CUDA 11: NEW FEATURES AND BEYOND

Stephen Jones, GTC 2020

NVIDIA
Huge breadth of platforms, systems, languages.
Weak Scaling
Larger Problem

Strong Scaling
Faster Solution
Mixed Scaling
Larger & Faster
HIERARCHY OF SCALES

Multi-System Rack
Unlimited Scale

Multi-GPU System
8 GPUs

Multi-SM GPU
108 Multiprocessors

Multi-Core SM
2048 threads
AMDAHL’S LAW

Program time = sum(serial times + parallel times)
AMDAHL’S LAW

Some Parallelism
Program time = \( \text{sum(serial times + parallel times)} \)

Increased Parallelism
Parallel sections take less time
Serial sections take same time

Time saved
AMDAHL’S LAW

Some Parallelism
Program time = \( \text{sum(serial times + parallel times)} \)

Increased Parallelism
Parallel sections take less time
Serial sections take same time

Infinite Parallelism
Parallel sections take no time
Serial sections take same time

Amdah’s Law
Shortest possible runtime is sum of serial section times
OVERCOMING AMDAHL: ASYNCHRONY & LATENCY

- **Task Parallelism**: Parallel sections overlap with serial sections. Program time = sum(serial times + parallel times).
- **Infinite Parallelism**: Parallel sections take no time, serial sections take same time.
- **Some Parallelism**: Parallel sections overlap with serial sections.
OVERCOMING AMDAHL: ASYNCHRONY & LATENCY

CUDA Concurrency Mechanisms At Every Scope

CUDA Kernel
- Threads, Warps, Blocks, Barriers

Application
- CUDA Streams, CUDA Graphs

Node
- Multi-Process Service, GPU-Direct

System
- NCCL, CUDA-Aware MPI, NVSHMEM
OVERCOMING AMDAHL: ASYNCHRONY & LATENCY

Execution Overheads
Non-productive latencies (waste)

Operation Latency
Network latencies
Memory read/write
File I/O
...

Execution Overheads are waste
Reduced through hardware & system efficiency improvements

Operation Latencies are the cost of doing work
Improve through hardware & software optimization
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
ANNOUNCING THE NVIDIA AMPERE GPU ARCHITECTURE

<table>
<thead>
<tr>
<th></th>
<th>V100</th>
<th>A100</th>
</tr>
</thead>
<tbody>
<tr>
<td>SMs</td>
<td>80</td>
<td>108</td>
</tr>
<tr>
<td>Tensor Core</td>
<td>FP16</td>
<td>FP64, TF32, BF16, FP16, I8, I4, B1</td>
</tr>
<tr>
<td>Precision</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Shared Memory</td>
<td>96 kB</td>
<td>160 kB</td>
</tr>
<tr>
<td>per Block</td>
<td></td>
<td></td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>6144 kB</td>
<td>40960 kB</td>
</tr>
<tr>
<td>Memory Bandwidth</td>
<td>900 GB/sec</td>
<td>1555 GB/sec</td>
</tr>
<tr>
<td>NVLink Interconnect</td>
<td>300 GB/sec</td>
<td>600 GB/sec</td>
</tr>
</tbody>
</table>

For more information see: [S21730 - Inside the NVIDIA Ampere Architecture](https://www.nvidia.com/nvidia-ampere-architecture-whitepaper)
ANNOUNCING THE NVIDIA AMPERE GPU ARCHITECTURE

NVIDIA GA100 Key Architectural Features

- Multi-Instance GPU
- Advanced barriers
- Asynchronous data movement
- L2 cache management
- Task graph acceleration
- New Tensor Core precisions
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
DATA CENTER GPU MANAGER (DCGM)
GPU Management in the Accelerated Data Center

Intended for
- Online monitoring of Data Center GPUs in production
- Production line testing/pre-production testing of servers

Active health monitoring | GPU Metrics
NVSwitch management | Comprehensive diagnostics
System alerts | Governance policies

