Project: Radar Resampling

University of Washington
EE 590 Winter 2017

Eric Wilson and Darrell Ross

February 27, 2017

Contents

1 Introduction

Professor John D. Sahr in the Electrical Engineering department at the University of Washington (UW) runs a passive receiver that samples at 4 giga-samples per second (GSPS). He is interested in down-sampling to 3.5 GSPS in real-time by leveraging CUDA on Nvidia Keplar K10 graphics cards.

This report serves two purposes. First, it meets project report requirements for EE590: Applied High-Performance GPU Computing. Second, it serves as a report for Professor Sahr on how we solved the down-sampling problem using parallelism.

1.1 Process Overview

An overview of our resampling process is shown in Figure 1. Walking through the figure, the signal from the receiver is sampled for t seconds. After we acquire the sample data, we perform a Least Squares Approximation (LSA) to get the curve of best fit with a sixth–order polynomial. Finally, the data is sampled at the lower rate of 3.5 GPSP.

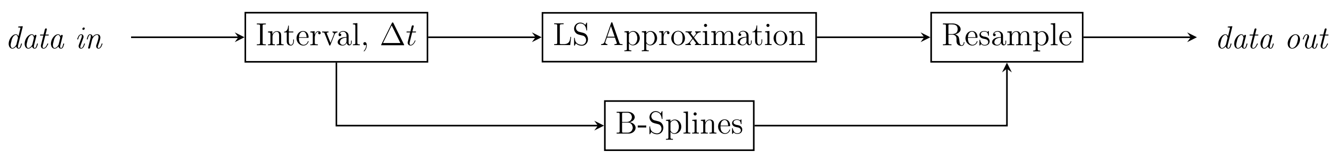

Figure 1: An overview of parts of the resampler.

1.2 Report Organization

- Concepts provides a conceptual review of each piece of the process.
- Algorithms covers the sequential and OpenCL algorithms used.
- Analysis evaluates performance of each piece of the process.

2 Concepts

Each of the entries from Figure 1 are covered in this section.

2.1 Sample Interval

As it turns out, the sample interval is not a simple choice. The interval chosen determines how long a single chunk of data will take to solve. Depending on the size, we may be able to pipeline several solutions at the same time. But pipelining will suffer from diminishing returns due to overhead. So we really need to calculate how much work it will take to run a single process all the way through. A simulation will probably be our best bet to determine optimal sample interval size.

2.2 Least Squares Approximation

The Least Squares Approximation (LSA) is a way of finding the curve of best fit to the sample data. We were given that the polynomial degree would be $k \leq 6$. Performing a Least Squares Approximation (LSA) can be done in many ways. We chose QR Decomposition followed by Back Substitution. Another option considered was Single Value Decomposition (SVD) which is covered in Appendix **??**.

2.2.1 QR Decomposition

QR decomposition involves taking a matrix A and decomposing it into an upper triangular matrix R and an orthogonal matrix Q such that $A = QR$. If A is mxn where $m > n$, then the bottom $(m - n)$ rows of an upper triangular matrix will consist of all zeros. In this situation, R is partitioned. Since $R_2 = 0$, the final result is $Q_1 R_1$.

$$\begin{aligned} A &= QR \\ &= Q \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} \\ &= [Q_1, Q_2] \begin{bmatrix} R_1 \\ R_2 \end{bmatrix} \\ &= Q_1 R_1 \end{aligned}$$

There are multiple techniques available for reaching the upper triangular matrix. We decided to go with the Givens Rotation. Other considered options are covered in Appendix A.2.

