Lab 5 - Introduction to OpenCL programming

Diego Martínez Baselga

735969@unizar.es

David Morilla Cabello

822899@unizar.es

Universidad de Zaragoza

Report for the course:

Programming and Architecture of Computing Systems

Grupo Arquitectura de Computadores

29th December 2020

1 Introduction

In this laboratory session, some fundamentals of the OpenCL API are presented. They are used in order to setup the environment, enable communication between host and device, simple kernel programming, and synchronization basics. In addition, some basic kernels are implemented. The design, implementation and analysis of the kernels are discussed in the report.

The kernels implemented are the three proposed: a kernel to flip an image, to rotate it and to obtain the histogram of each of its channels. Moreover, four different versions of the histogram kernel are implemented and explained in the report. Finally, some metrics are obtained for the fourth versions of the histogram kernel, comparing and analyzing them. This reports explains the methodology followed and discusses the obtained results.

The C++ library CImg is used in order to load the images and plot the results, as it is suggested in the assignment.

The system used to do the project is a system of the lab 102 of the EINA. It is a Linux system with a CentOS distribution with 4 cores of 3.2 GHz Intel Core i5-4570 CPU processors. The compiler used was g++ (GCC) 9.2.0.

2 Methodology

Six different kernels are implemented, which are the following:

- Image flip
- Image rotation
- Basic image histogram
- Image histogram optimizing memory access
- Image histogram with input data partition
- Image histogram with output data partition

The first three kernels are implemented looking only for correctness of their deign and result. In the other three kernels, some modifications and improvements have been made. Performance metrics are obtained for the image histogram kernels to compare and analyze them. The work [1] has been used as inspiration for the last two kernels.

Some OpenCL preferences of the system are used in the implementation of the kernels. The preferred vector sizes of all of the data types is 1 and the preferred workgroup size are multiples of 128. These preferences are used in the implementation.

2.1 Image flip

In this kernel, an image is transformed by flipping it over its vertical axis. This is done in-place, having that the same source buffer for the former image should hold the transformed image after the kernel execution.

The input buffer written in the kernel is the same input vector that stores the image in the CImg library. It stores all the pixel values of the image in a vector of unsigned chars whose direction may be accessed by using `img.data()`. The way how it stores the pixels in memory is all the red components of the pixels first, then the green and finally the blue, as it is shown in

R00	R01	R02	...	R10	R11	...	RNN	G00	...	GNN	B00	...	BNN

Table 1: Representation of how images are stored in memory by CImg library. The first component is the channel (RGB), the second is the row number and the third is the column number.

Table 1. Thus, the components of the same pixel are not contiguous in memory and they must be accessed separately.

As the whole pixel may not be accessed at the same time with this approach, the kernel used is a three dimensional kernel, iterating in the image rows, columns and channels. The approach taken to do the flip is swapping each element of the first half of the image with the corresponding one of the second half. If the number of columns is not even, the central pixel does not need to be modified. Therefore, the global work size of the kernel is: $\{number\_of\_columns, \frac{number\_of\_rows}{2}, number\_of\_channels\}$. The local work size (work-group size) is set to {16, 8, 1}, which is multiple of 128. Each work item gets its corresponding row, column and channel and performs the following swap:

$$img[channel, row, column] <-> img[channel, row, number\_of\_columns - 1 - column]$$

In this kernel only the image buffer is needed. It is a read and write kernel, as the input and the output is stored in the same one. The transformed image may be displayed directly from the buffer.

2.2 Image rotation

The kernel transforms an image by rotating it over the center of the image. The rotation is given to the program as an argument, and it passes it to the kernel.

The first important thing that needs to be noticed is that the origin in CImg library is not in the bottom-left of the image. Therefore, the angle that is used is the negative of the angle received, in order to have the rotation of the image in the angle expected.

As in the previous case, a three dimensional kernel is needed, as the pixels components may not be accessed together. In this case we need to access all the pixels, so the global work size of the kernel is: $\{number\_of\_columns, number\_of\_rows, number\_of\_channels\}$. The local work size will be a multiple of 128 again.

