HOW GPU ACCELERATION WORKS

Application Code

Compute-Intensive Functions

5% of Code

Rest of Sequential CPU Code

GPU + CPU
TAKE ADVANTAGE OF GPUS

- Use a GPU-accelerated applications
- Add calls to GPU-accelerated libraries
- Add OpenACC directives
- Write GPU-accelerated code by hand
Developer Platform With Open Ecosystem

Accelerate Applications Across Multiple CPUs

Libraries
- AmgX
- cuDNN
- cuBLAS
- OpenCV
- Thrust

Compiler Directives
- OpenACC

Programming Languages
- C/C+
- Fortran
- Python
- Java

x86
GPU ACCELERATED APPLICATIONS
WHY USE A GPU ACCELERATED APPLICATION?

- Already using it on CPUs
- Less work than writing code
- Not practical on CPUs
70% OF TOP HPC APPS ACCELERATED

INTERSECT360 SURVEY OF TOP APPS

9 of top 10 Apps Accelerated
35 of top 50 Apps Accelerated

TOP 25 APPS IN SURVEY

GROMACS
SIMULIA Abaqus
AMBER
NAMD
ANSYS Mechanical
Abaqus
Exelis IDL
LAMMPS
MSC NASTRAN

= All popular functions accelerated

= Some popular functions accelerated

= In development

= Not supported

Intersect360, Nov 2015
“HPC Application Support for GPU Computing”
GPU-ACCELERATED QUANTUM CHEMISTRY APPS

- Abinit
- ACES III
- ADF
- BigDFT
- CP2K
- GAMESS-US
- Gaussian
- GPAW
- LATTE
- LSDalton
- MOLCAS
- Mopac2012
- NWChem
- Octopus
- ONETEP
- Petot
- Q-Chem
- QMCPACK
- Quantum Espresso
- Quantum SuperCharger Library
- RMG
- TeraChem
- UNM
- VASP
- WL-LSMS
GPU-ACCELERATED MOLECULAR DYNAMICS APPS

- ACEMD
- AMBER
- CHARMM
- DESMOND
- ESPResSO
- Folding@Home
- GPUGrid.net
- GROMACS
- HALMD
- HOOMD-Blue
- LAMMPS
- mdcore
- MELD
- NAMD
- OpenMM
- PolyFTS
GPU-ACCELERATED DEEP LEARNING FRAMEWORKS

Caffe

torch

theano

TensorFlow

dmlc

mxnet

K

Microsoft CNTK

Chainer
THE GPU APPLICATIONS CATALOG

Over 400 applications listed in 11 categories
• Computational Finance
• Climate, Weather, and Ocean Modeling
• Data Science & Analytics
• Deep Learning and Machine Learning
• Federal Defense and Intelligence
• Manufacturing/AEC: CAD and XCAE
• Media and Entertainment
• Medical Imaging
• Oil and Gas
• Research: Higher Education and Supercomputing
• Safety & Security

https://www.nvidia.com/content/gpu-applications/PDF/gpu-applications-catalog.pdf
GROMACS Performance Equivalency
Single GPU Server vs Multiple CPU-Only Servers

To arrive at CPU node equivalence, we use measured benchmark with up to 8 CPU nodes. Then we use linear scaling to scale beyond 8 nodes.

GROMACS
Molecular Dynamics

Simulation of biochemical molecules with complicated bond interactions

VERSION
5.1.2

ACCELERATED FEATURES
PME, Explicit & Implicit Solvent

SCALABILITY
Multi-GPU and Multi-Node
Scales to 4xP100

More Information
http://www.gromacs.org
AMBER Performance Equivalency

Single GPU Server vs Multiple CPU-Only Servers

CPU Server: Dual Xeon E5-2690 v4@2.6GHz, GPU Servers: same CPU server w/ P100s PCIe (12GB or 16GB)
CUDA Version: CUDA 8.0.44, Dataset: GB-Myoglobin
To arrive at CPU node equivalence, we use measured benchmark with up to 8 CPU nodes. Then we use linear scaling to scale beyond 8 nodes.
GPU ACCELERATED LIBRARIES
• A parallel computing platform and application programming interface (API) model created by NVIDIA

• Allows software developers and software engineers to use a CUDA-enabled GPUs for general purpose processing

• The name CUDA was originally an acronym for Compute Unified Device Architecture
TEN YEARS OF GPU COMPUTING

