CJS: Custom Jacobi Solver

Andreea-Ingrid Cross
Imperial College London
London, UK
andreea.funie09@imperial.ac.uk

Liucheng Guo
Imperial College London
London, UK
liucheng@tg0.co.uk

Wayne Luk
Imperial College London
London, UK
w.luk@imperial.ac.uk

Mark Salmon
Cambridge University
Cambridge, UK
mhs39@cam.ac.uk

ABSTRACT

The classic Jacobi method, widely used for solving linear systems, is slow, especially when dealing with large matrices. This paper proposes a Custom Jacobi Solver (CJS) for large-scale linear systems. It is based on a column-wise Jacobi step operation which allows for increased dependence distance, enabling deep pipelining. Our solver allows customisation at run time between the classic Jacobi method and its more convergence efficient-counterpart, the weighted Jacobi method. It can be dynamically scaled to multiple FPGAs by appropriately partitioning the matrix data among the FPGAs. After evaluating our solver on a number of different datasets, CJS proves to be up to 71 times faster when comparing an 8-FPGA solution with a 12-core CPU C++ implementation.

ACM Reference Format:
Andreea-Ingrid Cross, Liucheng Guo, Wayne Luk, and Mark Salmon. 2018. CJS: Custom Jacobi Solver. In *The 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2018), June 20–22, 2018, Toronto, ON, Canada.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3241793.3241802

1 INTRODUCTION

Solving linear systems of equations is a common problem encountered in many scientific and engineering fields, such as: finance (e.g. portfolio optimization [1]), machine learning (e.g. regularization), energy load flow, etc. Nowadays we have access to abundant computational resources, allowing the scientific computing problems to grow both in size and complexity. Methods such as Jacobi, Gauss-Seidel, Conjugate Gradient, Multigrid or Steepest Descent are some of the most used across industrial and scientific applications. Among all of those, the Jacobi method is relatively simple, but stable.

One advantage of the Jacobi method is that rounding errors would not be accumulated as they are restricted just to the previous operation, however, this method is very time-consuming for large-scale linear systems. Its solving time usually dominates the total computing time, thus accelerating the solving process becomes of great importance. The Jacobi method has huge parallelism potential. For example, *O'Leary and White* [2] introduce the multi-splittings of the coefficient matrix parallel scheme which is used on many different architectures. Due to the algorithm's highly parallelizable nature, we could linearly scale the parallelism performance with the number of available cores, however the more cores used - the more expensive and the less-energy efficient the cluster becomes.

We are interested in achieving a trade-off between speedup, usability and energy-efficiency. Thus we will further show what has been achieved using tools such as GPUs and FPGAs. There are some examples from the state of art research performed on GPUs [3, 4], achieving 5 times to 74 times speedup over an equivalent CPU solutions. Some studies using FPGAs have shown more than 37 times speedup when compared to similar CPU solutions [5, 6].

Our study focuses on designing and implementing an efficient pipelined Jacobi method on the FPGA which targets large linear-systems and preserves accuracy. One issue technologies like FPGAs and GPUs have is their limited onboard memory. Our study addresses this by allowing the algorithm to dynamically scale to multiple FPGAs by appropriately partitioning the matrix data among them. We also address the convergence challenges when solving large linear systems by allowing the user to select at run-time between the classic Jacobi and the weighted Jacobi method. The latter highly improves the convergence rate of the algorithm, making it more usable in large-scale data-set scenarios. Finally, the last challenge addressed is energy efficiency: solving large scale linear-systems can lead to high power consumption when using CPUs and even GPUs, however, our solution proves to be approximately 57 times more energy efficient than its CPU counterpart. The main contributions of our research are:

- A novel linear system solver, CJS, based on column-wise Jacobi step operation to increase dependence distance, enabling deeply pipelined implementations;
- The first deeply pipelined Weighted Jacobi design for FPGAs;
- Customization of CJS at run-time, allowing users to select the classic Jacobi method or the weighted Jacobi method and choose other respective parameters such as the weight parameter or the convergence rate threshold;
- Run-time scalability which enables the use of multiple FPGAs, providing maximum parallel efficiency and allowing the solution of a large linear system ($> 94 * 10^9$ matrix coefficient elements);

HEART 2018, June 20–22, 2018, Toronto, ON, Canada

ACM ISBN 978-1-4503-6542-0/18/06…$15.00
https://doi.org/10.1145/3241793.3241802

- Experimental results showing the speedup and energy measurements when compared with the one-core/multi-core C++ CPU based implementation.