The rotation is achieved by performing the following operation, having that x_2 and y_2 are the resulting row and column, x_1 and y_1 are the input row and columns, and x_0 and y_0 are the row and column of the central pixel:

$$x_2 = \cos(angle) * (x_1 - x_0) - \sin(angle) * (y_1 - y_0) + x_0$$

$$y_2 = \sin(angle) * (x_1 - x_0) + \cos(angle) * (y_1 - y_0) + y_0$$

The resulting coordinates of the pixel may not be integer. The approach followed to deal with this problem is rounding them by using the OpenCL function `convert_int_rte()`.

Two buffers are needed, one read only buffer that stores the input image and one write only buffer that stores the result. The resulting image may be displayed directly from the output buffer.

2.3 Basic image histogram

This kernel processes a given image and calculate the histogram for each color channel. Each histogram has a fixed number of bins, which is 256.

A two dimensional kernel is needed. One dimension is needed for the kernel, as in the previous cases. However, in this one there is no need to know what are the coordinates of the pixel accessed. Thus, the global work size of the kernel is: $\{image\_size, number\_of\_channels\}$. The local work size will be {128, 1}.

Each of the work items will read each corresponding component of the pixel value and will increment in one the bin of that value. The problem is that all of them write in the same output buffer and some of them will need to increment the same bins. This must be done in mutual exclusion to avoid inconsistencies. In order to do it, atomic operations are used. Particularly, the operation `atomic_add` is used, which atomically loads a variable value from memory, adds it to a constant value and stores it. This may serialize a lot the program, but provides a correct result.

Two memory buffers are used. The first is a read only buffer that stores the input image as an unsigned char array. The second one is an unsigned int array that stores the resulting histograms. Its size is 256*3, meaning that it stores the histogram of each of the channels in the same vector. It is a read and write histogram. It is initialized to 0 in the host first, written in the device memory, read and written by the kernel and, after executing the kernel, read again by the host.

The resulting histograms are plotted with a function implemented using the CImg library. The function `draw_line()` is used to write each bar corresponding to each histogram bin.

2.4 Image histogram optimizing memory access

This kernel offers a small modification of the previous kernel. The CImg library stores the kernels in a way that all of the components of a pixel may not be accessed at the same time, as they are not contiguous in memory. The modification implemented aims to solve it.

The experiments that are shown in the section 3 focus in comparing kernel performance. Thus, adding some overhead in the kernel preparation is not a big issue. The image is processed and stored in an unsigned char array, whose size is $rows * columns * (4)$ with respect to the original image. The array will be used by the kernel as an unsigned integer array. Each integer is composed by 4 bytes: 3 bytes for the channels of the pixel and another extra byte (this byte is included to prevent the program from having alignment problems). Each work item accesses all the pixel by accessing the integer, gets the value of the components of the pixel and increments the bin of each color channel.

The kernel has only one dimension, which is the number of pixels in the image. The work group size is set to 128.

2.5 Image histogram with input data partition

In this version of the kernel, the input data is splitted into chunks. One of the chunks is assigned to to each of the work groups, which computes a local histogram. The histograms are finally merged.

The definition of the global work size does not depend on the image size. They are set manually to some values that work good. The work group size is set to $4 * 128$ and the number of work groups is set to 16. The global work size is obtained by multiplying these two values.

In the kernel, the work item computes the corresponding range of pixels of its work group and the corresponding range of pixels of itself. The image is divided evenly in the work groups, which divide its chunk among its work items too. The divisions are done using a function that splits evenly, assigning the same amount of elements to each of the compute units, with some of them having only one more than the others. Therefore, each work item will obtain the histogram of almost the same amount of pixels.

Each work groups have a local histogram. It is initialized to 0 first by all its work items concurrently (each of them initialize the same number of bins). A local barrier is used to synchronize all of them before starting to compute the histogram to ensure that it is initialized. Then, each work item increments the histogram bins of the pixels that have been assigned to it, using the same approach as in Section 2.4, but in their local work group histograms. A local barrier is used again to ensure that all of them have finished before the next step. Finally, the histograms are merged in the output global buffer. They are merged having that all items merge the same number of bins, in the same way as the local histograms are initialized.

Atomic operations are used to update the local histograms and to obtain the global one from the local. Nevertheless, all the work groups compute the local histograms together, and they only have to deal with mutual exclusion inside of them, achieving more concurrency. The atomic operations of the final merge affect all the work groups. However, if the number of work groups is not too high, it should not have a big effect on the performance.

