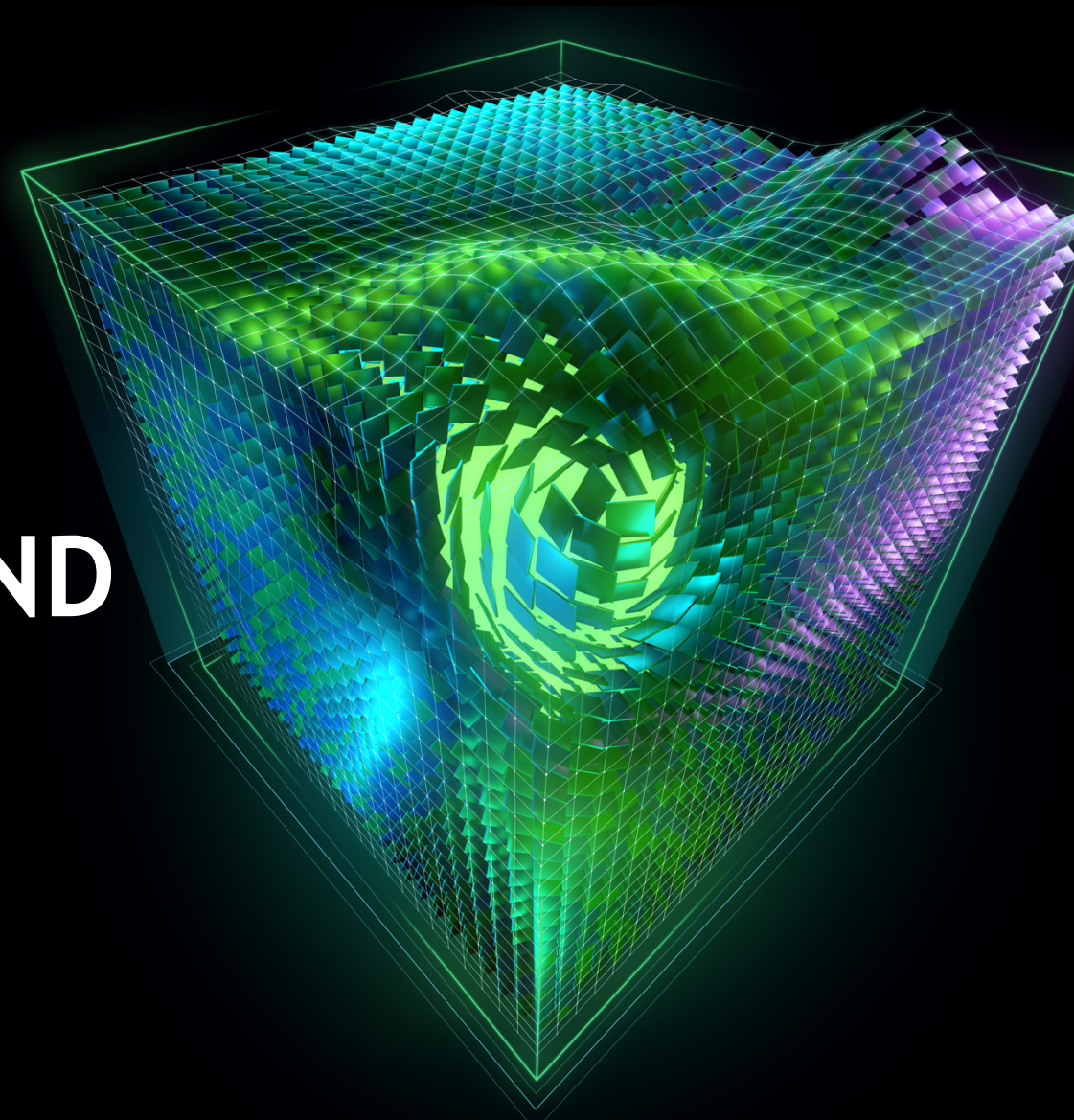
GPU TECHNOLOGY CONFERENCE

CUDA 9 AND BEYOND

Mark Harris, May 10, 2017

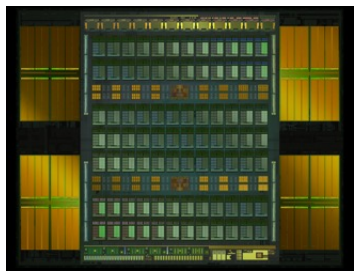


PRESENTED BY NVIDIA.

INTRODUCING CUDA 9

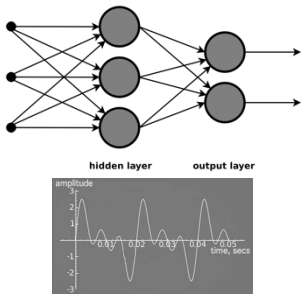
BUILT FOR VOLTA

Tesla V100
New GPU Architecture
Tensor Cores
NVLink
Independent Thread Scheduling



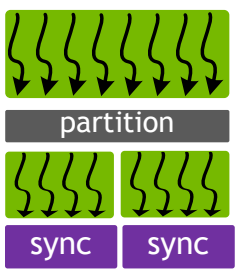
FASTER LIBRARIES

cuBLAS for Deep Learning
NPP for Image Processing
cuFFT for Signal Processing



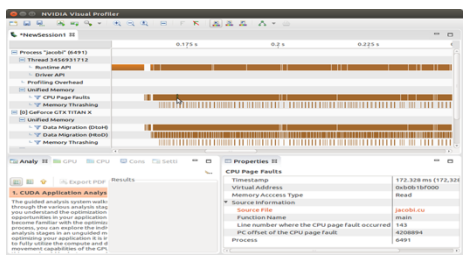
COOPERATIVE THREAD GROUPS

Flexible Thread Groups
Efficient Parallel Algorithms
Synchronize Across Thread Blocks in a Single GPU or Multi-GPUs



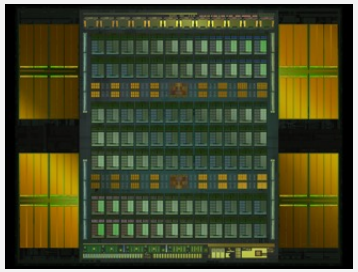
DEVELOPER TOOLS & PLATFORM UPDATES

Faster Compile Times
Unified Memory Profiling
NVLink Visualization
New OS and Compiler Support



