CUDA 11 UPDATE
Jeff Larkin <jlarkin@nvidia.com>
OLCF October Users Call
CUDA KEY INITIATIVES

Hierarchy
Programming and running systems at every scale

Asynchrony
Creating concurrency at every level of the hierarchy

Latency
Overcoming Amdahl with lower overheads for memory & processing

Language
Supporting and evolving Standard Languages
CUDA PLATFORM: TARGETS EACH LEVEL OF THE HIERARCHY
The CUDA Platform Advances State Of The Art From Data Center To The GPU

System Scope
- FABRIC MANAGEMENT
- DATA CENTER OPERATIONS
- DEPLOYMENT
- MONITORING
- COMPATIBILITY
- SECURITY

Node Scope
- GPU-DIRECT
- NVLINK
- LIBRARIES
- UNIFIED MEMORY
- ARM
- MIG

Program Scope
- CUDA C++
- OPENACC
- STANDARD LANGUAGES
- SYNCHRONIZATION
- PRECISION
- TASK GRAPHS
PROGRAMMING GPU-ACCELERATED HPC SYSTEMS

GPU | CPU | Interconnect
Incremental Performance Optimization with Directives

Maximize GPU Performance with CUDA C++/Fortran

GPU Accelerated Libraries

GPU Accelerated C++ and Fortran

Math Libraries | Standard Languages | Directives | CUDA

---

```cpp
#include <cuda_runtime.h>

__global__ void saxpy(int n, float a, float *x, float *y) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < n) y[i] += a * x[i];
}

int main(void) {
    cudaMemcpy(d_x, x, ...
    cudaMemcpy(d_y, y, ...
    saxpy<<<(N+255)/256,256>>>(...);
    cudaMemcpy(y, d_y, ...
    std::transform(par, x, x+n, y, y,
        [=](float x, float y) {
            return y + a*x;
        });
    do concurrent (i = 1:n)
        y(i) = y(i) + a*x(i)
    enddo

    #pragma acc data copy(x,y)
    {
        ...
        std::transform(par, x, x+n, y, y,
            [=](float x, float y) {
                return y + a*x;
            });
        ...
    }
```
CUDA 11.0
Major Feature Areas

**New Platform Capabilities**
- A100 Features
- CUDA on Arm Servers

**Programming Model Updates**
- Ampere Programming Model
- New APIs for CUDA Graphs
- Flexible Thread Programming
- Memory Management APIs

**Developer Tools**
- Support for Ampere
- Roofline plots with Nsight
- Next generation correctness tools

**Math Libraries**
- Low precision datatypes in Ampere
- 3rd Gen Tensor Core support
- Leverage increased memory bandwidth, shared memory and L2 cache

---

**CUDA C++**
- C++ Modernization
- Parallel standard C++ library
- Low precision datatypes and WMMA

**Nsight Compute**
- Kernel Profiling with Rooflining

**Nsight Systems**
- System trace for Ampere

**CUDA C++**
- Support for Ampere
- Roofline plots with Nsight
- Next generation correctness tools

**Hardware decoder acceleration with nvJPEG**

**Math Libraries**
- Low precision datatypes in Ampere
- 3rd Gen Tensor Core support
- Leverage increased memory bandwidth, shared memory and L2 cache
COMPILERS
## NVCC HIGHLIGHTS IN CUDA 11.0 TOOLKIT

### Key Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>Preview feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>ISO C++ 17 CUDA Support</td>
<td></td>
</tr>
<tr>
<td>Link-Time Optimization</td>
<td></td>
</tr>
</tbody>
</table>

### New in CUDA 11.0

- Accept duplicate CLI options across all NVCC sub-components
- Host compiler support for GCC 9, clang 9, PGI 20.1
- Host compiler version check override option --allow-unsupported-compiler
- Native AArch64 NVCC binary with ARM Allinea Studio 19.2 C/C++ and PGI 20 host compiler support
LINK-TIME OPTIMIZATION

Whole-Program Compilation

Separate Compilation
All cross-compilation-unit calls must link via ABI, e.g: $x() \rightarrow y()$

ABI calls incur call overheads
**LINK-TIME OPTIMIZATION**

Whole-Program Compilation

```
whole.cu
  x();
  y();
  .ptx
  ptxas
  Executable
```

