GPU programming for DL

Julie Bernauer and Ryan Olson
Outline
Presentation & Hands-on session

• Intro to GPU computing / Libraries for DL /Platform
• Intro to CUDA
• Hands-on labs
  • Accelerating Applications with CUDA C/C++
  • (optional) Accelerating Applications with GPU-Accelerated Libraries in C/C++
  • (optional) GPU Memory Optimizations (C/C++)
GPU Computing
GPU Computing
CUDA

Framework to Program NVIDIA GPUs

A simple sum of two vectors (arrays) in C

```c
void vector_add(int n, const float *a, const float *b, float *c)
{
    for(int idx = 0; idx < n; ++idx)
        c[idx] = a[idx] + b[idx];
}
```

GPU friendly version in CUDA

```c
__global__ void vector_add(int n, const float *a, const float *b, float *c)
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    if( idx < n )
        c[idx] = a[idx] + b[idx];
}
```
### GPU accelerated libraries

**“Drop-in” Acceleration for Your Applications**

<table>
<thead>
<tr>
<th>Linear Algebra</th>
<th>Numerical &amp; Math</th>
<th>Data Struct. &amp; AI</th>
<th>Visual Processing</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT, BLAS, SPARSE, Matrix, cuSolver</td>
<td>RAND, Statistics</td>
<td>Sort, Scan, Zero Sum</td>
<td>Image &amp; Video</td>
</tr>
</tbody>
</table>

**Linear Algebra**: NVIDIA cuFFT, cuBLAS, cuSPARSE

**Numerical & Math**: NVIDIA Math Lib

**Data Struct. & AI**: GPU AI - Board, GPU AI - Path Finding

**Visual Processing**: NVIDIA NPP, NVIDIA Video Encode, Sundog Software
Deep Neural Networks and GPUs
“Now You Can Build Google’s $1M Artificial Brain on the Cheap”

Deep learning with COTS HPC systems, A. Coates, B. Huval, T. Wang, D. Wu, A. Ng, B. Catanzaro  ICML 2013
Recent improvements

**Image Recognition**
- **IMAGENET**
  - Source: ImageNet
  - NVIDIA GPU

**Pedestrian Detection**
- **CALTECH**
  - CV-based
  - DNN-based
  - Source: CalTech
  - 2013: 80%, 2014: 85%, 2015: 90%, 2016: 93%

**Object Detection**
- **KITTI**
  - Top Score
  - Source: KITTI
  - NVIDIA DRIVEnet

**Image Recognition**
- **IMAGENET**
  - NVIDIA GPU

**Object Detection**
- **KITTI**
  - Top Score
  - Source: KITTI
  - NVIDIA DRIVEnet
NVIDIA cuDNN

Building blocks for accelerating deep neural networks on GPUs

- High performance deep neural network training
- Accelerates Deep Learning: Caffe, CNTK, Tensorflow, Theano, Torch
- Performance continues to improve over time

“NVIDIA has improved the speed of cuDNN with each release while extending the interface to more operations and devices at the same time.”

— Evan Shelhamer, Lead Caffe Developer, UC Berkeley

developer.nvidia.com/cudnn
Accelerating linear algebra: cuBLAS

Accelerated Level 3 BLAS
- GEMM, SYMM, TRSM, SYRK
- >3 TFlops Single Precision on a single K40

Multi-GPU BLAS support available in cuBLAS-XT

developer.nvidia.com/cublas
Accelerating sparse operations: cuSPARSE

The (Dense matrix) X (sparse vector) example

cusparse<T>gemvi()

\[
y = \alpha \ast \text{op}(A) \ast x + \beta \ast y
\]

A = dense matrix
x = sparse vector
y = dense vector

Sparse vector could be frequencies of words in a text sample

cuSPARSE provides a full suite of accelerated sparse matrix functions

developer.nvidia.com/cusparse
Multi-GPU communication: NCCL
Collective library

• Research library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

• Pattern the library after MPI’s collectives
• Handle the intra-node communication in an optimal way
• Provide the necessary functionality for MPI to build on top to handle inter-node

github.com/NVIDIA/nccl
NCCL Example

All-reduce

#include <nccl.h>
ncclComm_t comm[4];
ncclCommInitAll(comm, 4, {0, 1, 2, 3});
foreach g in (GPUs) { // or foreach thread
    cudaSetDevice(g);
    double *d_send, *d_recv;
    // allocate d_send, d_recv; fill d_send with data
    ncclAllReduce(d_send, d_recv, N, ncclDouble, ncclSum, comm[g], stream[g]);
    // consume d_recv
}
Developer workstation
Titan X Pascal

