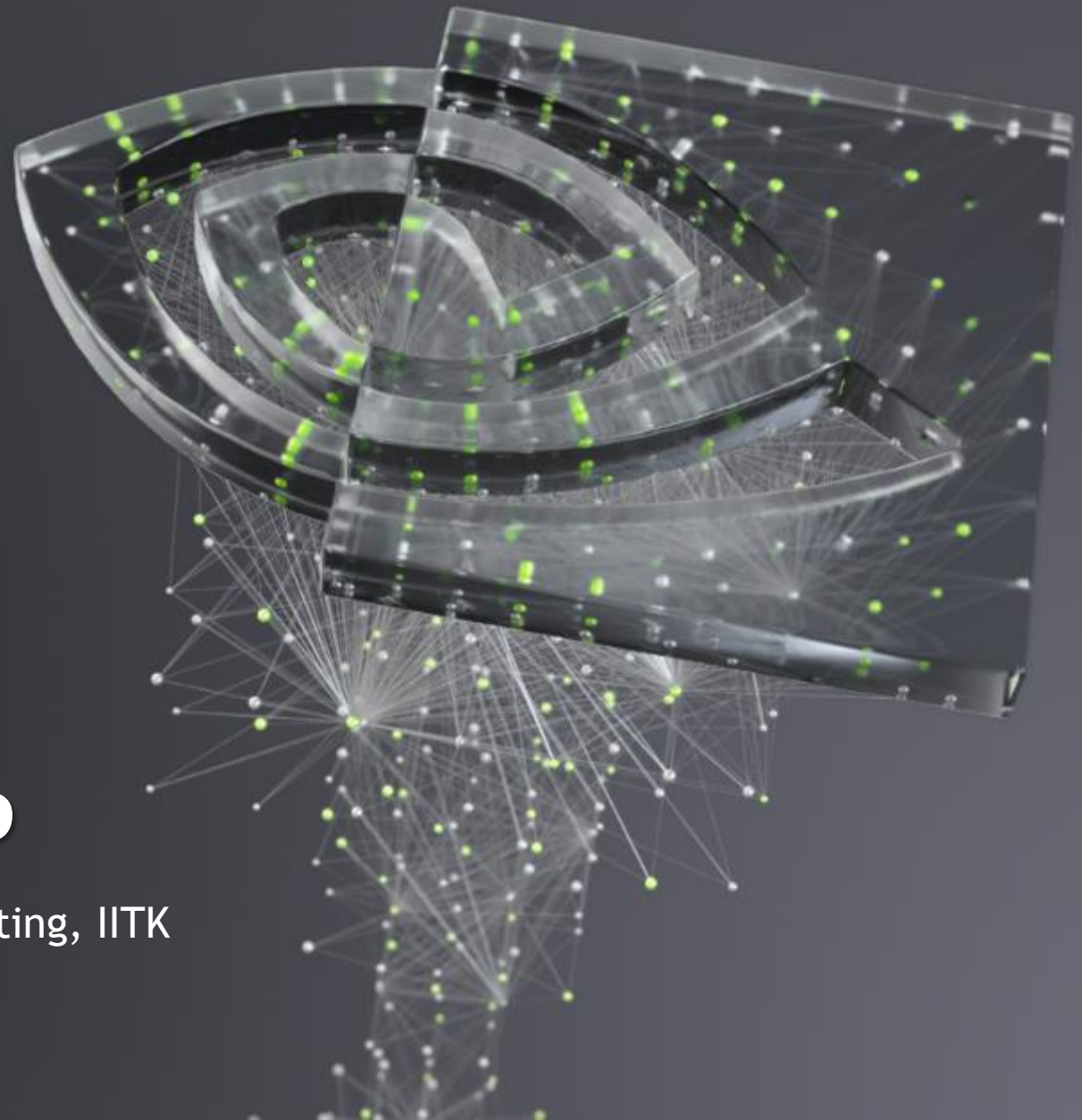


GPU BOOTCAMP

Winter School on High Performance Computing, IITK
Dec, 2019

GPU BOOTCAMP

What to expect?

- Fundamentals of GPU Computing
- Ways to GPU Programming
- Hands-on Session: From Simple to Real World Application Porting

INTRODUCTION TO GPU COMPUTING

What to expect?

- Parallel Programming: Perspective
- Evolution of Computing
- Fundamentals of GPU Architecture
- Broad view on GPU Stack
- Ways to GPU Computing
- Good starting point

WHAT IS PARALLEL PROGRAMMING?

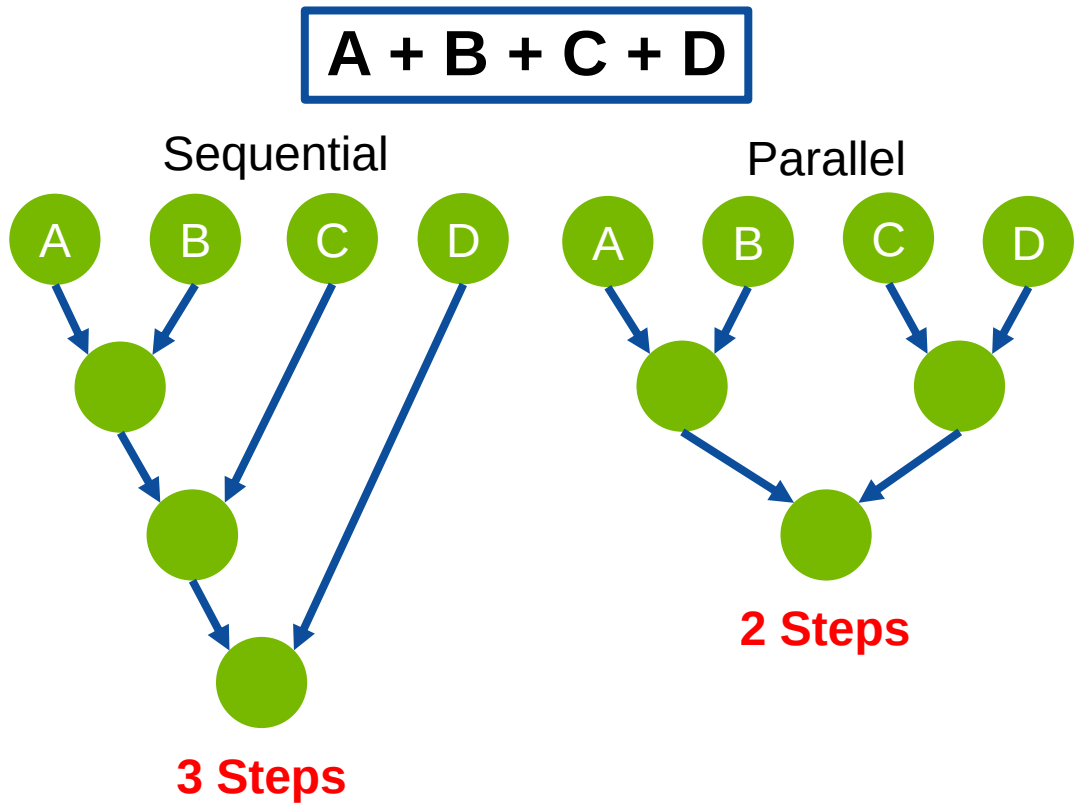
“Performance Programming”

Parallel programming involves exposing an algorithm’s ability to execute in parallel

This may involve breaking a large operation into smaller tasks (task parallelism)

Or doing the same operation on multiple data elements (data parallelism)

Parallel execution enables better performance on modern hardware



WHAT IS PARALLEL PROGRAMMING?

