GPU and Multi-Threaded CPU Enabled Fast Normalized Cross-Correlation

*Nafis Ahmed¹, *Evan Teters¹, Rumana Aktar¹, Ismail Akturk¹, Guna Seetharaman², and Kannappan Palaniappan¹

*Contributed equally

{na3x5, ejt8tc, rayy7}@mail.missouri.edu, {akturki, pal}@missouri.edu

¹Department of Electrical Engineering and Computer Science

¹University of Missouri - Columbia, Missouri

²US Naval Research Laboratory, Washington D.C., USA

ABSTRACT

Image matching has been a critical research topic in many computer vision applications, such as stereo vision, feature tracking, motion tracking, image registration and mosaicing, object recognition, and 3D reconstruction. Normalized Cross Correlation (NCC) is a template-based image matching approach which is invariant to linear brightness and contrast variations. As a first step in mosaicing, we use NCC to a great extent for matching images which is an expensive and time consuming operation. Thus, an attempt is made to implement NCC in GPU and multi-CPU to improve the execution time for real-time applications. We also show performance differences for our different parallelization techniques: dense and sparse NCC. Finally, we compare the enhancement in performance and efficiency in timing by switching NCC implementation from CPU to GPU.

Keywords: Normalized cross correlation, multi-threading, GPU, speedup

1. INTRODUCTION

Of the variety of problems for which image matching is a critical component, we are most concerned here with image mosaicing. Our previous works¹–³ show increasingly robust and capable image mosaicing on the VIRAT dataset⁴. We can create accurate mini-mosaics and show the potential to create a mosaic of an entire scene with a very large aerial imagery dataset. Examples of frames from VIRAT dataset and mosaics are presented in Figure 1 and Figure 2, respectively.

This paper is concerned with the time performance of our image mosaicing. Our current pipeline takes around 5.77 hours to generate 10 mini-mosaics from 9290 frames. This method considers a fused descriptor for image matching and registration combining Structure Tensor-based feature selection and NCC-based feature matching followed by ASIFT matching when necessary.¹ The major bottleneck of this approach is the use of NCC for matching prominent features that takes around 3.55 hours (68.31% of total time). This implementation is entirely for CPU, and is implemented in MATLAB. In an effort to increase the time efficiency, we parallelized NCC on CPU and conducted multi-threaded NCC computations, as well as GPU-enabled (CUDA) implementation. We have achieved roughly 40x speedup in NCC computation by using concurrent 28 threads, and an overall 4.9x speedup in mosaicing using multi-threading (20 threads) compared to standard MATLAB implementation on the CPU.

Depending on the type of GPU, we also achieved a speedup of at most 81.1x for the NCC. This is not yet integrated into the overall mosaicing, but should be capable of increasing the overall performance even more.

Section 2 lists related works, section 3 discusses NCC and why it is useful for us, section 4 describes our methods for implementing these parallelized methods, and 5 presents our analysis and results that compare different methods. Finally 6 presents future work and concludes the paper.
2. RELATED WORKS

There is a long history of image stitching techniques in the literature. A survey from Microsoft Research in 2006 splits the area up into different kinds of methods. Our method, especially with NCC, falls under the Feature-Based Registration approach. We recommend the reader see our previous work for exploration on methods related to our mini-mosaic method, which helps reduce image drift. Our pipeline also utilizes image geoprojection and stabilization. For more on making this faster, see our previous work.

Normalized Cross Correlation (NCC) is a powerful feature matching tool. It has been used for quite some time, and so many methods have been proposed to speed it up for template matching. Lewis proposed a method to speed it up using the pre-computed integral histogram, which many standard implementations (including the ones we use) take advantage of. Briechle and Hanebeck demonstrated NCC for template matching that achieves less computation by approximating the numerator. In 2009, Yoo and Han introduced a significantly faster signal-processing based method that takes advantage of logic operations to compute NCC without any multiplies. This method is more sensitive to noise, however. Gangodkar et al., parallelized the Fast NCC (which is based on pre-computed sum-tables to mitigate the computational complexity of conventional NCC) on CUDA-based GPU. NCC is parallelized across FPGA’s by Wang and Wang, although the implementation is hardware-specific. They do show the benefit of parallelizing the computation, as our GPU implementation does. In work on a tracker by Jakob Santner et al., they showed good results by also using the OpenCV NCC for the static part of their tracker, as it gives strong cues when the target reappears. They do not analyze or parallelize the NCC, however. Arunagiri and Jaloma demonstrated that for stereo matching, a NCC based cost function is fastest and consumes less energy when implemented using integers rather than floating point numbers on the GPU. They do not try other methods of parallelization or focus specifically on NCC, however. Fouda and Ragab parallelized the NCC by using OpenMP for shared-memory systems, but did not explore the parallelization on GPUs, in contrast to our work. Further experiments with NCC itself are presented as future work for this paper, in which we would implement NCC ourself. As it stands, the APIs we use for NCC appear to all use the integral image optimization and the denominator optimization presented in Lewis’ work. Our main consideration at the moment is parallelizing NCC, and understanding what improvement to expect.

