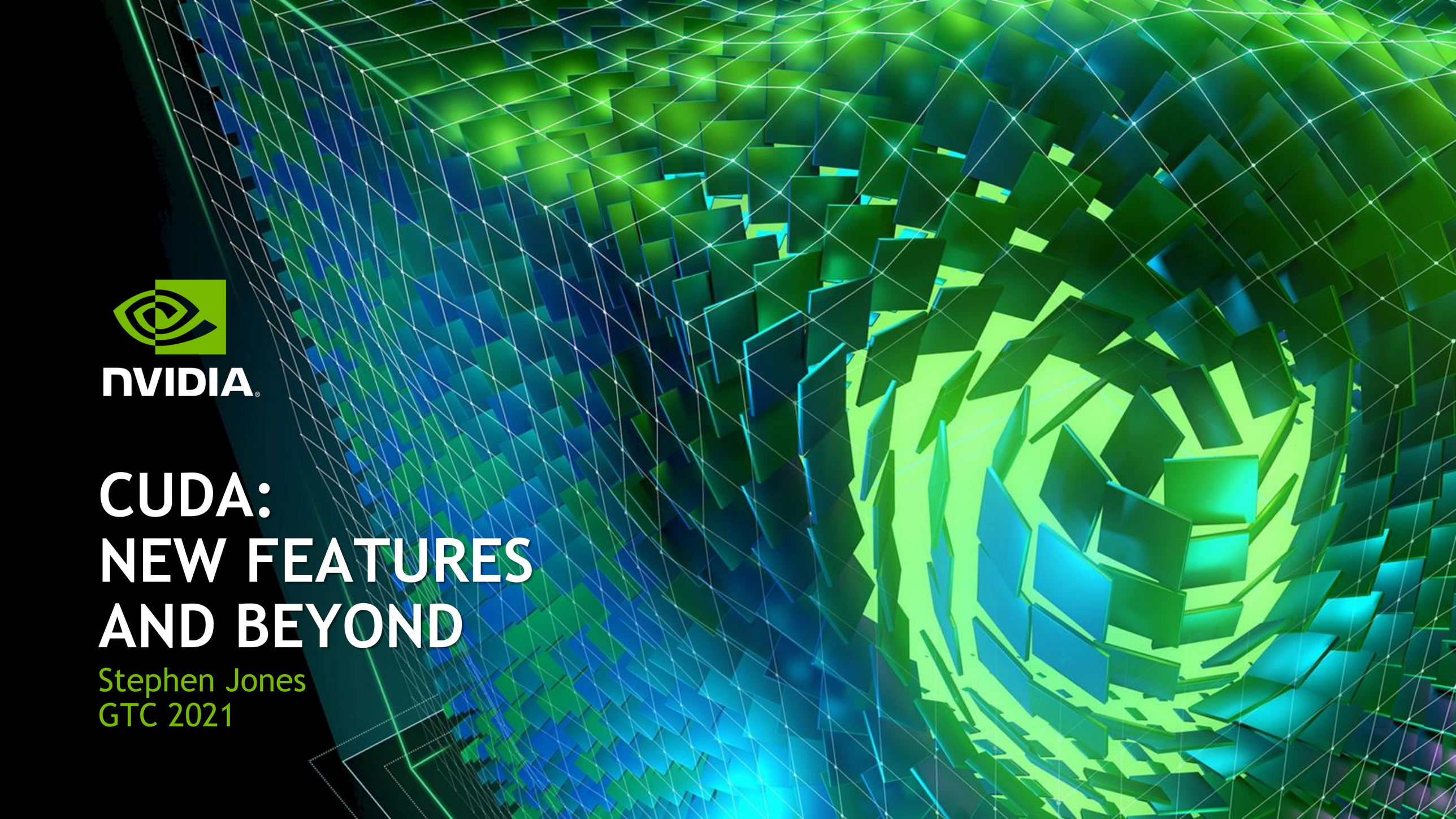
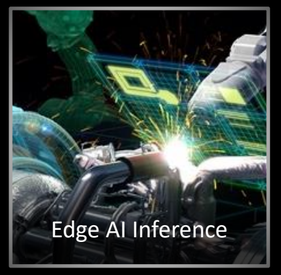
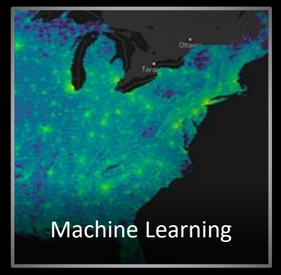


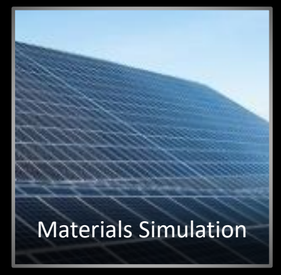
CUDA: NEW FEATURES AND BEYOND

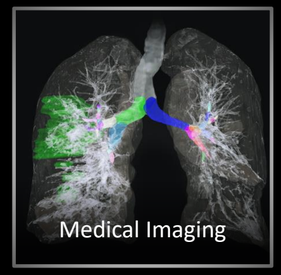
Stephen Jones

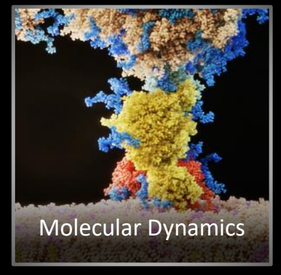
GTC 2021

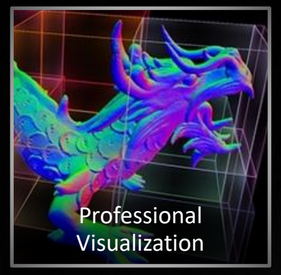


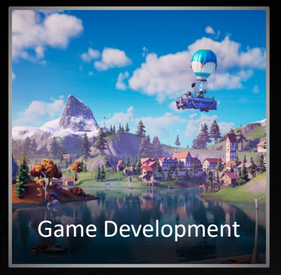


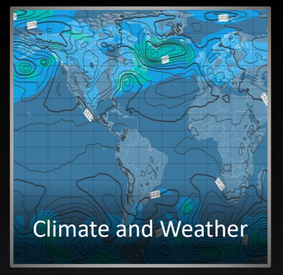


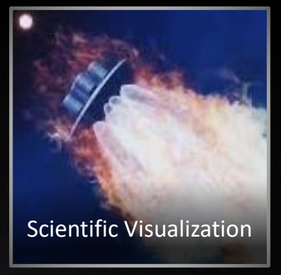






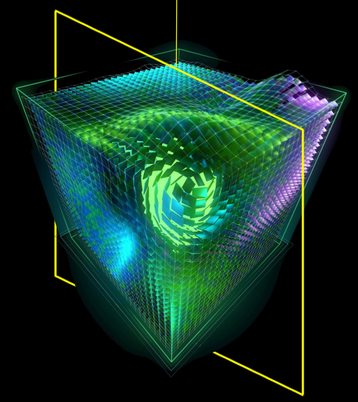




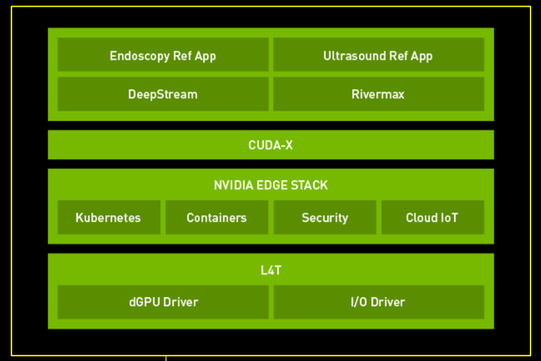


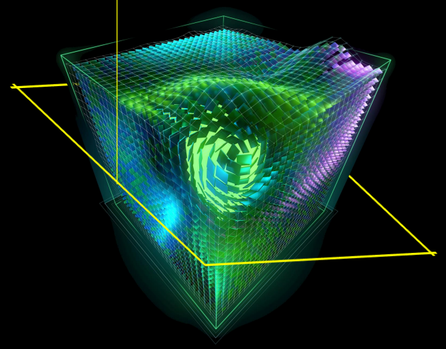
Universe of GPU Computing

HPC-SDK

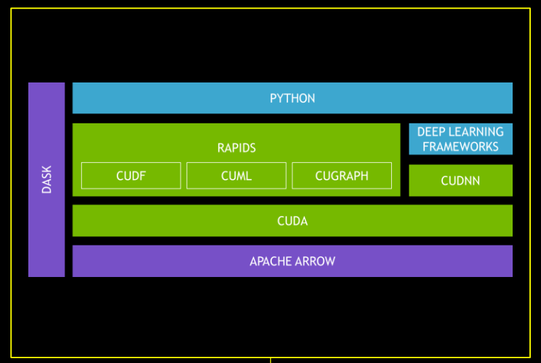


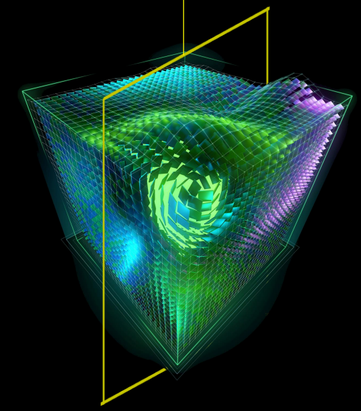
Clara





RAPIDS





NVIDIA

AXIS 1: ENABLING GPU COMPUTING ON DIFFERENT SYSTEMS

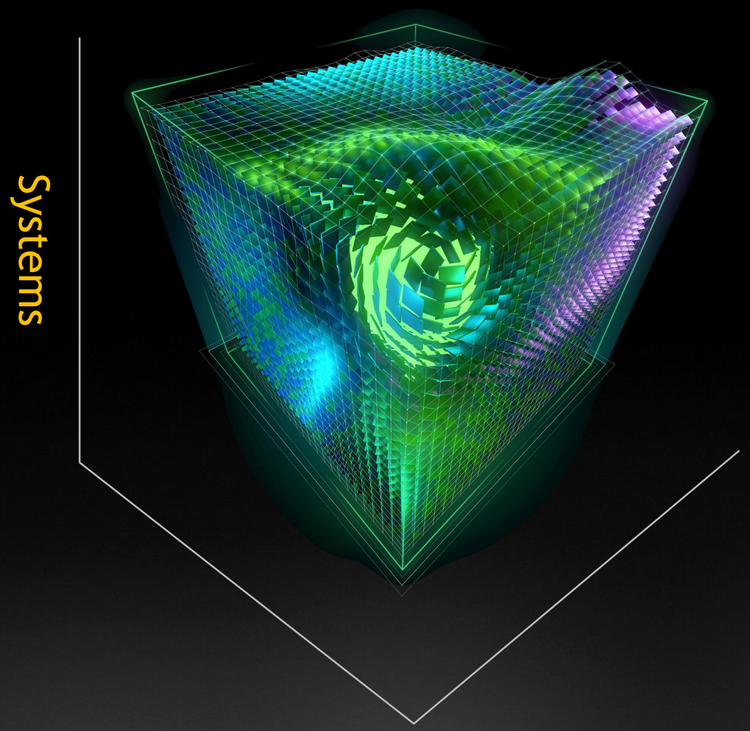
Linux

Windows

L4T

QNX

…



CUDA ON WINDOWS SUBSYSTEM FOR LINUX

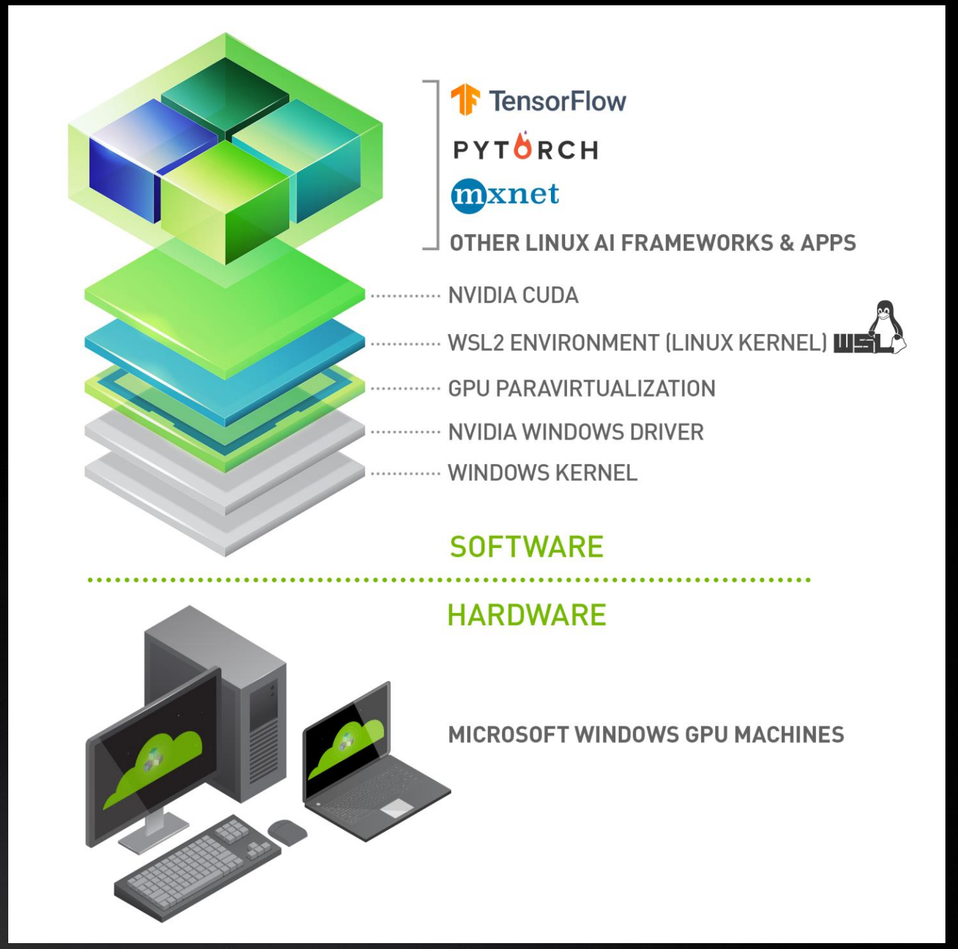
Run a Linux kernel **natively** on top of Windows 10

Runs Linux at native performance **without emulation**

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

Enables hybrid workloads on desktops or laptops as well as on enterprise scale deployments for production.

Developers and students **do not need dual-boot systems**



CUDA ON WINDOWS SUBSYSTEM FOR LINUX

Preview Available as Part of Microsoft Windows Insider Program

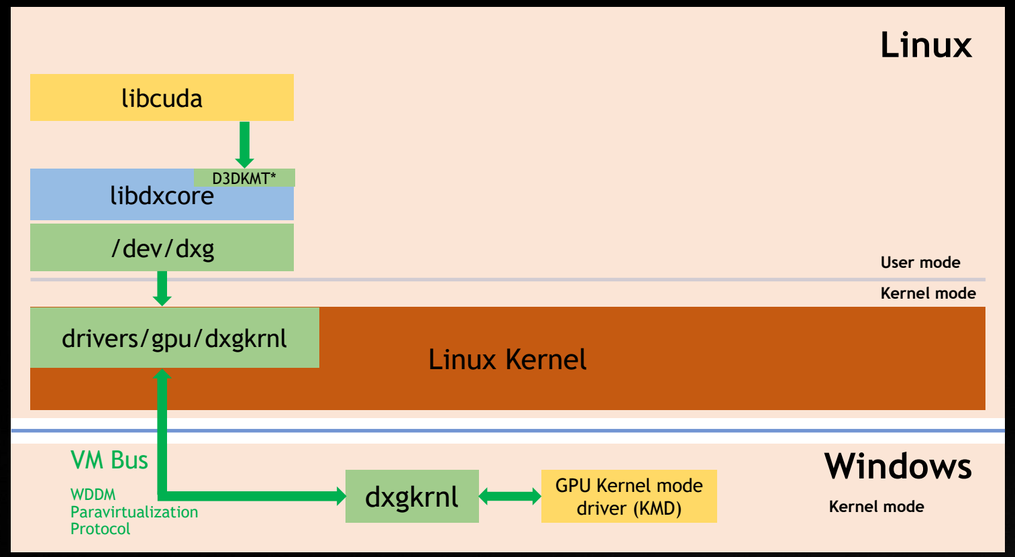


Diagram of the WDDM model supporting the CUDA user mode driver running inside Linux guest

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

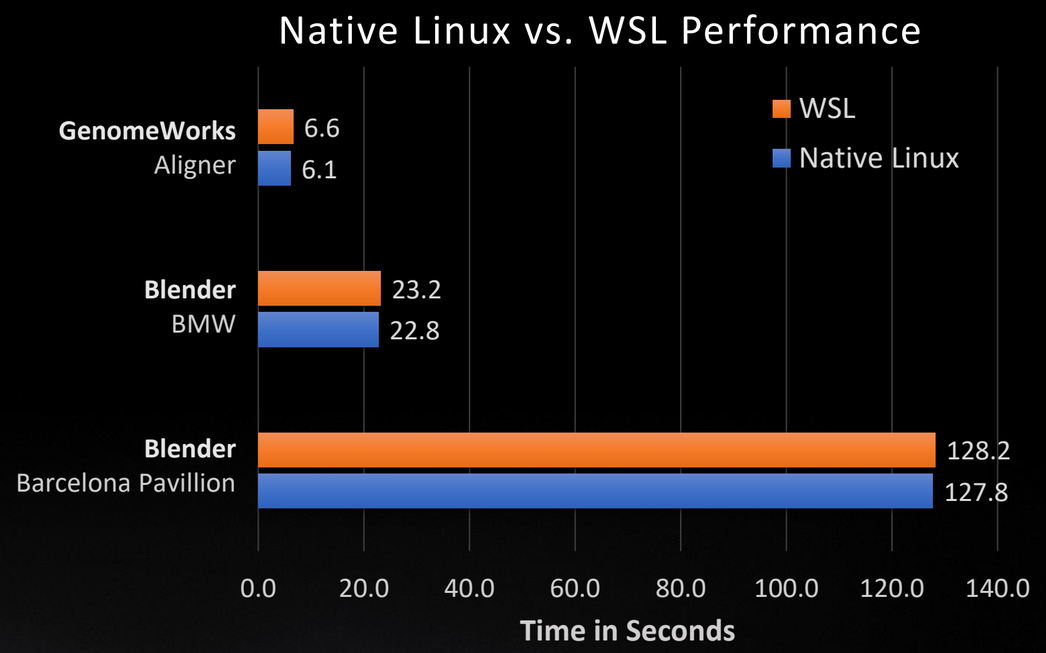
GPU-ACCELERATED DATA SCIENCE ON WSL

Get the latest version of Docker and run:

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

GPU-enabled DirectX, CUDA 11.3 and the NVIDIA Container Toolkit are all available on WSL today

NVML and NCCL support coming soon



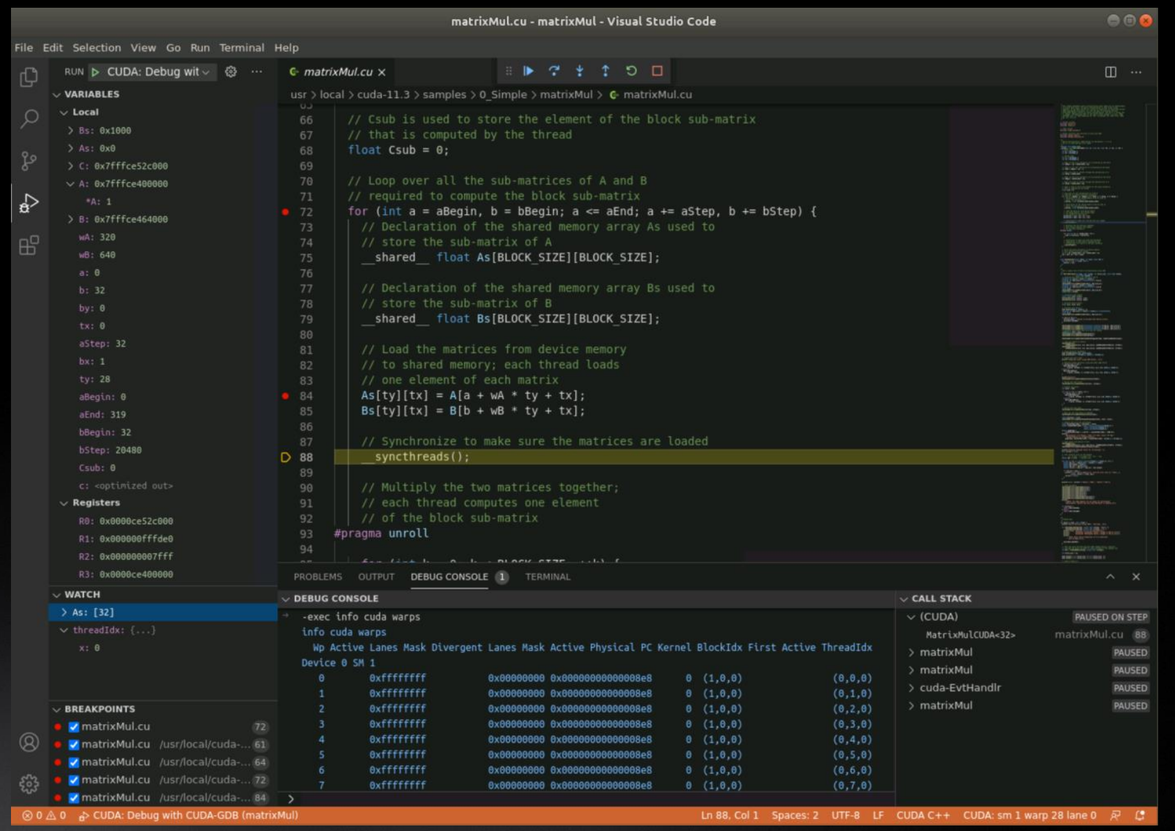
RAPIDS on Windows Subsystem for Linux [Article]

NVIDIA

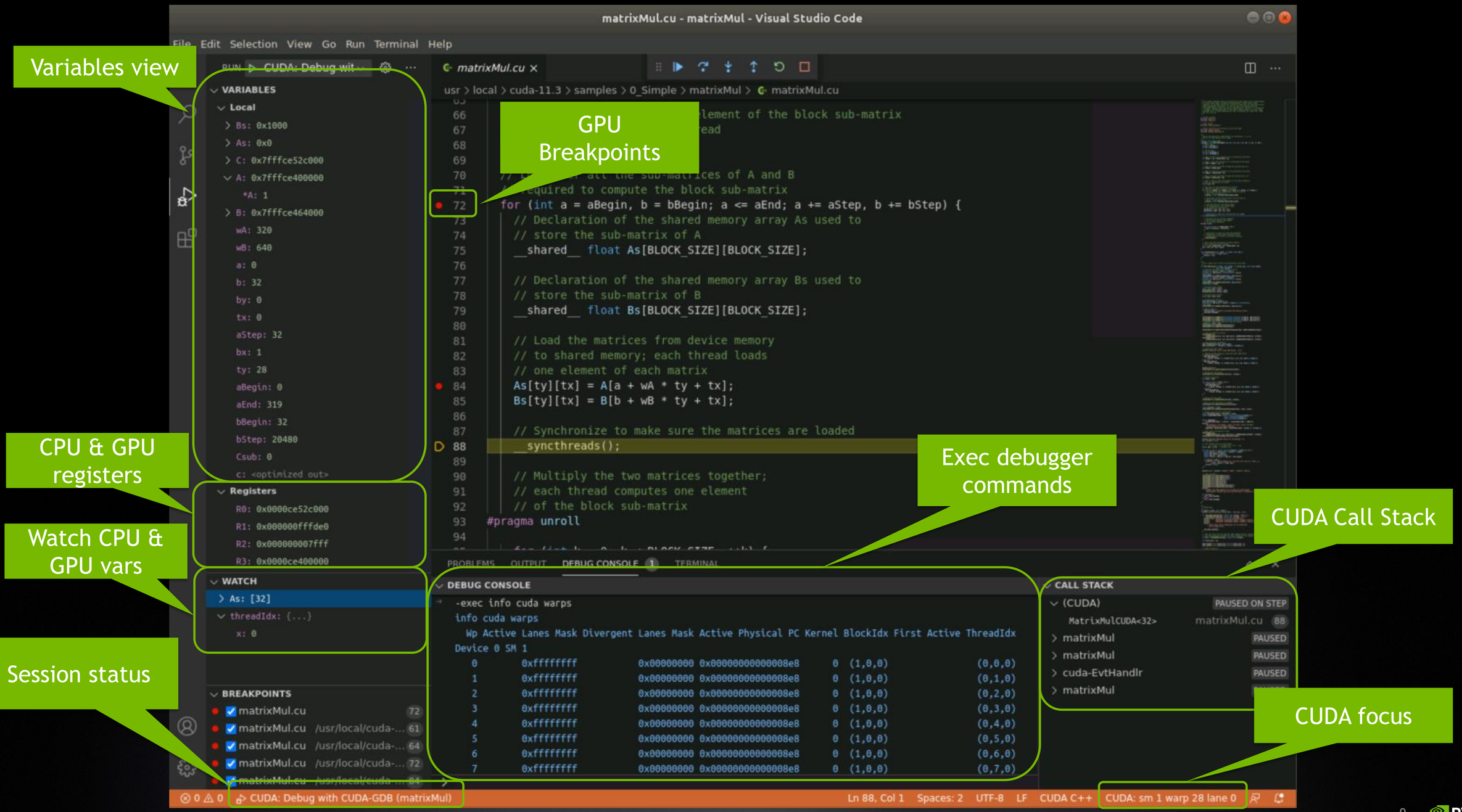
NSIGHT - VISUAL STUDIO CODE EDITION

CUDA is now a supported language in Microsoft Visual Studio Code, including extensions that provide

- CUDA code syntax highlighting
- CUDA code completion
- Build warning/errors
- Integrated debug of CPU & GPU code



Latest Enhancements to CUDA Debugger IDEs [S31884]


Variables view
GPU Breakpoints
CPU & GPU registers
Watch CPU & GPU vars
Exec debugger commands
CUDA Call Stack
Session status
CUDA focus
matrixMul.cu - matrixMul - Visual Studio Code
File Edit Selection View Go Run Terminal Help
VARIABLES
Local
Registers
WATCH
BREAKPOINTS
DEBUG CONSOLE
CALL STACK
CUDA: Debug with CUDA-GDB (matrixMul)
CUDA: sm 1 warp 28 lane 0