A real world example

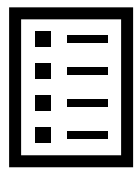
A professor and his 3 teaching assistants (TA) are grading 1,000 student exams

This exam has 8 questions on it

Let's assume it takes 1 minute to grade 1 question on 1 exam

To maintain fairness, if someone grades a question (for example, question #1) then they must grade that question on all other exams

The following is a sequential version of exam grading

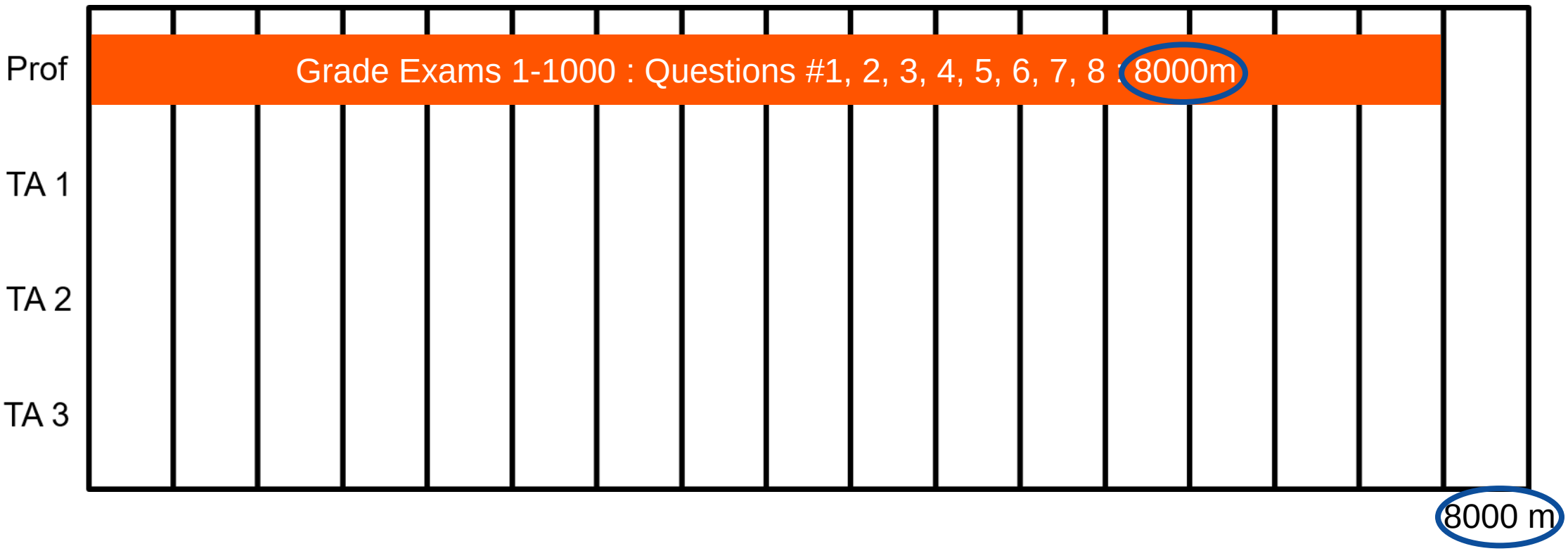


x1000

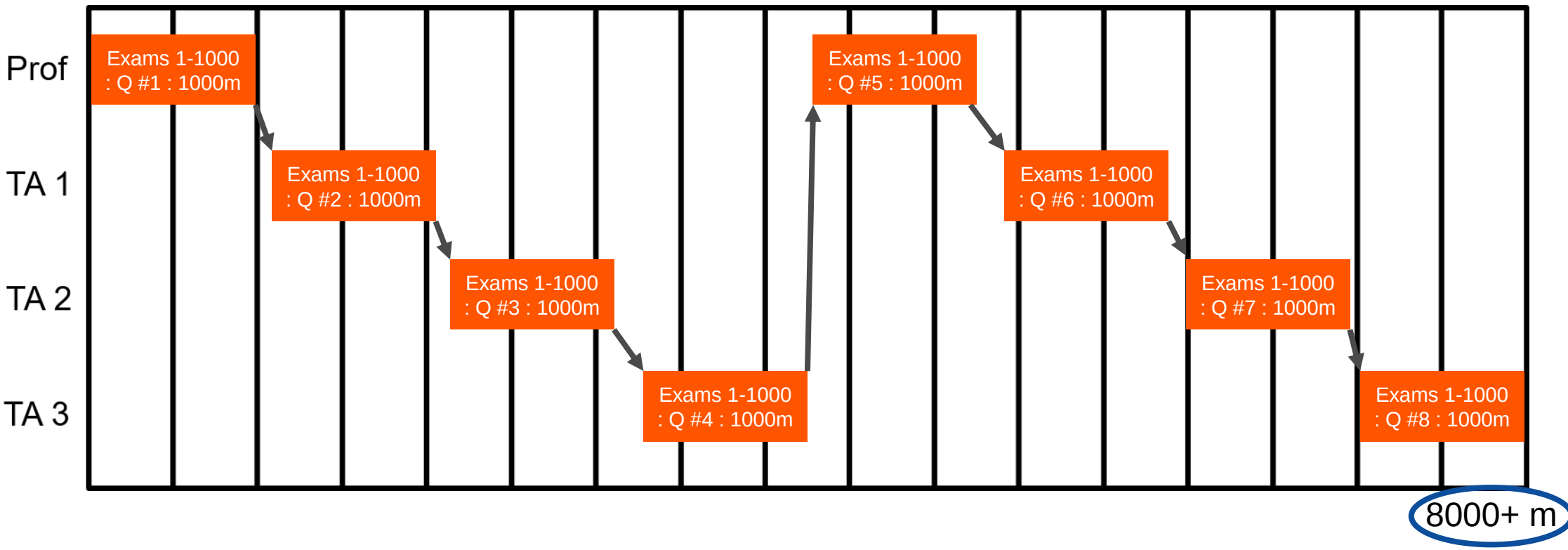
8 questions per exam
8,000 questions in total