- 11 TFLOPS FP32
- INT8
- 3,584 CUDA
- 12 GB DDR5X
DGX-1
World’s First Deep Learning Supercomputer

Engineered for deep learning
170 TF FP16
8x Tesla P100 in hybrid cube mesh
+ SDD RAID, 4x IB
Accelerates major AI frameworks

nvidia.com/dgx1
## Tesla p100 accelerator

<table>
<thead>
<tr>
<th>Feature</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute</td>
<td>5.3 TF DP · 10.6 TF SP · 21.2 TF HP</td>
</tr>
<tr>
<td>Memory</td>
<td>HBM2: 720 GB/s · 16 GB</td>
</tr>
<tr>
<td>Interconnect</td>
<td>NVLink (up to 8 way) + PCIe Gen3</td>
</tr>
</tbody>
</table>
| Programmability  | Page Migration Engine  
Unified Memory                                       |
EXAMPLE: DL EMBEDDED DEPLOYMENT

Jetson TX1 devkit

- Jetson TX1
  - Inference at 258 img/s
  - No need to change code
- Simply compile Caffe and copy a trained .caffemodel to TX1
GPU INFERENC ENGINE

Optimizations

- Fuse network layers
- Eliminate concatenation layers
- Kernel specialization
- Auto-tuning for target platform
- Select optimal tensor layout
- Batch size tuning

See the parallel for all blog post for GIE:
GPU INference Engine

Performance

GIE + GPU vs. Caffe + GPU
10 Most Time Consuming Caffe Kernels (GoogLeNet)
GPU architecture
GPU ARCHITECTURE

Two Main Components

- Global memory
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 24 GB
  - ECC on/off options for Quadro and Tesla products

- Streaming Multiprocessors (SM)
  - Perform the actual computation
  - Each SM has its own: Control units, registers, execution pipelines, caches
GPU ARCHITECTURE

Streaming Multiprocessor (SM)

- Many CUDA Cores per SM
  - Architecture dependent
- Special-function units
  - cos/sin/tan, etc.
- Shared mem + L1 cache
- Thousands of 32-bit registers
GPU MEMORY HIERARCHY REVIEW

- **SM-0**
  - Registers
  - L1
  - SMEM

- **SM-1**
  - Registers
  - L1
  - SMEM

- **SM-N**
  - Registers
  - L1
  - SMEM

- **L2**

- **Global Memory**
CUDA Programming model
ANATOMY OF A CUDA C/C++ APPLICATION

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements
CUDA C : C WITH A FEW KEYWORDS

```c
void saxpy_serial(int n, float a, float *x, float *y) {
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);

__global__ void saxpy_parallel(int n, float a, float *x, float *y) {
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n) y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nbblocks, 256>>>(n, 2.0, x, y);
```

Standard C Code

Parallel C Code
CUDA KERNELS

- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads

- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

<table>
<thead>
<tr>
<th>CPU</th>
<th>Host</th>
<th>Executes functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>Device</td>
<td>Executes kernels</td>
</tr>
</tbody>
</table>
CUDA KERNELS: PARALLEL THREADS

- A **kernel** is a function executed on the GPU as an array of threads in parallel.

- All threads execute the same code, can take different paths.

- Each thread has an ID:
  - Select input/output data
  - Control decisions

```c
float x = input[threadIdx.x];
float y = func(x);
output[threadIdx.x] = y;
```
CUDA Kernels: Subdivide into Blocks
CUDA Kernels: Subdivide into Blocks

Threads are grouped into blocks
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into **blocks**
- **Blocks** are grouped into a grid
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads
Kernel Execution

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

CUDA-thread
- Each thread is executed by a core
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks’ memory requirements and the SM’s memory resources
- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time
Thread blocks allow cooperation

- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other
THREAD BLOCKS ALLOW SCALABILITY

- Blocks can execute in any order, concurrently or sequentially.
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs.
Memory System Hierarchy
MEMORY HIERARCHY

- Thread:
  - Registers
MEMORY HIERARCHY

Thread:
- Registers
- Local memory
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory

- Block of threads:
  - Shared memory
MEMORY HIERARCHY : SHARED MEMORY

```c
__shared__ int a[SIZE];
```

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing gmem accesses)
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory

- Block of threads:
  - Shared memory

- All blocks:
  - Global memory
MEMORY HIERARCHY : GLOBAL MEMORY

- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemcpy (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)
CUDA memory management
MEMORY SPACES

CPU and GPU have separate memory spaces

- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU just like standard C

Pointers are just addresses

- Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must use cudaPointerGetAttributes(...)
- Must exercise care when dereferencing:
  - Dereferencing CPU pointer on GPU will likely crash
  - Dereferencing GPU pointer on CPU will likely crash
GPU MEMORY ALLOCATION / RELEASE

Host (CPU) manages device (GPU) memory

- cudaMemcpy (void ** pointer, size_t nbytes)
- cudaMemcpy (void * pointer, int value, size_t count)
- cudaMemcpy (void* pointer)

```c
int n = 1024;

int nbytes = 1024*sizeof(int);
int * d_a = 0;

cudaMalloc( (void**)d_a, nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Note: Device memory from GPU point of view is also referred to as global memory.
DATA COPIES

cudaMemcpy( void *dst, void *src, size_t nbytes, enum cudaMemcpyKind direction);

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided
Basic kernels and execution
CUDA PROGRAMMING MODEL REVISITED

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables
Threads launched for a parallel section are partitioned into thread blocks

- Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory
IDS AND DIMENSIONS

Threads
- 3D IDs, unique within a block

Blocks
- 2D IDs, unique within a grid

Dimensions set at launch time
- Can be unique for each grid

Built-in variables
- threadIdx, blockIdx
- blockDim, gridDim

(Continued)
IDS AND DIMENSIONS

Threads
- 3D IDs, unique within a block

Blocks
- 2D IDs, unique within a grid

Dimensions set at launch time
- Can be unique for each grid

Built-in variables
- threadIdx, blockIdx
- blockDim, gridDim
LAUNCHING KERNELS ON GPU

Launch parameters (triple chevron <<<>>> notation)

- grid dimensions (up to 2D), dim3 type
- thread-block dimensions (up to 3D), dim3 type
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

Examples:

```c
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernl1lss<32, 512>>>(...);
```
GPU KERNEL EXECUTION

- Kernel launches on a grid of blocks, `<<<grid,block>>>`(arg1,...)
- Each block is launched on one SM
  - A block is divided into warps of 32 threads each (think 32-way vector)
  - Warps in a block are scheduled and executed.
    - All threads in a warp execute same instruction simultaneously (think SIMD)
  - Number of blocks/SM determined by resources required by the block
    - Registers, shared memory, total warps, etc.
- Block runs to completion on SM it started on, no migration.
Any possible interleaving of blocks should be valid

- presumed to run to completion without pre-emption
- can run in any order
- can run concurrently OR sequentially

Blocks may coordinate but not synchronize

- shared queue pointer: OK
- shared lock: BAD ... any dependence on order easily deadlocks

Independence requirement gives scalability
Hands-on labs
Prepare and Start AWS Instance

• Open a browser, go to nvlabs.qwiklab.com
  • Register (it’s free) and Sign in.
  • Select the correct lab (Montreal GPU Programming Workshop) and once enabled press “Start Lab”
  • Instance can take up to 10 minutes to start.
• Three labs are available:
  • Accelerating Applications with CUDA C/C++
  • (optional) Accelerating Applications with GPU-Accelerated Libraries in C/C++
  • (optional) GPU Memory Optimizations (C/C++)
Wrap up
Software

• GPU Driver
• CUDA toolkit
  • Includes all the software necessary for developers to write applications
    • Compiler (nvcc), libraries, profiler, debugger, documentation
• CUDA Samples
  • Samples illustrating GPU functionality and performance
  • Examples illustrating important programming constructs and techniques.
• www.nvidia.com/getcuda  -- all above software is free
Want to try?

Links and resources


Hands-on labs https://nvidia.qwiklab.com/

Question? Email jbernauer@nvidia.com
COME DO YOUR LIFE’S WORK
JOIN NVIDIA

We are looking for great people at all levels to help us accelerate the next wave of AI-driven computing in Research, Engineering, and Sales and Marketing.

Our work opens up new universes to explore, enables amazing creativity and discovery, and powers what were once science fiction inventions like artificial intelligence and autonomous cars.

Check out our career opportunities:

• www.nvidia.com/careers
• Reach out to your NVIDIA social network or NVIDIA recruiter at DeepLearningRecruiting@nvidia.com