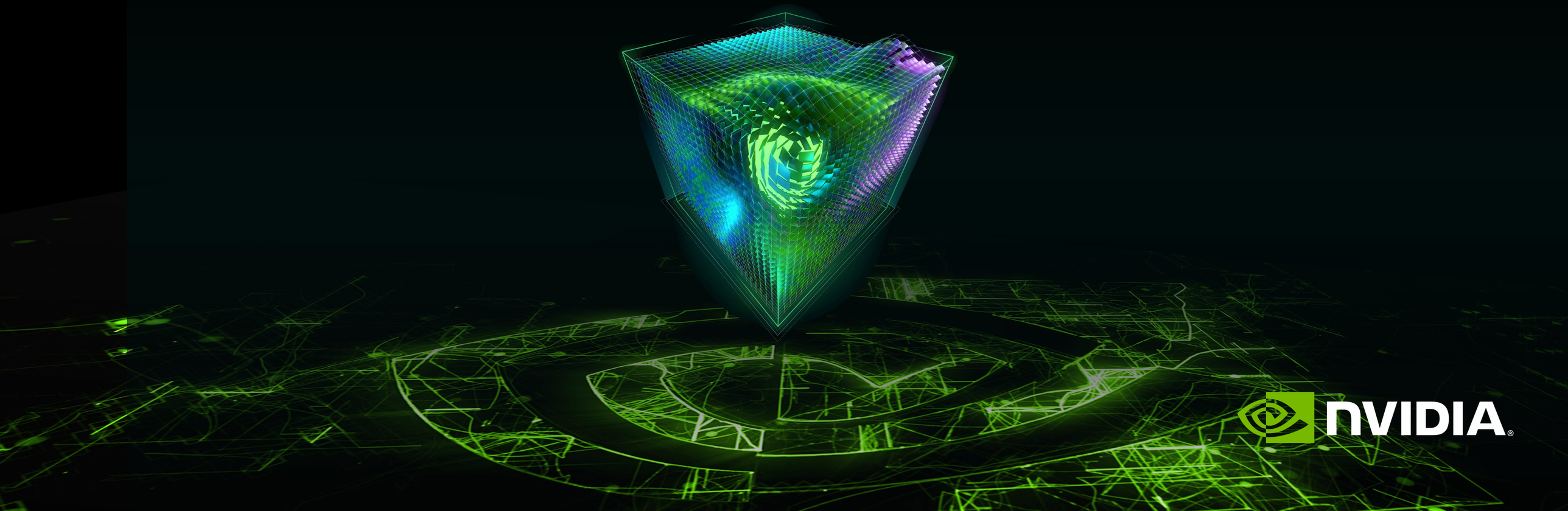
CUDA: NEW FEATURES AND BEYOND

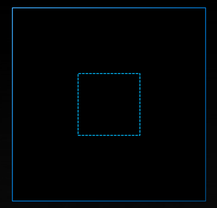
Stephen Jones, GTC 2022



THE FIRST ERA OF SOFTWARE DEVELOPMENT

Single-Thread Era
(until ~2007)

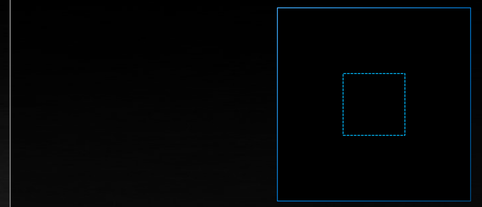
Straight-Line Code

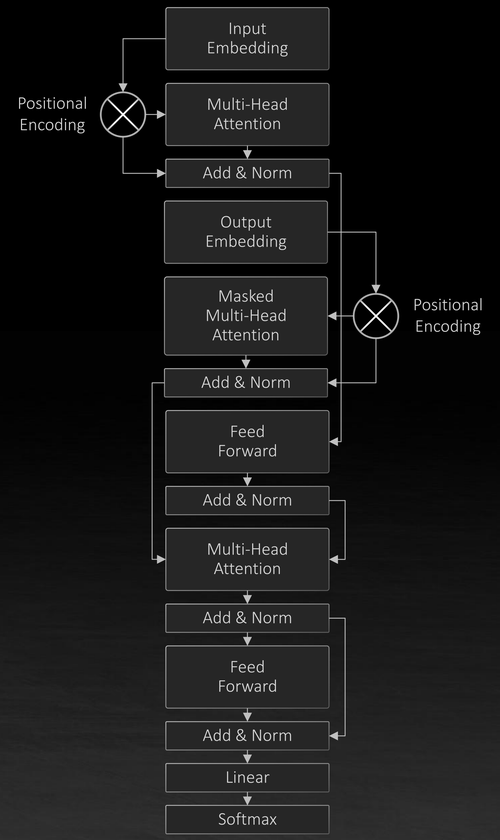


THE FIRST ERA OF SOFTWARE DEVELOPMENT

Single-Thread Era
(until ~2007)

Straight-Line Code

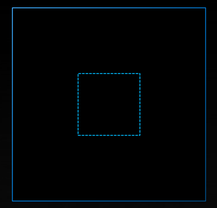


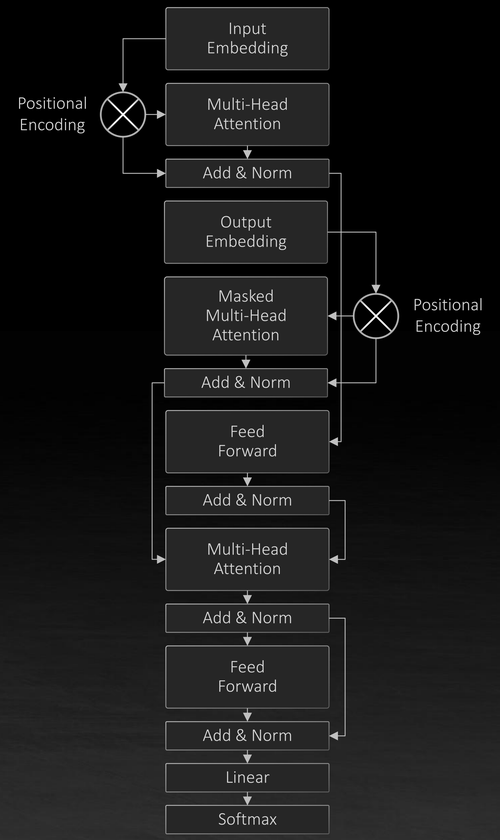


THE FIRST ERA OF SOFTWARE DEVELOPMENT

Single-Thread Era
(until ~2007)

Straight-Line Code

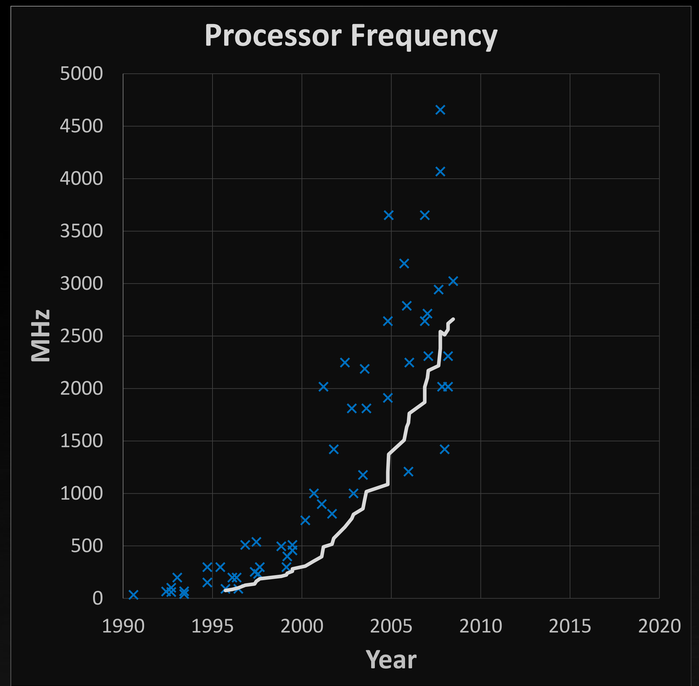




THE FIRST ERA OF SOFTWARE DEVELOPMENT

Single-Thread Era
(until ~2007)

Straight-Line Code

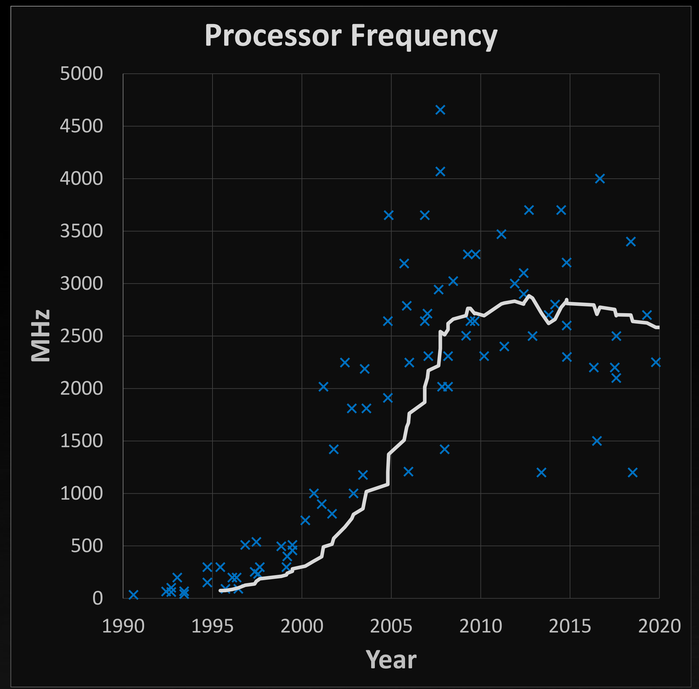


Source: Karl Rupp “40 years of microprocessor trend data”
https://github.com/karlrupp/microprocessor-trend-data

THE FIRST ERA OF SOFTWARE DEVELOPMENT

Single-Thread Era
(until ~2007)

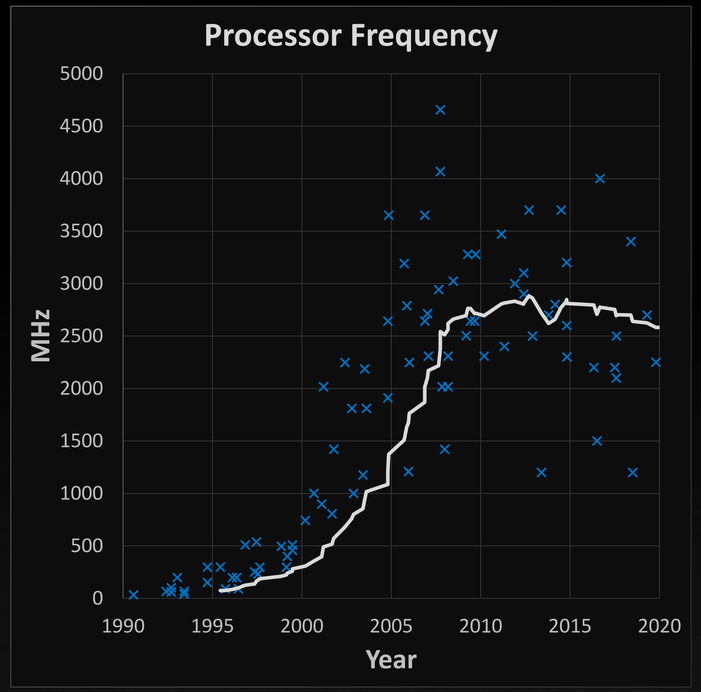
Straight-Line Code



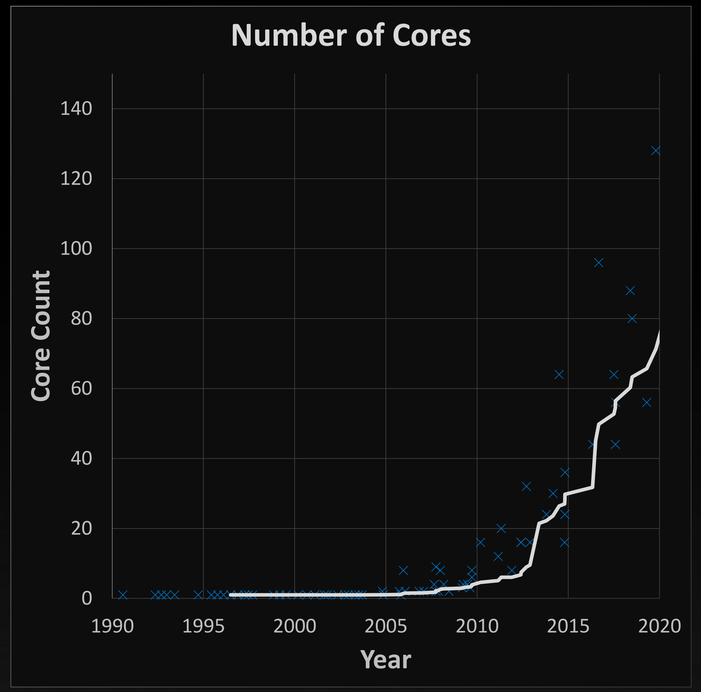
Source: Karl Rupp “40 years of microprocessor trend data”
https://github.com/karlrupp/microprocessor-trend-data

Single-Thread Era
(until ~2007)

Straight-Line Code



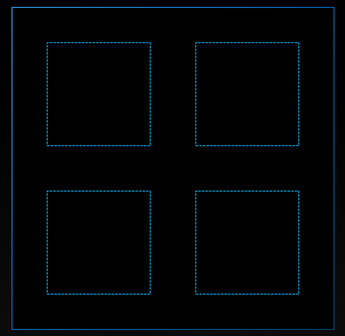
Multi-Core Era
(~2007 → Present)



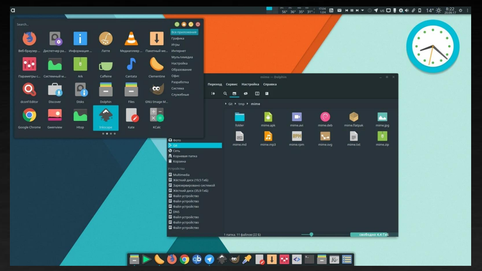
Source: Karl Rupp “40 years of microprocessor trend data”
https://github.com/karlrupp/microprocessor-trend-data

THE SECOND ERA OF SOFTWARE DEVELOPMENT

Multi-Core Era
(~2007 → Present)

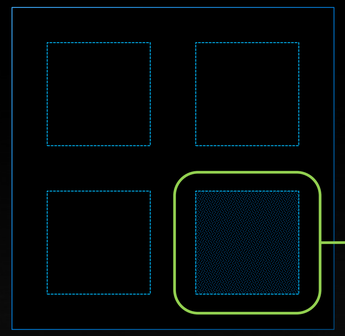


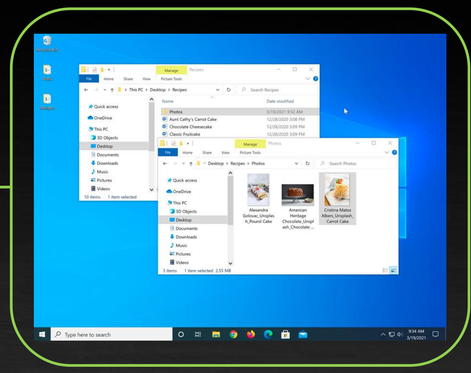
THE SECOND ERA OF SOFTWARE DEVELOPMENT



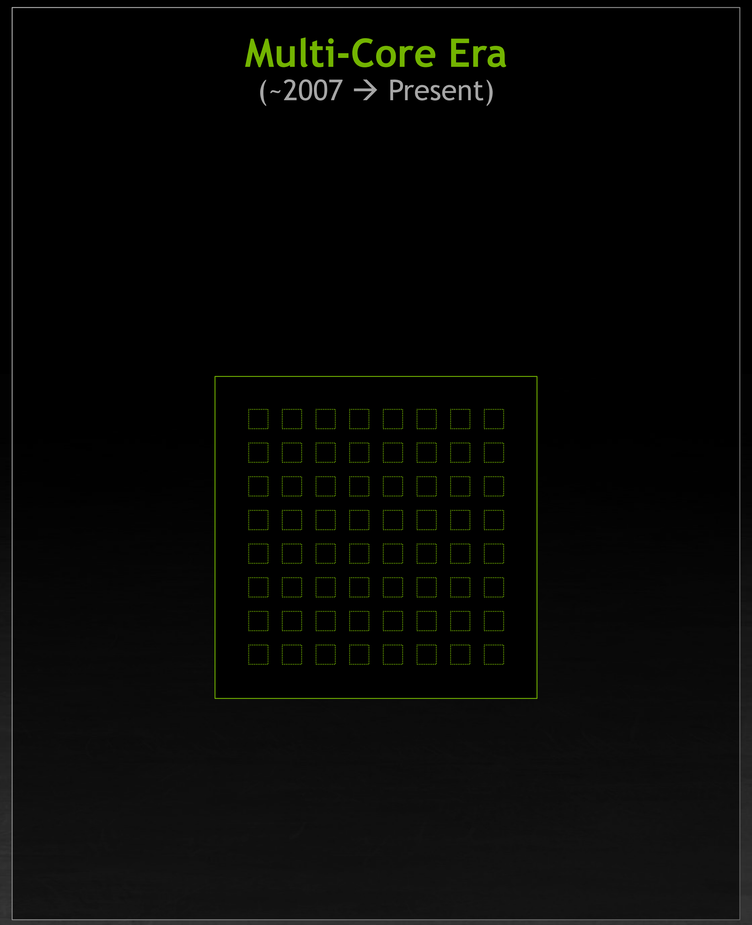
Multi-Core Era

(~2007 → Present)

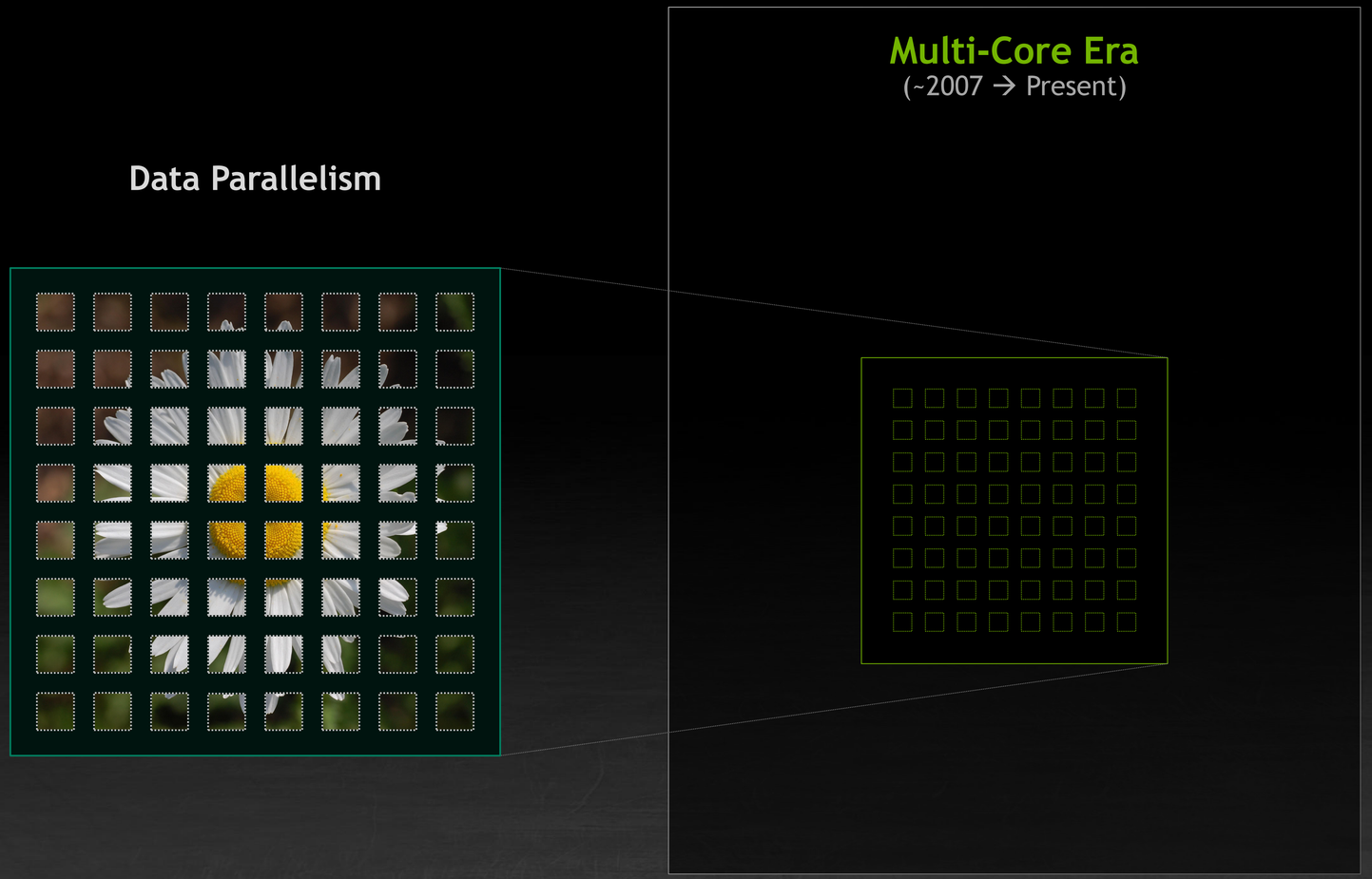




THE SECOND ERA OF SOFTWARE DEVELOPMENT

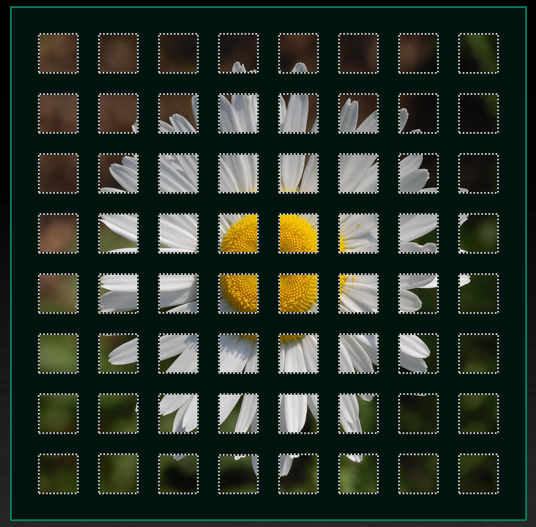


THE SECOND ERA OF SOFTWARE DEVELOPMENT



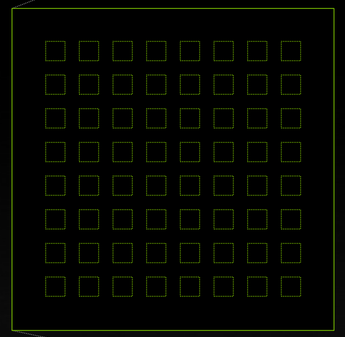
THE SECOND ERA OF SOFTWARE DEVELOPMENT

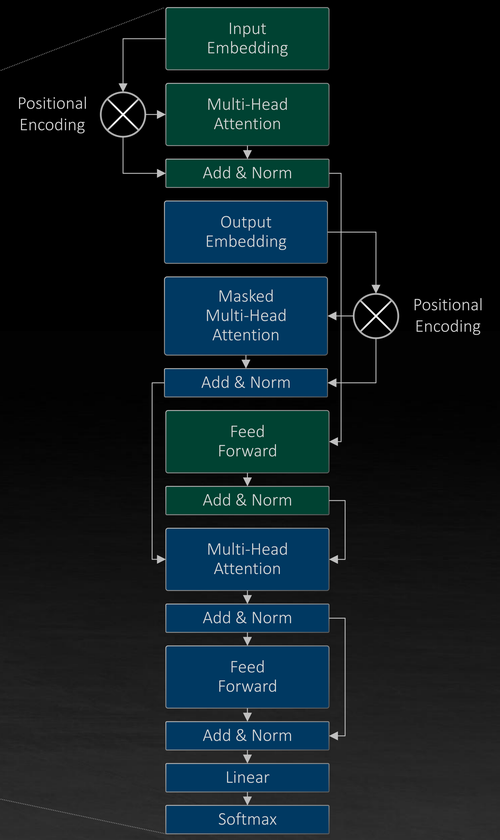
Data Parallelism



Multi-Core Era

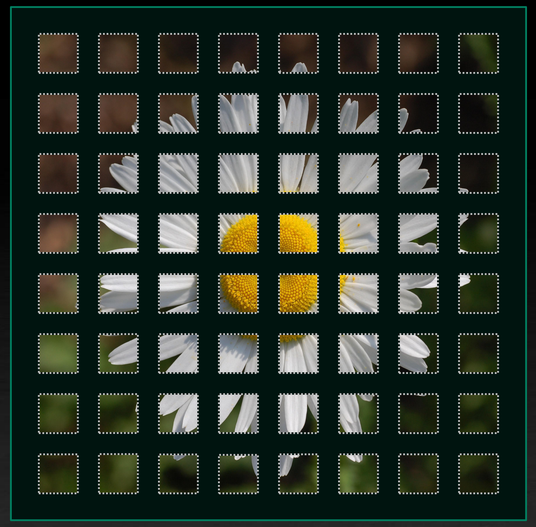
(~2007 → Present)





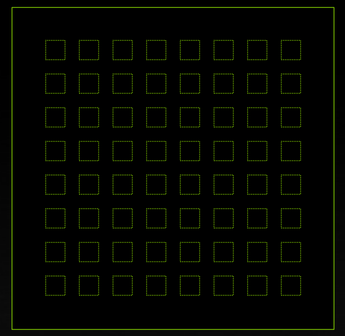
THE SECOND ERA OF SOFTWARE DEVELOPMENT

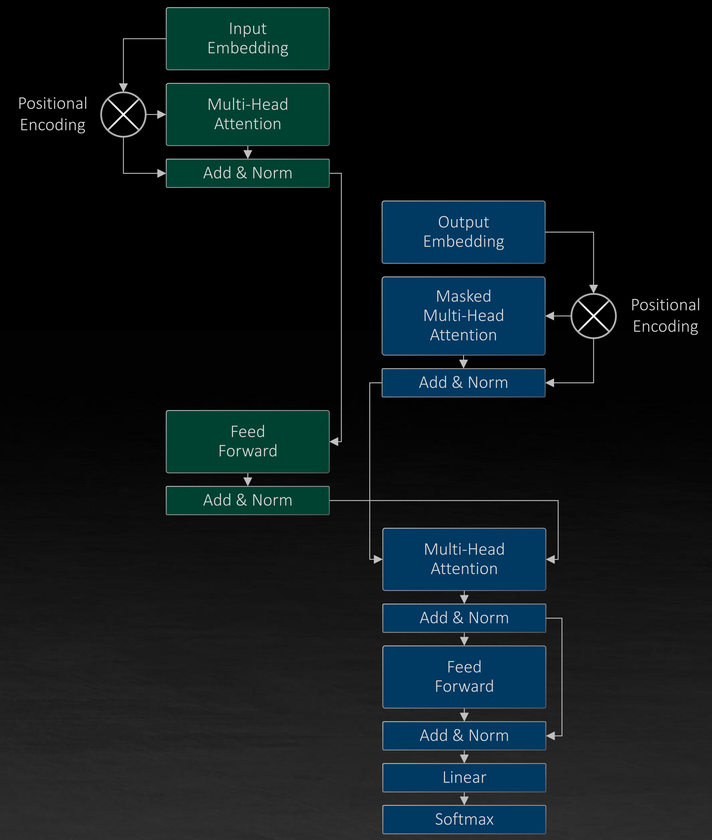
Data Parallelism



Multi-Core Era

(~2007 → Present)

