A Large-Scale Mobile Facial Recognition System Using Embedded GPUs

Ahmed El-Mahdy\textsuperscript{1,2} and Radwa Elsersy\textsuperscript{1}

\textsuperscript{1}Computer Science and Engineering Department
Egypt-Japan University of Science
\textsuperscript{2}On-leave from Alexandria University
\{ahmed.elmahdy, radwa.elsersy\}@ejust.edu.eg

Keywords: Embedded GPUs, Face Recognition, Mobile Devices

Abstract
With the rapid growth of mobile phones technology, there is a strong potential for porting server-class applications into the mobile platform. This paper considers the problem of face recognition, and proposes a system that exploits embedded GPUs for allowing for large scale, face recognition. The system provides a novel parallel Fisherface recogniser implementation that allows for a configurable and scalable number of parallel threads. The system also exploits the large secondary storage available, to further scale the recognition database. An initial prototype implementation allows up to 1024 face database size with a recognition time of 210ms on iPhone5 system. The GPU resulted in nearly 10x improvement in performance and 7x reduction in power.

1. INTRODUCTION
The processing capabilities of mobile phones is currently improving rapidly; as of this writing, mobile devices now include up to 8 processing cores and fast, 3-core graphics processing units (GPUs). Moreover, according to the technology roadmap [4], it is expected that such trend will continue at exponential rates, with an emphasis on using GPUs to manage large screen sizes [13]. However, the current trend in the mobile platform is to off-load specialized high performance operations, such as face recognition [2, 7] on the cloud, and keep the user interface on the mobile devices.

In this paper we analyse a highly unexplored area of utilizing the embedded GPU to improve the performance of the face recognition application. We chose face recognition as it is computationally demanding and not fully explored in the literature on the mobile platform [7, 10]. Such use would potentially avoid data communication with servers, avoid intrinsic issues such as data privacy as well as application availability when there is no network connectivity to the mobile device. Example applications include the use of facial recognition in disaster areas to distribute aid, and to detect registered criminals in low connectivity areas such as subways. To study the potential of mobile platform, we consider recognising a face out of a large set of faces typical in standard facial recognition benchmarks.

Designing large-scale facial recognition system requires considering various system components; the main processing power has to be fast and scale with technology improvement as well as being power efficient. We, therefore, focus on the embedded GPU as it is currently surpassing general-purpose processors, and scalability is possible through increasing the number of parallel threads, requiring a highly parallel algorithm. Moreover, the GPU generally has better power efficiency than general-purpose processors.

Another scaling bottleneck is memory; currently embedded memories are few GBytes large. This requires more efficient data structures representation, and also exploiting the significantly larger secondary storage; the latter has reached few hundreds GBytes in size, nowadays. We, therefore, focus on decreasing the memory requirements and potentially allowing for caching main memory on secondary storage.

A full facial recognition system generally includes facial detection, and facial recognition operations. Currently, the detection operation is widespread and routinely available on smart devices, allowing for multi-face detection at real-time speeds. We, therefore, focus on the recognition operation.

Face recognition generally involves two main operations, training and recognition. In this paper we focus on recognition, as it is the most frequent operation, and training can mostly be performed offline. We choose the well-known Fisherfaces algorithm [6] for facial recognition due to its fast speed and relatively low resource requirements. The algorithm relies on representing faces into an eigenface [14] representation, and then projecting the face into an eigenspace, significant reducing the space dimensionality. Such projection is mainly a vector matrix operation that we mainly target in the parallelisation process.

The main contributions of the papers are:

- Propose an efficient parallelisation method for the kernel vector matrix operation with controllable number of threads, matching the concurrency extend of the underlying embedded GPU.
- Implement a working prototype system running on iPhone5; the system uses a 16-bit training set representation. The system recognises up to 1024 images, using the main memory.
- Achieve up to 10x speedup and a 7x reduction in energy
for the embedded GPU when comparing with the serial embedded CPU.

