

GPU Intro and Python AI + GPU

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn

Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University



SAINT LOUIS
UNIVERSITY™

— EST. 1818 —

Jupyter Notebook

- <https://jupyter.org/>

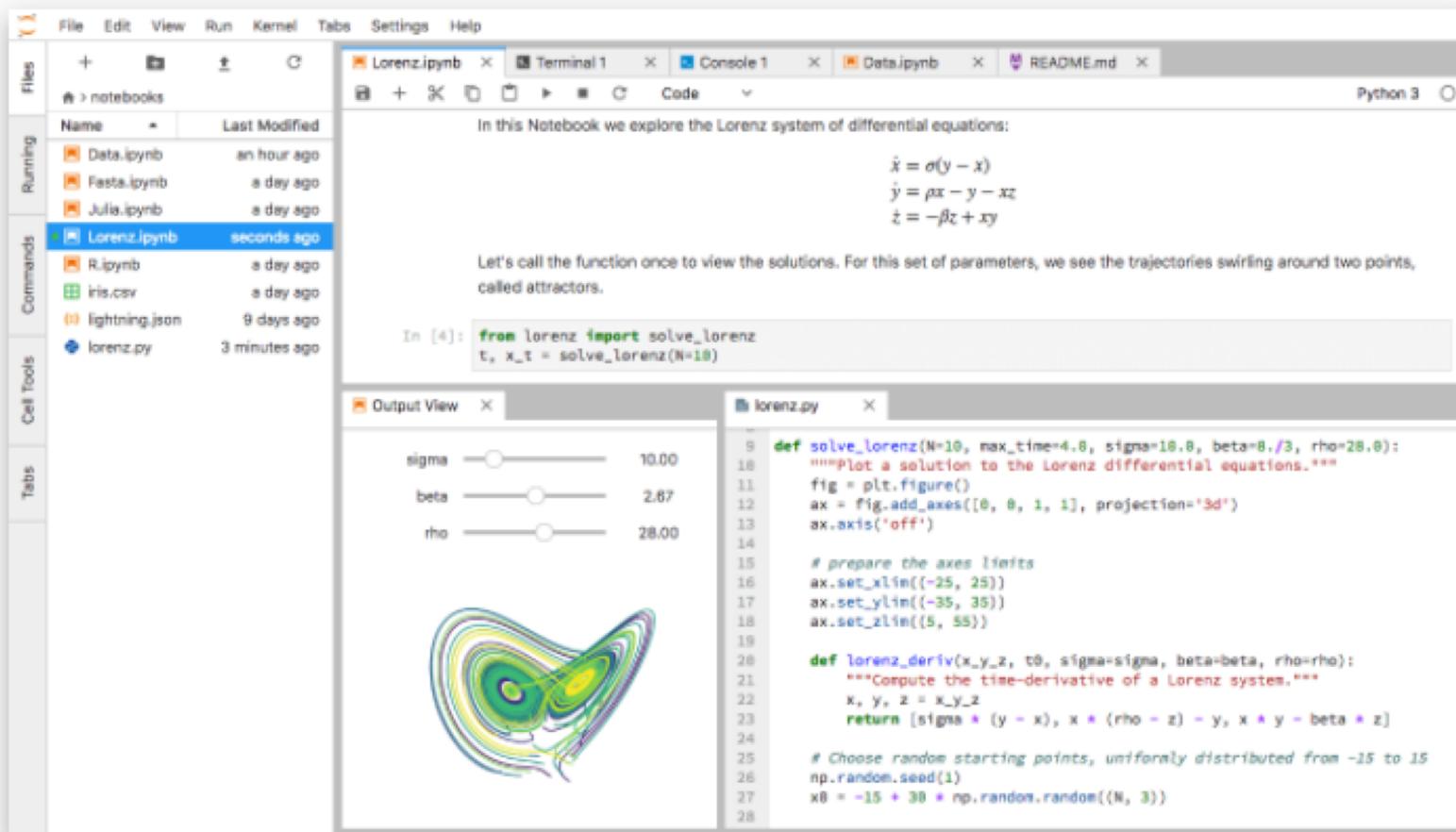


Project Jupyter exists to develop open-source software, open-standards, and services for interactive computing across dozens of programming languages.

Jupyter Lab

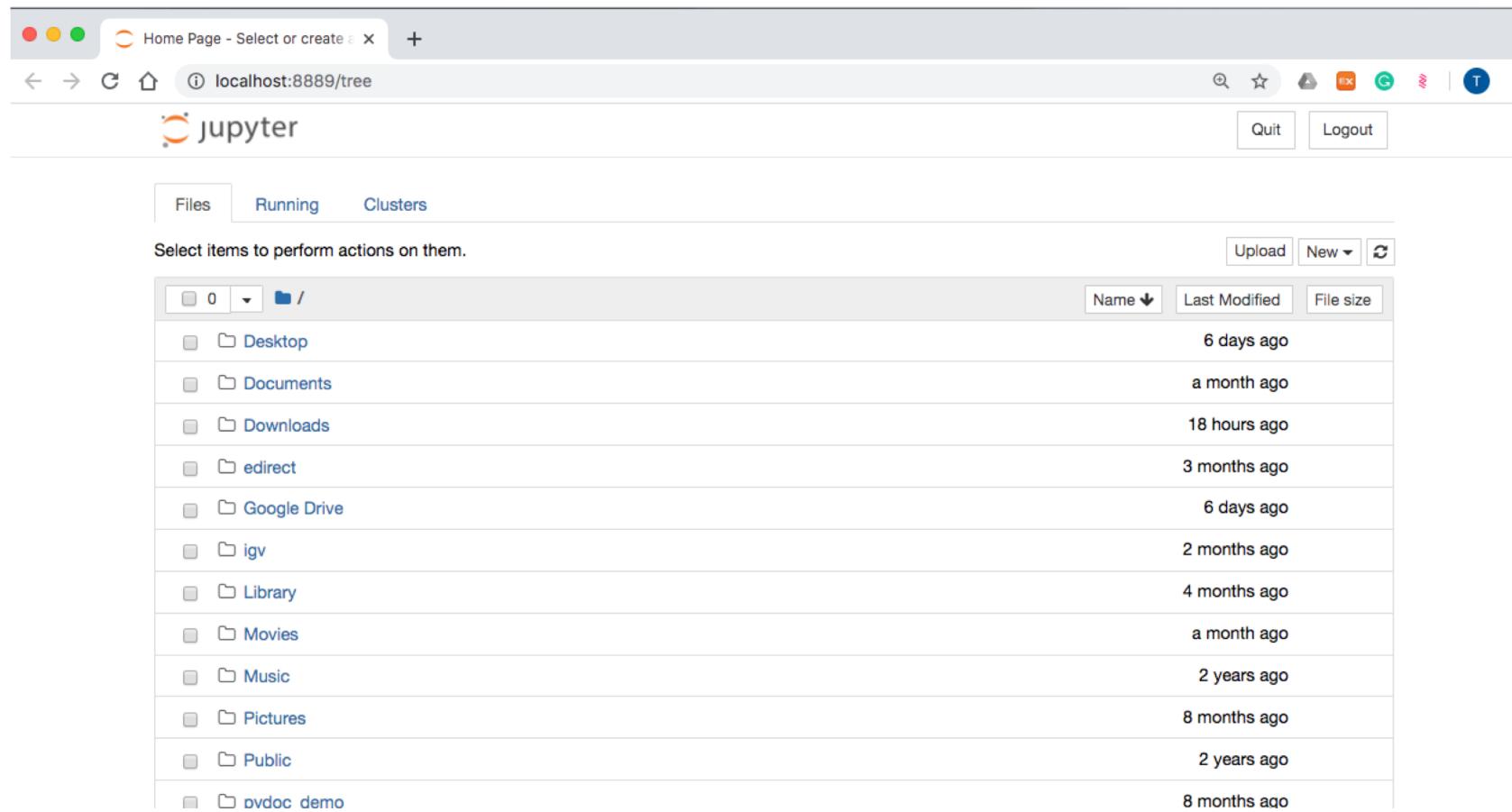
- <https://jupyterlab.readthedocs.io/en/stable/>

JupyterLab is the next-generation web-based user interface for Project Jupyter. Try it on Binder.
JupyterLab follows the Jupyter Community Guides.



Run Jupyter Notebook (Jupyter Lab)

- Install and run
 - Jupyter notebook: just type “jupyter notebook” and it will open a default web browser



How to run the program on GIST VM via Jupyter Notebook?

- <https://techtalktome.wordpress.com/2017/03/28/running-jupyter-notebooks-on-a-remote-server-via-ssh/>
- Remote (Server: our case is GIST VM):
 - \$ jupyter notebook password
 - \$ jupyter notebook --no-browser --port=8887
 - Be careful: Each user will use a unique port. If someone is using the port, change the 8887 to something else.
- Local (Mac or Windows):
 - \$ ssh -N -L localhost:8888:localhost:8887 YourAccount@210.125.85.148
 - Open a browser and type localhost:8888

How to run a program on GIST VM via Jupyter Notebook?

The screenshot shows a Jupyter Notebook interface. At the top, there is a navigation bar with the Jupyter logo, 'jupyter' text, and 'Quit' and 'Logout' buttons. Below the navigation bar, there are three tabs: 'Files' (selected), 'Running', and 'Clusters'. A message 'Select items to perform actions on them.' is displayed above a file list. On the right side of the file list, there are buttons for 'Upload', 'New ▾', and a refresh icon. The file list itself has columns for selection, count ('0'), a dropdown menu, a folder icon, the file name ('/'), and sorting options ('Name', 'Last Modified', 'File size'). Three files are listed: 'Assignments' (40 minutes ago), 'Lab' (2 hours ago), and '!' (7 days ago, 843 B).

	0		/	Name ↓	Last Modified	File size
<input type="checkbox"/>	40 minutes ago		Assignments			
<input type="checkbox"/>	2 hours ago		Lab			
<input type="checkbox"/>	7 days ago	843 B	!			

GPU

- GPU: Graphics Processing Unit
 - Designed for real-time graphics
 - Present in almost every PC
 - Increasing realism and complexity



What is GPU?

- It is a processor optimized for 2D/3D graphics, video, visual computing, and display.
- It is highly parallel, highly multithreaded multiprocessor optimized for visual computing.
- It provides real-time visual interaction with computed objects via graphics images, and video.
- It serves as both a programmable graphics processor and a scalable parallel computing platform.
- Heterogeneous Systems: combine a GPU with a CPU

Nvidia GPU

The Best Budget Graphics Cards in 2021

Featured



NVIDIA GEFORCE RTX 3070

› Cooling System: Fan
› Boost Clock Speed: 1.73 GHz
› GPU Memory Size: 8 GB

\$499.99

[CHECK AVAILABILITY](#)

[+ Compare](#)

Featured



NVIDIA GEFORCE RTX 3060 Ti

› Cooling System: Fan
› Boost Clock Speed: 1.67 GHz
› GPU Memory Size: 8 GB

\$399.99

[CHECK AVAILABILITY](#)

[+ Compare](#)



NVIDIA GEFORCE RTX 3080

› Cooling System: Fan
› Boost Clock Speed: 1.71 GHz
› GPU Memory Size: 10 GB

\$699.99

[CHECK AVAILABILITY](#)

[+ Compare](#)

RTX 3080

GEFORCE RTX 3080

THE ULTIMATE PLAY

The GeForce RTX™ 3080 delivers the ultra performance that gamers crave, powered by Ampere—NVIDIA's 2nd gen RTX architecture. It's built with enhanced RT Cores and Tensor Cores, new streaming multiprocessors, and superfast G6X memory for an amazing gaming experience.

STARTING AT \$699.00

2ND GENERATION
RT CORES

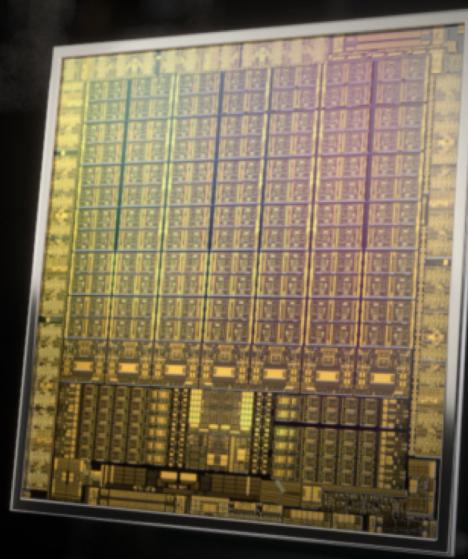
2X THROUGHPUT

3RD GENERATION
TENSOR CORES

UP TO 2X THROUGHPUT

NEW
SM

2X FP32 THROUGHPUT



RTX 3080

- Specs (<https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080/>)

GEFORCE RTX 3080		
GPU Engine Specs:	NVIDIA CUDA® Cores	8704
	Boost Clock (GHz)	1.71
	Base Clock (GHz)	1.44
Memory Specs:	Standard Memory Config	10 GB GDDR6X
	Memory Interface Width	320-bit
Technology Support:	Ray Tracing Cores	2nd Generation
	Tensor Cores	3rd Generation
	NVIDIA Architecture	Ampere
	Microsoft DirectX® 12 Ultimate	Yes

What is RTX?

- RTX stands for Ray Tracing Texel eXtreme and is also a variant under GeForce.
- The RTX cards were specially designed to support real-time ray tracing which made the video looked more beautiful.
- They were first announced in 2018 and uses Turing architecture.
- These cards support the DXR extension in Microsoft's DirectX12 and also DLSS (Deep Learning Super Sampling).

NVIDIA's Ampere architecture is an evolution of Turing, which powered the RTX 20 series. The new architecture includes huge gains in three key areas: streaming multiprocessors (SM) with double the throughput of their predecessors, second-gen RT cores that deliver significantly better performance, and third-gen Tensor cores that offer sizable gains in AI-based tasks.

