Assignment 2

COA LAB

Team members:
1. Ishan Mardani (21CS02023)
2.Akshit Dudeja (21CS01026)
3. Tushar Joshi(21CS01078)
4. Vishnu Tirth Bysani (21CS01077)
5. Joshua Dias Barreto (21CS01075)

CONTENTS

KERNEL

Kernel is central component of an operating system that manages operations of computer and hardware. It basically manages operations of memory and CPU time. It is core component of an operating system. Kernel acts as a bridge between applications and data processing performed at hardware level using inter-process communication and system calls.
Kernel loads first into memory when an operating system is loaded and remains into memory until operating system is shut down again. It is responsible for various tasks such as disk management, task management, and memory management.
Kernel has a process table that keeps track of all active processes

OBJECTIVES OF KERNEL:

- To establish communication between user level application and hardware.
- To decide state of incoming processes.
- To control disk management.
- To control memory management.
- To control task management.

THREAD

Within a program, a **Thread** is a separate execution path. It is a lightweight process that the operating system can schedule and run concurrently with other threads. The operating system creates and manages threads, and they share the same memory and resources as the program that created them. This enables multiple threads to collaborate and work efficiently within a single program.

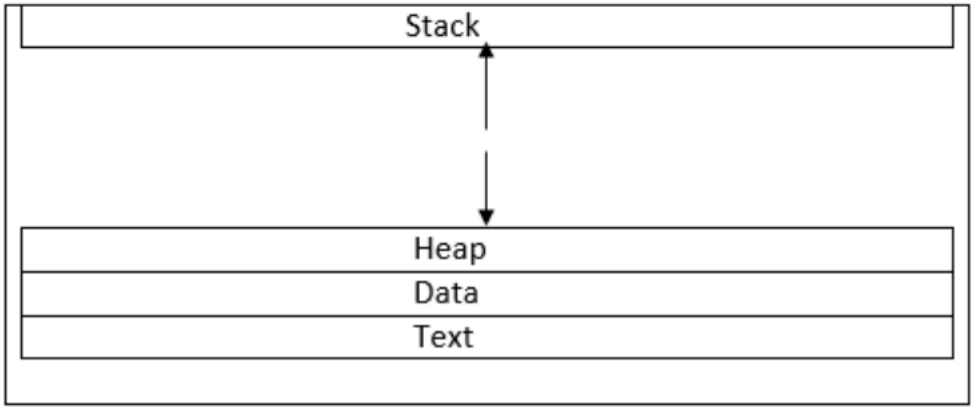
A thread is a single sequence stream within a process. Threads are also called lightweight processes as they possess some of the properties of processes. Each thread belongs to exactly one process. In an operating system that supports multithreading, the process can consist of many threads.

Why Do We Need Threads?

1 A thread is also known as a lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc. Multithreading is a technique used in operating systems to improve the performance and responsiveness of computer systems. Multithreading allows multiple threads (i.e., lightweight processes) to share the same resources of a single process, such as the CPU, memory, and I/O devices.

Process

A process is basically a program in execution which happens in a sequential manner. A program becomes a process when it is loaded into the memory. It can be divided into four sections- stack, heap , data ,text.



Process Diagram

Stack

The process stack stores temporary information such as method or function arguments, the return address, and local variables.process is dynamically allotted while it is running.

