electronics

MDPI

Article

Exploration of FPGA-Based Hardware Designs for QR Decomposition for Solving Stiff ODE Numerical Methods Using the HARP Hybrid Architecture

Carlos Alberto Oliveira de Souza Junior [1,*,†], João Bispo [2,†], João M. P. Cardoso [2,†], Pedro C. Diniz [3,†] and Eduardo Marques [1,†]

1 Institute of Mathematics and Computer Science, University of São Paulo, São Carlos 13566-590, Brazil; emarques@icmc.usp.br
2 Department of Informatics Engineering, Faculty of Engineering, University of Porto and INESC TEC, 4200-465 Porto, Portugal; jbispo@fe.up.pt (J.B.); jmpc@fe.up.pt (J.M.P.C.)
3 INESC-ID, 1000-029 Lisboa, Portugal; pedro.diniz@inesc-id.pt
* Correspondence: caosjr@usp.br
† These authors contributed equally to this work.

Received: 10 April 2020; Accepted: 14 May 2020; Published: 19 May 2020

Abstract: In this article, we focus on the acceleration of a chemical reaction simulation that relies on a system of stiff ordinary differential equation (ODEs) targeting heterogeneous computing systems with CPUs and field-programmable gate arrays (FPGAs). Specifically, we target an essential kernel of the coupled chemistry aerosol-tracer transport model to the Brazilian developments on the regional atmospheric modeling system (CCATT-BRAMS). We focus on a linear solve step using the QR factorization based on the modified Gram-Schmidt method as the basis of the ODE solver in this application. We target Intel hardware accelerator research program (HARP) architecture with the OpenCL programming environment for these early experiments. Our design exploration reveals a hardware design that is up to 4 times faster than the original iterative Jacobi method used in this solver. Still, even with hardware support, the overall performance of our QR-based hardware is lower than its original software version.

Keywords: ODE; linear-solver; QR factorization; parallel; heterogeneous-system; FPGA; OpenCL; Intel HARP architecture

1. Introduction

Several engineering problems that rely on physical laws and relations can be modeled in the form of differential equations classified, in general, as either ordinary differential equation (ODE) or partial differential equation (PDE) [1].

In this article, we focus on the chemical reaction problem, an ordinary differential equation, in the Brazilian atmospheric model application code CCATT-BRAMS (http://brams.cptec.inpe.br/). This problem relies on a system of stiff ODEs using the Rosenbrock which is organized as a series of 4 linear-solver steps, each of which currently uses a sequential implementation of LU decomposition named Sparse 1.3a.

In this work, we explore the implementation of a direct method, the QR factorization, rather than the iterative Jacobi LU factorization method currently used in CCATT-BRAMS. The current LU factorization is sequential, includes an expensive update operation due to its pivoting, even though it uses a sparse format to decrease the amount of computing. We opted to implement QR, a direct method without pivoting that exhibits some degree of concurrency which we explore in a field-programmable gate array (FPGA) device implementation.

This approach offers performance potential at various levels. Namely, fine-grain parallelization of the QR factorization itself, avoid thread divergence issues present in software implementations due to distinct time stepping, the ability to exploit performance trade-offs using different and non-standard floating-point precision.

Besides, we explored two variants of the modified Gram-Schmidt QR. As a vehicle for our experiments, we rely on the hybrid HARP architecture from Intel using the OpenCL high-level programming language. We used an Arria-10 FPGA coupled to a CPU, which allowed us better communication and floating-point DSP blocks.

The results reveal that our modified implementation, tuned in OpenCL, is approximately 4 times faster than the original unmodified algorithm of QR. We used some reordering of the algorithm, already described in the literature [2], to improve wall-clock time performance and FPGA area usage. That reordering also allowed us to explore floating-point optimizations on the compiler side. Still, and mainly due to data communication overhead to and from the CPU, the hybrid CPU + FPGA implementation of the modified QR did not improve the performance over the original software implementation of this application.

This article is organized as follows: In Section 2, we describe the CCATT-BRAMS application. In Section 3, we present iterative and direct methods for solving the linear systems that arose from chemical modeling. In Section 4, we show the results we obtained with our linear system solvers. In Section 5, we describe the related work. We conclude in Section 6.

2. CCATT-BRAMS Application Characterization

The CCATT-BRAMS application code implements a coupled chemistry aerosol-tracer transport model for the Brazilian regional atmospheric modeling system. That is an online regional chemical transport model designed for local and regional studies of atmospheric chemistry from the surface to the lower stratosphere specifically developed to handle the tropics. This application, suitable both for operational and research purposes runs operationally at CPTEC/INPE (CPTEC is the center of weather forecasting and climate studies, and it belongs to the national institute for space research (INPE) [3]) since 2003; it covers entire South America with a spatial resolution of 25 km enabling the prediction of the emission of Gases and Aerosols in real time (http://meioambiente.cptec.inpe.br/), as well as meteorological variables (http://previsaonumerica.cptec.inpe.br/golMapWeb/DadosPages?id=CCattBrams). Currently, the BRAMS source code is mostly in Fortran 90, with a small C/C++ code portion used in performance-sensitive sections.

