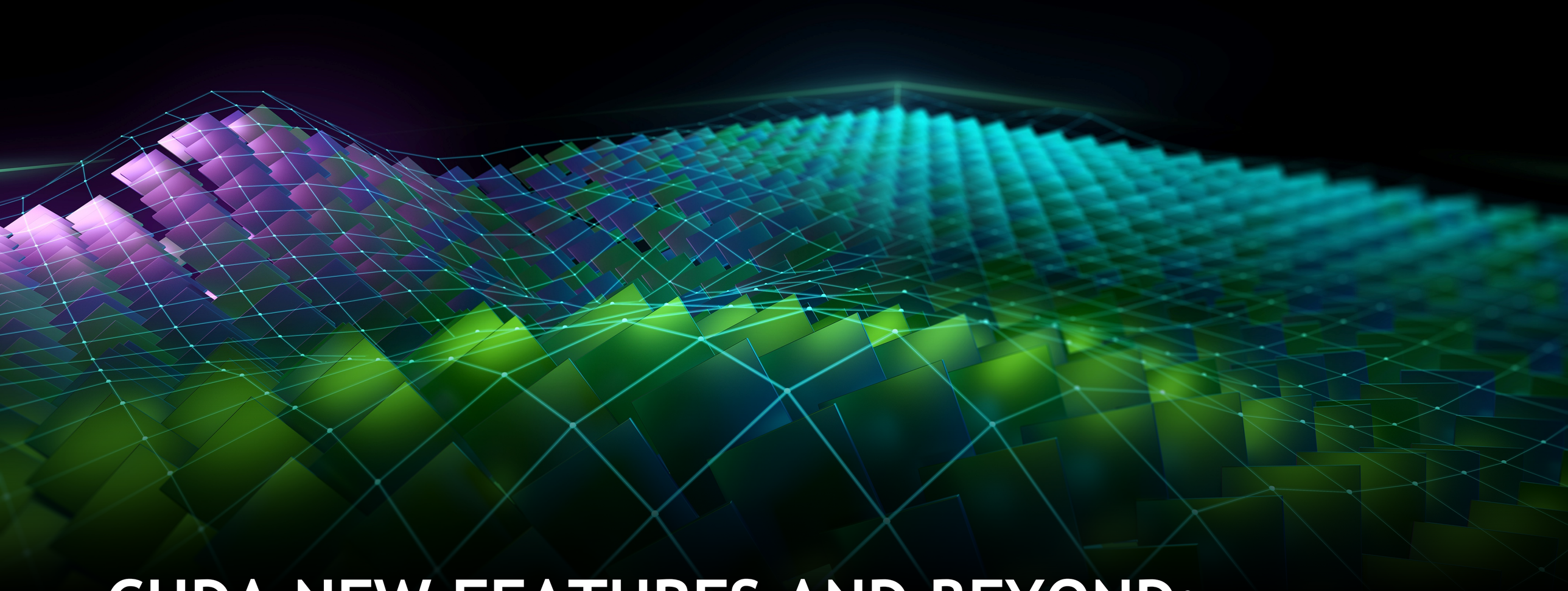


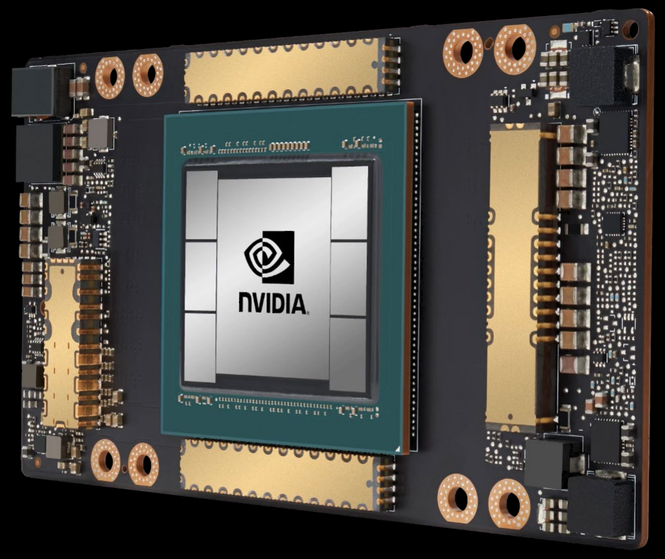
CUDA NEW FEATURES AND BEYOND: AMPERE PROGRAMMING FOR DEVELOPERS

Stephen Jones, GTC Fall 2020

NVIDIA

THE NVIDIA AMPERE GA102 GPU ARCHITECTURE

Adding new RTX desktop GPUs alongside the datacenter-class A100



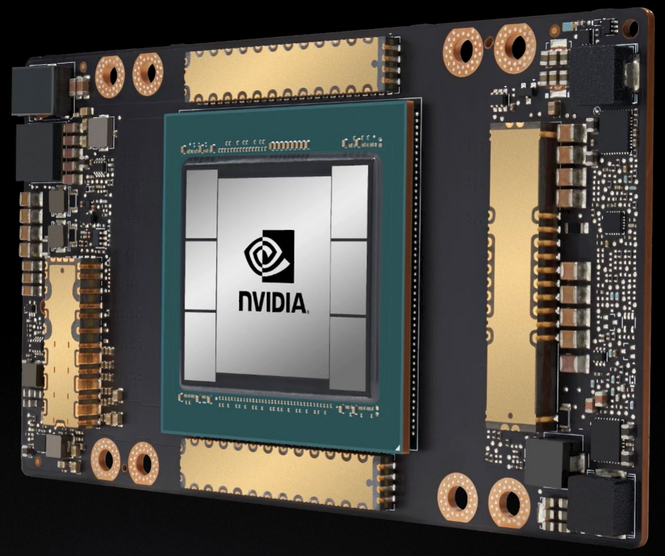
THE NVIDIA AMPERE GA102 GPU ARCHITECTURE

	Titan RTX	RTX 3090
SMs	72	82
Tensor Core Precision	FP16	TF32, BF16, FP16, I8, I4, B1
Shared Memory per Block	64 kB	96 kB
L2 Cache Size	6144 kB	6144 kB
Memory Bandwidth	672 GB/sec	936 GB/sec
RT Cores	72 (1st gen)	82 (2nd gen)

THE NVIDIA AMPERE GPU ARCHITECTURE

NVIDIA Ampere Architectural Features

- Multi-Instance GPU (A100 only)
- Asynchronous barriers
- Asynchronous data movement
- L2 cache management
- Task graph acceleration
- New Tensor Core precisions



For full complete details of CUDA on the NVIDIA Ampere GPU Architecture, see the GTC May 2020 CUDA 11.0 talk: https://developer.nvidia.com/gtc/2020/video/s21760

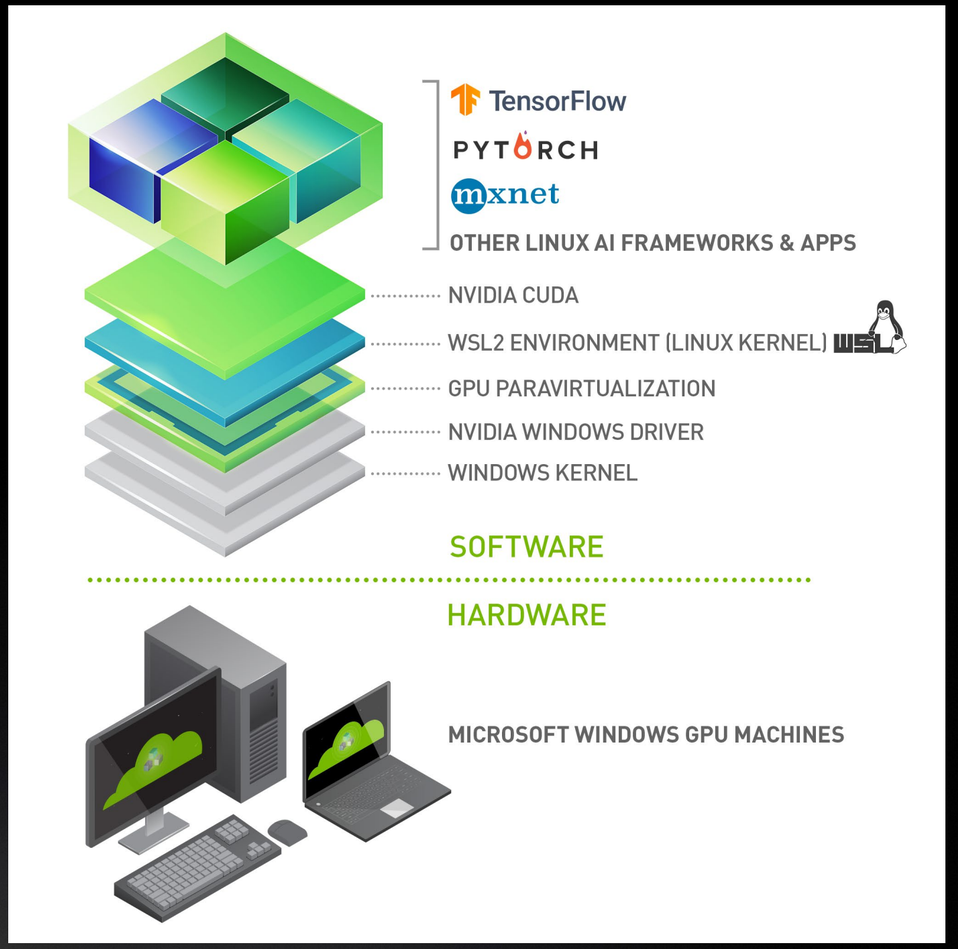
CUDA ON WINDOWS SUBSYSTEM FOR LINUX

Run a Linux kernel **natively** on top of Windows 10

Runs Linux at near full speed **without emulation**

Multi-OS development & testing from a single Windows desktop machine

No need for dual-boot systems - ideal for laptops



CUDA ON WINDOWS SUBSYSTEM FOR LINUX

Preview Available as Part of Microsoft Windows Insider Program

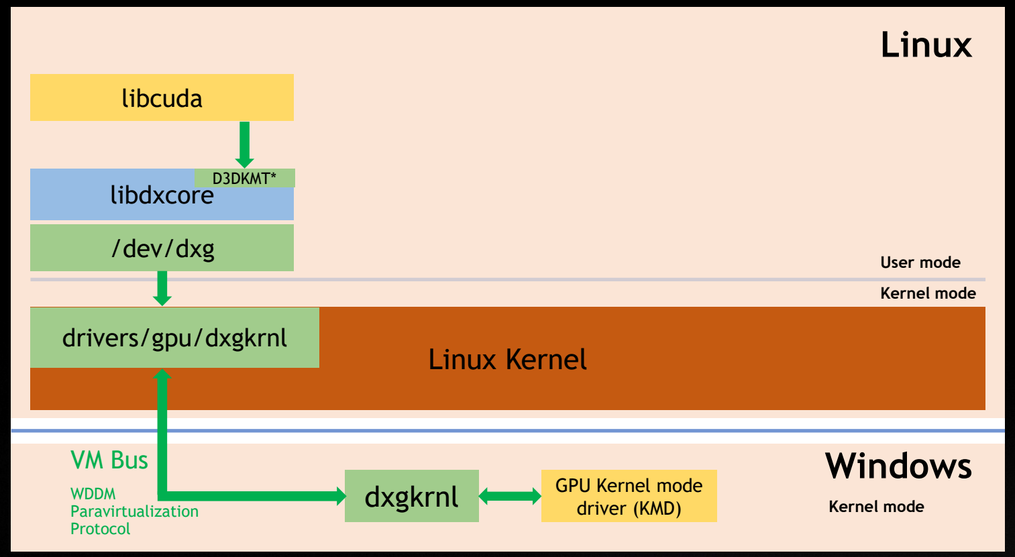


Diagram of the WDDM model supporting the CUDA user mode driver running inside Linux guest

GPU support is now available for WSL 2 users

Run GPU-accelerated Linux applications natively on your Windows desktop platform

Getting started is simple:

1. Enable WIP in your Windows system settings
2. Download preview CUDA WSL driver: https://developer.nvidia.com/cuda/wsl

GPU-ACCELERATED DATA SCIENCE ON WSL

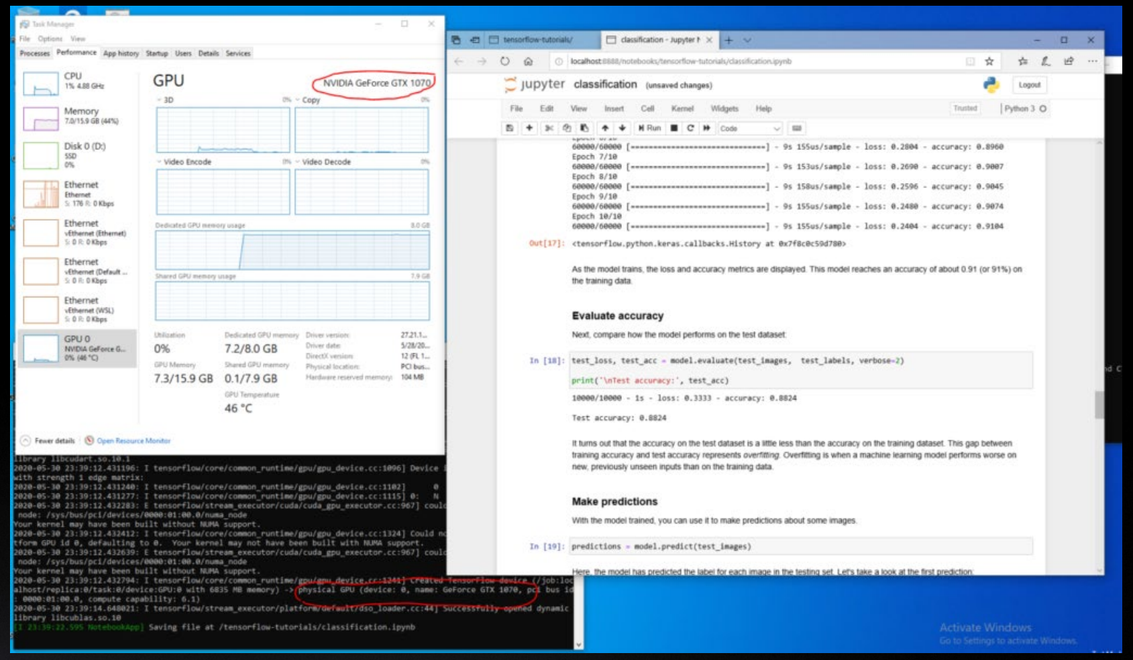
Get the latest version of Docker and run:

- AI Frameworks (PyTorch, TensorFlow)
- RAPIDS & ML Applications
- Jupyter Notebooks

GPU-enabled DirectX, CUDA 11.1 and the NVIDIA Container Toolkit are all available on WSL today

NVML and NCCL support coming soon

See CUDA-on-WSL blog for full details: https://developer.nvidia.com/blog/announcing-cuda-on-windows-subsystem-for-Linux-2/

