CovisionLab Project Report

Simone Dagnino Patrick Pastorelli

January 2024

NVJPEG and NVOF analysis

Contents

1 Introduction

For this final project we collaborated with CovisionLab for analysing and benchmarking two libraries developed by Nvidia: the NVJPEG library and the Optical flow library. The former allows to compress/decompress images to/from JPEG format by using all the power of an Nvidia GPU, while the latter exploits dedicated hardware accelerators introduced in the Turing architecture of Nvidia GPUs to calculate Optical flow, that is, the relative motion of pixels between a pair of images. Our goal was to understand how the APIs are structured and what parameters they expose to the user; after that we developed two scripts which actually use the APIs to understand how to correctly them inside a script, and finally we did qualitative analysis on the quality of the obtained results and quantitative analysis on the general performance of both libraries.

2 NVJPEG

2.1 Introduction

Part of the project was exploring and benchmarking the execution times and the compression rates of the NVJPEG library [5] to see if it was possible to create a high throughput compression pipeline, using GPU acceleration, to feed images to a machine learning model faster than a CPU compression algorithm. Our tests aimed to sample the execution times to compress n images in parallel and see how these results changed based on the parameters in use, but also as n varies. Besides this we also calculated compression rates and the loss of information caused by the lossy compression using the structural similarity index measure (SSIM). With this data we could find the best trade off between execution time, compression size and loss of information for the needed use case.

2.2 Work environment setup

The tests were done using a NVIDIA Quadro RTX 6000, CUDA toolkit version 12.2 and NVIDIA driver version 535.129 on Ubuntu 20.04. The library is already inside the CUDA toolkit from version 11.0 and on, otherwise it needs to be installed using the nvjpeg installer. During testing we also managed to check the compatibility of our scripts with the latest software realeases using a NVIDIA GeForce GTX 1050 Ti, CUDA toolkit version 12.3 and NVIDIA driver version 545.92 on WSL2 Ubuntu 22.04. The library works with every GPU with computing capability of at least 3.5 and has accelerated HW decode on the following GPUs: A100, A30 and H100.

2.3 NVJPEG installation guide

To use the NVJPEG library you have only one requirement:

1. A Nvidia GPU with computing capability of at least 3.5

The library comes already installed in CUDA toolkit version 11.0 or higher so the only step is to install it (this link can be used to download the last version) and, if needed, update the Nvidia driver to make it up to date with the CUDA library; a good way to check if the right version of the driver is installed is to use the command "nvidia-smi" and read its output. If the CUDA version in use is lower than 11.0 than there is also the need install the library using the NVJPEG installer.

2.4 API interface

The test were done on different configurations and combinations of parameters. Here is a schematic of the functions needed to set them and of what they do:

1. **nvjpegEncoderParamsCreate()**: Creates the structure that holds the compression parameters.
   Signature:
   ```
   nvjpegStatus_t nvjpegEncoderParamsCreate(
       nvjpegHandle_t handle,
       nvjpegEncoderParams_t* encoder_params,
       cudaStream_t stream
   )
   ```
   - **handle** [in]: Library handle.
   - **encoder_params** [out]: Pointer to the parameters structure which contains the encoding paramters.
   - **stream** [in]: The CUDA stream used for the operation.

2. **nvjpegEncoderParamsSetEncoding()**: Sets the encoding type in the encoder parameters structure.
   Signature:
   ```
   nvjpegStatus_t nvjpegEncoderParamsSetEncoding(
       nvjpegEncoderParams_t* encoder_params,
       nvjpegJpegEncoding_t etype,
       cudaStream_t stream
   )
   ```
   - **etype** [in]: Encoding type selected, the default is baseline. The available types are baseline and progressive.

3. **nvjpegEncoderParamsSetQuality()**: Sets the parameter quality in the encoder parameters structure.
   Signature:
   ```
   nvjpegStatus_t nvjpegEncoderParamsSetQuality(
       nvjpegEncoderParams_t* encoder_params,
       const int quality,
       cudaStream_t stream
   )
   ```

- **quality** [in]: Quality of the compressed image between 1 and 100 where 100 is the highest quality. The default is 70.

4. **nvjpegEncoderParamsSetOptimizedHuffman()**: Sets wheter or not to use optimized Huffman compression.
   Signature:
   ```
   nvjpegStatus_t nvjpegEncoderParamsSetOptimizedHuffman(
       nvjpegEncoderParams_t* encoder_params,
       const int optimized,
       cudaStream_t stream
   )
   ```
   - **optimized** [in]: boolean flag, if 0 then the optimized Huffamn compression is not used else it's used. The default is 0. Using optimized Huffman can make the compression slower but produces images of smaller size with the same quality compared to it's counterpart.

5. **nvjpegEncoderParamsSetSamplingFactors()**: sets the chroma subsampling in use for JPEG compression.
   Signature:
   ```
   nvjpegStatus_t nvjpegEncoderParamsSetSamplingFactors(
       nvjpegEncoderParams_t* encoder_params,
       const nvjpegChromaSubsampling_t chroma_subsampling,
       cudaStream_t stream
   )
   ```
   - **chroma_subsampling** [in]: Chroma subsampling that will be used for JPEG compression. The default value should be 4:4:4 but in our tests it was not set so it needs to be manually set before encoding.

For other basic functions like basic usage of the library and informations about the data types see the NVJPEG doc [6]

2.5 Tests

2.5.1 Execution times

One of our tests was sampling the execution times to encode a batch of n images with different compression parameters. The goal was to find the best trade off between execution time and compression quality but also to see if there were any gains in encoding multiple images in parallel rather than one at the time. The test were done using both pinned memory and mapped memory to analyze the impact of the chosen type on the performance. The main difference between pinned memory and mapped memory is that while pinned memory needs to be copied from the CPU to the GPU, mapped memory takes advantage of the 64-bits OSs unified memory address space and reads data directly from the CPU using PCIe.
In short, the varying parameters were:

1. n, the number of images [1, 4, 8, 16]
2. the chroma subsampling [410, 411, 420, 422, 440, 444]
3. the quality of the encoding [100, 90, 80, 70]
4. the use of optimized Huffman
5. the use of pinned memory vs mapped memory

The execution time defined as the sum of the individual time of three phases:

1. the copy of the image from the CPU's RAM to the GPU's (only for pinned memory)
2. the compression of the image (from .bmp to .jpeg)
3. the copy of the compressed image from the GPU's RAM to the CPU's

The following graphs show the obtained results for respectively 1, 4, 8 and 16 images (the original images are provided separately due to their size), the results were taken using a script that we made being particularly careful to avoid possible compiler optimizations that might have tipped the scale in favor of a particular combination under the hood due to the bench-marking code:

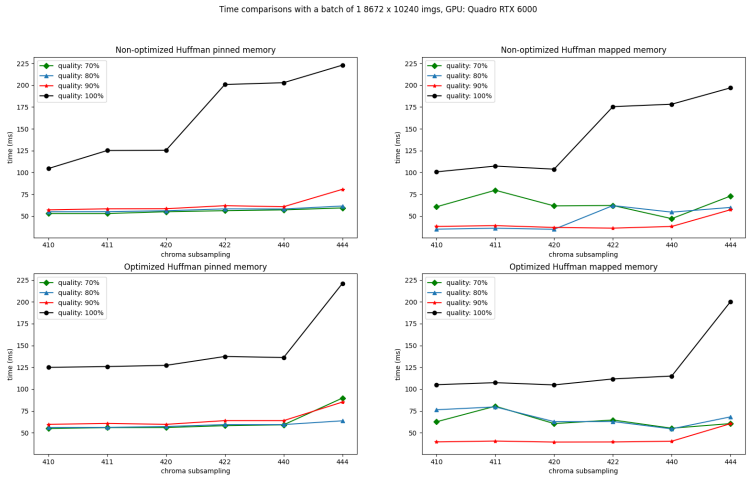


Figure 1: execution times with varying parameters on a batch 1 img/s

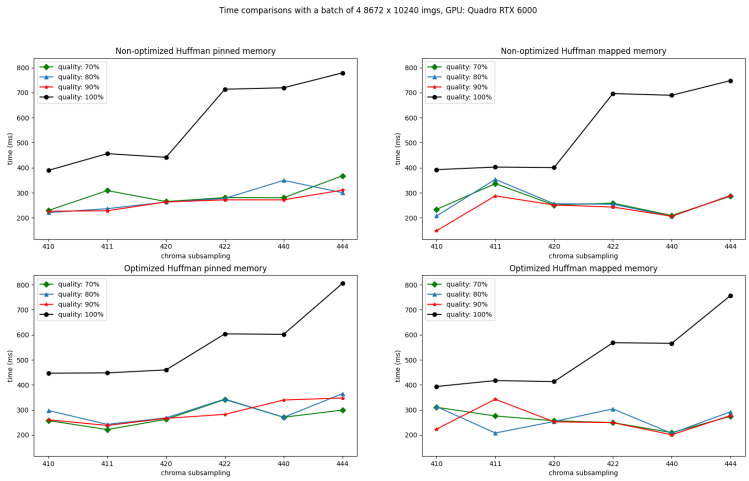


Figure 2: execution times with varying parameters on a batch 4 img/s

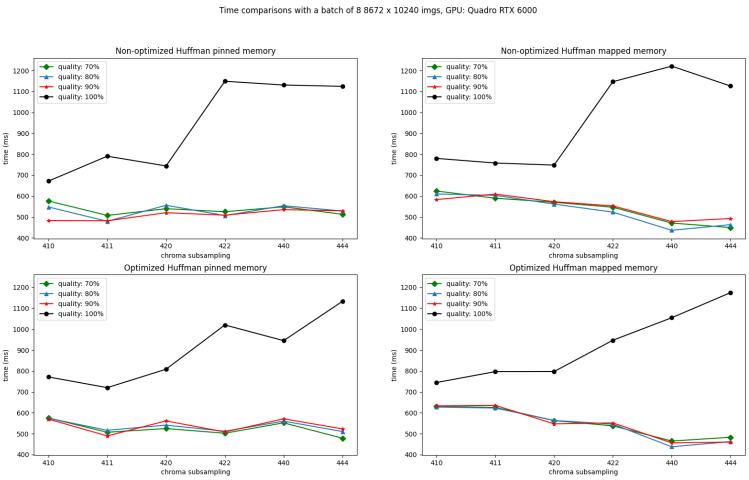


Figure 3: execution times with varying parameters on a batch 8 img/s

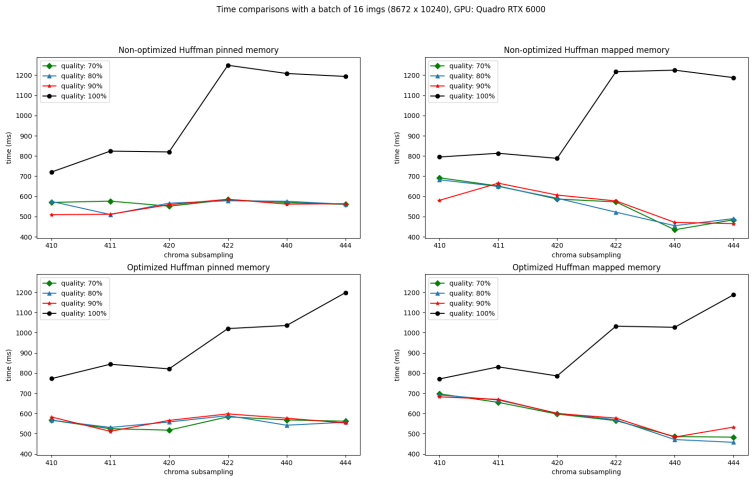


Figure 4: execution times with varying parameters on a batch 16 img/s

From the results in figs. 1 to 4 is clear that encoding 16 images is better. This is probably due to the fact that, with enough parallel threads running, the GPU can better hide memory latencies and exploit parallelism. It is also possible that increasing the number of parallel images even more would give better results, the only bottleneck of this method would be the amount of RAM available in our system. Another thing that we noticed is that quality 100 is probably never worth it due to the noticeable increase in execution time compared to 90. This is likely because the best possible quality produces compressed files that are way bigger, leading to an inevitable increase in execution time due to more computations. Notice also that quality 100 with no subsampling (so 4:4:4) does not imply lossless encoding because jpeg involves a process of quantization from floating point data to integer data leading to a loss of information even with this setup, this would be just the most accurate result obtainable from the algorithm. Plus, as we will show in the compression rates and ssim calculation the difference between quality 100 and 90 (with same subsampling type) is very small in terms of info loss so there isn't really a reason to consider 100 a viable option. In regard to the other possible values, on the other hand, we notice that the difference between the execution times is negligible making 90 a very good choice as it looks like the best trade off in execution time and information loss. It might also be important to consider that decreasing quality does not imply a gain in performance; one possible explanation is that, by reducing the quality, there are more operation to be done in order to further compress the image. Regarding chroma subsampling, if compression size is of no particular concern, 4:4:4 looks like a solid choice as it avoids subsampling entirely, offering the best resolution while also gaining some performance compared to the other setups if used with mapped memory. This is most likely because, being the simplest

operation (no subsampling at all), it works best with mapped memory which has an advantage over pinned memory when the data is read only once from RAM, used and then never accessed again. Anyway, in case of compression size being the most important metric one could easily look at the results and pick the best trade off based on the constraints of the use case. Lastly, regarding the use of optimized Huffman we see that there isn't much of a difference in terms of execution time so one could consider it as a choice in situations were compression size is more of concern.