- **2006**: CUDA Launched
- **2008**: World’s First GPU Top500 System
- **2010**: Fermi: World’s First HPC GPU
- **2012**: Discovered How H1N1 Mutates to Resist Drugs
- **2014**: Stanford Builds AI Machine using GPUs
- **2016**: World’s First 3-D Mapping of Human Genome
- **2016**: Google Outperforms Humans in ImageNet
- **2016**: GPU-Driven AI Machine Beats World Champion in Go
- **2010**: Oak Ridge Deploys World’s Fastest Supercomputer w/ GPUs
- **2012**: AlexNet beats expert code by huge margin using GPUs
- **2014**: World’s First Atomic Model of HIV Capsid
- **2016**: NVIDIA CUDA
CUDA TOOLKIT

GPU-Accelerated Libraries
- Fast Fourier Transforms (cuFFT)
- Basic Linear Algebra Subroutines (cuBLAS)
- Sparse Matrix Routines (cuSPARSE)
- Dense and Sparse Direct Solvers (cuSOLVER)
- Random Number Generation (cuRAND)
- Image & Video Processing Primitives (NPP)
- NVIDIA Graph Analytics Library (nvGRAPH)
- Templated Parallel Algorithms & Data Structures (Thrust)
- CUDA Math Library

Development Tools
- NVIDIA CUDA C/C++ Compiler (NVCC)
- Nsight Integrated Development Environments
- Visual Profiler
- CUDA-GDB Command Line Debugger
- CUDA-MEMCHECK Memory Analyzer

Reference Materials
- CUDA C/C++ code samples
- CUDA Documentation
GPU ACCELERATED LIBRARIES
“Drop-in” Acceleration for Your Applications

Domain-specific
Deep Learning, GIS, EDA, Bioinformatics, Fluids

Visual Processing
Image & Video

Linear Algebra
Dense, Sparse, Matrix

Math Algorithms
AMG, Templates, Solvers

developer.nvidia.com/gpu-accelerated-libraries
### POWERING THE DEEP LEARNING ECOSYSTEM

**NVIDIA SDK accelerates every major framework**

<table>
<thead>
<tr>
<th>COMPUTER VISION</th>
<th>SPEECH &amp; AUDIO</th>
<th>NATURAL LANGUAGE PROCESSING</th>
</tr>
</thead>
<tbody>
<tr>
<td>OBJECT DETECTION</td>
<td>VOICE RECOGNITION</td>
<td>RECOMMENDATION ENGINES</td>
</tr>
<tr>
<td>IMAGE CLASSIFICATION</td>
<td>LANGUAGE TRANSLATION</td>
<td>SENTIMENT ANALYSIS</td>
</tr>
<tr>
<td><strong>IMAGENET</strong></td>
<td><strong>Microphone</strong></td>
<td><img src="emoji" alt="Emoticons" /></td>
</tr>
<tr>
<td><img src="image" alt="Object Detection" /></td>
<td><img src="image" alt="Voice Recognition" /></td>
<td><img src="image" alt="Recommendation Engine" /></td>
</tr>
<tr>
<td><img src="image" alt="Image Classification" /></td>
<td><img src="image" alt="Language Translation" /></td>
<td><img src="image" alt="Sentiment Analysis" /></td>
</tr>
</tbody>
</table>

### DEEP LEARNING FRAMEWORKS

- Caffe
- DL4J
- Mocha.jl
- Julia
- Keras
- MXNet
- Msat
- OpenDeep
- Pylearn2
- TensorFlow
- Theano

### NVIDIA DEEP LEARNING SDK

- cuDNN
- TensorRT
- DeepStream SDK
- cuBLAS
- cuSPARSE
- NCCL

cuDNN

Deep Learning Primitives

- GPU-accelerated Deep Learning subroutines
- High performance neural network training
- Accelerates Major Deep Learning frameworks: Caffe, Theano, Torch
- Up to 3.5x faster AlexNet training in Caffe than baseline GPU

Millions of Images Trained Per Day

Thousands of Images Trained Per Day

Tiled FFT up to 2x faster than FFT

developer.nvidia.com/cudnn
GRAPH ANALYTICS
Insight from connections in big data

SOCIAL NETWORK ANALYSIS

CYBER SECURITY / NETWORK ANALYTICS

GENOMICS

... and much more: Parallel Computing, Recommender Systems, Fraud Detection, Voice Recognition, Text Understanding, Search
nvGRAPH
GPU Accelerated Graph Analytics

Parallel Library for Interactive and High Throughput Graph Analytics

- Solve graphs with up to 2.5 Billion edges on a single GPU (Tesla M40)
- Includes — PageRank, Single Source Shortest Path and Single Source Widest Path algorithms
- Semi-ring SPMV operations provides building blocks for graph traversal algorithms

