

How to accelerate graphics and computation

OpenCL

IN ACTION

Matthew Scarpino

ÙÇË ÛŠÒÂÏPÆÛVÒÛ





OpenCL in Action

by Matthew Scarpino

Chapter %&

Copyright 2012 Manning Publications

brief contents

PART 1 FOUNDATIONS OF OPENCL PROGRAMMING.....1

- 1 ■ Introducing OpenCL 3
- 2 ■ Host programming: fundamental data structures 16
- 3 ■ Host programming: data transfer and partitioning 43
- 4 ■ Kernel programming: data types and device memory 68
- 5 ■ Kernel programming: operators and functions 94
- 6 ■ Image processing 123
- 7 ■ Events, profiling, and synchronization 140
- 8 ■ Development with C++ 167
- 9 ■ Development with Java and Python 196
- 10 ■ General coding principles 221

PART 2 CODING PRACTICAL ALGORITHMS IN OPENCL.....235

- 11 ■ Reduction and sorting 237
- 12 ■ Matrices and QR decomposition 258
- 13 ■ Sparse matrices 278
- 14 ■ Signal processing and the fast Fourier transform 295

PART 3 ACCELERATING OpenGL WITH OPENCL319

15 ■ Combining OpenCL and OpenGL 321

16 ■ Textures and renderbuffers 340

12

Matrices and QR decomposition

This chapter covers

- Implementing matrix transposition and multiplication in OpenCL
- Understanding and coding the Householder transformation
- Factoring matrices with the QR decomposition algorithm

From physics and engineering to economics and sociology, there is no getting away from matrices. These mathematical structures can represent systems of equations, statistical data, DNA sequences, and the distribution of stresses within an object. Matrices have been used to structure data for centuries, and new applications appear on a regular basis.

Just as there are many uses for matrices, there are also many different ways to analyze them. But not all matrices are easy to work with. Mathematicians frequently find it necessary to factor a disordered matrix into matrices that are easy to analyze, and then perform their operations on the factors. This factorization is conceptually

similar to factoring an integer into its prime divisors or factoring a polynomial into its roots.

One of the most popular methods of factoring a matrix is called the QR decomposition. This factors a matrix into two matrices whose qualities make them simple to analyze and manipulate. The goal of this chapter is to explain the theory behind QR decomposition and show how it can be implemented with OpenCL.

There are a number of ways to compute QR decomposition, but this presentation will focus on using Householder transformations, which reflect vectors across a plane or hyperplane. But before we examine these transformations, it's important that you have a solid understanding of two fundamental matrix operations: transposition and multiplication. This chapter will present transposition first.

12.1 Matrix transposition

Taking the transpose of a matrix is one of the simplest operations in linear algebra. This section presents a brief overview of matrices, including their rows and columns, and then discusses how these rows and columns can be switched through matrix transposition. The last part of this section shows how this operation can be coded in OpenCL.

12.1.1 Introduction to matrices

A *matrix* is a rectangular arrangement of numbers. Matrices are represented graphically as a grid of numbers inside vertical bars. This is shown in figure 12.1.

In code, matrices are commonly represented by two-dimensional arrays, where the two dimensions identify the matrix's rows and columns. If a matrix has m rows and n columns, it's called an m -by- n matrix. If the number of rows equals the number of columns, it's called a *square matrix*.

Each row and column is a one-dimensional structure of numbers, and for this reason, we can refer to each row and column as a *vector*. These are mathematical vectors, not to be confused with the data types presented in chapter 4. We'll have much more to say about these vectors and their operations throughout this chapter.

The numbers that make up a matrix are called *elements*. Matrix notation gives each element a designation that identifies its row and column. In figure 12.1, the element c_{ij} belongs to the i^{th} row and the j^{th} column. If i equals j , then the element lies on an imaginary line called the matrix's *diagonal*, which runs from the upper left to the lower right.

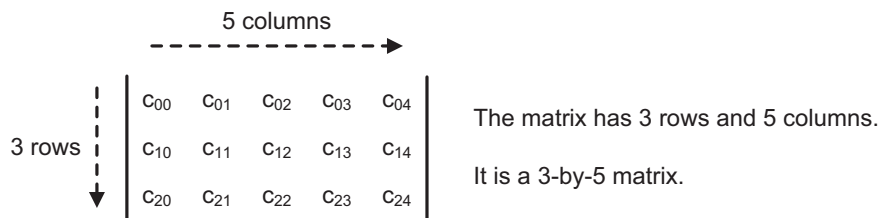


Figure 12.1 Matrix notation

Mathematicians have devised many categories for matrices, and one set of categories is based on the location of zeros within the matrix. If elements on the diagonal are nonzero and elements off the diagonal are zero, the matrix is a *diagonal matrix*. If the nonzero elements of a diagonal matrix all equal 1, the matrix is an *identity matrix*. If the elements above the diagonal, c_{ij} , equal the elements below the diagonal, c_{ji} , the matrix is a *symmetric matrix*.

If the overwhelming majority of elements are zero, the matrix is a *sparse matrix*, which the next chapter will discuss in detail. If the majority of elements, both on and off the diagonal, don't equal zero, the matrix is a *dense matrix*. This chapter focuses on dense matrices, particularly dense square matrices.

12.1.2 Theory and implementation of matrix transposition

The goal of computing a matrix's transpose is simple: to reflect each element across the diagonal so that each c_{ij} becomes c_{ji} . After a transpose, rows become columns and columns become rows. This is shown in figure 12.2, in which column 2 of the matrix becomes row 2. In text, you use T to denote a transposed matrix. For example, C^T is the transpose of the matrix C .

