Low-Cost, High-Speed Computer Vision Using NVIDIA's CUDA Architecture

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

Center of Human Computer Interaction,

Virginia Polytechnic Institute and University
Blacksburg, VA 24060, USA

[†]quek@vt.edu

Abstract— In this paper, we introduce real time image processing techniques using modern programmable Graphic Processing Units (GPU). GPUs are SIMD (Single Instruction, Multiple Data) device that is inherently data-parallel. By utilizing NVIDIA's new GPU programming framework, "Compute Unified Device Architecture" (CUDA) as a computational resource, we realize significant acceleration in image processing algorithm computations. We show that a range of computer vision algorithms map readily to CUDA with significant performance gains. Specifically, we demonstrate the efficiency of our approach by a parallelization and optimization of Canny's edge detection algorithm, and applying it to a computation and data-intensive video motion tracking algorithm known as "Vector Coherence Mapping" (VCM). Our results show the promise of using such common low-cost processors for intensive computer vision tasks.

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

1 Introduction

Image processing is important to such fields as computer vision and human computer interaction. For example, real-time motion tracking is required for applications such as autonomous vehicle control and gesture interaction. However, a long standing challenge to the field of image processing is that intensive computational power is required in order to achieve higher speed. Real-time image processing on video frames is difficult to attain even with the most powerful modern CPUs. Increasing resolution of video capture devices and increased requirement for accuracy make it is harder to realize real-time performance.

Recently, graphic processing units have evolved into an extremely powerful computational resource. For example, The NVIDIA GeForce GTX 280 is built on a 65nm process, with 240 processing cores running at 602 MHz, and 1GB of GDDR3 memory at 1.1GHz running through a 512-bit memory bus. Its processing power is theoretically 933 GFLOPS [1], billions of floating-point operations per second, in other words. As a comparison, the quad-core 3GHz Intel Xeon CPU operates roughly 96 GFLOPS [2]. The annual computation growth rate of GPUs is approximately up to 2.3x. In contrast to this, that of CPUs is 1.4x [3]. At the same time, GPU is becoming cheaper and cheaper.

As a result, there is strong desire to use GPUs as alternative computational platforms for acceleration of computational intensive tasks beyond the domain of graphics applications. To support this trend of GPGPU (General-Purpose Computing on GPUs) computation, graphics card vendors have provided programmable GPUs and high-level languages to allow developers to generate GPU-based applications.

This research aimed at accomplishing efficient and cheaper application of image processing for intensive computation using GPUs. We show that a range of computer vision algorithms map readily to CUDA by using edge detection and motion tracking algorithm as our example application. The remainder of the paper is organized as follows. In Section 2, we describe the recent advances in GPU hardware and programming framework, discuss previous efforts on application acceleration using CUDA framework, and the use of GPUs in computer vision applications. In Section 3, we detail the implementation of the Canny algorithm and various design and optimization choices we made. Then we demonstrate the efficiency of our approach by applying it to a data-intensive motion tracking algorithm, Vector Coherence Mapping, in Section 4. Finally we present our conclusion and future work in Section 5.

2 Background

2.1 The NVIDIA CUDA Programming Framework

Traditionally, general-purpose GPU programming was accomplished by using a shader-based framework [4]. The shader-based framework has several disadvantages. This framework has a steep learning curve that requires in-depth knowledge of specific rendering pipelines and graphics programming. Algorithms have to be mapped into vertex transformations or pixel illuminations. Data have to be cast into texture maps and operated on like they are texture data. Because shader-based programming was originally intended for graphics processing, there is little programming support for control over data flow; and, unlike a CPU program, a shader-based program cannot have random memory access for writing data. There are limitations on the number of branches and loops a program can have. All of these limitations hindered the use of the GPU for general-purpose computing. NVIDIA released CUDA, a new GPU programming model, to assist developers in general-purpose computing in 2007 [5]. In the CUDA programming framework, the GPU is viewed as a compute device that is a co-processor to the CPU. The GPU has its own DRAM, referred to as *device memory*, and execute a very high number of threads in parallel. More precisely, data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads. In order to organize threads running in parallel on the GPU, CUDA organizes them into logical blocks. Each block is mapped onto a multiprocessor in the GPU. All the threads in one block can be synchronized together and communicate with each other. Because there is a limited number of threads that a block can contain, these blocks are further organized into grids allowing for a larger number of threads to run concurrently as illustrated in Figure 1. Threads in different blocks cannot be synchronized, nor can they communicate even if they are in the same grid. All the threads in the same grid run the same GPU code.

