Introduction to GPUs and CUDA programming

RA Stringer

Table of contents

Preface

This short course aims to equip readers and participants with an understanding of GPU architecture and considerations for programming accelerated workloads effectively.

In an era where computational demands are soaring for training large machine learning models to rendering complex graphics, understanding GPU accelerators and how to program them effectively is an essential skill.

In this short course we will cover just enough to build a strong awareness of how accelerators work and how to approach their parallel programming paradigm in C and Python.

Topics we will cover include:

- Parallel vs sequential execution
- Thread hierarchy and organization
- CUDA kernel functions: threads, blocks and grids
- Machine learning performance optimization

If you have requests or suggestions, please submit a pull request here.

1 Getting to know the device and platform

GPUs are designed to tackle tasks that can be expressed as data-parallel computations, such as:

- genomics
- data analytics
- rendering pixels and verticies in graphics
- video encoding and decoding
- arithmetic operations eg matrix multiplications for neural networks

When getting to know the device, it can be helpful to have a quick look at PTX, SASS, warps, cooperative groups, Tensor Cores and memory hierarchy.

1.1 Abstractions

CUDA, or "Compute Unite Device Architecture" as it was introduced in 2006, is a parallel computing platform and programming model that uses the parallel engine in NVIDIA GPUs to solve computational tasks.

There are three principal abstractions:

- hierarchy of thread groups
- shared memories
- barrier synchronization

Basically CUDA allows developers to partition problems into sub-problems that can be solved by *threads* running in parallel, in *blocks*. Threads all run the same code, and an ID for each thread allows access to memory addresses and control decisions.

Threads are arranged as a *grid* of *thread blocks*.

1.2 PTX

PTX is a low-level, *parallel thread execution* virtual machine and instruction set architecture (ISA). In other words, it is a paradigm that leverages the GPU as a data-parallel computing device.

1.2.1 Programming model

PTX's programming model is parallel: it specifies the execution of a given thread of a parallel thread array. A CTA, or *cooperative thread array*, is an array of threads that execute a kernel concurrently or in parallel.

1.3 SASS

SASS is the low-level assembly language that compiles to binary microcode, which executes natively on NVIDIA GPUs.

1.4 High level architecture

GPUs have highly parallel processor architecture, comprising processing elements and memory hierarchy. Streaming processors do work on data, and that data and code are accessed from the high bandwidth memory (HMB3 in the diagram) via the L2 cache.

The A100 GPU, for example, has 108 SMs, a 40MB L2 cache, and up to 2039 GB/s bandwidth from 80GB of HBM2 memory.

NVLink Network Interconnect enables GPU-to-GPU communication among up to 256 GPUs across multiple compute nodes.

1.5 Streaming multiprocessors

Each Streaming Multiprocessor has a set of execution units, a reguster file and some shared memory.

We also notice the *warp scheduler* - this is a basic unit of execution and a collection of threads. Typically these are groups of 32 threads, which are executed together by a SM.

Tensor Cores are specialized units focused on speeding up deep learning workloads. They accel at mixed-precision matrix multiply and gradient accumulation calculations.

1.6 TeraFlops

It's worth familiarizing ourselves with TFLOPS, which stands for Trillion Floating Point Operations Per Second. This is commonly used to measure the performance of GPUs.

1 TFLOP = 1 trillion floating point calculations per second (what's a floating point? Just a number with a decimal eg 1.2 or 12.3456)

The H100 with SXM% board form-factor can perform 133.8 TFLOPs on FP16 inputs. FP16 just means *half precision*, or 1 bit for sign (+, -), 5 bits for the exponent, and 10 bits for decimal precision. This is a very popular format for AI training and inference, since a minimal drop in accuracy also means better speed and memory efficiency.

2 Getting started with CUDA

In this notebook, we dive into basic CUDA programming in C. If you don't know C well, don't worry, the code is straightforward with a focus on the CUDA considerations. Doing the exercises and following the examples can help with low-level understanding, which can often be abstracted away by the equivalent Python libraries.

2.1 Resources

We can do all exercises on the free `T4` GPU on Colab.

Let's check we have the Nvidia CUDA Compiler Driver (NVCC) installed:

```
!nvcc --version
```

```
!pip install nvcc4jupyter
```

```
%load_ext nvcc4jupyter
```

2.2 Writing and running C code in a Colab

Thanks to the `nvcc4jupyter` extension, we can run C/C++ code from our notebook cells.

Simply annotate each code cell with `%%cuda` at the top.

Syntax checking may mean a lot of read and yellow lines on our code, since it's focused on Python code, so it's best to turn this feature off: Settings -> Editor -> Code diagnostics -> None.

2.3 Hello, World!

```
%%cuda

#include <stdio.h>

void hello()
{
  printf("Hello from the CPU.\n");
}

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

2.4 and from the GPU

Let's adjust the code to make it run on the GPU.

We will need to annotate functions with

`__global__`

and synchronize our code on the completion of the kernel using the method

`cudaDeviceSynchronize();`

```
%%cuda

#include <stdio.h>

__global__ void helloGPU()
{
  printf("Hello from the GPU.\n");
}

int main()
{
  helloGPU<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

2.5 Blocks and threads

You may be wondering why we have triple angle brackets in the function call:

`<<<1, 1>>>`

These are required parameters for CUDA denoting the blocks and threads in which our tasks should run.

`<<< NUMBER_OF_BLOCKS, NUMBER_OF_THREADS_PER_BLOCK>>>`

Threads and blocks are fundamental for organizing parallel computation on GPUs. Threads are the smallest unit of action, each capable of running a single instance of the kernel function.

Threads are grouped into blocks, where they can cooperate and share resources.

Multiple blocks form a grid, the highest level of CUDA hierarchy.

`kernelA <<<1, 1>>>()` runs one block with a single thread so will run only once.

`kernelB <<<1, 10>>>()` runs one block with 10 threads and will run 10 times.

`kernelC <<<10, 1>>>()` runs 10 thread blocks, each with a single thread so will run 10 times.

`kernelD <<<10, 10>>>()` runs 10 blocks which each have 10 thread, so run 100 times.

