GPUs and CUDA in Python

Maciej Marchwiany, PhD

https://www.linkedin.com/pulse/ai-powered-physics-how-ai-revolutionizing-scientific-discovery-cain-ddbwc

Plan

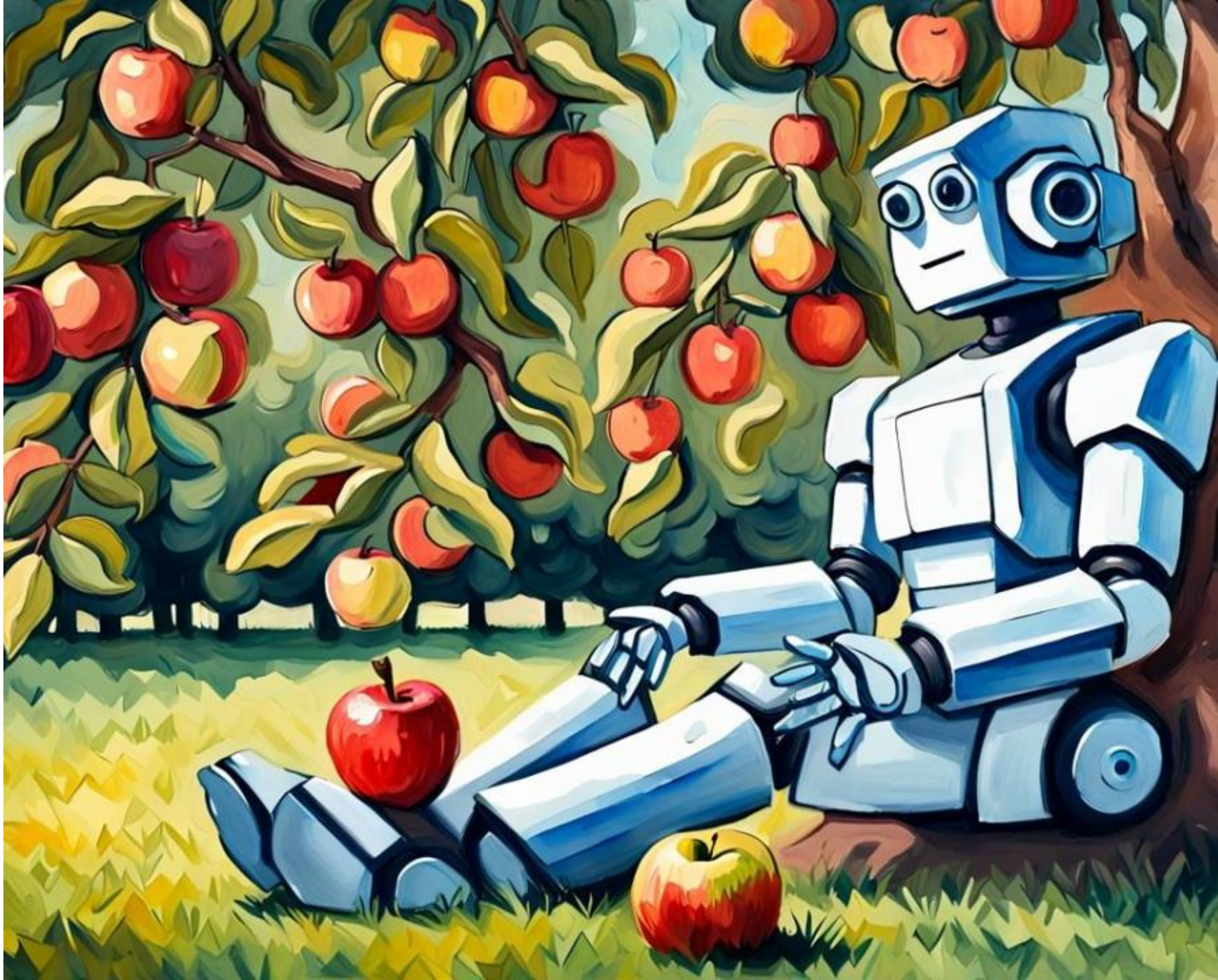
GPU

CUDA

cuPy/cuDF

profiling

GPU



Nvidia GPU

A **GPU (Graphics Processing Unit)** is a specialized processor designed to accelerate graphics rendering and parallel computing tasks. Originally developed to handle the complex calculations required for rendering images, videos, and animations, GPUs are now widely used in various computing applications, including artificial intelligence (AI), machine learning (ML), cryptocurrency mining, and scientific simulations.

Key Features

- **Parallel Processing:** Unlike a CPU (Central Processing Unit), which has a few powerful cores optimized for sequential tasks, a GPU consists of thousands of smaller cores that work together to process many tasks simultaneously.
- **High Performance in Graphics Rendering**: GPUs are essential for gaming, 3D modeling, video editing, and virtual reality applications.
- **Machine Learning and AI Acceleration**: Modern GPUs, such as those from NVIDIA (CUDA cores) and AMD (Stream Processors), are optimized for deep learning and data science applications.
- **General-Purpose Computing** (GPGPU): GPUs are increasingly used for tasks beyond graphics, such as scientific simulations, cryptography, and financial modeling.

