GPU-Accelerated Computation for Robust Motion Tracking Using the CUDA Framework

Jing Huang, Sean P. Ponce, Seung In Park, Yong Cao, and Francis Quek[†]

Center of Human Computer Interaction
Virginia Polytechnic Institute and State University
Blacksburg, VA 24060, USA
[†]quek@vt.edu

Keywords: General Purpose GPU processing, Dynamic Vision, Motion Tracking, Parallel Computing, Video Processing

Abstract

In this paper, we discuss our implementation of a graphics hardware acceleration of the Vector Coherence Mapping vision processing algorithm. Using this algorithm as our test case, we discuss our optimization strategy for various vision processing operations using NVIDIA's new CUDA programming framework. We also demonstrate how flexibly and readily vision processing algorithms can be mapped onto massively parallelized GPU architecture. Our results and analysis show the GPU implementation exhibits a performance gain of more than 40-fold of speedup over state-of-art CPU implementation of VCM algorithm.

1 Introduction

Advances in hardware for graphics computation provide a tantalizing possibility for significant gains in computer vision processing power. There is great potential for large performance gains at far lower costs than vision computation with traditional computational architecture. To realize these gains, it is critical that software development tools be available to support computer vision implementation. It is also important to show that computer vision and image processing algorithms can be readily mapped onto the software and hardware GPU architecture.

In this paper, we address the viability of GPUs for general purpose vision processing by showing that emerging programming frameworks support efficient mapping of vision algorithms onto graphics hardware. To this end, we implemented a computation and data-intensive algorithm for motion vector extraction on various GPUs and compare the performance against a state-of-the art CPU.

In section 2, we introduce the relevant background for our project by outlining recent gains in GPU hardware and programming frameworks, discuss the use of GPUs to vision processing, and discuss the rationale for our choice of algorithm to evaluate the viability of GPUs in vision processing. In section 3, we describe our test algorithm. In section 4, we detail the implementation of the algorithm, discussing the various design and optimization choices we made. We present our results in section 5, and end with our concluding remarks in section 6.

2 Relevant Background

2.1 General Purpose GPU Programming

With the speed increase of conventional microprocessor stabilizing, Graphics Processing Units (GPUs) are gaining attention for their low-cost computational power with their massively parallel computing architecture. For example, the GPU in NVIDIA's 8800 GTX has 128 processing cores, each of which is a Single Instruction Multiple Data (SIMD) processor that computes a 4-component vector in one clock cycle. This GPU has a theoretical peak 520 GFLOPS capacity and costs less than $500.

GPUs are originally designed only to enhance the performance and graphics-related applications. As GPUs have become more ubiquitous and powerful, the graphics community

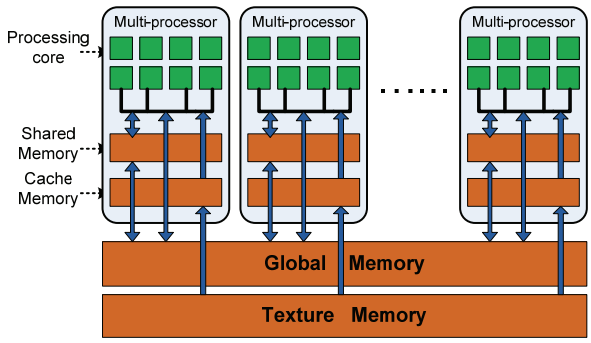

Figure 1 G80 GPU architecture. The G80 GPU consists of a set of Multi-processors. Each Multi-processor has 8 SIMD processing cores, that share 16K of shared memory and 8K of cache memory. *All Multi-processors access read*-only texture memory *and read-and-write* global memory*, each of which 100s of MB of memory space*

developed programming models and tools to help people to use GPUs. Until recently, such programming frameworks have been based on graphics computational concepts – *Shaders*, including "Vertex Shader" and "Fragment Shader". This has proved to be challenging for the use of GPUs outside graphics rendering [6] because general computing algorithms have to be mapped to the computing of vertex transformation or pixel illumination. In addition, shader-based programming restricts random memory access for writing, which distinguishes itself from general CPU programming models.

