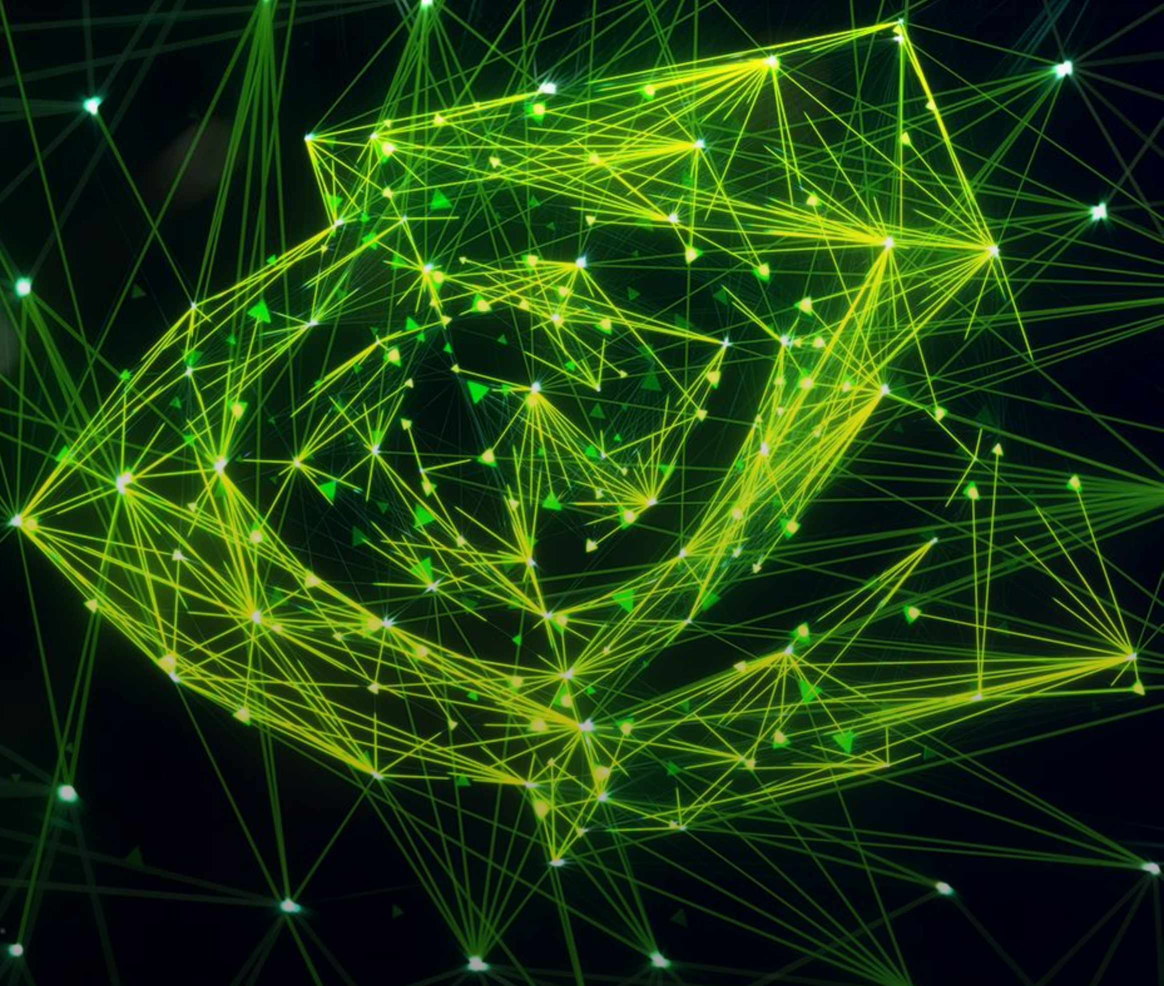


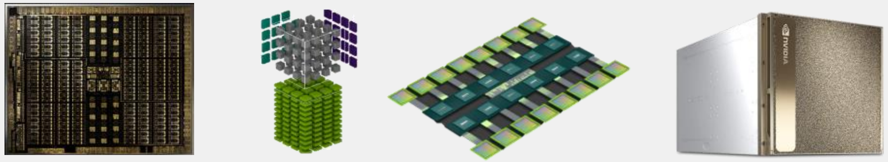
HIGHLIGHTS OF CUDA 10 FOR SUMMIT

March 27, 2019 | OLCF User Conference Call
Steve Abbott

INTRODUCING CUDA 10.0

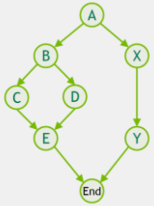
TURING AND NEW SYSTEMS

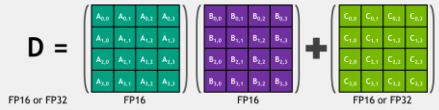
New GPU Architecture, Tensor Cores, NVSwitch Fabric



CUDA PLATFORM

CUDA Graphs, Vulkan & DX12 Interop, Warp Matrix



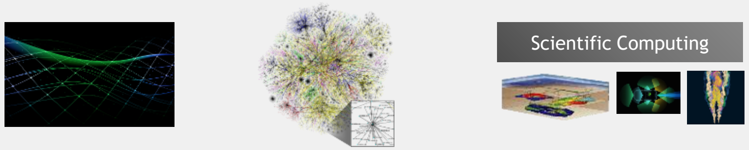


$$\underset{\text{FP16 or FP32}}{D} = \underset{\text{FP16}}{\begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix}} \underset{\text{FP16}}{\begin{pmatrix} B_{0,0} &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

$$D = AB + C$$

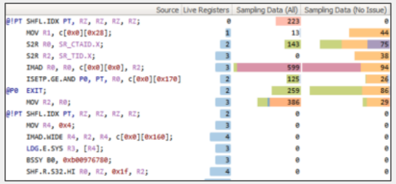
LIBRARIES

GPU-accelerated hybrid JPEG decoding,
Symmetric Eigenvalue Solvers, FFT Scaling



DEVELOPER TOOLS

New Nsight Products – Nsight Systems and Nsight Compute



Source	Live Registers	Sampling Data (All)	Sampling Data (No Issue)
`@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;`	0	223	0
`MOV R1, c[0x0][0x28];`	1	13	44
`S2R R0, SR_CTAID.X;`	2	143	75
`S2R R2, SR_TID.X;`	3	0	38
`IMAD R0, R0, c[0x0][0x0], R2;`	3	599	94
`ISETP.GE.AND P0, PT, R0, c[0x0][0x170]`	2	125	26
`@P0 EXIT;`	2	259	86
`MOV R2, R0;`	3	386	29
`@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;`	2	0	0
`MOV R4, 0x4;`	3	0	0
`IMAD.WIDE R4, R2, R4, c[0x0][0x160];`	4	0	0
`LDG.E.SYS R3, [R4];`	3	0	0
`BSSY B0, 0xb00976780;`	3	0	0
`SHF.R.S32.HI R0, RZ, 0x1f, R2;`	4	0	0