We profiled the CCATT-BRAMS on a Stratix V platform with Gprof [4] using real input data from CPTEC/INPE/ weather forecasting for the profiling and considered two scenarios: (A) with the chemical module disabled and (B) enabled. These profiling results reveal that the community aerosol and radiation model for atmospheres (CARMA) radiation and chemical reactivity steps are the most computationally intensive.

Without the chemical module, radiation is responsible for 75% of the total execution time. When we enable the chemical module, we observe that the ODEs from the chemical reactivity account for 52% of the total execution time, and the radiation takes only 19%. As the radiation function interacts with the chemical module, this function is always active, and as expected, the most expensive operation in this chemical module involves the solution of linear systems.

Currently, CCATT-BRAMS [5] is the largest high-performance application in Brazil with over 450 thousands lines of Fortran code. BRAMS is an atmospheric model based on RAMS, a Brazilian modification to include tropics/subtropics. CCATT is the chemical transport model designed for local and regional studies of the atmospheric chemistry coupling a chemical model in an online fashion with BRAMS [5].

These online chemistry models must combine implementation aspects related to spatial resolution, simulation length, and degree of complexity of the chemical mechanism. As such, fine-grained resolution, large domains, and detailed chemistry are infeasible in long production runs due to its

associated computing costs. In BRAMS, the chemical reaction model is a stiff problem solved by the Rosenbrock method as depicted by the pseudocode in Figure 1. This code reflects an implementation with 65 columns, corresponding to 65 3D columns in the partition of the physical space and are used for concurrency management purposes.

```
Algorithm: Rosenbrock Method
   Input: Sparse1.3 data structure
1  begin
2      foreach block do
3          foreach grad_point do
4              Read variables from BRAMS;
5              Update photolysis rate;
6          Compute initial kinetic reactions;
7          while Timestep < threshold do
8              Compute Jacobian of the matrix of concentrations;
9              Compute Equation (2.2);
10             foreach chemical_specie do
11                 foreach grad_point do
12                     Update F(ρ) on the data structure;
13             while error > tolerance do
14                 foreach chemical_specie do
15                     foreach grad_point do
16                         Update matrix A;
17                 Update b_i;
18                 foreach grad_point do
19                     Compute 1st Rosenbrock method;        } 65x
20                 Update b_i;
21                 foreach grad_point do
22                     Compute 2nd Rosenbrock method;        } 65x
23                 Update matrix of concentrations ρ;
24                 Update production term F(ρ);
25                 Update b_i;
26                 foreach grad_point do
27                     Compute 3rd Rosenbrock method;        } 65x
28                 Update matrix of concentrations ρ;
29                 Update production term F(ρ);
30                 Update b_i;
31                 foreach grad_point do
32                     Compute 4th Rosenbrock method;        } 65x
33                 Update matrix of concentrations ρ;
34                 Compute error and rounding;
35                 if tolerance - rounding > 1.0 then
36                     Accept solution;
37                 else
38                     Compute the new integration step;
39                 Update the integration step;
40         Update variables from BRAMS;
```

Figure 1. Pseudocode of the Rosenbrock method block for 65 columns. Equation (2.2) stands for the linear system solver.

Previous results showed that the parallel nature from Jacobi is not enough to beat the sequential algorithm in CPU. That is because of the efficient data structure associated with the decomposition reuse across the Rosenbrock stages. A small change of the matrix does not require the entire decomposition either; that is, they provide an update function.

3. Methods for Solving Linear Systems

In this section, we present Jacobi and the QR methods we implemented. We also describe which optimizations we applied to each of them.

3.1. Jacobi Single-Threaded Sparse

OpenCL standard defines two different programming models [6]: data-parallel and task-parallel programming model. Programmers must know both models, and consider which one is more suitable for the underlying hardware. Data parallelism is suitable for single instruction, multiple data (SIMD); this kind of parallelism is the basis for GPU acceleration. OpenCL API defines this programming model through NDRange [6–9]. NDRange programming imposes some restrictions to the programmers, different memory space for each group is one of them. Consequently, we are not able to compute the vector norm since we cannot share memory. Considering this problem, we chose the task parallelism, which works as a C to Hardware in OpenCL for FPGAs, where each loop represents a pipeline.

In Figure 2, we show our block diagram for Jacobi single sparse. Regarding loop pipelining, the single linear solve kernel executes multiple rows in flight. Not all loops can be executed in a pipelined fashion, as is the case with the outermost loop (A), for controlling the algorithm's iterations.

We use the compressed sparse row (CSR) format, a sparse representation suitable for sparse matrix-vector multiplication SpMV [10]. However, that format adds a complex loop exit condition in the pipeline (B) as each row has a different number of non-zeros (NNZ); Note the red border square (D), it means that it is necessarily two loads.

3.2. The QR Factorization Method

In this version, we implemented the QR factorization method based on Gram-Schmidt; we list it in Algorithm 1. We have implemented three main optimizations in this algorithm when porting it to hardware. The first version (QR) is a straightforward translation from C to OpenCL, without adding specific code for parallelism.

