QR DECOMPOSITION ON EPIPHANY USING HOUSEHOLDER TRANSFORMATION

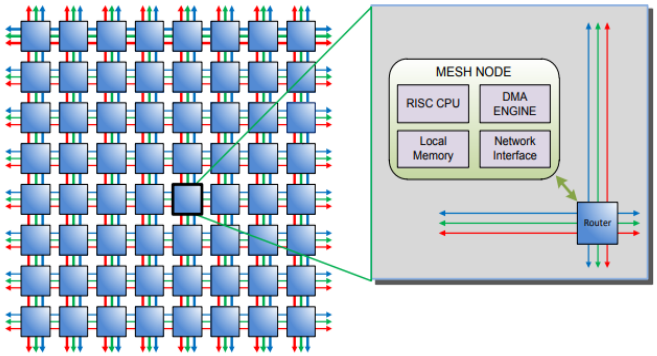
Embedded Parallel Computing Final Project

By Vinay Sawant

INTRODUCTION

Parallel computing is the study in which many calculations and execution of processes are implemented simultaneously. The one big task is divided into smaller tasks and can be run at the same time to get the desired output. There are many forms of parallelism such as TLP (Task Level Parallalism), DLP (Data Level Parallelism), ILP (Instruction Level Parallelism), TLP (Thread Level Parallelism), etc.

In this project we will work on the parallelism on "Parallella Board Epiphany Architecture". Epiphany Architecture consists of 16 cores, scalable, distributed shared memory, and parallel computing fabric that consist of a 2D mesh network-on-chip communication network. It includes floating-point RISC CPU in every mesh node as well as local memory for each node of size 32 KB, DMA engine, and network interface card as shown in fig. below.



The advantages of such architectures are –

- They are programmable in C/C++, openMP, actor programming languages;
- They utilize low power;
- They can be extended up to thousands of cores.

The implantation in a parallel manner is always time-efficient. There are many applications that can be implemented in a parallel manner. One such application is QR decomposition.

This project is supposed to implement the QR decomposition of a matrix on the host ARM processor, single core and multiple cores. The QR decomposition of a matrix is a decomposition of matrix A into the product A = QR where Q is an orthogonal matrix and R is an upper triangular matrix. There are numerous methods by which we can perform QR decomposition as Gram-Schmidt Reflections, Givens Rotation, Householder Transformation. In this project we will deal with the QR decomposition using the Householder Transformation method.

METHOD

This given project is implemented using 'Householder Transformation' algorithm. The advantage of using householder transformation is that it's most stable algorithm in terms of numerical calculations because of the use of reflections it uses to produce the upper triangular matrix R. The Householder algorithm sometimes called HH algorithm. This transformation takes a vector and reflects it about some hyper plane. We use this method to factorize m x n matrix A with m is equals to n into the product QR.

The step by step operations of this method is stated below.

- HH method is one of the orthogonal transformation which transforms a vector x into unit vector y parallel to it. We calculate the Householder reflection matrix using following formula.

$$H = I - 2\text{vv}^{\text{T}}$$

We need to build the H matrix so the Hx = α e1 where e1 = [1 0 0…]

- As H is orthogonal, we get α = +/- ||x||. Here the sign is considered because it has opposite sign of x1 the first element of x vector thus the vector u which we want can be obtained from below

$$u = \begin{bmatrix} x_1 + sign(x_1)||x_1|| \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

And the unit vector v we can use as u = v/||v||. Then the householder matrix can be calculated as

$$H(x) = I - 2vv^T = I - 2\frac{uu^T}{u^T u}$$

- Consider the input matrix as A and its first column as c1. The first step is to find the reflection of this column vector as shown below

v1 = a1 + sign(a11)||a1||e1.

- The next step is to calculate the first H matrix say H1 using above formula and we start to construct our upper triangular matrix R by producing sub diagonal elements as zeros using H1 A = $\begin{bmatrix} c1 & c2.. & cn \\ 0 & d1 & d2 \\ 0 & d3 & d4 \end{bmatrix}$

- Then we have to move on to the next iteration in which we consider the second column but not whole second column; instead we take the second column of sub-matrix d1, d2, d3, d4 i.e. d1 and d3.

