3

ML with GPUs

Topics

- Introduction to GPUs
- CUDA
- GPU Ecosystem
- GPU-based ML libraries: cuDF, cuML, and cuPY
- Summary

The Importance of GPU in ML Computations

- GPU
  - A Graphics Processing Unit (GPU) is a specialized circuit engineered for fast processing and image generation for screen display
- History of GPU Evolution
  - Primarily crafted for graphic rendering in gaming
  - Evolved into powerhouses for parallel processing
  - Integral for complex, computation-heavy tasks in machine learning and scientific domains.GPU's Transition to General-Purpose Computing
- GPU's Transition to General-Purpose Computing
  - Transitioned beyond graphics to multifaceted computing applications
  - Birth of GPGPU, diversifying GPUs' roles to encompass extensive computational functions.

The Role of GPU in Modern ML

GPUs in ML: From Graphics to Machine Learning Powerhouses

- GPUs have transitioned from graphic rendering to key roles in machine learning (ML) and artificial intelligence (AI) innovation.
- Their robust computing capabilities address the demands of intricate ML algorithms, particularly in deep learning.

Boosting Deep Learning Performance

cuDNN

- The extensive computation required for training and running deep neural networks in deep learning is efficiently managed by GPUs.
- By parallelizing data processing, GPUs have significantly shortened the time to train intricate models.

Enabling More Complex and Efficient Models

- GPUs have unlocked the potential for larger and more complex models, transforming what's possible in ML.
- This advancement has propelled significant innovations in sectors such as natural language processing, computer vision, and predictive analytics.

GPU vs. Traditional Computing

Features	GPU	CPU
Architecture Comparison	Designed for parallel processing, GPUs consist of hundreds of cores capable of handling thousands of threads simultaneously	Optimized for sequential processing, CPUs have fewer cores designed for a wide range of tasks, focusing on single-thread performance.
Processing Capabilities	Excelling in tasks that can be parallelized, making them ideal for matrix operations and data-heavy computations common in ML	Best suited for tasks that require sequential processing and complex decision-making.
Application in ML	Revolutionized the field of deep learning by significantly speeding up the training of large neural networks. Ideal for tasks like image and video processing, neural network training, and large-scale data analyses	More suitable for traditional ML algorithms that don't require intense parallel processing, like decision trees or smaller-scale data processing tasks

CPU - GPU

- Smp – (shared Memory
- Dmp.

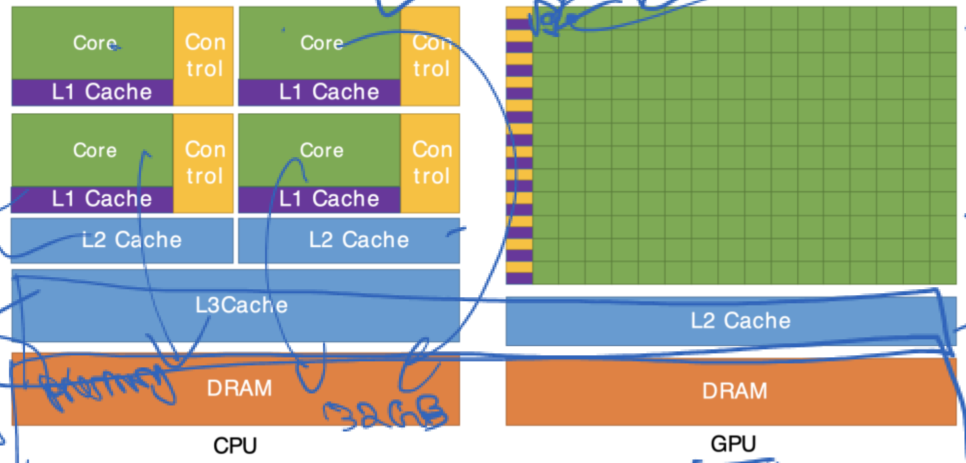
- CPU is designed to excel at executing a sequence of operations (threads) as fast as possible (low latency)
- CPU can execute a few tens of these threads in parallel
- GPU is designed to excel at executing thousands of threads in parallel (amortizing the slower single-thread performance to achieve greater throughput)

thread.

thread 1

1-Core

4-Core



# of threads ∝ 1K

– 32GB RAM

AMD M1808

Primary

Benefits of GPUs in Machine Learning

Enhanced Parallel Processing

- GPUs excel in parallel processing, capable of handling thousands of tasks at once.
- Essential for machine learning, particularly deep learning, where concurrent operations are the norm.

Speedier Data Handling

- Data processing tasks, like matrix operations crucial to ML, are expedited by GPUs.
- Results in rapid ML model training, facilitating more trials and optimizations in shorter timeframes.

Large Dataset Management

- Optimized for large dataset manipulation, pivotal in big data and AI initiatives.
- Superior bandwidth and memory features enable faster data handling, beneficial for extensive ML operations.

Superior Energy Efficiency

- For tasks that can be parallelized, GPUs offer better energy efficiency than CPUs.
- Their power-conscious performance is preferable for sustained, intensive ML

GPU

- **CUDA (Compute Unified Device Architecture)**
  - general-purpose parallel computing platform and programming model for NVIDIA GPUs
  - allows developers to ~~use~~ C++ as a high-level programming language

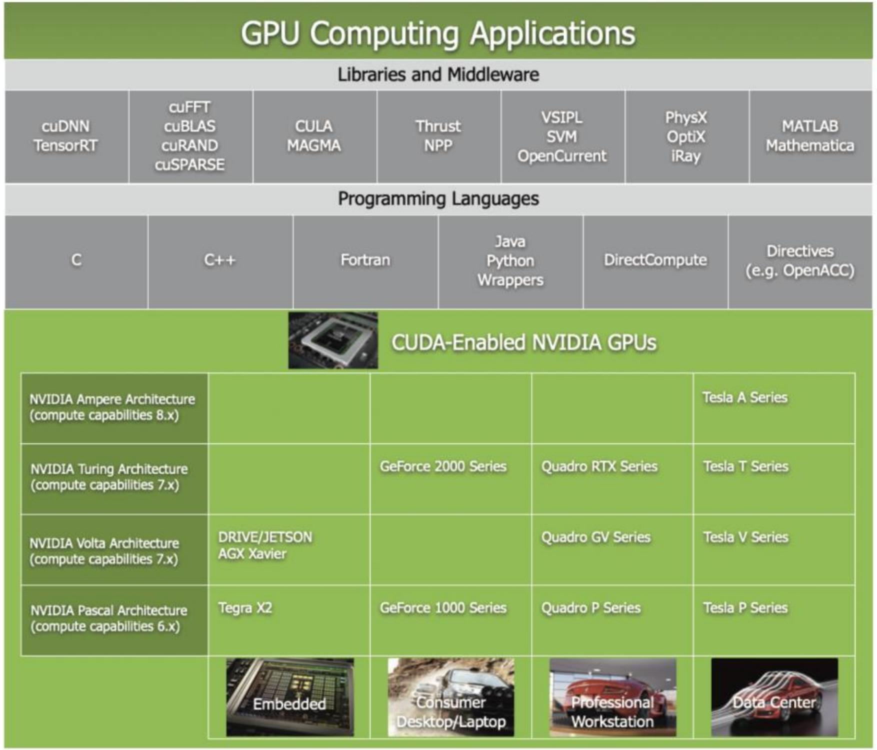
- **OpenCL**
  - general heterogenous computing framework
  - https://www.khronos.org/opencl/

- **OpenACC**
  - user-driven directive-based performance-portable parallel programming model
  - supports C, C++, Fortran programming language

Note: For python usage: Check Theano, pyCUDA

CUDA

A general-purpose parallel computing platform and programming model for NVIDIA GPUs



GPU Computing Applications

Libraries and Middleware

cuDNN TensorRT	cuFFT cuBLAS cuRAND cuSPARSE	CULA MAGMA	Thrust NPP	VSIPL SVM OpenCurrent	PhysX OptiX iRay	MATLAB Mathematica

Programming Languages

C	C++	Fortran	Java Python Wrappers	DirectCompute	Directives (e.g. OpenACC)

CUDA-Enabled NVIDIA GPUs

NVIDIA Ampere Architecture (compute capabilities 8.x)				Tesla A Series
NVIDIA Turing Architecture (compute capabilities 7.x)		GeForce 2000 Series	Quadro RTX Series	Tesla T Series
NVIDIA Volta Architecture (compute capabilities 7.x)	DRIVE/JETSON AGX Xavier		Quadro GV Series	Tesla V Series
NVIDIA Pascal Architecture (compute capabilities 6.x)	Tegra X2	GeForce 1000 Series	Quadro P Series	Tesla P Series
	Embedded	Consumer Desktop/Laptop	Professional Workstation	Data Center

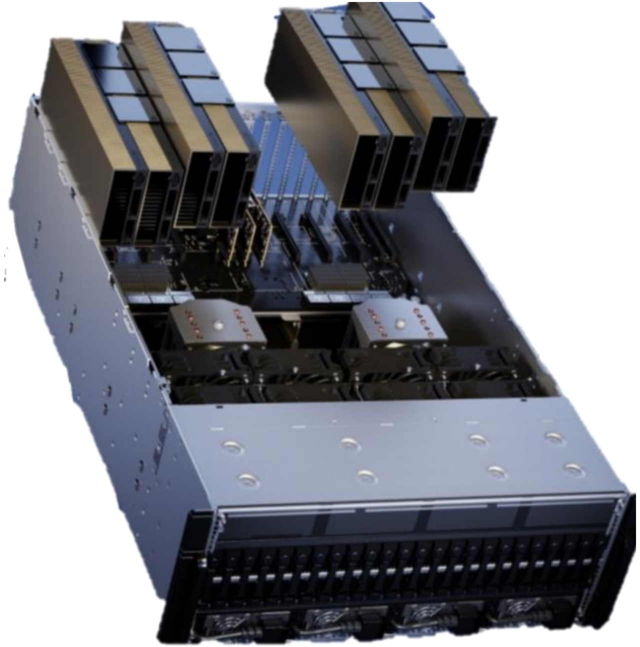
Scalable Programming Model

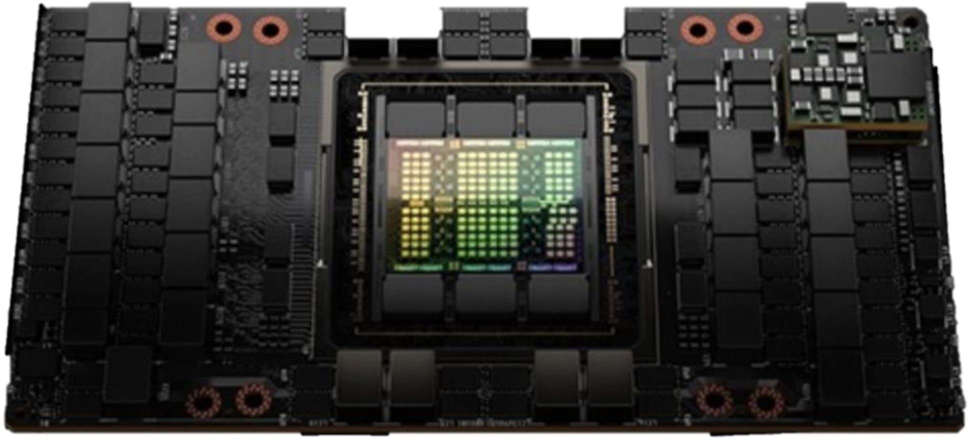
- GPU computing: steps involved
  - Setup inputs on the host (CPU-accessible memory)
  - Allocate memory for outputs on the host CPU
  - Allocate memory for inputs on the GPU device
  - Allocate memory for outputs on the GPU
  - Copy inputs from host to GPU (slow)
  - Start GPU kernel (function that executes on gpu)
  - Copy output from GPU to host (slow)