<https://www.windowscentral.com/nvidia-geforce-rtx-3080-review>

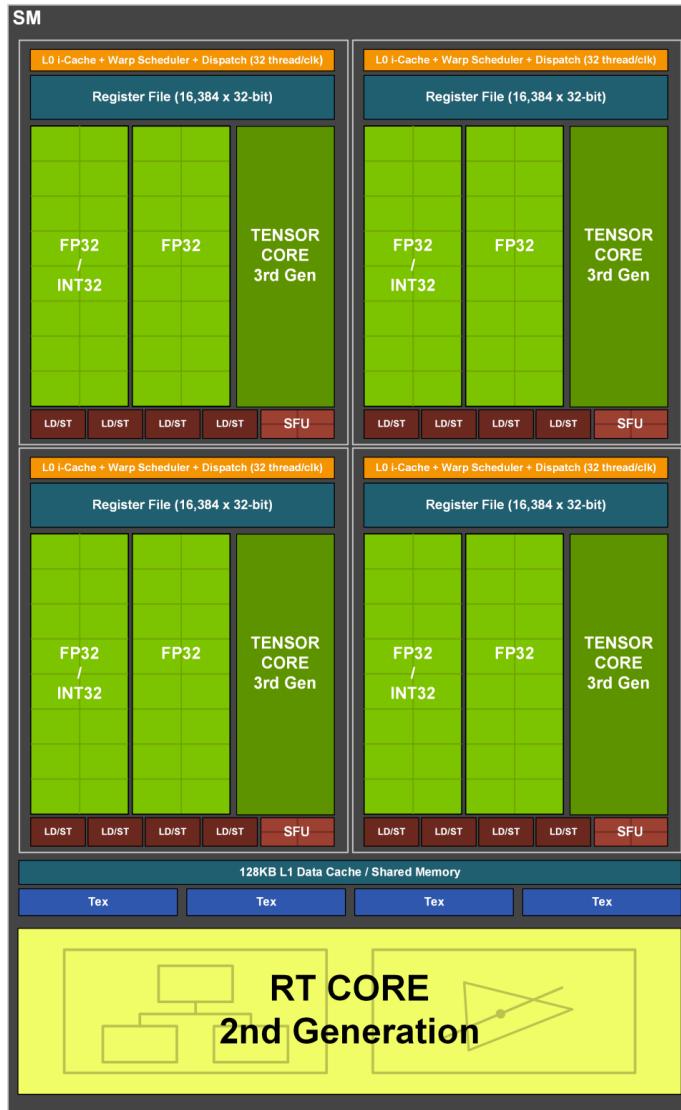
RTX 3080

Graphics Card	RTX 3080 FE
Architecture	GA102
Process (nm)	Samsung 8N
Transistors (Billion)	28.3
Die size (mm ²)	628.4
GPCs	6
SMs	68
FP32 CUDA Cores	8704
Tensor Cores	272
RT Cores	68
Boost Clock (MHz)	1710
VRAM Speed (Gbps)	19
VRAM (GB)	10
VRAM Bus Width	320
ROPs	96
TPCs	34



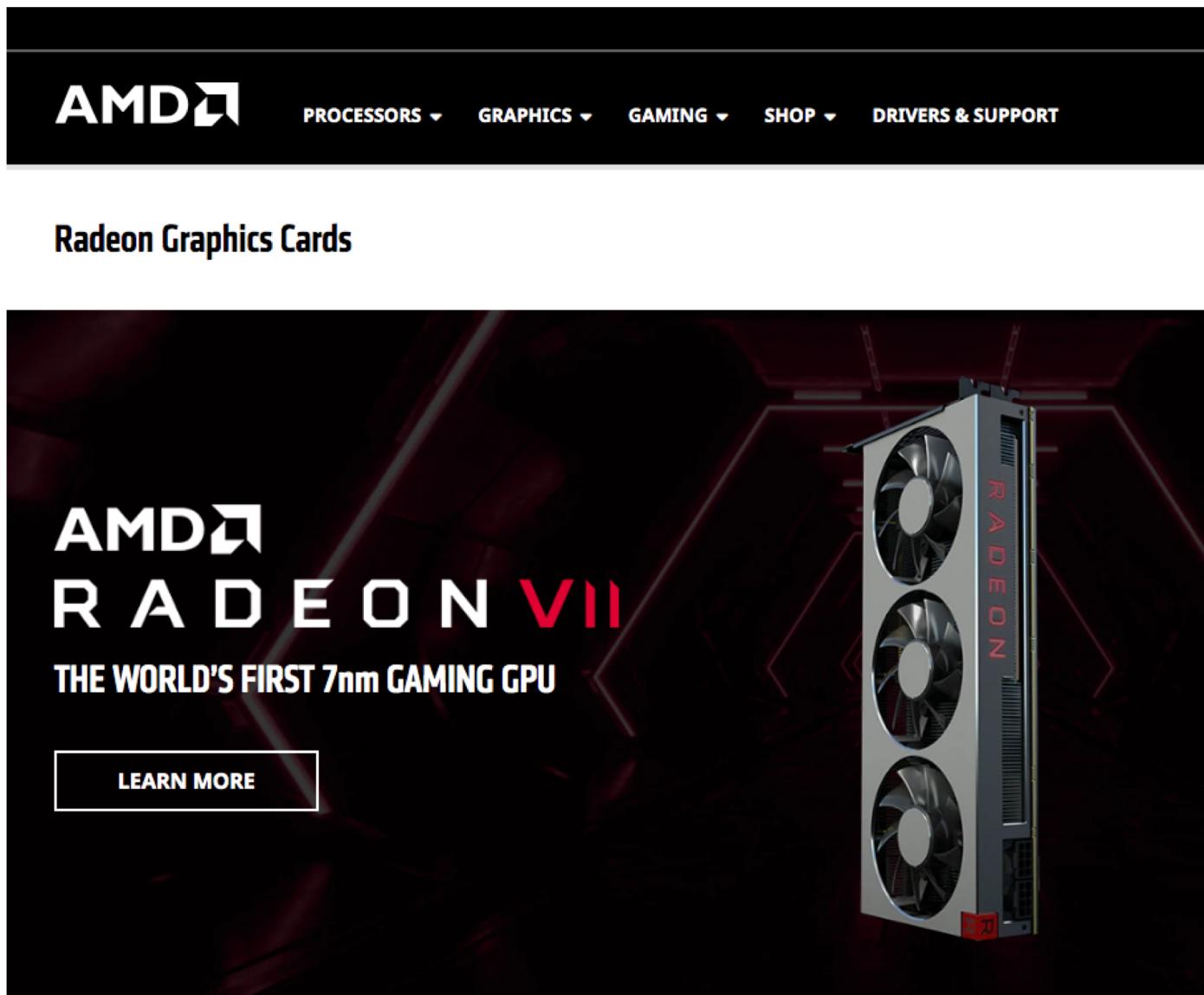
RTX 3080

Graphics Card	RTX 3080 FE
Architecture	GA102
Process (nm)	Samsung 8N
Transistors (Billion)	28.3
Die size (mm ²)	628.4
GPCs	6
SMs	68
FP32 CUDA Cores	8704
Tensor Cores	272
RT Cores	68
Boost Clock (MHz)	1710
VRAM Speed (Gbps)	19
VRAM (GB)	10
VRAM Bus Width	320
ROPs	96
TPCs	34



- Nvidia GPUs consist of several GPCs (Graphics Processing Clusters), each of which has some number of SMs (Streaming Multiprocessors). Nvidia splits each SM into four partitions that can operate on separate sets of data. With Ampere, each SM partition now has 16 FP32 CUDA cores, 16 FP32/INT CUDA cores, a third-gen Tensor core, load/store units, and a special function unit. The whole SM has access to shared L1 cache and memory, and there's a single second-gen RT core. In total, that means 64 FP32 cores and 64 FP32/INT cores, four Turing cores, and one RT core.
- <https://www.tomshardware.com/reviews/nvidia-geforce-rtx-3080-review>

AMD Radeon



The image shows a screenshot of the AMD Radeon website. At the top, there is a black navigation bar with the AMD logo and links for PROCESSORS, GRAPHICS, GAMING, SHOP, and DRIVERS & SUPPORT. Below the navigation bar, the text "Radeon Graphics Cards" is displayed. On the left side of the main content area, the text "AMD RADIATOR VII" is shown, followed by "THE WORLD'S FIRST 7nm GAMING GPU". A "LEARN MORE" button is located at the bottom left. On the right side, there is a large image of the AMD Radeon VII graphics card, which has three large fans and a red "RADEON" logo.

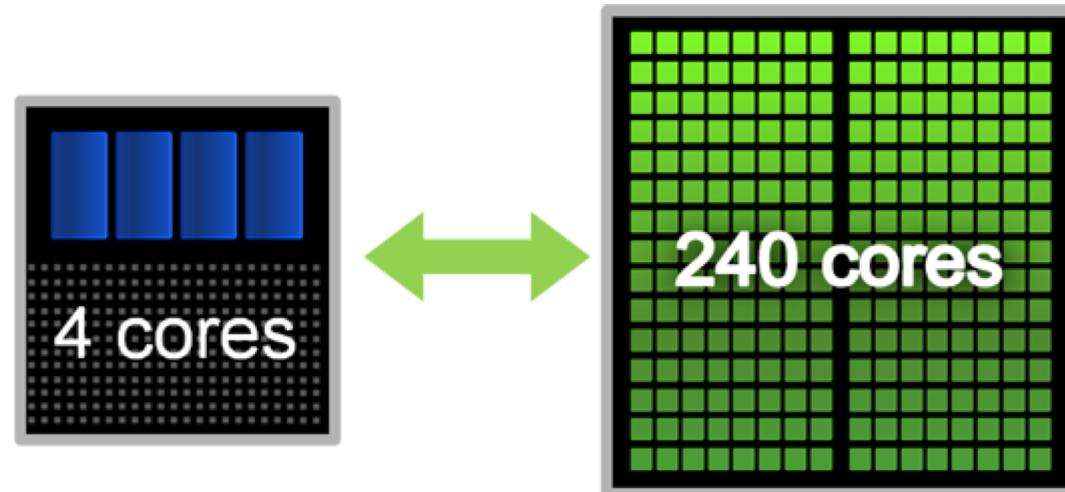
- My MacBook Pro has Radeon Pro 560



The image shows a screenshot of the macOS High Sierra System Report. At the top, it says "macOS High Sierra Version 10.13.6". Below that, it lists system specifications: "MacBook Pro (15-inch, 2017)", "Processor 2.9 GHz Intel Core i7", "Memory 16 GB 2133 MHz LPDDR3", "Graphics Radeon Pro 560 4096 MB Intel HD Graphics 630 1536 MB", and "Serial Number C02WH0NFHTDF". At the bottom, there are two buttons: "System Report..." and "Software Update...".

GPU and CPU

- Typically GPU and CPU coexist in a heterogeneous setting
- “Less” computationally intensive part runs on CPU (coarse-grained parallelism), and more intensive parts run on GPU (fine-grained parallelism)
- NVIDIA’s GPU architecture is called CUDA (Compute Unified Device Architecture) architecture, accompanied by CUDA programming model, and CUDA C/C++ language



Difference among CPU, GPU, APU, FPGA, DSP, and Intel MIC?

- <https://www.quora.com/What-is-the-difference-among-CPU-GPU-APU-FPGA-DSP-and-Intel-MIC>

- CPU
 - CPU is a general purpose processor. General Purpose in the sense that it is designed to perform a number of operations but the way these operations are performed may not be best for all applications. Graphics or Video Processing is one such example. Although a CPU can perform these tasks (which involve repeated additions/multiplications which may be performed in parallel), the performance achieved is not good enough for modern applications.
- GPU
 - Graphics processing Unit or GPU is designed to accelerate creation of images for a computer display. A CPU consists of a few cores optimized for sequential serial processing while a GPU consists of thousands of smaller, more efficient cores designed for handling multiple tasks simultaneously. They are designed to perform functions such as texture mapping, image rotation, translation, shading, etc. They may also support operations such as motion compensation, calculation of inverse DCT, etc. for accelerated video decoding.
- Intel MIC
 - Intel Many Integrated Core or Intel MIC Architecture combines many Intel cores onto a single chip. Unlike its competitor in HPC market, NVidia TESLA GPU, it provides up to 61 cores. One of the key advantages that Intel MIC offers over NVidia GPUs is that it runs same instruction set as 64-bit Pentium. Standard C, C++ can be used for writing source codes to program these cores.
- APU
 - APU stands for Accelerated Processing Unit which refers to the main processor of system which has additional functionality for accelerating execution of certain operations. This additional capability may be provided by an on chip graphics processing unit. In simple terms APU is CPU + GPU on a single chip.
- DSP
 - Digital Signal Processor or DSP is optimized for high speed processing of numeric data representing the analog signals in real time. They are designed for quickly performing large number of numeric operations repeatedly on a series of data samples and are ideal for processing streaming digital signals. They provide functionalities that are helpful for DSP applications such as bit reverse addressing which is helpful for FFT computation, architectural support for very tight extremely low overhead loops, saturation arithmetic, etc.
- FPGA
 - An FPGA (Field Programmable Gate Array) is entirely different from CPU, GPU, DSP, etc. in the sense that it is not a processor in itself i.e. it does not run a program stored in the program memory. In layman's terms, an FPGA is nothing but a bulk of reconfigurable digital logic suspended in a sea of programmable inter-connects. A typical FPGA may also have dedicated memory blocks, digital clock manager, IO banks and several other features which vary across different vendors and models. Since they can be configured after manufacturing at customer's end, they can be used to implement any logic function (including but not limited to a processor core). This makes them ideal for re-configurable computing and application specific processing. Intel has recently announced a new range of Xeon Processors with integrated FPGA so that each chip can be configured at run type depending upon application needs.

GIST AI-X Cluster System

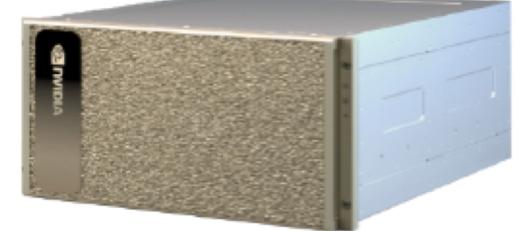


Computing



DGX A100

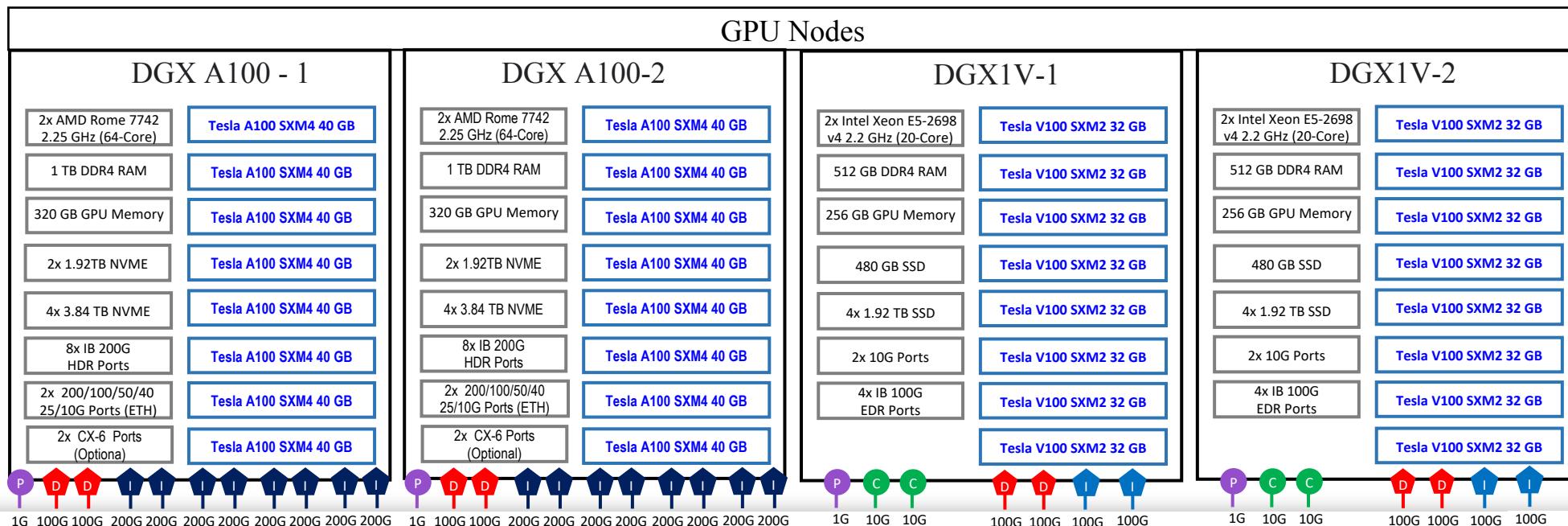
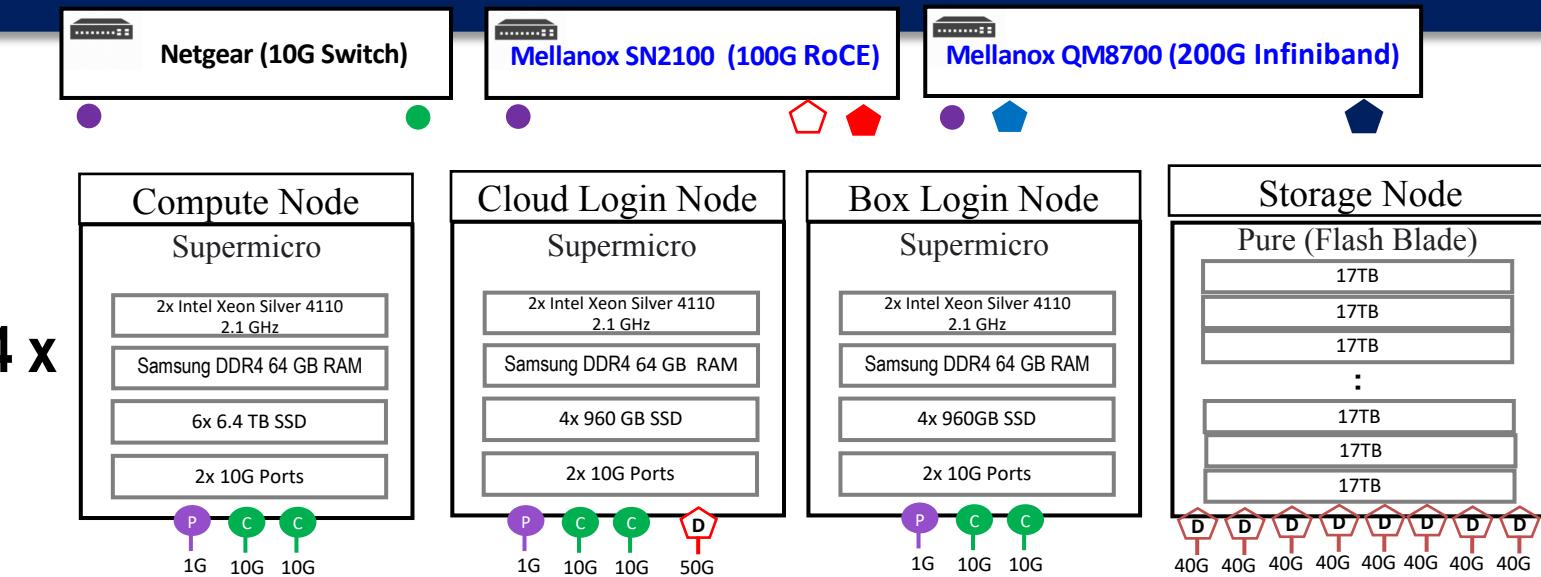
8x NVIDIA A100 GPUs
320 GB GPU Memory
1 TB System Memory
Dual AMD Rome
2.25 GHz (64-Core) CPU
9x 200Gb/s NIC