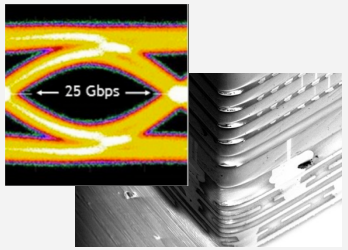
INTRODUCING TESLA V100

Volta Architecture



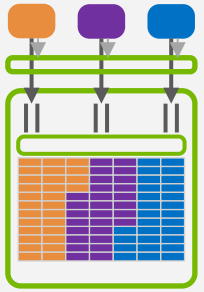
Most Productive GPU

Improved NVLink & HBM2



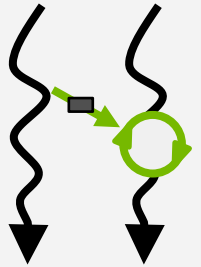
Efficient Bandwidth

Volta MPS



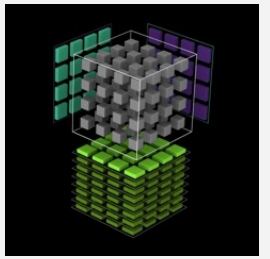
Inference Utilization

Improved SIMT Model



New Algorithms

Tensor Core

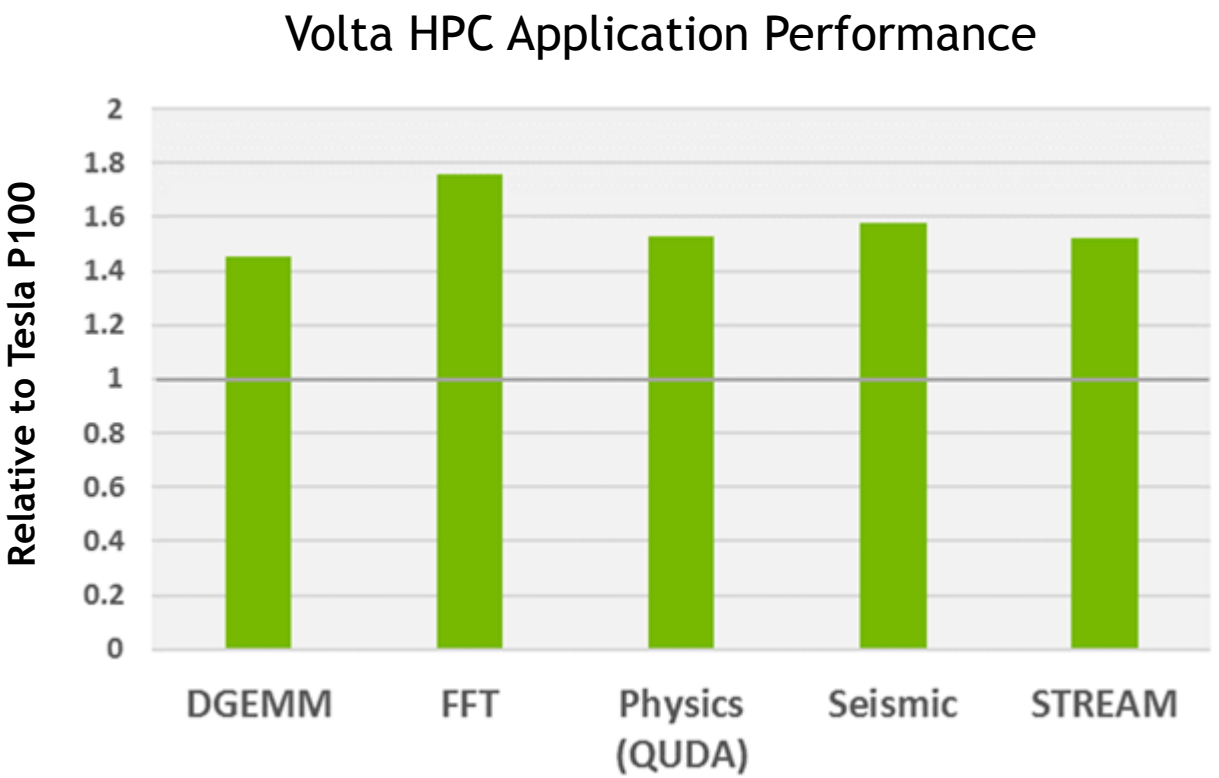


120 Programmable TFLOPS Deep Learning

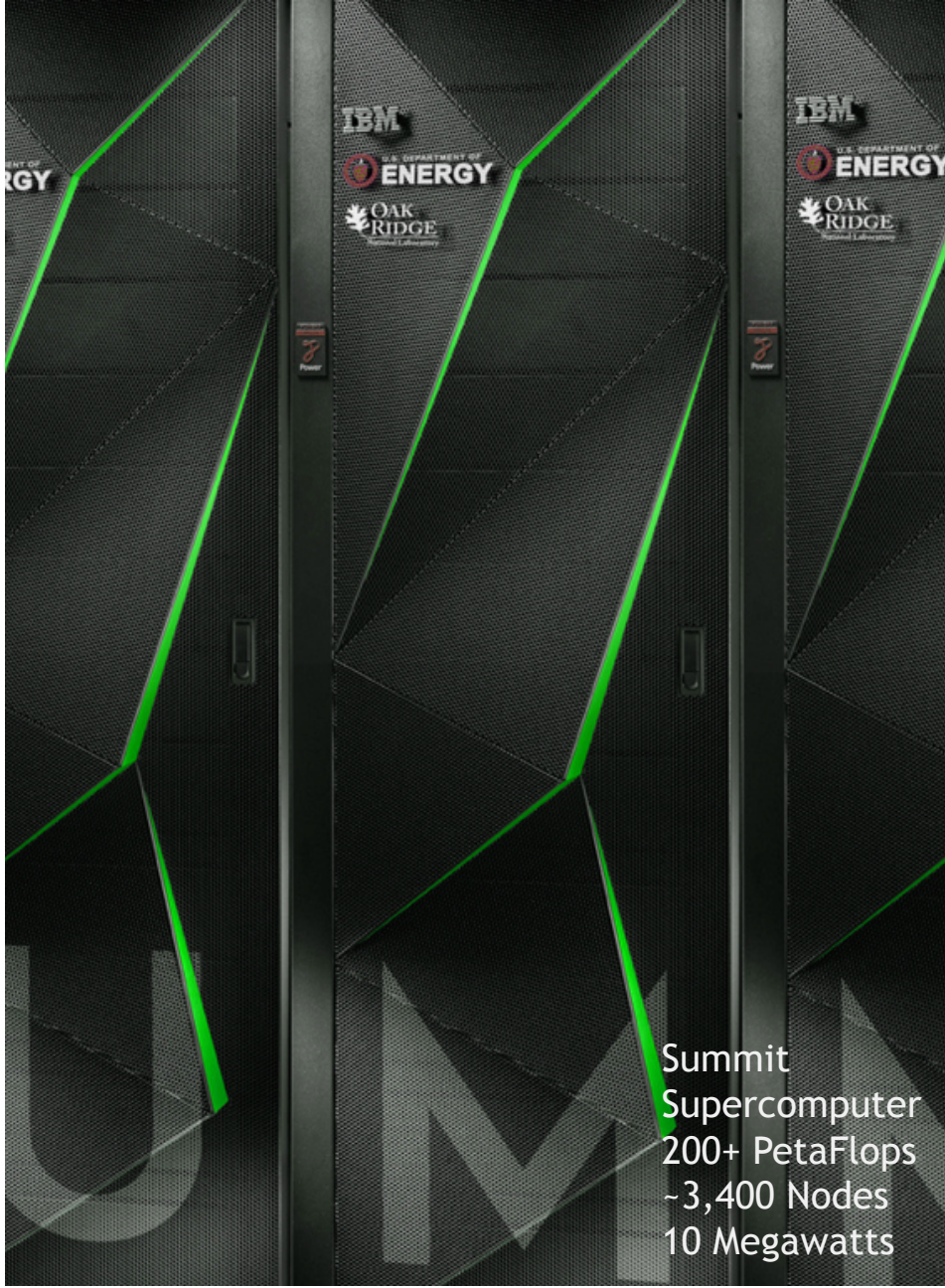
The Fastest and Most Productive GPU for Deep Learning and HPC

ROAD TO EXASCALE

Volta to Fuel Most Powerful US Supercomputers



System Config Info: 2X Xeon E5-2690 v4, 2.6GHz, w/ 1X Tesla P100 or V100. V100 measured on pre-production hardware.



FASTER LIBRARIES

CUDA 9: WHAT'S NEW IN LIBRARIES

VOLTA PLATFORM SUPPORT

Utilize Volta Tensor Cores

Volta optimized GEMMs (cuBLAS)

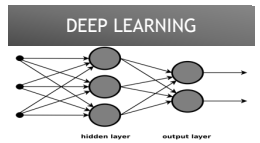
Out-of-box performance on Volta (all libraries)

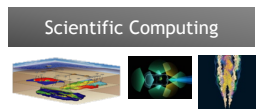
PERFORMANCE

GEMM optimizations for RNNs (cuBLAS)

Faster image processing (NPP)

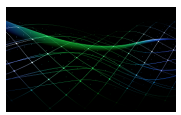
FFT optimizations across various sizes (cuFFT)



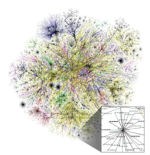


NEW ALGORITHMS

Multi-GPU dense & sparse solvers, dense eigenvalue & SVD (cuSOLVER)



Breadth first search, clustering, triangle counting, extraction & contraction (nvGRAPH)



