PES
UNIVERSITY

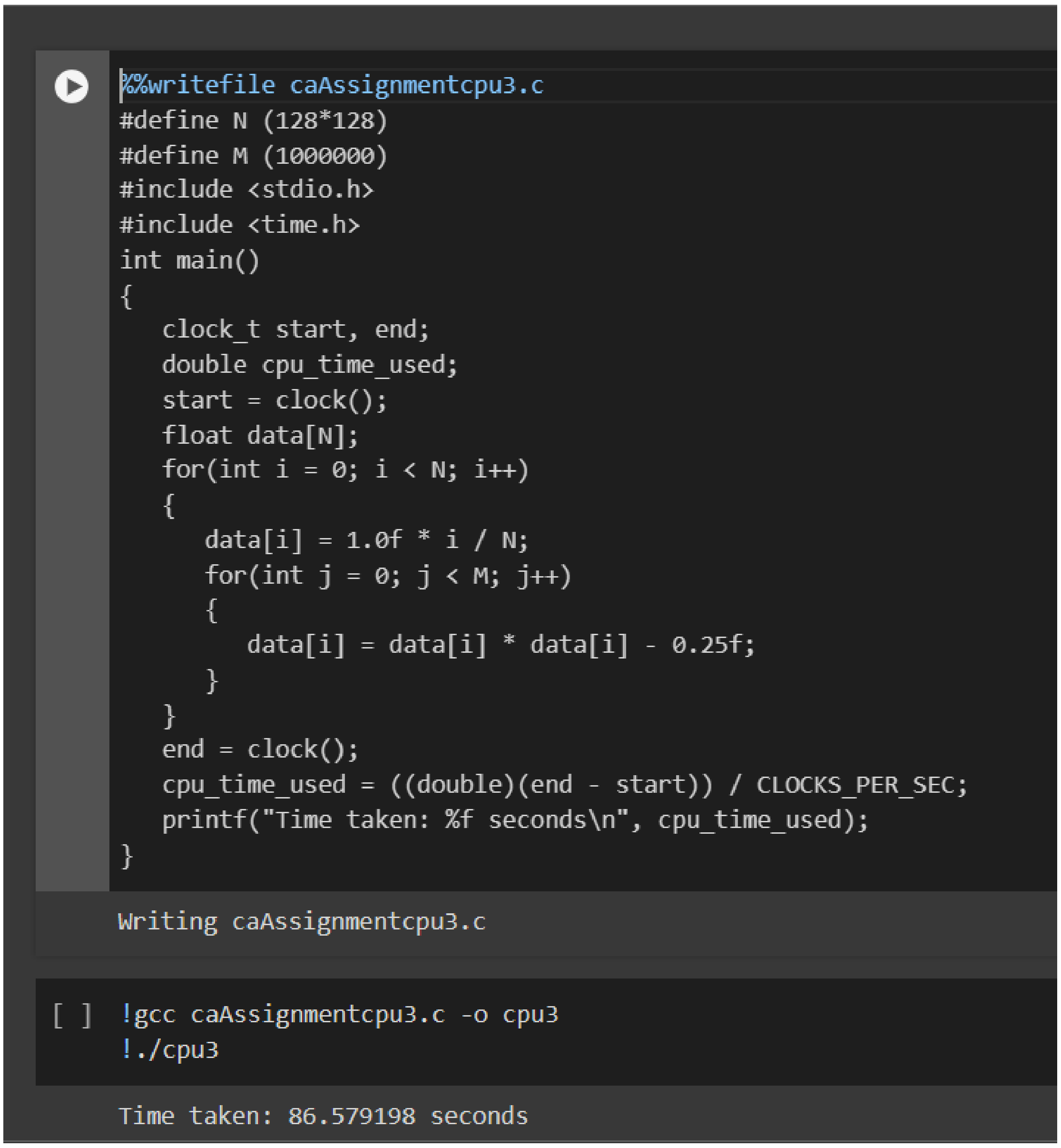
Computer Architecture
UE21EC341B

Project

Title: "GPU Parallel Computation Benchmark with CUDA"

Name : AKASH RAVI BHAT
SRN : PES1UG21EC025
Class : 6 “A”
Mentor: Dr.Rajeshwari B

CPU code :