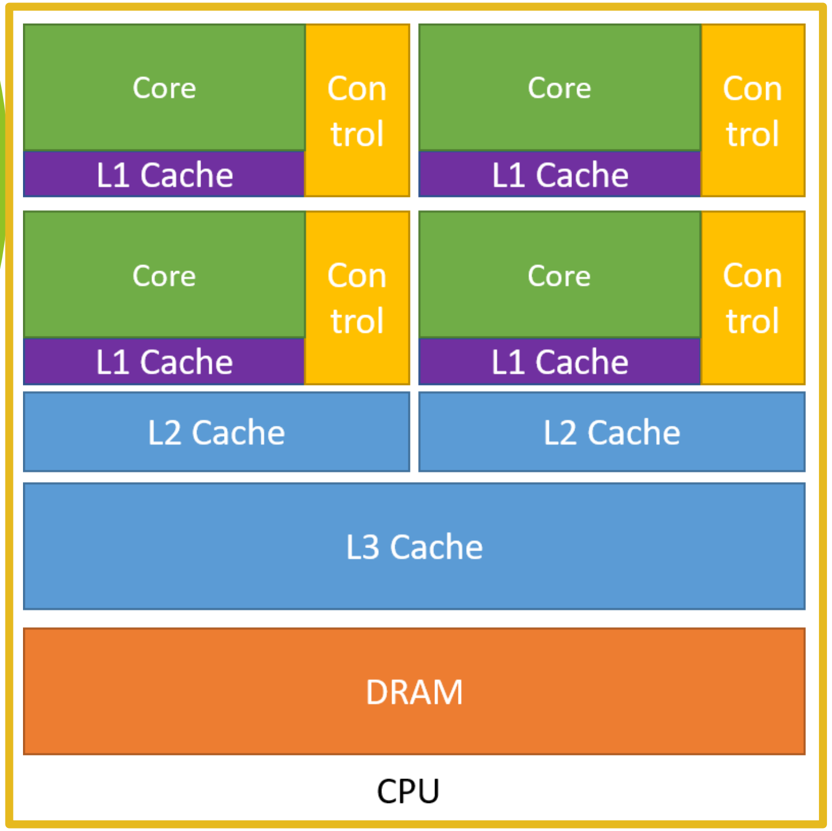
Core Components

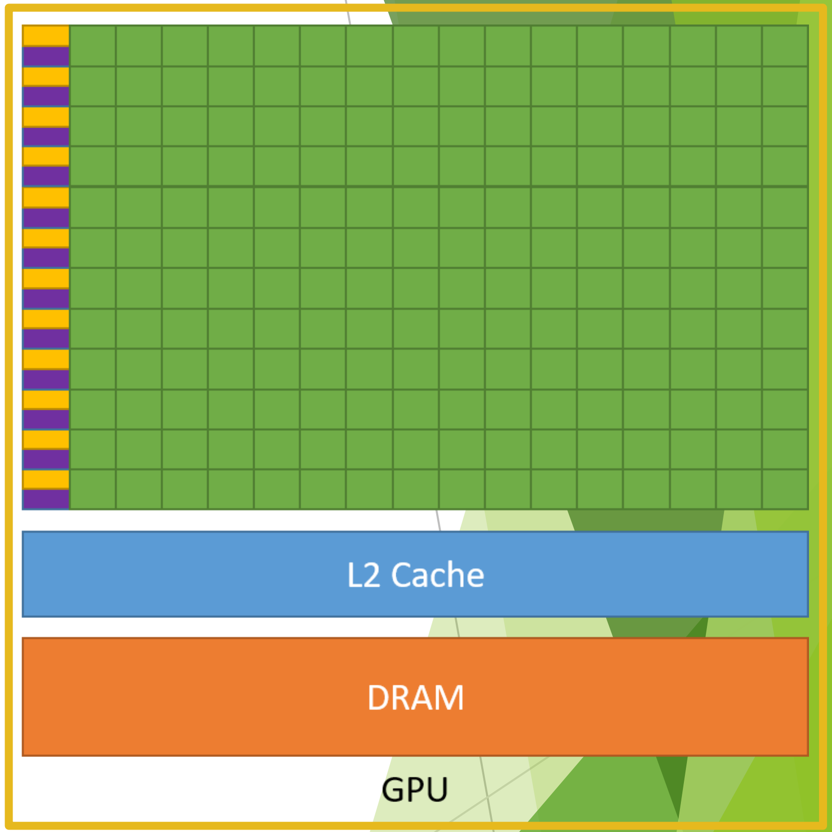
- **Streaming Multiprocessors** (SMs) – Fundamental processing units containing CUDA cores, Tensor cores, and shared memory.
- **CUDA Cores** - Execute floating-point and integer operations in parallel.
- **Tensor Cores** - Specialized cores for AI and deep learning matrix operations.
- **Ray Tracing** (RT) **Cores** - Accelerate real-time ray tracing for realistic lighting and reflections.
- **Warp Scheduler** - Manages execution of multiple threads in a warp (group of 32 threads).

Memory Hierarchy

- **Global Memory**: Offers high-speed data transfer for external memory access.
- **Shared Memory**: Enables collaborative computing among threads within an SM.
- L1 Cache: Provides fast access to frequently used data.
- L2 Cache: Balances speed and capacity for efficient data access.
- Texture Memory: a read-only, cached GPU memory.
- Register File: Stores temporary variables for fast thread execution.

GPU vs CPU





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

GPU vs CPU

Feature	GPU	CPU
Purpose	Optimized for **parallel processing**	Optimized for **sequential processing**
Parallelism	Executes **thousands** of threads simultaneously	Executes **a few** threads at high speed
Architecture	Many **smaller, efficient cores** (SMs)	Few **powerful, complex cores**
Instruction Throughput	High throughput, lower single-thread performance	High single-thread performance
Memory Bandwidth	High memory bandwidth for handling large datasets	Lower memory bandwidth but optimized for cache efficiency
Flexibility	Highly parallel but less flexible for general computing	More flexible for various workloads
Energy Efficiency	More efficient for **massively parallel workloads**	More efficient for **single-threaded tasks**

Streaming Multiprocessors

functions like a mini-CPU

32 threads executing in parallel

A **Streaming Multiprocessor (SM)** is the **core computational unit** in an NVIDIA GPU, responsible for executing **parallel threads**. Each GPU consists of multiple SMs, which contain smaller execution units like **CUDA Cores, Tensor Cores, and memory components**.

Each **SM** contains:

- **CUDA Cores** - Handle integer and floating-point calculations.
- **Tensor Cores** - Accelerate AI and matrix computations.
- **Ray Tracing (RT) Cores** - Process real-time lighting & reflections.
- **Warp Scheduler** - Manages execution of 32-thread groups (warps).
- **Register File** - Stores temporary thread variables for fast access.
- **L1 Cache & Shared Memory** - Improves memory access speed.

GPU and CPU Connection

- **PCI**
  Most common connection for NVIDIA GPUs in desktops, laptops, and servers. Uses the PCIe slot on the motherboard to transfer data between CPU and GPU. **Bandwidth up to 64 GB/s**
- **NVLink**
  Much **faster** than PCIe for **GPU-to-GPU** and **GPU-to-CPU** communication. Provides higher bandwidth (e.g., **300 GB/s for** NVLink 2.0).
- **SXM** (Server GPU Form Factor)
  Used in **NVIDIA Data Center** GPUs (e.g., A100 SXM4, H100 SXM5). Higher power and thermal efficiency than PCIe versions. **Directly** connects to the **CPU** via NVLink or proprietary interfaces. Mounted on specialized motherboards (e.g., NVIDIA DGX systems).
- **CXL** (Compute Express Link)

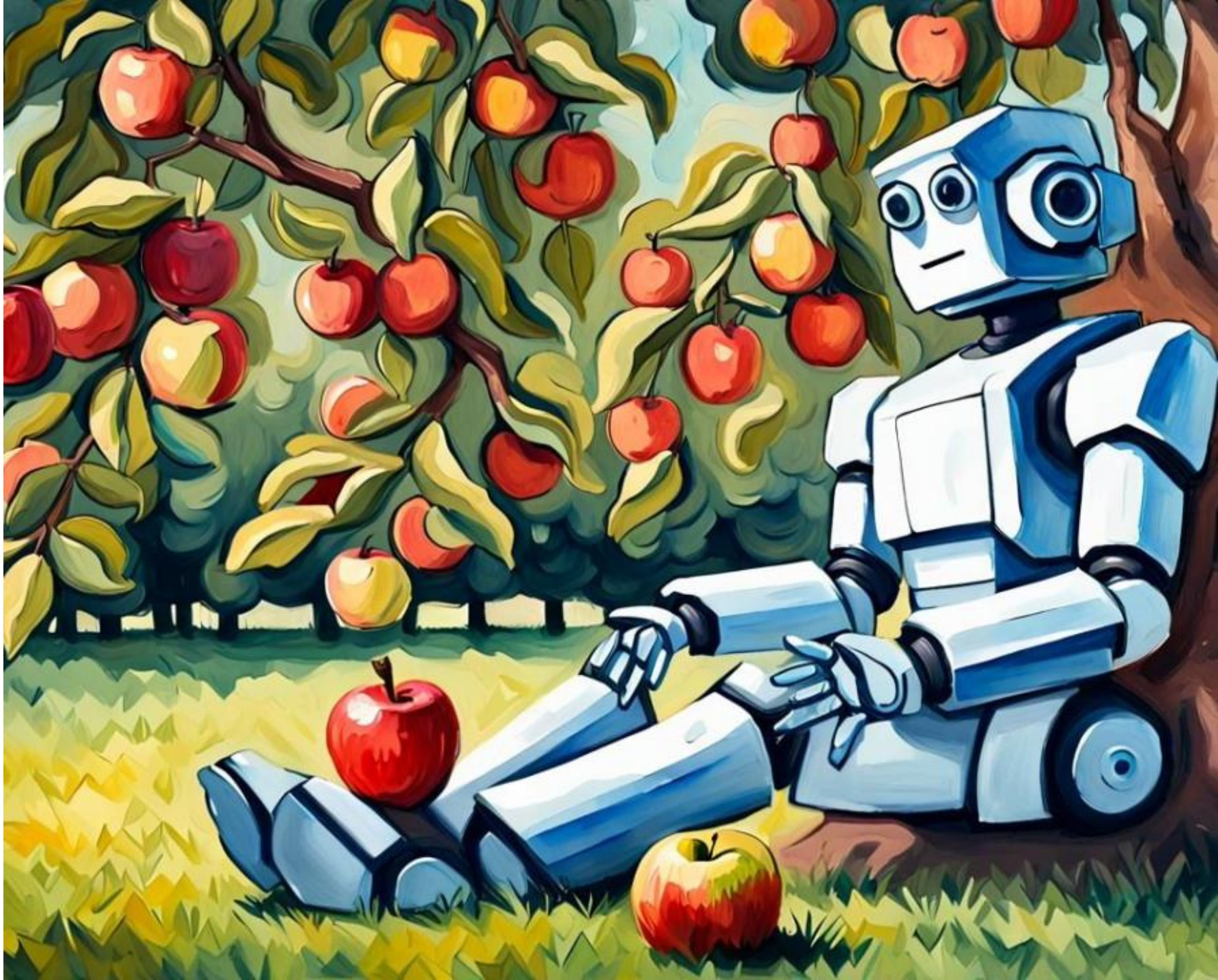
Compute Capability

Compute Capability (CC) in CUDA represents the **GPU's hardware features and capabilities.** Each NVIDIA GPU has a **Compute Capability version**, which determines:

- **Supported CUDA features**
- **Maximum number of threads, warps, and registers**
- **Hardware-accelerated operations (Tensor Cores, RT Cores, etc.)**

CC	Architecture	Example GPUs	Key Features
1.x	Tesla	GeForce 8, 9	Basic CUDA support
2.x	Fermi	GTX 400, 500	Faster atomic operations
3.x	Kepler	GTX 600, 700	Dynamic Parallelism
5.x	Maxwell	GTX 900	Unified memory
6.x	Pascal	GTX 10xx	Tensor cores introduced (6.1+)
7.x	Volta/Turing	RTX 20xx, V100	Tensor cores, FP16 support
8.x	Ampere	RTX 30xx, A100	More Tensor Cores, FP64 improvements
9.x	Hopper	H100	Transformer Engine, FP8 support

CUDA



CUDA

CUDA is **NVIDIA's parallel computing platform and programming model** that allows developers to use **GPUs for general-purpose computing (GPGPU)**. It enables **massively parallel execution** of computations, making it ideal for **AI, machine learning, scientific computing, and high-performance applications.**

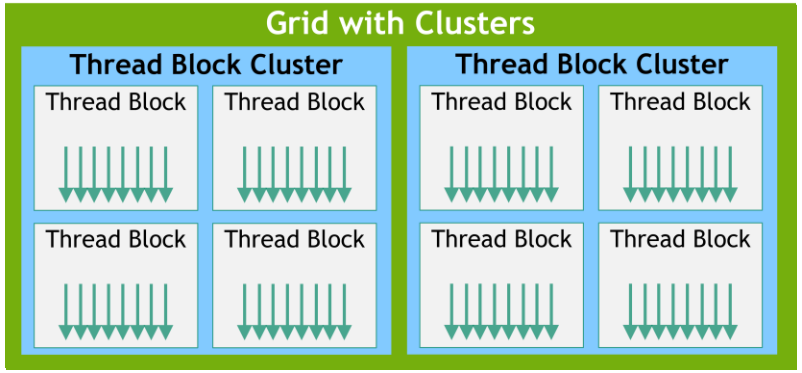
Key CUDA Concepts

Term	Description
CUDA Cores	Small processing units inside a GPU, executing threads in parallel.
Threads	Smallest execution units running on CUDA cores.
Blocks	Groups of threads that execute in parallel within a Streaming Multiprocessor (SM).
Grids	Collections of blocks working together on large-scale computations.
Global Memory	Large, slow-access memory shared across all threads.
Shared Memory	Faster, per-block memory used for thread collaboration.
Registers	Ultra-fast storage for individual threads.
Warp	A group of 32 threads executing in lockstep within an SM.

CUDA Kernel

A **CUDA kernel** is a **function that runs on the GPU** and executes **in parallel across multiple threads**. It is the core of **CUDA parallel programming**, enabling massive performance boosts for compute-heavy tasks.

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



Launching a CUDA Kernel

```
int N = 1024;
dim3 threadsPerBlock(256);
dim3 blocksPerGrid(
       (N + threadsPerBlock.x - 1) / threadsPerBlock.x  );

add<<<blocksPerGrid, threadsPerBlock>>>(d_a, d_b, d_c, N);
```

Synchronization

Synchronization in CUDA ensures that **threads, warps, or streams wait for each other** before proceeding to the next step. It helps **coordinate execution** and **prevent race conditions** when accessing shared resources.

CUDA synchronization can be categorized into:

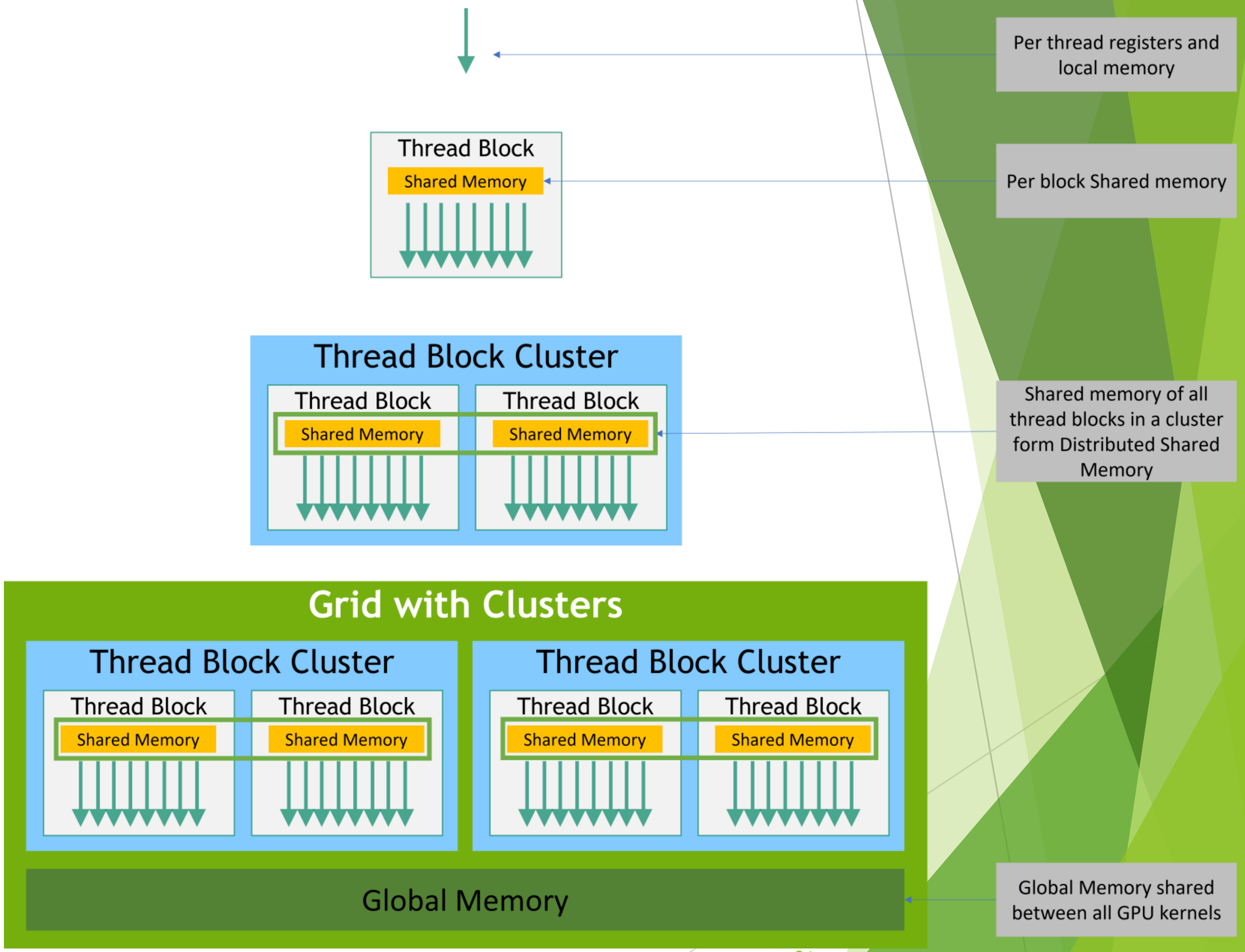
- **Thread-Level Synchronization** (within a block)
- **Block-Level Synchronization** (between blocks)