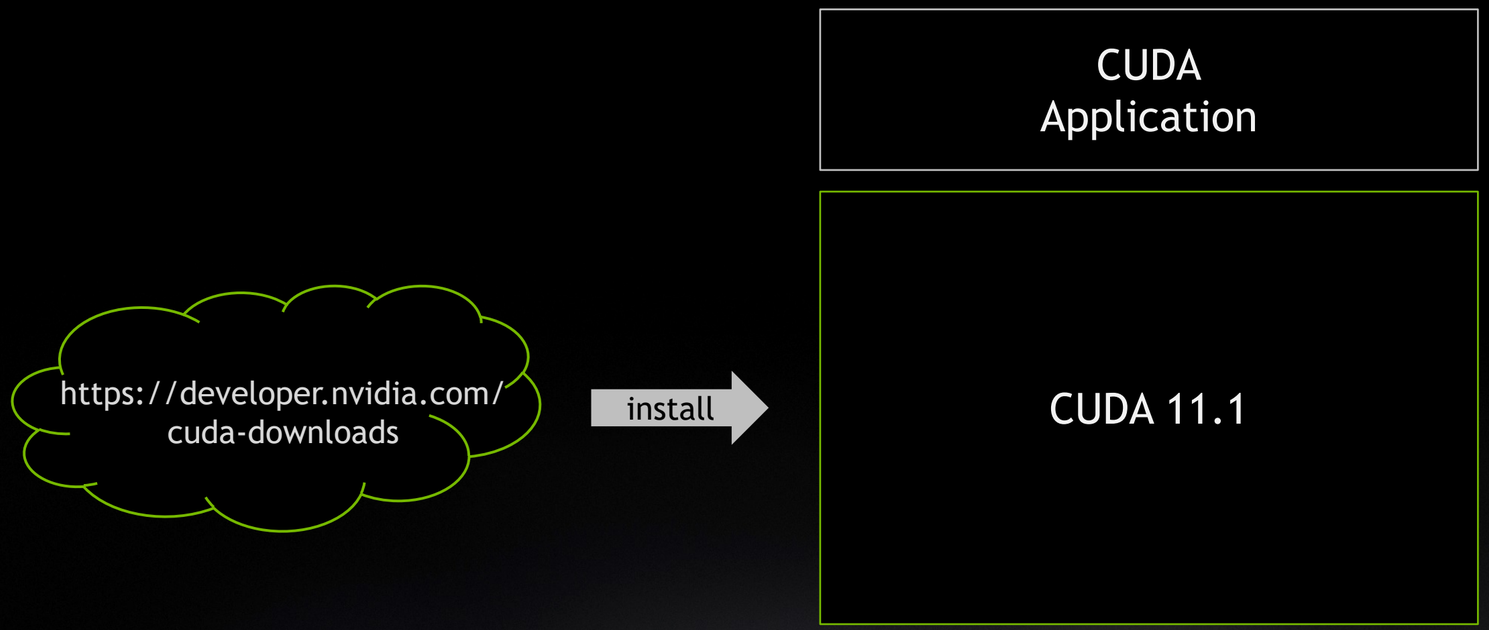


TensorFlow container running inside WSL 2

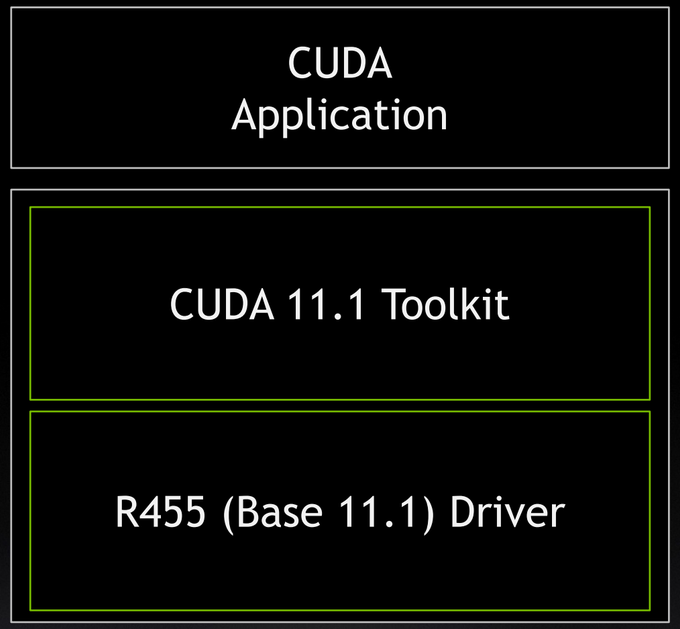
ANATOMY OF A CUDA APPLICATION

CUDA
Application

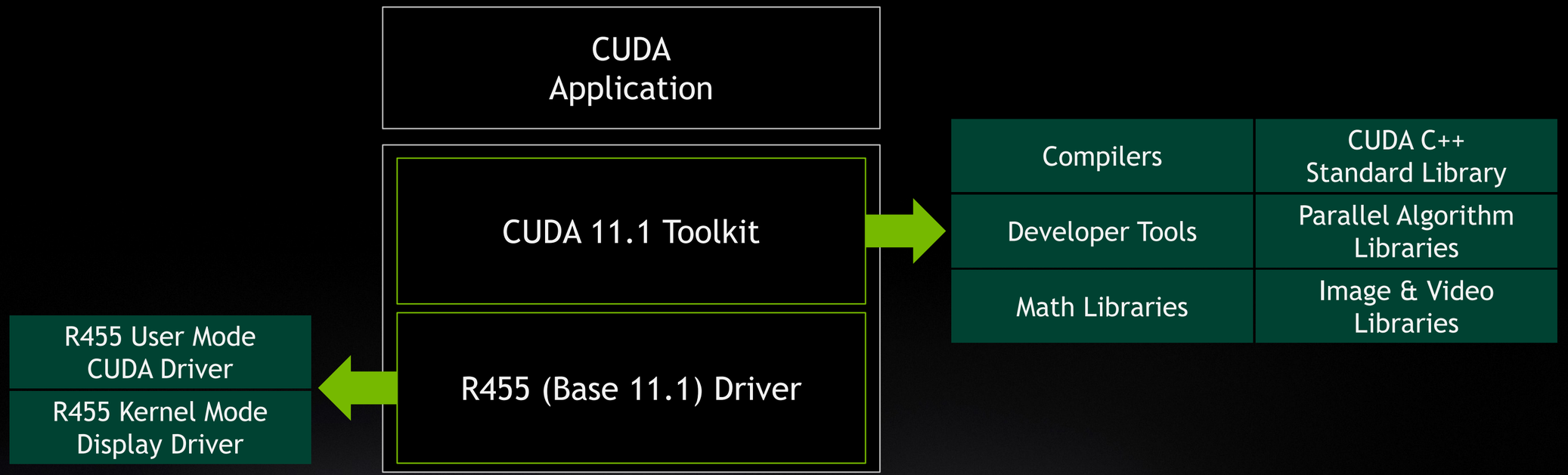
ANATOMY OF A CUDA APPLICATION



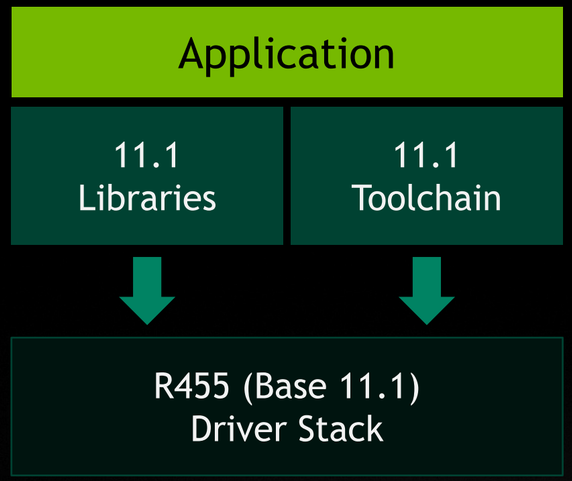
ANATOMY OF A CUDA APPLICATION



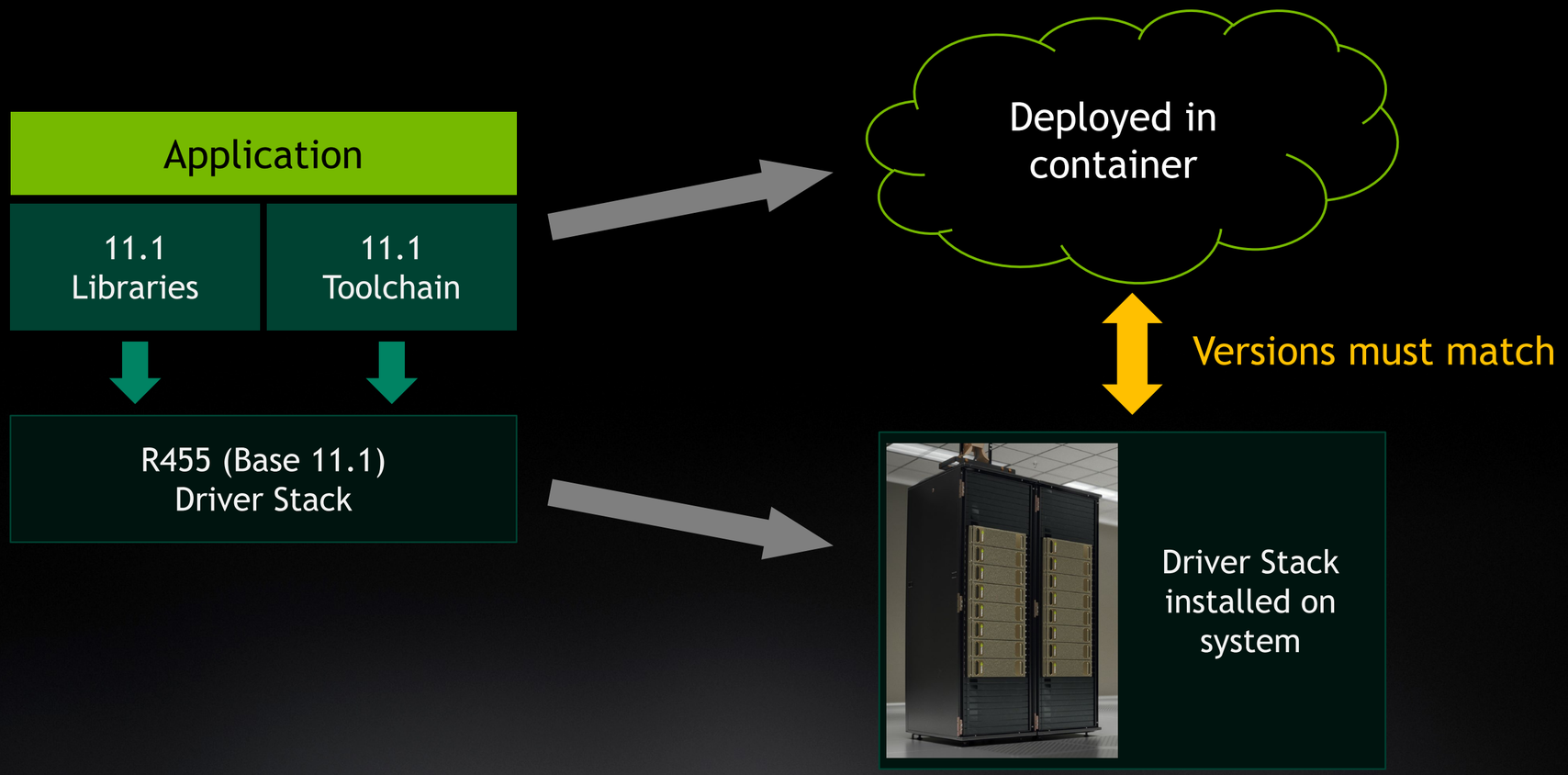
ANATOMY OF A CUDA APPLICATION



ANATOMY OF A CUDA APPLICATION



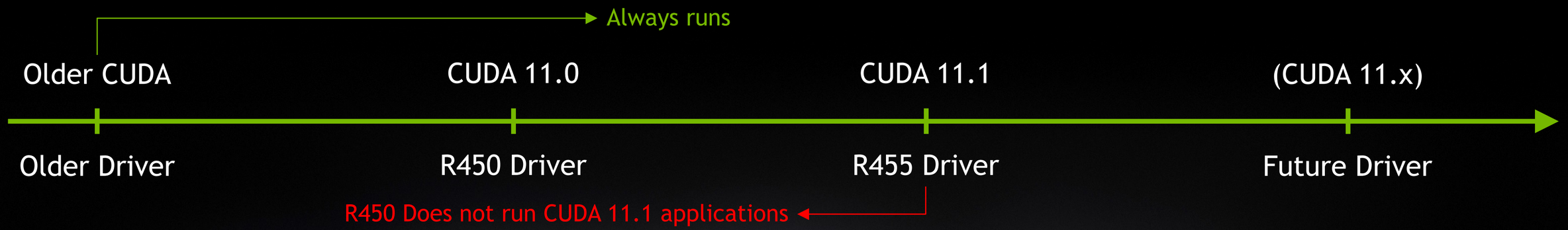
ENTERPRISE & CLOUD DEPLOYMENT, TODAY



BACKWARD COMPATIBILITY BEFORE CUDA 11.1

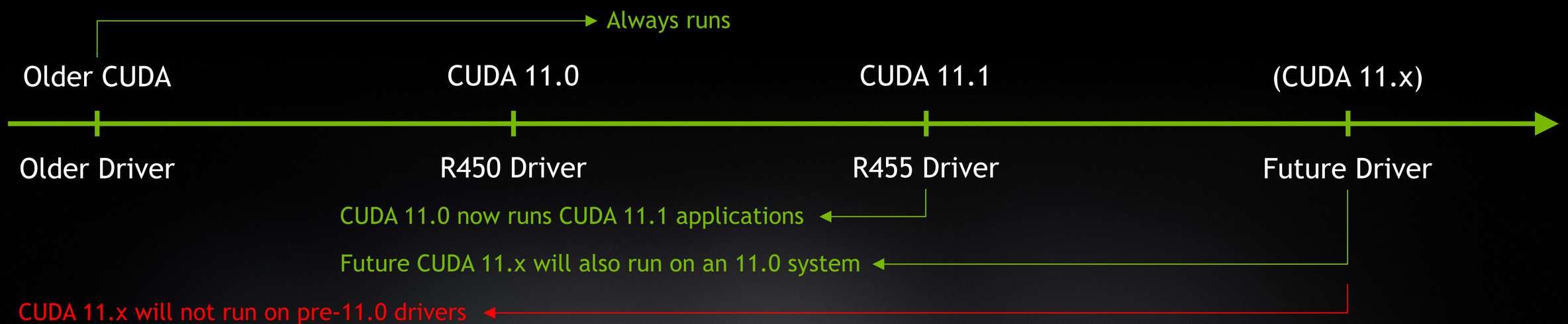
Older CUDA **always** runs on **newer** drivers

Future CUDA **does not** run on **older** drivers



ENHANCED COMPATIBILITY SINCE CUDA 11.1

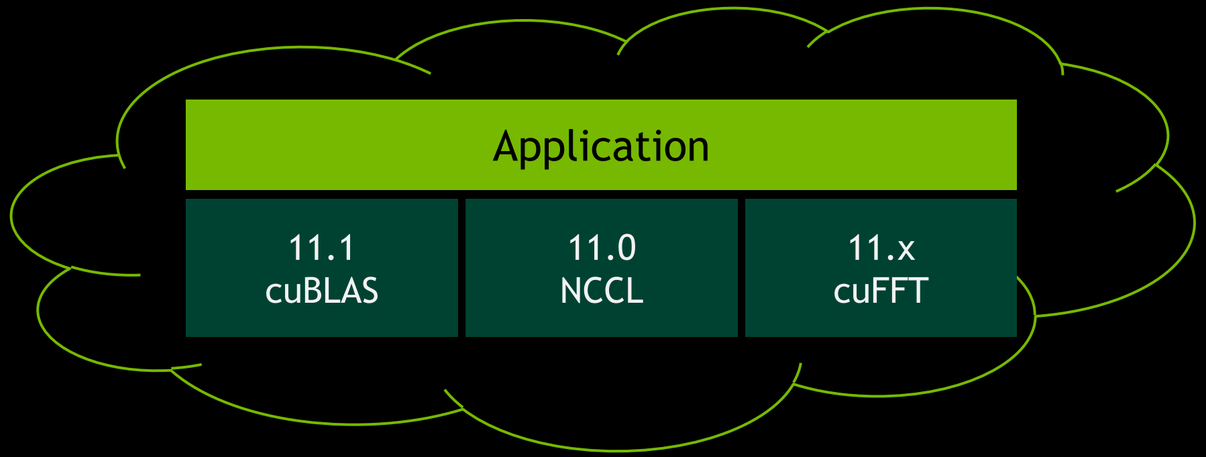
- Older CUDA **always** runs on **newer** drivers
- Newer CUDA 11.x **will** now run on older drivers with the **same major version**



CUDA ENHANCED COMPATIBILITY

Build with any 11.x toolchain

Libraries from different CUDA versions run on any base driver

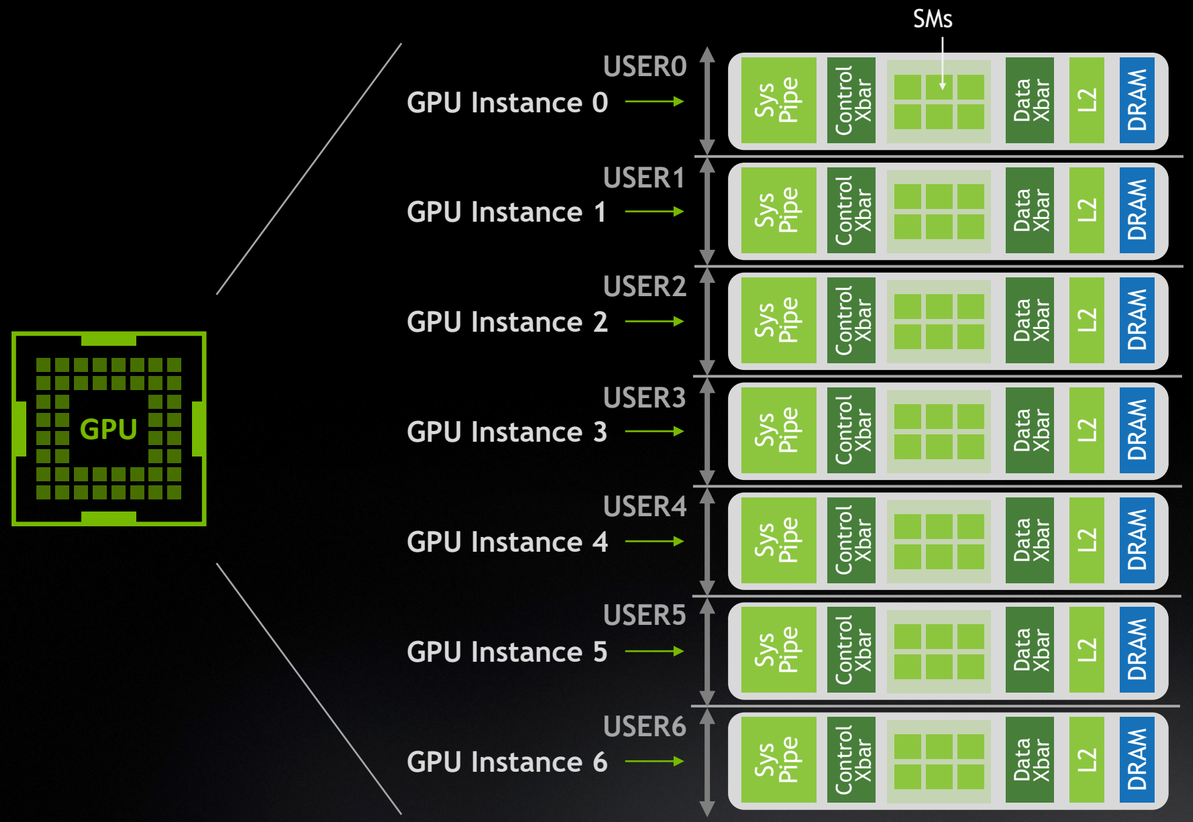


Deploy on any 11.x system

Runs on **any** CUDA 11.x base driver stack

NEW MULTI-INSTANCE GPU (MIG)

Divide a Single A100 GPU Into Multiple *Instances*, Each With Isolated Paths Through the Entire Memory System



Up To 7 GPU Instances In a Single A100

Full software stack enabled on each instance, with dedicated SM, memory, L2 cache & bandwidth

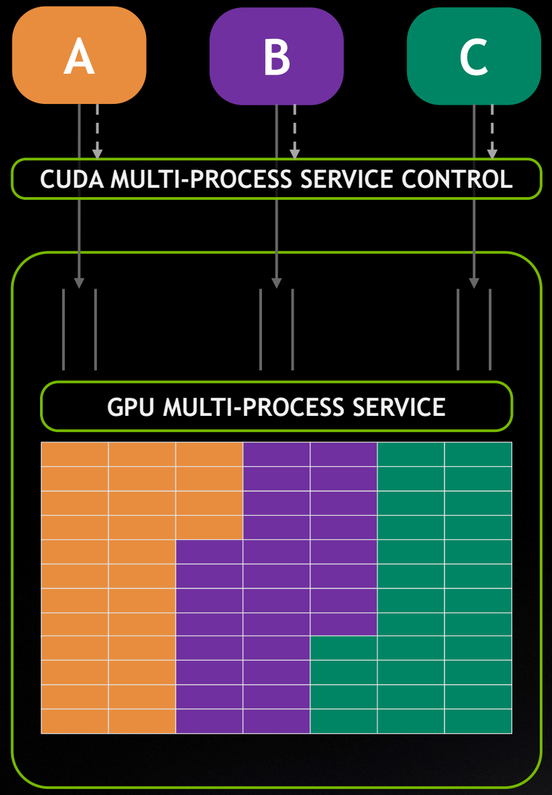
Simultaneous Workload Execution With Guaranteed Quality Of Service

All MIG instances run in parallel with predictable throughput & latency, fault & error isolation

Diverse Deployment Environments

Supported with Bare metal, Docker, Kubernetes Pod, Virtualized Environments

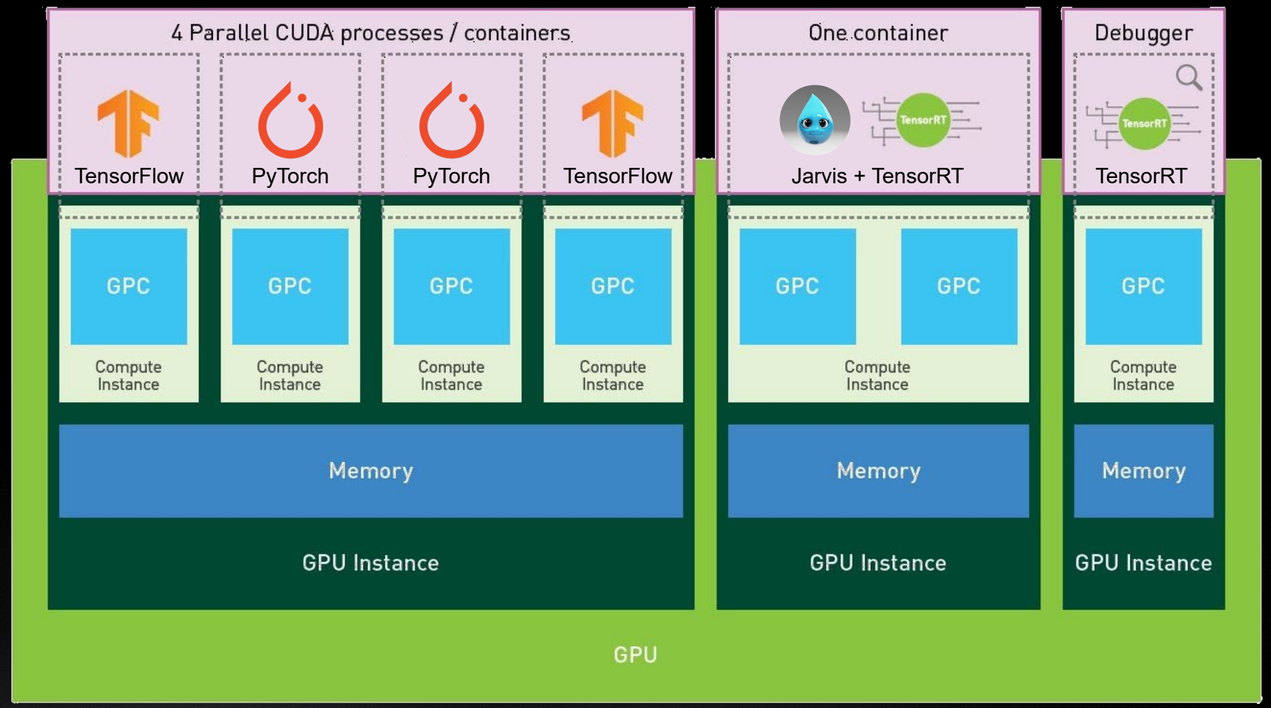
LOGICAL VS. PHYSICAL PARTITIONING



Multi-Process Service

Dynamic contention for GPU resources

Single tenant



Multi-Instance GPU

Hierarchy of instances with guaranteed resource allocation

Multiple tenants

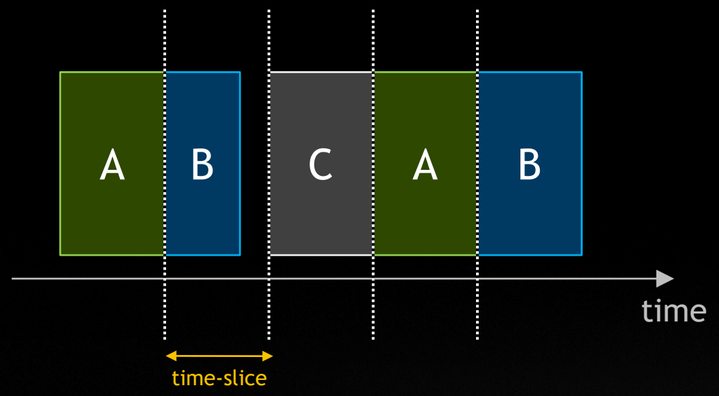
CUDA CONCURRENCY MECHANISMS

	Streams	MPS	MIG
Partition Type	Single process	Logical	Physical
Max Partitions	Unlimited	48	7
Fractional Provisioning	No	Yes	Yes
Memory Protection	No	Yes	Yes
Memory Bandwidth QoS	No	No	Yes
Fault Isolation	No	No	Yes
Cross-Partition Interop	Always	IPC	Limited IPC
Reconfigure	Dynamic	Process launch	When idle

EXECUTION SCHEDULING & MANAGEMENT

Pre-emptive scheduling

Processes share GPU through time-slicing
Scheduling managed by system

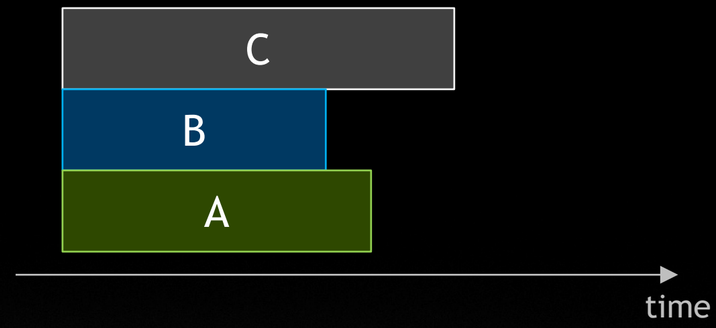



```
$ nvidia-smi compute-policy
  --set-timeslice={default, short, medium, long}
```

Time-slice configurable via **nvidia-smi**

Concurrent scheduling

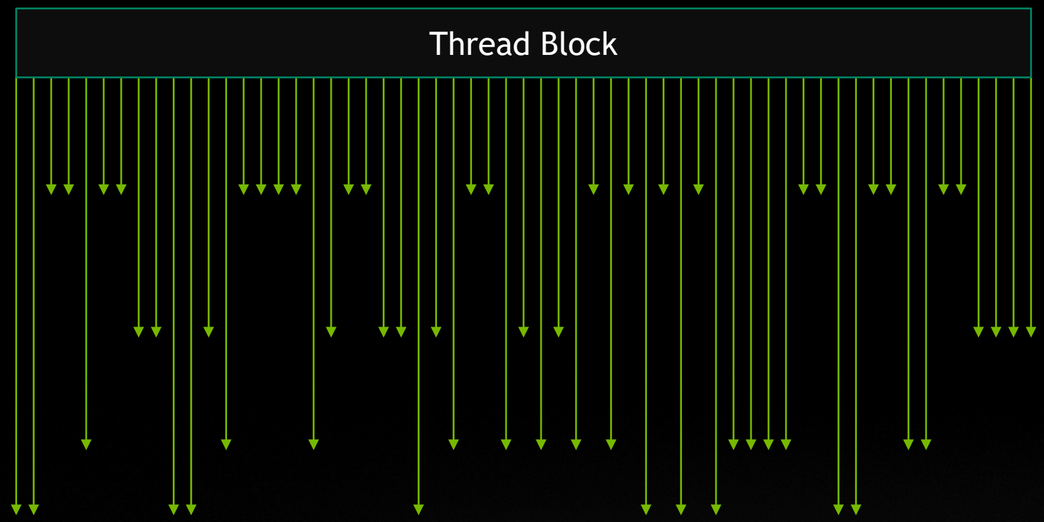
Processes run on GPU simultaneously
User creates & manages scheduling streams




```
cudaStreamCreateWithPriority(pStream, flags, priority);
cudaDeviceGetStreamPriorityRange(leastPriority, greatestPriority);
```