3. NORMALIZED CROSS CORRELATION

Normalized Cross Correlation (NCC) is a popular measure of similarity between two images or image blocks. It is less sensitive to absolute intensity changes than other methods, but is quite expensive to compute.
Figure 3. Example of Reference and Search blocks in VIRAT image pairs. (a) presents template/reference blocks in reference frame; two sizes of reference blocks are used: ST blocks of size 40x40 ($B_1$, $B_3$, $B_4$ and $B_5$) and NCC blocks of size 82x82 ($B_2$). (c) shows search blocks in source/current frame; five sizes of search blocks are used: $B_1$ of size 91x142, $B_2$ of size 184x184, $B_3$ of size 142x91, $B_4$ and $B_5$ of size 142x142, and of size 91x91 (not shown here). (b) presents the padding around search window used in MATLAB before the correlation matrix is computed.

Figure 4. Example of NCC computation in our mosaicing algorithm for a template/reference block with source/current image: (a) template/reference block, (b) source/current image, (c) cross correlation between template block and source block, and (d) template is matched and highlighted on source image.

The NCC between a template (or reference) image block $T(B, x, t - k)$ from the frame at time $(t - k)$ and search (or current/source) image block $I(B, x, t)$ from the frame at time $t$, is defined as

$$
\gamma(B, \Delta x_B, t) = \frac{\sum_{x \in B}[T(B(x), t - k) - \mu_T(B(x), t - k)][I(B(x + \Delta x_B), t) - \mu_I(B(x + \Delta x_B), t)]}{\sqrt{\sum_{x \in B}[T(B(x), t - k) - \mu_T(B(x), t - k)]^2 \sum_{x \in B}[I(B(x + \Delta x_B), t) - \mu_I(B(x + \Delta x_B), t)]^2}}
$$

(1)

where $\mu_B(B, t - k) = \langle I(B, x, t - k) \rangle$ and $\mu_B(B, t) = \langle I(B, x + \Delta, t) \rangle$ are the local intensity means in the target and template image regions, respectively. An example of NCC computation and template matching is presented in Figure 4.

We perform the NCC in a sliding window fashion on an area in order to determine the best match for a given larger current block and smaller reference block. Because of this sliding window computation, even when we narrow the current/source block down to an area around detected features, we have a great number of multiplies and adds. Element-wise multiplication $[I(X + \Delta X, t - k) - \mu_{t - k}][I(X, t) - \mu_t]$ results in $size(I)$ multiplies, and the sum is another $size(I)$ adds. On the bottom, the square roots are both $size(I)$ multiplies, and the sums are another two. This is 6 * $size(I)$ operations per time we compute the NCC. A typical size of the reference is 40x40, so 1600 pixels. To sweep over a current frame of size 142x142 pixels, we would need to do 103 NCC operations (no out-of-frame padding needed). This is 6 * 1600 * 103 = 988,800 operations for one block. Our main pipeline is roughly 8000 frames with about 120 blocks per frame, amounting to a total of 1,067,904,000,000 operations (not to mention that 8000 frames have to be loaded, although these are fairly small - 720x480 pixels). This changes, of course, with the use of integral image and the Fourier transform. These helps in employing
optimization techniques which brings down the linear computation required for each of the sum operation to almost constant time and allows for the algorithm to be much faster with far fewer operations. The calculation of NCC is a serial portion of the program that Amdahl’s law says is the limit of the parallelization. Our goal with parallelism is to allow many of these NCC operations happen concurrently, or in the case of the current GPU implementation, allow a single NCC calculation itself to use multiple threads.

4. METHODS

The dataset we tested on has a total of 9292 images. The video sequence is temporally segmented into 10 shots (scene) with a total of 8131 frames. Other frames are discarded as those are uneventful. In order to match a pair of images known as Reference (or template) and Current (or source), 120 prominent feature blocks are extracted from Reference frame and each block is matched using correlation within a search window in Current image. The dominant size of a reference block is (40*40) and for search window, it is (142*142). Each of our methods were run 50 times over this full data-set, to measure the computation time and speed-up with respect to the original MATLAB implementation.

4.1 Multi-Threaded CPU implementation

There are a few good libraries upon which a multi-threaded CPU implementation can be built, like Boost, OpenMP, Pthreads, etc. We have selected two widely used library and implemented Multi-Threading using both of them. One of them are standard C++ threads, which was built on top of Pthreads, and introduced to the standard C++ library with the release of C++11. The other multi-threaded implementation has been done with OpenMP, which is widely used library when it comes to multi-processing in C++ and Fortran.

