COA LAB

ASSIGNMENT-2

01/08/2023

TEAM MEMBERS

- SHLOK KR. SHAW(21CS02008)
- JATIN YADAV(21CS02007)
- ANIKET ROY(21CS01061)
- AJINKYA DESHMUKH(21CS01029)
- GIRISH JAIN(21CS01016)

KERNEL

TECHNICAL DEFINITION: A kernel is the fundamental part of an operating system that acts as an intermediary between the hardware and software layers. It provides essential services, such as process and memory management, device drivers, input/output handling, and resource allocation. The kernel ensures that different software applications can interact with the computer's hardware in an organized and protected manner. It resides in privileged mode, allowing it to control and manage the system's resources and enforce security policies.

ANALOGY: The kernel is like the conductor of an orchestra. Just as a conductor leads and directs musicians to play in harmony, the kernel orchestrates the different components of the computer system to work together seamlessly. It coordinates the allocation of resources, manages the execution of processes, and ensures that the various software applications (musicians) can perform their tasks effectively and without conflicts.

PROCESS

TECHNICAL DEFINITION: A process is an independent instance of an executing program in a computer system. It is a self-contained entity with its own memory space, system resources, and execution environment. Each process operates in isolation from other processes, ensuring stability and security. Processes are managed by the operating system, and communication between processes typically involves inter-process communication mechanisms, such as message passing or shared memory.

ANALOGY: A process is like a separate workroom in a factory. Each workroom represents a distinct task or program running on the computer. Just as workrooms are physically separated and have their tools and materials, processes are isolated from each other and have their own memory space and resources. This isolation ensures that the actions of one process do not directly affect others, providing a stable and secure computing environment.

THREADS

TECHNICAL DEFINITION: A thread is the smallest unit of execution within a process. It represents an independent flow of control that can perform tasks concurrently with other threads within the same process. Threads share the same memory space and resources of the parent process, enabling efficient communication and sharing of data. Threads are lightweight compared to processes, as they don't require a separate memory space but operate within the context of the parent process.

ANALOGY: Threads are like multiple actors performing different roles in a play. Each actor represents a separate thread of execution, contributing to the overall performance (process). Threads can interact with each other, just as actors on the same stage can communicate and collaborate during the play. Like actors sharing the same props and stage setting, threads share the same memory space and resources, making it easier for them to coordinate and work together on common tasks.

SIMD

TECHNICAL DEFINITION: SIMD (Single Instruction, Multiple Data) is a parallel computing technique where a single instruction is applied simultaneously to multiple data elements. It allows a processor to perform the same operation on multiple data points in parallel, exploiting data-level parallelism to accelerate computation. SIMD architectures typically consist of vector registers capable of storing multiple data elements and specialized instructions to perform parallel operations on these vectors.

ANALOGY: SIMD is like a bakery where a single chef prepares identical cookies in multiple trays at once. Instead of baking one cookie at a time, the chef uses a single set of instructions (recipe) to create batches of cookies in parallel. Similarly, SIMD processors use a single instruction to process multiple data elements simultaneously, significantly speeding up tasks that involve large amounts of data.

GPU MEMORY HIERARCHY

TECHNICAL DEFINITION: A The GPU memory hierarchy refers to the different levels of memory present in a Graphics Processing Unit (GPU) with varying characteristics. This hierarchy includes:

- SRAM (Static Random-Access Memory): Fast but limited in capacity, SRAM is used as cache memory on GPUs to store frequently accessed data and instructions, enabling quick access for processing.
- DRAM (Dynamic Random-Access Memory): Slower than SRAM but larger in capacity, DRAM serves as the primary memory on GPUs, storing data and instructions for ongoing computations.
- Shared memory: A fast and small memory space that is shared among threads within the same thread block on a GPU. It facilitates efficient communication and data sharing between threads, enhancing cooperation during parallel computations.
- Constant memory: A specialized read-only memory on GPUs used for storing data that remains constant throughout the execution of a kernel (a specific function or task running on the GPU).

ANALOGY: The GPU memory hierarchy is like a storage system in a library with different tiers:

- SRAM is like a librarian's desk where frequently accessed books are placed for quick retrieval. It's limited in capacity but provides fast access to essential resources.
- DRAM is like the library's main bookshelves, storing a vast collection of books. It's slower than SRAM but offers more significant storage capacity for less frequently accessed data.

CONTINUED...

- Shared memory is like a shared study area in the library where multiple students can access books and discuss their work together. It's a fast memory shared among threads within the same thread block, enabling efficient collaboration during computations.
- Constant memory is like a restricted section in the library with limited books, accessible to specific users. It is used for storing read-only data that remains constant throughout the execution of a specific task on the GPU.

