A GPU Based Implementation of Center-Surround Distribution Distance for Feature Extraction and Matching

Aditi Rathi, Michael DeBole, Weina Ge, Robert T. Collins, N. Vijaykrishnan
[1]The Department of Computer Science and Engineering,
The Pennsylvania State University, University Park PA, 16802

Abstract-**The release of general purpose GPU programming environments has garnered universal access to computing performance that was once only available to super-computers. The availability of such computational power has fostered the creation and re-deployment of algorithms, new and old, creating entirely new classes of applications. In this paper, a GPU implementation of the Center-Surround Distribution Distance (CSDD) algorithm for detecting features within images and video is presented. While an optimized CPU implementation requires anywhere from several seconds to tens of minutes to perform analysis of an image, the GPU based approach has the potential to improve upon this by up to 28X, with no loss in accuracy.**

I. INTRODUCTION

Feature extraction is an essential function for most computer vision and image processing applications. These features play a role in a variety of domains such as surveillance, media mining, camera processing, as well as many others. However, automatically determining important features within an image or frame for performing tasks such as scene matching or image recognition is still an ongoing research area. More importantly, more complex feature extraction mechanisms require a high degree of computation which in some cases can still not be provided by state-of-the-art multi-core CPUs.

With the introduction of user-friendly programmable Graphics Processing Units (GPUs), performance is now available to the general application programmer that was once only available to a small subset of the community. In particular, with the introduction of the Computer Unified Device Architecture (CUDA) from NVIDIA, the task of porting parallel portions of an application to the GPU has become easier than ever before. In this paper, a compute intensive feature extraction algorithm known as the Center Surround Distribution Distance (CSDD) [1] is investigated and accelerated on a GPU using CUDA. The CSDD algorithm is a feature detector which attempts to detect blobs of different sizes on an image and determine whether they standout perceptually from the background. To that end, because of the amount of computation and memory required to execute the algorithm, an optimized implementation on the CPU can take minutes per frame. Further, while the algorithm isn't fully parallelizable, there are portions which benefit greatly by being ported to a GPU leading to advantageous speedups.

This paper introduces the CSDD algorithm in terms of both a CPU implementation, followed by an implementation which would lend itself to a CUDA-based GPU. This includes the challenges involved in mapping the algorithm to the GPU as well as tradeoffs and optimizations that needed to be considered in order to achieve the best performance. Using the strategies employed here the GPU implementation was able to provide consistent speedup up of 14X for all image sizes.

The remainder of the paper is organized as follows: Related work is provided in Section II. A background on NVIDIA GPUs and the CUDA programming environment is given in Section III. Section IV introduces the CSDD algorithm and Section V details its implementation. Section VI provides the results and the paper is concluded in Section VII.

II. RELATED WORK

With the introduction of NVIDIA's Compute Unified Device Architecture (CUDA) there has been an extensive amount of work mapping algorithms to the GPU. In [2] the authors use GpuCV, an open source framework for acceleration image processing and computer vision applications on the GPU. Using GpuCV a comparison is performed against Cimg to compare a Deriche filter implementation on each device. By using the GpuCV library the authors were able to obtain up to 101X speedup, however this varied wildly depending on image size (2.5X~101X). While the GPU used is the same as with this paper, there are two distinct differences. The first is that with the GpuCV framework the authors limit themselves to a single image instead of having to consider a situation where many Deriche filtering operations must occur on a large set of data. The second is that only single-precision floating point is considered. [3] implements non-rigid registration for 3D volumes using Gaussian recursive filtering and compares implementations on a CPU with those using openVidia on the GPU. With openVidia and the GPU the authors were able to obtain a speedup of 10X for a 128^3 volume. This work differs in that it uses a non-CUDA enabled GPU which exclusively supports single-precision computation. As with [2], the authors in [3] also only consider a single volume rather than having to consider a GPU implementation consisting of performing filtering over multiple images. In [4] a number of filtering operations, including Deriche filtering are implemented and compared for speedup between a CUDA enabled GPU and a CPU. However, the authors again rely on single-precision and only consider single image filtering operations. Finally, the authors in [5] study general-purpose applications using CUDA. In [5] the authors use the same GPU as in this paper, however only consider single precision and do not consider Deriche filtering.