Recently, NVIDIA released the new G80 architecture, which dramatically changed vertex+fragment shader based streaming computational model. In the G80 architecture, the unified processing core can serve as either a vertex or a fragment processor depending on the specification from the application program. The unified processor can also write on-chip memory to random locations. Another major improvement of the G80 GPUs is the introduction of controllable high-speed memory shared by a block processing cores. The insertion of shared memory into GPU memory hierarchy (Figure 1) enables the flexibility of avoiding memory access latency.

In order to facilitate programming on G80 GPUs, NVIDIA released the Compute Unified Device Architecture (CUDA) [5] framework to assist developers in general propose computing. CUDA provides a C programming SDK, releasing people from learning graphics programming APIs. CUDA also supports random memory writes to on-chip shared memory and global memory, which makes it follow a canonical programming model.

CUDA is a programming framework for massively parallel computing on GPUs. It manages a large number of programming *threads* and organizes them into a set of logic *blocks*. Each thread block can be mapped onto one multi-processor to complete its computational work. An efficient hardware thread management is in place to organize and synchronize threads inside a block. The CUDA programming model also supports multiple *grids* of computational blocks, where each grid has its only function *kernel*.

The new G80 architecture with the CUDA programming framework has raised general purpose GPU computing to a higher level, where researchers from many different areas can take advantage of low-cost parallel computational power of GPUs without struggling with graphics concepts.

2.2 GPU Computation in Computer Vision

Since images are 2D arrays of pixel data, computer vision and image processing can take advantage of SIMD architectures effectively. Several applications of GPU technology for vision has already been reported in the literature. Ohmer et al. [1] demonstrated real-time tracking of a hand using gradient vector field computation, Canny edge extraction, and a geometric model of the hand. De Neve et al [2] implemented a pixelwise YCoCg-R color transform on a shader-based GPU program. Ready et al [11] describe a tracking system for aerial image stabilization that features extensive communication between the CPU and GPU. Their system can track up to 40 points in real-time, computes the stabilization transform in the CPU and does the image rectification in the GPU. This system suffers from steep overheads in CPU-GPU data transfer. Mizukami and Tadamura [4] presented a CUDA-based encoding of Horn and Schunck's [3] optical flow algorithm. Using CUDA's shared memory feature, they were able to get an approximately 2.25 time speed up using a NVIDIA GeForce 8800GTX card over a 3.2 GHz CPU on Windows XP.

2.3 Applying GPUs to Vector Coherence Mapping (VCM)

In order to test the effectiveness of GPUs for vision processing, we selected a test algorithm to facilitate comparison across computing architectures. We chose Vector Coherence Mapping (VCM) [1, 7, 8] as our test algorithm because it is intrinsically parallel, and is constituted in three stages, each of which presents a different memory access and computational profile. We expect that the lessons learned from the implementation of VCM in GPU architecture are transferable to other vision algorithms. VCM is discussed in greater detail in the following section.