AXIS 2: ENABLING GPU COMPUTING ON DIFFERENT PLATFORMS

Desktop

Mobile

Datacenter

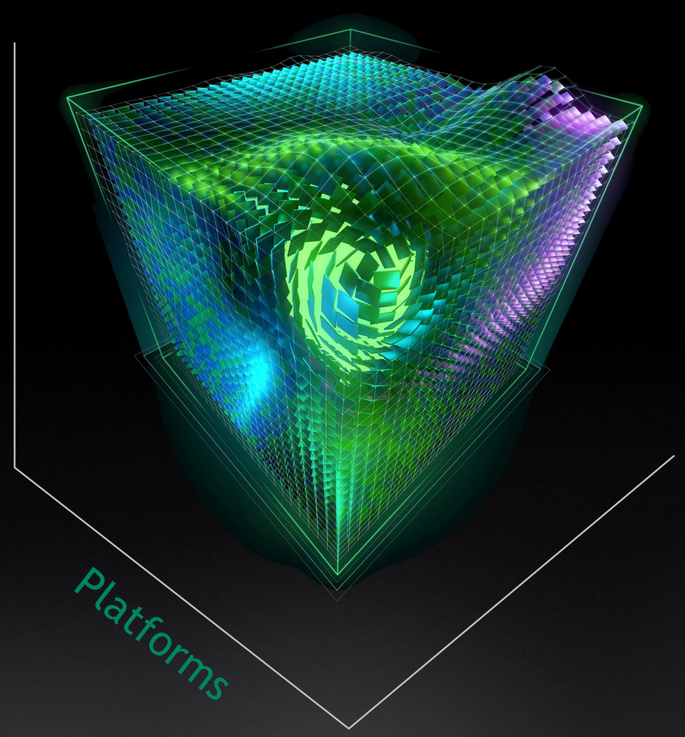
Cloud

HPC

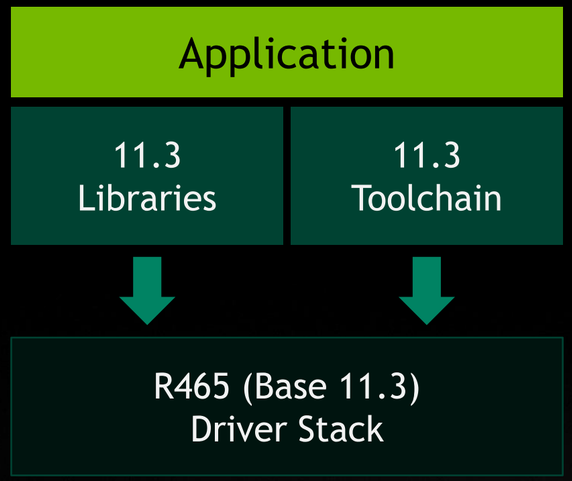
Edge

Nano

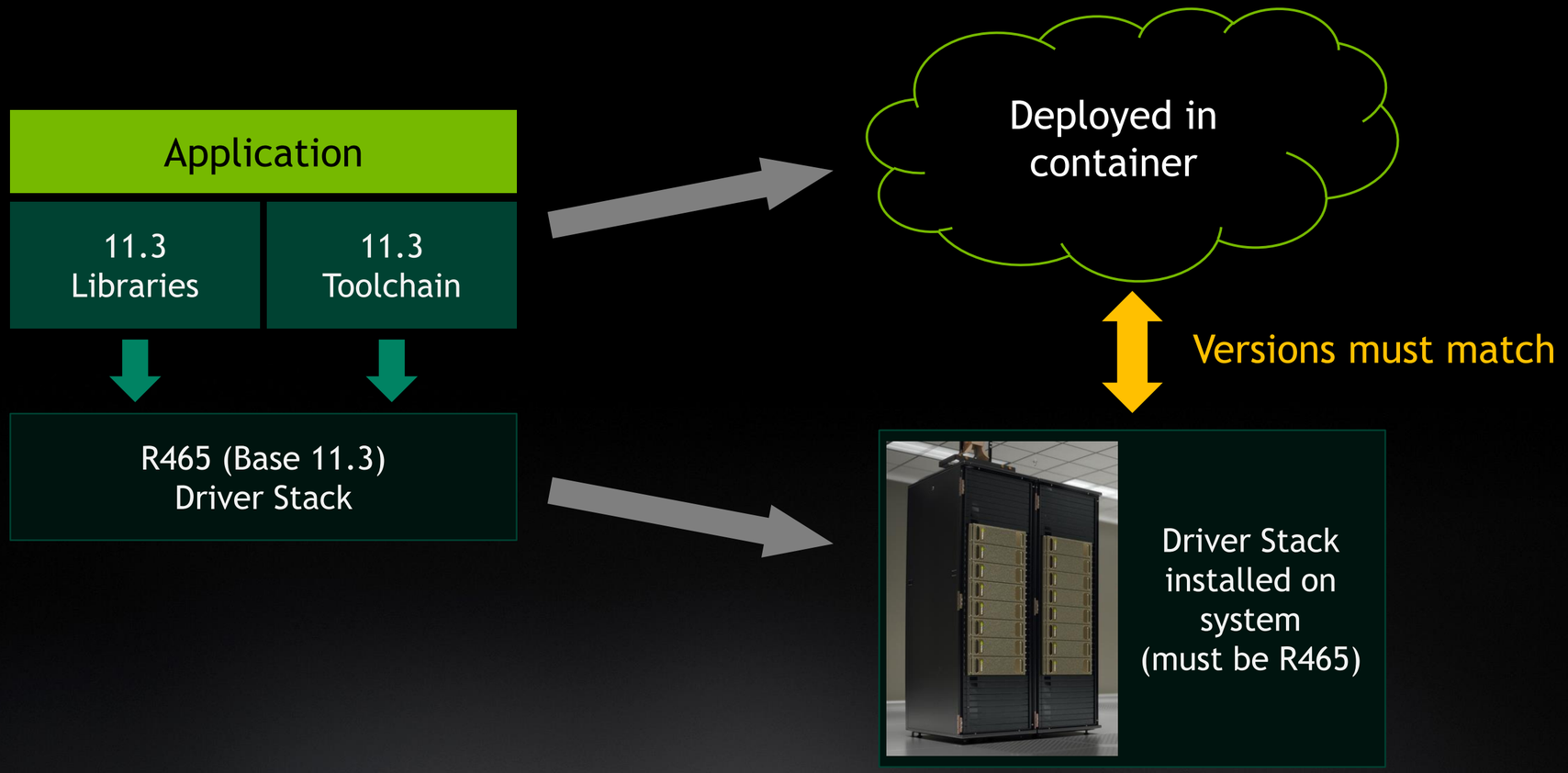
...



ANATOMY OF A CUDA APPLICATION



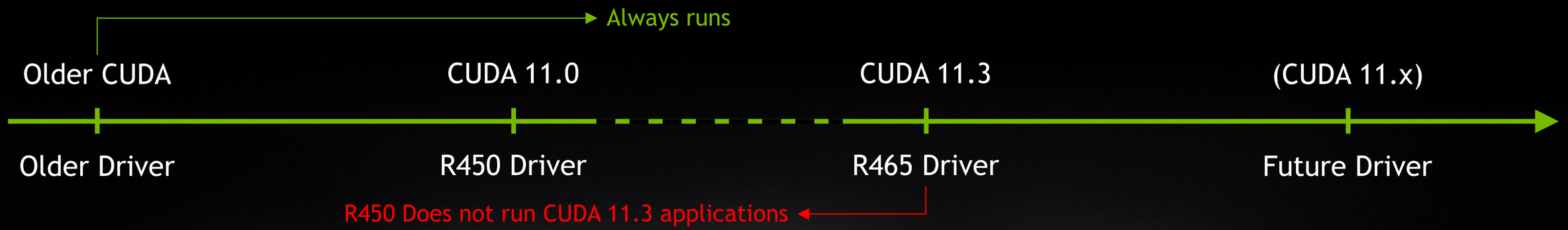
ENTERPRISE & CLOUD DEPLOYMENT, CUDA 11.0



LEGACY BACKWARD COMPATIBILITY

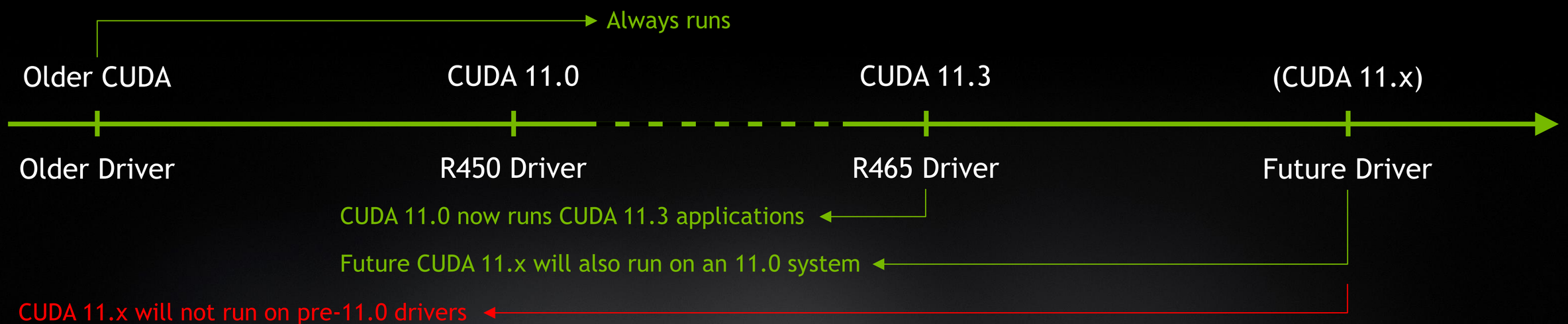
Older CUDA **always** runs on **newer** drivers

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



MINOR VERSION COMPATIBILITY

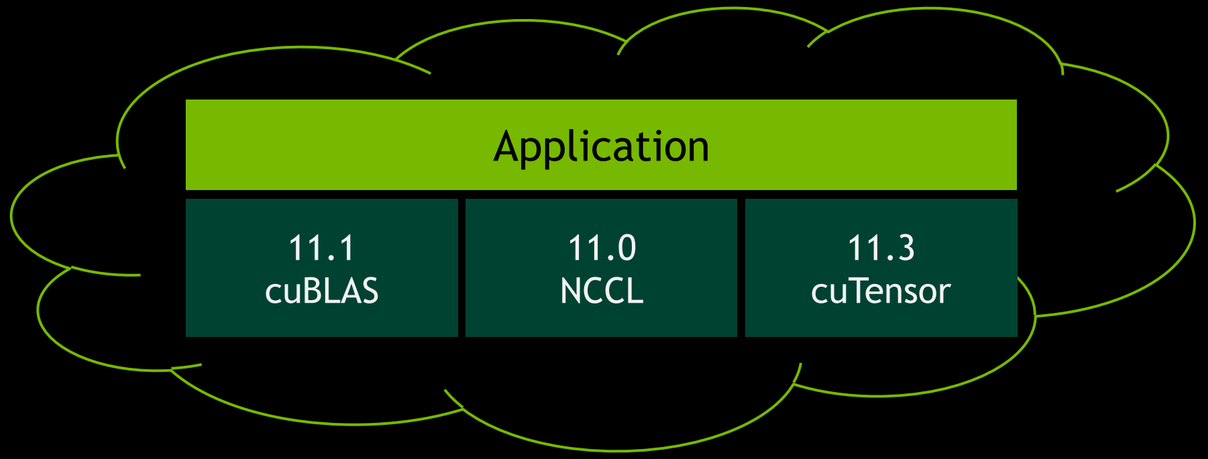
- Older CUDA **always** runs on **newer** drivers
- Newer CUDA 11.x **will** now run on older drivers with the **same major version**



MINOR VERSION COMPATIBILITY

Build with any 11.x toolchain

Libraries from different CUDA versions run on any base driver

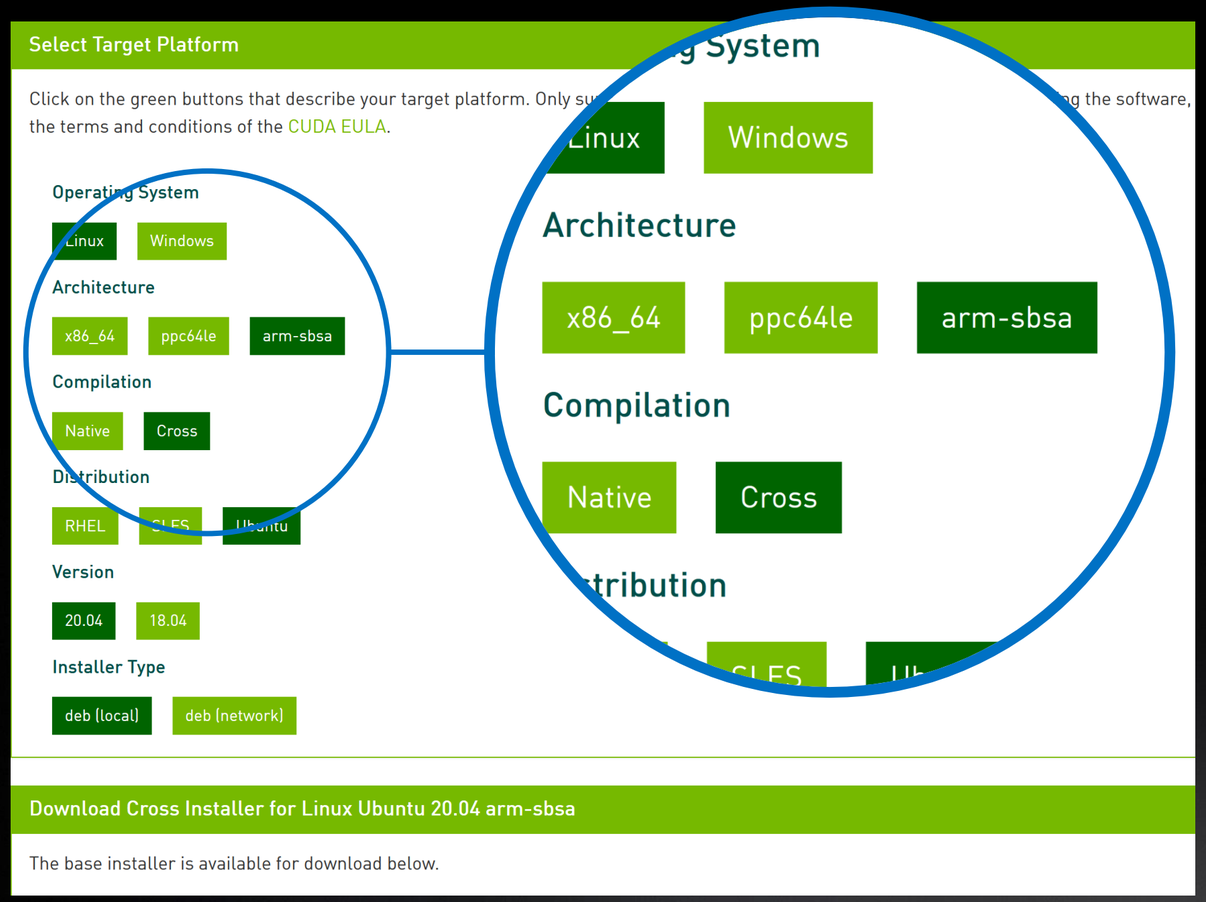


Deploy on any 11.x system

CUDA Programming Model and Compatibility Guarantees [CWES2484]

DOWNLOAD CUDA FOR ARM

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



Select SBSA (Server Base System Architecture)

Common specification for Arm servers

Compilation modes

Native for development on Arm system

Cross for x86-based development + Arm deployment

CUDA Toolkit for Arm

- CUDA runtime & drivers
- nvcc native & cross compilers
- Math libraries
- Developer tools

THE NVIDIA HPC COMPILER IS READY FOR ARM

Includes Auto Vectorization and explicit Vector Intrinsics

```
float32x2_t vadd_f32 (float32x2_t a, float32x2_t b);
float32x4_t vdupq_n_f32 (float32_t value);
float32x2_t vget_high_f32 (float32x4_t a);
```

Host Parallelism

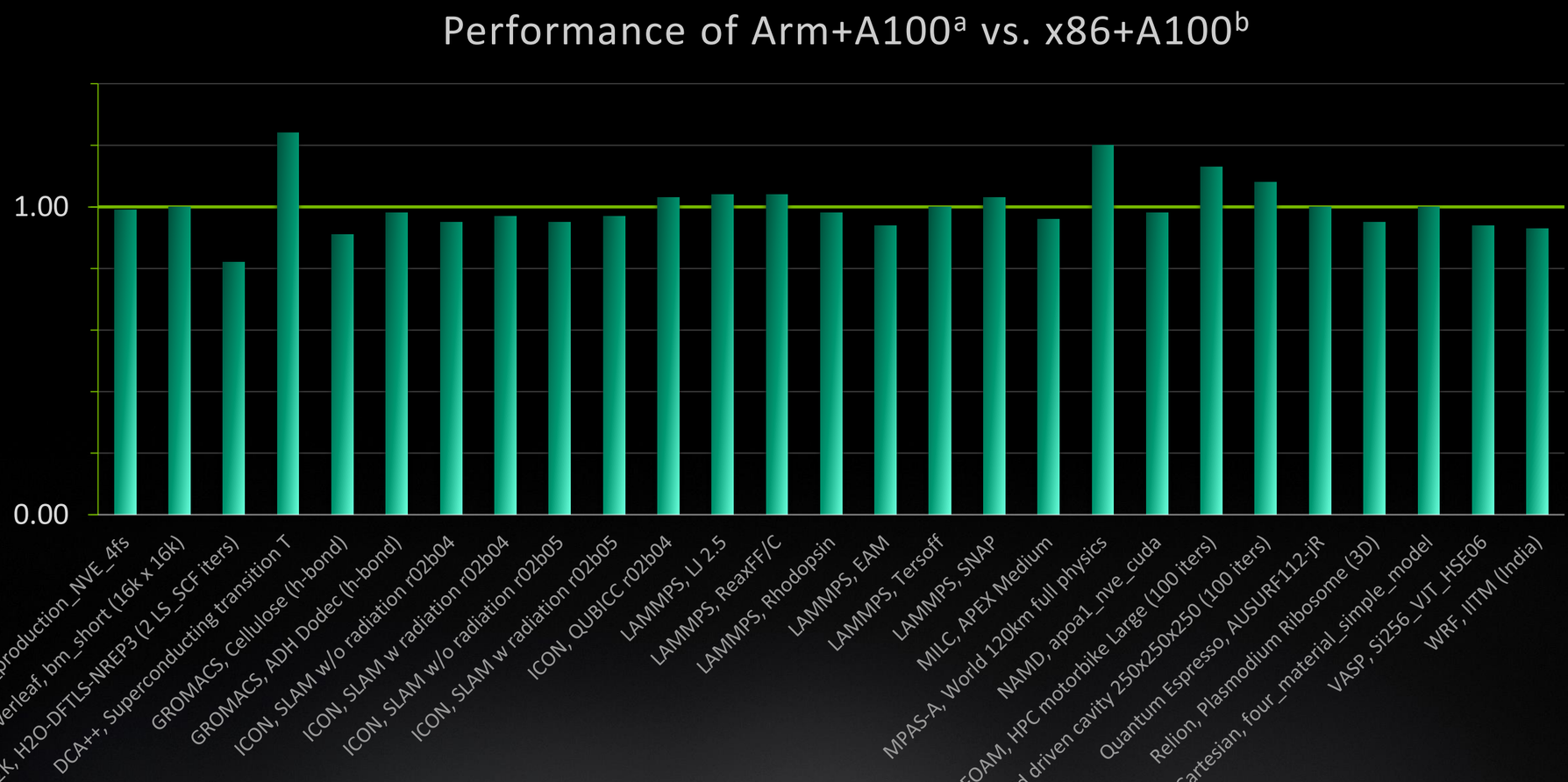
- Standard Parallelism
- OpenACC
- OpenMP

FP Models

- Precise
- Fast
- Relaxed

A Deep Dive into the Latest HPC Software [S31286]

GPU APPLICATION PERFORMANCE ON ARM



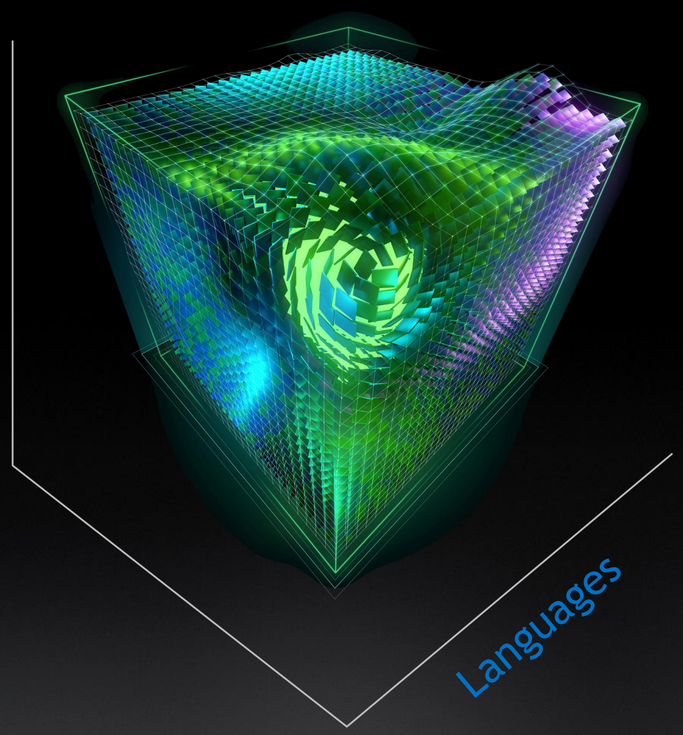
a) Ampere Altra Q80-30 + 2xA100-PCIe-40

b) EPYC 7742 + 2xA100-PCIe-40

HPC Applications on ARM+NVIDIA A100 [S32758]

NVIDIA

AXIS 3: ENABLING GPU COMPUTING IN ANY LANGUAGE



C++

Julia

FORTRAN

MATLAB

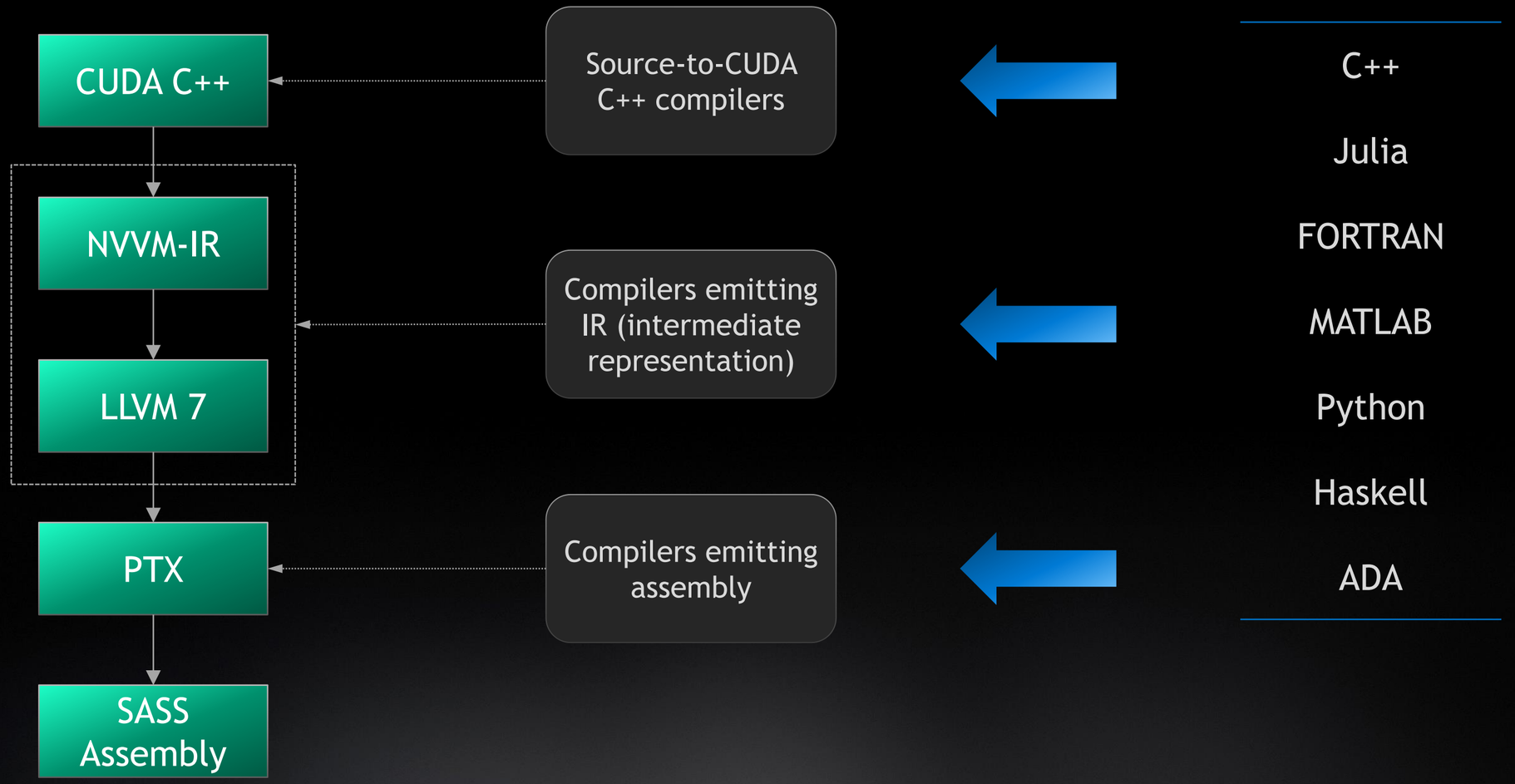
Python

Haskell

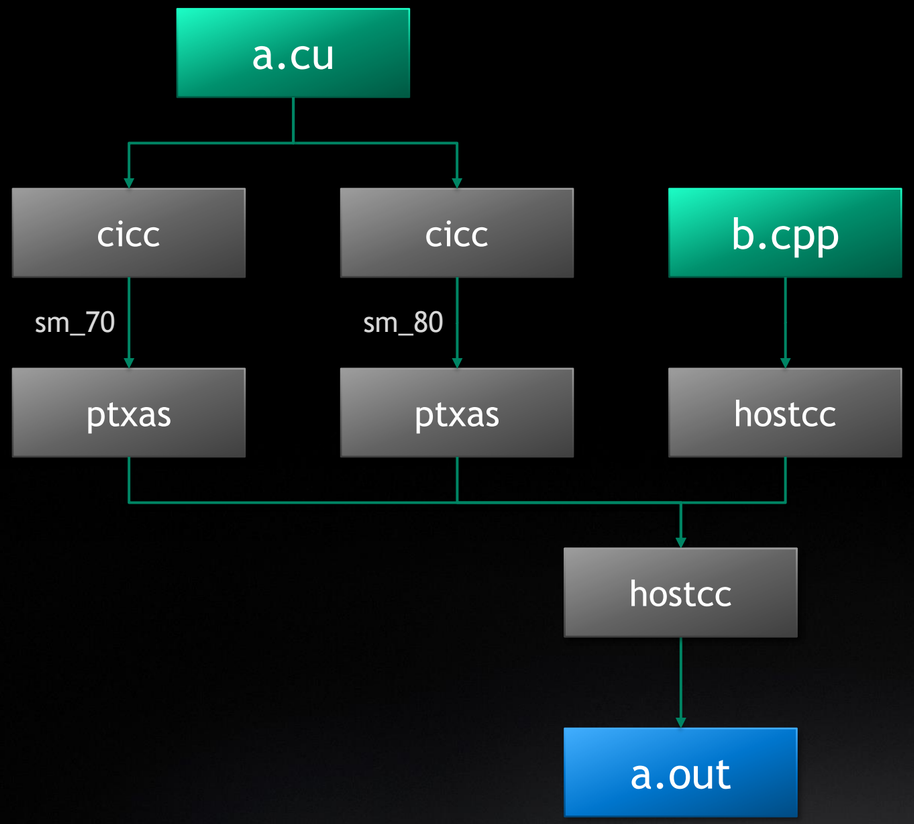
ADA

...

