Assignment 3

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)

Matrix Multiplication Code

Global Memory

```
#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() {
 freopen("GlobalTL.txt","w",stdout);
 // 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;
}
```

Shared Memory

```
#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;
}
```

Warp Schedulers

1. Greedy then Oldest (GTO)

- Global Memory

Runtime = 20s

Instruction per Second= 734822 (inst/sec)

Cycle per Second = 2326 (cycle/sec)

IPC = Instruction per Second/Cycle per Second

IPC = 734822/2326 = 315.86

- Shared Memory

Runtime = 16s

Instruction per Second= 1166336 (inst/sec)

Cycle per Second = 1266 (cycle/sec)

IPC = Instruction per Second/Cycle per Second

IPC = 1166336/1266 = 921.23

2. Loose Round Robin (LRR)

- Global Memory

Runtime = 21s

Instruction per Second= 699830 (inst/sec)

Cycle per Second = 2279 (cycle/sec)

IPC = Instruction per Second/Cycle per Second

IPC = 699830/2279 = 307.03

- Shared Memory

Runtime = 15s

Instruction per Second= 1244091 (inst/sec)

Cycle per Second = 1373 (cycle/sec)

IPC = Instruction per Second/Cycle per Second

IPC = 1244091/1373 = 906.02

3. Two Level (two_level_active:6:0:1)

- Global Memory

Runtime = 23s

Instruction per Second= 638976 (inst/sec)

Cycle per Second = 1992 (cycle/sec)

IPC = Instruction per Second/Cycle per Second

IPC = 638976/1992 = 320.75

- Shared Memory

Runtime = 16s

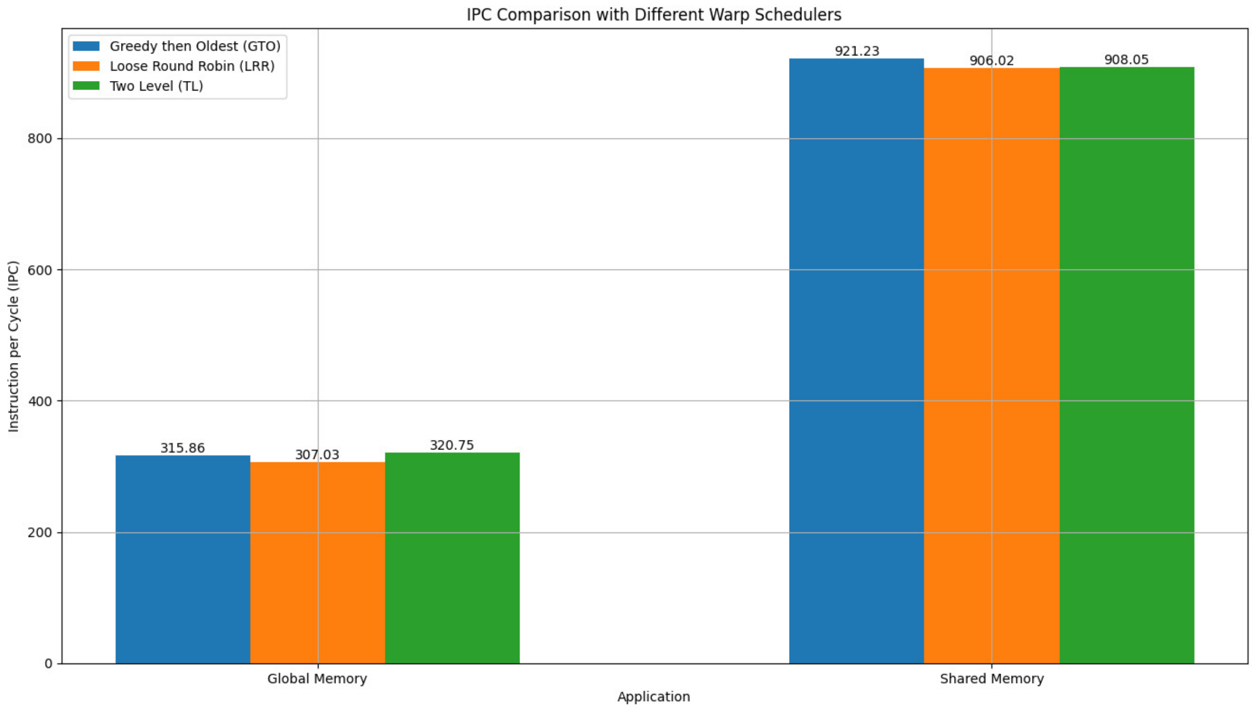
Instruction per Second= 1166366 (inst/sec)