3 Vector Coherence Mapping

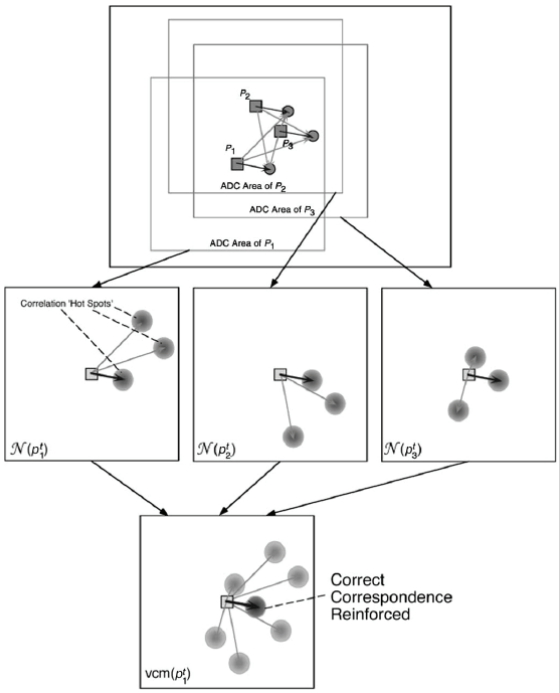

Figure 2 Spatial Coherence Voting in VCM

Vector Coherence Mapping (VCM) is a video motion tracking algorithm first introduced by Quek et al [1, 7, 8], and has been applied to a wide variety of gesture analysis [9, 10, 12]. VCM is an inherently parallel algorithm that tracks sets of interest points as a spatially and temporally coherent vector fields from raw video. The uniqueness of the algorithm is that it applies fuzzy image processing techniques to enforce a series of constraints transparently within a weighted voting process.

Figure 2 illustrates how our VCM applies a spatial-coherence constraint to minimize the directional variance of a flow field. Three interest points detected across two frames are shown. The shaded squares at the top of the figure represent the locations of three feature points at time *t*, and the shaded circles represent the corresponding position of these feature points at *t+1*. If all three interest points have equal correspondence in the next frame, correlation based matching from each of the feature point would provide three hotspots. The three frames labelled $\mathcal{N}(p_1')$, $\mathcal{N}(p_2')$, and $\mathcal{N}(p_3')$ in the middle of Figure 2 show the correlation results for the three points (each frame, called a *normal correlation map* (*ncm*) is centered on the corresponding interest point p_i^t). By using this weighted summa-

tion of neighboring *ncms* we can obtain the vector that minimizes the local variance of the vector field as shown at the bottom of Figure 2. Hence the *Vector Coherence Map* of point p_i^t, denoted $vcm(p_i^t)$ is given by the summation:

$$vcm\left(p_i^t\right) = \sum_{p_j^t} w\left(\left|p_i^t, p_j^t\right|\right) ncm\left(p_j^t\right) \quad (0.1)$$

Here, the summation denotes pixelwise addition of the corresponding *ncm*s, and $w(|p_i^t, p_j^t|)$ is a weighting function based on the distance between the two points. Hence, VCM allows the normal correlation maps of neighboring interest points to influence the vector computed at each point.

By maintaining a history of vectors extracted in previous frames, VCM is able to estimate the termination point of the next vector by applying a constant acceleration assumption. This 'temporal estimate' is used to bias the vote to pick a vector that maximizes both spatial and temporal coherence of the resulting vector field.

The selection of appropriate interest points is critical to the operation of VCM. Since *ncm*s are computed by template correlation, the observation is made that such correlations are most meaningful for patches of high spatial variance. Also, since VCM computes motion fields, the interest points selected should exhibit temporal variance across frames. Hence the interest points are computed at maxima of the "fuzzy AND" between the spatial gradient image (we use a Sobel operator to estimate this) and the temporal change image (we employ image differencing to estimate $\partial I/\partial t$).

4. Approach and Implementation

4.1 Implementation Overview

4.1.1 Choice of CUDA

GPUs are designed to enhance the performance of massively data parallel applications, such as the ones in graphics. The appropriate applications for GPUs are the ones that feature data independence between parallel computational units. Most of vision algorithms, like VCM, match this requirement.

The performance bottleneck for most GPU based data parallel algorithms is memory access, including data loading and storing. NVIDIA's G80 architecture also presents a performance sensitive memory hierarchy. Algorithm implementation has to be carefully designed to optimize memory performance.

