Shri Vile Parle Kelavani Mandal's
DWARKADAS J. SANGHVI COLLEGE OF ENGINEERING
(Autonomous College Affiliated to the University of Mumbai)
NAAC Accredited with "A" Grade (CGPA : 3.18)

Department of Computer Science and Engineering (Data Science)
High Performance Computing Laboratory (DJ19DSL802)

HPC Experiment-3

Bhuvi Ghosh
60009210191

Aim: Accelerating a For Loop

with a Single Block of Threads

Theory:

For loops in CPU-only applications are ripe for acceleration: rather than run each iteration of the loop serially, each iteration of the loop can be run in parallel in its own thread. Consider the following for loop, and notice, though it is obvious, that it controls how many times the loop will execute, as well as defining what will happen for each iteration of the loop:

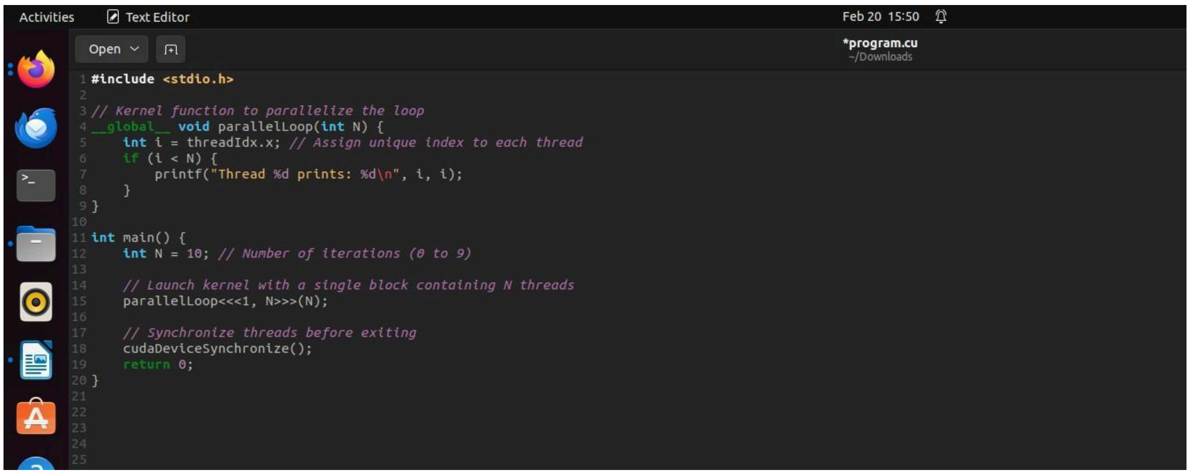
```
int N = 2<<20;

for (int i = 0; i < N; ++i)

{

  printf("%d\n", i);

}
```

In order to parallelize this loop, 2 steps must be taken:

1. A kernel must be written to do the work of a single iteration of the loop.
2. Because the kernel will be agnostic of other running kernels, the execution configuration must be such that the kernel executes the correct number of times, for example, the number of times the loop would have iterated.

Lab Experiment to be performed:

Currently, the loop function inside 01-single-block-loop.cu, runs a for loop that will serially print the numbers 0 through 9. Refactor the loop function to be a CUDA kernel which will launch to execute N iterations in parallel. After successfully refactoring, the numbers 0 through 9 should still be printed.

