Advanced QRD Optimization with Intel® HLS Compiler

QR Matrix Factorization

Author
Paul White
Application Engineer,
HLS Factory Applications
Intel® Corporation

Introduction
The Intel® HLS compiler is a new high-level synthesis (HLS) design tool that translates C++ code into a register transfer level (RTL) component that can be implemented in an FPGA. This means that a user can test that their algorithm is functionally correct by compiling it to x86 instructions (which takes relatively little time) instead of simulating RTL (which takes a relatively long time). When the user is satisfied that their code is functionally correct, they can simply change the target architecture to one of many supported FPGAs. This causes the compiler to generate RTL code instead of x86 instructions. The Intel HLS compiler uses the user's C++ testbench code to functionally verify the component so the user does not need to build a second testbench to simulate the generated RTL.

The potential of a tool like this is incredible: historically, designing for FPGAs was extremely time-consuming and required a user to write RTL by hand. The Intel HLS compiler makes embedded FPGA designs accessible to people with software backgrounds in addition to people with FPGA backgrounds. That being said, programming for an FPGA with a HLS tool is a bit different than programming for a CPU. The performance of a HLS design is very dependent on the style of the code given to it and some of these optimizations may not be obvious to people who are inexperienced with designing for FPGAs.

Document Overview
The purpose of this document is to illustrate best practices as applied to a real design, specifically the QR decomposition design example that ships with the HLS tool. We start with C++ code and optimize it for an Intel Arria® 10 FPGA using the Intel HLS compiler. This document is intended to help software engineers who are new to high-level synthesis to effectively use the Intel HLS Compiler by showing how HLS optimizations may be applied to a real piece of code. We assume that you are already familiar with the Intel HLS compiler, compiling designs, and viewing the reports. You can gain the necessary skills by working through the tutorials and viewing the user guide. If you encounter a HLS concept in this document that you do not understand, refer to the HLS Reference Manual.

This white paper contains a section that describes the source files included in the QR Decomposition (QRD) design example that ships with the HLS tool, as well as how to compile and run it. It also contains source codes for a naïve implementation of the QRD algorithm and a section that walks through the various HLS optimization steps.
Algorithm Summary

The design example used in this document implements a QR Decomposition using the Modified Gram-Schmidt (MGS) algorithm. This algorithm factorizes a matrix $A$ into two factor matrices $Q$ and $R$, therefore $A=QR$ (where '*' means matrix multiplication). For a detailed description of this algorithm, please see Appendix A. For the purposes of understanding this document, you do not need to understand the details of the algorithm, but you should understand the looping pattern. Figure 1 shows the QRD algorithm, and Figure 2 illustrates the looping pattern across a matrix with 4 rows and 4 columns.

Figure 1. QRD Algorithm Used in This Design Example

Figure 2. MGS Looping Pattern

For $i=1$, the algorithm computes the 1st column of matrix $Q$ using the 1st column of the input matrix, and modifies the remaining three columns of the input matrix. For $i=2$, the algorithm computes the 2nd column of matrix $Q$, and modifies the remaining 2 columns of the input matrix. For $i=3$, the algorithm computes the 3rd column of matrix $Q$, and modifies the single remaining column of the input matrix. For $i=4$, the algorithm computes the 4th column of matrix $Q$ using the 4th column of the input matrix, and then terminates because there are no more columns to modify in the input matrix.

There are two key factors that impact the parallelizability of this algorithm:

1. The $i$ iterations are dependent, so the $i=2$ iteration cannot begin until the $i=1$ iteration has completed.
2. The $j$ iterations are not dependent, so $j=2, j=3,$ and $j=4$ iterations could all run simultaneously during the $i=1$ iteration.
Using the Design

Table 1 lists the performance numbers for different sized QR decompositions.

<table>
<thead>
<tr>
<th>PRECISION</th>
<th>MATRIX SIZE</th>
<th>LATENCY (CYCLES)</th>
<th>DSPs</th>
<th>RAM BLOCKS</th>
<th>fMAX (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single-precision</td>
<td>8x8</td>
<td>367</td>
<td>38</td>
<td>39</td>
<td>344</td>
</tr>
<tr>
<td></td>
<td>16x16</td>
<td>2,167</td>
<td>70</td>
<td>69</td>
<td>333</td>
</tr>
<tr>
<td></td>
<td>32x32</td>
<td>6,039</td>
<td>134</td>
<td>131</td>
<td>286</td>
</tr>
<tr>
<td></td>
<td>64x64</td>
<td>20,375</td>
<td>262</td>
<td>311</td>
<td>252</td>
</tr>
<tr>
<td></td>
<td>128x128</td>
<td>90,128</td>
<td>518</td>
<td>1,230</td>
<td>166</td>
</tr>
</tbody>
</table>

Table 1. Design performance for different sized decompositions.¹

¹ Design targets the Intel Arria 10 FPGA with Intel Quartus Prime software v17.1 build 236 and the Intel HLS Compiler 17.1 build 236.

The QRD reference design contains the following files:

<project root>

- **MGS.cpp**
  - This file contains an HLS component that implements the QRD algorithm (i.e. the `qrd()` function).

- **MGS.h**
  - This header file configures parameters and defines types that may be used by the component.