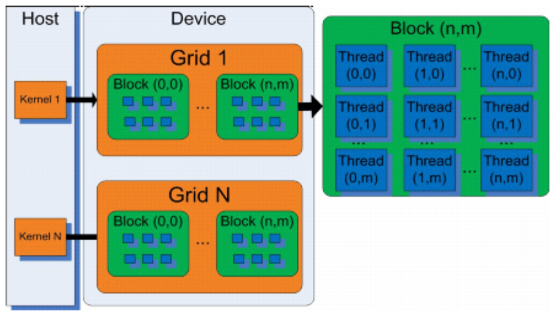


Figure 1.Thread and Block Structure of CUDA

CUDA has several advantages over the shader-based model. Because CUDA is an extension of C, there is no longer a need to understand shader-based graphics APIs. This reduces the learning curve for most of C/C++ programmers. CUDA also supports the use of memory pointers, which enables random memory-read and write-access ability. In addition, the CUDA framework provides a controllable memory hierarchy which allows the program to access the cache (shared memory) between GPU processing cores and GPU global memory. As an example, the architecture of the GeForce 8 Series, the eighth generation of NVIDIA's graphics cards, based on CUDA is shown in Figure 2.

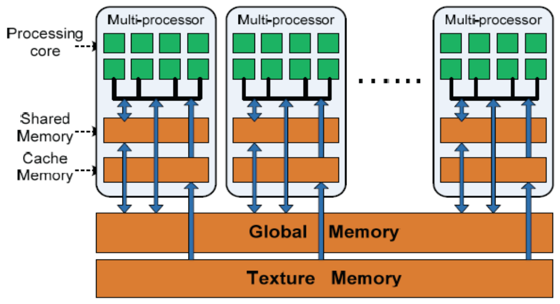


Figure 2. GeForce 8 series GPU architecture

The GeForce 8 GPU is a collection of multiprocessors, each of which has 16 SIMD (Single Instruction, Multiple Data) processing cores. The SIMD processor architecture allows each processor in a multiprocessor to run the same instruction on different data, making it ideal for data-parallel computing. Each multiprocessor has a set of 32-bit registers per processors, 16KB of shared memory, 8KB of read-only constant cache, and 8KB of read-only texture cache. As depicted in Figure 2, shared memory and cache memory are on-chip. The global memory and texture memory that can be read from or written to by the CPU are also in the regions of *device memory*. The global and texture memory spaces are persistent across all the multiprocessors.

2.2 GPU computation in image processing

Most computer vision and image processing tasks perform the same computations on a number of pixels, which is a typical data-parallel operation. Thus, they can take advantage of SIMD architectures and be parallelized effectively on GPU. Several applications of GPU technology for vision have already been reported in the literature.

De Neve et al. [6] implemented the inverse YCoCg-R color transform by making use of pixel shader. To track a finger with a geometric template, Ohmer et al. [7] constructed gradient vector field computation and Canny edge extraction on a shader-based GPU which is capable of 30 frames per second performance. Sinha et al. [8] constructed a GPU-based Kanade-Lucas-Tomasi feature tracker maintaining 1000 tracked features on 800x600 pixel images about 40 *ms* on NVIDIA GPUs. Although all these applications show real-time performance at intensive image processing calculations, they do not scale well on newer generation of graphics hardware including NVIDIA' CUDA. Mizukami and Tadamura [9] proposed implementation of Horn and Schunck's [10] regularization method for optical flow computation based on the CUDA programming framework. Making use of on-chip shared memory, they were able to get approximately 2.25 times speed up using a NVIDIA GeForce 8800GTX card over a 3.2-GHz CPU on Windows XP.