This paper is organized as follows: Section 2 discusses related work. Section 3 provides background on our experience in programming the iPhone/iPad GPU using OpenGL ES 2.0 [11], being one of the main available methods to program the embedded GPU. Section 4 provides overview of the Fisherfaces face recognition method and our parallel implementation of the subspace projection method. Section 5 evaluates the performance and discusses the results. Finally, Section 6 concludes the paper. We also provide pseudo-code for our main parallel kernel in Appendix A.

2. RELATED WORK

Chang and Wang [7] have considered the Gober facial recognition method; the method computes a ‘face feature descriptor’ that is then used for identifying a face from a faces database. The computation intensive operation is the generation of the descriptor; it requires processing non-overlapping patches of the given face, then processing each using Principal Component Analysis (PCA) and Linear Discriminant Analysis (LDA) methods. They utilised the Nvidia’s Tegra2 embedded GPU to accelerate the kernel; the main operation is FFT. They reported 1.2 seconds for the generation of the features, which is 4.25x faster than the underlying Tegra2’s Cortex A9 general purpose processor, with 3.98x energy reduction. The system uses much smaller face sizes than our system (64 $\times$ 84 vs. 256 $\times$ 256), and does not consider scaling the database size.

Leskela et al. [10] have reported performance and power measurements for an image processing kernel on the PowerVRGTM SGX 530 GPU and ARM Cortex A8 CPU. The image processing kernel included geometry, gaussian blur and colour adjustment operations. They reported 3.6x speedup over the CPU, with 7x reduction in power. However, the work did not consider the problem of facial recognition, but reported similar performance results to that of Chang and Wang [7] in terms of performance.

More recently Chang et al. [15] have reported using an Embedded Profile OpenCL on Qualcomm Snapdragon S4 with multi-core Krait CPU and Adreno GPU. They reported 4.1x speedup for an object removal algorithm. The results again do not consider the face recognition application, but reports similar performance to elsewhere.

F. Zuo and P. H. N. de With [17] have proposed an embedded system for facial recognition on a smart home setting. The system used a constrained statistical feature extraction method (Enhanced Active Shape Model). The method essentially overlays a shape model over the face under investigation; it then adapts the model to the face using PCA analysis. However, they considered a small database size of 25 persons and considered much powerful processors (Intel’s Pentium IV), achieving 30ms to recognise a face.

Chang and Wang [7] have also compared the performance of an embedded GPU and a desktop GPU. They have reported a maximum of 91x speedup when comparing Tegra GPU (ULP GeoForce) with the desktop’s GeForce 8800, and 282x when comparing an ARM Cortex A9 with the same GPU. The corresponding power for the the Cortex A9 is 3.67W, for Tegra CPU is 4.1W and the desktop’s GeForce 8800 is 483.89W, giving reductions of 132x and 118x, respectively, when comparing the embedded CPU and GPU with the desktop. However, energy consumption is much higher for the mobile platform, giving better performance/power ratio for the desktop.

In this paper we consider the iOS platform. Fig. 1 summarises the RAM size and processing performance scaling for various generations of the iPhone platform. Technology scaling shows significant improvements every two years. A two-year average improvement for CPU is 3.2x and for the GPU is 9.6x which accelerating at a rate of 3x faster than the CPU. Memory size increases with the average rate of 2.3x every two years.

3. GENERAL PURPOSE PROCESSING USING OPENGL ES

The objective of this section is to describe our experiences in programming the OpenGL ES 2.0 for general-purpose computing on embedded GPUs of the iOS platform. It is worth noting that Leskela et al. [10] have developed an OpenCL prototype for embedded devices, however, the software is not publicly available; OpenGL provides more functionality, in general. However, the OpenGL ES is currently the widely available development system for the embedded GPU.