In the first optimization (QR + SR), we used shift registers to improve pipeline parallelism and remove data dependency on the multiply-accumulate inherited from the dot product operation. In a second optimization (QR + SR + LM), we removed the use of global memory during QR decomposition, that is, computations are performed with data residing in the local memory.

In a third version, we compiled the same algorithm with the *-fp-relaxed* flag, which uses a balanced tree of floating operations by relaxing the order of the operations. Although useful for highly parallel hardware, it may incur in numerical errors. We could not generate this hardware design, as it did not fit on the FPGA.

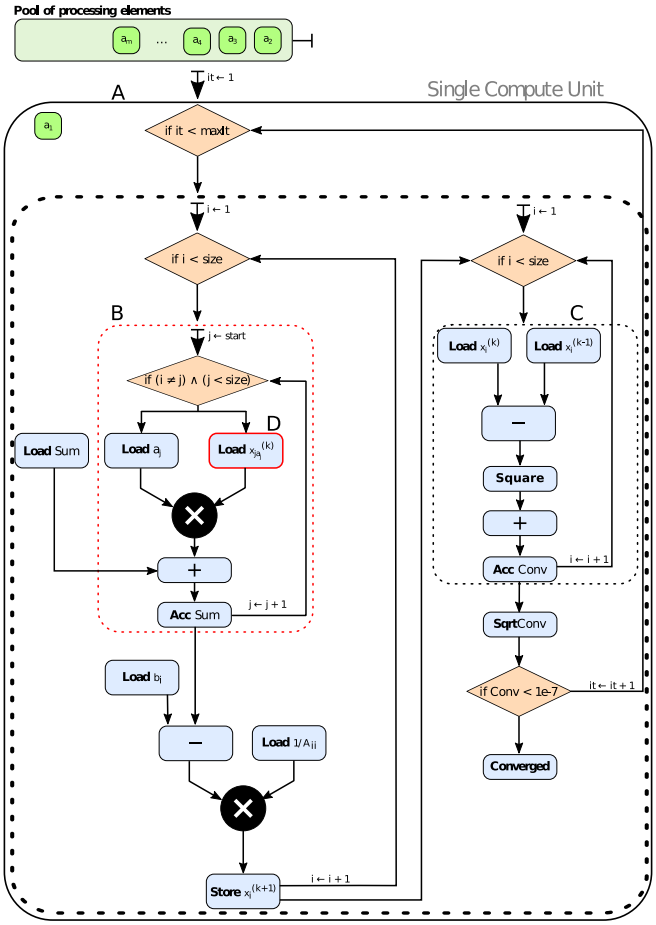

Figure 2. Block diagram of Jacobi method with single thread.

Algorithm 1: QR method without reordering (herein identified as QR).

Data: Initial matrix A
Result: Matrix decomposed in matrix Q and matrix R
$Q \leftarrow A$;
for $k \leftarrow 1$ **to** n **do**
  $R(k,k) \leftarrow norm(Q(1:m,k))$;
  for $m \leftarrow 1$ **to** n **do**
    $Q(m,k) \leftarrow Q(m,k)/R(k,k)$;
  end
  for $j \leftarrow k+1$ **to** n **do**
    $R(k,j) \leftarrow dot(Q(1:m,k), Q(1:m,j))/R(k,k)$;
    for $m \leftarrow 1$ **to** n **do**
      $Q(m,j) \leftarrow Q(m,j) - R(k,j)) * Q(m,k)$;
    end
  end
end

3.3. QR-Base Implementation

This implementation is a mapping to OpenCL and HARP of the main algorithm steps used in the original Intel's manual hardware implementation and presented in Algorithm 2. We opted by Intel's deeply pipelined version, using in our case double-precision floating-point data types (needed for achieving the precision required by CCATT-BRAMS). We present three versions of this algorithm.

In our first version (QR-base + SR + LM), we applied all the optimizations previously considered for the first QR algorithm, that is, we used shift registers and local memory without the specific compiler options.

Algorithm 2: QR pipeline suitable implementation by Reference [2] (herein identified as QR-base).

```
Data: Initial matrix A
Result: Matrix decomposed in matrix Q and matrix R
Q ← A;
for k ← 1 to n do
    R2(k) ← dot(Q(1 : m, k));
    for j ← k + 1 to n do
        Rn(k, j) ← dot(Q(1 : m, k), Q(1 : m, j));
    end
    for j ← k + 1 to n do
        for m ← 1 to n do
            Q(m, j) ← Q(m, j) − ((Rn(k, j)/R2(k)) ∗ Q(m, k));
        end
    end
end
for k ← 1 to n do
    R(k, k) ← sqrt(R2(k));
    for j ← k + 1 to n do
        R(k, j) ← Rn(k, j)/R(k, k);
        for m ← 1 to n do
            Q(m, k) ← Q(m, k)/R(k, k);
        end
    end
end
```

In a second version (QR-base + SR + FPR), we applied the *-fp-relaxed* compiler option. This version could fit on the FPGA, and we achieved improvements over the first version. The third version is a variant of the second. As we use an Arria 10 device, we can remove the shift register and fully unroll the dot product loop to make use of the floating-point DSPs. The compiler instantiates a chain of floating-point DSPs, which allows us to perform the dot product with the same cost as a single multiplication.

