Adaptable and Efficient Variable Size Template Matching in CUDA

Nicholas Moore and Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University
Boston, MA
{nmoore,mel}@coe.neu.edu

Laurie Smith King
Department of Mathematics and Computer Science
College of the Holy Cross
Worcester, MA
lking@holycross.edu

Introduction

Increasingly flexible GPUs and the advent of GPGPU (General Purpose GPU) languages, such as Nvidia's CUDA and the OpenCL standard, offer potential peak performance that far exceeds that of general purpose CPUs for a variety of problems. However, architectural and programming restrictions often prevent programmers from achieving peak performance. Even for problems that map well to current GPGPU environments, it can be difficult to develop general solutions that work well over a range of problem sizes. Current GPGPU platforms offer best performance at specific execution configurations, usually involving multiples of architecture and application specific values, with performance dropping off rapidly for programs not conforming to the ideal. Developing algorithm implementations and techniques that are widely parameterizable remains a roadblock to broader GPGPU adoption and the development of adaptable GPGPU libraries. It is not feasible for library developers to create collections of GPU kernels specific to particular problem instances that cover all of the cases users may require for a single algorithm.

As a case study, we are developing a parameterized CUDA implementation of a sliding-window two-dimensional correlation-based template matching algorithm. Our GPGPU implementation's ability to handle large template sizes and to adapt to variable input sizes is distinctive and is important for building adaptable GPGPU libraries.

Target Template Matching Algorithm

The template matching algorithm is based on Pearson's correlation, as discussed in previous work [1]. The particular application uses multiple templates as well as a sliding window, which increases the tracking ability, but significantly increases the computation: a correlation must be computed for each template for each possible location of the template within the region of interest (ROI). Pearson's correlation, represented by *corr2()* as it is in the MATLAB Image Processsing Toolbox, is defined in Figure 1.

$$corr2(A,B) = \frac{\sum_M \sum_N \left(A_{MN} - \bar{A}\right)\left(B_{MN} - \bar{B}\right)}{\sqrt{\left(\sum_M \sum_N \left(A_{MN} - \bar{A}\right)^2\right)\left(\sum_M \sum_N \left(B_{MN} - \bar{B}\right)^2\right)}}$$

Figure 1: $\bar{A}$ ($\bar{B}$) is the matrix average of A (B)

Our data sets are from a real-world medical imaging application and include widely varying template sizes, ranging from 23-by-21 to 156-by-116 pixels. The number of templates varies between 10 and 14. The area by which each template is allowed to move varies from 2 to 9 pixels horizontally and 9 to 18 pixels vertically.

This forms a good case study since the data sets are created by humans and not carefully selected to match GPU architecture sizes. Both the template size and shift area variation make it difficult to assume general relationships between the parameters, as is the case for building a general purpose library of GPU kernels.

Optimized use of shared memory in GPU kernel implementations is extremely important and has been well documented. However, the data sizes in our application make it more difficult to leverage these memories. Most template-based GPU kernels assume the image is too large for shared memory, but assume a small and square template size that easily fits into shared or constant memory. The size of the templates in this case study prevents the storage of one complete single precision floating-point template in constant or shared memory. Likewise, the typical approach of tiling the ROI and loading a subset of the image data into shared memory will not work as the template sizes are too large for storage in shared memory, let alone the extra space for a corresponding ROI.

Numerator Kernel Implementation

The numerator of the *corr2()* function, which has been implemented as a distinct kernel, is the focus of this abstract. It is similar to non-separable convolution except for the subtraction of the frame data average from each value. This average value, $\bar{B}$, is dependent upon the current template location, as are the values of B_{MN} for any specific M and N pair. This complicates the numerator calculation and prevents significant reuse of the frame data contribution.

To address the data working set size problems discussed above, we take advantage of the fact that the computation is formed from two nested summations. The template is broken down into tiled subregions, with each subregion's contribution to the final summation computed independently. A second step is required to add each part of the calculation to the final value. The tiles are mapped to thread blocks within a kernel launch. For the implementation to adapt to arbitrary template sizes it must be able to handle template sizes that are not multiples of any efficient tile size. This scenario, shown in Figure 2, results in leftover template pixels not covered by the regular set of template tiles. Padding the template is not possible as it affects the template average as well as the size of the underlying frame data and the averages used to compute the similarity score for a given template position.

By combining runtime compilation of kernels and separate kernel launches we are able to introduce adaptability and handle the edge cases while preserving the benefits of compile-time optimization, such as loop unrolling and

strength reduction, which are important for performance on GPUs. A main tile size is selected and mapped across the template size associated with the current problem. Then, as needed, separate kernels are compiled to handle the remaining tiles.