GPU COMPILER STACK



NVCC MULTI-TARGET PARALLEL COMPILATION



Parallel compilation for multiple architectures

With the '**-t**' flag, *nvcc* now compiles multiple inputs and/or different SM architectures in parallel

Multiple input files specified in a single *nvcc* invocation will be built in parallel

Multi-architecture builds such as libraries should see significant build speedups

```
# nvcc –t x.cu y.cu z.cu    ← Multi-target, single arch

# nvcc –t a.cu b.cpp –gencode=arch=compute_70,code=sm_70 \
                     –gencode=arch=compute_80,code=sm_80
```

RUN-TIME COMPILATION WITH NVRTC

Creating CUDA kernels dynamically from within your program

NVRTC - The run-time compiler built in to CUDA

Enables programmatic generation and tuning of kernels

Part of Minor Version Compatibility since CUDA 11.2

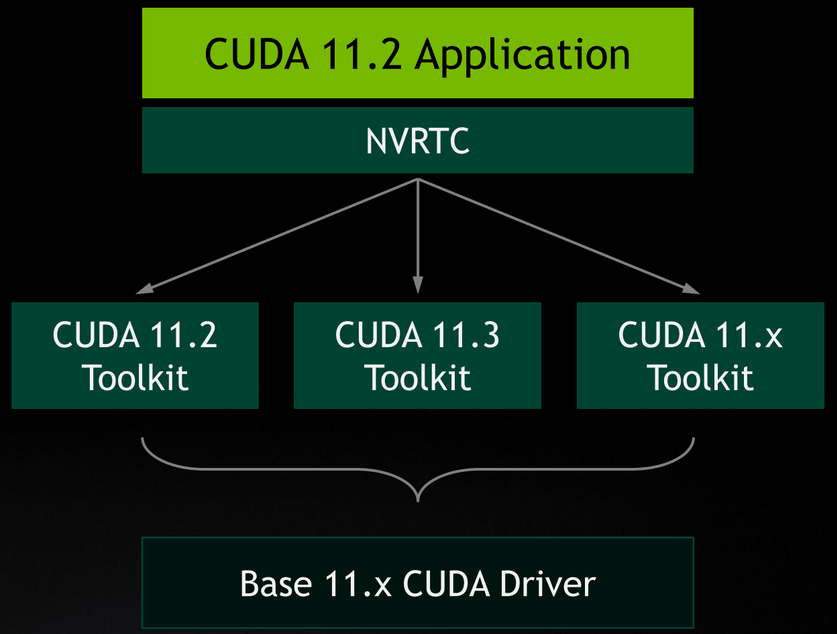
Generate code at run-time with any other minor driver version

Standalone redistributable library option

Linking NVRTC into your application guarantees the version you run with

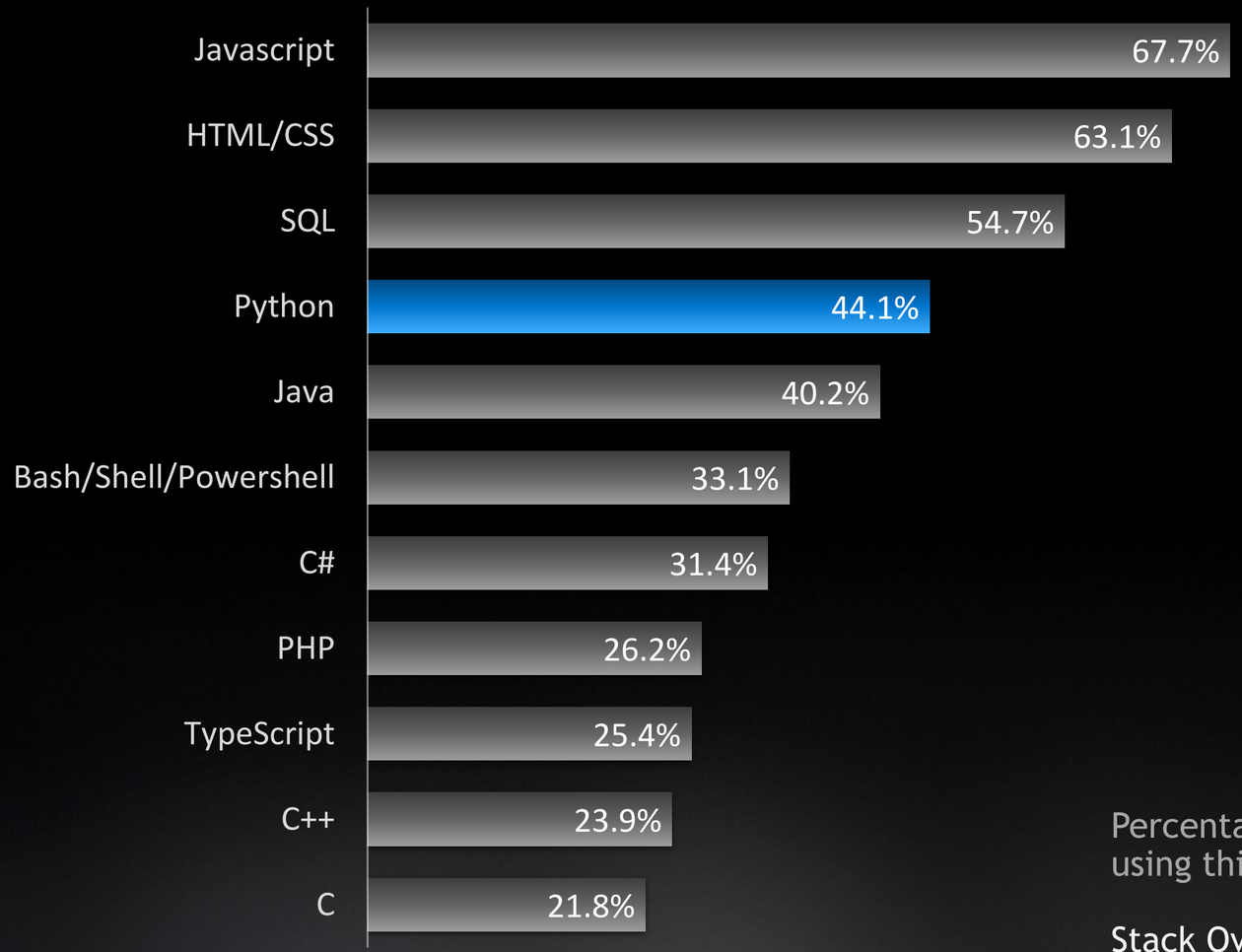
Direct output of SASS assembly code

Speeds up compilation by skipping the PTX intermediate layer and helps avoid PTX minor version mismatches



No longer necessary for application NVRTC version to match installed Toolkit version

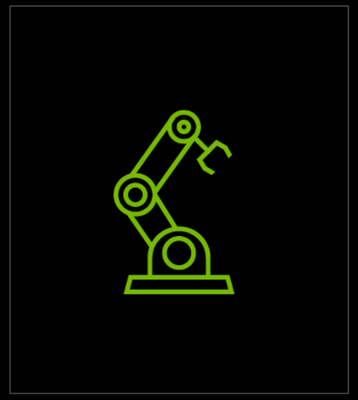
IN CASE YOU HADN'T NOTICED, PYTHON IS BIG



Percentage of 57,378 respondents who report using this language within the past year

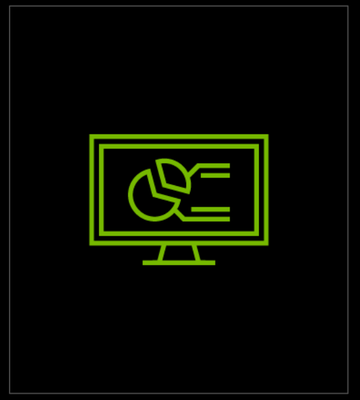
Stack Overflow 2020 Developer Survey

PYTHON IS BIG IN MOST GPU DOMAINS

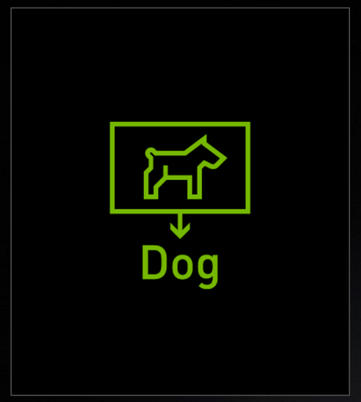


Embedded Systems

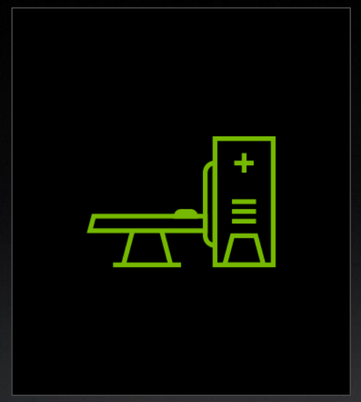
Autonomous Vehicles



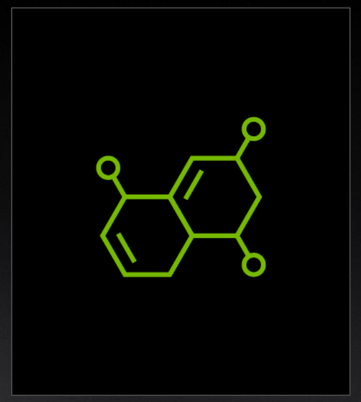
Data Science



Artificial Intelligence



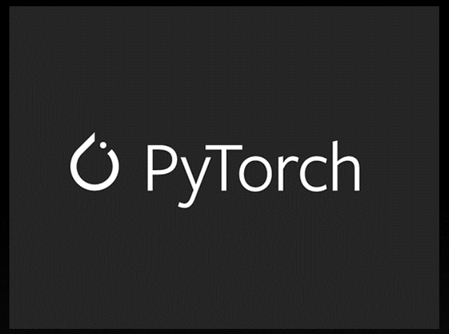
Healthcare

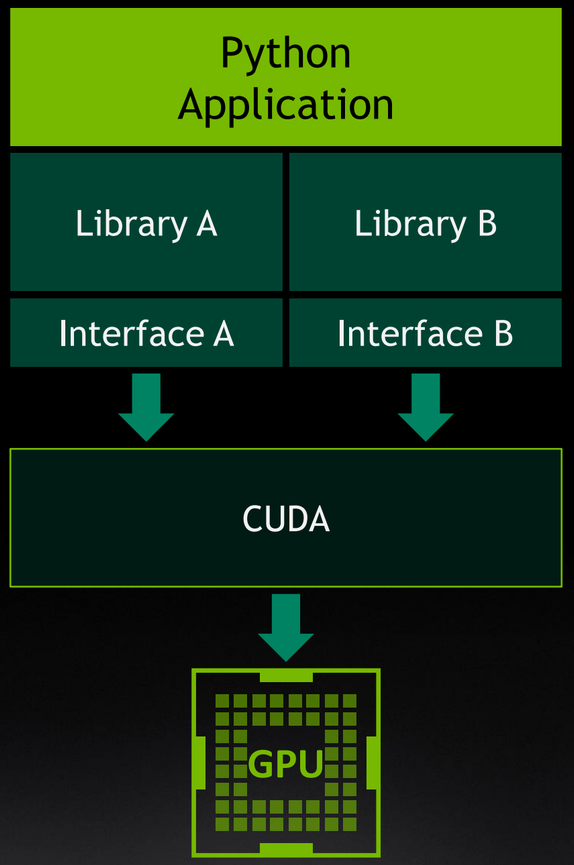


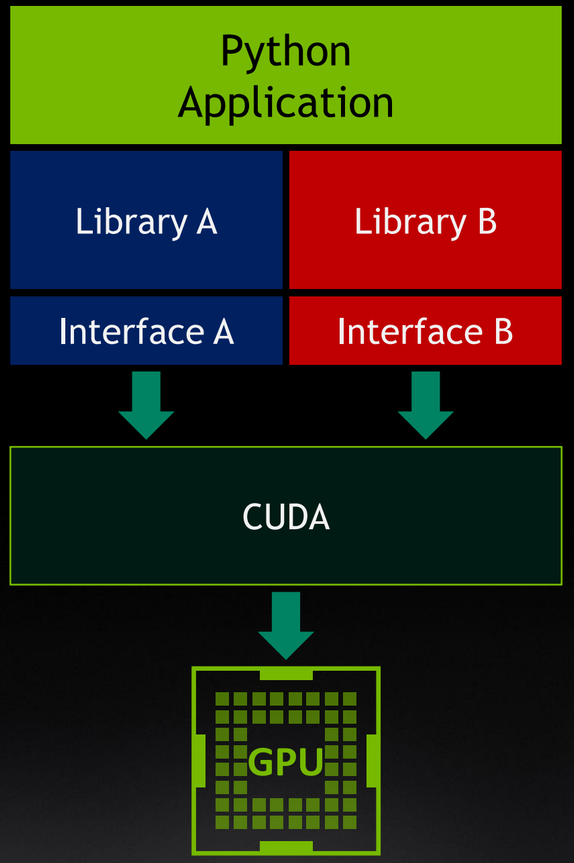
Scientific Computing

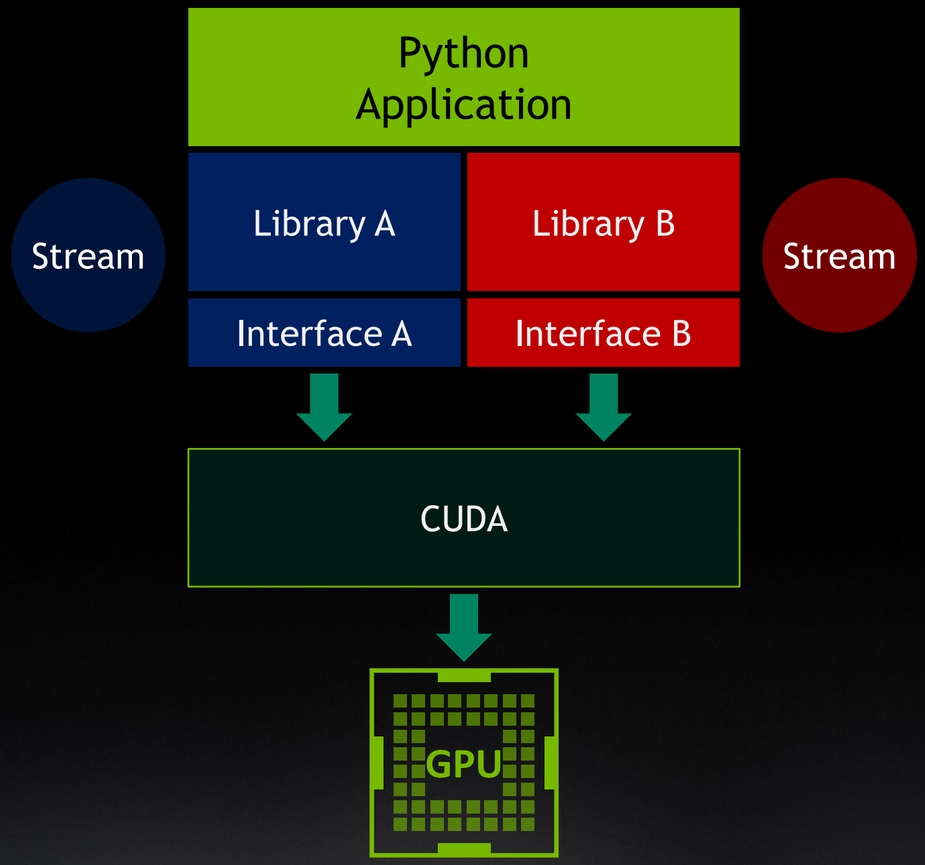
Finance

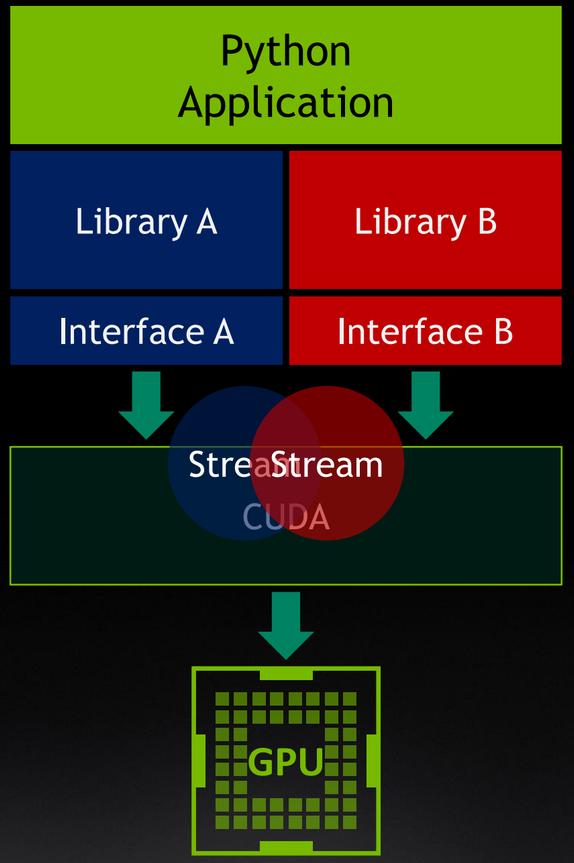
AN ECOSYSTEM OF PYTHON PROJECTS INTERFACE WITH CUDA

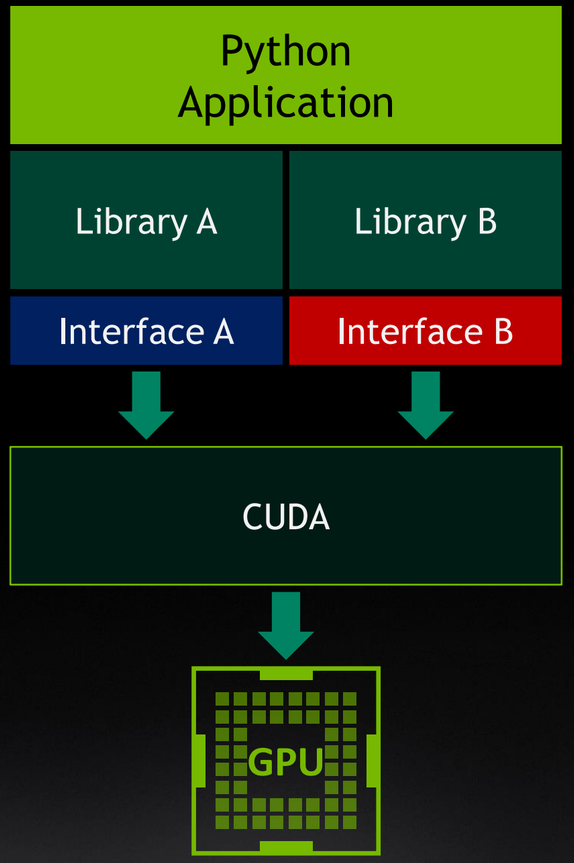


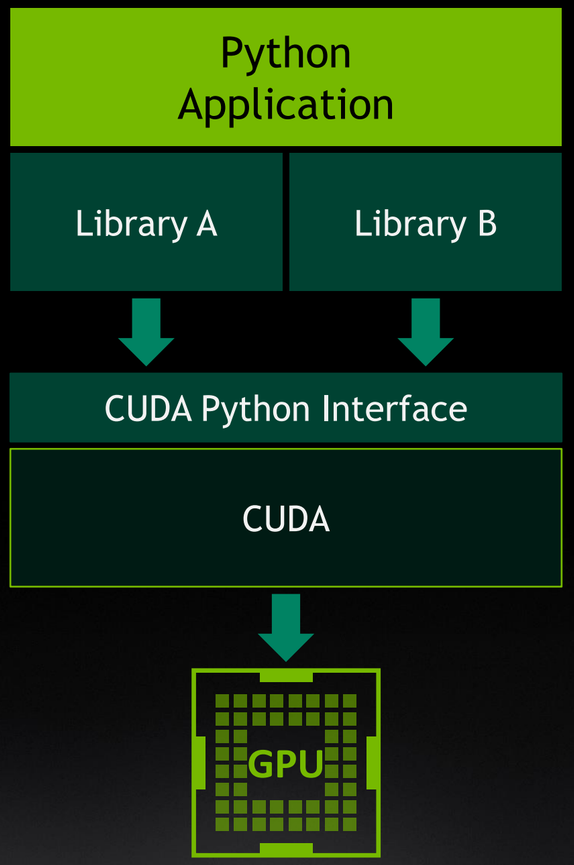

Python
Application
Library A
Library B
Interface A
Interface B
CUDA
GPU


Python
Application
Library A
Library B
Interface A
Interface B
CUDA
GPU

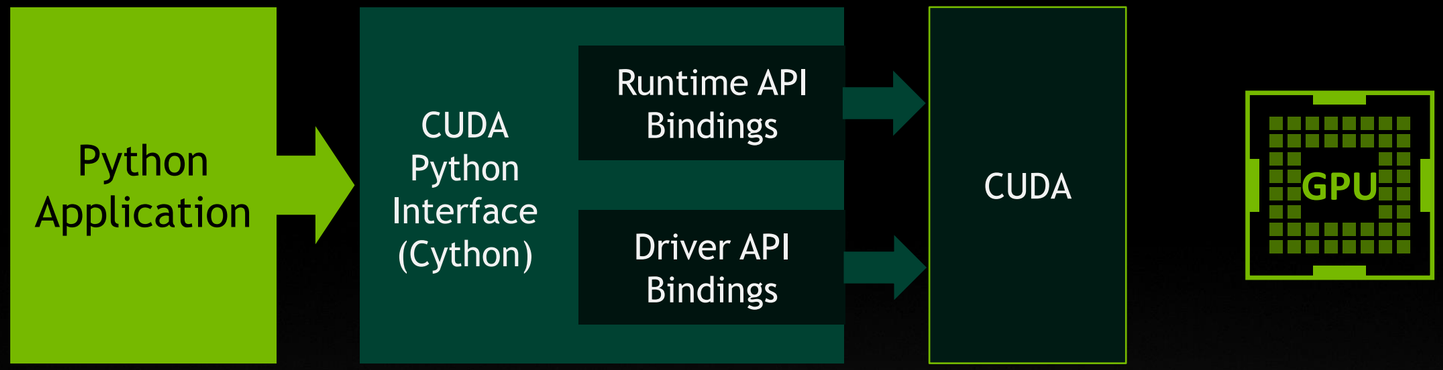

Python
Application
Stream
Library A
Library B
Stream
Interface A
Interface B
CUDA
GPU


Python
Application
Library A
Library B
Interface A
Interface B
Stream
CUDA
GPU


Python
Application
Library A
Library B
Interface A
Interface B
CUDA
GPU


Python
Application
Library A
Library B
CUDA Python Interface
CUDA
GPU

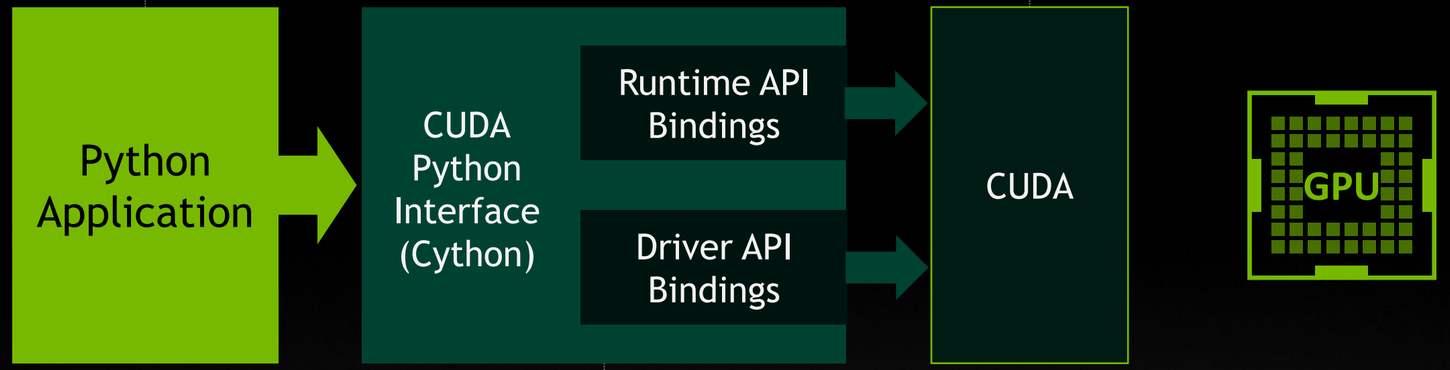
CUDA RUNTIME & DRIVER API BINDINGS



CUDA RUNTIME & DRIVER API BINDINGS

```
cudaErr, stream = cudaStreamCreate()
```

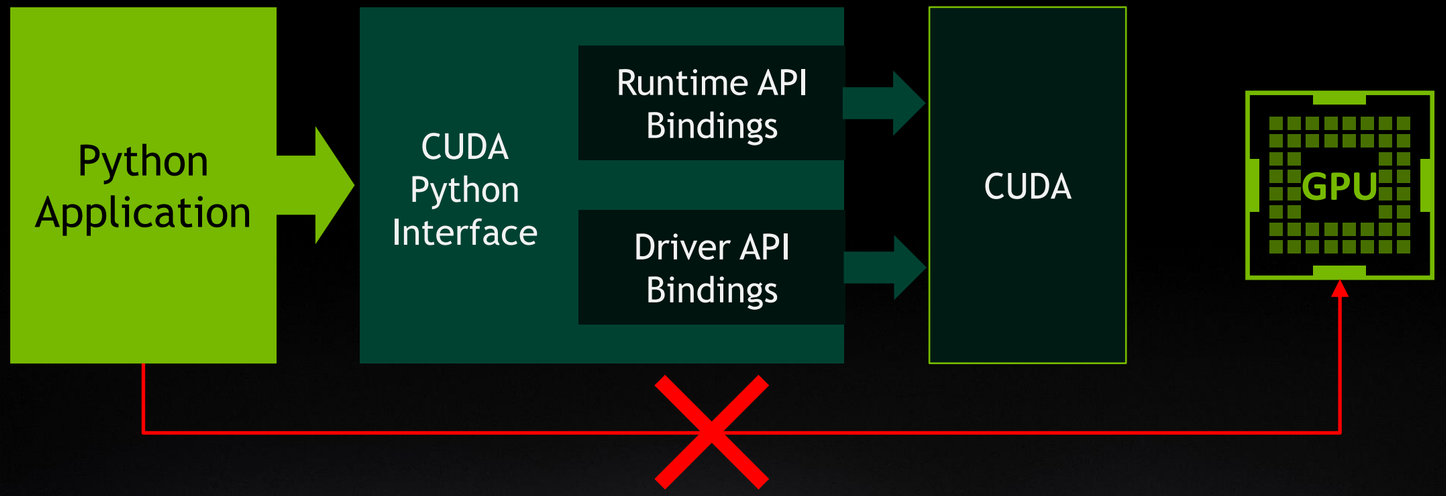
```
cudaError_t cudaStreamCreate(cudaStream_t *pStream)
```




```
cdef cudaStream_t stream
cdef cudaError_t cudaErr = cudaStreamCreate(&stream)
```