2 BACKGROUND

2.1 The Classic Jacobi Method

The Jacobi algorithm is used to solve linear algebraic systems of "n" equations with "n" unknowns which are diagonally dominant and stable. The method works as follows: an initial approximate solution x_0 is selected and through an iterative solver the algorithm attempts to find the real solution x. The iterative method terminates when the algorithm reaches the actual solution (if one exists), or when the predefined number of iterations have been exhausted [7]. To describe the method more mathematically, let us consider an algebraic system described by as single matrix equation `Ax=b`, where `A` is the matrix of coefficients, `x` is the solution of the system and `b` is the column-matrix with the constant terms respectively. To solve `Ax=b` iteratively, the algorithm tries to follow the classic approach of solving a quadratic equation in which the next value x^{k+1} is computed as a function of the current value x^k, i.e.$x^{k+1} \Leftarrow f(x^k)$ The Jacobi iteration point form is:

$$x_i^{k+1} \Leftarrow \frac{1}{a_{ii}}[b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k] \quad (1)$$

where x_i^{k+1} represents the jacobian form at iteration *i*. The classic Jacobi method converges slower than other similar techniques, such as Gauss Seidel or Successive over Relaxation and its convergence is only guaranteed for weakly diagonally dominant matrices. We do not analyze convergence in this paper, but only improve it by introducing an optimized Jacobi algorithm, the *weighted Jacobi*.

2.2 The Weighted Jacobi

The classical Jacobi method has high parallel efficiency; however it does not always show great improvement because of the rather poor convergence rate. Therefore, in order to improve the quality of the classic Jacobi solver, we introduce an improvement of the Jacobi method, which is known as a weighted-Jacobi type method or preconditioner.

This method is also called the damped Jacobi method or the relaxed Jacobi method and its point form becomes [8]:

$$x_i^{k+1} \Leftarrow (1-\omega) * x_j^k + \omega * \frac{1}{a_{ii}}[b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k] \quad (2)$$

where $\omega \epsilon R$ is called the weight parameter, with $\omega = 2/3$ being the usual choice [8]. The algorithm is similar to the classic Jacobi algorithm, but now a weight parameter is used as well when constructing the residual value in between iterations.

2.3 Hardware-based Jacobi Methods

Because of the feature of Jacobi Methods, researchers have used hardware to accelerate them, such as GPUs and FPGAs. Some examples from the state of art research performed on GPUs are: *Wang et al.* [3] present a GPU-based solution for the Jacobi method in which the performance of their tool scales linearly with the matrix size, achieving 5 times better than an equivalent CPU solution. *Lin et al.* [4] implemented an GPU-based Jacobi method which achieves a 10.2 times speedup over an equivalent CPU solution.

Some studies using FPGAs are: *Morris et al.* [5] have implemented the first FPGA version of a deeply pipelined classic Jacobi algorithm. It shows a 37 times speedup over its CPU counterpart. *Ruan et al.* [6] has implemented a pipelined friendly FPGA-based Jacobi algorithm eliminating the data dependence between iteration steps, which is 115 times faster than a similar CPU solution. Due to the lack of details in the paper such as, how the matrix is constructed, the exact size of the matrix, or how the CPU implementation is built, we cannot properly compare our work to theirs - as we will further see in the evaluation section, matrices with different characteristics show different results in terms of performance. However, to understand better the performance of our solution compared to other existent GPU and FPGA ones, we provide a comparison table in Section 5 of some of the most efficient solutions.

3 CUSTOM JACOBI SOLVER DESIGN

In this section we exploit the high level of internal parallelism which can be achieved with the use of FPGA-based technology (in our case, a MAIA dataflow engine (DFE) containing an FPGA and DRAM), to accelerate the Jacobi method. We show how our design can be extended to take advantage of larger commercial chips, where multiple parallel processing pipelines can be deployed concurrently to speedup the computation further. Our design can be used for numerous iterations as a standalone linear systems solver as well as dynamically scale with the number of FPGAs available, while preserving accuracy and achieving a throughput of one add per clock cycle.

