Dataflow Implementation of QR Decomposition on a Manycore

Süleyman Savas, Sebastian Raase, Essayas Gebrewahid,
Zain Ul-Abdin and Tomas Nordström
Centre for Research on Embedded Systems, Halmstad University, Halmstad, Sweden
Suleyman.Savas@hh.se, Sebastian.Raase@hh.se, Essayas.Gebrewahid@hh.se,
Zain-Ul-Abdin@hh.se, Tomas.Nordstrom@hh.se

ABSTRACT

While parallel computer architectures have become mainstream, application development on them is still challenging. There is a need for new tools, languages and programming models. Additionally, there is a lack of knowledge about the performance of parallel approaches of basic but important operations, such as the QR decomposition of a matrix, on current commercial manycore architectures.

This paper evaluates a high level dataflow language (CAL), a source-to-source compiler (Cal2Many) and three QR decomposition algorithms (Givens Rotations, Householder and Gram-Schmidt). The algorithms are implemented both in CAL and hand-optimized C languages, executed on Adapteva's Epiphany manycore architecture and evaluated with respect to performance, scalability and development effort.

The performance of the CAL (generated C) implementations gets as good as 2% slower than the hand-written versions. They require an average of 25% fewer lines of source code without significantly increasing the binary size. Development effort is reduced and debugging is significantly simplified. The implementations executed on Epiphany cores outperform the GNU scientific library on the host ARM processor of the Parallella board by up to 30x.

1. INTRODUCTION

Computer architectures are moving towards manycores for reasons such as performance and energy efficiency. The required parallelism to use these architectures efficiently requires new software tools, languages and programming models to abstract the differences between different architectures away. This reduces required knowledge about the architectures and their specific programming language extensions. Even with tools, writing efficient parallel applications is challenging and there is a lack of knowledge on performance of common applications such as QR decomposition when executed on manycores.

QR decomposition (QRD) [8] is one of the major factorizations in linear algebra. It is well known to be numerically stable and has many useful applications such as replacing matrix inversions to avoid precision loss and reduce the number of operations, being a part of the solution to the linear least squares problem and being the basis of an eigenvalue algorithm (the QR algorithm).

In this paper, we evaluate Cal2Many [6] source-to-source compiler, which translates CAL [5] code to native code for multiple manycore architectures. As a case study, we implemented three QRD algorithms (Givens Rotations, Householder and Gram-Schmidt) both in CAL and in native C for Adapteva's Epiphany [11] architecture. All implementations use our own communications library [13]. We used the Parallella platform to evaluate our implementations in terms of performance, development effort and scalability.

MES '16, June 19 2016, Seoul, Republic of Korea
© 2016 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ISBN 978-1-4503-4262-9/16/06...$15.00
DOI: http://dx.doi.org/10.1145/2934495.2934499

2. BACKGROUND

2.1 CAL Actor Language

The CAL actor language is a dataflow language consisting of actors and channels. Actors are stateful operators which execute code blocks (actions), take inputs and produce outputs usually with changing the state of the actor. The channels are used to connect the actors to each other. Therefore, interaction among actors happens only via input and output ports. CAL actors take a step by 'firing' actions that satisfy all the required conditions. These conditions depend on the value and the number of input tokens, and on the actor's internal state. The actors are instantiated and connected to each other via Network Language (NL) included in CAL.

2.2 Cal2Many

The Cal2Many compilation framework contains two intermediate representations (IRs): Actor Machines (AM) [10] and Action Execution IR (AEIR) [6]. Each actor is first translated to an AM, which describes how to schedule execution of actions. To execute AM, its constructs have to be transformed to a different programming language constructs, which have different implementations in different programming languages on different platforms. To stay language -agnostic and get closer to a sequential action scheduler, *AEIR* is introduced.Epiphany backend generates C code using our custom communications library and generates channels and mapping of actor instances by using the NL.