HOST API BINDINGS ONLY

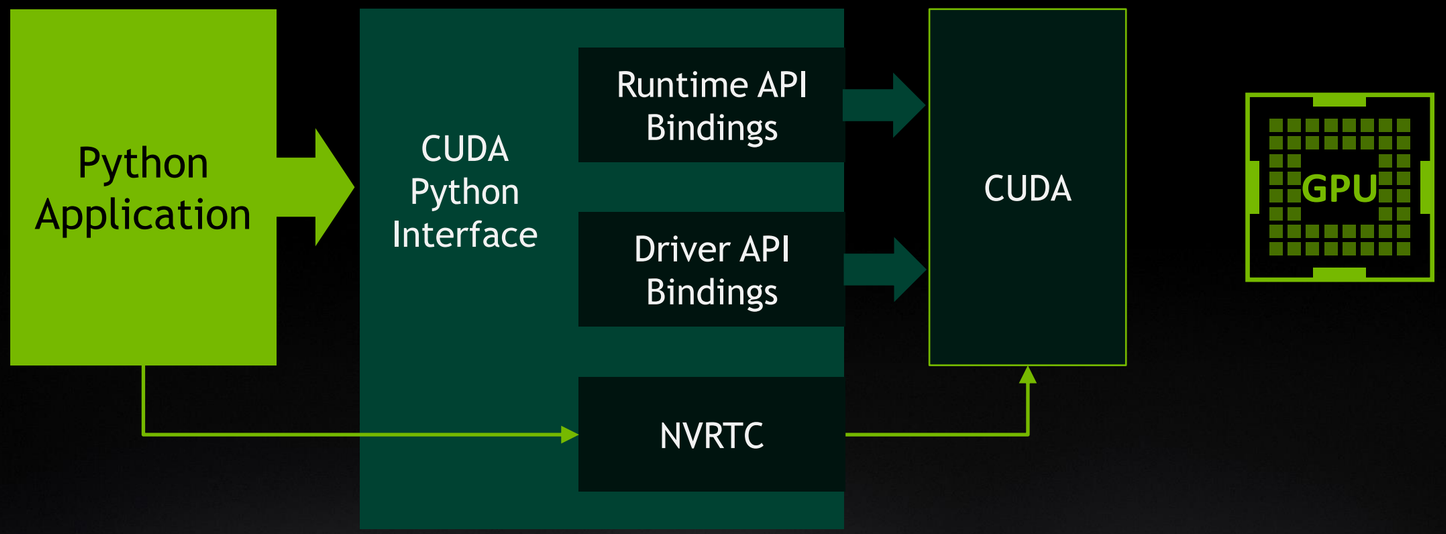
Use Numba, CuPy, PyCUDA, etc. to run native Python on the GPU



CUDA **is not** running Python code on the GPU
These are **host API bindings only**

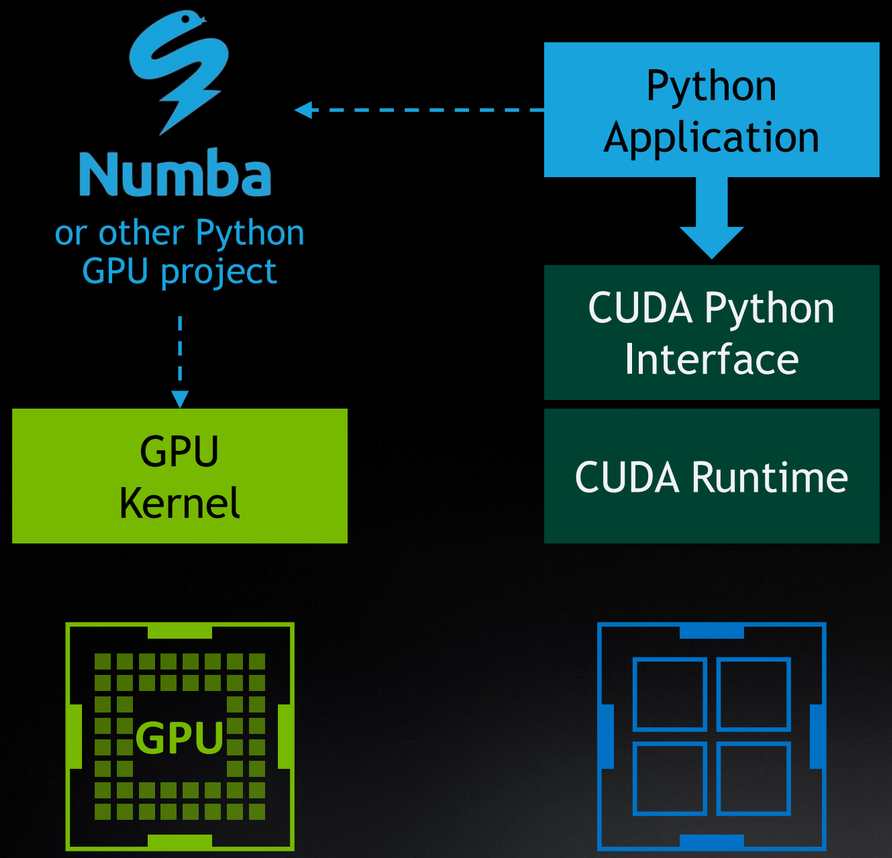
NVRTC BINDINGS INCLUDED

Use Numba, CuPy, PyCUDA, etc. to run native Python on the GPU



JIT CUDA C++ code via run-time compilation

AVAILABLE SOON



Distribution

- Source available on GitHub
- PIP & Conda packages
- Redistributable license

Bindings

Full coverage of and access to the CUDA host APIs from Python

Platforms

- Linux: x86_64, sbsa, ppc64le
- Windows: x86_64

“Anaconda is very supportive of NVIDIA’s effort to provide a unified and comprehensive set of interfaces to the CUDA host APIs from Python. We look forward to adopting this package in Numba's CUDA Python compiler to reduce our maintenance burden and improve interoperability within the CUDA Python ecosystem.”

- Peter Wang, CEO Anaconda

“Quansight is a leader in data science consulting specializing in Python open source solutions. The Python data technology landscape is constantly changing and Quansight endorses NVIDIA’s efforts to provide easy-to-use CUDA API Bindings for Python. This package will be a valuable resource for building our own NVIDIA accelerated solutions and bringing these solutions to our customers.”

- Travis Oliphant, CEO Quansight

LEGATE: SCALING PYTHON

```
import legate.numpy as np
import legate.pandas as pd

size = num_rows_per_gpu * num_gpus

key_l = np.arange(size)
payload_l = np.random.randn(size) * 100.0
lhs = pd.DataFrame({ "key": key_l, "payload": payload_l })

key_r = key_l // 3 * 3    # selectivity: 0.33
payload_r = np.random.randn(size) * 100.0
rhs = pd.DataFrame({ "key": key_r, "payload": payload_r })

out = lhs.merge(rhs, on="key")
```

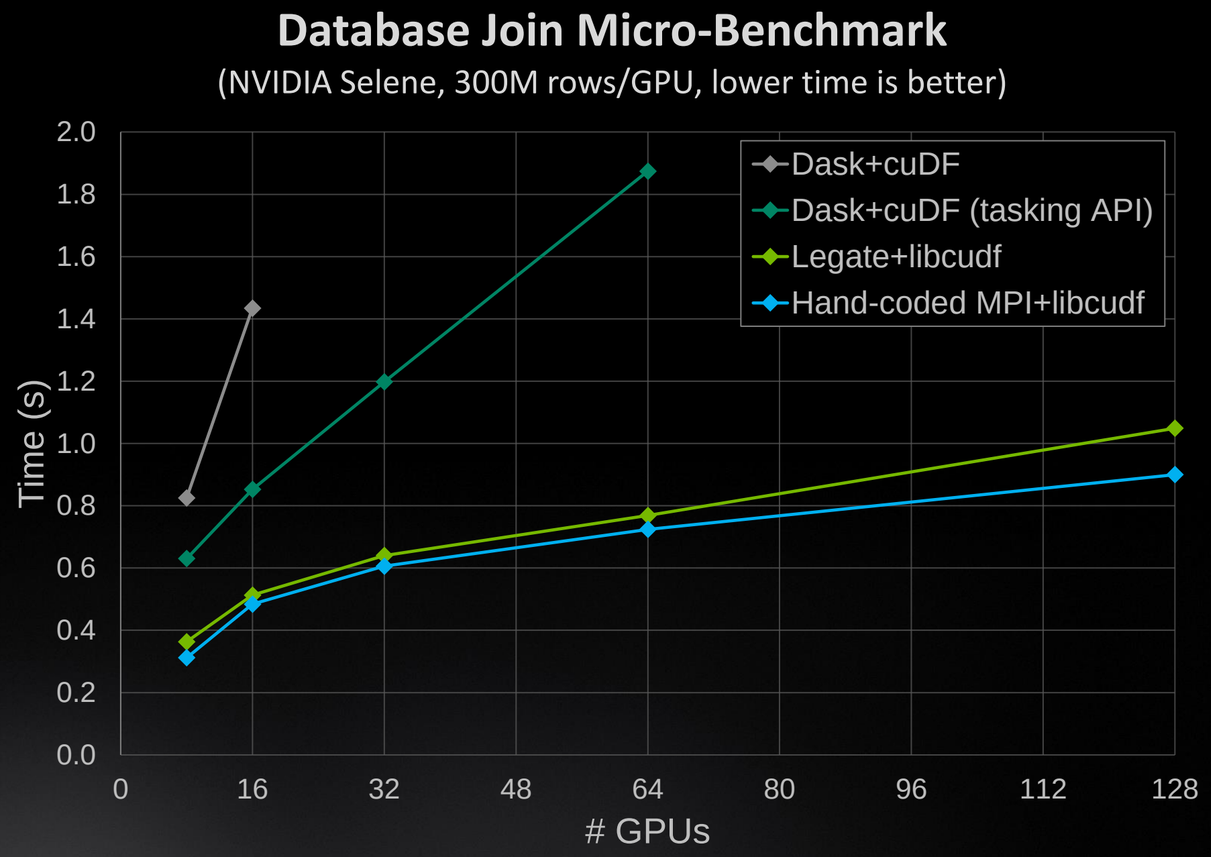
NVIDIA Selene Supercomputer - #5 on Top500

LEGATE: SCALING PYTHON

A framework for programming large numbers of GPUs as if they were a single processor

Pass data between Legate libraries without worrying about distribution or synchronization requirements

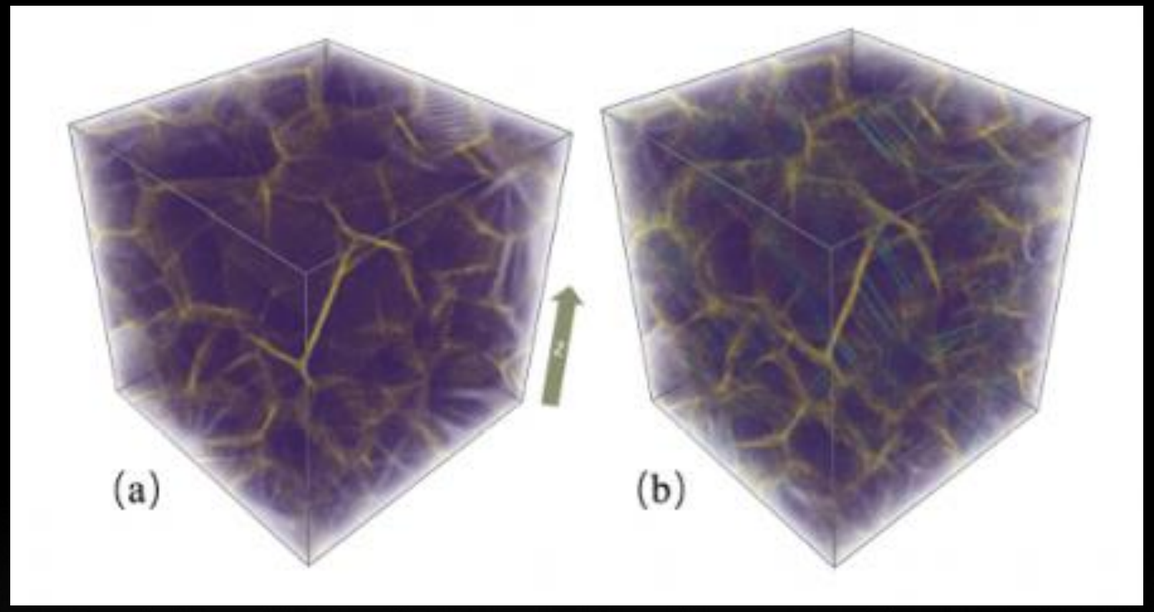
Legate NumPy and Pandas aim to transparently scale existing Numpy and Pandas workloads



HIGH PERFORMANCE PYTHON COMPUTING

Gordon Bell Prize 2016 Best Paper Finalist

Towards Green Aviation with Python at Petascale
P. Vincent, F. Witherden, Brian C. Vermeire, J. S. Park, A. Iyer



Gordon Bell Prize 2020 Winner

Pushing the limit of molecular dynamics with ab initio accuracy to 100 million atoms with machine learning
W. Jia, H. Wang, M. Chen, D. Lu, L. Lin, R. Car, Weinan E, L. Zhang

HPL-AI

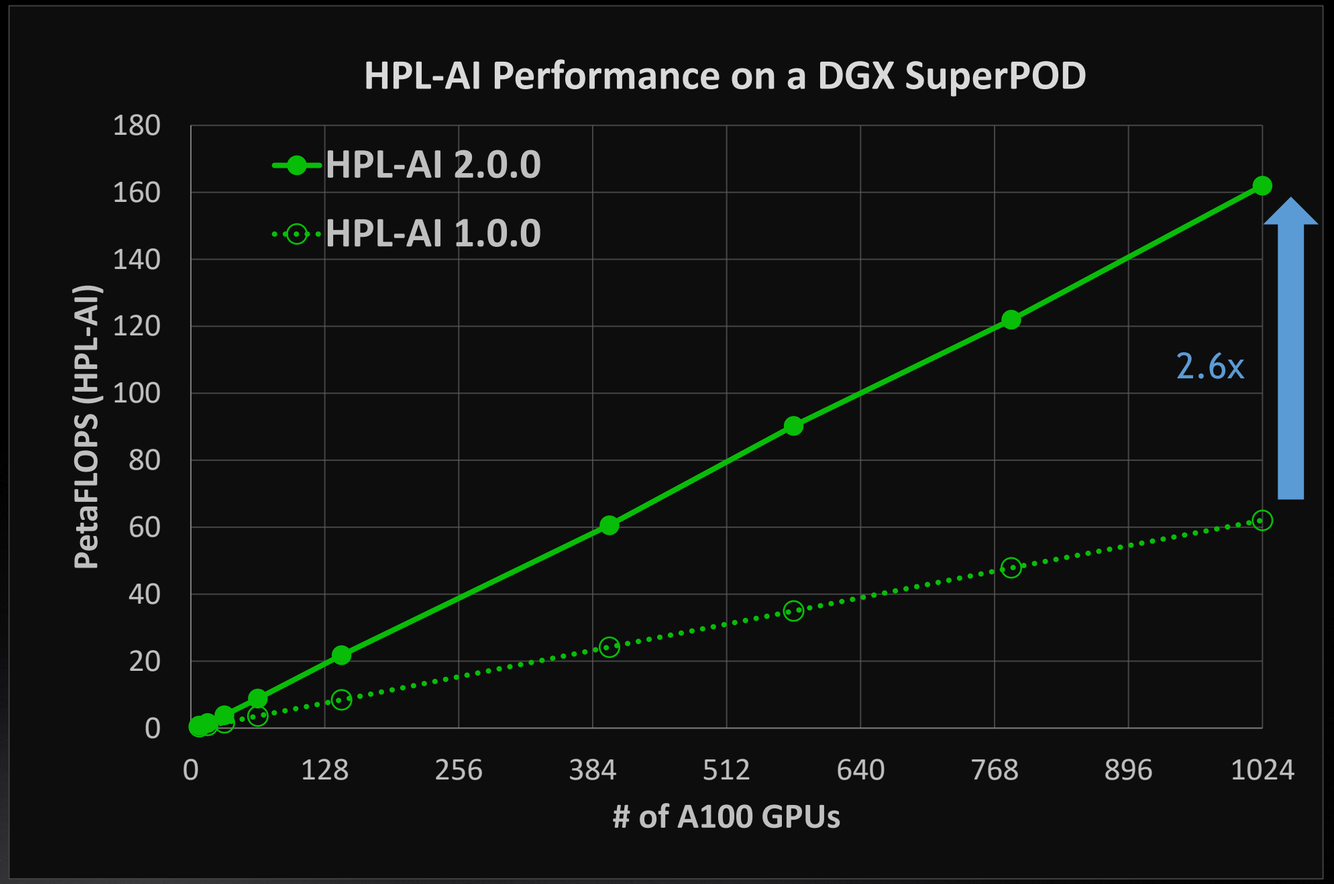
Tensor Core Accelerated Iterative Refinement Solver (TCAIRS) for HPL

First HPC Benchmarks release 2020.10 on NGC provides benchmarks with optimized for performance on NVIDIA accelerated HPC systems

- HPL
- HPL-AI
- HPCG

2021.04 release comes with HPL-AI 2.0.0 which delivers a **2.6x speed-up** on a DGX SuperPOD

Improved TCAIRS available in cuSOLVER



https://ngc.nvidia.com/catalog/containers/nvidia:hpc-benchmarks

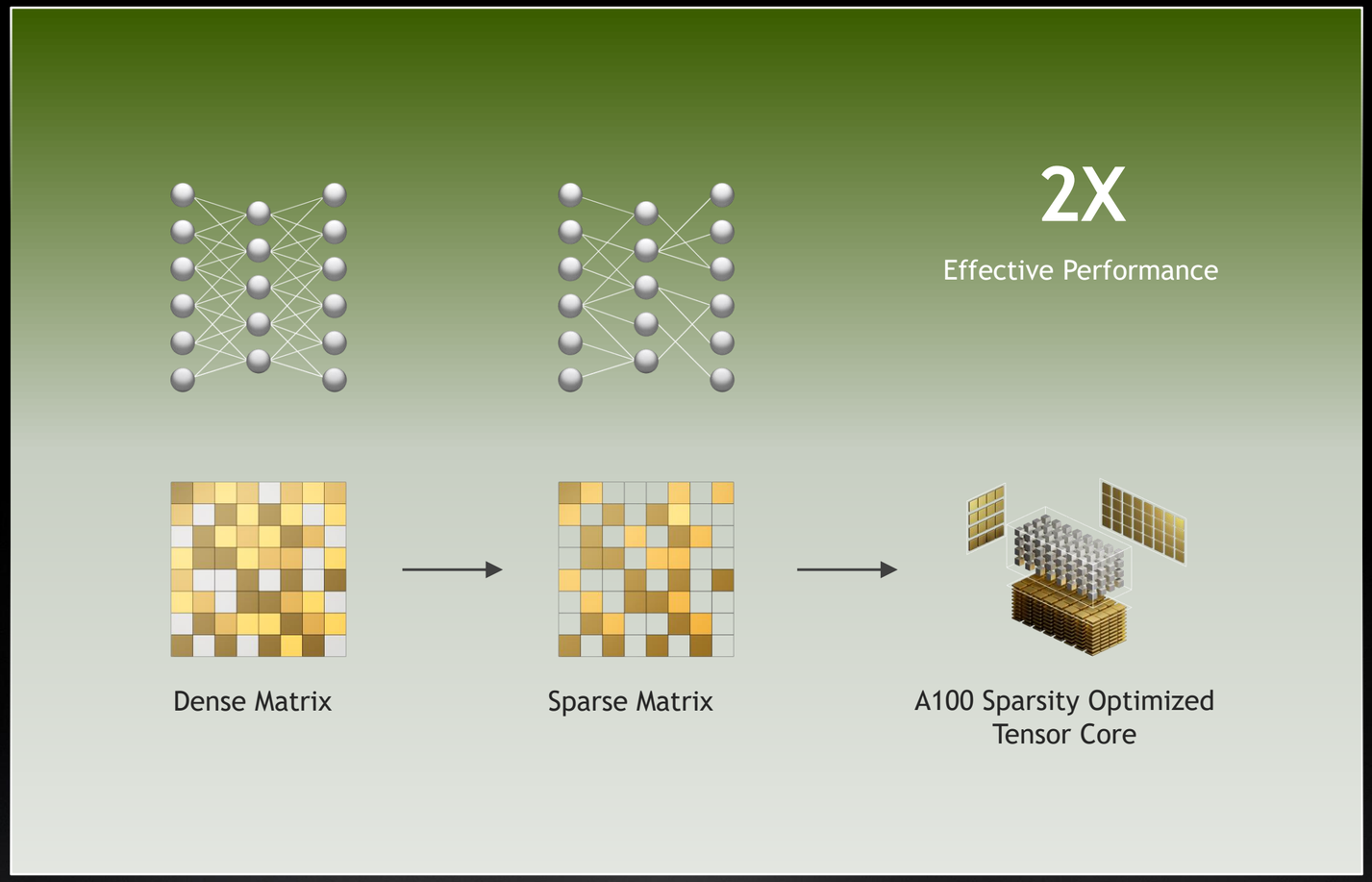
TENSOR CORE SUPPORT IN MATH LIBRARIES

High-level overview of supported functionality by each library

Library and Tensor Core Functionality	INT4		INT8		FP16		BF16		TF32		FP64
	Dense	Sparse	Dense	Sparse	Dense	Sparse	Dense	Sparse	Dense	Sparse	Dense
cuBLAS & cuBLASLt Dense GEMM			✓		✓		✓		✓		✓
cuTENSOR Tensor Contractions					✓		✓		✓		✓
cuSOLVER Linear System Solvers					✓		✓		✓		✓
cuSPARSE Block-SpMM			✓		✓		✓		✓		✓
cuSPARSELt SpMM				✓		✓		✓		✓	
CUTLASS Dense GEMM and SpMM	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CUTLASS Convolutions	✓		✓		✓		✓		✓		

Accelerating Convolution with Tensor Cores in CUTLASS [S31883]

A100 SPARSE MATRIX TENSOR CORES

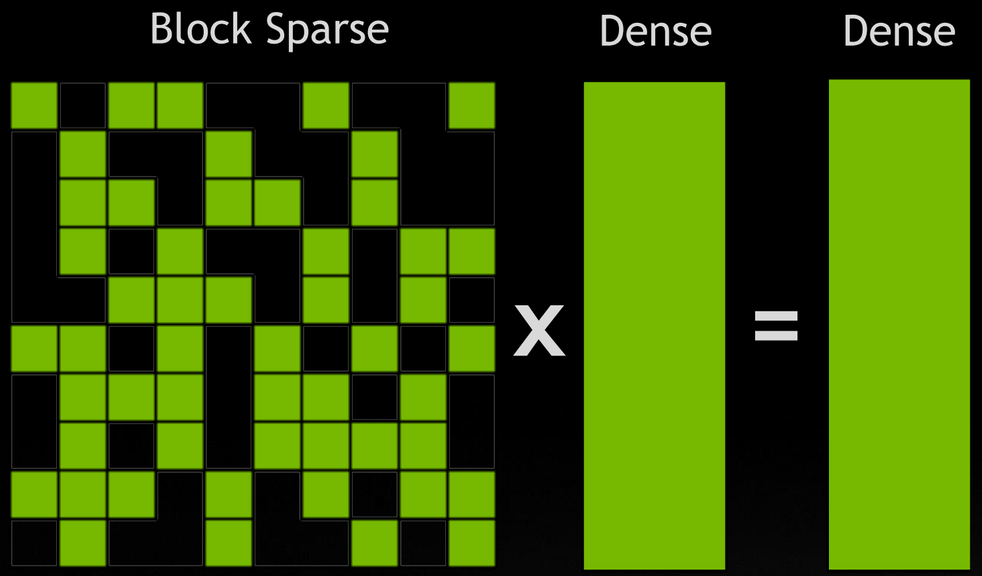


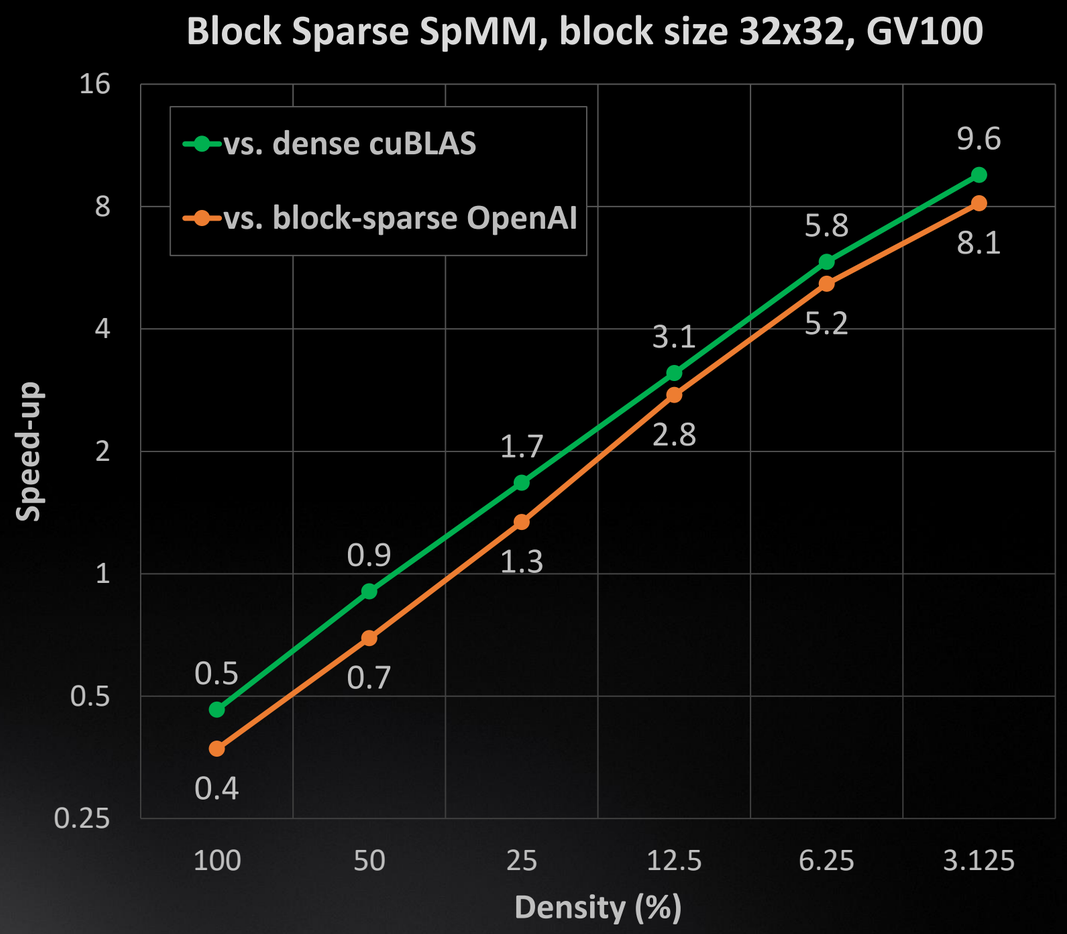
3rd Generation Tensor Cores add Sparse as well as Dense Matrix Multiply

Recent Developments in NVIDIA Math Libraries [S31754]

