Challenge #13

1. Here are the steps for setting up the Google Collab environment to execute CUDA files.

**Enable GPU Runtime**

Go to: Runtime > Change runtime type > Hardware accelerator > GPU > Save

**Install CUDA Toolkit (e.g., CUDA 11.8):**

# Download the installer

!wget https://developer.download.nvidia.com/compute/cuda/11.8.0/local\_installers/cuda\_11.8.0\_520.61.05\_linux.run

# Make it executable

!chmod +x cuda\_11.8.0\_520.61.05\_linux.run

# Silent install (toolkit only)

!sudo ./cuda\_11.8.0\_520.61.05\_linux.run --silent –toolkit

**Add CUDA to PATH:**

import os

os.environ['PATH'] += ':/usr/local/cuda/bin'

os.environ['LD\_LIBRARY\_PATH'] = '/usr/local/cuda/lib64'

# Check nvcc version

!nvcc –version

**Compile the CUDA Code:** !nvcc -arch=sm\_70 saxpy.cu -o saxpy

**Run the Executable:** !./saxpy

If you want to use the default setup in Google Collab, use the following steps.

**Check GPU is enabled:** !nvidia-smi

If it fails or says “No devices were found,” your GPU is not active. Go to:

Runtime > Change runtime type > Hardware accelerator > GPU

Then **restart the runtime**.

**Check the version:** !nvcc --version

Later compile and execute the code.

1. Starting with the code.

Below is the code I wrote in Google Collab.

%%writefile saxpy.cu

#include <stdio.h>

#include <math.h> // for fabsf

\_\_global\_\_

void saxpy(int n, float a, float \*x, float \*y)

{

  int i = blockIdx.x\*blockDim.x + threadIdx.x;

  if (i < n) y[i] = a\*x[i] + y[i];

}

int main(void)

{

  int N = 1<<20;

  float \*x, \*y, \*d\_x, \*d\_y;

  x = (float\*)malloc(N\*sizeof(float));

  y = (float\*)malloc(N\*sizeof(float));

  cudaMalloc(&d\_x, N\*sizeof(float));

  cudaMalloc(&d\_y, N\*sizeof(float));

  for (int i = 0; i < N; i++) {

    x[i] = 1.0f;

    y[i] = 2.0f;

  }

  cudaMemcpy(d\_x, x, N\*sizeof(float), cudaMemcpyHostToDevice);

  cudaMemcpy(d\_y, y, N\*sizeof(float), cudaMemcpyHostToDevice);

  // Perform SAXPY on 1M elements

  saxpy<<<(N+255)/256, 256>>>(N, 2.0f, d\_x, d\_y);

  cudaMemcpy(y, d\_y, N\*sizeof(float), cudaMemcpyDeviceToHost);

  float maxError = 0.0f;

  for (int i = 0; i < N; i++)

    maxError = fmaxf(maxError, fabsf(y[i]-4.0f));

  printf("Max error: %f\n", maxError);

  cudaFree(d\_x);

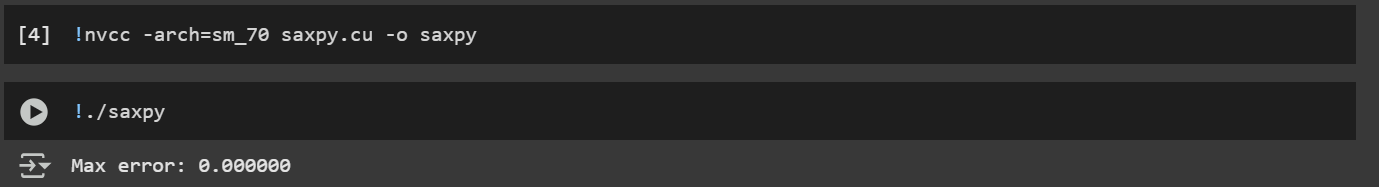
  cudaFree(d\_y);

  free(x);

  free(y);

}

After executing, I got the following output.



1. Modify the code so that you can simulate and measure the execution times for the following  
   matrix sizes: N = 215 , 216 , ... 225 (or as high as you can).  
     