2.3 QR Decomposition

QR decomposition is decomposition of a matrix into an upper triangular matrix R and an orthogonal matrix Q. The equation of a QRD for a square matrix A is simply $A = QR$. The matrix A does not necessarily need to be square. The equation for an $m\ x\ n$ matrix, where $m \geq n$, is as follows:

$$A = QR = Q \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = \begin{bmatrix} Q_1 & Q_2 \end{bmatrix} \begin{bmatrix} R_1 \\ 0 \end{bmatrix} = Q_1 R_1.$$

We have implemented three QRD algorithms (Givens Rotations, Householder and Gram-Schmidt) in both CAL and native C for the Epiphany architecture.

2.4 Epiphany Architecture

Adapteva's manycore architecture is a two-dimensional array of cores connected by a mesh network-on-chip [11]. It operates in a shared, flat 32-bit address space. The network-on-chip is made of three meshes called *rMesh*, *cMesh* and *xMesh*. The former is used exclusively for read requests, while the latter two carry write transactions destined for on-chip and off-chip, respectively. The mesh uses a static XY routing algorithm.

Each core contains a single-precision floating-point RISC CPU, 32KB of local memory, a two-channel DMA engine and a mesh interface. Two event timers allow cycle accurate measurements of different events.

We have used the Parallella-16 platform, which contains a 16-core Epiphany-III running at 600 MHz in addition to a dual-core ARM Cortex-A9 host processor running at 667 MHz.

2.5 Communication Library

We have implemented a custom communications library for the Epiphany architecture [13], which is used in all implementations. It is centered around token-based, unidirectional communication channels. These channels support blocking *read*, *write* and non-blocking *peek* operations and allow checking the current number of immediately read- or writable tokens for block-free operations.

A global table describes all channels in the system. Local data structures and buffers are allocated at run-time by each core. Channels are implemented as ring buffers and it is possible to use the event timers to gauge blocking overhead.

3. RELATED WORKS

There are many approaches for QRD, however, they focus either on architectures or scheduling and use only one algorithm. We have executed three algorithms in two programming languages and compared them to each other.

Buttari et al. [3] present a QRD method where they run a sequence of small tasks on square blocks of data and use a dynamic scheduling mechanism to assign tasks to computational units. We execute small tasks in a dataflow fashion.

Hadri et al. [9] present a QRD method for shared-memory multicore architectures and modified an existing algorithm to perform panel factorization in parallel. They aim tall or small square matrices whereas we aim square matrices.

Agullo et al. [2] implement a three step QRD on a multicore CPU which is enhanced with multiple GPU cores. They divide the decomposition into a sequence of tasks as we have done. Then they schedule these tasks on to individual computational units. Their approach suffers if the number of CPUs and GPUs are not the same.

While the CAL2C compiler [15] generates sequential C code, the Open-RVC CAL Compiler (ORCC) [12] and d2c [4] compilers generate multi-threaded C code, but require dedicated run-time system libraries. Our compilation framework generates separate C code for each actor instance to be executed on individual cores and does not require any run-time system support.

4. IMPLEMENTATIONS

All implementations use the same library to implement communication between the Epiphany cores, which is used by the Cal2Many as well. However, the communication characteristics differ naturally between algorithms and, to a lesser extent, between the hand-written (C) and the generated (CAL) implementations of the same algorithm.

4.1 Givens Rotations

The Givens Rotations (GR) algorithm applies a set of unitary rotation G matrices to the data matrix A. In each step, one of the sub-diagonal values of the matrix A is turned into zero, forming the R matrix. The multiplication of all rotation matrices forms the orthogonal Q matrix.

We implemented Givens Rotations (GR) with a modified Gentleman-Kung systolic array [7] [16] using 1, 5 and 12 cores individually. In parallel versions, two cores are used for distributing and collecting inputs and outputs, the rest of the cores are used for computations.