TENSOR CORE ACCELERATED BLOCK SPARSE SpMM

Released with cuSPARSE 11.4.0 included with CUDA Toolkit 11.2.1





Blocked ELL format used for A

Multiple block-sizes are supported

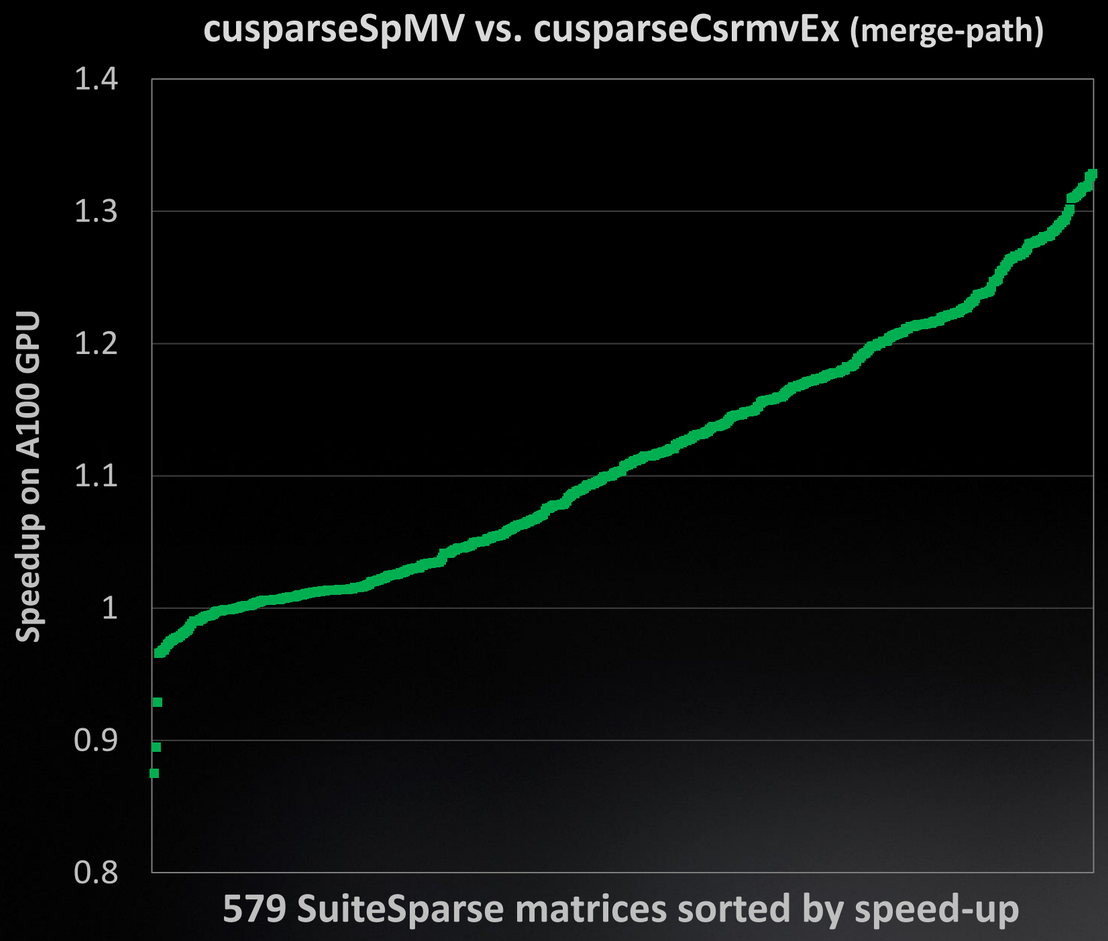
Larger block sizes perform better

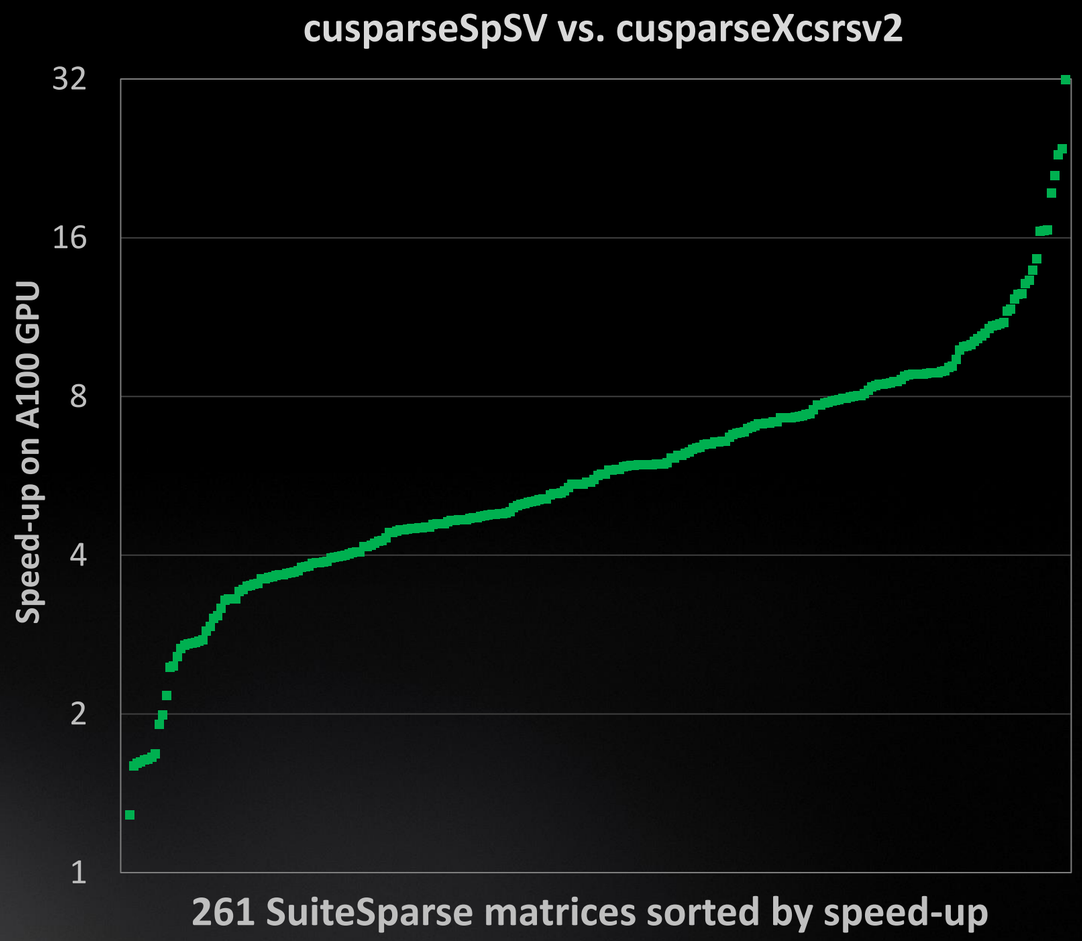
Can be adopted in deep learning models to reduce the complexity of the standard self-attention mechanism

Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores [Blog]

IMPROVED PERFORMANCE ON KEY HPC KERNELS

Sparse Matrix x Vector and Sparse Triangular Solver





A100 GPU, Upcoming release

NVIDIA

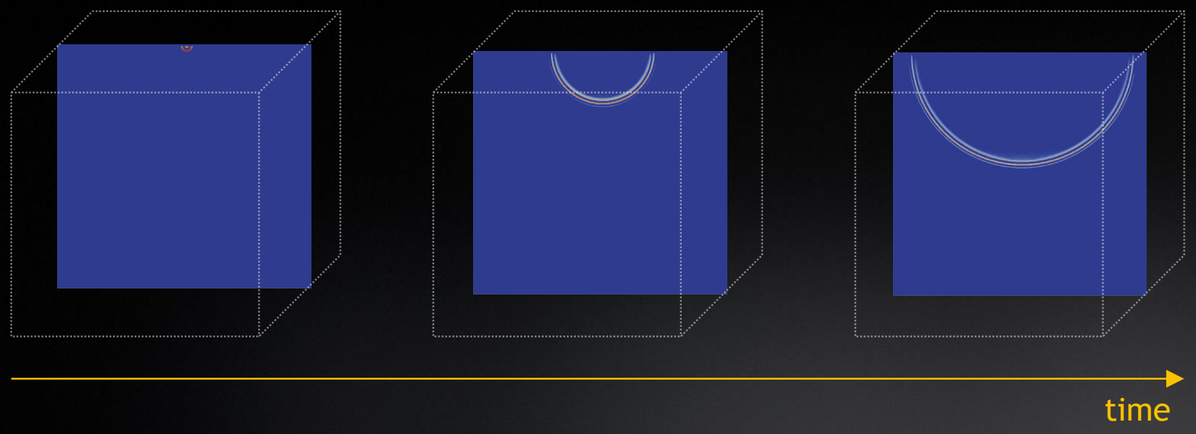
HARDWARE ACCELERATED OPTIMIZATION

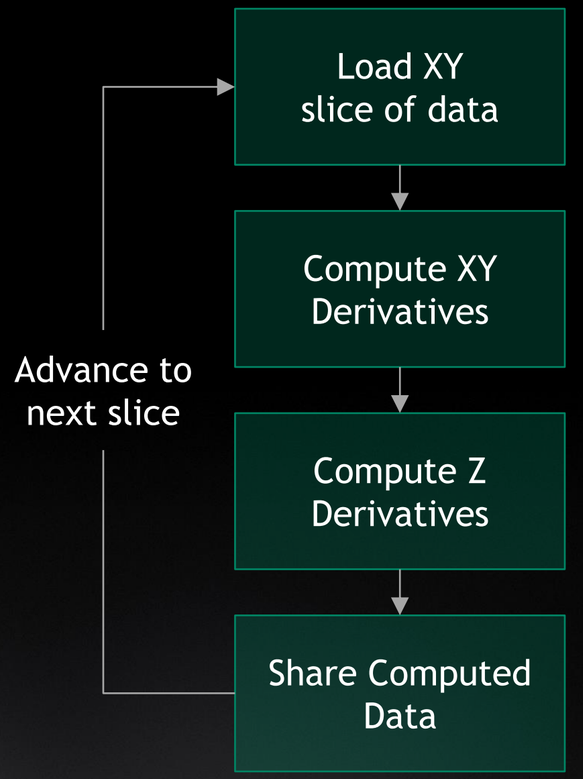
Reverse Time Migration (RTM) in Tilted Transversely Isotropic (TTI) media

Reverse Time Migration (RTM)

Seismic imaging method to analyse recorded waveforms

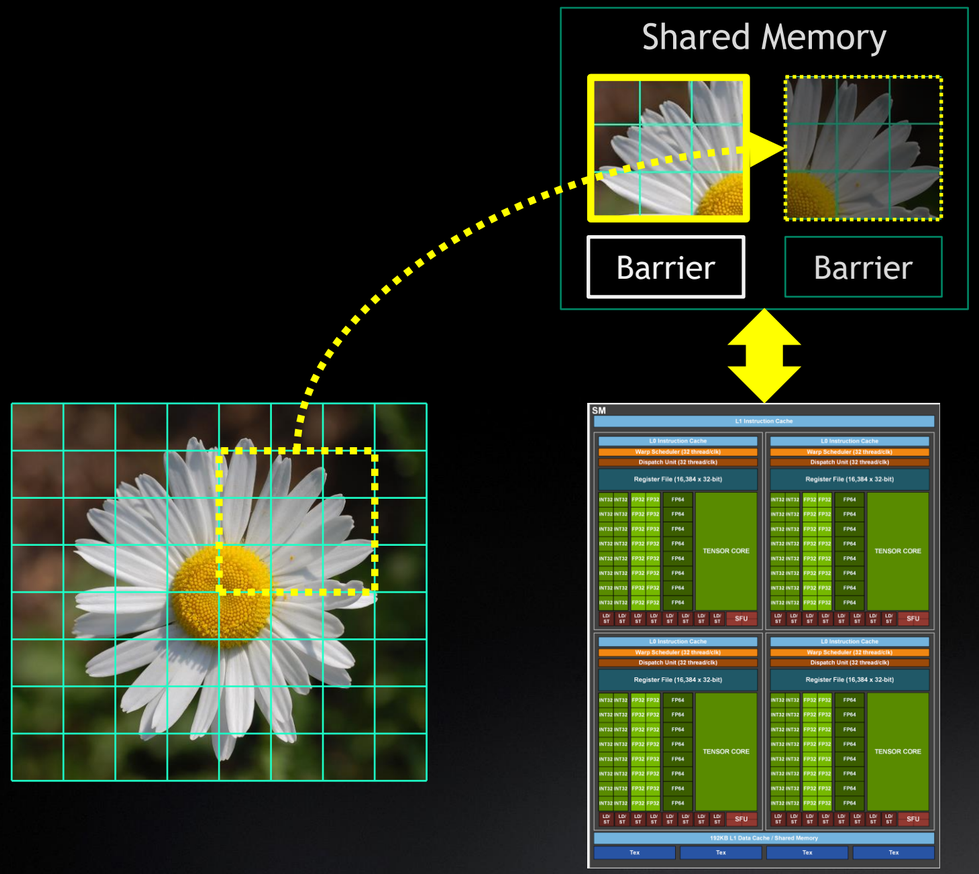
Simulates waves in 3D data to map underground structure

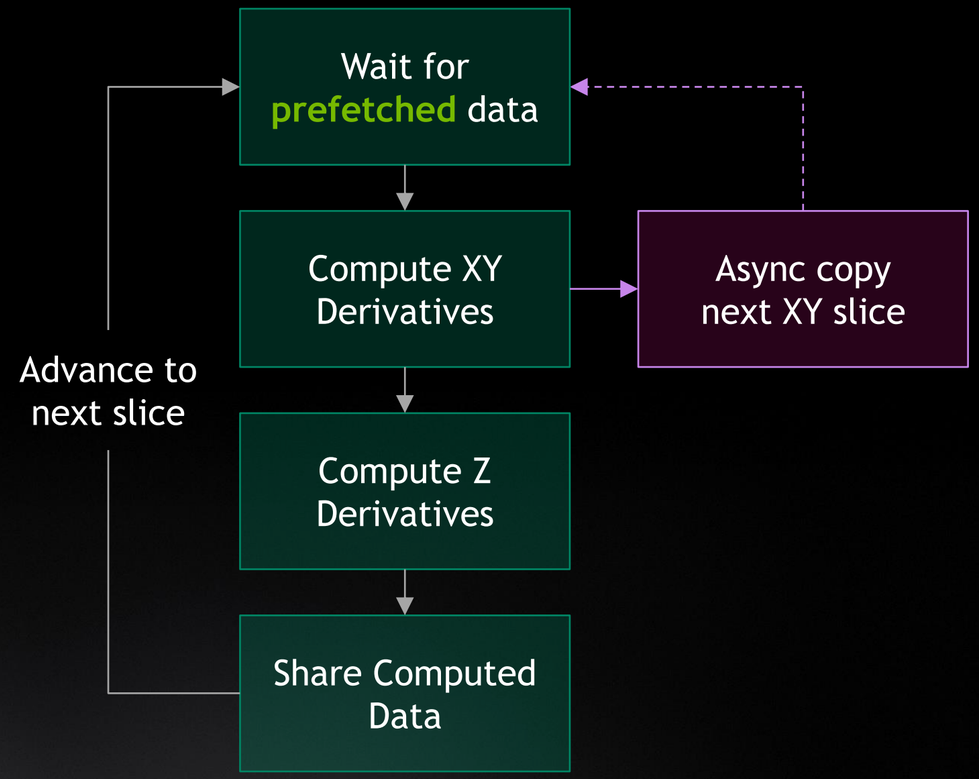




APPLYING A100's ASYNC MEMCPY & BARRIERS

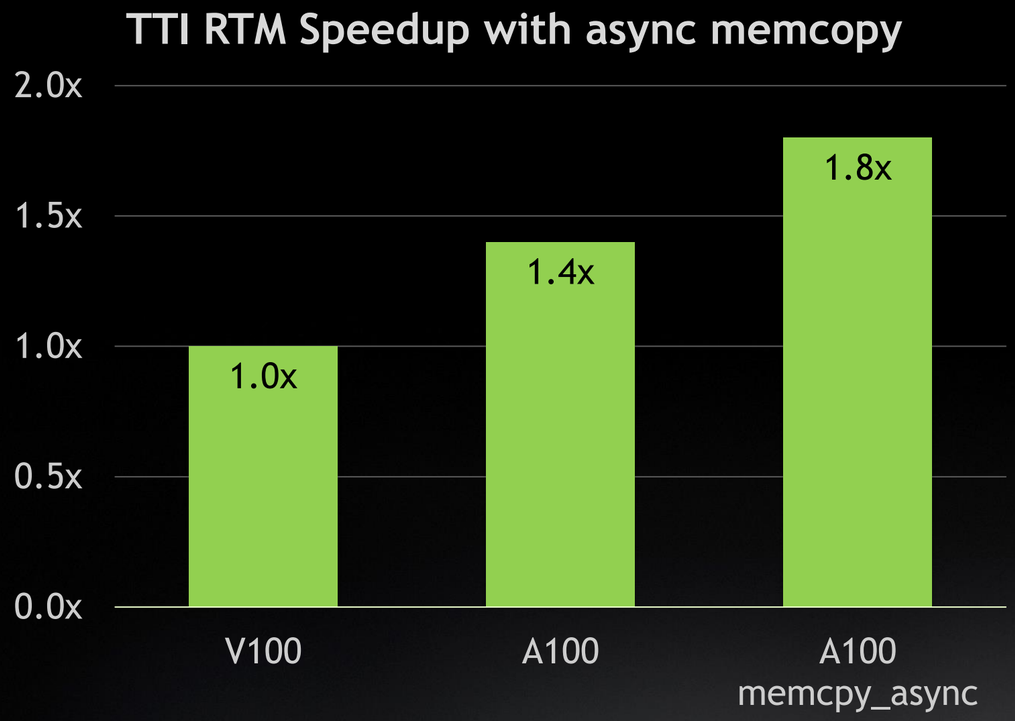
Prefetch data into shared memory

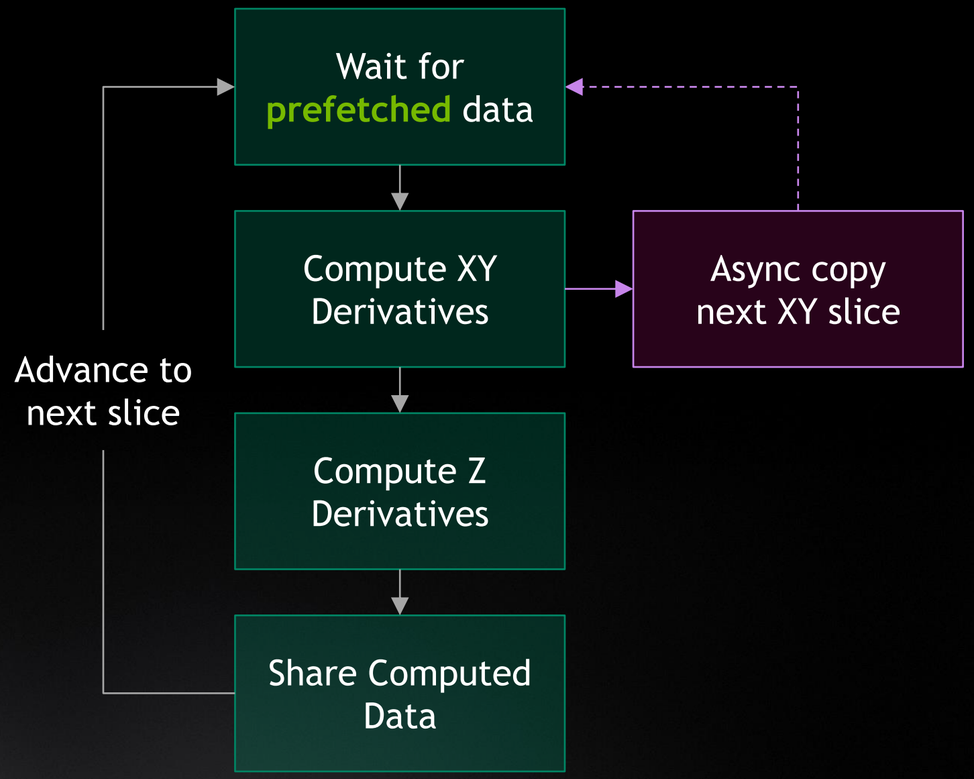




Requests, Wavefronts, Sectors Metrics: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute [S32089]

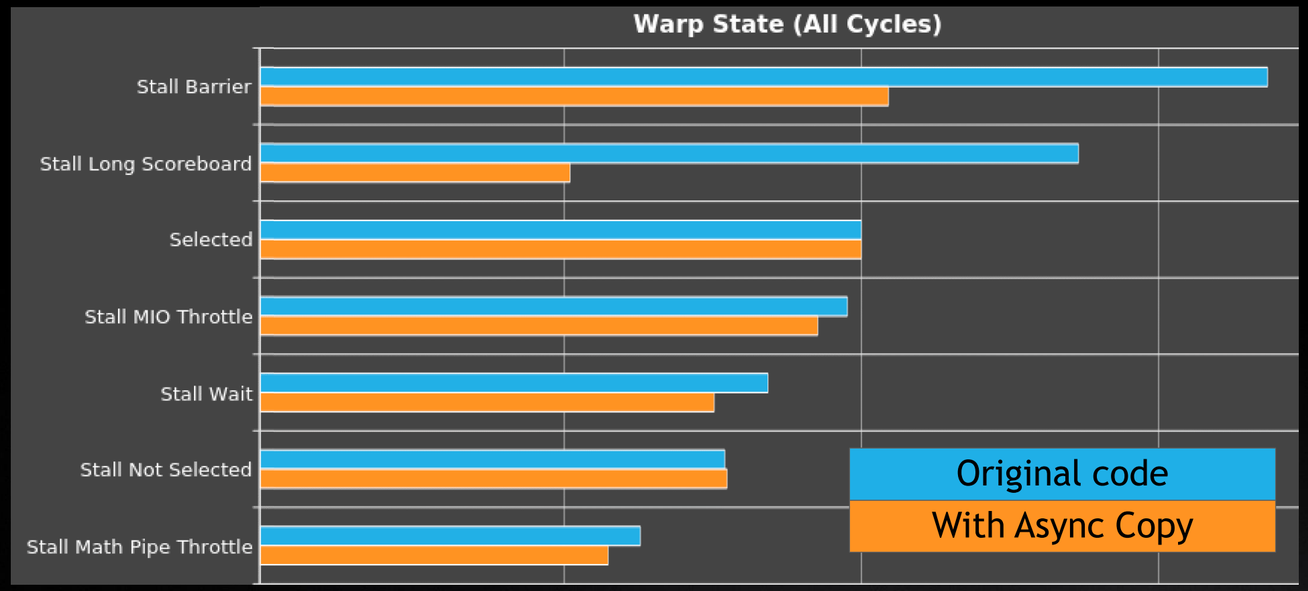
~30% SPEEDUP WITH ASYNC MEMCPY

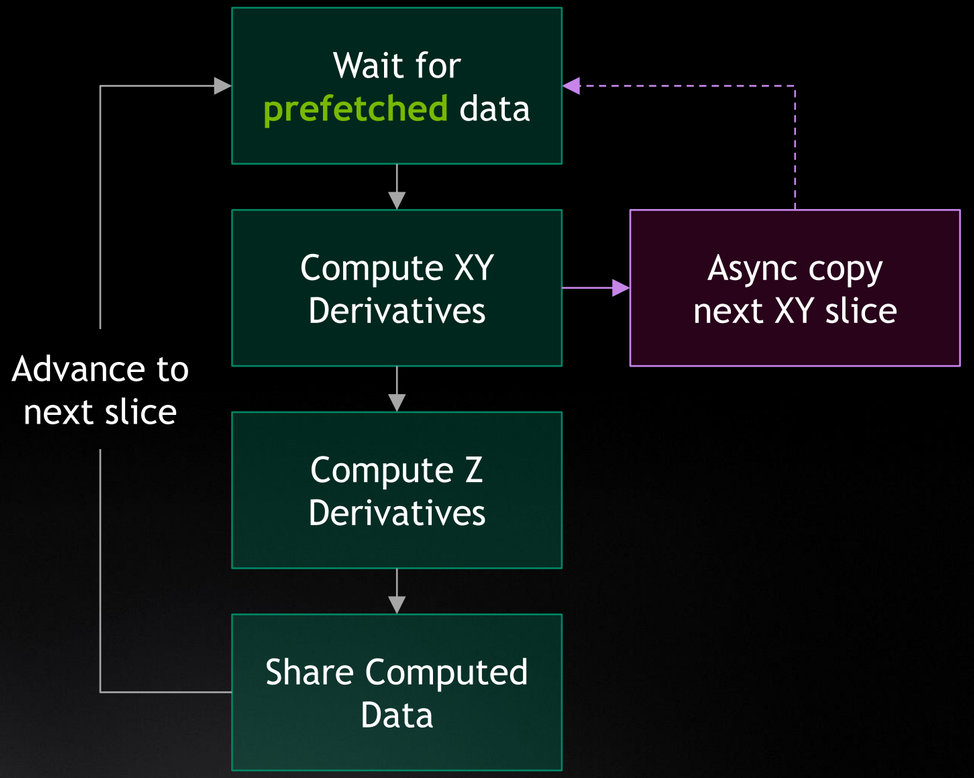




Optimizing Applications with Asynchronous GPU programming in CUDA C++ [E31888]

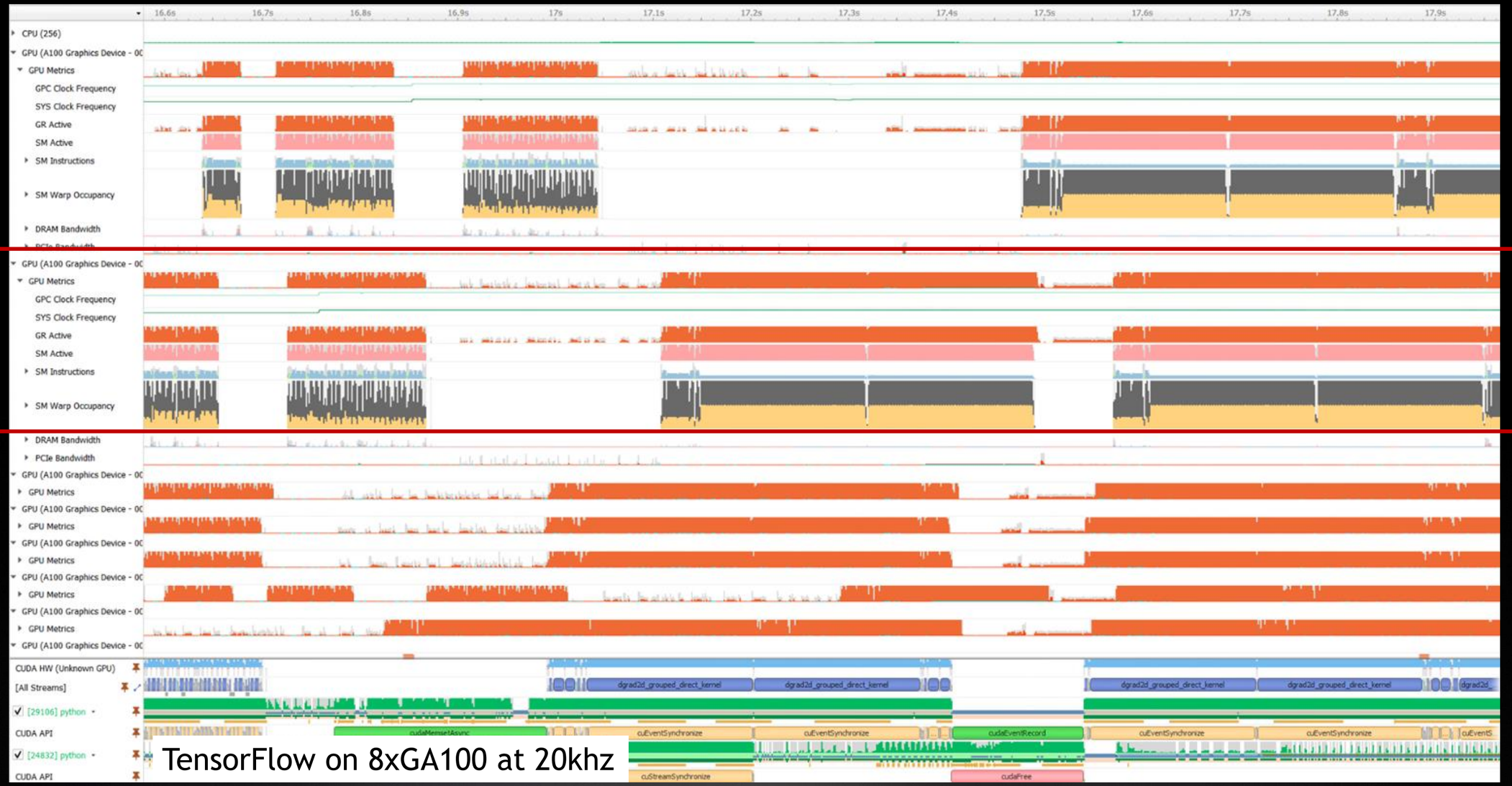
GPU METRICS TELL THE STORY





Streamline Your Development Workflow with CUDA Profiling, Optimization, and Debugging Tools [CWES1572]