![Figure 1. Summary of RAM sizes, CPU and GPU speeds over iPhone generations](image-url)
The hardware of the embedded GPU [5] is specifically designed for graphics processing and does not have support for general-purpose processing as the desktop. For instance, write/read memory is very limited, and data types are particularly of low precision. The hardware architecture is based on the ‘standard’ graphics pipeline as depicted in Fig. 2.

The input to the pipeline is a set of 3D vertices; the first stage of the pipeline performs geometrical transformations. The transformations are performed by the vertex shader using a C-like language for general processing [9]: the next stage, the rasterizer, generates discrete pixel values called fragments; a fragment is a screen pixel but with depth information (z coordinate). The fragment shader determines the colour of each fragment; a C-like language, generally referred to as OpenGL Shading Language (GLSL), programs the shader. The language allows general shading processing; the shader also reads from texture units to perform texture mapping. The pixel colours are then written into the frame buffer for outputting on the screen.

Four-values vectors represent pixel values, generally. The values correspond to the three-colour components R, G, and B, and a fourth transparency value (a-values). In OpenGL terminology, a four-component short vector type, called ‘vec4’, represents the components.

We use the fragment shader to allow for data parallel processing; we setup simple primitive vertices to generate a rectangular region of pixels that covers the entire frame buffer; thus a shader program acts as a thread with the pixel coordinates as its identity key.

There are three main memory types: scalar constant variables, 1D/2D constant data array and output write-only 2D array; the constant data array can be a read-only array that was written in an earlier shader invocation (render-to-texture method [8]).

We have experienced some intricacies while programming the GPU. A notable one is that the texture data are indexed by a normalised float value, pointing to the middle of the indexed pixel, computed as: 
\[(x + 0.5)/\text{dim size}, \text{where } x \text{ is the integer pixel index (starting from 0). It is also worth noting that there is a limitation on the number of instructions executed by the shader (we found out that programs consuming more than 5sec on the shader crashes on the iPhone5 platform). Finally, 16-bit floating-point support is available starting from iPhone4S and iPad2 (GL_HALF_FLOAT_OES data type). More recently, the iPhone5s supports 32-bit floating-point operations.}

4. PARALLEL SUBSPACE PROJECTION

4.1. Fisherfaces Method

Fisherfaces method relies on the linear projection the image space into a lower dimensional feature subspace, such as the Principle Component Analysis (PCA) and eigenfaces methods [14]. Generally, an image is represented as a vector, \(x\), of pixel values of length \(n\). The image space is thus \(n\)-dimensional. Each image \(x\) belongs to a class; the class represents a person, and the image is one sample of the person. The recognition problem is then to identify the class, given the sample image. More formally, the projection is done by computing \(y = x^T W\), where \(y\) is the projected vector in an \(m\)-dimensional feature space \((m \leq n)\), and \(W\) is the linear transformation (projection) matrix with dimensions \(m \times n\).

Fisherfaces method selects \(W\) in a way to maximise the ratio of inter-class scatter to intra-class scatter; the columns of the \(W\) matrix are eigenvectors of the corresponding scatter matrix. For complete details, the reader is advised to refer to Belhumeur et al. [6].

4.2. Parallel GPU Implementation

As indicated above, the main operation for recognition is projecting a vector image into an eigen subspace, which is a vector matrix multiplication operation. For our implementation, an image is \(256 \times 256\), grey scale (1-byte). Our target implementation is done on the iPad2 and iPhone 5; they include PowerVR SGX543MP2 and SGX543MP3 [3] embedded GPU respectively. The GPU has 8 texture units with 4096 × 4096 dimensions.

