Research Journal of Applied Sciences, Engineering and Technology 6(23): 4459-4463, 2013
DOI:10.19026/rjaset.6.3452
ISSN: 2040-7459; e-ISSN: 2040-7467
© 2013 Maxwell Scientific Publication Corp.
Submitted: March 01, 2013 Accepted: March 14, 2013 Published: December 15, 2013

Research Article

Jacobi Solver: A Fast FPGA-based Engine System for Jacobi Method

[1]Huabin Ruan, [2]Xiaomeng Huang, [3]Haohuan Fu and [4]Guangwen Yang
[1, 4]Department of Computer Science and Technology,
[2, 3]Center for Earth Science System, Tsinghua University, Beijing, 100084, China

Abstract: The classical Jacobi method is widely used for solving linear systems. This method is considerably time-consuming to compute millions upon millions of linear equations. In this study, we design a novel FPGA-based Jacobi Solver. The kernel of the Jacobi Solver is a pipeline-friendly iteration algorithm which can eliminate the data dependence between iteration steps. This algorithm is suitable for pipeline-friendly hardware architecture. The experimental results show that the Jacobi Solver can solve more than 6.5 million of linear equations in one second and achieves up to 341x speedup compared to a single-thread CPU version.

Keywords: Dependency, hardware, Jacobi method, speedup

INTRODUCTION

With abundant computational resource being available, scientific computing are growing in size and complexity. Many scientific computing problems lead to the demand of solving large dimension linear systems, e.g., $>10^7$ (Young, 2003).

Numerous methods have been proposed to solve linear systems, such as Direct Solution method, Jacobi method, Gauss-Seidel method, Conjugate Gradient method, Multigrid method and Saad and Van Der Vorst (2000). Among all these methods, Jacobi Method is relatively simple and independent of the numbering of the unknowns (Cavallaro and Luk, 1988). The obvious advantage seen was that rounding errors would not be accumulated, they are restricted to the last operation. However, Jacobi method is very time-consuming to compute millions upon millions of linear equations.

Fortunately, Jacobi method has much more potential for parallelization. For example, O'Leary and White (1985) introduce a parallel scheme based on multi-splittings of the coefficient matrix and they prove the convergence of this method with some sufficient conditions on the coefficient matrix and on its splittings (Bru and Fuster, 1990). Morris and Prasanna (2005) implemented a FPGA-based single iteration of Jacobi method without considering convergence and achieved 1.7x faster than CPU implementation in one iteration. Wang *et al*. (2009) present a GPU-based implementation of Jacobi method. Their experimental results show that the performance of GPU-based Jacobi method algorithm increases linearly along with the scale of matrix increasing and finally achieved 3x faster than Barrachina *et al*. (2008) GPU-based dense linear system.

Our study also focuses on modifying Jacobi method to make it suitable for FPGA architecture. We design a FPGA-based Jacobi Solver which mainly includes a new pipeline-friendly iteration algorithm. The main idea of the pipeline-friendly iteration algorithm is eliminating the data dependence between iteration steps. To achieve this goal, we have to reconstruct the work flow of the classical Jacobi method with algorithmic transformations. After enabling pipeline of different computation stages, we finally implement the Jacobi Solver in hardware logic and successfully make it run on Vitex-6 SX475T FPGA. In order to evaluate the performance of our FPGA-based Jacobi Solver, we implement the other three different CPU-based Jacobi method solutions. These CPU-based solutions are all fully optimized. The Experimental results show that our FPGA-based Jacobi Solver is significantly faster than all CPU-based solutions. Our Jacobi Solver achieves a maximum of 341x speedup over a single-thread CPU-based solution, 115x speedup over a multi-thread CPU-based solution and 35x speedup over a MPI-based solution.