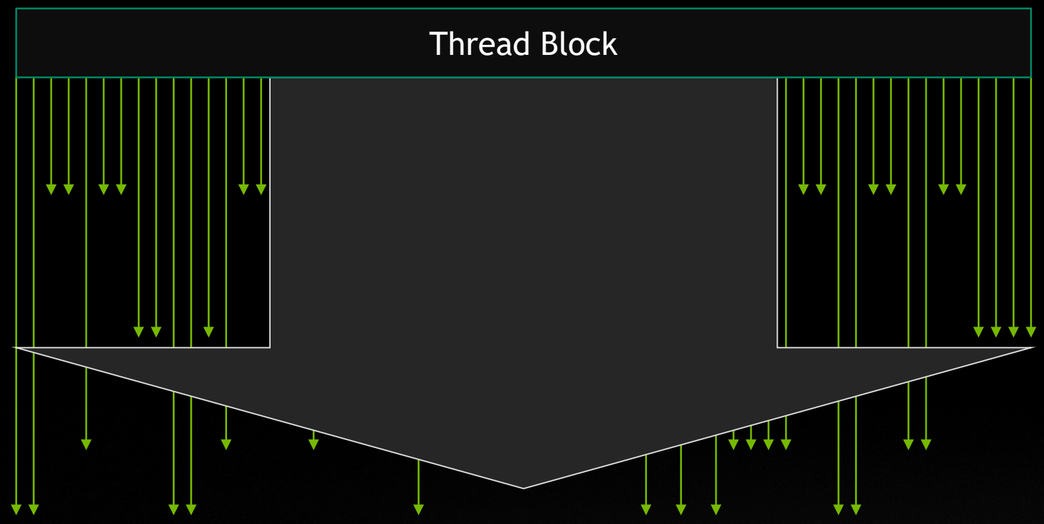
CUDA 11.0 adds a new **stream priority** level

FINE-GRAINED SYNCHRONIZATION




```
__syncthreads()
```

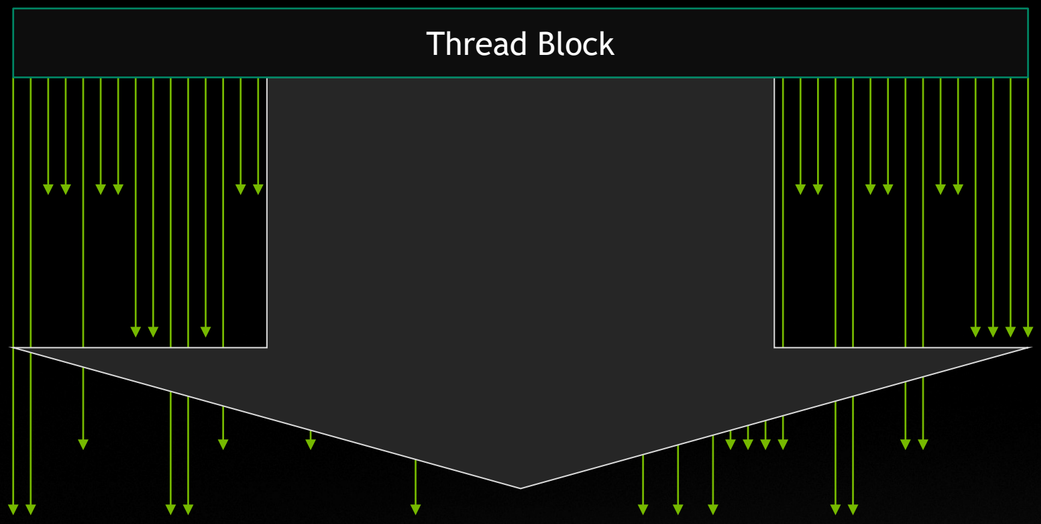
FINE-GRAINED SYNCHRONIZATION

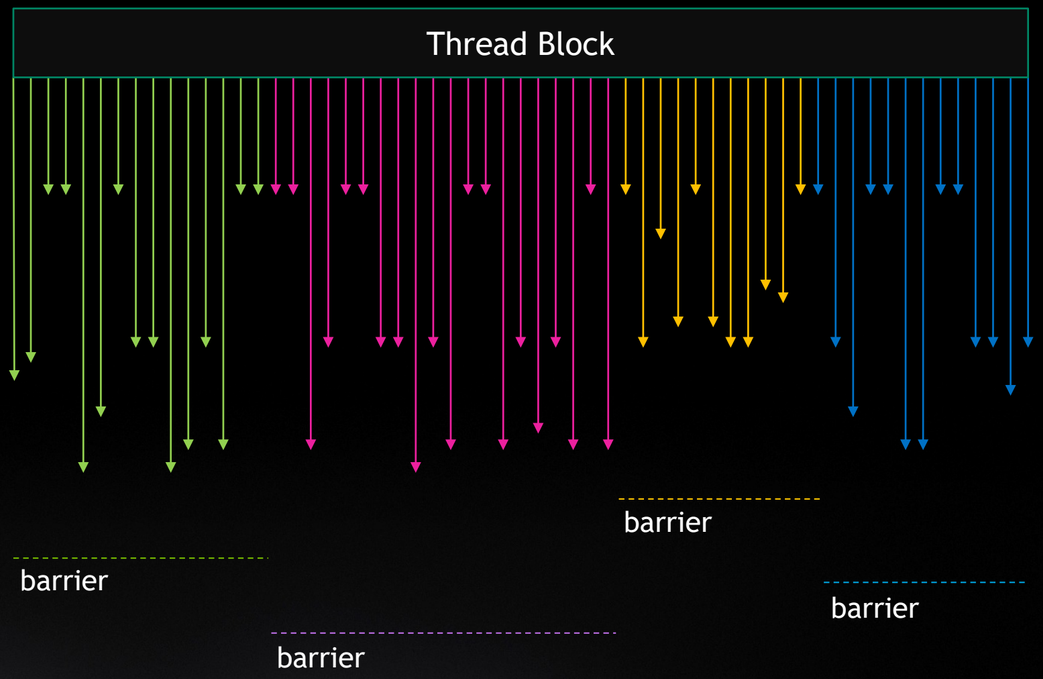



```
__syncthreads()
```

FINE-GRAINED SYNCHRONIZATION

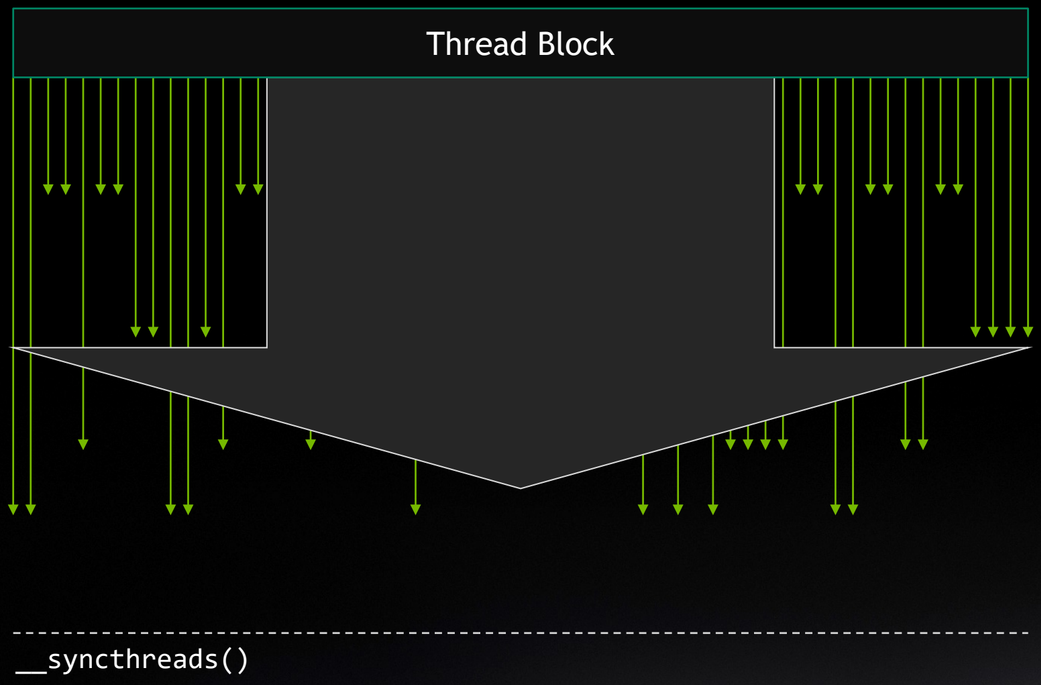
NVIDIA Ampere GPU Architecture Allows Creation Of Arbitrary Barriers

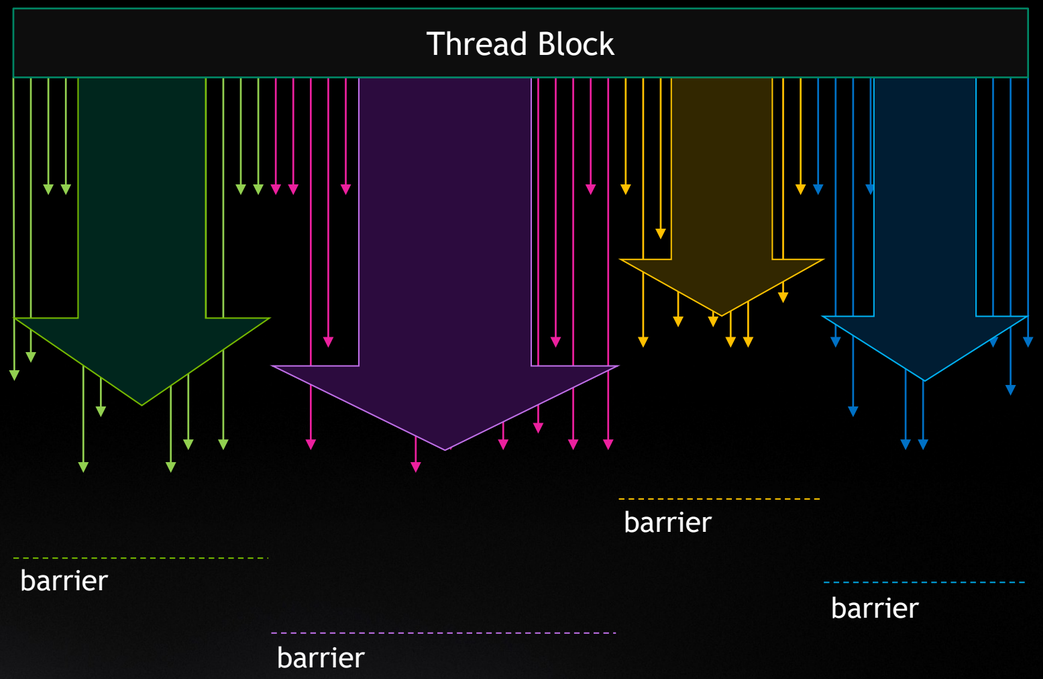




FINE-GRAINED SYNCHRONIZATION

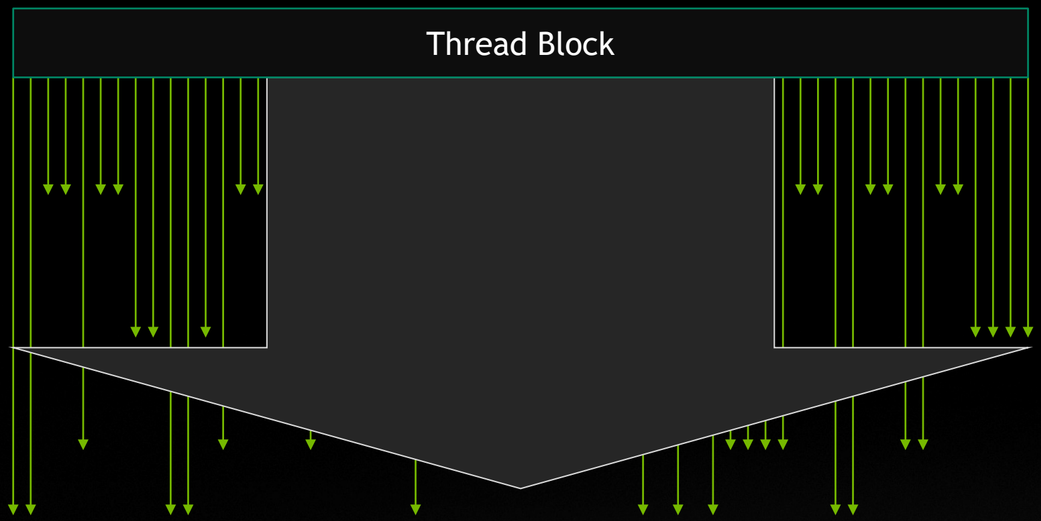
NVIDIA Ampere GPU Architecture Allows Creation Of Arbitrary Barriers