1 minute per question

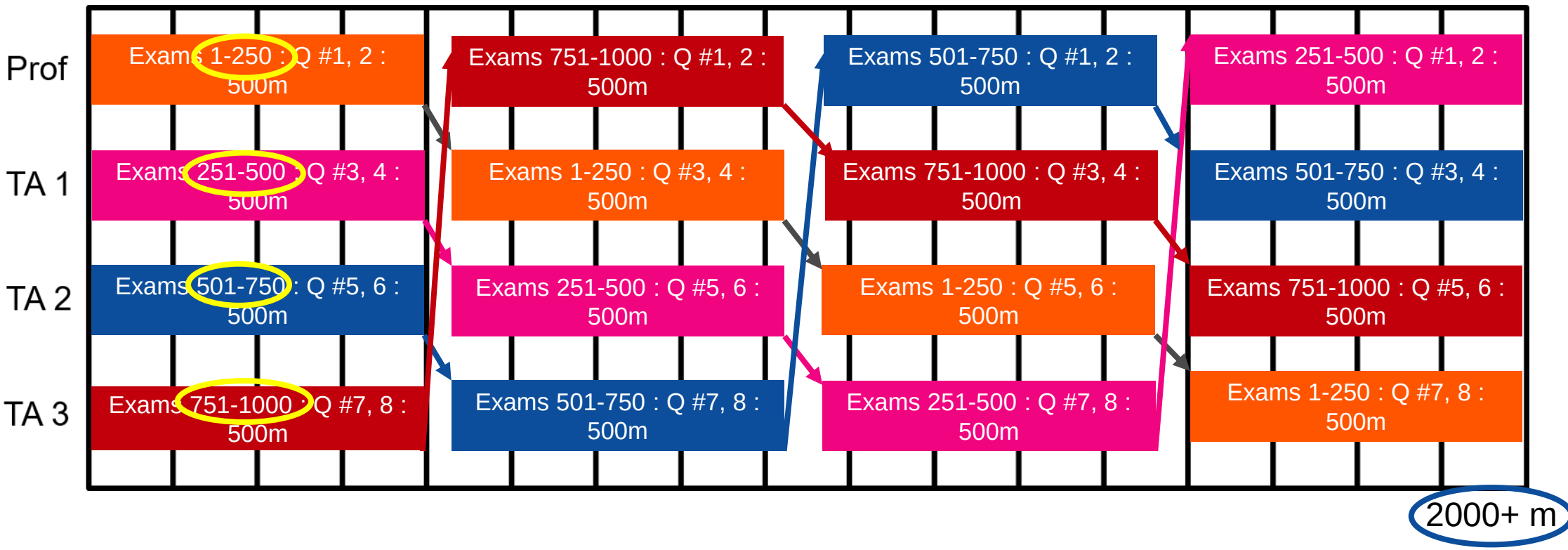
SEQUENTIAL SOLUTION



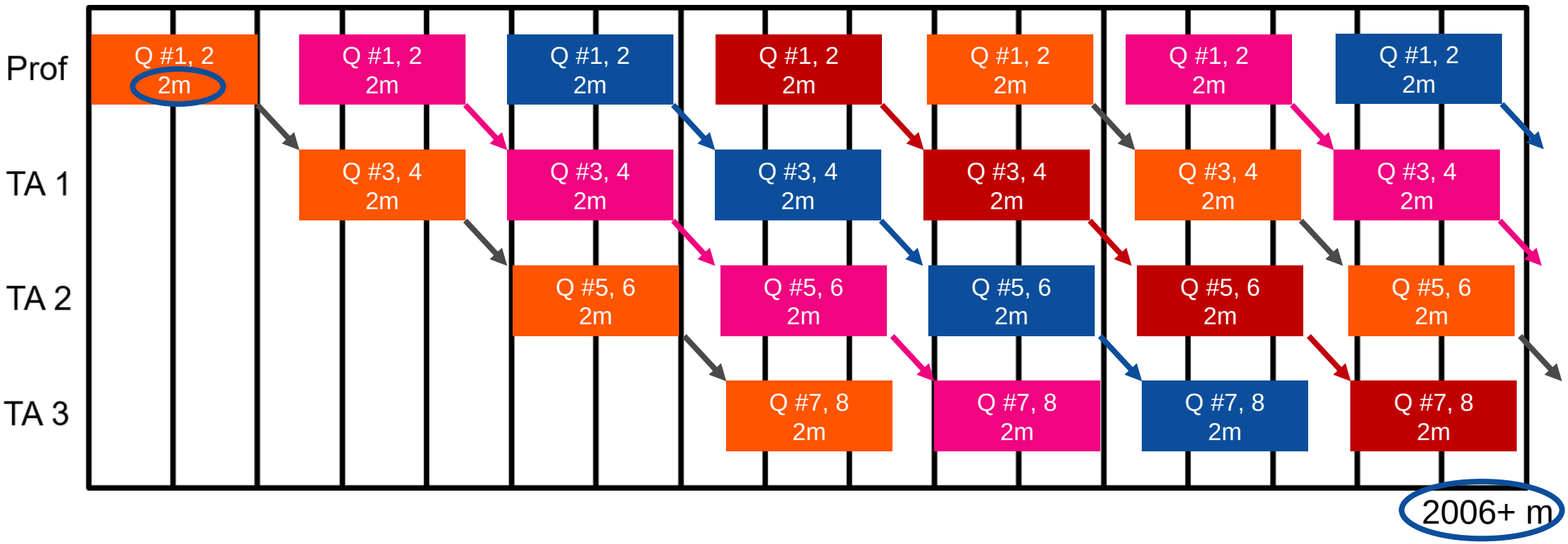
SEQUENTIAL SOLUTION



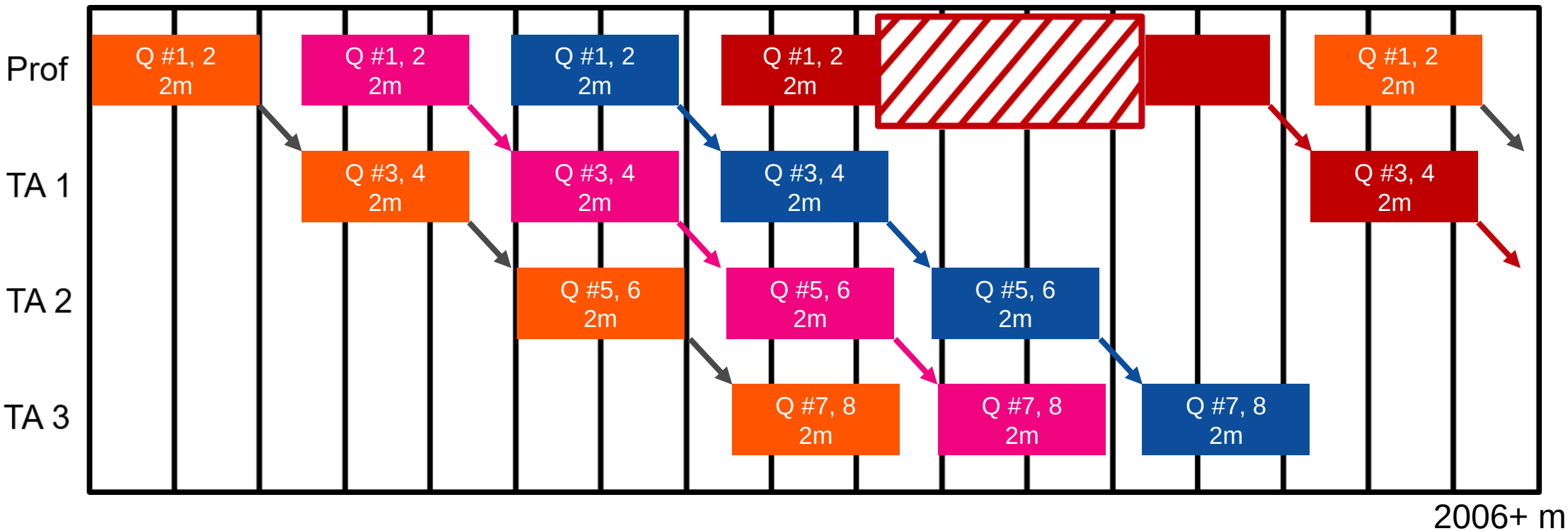
PARALLEL SOLUTION



PIPELINE



PIPELINE STALL

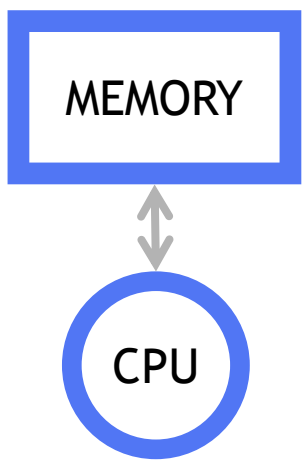


STATE OF THE ART 2019

CORAL Summit System
5-10x Faster
1/5th the Nodes,
Same Energy Use as Titan

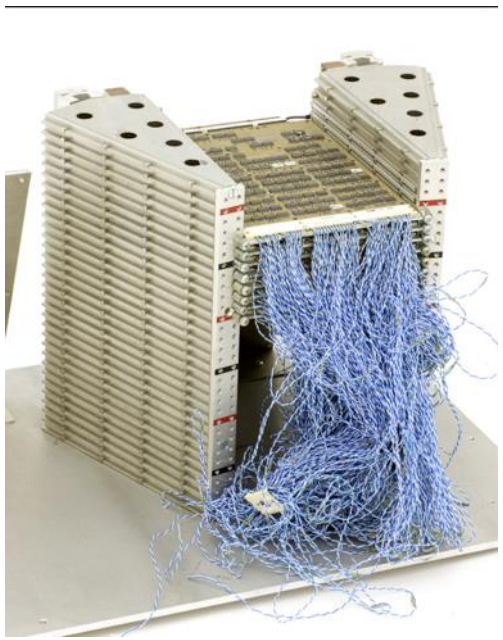
Earth Simulator
40.96 TF
Top500 #1 for 3 years

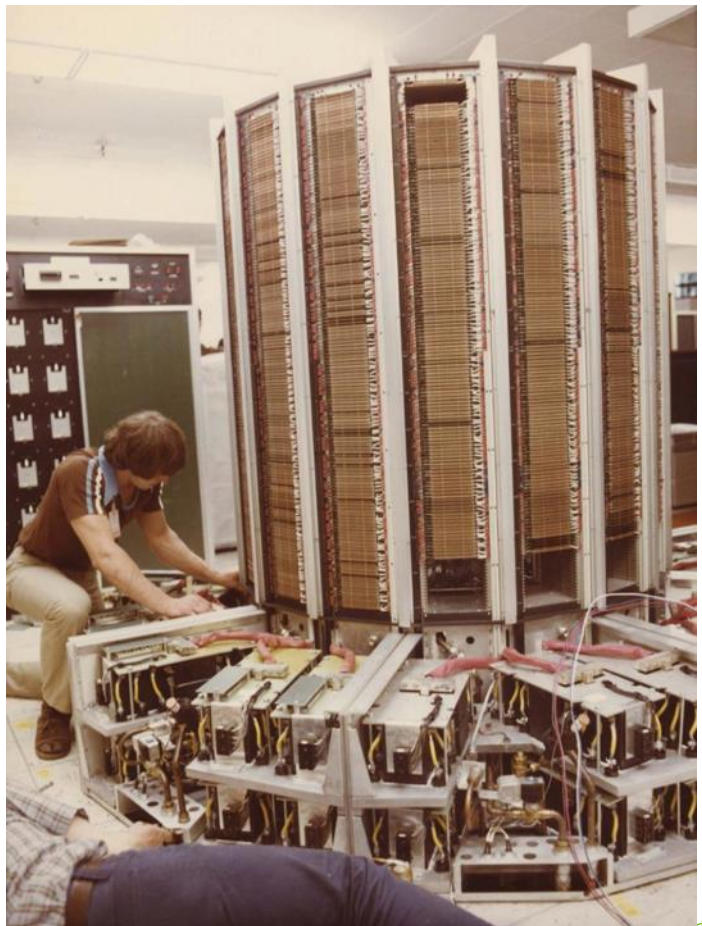
Just 1 Node in Summit
Is the same performance as the Earth Simulator in 2002