CUDA DEVELOPMENT ECOSYSTEM

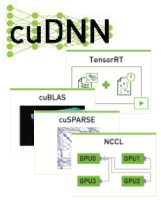
GPU Users

Applications

Domain Specialists

Frameworks

Problem Specialists



Libraries

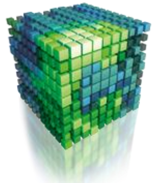
New Algorithm Developers and Optimization Experts

OpenACC
Directives for Accelerators

Thrust

Directives and Standard Languages

CUDA-C++
CUDA Fortran



Extended Standard Languages

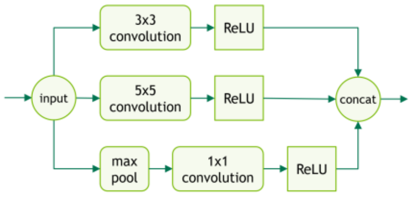
Ease of use → *Specialized Performance*

CUDA: Programming Model, GPU Architecture, System Architecture

PROGRAMMING MODEL

NEW PROGRAMMING MODEL FEATURES

Execution

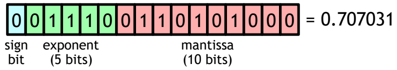


Asynchronous Task Graphs

Precision

```
atomicAdd(&h, (half)1.15f);

half2 hvec(0.94f, -2.13f);
atomicAdd(&h2, hvec);
```

IEEE-754.2008 FP16 Specification



FP16 Operations

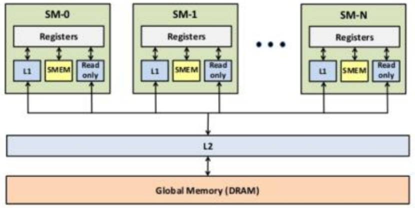
Interop

Windows 10

DirectX 12

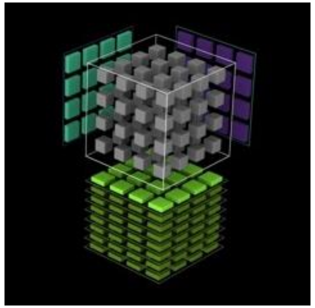
Lightweight Graphics Interop

Efficiency



NVCC Enhancements

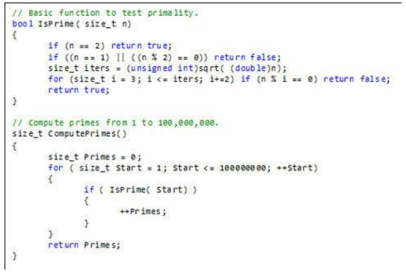
Turing



Multi-Precision Tensor Cores

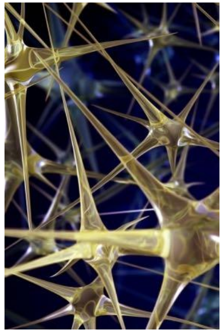
ASYNCHRONOUS TASK GRAPHS

Execution Optimization When Workflow is Known Up-Front

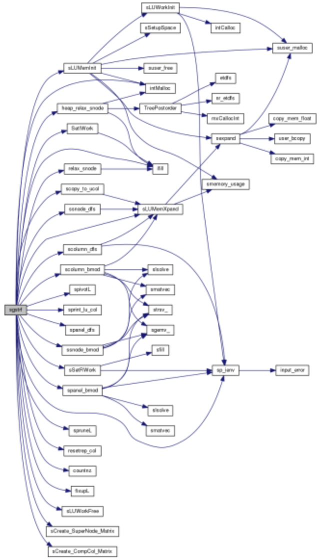


```
// Basic function to test primality.
bool IsPrime( size_t n)
{
        if (n == 2) return true;
        if ((n == 1) || ((n % 2) == 0)) return false;
        size_t iters = (unsigned int)sqrt( (double)n);
        for (size_t i = 3; i <= iters; i+=2) if (n % i == 0) return false;
        return true;
}

// Compute primes from 1 to 100,000,000.
size_t ComputePrimes()
{
        size_t Primes = 0;
        for ( size_t Start = 1; Start <= 100000000; ++Start)
        {
                if ( IsPrime( Start) )
                {
                        ++Primes;
                }
        }
        return Primes;
}
```