2.2.2 Givens Rotation

A sequence of Givens Rotations are used to construct the upper–right triangular matrix R and the combination of the Givens Rotations results in the matrix Q. Each multiplication of the matrix A by the Givens Rotation matrix G results in a zero being inserted into A, as shown in (1).

$$\begin{bmatrix} c & -s \\ s & c \end{bmatrix} \begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} r \\ 0 \end{bmatrix} \tag{1}$$

The Givens rotation matrix G is constructed from c and s where

$$c = \frac{a}{r}$$
$$s = -\frac{b}{r}$$
$$r = \sqrt{a^2 + b^2}$$

2.2.3 Back Substitution

Back Substitution is a quick method of solving for x in (2) without having to invert R. This is quick when R is an upper triangular matrix as it is in our case. An example is provided in Appendix A.3. For our usage, the QR Decomposition with the Givens Rotation will provide us the R and Q matrices so we can quickly solve for x.

$$R \cdot x = Q^T \cdot b \tag{2}$$

2.3 Resampling

Resampling is the process of taking a set of data acquired at one sample rate and converting it to another sample rate. In the case of passive RADAR resampling, the input data rate to the resampler is 4 GSPS and the output rate is 3.5 GSPS. This is a 7/8 resample rate. To achieve a lower sample rate the discrete time input signal is reconstructed as an analog (continuous-time) signal using a polynomial curve fit. Least squares approximation calculates the coefficients for the polynomial curve fit. The polynomial representing the continuous time signal can now be evaluated at the new sample rate to reconstruct the original discrete time signal at a new sample rate.

3 Algorithms

3.1 Interval Sample

The interval sample will determine the number of data points. The number of data points determines the height of the A matrix which will have a maximum width of $k+1$ where k is the order of the polynomial – ours is maximum 6^{th} order. There are several constraints to consider:

- A sixth-order polynomial will fail to accurately match the data if too many data points are used. It is important to know the approximate frequencies of the data so we can avoid this issue.
- The GPU has a limit on how much it can process in parallel which will also constrain the sample size.

Thus, choosing the interval requires knowing the rest of our algorithm and the limits of the GPU being used. The timing of the full algorithm t_{solution} can be split into the three segments depicted in Figure 2. In order to maintain real-time output, our pipeline count p must be such that:

$$t_{\text{solution}} = \Delta t \cdot p$$

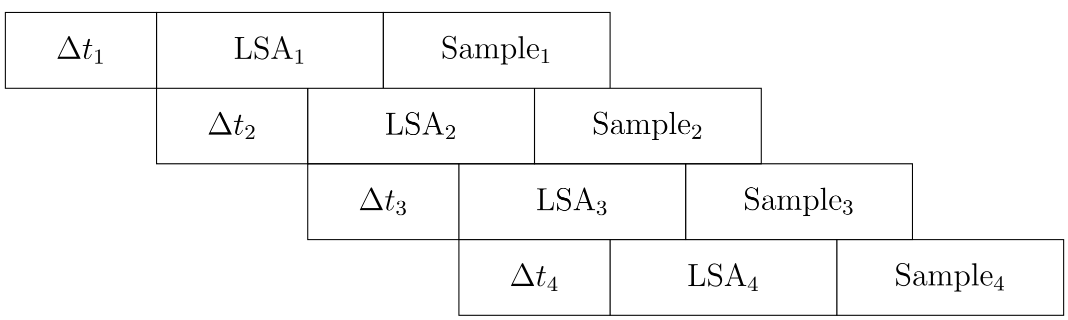

Figure 2: Pipelines with the interval and algorithm.

Once we have a full set of data describing expected execution time and we know what the frequencies are and we know the GPU we are running on, we will finally be able to choose the optimal interval. Until then, we will use sample data.

3.2 Least Squares Approximation

Although the LSA constitutes two algorithms combined together, analysis of them makes more sense when done in one block.

3.2.1 Givens Rotations and QR Decomposition

Givens rotations have a limited scope when it comes to speedup via parallelism. This is due to some interdependency between the order in which the Givens rotations are applied. Because the Givens rotations are only applied to two rows during an iteration, the Givens rotations can be parallelized on different colunms after a certain number of iterations. If each element in the R matrix is zeroed after iteration n, then the following matrix shows the iteration when each element can be zeroed.

$$\begin{bmatrix} x &&&&&& \\ 6 & x &&&&& \\ 5 & 7 & x &&&& \\ 4 & 6 & 8 & x &&& \\ 3 & 5 & 7 & 9 & x && \\ 2 & 4 & 6 & 8 & 10 & x & \\ 1 & 3 & 5 & 7 & 9 & 11 & x \end{bmatrix} \tag{3}$$