   %%writefile saxpy\_debug.cu

#include <stdio.h>

#include <math.h>

#include <cuda\_runtime.h>

\_\_global\_\_

void saxpy(int n, float a, float \*x, float \*y)

{

  int i = blockIdx.x \* blockDim.x + threadIdx.x;

  if (i < n) y[i] = a \* x[i] + y[i];

}

int main(void)

{

  for (int exp = 15; exp <= 25; exp++) {

    int N = 1 << exp;

    float \*x, \*y, \*d\_x, \*d\_y;

    x = (float\*)malloc(N \* sizeof(float));

    y = (float\*)malloc(N \* sizeof(float));

    if (cudaMalloc(&d\_x, N \* sizeof(float)) != cudaSuccess ||

        cudaMalloc(&d\_y, N \* sizeof(float)) != cudaSuccess) {

      printf("CUDA malloc failed for N = %d\n", N);

      free(x); free(y);

      continue;

    }

    for (int i = 0; i < N; i++) {

      x[i] = 1.0f;

      y[i] = 2.0f;

    }

    cudaMemcpy(d\_x, x, N \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_y, y, N \* sizeof(float), cudaMemcpyHostToDevice);

    cudaEvent\_t start, stop;

    cudaEventCreate(&start);

    cudaEventCreate(&stop);

    cudaEventRecord(start);

    saxpy<<<(N + 255) / 256, 256>>>(N, 2.0f, d\_x, d\_y);

    cudaEventRecord(stop);

    // ✅ Check for kernel launch errors

    cudaError\_t err = cudaGetLastError();

    if (err != cudaSuccess) {

      printf("CUDA kernel launch failed: %s\n", cudaGetErrorString(err));

      cudaFree(d\_x); cudaFree(d\_y); free(x); free(y);

      continue;

    }

    cudaEventSynchronize(stop);

    float milliseconds = 0;

    cudaEventElapsedTime(&milliseconds, start, stop);

    cudaMemcpy(y, d\_y, N \* sizeof(float), cudaMemcpyDeviceToHost);

    float maxError = 0.0f;

    for (int i = 0; i < N; i++)

      maxError = fmaxf(maxError, fabsf(y[i] - 4.0f));

    printf("N = 2^%d (%d): Max error = %f, Time = %.3f ms\n", exp, N, maxError, milliseconds);

    // ✅ Print a few y values to confirm correctness

    printf("Sample values: y[0] = %.1f, y[1] = %.1f, y[N-1] = %.1f\n\n", y[0], y[1], y[N-1]);

    cudaFree(d\_x);

    cudaFree(d\_y);

    free(x);

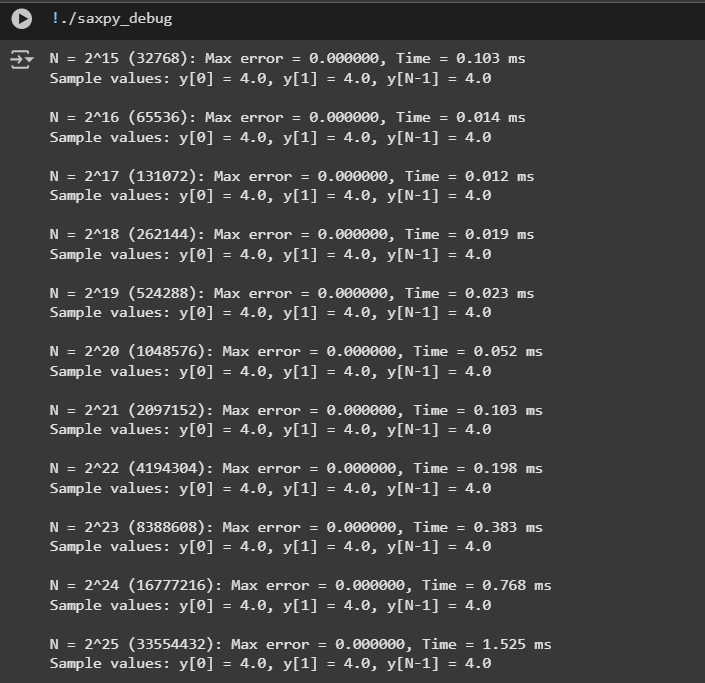
    free(y);