NVIDIA's CUDA provides a simple framework to manage data-parallel programs. No code is needed for thread/process management as these are done efficiently by the GPU. CUDA also allows programmers to have very flexible control over memory access. We believe that CUDA is the right framework to implement and analyse vision algorithms on GPUs.

4.1.2 Problem Decomposition

Our test algorithm, VCM is composed of three main phases: Interest Point (*IP*) extraction, NCM computation and VCM computation. These phases repeat for each video frame processed. The output of each iteration is a list of vectors of interest point motion across the two frames. For the purposes of studying the mapping of vision algorithms and their efficiency on GPU architectures, we divided our task into three test components corresponding to the three VCM phases. Each of these components exhibits different memory access and algorithmic profiles. The characteristics of each phase are

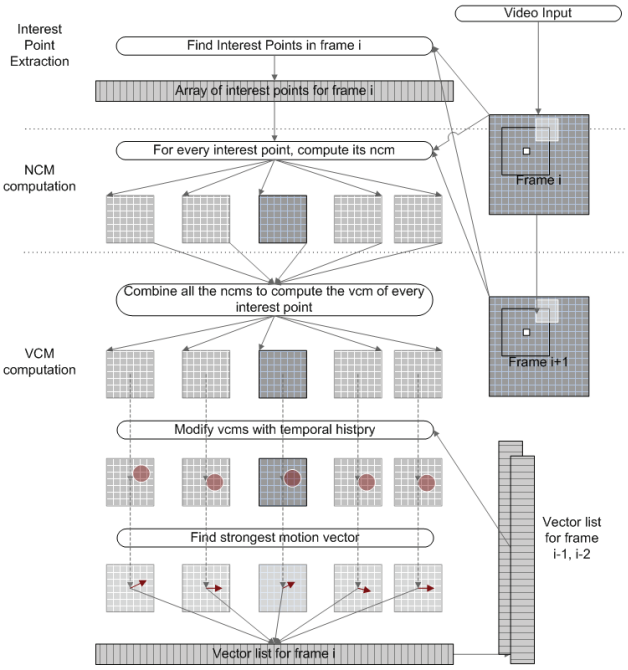

Figure 3 Data flow of the VCM Algorithm

summarized below and in Table 1.

IP extraction involves the application of convolution kernels over the input image, image differencing, and *IP* sorting and selection. These are typical image processing requirements.

NCM computation performs image correlations and produces a 2D correlation map for each *IP*. This phase also requires that the 2D correlation array be instantiated within the parallel computational module, and for the entire array to be returned to main graphics memory.

VCM computation involves the summing of values from multiple NCMs. As such, the parallel computation module must make multiple memory accesses to the *IP* list and NCMs in main graphics memory. It allocates memory for the VCM, performs array value summation and scans the resulting VCM for the 'vote maximum' and returns the 'best vector'.

Apart from the three computational phases, we also address the issues of loading the video frames to graphics memory as this is potentially quite time consuming. The computational phases and data flow are summarized in Figure 3

Table 1 Computation phases in the VCM algorithm

Phase	Computation	Input	Output
IP Extraction	Convolution, Image differencing, *IP* sorting/selection	2 images	*IP* List (O1)
NCM	Correlation	2 images, O1	1 NCM per *IP* (O2)
VCM	Accumulation, Maximum detection	O1, O2	Resulting vector

4.1.3 Approach

Because the algorithm profiles, especially memory access patterns, of the three VCM phases are different, the GPU optimization for each phase has to be implemented differently. We will discuss in the following subsections about the detail of performance enhancement issues for each phase.

We will also provide algorithm performance comparison between CPU, laptop GPU and desktop GPU, so that we can have an estimation model from performance speedup between different types of GPUs.

