Implementation of Kernel Methods on the GPU

Julius Ohmer
Queensland University of Technology
School of Software Engineering and Data Communication
126 Margaret Street, Brisbane QLD 4000, Australia
j.ohmer@student.qut.edu.au

Frederic Maire
NICTA
Level 19, 300 Adelaide Street
Brisbane, QLD 4000, Australia
Frederic.Maire@nicta.com.au

Ross Brown
Queensland University of Technology
Faculty of Information Technology
126 Margaret Street, Brisbane QLD 4000, Australia
r.brown@qut.edu.au

Abstract

Kernel methods such as kernel principal component analysis and support vector machines have become powerful tools for pattern recognition and computer vision. Unfortunately the high computational cost of kernel methods is a limiting factor for real-time classification tasks when running on the CPU of a standard PC. Over the last few years, commodity Graphics Processing Units (GPU) have evolved from fixed graphics pipeline processors into more flexible and powerful data-parallel processors. These stream processors are capable of sustaining computation rates of greater than ten times that of a single CPU. GPUs are inexpensive and are becoming ubiquitous (desktops, laptops, PDAs, cell phones). In this paper, we present a face recognition system based on kernel methods running on the GPU. This GPU implementation is twenty eight times faster than the same optimized application running on the CPU.

1. Introduction

GPU are the first commodity, programmable parallel architecture that can take advantage of data-parallelism. Remarkably, GPU performance is increasing much faster than CPU performance. GPU evolution and low cost are driven by the computer game market [9]. Harnessing the power of a GPU is hard because the data-parallel algorithms' mapping to graphics primitives presents many performance pitfalls. GPUs were designed for graphics applications. These are characterized by a lot of arithmetic, intrinsic parallelism, simple control, multiple stages, and feed forward pipelines. Computer vision and computer graphics are both characterized by a high degree of local processing which must occur per pixel (or in a small region, achieved perhaps by filtering) and can be considered (to a certain extent) as inverse processes. Basic image manipulation operations, such as convolution filters and color transformations, are well suited for GPU processing. Real-time computer vision has many important real-world applications such as video surveillance, human-computer interaction and computer vision. Realizing real-time computer vision on the GPU create new opportunities by drastically lowering the cost of computer vision systems. One of the most successful face recognition techniques is based on kernel principal component analysis (KPCA). In this paper, we present the implementation on the GPU of a face recognition system based on KPCA. This implementation is 28 times faster than the same application running on the CPU.

Section 2 gives a brief overview of general purpose computation on the GPU. In Section 3 we recall some basics of kernel methods. Section 4 presents our approach for implementing kernel methods on the GPU. Section 5 explains some implementation details. Section 6 discusses some experimental results.

2. General Purpose Computation on Graphics Hardware

Three key factors make the GPU an attractive platform for general purpose computation:

Speed - The computational capacity of GPU increases at an astonishing rate. The rate of growth outpaces Moore's

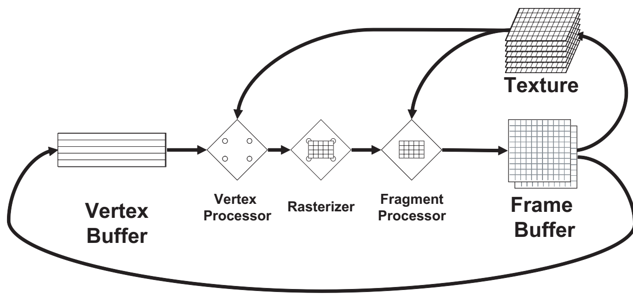

Figure 1. The modern graphics hardware pipeline. The vertex and fragment processor stages are both programmable by the user. Source:[9]

Law. An ATI X800 XT graphics card can achieve over 63 GFLOPS. A 3.7 GHz Intel Pentium 4 SSE has a theoretical peak of 14.8 GFLOPS. Modern GPUs offer a tremendous memory bandwidth compared to CPUs. The CPU is optimized for high performance on sequential code and many of their transistors are dedicated to support non-computational tasks, like branch prediction and caching. The highly parallel nature of the GPU allows to use additional transistors for computation, and achieve a higher arithmetic intensity with the same transistor count. Modern GPUs are also at the cutting edge of processor technology: They contain over 300 million transistors and are built on a 110-nanometer fabrication process.