- By using these sub-matrix elements we again repeat the procedure to calculate v vector and H matrix until we get the upper triangular matrix R.

- Thus by above stated procedure we can say that we will get the R by multiplying all generated H matrices with A i.e HnHn-1….H1A = R and for calculating Q matrix we have to take product of only generated H matrices as H1H2H3……Hn = Q.

- Thus we will get final output as R and Q matrices and if we take product of QR, we can finally able to regenerate input matrix A.
- If the matrix is not square matrix i.e m> n then the equations become

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

IMPLEMENTATION

1. The implementation of Householder Transformation is carried out on host ARM processor, single core and multiple cores. On ARM, the implementation is in sequential form and it is done in parallel way by using different number of cores. **The implementation for Epiphany core/s is same for all cores starting from 1 to16 for 4X4, 8X8 and 16X16 matrices.** The preprocessor macro *"COREUSED"* can be defined in the core program (testcode.c) as per the number of cores required for the parallelism**.** As we change the numbers of this variable with the number of cores we wanted, the program will run on that number of cores.
2. The implementation is divided into seven stages.
3. **Stage 0** – Input stage. This stage is used to initialize timer 0 and timer 1. Also, it is responsible to get the input matrix and set the stage flag to next stage.
4. **Stage 1** – This stage is used to calculate the householder matrices. The number of matrices varies as per the input matrix size. Eg. if we have 4 X 4 matrix then we will get 4 householder matrices. From the array of such householder matrices, last array index is stored with identity matrix. This is very useful in achieving parallelism.

5. **Stage 2** – This is the main stage of parallelization. In this stage, all HH matrices generated above are gathered and the consecutive matrices pairs are passed to different cores to receive fast matrix product. Such parallel process reduces the number of cycles in Q transpose matrix calculation. Let's consider 16X16 input matrix, say HH matrices as H_1, H_2, ….., H_{15}, H_{16}. The sequential algorithm requires M-1=15 iterations to get their final product. In parallel implementation, consecutive pairs H_2&H_1, H_4&H_3, and so are multiplied in parallel. Then the product of H1&H2 say Ha and the product of H3&H4 say Hb and so on will be multiplied to get further matrices. And, finally one Q transpose gets generated after such matrix multiplications. In such a way the Q transpose parallelism is done.
6. **Stage 3** – In this stage R matrix is calculated by taking the left side product of all H matrices with input matrix A. Meaning, $H_{m-1}*H_{m-2}*….H_1*H_0*A=R$. On other hand, $Q^T*A = R$.
7. **Stage 4** – Final Q matrix is calculated here from Q transpose taken from Stage-2.
8. **Stage 5** – In this stage, the product of upper triangular matrix R and orthogonal matrix Q is calculated in order to check if we get correct input matrix.
9. **Stage 6** – Output stage. The flag is incremented to indicate completion of core processes and thus the host core now can show the results and terminate the execution of core/s and host.
10. For single core all stages run on the single core. The 0^{th} core is used for single core implementation.
11. For multicore implementation, only the Stage-2 uses the core-0 and other cores. All other stages are implemented only on core-0.

[Note: here, core-0 has row id = 0 and column id = 0;

core-1 has row id = 0 and column id = 1;