For both of the implementation, we have created a function which takes a reference patch of an image, and a search window patch, or two images, the first one is the reference image and the second one is the current image which will be compared with the reference block, coupled with the reference block coordinates and the search window coordinates, And outputs the coordinate for the best match using the normalized cross correlation and it’s normalized cross correlation score.

For calculating NCC with the multi-threaded CPU implementation, we have used the `matchTemplate` function from the well known and widely used OpenCV library, and we have used OpenCV v3.4.8 for our implementation. The `matchTemplate` function call has most widely known optimization, i.e., Integral Image usage, already implemented inside it, and has various methods for calculating NCC, which can be selected while calling the function. For our purpose, we have used the method ‘CV_TM_CCOEFF_NORMED’ for calculating NCC with the `matchTemplate` function. The `matchTemplate` function takes two images, the first one is the input (i.e., Current) image, the second image is the Reference (i.e., Template) image, and the method, which will be used to calculate the NCC matrix. The output of this function is a correlation matrix, from which we can determine the best matched block from the scores from each cell of correlation matrix. If I is the source image, T is the template image, and R is the correlation matrix, then the equation for NCC with the method ‘CV_TM_CCOEFF_NORMED’ becomes:

\[
R(x, y) = \frac{\sum_{x',y'} (T'(x', y') \cdot I'(x + x', y + y'))}{\sqrt{\sum_{x',y'} T'(x', y')^2 \cdot \sum_{x',y'} I'(x + x', y + y')^2}}
\]

where,

\[
T'(x', y') = T(x', y') - \frac{1}{(w \cdot h)} \sum_{x'',y''} T(x'', y'')
\]

\[
I'(x + x', y + y') = I(x + x', y + y') - \frac{1}{(w \cdot h)} \sum_{x'',y''} I(x + x'', y + y'')
\]

(2)
Figure 5. This shows the row-interleaved assignment technique. Each block’s NCC calculation is done by a single predetermined thread, so no redundant calculation is done. If there are a total of \( t \) number of threads, each having an id, \( tid \), ranging from 0 to \((t-1)\), then each thread is responsible for calculating the blocks, \( tid \), \( (tid + t) \), \( (tid + 2^*t) \), and so on.

### 4.1.1 C++ Standard Threading Library

The implementation with standard C++ threads uses a row-interleaved technique for work distribution between the threads. The total threads can be of any number, but it results in the best performance when it is equal to the number of CPU threads. The row-interleaved technique can be described as follows. If the total number of threads are \( T \), and there are a total of \( N \) blocks, then the first block is assigned to the first thread, the second block is assigned to the second thread, and likewise the first \( T \) blocks are assigned to each of the \( T \) threads to perform the NCC calculation, then the \( T+1 \) block is again assigned to the first thread, and assignment goes on like this, with the final block of \( N \) is assigned to the thread \( (N \mod T) + 1 \).

### 4.1.2 OpenMP

OpenMP is a shared memory multi-processing programming interface, which uses a set of compiler directives, routines and environment variables to influence the runtime behavior of a program. For the implementation using OpenMP, we have used the same function to calculate the coordinate with the best NCC score, and initiated it’s call from a standard for-loop which has been preceded by the OpenMP ‘for’ directive. We have used the specification ‘schedule(static, 1) nowait’ for this particular directive, so that it’s task assignment to threads is similar to the implementation with the C++ standard library. Here, the ‘static’ keyword implies that each of the threads will be statically assigned to a core. Also, a chunk size of 1 is used, so that each block of NCC calculation is considered as a single chunk of work. The ‘nowait’ keyword is used to speedup of calculation, as we don’t need the threads to synchronize after each iteration.

For both of the implementation, the algorithm, 4.1.2, remains the same, as only the multi-threading part is different for the two implementation with the two multi-threading libraries.

### 4.2 GPU Implementation

For the GPU implementation, we use Nvidia Performance Primitives (NPP), a CUDA API that allows for easy CUDA programming. The various APIs handle tasks on the GPU to accelerate performance.

NPP uses managed memory to make it easier for the programmer to transfer data to and from the GPU, and use their APIs on the GPU. Our implementation must load the relevant images onto the GPU’s memory.