Value for money - GPUs can be found in off-the-shelf graphic cards which are built for the PC video game market. The latest generation usually GPUs cost about US $450-500. Their price drops significantly each time a new generation is released.

Technical maturity - Modern GPUs have reached an advanced state of maturity. Today, two processors are fully programmable on the graphic hardware; the vertex and the fragment processor. They are placed in a hardware pipeline as illustrated in Figure 1. The nVidia GeForce6800 fully supports IEEE 32-bit floating point numbers (single precision). The number of inputs, outputs and instructions for each of the processors has been significantly increased. GPUs support full branching in the vertex pipeline and limited branching in the fragment pipeline.

But modern GPUs present serious challenges. The fact that they adopt the SIMD / MIMD parallel processing scheme means that algorithms only benefit from the architectural properties, if they can be adapted to that scheme. Moreover, important fundamental constructs such as integer data operands are missing and associated operations such as bit-shifts and bitwise logical operations. The GPU lacks 64-bit double precision number formats. The communication between a program running on the CPU and the graphics hardware has to be established by using a graphics API such as *OpenGL*. To perform computations on graphics hardware, the programmer must use graphic primitives such as lines or triangles and stores the data in texture maps. An alternative way we have not pursued is to use one of the stream computing languages for the GPU that are in development. For futher information, the reader is referred to the comprehensive survey of general purpose computation on the GPU conducted by Owens et al. [9].

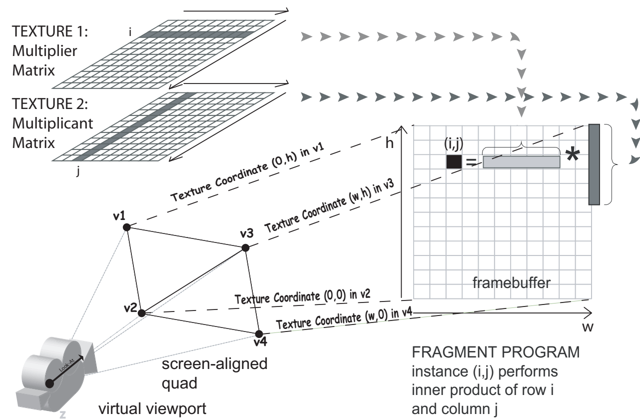

Figure 2. Matrix multiplication on the GPU.

Next (and in Figure 2), we describe a sequence of steps for performing a matrix-matrix multiplication on the GPU (following a method which was pioneered by Larsen and McAllister [6]).

1. Upload data arrays into the GPU memory which are later used by the vertex or fragment programs. These data fields are called *textures*. They are one-, two- or three-dimensional data arrays. Each entry in the array can hold up to four values (red, green, blue and alpha components). In this example, *texture 1* contains the data of the multiplier matrix and *texture 2* the data of the multiplicand matrix.

2. Define a virtual viewport in OpenGL.

3. Activate a previously defined fragment program.

4. Render a screen aligned quad which must exactly match the viewport dimension. The fragment program performs the computation by rendering the quad. One *processing instance* of the program runs for each pixel covered by the quad. The programs have read-only access to the data resource (textures). The information about the location (the values of i and j) of the $(i,j)^{th}$ fragment processing instance is provided by interpolating the *texture coordinate* values. The texture coordinates are defined in the vertices, which bound

the quad. Here, the processing instance (i,j) of the fragment program performs the inner product between the ith row of *texture 1* and the jth column of *texture 2*. The processing instance uses the values i and j to look up the proper locations in the textures. The result of each fragment program instance is written at location (i,j) to the *framebuffer* which is essentially two-dimensional data array very similar to a texture.

5. (a) Retrieve the content of the framebuffer by performing a read back from GPU memory to CPU memory.

   (b) Provide alternatively the content of the framebuffer as a texture for a subsequent GPU computation. We use the *Framebuffer Objects* in OpenGL, which support this functionality.

The process described above is known as a *rendering pass*. An eclectic range of applications in non-graphics areas for the GPU has already been explored; Krüger et al. [4] and Bolz et al. [1] proposed to use the GPU for numerical computation. Even image processing methods such as convolution filters, color conversion have been efficiently implemented [2]. Fast implementations on the GPU of more complex algorithms include the fast Fourier [7] and wavelet transforms [14]. For more information please refer to [9].