Implicit Synchronization Case	Why It Happens	Solution to Avoid Blocking
Default Stream (0)	Enforces sequential execution	Use multiple streams for concurrency
cudaMemcpy()	Blocks CPU until previous GPU work is complete	Use cudaMemcpyAsync()
cudaFree()	Waits for all GPU work to finish before freeing memory	Free memory after checking completion
cudaDeviceSynchronize()	Blocks CPU until all GPU work is done	Use only when necessary

Memory Types

Memory Type	Scope	Speed	Use Case
Global Memory	All threads	Slow	Large datasets, long-term storage
Shared Memory	Threads in the same block	Fast	Fast intra-block communication
Registers	Single thread	Fastest	Temporary thread-local variables
Constant Memory	All threads	Fast (read-only)	Frequent but unchanging data
Texture memory	All threads	Fast (read-only)	Neighboring thread access

Memory Hierarchy



https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html?highlight=warp#memory-hierarchy

Unified Memory

Unified Memory (**UM**) in CUDA is a **single memory space** that is **shared between the CPU and GPUs.** It allows **seamless memory access** without requiring **explicit memory copies (cudaMemcpy)** between host and device.

- **Single Memory Space:** No need to manually allocate separate memory for CPU (host) and GPU (device).
- **Automatic Data Movement:** CUDA handles memory transfers on-demand between CPU and GPU.
- **Simplifies Programming:** Reduces memory management complexity (no explicit cudaMemcpy()).
- **Accessible from Multiple GPUs:** Supports multi-GPU memory access.
- **Page Migration:** Moves data automatically between CPU and GPU only when needed.

Streams

A **CUDA stream** is a **sequence of GPU operations (kernels, memory copies, etc.)** that execute **in order** within that stream, but **can run concurrently** with operations in other streams.

Think of streams as independent execution pipelines on the GPU.

Streams are for:

- **Overlapping Computation & Memory Transfers**
  (Hide memory copy overhead)
- **Concurrent Kernel Execution**
  (Run multiple kernels at the same time)
- **Efficient Multi-GPU Workloads**
  (Assign different streams to different GPUs)
- **Pipeline Processing**
  (Process chunks of data in parallel)

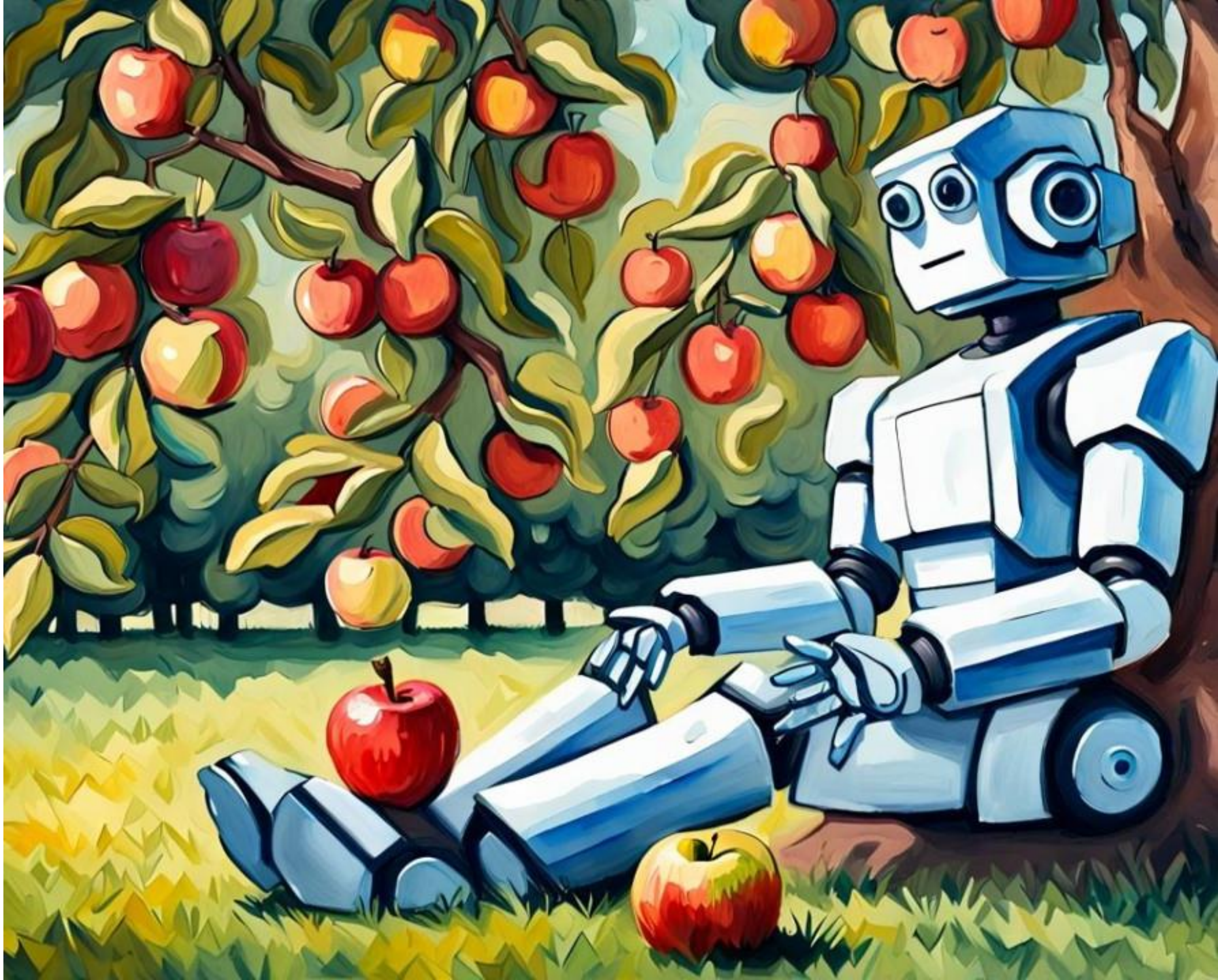
Code

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

int main(){
    int N = ...;
    size_t size = N * sizeof(float);

    float* h_A = (float*)malloc(size);
    float* h_B = (float*)malloc(size);
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    Float *d_A, * d_B, * d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

Code

```
    float* h_C = (float*)malloc(size);

    // Initialize input vectors
    ...

    Float *d_A, * d_B, * d_C;
    cudaMalloc(&d_A, size);
    cudaMalloc(&d_B, size);
    cudaMalloc(&d_C, size);

    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    int thrPerBlock = 256;
    int blPerGrid = (N + thrPerBlock - 1) / thrPerBlock;
    VecAdd<<<blPerGrid, thrPerBlock>>>(d_A, d_B, d_C, N);

    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    cudaFree(d_A);    cudaFree(d_B);    cudaFree(d_C);
    // Free host memory
    ...
}
```

cuPy/cuDF



cuPy

CuPy is a **GPU-accelerated array library** that provides a **NumPy-compatible interface**, allowing you to perform **high-speed numerical computations on NVIDIA GPUs using CUDA**. It is designed as a drop-in replacement for NumPy, meaning you can switch to CuPy with minimal code changes to benefit from GPU acceleration.

