Adaptable Two-Dimension Sliding Windows on NVIDIA GPUs with Runtime Compilation

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

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

***Abstract*—For some classes of problems, NVIDIA CUDA abstraction and hardware properties combine with problem characteristics to limit the specific problem instances that can be effectively accelerated. As a real-world example, a two-dimensional correlation-based template-matching MATLAB application is considered. While this problem has a well known solution for the common case of linear image filtering—small fixed templates of a known size applied to a much larger image—the application considered here uses large arbitrarily-sized templates, up to 156-by-116 pixels, with small search spaces containing no more than 703 window positions per template. Our CUDA implementation approach employs template tiling and problem-specific kernel compilation to achieve speedups of up to 15 when compared to an optimized multi-threaded implementation running on a 3.33 GHz four core Intel Nehalem processor. Tiling the template enables exploiting the parallelism within the computation and shared memory usage. At the same time, problem-specific kernel compilation allows greater levels of adaptability than would otherwise be possible.**

***Keywords*-GPGPU; multicore; image processing; 2D correlation;**

I. Introduction

In the past few years, General Purpose Graphics Processing Unit (GPGPU) computing has been applied to an increasing variety of problems. Many of the most impressive speedups over general purpose processors are still observed in problem domains that have characteristics similar to the historical role of the GPU - image and video processing.

However, even in problem areas where GPGPU exhibits better performance, existing implementations may only be applicable to a limited range of problem instances. Restrictions imposed by both the CUDA architecture abstraction and the hardware implementation complicate the development of highly adaptable code that can solve a general version of the problem at hand.

An example of a problem of this type is given by a real-world application: lung tumor tracking. This application relies on non-separable, time-domain, correlation-based template matching for tracking. Despite using a type of computation, correlation, that has been shown to perform well on GPUs, the particular range of problem parameters specified by this application rules out common implementations. Existing implementations assume that the template fits completely into shared memory and that the small templates are applied to many locations in the image, thus exhibiting a natural thread level parallelism. For larger templates, such as those in the lung tumor tracking application, the limited size of fast on-chip memories mean that standard approaches cannot be applied. The implementation approach discussed here tiles the template to reduce the working set size as well as to generate more parallelism. This alleviates a major restriction from the general correlation problem, limitations on template sizes, while maintaining good performance.

In general, adaptability limitations produced by common GPGPU coding practices, which stem from accommodating GPU hardware characteristics necessary for achieving maximum performance, prevent implementations from adapting to arbitrary problem parameters. However, users may encounter real-world problems that do not conform to these limitations, including scenarios where different problem sets require different parameters, preventing users from employing GPGPU solutions. This can be particularly problematic in scenarios where parameters are dependent on the problem instance, as is the case with the template matching application considered here; each data set requires widely varying and large arbitrary template sizes.

Many of these restrictions are a result of hardware parameters and characteristics as well as the compile-time nature of many important GPGPU performance optimizations. To help mitigate these fundamental restrictions, our implementation approach relies on problem-specific kernel compilation.

In this paper, Section II introduces some aspects of the CUDA abstraction and realization, and Section III discusses how these characteristics can affect adaptable GPGPU implementations. Section IV introduces the particular template-matching application considered here and how it requires a more flexible implementation approach than has been considered in the past. Section V discuss similar correlation-like GPGPU implementations and their associated adaptability limitations. Our experimental approach and performance results are presented in Section VII, followed by some additional related work in Section VIII. The paper ends with discussion and conclusions in Sections IX and X.

II. Background

The major reason behind the increase in enthusiasm for GPGPU solutions is the possibility of significant performance improvements over CPUs. NVIDIA CUDA is particularly popular and used for this research. CUDA provides a software environment for writing GPGPU kernels and specifies an abstract hardware target with large amounts of available parallelism. The abstractions provided shield the application developer from a number of low-level hardware considerations and allow NVIDIA to change the underlying implementation of the hardware, increasing peak performance over time. However, this paradigm still requires application programmers to contend with a number of issues not often considered when developing software for general purpose processors. Additionally, developers have not been completely isolated from changes in the CUDA hardware realization, which NVIDIA has been updating rapidly over time. New GPUs have changed important hardware parameters and introduced completely new capabilities.

In CUDA, parallelism is presented in the form of thousands of threads. To organize these threads, CUDA provides a two-level thread hierarchy composed of thread blocks and the grid. Thread blocks can have one, two, or three logical dimensions. The grid specifies a one or two dimensional space consisting of identical thread blocks. Together, the grid and blocks can be used to define a large thread space that is defined independently for each kernel invocation but fixed for the duration of a kernel's execution.

Inter-thread communication is possible at block scope via shared memory, which is available for reading and writing by all threads. Block-level thread barriers allow for synchronized shared memory access by all the threads within a block. However since thread blocks are limited to a relatively small number of threads, large scale inter-thread communication is generally not possible. Combined, the thread hierarchy and communication restrictions allow for the scalability of CUDA applications by constraining thread blocks to relatively small and independent chunks of computation. This requires the given problem to be amenable to this implementation approach.

The amount of memory and execution resources used at the block level can affect performance and must be considered by kernel developers. Within CUDA GPUs, streaming multiprocessors (SMs) execute blocks. Full thread blocks are assigned to a SM, and the threads within a block are grouped into sets of 32 consecutive threads, collectively referred to as a warp. Warps are the unit of execution, with the threads within a warp executed in a single-instruction, multiple-thread (SIMT) manner. Fast context switching between warps and the ability to execute multiple thread blocks within the same SM generates a significant amount of thread-level parallelism. In addition, kernel configurations can have a significant impact on the ability of the GPU to execute more warps simultaneously. SMs contain a limited number of shared resources, including registers, shared memory, and warp and block state tracking hardware. The number of blocks simultaneously executed by a SM is limited by the block-level resource usage required by a kernel.

