

Using Multiple Graphics Cards as a General Purpose Parallel Computer : Applications to Computer Vision

James Fung and Steve Mann*

University of Toronto

Department of Electrical and Computer Engineering

10 King's College Road, Toronto, Ontario, Canada

{fungja,mann}@eecg.toronto.edu

Abstract

Pattern recognition and computer vision tasks are computationally intensive, repetitive, and often exceed the capabilities of the CPU, leaving little time for higher level tasks. We present a novel computer architecture which uses multiple, commodity computer graphics devices to perform pattern recognition and computer vision tasks many times faster than the CPU. This is a parallel computing architecture that is quickly and easily constructed from readily available hardware. It is based on parallel processing done on multiple Graphics Processing Units (GPUs). An eigenspace image recognition approach is implemented on this parallel graphics architecture. This paper discusses methods of mapping computer vision algorithms to run efficiently on multiple graphics devices to maximally utilize the underlying graphics hardware. The additional memory and memory bandwidth provided by the graphics hardware provided for significant speedup of the eigenspace approach. We show that graphics devices parallelize well and provide significant speedup over a CPU implementation, providing an immediately constructible low cost architecture well suited for pattern recognition and computer vision.

1. Introduction

In this paper, we present results from a system which uses multiple computer graphics cards to operate as a parallel computer architecture which carries out pattern recognition and computer vision computations much faster than any single processor system. Many tasks in pattern recognition and computer vision are computationally intensive and repetitive (and therefore good candidates for parallel architecture). The requirements of pattern recognition and com-

*Thanks to NSERC, SSHRC, Canada Council for the Arts, Ontario Arts Council, Toronto Arts Council, and Ontario Graduate Scholarships/Lewfam Foundation Scholarships in Science and Technology agency for support. Thanks to nVIDIA, ATI, and Viewcast for equipment donations.

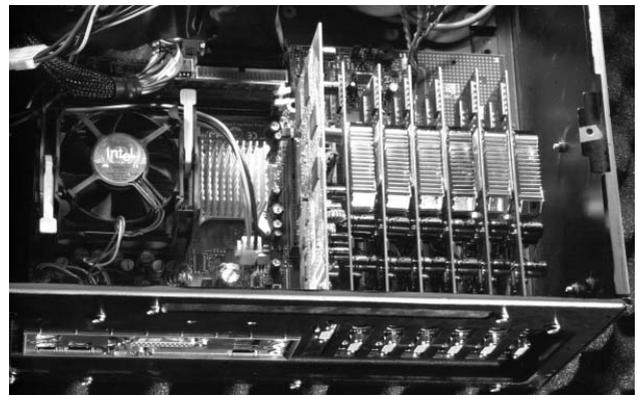


Figure 1: A computer vision machine with 6 PCI graphics cards, and 1 AGP graphics card. Each PCI cards has a GeForce FX 5200 GPU which runs pattern recognition and computer vision tasks in parallel, creating a cheap, powerful, and easily constructible parallel architecture well suited for pattern recognition and computer vision.

puter vision often exceed the CPUs capabilities. Moreover, freeing up the processor from the pattern recognition tasks allows it to perform other, higher level tasks.

Modern computers now incorporate a Graphics Processing Unit (GPU), which carries out the computation required for graphics applications efficiently.

In fact, graphics cards themselves are now becoming a significant part of the overall compute power of any modern system. For instance, modern GPUs now have more transistors than modern CPUs. Consider that the Intel Xeon processor has 108 million transistors, the Radeon R300 GPU has 110 million transistors, and the GeForce FX GPU has 125 million transistors. Furthermore, the transistors on the Xeon are largely (two thirds) used for implementing cache memory, whereas the majority of the graphics cards transistors are used in the implementation of multiple floating point SIMD units. Additionally, the graphics cards still boast a very high memory access rate using DDR and DDR2 memory.

It has often been said that computer graphics and com-

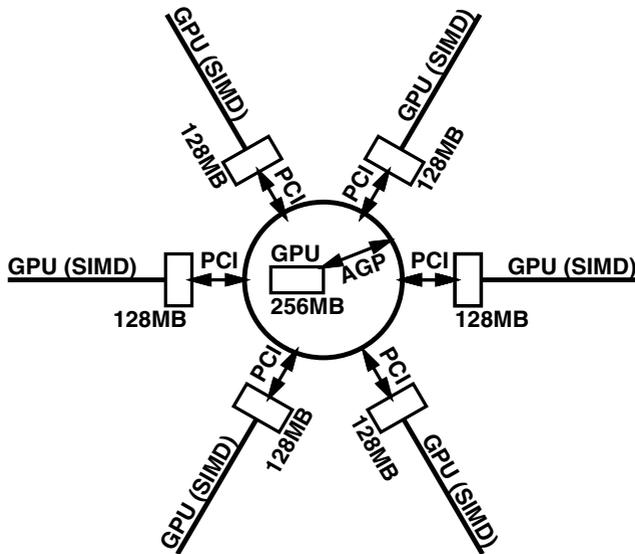


Figure 2: Parallel GPU Architecture. 6 PCI graphics cards communicate with a faster AGP card via the PCI and AGP buses.

