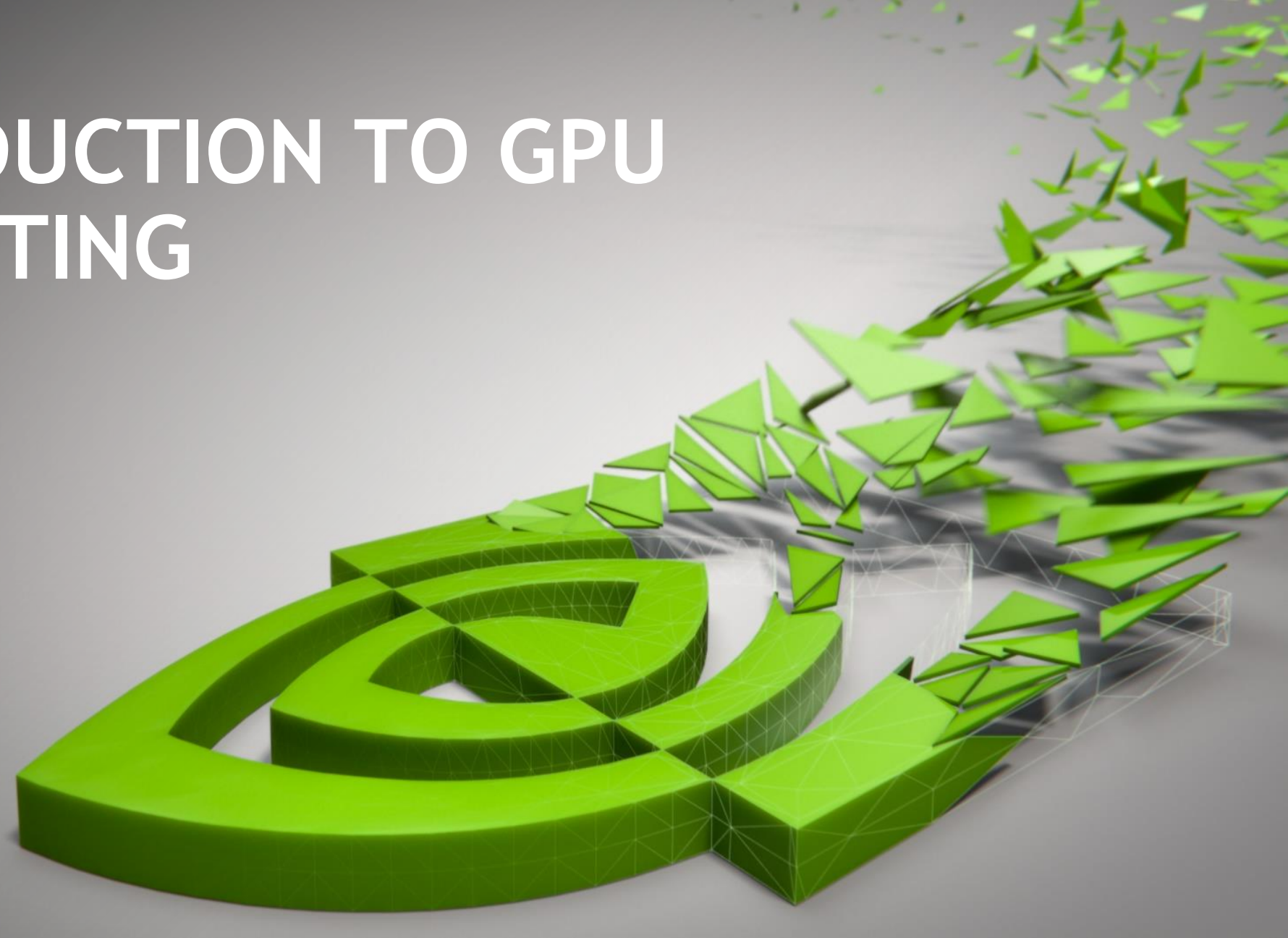
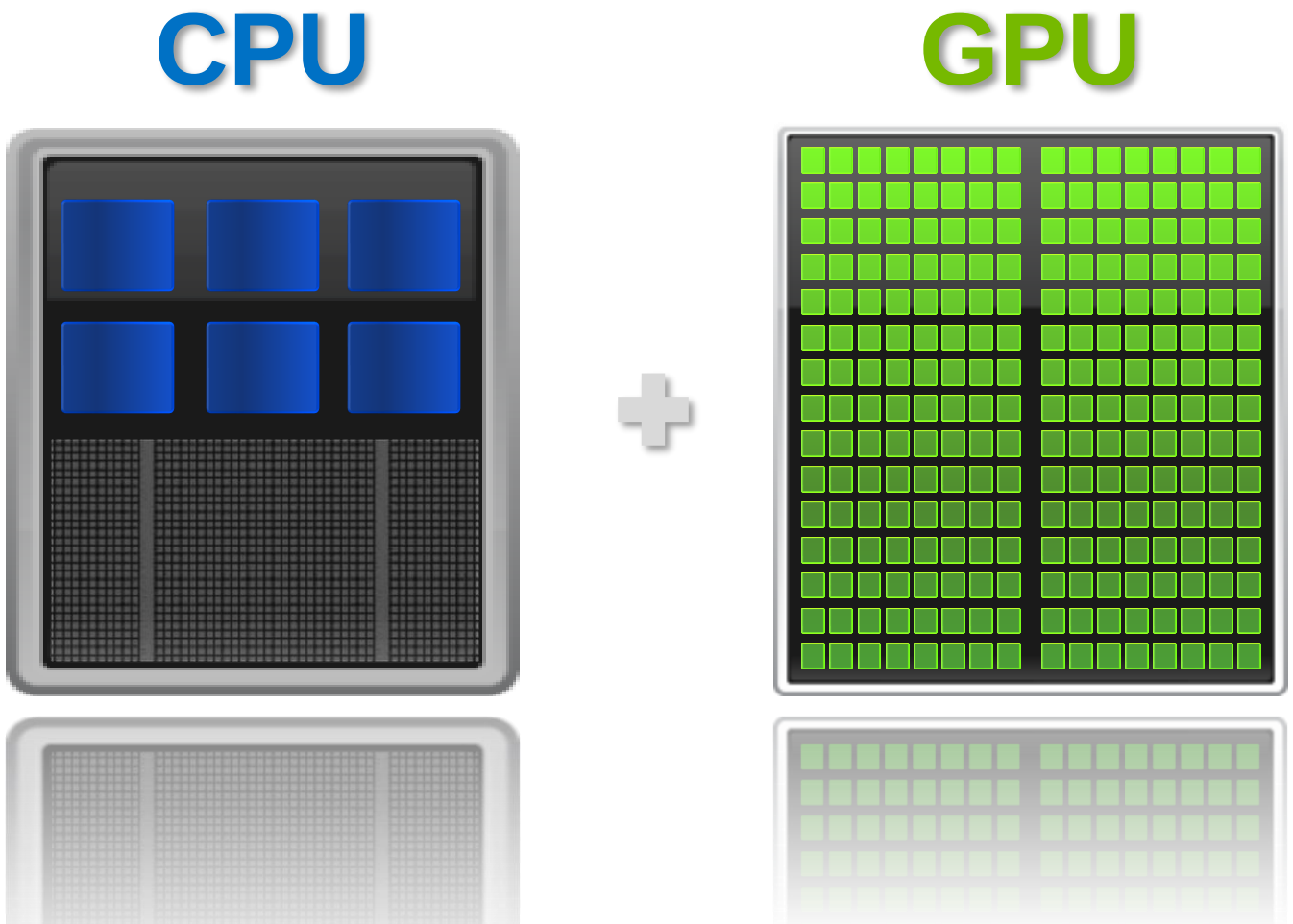
INTRODUCTION TO GPU COMPUTING