In 2004, Kyoung-Su Oh and Keechul Jung introduced an efficient implementation of a Neural Network on the GPU [5] and demonstrated that pattern recognition and classification methods can also benefit from the architectural properties of modern GPU. The problem of simulating a network with k layers was subdivided into a k - pass rendering problem. The pass i is performed by taking the results from pass $i-1$ and computing a linear combination between them and the weights of the nodes in the current layer. This operation was found to be very similar to a matrix-matrix multiplication, which can be efficiently implemented on the GPU [6]. The GPU implementation of the Neural Network outperformed the CPU implementation by a factor of 20.

3. Kernel Methods and Their Applications

Kernel methods are a modular approach in pattern analysis which is efficient, robust and statistically stable [11]. First the data items are embedded into a vector space, also called the *feature space*. Then linear relations are sought among the images of the data items in the feature space. The algorithms are implemented in such a way that coordinates of the embedded points are not needed, only their pairwise inner products. They can be efficiently computed directly from the original data by using a kernel function.

There are two reasons why this approach works well:

1. Detecting linear relations in patterns has been the focus of much research in statistics and machine learning for decades. The resulting algorithms are both well understood and efficient.

2. There is a computational shortcut which makes it possible to represent linear patterns efficiently in high dimensional spaces to ensure adequate representational power (this shortcut is known as *the kernel trick*).

A kernel function measures the similarity of two data vectors x and y. This similarity is expressed as the inner-product $k(x,y) = \langle \Phi(x), \Phi(y) \rangle$ of the images of x and y through an implicit mapping Φ into an Hilbert space F (known as the *feature space*).

The most popular kernels are the *polynomial*, *radial basis function* and *sigmoid kernel*;

$$k_{poly}(x,y) = (s(x \cdot y) + c)^d \quad (1)$$

$$k_{rbf}(x,y) = \exp(-\gamma ||x-y||^2) \quad (2)$$

$$k_{sigmoid}(x,y) = \tanh(s(x \cdot y) + c) \quad (3)$$

where s, c, d and γ are real constants of the kernels.

3.1. Support Vector Machines

Vapnik et al. [13] introduced SVM classifiers f whose decision surfaces are hyperplanes in some dot product space K

$$f(x) = \text{sgn}(\langle w, x \rangle + b) = 0, \text{ where } w \in K, b \in \Re$$

The parameters of f are derived from a training set. The SVM solution corresponds to the hyperplane with the maximum margin between positive and negative training examples. The decision function $f(x)$ can be expressed using the m training examples:

$$f(x) = \text{sgn}(\sum_{i=1}^{m} \alpha_i \langle x, x_i \rangle + b)$$

The coefficients $\alpha_i \in \Re$ and b are found by solving a quadratic optimization problem (see [10] for further details). In fact, only for some vectors x_i the coefficients α_i are non-zero. These vectors x_i determine the position and orientation of the hyperplane and are thus called support vectors. Replacing the inner product term $\langle x, x_i \rangle$ with $\langle \Phi(x), \Phi(x_i) \rangle$ creates non-linear decision surfaces in the input space corresponding to linear decision surfaces in a higher dimensional

feature space;

$$f(x) = \text{sgn}(\sum_{i=1}^{m} y_i \alpha_i \langle \Phi(x), \Phi(x_i) \rangle + b)$$
$$= \text{sgn}(\sum_{i=1}^{m} y_i \alpha_i k(x, x_i) + b) \quad (4)$$

By using the non-linear kernel function, the input patterns are mapped non-linearly into a high dimensional feature space, where a separating hyperplane between two classes of data is more likely to exist. The left diagram of Figure 4 illustrates the working of a SVM classifier (see Equation(4)) .

Support vector machines have been successfully applied to computer vision tasks like face detection and recognition [8]. It should be pointed out that support vector machines can also be used for non-linear regression [10].

3.2. Kernel Principal Component Analysis

Principal Component Analysis (PCA) performs a dimension reduction on the input data by computing the principal axes of the training data. New input data are projected onto these principal axes for reducing their dimension. The first k principal axes are the first k eigenvectors of the covariance matrix of the training set where the eigenvectors of the covariance matrix are ordered by decreasing eigenvalue. We must have $k \leq M$ where M is the size of the training set. The new k coordinates are known as *principal coordinates*.