For implementing the application on the embedded GPU, we need to partition the data to fit into available GPU texture memory, and processing limitations. We describe below our proposed design, highlighting reasons behind choices. For our implementation, we choose up to 1024 images to predict from the Face Recognition Technology (FERET) database [12]. Each image is cropped into \(256 \times 256\) and converted into grey scale. Thus, we need to compute:

\[y_{1024 \times 1} = x_{64k \times 1}^T W_{64K \times 1024}\]

A major performance limitation with this calculation is the relatively small output vector size with respect to the input vector size. The solution to this problem is to use a larger input vector size while maintaining the output size. This is achieved by padding the input vector with zeros. The padded vector is then multiplied by the projection matrix to obtain the final result.
vector, especially when the database size is small, for example 64. A straightforward implementation would map \( y \) into an output pixel, thereby having only 64 parallel threads processing 1K data element each for the database size of 64. This significantly affects performance. Larger database sizes significantly increases the degree of parallelism, however, usually large number of threads is required to achieve significant speedups.

To handle this limitation, we generate partial results so as to increase the degree of parallelism; the partial results are further processed by the embedded CPU, however the operations are mainly additions requiring negligible processing time.

Another limitation is the 2-dimensional restriction on the texture memory; we cannot directly map the 64K input vector, for instance, into a 1D texture memory. Therefore to increase the degree of parallelism, as well as to facilitate mapping into the texture dimensions, we divide our vectors and matrix into chunks. Each chunk represents a contiguous group of values. Vector matrix operations are performed on chunks rather than individual elements; this results in an increased degree of parallelism, proportional to the number of chunks.

4.2.1. The Input Vector Mapping

![Figure 3. Mapping from 1D probe image into vec4 2D texture images for processing.](image)

The input vector is 64K long. Fig. 3 illustrates our mapping for 32 chunks. The input is mapped into \( 512 \times 32 \) matrix; the elements of which are 16-bit float vector elements (vec4); thereby a row of 512 vec4 holds 2048 elements; each row of the matrix is a chunk or more.

The total number of chunks is the \( \text{chunks\_per\_row} \times \text{rows\_per\_image} \). The dimensions of the input matrix are therefore:

\[
\begin{align*}
X_{\text{cols}} &= \text{chunks\_per\_row} \times \text{image\_length}/\text{number\_chunks}/4 \\
X_{\text{rows}} &= \text{number\_chunks}/\text{chunks\_per\_row}
\end{align*}
\]

4.2.2. The Projection Matrix Mapping

![Figure 4. Mapping the projection matrix, \( W \), into a texture-buffer.](image)

Fig. 4 illustrates the mapping for the projection matrix assuming 18 chunks (\( \text{chunk\_rows} = 3 \) and \( \text{chunks\_per\_row} = 6 \)). The matrix is transposed to utilise the vectorization parallelisation of the embedded GPU. The transposed matrix is represented as a texture of dimensions \( 512 \times 2048 \) of vec4 elements. Generally, the dimensions of the matrix are given by:

\[
\begin{align*}
W_{\text{cols}} &= \text{chunks\_per\_row} \times \text{image\_length}/\text{number\_chunks}/4 \\
W_{\text{rows}} &= \text{number\_chunks} \times \text{number\_images}/\text{chunks\_per\_row}
\end{align*}
\]

4.2.3. Output Vector Mapping and Computation

![Figure 5. Mapping the output into framebuffer.](image)

Fig. 5 illustrates the mapping of the output vector assuming 32 chunks and 64 images. The vector is mapped into a \( 32 \times 64 \) matrix of vec4. The matrix is allocated into the framebuffer. Every row represents the dot product partial values for a whole image; the partial values are grouped according to the chunk used; in other words, each chunk is reduced into
The main dot-product operation is implemented by successive vector multiply-add operations for each chunk. The shader code is provided in Appendix A.

5. EXPERIMENTAL EVALUATION

To evaluate our system, we have developed an implementation using the iOS platform. In particular we consider the iPad2 and iPhone5 systems. The iPad2 is based on dual-core ARM Cortex-A9 CPU and PowerVR SGX543MP2 GPU, and was introduced in 2011. The iPhone5 is a more recent system introduced on 2012, and is based on a dual-core Apple proprietary processor, A6 [1], PowerVR543MP3 GPU. Both systems have 1GB of RAM and supports OpenGL ES 2.0, as mentioned before. We utilised iOS 6 as it supports a private power measurement library [16].