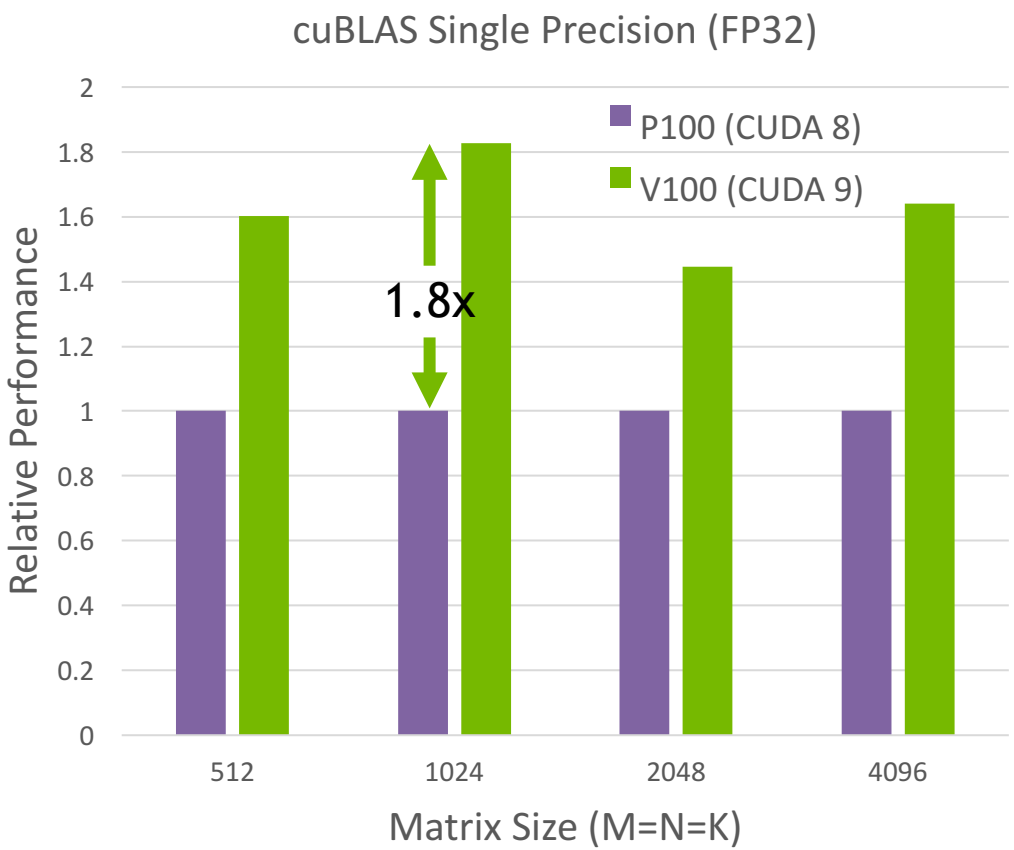
IMPROVED USER EXPERIENCE

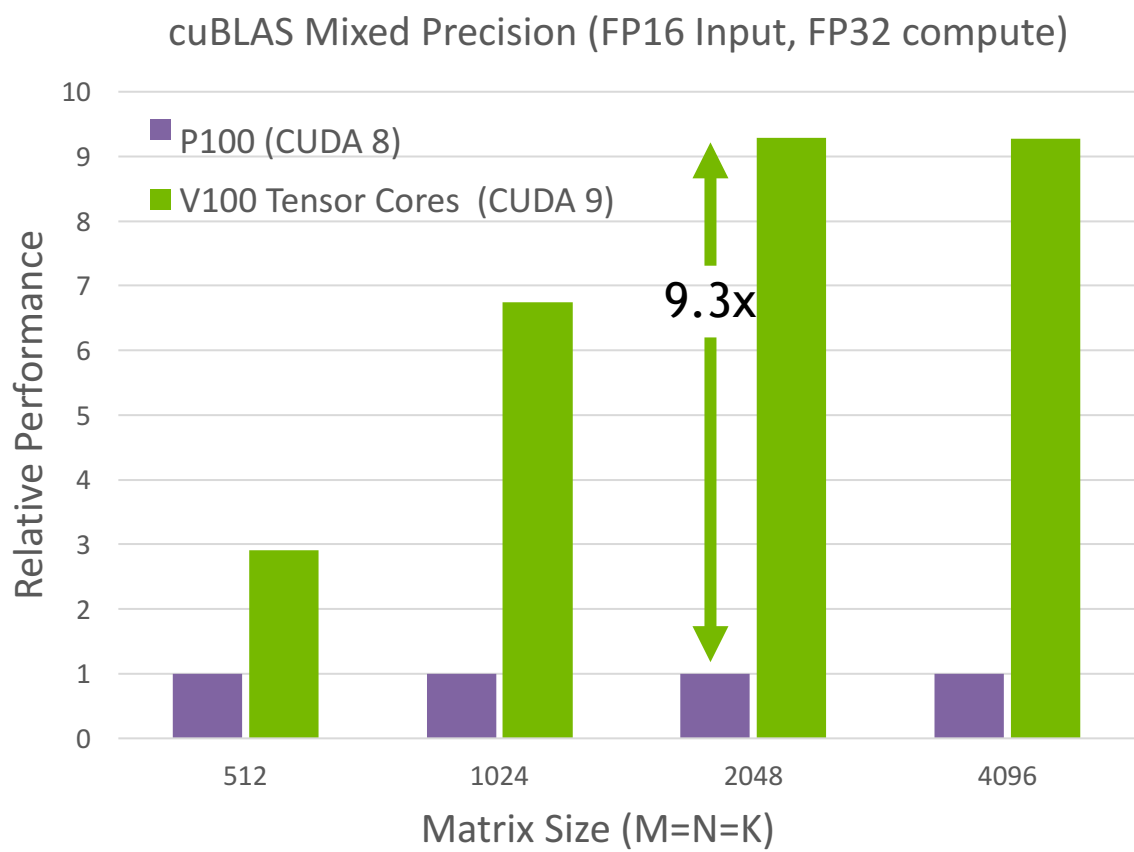
New install package for CUDA Libraries (library-only meta package)

Modular NPP with small footprint, support for image batching

cuBLAS GEMMS FOR DEEP LEARNING

V100 Tensor Cores + CUDA 9: over 9x Faster Matrix-Matrix Multiply





Note: pre-production Tesla V100 and pre-release CUDA 9. CUDA 8 GA release.

Learn More

Connect with The Experts

H7129 Accelerated Libraries:
cuFFT, cuSPARSE, cuSOLVER, nvGRAPH
Wednesday 4pm – Lower Level Pod B

S7121: Jacobi-Based Eigenvalue Solver on GPU (cuSOLVER)
Lung Sheng Chien
Tuesday, May 9, 11:00 AM - 11:25 AM, Marriott Salon 3

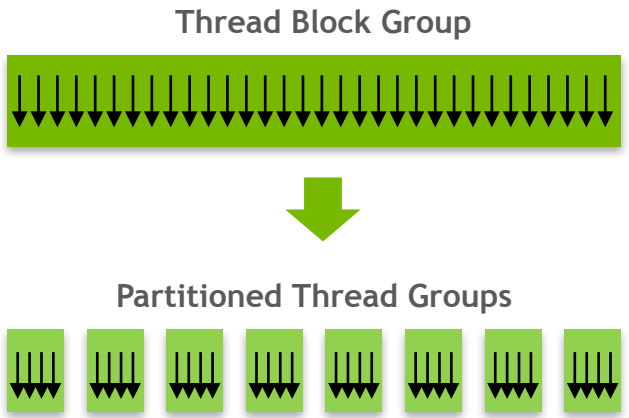
COOPERATIVE GROUPS

COOPERATIVE GROUPS

Flexible and Scalable Thread Synchronization and Communication

Define, synchronize, and partition groups of cooperating threads

- Clean composition across software boundaries
- Optimize for hardware fast path
- Scalable from a few threads to all running threads
- Deploy Everywhere: Kepler and Newer GPUs
- Supported by CUDA developer tools