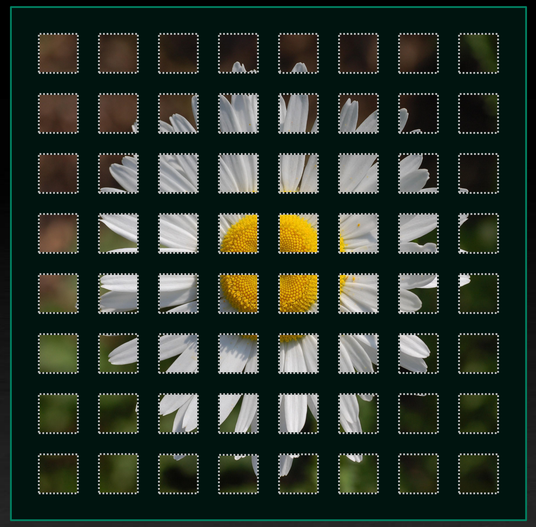




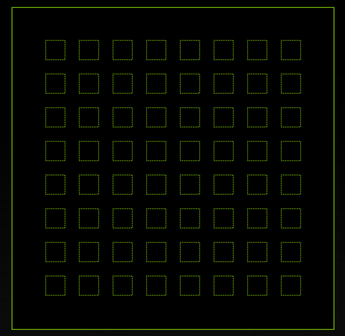
THE SECOND ERA OF SOFTWARE DEVELOPMENT

Multi-Core Era
(~2007 → Present)

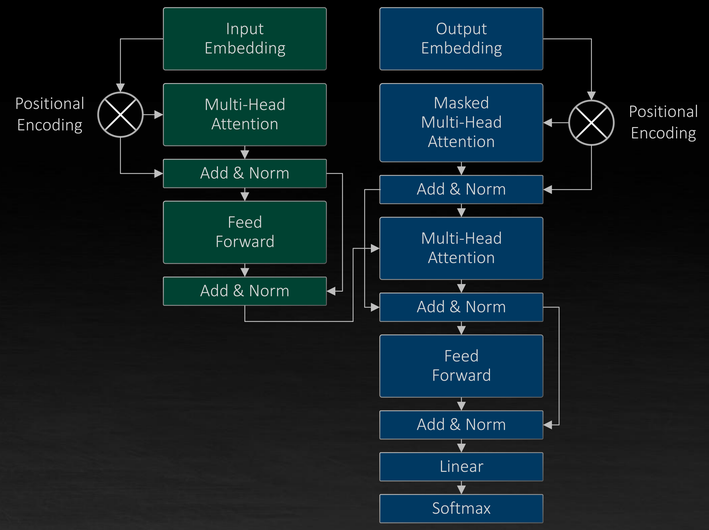
Data Parallelism



Data- and Task-Parallelism



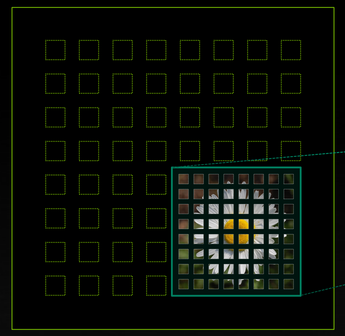
Task Parallelism



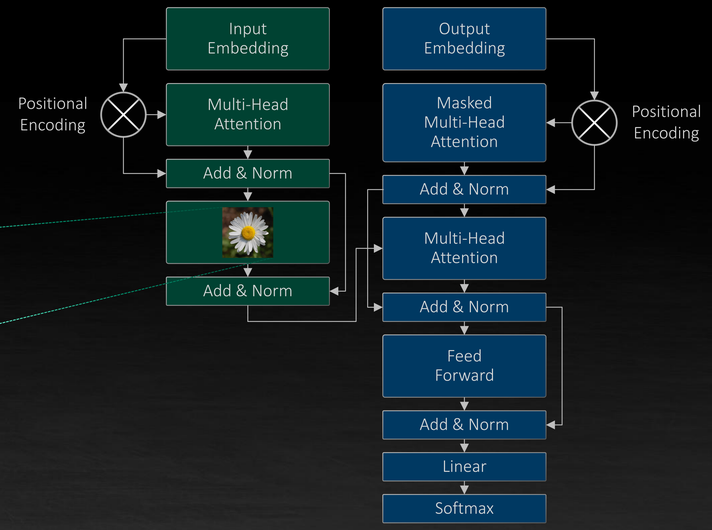
HIERARCHICAL PARALLELISM

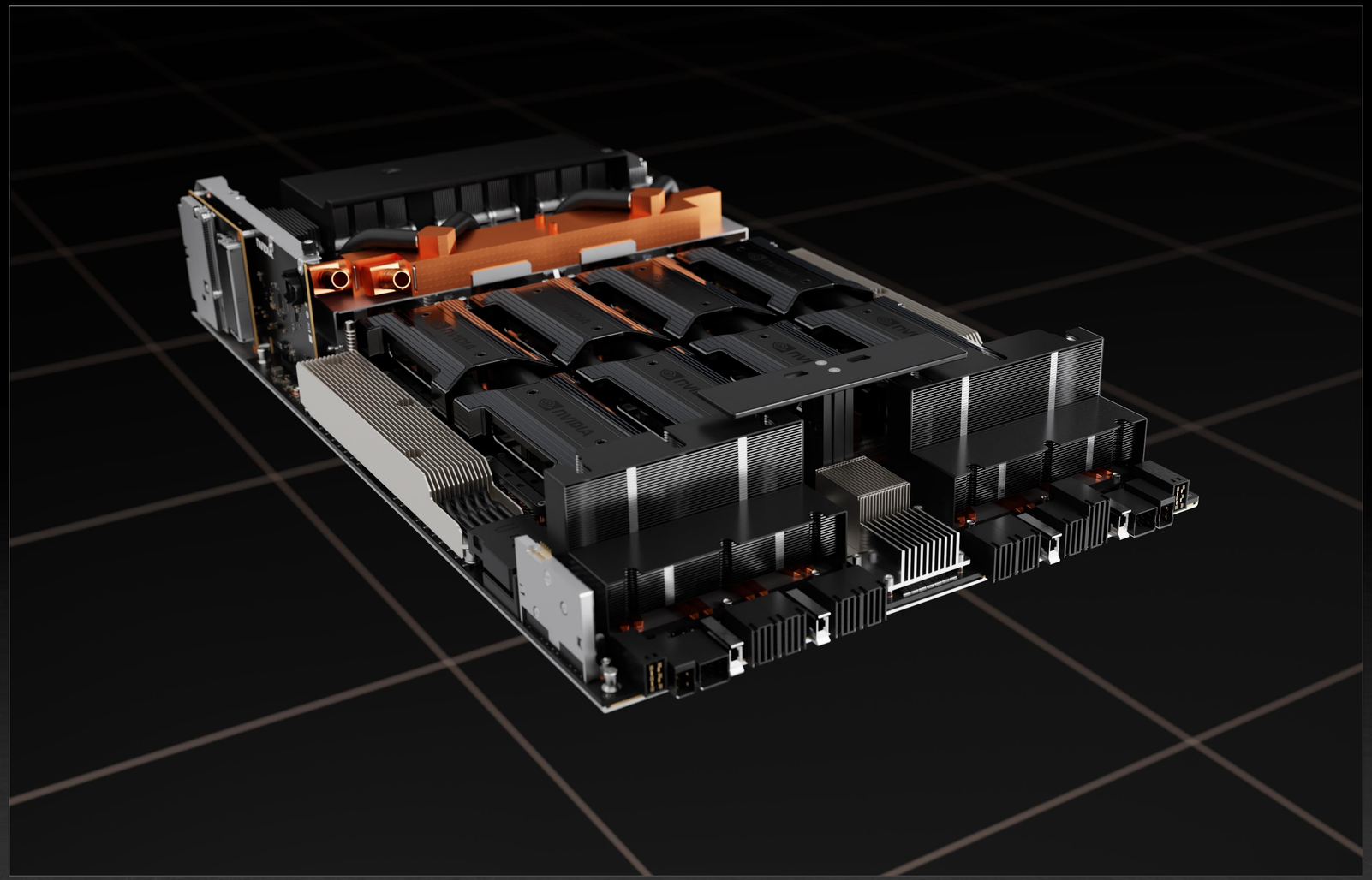
Multi-Core Era
(~2007 → Present)

Data- and Task-Parallelism

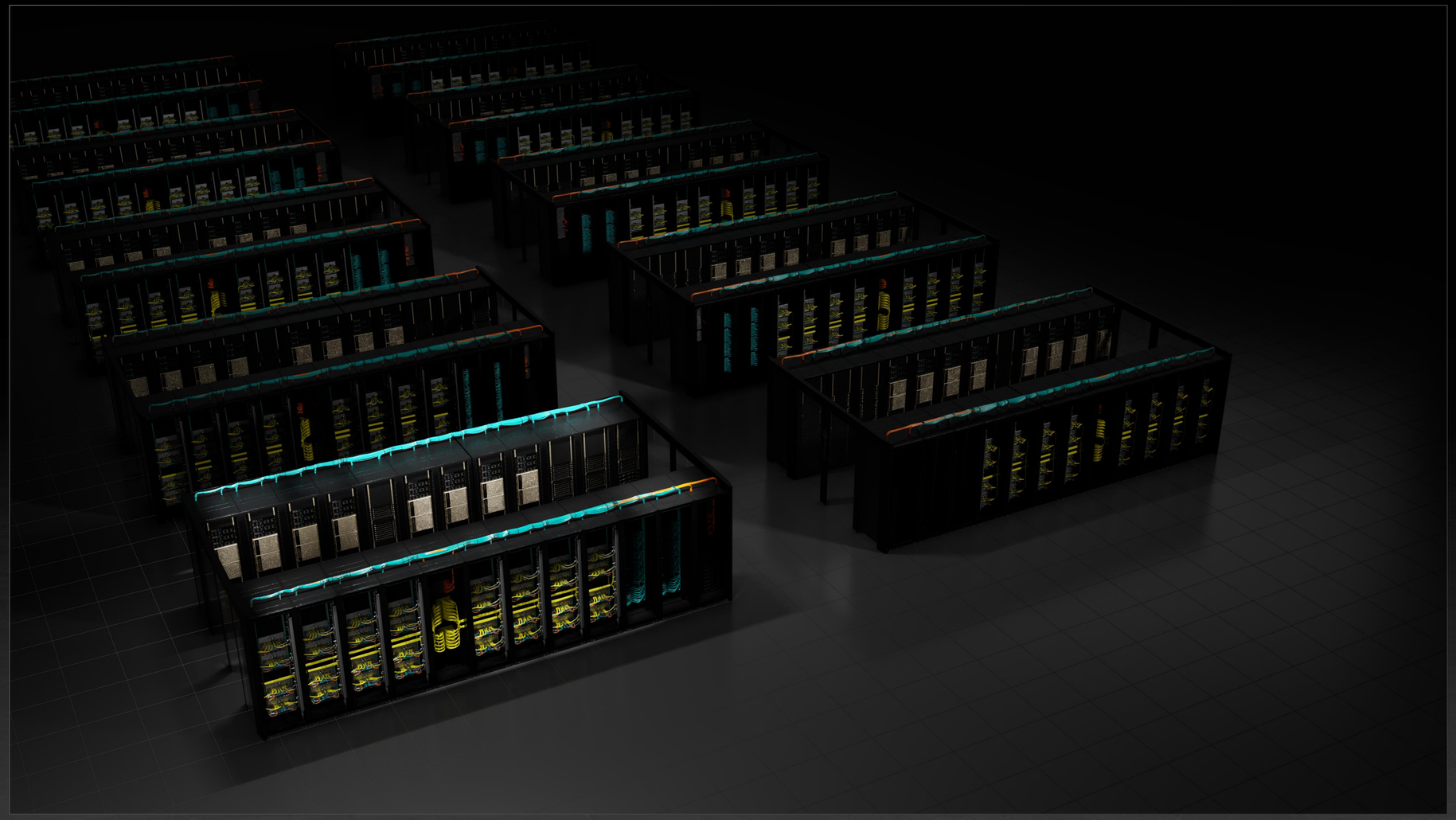


Task Parallelism

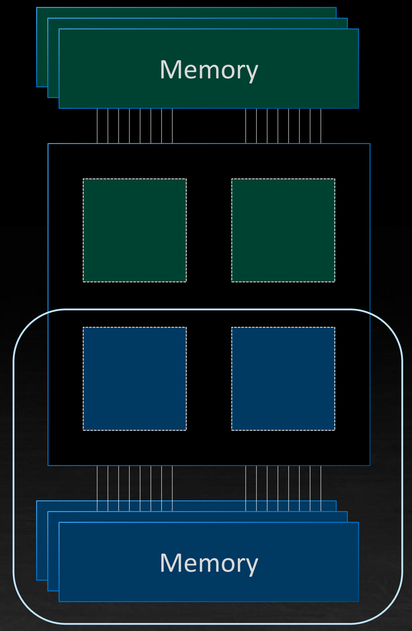




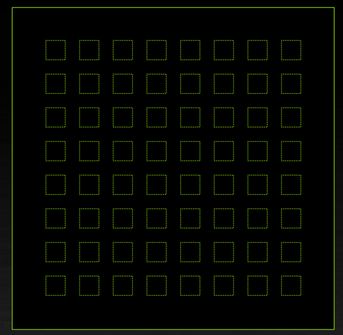
DATACENTER-SCALE COMPUTING



MANAGING LOCALITY IS NOT NEW

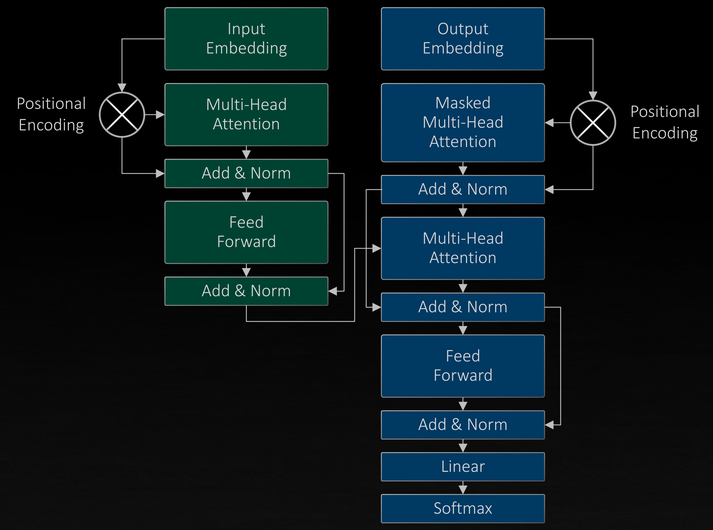


THE THIRD ERA OF SOFTWARE DEVELOPMENT

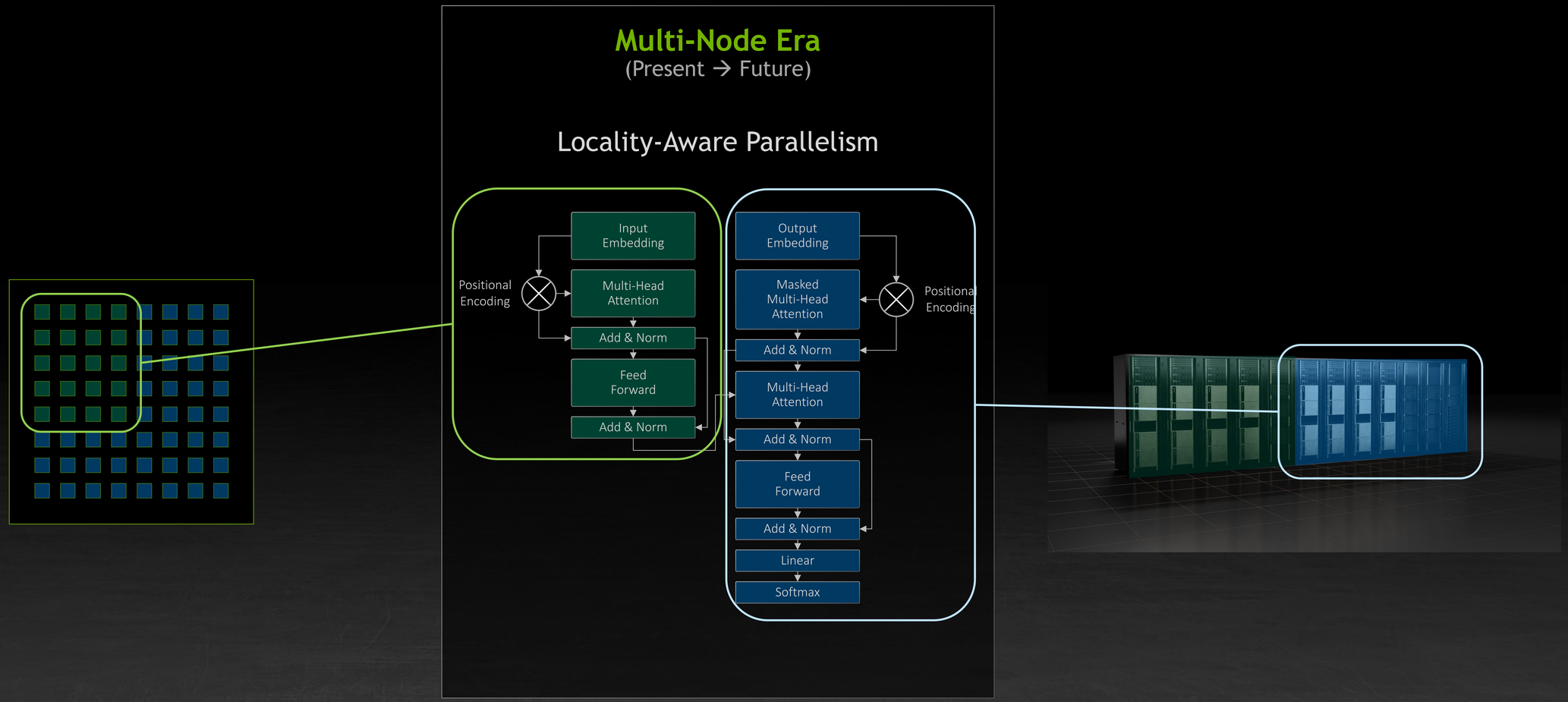


Multi-Node Era
(Present → Future)

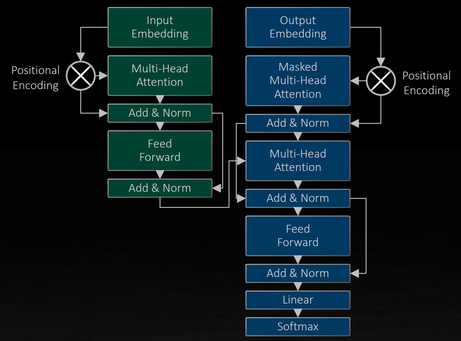
Locality-Aware Parallelism



LOCALITY: THE THIRD ERA OF SOFTWARE DEVELOPMENT

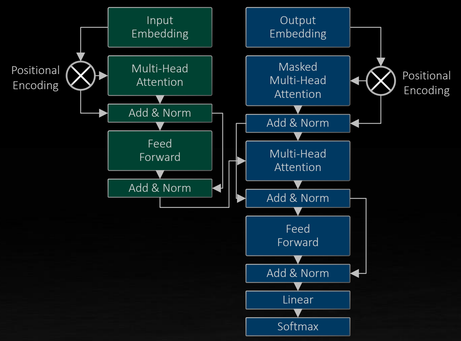


PROGRAMMING TO THE HIERARCHY



Application level

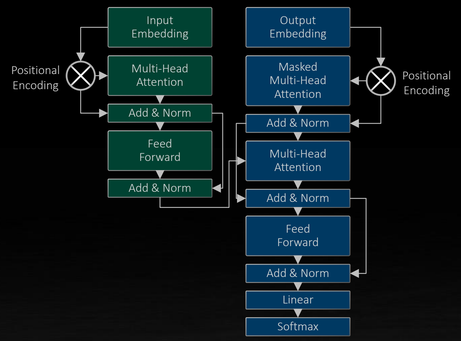
PROGRAMMING TO THE HIERARCHY



Application level

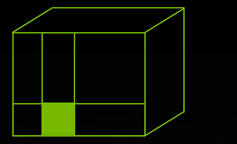
Framework level

PROGRAMMING TO THE HIERARCHY



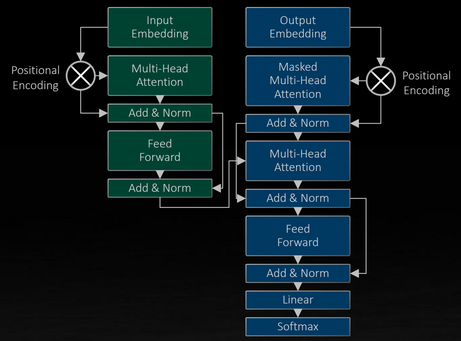
Application level

Framework level



Library level

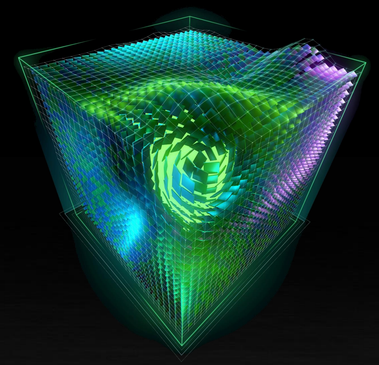
PROGRAMMING TO THE HIERARCHY



Application level

Framework level

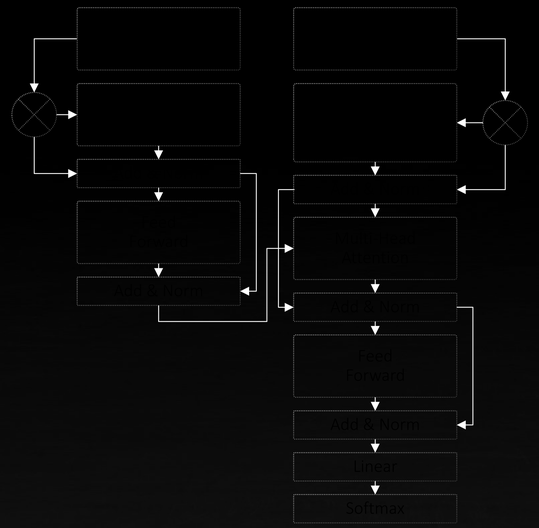
Library level



Runtime level

SCALING: TASK PARALLELISM + LOCALITY OF EXECUTION

Scheduling based on dependencies

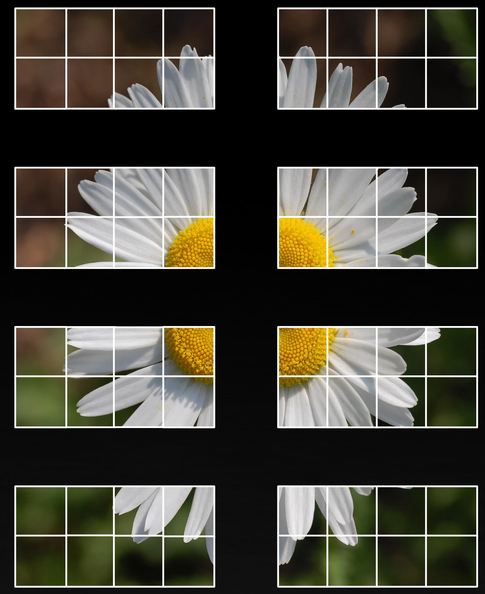


Placement based on data exchange

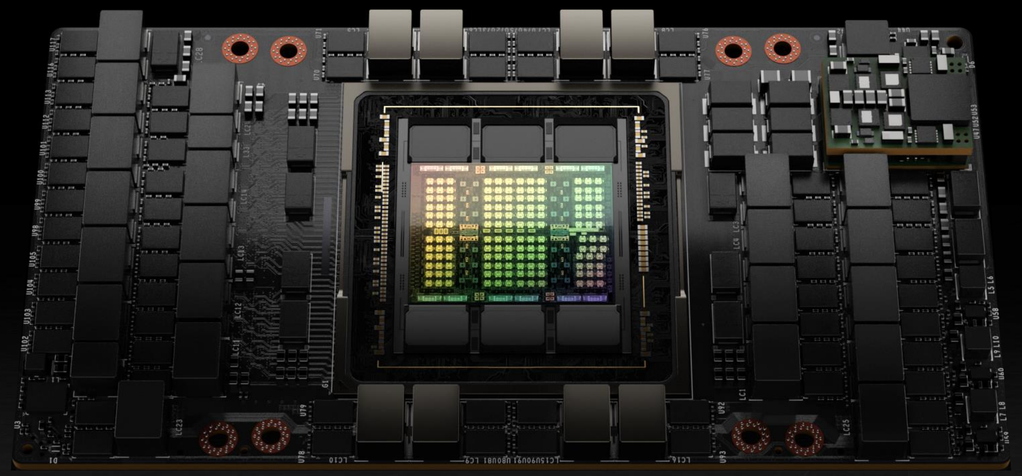
SCALING: DATA PARALLELISM + LOCALITY OF DATA

Scheduling based on throughput

Placement based on data reuse



INTRODUCING HOPPER



H100 Physical Architectural Features

132 Streaming Multiprocessors (SMs)

PCIe Gen5 with PCIe Atomics

HBM3 Memory with 3TB/sec Bandwidth

50MB L2 Cache

4th Generation NVLink @ 900GB/sec total bandwidth

H100 Next-Generation Capabilities

Thread Block Clusters

Distributed Shared Memory

Tensor Memory Accelerator (TMA)

Tensor Core Transformer Engine

Confidential Computing Support

Asynchronous Architecture

Inside the NVIDIA Hopper Architecture [S42663]