In addition to execution constraints, CUDA requires developers to manually select between a number of memory types, each with different characteristics that can affect performance. Table I summarizes a number of important aspects of each of the memory types. Global memory accesses, for example, can be grouped together into larger batch transactions in a process called coalescing. The details differ between GPUs, but coalescing generally requires threads in a warp to access consecutive or mostly contiguous locations. When the coalescing requirements are not met, the accesses are serialized introducing additional latency. Coalescing of global memory accesses is considered one of the single most important factors for achieving high performance for CUDA kernels [1].

Bank aliasing can also be a factor in some scenarios for global and shared memory. Global and shared memory are organized into a device-dependent and fixed number of banks, respectively. When bank aliasing occurs, the conflicting memory accesses are serialized by the device, thereby reducing performance. Constant memory is effectively organized into a single bank so that all threads within a half-warp must access the same address to avoid serialization. Finally, texture memory, unlike global memory, provides caching optimized for 2D locality and provides some special interpolation and addressing modes that can simplify kernel implementation. These addressing operations can be used both to simplify memory calculation code and to implement some mathematical operations in hardware, effectively reducing their cost to zero.

III. GPGPU Adaptability Issues

The restrictions discussed in Section II have varying levels of impact on different applications, but in many cases result in design trade-offs that limit the range of problem instances to which a kernel can adapt. Techniques such as runtime guards or data padding can often be used to allow a single kernel to handle any instance of a problem. There are classes of problems, however, where it is not obvious how to make implementations adaptable to an arbitrary instance of the problem. Depending on the problem, a number of difficulties may arise.

First, widely adaptable GPGPU kernels cannot make assumptions about parameter values. This limits the implementer and compiler from performing optimizations that are based on assuming certain relationships between values or only certain values of a single parameter. Examples at the compiler level include loop unrolling and strength reduction on expensive modulus or division operations when the divisor is a power of two. When loop counts depend on adjustable parameter values, the compiler cannot perform this important GPU performance optimization.

Second, an arbitrary problem instance may specify a working set that is too large to store in some of the limited memories, particularly shared or constant memory. For a given

Memory Type	Scope	Size Limitations	Read/ Write	Persistent	Cached	*O*(Latency)	Other Considerations
Register	Thread	32 to 128 KB/SM	RW	No		1 cycle	Overflow
Shared	Block	16 or 48 KB/SM	RW	No	No	10 cycles	Bank aliasing
Constant	Global	64 KB	R	Yes	Yes	100 cycles	Single address
Texture	Global	Varies	R	Yes	Yes	100 cycles	Addressing and interpolation modes
Surface	Global	Varies	RW	Yes	Yes	100 cycles	Addressing modes; Newer devices only
Global	Global	64 MB to 6 GB	RW	Yes	No or Yes	100 cycles	Coalescing and bank aliasing

TABLE I
CHARACTERISTICS OF THE MEMORIES EXPOSED BY CUDA.

implementation approach, using these memories can place upper limits on parameter values.

Third, parameter values may affect not only the number of threads in a block, but also how threads are assigned units of work from within the work space defined by the particular parameters. This can complicate efforts to efficiently utilize the execution and memory resources available in a block while accounting for issues like warp boundaries and memory bank conflicts.

Thread work assignments can also be complicated by variations in the structure of the work required at different points during a single kernel execution. Threads will have been allocated to match a particular stage of kernel execution, such as one thread per unit of data that must be loaded, or one thread per one or more output values. CUDA provides a fixed hierarchy with three dimensions at the block level and two grid dimensions. Each block and thread must compute a mapping from the thread's location within the thread space into the problem space. As more problem and implementation parameters are supported, the ratio of compute to non-compute instructions will decrease, since more instructions will be needed for tasks such as calculating a particular thread's work assignment. It is important to try to minimize non-compute instruction overhead, particularly division and modulo operations, as they are expensive on NVIDIA GPUs.

While any one of the above issues can be addressed in isolation, simultaneously accounting for several may result in significant design and code complexity. The interplay of the different dimensions for adaptability can create a large and complex tradeoff space, and the additional complexity associated with adaptability can increase overhead in terms of non-compute code and the use of extra registers, which are often in limited supply on NVIDIA GPUs.

IV. 2D CORRELATION-BASED TUMOR TRACKING

As a case study highlighting the issues associated with large templates and adaptability in general, this paper considers a 2D correlation-based template matching application originally written in MATLAB [2]. The goal of this work is to track lung tumors in fluoroscopic imagery in real time without the use of visual markers surgically implanted in the body. Once the tumor position can be tracked reliably, higher-intensity radiation focused on the tumor may be used, which improves the efficacy of treatment. Currently, lung radiotherapy relies on a wide-area beam irradiating the entire range of tumor movement throughout the respiration cycle. Since this approach also targets healthy tissue, a lower intensity beam must be used.

The application studied for implementation in CUDA generates templates from training data and uses Pearson's correlation to generate similarity scores between templates and the current image data. Computing the similarity score represents the bulk of the work involved in the application and is the only part of the application targeted for GPU acceleration. The template generation algorithm is specific to the application, but the template-specific processing can be done once at application setup time and does not constitute a significant amount of processing. For any given problem instance, template related data are considered static inputs.

Two methods are used to account for the periodic and irregular variation associated with the respiratory cycle: using multiple templates and shifting the templates over a region of interest (ROI). These provide significant accuracy improvements at the cost of significant increases in the total processing required. For each template, a similarity score must be calculated for each possible position of the template within the shift area, and this is repeated for each frame. Scanning the resultant scores for the best match is not considered here. The original reference application uses the *corr2()* function from the MATLAB Image Processing Toolbox to implement Pearson's correlation; the definition is shown in Figure 1.

