

An Investigation of the QR Decomposition Algorithm on Parallel Architectures

Vito Dai and Brian Limketkai

Abstract— This paper presents an implementation of a QR decomposition algorithm on both the Trimedia VLIW processor and the Berkeley V-IRAM vector processor. A comparison based on the number of cycles required to execute QR decomposition is made to highlight the different architectural features of the processors which determine the performance of the QR algorithm. Using Givens rotations, implemented by CORDICs, as the algorithm for QR decomposition, it is found that the VLIW outperformed the V-IRAM for small matrices, but degraded quadratically as matrix size increased.

I. INTRODUCTION

QR decomposition is a matrix factorization used to facilitate the solution of linear systems. It is often used to perform adaptive digital signal processing in modern communications systems. While other matrix factorizations do exist, QR decomposition is often chosen over other methods because of its simplicity and ease of implementation.

With the current growth of high-performance wireless research, the need for adaptive computation has sparked an increased interest in the use of QR decomposition in wireless systems. QR decomposition can be used in multiple-antenna systems to direct the focus of an antenna array through adaptive beamforming techniques. With the higher data rates employed by current systems, it is very clear that the QR kernel must execute faster in order to process the incoming data from the antennas in real-time. For this reason, many communication systems implement QR using ASICs (Application-Specific Integrated Circuits) to achieve the highest performance possible. However, one must not forget that other computations must execute in addition to the QR kernel and the programmability of these other functions may be desirable. Hence, it is the purpose of this paper to investigate the capabilities of programmable processors when executing the QR kernel.

This paper describes the implementation of a particular algorithm for QR decomposition, using Givens rotations, on two different parallel processors. The two processors, the Trimedia VLIW processor and the Berkeley V-IRAM vector processor, are chosen because of their differing methods of exploiting parallelisms. Interesting results are derived by comparing the performance of the two processors when executing the QR algorithm, and by observing how each processor takes advantage of the different levels of parallelism inherent in Givens rotations.

Section II of this paper describes QR decomposition in more detail, focusing mainly on Givens rotations. When implemented by CORDICs (COordinate Rotation DIGital Com-

puter), Givens rotations are easily adapted to using fixed-point arithmetic, thus leading to a simple and efficient realization of the QR algorithm on several architectures. Section III then introduces the two processors, first motivating the use of a VLIW processor followed by the arguments for a vector processor. Architectural features specific to the Trimedia VLIW processor and the Berkeley V-IRAM are also presented. A mapping of the QR algorithm onto each processor is proposed and implemented. Section IV presents the results of this implementation followed by concluding remarks in section V.

II. QR ALGORITHM

Among the many different ways of implementing QR decomposition, one of the simplest and most easily implementable algorithms uses Givens rotations. Other techniques, such as Householder transformations and Squared Givens Rotation, can be used instead, but the QR algorithm using Givens rotations is chosen because it is easily adapted to using fixed-point arithmetic. Moreover, this QR algorithm allows for incremental updates of the R matrix, making it attractive for use in a multiple-antenna system which continuously samples data from the antennas.

A. Givens Rotations

QR decomposition is a matrix factorization which decomposes any matrix A into a product of an orthogonal matrix Q and an upper-triangular matrix R .

$$A = QR \quad (1)$$

From the point of view of a multiple-antenna system, A is a data-matrix which contains data sampled from multiple antennas. Each row of A contains N elements representing signal values at each of the N antennas sampled at a particular instance in time. As more time samples are collected, the A matrix expands vertically to include more rows, thus requiring the Q and R matrix to be updated accordingly. Using Givens rotations, this incremental update of Q and R is easily realizable.

A Givens rotation is simply a rotation of a 2-dimensional vector in a plane, which can be represented by the following orthogonal matrix.

$$Q(\theta) \triangleq \begin{pmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{pmatrix} \quad (2)$$

With each added row of A , the QR algorithm must find a set of N Givens rotations that will correctly update the upper-triangular matrix R . Since the product of orthogonal matrices is simply another orthogonal matrix, finding this set of rotations is equivalent to finding the incremental update of the Q matrix which is never explicitly computed. How these rotations are found and applied is shown by the systolic array in Fig. 1 for the case $N = 6$.