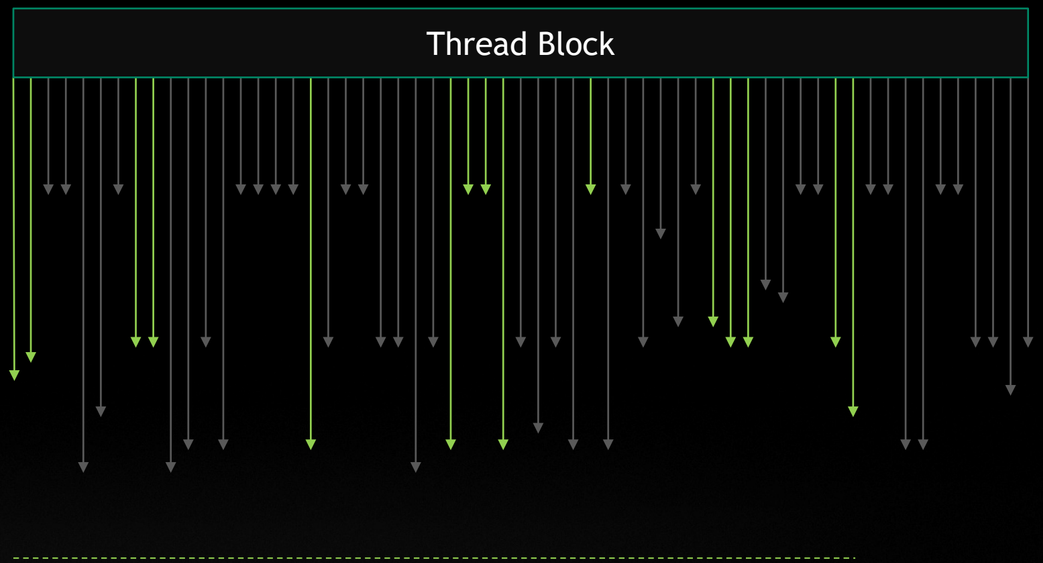


FINE-GRAINED SYNCHRONIZATION

NVIDIA Ampere GPU Architecture Allows Creation Of Arbitrary Barriers

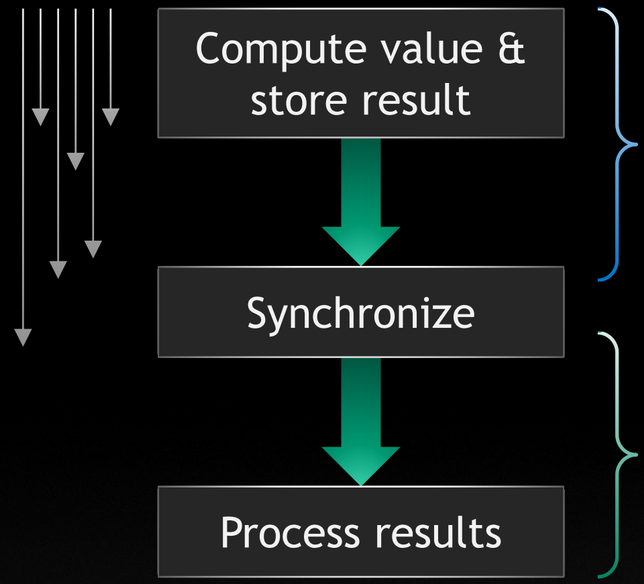


__syncthreads()

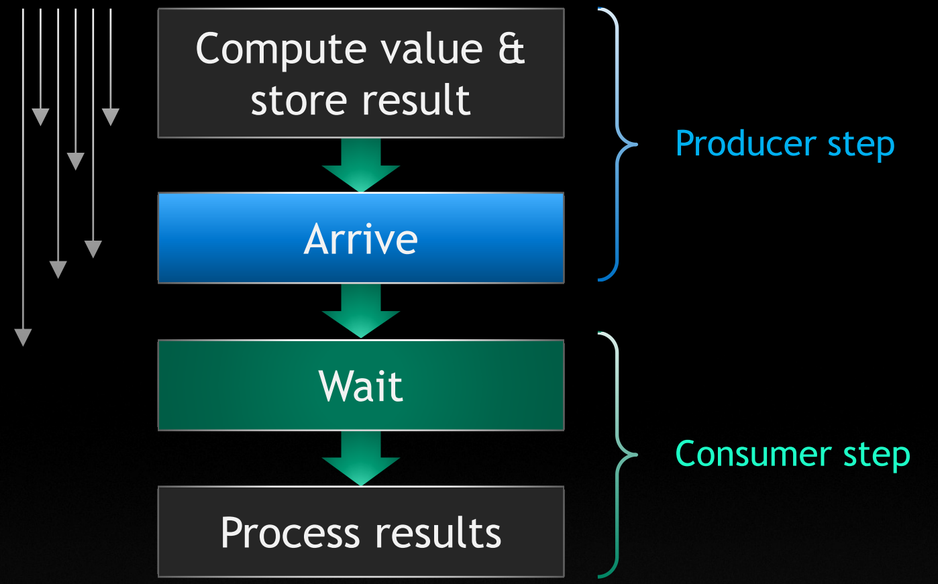


barrier

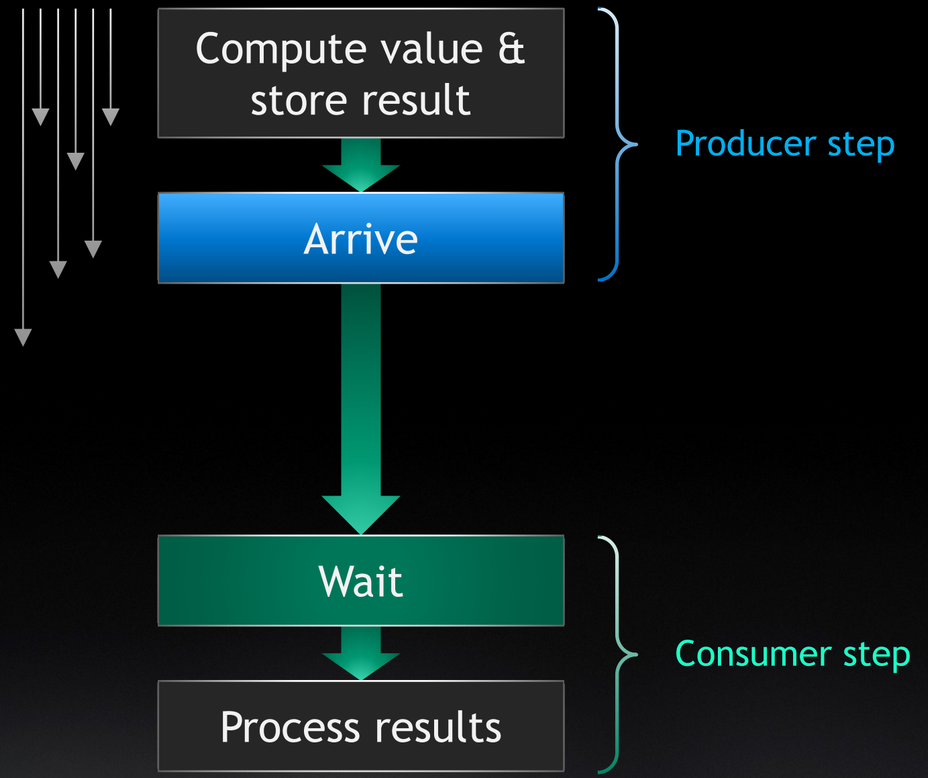
BARRIERS ALLOW EXCHANGE OF INFORMATION



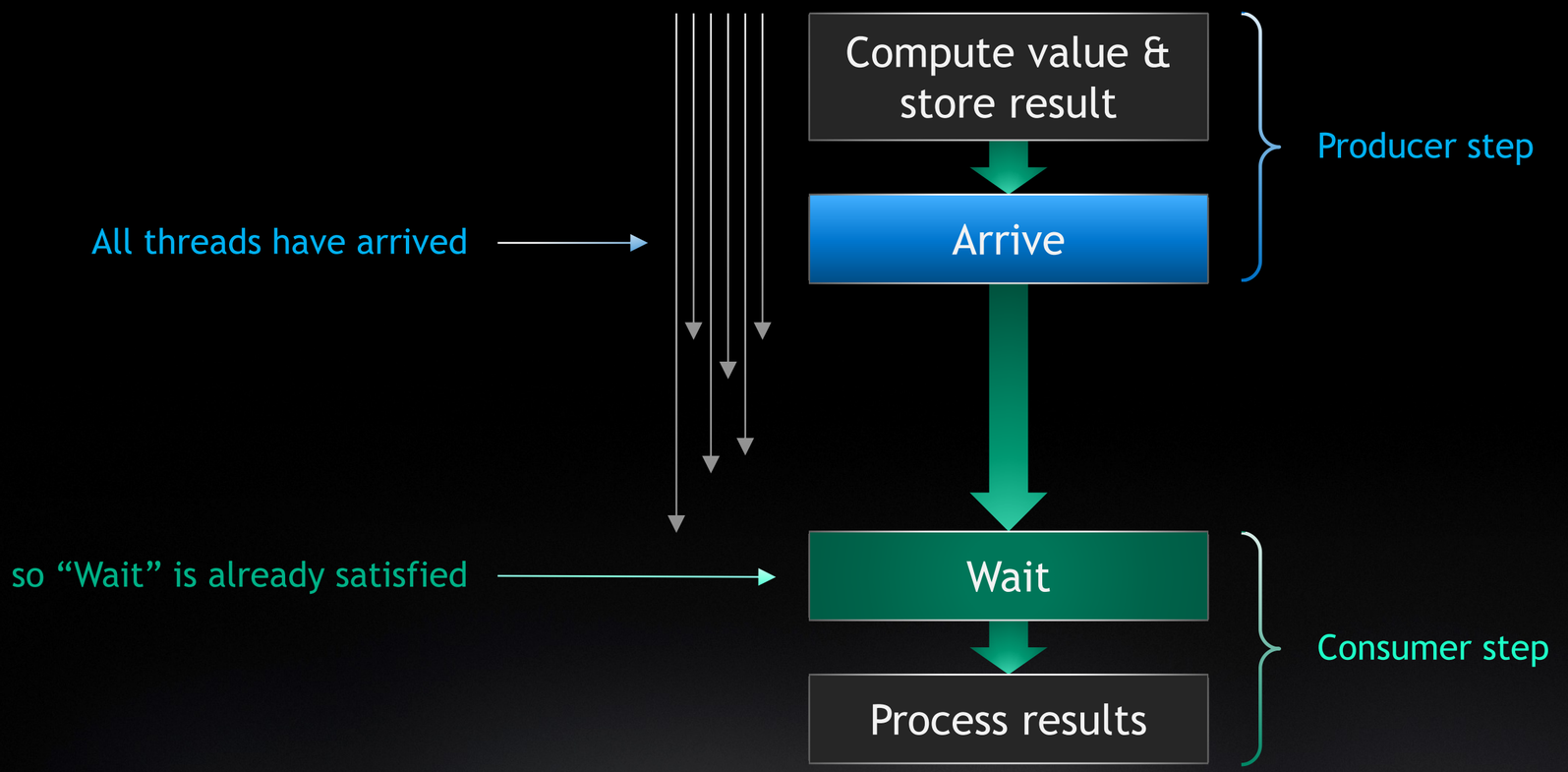
BARRIERS ALLOW EXCHANGE OF INFORMATION



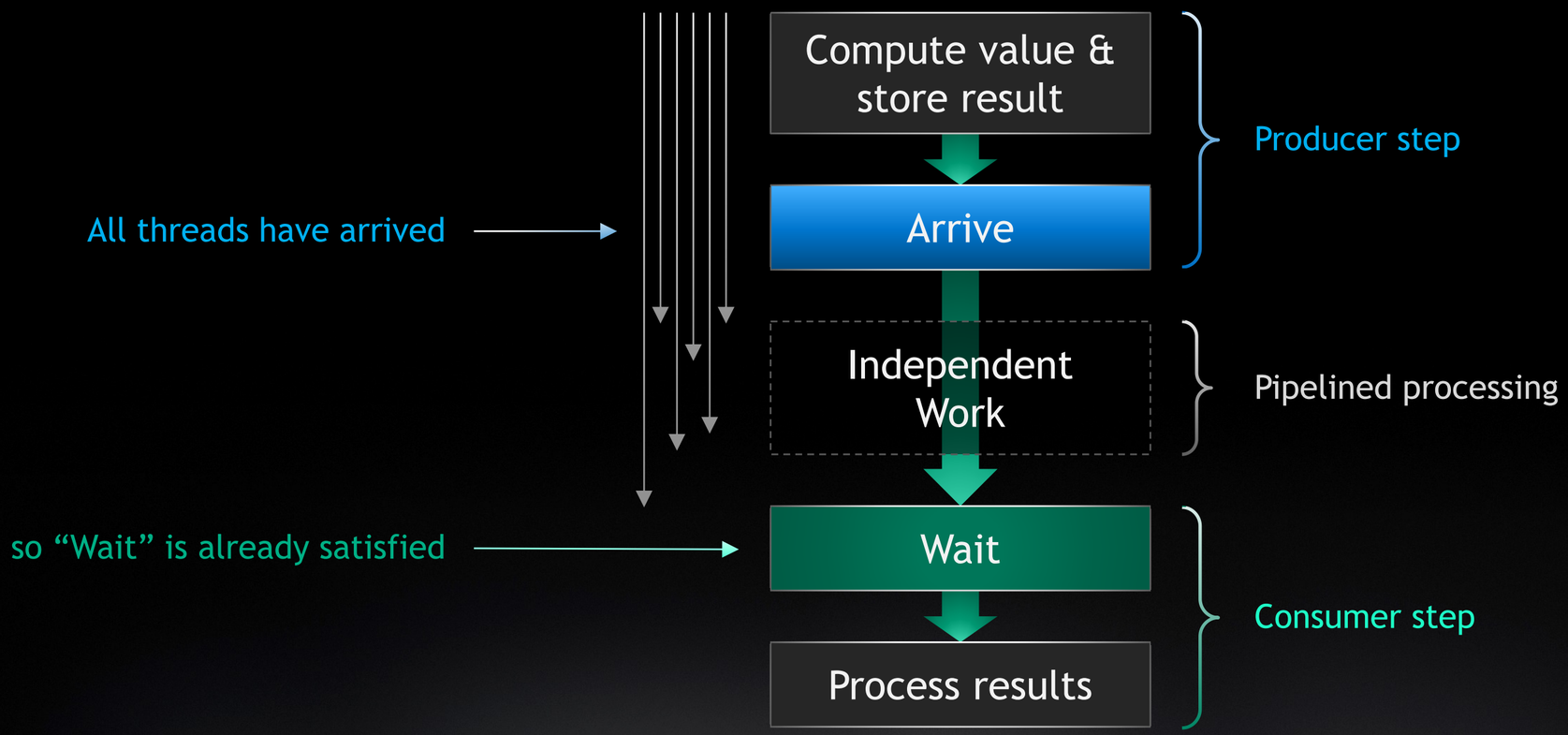
ASYNCHRONOUS BARRIERS



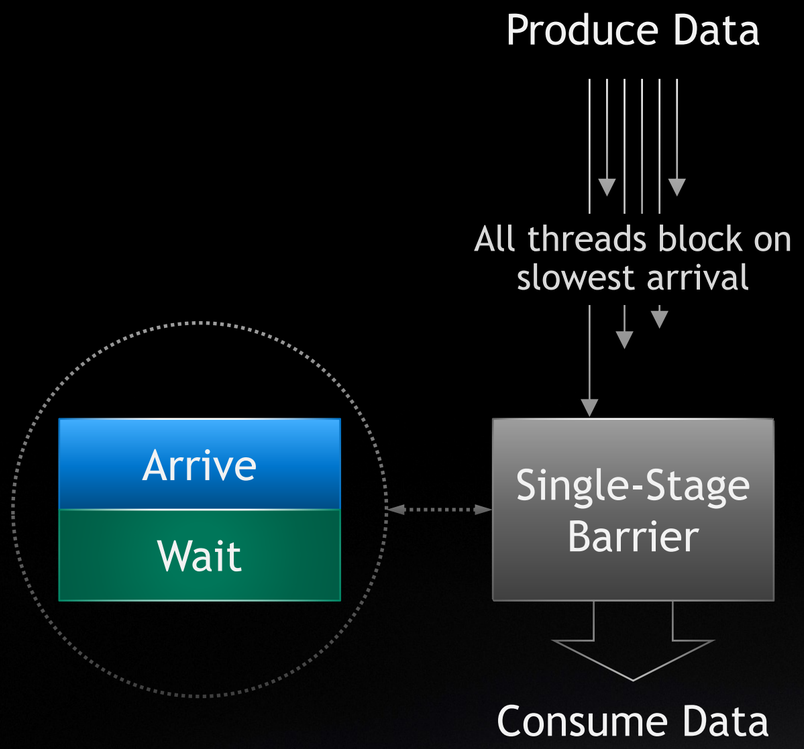
ASYNCHRONOUS BARRIERS

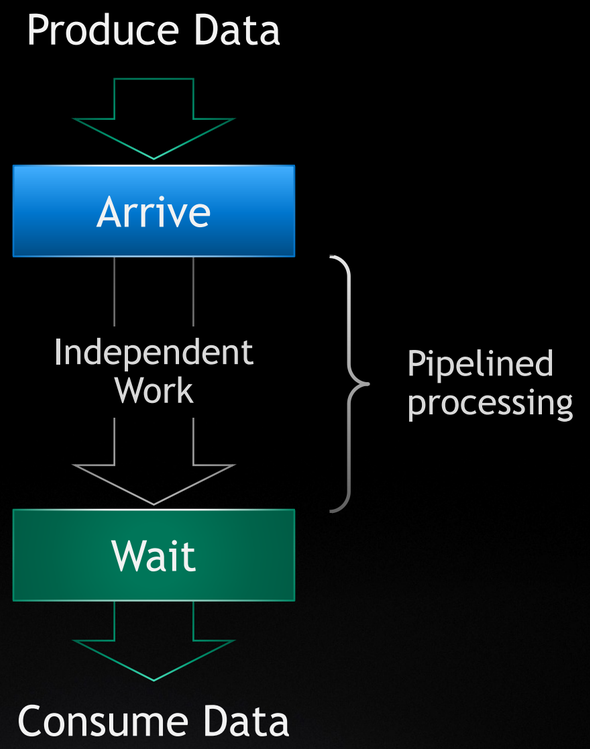


ASYNCHRONOUS BARRIERS



SINGLE-STAGE vs. ASYNCHRONOUS BARRIERS

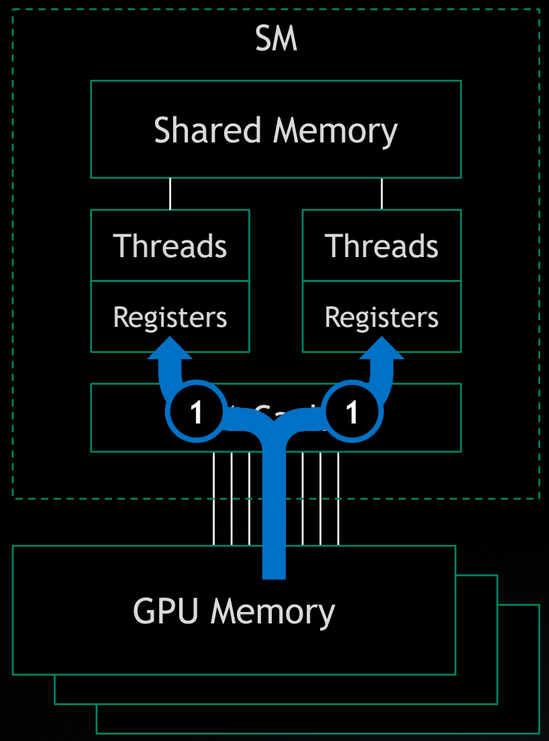




Single-Stage barriers combine back-to-back arrive & wait

Asynchronous barriers enable pipelined processing

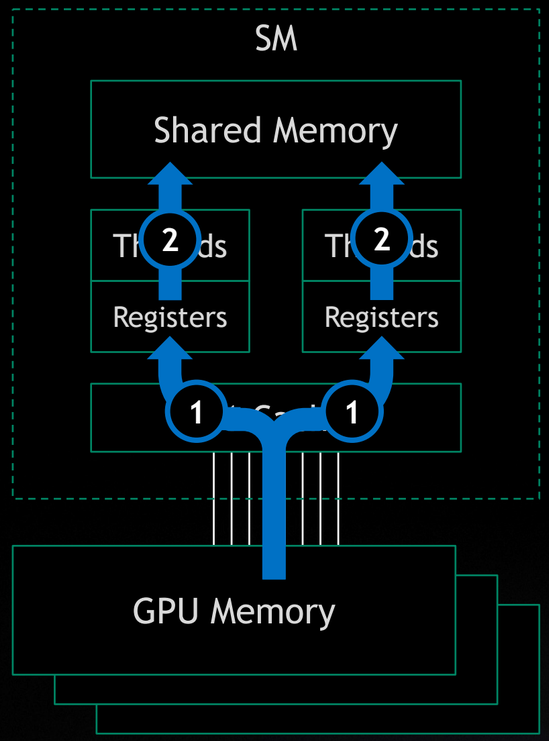
COPYING DATA INTO SHARED MEMORY



Two step copy to shared memory via registers

1. Thread loads data from GPU memory into registers

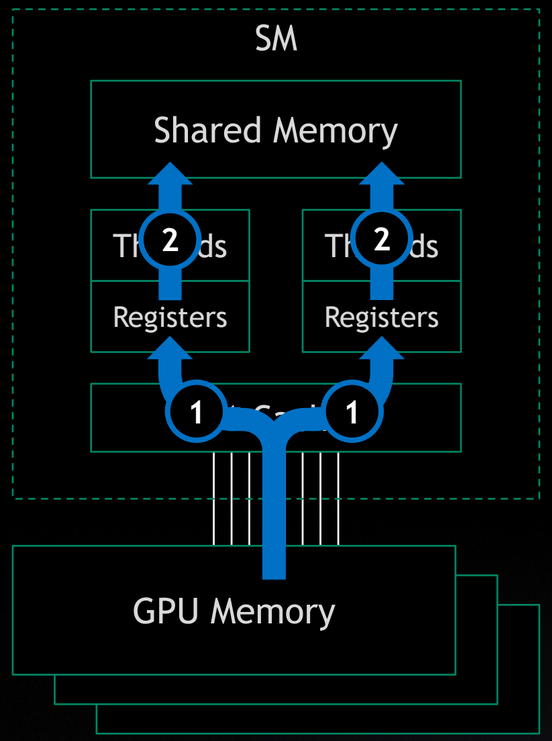
COPYING DATA INTO SHARED MEMORY



Two step copy to shared memory via registers

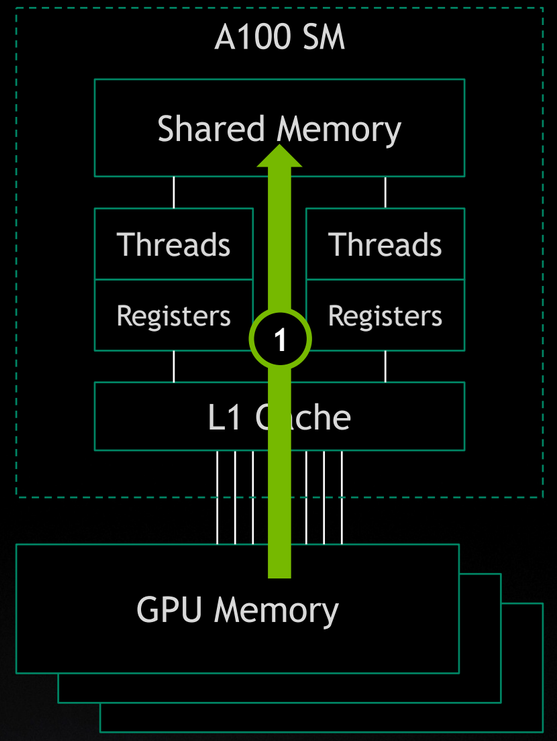
1. Thread loads data from GPU memory into registers
2. Thread stores data into SM shared memory

ASYNC MEMCOPY: DIRECT TRANSFER INTO SHARED MEMORY



Two step copy to shared memory via registers

1. Thread loads data from GPU memory into registers
2. Thread stores data into SM shared memory



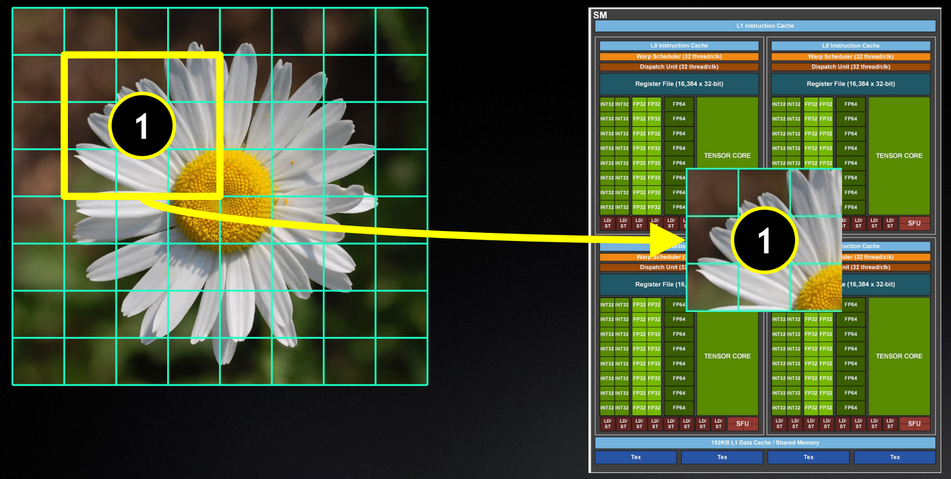
Asynchronous direct copy to shared memory

