

## PRACTICAL – 5

**AIM :** Write a program to check task distribution using Gprof.

### **THEORY :**

#### **WHAT ARE PROFILERS?**

Profilers are tools used in software development to analyze and measure the performance of a program. They help developers identify bottlenecks, memory leaks, and other inefficiencies in their code. Profilers work by collecting data about various aspects of a program's execution, such as CPU usage, memory usage, function call frequency, and execution time.

There are two main types of profilers:

- 1. CPU Profilers:** These tools monitor the usage of the central processing unit (CPU) during the execution of a program. They provide insights into which functions or code segments consume the most CPU time. This information is valuable for optimizing code and improving overall performance.
- 2. Memory Profilers:** Memory profilers focus on a program's memory usage. They help identify memory leaks, excessive memory consumption, and inefficient memory management. By analyzing memory usage patterns, developers can optimize their code to use memory more efficiently.

Profiling can be done at different levels, including function-level profiling, line-level profiling, and even profiling for specific hardware components. Profilers generate reports or visualizations that aid developers in understanding how their code performs under various conditions. Using profilers is a common practice in performance tuning and optimization, as it allows developers to target specific areas of improvement in their software.

#### **WHAT IS GPROF ?**

Gprof is a powerful profiling tool included in the GNU Compiler Collection (GCC). It is designed to assist developers in analyzing the performance of their C, C++, and Fortran programs. Profiling is crucial for identifying performance bottlenecks and optimizing code for efficiency.

#### **Workflow:**

##### **1. Compilation:**

- Compile the code using the `-pg`` flag to instruct GCC to include profiling information in the executable.

##### **2. Execution:**

- Run the compiled program to generate a profiling data file.

##### **3. Analysis:**

- Employ Gprof to analyze the generated data file, producing a detailed report with function-level statistics, call graphs, and other relevant insights.

#### **Interpretation:**

##### **1. Flat Profile:**

- Gprof's flat profile section summarizes individual function performance metrics, offering a quick overview of time-related statistics.

##### **2. Call Graph:**

- The call graph section provides a visual representation of how functions are interlinked, helping developers grasp the program's execution flow and identify critical paths.

#### **Benefits:**

##### **1. Identifying Hotspots:**

- Gprof aids developers in identifying performance hotspots—functions consuming a significant portion of execution time—allowing for targeted optimization efforts.

##### **2. Optimization Guidance:**

- Armed with Gprof's insights, developers can make informed decisions about code optimization, focusing on critical sections to enhance overall performance.

##### **3. Resource Efficiency:**

- Gprof contributes to efficient resource utilization by highlighting areas for improvement in terms of both time and memory.

#### **Key Features:**

##### **1. Function-Level Profiling:**

- Gprof excels in function-level profiling, offering detailed information on the time spent in each function during program execution.
- It records the number of calls to each function, the time spent in each, and the percentage of total execution time attributed to specific functions.

##### **2. Call Graphs:**

- One of Gprof's standout features is its ability to generate call graphs, illustrating the relationships between functions and how they are invoked.
- Call graphs provide a visual representation of the program's execution flow, aiding developers in understanding complex code structures.

##### **3. Time and Percentage Data:**

- Gprof provides comprehensive time-related metrics, including the total time spent in the program, time spent in individual functions, and the percentage of overall time allocated to specific functions.