2.6 Image histogram with output data partition

The kernel explained in Section 2.5 is modified to allow output data partition among the work groups instead of input data partition. Each work group computes the histogram of all the image, but only for some specific bins. Then, they are merged to the global buffer.

The main difference in this kernel is the divisions computed at the beginning. The function that splits evenly is used to divide the number of bins of the histograms among the number of work groups. Then, the pixels of the image are divided among the work items of the work group. Therefore, each work item has a range of pixels to compute the histogram, but only if they have a value that is in a specific range.

The aim of this version is to limit the parts that must be accessed in mutual exclusion. Atomic operations are not longer needed to merge the result in the global histogram, and they are only used to increment the bins of the local ones.

As in the previous case, barriers have been used to enable synchronization among work items of the same work group. The number of work groups has been set to 8, and the number of work items per work group to 512.

2.7 Performance metrics

Some metrics have been computed to compare and analyze the different versions of the kernel histogram. The approaches taken are the following:

- **Execution time of the overall program**: The class `std::chrono::steady_clock` is used to measure times. The value of the clock is taken at the beginning and at the end of the program. Their difference is the execution time of the overall program.

- **Execution time of the kernel**: In this case, the profiling methods offered by OpenCL are used. With the function `clGetEventProfilingInfo()`, the value of the times when the kernel starts and ended are obtained. Their difference is the execution time of the kernel.

- **Bandwidth**: The maximum bandwidth may not be obtained, as memory hardware specifications are not known. They may not be accessed using the terminal either, as to use the tool `dmidecode`, root permissions are needed. It could be computed by multiplying the clock rate of the memory and the width of the memory interface. To measure times ensuring that the memory transference has finished, `clFinish()` has been used. Three different bandwidths are calculated:
  - Write bandwidth: Bandwidth of writing data from host to device memory. It is computed with the data transferred by `clWrite` and the time that it takes, measured with the class `std::chrono::steady_clock`.
  - Read bandwidth: Bandwidth of moving data from device to host memory. It is computed with the data transferred by `clRead` and the time that it takes, measured with the class `std::chrono::steady_clock`.
  - Kernel bandwidth: Bandwidth of accessing data by the kernel work items and the device memory. It is computed with the data read and written in device buffers and the time that it takes, measured with the class `std::chrono::steady_clock`. The local histograms (local memory) are not taken into account.
- **Throughput of the kernel**: It is the amount of work per unit of time. It is computed with the size of the image and the execution time of the kernel.
- **Memory footprint**: The memory used by the process is computed with the following command: `/bin/ps -p <pid> -o size | /bin/tail -1 | /bin/tr '\n' ' '`, having that <pid> is the process of the actual process. It returns the memory used by the program itself. It is computed two times: When every OpenCL element has been released at the end of the program (memory footprint of the whole program) and when the kernel is executed. The difference between them is the memory footprint of the kernel itself.

3 Results

In this section the results are computed and discussed.

3.1 Correctness of the kernels

The correctness of the execution of the kernels is proved by executing them with the example image provided. The result of the flip kernel and the rotate kernel with 1 radian is shown in Figure 1. The result of the histogram kernel (all of them produce the same result) is shown in figure 2.

3.2 Performance metrics and analysis

To get the metrics of the histogram kernels, the following approach was taken. 5 images were generated of sizes 640x640, 1280x1280, 1920x1920, 2560x2560 and 3200x3200 pixels. The values of the pixels were uniformly random, generated with values from 0 to 255 on the three channels.

For the execution of each kernel and each image, the execution mean time of 5 different executions is computed. In addition, before storing the results, it is executed 2 times as a warm up.

The program and the kernel times are shown in Figure 3 and Figure 4. From the total program times, it may be seen that the overhead produced by the array creation makes the second and

