Parallelizing QR Decompositions with the R-Stream Compiler

Allen Leung, Nicolas Vasilache, Benoît Meister, Richard Lethin
Reservoir Labs

May 22, 2008

1 Introduction

Rapid performance of the QR decomposition on incoming data matrices is a key computational element in building Square-Root RLS linear adaptive filters, a component in many advanced embedded sensor systems (radar, etc) [Hay02]. While efficient systolic implementations of QR decomposition based on Givens rotations have been known for close to two decades, reported implementations are still mostly hand crafted affairs [SBM00, RSVN01, NHE+05]. In our efforts to develop technologies for programming high performance embedded computers, we have recently been experimenting with the automatic parallelization of QR decomposition algorithms with the R-Stream compiler [LLM+08]. In this paper we show how our compiler can automatically extract parallelism suitable for a variety of emerging computational platforms (accelerators, FPGA, multicore, etc.) from a single sequential QR implementation.

2 Givens QR

We start with a sequential implementation of QR decomposition using Givens rotations :

```
for (int k = 0; k < N-1; k++)
  for (int i = N-2; i >= k; i--)
    float a = A[i][k];
    float b = A[i+1][k];
    float d=sqrt(a*a+b*b), c=a/d, s=-b/d;
    for (j = k; j < N; j++)
      float t1 = A[i][j]*c + A[i+1][j]*s;
      float t2 = A[i+1][j]*c - A[i][j]*s;
      A[i][j]   = t1;
      A[i+1][j] = t2;
```

The parallelization steps are as follows. The R-Stream compiler first normalizes the above loop nests and puts it in an internal polyhedral form where all dependencies are expressed as polyhedral constraints (the indicators <, > and <> specify whether a variable is read-only, write-only or read/write). Since the loop nests involve only affine indexing functions and affine loop bounds, R-Stream can encode the dependencies precisely, without approximations.

In the example below, the original code has been converted and the statement S_0 reads array value A[1023-j,i] and writes the internal variable _v1 that corresponds to variable 'a' in the original program. S_3 results from the aggregation by R-Stream into a single statement of all the 4 statements originally below loop `j`. Also when pretty-printing its output, R-Stream reindexes the loops in the lexicographic order `i,j,k` but this does no correspond to any transformation.

```
for (int i = 0; i <= 1022; i++)
  for (int j = 0; j <= - i + 1022; j++)
    S0(<A[1023-j,i], >_v1);
    S1(<A[1022-j,i], >_v2);
    S2(<_v1, <_v2, >_v3, >_v4);
    for (int k = 0; k <= - i + 1023; k++)
      S3(<>A[1022-j,i+k], <>A[1023-j,i+k],
         <_v3, <_v4);
```

After performing scalar expansion to remove false dependencies, we obtain the following loop nests where variables _v1, ..., _v4 (originally a, b, c, s) have been expanded to bidimensional arrays:

```
for (int i = 0; i <= 1022; i++)
  for (int j = 0; j <= - i + 1022; j++)
    S0(<A[1023-j,i], >_v1[i,j]);
    S1(<A[1022-j,i], >_v2[i,j]);
    S2(<_v1[i,j], <_v2[i,j],
       >_v3[i,j], >_v4[i,j]);
    for (int k = 0; k <= - i + 1023; k++)
      S3(<>A[1022-j,i+k], <>A[1023-j,i+k],
         <_v3[i,j], <_v4[i,j]);
```

Our compiler computes a schedule which maximizes the amount of coarse-grained parallelism (both synchronization-free and pipelined) while simultaneously maximizing the amount of possible reuse based on locality. The search is performed in a unified manner using a single integer linear program (ILP) over the entire feasible space of schedules. In this example, it finds a set of loop nests with two outer permutable dimensions (i and j) and an inner vectorizable loop (k). The two outermost permutable loops can then be skewed and yield a pipelined parallel loop at the second level.

```
for (int i = 0; i <= 1022; i++)     // p=0
  for (int j = i; j <= 1022; j++)   // p=0
    S0(<A[1023+i-j,i], >_v1[i,-i+j]);
    S1(<A[1022+i-j,i], >_v2[i,-i+j]);
    S2(<_v1[i,-i+j], <_v2[i,-i+j],
       >_v3[i,-i+j], >_v4[i,-i+j]);
    doall (int k = 0; k <= - i + 1023; k++)
      S3(<>A[1022+i-j,i+k],
         <>A[1023+i-j,i+k],
         <_v3[i,-i+j], <_v4[i,-i+j]);
```

Depending on the specific architecture, more transformations have to be applied in order to obtain a target code that properly exploits the processing elements. For example, to target a distributed memory, multicore architecture, blocking can be applied to the permutable loop nest to form tasks that can be executed atomically without internal communication. The block/task sizes are chosen so that (*i*) each task can fit into the distributed memory of the processing elements, and (*ii*) communication between tasks are minimized. Tasks can be further scheduled to expose task level parallelism. Bulk communications at the boundary of the tasks can be grouped into DMA operations, and these are overlapped with computations via multi-buffering schemes. In the case of Givens QR, we obtain a final mapping that executes as a wave-front of coarse-grained tasks to exploit multiple processors. Each such coarse-grained task can then be subdivided by hierarchical tiling into local tasks that would exploit potential multi-threading capabilities and/or SIMD processing elements. A further level of tiling can also allow us to create fine-grained tasks whose operands fit the register file and unroll to remove spurious instructions at the assembly level taking advantage of register reuse.