Cycle per Second = 1284 (cycle/sec)

IPC = Instruction per Second/Cycle per Second

IPC = 1166366/1284 = 908.05

IPC Comparison with Different Warp Schedulers



Cache

A cache miss occurs when a computer processor needs data that is not currently stored in its fast cache memory, so it has to retrieve it from a slower main memory. Cache misses can slow down performance, so computer systems use caching strategies to minimize their impact.

When a cache miss occurs, the caching system needs to search further into the CPU memory units to find the stored information. In other words, if a cache can't find the requested data in the L1 cache, it will then search for it in L2.

Cache Hit Ratio

Cache hit ratio = Cache misses/(Cache hits + cache misses) x 100

L1

Shared TL

L1D_total_cache_accesses = 67584
L1D_total_cache_misses = 67584
L1D_total_cache_miss_rate = 1.0000

Shared LRR

L1D_total_cache_accesses = 67584
L1D_total_cache_misses = 67584
L1D_total_cache_miss_rate = 1.0000

Shared GTO

L1D_total_cache_accesses = 591872
L1D_total_cache_misses = 436737
L1D_total_cache_miss_rate = 0.7379

Global TL

L1D_total_cache_accesses = 591872
L1D_total_cache_misses = 435018
L1D_total_cache_miss_rate = 0.7350

Global GTO

L1D_total_cache_accesses = 591872
L1D_total_cache_misses = 434299
L1D_total_cache_miss_rate = 0.7338

Global LRR

L1D_total_cache_accesses = 591872
L1D_total_cache_misses = 436737
L1D_total_cache_miss_rate = 0.7379

L2

Shared TL

L2_total_cache_accesses = 65536
L2_total_cache_misses = 2048
L2_total_cache_miss_rate = 0.0312

Shared LRR

L2_total_cache_accesses = 65536
L2_total_cache_misses = 2048
L2_total_cache_miss_rate = 0.0312

Shared GTO

L2_total_cache_accesses = 65536
L2_total_cache_misses = 2048
L2_total_cache_miss_rate = 0.0312

Global TL

L2_total_cache_accesses = 280576
L2_total_cache_misses = 2048
L2_total_cache_miss_rate = 0.0073

Global GTO

L2_total_cache_accesses = 280576
L2_total_cache_misses = 2048
L2_total_cache_miss_rate = 0.0073

Global LRR

L2_total_cache_accesses = 280576
L2_total_cache_misses = 2048
L2_total_cache_miss_rate = 0.0073

Latency

Latency refers to the period of time that elapses between the initiation of a request or action and the corresponding response or outcome. The metric is quantified in units of milliseconds and is precisely characterized as the cumulative duration encompassing the interval between an input or directive and the intended output.

The MF (maximum memory fetch) latency of a DRAM module is contingent upon several factors, such as the particular type of DRAM utilized, the operating frequency at which it functions, and the design of the memory controller. The term "DRAM latency" pertains to the duration required for a memory module to fulfill a memory request.

DRAM

Global TL

averagemflatency = 198
avg_icnt2mem_latency = 33
avg_icnt2sh_latency = 2

Global LRR

averagemflatency = 197
avg_icnt2mem_latency = 32
avg_icnt2sh_latency = 2

GLobal GTO

averagemflatency = 199
avg_icnt2mem_latency = 34
avg_icnt2sh_latency = 2

Shared TL

averagemflatency = 191
avg_icnt2mem_latency = 27
avg_icnt2sh_latency = 2

Shared GTO

averagemflatency = 192
avg_icnt2mem_latency = 27
avg_icnt2sh_latency = 2

Shared LRR

averagemflatency = 193
avg_icnt2mem_latency = 28
avg_icnt2sh_latency = 2

TREND

DRAM

Based on the evidence presented, it can be inferred that the global implementation exhibits higher latency compared to the shared version, thereby indicating the enhanced performance of the shared memory code.

L1

L1 Caches are only accessed by threads from the same Streaming Multiprocessors (SM). Hence the number of accesses are more in Shared Memory.

L2

L2 Caches are accessed by all threads of the kernel. Hence the number of accesses are more in Global Memory.