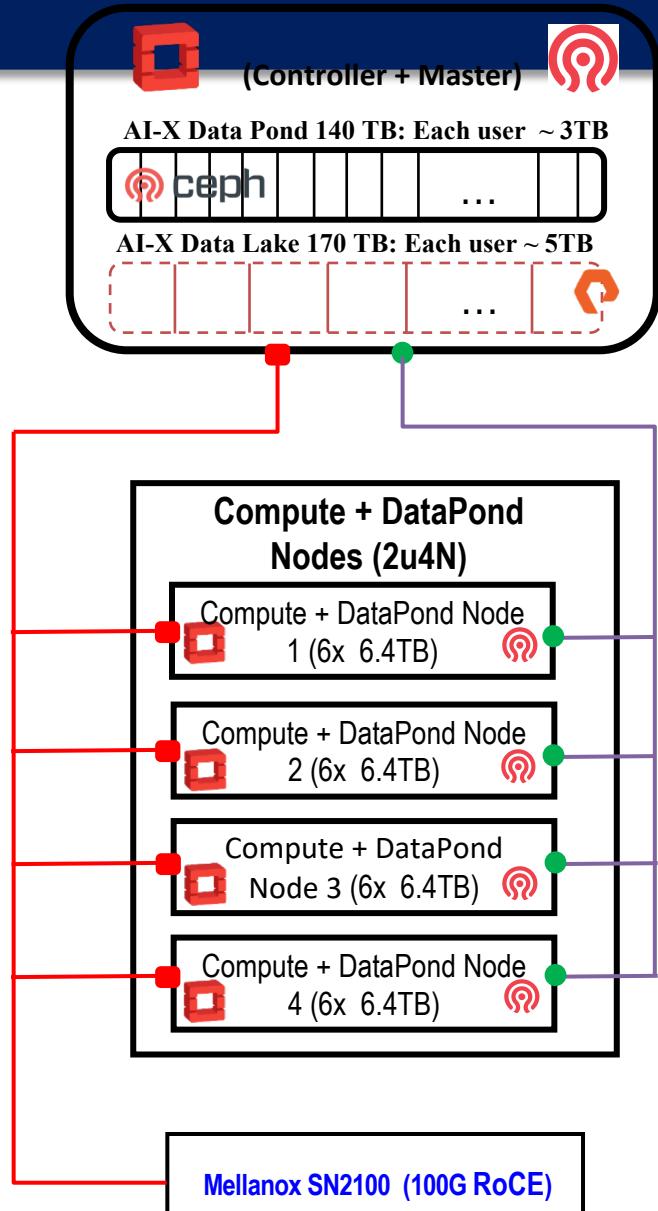
DGX-1V

8x NVIDIA V100 GPUs
256 GB GPU Memory
512 GB System Memory
Dual Intel Xeon E5-2698 v4
2.2 GHz (20-Core) CPU
4x 100Gb/s NIC

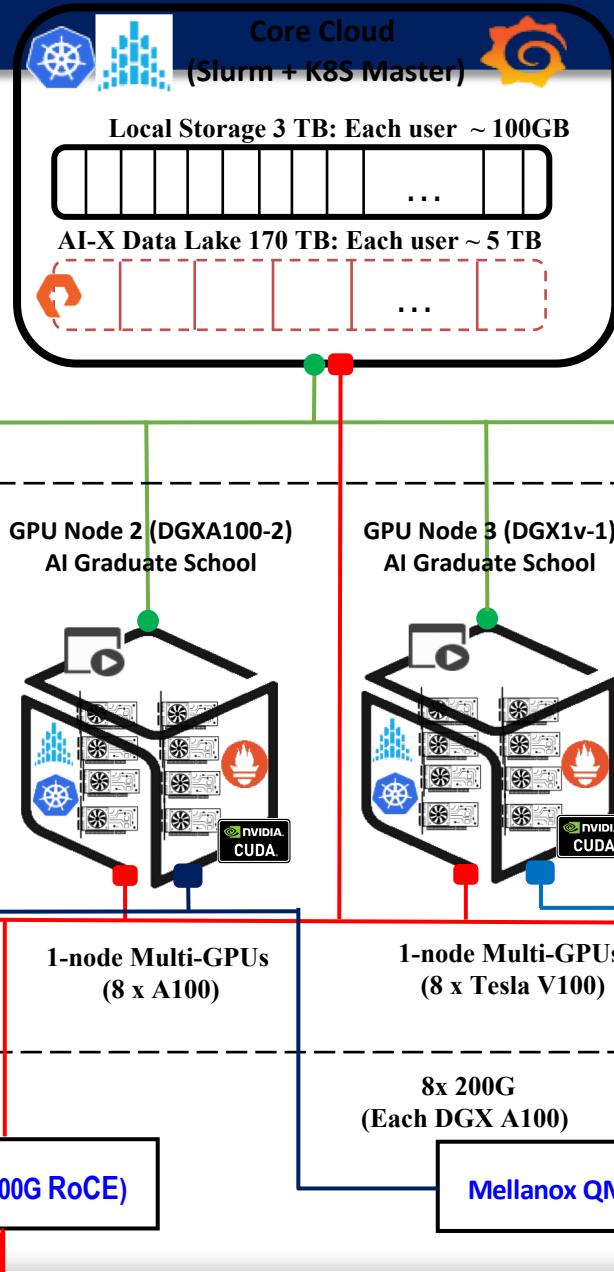




Front Cluster



Backend Cluster



- Management Network (10G)
- Data Network (100G RoCE)
- Internal Network (100G IB)
- Internal Network (200G IB)

GPU Programming Option

CUDA Versus OpenCL Versus OpenACC

 NVIDIA. ACCELERATED COMPUTING Downloads Training Ecosystem Forums [Join](#) [Login](#)

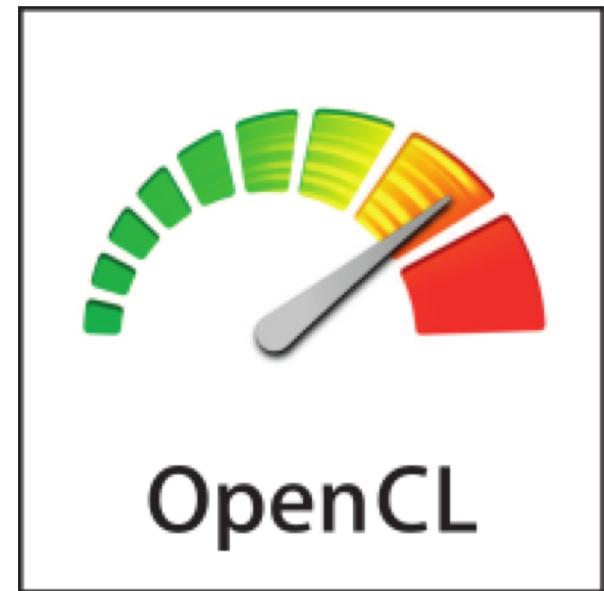
CUDA Zone

CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units [GPUs]. With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

The [CUDA Toolkit](#) from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.

[Download Now >](#)



[ABOUT](#) [BLOG](#) [TOOLS](#) [NEWS](#) [STORES](#) [EVENTS](#) [RESOURCES](#) [SPEC](#) [COMMUNITY](#)

What is OpenACC?

OpenACC is a user-driven directive-based performance-portable parallel programming model designed for scientists and engineers interested in porting their codes to a wide-variety of heterogeneous HPC hardware platforms and architectures with significantly less programming effort than required with a low-level model.

[Get Started](#) or [take the next steps](#)

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc kernels {
#pragma acc loop independent collapse(2)
    for ( int j = 1; j < n-1; j++ ) {
        for ( int i = 1; i < m-1; i++ ) {
            Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                                      A [j-1] [i] + A [j+1] [i]);
            error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
        }
    }
}
```

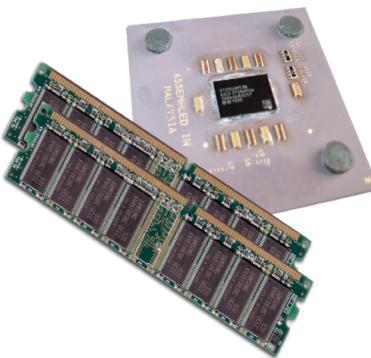
What is CUDA?

- NVIDIA CUDA Architecture
 - Expose GPU computing for general purpose
 - Retain performance
- CUDA C/C++
 - Based on industry-standard C/C++
 - Small set of extensions to enable heterogeneous programming
 - Straightforward APIs to manage devices, memory etc.
- This session introduces CUDA C/C++

Heterogeneous Computing

- Terminology:

- *Host* The CPU and its memory (host memory)
- *Device* The GPU and its memory (device memory)



Host



Device

<https://developer.nvidia.com/cuda-education>

© NVIDIA 2013

Heterogeneous Computing

```
#include <iostream>
#include <algorithm>

using namespace std;

#define N          1024
#define RADIUS     3
#define BLOCK_SIZE 16

__global__ void stencil_1d(int *in, int *out) {
    __shared__ int temp[BLOCK_SIZE + 2 * RADIUS];
    int index = threadIdx.x + RADIUS;
    int gindex = threadIdx.x + blockDim.x * blockDim.x;
    int lindex = threadIdx.x + RADIUS;

    // Read input elements into shared memory
    temp[lindex] = in[gindex];
    if (threadIdx.x < RADIUS) {
        temp[lindex - RADIUS] = in[gindex - RADIUS];
        temp[lindex + BLOCK_SIZE] = in[gindex + BLOCK_SIZE];
    }

    // Synchronize (ensure all the data is available)
    __syncthreads();

    // Apply the stencil
    int result = 0;
    for (int offset = -RADIUS ; offset <= RADIUS ; offset++)
        result += temp[lindex + offset];

    // Store the result
    out[gindex] = result;
}

void fill_ints(int *x, int n) {
    fill_n(x, n, 1);
}

int main(void) {
    int *in, *out;           // host copies of a, b, c
    int *d_in, *d_out;       // device copies of a, b, c
    int size = (N + 2*RADIUS) * sizeof(int);

    // Alloc space for host copies and setup values
    in = (int *)malloc(size); fill_ints(in, N + 2*RADIUS);
    out = (int *)malloc(size); fill_ints(out, N + 2*RADIUS);

    // Alloc space for device copies
    cudaMalloc((void **)&d_in, size);
    cudaMalloc((void **)&d_out, size);

    // Copy to device
    cudaMemcpy(d_in, in, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_out, out, size, cudaMemcpyHostToDevice);

    // Launch stencil_1d() kernel on GPU
    stencil_1d<<<N/BLOCK_SIZE,BLOCK_SIZE>>>(d_in + RADIUS,
d_out + RADIUS);

    // Copy result back to host
    cudaMemcpy(out, d_out, size, cudaMemcpyDeviceToHost);