The accelerator model targeted by our design is represented by a CPU based system which connects via a slow interconnect to an FPGA accelerator. Most of the computation is performed on the FPGA. Both CPU node and FPGA acceleration board have large on-board memory available, of which we make use, as the transfer speed from on-board memory is much faster than via the interconnect. All data are residing initially in the CPU DRAM.

We make the following choices as part of our design and implementation characteristics: 1) We verify that the matrix is weakly diagonal dominant on the CPU before sending it to the FPGA; 2) Because we perform a column-wise summation instead of a row-wise one, in order to improve pipeline efficiency on the FPGA, we transpose the matrix on CPU beforehand and send its transpose to the accelerator DRAM; 3) For the weighted Jacobi scenario we use the popular value of $\omega = \frac{2}{3}$ for the weight parameter [8], to reduce the computation overhead needed to pre-compute the eigenvalues on CPU beforehand; 4) We guess the initial solution X_0 as being the null vector; 5) Matrix coefficients, the initial solution values as well as the constant terms column-matrix b are single precision floating point values on DRAM input; 6) The maximum number of iterations to run the Custom Jacobi Solver for is another popular choice of $2 * n * n$ [7], where n is equal to the number of equations and unknowns of the linear system to solve.

The system needed to be evaluated, is modeled in a linear algebra form on CPU and its components are then transferred to the FPGA for the Jacobi iteration to be computed. CJS allows for the following parameters to be customised by the user: number of FPGAs, number

of pipes that each FPGA uses, iteration number, weight parameter, convergence rate threshold, Jacobi method type and the row and column dimensions of the matrix.

Single Iteration of the Custom Jacobi Solver For an FPGA pipeline-friendly version of the Jacobi method we introduce a new vector, *Diag* which consists of each element from the main diagonal of the coefficient matrix. We replace the main diagonal elements from the matrix with 0s, so they do not add any value when included into the partial adder computation performed on the FPGA.

Our single iteration Jacobi solver design is: 1) Load coefficient matrix A as an array, together with the constant-terms column-matrix b, the initial solution x_0 and the main-diagonal elements vector *Diag*, to accelerator DRAM; 2) Load scalar value *JacobiType* onto FPGA BRAM, with a 0 value for the weighted Jacobi method and a 1 value for the classic Jacobi method; load all other scalar values (weight parameter, threshold value); 3) Compute the classic or weighted Jacobi iteration as in Equation 1 or Equation 2, and write new x_1 vector solution to accelerator DRAM by overlapping the old x_0; 4) Read new x_1 vector solution from accelerator DRAM and output results to CPU.

Parallelism and Pipelining. We can parallelise the design efficiently as long as sufficient coefficients need to be used in the Jacobi computation. Hence, we further improve the performance of the proposed design by implementing multiple parallel processing pipelines on-chip (*pipes*). Each pipe is a computation architecture as presented below. Multiple blocks of rows — up to p pipes can be evaluated in parallel, substantially reducing the overall computation time. This is an efficient method to take advantage of the high degree of fine grained parallelism on the FPGA: at each point in time a number of floating point elements equal to the number of matrix coefficients in the row blocks are used to achieve one equation solution result per row.

In our design we perform a column-wise Jacobi step operation instead of a row-wise one. This is achieved by transposing the original coefficient matrix and sending it to the FPGA in this form. When doing a column sum in this manner, each iteration of the inner loop is not dependent on the previous iteration of the inner loop, as it is in a row-wise computation, but it is instead dependent on the previous iteration of the outer loop. This increased dependence distance allows us to push a different sum with each input into the adder pipeline every tick, as we have plenty of time for the results to come back round from the adder output. Thus, for summation column-wise, each iteration depends on a value from m iterations earlier, so the dependence distance is m - width of array. Figure 1 describes best the parallelism present in our design.