Starts @ 11 am

Note: Unified Memory Architecture in CUDA is a feature that simplifies memory management between the CPU (host) and GPU (device). It provides a single memory space accessible by both the host and the device, allowing them to share data without the need to explicitly transfer it between the host and the device memory.

GPU: Thread Hierarchy





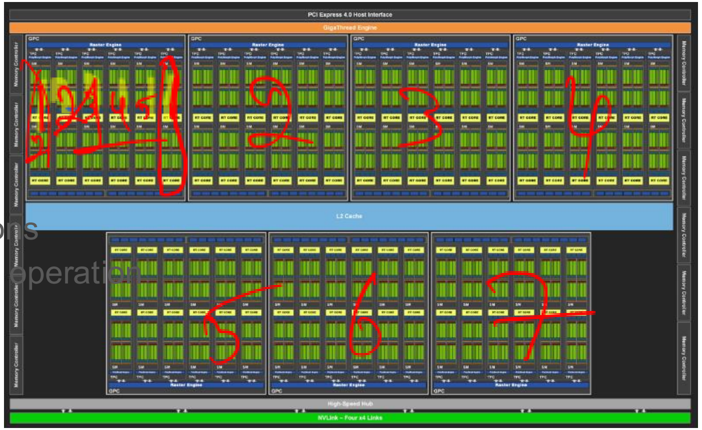
NVIDIA H100 Tensor Core GPU

https://www.nvidia.com/en-in/data-center/h100/

GPU: Thread Hierarchy

NVIDIA RTX A6000: Ampere GPU Architecture In-Depth

- 7 GPCs, each with
  - 6 TPCs, each with 2 SMs
- In Total
  - 42 TPCs & 84 SMs
- Each SM with
  - 128 CUDA (FP32) cores/SM (10752 in total)
    - Performs FP32, FP16, INT8, and INT4 precision operations
    - 168 FP64, TFLOP rate is 1/64th the TFLOP rate of FP32 operation
  - 4 Tensor cores/SM (336 in total, 3rd Gen)
  - 336 Texture units (Graphics)
  - 84 RT cores (Graphics)
  - 38.7 TFLOps (non-tensor) Peak (FP32, FP16, BF16)
  - FP32: 77.4/154.8 TOPS/TFLOPS Tensor TFLOPS Peak
  - INT8: 309.7/619.4 TOPS/TFLOPS Tensor TOPS Peak
  - INT4: 619.4/1238.6 TOPS/TFLOPS Tensor TOPS Peak

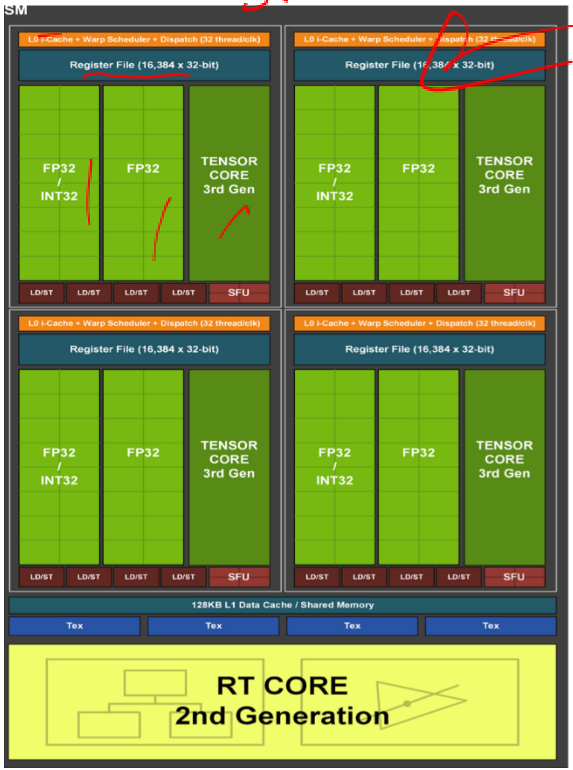


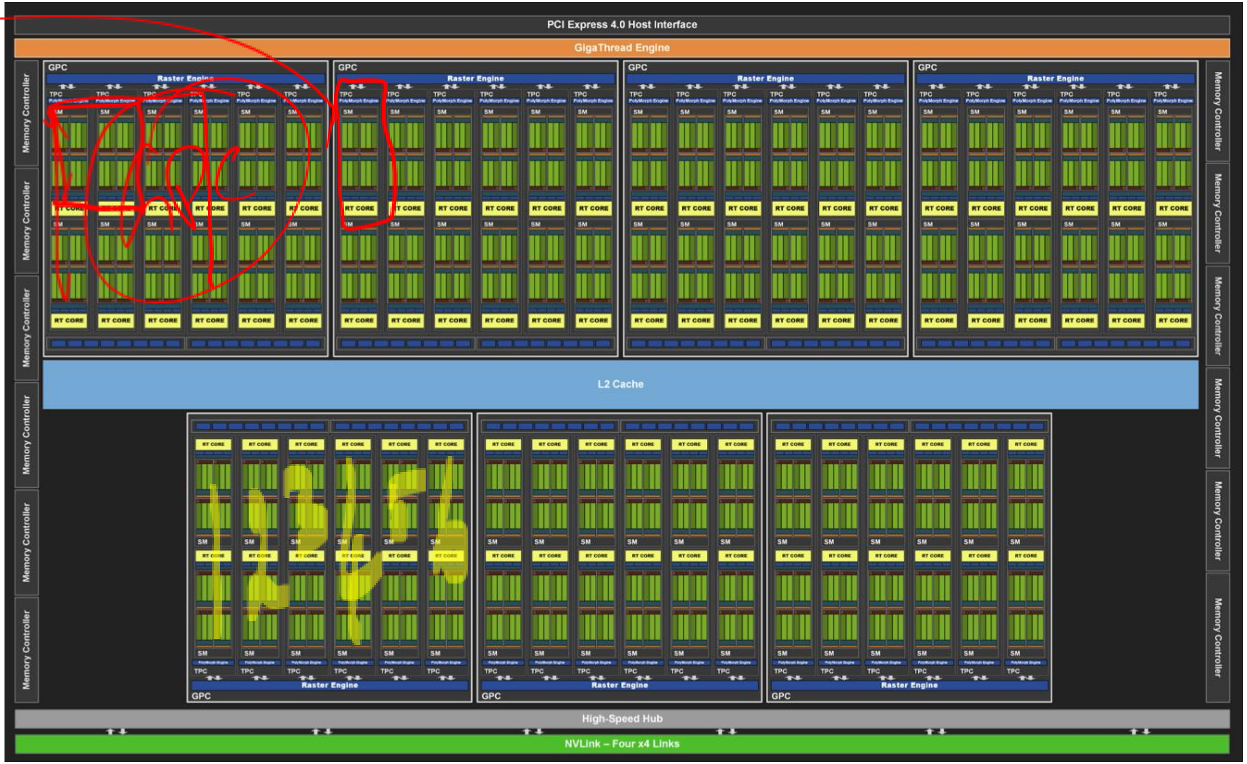
GPC - Graphics Processing Clusters, TPCs -Texture Processing Clusters, SM - Streaming Multiprocessors, Raster Operators (ROPS), and memory controllers

GPU: Thread Hierarchy

Sm

NVIDIA RTX A6000: Ampere GPU Architecture In-Depth


SM
L0 i-Cache + Warp Scheduler + Dispatch (32 thread/clk)
Register File (16,384 x 32-bit)
FP32 / INT32
FP32
TENSOR CORE 3rd Gen
LD/ST
SFU
128KB L1 Data Cache / Shared Memory
Tex
RT CORE
2nd Generation



https://www.nvidia.com/content/PDF/nvidia-ampere-ga-102-gpu-architecture-whitepaper-v2.pdf

GPU: Thread Hierarchy

NVIDIA RTX A6000: Ampere GPU Architecture In-Depth

- CUDA Cores
  - 128 CUDA cores/SM (10752 in total)
  - Mixed-precision tensor operations: The cores support efficient tensor operations for AI/ML workloads, including FP32, FP16, INT8, and INT4 precisions.