Link-Time Optimization

Permits inlining of device functions across modules
Mitigates ABI call overheads
Facilitates Dead Code Elimination

```
a.cu
  x();
  -dlto
  cicc
  LTO
  nvlink
  ptxas
  Executable

b.cu
  y();
  -dlto
  cicc
  libnvvm
```
LINK-TIME OPTIMIZATION
Preview Release in CUDA 11.0

Enabled through `-dlto` option for compile and link steps
Partial LTO (mix of separate compilation & LTO) supported
AVAILABLE NOW: THE NVIDIA HPC SDK
Available at developer.nvidia.com/hpc-sdk, on NGC, and in the Cloud

NVIDIA HPC SDK

### DEVELOPMENT

<table>
<thead>
<tr>
<th>Programming Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standard C++ &amp; Fortran</td>
</tr>
<tr>
<td>nvcc, nvc</td>
</tr>
<tr>
<td>OpenACC &amp; OpenMP</td>
</tr>
<tr>
<td>nvc++</td>
</tr>
<tr>
<td>CUDA</td>
</tr>
<tr>
<td>nvfortran</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Compilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>nvcc</td>
</tr>
<tr>
<td>nvc++</td>
</tr>
<tr>
<td>nvc</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Core Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>libc++</td>
</tr>
<tr>
<td>Thrust</td>
</tr>
<tr>
<td>CUB</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Math Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuBLAS</td>
</tr>
<tr>
<td>cuSOLVER</td>
</tr>
<tr>
<td>cuRAND</td>
</tr>
<tr>
<td>cuSPARSE</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Communication Libraries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Open MPI</td>
</tr>
<tr>
<td>NVSHMEM</td>
</tr>
<tr>
<td>NCCL</td>
</tr>
</tbody>
</table>

### ANALYSIS

<table>
<thead>
<tr>
<th>Profilers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nsight</td>
</tr>
<tr>
<td>Systems</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Debugger</th>
</tr>
</thead>
<tbody>
<tr>
<td>cuda-gdb</td>
</tr>
<tr>
<td>Host</td>
</tr>
<tr>
<td>Device</td>
</tr>
</tbody>
</table>

Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect
HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA
7-8 Releases Per Year | Freely Available
HPC COMPILERS
NVC | NVC++ | NVFORTRAN

Accelerated
Latest GPUs
Automatic Acceleration

Programmable
Standard Languages
Directives
CUDA

Multicore
Directives
Vectorization

Multi-Platform
x86_64
Arm
OpenPOWER

OpenACC
CUDA
OpenMP
Fortran
C++
## HPC Programming in ISO C++

ISO is the place for portable concurrency and parallelism

### C++17
- **Parallel Algorithms**
  - In NVC++ 20.5
  - Parallel and vector concurrency
- **Forward Progress Guarantees**
  - Extend the C++ execution model for accelerators
- **Memory Model Clarifications**
  - Extend the C++ memory model for accelerators

### C++20
- **Scalable Synchronization Library**
  - Express thread synchronization that is portable and scalable across CPUs and accelerators
  - In libcu++ in CUDA 10.2:
    - `std::atomic<T>`
  - In libcu++ in CUDA 11.0:
    - `std::barrier`
    - `std::counting_semaphore`
    - `std::atomic<T>::wait/notify_*`
  - In Libcu++ in the future:
    - `std::atomic_ref<T>`

### C++23 and Beyond
- **Executors**
  - Simplify launching and managing parallel work across CPUs and accelerators
  - `std::mdspan/mdarray`
  - HPC-oriented multi-dimensional array abstractions.
- **Linear Algebra**
  - C++ standard algorithms API to linear algebra
  - Maps to vendor optimized BLAS libraries
- **Extended Floating Point Types**
  - First-class support for formats new and old: `std::float16_t/float64_t`
static inline
void CalcHydroConstraintForElems(Domain &domain, Index_t length,
    Index_t *regElemlist, Real_t dvovmax, Real_t &dthydro)
{
    #if _OPENMP
    const Index_t threads = omp_get_max_threads();
    Index_t hydro_elem_per_thread[threads];
    Real_t dthydro_per_thread[threads];
    #else
    Index_t threads = 1;
    Index_t hydro_elem_per_thread[1];
    Real_t dthydro_per_thread[1];
    #endif
    #pragma omp parallel firstprivate(length, dvovmax)
    {
        Real_t dthydro_tmp = dthydro;
        Index_t hydro_elem = -1;
        #if _OPENMP
        Index_t thread_num = omp_get_thread_num();
        #else
        Index_t thread_num = 0;
        #endif
        #pragma omp for
        for (Index_t i = 0; i < length; ++i) {
            Index_t indx = regElemlist[i];
            if (domain.vdov(indx) == Real_t(0.0)) {
                Real_t dtdvov = dvovmax / (FABS(domain.vdov(indx)) + Real_t(1.e-20));
                if (dthydro_tmp > dtdvov) {
                    dthydro_tmp = dtdvov;
                    hydro_elem = indx;
                }
            }
        }
        dthydro_per_thread[thread_num] = dthydro_tmp;
        hydro_elem_per_thread[thread_num] = hydro_elem;
    }
    for (Index_t i = 1; i < threads; ++i) {
        if (hydro_elem_per_thread[i] < hydro_elem_per_thread[0]) {
            hydro_elem_per_thread[0] = hydro_elem_per_thread[i];
            hydro_elem_per_thread[0] = hydro_elem_per_thread[i];
        }
    }
    if (hydro_elem_per_thread[0] != -1) {
        dthydro = dthydro_per_thread[0];
    }
    return;
}