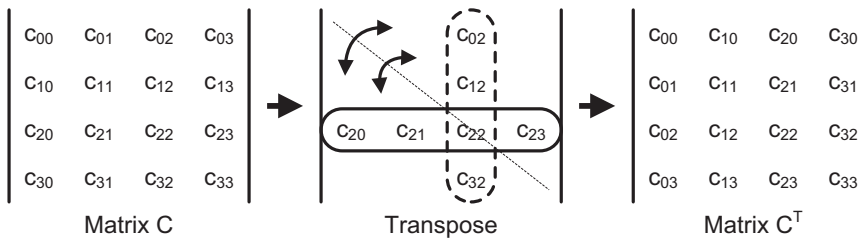


Figure 12.2 Matrix transposition

The following listing presents an in-place implementation of the transpose in OpenCL. Each work-item is assigned a block containing 16 values in 4 `float4` vectors. If a block lies on the diagonal, the work-item will swap its rows and columns. If not, the work-item will swap elements with the block across the diagonal, replacing rows with columns.

Listing 12.1 Matrix transposition: `transpose.cl`

```
kernel void transpose(__global float4 *g_mat,
    __local float4 *l_mat, uint size) {
    __global float4 *src, *dst;

    int col = get_global_id(0);
    int row = 0;
    while(col >= size) {
        col -= size--;
        row++;
```

Find row/column
location



```

}
col += row;
size += row;

src = g_mat + row * size * 4 + col;
l_mat += get_local_id(0)*8;
l_mat[0] = src[0];
l_mat[1] = src[size];
l_mat[2] = src[2*size];
l_mat[3] = src[3*size];

if(row == col) {
    src[0] = (float4)(l_mat[0].x, l_mat[1].x,
                    l_mat[2].x, l_mat[3].x);
    src[size] = (float4)(l_mat[0].y, l_mat[1].y,
                       l_mat[2].y, l_mat[3].y);
    src[2*size] = (float4)(l_mat[0].z, l_mat[1].z,
                          l_mat[2].z, l_mat[3].z);
    src[3*size] = (float4)(l_mat[0].w, l_mat[1].w,
                          l_mat[2].w, l_mat[3].w);
}
else {
    dst = g_mat + col * size * 4 + row;
    l_mat[4] = dst[0];
    l_mat[5] = dst[size];
    l_mat[6] = dst[2*size];
    l_mat[7] = dst[3*size];

    dst[0] = (float4)(l_mat[0].x, l_mat[1].x,
                    l_mat[2].x, l_mat[3].x);
    dst[size] = (float4)(l_mat[0].y, l_mat[1].y,
                       l_mat[2].y, l_mat[3].y);
    dst[2*size] = (float4)(l_mat[0].z, l_mat[1].z,
                          l_mat[2].z, l_mat[3].z);
    dst[3*size] = (float4)(l_mat[0].w, l_mat[1].w,
                          l_mat[2].w, l_mat[3].w);
    src[0] = (float4)(l_mat[4].x, l_mat[5].x,
                    l_mat[6].x, l_mat[7].x);
    src[size] = (float4)(l_mat[4].y, l_mat[5].y,
                       l_mat[6].y, l_mat[7].y);
    src[2*size] = (float4)(l_mat[4].z, l_mat[5].z,
                          l_mat[6].z, l_mat[7].z);
    src[3*size] = (float4)(l_mat[4].w, l_mat[5].w,
                          l_mat[6].w, l_mat[7].w);
}
}
}

```

↑ Find row/column location

Process block on diagonal

Process block off diagonal

The host application sets the dimensionality of each work-item to 1 instead of 2, and this may seem odd at first. But as shown in figure 12.3, you don't need a work-item for every block in the matrix. You only need work items to process blocks on or above the diagonal.

The number of blocks that need to be processed is $n(n+1)/2$, where n is the number of blocks in a row. For example, a 256-by-256 matrix contains 64-by-64 blocks, so the host will generate $64(64+1)/2 = 2,080$ work-items to execute the kernel.

0	1	2	3	4	5	6	7
	8	9	10	11	12	13	14
		15	16	17	18	19	20
			21	22	23	24	25
				26	27	28	29
					30	31	32
						33	34
							35

8 blocks per row

$8(8+1)/2 = 36$ blocks
assigned to work-items

Figure 12.3 Work-items and the transpose

Transposition is a crucial operation in linear algebra, and any professional library of matrix routines will contain a transpose routine. One important operation that makes use of the transpose is matrix multiplication, which is the topic of the next section.

12.2 Matrix multiplication

When a company wants to show off its new high-performance computing system, they'll frequently have it perform matrix multiplication. It's easy to see why. Matrix multiplication requires high-speed number crunching and high-speed data transfer, but few decisions. It's also a vital building block of many large-scale linear algebra routines. If a supercomputer is performing any large-scale linear algebra operation, the odds are that a great deal of its time is spent multiplying matrices.

This section presents the theory of matrix multiplication, which relies on an important vector operation called the dot product. Then we'll examine how to implement multiplication with OpenCL.

12.2.1 The theory of matrix multiplication

The product of two matrices, A and B , is obtained by multiplying each row of A with each column of B . This multiplication is implemented using the dot product, which was briefly discussed in chapter 5. The dot product multiplies the corresponding elements of two vectors and returns the sum of the products. For example, if vector p contains $[p_0, p_1, p_2, p_3]$ and vector q contains $[q_0, q_1, q_2, q_3]$, their dot product can be computed as follows:

$$p \cdot q = p_0q_0 + p_1q_1 + p_2q_2 + p_3q_3$$

In OpenCL, the dot product of two vectors is computed using the `dot` function discussed in chapter 5. This and the next chapter will make extensive use of this function.

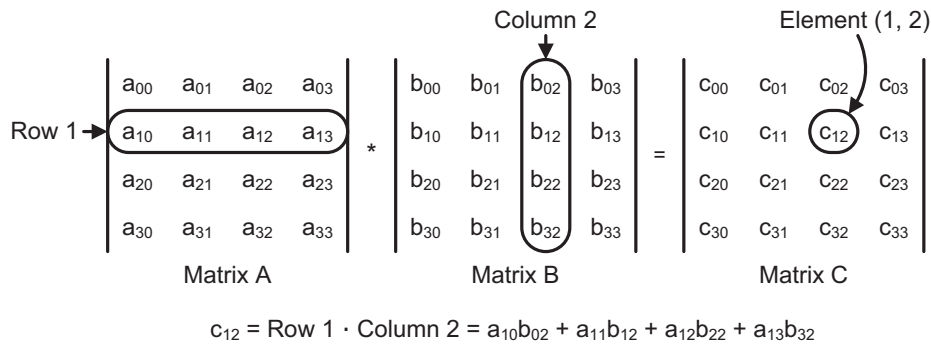


Figure 12.4 Matrix multiplication

Figure 12.4 shows the matrix multiplication of two 4-by-4 matrices, A and B , and their product matrix C . The c_{12} element of the C matrix is obtained by multiplying row 1 of A and column 2 of B .

