GPU BOOTCAMP

KTH September 2019

Paul Graham : pgraham@nvidia.com

INTRODUCTION TO GPU COMPUTING

What to expect?

- Broad view on GPU Stack
- Fundamentals of GPU Architecture
- Ways to GPU Computing
- Good starting point

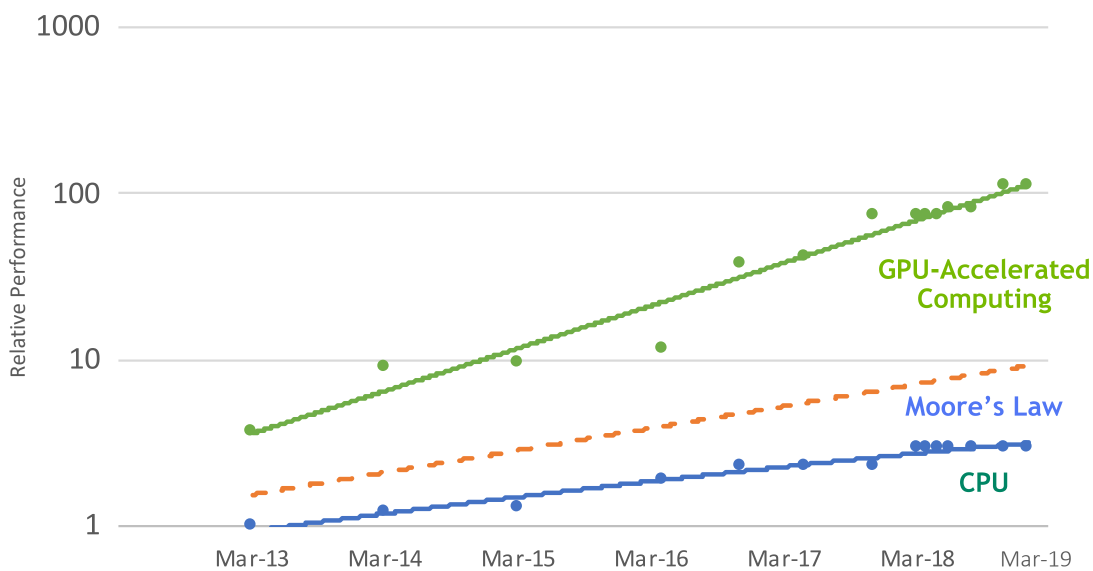
FULL STACK OPTIMIZATION

Progress Of Stack In 6 Years

2013

- cuBLAS: 5.0
- cuFFT: 5.0
- cuRAND: 5.0
- cuSPARSE: 5.0
- NPP: 5.0
- Thrust: 1.5.3
- CUDA: 5.0
- Resource Mgr: r304
- Base OS: CentOS 6.2

Accelerated Server With Fermi



Measured performance of Amber, CHROMA, GTC, LAMMPS, MILC, NAMD, Quantum Espresso, SPECFEM3D