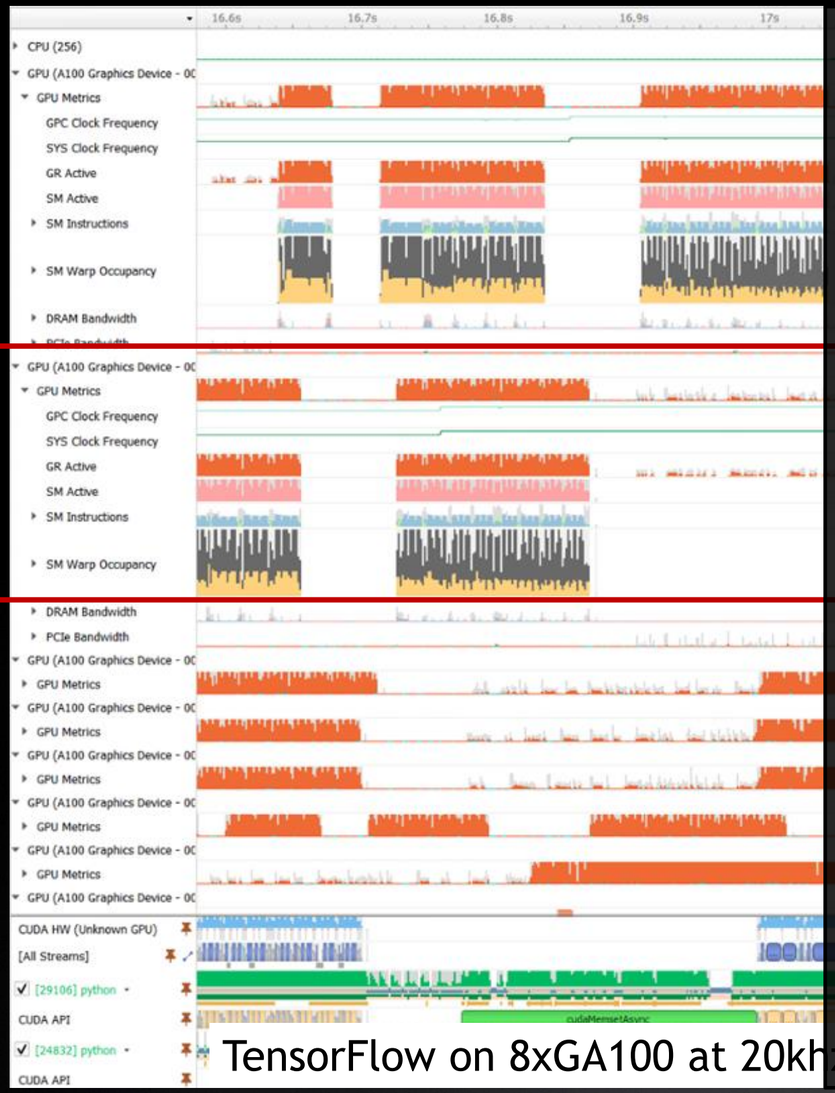
GPU METRICS SAMPLING IN NSIGHT SYSTEMS



TensorFlow on 8xGA100 at 20khz

NVIDIA

GPU METRICS SAMPLING IN NSIGHT SYSTEMS



TensorFlow on 8xGA100 at 20khz

Hardware metrics collected at runtime to answer

- Is my GPU full? Sufficient grid size & streams?
- Is my instruction rate low (possibly I/O bound)?
- Am I using TensorCores?
- Can I see GPU Direct RDMA|Storage or other transfers?

System-wide GPU observation

10khz default can be increased depending on GPU

SM utilization metrics	IO throughput metrics
SMs active	PCIe
Instructions	NVLink
TensorCores	DRAM
Warp occupancy	
Unallocated slots	

COOPERATIVE GROUPS MULTI-WARP GROUPS

`cg::experimental::block_tile_memory`

Takes in size of memory to allocate for communication across tiles (e.g. for reductions) and the number of threads in your block.

`cg::experimental::this_thread_block(scratch)`

Takes in the scratch memory and associates it with the resulting group handle.

Takes user-provided memory for barriers/collectives

Shared provides best performance, but can also be in global memory.

Experimental in CUDA 11.1+
Support for thread_block_tile (s) of 64/128/256/512

```
// 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::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;
});
```

SIMULTANEOUS COOPERATIVE LAUNCH

Prior to CUDA 11.2

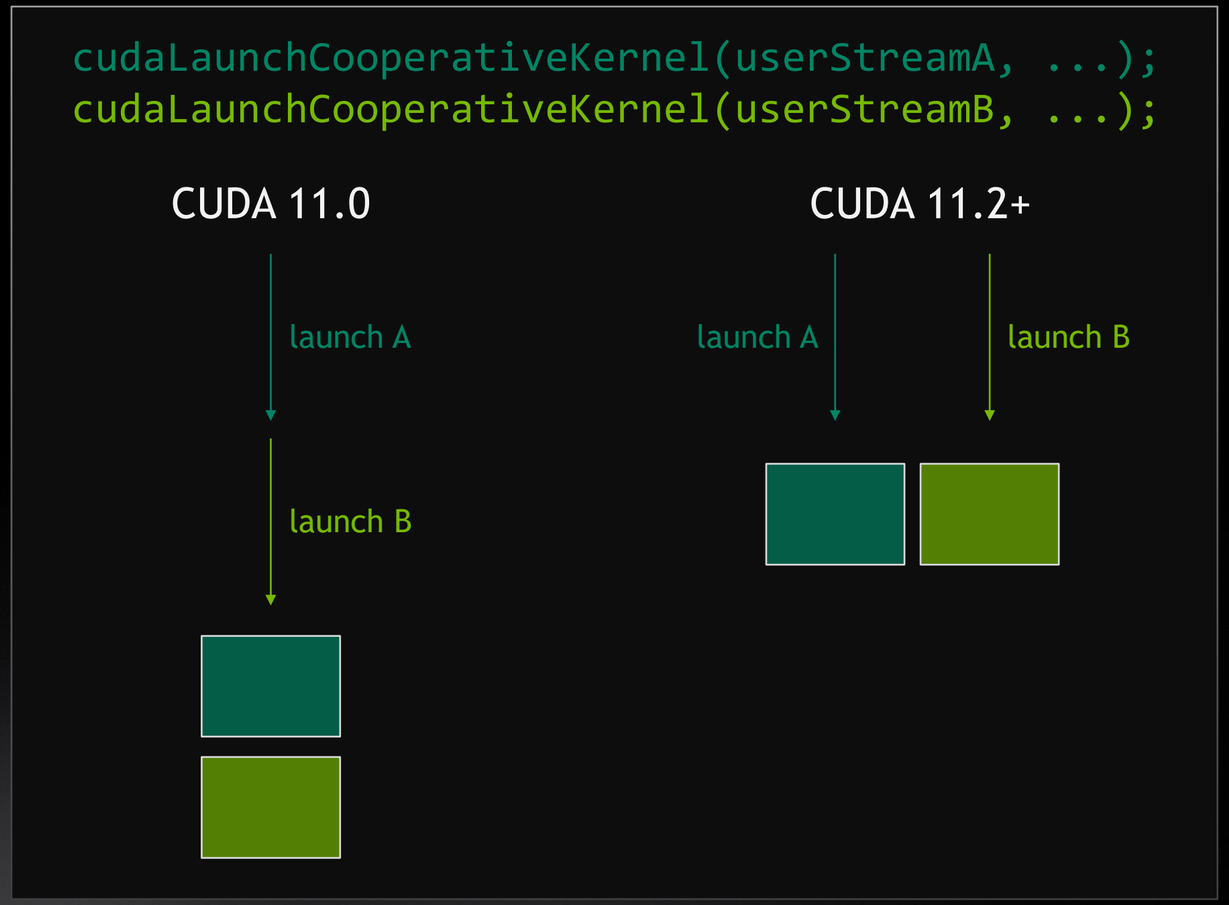
Launches through cudaLaunchCooperativeKernel only permitted one cooperative launch at a time

Since CUDA 11.2

Multiple cooperative grids may be resident at the same time, allowing concurrent multi-block grids

Cooperative launch now also supported in CUDA graphs

Same restrictions on resources apply – check occupancy before launching



THREAD-LOCAL DEVICE MEMORY ALLOCATION

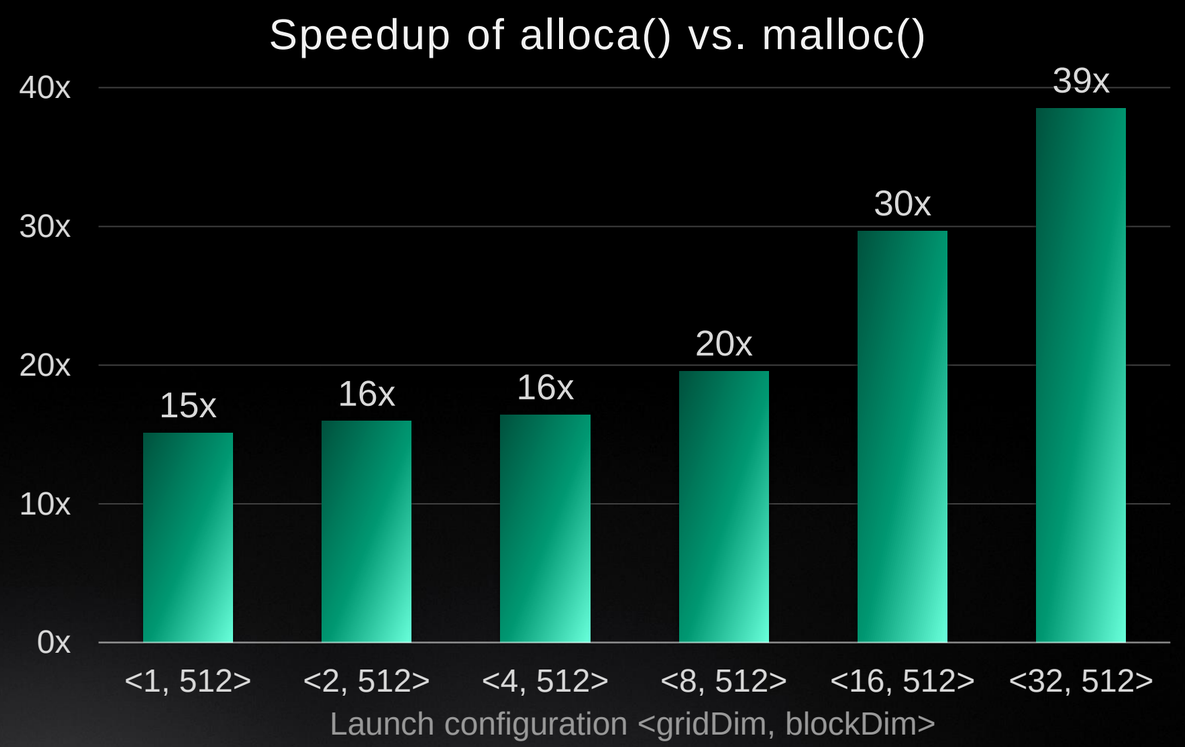
alloca()

Allocates data private to a thread, on its stack; only the allocating thread can access data.

Faster than *malloc()* which uses a global heap but data can be accessed by all threads.

To manage per-thread stack size, set the **StackSize** device limit:

```
cudaDeviceSetLimit( cudaLimitStackSize, [stack_size] )
```



EXPANDED C++ LANGUAGE SUPPORT

Support for *'constexpr'* and *'auto'*

Can now use *constexpr* for __device__ and __constant__ variables, including where constant expressions are not required

Can now use *auto* for __device__ variables except for the __shared__ memory space

```
namespace N1 { namespace N2 { struct longStructName { int x; }; } }
constexpr __device__ N1::N2::longStructName foo() {
      return N1::N2::longStructName{10};
}
__device__ auto x = foo;   // x has 'int' type
```

C++ AND CUDA

ISO C++ == Language + Standard Library

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

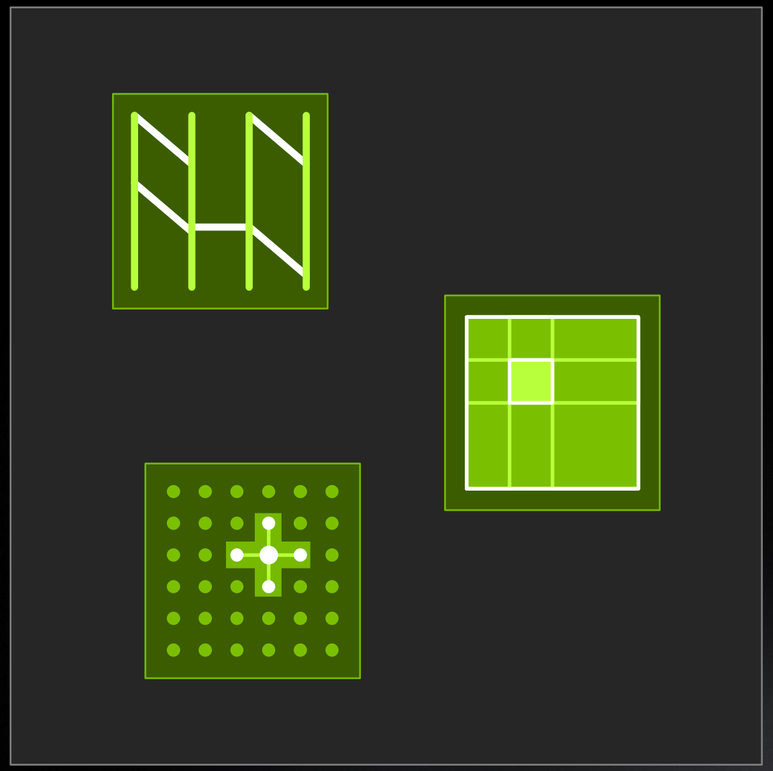
Strictly conforming to ISO C++, plus conforming extensions

Opt-in, Heterogeneous, Incremental

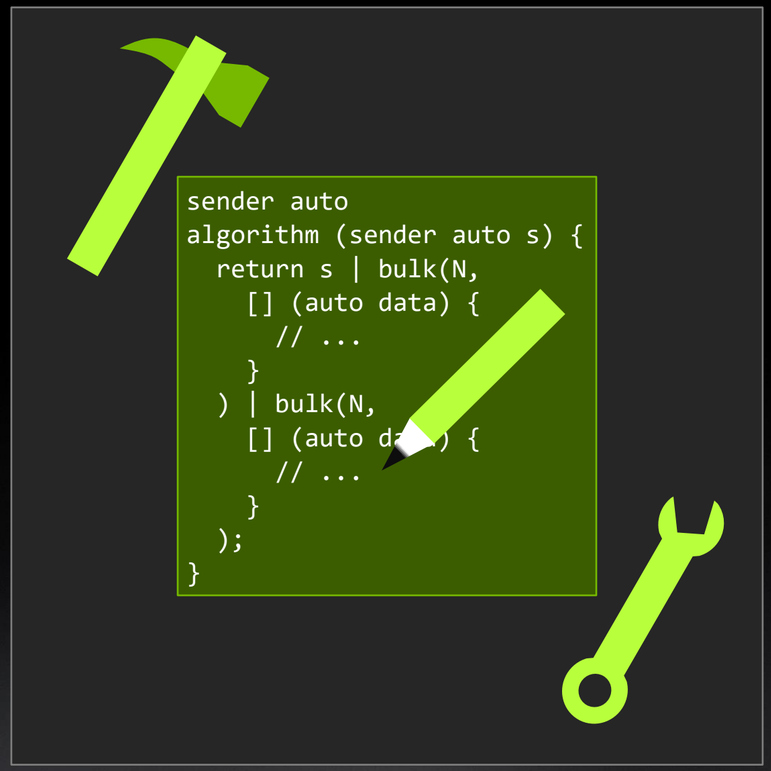
The NVIDIA C++ Standard Library [S31359]

C++ STANDARD PARALLELISM

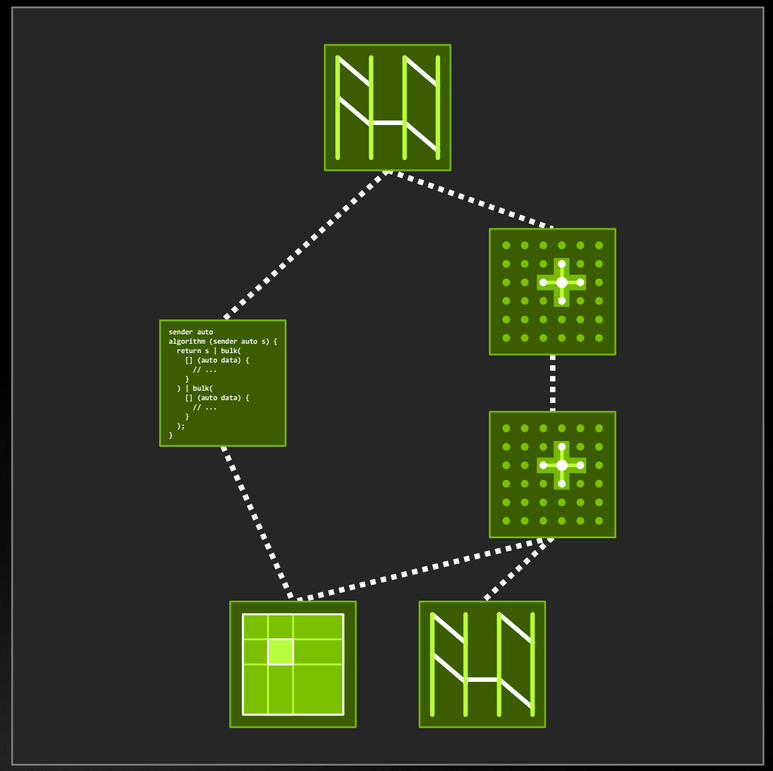
Common Algorithms that Dispatch to Vendor-Optimized Parallel Libraries



Tools to Write Your Own Parallel Algorithms that Run Anywhere



Mechanisms for Composing Parallel Invocations into Task Graphs



C++ STANDARD PARALLELISM

C++17

Parallel Algorithms

In NVC++

With parallel and vector concurrency

Forward Progress Guarantees

Extend the C++ execution model for accelerators

Memory Model Clarifications

Extend the C++ memory model for accelerators

C++20

Scalable Synchronization Library

Express thread synchronization that is portable and scalable across CPUs and accelerators

In libcu++:

- `std::atomic<T>`
- `std::barrier`
- `std::counting_semaphore`
- `std::atomic<T>::wait/notify_*`
- `std::atomic_ref<T>`

C++23 and Beyond

Executors

Simplify launching and managing parallel work across CPUs and accelerators

`std::mdspan/mdarray`

HPC-oriented multi-dimensional array abstractions.

Linear Algebra

Modern C++ linear algebra interface

Maps to vendor optimized BLAS libraries

Extended Floating Point Types

First-class support for formats new and old: `std::float16_t/float64_t`

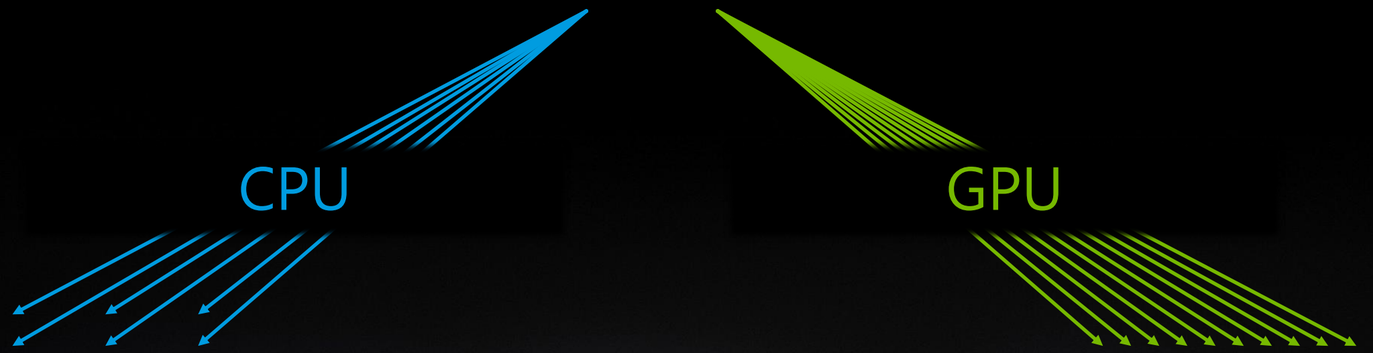
Future of Standard and CUDA C++ [CWES1802]

STANDARD FORTRAN DO CONCURRENT

```
do concurrent (j=1:N, i=1:N)
  B(j,i) = A(i,j)
enddo
```

STANDARD FORTRAN DO CONCURRENT

```
do concurrent (j=1:N, i=1:N)
  B(j,i) = A(i,j)
enddo
```



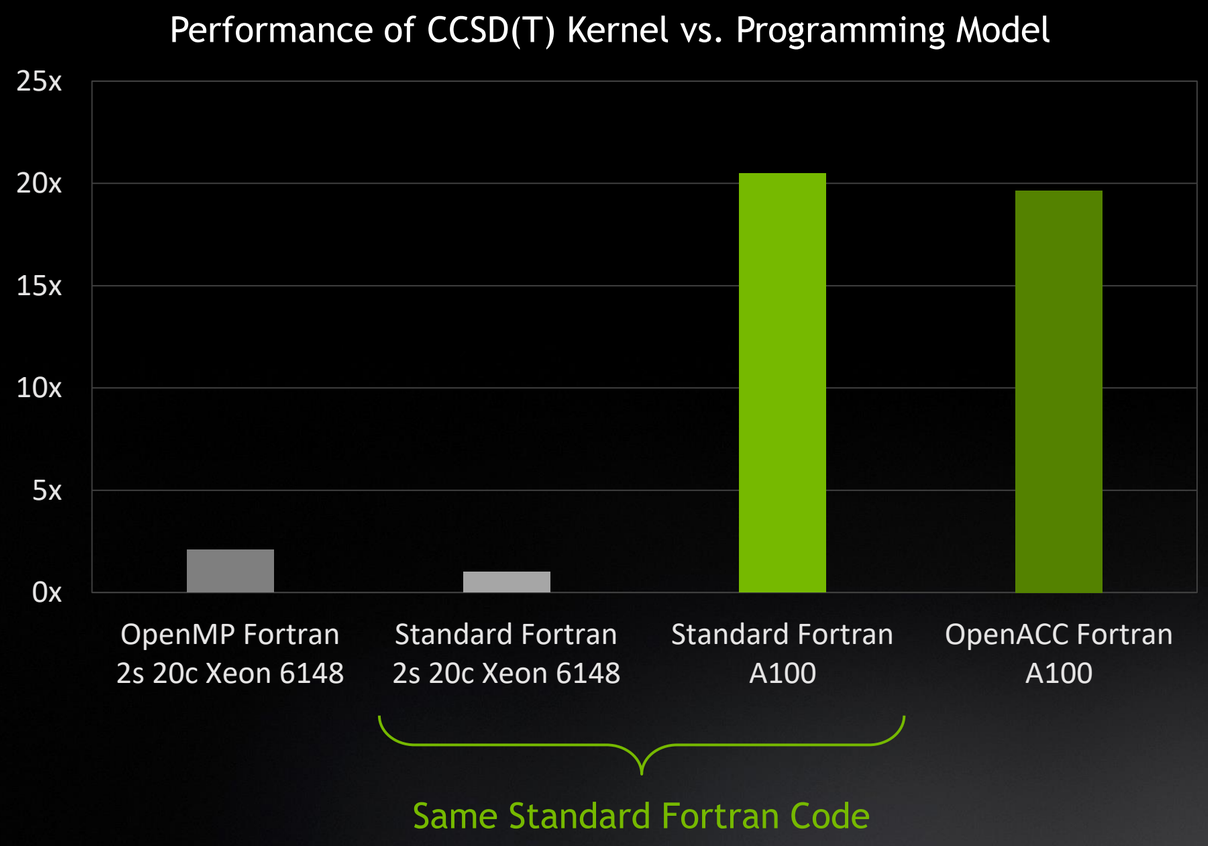

```
nvfortran –stdpar=multicore
```

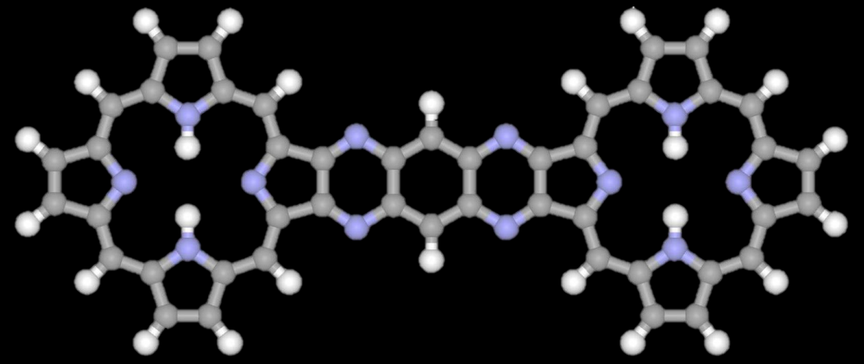
```
nvfortran –stdpar=gpu
```

Inside NVC++ and NVFORTRAN [S31358]