The next example illustrates the use of threads for parallel action. Experiment with changing the `<<<blocks, threads>>>` and see the results.

```
%%cuda

#include <stdio.h>

__global__ void printInts()
{
  for(int i=0; i<10; i++)
  {
      printf("%d ", i);
  }
}

int main()
{
  printInts<<<1, 1>>>();
  cudaDeviceSynchronize();
}
```

2.6 Threads and indexes

Each thread has an index denoting its place in a block. Blocks also are indexed, and grouped into a grid.

There are useful methods for identifying these indexes:

`threadIdx.x` : identifies the index of the thread `blockIdx.x` : identifies the index of the block `blockDim.x` : represents the number of threads in a block

For example, here is a classical loop:

```
%%cuda

#include <stdio.h>

void loop(int n)
{
    for(int i=0; i<n; i++)
  {
      printf("This is loop cycle %d ", i);
  }
}

int main()
{
  loop(10);
}
```

We can accelerate this operation by launching the iterations in parallel, a *multi-block loop*. Let's use two blocks of threads.

Here, `blockIdx.x * blockDim.x + threadIdx.x` gives threads unique indexes in a grid.

```
%%cuda

#include <stdio.h>

__global__ void loop()
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  printf("This is loop cycle %d\n", i);
}
```

```
int main()
{

  loop<<<2, 5>>>();
  cudaDeviceSynchronize();
}
```

2.7 Allocating memory

CUDA version 6 and above has simplified memory allocation for both the CPU host and as or or many GPU devices with little additional work necessary by the developer.

C uses calls to `malloc` and `free` to allocate and liberate memory; we simply replace these with `cudaMallocManaged` and `cudaFree`.

```
%%cuda

#include <stdio.h>
#include <stdlib.h>

// CPU-only

int square(int x)
{
  return x = x * x;
}

int main()
{
    int N = 100000;
    size_t size = N * sizeof(int);
    int *a = (int *)malloc(size);

    // Fill array with numbers 1-20
    for (int i = 0; i < N; i++) {
        a[i] = i + 1;
    }

    // Square each number and print
    printf("Original -> Squared\n");
    for (int i = 0; i < N; i++) {
```

```
        int result = square(a[i]);
        printf("%2d -> %4d\n", a[i], result);
    }
    free(a);
    return 0;
}
```

2.8 Profiling

Let's create an executable with the same code so we can profile and time it. To do that, we write a file using cell magic functions, compile and run.

```
%%writefile square.c

#include <stdio.h>
#include <stdlib.h>

// CPU-only

int square(int x)
{
  return x = x * x;
}

int main()
{
    int N = 100000;
    size_t size = N * sizeof(int);
    int *a = (int *)malloc(size);

    // Fill array with numbers 1-20
    for (int i = 0; i < N; i++) {
        a[i] = i + 1;
    }

    // Square each number and print
    printf("Original -> Squared\n");
    for (int i = 0; i < N; i++) {
        int result = square(a[i]);
        printf("%2d -> %4d\n", a[i], result);
```

```
    }
    free(a);
    return 0;
}
```

```
!gcc square.c -o square
```

2.9 Time it

```
!time ./square
```

2.10 Exercise: CUDA version

Let's use `cudaMallocManaged` and `cudaFree`. Developers often prefix variables to be put on the device with `d_`, we will write `device_` to make this clear.

```
%%cuda

#include <stdio.h>
#include <stdlib.h>

// Accelerated

int N = 100000;
size_t size = N * sizeof(int);

int *device_a;

/*
 Here is our earlier square function:
 int square(int x)
{
  return x = x * x;
}
 */

// Initialize the array
```

```
void init(int *a, int N)
{
  int i;
  for (i = 0; i < N; ++i)
  {
    a[i] = i;
  }
}

/*
 Here is our earlier square function:
 int square(int x)
{
  return x = x * x;
}
How can we make this into a CUDA function?
Remember the threadIdx.x etc methods to
create a unique index for each thread across all blocks
 */

__global__ void square_kernel(int *device_a, int n)
{
    // initialize an index variable
    // Your code here;
    if (idx < n) {
        // square the device array at the index
        // Your code here
    }
}

// Use `a` on the CPU and/or on any GPU in the accelerated system.

int main() {
    // Use cudaMallocManaged to allocate memory for the device array
    // and the size variable
    // Your code here

    // Initialize the array
    for (int i = 0; i < N; i++) {
        device_a[i] = i + 1;  // Values will be 1, 2, 3, ..., 10
    }
```

```
    int threadsPerBlock = 256;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    square_kernel<<<blocksPerGrid, threadsPerBlock>>>(device_a, N);

    cudaDeviceSynchronize();

    printf("Squared array:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", device_a[i]);
    }

    cudaFree(device_a);

    return 0;
}
```

2.11 Solution

```
%%cuda

#include <stdio.h>
#include <stdlib.h>

// Accelerated

int N = 100000;
size_t size = N * sizeof(int);

int *device_a;

__global__ void square_kernel(int *device_a, int n)
{
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        device_a[idx] = device_a[idx] * device_a[idx];
    }
}

// Use `a` on the CPU and/or on any GPU in the accelerated system.
```

```
int main() {
    // Note the address of `a` is passed as first argument.
    cudaMallocManaged(&device_a, size);

    // Initialize the array
    for (int i = 0; i < N; i++) {
        device_a[i] = i + 1;  // Values will be 1, 2, 3, ..., 10
    }

    int threadsPerBlock = 8;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    square_kernel<<<blocksPerGrid, threadsPerBlock>>>(device_a, N);

    cudaDeviceSynchronize();

    printf("Squared array:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", device_a[i]);
    }

    cudaFree(device_a);

    return 0;
}
```