Each element in C is computed in a similar manner. In this example, the full matrix multiplication requires 16 dot products—each of the four rows of A must be multiplied by each of the four columns of B .

A and B are square matrices in this example, but nonsquare matrices can also be multiplied. But the dot product requires vectors of equal length, so the rows of the first matrix must have the same size as the columns of the second. In other words, if the first matrix has n columns, the second matrix must have n rows. Taking this a step further, if the first matrix has dimensions m by n and the second matrix has dimensions n by p , the product matrix will have dimensions m by p .

We can generalize the multiplication of rectangular matrices as follows: if matrix $C = AB$, A has dimensions m by n , and B has dimensions n by p , element c_{ij} can be computed as follows:

$$c_{ij} = \sum_{k=0}^{n-1} a_{ik}b_{kj}$$

Matrix multiplication is *not* commutative— AB does not equal BA . But matrix multiplication is *associative*: $(AB)C = A(BC)$. This property will become important later on when we look at the QR decomposition.

12.2.2 Implementing matrix multiplication in OpenCL

Matrix multiplication isn't hard to implement in code, but there are many details to keep in mind. The kernel needs to know the dimensions and the data types of the elements, and also how the elements are stored in memory. If the matrix data is stored in *row-major* order, the elements will be stored row by row. That is, the elements of row 0 will be followed by the elements of row 1, then row 2, and so on. If the data is stored in *column-major* order, the elements will be stored column by column. The elements of column 0 will be followed by the elements of column 1, then column 2, and so on.

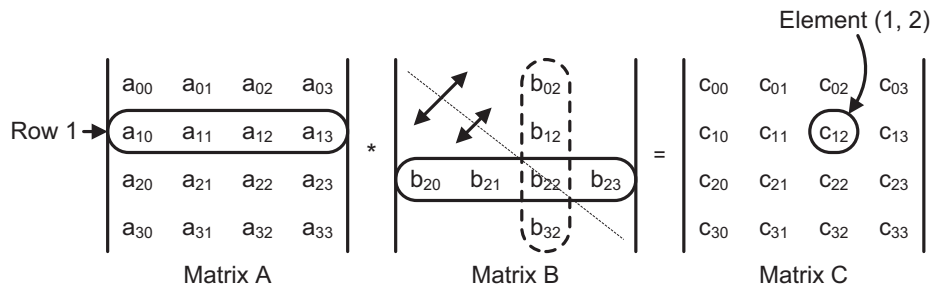


Figure 12.5 Matrix multiplication after transposition

The development kits released by Nvidia and AMD both contain OpenCL code that multiplies matrices. In both cases, the multiplication is based on scalars—input elements are multiplied one at a time. This chapter presents a different implementation. The code in the Ch12/matrix_mult project combines elements into vectors and performs the dot product with the dot function.

This presents a problem. Matrix multiplication requires dot products of rows and columns, and if the matrices are stored in row-major format (the usual format), you can't load multiple elements from a single column at a time. But you can fix this by taking the transpose of the second matrix. Figure 12.5 presents the same multiplication as in figure 12.4, but because B is transposed, the elements in A and B can both be accessed by row.

If you look through the matrix_mult project in the Ch12 folder, you'll see that the host application (matrix_mult.c) enqueues two kernels: one to transpose the second matrix and one to multiply the two matrices together. The following code presents the kernel that performs the actual multiplication.

NOTE The dot products are computed with the dot function, which was briefly discussed in chapter 5.

Listing 12.2 Matrix multiplication: matrix_mult.cl

```
kernel void matrix_mult(__global float4 *a_mat,
    __global float4 *b_mat, __global float *c_mat) {
    float sum;

    int num_rows = get_global_size(0);
    int vectors_per_row = num_rows/4;

    int start = get_global_id(0) * vectors_per_row;
    a_mat += start;
    c_mat += start*4;

    for(int i=0; i<num_rows; i++) {
        sum = 0.0f;
        for(int j=0; j<vectors_per_row; j++) {
            sum += dot(a_mat[j],
                b_mat[i*vectors_per_row + j]);
        }
    }
}
```

Find input/output
row addresses

Multiply A row
by B rows

```

    }
    c_mat[i] = sum;
  }
}

```

↑
Multiply A row
by B rows

This kernel doesn't access local memory because there are no intermediate results to store. This code executes quickly, but it's important to remember that it expects the second matrix, `b_mat`, to be in column-major order.

Matrix multiplication is an important part of many matrix operations, including QR decomposition. Another crucial subroutine in this chapter's implementation of QR decomposition is the Householder transformation, which is the topic of the next section.

12.3 The Householder transformation

Most discussions of vector operations include addition, subtraction, and multiplication, but vector *reflection* is also a critical operation in many algorithms. The concept is simple: given an input vector and a vector perpendicular to a surface, the goal is to find the reflection of the input vector across the surface. The procedure for computing this reflection is called the *Householder transformation*, and this section will examine this transformation in detail. But first, it's important to be familiar with the theory of vector projection.

12.3.1 Vector projection

The dot product of two vectors provides an idea of their relative directions. If the product is positive and large compared to the vectors' lengths, it implies that the two vectors are pointing in similar directions. If the dot product is negative, it implies that the two vectors are pointing in different directions. If the dot product is 0, it means the two vectors point at right angles to one another.