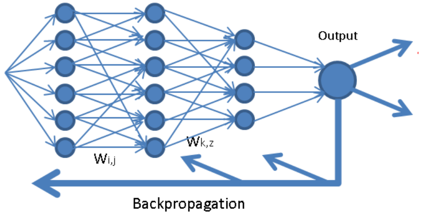
Loop & Function offload



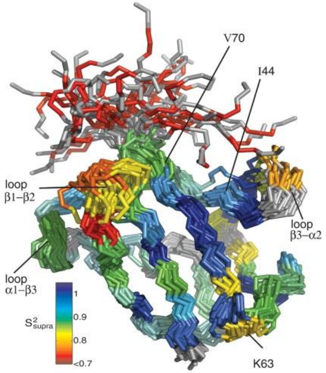
DL Inference



Linear Algebra



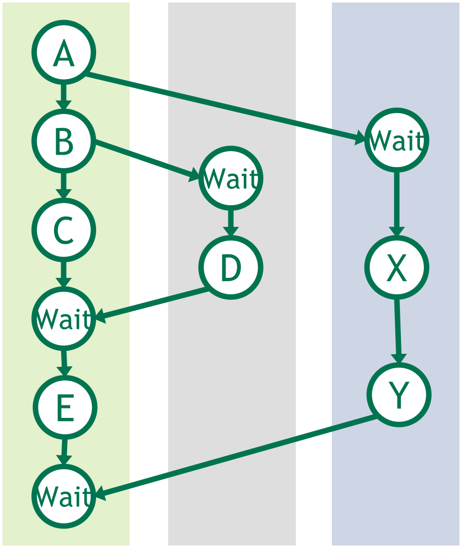
Deep Neural Network Training



HPC Simulation

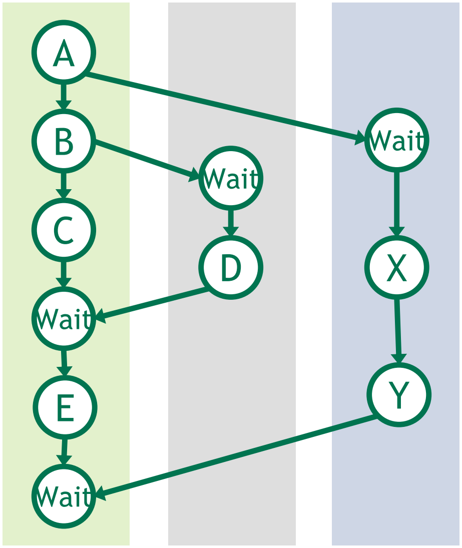
ALL CUDA WORK FORMS A GRAPH

CUDA Work in Streams



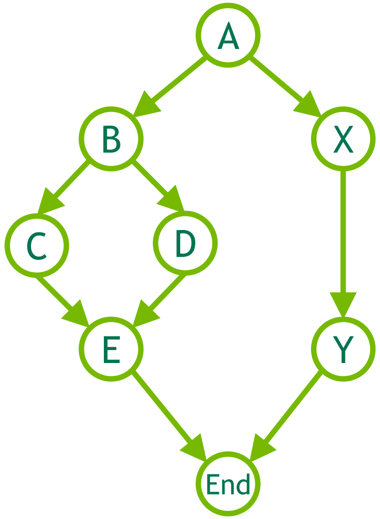
ALL CUDA WORK FORMS A GRAPH

CUDA Work in Streams



Any CUDA stream can be mapped to a graph

Graph of Dependencies



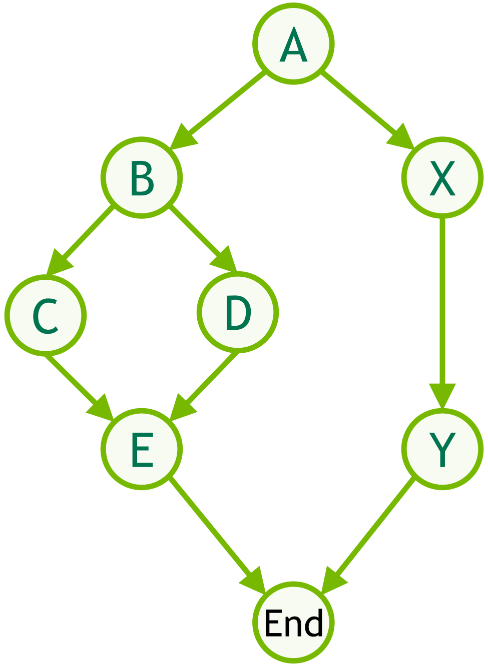
DEFINITION OF A CUDA GRAPH

Graph Nodes Are Not Just Kernel Launches

Sequence of operations, connected by dependencies.

Operations are one of:

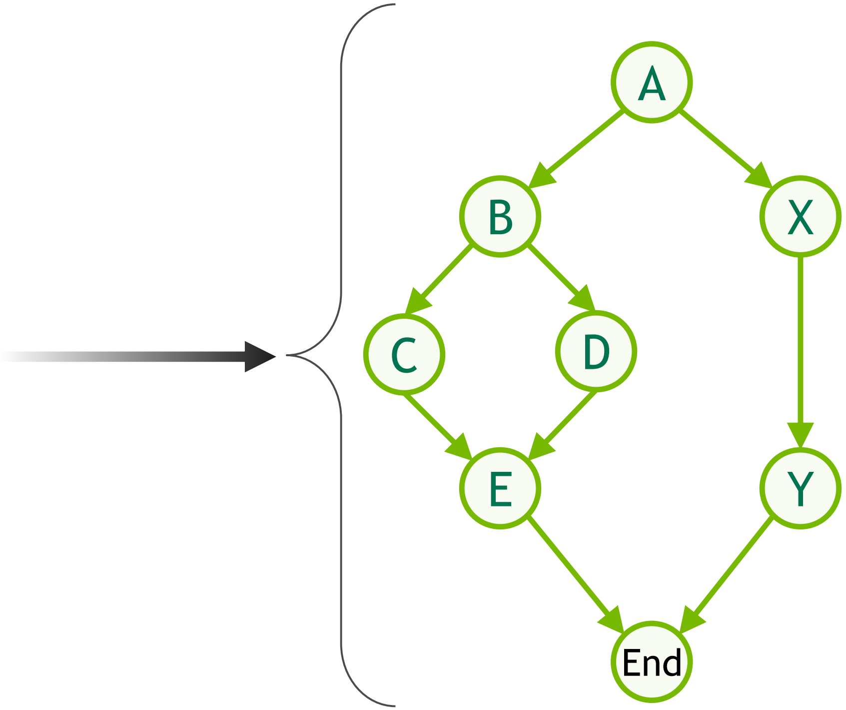
Kernel Launch	CUDA kernel running on GPU
CPU Function Call	Callback function on CPU
Memcopy/Memset	GPU data management
Sub-Graph	Graphs are hierarchical



NEW EXECUTION MECHANISM

Graphs Can Be Generated Once Then Launched Repeatedly

```
for(int i=0; i<1000; i++) {
    launch_graph( G );
}
```

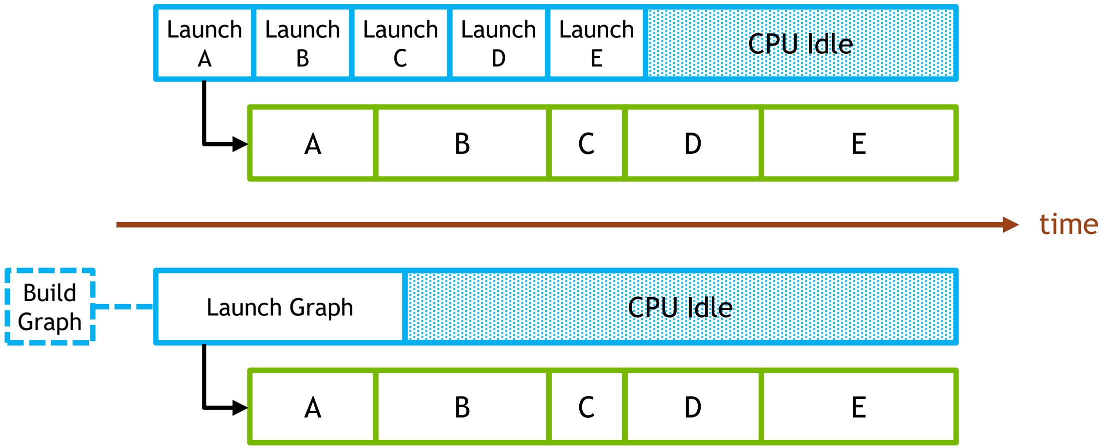


EXECUTION OPTIMIZATIONS

Latency & Overhead Reductions

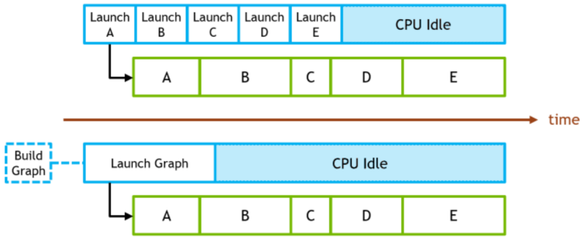
Launch latencies:

- CUDA 10.0 takes at least 2.2us CPU time to launch **each** CUDA kernel on Linux
- Pre-defined graph allows launch of **any number** of kernels in **one single operation**