[1] This work was supported in part by NSF grants 0702617 and 0903432.

III. NVIDIA GPUs AND CUDA

Today, modern GPUs are a collection of a large number of multi-threaded cores intended to take advantage of highly parallel tasks. While different GPU vendors have different architectures, NVIDIA provides an architecture consisting of an array of Streaming Multiprocessors (SM) with each SM containing eight, Single Instruction, Multiple Data (SIMD) Stream Processors (SP). In addition, each SM contains two special function units, a multi-threaded instruction unit, and on-chip shared memory [6]. The number of available SMs depends on the GPU version and can range anywhere from 8 to 128 [7]. The key to general purpose use of the GPU however, is CUDA. CUDA is the general purpose parallel computing architecture which provides a C/Java/Python/Fortran-based API to the CUDA Instruction Set Architecture (ISA) and parallel compute engine in the GPU [7]. The CUDA core consists of three abstractions: a hierarchical model of threads, a controllable memory hierarchy, and barrier synchronization.

In CUDA, the basic unit of parallelism is a thread. Each SM is capable of executing 32 threads simultaneously, known as a *warp*, providing for the possibility that hundreds of threads could be running concurrently. As a result, CUDA adopts an architecture known as Single Instruction, Multiple Thread (SIMT) in order to manage the complexity associated with concurrently executing such a large number of threads. From a programmers perspectives these threads are managed at a much higher level, in the form of a kernel. A kernel is a collection of threads (known as a *grid* of threads) where each *grid* consists of a number of thread blocks. Thread blocks are scheduled to execute on a particular SM and are split into *warp*s. The SIMT unit is responsible for scheduling the *warps* within a SM.

The controllable memory hierarchy within CUDA provides the programmer with four distinct types of memories accessible by an SM, not including registers. Closest to the SM, and consequently that with the fastest access time, is the *shared memory* to provide communication amongst threads within a thread block. There are two other memories, the *constant cache* and *texture cache* which are also shared by all SPs providing read-only access to their contents. Between SMs there is a much larger, albeit slower, *global memory*. The global memory is, in part, used as the *local memory* which is shared amongst the SMs to handle per thread register spills. CUDA also provides methods for synchronization, however while there exists a barrier synchronization mechanism for threads within a block, there are no synchronization mechanisms available for threads across blocks.

Despite all the unique features of CUDA, there are still a number of issues which can limit the performance achieved by a GPU application developer. The biggest challenge is that of striking the right balance between a thread's resource usage and the number of simultaneously active threads. Ultimately, it is the programmer's responsibility for ensuring that there is an efficient implementation which properly balances computation and memory accesses in order to achieve the desired performance.

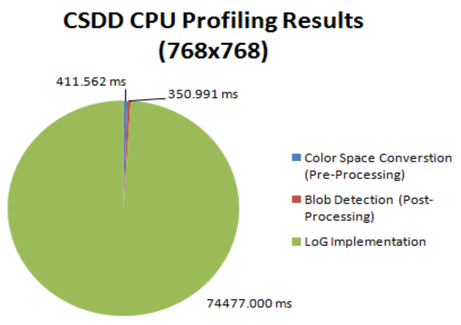


Figure 1. Profiling Results of CSDD on CPU

IV. CENTER-SURROUND DISTRIBUTION DISTANCE

Detection and extraction of features within images and video has become an important part to a number of computer vision applications. Of particular importance is the ability to use extracted features to match corresponding image regions across large changes in viewpoint. The authors in [1] use the formulation of CSDD in order to obtain an interest region operator capable of determining "important" features within an image. The primary objective of the CSDD operator is to compare feature distributions between a foreground region, the *center region*, and an immediate background region, the *surrounding region*, to detect blobs that perceptually "stand out" because they look different than the background. The remainder of this section lends a brief description of a practical implementation of the CSDD algorithm followed by a high level view of its partitioning on the GPU. For a more complete description of the algorithm the reader is encouraged to consult [1].