2.5.2 Compression rates

Besides execution times one could also be interested in the compression ratio of the images so, in our test, we also calculated the compression ratio on two sample images. Here are the graphs regarding compression ratio (the original images are provided separately due to their size):

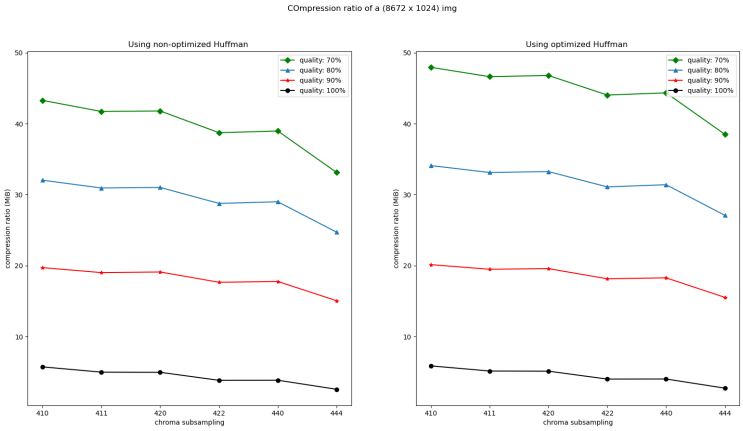


Figure 5: compression ratios of 1st image

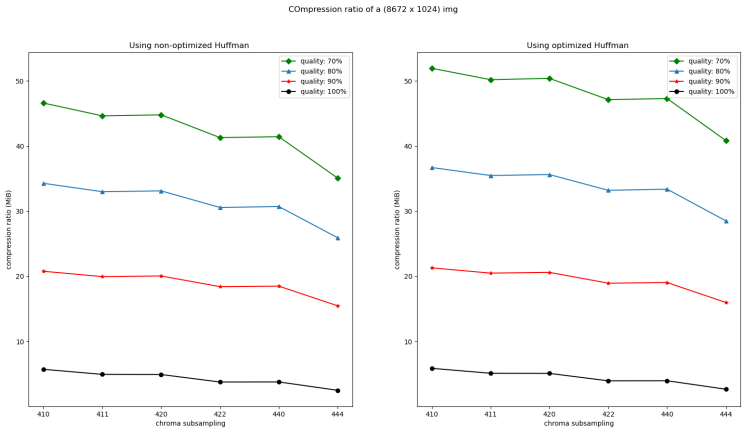


Figure 6: compression ratios of 2nd image

Where the compression ratio is defined as:

$$compression\_ratio = \frac{uncompressed\_size}{compressed\_size}$$

Another useful metrics that could be used with the provided data is the space saving metric, defined as:

$$space\_saving = 1 - \frac{compressed\_size}{uncompressed\_size}$$

This metric defines the reduction in size relative to the uncompressed size and can sometimes be more intuitive to use depending on the use case.

The results obtained in figs. 5 and 6 are in line with what anyone would expect: the ratio increases as the subsampling resolution decreases; with the quality playing a key role as well. The compression ratio seems to be going up linearly in relation to the changing of the quality. The subsampling, being almost irrelevant for quality 100 but offering a noticeable difference when the quality is set to 70, seems to have a bigger weight as quality goes down. It is also interesting to see that the use optimized Huffman encoding has an evident gain, especially as quality goes down. Judging only by compression ratio is clear that, as expected, the smallest quality is the clear winner. Obviously this might not always be the best solution overall if we have other constraints that we need to factor in such as information loss. If information loss is of particular interest, quality 90 might offer acceptable results even in this test with a ratio of about 20:1, producing images that are 20 times smaller than the original. Quality 100 still offers some space gains (with a ratio less than 5:1) so it could be a nice option if we want to be as close as possible to the original image while still saving some space and, also, assuming that it behaves better other lossless algorithm which should be the case.

2.5.3 SSIM

The last important metric that we analyzed was the SSIM, SSIM is a method for predicting the perceived quality of digital images and videos and is often more reliable than approaces that just aim to estimate the absolute error. This test was done to get an idea on the impact of the lossy compression in respect to the original image seeing how the different parameters affect the SSIM and what we can achieve with the best possible jpeg quality.

The SSIM was calculated using python's skimage library [7] and is given by the following formula:

$$SSIM(x,y) = \frac{(2\mu_x\mu_y + c_1)(2\sigma_{xy} + c_2)}{(\mu_x^2 + \mu_y^2 + c_1)(\sigma_x^2 + \sigma_y^2 + c_2)}$$

where:

1. μ_x is the pixel sample mean of x;
2. μ_y is the pixel sample mean of y;
3. σ_x^2 is the variance of x;
4. σ_y^2 is the variance of y;
5. σ_{xy} is the covariance of x and y;
6. $c_1 = (k_1 L)^2$, $c_2 = (k_2 L)^2$ are two variables to stabilize the division;
7. L is the dynamic range of the pixel-values (usually $2^{\#bits\_per\_pixel} - 1$);
8. $k_1 = .01$, $k_2 = .03$ (standard values)

The following graph shows the obtained results on one of the testing images:

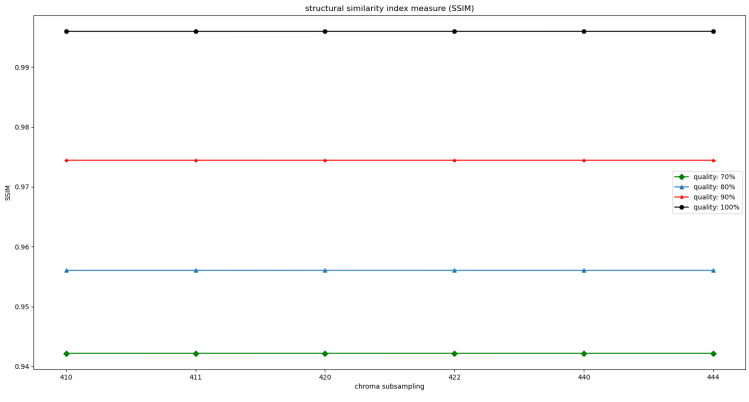


Figure 7: SSIM of original image and compressed image

We can notice that the results in fig. 7 are not really affected by the chosen subsampling and the only real factor is the chosen quality with the SSIM dropping by about 2% every time it is reduced by 10. The information loss from the best quality is negligible which is expected given that the loss of information in this specific setup is in the quantization phase only. As quality goes down, just like the compression ratio, the SSIM seems to be decreasing linearly; this is probably an effect of the approximations made by jpeg to compress the image. Having a quality of 90 still seems like a solid trade off, giving that the results are very close to the original image (SSIM of about 97%) and by factoring in the results from the other tests as well.

2.6 Conclusions

In conclusion it would certainly be possible to exploit GPUs to create a high throughput compression pipeline to speedup operations in comparison to a CPU only setup. In our tests we have covered a lot of possible combinations of parameters and also analyzed the performance changes of compressing images in parallel.