NWCHEM TCE CCSD(T) KERNEL

Computational Chemistry with Fortran Standard Parallelism



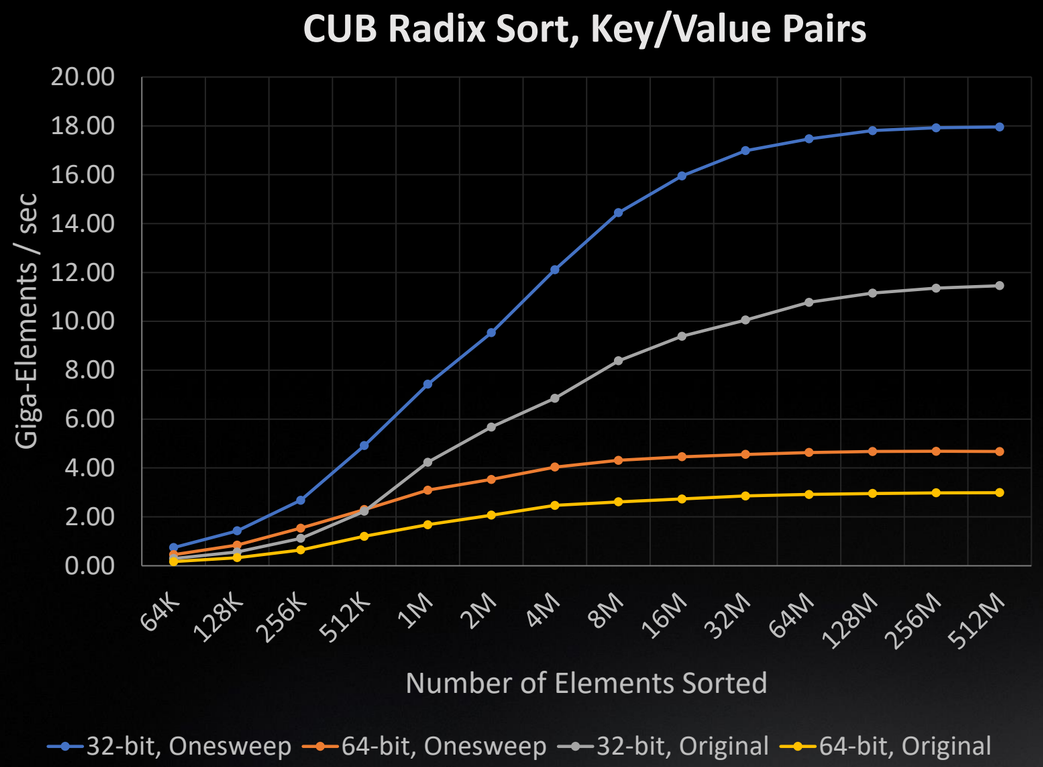


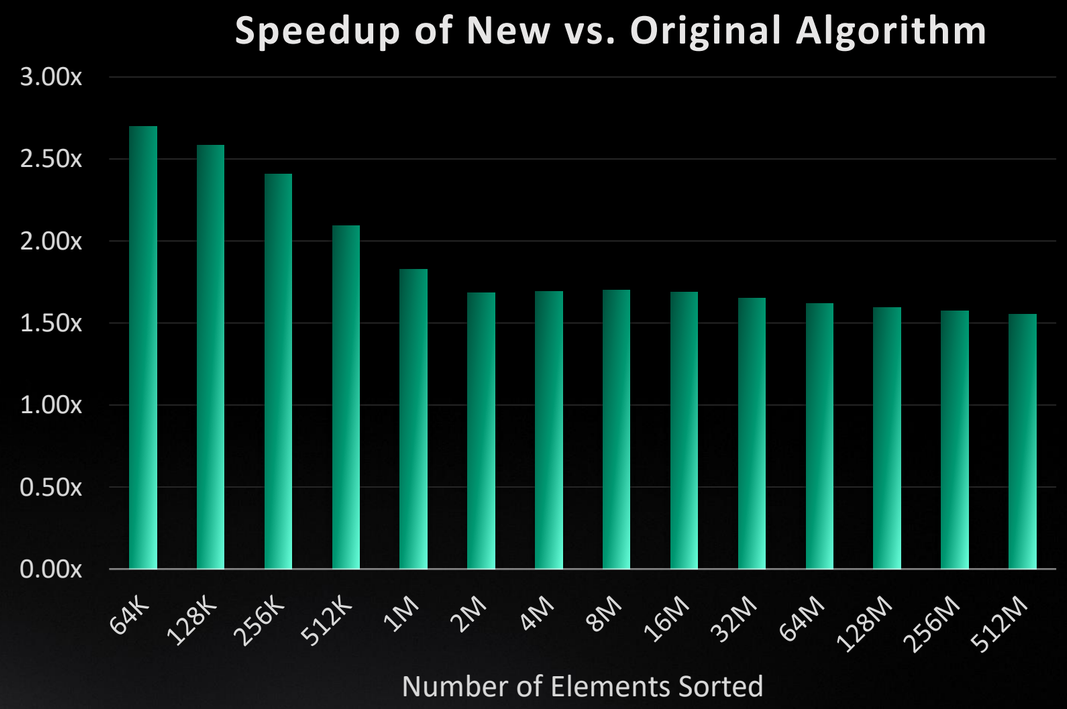
NWChem provides a massively parallel implementation of the "gold standard" CCSD(T) method that scales to hundreds of thousands of CPU cores.

The compute bottleneck is a set of 27 loop-driven tensor contractions, which are part of the >100k LOC TCE module.

https://github.com/jeffhammond/nwchem-tce-triples-kernels

CUB RADIX SORT: NEW "*ONESWEEP*" ALGORITHM





A Faster Radix Sort Implementation

Thrust, CUB, and libcu++ User's Forum [CWES1801]

NVIDIA

THE TRAGEDY OF THE COMMONS

IMPERATIVE VS. ASYNCHRONOUS PROGRAMMING

Imperative

```
void run() {
      x();
      y();
      z();
}
```

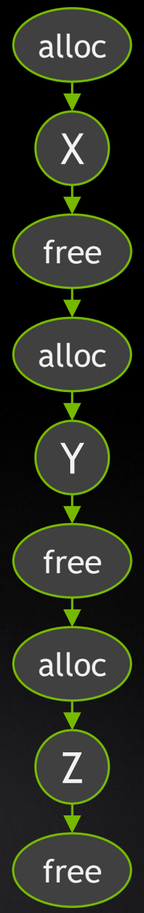
Asynchronous

```
void cuda_run() {
      x<<< ... >>>();
      y<<< ... >>>();
      z<<< ... >>>();
      cudaDeviceSynchronize();
}
```

IMPERATIVE VS. ASYNCHRONOUS MEMORY ALLOCATION

```
void run() {
      void *x_ptr = malloc(N);
      x(x_ptr);
      free(x_ptr);

      void *y_ptr = malloc(N*2);
      y(y_ptr);
      free(y_ptr);

      void *z_ptr = malloc(N/2);
      z(z_ptr);
      free(z_ptr);
}
```

```
void cuda_run() {
      cudaMalloc(&x_ptr, N);
      x<<< ... >>>(x_ptr);
      cudaFree(x_ptr);

      cudaMalloc(&y_ptr, N*2);
      y<<< ... >>>(y_ptr);
      cudaFree(y_ptr);

      cudaMalloc(&z_ptr, N/2);
      z<<< ... >>>(z_ptr);
      cudaFree(z_ptr);

      cudaDeviceSynchronize();
}
```

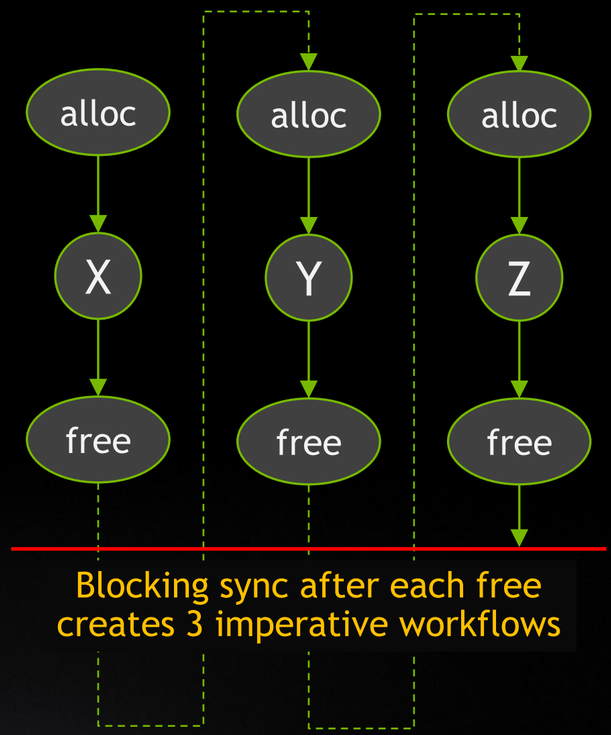


BUT ... THERE IS NO ASYNCHRONOUS “FREE”

cudaFree() must wait for kernel “x” to finish before freeing x_ptr

Our asynchronous program has become imperative

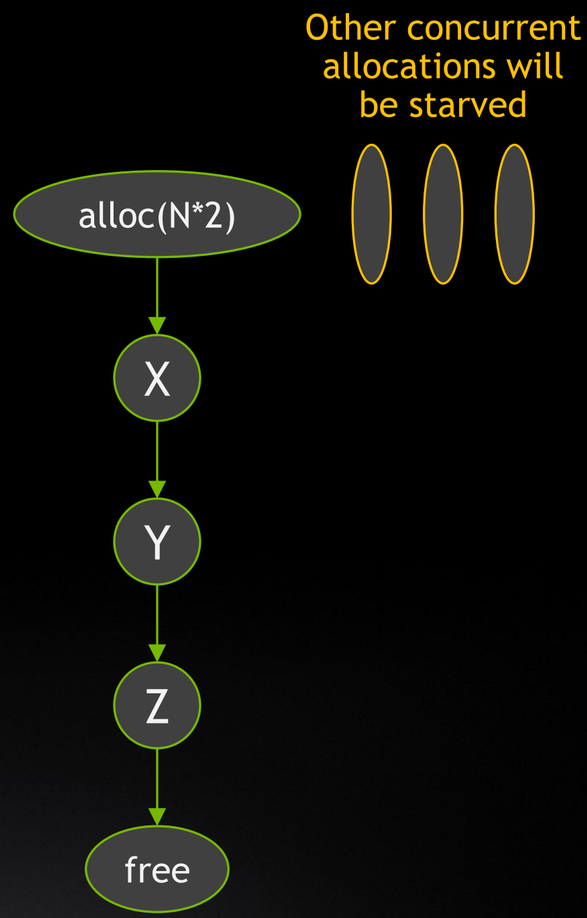
```
void cuda_run() {
      cudaMalloc(&x_ptr, N);
      x<<< ... >>>(x_ptr);
      cudaFree(x_ptr);

      cudaMalloc(&y_ptr, N*2);
      y<<< ... >>>(y_ptr);
      cudaFree(y_ptr);

      cudaMalloc(&z_ptr, N/2);
      z<<< ... >>>(z_ptr);
      cudaFree(z_ptr);

      cudaDeviceSynchronize();
}
```



THE TRAGEDY OF THE GLOBAL MEMORY COMMONS

```
void run() {
      void *ptr = malloc(N*2);

      x(ptr);
      y(ptr);
      z(ptr);

      free(z_ptr);
}
```

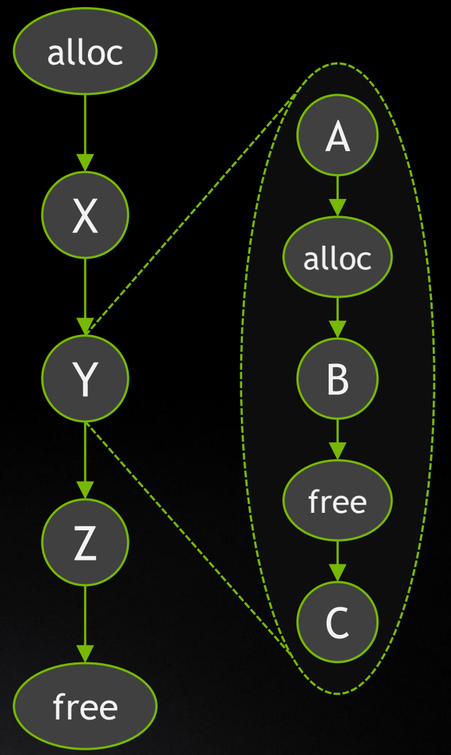
```
void cuda_run() {
      cudaMalloc(&ptr, N*2);

      x<<< ... >>>(ptr);
      y<<< ... >>>(ptr);
      z<<< ... >>>(ptr);
      cudaDeviceSynchronize();

      cudaFree(ptr);
}
```



ON THE NON-COMPOSABILTY OF GLOBAL RESOURCES

```
void run() {
      void *ptr = malloc(N*2);

      x(ptr);
      y(ptr);
      z(ptr);

      free(z_ptr);
}

void y(void *ptr) {
      a();
      void *b_ptr = malloc(N*3);
      b(b_ptr);
      free(b_ptr);
      c();
}
```

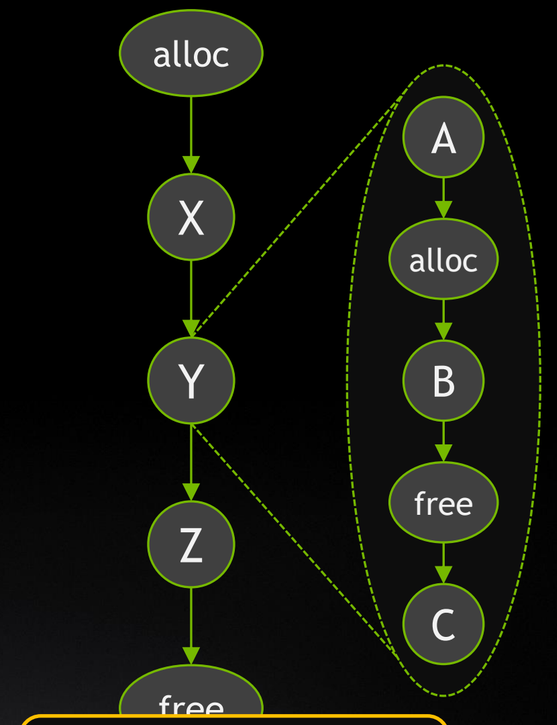
```
void cuda_run() {
      cudaMalloc(&ptr, N*2);

      x<<< ... >>>(ptr);
      y(ptr);
      z<<< ... >>>(ptr);
      cudaDeviceSynchronize();

      cudaFree(ptr);
}

void y(void *ptr) {
      a<<< ... >>>();
      cudaMalloc(&b_ptr, N*3);
      b<<< ... >>>(b_ptr);
      cudaFree(b_ptr);
      c<<< ... >>>();
}
```



THE TRAGEDY OF THE ASYNCHRONOUS COMMONS

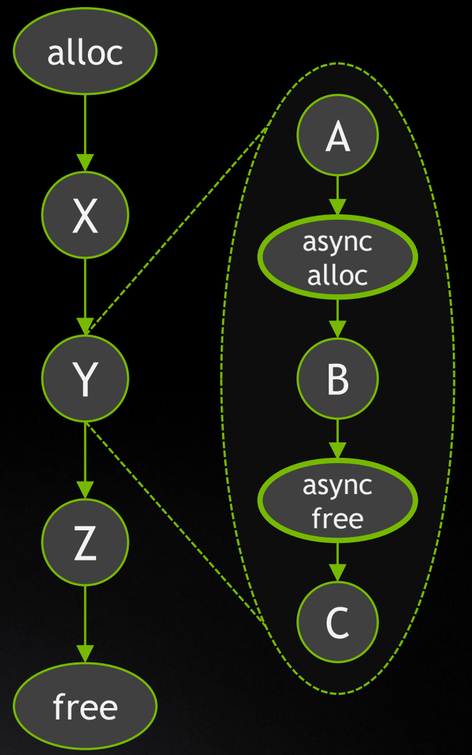
```
void run() {
      void *ptr = malloc(N*2);

      x(ptr);
      y(ptr);
      z(ptr);

      free(z_ptr);
}

void y(void *ptr) {
      a();
      void *b_ptr = malloc(N*3);
      b(b_ptr);
      free(b_ptr);
      c();
}
```

```
void cuda_run() {
      cudaMalloc(&ptr, N*2);

      x<<< ... >>>(ptr);
      y(ptr);
      z<<< ... >>>(ptr);
      cudaDeviceSynchronize();

      cudaFree(ptr);
}

void y(void *ptr) {
      a<<< ... >>>();
      cudaMalloc(&b_ptr, N*3);
      b<<< ... >>>(b_ptr);
      cudaFree(b_ptr);
      c<<< ... >>>();
}
```



STREAM ORDERED MEMORY ALLOCATION

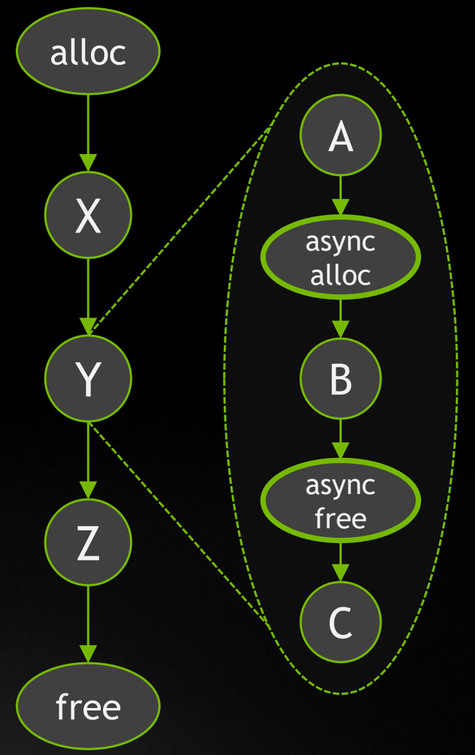
```
void cuda_run() {
      cudaMalloc(&ptr, N*2);

      x<<< ... >>>(ptr);
      y(ptr);
      z<<< ... >>>(ptr);
      cudaDeviceSynchronize();

      cudaFree(ptr);
}

void y(void *ptr) {
      a<<< ... >>>();
      cudaMallocAsync(&b_ptr, N*3, stream);
      b<<< ... >>>(b_ptr);
      cudaFreeAsync(b_ptr, stream);
      c<<< ... >>>();
}
```



STREAM ORDERED MEMORY ALLOCATION

```
void cuda_run() {
      cudaMalloc(&ptr, N*2);

      x<<< ... >>>(ptr);
      y(ptr);
      z<<< ... >>>(ptr);
      cudaDeviceSynchronize();

      cudaFree(ptr);
}

void y(void *ptr) {
      a<<< ... >>>();
      cudaMallocAsync(&b_ptr, N*3, stream);
      b<<< ... >>>(b_ptr);
      cudaFreeAsync(b_ptr, stream);
      c<<< ... >>>();
}
```

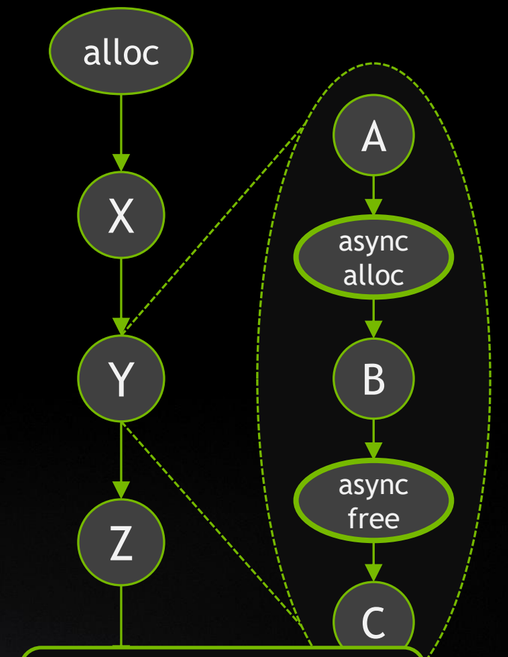
Preserves asynchrony



STREAM ORDERED MEMORY ALLOCATION

```
void cuda_run() {
      cudaMalloc(&ptr, N*2);

      x<<< ... >>>(ptr);
      y(ptr);
      z<<< ... >>>(ptr);
      cudaDeviceSynchronize();

      cudaFree(ptr);
}

void y(void *ptr) {
      a<<< ... >>>();
      cudaMallocAsync(&b_ptr, N*3, stream);
      b<<< ... >>>(b_ptr);
      cudaFreeAsync(b_ptr, stream);
      c<<< ... >>>();
}
```

Preserves asynchrony



The allocation is now local, not global

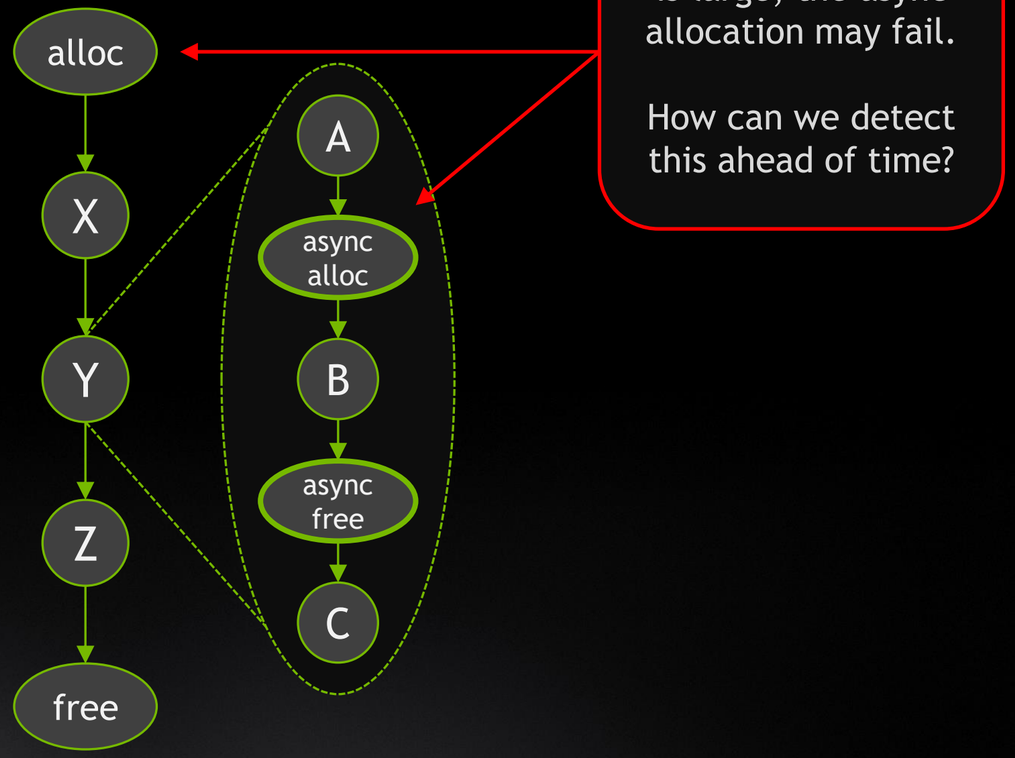
THE CHALLENGE OF ASYNCHRONOUS ALLOCATION

Two Conflicting Goals

Asynchronous allocation has two conflicting requirements:

1. To maintain asynchrony, *allocate* and *free* must be done **in stream order** instead of globally

2. But deferred allocation **is not guaranteed to succeed** - a program risks being out of memory at runtime



PROGRAMMING MODEL

```
cudaMallocAsync(&ptr1, size, stream);       // Allocates physical memory
kernel_1<<<...,stream>>>(ptr1);
cudaFreeAsync(ptr1, stream);                // Doesn’t release memory

cudaMallocAsync(&ptr2, size, stream);       // Reuses previously freed pointer
kernel_2<<<...,stream>>>(ptr2);
cudaFreeAsync(ptr2, stream);                // Doesn’t release memory

cudaStreamSynchronize(stream);              // Releases memory that exceeds cached threshold
```

CUDA Memory Management [CWES1175]

PROGRAMMING MODEL

1. cudaMallocAsync returns an error immediately on future out-of-memory

2. Pointer is returned immediately otherwise

```
cudaMallocAsync(&ptr1, size, stream);
kernel_1<<<...,stream>>>(ptr1);
cudaFreeAsync(ptr1, stream);

cudaMallocAsync(&ptr2, size, stream);
kernel_2<<<...,stream>>>(ptr2);
cudaFreeAsync(ptr2, stream);

cudaStreamSynchronize(stream);
```

3. Pointer can be passed to downstream kernels and memcopies

4. No implicit synchronization when freeing

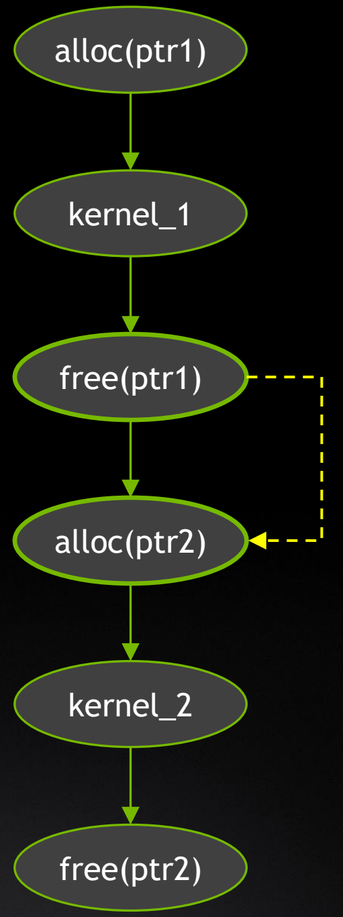
PROGRAMMING MODEL

ptr1 is now logically invalid but physical memory is still there

ptr2 can then re-use ptr1's memory without re-allocating

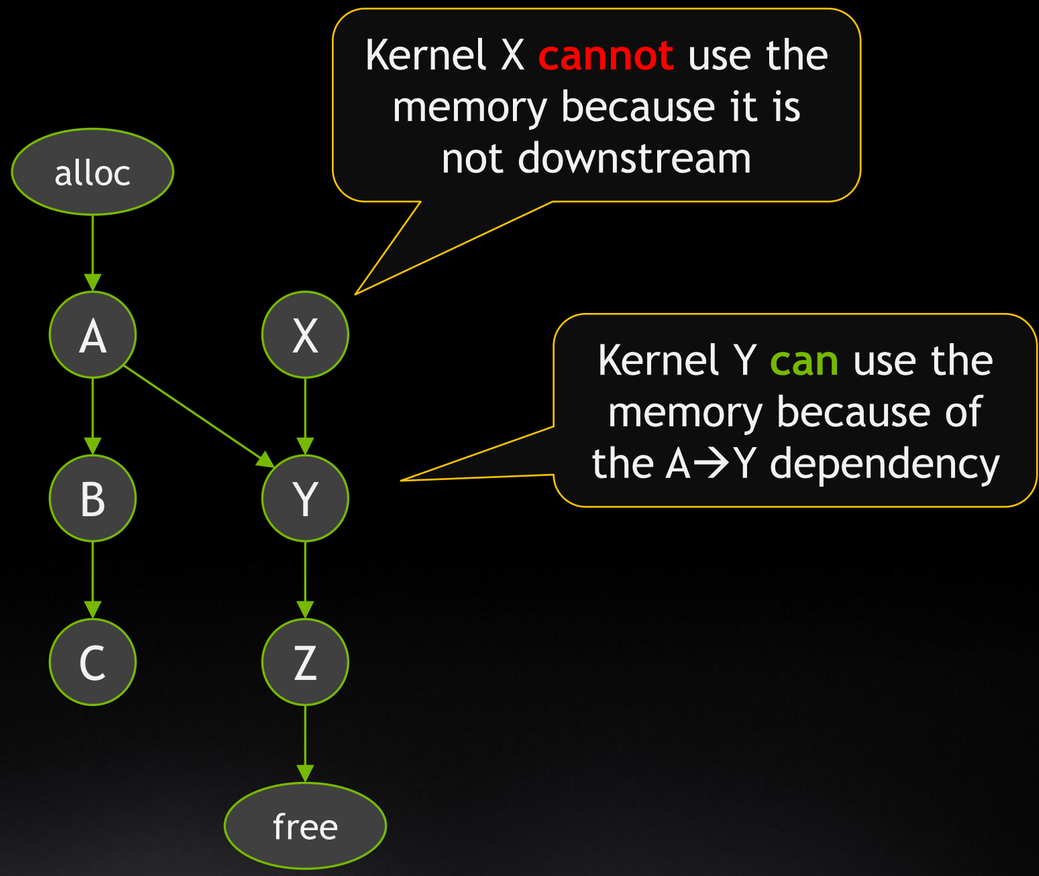
```
cudaMalloc...c(&ptr1, size...
kernel_1<<<...,stream>>>(ptr...
cudaFreeAsync(ptr1, stream);

cudaMallocAsync(&ptr2, size, stream);
kernel_2<<<...,stream>>>(ptr2);
cudaFreeAsync(ptr2, stream);

cudaStreamSynchronize(stream);
```