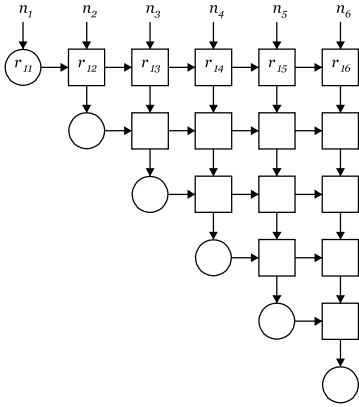


Fig. 1. Systolic QR Array.

The circles and squares in Fig. 1 store the elements of the last update of the upper-triangular matrix R . At each time interval, six samples of antenna data, $n_1 \dots n_6$, are collected and passed down to the top row of the QR array. Each row of this array then performs a Givens rotation that zeroes one of the input samples and modifies the remaining samples. These remaining samples are passed on to the next row of the array and the operations repeat. For example, the first row of the QR array accepts six sampled antenna values. It performs one Givens rotation on the column vector formed by (r_{11}, n_1) to eliminate n_1 . The same rotation is then applied to the 5 remaining column vectors formed by $(r_{12}, n_2) \dots (r_{16}, n_6)$. Since the elements in rows $2 \dots 6$ of the array represent vector components orthogonal to the plane of rotation, these elements are not affected by the rotation. The residual values for $n_2 \dots n_6$ are then passed down to the second row. Repeating the operations, each row successively zeroes one input, finishing when the sixth input, n_6 , has been nulled, thus completing the update of R .

To find and apply a Givens rotation, each row performs two types of operations: *vectorize*, represented by the circle, and *rotate*, represented by the square. These operations are shown graphically in Fig. 2. The *vectorize* operation considers the 2-dimensional vector (x_i, y_i) and computes the magnitude x_o and angle θ as depicted in Fig. 2a. The magnitude is then stored in the current position of the R matrix. The *rotate* operation simply rotates the 2-dimensional vector (x_i, y_i) by θ . The *rotate* operation (Fig. 2b) then stores the modified component x_o in the current position of the R matrix while passing the residual component y_o down to the next row.

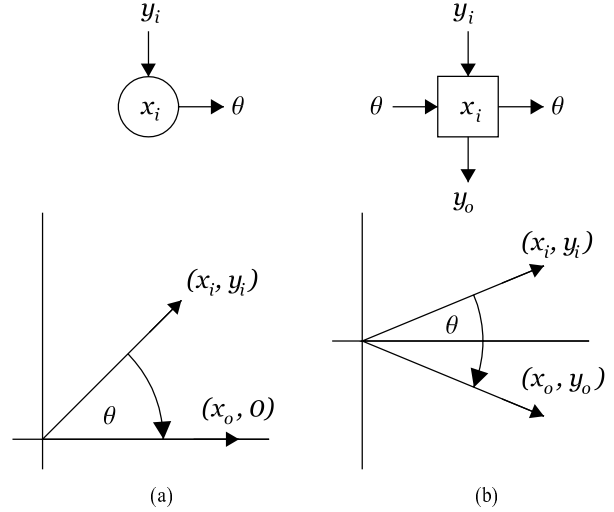


Fig. 2. (a) Vectorize and (b) rotate functions.

Clearly, the *vectorize* and *rotate* operations can be implemented directly using floating-point numbers and trigonometric functions found in C-libraries. However, better performance can be achieved by using other methods. This paper presents the implementation and performance of QR using a fixed-point CORDIC algorithm [2] on both the Trimedia VLIW processor and the Berkeley V-IRAM vector processor. Their performance is compared against a basic floating-point implementation using floating-point functions, such as `sin`, `cos`, `arctan` and `sqrt`, on the VLIW processor and a fixed-point CORDIC implementation on a standard superscalar architecture.

III. IMPLEMENTATION OF THE QR ALGORITHM ON TWO PARALLEL ARCHITECTURES

The two processors explored in this project, the Trimedia TM-1300 VLIW processor and the Berkeley V-IRAM vector processor, are chosen because of the stark contrast in how they exploit data independence. The amount of data independence that can be exploited depends on both the algorithm used to compute QR and the architecture used to implement it.