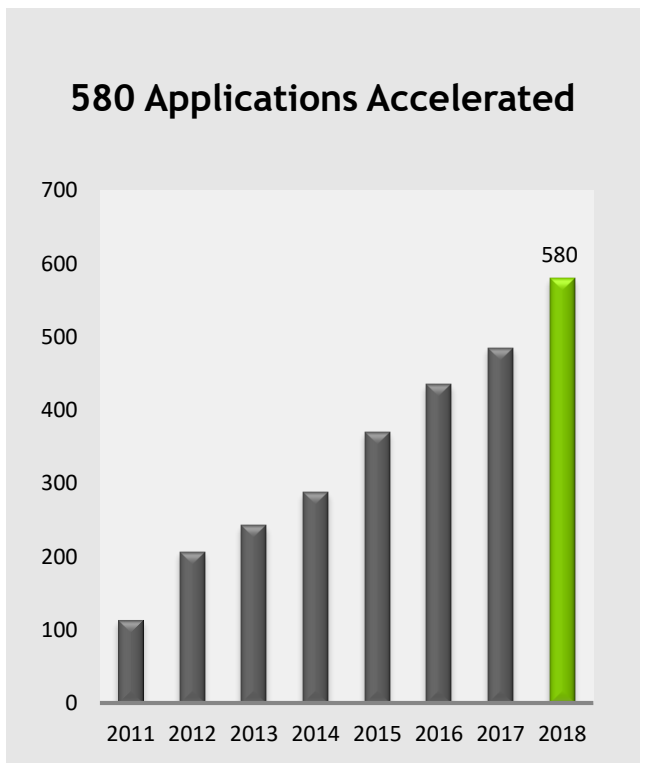
February 28, 2020

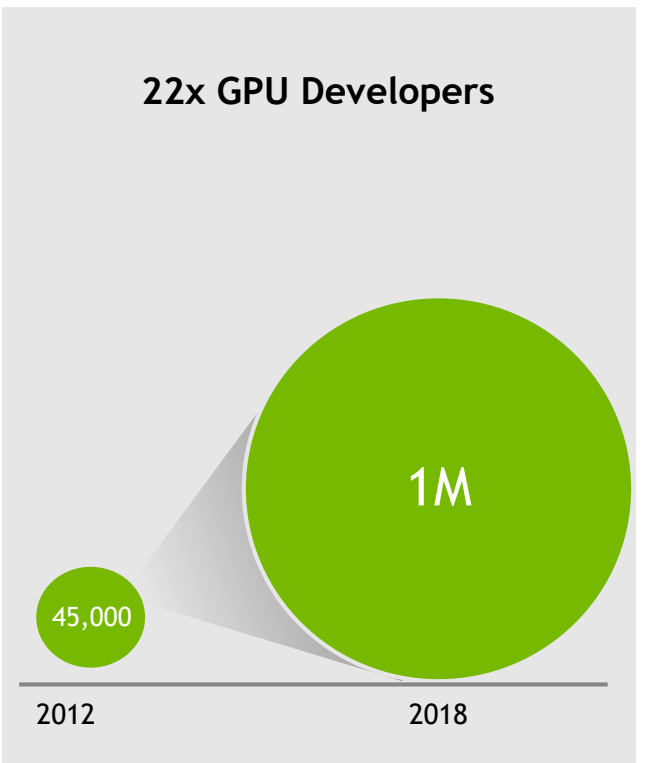


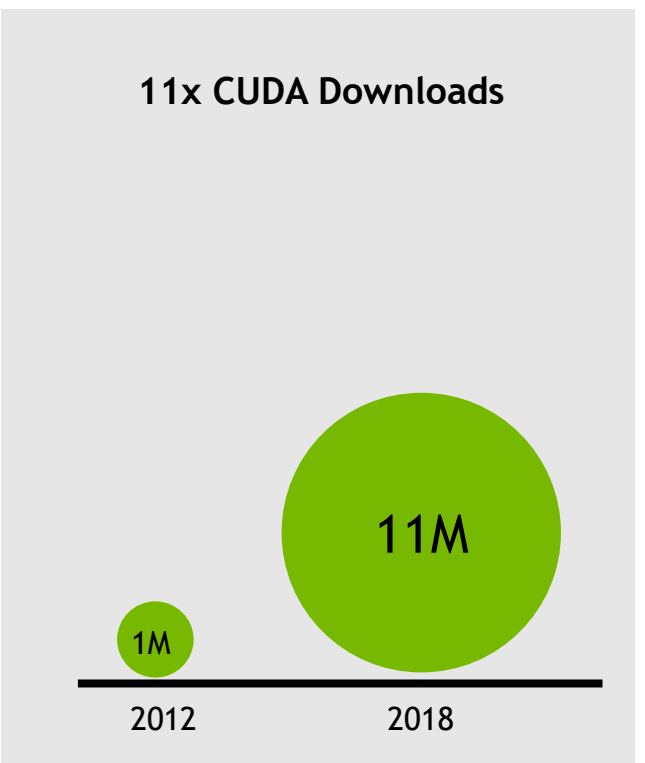
Add GPUs: Accelerate Science Applications



ACCELERATED COMPUTING IS GROWING RAPIDLY



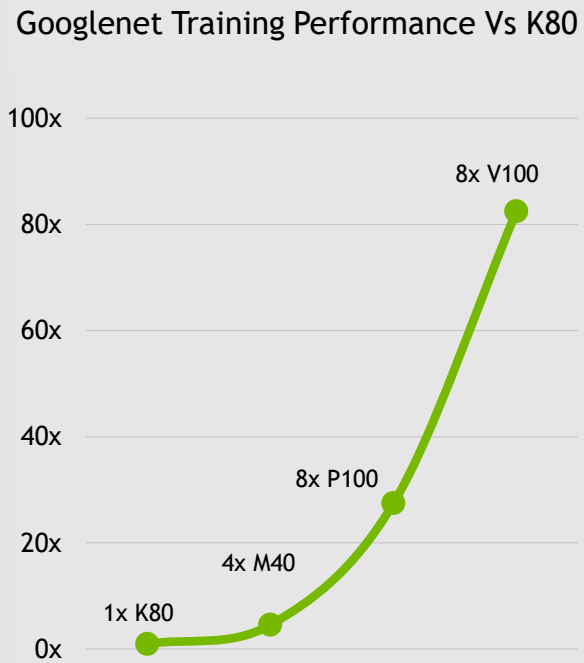




CUDA POWERS ACCELERATED COMPUTING

Caffe2 · Chainer · PYTORCH · mxnet · TensorFlow · theano · ANSYS FLUENT · GAUSSIAN · SIMULIA Abaqus · GROMACS FAST. FLEXIBLE. FREE. · VASP · OpenFOAM The Open Source CFD Toolbox · LS-DYNA · NAMD Scalable Molecular Dynamics · +400 More Applications