SCHEDULER

TECHNICAL DEFINITION: A scheduler is a crucial component of the operating system responsible for managing the allocation of CPU time to different processes and threads. It decides the order and timing of task execution, ensuring fair distribution of CPU resources and efficient utilization. The scheduler employs scheduling algorithms to prioritize and switch between tasks, aiming to maximize overall system performance and responsiveness.

ANALOGY: A scheduler is like a traffic controller at a busy intersection. Just as the traffic controller manages the flow of vehicles, stopping and letting them go at specific intervals, the scheduler manages the flow of processes and threads on the CPU. It determines which tasks get to execute and for how long, preventing resource congestion and ensuring that each task gets its fair share of processing time.

WARP

TECHNICAL DEFINITION: A warp is the smallest unit of threads that can be scheduled and executed together on a GPU. It typically consists of 32 threads that execute the same instruction on different data elements in parallel. Warps are the building blocks of execution on GPUs, and all threads within a warp follow the same execution path, with each thread processing a unique data element.

ANALOGY: A warp is like a group of students in a class, moving together to the same destination. The students in a warp represent individual threads, and they advance through the course material (instructions) in unison. Just as all students in a warp follow the same curriculum, all threads in a warp execute the same instruction on different data elements in parallel, optimizing the GPU's performance and efficiency.

THREAD BLOCK

TECHNICAL DEFINITION: A thread block is a group of threads that cooperate and execute a specific task together on a GPU. Threads within a block can communicate and synchronize using shared memory, facilitating collaborative computation. Thread blocks are organized to form the basis of parallelism on GPUs, and they are scheduled and executed as atomic units.

ANALOGY: A thread block is like a team of workers collaborating on a project. Each worker represents an individual thread, and they work together on a shared task, dividing the workload among themselves efficiently. The team members communicate and coordinate their efforts to achieve the common goal, just as threads within a block communicate and synchronize using shared memory to perform parallel computations on the GPU.

MATRIX MULTIPLICATION USING CUDA PROGRAMMING

NOTE: In this code, we have implemented both, the Naive Approach and the Shared Memory Approach. To run the Naive approach, set the variable - "**SHARED**" to 0 and for Shared Memory approach, set it to 1, and run the code.

CODE:

```
#include <cassert>
#include <iostream>
#include <time.h>
using namespace std;

const bool SHARED = true;

// Matrix dimensions
const int M = 1e3 + 7;
const int N = 1e3 + 9;
const int K = 1e3 + 11;

// Threads per CTA dimension
const int THREADS = 16;

// Padded matrix dimensions
const int M_padded = M + THREADS - M % THREADS;
const int N_padded = N + THREADS - N % THREADS;
const int K_padded = K + THREADS - K % THREADS;

// Size of shared memory per TB
const int SHMEM_SIZE = THREADS * THREADS;

__global__ void matrixMul(const int *a, const int *b, int *c) {

  //Shared Memory Approach
  if(SHARED){
    // Compute each thread's global row and column index
    int row = blockIdx.y * blockDim.y + threadIdx.y;
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // Statically allocated shared memory
    __shared__ int s_a[SHMEM_SIZE];
    __shared__ int s_b[SHMEM_SIZE];

    int tmp = 0;
```

CONTINUED...

```
// Sweep tile across matrix
 for (int i = 0; i < K_padded; i += blockDim.x) {
 // Load in elements for this tile
 s_a[threadIdx.y * blockDim.x + threadIdx.x] = a[row * K + i + threadIdx.x];
 s_b[threadIdx.y * blockDim.x + threadIdx.x] =
 b[i * N + threadIdx.y * N + col];

 // Wait for both tiles to be loaded in before doing computation
 __syncthreads();

 // Do matrix multiplication on the small matrix
 for (int j = 0; j < blockDim.x; j++) {
 tmp +=
 s_a[threadIdx.y * blockDim.x + j] * s_b[j * blockDim.x + threadIdx.x];
 }

 // Wait for all threads to finish using current tiles before loading in new
 // ones
 __syncthreads();
 }

 // Write back results
 if (row < M && col < N) c[row * N + col] = tmp;
 }

 //Naive(Global Memory) Approach
 else{
 // Compute each thread's global row and column index
 int row = blockIdx.y * blockDim.y + threadIdx.y;
 int col = blockIdx.x * blockDim.x + threadIdx.x;

 // Iterate over row, and down column
 int temp = 0;
 for (int k = 0; k < K; k++) {
 // Accumulate results for a single element
 temp += a[row * K + k] * b[k * N + col];
 }
 if (row < M && col < N) c[row * N + col] = temp;
 }
 return;
}

// Checking result on the CPU
void verify_result(int* a, int* b, int* c) {
 clock_t start_time, end_time;
 printf("CPU verification has started\n");
 start_time = clock();

 for (int row = 0; row < M_padded; row++) {
 if (row >= M) continue;
 for (int col = 0; col < N_padded; col++) {
 if (col >= N) continue;
 int tmp = 0;
 for (int i = 0; i < K_padded; i++) {
 tmp += a[row * K + i] * b[i * N + col];
 }
```

CONTINUED...