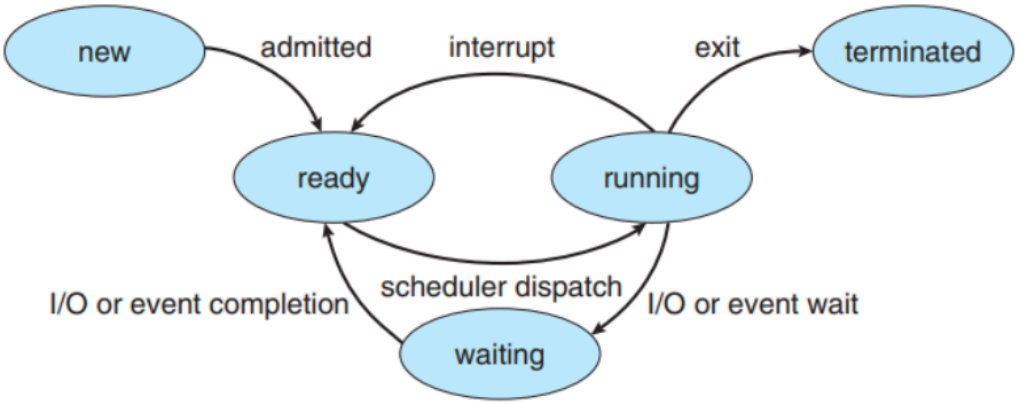
Heap

This is the memory where a Text. This consists of the information stored in the processor’s registers as well as the most recent activity indicated by the program counter's value.

Data

This section contains the global and the static variables.

Process Life Cycle



When a process executes, it passes through different states.

In general, a process can have one of the following five states at a time.

- New. The process is being created.
- Running. Instructions are being executed.
- Waiting. The process is waiting for some event to occur
- (such as an I/O completion or reception of a signal).
- Ready. The process is waiting to be assigned to a processor.
- Terminated. The process has finished execution.

SCHEDULER

There are two types of schedulers in operating system. These are Long Term Scheduler and Short-Term Scheduler.

Long Term Scheduler:

Long term scheduler is also known as job scheduler. It chooses the processes from the pool (secondary memory) and keeps them in the ready queue maintained in the primary memory. Long term Scheduler is necessary to ensure a degree of multiprogramming. It is important that the long-term scheduler make a careful selection of both I/O and CPU bound processes. I/O-bound tasks are which use much of their time in input and output operations while CPU-bound processes are which spend their time on the CPU. The job scheduler increases efficiency by maintaining a balance between the two.

Short-Term Scheduler

Short Term Scheduler is also known as CPU-Scheduler. It selects the process from the ready state to the running state. Short-term scheduler only selects the process to schedule, it does not load the process on running. The Job of the short-term scheduler can be very critical in the sense that if it selects job whose CPU burst time is very high then all the jobs after that, will have to wait in the ready queue for a very long time. The dispatcher is responsible for loading the process selected by the Short-term scheduler on the CPU.

Job of the dispatcher:

1. Switching context.
2. Switching to user mode.
3. Jumping to the proper location in the newly loaded program.

Medium Term Scheduler

Medium term scheduler takes care of the swapped-out processes. If the running state processes needs some IO time for the completion, then there is a need to change its state from running to waiting.Medium term scheduler is used for this purpose. It removes the process from the running state to make room for the other processes. Such processes are the swapped-out processes and this procedure is called swapping. The medium-term scheduler is responsible for suspending and resuming the processes.
It reduces the degree of multiprogramming.The swapping is necessary to have a perfect mix of processes in the ready queue.
There are various algorithms which are used by the Operating System to schedule the processes on the processor in an efficient way.

The Purpose of a Scheduling algorithm

1. Maximum CPU utilization
2. Fare allocation of CPU
3. Maximum throughput
4. Minimum turnaround time
5. Minimum waiting time
6. Minimum response time

There are some algorithms used in scheduling problem such as:

1. First Come First Serve

It is the simplest algorithm to implement. The process with the minimal arrival time will get the CPU first. The lesser the arrival time, the sooner will the process gets the CPU. It is the non-pre-emptive type of scheduling.

2. Round Robin

In the Round Robin scheduling algorithm, the OS defines a time quantum (slice). All the processes will get executed in the cyclic way. Each of the process will get the CPU for a small amount of time (called time quantum) and then get back to the ready queue to wait for its next turn. It is a pre-emptive type of scheduling.

3. Shortest Job First

The job with the shortest burst time will get the CPU first. The lesser the burst time, the sooner will the process get the CPU. It is the non-pre-emptive type of scheduling.

4. Shortest remaining time first

It is the pre-emptive form of SJF. In this algorithm, the OS schedules the Job according to the remaining time of the execution.