DL FRAMEWORKS AND HPC APPLICATIONS ACCELERATED



Over 80x in 3 Years

127 Systems on Top 500

World's #1 Summit: 144 PF

World's #2 Sierra: 95 PF

Europe's #1 Piz Daint: 21 PF

Japan's #1 ABCI: 20 PF

Industrial #1 ENI: 12 PF

DEFINING THE NEXT GIANT WAVE IN HPC

END-TO-END PRODUCT FAMILY

HPC / TRAINING

INFERENCE

FULLY INTEGRATED AI SYSTEMS

DGX-1

DGX-2

DESKTOP

TITAN/ GeForce

WORKSTATION

DGX Station

VIRTUAL WORKSTATION

Virtual GPU

DATA CENTER

Tesla V100/T4

SERVER PLATFORM

HGX1/ HGX2

DATA CENTER

Tesla T4

Tesla V100

AUTOMOTIVE

Drive AGX Pegasus

EMBEDDED

Jetson AGX Xavier

TESLA V100 TENSOR CORE GPU

World's Most Powerful Data Center GPU

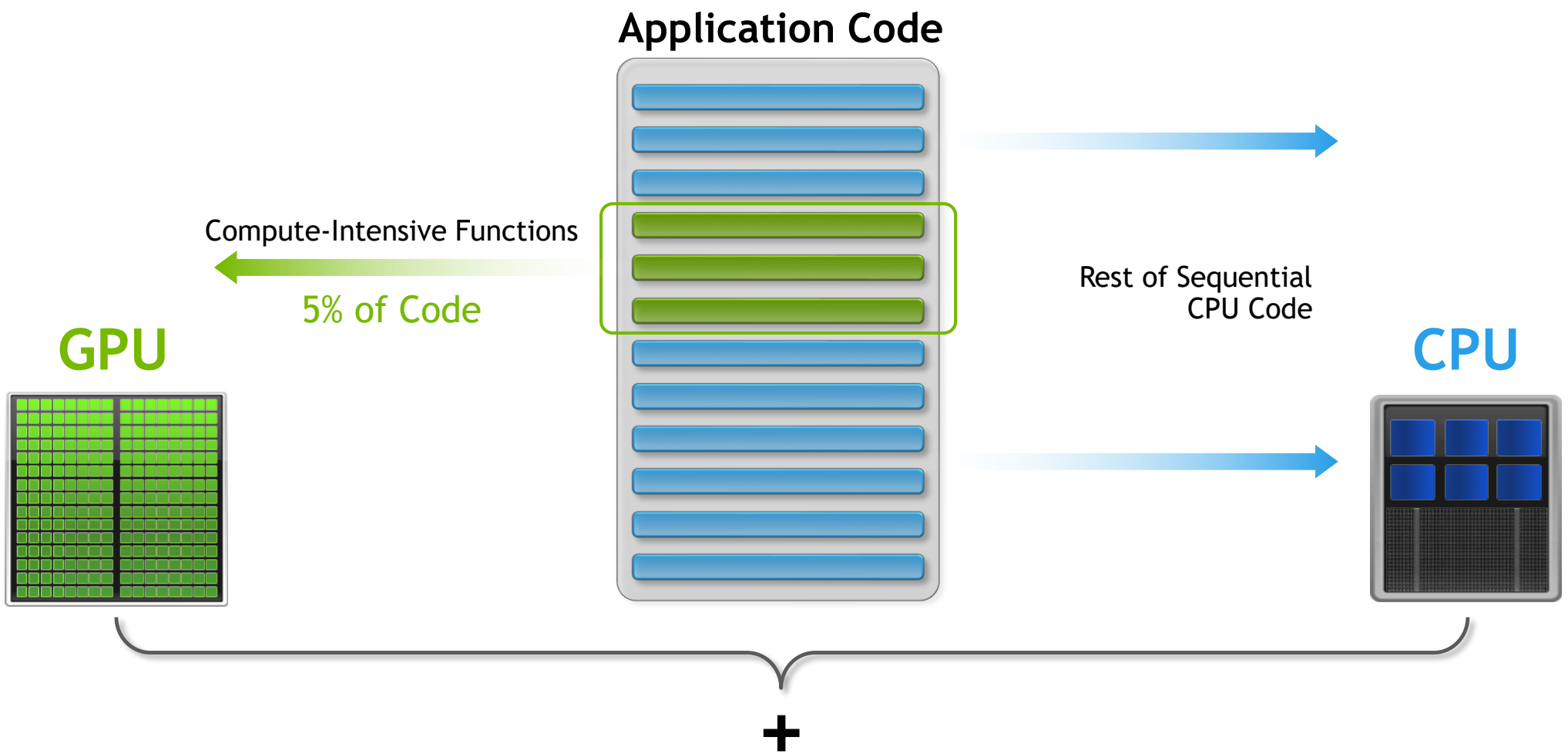
5,120 CUDA cores
640 NEW Tensor cores
7.8 FP64 TFLOPS | 15.7 FP32 TFLOPS | 125 Tensor TFLOPS
20MB SM RF | 16MB Cache
32 GB HBM2 @ 900GB/s | 300GB/s NVLink

TESLA T4

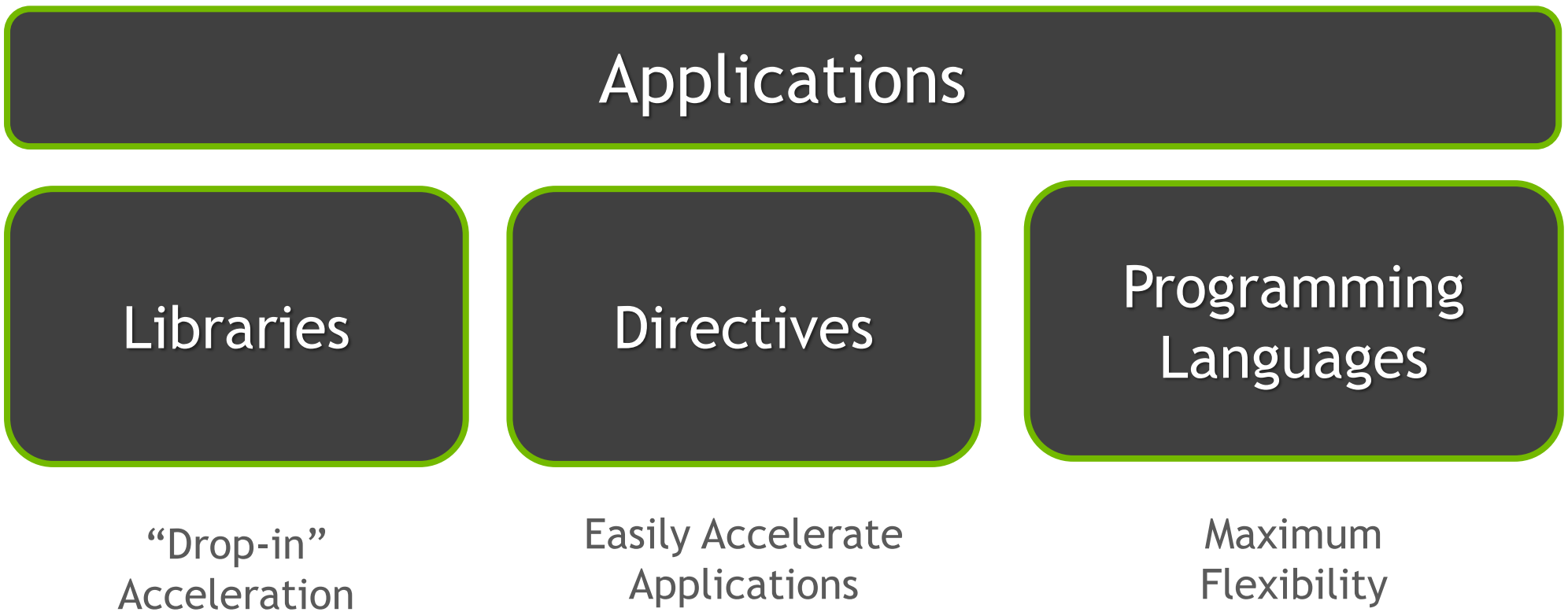
WORLD'S MOST EFFICIENT GPU FOR MAINSTREAM SERVERS