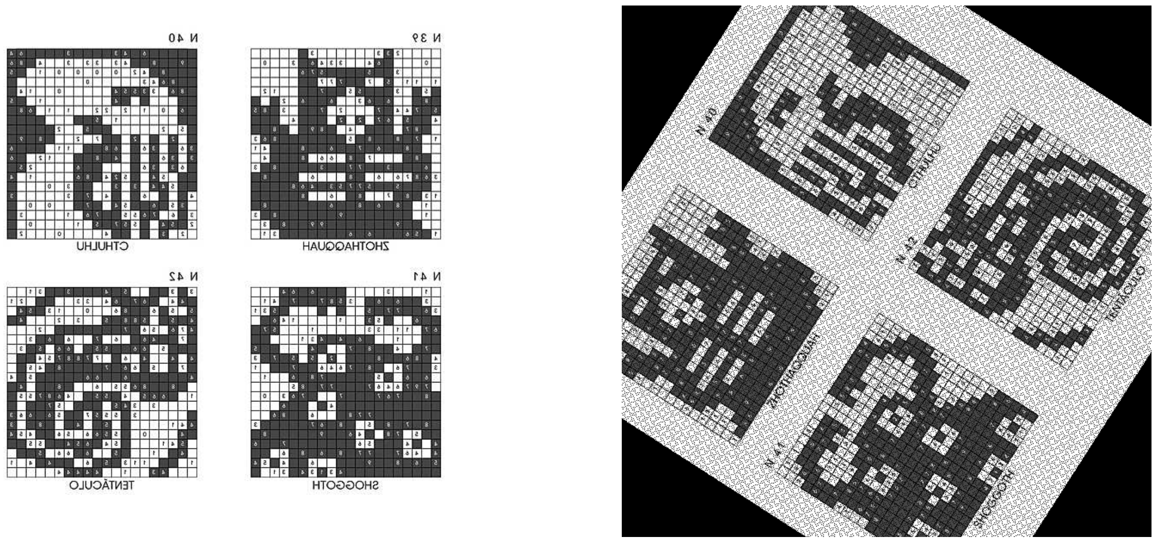

Figure 1: Results of the flip and rotate kernels.

the third kernel take longer times than the first one. In the kernel times, several conclusions may be extracted. The input data partition provides much better results than the other three kernels. In addition, doing output data partition improves slightly performance over the basic version with the same memory access (the second kernel). Furthermore, the memory access modification implemented in the second kernel does not improve its performance. This may be due to the fact that the vectorization instructions that the compiler uses provide an efficient access in the basic kernel, and the computation performed by the two dimensional kernel is more efficient than the one dimensional one, as it is more concurrent.

In Figure 5 the throughput of the kernels may be seen. The information of this plot is similar to the ones before. The write and read bandwidth are some features relative to the system, so their values are similar among the kernels, as it may be seen in Figures 7 and 6. However, the kernel bandwidth is affected by the throughput of the kernel, as it may be seen in Figure 8.

In Figures 10 and 9 it may be seen that the footprint of the programs that have the same memory access is similar, whereas they differ from the other one. This is due to the fact that those programs use another array, which needs more memory.

References

[1] Juan Gómez-Luna et al. 'Chai: Collaborative heterogeneous applications for integrated-architectures'. In: *2017 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE. 2017, pp. 43–54.