A. *Algorithm Overview*

The central operation towards performing feature extraction using CSDD is performing a number of comparisons using Mallow's distance [8] for center surround distributions, color histograms extracted at each pixel, over a scale space formed by a discrete number of scales. These distributions can be made as joint RGB distributions, however to reduce computational complexity the joint RGB distribution can be approximated as three 1D marginals. This is done by first transforming the RGB color space to the Ohta color space [9] yielding a set of color planes which are approximately uncorrelated for natural images and concatenating different color channels.

It is then possible to form a CSDD measure at every pixel by extracting distributions corresponding to the center region, $F(v)$, and those corresponding to the outer region, $G(v)$, using the images that result from finely sampling along each of the marginal color axes. The operation involves performing convolution with the binary indicator function $\delta(I(x)\leq v)$ and with a Laplacian of Gaussian (LoG) filter at scale σ:

$$F(v) - G(v) = \frac{e\sigma^2}{2} \int_{R^2} \delta(I(x) \leq v) \nabla^2 G(x; x_0, \sigma)\, dx \quad (1)$$

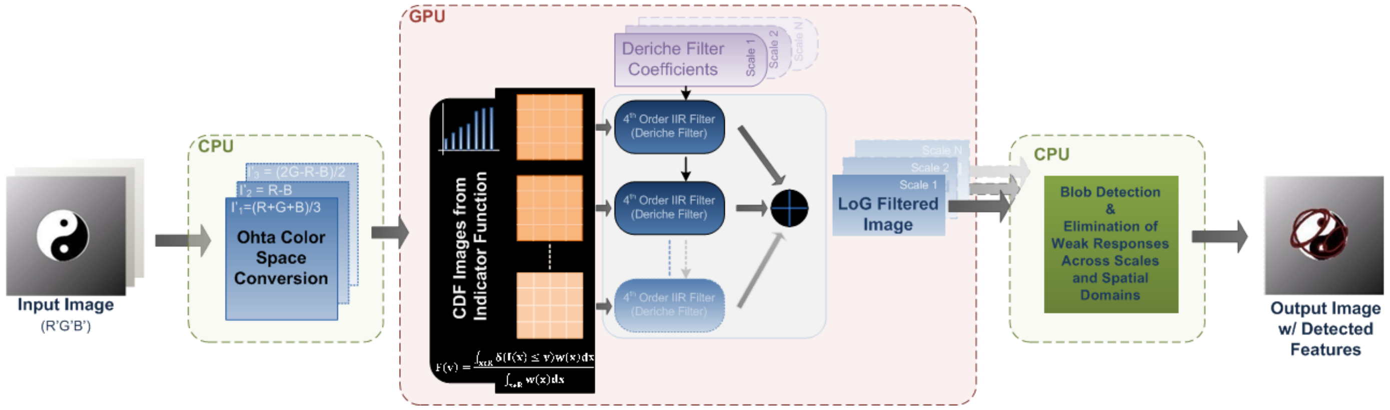

GPU
Deriche Filter Coefficients
Scale 1
Scale 2
Scale N
CPU
$I'_3 = (2G-R-B)/2$
$I'_2 = R-B$
$I'_1=(R+G+B)/3$
Ohta Color Space Conversion
Input Image
(R'G'B')
CDF Images from Indicator Function
$F(v) = \frac{\int_{x \in R} \delta(I(x) \le v) w(x) dx}{\int_{x \in R} w(x) dx}$
4th Order IIR Filter (Deriche Filter)
4th Order IIR Filter (Deriche Filter)
4th Order IIR Filter (Deriche Filter)
Scale N
Scale 2
Scale 1
LoG Filtered Image
CPU
Blob Detection & Elimination of Weak Responses Across Scales and Spatial Domains
Output Image w/ Detected Features

Figure 2. High-Level View of CSDD Algorithm with Partitioning across CPU and GPU

Computationally, the most important aspect of computing the CSDD is performing efficient LoG filtering. This is of particular importance for this algorithm because LoG filtering has to be performed once for each set of finely sampled values and across scales. To reduce LoG filtering to constant time, the implementation used in this work is based on a set of recursive IIR filters proposed by Deriche and later improved by Farneback and Westin [10]. These filters keep constant the number of floating point operations (8 multiplies, 7 additions) per pixel, regardless of the spatial size of the operator, σ. This is done by using a fixed-term recurrence relation to define what would typically be a convolution of a spatial filter at different sizes. As shown later, the data dependence of a recurrence relation will ultimately impose some operations to be performed sequentially on the GPU.