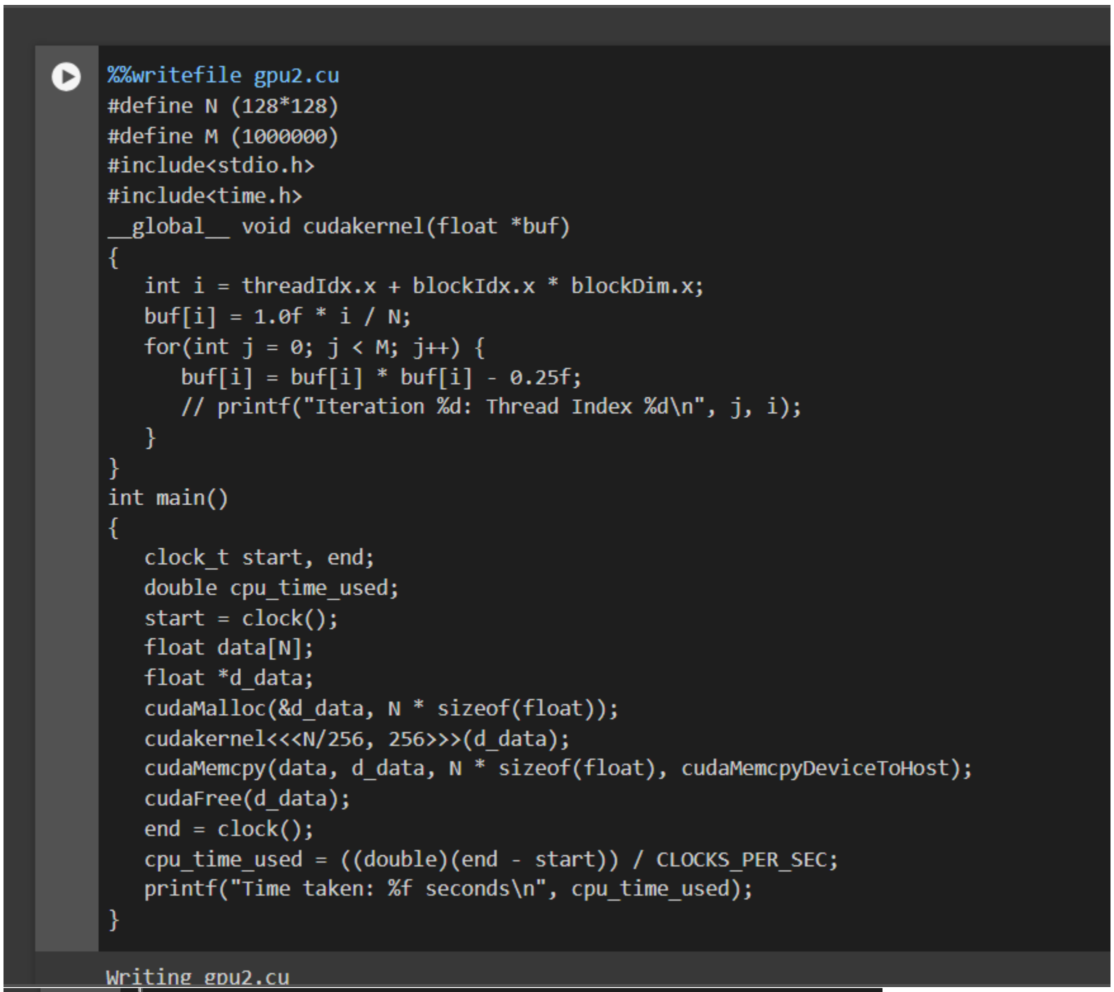
```
%%writefile caAssignmentcpu3.c
#define N (128*128)
#define M (1000000)
#include <stdio.h>
#include <time.h>
int main()
{
   clock_t start, end;
   double cpu_time_used;
   start = clock();
   float data[N];
   for(int i = 0; i < N; i++)
   {
      data[i] = 1.0f * i / N;
      for(int j = 0; j < M; j++)
      {
         data[i] = data[i] * data[i] - 0.25f;
      }
   }
   end = clock();
   cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Time taken: %f seconds\n", cpu_time_used);
}
```

```
Writing caAssignmentcpu3.c
```

```
!gcc caAssignmentcpu3.c -o cpu3
!./cpu3
```