The concept of the vector projection allows you to be more precise about the similarity between two vectors. A vector projection is the portion of one vector that points in the same direction as a second vector. In figure 12.6, vector b is split into two components: a component called p , which points in the same direction as a , and q , which points in a direction orthogonal to a . It should be clear that $b = p + q$.

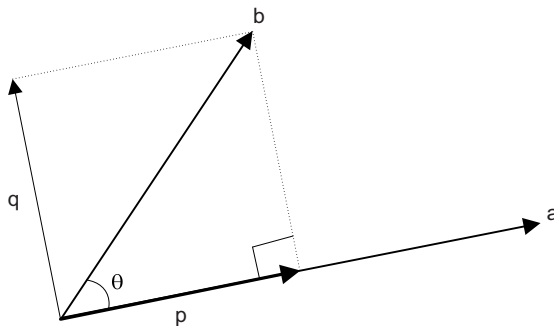


Figure 12.6 Vector projection

In this diagram, p is the vector projection of b on a . The larger p is, the more similar b is to a . Using trigonometry, you can compute the length of p as $|b|\cos\theta$. To make p point in the direction of a , you need to multiply it by a 's *unit vector*. This is obtained by dividing a by its length, denoted $|a|$. The following equation shows the result of the multiplication:

$$p = |b|\cos(\theta)\frac{a}{|a|}$$

The vector projection would be easy to compute if weren't for the cosine. Thankfully, a relationship exists between the cosine of the angle between two vectors and the vectors' dot product. The proof is lengthy, but the result is as follows:

$$\cos(\theta) = \frac{a \cdot b}{|a||b|}$$

By placing this into the previous equation, you can arrive at a more workable expression for p :

$$p = |a|\frac{a \cdot b}{|a||b||a|} = \frac{a \cdot b}{|a|} \frac{a}{|a|} = \frac{a \cdot b}{|a|^2} a$$

In general, the vector projection of vector b on vector a is expressed with the term $\text{proj}_a b$. $\text{proj}_a b$ has the same direction as a , and $b - \text{proj}_a b$ is orthogonal to a .

12.3.2 Vector reflection

Many algorithms in linear algebra require vector reflection, and the reason for this will become clear later in this chapter. Figure 12.7 presents a simple two-dimensional case of how this reflection works. You'll start with two vectors: x and u . u is perpendicular to M (which stands for mirror). The goal is to find x' , the vector obtained by reflecting x in M . Note that, in two dimensions, M is simply a line. It's a plane in three dimensions and a hyperplane in four or more dimensions.

To find x' in terms of x and u , you need to take vector projections. Figure 12.8 shows how this works. x is split into p and q , where p is the vector projection of x on u and q is orthogonal to p . Similarly, x' is split into p' and q' , where p' is the vector projection of x' on u and q' is orthogonal to p' .

Figure 12.8 makes clear the relationships between p and p' and q and q' . Because x' equals the sum of p' and q' , you can compute it as shown on the next page.

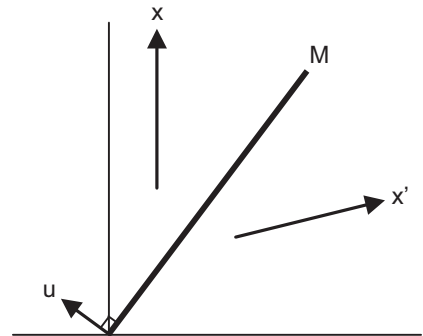


Figure 12.7 Vector reflection

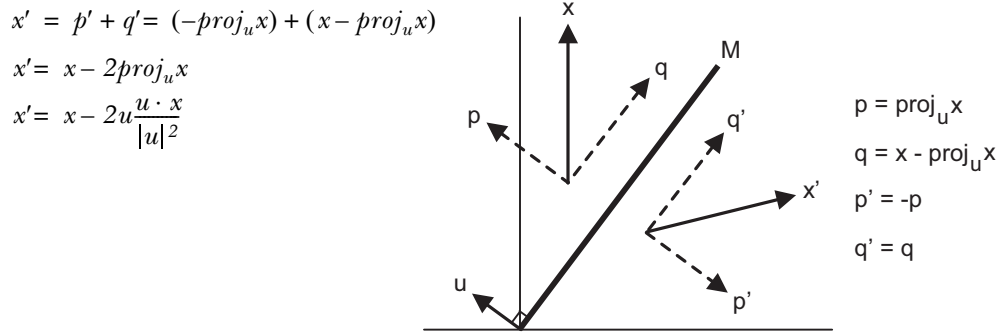


Figure 12.8 Vector reflection in terms of vector projections

This final equation gives a clear relationship between x' , x , and u . This equation will help you compute the QR decomposition of a matrix. But before we discuss the decomposition algorithm, we need to come to terms with the outer product and how it can be used to form Householder matrices.

12.3.3 Outer products and Householder matrices

A vector can be thought of as a matrix with a single row or a single column. By default, mathematicians treat vectors as matrices with a single column, and such vectors are called *column vectors*. In contrast, transposed column vectors are considered matrices with a single row, or *row vectors*.

With this new interpretation of vectors, we can arrive at a new interpretation of the dot product. Instead of multiplying vectors, the dot product can be thought of as multiplying a 1-by- n matrix and an n -by-1 matrix. The result is a 1-by-1 matrix, better known as a *scalar*. As a result, we can refer to the dot product using matrix terminology: $a^T b$ instead of $a \cdot b$. The following equation makes this clearer:

$$a \cdot b = a^T b = \begin{bmatrix} a_0 & a_1 & a_2 & a_3 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = a_0 b_0 + a_1 b_1 + a_2 b_2 + a_3 b_3$$