1. Direct transfer into shared memory, **bypassing** thread resources

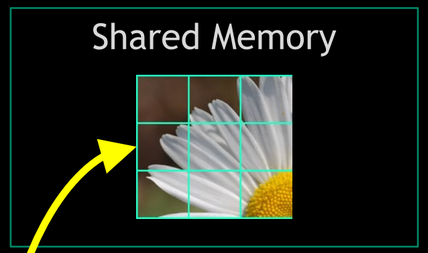
SIMPLE DATA MOVEMENT

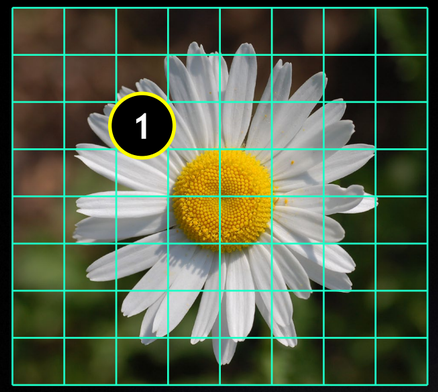
Shared Memory

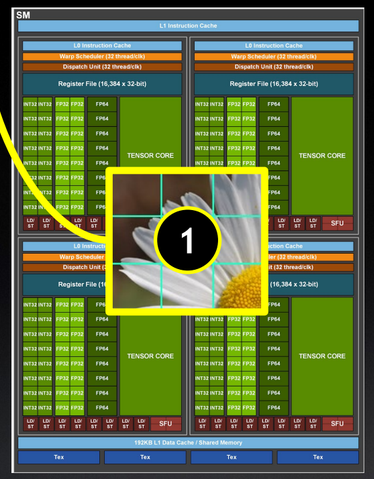
1 Load image element into registers



SIMPLE DATA MOVEMENT



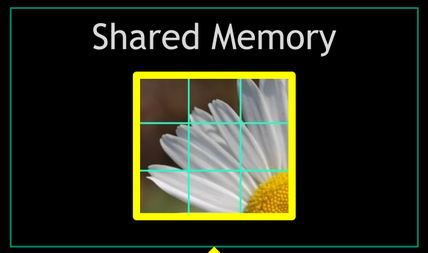


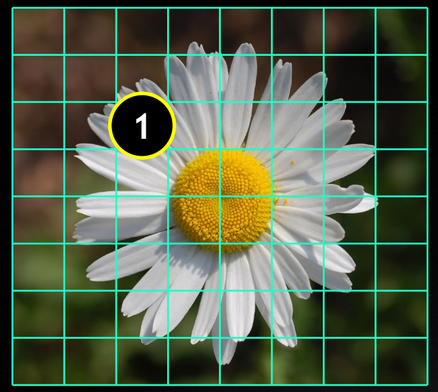


2

1. Load image element into registers
2. Store image element into shared memory

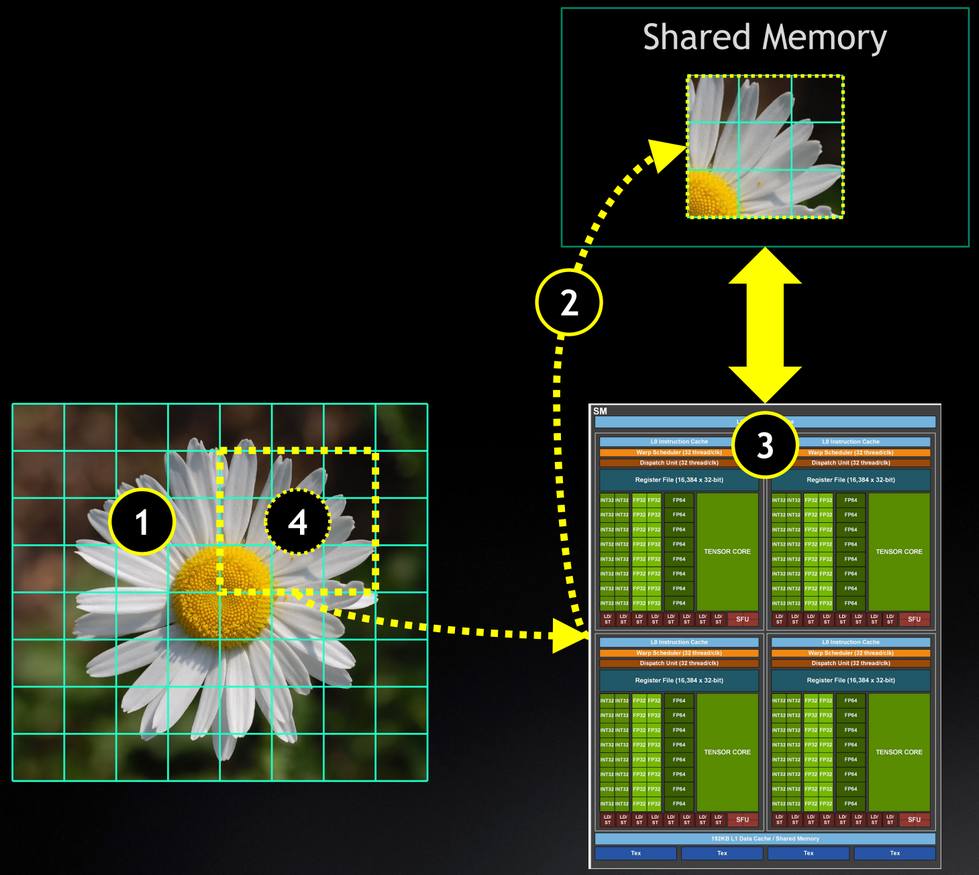
SIMPLE DATA MOVEMENT





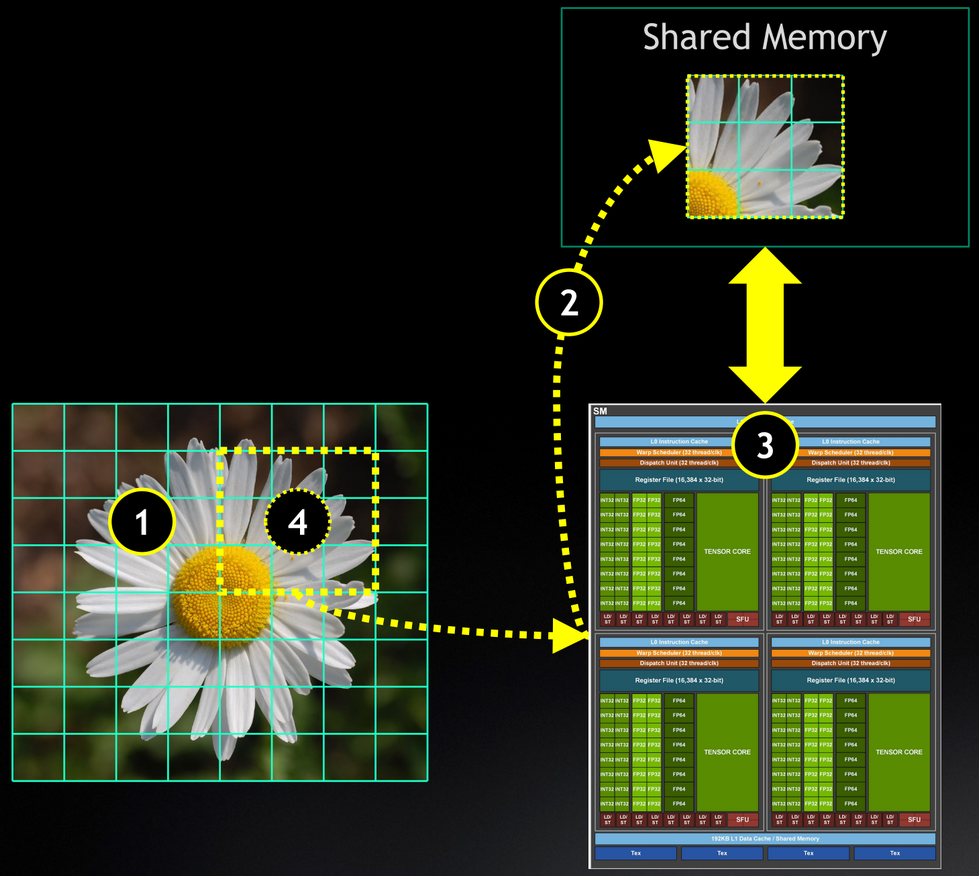
1. Load image element into registers
2. Store image element into shared memory
3. Compute using shared memory data

SIMPLE DATA MOVEMENT



1. Load image element into registers
2. Store image element into shared memory
3. Compute using shared memory data
4. Repeat for next element

SIMPLE DATA MOVEMENT

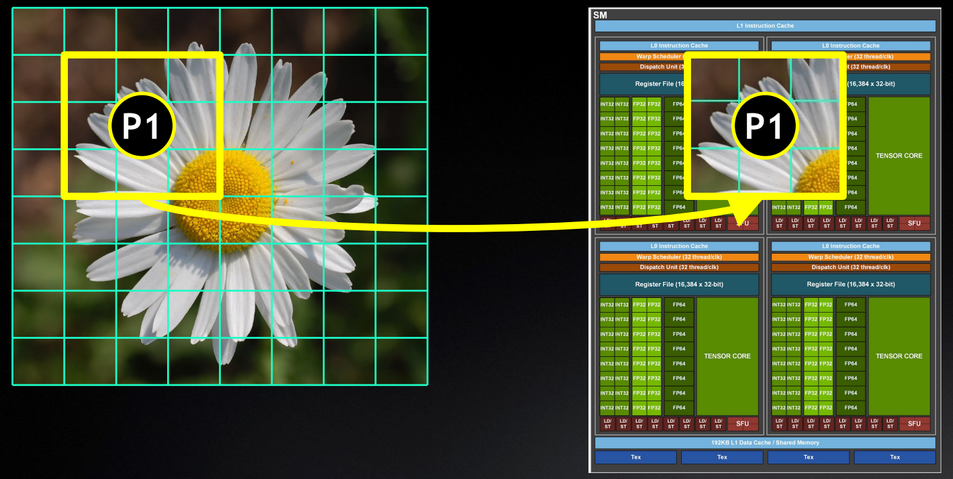



```
__shared__ float smem[ELEM_SIZE];

for( e = 0; e < NUM_ELEMS; e++ ) {
      // Load an image element into shared mem
      pixel = image[image_offset(e)];                 // Step 1
      smem[shared_offset(threadIdx.x)] = pixel;       // Step 2
      __syncthreads();                                // Step 3

      // Compute using this element
      result = compute(smem);                        // Step 4
      __syncthreads();                               // Sync & repeat
}
```

DOUBLE-BUFFERED DATA MOVEMENT

Shared Memory

P1 Prefetch initial image element into registers

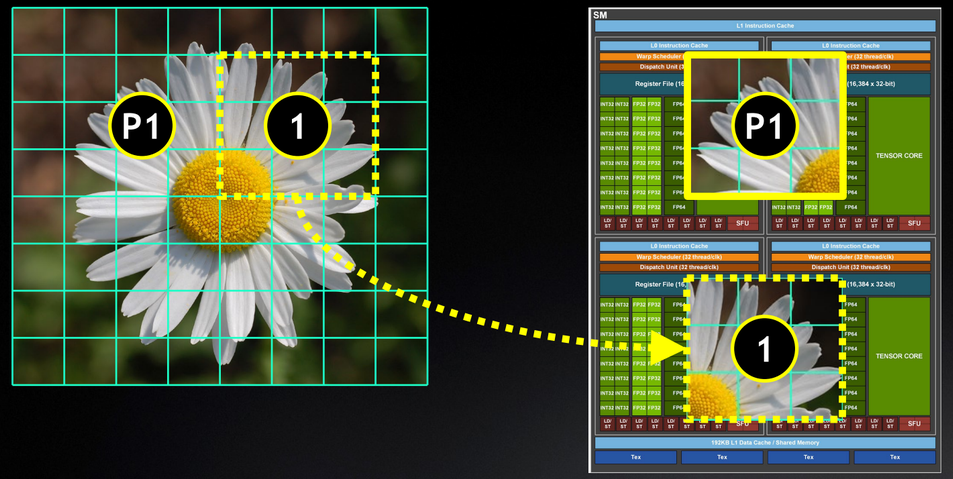


DOUBLE-BUFFERED DATA MOVEMENT

Shared Memory

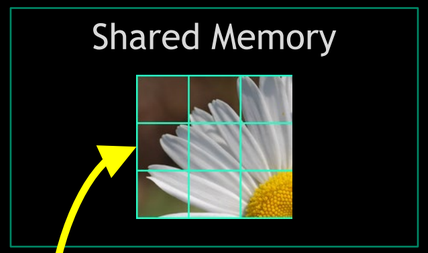
P1 Prefetch initial image element into registers

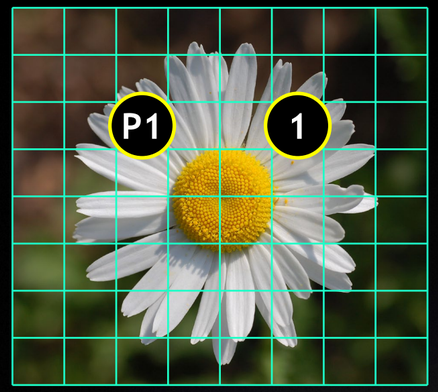
1 Prefetch **next** element into **more** registers

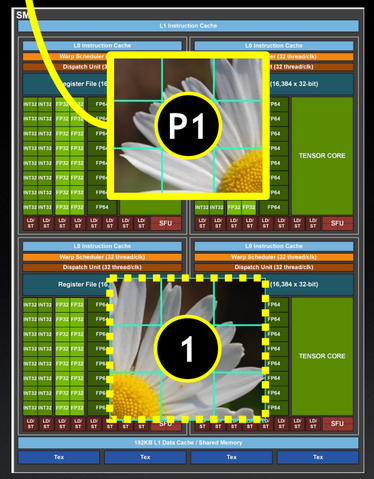


NVIDIA

DOUBLE-BUFFERED DATA MOVEMENT

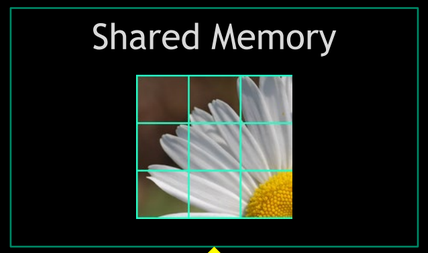


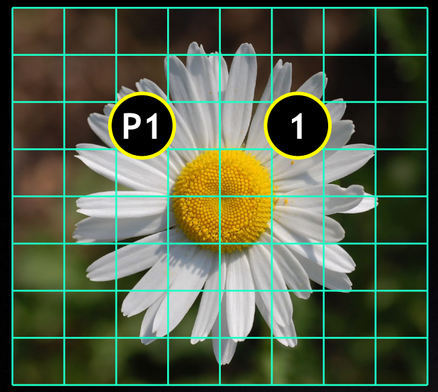


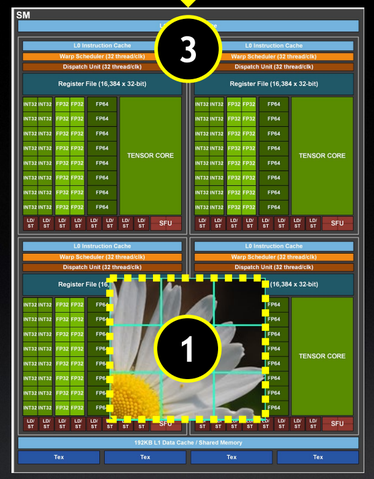


- **P1** Prefetch initial image element into registers
- **1** Prefetch **next** element into **more** registers
- **2** Store **current** element into shared memory

DOUBLE-BUFFERED DATA MOVEMENT

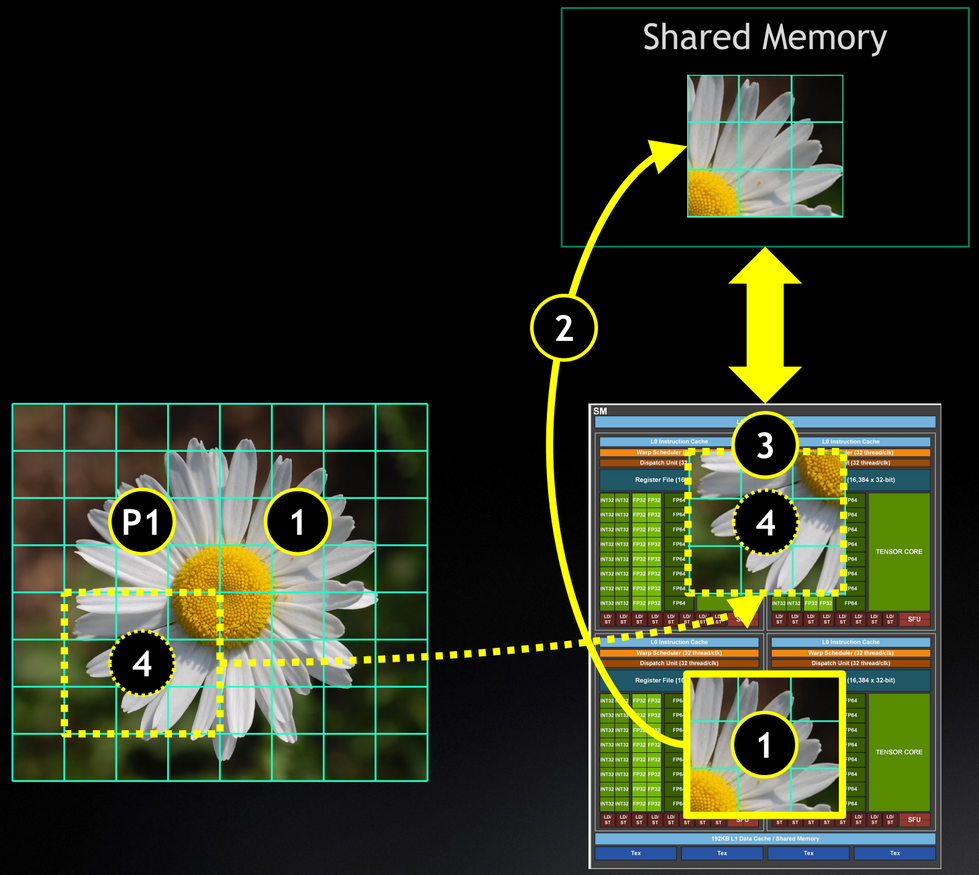






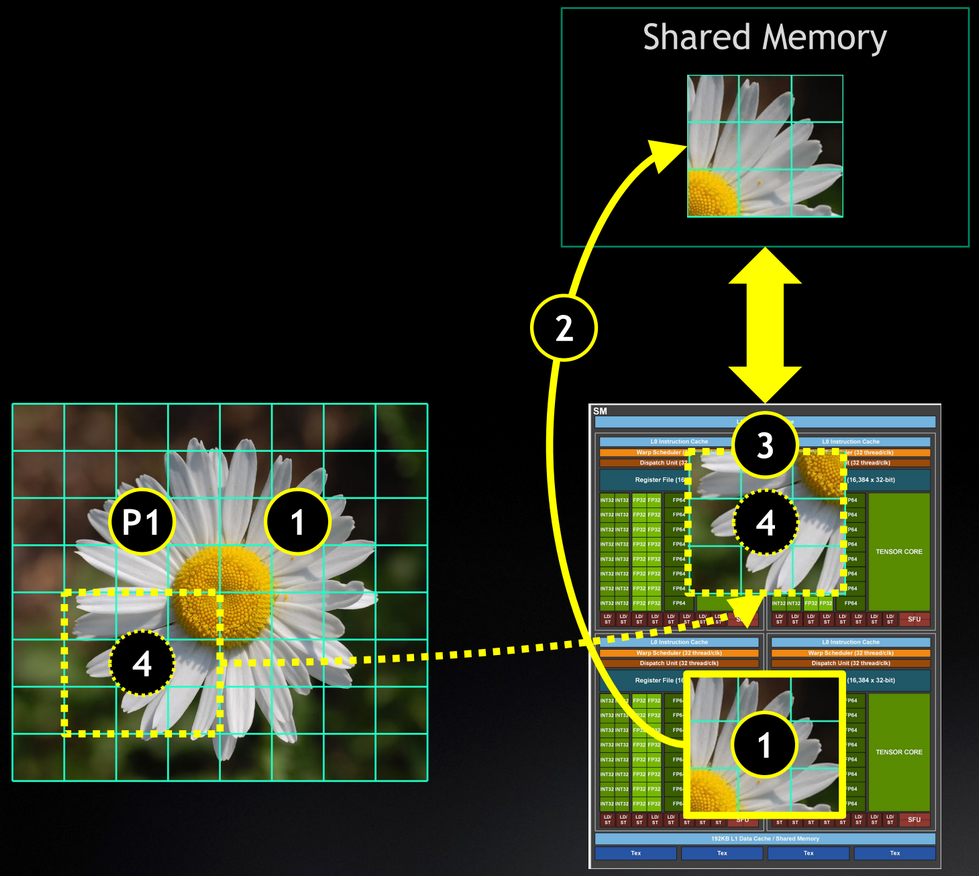
- P1 Prefetch initial image element into registers
- 1 Prefetch **next** element into **more** registers
- 2 Store **current** element into shared memory
- 3 Compute using shared memory data

DOUBLE-BUFFERED DATA MOVEMENT



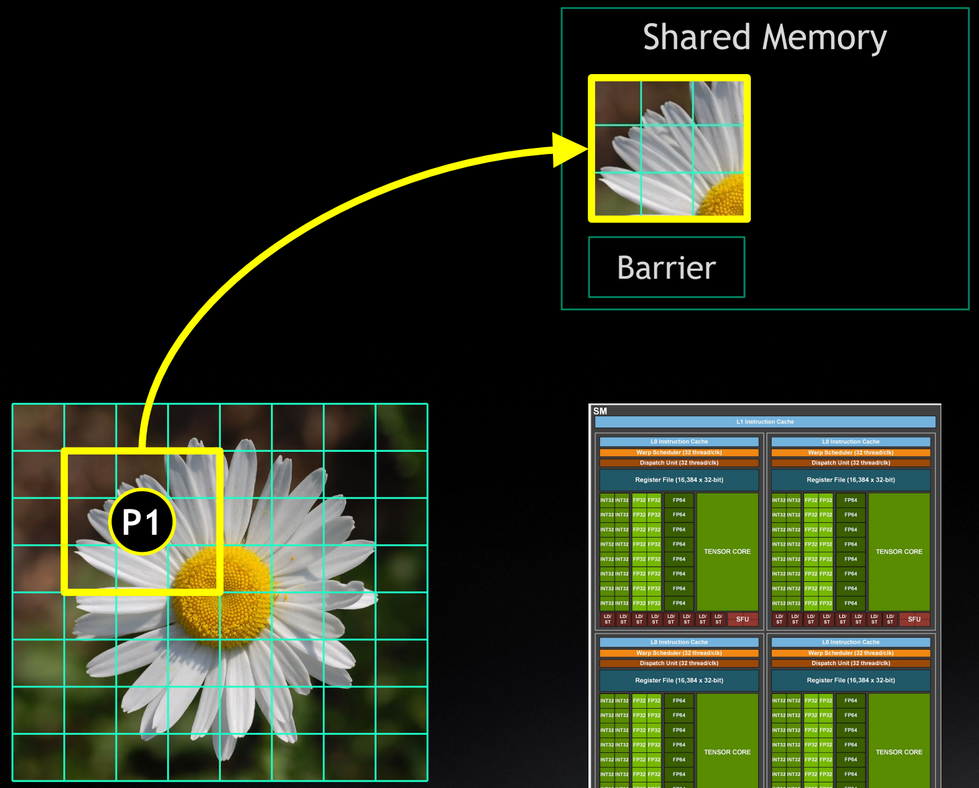
- **P1** Prefetch initial image element into registers
- **1** Prefetch **next** element into **more** registers
- **2** Store **current** element into shared memory
- **3** Compute using shared memory data
- **4** Repeat for next element

DOUBLE-BUFFERED DATA MOVEMENT



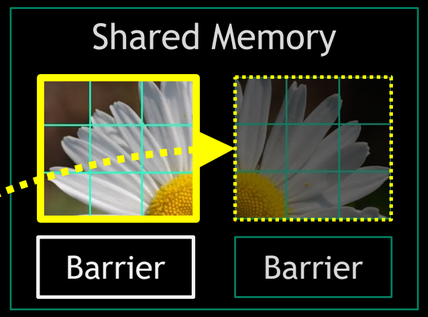

```
__shared__ float smem[ELEM_SIZE];
float pixel[2];

// Prefetch first element
pixel[0] = image[image_offset(0)];

#pragma unroll 2
for( e = 0; e < NUM_ELEMS; e++ ) {
      // Kick off load of next image
      pixel[(e+1)&1] = image[image_offset(e+1)];

      // Write prefetched data into shared mem
      smem[shared_offset()] = pixel[e&1];
      __syncthreads();

      // Compute first image while second loads
      result = compute(smem);
      __syncthreads();
}
```