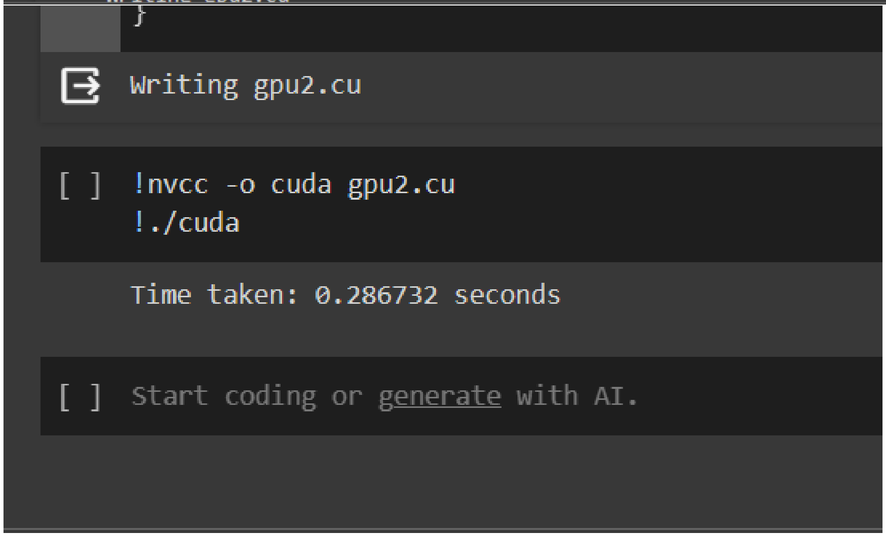
```
Time taken: 86.579198 seconds
```

CUDA code :



```
%%writefile gpu2.cu
#define N (128*128)
#define M (1000000)
#include<stdio.h>
#include<time.h>
__global__ void cudakernel(float *buf)
{
   int i = threadIdx.x + blockIdx.x * blockDim.x;
   buf[i] = 1.0f * i / N;
   for(int j = 0; j < M; j++) {
      buf[i] = buf[i] * buf[i] - 0.25f;
      // printf("Iteration %d: Thread Index %d\n", j, i);
   }
}
int main()
{
   clock_t start, end;
   double cpu_time_used;
   start = clock();
   float data[N];
   float *d_data;
   cudaMalloc(&d_data, N * sizeof(float));
   cudakernel<<<N/256, 256>>>(d_data);
   cudaMemcpy(data, d_data, N * sizeof(float), cudaMemcpyDeviceToHost);
   cudaFree(d_data);
   end = clock();
   cpu_time_used = ((double)(end - start)) / CLOCKS_PER_SEC;
   printf("Time taken: %f seconds\n", cpu_time_used);
}
```

```
Writing gpu2.cu
```



```
}
```

```
Writing gpu2.cu
```

```
!nvcc -o cuda gpu2.cu
!./cuda
```

```
Time taken: 0.286732 seconds
```

- Variable Declarations: The program declares variables start, end, and cpu_time_used of types clock_t and double to record the start time, end time, and CPU time used, respectively.
- Array Declaration: An array data of size N (16384) is declared to store floating-point numbers.
- Nested Loops: Two nested loops are used to perform arithmetic operations on each element of the data array. The outer loop iterates over each element of the array, and the inner loop performs a series of arithmetic operations M times on each element.
- Timing Measurement: The clock() function is used to record the start and end times of the computation. The CPU time used is calculated by subtracting the start time from the end time and dividing by CLOCKS_PER_SEC, which gives the time in seconds.
- Output: The program prints the time taken for the computation in seconds.

1. **!gcc caAssignmentcpu3.c -o cpu3**: This command compiles the C code (**caAssignmentcpu3.c**) using the GNU Compiler Collection (**gcc**). It generates an executable file named **cpu3** using the **-o** flag. After successful compilation, you'll have an executable file named **cpu3** in the current directory.
2. **!./cpu3**: This command executes the compiled program named **cpu3**. The **./** before **cpu3** specifies that the program should be executed from the current directory. When you run this command, it will execute the compiled C program and print the output to the console.

So, in summary, these commands first compile the C code and then run the compiled executable, which results in the execution of the C program and prints the time taken for the computatio

- **threadIdx.x** gives the index of the thread within its block.
- **blockIdx.x** gives the index of the block within the grid.
- **blockDim.x** gives the number of threads per block.

The variable **i** is calculated to determine the global index of the thread. Each thread computes its corresponding element of the array **buf**. The kernel performs a series of arithmetic operations on each element of the array **buf** for **M** iterations. Additionally, it prints the index of the current thread and the iteration number for monitoring purposes.