Let's take the same approach to time our CUDA equivalent `square` function.

```
%%writefile square.cu

#include <stdio.h>
#include <stdlib.h>

// Accelerated

int N = 100000;
size_t size = N * sizeof(int);

int *device_a;

__global__ void square_kernel(int *device_a, int n)
{
```

```
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < n) {
        device_a[idx] = device_a[idx] * device_a[idx];
    }
}

// Use `a` on the CPU and/or on any GPU in the accelerated system.

int main() {
    // Note the address of `a` is passed as first argument.
    cudaMallocManaged(&device_a, size);

    // Initialize the array
    for (int i = 0; i < N; i++) {
        device_a[i] = i + 1;  // Values will be 1, 2, 3, ..., 10
    }

    int threadsPerBlock = 8;
    int blocksPerGrid = (N + threadsPerBlock - 1) / threadsPerBlock;
    square_kernel<<<blocksPerGrid, threadsPerBlock>>>(device_a, N);

    cudaDeviceSynchronize();

    printf("Squared array:\n");
    for (int i = 0; i < N; i++) {
        printf("%d ", device_a[i]);
    }

    cudaFree(device_a);

    return 0;
}
```

```
!nvcc -o cuda_square square.cu
```

```
!nvprof ./cuda_square
```

```
!time ./cuda_square
```

2.12 Grid stride

Let's remind ourselves: grids are the highest level of the GPU hierarchy:

Threads -> Blocks -> Grids

Often in CUDA programming, the number of threads in a grid is smaller than the data upon which we operate.

If we have an array of 100 elements to do something with, and a grid of 25 threads, each grid will have to perform computation 4 times.

A thread at index 20 in the grid would:

- Perform its operation (eg squaring the element) on element 20 of the array
- Increment its index by 25, the size of the grid, to 45
- Perform its operation on element 45 of the array
- Increment its index by 25, to 70
- Perform its operation on element 70 of the array
- Increment its index by 25, to 95
- Perform its operation on element 95 of the array
- Stop its work, since 120 is out of range for the array

We can use the CUDA variable `gridDim.x` to calculate the number of blocks in the grid. Then we calculate the number of threads in each block, using

```
gridDim.x * blockDim.x
```

Here is a grid stride loop in a kernel:

```
__global__ void kernel(int *a, int N)
{
  int indexInGrid = threadIdx.x + blockIdx.x * blockDim.x;
  int gridStride = gridDim.x * blockDim.x;

  for (int i = indexInGrid; i < N; i += gridStride)
  {
    // do something to a[i];
  }
}
```

2.13 Exercise

For this exercise, we use the `%%writefile` magic function to create a file so we can then generate an outputs for a visualization, rather than running it inline with `%%cuda`.

```
%%writefile grid_stride.cu

#include <stdio.h>

void init(int *a, int N)
{
  int i;
  for (i = 0; i < N; ++i)
  {
    a[i] = i;
  }
}

__global__
void squareElements(int *a, int N)
{

  /*
   * Use a grid-stride loop to ensure each thread works
   * on more than one array element
   */

  int idx = // Your code here
  int stride = // Your code here

  for (int i = idx; i < N; i += stride)
  {
    int old_value = a[i];
    a[i] *= i;
    printf("Thread %d (Block %d, Thread in Block %d) processing element %d. Old value: %d, N
           idx, blockIdx.x, threadIdx.x, i, old_value, a[i]);
  }
}

bool checkElementsSquared(int *a, int N)
{
  int i;
```

```
  for (i = 0; i < N; ++i)
  {
    if (a[i] != i*i) return false;
  }
  return true;
}

int main()
{
  int N = 500;
  int *a;

  size_t size = N * sizeof(int);

  // Use cudaMallocManaged to allocate memory for the array
  // and the size variable
  // Your code her

  init(a, N);

  // The size of this grid is 356 (32 x 8)
  size_t threads_per_block = 32;
  size_t number_of_blocks = 8;

  squareElements<<<number_of_blocks, threads_per_block>>>(a, N);
  cudaDeviceSynchronize();

  bool areSquared = checkElementsSquared(a, N);
  printf("All elements were doubled? %s\n", areSquared ? "TRUE" : "FALSE");

  cudaFree(a);
}
```

2.14 Solution

```
%%writefile grid_stride.cu

#include <stdio.h>
```

```
void init(int *a, int N)
{
  int i;
  for (i = 0; i < N; ++i)
  {
    a[i] = i;
  }
}

__global__
void squareElements(int *a, int N)
{

  /*
   * Use a grid-stride loop to ensure each thread works
   * on more than one array element
   */

  int idx = blockIdx.x * blockDim.x + threadIdx.x;
  int stride = gridDim.x * blockDim.x;

  for (int i = idx; i < N; i += stride)
  {
    int old_value = a[i];
    a[i] *= i;
    printf("Thread %d (Block %d, Thread in Block %d) processing element %d. Old value: %d, Ne
           idx, blockIdx.x, threadIdx.x, i, old_value, a[i]);
  }
}

bool checkElementsSquared(int *a, int N)
{
  int i;
  for (i = 0; i < N; ++i)
  {
    if (a[i] != i*i) return false;
  }
  return true;
}

int main()
{
```

```
  int N = 500;
  int *a;

  size_t size = N * sizeof(int);
  cudaMallocManaged(&a, size);

  init(a, N);

  // The size of this grid is 356 (32 x 8)
  size_t threads_per_block = 32;
  size_t number_of_blocks = 8;

  squareElements<<<number_of_blocks, threads_per_block>>>(a, N);
  cudaDeviceSynchronize();

  bool areSquared = checkElementsSquared(a, N);
  printf("All elements were doubled? %s\n", areSquared ? "TRUE" : "FALSE");

  cudaFree(a);
}
```

```
!nvcc -o grid_stride grid_stride.cu
!./grid_stride > output.txt
```