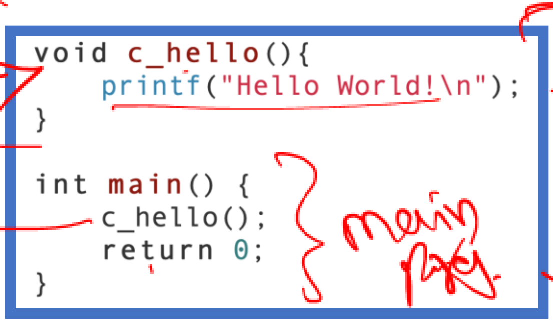
GPC - Graphics Processing Clusters, TPCs -Texture Processing Clusters, SM - Streaming Multiprocessors, Raster Operators (ROPS), and memory controllers

Questions

1. GPUs can be classified as shared memory machines. (T/F) T
2. GPU is designed to excel at executing thousands of threads in parallel to achieve low latency (as fast as possible). (T/F) F

CPU

GPU: Hello World



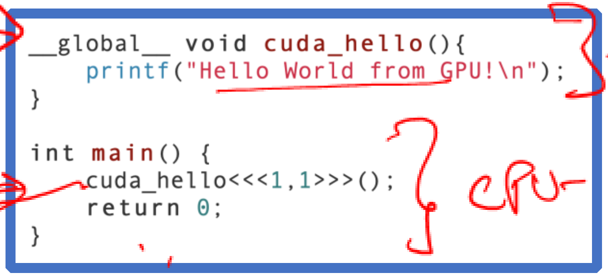
```
void c_hello(){
    printf("Hello World!\n");
}

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

fn.

CPU

CPU code

main prog.



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

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

Cuda kernel.

GPU

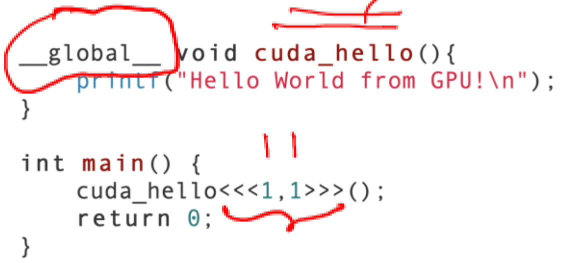
CPU code

```
$> nvcc hello.cu -o hello
```

https://cuda-tutorial.readthedocs.io/en/latest/tutorials/tutorial01/

GPU: Hello World

Cuda kernel.

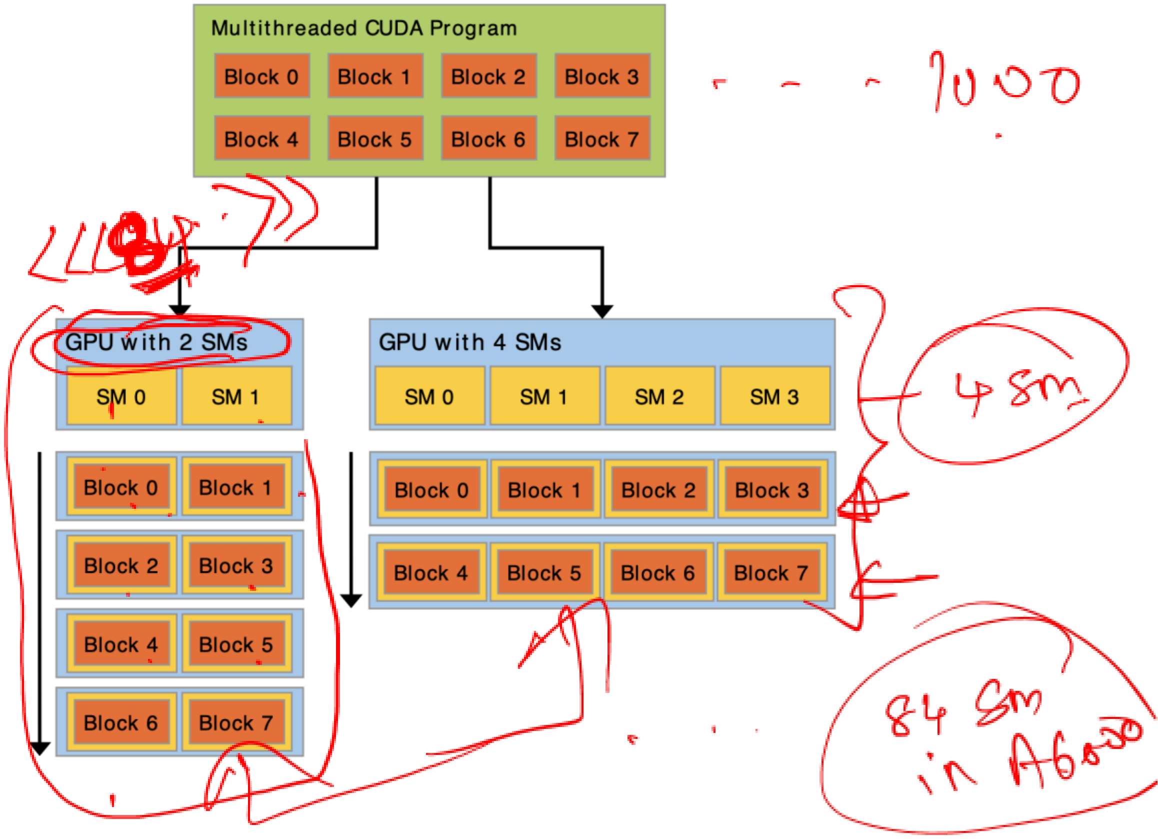


```
__global__ void cuda_hello(){
    printf("Hello World from GPU!\n");
}

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

- __global__ specifier indicates a function that runs on device (GPU)
- the CUDA kernel cuda_hello() can be called from host [CPU]
- kernel execution configuration is provided through <<<…>>> syntax, called kernel launch
- the number of GPU threads “M" to be launched in each thread block is indicated through kernel launch: <<<B,M>>>, where “B” is the number of thread blocks

GPU Computing

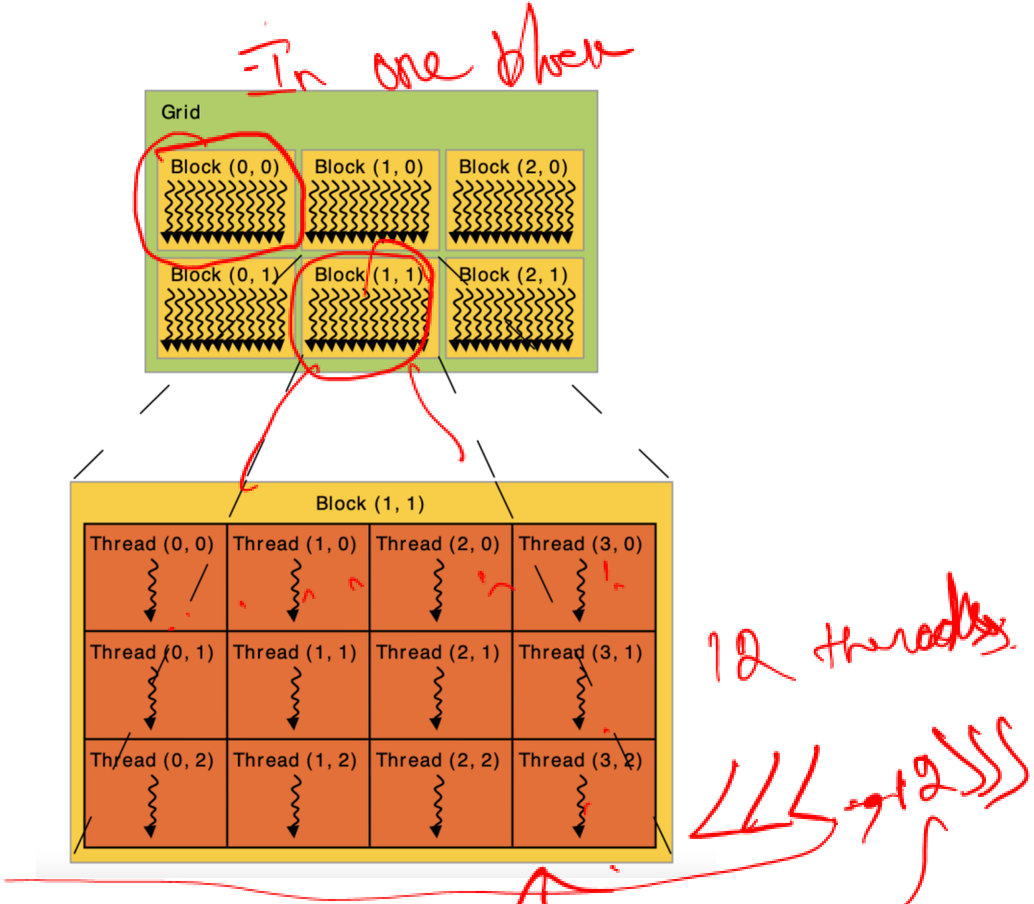
- Thread - distributed by the CUDA runtime (threadIdx)
- Block - user defined group of 1 to ~T threads (blockIdx)
- Grid - a group of one or more blocks. A grid is created for each CUDA kernel function call.



999 1000

GPU Computing

- Thread - distributed by the CUDA runtime (threadIdx)
- Block - user defined group of 1 to ~512 threads (blockIdx)
- Grid - a group of one or more blocks. A grid is created for each CUDA kernel function call.



SM

GPU Computing

<<<4, 8>>> → 32

for(i = 1 to 32){

c[i] = a[i]+b[i]

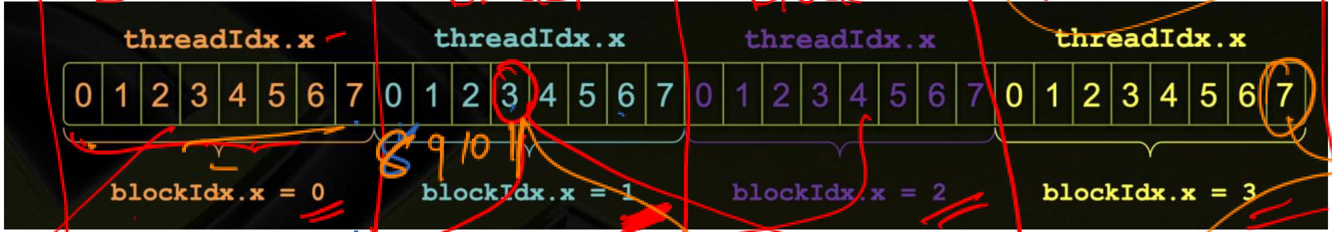
}

each "i" will be assigned to a unique thread

Indexing Arrays with Blocks and Threads

- consider indexing an array with one element per thread (8 threads/block)

Block 0 Block 1 Block 2 Block 3



threadIdx.x								threadIdx.x								threadIdx.x								threadIdx.x							
0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7
blockIdx.x = 0								blockIdx.x = 1								blockIdx.x = 2								blockIdx.x = 3							

31

c[31] = a[31]+b[31]

local thread index

- with M threads per block, a unique index for each thread is given by:
  - index = M * blockIdx.x + threadIdx.x
- use the built-in variable blockDim.x for threads per block (M)

= i [Global index]

c[i] = a[i]+b[i]

i = index = 8*1+3

= 11

GPU Computing

Indexing Arrays with Blocks and Threads

- blockIdx.x, blockIdx.y, blockIdx.z built-in variables return the block ID in the x-axis, y-axis, and z-axis of the block
- threadIdx.x, threadIdx.y, threadIdx.z built-in variables return the thread ID in the x-axis, y-axis, and z-axis of the thread in a particular block
- blockDim.x, blockDim.y, blockDim.z built-in variables return the “block dimension” (number of threads in a block in the x-axis, y-axis, and z-axis)