- **QrdTestBench.cpp**
  - This file calls the component function `qrd()`, and verifies that it produces functionally correct output. The testbench can test several known matrices of various sizes, and can also test dynamically generated matrices.

- **QrdTestBench.h**
  - This file contains function definitions for the functions and custom types used in the testbench.

- **TestbenchHelper.cpp/TestbenchHelper.h**
  - These files declare and define several helper functions that are used by the testbench, such as displaying debug information, and verifying the correctness of a QR factorization.

- **Makefile**
  - This file automates the build process on Linux for the different HLS build targets: test-x86-64, test-fpga, and test-gpp.

- **build.bat**
  - This file automates the build process on Windows for the different HLS build targets: test-x86-64, test-fpga, and test-msvc.

To build the QRD design example, at a command line simply execute one of the commands shown in Table 2 or Table 3.

Table 2.: Build Commands for Windows* Operating System

<table>
<thead>
<tr>
<th>TARGET</th>
<th>WINDOWS BUILD COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>i++ x86 build and execution</td>
<td>build.bat test-x86-64</td>
</tr>
<tr>
<td>i++ FPGA build and simulation</td>
<td>build.bat test-fpga</td>
</tr>
<tr>
<td>msvc build (no i++ compiler)</td>
<td>build.bat test-msvc</td>
</tr>
</tbody>
</table>

Table 3.: Build Commands for Linux* Operating System

<table>
<thead>
<tr>
<th>TARGET</th>
<th>LINUX BUILD COMMAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>i++ x86 build and execution</td>
<td>make test-x86</td>
</tr>
<tr>
<td>i++ FPGA build and simulation</td>
<td>make test-fpga</td>
</tr>
<tr>
<td>gpp build (no i++ compiler)</td>
<td>make test-gpp</td>
</tr>
</tbody>
</table>
QRD Component (MGS.cpp and MGS.h)

One important feature of this reference design is that matrix sizes may be specified at runtime, however the component is limited to processing matrices of a maximal size specified at compile time. Therefore, if the component size is set to 16x16, the synthesized component will be able to factorize a matrix of any size up to and including 16x16 elements.

There are two ways that you can adjust the behavior of the QRD reference design:

- **Maximal component size** - You can adjust the maximal rows and maximal columns that the component can support using the `ROWS_COMPONENT` and `COLS_COMPONENT` macros.
- **Precision** - You can choose whether the component operates at double-precision or single-precision, by defining the `DOUBLE_PRECISION` macro. This will then define the `QrdFloatingPoint` type as either a single-precision float or a double-precision double.

When adjusting the component size, ensure that the `hls_bankbits()` parameter is set correctly. Examples of how to set `hls_bankbits()` are given for matrices of size 4x4, 8x4, 8x8, 16x8, and 64x64. If you set the `ROWS_COMPONENT` and `COLS_COMPONENT` macros to sizes other than these, you will need to define the `BANKBITS` macro yourself. Also note that this component works best when `ROWS_COMPONENT` and `COLS_COMPONENT` are set to powers of 2. Using non-powers of 2 will result in poor performance. More details on setting the component size can be found in the Memory Access section on page 12.

QRD Testbench (QRD_Testbench.cpp, QRD_Testbench.h, TestbenchHelpers.cpp, TestbenchHelpers.h)

**QRD_Testbench.cpp** contains the `main()` function that actually calls the component function (`qrd()`). It contains four functions for testing the `qrd()` function:

- `randomMatrixTest()` generates a random matrix, and checks that the resulting `Q` and `R` matrices meet the necessary requirements:
  - `Q` must have orthogonal columns
  - `R` must be triangular
  - `Q \times R` should result in a matrix that is equivalent to the input matrix within a margin of error $\epsilon$ (`EPSILON`).
- `qrd_3x3()`, `qrd_4x3()`, `qrd_8x8()` operate on static matrices whose MGS QR decompositions are known, and compare the calculated `Q` and `R` matrices with pre-calculated `Q` and `R` matrices.

**TestbenchHelpers.cpp** contains helper functions for **QRD_Testbench.cpp**. These include functions for carrying out the matrix comparisons and formatting matrix data for display in the terminal.

There are many ways to adjust the testbench by configuring the macro definitions in **QRD_Testbench.cpp**.

- **Testing Options** - You can choose whether to test the component against pre-defined matrices or against randomly generated matrices, or both by defining `PREDEFINED_TEST`, `RANDOM_TEST` and/or `SHOW_MATRICES`. If `RANDOM_TEST` is defined, you can set the size of the randomly generated matrices using the `TEST_ROWS` and `TEST_COLS` macros. Note that you should not set the `TEST_ROWS` and `TEST_COLS` macros to be larger than the `ROWS_COMPONENT` and `COLS_COMPONENT` macros.
- **Display Options** - You can choose whether or not the testbench prints the calculated `Q` and `R` matrices.

Naïve Implementation

The MGS algorithm is a good example of a rather complex algorithm that can be accelerated by an FPGA. There are two primary challenges to implementing the MGS algorithm using HLS tools.