By the results obtained we think that the best trade off is probably given by the following parameters:

1. A high number n of images to compress in parallel (in our example we propose 16 but this highly depends on the memory constraints of the system).
2. Chroma subsampling of 4:4:4.
3. Compression quality of 90.
4. The use of non-optimized Huffman as the use of the optimized one seems to have little effects with high qualities.

For the memory type (pinned or mapped) we think that, while mapped memory could potentially offer some gains in performance compared to pinned memory, this might depend on the specific system in use so we cannot declare a clear winner. This setup is also probably the best one if execution time is the the most important factor to consider as it provides very good performance and provides reasonable results even in respect to the other factors. If one is instead interested mostly in compression ratio then it's clear that optimized Huffman and a low quality choice are the key aspects of the use case. Lastly if SSIM is the most important factor in the equation a good combination of paramters should have quality 100 and, possibly, a small resolution subsampling (such as 4:2:0) to have some gains in performance. Lastly, we emphasize that in all the possible use cases the compression of multiple images in parallel always leads to faster execution times without the need of compromises in compression ratio or SSIM so it should always be preferred, if possible.

3 NVOF

3.1 Introduction

This part of the project aimed at exploring the Optical Flow (OF) library provided by Nvidia and its performance [3]. OF is a technique to calculate the relative motion of the pixel between a pair of images, the result is a 2D vector field where each vector represents the movement of the corresponding pixel from first to second frame. Since the algorithm looks for the same pixels in different locations, the most useful use-case is to analyze a series of frames acquired from the same video, because in this situation there is coherence between the content of the frames. OF vectors are useful in different scenarios such as object detection and tracking and video frame rate up-conversion.

3.2 Work environment setup

All the tests were done using an Nvidia Quadro RTX 6000 (Turing architecture) on Ubuntu 20.04, CUDA Toolkit 12.2, Nvidia driver 535.129 and Nvidia OF SDK 5.0. The GPU that we used belongs to the first architecture that implements hardware accelerators for calculating OF, therefore it has less capabilities than its successors Ampere and Lovelace. It doesn't support 2x2 and 1x1 grid size, the maximum resolution of the images is 4096x4096, which doubles in newer GPUs, and the ROI feature is not available.

3.3 Nvidia SDK usage guide

Requirements to use Nvidia SDK [4]:

1. A Nvidia GPU from the Turing architecture or newer ones
2. Nvidia driver 525 or newer
3. CUDA Toolkit 11.1 or newer
4. Cmake 3.14 or newer
5. Freeimage 3.18 or newer

The SDK contains both the libraries and some working examples for all the different interfaces (CUDA, directx...). To run these scripts we need to download the SDK and then build the scripts. To download the SDK follow this link and once it is downloaded:

1. Extract the contents of the SDK into Optical_Flow_SDK_x.y.z
2. Create the build dir in Optical_Flow_SDK_x.y.z/NvOFBasicSamples
3. Move into build directory and run the following commands to build the scripts

4. cmake -DCMAKE_BUILD_TYPE=Release \
   -DCMAKE_INSTALL_PREFIX=. ..

5. make

6. sudo make install

The last step build all the scripts for the diferent interfaces (CUDA, directx..); to build only the CUDA script execute "make AppOFCuda" instead of "sudo make install". Now there should be executable "AppOFCuda" inside Optical_Flow_SDK_x.y.z/NvOFBasicSamples/build/AppOFCuda. To launch it run the command:
./AppOFCuda –input=<folder/*.png>–output=<folder/output_file>
Other useful flags are:

– preset (slow - medium - fast)

– gridSize (1 - 2 - 4)

– measureFPS (true - false)

3.4 API interface

The Nvidia OF API (NVOFAPI) provides different types of interface: CUDA, Directx11 and 12, and Vulkan, but we focused on the CUDA interface. The NVOFAPI provides 3 main functions with various parameters to set up the environment (here we show the most important only):

1. **Create()**: It creates the structure that holds the OF handle.
   Signature:
   ```
   NvOFObj Create(
       NV_OF_CUDA_BUFFER_TYPE inputBufferType,
       NV_OF_CUDA_BUFFER_TYPE outputBufferType,
       NV_OF_PERF_LEVEL preset,
       CUstream inputStream,
       CUstream outputStream,
       ... )
   ```
   – **inputBufferType** [in]: it specifies the format that the input buffer of frames will have when used in OF execution. The value could be either cudaAyrray or cudaDevicePtr.
   – **outputBufferType** [in]: it specifies the format that the output buffer containing OF result will have when used in OF execution. The value could be either cudaAyrray or cudaDevicePtr.
   – **preset** [in]: it could be set either to slow, medium or fast. It changes the quality/performance trade-off of the OF, starting from the slow level, which produces the best result extending the computation time, up to the fast level, which has the opposite features.

- **inputStream** [in]: CUDA stream used for copying the image from the host to the device. It can be omitted to use the default stream.
- **outputStream** [in]: CUDA stream used for copying the result from the device to the host. It can be omitted to use the default stream.

2. **Init()**: It sets up the grid size for the OF.
   Signature:
   ```
   void Init(
       uint32_t gridSize,
       ... )
   ```
   - **griSize** [in]: It can be set either to 1, 2 or 4. It represents the size of the square of pixels on which the OF is calculated. Ideally, the OF is calculated for each pixel (grid size is 1), but in practice the Turing architecture is not capable of doing this, instead it uses a grid size of 4, which means to consider a grid made of squares of 4x4 pixels, and it calculates the OF on this coarser version of the image. If I specify the value 1, the GPU still calculates OF with 4x4 grid, which leads to a result that is $\frac{1}{16}$ of the original image, therefore it should be upsampled to obtain an output with a coherent dimension with respect to the original image. This upsample is not done by the GPU but the developer must take care of it. More modern architecture like Ampere and Lovelace allows at a hardware level to calculate OF with even 1x1 grid size.

3. **Execute()**: It performs the OF calculation.
   Signature:
   ```
   void Execute(
       NvOFBuffer* image1,
       NvOFBuffer* image2,
       NvOFBuffer* hintBuffer,
       NV_OF_ROI_RECT* ROIData,
       ... )
   ```
   - **image1** [in]: Starting frame for OF calculation.
   - **image2** [in]: Ending frame for OF calculation.
   - **hintBuffer** [in]: If a ground truth is available, that is, a correct result which is known a priori, it could be used as a hint during the calculation of the OF, in order to increase the accuracy of the result.
   - **ROIData** [in]: If I'm interested in just a portion of the frame, I can specify it with this parameter and the GPU will calculate OF only in that portion of the image, with a consequent boost of the performance. Turing architecture doesn't have this capability.