    // Cleanup
    free(in); free(out);
    cudaFree(d_in); cudaFree(d_out);
    return 0;
}
```

parallel fn

serial code

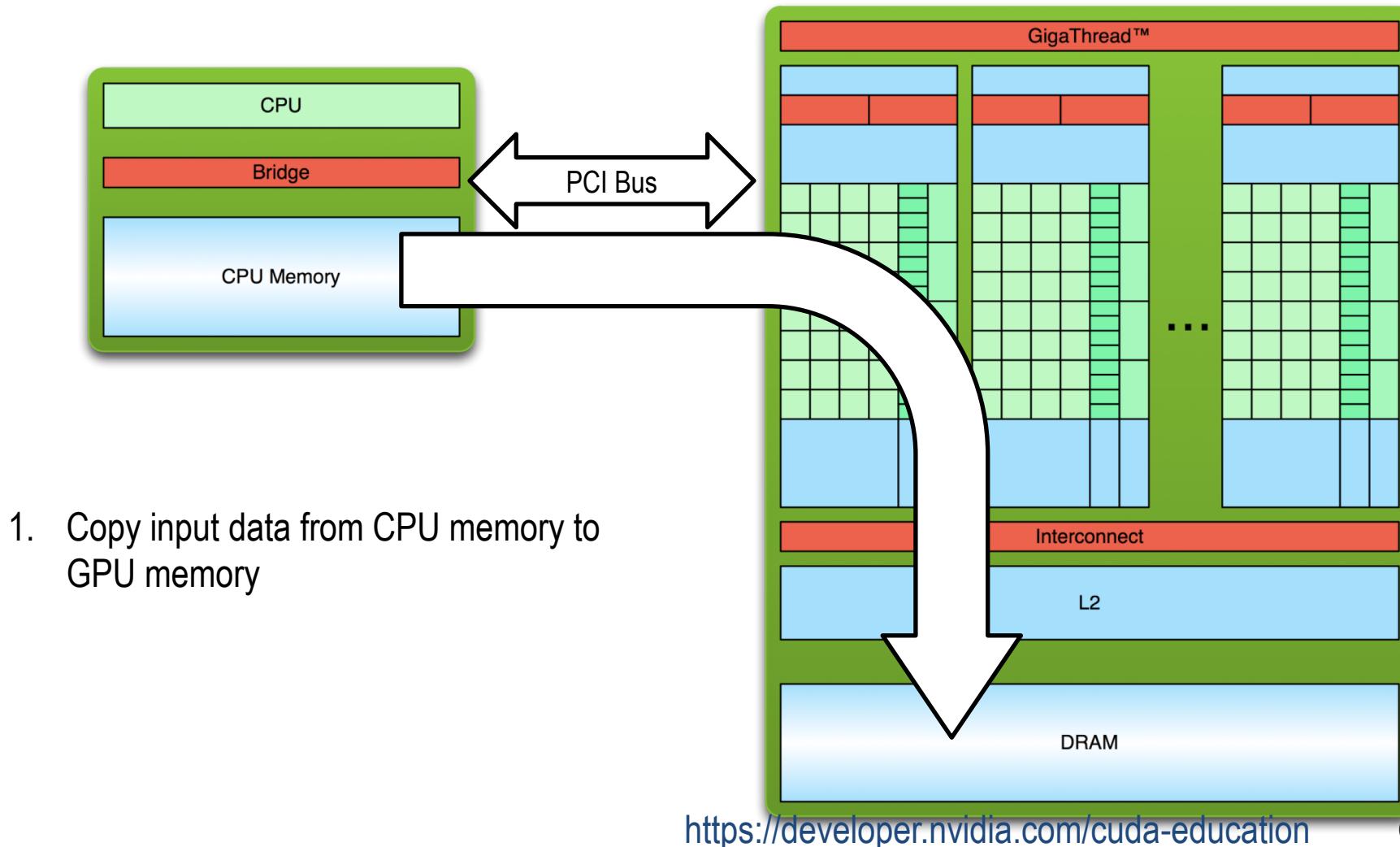
parallel code
serial code

<https://developer.nvidia.com/cuda-education>

© NVIDIA 2013

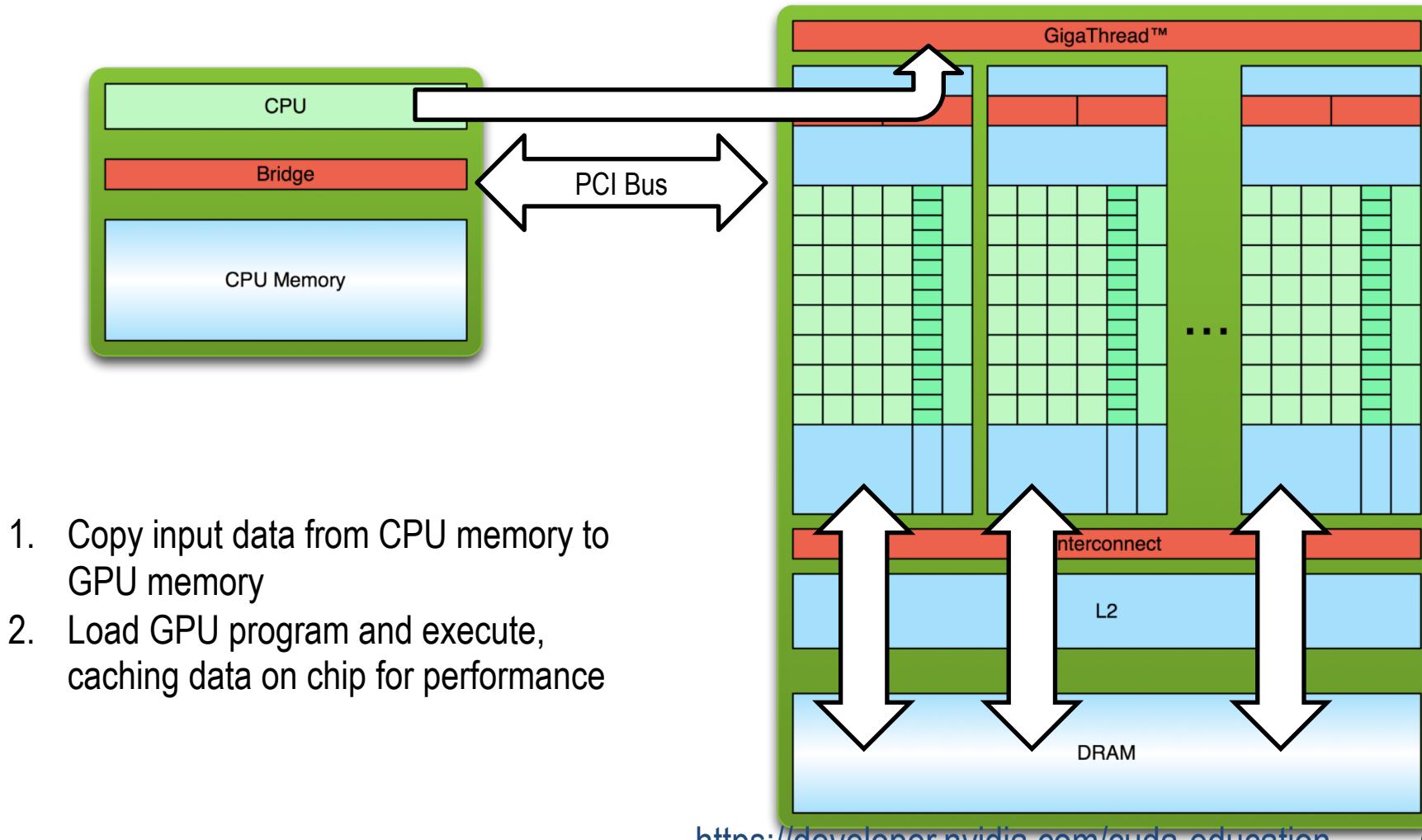


Simple Processing Flow



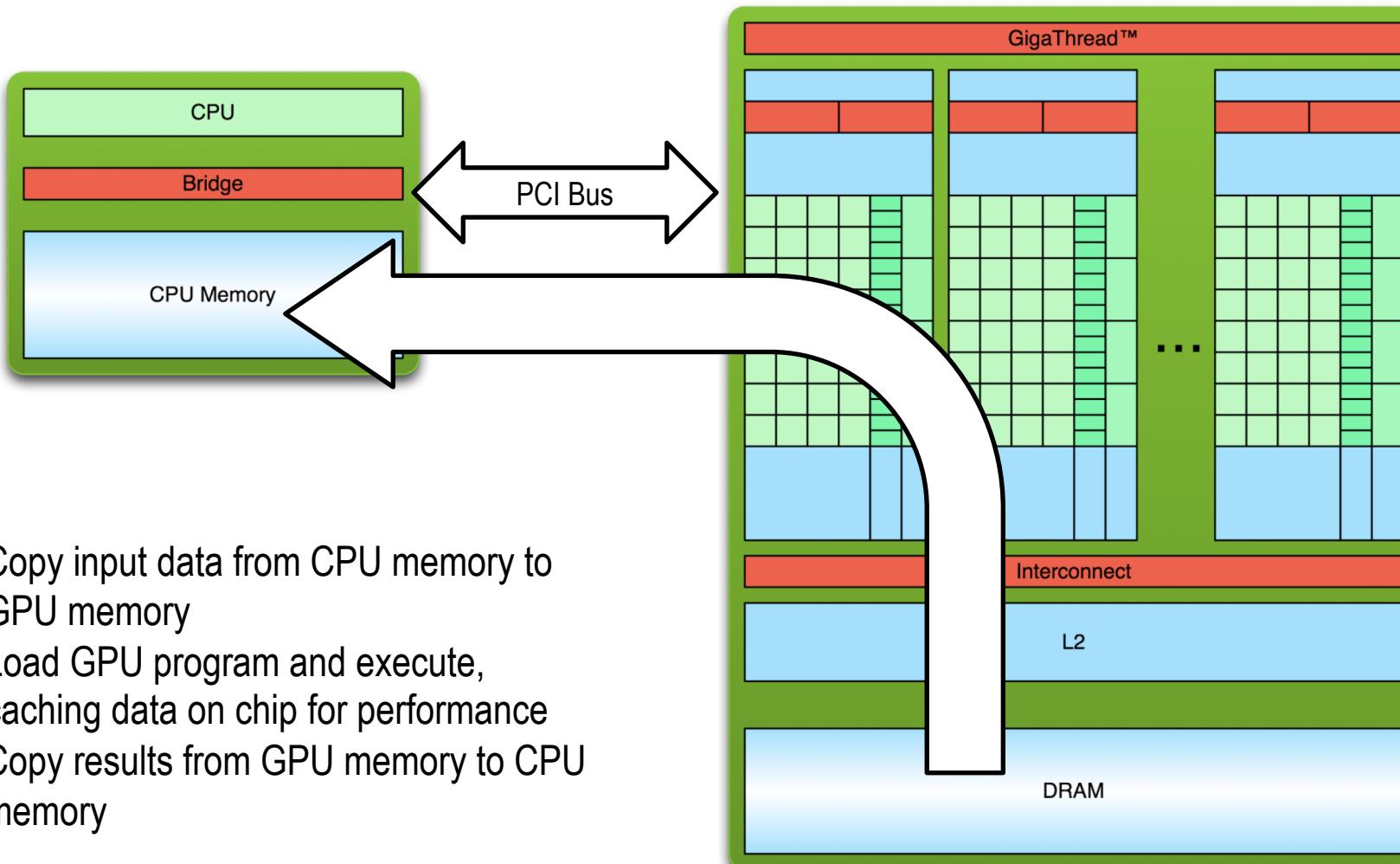
© NVIDIA 2013

Simple Processing Flow



© NVIDIA 2013

Simple Processing Flow



© NVIDIA 2013

NVIDIA CUDA Toolkit

<https://developer.nvidia.com/cuda-toolkit>

CUDA Toolkit

Develop, Optimize and Deploy GPU-Accelerated Apps

The NVIDIA® CUDA® Toolkit provides a development environment for creating high performance GPU-accelerated applications. With the CUDA Toolkit, you can develop, optimize, and deploy your applications on GPU-accelerated embedded systems, desktop workstations, enterprise data centers, cloud-based platforms and HPC supercomputers. The toolkit includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler, and a runtime library to build and deploy your application on major architectures including x86, Arm and POWER.

Using built-in capabilities for distributing computations across multi-GPU configurations, scientists and researchers can develop applications that scale from single GPU workstations to cloud installations with thousands of GPUs.

Download Now

A Quick Comparison between CUDA and C

C

```
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

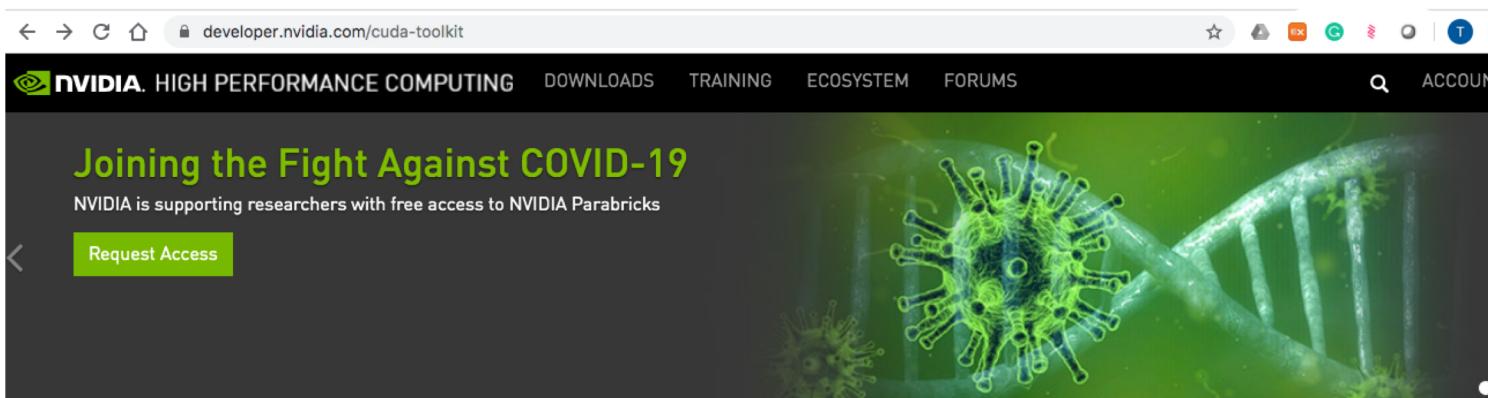
CUDA

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

CUDA

- <https://developer.nvidia.com/cuda-toolkit>
- <https://devblogs.nvidia.com/even-easier-introduction-cuda/>
- <https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/>
- <https://www.nvidia.com/docs/IO/116711/sc11-cuda-c-basics.pdf>



Develop, Optimize and Deploy GPU-accelerated Apps

The NVIDIA® CUDA® Toolkit provides a development environment for creating high performance GPU-accelerated applications. With the CUDA Toolkit, you can develop, optimize and deploy your applications on GPU-accelerated embedded systems, desktop workstations, enterprise data centers, cloud-based platforms and HPC supercomputers. The toolkit includes GPU-accelerated libraries, debugging and optimization tools, a C/C++ compiler and a runtime library to deploy your application.

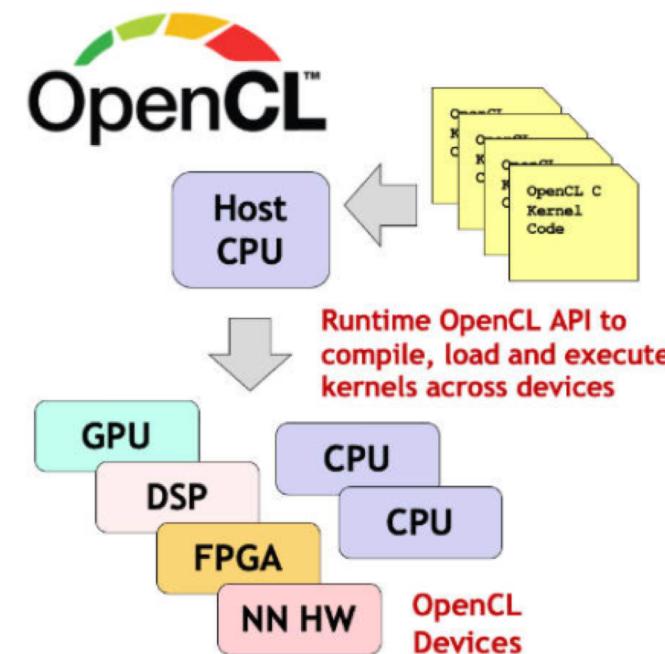
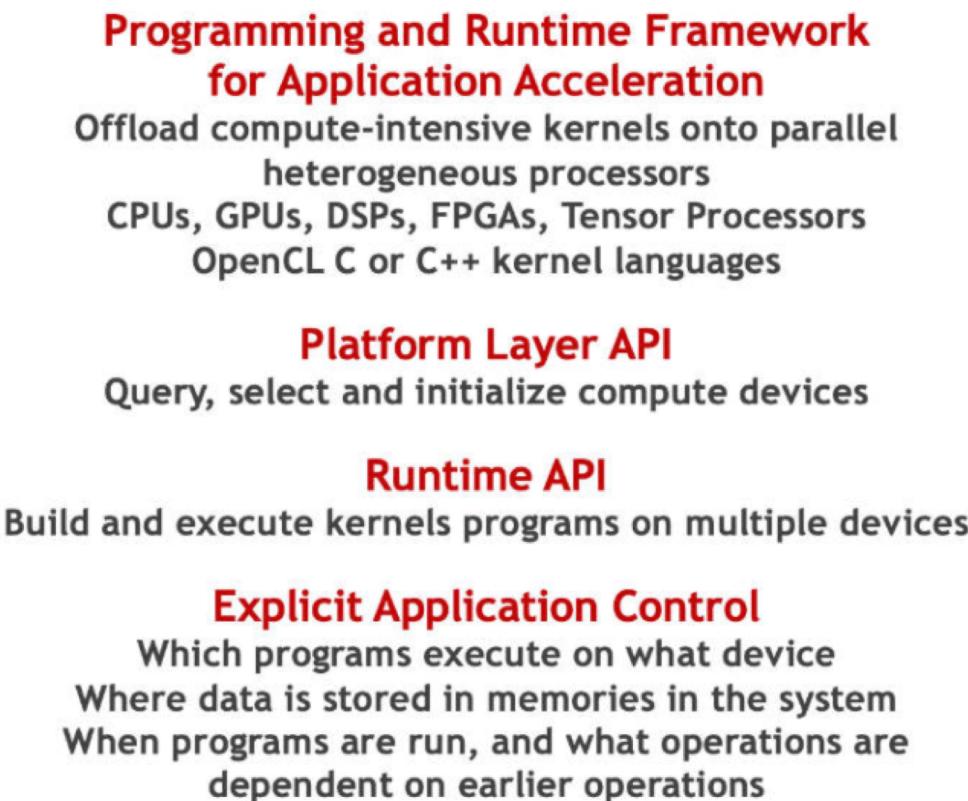
GPU-accelerated CUDA libraries enable drop-in acceleration across multiple domains such as linear algebra, image and video processing, deep learning and graph analytics. For developing custom algorithms, you can use available integrations with commonly used languages and numerical packages as well as well-published development APIs. Your CUDA applications can be deployed across all NVIDIA GPU families available on premise and on GPU instances in the cloud. Using built-in capabilities for distributing computations across multi-GPU configurations, scientists and researchers can develop applications that scale from single GPU workstations to cloud installations with thousands of GPUs.