A. Trimedia TM-1300 VLIW Processor

The Trimedia TM-1300 is a 5-way issue VLIW processor. The number of operations that can be performed depends on both the presence of data hazards, and the number of functional units available. In Table 1, the number of functional units of various types are listed as well as their performance in terms of latency, the number of cycles before results may be used, and recovery, the number of cycles before units may be reused. For the Trimedia chip, there are 5 integer adders which can be used every cycle, producing results one cycle after issue. In contrast, there are only 2 floating-point adders, and these produce results 3 cycles after issue. In general, the performance

potential of the VLIW architecture is realized by issuing more independent operations each cycle. The presence of faster, more numerous integer units suggest that a fixed-point implementation might perform better than a corresponding floating-point implementation.

TABLE I
TRIMEDIA TM1300 VLIW PROCESSOR

| Functional Unit | Quantity | Latency | Recovery |
|-----------------|----------|---------|----------|
| Integer ALU | 5 | 1 | 1 |
| Shifter | 2 | 1 | 1 |
| Int/float MUL | 2 | 3 | 1 |
| Float ALU | 2 | 3 | 1 |
| Float sqrt/div | 1 | 17 | 16 |

The results of the basic floating-point QR algorithm implemented on the Trimedia processor using trigonometric functions is listed in the first row of Table 2. On average, 1.76 instructions are issued per cycle, which is slightly lower than the number of floating-point adders and multipliers available. Using a CORDIC implementation, which replaces trigonometric functions with integer shifts and adds, results in a much faster implementation as shown in the second row of Table 2. The VLIW hardware is better utilized, averaging 2.61 operations per cycle.

TABLE II
SIMULATED CYCLE COUNTS AND AVERAGE ISSUE

| QR Algorithm | Cycles | Average Issue |
|---------------------|--------|---------------|
| Floating Point | 141k | 1.76 |
| C CORDIC | 6.6k | 2.61 |
| Assembly CORDIC | 4.8k | 2.54 |
| C improved | 4.4k | 2.97 |
| Manual loop unroll | 2.21k | 3.7 (4.19) |
| C optimized | 2.14k | 3.63 (4.09) |
| C with 2-way rotate | 2.08k | 3.72 (4.70) |
| C rotate 2 (N=32) | 37.1k | 4.25 (4.70) |

To ensure that architectural performance is not limited by the compiler, the CORDIC algorithm is also implemented in assembly, as shown in the third row. Using predicated instructions as an alternative to branching, both branches of an `if`-clause can be simultaneously evaluated. By issuing both predicated operations on the same VLIW instruction, a branch is evaluated in a single cycle. This technique increases the performance of the CORDIC while slightly decreasing the average issue per instruction, as shown on the third row.

Learning from the assembly code, the C code is modified to induce the compiler into using predicated instructions. This leads to increased performance beyond that of the assembly code, and once again improves the average operation issued per cycle to 2.97 as shown in the fourth row of Table 2. Going one step further, the average issue per cycle can be improved by unrolling the CORDIC inner loop.

The CORDIC algorithm involves an inner loop with a fixed number of iterations equal to the number of bits of precision required. Exploiting loop-level parallelism, this loop can be unrolled manually or by the compiler to increase the number of issues per cycle. As seen in the fifth and sixth rows of Table 2, both manual (assembly) and automated (C) loop unrolling perform similarly. Performance is increased by a factor of 2 and the average number of issues increases to 3.7 per cycle. Shown in parentheses is the average issue of the `vectorize` and `rotate` operations alone, which is about 4.1 per cycle, nearing the peak issue of 5 operations per cycle.

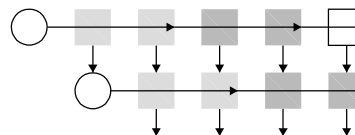


Fig. 3. 2-way rotate function.