The output format that Nvidia uses to encode its OF it's based on the one proposed by Middlebury [2]. The output files have the .flo extension and they contain:

1. 4 bytes of header
2. 4 bytes that specify the width of the result
3. 4 bytes that specify the height of the result
4. two matrices which store respectively the horizontal and vertical components of the OF vectors, stored as float in little-endian order

3.5 Tests

3.5.1 Tests with Nvidia example script

We carried out various tests to understand how the library behaves with different types of input. For this part we used the script provided by Nvidia inside the SDK, which is an already working example for calculating OF. We started with a qualitative analysis of the output produced on different frames with different preset level, which is the only parameter that affects the quality of the result. We started by extrapolating some frames from a Youtube video, but the result turned out to be of poor quality. In the frame in fig. 8 you can see an ant which is carrying a leaf, but the OF generated doesn't reflect the movement of the ant, which is moving towards the bottom-left corner of the screen. One possible reason could be that the compression that Youtube applies on its video leads to a poor quality OF.

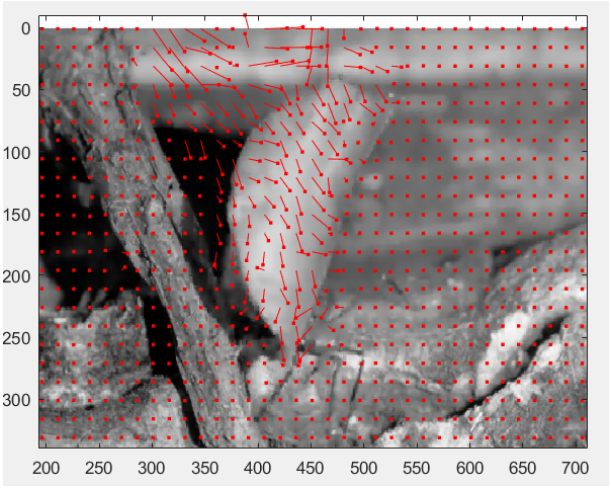
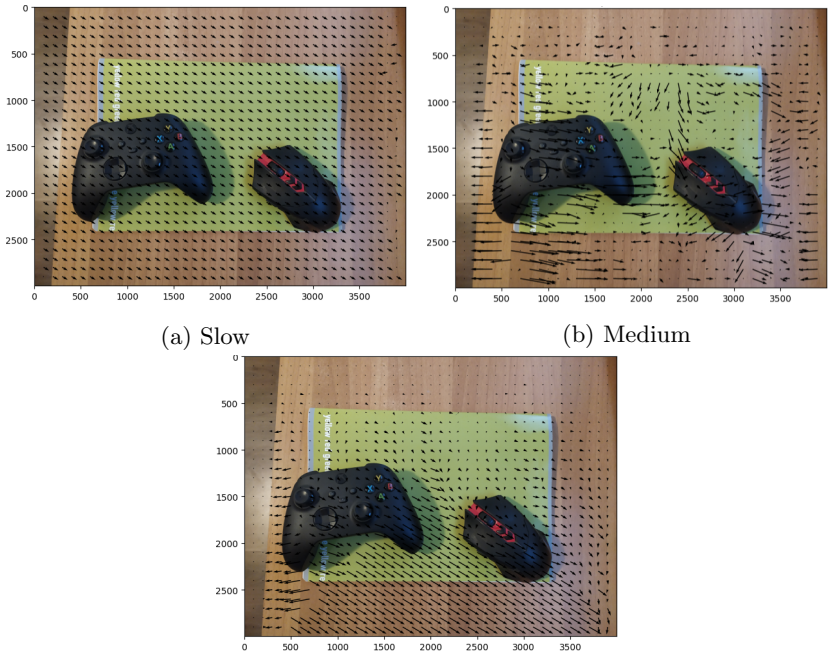


Figure 8: Youtube frame

To avoid this problem we tried a simpler scenario: we built a pair of images in order to obtain a fixed result which is known a priori. To obtain this result, we traslated an image by 250 pixel to the right, and 150 pixel to the bottom. Here we started noticing that the library struggles when it encounters a portion of the image which is of the same color, e.g a wall or the floor. This problem is evident in this test in fig. 9: all the vectors should point in the same direction, but only the slow presets correctly identifies the traslation, while the other introduce a lot of errors on the entire surface of the table and also on the two objects.



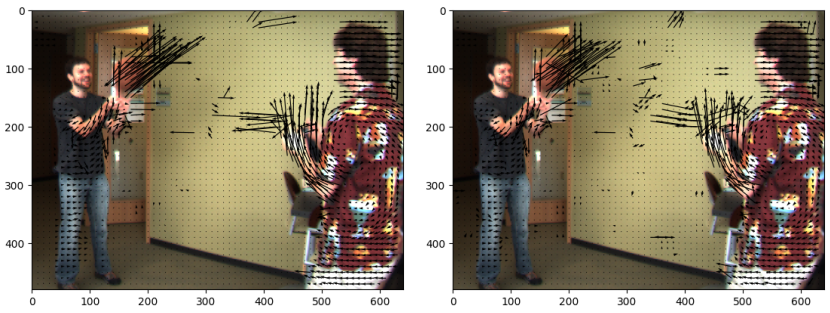
(a) Slow

(b) Medium

(c) fast

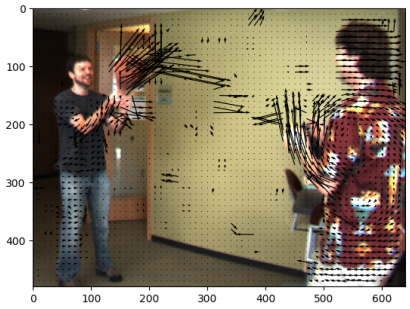
Figure 9: Traslation frame

We tested other images provided by Middlebury in a small dataset [1] available on their website. The problem appears also with these new images, and in particular in the basket frame in fig. 10, where all the three presets produce strange vectors on the wall in the background. Despite this issue, both slow and medium presets produce an acceptable result, since they manage to correctly identify the movement of the ball, and also the movement of the hands of both men. The fast preset produces instead many errors on the entire image, and also the movement of the ball is much less precise, with many vectors that points towards the right side of the image, rather than following the real trajectory of the ball.