The resulting dissimilarity value determines whether a region of interest achieves a local maximum across the spatial domain and the scales. An additional advantage to this technique is that it is capable of determining rotationally invariant and scale covariant interest regions.

B. GPU Considerations

Loop 1:
Gaussian: Forward pass along rows (causal)
Loop 2:
Gaussian: Backward pass along rows (anti-causal)
Loop 3:
2nd Derivative of Gaussian: Forward pass along columns (causal)
Loop 4:
2nd Derivative of Gaussian: Backward pass along columns (anti-causal)
Loop 5:
2nd Derivative of Gaussian: Forward pass along rows (causal)
Loop 6:
2nd Derivative of Gaussian: Backward pass along rows (anti-causal)
Loop 7:
Gaussian: Forward pass along columns (causal)
Loop 8:
Gaussian: Backward pass along columns (anti-causal)
Loop 9:
Accumulation of results from ***loop 1-4*** *and* ***loops 5-8*** from **ALL** CDF images

Figure 3. Loop outline of LoG Filter Implementation

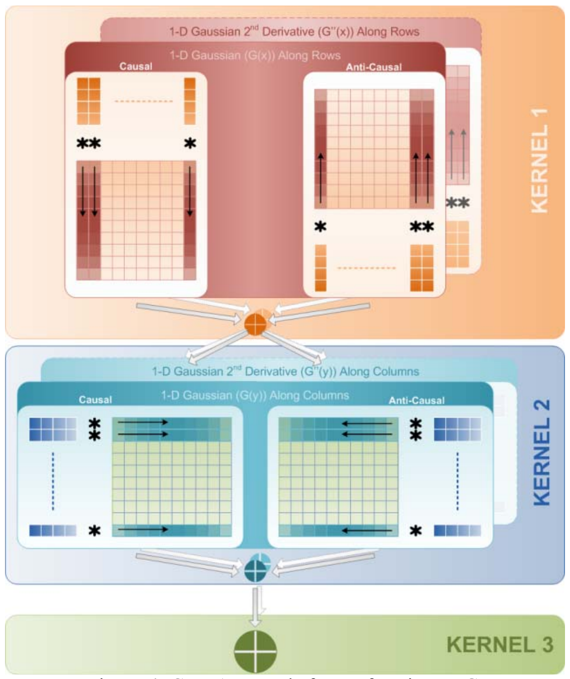
The GPU has the potential to provide a tremendous amount of speedup for many applications, but it is critical to first determine which portions of the algorithm to map onto the GPU. For the CSDD implementation this was done by profiling an optimized CPU code and observing which portions would achieve the most benefit by being run on the GPU. The results of profiling are shown in figure 1. As expected, the most time consuming portions are those which finely sample the color axes, create cumulative density function (CDF) images from the binary indicator functions, and then perform LoG filtering.

A high-level CPU/GPU partitioning of the CSDD algorithm is shown in figure 2. After reading the input image, the first operation is to convert the color space from RGB to Ohta. These pixel operations remained on the CPU because it contributed a small percentage to the overall execution time. On the GPU, the generation of the CDF images and Deriche filtering operations were performed. In this particular application each axis has been quantized into 128 values resulting in 128 images per channel. Following the generation of the CDF images the 4th order IIR filters are convolved with each CDF image, and for each scale a different set of coefficients are used. An additional requirement is that the LoG filter results must be in double precision in order to maintain the accuracy requirements necessary for post-processing non-maximal suppression and blob detection.

After performing filtering on the CDF images the resulting filtered images are combined and moved back to the CPU to perform blob detection across scales and remove any weak responses. Finally, the resulting image contains those regions which had a suitable dissimilarity score to be considered a region of interest.

V. Implementation on GPU

Efficiently mapping the LoG filter to the GPU requires that several issues associated with GPU implementations be discussed. This includes a more detailed explanation of the LoG filter implementation, a discussion of the kernels needed, and how to organize data in the GPU memory.