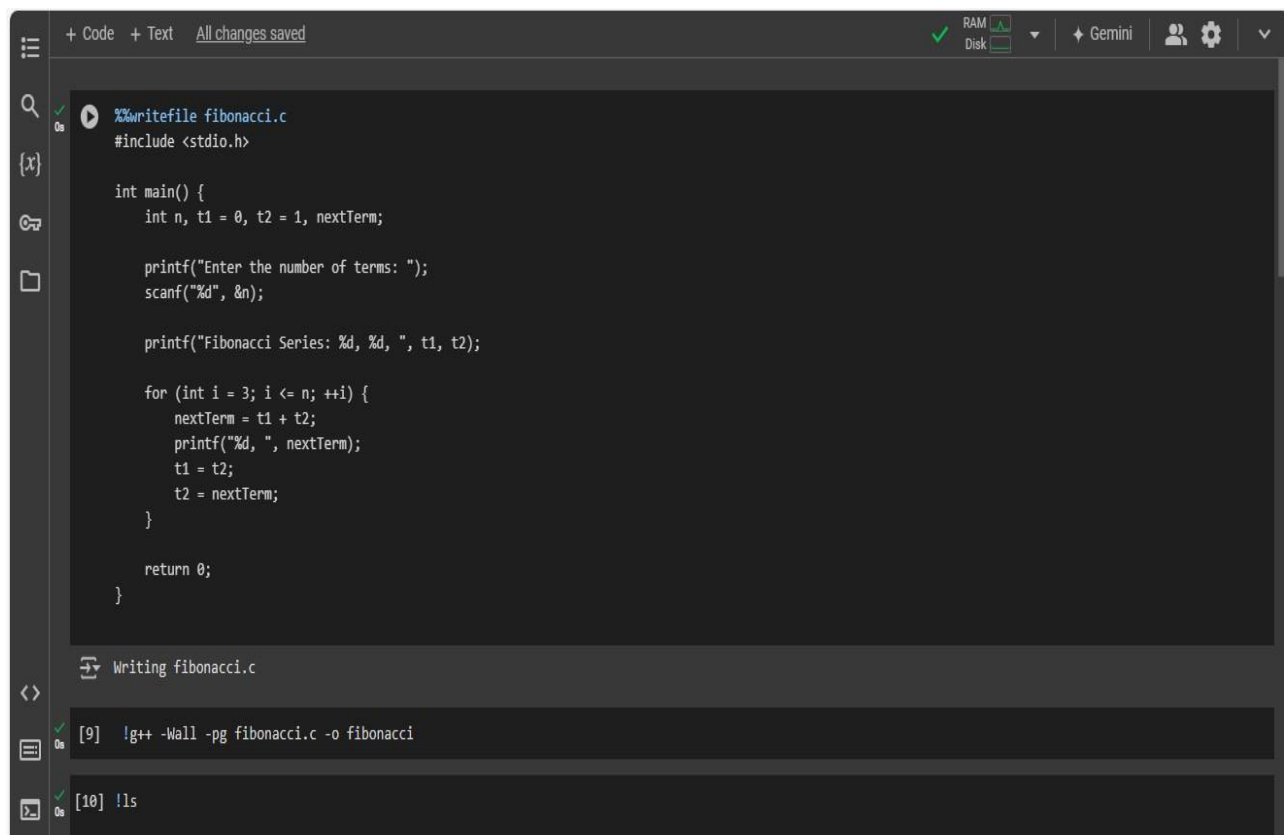
#### 4. Memory Profiling:

- While Gprof primarily focuses on time-based profiling, it offers insights into memory usage, contributing to a more holistic performance analysis.

#### 5. Easy Integration:

- Gprof seamlessly integrates into the GCC workflow. Developers can activate profiling during compilation by adding the `-pg` flag, minimizing the effort required to enable profiling.