(a) Slow

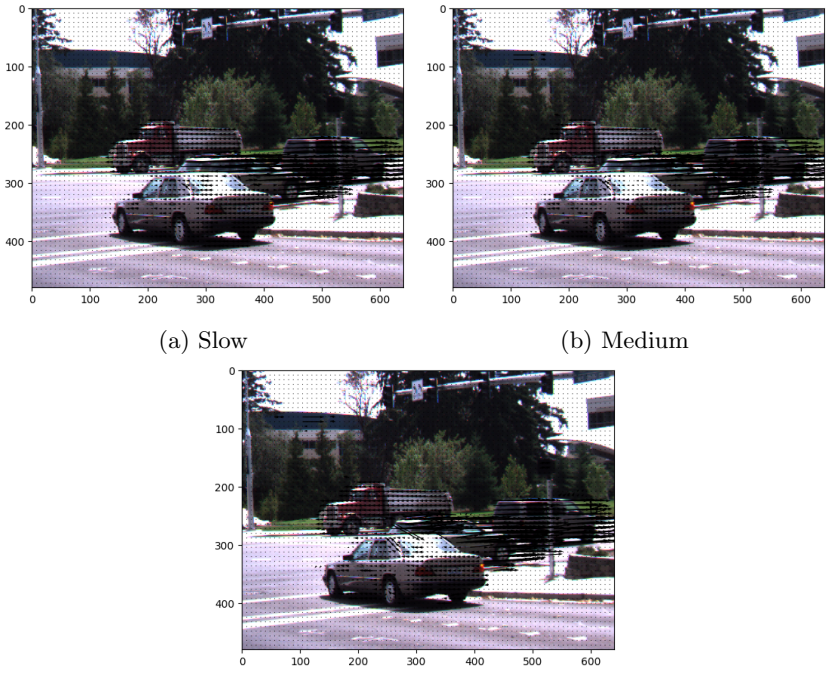
(b) Medium



(c) Fast

Figure 10: Basketball frame

We tested another set of frames from the same dataset, which represent some cars at a crossroad. In this situation the library behaves very well, the movement of the cars are correctly identified by all the presets, and only the fast one produced minor errors which are visible on the car in the foreground and on the roof of the building in the top-left corner.



(a) Slow

(b) Medium

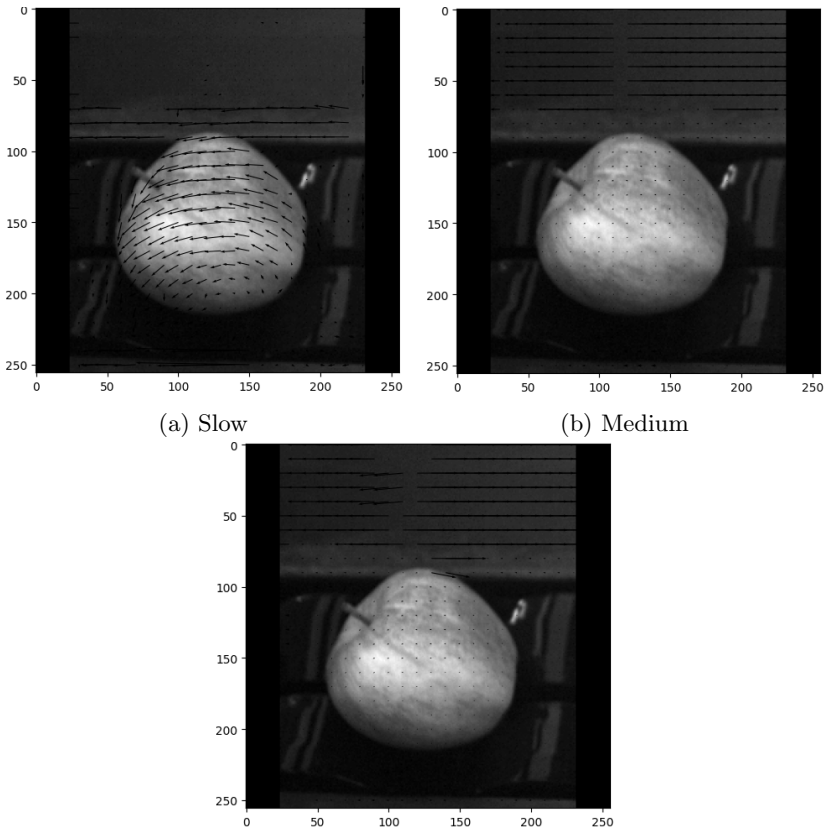
(c) Fast

Figure 11: Cars frame

3.5.2 Tests with our script

Up until now, all the tests that we carried out were done using the script povided by Nvidia in its SDK. The program works very well but it has one big limitation: it accepts only PNG images. That's because the additional library that Nvidia uses to manage the input images provides a loader class only for PNG files. To solve this problem we developed a much simpler script which get rid of all the additonal libraries of the Nvidia's example. We use the library stb_image [8] to load generic image files, and we use the images handler provided by the library to directly load the frames inside the input buffer, which is then used by the NVOFAPI. This allows us to load also BMP and JPEG images, both RGB and black and white. When the result is ready the API write it in the output buffer and we proceed to do an upsample of the result if needed, i.e if the specified grid size is 1 or 2; this step is needed in order to obtain a result with a coherent dimension w.r.t the frames which have generated it. If we had a newer GPU which could use also 2x2 and 1x1 grid size, an upsample step would not be needed. The last pair of tests is done using our script to load black and white, BMP frames.

In the example in fig. 12 there is an apple which is slowly rotating around its axis. It's evident that neither the medium nor the fast presets were able to identify the motion of the apple but, instead, they still produce the same errors on the background that we have seen on many previous examples. Only the slow preset correctly identifies the rotation of the apple, despite some imprecision just above the apple.

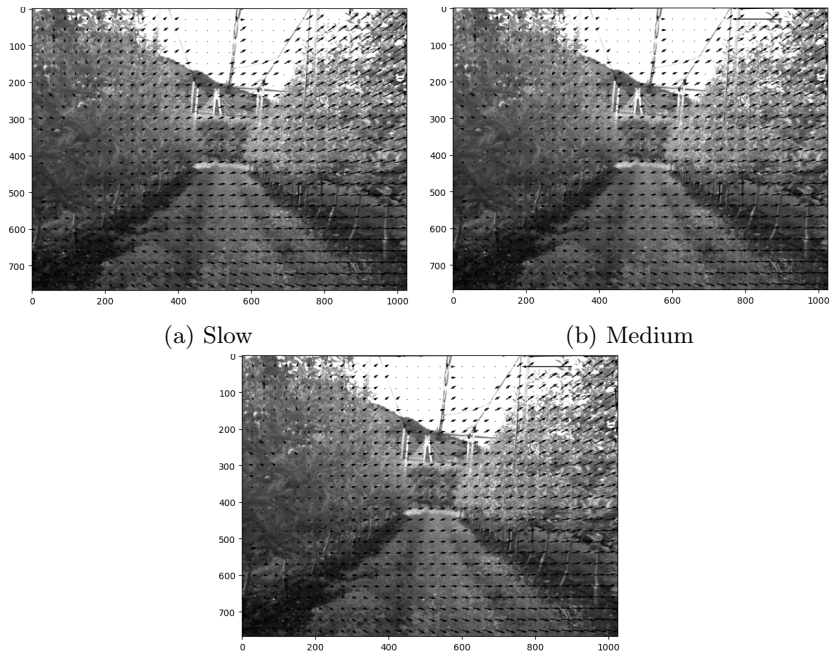


(a) Slow (b) Medium

(c) Fast

Figure 12: Apple frame

The last test is actually the one which produced the most precise result. All the three presets correctly identify the movement that the camera does in the sequence; there aren't visible errors produced except for a vector in the top-right corner in medium and fast preset, and the difference between slow and fast preset is incredibly low, especially if we compare with previous examples.



