CUDA 9 AND BEYOND

Mark Harris, May 10, 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

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

Volta HPC Application Performance

<table>
<thead>
<tr>
<th>Relative to Tesla P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>DGEMM 1.40</td>
</tr>
<tr>
<td>FFT 1.72</td>
</tr>
<tr>
<td>Physics (QUDA) 1.59</td>
</tr>
<tr>
<td>Seismic (STAC-A2) 1.62</td>
</tr>
<tr>
<td>Finance 1.48</td>
</tr>
<tr>
<td>STREAM 1.71</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>VOLTA PLATFORM SUPPORT</th>
<th>PERFORMANCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Utilize Volta Tensor Cores</td>
<td>GEMM optimizations for RNNs (cuBLAS)</td>
</tr>
<tr>
<td>Volta optimized GEMMs (cuBLAS)</td>
<td>Faster image processing (NPP)</td>
</tr>
<tr>
<td>Out-of-box performance on Volta (all libraries)</td>
<td>FFT optimizations across various sizes (cuFFT)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NEW ALGORITHMS</th>
<th>IMPROVED USER EXPERIENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-GPU dense &amp; sparse solvers, dense eigenvalue &amp; SVD (cuSOLVER)</td>
<td>New install package for CUDA Libraries (library-only meta package)</td>
</tr>
<tr>
<td>Breadth first search, clustering, triangle counting, extraction &amp; contraction (nvGRAPH)</td>
<td>Modular NPP with small footprint, support for image batching</td>
</tr>
</tbody>
</table>
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.
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 Multi-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

```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];  g.sync();
    }
    return val;
}
```

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

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

<table>
<thead>
<tr>
<th>Block or Sub-Block Sync</th>
<th>Launch with $&lt;&lt;&lt;&gt;&gt;&gt;$ or cudaLaunchKernel()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multi-Block Sync</td>
<td>Launch with cudaLaunchCooperativeKernel()</td>
</tr>
<tr>
<td>Multi-Device Sync</td>
<td>Launch with cudaLaunchCooperativeKernelMultiDevice()</td>
</tr>
</tbody>
</table>
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

__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(…)

---

19 NVIDIA
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
NEW UNIFIED MEMORY EVENTS
Visualize Virtual Memory Activity
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

```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

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

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

NVIDIA cuDNN, cuBLAS, TensorRT
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}

\[ D = AB + C \]
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)

\[ D = (A \times B) + C \]

- \( A \) and \( B \) are multiplied and stored in FP16 format.
- \( C \) is added with the result in FP16 or FP32 format.

FP16 or FP32  \( \rightarrow \)  FP16  \( \rightarrow \)  FP16 or FP32
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

```
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

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

\[
\begin{align*}
D &= \begin{pmatrix}
\text{Green Matrix} \\
\end{pmatrix}
+ \begin{pmatrix}
\text{Purple Matrix} \\
\end{pmatrix}
\end{align*}
\]
CUDA TENSOR CORE PROGRAMMING

New WMMA load and store operations

Warp-level operation to fetch components of matrices into fragments

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

Result
FUTURE COOPERATIVE GROUPS
Volta Enables Greater Flexibility

Partition using an arbitrary label:

```c
// 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

JOIN THE CONVERSATION
#GTC17  
http://parallelforall.com
mharris@nvidia.com
@harrism
BACKUP
A thread can access the size of its group and its index (rank) in the group:

```c
thread_group group = this_thread_block();
```

```
int index = group.thread_rank();
int size = group.size();
```

Thread block groups are a special type with more functions:

```c
thread_block block = this_thread_block();
```

```
int index = block.thread_rank();
dim3 tid = block.thread_index();
dim3 bid = block.group_index();
```

Intrinsic group

Linear index

Equivalent to threadIdx (3D)

Equivalent to blockIdx (3D)
DISCOVERED CONCURRENCE
Simple, Robust Cooperation Within Warps

CUDA 8

```c
__device__ int atomicAggInc(int *p)
{
    unsigned mask = __ballot(1);
    unsigned total = __popc(mask);
    unsigned prefix = __popc(mask & __lanemask_lt());
    int lane = __ffs(mask) - 1;
    int offset = 0;
    if (prefix == 0)
        offset = atomicAdd(p, total);
    return prefix + __shfl_sync(mask, offset, lane);
}
```

CUDA 9 Cooperative Groups

```c
__device__ int atomicAggInc(int *p)
{
    coalesced_group g = coalesced_threads();
    int prev;
    if (g.thread_rank() == 0)
        offset = atomicAdd(p, g.size());
    return g.thread_rank() + g.shfl(offset, 0);
}
```

coalesced_threads() returns the group of threads that called it together (often a warp)

coalesced_group supports warp shfl()
FUTURE COOPERATIVE GROUPS
Volta Enables Greater Flexibility

Partition using an arbitrary label:

```cpp
// Group of first four threads of all warps
auto tile = tiled_partition<32>(this_thread_block());
thread_group block = partition(this_thread_block(), tile.thread_rank() < 4);

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

Use with care: random groups can lead to SIMT execution inefficiency
NPP IMAGE PROCESSING PRIMITIVES
Redesigned NPP boosts performance with smaller footprint

Over 2500 accelerated image, video & computer vision primitives

CUDA 9 streamlines NPP library

Small memory footprint

Image batching support

NPP Image Processing: 20-100x vs. CPU

<table>
<thead>
<tr>
<th>Category</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>Morphological Operations</td>
<td>100</td>
</tr>
<tr>
<td>JPEG</td>
<td>90</td>
</tr>
<tr>
<td>Geometry Transforms</td>
<td>80</td>
</tr>
<tr>
<td>Filters</td>
<td>70</td>
</tr>
<tr>
<td>Color Processing</td>
<td>60</td>
</tr>
</tbody>
</table>

NPP/Tesla V100 Speedup vs. IPP / Xeon E5-2690 (Broadwell)
EXAMPLE: PARTICLE SIMULATION

Phase 1: Integration

Phase 2: Collision Detection