Kernel PCA (KPCA) is a method that finds directions that maximizes the variance in a feature space rather than in the input space of the data. The projections onto the feature space eigenvectors can be computed through a dual representation computed from the eigenvectors and eigenvalues of the kernel matrix.

In the following, A' will denote the transpose of a matrix A. The projection onto the k-dimensional subspace U_k spanned by the principal axes of the image $\Phi(x)$ of the data vector x can be computed in following way:

$$P_{U_k}(\Phi(x)) = (u'_j \Phi(x))_{j=1}^{k} = (\sum_{i=1}^{M} \alpha_i^j k(x_i, x))_{j=1}^{k} \quad (5)$$

where $\alpha^j = \frac{1}{\sqrt{\lambda_j}} v_j$

The eigenvalues λ_j and eigenvectors v_j can be found by

1. Constructing the kernel matrix from the training set $S = x_1, ..., x_M$ and dimension k:

$$K_{ij} = k(x_i, x_j) \text{ with } i, j = 1, ..., M$$

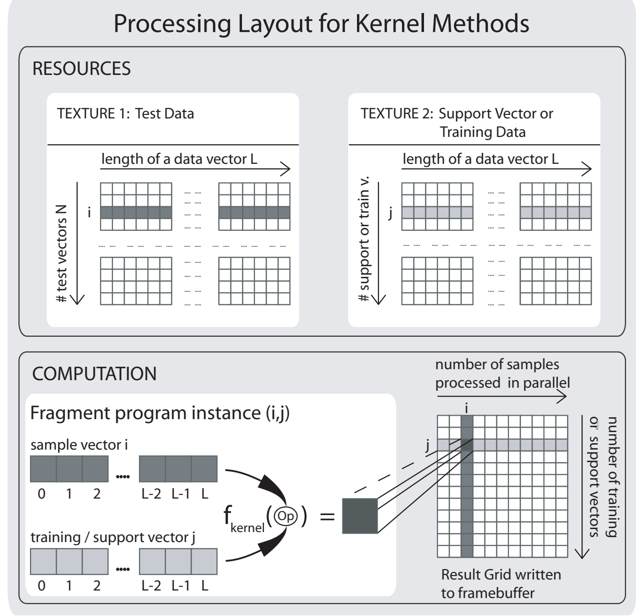

Figure 3. Parallel processing for kernel evaluations.

2. Centering the data in feature space as follows:

$$K - \frac{1}{M} j j' K - \frac{1}{M} K j j' + \frac{1}{M^2} (j' K j) j j'$$

3. Extracting the eigenvalues and eigenvectors from K:

$$[V, \Lambda] = \text{eig}(K)$$

4. Normalizing the eigenvectors in feature space F as follows:

$$\alpha^j = \frac{1}{\sqrt{\lambda_j}} v_j \ , \ j = 1, ..., k$$

The principal coordinates of new data can be computed using Formula (5). The above description of KPCA follows Taylor et al. [11]. Yang et al. [15] found that the Kernel PCA performs better than the PCA for face recognition problems. They achieved good results using the AT&T face data base [12].

4. Approach

The kernel evaluations constitute the main computational burden for an online recognition system using kernel methods. Data vectors commonly used in image processing tend to be long; even the representation of a relatively small grayscale image with the dimension of 20 by 20 is a data

vector with 400 luminance values. When color images are processed, the data vector size triples, since the values for all three color components must be stored per pixel.

Both SVM and KPCA perform a kernel evaluation between new sample vectors and stored data vectors involving the inner product (Equations (1) and (3)) or the sum of squared differences (Equation (2)) for the three most common kernels.

The main idea of our apporach is to maximize the benefit coming from the SIMD parallel architecture of the GPU. We exploit parallelism when kernel methods are mapped to the GPU in the following two ways:

Processing kernel evaluations in parallel - The SVM performs the kernel evaluation between a new sample and each of the support vectors (as shown in Figure 4). Since there is no dependency among these computations, they can be done in parallel.

The situation for KPCA is similar. For a new data vector x, the kernel function must be evaluated between x and each of the stored training vectors. Figure 4 illustrates this process.

Processing many candidates at once - Parallelism can also be achieved by just scaling up the problem; instead of just processing one example at a time, the system can handle many examples in parallel. This is possible since the data can be shared among them and is not altered by an individual example. The motivation to consider this approach was driven by the fact that real-time recognition systems often evaluate many candidate regions in a movie frame to track a certain feature.