(a) Slow (b) Medium

(c) Fast

Figure 13: Field frame

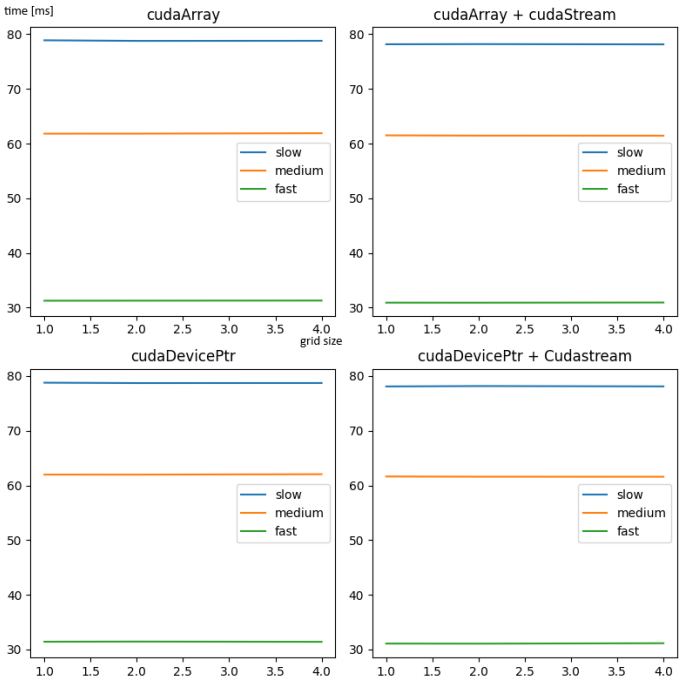
3.5.3 Performance analysis

After a qualitative analysis of the results produced by library, we finished this exploration of the Nvidia OF library by doing some quantitative consideration about the performance achieved by the library. All the measurements have been done using the Nvidia example script, which already contains the option to measure the time for the OF calculation. The results are shown in figs. 14 to 16 and they are firstly grouped by input_type_buffer and output_type_buffer, and then by preset level. We tested the following parameters:

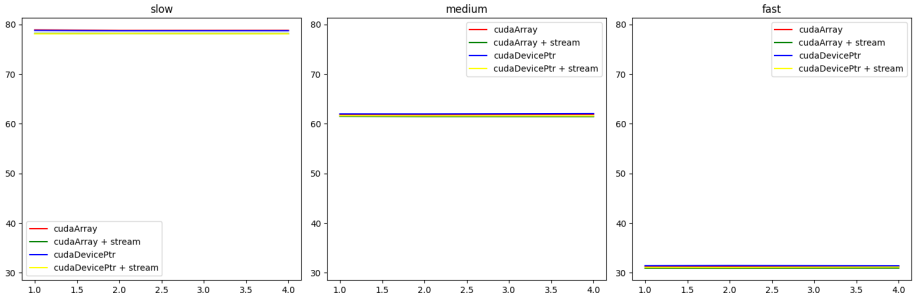
1. preset level (slow - medium - fast)
2. input/output buffer type (cudaArray - cudaDevicePtr)
3. grid size (1 - 2 - 4)
4. with and without CUDA stream

We tested sets of 4, 16 and 32 frames. In Figure 14 we can see that for a low number of frames there is no practical difference between the different buffer types; there is instead for different preset level, which clearly take different time to execute and produce a visible difference as we have seen during the qualitative analysis. The grid size parameter doesn't affect the time by any mean, this is due

to the fact that the RTX 6000 is capable of doing only the 4x4 OF calculation as discussed earlier. With 16 and 32 frames the differences between the setups become visible; even if the grid size is always 4, using different types of buffer lead to different execution times which are still quite similar between the setups. The preset level is still the most meaningful parameter, with a big jump from fast to medium, where the time almost doubles, and a smaller but still visible gap from medium to slow. From these results it's clear that the preset level is the most critical parameter when dealing with execution time, while the others have almost no impact. We think this is a reasonable behaviour because all parameters but preset level are not directly involved in the OF calculation but they just provide a way to customize the environment around the actual OF process. The preset level, instead, directly specifies how the OF calculation is done and it's logical that it has an impact on the execution time. The grid size should also be a relevant parameter for the performance because with a 4x4 grid, the image is considered as if it was $\frac{1}{16}$ of its actual size, but this would be noticeable only with newer GPUs which can actually perform a 2x2 and 1x1 OF calculation.