To test the GPU as a comparison to the multi-threaded OpenCV versions, we use the NPP API ‘nppiCross-CorrValid_NormLevel_32f_C1R’. This returns the correlation matrix, as shown in Figure 4, from which we can compute the max. This is still on the GPU, so we use ‘nppiMax_32f_C1R’. Both of these functions require use of buffers which use GPU memory, so there is some slight memory overhead for allocating the right amount of space for the correlation matrix and the necessary buffers.
Algorithm 1 Multi-threaded calculation of NCC for all reference/current frame pairs

Load all of the Block-Information files into BlockInfoList:
Having all frame pairs, and the bounding box pairs (blocks) needed within frames

for BlockInfo in BlockInfoList do
    Load currentFrame and referenceFrame if different from previous BlockInfo
    for Block in BlockInfo do
        Inside the assigned Thread determined for this Block do
            Start Timing
            \( \Gamma = \text{NCC}(\text{currentFrame}, \text{referenceFrame}) \)
            End Timing
            \( \gamma_{opt}, (x,y)_{opt} \leftarrow \max(\Gamma) \)
            Save \( \gamma_{opt}, (x,y)_{opt} \)
        end for
    end for
end for

Algorithm 4.2 shows our pseudocode for the GPU implementation. Importantly, our timing throughout this paper has been focused on the actual NCC calculation. There may be some overhead associated with GPU implementation and integration (mainly) due to data staging in/out, but we wanted to analyze only the timing incurred by computing the NCC’s for now.

We also tested OpenCV’s GPU support for the \textit{matchTemplate} function. This was done on a GeForce GTX 1080 with OpenCV 3.4 compiled with CUDA 10.1 support.

Algorithm 2 GPU Calculation of NCC Coefficients for all reference/current frame pairs

Load all csv files into BlockInfoList: Know all frame pairs, and the bounding box pairs (blocks) needed within frames.

for BlockInfo in BlockInfoList do
    Load referenceFrame if different from previous BlockInfo referenceFrame
    Load currentFrame if different from previous BlockInfo currentFrame
    Allocate all necessary space on the GPU, including with \( \gamma_{opt} \) with size \((\text{currentFrame}.\text{width} - \text{referenceFrame}.\text{width} + 1, \text{currentFrame}.\text{height} - \text{referenceFrame}.\text{height} + 1)\)
    Start Timing
    \( \Gamma = \text{NCC}(\text{currentFrame}, \text{referenceFrame}) \)
    End Timing
    \( \gamma_{opt} = \max(\Gamma) \)
    Save \( \gamma_{opt} \) and \( \Gamma \) to compare against current CPU results.
    Free GPU resources, unless they are the right size to be reused in the next calculation.
end for

5. RESULTS & ANALYSIS

5.1 Methods
For timing analysis of our methods with respect to the original MATLAB implementation, we selected three main factors. As stated previously, our dataset consists of 8131 image files, which contains a total of 933538 blocks. The factors we have chosen for our timing analysis are, \textit{Total – Time}, the time required to calculate NCC matching for all of the 8131 images, \textit{Time – Per – Block}, the average time required to calculate NCC matching over a single block, and \textit{Time – Per – Frame}, the average time required to calculate NCC matching over a source image, searching for the template image in it. It can also be described as the average time required to calculate all of the blocks in a single image file.
For running our CPU based multi-threaded implementations, we have used an Intel(R) Xeon(R) CPU E5-2680 v4 @ 2.40GHz server with 14 cores with Hyper-Threading, so essentially 28 threads, and 252 GB of RAM.

To experiment on the GPU, we tested with two different GPUs with varying amounts of Streaming Multiprocessors (SMs) and CUDA Cores. First was a GeForce GTX-1080 GPU. This has 20 SMs containing a total of 2560 CUDA Cores. Second was a Tesla V100, with 5120 CUDA Cores across 80 SMs. Because of increased interest in performing computation on the edge, we also tested on a Nvidia Jetson Xavier AGX. This unique device is very small and medium power. Its uses a GPU running NVIDIA Volta architecture. It contains 512 Nvidia CUDA Cores across 64 Tensor Cores.

We also tried our multi-threaded code on the Xavier. It contains 8 CPU cores in a Carmel ARM v9.2 CPU which has a memory bandwidth of 137 GB/s.

5.2 Results

For our two separate multi-threaded CPU implementation, in the Table 1, we have only shown the results for the implementation that used C++ standard threading library. As we will also show, the timing differences between the two implementations, C++ standard threads and OpenMP, tends to be minimal as the number of threads increases. We have also tested on threads numbering from 1 to 27, but for the timing comparison, we have selected to show only the results of four instances, and included the timing results when running on a single thread, and when running on 7, 14 and 27 threads concurrently. From the selected set of timing for different number of threads, the actual timing trend for all of the threads can be easily seen. The top row of Figure 6 describes the timing comparison for all of the thread configurations.