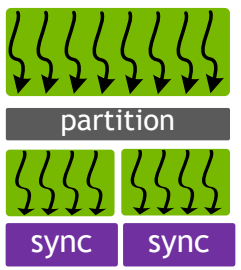
* Note: Multi-Block and Mult-Device Cooperative Groups are only supported on Pascal and above GPUs

SYNCHRONIZE AT ANY SCALE

Three Key Capabilities

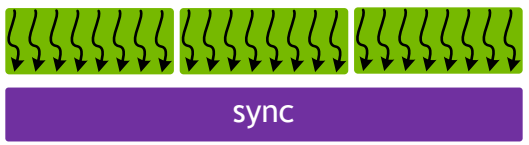
FLEXIBLE GROUPS

Define and Synchronize Arbitrary Groups of Threads

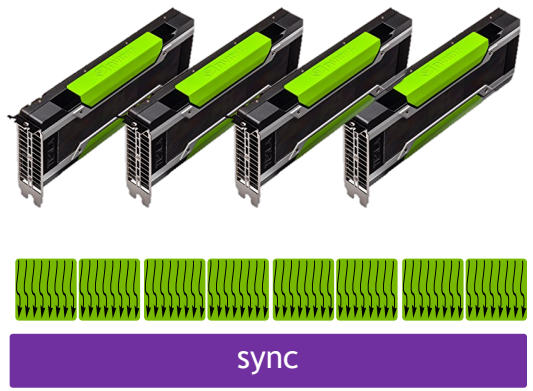


WHOLE-GRID SYNCHRONIZATION

Synchronize Multiple Thread Blocks



MULTI-GPU SYNCHRONIZATION

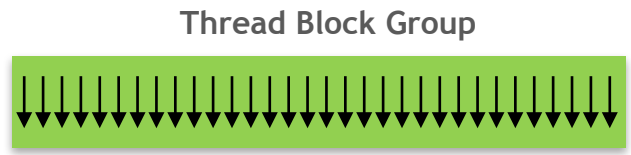


COOPERATIVE GROUPS BASICS

Flexible, Explicit Synchronization

Thread groups are explicit objects in your program

```
thread_group block = this_thread_block();
```

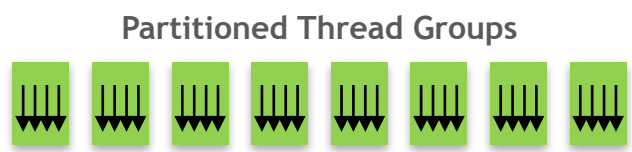


You can synchronize threads in a group

```
block.sync();
```

Create new groups by partitioning existing groups

```
thread_group tile32 = tiled_partition(block, 32);
thread_group tile4 = tiled_partition(tile32, 4);
```



Partitioned groups can also synchronize

```
tile4.sync();
```

Note: calls in green are part of the cooperative_groups:: namespace

EXAMPLE: PARALLEL REDUCTION

Composable, Robust and Efficient

Per-Block

```
g = this_thread_block();
reduce(g, ptr, myVal);
```

Per-Warp

```
g = tiled_partition<32>(this_thread_block());
reduce(g, ptr, myVal);
```

```
__device__ int reduce(thread_group g, int *x, int val) {
  int lane = g.thread_rank();
  for (int i = g.size()/2; i > 0; i /= 2) {
    x[lane] = val;        g.sync();
    val += x[lane + i];   g.sync();
  }
  return val;
}
```

LAUNCHING COOPERATIVE KERNELS

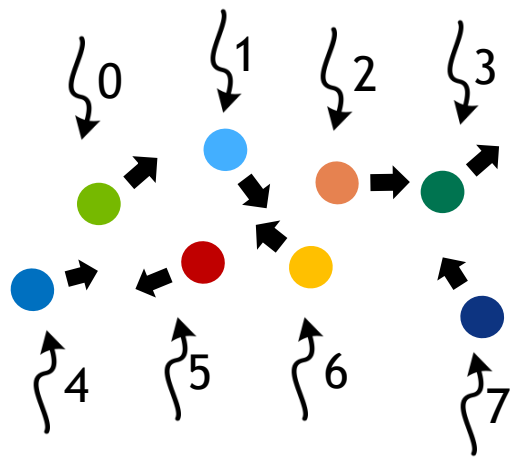
Three Synchronization Scales

Block or Sub-Block Sync	Launch with <<<>>> or `cudaLaunchKernel()`
Multi-Block Sync	Launch with `cudaLaunchCooperativeKernel()`
Multi-Device Sync	Launch with `cudaLaunchCooperativeKernelMultiDevice()`

EXAMPLE: PARTICLE SIMULATION

Without Cooperative Groups

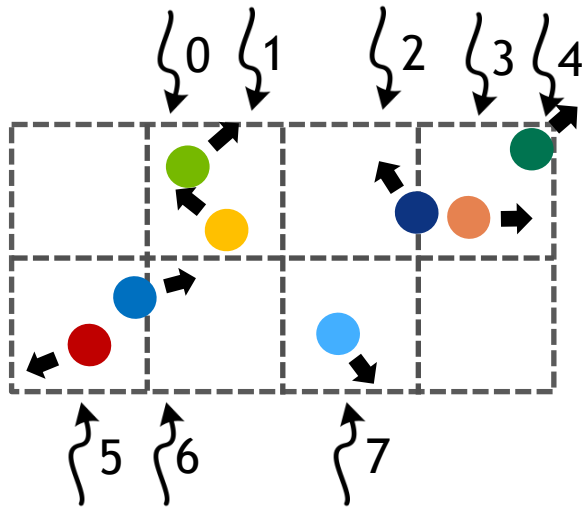
```
// threads update particles in parallel
integrate<<<blocks, threads, 0, stream>>>(particles);
```



EXAMPLE: PARTICLE SIMULATION

Without Cooperative Groups

```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);



// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

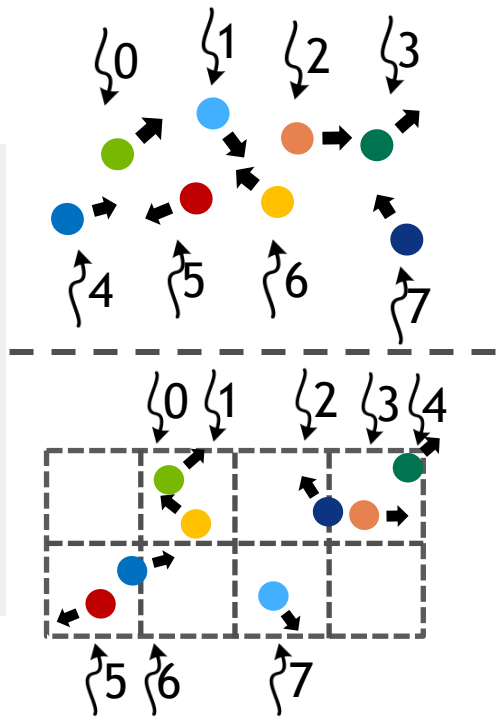


Note change in how threads map to particles in acceleration data structure

EXAMPLE: PARTICLE SIMULATION

Without Cooperative Groups

```
// threads update particles in parallel
integrate<<<blocks, threads, 0, s>>>(particles);

// Note: implicit sync between kernel launches

// Collide each particle with others in neighborhood
collide<<<blocks, threads, 0, s>>>(particles);
```

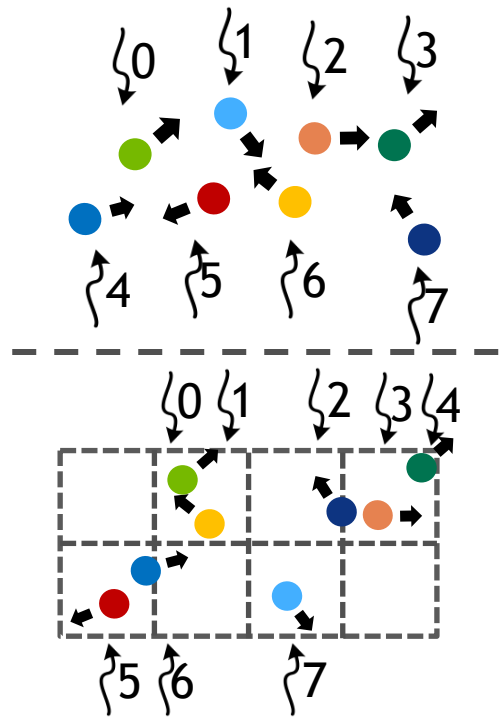


Note change in how threads map to particles in acceleration data structure

WHOLE-GRID COOPERATION

Particle Simulation Update in a Single Kernel

```
__global__ void particleSim(Particle *p, int N) {

  grid_group g = this_grid();

  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // Sync whole grid!