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

Fig. 1. $\bar{A}$ and $\bar{B}$ are the matrix averages of A and B

Table II shows characteristics of the reference data sets studied [2]. They display significant variation, and none of the values are powers of two. The per frame computational requirements vary widely between patients and are affected by the template size and the number of times *corr2()* is called, which is determined by the shift values. The template size increases the iteration counts associated with the double summations used in both the numerator and the denominator of the *corr2()* function. The total per-frame invocation count of the *corr2()* function depends on both the number of templates and the shift values.

In this application, the static nature of the templates allows

Patient	Template Size	Shift V/H	*corr2()* Calls Per Frame
1	53 × 54	±18/±9	8436
2	23 × 21	±11/±5	3289
3	76 × 45	±9/±4	1710
4	156 × 116	±9/±3	1463
5	86 × 78	±11/±6	3588
6	141 × 107	±9/±2	1330

TABLE II
PER PATIENT, TEMPLATE SIZE, VERTICAL/HORIZONTAL SHIFT WITHIN ROI, AND NUMBER OF *corr2()* CALLS.

us to eliminate significant redundant computation. Calling the *corr2()* function multiple times with the same template results in recalculating the template matrix averages for use in the denominator and numerator. Similarly, the total template contribution to the denominator is, for a given patient run, constant across correlations and can be computed once and reused. However, the corresponding frame data values must be computed for each position within the ROI and for each incoming frame. As the template-sized sliding window traverses the ROI, the underlying frame data changes, changing the frame's contribution to both the average of the frame data and the frame's contribution to the denominator. All implementations of the sliding window *corr2()* take advantage of this redundancy to improve performance. Assuming that the template data is represented by A and the frame data is represented by B, the actual implemented functionality is reduced to the equation shown in Figure 2.

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

Fig. 2. $\bar{B}$ is the matrix average of B, A^C is the current template with the template average subtracted from each value, and A^D is the template contribution to the denominator.

A. Challenges

For the CUDA implementation, this template matching application is an interesting case study because the data sets are created by humans, vary from one patient to the next, and cannot easily be padded to convenient sizes. Both the template size and shift area vary significantly and make it difficult to assume general relationships between the parameters.

For each patient a single ROI is defined, but it is of relatively small size (between 95 and 703 total template positions) compared to the entire frame that is typical in linear image filtering. This introduces concerns about generating enough parallelism, especially when each frame is processed individually in a streaming manner.

The individual computation for each window position is also relatively complex, even after the simplifications made by eliminating redundant computation. The subtraction of the average of the data under the current window position makes an effective frequency domain implementation difficult, and it cannot be assumed that the template values represent separable filters.

Finally, the data sizes in the tumor tracking application make it more difficult to leverage constant and shared 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. As seen in Table II, the template sizes under consideration are as large as 156 by 116 pixels, which prevents the storage of a single 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 any additional space for a corresponding ROI.

These differences put this tracking application in a different part of the sliding window parameter space than is usually considered for GPGPU acceleration and outside the range of problems existing implementations can accommodate.

V. RELATED WORK: GPGPU CORRELATION

As discussed in Section III, the complexities of developing GPGPU solutions has resulted in current programming practices that often restrict implementations to a specific problem instance or a small range of similar problem instances. This is usually done to take advantage of a GPU's architectural characteristics. While this practice is generally sufficient when building a single application, it prevents significant code reuse and is a problem for library developers. Implementing and maintaining a large number of problem-specific GPU kernels is, in general, not a feasible approach.

A significant majority of the execution time of the lung tumor tracking application introduced above is spent computing a sliding-window Pearson's correlation. It should be noted that this operation, the *corr2()* operation, is similar to non-separable two-dimensional convolution. Given its importance in image processing, two dimensional convolution on GPUs has been widely studied. However, current implementations restrict the range of problems that can be addressed. For the application studied here, the template sizes are large and data set dependent, varying significantly from one patient to the next. At the same time, the area covered by the sliding-window operation is small. This combination of problem characteristics puts it outside the range of many existing GPGPU convolution implementations.

For example, Kong et al. [3] have created OpenCL and CUDA implementations for a number of the functions in the MATLAB Image Processing Toolbox, including the two-dimensional convolution (*conv2()*) function. For AMD GPUs, the authors created a vectorized implementation where each thread processes multiple rows. The AMD GPU implementation did not use the shared memory equivalent (local data store) and relied on the larger register count relative to NVIDIA GPUs to store active data. Each thread loaded 7-by-7 pixel template values into registers and implemented data reuse between vertically adjacent filter positions by only loading

the next row into registers. The threads in a warp move down columns progressing in lockstep. The kernels read data through texture caches so that horizontally adjacent threads will read overlapping data from the same row, reducing global memory pressure.

Two CUDA versions were also tested by the group. The first implementation is similar in design to the AMD kernel approach but used shared memory to store the template due to register pressure. Neighborhood data was read through texture memory. The second approach appears to have used shared memory for all of the data reuse instead of relying on texture memory, and the authors report lower performance in this case.

Regardless of the implementation kernel version, a fixed template size of 7-by-7 pixels was reported, and these are much smaller than the templates considered here. The approach will not scale to larger kernel sizes, as the entire template must fit into shared memory on NVIDIA GPUs and registers on AMD devices.

Another *conv2()* implementation is contained in the AccerEyes Jacket product, which adds support for executing a subset of MATLAB functionality on CUDA-enabled GPUs. The last version of Jacket that documented *conv2()* functionality limitations listed support for arbitrary kernels up to 5-by-5 pixels and square kernels up to 10-by-10 pixels in size [4]. A *filter2()*, which performs the same function as *conv2()* but rotates the input template by 180 degrees, is listed as supporting fixed 3-by-3 pixel templates [5]. A *corr2()* implementation has also been added, but limitations are not documented. It is not clear how well loop parallelization techniques supported by Jacket would be able to take advantage of the computation reduction opportunities discussed in Section IV.