It is evident from Table 1 that different threading libraries, like C++ standard threads and OpenMP, for multi-threaded implementation, performs very similar and has little effect on the total time for running NCC computations as the number of threads increases. From here on, we will only focus on the multi-threaded implementation with the C++ standard threads library.

Table 1 shows that switching to OpenCV, from MATLAB, gives some speedup straightaway, with no parallelism added. Further, it shows that we have a speedup as we increase the number of threads, but the speed increase does not increase linearly as the thread count increases. Finally, it shows that there is a speedup when using the GPU, but that speedup is more conservative than expected. This is because the NPP API does not take advantage of the full breadth of parallelism available, since it only parallelizes the given single NCC operation. The V100 shows a much better improvement, likely due to better hardware specification. In Section 6, we discuss how NPP may become restrictive to parallelize NCC to its full potential on a GPU.

5.3 Comparing the Output

We computed the difference of the NCC results from the original MATLAB implementation with our multi-threaded CPU, and the GPU implementations using NPP. The correlation matrix result for the GPU implementation using NPP is very different from what we get from MATLAB and the multi-threaded CPU implementation using OpenCV, while the MATLAB result and the result from our multi-threaded implementation using OpenCV is quite similar. Below, we measured the peak points in the NCC correlation matrix from MATLAB and OpenCV, which is essentially the best point where the reference block matches in the search window. An exact match is defined where (X-Peak, Y-Peak) is exactly same in MATLAB and OpenCV, and they have an euclidean distance of zero. In our tests, we have seen that around 94.9% of blocks per frame has an exact match or zero euclidean distance. Including closely matching block-pairs with less than or equal to 4 pixels euclidean distance increases the percentage agreement between the MATLAB and multi-threaded OpenCV NCC peak results. In our experiment, we got 99.37% of blocks with a close match or less than or equal to 4 pixel (radius) euclidean distance. Thus, only 0.63% of blocks have error greater than 4 pixels. In our experiment, we pick blocks (Structure Tensor (ST) feature points) which are far from image edges. The distance of any ST points should satisfy the following condition where $edge_{distance_{ST}}$ refers to the minimum distance of a ST point from image edge, and $SW_d$ and temp_d stands for size of search block and template block respectively.

*Speed-Up with respect to MATLAB implementation
Timing comparison between the different methods

<table>
<thead>
<tr>
<th>NCC Computation Method</th>
<th>Total-Time (s)</th>
<th>Time-Per-Block (ms)</th>
<th>Time-Per-Frame (ms)</th>
<th>Speed-Up*</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>12374.7</td>
<td>14.1</td>
<td>1522.0</td>
<td>1.0</td>
</tr>
<tr>
<td>C++ CPU - Threads = 1</td>
<td>3168.9</td>
<td>3.4</td>
<td>423.8</td>
<td>3.8</td>
</tr>
<tr>
<td>OpenMP - Threads = 1</td>
<td>2791.3</td>
<td>3.0</td>
<td>343.3</td>
<td>4.4</td>
</tr>
<tr>
<td>C++ CPU - Threads = 7</td>
<td>822.3</td>
<td>0.9</td>
<td>99.7</td>
<td>15.0</td>
</tr>
<tr>
<td>OpenMP - Threads = 7</td>
<td>810.7</td>
<td>0.9</td>
<td>99.7</td>
<td>15.3</td>
</tr>
<tr>
<td>C++ CPU - Threads = 14</td>
<td>497.1</td>
<td>0.5</td>
<td>60.4</td>
<td>24.9</td>
</tr>
<tr>
<td>OpenMP - Threads = 14</td>
<td>489.5</td>
<td>0.5</td>
<td>60.2</td>
<td>25.4</td>
</tr>
<tr>
<td>C++ CPU - Threads = 27</td>
<td>313.7</td>
<td>0.3</td>
<td>39.2</td>
<td>39.5</td>
</tr>
<tr>
<td>OpenMP - Threads = 27</td>
<td>319.2</td>
<td>0.3</td>
<td>39.3</td>
<td>38.8</td>
</tr>
<tr>
<td>Xavier CPU - Threads = 8</td>
<td>446.4</td>
<td>0.5</td>
<td>54.9</td>
<td>27.7</td>
</tr>
<tr>
<td>Xavier GPU</td>
<td>1167.0</td>
<td>1.3</td>
<td>143.5</td>
<td>10.6</td>
</tr>
<tr>
<td>GTX-1080 GPU</td>
<td>854.4</td>
<td>0.9</td>
<td>103.6</td>
<td>14.5</td>
</tr>
<tr>
<td>V-100 GPU</td>
<td>152.5</td>
<td>0.2</td>
<td>18.7</td>
<td>81.1</td>
</tr>
</tbody>
</table>