Using this notation, we can arrive at a new type of product called the *outer product*. Instead of multiplying a row vector by a column vector, this product reverses the operation and computes ab^T instead of $a^T b$. Despite the similar appearance, the result of the outer product is significantly different than that produced by the dot product. The dot product multiplies a 1-by- n matrix with an n -by-1 matrix and produces a 1-by-1 matrix. The outer product multiplies an n -by-1 matrix with a 1-by- n matrix and produces an n -by- n matrix. This is shown in the following equation:

$$ab^T = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} b_0 & b_1 & b_2 & b_3 \end{bmatrix} = \begin{bmatrix} a_0b_0 & a_0b_1 & a_0b_2 & a_0b_3 \\ a_1b_0 & a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_0 & a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_0 & a_3b_1 & a_3b_2 & a_3b_3 \end{bmatrix}$$

You can use this vector-as-matrix interpretation to manipulate the equation for vector reflection. If you replace $u \bullet x$ with $u^T x$, the new relationship is as follows:

$$x' = x - 2u \frac{u \bullet x}{|u|^2} = x - 2 \frac{u(u^T x)}{|u|^2}$$

As mentioned earlier, matrix multiplication is associative, so $A(BC) = (AB)C$. You can use this relationship to change $u(u^T x)$ in the reflection equation to $(uu^T)x$. This provides the following relationship:

$$x' = x - 2 \frac{u(u^T x)}{|u|^2} = x - 2 \frac{(uu^T)x}{|u|^2} = x - 2 \frac{uu^T}{|u|^2} x$$

Instead of finding x' with a dot product, you now need to compute an outer product, which is more difficult. But there is a good reason to do this. If you factor the vector x out of the equation, you can arrive at the following relationship:

$$x' = \left(I - 2 \frac{uu^T}{|u|^2} \right) x = Px$$

In this equation, I is the identity matrix and the term inside the parentheses is a matrix. This matrix is commonly denoted by P , and when P premultiplies a vector x , the result is the reflection of x through the hyperplane perpendicular to u . This procedure was conceived by Alston Householder; u is called the Householder vector, and P is called the Householder matrix. The reflection operation represented by P is called the Householder transformation.

When a vector is reflected twice, the result will be the vector itself. That is, $P(Px) = x$ for all x . P is its own inverse and any matrix with this property is called *involutary*.

One more point about the Householder transformation needs to be addressed. The preceding discussion explained how to find x' given x and u , but what if you start with x and x' and want to find a vector with u 's direction? The answer is surprisingly simple: $u = x - x'$. Figure 12.9 presents this graphically, using the same x and x' vectors from earlier figures.

This result may not be immediately obvious, but remember that x and x' are symmetrical about M , and that u is perpendicular to M . The u vector in figure 12.9 has a different length than the u vector in previous figures, but this isn't a concern—the only requirement you have for u is that its

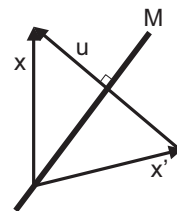


Figure 12.9 Finding the reflection vector u

direction be perpendicular to M . Also, the length of u is divided out as part of the Householder transformation.

12.3.4 Vector reflection in OpenCL

Implementing vector reflection in code is a straightforward process, consisting of three main steps:

- 1 Multiply u by $-\text{sqrt}(2)/|u|$.
- 2 Compute the matrix uu^T .
- 3 Add the identity matrix, I , to the resulting matrix.

The next listing demonstrates how this is accomplished. x_vec equals $[1.0, 2.0, 3.0, 4.0]$ and u equals $[0.0, 5.0, 0.0, 0.0]$. Because u is perpendicular to the x - z - w hyperplane, the reflection of x_vec can be determined by inspection: $[1.0, -2.0, 3.0, 4.0]$.

Listing 12.3 Vector reflection: `vec_reflect.cl`

```
__kernel void vec_reflect(float4 x_vec, float4 u,
    __global float4* x_prime) {
    float4 p_mat[4];
    u *= M_SQRT2_F/length(u);
    p_mat[0] = (float4)(1.0f, 0.0f, 0.0f, 0.0f)
        - (u * u.x);
    p_mat[1] = (float4)(0.0f, 1.0f, 0.0f, 0.0f)
        - (u * u.y);
    p_mat[2] = (float4)(0.0f, 0.0f, 1.0f, 0.0f)
        - (u * u.z);
    p_mat[3] = (float4)(0.0f, 0.0f, 0.0f, 1.0f)
        - (u * u.w);
    x_prime[0].x = dot(p_mat[0], x_vec);
    x_prime[0].y = dot(p_mat[1], x_vec);
    x_prime[0].z = dot(p_mat[2], x_vec);
    x_prime[0].w = dot(p_mat[3], x_vec);
}
```

Compute
Householder matrix

Reflect
 x to x'

If you're only interested in finding the reflection of a vector, you don't have to worry about the outer product and its matrix operations. It's much simpler to compute x' with the following equation:

$$x' = x - 2u \frac{u \cdot x}{|u|^2}$$

But many applications of the Householder transformation require finding both a vector's reflection and the Householder matrix corresponding to the reflection. One such application is the QR decomposition, which is the topic of the next section.

12.4 The QR decomposition

In linear algebra, a common task involves factoring a complex matrix into simpler matrices that are easier to analyze. This factorization is helpful when solving linear

$$A = \begin{pmatrix} | & | & | & | \\ x & x & x & x \\ | & | & | & | \\ x & x & x & x \\ | & | & | & | \\ x & x & x & x \\ | & | & | & | \\ x & x & x & x \\ | & | & | & | \\ c_0 & c_1 & c_2 & \end{pmatrix} \quad R = \begin{pmatrix} | & | & | & | \\ x & x & x & x \\ | & | & | & | \\ 0 & x & x & x \\ | & | & | & | \\ 0 & 0 & x & x \\ | & | & | & | \\ 0 & 0 & 0 & x \\ | & | & | & | \\ c_0' & c_1' & c_2' & \end{pmatrix}$$

Figure 12.10 Finding the reflection vectors u_k

systems, computing determinants, or obtaining eigenvalues. One of the most popular methods of factoring a matrix is called *QR decomposition*. This operates on a rectangular matrix A and produces two matrices with interesting properties. The first matrix, denoted Q , is *orthogonal*, which means its transpose equals its inverse. The second matrix, denoted R , is upper triangular, which means every element below the main diagonal equals zero.