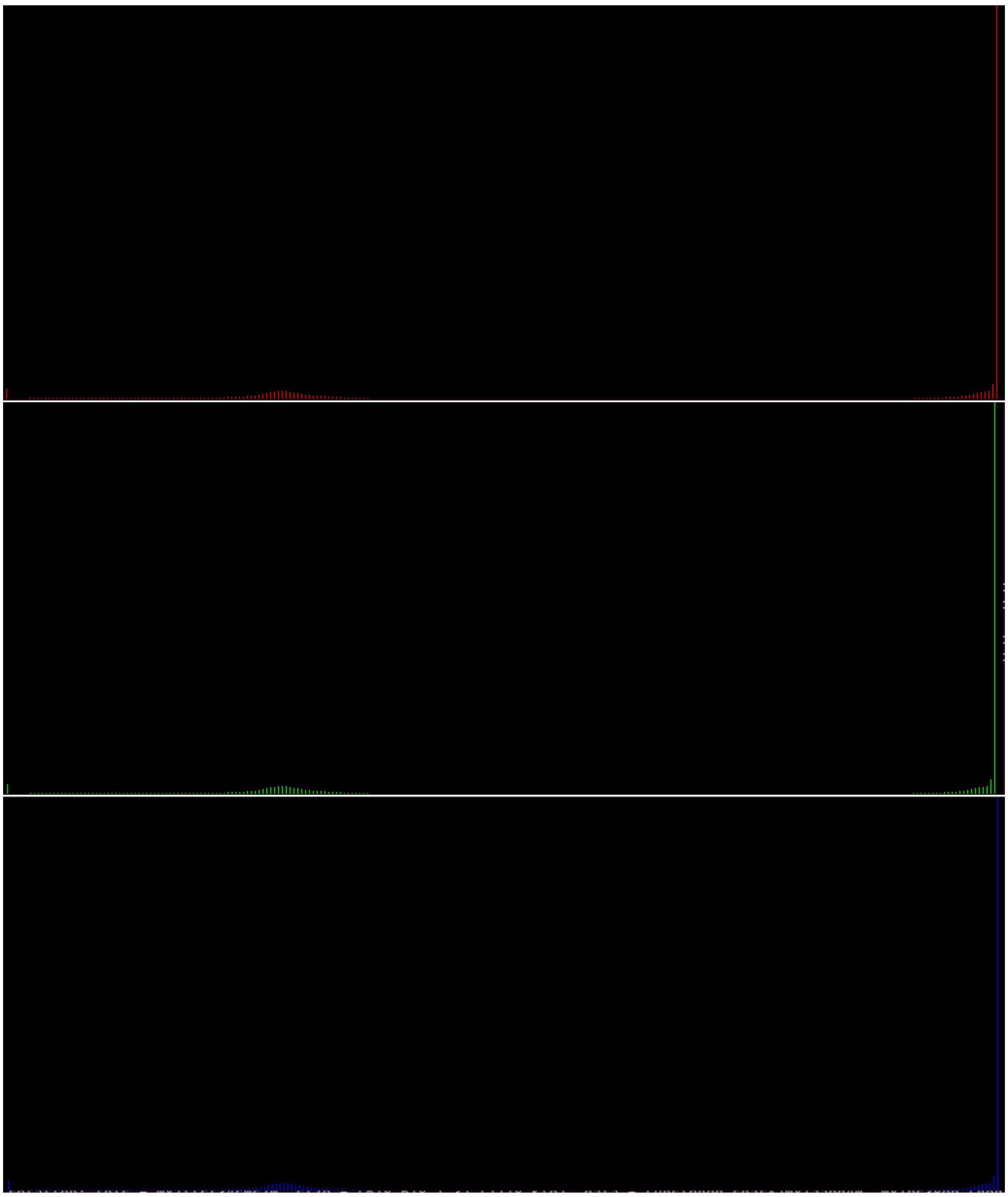
Figure 2: Results of the flip and rotate kernels. It is a black and white image, so the histograms of the channels are the same