INTRODUCING HOPPER

H100 SM

256kB Shared Memory / L1-Cache

DPX Instruction Set

4th Generation Tensor Core

Fully-Asynchronous Architecture

Thread Block Clusters

Tensor Memory Accelerator

Asynchronous Transaction Barriers

SOME HISTORY: THE KEPLER GK110 GPU, 2012

Kepler GK110 Full Chip

15 SMs

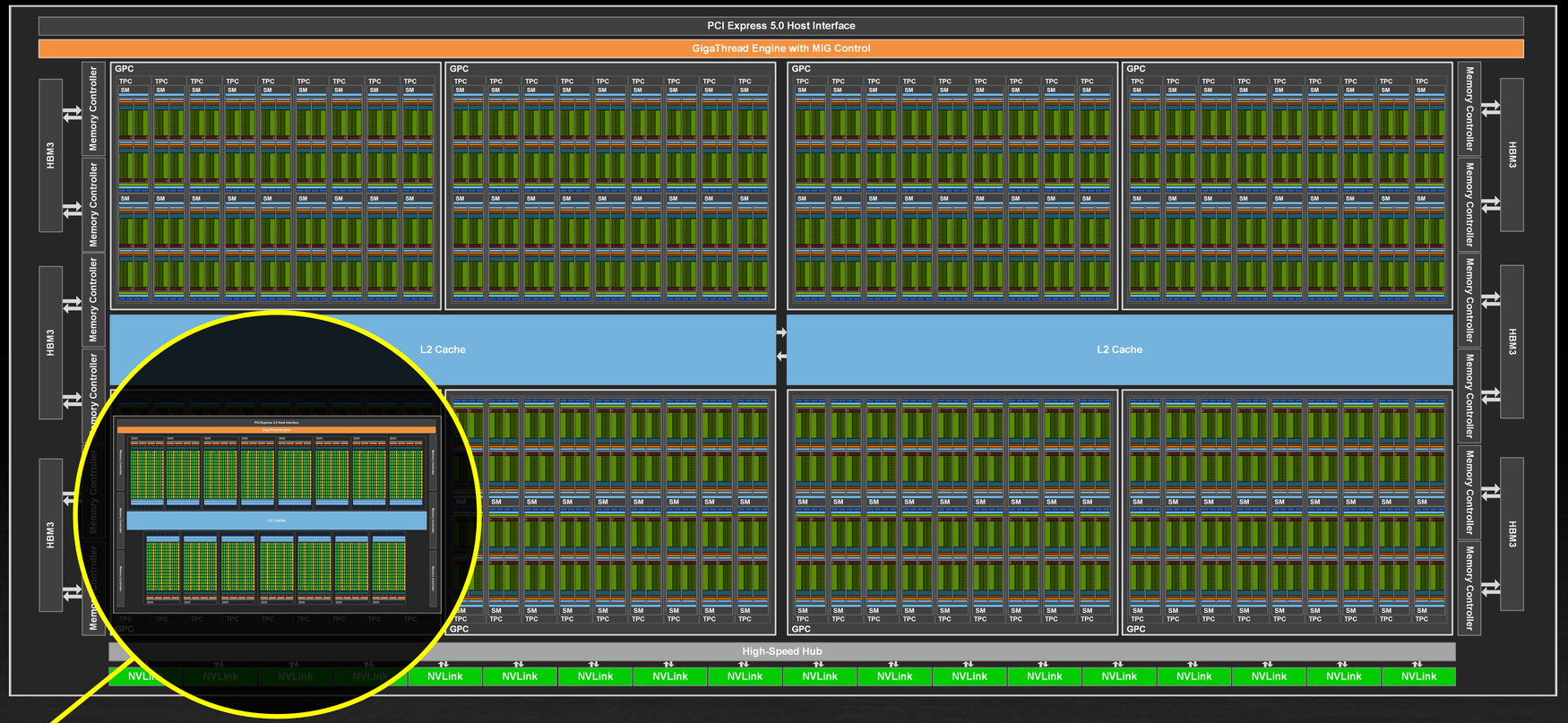
THE HOPPER H100 GPU, 2022

Hopper H100 Full Chip

132 SMs

NVIDIA

9x SMs OVER 10 YEARS



Hopper H100 Full Chip

132 SMs

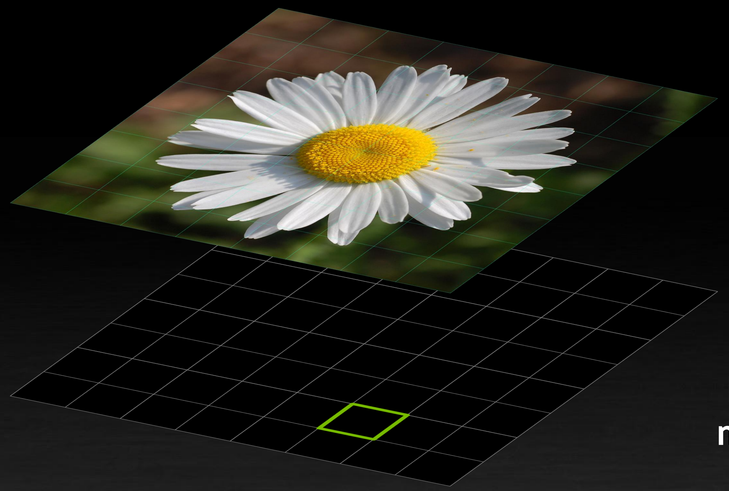
Kepler GK110 Full Chip

15 SMs

THE CUDA PROGRAMMING MODEL: GRID → BLOCKS → THREADS

Grid
of work

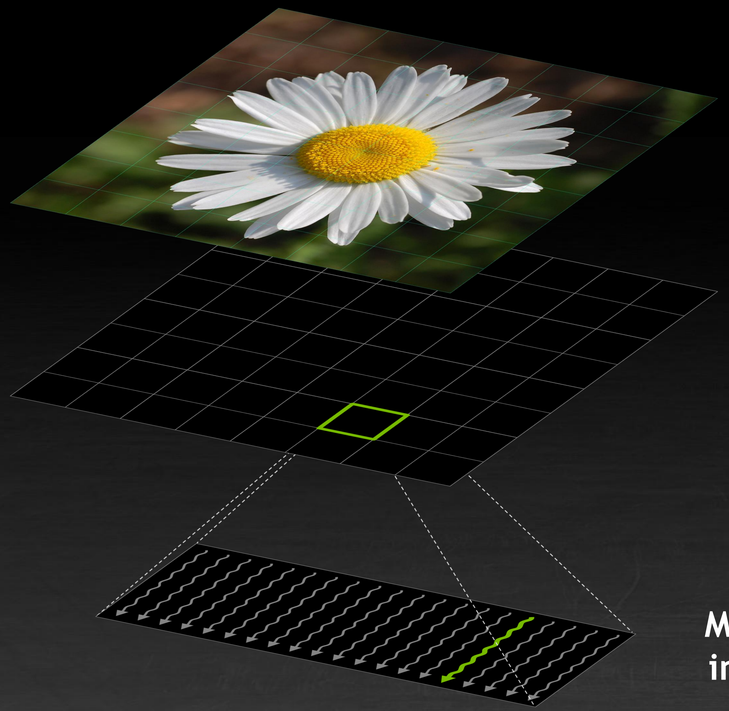
DIVIDE THE WORK INTO A GRID OF EQUAL BLOCKS



Grid
of work

**Divide into
many Blocks**

EACH BLOCK RUNS AS IF IT'S AN INDEPENDENT PROGRAM



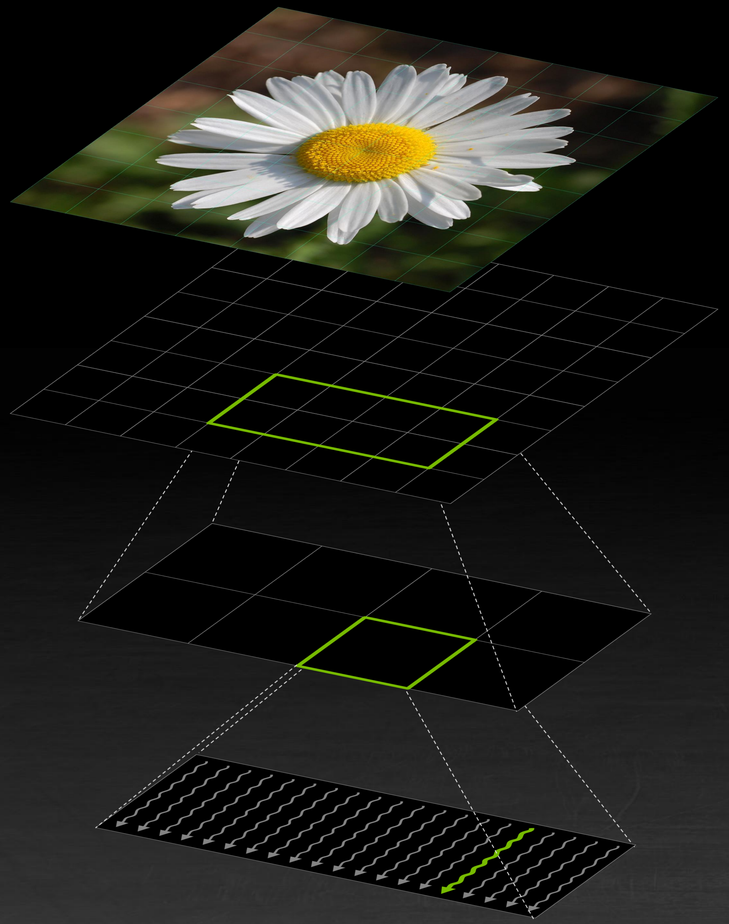
Grid
of work

Blocks
of Threads

**Many Threads
in each Block**

THREAD BLOCK CLUSTER

A collective of blocks, co-scheduled on adjacent multiprocessors

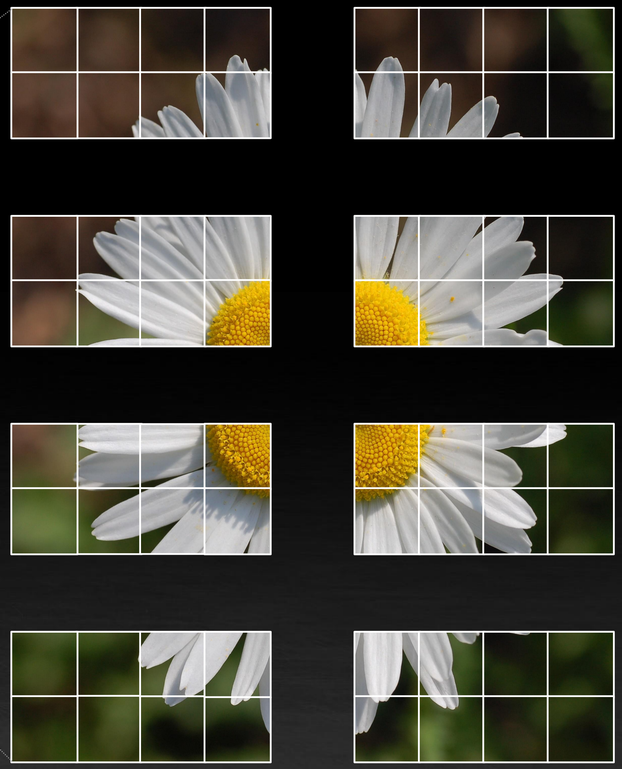


Grid
of work

**Cluster
of Blocks**

Blocks
of Threads

Threads

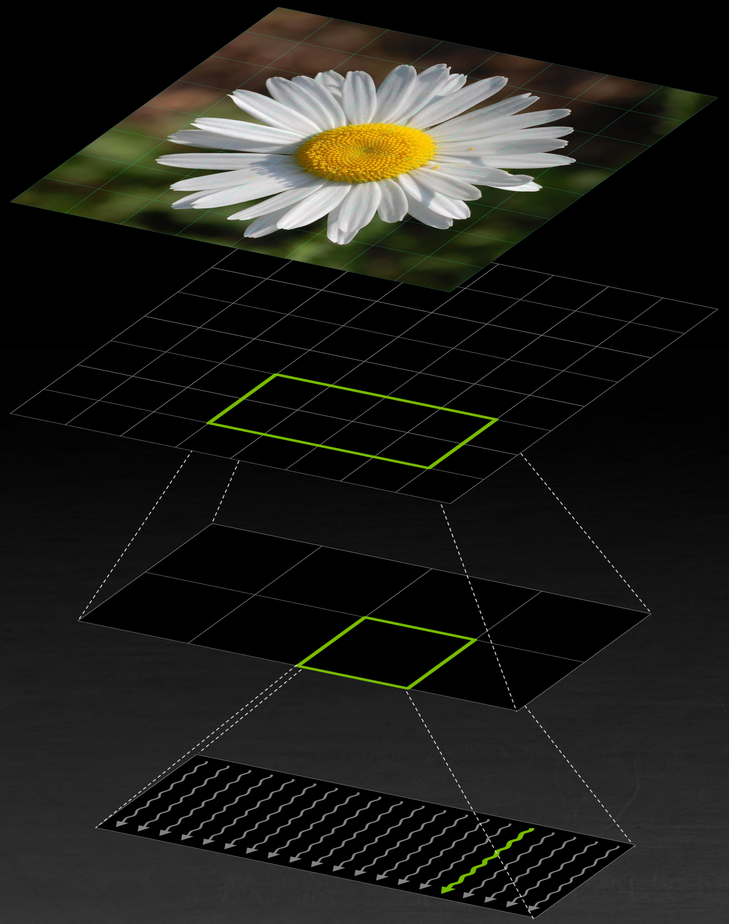


TAKING ADVANTAGE OF LOCALITY AT A GPU SCALE

Guaranteed co-located blocks
New tier of guaranteed concurrency
Fast data exchange & sync

THREAD BLOCK CLUSTER

Building hierarchy into a program



A cluster is a collective of up to 16 blocks

Guaranteed to be on different SMs

Guaranteed to be running at the same time

1D, 2D or 3D, just like blocks

Annotate a kernel with its required cluster size

New cluster dimension annotation for __global__ functions:

```
__cluster_dims__(x, [y, [z]])
```

```
__cluster_dims__(4, 2, 1)            // 8-block cluster of size 4x2x1
__global__ void helloCluster()
{
    cooperative_groups::cluster_group cluster = this_cluster();
    cluster.sync();

    printf("Hello from cluster elem %d\n", cluster.cluster_rank());
}
```

Plus: New extensible launch API allows configuration at launch time

CLUSTER DISTRIBUTED SHARED MEMORY (DSMEM)

Blocks within a cluster are able to access each others' shared memory directly

Full load/store/atomic access to all shared memory between blocks within a cluster

CLUSTER DISTRIBUTED SHARED MEMORY (DSMEM)

Blocks within a cluster are able to access each others' shared memory directly

Distributed Shared Memory (DSMEM) Programming Model

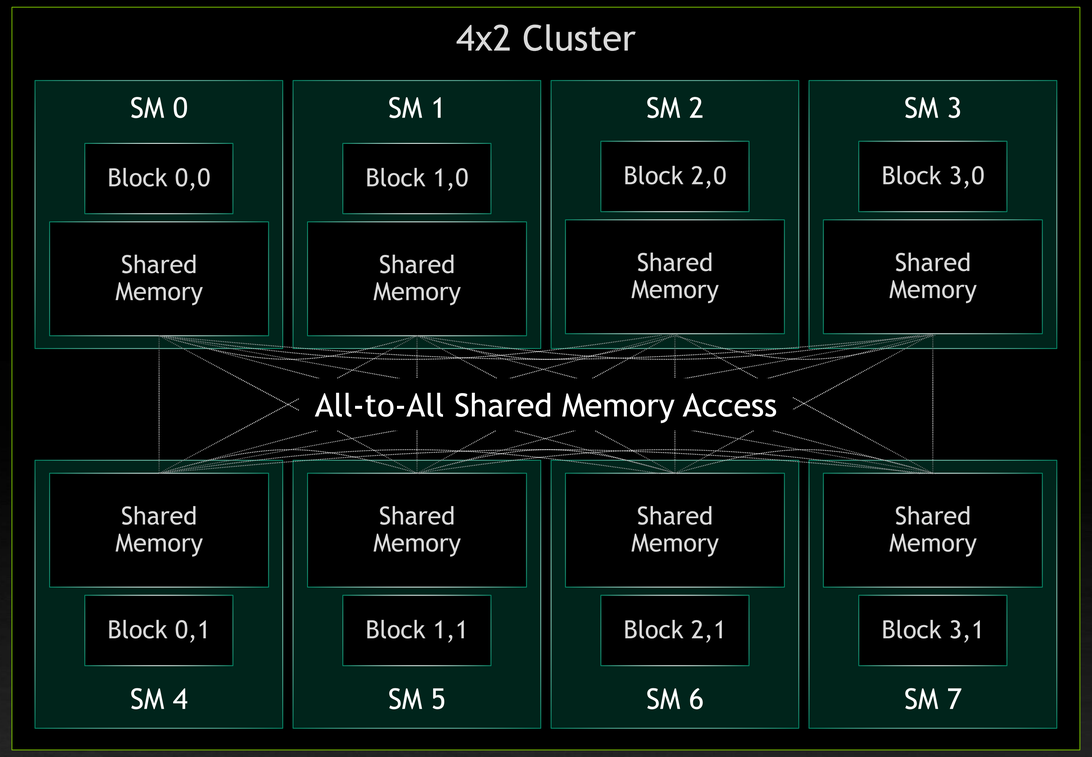
All blocks in a cluster can collaborate using DSMEM

Blocks in a cluster can synchronize together via **barriers in DSMEM**

DSMEM is laid out as a **Partitioned Global Address Space**

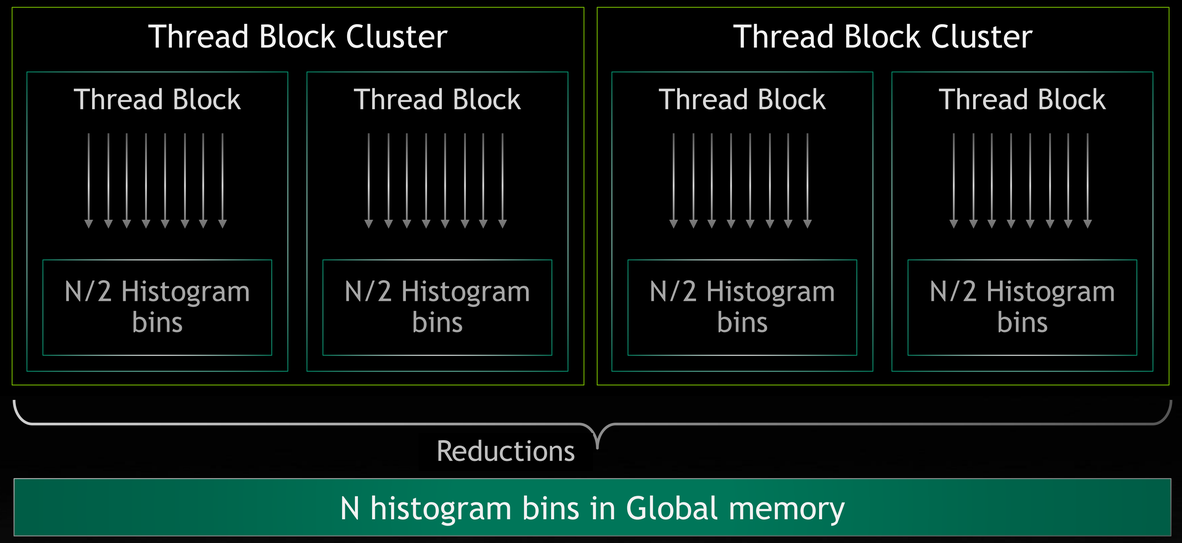
DSMEM may be indexed by block **rank** – a 1D expansion of cluster index – or by (xyz) index in cluster

```
// Example of getting DSMEM pointer from cluster neighbours
__shared__ int x;

cg::cluster_group = this_cluster();
int *x_ptr = cluster.map_shared_rank(&x, 2);
*x_ptr = 5;

cluster_barrier_arrive();      // Hardware-managed barrier
cluster_barrier_wait();
```



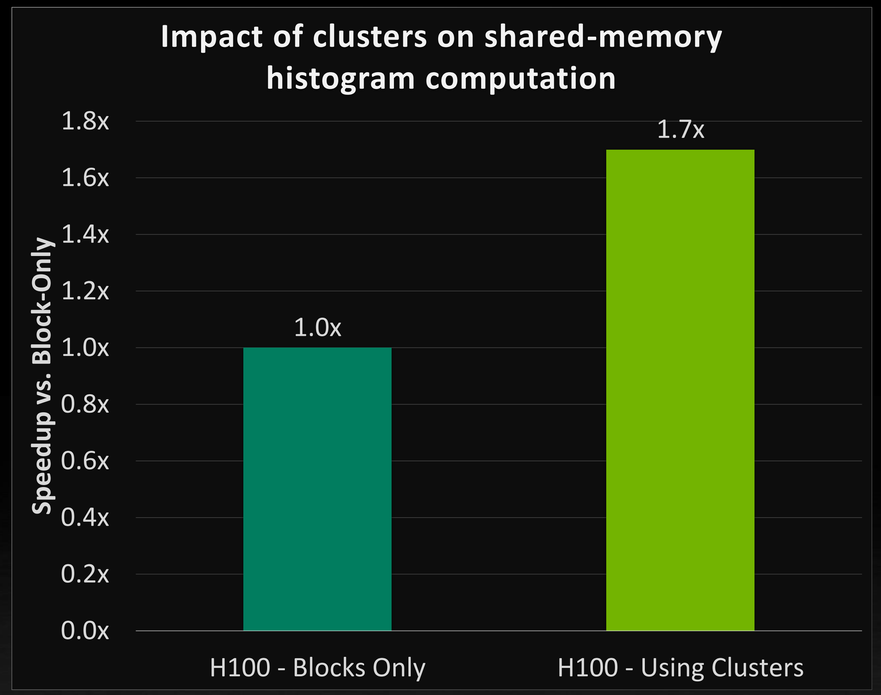
Full load/store/atomic access to all shared memory between blocks within a cluster

EXAMPLE: HIERARCHICAL HISTOGRAM USING CLUSTER DSMEM



Histograms in CUDA are typically computed in shared memory, followed by reductions in global memory.

For large histograms, shared memory capacity of a single block is not sufficient.

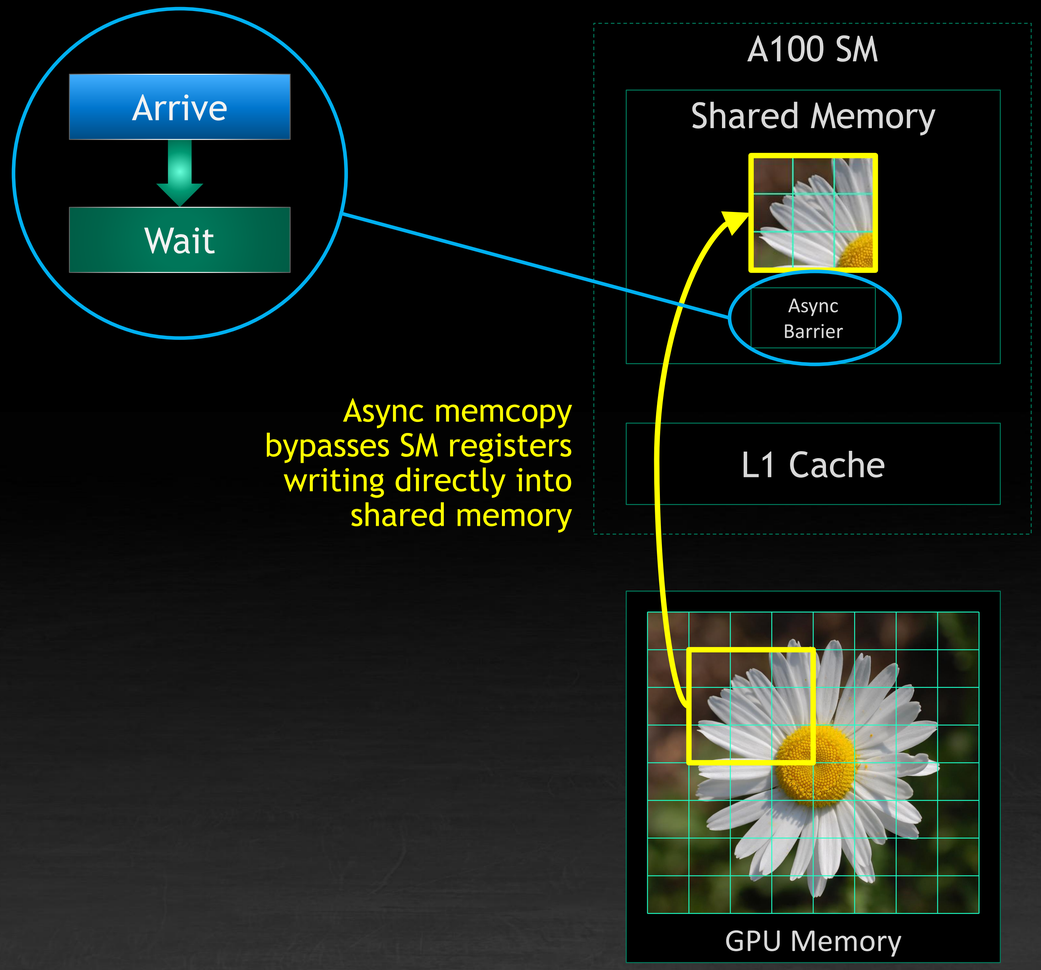


75K Histogram bins (300KB) fit in distributed shared memory of 2-block clusters → 37.5K (150KB) per thread block

Optimizing CUDA Applications for NVIDIA Hopper Architecture [S41489]

ASYNCHRONOUS COPY TO SHARED MEMORY

cuda::memcpy_async



cuda::memcpy_async()

Bypasses SM registers to write directly to shared

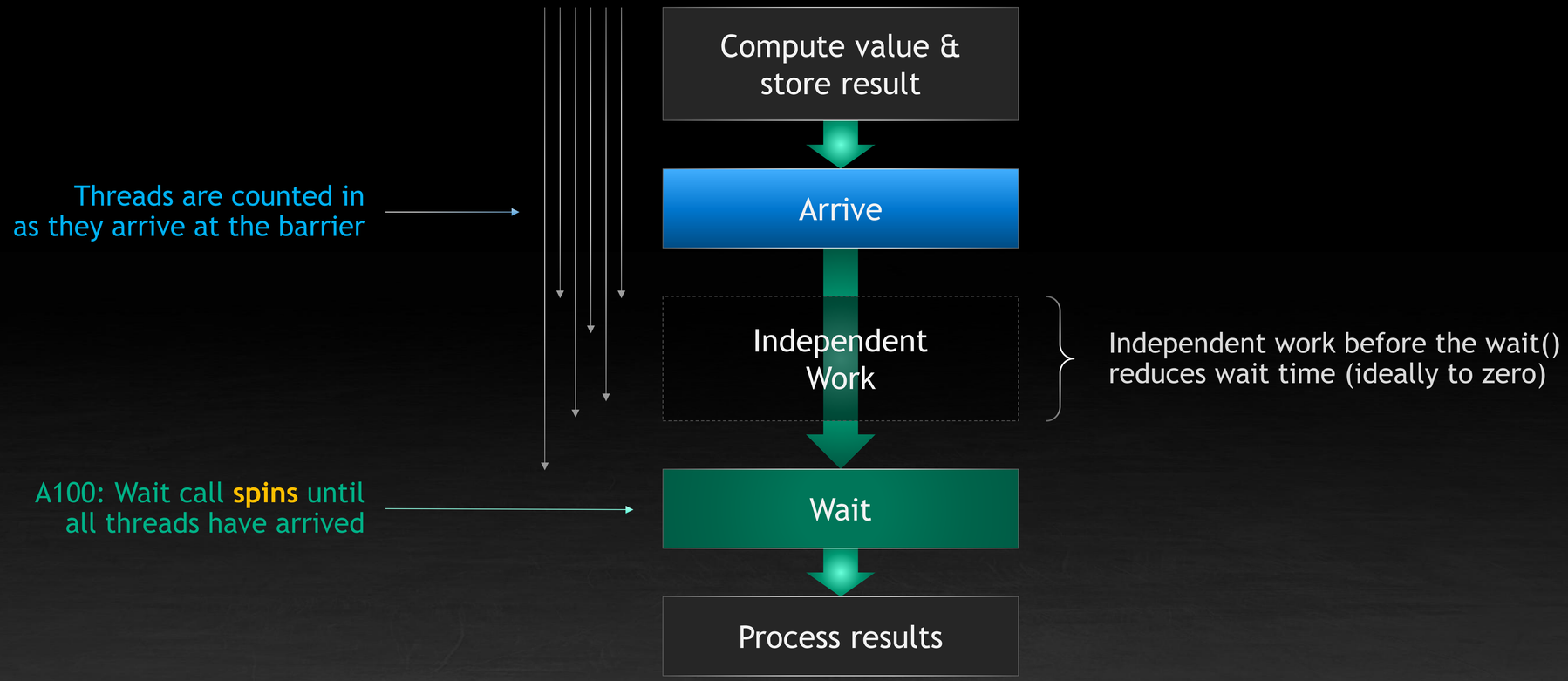
Uses Asynchronous Barrier to signal completion