From a higher level point of view, once the angle of rotation θ is determined by a `vectorize` operation, all the (x, y) pairs in the same row can be independently rotated by the same angle. Thus, taking advantage of this higher level of parallelism, `rotate` operations can be grouped in pairs, as depicted by the shaded squares in Fig. 3, leading to an increase in average issue while decreasing total cycle count. Using 2-way rotates, `rotate2`, increase the number of operations being performed in parallel, resulting in an average issue of 3.72, as shown in the seventh row of Table 2. Considering `rotate2` alone, its average issue, shown in parenthesis, is 4.7, which is much higher than the overall average issue of 3.7. This large discrepancy results because the `vectorize` operation cannot be parallelized in this fashion, nor can some of the `rotate` operations, as shown by the circles and non-shaded squares in Fig. 3. By increasing N , the size of the matrix, from 6 to 32, the relative number of 2-way rotates increases, resulting in a higher average issue of 4.25, as shown in the last row of Table 2.

B. Berkeley V-IRAM Processor

The Berkeley V-IRAM processor is a vector processor capable of issuing one vector instruction per cycle. With two 256-bit execution units, the V-IRAM can perform 16 32-bit integer operations per cycle, offering a much higher computational throughput than the VLIW processor, if all operations can be vectorized. However, this parallel processing cannot be applied to the inner loop of the CORDIC

algorithm, since each iteration of the CORDIC loop depends upon the result of the previous iteration. Thus, a different level of parallelism that is suitable to vector processing would have to be identified in the QR algorithm.

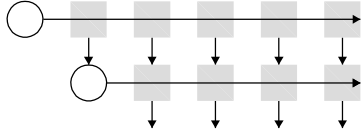


Fig. 4. `vector-rotate` function.

Like the VLIW processor, the V-IRAM can take advantage of the higher-level parallelism due to the independence of rotate elements in a row. However, the vector processor has the ability to execute all of the rotate operations at once, without incurring significant performance degradations, as depicted by the shaded squares in Fig. 4. This means that each row of the array is updated using a fixed number of operations regardless of matrix size. Hence, the larger the matrix, the more parallelism the V-IRAM architecture is capable of exploiting, leading to better performance overall as the number of operations scale linearly with the matrix size. This contrasts the quadratic scaling of a VLIW processor, where the number of operations per row is proportional to the number of elements in the row.

IV. RESULTS

The results for the Trimedia VLIW processor are based on simulation data using `tmsim` and `tmprof`, the TM-1x00 simulator and profiler provided by Philips. The results for the Berkeley V-IRAM vector processor come from simulation data using `vsim-p`, the IRAM profiler and simulator. The QR algorithm was run 1,000 times to weigh the QR kernel more heavily over the surrounding code.

A. Cycle Count and Average Issues

The average number of cycles used to execute the QR program on the Trimedia processor is listed in Table 2. From these results, it can be seen that loop unrolling leads to significant improvements on the VLIW over simple unrolled implementations. Table 2 also shows the average number of issue slots utilized per execution of the code. The large increase in utilization after unrolling the loop corresponds to the decrease in cycle count, implying performance benefits due to a more efficient implementation.

Because the V-IRAM C compiler (`vcc`) was not fully functional at the time of this project, all of the code for this vector processor was written in assembly. Thus, the QR algorithm was implemented only once, using all of the techniques learned from the VLIW processor, such as loop unrolling and CORDICs.

A comparison of the two processors can best be seen by looking at Fig. 5. Here, the total number of cycles used to execute 10 iterations of the QR kernel is plotted versus the

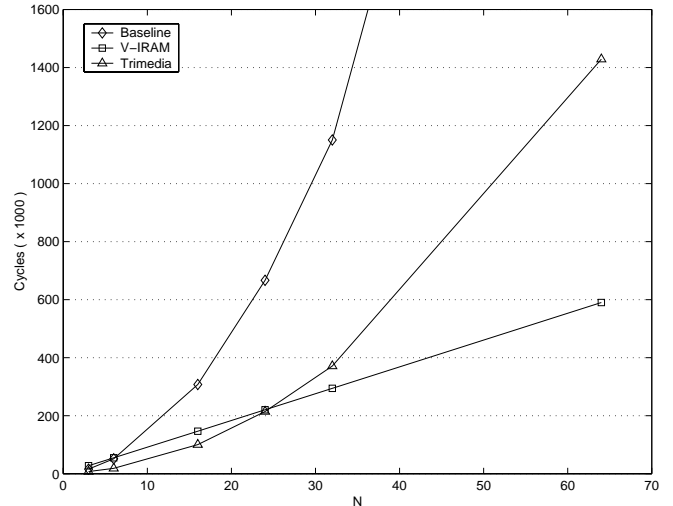


Fig. 5. Number of cycles required to execute QR code with 10 iterations of the QR kernel versus matrix size.

