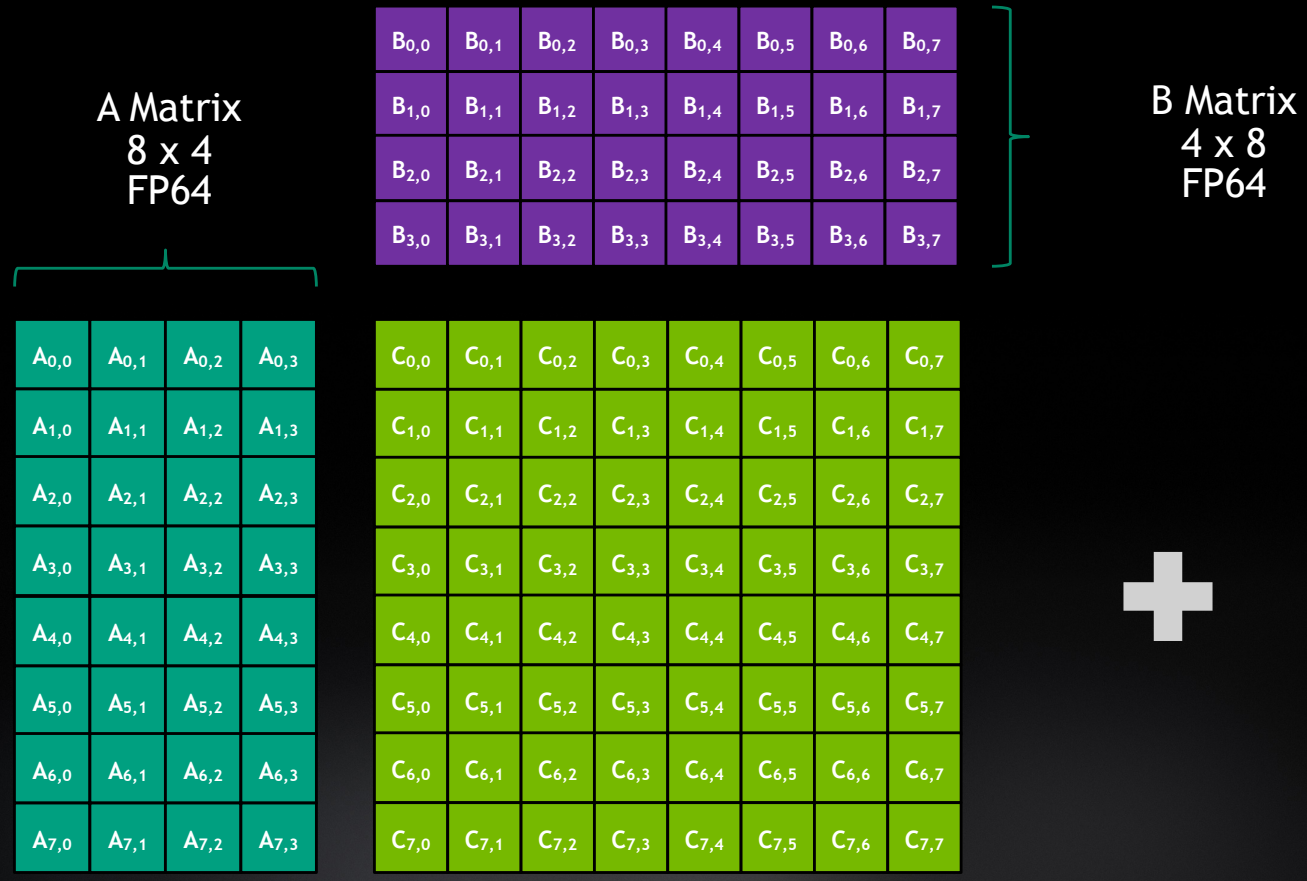


NVIDIA A100 Tensor Core FP64



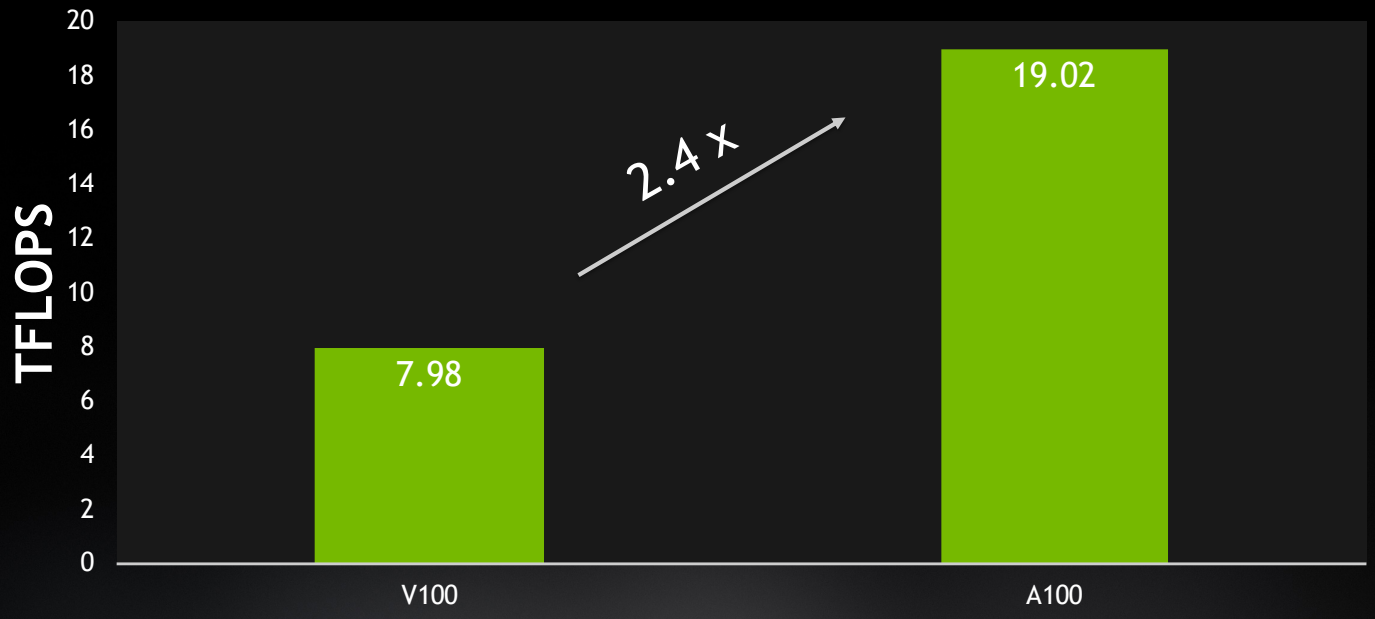
THIRD GENERATION TENSOR CORES

Warp Wide Double Precision Tensor Core (DMMA)