Table 1. This compares the performance at a high level, then at increasingly granular levels. Time-Per-Block for multi-threaded CPU implementation is the observed time calculated from the total time and number of blocks. *Speed-up is shown with respect to total time, but all columns show about the same results. The blue color uses OpenCV, the green color shows results for NPP to do GPU computation. Performance statistics reflect a real application workload with variable block sizes (majority of search blocks are 142x142), template block sizes (majority of search blocks are 40x40), an average of 120 blocks per frame, and 8132 frames.

\[
\text{edge distance}_{ST} \geq \frac{SW_d}{2} + temp_d
\]  

(3)

For a comparison of the OpenCV and MATLAB results to the significantly non-matching results from NPP, it is best seen in Figure 8. Here it is clear that for these example images the correlation responses are quite different. NPP does not find values that are nearly as high except for where the peak is, which could actually be considered a more ideal (less noisy) NCC map. Still, because our implementation for mosaicing is based on the accuracy of MATLAB, we evaluate where the differences in the three algorithms comes from.

We have thoroughly checked the correlation matrix for all of our implementations and libraries used by NCC methods. And while OpenCV is open-source, and their implementation details can be seen directly from their code base, both MATLAB and NPP’s NCC implementation details cannot be checked due to their closed-source nature. We have summarized that the difference in NCC correlation matrix and the peaks might be caused for multiple reasons.

1. For OpenCV’s Cross-Correlation operation, using it’s `matchTemplate` function, had 6 different methods for calculating the correlation matrix. We have selected the method ‘CV_TM_CCOEFF_NORMED’, which is one of the normalized methods available, and the equation for this method is given in the documentation. Internally, this method uses an Integral-Image and Fourier-Transformation for optimization. But for MATLAB’s and NPP’s NCC operation, using it’s `normxcorr2` and ‘nppiCrossCorrValid_NormLevel_32f_C1R’, respectively, they don’t give any implementation details, and work as a black-box. So we could not be sure of the exact method, implementation and optimization techniques being used in it’s case.

2. The peak finding algorithm used for MATLAB after calculating the correlation matrix was selecting the peak in a column-wise order, where in OpenCV/C++ it was finding the peak in the correlation matrix in a row-wise order. NPP chooses the tied max at the top left of the image. This only creates a mismatch if there are multiple peaks in the correlation matrix with the same maximum value.

3. Checking the correlation matrix from both OpenCV/C++ and MATLAB’s NCC computation, it was evident that the size of the correlation matrix for MATLAB is larger in both dimension than the correlation matrix for OpenCV. We found out that this change initially occurs in the Search-Window, where the `normxcorr2`
Figure 6. This is the timing comparison between all of our methods. The first column of plots are for the factor Total-Time (for all frames), second column of plots are for Time-Per-Block, and the third ones are for Time-Per-Frame. All of the plots in the first row are in Log scale for the Y (time) axis for including the MATLAB timing while keeping all of the other method timings distinguishable. The second row of plots excludes the MATLAB results, and are in Linear scale. In each of the plot, GPU results are shown under a single X-axis value, as well as the MATLAB result for the first row of plots, and the rest of the X-axis values, from 1 to 27, denote the number of threads used for the CPU implementation.

The correlation-matrix function of MATLAB automatically applies a padding so that it can also calculate the correlation in the border regions of the search-window. And this padding in the search-window causes the computed correlation matrix to be larger in size as well. The `matchTemplate` function of OpenCV doesn’t apply any kind of padding while performing the NCC calculation. The dimensions for correlation-matrix can be computed from the search-window and template sizes, where $CM_d$ is the Correlation-Matrix size, $SW_d$ is the Search-Window size, and $Temp_d$ is the template block size (assuming square blocks),

$$CM_d = (SW_d - Temp_d) + 1$$

This Correlation-Matrix size computation is same for both MATLAB and OpenCV. But the Search-Window (SW) size for MATLAB differs from OpenCV as the additional padding in applied on it. For MATLAB, the Search-Window size increases by following this equation, where $SWP_d$ is the Search-Window size with the added padding,

$$SWP_d = SW_d + (2 \times (Temp_d - 1))$$

It can be seen from Figure 8 that the correlation-matrix for MATLAB is larger than the other two methods and contains border effects, which is the reason for the zero-padding added to the Search-Window. There are different variations of the NPP function we used that use "Full" mode, which allow us control over when 0 padding is used - thus it is not an issue for NPP.