### CODE :



```
+ Code + Text All changes saved
RAM
Disk
Gemini
[?]

%writefile fibonacci.c
#include <stdio.h>

int main() {
    int n, t1 = 0, t2 = 1, nextTerm;

    printf("Enter the number of terms: ");
    scanf("%d", &n);

    printf("Fibonacci Series: %d, %d, ", t1, t2);

    for (int i = 3; i <= n; ++i) {
        nextTerm = t1 + t2;
        printf("%d, ", nextTerm);
        t1 = t2;
        t2 = nextTerm;
    }

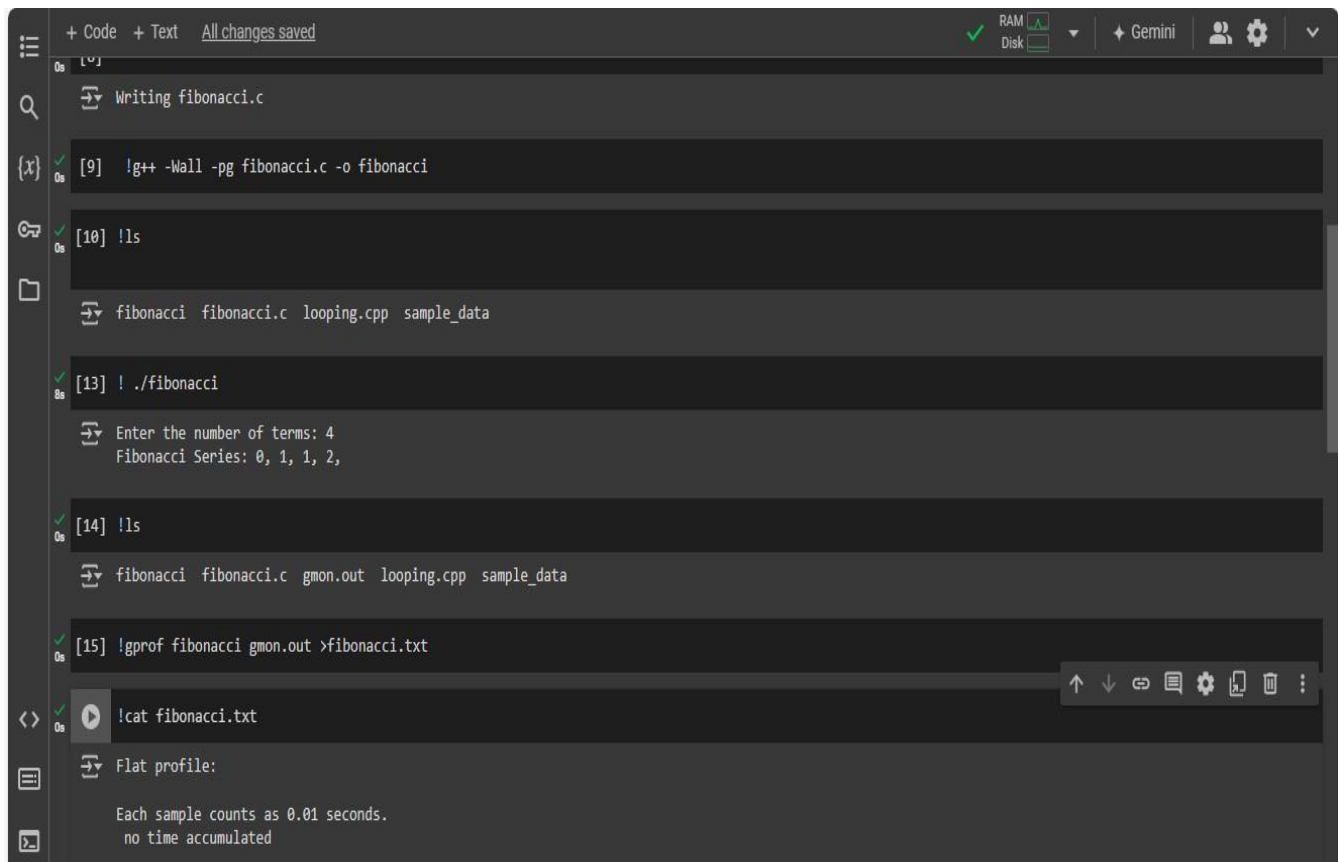
    return 0;
}

Writing fibonacci.c

[9] !g++ -Wall -pg fibonacci.c -o fibonacci

[10] !ls
```

## OUTPUT :



```
+ Code + Text All changes saved
[9] !g++ -Wall -pg fibonacci.c -o fibonacci
[10] !ls
fibonacci fibonacci.c looping.cpp sample_data
[13] ! ./fibonacci
Enter the number of terms: 4
Fibonacci Series: 0, 1, 1, 2,
[14] !ls
fibonacci fibonacci.c gmon.out looping.cpp sample_data
[15] !gprof fibonacci gmon.out >fibonacci.txt
!cat fibonacci.txt
Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated
```

```
+ Code + Text All changes saved
!cat fibonacci.txt
Flat profile:

Each sample counts as 0.01 seconds.
no time accumulated

% cumulative self self total
time seconds seconds calls Ts/call Ts/call name

% the percentage of the total running time of the
time program used by this function.

cumulative a running sum of the number of seconds accounted
seconds for by this function and those listed above it.

self the number of seconds accounted for by this
seconds function alone. This is the major sort for this
listing.

calls the number of times this function was invoked, if
this function is profiled, else blank.

self the average number of milliseconds spent in this
ms/call function per call, if this function is profiled,
else blank.

total the average number of milliseconds spent in this
ms/call function and its descendents per call, if this
function is profiled, else blank.

name the name of the function. This is the minor sort
for this listing. The index shows the location of
the function in the source file.

✓ 0s completed at 1:56 PM
```