3 Canny Edge Detection

3.1 Problem Decomposition

In the context of GPU computation, Canny algorithm consists of set of convolution operation which is just the scalar product of the filter weights with the input pixels and thresholding operation. These operations map well to computation on GPUs because it follows the SIMD paradigm. As shown on Figure 3, a series of steps must be followed to implement the Canny edge detector. The first step is applying 5x5 Gaussian blur filter to the original image to filter out noise, then 3x3 step edge operator is used to determine edge gradient and direction. In the third stage, non-maximum suppression is carried out to find a local maximum in the gradient direction to generate thin edges in the image. Finally, thresholding with hysteresis is processed to determine continuous edges without streaking [11].

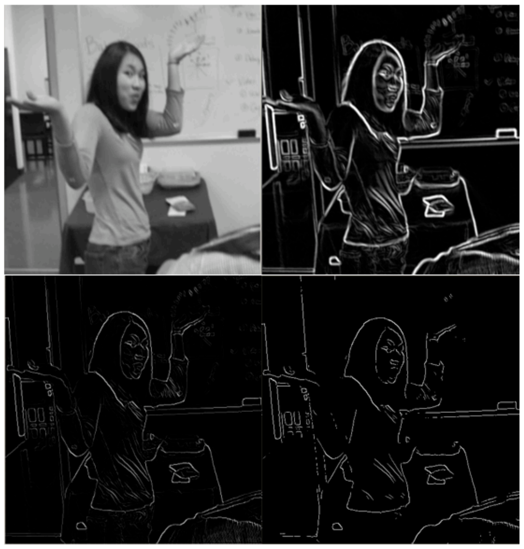


Figure 3. Four steps on Canny algorithm computation

Top-Left: Image with Gaussian filter*; Top-Right:* Image with Sobel filter*; Bottom-Left:* Image with non- maximum suppression*; Bottom-Right:* image with Hysteresis

3.2 GPU implementation Details

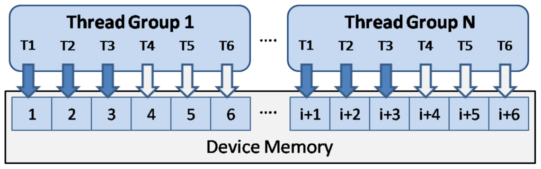
To characterize the effects of various CUDA-based programming optimization strategies for image processing algorithm, we implement Canny algorithm using different memory regions and access patterns.

Texture memory The result of each step is an image itself that can readily be mapped to texture for the use in next phase of computation. The texture memory space are cached, a texture fetch costs just one texture cache read on a cache hit, otherwise costs one *device memory* read on a cache miss. Since neighboring pixel data are required to work on a pixel, convolution operations are of high reference of locality in terms of memory access. This means that GPUs are able to exploit texture memory for high performance in convolution operations. Another advantageous point of using texture memory is that the hardware provides automatic handling of boundary cases of the image. When the referenced coordinate of texture is out of valid range, the hardware clamps the coordinate to the valid range or wraps to the valid range depending on the addressing mode set.

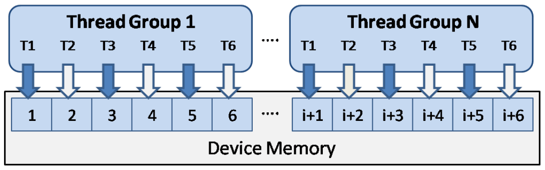
Kernel code is not able to write data into texture memory though it can read from texture memory. It means execution should be switched between CPU code and GPU kernel code repeatedly whenever it generates a texture with the calculation results at each step. Therefore four times data transferring between host (CPU) and device accompanies in the implementation using texture memory. This is costly because the bandwidth between the device memory and the host memory (CPU's own DRAM) is much lower compared to the bandwidth between the device and the device memory. The transfer of data between the host and the device is limited by the bus; for most setups, this is limited by the PCI-Express x16 maximum theoretical bandwidth of 4GB/s. Memory transfer speeds within the device has a theoretical maximum of 86GB/s [5].