As a benchmark, we used the grey-scale test from the FERET [12] face database. The test uses two face sets, the ‘gallery’ and the ‘probe’ sets. The gallery and probe sets contain 984 faces each. The gallery basically contains a reference set of faces, and the probe set contains the same subjects faces in the gallery set but taken with a different pose and, possibly, date. The benchmark serves as a means to evaluate performance and power of our proposed system. Both the serial and the parallel version resulted in the same number of recognised faces; both recognised 702 faces out of the 984 in the gallery. The serial version uses the standard optimised OpenCV implementation for Fisherfaces, thereby validating the correctness of our parallel implementation.

For the serial implementation, we compiled the standard optimised OpenCV implementation using the LLVM optimising compiler within the Apple Xcode development framework. We compared the serial implementation with a straight unoptimised forward version, and found around 5x speedup is obtained mainly due to caching effect; the straightforward implementation has row major accesses, which significantly affected performance. We did not explore further optimisation using Neon instruction set, we leave this study for future work.

The purpose of the implementation is to analyse the performance and power efficiency of the GPU based system. Our performance metric is the total execution time for the whole process of recognition. That involves projecting the input face into the eigenface space as well as matching the image against the training set images. The time includes all communications among the host CPU and the GPU, as well as other CPU overheads. Similarly, we report power usage for the whole application and the whole underlying hardware system for the iPad2 and iPhone5.

The parameters of our system include the following:

- Database Size: We vary the size of the database to be 64, 128, 256, 512, and 1024 (we padded the empty faces in the 1024 with empty faces).
- Number of Chunks: We vary the number of chunks to be 16, 32, 64, 128, 256, 512.

The database size and the number of chunks determine the total number of threads in the system. In particular

\[
\text{Number of Threads} = \text{Database Size} \times \text{Number of Chunks}
\]

Clearly increasing the number of chunks decrease the granularity of threads and vice versa. Increasing the database size increases the number of threads while keeping the granularity of threads constant.

![Figure 6. Speedup Curves for the iPad2.](image)

Fig. 6 shows speedup curves of the iPad2. The speedup is calculated relative to the CPU case. The curves are plotted against changing the number of chunks, and each curve represents a particular database size. For up to 128 image database, the performance shows slight improvement when increasing the number of chunks, then it degrades. The reason is that with increasing the number of chunks, the degree of parallelism increases as well as the serial reduction operation; therefore the first part of the curve parallelism is dominant, and the second part the serial reduction operation is dominant.

For database sizes larger than 128 images a significant performance jump occurs. This is mainly due to significant degradation in the serial processing time (rather than improvement in the parallel GPU version); we expect the main reason for that is caching effects as the work set size is significantly increased. The GPU does not suffer from that as the main GPU memory is software managed and is large enough to hold the whole database images. The performance still follows similar trend as the other cases, where it peaks due to the trade-off between parallelism and increased serial work.

The serial implementation has a linear execution complexity with respect to the number of images (problem size). In fact we have observed a linear increase in the serial execution
time of rate 1.77 for up to 128 images case. The rate suddenly jumps to 5.3 for the 256 case, then it becomes steady at the rate of 2. We presume the initial rate of 1.77 is less than 2 due other initialisation overheads that do not depend on the problem size; for larger sizes the overhead diminishes and the the rate stabilises at 2. To negate the cache effect (to have a fair comparison), we scale the serial execution times by a factor of 2 from sizes greater than 128. Fig. 7 reports the corresponding results against the total number of threads. Performance reaches up to 5x for 1024 images, and 32768 threads. It is worth noting that the maximum number of possible threads are 16M threads.