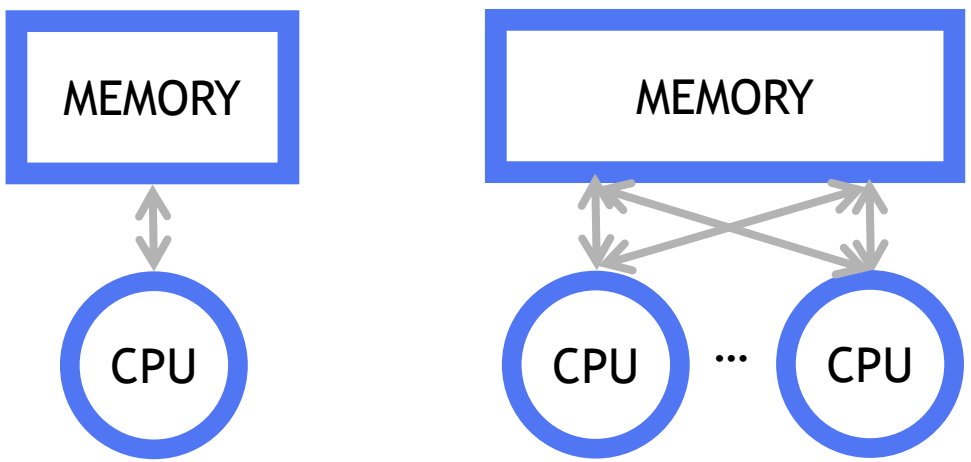


CRAY-1

LATENCY-HIDING SINGLE VECTOR CPU, 160 MFLOPS PEAK

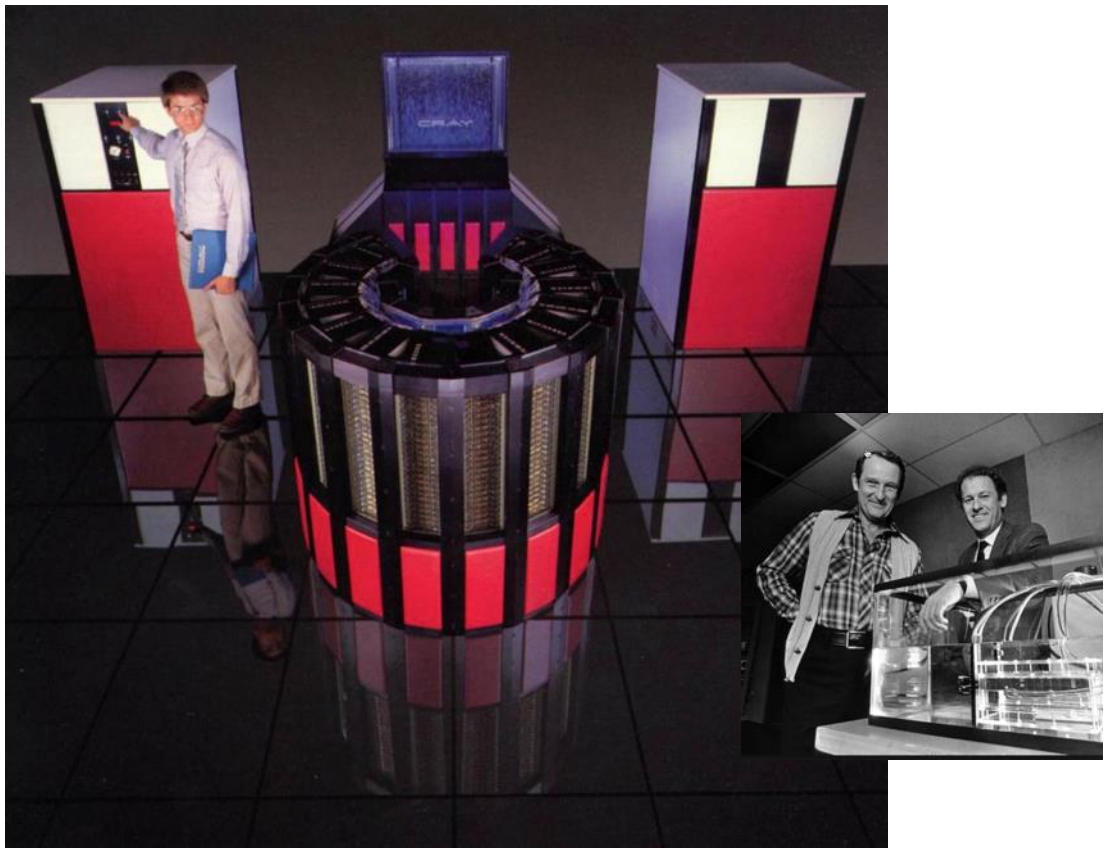


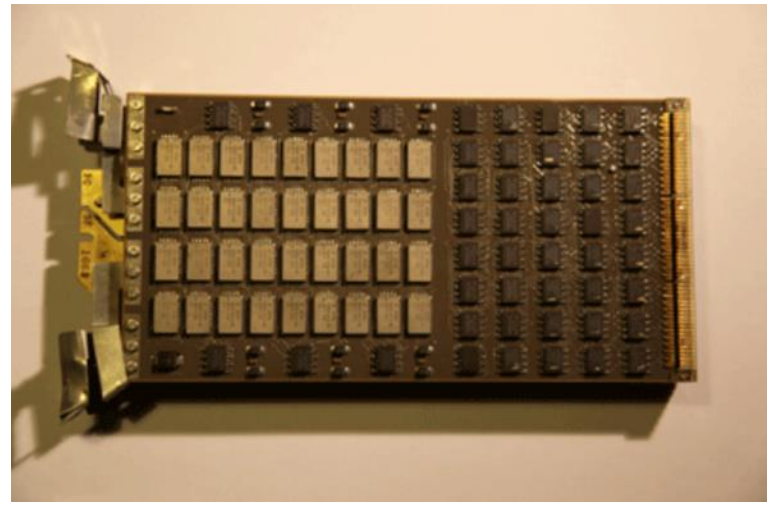


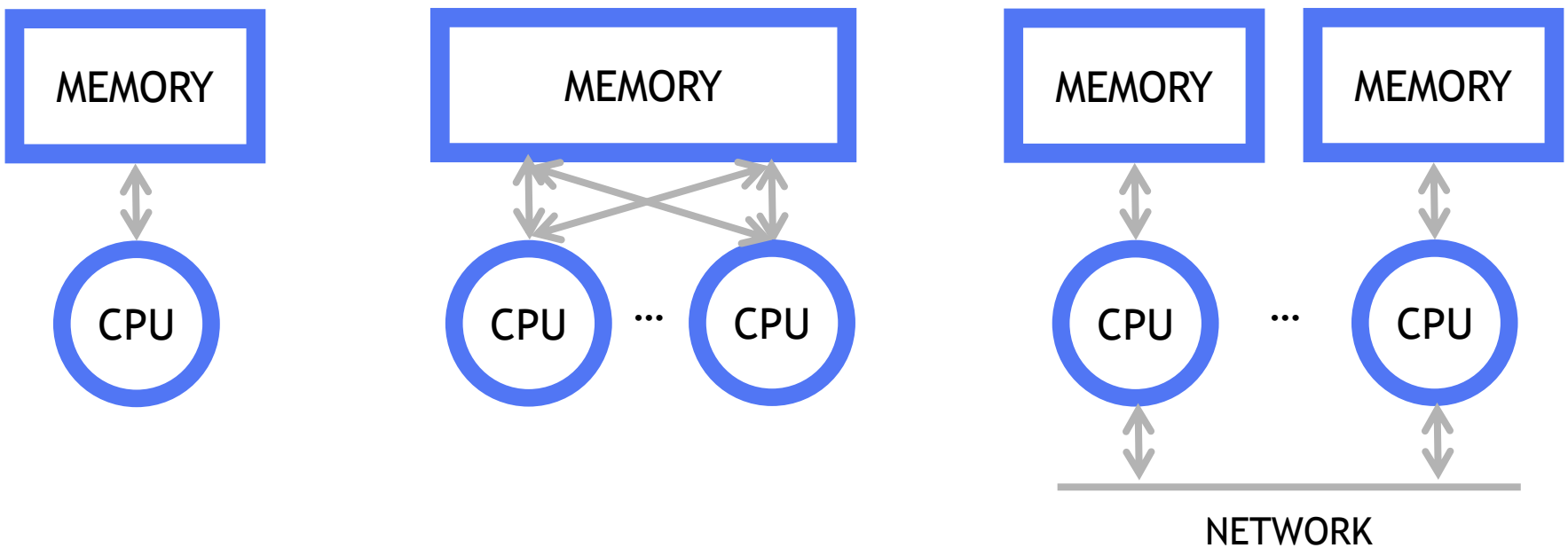

MEMORY
CPU
MEMORY
CPU
CPU

CRAY-2

4 LATENCY-HIDING VECTOR CPUS, 2 GFLOPS PEAK, 1985



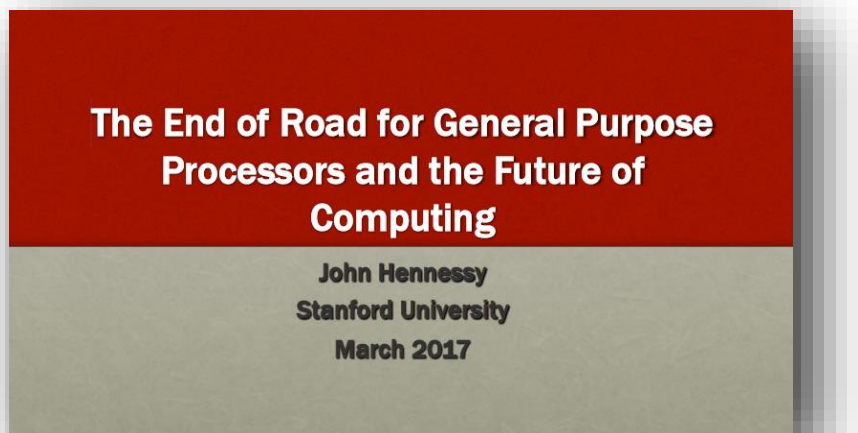


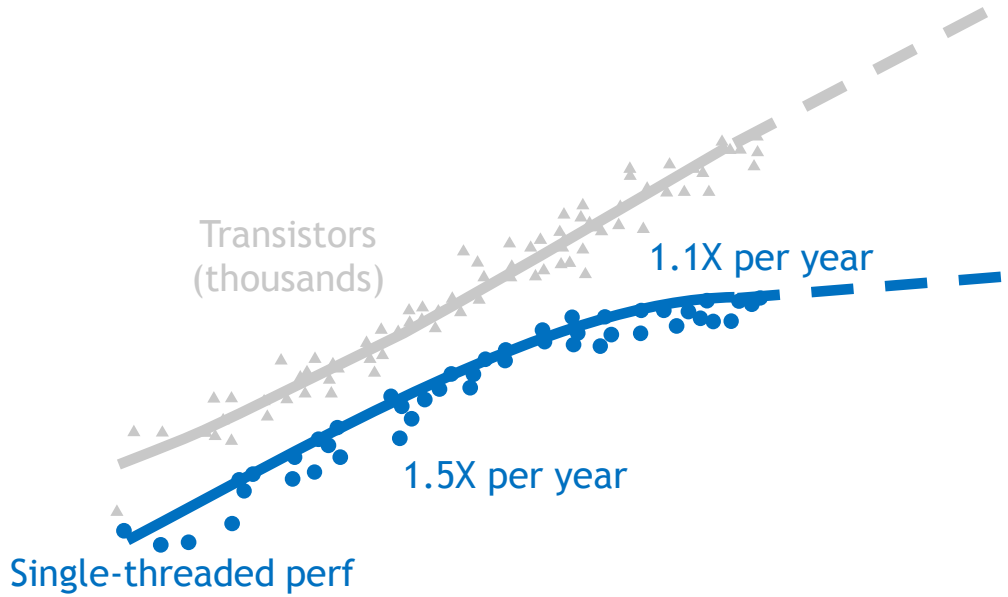

MEMORY
CPU
MEMORY
CPU
CPU
MEMORY
MEMORY
CPU
CPU
NETWORK

CRAY T3D

DEC ALPHA EV4 MICROPROCESSORS, 1 TFLOPS PEAK, 1993

