CUDA 9 AND BEYOND
Mark Harris, November 1, 2017
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

The Fastest and Most Productive GPU for Deep Learning and HPC
ROAD TO EXASCALE

Volta to Fuel Most Powerful US Supercomputers

1.5x HPC Performance in 1 Year

System Config Info: 2X Xeon E5-2690 v4, 2.6GHz, w/ 2X Tesla P100 or V100.
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.
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
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

* Note: Multi-Block and Multi-Device Cooperative Groups are only supported on Pascal and above GPUs
COOPERATIVE GROUPS BASICS
Flexible, Explicit Synchronization

Thread groups are explicit objects in your program

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

You can synchronize threads in a group

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

Create new groups by partitioning existing groups

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

Partitioned groups can also synchronize

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

Note: calls in green are part of the `cooperative_groups::` namespace
EXAMPLE: PARALLEL REDUCTION
Composable, Robust and Efficient

Per-Block

```cpp
__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];
  }
  return val;
}
```

Per-Warp

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

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

```c
__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

```c
__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: Flexible CUDA Thread Programming”
https://devblogs.nvidia.com/parallelforall/cooperative-groups/

GTC San Jose 2017: “Cooperative Groups”
Kyrylo Perelygin and Yuan Lin
http://on-demand-gtc.gputechconf.com/gtc-quicklink/pTT9h
DEVELOPER TOOLS
UNIFIED MEMORY PROFILING
Correlate CPU Page Faults with Source

Page Fault

Correlation
NEW UNIFIED MEMORY EVENTS

Visualize Virtual Memory Activity
THE BEYOND SECTION
FUTURE: UNIFIED SYSTEM ALLOCATOR

Allocate unified memory using standard malloc

CUDA 9 Code with System Allocator

```c
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

Progress Update:
HMM patchset will be integrated into Linux Kernel 4.14
NVIDIA Driver support coming
USING TENSOR CORES

Volta Optimized
Frameworks and Libraries

CUDA C++
Warp-Level Matrix Operations
TENSOR CORE
Mixed Precision Matrix Math
4x4 matrices

\[ D = AB + C \]

FP16 or FP32

FP16

FP16 or FP32
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

```cpp
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

```
wmma::load_matrix_sync(Amat, a, stride);
```
CUDA TENSOR CORE PROGRAMMING
New WMMA Matrix Multiply and Accumulate Operation

Warp-level operation to perform matrix multiply and accumulate

\[
\text{wmma::mma_sync(Dmat, Amat, Bmat, Cmat)};
\]

\[
D = \begin{pmatrix}
    \text{Green}\ 
    \end{pmatrix}
    \begin{pmatrix}
    \text{Purple}\ 
    \end{pmatrix}
    + \begin{pmatrix}
    \text{Green}\ 
    \end{pmatrix}
\]
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);
```
Learn More

Programming Tensor Cores in CUDA 9

Partition using an arbitrary label:

```cpp
// 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.

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

```cpp
// 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

JOIN THE CONVERSATION
#GTC17  

http://parallelforall.com

mharris@nvidia.com
@harrism