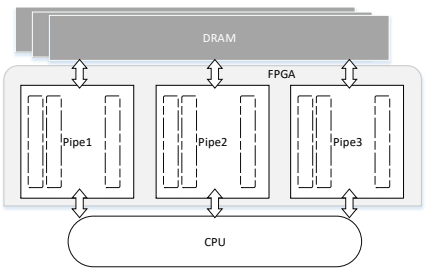

Figure 1: CJS using Multiple Pipes on One FPGA

Our design consists of a binary reduction tree, a subtraction unit, a divider and some control units organized in processing elements. Firstly each matrix diagonal element a_{ii} and each constant-term b_i are fed into the divider, respectively the subtraction unit. We then ingest the coefficient matrix A column elements together with the vector x_k. The elements we add every tick at iteration $k+1$ are of the form $a_{ij}x_j^k$ and the binary tree reduces the inputs to produce the sum $\sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k$. We use a multiplexer to chose between computing the classic Jacobi or the weighted Jacobi method, according to the value of the scalar input *JacobiType* (a 0 value picks the weighted Jacobi, while a 1 value chooses the classic Jacobi method). Using the processing element *JacobiStep* we feed the produced sum together with the respective b_i element into the subtraction unit to produce a value *interim* equal to $b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k$. This *interim* value and the appropriate a_{ii} value, enter the division unit which delivers the vector x_{k+1}, one value per clock cycle. To compute the weighted Jacobi method, we use the processing element *WeightedJacobiStep*. In this unit we feed the value produced by the *JacobiStep* processing element to a multiplier, together with the weight parameter ω and obtain $\omega * \frac{1}{a_{ii}}[b_i - \sum_{j=1;j\neq i}^{j=n} a_{ij} * x_j^k]$. We then feed the result into an adder, together with a multiplier whose result corresponds to $x_j^k * (1-\omega)$. This then delivers the vector x_{k+1}, one value per clock cycle, according to Equation 1.

We introduce *readEnable* delay units to ensure that the correct $Diag_i$ and b_i values are applied to the division and subtraction units when needed. We ensure that the term $a_{ii} * x_i$ is ignored by introducing a 0 value for the a_{ii} term existent in the coefficient array A saved on DRAM.

Multiple Iterations Jacobi Solver Model We can compute multiple iterations of the Jacobi algorithm, using a hybrid CPU-FPGA tool. To allow for convergence checking we implement an extra processing element *DidConverge*. This is computed on FPGA in the same time with the *ClassicJacobiStep* or the *WeightedJacobiStep*. and at the end of the iteration outputs the result back to the CPU as an integer value, namely 0 (false - convergence not reached) or 1 (true - convergence reached). This *DidConverge* processing element feeds in the new x_j^{k+1} value and the old x_j^k value to a subtraction unit. The subtraction result is compared to a preset threshold (i.e. *eps = 1.0e-4*) and if it is below the threshold then we can assume we reached convergence (i.e. send a 1 value to CPU).

Custom Jacobi Solver Model on Multiple FPGAs As described previously, we are bound to the available DRAM memory size of each FPGA. To solve this, we calculate the number of bytes required for our linear system to be computed, knowing that we save everything on accelerator DRAM as single precision floating-point. We attempt to split the data to be stored, equally between the FPGAs, such as if we have a matrix of $n \times n$ and m available FPGAs, we would split the matrix in m matrices with $\frac{n}{m}$ rows and n columns each. If an equal split is not possible, we attempt to fill as many FPGAs as possible with blocks of rows as follows: we get the quotient of $\frac{n}{m}$ and this is how many full FPGAs we can fill, each with a matrix of $\frac{n}{m}$ rows and n columns each. The remainder data goes on a final FPGA. Each 1U unit contains 8 DFEs (each with an FPGA and DRAM). If we need to use more than 8 FPGAs, we

would need to manage communication between 1U units to avoid bottlenecks. We leave this as further work.

Due to the nature of our hardware design and implementation, we need to check for convergence on every sub matrix to see that the whole matrix converges. After each iteration, we read the current iteration x solution vector from each of the accelerator's DRAM, as now each FPGA will provide a solution vector with a size equal to $\frac{1}{m}$ of the original x solution vector one, corresponding to the matrix size computed. After all partial x solution vector results are read from all DFEs, we need to reconstruct the whole current x solution vector and re-write it back to the same location in each of the FPGA's DRAM. If convergence has been reached, we need to output it back to the CPU as the main linear system solution. Thus, the start of every iteration is depending on when the update of the new x vector is finished. The CPU controls the computing process on all DFEs and maintains their synchronization.

4 HARDWARE IMPLEMENTATION

The accelerator system consists of a Maxeler MPCX node with a Maia DFE containing a Stratix V 5SGSMD8N1F45C2 FPGA and 48 GB of DRAM. It is connected to a CPU node with a Dual Intel Xeon E5-2640 via Infiniband through a Mellanox FDR Infiniband switch.