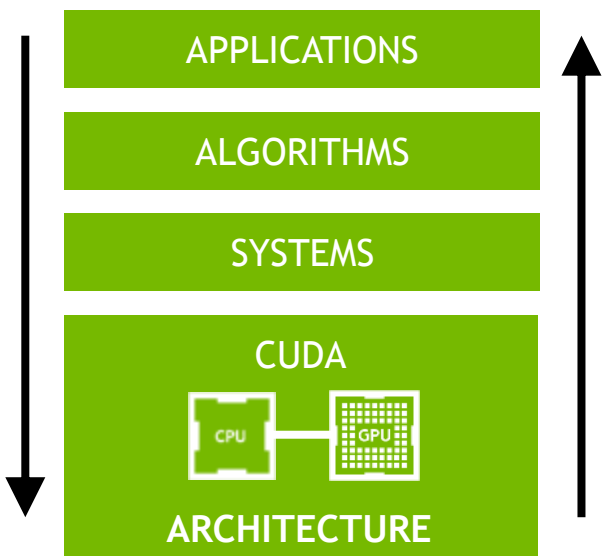
LIFE AFTER MOORE'S LAW

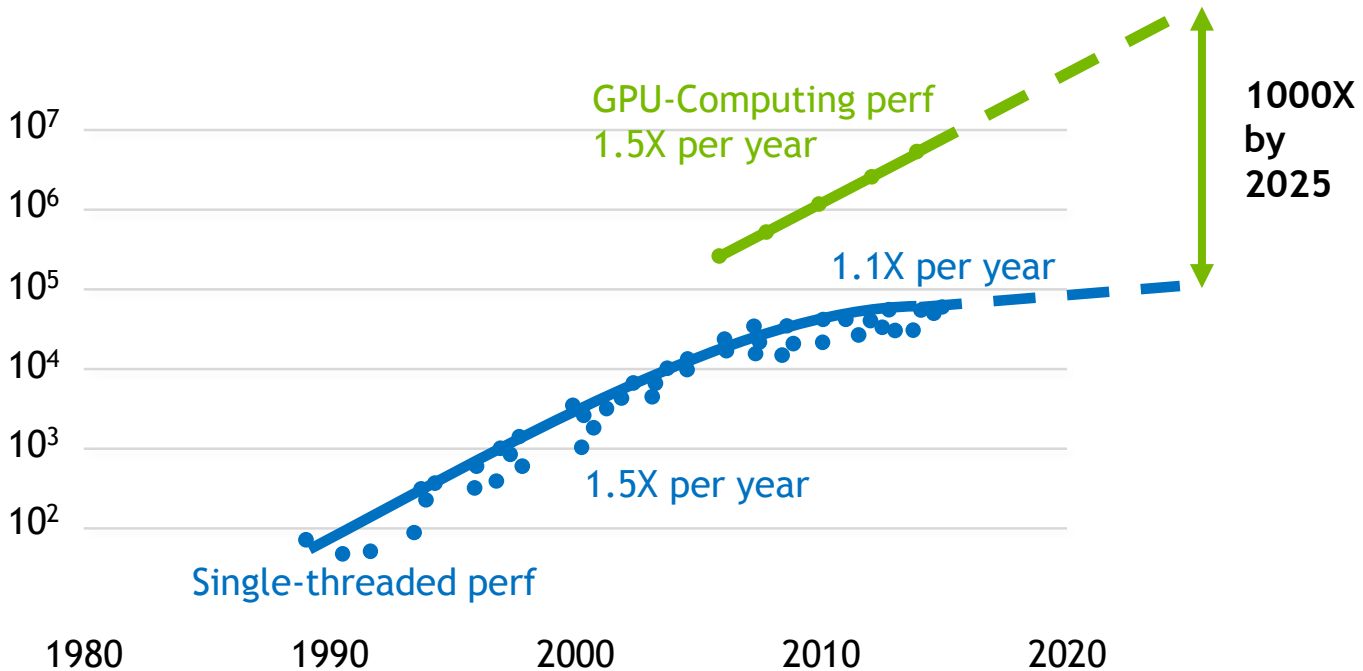




Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

RISE OF GPU COMPUTING





Original data up to the year 2010 collected and plotted by M. Horowitz, F. Labonte, O. Shacham, K. Olukotun, L. Hammond, and C. Batten New plot and data collected for 2010-2015 by K. Rupp

CUDA PLATFORM

People

GPU Users | *Domain Specialists* | *Problem Specialists* | *New Developers and Optimization Experts*