  }

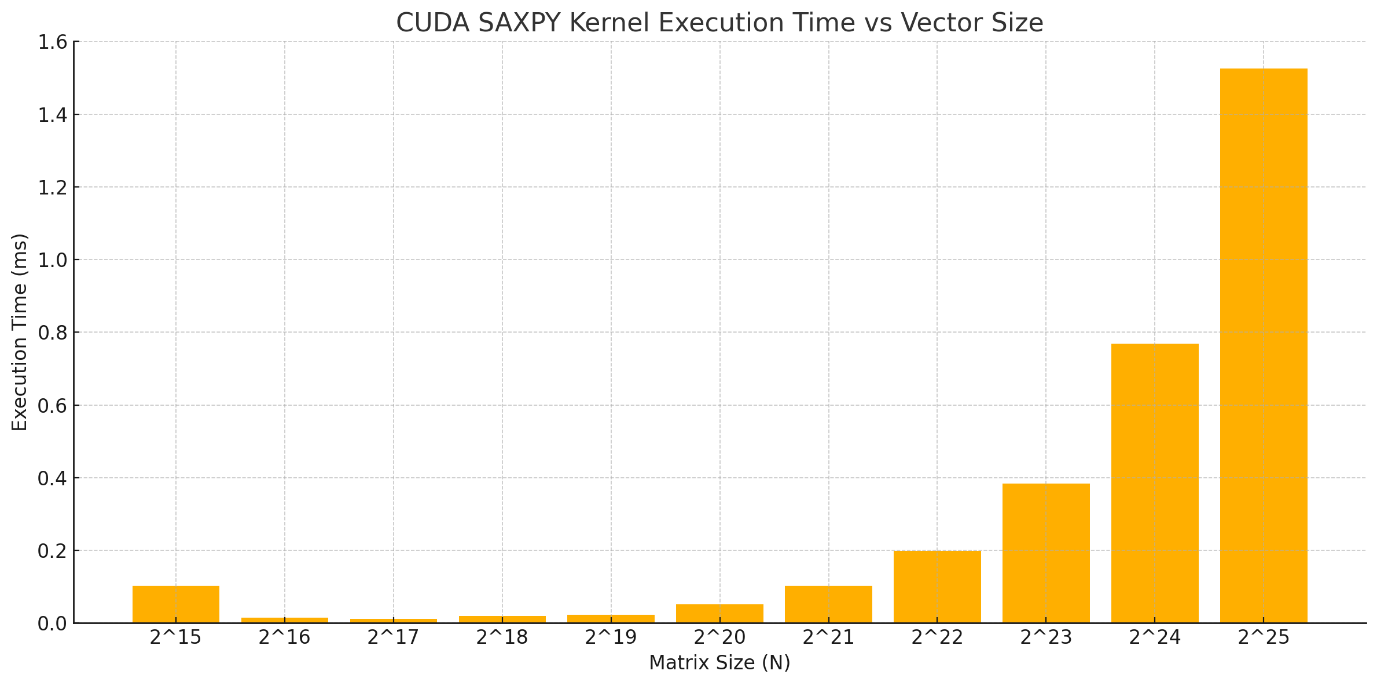
  return 0;

}

Below is the output of the above code.



1. Visualize the execution times in a bar plot. You can use a spreadsheet or Matplotlib for that. What do you observe?



Bar Plot comparing execution times with matrix size

The bar plot of the CUDA SAXPY kernel execution time versus vector size clearly illustrates that execution time increases with the input size NN, as expected from a linear time complexity operation like SAXPY. For smaller vector sizes (e.g., 2152^{15} to 2192^{19}), the execution times remain extremely low and stable, suggesting that the GPU is underutilized and the kernel launch overhead dominates the actual computation time. However, as the vector size increases beyond 2202^{20}, the execution time grows gradually, indicating that the GPU is able to distribute the workload more efficiently across its cores. This trend continues until N=225N = 2^{25}, where the kernel reaches its peak execution time around 1.5 milliseconds. Overall, the SAXPY kernel demonstrates excellent scalability and performance for large datasets, confirming that such data-parallel operations are well-suited for GPU acceleration. For smaller vectors, however, the benefits of GPU parallelism may not outweigh the overhead, making CPU execution competitive in those cases.

1. Measure the total execution time (including memory transfers and allocations and the kernel execution time only (just the GPU computation) separately. Hint: cudaEvent

%%writefile saxpy\_fancy.cu

#include <stdio.h>

#include <math.h>

#include <cuda\_runtime.h>

\_\_global\_\_

void saxpy(int n, float a, float \*x, float \*y) {

  int i = blockIdx.x \* blockDim.x + threadIdx.x;

  if (i < n) y[i] = a \* x[i] + y[i];

}

int main(void) {

  for (int exp = 15; exp <= 25; exp++) {

    int N = 1 << exp;

    float \*x, \*y, \*d\_x, \*d\_y;