PERFORMANCE IMPACT

Optimizations for Short-Runtime Operations



Launch A | Launch B | Launch C | Launch D | Launch E | CPU Idle

A | B | C | D | E

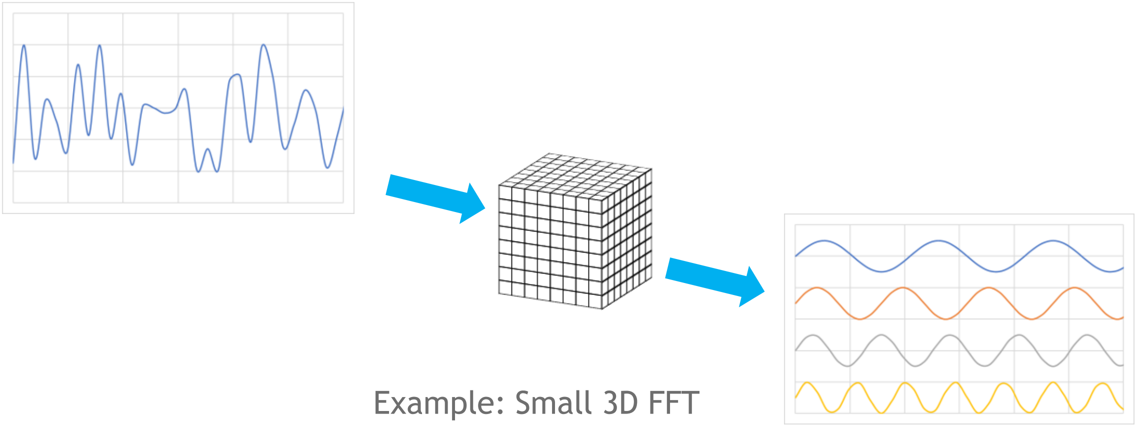
time

Build Graph | Launch Graph | CPU Idle

A | B | C | D | E

CPU launch time improvements

Typical: 33% faster than stream launch



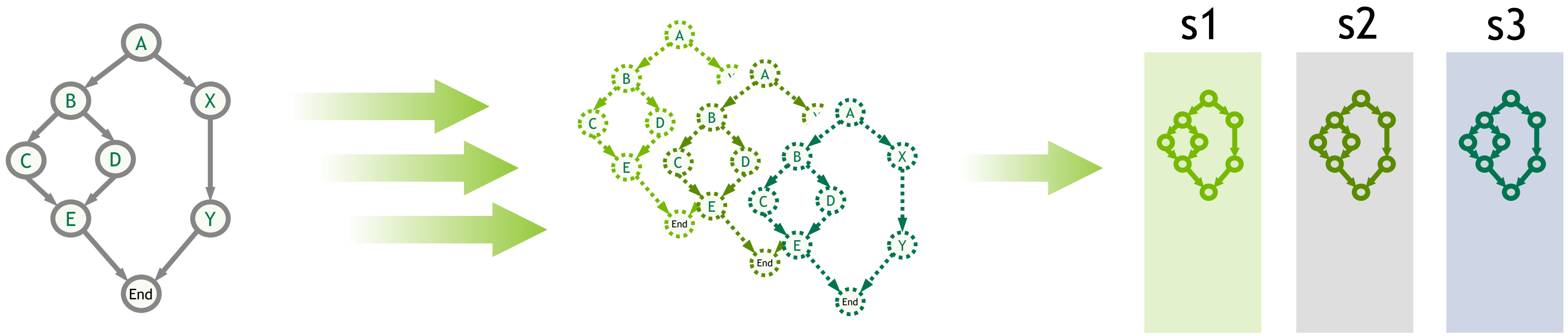
Example: Small 3D FFT

25% end-to-end improvement for 32^3 3D-FFT
(16us with stream launch, 12us with graph launch)

NOTE: Performance impact is workload-dependent

Benefits especially short-running kernels, where overheads account for more runtime

THREE-STAGE EXECUTION MODEL



Define

Single Graph "Template"

Created in host code
or built up from libraries

Instantiate

Multiple "Executable Graphs"

Snapshot of template
Sets up & initializes GPU
execution structures
(create once, run many times)

Execute

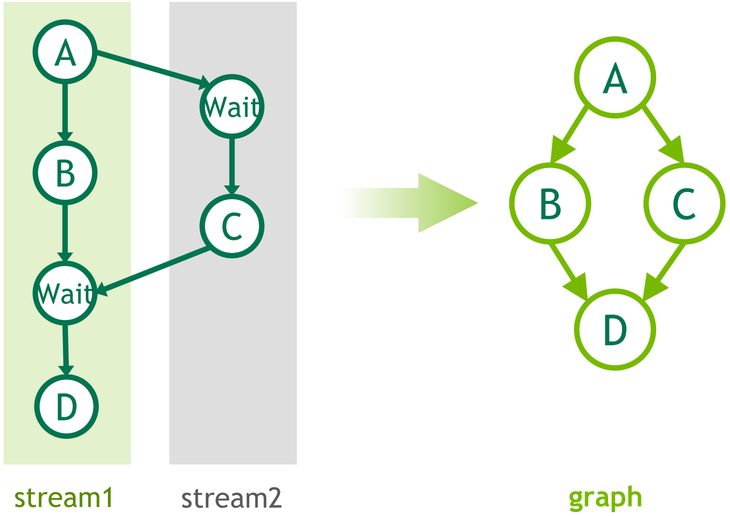
**Executable Graphs
Running in CUDA Streams**

Concurrency in graph
is not limited by stream

CONVERT CUDA STREAM INTO A GRAPH

Construct a graph from normal CUDA stream syntax

```
// Start by initating stream capture
cudaStreamBeginCapture(&stream1, cudaStreamCaptureModeGlobal);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

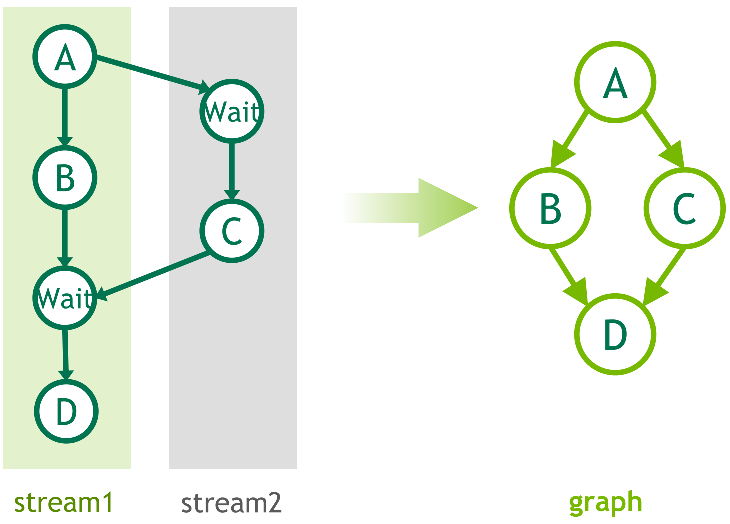


CONVERT CUDA STREAM INTO A GRAPH

Construct a graph from normal CUDA stream syntax

```
// Start by initating stream capture
cudaStreamBeginCapture(&stream1, cudaStreamCaptureModeGlobal);