THIRD GENERATION TENSOR CORE

DGEMM Performance using FP64 Tensor Core



cuBLAS DGEMM Performance. Matrix Dimensions M = 4096, N = 4096, K =4096

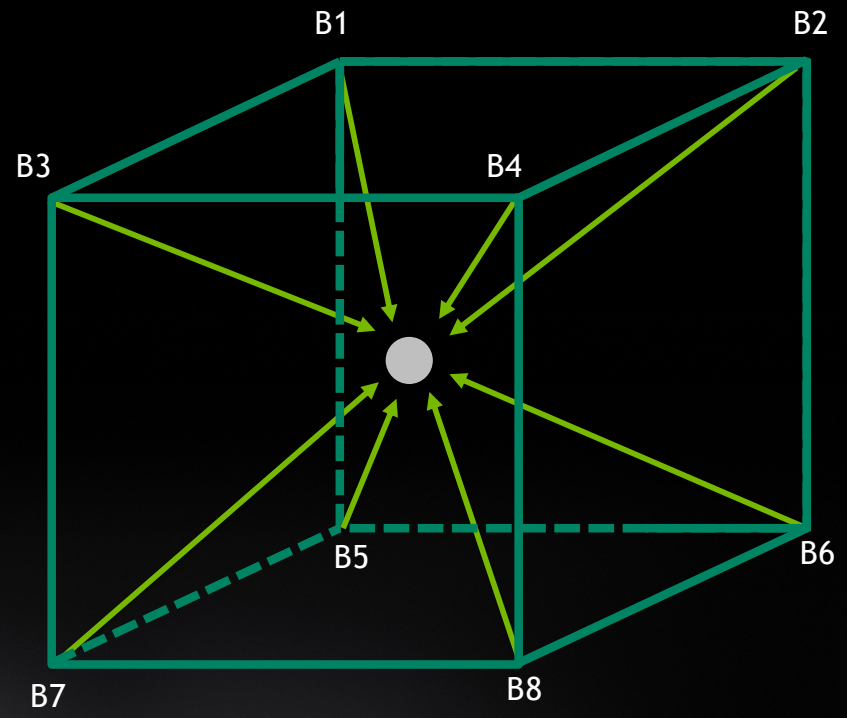
THIRD GENERATION TENSOR CORES

Particle in Cell

- Thread Block level GEMM using CUDA WMMA API
- The governing equation for particle velocity in magnetic field is given by:

$$\frac{\mathrm{d}v}{\mathrm{d}t} = \frac{q}{m(\mathrm{v} \times \mathrm{B})},$$

$v = velocity, q = charge, m = mass, B = magnetic\ field$

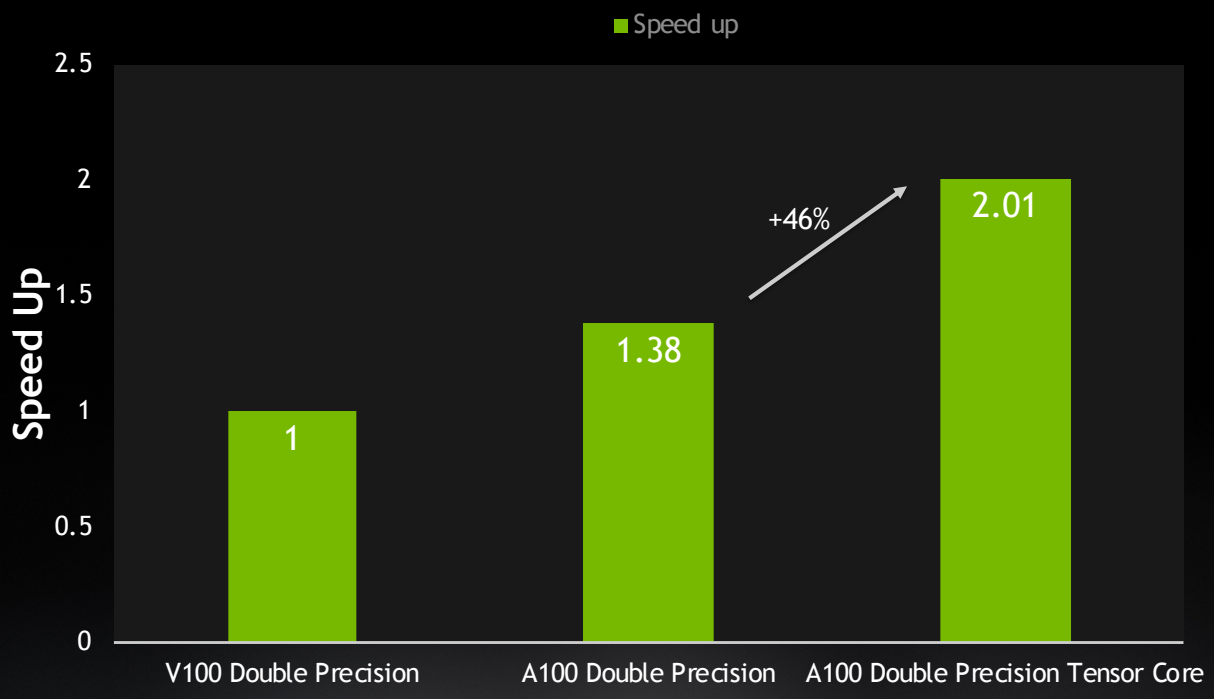


Gather by magnetic forces from the cell vertices.

*Ref: https://www.particleincell.com/2011/vxb-rotation/

THIRD GENERATION TENSOR CORES

Expressing algorithms as small matrix product to leverage Tensor Cores



CONCLUSION

Lots of new features in A100!