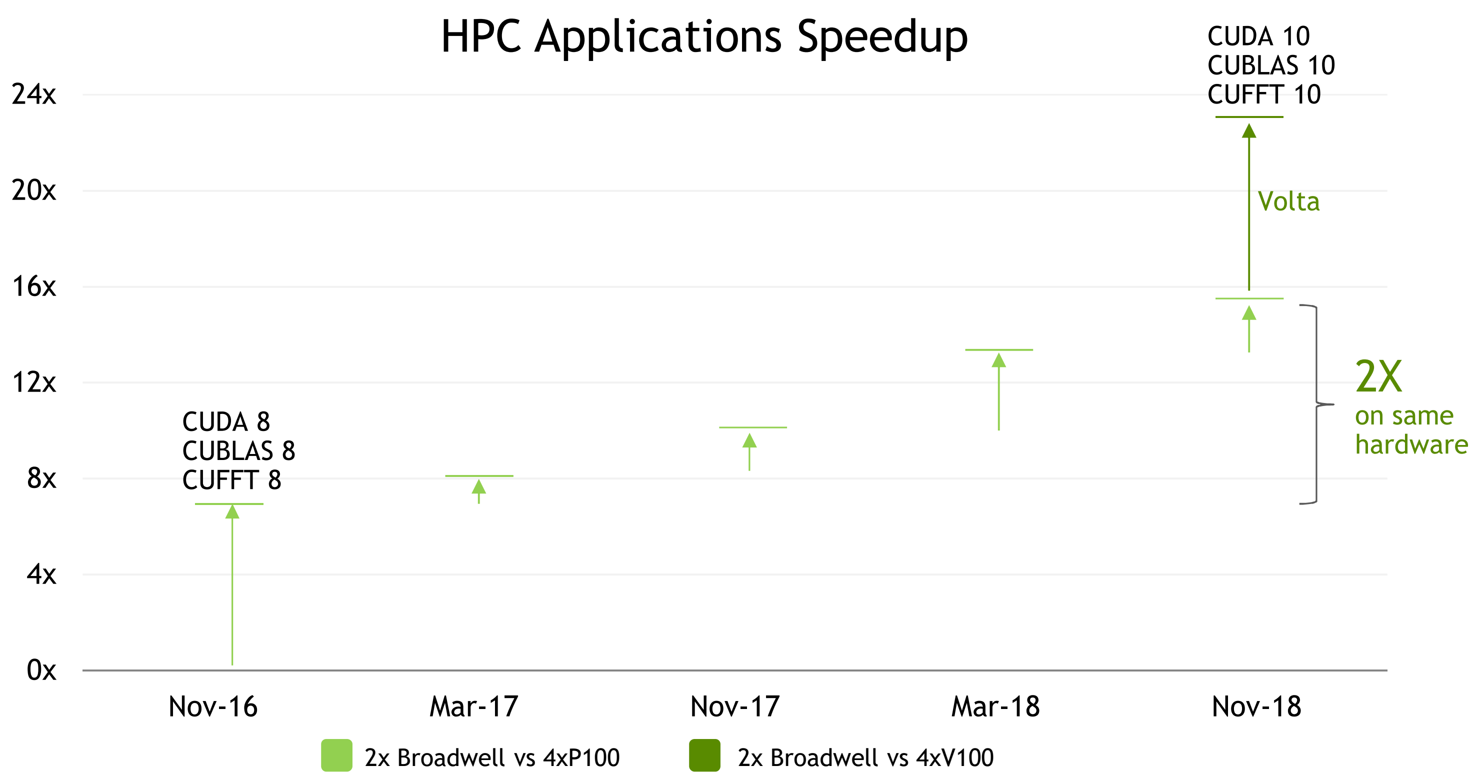
2019

- cuBLAS: 10.0
- cuFFT: 10.0
- cuRAND: 10.0
- cuSOLVER: 10.0
- cuSPARSE: 10.0
- NPP: 10.0
- Thrust: 1.9.0
- CUDA: 10.0
- Resource Mgr: r384
- Base OS: Ubuntu 16.04

Accelerated Server with Volta

ACCELERATED COMPUTING IS FULL-STACK OPTIMIZATION

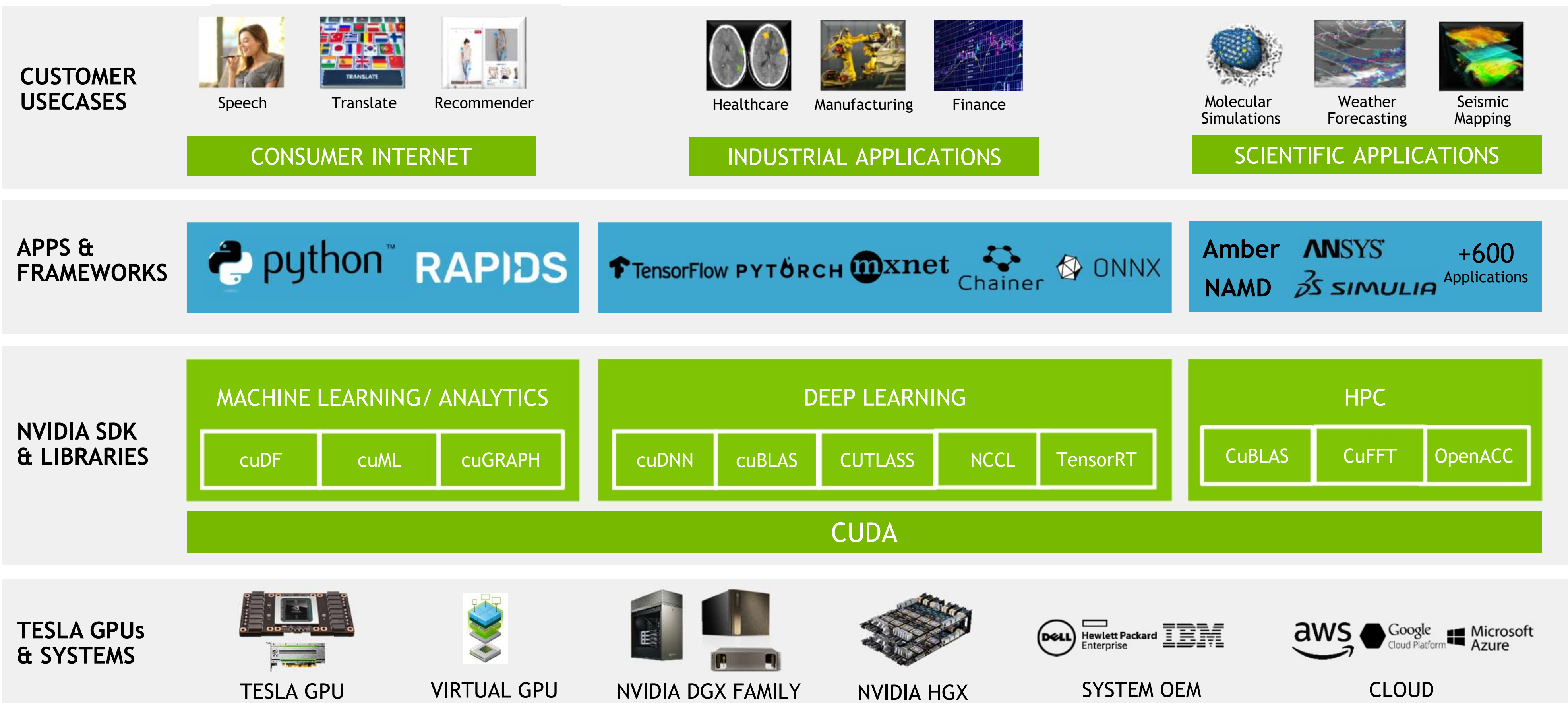
2X More Performance with Software Optimizations Alone