5. Priority based scheduling

In this algorithm, the priority will be assigned to each of the processes. The higher the priority, the sooner will the process get the CPU. If the priority of the two processes is same then they will be scheduled according to their arrival time.14

6. Highest Response Ratio Next

In this scheduling Algorithm, the process with highest response ratio will be scheduled next. This reduces the starvation in the system.

Thread-Block

A thread block is a programming abstraction that represents a group of threads that can be executed serially or in parallel. For better process and data mapping, threads are grouped into thread blocks. Multiple blocks are combined to form a grid. All the blocks in the same grid contain the same number of threads. The number of threads in a block is limited, but grids can be used for computations that require a large number of thread blocks to operate in parallel and to use all available multiprocessors. As many parallel applications involve multidimensional data, it is convenient to organize thread blocks into 1D, 2D or 3D arrays of threads.

GPU-MEMORY-HIERARCHY

SRAM:

Static random-access memory (static RAM or SRAM) is a type of random-access memory (RAM) that uses latching circuitry (flip-flop) to store each bit. SRAM is volatile memory; data is lost when power is removed.
SRAM will hold its data permanently in the presence of power, while data in DRAM decays in seconds and thus must be periodically refreshed.
There are two key features to SRAM - Static Random Access Memory, and these set it out against other types of memory that are available:

- The data is held statically: This means that the data is held in the semiconductor memory without the need to be refreshed as long as the power is applied to the memory.
- SRAM memory is a form of random-access memory: A random access memory is one in which the locations in the semiconductor memory can be written to or read from in any order, regardless of the last memory location that was accessed. Access to the SRAM memory cell is enabled by the Word Line. This controls the two access control transistors which

control whether the cell should be connected to the bit lines. These two lines are used to transfer data for both read and write operations.

DRAM

As the name DRAM, or dynamic random access memory, implies, this form of memory technology is a type of random access memory. It stores each bit of data on a small capacitor within the memory cell. The capacitor can be either charged or discharged and this provides the two states,"1" or "0" for the cell.
Since the charge within the capacitor leaks, it is necessary to refresh each memory cell periodically. This refresh requirement gives rise to the term dynamic - static memories do not have a need to be refreshed.

The need to refresh DRAM demands more complicated circuitry and timing than SRAM. This is offset by the structural simplicity of DRAM memory cells: only one transistor and a capacitor are required per bit, compared to four or six transistors in SRAM. This allows DRAM to reach very high densities with a simultaneous reduction in cost per bit. Refreshing the data consumes power and a variety of techniques are used to manage the overall power consumption.

SHARED MEMORY

Shared memory is a type of memory that is physically located on the GPU's multiprocessor and is shared among threads within a thread block. It is a fast and low-latency memory space that enables efficient communication and data sharing between threads executing on the same multiprocessor.

Each thread block in CUDA has its own dedicated shared memory. Threads within the block can read and write data to this shared memory, allowing them to cooperate and exchange intermediate results. Sharing data in shared memory is much faster than accessing global memory (main GPU memory) since it avoids the high-latency overhead associated with global memory accesses.

CONSTANT MEMORY:

Constant memory is another type of on-chip memory available in CUDA, specifically designed for storing read-only data that remains constant across all threads in a kernel launch. This memory space is read-only, meaning that threads cannot write to it.

Constant memory is optimized for broadcasting the same data value to all threads in a thread block. When multiple threads access the same constant memory location, the memory controller broadcasts the value to all threads in a single memory transaction. This property can be beneficial for certain read-only data, as it reduces memory traffic and improves memory access efficiency.

Developers can use constant memory when they have data that does not change throughout a kernel launch, such as lookup tables or constant parameters used in computations.

SIMD

SIMD (Single Instruction, Multiple Data) is a type of parallel computing architecture that allows a single instruction to operate on multiple data elements simultaneously. In a SIMD architecture, a processor can perform the same operation on multiple data items in parallel, thereby increasing the throughput of data processing.

The SIMD model is designed to take advantage of data-level parallelism, where multiple data elements are processed at the same time using a single instruction. This is particularly useful for tasks that involve repetitive computations on large sets of data, such as multimedia processing, image and video processing, scientific simulations, and certain types of mathematical computations.