The implementations consist of 4 types of units named as cells. *Boundary* and *inner* cells perform the computation while *splitter* and *joiner* cells distribute and collect data. Figure 1 gives the layout of the systolic array mapped on the 4x4 core matrix of the Epiphany architecture. The inputs are read row by row. Each row is divided into four pieces and distributed to the first row of 4x4 Epiphany core matrix. Token size is defined as the size of each piece and the cores send and receive one token at a time for communication. Apart from the input elements, c, s and final r values are communicated.

Each cell calculates one r value. For 4×4 matrix, each cell is mapped onto an Epiphany core. However for 16×16 matrix, 16 *boundary* and 120 *inner* cells are required. Therefore the implementations are modified to combine a number of cells in one single core. Each core has to store the calculated r values which becomes a problem when large matrices are used. For 512×512 input matrix, an inner core consists of 128×128 *inner* cells which results in storing a 128×128 r matrix. Since r is a *float*, the required memory is 4 x 128 x 128 = 65536 bytes. Due to the local memory limitation, the largest matrix size that can be decomposed with Givens Rotations method is 256×256. The implementations can scale when the number of cores increases, i.e. with 64 cores the implementations can decompose a 1024×1024 matrix.

4.2 Householder Transformation

The Householder (HH) algorithm describes a reflection of a vector across a hyperplane containing the origin [18].

In our implementation, the Epiphany cores are connected as a one-dimensional chain of processing elements. Each core handles an equal amount of matrix columns and runs the same program. The communication is wave-like and next-neighbor only.

First, the input matrix is shifted into the processing chain, column by column, until it is fully distributed among the cores. Then, the last core in the chain computes a reflection vector w for each of its columns, updates them, and sends the vector towards the beginning of the chain, forming a

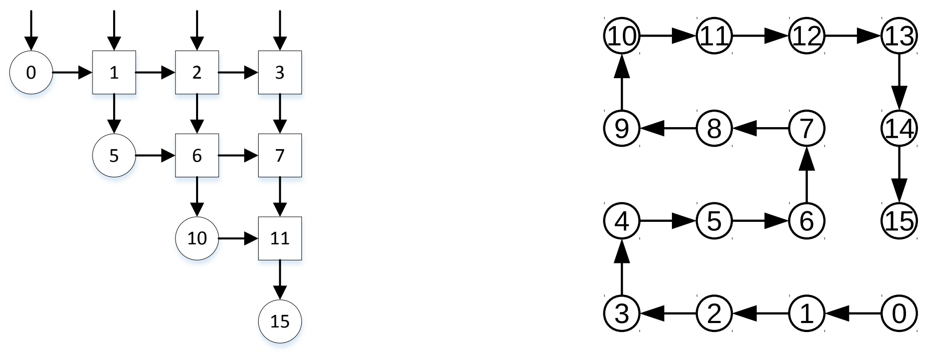

Figure 1: Mapping layout of all implementations. Givens rotations on the left, Householder and Gram-Schmidt on the right. On left, circles are boundary cells, squares are inner cells. On right, each core executes the same code. No shape relation across layouts.

communication wave each. All previous cores forward these vectors and update their own columns. After these waves have been forwarded by the penultimate core, it – after updating its own columns – computes its own reflection vectors and sends them along, forming new waves. When a core has sent its last reflection vector, it will form a final set of waves containing its result columns.

4.3 Gram-Schmidt

The Gram-Schmidt (GS) algorithm produces the upper-triangular matrix R row-by-row and the orthogonal matrix Q as a set of column vectors q from the columns of the data matrix A in a sequence of steps. In each step, we pick a column a of the matrix A. The dot product of this column with itself is calculated. Then, the square root of the result is taken to generate an element of matrix R. This element is later used to normalize the column a to produce a column of matrix Q. Then the column of matrix A is updated by subtracting a multiple of vector q with a value from matrix R to produce an orthogonalized vector that is then used to compute the next columns of matrix R.