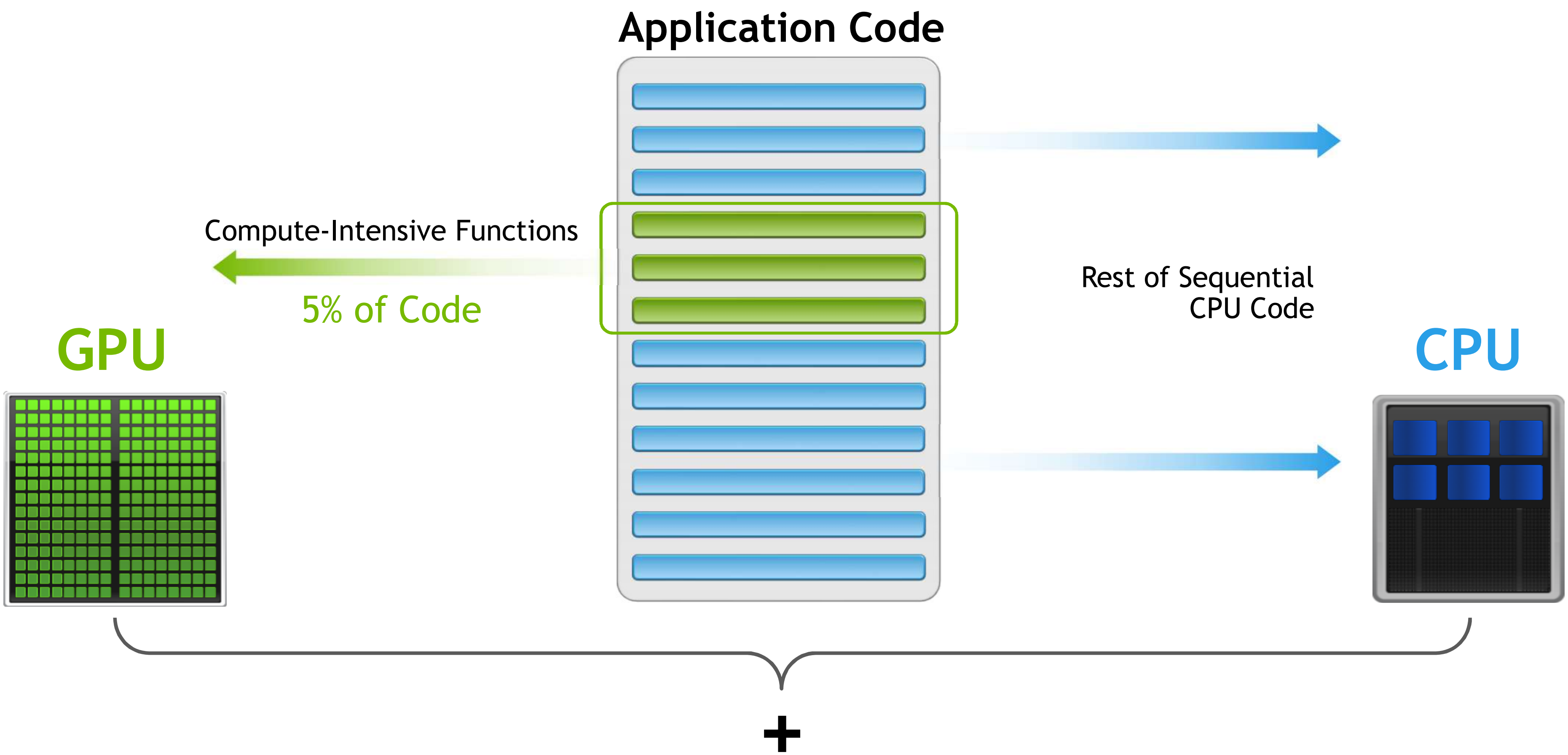
HPC Apps: AMBER, Chroma, GROMACS, GTC, LAMMPS, MILC, NAMD, QE, RTM, SPECFEM3D, VASP