PARALLEL C++

➢ Composable, compact and elegant
➢ Easy to read and maintain
➢ ISO Standard
➢ Portable - nvc++, g++, icpc, MSVC, ...

static inline void CalcHydroConstraintForElems(Domain &domain, Index_t length,
    Index_t *regElemlist, Real_t dvovmax, Real_t &dthydro)
{
    dthydro = std::transform_reduce(
        std::execution::par, counting_iterator(0), counting_iterator(length),
        dthydro, [](Real_t a, Real_t b) { return a < b ? a : b; },
        [=, &domain](Index_t i)
    {
        Index_t indx = regElemlist[i];
        if (domain.vdov(indx) == Real_t(0.0)) {
            return std::numeric_limits<Real_t>::max();
        } else {
            return dvovmax / (std::abs(domain.vdov(indx)) + Real_t(1.e-20));
        }
    });
}
LULESH PERFORMANCE

Speedup - Higher is Better

Same ISO C++ Code
PARALLEL C++ & CYTHON
Using NVC++ and CYTHON to Accelerate Python

A100 Performance for Python

- Access to C++ performance with Cython
- A100 Acceleration with NVC++ stdpar
- Up to 30X Speed-up over Numpy

```python
def cppsort(np.ndarray[np.float_t, ndim=1] x):
    cdef vector[float] vec
    vec.resize(x.shape[0])
    copy_n(&x[0], len(x), vec.begin())
    sort(par, vec.begin(), vec.end())
    copy_n(vec.begin(), len(x), &x[0])
```

Cython cppsort Speed-up over Numpy

- seq  execution policy with g++
- par  execution policy with nvc++ on A100
## HPC PROGRAMMING IN ISO FORTRAN

ISO is the place for portable concurrency and parallelism

### Fortran 2018

**Array Syntax and Intrinsics**
- NVFORTRAN 20.5
- Accelerated matmul, reshape, spread, etc

**DO CONCURRENT**
- NVFORTRAN 20.x
- Auto-offload & multi-core

**Co-Arrays**
- Coming Soon
- Accelerated co-array images

### Fortran 202x

**DO CONCURRENT Reductions**
- REDUCE subclause added
- Support for +, *, MIN, MAX, IAND, IOR, IEOR.
- Support for .AND., .OR., .EQV., .NEQV on LOGICAL values
- Atomics
Fortran with OpenACC

DO CONCURRENT (kx=mx:my, jx=mx:jy)
    LOCAL (left_flux, right_flux, top_flux, bottom_flux, total_flux, min_cell_volume, energy_change, recip_volume)
    DO jx=mx:jy
        left_flux = (xarea(j,k) * xvel0(j,k) + xvel1(j,k)) * 0.25 * dt * 0.5
        right_flux = (xarea(j+1,k) * xvel0(j+1,k) + xvel1(j+1,k)) * 0.25 * dt * 0.5
        bottom_flux = (yarea(j,k) * yvel0(j,k) + yvel1(j,k)) * 0.25 * dt * 0.5
        top_flux = (yarea(j,k+1) * yvel0(j,k+1) + yvel1(j,k+1)) * 0.25 * dt * 0.5
        total_flux = right_flux - left flux + top_flux - bottom_flux
        volume_change(j,k) = volume(j,k) / total_flux
        min_cell_volume = MIN(volume(j,k), right_flux - left_flux, top_flux - bottom_flux)
        recip_volume = 1.0 / volume(j,k)
        energy_change = (pressure(j,k) * density0(j,k) + viscosity(j,k) / density(k,j)) ... energy(j,k) * energy(j,k) / density(j,k) * volume_change(j,k)
    ENDDO
ENDDO

