Assignment 4 : Image Filters using CUDA - 2

CSCE-435 Fall 2025
Texas A&M University

November 25, 2025

1 Compile and execute project

- Upload the starter code to your scratch directory after logging into the grace portal.
- Navigate to the directory that the files were uploaded to after logging into the grace portal.
- Intialize the CMake build using the command:

```
$ . build.sh
```

- **Whenever** a change is made to the code, run make to re-build:

```
$ make
```

- Run the batch file by running the following command:

```
$ sbatch filter.grace_job
```

- Once the job is completed, you will be able to see the output file named according to the job id.

2 Assignment

You will be implementing the following methods for image filtering

- **Shared** memory to read the image and **Constant** memory to read the filter.
- **Shared** memory to read the image and **Constant** memory to read the filter. In addition, also use vectorized data types to allow each thread to read and process **4 elements** instead of one as discussed in class.

Implementation

Implement the following functions present in the file `filter.cu`:

- `filter_shared()`: Kernel that filters the input image with the image read from **shared** memory and the filter read from **constant** memory.
- `filter_shared_vectorized()` : Kernel that filters the input image with the image read from **shared** memory and the filter read from **constant** memory. Uses vector data types to load and process multiple elements per thread.

In addition, you will also need to fill in additional sections(such as initializing the image, allocating host memory, etc.) marked with a **TODO:** comment in `filter.cu`.

HINT: Use the `uchar4` data type to read the data from global memory.

HINT: There is a CPU version of the filter already implemented in the starter code that can be used to check the output from the kernel.

IMPORTANT: The number of threads per block(`THREADS`) and the number of blocks(`BLOCKS`) needs to be decided carefully to optimally distribute work among the threads. Consider the domain of the input(2D grid of numbers) when deciding on the dimension and number of threads per block and the dimension and the number of blocks. Calculation for the `BLOCKS` will typically depend upon the size of the image.

CUDA Timers

Use CUDA timers to mark the following regions

- `cudaMemCpy` function for transferring data from host to device **and** device to host.
- The two kernels `filter_global` and `filter_constant`

Cuda timers can be implemented using CUDA Events API. Some useful elements are

- `cudaEvent_t`
- `cudaEventElapsedTime()`
- `cudaEventRecord()`
- `cudaEventCreate()`
- `cudaEventSynchronize()`

Caliper

Use **Caliper** to calculate the `cudaMemCpy` times and the runtime of the **two** kernels `filter_global()` and `filter_constant()`.

You can simply mark regions with caliper similar to previous labs. Make sure the mark the regions with cuda timers first.

NOTE: Caliper intercepts CUDA events to actually measure times so you will still need CUDA events in your code.

Effective Bandwidth

Calculate the effective bandwidth of the two kernels(in GB/s). The following link might be helpful Performance Metrics.

Analysis

Analyze and plot the following

- **Image Size v/s Kernel Time**: Vary the image size as **1024 x 1024, 4096 x 4096, 8192 x 8192**. Plot the **two** kernels on the **same** plot.
- **Image Size v/s Effective Bandwidth**: Vary the image size as **1024 x 1024, 4096 x 4096, 8192 x 8192**. Plot the **two** kernels on the **same** plot.

NOTE: The filter size you are implementing is **3 x 3** and **5 x 5**.

NOTE: Since these filters are going to be computed very fast, there can be outliers in your observed timings. So make sure to run your code multiple times and pick the minimum time you observe over multiple runs.

Observations

Write you observations on the variance of the time and effective bandwidth as the image size, kernel filter size changes. Make sure to explain the observed trends.

Also, answer the following questions based on the lecture on 3D stencils for a 27-point stencil(see fig.1). Assume that the output tile dimension is `out_tile_dim`.

1. Calculate the ratio of arithmetic operations per byte read from global memory for a shared memory implementation of the 3D stencil in terms of `out_tile_dim`. What variables should be maximized/minimized in order to get a higher ratio of arithmetic operations per byte read?
2. Calculate the ratio of arithmetic operations per byte read from global memory for a shared memory implementation of the 3D stencil that also uses **thread coarsening** in terms of `out_tile_dim`. Why is this better/worse than the version without thread coarsening?

Grading

Exercise	Points
Correctly implemented `fliter_shared()`	30
Correctly implemented `filter_shared_vectorized()`	30
Graphs	10
Observations	30

3 Submission

Submit a `.zip` file on Canvas containing

- A pdf with the graphs and your observations.
- The file `filter.cu` with your code changes.
- A `.zip` file containing your `.cali` files.

(a) 7-point (b) 27-point

Figure 1: Stencil types.