4.2 Early Processing

We segment the video image into 16×16 sub-windows for *IP* extraction. A set of CUDA blocks process these sub-windows until all the sub-windows are completed. Within the block, a 16×16 'result array' each processing thread is responsible for a pixel, computing the Sobel gradients, image difference, and fuzzy-And operation. The resulting *spatio-temporal* (*s-t*) *gradient* is entered into a16×16 array in shared memory. Since we keep the source video images in texture memory, most of the memory access to the images are cache hits. Once all the pixels have been processed, they are sorted to find the best *n IPs* subject to a *minimum s-t threshold* (we don't want *IP*s beneath a minimum *s-t* variance). *n* is a parameter of our system that can be tuned to ensure a good *IP* distribution. We implemented a novel sorting algorithm where a thread picks up a value in the 16×16 array and compares it sequentially with other values within the array. Once more than *n* other points with higher *s-t* variance are encountered, the thread abandons the point. Hence only up to *n* points will run to termination on their threads and be returned as detected *IP*s to global GPU memory.

4.3 NCM Computation

NCM computation does the real work of image correlation to extract correspondences. We choose 64×64 as the NCM size on the assumption that no object of interest will traverse the breadth of a typical 640×480 video frame in less than 20 frames (667 *ms*).

We segment NCM computation by *IP*. Each *IP* is assigned to a computational block. The block allocates a 64×64 convolution array for the NCM. Since we use a 5×5 correlation template, the block also allocates memory for, and reads a 68×68 sub-window from the source video frame I_{i+1}, and constructs the 5×5 correlation template from the region around the *IP* in frame I_i. The correlation is parallelized across the available computational threads where each thread computes the correlation result of its assigned pixels sequentially. The pixels are distributed evenly across the threads. The resulting NCM is returned to global GPU memory. Since reading/writing to global memory is costly (400-600 cycles per read/write, as opposed to approx 10 cycles for shared memory), we use a parallel writing mechanism known as '*coalesced access*' mode where multiple threads can access global memory in parallel in a single read/write process.

4.4 VCM Computation

VCM computation is also segmented by *IP*. Each *IP* is assigned to a block, which instantiates a 64×64 array for maintaining the VCM being accumulated. Similar to NCM computation, every thread is in charge of the accumulation results of its assigned VCM pixels. To enable caching between global and shared memory, the *IP*s were cast into texture memory and unpacked in the block. The threads read relevant NCM pixels from global memory in a coalesced manner for the reason we discussed in the last section. After the accumulation of all the VCM pixels are complete, we select the highest VCM point and return the vector starting at the *IP* being processed and ending at this point.

4.5 Data Access and Pipelining

The CPU is capable of running concurrently with the GPU. The overall running time can be reduced if expensive CPU operations are run during GPU kernel execution. In our implementation, disk access requires the most CPU time. The best opportunity to load the next video frame is during the VCM phase. This phase requires only the NCMs from the previous phase, not the video frames. Therefore, modifying a video frame during calculation will not affect the result. Also, the VCM phase requires the most time, which leaves the CPU idle longer than any other phase.

On one of our test computers, loading a single frame of video requires 0.0521 seconds. When frame loading occurs sequentially after the VCM phase is complete, the average overall running time per frame was 0.177 seconds. When frame loading occurs concurrently with the VCM phase, the average loading time decreases to 0.125 seconds. This shows that using the GPU allows opportunities for concurrent execution, resulting in a significant speedup.

5. Results

To better understand the speedup offered by CUDA-based GPU programs, we separated the analysis of VCM into the three computational phases. We provide comparisons between GPU computation of each phase against CPU-based computation. To understand how different classes of GPUs may enhance computation speed, we also provide this three-phase analysis between two GPUs.

Both GPUs in our tests are in the NVIDIA G80 family. This gives us a sense of what different clock rates and hardware resources (like number of multiprocessors) affect computation. One is an NVIDIA 8600MGT, which is a typical configuration for a laptop computer. Another is an NVIDIA 8800 GTS-512 (512M memory version), which is a standard configuration for a desktop computer. By comparing the performance of VCM algorithm between these two different GPUs with different number of processing cores, we can find how each computation phases scales with respect to the degree of data parallelism.