FPGA Implementation Most accelerator configurations have a reduced bandwidth between the host CPU and the accelerator card in comparison to the memory bandwidth. In our design most of the input values will be reused for each iteration of CJS. As such these will be stored in DRAM and only incur the transfer penalty over the slow interconnect between the CPU and FPGA once. The only values that change are the values of the x solution vector - hence, we write those values on DRAM as they are computed. We read the x value from DRAM only when convergence was reached.

Each of our FPGA has 48GB of DRAM which equals to a storage of approximately $12 * 10^9$ single precision floating-point (4 bytes) values or $6 * 10^9$ double precision floating-point (8 bytes) In our case, we need to store on DRAM the new x solution vector values, as well as the B constant-term column-matrix elements, the main diagonal elements $Diag$, alongside with the corresponding array of coefficients for the matrix A. Thus, we restrict ourselves to a matrix with a smaller number of elements, due to the additional storage required for the remaining arrays. With a total of 8 FPGAs available (all 8 FPGAs are accommodated in a single 1U node), we are able to increase the restriction of the matrix size further, by scaling up and splitting the matrix to be computed between multiple FPGAs. In the next section we show the performance of our system when computing the Jacobi method on different size dense and sparse matrices, using one FPGA as well as multiple ones.

CPU Implementation The CPU implementation is built using C++11, parallelised using OpenMP and compiled using g++ 4.9.2 with flags `-O3 -march=native -fopenmp` to enable general performance and architectural optimisations for the Intel XEON and the use of multi-threading. The CPU code is parallelised in a similar manner to the hardware implementation, by dividing the matrix by groups of rows/columns. To efficiently parallelise the Jacobi algorithm, the devised schemes should achieve data locality, minimize the number of communications, and maximize the overlapping between the communications and the computations. The distribution of the cells can be performed in a row-wise or a column-wise fashion. The row-wise approach is preferable for the software implementation as it makes better use of data locality. By assuming, for simplicity, that the number p of threads divides exactly the dimension n of the $n \times n$ matrix A and the vectors x and B, blocks of $m = n/p$ rows of the matrix A are distributed to the threads, while, vectors x and b are scattered in the same way.

Table 1 shows our CPU implementation's scalability with the number of threads for a $107,520 \times 107,520$ matrix. We present the average time taken to perform one Jacobi iteration on CPU after 10 independent runs. We choose to disable HyperThreading and only use 6 threads per CPU therefore preventing the CPU implementation from scaling sub-linearly. We notice close to linear scaling for the CPU implementation of one iteration. These are expected results given that our parallelisation strategy requires minimal communication between threads. For multiple Jacobi iterations however, communication overhead will be introduced by waiting for each thread to compute its result before reconstructing the x vector and copying it back to each thread for the next Jacobi iteration.

Table 1: CPU scalability results for up to 12 threads and one Jacobi iteration for a $107,520 \times 107,520$ matrix

# Threads	1	2	4	8	10	11	12
CPU Time (s)	457.31	240.69	117.26	57.89	46.66	42.66	39.66
Speedup	1	1.9	3.9	7.9	9.8	10.7	11.5

5 EVALUATION

We evaluate the software results on the CPU node attached to the MPCX node, namely a Dual Intel Xeon E5-2640 with 6 cores. The Stratix V FPGA node adopts a 28nm transistor technology, while the Dual Intel Xeon E5-2640 adopts a 22nm transistor technology.

Our FPGA implementation runs at a clock frequency of $190MHz$. All run times are measured using the `chrono::high_resolution_clock::now()` high resolution clock which is part of the C++11 standard library. We perform different tests to evaluate CJS:

(1) A large scale linear system benchmark which consists of a matrix randomly generated that has the following properties: 1) n × n matrix; 2) n + 1 on the diagonal and 1 elsewhere; 3) the constant-term elements of the column matrix b are all equal to $2 * n$. We use those properties for testing purposes, such that we know that the exact solution at convergence x will be a vector of ones. We show speedup of CJS for one iteration and multiple iterations of the Jacobi method. We use just 9 iterations results for the purpose of this paper, as this is how long it takes for our algorithm to reach convergence when using the weighted Jacobi method.
(2) Predicting the price of a house sale.
(3) Numerous speedup results on different sparse matrices.