- **NumPy-compatible**: Use familiar NumPy syntax (cupy.array, cupy.dot, etc.).
- **CUDA-powered**: Runs computations on NVIDIA GPUs for massive speedups.
- **Efficient memory management**: Avoids CPU-GPU memory transfer overhead.
- **Supports advanced operations**: Linear algebra, FFTs, random number generation, sparse matrices, etc.
- **Seamless interoperability**: Easily switch between NumPy (CPU) and CuPy (GPU).

cuPy

```python
import cupy as cp

gpu_array = cp.array([1, 2, 3, 4, 5])

# Perform operations (executed on GPU)
gpu_array = gpu_array * 2

print(gpu_array)  # Output: [ 2  4  6  8 10 ]
```

```python
import numpy as np

cpu_array = np.array([1, 2, 3, 4, 5])

# Perform operations (executed on CPU)
cpu_array = cpu_array * 2

print(cpu_array)  # Output: [ 2  4  6  8 10 ]
```

Data transfer

- Move arrays to a device

```
x_cpu = np.array([1, 2, 3])
# move the data to the current device.
x_gpu = cp.asarray(x_cpu)
```

- Move arrays to the host

```
x_gpu = cp.array([1, 2, 3])
# move the data to the host.
x_cpu = cp.asnumpy(x_gpu)
```

Elementwise kernels

An **Elementwise Kernel** in **CuPy** is a way to apply **custom element-wise operations** to arrays on a **GPU**, using **CUDA C++ code**. This provides massive speedups over traditional Python loops.

- **Highly optimized** for **parallel** computation on the GPU.
- Much **faster** than looping in Python.
- **More efficient** than cp.vectorize() for element-wise operations.
- **Works** directly on **GPU** (no CPU-GPU memory transfers).

```
cp.ElementwiseKernel(
    'input_signature',   # Input data types
    'output_signature',  # Output data types
    'kernel_code',       # CUDA C++ code
    'kernel_name'        # Unique kernel identifier
)
```

Elementwise kernels

```
import cupy as cp

# Define the elementwise kernel
custom_add = cp.ElementwiseKernel(
    'float32 x, float32 y',  # Inputs: x and y (float32)
    'float32 z',             # Output: z (float32)
    'z = (x + y) * 2;',      # CUDA C++ code (runs on GPU)
    'custom_add_kernel'      # Kernel name
)

# Create GPU arrays
a = cp.array([1, 2, 3, 4, 5], dtype=cp.float32)
b = cp.array([10, 20, 30, 40, 50], dtype=cp.float32)

# Apply the kernel (runs on GPU)
result = custom_add(a, b)
print(result)  # Output: [ 22.  44.  66.  88. 110.]
```

Reduction kernels

A **Reduction Kernel** in CuPy is used to reduce an array **along a given axis** using custom operations in parallel on the GPU. It is similar to NumPy's *reduce* operations like `sum()`, `max()`, or `mean()`, but runs on the GPU for high performance.

- **Highly optimized** for **parallel computation** on the GPU.
- Much **faster** than looping in Python.
- **More flexible** than built-in cp.sum() or cp.max(), as you can define custom reductions.
- **Handles large datasets efficiently** without moving data to the CPU.

```
cp.ReductionKernel(
    'input_signature',   # Input data type
    'output_signature',  # Output data type
    'map_expr',          # Mapping operation (per element)
    'reduce_expr',       # Reduction operation
    'post_map_expr',     # Final transformation (optional)
    'identity',          # Identity value for the reduction
    'kernel_name'        # Unique kernel identifier
)
```

Reduction kernels

```
import cupy as cp

# Define a sum reduction kernel
custom_sum = cp.ReductionKernel(
    'float32 x',         # Input: x (float32)
    'float32 y',         # Output: y (float32)
    'x',                 # Map: Each element is used as-is
    'a + b',             # Reduce: Sum operation
    'y = a',             # Post-process
    '0',                 # Identity: Start from 0
    'custom_sum_kernel' # Kernel name
)

# Create a GPU array
arr = cp.array([1, 2, 3, 4, 5], dtype=cp.float32)

# Apply the reduction kernel
result = custom_sum(arr)
print(result)  # Output: 15.0
```

Raw kernels

```
add_kernel = cp.RawKernel(r'''
extern "C" __global__
void my_add(const float* x1, const float* x2, float* y) {
    int tid = blockDim.x * blockIdx.x + threadIdx.x;
    y[tid] = x1[tid] + x2[tid];
}
''', 'my_add')

x1 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
x2 = cp.arange(25, dtype=cp.float32).reshape(5, 5)
y = cp.zeros((5, 5), dtype=cp.float32)
add_kernel((5,), (5,), (x1, x2, y))
```

```
array([[ 0.,  2.,  4.,  6.,  8.],
       [10., 12., 14., 16., 18.],
       [20., 22., 24., 26., 28.],
       [30., 32., 34., 36., 38.],
       [40., 42., 44., 46., 48.]], dtype=float32)
```

Streams

```
s = cp.cuda.Stream()
with s:
    a_cp = cp.asarray(a_np)
    b_cp = cp.sum(a_cp)
    assert s == cp.cuda.get_current_stream()
```

```
s = cp.cuda.Stream()
s.use()
b_np = cp.asnumpy(b_cp)
assert s == cp.cuda.get_current_stream()
cp.cuda.Stream.null.use()
assert cp.cuda.Stream.null == cp.cuda.get_current_stream()
```

cuDF

cuDF is a GPU-accelerated DataFrame library from RAPIDS AI that provides a Pandas-like interface for working with tabular data on NVIDIA GPUs. It is designed for high-performance data manipulation and analytics by leveraging CUDA and GPU parallelism.
Features:

- **Pandas-like API**: It allows users to write code similar to Pandas but runs on GPUs for faster execution.
- **GPU Acceleration**: Utilizes NVIDIA CUDA to process large datasets significantly faster than CPU-based libraries.
- **Interoperability**: Works with other RAPIDS libraries such as cuML (machine learning) and cuGraph (graph analytics).
- **Seamless Data Transfer**: Supports interoperability with PyArrow, Dask, and other GPU-accelerated libraries like CuPy.
- **Efficient Memory Management**: Handles large datasets without the memory limitations of CPUs.

cuDF is Pandas

```python
import cudf

# Load data
df = cudf.read_csv('data.csv')

# Perform some operations
df_filtered = df[df['column'] > 10]
df_grouped = df.groupby('category').agg({'value': 'mean'})

# Display results
print(df_filtered.head())
print(df_grouped.head())
```

```python
import pandas

# Load data
df = pandas.read_csv('data.csv')

# Perform some operations
df_filtered = df[df['column'] > 10]
df_grouped = df.groupby('category').agg({'value': 'mean'})

# Display results
print(df_filtered.head())
print(df_grouped.head())
```

cuML

cuML is a **GPU-accelerated machine learning library** that is part of the RAPIDS AI ecosystem. It provides **scikit-learn-like APIs** but runs computations on NVIDIA GPUs using CUDA, enabling **massively parallel processing** for faster model training and inference. Features:

- **Scikit-learn-like API**: Easily switch from CPU-based scikit-learn to GPU-accelerated cuML with minimal code changes.
- **GPU Acceleration**: Leverages CUDA and RAPIDS for high-speed data processing.
- **Integration with cuDF**: Works seamlessly with cuDF (GPU DataFrames) for efficient end-to-end workflows.
- **Multi-GPU and Multi-node Support**: Supports distributed training using Dask.
- **Interoperability**: Can exchange data with other GPU-based libraries like PyTorch, TensorFlow, Dask, and Numba.