Park et al. [6] analyze a variety of image processing algorithms for implementation in CUDA. The authors focus on four areas of image processing (3D imaging, feature-extraction, image compression, and computational photography) and develop a set of six algorithmic characteristics that can be used to predict the level of parallelization and performance that can be achieved with CUDA. The derived characteristics include branching diversity, computation to memory access ratios and the inherent level of serialization. While a focus on per-pixel computation to memory access ratios is novel, the characteristics do not seem to be confined to image processing algorithms and are considered important for all GPGPU applications and parallel computing in general. Several sliding window problems are considered, including Gaussian smoothing, Canny edge detection, and separable convolution. Few details are given about the actual implementations, but in keeping with usual image processing problems, the template sizes appear to be relatively small throughout and shared memory is used to cache working subsets of global memory.

Also available for NVIDIA GPUs is the NVIDIA Performance Primitives (NPP) library [7] which focuses on image and video processing. The library provides general 2D convolution functionality, but documentation does not indicate the implementation approach or size of templates that can be handled efficiently. Eight bit, unsigned integer is the only type of input data supported. The OpenCV library [8] also provides GPU-based 2D filtering. Separable filters are limited to 16 pixels and non-separable filters are stored in constant memory. OpenCV also explicitly instantiates multiple versions of kernels for each supported template size. This is a common CUDA library technique that allows for unrolled loops but is inflexible and increases binary code size.

These existing implementations are not suitable for the lung-tumor tracking application, as a significant factor is reliance on fast memory types with limited sizes. The template sizes considered here exceed 100×100 pixels. For most existing CUDA-capable GPUs, the size of shared memory per block is 16 KB. Not counting the need to store the underlying image data, store kernel arguments (also placed in shared memory), or occupancy issues resulting from a single block using all of the shared memory available in a streaming multiprocessor, this limits single-precision floating-point templates to 64×64 pixels, assuming square templates.

Constant memory, which is limited to 64 KB for all devices, allows for a larger maximum template size of 128×128 pixels for single-precision floating-point data. However, this is still not adequate for the largest template size in the sample data sets. Additionally, the tumor tracking application uses multiple templates, multiplying the size needed to store the data for one patient.

As mentioned, the search area covered by each template in the tumor tracking application is small. Existing implementations assume a large search area and use this as the main source of parallelism. Here, the search area can be as small as 95 window positions.

A. Ad Hoc Solutions

Beyond convolution-like processing, the need for wider adaptability in GPGPU kernels has been documented in the literature. The most common approach is the manual selection of kernel variants that forgo usage of small memories when problem instances overwhelm the original implementation approach.

Stivala et al. [9] created a CUDA implementation of a bioinformatics protein structures database search problem. Before performing a search, each thread block loads information about the query into shared memory. However, for some searches, the query information is too large for shared memory. To handle this, they compile a second version of the kernel that only uses global memory. While the kernel that only uses global memory is much slower, the number of queries in their data sets that exceed shared memory usage is small so it is still faster to run the inefficient kernel on the GPU instead of transferring the data to the CPU for handling the outliers and then back to the GPU.

Bergen et al. [10] discuss their experiences using OpenCL in an HPC environment creating a compressible gas dynamics simulator that runs on GPUs from multiple vendors. At runtime, one of two kernel implementations is selected based on the target hardware. One variation consists of separate stages

that increases the memory to computation ratio but uses fewer registers. The second variation fuses the two stages but uses more register resources.

Stone et al. [11] have provided a real-world demonstration of the importance of loop unrolling in their molecular orbital visualization software. The group manually generated variants of their kernels for specific problem instances so that loops are unrolled. They report a 40% performance improvement for the problem-specific kernels.

VI. CUDA Implementation: Tracking

To address the needs of the lung tumor tracking application, we have created a unique CUDA-based *corr2()* implementation. The GPU version consists of several CUDA kernels over four stages: (1) calculation of the frame data averages, (2) frame data contribution to the denominator, (3) calculation of the numerator, and (4) final result. With the exception of the final result kernel, each stage consists of two kernels that have nearly identical structure: a tiled kernel that computes partial results and a reduction kernel that combines the partial results into a final result for that stage. Since these first three stages are similar in design, only the numerator stage is presented in detail.

A. Numerator Stage

The numerator of Figure 2 is similar to non-separable convolution except for the subtraction of the frame data average from each value. This averaging makes padding of frame sizes and frequency domain computation inefficient. To address the data set working size and parallelism generation problems discussed in Section IV-A, we take advantage of the fact that the computation is formed from two nested summations, which are associative and can be computed in parallel. The template is broken down into subregions, or tiles, as shown in Figure 3, with each tile's contribution to the final summation computed independently. By selecting tile sizes that are amenable to the size restrictions of shared memory, it becomes possible to take advantage of one of the keys to improving CUDA kernel performance. A reduction sum is then required to combine the contribution of each tile into a final value.

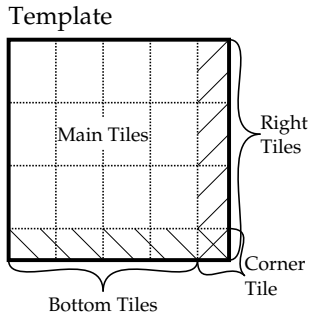

Fig. 3. Different template tile regions

The individual tiles are treated as independent sliding window operations with independent templates, with each tile assigned to a thread block. This allows for the processing of arbitrary template sizes by growing the grid dimensions. This scales up to 65,536 tiles in each direction per kernel launch; additional kernel launches can handle any remaining tiles. The remaining tiles are for edge cases that are needed since padding cannot be done efficiently due to the averaging operation.