GPU Computing: Vector addition

```
#define N 1618
#define T 1024

// Device code
__global__ void VecAdd(int* A, int* B, int* C, int N)
{
 int i = blockDim.x * blockIdx.x + threadIdx.x;
    if (i < N)
    C[i] = A[i] + B[i];

}
```

```
// main code
int main() {
 int a[N], b[N], c[N];
 int *dev_a, *dev_b, *dev_c;
 // initialize a and b with int values

 size = N * sizeof(int);
 cudaMalloc((void**)&dev_a, size);
 cudaMalloc((void**)&dev_b, size);
 cudaMalloc((void**)&dev_c, size);

 cudaMemcpy(dev_a, a, size,cudaMemcpyHostToDevice);
 cudaMemcpy(dev_b, b, size,cudaMemcpyHostToDevice);

 vecAdd<<<(int)ceil(N/T),T>>>(dev_a,dev_b,dev_c);

 cudaMemcpy(c, dev_c, size,cudaMemcpyDeviceToHost);

 cudaFree(dev_a);  cudaFree(dev_b);  cudaFree(dev_c);


exit (0); }
```

1024 * 1 + 1023

Ceil $\left(\frac{N}{T} = \frac{1618}{1024}\right)$

2

Global index

threads : 2048

A[1618]

range of 'i' [0 - 2047]

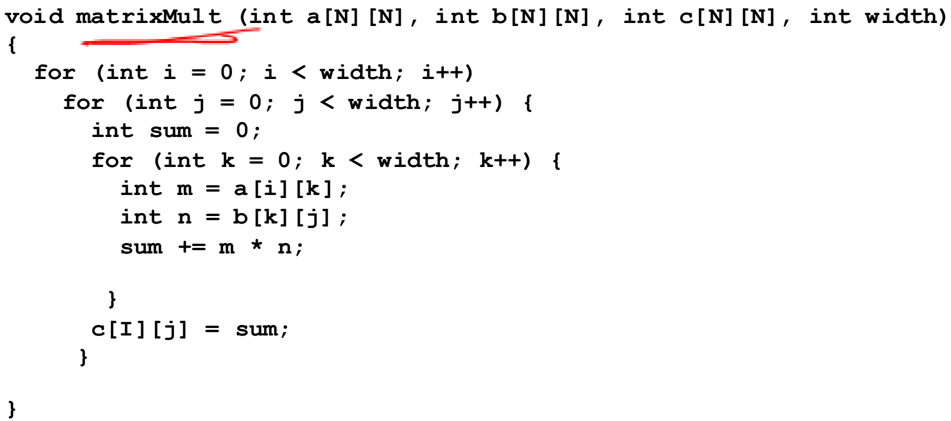
Kernel launch

<<<2, 1024>>>

GPU

additional overhead

GPU Computing: Matrix Multiplication



```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++)
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;

       }
      c[I][j] = sum;
     }

}
```

Can it be parallelized?

GPU Computing: Matrix Multiplication

n^2 – dot Product

```
void matrixMult (int a[N][N], int b[N][N], int c[N][N], int width)
{
  for (int i = 0; i < width; i++)
    for (int j = 0; j < width; j++) {
      int sum = 0;
      for (int k = 0; k < width; k++) {
        int m = a[i][k];
        int n = b[k][j];
        sum += m * n;

       }
      c[I][j] = sum;
     }

}
// Device code
__global__ void matrixMult (int *a, int *b, int *c, int width) {
 int k, sum = 0;
 int col = threadIdx.x + blockDim.x * blockIdx.x;
 int row = threadIdx.y + blockDim.y * blockIdx.y;
 if(col < width && row < width) {
  for (k = 0; k < width; k++)
   sum += a[row * width + k] * b[k * width + col];
   c[row * width + col] = sum;
 }

}
```

launch n^2 threads

each thread Performs one dot Prod.

$c[i,j]$

Global index

CU Block

each C_{ij} is mapped to a thread.

Asynchronous Concurrent Execution

CUDA exposes the following operations as independent tasks that can operate concurrently with one another:

- Computation on the host;
- Computation on the device;
- Memory transfers from the host to the device;
- Memory transfers from the device to the host;
- Memory transfers within the memory of a given device;
- Memory transfers among devices.

More Details: https://docs.nvidia.com/cuda/cuda-c-programming-guide/

low Pre

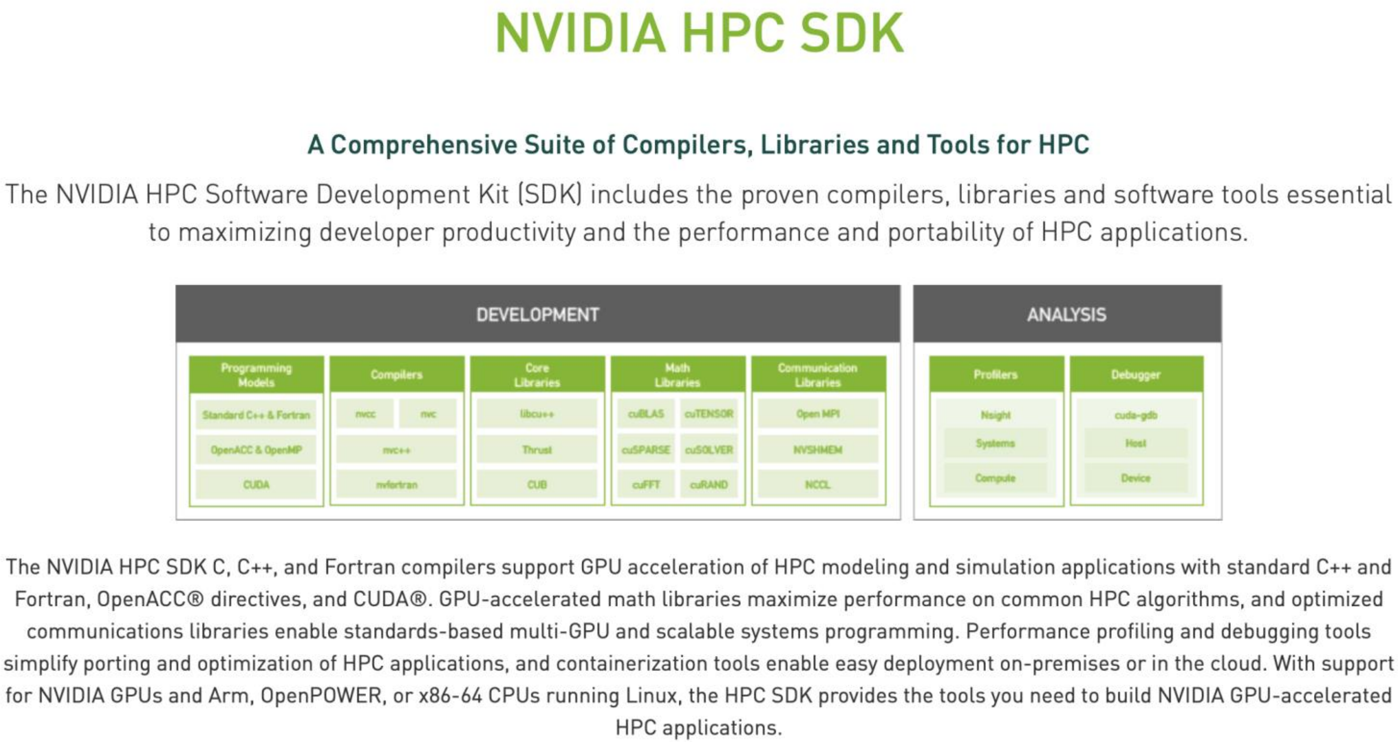
4 - 62

GPU Computing

EX: Write CUDA parallel code for

1. Matrix addition
2. Matrix vector multiplication
3. Matrix multiplication

GPU Computing



NVIDIA HPC SDK

A Comprehensive Suite of Compilers, Libraries and Tools for HPC

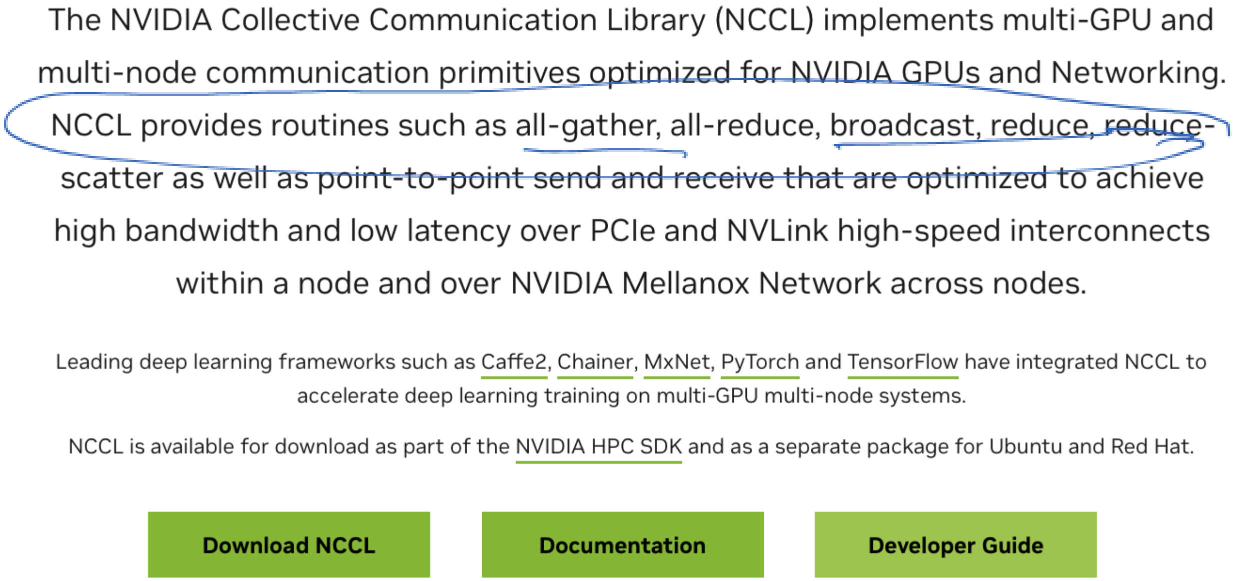
The NVIDIA HPC Software Development Kit (SDK) includes the proven compilers, libraries and software tools essential to maximizing developer productivity and the performance and portability of HPC applications.