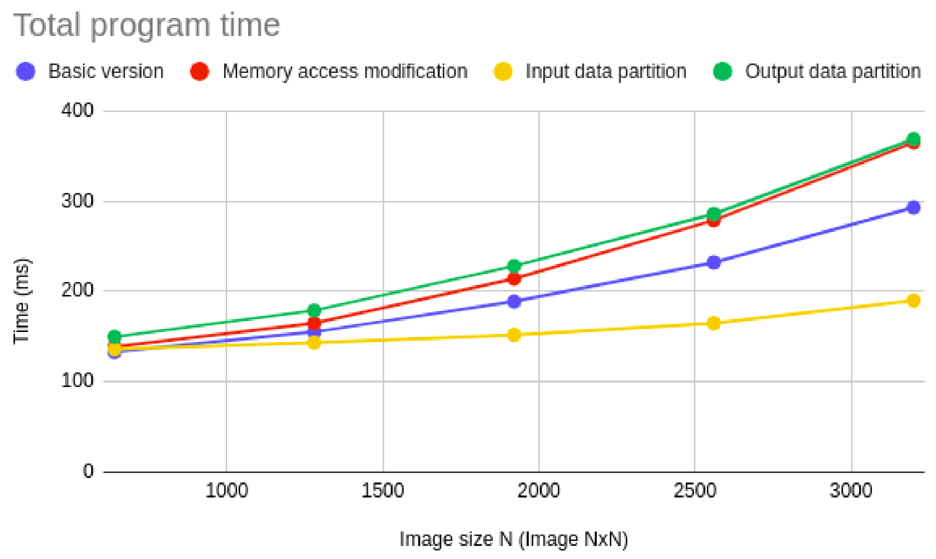

Figure 3: Program time comparison

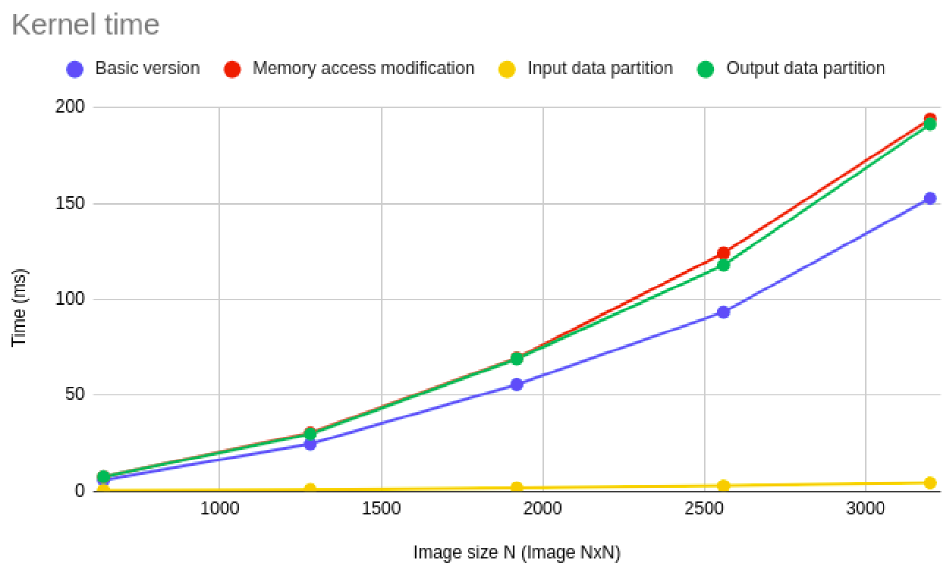

Figure 4: Kernel time comparison

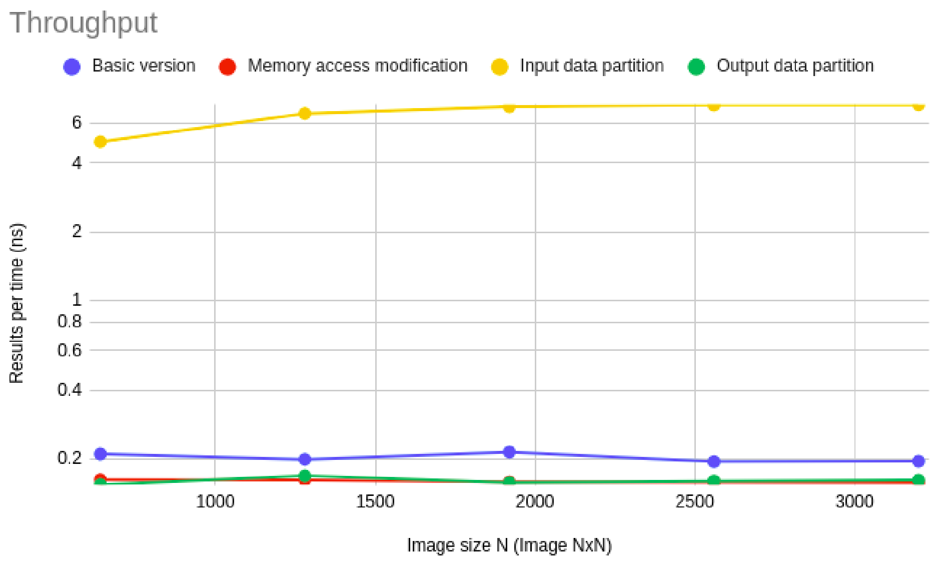

Figure 5: Throughput comparison