<table>
<thead>
<tr>
<th>PageRank</th>
<th>Single Source Shortest Path</th>
<th>Single Source Widest Path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Search</td>
<td>Robotic Path Planning</td>
<td>IP Routing</td>
</tr>
<tr>
<td>Recommendation Engines</td>
<td>Power Network Planning</td>
<td>Chip Design / EDA</td>
</tr>
<tr>
<td>Social Ad Placement</td>
<td>Logistics &amp; Supply Chain Planning</td>
<td>Traffic sensitive routing</td>
</tr>
</tbody>
</table>
NVIDIA DeepStream SDK

High performance deep learning inference for video analytics

DeepStream SDK simplifies development of high performance video analytics applications powered by deep learning

Delivers responsive and interactive AI-powered video services such as real-time video categorization

Simple high-level C++ API for GPU-accelerated transcoding, image resizing and scaling and deep learning inference powered by TensorRT™

developer.nvidia.com/deepstream-sdk
**NVIDIA TensorRT**

High-performance deep learning inference for production deployment

High performance neural network inference engine for production deployment

- Generate optimized and deployment-ready models for datacenter, embedded and automotive platforms
- Deliver high-performance, low-latency inference demanded by real-time services
- Deploy faster, more responsive and memory efficient deep learning applications with INT8 and FP16 optimized precision support

---

*developer.nvidia.com/tensorrt*
INT8 PRECISION
New in TensorRT

PERFORMANCE
Up To 3x More Images/sec with INT8 Precision

EFFICIENCY
Deploy 2x Larger Models with INT8 Precision

ACCURACY
Deliver full accuracy with INT8 precision

GoogLeNet, FP32 vs INT8 precision + TensorRT on Tesla P40 GPU, 2 Socket Haswell E5-2698 v3@2.3GHz with HT off
GOAL:
• Build a research library of accelerated collectives that is easily integrated and topology-aware so as to improve the scalability of multi-GPU applications

APPROACH:
• 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 DEMO

git clone https://github.com/NVIDIA/nccl

cd nccl/nccl-master

make CUDA_HOME=/usr/local/cuda test

export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.:/build/lib

Sample test is:

./build/test/single/all_reduce_test 10000000 4 0 1 2 3

Usage: all_reduce_test <data size in bytes> [number of GPUs] [GPU 0] [GPU 1] …
OPENACC ACCELERATION
How it works:
• Insert compiler hints into the compute-intensive portions of Fortran or C application
• Compiler automatically maps that code to an accelerator (NVIDIA GPUs)
• fully compatible — and interoperates — with OpenMP and MPI.

OpenACC compilers are:
• Portable: Future-proof your codes with this open standard
• Fast: Straight forward, high-level, compiler driven approach to parallel computing
• Powerful: Ideal for accelerating large, legacy Fortran or C codes
OPENACC ACCELERATION

1. Identify Available Parallelism
2. Express Parallelism
3. Express Data Movement
4. Optimize Loop Performance
LSDalton: Quantum Chemistry, 12X speedup in 1 week

Numeca: CFD, 10X faster kernels, 2X faster app

PowerGrid: Medical Imaging, 40 days to 2 hours

INCOMP3D: CFD, 3X speedup

NekCEM: Computational Electromagnetics, 2.5X speedup, 60% less energy

COSMO: Climate Weather, 40X speedup, 3X energy efficiency

CloverLeaf: CFD, 4X speedup, Single CPU/GPU code

MAESTRO CASTRO: Astrophysics, 4.4X speedup, 4 weeks effort
## OPENACC FOR EVERYONE

New PGI Community Edition Now Available

### PROGRAMMING MODELS
- OpenACC, CUDA Fortran, OpenMP, C/C++/Fortran Compilers and Tools

### PLATFORMS
- x86, OpenPOWER, NVIDIA GPU

### UPDATES
- 1-2 times a year
- 6-9 times a year
- 6-9 times a year

### SUPPORT
- User Forums
- PGI Support
- PGI Enterprise Services

### LICENSE
- Annual
- Perpetual
- Volume/Site
OPENACC DIRECTIVES

Simple Compiler hints

Compiler Parallelizes code

Works on many-core GPUs & multicore CPUs

Your original Fortran or C code

Program myscience
  ... serial code ...
  !$acc kernels
  do k = 1,n1
    do i = 1,n2
      ... parallel code ...
    enddo
  enddo
  !$acc end kernels
  ...
End Program myscience
FAMILIAR TO OPENMP PROGRAMMERS

OpenMP

```
main() {
    double pi = 0.0; long i;

    #pragma omp parallel for reduction(+:pi)
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f
", pi/N);
}
```