5.1 Computation Results

Before we discuss the performance enhancement for GPU implementation of VCM algorithm, we verify the vector

tracking results. Figure 4 shows the VCM results on four different videos computed using our CUDA code. The computed vectors are rendered on top of the respective images. At

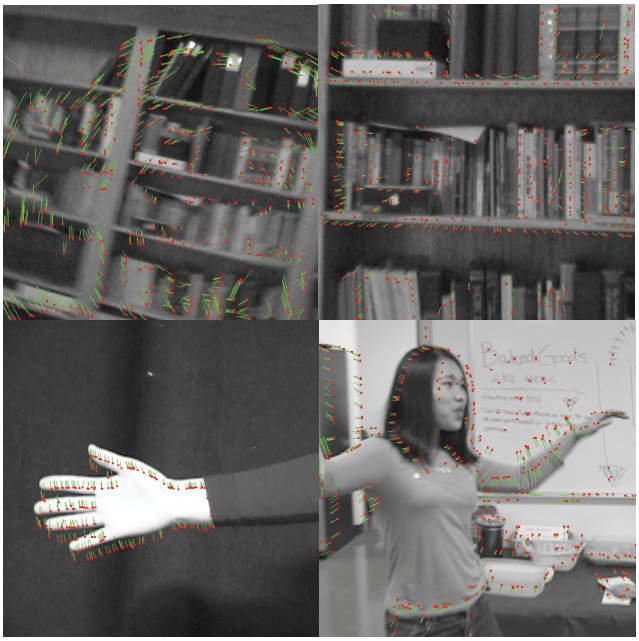

Figure 4 VCM Processing Results with GPU implementation. Top-Left: Camera rotating on its optical axis; Top-Right: Camera zooming out; Bottom-Left: Hand moving up; Bottom-Right: Girl dancing with camera zooming in

the top-left, the camera is rotated on its optical axis. The light was low, and there was a fair amount of motion blur. VCM was able to extract the rotating flow field. At the top-right, the camera was zooming out. Again, the light was low and the video was grainy. Since zooming is slower than the earlier rotation, this video tested VCM's ability to compute very small vectors that converged at the scene center. At the bottom-left, VCM tracked a video of a moving hand. This demonstrates the algorithm's ability to function with localized *IP*s (i.e. we test the algorithm's *IP* distribution strategy). At the bottom-right, the subject is dancing in a zooming video sequence. VCM correctly extracted both the motion vectors on the subject and the zooming vectors elsewhere.

5.2 Performance Results Compared with CPUs

To evaluate effectiveness of our approach, we used two different implementations of VCM algorithm; one is based on GPU and the other is based on CPU. The GPU version was run on two different GPUs: a 8600MGT (laptop GPU of the Apple MacBookPro), and a 8800GTS-512. The algorithm was run with 2048 *IP*s. The CPU is a 3.2 GHz Pentium 4 running on Windows XP. Our results show considerable speed increases of up to 45 times for the 8800GTS over the CPU. Figure 5 (a) and (b) show the time and speedup comparisons of our three configurations respectively. For the time comparison, we had to multiply the 8600MGT time by 5 and the 8800GTS time by 25 to make the bars visible on the chart. The times are shown in seconds (hence the 8800GTS total