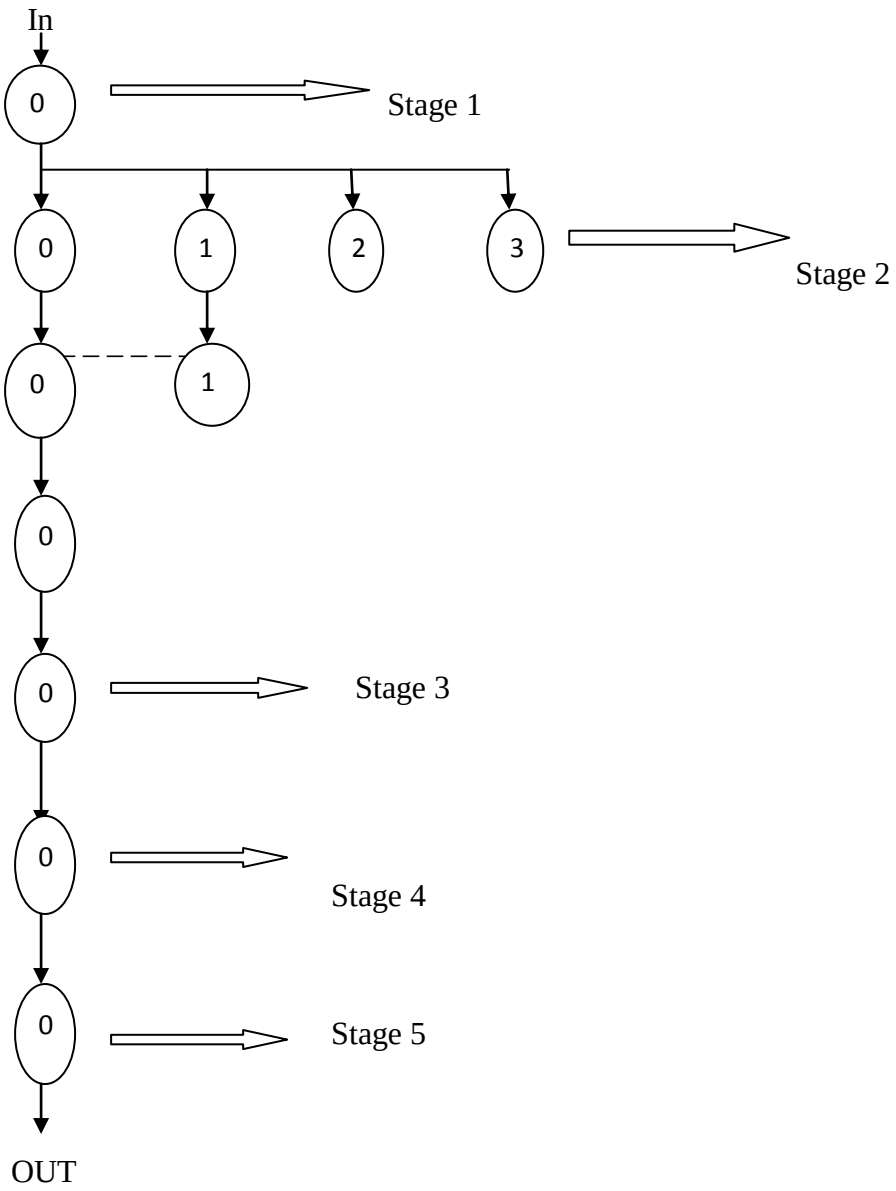
core-2 has row id = 1 and column id = 1; ….

Core-14 has row id = 3 and column id = 2;

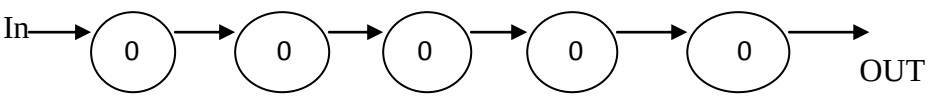
core-15 has row id = 3 and column id = 3]