A Givens rotation kernel can be started on the first column at the start of algorithm execution and works its way up the first colunm. When the Givens rotation kernel on column 1 gets row 5 on iteration 3, the next Givens rotation kernel can be started and works up column 2. This process can be continued until the last colunm, which in our case is 7 since the largest number of polynomial coefficients we will be using is $k = 6$ and matrix R has $k + 1$ colunms. This means that we can have a maximum of 7 Givens kernels running in parallel at the same time. The following pseudocode implements a method to start each Givens rotation kernel on a colunm after a certain iteration.

```
function [matrix R, matrix Q] = GivensKernel(matrix A, numRows)
    R = A;
    for (iter = 0..12)
        [R, Q1] = GivensKernel(R[colunm 1])
        if (iter > 2)
            [R, Q2] = GivensKernel(R[colunm 2])
        if (iter > 4)
            [R, Q3] = GivensKernel(R[colunm 3])
        if (iter > 6)
            [R, Q4] = GivensKernel(R[colunm 4])
        if (iter > 8)
            [R, Q5] = GivensKernel(R[colunm 5])
        if (iter > 10)
            [R, Q6] = GivensKernel(R[colunm 6])
        if (iter > 11)
            [R, Q7] = GivensKernel(R[colunm 7])
    endfor
    Q = Q1*Q2*Q3*Q4*Q5*Q6*Q7;
endfunction
```

3.2.2 Back Substitution

Back Substitution is really a linear operation. For our problem the final matrix we solve will be small enough (7x7) to make parallelization unnecessary. We do not currently intend to use a kernel for this part of the solution.

3.3 Resampling

Resampling is simply passing our new time interval values into the continuous LSA solution. Since we are guaranteed an order ≤ 6, we can do this in OpenCL using a *cl_float8* for the coefficients. This produces the Resample() kernel shown in Appendix B.1.

4 Performance

This sections covers performance estimates.

4.1 Interval Sample

[Section needs populated once we have run some full simulations and drawn conclusions about what the right sample interval is.]

4.2 Givens Rotations and QR Decomposition

A single Givens Rotation requires work in the following manner. First the calculation of the Givens values:

$$
\begin{aligned}
c &= \frac{a}{r} && W = 1 \\
s &= \frac{b}{r} && W = 1 \\
r &= \sqrt{a^2 + b^2} && W \approx 4 \\
\therefore W_{cs} &= 6
\end{aligned}
$$

Next, the Givens matrix is multiplied by the A matrix which has size MxK. The Givens matrix will only change two rows of the A matrix. This reduces the work to:

$$
\begin{aligned}
W_{gr} &= 3 \cdot 2 \cdot K = 6K && \text{2 products, 1 sum, 2 rows, K columns} \\
\therefore W_{GR} &= W_{cs} + W_{gr} = 6 + 6K
\end{aligned}
$$

In addition, we must maintain the KxK Q matrix by multiplying it by the new KxK G matrix each time. This follows the same pattern as the product with the A matrix in terms of only affecting two rows:

$$
W_{QG} = 3 \cdot 2 \cdot M = 6M \qquad \text{2 products, 1 sum, 2 rows, M samples}
$$

So a single Givens Rotation calculation runs us:

$$
W_T = 6 + 6K + 6M
$$

The number of Givens Rotations necessary will be equivalent to the original MxK matrix minus the upper triangular portion of $2(k+1)$ entries.

$$
W_{\text{total}} = (MK - 2(K+1))(6 + 6K + 6M)
$$

Calculating the memory operations is a bit more straight forward assuming we write a kernel that can do all givens rotations without having to return each time:

$$
\begin{aligned}
Q_r &= DMK && \text{MxK matrix with D bytes per value} \\
Q_w &= DMM && \text{MxM matrix with D bytes per value} \\
D &= 4 && \text{4 bytes per float} \\
\therefore Q &= Q_r + Q_w = 4M^2 + 4MK
\end{aligned}
$$

$$\therefore AI = \frac{W}{Q} = \frac{(MK - 2(K+1))(6 + 6K + 6M)}{4M^2 + 4MK}$$

This is all easier to see with an example. For our example, we will use an input matrix of size 5x3. This would map to our problem as 5 samples of a second order polynomial. Running the numbers:

$$\begin{aligned} W_{\text{total}} &= 9(6 + 18 + 30) = 486 \\ Q &= 160 \\ AI &= \frac{486}{160} = 3.03 \end{aligned}$$