Supports Data Center SKUs (Kepler+) on Linux x86_64, POWER architectures

https://developer.nvidia.com/dcgm
GPU & PLATFORM SUPPORT ACROSS DEVELOPER TOOLS

- **Chips Update**
  - A100 GPU Support

- **CUDA 11.0 support**

- **Arm SBSA support**

- **OS support updates**
  - POWER9 support
  - MacOSX host platform only
  - Removal of Windows 7 support

For more information see: [S22043 - CUDA Developer Tools: Overview and Exciting New Features](https://www.nvidia.com/content/S22043-CUDA-Developer-Tools-Overview-and-Exciting-New-Features)
NEW MULTI-INSTANCE GPU (MIG)

Divide a Single GPU Into Multiple Instances, Each With Isolated Paths Through the Entire Memory System

Up To 7 GPU Instances In a Single A100

Full software stack enabled on each instance, with dedicated SM, memory, L2 cache & bandwidth

Simultaneous Workload Execution With Guaranteed Quality Of Service

All MIG instances run in parallel with predictable throughput & latency, fault & error isolation

Diverse Deployment Environments

Supported with Bare metal, Docker, Kubernetes Pod, Virtualized Environments
LOGICAL VS. PHYSICAL PARTITIONING

Multi-Process Service
Dynamic contention for GPU resources
Single tenant

Multi-Instance GPU
Hierarchy of instances with guaranteed resource allocation
Multiple tenants
# CUDA CONCURRENCY MECHANISMS

<table>
<thead>
<tr>
<th>Partition Type</th>
<th>Streams</th>
<th>MPS</th>
<th>MIG</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partition Type</td>
<td>Single process</td>
<td>Logical</td>
<td>Physical</td>
</tr>
<tr>
<td>Max Partitions</td>
<td>Unlimited</td>
<td>48</td>
<td>7</td>
</tr>
<tr>
<td>Fractional Provisioning</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Protection</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Memory Bandwidth QoS</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Fault Isolation</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Cross-Partition Interop</td>
<td>Always</td>
<td>IPC</td>
<td>Limited IPC</td>
</tr>
<tr>
<td>Reconfigure</td>
<td>Dynamic</td>
<td>Process launch</td>
<td>When idle</td>
</tr>
</tbody>
</table>

For more information see: [S21975 - Inside NVIDIA's Multi-Instance GPU Feature](S21975-Inside-NVIDIAs-Multi-Instance-GPU-Feature)
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

CUDA VIRTUAL MEMORY MANAGEMENT

Basic Memory Allocation Example

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`

// Allocate memory
`cuMemCreate(&handle, size, &allocProps, 0);`

// Reserve address range
`cuMemAddressReserve(&ptr, size, alignment, fixedVa, 0);`

// Map memory to address range
`cuMemMap(ptr, size, offset, handle, 0);`

// Make the memory accessible on all devices
`cuMemSetAccess(ptr, size, rwOnDeviceSet, deviceCount);`
Pre-emptive scheduling
Processes share GPU through time-slicing
Scheduling managed by system

Concurrent scheduling
Processes run on GPU simultaneously
User creates & manages scheduling streams

$ nvidia-smi compute-policy
--set-timeslice={default, short, medium, long}

Time-slice configurable via nvidia-smi

CUDA 11.0 adds a new stream priority level

cudaStreamCreateWithPriority(pStream, flags, priority);
cudaDeviceGetStreamPriorityRange(leastPriority, greatestPriority);
FINE-GRAINED SYNCHRONIZATION

NVIDIA Ampere GPU Architecture Allows Creation Of Arbitrary Barriers
FINE-GRAINED SYNCHRONIZATION
NVIDIA Ampere GPU Architecture Allows Creation Of Arbitrary Barriers

__syncthreads()
**ASYNCHRONOUS BARRIERS**

**Single-Stage barriers** combine back-to-back arrive & wait

**Asynchronous barriers** enable pipelined processing
ASYNC MEMCOPY: DIRECT TRANSFER INTO SHARED MEMORY

Two step copy to shared memory via registers:
1. Thread loads data from GPU memory into registers
2. Thread stores data into SM shared memory

Asynchronous direct copy to shared memory:
1. Direct transfer into shared memory, bypassing thread resources
THIS FEATURE WILL CHANGE EVERY KERNEL THAT I WRITE
Free performance, fewer resources, cleaner code