Algorithms in cuML

Regression & Classification

- Linear Regression (LinearRegression)
- Ridge Regression (Ridge)
- Logistic Regression (LogisticRegression)
- K-Nearest Neighbors (KNeighborsClassifier, KNeighborsRegressor)
- Support Vector Machines (SVC, SVR)

Clustering

- K-Means (KMeans)
- DBSCAN (DBSCAN)
- Mean-Shift (MeanShift)

Dimensionality Reduction

- Principal Component Analysis (PCA)
- Truncated Singular Value Decomposition (TSVD)
- UMAP (UMAP)
- SNE (TSNE)
- Non-Negative Matrix Factorization (NMF)

Model Evaluation & Utilities

- Train-test splitting (train_test_split)
- Cross-validation (cross_val_score)
- Randomized search (RandomizedSearchCV)

cuML: Logistic Regression

```
import cudf
from cuml.linear_model import LogisticRegression


# Load data
df = cudf.read_csv('data.csv’)


# Split into features and target
X = df[['feature1', 'feature2’]]
y = df['label']


# Train a logistic regression model
model = LogisticRegression()
model.fit(X, y)


# Make predictions
predictions = model.predict(X)
print(predictions)
```

cuML: Logistic Regression

```
import pandas
from sklearn.linear_model import LogisticRegression


# Load data
df = pandas.read_csv('data.csv’)


# Split into features and target
X = df[['feature1', 'feature2’]]
y = df['label']


# Train a logistic regression model
model = LogisticRegression()
model.fit(X, y)


# Make predictions
predictions = model.predict(X)
print(predictions)
```

GPU in PyTorch

- Check GPU Availability

```
import torch
print("CUDA Available:", torch.cuda.is_available())
```

- Move data to GPU

```
device = torch.device(
     "cuda" if torch.cuda.is_available() else "cpu")
x = torch.rand(3, 3).to(device)
```

```
x = torch.rand(3, 3, device=device)
```

- Move model to GPU

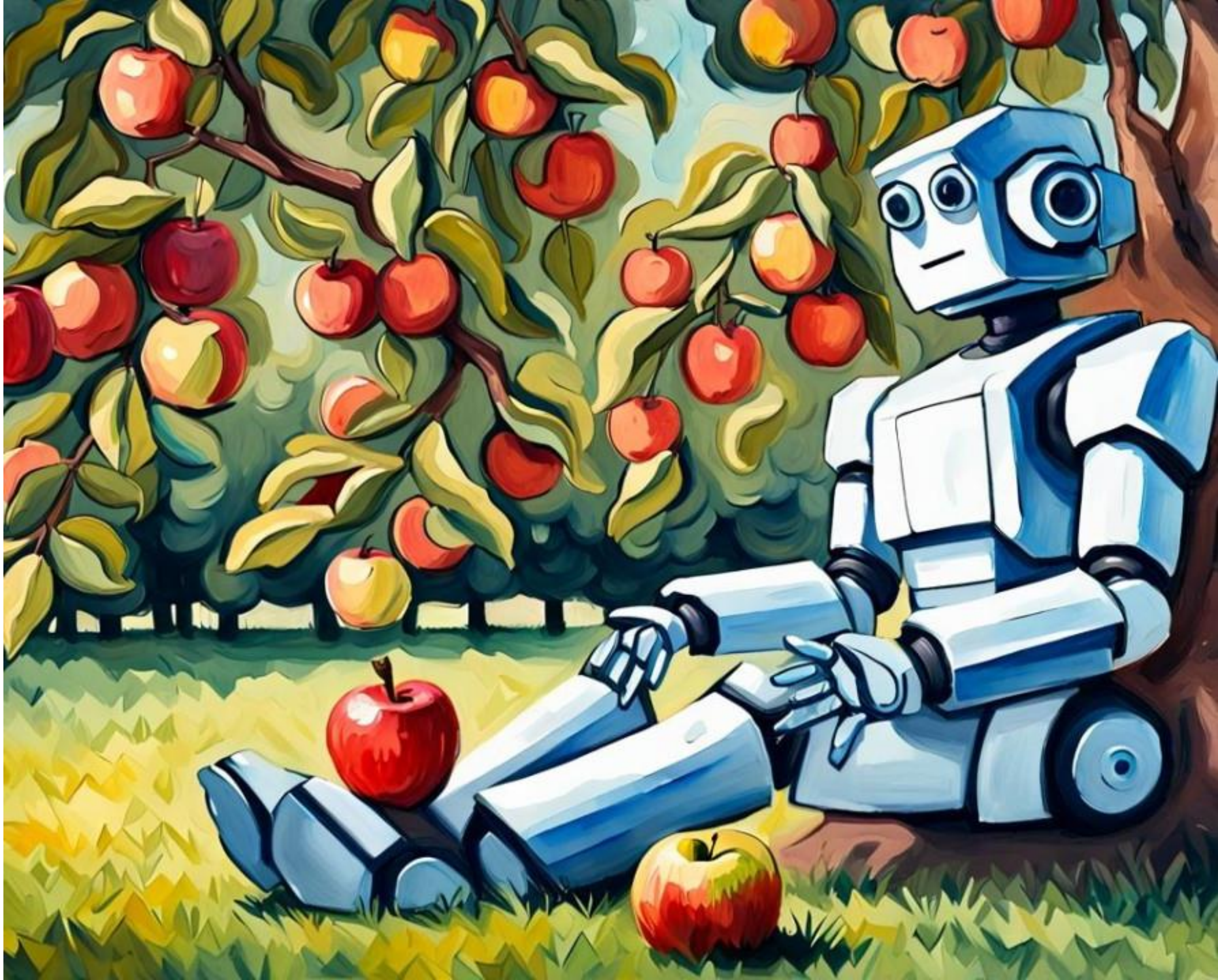
```
class SimpleNN(nn.Module):
...
model = SimpleNN().to(device)
```

- Move data to host

```
model.cpu()
x_cpu = torch.rand(3, 10)  # CPU tensor
output = model(x_cpu)
print(output.device)  # Output: cpu
```

If the model and data are on GPU, training is on GPU (without code modification)

Profiling



Profiling

- **GPU profiling** is the process of analyzing how a **Graphics Processing Unit (GPU)** executes workloads, including computation, memory usage, and efficiency. This helps in identifying bottlenecks, optimizing performance, and improving parallel execution in GPU-accelerated applications.

Types of Code Profiling

- CPU Profiling – Measures how much time each function takes.
- Memory Profiling – Tracks memory allocation to detect leaks.
- Line-by-Line Profiling – Analyzes execution at the line level.
- Real-Time Profiling – Monitors performance while the program runs.

Why Profile a GPU

- **Optimizing performance** – Identify slow code sections and reduce execution time.
- **Memory analysis** – Detect excessive GPU memory usage and optimize data transfers.
- **Kernel execution analysis** – Evaluate how GPU compute kernels are executed.
- **Improving parallelism** – Balance workloads across GPU cores.
- **Reducing power consumption** – Optimize energy efficiency in HPC and AI workloads.

nvidia-smi

nvidia-smi is a command-line tool used to monitor and manage NVIDIA GPUs. It provides real-time information on GPU utilization, memory usage, temperature, power consumption, and active processes.

```
nvidia-smi
```

```
+---------------------------------------------------------------------------------------+
| NVIDIA-SMI 535.216.03             Driver Version: 535.216.03   CUDA Version: 12.2     |
|-----------------------------------------+----------------------+----------------------+
| 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 V100-PCIE-32GB           On  | 00000000:12:00.0 Off |                    0 |
| N/A   27C    P0              24W / 250W |      0MiB / 32768MiB |      0%      Default |
|                                         |                      |                  N/A |
+-----------------------------------------+----------------------+----------------------+

+---------------------------------------------------------------------------------------+
| Processes:                                                                            |
|  GPU   GI   CI        PID   Type   Process name                            GPU Memory |
|        ID   ID                                                             Usage      |
|=======================================================================================|
|  No running processes found                                                           |
+---------------------------------------------------------------------------------------+
```

nvidia-smi

```
nvidia-smi pmon
```

Shows which **processes are using the GPU**, including **PID, memory usage, and compute mode**.

```
# gpu    pid  type   sm   mem   enc   dec   command
# Idx      #   C/G    %     %     %     %   name
    0      -     -    -     -     -     -   -
    0      -     -    -     -     -     -   -
```

```
nvidia-smi --query-gpu=memory.total,memory.used,memory.free --format=csv
```

```
memory.total [MiB], memory.used [MiB], memory.free [MiB]
32768 MiB, 0 MiB, 32501 MiB
```

nsys

Nsight Systems captures **CPU-GPU interactions**, **kernel launches**, and **API calls**.