4.3 Back Substitution

Assuming all coefficients are non-zero and not one:

$$\begin{aligned} W_7 &= 1 && \text{1 product} \\ W_6 &= 3 && \text{2 products, 1 sum} \\ W_5 &= 5 && \text{3 products, 2 sums} \\ W_4 &= 7 && \text{4 products, 3 sums} \\ &\dots \\ W_1 &= 13 && \text{7 products, 6 sums} \\ \therefore W_T &= 49 \end{aligned}$$

4.4 Resampling

Reviewing the Resample() kernel, we can calculate the Arithmetic Intensity AI. In this instance, we assume that the work done by the power function W_{pow} is equivalent to $P - 1$ where P is the exponent value. Applying this to an evaluation for a single time x_i produces the output y_i:

$$\begin{aligned} y_i &= a_0 x_i^0 + a_1 x_i^1 + a_2 x_i^2 + a_3 x_i^3 + a_4 x_i^4 + a_5 x_i^5 + a_6 x_i^6 \\ \therefore W &= 1N + 2N + 3N + 4N + 5N + 6N + 5 = 26N && \text{21 products and 5 sums for each input N} \\ Q_r &= DN + 8D && \text{read N entries and one float8} \\ Q_w &= DN && \text{write N entries} \\ \therefore Q &= Q_r + Q_w = 2DN + 8D \\ D &= 4 && \text{4 bytes per float} \\ \therefore Q &= 8N + 32 \\ AI &= \frac{W}{Q} = \frac{26N}{8N + 32} \approx \frac{26}{8} \\ \therefore AI &= 3.25 \end{aligned}$$

5 Analysis

5.1 Resampling

Performance was evaluated using an Intel HD 5500 GPU which has a single-precision floating point maximum of 38.4 GFLOPS and a bandwidth of 12.8 Gbps. With these specs, an AI of at least 38.4/12.8=3 FLOPS/byte is desired. Since this kernel achieves 3.25 FLOPS/byte, it should be an effective application of parallelism.

When analyzed using OpenCL Kernel Development feature in Intel CodeBuilder in Visual Studio 2015, the GPU was faster as shown in Table 1. Presumably this difference will be more stark on

Processor	Sample Size	Time(ms)
Intel® Core™ i3-5010U CPU @ 2.1GHz	1024	0.078
Intel® HD Graphics 5500	1024	0.041

Table 1: Simulation results for the *Resample* kernel.

higher powered GPUs with larger sample sizes. For reference, a sample size of 1024 would be a Δt of only 0.256 μs.

5.2 QR Decomposition

The sequential QR decomposition algorithm was executed on an Intel I7-6500U CPU. The input data size was a 32x32 matrix. Execution time was averaged over 10,000 iterations.

Processor	Sample Size	Time(ms)
Intel® Core™ i7-6500U CPU @ 2.5GHz	1024	0.388

Table 2: Simulation results for the *QR* kernel.

6 Conclusions

Appendix A Algorithm Examples

This section provides working examples for each algorithm used to help demonstrate how they work.