While the thread blocks of a grid are used to handle the different template tiles, the various positions of the template within the frame's region of interest are mapped to threads within each thread block. This works well for most of the test cases in our data sets, where the total number of window positions per template per frame are 95, 133, 171, 253, 299, and 703. However, the general sliding-window problem, including the patient with the largest search domain, requires searches with more than 512 locations – the maximum number of threads allowed per block. The maximum number of threads per kernel launch is an implementation parameter, and the current implementation uses additional kernel launches with shift offsets to handle large shift areas.

Each block of the first kernel of the numerator stage writes the output for each tile over the tile's search space contiguously in memory. As a result, the data layout for each tile is consistent across the set of tiles regardless of shape, which simplifies the inclusion of the edge case tile contributions. As a result, the second kernel of the numerator stage, which performs a reduction sum over the contributions of each tile, is relatively simple, with consecutive threads accumulating the contribution of consecutive shift offsets. The data accesses are fully coalesced.

B. On-Demand Kernel Compilation

The implementation needs to be able to adapt to the unique and arbitrary template size associated with each patient data set, including sizes that are not integer multiples of any tile size, let alone a tile size that executes efficiently on the GPU. Data padding is made difficult by the definition of the *corr2()* function, as the average of the data under the window is different at each window position. This scenario, as shown in Figure 3, results in leftover template pixels not covered by the regular set of template tiles. The main tile size chosen will affect the number and shape of any edge tiles that are present. As an example, Table III shows the dimensions and number of each type of tile for the 156×116 pixel template size of Patient 4 for three different main tile sizes. The included tile sizes are examples of how tile sizes affect the total number of tiles and presence of irregular edge case tiles. Although $4times4$ pixel tiles eliminate edge cases for Patient 4, small tile size do not execute efficiently on NVIDIA GPUs. This is discussed in Section VII.

Blocks that are assigned tiles around the right and bottom edges of the templates may be performing the nested summation over a differently shaped area. As a result, the nested *for* loops over these areas may have different loop bounds that have to be evaluated at runtime, which will prevent loop unrolling optimizations.

Main Tiles		Right Tiles		Bottom Tiles		Corner Tile	
Size	Count	Size	Count	Size	Count	Size	Count
$\mathbf{8 \times 8}$	19×14	8×4	19×1	4×8	1×14	4×4	1×1
$\mathbf{16 \times 10}$	9×11	16×6	9×1	12×16	1×11	12×6	1×1
$\mathbf{4 \times 4}$	39×29	–	0	–	0	–	0

TABLE III
TEMPLATE TILING EXAMPLES FOR THE TEMPLATE SIZE ASSOCIATED WITH PATIENT 4 (156×116 PIXELS).

However, through runtime compilation of kernels customized for the current problem, we are able to introduce adaptability and handle the edge cases while preserving the benefits of compile-time optimization. The kernel code that processes a tile is implemented as a `__device__` template function with the tile dimensions as template parameters. Up to four separate explicit specializations of the processing code are instantiated within a wrapper `__global__` function, one for each needed tile size, and the appropriate version is selected by each thread block. By compiling kernel instances on demand once the loop counts are known and fixed at compile time, the compiler can unroll loops for all four tile regions. Compiled-in constants are also used to calculate a number of offsets and strides, eliminating repeated runtime evaluation of some non-compute code by each thread or block.

This second reduction kernel is also compiled on demand. Each thread will loop over the tiles accumulating each partial result for a given shift offset. The total number of tiles, including edge cases, is provided as a compile time parameter so that this loop may be unrolled.

The CUDA API allows compilation of PTX through the driver-level API, but API-based compilation of C for CUDA source is not supported. To continue to enable development of GPU kernels at the higher abstraction levels of C for CUDA, nvcc (the CUDA tool chain compiler) is called at runtime. Loop sizes and other parameters that can result in compile time optimizations are left undefined in the kernel source in terms of preprocessor macros. The macro values are supplied on the nvcc command line with the standard `-D` compiler flag for use by the preprocessor, which can provide custom values for otherwise hard coded `#define` preprocessor directives. Once defined, these values can be used as template arguments, loop trip counts, and array sizes.

C. Other Stages

The remaining stages share a number of similarities with the numerator stage. While the working set requirements are not as high for the frame data average and denominator calculations as the numerator kernel since no template data is needed, the non-tiled working set for most of the patients would still exceed the available shared memory. As a result, the frame data averages and contribution to the denominator are each implemented as a two kernel solution similar to that of the numerator. The frame denominator and numerator share the same reduction sum kernel, but the frame averages reduction relies on a different kernel that also performs a normalization to produce average values.

The final kernel from the *corr2()* implementation computes the fraction and square root operation from the denominator, a straightforward element-wise operation. As is the case with the numerator stage, the final kernel is implemented assuming that the template data is precomputed and static.

D. Runtime Operation

At runtime, each frame is streamed onto the GPU and processed independently during execution. Currently, only the necessary ROI data from each frame is pushed to the GPU, which limits the size of the data transfers. To reduce complexity, the numerator kernel processes a single template at a time, requiring multiple calls to process each frame. The kernels read the original frame and template data through textures, but reads and writes to data generated on the GPU are coalesced accesses to global memory. In the case of the frame data, new frame data transferred from the host overwrites old frame data, allowing reuse of the same frame data texture binding between frames. The template data is static for the duration of the processing of the current patient and is only pushed once; each template is placed into a separate pitched global memory allocation. The texture reference is rebound between the processing of each template to iterate among the templates.