NVIDIA UNIVERSAL ACCELERATION PLATFORM

Single Platform Drives Utilization and Productivity

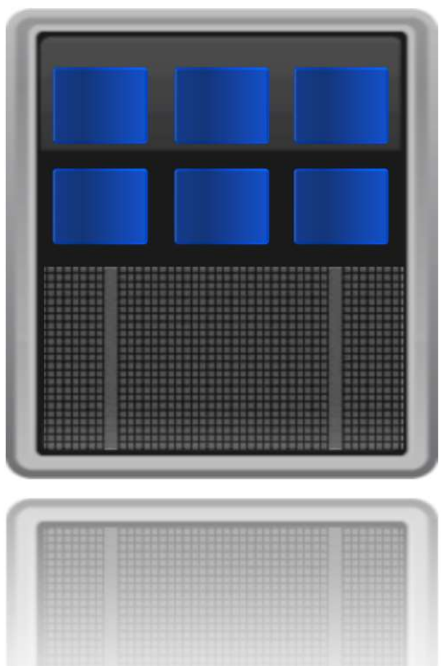


HOW GPU ACCELERATION WORKS

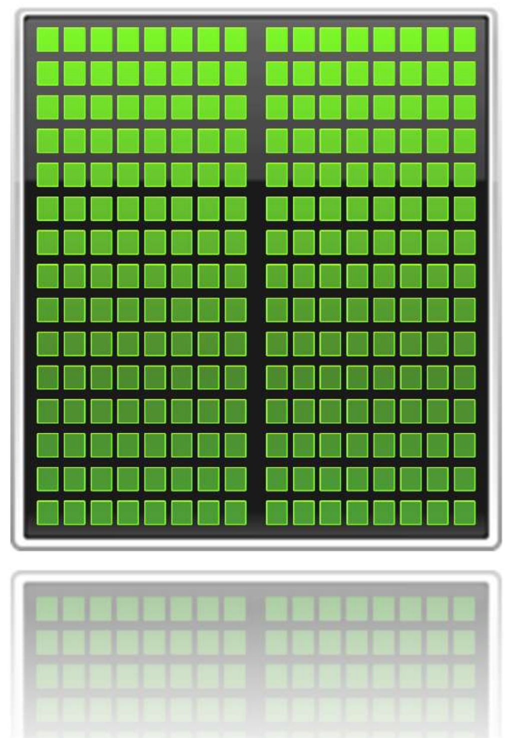


ACCELERATED COMPUTING

CPU
Optimized for
Serial Tasks

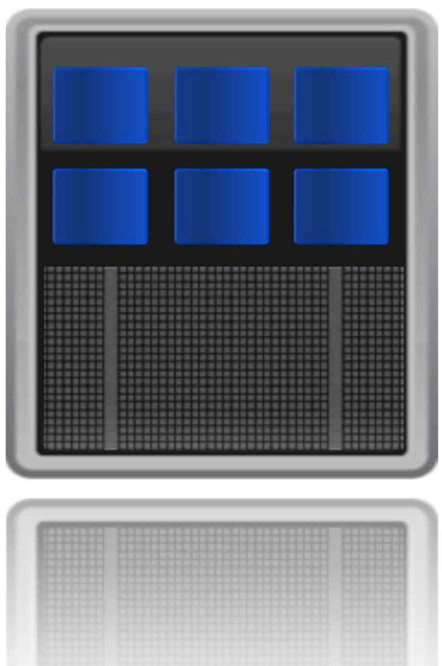


GPU Accelerator
Optimized for
Parallel Tasks



CPU IS A LATENCY REDUCING ARCHITECTURE

CPU
Optimized for
Serial Tasks



CPU Strengths

- Very large main memory
- Very fast clock speeds
- Latency optimized via large caches
- Small number of threads can run very quickly

CPU Weaknesses

- Relatively low memory bandwidth
- Cache misses very costly
- Low performance/watt

GPU IS ALL ABOUT HIDING LATENCY

GPU Strengths

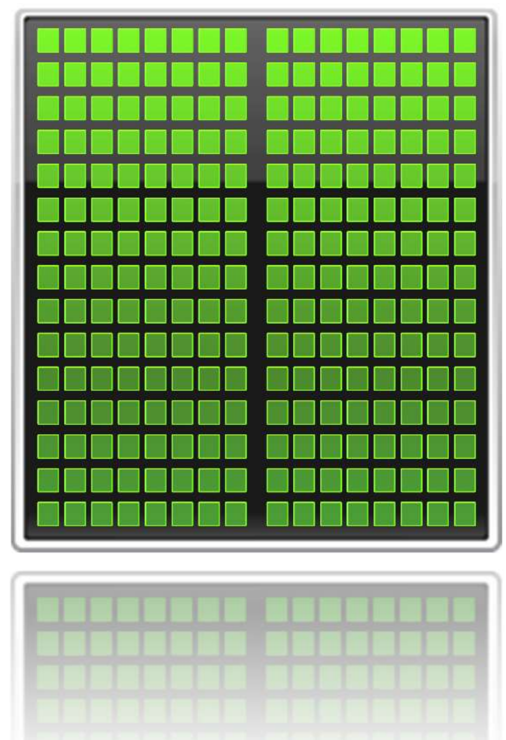
- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