ISO Fortran
CLOVERLEAF PERFORMANCE

Time - Lower is Better

CPU ACC
V100 DO CONCURRENT
V100 ACC

CPU System: Skylake 2x20 core Xeon Gold server, one thread per core
HPC PROGRAMMING IN ISO FORTRAN

NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

MATMUL FP64 matrix multiply

```
real(8), dimension(n1,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(n1,nj) :: c, d

!$acc enter data copyin(a,b,c) create(d)
do nt = 1, ntimes
  !$acc kernels
  do j = 1, nj
    do i = 1, n1
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
end do
!$acc end kernels
!$acc end data copyout(d)
```

Inline FP64 matrix multiply

```
real(8), dimension(n1,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(n1,nj) :: c, d

!$acc enter data copyin(a,b,c) create(d)
do nt = 1, ntimes
  !$acc kernels
  do j = 1, nj
    !$acc host_data use_device(a,b,c)
    do i = 1, n1
      d(i,j) = c(i,j)
      do k = 1, nk
        d(i,j) = d(i,j) + a(i,k) * b(k,j)
      end do
    end do
  end do
!$acc end host_data
end do
!$acc exit data copyout(d)
```
INTRODUCING NVSHMEM
GPU Optimized OpenSHMEM

- Initiate from CPU or GPU
- Initiate from within CUDA kernel
- Issue onto a CUDA stream
- Interoperable with MPI & OpenSHMEM

Pre-release Impact
- LBANN, Kokkos/CGSolve, QUDA
INTRODUCING NVSHMEM

Impact in HPC Applications

➢ Up to 1.7X Single Node Speedup

QUDA: Quantum Chromodynamics on CUDA
➢ Up to 1.4X Multi Node Speedup
MULTI GPU WITH THE NVIDIA HPC SDK

Cloverleaf Hydrodynamics Mini-App

Full Integration provided by HPC SDK

- Fortran + OpenACC + Open MPI

Strong Scaling - Cloverleaf BM128

- Perfect scaling to 4 A100 GPUs
- 7.5X speed-up on 8 A100 GPUs
COMPUTE DEVELOPER TOOLS

Nsight Systems
System-wide application algorithm tuning

Nsight Compute
CUDA Kernel Profiling and Debugging

Nsight Graphics
Graphics Shader Profiling and Debugging

IDE Plugins
Nsight Eclipse Edition/Visual Studio (Editor, Debugger)

cuda-gdb
CUDA Kernel Debugging

Compute Sanitizer
Memory, Race Checking

// Out of bounds Array Access

__global__ void oobAccess(int* in, int* out)
{
    int bid = blockIdx.x;
    int tid = threadIdx.x;
    if (bid == 4)
    {
        out[tid] = in[dMem[tid]];
    }
}

int main()
{
    ...
    // Array of 8 elements, where element 4 causes the OOB
    std::array<int, Size> hMem = {0, 1, 2, 10, 4, 5, 6, 7};
    cudaMemcpy(d_mem, hMem.data(), size, cudaMemcpyHostToDevice);
    oobAccess<<<10, Size>>>(d_in, d_out);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);
    cudaDeviceSynchronize();
    ...
}
Key Features:

- System-wide application algorithm tuning
  - Multi-process tree support
- Locate optimization opportunities
  - Visualize millions of events on a very fast GUI timeline
  - Or gaps of unused CPU and GPU time
- Balance your workload across multiple CPUs and GPUs
  - CPU algorithms, utilization and thread state
  - GPU streams, kernels, memory transfers, etc
- Command Line, Standalone, IDE Integration

OS: Linux (x86, **Power**, Arm SBSA, Tegra), Windows, MacOSX (host)

GPUs: Pascal+

NSIGHT COMPUTE 2020

Chips Update
A100 GPU Support

Advanced Analysis
Roofline
New Memory Tables

Workflow Improvements
Hot Spot Tables
Section Links

Other Changes
New Rules, Names

For more information see: S21771 - Optimizing CUDA kernels using Nsight Compute
Efficient way to evaluate kernel characteristics, quickly understand potential directions for further improvements or existing limiters
Next-Gen Replacement Tool for \texttt{cuda-memcheck}

Significant performance improvement of 2x - 5x compared with \texttt{cuda-memcheck} (depending on application size)