1-D Gaussian 2nd Derivative (G''(x)) Along Rows
1-D Gaussian (G(x)) Along Rows
Causal
Anti-Causal
KERNEL 1
1-D Gaussian 2nd Derivative (G''(y)) Along Columns
1-D Gaussian (G(y)) Along Columns
Causal
Anti-Causal
KERNEL 2
KERNEL 3
Figure 4. CUDA Kernels for Performing LoG

A. *LoG Filter Implementation Overview*

The LoG filter is implemented as a set of 4th order IIR filters arranged over nine loops. Figure 3 outlines the LoG filter if it were to be executed sequentially. The first eight loops can be arranged into two independent sections, loops 1-4, denoted Gau_x-$Deriv_y$ (Gaussian in the X-direction, 2nd derivative of Gaussian in Y-direction) and loops-5-8, denoted $Deriv_x$-Gau_y (Gaussian in the Y-direction, 2nd derivative in the X-direction). Both $Deriv_x$-Gau_y and Gau_x-$Deriv_y$ independently operate on the CDF images and each produce a resulting LoG filtered image. This is repeated for each CDF image at which time loop 9 is responsible for accumulating all the results into a single filtered image. This process is then repeated for each channel, $I_1I_2I_3$, and across scales. The main constraint for parallelizing the LoG filter is the Deriche filter's organization as a recurrence relation. The Deriche filter requires that each pass, either a 4x1 (rows) or 1x4 mask (columns), operate on the current pixel and the previous four results from the mask (eg. $a_n = p_n m_0 + a_{n-1} m_1 + a_{n-2} m_2 + a_{n-3} m_4 + a_{n-4} m_5$). This introduces a portion of code which must be executed serially as well as additional information which must be stored per pixel.

B. *CUDA Kernels for LoG*

When using CUDA all operations that will be executed onto the GPU must be organized into kernels. For the LoG implementation those kernels performed both the creation of CDF images and LoG filter implementations where the indicator function is carried out during *kernel*-1. Figure 4 shows how the loops are distributed into three CUDA kernels in order to exploit thread level parallelism (*kernel-1*: 1,2,5,6; *kernel-2*: 3,4,7,8; *kernel-3*: 9). In this figure rows of the image are stored along the rows (rather than the columns) and vice versa for columns of the image. The most important determination in choosing this organization was the synchronization constraints of CUDA. With CUDA the only reliable way to perform global synchronization between threads in different blocks was to explicitly kill the kernel because of the lack of synchronized write mechanisms. Since the operations between loops 1-2 (5-6) to 3-4 (7-8) must be carried out sequentially, while loops 1-2 (3-4), 5-6 (7-8) can occur in parallel, the producer-consumer relationship would not have been able to be maintained if the original loop ordering was used. In contrast, loops 1, 2, 5, 6 (3, 4, 7, 8) in kernel-1 (-2) are serialized, despite being independent, in order to improve memory re-use. Other factors which must be taken into consideration in order to achieve good performance include memory considerations, data dependencies, thread synchronization, and finally, performance scalability across image sizes.

One particular concern which effects computation, memory requirements, and bandwidth is how to efficiently use the available global memory. In this application, the global memory is used to store the input image, filter coefficients, and the intermediate results that occur between kernels given a particular image size. However, the global memory is not used to store computed CDF images, but instead the comparison is carried out when each CDF image is actually needed, due to limited CPU-GPU communication bandwidth.

With each kernel it is also necessary to determine which portions of the computation to break into thread blocks. With the Deriche filter, individual pixels cannot be computed independently because there are row-wise data dependencies (*kernel*-1) and column-wise data dependencies (*kernel*-2). However, across rows (columns) it is possible to execute the operations independently and therefore each row-wise (column-wise) operation is the basis of a thread. Unfortunately, it is not possible to perform all row (column) threads simultaneously, across all CDF images, because there are not enough computational resources available, especially for larger images. As a result, the image is further *tiled* into multiples of 64 threads, as shown in figure 5, and these tiles are assigned to thread-blocks. Tiling at this granularity has two motivations. The first is to help overcome latencies by allowing the scheduler to efficiently split the entire problem such that the occupancy is as high as possible. The second is to allow for the performance to scale as GPU hardware improves and the number of SMs increase.

Kernel-3 is organized in a more serial fashion across CDF images because there is no provision of synchronized writes to the same location within a kernel. To handle this, *kernel*-3 performs the final accumulations by parallelizing the additions within a single CDF image and serializing the accumulations across all CDF images.