  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```

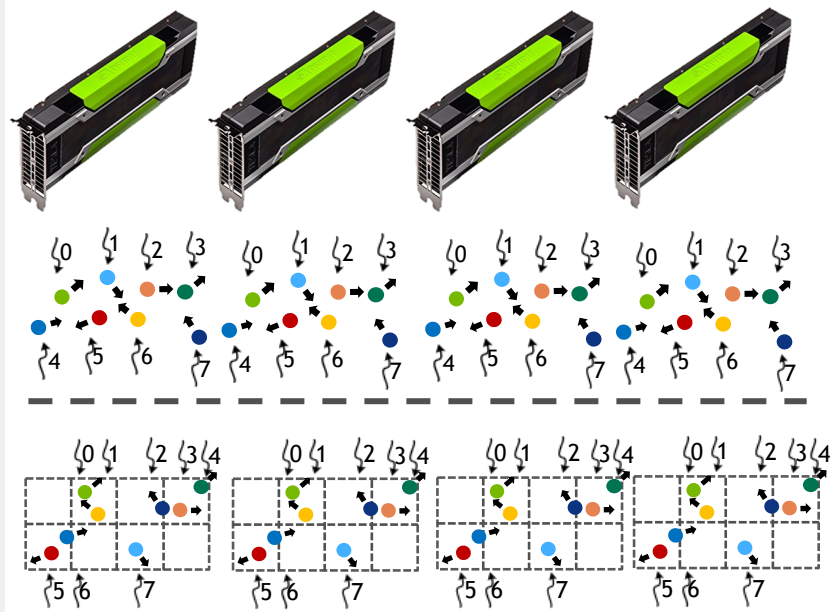


Launch using `cudaLaunchCooperativeKernel(…)`

MULTI-GPU COOPERATION

Large-scale Multi-GPU Simulation in a Single Kernel

```
__global__ void particleSim(Particle *p, int N) {

  multi_grid_group g = this_multi_grid();

  for (i = g.thread_rank(); i < N; i += g.size())
    integrate(p[i]);

  g.sync() // Sync all GPUs!

  for (i = g.thread_rank(); i < N; i += g.size())
    collide(p[i], p, N);
}
```



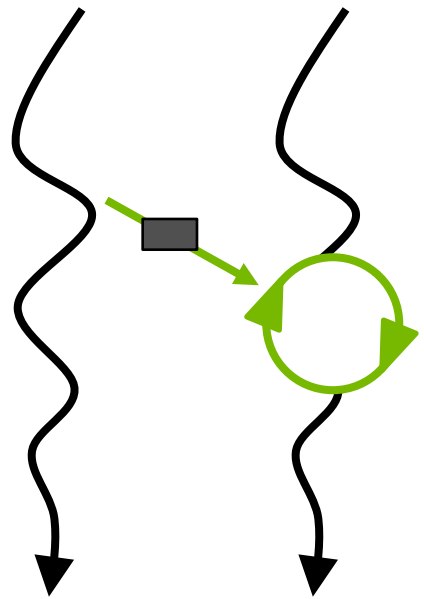
Launch using `cudaLaunchCooperativeKernelMultiDevice(…)`

ROBUST AND EXPLICIT WARP PROGRAMMING

Adapt Legacy Code for New Execution Model

Volta Independent Thread Scheduling:

- Program familiar algorithms and data structures in a natural way
- Flexible thread grouping and synchronization



Use explicit synchronization, don’t rely on implicit convergence

- CUDA 9 provides a fully explicit synchronization model

ROBUST AND EXPLICIT WARP PROGRAMMING

Adapt Legacy Code for New Execution Model

Eliminate *implicit* warp synchronous programming on all architectures

- Use explicit synchronization
- Focus synchronization granularity with Cooperative Groups

Transition to new *_sync() primitives

- __shfl_sync(), __ballot_sync(), __any_sync(), __all_sync(), __activemask()
- CUDA 9 deprecates non-synchronizing __shfl(), __ballot(), __any(), __all()