Domain Descriptors

Applications

Frameworks

Libraries

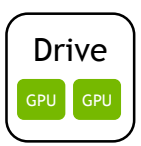
Directives and Standard Languages

CUDA-C++
CUDA Fortran

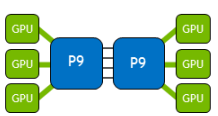
Extended Standard Languages

Ease of use ——— *Specialized Performance* →

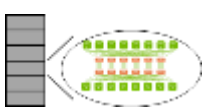
Systems



GeForce
Quadro
Tesla PCI



Supercomputing

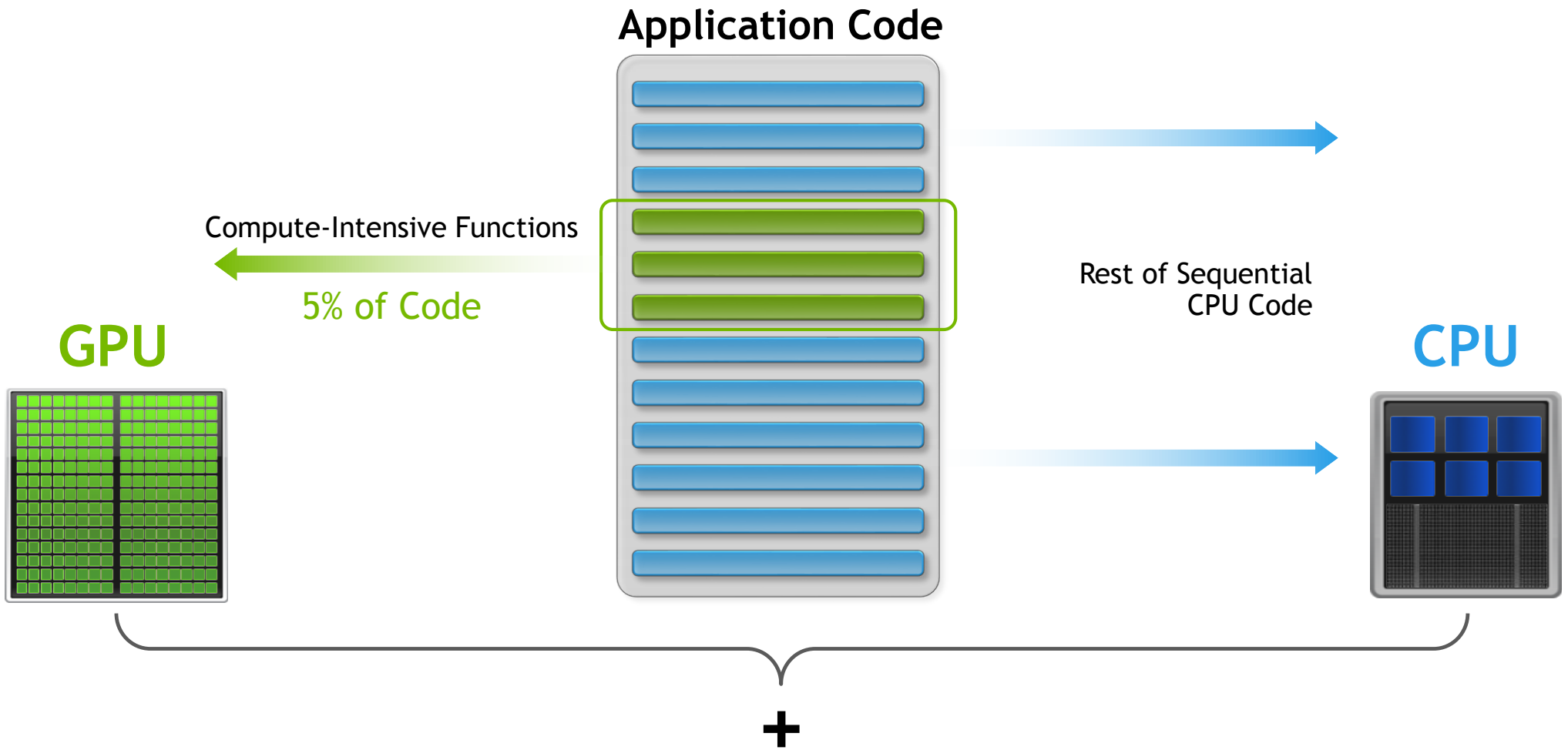


Cloud + HGX/DGX

Foundation

Programming Model, GPU Architecture, System Architecture, Tools

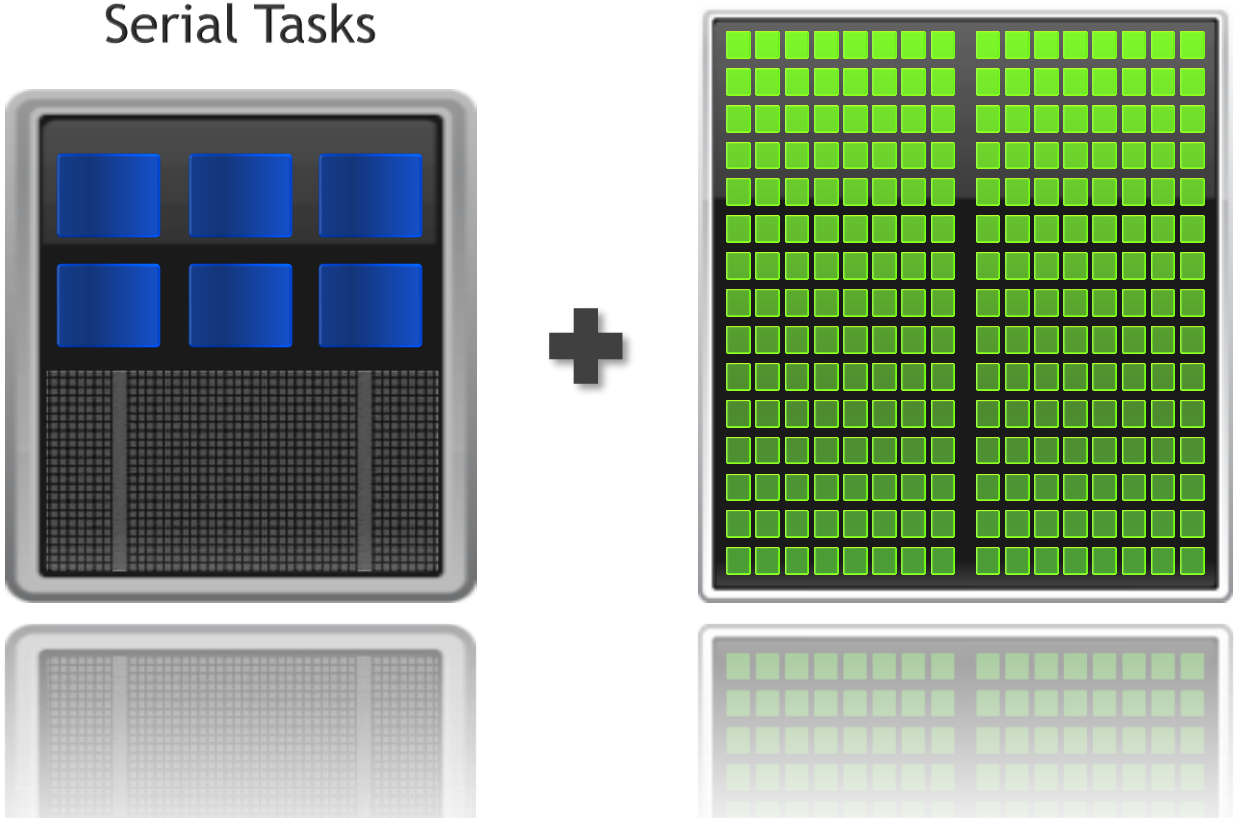
HOW GPU ACCELERATION WORKS



ACCELERATED COMPUTING

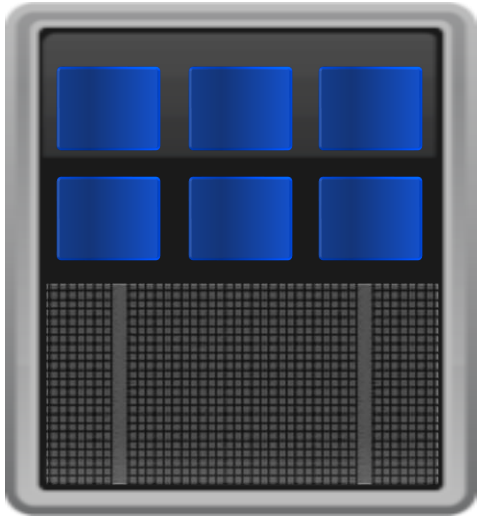
CPU
Optimized for
Serial Tasks

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

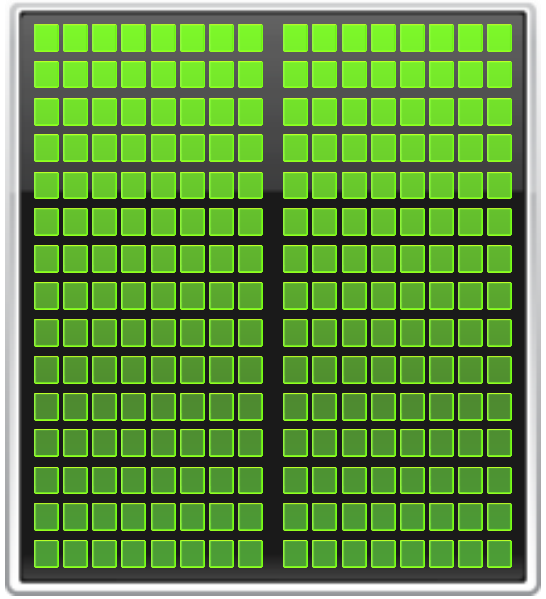
- High bandwidth main memory
- Significantly more compute resources
- Latency tolerant via parallelism
- High throughput
- High performance/watt