The stage distribution among the cores is shown below

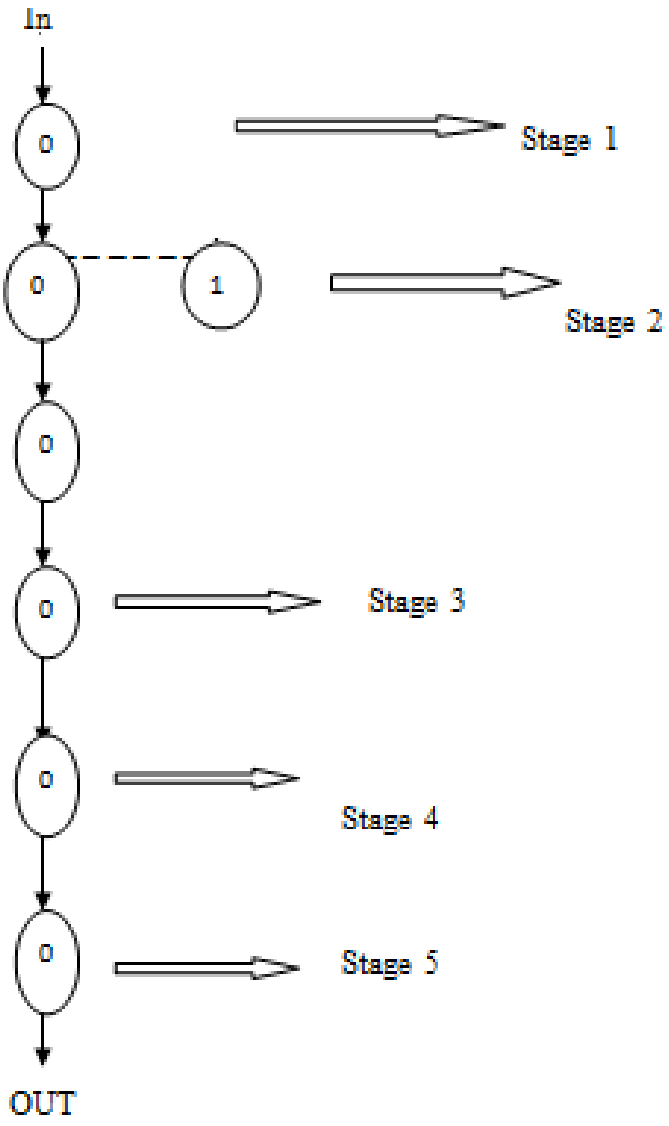
1. 4 X 4 matrix on 4 cores



2. 4 x 4 matrix on single core



3. 4 X 4 matrix on 2 cores


In
0
Stage 1
0
1
Stage 2
0
0
Stage 3
0
Stage 4
0
Stage 5
OUT

RESULTS

1. 4 x 4 matrix results on ARM processor

```
R:

 10.93226  5.55947  8.33346  7.84045
  0.00000  6.11881  6.22626  1.72934
  0.00000 -0.00000  8.87492  8.39817
  0.00000 -0.00000 -0.00000 -3.21930

Q:

  0.46876  0.86502 -0.15166  0.09487
  0.23076 -0.04425  0.80325  0.54734
  0.78072 -0.49954 -0.34791  0.14103
  0.34276 -0.01538  0.45907 -0.81947

A:

  5.12458  7.89895  7.94621  3.59206
  2.52277  1.01217  8.77633  6.71655
  8.53506  1.28380  0.30817  1.88151
  3.74719  1.81150  6.83489  9.15431

parallella@parallella:~/vinsaw19/EpiphanyProject/demo_arm/4x4$ time ./ep_main
-bash: ./ep_main: No such file or directory

real    0m0.004s
user    0m0.000s
sys     0m0.000s
```

2. 4 X 4 on single core

```
Q=

   0.46876   0.86502  -0.15166   0.09487
   0.23076  -0.04425   0.80325   0.54734
   0.78072  -0.49954  -0.34791   0.14103
   0.34276  -0.01538   0.45907  -0.81947

R=

  10.93226   5.55947   8.33346   7.84045
   0.00000   6.11881   6.22626   1.72934
   0.00000   0.00000   8.87492   8.39817
   0.00000  -0.00000   0.00000  -3.21930

A=

   5.12458   7.89895   7.94621   3.59206
   2.52277   1.01217   8.77633   6.71655
   8.53506   1.28380   0.30817   1.88151
   3.74719   1.81150   6.83488   9.15431

stage-0 cylces = 201
stage-1 cylces = 2838888
stage-2 cylces = 3600813053
stage-3 cylces = 75153
stage-4 cylces = 16058
stage-5 cylces = 106360
stage-6 cylces = 221
stage-7 cylces = 0
stage-8 cylces = 0
stage-9 cylces = 0
stage-10 cylces = 1200200158
shm.stage:6.000000
shm.test:0.000000
Size of shm:560
Program finished.
```