Unrolling the dot product significantly increases hardware utilization due to memory replication. To avoid such a problem, we buffered one column of the matrix used for the dot product in another private memory array. In this manner, the compiler creates a copy of a single column of the matrix.

In the third version (QR-base + LM + FPR + FPC), we used the *-fpc* flag that is responsible for avoiding rounding operations. Both compiler flags incur in numerical errors, that is why we performed accuracy tests. In Figure 3, we present the block diagram for the third version of this algorithm.

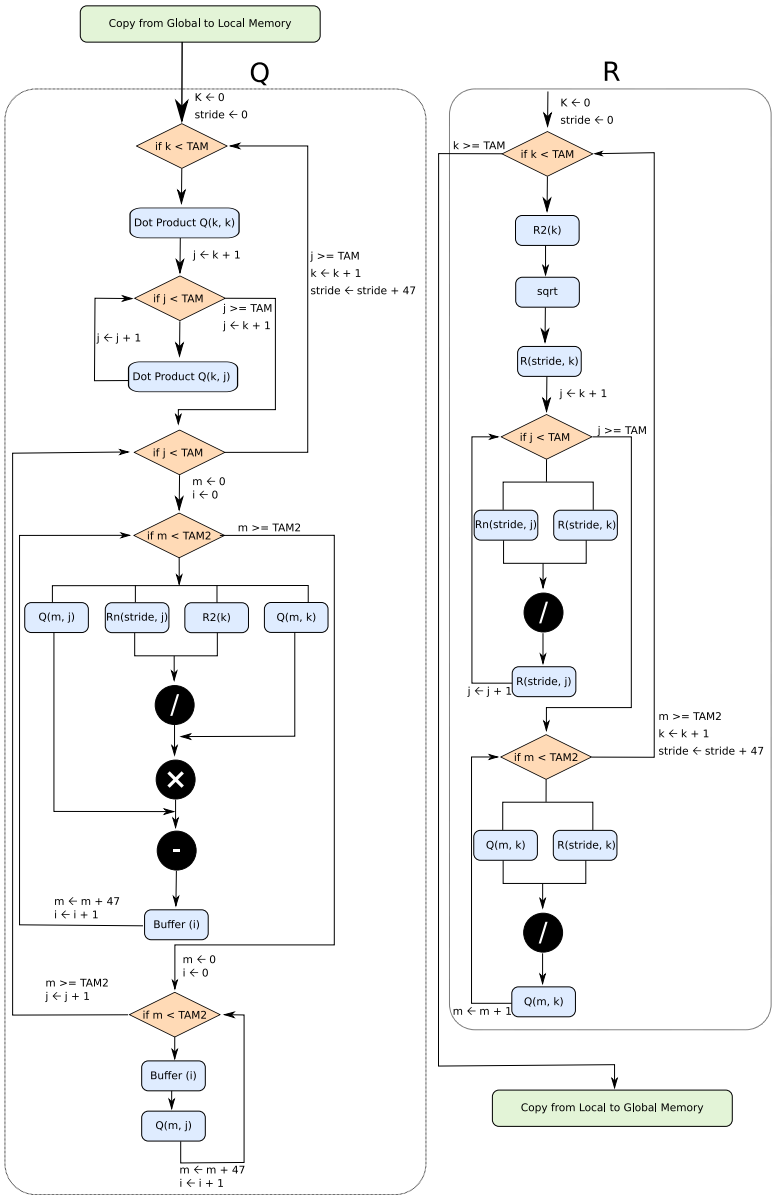

Figure 3. Block diagram of the third version of QR-base.

4. Implementation Results

In this section, we present experimental results regarding the implementation in hardware of the computational structures that support iterative and direct linear solver methods. For the QR methods, we also implemented their software counterpart. We summarize our results in Figure 4 for execution time (using log base 2 scale), and Figure 5 for FPGA resources usage normalizing the results by showing the percentage of usage of each type of resource on the FPGA, except for the maximum clocking frequency.

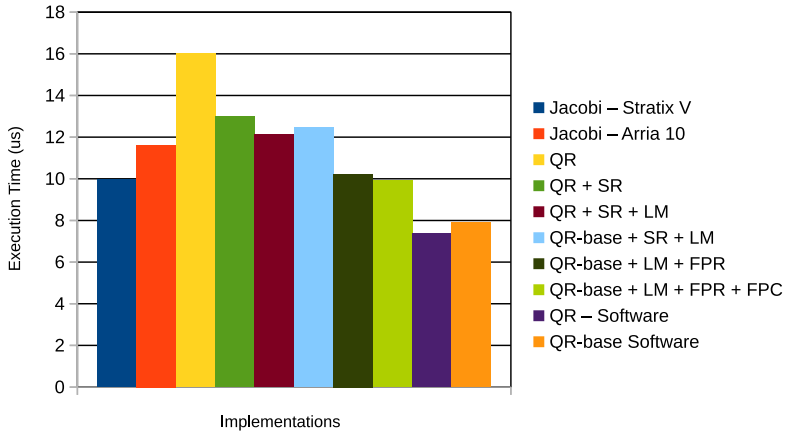

Figure 4. Execution time for each design implementation.