VII. EXPERIMENTS AND ANALYSIS

The performance of the tiled GPU *corr2()* implementation was evaluated through comparison to a MATLAB version and a pthreads multi-threaded, but not vectorized, C implementation. All three versions are optimized to use the computational reduction discussed in Section IV that results from static templates. The CPU-based results were conducted on an Intel Xeon W3580 (4 Nehalem cores at 3.33 GHz and 6 MB cache), while the GPU results were generated with an NVIDIA GeForce GTX 480. The software environment characteristics include 64-bit Linux with GCC 4.4.3, MATLAB R2010a and CUDA 3.2.

All three implementations were benchmarked on the six data sets in Table II. On the GPU, main tile sizes ranging from 4-by-4 to 16-by-16 pixels were tested. In these experiments, the tile sizes for the edge cases were selected by only adjusting one of the dimensions. For example, on the bottom row the tile height was adjusted but not the tile width. For the three two-kernel stages, the thread count (shift positions handled per kernel launch) was fixed at a maximum of 192 threads. The reduction kernels were fixed at 64 threads per block across all the tests.

For each of the six data sets, the per-frame execution times for the three implementations are shown in Figure 4. The GPU results show only the best performing main tile size for each patient and include all data transfers both to and from the GPU. Table IV shows these results in terms of speedups of the CUDA version over the multi-threaded CPU implementation and includes the main tile size that was observed to result in the highest performance. The tiled CUDA implementation provides good performance, achieving rates well beyond real-time, across the data sets despite the arbitrary problem dimensions and large template sizes.

Based on particulars of the GPU implementation, strength reduction optimization opportunities are dominated by computations related to the first tile-size dimension. As a result, the compiler will produce more optimized code when the first dimension is a power of two, as shown in Table IV.

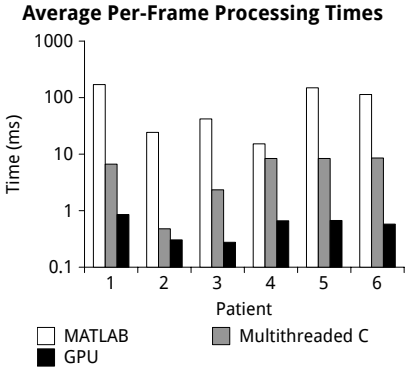

Fig. 4. A comparison of the average per-frame processing times for the three implementations.

The graph in Figure 5 displays the contribution of each action to the GPU run times for a single frame. As can be seen, the first kernel of the numerator stage, which computes the individual contributions of each template tile, dominates the overall runtime. This is mainly because the numerator has to be computed independently for each template, and only processes one template per invocation. The frame data averages and contribution to the denominator are computed once per video frame. As an optimization, the reduction sum kernel for the numerator operates across the full set of templates and is called only once per frame as well.

In the existing *corr2()* implementation, run time compilation is used to compile in up to four customized copies of the main computation loops, one for each needed tile size. This allows for full unrolling of the loops, an important GPU compile-time optimization that is not available when the kernel is compiled once before the template dimensions have been determined. In addition, when main tile sizes that are powers of two are selected, the compiler performs strength reduction on modulus and division operations converting them to bitwise operations. Loops in the reduction kernels are also unrolled.

The performance impact associated with problem-specific kernels can be seen with the fourth patient data set. A 4-by-4 pixel main tile size eliminates edge-case tiles allowing

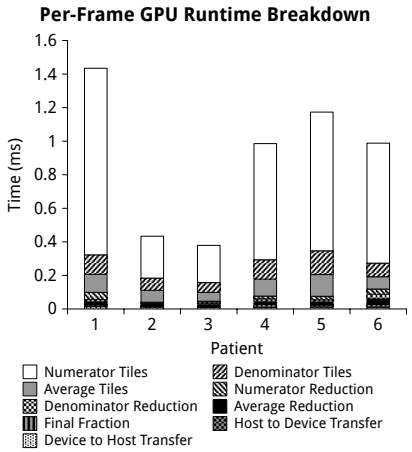

Fig. 5. Average contributions of each action, including data transfers, to the GPU runtime for a single frame.

uniform processing across the tiles, as shown in Table III. However, the best performance was achieved with a 16-by-10 pixel main template size with edge case tiles. The 4-by-4 main tile size resulted in a per-frame GPU-processing time of approximately 1.16 ms per frame, while the corresponding time for the 16-by-10 main tile size was approximately 0.66 ms per frame. Using the latter tile size improves the speedup from approximately 7.23 to about 12.67.

Not only does problem-specific kernel compilation eliminate overhead associated with handling the edge case tiles and their random tile sizes, but it also allows the implementation to better adapt the problem to the GPU hardware. Smaller tile sizes will generate more independent thread blocks that can be important for generating maximum parallelism, as can be observed with the second patient's data set. On the other hand, increasing the tile size improves data reuse out of shared memory and reduces demand on global memory.

Table IV shows positive correlation between increasing template sizes and increases in the fastest observed main tile size. CUDA-capable GPUs have a fixed number of execution resources, so increasing the parallelism through smaller tile sizes will not provide performance improvement once all streaming multiprocessors on a GPU are occupied. After this point, until shared memory limitations are hit, covering larger template sizes by increasing the main tile size will improve memory hierarchy performance.