3. 4 X 4 on two cores

```
Q=

   0.46876   0.86502 -0.15166   0.09487
   0.23076 -0.04425   0.80325   0.54734
   0.78072 -0.49954 -0.34791   0.14103
   0.34276 -0.01538   0.45907 -0.81947

R=

  10.93226   5.55947   8.33346   7.84045
   0.00000   6.11881   6.22626   1.72934
   0.00000   0.00000   8.87492   8.39817
   0.00000 -0.00000   0.00000 -3.21930

A=

   5.12458   7.89895   7.94621   3.59206
   2.52277   1.01217   8.77633   6.71655
   8.53506   1.28380   0.30817   1.88151
   3.74719   1.81150   6.83488   9.15431

stage-0 cylces = 201
stage-1 cylces = 2846095
stage-2 cylces = 2400542438
stage-3 cylces = 75160
stage-4 cylces = 16369
stage-5 cylces = 106839
stage-6 cylces = 221
stage-7 cylces = 0
stage-8 cylces = 0
stage-9 cylces = 0
stage-10 cylces = 1200200158
shm.stage:6.000000
shm.test:0.000000
Size of shm:568
Program finished.
```

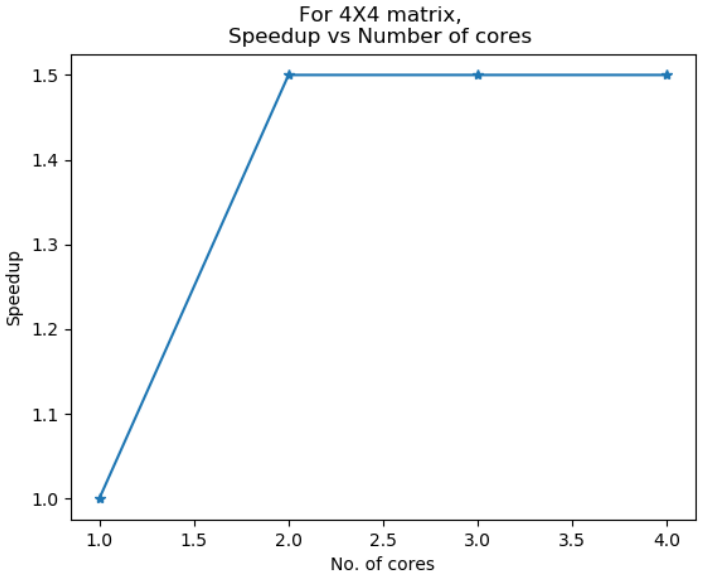
4. 4 X 4 Matrix on 3 cores

```
   0.46876   0.23076   0.78072   0.34276
   0.86502 -0.04425 -0.49954 -0.01538
  -0.15166   0.80325 -0.34791   0.45907
   0.09487   0.54734   0.14103 -0.81947

Q=

   0.46876   0.86502 -0.15166   0.09487
   0.23076 -0.04425   0.80325   0.54734
   0.78072 -0.49954 -0.34791   0.14103
   0.34276 -0.01538   0.45907 -0.81947

R=

  10.93226   5.55947   8.33346   7.84045
   0.00000   6.11881   6.22626   1.72934
   0.00000   0.00000   8.87492   8.39817
   0.00000 -0.00000   0.00000 -3.21930

A=

   5.12458   7.89895   7.94621   3.59206
   2.52277   1.01217   8.77633   6.71655
   8.53506   1.28380   0.30817   1.88151
   3.74719   1.81150   6.83488   9.15431

stage-0 cylces = 201
stage-1 cylces = 2828240
stage-2 cylces = 2400542279
stage-3 cylces = 72073
stage-4 cylces = 16122
stage-5 cylces = 108304
stage-6 cylces = 221
stage-7 cylces = 0
stage-8 cylces = 0
stage-9 cylces = 0
stage-10 cylces = 1200200158
shm.stage:6.000000
shm.test:0.000000
Size of shm:576
Program finished.
```