320 Turing Tensor Cores
2,560 CUDA Cores
65 FP16 TFLOPS | 130 INT8 TOPS | 260 INT4 TOPS
16GB | 320GB/s
70 W

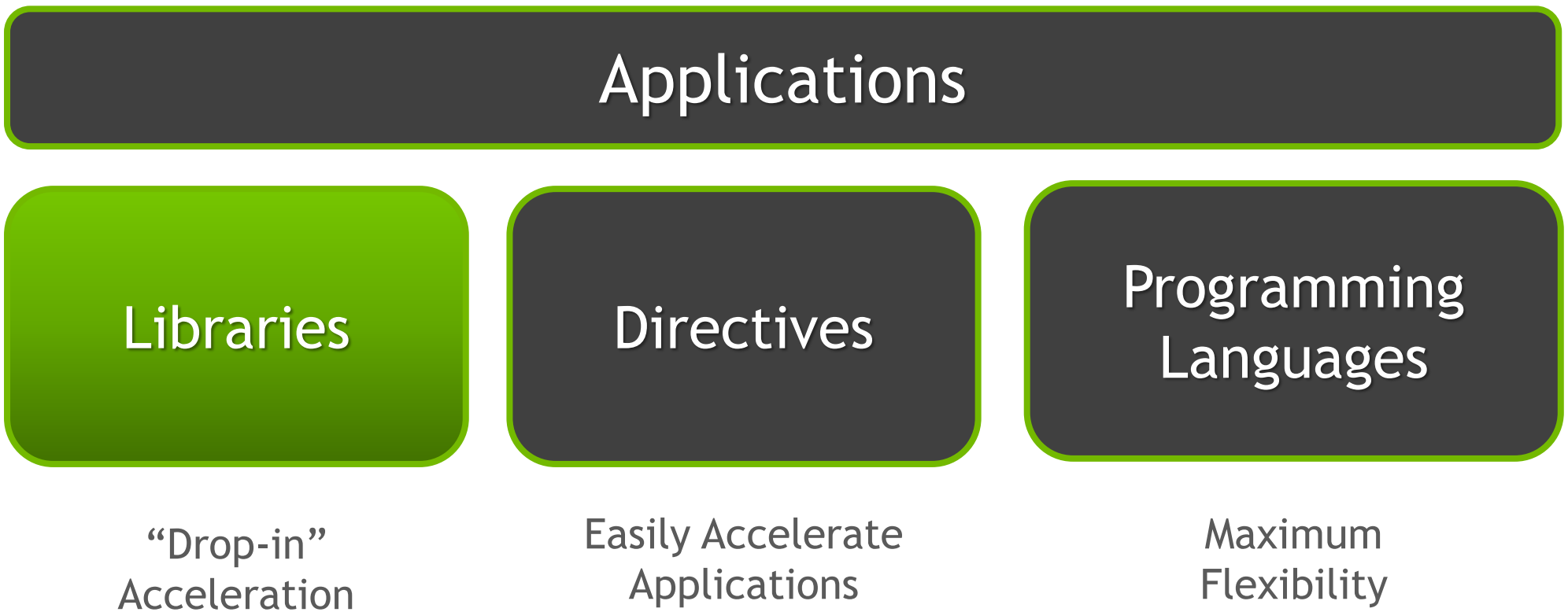
SMALL CHANGES, BIG SPEED-UP



3 WAYS TO ACCELERATE APPLICATIONS



3 WAYS TO ACCELERATE APPLICATIONS



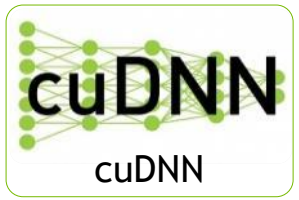
LIBRARIES: EASY, HIGH-QUALITY ACCELERATION

EASE OF USE	Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
"DROP-IN"	Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
QUALITY	Libraries offer high-quality implementations of functions encountered in a broad range of applications
PERFORMANCE	NVIDIA libraries are tuned by experts

GPU ACCELERATED LIBRARIES

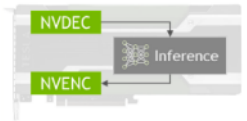
“Drop-in” Acceleration for Your Applications

DEEP LEARNING



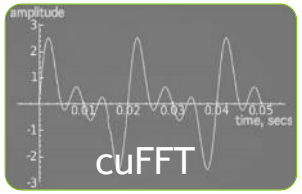
cuDNN


TensorRT



DeepStream SDK

SIGNAL, IMAGE & VIDEO



cuFFT


NVIDIA NPP


CODEC SDK

LINEAR ALGEBRA


cuBLAS

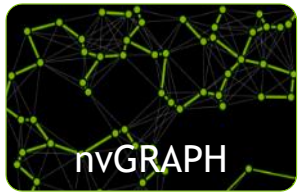

cuSPARSE

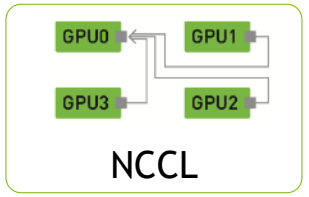

CUDA
Math library


cuSOLVER


cuRAND

PARALLEL ALGORITHMS


nvGRAPH



NCCL

3 STEPS TO CUDA-ACCELERATED APPLICATION

Step 1: Substitute library calls with equivalent CUDA library calls

```
saxpy ( … ) ► cublasSaxpy ( … )
```

Step 2: Manage data locality

- with CUDA: `cudaMalloc(), cudaMemcpy(), etc.`
- with CUBLAS: `cublasAlloc(), cublasSetVector(), etc.`

Step 3: Rebuild and link the CUDA-accelerated library

```
gcc myobj.o –l cublas
```

DROP-IN ACCELERATION (STEP 1)

```
int N = 1 << 20;









// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

DROP-IN ACCELERATION (STEP 1)

```
int N = 1 << 20;









// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

Add “cublas” prefix
and use device variables

DROP-IN ACCELERATION (STEP 2)

```
int N = 1 << 20;
cublasInit();




// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);




cublasShutdown();
```

Initialize cuBLAS

Shut down cuBLAS

DROP-IN ACCELERATION (STEP 2)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);



// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);



cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Allocate device vectors

Deallocate device vectors

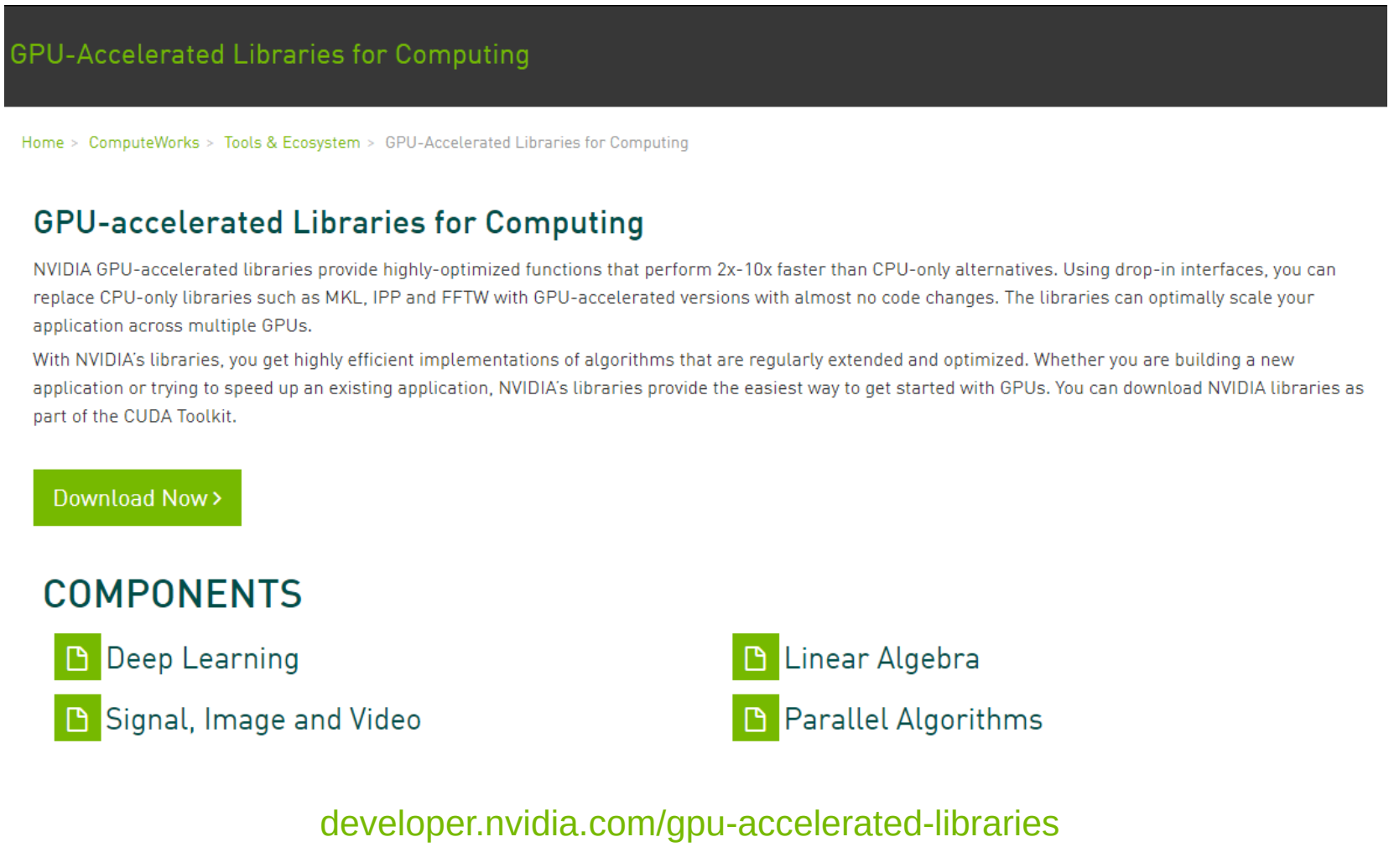
DROP-IN ACCELERATION (STEP 2)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);

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: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

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

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Transfer data to GPU

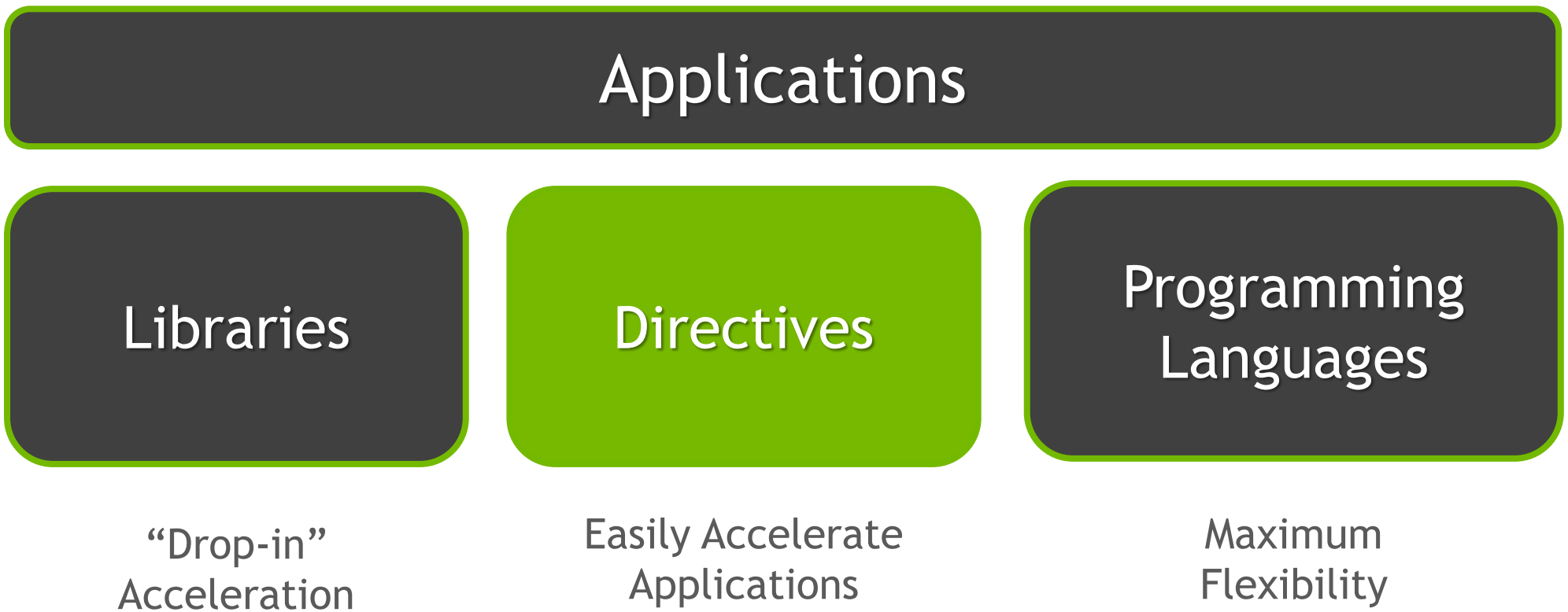
Read data back GPU

EXPLORE CUDA LIBRARIES



developer.nvidia.com/gpu-accelerated-libraries

3 WAYS TO ACCELERATE APPLICATIONS



OpenACC is a directives-based programming approach to **parallel computing** designed for **performance** and **portability** on CPUs and GPUs for HPC.