VIII. Related Work: GPU Adaptability

A few examples of problem-specific kernel compilation for specific applications have been reported. The compressible gas dynamics simulator by Bergen et al. [10] and molecular orbital visualization by Stone et al. [11] incorporate some problem-specific kernel concepts. The former documents several middle-level abstractions the authors developed to increase programmer productivity. Of particular interest, the authors use C-based macros to compile in parameter constants to reduce memory usage and specialize kernels. In the latter work, the authors developed custom loop-unrolled GPGPU kernels

Patient	1	2	3	4	5	6
Template Size	53×54	23×21	76×45	156×116	86×78	141×107
Best Tile Size	16×2	4×4	8×8	16×10	8×8	16×10
Total Speedup	7.80	1.57	8.48	12.67	12.50	14.78

TABLE IV
BEST PERFORMING MAIN TILE SIZE AND TOTAL SPEEDUP OF THE CUDA GPU VERSION OVER THE MULTI-THREADED CPU IMPLEMENTATION.

demonstrating the potential for improving the performance of GPU kernels through the use of custom compilation for a specific problem instance, but did not integrate runtime compilation of GPU kernels into their software.

Another example is presented by Linford et al. [12], who used CUDA for a GPGPU implementation of large scale atmospheric chemical kinetics simulations. The basis of their GPU implementation is a problem instance specific C kernel generated by the Kinetic PreProcessor [13] software package. While the serial C code is problem specific and generated automatically, the conversion to CUDA C appears to be a manual process.

In addition to problem-specific kernel compilation for a single application, a number of more general uses of runtime-based problem-specific kernel compilation have appeared. A number of projects focus on automatically generating GPGPU kernels from interpreted languages. Working with the AMD Compute Abstraction Layer (CAL) [14], which provides a low-level intermediate language and API for AMD GPUs, Garg et al. [15] have created a framework that automatically generates CAL compatible GPU code from parallelism and type annotated Python code. An ahead-of-time compilation step converts Python functions to an intermediate state and a runtime just-in-time (JIT) compiler generates both GPU kernel and host control code using program information that only becomes known at runtime. The JIT compiler can perform loop unrolling and fusion and optimizes some memory access patterns.

Dotzler et al. [16] present a framework (JCudaMP) that takes Java code with OpenMP parallelization pragmas and automatically generates code to run on NVIDIA CUDA capable GPUs. A custom Java class loader converts a subset of Java/OpenMP parallel code sections with additional JCudaMP annotations and dynamically generates C for CUDA source. JCudaMP relies on the CUDA nvcc compiler and invokes it at runtime to compile and link C for CUDA to a shared object that is loaded by the Java virtual machine. The generated CUDA code will check for and indicate addressing violations for exception handling and uses Java's multi-dimensional array organization. While tiling of problems larger than the target GPU's global memory is supported, important aspects of CUDA performance, like shared memory, are not. Overhead information is provided for CUDA compilation but not for CUDA code generation.

Also starting with Java, Leung et al. [17] have modified the JikesRVM Java virtual machine to connect to the RapidMind framework for the purpose of running Java code on NVIDIA GPUs. After parallelization, their modifications generate RapidMind IR to feed directly into the RapidMind backend, which generates target specific code. The RapidMind template libraries and front-end stage is not used. Loops are automatically identified and parallelized, and a cost-benefit analysis is used to decide which loops should be executed on the GPU.

While these interpreted language frameworks do provide automatic runtime kernel generation, which can significantly ease GPGPU development difficulty, performance is limited by the ability of the JIT compiler to identify opportunities for acceleration and optimization. Their fully automated nature, combined with the requirement that runtime kernel generation and compilation introduce as little overhead as possible, places limitations on the ability to effectively utilize many CUDA resources and target arbitrary problems.

A couple of projects rely on PTX (NVIDIA's GPU intermediate representation before final JIT compilation to a GPU's actual ISA). The NVIDIA OptiX ray tracing engine [18] uses run time manipulation and compilation of PTX kernels, but requires them to be compiled offline using the CUDA nvcc compiler. While operating only on PTX, the OptiX PTX to PTX compilation provides a number of domain specific optimizations at runtime by analyzing both the PTX source and the current problem. These are not well documented, but include memory selection, inlining object pointers, register reduction, and efforts to reduce the impact of divergent code.

Ocelot[19], dynamically translates NVIDIA PTX code to Low Level Virtual Machine (LLVM) IR. From there, a just-in-time compiler can generate code for either a multi-core CPU system or NVIDIA GPUs at runtime. The focus of the project is on restructuring GPU-oriented threads and memory use to match multi-core CPU platforms. In addition, these PTX approaches already require CUDA C source to be compiled into a PTX representation. While the CUDA PTX JIT applies a number of optimizations, some are only applied during compilation from CUDA C to PTX.

For many GPGPU users, including library maintainers, working at the higher level of abstraction provided by CUDA C is an attractive option. In most cases, the C and C++ derived syntax is more productive than PTX and provides opportunities for better code development practices. C++ templates can be used to customize or select different implementation approaches for higher levels of adaptability. Our approach also allows for problem and domain specific optimizations to be embedded in the library that automatic tools are not likely to discover. This allows for quick adaptation to arbitrary problems and high performance.

IX. Discussion and Future Work

While the current tiled template implementation has demonstrated good performance, there are a number of avenues for further research. Beyond improving performance though reducing the total number of kernel launches, the application described here is being used as a case study for exploring issues around GPGPU library parameterization and adaptability. The implementation parameter of threads per block was fixed for the various kernels during benchmarking and the optimal tile size was determined from an empirical evaluation. Both these and additional implementation parameters, such as the amount of work per thread, need to be benchmarked with different problems and different GPUs to determine which problem and hardware parameters are relevant and how each affects performance. Autotuning tools may be helpful for this benchmarking and analysis, as well as additional kernel development. However, it is unlikely that autotuning tools alone would be able to develop a solution as effective as the tiled implementation discussed here.

