**Lab Exercise 11.3 – Optimizing Memory Usage in CUDA**

**Objective:**

* Understand and apply memory optimization techniques in CUDA.
* Learn how to minimize global memory access and utilize faster memory types like shared memory.
* Observe performance benefits of optimizing memory usage.

**1. Key Optimization Techniques**

| **Technique** | **Benefit** |
| --- | --- |
| Use of shared memory | Faster access compared to global memory |
| Coalesced global memory access | Reduces memory transaction overhead |
| Minimizing memory allocation | Lowers overhead and improves efficiency |
| Avoiding bank conflicts | Ensures parallel efficiency in shared memory |

**2. Program: Using Shared Memory for Optimization**

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void sharedMemoryAdd(int \*a, int \*b, int \*c, int N) {

\_\_shared\_\_ int s\_a[256];

\_\_shared\_\_ int s\_b[256];

int tid = threadIdx.x;

if (tid < N) {

// Load global memory into shared memory

s\_a[tid] = a[tid];

s\_b[tid] = b[tid];

\_\_syncthreads(); // Synchronize threads after loading

// Perform computation in shared memory

c[tid] = s\_a[tid] + s\_b[tid];

}

}

int main() {

const int N = 256;

int h\_a[N], h\_b[N], h\_c[N];

int \*d\_a, \*d\_b, \*d\_c;

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

h\_a[i] = i;

h\_b[i] = i \* 2;

}

cudaMalloc((void\*\*)&d\_a, N \* sizeof(int));

cudaMalloc((void\*\*)&d\_b, N \* sizeof(int));

cudaMalloc((void\*\*)&d\_c, N \* sizeof(int));

cudaMemcpy(d\_a, h\_a, N \* sizeof(int), cudaMemcpyHostToDevice);

cudaMemcpy(d\_b, h\_b, N \* sizeof(int), cudaMemcpyHostToDevice);

sharedMemoryAdd<<<1, N>>>(d\_a, d\_b, d\_c, N);

cudaMemcpy(h\_c, d\_c, N \* sizeof(int), cudaMemcpyDeviceToHost);

std::cout << "First 10 results:\n";

for (int i = 0; i < 10; i++)

std::cout << h\_c[i] << " ";

std::cout << std::endl;

cudaFree(d\_a);

cudaFree(d\_b);

cudaFree(d\_c);

return 0;

}

**3. Explanation**

* \_\_shared\_\_ int s\_a[256], s\_b[256];  
  Allocates shared memory arrays accessible by all threads in a block.
* s\_a[tid] = a[tid];  
  Loads global memory into faster shared memory to reduce access time.
* \_\_syncthreads();  
  Ensures all threads have completed memory loads before computation begins.
* Computation is performed using shared memory, improving performance due to faster access and reduced global memory transactions.

**4. Compile and Run**

nvcc -o optimize\_mem optimize\_mem.cu

./optimize\_mem

Expected output:

First 10 results:

0 3 6 9 12 15 18 21 24 27

**5. Summary**

* Shared memory dramatically improves performance when multiple threads access the same data.
* Always minimize global memory access in favor of shared memory for intra-block operations.
* Proper synchronization is essential when using shared memory.