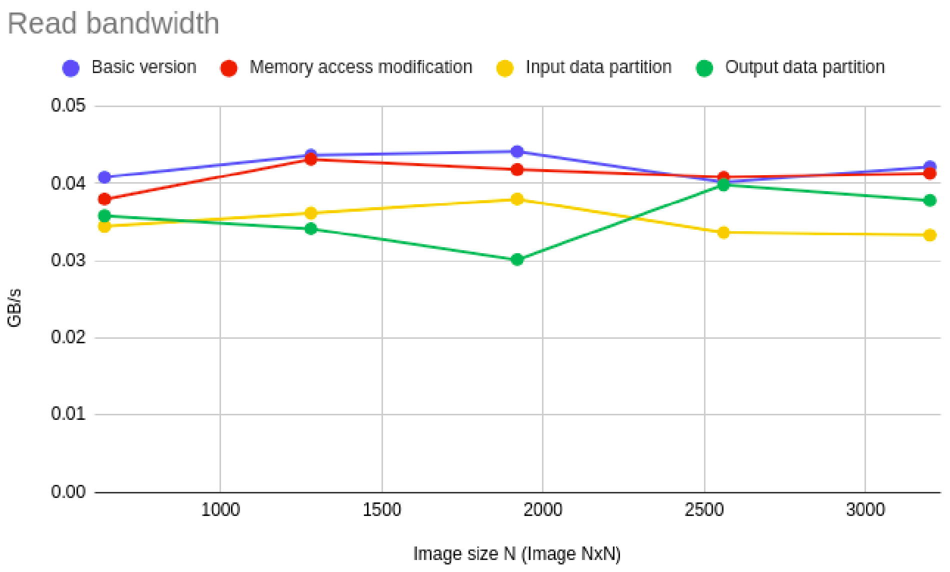

Figure 6: Read bandwidth time comparison

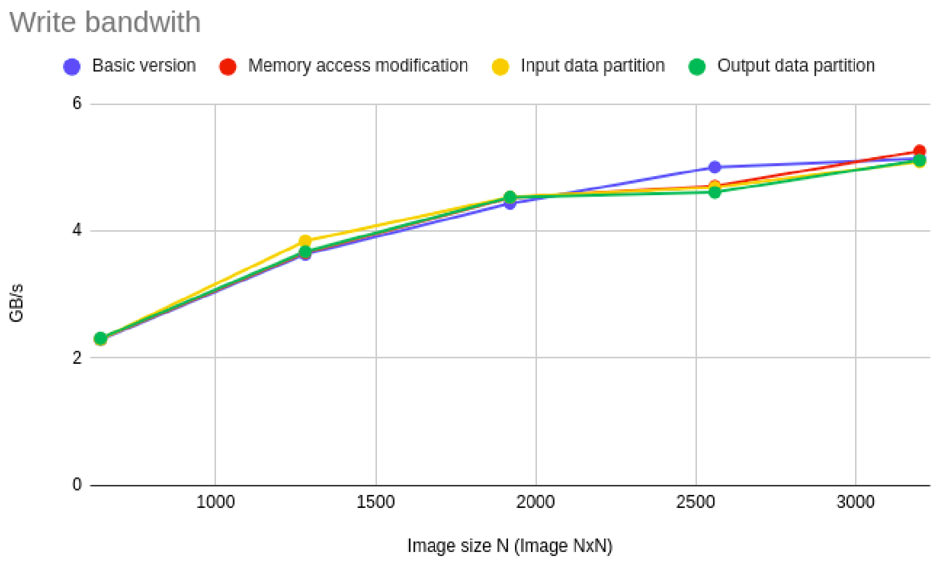

Figure 7: Write bandwidth comparison

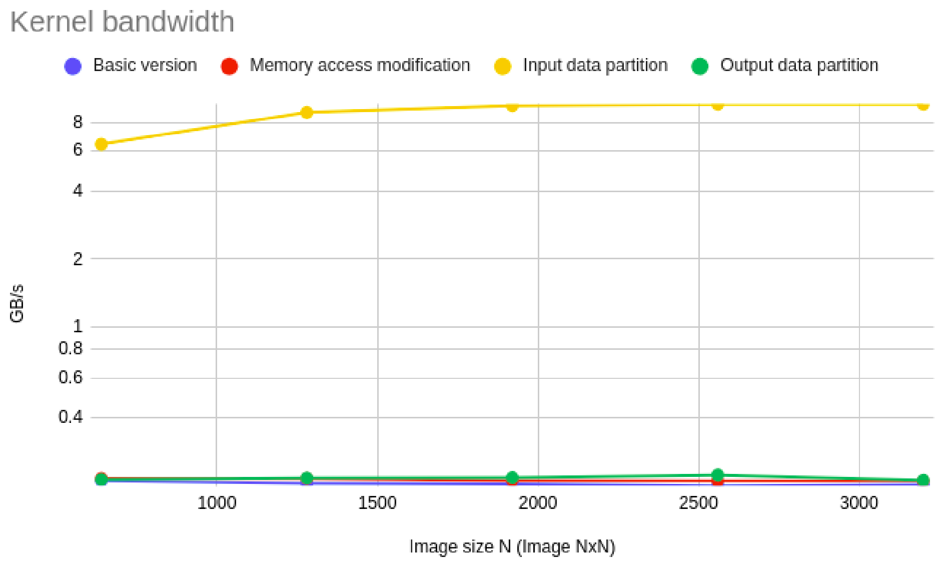