    // Start total timer

    cudaEvent\_t total\_start, total\_stop;

    cudaEventCreate(&total\_start);

    cudaEventCreate(&total\_stop);

    cudaEventRecord(total\_start);

    x = (float\*)malloc(N \* sizeof(float));

    y = (float\*)malloc(N \* sizeof(float));

    if (cudaMalloc(&d\_x, N \* sizeof(float)) != cudaSuccess ||

        cudaMalloc(&d\_y, N \* sizeof(float)) != cudaSuccess) {

      printf("CUDA malloc failed for N = %d\n", N);

      free(x); free(y);

      continue;

    }

    for (int i = 0; i < N; i++) {

      x[i] = 1.0f;

      y[i] = 2.0f;

    }

    cudaMemcpy(d\_x, x, N \* sizeof(float), cudaMemcpyHostToDevice);

    cudaMemcpy(d\_y, y, N \* sizeof(float), cudaMemcpyHostToDevice);

    // Start kernel-only timer

    cudaEvent\_t kernel\_start, kernel\_stop;

    cudaEventCreate(&kernel\_start);

    cudaEventCreate(&kernel\_stop);

    cudaEventRecord(kernel\_start);

    saxpy<<<(N + 255) / 256, 256>>>(N, 2.0f, d\_x, d\_y);

    cudaEventRecord(kernel\_stop);

    cudaEventSynchronize(kernel\_stop);

    cudaMemcpy(y, d\_y, N \* sizeof(float), cudaMemcpyDeviceToHost);

    // End total timer

    cudaEventRecord(total\_stop);

    cudaEventSynchronize(total\_stop);

    // Calculate times

    float kernel\_time = 0, total\_time = 0;

    cudaEventElapsedTime(&kernel\_time, kernel\_start, kernel\_stop);

    cudaEventElapsedTime(&total\_time, total\_start, total\_stop);

    float maxError = 0.0f;

    for (int i = 0; i < N; i++)

      maxError = fmaxf(maxError, fabsf(y[i] - 4.0f));

    printf("N = 2^%d (%d): Max error = %.6f\n", exp, N, maxError);

    printf(" → Kernel-only time: %.3f ms\n", kernel\_time);

    printf(" → Total execution time: %.3f ms\n", total\_time);

    printf(" → Sample y[0] = %.1f, y[N-1] = %.1f\n\n", y[0], y[N - 1]);

    // Cleanup

    cudaFree(d\_x); cudaFree(d\_y); free(x); free(y);

    cudaEventDestroy(total\_start); cudaEventDestroy(total\_stop);

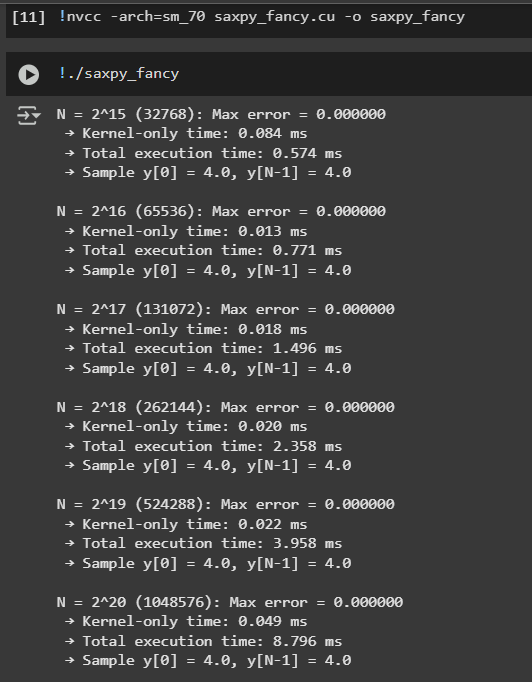
    cudaEventDestroy(kernel\_start); cudaEventDestroy(kernel\_stop);