5. Implementation

We developed our implementation using the graphics API *OpenGL* to communicate with the graphics hardware and the high-level shading language *Cg* from *nVidia* to define programs for the vertex and fragment processor.

We use the considerations made before in Section 4 to define a grid with the dimension *MxN*. The number of rows M correspond to the number of stored vectors containing training data (KPCA) or support vectors (SVM). The number of columns N correlate with the number of samples processed in parallel. OpenGL is used to render a screen aligned quad matching the dimensions stated above. When the geometry of the quad arrives at the fragment processor, a fragment program instance runs for every cell of the grid. The result of the kernel evaluation between the ith test sample and the jth training/support vector is stored at position (i,j) in the grid, just as demonstrated in Section 2. The data are provided to the fragment program as textures. Figure 3 illustrates the whole process.

5.1. Support Vector Machines

The implementation of our SVM classifier follows the diagrams of Figure 3 and Figure 4. We translated this process into a rendering problem with two passes:

1. The first pass computes the *NxM* grid as in Figure 3. The evaluation of the kernel between the test vector of the ith SVM instance and the jth support vector is stored at position (i,j) in the grid. The support vectors x_i and their associated weights α_i $(i = 1,...,m)$ were determined offline using Thorsten Joachim's SVM implementation SVM^{light} [3] and were uploaded as a texture into the GPU memory. The sample regions are extracted from a video stream, which is available as a dynamic texture.
2. The second pass performs a parallel linear reduction of the results of pass one. Each cell i of the one dimensional vector of size N contains now the classification result of the SVM for the ith sample vector.

5.2. Kernel Principal Component Analysis

Our experiments for the KPCA was done independently of our SVM implementation. We implemented the experiment proposed by Yang et al. [15] and performed the KPCA using the AT&T image library [12] as a training set. The complete layout of our experiment can be seen in Figure 5.

The training of the KPCA was done in Matlab. All greyscale images from the image library were rescaled to 33 by 40 pixels. Each image is represented by a vector of 1320 elements. The *MxM* kernel matrix K_{train} was computed using all 400 image vectors of the image library. A polynomial kernel with degree 2 was used. After the matrix was centered, the eigenvectors were extracted and normalized in the feature space. The first 48 eigenvectors were used to extract the features (principal coordinates) of all 400 training samples. The recognition part of the KPCA is performed by a C++ program. Data from the Matlab implementation is loaded into floating point arrays:

Image data - the (re-scaled) image data from the AT&T data base. For simplicity sake we use the image data in the following experiment as test data to measure the performance and the precision.

$\mathbf{K_{train}}$ - the kernel matrix built from the training data. It has to be available to center the test data.