Due to space constraints, the final output code has been omitted.

3 Modified Gram-Schmidt QR

The next algorithm that we consider is an implementation of a modified Gram-Schmidt version of QR, as shown below:

```
for (int k = 0; k < N; k++)
  float nrm = 0;
  for (int i = 0; i < M; i++)
    nrm += A[i][k] * A[i][k];
    R[k][k] = sqrt(nrm);
    for (int i = 0; i < M; i++)
      Q[i][k] = A[i][k] / R[k][k];

    for (int j = k+1; j < N; j++)
      R[k][j] = 0;
      for (int i = 0; i < M; i++)
        R[k][j] += Q[i][k] * A[i][j];
      for (int i = 0; i < M; i++)
        A[i][j] -= Q[i][k] * R[k][j];
```

After performing array expansion and scheduling, our compiler obtains the following parallelized loop nests. In this example, certain synchronization barriers have been introduced between the inner parallelized loop nests to ensure correctness. A more care-

ful look at the dependencies allows us to remove some of the barriers which correspond to independent regions that are equivalent to the OpenMP construct *#pragma omp parallel region*. Eventually, the discussion in the previous paragraph on hierarchical tiling for tasks creation and locality optimization also applies here.

```
// prologue elided
for (int i = 0; i <= 1022; i++)
  for (int j = 0; j <= 1023; j++)
    S1(<A[j,i], <>nrm[i]); // reduction
  S2(>R[i,i], <nrm[i]);
  doall (int j = 0; j <= 1023; j++)
    S3(<R[i,i], >Q[j,i], <A[j,i]);
  // barrier
  doall (int j = 0; j <= - i + 1022; j++)
    for (int k = 0; k <= 1023; k++)
      S5(<>R[i,1+i+j], <Q[k,i], <A[k,1+i+j]);
    doall (int k = 0; k <= 1023; k++)
      S6(<R[i,1+i+j], <>A[k,1+i+j], <Q[k,i]);
    // barrier
  // barrier
// epilogue elided
```

4 Conclusion

This paper shows how R-Stream automatically extracts parallelism from QR decomposition algorithms. Our scheduler balances the extraction of coarse-grained parallelism with reuse that can be utilized downstream. Output from the scheduler can be further rendered to a variety of execution models and hardware execution engines (such as distributed memory, bulk-communication, SIMD, shared memory, FPGA). This provides potential performance, productivity, and portability benefits for embedded signal processing codes. The results from the automatic mapping procedure are along the lines of the structures produced and used by domain experts in hand mappings.

Note that these original algorithm expressions are specifically lacking directives to indicate parallelism or other physical mapping considerations (e.g., as in OpenMP or with Parallel VSIPL++). A streaming programming language (e.g., StreaMIT, Brook) is not needed; it would probably be a lot of work to express these algorithms in those languages at all, let alone with the multiple dimensions of parallelism that lead to efficient renderings on hardware. The compiler can detect, shape, and tradeoff the physical consideration automatically from C. Like the long forgotten `register` annotation for variables in C, such physical mapping annotations for parallelism may actually be obsolete thanks to better compiler technology. In many cases, it can be worse; such physical mapping information in the original algorithm expressions can actually hamper the portability that can be obtained from automatic mapping tools, because it imposes an often insurmountable burden of undoing the physical mapping that may (or may not) be good for one architecture before the computation can be efficiently mapped.

However, the mapping process does not alleviate all the programming effort. Our mapper requires the expression to be in a form that can be raised to the polyhedral form. In the case of QR this is not burdensome; we find that textbook expressions of the two relatively different algorithms for QR are nearly exactly in this form.[1]. Furthermore, this expression can be easily achieved using an existing language; a new programming language is not needed.

We obtain the results from a general automatic mapping procedure based on polyhedral representations; this result is not from a tool specific to the QR algorithm itself (a sort of "fastest QR in the West" tool analagous to FFTW). This tool works with other algorithms, or on codes with QR expressiing in a larger pipeline. We caution, though, that there is still work to do to render FFT itself in the polyhedral domain.

This suggests that the route to more portable code libraries is through less specificity and higher levels of abstraction, to enable the use of powerful emerging automatic mapping technologies.

[1] Householder fits easily in the form too and is automatically parallelisable; it is omitted here for space reasons, but will be discussed during the workshop.

References

[Hay02] S. Haykin. *Adaptive Filter Theory*. Prentice Hall, 2002.

[LLM+08] Richard Lethin, Allen Leung, Benoît Meister, Peter Szilagyi, Nicolas Vasilache, and David Wohlford. Final report on the the R-Stream 3.0 compiler. Technical report, Reservoir Labs, Inc. Delivered to Air Force Research Laboratory, Rome, NY for Contract F03602-03-C-0033, May 2008.

[NHE+05] H. Nguyen, J. Haupt, M. Eskowitz, B. Bekirov, J. Scalera, T. Anderson, M. Vai, and K. Teitelbaum. High-performance FPGA-based QR decomposition. In *HPEC*, 2005.

[RSVN01] D. V. Rabinkin, W. Song, M. M. Vai, and H. Nguyen. Adaptive array beamforming with fixed-point arithmetic matrix inversion using Givens rotations. In *Proceedings of SPIE*, volume 4474, 2001.

[SBM00] W. S. Song, E. J. Baranoski, and D. R. Martinez. One trillion operations per second on-board VLSI signal processor for Discoverer II space-based radar. In *Proceedings IEEE Aerospace Conference*, 2000.