GPU Weaknesses

- Relatively low memory capacity
- Low per-thread performance

GPU Accelerator

Optimized for Parallel Tasks



GIANT LEAP FOR AI & HPC
VOLTA WITH NEW TENSOR CORE

21B xtors | TSMC 12nm FFN | 815mm^2

5,120 CUDA cores

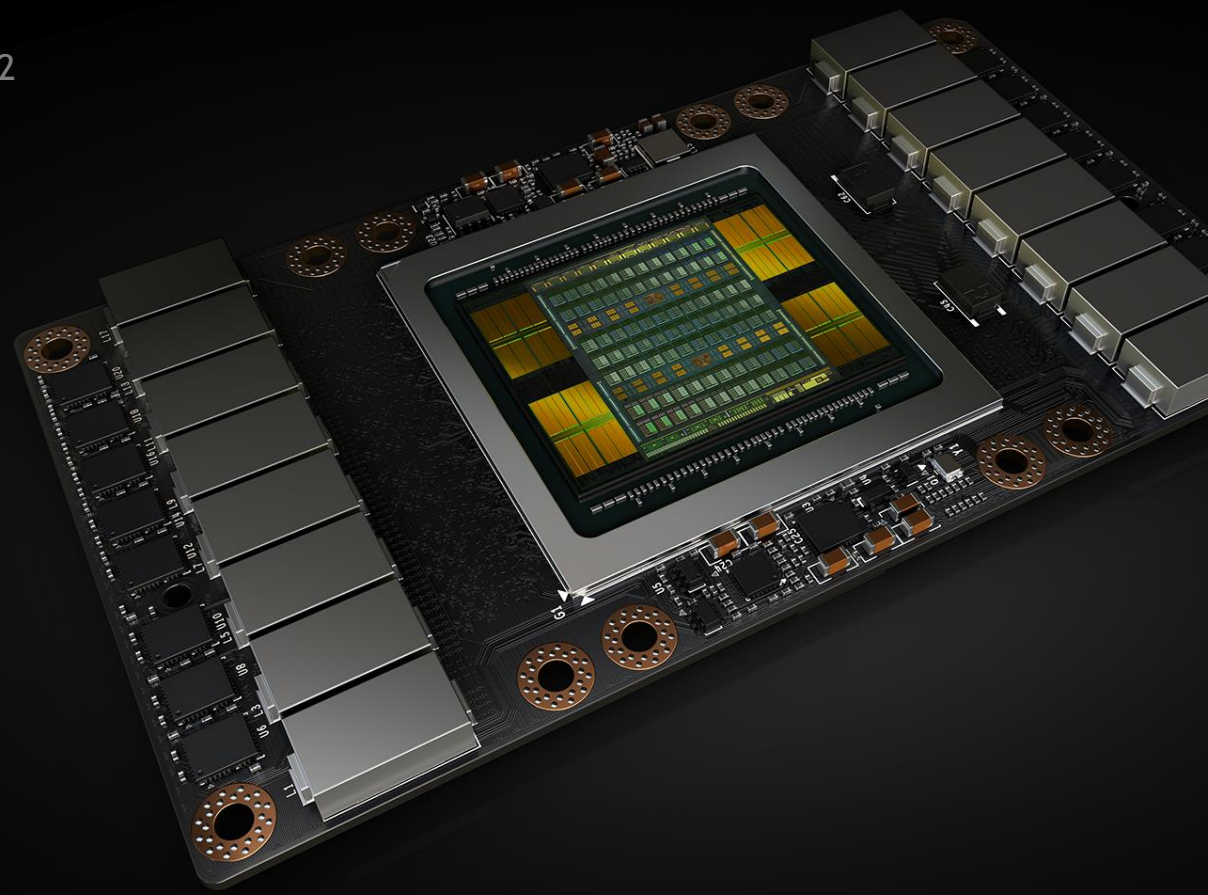
7.8 FP64 TFLOPS | 15 FP32 TFLOPS

NEW 120 Tensor TFLOPS

20MB SM RF | 16MB Cache

32GB HBM2 @ 900 GB/s

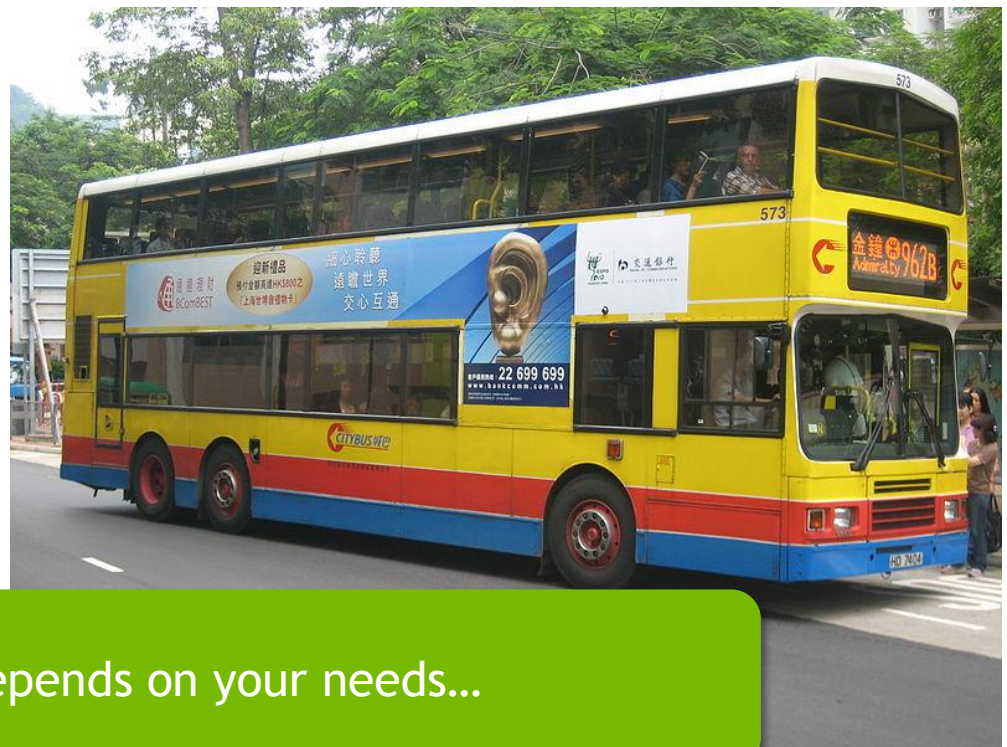
300 GB/s NVLink



SPEED V. THROUGHPUT

Speed

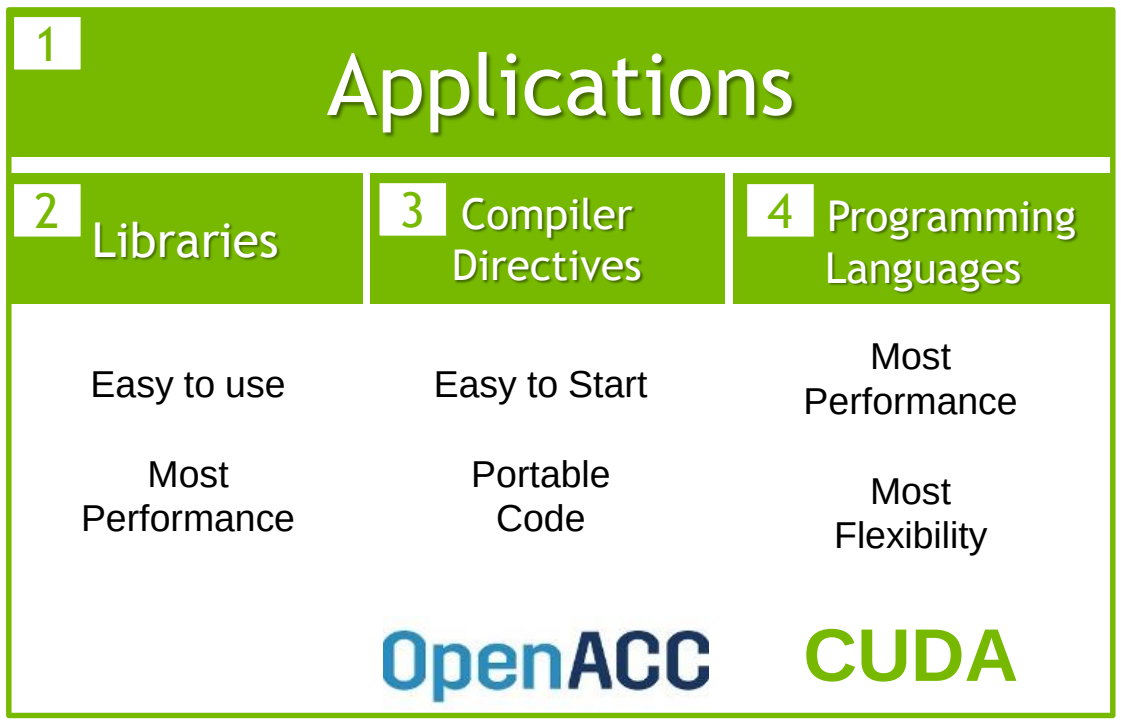
Throughput



Which is better depends on your needs…

*Images from Wikimedia Commons via Creative Commons

HOW TO START WITH GPUS



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

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

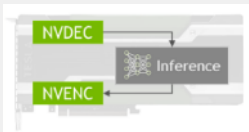
GPU ACCELERATED LIBRARIES

“Drop-in” Acceleration for Your Applications

DEEP LEARNING

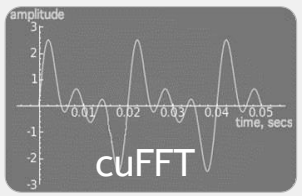
cuDNN


TensorRT



DeepStream SDK

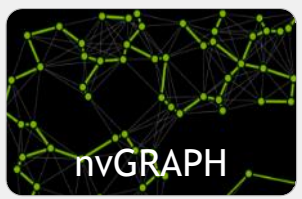
SIGNAL, IMAGE & VIDEO

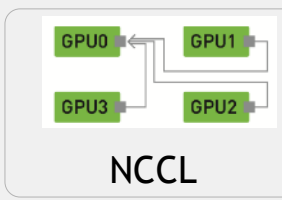


LINEAR ALGEBRA


cuSPARSE

PARALLEL ALGORITHMS





NCCL

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>
  }
 }
```

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

Simple

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

$$z = \alpha x + y$$

x, y, z : vector

α : scalar

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