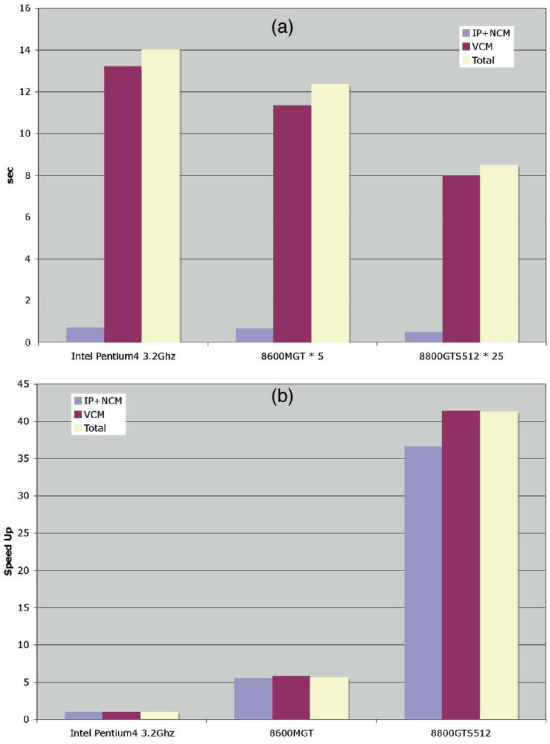

Figure 5 Comparison between GPU and CPU operation. (a) Time Comparison: For visibility the 8600MGT time was multiplied by 5, and the 8800GTS-512 time was multiplied by 25, (b) Speed Comparison: The speed of the 3.2 GHz Intel Pentium 4 was used as the base comparison at 1.0

time was approximately 8/25 ≈ 0.34 s as compared to the 14.55 s required by the Pentium). The speedup graph was plotted with the CPU speed set at 1.0. The 8600MGT was 5.67 times faster and the 8800GTS was 41.31 times faster.

Hence, GPUs offer significant speedup over the CPU, and there is significant advantage of the faster GPU over the slower one.

5.3 Comparison Among Different GPUs

There are a variety of cards available from NVIDIA that are capable of running CUDA. We tested four cards that have different properties, such as number of cores, core clock speed, memory clock speed, etc. It is difficult to predict the performance of a particular card because different algorithms have different computational demand, memory bandwidth demands, and memory access patterns.

As shown in Figure 6, the total computation time of the algorithm presented varies linearly with the number of *IP*s.

The *IP* extraction phase also varies linearly with the number of *IP*s. The heavier computation is in the convolution process that must be performed irrespective of number of *IP*s. The variation arises from the sorting of the *spatio-temporal* gradient points and in the memory access to store the output to global GPU memory. In all *IP* computation time is dwarfed by the *NCM* and *VCM* computation time.

The NCM phase computation time is linear with the number of *IP*s detected. One NCM is computed per *IP*. Other parame-

ters could affect the NCM computation time, such as the size of the NCM and the size of the correlation kernel.