matrix size. A dual-issue superscalar MIPS processor was used as a baseline of comparison. From Fig. 5, it is clear that the execution of the QR kernel increases quadratically for both the VLIW and baseline processor, while execution on the vector processor scales only linearly. As mentioned before, this is because the number of operations required to complete a vectorize and row rotate is constant in the vector processor. An interesting point to note is the intersection of the VLIW and V-IRAM curves. At $N = 24$, both processors perform equally well with respect to the QR algorithm. For matrix sizes larger than this, the vector processor outperforms the VLIW because of the linear versus quadratic scaling. However, one should always keep the application in mind when interpreting these results; 24 antennas may not be very realistic in a multiple-antenna radio.

A magnified view of Fig. 5 is shown in Fig. 6. For $N = 3$, the V-IRAM executes at almost twice as many cycles as the baseline processor to execute the QR algorithm. For such small matrices, there is little that the vector processor can perform in parallel. Therefore, the baseline processor outperforms the V-IRAM because it has a dual-issue superscalar core, as opposed to the single-issue simulated by the V-IRAM. Going from $N = 3$ to $N = 6$, the linear versus quadratic trend is visible as the vector processor merely doubles in cycle count, as opposed to the baseline which quadruples with the doubling of the matrix size.

B. Speedup

Another interesting performance metric to look at is the speedup achieved by the two parallel architectures relative to the baseline processor (Fig. 7). Because the VLIW processor can exploit ILP in the CORDIC loop, it achieves close to its maximum speedup for small values of N . As N increases, the added parallelism from the 2-way rotate offers a slight increase in the speedup.

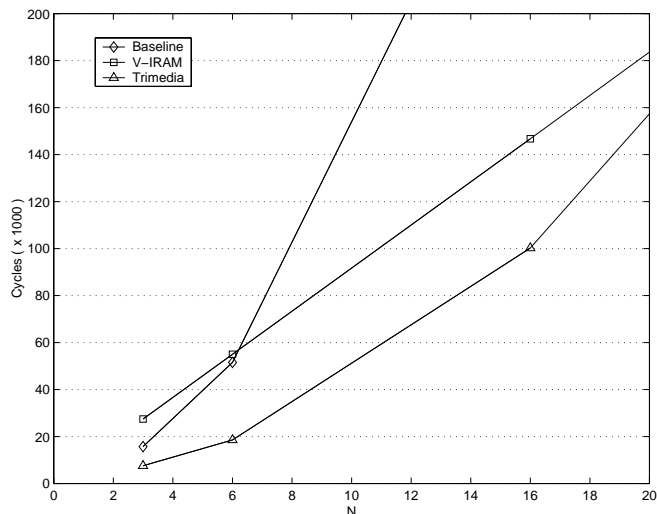


Fig. 6. Magnified view of Fig. 5.

The maximum simulated speedup for the VLIW processor is about 3. Although rough calculations say that a 5-way issue VLIW should have a speedup of 2.5 over a dual-issue superscalar, the actual speedup is usually slightly higher since the superscalar may not be able to issue 2 instructions each cycle.

In contrast to the VLIW, the V-IRAM starts off with a speeddown of 0.5 for small values of N . With no data independence to exploit, the vector code simulates in just as many cycles as a scalar processor, but with only a single-issue execution pipe. Nevertheless, the performance of the V-IRAM processor increases dramatically as the matrix size increases. The linear scaling in cycle count leads to a linear increase in the speedup.

However, this linear increase is not without bound. Because the V-IRAM can only operate on 16 words each cycle, the maximum speedup is expected to be slightly higher than 8 since the dual-issue superscalar may not necessarily issue two instructions per cycle.