Learn More

Cooperative Groups
Session S7622

Kyrylo Perelygin and Yuan Lin

Wednesday, 4pm Marriott Ballroom 3

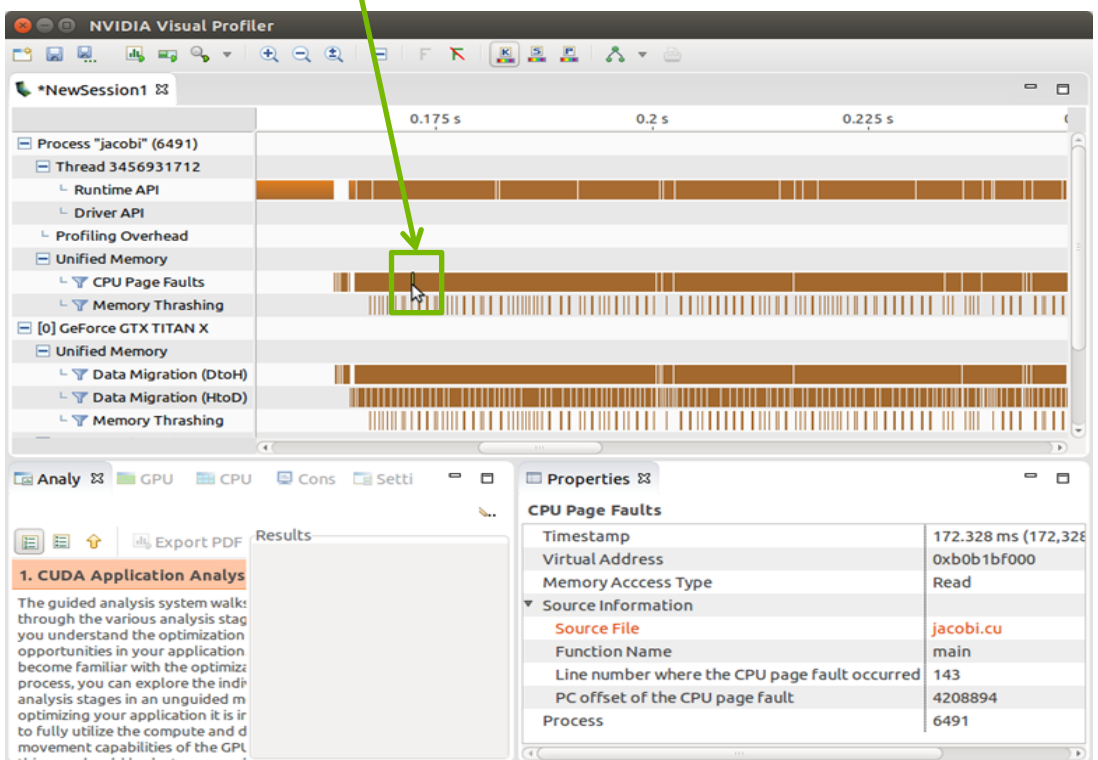
DEVELOPER TOOLS

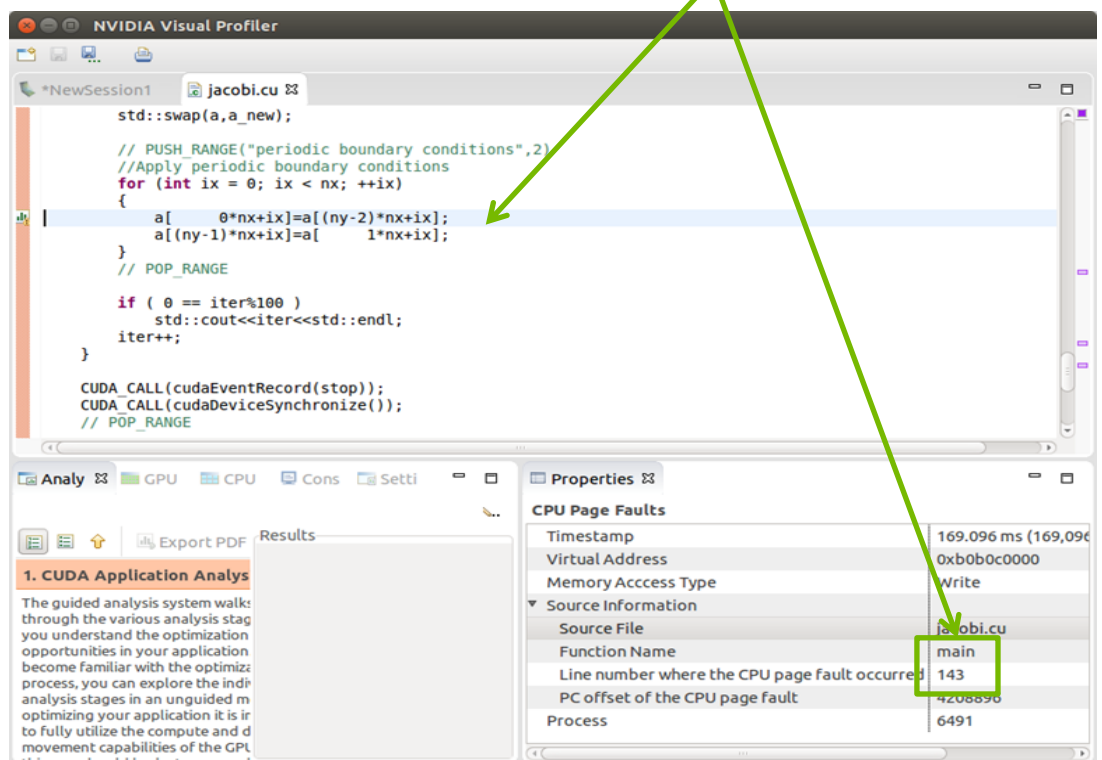
UNIFIED MEMORY PROFILING

Correlate CPU Page Faults with Source

Page Fault

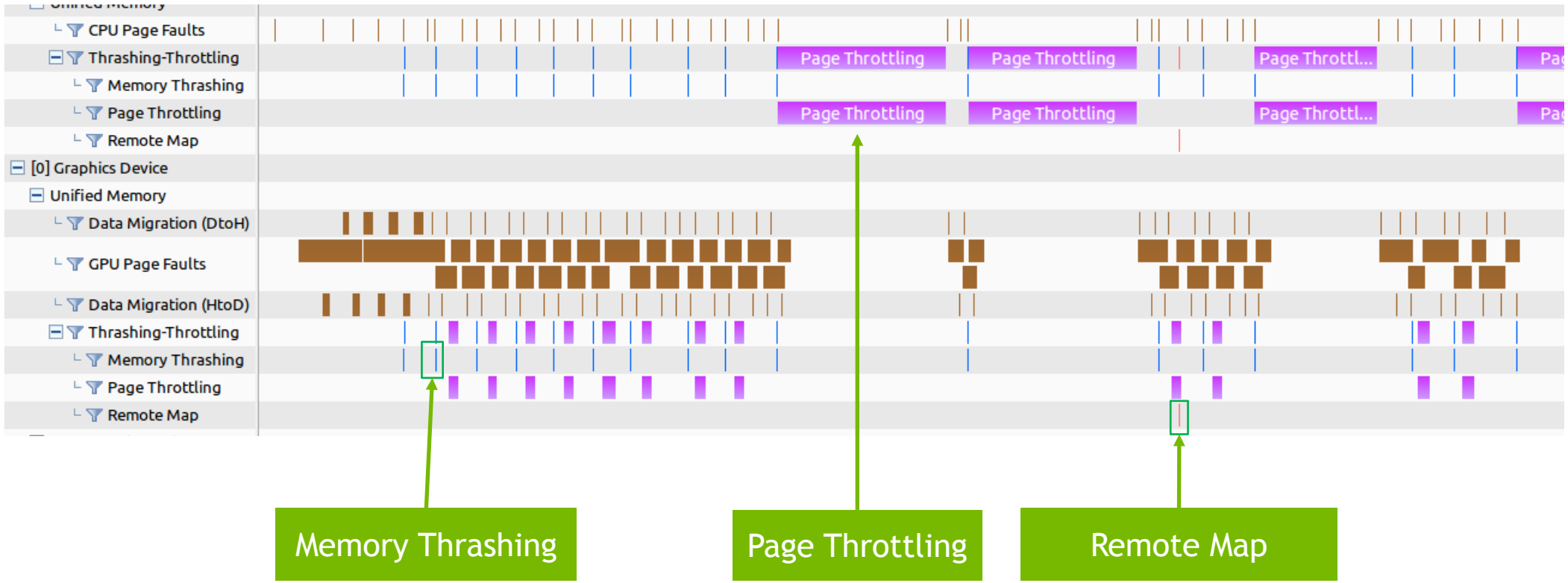
Correlation





NEW UNIFIED MEMORY EVENTS

Visualize Virtual Memory Activity



Learn More

S7495: Optimizing Application Performance with CUDA Profiling Tools
Rahul Dhoot, Sanjiv Satoor, Mayank Jain
Thursday, 10am Marriott Ballroom 3

S7824: Developer Tools update in CUDA 9.0
Rafael Campana
Wednesday, 4pm 212A

THE BEYOND SECTION

FUTURE: UNIFIED SYSTEM ALLOCATOR

Allocate unified memory using standard malloc

CUDA 8 Code with System Allocator

```
void sortfile(FILE *fp, int N) {
  char *data;

  // Allocate memory using any standard allocator
  data = (char *) malloc(N * sizeof(char));

  fread(data, 1, N, fp);

  sort<<<...>>>(data,N,1,compare);

  use_data(data);

  // Free the allocated memory
  free(data);
}
```

Removes CUDA-specific allocator restrictions

Data movement is transparently handled

Requires operating system support:

HMM Linux Kernel Module

Learn More:
HMM, Session 7764
John Hubbard
4pm Wednesday (room 211B)

USING TENSOR CORES

NVIDIA cuDNN, cuBLAS, TensorRT

Volta Optimized Frameworks and Libraries

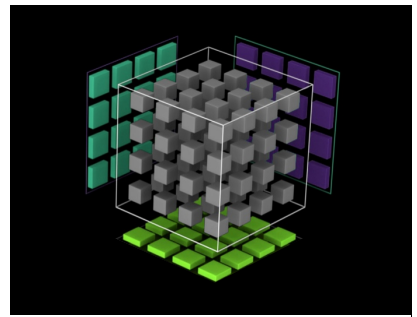
```
__device__ void tensor_op_16_16_16(
    float *d, half *a, half *b, float *c)
{
  wmma::fragment<matrix_a, …> Amat;
  wmma::fragment<matrix_b, …> Bmat;
  wmma::fragment<matrix_c, …> Cmat;

  wmma::load_matrix_sync(Amat, a, 16);
  wmma::load_matrix_sync(Bmat, b, 16);
  wmma::fill_fragment(Cmat, 0.0f);

  wmma::mma_sync(Cmat, Amat, Bmat, Cmat);

  wmma::store_matrix_sync(d, Cmat, 16,
                          wmma::row_major);
}
```