4. As for the difference between OpenCV/MATLAB and the NPP API used, we have found a few major contributing factors. First, precision and some normalization seem to be responsible for reducing the size and
Figure 7. Error histogram for comparing (X-Peak, Y-Peak)-location of matched point-in MATLAB and our Multi-
Threaded NCC implementation using OpenCV. X-axis represents pixel error between same block peak in MATLAB and
OpenCV where y-axis stands for number of blocks with that error. From this graph, we can see that around 94.9% of
blocks have an exact match, i.e., zero error between them. As the error increases, number of blocks drops significantly.
For example, only 3.5%, 0.7%, 0.1% and 0.2% of blocks have pixel error of 1, 2, 3 and 4 respectively. Only 0.63% of blocks
have error greater than 4 pixels.

number of peaks, this is evident from the correlation matrices and corresponding 3D response maps shown in
Figure 8. Because the implementation is hidden from us, we cannot be sure, but it seems to be a regional
normalization. We found this to be proven by NCC performed with templates taken from the image itself. The
peaks were never exactly one, likely because they were averaged with the scores around them.

5. The more significant difference is a lack of instability handling in NPP. Because the NCC algorithm
divides by the variance, there can be issues when a homogeneous region is taken as the source or template, this
is illustrated in Figure 9 and Figure 10. Divide by zero or divide by very small amounts causes overflow or
other related problems in the output. This can be handled by discarding bad values or by regularizing, adding a
constant to the numerator and denominator. OpenCV and MATLAB implementations handle these cases, while
NPP does not. This makes it unreliable, as in natural imagery we will use many homogeneous regions and the
issue cannot be ignored. Luckily, the OpenCV GPU matchTemplate algorithm handles these cases, so we still
have a GPU option. However, this API has poor performance shown in experiments for Section 5.5.

5.4 Analysis

From the speedup results, we can conclude that parallelization results in a significant speed improvement for
NCC. Parallelizing across all the template matching needed for a pair of frames, as in the CPU multi-threaded
implementation, brought us up to a 40x improvement, although this requires a large number of threads. The
performance increase does not scale linearly with threads, as it would seem some overhead prevents this.

Parallelizing the NCC operation itself, as in the GPU implementation, also saw great improvements. Per
NCC operation, we saw a 10-80x improvement depending on the hardware. It is much harder to draw any
conclusions about what this improvement scales with, as we did not have enough GPUs of similar hardware to
test with.

We also saw good performance on the Jetson Xavier AGX, where there is potential to pre-calculate NCC
scores for mosaicing on the edge.
Figure 8. This figure illustrates the correlation-matrix and peak comparison between all of our methods. The first row contains the source image and the template, which is to be searched within the source image. The next rows show the correlation matrix and template matched to the source image for each method. The MATLAB result in Row 2 includes the padding which is the default behavior.
Figure 9. For demonstrating the differences in the results for NPP from both MATLAB and Multi-Threaded CPU implementation using OpenCV, we have generated a random image as a source image of dimension (300x240), within which we will search our template-patch, and selected a patch from it as the template of dimension (22x23). For computing the difference map, we have taken the correlation matrix from both OpenCV and NPP, and for the first image we can see that most of the differences are in the zero padded region, which is homogeneous, and in the rest of the image where variance is high, the difference is very small. This is also apparent from the second difference map, as there is no homogeneous region due to no padding, all of the differences are in the $10^{-3}$ region.
Figure 10. We have also generated a random image with a flat patch (homogeneous) inside it as a source image of dimension (300x240), within which we will search our template-patch, and selected a patch from it as the template of dimension (22x23). Similar to the previous figure 9, for the difference map, we have taken the correlation matrix from both OpenCV and NPP, and for the first image we can see the differences are in the zero padded region as well as in the patch with flat region, which is homogeneous, and in the rest of the image where variance is high, the difference is very small. This is also apparent from the second difference map, as the only homogeneous region there is the flat patch and the difference is high in that region, and for all other region, the differences are in the $10^{-3}$ region.
<table>
<thead>
<tr>
<th>Method</th>
<th>Measure</th>
<th>Template Size</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Time-Per-Block (ms)</td>
<td>(16 x 16)</td>
</tr>
<tr>
<td>MATLAB</td>
<td>Speedup</td>
<td>1</td>
</tr>
<tr>
<td>C++/OpenCV Thread-1</td>
<td>Time-Per-Block (ms)</td>
<td>229.9</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.5</td>
</tr>
<tr>
<td>C++/OpenCV Threads-7</td>
<td>Time-Per-Block (ms)</td>
<td>41.6</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>8.3</td>
</tr>
<tr>
<td>C++/OpenCV Threads-14</td>
<td>Time-Per-Block (ms)</td>
<td>22.3</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>15.5</td>
</tr>
<tr>
<td>C++/OpenCV Threads-27</td>
<td>Time-Per-Block (ms)</td>
<td>12.8</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>26.9</td>
</tr>
<tr>
<td>C++/OpenCV Xavier CPU 8 Thread</td>
<td>Time-Per-Block (ms)</td>
<td>128.1</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>2.6</td>
</tr>
<tr>
<td>Xavier GPU NPP</td>
<td>Time-Per-Block (ms)</td>
<td>23.8</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>14.5</td>
</tr>
<tr>
<td>GTX-1080 GPU NPP</td>
<td>Time-Per-Block (ms)</td>
<td>24.3</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>14.2</td>
</tr>
<tr>
<td>V-100 GPU NPP</td>
<td>Time-Per-Block (ms)</td>
<td>2.1</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>164.2</td>
</tr>
<tr>
<td>Xavier GPU OpenCV CUDA NCC</td>
<td>Time-Per-Block (ms)</td>
<td>205.3</td>
</tr>
<tr>
<td></td>
<td>Speedup</td>
<td>1.7</td>
</tr>
</tbody>
</table>