// Build stream work as usual
A<<< ..., stream1 >>>();
cudaEventRecord(e1, stream1);
B<<< ..., stream1 >>>();
cudaStreamWaitEvent(stream2, e1);
C<<< ..., stream2 >>>();
cudaEventRecord(e2, stream2);
cudaStreamWaitEvent(stream1, e2);
D<<< ..., stream1 >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream1, &graph);
```

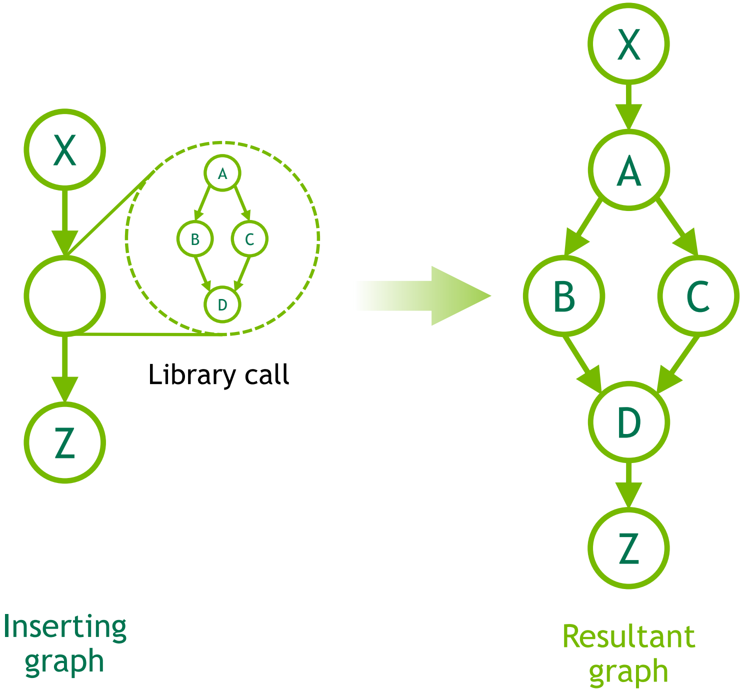
Capture follows
inter-stream dependencies
to create forks & joins



CAPTURE EXTERNAL WORK

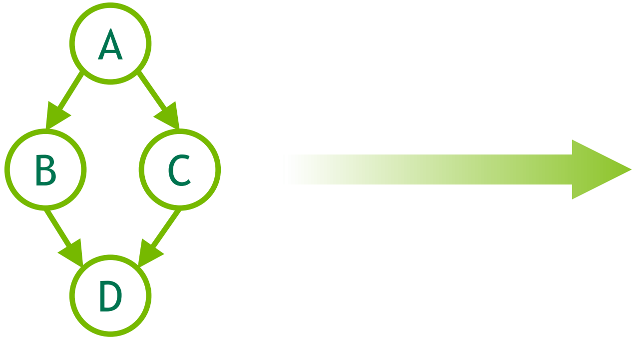
Stream Capture Continues Into Library Calls

```
// Start by initating stream capture
cudaStreamBeginCapture(&stream, cudaStreamCaptureModeGlobal);

// Captures my kernel launches, recurse into library calls
X<<< ..., stream >>>();
libraryCall(stream);         // Launches A, B, C, D
Z<<< ..., stream >>>();

// Now convert the stream to a graph
cudaStreamEndCapture(stream, &graph);
```



CREATE GRAPHS DIRECTLY

Map Graph-Based Workflows Directly Into CUDA



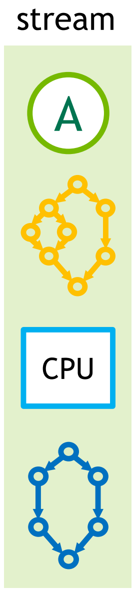
Graph from framework

```
// Define graph of work + dependencies
cudaGraphCreate(&graph);

cudaGraphAddNode(graph, kernel_a, {}, ...);
cudaGraphAddNode(graph, kernel_b, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_c, { kernel_a }, ...);
cudaGraphAddNode(graph, kernel_d, { kernel_b, kernel_c }, ...);

// Instantiate graph and apply optimizations
cudaGraphInstantiate(&instance, graph);

// Launch executable graph 100 times
for(int i=0; i<100; i++)
     cudaGraphLaunch(instance, stream);
```

GRAPH EXECUTION SEMANTICS

Order Graph Work With Other Non-Graph CUDA Work

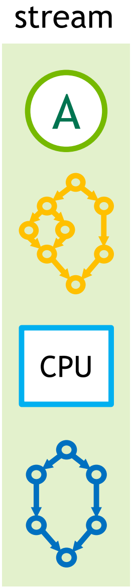
```
launchWork(cudaGraphExec_t i1, cudaGraphExec_t i2,
           CPU_Func cpu, cudaStream_t stream) {

    A <<< 256, 256, 0, stream >>>();          // Kernel launch
    cudaGraphLaunch(i1, stream);              // Graph1 launch
    cudaStreamAddCallback(stream, cpu);       // CPU callback
    cudaGraphLaunch(i2, stream);              // Graph2 launch

    cudaStreamSynchronize(stream);
}
```



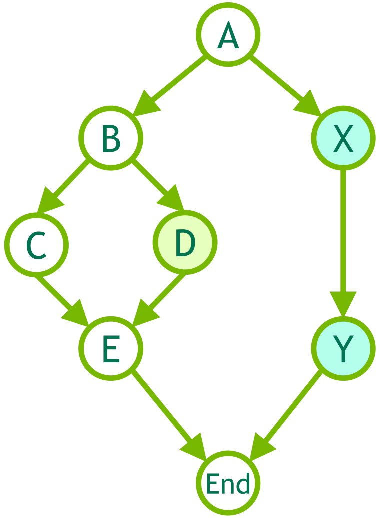
If you can put it in a CUDA stream, you can run it together with a graph

GRAPHS IGNORE STREAM SERIALIZATION RULES

Launch Stream Is Used Only For Ordering With Other Work



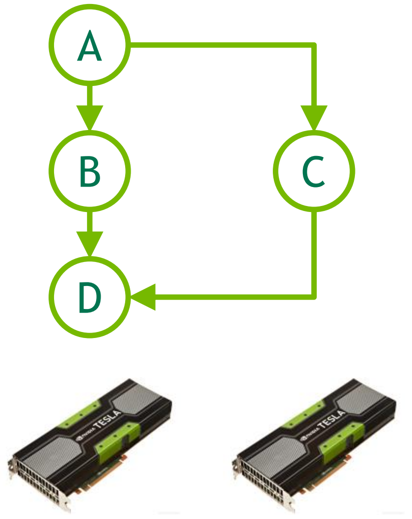
Branches in graph still execute concurrently even though graph is launched into a stream



CROSS-DEVICE DEPENDENCIES

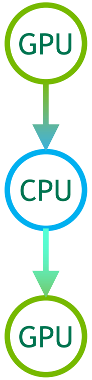
Graphs May Span Multiple GPUs

Multi-Device Execution



GPU 0 GPU 1

Heterogeneous Execution



CUDA is closest to the O/S and the hardware

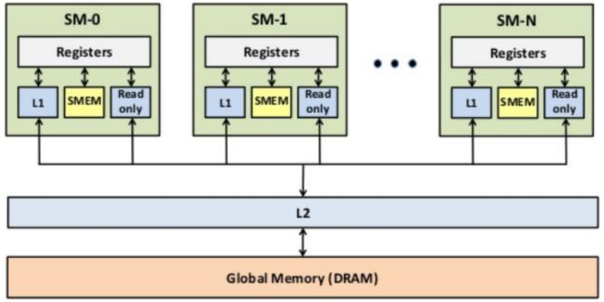
- Can optimize **multi-device** dependencies
- Can optimize **heterogeneous** dependencies
- Define locality per-node

NVCC IN CUDA 10

Improving Efficiency

Extensible Whole Program (-ewp) mode compilation support

Enables efficient compilation with use of CUDA run-time device library



Efficient Code Generation for Chip Architecture

Compiler optimization and code generation heuristics tuning for Volta and Turing

ENHANCED HALF-PRECISION FUNCTIONALITY

Includes Limited *half* Type Support For CPU Code

Half-precision atomic ADD (Volta+) (round-to-nearest mode)