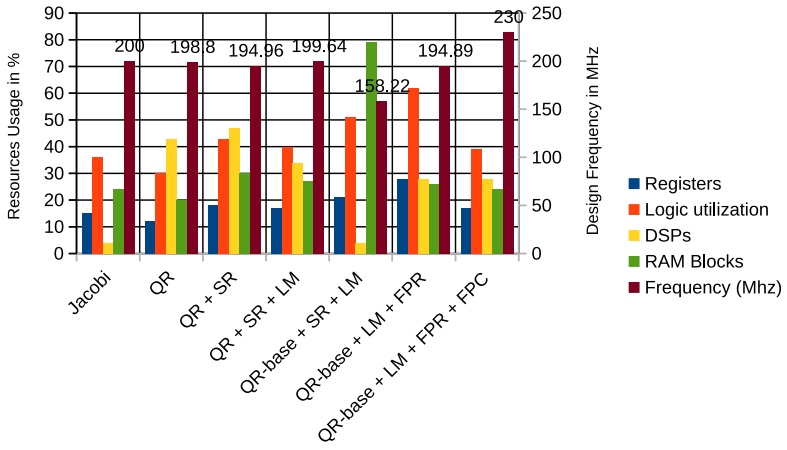

Figure 5. Arria 10 field-programmable gate array (FPGA) resources usage and maximum clock frequency of each design implementation.

4.1. Platforms

In this work, we considered two target hardware platforms. First, a hybrid CPU + FPGA board including a Xeon E5-1607 (3.1 GHz) (Intel, Santa Clara, CA, USA), 32 GB main memory, and running CentOS 6.7 coupled via PCIe to a Stratix V (Bittware S5HPQ_A7) FPGA. This first platform was used to derive previous results and for profiling the CCATT-BRAMS application used in this research.

A second platform consisted of the Intel's HARP hybrid computing system including a Xeon (E5-2600 v4 2.0 GHz) processor, 68 GB of main memory running CentOS 7, and coupled to an Arria 10 GX FPGA (10AX115N2F40E2LG).

We used this second platform to derive the experimental results presented in this article for both the modified Jacobi method and the QR decomposition. In any case, our HARP experiments use data from the software version of the CCATT-BRAMS application.

4.2. Jacobi Method

In this section, we present an implementation of the same algorithm of our previous research [11], but now targeting the HARP system. We present the results in Tables 1 and 2, Table 1 includes the cost of communication between the CPU and the FPGA hardware design.

Table 1. Results for the Jacobi method.

	Stratix V	**Arria 10**
Send (µs)	92	29
Execution (µs)	912	3111
Receive (µs)	9	21
Total (µs)	1013	3161

In Table 2, we show the cost of the FPGA implementation in terms of 1-bit registers (Registers), the overall amount of logic resources including the number of DSP blocks, and Block RAMs, along with the percentage usage of each resource.

Table 2. Hardware resources for Jacobi method.

	Stratix V	**Arria 10**
Registers	153,046 (17%)	254,753 (15%)
Logic	80,837 (34%)	155,531 (36%)
DSPs	71 (28%)	64 (4%)
RAM blocks	686 (27%)	662 (24%)
Maximum Clock Frequency (MHz)	269.39	200

We observe that the Jacobi hardware design consumes more space on an Arria 10 than on a Stratix V device, which in turn leads to a lower operating clock frequency. Regarding the DSPs and block RAMs, both FPGA designs exhibit nearly identical figures. Although the Arria-10 implementation could overcome the loop-carried dependency of the accumulator in the dot product. This improvement is not enough to beat the performance of Stratix V implementation.

On the Stratix V, we did not have enough parallelism to exploit more registers. By unrolling the dot product loop in Arria-10, we expose more concurrency, which demands more data per clock cycle. Using more resources dropped the performance of our circuit.

In contrast to the computing performance, the communication improved by 50% in the HARP environment. This improvement, resulting from the on-die communication between the CPU and the FPGA hardware, is significant.

4.3. QR Method

A first implementation of the QR decomposition method, labeled as QR, corresponds to a straightforward translation from C to OpenCL experimenting with transformations regarding the efficient implementation of the square root operations. In a second version (QR + SR), we used shift registers to remove data dependencies that caused pipeline stalls. That improves the algorithm execution time by a factor of 8 over the first version. The third version (QR + SR + LM) performs the entire computation over local memory instead of global memory. That required two copies, the first to store the content of global memory into local memory, and the second to write the results back.

We also tried to unroll the loop that copies data from the global to the local memory. Full unroll is, however, not advantageous as it cannot fetch all the necessary data in the same clock cycle. Partial unrolling did not improve the results either.

Even though we declared auxiliary matrices for Q and R, the resulting hardware uses fewer RAM blocks. We attribute this to the compiler's ability to infer local memory based on the algorithm's behavior. In this version, we also avoided complex index computation in order to save DSPs.

Using local memory turned version QR + SR + LM 44% faster compared to the second version. We could improve the QR method in 15× at the cost of 10% more area resources.