2.15 Visualizing the grid stride

```
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import re
from google.colab import files
%matplotlib inline

def parse_line(line):
    pattern = r"Thread (\d+) \(Block (\d+), Thread in Block (\d+)\) processing element (\d+)"
    match = re.search(pattern, line)
    if match:
        return {
            'Thread ID': int(match.group(1)),
```

```
            'Block ID': int(match.group(2)),
            'Thread in Block': int(match.group(3)),
            'Element Index': int(match.group(4)),
            'Old Value': int(match.group(5)),
            'New Value': int(match.group(6))
        }
    return None

# Read the output file and parse it into a DataFrame
data = []
print("Reading file...")
with open('output.txt', 'r') as f:
    content = f.read()
    print(f"File content (first 500 characters):\n{content[:500]}")

    # Use regex to find all matches in the entire content
    pattern = r"Thread (\d+) \(Block (\d+), Thread in Block (\d+)\) processing element (\d+)"
    matches = re.finditer(pattern, content)

    for i, match in enumerate(matches, 1):
        data.append({
            'Thread ID': int(match.group(1)),
            'Block ID': int(match.group(2)),
            'Thread in Block': int(match.group(3)),
            'Element Index': int(match.group(4)),
            'Old Value': int(match.group(5)),
            'New Value': int(match.group(6))
        })
        if i % 100 == 0:
            print(f"Processed {i} matches...")

print(f"Number of parsed data points: {len(data)}")

df = pd.DataFrame(data)

# Print DataFrame info for debugging
print("\nDataFrame Info:")
print(df.info())

print("\nFirst few rows of the DataFrame:")
print(df.head())
```

```
if df.empty:
    print("The DataFrame is empty. No visualizations will be created.")
else:
    # Create a scatter plot
    plt.figure(figsize=(12, 8))
    sns.scatterplot(data=df, x='Element Index', y='Thread ID', hue='Block ID', palette='viri
    plt.title('Grid Stride Pattern Visualization')
    plt.xlabel('Array Element Index')
    plt.ylabel('Thread ID')
    plt.legend(title='Block ID', bbox_to_anchor=(1.05, 1), loc='upper left')
    plt.tight_layout()
    plt.show()

    # Create a heatmap to show the distribution of work across threads and blocks
    plt.figure(figsize=(12, 8))
    heatmap_data = df.pivot_table(values='Element Index', index='Block ID', columns='Thread
    sns.heatmap(heatmap_data, cmap='YlOrRd', annot=True, fmt='d', cbar_kws={'label': 'Number
    plt.title('Distribution of Work Across Threads and Blocks')
    plt.xlabel('Thread in Block')
    plt.ylabel('Block ID')
    plt.tight_layout()
    plt.show()

    # Calculate and print the stride
    stride = df.groupby('Thread ID')['Element Index'].diff().dropna().mode().iloc[0]
    print(f"\nStride (most common difference between consecutive elements for a thread): {str

print("Script execution completed.")
```

3 Python and CUDA

One of the most popular libraries for writing CUDA code in Python is *Numba*, a just-in-time, type-specific, function compiler that runs on either CPU or GPU.

There is also pyCUDA, which requires writing C code within Python and generally requires more modifications than Numba. We will focus on Numba since it is easier to begin with.

```
from numba import jit, cuda
import math
import numpy as np

@cuda.jit
def multiply_kernel(x, y, out):
    i = cuda.grid(1)
    if i < x.shape[0]:
        # Perform the addition
        result = x[i] * y[i]
        out[i] = result
```

```
n = 3200
x = np.arange(n).astype(np.int32)
print(x)
```

```
[   0    1    2 ... 3197 3198 3199]
```

```
y = np.full_like(x, 10)
print(y)
```

```
[10 10 10 ... 10 10 10]
```

```
d_x = cuda.to_device(x)
d_y = cuda.to_device(y)
d_out = cuda.device_array_like(d_x)
```

```
threads_per_block = 256
# math.ceil rounds up to the nearest integer, ensuring we have enough blocks  to cover all el
blocks_per_grid = math.ceil(n / threads_per_block)
print(f"threads per block: {threads_per_block}, blocks per grid: {blocks_per_grid}")
```

```
threads per block: 256, blocks per grid: 13
```

```
multiply_kernel[blocks_per_grid, threads_per_block](d_x, d_y, d_out)
cuda.synchronize()

# Copy the result back to the host
result = d_out.copy_to_host()

print("First 10 results:", result[:10])
print("Last 10 results:", result[-10:])
```

```
First 10 results: [ 0 10 20 30 40 50 60 70 80 90]
Last 10 results: [31900 31910 31920 31930 31940 31950 31960 31970 31980 31990]
```

```
/usr/local/lib/python3.10/dist-packages/numba/cuda/dispatcher.py:536: NumbaPerformanceWarning
  warn(NumbaPerformanceWarning(msg))
```

3.1 Mandelbrot set

The Mandelbrot set is a two-dimensional, geometric representation of a fractal and can create visually stunning images.

In the following code, we see

```
c_real = min_x + (max_x - min_x) * col / width
c_imag = min_y + (max_y - min_y) * row / height
```

These lines map pixel coordinates to a point in a complex plane.

We iterate using the function `f(z) = z^2 + c`, where z and c are complex numbers.

```
x = 0  # Real part of z
y = 0  # Imaginary part of z
for i in range(max_iters):
    if x*x + y*y > 4.0:
        break
    x_new = x*x - y*y + c_real
    y = 2*x*y + c_imag
    x = x_new
```

By checking if the absolute value of z is greater than 2. If $|z| > 2$, we know it will escape to infinity.

With, `x_new = x*x - y*y + c_real` we calculate the real part of z^2 + c. For a complex number z = a + bi, the real part of z^2 is a^2 - b^2.

`y = 2*x*y + c_imag`: This calculates the imaginary part of z^2 + c.

The imaginary part of z^2 is 2ab.

Convergence Test:

The Mandelbrot set is defined as the set of points c in the complex plane for which the function f(z) = z^2 + c does not diverge when iterated from z = 0. In practice, we use a finite number of iterations (max_iters) and check if |z| <=

If the iteration reaches max_iters without |z| exceeding 2, we consider the point to be in the Mandelbrot set. The number of iterations before $|z| > 2$ determines the color of points outside the set, creating the characteristic fractal patterns.

`output[row, col] = i` stores the number of iterations it took for the point to escape – this value is used for coloring the plot.