GPU Accelerator

Optimized for
Parallel Tasks



HOW TO START WITH GPUS

1 Applications

2 Libraries	3 Compiler Directives	4 Programming Languages
Easy to use	Easy to Start	Most Performance
Most Performance	Portable Code	Most Flexibility
	OpenACC	CUDA

1. Review available GPU-accelerated applications
2. Check for GPU-Accelerated applications and libraries
3. Add OpenACC Directives for quick acceleration results and portability
4. Dive into CUDA for highest performance and flexibility

GPU COMPUTING PLATFORM FOR HPC SIMULATION

GPU-ACCELERATED APPLICATIONS

620 Applications Across Domains

- Life Sciences
- Manufacturing
- Physics
- Oil & Gas
- Climate & Weather
- Media & Entertainment
- Deep Learning
- Federal & Defense
- Data Science & Analytics
- Safety & Security
- Computational Finance
- Tool & Management

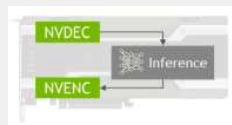
GPU ACCELERATED LIBRARIES

“Drop-in” Acceleration for Your Applications

DEEP LEARNING

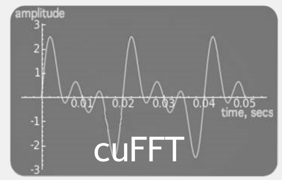
cuDNN

TensorRT



DeepStream SDK

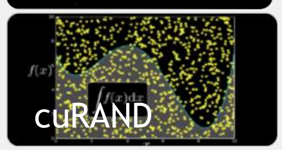
SIGNAL, IMAGE & VIDEO



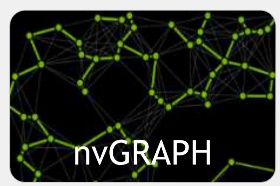
CODEC SDK

LINEAR ALGEBRA

cuSPARSE



PARALLEL ALGORITHMS



NCCL

Thrust

More libraries: https://developer.nvidia.com/gpu-accelerated-libraries

WHAT IS OPENACC

Programming Model for an Easy Onramp to GPUs

Directives-based programming model for **parallel computing**

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
 }
```

Simple

Designed for **performance portability** on CPUs and GPUs

Powerful & Portable

Read more at www.openacc.org/about

OpenACC is an open specification developed by OpenACC.org consortium

SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

GPU SAXPY in multiple languages and libraries

Part of Basic Linear Algebra Subroutines (BLAS) Library

$$\boldsymbol{z} = \alpha \boldsymbol{x} + \boldsymbol{y}$$

$\boldsymbol{x}, \boldsymbol{y}, \boldsymbol{z}$: vector

α : scalar

1 SAXPY: OPENACC COMPILER DIRECTIVES

Parallel C Code

```
void saxpy(int n,
           float a,
           float *x,
           float *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);
...
```

Parallel Fortran Code

```
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)
...
```

www.openacc.org

2

SAXPY: CUBLAS LIBRARY

Serial BLAS Code

```
int N = 1<<20;


...

// Use your choice of blas library

// Perform SAXPY on 1M elements
blas_saxpy(N, 2.0, x, 1, y, 1);
```

Parallel cuBLAS Code

```
int N = 1<<20;

cublasInit();
cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasShutdown();
```

You can also call cuBLAS from Fortran, C++, Python, and other languages:
http://developer.nvidia.com/cublas

3 SAXPY: CUDA C

Standard C

```
void saxpy(int n, float a,
      float *x, float *y)
{
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, x, y);
```

Parallel C

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

int N = 1<<20;
cudaMemcpy(d_x, x, N, cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, N, cudaMemcpyHostToDevice);

// Perform SAXPY on 1M elements
saxpy<<<4096,256>>>(N, 2.0, d_x, d_y);

cudaMemcpy(y, d_y, N, cudaMemcpyDeviceToHost);
```

http://developer.nvidia.com/cuda-toolkit

4 SAXPY: THRUST C++ TEMPLATE LIBRARY

Serial C++ Code (with STL and Boost)

```
int N = 1<<20;
std::vector<float> x(N), y(N);

...




// Perform SAXPY on 1M elements
std::transform(x.begin(), x.end(),
               y.begin(), y.end(),
          2.0f * _1 + _2);
```

www.boost.org/libs/lambda

Parallel C++ Code

```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);

...

thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

// Perform SAXPY on 1M elements
thrust::transform(d_x.begin(), d_x.end(),
                  d_y.begin(), d_y.begin(),
                  2.0f * _1 + _2);
```

http://thrust.github.com

SAXPY: CUDA FORTRAN

5

Standard Fortran

```
module mymodule contains
  subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    do i=1,n
      y(i) = a*x(i)+y(i)
    enddo
  end subroutine saxpy
end module mymodule

program main
  use mymodule
  real :: x(2**20), y(2**20)
  x = 1.0, y = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy(2**20, 2.0, x, y)

end program main
```