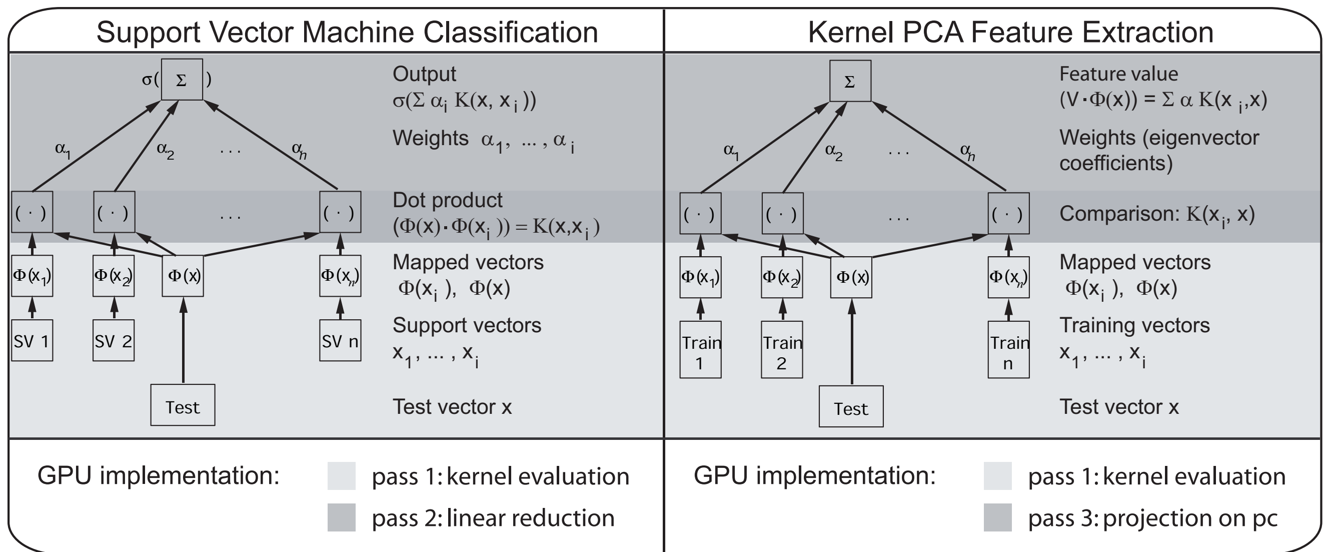

Figure 4. The two tasks performing recognition (SVM) and feature extraction (KPCA), involve kernel evaluations as a first processing step.

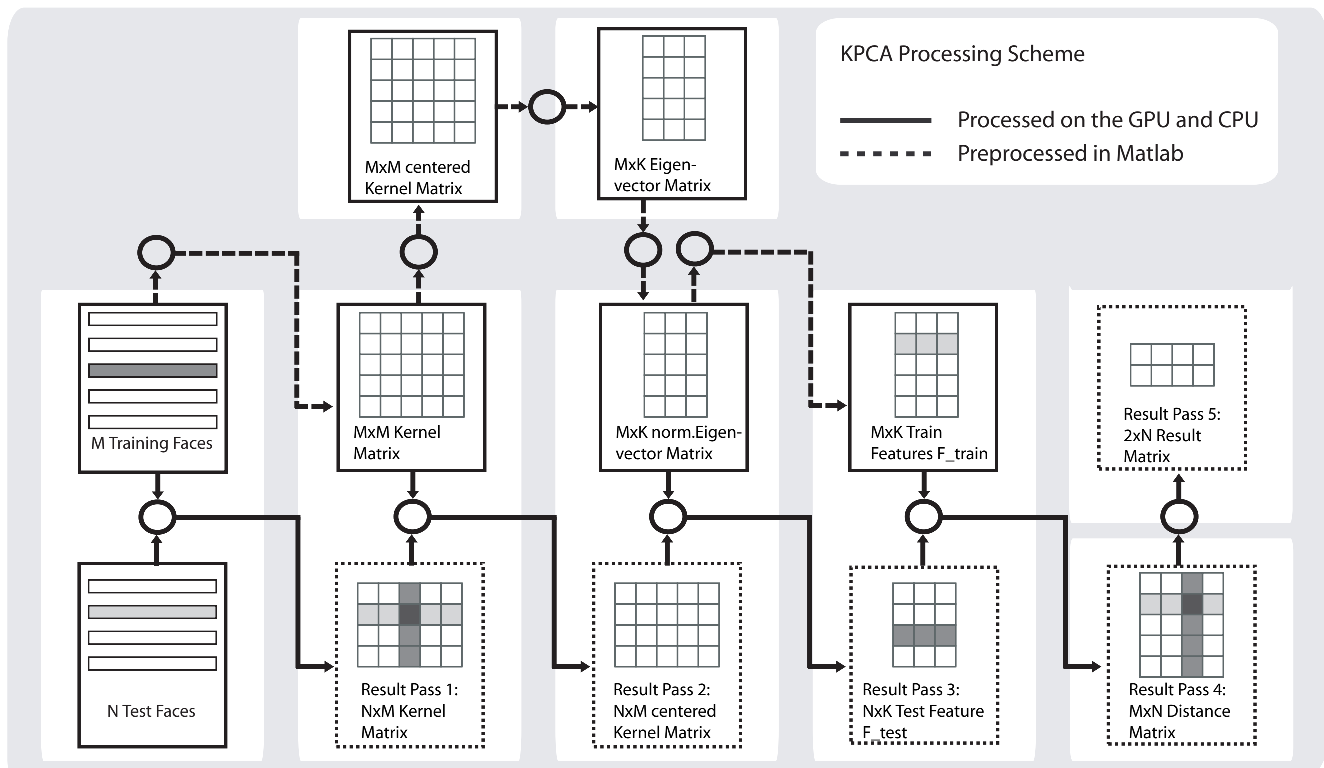

Figure 5. Schematic processing layout of the KPCA experiment.