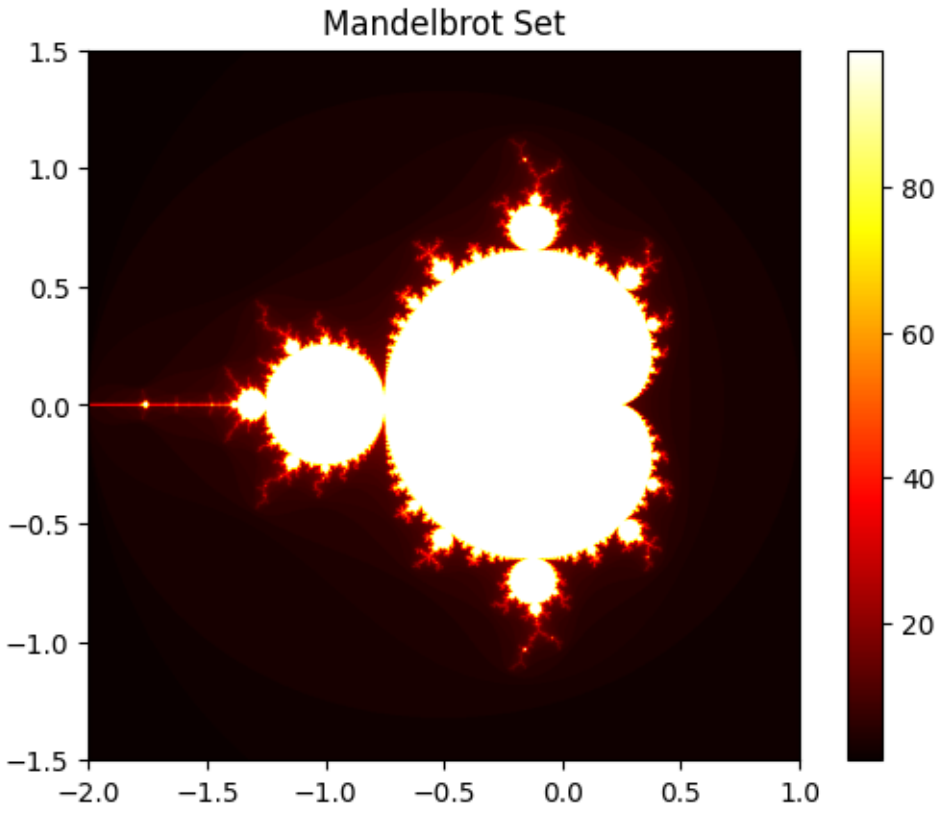
The Mandelbrot set can be useful for learning CUDA concepts since we can parallelize across threads and use memory access patterns, as well as GPU-CPU interaction. The calculations are done on the GPU, then sent back to the CPU for the display plot.

```
import numpy as np
from numba import cuda
import matplotlib.pyplot as plt

# CUDA kernel function
@cuda.jit
def mandelbrot_kernel(min_x, max_x, min_y, max_y, width, height, max_iters, output):
    # Get the 2D thread position within the grid
    row, col = cuda.grid(2)
```

```
    # Check if the thread is within the image bounds
    if row < height and col < width:
        c_real = min_x + (max_x - min_x) * col / width
        c_imag = min_y + (max_y - min_y) * row / height

        # Mandelbrot set iteration
        x = 0
        y = 0
        for i in range(max_iters):
            # Check if the point has escaped
            if x*x + y*y > 4.0:
                break
            # Update x and y
            x_new = x*x - y*y + c_real
            y = 2*x*y + c_imag
            x = x_new

        # Store the number of iterations in the output array
        output[row, col] = i

# Set up and launch kernel
def plot_mandelbrot(min_x, max_x, min_y, max_y, width, height, max_iters):
    # Output array
    output = np.zeros((height, width), dtype=np.uint32)
    # Thread block and grid dimensions
    threads_per_block = (16, 16)
    blocks_per_grid = ((width + threads_per_block[0] - 1) // threads_per_block[0],
                       (height + threads_per_block[1] - 1) // threads_per_block[1])

    # Launch kernel
    mandelbrot_kernel[blocks_per_grid, threads_per_block](
        min_x, max_x, min_y, max_y, width, height, max_iters, output
    )

    # Plot image
    plt.imshow(output, cmap='hot', extent=[min_x, max_x, min_y, max_y])
    plt.colorbar()
    plt.title('Mandelbrot Set')
    plt.show()

# Example usage
plot_mandelbrot(-2, 1, -1.5, 1.5, 1000, 1000, 100)
```

```
# Additional puzzle: Modify the code to zoom into an interesting part of the Mandelbrot set
# Hint: Adjust the min_x, max_x, min_y, max_y parameters
```

```
/usr/local/lib/python3.10/dist-packages/numba/cuda/cudadrv/devicearray.py:888: NumbaPerforma
  warn(NumbaPerformanceWarning(msg))
```



4 Profiling and optimizing PyTorch training

```
!pip install jupyterlab-nvidia-nsight
```

(Make sure you are using the free T4 runtime in Colab)

Since using GPUs is the most expensive step in ML training and inference, no small amount of work goes into optimizing their use. In the real world, very few organizations and developers work on low-level kernel optimizations. They typically work further up the stack with frameworks such as PyTorch, leaving PyTorch's optimizations to those working on its backend (which of course uses CUDA).

To give us a lens into the operations being performed on the accelerator and their efficiency in this scenario, there are a variety of profiling tools available. In this notebook, we will explore the use of Nvidia's Nsight. The software is available as a desktop application and command line tool.

4.0.1 Install Nsight tools

Since Colabs are essentially a linux-based virtual machine, we can use `apt get` to install the Nvidia tools

```
%%bash

apt update
apt install -y --no-install-recommends gnupg
echo "deb http://developer.download.nvidia.com/devtools/repos/ubuntu$(source /etc/lsb-release
apt-key adv --fetch-keys http://developer.download.nvidia.com/compute/cuda/repos/ubuntu1804/
apt update
apt install nsight-systems-cli
```

4.0.2 Check the installation

```
!nsys status -e
```

```
Timestamp counter supported: Yes

CPU Profiling Environment Check
Root privilege: enabled
Linux Kernel Paranoid Level = 2
Linux Distribution = Ubuntu
Linux Kernel Version = 6.1.85+: OK
Linux perf_event_open syscall available: OK
Sampling trigger event available: OK
Intel(c) Last Branch Record support: Not Available
CPU Profiling Environment (process-tree): OK
CPU Profiling Environment (system-wide): OK

See the product documentation at https://docs.nvidia.com/nsight-systems for more information
including information on how to set the Linux Kernel Paranoid Level.
```

4.0.3 Simple attention

Here's our basic attention mechanism that computes query, key, and value matrices to generate weighted representations of input data. The SimpleTransformer class combines this attention mechanism with layer normalization in a residual connection setup.