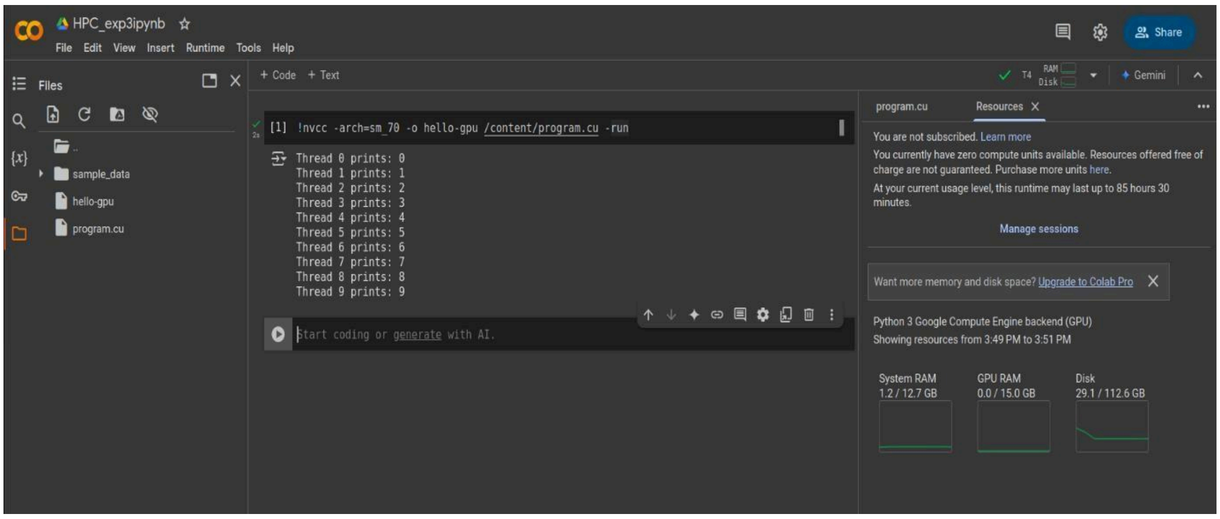


```
#include <stdio.h>

// Kernel function to parallelize the loop
__global__ void parallelLoop(int N) {
    int i = threadIdx.x; // Assign unique index to each thread
    if (i < N) {
        printf("Thread %d prints: %d\n", i, i);
    }
}

int main() {
    int N = 10; // Number of iterations (0 to 9)

    // Launch kernel with a single block containing N threads
    parallelLoop<<<1, N>>>(N);

    // Synchronize threads before exiting
    cudaDeviceSynchronize();
    return 0;
}
```

This CUDA program demonstrates how parallel execution works using GPU threads. The key concept here is that each thread operates independently and prints its unique thread index, showcasing how parallel processing distributes workload across multiple threads.

Key Components of the Code:

1. Kernel Function (parallelLoop)
   - This function runs on the GPU (_global____keyword).
   - Each thread calculates its unique index (threadIdx.x).
   - If the index is within the limit N, the thread prints its ID.
2. Main Function (main())
   - Defines N = 10, meaning 10 threads will be launched.
   - The kernel function is executed using parallelLoop<<<1, N>>>();
     - 1 block with N threads is launched.
   - cudaDeviceSynchronize(); ensures all threads complete execution before the program terminates.



```
[1] !nvcc -arch=sm_70 -o hello-gpu /content/program.cu -run
Thread 0 prints: 0
Thread 1 prints: 1
Thread 2 prints: 2
Thread 3 prints: 3
Thread 4 prints: 4
Thread 5 prints: 5
Thread 6 prints: 6
Thread 7 prints: 7
Thread 8 prints: 8
Thread 9 prints: 9
```

You are not subscribed. Learn more
You currently have zero compute units available. Resources offered free of charge are not guaranteed. Purchase more units here.
At your current usage level, this runtime may last up to 85 hours 30 minutes.
Manage sessions
Want more memory and disk space? Upgrade to Colab Pro
Python 3 Google Compute Engine backend (GPU)
Showing resources from 3:49 PM to 3:51 PM
System RAM 1.2 / 12.7 GB
GPU RAM 0.0 / 15.0 GB
Disk 29.1 / 112.6 GB

- Each thread prints its thread index from 0 to 9.
- Due to the parallel nature of GPU execution, the order of outputs may not always be sequential, depending on how threads are scheduled.
- The program confirms that each thread operates independently and executes its task in parallel.

Conclusion

In this experiment, we transformed a serial for loop into a parallel execution model using CUDA. By launching a kernel with multiple threads, each thread executed a portion of the loop iterations independently, demonstrating the power of GPU parallelism. The output remained consistent, confirming the correctness of the parallelized approach. This optimization significantly improves execution speed for large-scale iterations, highlighting the advantages of CUDA programming in high-performance computing.