Both CAL and C implementations work as a chain of processes that work on a certain number of columns, depending on the number of processes and the matrix size. All processes perform the steps required to compute a column of matrix R. In the first step, each process stores its local columns and pushes the remaining columns into the chain. In the second step, the processes read and forward the previous orthogonalized vectors after they use the vectors to produce the elements of matrix R and to update their local columns. In the third step, the processes compute and forward the local, orthogonalized vectors using their updated local columns. In the final step, the processes forward the elements of matrix R. The implementation can scale up to any number of processors num_p and any m x n matrix, where n is a multiple of num_p.

5. RESULTS & DISCUSSION

Our implementations are executed on the Parallella board and tested with different number of cores and different sized square matrices.

5.1 Performance Analysis

There are several aspects such as memory usage, input size and number of cores which affect the performances. Therefore we executed the implementations with different configurations and Table 1 presents the number of execution cycles while using external and internal memories to store input and output matrices. Additionally, last two columns give the number of cores used and the total code size in bytes.

As a reference point, we measured the execution time of a QRD implementation, that is a part of GNU Scientific Library [1], on the ARM processor of Parallella board. The library implementation uses Householder approach with double precision, whereas the Epiphany architecture supports only single precision. Decomposition of a 128×128 matrix on a single ARM core takes 90 milliseconds whereas on the Epiphany cores it takes 6.9, 4.1 and 2.9 milliseconds (shown in Table 1) for hand-written parallel GR, GS and HH implementations respectively. GR implementation is the only one that can decompose 256×256 matrices. While decomposing 128×128 matrices, it outperforms the library by 13x, however, with 256×256 matrices this number increases to 23x. When the matrix sizes increase, parallel implementations perform better due to increased computation/communication ratio as a result of communication patterns used in the implementations.

We implemented a message passing mechanism for inter-core communication as a library. If messages (or as we call them, 'tokens') are passed very frequently, the overhead of communication dominates the execution time due to library calls and copy operations. The communication patterns we use in our implementations keep the number of message passes constant regardless the matrix size. If the matrix sizes increase, instead of increasing the number of messages, the size is increased. By keeping the number constant, we avoid extra overhead of calling library functions. The communication cost still increases due to increased size of data that needs to be copied. However, it does not increase as fast as the cost of computation. Hence its effect on overall execution time decreases.

Figure 2 shows performance of each hand-written algorithm decomposing a 64×64 input matrix that is stored in the internal memory. We chose this matrix size because

	External mem	Local mem	SLoC	#cores	Footprint
GR Hand-written	9.51 M (15.8 ms)	4.16 M (6.9 ms)	647	12	71 k
GR CAL	10.45 M (17.4 ms)	6.11 M (10.1 ms)	400	12	72 k
HH Hand-written	10.45 M (17.4 ms)	1.76 M (2.9 ms)	219	16	225 k
HH CAL	10.69 M (17.8 ms)	2.00 M (3.3 ms)	170	16	223 k
GS Hand-written	11.17 M (18.6 ms)	2.47 M (4.1 ms)	188	16	179 k
GS CAL	11.40 M (19 ms)	2.70 M (4.5 ms)	160	16	193 k

Table 1: Execution times (in clock cycles and milliseconds), source lines of code, number of used cores and code size in bytes, 128×128 matrix. GR = Givens Rotations, HH = Householder, GS = Gram-Schmidt

it is the biggest size that is supported by all implementations with different core numbers (except for the single core implementation of Gram-Schmidt algorithm due to memory limitations). Gram-Schmidt implementation can decompose 64×64 matrix by using 2, 4, 8 and 16 cores and achieve 4x speed-up by going from 2 core to 16 cores. Householder implementation can decompose on 1, 2, 4, 8 and 16 cores and achieve 5.2x speed-up going from single core to 16 cores. Givens Rotations implementation decomposes the same matrix on 1, 3 and 10 computational cores due to it's structure and achieves 3.4x speed-up while going from single core to 10 cores. When decomposing small sized matrices such as 64×64, communication overhead plays a significant role in the execution time and decreases the speed-up. However, as the matrix size increases, the effect of communication decreases.