We will include profiling code to measure CPU and GPU performance metrics when running the model on sample input data.

```
%%writefile profiler.py

import torch
import torch.nn as nn

class SimpleAttention(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)

    def forward(self, x):
        q = self.query(x)
```

```
        k = self.key(x)
        v = self.value(x)

        attn_weights = torch.matmul(q, k.transpose(-2, -1))
        attn_weights = torch.softmax(attn_weights, dim=-1)

        return torch.matmul(attn_weights, v)

class SimpleTransformer(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.attention = SimpleAttention(embed_dim)
        self.norm = nn.LayerNorm(embed_dim)

    def forward(self, x):
        attn_output = self.attention(x)
        return self.norm(x + attn_output)

# Create a model and sample input
embed_dim = 256
seq_length = 100
batch_size = 32

model = SimpleTransformer(embed_dim, num_heads=1).cuda()
sample_input = torch.randn(batch_size, seq_length, embed_dim).cuda()

import torch.cuda.profiler as profiler

# Warm-up run
model(sample_input)

# Profile the model
with profiler.profile(activities=[profiler.ProfilerActivity.CPU, profiler.ProfilerActivity.CU
    with profiler.record_function("model_inference"):
        model(sample_input)

# Print profiling results
print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10))
```

```
Writing profiler.py
```

```
!nsys profile --stats=true python profiler.py
```

```
Collecting data...
Traceback (most recent call last):
  File "/content/profiler.py", line 46, in <module>
    with profiler.profile(activities=[profiler.ProfilerActivity.CPU, profiler.ProfilerActivit
AttributeError: module 'torch.cuda.profiler' has no attribute 'ProfilerActivity'
Generating '/tmp/nsys-report-4b28.qdstrm'
[1/8] [========================100%] report1.nsys-rep
[2/8] [========================100%] report1.sqlite
[3/8] Executing 'nvtx_sum' stats report
SKIPPED: /content/report1.sqlite does not contain NV Tools Extension (NVTX) data.
[4/8] Executing 'osrt_sum' stats report
```

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	St
76.0	1,702,896,538	29	58,720,570.3	77,869,201.0	4,006	100,150,826	44
13.2	295,663,488	1,672	176,832.2	3,329.0	1,004	15,374,049	
4.0	89,772,667	611	146,927.4	16,308.0	1,509	23,241,632	
3.9	86,647,314	5,802	14,934.0	3,006.5	1,006	13,686,743	
1.7	38,487,406	992	38,797.8	13,551.0	2,049	844,849	
0.5	10,831,409	7,221	1,500.0	1,469.0	1,000	18,716	
0.3	6,090,781	1	6,090,781.0	6,090,781.0	6,090,781	6,090,781	
0.2	3,767,157	76	49,567.9	10,444.5	3,410	2,163,110	
0.2	3,716,694	1,853	2,005.8	1,794.0	1,018	25,280	
0.0	596,671	4	149,167.8	49,473.0	35,192	462,533	
0.0	521,407	74	7,046.0	3,520.5	1,697	169,539	
0.0	333,517	20	16,675.8	9,374.5	2,751	105,415	
0.0	303,794	16	18,987.1	12,165.5	1,402	57,334	
0.0	261,847	8	32,730.9	32,683.5	23,171	39,984	
0.0	207,125	3	69,041.7	68,447.0	66,655	72,023	
0.0	142,461	70	2,035.2	1,419.5	1,079	13,265	
0.0	118,808	2	59,404.0	59,404.0	56,211	62,597	
0.0	114,648	9	12,738.7	13,780.0	7,881	21,544	
0.0	92,736	12	7,728.0	4,024.0	1,623	27,163	
0.0	85,302	15	5,686.8	4,267.0	2,028	22,918	
0.0	33,190	5	6,638.0	5,099.0	1,635	16,932	
0.0	23,712	2	11,856.0	11,856.0	8,025	15,687	
0.0	10,493	1	10,493.0	10,493.0	10,493	10,493	
0.0	8,915	1	8,915.0	8,915.0	8,915	8,915	
0.0	7,372	1	7,372.0	7,372.0	7,372	7,372	
0.0	6,383	1	6,383.0	6,383.0	6,383	6,383	

```
     0.0           5,305          1       5,305.0       5,305.0      5,305        5,305
     0.0           5,009          2       2,504.5       2,504.5      1,829        3,180
     0.0           4,778          4       1,194.5       1,195.0      1,043        1,345
     0.0           3,201          3       1,067.0       1,055.0      1,033        1,113
     0.0           2,580          1       2,580.0       2,580.0      2,580        2,580
     0.0           2,300          1       2,300.0       2,300.0      2,300        2,300
     0.0           1,609          1       1,609.0       1,609.0      1,609        1,609
     0.0           1,505          1       1,505.0       1,505.0      1,505        1,505
     0.0           1,318          1       1,318.0       1,318.0      1,318        1,318

[5/8] Executing 'cuda_api_sum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)      Med (ns)    Min (ns)    Max (ns)   Sto
 --------  ---------------  ---------  ------------  ------------  ---------  ----------  ---
     59.0      133,942,988          8  16,742,873.5      44,480.0     18,304  75,424,540  29
     19.9       45,123,527          9   5,013,725.2      21,450.0      7,039  44,149,269  14
     19.1       43,452,758          2  21,726,379.0  21,726,379.0  5,346,866  38,105,892  23
      0.8        1,898,877          6     316,479.5     216,731.0     12,598     976,777
      0.6        1,421,966         18      78,998.1         668.0        616   1,401,568
      0.3          658,905          3     219,635.0       3,212.0      2,603     653,090
      0.2          469,330      1,149         408.5         256.0        126     146,038
      0.1          182,122          9      20,235.8       7,191.0      5,845      68,490
      0.0            7,048          3       2,349.3       2,324.0      2,234       2,490
      0.0            1,790          4         447.5         300.5        252         937

[6/8] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances  Avg (ns)   Med (ns)   Min (ns)  Max (ns)  StdDev (ns)
 --------  ---------------  ---------  ---------  ---------  --------  --------  -----------
     61.8          632,402          3  210,800.7  210,491.0   209,947   211,964      1,043.5
     14.3          146,205          1  146,205.0  146,205.0   146,205   146,205          0.0
     11.2          114,973          1  114,973.0  114,973.0   114,973   114,973          0.0
      8.1           82,590          1   82,590.0   82,590.0    82,590    82,590          0.0
      3.1           32,127          1   32,127.0   32,127.0    32,127    32,127          0.0
      1.4           14,240          1   14,240.0   14,240.0    14,240    14,240          0.0

[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)  Med (ns)  Min (ns)  Max (ns)  StdDev (ns)
 --------  ---------------  -----  --------  --------  --------  --------  -----------  -----
    100.0          675,793      9  75,088.1     768.0       735   599,634    197,031.0  [CUDA

[8/8] Executing 'cuda_gpu_mem_size_sum' stats report
```

```
 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  --------------------
      4.068      9     0.452     0.001     0.001     3.277        1.067  [CUDA memcpy Host-t

Generated:
    /content/report1.nsys-rep
    /content/report1.sqlite
```