Table 2. Results for our randomly generated 2048x2048 image for different sized template blocks. Workload is based on 100 random template blocks of five sizes, with the whole image as the search window, averaged over 400 million NCC computations.

Furthermore, the data in Figure 7 shows that our NCC implementations are not pixel perfect identical, and there are many possible reasons for that still being investigated, as evident from the above discussion.

5.5 Further Results

Due to our findings showing that the methods only have the same output with a completely random image, we also compiled speedup results with this style of input. Since we were generating random input, we were now able to characterize the timing performance with respect to the size of template blocks. Table 2 gives the results and new speedup for five different selected template sizes, 16x16, 32x32, 64x64, 128x128 and 256x256, respectively.

First, we show that the speedups reported previously are still accurate with exactly matching output. While these speedups shown for the multi-threaded CPU C++ implementation in both tables are slightly slower, this can be attributed to the significantly larger source image (2048x2048), and the resulting much higher number of computations required.

The GPU implementation in Table 2 shows a higher performance than with the VIRAT data from Table 1, but it slows down significantly with larger template sizes like 256x256, shown with Table 2. We have a few hypotheses for why, but we cannot be sure due to the closed off nature of the NPP API’s. One possible reason is an increased number of bank conflicts as the amount of data grows unwieldy. This is especially likely, as the search window size in this case is very large at 2048x2048. With the VIRAT data, our search window sizes were 10-16x smaller. We found that GPU utilization rises to 100% at template size 64, showing that their algorithm correctly scales as much as possible with number of threads while it must be dealing with the size of the data, causing the slowdown. Also possible is an algorithm difference where NPP does not implement the integral image or the Fourier transform denominator.

This shows that it may be best to scale NCC by allowing each thread to handle an entire NCC calculation, as was done with our CPU multi-threading. It seems to scale better and with more available hardware.

We also had the opportunity to test the CUDA enabled OpenCV matchTemplate function. The correlation matrix output is exactly identical to the CPU version, but it parallelizes the NCC computation like NPP does.
Figure 11. Speed-Up shown for two different workloads, 1 and 2, for our different methods with respect to the original MATLAB implementation. Five instances of CPU thread configurations are shown here marked as blue bars, and the green bars denote the speed-up for the three different GPUs.

Table 2 shows that it is not similar in implementation to NPP at all, however. It shows that the speedup actually increases with template size, the opposite of NPP. It is also significantly slower than most threaded CPU implementations. Thus, it has some merit as a slower, more scalable and numerically stable version of NCC for the GPU.

6. CONCLUSION

In this paper we demonstrate a significant and straightforward way to vastly speed up the computation of Normalized Cross Correlation. An important matching technique that is part of many pipelines, including that of our own image mosaicing, can no longer be such a significant bottleneck. Furthermore, the difference between the NCC calculation of various sources (MATLAB and OpenCV) is not significant.

It is, however, still going to be one of the slower aspects of the system. The fastest GPU implementation, with expensive hardware, runs for two and a half minutes, the highest level of multi-threading version taking four, for our example sequence of around 8000 frames. We also demonstrated how the difference between different NCC API’s can give different result due to implementation differences, causing issues with practical use. For these two reasons, we are exploring future work in a few key areas: we believe it possible to construct a hybrid system, that uses a portion of the CPU allotment to launch the GPU kernels for NCC that can be individually faster than the CPU computation being run in parallel. This will require an in-depth theoretical and experimental analysis, but should be able to combine the speed improvement of both systems presented here. We also hope to write our own implementations of the normalized cross correlation operation so we can experiment with other fine-grained optimizations for our mosaic generation. In the GPU especially, we believe that writing our own kernel will give us more flexibility to experiment.

ACKNOWLEDGMENTS

This work was partially supported by awards from U.S. Army Research Laboratory W911NF-1820285 and Army Research Office DURIP W911NF-1910181. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the U.S. Government or agency thereof.
REFERENCES


