CUDA 11 AND A100 - WHAT’S NEW?

Markus Hrywniak, 23rd June 2020
TOPICS FOR TODAY

Ampere architecture - A100, powering DGX-A100, HGX-A100... and soon, FZ Jülich‘s JUWELS Booster

New CUDA 11 Toolkit release

Overview of features

Talk next week: Third generation Tensor Cores

GTC talks go into much more details. See references!
• 4 A100 with NVLINK

• 8 A100 with NVSwitch
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

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

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

Some Parallelism
Program time = sum(serial times + parallel times)

Task Parallelism
Parallel sections overlap with serial sections

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

For more information see: S21730 - Inside the NVIDIA Ampere Architecture & www.nvidia.com/nvidia-ampere-architecture-whitepaper
A100 TENSOR-CORE GPU

54 billion transistors in 7 nm

108 SMs
6912 CUDA Cores

40 MB L2
6.7x capacity

1.56 TB/s HBM2
1.7x bandwidth

Scale UP

Scale OUT
Third-generation Tensor Core

- Faster and more efficient
- Comprehensive data types
- FP64 support
- Sparsity acceleration

Asynchronous data movement and synchronization

Increased L1/SMEM capacity
ACCELERATING HPC

**Molecular Dynamics**
- AMBER based on PME
- Cellulose
- GROMACS with STMV (h-bond)
- LAMMPS with Atomic Fluid LJ-2.5
- NAMD with v3.0a1 STMV_NVE

**Physics**
- Chroma with szscl24_24_128
- FUN3D with dpw
- RTM with Isotropic Radius 4 1024^3
- SPECFEM3D with Cartesian four material model

**Engineering**
- BerkeleyGW based on Chi Sum

**Geo Science**
- SPECFEM3D

All results are measured

Except BerkeleyGW, V100 used is single V100 SXM2, A100 used is single A100 SXM4

More apps detail:
- AMBER based on PME-Cellulose, GROMACS with STMV (h-bond), LAMMPS with Atomic Fluid LJ-2.5, NAMD with v3.0a1 STMV_NVE
- Chroma with szscl24_24_128, FUN3D with dpw, RTM with Isotropic Radius 4 1024^3, SPECFEM3D with Cartesian four material model

BerkeleyGW based on Chi Sum and uses 8xV100 in DGX-1, vs 8xA100 in DGX-A100
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
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
ASYNC COPY
Asynchronous load + store in shared Memory

Typical way of using shared memory:

```cpp
__shared__ int smem[1024];
smem[threadIdx.x] = input[index];
```

- Wasting registers
- Stalling while the data is loaded
- Wasting L1/SHM bandwidth

LDG.E.SYS R0, [R2] ;
STS [R5], R0 ;

*STALL*
ASYNC COPY
Asynchronous load + store in shared Memory

```c
__shared__ int smem[1024];
pipeline_memcpy_async(&smem[threadIdx.x], &input[index], sizeof(int));
pipeline_commit();
pipeline_wait_prior(0);
```

Copies the data straight to shared memory asynchronously with 2 possible paths:

- L1 Access (Data gets Cached in L1)
- L1 Bypass (No L1 Caching, 16-Byte vector LDGSTS)

Very flexible scheduling (e.g. multi-stage)

For more details: S21170 (Carter Edwards)
ASYNCHRONOUS BARRIERS

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

Asynchronous barriers enable pipelined processing
__device__ void memcpy_example()
{
  __shared__ barrier b1, b2;
  // initialization omitted
  cudaMemcpyAsync(/* ... */ , b1);
  cudaMemcpyAsync(/* ... */ , b2);
  b1.arrive_and_wait();
  compute();
  b2.arrive_and_wait();
  compute();
}