```
half atomicAdd(half *address, half val);
half2 atomicAdd(half2 *address, half2 val);
```

Host-side conversion operators between *float* and *half* types

```
half pi = 3.1415f;                // Convert float to half
float fPI = (float)hPI;           // Convert half to float
```

Host-side construction and assignment operators for *half* and *half2* types

```
half pi = 3.1415f;
half also_pi = pi;                // Assign half to half
half2 vector_pi(pi, also_pi);     // Construct half2 from half
```

NOTE: Half-precision *arithmetic* operations remain only available in device code

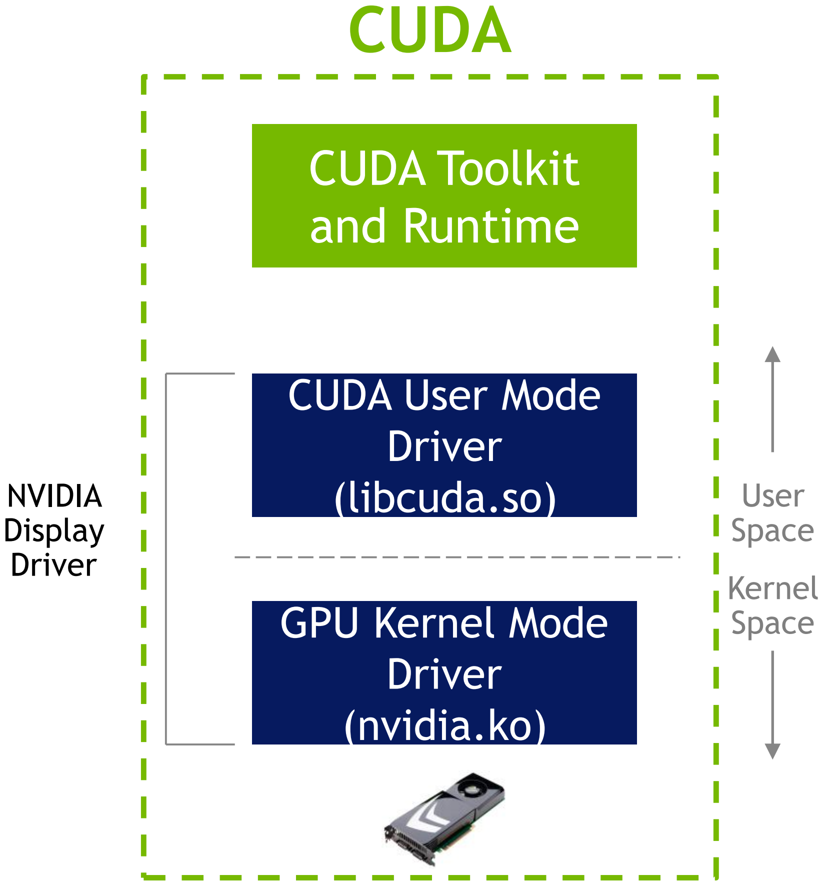
CUDA DEPLOYMENT

CUDA INSTALLED COMPONENTS

CUDA is comprised of three components

1. CUDA Toolkit (build applications)
2. CUDA User Mode Driver (run applications)
3. NVIDIA Kernel Mode Driver (run applications)

Note that components 2 and 3 are delivered together in the NVIDIA Display Driver package



CUDA

CUDA Toolkit and Runtime

CUDA User Mode Driver (libcuda.so)

GPU Kernel Mode Driver (nvidia.ko)

NVIDIA Display Driver

User Space

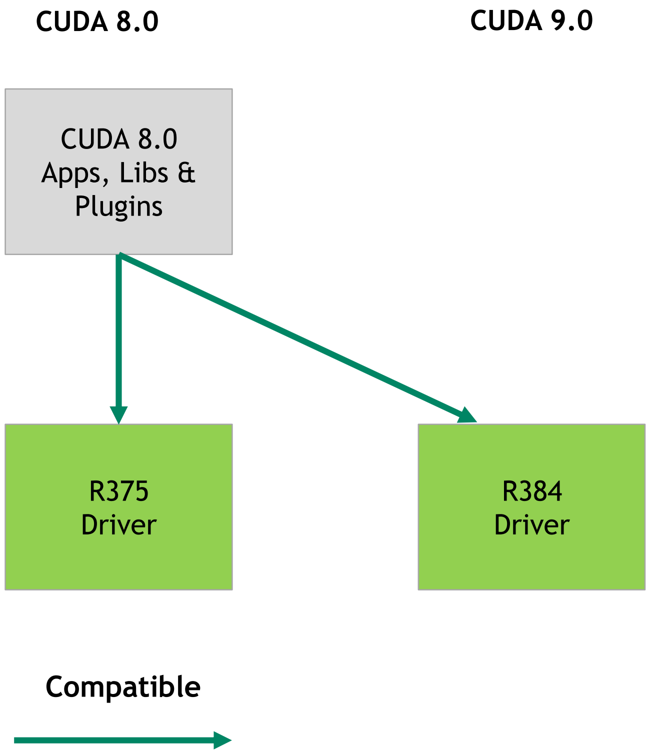
Kernel Space

CUDA COMPATIBILITY - TODAY

Older CUDA Version **Runs** on *Newer* Display Driver

CUDA driver API is backward compatible but not forward compatible

- Each CUDA release has a minimum driver requirement
- Applications compiled against a particular version of CUDA API will work on later driver releases
- E.g.
  - CUDA 8.0 needs >= R375
  - CUDA 9.0 needs >= R384



CUDA 8.0

CUDA 9.0

CUDA 8.0
Apps, Libs &
Plugins

R375
Driver

R384
Driver

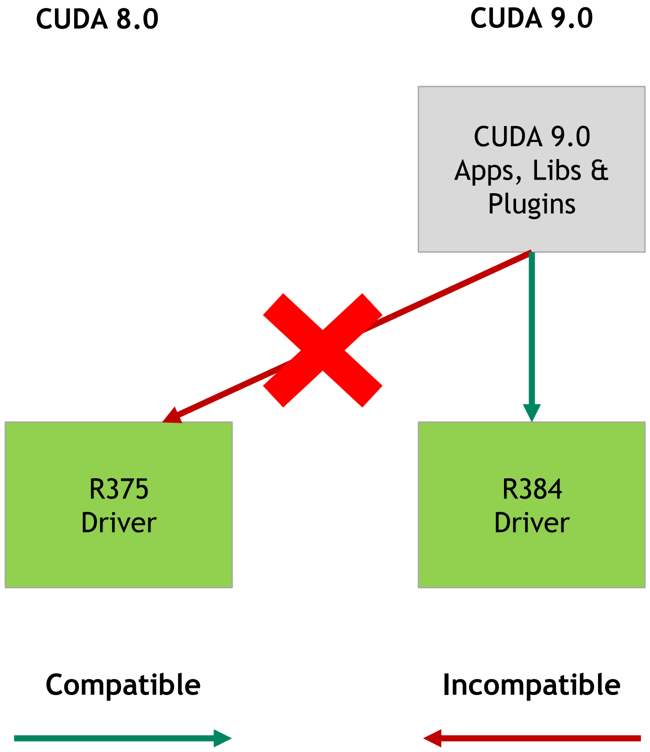
Compatible

CUDA COMPATIBILITY - TODAY

Newer CUDA Version **DOES NOT** Run on *Older* Display Driver