The accuracy of the CPU and FPGA versions are checked for all the tests performed and they are very similar.

Resource Usage Table 2 shows the FPGA total resource usage expressed as a percentage of the total available resource on the chip for the *single precision floating- point* implementation based on 1, 8 and 16 pipes. The resource usage is shown for the manager and

Table 2: FPGA total resource usage for single precision floating-point arithmetic implementation

# Pipes	LUTs	FFs	BRAMs	DSPs	of use
1	3.96%	4.13%	15.35%	0.00%	by manager
1	1.37%	0.82%	8.65%	0.15%	by kernels
1	11.97%	8.44%	28.01%	0.15%	total resources
8	6.32%	5.38%	27.97%	0.00%	by manager
8	8.82%	5.07%	16.24%	1.22%	by kernels
8	21.91%	13.97%	48.23%	1.22%	total resources
16	6.14%	5.45%	28.75%	0.00%	by manager
16	27.16%	15.48%	26.80%	4.89%	by kernels
16	40.04%	24.45%	59.56%	4.89%	total resources

kernels of our design, as well the total design resource usage which is represented by the kernels resource usage and the IO resource usage. The kernels provide an environment concentrated around data flow and arithmetic. The manager provides an interface to the kernels which incorporates the configuring connectivity between kernels and external I/O, as well as the build process control.

Energy and Power Consumption Table 3 shows that solving a linear system with 107,520 unknowns and 107,520 equations using the Stratix V FPGA, proves to be up to 57 times more energy efficient than its CPU counterpart implementation on a Dual Intel Xeon E5-2640. The matrix was randomly generated using the method described earlier and each FPGA uses 16 pipes for computation.

Table 3: Energy and Power Consumption for CPU and FPGA technology for a $107,520 \times 107,520$ matrix

Technology	12-core CPU	1 FPGA	8 FPGA
Power (Watt)	90.00	16.40	124.80
Energy (Joules)	34,189.2	598.60	667.68
Energy Efficiency	1x	57x	51x

5.1 Performance Results and Discussion

For simplicity, we ignore all initial CPU-DRAM transfer time. We thus show speedup obtained just by measuring the execution times of both the CPU and FPGA implementations. We note that even platforms with large amounts of DRAM will likely require in an order of tens of seconds at most to re-load data (loading 48GB over an Infiniband 2GB/s connection). The CPU-DRAM transfer issue can be addressed efficiently either by increasing problem sizes (and using adequate input distribution) – the DRAM transfer overhead becomes negligible compared to the savings in execution time, or by tighter integration between CPU and FPGA, such as Intel's new Xeon/Altera CPUs to reduce CPU to FPGA transfer time. Since these points correspond to trends in industry at the moment of writing, we believe we can safely ignore the initial CPU-DRAM transfer time for the time being and focus on the actual execution times.

5.1.1 Single FPGA Parallel Jacobi. Table 4 shows speedup obtained when comparing one as well as multiple iterations (i.e. our method reaches convergence in 9 iterations) of the Jacobi method, implemented on a single FPGA across a different number of pipes. The performance is evaluated against a 12-thread C++ implementation.

Table 4: Single FPGA results for 107,520 x 107,520 Matrix

# of pipes	1	4	8	16
# of iterations	1			
CPU Time (s)	39.66	39.66	39.66	39.66
FPGA Time (s)	60.90	15.28	7.68	3.84
FPGA Speedup	**0.69**	**2.60**	**5.16**	**10.33**
# of iterations	9			
CPU Time (s)	379.88	379.88	379.88	379.88
FPGA Time (s)	552.31	141.89	72.75	36.50
FPGA Speedup	**0.69**	**2.68**	**5.22**	**10.41**

Our FPGA runs at $190MHz$, a much lower clock frequency than that of our CPU processor of $2.60GHz$. Hence, the multi-threaded CPU implementation outperforming the single pipe FPGA doesn't come as a surprise. We further make some comparisons which are not present in the table: 1) if we were to compare a single-core C++ implementation against a single pipe on the FPGA then the FPGA would prove 7.51 times faster; 2) if we were to compare an equal number of CPU threads implementation to an equal number of pipes, then an 8-pipe FPGA solution would prove 7.54 times faster than an equivalent 8-threads C++ implementation. From our tables it is also clear that the more parallelism inside the FPGA, the greater the performance achieved. A considerable increase in the performance is also noticed with an increase in the number of iterations needed for the Jacobi method to converge.