QR Method in Software

Although we achieved a significant improvement over the first version (QR), we still present a hardware design 4× slower than the sequential software. Such results worsened when vectorization

was applied as enabled by ICC ("-O3" flag) to our software. Our hardware design becomes 22× slower than the vectorized software.

Decomposing the matrix into Q and R is the most expensive part of the $Ax = b$ linear system. However, we still have to perform the backward substitution to get the results. That substitution is inherently serial, which means that an FPGA cannot improve the running time.

In that manner, we implemented the backward substitution on C in our software side of the codesign. We use QR-Software label to represent this software version.

4.4. QR Intel Baseline

Our first version of Algorithm 2 includes all the optimizations from the previous algorithm, excluding compiler optimizations. We use the label QR-base + SR + LM to represent this version.

This QR Intel Baseline version (or just QR-base) did not improve the running time. We can justify it with two main factors: lower clock frequency, and almost no increase in DSP usage. The new arrangement of the algorithm incurred in more area resources, especially memory, that caused the frequency drop. We also observed to be better to use floating-point DSPs rather than of shift registers.

In version QR-base + LM + FPR, we use the *-fp-relaxed* compilation flag. This improves the execution time by a factor of 5× and uses 53% fewer RAM blocks. That was possible because we replaced the shift registers by floating-point DSPs, and we reused the dot product critical datapath, leading to an increase in clock frequency. While, *-fp-relaxed* incurs in numerical errors, we compared the absolute error between the hardware and software solutions and found this absolute error to be close to zero (0.002), prompting us to explore additional floating-point optimizations.

In version QR-base + LM + FPR + FPC, we use the *-fpc* compilation flag. That improved performance in 18%, and decreased hardware resources in 23%. The numeric results exhibit the same absolute error as the QR-base + LM + FPR version.

This third version is almost 5× better than the best QR implementation (label QR + SR + LM). The results also reveal the availability of resources for design exploration. As such, we experimented with full and partial unroll over other loops. Most of those experiments led to hardware designs that are unfeasible as they exceed the number of available FPGA resources. On the other hand, the hardware designs that fitted on the FPGA did not improve the overall computation performance.

This third hardware design version can process up to 102 matrices per second. However, the HDL-based design proposed by Intel can process up to 50,853 matrices per second, an improvement of 498×. We consider that this difference is mostly due to the way memory is used in both solutions, the use of single-precision instead of double-precision, and the communication overhead.

In our implementations, data are stored in global memory. Thus, the initial matrix is copied from global memory to the local memory to improve performance. When the computation finishes, the results are copied back to the global memory so the CPU can access the results generated by the FPGA.

Intel's design relies exclusively on local memory, thus avoiding the latency of memory copies and hence communication overhead. We consider this infeasible for our purposes as the FPGA used has around 6 MB of block RAMS, which would fit at most thirteen 47 × 47 matrices at a time, according to our tests.

Regarding hardware logic, it uses around 4× more resources, which we attribute to the interface with the processor and the memory hierarchy that supports DDR4 RAM, absent in the baseline. We must also consider that we are using double precision, which incurs in additional latency due to the fetching of double data types from global memory.

QR-Base in Software

For the same reason as before, we did not implement backward substitution in hardware. In this section, we decided to compare only the vectorized software with our best hardware design. According to our results, we still have a slowdown of 4× when compared with its software counterpart.

However, we are comparing an FPGA design running at 230 MHz (our best hardware design) with a Xeon processor that can operate at clock frequencies between 2.3 GHz and 3.6 GHz. We consider that the current FPGA implementation is interesting enough to be considered as part of a heterogeneous solution where the matrices to be computed are distributed among CPUs and FPGAs.

We also noticed that the C implementation of QR-base (presented in Algorithm 2) does not perform as well in CPU. It is 38% slower than the best software implementation of QR (presented in Algorithm 1). This slower performance is related to the number of loops in the QR-base algorithm—The version represented as QR-base-Software .

4.5. Final Remarks

In this section, we summarize our results. As seen in Figure 4, the hardware designs are still much slower than the software version. We believe that high-level synthesis plays a critical role in those poor results; converting software to hardware is still challenging.

At first, Jacobi-Stratix V seems to be one of the fastest versions among the hardware designs (https://github.com/caosjr/qrmethod_opencl_intel_fpga). However, since Jacobi is an iterative method, there is no advantage of its application in settings where this method needs to be used multiple times over the same matrix A. Instead, methods that carry out decomposition have a clear advantage as the decomposition of the A matrix can be reused.

In the case of the Rosenbrock method, we need to assess the performance of executing the Jacobi method 4 times versus a single QR decomposition.

Also, the Jacobi algorithm uses a sparse-matrix format, since CCATT-BRAMS matrices have around 10% of non-zero values, while our QR implementation is using dense matrices. We did not improve Intel's results, but we realized that their algorithm reordering could also improve high-level synthesis results.

We also performed an experiment where we measured the execution time between the global and local memory copies. Both these copy operations require 204 µs to complete, which is around 21% of the execution time of the best QR-base solution (QR-base + LM + FPR + FPC).