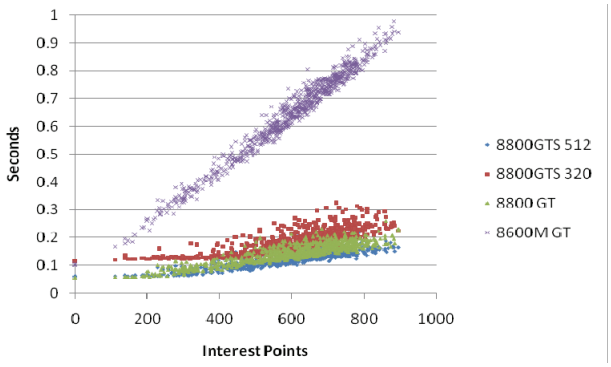

Figure 6 Total computation time vs. Number of IPs

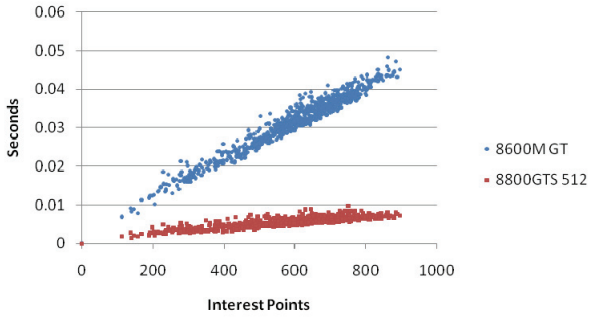

Figure 7 NCM phase computation time vs Number of IPs

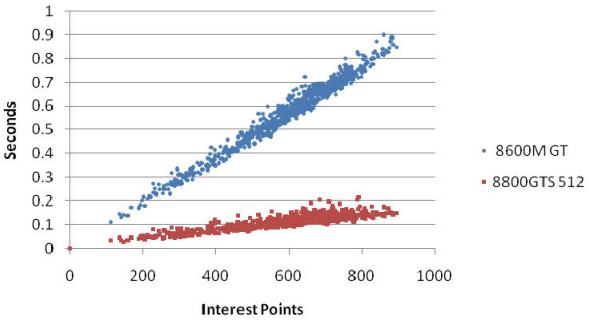

Figure 8 VCM phase computation time vs. Number of IPs

The VCM phase is also linear with respect to the number of *IP*s. The VCM will be affected more by the parameters set before the algorithm is run, such as the weighting function, interest points per subwindow, and the size of the subwindows. Changes to all of the aforementioned parameters could increase the VCM phase computation non-linearly.

6. Conclusion and Future Work

We have presented a CUDA-based GPU implementation of the parallel VCM algorithm. We show that common desktop graphics hardware can accelerate the algorithm more than 41 times over a state-of-the-art CPU. To achieve such a performance gain, care has to be taken in optimization to map computation to the GPU architecture. Since VCM is a more computationally and data intensive algorithm, we believe that we have demonstrated the viability of applying GPUs to general computer vision processing. We have also shown that the CUDA programming framework is amenable to coding vision algorithms. We expect to test this premises against a broader array of computer vision algorithms, and to test more optimization strategies to gain insights on how these affect computation efficiency.

Acknowledgements

This research has been partially supported by NSF grants "Embodied Communication: Vivid Interaction with History and Literature," IIS-0624701, "Interacting with the Embodied Mind," CRI-0551610, and "Embodiment Awareness, Mathematics Discourse and the Blind," NSF-IIS- 0451843.

References

[1] Bryll, R. and F. Quek, Fusing Vector Magnitudes and Cluster Centroids for Extended Duration Vector-Based Gesture Tracking, in CVPR 2001.

[2] De Neve, W., et al., GPU-assisted decoding of video samples represented in the YCoCg-R color space, in Proc. ACM Int. Conf. on Multimedia. 2005.

[3] Horn, B.K.P. and B.G. Schunck, Determining optical flow. Art. Intel., 1981. 17: p. 185--204.

[4] Mizukami, Y. and K. Tadamura. Optical Flow Computation on Compute Unified Device Architecture. in Image Analysis and Processing, ICIAP 2007.

[5] NVIDIA, CUDA Programming Guide Version 1.1. 2007, NVIDIA Corporation: Santa Clara, California.

[6] Owens, J., Streaming architectures and technology trends, in ACM SIGGRAPH 2005 Courses. 2005, ACM: Los Angeles, California.

[7] Quek, F. and R. Bryll. Vector Coherence Mapping: A Parallelizable Approach to Image Flow Computation. in ACCV 1998.

[8] Quek, F., X. Ma, and R. Bryll. A parallel algorithm for dynamic gesture tracking. in ICCV'99 Wksp on RATFG-RTS. 1999.

[9] Quek, F., et al., Gesture, speech, and gaze cues for discourse segmentation, in CVPR. 2000: p. 247-254.

[10] Quek, F., et al., Multimodal Human Discourse: Gesture and Speech. ACM ToCHI, 2002. 9(3): p. 171-193.

[11] Ready, J.M. and C.N. Taylor. GPU Acceleration of Real-time Feature Based Algorithms. in Motion and Video Computing, 2007.

[12] Xiong, Y. and F. Quek, Hand Motion Oscillatory Gestures and Multimodal Discourse Analysis. International Journal of HCI, 2006. 21(3): p. 285-312.