Parallel Fortran

```
module mymodule contains
  attributes(global) subroutine saxpy(n, a, x, y)
    real :: x(:), y(:), a
    integer :: n, i
    attributes(value) :: a, n
    i = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if (i<=n) y(i) = a*x(i)+y(i)
  end subroutine saxpy
end module mymodule

program main
  use cudafor; use mymodule
  real, device :: x_d(2**20), y_d(2**20)
  x_d = 1.0, y_d = 2.0

  ! Perform SAXPY on 1M elements
  call saxpy<<<4096,256>>>(2**20, 2.0, x_d, y_d)

end program main
```

http://developer.nvidia.com/cuda-fortran

SAXPY: PYTHON

6

Standard Python

```
import numpy as np


def saxpy(a, x, y):
  return [a * xi + yi
          for xi, yi in zip(x, y)]

x = np.arange(2**20, dtype=np.float32)
y = np.arange(2**20, dtype=np.float32)





cpu_result = saxpy(2.0, x, y)
```

http://numpy.scipy.org

Numba: Parallel Python

```
import numpy as np
from numba import vectorize

@vectorize(['float32(float32, float32,
float32)'], target='cuda')
def saxpy(a, x, y):
    return a * x + y

N = 1048576

# Initialize arrays
A = np.ones(N, dtype=np.float32)
B = np.ones(A.shape, dtype=A.dtype)
C = np.empty_like(A, dtype=A.dtype)

# Add arrays onGPU
C = saxpy(2.0, A, B)
```

https://numba.pydata.org

ENABLING ENDLESS WAYS TO SAXPY

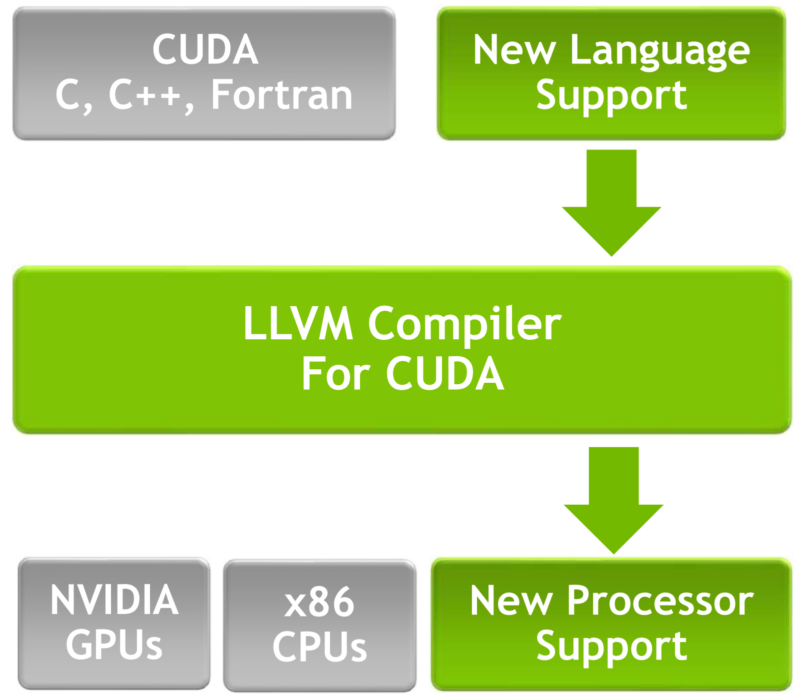
Developers want to build front-ends for:

- Java, Python, R, DSLs

Target other processors like:

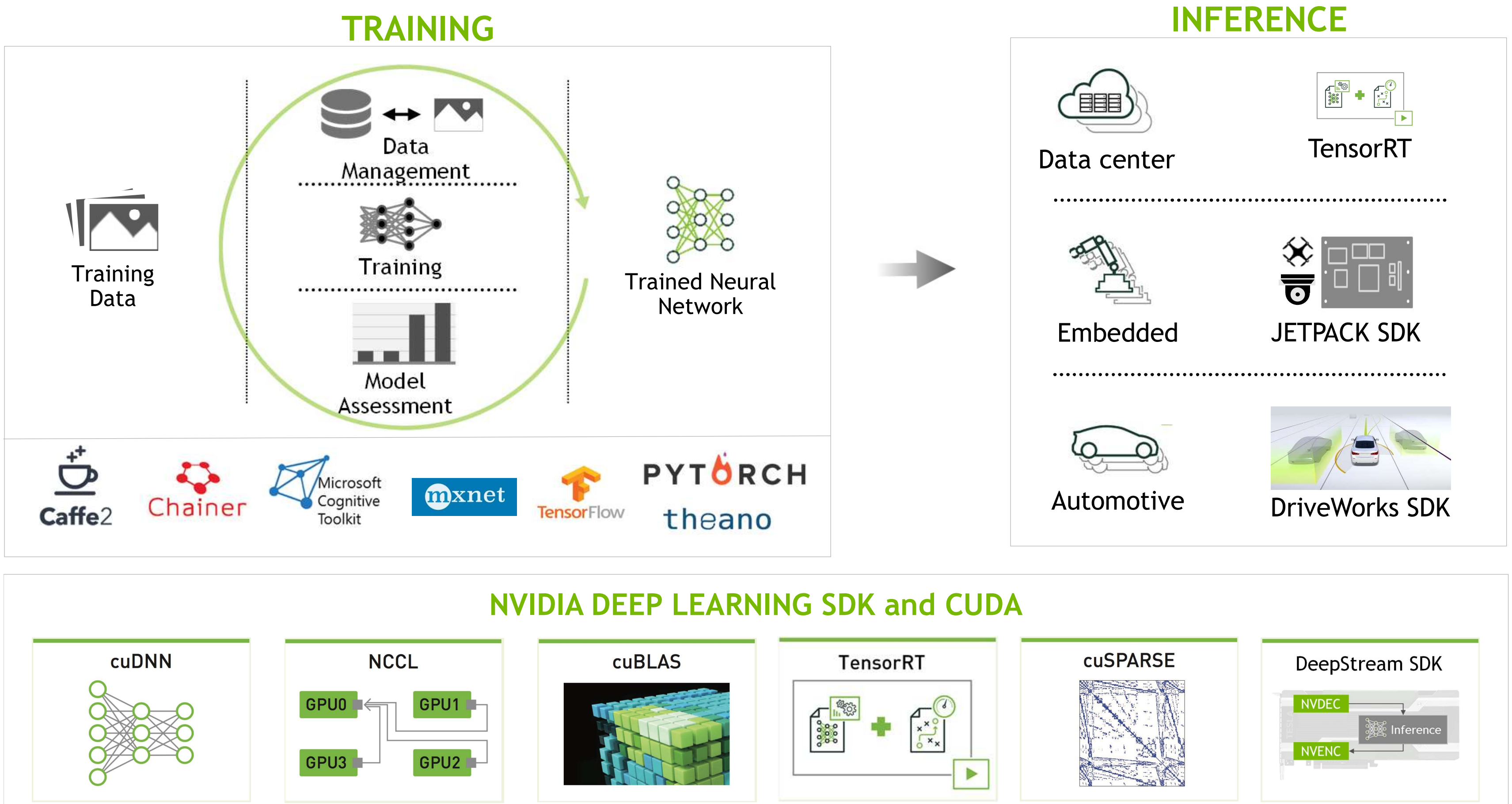
- ARM, FPGA, GPUs, x86

CUDA Compiler Contributed to Open Source LLVM



GPU COMPUTING PLATFORM FOR DL/ML

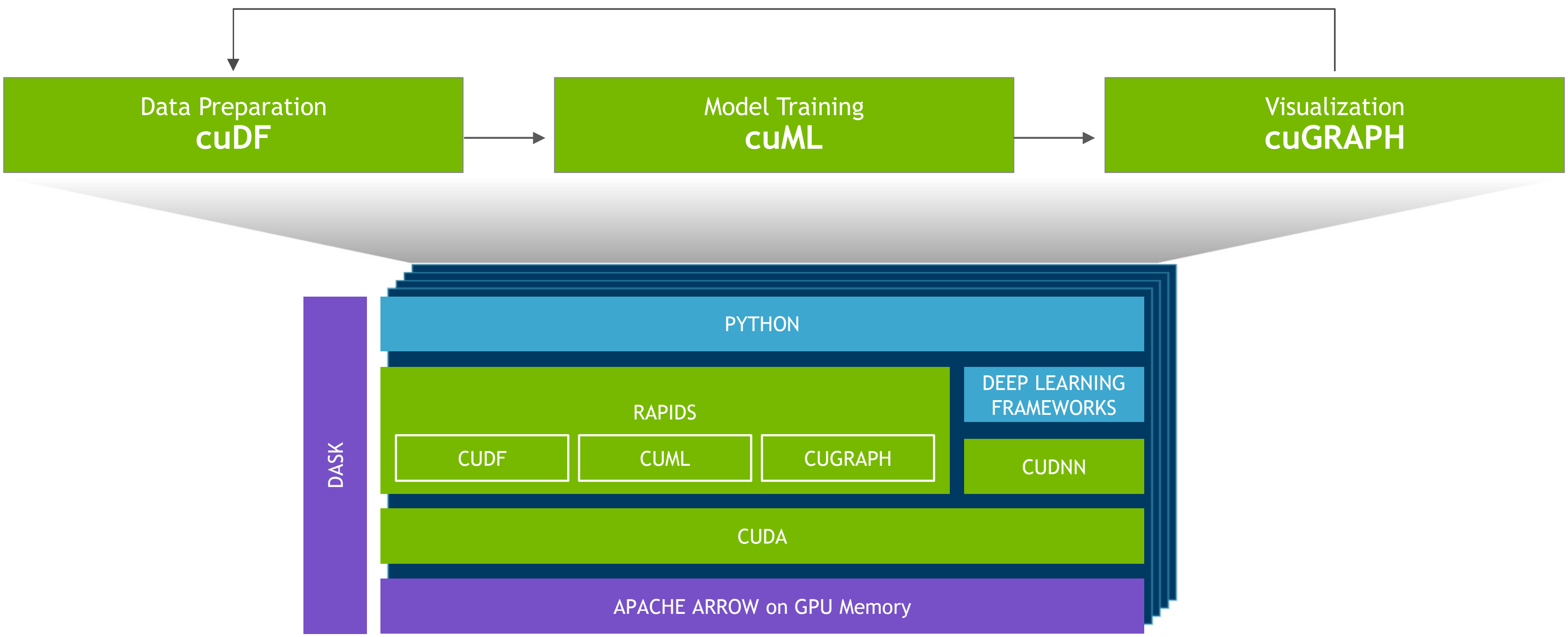
NVIDIA DEEP LEARNING SOFTWARE STACK



developer.nvidia.com/deep-learning-software

RAPIDS — OPEN GPU DATA SCIENCE

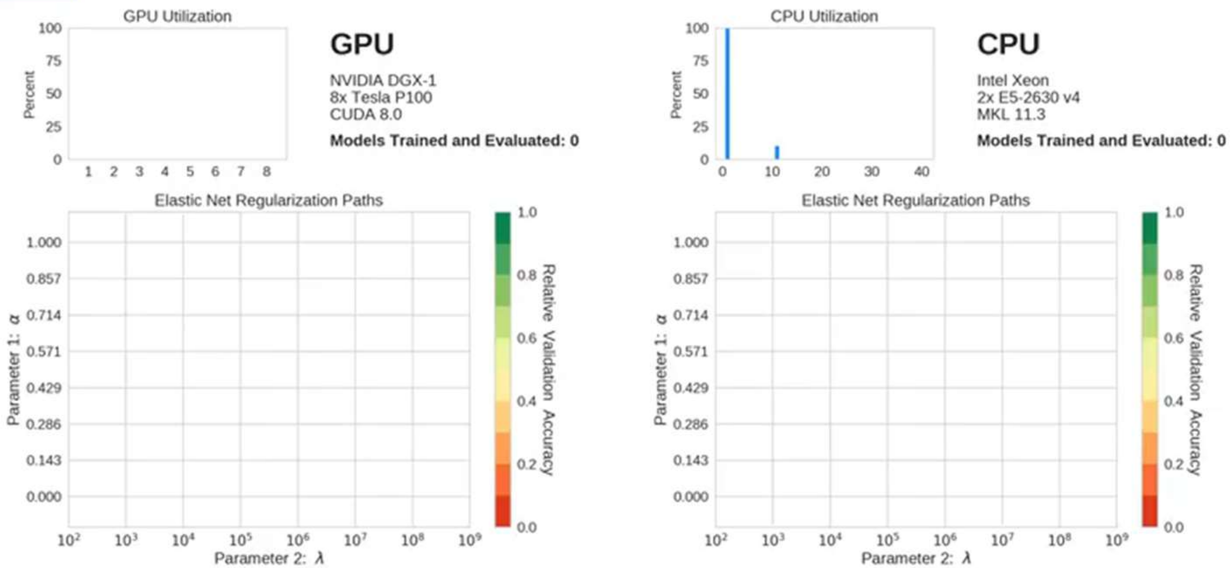
Software Stack Python



WHY RAPIDS

World's Fastest Machine Learning

H2O.ai Machine Learning — Generalized Linear Modeling



U.S. Census dataset (predict Income): 45k rows, 10k cols

Parameters: 5-fold cross-validation, $\alpha = \{\frac{i}{7}, i = 0 \ldots 7\}$, full λ-search

NGC FOR EFFICIENCY

CHALLENGES UTILIZING AI & HPC SOFTWARE

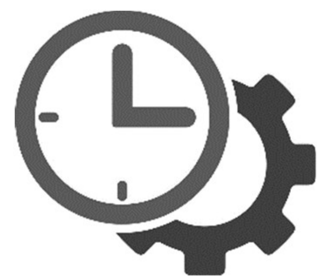
Installation

Complex, time consuming, and error-prone

Optimization

Requires expertise to optimize framework performance

Maintenance



IT can’t keep up with frequent software upgrades

Productivity

Users limited to older features and lower performance

NGC

The GPU-Optimized Software Hub

Simplify Deployments with Performance-optimized Containers

Innovate Faster with Ready-to-Use Solutions

Deploy Anywhere

https://www.nvidia.com/gpu-cloud/

SUMMARY

- Full Stack Optimization is key to performance
- Multiple choices for programming on GPU
- One is not an alternative to other. They co-exisit
- Universal hardware with Software stack is key to GPU computing

Thank You