C. *CUDA Execution Configuration and Optimizations*

There are a number of additional parameters which need to be appropriately tuned in order to achieve the maximum performance on the GPU. Choosing these execution parameters is a matter of striking a balance between latency hiding and efficient resource utilization.

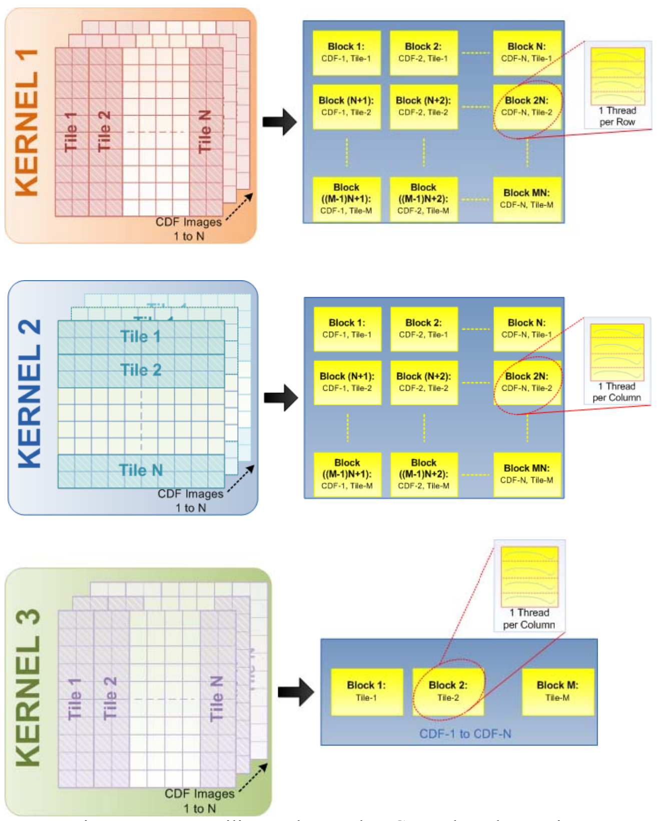


Figure 5. Image Tiling and Kernel to GPU Thread Mapping

In many cases, performance is achieved through experimentation in an attempt to find the best combination of parameters. For the CSDD application the maximum number of available registers per thread was varied along with threads per block. The number of registers was experimentally found to produce the best performance when set at a soft limit of 20 for all three of the kernels. The number of threads per block was set to be a multiple of 64 in order to achieve efficient utilization of compute resources, provide coalesced memory accesses, avoid register memory bank conflicts, and increase occupancy. This has one caveat where, in order to provide threads at a multiple of 64, the image had to be padded during pre-processing. The final numbers of threads per block varied between 64, 192, or 256 depending on the kernel and image size.

In General, values for the different parameters were decided by both CUDA- and CSDD-specific considerations with a heavy reliance on heuristic information gathered during experimentation. Figure 6 shows this tradeoff in the form of occupancy versus resources used. For example, by examining the tradeoffs it was desired to achieve greater than 50% occupancy (recommended for best performance) while having at least 3 active blocks per SM for this particular application.

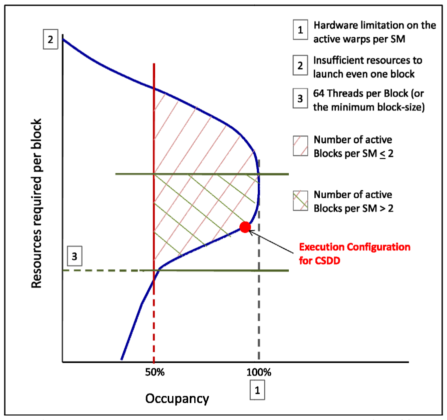
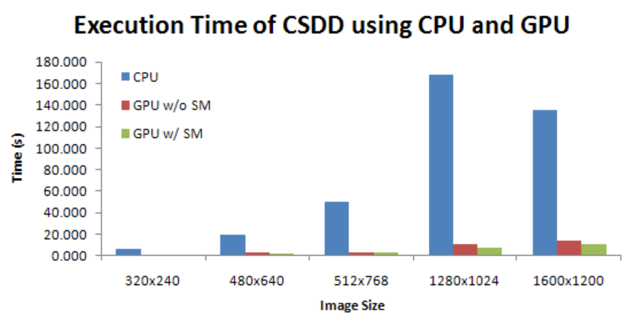