OpenACC

```
main() {
    double pi = 0.0; long i;

    #pragma acc kernels
    for (i=0; i<N; i++)
    {
        double t = (double)((i+0.05)/N);
        pi += 4.0/(1.0+t*t);
    }

    printf("pi = %f\n", pi/N);
}
```
DIRECTIVE SYNTAX

Fortran

!$acc directive [clause [,] clause] ...]
Often paired with a matching end directive surrounding a structured code block

!$acc end directive

C

#pragma acc directive [clause [,] clause] ...
Often followed by a structured code block
KERNELS: YOUR FIRST OPENACC DIRECTIVE

Each loop executed as a separate *kernel* on the GPU.

```c
!$acc kernels
do i=1,n
   a(i) = 0.0
   b(i) = 1.0
   c(i) = 2.0
end do

  kernel 1

  Kernel:
  A parallel function that runs on the GPU

  a(i) = b(i) + c(i)
end do

!$acc end kernels
```

kernel 2
A VERY SIMPLE EXERCISE: SAXPY

SAXPY in C

```c
void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...```

SAXPY in Fortran

```fortran
subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
$!acc kernels
    do i=1,n
        y(i) = a*x(i)+y(i)
    enddo$!acc end kernels
end subroutine saxpy
...
$ Perform SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...```
KERNELS CONSTRUCT

Fortran

```fortran
!$acc kernels [clause ...]
    structured block
!$acc end kernels
```

C

```c
#pragma acc kernels [clause ...]
    { structured block }
```

Clauses

```
if( condition )

async( expression )
```

Also, any data clause
#include <stdlib.h>

void saxpy(int n,
    float a,
    float *x,
    float *restrict y)
{
    #pragma acc kernels
    for (int i = 0; i < n; ++i)
        y[i] = a * x[i] + y[i];
}

int main(int argc, char **argv)
{
    int N = 1<<20; // 1 million floats
    if (argc > 1)
        N = atoi(argv[1]);
    float *x = (float*)malloc(N * sizeof(float));
    float *y = (float*)malloc(N * sizeof(float));
    for (int i = 0; i < N; ++i) {
        x[i] = 2.0f;
        y[i] = 1.0f;
    }
    saxpy(N, 3.0f, x, y);
    return 0;
}
**COMPILE AND RUN**

C: `pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c`

Fortran: `pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90`

Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
  8, Generating copyin(x[:n-1])
  Generating copy(y[:n-1])
  Generating compute capability 1.0 binary
  Generating compute capability 2.0 binary
  9, Loop is parallelizable
  Accelerator kernel generated
  9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
  CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
  CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```
OTHER DIRECTIVES

OpenACC defines an extensive list of pragmas (directives), for example:

`#pragma acc parallel`

`#pragma acc kernels`

Both are used to define parallel computation kernels to be executed on the accelerator, using distinct semantics

`#pragma acc data`

Is the main directive to define and copy data to and from the accelerator.

`#pragma acc loop`

Is used to define the type of parallelism in a parallel or kernels region.
DIRECT PROGRAMMING
DIRECT PROGRAMMING

Advantages:
• Highest performance
• Greater flexibility
• Tighter code control

Disadvantages:
• Higher level of effort
• Less portable
1. Copy input data from CPU memory to GPU memory
1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
PROCESSING FLOW

1. Copy input data from CPU memory to GPU memory
2. Load GPU program and execute, caching data on chip for performance
3. Copy results from GPU memory to CPU memory
DEVELOPMENT ENVIRONMENT

• Compile CUDA c/c++ with nvcc
• Compile CUDA fortran with PGI pgfortran
• Compute OpenACC with PGI pgc++, pgcc, pgfortran
• Tools
  • Profile with nvprof, nvvp, pgprof
  • Debug with cuda-gdb, pgdbg
  • Memcheck with cuda-memcheck
• pyCUDA maps all of CUDA into Python
• Numba Python compiler with Anaconda Accelerate
NVPROF AND CUDA-MEMCHECK DEMO

cd /usr/local/cuda/samples/5_Simulations/nbody
make
nvprof --print-summary ./nbody -benchmark -numdevices=8

cuda-memcheck ./nbody -benchmark -numdevices=8
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 ARCHITECTURE

CUDA Core

Floating point & Integer unit

IEEE 754-2008 floating-point standard

Fused multiply-add (FMA) instruction for both single and double precision

Logic unit

Move, compare unit

Branch unit
SXM2 MODULE
TESLA P100 GPU: GP100

56 SMs
3584 CUDA Cores
5.3 TF Double Precision
10.6 TF Single Precision
21.2 TF Half Precision
16 GB HBM2
720 GB/s Bandwidth
TESLA GP102
GP100 STREAMING MULTIPROCESSOR
TESLA P100 COMPARED TO PRIOR GENERATION TESLA PRODUCTS