A.1 Least Squares Approximation

A.2 Givens Rotation and QR Decomposition

A.3 Back Substitution

Recall that as we complete our QR Decomposition, we have our Q matrix and our R matrix and we know b so only to solve for x in the following:

$$R \cdot x = Q^T \cdot b$$

Solving by example:

$$\left[\begin{array}{ccc|c} 1 & -2 & 1 & 4 \\ 0 & 1 & 6 & -1 \\ 0 & 0 & 1 & 2 \end{array}\right]$$

$$\begin{aligned} x - 2y + z &= 4 \\ y + 6z &= -1 \\ z &= 2 \end{aligned}$$

Back substitution means that we plug in the last entry, $z = 2$ to the second equation to solve for y and then plug the z and the results for y into the first equation to solve for x.

$$\begin{aligned} & & \Rightarrow z = 2 \\ \therefore y + 12 &= -1 & \Rightarrow y = 11 \\ \therefore x - 22 + 2 &= 4 & \Rightarrow x = 28 \end{aligned}$$

Appendix B Kernels

B.1 Resampling()

```
__kernel void Resample(float8 coeffs, __global float* t, __global float* result)
{
        const unsigned int id = get_global_id(0);
        float val = t[id];
        float out = 0;
        for(unsigned int i=0; i<6; ++i)
                out += coeffs[i]*pow(val,i);

        result[id] = out;
}
```

Appendix C Program Instructions

When we complete our program, this section will contain instructions on how to use it.