All my programs look like this

Kernel Start
Prologue
Load Data
Sync
Compute
Exit

Now they will look like this

Kernel Start
Async Load Data
Prologue
Sync
Compute
Exit

Use fewer resources (reduced register pressure)
Increase occupancy
“Prologue” is now free
Enables Pipelined iteration with split barriers (see upcoming)
SIMPLE DATA MOVEMENT

1. Load image element into registers
SIMPLE DATA MOVEMENT

1. Load image element into registers
2. Store image element into shared memory
SIMPLE DATA MOVEMENT

1. Load image element into registers
2. Store image element into shared memory
3. Compute using shared memory data
SIMPLE DATA MOVEMENT

1. Load image element into registers
2. Store image element into shared memory
3. Compute using shared memory data
4. Repeat for next element
DOUBLE-BUFFERED DATA MOVEMENT

Shared Memory

Prefetch initial image element into registers
DOUBLE-BUFFERED DATA MOVEMENT

Shared Memory

1. Prefetch initial image element into registers
2. Prefetch next element into more registers
DOUBLE-BUFFERED DATA MOVEMENT

1. Prefetch initial image element into registers
2. Prefetch next element into more registers
3. Store current element into shared memory
DOUBLE-BUFFERED DATA MOVEMENT

1. Prefetch initial image element into registers
2. Prefetch next element into more registers
3. Store current element into shared memory
4. Compute using shared memory data
**DOUBLE-BUFFERED DATA MOVEMENT**

1. **P1** - Prefetch initial image element into registers
2. **1** - Prefetch next element into more registers
3. **2** - Store current element into shared memory
4. **3** - Compute using shared memory data
5. **4** - Repeat for next element
ASYNCHRONOUS DIRECT DATA MOVEMENT

Async copy initial element into shared memory
ASYNCHRONOUS DIRECT DATA MOVEMENT

- P1: Async copy initial element into shared memory
- 1: Async copy next element into shared memory
ASYNCHRONOUS DIRECT DATA MOVEMENT

1. **Async copy** initial element into shared memory
2. Threads **synchronize** with current async copy
3. Compute using shared memory data
4. Repeat for next element
ASYNCHRONOUS COPY PIPELINES
Prefetch multiple images in a continuous stream

For more information see: S21170 - CUDA on NVIDIA GPU Ampere Architecture, Taking your algorithms to the next level of performance
HIERARCHY OF LATENCIES

- GPU
  - SM
  - shmemb L1
  - L2
  - shmemb L1
  - HBM
  - shmemb L1
  - PCIe
  - DRAM
  - Network
  - CPU

Latencies:
- 1x
- 5x
- 15x
- 25x
- 50x
MANAGING LATENCY: L2 CACHE RESIDENCY CONTROL

<table>
<thead>
<tr>
<th>Latency</th>
<th>Shared Memory</th>
<th>L2 Cache</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bandwidth</td>
<td>1x</td>
<td>5x</td>
<td>15x</td>
</tr>
<tr>
<td></td>
<td>13x</td>
<td>3x</td>
<td>1x</td>
</tr>
</tbody>
</table>
MANAGING LATENCY: L2 CACHE RESIDENCY CONTROL

Latency & Bandwidth:
- Shared Memory: 1x Latency, 13x Bandwidth
- L2 Cache: 5x Latency, 3x Bandwidth
- GPU Memory: 15x Latency, 1x Bandwidth

L2 Cache Residency Control:
- Specify address range up to 128MB for persistent caching.
- Normal & streaming accesses cannot evict persistent data.
- Load/store from range persists in L2 even between kernel launches.
- Normal accesses can still use entire cache if no persistent data is present.
MANAGING LATENCY: L2 CACHE RESIDENCY CONTROL

Output Histogram

Normalized Histogram Construction Time

256 million items counted into 5 million histogram bins

For more information see: S21819 - Optimizing Applications for NVIDIA Ampere GPU Architecture
ANATOMY OF A KERNEL LAUNCH