<table>
<thead>
<tr>
<th>Tesla Products</th>
<th>Tesla K40</th>
<th>Tesla M40</th>
<th>Tesla P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>GK110 (Kepler)</td>
<td>GM200 (Maxwell)</td>
<td>GP100 (Pascal)</td>
</tr>
<tr>
<td>SMs</td>
<td>15</td>
<td>24</td>
<td>56</td>
</tr>
<tr>
<td>FP32 CUDA Cores / SM</td>
<td>192</td>
<td>128</td>
<td>64</td>
</tr>
<tr>
<td>FP32 CUDA Cores / GPU</td>
<td>2880</td>
<td>3072</td>
<td>3584</td>
</tr>
<tr>
<td>FP64 CUDA Cores / SM</td>
<td>64</td>
<td>4</td>
<td>32</td>
</tr>
<tr>
<td>FP64 CUDA Cores / GPU</td>
<td>960</td>
<td>96</td>
<td>1792</td>
</tr>
<tr>
<td>Base Clock</td>
<td>745 MHz</td>
<td>948 MHz</td>
<td>1328 MHz</td>
</tr>
<tr>
<td>GPU Boost Clock</td>
<td>810/875 MHz</td>
<td>1114 MHz</td>
<td>1480 MHz</td>
</tr>
</tbody>
</table>
# TESLA P100 COMPARED TO PRIOR GENERATION TESLA PRODUCTS

<table>
<thead>
<tr>
<th>Tesla Products</th>
<th>Tesla K40</th>
<th>Tesla M40</th>
<th>Tesla P100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peak FP32 GFLOPs</td>
<td>5040</td>
<td>6840</td>
<td>10600</td>
</tr>
<tr>
<td>Peak FP64 GFLOPs</td>
<td>1680</td>
<td>210</td>
<td>5300</td>
</tr>
<tr>
<td>Memory Interface</td>
<td>384-bit GDDR5</td>
<td>384-bit GDDR5</td>
<td>4096-bit HBM2</td>
</tr>
<tr>
<td>Memory Size</td>
<td>Up to 12 GB</td>
<td>Up to 24 GB</td>
<td>16 GB</td>
</tr>
<tr>
<td>L2 Cache Size</td>
<td>1536 KB</td>
<td>3072 KB</td>
<td>4096 KB</td>
</tr>
<tr>
<td>Register File Size / SM</td>
<td>256 KB</td>
<td>256 KB</td>
<td>256 KB</td>
</tr>
<tr>
<td>Register File Size / GPU</td>
<td>3840 KB</td>
<td>6144 KB</td>
<td>14336 KB</td>
</tr>
<tr>
<td>TDP</td>
<td>235 Watts</td>
<td>250 Watts</td>
<td>300 Watts</td>
</tr>
<tr>
<td>Transistors</td>
<td>7.1 billion</td>
<td>8 billion</td>
<td>15.3 billion</td>
</tr>
<tr>
<td>GPU Die Size</td>
<td>551 mm²</td>
<td>601 mm²</td>
<td>610 mm²</td>
</tr>
<tr>
<td>Manufacturing Process</td>
<td>28-nm</td>
<td>28-nm</td>
<td>16-nm FinFET</td>
</tr>
</tbody>
</table>
HIGHER THROUGHPUT THROUGH LOWER PRECISION COMPUTATION

Deep Learning  
cuBLAS: FP16 and INT8 GEMMS

Radio Astronomy  
cuFFT: native FP16 operations

Fluid Dynamics  
cuSPARSE: FP16 CSRMOV
HALF-PRECISION FLOATING POINT (FP16)

- 16 bits
  - 1 sign bit, 5 exponent bits, 10 fraction bits
- $2^{40}$ Dynamic range
  - Normalized values: 1024 values for each power of 2, from $2^{-14}$ to $2^{15}$
  - Subnormals at full speed: 1024 values from $2^{-24}$ to $2^{-15}$
- Special values
  - +/- Infinity, Not-a-number

USE CASES
- Deep Learning Training
- Radio Astronomy
- Sensor Data
- Image Processing
NVLink
High speed GPU communication
```\text{Legend:}
\begin{itemize}
\item \textbf{X} = Self
\item \textbf{SOC} = Connection traversing PCIe as well as the SMP link between CPU sockets (e.g. QPI)
\item \textbf{PHB} = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
\item \textbf{PIXB} = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
\item \textbf{PIX} = Connection traversing a single PCIe switch
\item \textbf{NV#} = Connection traversing a bonded set of # NVLinks
\end{itemize}
```
CUDA KERNELS
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>
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 KERNELES: 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