  }

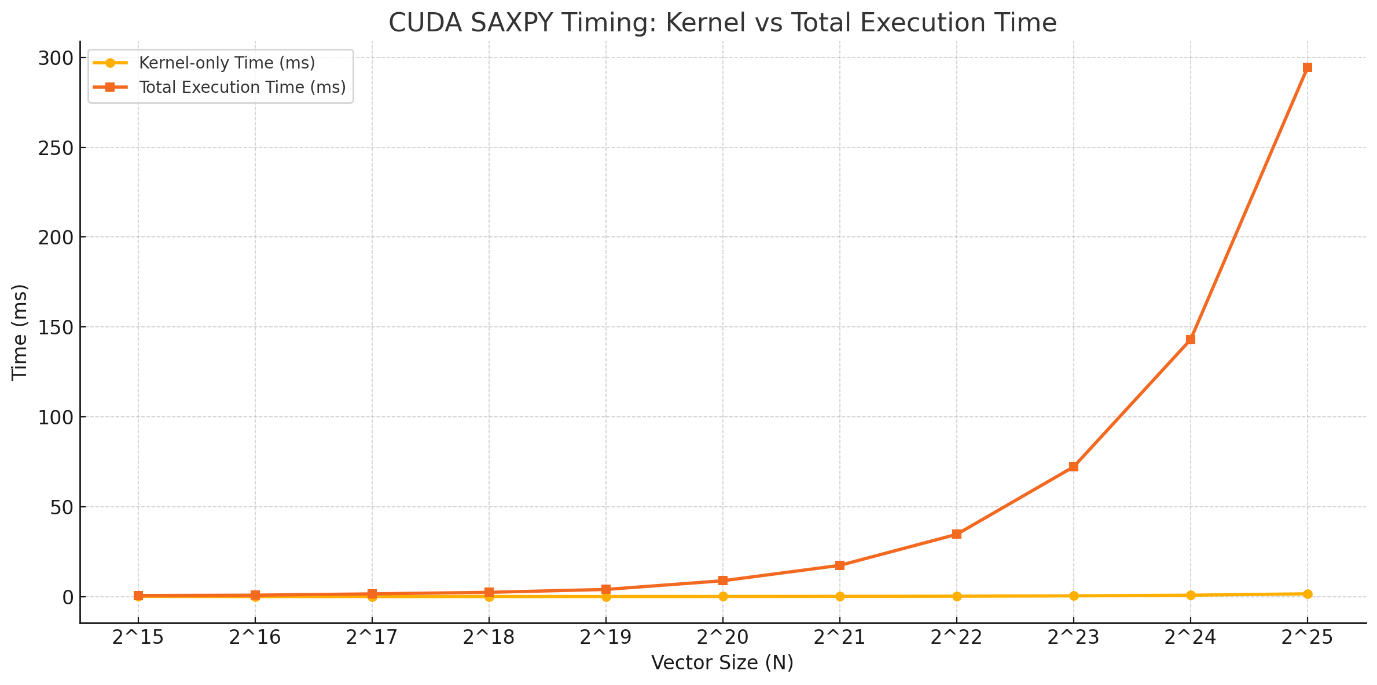
  return 0;

}

Below is the output of the above code.



The line plot comparing the kernel-only execution time and total execution time of the CUDA SAXPY operation reveals a clear performance distinction between the computation itself and the surrounding overhead. The kernel-only time remains very low and increases gradually as the vector size grows, highlighting the efficiency and scalability of GPU-based parallel computation for element-wise operations like SAXPY. In contrast, the total execution time—which includes memory allocation, data transfer between host and device, and kernel launch overhead—rises significantly faster, especially beyond vector sizes of 2202^{20}. This growing gap between the two curves indicates that for large data sizes, the memory management and data transfer overheads dominate the total time. It also reinforces the fact that while GPU kernels are extremely fast, efficient overall performance depends heavily on minimizing host-device communication and optimizing memory operations.



Line graph comparing time with vector size

**Challenges Faced:**

1. **Initial nvcc and architecture errors**: Had to fix version mismatch and install CUDA Toolkit properly in Colab.
2. **Missing nvcc compiler**: Required setting up PATH and LD\_LIBRARY\_PATH manually after toolkit installation.
3. **Incorrect malloc behavior**: Encountered failed cudaMalloc due to improper setup or resource constraints.
4. **Understanding performance**: Needed to separate **kernel-only time** from **total execution time** to analyze bottlenecks.

**What I Learned:**

* How to install and configure CUDA in Google Colab.
* Difference between computation time (kernel) and total execution time, including memory transfers and allocations.
* SAXPY is a classic example of a GPU-friendly operation due to its simple, independent element-wise calculation.
* Importance of benchmarking with increasing input sizes to analyze scalability.

**Conclusion:**

SAXPY was an ideal use case to explore the power of GPU parallelism. It showcased that GPUs handle large-scale vectorized operations efficiently, but performance heavily depends on minimizing host-device memory transfers. I gained hands-on experience with CUDA memory management, kernel timing, and profiling.