Second, we plan to quantify the benefits of and expand the usage of problem-specific kernel compilation to explore the limitations of the technique, which has the potential to offer significantly increased GPGPU kernel adaptability without sacrificing performance. Both parameterization and problem-specific kernel compilation will be explored through continued work on lung tumor tracking and other applications with large template sizes including particle image velocimetry, dense block matching for stereo disparity maps, and other tracking applications. We also plan to add support for handling the larger template shift areas associated with typical image processing problems within a single kernel launch. This would enable a single two-dimensional sliding-window correlation implementation to handle a wider range of problem instances within the two-dimensional sliding window problem space.

X. Conclusion

This paper discusses a tiled-template correlation-based CUDA implementation of tracking that offers the ability to handle arbitrary problem instances with data set dependent parameters that are not normally considered for GPGPU acceleration and exhibit good performance. Tiling of the template introduces implementation parameters that allow adaptation to both problem instance and hardware characteristics and is the best solution for large template matching. Additionally, problem-specific kernel compilation at runtime provides a mechanism to increase adaptability while maintaining high performance.

Acknowledgment

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

References

[1] *NVIDIA CUDA Programming Guide*, 3rd ed., NVIDIA Corporation, 2701 San Tomas Expressway, Santa Clara, CA 95050, June 2010, http://developer.nvidia.com/object/cuda_3_1_downloads.html, last accessed 1 April 2011.

[2] Y. Cui, J. G. Dy, G. C. Sharp, B. Alexander, and S. B. Jiang, "Multiple template-based fluoroscopic tracking of lung tumor mass without implanted fiducial markers." *Physics in Medicine and Biology*, vol. 52, no. 20, pp. 6229–42, 2007.

[3] J. Kong, M. Dimitrov, Y. Yang, J. Liyanage, L. Cao, J. Staples, M. Mantor, and H. Zhou, "Accelerating MATLAB image processing toolbox functions on GPUs," in *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. New York, NY, USA: ACM, 2010, pp. 75–85.

[4] *Jacket Function Reference*, 1st ed., AccelerEyes, LLC, AccelerEyes, 75 5th St NW STE 204, Atlanta, GA 30308, 2010, http://www.accelereyes.com/content/doc/FunctionReferenceGuide.pdf.

[5] AccelerEyes Website, "filter2() documentation," http://wiki.accelereyes.com/wiki/index.php/FILTER2, last accessed 29 April 2011.

[6] I. K. Park, N. Singhal, M. H. Lee, S. Cho, and C. Kim, "Design and performance evaluation of image processing algorithms on GPUs," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 22, no. 1, pp. 91 –104, January 2011.

[7] NVIDIA NPP Website, http://developer.nvidia.com/object/npp_home.html, last accessed 1 April 2011.

[8] OpenCV Website, http://opencv.willowgarage.com/, last accessed 1 April 2011.

[9] A. Stivala, P. Stuckey, and A. Wirth, "Fast and accurate protein substructure searching with simulated annealing and GPUs," *BMC Bioinformatics*, vol. 11, no. 1, p. 446, 2010. [Online]. Available: http://www.biomedcentral.com/1471-2105/11/446

[10] B. Bergen, M. Daniels, and P. Weber, "A hybrid programming model for compressible gas dynamics using OpenCL," in *Parallel Processing Workshops (ICPPW), 2010 39th International Conference on*, September 2010, pp. 397 –404.

[11] J. E. Stone, J. Saam, D. J. Hardy, K. L. Vandivort, W.-m. W. Hwu, and K. Schulten, "High performance computation and interactive display of molecular orbitals on GPUs and multi-core CPUs," in *GPGPU-2: Proceedings of 2nd Workshop on General Purpose Processing on Graphics Processing Units*. New York, NY, USA: ACM, 2009, pp. 9–18.

[12] J. C. Linford, J. Michalakes, M. Vachharajani, and A. Sandu, "Multi-core acceleration of chemical kinetics for simulation and prediction," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–11.

[13] V. Damian, A. Sandu, M. Damian, F. Potra, and G. R. Carmichael, "The kinetic preprocessor kpp-a software environment for solving chemical kinetics," *Computers & Chemical Engineering*, vol. 26, no. 11, pp. 1567 – 1579, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/B6TFT-46W0XYR-2/2/5e0b300696688c6e1ffe10995dee770e

[14] AMD Accelerated Parallel Processing (APP) SDK Website, http://developer.amd.com/gpu/AMDAPPSDK/Pages/default.aspx, last accessed 1 April 2011.

[15] R. Garg and J. N. Amaral, "Compiling Python to a hybrid execution environment," in *GPGPU '10: Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units*. New York, NY, USA: ACM, 2010, pp. 19–30.

[16] G. Dotzler, R. Veldema, and M. Klemm, "JCudaMP: OpenMP/Java on CUDA," in *IWMSE '10: Proceedings of the 3rd International Workshop on Multicore Software Engineering*. New York, NY, USA: ACM, 2010, pp. 10–17.

[17] A. Leung, O. Lhoták, and G. Lashari, "Automatic parallelization for graphics processing units," in *PPPJ '09: Proceedings of the 7th International Conference on Principles and Practice of Programming in Java*. New York, NY, USA: ACM, 2009, pp. 91–100.

[18] S. G. Parker, J. Bigler, A. Dietrich, H. Friedrich, J. Hoberock, D. Luebke, D. McAllister, M. McGuire, K. Morley, A. Robison, and M. Stich, "Optix: a general purpose ray tracing engine," in *SIGGRAPH '10: ACM SIGGRAPH 2010 papers*. New York, NY, USA: ACM, 2010, pp. 1–13.

[19] G. F. Diamos, A. R. Kerr, S. Yalamanchili, and N. Clark, "Ocelot: a dynamic optimization framework for bulk-synchronous applications in heterogeneous systems," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, ser. PACT '10. New York, NY, USA: ACM, 2010, pp. 353–364. [Online]. Available: http://0-doi.acm.org.ilsprod.lib.neu.edu/10.1145/1854273.1854318