Before we discuss how to compute Q , let's focus our attention on R . Figure 12.10 presents the goal: to transform the columns of A (denoted c_0 through c_{k-1}) into columns of R (c_0' through c_{k-1}'), where k identifies the number of columns in A . A is square in this figure, but QR decomposition can be applied to any rectangular matrix.

Computer scientists have found many ways to convert a square matrix like A into an upper-triangular matrix like R . One of the most common and most efficient methods relies on Householder transformations. This method computes the u_k vectors needed to transform the columns of A into the columns of R . Then, once the Householder vectors are obtained, this method computes their corresponding Householder matrices and multiplies them together. This combined matrix is Q .

The steps for finding Q and R are as follows:

- 1 Find u_0 that reflects c_0 into c_0' .
- 2 Transform each column of A with the u_0 reflection.
- 3 Find u_1 that reflects c_1 into c_1' .
- 4 Transform each column of A with the u_1 reflection.
- 5 Construct the Householder matrix P_k for each u_k vector.
- 6 Repeat steps 3–5 for columns up to $k-1$.
- 7 Multiply the Householder matrices to form Q .

The rest of this section will elaborate on these steps, and we'll walk through a QR decomposition of a 4-by-4 matrix. Then we'll look at how to implement QR decomposition in OpenCL.

12.4.1 Finding the Householder vectors and R

You've seen how to find a Householder vector u given a vector x and its reflection x' . You can use this method to determine the u_k vectors that transform the columns c_k in figure 12.10 to their reflections c_k' . But first, you need to obtain the nonzero elements in each c_k' . This isn't difficult as long as you remember that, because x' is the reflection of x , both vectors must have the same length.

An example will show how this works. The matrix A has three rows and three columns. You want to find two Householder vectors, u_0 and u_1 , that will transform the first two columns of A so as to make A upper triangular:

$$A = \begin{bmatrix} 13 & -17 & -10 \\ 4 & 18 & -32 \\ -16 & -8 & -24 \end{bmatrix}$$

You'll start with the leftmost column vector, which you'll call c_0 . You want the reflection, c_0' , to have one nonzero element on top and two zero elements below. To make sure that c_0 and c_0' have the same length, you'll set the nonzero element of c_0' equal to the length of c_0 . Then you can find u_0 by subtracting c_0' from c_0 . This is done as follows:

$$c_0' = \begin{bmatrix} |c_0| \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} 21 \\ 0 \\ 0 \end{bmatrix} \quad u_0 = c_0 - c_0' = \begin{bmatrix} 13 \\ 4 \\ -16 \end{bmatrix} - \begin{bmatrix} 21 \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} -8 \\ 4 \\ -16 \end{bmatrix}$$

Now that you have u_0 , you need to transform each column of A according to the reflection corresponding to u_0 . You can perform these reflections using an equation presented earlier:

$$c_k' = c_k - 2u_0 \frac{u_0 \cdot c_k}{|u_0|^2}$$

This transformation gives you a new A matrix. As desired, the first column has two zeros beneath the nonzero element:

$$A = \begin{bmatrix} 21 & -1 & 6 \\ 0 & 10 & -40 \\ 0 & 24 & 8 \end{bmatrix}$$

Now you want to transform the second column, c_1 , so that its bottom element equals zero. You can do this by setting the first element of the reflection, c_1' , equal to the first element of c_1 . Then, to make sure $|c_1'| = |c_1|$, you need to set the second element of c_1' equal to the length of the subvector containing the lower two elements of c_1 :

$$c_1' = \begin{bmatrix} -1 \\ \sqrt{(10)^2 + (24)^2} \\ 0 \end{bmatrix} = \begin{bmatrix} -1 \\ 26 \\ 0 \end{bmatrix} \quad u_1 = c_1 - c_1' = \begin{bmatrix} -1 \\ 10 \\ 24 \end{bmatrix} - \begin{bmatrix} -1 \\ 26 \\ 0 \end{bmatrix} = \begin{bmatrix} 0 \\ -16 \\ 24 \end{bmatrix}$$

Again, you need to transform each column of A according to the new reflection identified by u_1 . This gives you the matrix on the following page.

It's important to note that transforming c_0 with the reflection corresponding to u_1 leaves c_0 unchanged. This is because the dot product of c_0 and u_1 equals zero. When this dot product equals zero, the reflection produces the original vector. This means you

$$A = \begin{bmatrix} 21 & -1 & 6 \\ 0 & 26 & -8 \\ 0 & 0 & -40 \end{bmatrix} = R$$

don't have to transform columns that have already been transformed—once you compute a Householder vector u_k , you only have to find reflections for columns c_k and higher.

After this last set of reflections, A is upper triangular. This transformed version of A is the R matrix generated by the QR decomposition. Next, you'll see how to use the Householder vectors u_k to create the Q matrix.

12.4.2 Finding the Householder matrices and Q

Now that you've computed the Householder vectors u_k that transform A into R , you need to find the matrix Q that serves as the inverse of this transformation. That is, you want to find Q such that $Q^{-1}A = R$, or $A = QR$.

The previous section explained how to create a Householder matrix P from a Householder vector u . The relationship is given by

$$P = I - 2 \frac{uu^T}{|u|^2}$$

As discussed earlier, uu^T is an outer product that generates a square matrix. If u contains k elements, P has k rows and k columns.

