Analysis of Template Matching by Comparing C++ Concurrency with CUDA and OpenCV

Aditya Immaneni[1], Victor R. Cabrera[1], Vadim Pinskiy,[1] and Matthew C. Putman[1]
[1]Nanotronics Imaging
{aimmaneni, vcabrera, vpinskiy, matthew} @nanotronics.co

Introduction

Template matching is a baseline technique for comparing acquired images to a known standard. Incorporated as part of OpenCV and other primary libraries, template matching allows for registered images to be used in the same coordinate space and individual attributes to be referenced as such. Currently, our implementation of template matching through OpenCV fails to scale for high-throughput applications.

Due to this, we implemented a hybrid version of CUDA's C++ API alongside the modern C++ concurrency extensions in the standard library, with the goal of achieving considerable improvement of performance and scale.

Goals

1. Integration of NPP with multithreading

Utilize the NVIDIA Performance Primitives (NPP) library, a CUDA C++ framework, in order to have more direct control over CUDA operations, with the goal of optimization and speed ups for our use cases. Working closer to CUDA would allow us more control when working with different image types and different algorithms, which should allow us to better utilize streams and multithreading.

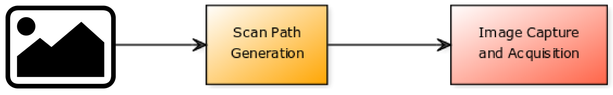
2. GPU Wrapper Class

Create a wrapper for GPU memory, that allows us to store all the data related to that memory together, such as the image size and width, and what device it is currently allocated to. Following the Resource Acquisition is Initialization (RAII) idiom, we can enforce the correct device on startup, allocate the proper memory for GPU operations, and release memory as needed.

Process and Motivation

Our process cycles from hardware image capture, through data storage, GPU based processing and archival storage and image retrieval for further analysis.

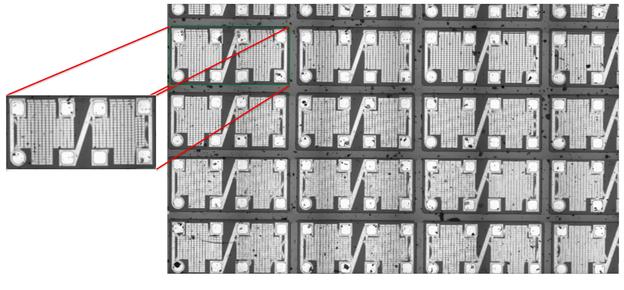
1. Acquisition



2. Processing

3. Storage

The goal of template matching is to find a correspondence between a reference image and a subset of the acquired image. Translation and rotation are free parameters; while scale is fixed. We allow a customizable threshold to account for acquisition noise.

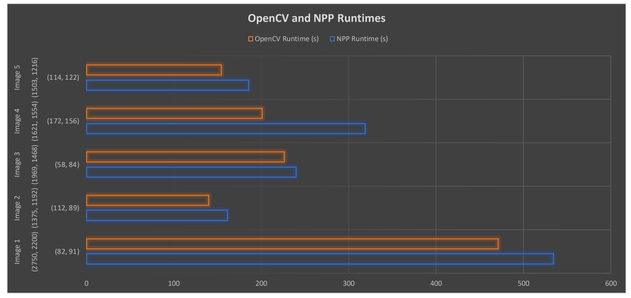


Data

1. Test Data:

We ran 10000 images in total, measured by selecting 5 images to run 2000 times each.

Image	Image Size(W,H) pixels	Template Size(W,H) pixels	NPP Runtime (s)	OpenCV Runtime (s)
Image 1	(2750, 2200)	(82, 91)	534.32	471.09
Image 2	(1375, 1192)	(112, 89)	161.49	139.88
Image 3	(1969, 1468)	(58, 84)	239.77	226.61
Image 4	(1621, 1554)	(172, 156)	318.82	201.01
Image 5	(1503, 1216)	(114, 122)	185.93	154.45

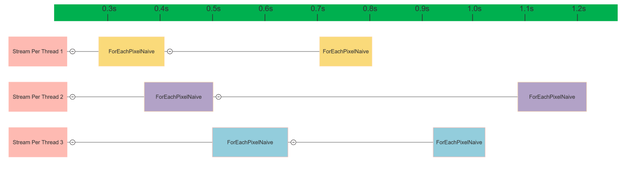


- NPP Total Duration **(s)**: 1440.32
- OpenCV Total Duration **(s)**: 1193.05

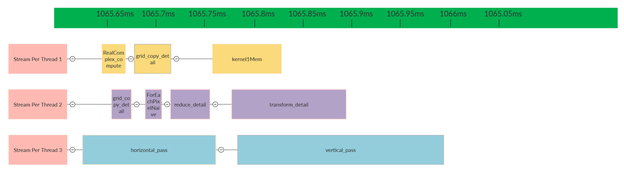
2. NVIDIA Visual Profiler

We ran our application through the NVIDIA Visual Profiler to obtain data regarding the computation performed by NPP and OpenCV. The data below is summarizing a subset of the template matching algorithm.

NPP Visual Profiler



OpenCV Visual Profiler



Analysis

- OpenCV performs 17.1% better than NPP for template matching on monochrome images
- Template matching performed faster for single-threading than multi-threading for both OpenCV and NPP for our use cases. The stream overhead for both NPP and OpenCV causes the template matching to take longer
- In the visual profiler, OpenCV splits the computations performed during template matching into many smaller functions, while NPP does many computations in the span of a single function

Conclusion

- This work established that OpenCV based template matching is a well optimized method to achieve highly efficient continuous processing
- For continuous operation, other OpenCV functions, such as merge and gaussian kernels
- While OpenCV performed better than NPP on all our images, our analysis of these approaches to using the GPU through CUDA showed us how to better approach utilizing the GPU through small, parallelizable operations, which can be applied through a more manually constructed template matching algorithm with the CUDA C++ API

Acknowledgements

Special thanks to the entire Nanotronics technical and research staff.