- Run your code in profiler

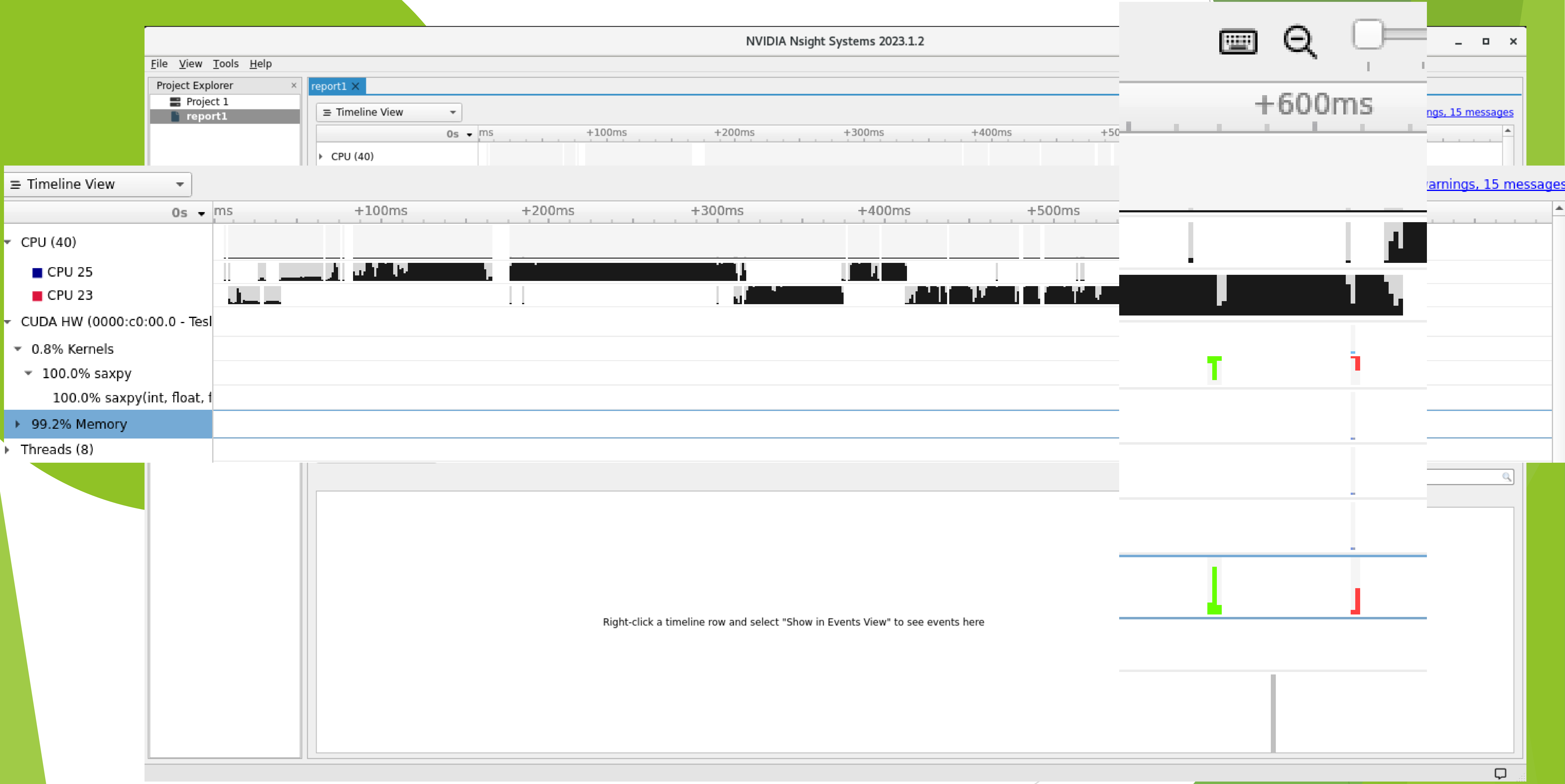
```
nsys profile -o output_report ./my_cuda_program
```

- Analize profile
  In command line

```
nsys stats output_report.qdrep
```

  with GUI

```
nsys-ui output_report.qdrep
```



NVIDIA Nsight Systems 2023.1.2
File View Tools Help
Project Explorer
Project 1
report1
report1
Timeline View
0s
ms
+100ms
+200ms
+300ms
+400ms
+500ms
+600ms
warnings, 15 messages
CPU (40)
CPU 25
CPU 23
CUDA HW (0000:c0:00.0 - Tesl
0.8% Kernels
100.0% saxpy
100.0% saxpy(int, float, f
99.2% Memory
Threads (8)
Right-click a timeline row and select "Show in Events View" to see events here

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
55.5	300,246,951	18	16,680,386.2	6,189,344.5	1,727	100,544,903	30,257,271.8	poll
42.1	227,393,986	753	301,984.0	53,082.0	1,105	55,214,590	2,650,216.0	ioctl
0.7	3,626,566	48	75,553.5	8,183.0	6,172	2,707,020	388,478.1	mmap64
0.5	2,667,418	39	68,395.3	4,414.0	2,453	805,909	196,644.6	fopen
0.4	2,358,455	15	157,230.3	53,370.0	1,276	1,359,476	364,633.7	write
0.1	742,906	88	8,442.1	7,275.0	1,751	61,091	7,492.9	open64
0.1	460,356	6	76,726.0	8,158.0	2,801	418,019	167,275.0	fread
0.1	358,231	4	89,557.8	1,583.5	177	354,887	176,891.1	fwrite
0.1	333,743	6	55,623.8	21,508.0	16,178	208,662	75,973.3	sem_ti
0.0	258,949	81	3,196.9	1,094.0	799	52,993	8,999.9	fcntl
0.0	168,156	15	11,210.4	4,269.0	2,254	73,394	18,034.9	mmap
0.0	96,452	32	3,014.1	1,986.0	1,413	17,532	3,373.5	fclose
0.0	78,524	47	1,670.7	51.0	42	76,080	11,089.7	fgets
0.0	42,606	7	6,086.6	6,235.0	2,215	8,695	2,350.9	open
0.0	27,468	18	1,526.0	1,316.5	979	3,377	692.2	read
0.0	23,321	5	4,664.2	4,200.0	3,103	6,583	1,480.1	munmap
0.0	15,491	1	15,491.0	15,491.0	15,491	15,491	0.0	fflush

…

CUDA API Summary

Time(%)	Total Time(ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
86.2	250,944,994	2	125,472,497.0	125,472,497.0	207,785	250,737,209	177,151,054.6	cudaMalloc
12.5	36,241,241	1	36,241,241.0	36,241,241.0	36,241,241	36,241,241	0.0	udaLaunchKernel
1.0	2,941,159	3	980,386.3	968,350.0	942,229	1,030,580	45,388.7	cudaMemcpy
0.3	816,524	2	408,262.0	408,262.0	209,410	607,114	281,219.2	cudaFree
0.0	11,485	1	11,485.0	11,485.0	11,485	11,485	0.0	cudaDeviceReset

CUDA GPU Kernel Summary

Time (%)	Total Time (ns)	Instances	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Name
100.0	17,440	1	17,440.0	17,440.0	17,440	17,440	0.0	saxpy(…)

CUDA GPU MemOps Summary

Time (%)	Total Time (ns)	Count	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDev (ns)	Operation
69.8	1,487,672	2	743,836.0	743,836.0	740,092	747,580	5,294.8	[CUDA memcpy HtoD]
30.2	643,612	1	643,612.0	643,612.0	643,612	643,612	0.0	[CUDA memcpy DtoH]

nsys

- For multi-GPU jobs use:

```
mpirun nsys profile  -o report.%q{SLURM_PROCID}./a.out
```

  what creates #processes of report files

- You can checek profile from each processe

```
nsys stats report.9.nsys-rep
```

ncu

- Nsight Compute provides in-depth GPU kernel profiling, including warp efficiency, memory utilization, and execution timeline.