CUDA thread block

CUDA kernel grid

CUDA core

CUDA Streaming Multiprocessor

CUDA-enabled GPU

- 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.
## Compute Capability

<table>
<thead>
<tr>
<th>GPU</th>
<th>Kepler GK110</th>
<th>Maxwell GM200</th>
<th>Pascal GP100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compute Capability</td>
<td>3.5</td>
<td>5.2</td>
<td>6.0</td>
</tr>
<tr>
<td>Max Threads/Multiprocessor</td>
<td>2048</td>
<td>2048</td>
<td>2048</td>
</tr>
<tr>
<td>Max Thread Blocks/Multiprocessor</td>
<td>16</td>
<td>32</td>
<td>32</td>
</tr>
<tr>
<td>Max 32-bit Registers/SM</td>
<td>65536</td>
<td>65536</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers/Block</td>
<td>65536</td>
<td>32768</td>
<td>65536</td>
</tr>
<tr>
<td>Max Registers/Thread</td>
<td>255</td>
<td>255</td>
<td>255</td>
</tr>
<tr>
<td>Max Thread Block Size</td>
<td>1024</td>
<td>1024</td>
<td>1024</td>
</tr>
<tr>
<td>Shared Memory Size/SM</td>
<td>16KB/32KB/48KB</td>
<td>96KB</td>
<td>64KB</td>
</tr>
</tbody>
</table>
GPU MEMORY HIERARCHY REVIEW

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

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

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

- **L2**
- **Global Memory**
GPU ARCHITECTURE

Memory System on each SM

Extremely fast, but small, i.e., 10s of Kb

Programmer chooses whether to use cache as L1 or Shared Mem

L1

Hardware-managed—used for things like register spilling

Should NOT attempt to utilize like CPU caches

Shared Memory—programmer MUST synchronize data accesses!!!

User-managed scratch pad

Repeated access to same data or multiple threads with same data
GPU ARCHITECTURE

Memory system on each GPU board

Unified L2 cache (100s of Kb)
  Fast, coherent data sharing across all cores in the GPU

ECC protection

DRAM
  ECC supported for GDDR5 memory (built-in for HBM2)

All major internal memories are ECC protected
  Register file, L1 cache, L2 cache
Thread:

Registers
Thread:
  - Registers
  - Local memory
MEMORY HIERARCHY

- Thread:
  - Registers
  - Local memory

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

__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

cudaMalloc (void ** pointer, size_t nbytes)
cudaMemset (void * pointer, int value, size_t count)
cudaFree (void* pointer)

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
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 SYNTAX

__global__ keyword used to tell the CUDA compiler that the function is to be compiled for the GPU
blockIdx.x read-only, defined for you, the ID of the block which is currently executing code
threadIdx.x read-only, defined for you, ID of the thread which is currently executing code in the active block
blockDim.x read-only, defined for you, number of threads there are per block
myKernel <<< number_of_blocks, threads_per_block>> (…) syntax used to launch a kernel on the GPU
COMPILING AND LINKING

The `nvcc` compiler does the following basic steps:
1. From the `.cu` source file, separate code which should be compiled for the GPU and the code which should be compiled for the CPU
2. `nvcc` will compile the GPU code itself
3. `nvcc` will give the host compiler, in our case `gcc`, the CPU code to compile
4. Link the compiled code from #2 and #3 and create the executable
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);
GPUDIRECT

- Accelerated communication with network and storage devices
- Peer-to-Peer Transfers between GPUs
- Peer-to-Peer memory access
- RDMA
- GPUDirect for Video
IMPORTANCE OF GPU TOPOLOGY

GPUs on same PCIe Host support GPUDirect P2P

P2P limits performance across QPI
QPI does not entirely cover all features of the PCIE protocol
Without GPUDirect

Same data copied three times:
1. GPU writes to pinned sysmem1
2. CPU copies from sysmem1 to sysmem2
3. InfiniBand driver copies from sysmem2

With GPUDirect

Data only copied twice
Sharing pinned system memory makes sysmem-to-sysmem copy unnecessary
GPU-AWARE MPI LIBRARIES
INTEGRATED SUPPORT FOR GPU COMPUTING

OpenMPI
MVAPICH
IBM Spectrum MPI
Cray MPI

GPUDirect™ P2P Transfers
cudaMemcpy()

GPUDirect™ RDMA

developer.nvidia.com/gpudirect
WHY COMBINE MPI AND CUDA?