OpenCL

- <https://www.khronos.org/opencl/>

OpenCL for Low-level Parallel Programming

OpenCL speeds applications by offloading their most computationally intensive code onto accelerator processors - or devices. OpenCL developers use C or C++-based kernel languages to code programs that are passed through a device compiler for parallel execution on accelerator devices.



Complements GPU-only APIs

- Simpler programming model
- Relatively lightweight run-time
- More language flexibility, e.g., pointers
- Rigorously defined numeric precision

OpenACC

- <https://www.openacc.org/>

What is OpenACC?

The OpenACC Organization is dedicated to helping the research and developer community advance science by expanding their accelerated and parallel computing skills. We have 3 areas of focus: participating in computing ecosystem development, providing training and education on programming models, resources and tools, and developing the OpenACC specification.

Get Started

or [take the next steps](#)

```
#pragma acc data copy(A) create(Anew)
while ( error > tol && iter < iter_max ) {
    error = 0.0;
#pragma acc kernels
{
#pragma acc loop independent collapse(2) reduction(max:error)
    for ( int j = 1; j < n-1; j++ ) {
        for ( int i = 1; i < m-1; i++ ) {
            Anew [j] [i] = 0.25 * ( A [j] [i+1] + A [j] [i-1] +
                A [j-1] [i] + A [j+1] [i]);
            error = max ( error, fabs (Anew [j] [i] - A [j] [i]));
        }
    }
    ...
}
```

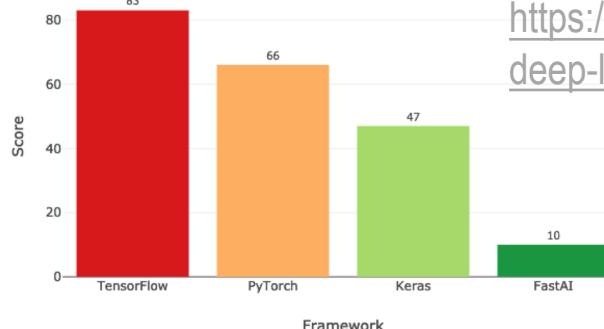
Popular Deep Learning Framework



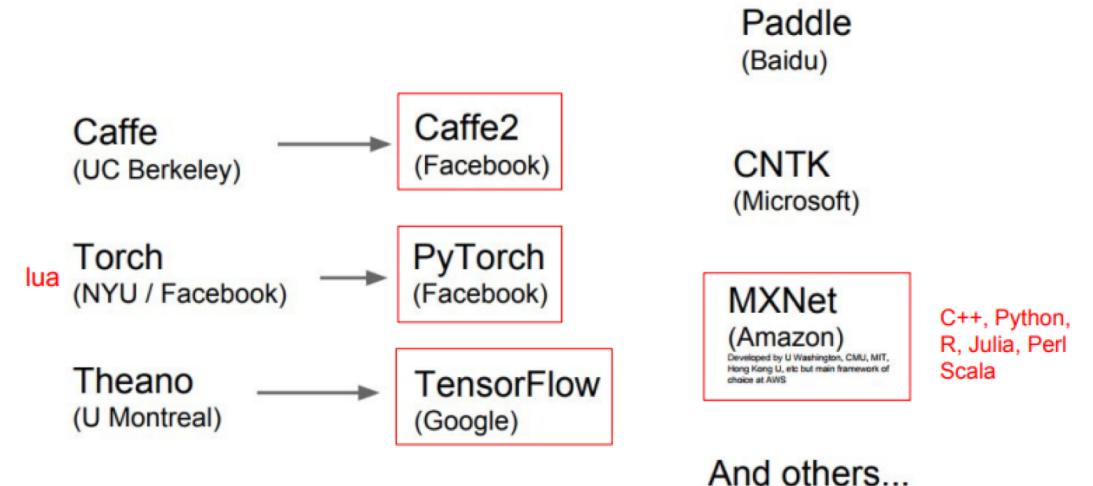
Deep Learning with Fast.Ai



Deep Learning Framework Six-Month Growth Scores 2019

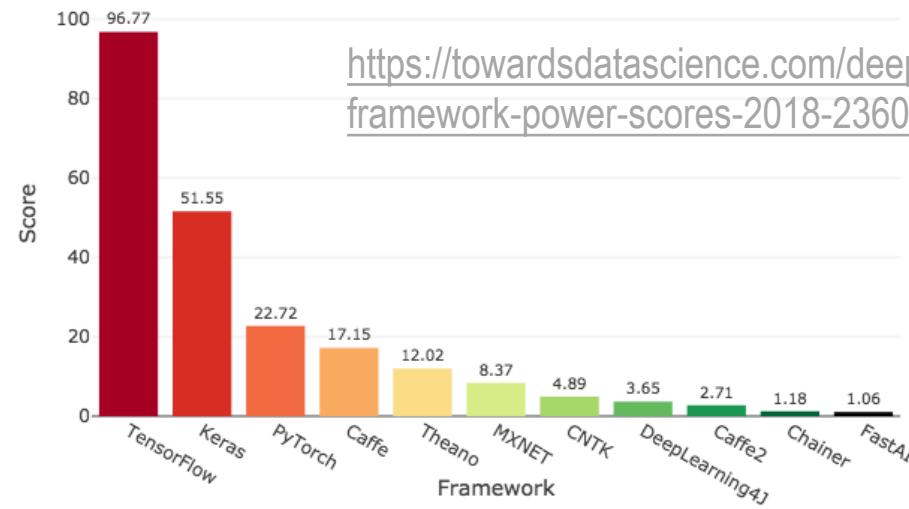


<https://www.kdnuggets.com/2019/05/which-deep-learning-framework-growing-fastest.html>



https://web.cs.ucdavis.edu/~yjlee/teaching/ecs289g-winter2018/Pytorch_Tutorial.pdf

Deep Learning Framework Power Scores 2018



<https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a>

Machine/Deep Learning + GPU

TensorFlow

Install Learn API Resources Community Why TensorFlow

Search GitHub

Install

Install TensorFlow

Packages pip Docker

Additional setup **GPU support**

Problems

Build from source Linux / macOS Windows Raspberry Pi

Language bindings Java C Go

Apply to speak at TensorFlow World. Deadline April 23rd. [Propose talk](#)

TensorFlow > Install 

GPU support

TensorFlow GPU support requires an assortment of drivers and libraries. To simplify installation and avoid library conflicts, we recommend using a [TensorFlow Docker image with GPU support](#) (Linux only). This setup only requires the [NVIDIA® GPU drivers](#).

These install instructions are for the latest release of TensorFlow. See the [tested build configurations](#) for CUDA and cuDNN versions to use with older TensorFlow releases.

Pip package

See the [pip install guide](#) for available packages, systems requirements, and instructions. To `pip` install a TensorFlow package with GPU support, choose a stable or development package:

```
$ pip install tensorflow-gpu # stable  
$ pip install tf-nightly-gpu # preview
```

Contents

- Pip package
 - TensorFlow 2.0
 - Alpha
- Hardware requirements
- Software requirements
- Linux setup
- Install CUDA with apt
- Windows setup

- <https://www.amd.com/en/graphics/servers-radeon-instinct-deep-learning>

The screenshot shows the AMD Deep Learning Solutions page. At the top, there's a navigation bar with the AMD logo, menu items for PROCESSORS, GRAPHICS, GAMING, SHOP, and DRIVERS & SUPPORT, and a search bar. Below the header, the page title "Deep Learning Solutions" is displayed next to a "Share this page" button with social media icons for Facebook, Twitter, Pinterest, Email, and Print. A large, dark banner features the text "Unleash Deep Learning Discovery" overlaid on a background of glowing blue network nodes and lines. Below this, a navigation menu includes links for OVERVIEW, SOLUTIONS, DEEP LEARNING FRAMEWORKS, and TECHNOLOGIES. A call-to-action section at the bottom left says "Ready-To-Deploy Deep Learning Solutions". On the right side, a sidebar contains a "Why We Use Cookies" section with a detailed explanation of cookie usage and links to AMD's privacy policy and cookie policy.

AMD

PROCESSORS ▾ GRAPHICS ▾ GAMING ▾ SHOP ▾ DRIVERS & SUPPORT

Search

Deep Learning Solutions

Share this page

f t p e +

Unleash Deep Learning Discovery

OVERVIEW SOLUTIONS DEEP LEARNING FRAMEWORKS TECHNOLOGIES

Ready-To-Deploy Deep Learning Solutions

AMD unleashes your deep learning innovation with Radeon Instinct™ server solutions.

Why We Use Cookies

This site uses cookies to make your browsing experience more convenient and personal. Cookies store useful information on your computer to help us improve the efficiency and relevance of our site for you. In some cases, they are essential to making the site work properly. By accessing this site, you consent to the use of cookies. For more information, refer to AMD's [privacy policy](#) and [cookie policy](#).

PyTorch + GPU

- <https://pytorch.org/>

The screenshot shows the official PyTorch website. At the top, there is a navigation bar with links: Get Started, Ecosystem, Mobile, Blog, Tutorials, Docs, Resources, GitHub, and a search icon. Below the navigation bar, the main title "FROM RESEARCH TO PRODUCTION" is displayed in large white text against a purple-to-orange gradient background. Underneath the title, a subtitle reads: "An open source machine learning framework that accelerates the path from research prototyping to production deployment." A prominent "Get Started" button is located below the subtitle. At the bottom of the page, there is a black banner containing three news items with arrows for navigation:

- PyTorch 1.5 released, new and updated APIs including C++ frontend API parity with Python.
- PyTorch library updates including new model serving library
- PyTorch 1.4 is now available - adds ability to do fine grain build level customization for PyTorch Mobile, updated domain libraries, and new experimenta...

PyTorch and PyCUDA

- What is PyTorch?
 - <https://pytorch.org/>
 - Open source machine learning library
 - Developed by Facebook's AI Research lab
 - It leverages the power of GPUs
 - Automatic computation of gradients
 - Makes it easier to test and develop new ideas.
- What is PyCUDA?
 - <https://document.tician.de/pycuda/>
 - Pythonic access to Nvidia's CUDA parallel computation API.
 - Convenience. Abstractions like `pycuda.compiler.SourceModule` and `pycuda.gpuarray.GPUArray` make CUDA programming even more convenient than with Nvidia's C-based runtime.

Again, this course is not a Machine Learning, Deep Learning, or Natural Language Processing. course. If you are interested in ML/DL/NLP, consider to take the course coming soon.

Multi-GPU Framework Comparison

Train+Val w/ data-loader + data-augmentation on real-data on SSD

DL Library	1xV100/CUDA 9/CuDNN 7	4xV100/CUDA 9/CuDNN 7
Pytorch	27min	10min
Keras(TF)	38min	18min
Tensorflow	33min	22min
Chainer	29min	8min
MXNet(Gluon)	29min	10min

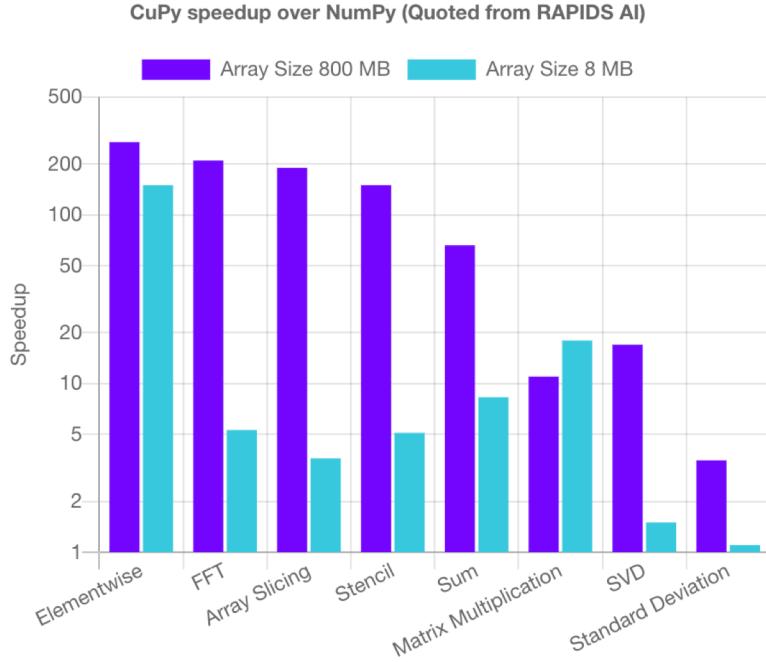
Train w/ synthetic-data in RAM

DL Library	1xV100/CUDA 9/CuDNN 7	4xV100/CUDA 9/CuDNN 7
Pytorch	25min	8min
Keras(TF)	36min	15min
Tensorflow	25min	14min
Chainer	27min	7min
MXNet(Gluon)	28min	8min

<https://medium.com/@iliakarmanov/multi-gpu-rosetta-stone-d4fa96162986>

Numpy vs CuPy

- <https://cupy.dev/>



HIGH PERFORMANCE WITH GPU

CuPy is an open-source array library for GPU-accelerated computing with Python. CuPy utilizes CUDA Toolkit libraries including cuBLAS, cuRAND, cuSOLVER, cuSPARSE, cuFFT, cuDNN and NCCL to make full use of the GPU architecture. The figure shows CuPy speedup over NumPy. Most operations perform well on a GPU using CuPy out of the box. CuPy speeds up some operations more than 100X. Read the original benchmark article [Single-GPU CuPy Speedups](#) on the RAPIDS AI Medium blog.



- <https://towardsdatascience.com/heres-how-to-use-cupy-to-make-numpy-700x-faster-4b920dda1f56>

Here's How to Use CuPy to Make Numpy Over 10X Faster

GPU CUDA

High-Performance Computing

Summer 2021 at GIST

Tae-Hyuk (Ted) Ahn

Department of Computer Science
Program of Bioinformatics and Computational Biology
Saint Louis University



SAINT LOUIS
UNIVERSITY™

— EST. 1818 —

← → ⌛ ⌂ 🔒 developer.nvidia.com/cuda-zone

NVIDIA DEVELOPER HOME BLOG FORUMS DOCS DOWNLOADS TRAINING ACCOUNT

DOWNLOADS TRAINING ECOSYSTEM FORUMS

TRAIN MODELS FASTER

Explore exclusive discounts for higher education

Learn more



○ ○ ○ ● ○

CUDA® is a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs). With CUDA, developers are able to dramatically speed up computing applications by harnessing the power of GPUs.

In GPU-accelerated applications, the sequential part of the workload runs on the CPU – which is optimized for single-threaded performance – while the compute intensive portion of the application runs on thousands of GPU cores in parallel. When using CUDA, developers program in popular languages such as C, C++, Fortran, Python and MATLAB and express parallelism through extensions in the form of a few basic keywords.

The [CUDA Toolkit](#) from NVIDIA provides everything you need to develop GPU-accelerated applications. The CUDA Toolkit includes GPU-accelerated libraries, a compiler, development tools and the CUDA runtime.

Download Now >

How to compile?

```
1 #include <stdio.h>
2
3 int main() {
4     printf("Hello, world!\n");
5     return 0;
6 }
```

- \$ nvcc hello.cu -o hello
- ./hello

```
[ahnt@gpu1:CUDA]$ nvcc hello.cu -o hello
[ahnt@gpu1:CUDA]$ ./hello
Hello, world!
```

A Quick Comparison between CUDA and C

C

```
void c_hello(){
    printf("Hello World!\n");
}

int main() {
    c_hello();
    return 0;
}
```

CUDA

```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

int main() {
    cuda_hello<<<1,1>>>();
    return 0;
}
```

Compiling CUDA Program

- Compiling a CUDA program is similar to C program. NVIDIA provides a CUDA compiler called **nvcc** in the CUDA toolkit to compile CUDA code, typically stored in a file with extension .cu. For example

```
$ nvcc cuda_helloworld.cu -o cuda_helloworld
```

Hello World!

```
1 #include <stdio.h>
2
3 __global__ void mykernel() {
4     printf("Hello World from GPU!\n");
5 }
6
7 int main() {
8     mykernel<<<1,1>>>();
9     return 0;
10 }
11
```

Output:

```
[ahnt@gpu1:CUDA]$ nvcc hello_cuda.cu -o hello_cuda
[ahnt@gpu1:CUDA]$ ./hello_cuda
[ahnt@gpu1:CUDA]$ █
```

NO Output.. ??