It is worth noting that the absolute time for recognising a face in the 1024 case is 210ms, for the parallel version.

Figures 8 and 9 show the same experiment but for the iPhone5\(^1\). The iPhone5 has a larger cache and did not suffer from the cache penalty with increasing the database sizes. Moreover, the speedup reached up to 10x for the 1024 case.

\(^1\)The results for the 512 database was obtained in iOS6 due to the unavailability of iOS6 by the time we prepared the final version of the paper; however, we observed negligible effects on the results for the other previously obtained on iOS6.
optimised serial kernel with low precision arithmetic as op-
data, in high speed. Also, future work might consider hand
utilise host processor multicore capability to uncompress the
achieved. Another strategy is to compress the loaded data, and
per image. For surveillance application, such setting can be
input image. We would thus require 6.6 faces to recognise
proach would be to recognise more than one face in the same
faces. We therefore would need to overlap the loading time
be to divide the projection matrix into parts of 1024 eigen-
tions, such as main memory, is necessary.
However, careful coordination and observing resource limita-
tions, such as main memory. Moreover, further parallelisation of the reduc-
tion operation would be necessary for larger images sets. Fu-
ture work will consider utilising the multicore parallelism in
embedded CPU to allow for compressing the loading of the
projection matrix allowing for better scalability. Also, adap-
tive training can be pursed to allow for further enhancement
to the recognition application.

5.0.4. Scaling Aspects
The current prototype can recognise up to 1024 images as
constrained by the size of texture-buffer size. Generally, the
system can be scaled through utilising more than one texture-
buffer, and also overlapping processing with communication.
However, careful coordination and observing resource limita-
tions, such as main memory, is necessary.

One important issue that we faced in our prototype is han-
dling the loading for the projection matrix, W. OpenCV gen-
erally utilise a textual representation; we converted the matrix
file into a binary representation, and reduced the precision
form 64-bit to 16-bit to match the hardware capability. That
effectively reduced the file size 12x. Such precision is enough
for recognising 1024 face (validated by comparison with a se-
rial 64-bit implementation), but further analysis would be re-
quired for larger database set. However, we are already aware
of recent embedded GPUs that supports 32-bit, that would
clearly allow for higher scalability.

For the iPhone5 system, the corresponding matrix loading
time is 1380ms, recognising a face consumes 210ms on aver-
age. Thus to double the number of images, one strategy would
be to divide the projection matrix into parts of 1024 eigen-
faces. We therefore would need to overlap the loading time
with computation. Since recognition time is faster, one ap-
proach would be to recognise more than one face in the same
input image. We would thus require 6.6 faces to recognise
per image. For surveillance application, such setting can be
achieved. Another strategy is to compress the loaded data, and
utilise host processor multicore capability to uncompress the
data, in high speed. Also, future work might consider hand
optimised serial kernel with low precision arithmetic as op-
posed to the used 32-bit.

6. CONCLUSIONS
In this paper we have developed a scalable facial recogni-
tion system on the mobile platform exploiting embedded
GPUs. A prototype is implemented for recognising the whole
set of 984 test set of the FERET face database. The system
provides a parallel implementation of the Fisherfaces face
recognition method, resulting in 32K threads, and achieving
10x reduction in time, and 7.2x reduction in energy when
compared with the serial embedded CPU implementation.

Future work would extend the prototype to exploit the vast
storage of mobile phones through caching images into the
main memory. Moreover, further parallelisation of the reduc-
tion operation would be necessary for larger images sets. Fu-
ture work will consider utilising the multicore parallelism in
embedded CPU to allow for compressing the loading of the
projection matrix allowing for better scalability. Also, adap-
tive training can be pursed to allow for further enhancement
to the recognition application.