Appendix D Attachments

- *control.cpp - C++ code used to execute sequential and kernel functions.*

```
#include "CL/cl.h"
#include "ocl.h"
#include "tools.h"
#include "utils.h"
#include "data.h"
#include "control.h"
#include "profiler.h"
#include "enums.h"
#include <iostream>
#include <vector>
#include <algorithm>

namespace
{
    const char* FILENAME = "resample.cl";
}

ControlClass::ControlClass()
    : GroupManager("Control")
{
    groups_ = GroupFactory();
}


std::map<int, ProblemGroup*> ControlClass::GroupFactory()
{
    std::map<int, ProblemGroup*> pgs;

    ProblemGroup* InputControl = GroupManagerInputControlFactory();
    pgs[InputControl->GroupNum()] = InputControl;

    ProblemGroup* projectFuncs = new ProblemGroup(1, "Control");
    projectFuncs->problems_[projectFuncs->problems_.size() + 1] = new Problem(&exCL_Resample, "OpenCL: Apply
    sixth-order polynomial");
    projectFuncs->problems_[projectFuncs->problems_.size() + 1] = new Problem(&exSeq_Resample, "Sequental: Apply
    sixth-order polynomial");
    pgs[projectFuncs->GroupNum()] = projectFuncs;
    return pgs;
}



////////////////// RESAMPLE USING POLYNOMIAL APPROXIMATION /////////////////
int exCL_Resample(ResultsStruct* results)
{
    cl_int err;
    ocl_args ocl(CL_DEVICE_TYPE_GPU);

    // Create Local Variables and Allocate Memory
    // The buffer should be aligned with 4K page and size should fit 64-byte cached line
    cl_uint sampleSize = 1024;
    cl_uint optimizedSizeFloat = ((sizeof(cl_float) * sampleSize - 1) / 64 + 1) * 64;
```

```
    cl_float* inputA = (cl_float*)_aligned_malloc(optimizedSizeFloat, 4096);
    cl_float* outputC = (cl_float*)_aligned_malloc(optimizedSizeFloat, 4096);
    if (NULL == inputA || NULL == outputC)
    {
        LogError("Error: _aligned_malloc failed to allocate buffers.\n");
        return -1;
    }
    // Generate Random Input
    data::generateInputCLSeq(inputA, sampleSize, 1);

    // Create OpenCL buffers from host memory for use by Kernel
    cl_float8        coeffs = {1,2,3,4,5,6,7,0};
    cl_mem           srcA;             // hold first source buffer
    cl_mem           dstMem;           // hold destination buffer
    if (CL_SUCCESS != CreateReadBufferArg(&ocl.context, &srcA, inputA, sampleSize, 1))
        return -1;
    if (CL_SUCCESS != CreateWriteBufferArg(&ocl.context, &dstMem, outputC, sampleSize, 1))
        return -1;

    // Create and build the OpenCL program - imports named cl file.
    if (CL_SUCCESS != ocl.CreateAndBuildProgram(FILENAME))
        return -1;

    // Create Kernel - kernel name must match kernel name in cl file
    ocl.kernel = clCreateKernel(ocl.program, "Resample", &err);
    if (CL_SUCCESS != err)
    {
        LogError("Error: clCreateKernel returned %s\n", TranslateOpenCLError(err));
        return -1;
    }

    // Set OpenCL Kernel Arguments - Order Indicated by Final Argument
    if (CL_SUCCESS != SetKernelArgument(&ocl.kernel, &coeffs, 0))
        return -1;
    if (CL_SUCCESS != SetKernelArgument(&ocl.kernel, &srcA, 1))
        return -1;
    if (CL_SUCCESS != SetKernelArgument(&ocl.kernel, &dstMem, 2))
        return -1;

    // Enqueue Kernel (wrapped in profiler timing)
    ProfilerStruct profiler;
    profiler.Start();
    size_t globalWorkSize[1] = { sampleSize };
    // hard code work group size after finding optimal solution with KDF Sessions
    size_t localWorkSize[1] = { 16 };
    if (CL_SUCCESS != ocl.ExecuteKernel(globalWorkSize, 1, localWorkSize))
        return -1;
    profiler.Stop();
    float runTime = profiler.Log();

    if (!SKIP_VERIFICATION)
    {
```

```
        // Map Host Buffer to Local Data
        cl_float* resultPtr = NULL;
        if (CL_SUCCESS != MapHostBufferToLocal(&ocl.commandQueue, &dstMem, sampleSize, 1, &resultPtr))
        {
            LogError("Error: clEnqueueMapBuffer failed.\n");
            return -1;
        }

        // VERIFY DATA
        // We mapped dstMem to resultPtr, so resultPtr is ready and includes the kernel output !!!
        // Verify the results
        bool failed = false;
        /// @TODO WRITE SEQUENTIAL VERIFICATION CODE
        /*
        float cumSum = 0.0;
        for (size_t i = 0; i < sampleSize; ++i)
        {
            cumSum += inputA[i];
            if (resultPtr[i] != cumSum)
            {
                LogError("Verification failed at %d: Expected: %f. Actual: %f.\n", i, cumSum, resultPtr[i]);
                failed = true;
            }
        }
        */
        if (!failed)
            LogInfo("Verification passed.\n");

        // Unmap Host Buffer from Local Data
        if (CL_SUCCESS != UnmapHostBufferFromLocal(&ocl.commandQueue, &dstMem, resultPtr))
            LogInfo("UnmapHostBufferFromLocal Failed.\n");