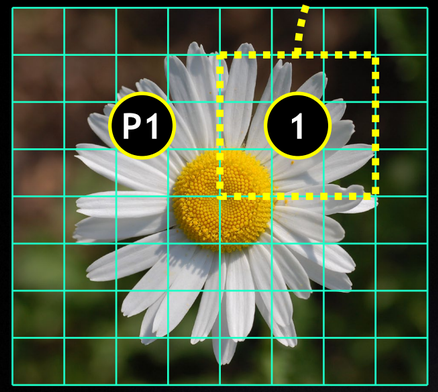
ASYNCHRONOUS DIRECT DATA MOVEMENT



P1 **Async copy** initial element into shared memory

ASYNCHRONOUS DIRECT DATA MOVEMENT

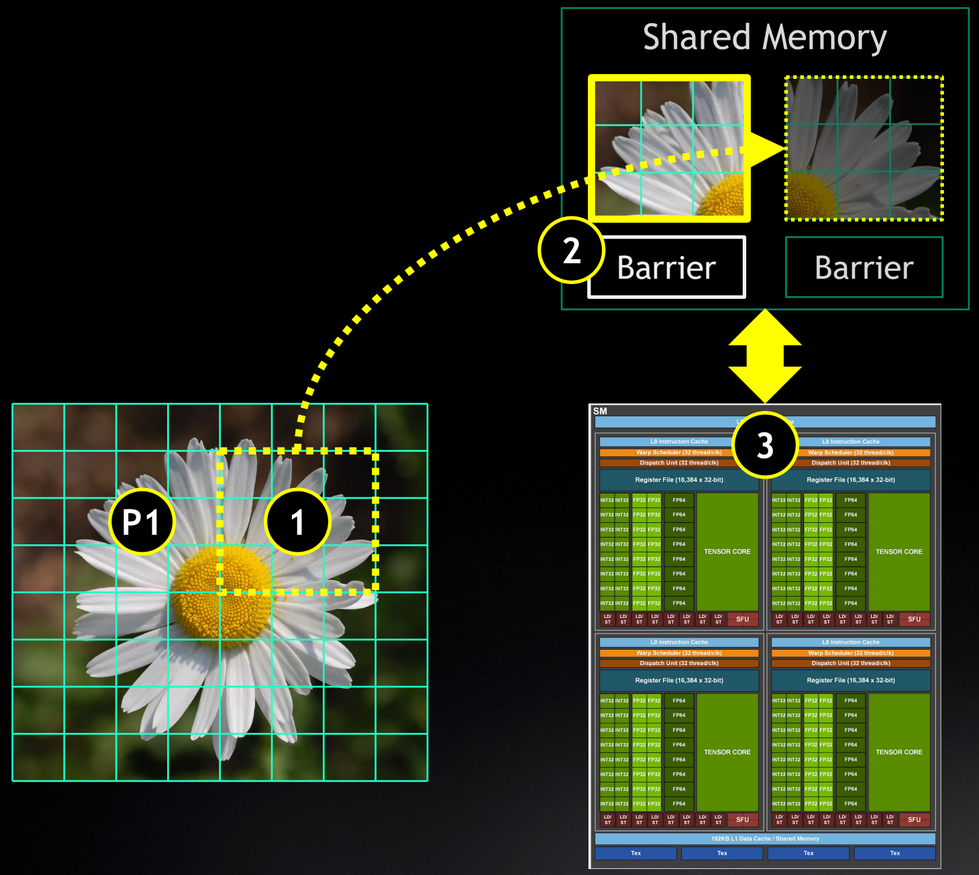




P1 **Async copy** initial element into shared memory

1 **Async copy next** element into shared memory

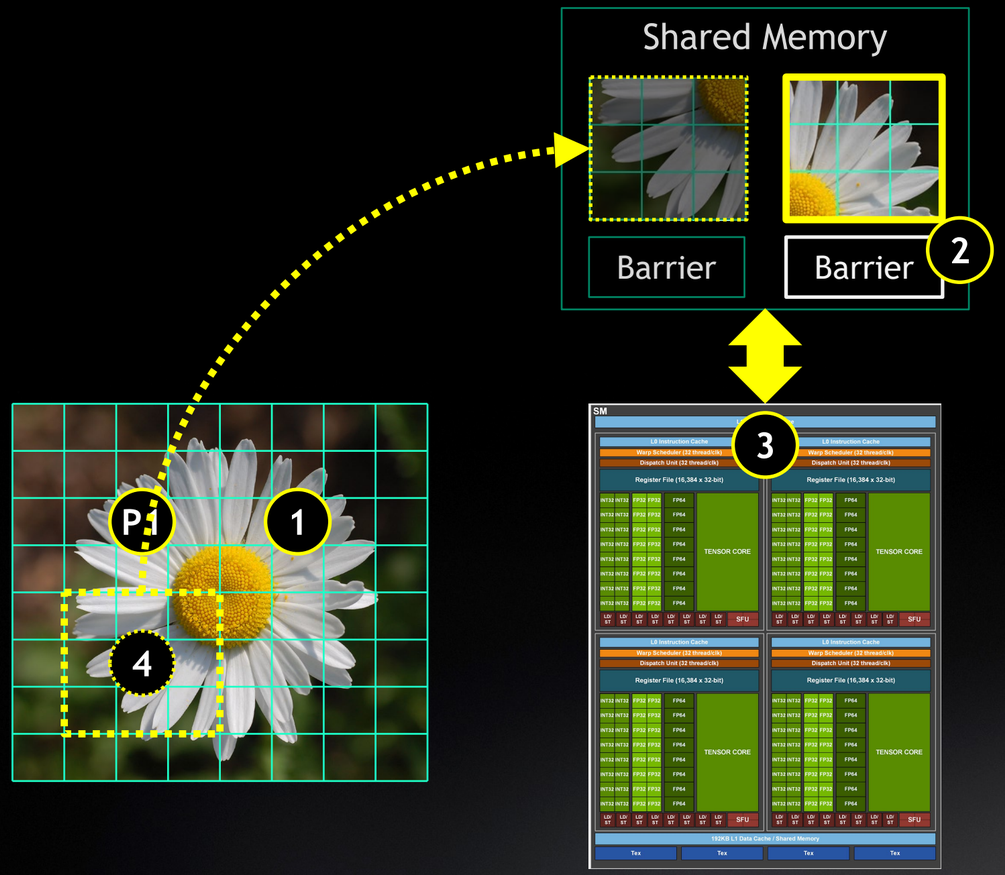
ASYNCHRONOUS DIRECT DATA MOVEMENT



- **P1** **Async copy** initial element into shared memory
- **1** **Async copy** **next** element into shared memory
- **2** Threads **synchronize** with **current** async copy
- **3** Compute using shared memory data

Async copy notifies an **asynchronous barrier** when it is done – the copy **arrives**, and threads can **wait** for it.

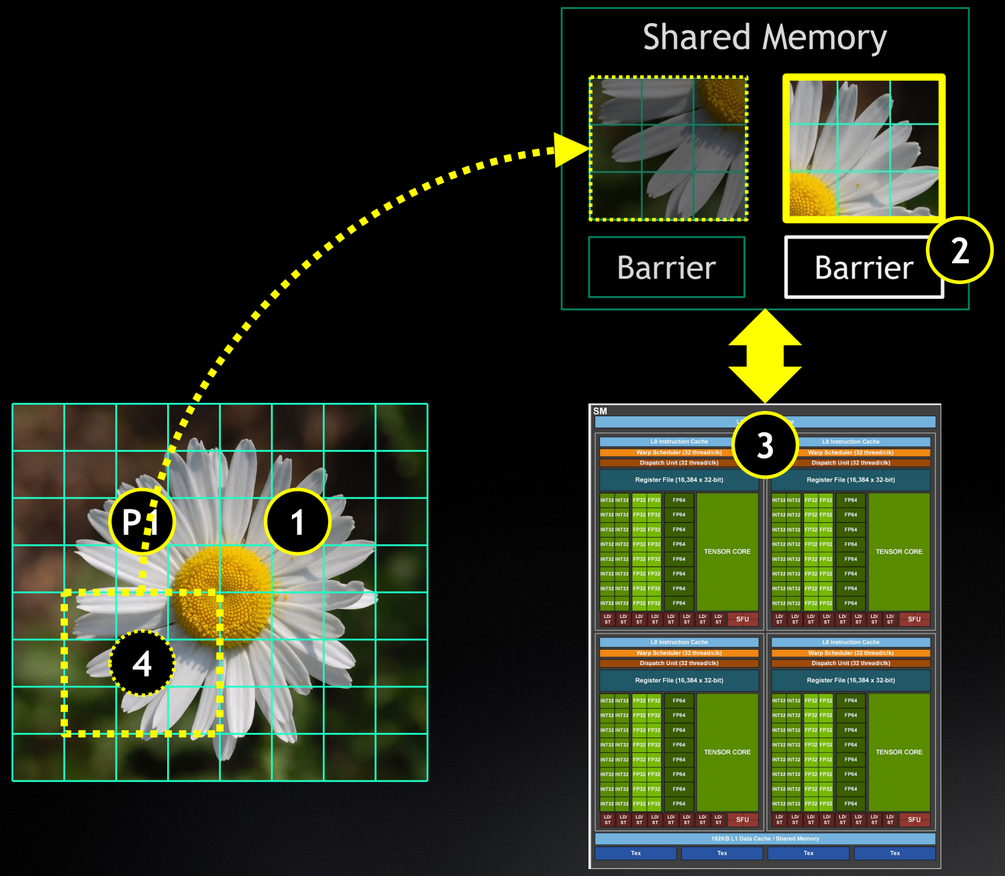
ASYNCHRONOUS DIRECT DATA MOVEMENT



- **P1** **Async copy** initial element into shared memory
- **1** **Async copy** **next** element into shared memory
- **2** Threads **synchronize** with **current** async copy
- **3** Compute using shared memory data
- **4** Repeat for next element

Async copy notifies an **asynchronous barrier** when it is done – the copy **arrives**, and threads can **wait** for it.

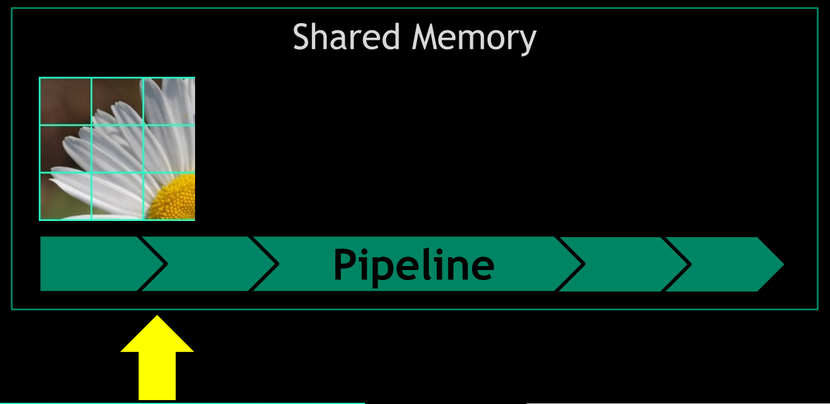
ASYNCHRONOUS DIRECT DATA MOVEMENT




```
__shared__ float smem[2][ELEM_SIZE];
cuda::barrier barrier[2];
buf_id = 0;

// Kick off initial copy – it will arrive on barrier
memcpy_async(&smem[buf_id][shared_offset()],
             &image[image_offset(0)], size,
             barrier[buf_id]);

for( e = 1; e < NUM_ELEMS; e++ ) {
     // Start by issuing copy of next chunk
     memcpy_async(&smem[!buf_id][shared_offset()],
                  &image[image_offset(e)], size,
                  barrier[!buf_id]);

     // Sync on current chunk then compute
     barrier[buf_id].arrive_and_wait();
     result = compute(smem[buf_id]);
     buf_id = !buf_id;     // Flip buffers
}
```

ASYNCHRONOUS COPY PIPELINES

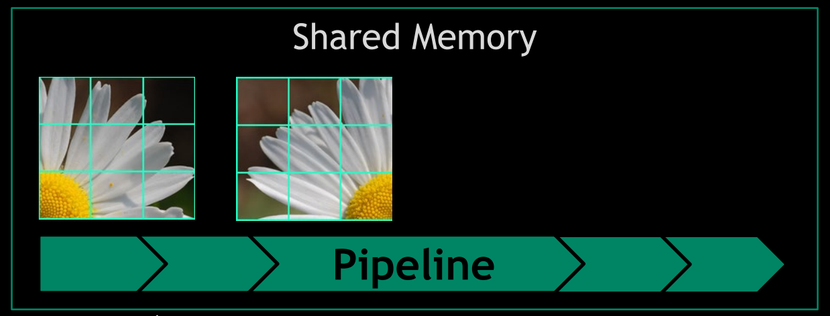
Prefetch multiple images in a continuous stream

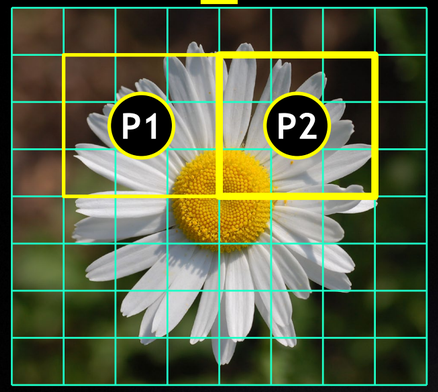


P1 **Async copy** multiple elements into shared memory

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream

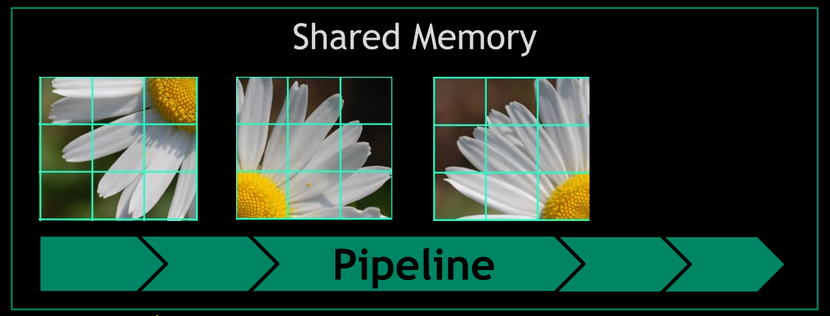


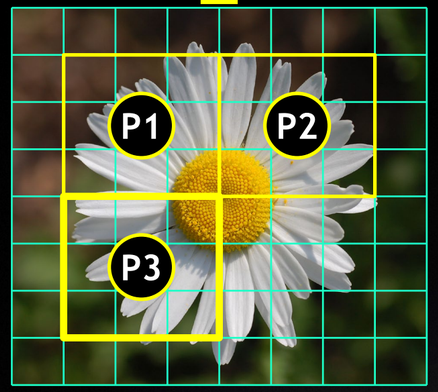


P1 P2 **Async copy** multiple elements into shared memory

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream



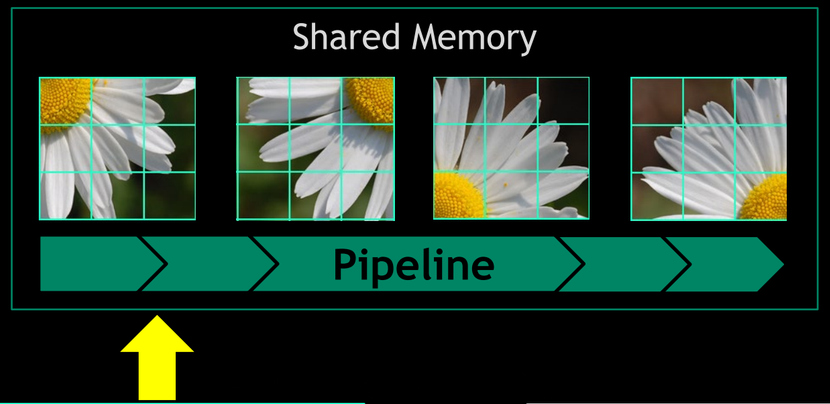


P1 P2 P3 **Async copy** multiple elements into shared memory

NVIDIA

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream



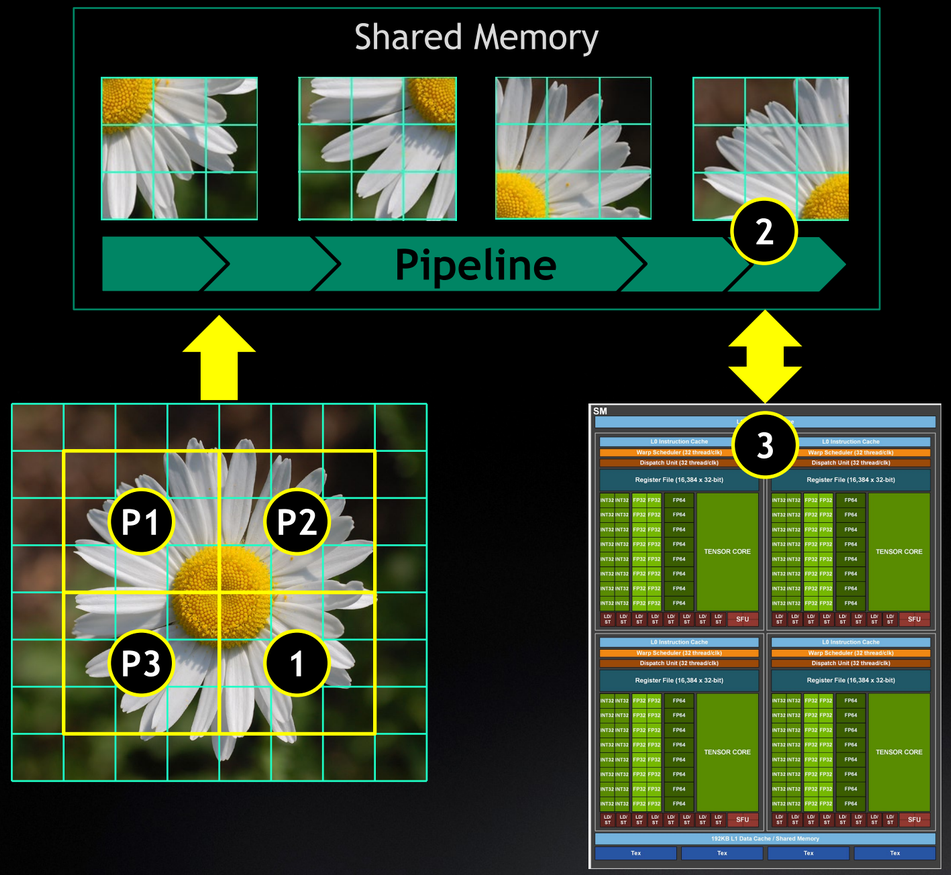
P1 P2 P3 **Async copy** multiple elements into shared memory

1 **Async copy next** element into shared memory

NVIDIA.

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream



P1 P2 P3 **Async copy** multiple elements into shared memory

1 **Async copy next** element into shared memory

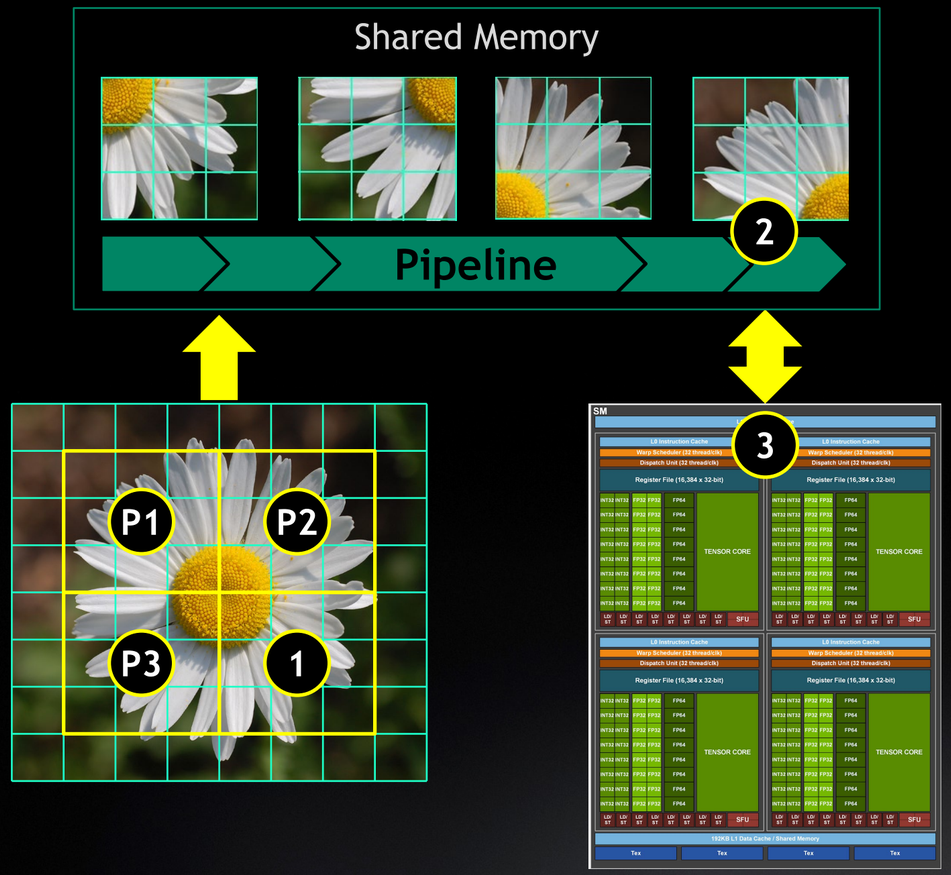
2 Threads **synchronize** with **oldest** pipelined copy

3 Compute using shared memory data

4 Repeat for next element

ASYNCHRONOUS COPY PIPELINES

Prefetch multiple images in a continuous stream



Async Copy using Pipeline vs. Barrier

Allows batching of multiple copy operations into a single transaction

Many in-flight transactions, completing in FIFO order

Fastest possible synchronization performance

For full details see the developer blog: https://developer.nvidia.com/blog/controlling-data-movement-to-boost-performance-on-ampere-architecture/

ISO C++ == Language + Standard Library

ISO C++ == Language + Standard Library

CUDA C++ == Language

libcu++ : THE CUDA C++ STANDARD LIBRARY

ISO C++ == Language + Standard Library

CUDA C++ == Language + **libcu++**

Strictly conforming to ISO C++, plus conforming extensions

Opt-in, Heterogeneous, Incremental

cuda::std::

Opt-in	Does not interfere with or replace your host standard library
Heterogeneous	Copyable/Movable objects can migrate between host & device Host & Device can call all member functions Host & Device can concurrently use synchronization primitives*
Incremental	A subset of the standard library today Each release adds more functionality