Add Simple Compiler Directive

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

SINGLE CODE FOR MULTIPLE PLATFORMS

OpenACC - Performance Portable Programming Model for HPC

POWER

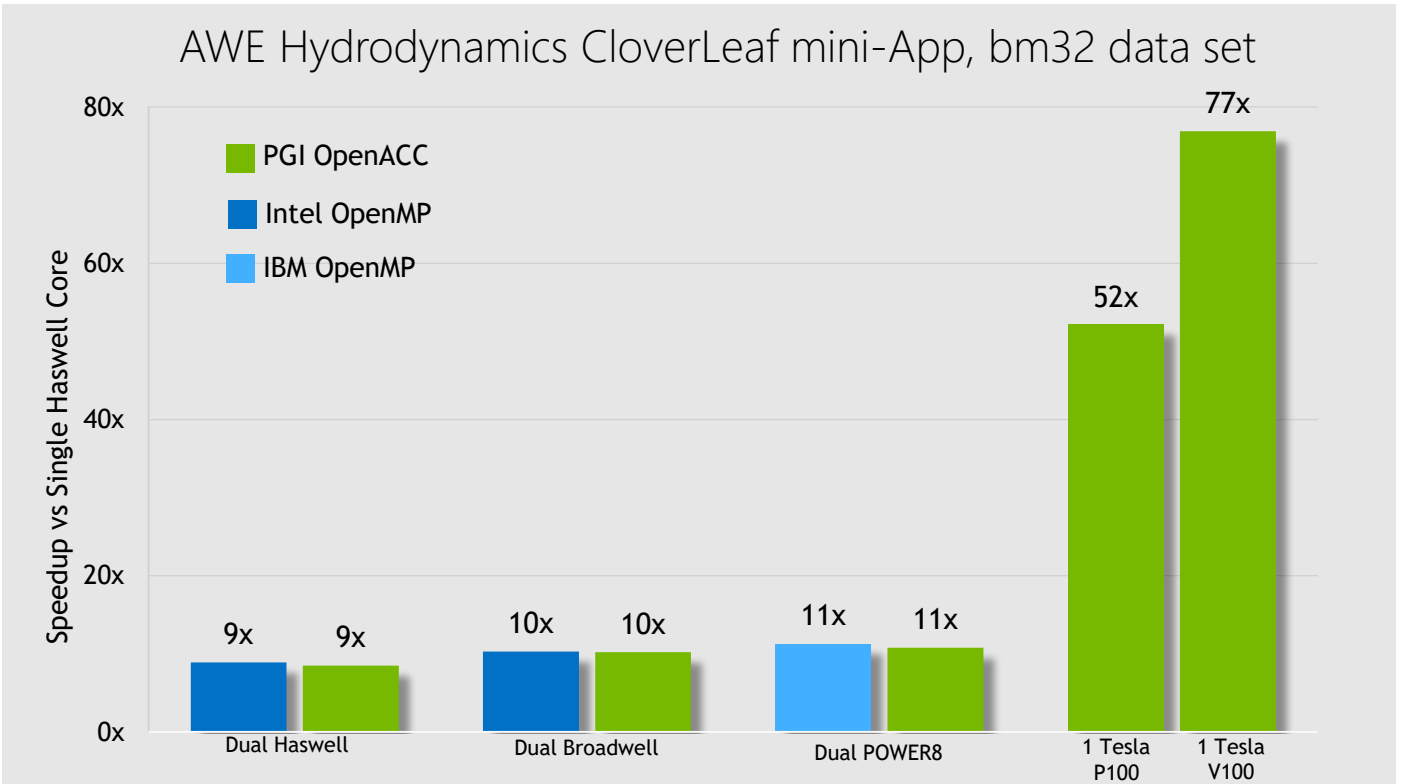
Sunway

x86 CPU

x86 Xeon Phi

NVIDIA GPU

PEZY-SC



Systems: Haswell: 2x16 core Haswell server, four K80s, CentOS 7.2 (perf-hsw10), Broadwell: 2x20 core Broadwell server, eight P100s (dgx1-prd-01), Minsky: POWER8+NVLINK, four P100s, RHEL 7.3 (gsn1).
Compilers: Intel 17.0, IBM XL 13.1.3, PGI 16.10.
Benchmark: CloverLeaf v1.3 downloaded from http://uk-mac.github.io/CloverLeaf the week of November 7 2016; CloverlLeaf_Serial; CloverLeaf_ref (MPI+OpenMP); CloverLeaf_OpenACC (MPI+OpenACC)
Data compiled by PGI November 2016, Volta data collected June 2017

TOP HPC APPS ADOPTING OPENACC

OpenACC - Performance Portability And Ease of Programming

ANSYS Fluent Gaussian
VASP

3 of Top 10 Apps

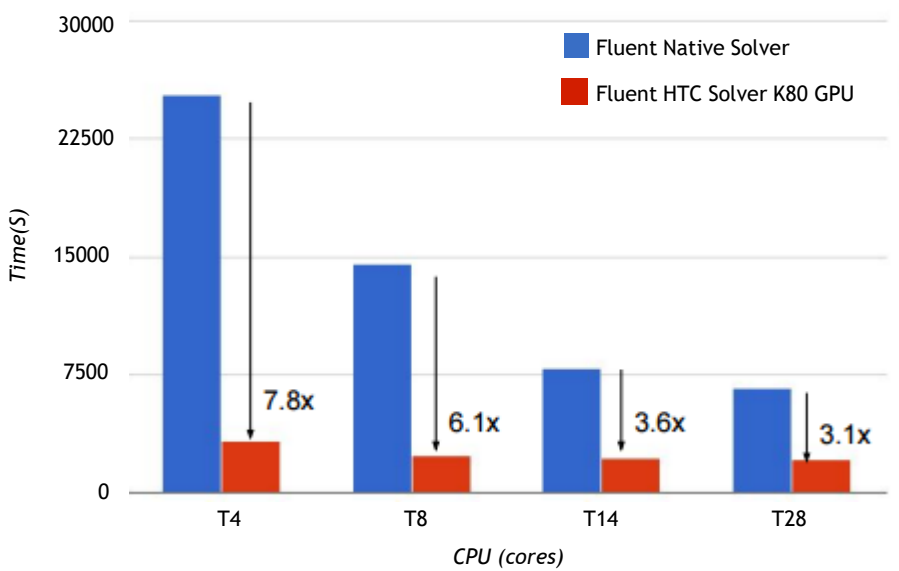
GTC
XGC
ACME
FLASH
LSDalton

5 ORNL CAAR Codes

COSMO
ELEPHANT
RAMSES
ICON
ORB5

5 CSCS Codes

ANSYS Fluent R18.0 Radiation Solver



CPU: (Haswell EP) Intel(R) Xeon(R) CPU E5-2695 v3 @2.30GHz, 2 sockets, 28 cores
GPU: Tesla K80 12+12 GB, Driver 346.46

2 BASIC STEPS TO GET STARTED

Step 1:

```
!$acc data copy(util1,util2,util3) copyin(ip,scp2,scp2i)
  !$acc parallel loop
  …
  !$acc end parallel
!$acc end data
```

Step 2:

```
pgf90 -ta=nvidia -Minfo=accel file.f
```

OpenACC DIRECTIVES EXAMPLE

```
!$acc data copy(A,Anew)
 iter=0
 do while ( err > tol .and. iter < iter_max )

  iter = iter +1
  err=0._fp_kind

!$acc kernels
   do j=1,m
    do i=1,n
     Anew(i,j) = .25_fp_kind *( A(i+1,j  ) + A(i-1,j  ) &
                               +A(i  ,j-1) + A(i  ,j+1))
     err = max( err, Anew(i,j)-A(i,j))
    end do
   end do
!$acc end kernels
   IF(mod(iter,100)==0 .or. iter == 1)    print *, iter, err
   A= Anew

 end do
!$acc end data
```

Copy arrays into GPU memory within data region

Parallelize code inside region

Close off parallel region

Close off data region, copy data back

OPENACC FOR EVERYONE

New PGI Community Edition Now Available

	FREE PGI Community Edition	PGI Professional Edition	PGI Enterprise Edition
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 Professional Services
LICENSE	Annual	Perpetual	Volume/Site