CUDA driver API is backward compatible but not forward compatible

- Each CUDA release has a minimum driver requirement
- Applications compiled against a particular version of CUDA API will work on later driver releases
- E.g.
  - CUDA 8.0 needs >= R375
  - CUDA 9.0 needs >= R384



CUDA 8.0

CUDA 9.0

CUDA 9.0
Apps, Libs &
Plugins

R375
Driver

R384
Driver

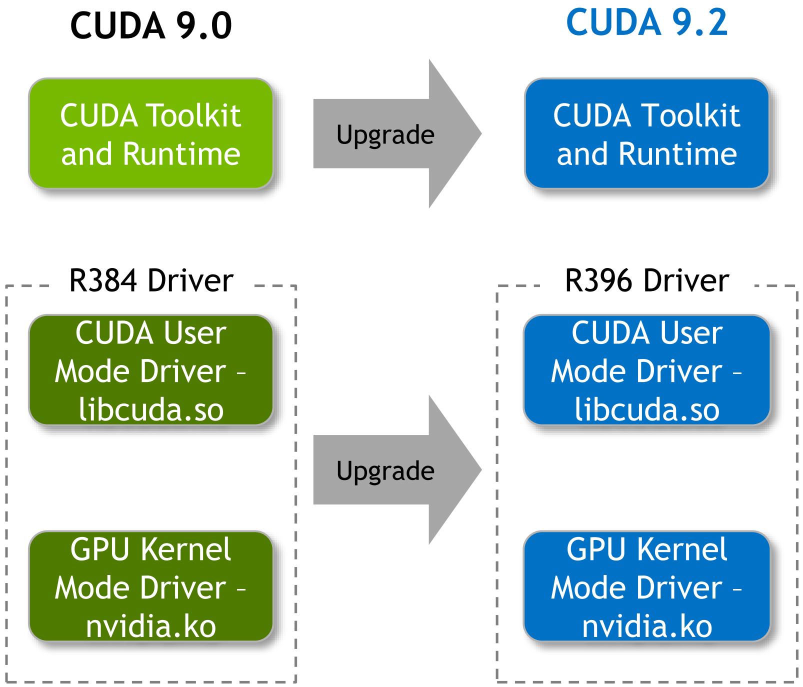
Compatible

Incompatible

CUDA COMPATIBILITY – UPGRADE PATHS

Backward Compatibility

Upgrade *both* toolkit and driver



CUDA 9.0

CUDA Toolkit and Runtime

Upgrade

CUDA 9.2

CUDA Toolkit and Runtime

R384 Driver

CUDA User Mode Driver - libcuda.so

GPU Kernel Mode Driver - nvidia.ko

Upgrade

R396 Driver

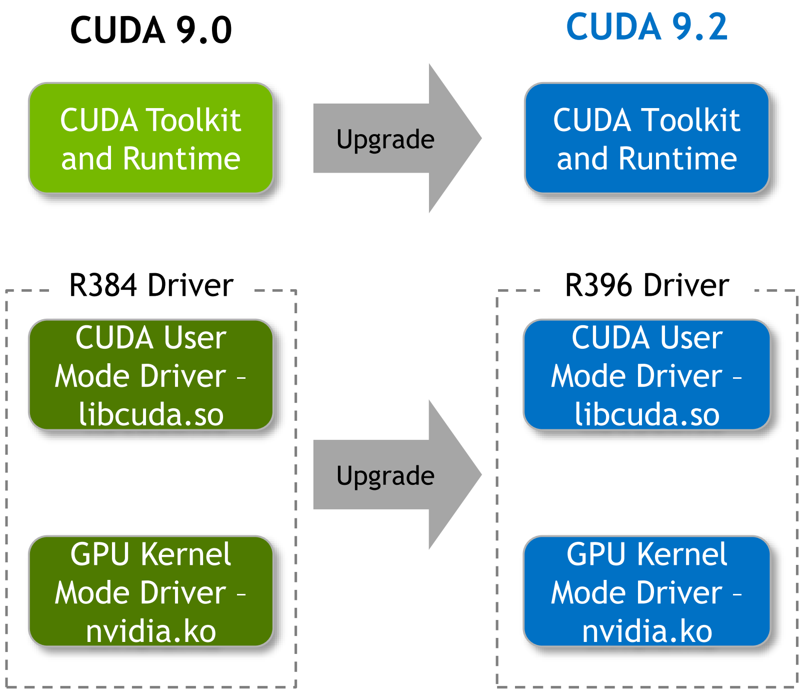
CUDA User Mode Driver - libcuda.so

GPU Kernel Mode Driver - nvidia.ko

CUDA COMPATIBILITY – UPGRADE PATHS

Backward Compatibility

Upgrade *both* toolkit and driver



CUDA 9.0

CUDA 9.2

CUDA Toolkit and Runtime

Upgrade

CUDA Toolkit and Runtime

R384 Driver

CUDA User Mode Driver - libcuda.so

GPU Kernel Mode Driver - nvidia.ko

Upgrade

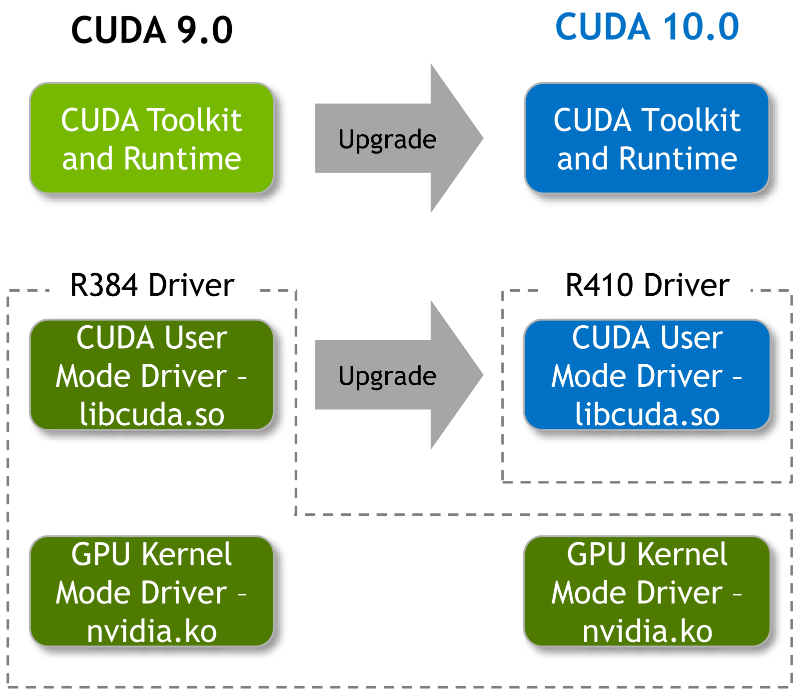
R396 Driver

CUDA User Mode Driver - libcuda.so

GPU Kernel Mode Driver - nvidia.ko

NEW Forward Compatibility *Option*

Upgrade *only* user-mode CUDA components*



CUDA 9.0

CUDA 10.0

CUDA Toolkit and Runtime

Upgrade

CUDA Toolkit and Runtime

R384 Driver

CUDA User Mode Driver - libcuda.so

Upgrade

R410 Driver

CUDA User Mode Driver - libcuda.so

GPU Kernel Mode Driver - nvidia.ko

GPU Kernel Mode Driver - nvidia.ko

*requires new ‘cuda-compat-10-0’ package

CUDA COMPATIBILITY – UPGRADE PATHS

Starting with CUDA 10.0

New compatibility platform upgrade path available