4.6. Discussion

While the data communication rate is improved in the HARP system, the results reveal that updating from Stratix V to Arria 10 in support of implementations of the Jacobi Method does not yield performance gains. The performance degrades as the resulting solutions required more resources, thus lowering the design clock frequency.

The experiments with the QR factorization reveal that using OpenCL HLS can generate a design that exhibits a performance that is several orders of magnitude lower than a manual design. We attribute this to the more efficient local memory management of a manual solution, which only uses block RAM. Our design considers a complete system with communication between CPU and FPGA, whereas the manual version requires all data to be present in FPGA block RAM.

Still, and using several transformations, our best QR solution is 4× faster than the best QR design without code transformations. Using Intel's reordering of the algorithm was critical for improving performance, and area usage. That also allowed us to explore floating-point optimization on the compiler side. We could not explore such floating-point optimizations with the QR implementation; they did not fit on the FPGA.

5. Related Work

Several researchers explored ways to boost linear system solver performance using concurrency and algorithmic transformations. In this description, we focus exclusively on efforts that leverage FPGA-based accelerators either stand-alone or by the adoption of hybrid solutions where a CPU and its coupled FPGA cooperatively work towards the computation.

In our previous research [11], we developed a hardware design for the implementation of the iterative Jacobi method using a Stratix-V FPGA using double-precision arithmetic. Since this FPGA does not have native floating-point

DSP blocks, we had to use part of the FPGA area to implement them. Kapre and DeHon [12] provide a Parallel Sparse Matrix Solver for SPICE (Simulation Program with Integrated Circuit Emphasis), where they replace the existing library, Sparse 1.3a, with KLU as Sparse 1.3a was not suitable for FPGA parallelism due to the frequent change of matrices non-zero pattern. Both of these two libraries are LU-based and use double-precision.

Their solution achieves a range from 300 MFlop/s to 1300 MFlop/s on a Xilinx Virtex-5, while the processor (Intel Core i7 965) achieved 6 MFlop/s to 500 MFlops/s. However, this work does not explore a hybrid solution as the sparse matrix is stored inside the DDR2 memory attached to the FPGA board.

In another work, Daga et al. [13] implement an LU decomposition in double-precision in FPGA. They consider only non-singular matrices to avoid costly pivoting. They compare their results with a general-purpose processor, and their speedups range from 19 to 23. The entire solution is decoupled from the CPU, including the initial data.

In a similar FPGA-only design approach, Zhuo and Prasanna [14], also report substantial performance improvements over their own earlier versions making extensive use of block RAMs.

In a genuinely hybrid design solution, Wu et al. [15] propose a solution for sparse matrices, where the preprocessing is carried out in the CPU, and the factorization in the FPGA. Their simulated performance results rely on counting the cycles of factorization and the clock frequency of the processing element (PE) but exclude the data communication overhead.

Ruan et al. [16] present a similar approach to our previous research with Jacobi. They use a Java high-level synthesis (the MaxJ compiler) to implement the Jacobi method targeting a hybrid architecture including a Virtex-6 FPGA and a CPU. They also explored software parallelism with MPI and multi-threading. Their hybrid solution is the fastest among their results, although it does not support matrices bigger than 200×200.

Regarding the implementation of QR decomposition, Parker et al. [2] describe a pipelined implementation of the QR decomposition in single-precision targeting an Arria 10 FPGA and making heavy use of DSP Builder Tools. Their solution, with a peak performance rate of 78 Gflops, is not hybrid, as all the input data resides in FPGA block RAMs.

Recent work by Langhammer and Pasca [17] describes an implementation of a QR decomposition based on the modified Gram-Schmidt in a core generator in C++, with the math operations implemented by the DSP Builder Tools. They present some modifications to the pipeline and the square-root operation using the reciprocal square-root, the same operation we exploited in this article. Still, this work uses single-precision, which requires fewer cycles for divisor and reciprocal square-root operations. According to their results, they achieved three times the performance of Parker et al. [2].

6. Conclusions

In this paper, we explored FPGA implementations for solving the linear systems that arise from the Rosenbrock Stiff ODE solver. Our experiments target the Intel's HARP architecture with the OpenCL programming environment and are focused on the QR decomposition stages. Each proposed version uses a different approach to improve the hardware pipeline implementation on the FPGA. Our best QR FPGA design implementation is $4\times$ faster than the unmodified original QR design. The Intel's algorithm reordering, previously proposed, was critical for improving performance and area usage, also enabling the exploration of floating-point optimizations by the OpenCL compiler. Ongoing work is addressing further algorithm reordering and optimizations regarding data distribution/replication. Future plans include the use of sparse formats and other variants of the QR decomposition, in particular Householder, and Givens Rotation. Lastly, we also intend to consider a solution where both CPU and FPGA execute concurrently.

Author Contributions: Conceptualization, C.A.O.d.S.J., and J.B.; Methodology, C.A.O.d.S.J., and J.B.; Software, C.A.O.d.S.J.; validation, C.A.O.d.S.J.; Writing—review & editing, C.A.O.d.S.J., J.B., J.M.P.C., E.M., and P.C.D. All authors have read and agreed to the published version of the manuscript.