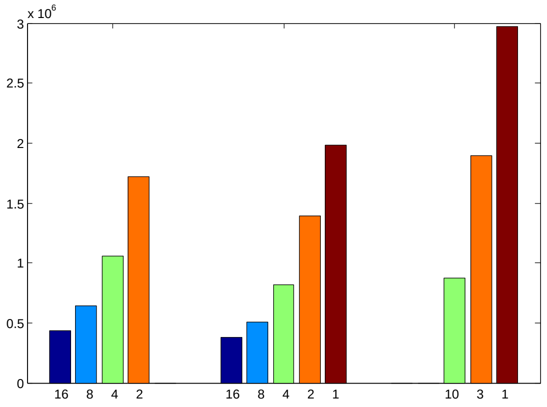

Figure 2: Execution cycles for GS, HH and GR respectively with different number of cores and 64x64 input matrix. X axis represent number of cores and Y axis represents number of clock cycles.

In our previous works [17, 14] we have experienced that the external memory access can be a bottleneck in the Epiphany architecture due to the slow link between the memory and the processing cores. Therefore we tested the implementations with and without using the external memory for storing the input matrix. We observed that when the input size increases, which means increased computation, the influence of external memory decreases however, it is still a bottleneck. Table 1 shows that using external memory to store the input matrix slows down the execution by 56% to 83% depending on the implementation. Givens Rotations seems to be the least influenced algorithm due to overlap between memory reads and computation.

An interesting point is the comparison of C and CAL implementations. Looking at Table 1, one can see that there is not much difference between hand-written and CAL implementations while using external memory. Even while using internal memory, the difference increases only for the GR implementation due to a slightly more complicated structure compared to the other implementations such as different communication pattern or having different number of channels depending on the position of the core. These small details increase the number of actions which are converted into functions in C. Overhead of repetitive calls to these functions increases the execution time. The main reasons of slow down for the generated code is having a scheduler and function calls. In the hand-written code, the actions are combined in the main function and there is no scheduler. Therefore there is neither function call overheads nor switch statements to arrange the order of these functions.

5.2 Productivity Analysis

In addition to performance, we compare the CAL and C implementations in terms of development effort, which is difficult to measure. It should be noted that our approaches to QRD have been implemented by three developers, who have more knowledge and experience with C rather than CAL. Nonetheless, the CAL implementations required about 25% less source code, while the binary code size stays approximately the same. The numbers shown in Table 1 do not include the additional source code for the ARM host or the communication library, since it is similar in all cases and used by the Cal2Many code generator as well.

In each case, about half of the development time was spent on understanding both the problem and the algorithm, which is independent of the choice of programming language. The actual implementation times for each algorithm varied. One of the developers had no prior experience with CAL and required approximately 20% more time for the CAL version, while the other developers required slightly more time for their C implementations. While this is by no means a hard measure, it provides an idea on the complexity of CAL. More importantly, the CAL implementations are completely oblivious of the underlying hardware and are easily portable, while the C implementations are quite restricted to the Epiphany system architecture. This higher level of abstraction also reduced the debugging effort, which is extremely cumbersome in low-level parallel programming.

6. CONCLUSIONS

Parallel implementations show up to 30x better performance in terms of execution time when compared to the library implementation. However, the Givens Rotations implementation shows that with bigger matrices the speed-up increases. Since the implementations are scalable, we believe that with larger local memory or larger number of cores the implementations can decompose bigger matrices and achieve higher speed-ups.

While using the external memory, Givens Rotations is slightly better than the other implementations, however, Householder method outperforms the others when local memory is used. Givens Rotations has higher amount of computation and more overlap between memory reads and computation. High computation amount increases the execution time when local memory is used. However, when external memory is used, due to the overlap, it shows the best performance. One should keep in mind that the number of cores is smaller for Givens Rotations implementation. In case of development complexity, Table 1 shows that implementing Givens Rotation requires more coding whereas the other implementations have similar code sizes.