In a SIMD processor, the data is divided into "vectors" or "chunks," and each vector contains multiple data elements of the same data type (e.g., integers, floating-point numbers). The SIMD processor processes these vectors in parallel by applying the same operation to each element of the vector simultaneously. This approach contrasts with Single Instruction, Single Data (SISD) processors, where a single operation is performed on a single data item at a time.

Modern CPUs and GPUs often incorporate SIMD instructions and hardware units to accelerate specific tasks that can benefit from data parallelism.

Warp

In computer architecture, "warp" typically refers to a group of threads in a parallel computing environment, particularly in the context of Graphics Processing Units (GPUs). It is a fundamental concept in modern GPU architectures and is closely related to the concept of SIMD (Single Instruction, Multiple Data) execution.

GPUs are designed to handle massive parallelism, enabling them to perform computations on multiple data elements simultaneously. To achieve this, GPUs group threads into warps, and each warp is executed together by a

single GPU core. The number of threads in a warp can vary depending on the GPU architecture, but a common size is 32 threads per warp.

When an application or program is running on a GPU, it is divided into numerous threads. These threads are organized into blocks, and each block contains multiple warps. The GPU scheduler assigns warps to available cores for execution, and the cores execute the same instruction on each thread within a warp in a lockstep manner. This meansthat all threads within a warp perform the same operation at the same time, but on different data.
The benefit of using warps and SIMD execution is that it allows the GPU to efficiently process a large number of data elements in parallel, leveraging the massive parallelism provided by the GPU's architecture. However, it's crucial for developers to design their algorithms and code with this parallel execution model in mind to fully exploit the GPU’s capabilities and avoid performance bottlenecks. Properly utilizing warps can lead to significant speedup in applications that can be parallelized effectively.

MATRIX MULTIPLICATION CODES

Code 1

```
// This program computes a simple version of matrix multiplication

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::generate;
using std::vector;

__global__ void matrixMul(const int *a, const int *b, int *c, int N) {
  // 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
  c[row * N + col] = 0;
  for (int k = 0; k < N; k++) {
    // Accumulate results for a single element
    c[row * N + col] += a[row * N + k] * b[k * N + col];
```

```
  }
}

// Check result on the CPU
void verify_result(vector<int> &a, vector<int> &b, vector<int> &c, int N) {
  // For every row...
  for (int i = 0; i < N; i++) {
    // For every column...
    for (int j = 0; j < N; j++) {
      // For every element in the row-column pair
      int tmp = 0;
      for (int k = 0; k < N; k++) {
        // Accumulate the partial results
        tmp += a[i * N + k] * b[k * N + j];
      }

      // Check against the CPU result
      assert(tmp == c[i * N + j]);
    }
  }
}

int main() {
  // Matrix size of 1024 x 1024;
  int N = 1 << 7;

  // Size (in bytes) of matrix
  size_t bytes = N * N * sizeof(int);
```

```
// Host vectors
vector<int> h_a(N * N);
vector<int> h_b(N * N);
vector<int> h_c(N * N);

// Initialize matrices
generate(h_a.begin(), h_a.end(), []() { return rand() % 100; });
generate(h_b.begin(), h_b.end(), []() { return rand() % 100; });

// Allocate device memory
int *d_a, *d_b, *d_c;
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

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

// Threads per CTA dimension
int THREADS = 32;

// Blocks per grid dimension (assumes THREADS divides N evenly)
int BLOCKS = N / THREADS;

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

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

    // Copy back to the host
    cudaMemcpy(h_c.data(), d_c, bytes, cudaMemcpyDeviceToHost);

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

    cout << "COMPLETED SUCCESSFULLY\n";

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

    return 0;
}
```

EXPLANATION

1. Matrix Multiplication Kernel (`matrixMul`):

The `matrixMul` function is the GPU kernel responsible for performing matrix multiplication in parallel. It takes three input arguments:

- `a`: Pointer to the first input matrix `a`.
- `b`: Pointer to the second input matrix `b`.
- `c`: Pointer to the output matrix `c`.27
- `N`: The size of the square matrices.