- **MGS has loop-carried dependencies.** This means that iterations of the main loop rely on calculations done in previous iterations, and therefore cannot be pipelined or unrolled. In MGS, the loop-carried dependency is the `T` matrix, which is read at the beginning of the main loop (Line 2 of Figure 15), and then written to during each iteration of the inner loop (Line 7 of Figure 1).
- **The inner loop has a non-constant number of iterations.** This makes it difficult for the compiler to unroll this loop. However, we can work around this issue by using a technique called predication. Predication is illustrated in Listing 2 below.
There are many ways to optimize this algorithm to run well on an FPGA, but first we should establish what results the tool is able to produce with a naïve implementation of the algorithm in Figure 1. Listing 1 shows a C++ implementation of MGS.

1. component void qrd(double a_matrix[], int rows, int cols, double r_matrix[]) {
2.    double r_matrix[];
3.    int total_elements = rows * cols;
4.    int rDimension = cols;
5.    double t_matrix[total_elements]; // dynamically allocated
6.    // line 1 of algorithm
7.    // copy A matrix to T matrix
8.    for (int index = 0; index < total_elements; index++) {
9.        t_matrix[index] = a_matrix[index];
10.   }
11.   // line 2, 3 of algorithm: calculated alpha (t_magnitude)
12.   double t_magnitude = magnitudeColumn(t_matrix, rows, cols, i);
13.   // line 4 of algorithm
14.   // generate the ith column of the Q matrix
15.   for (int mRow = 0; mRow < rows; mRow++) {
16.       q_matrix[mRow * cols + i] = t_matrix[mRow * cols + i] / t_magnitude;
17.   }
18.   // line 5, 6 of algorithm
19.   for (int j = i + 1; j < cols; j++) {
20.       // line 7 of algorithm
21.       double dotProduct = 0;
22.       for (int mRow = 0; mRow < rows; mRow++) {
23.           dotProduct += q_matrix[mRow * cols + j] * t_matrix[mRow * cols + j];
24.       }
25.       r_matrix[i * rDimension + j] = dotProduct;
26.       r_matrix[j * rDimension + i] = 0.0;
27.   }
28.   // line 8 of algorithm
29.   // generate the next column of the T-matrix.
30.   for (int mRow = 0; mRow < rows; mRow++) {
31.       double t = t_matrix[mRow * cols + j];
32.       double q = q_matrix[mRow * cols + i];
33.       t_matrix[mRow * cols + j] = t - (dotProduct * q);
34.   }
35.   double magnitudeColumn(double matrixData[], int rows, int cols, int targetColumn) {
36.    double sum = 0;
37.    for (int row = 0; row < rows; row++) {
38.        double val = matrixData[row * cols + targetColumn];
39.        sum += val * val;
40.    }
41.    return sqrt(sum);
42. }
43. }  
44. }  
45. 
46. Listing 1: Naïve C++ Implementation of MGS

The code in this listing will compile for x86, but not when targeting an FPGA. This is because dynamic memory allocation is not supported for FPGAs. It is not possible to dynamically allocate or deallocate RAM blocks at runtime; this would require re-routing logic. We can avoid this dynamic allocation in one of two simple ways:

- **Statically allocate the t_matrix array with a maximal number of elements that we wish to support (e.g. 8 rows x 8 columns = 64 elements)**

- **If we relax the requirement that a_matrix is not to be modified, we can remove the need to use t_matrix entirely, and perform the QRD algorithm on the data passed in a_matrix through system memory.**
HLS Optimization

Now we are ready to begin optimizing the design. The performance of a HLS design is highly dependent on the memory access patterns and datatypes chosen by the designer. The Intel HLS Compiler includes many ways to allow the designer to specify memory options and looping behavior. The following flow is used to optimize this design example (although some iteration will likely be necessary in your design):

1. Interfaces - How does your component communicate with the rest of the system? Will you pass data by value or by reference? Which Avalon® interface will you use (for example Master, Slave, Stream)? Consider system design.

2. Looping - How does your algorithm flow? Where can loop structures be parallelized? Where can loops be merged together?

3. Memory optimization - How does your component access local memories? This is tightly coupled with looping, and your looping decisions will impact memory optimizations.

4. "Finishing Touches" - small optimizations that can yield performance improvements. For example, relaxing floating-point accuracy requirements, choosing specialized math operations, and optimizing loop counters.

A summary of the performance benefits introduced by each of the optimizations described in this document is shown in Table 4. These results were obtained for an 8x8 QR decomposition using a component capable of handling up to an 8x8 QR decomposition. "Latency" defines how many cycles the component took to execute in its testbench, and is reported in the HTML report. The "DSPs" and "RAMs" columns indicate how many hardened Digital Signal Processors (DSPs) and how many M20K Random Access Memory (RAM) blocks were consumed by the design. The "ALMs" column indicates how many logic units were consumed. This is a representation of generic logic that is used by a design, and includes routing logic. The "fMAX" column indicates the maximum clock frequency that the compiled component could execute at, as determined by the Intel Quartus Prime software.