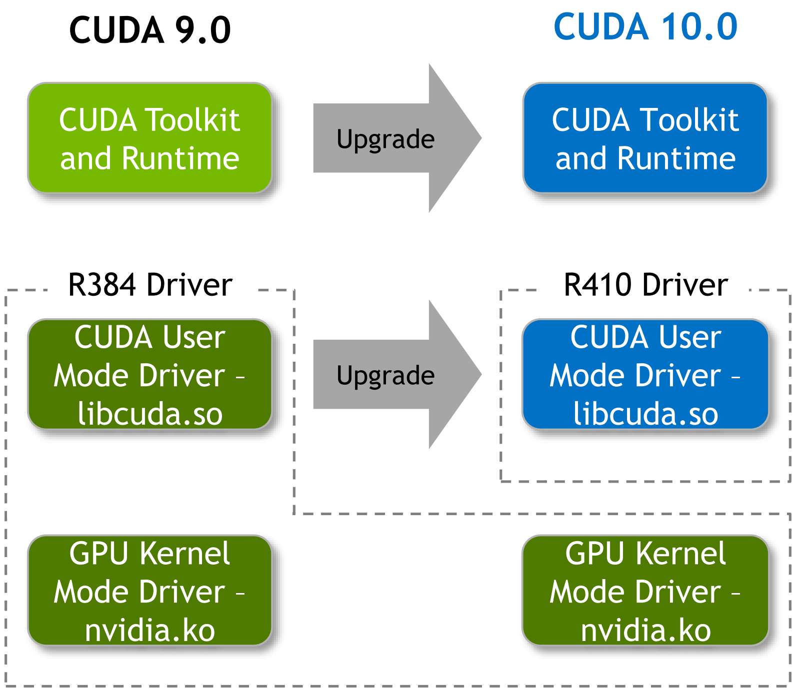
- Use newer CUDA toolkits on older driver installs
- Compatibility only with specific older driver versions

System requirements

- Tesla GPU support only – no Quadro or GeForce
- Only available on Linux

NEW Forward Compatibility *Option*

Upgrade *only* user-mode CUDA components*



CUDA 9.0

CUDA 10.0

CUDA Toolkit and Runtime

Upgrade

CUDA Toolkit and Runtime

R384 Driver

R410 Driver

CUDA User Mode Driver – libcuda.so

Upgrade

CUDA User Mode Driver – libcuda.so

GPU Kernel Mode Driver – nvidia.ko

GPU Kernel Mode Driver – nvidia.ko

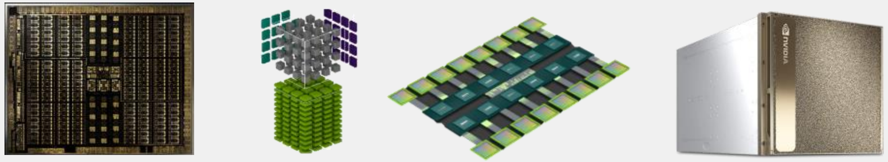
*requires new ‘cuda-compat-10-0’ package

CUDA 10.1 – COMING TO SUMMIT SOON!

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

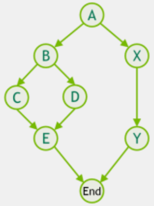
TURING AND NEW SYSTEMS

New GPU Architecture, Tensor Cores, NVSwitch Fabric



CUDA PLATFORM

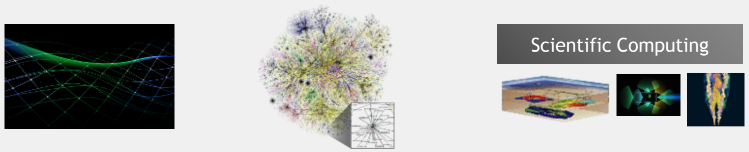
CUDA Graphs, Vulkan & DX12 Interop, Warp Matrix



D = AB + C

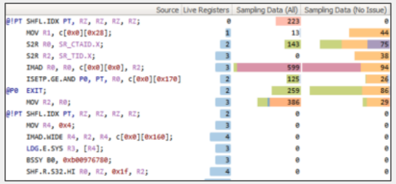
LIBRARIES

GPU-accelerated hybrid JPEG decoding,
Symmetric Eigenvalue Solvers, FFT Scaling



DEVELOPER TOOLS

New Nsight Products – Nsight Systems and Nsight Compute



Source	Live Registers	Sampling Data (All)	Sampling Data (No Issue)
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;	0	223	0
MOV R1, c[0x0][0x28];	1	13	44
S2R R0, SR_CTAID.X;	2	143	75
S2R R2, SR_TID.X;	3	0	38
IMAD R0, R0, c[0x0][0x0], R2;	3	599	94
ISETP.GE.AND P0, PT, R0, c[0x0][0x170]	2	125	26
@P0 EXIT;	2	259	86
MOV R2, R0;	3	386	29
@!PT SHFL.IDX PT, RZ, RZ, RZ, RZ;	2	0	0
MOV R4, 0x4;	3	0	0
IMAD.WIDE R4, R2, R4, c[0x0][0x160];	4	0	0
LDG.E.SYS R3, [R4];	3	0	0
BSSY B0, 0xb00976780;	3	0	0
SHF.R.S32.HI R0, RZ, 0x1f, R2;	4	0	0