__device__ void split_barrier_example()
{
  __shared__ barrier b1, b2;
  // initialization omitted
  compute_head(part_one);
  auto t1 = b1.arrive();
  compute_head(part_two);
  auto t2 = b2.arrive();
  b1.wait(t1);
  compute_tail(part_one);
  b2.wait(t2);
  compute_tail(part_two);
}
HIERARCHY OF LATENCIES
MANAGING LATENCY: L2 CACHE RESIDENCY CONTROL

<table>
<thead>
<tr>
<th>Latency</th>
<th>L2 Cache</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x</td>
<td>5x</td>
<td>15x</td>
</tr>
<tr>
<td>13x</td>
<td>3x</td>
<td>1x</td>
</tr>
</tbody>
</table>

Bandwidth

- Shared Memory
- L1
- SM
- HBM
- GPU Memory
MANAGING LATENCY: L2 CACHE RESIDENCY CONTROL

<table>
<thead>
<tr>
<th>Latency Bandwidth</th>
<th>Shared Memory</th>
<th>L2 Cache</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>1x 13x</td>
<td>5x 3x</td>
<td>15x 1x</td>
<td></td>
</tr>
</tbody>
</table>

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
TUNING FOR L2 CACHE

Global Memory Histogram

More frequently accessed histogram bins stay pinned in L2. Increases hit rate for global memory atomics

```c
__global__ void histogram(
    int *hist, int *data, int nbins)
{
    int tid = blockIdx.x * blockDim.x
               + threadIdx.x;
    int bin_id = data[tid];
    // Performing atomics in global memory
    atomicAdd(hist + bin_id, 1);
}
```
Global memory region can be marked for persistence access using `accessPolicyWindow`.

Subsequent kernel launches in the stream or Cuda graph have persistence property on the marked data region.

```c
cudaStreamAttrValue attribute;
auto &window = attribute.accessPolicyWindow;

window.base_ptr = data_ptr;
window.num_bytes = num_bytes;
window.hitRatio = 1.0;
window.hitProp = cudaAccessPropertyPersisting;
window.missProp = cudaAccessPropertyStreaming;

cudaStreamSetAttribute(stream,
cudaStreamAttributeAccessPolicyWindow,
&attribute);
cuda_kernel<<<grid_size,block_size,0,stream>>>(data_ptr);
```

For more detailed API: S21170 (Carter Edwards)
TUNING FOR L2 CACHE

Global Memory Histogram

Dataset Size = 1024 MB* (256 Million integers)

Size of Persistent Histogram bins = 20 MB* (5 Million integer bins)

For more information see: S21819 - Optimizing for NVIDIA Ampere
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
A100 STRONG SCALING INNOVATIONS
Delivering unprecedented levels of performance

A100 improvements over V100

- Tensor Core math BW (FP16): 2.5x
- Effective RF BW with A100 Tensor Core: 2.9x
- Effective RF capacity with Async-Copy bypassing RF: 2.8x
- Effective SMEM BW with A100 Tensor Core and Async-Copy: 3.0x
- SMEM capacity: 2.3x
- L2 BW: 2.3x
- L2 capacity, +Residency Control: 6.7x
- DRAM BW: 1.7x
- DRAM capacity: 1.3x
- NVLINK BW: 2.0x

- Sparse Tensor Core (FP16): 5.0x
- Compute Data Compression (max): 9.2x
- L2 capacity, +Residency Control: 13.3x
- DRAM BW: 6.8x
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](https://www.nvidia.com/research/cuda/docs/S21681)
THE NVIDIA HPC SDK
Apply now at developer.nvidia.com/hpc-sdk

NVIDIA HPC SDK

Develop for the NVIDIA HPC Platform: GPU, CPU and Interconnect
HPC Libraries | GPU Accelerated C++ and Fortran | Directives | CUDA
Compatible with HPC Container Maker and 99% of Top500 Systems
cuSOLVER
DENSE LINEAR ALGEBRA PERFORMANCE ON THE NEW NVIDIA A100 & DGX-A100™

Results comparing CUDA 11.0 cuSOLVER NVIDIA A100 to CUDA 10.2 on V100.