<table>
<thead>
<tr>
<th>LATENCY (CYCLES)</th>
<th>DSPs</th>
<th>RAMs (M20K)</th>
<th>ALMs</th>
<th>fMAX (MHz)</th>
<th>COMPIL TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stock algorithm (static allocation)</td>
<td>12,834</td>
<td>54</td>
<td>43</td>
<td>15209.5</td>
<td>357.27</td>
</tr>
<tr>
<td>Streams, 2D Arrays</td>
<td>7,498</td>
<td>41</td>
<td>46</td>
<td>11000.5</td>
<td>304.97</td>
</tr>
<tr>
<td>Single precision conversion</td>
<td>2,274</td>
<td>10</td>
<td>14</td>
<td>2099.5</td>
<td>333.11</td>
</tr>
<tr>
<td>Loop Unrolling/pipelining</td>
<td>1,415</td>
<td>73</td>
<td>71</td>
<td>16827.5</td>
<td>257.67</td>
</tr>
<tr>
<td>RAM banks: Column Major</td>
<td>1,222</td>
<td>73</td>
<td>57</td>
<td>6017</td>
<td>321.75</td>
</tr>
<tr>
<td>RAM banks: Row Major</td>
<td>1,260</td>
<td>73</td>
<td>61</td>
<td>7936.5</td>
<td>303.67</td>
</tr>
<tr>
<td>Division avoidance</td>
<td>1,196</td>
<td>52</td>
<td>41</td>
<td>5515</td>
<td>305.72</td>
</tr>
<tr>
<td>Dot products and fp-relaxed</td>
<td>858</td>
<td>38</td>
<td>39</td>
<td>4576</td>
<td>313.28</td>
</tr>
<tr>
<td>Bit-accurate loop variables</td>
<td>855</td>
<td>38</td>
<td>39</td>
<td>4368.5</td>
<td>347.22</td>
</tr>
</tbody>
</table>

Table 4. Performance Changes After Various Optimizations. These Designs Are Iterations on an 8x8 QR Decomposition.¹

¹ Design targets the Intel Arria 10 FPGA with Intel Quartus Prime software v17.1 build 236 and HLS 17.1 build 236.

Subroutines²

The Intel HLS Compiler supports code re-use by subroutines, but this does not lead to hardware re-use. If your component invokes a subroutine, (such as the magnitudeColumn() subroutine defined on line 46 of Listing 1) the subroutine will simply be in-lined into the component function at compile-time, forming a single, monolithic piece of code. In this design, we explicitly in-line the subroutine to simplify debugging. Debugging repeated calls to a subroutine can be confusing because the order in which subroutine calls occur may not be clear from the Loops Analysis section of the HLD report, as shown in Figure 3

² This subsection is not really an "optimization" as described above, and it does not affect the component's performance or area consumption.
The optimized version of this design uses streaming interfaces, which allow a testbench or other component to stream data in or out one word at a time. Streaming interfaces can also be a useful mechanism for designs that can begin processing input data before the whole input data is available (e.g. a filter), and even better for designs that do not need to store the entire input data. Streaming outputs work well for designs that have feed-forward outputs (that is, once the output has been computed, it does not need to be stored by the component for future computation).

Unfortunately, since MGS requires the entire input matrix during the first iteration, we need to store all $m \times n$ elements of the $T$ matrix. We can declare buffers that are local to the component, and then read data from the streams into the local buffers, as shown in Figure 4 and Listing 2.

A final point to note concerning optimizing streaming reads and writes: since streams cannot be written to in parallel, you should not unroll a loop that contains a stream read or a stream write. However, we may optimize nested loops using #pragma loop_coalesce as shown on line 5 and line 16 of Listing 2. This pragma instructs the compiler to simplify nested loops by re-configuring them as a single loop. This simplifies the loop control logic and improves latency and area consumption.
Loop Unrolling and Pipelining

One of the main benefits of using an FPGA as opposed to a microprocessor is that FPGAs use a spatial compute structure. This means that a design can have a reduced latency in exchange for additional area consumption. A user can accelerate loop structures by pipelining them, or even unrolling them and executing multiple loop iterations fully concurrently.

- **Loop Unrolling**

  Unrolling is the most intuitive form of parallelization. Since each iteration of the loop is replicated in hardware and executes simultaneously, this strategy trades an increase in area for a reduction in latency. Figure 5 shows a basic loop with three stages and three iterations. This loop has a latency of nine cycles (assuming that each stage takes one cycle to execute).

```c
# include <iostream>

int main() {
    int a[10];
    for (int i = 0; i < 10; i++) {
        a[i] = i * i;
    }
    return 0;
}
```

**Figure 5. Basic Loop with Three Stages and Three Iterations**
Figure 6 shows the same loop unrolled three times. Note that three iterations of the loop can now be completed in only three cycles, but three times as many hardware resources are required. A loop can be unrolled using the `#pragma unroll` directive, but this only works if the compiler knows in advance how many times a loop will need to repeat (or at least, if it knows the maximal number of times that a loop will need to repeat).

Loop Pipelining

Pipelining is another form of parallelization that allows multiple iterations of a loop to execute concurrently. Pipelining is possible when each stage of a loop is independent and can work concurrently, like an assembly line. Figure 7 illustrates the pipelining of a three-stage loop. For our example, this strategy has a latency of five cycles for three iterations (and six cycles for four iterations), but there is no area tradeoff. During cycle 2, Stage 1 of the pipeline is processing iteration 2, Stage 2 is processing iteration 1, and Stage 3 is inactive. This loop is pipelined with an Initiation Interval (II) of 1. That means that there is a delay of one cycle between starting each successive loop iteration. The Intel HLS Compiler attempts to pipeline loops by default, and loop pipelining is not subject to the same constant iteration count constraint that loop unrolling is.