Global memory Device memory reads through texture fetching present several benefits over reads from global or constant memory; 1. Texture memory are cached, potentially exhibiting higher bandwidth if there is high locality in the texture fetches. 2. The method of addressing is much easier because they are not subject to the constraints on memory access pattern that global or constant memory reads have to obey to get good performance. 3. The latency of addressing calculations is better hidden. However, performance trade-offs are not yet fully understood, especially in the case; switching between device and host in order to write data into texture memory versus no switching between device and host but using global memory for manipulating data.

In this approach, a video image is loaded into a global memory instead of texture memory, a region of the image block is multiplied pixel-wise with the convolution filter, the results are summed, and then output pixel is written back to global memory for next step. Since whole image is loaded into the global memory, we only need to handle boundary cases of when a pixel is at the edge of the image. Then the video image is segmented into 16×16 sub-windows and a set of CUDA blocks process these sub-windows for convolution operations until all the sub-windows are completed. However, memory access patterns should be carefully chosen to get good performance. In order to hide the high latency to access global memory, the access pattern for all threads should follow a coalesced fashion as depicted in Figure 4.



(a) Example of a coalesced memory access pattern



(b) Example of a non-coalesced memory access pattern
Figure 4. Memory access pattern constraint

Shared memory Shared memory access is faster than global memory access since it is on-chip. To explore how much of performance gain we can achieve by using shared memory, we implement Canny algorithm with shared memory. A block of pixels from the image is loaded into an array in shared memory, convolution or thresholding operation is applied to the pixel block, then the output image is written into shared memory array for the use on the next step. In this approach, the pixels at the edge of the shared memory array will depend on pixels not in shared memory. Therefore, padding pixels of the mask radius is required around the image block within a thread block as shown on Figure 5.

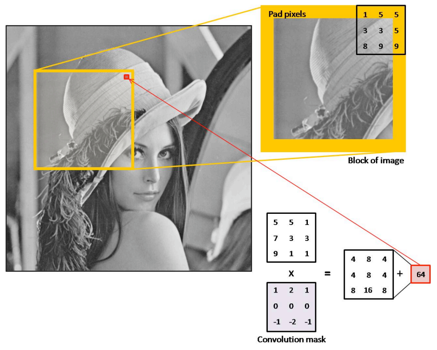


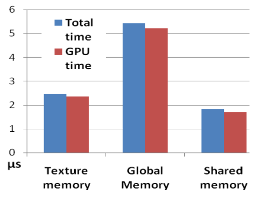
Figure 5. Image block with pad pixels

In order to achieve high memory throughput, shared memory is divided into successive 16 memory modules of 32-bit words, called banks, which can be serviced in a single memory transaction. If all threads within a half warp (16 threads) simultaneously access consecutive words then single large read of the 16 values can be performed at maximum speed, or if memory requests from several threads fall in the same memory bank, performance will be much lower. It is therefore important to have threads access a single 32-bit word at a time, with each word being assigned to one thread.

3.3 Optimization Strategies

Memory Space We explored different memory space for source image location and intermediate calculation results storage in order to propose optimization strategies. In all cases of implementation, the kernel configuration is of 16×16 threads of each block and 32 of blocks on 512x512 pixel image. The convolution is parallelized across the available computational threads where each thread computes the convolution result of its assigned pixels sequentially. Pixels are distributed evenly across the threads. The result is shown on Figure 6. On 8800GTS-512, the average processing time per frame is 1.818ms with shared memory,. It takes 2.344ms per frame to perform Canny algorithm with texture memory. Performance time per frame is 5.433ms in average using global memory for data manipulation without execution switching between CPU code and GPU kernel code.

Threads Configuration Number of blocks each multiprocessor can process depends on how many registers per thread and how much shared memory per block is required for a given kernel. Since shared memory is not used in the implementation with texture memory, we only need to be concerned about the number of registers used and we can maximize the size of block and grid as much as possible. With 348 threads per block and 512 blocks per image, we could enhance the performance up to 1.978ms per frame in average. This demonstrates that when the data come from global memory, high-latency memory access cost can be hidden by increasing the number of concurrently running blocks.



	Total	GPU time	%GPU time
Texture memory	2.459	2.344	95.34
Global Memory	5.433	5.225	96.17
Shared memory	1.818	1.706	93.87

Figure 6. Total execution time vs. GPU time

Switching Between CPU and GPU To further explore on performance trade-offs, we also implemented the algorithm separated as two kernel codes with global memory, and the average calculation time is 6.676ms per frame. Switching between CPU and GPU can severely harm the performance.