Multi-GPU Dense Symmetric Eigen solver (LAPACK: DSYEVD)

2.4X Speed-up per GPU performance
GPU PROGRAMMING IN 2020 AND BEYOND
Math Libraries | Standard Languages | Directives | CUDA

GPU Accelerated C++ and Fortran

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
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::**

<table>
<thead>
<tr>
<th>Opt-in</th>
<th>Does not interfere with or replace your host standard library</th>
</tr>
</thead>
<tbody>
<tr>
<td>Heterogeneous</td>
<td>Copyable/Movable objects can migrate between host &amp; device</td>
</tr>
<tr>
<td></td>
<td>Host &amp; Device can call all member functions</td>
</tr>
<tr>
<td></td>
<td>Host &amp; Device can concurrently use synchronization primitives*</td>
</tr>
<tr>
<td>Incremental</td>
<td>A subset of the standard library today</td>
</tr>
<tr>
<td></td>
<td>Each release adds more functionality</td>
</tr>
</tbody>
</table>

*Synchronization primitives must be in managed memory and be declared with `cuda::std::thread_scope_system`*
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
HPC PROGRAMMING IN ISO C++

C++ Parallel Algorithms

- Introduced in C++17
- Parallel and vector concurrency via execution policies
  - `std::execution::par`, `std::execution::par_seq`, `std::execution::seq`
- Several new algorithms in C++17 including
  - `std::for_each_n(POLICY, first, size, func)`
- Insert `std::execution::par` as first parameter when calling algorithms
- **NVC++ 20.5**: automatic GPU acceleration of C++17 parallel algorithms
  - Leverages CUDA Unified Memory

```cpp
std::sort(std::execution::par, c.begin(), c.end());
std::unique(std::execution::par, c.begin(), c.end());
```
C++ PARALLEL ALGORITHMS

Lulesh Hydrodynamics Mini-app

- ~9000 lines of C++
- Parallel versions in MPI, OpenMP, OpenACC, CUDA, RAJA, Kokkos, ...
- Designed to stress compiler vectorization, parallel overheads, on-node parallelism

https://computing.llnl.gov/projects/co-design/lulesh
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 / (std::abs(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;
}
#endif
}

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; },
    [s, &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));
        }
    });
}

---

**PARALLEL C++**

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

---

**Parallel C++17**
LULESH PERFORMANCE

Speedup - Higher is Better

Same ISO C++ Code

- C++ on 2s 20c Xeon Gold 6148
- C++ on A100
- OpenACC 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
HPC PROGRAMMING IN ISO FORTRAN

NVFORTRAN Accelerates Fortran Intrinsics with cuTENSOR Backend

```
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,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, ni
      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
  !$acc end kernels
end do
!$acc exit data copyout(d)
```

```
real(8), dimension(ni,nk) :: a
real(8), dimension(nk,nj) :: b
real(8), dimension(ni,nj) :: c, d
...
!$acc enter data copyin(a,b,c) create(d)
...
!$acc host_data use_device(a,b,c,d)
do nt = 1, ntimes
  d = c + matmul(a,b)
end do
!$acc end host_data
...
!$acc exit data copyout(d)
```

Inline FP64 matrix multiply

MATMUL FP64 matrix multiply
NSIGHT COMPUTE 2020.1

New Roofline Analysis

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

CUDA 11 and Ampere key architecture improvements go hand-in-hand
Huge performance improvement (raw compute, automatic gains through libraries)
New programming model improvements (asynchrony)
More focus on modern C++, standard libraries
HPC SDK as focused distribution of compilers/libraries
REFERENCES AND FURTHER DETAILS

nvidia.com/nvidia-ampere-architecture-whitepaper

GTC talks, also check their references:

S21730: Inside the NVIDIA Ampere Architecture

S21760: CUDA New Features And Beyond

S21766: Inside the NVIDIA HPC SDK
THANK YOU