Figure 6. Occupancy vs. Resources Tradeoffs

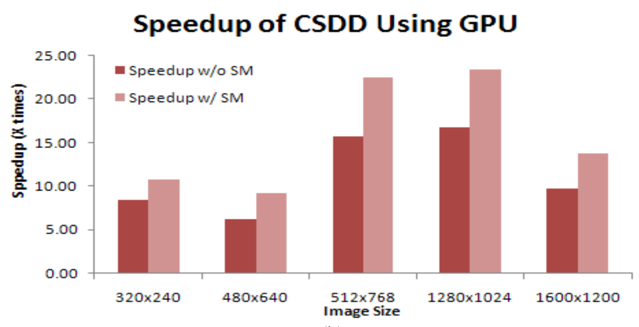
The last major optimization considered was the use of the low latency memories. This was extremely beneficial for *kernel*-1 and *kernel*-2 because there is no concurrent reuse of the CDF images by threads of different blocks and threads of the same block. First, for all three kernels it is possible for the filter coefficients and other constants to be read into the shared memory once per kernel and reused. Despite introducing warp serializations due to divergence this reduced the execution time of the LoG by up to 15%. Additionally, in *kernel*-1 the initial columns can be read into shared memory and reused causing an improvement of 35%. In *kernel*-2 the output can be written into the local memory and finally updated in global memory after the result has been accumulated giving a performance improvement of around 25%. Finally, in *kernel*-3 using shared memory yields another 25% improvement where every location of the output image is read into shared memory and only after the thread writes the final result is it written back. The net improvement after applying all shared memory optimizations was an additional 30% for most image sizes.

VI. RESULTS

The CSDD algorithm was mapped onto a system which contained a dual core Intel Xeon processor (3.59 GHz) with

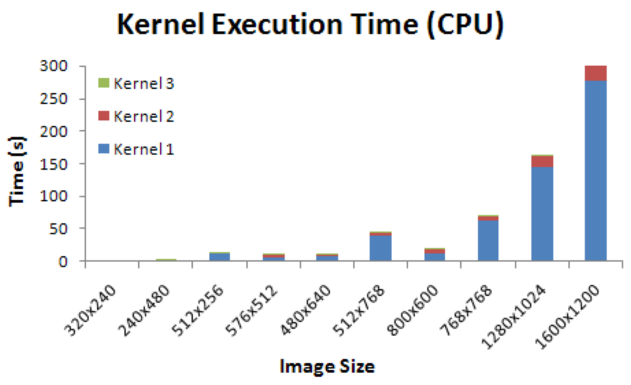


(a)

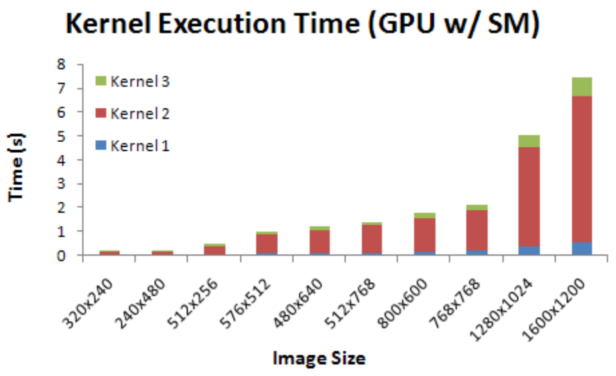


(b)

Figure 7. (a) Execution time and (b) Speedup of the Entire CSDD Application



(a)



(b)

Figure 8. Comparison of kernel execution time on the (a) CPU and (b) GPU

3GB RAM and a NVIDIA GeForce GTX 280 GPU. Using this system a variety of frame sizes and shapes were used as inputs to the CSDD implementations. Only a small sample of image sizes are shown here due to space limitations and the results are limited to gray-scale images consisting of a single channel. The only implication of limiting this discussion to gray-scale is that the execution times for 3-channel images should be three times as large (for both CPU and GPU). The overall speedup remains relatively unchanged.