In the example, you obtained R by transforming the column vectors of A with u_0 's reflection first and u_1 's reflection second. Therefore, R equals P_1P_0A . Because every Householder transformation is its own inverse, you can set Q equal to P_0P_1 . The following equation shows how Q is computed for the example:

$$\begin{aligned} Q &= P_0P_1 = \left(I - 2 \frac{u_0u_0^T}{|u_0|^2} \right) \left(I - 2 \frac{u_1u_1^T}{|u_1|^2} \right) \\ Q &= \left(I - \frac{2}{336} \begin{bmatrix} 64 & -32 & 128 \\ -32 & 16 & -64 \\ 128 & -64 & 256 \end{bmatrix} \right) \left(I - \frac{2}{832} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 256 & -384 \\ 0 & -384 & 576 \end{bmatrix} \right) \\ Q &= \begin{bmatrix} 0.61905 & 0.19048 & 0.76190 \\ 0.19048 & 0.90476 & 0.38095 \\ -0.76190 & 0.38095 & 0.52381 \end{bmatrix} \cdot \begin{bmatrix} 1.0 & 0 & 0 \\ 0 & 0.38462 & 0.92308 \\ 0 & 0.92308 & -0.38462 \end{bmatrix} \\ Q &= \begin{bmatrix} 0.61905 & -0.63004 & 0.46886 \\ 0.19048 & 0.69963 & 0.68864 \\ -0.76190 & -0.33700 & -0.55311 \end{bmatrix} \end{aligned}$$

If you multiply this Q matrix by the R matrix computed earlier, the product will be the original A matrix. This is because $QR = (P_0P_1)(P_1P_0A) = A$.

12.4.3 Implementing QR decomposition in OpenCL

The `qr` project in the `Ch12` folder computes the QR decomposition of a k -by- k matrix A , where k is set equal to 64 by default. The host application generates one work-item for every column of the matrix. These items all belong to the same work-group, so you can use the `barrier` command to synchronize their access to global and local memory.

Coding the QR decomposition isn't a simple process, but the main difficulty isn't computing R . Instead, the main question involves how to store and multiply the Householder matrices needed to form Q . If the matrices are large, you can't store each P_k separately. Instead, you need to initialize the Q matrix and update it as each new P_k is obtained.

For this reason, we'll split the discussion of the QR decomposition code into two parts. In the first part, we'll look at how to transform the first column of A and use it to initialize Q . In the second part, we'll examine how to transform the second through k^{th} columns of A and update Q with each new Householder matrix.

TRANSFORMING THE FIRST COLUMN AND INITIALIZING Q

The following listing presents the first part of the QR decomposition kernel. This computes the first Householder vector, u_0 , needed to transform the first column into a column of an upper-triangular matrix. Then it creates the Householder matrix, P_0 , from u_0 and sets Q equal to this matrix.

Listing 12.4 QR decomposition: qr.cl (part one)

```
__kernel void qr(__local float *u_vec, __global float *a_mat,
                __global float *q_mat, __global float *p_mat,
                __global float *prod_mat) {

    local float u_length_squared, dot;
    float prod, vec_length = 0.0f;

    int id = get_local_id(0);
    int num_cols = get_global_size(0);

    u_vec[id] = a_mat[id*num_cols];
    barrier(CLK_LOCAL_MEM_FENCE);

    if(id == 0) {
        for(int i=1; i<num_cols; i++) {
            vec_length += u_vec[i] * u_vec[i];
        }
        u_length_squared = vec_length;
        vec_length = sqrt(vec_length +
                          u_vec[0] * u_vec[0]);
        a_mat[0] = vec_length;
        u_vec[0] -= vec_length;
        u_length_squared += u_vec[0] * u_vec[0];
    }
    else {
        a_mat[id*num_cols] = 0.0f;
    }
    barrier(CLK_GLOBAL_MEM_FENCE);
}
```

← Load column into local memory

Find lengths of vectors

```

for(int i=1; i<num_cols; i++) {
    dot = 0.0f;
    if(id == 0) {
        for(int j=0; j<num_cols; j++) {
            dot += a_mat[j*num_cols + i] * u_vec[j];
        }
        barrier(CLK_LOCAL_MEM_FENCE);
        a_mat[id*num_cols + i] -= 2 * u_vec[id] *
            dot / u_length_squared;
    }

    for(int i=0; i<num_cols; i++) {
        q_mat[id*num_cols + i] = -2 * u_vec[i] *
            u_vec[id] / u_length_squared;
    }
    q_mat[id*num_cols + id] += 1;
    barrier(CLK_GLOBAL_MEM_FENCE);
}

```

Transform
columns of A

Initialize Q

It's important to understand how this code obtains and uses the first Householder vector, u_0 . First it loads the first column of A into local memory. Then it computes the length of the vector and subtracts this length from the column's first element. This sets the local memory vector equal to u_0 , and once this is obtained, the kernel uses it to transform each succeeding column of A using the following equation:

$$c_k' = c_k - 2u_0 \frac{u_0 \cdot c_k}{|u_0|^2}$$

After updating A , the kernel forms the Householder matrix P_0 from the Householder vector u_0 using the following equation:

$$P_0 = I - 2 \frac{u_0 u_0^T}{|u_0|^2}$$

Once the kernel obtains P_0 , it places its elements in the Q matrix. As the rest of the kernel executes, Q will multiply further Householder matrices to arrive at its final value.

TRANSFORMING SUCCESSIVE COLUMNS AND UPDATING Q

The next listing presents the second part of the QR decomposition kernel. This loops through the remaining columns of A and computes the Householder vectors needed to transform A into an upper-triangular matrix. As each Householder vector is computed, the kernel finds the corresponding Householder matrix, P_0 , and uses this to update Q .