DEVELOPMENT					ANALYSIS	
Programming Models	Compilers	Core Libraries	Math Libraries	Communication Libraries	Profilers	Debugger
Standard C++ & Fortran	nvcc nvc	libcu++	cuBLAS cuTENSOR	Open MPI	Nsight	cuda-gdb
OpenACC & OpenMP	nvc++	Thrust	cuSPARSE cuSOLVER	NVSHMEM	Systems	Host
CUDA	nvfortran	CUB	cuFFT cuRAND	NCCL	Compute	Device

The NVIDIA HPC SDK C, C++, and Fortran compilers support GPU acceleration of HPC modeling and simulation applications with standard C++ and Fortran, OpenACC® directives, and CUDA®. GPU-accelerated math libraries maximize performance on common HPC algorithms, and optimized communications libraries enable standards-based multi-GPU and scalable systems programming. Performance profiling and debugging tools simplify porting and optimization of HPC applications, and containerization tools enable easy deployment on-premises or in the cloud. With support for NVIDIA GPUs and Arm, OpenPOWER, or x86-64 CPUs running Linux, the HPC SDK provides the tools you need to build NVIDIA GPU-accelerated HPC applications.

https://developer.nvidia.com/hpc-sdk

GPU Computing for Deep Learning



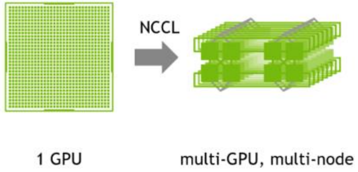
The NVIDIA Collective Communication Library (NCCL) implements multi-GPU and multi-node communication primitives optimized for NVIDIA GPUs and Networking. NCCL provides routines such as all-gather, all-reduce, broadcast, reduce, reduce-scatter as well as point-to-point send and receive that are optimized to achieve high bandwidth and low latency over PCIe and NVLink high-speed interconnects within a node and over NVIDIA Mellanox Network across nodes.

Leading deep learning frameworks such as Caffe2, Chainer, MxNet, PyTorch and TensorFlow have integrated NCCL to accelerate deep learning training on multi-GPU multi-node systems.

NCCL is available for download as part of the NVIDIA HPC SDK and as a separate package for Ubuntu and Red Hat.

GitHub   Watch GTC Webinar

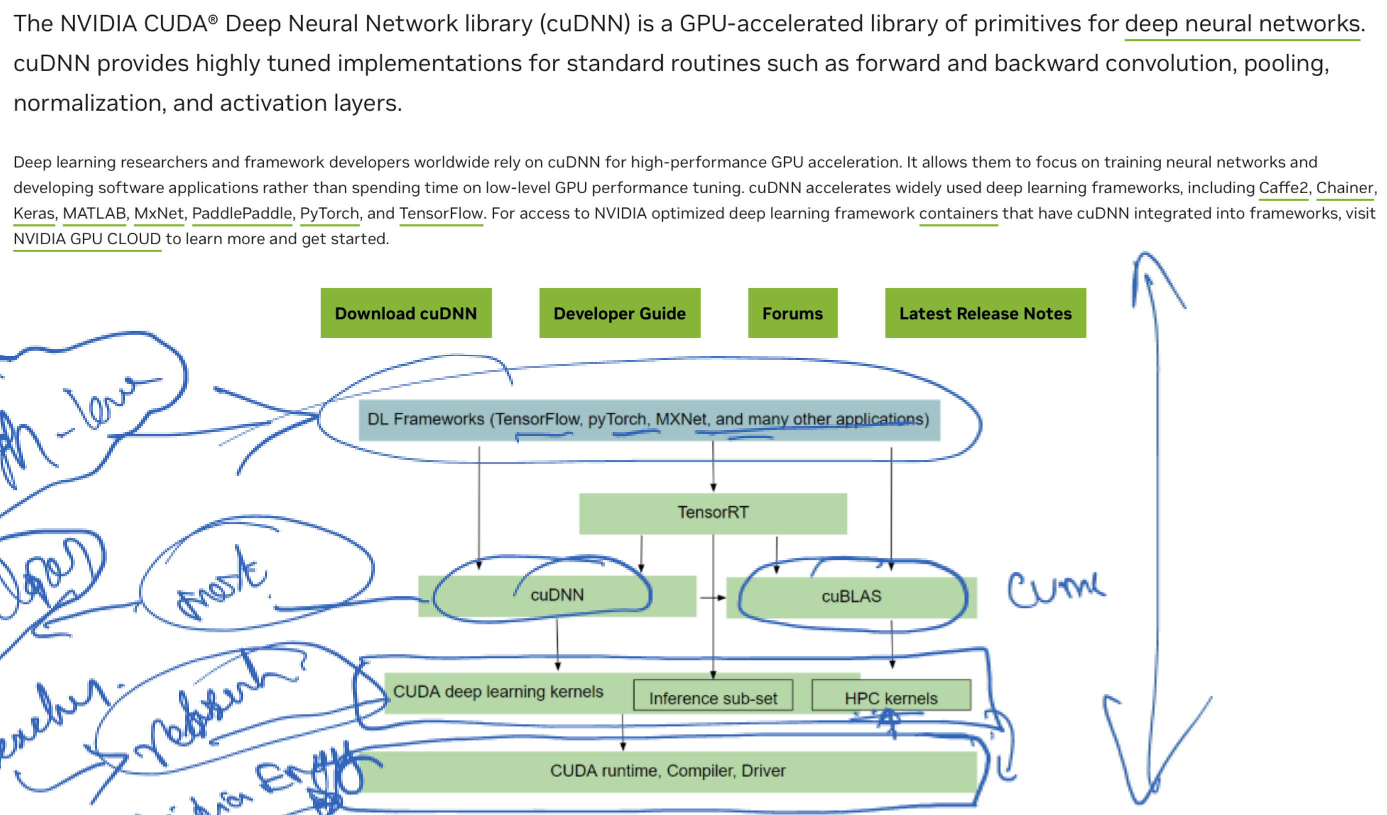
GPU1
GPU2
GPU3
GPU3



https://developer.nvidia.com/nccl

GPU Computing (cuDNN) in Data Sceine

Good to know

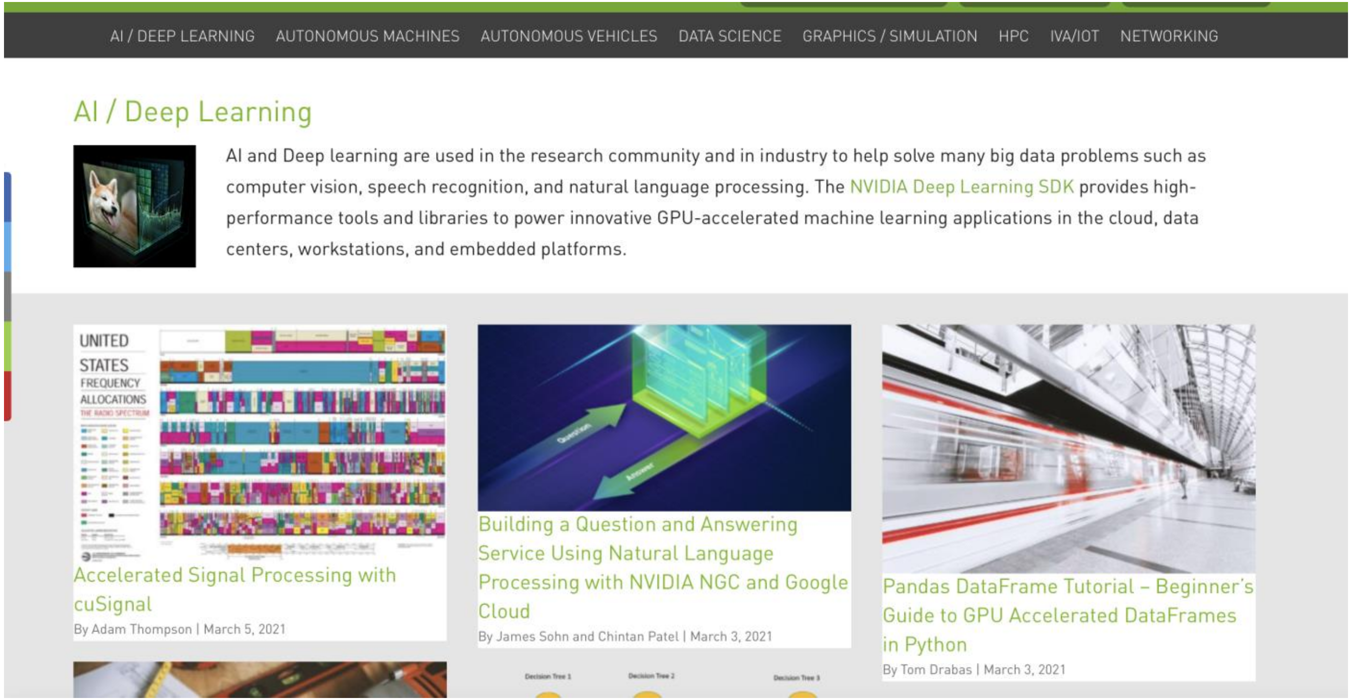


The NVIDIA CUDA® Deep Neural Network library (cuDNN) is a GPU-accelerated library of primitives for deep neural networks. cuDNN provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.

Deep learning researchers and framework developers worldwide rely on cuDNN for high-performance GPU acceleration. It allows them to focus on training neural networks and developing software applications rather than spending time on low-level GPU performance tuning. cuDNN accelerates widely used deep learning frameworks, including Caffe2, Chainer, Keras, MATLAB, MxNet, PaddlePaddle, PyTorch, and TensorFlow. For access to NVIDIA optimized deep learning framework containers that have cuDNN integrated into frameworks, visit NVIDIA GPU CLOUD to learn more and get started.

https://developer.nvidia.com/cudnn

GPU Computing (cuDNN) in Data Science



AI / Deep Learning

AI and Deep learning are used in the research community and in industry to help solve many big data problems such as computer vision, speech recognition, and natural language processing. The NVIDIA Deep Learning SDK provides high-performance tools and libraries to power innovative GPU-accelerated machine learning applications in the cloud, data centers, workstations, and embedded platforms.