Figure 7 depicts the overall application time of the CSDD implementation on both the CPU and GPU for the selected image sizes. Even using optimized CPU code the algorithm still requires anywhere from several seconds to a few minutes for a single frame. The GPU however, improves upon this time significantly, consistently achieving speedup of at least 14X depending on the image size. The overall execution time for the GPU ranges from a few hundred milliseconds up to a several seconds for single channel.

In a few instances the GPU is able to improve upon the CPU execution time up to 25X~30X. In particular this occurs with images that are multiples of 256, a multiple of the cache line size of the CPU. This increase in speedup is due to cache hit/miss behavior causing an increase in the CPUs execution time during *kernel*-1, where the elements are accessed row-wise when the data is actually stored in column-major order. This is shown in figure 8, where the contributions of each kernel to the total execution time are broken down. Another interesting observation is that the dominant kernel changes moving from the CPU to GPU implementation. The reason for *kernel*-2 having a higher execution time is because there is increased use of the global and local memories (9Xs) in order to allow for the outputs of common variables to be re-used. This is in contrast to the CPU where the cache behavior of *kernel*-1, due to the format of data storage, causes it to dominate the execution time.

VII. CONCLUSION

This paper presented a GPU-based CSDD implementation which focused on the parallelization of the LoG filtering operation. Through the judicious use of features present on the GPU such as local and shared memories, intelligent partition of kernels and threads, and an attempt to maximize occupancy, the GPU was able to provide a consistent ~14X average speedup and in certain cases up to 28X. This implementation is highly scalable with SMs and different image sizes. This will allow moving the implementation to more powerful GPUs easy in order to achieve higher performance in the future.

REFERENCES

[1] R. T. Collins and W. Ge, "CSDD Features: Center-Surround Distribution Distance for Feature Extraction and Matching," in *European Conference on Computer Vision (ECCV)*, Marseille, France, October 2008.

[2] Y. Allusse, P. Horain, A. Agarwal, and C. Saipriyadarshan, "GpuCV: A GPU-Accelerated Framework for Image Processing and Computer Vision," in *Lecture Notes in Computer Science*. Vancouver, British Columbia, Canada: Springer Berlin / Heidelberg, 2008, pp. 430-439.

[3] N. Courty and P. Hellier, "Accelerating 3d Non-rigid Registration Using Graphics Hardware," *International Journal of Image and Graphics (IJIG)*, vol. 8, no. 1, pp. 81-98, Jan. 2008.

[4] D. Castano-Diez, D. Moser, A. Schoenegger, S. Pruggnaller, and A. S. Frangakis, "Performance evaluation of image processing algorithms on the GPU," *Journal of Structural Biology*, vol. 164, no. 1, pp. 153-160, Oct. 2008.

[5] S. Che, et al., "A performance study of general-purpose applications on graphics processors using CUDA," *Journal of Parallel and Distributed Computing*, vol. 68, no. 10, pp. 1370-1380, Oct. 2008.

[6] NVIDIA. (2008, Jun.) NVIDIA CUDA. [Online]. http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

[7] NVIDIA Website. [Online]. www.nvidia.com

[8] E. Levina and P. Bickel, "The Earth Mover's Distance is the Mallows Distance: Some Insights from Statistics," in *Proceedings of the 8th IEEE International Conference on Computer Vision*, Vancouver, 2001, pp. 251-256.

[9] Y.-I. Ohta, T. Kanade, and T. Sakai, "Color Information for Region Segmentation," in *Computer Graphics and Image Processing 13*, 1980, pp. 222-241.

[10] G. Farneback and C.-F. Westin, "Improving Deriche-style Recursive Gaussian Filters," *Journal of Mathematical Imaging and Vision*, vol. 26, no. 3, pp. 293-299, Dec. 2006.

[11] S. May, M. Klodt, E. Rome, and R. Breithaupt, "GPU-accelerated Affordance Cueing based on Visual Attention," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, San Diego, CA, 2007, pp. 3385-3390.

[12] R. Deriche, "Fast algorithms for low-level vision," in *9th International Conference on Pattern Recognition*, vol. 1, 1988, pp. 434-438.