CUDA C++

Warp-Level Matrix Operations

TENSOR CORE

Mixed Precision Matrix Math
4x4 matrices



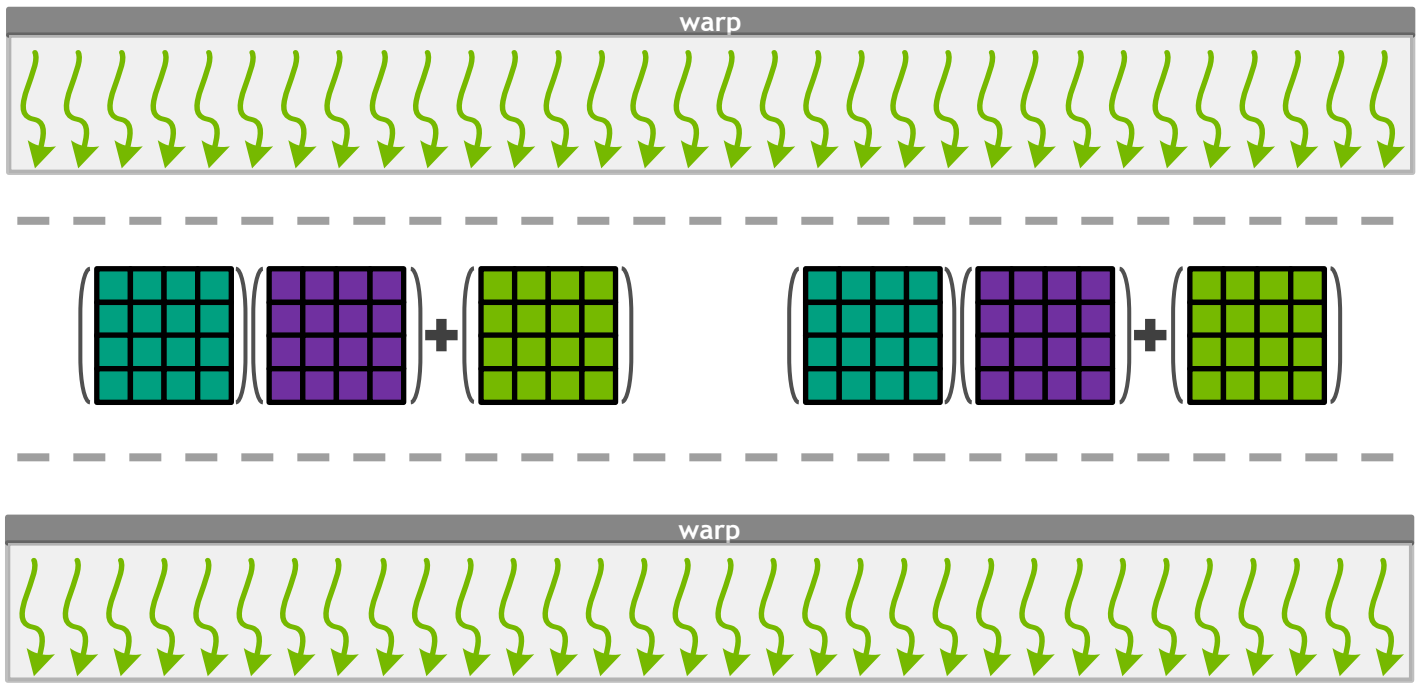
$$D = \begin{pmatrix} A_{0,0} & A_{0,1} & A_{0,2} & A_{0,3} \\ A_{1,0} & A_{1,1} & A_{1,2} & A_{1,3} \\ A_{2,0} & A_{2,1} & A_{2,2} & A_{2,3} \\ A_{3,0} & A_{3,1} & A_{3,2} & A_{3,3} \end{pmatrix} \begin{pmatrix} B_{0,0} & B_{0,1} & B_{0,2} & B_{0,3} \\ B_{1,0} & B_{1,1} & B_{1,2} & B_{1,3} \\ B_{2,0} & B_{2,1} & B_{2,2} & B_{2,3} \\ B_{3,0} & B_{3,1} & B_{3,2} & B_{3,3} \end{pmatrix} + \begin{pmatrix} C_{0,0} & C_{0,1} & C_{0,2} & C_{0,3} \\ C_{1,0} & C_{1,1} & C_{1,2} & C_{1,3} \\ C_{2,0} & C_{2,1} & C_{2,2} & C_{2,3} \\ C_{3,0} & C_{3,1} & C_{3,2} & C_{3,3} \end{pmatrix}$$

FP16 or FP32 | FP16 | FP16 | FP16 or FP32

$$D = AB + C$$

TENSOR CORE COORDINATION

Full Warp 16x16 Matrix Math



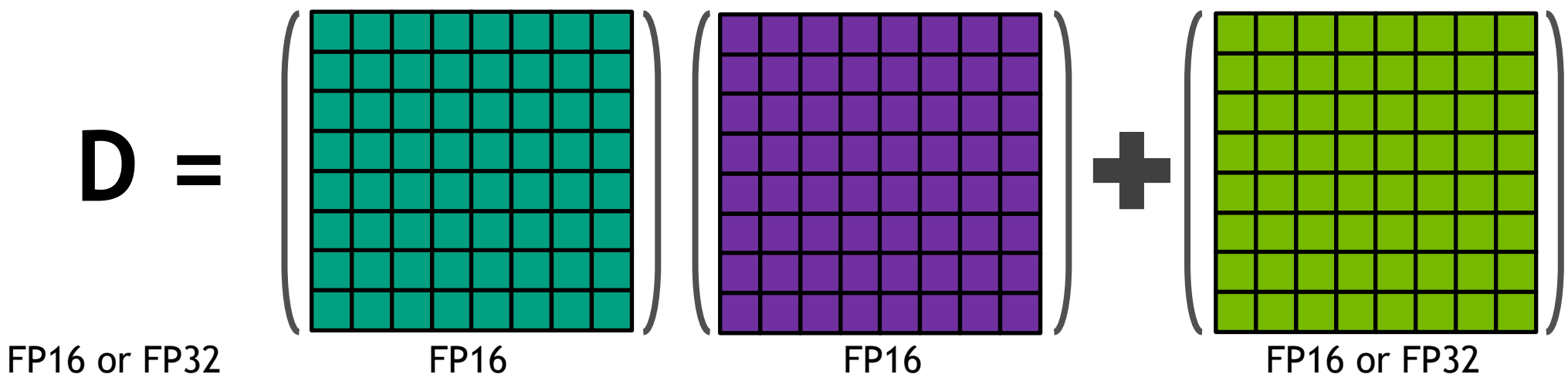
Warp-synchronizing operation for cooperative matrix math

Aggregate Matrix Multiply and Accumulate for **16x16** matrices

Result distributed across warp

CUDA TENSOR CORE PROGRAMMING

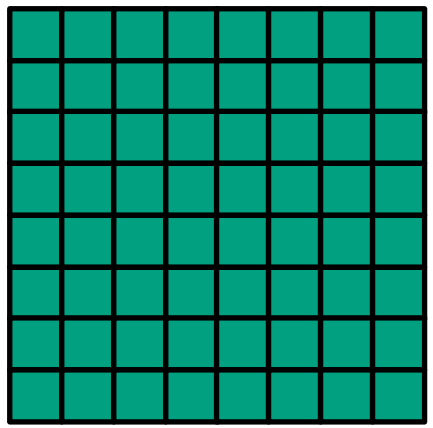
16x16x16 Warp Matrix Multiply and Accumulate (WMMA)