Threads work together to copy data in parallel

```
// Async block-wide memcopy using Cooperative Groups
cg::thread_block tb = cg::this_thread_block();
size_t copy_count, index = 0;
while (index < elementsPerThreadBlock) {
    cg::memcpy_async(
        tb,
        local_smem,
        elementsInShared,
        global_data + index,
        elementsPerThreadBlock - index);
    copy_count = min(elementsInShared,
                     elementsPerThreadBlock - index);
    cg::wait(tb);

    // ... use data copied into shared memory ...
    index += copyCount;
}
```

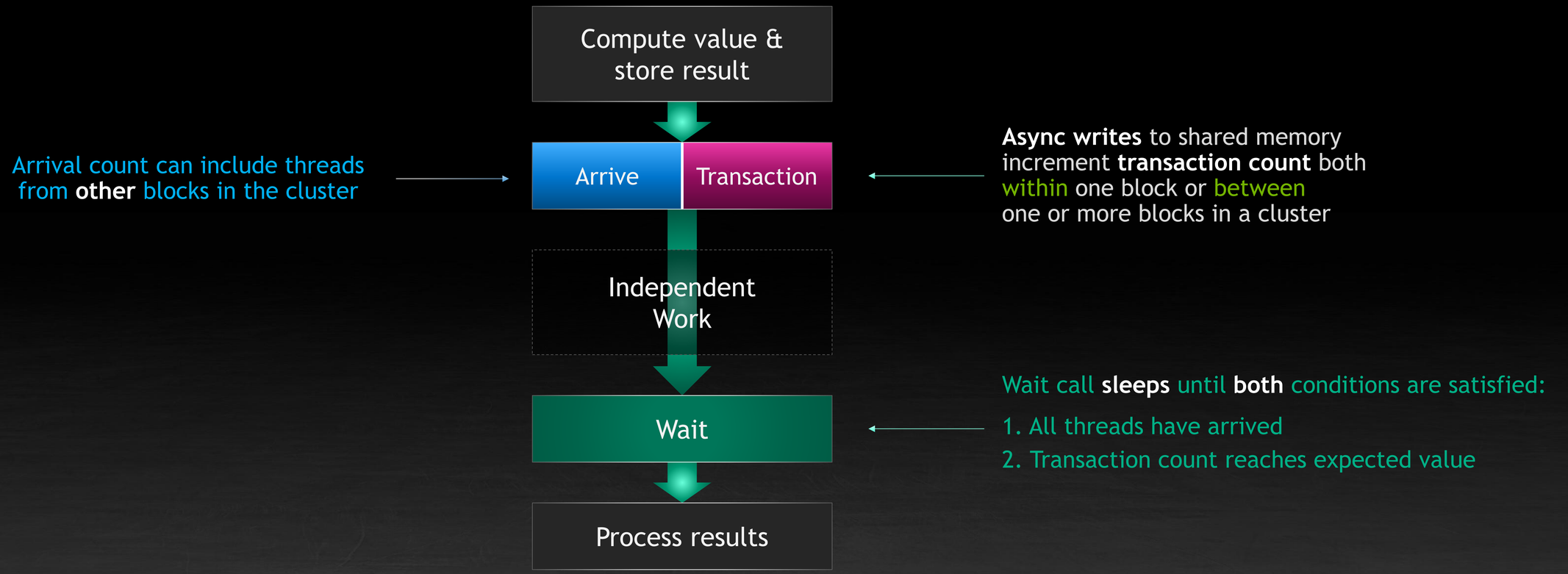
ASYNCHRONOUS BARRIERS

cuda::barrier



ASYNCHRONOUS TRANSACTION BARRIERS

Data can now trigger a barrier as well as just threads



Drop-in enhancement to existing cuda::barrier

ASYNCHRONOUS ONE-SIDED MEMORY COPIES

cuda::memcpy_async within & between blocks in a cluster automatically updates transaction count without handshake

Data arrival increments transaction count

One-sided communication within cluster

Accepts async store **and** atomic operations

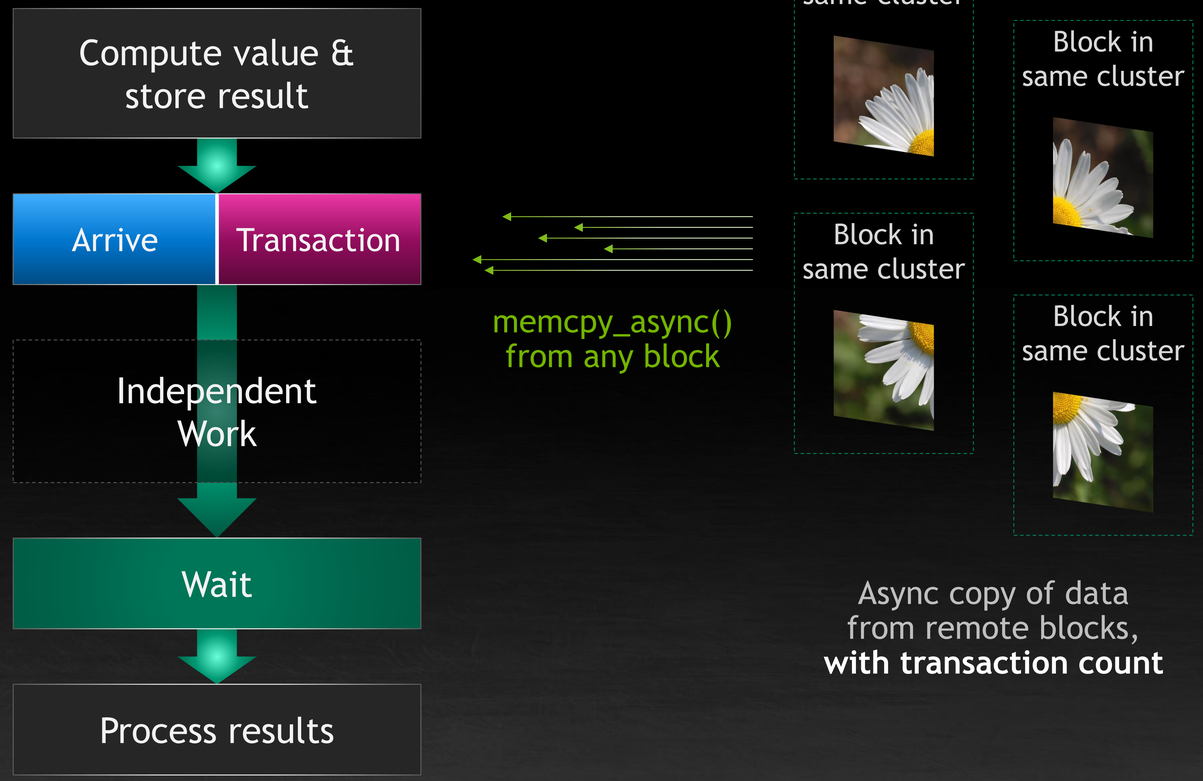
Highly efficient control flow

Self-synchronizing: no memory fence required to update transaction count

7x faster than { data + fence + flag } handshake

Receiver threads **sleep** on wait (instead of busy-waiting) until data arrives

Fast wakeup signal directly from barrier

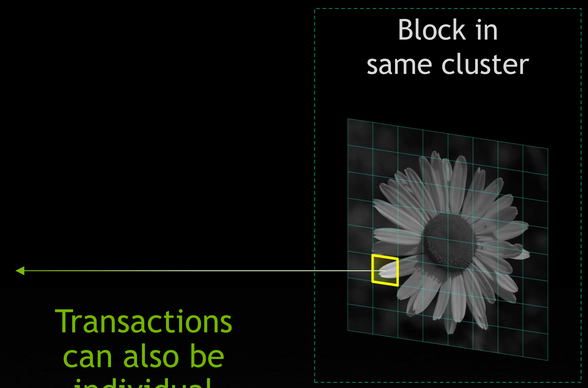


ASYNCHRONOUS ONE-SIDED COMMUNICATION

No-handshake messaging & data transfer within a block or a cluster

```
cuda::atomic_ref ref;                    // Initialization not shown
cuda::barrier admirer;                   // Initialization not shown

// Remote store with transaction count
void atomic_store_async(ref, value, barrier);

// Remote atomic reduction operations
void atomic_add_async(ref, value, barrier);
void atomic_min_async(ref, value, barrier);
void atomic_max_async(ref, value, barrier);
void atomic_inc_async(ref, value, barrier);
void atomic_dec_async(ref, value, barrier);
void atomic_and_async(ref, value, barrier);
void atomic_or_async(ref, value, barrier);
void atomic_xor_async(ref, value, barrier);

// Note: CAS is not supported as it is not a reduction operation
```

For these individual async store/atomic operations,
initialise barrier to expected message size before sending



Transactions can also be individual async stores and atomics

Async write of data from remote block, **with transaction count**

TENSOR MEMORY ACCELERATOR UNIT (TMA) FOR ASYNC DATA MOVEMENT

Example: One-sided halo exchange within a cluster

Tensor Memory Accelerator Unit

Hardware-accelerated bi-directional bulk copy

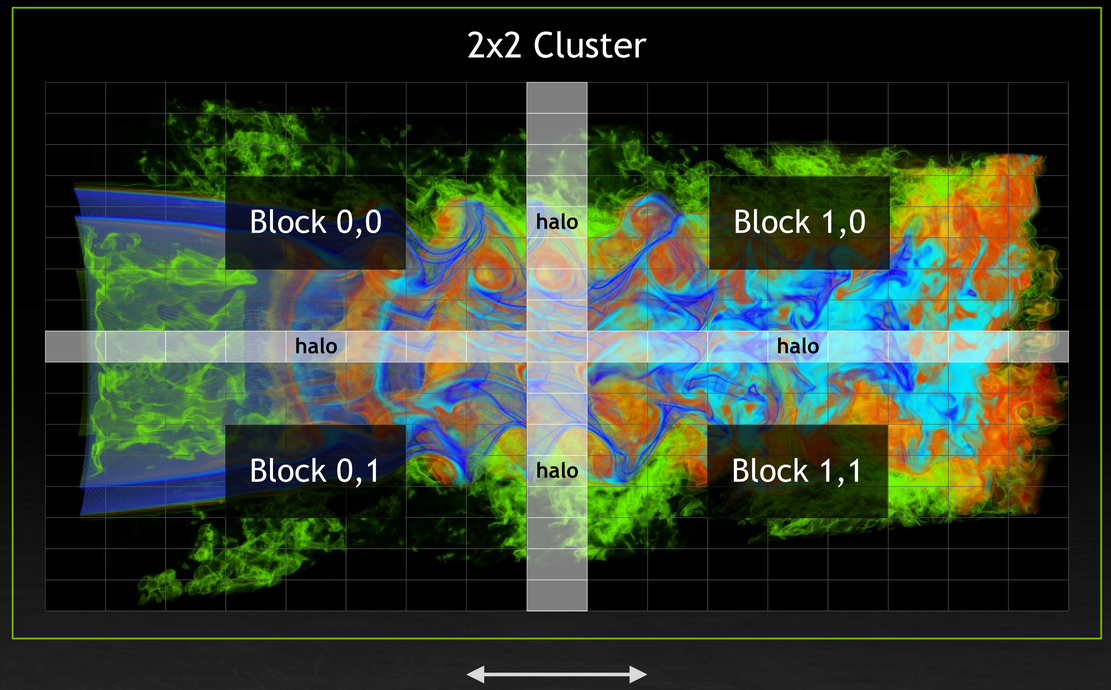
- Global ←→ Shared Memory
- DSMEM ←→ DSMEM within cluster

Uses Asynchronous Transaction Barrier to track completion

Low-Latency One-Sided Data Transfer

Supports one-sided bulk copying within cluster

Supports element-wise reductions when copying to global memory



Self-synchronizing transactions enable 7x faster halo exchange between blocks in the same cluster

TENSOR MEMORY ACCELERATOR UNIT (TMA) FOR ASYNC DATA MOVEMENT

Tensor Memory Accelerator Unit

Hardware-accelerated bi-directional bulk copy

- Global ←→ Shared Memory
- DSMEM ←→ DSMEM within cluster

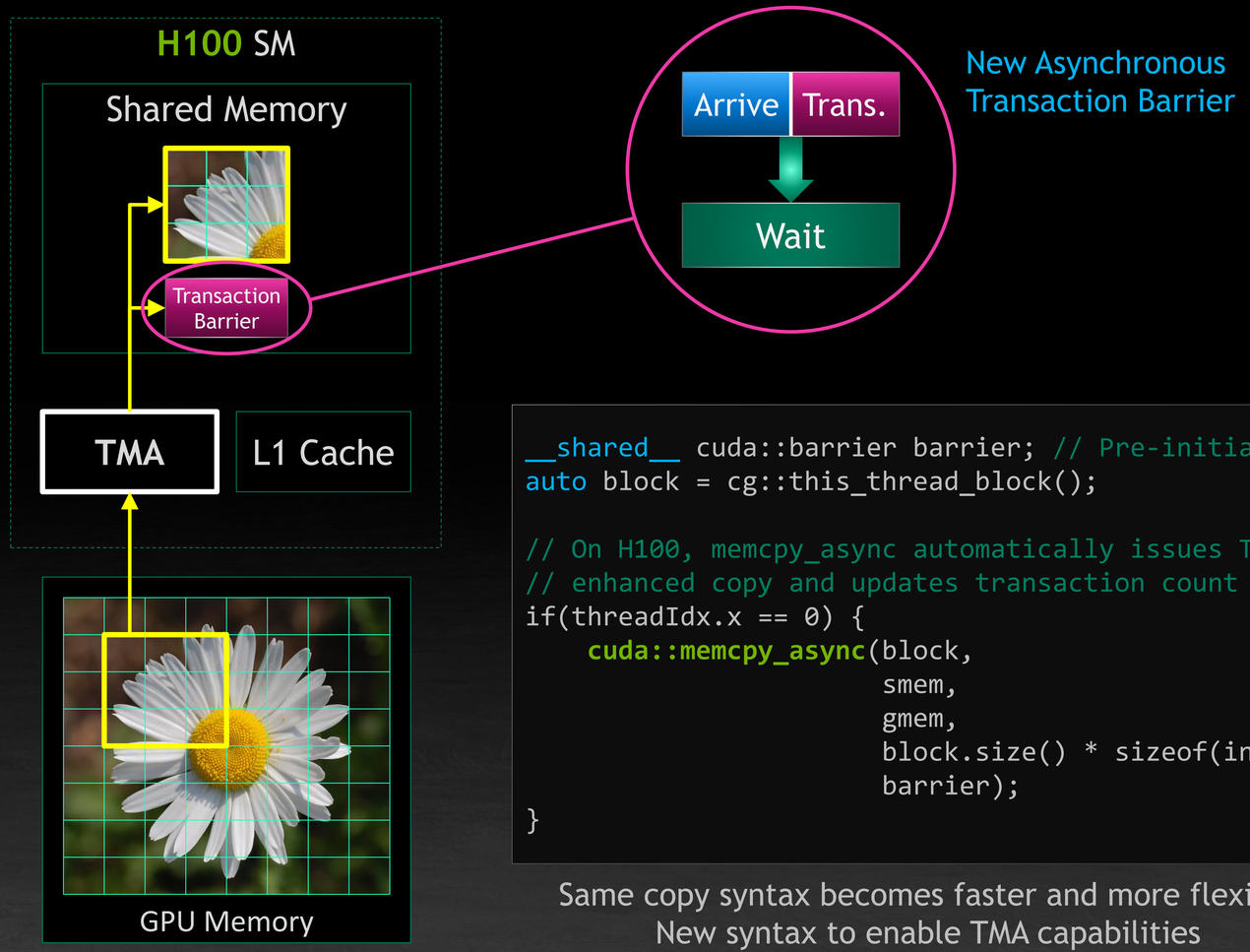
Uses Asynchronous Transaction Barrier to track completion

Faster And More Flexible Async Copying

Drop-in enhancement to existing cuda::memcpy_async() function

Single thread triggers any-size copy – no need for looping or collective copying

Multiple copy operations can contribute to a single cluster-wide async transaction barrier




```
__shared__ cuda::barrier barrier; // Pre-initialised
auto block = cg::this_thread_block();

// On H100, memcpy_async automatically issues TMA
// enhanced copy and updates transaction count
if(threadIdx.x == 0) {
    cuda::memcpy_async(block,
                       smem,
                       gmem,
                       block.size() * sizeof(int),
                       barrier);
}
```

Same copy syntax becomes faster and more flexible
New syntax to enable TMA capabilities

HARDWARE ACCELERATED 1D-5D TENSOR MEMORY COPY

Tensor Memory Accelerator Unit

Hardware-accelerated bi-directional bulk copy

- Global ←→ Shared Memory
- DSMEM ←→ DSMEM within cluster

Uses Asynchronous Transaction Barrier to track completion

Multi-Dimensional Tensor Copying

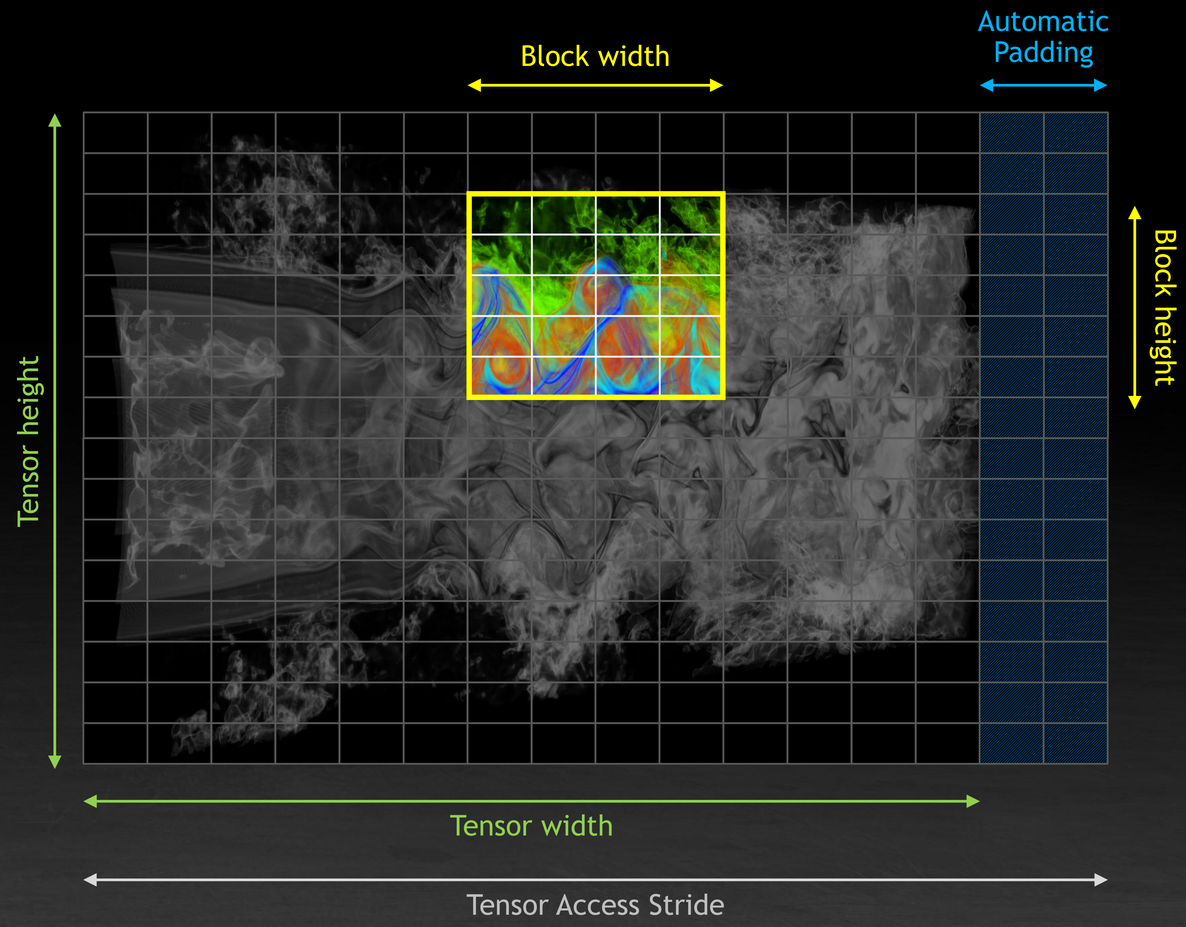
Automatic stride & address generation up to tensors of rank 5

Boundary padding for out-of-bounds accesses

Fire-and-forget from a single thread – everything handled by TMA

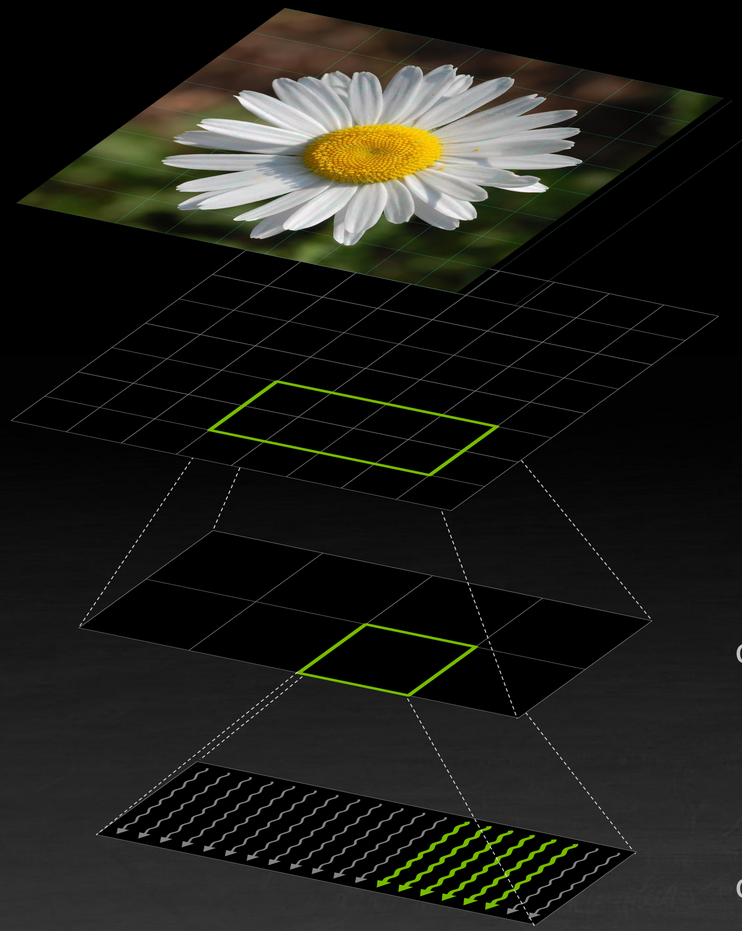
No iteration or bounds-checking code required

The TMA can copy sub-regions of a multi-dimensional tensor



COOPERATIVE GROUPS: PROGRAMMING TO THE NATURAL EXECUTION HIERARCHY

Cooperative Groups is an **explicit** model for expressing cooperation & synchronization among threads



Grid of work

```
cooperative_groups::grid_group grid = this_grid();
```

Cluster of Blocks

```
cooperative_groups::cluster_group cluster = this_cluster();
```

Block of Threads

```
cooperative_groups::thread_block block = this_thread_block();
```

Warp of Threads

```
cooperative_groups::thread_block_tile<32> warp = tiled_partition<32>(block);
```

CUDA C Programming Guide: Cooperative Groups

COLLECTIVE OPERATIONS AT EVERY LEVEL OF HIERARCHY

Synchronization

Grid must be launched cooperatively in order to use grid-scope collectives

```
cooperative_groups::grid_group grid = this_grid();
```

```
cooperative_groups::cluster_group cluster = this_cluster();
```

Synchronization

Reduction

Prefix-sum

```
cooperative_groups::thread_block block = this_thread_block();
```

Warp & smaller groups can use: shfl, any, all, ballot, match

```
cooperative_groups::thread_block_tile<32> warp = tiled_partition<32>(block);
```

COLLECTIVE OPERATIONS INCLUDING FOR MULTI-WARP GROUPS

Supported `thread_block_tile` sizes of 64, 128, 256, 512

Synchronization

```
cooperative_groups::grid_group grid = this_grid();
```

```
cooperative_groups::cluster_group cluster = this_cluster();
```

Synchronization

Reduction

Prefix-sum

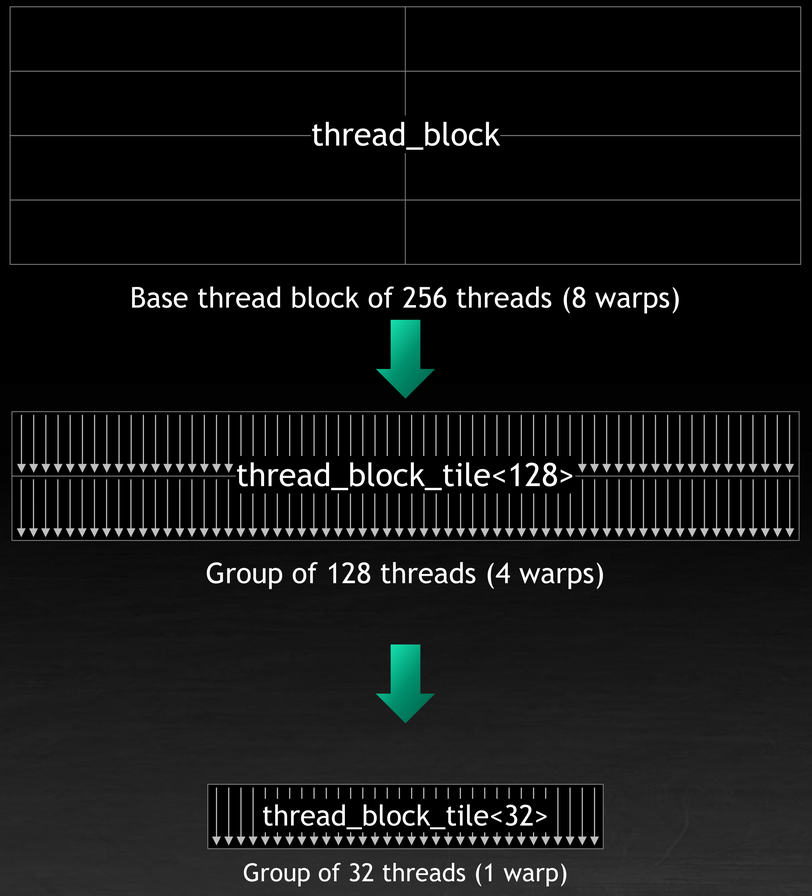
```
cooperative_groups::thread_block block = this_thread_block();
```

```
cooperative_groups::thread_block_tile<128> quadwarp = tiled_partition<128>(block);
```

```
cooperative_groups::thread_block_tile<32> warp = tiled_partition<32>(quadwarp);
```