CUDA Kernel Launch

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

Stream Queues

Grid Management

Execution

Grid Completion

Block A0
SM 0

Block A1
SM 1
ANATOMY OF A GRAPH LAUNCH

CUDA Graph Launch

A → B → C → D

cudaGraphLaunch(g1, s1);

Stream Queues

Grid Management

Execution

D → C → B → A

Block A0
SM 0

Block A1
SM 1

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

```c
cudaGraphLaunch(g1, s1);
```
LATENCIES & OVERHEADS: GRAPHS vs. STREAMS

Empty Kernel Launches - Investigating System Overheads

CPU Launch Speedup, Graphs vs. Streams
(32-Node graph, DGX-1V & DGX-A100)

- Straight line
- Single fork-join
- Repeated fork-join

Grid-to-Grid Latency Speedup
(32-Node graph, DGX-1V & DGX-A100)

- Straight line
- Single fork-join
- Repeated fork-join

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, memcpy args, etc.
Topology of graph may not change

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

Effect of Graph Update on Performance

<table>
<thead>
<tr>
<th></th>
<th>CPU Update + Launch</th>
<th>End-to-End Overhead Reduction</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full Graph Creation</td>
<td>1.0x</td>
<td>1.0x</td>
</tr>
<tr>
<td>Graph Update</td>
<td>1.9x</td>
<td>1.5x</td>
</tr>
<tr>
<td>Graph Relaunch</td>
<td>13.0x</td>
<td>2.5x</td>
</tr>
</tbody>
</table>
FLOATING POINT FORMATS & PRECISION

value = \((-1)^{\text{sign}} \times 2^{\text{exponent}} \times (1 + \text{mantissa})\)
NEW Floating Point Formats: BF16 & TF32
Both Match fp32 8-bit Exponent: Covers The Same Range of Values

**bfloat16**
- **8-bit Exponent**: Covers the same range of values as 8-bit exponent in fp32.
- **7-bit Fraction**: Available in CUDA C++ as `nv_bfloat16` numerical type.
- Full CUDA C++ numerical type – `#include <cuda_fp16.h>`
- Can use in both host & device code, and in templated functions*
- **16-bit Storage Size**

**TF32**
- **8-bit Exponent**: Tensor Core math mode for single-precision training.
- **10-bit Fraction**: Not a numerical type – tensor core inputs are rounded to TF32.
- **32-bit Storage Size**

*Available in CUDA C++ as `nv_bfloat16` numerical type
Full CUDA C++ numerical type – `#include <cuda_fp16.h>`
Can use in both host & device code, and in templated functions*

Tensor Core math mode for single-precision training
Not a numerical type – tensor core inputs are rounded to TF32
CUDA C++ programs use `float` (fp32) throughout

*(similar to CUDA's IEEE-FP16 “half” type)
TENSOR FLOAT 32 - TENSOR CORE MODE

A100 Tensor Core Input Precision
All Internal Operations Maintain Full FP32 Precision

Convert to TF32

FP32 → Full precision product

FP32 →

x

more products

Sum with FP32 accumulator → FP32 output

TF32 MMA Dimensions: m,n,k = 16x8x8

For more information see: S22082: Mixed-Precision Training of Neural Networks
A100 INTRODUCES DOUBLE PRECISION TENSOR CORES
All A100 Tensor Core Internal Operations Maintain Full FP64 Precision

DMMA Dimensions: m,n,k = 8x8x4

NVIDIA V100 FP64
NVIDIA A100 Tensor Core FP64

A100 Peak Double Precision Performance

<table>
<thead>
<tr>
<th></th>
<th>FP64 TFLOPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>V100</td>
<td>7.8</td>
</tr>
<tr>
<td>A100 Arithmetic</td>
<td>9.7</td>
</tr>
<tr>
<td>A100 DMMA</td>
<td>19.5</td>
</tr>
</tbody>
</table>
A100 GPU ACCELERATED MATH LIBRARIES IN CUDA 11.0