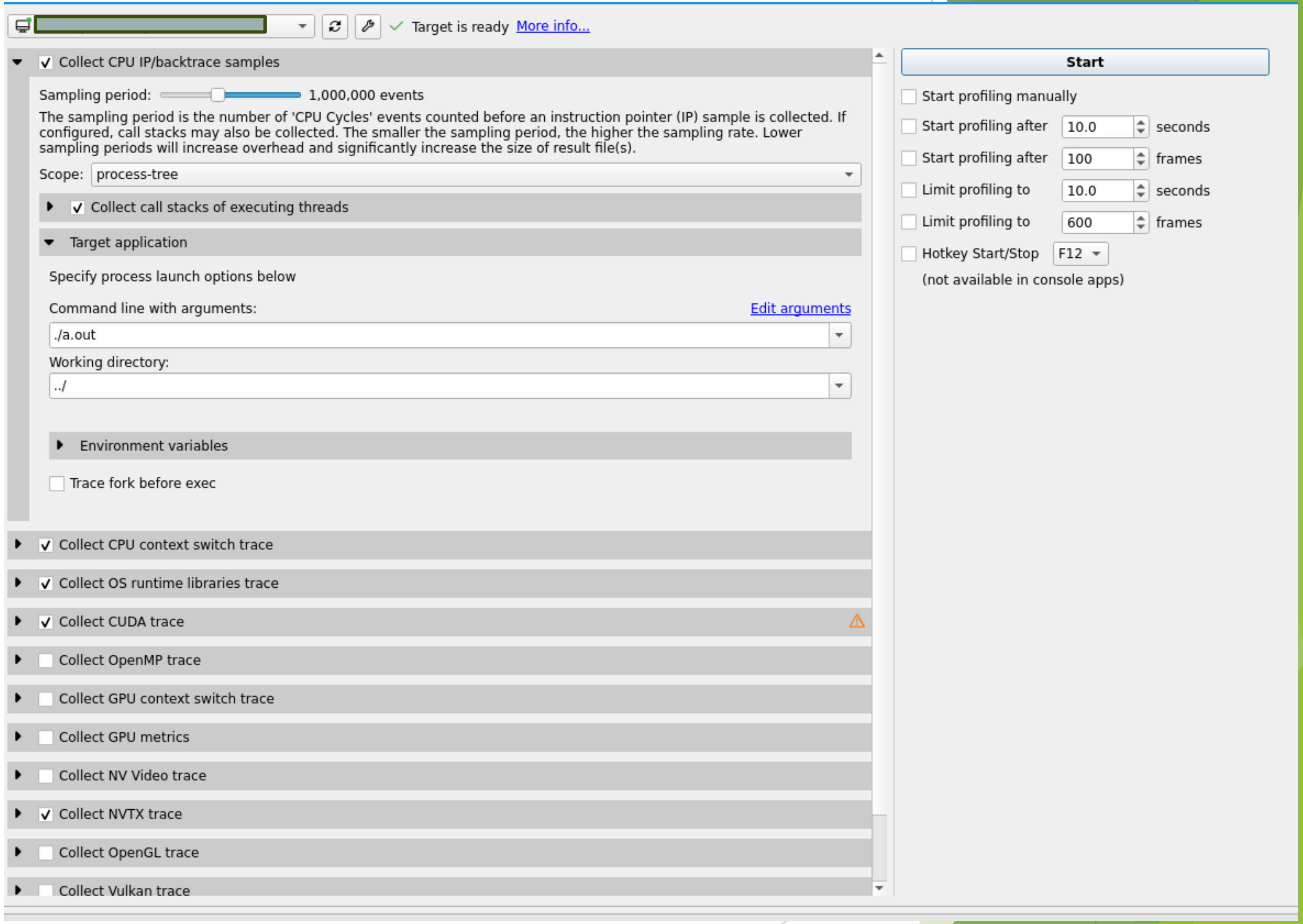
```
ncu ./my_cuda_program
```

Needs sudo

```
--------------------------------------------------------------------
| Metric                                | Value                    |
--------------------------------------------------------------------
| Kernel Execution Time                 | 5.32 ms                  |
| Registers per Thread                  | 32                       |
| Shared Memory per Block               | 16 KB                    |
| Global Memory Throughput              | 200 GB/s                 |
| L1 Cache Hit Rate                     | 78.5%                    |
| L2 Cache Hit Rate                     | 65.2%                    |
| Warp Execution Efficiency             | 90.3%                    |
| Branch Efficiency                     | 96.7%                    |
| DRAM Read Transactions                | 5,400                    |
| DRAM Write Transactions               | 3,200                    |
| Compute Throughput (FP32)             | 1.5 TFLOPS               |
--------------------------------------------------------------------
```

nsight-sys

```
nsight-sys
```




```
nsight-sys --capture-start full --target-app ./a.out
```

cuda-memcheck

Detects **memory leaks, race conditions, and uninitialized memory** in CUDA programs.

```
cuda-memcheck ./my_cuda_program
```

```
========= CUDA-MEMCHECK
========= No errors were detected
```

```
========= CUDA-MEMCHECK
========= Invalid __global__ read of size 4
=========     at 0x000000000400c00 in matrixMulKernel
=========     by thread (1,0,0) in block (0,0,0)
=========     Address 0x7f9b8a000000 is out of bounds
========= ERROR SUMMARY: 1 error
```

cuda-memcheck is not distributed with CUDA 12 nvprof is a command-line profiler that measures CUDA kernel execution times and memory usage.

```
nvprof ./my_cuda_program
```

Type	Time(%)	Time	Calls	Avg	Min	Max	Name
GPU activities:	73.14%	1.4454ms	2	722.70us	720.44us	724.95us	[CUDA memcpy HtoD]
	25.98%	513.43us	1	513.43us	513.43us	513.43us	[CUDA memcpy DtoH]
	0.88%	17.472us	1	17.472us	17.472us	17.472us	saxpy(int, float,…)
API calls:	96.67%	159.36ms	2	79.682ms	196.68us	159.17ms	cudaMalloc
	1.62%	2.6646ms	3	888.19us	840.05us	929.20us	cudaMemcpy
	0.51%	840.79us	2	420.39us	204.79us	635.99us	cudaFree
	0.42%	693.56us	1	693.56us	693.56us	693.56us	cudaLaunchKernel
	0.41%	680.32us	1	680.32us	680.32us	680.32us	cudaDeviceReset
	0.34%	564.28us	114	4.9490us	151ns	274.87us	cuDeviceGetAttribute
	0.01%	22.612us	1	22.612us	22.612us	22.612us	cuDeviceGetName
	0.01%	10.708us	1	10.708us	10.708us	10.708us	cuDeviceGetPCIBusId
	0.01%	8.3000us	1	8.3000us	8.3000us	8.3000us	cuDeviceTotalMem
	0.00%	2.0110us	3	670ns	183ns	1.6280us	cuDeviceGetCount
	0.00%	1.6310us	2	815ns	621ns	1.0100us	cudaGetLastError
	0.00%	1.3190us	2	659ns	601ns	718ns	cudaGetErrorString
	0.00%	865ns	2	432ns	179ns	686ns	cuDeviceGet
	0.00%	404ns	1	404ns	404ns	404ns	cuDeviceGetUuid
	0.00%	301ns	1				

Does not work on compute capability > 8.0

nvvp

Does not work on compute capability > 8.0