(Numbers will differ slightly each time we run these cells)

Let's analyze the "cuda_gpu_kern_sum" report, which shows the GPU kernel executions:

- `volta_sgemm_128x64_tn` (61.9% of GPU time): This is likely the matrix multiplication for computing attention weights (q * k.transpose(-2, -1)). It's using NVIDIA's optimized GEMM (General Matrix Multiplication) kernel. Typically the most compute-intensive operation in a. transformer model.
- `volta_sgemm_64x64_tn` (14.2% of GPU time): This could be another part of the attention computation, possibly the final matrix multiplication with the value matrix (attn_weights * v).
- `volta_sgemm_128x64_nn` (11.3% of GPU time): This might be the matrix multiplication in one of the linear layers (query, key, or value projection).
- `vectorized_layer_norm_kernel` (8.1% of GPU time): This corresponds to the LayerNorm operation in the SimpleTransformer class. vectorized_elementwise_kernel (3.1% of GPU time): This could be the element-wise addition in the residual connection (x + attn_output).
- `softmax_warp_forward` (1.4% of GPU time): This is the softmax operation applied to the attention weights.

The `SimpleAttention` class operations are primarily represented by items 1, 2, 3, and 6 in this list. These operations account for about 88.8% of the GPU kernel execution time, which indicates that the attention mechanism is indeed a significant part of the computation. To optimize this, we could:

- Use the optimized attention mechanism as suggested in the tutorial (torch.nn.functional.scaled_dot_produ
- Experiment with different batch sizes or sequence lengths to find the optimal configuration for your hardware.
- Consider using mixed precision (float16).

4.1 Flash Attention

Let's optimize our attention mechanism to use the `torch.nn.functional.scaled_dot_product_attention` function, optimized for GPUs. This method uses the Flash Attention algorithm when available.

```
%%writefile profiler.py

import torch
import torch.nn as nn

import torch.nn.functional as F

class OptimizedAttention(nn.Module):
    def __init__(self, embed_dim):
        super().__init__()
        self.query = nn.Linear(embed_dim, embed_dim)
        self.key = nn.Linear(embed_dim, embed_dim)
        self.value = nn.Linear(embed_dim, embed_dim)
        self.scale = embed_dim ** -0.5

    def forward(self, x):
        q = self.query(x)
        k = self.key(x)
        v = self.value(x)

        return F.scaled_dot_product_attention(q, k, v, scale=self.scale)

# Update the SimpleTransformer class to use OptimizedAttention
class OptimizedTransformer(nn.Module):
    def __init__(self, embed_dim, num_heads):
        super().__init__()
        self.attention = OptimizedAttention(embed_dim)
        self.norm = nn.LayerNorm(embed_dim)

    def forward(self, x):
        attn_output = self.attention(x)
        return self.norm(x + attn_output)

# Create a model and sample input
embed_dim = 256
seq_length = 1000
```

```
batch_size = 32

# Create a new model with the optimized attention
optimized_model = OptimizedTransformer(embed_dim, num_heads=1).cuda()
sample_input = torch.randn(batch_size, seq_length, embed_dim).cuda()

import torch.cuda.profiler as profiler

# Warm-up run
optimized_model(sample_input)

# Profile the optimized model
with profiler.profile(activities=[profiler.ProfilerActivity.CPU, profiler.ProfilerActivity.CU
    with profiler.record_function("optimized_model_inference"):
        optimized_model(sample_input)

# Print profiling results
print(prof.key_averages().table(sort_by="cuda_time_total", row_limit=10))
```

```
Overwriting profiler.py
```

```
!nsys profile --stats=true python profiler.py
```

```
Collecting data...
Traceback (most recent call last):
  File "/content/profiler.py", line 48, in <module>
    with profiler.profile(activities=[profiler.ProfilerActivity.CPU, profiler.ProfilerActivit
AttributeError: module 'torch.cuda.profiler' has no attribute 'ProfilerActivity'
Generating '/tmp/nsys-report-ed31.qdstrm'
[1/8] [========================100%] report2.nsys-rep
[2/8] [========================100%] report2.sqlite
[3/8] Executing 'nvtx_sum' stats report
SKIPPED: /content/report2.sqlite does not contain NV Tools Extension (NVTX) data.
[4/8] Executing 'osrt_sum' stats report

 Time (%)  Total Time (ns)  Num Calls    Avg (ns)     Med (ns)    Min (ns)    Max (ns)    Sto
 --------  ---------------  ---------  ------------  -----------  ---------  -----------  ---
     79.1      501,166,791         17  29,480,399.5  2,985,176.0      3,734  100,157,864  38
     11.6       73,748,312        639     115,412.1     12,665.0      1,374   18,555,420   1
      2.9       18,137,686      5,796       3,129.3      2,490.0      1,000       47,046
      1.6        9,881,832      6,366       1,552.3      1,470.0      1,000       21,190
```