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

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

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

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

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

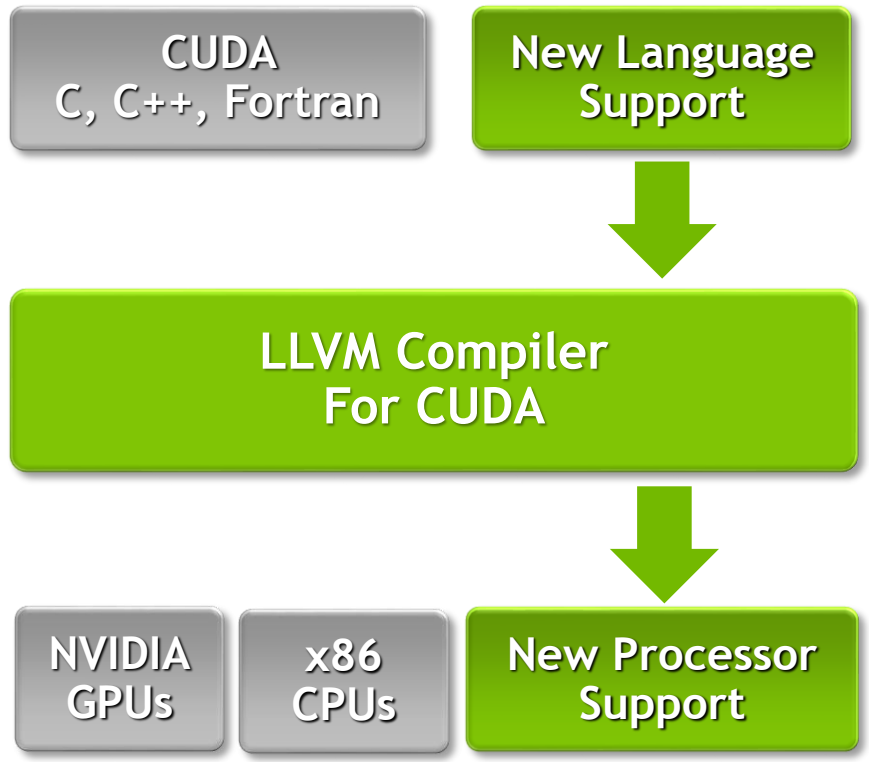
ENABLING ENDLESS WAYS TO SAXPY

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

LLVM COMPILER INFRASTRUCTURE


GPU COMPUTING PLATFORM
FOR DL/ML

NVIDIA DEEP LEARNING SOFTWARE STACK

TRAINING

Training Data

Data Management

Training

Model Assessment

Trained Neural Network

Caffe2 | Chainer | Microsoft Cognitive Toolkit | mxnet | TensorFlow | PYTORCH | theano

INFERENCE

Data center

TensorRT

Embedded

JETPACK SDK

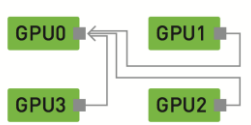
Automotive

DriveWorks SDK

NVIDIA DEEP LEARNING SDK and CUDA

cuDNN

NCCL

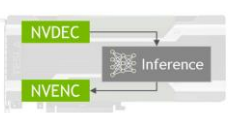


cuBLAS

TensorRT

cuSPARSE

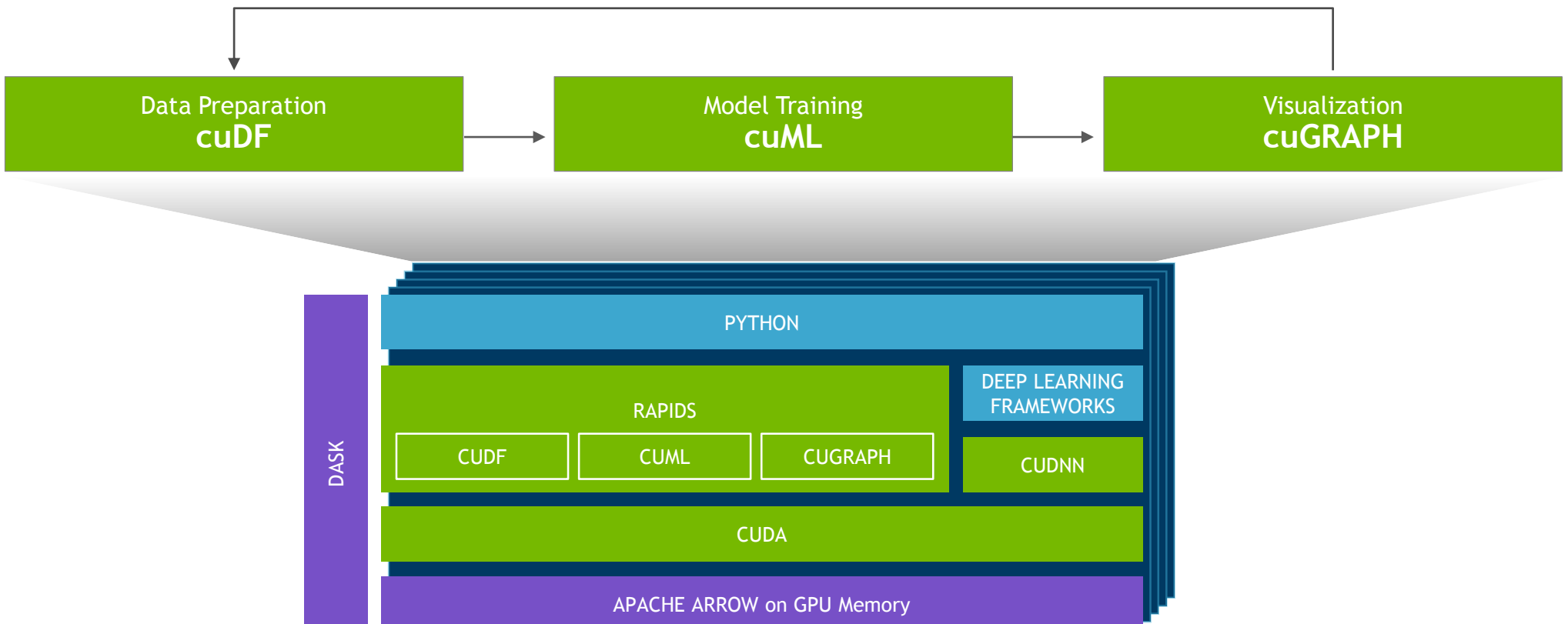
DeepStream SDK



developer.nvidia.com/deep-learning-software

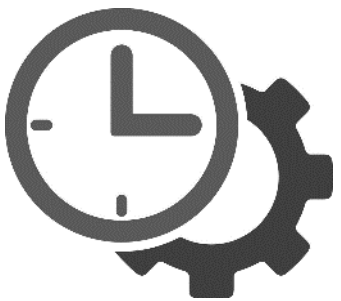
RAPIDS — OPEN GPU DATA SCIENCE

Software Stack Python




NGC FOR EFFICIENCY

CHALLENGES UTILIZING AI & HPC SOFTWARE

Installation	Optimization	Maintenance	Productivity
			
Complex, time consuming, and error-prone	Requires expertise to optimize framework performance	IT can’t keep up with frequent software upgrades	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


NVIDIA
Thank You