puter vision are inverses of one another, since computer graphics turns a numerical representation of a scene into an output image, while computer vision takes an image and produces a numerical representation of it. Thus, carrying out computer vision tasks in graphics hardware uses the graphics hardware in an “inverse” fashion. It has been shown that graphics hardware is also capable of efficiently running computer vision algorithms [2, 6], as well as other general purpose computations¹. Much work has been done in graphics architectures themselves. Our interest here is to examine the system performance of these architectures for parallel computer vision.

2. Graphics for Vision Architecture

The architecture we propose is composed of a plurality of graphics devices. Most graphics devices come in the form of peripheral cards, which makes them easily installable, removable. Peripheral cards use the same bus, and are designed to be easily inter-operable. We exploit this interoperability by adding a number graphics cards to a single bus. This allows us to easily and quickly create a parallel architecture which uses multiple graphics cards for processing. We select a computer which uses six PCI slots. 6 PCI slots is the most number of PCI slots available on most common, commodity motherboards. This allows for “same day” architecting since the components are relatively low cost and quick to obtain, and this is considered a strength of the architecture. While graphics devices are available on the AGP bus, AGP is typically used for graphics, and hence not more than one AGP slot is found on typical motherboards. PCI, on the other hand is intended for general purpose use

¹see <http://www.gpgpu.org> for other resources

and allows us the flexibility of putting multiple cards on the PCI bus.

The architecture is constructed by placing graphics cards on the available PCI and AGP slots. Our implementation consists of one AGP GeForce GPU on the AGP bus, and six GeForce FX 5200 GPUs on the PCI bus. These GeForce FX GPUs provide 128 bit wide computation in 4 32 bit elements and can operate on and produce standard IEEE 32-bit floating point arithmetic. The GeForce FX 5200 GPUs were chosen because they are the only GeForce FX class GPUs which are available on PCI graphics cards. The PCI bus allows for multiple graphics cards to be used simultaneously. The parallel architecture presented here is created by commercial, off the shelf components making it easily constructible. Our focus is on the parallelism provided by these additional PCI cards.

Graphics cards hardware have features which help parallelism. Firstly, each graphics card has a Graphics Processing Unit (GPU). Each GPU contains a multiple number of pixel pipelines which process data in parallel (four in our case). These pixel pipelines are each SIMD [3] processing elements, carrying out operations typically on four colour components in parallel. We have six GPUs, each of which provides a set of SIMD parallel processing pipelines to the architecture.

The GPUs are capable of executing locally stored programs independent of the CPU supervision. Furthermore, they have each their own memory, so parallel graphics cards do not need to contend with each other for access to a shared memory. This is another advantage of the architecture over a single CPU since the memory access becomes a significant bottleneck to performance when locality of reference cannot be exploited. For the image recognition task we will demonstrate, the system operates on new data being compared to a large amount stored data. What is needed is a high throughput memory architecture. This can be achieved by breaking the data down into separate parts to be accessed in parallel. The ability of the GPUs to each access their own memories in parallel with each other increases the overall memory bandwidth of the system, providing greater throughput. In the system, each of the 6 PCI graphics cards as 128 MB of RAM (clocked at 405 MHz, with a 128-bit data path), which gives an additional 768 MB of RAM, all of which is dedicated for running the pattern recognition algorithm, and can be accessed in parallel.

The code developed for this work is freely available at <http://openvidia.org> and <http://eyetap.org>.

3. Eigenspace Image Recognition

We implement an eigenspace image recognition [5, 1] approach on the parallel graphics architecture. Given a set of p input images of size $n \times m$, we express them as a set of p column vectors, each of size $nm \times 1$, which form a $nm \times p$ matrix, A . A Singular Value Decomposition (SVD)

expresses this matrix as $A = U\Sigma V^\dagger$ where U is an orthogonal matrix of principle components, Σ is a matrix with singular values along the diagonal, and V^\dagger hold coefficients for expanding A in terms of columns of U . \dagger denotes a complex conjugate.

A new input image, e can be expressed as a linear combination of the orthogonal column vectors U_k of U . This approximation of e is denoted \hat{e} , and formed as $\hat{e} = \sum_{k=0}^p c_k U_k$ where c_k is a coefficient determined by projecting, via an inner product, the input image e onto the k th column vector as: $c_k = \sum_{i=0}^{nm} e_i U_{k,i}$ where e_i and $U_{k,i}$ are the i th elements of e and the k th column of U respectively.