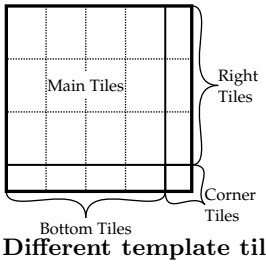

Figure 2: Different template tile regions.

Each tile writes output to a contiguous pitch-aligned location, allowing for fully coalesced memory accesses in the summation kernel.

Performance

To evaluate the performance of the numerator implementation, it was compared to an older untiled GPU implementation, presented previously [1], and a MATLAB numerator implementation. All of the experiments were conducted on a workstation with an Intel Core 2 Duo E8400 (3.00 GHz, 6MB L2 cache) and an Nvidia GeForce 8800 GTX running Ubuntu 9.04 64-bit, CUDA 2.3, and MATLAB R2010a. The untiled GPU version of the numerator does not use shared memory and relies on texture memory to cache the template data. For the tiled GPU kernel, arbitrary tile sizes of 16-by-4 (rows-by-columns), 8-by-8, and 4-by-4 were selected.

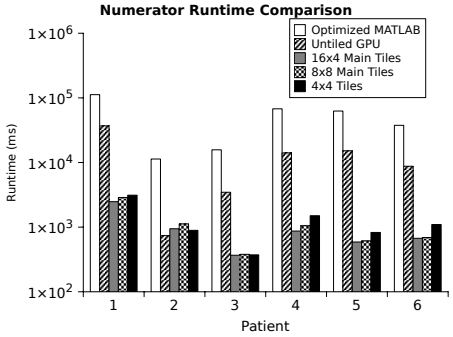

Figure 3: Comparison of numerator runtimes.

The execution times of the numerator implementations are shown in Figure 3. These results do not include data transfer to the GPU, as the numerator is one step in the process of calculating Pearson's correlation. The tiled template implementation provides significant performance improvements over the untiled GPU implementation, especially for larger template sizes. This can be seen in Table 1. However, the tiled template actually results in slow down for the second patient. This is due to the use of the smallest template size among the data sets. The small template size does not generate enough tiles, and therefore thread blocks, to keep all of the GPU's streaming multiprocessors occupied.

The untiled GPU implementation uses a single launch to process a single template applied to all of the frames at once. This works well for the large static data sets provided for testing, but is not a workable solution for processing a stream of images. The tiled GPU kernel processes one frame at a time and each template is independently processed, allowing for streaming an indeterminate number of frames. For the tiled version, the smaller tile size of 4-by-4 generates more blocks, but places more pressure on the memory hierarchy. This provides some benefit for the smallest template size associated with the second patient, but not for the other patients.

	Main Tile Size		
Patient	16×4	8×8	4×4
1	14.89	12.82	11.81
2	0.78	0.66	0.83
3	9.43	9.13	9.30
4	16.23	13.42	9.40
5	25.73	24.80	18.30
6	13.03	12.67	7.98

Table 1: Speedups of the tiled numerator kernel over the untiled kernel for various main tile sizes.

Figure 4, displays the runtime breakdown for each region of the template tiles and the final summation. Note that, when present, the edge tile kernels often take roughly the same time to execute as each other. This is a result of each kernel launch containing fewer blocks than there are streaming multiprocessors on the GPU. The block execution latency determines the kernel launch execution time.

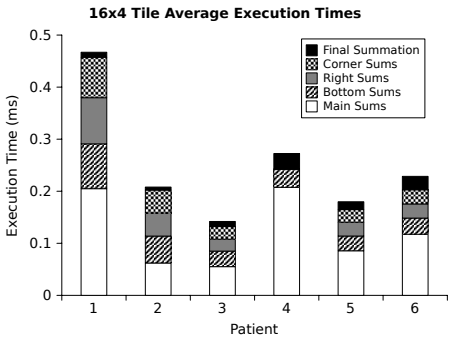

Figure 4: Plot of the contribution of each tile region to the processing of a frame and template pair.

Using separate GPU kernel executions is somewhat inefficient on current GPUs that only execute one kernel at a time, but may work well on upcoming devices that can execute multiple kernels simultaneously, like Nvidia's Fermi GPUs. Breaking the calculation into more kernel calls generates more total blocks and will allow better scheduling of these blocks on GPU platforms supporting concurrent execution.

Acknowledgments

The authors would like to thank The MathWorks for supporting this research.

References

[1] Nicholas Moore and Miriam Leeser. Accelerating a MATLAB application with Nvidia GPUs: a case study for GPU library construction. In *High Performance Embedded Computing Workshop 2009*, Lexington, MA, USA, 2009. MIT Lincoln Laboratory.