*Synchronization primitives must be in managed memory and be declared with **`cuda::std::thread_scope_system`**

libcu++ NAMESPACE HIERARCHY

```
// ISO C++, __host__ only
#include <atomic>
std::atomic<int> x;

// CUDA C++, __host__ __device__
// Strictly conforming to the C++ Standard
#include <cuda/std/atomic>
cuda::std::atomic<int> x;

// CUDA C++, __host__ __device__
// Conforming extensions to the C++ Standard
#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;
```

cuda::std::

libcu++, the NVIDIA C++ Standard Library,
is the C++ Standard Library for your entire system

libcu++ is NVIDIA's variant of **LLVM's libc++**

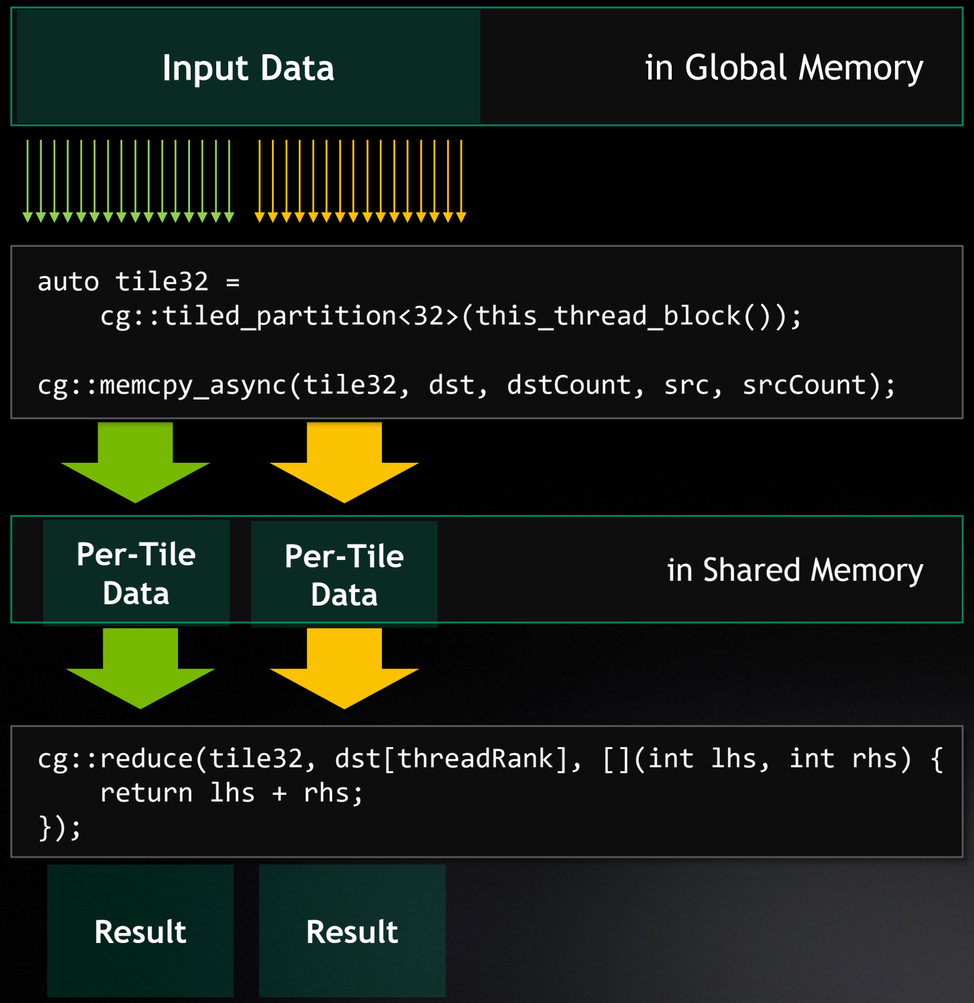
It is opt-in, heterogeneous and incremental

Available on GitHub at: https://nvidia.github.io/libcudacxx/

Open source, licensed under the **Apache License 2.0 with LLVM Exceptions**

COOPERATIVE GROUPS

Cooperative Groups Features Work On All GPU Architectures (incl. Kepler)



Cooperative Groups Updates

No longer requires separate compilation

30% faster grid synchronization

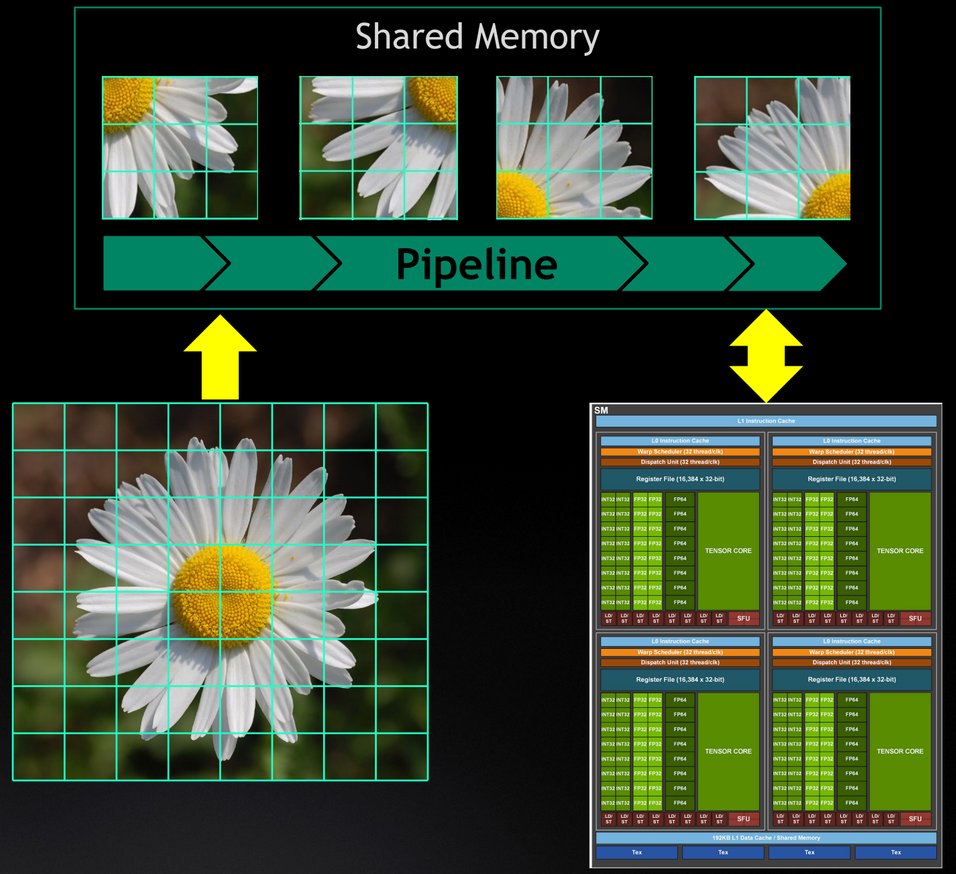
New platforms Support (Windows and Linux + MPS)

Can now capture cooperative launches in a CUDA graph

cg::reduce also accepts C++ lambda as reduction operation

COOPERATIVE GROUPS MAPS NATURALLY TO PIPELINES

```
constexpr int N = 4;                    // 4-stage pipeline
__shared__ float smem[N][NUM_ELEMS];
__shared__ cuda::pipeline_shared_state<thread_scope_block, N> ps;
auto group = cooperative_groups::this_thread_block();
auto pipe = cuda::make_pipeline(group, &ps);

for ( e = f = 0 ; e < NUM_ELEMS ; e++ ) {
      // Fetch-ahead empty buffers
      for ( ; f < NUM_ELEMS && f < (e+N) ; f++ ) {
            // Begin pipeline push
            pipe.producer_acquire();
            cuda::memcpy_async(group, &smem[f % N],
                        &global1[f * group.size()],
                        sizeof(float) * group.size(), pipe);
            pipe.producer_commit();
            // End pipeline push
      }

      // Pop the oldest copy off the pipeline
      pipe.consumer_wait();         // Begin pipeline pop
      compute(smem[e % N]);
      pipe.consumer_release();      // End pipeline pop
}
```



NVIDIA

MULTI-WARP COOPERATIVE GROUPS

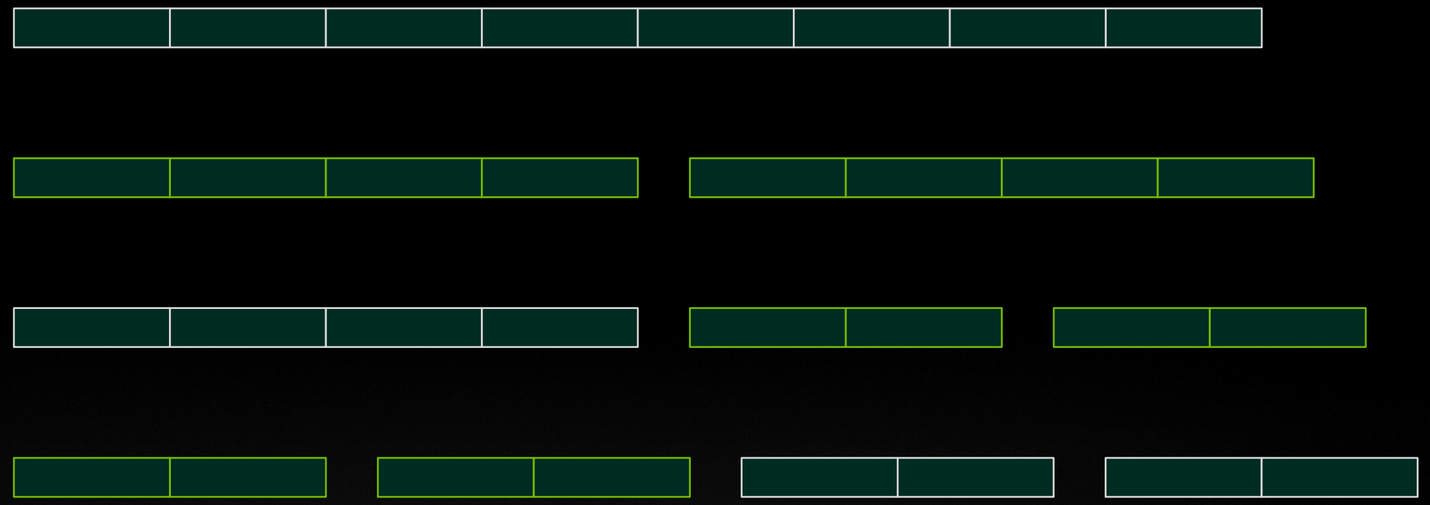
Experimental in CUDA 11.1

Original group of 8 warps = 256 threads

2 groups of 128 threads

1 group of 128 threads + 2 groups of 64 threads

4 groups of 64 threads



Create groups with any power-of-2 number of threads, even spanning multiple warps

Sync and perform collective operations within these groups

MULTI-WARP COOPERATIVE GROUPS

Caller provides memory for barriers & collectives

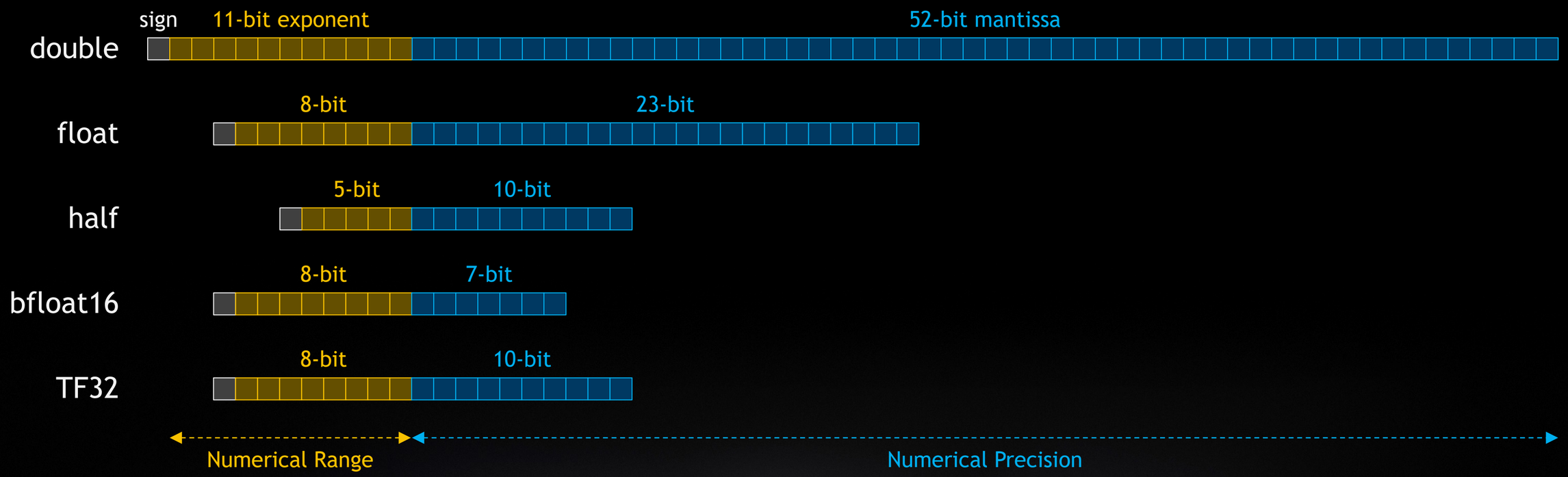
`cg::experimental::block_tile_memory`
Declares scratch memory for use in communication between tiles, based on the number of threads in your block.

`cg::this_thread_block(…& scratch)`
Takes in scratch memory and associates it with the resulting group handle.

Scratch memory can be shared or global memory (shared is higher performance)

```
// in cg::experimental
#define _CG_ABI_EXPERIMENTAL
#include <cooperative_groups.h>
#include <cooperative_groups/reduce.h>

namespace cg = cooperative_groups;
__shared__ cg::experimental::block_tile_memory
    < sizeof(float), BlockSize > scratch;

cg::experimental::thread_block cta =
                          this_thread_block(scratch);
auto tile = tiled_partition<128>(cta);

cg::reduce(g, dst[threadRank], [](int lhs, int rhs) {
    return lhs + rhs;
});
```

FLOATING POINT FORMATS & PRECISION



$$value = (-1)^{sign} \times 2^{exponent} \times (1 + mantissa)$$

NEW FLOATING POINT FORMATS: BF16 & TF32

Both Match fp32 8-bit Exponent: Covers The Same Range of Values

bfloat16

Available in CUDA C++ as nv_bfloat16 numerical type

Full CUDA C++ numerical type – `#include <cuda_fp16.h>`

Can use in both host & device code, and in templated functions*

TF32

Tensor Core math mode for single-precision training

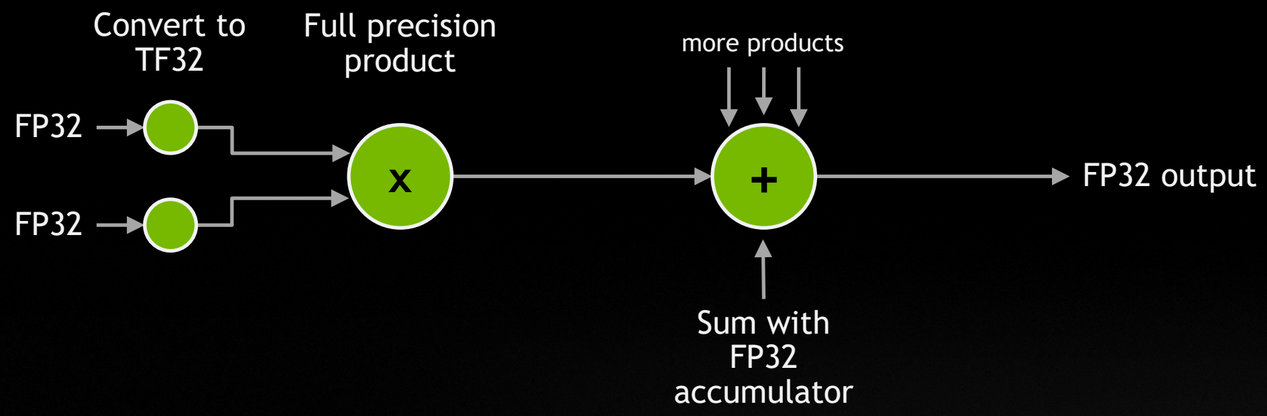
Not a numerical type – tensor core inputs are rounded to TF32

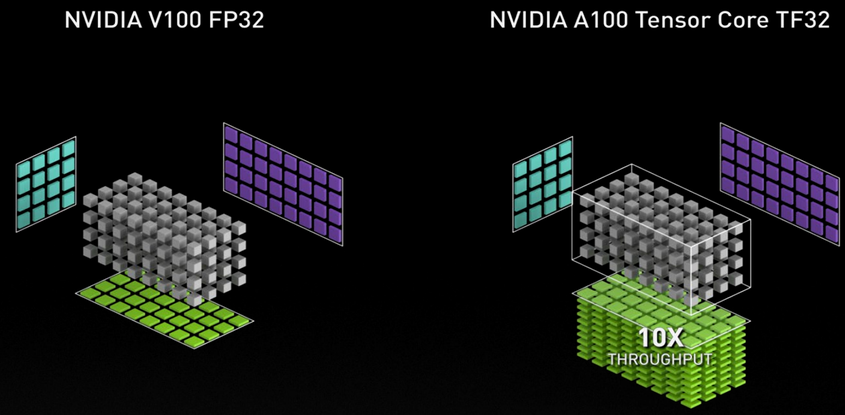
CUDA C++ programs use `float` (fp32) throughout

*(similar to CUDA's IEEE-FP16 "`half`" type)

TENSOR FLOAT 32 - TENSOR CORE MODE

A100 Tensor Core Input Precision
All Internal Operations Maintain Full FP32 Precision

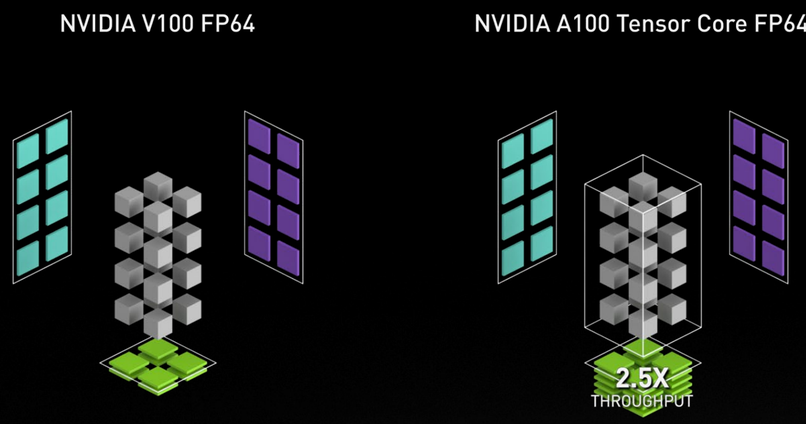




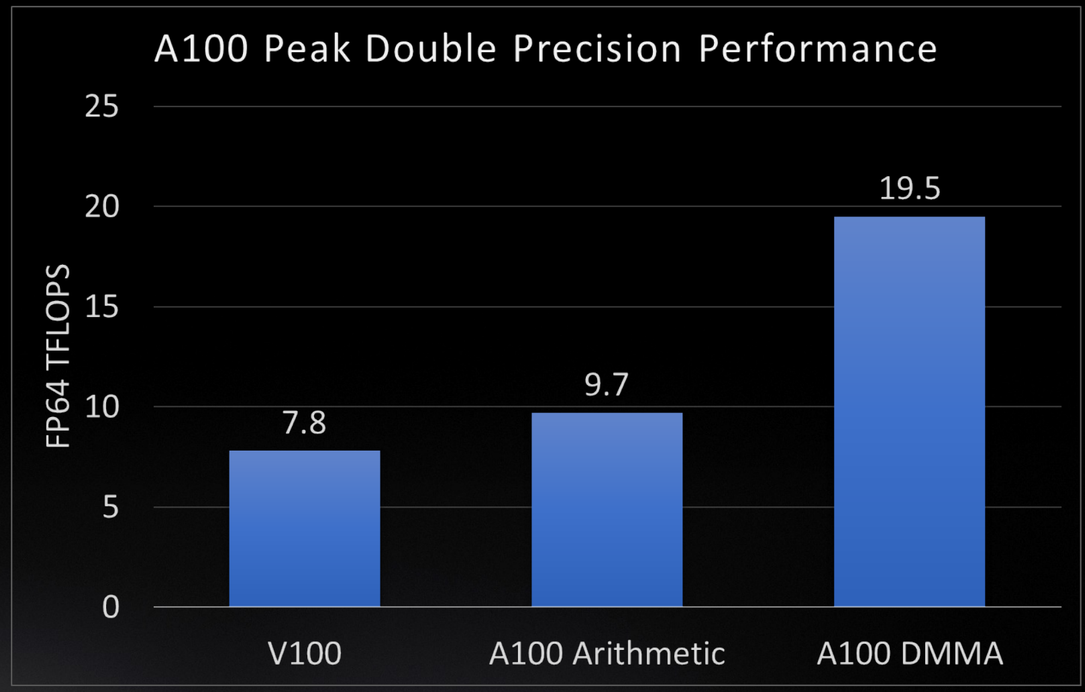
TF32 MMA Dimensions: m,n,k = 16x8x8

A100 INTRODUCES DOUBLE PRECISION TENSOR CORES

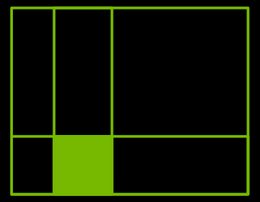
All A100 Tensor Core Internal Operations Maintain Full FP64 Precision