Not all loops can be nicely pipelined as shown in Figure 7, particularly loops where each iteration depends on a value computed in a previous iteration. For example, consider if stage 1 of the loop shown in Figure 5 and Figure 7 depended on a value computed during stage 3 of the previous loop iteration. This would mean that the orange iteration would not be able to start executing until the light blue iteration had reached stage 3. This type of dependency is called a loop-carried dependency. Loops with loop-carried dependencies cannot be pipelined with II = 1. In this example, the loop would be pipelined with II=3. Since the II is the same as the latency of a loop iteration, the loop would not actually be pipelined at all.
• **Optimizing the QRD Design’s Loops**

Looking at the code in Listing 1, we can determine which of the loops can be pipelined or unrolled. The \(i\)-loop at line 14 cannot be pipelined because the \(T\)-matrix imposes a loop-carried dependency.

The \(j\)-loop at line 25 can be pipelined because each of the dot-products that it uses do not interfere with each other. Each iteration of the \(j\)-loop depends on \(Q\) and \(T\), but each \(T\) that is calculated only depends on the \(Q\) that was calculated in the current iteration of the \(i\)-loop and the \(T\) that was calculated in the previous iteration of the \(i\)-loop. That iteration is guaranteed to have completed because there will never be more than one iteration of the \(i\)-loop running at a time, so the \(j\)-loop can be pipelined or even unrolled. Since unrolling the \(j\)-loop requires concurrent accesses of the entire \(T\)-matrix, it is best to leave this loop rolled and simply let the compiler pipeline it. If we do that though, we can see that the compiler pipelines the \(j\)-loop with an II of 83. This is caused by a perceived loop-carried dependency across \(T\)-matrix. Figure 8 illustrates how to break this dependency using \#pragma ivdep. This pragma is safe to use because we know that there is no dependency, however if you use this pragma when there really is a dependency, the generated RTL will be functionally incorrect. In the final source code, we break the dependency using a register and do not use \#pragma ivdep.

![Figure 8. \#pragma ivdep Can be Used to Break Loop-Carried Dependencies](image)

We can improve performance by unrolling the loops that iterate across rows of a particular column. Figure 1 conceals an additional layer of looping by summarizing operations that occur over a vector, such as the dot-product at line 5 (Line 27 of Listing 1). In fact, the multiplication referred to at line 5 refers to \(m\) floating point multiplication operations, one for each row of \(T\). This can be effectively unrolled since none of the operations depends on any of the others. Listing 3 shows how this may be done using \#pragma unroll. Note that ROWS_COMPONENT is a constant defined at compile-time, ensuring that this loop has a fixed number of iterations. Note that the write to \(q\)-matrix is predicated by an if-statement to prevent writing invalid data.

1. double dotProduct = 0;
2. \#pragma unroll
3. for (int mRow = 0; mRow < ROWS_COMPONENT; mRow++)
4. {
5. \hspace{1em} if (mRow > rows) \{ \hspace{0.5em} // more predication
6. \hspace{2em} dotProduct += q_matrix[(mRow * cols) + i]
7. \hspace{2.5em} * t_matrix[(mRow * cols) + j];
8. \hspace{1em} \}
9. \}

Listing 3. Unrolling a Row-Wise Loop
Memory Allocation

The Intel HLS Compiler gives two options for storing local data—RAM blocks and registers. For large memory structures, registers consume more area than RAM blocks (that is, 20k bits of registers consumes more area than 20k bits of RAM blocks), but registers can easily be accessed in parallel. RAM blocks may also be accessed in parallel, however they must be properly configured. Since this design targets larger matrix factorizations (matrices with more than 4x4 elements), RAM blocks were chosen as a storage medium. Even though the HLS compiler automatically decides to store larger array structures in RAM blocks, we need to use the hls_memory attributes when declaring the Q-matrix, R-matrix, and T-matrix buffers to ensure that the RAM blocks are configured properly, as illustrated in Listing 4.

```c
1. hls_memory hls_singlepump hls_bankbits(3,4,5) hls_bankwidth(sizeof(float))
2. float q_matrix[8][8];
3. 
4. hls_memory hls_singlepump hls_bankwidth(sizeof(float))
5. float r_matrix[8][4];
6. 
7. hls_memory hls_bankbits(2,3,4) hls_bankwidth(sizeof(float))
8. float t_matrix[8][4];
9. 
10. // rest of code...
```

Listing 4. Configuring Local Memories in a Component Function

Since `q_matrix` and `r_matrix` are not accessed frequently, we can specify the hls_singlepump attribute to prevent the compiler from double-pumping the memories. This can improve \( f_{\text{MAX}} \). Since the `t_matrix` is read/written frequently, it should not be single-pumped. Forcing the memory to be single-pumped will result in arbitration logic, which can impose a heavy performance penalty as shown in Figure 9.