Performance gain for applications using libraries such as CUSOLVER, CUFFT or DL frameworks

\texttt{cuda-memcheck} still supported in CUDA 11.0 (does not support Arm SBSA)

\texttt{https://docs.nvidia.com/cuda/compute-sanitizer}
CUDA ON WINDOWS SUBSYSTEM FOR LINUX

Run a Linux kernel **natively** on top of Windows 10

Runs Linux at near full speed **without emulation**

**Multi-OS development** & testing from a single Windows desktop machine

**No need for dual-boot systems** - ideal for laptops
Get the latest version of Docker and run:

- AI Frameworks (PyTorch, TensorFlow)
- RAPIDS & ML Applications
- Jupyter Notebooks

GPU-enabled DirectX, CUDA 11.1 and the NVIDIA Container Toolkit are all available on WSL today.

NVML and NCCL support coming soon.

See CUDA-on-WSL blog for full details:
NEW FEATURES & IMPROVEMENTS IN CUDA 11
CUTLASS - TENSOR CORE PROGRAMMING MODEL
Warp-Level GEMM and Reusable Components for Linear Algebra Kernels in CUDA

CUTLASS 2.2
Optimal performance on NVIDIA Ampere microarchitecture
New floating-point types: nv_bfloat16, TF32, double
Deep software pipelines with async memcpy

CUTLASS 2.1
BLAS-style host API

CUTLASS 2.0
Significant refactoring using modern C++11 programming

For more information see: S21745 - Developing CUDA Kernels to Push Tensor Cores to the Absolute Limit
cuBLAS
Eliminating Alignment Requirements To Activate Tensor Cores for MMA

AlignN means alignment to 16-bit multiplies of N. For example, align8 are problems aligned to 128 bits or 16 bytes.
MATH LIBRARY DEVICE EXTENSIONS
Introducing cuFFTDx: Device Extension

Available in Math Library EA Program
Device callable library
Retain and reuse on-chip data
Inline FFTs in user kernels
Combine multiple FFT operations

https://developer.nvidia.com/CUDAMathLibraryEA
ISO C++ == Language + Standard Library
ISO C++ == Language + Standard Library

CUDA C++ == Language
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**
libcu++ NAMESPACE HIERARCHY

// ISO C++, __host__ only
#include <atomic>
std::atomic<int> x;

// CUDA C++, __host__ __device__
// Strictly conforming to the ISO C++
#include <cuda/std/atomic>
cuda::std::atomic<int> x;

// CUDA C++, __host__ __device__
// Conforming extensions to ISO C++
#include <cuda/atomic>
cuda::atomic<int, cuda::thread_scope_block> x;

For more information see: S21262 - The CUDA C++ Standard Library
CUB is now a fully-supported component of the CUDA Toolkit. Thrust integrates CUB’s high performance kernels.
CUB: CUDA UNBOUND
Reusable Software Components for Every Layer of the CUDA Programming Model

Device-wide primitives
Parallel sort, prefix scan, reduction, histogram, etc.
Compatible with CUDA dynamic parallelism

Block-wide "collective" primitives
Cooperative I/O, sort, scan, reduction, histogram, etc.
Compatible with arbitrary thread block sizes and types

Warp-wide "collective" primitives
Cooperative warp-wide prefix scan, reduction, etc.
Safely specialized for each underlying CUDA architecture
WARP-WIDE REDUCTION USING __shfl

```c
__device__ int reduce(int value) {
    value += __shfl_xor_sync(0xFFFFFFFF, value, 1);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 2);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 4);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 8);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 16);
    return value;
}
```
WARP-WIDE REDUCTION IN A SINGLE STEP

__device__ int reduce(int value) {
    value += __shfl_xor_sync(0xFFFFFFFF, value, 1);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 2);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 4);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 8);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 16);

    return value;
}

int total = __reduce_add_sync(0xFFFFFFFF, value);

Supported operations:
- add
- min
- max
- and
- or
- xor
WARP-WIDE REDUCTION IN A SINGLE STEP

__device__ int reduce(int value) {
    value += __shfl_xor_sync(0xFFFFFFFF, value, 1);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 2);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 4);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 8);
    value += __shfl_xor_sync(0xFFFFFFFF, value, 16);
    return value;
}

int total = __reduce_add_sync(0xFFFFFFFF, value);

thread_block_tile<32> tile32 =
    tiled_partition<32>(this_thread_block());

// Works on all GPUs back to Kepler
cg::reduce(tile32, value, cg::plus<int>());
**COOPERATIVE GROUPS**