ACKNOWLEDGEMENTS
This work is funded by ITIDA ITAC Fellowship research

REFERENCES
[1] AnandTech - the iPhone 5’s A6 SoC: not A15 or A9, a custom apple core instead. http://www.anandtech.com/show/6292/iphone-5-
yogy Roadmap For Semiconductors, 2011.
[5] AKENINE-MOLLER, T., AND STROM, J. Graphics pro-
cessing units for handhelds. Proceedings of the IEEE
[6] BELHUMEUR, P. N., HESPANHA, J. P., AND KRIEG-
MAN, D. J. Eigenfaces vs. fisherfaces: Recognition us-
ing class specific linear projection. Pattern Analysis
and Machine Intelligence, IEEE Transactions on 19, 7
for general-purpose computing - a case study of face
recognition on smartphones. In 2011 International Sym-
posium on VLSI Design, Automation and Test (VLSI-

Table 1. Power Consumption Comparison
<table>
<thead>
<tr>
<th>Power Metric</th>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Instr. Tool</td>
<td>18/20</td>
<td>18/20</td>
</tr>
<tr>
<td>PowerGremlin (Watt)</td>
<td>1.7</td>
<td>1.2</td>
</tr>
</tbody>
</table>
The shader code for performing the dot product operation is listed above. The kernel operates on four elements at the same time (vec4), and each image chunk is reduced into one vec4 value. The algorithm below computes the DotProduct of the input eigenface (line 15); the chunks are stored in a column-wise orientation. Since chunks can occupy more than one row in the face matrix X, chunk_yoffset is assigned to the vertical displacement within X (line 16). Note that rows_per_image and chunks_per_row are constants representing the number of rows in X per image and chunks inside each row, respectively. Note that the total number of chunks is chunks_per_row × chunks_per_row. The corresponding row in W is stored into row ‘basey’, which is the start row index of the corresponding group (line 17) offseted by the chunk_yoffset within the chunk (line 18).

The main loop (line 21) iterates over the vec4 elements of the chunk given by the constant chunk_qsize; the iterator k starts at the column address at W. The body of the loop (line 23) performs the dot product operation over vec4 values storing partial result on sum. Finally, the total sum is stored into the output matrix Z.

A DOT PRODUCT KERNEL

The shader code for performing the dot product operation above is listed above. The kernel operates on four elements at the same time (vec4), and each image chunk is reduced into one vec4 value. The algorithm below computes the DotProduct of the input eigenface (line 15); the chunks are stored in a column-wise orientation. Since chunks can occupy more than one row in the face matrix X, chunk_yoffset is assigned to the vertical displacement within X (line 16). Note that rows_per_image and chunks_per_row are constants representing the number of rows in X per image and chunks inside each row, respectively. Note that the total number of chunks is chunks_per_row × chunks_per_row. The corresponding row in W is stored into row ‘basey’, which is the start row index of the corresponding group (line 17) offseted by the chunk_yoffset within the group (line 18).

The main loop (line 21) iterates over the vec4 elements of the chunk given by the constant chunk_qsize; the iterator k starts at the column address at W. The body of the loop (line 23) performs the dot product operation over vec4 values storing partial result on sum. Finally, the total sum is stored into the output matrix Z.

procedure DotProduct (X, W, x, y, Z)
X : in FaceMatrix
W : in ProjectionMatrix
x, y : in RenderCoordinates
Z : out OutputMatrix
pic_index : Integer
chunk_per_b_row : in Integer
chunk_index : Integer
chunk_yoffset : Integer
basex : Integer
basey : Integer
sum : vec4
begin
eigenface_index := y
chunk_index := x
chunk_yoffset := chunk_index / chunks_per_row
basey := eigenface_index * rows_per_image
basey := basey + chunk_yoffset
base := (mod(chunk_index, chunks_per_row)) * chunk_qsize
sum := vec4(0, 0, 0, 0)
for (k = basey; k < basey + chunk_qsize; k++)
begin
sum := sum + X[k, chunk_yoffset] * W[k, basey]
end
Z[x, y] := sum
end