**Lab Exercise 10.1 – Implementing Basic Error Handling in CUDA**

**Objective:**

* Learn how to detect and handle errors in CUDA programs.
* Understand how to use built-in CUDA error handling functions.
* Practice writing robust CUDA code by checking return values and kernel errors.

**1. Why Error Handling is Important in CUDA?**

In CUDA, many functions return error codes that must be checked to ensure the GPU code is working correctly. Common issues include:

* Invalid memory access
* Kernel launch failures
* Synchronization errors

**2. Program: Basic Error Handling Example**

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void kernelWithError() {

int idx = threadIdx.x;

// Intentional out-of-bounds access to cause an error (invalid write)

int \*p = nullptr;

if (idx == 0) p[0] = 10;

}

void checkCudaError(cudaError\_t err, const char\* message) {

if (err != cudaSuccess) {

std::cerr << "CUDA Error: " << message << " - " << cudaGetErrorString(err) << std::endl;

exit(EXIT\_FAILURE);

}

}

int main() {

// Launch kernel with 1 block and 1 thread

kernelWithError<<<1, 1>>>();

// Check for kernel launch error

checkCudaError(cudaGetLastError(), "Kernel launch");

// Check for runtime error (after execution)

checkCudaError(cudaDeviceSynchronize(), "Kernel execution");

std::cout << "Program completed successfully." << std::endl;

return 0;

}

**3. Explanation (Line-by-Line)**

* kernelWithError: A dummy kernel that intentionally writes to a nullptr to cause an error.
* checkCudaError: A helper function that checks the result of a CUDA call and prints an error message if it fails.
* cudaGetLastError: Returns the last error from a kernel launch.
* cudaDeviceSynchronize: Waits for the kernel to complete and catches runtime errors.
* If any error occurs, the program exits immediately with a message.

**4. Compilation and Execution**

nvcc -o cuda\_error\_handling cuda\_error\_handling.cu

./cuda\_error\_handling

You should see an error like:

CUDA Error: Kernel execution - an illegal memory access was encountered

**5. Best Practices in CUDA Error Handling**

* Always check return values from CUDA API calls.
* Use cudaGetLastError() immediately after kernel launches.
* Use cudaDeviceSynchronize() to catch runtime execution errors.
* Wrap repetitive error checks in a helper function (as shown).

**6. Conclusion**

This lab taught you:

* How to implement basic CUDA error handling.
* The importance of checking for both kernel launch and execution errors.
* How to improve program stability using consistent error checking.

**Implementing Various Error Handling Concepts in CUDA**

**Objective:**

**Exercise 1: Basic Error Checking with cudaError\_t**

**Task:** Allocate memory on the device and check for allocation errors.

#include <iostream>

#include <cuda\_runtime.h>

int main() {

int \*d\_array;

cudaError\_t err = cudaMalloc((void\*\*)&d\_array, 1e10); // Large memory to force error

if (err != cudaSuccess) {

std::cerr << "CUDA Error: " << cudaGetErrorString(err) << std::endl;

} else {

std::cout << "Memory allocation successful." << std::endl;

cudaFree(d\_array);

}

return 0;

}

**Exercise 2: Kernel Launch and Error Check**

**Task:** Launch a dummy kernel and check for errors post-launch and after synchronization.

**Without Error**

#include <iostream>

#include <cuda\_runtime.h>

// Kernel with an intentional out-of-bounds memory access

\_\_global\_\_ void faultyKernel(int \*d\_array) {

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

// Access an element far beyond the allocated memory

// This will cause an illegal memory access

int value = d\_array[idx + 10000]; // Forced error

printf("Thread %d read value: %d\n", idx, value);

}

int main() {

const int N = 4;

int \*d\_array;

// Allocate a small device array

cudaError\_t err = cudaMalloc((void\*\*)&d\_array, N \* sizeof(int));

if (err != cudaSuccess) {

std::cerr << "Memory allocation failed: " << cudaGetErrorString(err) << std::endl;

return -1;

}

// Launch kernel with a few threads

faultyKernel<<<1, N>>>(d\_array);

// Check for kernel launch error

err = cudaGetLastError();

if (err != cudaSuccess) {

std::cerr << "Kernel launch error: " << cudaGetErrorString(err) << std::endl;

}

// Check for kernel execution/runtime error

err = cudaDeviceSynchronize();

if (err != cudaSuccess) {

std::cerr << "Kernel execution error: " << cudaGetErrorString(err) << std::endl;

} else {

std::cout << "Kernel execution completed without error (unexpected)." << std::endl;

}

// Free memory and reset device

cudaFree(d\_array);

cudaDeviceReset();

return 0;

}

**With Error**

#include <iostream>

#include <cuda\_runtime.h>

\_\_global\_\_ void faultyKernel(int \*d\_array) {

int idx = threadIdx.x + blockIdx.x \* blockDim.x;

// This access is guaranteed to be invalid

int value = d\_array[(1 << 30) + idx]; // Large invalid access

printf("Thread %d read value: %d\n", idx, value);

}

int main() {

const int N = 4;

int \*d\_array;

// Allocate a small device array

cudaError\_t err = cudaMalloc((void\*\*)&d\_array, N \* sizeof(int));

if (err != cudaSuccess) {

std::cerr << "Memory allocation failed: " << cudaGetErrorString(err) << std::endl;

return -1;

}

// Launch kernel with a few threads

faultyKernel<<<1, N>>>(d\_array);

// Check for kernel launch error

err = cudaGetLastError();

if (err != cudaSuccess) {

std::cerr << "Kernel launch error: " << cudaGetErrorString(err) << std::endl;

}

// Check for kernel execution/runtime error

err = cudaDeviceSynchronize();

if (err != cudaSuccess) {

std::cerr << "Kernel execution error: " << cudaGetErrorString(err) << std::endl;

} else {

std::cout << "Kernel execution completed without error (unexpected)." << std::endl;

}

// Free memory and reset device

cudaFree(d\_array);

cudaDeviceReset();

return 0;

}

**Exercise 3: Handling Device-side Assertion Failures**

**Task:** Use assert() in a kernel and enable runtime device assertions.

#include <iostream>

#include <assert.h>

\_\_global\_\_ void assertKernel(int \*data) {

int idx = threadIdx.x;

assert(data[idx] >= 0); // Fails if any element is negative

}

int main() {

int h\_data[4] = {1, -2, 3, 4};

int \*d\_data;

cudaMalloc((void\*\*)&d\_data, sizeof(h\_data));

cudaMemcpy(d\_data, h\_data, sizeof(h\_data), cudaMemcpyHostToDevice);

assertKernel<<<1, 4>>>(d\_data);

cudaError\_t err = cudaDeviceSynchronize();

if (err != cudaSuccess) {

std::cerr << "Assertion failure or runtime error: " << cudaGetErrorString(err) << std::endl;

}

cudaFree(d\_data);

return 0;

}

**Expected Outcomes:**

* Ability to detect and interpret CUDA runtime errors.
* Implement reusable macros for error checking.
* Understand kernel launch, sync, and memory operation errors.
* Debug device-side assertion failures.