Cooperative Groups Features Work On All GPU Architectures (incl. Kepler)

<table>
<thead>
<tr>
<th>Input Data</th>
<th>Global Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto tile32 = cg::tiled_partition&lt;32&gt;(this_thread_block());</td>
<td></td>
</tr>
<tr>
<td>cg::memcpy_async(tile32, dst, dstCount, src, srcCount);</td>
<td></td>
</tr>
<tr>
<td>cg::reduce(tile32, dst[threadRank], [](int lhs, int rhs) { return lhs + rhs; });</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Per-Tile Data</th>
<th>Per-Tile Data</th>
<th>Thread Block Shared Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>cg::reduce(tile32, dst[threadRank], [](int lhs, int rhs) { return lhs + rhs; });</td>
<td></td>
<td></td>
</tr>
<tr>
<td>cg::reduce also accepts C++ lambda as reduction operation</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Cooperative Groups Updates**

- No longer requires separate compilation
- 30% faster grid synchronization
- New platforms Support (Windows and Linux + MPS)
- Can now capture cooperative launches in a CUDA graph
ANATOMY OF A KERNEL LAUNCH

CUDA Kernel Launch

A<<< ..., s1 >>>( ... );
B<<< ..., s2 >>>( ... );
C<<< ..., s1 >>>( ... );
D<<< ..., s1 >>>( ... );

Stream Queues

<table>
<thead>
<tr>
<th>A</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>B</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Grid Management

Execution

Block A0
SM 0

Block A1
SM 1

Grid Completion
ANATOMY OF A GRAPH LAUNCH

cudaGraphLaunch(g1, s1);

Graph allows launch of multiple kernels in a **single operation**

Graph pushes multiple grids to Grid Management Unit allowing **low-latency dependency resolution**
A100 ACCELERATES GRAPH LAUNCH & EXECUTION

New A100 Execution Optimizations for Task Graphs

1. Grid launch latency reduction via whole-graph upload of grid & kernel data
2. Overhead reduction via accelerated dependency resolution
LATENCIES & OVERHEADS: GRAPHS vs. STREAMS
Empty Kernel Launches - Investigating System Overheads

Note: Empty kernel launches - timings show reduction in latency only
GRAPH PARAMETER UPDATE
Fast Parameter Update When Topology Does Not Change

Graph Update

Modify parameters without rebuilding graph
Change launch configuration, kernel parameters, memcopy args, etc.
Topology of graph may not change

Nearly 2x speedup on CPU
50% end-to-end overhead reduction

![Graph Update Diagram]

Effect of Graph Update on Performance

- Full Graph Creation: 1.0x
- Graph Update: 1.9x
- Graph Relaunch: 13.0x

- CPU Update + Launch: 2.5x
- End-to-End Overhead Reduction: 2.5x

Update Graph
launch graph
iterate
CUDA VIRTUAL MEMORY MANAGEMENT
Breaking Memory Allocation Into Its Constituent Parts

1. Reserve Virtual Address Range
   `cuMemAddressReserve/Free`

2. Allocate Physical Memory Pages
   `cuMemCreate/Release`

3. Map Pages To Virtual Addresses
   `cuMemMap/Unmap`

4. Manage Access Per-Device
   `cuMemSetAccess`

Control & reserve address ranges
Can remap physical memory
Fine-grained access control
Manage inter-GPU peer-to-peer sharing on a per-allocation basis
Inter-process sharing

REFERENCES
Deep dive into any of the topics you’ve seen by following these links

S21730  Inside the NVIDIA Ampere Architecture
        https://www.nvidia.com/nvidia-ampere-architecture-whitepaper
S22043  CUDA Developer Tools: Overview and Exciting New Features
S21975  Inside NVIDIA’s Multi-Instance GPU Feature
S21170  CUDA on NVIDIA GPU Ampere Architecture, Taking your algorithms to the next level of...
S21819  Optimizing Applications for NVIDIA Ampere GPU Architecture
S22082  Mixed-Precision Training of Neural Networks
S21681  How CUDA Math Libraries Can Help You Unleash the Power of the New NVIDIA A100 GPU
S21745  Developing CUDA Kernels to Push Tensor Cores to the Absolute Limit
S21766  Inside the NVIDIA HPC SDK: the Compilers, Libraries and Tools for Accelerated Computing
S21262  The CUDA C++ Standard Library
S21771  Optimizing CUDA kernels using Nsight Compute