Accelerated Signal Processing with cuSignal

By Adam Thompson | March 5, 2021

Building a Question and Answering Service Using Natural Language Processing with NVIDIA NGC and Google Cloud

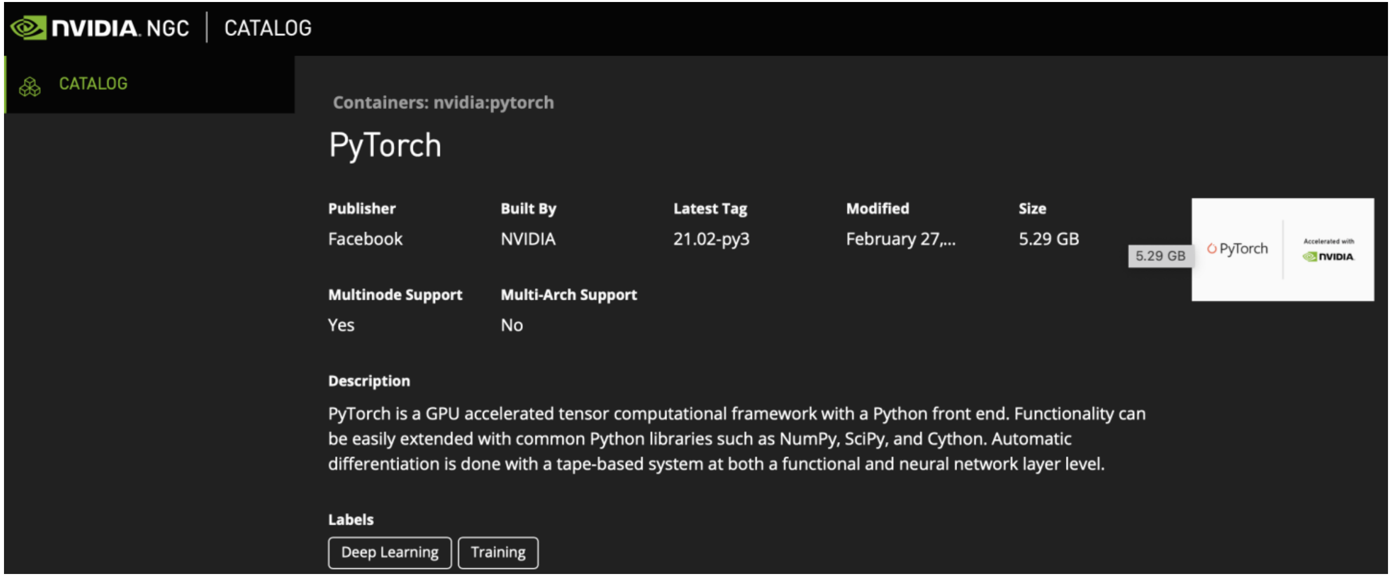
By James Sohn and Chintan Patel | March 3, 2021

Pandas DataFrame Tutorial – Beginner's Guide to GPU Accelerated DataFrames in Python

By Tom Drabas | March 3, 2021

Source: https://developer.nvidia.com/blog/category/artificial-intelligence/

GPU Computing (cuDNN) in Data Science



Containers: nvidia:pytorch

PyTorch

Publisher	Built By	Latest Tag	Modified	Size
Facebook	NVIDIA	21.02-py3	February 27,...	5.29 GB

Multinode Support	Multi-Arch Support
Yes	No

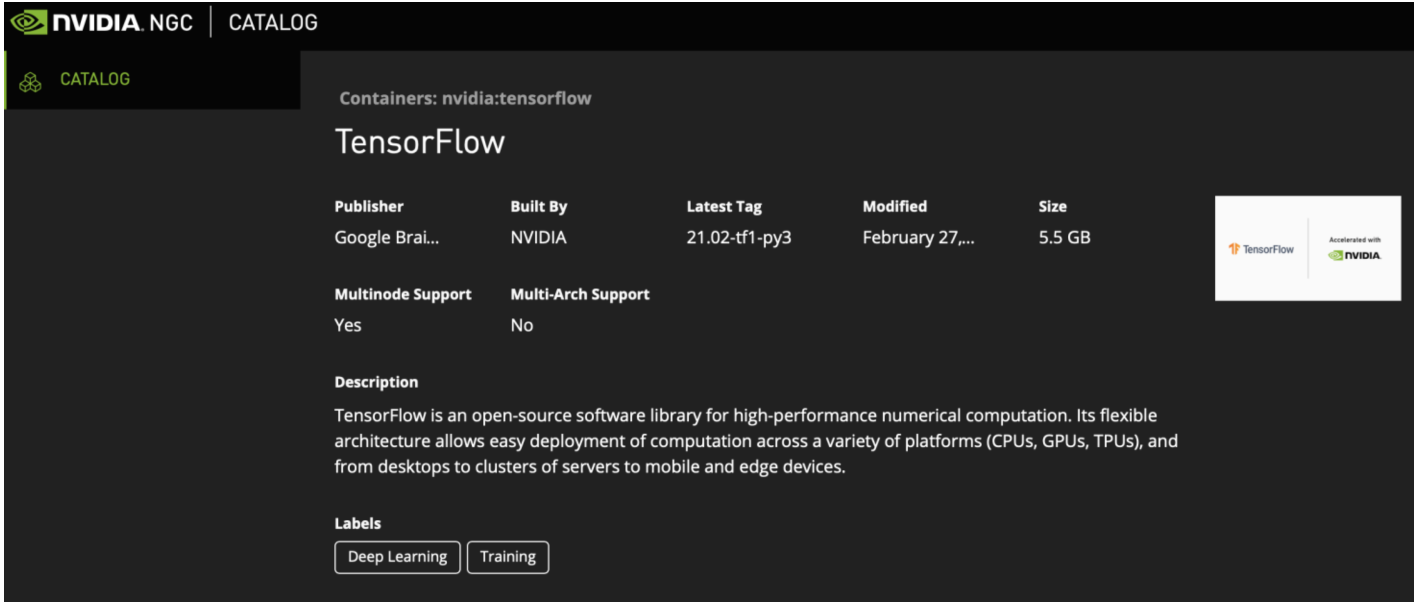
Description

PyTorch is a GPU accelerated tensor computational framework with a Python front end. Functionality can be easily extended with common Python libraries such as NumPy, SciPy, and Cython. Automatic differentiation is done with a tape-based system at both a functional and neural network layer level.

Labels

Source: https://developer.nvidia.com/deep-learning-frameworks

GPU Computing (cuDNN) in Data Science



Containers: nvidia:tensorflow

TensorFlow

Publisher	Built By	Latest Tag	Modified	Size
Google Brai...	NVIDIA	21.02-tf1-py3	February 27,...	5.5 GB

Multinode Support	Multi-Arch Support
Yes	No

Description

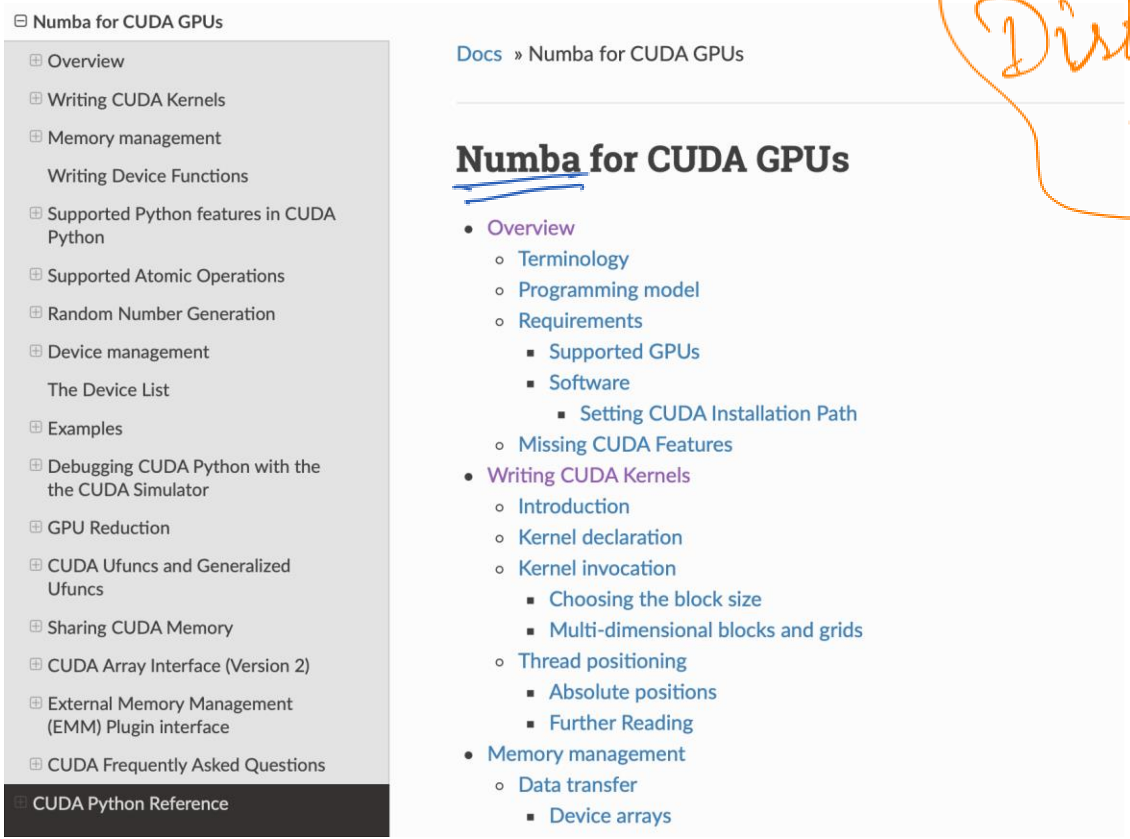
TensorFlow is an open-source software library for high-performance numerical computation. Its flexible architecture allows easy deployment of computation across a variety of platforms (CPUs, GPUs, TPUs), and from desktops to clusters of servers to mobile and edge devices.