RESOURCES

FREE Compiler

Success stories

Guides

Tutorials

Videos

Courses

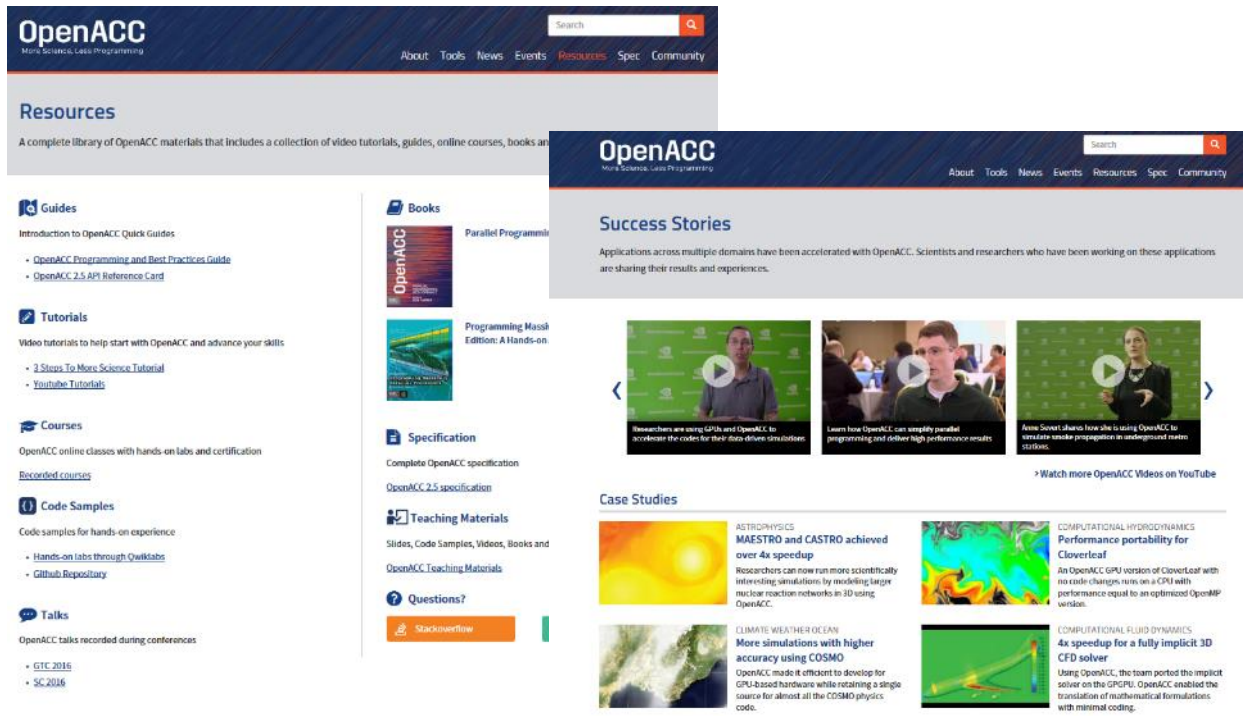
Code Samples

Talks

Books Specification

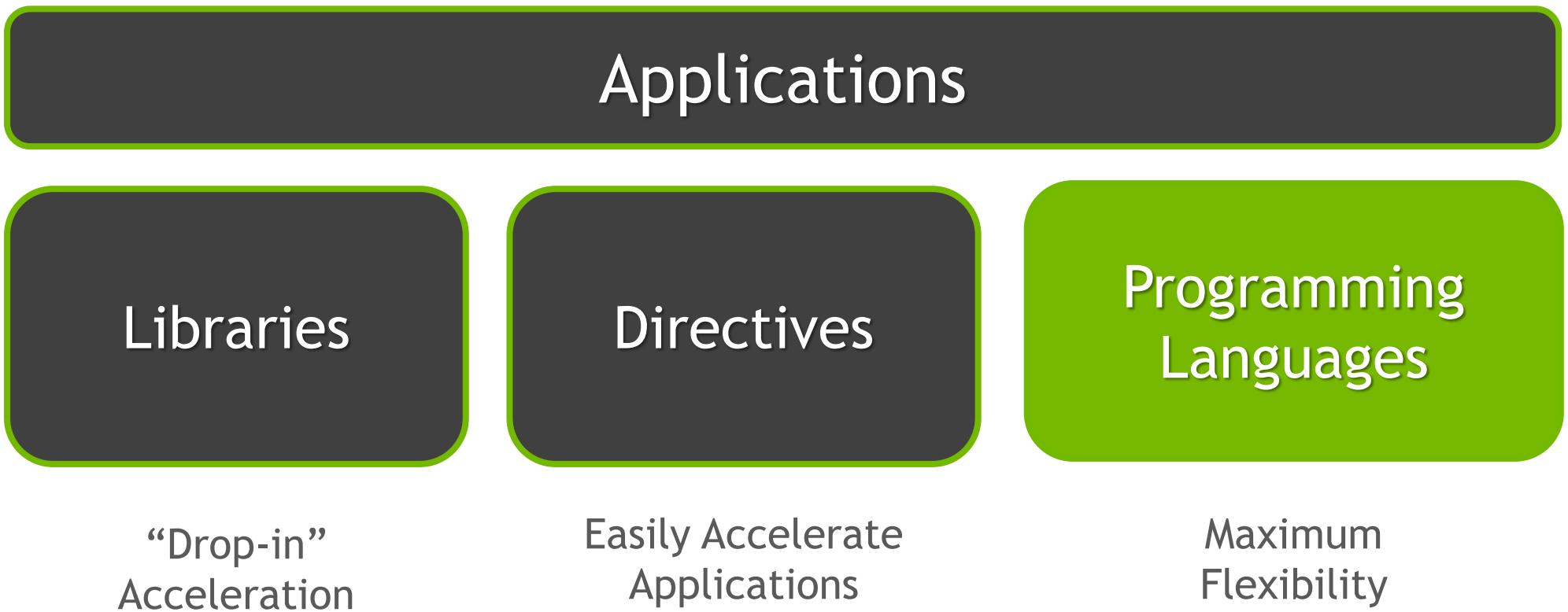
Teaching Materials

Slack&StackOverflow



Success stories: https://www.openacc.org/success-stories
Resources: https://www.openacc.org/resources
Free Compiler: https://www.pgroup.com/products/community.htm

3 WAYS TO ACCELERATE APPLICATIONS



GPU PROGRAMMING LANGUAGES

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	CUDA Fortran, OpenACC
C, C++	CUDA C++, OpenACC
Python	CUDA Python, PyCUDA
C#	Altimesh Hybridizer, Alea GPU

CUDA 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];
}

// Perform SAXPY on 1M elements
saxpy_serial(4096*256, 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];
}

// Perform SAXPY on 1M elements
saxpy_parallel<<<4096,256>>>(n,2.0,x,y);
```

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

CUDA C++: DEVELOP GENERIC PARALLEL CODE

CUDA C++ features enable sophisticated and flexible applications and middleware

- Class hierarchies
- __device__ methods
- Templates
- Operator overloading
- Functors (function objects)
- Device-side new/delete
- More…

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

```
template <typename T>
struct Functor {
  __device__ Functor(_a) : a(_a) {}
  __device__ T operator(T x) { return a*x; }
  T a;
}

template <typename T, typename Oper>
__global__ void kernel(T *output, int n) {
  Oper op(3.7);
  output = new T[n]; // dynamic allocation
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n)
    output[i] = op(i);  // apply functor
}
```

RAPID PARALLEL C++ DEVELOPMENT

- Resembles C++ STL
- High-level interface
  - Enhances developer productivity
  - Enables performance portability between GPUs and multicore CPUs
- Flexible
  - CUDA, OpenMP, and TBB backends
  - Extensible and customizable
  - Integrates with existing software
- Open source