• To solve problems with a data size too large to fit into the memory of a single GPU
• To solve problems that would require unreasonably long compute time on a single node
• To accelerate an existing MPI application with GPUs
• To enable a single-node multi-GPU application to scale across multiple nodes

CUDA-aware implementations of MPI have several advantages:
• CUDA-aware MPI is relatively easy to use
• Applications run more efficiently with CUDA-aware MPI
  • Operations that carry out the message transfers can be pipelined
  • CUDA-aware MPI takes advantage of best GPU Direct technology available
GPU-AWARE MPI LIBRARIES

**MVAPICH2** is an open source implementation of Message Passing Interface (MPI) and simplifies the task of porting MPI applications to run on clusters with NVIDIA GPUs by supporting standard MPI calls from GPU device memory.

**IBM™ Spectrum MPI** is a high-performance, production-quality implementation of MPI designed to accelerate application performance in distributed computing environments.

**The Open MPI Project** is an open source MPI-2 implementation that is developed and maintained by a consortium of academic, research, and industry partners. GPUs are supported by version 1.7 and later.
UNIFIED MEMORY
Before the release of CUDA 6:
• the developer handled allocation and movement of data between CPU/GPU memory spaces

Starting in CUDA 6:
• Alternate method introduced called Unified Memory
• the underlying system handles moving data between CPU and GPU memory

Unified Memory API calls:

    cudaMallocManaged ( T** devPtr, size_t size ); - allocate size bytes in managed memory and store in devPtr

    cudaFree ( void* devPtr ) - to free any memory allocated in managed memory
UNIFIED MEMORY
Implicit Memory Management

Past Developer View

Starting with Kepler and CUDA 6

CPU

Pascal GPU

System Memory

GPU Memory

Unified Memory
CUDA 8 UNIFIED MEMORY — EXAMPLE

Allocating 4x more than P100 physical memory

```c
void foo() {
    // Allocate 64 GB
    char *data;
    size_t size = 64*1024*1024*1024;
    cudaMallocManaged(&data, size);
}
```

64 GB unified memory allocation on P100 with 16 GB physical memory

Transparent - No API changes

Works on Pascal & future architectures
CUDA 8 UNIFIED MEMORY — EXAMPLE
Accessing data simultaneously by CPU and GPU codes

```c
__global__ void mykernel(char *data) {
    data[1] = 'g';
}

void foo() {
    char *data;
    cudaMallocManaged(&data, 2);
    mykernel<<<...>>>(data);
    // no synchronize here
    data[0] = 'c';
    cudaFree(data);
}
```

Both CPU code and CUDA kernel accessing ‘data’ simultaneously
Possible with CUDA 8 unified memory on Pascal
DEEP LEARNING WORKFLOWS WITH GPUS
DEEP LEARNING WORKFLOW
NVIDIA DEEP LEARNING SDK

Powerful tools and libraries for designing and deploying GPU-accelerated deep learning applications

High performance building blocks for training and deploying deep neural networks on NVIDIA GPUs

Industry vetted deep learning algorithms and linear algebra subroutines for developing novel deep neural networks

Multi-GPU scaling that accelerates training on up to eight GPU

“We are amazed by the steady stream of improvements made to the NVIDIA Deep Learning SDK and the speedups that they deliver.”

— Frédéric Bastien, Team Lead (Theano) MILA

developer.nvidia.com/deep-learning-software
POWERING THE DEEP LEARNING ECOSYSTEM

NVIDIA SDK accelerates every major framework

COMPUTER VISION
- OBJECT DETECTION
- IMAGE CLASSIFICATION

SPEECH & AUDIO
- VOICE RECOGNITION
- LANGUAGE TRANSLATION

NATURAL LANGUAGE PROCESSING
- RECOMMENDATION ENGINES
- SENTIMENT ANALYSIS

DEEP LEARNING FRAMEWORKS
- Caffe
- DL4J
- Mocha.jl
- Julia
- Keras
- Microsoft CNTK
- MxNet
- Purine
- TensorFlow
- Torch
- Minerva
- OpenDeep
- Pylearn2
- Theano

NVIDIA DEEP LEARNING SDK
- cuDNN
- TensorRT
- DeepStream SDK
- cuBLAS
- cuSPARSE
- NCCL

developer.nvidia.com/deep-learning-software
NVIDIA DIGITS
Interactive Deep Learning GPU Training System

Interactive deep learning training application for engineers and data scientists

Simplify deep neural network training with an interactive interface to train and validate, and visualize results

Built-in workflows for image classification, object detection and image segmentation

Improve model accuracy with pre-trained models from the DIGITS Model Store

Faster time to solution with multi-GPU acceleration
# Deep Learning Workflows

## Image Classification

<table>
<thead>
<tr>
<th>Image</th>
<th>98% Dog</th>
<th>2% Cat</th>
</tr>
</thead>
</table>

- Classify images into classes or categories
- Object of interest could be anywhere in the image

## Object Detection

- Find instances of objects in an image
- Objects are identified with bounding boxes

## Image Segmentation

- Partition image into multiple regions
- Regions are classified at the pixel level

New in DIGITS 5
MNIST data set of handwritten digits from Yann Lecun’s website.