## PRACTICAL – 9

**AIM :** Write a simple CUDA program to print “Hello World!”

### **THEORY :**

CUDA, an acronym for Compute Unified Device Architecture, stands as a revolutionary paradigm developed by NVIDIA, redefining the landscape of parallel computing. This cutting-edge platform and programming model have been meticulously crafted to harness the formidable processing prowess embedded in Graphical Processing Units (GPUs). Unlike its predecessors, CUDA extends beyond the traditional realm of graphics rendering, empowering developers to orchestrate GPUs for a myriad of general-purpose computing tasks.

### **FEATURES OF CUDA :**

**1. Parallel Computing Evolution :**

At its core, CUDA marks a transformative shift by unlocking the latent potential of GPUs, traditionally confined to rendering graphics, for a broader range of computational tasks.

**2. GPU Acceleration Unleashed:**

A cornerstone feature of CUDA is its ability to unlock the untapped potential of GPU acceleration. This capability is especially valuable for scientific simulations, deep learning algorithms, and applications with high computational demands.

**3. Flexible Programming Model:**

CUDA introduces a flexible programming model that extends conventional languages like C, C++, and Fortran. Developers can seamlessly incorporate CUDA constructs into their code, facilitating the expression of parallelism.

**4. Memory Hierarchy Optimization:**

CUDA incorporates a sophisticated memory model, encompassing global, shared, and constant memory. This intricate hierarchy optimizes data access patterns, enhancing the efficiency of parallel processing.

**5. Threaded Execution Paradigm:**

The heart of CUDA lies in its threaded execution paradigm. Tasks are executed concurrently through parallel threads organized into blocks and grids, ensuring optimal GPU utilization.

**6. Unified Virtual Addressing (UVA):**

Simplifying the programming model, UVA provides a unified view of memory. This unification streamlines data access across both the CPU and GPU, fostering a seamless integration of processing resources.

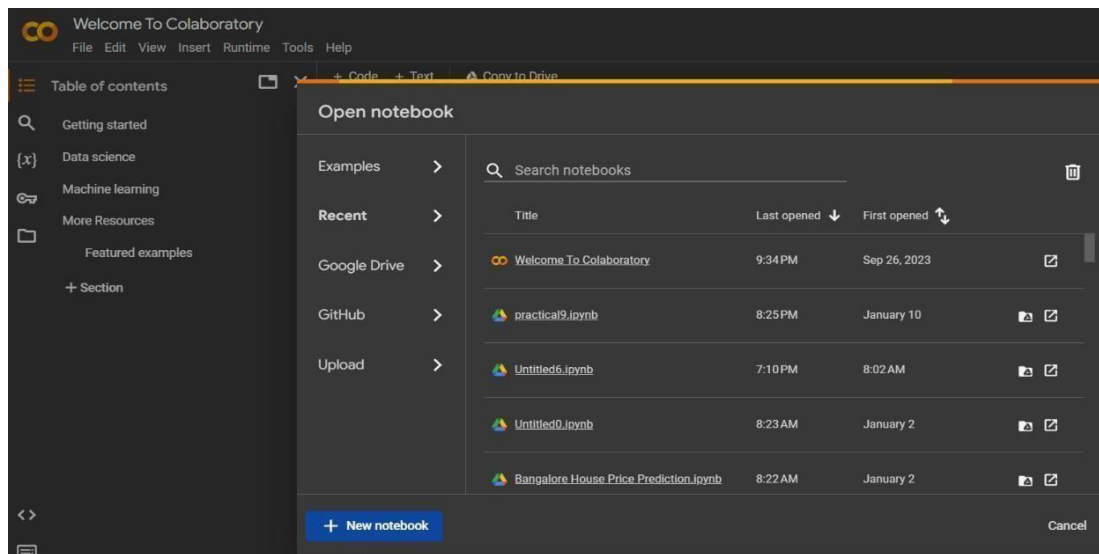
**7. Rich Set of GPU Libraries:**

NVIDIA complements CUDA with a rich set of GPU-accelerated libraries. These libraries span diverse domains, from linear algebra to signal processing, enriching the development process and overall computational efficiency.