$$D = AB + C$$

CUDA TENSOR CORE PROGRAMMING

New WMMA datatypes



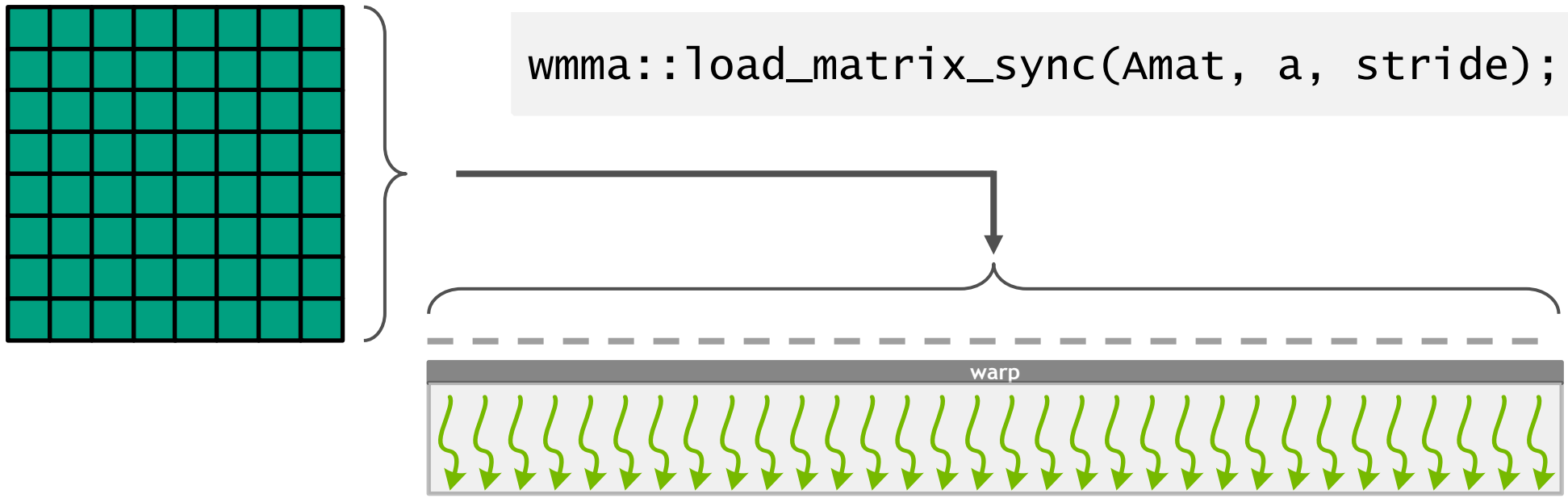
Per-Thread fragments to hold components of matrices for use with Tensor Cores

```
wmma::fragment<matrix_a, …> Amat;
```

CUDA TENSOR CORE PROGRAMMING

New WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments

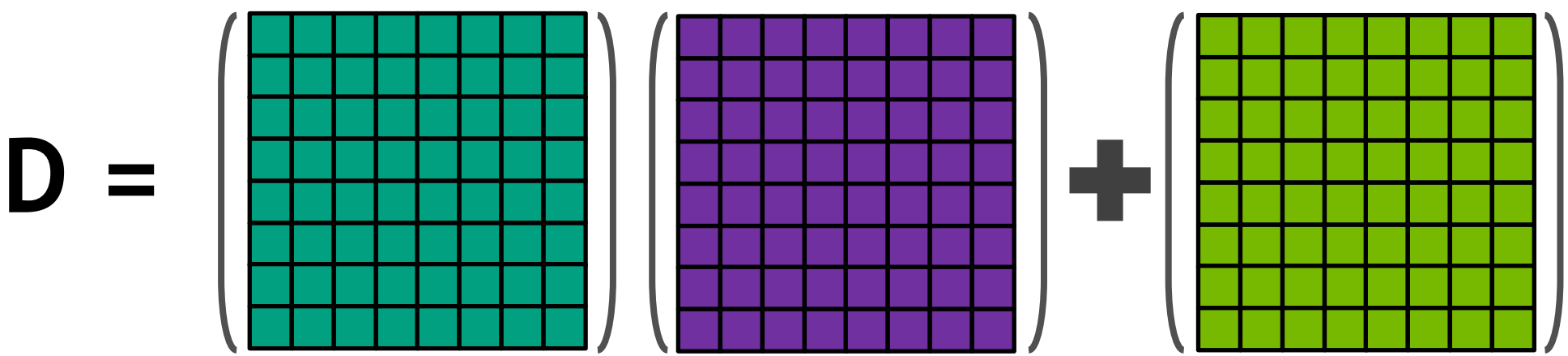


CUDA TENSOR CORE PROGRAMMING

New WMMA Matrix Multiply and Accumulate Operation

Warp-level operation to perform matrix multiply and accumulate

```
wmma::mma_sync(Dmat, Amat, Bmat, Cmat);
```

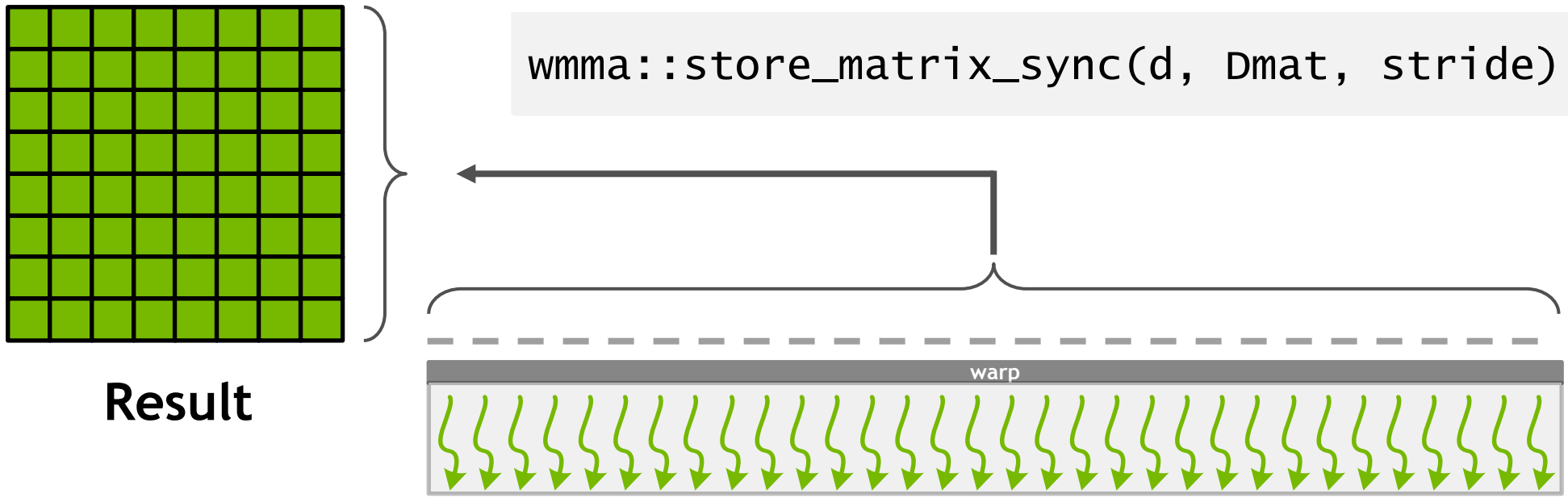


CUDA TENSOR CORE PROGRAMMING

New WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments

```
wmma::store_matrix_sync(d, Dmat, stride);
```

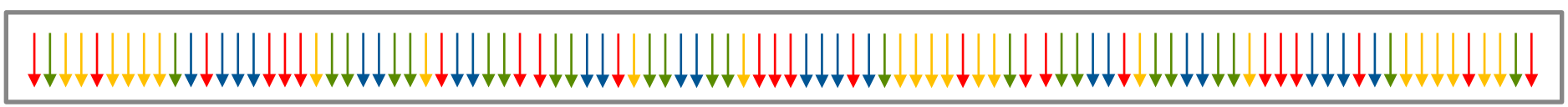


FUTURE COOPERATIVE GROUPS

Volta Enables Greater Flexibility

Partition using an arbitrary label:

```
// Four groups of threads with same computed value
int label = foo() % 4;
thread_group block = partition(this_thread_block(), label);
```



Use with care: random groups can lead to SIMT execution inefficiency

FUTURE COOPERATIVE GROUPS

Library of Collective Algorithms

Reductions, sorting, prefix sum (scan), etc.

```
// collective key-value sort using all threads in the block
cooperative_groups::sort(this_thread_block(), myValues, myKeys);
```

```
// collective scan-based allocate across block
int sz = myAllocationSize(); // amount each thread wants
int offset = cooperative_groups::exclusive_scan(this_thread_block(), sz);
```

Note: preliminary API sketch

CUDA 9 AND BEYOND

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

http://parallelforall.com

mharris@nvidia.com
@harrism