All images are 28x28 grayscale

- Pixel values from 0 to 255
- 60k training examples, 10k test examples
- Input vector of size 784
- Output value is integer from 0-9
WHAT’S NEW IN DIGITS 5

**IMAGE SEGMENTATION**
Partition images into regions of interest

**MODEL STORE**
Download pre-trained neural networks
GPU STATUS CHECKING AND MONITORING
```
$ nvidia-smi

<table>
<thead>
<tr>
<th>NVIDIA-SMI 375.39</th>
<th>Driver Version: 375.39</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU Name</td>
<td>Persistence-M</td>
</tr>
<tr>
<td>-------------------</td>
<td>-------------------</td>
</tr>
<tr>
<td>0 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>1 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>2 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>3 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>4 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>5 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>6 Tesla P100-SXM2</td>
<td>On</td>
</tr>
<tr>
<td>7 Tesla P100-SXM2</td>
<td>On</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Processes:</th>
<th>GPU Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU</td>
<td>PID</td>
</tr>
<tr>
<td>-------------------</td>
<td>--------</td>
</tr>
<tr>
<td>0</td>
<td>3827</td>
</tr>
<tr>
<td>0</td>
<td>80108</td>
</tr>
<tr>
<td>0</td>
<td>80109</td>
</tr>
<tr>
<td>2</td>
<td>80110</td>
</tr>
<tr>
<td>2</td>
<td>80111</td>
</tr>
</tbody>
</table>
```
<table>
<thead>
<tr>
<th>GPU0</th>
<th>GPU1</th>
<th>GPU2</th>
<th>GPU3</th>
<th>GPU4</th>
<th>GPU5</th>
<th>GPU6</th>
<th>GPU7</th>
<th>mlx5_0</th>
<th>mlx5_2</th>
<th>mlx5_1</th>
<th>mlx5_3</th>
<th>CPU Affinity</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU0</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
</tr>
<tr>
<td>GPU1</td>
<td>PIX</td>
<td>X</td>
<td>PHB</td>
<td>PHB</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
</tr>
<tr>
<td>GPU2</td>
<td>NV1</td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>PHB</td>
<td>PHB</td>
<td>SOC</td>
</tr>
<tr>
<td>GPU3</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
</tr>
<tr>
<td>GPU4</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
</tr>
<tr>
<td>GPU5</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>NV1</td>
<td>SOC</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
</tr>
<tr>
<td>GPU6</td>
<td>SOC</td>
<td>SOC</td>
<td>NV1</td>
<td>SOC</td>
<td>NV1</td>
<td>NV1</td>
<td>X</td>
<td>NV1</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>PIX</td>
</tr>
<tr>
<td>mlx5_0</td>
<td>PIX</td>
<td>PIX</td>
<td>PHB</td>
<td>PHB</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>X</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
</tr>
<tr>
<td>mlx5_1</td>
<td>PHB</td>
<td>PHB</td>
<td>PIX</td>
<td>PIX</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>X</td>
<td>SOC</td>
</tr>
<tr>
<td>mlx5_3</td>
<td>SOC</td>
<td>SOC</td>
<td>SOC</td>
<td>PHB</td>
<td>PHB</td>
<td>PIX</td>
<td>PIX</td>
<td>SOC</td>
<td>PHB</td>
<td>SOC</td>
<td>PHB</td>
<td>X</td>
</tr>
</tbody>
</table>

Legend:

- **X** = Self
- **SOC** = Connection traversing PCIe as well as the SMP link between CPU sockets (e.g. QPI)
- **PHB** = Connection traversing PCIe as well as a PCIe Host Bridge (typically the CPU)
- **PIX** = Connection traversing multiple PCIe switches (without traversing the PCIe Host Bridge)
- **NV#** = Connection traversing a bonded set of # NVLinks
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.
THREAD HIERARCHY

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

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

Kernel launches on a grid of blocks, \(<<<\text{grid,block}>>>(\text{arg1,...})\)

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.