Hello World! with Device Code

```
__global__ void mykernel(void) {  
}  
  
int main(void) {  
    mykernel<<<1,1>>>();  
    cout << "Hello CUDA World!" << endl;  
    return 0;  
}
```

- Two new syntactic elements...

Hello World! with Device Code

```
__global__ void mykernel(void) {  
}
```

- CUDA C/C++ keyword `__global__` indicates a function that:
 - Runs on the device
 - Is called from host code
- nvcc separates source code into host and device components
 - Device functions (e.g. `mykernel()`) processed by NVIDIA compiler
 - Host functions (e.g. `main()`) processed by standard host compiler
 - `gcc, cl.exe`

Hello World! with Device Code

```
mykernel<<<1,1>>>();
```

- Triple angle brackets mark a call from *host* code to *device* code
 - Also called a “kernel launch”
 - We’ll return to the parameters (1,1) in a moment
- That’s all that is required to execute a function on the GPU!

Vector addition using CUDA

vector_add.c

```
1 #include <math.h>
2 #include <assert.h>
3
4 #define N 10000000
5 #define MAX_ERR 1e-6
6
7 void vector_add(float *out, float *a, float *b, int n) {
8     for(int i = 0; i < n; i++){
9         out[i] = a[i] + b[i];
10    }
11 }
12
13 int main(){
14     float *a, *b, *out;
15
16     // Allocate memory
17     a = (float*)malloc(sizeof(float) * N);
18     b = (float*)malloc(sizeof(float) * N);
19     out = (float*)malloc(sizeof(float) * N);
20
21     // Initialize array
22     for(int i = 0; i < N; i++){
23         a[i] = 1.0f;
24         b[i] = 2.0f;
25     }
26
27     // Main function
28     vector_add(out, a, b, N);
29
30     // Verification
31     for(int i = 0; i < N; i++){
32         assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
33     }
34
35     printf("out[0] = %f\n", out[0]);
36     printf("PASSED\n");
37 }
```

Converting vector addition to CUDA

- Copy vector_add.c to vector_add.cu
- Convert vector_add() to GPU kernel

```
__global__ void vector_add(float *out, float *a, float *b, int n) {
    for(int i = 0; i < n; i++){
        out[i] = a[i] + b[i];
    }
}
```

- Change vector_add() call in main() to kernel call

```
// Main function
vector_add<<<1,1>>>(out, a, b, N);
```

- Compile and run the program

Converting vector addition to CUDA

- You will notice that the program does not work correctly.
- The reason is CPU and GPUs are separate entities.
- Both have their own memory space. CPU cannot directly access GPU memory, and vice versa.
- In CUDA terminology, CPU memory is called **host memory** and GPU memory is called **device memory**. Pointers to CPU and GPU memory are called host pointer and device pointer, respectively.

Converting vector addition to CUDA

For data to be accessible by GPU, it must be presented in the device memory. CUDA provides APIs for allocating device memory and data transfer between host and device memory. Following is the common workflow of CUDA programs.

1. Allocate host memory and initialized host data
2. Allocate device memory
3. Transfer input data from host to device memory
4. Execute kernels
5. Transfer output from device memory to host

So far, we have done step 1 and 4. We will add step 2, 3, and 5 to our vector addition program and finish this exercise.

Device memory management

- CUDA provides several functions for allocating device memory. The most common ones are `cudaMalloc()` and `cudaFree()`. The syntax for both functions are as follow

```
cudaMalloc(void **devPtr, size_t count);  
cudaFree(void *devPtr);
```

- `cudaMalloc()` allocates memory of size `count` in the device memory and updates the device pointer `devPtr` to the allocated memory.
- `cudaFree()` deallocates a region of the device memory where the device pointer `devPtr` points to.
- They are comparable to `malloc()` and `free()` in C, respectively

Memory transfer

- Transferring date between host and device memory can be done through `cudaMemcpy` function, which is similar to `memcpy` in C. The syntax of `cudaMemcpy` is as follow

```
cudaMemcpy(void *dst, void *src, size_t count, cudaMemcpyKind kind)
```

- The function copy a memory of size count from src to dst.
- `kind` indicates the direction. For typical usage, the value of kind is either `cudaMemcpyHostToDevice` or `cudaMemcpyDeviceToHost`. There are other possible values but we will not touch them in this course.

Completing vector addition

- Allocate and deallocate device memory for array a, b, and out.
- Transfer a, b, and out between host and device memory.

```
// Allocate device memory
cudaMalloc((void**)&d_a, sizeof(float) * N);
cudaMalloc((void**)&d_b, sizeof(float) * N);
cudaMalloc((void**)&d_out, sizeof(float) * N);

// Transfer data from host to device memory
cudaMemcpy(d_a, a, sizeof(float) * N, cudaMemcpyHostToDevice);
cudaMemcpy(d_b, b, sizeof(float) * N, cudaMemcpyHostToDevice);

// Executing kernel
vector_add<<<1,1>>>(d_out, d_a, d_b, N);

// Transfer data back to host memory
cudaMemcpy(out, d_out, sizeof(float) * N, cudaMemcpyDeviceToHost);

// Verification
for(int i = 0; i < N; i++){
    assert(fabs(out[i] - a[i] - b[i]) < MAX_ERR);
}
printf("out[0] = %f\n", out[0]);
printf("PASSED\n");

// Deallocate device memory
cudaFree(d_a);
cudaFree(d_b);
cudaFree(d_out);

// Deallocate host memory
free(a);
free(b);
free(out);
```

Compile and measure performance.

```
$ nvcc vector_add.cu -o vector_add  
$ time ./vector_add  
$ nvprof ./vector_add
```

- Using time does not give much information about the program performance. NVIDIA provides a cmdline profiler tool called nvprof, which give a more insight information of CUDA program performance.

```
ubuntu@ip-172-31-5-141:~/cuda_lab/instructor$ time ./vector_add  
out[0] = 3.000000  
PASSED  
  
real    0m3.189s  
user    0m2.012s  
sys     0m1.172s  
ubuntu@ip-172-31-5-141:~/cuda_lab/instructor$ nvprof ./vector_add  
==2117== NVPROF is profiling process 2117, command: ./vector_add  
out[0] = 3.000000  
PASSED  
==2117== Profiling application: ./vector_add  
==2117== Profiling result:  
          Type  Time(%)      Time      Calls       Avg       Min       Max  Name  
GPU activities:  99.28%  4.22754s           1  4.22754s  4.22754s  4.22754s  vector_add(float*, float*, float*, int)  
                  0.41%  17.392ms           1  17.392ms  17.392ms  17.392ms  [CUDA memcpy DtoH]  
                  0.31% 13.405ms            2  6.7026ms  6.6934ms  6.7118ms  [CUDA memcpy HtoD]
```

Going Parallel

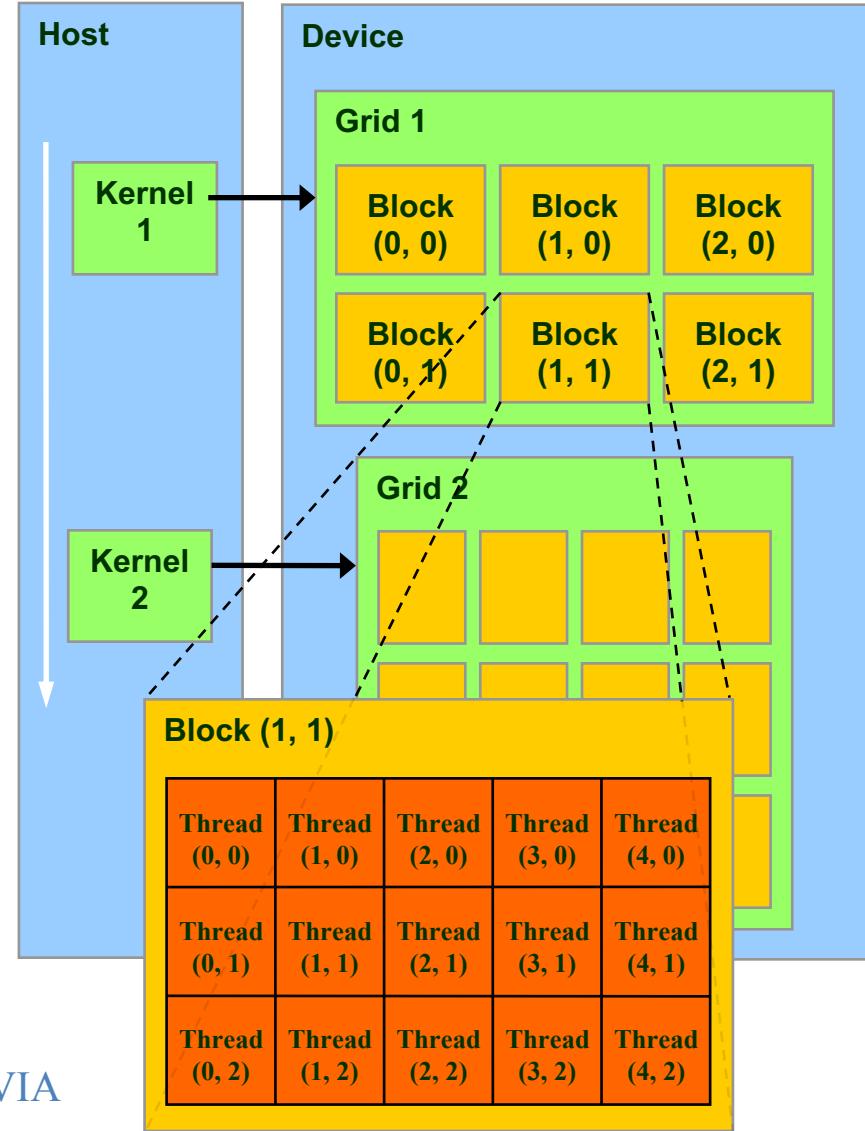
- CUDA use a kernel execution configuration `<<<...>>>` to tell CUDA runtime how many threads to launch on GPU. CUDA organizes threads into a group called "thread block". Kernel can launch multiple thread blocks, organized into a "grid" structure.

```
<<< M , T >>>
```

- Which indicate that a kernel launches with a grid of M thread **blocks**. Each thread block has T parallel **threads**.

Thread Batching: Grids and Blocks

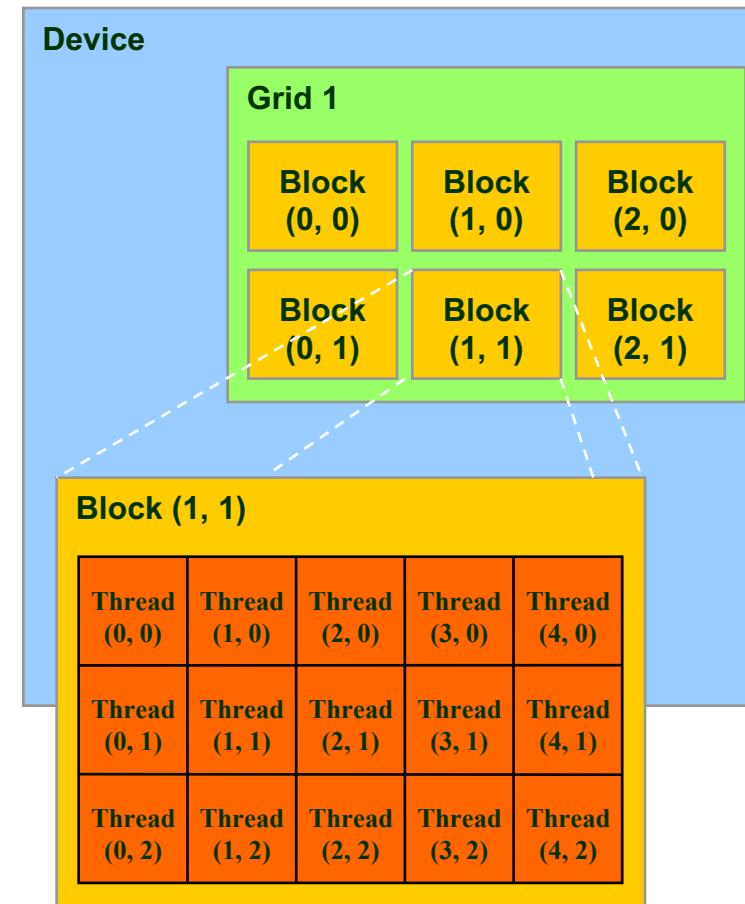
- A kernel is executed as a **grid of thread blocks**
 - All threads share data memory space
- A **thread block** is a batch of threads that can **cooperate** with each other by:
 - Synchronizing their execution
 - For hazard-free shared memory accesses
 - Efficiently sharing data through a low latency **shared memory**
- Two threads from two different blocks cannot cooperate



Courtesy: NDVIA

Block and Thread IDs

- Threads and blocks have IDs
 - So each thread can decide what data to work on
 - Block ID: 1D, 2D, or 3D
(`blockIdx.{x,y,z}`)
 - Thread ID: 1D, 2D, or 3D
(`threadIdx.{x,y,z}`)
- Simplifies memory addressing when processing multidimensional data
 - Image processing
 - Solving PDEs on volumes
 - ...