4 Vector Coherence Mapping

4.1 Vector Coherence Mapping

VCM algorithm is for the computation of an optical flow field from a video image sequence first introduced by Quek et al [12][13]. By applying spatial and temporal coherence constraints with fuzzy image processing operation, it tracks sets of interest points in the frame in parallel. A voting scheme is featured to enforce the constraints on vector fields.

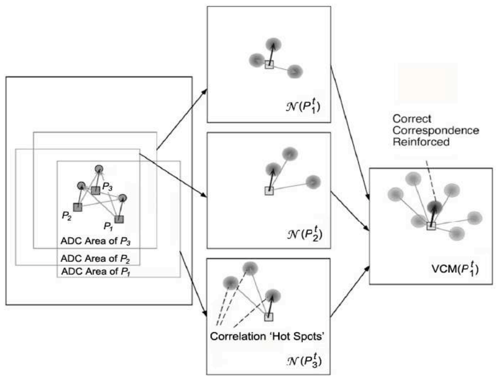


Figure 7: Algorithm illustration for Vector Coherence Mapping

Figure 7 describes how VCM incorporates spatial coherence constraints with the correlation for tracking interest points in a flow field. Three detected points are shown at the left of the picture where the shaded squares represent the position of the three interest points at time *t* and the shaded circles represent the corresponding position of those points at *t* + *1*. Correlation matching results for each point, which is labeled $N(P_1^t)$, $N(P_2^t)$ and $N(P_i^t)$, provide three hotspots as shown at middle of picture. (The correlation matching result is called

Normal Correlation Map (NCM).) By using weighted summation of these neighboring NCMs, we can obtain Vector Coherence Map of point P_i' with minimizing the local variance of the vector field as shown at the right of figure. VCM algorithm is inherently parallel since it employs convolution and pixel-wise summation as its dominant operations.

4.2 Problem Decomposition

Computational processing in VCM can be decomposed to three phases by their data dependencies on calculation and pattern of computation: 1. Interest Point (IP) extraction, 2. Normal Correlation Map (NCM) computation, and 3. Vector Coherence Map (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 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 Table 1.

Table 1. Computation phases in the VCM algorithm

Phase	Computation	Input	Output
IP Extraction	Convolution Image differencing IP sorting/selection	2 images	IP list
NCM	Correlation	2 images, IP list	1 NCM per IP
VCM	Accumulation Maximum detection	IP list, 1 NCM per IP	Resulting vector

4.3 GPU Implementation Detail

4.3.1 Interest Point Extraction

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.

4.3.2 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.

4.3.3 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.

4.3.4 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.

4.3.5 Results

In this section we present the results of video stream analysis with GPU implementation of VCM algorithm. Figure 8 shows the VCM results on four different videos computed using our CUDA code. The computed vectors are rendered on top of the respective image. At the top-left, the camera is rotated on its optical axis. Even fair amount of motion is blurred, camera movement detected precisely. The top-right shows the results of zoom-out video stream. The result shows VCM's ability to track the motion exactly though image is grainy and movement is very subtle (Small vectors that converges at the center of scene). At the bottom-left, the

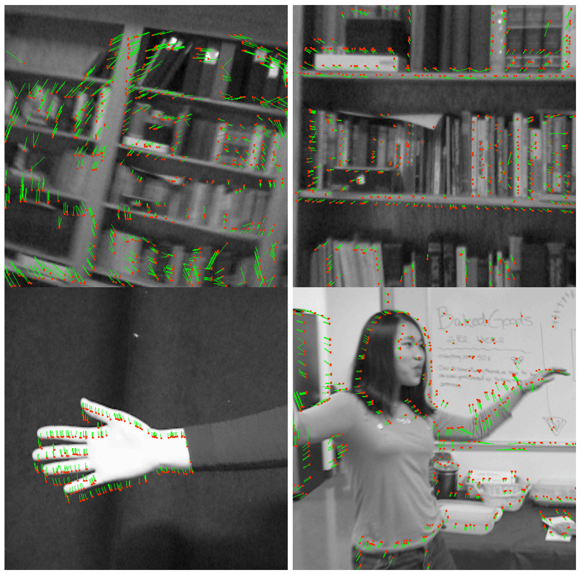


Figure 8. VCM Processing Results with GPU implementation.