![Single-pumped t_matrix. There are 2 read/write sites per bank.](image1)

![Double-pumped t_matrix. There are 4 read/write sites per bank.](image2)

Figure 9. Single-Pumped vs Double-Pumped
• Memory Access

To access multiple hls_memory locations in parallel, each of those memory locations needs to exist in a separate RAM bank. Therefore, the address of a given word in a memory has two partitions: a bank partition and a word partition. The hls_bankbits attribute is used to specify which bits of a memory address to use to address a memory bank, and which bits to use to address specific words within that bank. This attribute is described in greater detail in the HLS Compiler Reference Manual [2]. It is good practice to implement arrays which will be banked for parallel access as multi-dimensional arrays, because this makes it easier to see which elements will be in the same bank and which elements can be accessed in parallel, as shown in Figure 10.

Consider the unrolled for-loop in Listing 3. Each iteration of the unrolled loop accesses a different row of q_matrix, so each row needs to exist in a separate bank. We can tell the compiler this by specifying the hls_bankbits() and hls_bankwidth() attributes when we declare q_matrix, as shown in Listing 2. The T matrix requires eight banks (with 4 32-bit words in each bank). Since each bank has a width of one word, we specify a bank width of 4 bytes (sizeof(float)). This means that the T-matrix requires 3 bits to address each bank, and a further 2 bits to address each of the words in each bank. The address bits are assigned as shown in Figure 11.

![Figure 10. Banking Memory for Parallel Access](image)

Consider the unrolled for-loop in Listing 3. Each iteration of the unrolled loop accesses a different row of q_matrix, so each row needs to exist in a separate bank. We can tell the compiler this by specifying the hls_bankbits() and hls_bankwidth() attributes when we declare q_matrix, as shown in Listing 2. The T matrix requires eight banks (with 4 32-bit words in each bank). Since each bank has a width of one word, we specify a bank width of 4 bytes (sizeof(float)). This means that the T-matrix requires 3 bits to address each bank, and a further 2 bits to address each of the words in each bank. The address bits are assigned as shown in Figure 11.

<table>
<thead>
<tr>
<th>Bank Bits</th>
<th>Word Bits</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>3</td>
</tr>
</tbody>
</table>

![Figure 11. Bit Banking for T Matrix](image)

By default, the compiler automatically assigns the lowest-order bits of an address as the bank bits. For instance, consider the case where we store the Q and T matrices in column-major order instead of row-major order. To realize this design in hardware, multiple RAM banks are still needed, but the banking happens transparently to the user because a bank width of n*sizeof(float) is specified. The memory can be envisioned as shown in Figure 10. A column of the matrix would be represented by a row of memory addresses at the same depth in each RAM block. This means that the lower order bits are used to index each row instead of the higher-order bits as shown in Figure 11. In this case, the hls_bankwidth and hls_bankbits attributes do not even need to be specified.
Figure 12 shows that memory usage does not vary substantially between the row-major or column-major implementations of this algorithm (except in the 64x64 case), and performance and overall area consumption is also comparable, even as matrix size scales.

![Figure 12: Performance and Area Consumption as a Function of Problem Size for Row-Major and Column Major Implementations of QRD](image)

- **Power of 2 Requirement**
  
  Note that banking RAM blocks only works well when the size of the RAM block is a power of 2. If $\text{COLS\_COMPONENT}$ is set to 24, the bit-alignment does not work anymore and arbitration blocks are added. However, if the $Q$ and $T$ matrices have their column initialization overridden to be a power of 2, (such as 32) and $\text{COLS\_COMPONENT}$ is left set to 24, then area consumption and latency improve.

- **Index Masking**
  
  Consider the optimized code snippet shown in Listing 5 (the loop at line 6 corresponds to line 49 of the code shown in Listing 1). When accessing parallel banks of memory, it may be necessary to mask accesses to some of the array dimensions. This is necessary because the compiler needs to be able to prove that a particular unrolled loop will not try to access more than one memory address from each bank. In this design, we mask values that could potentially exceed their bank size to clamp them to a specific range. For example, if $t\_\text{matrix}$ is an 8x8 matrix, then in the code in Listing 5, $\text{COLS\_MASK}$ should be set to 0x7 because 3 bits are required to address the lower-order memory addresses of $t\_\text{matrix}$. This must be done to cover cases where $i$ is not less than 8. The compiler cannot prove that $i$ is always less than 8 because $i$ is limited by the variable rows. Since $\text{rows}$ comes from the outside world, the compiler cannot guarantee that it will not exceed the bank size.
For example, consider the case where $i$ reaches a value of 12. Referring to $t_{\text{matrix}[0][12]}$ is syntactically legal even if the number of columns is restricted to 8, and would be equivalent to accessing $t_{\text{matrix}[1][4]}$. However, exceeding the higher order index is guaranteed to access memory outside the bounds of the array and cause a segmentation fault, so we do not need to mask the higher-order index.

### Listing 5: Index Masking Example

```c
for (int i = 0; i < rows; i++) {
    // find magnitude of t_{i} (i-th column)
    QrdFloatingPoint sum = 0;
    #pragma unroll
    for (int row = 0; row < ROWS_COMPONENT; row++) {
        // hardened dot-product
        QrdFloatingPoint val = t_matrix[row][i & COLS_MASK];
        sum = sum + (val * val);
    }
    t_magnitude_inv = sqrt(sum);
}
```

Failure to properly mask indices in this way can lead to arbitration logic being synthesized.