- cuBLAS: BF16, TF32 and FP64 Tensor Cores
- cuSPARSE: Increased memory BW, Shared Memory & L2
- cuTENSOR: BF16, TF32 and FP64 Tensor Cores
- cuSOLVER: BF16, TF32 and FP64 Tensor Cores
- nvJPEG: Hardware Decoder
- cuFFT: BF16, TF32 and FP64 Tensor Cores
- CUDA Math API: Increased memory BW, Shared Memory & L2
- CUTLASS: BF16 & TF32 Support

For more information see: S21681 - How CUDA Math Libraries Can Help You Unleash the Power of the New NVIDIA A100 GPU
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 memcopy

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

Mixed-Precision FP16/FP32 Tensor Core Accelerated Matrix Multiply Improvements on V100

AlignN means alignment to 16-bit multiplies of N. For example, align8 are problems aligned to 128bits or 16 bytes.
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
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

```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;
}
```

```c
int total = __reduce_add_sync(0xFFFFFFFF, value);
```
__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)

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

cg::reduce also accepts C++ lambda as reduction operation

```
auto tile32 =
    cg::tiled_partition<32>(this_thread_block());

cg::memcpy_async(tile32, dst, dstCount, src, srcCount);

cg::reduce(tile32, dst[threadRank], [] (int lhs, int rhs) {
    return lhs + rhs;
});
```
GPU PROGRAMMING IN 2020 AND BEYOND

Math Libraries | Standard Languages | Directives | CUDA

Incremental Performance Optimization with Directives

Maximize GPU Performance with CUDA C++/Fortran

GPU Accelerated Math Libraries

For more information see: S21766 - Inside the NVIDIA HPC SDK: the Compilers, Libraries and Tools for Accelerated Computing
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
CUDA C++ HETEROGENEOUS ARCHITECTURE

Thrust
Host code Standard Library-inspired primitives
e.g: for_each, sort, reduce

CUB
Re-usable building blocks, targeting 3 layers of abstraction

libcu++
Heterogeneous ISO 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
# NVCC HIGHLIGHTS IN CUDA 11.0 TOOLKIT

## Key Features

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

- **whole.cu**
  - `x();`
  - `y();`

  ![Diagram of Whole-Program Compilation](image)

- **cicc**
- **ptxas**
- **Executable**

### Separate Compilation

- **a.cu**
  - `x();`

- **b.cu**
  - `y();`

- **cicc**
- **ptxas**
- **nvlink**
- **Executable**

**ABI calls incur call overheads**

- **x() → y()**

---

**Notice:**

- `<file>.c` is compiled with `cc`
- `<file>.cu` is compiled with `nvcc` (Cuda)
- `<file>.ptx` is compiled using `ptxas`
- `<file>.o` is compiled using `icc`
- `<file>.exe` is compiled using `nvlink`
- `<file>.pgi` is compiled using `pgcc`
- `<file>.dll` is compiled using `link`
- `<file>.so` is compiled using `gcc`
- `<file>.lib` is compiled using `gmake`
**LINK-TIME OPTIMIZATION**

Whole-Program Compilation

- **Executable**
  - **pxas**
  - **ptxas**
  - **cicc**
  - **whole.cu**
    - x(); y();

Link-Time Optimization

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

```
whole.cu
x(); y();
```

```
a.cu
x();
b.cu
y();
```
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
NSIGHT COMPUTE 2020.1

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
NSIGHT COMPUTE 2020.1
New Roofline Analysis

Efficient way to evaluate kernel characteristics, quickly understand potential directions for further improvements or existing limiters

Inputs
- Arithmetic Intensity (FLOPS/bytes)
- Performance (FLOPS/s)

Ceilings
- Peak Memory Bandwidth
- Peak FP32/FP64 Performance
Next-Gen Replacement Tool for CUDA-memcheck

Significant performance improvement of 2x - 5x compared with CUDA-memcheck (depending on application size)

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

cuda-memcheck still supported in CUDA 11.0 (does not support Arm SBSA)

https://docs.nvidia.com/cuda/compute-sanitizer
CUDA 11.0: AVAILABLE FOR DOWNLOAD SOON

Hierarchy

Asynchrony

Latency

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

S21730  Inside the NVIDIA Ampere Architecture
Whitepaper  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