TYPE SAFETY ENABLES COMPOSABLE LIBRARIES OF PARALLEL FUNCTIONS

Be explicit about how many threads a function wants – e.g. Radix-128 FFT can enforce 128 threads

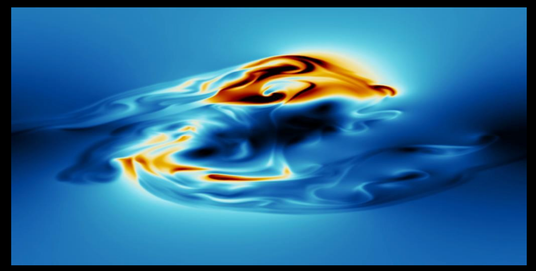



```
using namespace cooperative_groups;
__global__ void kernel(float *in, float *out, ...) {
    // Subdivide my block into 128-thread tiles for the FFT call
    thread_block_tile<128> tile128 = tiled_partition<128>(this_thread_block());
    fft128(tile128, in, out, ...);
}
```

```
// A radix-128 FFT function which *must* be called with 128 threads
__device__ void fft128(const thread_block_tile<128> &group,
                       float *in, float *out, ...) {
    do_fft(in, out, ...);
    group.sync();       // Sync all 128 threads in the group
}
```

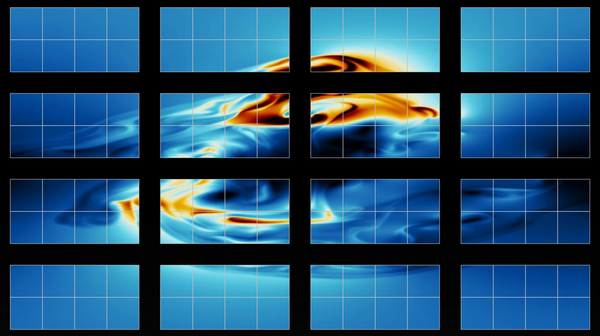
```
// Functions can require an explicit thread count via the type system
__device__ int sum(const thread_block_tile<32> &group,
                       float *in, float *out, ...) {
    // Collective operations are common to all block and sub-block groups
    return group.reduce(in[group.thread_rank()]);
}
```

EXAMPLE: HIERARCHICAL EXECUTION, DATA EXCHANGE AND SYNCHRONIZATION



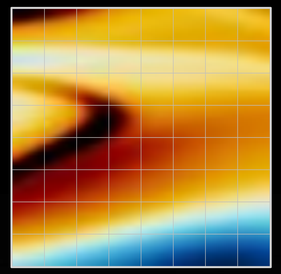
1

Launch cooperative grid of 4x4 clusters, each with 4x2 blocks



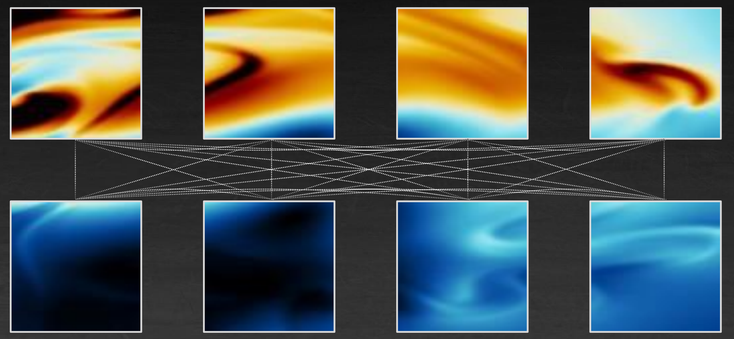
2

Use cooperative cluster indexing to pull data from neighbours



3

Solve locally within each block



4

Synchronize with cluster and solve at cluster level

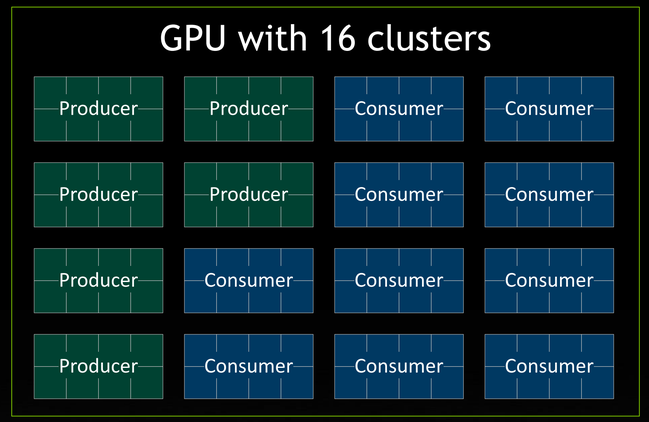
5

Synchronize with grid and solve at grid level; go to 2

Use distributed shared memory for fast inter-block communication.

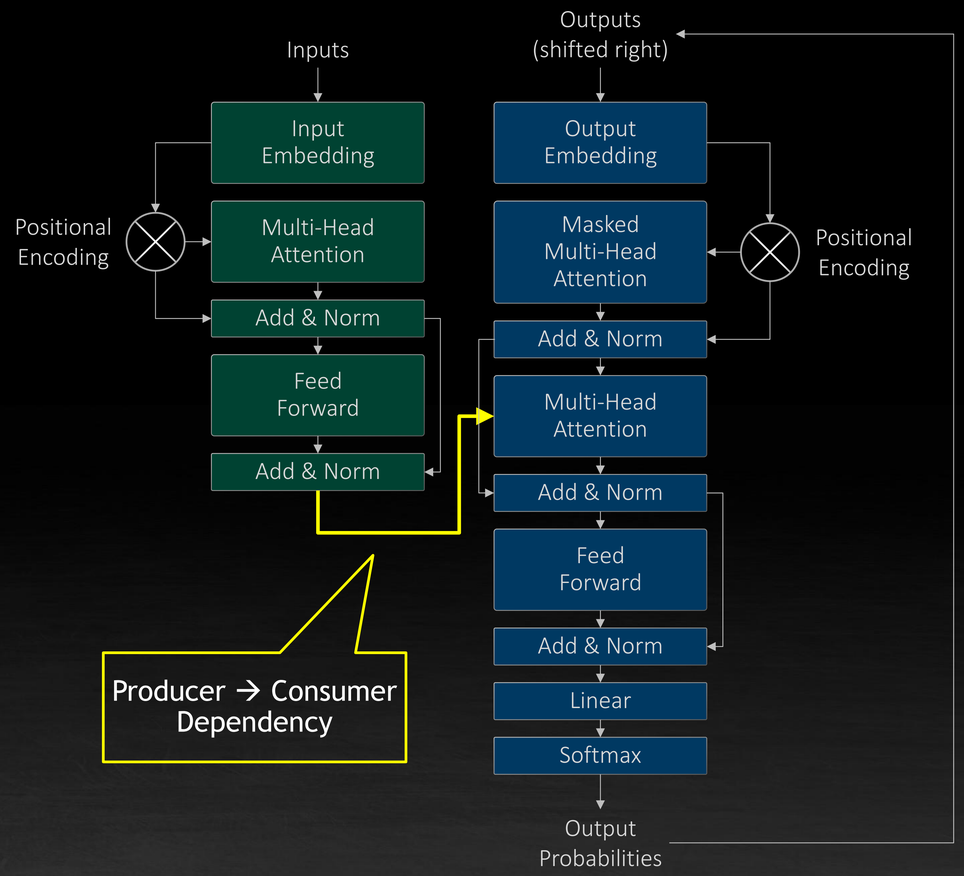
EXAMPLE: PRODUCER/CONSUMER TASK PARALLELISM AT ANY SCALE

Producer/Consumer can operate at every scale – grid, cluster or block



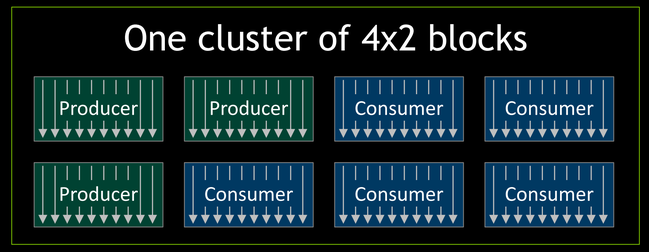
Specialise **clusters** across **whole GPU** into "producer" and "consumer" roles

Specialise	Into	Exchange data via	Synchronize using
Grid	Clusters	Global memory	this_grid().sync()



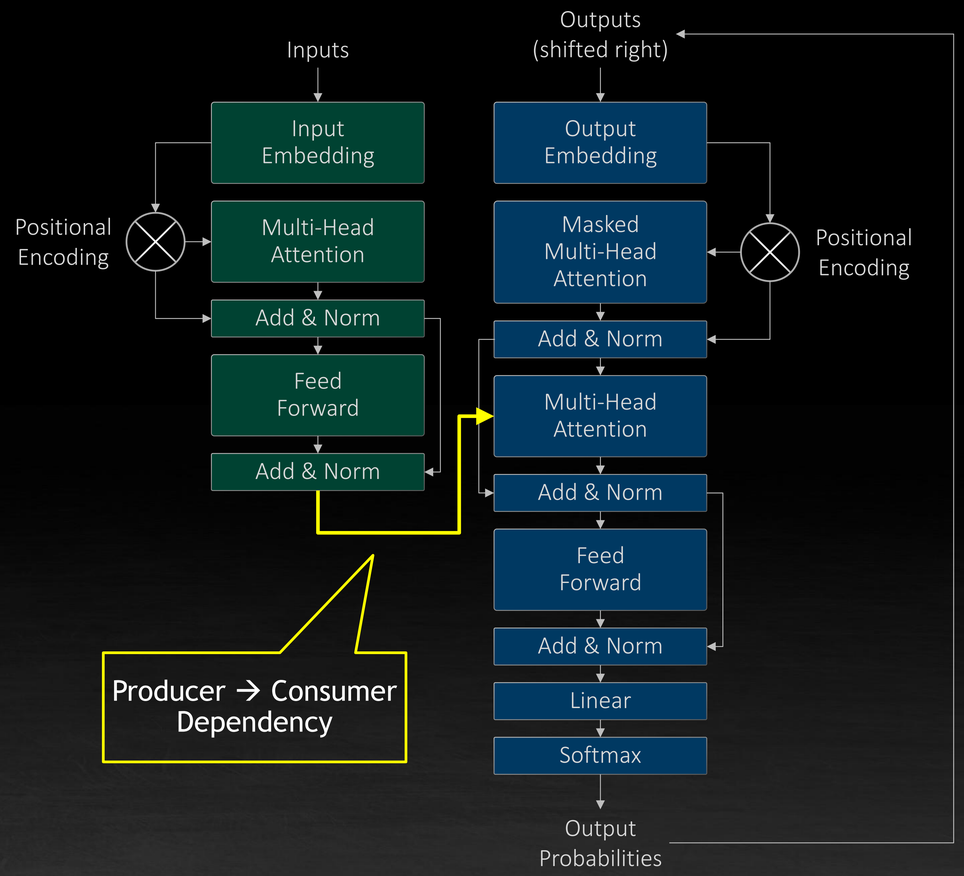
EXAMPLE: PRODUCER/CONSUMER TASK PARALLELISM AT ANY SCALE

Producer/Consumer can operate at every scale – grid, cluster or block



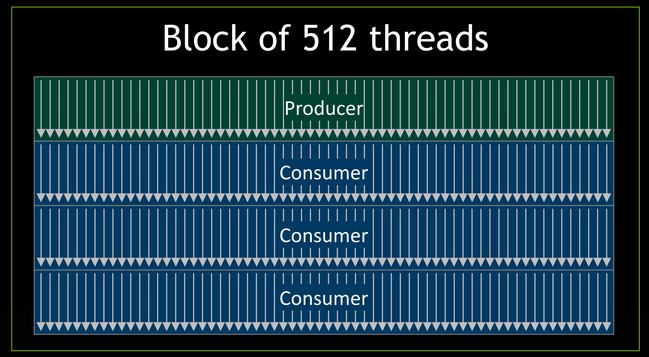
Specialise **blocks** within each **cluster** into "producer" and "consumer" roles

Specialise	Into	Exchange data via	Synchronize using
Grid	Clusters	Global memory	this_grid().sync()
Cluster	Blocks	Distributed-shared or global	this_cluster().sync()



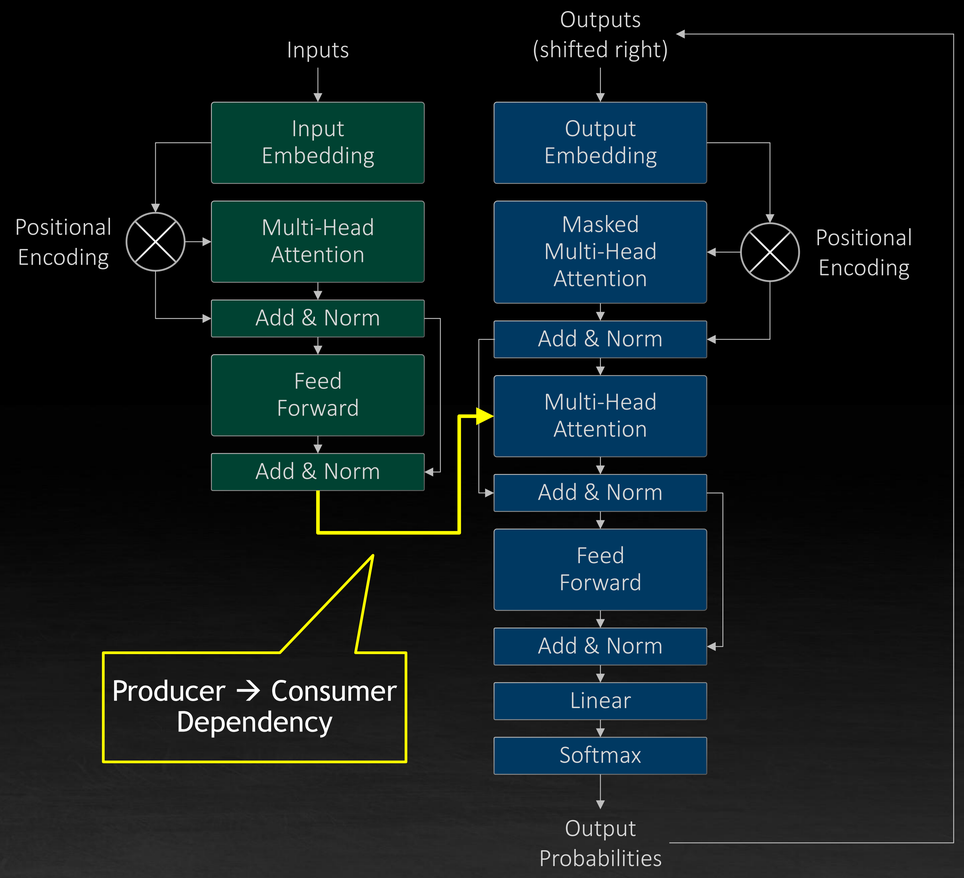
EXAMPLE: PRODUCER/CONSUMER TASK PARALLELISM AT ANY SCALE

Producer/Consumer can operate at every scale – grid, cluster or block



Specialise **groups of warps** within each **block** into “producer” and “consumer” roles

Specialise	Into	Exchange data via	Synchronize using
Grid	Clusters	Global memory	this_grid().sync()
Cluster	Blocks	Distributed-shared or global	this_cluster().sync()
Block	Warps	Shared or global memory	this_thread_block().sync()



EXAMPLE: LONGSTAFF SCHWARTZ PRICING MODEL

Producer/Consumer task parallelism

What is Longstaff Schwartz?

Pricing technique for Quantitative Finance

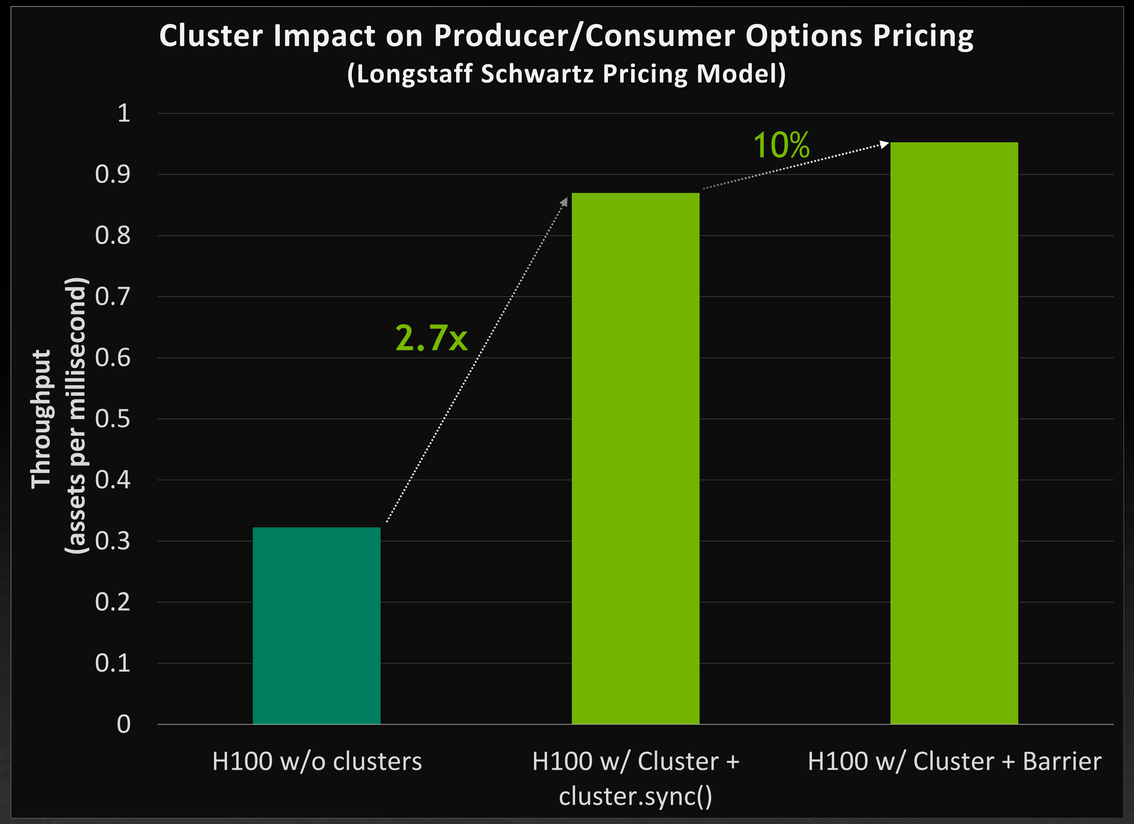
Monte-Carlo simulation with backward iteration to determine financial option values

Analysis of Example

Larger capacity of **Distributed shared memory** enables new approach to programming efficient kernels

Thread Block level **Producer Consumer models can be now implemented in shared memory** with fast synchronization

Hardware accelerated cluster synchronization is easy to use and efficient



Optimizing CUDA Applications for NVIDIA Hopper Architecture [S41489]

CUDA C++ SUPPORT FOR 128-BIT INTEGERS

```
int main() {
  __int128 a, b, c;

  // Init “a” as product of two 64-bit numbers
  a = 197746253418ULL;
  a *= 4060883912384ULL;

  // Perform an arithmetic operation with “a”
  b = a + 10;

  // Launch a kernel, passing in 128-bit int*’s
  cudaMemcpy(device_a, &a, sizeof(a));
  add<<< ... >>>(device_a, device_c);
  cudaMemcpy(&c, device_c, sizeof(c));

  // Verify same result on GPU and CPU
  assert(b == c);
}
```

CUDA 11.7 supports __int128 across toolchain & libraries

Usable with compatible host compilers (gcc, Clang, ICC) or with run-time compilation (NVRTC)

Full release

Supports arithmetic, logical & bitwise operations, math operations, library support, dev tools, etc.

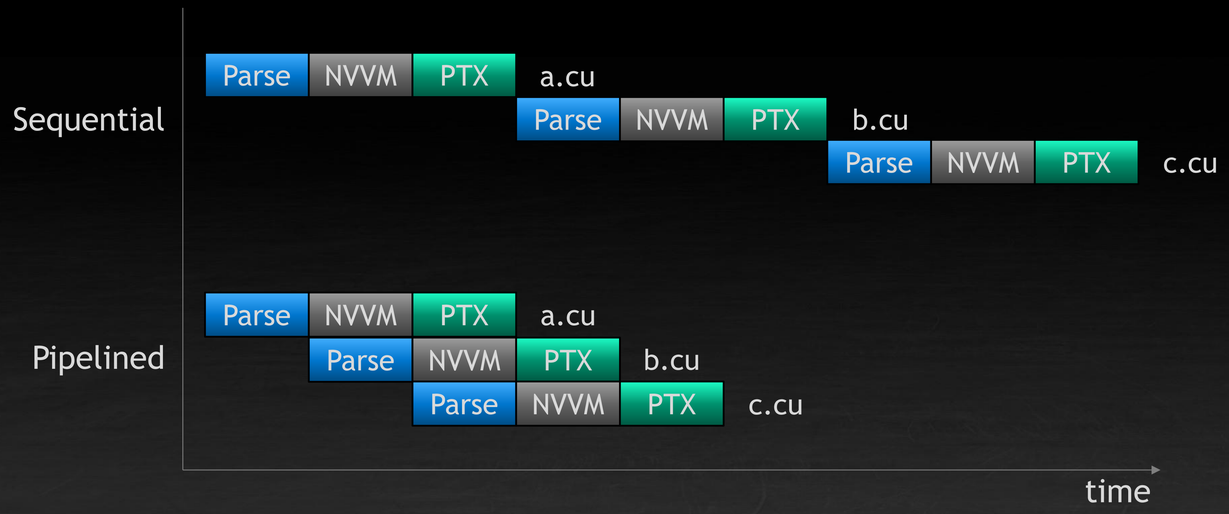
```
__global__ void add (__int128 *in1, __int128 *out) {
  // Perform the same arithmetic operation
  // as we did in host code
  *out = *in1 + 10;
}
```

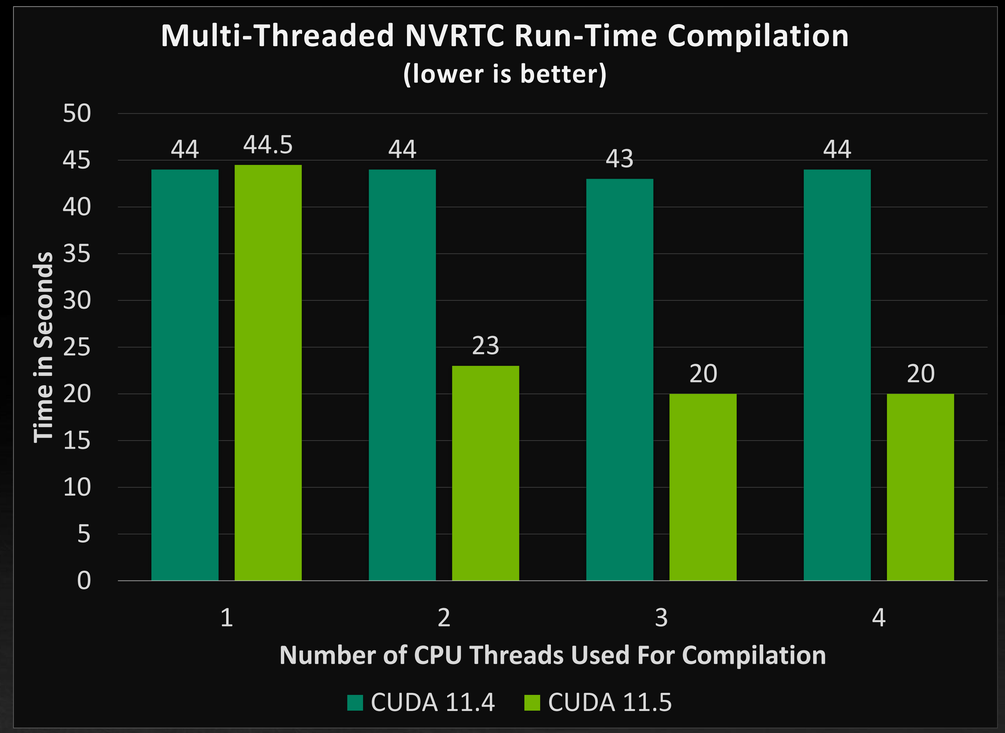
Implementing High-Precision Decimal Arithmetic with CUDA int128 [Blog]

NVRTC MULTI-THREADED COMPILATION

Moving from **global** to **per-stage** locking in NVRTC & PTX JIT

Enables concurrent compilation when compiling with multiple CPU threads





Reducing Application Build Times Using CUDA C++ Compilation Aids [Blog]

RECENT COMPILER UPDATES

C++20 Support in CUDA 11.7

C++20 supported as preview with these host compilers:
GCC 10+, Clang 10+, nvc++ 20.7+

MSVC support coming in next release

Programmatic management of diagnostics

```
#pragma nv_diag_suppress error_number[,error_number...]
#pragma nv_diag_warning  error_number[,error_number...]
#pragma nv_diag_error    error_number[,error_number...]
#pragma nv_diag_default  error_number[,error_number...]
#pragma nv_diag_once     error_number[,error_number...]
#pragma nv_diag_push
#pragma nv_diag_pop
```

Grid-private “Hard” Constants

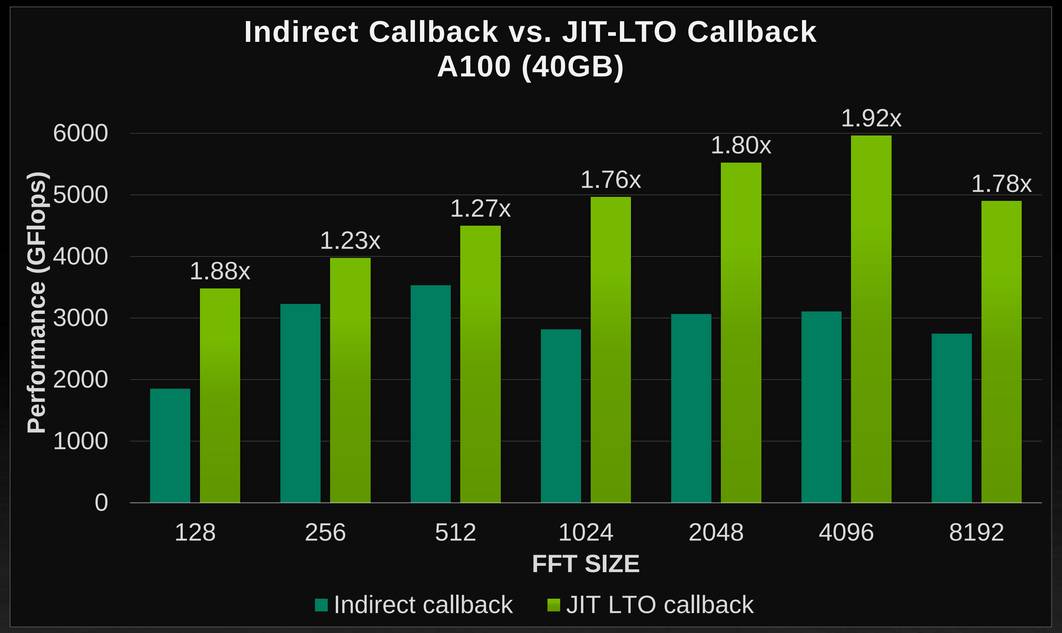
New **__grid_constant__** annotation for kernel parameters allows compiler to avoid creating per-thread copies of read-only parameters