C. Real-time Issues

The real-time issues of the QR algorithm must be considered when deciding whether the use of programmable processors is feasible in a real implementation of a multiple-antenna system. Faced with the computational challenge of decoding streams of data from multiple antennas, most multiple-antenna radios today only consider a few antennas. The results of this project indicate that the Trimedia VLIW processor may be suitable for performing the digital backend processing in a real system.

In order to get rough estimates on the maximum amount of time allowable to do QR decomposition, the sampling period of antenna data must be considered. To find the sampling period, consider a simple system with a simple

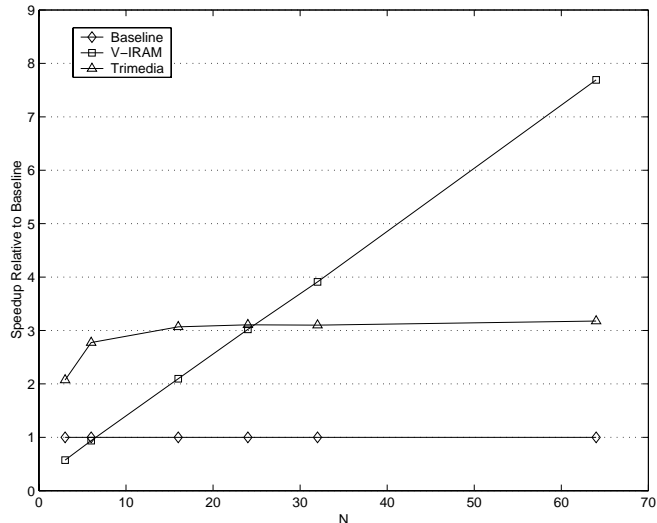


Fig. 7. Speedup of the VLIW and vector processors relative to a dual-issue superscalar MIPS processor.

modulation scheme such as QPSK (Quadrature Phase-Shift-Keying) without channel coding and 5 antennas. The total rate is then 10 bits/symbol (2 bits/symbol for QPSK multiplied by 5 antennas). Assuming a wireless ethernet application with an aggregate data rate of 10 Mbps, the sampling rate is roughly 1 Msymbol/s. This corresponds to a sampling period of 1 μ s.

The Trimedia VLIW chip [3] can be clocked at up to 166 MHz. With the average cycle count of about 2000 cycles/iteration for the fastest QR kernel simulated, the time required to execute the QR kernel can be as low as 12 μ s. Thus, the simulations of the VLIW processor in this project are still a factor of 10 from being usable in this example application.

However, by reducing the number of antennas to 4 and relaxing the data rate to about 100 kbps, the maximum allowable time for executing the QR kernel increases to 80 μ s, permitting the QR algorithm to execute easily with time to spare. Thus, the actual feasibility of using QR on any of these parallel processors depends very strongly on the specifics of the application.

V. CONCLUSIONS

Implementations of the QR algorithm on two parallel architectures, a VLIW and a vector architecture, were presented. It was found that the VLIW processor outperformed the vector processor for small matrix sizes, as it was able to exploit the low-level parallelism present in the CORDIC algorithm. However, larger matrices offered more parallelism that the vector processor could capitalize on, leading to more efficient scaling on the vector processor. When applied to specific real-time applications, the VLIW processor could be a feasible choice if a programmable processor is desired without requiring the absolute best performance.

REFERENCES

- [1] Walke R.L. and Smith R.W.M., "Architectures for Adaptive Weight Calculation on ASIC and FPGA," *Some journal*, 1999.
- [2] Volder J., "The CORDIC Trigonometric Computing Technique," *IRE Trans. Electron. Comput.*, vol. EC-8, pp. 330–334, 1959.
- [3] Philips Semiconductor, *Trimedia Documentation Set 2.1*. Philips, 2000.
- [4] IRAM team, *Vector IRAM Documentation*. CS Dept. U.C. Berkeley, 2000.
- [5] Hennessy J.L. and Patterson D.A., *Computer Architecture: A Quantitative Approach 2nd Edition*. Morgan Kaufmann, 1996.