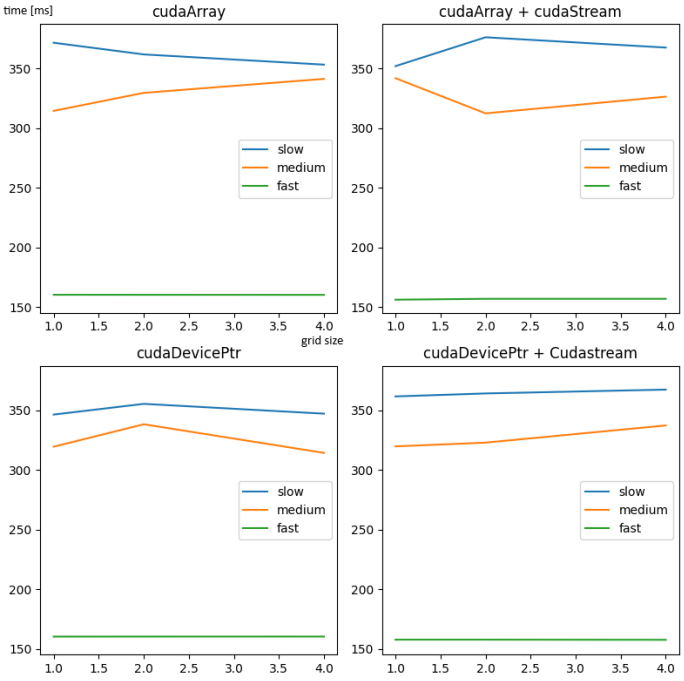


(a) Buffer type

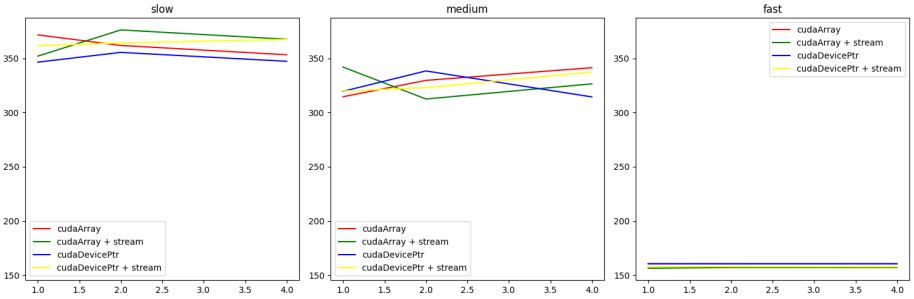


(b) Preset

Figure 14: Test with 4 frames

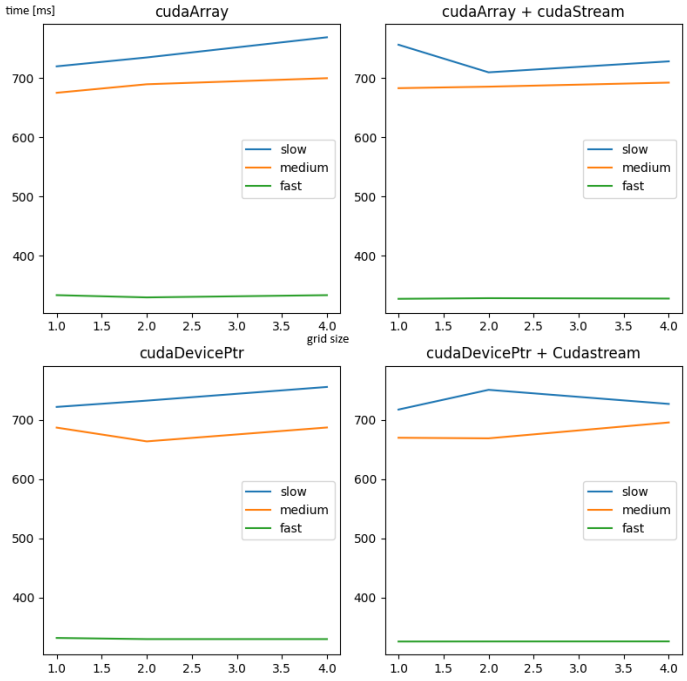


(a) Buffer type

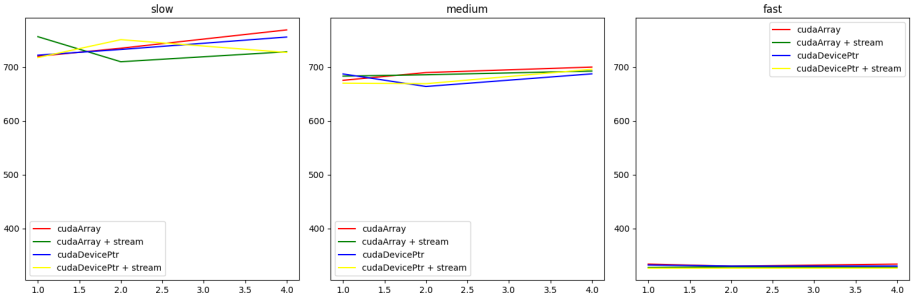


(b) Preset

Figure 15: Test with 16 frames



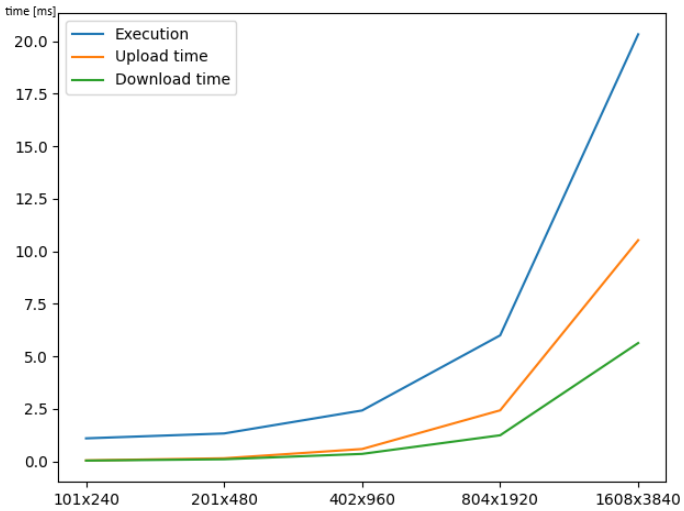
(a) Buffer type



(b) Preset

Figure 16: Test with 32 frames

For the last test we fixed the preset at slow and we measured the execution time and the upload/download time of the image to/from the GPU. Width and height of the images are doubled at every step, therefore the total resolution quadruples at every step; we had to stop at 1608x3840 pixel because the RTX 6000 is limited at 4096x4096 pixels. The behavior of upload and download time follows this pattern, since at each step they quadruple as well; the execution time instead has a different behaviour: until the last step the time tends to double, while on the last one it actually quadruples as it should do. This is probably due to the fact that initially the images are so small that the GPU manages to handle them faster than their grow rate; only at the last step the dimension become so large that it actually has to quadruple the execution time.



(a) Time for execution, upload and download

3.6 Conclusions

What emerged from our analysis is that the NVOFAPI produces acceptable results only if the slow preset is selected; the other presets often produce wrong or partially wrong results which would require a heavy post-processing to be used in other contexts. The execution time is indeed longer but we think it is an affordable trade-off in this specific scenario, considering that both medium and fast preset produce poor quality outputs. We think that also the grid size would be an extremely important parameter but, unfortunately, the GPU we used was limited at a grid size of 4. In out tests we didn't notice a big impact of the other parameters on the total cost of OF execution, therefore they should be tested in every different scenario to choose the optimal configuration. We didn't completely understand why the library struggles when it elaborates a big portion of an image which is of the same color but we suppose that, since the algorithm look for pixels which have the same content, in this particular situation it can't precisely understand which pixel is mapped into which because they are very similar with each other.

4 Conclusion

In conclusion we think that both libraries could be a valid choice for their respective use-cases, since they are built to exploit all the power of Nvidia GPUs and they also give the user the freedom to customize the environment through all the parameters that the APIs expose. In our opinion the NVJPEG library is the simplest to adapt to the various scenarios and to find the correct configuration of the parameters; NVOF instead requires more work in general, especially to understand how to set up the parameters in order to obtain a high-quality output; the inputs themselves should not contain specific structures to avoid errors in the output (e.g big portion of the image which is of the same color)

References

[1] Middlebury dataset. https://vision.middlebury.edu/flow/data/.

[2] Middlebury .flo format. https://vision.middlebury.edu/flow/code/flow-code/README.txt.

[3] Nvidia SDK. https://developer.nvidia.com/optical-flow-sdk.

[4] Nvidia SDK documentation. https://docs.nvidia.com/video-technologies/optical-flow-sdk/index.html.

[5] NVJPEG. https://developer.nvidia.com/nvjpeg.

[6] NVJPEG documentation. https://docs.nvidia.com/cuda/nvjpeg/index.html.

[7] Skimage library. https://scikit-image.org/.

[8] Stb_image github page. https://github.com/nothings/stb/blob/master/stb_image.h.