Lab0: CUDA Programming 101, Arrays a Matrices

Introduction **1**
Objectives **1**
Exercices **1**
Get GPU information 1
Sequential Code for Array Addition with Array Initialization Function 2
Parallel Array Addition with CUDA 2
Sequential vs. Parallel Matrix Addition 3

Introduction

Welcome to your first CUDA learning lab: Arrays and Matrices! Here's a simple explanation of the concept: We're going to work with CUDA, which helps us utilize graphics cards (GPUs) to speed up mathematical operations. Today, we focus on adding two arrays together, similar to how you might add numbers in basic arithmetic. First, we write regular C code to perform array addition. Then, we convert this code into CUDA programs to benefit from the parallelism provided by GPUs. Finally, we optimize our CUDA programs by initializing arrays simultaneously, making our code more efficient. To summarize, CUDA lets us process large amounts of data quickly by utilizing the massive computational power of modern graphics cards. It's perfect for tasks such as numerical computations, image processing, and even some types of artificial intelligence. With CUDA, we can achieve impressive speeds compared to traditional CPU-based methods. Happy coding!

Objectives

By completing this CUDA lab, you'll acquire practical skills in: Writing CUDA kernels to execute parallel array and matrix addition efficiently. Optimizing performance by parallelizing array and matrix initialization. Assessing the effectiveness of sequential versus parallel implementations. Understanding the advantages and limitations of GPU-accelerated computing. This lab offers a hands-on opportunity to develop proficiency in CUDA programming, enabling you to tackle complex problems involving numerical computations, image processing, and machine learning. As you progress through the exercises, you'll become increasingly adept at designing and executing GPU-optimized solutions. Good luck!

Exercices

Get GPU information

1. Copy the following code and name it with GPU_info.cu

```cpp
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
    int deviceCount;
    cudaGetDeviceCount(&deviceCount);

    if (deviceCount == 0) {
        printf("No CUDA-capable devices detected\n");
        return 0;
    }

    printf("Number of CUDA devices: %d\n", deviceCount);

    for (int i = 0; i < deviceCount; ++i) {
        cudaDeviceProp deviceProp;
        cudaGetDeviceProperties(&deviceProp, i);

        printf("\nDevice %d:\n", i);
        printf("  Device name: %s\n", deviceProp.name);
        printf("  Compute capability: %d.%d\n", deviceProp.major, deviceProp.minor);
        printf("  Total global memory: %.2f GB\n", static_cast<float>(deviceProp.totalGlobalMem
) / (1024 * 1024 * 1024));
        printf("  Number of multiprocessors: %d\n", deviceProp.multiProcessorCount);
        printf("  Maximum threads per block: %d\n", deviceProp.maxThreadsPerBlock);
        printf("  Maximum threads per multiprocessor: %d\n", deviceProp.maxThreadsPerMultiProce
ssor);
    }

    return 0;
}
```

2. Compile the code using the NVIDIA compiler as follows:
   `$nvcc GPU_info.cu -o info-GPU`
3. Run the output file and try to explain the output.

Sequential Code for Array Addition with Array Initialization Function

1. Write a C code that implements a function for a sequential array addition and another separate function that initializes the input arrays with random integers. The input array size is given as a parameter to the program during its execution (i.e. `argc` and `argv` parameters in the `main` function).
2. Add functions to measure the execution time of each function (i.e. using clock() or gettimeofday() of time.h)
3. Compile and run your program by giving different array sizes following a logarithmic scale of base of 2 in the range [16, 2024].
   NOTE: for every input size, you must run the program multiple times (at least 5 times). You can apply some statistics on the obtained results (i.e. mean, standard deviation, etc).
4. Present the obtained performance results on a bar chart where each bar showcases the execution time for a given input size (total = initialization + addition + delta).
5. Try to analyze the results and to find out the bottlenecks.

Parallel Array Addition with CUDA

Starting from your previous C code, you will parallelize the addition function following these steps:

1. Allocate memory on the host for input arrays and the output array.

2. Initialize input arrays with random values using the initialization function. This function should be run by the host.
3. Allocate memory on the device (GPU) for the input and output arrays.
4. Copy input arrays from host to device.
5. Modify the addition function to make it a CUDA kernel function.
6. Launch the CUDA kernel with appropriate grid and block dimensions. Like input array size, grid dimension and block dimension must be given as input parameter to your program.
7. Copy the result array C from device to host.
8. Free memory on both host and device.
9. Compile your CUDA code and run it with a fixed input size using 1 thread, 1 block, all the blocks. Instead of using C function to measure execution time, use the nvidia profiler `nvprof` as follows:

```
$/usr/local/cuda/nvprof ./your-cuda-program <paramters>
```

10. Calculate the speed-ups.
11. Re-run the program with varying the input size, the grid and block dimension. Profile the performances using nvprof using –log-file and –csv options. Automate the execution using a shell or a python script.

```
For expNum in [1 2 3 4 5]
      For inputSize in [16 32 64 128 256 512 1024 2048]
            For gridSize in [1… MaxNumOfBlocksPerGrid]
                  For blkSize in [1… MaxNumOfThreadPerBlock]
                  $/usr/local/cuda/nvprof --log-file arraay-
add_inputSize_gridSize_blkSize_expNum.csv --csv ./your-cuda-program
inputSize gridSize blkSize
```

12. Analyze the outputs using bar charts.

Sequential vs. Parallel Matrix Addition

1. Guess what? You will be redoing the same thing for matrix addition! That's it.

GPU_info

```
#include <stdio.h>
#include <cuda_runtime.h>

int main() {
   int deviceCount;
   cudaGetDeviceCount(&deviceCount);

   if (deviceCount == 0) {
      printf("No CUDA-capable devices detected\n");
      return 0;
   }

   printf("Number of CUDA devices: %d\n", deviceCount);

   for (int i = 0; i < deviceCount; ++i) {
      cudaDeviceProp deviceProp;
      cudaGetDeviceProperties(&deviceProp, i);
      printf("\nDevice %d:\n", i);
      printf("  Device name: %s\n", deviceProp.name);
      printf("  Compute capability: %d.%d\n", deviceProp.major, deviceProp.minor);
```

```
        printf("  Total global memory: %.2f GB\n", static_cast<float>(deviceProp.totalGlobalMem) / (1024 * 1024 * 1024));
        printf("  Number of multiprocessors: %d\n", deviceProp.multiProcessorCount);
        printf("  Maximum threads per block: %d\n", deviceProp.maxThreadsPerBlock);
        printf("  Maximum threads per multiprocessor: %d\n", deviceProp.maxThreadsPerMultiProcessor);
    }

    return 0;
}
```