Courtesy: NDVIA

Built-in Variables for Block and Thread

Built-in variables:

`threadIdx.{x,y,z}` – thread ID within a block

`blockIdx.{x,y,z}` – block ID within a grid

`blockDim.{x,y,z}` – number of threads within a block

`gridDim.{x,y,z}` – number of blocks within a grid

`kernel<<<nBlocks,nThreads>>>(args)`

Invokes a parallel kernel function on a grid of nBlocks where each block instantiates nThreads concurrent threads

Parallelizing vector addition using multithread

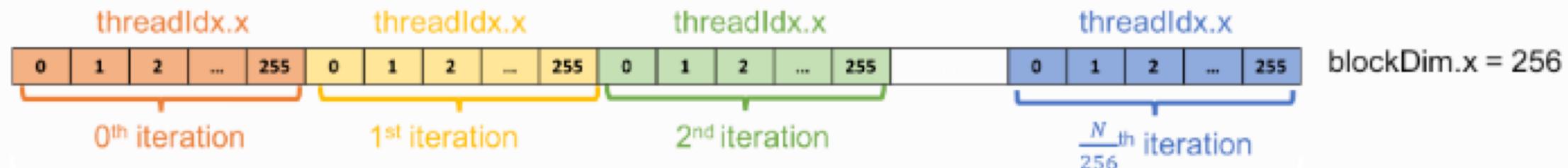
```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

- threadIdx.x contains the index of the thread within the block
- blockDim.x contains the size of thread block (number of threads in the thread block).
- For the vector_add() configuration, the value of threadIdx.x ranges from 0 to 255 and the value of blockDim.x is 256.

Parallelizing vector addition using multithread

```
vector_add <<< 1 , 256 >>> (d_out, d_a, d_b, N);
```

Parallelizing vector addition using multithread



- For the k-th thread, the loop starts from k-th element and iterates through the array with a loop stride of 256.
- For example, in the 0-th iteration, the k-th thread computes the addition of k-th element.
- In the next iteration, the k-th thread computes the addition of (k+256)-th element, and so on.

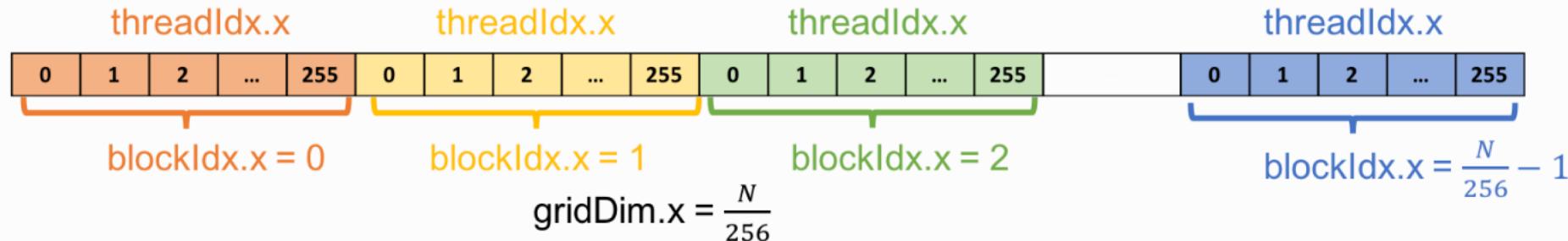
vector_add_thread.cu

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <math.h>
4 #include <assert.h>
5 #include <cuda.h>
6 #include <cuda_runtime.h>
7
8 #define N 10000000
9 #define MAX_ERR 1e-6
10
11 __global__ void vector_add(float *out, float *a, float *b, int n) {
12     int index = threadIdx.x;
13     int stride = blockDim.x;
14
15     for(int i = index; i < n; i += stride){
16         out[i] = a[i] + b[i];
17     }
18 }
19
```

40

```
44     // Executing kernel
45     vector_add<<<1,256>>>(d_out, d_a, d_b, N);
46
```

Adding more threads blocks

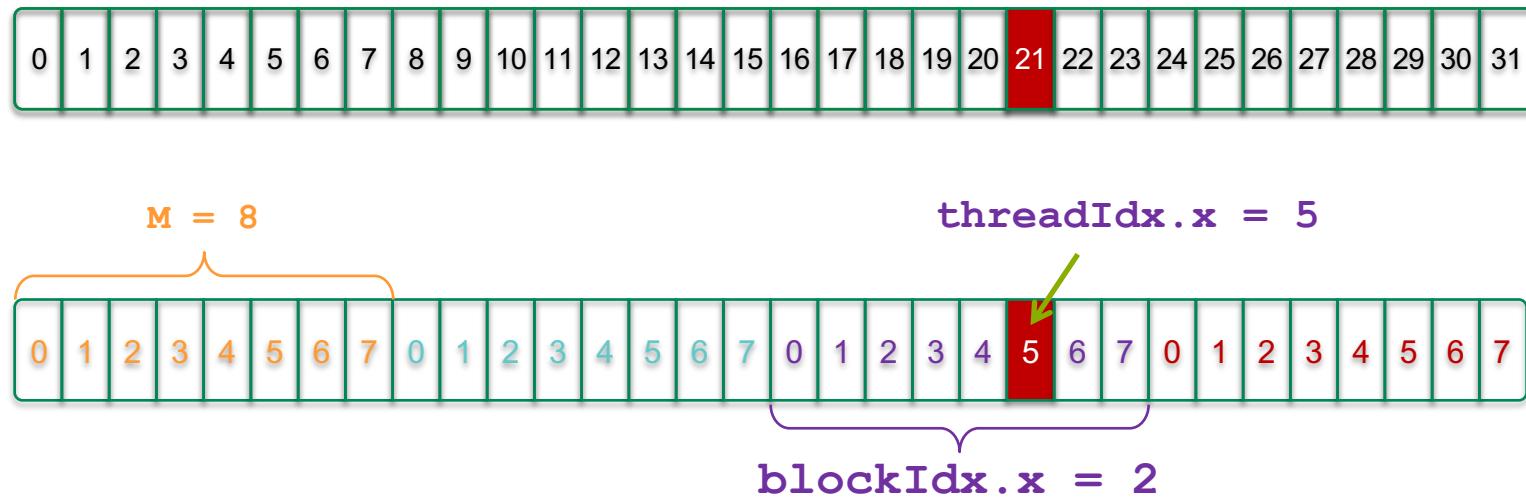


- We will use two of them: `blockIdx.x` and `gridDim.x`.
- `blockIdx.x` contains the index of the block within the grid
- `gridDim.x` contains the size of the grid
- With 256 threads per thread block, we need at least $N/256$ thread blocks to have a total of N threads. To assign a thread to a specific element, we need to know a unique index for each thread. Such index can be computed as follow

```
int tid = blockIdx.x * M + threadIdx.x;
```

Indexing Arrays: Example

- Which thread will operate on the red element?



```
int index = threadIdx.x + blockIdx.x * M;  
= 5 + 2 * 8;  
= 21;
```

Vector Addition with Blocks and Threads

- Use the built-in variable `blockDim.x` for threads per block

```
int index = threadIdx.x + blockIdx.x * blockDim.x;
```

- Combined version of `add()` to use parallel threads *and* parallel blocks

```
__global__ void add(int *a, int *b, int *c) {
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    c[index] = a[index] + b[index];
}
```

- What changes need to be made in `main()`?

main()

```
// Launch add() kernel on GPU
add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
```

vector_sum_bt.cu

```
#include <iostream>
#include <cstdlib>

#define N (4*8)
#define THREADS_PER_BLOCK 8

using namespace std;

__global__ void add(int *a, int *b, int *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main(int argc, char *argv[])
{
    int a[N], b[N], c[N];           // host copies of a, b, c
    int *d_a, *d_b, *d_c;          // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, N*size);
    cudaMalloc((void **)&d_b, N*size);
    cudaMalloc((void **)&d_c, N*size);

    // Setup input values
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // Copy inputs to device
    cudaMemcpy(d_a, &a, N*size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, N*size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b,
d_c);
    //add<<<4,8>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, N*size, cudaMemcpyDeviceToHost);

    // Display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free the memory allocated on the GPU
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

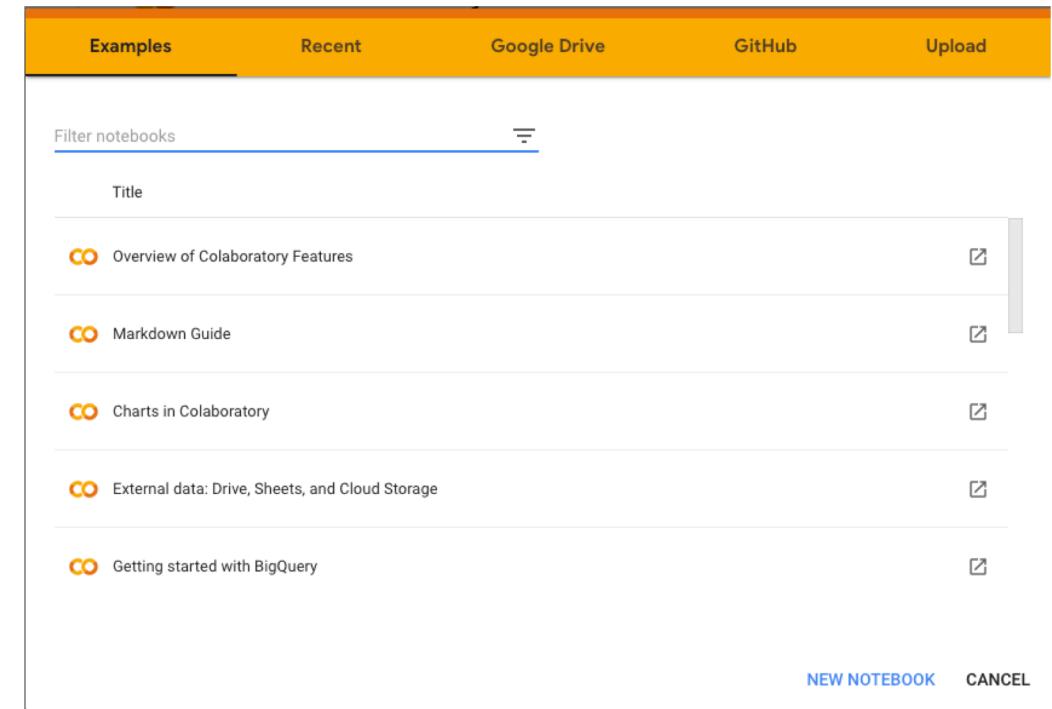
CUDA Thread Indexing Cheatsheet

- <http://cs.calvin.edu/curriculum/cs/374/CUDA/CUDA-Thread-Indexing-Cheatsheet.pdf>

Google Colab

Colaboratory, or "Colab" for short, allows you to write and execute Python in your browser, with

- Zero configuration required
- Free access to GPUs
- Easy sharing
- Built on top of Jupyter Notebook
- Google Colab notebooks are stored on the drive



Google Colab Let's Begin!

Create a Colab Notebook

- 1. Open Google Colab.
- 2. Click on 'New Notebook'.

OR

1. Open Google Drive.
2. Create a new folder for the project.
3. Click on 'New' > 'More' > 'Colaboratory'.

I have below default Colab Notebooks directory. Check it on your Google Drive and create HPC_Lab



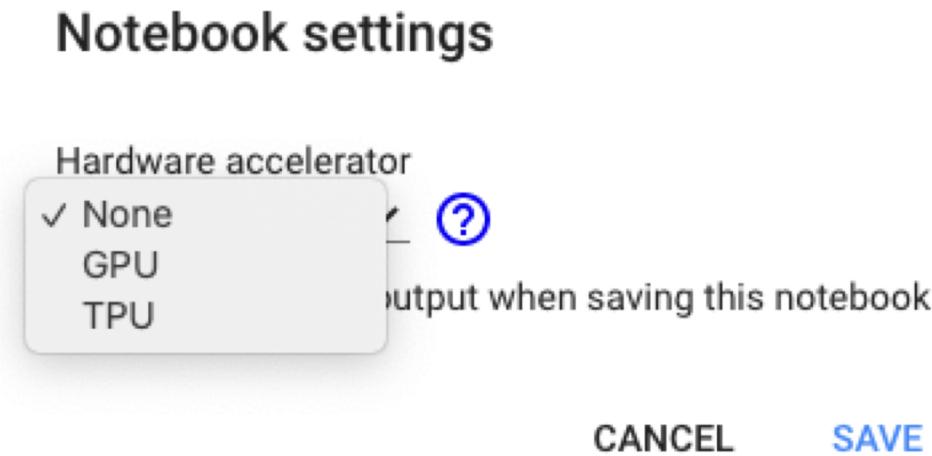
me

Feb 12, 2020

Setting GPU Accelerator

The default hardware of Google Colab is CPU or it can be GPU.

1. Click on ‘Edit’ > ‘Notebook Settings’ > ‘Hardware Accelerator’ > ‘GPU’.

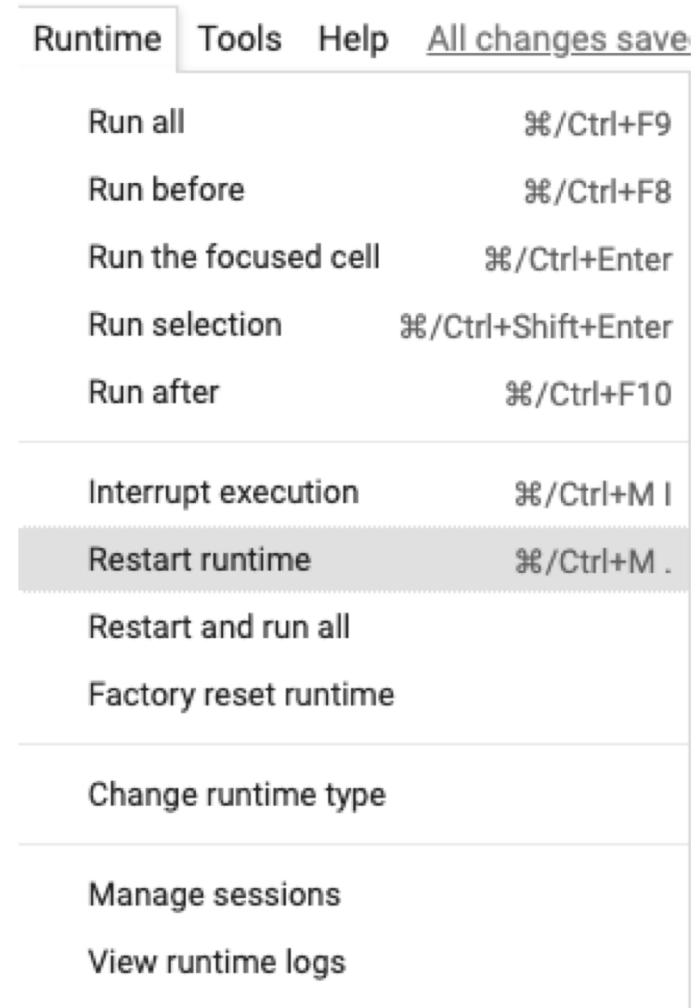


Running a Cell

- Make sure the runtime is connected. The notebook shows a green check and ‘Connected’ on the top right corner.
- There are various runtime options in ‘Runtime’.

OR

- To run the current cell, press SHIFT + ENTER.



Running a Cell

- Bash commands can be run by prefixing the command with '!'.

```
[1] !ls
```

```
sample_data
```



```
!ls sample_data/
```

anscombe.json	mnist_test.csv
california_housing_test.csv	mnist_train_small.csv
california_housing_train.csv	README.md

Connect to your Google Drive

```
1 from google.colab import drive  
2 drive.mount('/content/drive/')
```

- When you run the code above, you should see a result like this:

A screenshot of a Jupyter Notebook cell. The cell contains the following Python code:

```
from google.colab import drive  
drive.mount('/content/drive/')
```

Below the code, there is a message: "Go to this URL in a browser: https://accounts.google.com/o/oauth2/auth?client_id=947318989803-6bn6qk8qdgf4n4g3p". Underneath the link, it says "Enter your authorization code:" followed by an empty text input field.

- Click the link, copy verification code and paste it to text box. Then, it will be

```
[5] from google.colab import drive  
     drive.mount('/content/drive/')
```

Mounted at /content/drive/

Connect to your Google Drive

```
[6] !ls
```

```
drive sample_data
```

```
[8] !ls drive/MyDrive/
```

```
22.160P_A_Tilt_FPS26_Live5
23.160M_S_Automotive_Live5
'BCB-5250 Spring 2021 Survey.gform'
Career
'Colab Notebooks'
ColabTest
Commitment
'CSCI-2100 5002 Final Exam Schedule Voting.gform'
'CSCI-4850 5850 HPC Spring 2021 Survey.gform'
EBSCO
Kirmizis_2007_Sir2_wildtype_histone_H3.merge.bw
PersonalDocuments
Research
Software
Teaching
'-$summary.docx'
'Worksheet (1).gform'
Worksheet.gform
```

Check CPU and RAM specifications



```
!cat /proc/cpuinfo  
!cat /proc/meminfo
```

```
CPU: processor : 0  
      vendor_id : GenuineIntel  
      cpu family : 6  
      model : 63  
      model name : Intel(R) Xeon(R) CPU @ 2.30GHz  
      stepping : 0  
      microcode : 0x1  
      cpu MHz : 2299.998  
      cache size : 46080 KB  
      physical id : 0  
      siblings : 2  
      core id : 0  
      cpu cores : 1  
      apicid : 0  
      initial apicid : 0  
      fpu : yes  
      fpu_exception : yes  
      cpuid level : 13  
      wp : yes  
      flags : fpu vme de pse tsc msr pae mce cx8  
      bugs : cpu_meltdown spectre_v1 spectre_v2  
      bogomips : 4599.99  
      clflush size : 64  
      cache_alignment : 64  
      address sizes : 46 bits physical, 48 bits virtual  
      power management:  
  
processor : 1  
vendor_id : GenuineIntel  
cpu family : 6  
model : 63
```

MemTotal:	13333564 kB
MemFree:	10529740 kB
MemAvailable:	12432924 kB
Buffers:	85436 kB
Cached:	1958172 kB
SwapCached:	0 kB
Active:	1025112 kB
Inactive:	1462468 kB
Active(anon):	409852 kB
Inactive(anon):	360 kB
Active(file):	615260 kB
Inactive(file):	1462108 kB
Unevictable:	0 kB
Mlocked:	0 kB
SwapTotal:	0 kB
SwapFree:	0 kB
Dirty:	372 kB
Writeback:	0 kB
AnonPages:	443960 kB
Mapped:	245996 kB
Shmem:	1012 kB
SLAB:	170772 kB

Check GPU