Labels

Deep Learning | Training

Source: https://developer.nvidia.com/deep-learning-frameworks

GPU Computing in Data Science



⊟ Numba for CUDA GPUs

- ⊞ Overview
- ⊞ Writing CUDA Kernels
- ⊞ Memory management
- Writing Device Functions
- ⊞ Supported Python features in CUDA Python
- ⊞ Supported Atomic Operations
- ⊞ Random Number Generation
- ⊞ Device management
- The Device List
- ⊞ Examples
- ⊞ Debugging CUDA Python with the the CUDA Simulator
- ⊞ GPU Reduction
- ⊞ CUDA Ufuncs and Generalized Ufuncs
- ⊞ Sharing CUDA Memory
- ⊞ CUDA Array Interface (Version 2)
- ⊞ External Memory Management (EMM) Plugin interface
- ⊞ CUDA Frequently Asked Questions
- ⊞ CUDA Python Reference

Docs » Numba for CUDA GPUs

Numba for CUDA GPUs

- Overview
    - Terminology
    - Programming model
    - Requirements
        - Supported GPUs
        - Software
            - Setting CUDA Installation Path
    - Missing CUDA Features
- Writing CUDA Kernels
    - Introduction
    - Kernel declaration
    - Kernel invocation
        - Choosing the block size
        - Multi-dimensional blocks and grids
    - Thread positioning
        - Absolute positions
        - Further Reading
- Memory management
    - Data transfer
        - Device arrays

Distribute Training

MPI

https://numba.pydata.org/numba-doc/latest/cuda/index.html

https://nyu-cds.github.io/python-numba/05-cuda/

Questions

1. GPUs can be used as a stand-alone parallel system. (T/F)
2. CUDA tools can be used in any GPU architecture. (T/F)
3. Parallel computations can be performed on multiple GPUs on single node or multiple GPUs on many nodes . (T/F)
4. CUDA and MPI can be used together for an efficient parallel communication. (T/F)

GPUs for Machine Learning algorithms

GPUs for ML

Parallel Processing Capability

- High Throughput:
  - GPUs are designed to handle thousands of threads simultaneously
  - Exceptionally good at performing large matrix and vector operations
- Parallel Architecture:
  - Have a massively parallel architecture consisting of thousands of smaller, efficient cores designed for handling multiple tasks simultaneously.
  - This is in contrast to CPUs, which have fewer cores optimized for sequential serial processing.

GPUs for ML

Optimized for Matrix Operations

for - loops

- Core ML operations involve heavy matrix calculations.
- GPUs are built to perform these operations quickly and efficiently.
- This results in significant speedups in training and inference times for neural networks.

Accelerated Model Training

- Traditional CPU cores are optimized for sequential tasks, limiting their speed for ML tasks.
- GPU's parallel processing drastically cuts down model training time, making iterative development and complex model training feasible.

GPUs for ML

Scalability for Big Data

- GPUs manage large datasets more efficiently than CPUs.
- Its architecture is suited for scaling up to meet the demands of big data in ML, facilitating faster data processing and insights.

Energy Efficiency

- GPUs perform more computations per watt than CPUs.
- Lower energy costs for running intensive ML algorithms.
- The balance of performance and power consumption in GPU-enabled ML.

vs

GPUs for ML

Expansive Software Ecosystem

- Strong Ecosystem for Accelerated Computing
  - Major frameworks: TensorFlow, PyTorch, CUDA.
- Comprehensive GPU Support Optimized for GPU execution.
  - Streamlines development of ML models.
- Optimized Libraries Enhance Performance
  - Linear algebra, Fourier transforms, etc.
  - Facilitate efficient, scalable model development.

SC

GPUs for ML

Versatility

- Broad Applicability
  - GPUs accelerate a wide array of ML algorithms, not just deep learning.
- Natural Language Processing (NLP)
  - Language translation, sentiment analysis
  - Faster processing of large language models and datasets

GPT

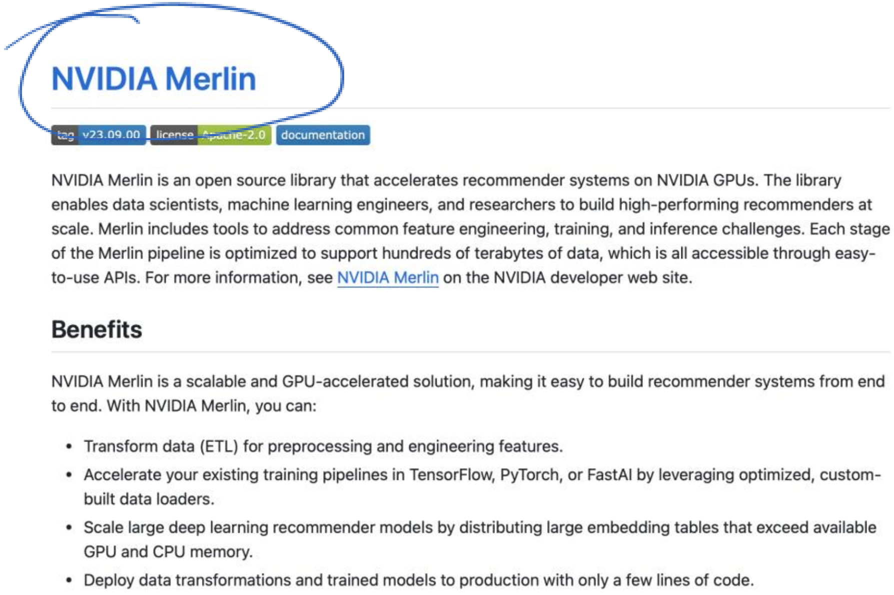
- Computer Vision
  - Image recognition, object detection
  - Real-time processing and analysis of high-resolution images and videos
- Unsupervised Learning
  - Clustering, dimensionality reduction
  - Efficient handling of large-scale datasets for pattern discovery

GPU Ecosystem Today

Contributions from Key Players

GPU Ecosystem Today

- NVIDIA's Dominance
  - NVIDIA plays pioneering role in GPU development for ML and AI.
  - NVIDIA CUDA-X AI is a complete deep learning software stack
  - Setting industry standards in GPU technology.



NVIDIA Merlin

tag v23.09.00 | license Apache-2.0 | documentation

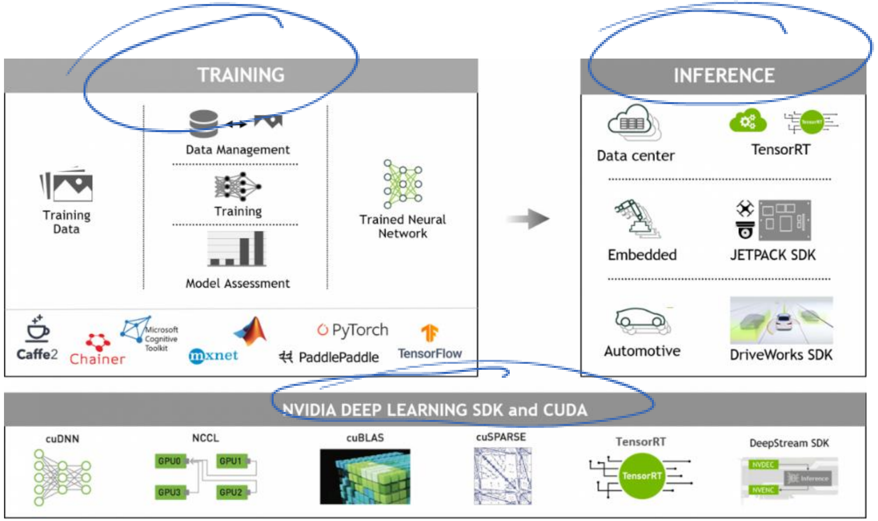
NVIDIA Merlin is an open source library that accelerates recommender systems on NVIDIA GPUs. The library enables data scientists, machine learning engineers, and researchers to build high-performing recommenders at scale. Merlin includes tools to address common feature engineering, training, and inference challenges. Each stage of the Merlin pipeline is optimized to support hundreds of terabytes of data, which is all accessible through easy-to-use APIs. For more information, see NVIDIA Merlin on the NVIDIA developer web site.

Benefits

NVIDIA Merlin is a scalable and GPU-accelerated solution, making it easy to build recommender systems from end to end. With NVIDIA Merlin, you can:

- Transform data (ETL) for preprocessing and engineering features.
- Accelerate your existing training pipelines in TensorFlow, PyTorch, or FastAI by leveraging optimized, custom-built data loaders.
- Scale large deep learning recommender models by distributing large embedding tables that exceed available GPU and CPU memory.
- Deploy data transformations and trained models to production with only a few lines of code.

Source: https://github.com/NVIDIA-Merlin/Merlin

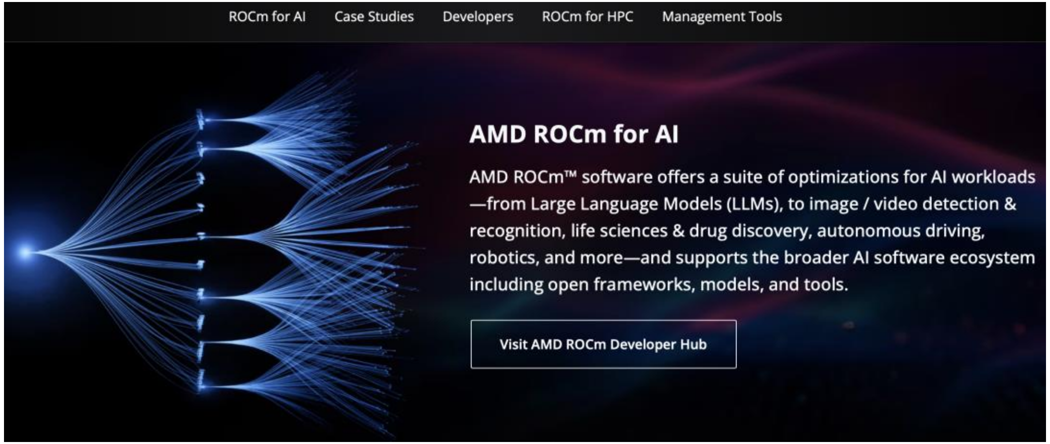


Source: https://developer.nvidia.com/deep-learning-software

GPU Ecosystem Today

AMD's Advancements

- AMD's focus on open-source platforms.
- ROCm (Radeon Open Compute) platform and its support for deep learning frameworks.
- Providing an alternative to NVIDIA's ecosystem with competitive performance.