    }

    _aligned_free(inputA);
    _aligned_free(outputC);

    if (CL_SUCCESS != clReleaseMemObject(srcA))
        LogError("Error: clReleaseMemObject returned '%s'.\n", TranslateOpenCLError(err));
    if (CL_SUCCESS != clReleaseMemObject(dstMem))
        LogError("Error: clReleaseMemObject returned '%s'.\n", TranslateOpenCLError(err));

    results->WindowsRunTime = runTime;
    results->HasWindowsRunTime = true;
    results->OpenCLRunTime = ocl.RunTimeMS();
    results->HasOpenCLRunTime = true;
    return 0;
}

int exSeq_Resample(ResultsStruct* results)
{
    return 0;
}
```

```
////////////////// QR DECOMPOSITION ///////////////
int exCL_QRD(ResultsStruct* results)
{
    return 0;
}

/*
 * Sequential QR decomposition function
 */
void QR(cl_float* R, cl_float* Q, cl_uint arrayWidth, cl_uint arrayHeight)
{
    cl_float a;
    cl_float b;
    cl_float c;
    cl_float s;
    cl_float r;
    cl_float Rnew1[2048];
    cl_float Rnew2[2048];
    cl_float Qnew1[2048];
    cl_float Qnew2[2048];
    for (int j = 0; j < arrayWidth; j++)
    {
        for (int i = arrayHeight - 1; i > j; i--)
        {
            // Calculate Givens rotations
            a = R[arrayWidth * (i - 1) + j];
            b = R[arrayWidth * i + j];
            r = sqrt(a * a + b * b);
            c = a / r;
            s = -b / r;
            // Zero out elements in R matrix
            for (int k = j; k < arrayWidth; k++)
            {
                Rnew1[k] = R[arrayWidth * (i - 1) + k] * c - R[arrayWidth * i + k] * s;
                Rnew2[k] = R[arrayWidth * (i - 1) + k] * s + R[arrayWidth * i + k] * c;
            }
            // Copy new values back to R matrix
            for (int k = j; k < arrayWidth; k++)
            {
                R[arrayWidth * (i - 1) + k] = Rnew1[k];
                R[arrayWidth * i + k] = Rnew2[k];
            }
            // Update Q matrix
            for (int k = 0; k < arrayHeight; k++)
            {
                Qnew1[k] = Q[arrayHeight * (i - 1) + k] * c + Q[arrayHeight * i + k] * s;
                Qnew2[k] = -Q[arrayHeight * (i - 1) + k] * s + Q[arrayHeight * i + k] * c;
            }
            for (int k = 0; k < arrayHeight; k++)
            {
```

```
                Q[arrayHeight * (i - 1) + k] = Qnew1[k];
                Q[arrayHeight * i + k] = Qnew2[k];
            }
        }
    }

}

int exSeq_QRD(ResultsStruct* results)
{
    const cl_uint arrayWidth = 3;
    const cl_uint arrayHeight = 5;
    cl_uint numIter = 10000; // Number of iterations for runtime averaging

                         // allocate working buffers.
                         // the buffer should be aligned with 4K page and size should fit 64-byte cached line
    cl_uint optimizedSize = ((sizeof(cl_float) * arrayWidth * arrayHeight - 1) / 64 + 1) * 64;
    cl_float* A = (cl_float*)_aligned_malloc(optimizedSize, 4096);

    optimizedSize = ((sizeof(cl_float) * arrayHeight * arrayHeight - 1) / 64 + 1) * 64;
    cl_float* Q = (cl_float*)_aligned_malloc(optimizedSize, 4096);

    cl_float Atmp[] = { 0.8147, 0.0975, 0.1576,
        0.9058, 0.2785, 0.9706,
        0.1270, 0.5469, 0.9572,
        0.9134, 0.9575, 0.4854,
        0.6324, 0.9649, 0.8003 };

    cl_float Qtmp[] = { 1.0,   0,   0,   0,   0,
                          0, 1.0,   0,   0,   0,
                          0,   0, 1.0,   0,   0,
                          0,   0,   0, 1.0,   0,
                          0,   0,   0,   0, 1.0 };

    if (NULL == A)
    {
        LogError("Error: _aligned_malloc failed to allocate buffers.\n");
        return -1;
    }

    // Initialize A
    for (int i = 0; i < arrayWidth * arrayHeight; i++)
    {
        A[i] = Atmp[i];
    }

    // Initialize Q
    for (int i = 0; i < arrayHeight * arrayHeight; i++)
    {
        Q[i] = Qtmp[i];
    }

```

```
    // add
    ProfilerStruct profiler;
    profiler.Start();

    QR(A, Q, arrayWidth, arrayHeight);

    profiler.Stop();
    float runTime = profiler.Log();

    _aligned_free(A);

    results->WindowsRunTime = (double)runTime;
    results->HasWindowsRunTime = true;
    return 0;

    return 0;
}
```

References

[1] S. C. Kim and S. S. Bhattacharyya. "Implementation of a low-complexity low-latency arbitrary resampler on GPUs". In: *2014 IEEE Dallas Circuits and Systems Conference (DCAS)*. 2014, pp. 1–4. DOI: 10.1109/DCAS.2014.6965333.

[2] Mitra. *Digital Signal Processing: A Computer Based Approach*. 4th International edition. Mcgraw Hill Higher Education, 2010. ISBN: 9780071289467. URL: http://amazon.com/o/ASIN/0071289461/.

[3] Eric W. Weisstein. *Least Squares Fitting–Polynomial. From MathWorld—A Wolfram Web Resource*. Last visited on 21/2/2017. URL: http://mathworld.wolfram.com/LeastSquaresFittingPolynomi html.