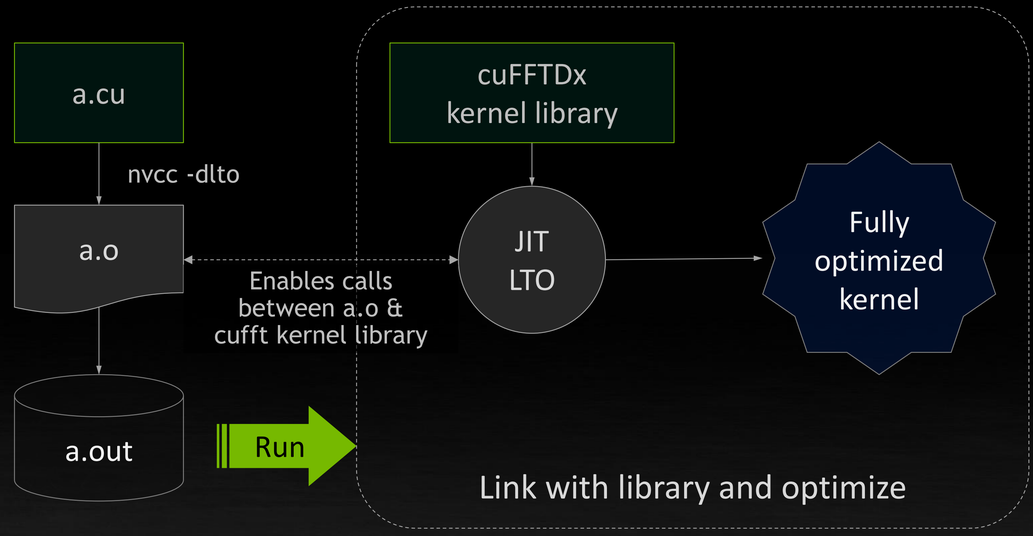
```
__global__ void kernel( __grid_constant__ const struct param_t p ) {
    foo( &p );
}
```

New nvcc target -arch options

-arch=all	generates codes for all supported architectures
-arch=all-major	generates codes for all supported major architectures
-arch=native	generates codes for all visible GPUs on the system

JIT LINKING WITH LINK-TIME OPTIMIZATIONS





JIT-LTO

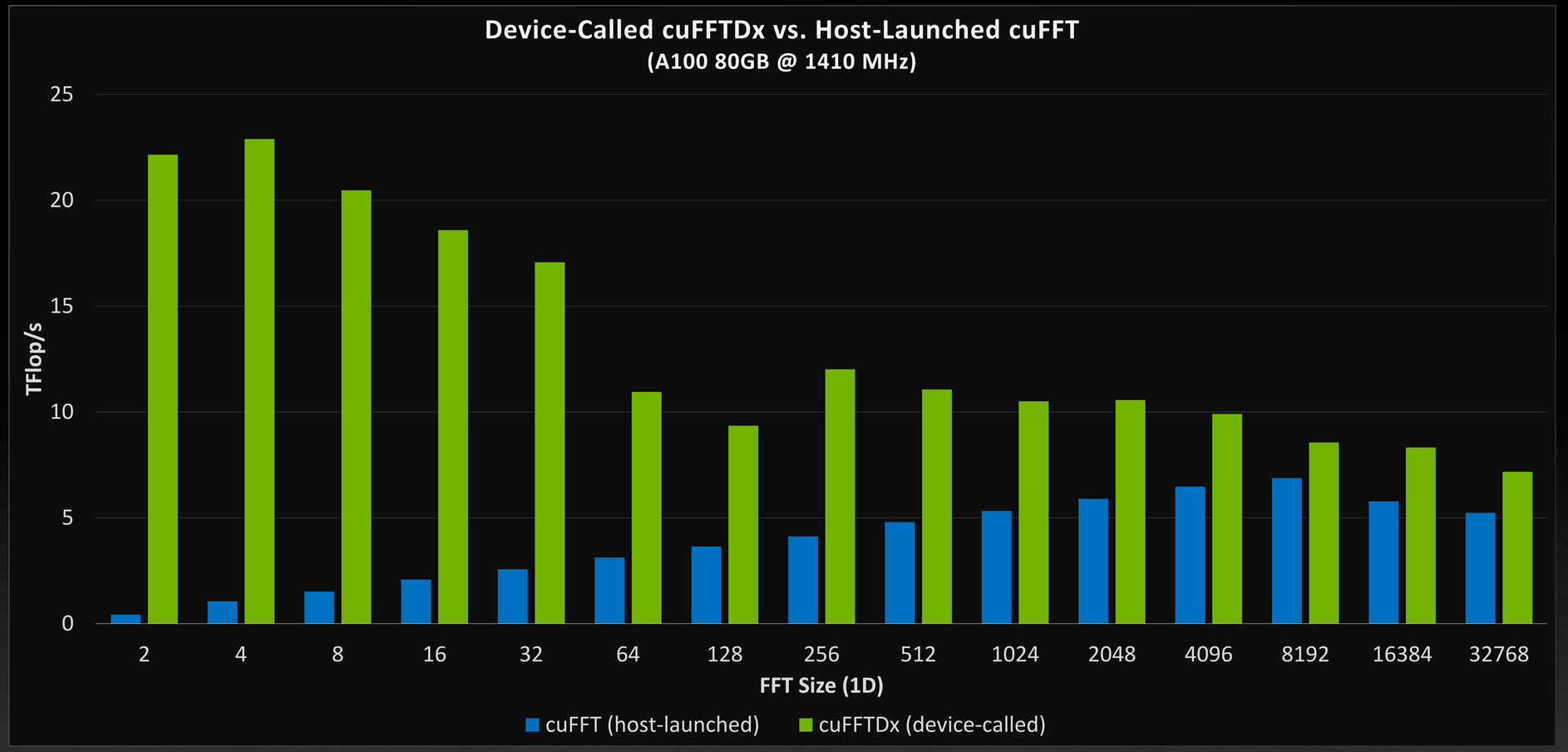
Object files are combined
and optimized at runtime

[JIT LTO Adoption in cuSPARSE/cuFFT: Use Case Overview [A31155, GTC Fall 2021]]

NVIDIA

MATHS LIBRARIES DEVICE EXTENSIONS

Calling cuFFTDx from device code significantly speeds up small problem sizes (blue bars)



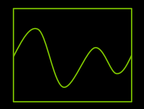
Recent Developments in NVIDIA Math Libraries [S41491]

MATHS LIBRARIES DEVICE EXTENSIONS

Invoke custom kernel code from within high performance Math Libraries (green bars)

Libraries With Device-Callable Interfaces

Available Now



cuFFTDx

Fast Fourier Transforms

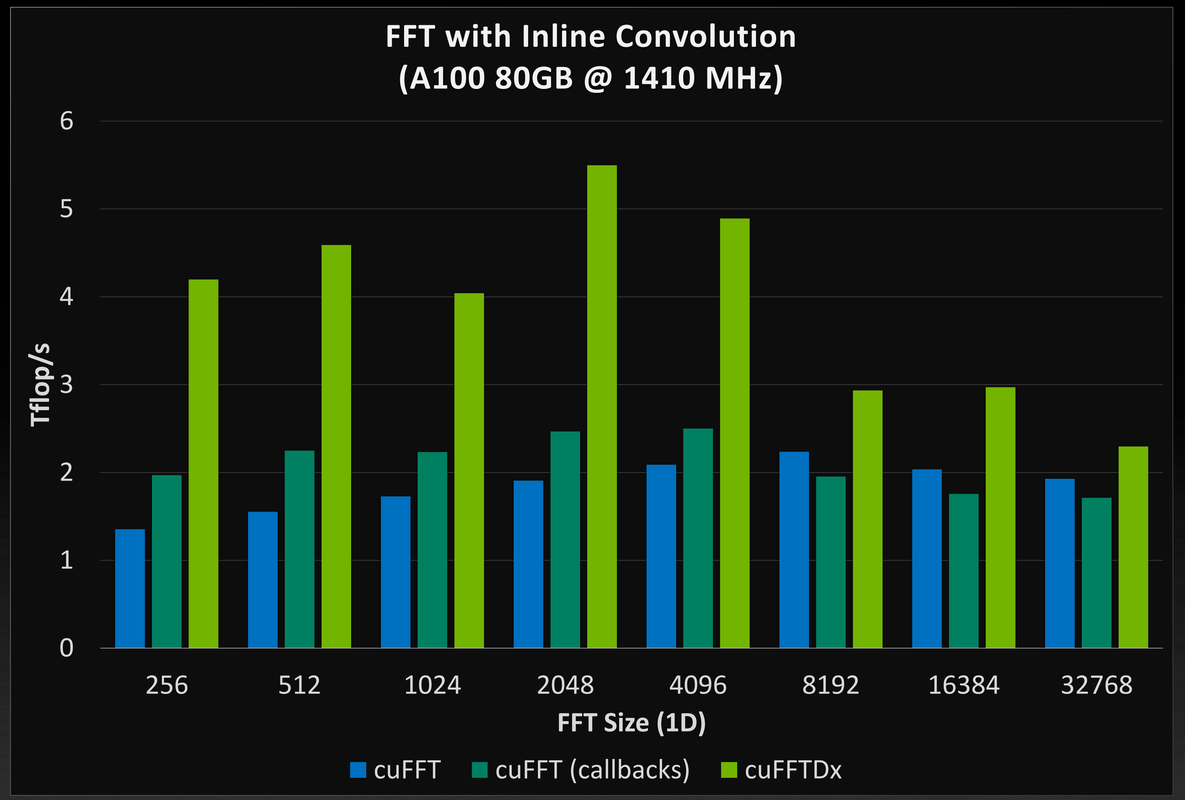
Coming In Future

cuBLASDx

GPU-Accelerated BLAS-1,2,3

cuSOLVERDx

GPU-Accelerated Dense Linear Algebra



Connect with the Experts: NVIDIA Math Libraries [CWE41721]

NVIDIA

CUTLASS

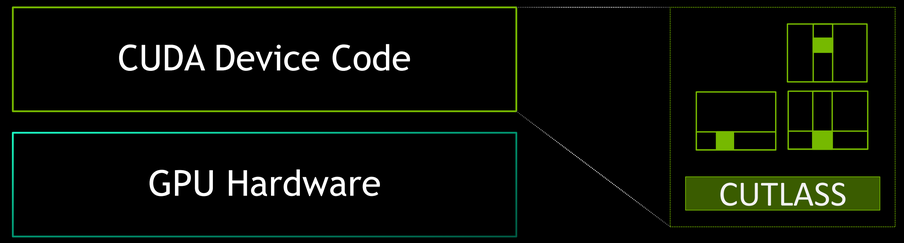
CUDA C++ Template Library for Deep Learning and High Performance Computing

Optimal CUDA C++ matrix operation templates at all scopes and scales

Tensor-core accelerated matrix & tensor operations & more

GEMM, convolution, reduction, fused input & output operations

Open-source header file C++ template library



Cutlass is compiled in to CUDA kernels, interfacing directly with GPU hardware

Tensor-core accelerated operations at all granularities

Device	{ GEMM, Convolution, Reductions } x { all data types } x { SIMT, Tensor Cores } x { all architectures }
Kernel	GEMM, Batched GEMM, Convolution, Reduction, Fused output operations, Fused input operations
Thread Block	Pipelined Matrix Multiply, Epilogue, Collective access to tensors, Convolution matrix access
Warp	Tensor Core Multiply-Add operations, Efficient access to permuted tensor layouts
Thread	Numeric conversion, <functional> operators on arrays, complex<T>, fast math algorithms
Architecture Intrinsic	Templates wrapping architecture-specific PTX instructions (e.g. mma, cp.async, ldmatrix, cvt)

https://github.com/NVIDIA/cutlass

CUTLASS: ACCELERATED SINGLE PRECISION USING TENSOR CORES

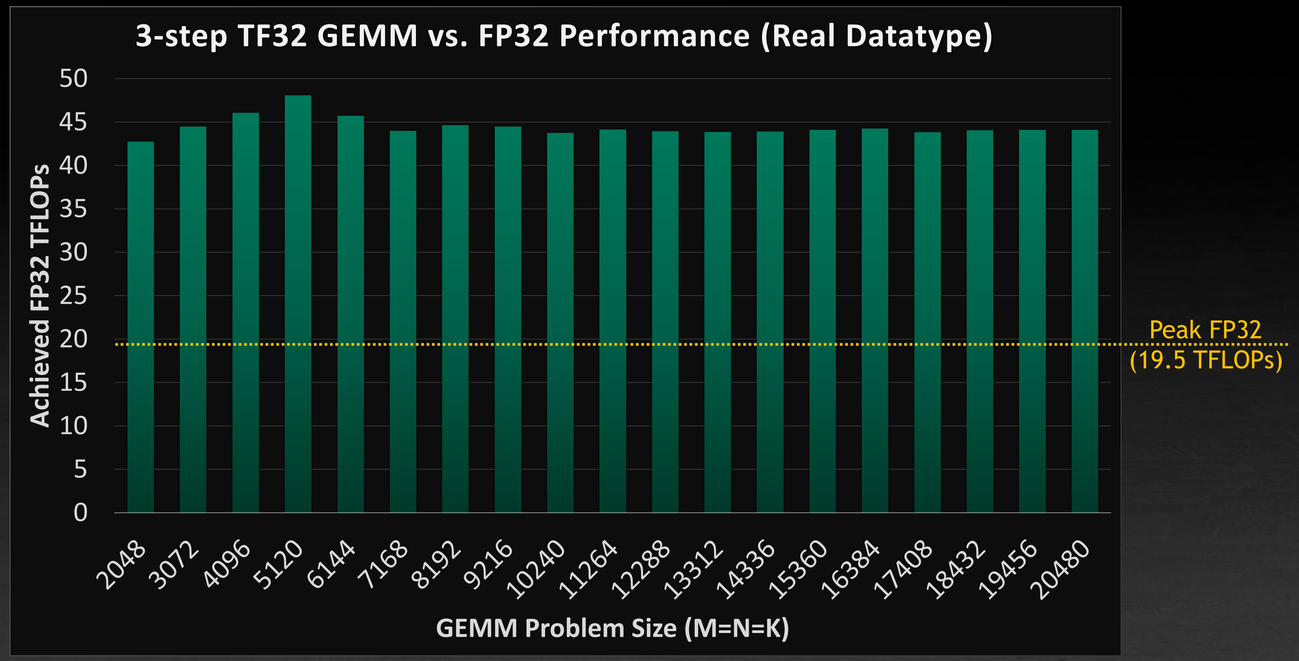
FP32 precision at 48 TFLOPs on A100, using 3-step TF32 sequence

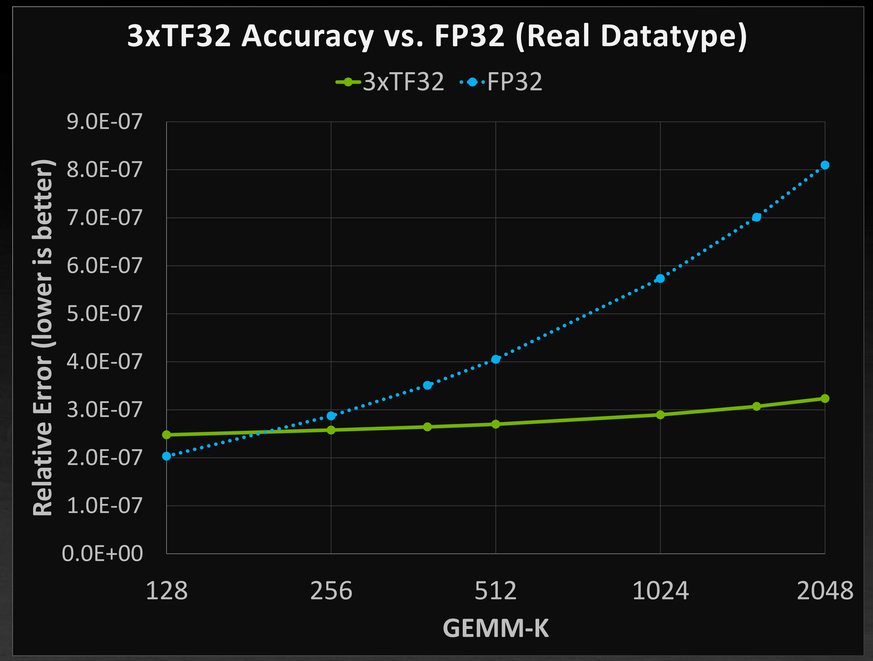
3-step sequence compensates for round-off error, enabling use of TF32 tensor cores

More than 2x performance vs. peak single-precision FFMA

Better accuracy vs. single-precision (not IEEE compliant)

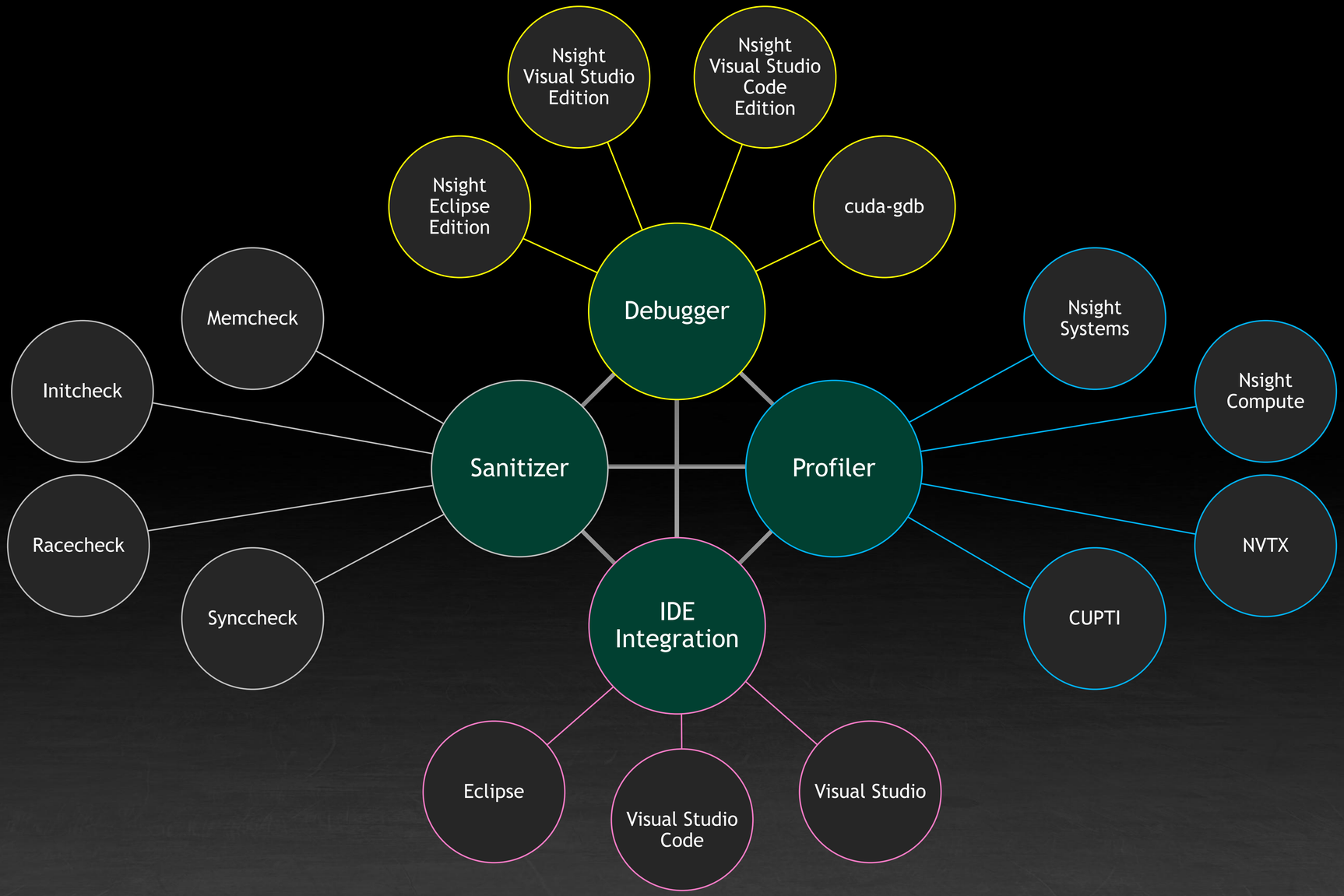
Example implementation for GEMMs and Convolutions





Recent Developments in NVIDIA Math Libraries [S41491]

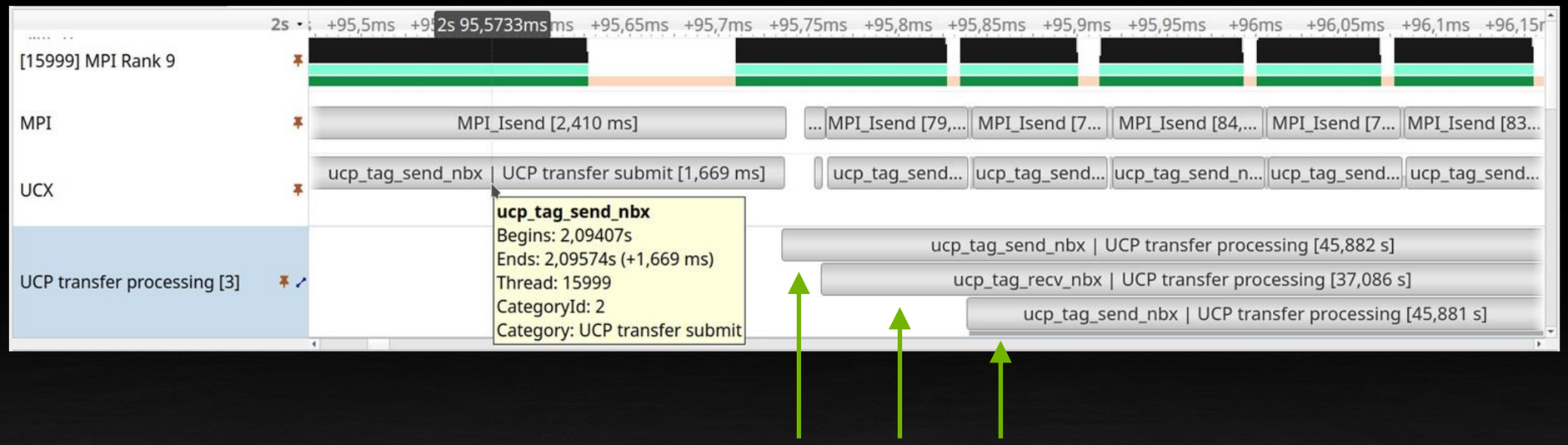
FAMILY OF CUDA DEVELOPER TOOLS



What, Where, and Why? Use CUDA Developer Tools to Detect, Locate, and Explain Bugs and Bottlenecks [S41493]

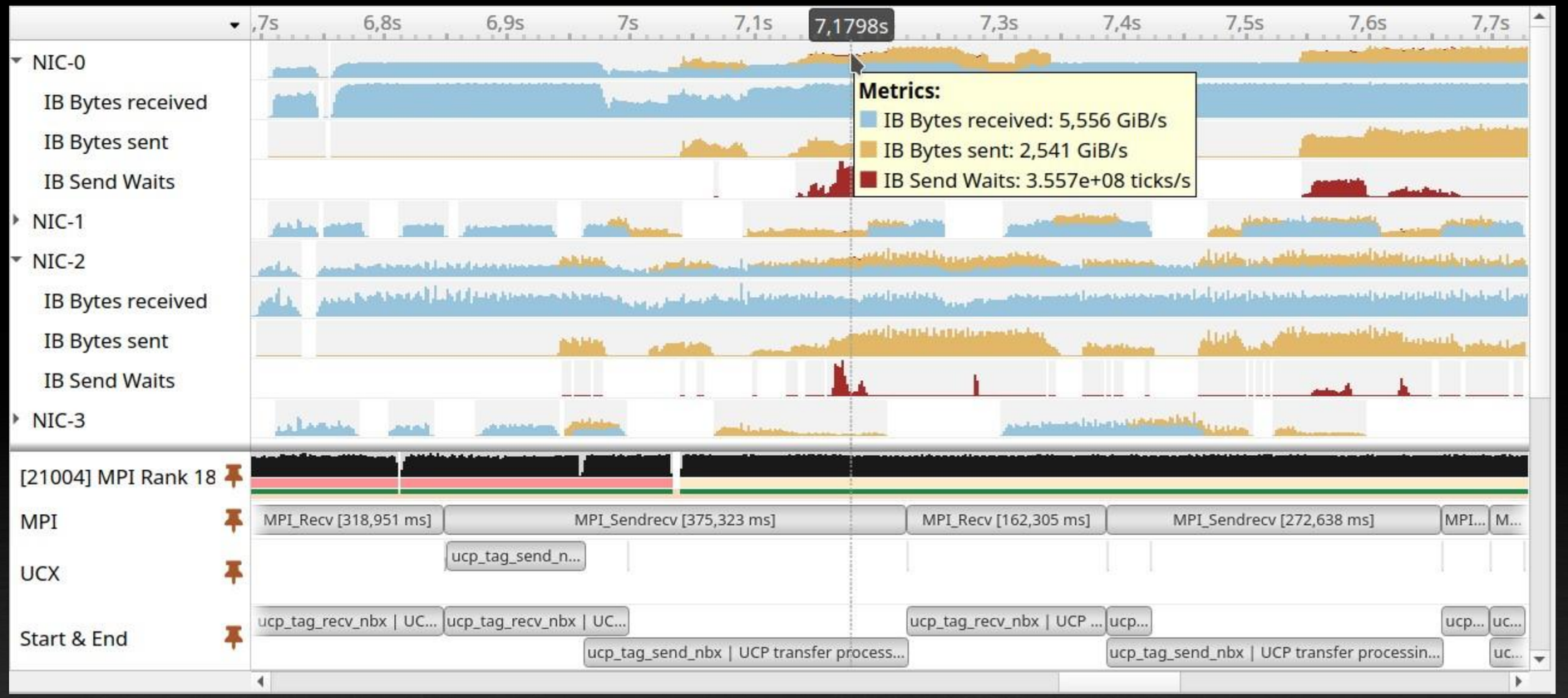
NEW NETWORK PROFILING

Intercept and trace calls into the UCX protocol layer



Completion tracking of non-blocking
UCP communication operations

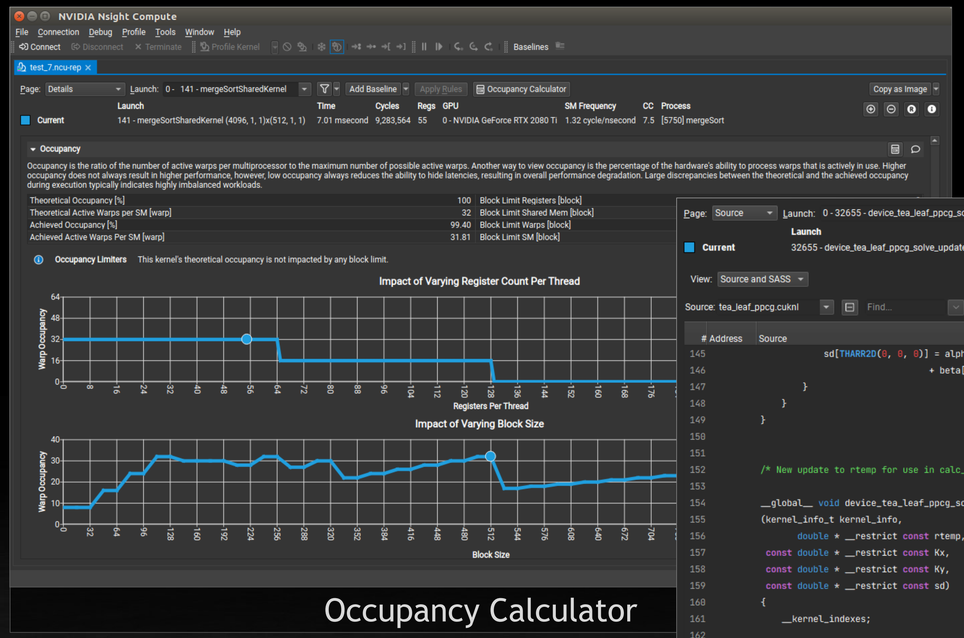
NIC PERFORMANCE METRICS IN NSIGHT SYSTEMS