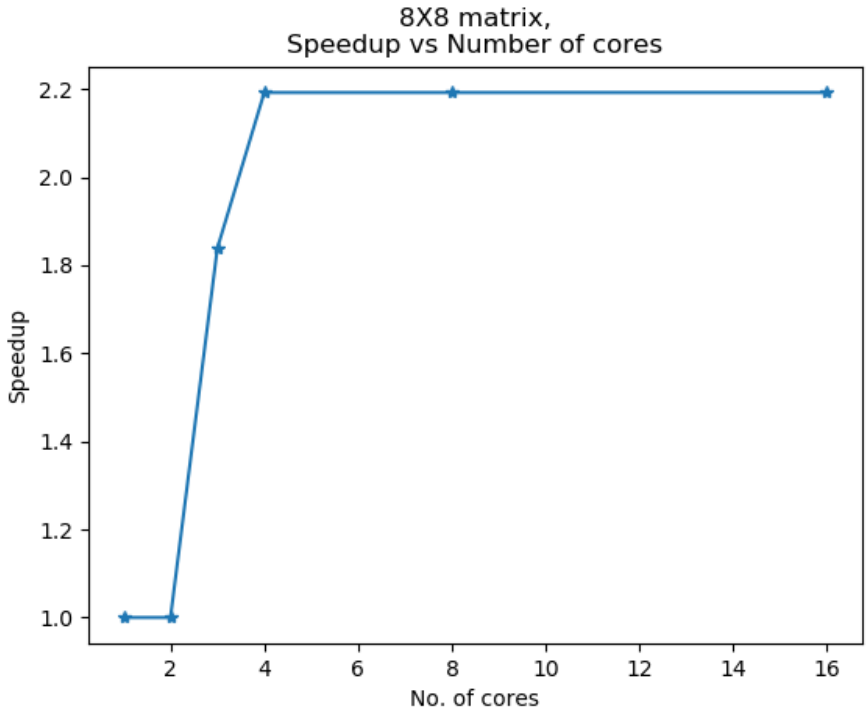
[Note: If number of cores is equal or more than number of rows of input matrix then the execution cycles doesn't change significantly].

RESULTS:

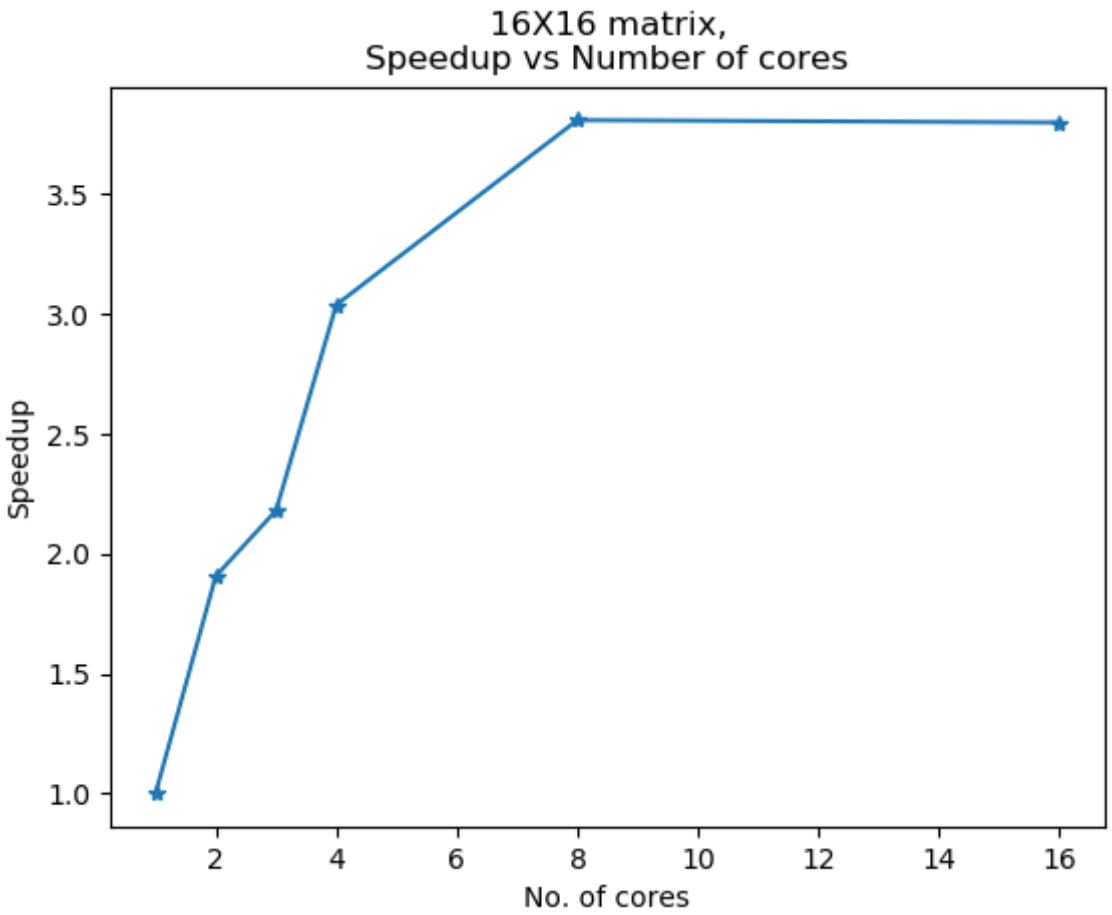
M = 4 , N =4			
No. of cores used (P)	Execution time for parallel process (Stage 2 in program)	Speedup	Efficiency
1	3600813053	1.0	1.0
2	2400542438	1.499999	0.75
3	2400542279	1.499999	0.75
4	2400542279	1.499999	0.75



M = 8 , N =8			
No. of cores used (P)	Execution time for parallel process (Stage 2 in program)	Speedup	Efficiency
1	7897255131	1.0	1
2	7897255131	1.0	0.5
3	4294967467	1.8387229220425665	0.61290764
4	3602291964	2.192286247178825	0.54807156
8	3602291676	2.1922864224501515	0.2740358
16	3602289676	2.1922876396129114	0.13701798



M = 16 , N =16			
No. of cores used (P)	Execution time for parallel process (Stage 2 in program)	Speedup	Efficiency
1	68407193	1.0	1
2	35873448	1.906903205958903	0.9534516
3	31368478	2.180762260763815	0.72692075
4	22509608	3.0390219589785836	0.75975549
8	17957018	3.809496264914364	0.47618703
16	1800388	3.799580590406068	0.23747379


16X16 matrix,
Speedup vs Number of cores
Speedup
No. of cores
1.0
1.5
2.0
2.5
3.0
3.5
2
4
6
8
10
12
14
16

DISCUSSION

Householder transformation is simple and numerically stable QR factorization algorithm. There are many instances to perform parallelism during householder reflection operations viz. matrix multiplications, additions, divisions, square-root, transpose, etc. One of the way is discussed in this report.

Talking about the Epiphany architecture, it is easy to use distributed shared memory. However, the implementation using shared memory has some limitations. The update of memory locations need some time. To illustrate, if one core has written new value to a variable in shared memory; other core have old value and it can't have new value until some time elapsed. For this reason, software delay is introduced in between write and read cycles. Moreover, it is hard to do programming using pointers in Epiphany cores. The example codes in the reference manual could have helped to simplify the work.

REFERENCES

1. Dataflow Implementation of QR Decomposition on a Manycore **.**Süleyman Savas, Sebastian Raase, Essayas Gebrewahid, Zain Ul-Abdin and Tomas Nordström
2. Högskolan i Halmstad / IDE Department 2008. Linear Algebra for Array Signal Processing on a Massively Parallel Dataflow Architecture. Author Süleyman Savas 861006-N494
3. https://en.wikipedia.org/wiki/QR_decomposition
4. https://rpubs.com/aaronsc32/qr-decomposition-householder