It computes the global row and column indices for the thread, and then each thread calculates a single element of the output matrix `c` using the formula `c[row * N + col] = a[row * N + k] * b[k * N + col]`. The multiplication is done for all values of `k` from 0 to `N-1`, and the partial results are accumulated to calculate the final result.

2. Verification Function (`verify_result`):

The `verify_result` function checks the correctness of the GPU-computed result against the CPU-computed result. It takes three input vectors `a`, `b`, and `c`, along with the size `N`. It performs the matrix multiplication on the CPU using three nested loops and stores the result in a temporary variable `tmp`. Then, it compares the GPU result stored in `c` with the CPU-computed `tmp` for each element, and if any discrepancy is found, it will throw an assertion error.

3. Main Function (`main`):

In the `main` function:

- It defines the size of the square matrices `N` as 1024 (1 << 10) and calculates the total number of bytes required for the matrices.
- Three host vectors `h_a`, `h_b`, and `h_c` are declared to store the input matrices `a`, `b`, and the output matrix `c`.
- The input matrices `a` and `b` are initialized with random integer values between 0 and 99 using the `generate` function and lambda expressions.
- Device memory is allocated for the matrices `d_a`, `d_b`, and `d_c` using `cudaMalloc`.
- The data from host vectors `h_a` and `h_b` are copied to the device memory using `cudaMemcpy`.
- The number of threads per block (`THREADS`) and the number of blocks per grid (`BLOCKS`) are calculated based on the matrix size `N`.

- The kernel is launched with the specified block and thread dimensions using `matrixMul<<<blocks, threads>>>(d_a, d_b, d_c, N)`.28
- The result matrix `c` is copied back from device memory to the host vector `h_c` using `cudaMemcpy`.
- The `verify_result` function is called to check the correctness of the GPU-computed result.
- If the verification passes, the program prints "COMPLETED SUCCESSFULLY."
- Device memory is freed using `cudaFree`.

4. Compilation and Execution:

To run this program, you need an NVIDIA GPU with CUDA support and the NVIDIA CUDA Toolkit installed. Compile the program with a CUDA-capable C++ compiler (e.g., nvcc) and execute the resulting executable.
Please note that this program assumes the matrix size (`N`) is a power of 2 and that the number of threads per block (`THREADS`) evenly divides the matrix size (`N`). For more generic matrix multiplication code, you may need to handle cases where these assumptions are not met.

Terminal Output

```
gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 22 sec (22 sec)
gpgpu_simulation_rate = 668020 (inst/sec)
gpgpu_simulation_rate = 2114 (cycle/sec)
gpgpu_silicon_slowdown = 645695x
COMPLETED SUCCESSFULLY
GPGPU-Sim: *** exit detected ***
```

Code 2

```
// This program computes matrix multiplication using shared memory tiling
// By: Nick from CoffeeBeforeArch

#include <algorithm>
#include <cassert>
#include <cstdlib>
#include <functional>
#include <iostream>
#include <vector>

using std::cout;
using std::generate;
using std::vector;

// Pull out matrix and shared memory tile size
const int N = 1 << 7;
const int SHMEM_SIZE = 1 << 7;

__global__ void matrixMul(const int *a, const int *b, int *c) {
  // 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];
```

```
  // Accumulate in temporary variable
  int tmp = 0;

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

```
// Check result on the CPU
void verify_result(vector<int> &a, vector<int> &b, vector<int> &c) {
  // For every row...
  for (int i = 0; i < N; i++) {
    // For every column...
    for (int j = 0; j < N; j++) {
      // For every element in the row-column pair
      int tmp = 0;
      for (int k = 0; k < N; k++) {
        // Accumulate the partial results
        tmp += a[i * N + k] * b[k * N + j];
      }

      // Check against the CPU result
      assert(tmp == c[i * N + j]);
    }
  }
}

int main() {
  // Size (in bytes) of matrix
  size_t bytes = N * N * sizeof(int);

  // Host vectors
  vector<int> h_a(N * N);
  vector<int> h_b(N * N);
  vector<int> h_c(N * N);