```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);
// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;
// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());
// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

http://developer.nvidia.com/thrust or http://thrust.googlecode.com

CUDA FORTRAN

- Program GPU using Fortran
  - Key language for HPC
- Simple language extensions
  - Kernel functions
  - Thread / block IDs
  - Device & data management
  - Parallel loop directives
- Familiar syntax
  - Use allocate, deallocate
  - Copy CPU-to-GPU with assignment (=)

http://developer.nvidia.com/cuda-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
  call saxpy<<<4096,256>>>(2**20,3.0,x_d,y_d,)
  y = y_d
  write(*,*) 'max error=', maxval(abs(y-5.0))
end program main
```

GET STARTED TODAY

These languages are supported on all CUDA-capable GPUs.

You might already have a CUDA-capable GPU in your laptop or desktop PC!

CUDA C/C++
http://developer.nvidia.com/cuda-toolkit

CUDA Python
http://developer.nvidia.com/how-to-cuda-python

Thrust C++ Template Library
http://developer.nvidia.com/thrust

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

MATLAB
http://www.mathworks.com/discovery/matlab-gpu.html

Mathematica
http://www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support/


SIX WAYS TO SAXPY
Programming Languages for GPU Computing

SINGLE PRECISION ALPHA X PLUS Y (SAXPY)

Part of Basic Linear Algebra Subroutines (BLAS) Library

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

GPU SAXPY in multiple languages and libraries

A menagerie* of possibilities, not a tutorial

*technically, a *program chrestomathy*: http://en.wikipedia.org/wiki/Chrestomathy

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)
...
```

http://developer.nvidia.com/openacc or http://openacc.org

2

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

CUDA 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);
```

```
__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

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

CUDA FORTRAN

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

PYTHON

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

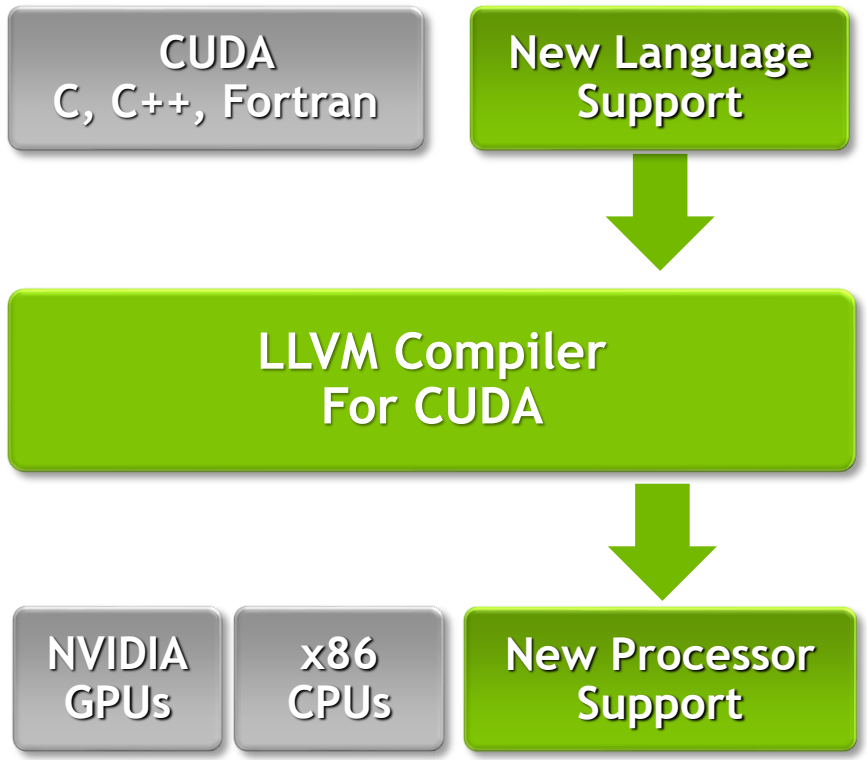
Numba Parallel Python

```
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, X, Y)
```

https://numba.pydata.org

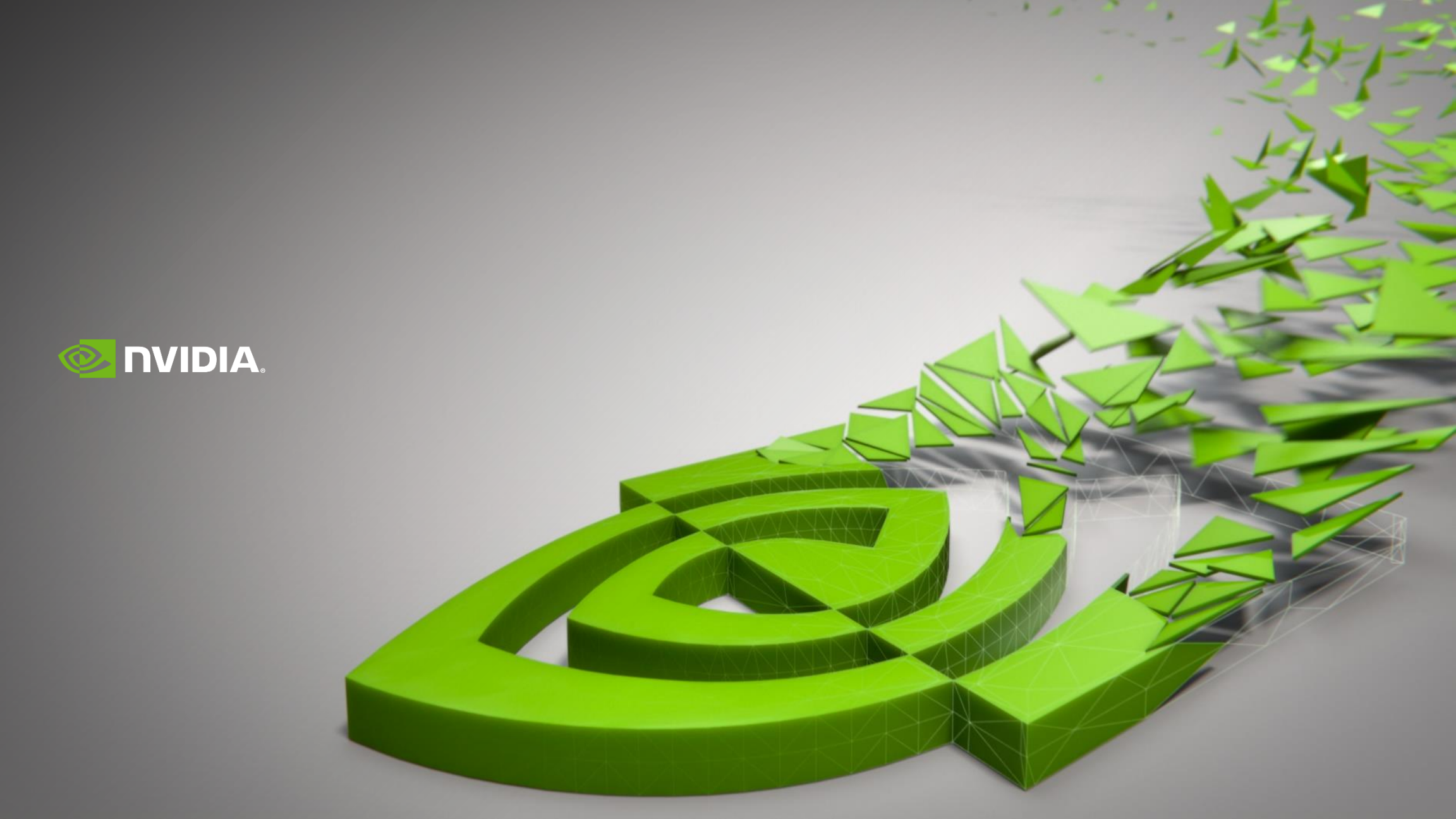
ENABLING ENDLESS WAYS TO SAXPY

- Build front-ends for Java, Python, R, DSLs
- Target other processors like ARM, FPGA, GPUs, x86



CUDA Compiler Contributed to Open Source LLVM

LLVM COMPILER INFRASTRUCTURE


NVIDIA