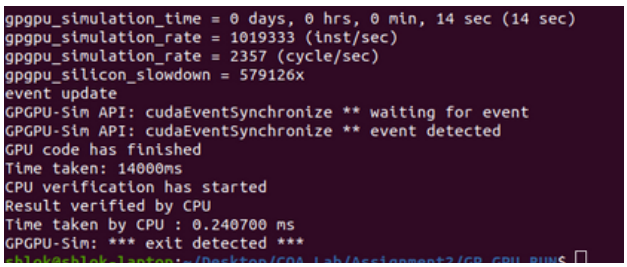
```
assert(tmp == c[row * N + col]);
 }
 }
 end_time = clock();
 cout << "Result verified by CPU\n";
 printf("Time taken by CPU : %f ms\n", (((double)end_time-
start_time)/CLOCKS_PER_SEC)*100);

}

int main() {

 if(SHARED){
 cout<<"Shared Memory Approach"<<endl;
 }
 else{
 cout<<"Naive Approach"<<endl;
 }

 // Size (in bytes) of matrix
 // MxN = MxK * KxN
 size_t bytes_a = M_padded * K_padded * sizeof(int);
 size_t bytes_b = K_padded * N_padded * sizeof(int);
 size_t bytes_c = M * N * sizeof(int);

 int *h_a = (int *)malloc(bytes_a);
 int *h_b = (int *)malloc(bytes_b);
 int *h_c = (int *)malloc(bytes_c);

 srand(time(NULL));
 // Initialize matrices
 for (int i = 0; i < M_padded; i++) {
 for (int j = 0; j < K_padded; j++) {
 if (i < M && j < K) h_a[i * K + j] = rand() % 100;
 }
 }

 for (int i = 0; i < K_padded; i++) {
 for (int j = 0; j < N_padded; j++) {
 if (i < K && j < N) h_b[i * N + j] = rand() % 100;
 }
 }

 // Allocate device memory
 int *d_a, *d_b, *d_c;
 cudaMalloc(&d_a, bytes_a);
 cudaMalloc(&d_b, bytes_b);
 cudaMalloc(&d_c, bytes_c);

 // Copy data to the device
 cudaMemcpy(d_a, h_a, bytes_a, cudaMemcpyHostToDevice);
 cudaMemcpy(d_b, h_b, bytes_b, cudaMemcpyHostToDevice);

 // Blocks per grid dimension (assumes THREADS divides M and N evenly)
 int BLOCKS_X = N_padded / THREADS;
 int BLOCKS_Y = M_padded / THREADS;
```

CONTINUED...

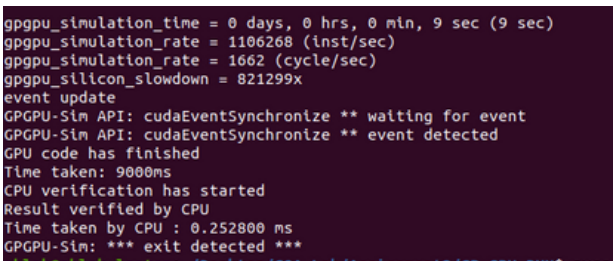
```
// Use dim3 structs for block and grid dimensions
 dim3 threads(THREADS, THREADS);
 dim3 blocks(BLOCKS_X, BLOCKS_Y);

 cudaEvent_t start, stop;
 float time_taken;
 cudaEventCreate(&start);
 cudaEventCreate(&stop);

 cout << "GPU code has started"<<endl;
 cudaEventRecord(start,0);

 // Launch kernel
 matrixMul<<<blocks, threads>>>(d_a, d_b, d_c);

 cudaEventRecord(stop,0);
 cudaEventSynchronize(stop);
 cudaEventElapsedTime(&time_taken, start, stop);
 cout << "GPU code has finished"<<endl;
 cout<<"Time taken: "<<time_taken<<"ms"<<endl;

 // Copy back to the host
 cudaMemcpy(h_c, d_c, bytes_c, cudaMemcpyDeviceToHost);

 // Check result
 verify_result(h_a, h_b, h_c);

 // Free memory on device
 cudaFree(d_a);
 cudaFree(d_b);
 cudaFree(d_c);

 return 0;
}
```

CODE ANALYSIS

OUTPUTS:



Naive Approach



Shared Memory Approach

ANALYSIS:

As it can be seen from the outputs, the simulation time of the Naive Approach is 14000ms whereas, of the Shared Memory Approach is only 9000ms.
Hence, the shared memory approach is faster by approximately 155%.
This is because the access time of shared memory is much lower as compared to that of global memory, due to which the lookup time reduces leading to faster execution.

THANK YOU!