40 GB of HBM2, with 1.55 TB/s Memory Bandwidth

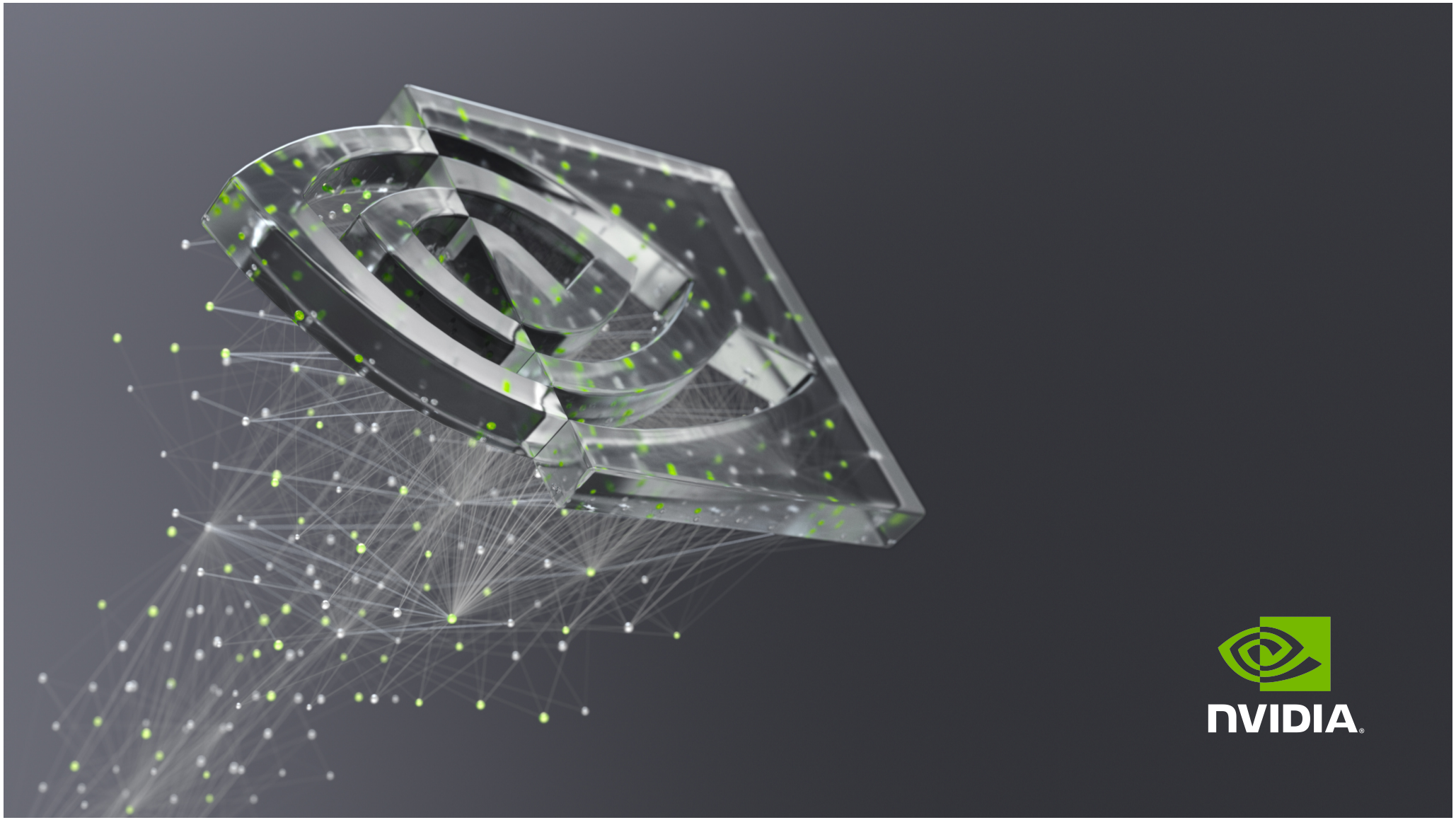
40 MB L2 Cache + L2 Residency Control to improve L2 efficiency

Compute Data Compression can increase your effective bandwidth

192 KB of combined L1/Shared Memory + Async Copy helps hide latencies

More FP format choices, faster 3rd Gen Tensor Core support across all formats

Not an extensive list! See other GTC'20 Talks!


NVIDIA