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

sequence of hand moving is analyzed. The subject is dancing while camera is zooming in the subject at the bottom-right. VCM correctly extracted both the motion vectors on the subject and the zooming vectors elsewhere.

To evaluate effectiveness of our approach for utilizing GPUs as massive data-parallel processor, we used two different implementation of VCM algorithm; one supports utilizing GPU and the other is based on CPU. For the comparison, two different GPUs executed the GPU version code: a 8600MGT which is equipped on Apple MacBookPro, and a 8800GTS-512. The CPU version is executed on 2.4 GHz Inter Core 2 with Windows XP. The algorithm is implemented in the similar way as much as possible; both of them used the same data structure for IP, NCM, and VCM. The same number and size of sub-windows and the same size

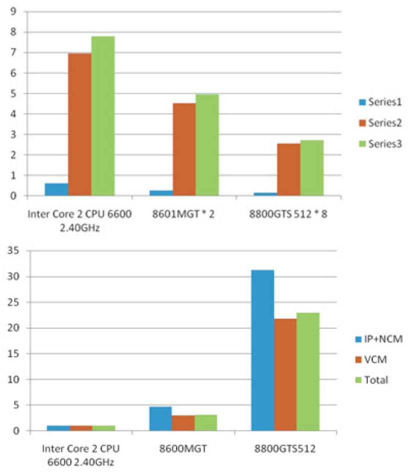


Figure 9. (a) Time Comparison between GPU and CPU operation: The 8600 MGT tie was multiplied by 2, and the 8800 GTS-512 was multiplied by 8 for visibility. (b) Speed Comparison between GPU and CPU operation: The speed of CPU was set at 1.0 for to be base comparison

of IP list, NCM, and VCM were used.

Graph 9(a) and 9(b) show experiment results with 2048 IPs. In the time graph, 8600MGT operation time had to be multiplied by 2 and 8800GTS-512 operation had to be multiplied 8 for visibility. The speed of CPU operation was used as the base comparison in the speed up graph. 8600MGT showed 3.15 times speed enhancement, and 8800GTS-512 showed 22.96 times performance enhancement.

5 Conclusion and future work

In this paper, we analyze fundamental image processing operations: Canny Edge detection, with respect to the CUDA enabled GPU implementation. We identify general GPU optimization strategies, memory latency hiding and CPU/GPU switching avoidance. Based on the experiment results, we present a CUDA-based GPU implementation of the parallel VCM algorithm. We show that common desktop graphics hardware can accelerate the algorithm more than 22 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] NVIDIA, CUDA Programming Guide Version 2.0. 2008, NVIDIA Corporation: Santa Clara, California.

[2] Intel, Quad-Core Intel® Xeon® Processor 5400 Series. 2008, Intel Corporation: Santa Clara, California.

[3] Fan, Z., Qiu, F., Kaufman, A., and Yoakum-Stover, S. GPU Cluster for High Performance Computing. in Proc. of the 2004 ACM/IEEE Conf. on Supercomputing, 2004.

[4] Allard, J. and Raffin, B., A shader-based parallel rendering framework. in Visualization, 2005. VIS 05. IEEE, (2005), 127-134.

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

[6] 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.

[7] Ohmer, J. F., Maire, F., and Brown, R. 2006. Real-Time Tracking with Non-Rigid Geometric Templates Using the GPU. In Proc. of the Int. Conf. on Computer Graphics, Imaging and Visualisation (July 26 - 28, 2006). CGIV. IEEE Computer Society, Washington, DC, 200-206.

[8] Sinha, S.N., Frahm J.-M., Pollefeys M., and Genc Y. Feature tracking and matching in video using programmable graphics hardware. Machine Vision and Applications (MVA), 2007.

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

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

[11] Canny, J., A Computational Approach To Edge Detection, IEEE Trans. Pattern Analysis and Machine Intelligence, 8:679-714, 1986.

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

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