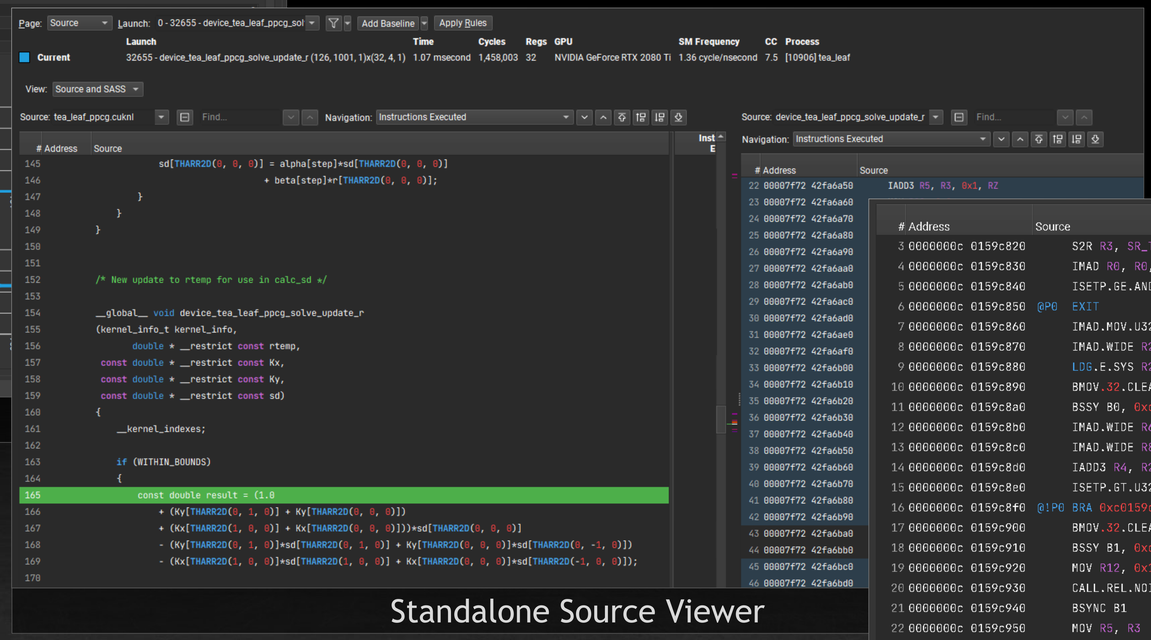

```
$ nsys profile –nic-metrics=[true|false] ...
```

Optimizing Communication with Nsight Systems Network Profiling [S41500]

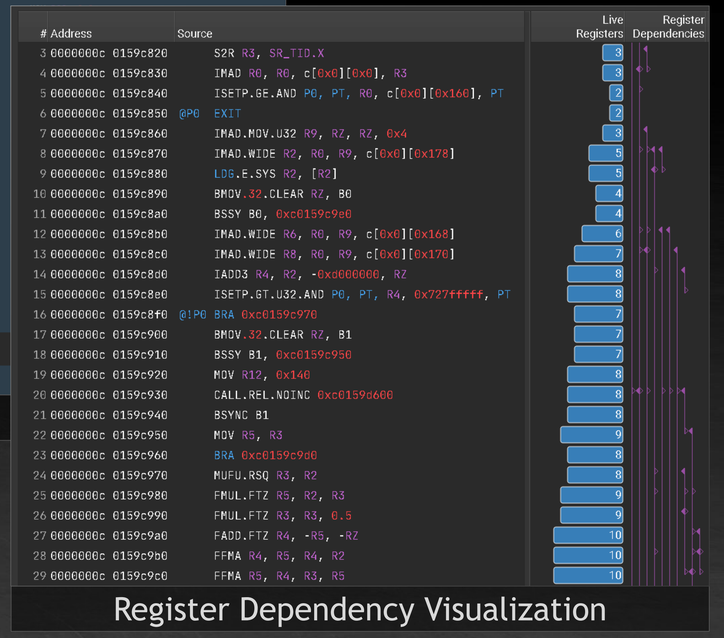
NSIGHT COMPUTE: NEW CUDA PLATFORM DEVELOPER TOOLS



Occupancy Calculator



Standalone Source Viewer



Register Dependency Visualization

[How To Understand and Optimize Shared Memory Accesses using Nsight Compute [S41723]]

NVIDIA

NVTX v3 OVERVIEW

```
#include <nvtx3/nvToolsExt.h>

void LaunchKernel() {
    nvtxRangePush(__FUNCTION__);
    Kernel<<<20000, 256>>>();
    nvtxRangePop();
}

void Example() {
    nvtxRangePush("Creating CUDA context");
    cudaFree(0);
    nvtxRangePop();

    nvtxRangePush("Launching & waiting for kernels");
    for (int i = 0; i < 3; ++i) {
        LaunchKernel();
    }
    cudaDeviceSynchronize();
    nvtxRangePop();
}
```

What is it?

Annotation library inserting markers for tools analysis

Negligible overhead

No dependencies – just header files

How does it work?

Markers record a point of time

Push/pop ranges to record a span of time

Ranges form a stack to show nested detail

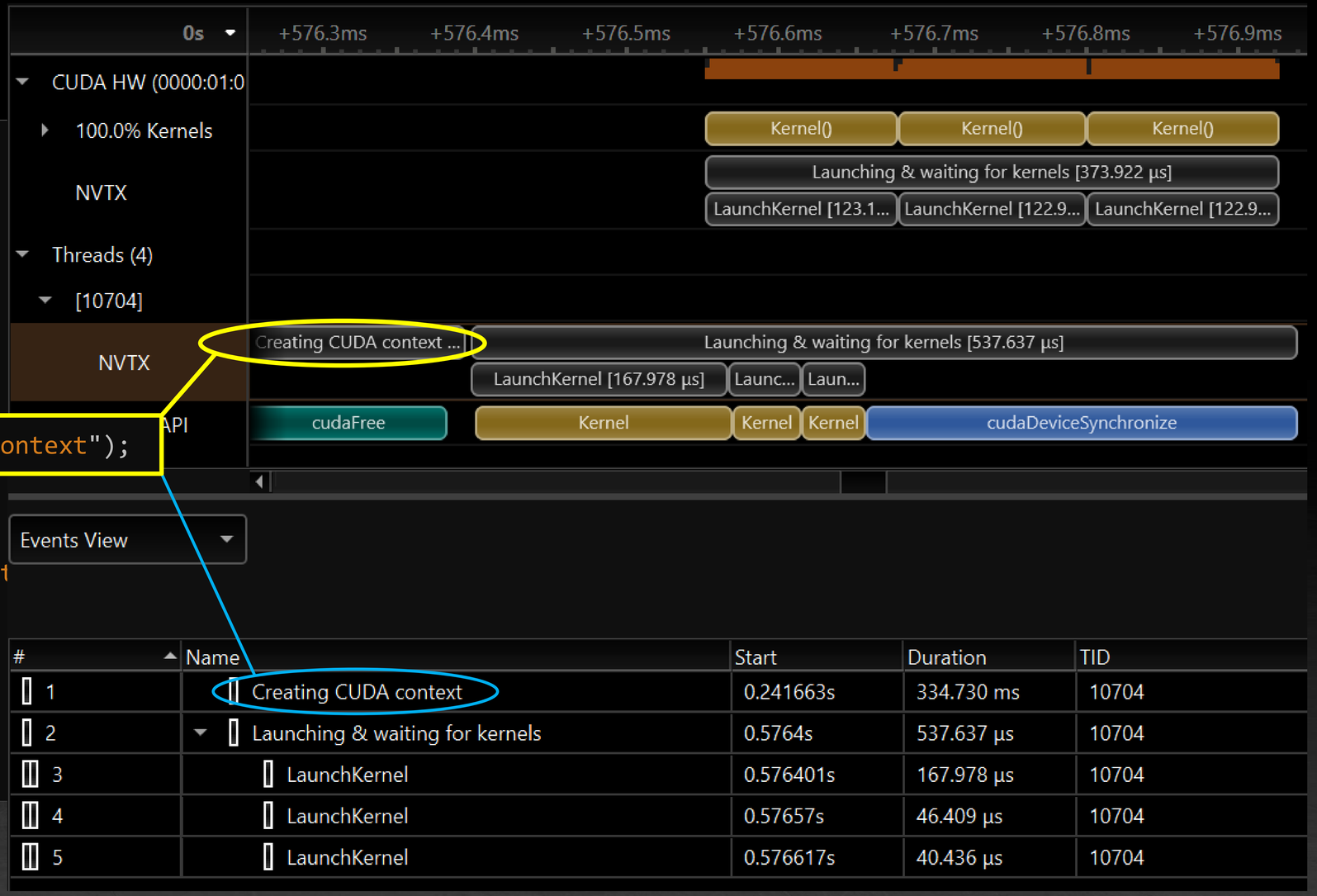
Tools track NVTX calls to display timelines (see next slide)

See link below for details & download

https://github.com/NVIDIA/NVTX

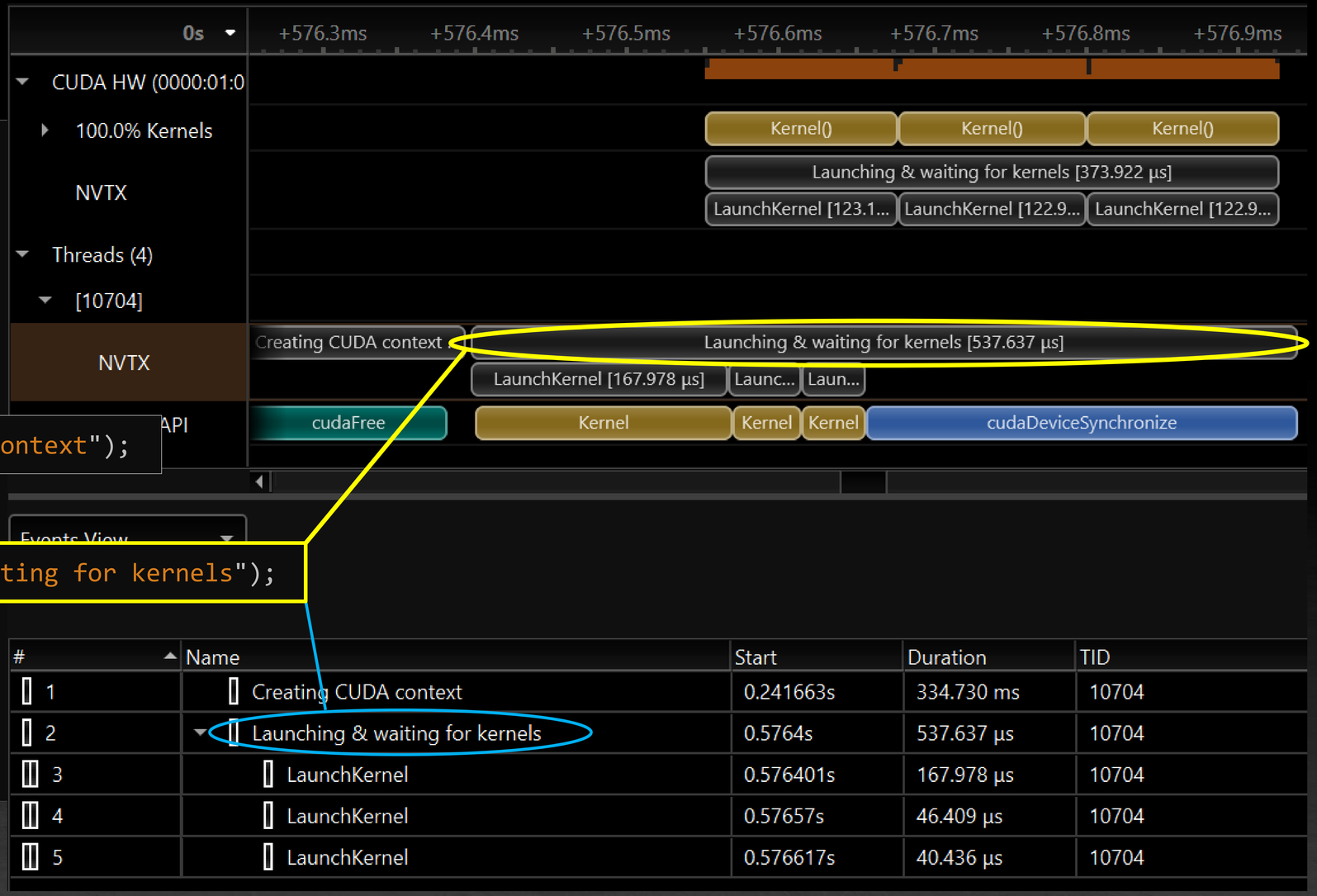
NVTX v3 OVERVIEW

```
#include <nvtx3/nvToolsExt.h>

void LaunchKernel() {
    nvtxRangePush(__FUNCTION__);
    Kernel<<<20000, 256>>>();
    nvtxRangePop();
}

void Example() {
    nvtxRangePush("Creating CUDA context");
    cudaFree(0);
    nvtxRangePop();

    nvtxRangePush("Launching & wait
    for (int i = 0; i < 3; ++i) {
        LaunchKernel();
    }
    cudaDeviceSynchronize();
    nvtxRangePop();
}
```



NVTX v3 OVERVIEW

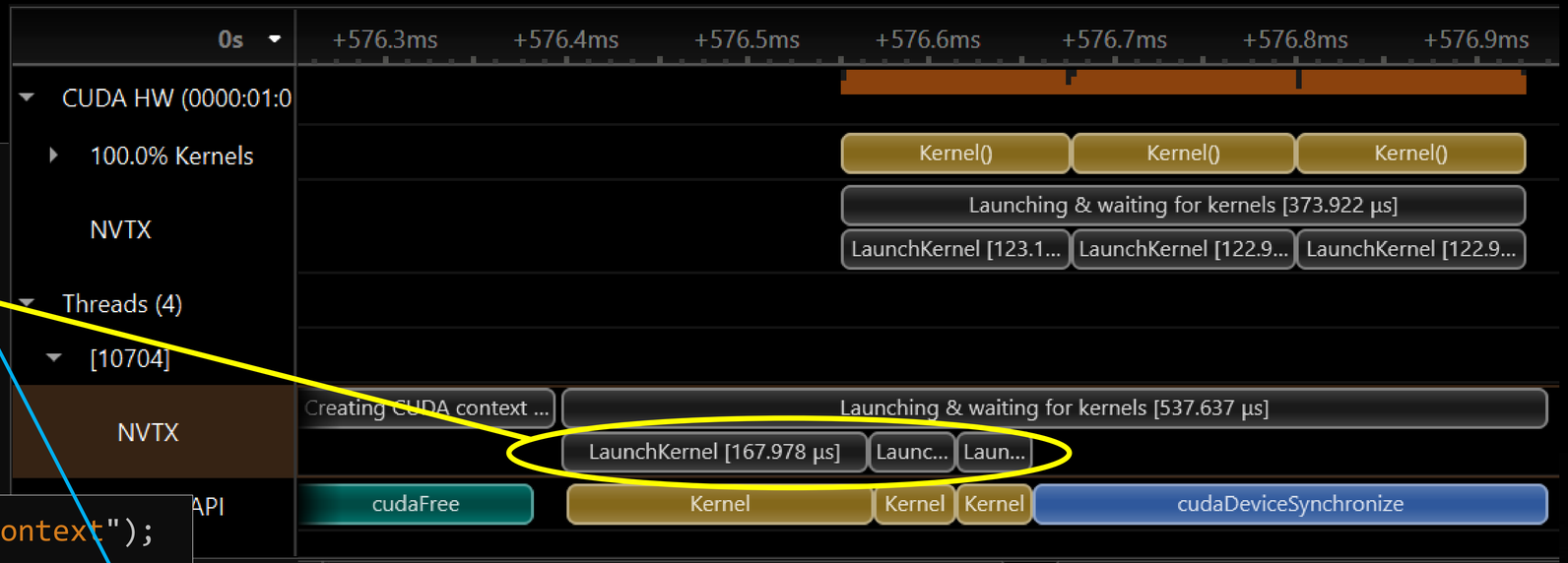
```
#include <nvtx3/nvToolsExt.h>

void LaunchKernel() {
    nvtxRangePush(__FUNCTION__);
    Kernel<<<20000, 256>>>();
    nvtxRangePop();
}

void Example() {
    nvtxRangePush("Creating CUDA context");
    cudaFree(0);
    nvtxRangePop();

    nvtxRangePush("Launching & waiting for kernels");
    for (int i = 0; i < 3; ++i) {
        LaunchKernel();
    }
    cudaDeviceSynchronize();
    nvtxRangePop();
}
```



#	Name	Start	Duration	TID
1	Creating CUDA context	0.241663s	334.730 ms	10704
2	Launching & waiting for kernels	0.5764s	537.637 µs	10704
3	LaunchKernel	0.576401s	167.978 µs	10704
4	LaunchKernel	0.57657s	46.409 µs	10704
5	LaunchKernel	0.576617s	40.436 µs	10704

NVTX v3 OVERVIEW

```
#include <nvtx3/nvToolsExt.h>

void LaunchKernel() {
    nvtxRangePush(__FUNCTION__);
    kernel<<<20000, 256>>>();
    nvtxRangePop();
}

void Example() {
    nvtxRangePush("Creating CUDA context");
    cudaFree(0);
    nvtxRangePop();

    nvtxRangePush("Launching & waiting for kernels");
    for (int i = 0; i < 3; ++i) {
        LaunchKernel();
    }
    cudaDeviceSynchronize();
    nvtxRangePop();
}
```



Events View

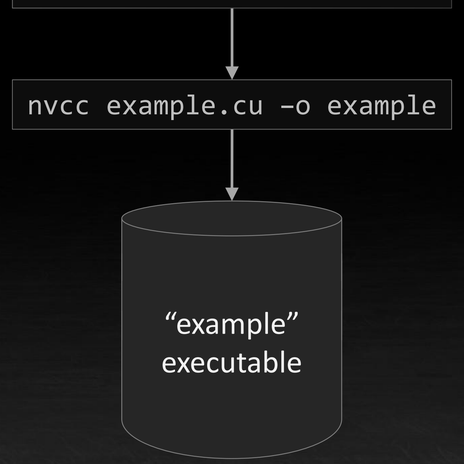
#	Name	Start	Duration	TID
1	Creating CUDA context	0.241663s	334.730 ms	10704
2	Launching & waiting for kernels	0.5764s	537.637 µs	10704
3	LaunchKernel	0.576401s	167.978 µs	10704
4	LaunchKernel	0.57657s	46.409 µs	10704
5	LaunchKernel	0.576617s	40.436 µs	10704

What's in your CUDA toolbox: CUDA Profiling, Optimization, and Debugging Tools [CWE41541]

ANATOMY OF A CUDA BINARY

Hello world example

```
__global__ void kernel() {
    printf("Hello, CUDA\n");
}

void main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

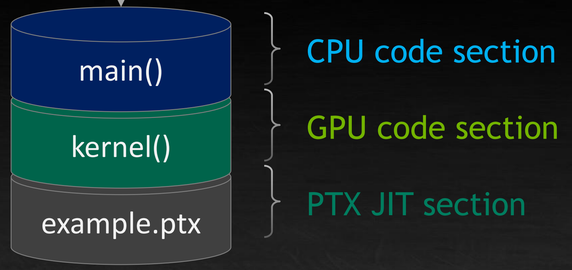


ANATOMY OF A CUDA BINARY

Hello world example