5.1.2 Multiple FPGAs Parallel Jacobi. Table 5 shows the potential speedup when comparing one as well as multiple iterations of the Jacobi method, implemented across multiple FPGAs, each of them using 16 pipes and evaluated against a 12-thread C++ implementation. The CPU controls the computing process on all DFEs and maintains their synchronization. All multi-FPGA performance results include synchronization time. We notice a linearly increase in the speedup with the number of FPGAs used when compared to the CPU implementation. We observe a peak of 71 times speedup when comparing an 8-FPGA system versus a 12-core CPU one.

Table 5: Multi-FPGAs results for 107,520 x 107,520 Matrix

# of FPGAs	1	2	4	8
# of iterations	1			
CPU Time (s)	39.66	39.66	39.66	39.66
FPGA Time (s)	3.84	1.96	1.00	0.56
FPGA Speedup	**10.33**	**20.23**	**39.66**	**70.82**
# of iterations	9			
CPU Time (s)	379.88	379.88	379.88	379.88
FPGA Time (s)	36.50	18.54	9.54	5.35
FPGA Speedup	**10.41**	**20.48**	**39.84**	**70.97**

Table 6 shows how CJS is able to scale to make use of all 8-available FPGAs by splitting the original matrix of size $n \times n$ (rowSize by colSize) in sub-matrices, each with a corresponding size of $m \times n$, where $m = n/8$ (in our case, each FPGA will evaluate a 38,400 x

307,200 matrix). Each of the FPGAs will use 16-pipes, in order to achieve great parallelism. We only show the 8-FPGA system because we generated a matrix (restricted to 48GB) that will not fit into the DRAM of one FPGA only, but it will fit into 8 FPGAs. Hence, with our design we could solve a fairly large-scale linear system (a matrix with over $94 * 10^9$ coefficient elements), in under 37 seconds. We are unable to provide a comparison with the CPU implementation, since the matrix does not fit within our CPU's DRAM.

Table 6: 8-FPGA System for 307,200 x 307,200 Martix

Number of iterations	1	9
FPGAs Time (s)	3.91	36.97
FPGA Energy (Joules)	487.97	4,613.86

5.1.3 Predicting the Price of a House Sale. The House Prices challenge was introduced by Kaggle and it includes 79 explanatory variables describing aspects of residential homes in Ames, Iowa, from 2006 until 2010, for a total of 2930 observations, as a Boston dataset [9] extension. The problem reduces to a multiple linear regression which aims to forecast the final price of each home and is solved using the Least Squares method. This problem reduces to solving the linear system $\hat{\beta} = (X'X)^{-1}X'Y$, where $'$ is the transpose of the matrix, *-1* is the matrix inverse, matrix X is the feature matrix (i.e.characteristics of residential homes) and vector y is the response value (i.e.price of a home). This can we rewritten as $\hat{y} = X\hat{\beta}$ and we use CJS, to find the vector of $\hat{\beta}$ values. Table 7 shows that with one FPGA we are able to achieve a 20 times speedup when compared to our own fully optimized software implementation.

Table 7: Performance Results Comparison

Study	Accelerator (Threads/Pipes)	Speedup
CJS	Intel E5-2640 (12)	**1x**
CJS	Altera Stratix V(1)	**1.26x**
CJS	Altera Stratix V(16)	**19.68x**

5.1.4 Speedup Results for Different Sparse Matrices. We apply the Least Squares method using CJS on some sparse matrices from the *Suite Sparse Matrix Collection* [10]. Our design is not optimized for sparse matrices, but we show some speedup in Table 8. The results are provided after an average of 10 runs of one iteration of CJS when comparing the 12-core CPU implementation and one FPGA.

Table 8: Performance Results Comparison

Matrix	Speedup
$Maragal_6$ $(21,255 \times 10,152)$	**3.05x**
$Maragal_7$ $(46,845 \times 26,564)$	**3.16x**
$Maragal_8$ $(33,212 \times 75,077)$	**3.31x**

Qualitative Comparisons. Table 9 shows speedup obtained by our work and other different Jacobi solutions implemented on GPU and FPGA. All the different technologies, parameters, operating frequencies and implementations used make it difficult to obtain an exact view on the performance. In the future, we plan to compare our current work's efficiency by extrapolating the clock frequency of the other available solutions tested on older technologies.