```
!nvidia-smi
```

Thu Apr 29 13:36:22 2021

NVIDIA-SMI 465.19.01 Driver Version: 460.32.03 CUDA Version: 11.2							
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap	Memory-Usage	GPU-Util	Compute M.	MIG M.
0	Tesla P4	Off	00000000:00:04.0	Off			0
N/A	39C	P0	23W / 75W	199MiB / 7611MiB	0%	Default	N/A

Processes:

GPU	GI	CI	PID	Type	Process name	GPU Memory
ID						Usage

- <https://developer.nvidia.com/blog/nvidia-tesla-p4-gpus-available-now-on-the-google-cloud-platform/>

Tesla P4 Specs

- <https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/solutions/resources/documents1/Tesla-P4-Product-Brief.pdf>

Table 1. Product Specifications

Specification		Description
Product SKUs		PG414 SKU 200; NVPN: 699-2G414-0200-100 PG414 SKU 201; NVPN: 699-2G414-0201-100
Total board power		PG414 SKU 200: 75 Watts PG414 SKU 201: 50 Watts
GPU SKU		GP104-895-A1
PCI Device ID		0x1BB3
SSID		SKU 200: 0x11D8 SKU 201: 0x11E0
Board ID		0x10DE
NVIDIA® CUDA® cores		2560
GPU clocks	Base	885 MHz
	Maximum boost	1531 MHz (1113 MHz default)
	Idle	405 MHz
VBIOS	EEPROM size	4 Mbit
	UEFI	Supported
PCI Express interface		PCI Express 3.0 ×16 Lane and polarity reversal supported
Weight	Board	240 Grams
	Bracket with screws	9.7 Grams (low-profile bracket) 15.6 Grams (ATX bracket)

Table 2. Memory Specifications

Specification		Description
Memory clocks	Performance	2.8 GHz
	Idle	324 MHz
Memory size		8 GB
Memory bus width		256-bit
Memory configuration		8 pcs 256M × 32 GDDR5
Peak memory bandwidth		Up to 192 GBytes/s

Need more powerful GPUs on Google Colab?



Get more from Colab

UPGRADE NOW

\$9.99/month

Recurring billing • Cancel anytime

[Restrictions apply, learn more here](#)



Faster GPUs

Priority access to faster GPUs and TPUs means you spend less time waiting while code is running. [Learn more](#)



Longer runtimes

Longer running notebooks and fewer idle timeouts mean you disconnect less often. [Learn more](#)



More memory

More RAM and more disk means more room for your data. [Learn more](#)



What kinds of GPUs are available in Colab Pro?

With Colab Pro you get priority access to our fastest GPUs. For example, you may get access to T4 and P100 GPUs at times when non-subscribers get K80s. You also get priority access to TPUs. There are still usage limits in Colab Pro, though, and the types of GPUs and TPUs available in Colab Pro may vary over time.

In the free version of Colab there is very limited access to faster GPUs and to TPUs, and usage limits are much lower than they are in Colab Pro.

Google Colab TPUs

- <https://colab.research.google.com/notebooks/tpu.ipynb>

TPUs in Colab



In this example, we'll work through training a model to classify images of flowers on Google's lightning-fast Cloud TPUs. Our model will take as input a photo of a flower and return whether it is a daisy, dandelion, rose, sunflower, or tulip.

We use the Keras framework, new to TPUs in TF 2.1.0. Adapted from [this notebook](#) by [Martin Gorner](#).

Tensorflow with GPU

- <https://colab.research.google.com/notebooks/gpu.ipynb>
- Open <https://colab.research.google.com/notebooks/gpu.ipynb> copy to your Drive and move it to your working directory!
- First, you'll need to enable GPUs for the notebook:
 - Navigate to Edit→Notebook Settings
 - select GPU from the Hardware Accelerator drop-down
- Next, we'll confirm that we can connect to the GPU with tensorflow:



```
%tensorflow_version 2.x
import tensorflow as tf
device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))
```

Found GPU at: /device:GPU:0

Tensorflow with GPU

- Observe TensorFlow speedup on GPU relative to CPU

```
▶ %tensorflow_version 2.x
import tensorflow as tf
import timeit

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
    print(
        '\n\nThis error most likely means that this notebook is not '
        'configured to use a GPU. Change this in Notebook Settings via the '
        'command palette (cmd/ctrl-shift-P) or the Edit menu.\n\n')
    raise SystemError('GPU device not found')

def cpu():
    with tf.device('/cpu:0'):
        random_image_cpu = tf.random.normal((100, 100, 100, 3))
        net_cpu = tf.keras.layers.Conv2D(32, 7)(random_image_cpu)
        return tf.math.reduce_sum(net_cpu)

def gpu():
    with tf.device('/device:GPU:0'):
        random_image_gpu = tf.random.normal((100, 100, 100, 3))
        net_gpu = tf.keras.layers.Conv2D(32, 7)(random_image_gpu)
        return tf.math.reduce_sum(net_gpu)

# We run each op once to warm up; see: https://stackoverflow.com/a/45067900
cpu()
gpu()

# Run the op several times.
print('Time (s) to convolve 32x7x7x3 filter over random 100x100x100x3 images '
      '(batch x height x width x channel). Sum of ten runs.')
print('CPU (s):')
cpu_time = timeit.timeit('cpu()', number=10, setup="from __main__ import cpu")
print(cpu_time)
print('GPU (s):')
gpu_time = timeit.timeit('gpu()', number=10, setup="from __main__ import gpu")
print(gpu_time)
```

Time (s) to convolve 32x7x7x3 filter over random 100x100x100x3 images (batch x height x width x channel). Sum of ten runs.

CPU (s):
3.042170389000006

GPU (s):
0.04080971800001976

GPU speedup over CPU: 74x

Running CUDA in Google Colab

- CUDA comes already pre-installed in Google Colab.

```
!nvidia-smi
```

Thu Apr 29 14:38:42 2021

NVIDIA-SMI 465.19.01 Driver Version: 460.32.03 CUDA Version: 11.2							
GPU	Name	Persistence-M	Bus-Id	Disp.A	Volatile	Uncorr.	ECC
Fan	Temp	Perf	Pwr:Usage/Cap		Memory-Usage	GPU-Util	Compute M.
							MIG M.
0	Tesla K80		Off	00000000:00:04.0	Off		0
N/A	73C	P8	32W / 149W	0MiB / 11441MiB	0%	Default	N/A

Processes:

GPU	GI	CI	PID	Type	Process name	GPU Memory Usage
ID	ID					
No running processes found						

Running CUDA in Google Colab

- CUDA comes already pre-installed in Google Colab.

```
[2] !nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2020 NVIDIA Corporation
Built on Wed_Jul_22_19:09:09_PDT_2020
Cuda compilation tools, release 11.0, v11.0.221
Build cuda_11.0_bu.TC445_37.28845127_0
```

- As it can be seen, the installation regards CUDA 11.0.

Running CUDA in Google Colab

- To enable CUDA **programming** and **execution** directly under Google Colab, you can install the [nvcc4jupyter](#) plugin as

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git
```

```
!pip install git+git://github.com/andreinechaev/nvcc4jupyter.git

Collecting git+git://github.com/andreinechaev/nvcc4jupyter.git
  Cloning git://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-ugiskyks
    Running command git clone -q git://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-
Building wheels for collected packages: NVCCPlugin
  Building wheel for NVCCPlugin (setup.py) ... done
  Created wheel for NVCCPlugin: filename=NVCCPlugin-0.0.2-cp37-none-any.whl size=4307 sha256
  Stored in directory: /tmp/pip-ephem-wheel-cache-lrani5l9/wheels/10/c2/05/ca241da37bff77d60
Successfully built NVCCPlugin
Installing collected packages: NVCCPlugin
Successfully installed NVCCPlugin-0.0.2
```

Running CUDA in Google Colab

- After that, you should load the plugin as

```
%load_ext nvcc_plugin
```

```
▶ %load_ext nvcc_plugin  
created output directory at /content/src  
Out bin /content/result.out
```

Now it is ready to run CUDA code!!

Running CUDA in Google Colab

- Write and run CUDA code by adding below prefix

```
%%cu
```

- Compilation and execution occurs when pressing the play button to run a code cell.
- Test below Hello World again!

The screenshot shows a Google Colab interface. On the left, there is a code cell containing the following CUDA code:

```
%%cu
#include <iostream>
int main()
{
    std::cout << "Hello World CUDA on Google Colab GPU!\n";
    return 0;
}
```

To the left of the code cell is a play button icon. Below the code cell, the output is displayed in a text area:

```
Hello World CUDA on Google Colab GPU!
```

Running CUDA in Google Colab

- Run the code that I provided in the last lecture

```
%%cu
#include <iostream>
#include <cstdlib>

#define N (4*8)
#define THREADS_PER_BLOCK 8

using namespace std;

__global__ void add(int *a, int *b, int *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main()
{
    int a[N], b[N], c[N];      // host copies of a, b, c
    int *d_a, *d_b, *d_c;      // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **)&d_a, N*size);
    cudaMalloc((void **)&d_b, N*size);
    cudaMalloc((void **)&d_c, N*size);

    // Setup input values
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // Copy inputs to device
    cudaMemcpy(d_a, &a, N*size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, N*size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
    //add<<<4,8>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, N*size, cudaMemcpyDeviceToHost);

    // Display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free the memory allocated on the GPU
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

Running CUDA in Google Colab

● Results! ??

```
↳ 0 + 0 = 0  
-1 + 1 = 0  
-2 + 4 = 0  
-3 + 9 = 0  
-4 + 16 = 0  
-5 + 25 = 0  
-6 + 36 = 0  
-7 + 49 = 0  
-8 + 64 = 0  
-9 + 81 = 0  
-10 + 100 = 0  
-11 + 121 = 0  
-12 + 144 = 0  
-13 + 169 = 0  
-14 + 196 = 0  
-15 + 225 = 0  
-16 + 256 = 0  
-17 + 289 = 0  
-18 + 324 = 0  
-19 + 361 = 0  
-20 + 400 = 0  
-21 + 441 = 0  
-22 + 484 = 0  
-23 + 529 = 0  
-24 + 576 = 0  
-25 + 625 = 0  
-26 + 676 = 0  
-27 + 729 = 0  
-28 + 784 = 0  
-29 + 841 = 0  
-30 + 900 = 0  
-31 + 961 = 0
```

```
[ahnt@gpu1:CUDA]$ ./vector_sum_bt  
0 + 0 = 0  
-1 + 1 = 0  
-2 + 4 = 2  
-3 + 9 = 6  
-4 + 16 = 12  
-5 + 25 = 20  
-6 + 36 = 30  
-7 + 49 = 42  
-8 + 64 = 56  
-9 + 81 = 72  
-10 + 100 = 90  
-11 + 121 = 110  
-12 + 144 = 132  
-13 + 169 = 156  
-14 + 196 = 182  
-15 + 225 = 210  
-16 + 256 = 240  
-17 + 289 = 272  
-18 + 324 = 306  
-19 + 361 = 342  
-20 + 400 = 380  
-21 + 441 = 420  
-22 + 484 = 462  
-23 + 529 = 506  
-24 + 576 = 552  
-25 + 625 = 600  
-26 + 676 = 650  
-27 + 729 = 702  
-28 + 784 = 756  
-29 + 841 = 812  
-30 + 900 = 870  
-31 + 961 = 930
```

Running CUDA in Google Colab

- Notice that Google Colab currently provides the newer **T4** or **P100** GPU or the older **K80** if other GPUs are not available. CUDA 11 is not compatible with those!
- Let's change to CUDA 10.1

```
!rm -rf cuda
!ln -s /usr/local/cuda-10.1 /usr/local/cuda
!stat cuda
```

File: cuda => /usr/local/cuda-10.1
Size: 20 Blocks: 0 IO Block: 4096 symbolic link
Device: 32h/50d Inode: 4207961 Links: 1
Access: (0777/lrwxrwxrwx)Uid: (0/ root) Gid: (0/ root)
Access: 2021-04-29 15:07:24.122422198 +0000
Modify: 2021-04-29 15:07:24.021422297 +0000
Change: 2021-04-29 15:07:24.021422297 +0000
Birth: -

Running CUDA in Google Colab

- Run again! It works!!

```
↳ 0 + 0 = 0  
-1 + 1 = 0  
-2 + 4 = 2  
-3 + 9 = 6  
-4 + 16 = 12  
-5 + 25 = 20  
-6 + 36 = 30  
-7 + 49 = 42  
-8 + 64 = 56  
-9 + 81 = 72  
-10 + 100 = 90  
-11 + 121 = 110  
-12 + 144 = 132  
-13 + 169 = 156  
-14 + 196 = 182  
-15 + 225 = 210  
-16 + 256 = 240  
-17 + 289 = 272  
-18 + 324 = 306  
-19 + 361 = 342  
-20 + 400 = 380  
-21 + 441 = 420  
-22 + 484 = 462  
-23 + 529 = 506  
-24 + 576 = 552  
-25 + 625 = 600  
-26 + 676 = 650  
-27 + 729 = 702  
-28 + 784 = 756  
-29 + 841 = 812  
-30 + 900 = 870  
-31 + 961 = 930
```

Lab Assignment

- Run below CUDA code in Google Colab and get the correct results

```
%%cu
#include <iostream>
#include <cstdlib>

#define N (4*8)
#define THREADS_PER_BLOCK 8

using namespace std;

__global__ void add(int *a, int *b, int *c)
{
    int tid = threadIdx.x + blockIdx.x * blockDim.x;
    if (tid < N)
        c[tid] = a[tid] + b[tid];
}

int main()
{
    int a[N], b[N], c[N];      // host copies of a, b, c
    int *d_a, *d_b, *d_c;      // device copies of a, b, c
    int size = sizeof(int);

    // Allocate space for device copies of a, b, c
    cudaMalloc((void **) &d_a, N * size);
    cudaMalloc((void **) &d_b, N * size);
    cudaMalloc((void **) &d_c, N * size);

    // Setup input values
    for (int i=0; i<N; i++) {
        a[i] = -i;
        b[i] = i * i;
    }

    // Copy inputs to device
    cudaMemcpy(d_a, &a, N * size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_b, &b, N * size, cudaMemcpyHostToDevice);

    // Launch add() kernel on GPU
    add<<<N/THREADS_PER_BLOCK,THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
    //add<<<4,8>>>(d_a, d_b, d_c);

    // Copy result back to host
    cudaMemcpy(&c, d_c, N * size, cudaMemcpyDeviceToHost);

    // Display the results
    for (int i=0; i<N; i++) {
        printf("%d + %d = %d\n", a[i], b[i], c[i]);
    }

    // Free the memory allocated on the GPU
    cudaFree(d_a);
    cudaFree(d_b);
    cudaFree(d_c);

    return 0;
}
```

```
0 + 0 = 0
-1 + 1 = 0
-2 + 4 = 2
-3 + 9 = 6
-4 + 16 = 12
-5 + 25 = 20
-6 + 36 = 30
-7 + 49 = 42
-8 + 64 = 56
-9 + 81 = 72
-10 + 100 = 90
-11 + 121 = 110
-12 + 144 = 132
-13 + 169 = 156
-14 + 196 = 182
-15 + 225 = 210
-16 + 256 = 240
-17 + 289 = 272
-18 + 324 = 306
-19 + 361 = 342
-20 + 400 = 380
-21 + 441 = 420
-22 + 484 = 462
-23 + 529 = 506
-24 + 576 = 552
-25 + 625 = 600
-26 + 676 = 650
-27 + 729 = 702
-28 + 784 = 756
-29 + 841 = 812
-30 + 900 = 870
-31 + 961 = 930
```