```
__global__ void kernel() {
    printf("Hello, CUDA\n");
}

void main() {
    kernel<<< 1, 1 >>>();
    cudaDeviceSynchronize();
}
```

```
nvcc example.cu –o example
```

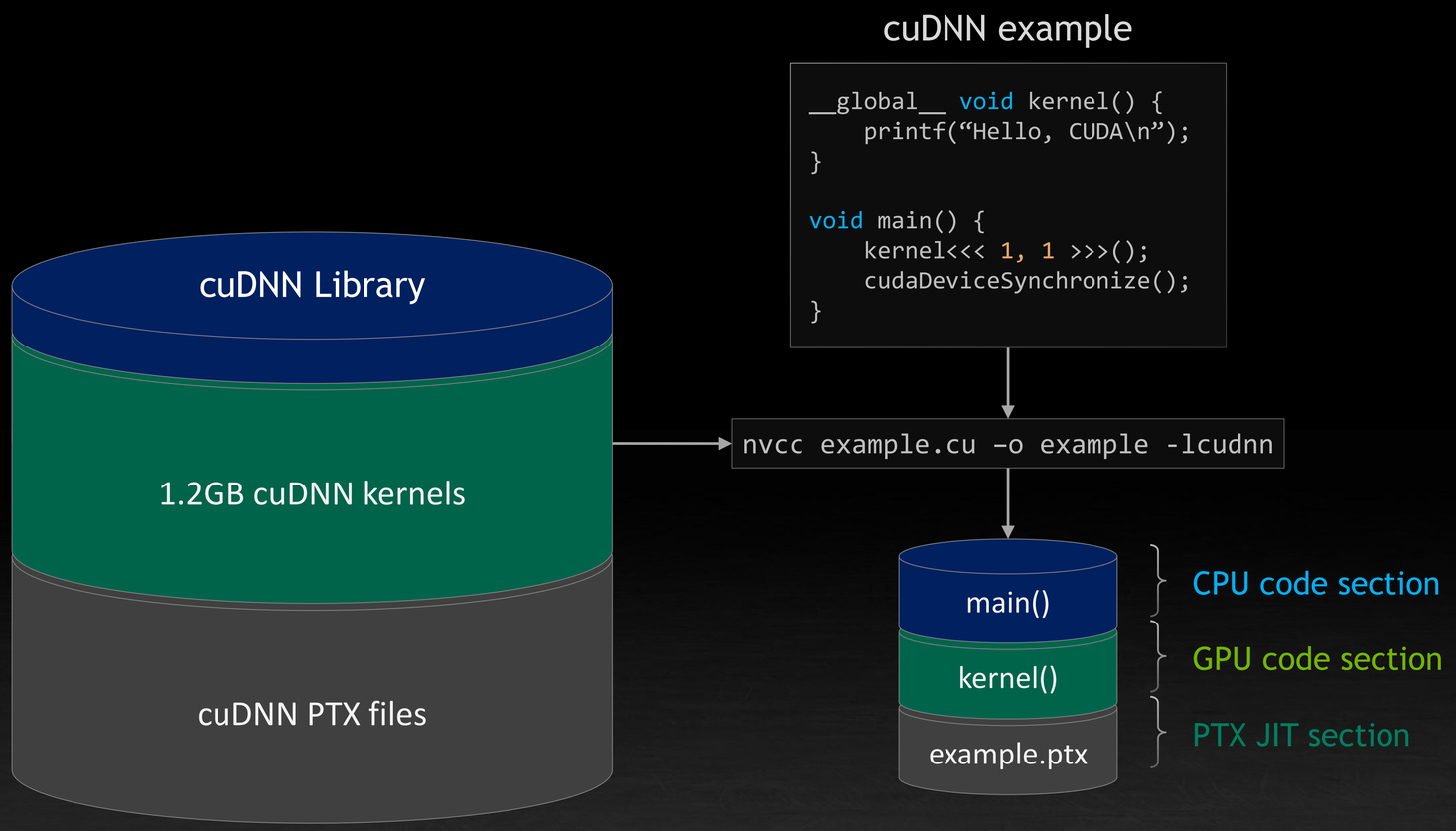


CPU code section

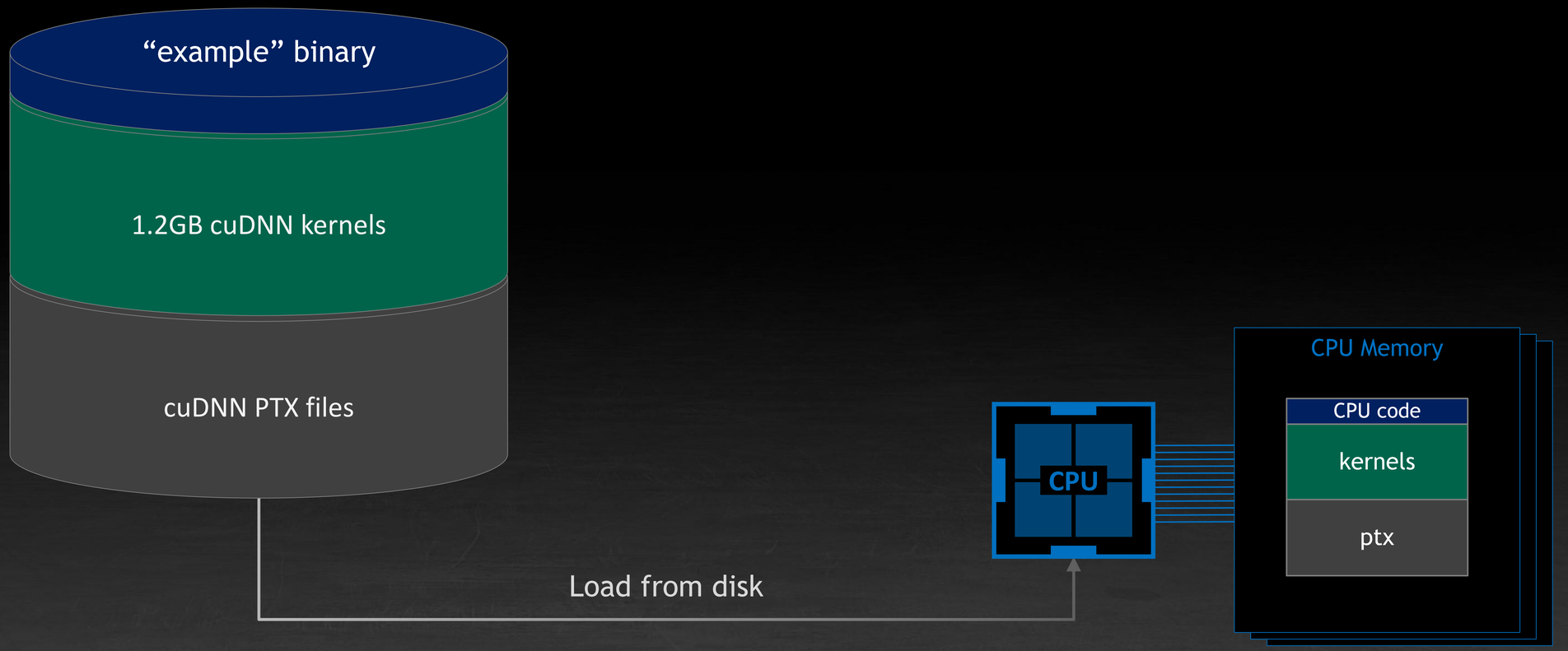
GPU code section

PTX JIT section

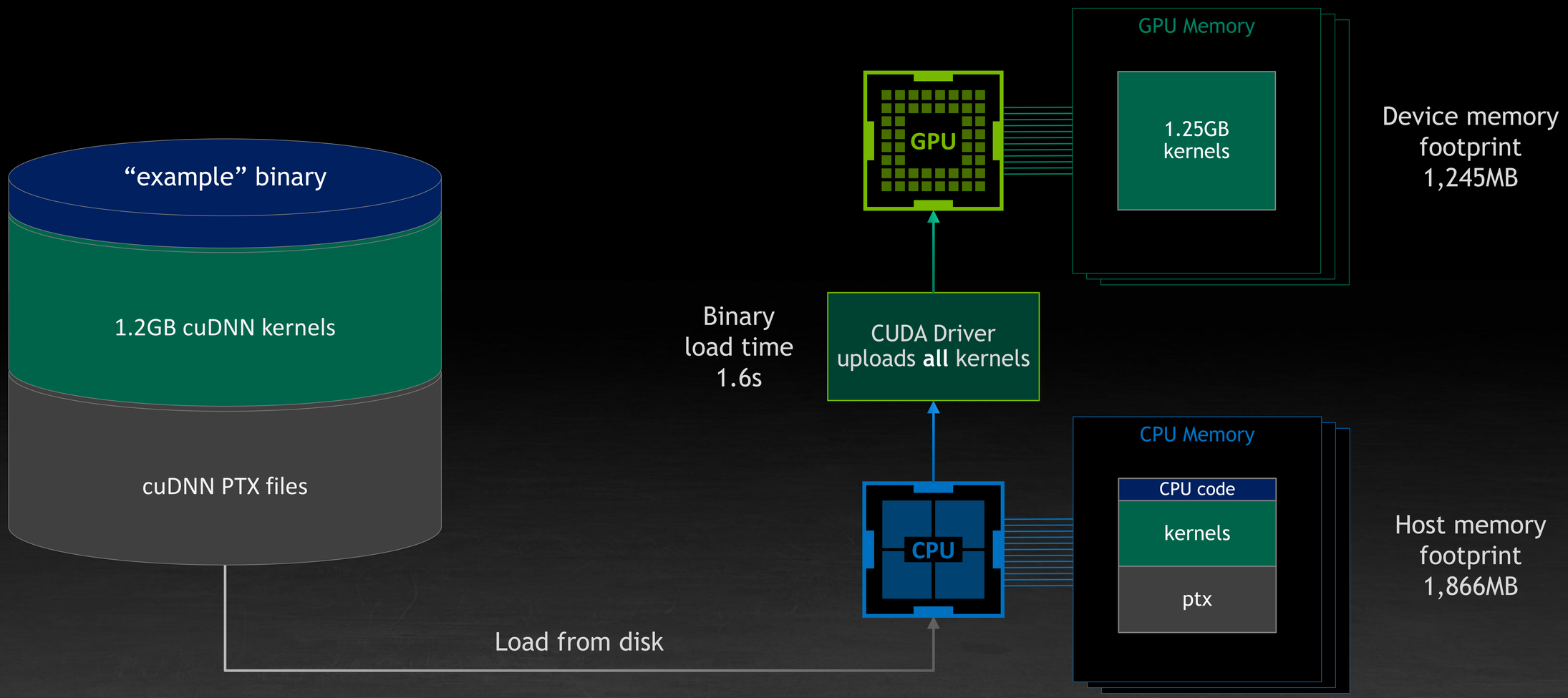
ANATOMY OF A CUDA BINARY



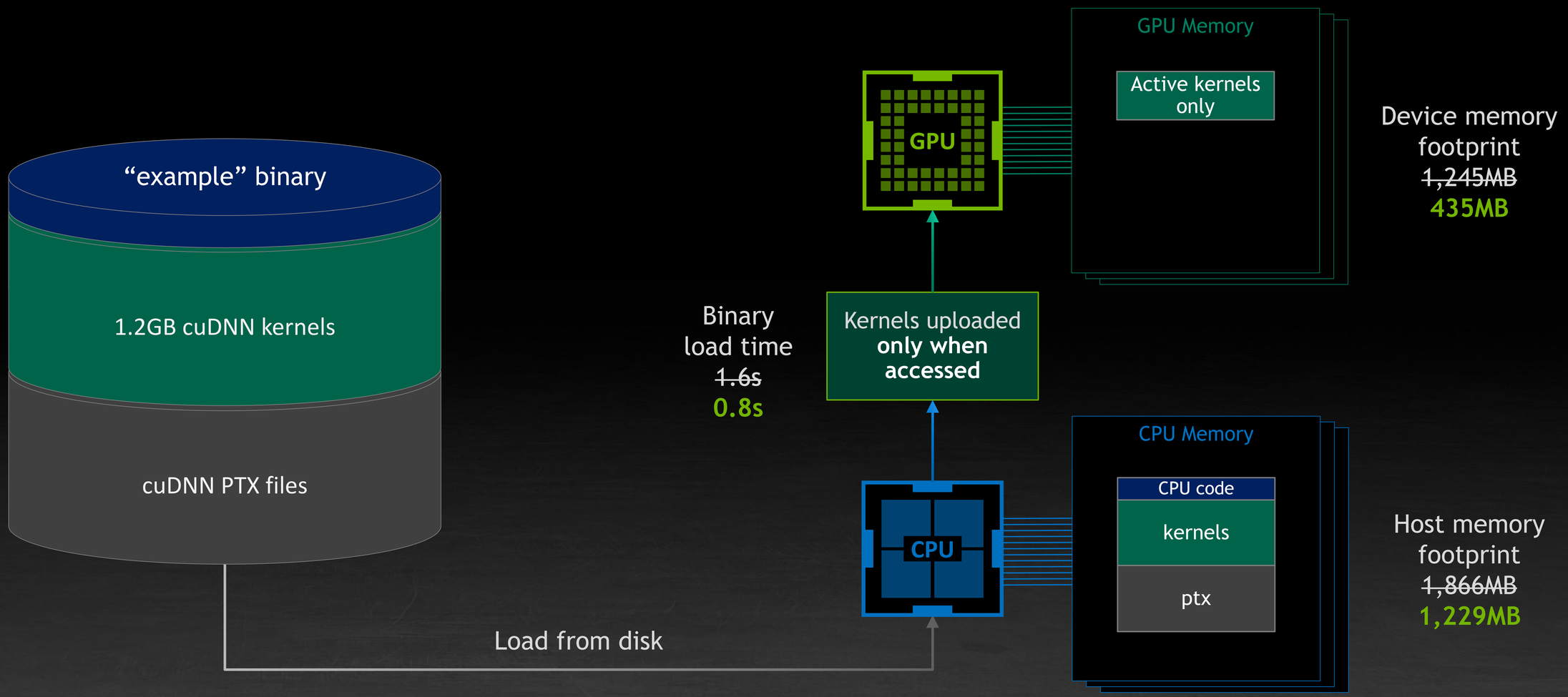
LOADING A CUDA BINARY



LOADING A CUDA BINARY

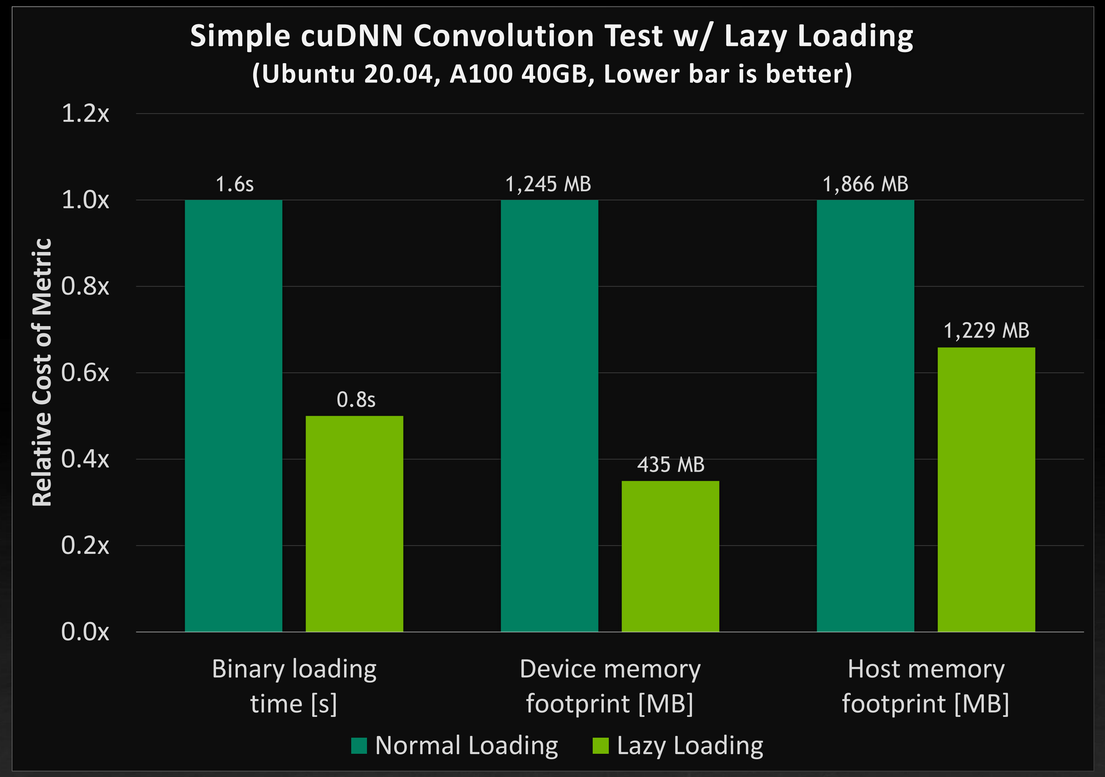


LAZY LOADING: DO NOT UPLOAD A FUNCTION UNTIL REFERENCED

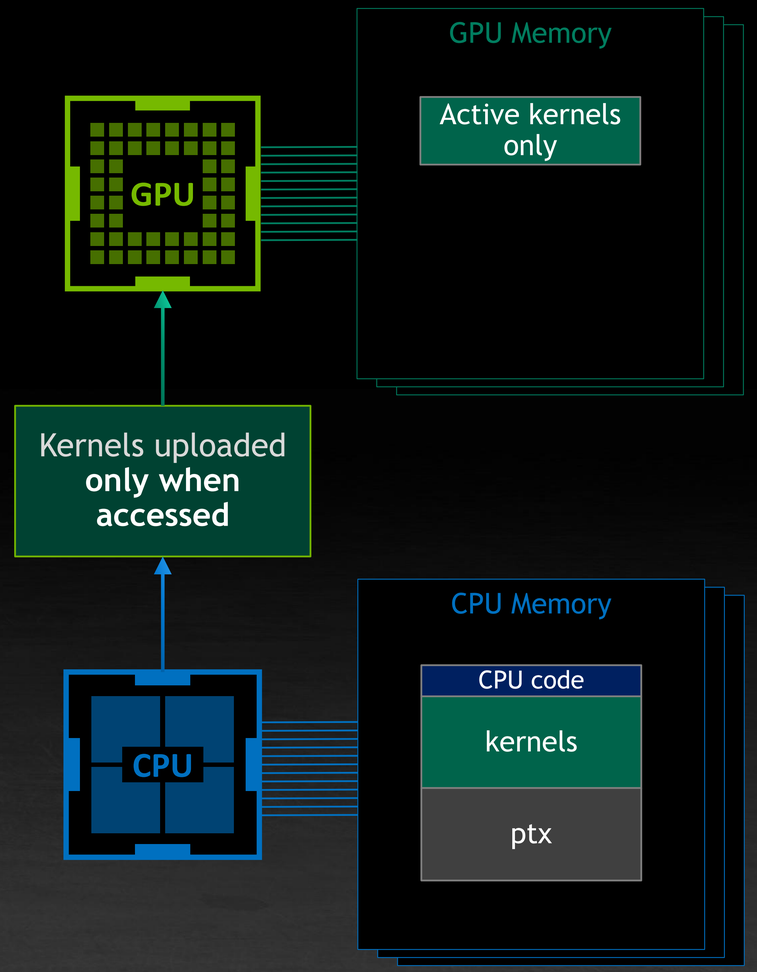


ACTIVATE USING AN ENVIRONMENT VARIABLE

CUDA_MODULE_LOADING=LAZY




```
set CUDA_MODULE_LOADING=LAZY
```

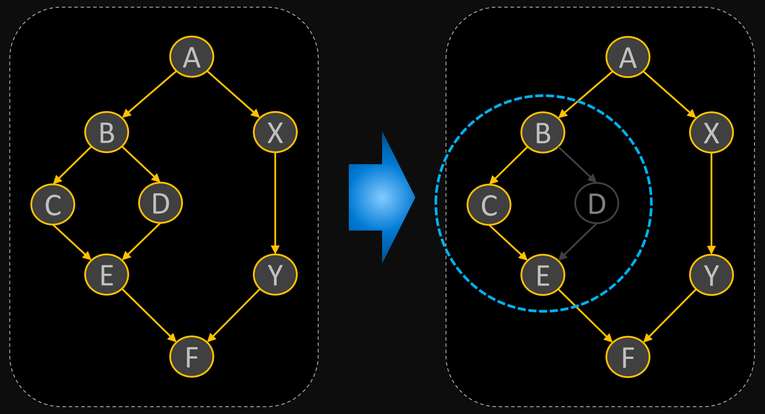


EXECUTION MANAGEMENT IN CUDA GRAPHS

Graph Node Enable/Disable

Modify execution flow prior to launch by disabling nodes in the graph

In this example, node “D” will be skipped, so left branch will execute only “BCE”



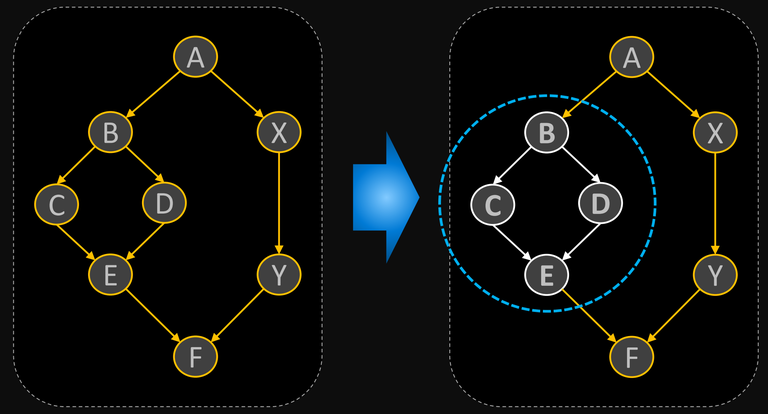
Original Graph

Node “D” disabled

Node-Specific Priority

Create high priority branches in a task graph to manage scheduling and execution order

In this example, white edges indicate high priority so “BCDE” will run with priority over “XY”



Original Graph

“BCDE” set to high priority

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::

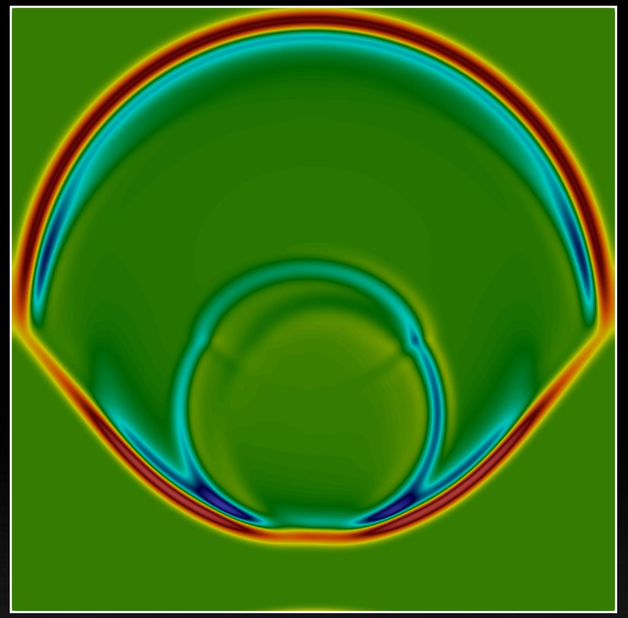
Opt-in	Does not interfere with or replace your host standard library
Heterogeneous	Copyable/Movable objects can migrate between host & device Host & Device can call all member functions Host & Device can concurrently use synchronization primitives*
Incremental	A subset of the standard library today Each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

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

STANDARD SENDERS & RECEIVERS

Looking ahead to asynchronous task graphs in Standard C++

```
sender auto maxwell_eqs(scheduler auto &compute,
                        grid_accessor A, ...) {
  return repeat_n(n_outer_iterations,
           repeat_n(n_inner_iterations,
               schedule(compute)
             | bulk(G.cells, update_h(G))
             | halo_exchange(G, hx, hy)
             | bulk(G.cells, update_e(time, dt, G))
             | halo_exchange(G, hx, hy))
           | transfer(cpu_serial_scheduler)
           | then(output_results))
         );
}
```

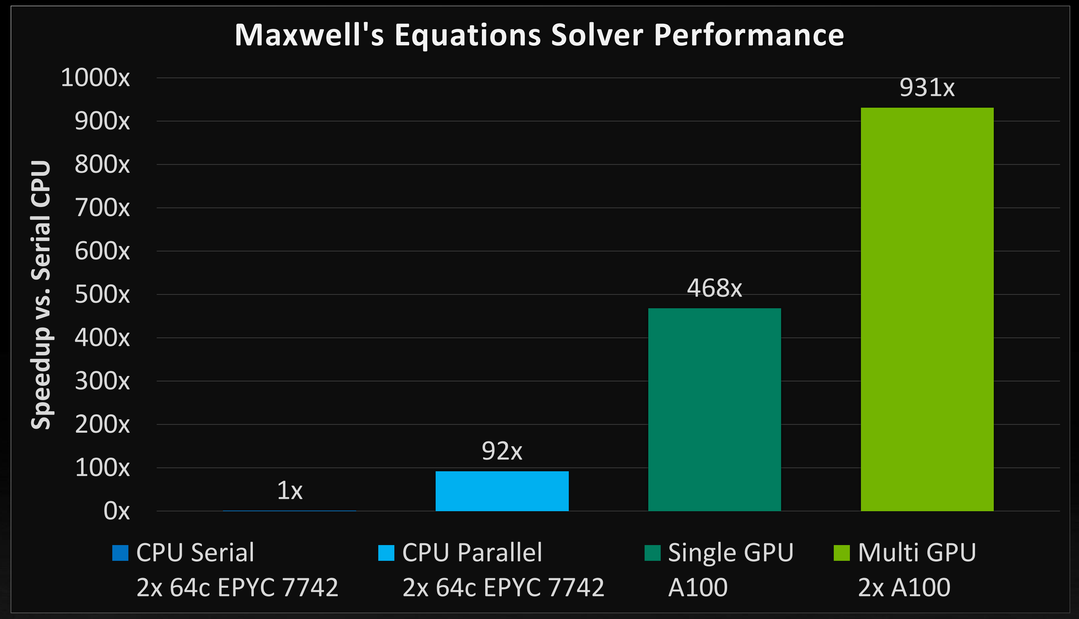


C++ Standard Parallelism [S41690]

STANDARD SENDERS & RECEIVERS

Looking ahead to asynchronous task graphs in Standard C++

```
sender auto maxwell_eqs(scheduler auto &compute,
                        grid_accessor A, ...) {
  return repeat_n(n_outer_iterations,
           repeat_n(n_inner_iterations,
               schedule(compute)
             | bulk(G.cells, update_h(G))
             | halo_exchange(G, hx, hy)
             | bulk(G.cells, update_e(time, dt, G))
             | halo_exchange(G, hx, hy))
           | transfer(cpu_serial_scheduler)
           | then(output_results))
         );
}
```




```
sync_wait(maxwell_eqs(cpu_serial_scheduler));
sync_wait(maxwell_eqs(cpu_parallel_scheduler));
sync_wait(maxwell_eqs(single_gpu_scheduler));
sync_wait(maxwell_eqs(multi_gpu_scheduler));
```

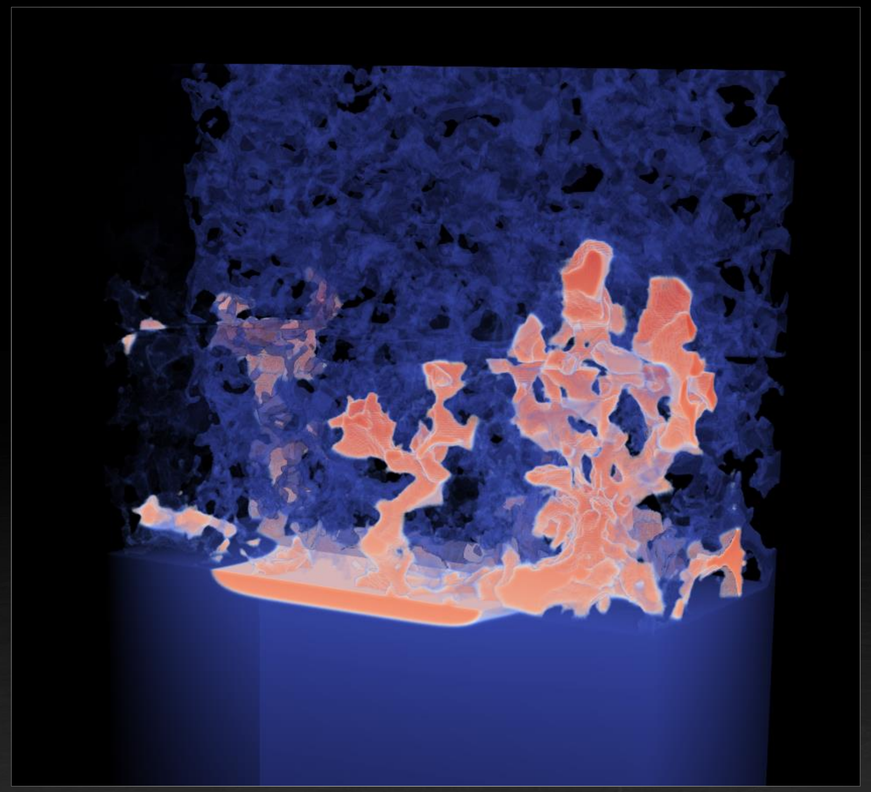
Portable standard C++ runs on any target
Change a single line of code to scale
from single-core to multi-GPU

Future of Standard and CUDA C++ [S41961, Panel]

NVIDIA

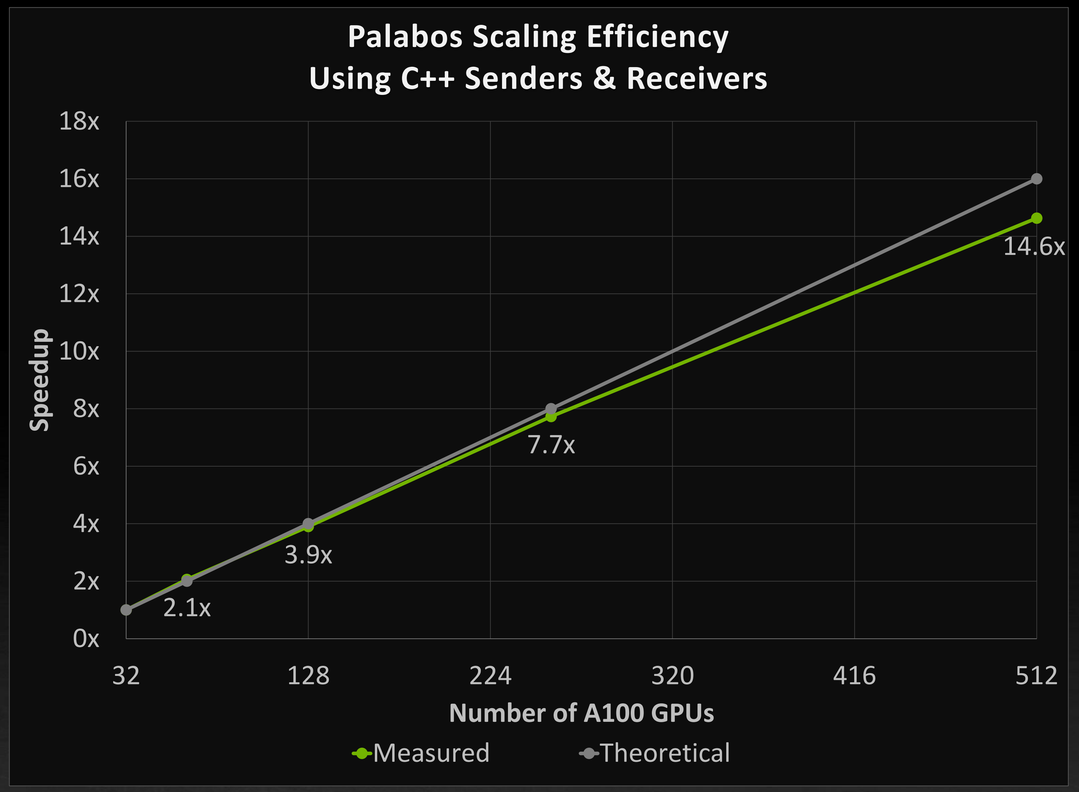
PALABOS CARBON SEQUESTRATION SIMULATION

Scale-Up Example Using Standard C++ Senders & Receivers



Buoyancy-driven diffusion of CO_2 into sandstone

Jonas Latt (University of Geneva), Christian Huber (Brown University)
Georgy Evtushenko (NVIDIA), Gonzalo Brito (NVIDIA)



Palabos is a framework for parallel computational fluid dynamics simulations using Lattice-Boltzmann methods

NVIDIA

REFERENCES

S42663	Inside the NVIDIA Hopper Architecture
S41489	Optimizing CUDA Applications for NVIDIA Hopper Architecture
(Fall ‘21) A31155	JIT LTO Adoption in cuSPARSE/cuFFT: Use Case Overview
S41491	Recent Developments in NVIDIA Math Libraries
S41493	What, Where, and Why? Use CUDA Developer Tools to Detect, Locate, and Explain Bugs and Bottlenecks
S41500	Optimizing Communication with Nsight Systems Network Profiling
S41723	How To Understand and Optimize Shared Memory Accesses using Nsight Compute
S41690	C++ Standard Parallelism
(Panel) S41961	Future of Standard and CUDA C++
CWE41721	Connect with the Experts: NVIDIA Math Libraries
CWE41541	What's in your CUDA toolbox: CUDA Profiling, Optimization, and Debugging Tools
Docs	CUDA C Programming Guide: Cooperative Groups
Blog	Implementing High-Precision Decimal Arithmetic with CUDA int128
Blog	Reducing Application Build Times Using CUDA C++ Compilation Aids
Repo	https://github.com/NVIDIA/cutlass
Repo	https://github.com/NVIDIA/NVTX