Table 9: Performance Results Qualitative Comparison

	CPU(Threads)	Accelerator(#)	Size	Speedup
[11]	Intel i7-3960x(1)	Stratix IV(1)	1024	**155.28x**
[12]	Intel i7-960(8)	Nvidia C2070(1)	1024	**73.50x**
[13]	AMD 4200(4)	Nvidia 8800GT(1)	1024^2	**31.74x**
CJS	Intel E5-2640(12)	Stratix V(8)	$11.6 * 10^9$	**70.97x**

6 CONCLUSION AND FUTURE WORK

In our study we show the effectiveness of FPGAs in accelerating both the classic Jacobi and the weighted Jacobi method. Using both our deeply-pipelined single floating-point implementation and the multiple FPGA scalable solution, we show that one of the most computationally intensive tasks associated with solving large-scale linear systems of equations, the Jacobi algorithm, can be accelerated substantially by exploiting the massive amounts of on-chip parallelism available on commercial FPGAs.

When evaluating CJS on around $11.56 * 10^9$ values, our floating-point precision multiple FPGAs system proves to be up to 71 times faster and 57 times more energy efficient when compared to a corresponding multi-threaded C++11 implementation running on two six-core Intel Xeon E5-2640 processors. We have also tested our system on a real-world application with various sparse matrices, achieving a considerable speedup.

Future work includes supporting mixed precision formats, optimizing for sparse matrices, and computing multiple iterations on chip. We also plan to apply CJS to well-known applications, such as market portfolio optimization [1].

ACKNOWLEDGMENTS

The support of the United Kingdom EPSRC (grant numbers EP/P010040/1, EP/L00058X/1 and EP/N031768/1), European Union Horizon 2020 Research and Innovation Programme (grant number 671653), Intel, Maxeler, and China Scholarship Council is gratefully acknowledged.

REFERENCES

[1] D. Leon, et al., *Clustering algorithms for Risk-Adjusted Portfolio Construction*, ICCS, 2017.
[2] D.P. O'Leary, R.E. White, *Multi-splittings of matrices and parallel solution of linear systems*, SIAM J. Algebr. Discrete Methods, 6(4), pp. 630-640, 1985.
[3] T. Wang, et al.,*Implementation of jacobi iterative method on graphics processor unit*, Proceeding of IEEE ICIS, 3, pp. 324-327, 2009.
[4] J. Lin, et al., *Design and Implementation of Jacobi Algorithms on GPU*, AICI, 2010.
[5] G R. Morris, et al., *An FPGA-Based Floating-Point Jacobi Iterative Solver*, ISPAN '05 Proceedings of ISPAN, pp. 420-427, 2005.
[6] H. Ruan, et al., *Jacobi-Solver:A Fast FPGA-based Engine System for Jacobi Method*, Res. J. Appl. Sci. Eng. Tech., 6(23), pp. 4459-4463, 2013.
[7] M. Oieshanskii, et al., *Iterative Methods for Linear Systems : Theory and Applications*, Society for Industrial and Applied Mathematics, 2014.
[8] A. Imakura, et al. *A parameter optimization technique for a weighted Jacobi-type preconditioner*, JSIAM Letters, 4, pp. 41-44, Japan Society for Industrial and Applied Mathematics, 2012.
[9] D. De Cock, *Ames, Iowa: Alternative to the Boston Housing Data as an End of Semester Regression Project*, Journal of Statistics Education, 19(3), 2011.
[10] Suite Sparse Matrix, *https://sparse.tamu.edu.*
[11] M. Torun, et al., *FPGA, GPU, and CPU implementations of Jacobi algorithm for eigenanalysis*, JPDC, 96(C), pp. 172-180, 2016.
[12] M. Torun, et al., *Novel GPU implementation of Jacobi algorithm for Karhunen-LoÃİve transform of dense matrices*, CISS, 2012.
[13] R. Amorim, et al., *Comparing CUDA and OpenGL implementations for a Jacobi iteration*, HPCS, 2009.