"Finishing Touches": Arbitrary-Precision Loop Counters

Another way to improve loop performance is to limit the precision of loop counters using ac_int arbitrary-precision integers. This is not necessary for loops that are unrolled, but can provide a small area and performance improvement for rolled loops. For example, consider the loop at line 14 in Listing 1. This loop iterates across the columns of the input matrix, so it will never be larger than cols, which itself should not be larger than COLS COMPONENT. Therefore, we can define an ac_int that is wide enough to contain COLS COMPONENT. If COLS COMPONENT is a power of 2, then this ac_int should be configured to have enough room to roll over. For example, if COLS COMPONENT is 8, then the loop counter should have 4 bits. If the loop counter has just 3 bits, then the loop will never exit. Listing 6 shows how a type like this may be defined, and then used in the loop in Listing 5 instead of the int type. This type definition is in MGS.h.

### Listing 6: Defining an ac_int to Optimize Loop Counters

```c
#define BITS_EXCL(N)

// unsigned int just large enough to enumerate COLS COMPONENT + 1 elements
typedef ac_int<BIT_EXCL(COLS_COMPONENT), false> QRD_COL_LOOP;

for (QRD_COL_LOOP i = 0; i < rows; i++) {
    ...
}
```

"Finishing Touches": Mathematics

Since this design performs floating-point arithmetic, hardware floating-point DSPs are used when targeting newer FPGA architectures such as Intel Arria 10 devices. By carefully choosing which operations we use, we can save area by minimizing digital signal processing (DSP) usage. Mathematical optimizations include careful datatype selection, avoiding unnecessary operations, and taking advantage of built-in hardened intellectual property (IP) cores (such as the hardened dot-product IP).
Datatype Selection

Great area and latency savings can be achieved by choosing to use single-precision floating point numbers (floats) instead of double-precision floating-point numbers (doubles). If you decide to use single-precision datatypes, be sure to use single-precision operations (such as sqrtf() and fabsf()). Figure 13 shows how the HLD report may be used to check the area consumption of floating-point operations. For more information, see the single-precision vs double-precision tutorial (tutorials/best_practices/single_vs_double_precision_math).

If your design can tolerate small inaccuracies, area may also be saved by using the --fpc and --fp-relaxed compiler flags. The --fpc flag optimizes intermediate rounding and conversion operations during chained floating-point operations (when DSP blocks cannot be used), and the --fp-relaxed flag allows the compiler to generate shallow adder trees for sequential additions. An example of such an adder tree is shown on the right in Figure 14.

![Figure 13. Comparison of a Single-Precision Floating-Point Division (top) with a double-precision Floating-Point Division (bottom)](image)

**Figure 13.** Comparison of a Single-Precision Floating-Point Division (top) with a double-precision Floating-Point Division (bottom)

![Figure 14. An Order-Preserving Adder Tree and a Shallower, but Less Predictable Adder Tree.](image)

**Figure 14.** An Order-Preserving Adder Tree and a Shallower, but Less Predictable Adder Tree.

For more information about these flags, see the floating-point operations tutorial that is included with the HLS installation (tutorials/best_practices/floating_point_ops).
• Dot-Products

The Intel HLS compiler can make use of Intel hardened dot-product IPs when it detects a design pattern that would benefit from them. The hardened dot-products are more efficient than chaining multiply-adds, so we need to be careful to structure code to take advantage of this. You need to use the `--fp-relaxed` flag to take advantage of this optimization because the hardened dot-product leverages shallow adder trees. In Listing 3, we showed how to unroll a loop to concurrently execute each iteration. While this structure is not able to take advantage of hardened dot-products, it can be modified as shown in Listing 7 below. If we ensure that the entire matrix buffer was initialized to 0 before storing the input data, we can guarantee that the dot-product will still produce correct results since 

\[
\begin{bmatrix}
1 & 2 & 3 \\
1 & 2 & 3 & 0 & 0
\end{bmatrix},
\]

and we can safely remove the predication statement.

```c
1. #pragma unroll
2. float dotProduct = 0.0f;
3. for (int mRow = 0; mRow < ROWS_COMPONENT; mRow++) {
4.      if (mRow > rows)
5.      dotProduct += q_matrix[(mRow * cols) + i] 
6.      * t_matrix[(mRow * cols) + j];
7. }
```

Listing 7: Code that Takes Advantage of Hardened Dot-Products

- Avoiding Division

Floating-point division is an expensive operation, and is in fact more expensive than floating-point multiplication. This means that especially in unrolled loops, multiplications should be used instead of divisions if possible. A good candidate for this is the division operation on line 4 of Figure 1. Instead of dividing by \( \alpha \), it would be better to multiply by \( \frac{1}{\alpha} \). Recall that \( \alpha \) is computed by taking a square root of a sum of squared numbers. HLS provides an inverse-square root function \( \text{rsqrt} \) and \( \text{rsqrtf} \) in `extendedmath.h` that is more efficient than calculating \( 1.0/\sqrt{x} \) (or \( 1.0/\sqrt{f(x)} \)). Listing 9 shows how this may be done. Note that the functions in `extendedmath.h` are not available when targeting the g++ compiler, however they do work when targeting x86-64 with i++. See `MGS.h` for an example of how to create an implementation of \( \text{rsqrtf}() \) that will be compatible with g++.

```c
1. #include "HLS/extendedmath.h"
2. ...
3. #pragma unroll
4. for (int row = 0; row < ROWS_COMPONENT; row++)
5. { // hardened dot-product
6.     float val = t_matrix[row][i];
7.     sum = sum + (Val * Val);
8. }
9. t_magnitude_inv = rsqrtf(sum);
10.
11. ...
```