After the projection coefficients c_k have been calculated, the mean squared error ϵ between the approximated input image \hat{e} and the input image e can be computed to judge the degree of fit.

3.1 Eigenspace matching

Essentially, input images are represented as textures mapped onto quadrilaterals. The processing is done in a fragment shader program. The fragment shader can be considered as a short program which is run on each of the pixels of the texture map. The output of the fragment shader is a pixel value in the frame buffer which can sent to the processor, or be used again in another fragment shader program.

Given any new image e , the task of matching consists of calculating an inner product of e and each column of U . The operation of an inner product is an element wise multiplication followed by a summation of each result. We map this onto the graphics card by placing the k th $mn \times 1$ column of U (denote this by U_k) into a $m \times n$ two dimensional texture. These images may be considered eigen basis images.

The normalized input image and each of the basis textures are then each displayed (four at a time), using the multitexturing capabilities of OpenGL. A fragment shader program runs on each pixel, calculating an inner product of the texel of e with each texel of each U_k , placing each result in one of the four components of the output pixel. In a single pass, for non-colour lightspace images, it is possible to take the inner product of e with four distinct U_k , and place the results into four 32-bit floating point values in the 128-bit pixel output from the graphics hardware.

Figure 3 depicts the operation of the fragment shader. The images are 640×480 greyscale, which pack into a 160×480 texture, with each texture element (texel) holding 4 pixel values. This packing is desirable since it makes use of the wide SIMD computations carried out by the GPU.

After the fragment program has been run, the partially summed texture 160×480 in our implementation, is read back into the graphics memory² and stored as a texture. This texture is then redisplayed in a second pass, and another fragment shader program is run to calculate a sum.

²on OpenGL GNU/Linux systems, this is achieved by using `glCopyTexSubimage2D`

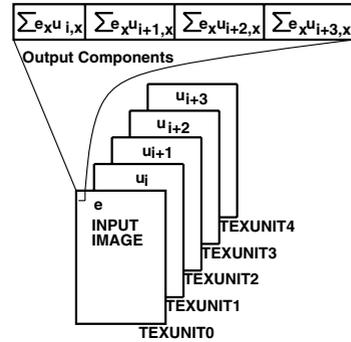


Figure 3: Projecting input image onto eigen images using 5 textures. Each resultant pixel contains four inner products of the input image with each of the four basis vectors in a single pass.

To do this, only a portion of the texture is displayed, and, at each output fragment, the sum of a neighborhood of texels is calculated. In our implementation, we take a sum of a 2×160 area of the texture. Thus, we display an 80×3 quadrilateral. A complete sum was not possible, as the number of instructions in a fragment program is limited to 1024. This needs to be rendered again to the video memory as a texture and a third pass computes a final sum.

The approximate image \hat{e} and a mean squared error calculation can be achieved as above, replacing the inner product fragment shader appropriately.

To classify an input image against a number of different eigenspace representation, the above can be repeated for each eigenspace. This lends itself well to parallelization, where each graphics card is responsible for matching an image to a particular eigenspace, or small number of eigenspaces. This exploits the parallel memory bandwidth of the multiple cards.

If the graphics card needs to be updated many thousand times a second, the CPU will tend to become a bottleneck. This is avoided by making sure as much work is done per pass as possible in the fragment shader programs. Additionally, OpenGL provides *display lists* which are essentially compiled OpenGL calls stored on the graphics memory. A single CPU function call activates a display list, which the GPU runs without further CPU intervention reducing CPU load, preventing it from becoming the bottleneck.

Reading back data from the graphics devices stalls the graphics pipelines, and can cause contention for CPU time between the graphics devices. Our implementation minimizes the amount of data to be read back since only a set of coefficients or mean squared error values are read back from the GPU. Without readback the graphics cards ran completely independently at their maximum speeds. We found that allowing the process to sleep while the GPU completed calculations smoothed the processes, as the CPU was likely spinlocking waiting for the GPU to complete. More processes resulted in spurious increased completion times as

| # of Cards | Time (msecs) | Speedup | CPU Load % |
|------------|--------------|---------|------------|
| 0 (CPU) | 99.0 | – | 100 |
| 1 | 105.5 | – | 17 |
| 2 | 105.5 | 1.9 | 30 |
| 3 | 105.6 | 2.8 | 40 |
| 4 | 107.1 | 3.7 | 55 |
| 5 | 108.1 | 4.6 | 66 |

Table 1: Average Completion Times for increasing numbers of parallel GPUs. Sleep statements relieved CPU load/swapping. Each completion produces 48 coefficients of projections of 640x480 floating point images, comparing the input image to approximately 59 MB of eigen bases. Our test system uses 5 PCI cards, the 6th slot lost to a dual size AGP card.