Physical memory is released on synchronization

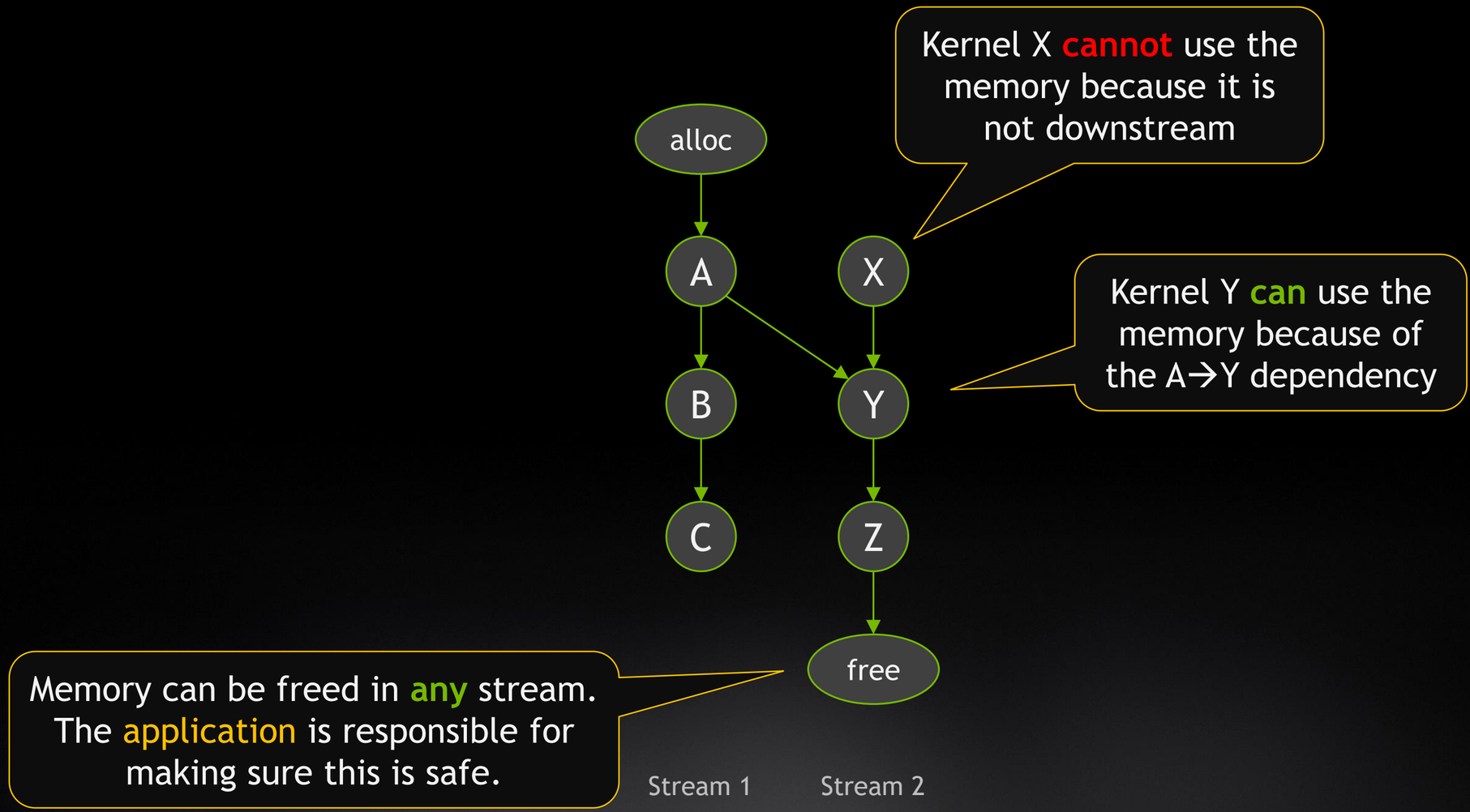


CUDA's stream ordering guarantees when it's safe to re-use memory

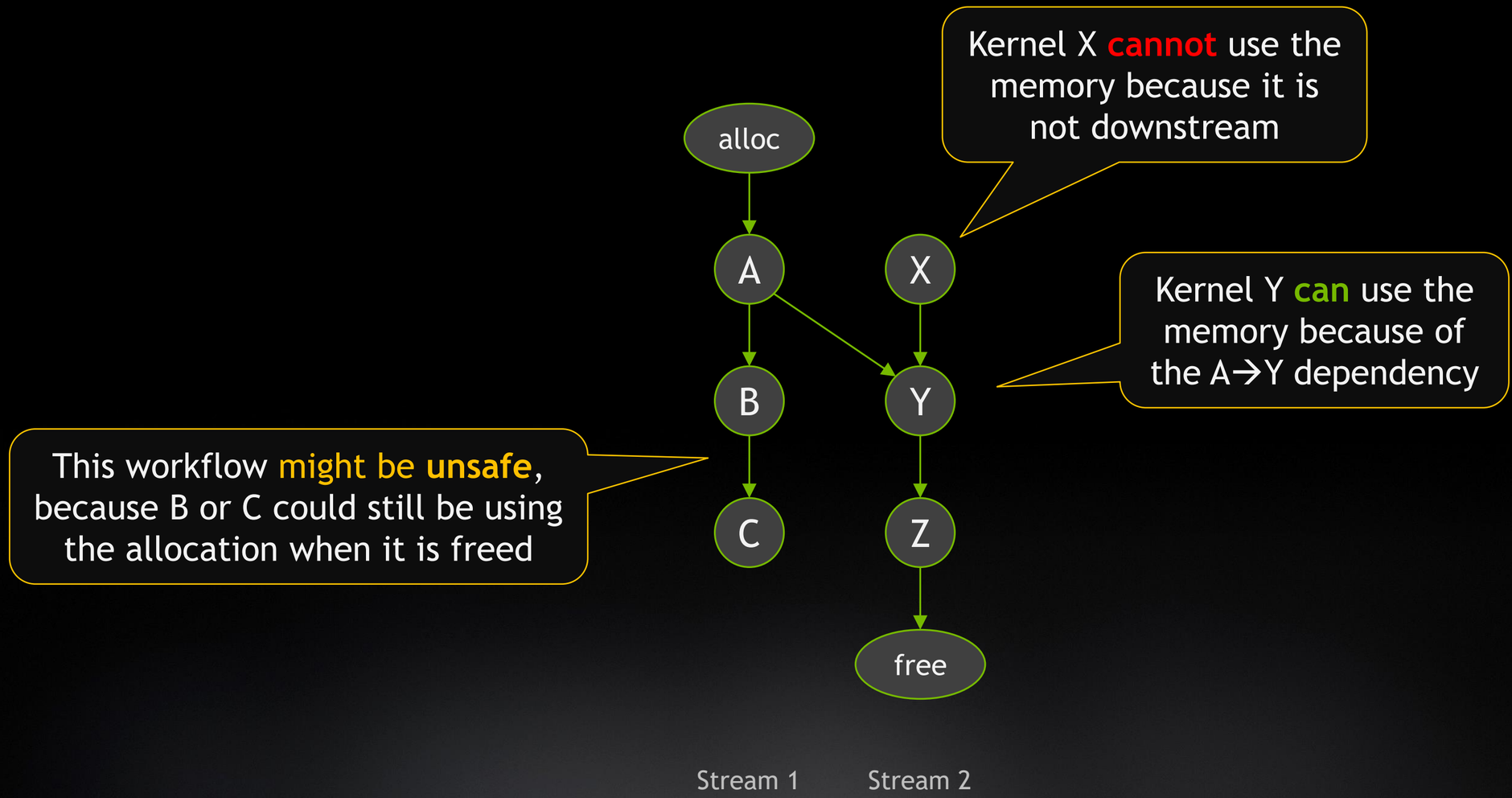
ALLOCATION VISIBILITY FOLLOWS STREAM DEPENDENCIES



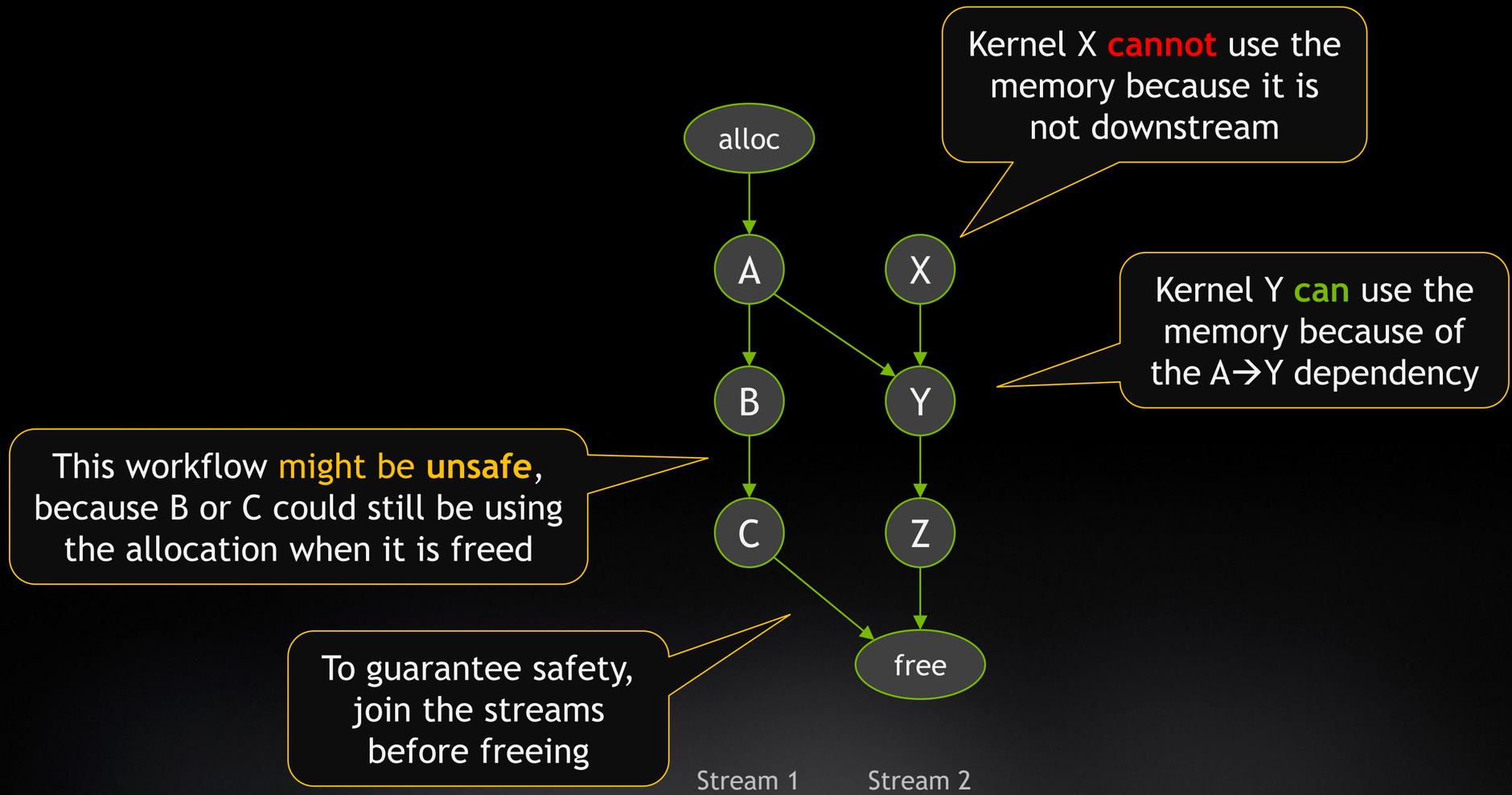
ALLOCATION VISIBILITY FOLLOWS STREAM DEPENDENCIES



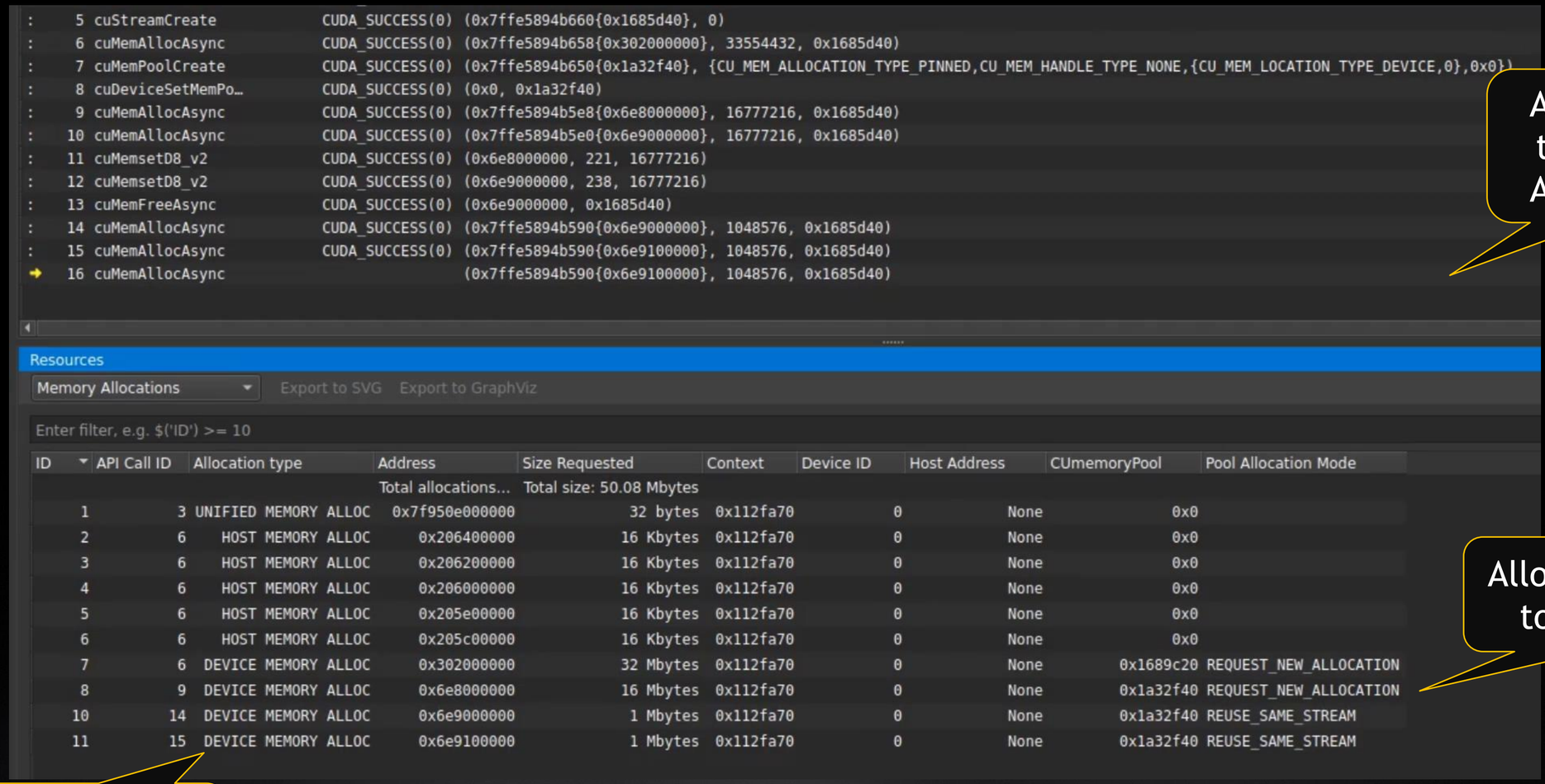
ALLOCATION VISIBILITY FOLLOWS STREAM DEPENDENCIES



ALLOCATION VISIBILITY FOLLOWS STREAM DEPENDENCIES



ASYNCHRONOUS ALLOCATION SUPPORT IN NSIGHT COMPUTE



```
:     5 cuStreamCreate            CUDA_SUCCESS(0) (0x7ffe5894b660{0x1685d40}, 0)
:     6 cuMemAllocAsync           CUDA_SUCCESS(0) (0x7ffe5894b658{0x302000000}, 33554432, 0x1685d40)
:     7 cuMemPoolCreate           CUDA_SUCCESS(0) (0x7ffe5894b650{0x1a32f40}, {CU_MEM_ALLOCATION_TYPE_PINNED,CU_MEM_HANDLE_TYPE_NONE,{CU_MEM_LOCATION_TYPE_DEVICE,0},0x0})
:     8 cuDeviceSetMemPo…         CUDA_SUCCESS(0) (0x0, 0x1a32f40)
:     9 cuMemAllocAsync           CUDA_SUCCESS(0) (0x7ffe5894b5e8{0x6e8000000}, 16777216, 0x1685d40)
:    10 cuMemAllocAsync           CUDA_SUCCESS(0) (0x7ffe5894b5e0{0x6e9000000}, 16777216, 0x1685d40)
:    11 cuMemsetD8_v2             CUDA_SUCCESS(0) (0x6e8000000, 221, 16777216)
:    12 cuMemsetD8_v2             CUDA_SUCCESS(0) (0x6e9000000, 238, 16777216)
:    13 cuMemFreeAsync            CUDA_SUCCESS(0) (0x6e9000000, 0x1685d40)
:    14 cuMemAllocAsync           CUDA_SUCCESS(0) (0x7ffe5894b590{0x6e9000000}, 1048576, 0x1685d40)
:    15 cuMemAllocAsync           CUDA_SUCCESS(0) (0x7ffe5894b590{0x6e9100000}, 1048576, 0x1685d40)
➔    16 cuMemAllocAsync                           (0x7ffe5894b590{0x6e9100000}, 1048576, 0x1685d40)
```

Resources

Memory Allocations | Export to SVG | Export to GraphViz

Enter filter, e.g. $('ID') >= 10

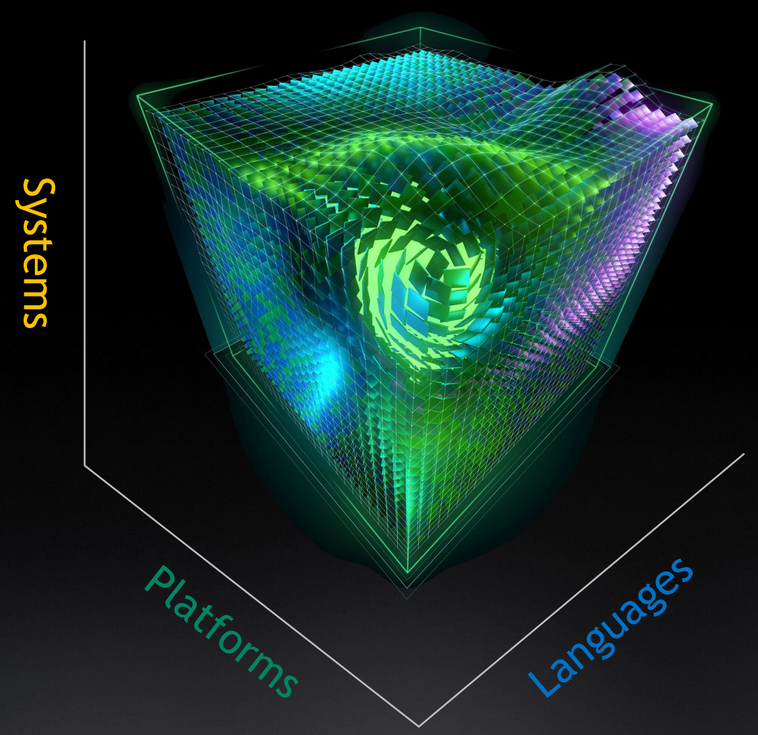
ID	API Call ID	Allocation type	Address	Size Requested	Context	Device ID	Host Address	CUmemoryPool	Pool Allocation Mode
			Total allocations...	Total size: 50.08 Mbytes					
1	3	UNIFIED MEMORY ALLOC	0x7f950e000000	32 bytes	0x112fa70	0	None	0x0	
2	6	HOST MEMORY ALLOC	0x206400000	16 Kbytes	0x112fa70	0	None	0x0	
3	6	HOST MEMORY ALLOC	0x206200000	16 Kbytes	0x112fa70	0	None	0x0	
4	6	HOST MEMORY ALLOC	0x206000000	16 Kbytes	0x112fa70	0	None	0x0	
5	6	HOST MEMORY ALLOC	0x205e00000	16 Kbytes	0x112fa70	0	None	0x0	
6	6	HOST MEMORY ALLOC	0x205c00000	16 Kbytes	0x112fa70	0	None	0x0	
7	6	DEVICE MEMORY ALLOC	0x302000000	32 Mbytes	0x112fa70	0	None	0x1689c20	REQUEST_NEW_ALLOCATION
8	9	DEVICE MEMORY ALLOC	0x6e8000000	16 Mbytes	0x112fa70	0	None	0x1a32f40	REQUEST_NEW_ALLOCATION
10	14	DEVICE MEMORY ALLOC	0x6e9000000	1 Mbytes	0x112fa70	0	None	0x1a32f40	REUSE_SAME_STREAM
11	15	DEVICE MEMORY ALLOC	0x6e9100000	1 Mbytes	0x112fa70	0	None	0x1a32f40	REUSE_SAME_STREAM

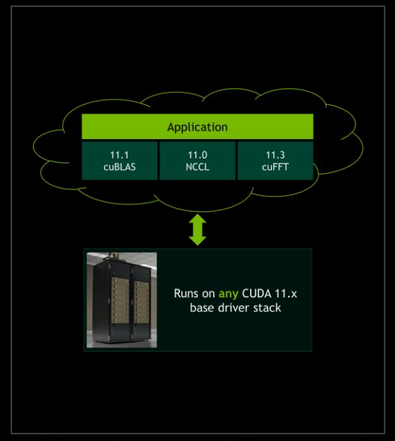
Allocations tracked in API Stream

Allocation mode to see reuse

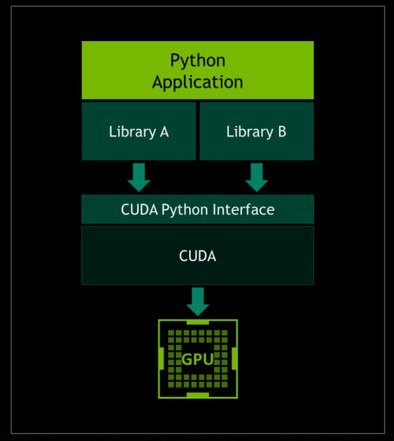
Allocation type and size

THE AXES OF THE CUDA PLATFORM

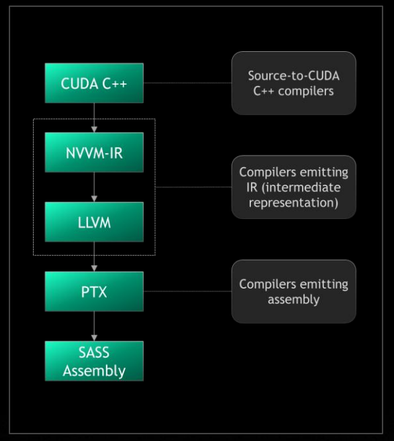




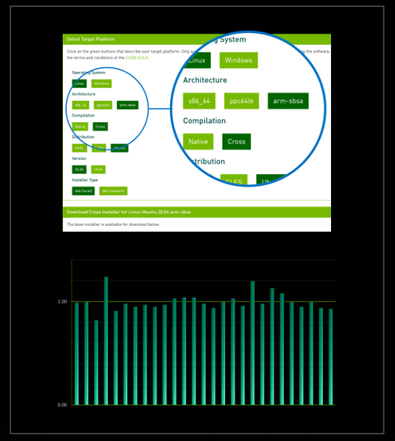
Compatibility



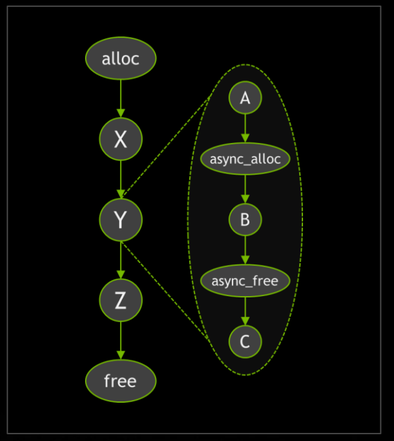
CUDA Python
Bindings



GPU Computing
In Any Language



CUDA on Arm



Asynchronous
Memory Allocation

Download CUDA 11.3 Today: https://developer.nvidia.com/cuda-downloads

References

S31358 Inside NVC++ and NVFORTRAN [S31358]

S31359 The NVIDIA C++ Standard Library [S31359]

S31754 Recent Developments in NVIDIA Math Libraries [S31754]

S31286 A Deep Dive into the Latest HPC Software [S31286]

S31883 Accelerating Convolution with Tensor Cores in CUTLASS [S31883]

S32758 HPC Applications on ARM+NVIDIA A100 [S32758]

S31884 Latest Enhancements to CUDA Debugger IDEs [S31884]

S31747 CUDA is Evolving, and the Latest Developer Tools are Adapting to Keep Up [S31747]

S32089 Requests, Wavefronts, Sectors Metrics: Understanding and Optimizing Memory-Bound Kernels with Nsight Compute [S32089]

E31888 Optimizing Applications with Asynchronous GPU programming in CUDA C++ [E31888]

GTC-2020 A Faster Radix Sort Implementation [GTC 2020]

CWES2484 CUDA Programming Model and Compatibility Guarantees [CWES2484]

CWES1802 Future of Standard and CUDA C++ [CWES1802]

CWES1801 Thrust, CUB, and libcu++ User's Forum [CWES1801]

CWES1175 CUDA Memory Management [CWES1175]

CWES1572 Streamline Your Development Workflow with CUDA Profiling, Optimization, and Debugging Tools [CWES1572]

Article RAPIDS on Windows Subsystem for Linux [Article]

Blog Accelerating Matrix Multiplication with Block Sparse Format and NVIDIA Tensor Cores [Blog]