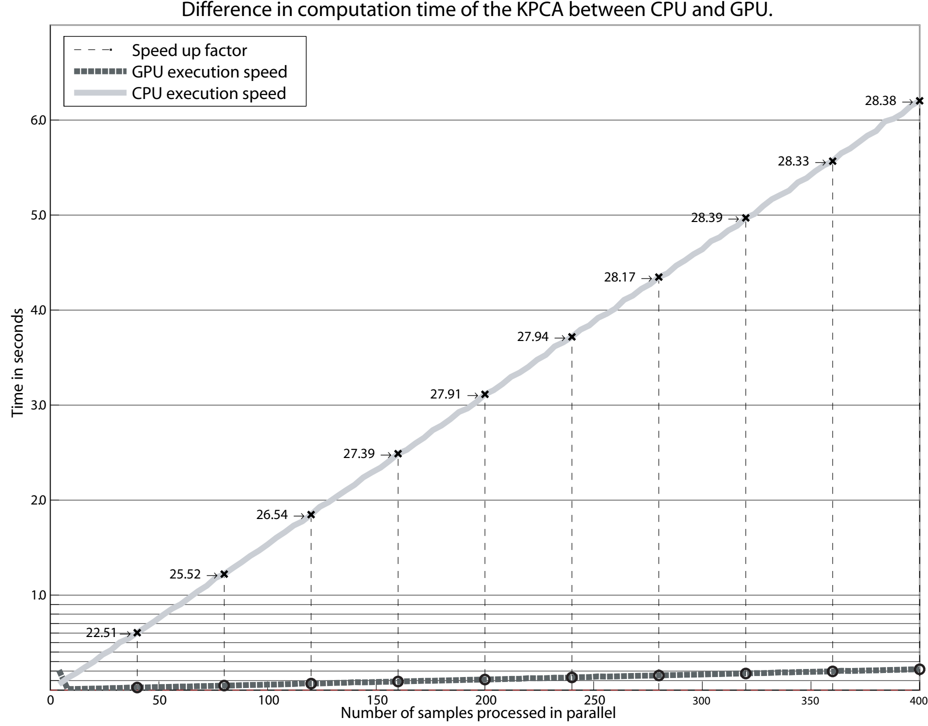

Figure 6. The increase in execution speed of the GPU implementation. The GPU used for this experiment was a nVidia GeForce6800. The CPU was an Athlon64 3200+ (32-bit mode).

Eigenvectors - the normalized eigenvectors are necessary to extract features from the test images.

$\mathbf{F_{train}}$ - the training features are used to perform the recognition after the feature of the test data has been extracted.

The C++ implementation performs the following computation on both the GPU and the CPU. The motivation for an additional CPU implementation has three reasons; debugging the GPU implementation, measuring a possible loss in precision, and measuring the difference in execution speed. In our implementation N test vectors are processed in parallel, where $4 \leq N \leq 400$. The face recognition with the KPCA is computed in five rendering passes:

1. The $N \times M$ kernel matrix K_{test} is computed following the processing scheme presented in Section 4.
2. The matrix K_{test} is centered in feature space, using the formula (see [10] for the theory)

$$K'_{test_{ij}} = K_{test_{ij}} - \frac{1}{M}\sum_{k=1}^{M} K_{train_{jk}}$$
$$-\frac{1}{M}\sum_{k=1}^{M} K_{test_{ik}} + \frac{1}{M^2}\sum_{k,l=1}^{M} K_{train_{lk}}$$

   The terms which just require information about K_{train} are precomputed.

3. The test features F_{test} are extracted for the N test examples, by using the first k eigenvectors and the previously centered matrix K'_{test}.
4. A $M \times N$ distance matrix D is constructed where the entry D_{ij} devotes the Euclidean distance between the ith feature vector from F_{train} and the jth feature vector from F_{test}.
5. The distance matrix is linearly reduced to a $2xN$ result matrix R. Each entry pair (R_{1i}, R_{2i}) contains the value of:
   - R_{1i} - the value of the smallest distance found in the ith column of D, and
   - R_{2i} - its position (the associated row number where this distance was found in D).