## CUDA Deployment: Streamlining GPU Acceleration

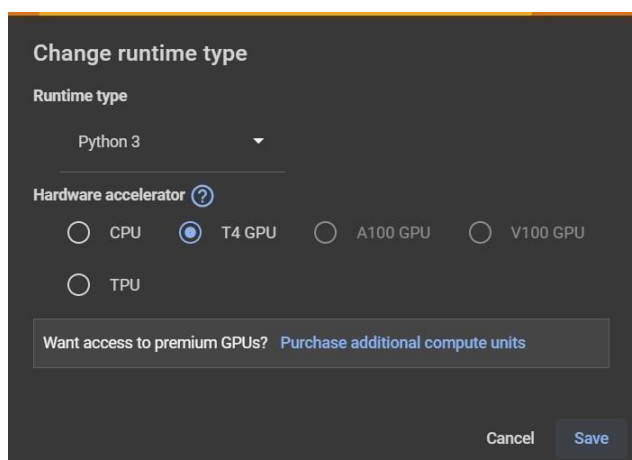
### 10. Run CUDA-Enabled Environment:

Launch your development environment, ensuring it supports CUDA. Google Colab which we can run multiple languages.



### 11. Select GPU:

Ensure your system has a compatible NVIDIA GPU. Run the command `nvidia-smi` to verify GPU availability and details. Select a change runtime type and select T4 GPU which supports the CUDA environment.



### 12. Check CUDA Version:

Confirm the CUDA Toolkit version on your system using `nvcc --version` or `cat /usr/local/cuda/version.txt`. This step ensures compatibility with CUDA-dependent applications.

```
!nvcc --version

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

### 13. Install CUDA Toolkit:

If CUDA isn't installed, navigate to the official NVIDIA CUDA Toolkit website and download the appropriate version. We can see the version of CUDA by the command '`!nvcc --version`'.

```
!nvcc --version

nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

### 14. Pip Install CUDA-Supported Libraries:

Utilize pip to install CUDA-supported Python libraries, such as TensorFlow or PyTorch. Example: `pip install tensorflow-gpu`.

We can install by the command '`!pip install git+https://github.com/andreinechaev/nvcc4jupyter.git`'.