Acknowledgments: The authors would like to thank CAPES, and process nº 2017/14268-6 and 2019/08153-7, Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP) for supporting our research. João Bispo acknowledges the support provided by Fundação para a Ciência e a Tecnologia, Portugal, under Post-Doctoral grant SFRH/BPD/118211/2016. Research (partially) carried out using the computational resources of the Center for Mathematical Sciences Applied to Industry (CeMEAI) funded by FAPESP (grant 2013/07375-0). The presented results were (partially) obtained on resources hosted at the Paderborn Center for Parallel Computing (PC^2) in the Intel Hardware Accelerator Research Program (HARP2).

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kreyszig, E. *Advanced Engineering Mathematics*; John Wiley & Sons: Hoboken, NJ, USA, 2010.
2. Parker, M.; Mauer, V.; Pritsker, D. QR decomposition using FPGAs. In Proceedings of the 2016 IEEE National Aerospace and Electronics Conference (NAECON) and Ohio Innovation Summit (OIS), Dayton, OH, USA, 25–29 July 2016; pp. 416–421.
3. Moreira, D.; Freitas, S.; Bonatti, J.; Mercado, L.; Rosário, N.; Longo, K.; Miller, J.; Gloor, M.; Gatti, L. Coupling between the JULES land-surface scheme and the CCATT-BRAMS atmospheric chemistry model (JULES-CCATT-BRAMS1.0): Applications to numerical weather forecasting and the CO_2 budget in South America. *Geosci. Model Dev.* **2013**, *6*, 1243–1259. [CrossRef]
4. Graham, S.L.; Kessler, P.B.; Mckusick, M.K. Gprof: A call graph execution profiler. *ACM Sigplan Not.* **1982**, *17*, 120–126. [CrossRef]
5. Longo, K.M.; Freitas, S.R.D.; Pirre, M.; Marecal, V.; Rodrigues, L.F.; Panetta, J.; Alonso, M.F.; Rosario, N.E.; Moreira, D.S.; Gacita, M.S.; et al. The Chemistry CATT-BRAMS model (CCATT-BRAMS 4.5): A regional atmospheric model system for integrated air quality and weather forecasting and research. *Geosci. Model Dev.* **2013**. [CrossRef]
6. Khronos OpenCL Working Group. *The OpenCL Specification*; In Proceedings of the IEEE Hot Chips 21 Symposium (HCS), Stanford, CA, USA, 23–25 August 2009.
7. Munshi, A.; Gaster, B.; Mattson, T.G.; Ginsburg, D. *OpenCL Programming Guide*; Pearson Education: London, UK, 2011.
8. Buchty, R.; Heuveline, V.; Karl, W.; Weiss, J.P. A survey on hardware-aware and heterogeneous computing on multicore processors and accelerators. *Concurr. Comput. Pract. Exp.* **2012**, *24*, 663–675. [CrossRef]
9. Tsuchiyama, R.; Nakamura, T.; Iizuka, T.; Asahara, A.; Son, J.; Miki, S. *The OpenCL Programming Book*; Fixstars: Tokyo, Japan, 2012.
10. Bell, N.; Garland, M. *Efficient Sparse Matrix-Vector Multiplication on CUDA*; Technical Report, Nvidia Technical Report NVR-2008-004; Nvidia Corporation: Santa Clara, CA, USA, 2008.
11. De Souza, C.A.O.; Pereira, E.D.S.; Marques, E. A Hardware/Software Codesign for the Chemical Reactivity of BRAMS. In Proceedings of the 2017 Euromicro Conference on Digital System Design (DSD), Vienna, Austria, 30 August–1 September 2017; pp. 70–77.
12. Kapre, N.; DeHon, A. Parallelizing sparse matrix solve for SPICE circuit simulation using FPGAs. Field-Programmable Technology, 2009. In Proceedings of the 2009 International Conference on Field-Programmable Technology, Sydney, Australia, 9–11 December 2009; pp. 190–198.
13. Daga, V.; Govindu, G.; Prasanna, V.; Gangadharapalli, S.; Sridhar, V. Efficient floating-point based block lu decomposition on fpgas. In Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms, Las Vegas, NV, USA, 21–24 June 2004; pp. 21–24.
14. Zhuo, L.; Prasanna, V.K. High-performance and parameterized matrix factorization on FPGAs. In Proceedings of the 2006 International Conference on Field Programmable Logic and Applications, Madrid, Spain, 28–30 August 2006; pp. 1–6.
15. Wu, W.; Shan, Y.; Chen, X.; Wang, Y.; Yang, H. FPGA accelerated parallel sparse matrix factorization for circuit simulations. In *International Symposium on Applied Reconfigurable Computing*; Springer: Berlin, Germany, 2011; pp. 302–315.

16. Ruan, H.; Huang, X.; Fu, H.; Yang, G. Jacobi Solver: A Fast FPGA-based Engine System for Jacobi Method. *Res. J. Appl. Sci. Eng. Technol.* **2013**, *6*, 4459–4463. [CrossRef]
17. Langhammer, M.; Pasca, B. High-performance qr decomposition for fpgas. In Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, CA, USA, 25–27 February 2018; pp. 183–188.