6. Results

We conducted experiments to measure the speed difference between the GPU and CPU implementations of the KPCA. We attempted to equally optimize both implementations. We measured the execution time of both implementations

using a high precision timer. The timer was started just before the execution on the GPU was triggered by using appropriate OpenGL calls. The command glFinish() was used to ensure that all GPU calculation have finished before the timer was stopped. It was found that in particular the first rendering pass has a clear speed advantage on the GPU compared to the CPU. Since it is by far the most expensive operation, the whole implementation experienced a dramatic increase in execution speed. A similar phenomenon occurs with support vector machines, since the processing layout of the first pass is the same. Figure 6 plots the performance results versus the number of samples N processed in parallel. The graph was constructed by repeating the experiment 100 times. N was initially set to 4 and in each iteration incremented by 4. The figure shows already a speed-up factor above 20 with only a few sample evaluations running in parallel.

We found that the recognition results of the KPCA suffered from the lack of 64 bit double precision numbers on the GPU when polynomial kernels with a higher degree were used. In our application, the results become unpredictable if the degree $d \geq 4$. This restriction occured on both platforms, the GPU and the CPU (using 32bit floating point numbers), and is thus not a GPU related problem. Yang et al. reported very good results when a polynomial degree of 3 was used [15].

7. Conclusions

We have demonstrated how kernel methods can benefit from the architectural properties of modern GPUs by exploiting the intrinsic parallelism of these methods. GPU implementations of kernel methods are well suited for computer vision applications where patterns need to be detected and tracked in real-time. It is possible to process many candidate regions in parallel and thus allow stable tracking of those patterns. Our proposed implementations are, in contrast to hardware implementations, very flexible. They can be applied for classification, regression and feature extraction with any standard kernel. However, currently polynomial kernels on the GPU can suffer from the lack of high numeric precision (no long double type). In future work, we plan to adapt the SVM coefficients directly on the GPU.

References

[1] J. Bolz, I. Farmer, E. Grinspun, and P. Schroeder. Sparse matrix solvers on the GPU: Conjugate gradients and multigrid. *ACM Transactions on Graphics (TOG)*, 22(3):917–924, July 2003.

[2] P. Colantoni, N. Boukala, and J. D. Rugna. Fast and accurate color image processing using 3d graphics cards. In *8th International Fall Workshop - Vision Modeling and Visualization 2003, Proceedings November 19-21, 2003 , München, Germany*, pages 383–390. IOS Press, 2003.

[3] T. Joachims. Making large-scale learning practical. In B. Schölkopf, C. Burges, and A. Smola, editors, *Advances in Kernel Methods - Support Vector Learning*. MIT Press, Cambridge, MA, USA, 1999.

[4] J. Krueger and R. Westermann. Linear algebra operators for GPU implementation of numerical algorithms. *ACM Transactions on Graphics (TOG)*, 22(3):908–916, July 2003.

[5] K. J. Kyoung-Su Oh. GPU implementation of neural networks. *Pattern Recognition*, 37(6):1311–1314, June 2004.

[6] E. S. Larsen and D. McAllister. Fast matrix multiplies using graphics hardware. In *Proceedings of the 2001 ACM/IEEE conference on Supercomputing (SC'01)*, pages 55–55, Denver, Colorado, USA, November 10–16 2001. ACM Press.

[7] K. Moreland and E. Angel. The fft on a GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 112–119, San Diego, California, USA, July 26 - 27 2003. Eurographics Association.

[8] E. Osuna, R. Freund, and F. Girosi. Training support vector machines: an application to face detection. In *CVPR '97: Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, page 130, Washington, DC, USA, 1997. IEEE Computer Society.

[9] J. D. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. E. Lefohn, and T. J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August, 2005.

[10] B. Schölkopf. *Learning with Kernels*. MIT Press, Cambridge, Massachusetts, USA, 2002.

[11] J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, Cambridge, UK, 1 edition, 2004.

[12] AT&T Laboratories Cambridge. The database of faces, 1994.

[13] V. Vapnik and A. Lerner. Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24:774–780, 1963.

[14] J. Wang, T.-T. Wong, P.-A. Heng, and C.-S. Leung. Discrete wavelet transform on GPU. In *Proceedings of ACM Workshop on General Purpose Computing on Graphics Processors*, pages C–41, Los Angeles, California, USA, August 7–8 2004. ACM Press.

[15] M. Yang, N. Ahuja, and D. Kriegman. Face recognition using kernel eigenfaces. In *Image Processing, 2000. Proceedings. 2000 International Conference on*, volume 1, pages 37–40, 2000.