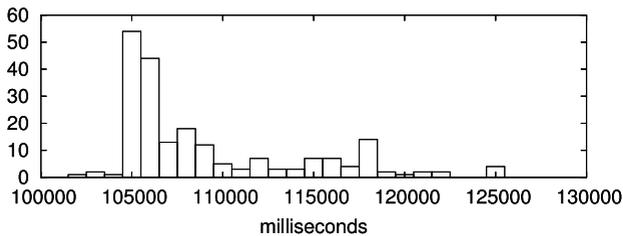


Figure 4: Completion times histogram for 210 recorded completions, with 5 parallel GPUs active

shown in figure 4.

Repeated generation of texture coordinates uses GPU cycles, but does not get useful work done. For the summation pass, a small optimization was made in which two columns are used in the sum, thus a single addition of the y coordinates of each column resulted in two new texture coordinates, at the cost of a single 4 component addition. For the final summation pass, it was possible to hard code the texture coordinates for lookup.

4. Results

We ran our system projecting an input image repeatedly onto 48 eigen basis images. Starting with a single GPU, we added more GPUs and noted the resulting speeds which occurred with more parallelism. This examines the system behavior as more GPUs are added for more throughput. Table 4 shows timings for increasing numbers of graphics cards running in parallel. We used an input image size of 640×480 , and projected it onto 48 eigen basis vectors. Included in the times are the tasks of sending the 640×480 24-bit per pixel image to the card, and readback of the results (48 floating point coefficients). An AMD XP2800+ CPU performs the same calculations in 98.977 milliseconds. If the same images are reused repeatedly the calculations are conducted fast on the CPU (even without SIMD extensions such as MMX). However, for large data sets, the computation slowed down to the measured steady state as more images are considered. This was likely due to the difference in cache utilization, which is a consideration even if SIMD extensions are used. Thus, we conclude that the primary bottleneck in these eigenspace calculations is the memory,

rather than the floating point speed of the CPU. Our PC was equipped with 333 MHz DDR RAM, and the graphics cards each have DDR RAM clocked at 405 MHz or 333 MHz.

The examined algorithm required two rendering passes to compute the sum of the texture. On the CPU, this would not be required, since, as each individual dot product is calculated, the result can be added to an accumulation register, which would hold the sum at the end of a single pass. This greatly alleviates the memory access requirements. On the GPU and with the programming environment allowed by fragment shader programs, however, there is no such available accumulation register. The multiple passes thus require an additional write and read at each fragment for each additional pass (one to write the results of the previous pass to memory, and another to read this result in the next pass).

5. Conclusion

We have shown that multiple graphics devices can be used to create a parallel architecture which carries out pattern recognition and computer vision tasks faster than the CPU. The 5200 FX GPUs used here will likely be replaced by faster GPUs and the PCI-X bus in the near future, providing greater speedup. For instance, the GeForce FX 5900 runs the same task in about 25 msecs. This lets the CPU conduct other, higher level tasks. In our system, an additional five GPUs provided $4.5 \times$ more throughput than the CPU implementation. The speedup is primarily achieved because each graphics card accesses its own memory, allowing parallel, contention free memory access, increasing the overall memory bandwidth of the system. Greater memory bandwidth was well utilized in the eigenspace recognition task examined.

References

- [1] M. Black and A. Jepson. Eigenttracking: Robust matching and tracking of articulated objects using a view-based representation. *Proc. 4th European Conf. on Computer Vision*, pages 329–342, April 1996.
- [2] J. Fung, F. Tang, and S. Mann. Mediated reality using computer graphics hardware for computer vision. In *Proceedings of the International Symposium on Wearable Computing 2002 (ISWC2002)*, pages 83–89, Seattle, Washington, USA, Oct. 7–10 2002.
- [3] F. M. Some computer organizations and their effectiveness.. *IEEE Trans. on Computers*, C-21(9):948–960, Sept. 1972.
- [4] S. Mann. Comparametric equations with practical applications in quantigraphic image processing. *IEEE Trans. Image Proc.*, 9(8):1389–1406, August 2000. ISSN 1057-7149.
- [5] H. Murase and S. Nayar. Visual learning and recognition of 3-d objects from appearance. *International Journal of Computer Vision*, 14:5–24, 1995.
- [6] R. Yang and P. M. Multi-resolution real-time stereo on commodity graphics hardware. *Computer Vision and Pattern Recognition, 2003. Proceedings. 2003 IEEE*, 1:211–217, 2003.