As an average result of the three implementations, the generated code runs 4.3% slower than the hand-written code while using the external memory. When the internal memory is used, the slowdown is around 17% whereas the average source lines of code that is needed for the CAL implementations is 25% smaller. When the required knowledge level and development time and complexity is taken into account, the slowdown seems reasonable. It is easier to develop and debug parallel applications in CAL rather than in low level languages provided by manycore developers. The custom communication library reduces the burden however, it does not help with debugging and requires debugging itself.

In the future, Parallella board with 64 cores can be used for gaining higher speed-ups, and in order to decrease the effect of communication even further, direct memory access feature of Epiphany architecture can be analyzed.

Acknowledgment

This work was supported in part by the ESCHER Project funded by the Swedish Knowledge Foundation, the HiPEC project funded by the Swedish Foundation for Strategic Research, and the ELLIIT Strategic Research Initiative funded by the Swedish Government.

7. REFERENCES

[1] Gnu scientific library [online accessed] http://www.gnu.org/software/gsl/, March 2016.

[2] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 932 –943, may 2011.

[3] A. Buttari, J. Langou, J. Kurzak, and J. Dongarra. Parallel tiled qr factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[4] D2C. CAL ARM compiler. http://sourceforge.net/projects/opendf/, Accessed: 3 Aug 2013, 2013.

[5] J. Eker and J. W. Janneck. Dataflow programming in CAL – balancing expressiveness, analyzability, and implementability. In *Conference Record of the Forty Sixth Asilomar Conference on Signals, Systems and Computers (ASILOMAR), 2012*, pages 1120–1124. IEEE, 2012.

[6] E. Gebrewahid, M. Yang, G. Cedersjo, Z. Ul-Abdin, V. Gaspes, J. W. Janneck, and B. Svensson. Realizing efficient execution of dataflow actors on manycores. In *Embedded and Ubiquitous Computing (EUC), 2014 12th IEEE International Conference on*, pages 321–328. IEEE, 2014.

[7] W. M. Gentleman and H. Kung. Matrix triangularization by systolic arrays. In *25th Annual Technical Symposium*, pages 19–26. International Society for Optics and Photonics, 1982.

[8] G. H. Golub and C. F. Van Loan. *Matrix computations*, volume 3. JHU Press, 2012.

[9] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Tile qr factorization with parallel panel processing for multicore architectures. In *Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–10. IEEE, 2010.

[10] J. W. Janneck. Actor machines - a machine model for dataflow actors and its applications. *Department of Computer Science, Lund University, Tech. Rep. LTH*, pages 96–2011, 2011.

[11] A. Olofsson, T. Nordström, and Z. Ul-Abdin. Kickstarting high-performance energy-efficient manycore architectures with epiphany. In *48th Asilomar Conference on Signals, Systems and Computers*, 2014.

[12] ORCC. Open RVC-CAL compiler. http://orcc.sourceforge.net/, Accessed: 3 Aug 2013.

[13] S. Raase. A dataflow communications library for adapteva's epiphany. Technical report, Halmstad University, 2015.

[14] S. Raase and T. Nordström. On the use of a many-core processor for computational fluid dynamics simulations. *Procedia Computer Science*, 51:1403–1412, 2015.

[15] G. Roquier, M. Wipliez, M. Raulet, J.-F. Nezan, and O. Déforges. Software synthesis of CAL actors for the MPEG reconfigurable video coding framework. In *15th IEEE International Conference on Image Processing, 2008. ICIP 2008.*, pages 1408–1411. IEEE, 2008.

[16] S. Savas. Linear algebra for array signal processing on a massively parallel dataflow architecture. Technical Report ID: 2082/3452, Halmstad University, 2008.

[17] S. Savas, E. Gebrewahid, Z. Ul-Abdin, T. Nordström, and M. Yang. An evaluation of code generation of dataflow languages on manycore architectures. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2014 IEEE 20th International Conference on*, pages 1–9, 2014.

[18] B. Zhou, R. P. Brent, et al. Parallel implementation of qrd algorithms on the fujitsu ap1000. *17th Australian Computer Science Conference, Christchurch, New Zealand, January*, 1994.