  // Initialize matrices
```

```
generate(h_a.begin(), h_a.end(), []() { return rand() % 100; });
generate(h_b.begin(), h_b.end(), []() { return rand() % 100; });

// Allocate device memory
int *d_a, *d_b, *d_c;
cudaMalloc(&d_a, bytes);
cudaMalloc(&d_b, bytes);
cudaMalloc(&d_c, bytes);

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

// Threads per CTA dimension
int THREADS = 32;

// Blocks per grid dimension (assumes THREADS divides N evenly)
int BLOCKS = N / THREADS;

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

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

// Copy back to the host
cudaMemcpy(h_c.data(), d_c, bytes, cudaMemcpyDeviceToHost);
```

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

  cout << "COMPLETED SUCCESSFULLY\n";

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

  return 0;
}
```

EXPLANATION:

1. Matrix Multiplication Kernel (`matrixMul`):

The `matrixMul` function is the GPU kernel responsible for performing matrix multiplication in parallel. It takes three input arguments:

- `a`: Pointer to the first input matrix `a`.
- `b`: Pointer to the second input matrix `b`.
- `c`: Pointer to the output matrix `c`.

It computes the global row and column indices for the thread, and then each thread loads a tile of the input matrices `a` and `b` into shared memory `s_a` and `s_b`, respectively. The size of the shared memory tile is defined as `SHMEM_SIZE`, which is 1024 in this case. The kernel uses two nested loops

to sweep a tile across the matrix, performing matrix multiplication on the small tile loaded in shared memory. The intermediate results are accumulated in a temporary variable `tmp`. After each iteration of the outer loop, the kernel waits for all threads to finish using the current tiles before loading in new ones using `__syncthreads()`.

2. Verification Function (`verify_result`):

The `verify_result` function checks the correctness of the GPU-computed result against the CPU-computed result. It takes three input vectors `a`, `b`, and `c`, and it performs the matrix multiplication on the CPU using three nested loops and stores the result in a temporary variable `tmp`. Then, it compares the GPU result stored in `c` with the CPU- computed `tmp` for each element, and if any discrepancy is found, it will throw an assertion error.

3. Main Function (`main`):

In the `main` function:

- The size of the square matrices `N` is defined as 1024 (1 << 10).
- The host vectors `h_a`, `h_b`, and `h_c` are declared to store the input matrices `a`, `b`, and the output matrix `c`.
- The input matrices `a` and `b` are initialized with random integer values between 0 and 99 using the `generate` function and lambda expressions.
- Device memory is allocated for the matrices `d_a`, `d_b`, and `d_c` using `cudaMalloc`.
- The data from host vectors `h_a` and `h_b` are copied to the device memory using `cudaMemcpy`.
- The number of threads per block (`THREADS`) and the number of blocks per grid (`BLOCKS`) are calculated based on the matrix size `N`.
- The kernel is launched with the specified block and thread dimensions using `matrixMul<<<blocks, threads>>>(d_a, d_b, d_c)`.35

- The result matrix `c` is copied back from device memory to the host vector `h_c` using `cudaMemcpy`.
- The `verify_result` function is called to check the correctness of the GPU-computed result.
- If the verification passes, the program prints "COMPLETED SUCCESSFULLY."
- Device memory is freed using `cudaFree`.

4. Compilation and Execution:

To run this program, you need an NVIDIA GPU with CUDA support and the NVIDIA CUDA Toolkit installed. Compile the program with a CUDA-capable C++ compiler (e.g., nvcc) and execute the resulting executable.

Please note that this program assumes the matrix size (`N`) is fixed at 1024. For more generic matrix multiplication code, you may need to handle cases with different matrix sizes. Additionally, this implementation assumes that the number of threads per block (`THREADS`) evenly divides the matrix size (`N`). For non-divisible cases, you would need to handle the boundary conditions accordingly.

Terminal Output

```
gpgpu_simulation_time = 0 days, 0 hrs, 0 min, 11 sec (11 sec)
gpgpu_simulation_rate = 1178158 (inst/sec)
gpgpu_simulation_rate = 1811 (cycle/sec)
gpgpu_silicon_slowdown = 753727x
```