DMMA Dimensions: m,n,k = 8x8x4

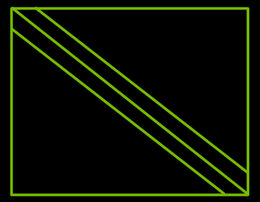


A100 GPU ACCELERATED MATH LIBRARIES IN CUDA 11.0



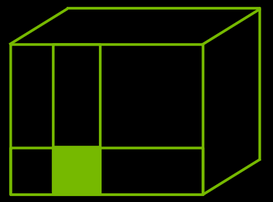
cuBLAS

BF16, TF32 and FP64 Tensor Cores



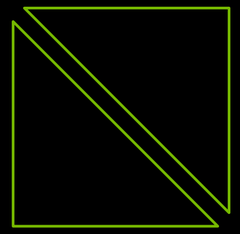
cuSPARSE

Increased memory BW, Shared Memory & L2



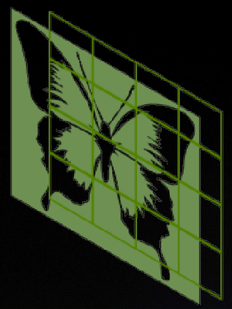
cuTENSOR

BF16, TF32 and FP64 Tensor Cores



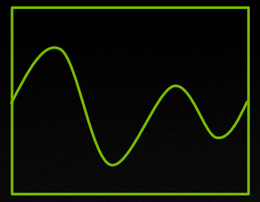
cuSOLVER

BF16, TF32 and FP64 Tensor Cores



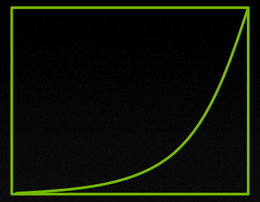
nvJPEG

Hardware Decoder



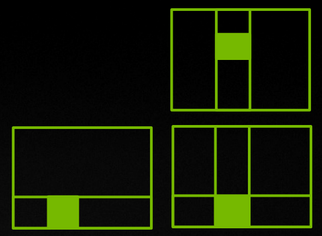
cuFFT

BF16, TF32 and FP64 Tensor Cores



CUDA Math API

Increased memory BW, Shared Memory & L2

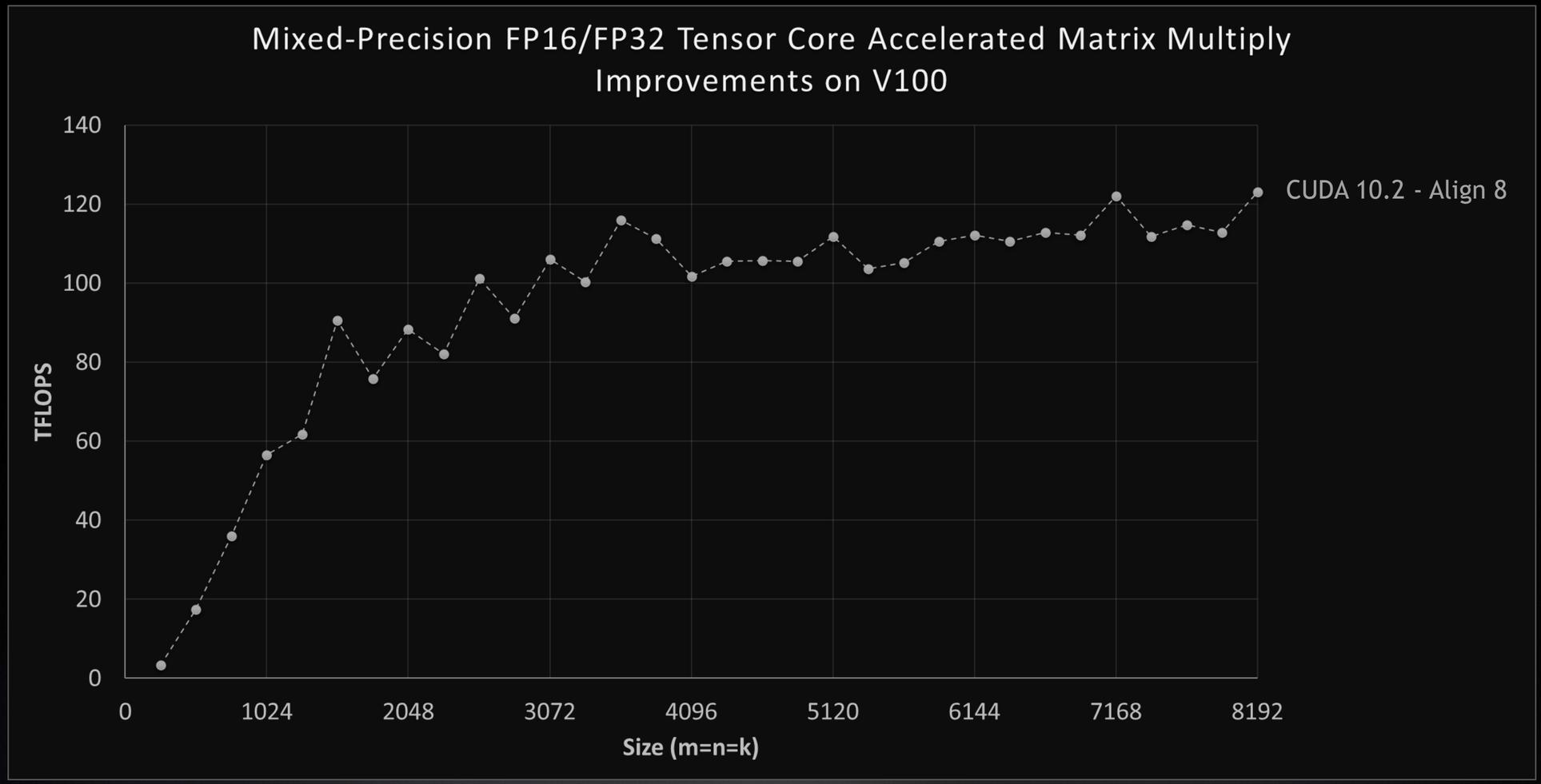


CUTLASS

BF16 & TF32 Support

cuBLAS

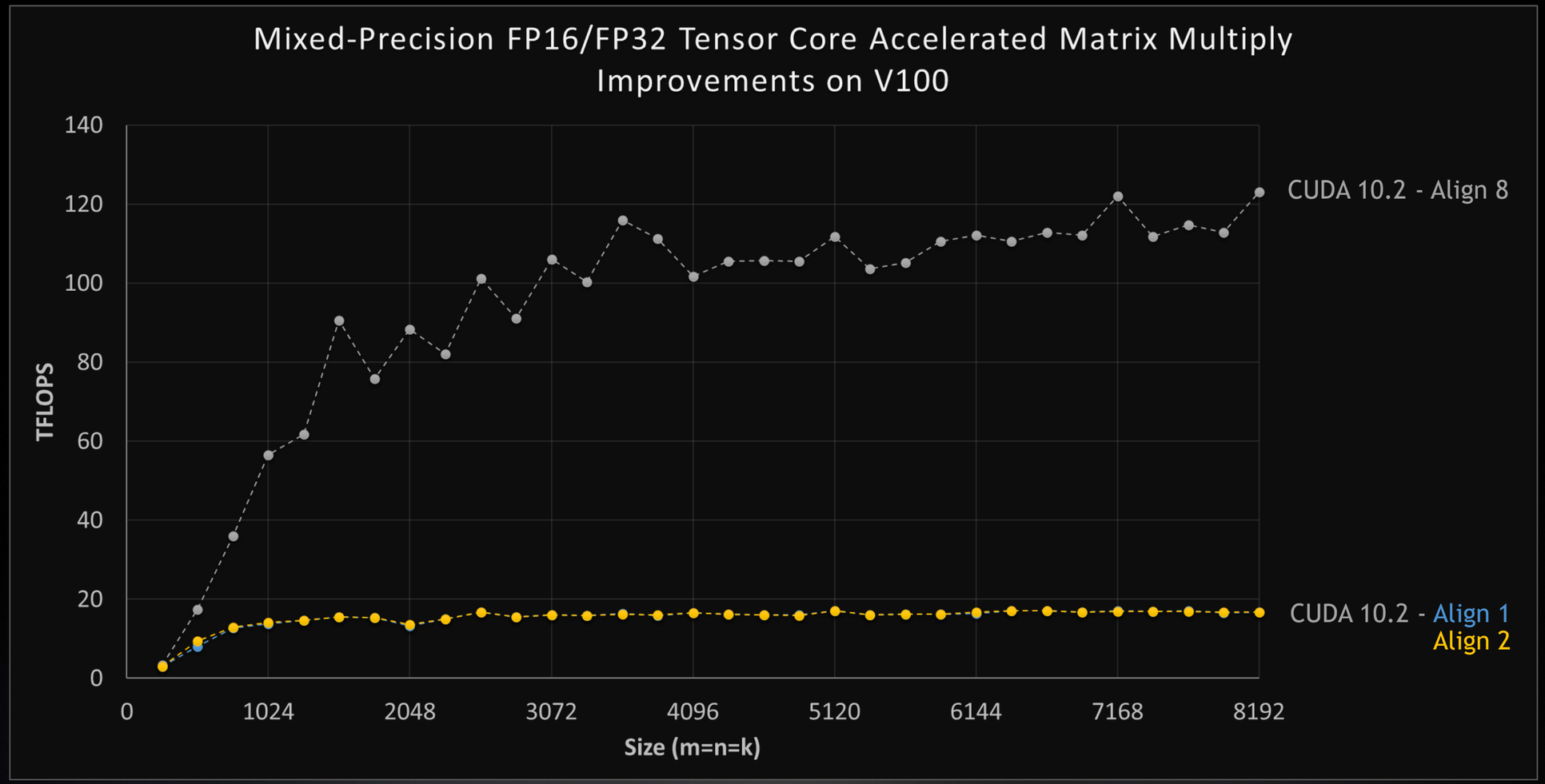
Eliminating Alignment Requirements To Activate Tensor Cores for MMA



AlignN means alignment to 16-bit multiplies of N. For example, align8 are problems aligned to 128bits or 16 bytes.

cuBLAS

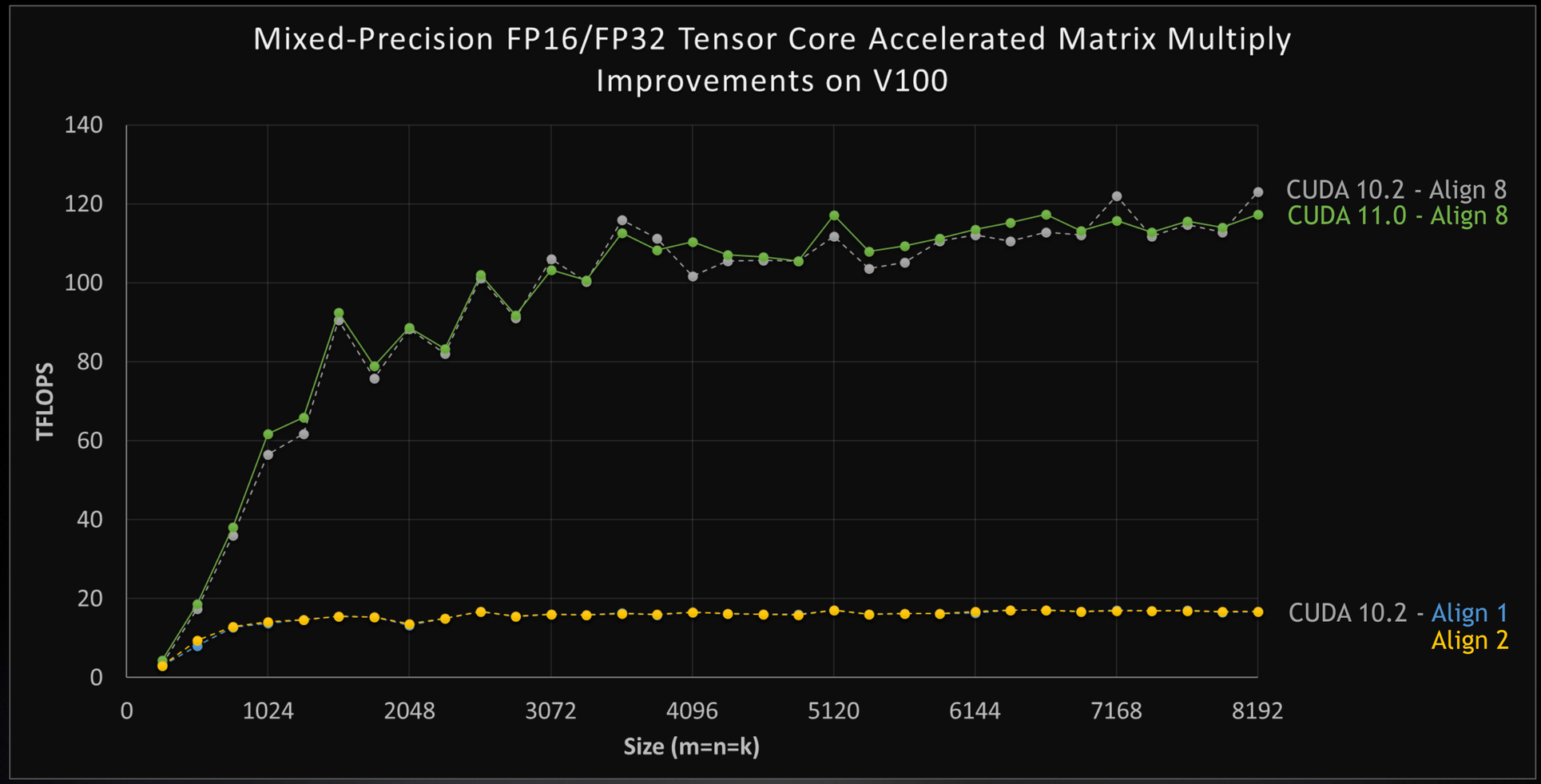
Eliminating Alignment Requirements To Activate Tensor Cores for MMA



AlignN means alignment to 16-bit multiplies of N. For example, align8 are problems aligned to 128bits or 16 bytes.

cuBLAS

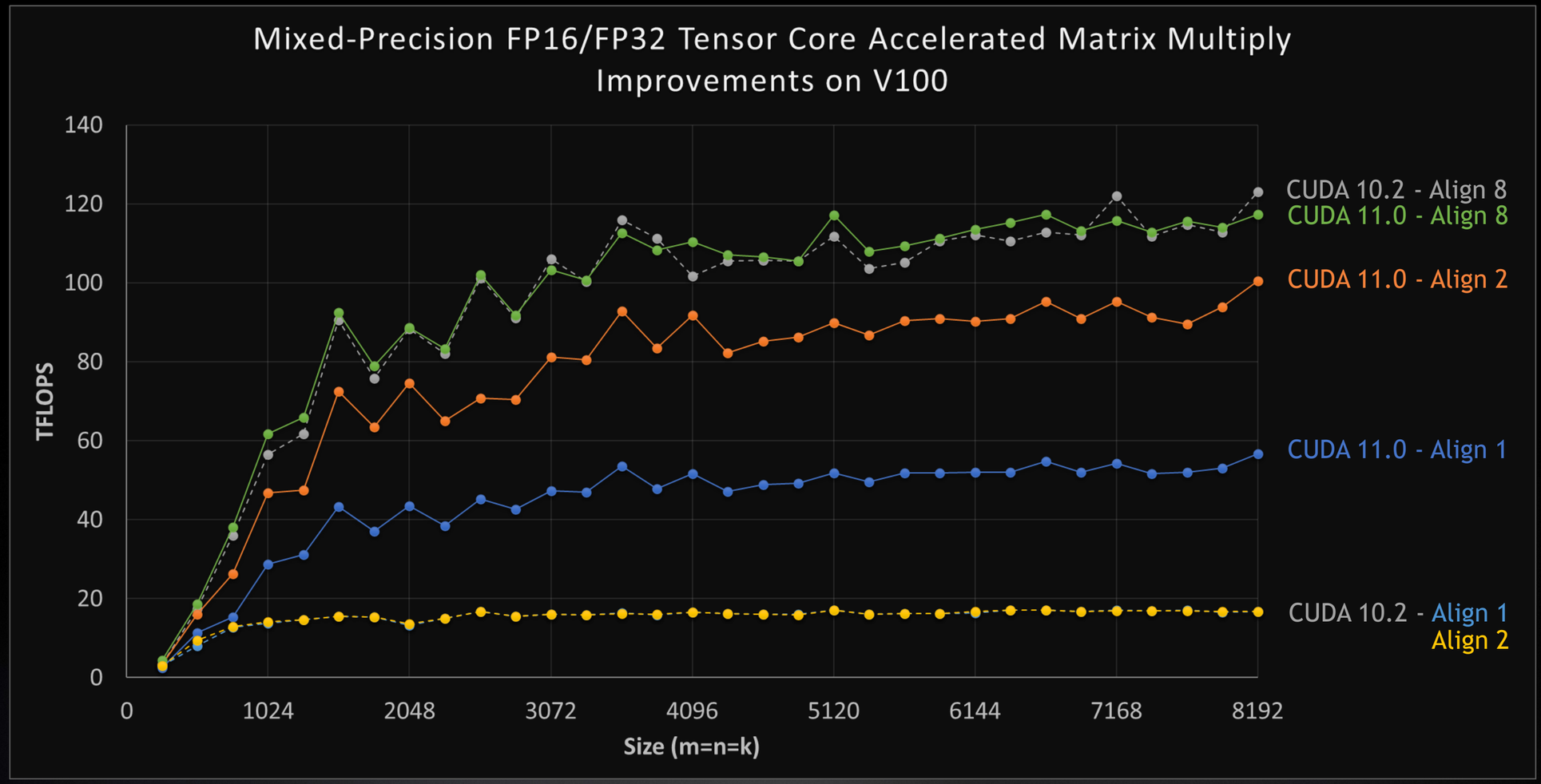
Eliminating Alignment Requirements To Activate Tensor Cores for MMA



AlignN means alignment to 16-bit multiplies of N. For example, align8 are problems aligned to 128bits or 16 bytes.

cuBLAS

Eliminating Alignment Requirements To Activate Tensor Cores for MMA



AlignN means alignment to 16-bit multiplies of N. For example, align8 are problems aligned to 128bits or 16 bytes.

MATH LIBRARY DEVICE EXTENSIONS

Introducing cuFFTDx: Device Extension

Available in Math Library EA Program

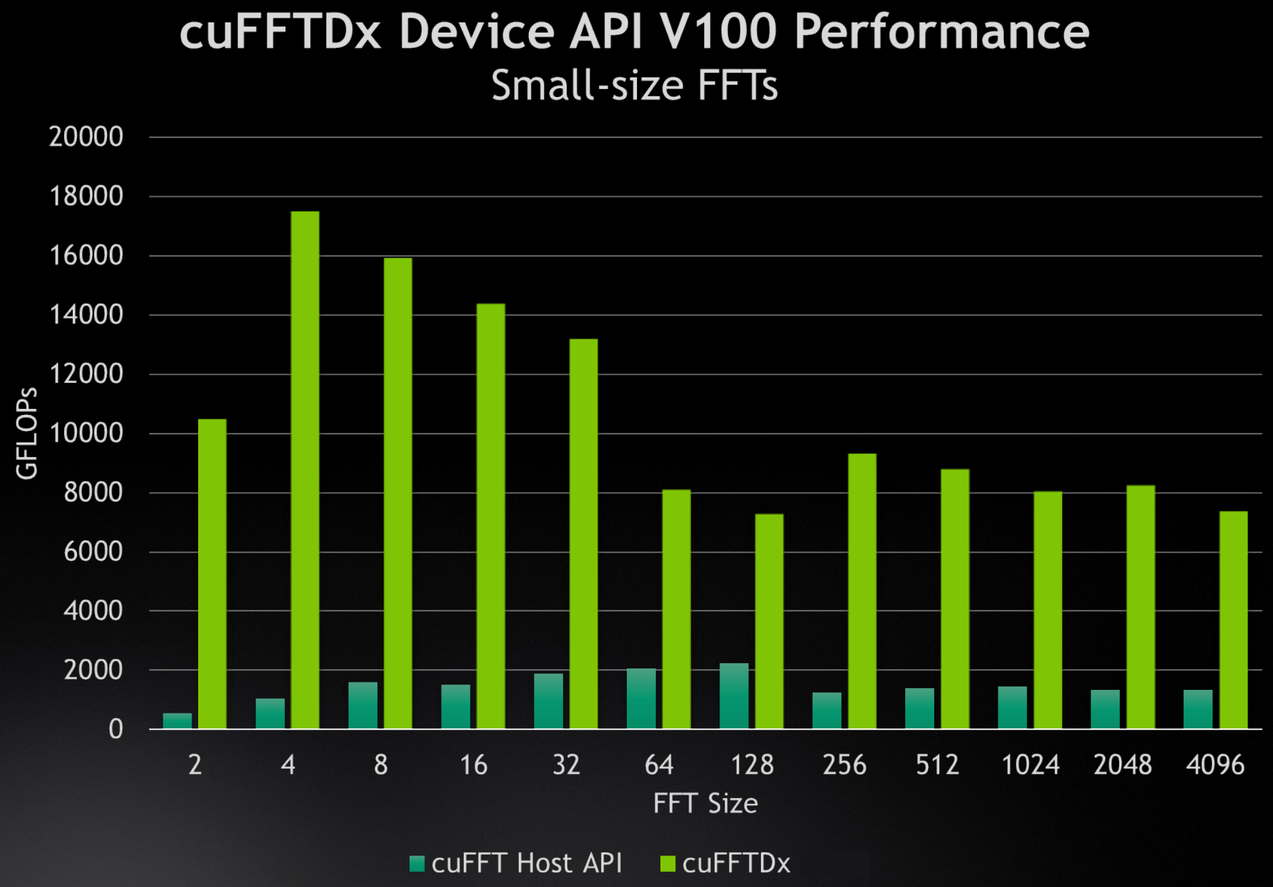
Device callable library

Retain and reuse on-chip data

Inline FFTs in user kernels

Combine multiple FFT operations

https://developer.nvidia.com/CUDAMathLibraryEA



GPU PROGRAMMING IN 2020 AND BEYOND

Math Libraries | Standard Languages | Directives | CUDA

```
std::transform(par, x, x+n, y, y,
    [=](float x, float y) {
        return y + a*x;
});
```

```
do concurrent (i = 1:n)
   y(i) = y(i) + a*x(i)
enddo
```

GPU Accelerated ISO C++ and Fortran

```
#pragma acc data copy(x,y)
{

...

std::transform(par, x, x+n, y, y,
    [=](float x, float y) {
        return y + a*x;
});

...

}
```

Incremental Performance Optimization with Directives

```
__global__
void saxpy(int n, float a,
           float *x, float *y) {
  int i = blockIdx.x*blockDim.x +
          threadIdx.x;
  if (i < n) y[i] += a*x[i];
}

int main(void) {
  cudaMallocManaged(&x, ...);
  cudaMallocManaged(&y, ...);
  ...
  saxpy<<<(N+255)/256,256>>>(...,x, y)
  cudaDeviceSynchronize();
  ...
}
```

Maximize GPU Performance with CUDA C++/Fortran

GPU Accelerated Math Libraries

DOWNLOAD CUDA 11.1 TODAY

https://developer.nvidia.com/cuda-downloads