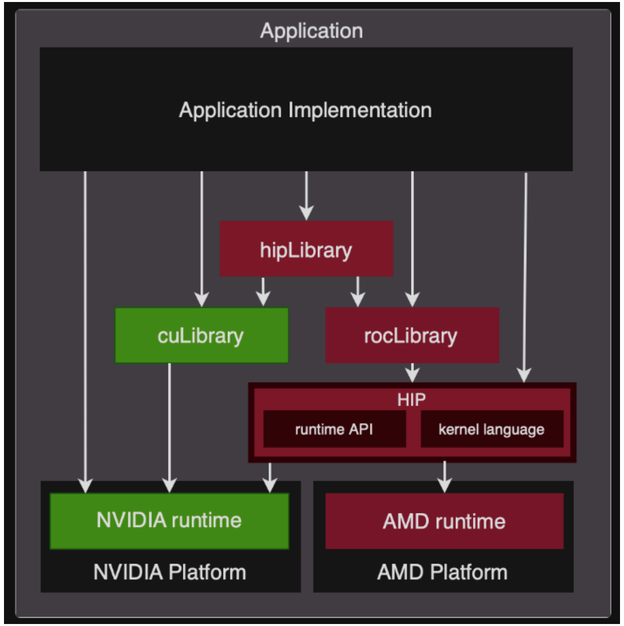
AMD ROCm for AI

AMD ROCm™ software offers a suite of optimizations for AI workloads—from Large Language Models (LLMs), to image / video detection & recognition, life sciences & drug discovery, autonomous driving, robotics, and more—and supports the broader AI software ecosystem including open frameworks, models, and tools.

Source: https://www.amd.com/en/products/software/rocm.html

GPU Ecosystem Today

- **Heterogeneous-computing Interface for Portability (HIP)**
  - a C++ runtime API and kernel language that lets developers create portable applications running in heterogeneous systems, using CPUs and AMD GPUs or NVIDIA GPUs from a single source code.
  - Provides a simple marshalling language to access either the AMD ROCM back-end, or NVIDIA CUDA back-end, to build and run application kernels.

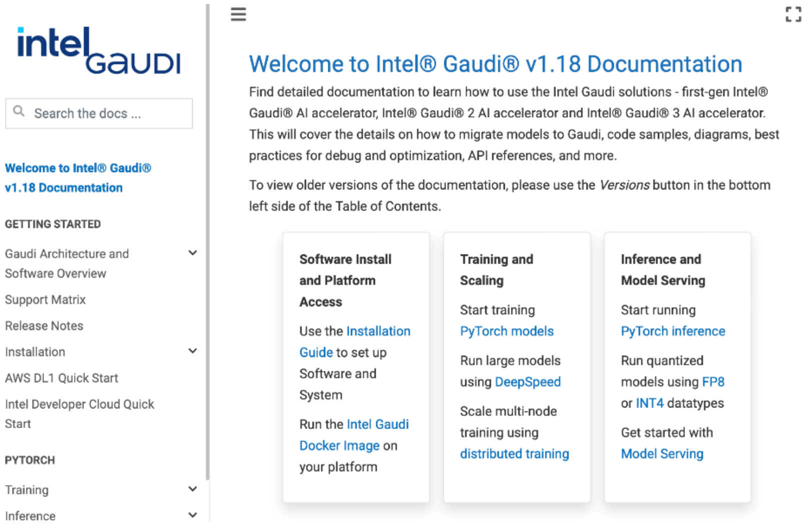


Source: https://rocm.docs.amd.com/projects/HIP/en/latest/what_is_hip.html

GPU Ecosystem Today

Intel GaUDI

- Intel's recent efforts in GPU technology for AI and ML, focusing on versatility.
- Intel's Xe architecture GPUs and oneAPI.
- Diversifying the GPU market and offering integrated CPU-GPU solutions.



Welcome to Intel® Gaudi® v1.18 Documentation

Find detailed documentation to learn how to use the Intel Gaudi solutions - first-gen Intel® Gaudi® AI accelerator, Intel® Gaudi® 2 AI accelerator and Intel® Gaudi® 3 AI accelerator. This will cover the details on how to migrate models to Gaudi, code samples, diagrams, best practices for debug and optimization, API references, and more.

To view older versions of the documentation, please use the *Versions* button in the bottom left side of the Table of Contents.

Software Install and Platform Access

Use the Installation Guide to set up Software and System

Run the Intel Gaudi Docker Image on your platform

Training and Scaling

Start training PyTorch models

Run large models using DeepSpeed

Scale multi-node training using distributed training

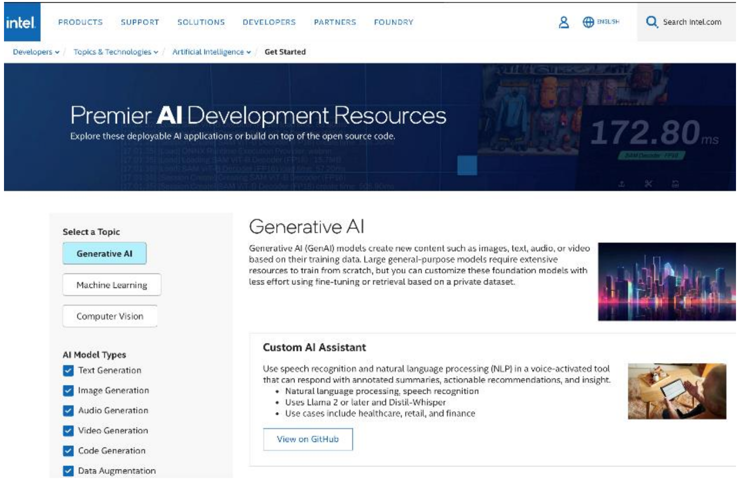
Inference and Model Serving

Start running PyTorch inference

Run quantized models using FP8 or INT4 datatypes

Get started with Model Serving

Source: https://habana.ai/products/



Premier AI Development Resources

Explore these deployable AI applications or build on top of the open source code.

Select a Topic

Generative AI

Machine Learning

Computer Vision

AI Model Types

☑ Text Generation
☑ Image Generation
☑ Audio Generation
☑ Video Generation
☑ Code Generation
☑ Data Augmentation

Generative AI

Generative AI (GenAI) models create new content such as images, text, audio, or video based on their training data. Large general-purpose models require extensive resources to train from scratch, but you can customize these foundation models with less effort using fine-tuning or retrieval based on a private dataset.

Custom AI Assistant

Use speech recognition and natural language processing (NLP) in a voice-activated tool that can respond with annotated summaries, actionable recommendations, and insight.

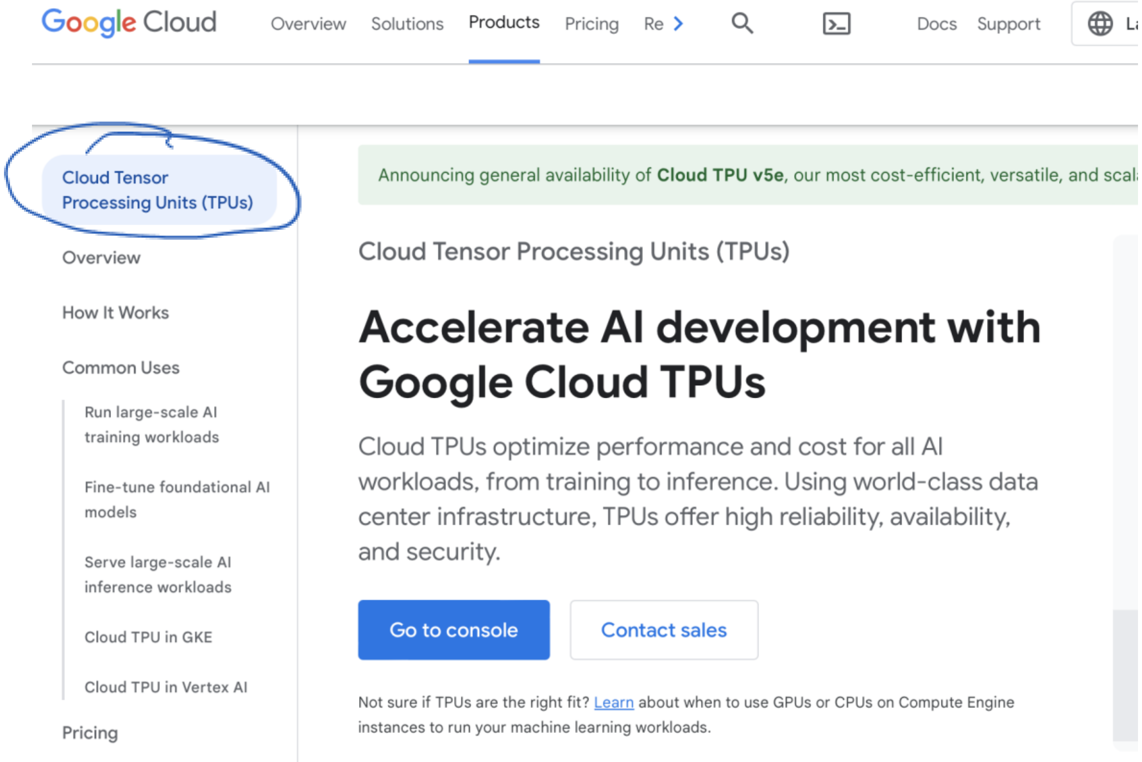
- Natural language processing, speech recognition
- Uses Llama 2 or later and Distil-Whisper
- Use cases include healthcare, retail, and finance

View on GitHub

Source: https://www.intel.com/content/www/us/en/developer/topic-technology/artificial-intelligence/get-started.html

GPU Ecosystem Today

Source:
https://cloud.google.com/tpu



Announcing general availability of **Cloud TPU v5e**, our most cost-efficient, versatile, and scal

Cloud Tensor Processing Units (TPUs)

Accelerate AI development with Google Cloud TPUs

Cloud TPUs optimize performance and cost for all AI workloads, from training to inference. Using world-class data center infrastructure, TPUs offer high reliability, availability, and security.

Not sure if TPUs are the right fit? Learn about when to use GPUs or CPUs on Compute Engine instances to run your machine learning workloads.

Nvidia CUDA: Machine Learning Libraries

- **Major ML Libraries Based on CUDA**
  - **TensorFlow**: A popular deep learning framework that can utilize CUDA for accelerated model training and inference.
  - **PyTorch**: Another widely-used framework for deep learning, leveraging CUDA for faster computation and model experimentation.
  - **cuDNN** (CUDA Deep Neural Network library): A GPU-accelerated library for deep neural networks that provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers.
  - **RAPIDS**: A suite of open-source software libraries, including cuDF and cuML, for executing end-to-end data science and analytics pipelines entirely on GPUs.

GPUs for ML

Summary

- GPUs are critical to machine learning (ML) computations.
- They provide substantial improvements in speed, efficiency, and scalability.
- There are several open-source GPU libraries, mostly aligned with a single hardware vendor.
- With the increasing complexity and size of ML models and datasets, GPUs are set to become increasingly indispensable in ML.