Listing 9: Replacing Parallel Divisions with Parallel Multiplications
Conclusion

The Intel HLS compiler is a powerful tool that can help streamline the process of creating FPGA designs. It allows a user to develop high-performance designs with good area consumption in a short time (relative to lower-level tools such as RTL). This white paper illustrated how to adjust a standard C++ implementation of QRD to improve performance and area usage using the Intel HLS Compiler. The strategies illustrated in this application note are applicable to a wide range of designs, and can help you fully realize the potential of your Intel FPGA design.

References


Appendix—Explanation of QR Decomposition and Modified Gram-Schmidt Algorithm

The QR decomposition commonly forms the basis of matrix inversion, linear equation solvers, and more. This appendix defines the QR decomposition, and explains the Modified Gram-Schmidt decomposition algorithm.

A square matrix $A$ may be factored as

$$A = QR$$

where $R$ is a square, upper-triangular matrix:

$$R = \begin{bmatrix} R_{11} & R_{12} & \cdots & R_{1n} \\ 0 & R_{22} & \cdots & R_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & R_{nn} \end{bmatrix}$$

and $Q$ is a square, ortho-normal matrix. Let us further define $Q$ as a column matrix:

$$Q = [q_1, q_2, \ldots, q_n]$$

where $q_1, q_2, \ldots, q_n$ are column-vectors of length $n$. Since we have required $Q$ to be orthonormal, each of the columns $q_1, q_2, \ldots, q_n$ is perpendicular to the others ($q_1 \cdot q_2 = 0$), and each column has a Euclidean magnitude of 1.

One popular application of the QR decomposition is matrix inversion. The QR decomposition expresses a matrix $A$, whose inverse may not be obvious, as two matrices that are straightforward to invert. Orthogonal, square matrices such as $Q$ may be easily inverted by computing the transpose, which takes 0 time if clever data structures are used. Upper-triangular matrices such as $R$ may be easily inverted by using back-substitution [1], [3].

$$A^{-1} = R^{-1}Q^{-1}$$
$$Q^{-1} = Q^T$$
$$R * R^{-1} = I$$

Since the inversion itself is simplified, a more interesting problem is how to efficiently compute the $Q$ and $R$ matrices.

The MGS algorithm is a well-understood algorithm for computing the QR decomposition, and is commonly taught in linear algebra classes. In general, MGS operates on an $m \times n$ matrix $A$, where each column is linearly independent (i.e. $A$ is full column-rank) and $m \geq n$. This algorithm is presented in Figure 1. The following notation conventions are used in this document:

- Matrices are referred to using a capital letter, e.g. $A$.
- Elements of a matrix are referred to using subscripts, e.g. $A_{ij}$ refers to an element in matrix $A$ at row $i$ and column $j$.
  - A row or column of a matrix will be denoted by using the $\bullet$ character. E.g. $A_{\bullet j}$ refers to the $j$-th column of $A$, and $A_{i \bullet}$ refers to the $i$-th row of $A$.
- Outputs are in bold.
- Text following a “//” may be considered as comments, and does not affect the flow of an algorithm.
MGS operates by computing the Q-matrix using its characteristic of column ortho-normality. That is, each column of the Q-matrix is orthogonal (each column is perpendicular to the others) and normal (has Euclidean norm of 1: || Q || = 1). The Gram-Schmidt process is used for constructing ortho-normal vector spaces from matrices, and can therefore be used for computing the QR decomposition. The R-matrix may be generated after computing the Q-matrix using the relation $R = Q^T A$, or it may be generated using intermediate calculations (Figure 15).

The algorithm illustrated in Figure 15 is described in greater detail:

1. The $A$ matrix is copied to a temporary matrix $T$, since it would need to be modified to compute the $Q$-matrix. The data in the $T$-matrix is then processed column-by-column.

2. & 3. The $R$-matrix will be computed at the same time as the $Q$-matrix (since its values are intermediate values used in computing the $Q$-matrix) and it will be constructed row-by-row, beginning with each element on the main diagonal. For the $i$-th row, the element of the main diagonal, $R_{ii}$, is the Euclidean magnitude of the $i$-th column in the $T$-matrix (that is, $R_{ii} = \alpha_i$).

4. The $i$-th column of the $T$-matrix is orthogonal with all the previous columns (i.e. $T_{i} \cdot T_{i-1}$), so it is normalized and copied into $Q$.

5. The remaining columns of the $T$-matrix are now modified by subtracting any components of the $i$-th column of the $T$-matrix. Therefore, if any of the columns of the $A$-matrix are linearly dependent, they will be eliminated and cause an error. Dot products are used to determine components of $T_{i} \cdot T_{j}$ in the remaining columns of $T$.

6. These dot-products are the values needed by the $R$-matrix, so we store them now rather than computing the $R$-matrix later.

7. Perform the subtraction: $T_{ij} = T_{ij} - (Q_{ij} \cdot T_{ij}) \cdot Q_{ij}$