Figure 8: Kernel bandwidth comparison

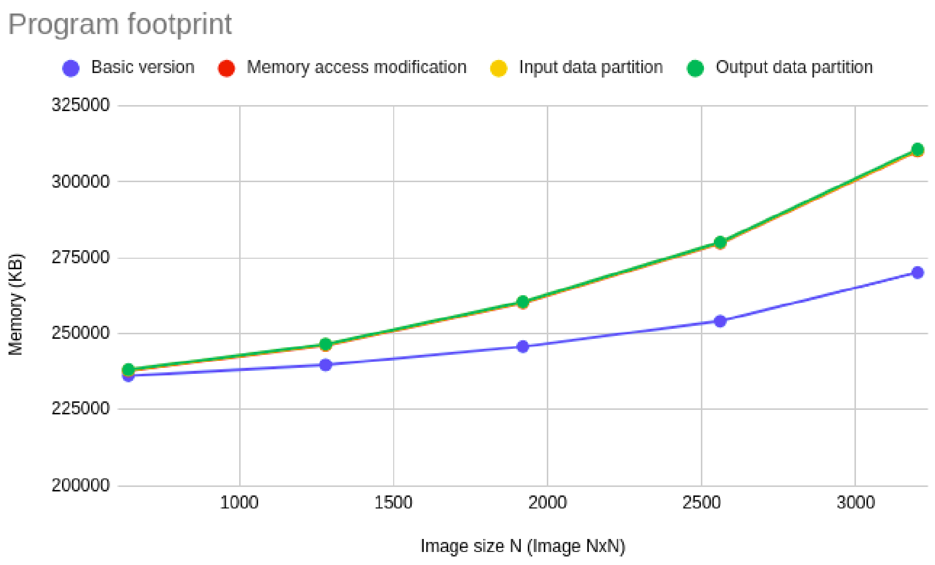

Figure 9: Program footprint comparison

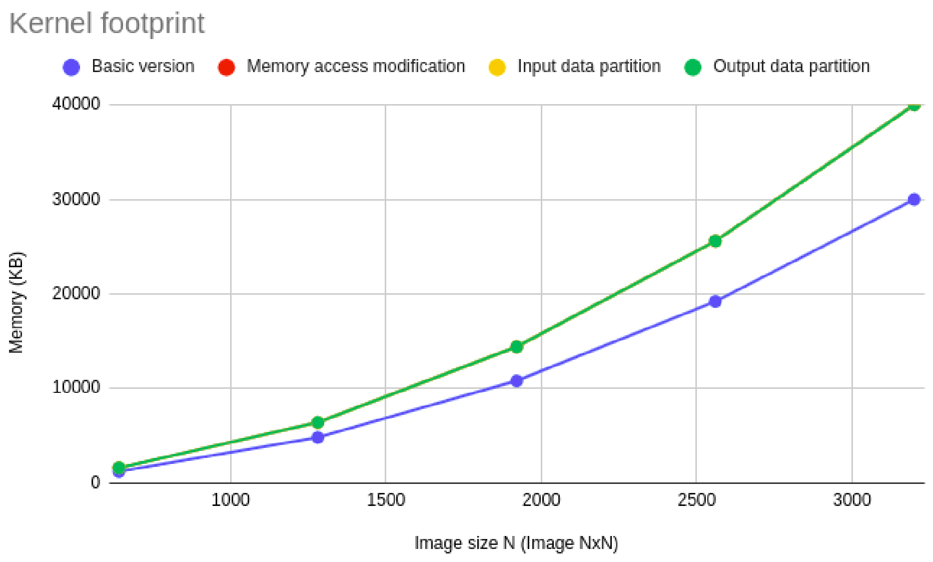

Figure 10: Kernel footprint comparison