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

collecting git+https://github.com/andreinechaev/nvcc4jupyter.git
Cloning https://github.com/andreinechaev/nvcc4jupyter.git to /tmp/pip-req-build-uqpw_kl
Running command git clone --filter=blob:none --quiet https://github.com/andreinechaev/nvcc4jupyter.git /tmp/pip-req-build-uqpw_kl
Resolved https://github.com/andreinechaev/nvcc4jupyter.git to commit 0d2ab99ccbbc682722e708515fe9c4cfc50185a
Preparing metadata (setup.py) ... done
```

### 15. Load CUDA Plugins:

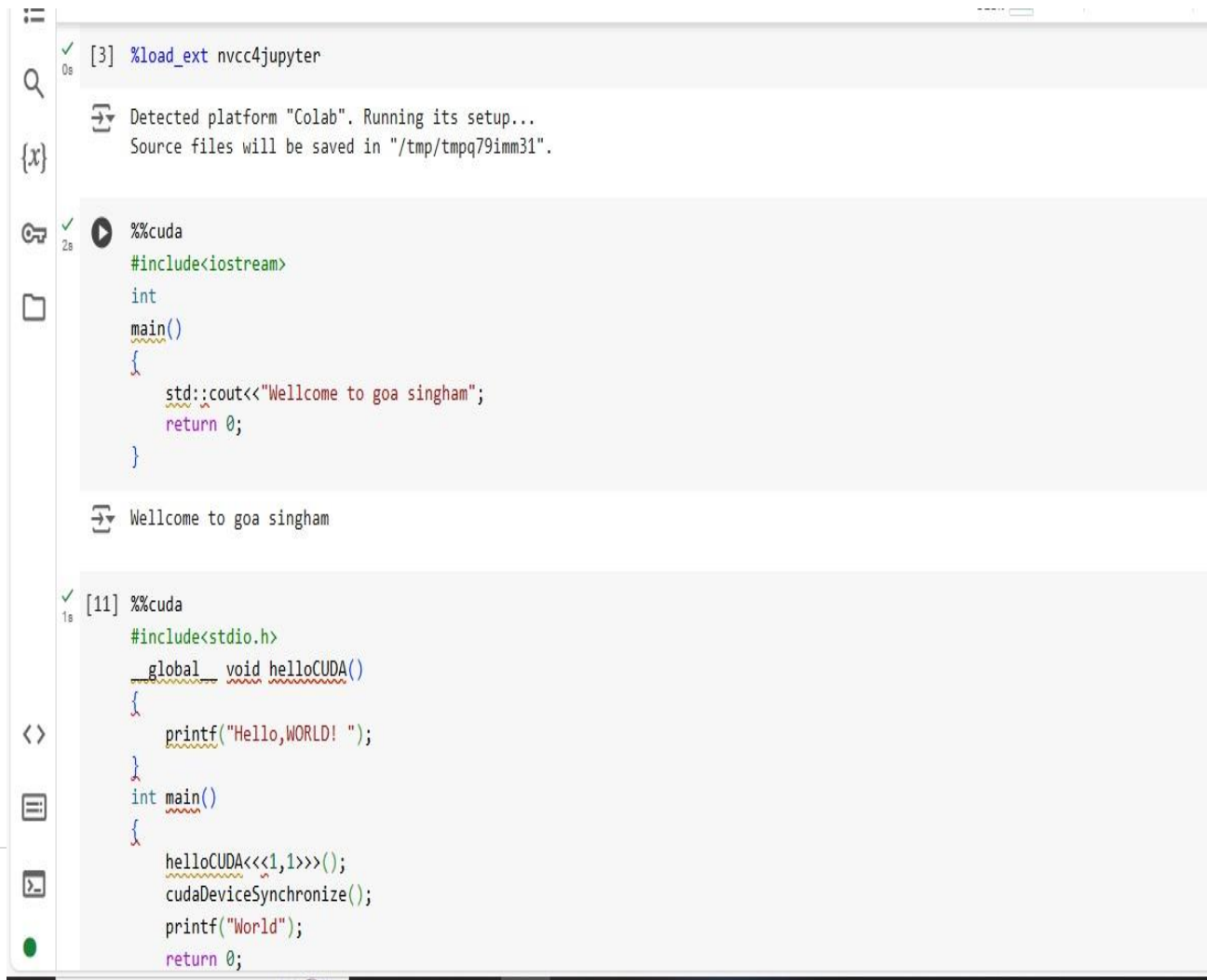
If working within an environment like PyTorch, ensure CUDA support is enabled. This usually involves importing the necessary CUDA modules and checking for GPU availability using `torch.cuda.is_available()`.