1.3	8,400,908	1,104	7,609.5	2,719.5	1,007	251,640
1.2	7,379,916	992	7,439.4	6,844.5	1,883	28,734
0.8	5,069,938	1	5,069,938.0	5,069,938.0	5,069,938	5,069,938
0.6	3,616,562	76	47,586.3	10,088.5	4,364	2,222,049
0.5	3,406,279	1,852	1,839.2	1,636.5	1,070	49,459
0.1	906,478	9	100,719.8	67,874.0	41,581	396,079
0.1	354,877	74	4,795.6	3,567.5	1,586	25,689
0.0	287,427	27	10,645.4	7,683.0	2,462	61,914
0.0	252,649	8	31,581.1	32,863.0	12,821	39,628
0.0	221,124	3	73,708.0	71,618.0	68,171	81,335
0.0	157,808	16	9,863.0	8,608.0	4,446	27,809
0.0	141,700	69	2,053.6	1,431.0	1,007	14,453
0.0	123,084	16	7,692.8	5,635.5	1,063	22,553
0.0	113,446	2	56,723.0	56,723.0	52,791	60,655
0.0	80,095	15	5,339.7	4,115.0	1,789	15,904
0.0	29,997	5	5,999.4	3,918.0	1,410	16,372
0.0	25,575	2	12,787.5	12,787.5	9,492	16,083
0.0	13,661	5	2,732.2	1,858.0	1,045	6,703
0.0	12,962	1	12,962.0	12,962.0	12,962	12,962
0.0	12,710	2	6,355.0	6,355.0	1,101	11,609
0.0	10,017	1	10,017.0	10,017.0	10,017	10,017
0.0	6,291	1	6,291.0	6,291.0	6,291	6,291
0.0	4,856	1	4,856.0	4,856.0	4,856	4,856
0.0	3,791	3	1,263.7	1,188.0	1,062	1,541
0.0	3,692	3	1,230.7	1,150.0	1,078	1,464
0.0	3,620	2	1,810.0	1,810.0	1,524	2,096
0.0	2,106	1	2,106.0	2,106.0	2,106	2,106
0.0	1,780	1	1,780.0	1,780.0	1,780	1,780
0.0	1,637	1	1,637.0	1,637.0	1,637	1,637
0.0	1,376	1	1,376.0	1,376.0	1,376	1,376

[5/8] Executing 'cuda_api_sum' stats report

Time (%)	Total Time (ns)	Num Calls	Avg (ns)	Med (ns)	Min (ns)	Max (ns)	StdDe
64.9	73,600,176	10	7,360,017.6	25,209.5	13,686	29,189,994	12,03
19.6	22,201,745	9	2,466,860.6	11,600.0	3,985	15,341,866	5,31
13.2	15,018,628	2	7,509,314.0	7,509,314.0	2,117,769	12,900,859	7,62
2.0	2,217,914	13	170,608.8	172,874.0	4,544	329,658	8
0.2	222,195	1,149	193.4	165.0	89	1,779	
0.2	173,034	9	19,226.0	6,419.0	5,973	55,838	1
0.0	25,834	18	1,435.2	371.0	354	15,734	
0.0	24,330	10	2,433.0	2,015.5	1,109	6,827	

```
     0.0            4,335          3      1,445.0      1,419.0      1,209       1,707
     0.0            2,221          4        555.3        217.5        150       1,636

[6/8] Executing 'cuda_gpu_kern_sum' stats report

 Time (%)  Total Time (ns)  Instances   Avg (ns)     Med (ns)    Min (ns)   Max (ns)   StdDev
 --------  ---------------  ---------  -----------  -----------  ---------  ---------  -----
     57.5       11,818,611          4  2,954,652.8  1,712,425.5  1,701,017  6,692,743  2,492
     28.4        5,834,459          1  5,834,459.0  5,834,459.0  5,834,459  5,834,459
      5.8        1,197,893          1  1,197,893.0  1,197,893.0  1,197,893  1,197,893
      3.7          763,694          1    763,694.0    763,694.0    763,694    763,694
      2.6          535,827          2    267,913.5    267,913.5    263,226    272,601      6
      1.9          389,943          1    389,943.0    389,943.0    389,943    389,943

[7/8] Executing 'cuda_gpu_mem_time_sum' stats report

 Time (%)  Total Time (ns)  Count  Avg (ns)   Med (ns)  Min (ns)  Max (ns)   StdDev (ns)
 --------  ---------------  -----  ---------  --------  --------  ---------  -----------  ---
    100.0        6,588,201      9  732,022.3     736.0       704  6,510,028  2,166,783.8  [CU

[8/8] Executing 'cuda_gpu_mem_size_sum' stats report

 Total (MB)  Count  Avg (MB)  Med (MB)  Min (MB)  Max (MB)  StdDev (MB)           Operation
 ----------  -----  --------  --------  --------  --------  -----------  --------------------
     33.560      9     3.729     0.001     0.001    32.768       10.890  [CUDA memcpy Host-to

Generated:
    /content/report2.nsys-rep
    /content/report2.sqlite
```

(Numbers will differ slightly each time we run these cells)

Looking at the "cuda_gpu_kern_sum" report, we notice:

- volta_sgemm_128x64_tn (58.4% of GPU time, previously 61.9%):
  - We see a slight decrease in what is likely the matrix multiplication for computing attention weights. Though small on some tiny sample data, imagine these gains multiplied exponentially on real world training and inference involving text, images, video etc.
- volta_sgemm_64x64_tn (13.3%, previously 14.2%):
  - Final matrix multiplication with the value matrix.

- volta_sgemm_128x64_nn (10.6%, previously 11.3%):
  - The linear layer matrix multiplications.
- vectorized_layer_norm_kernel (7.7%, previously 8.1%):
  - This corresponds to the LayerNorm operation in the SimpleTransformer class.
- vectorized_elementwise_kernel (4.7% + 3.9% = 8.6%, previously 3.1%):
  - This now appears as two separate kernels, possibly for different elementwise operations.
- softmax_warp_forward (1.3%, previously 1.4%):
  - This is still the softmax operation applied to the attention weights.

5 Summary

In summary, we have looked at GPU architecture, threads, blocks, grids and programming with CUDA in C and Python.

To dive further into the world of accelerators, here are some suggested resources:

Nvidia courses

GPU Puzzles - very popular GitHub repo

Learn GPU Programming in Your Browser – from answer.ai, uses WebGPU Shading Language.

PyTorch CUDA backend – Look at some of the PyTorch CUDA backend code samples.