Bhuvi Ghosh
60009210191

HPC Experiment-7

Aim: Accelerating a For Loop with a Mismatched Execution Configuration

Theory:

It may be the case that an execution configuration cannot be expressed that will create the exact number of threads needed for parallelizing a loop.

A common example has to do with the desire to choose optimal block sizes. For example, due to GPU hardware traits, blocks that contain a number of threads that are a multiple of 32 are often desirable for performance benefits. Assuming that we wanted to launch blocks each containing 256 threads (a multiple of 32), and needed to run 1000 parallel tasks (a trivially small number for ease of explanation), then there is no number of blocks that would produce an exact total of 1000 threads in the grid, since there is no integer value 32 can be multiplied by to equal exactly 1000.

This scenario can be easily addressed in the following way:

Write an execution configuration that creates more threads than necessary to perform the allotted work.

Pass a value as an argument into the kernel (N) that represents to the total size of the data set to be processed, or the total threads that are needed to complete the work.

After calculating the thread's index within the grid (using tid+bid*bdim), check that this index does not exceed N, and only perform the pertinent work of the kernel if it does not.

Here is an example of an idiomatic way to write an execution configuration when both N and the number of threads in a block are known, and an exact match between the number of threads in the grid and N cannot be guaranteed. It ensures that there are always at least as many threads as needed for N, and only 1 additional block's worth of threads extra, at most:

```
// Assume `N` is known

int N = 100000;

// Assume we have a desire to set `threads_per_block` exactly to `256`

size_t threads_per_block = 256;

// Ensure there are at least `N` threads in the grid, but only 1 block's worth extra

size_t number_of_blocks = (N + threads_per_block - 1) / threads_per_block;

some_kernel<<<number_of_blocks, threads_per_block>>>(N);
```

Because the execution configuration above results in more threads in the grid than N, care will need to be taken inside of the some_kernel definition so that some_kernel does not attempt to access out of range data elements, when being executed by one of the "extra" threads:

```
__global__ some_kernel(int N)
{
  int idx = threadIdx.x + blockIdx.x * blockDim.x;

  if (idx < N) // Check to make sure `idx` maps to some value within `N`
  {
    // Only do work if it does
  }
}
```

Lab Experiment to be performed:

The program in 02-mismatched-config-loop.cu allocates memory, using cudaMallocManaged for a 1000 element array of integers, and then seeks to initialize all the values of the array in parallel using a CUDA kernel. This program assumes that both N and the number of threads_per_block are known. Your task is to complete the following two objectives, refer to the solution if you get stuck:

1. Assign a value to number_of_blocks that will make sure there are at least as many threads as there are elements in a to work on.
2. Update the initializeElementsTo kernel to make sure that it does not attempt to work on data elements that are out of range.

```
[ ] !nvcc --version
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
```

```
pip install nvcc4jupyter
```

```
Collecting nvcc4jupyter
  Downloading nvcc4jupyter-1.2.1-py3-none-any.whl.metadata (5.1 kB)
Downloading nvcc4jupyter-1.2.1-py3-none-any.whl (10 kB)
Installing collected packages: nvcc4jupyter
Successfully installed nvcc4jupyter-1.2.1
Note: you may need to restart the kernel to use updated packages.
```

```
[ ] !python --version
    !nvcc --version
    !pip install nvcc4jupyter
    %load_ext nvcc4jupyter
```

```
Python 3.10.12
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Tue_Aug_15_22:02:13_PDT_2023
Cuda compilation tools, release 12.2, V12.2.140
Build cuda_12.2.r12.2/compiler.33191640_0
Requirement already satisfied: nvcc4jupyter in /usr/local/lib/python3.10/dist-packages (1.2.1)
Detected platform "Kaggle". Running its setup...
Updating the package lists...
Installing nvidia-cuda-toolkit, this may take a few minutes...
Source files will be saved in "/tmp/tmpjhwkkwbx".
```

```
%%cuda
#include <iostream>
#include <iostream>

__global__ void initializeElementsTo(int *a, int val, int N) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        a[idx] = val;
    }
}

int main() {
    int N = 1000;
    int *a;
    cudaMallocManaged(&a, N * sizeof(int));

    int threads_per_block = 256;
    int number_of_blocks = (N + threads_per_block - 1) / threads_per_block;

    initializeElementsTo<<<number_of_blocks, threads_per_block>>>(a, 42, N);
    cudaDeviceSynchronize();

    // Print first 10 values to check
    for (int i = 0; i < 10; ++i) {
        std::cout << a[i] << " ";
    }
    std::cout << std::endl;

    cudaFree(a);
    return 0;
}
```

```
42 42 42 42 42 42 42 42 42 42
```

Conclusion:

In this experiment, we handled a mismatched execution configuration by calculating the number of blocks using (N + threads_per_block - 1) / threads_per_block to ensure at least N threads are launched. Since this may create extra threads, we added a boundary check (if (idx < N)) inside the kernel to avoid out-of-bounds access. This method ensures correctness while leveraging optimal thread block sizes for better GPU performance.