```
%load_ext nvcc_plugin

The nvcc_plugin extension is already loaded. To reload it, use:
%reload_ext nvcc_plugin
```



## 16.Code Execution:



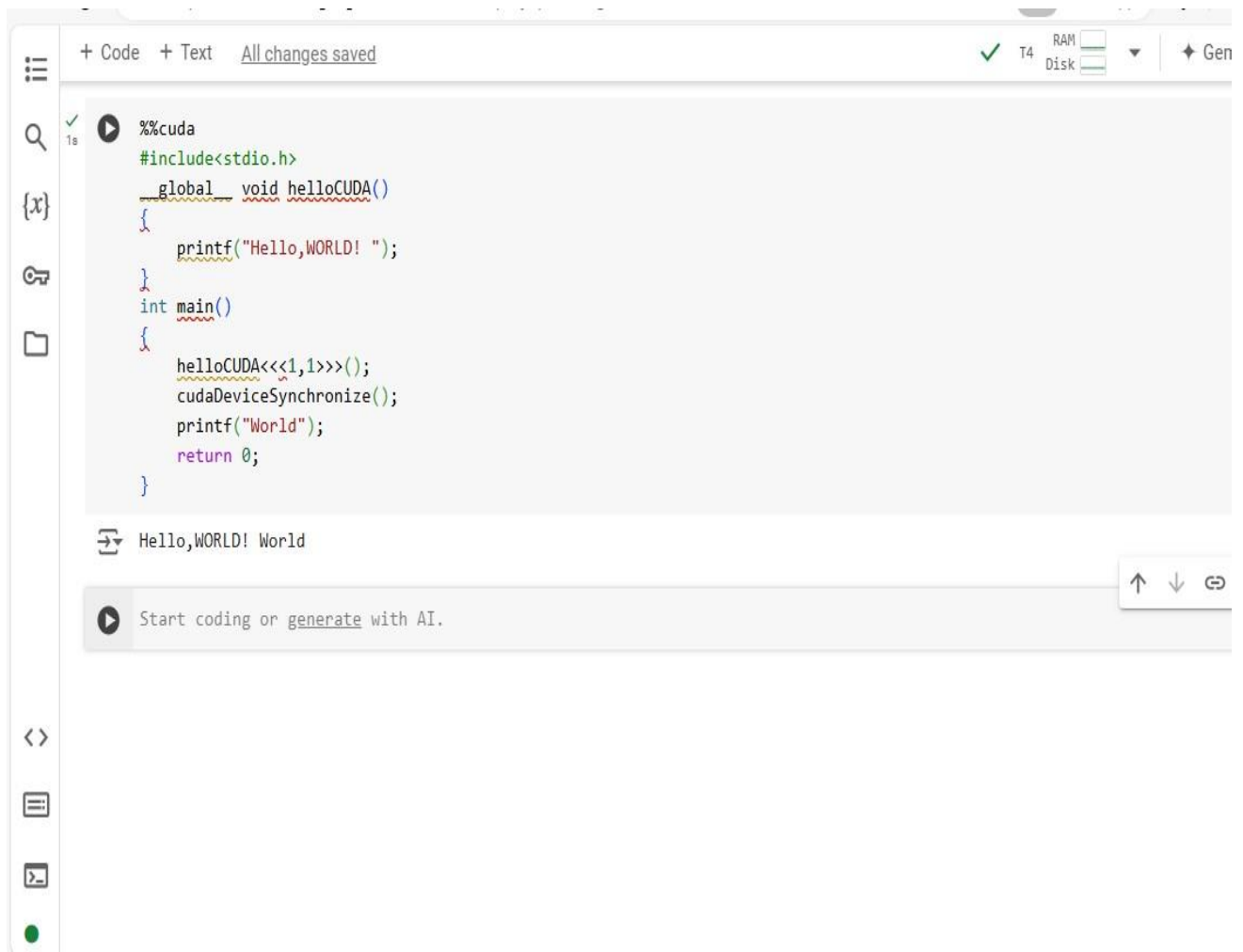
```
[3] %load_ext nvcc4jupyter

Detected platform "Colab". Running its setup...
Source files will be saved in "/tmp/tmpq79imm31".

%%cuda
#include<iostream>
int
main()
{
    std::cout<<"Wellcome to goa singham";
    return 0;
}

Wellcome to goa singham

[11] %%cuda
#include<stdio.h>
__global__ void helloCUDA()
{
    printf("Hello,WORLD! ");
}
int main()
{
    helloCUDA<<<1,1>>>>();
    cudaDeviceSynchronize();
    printf("World");
    return 0;
}
```



The screenshot shows a code editor interface with a sidebar on the left containing icons for file explorer, search, and other tools. The main editor area displays a C++ program with CUDA syntax. The code includes a header, a global function, and a main function. The output of the program is shown in a terminal window below the code. At the bottom, there is a prompt to start coding or generate with AI.

```
%%cuda
#include<stdio.h>
__global__ void helloCUDA()
{
    printf("Hello,WORLD! ");
}
int main()
{
    helloCUDA<<<1,1>>>>();
    cudaDeviceSynchronize();
    printf("World");
    return 0;
}
```

1s Hello,WORLD! World

Start coding or [generate](#) with AI.