Listing 12.5 QR decomposition: qr.cl (part two)

```

...
for(int col = 1; col < num_cols-1; col++) {
    u_vec[id] = a_mat[id * num_cols + col];
    barrier(CLK_LOCAL_MEM_FENCE);

    if(id == col) {

```

```

    vec_length = 0.0f;
    for(int i = col + 1; i < num_cols; i++) {
        vec_length += u_vec[i] * u_vec[i];
    }
    u_length_squared = vec_length;
    vec_length = sqrt(vec_length + u_vec[col] * u_vec[col]);
    u_vec[col] -= vec_length;
    u_length_squared += u_vec[col] * u_vec[col];
    a_mat[col * num_cols + col] = vec_length;
}
else if(id > col) {
    a_mat[id * num_cols + col] = 0.0f;
}
barrier(CLK_GLOBAL_MEM_FENCE);

/* Transform further columns of A */
for(int i = col+1; i < num_cols; i++) {
    if(id == 0) {
        dot = 0.0f;
        for(int j=col; j<num_cols; j++) {
            dot += a_mat[j*num_cols + i] * u_vec[j];
        }
    }
    barrier(CLK_LOCAL_MEM_FENCE);
    if(id >= col)
        a_mat[id*num_cols + i] -= 2 * u_vec[id] *
            dot / u_length_squared;
    barrier(CLK_GLOBAL_MEM_FENCE);
}

if(id >= col) {
    for(int i=col; i<num_cols; i++) {
        p_mat[id*num_cols + i] = -2 * u_vec[i] *
            u_vec[id] / u_length_squared;
    }
    p_mat[id*num_cols + id] += 1;
}
barrier(CLK_GLOBAL_MEM_FENCE);

/* Multiply q_mat * p_mat = prod_mat */
for(int i=col; i<num_cols; i++) {
    prod = 0.0f;
    for(int j=col; j<num_cols; j++) {
        prod += q_mat[id*num_cols + j] *
            p_mat[j*num_cols + i];
    }
    prod_mat[id*num_cols + i] = prod;
}
barrier(CLK_GLOBAL_MEM_FENCE);

/* Place the content of prod_mat in q_mat */
for(int i=col; i<num_cols; i++) {
    q_mat[id*num_cols + i] =
        prod_mat[id*num_cols + i];
}
barrier(CLK_GLOBAL_MEM_FENCE);
}

```

Update the
P matrix

Place product
in prod_mat

Move prod_mat
to Q

This code iterates through the columns of A and performs most of the same tasks as the code in listing 12.4. The important difference is how Q is updated. In theory, Q is obtained with the following equation:

$$Q = P_0 P_1 P_2 \dots P_{k-2} P_{k-1}$$

In practice, however, you can't store $k - 1$ Householder matrices in computer memory. The code in listing 12.5 uses three matrices, `q_mat`, `p_mat`, and `prod_mat`, and updates `q_mat` using three steps:

- 1 When the kernel computes a new u_k , it sets `p_mat` equal to the corresponding P_k matrix.
- 2 The kernel computes the product of `q_mat` and `p_mat`, and places the result in `prod_mat`.
- 3 The kernel moves the elements of `prod_mat` to `q_mat`, and prepares for another multiplication.

With so much data transferred to and from global memory, this procedure is not particularly fast. But because there are only three matrices involved, this process is more memory efficient than algorithms that require $k - 1$ Householder matrices.

Once the kernel finishes the last transformation of the columns of A , the result will be the upper-triangular matrix R . After the final multiplication of Householder matrices, the resulting transformation matrix will be Q . To test the decomposition, the host application multiplies Q and R and compares the result to the original values in A .

12.5 Summary

Matrix manipulation is one of the most common tasks that programmers associate with high-performance computing. This is because so many real-world matrices may have thousands or tens of thousands of elements. Matrix operations commonly involve a great deal of number crunching and data transfer, but very little decision making. For this reason, these operations are ideal for implementation with OpenCL.

This chapter has proceeded from the simplest of matrix operations to the complex. The matrix transpose doesn't perform any mathematical operations, but simply moves elements within a matrix. The example code in this chapter demonstrates how the transpose can be performed efficiently by dividing the matrix into blocks and assigning each block to a work-item.

Matrix multiplication relies on the dot product of rows and columns. More specifically, each row of the first matrix multiplies each column of the second, and the result of each dot product is a scalar. If the first matrix has dimensions n by k and the second has dimensions k by p , the product will have dimensions n by p . Vectors can be thought of as matrices—a row vector has dimensions 1 by n and a column vector has dimensions n by 1.

The Householder transformation reflects a vector across a region perpendicular to another vector. This transformation can be performed using a dot product or an

outer product. Dot products are easier to compute, but when you need to combine multiple reflections, you need to use the matrices associated with the outer products.

The QR decomposition discussed in this chapter relies on Householder transformations to reflect the columns of a matrix so that they become upper-triangular. You can combine these transformations by multiplying Householder matrices to obtain the Q matrix. Once the input matrix has been transformed into upper-triangular form, it becomes the R matrix. You can test the accuracy of the decomposition by checking whether $A = QR$.

OpenCL IN ACTION

Matthew Scarpino

Whatever system you have, it probably has more raw processing power than you're using. OpenCL is a high-performance programming language that maximizes computational power by executing on CPUs, graphics processors, and other number-crunching devices. It's perfect for speed-sensitive tasks like vector computing, matrix operations, and graphics acceleration.

OpenCL in Action blends the theory of parallel computing with the practical reality of building high-performance applications using OpenCL. It first guides you through the fundamental data structures in an intuitive manner. Then, it explains techniques for high-speed sorting, image processing, matrix operations, and fast Fourier transform. The book concludes with a deep look at the all-important subject of graphics acceleration. Numerous challenging examples give you different ways to experiment with working code.

What's Inside

- Learn OpenCL step by step
- Tons of annotated code
- Tested algorithms for maximum performance

A background in C or C++ is helpful, but no prior exposure to OpenCL is needed.

Matthew Scarpino has over 12 years of experience developing high-performance applications for embedded systems. He's the author of *Programming the Cell Processor*.

For access to the book's forum and a free eBook for owners of this book, go to manning.com/OpenCLinAction

Covers OpenCL v.1.1

“Thorough coverage of a difficult topic ... excellent explanations of concepts.”

—John J. Ryan III
Princingration LLC

“Well-researched and a good read. It's difficult to find this information elsewhere.”

—Seth Price
Utah State University

“Lucid explanation of OpenCL with many well-chosen applications.”

—Jörn Dinkla, Consultant

“Clearly the best OpenCL reference and hands-on guide out there, packed with amazing real-world examples!”

—Olivier Chafik, Creator of JavaCL and ScalaCL



ISBN 13: 978-1-617290-17-6
ISBN 10: 1-61729-017-3



9 781617 129017