- **Variable Declarations**: The program declares variables **start**, **end**, and **cpu_time_used** of types **clock_t** and **double** to record the start time, end time, and CPU time used, respectively.
- **Array Declaration**: An array **data** of size **N** (16384) is declared to store floating-point numbers.
- **Memory Allocation**: Memory is allocated on the GPU using **cudaMalloc** for the array **d_data**.
- **Kernel Launch**: The CUDA kernel **cudakernel** is launched with a grid of **N/256** blocks, each containing 256 threads.
- **Data Transfer**: The computed data is copied from the device (GPU) to the host (CPU) using **cudaMemcpy**.
- **Memory Deallocation**: The memory allocated on the device is freed using **cudaFree**.
- **Timing Measurement**: The **clock()** function is used to record the start and end times of the computation. The CPU time used is calculated by subtracting the start time from the end time and dividing by **CLOCKS_PER_SEC**, which gives the time in seconds.
- **Output**: The program prints the time taken for the computation in seconds.

1. **!nvcc -o cuda gpu2.cu**: This command compiles the CUDA C code (**gpu2.cu**) using the NVIDIA CUDA Compiler (**nvcc**). It generates an executable file named **cuda** using the **-o** flag. After successful compilation, you'll have an executable file named **cuda** in the current directory.
2. **!./cuda**: This command executes the compiled CUDA program named **cuda**. The **./** before **cuda** specifies that the program should be executed from the current directory. When you run this command, it will execute the compiled CUDA program and print the output to the console.

So, in summary, these commands first compile the CUDA C code and then run the compiled executable, which results in the execution of the CUDA program and prints the time taken for the computation.
The provided codes are implementations of the same computation task using different computing architectures:

1. **caAssignmentcpu3.c**:
   - This code is written in standard C and executed on the CPU.
   - It performs floating-point arithmetic operations on an array of floating-point numbers using nested loops.
   - The outer loop iterates over each element of the array, and the inner loop performs a series of arithmetic operations on each element.
   - The computation is sequential and executed on the CPU.

2. **gpu2.cu**:
   - This code is written in CUDA C and executed on the GPU.
   - It utilizes NVIDIA's CUDA platform for parallel processing.
   - The computation is parallelized using CUDA threads, where each thread computes a portion of the array independently.

   - The CUDA kernel function `cudakernel` is launched with multiple threads, and each thread computes its corresponding element of the array in parallel.
   - The computation is parallel and executed on the GPU.

Differences:
- **Architecture**: The first code runs on the CPU, while the second code runs on the GPU using CUDA.
- **Parallelism**: The GPU code exploits parallelism by executing multiple threads simultaneously, while the CPU code performs computation sequentially.
- **Implementation**: While both codes perform the same arithmetic operations, the GPU code utilizes CUDA-specific syntax and constructs for parallel execution, such as kernel functions and thread management.
- **Performance**: Generally, GPU code can offer significantly higher performance for parallelizable tasks compared to CPU code due to the massively parallel architecture of GPUs. The provided codes, one for CPU and the other for GPU, can be applied to various numerical computation tasks. Here are some potential applications for each code:

Application of CPU Code (caAssignmentcpu3.c):
1. **Scientific Computing**: The CPU code can be used for numerical simulations in scientific computing, such as solving differential equations, finite element analysis, or computational fluid dynamics.

2. **Data Processing**: It can be applied to process large datasets in data analytics tasks, such as data filtering, transformation, or statistical analysis.

3. **Algorithm Development**: The CPU code can serve as a reference implementation for developing and testing

algorithms before optimizing them for parallel execution on GPUs.

4. **Educational Purposes**: It can be used as a teaching tool in computer science and engineering courses to illustrate sequential programming concepts and basic numerical computations.

Application of GPU Code (gpu2.cu):
1. **Image Processing**: The GPU code can be utilized for real-time image processing tasks, such as image filtering, edge detection, or image segmentation, where parallel computation can significantly improve performance.

2. **Scientific Simulations**: It can be applied to accelerate scientific simulations, such as molecular dynamics simulations, Monte Carlo simulations, or computational finance models, by leveraging the parallel processing power of GPUs.

3. **Machine Learning**: The GPU code can accelerate machine learning algorithms, especially deep learning training tasks, by parallelizing matrix operations involved in neural network computations.

4. **Numerical Solvers**: It can be used to solve large-scale numerical problems, such as solving systems of linear equations, numerical integration, or optimization problems, by distributing computations across GPU threads.

5. **Data Visualization**: The GPU code can be employed for real-time data visualization tasks, such as rendering complex 3D scenes, visualizing large datasets, or creating interactive visualizations, benefiting from the GPU's rendering capabilities.

CONCLUSION:

In summary, while the CPU code is suitable for sequential numerical computations and data processing tasks, the GPU code can accelerate parallelizable computations, making it suitable for tasks that benefit from massive parallelism and high throughput, such as image processing, scientific simulations, machine learning, and data visualization.