MATERIALS AND METHODS

Fundamental of Jacobi method: The Jacobi method is used to solve linear systems with matrix that without zeros along its main diagonal. Each diagonal element is solved for and an approximate value is plugged in. Referring to the expressions in Bronshtein *et al*. (1985), we given a system of n linear equations:

$$Ax = b \tag{1}$$

Corresponding Author: Huabin Ruan, Department of Computer Science and Technology, Tsinghua University, Beijing, 100084, China
This work is licensed under a Creative Commons Attribution 4.0 International License (URL: http://creativecommons.org/licenses/by/4.0/).

where,

$$A = \begin{bmatrix} a_{11}, a_{12}, \ldots, a_{1n} \\ a_{21}, a_{22}, \ldots, a_{2n} \\ . \\ . \\ . \\ a_{n1}, a_{n2}, \ldots, a_{nn} \end{bmatrix}, \quad x = \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ x_n \end{bmatrix}, \quad b = \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ b_n \end{bmatrix} \tag{2}$$

Then A can be decomposed into a diagonal component D and the remainder R:

$$A = D + R \tag{3}$$

where,

$$D = \begin{bmatrix} a_{11}, 0, \ldots, 0 \\ 0, a_{22}, \ldots, 0 \\ . \\ . \\ . \\ 0, 0, \ldots, a_{nn} \end{bmatrix}, \quad R = \begin{bmatrix} 0, a_{12}, \ldots, a_{1n} \\ a_{21}, 0, \ldots, a_{2n} \\ . \\ . \\ . \\ a_{n1}, a_{n2}, \ldots, 0 \end{bmatrix} \tag{4}$$

The solution is then iteratively solved by:

$$x^{(k+1)} = D^{-1}(b - Rx^{(k)}) \tag{5}$$

The Eq. (5) can be converted into an element-based formula:

$$x_i^{(k+1)} = \frac{1}{a_{ii}}\left(b_i - \sum_{j \neq i} a_{ij} x_j^{(k)}\right), i = 1, 2, \ldots, n \tag{6}$$

where, k is the iteration count. The x_i will be computed iteratively until the solution x = $(x_1, x_2, \ldots, x_n)^T$ reaches convergence. The condition for the convergence is that for a given precision ε, for example, $\varepsilon = 10^{-9}$, it satisfies following condition:

$$\frac{\|b - Ax^k\|}{\|b\|} < \varepsilon \tag{7}$$

Based on Eq. (6), we write the pseudo-code of Jacobi method in Algorithm 1. Line 7 to Line 12 in Algorithm 1 continuously update component xi in current $(k+1)^{th}$ iteration and δ in Line 9 is the accumulating result of $\sum_{j \neq i} a_{ij} x_j^{(k)}$. When all components $(x_1, x_2, \ldots, x_n)^T$ for solution x are updated, the algorithm starts to check whether the result is convergence or not. If solution x = $(x_1,\ x_2, \ldots, x_n)^T$ reaches convergence at the i_{th} iteration, then the iteration process will finish. Otherwise the algorithm starts next iteration step.

It is obvious that there is data dependency between δ and x_j when new iteration starts. This kind of data dependency makes the classical Jacobi method impossible to be implemented in hardware logic (Weaver *et al*., 2003). We need to design a pipeline-friendly algorithm for FPGA architecture.

Algorithm 1: The Classical Jacobi Method:

```
Choose an initial guess value $x_0$;
k = 0;
Check if convergence is reached;
While convergence not reached do
  for i: = 1 step until n do
     δ = 0;
     for j: = 1 step until n do
       if j ≠ i then
          δ = δ + $a_{ij}\ x_j^{(k)}$;
       end if
     end for
     $x_i^{(k+1)}$ = ($b_i$ - δ)/ $a_{ii}$ ;
  end for
  Check whether convergence is reached;
  k = k + 1;
end while
```

The target hardware: We use Maxeler's MAX3 Acceleration card to implement our engine system of Jacobi method. MAX3 acceleration card consists of one Vitex-6 SX475T FPGA and 24 GB DDR3 onboard memory. Figure 1 illustrates the architecture of our used Maxeler acceleration system. The host application runs on the conventional CPU and manages the interaction between the host and the FPGA accelerators. On the FPGA, kernels are the hardware designs implementing the arithmetic and logical computations needed within an algorithm. The manager is the collective term for the FPGA logic that orchestrates data flow between kernel and off-chip I/O.

Dividing an application into kernels and a manager lets us separate computation from communication. This is beneficial because it enables deeply pipelined kernels without control flow constraints, which is key to achieving high performance. Managers use a streaming model for off-chip I/O to PCI Express and DDR3 RAM memory and are optimized to achieve high use of available bandwidth in off-chip communication channels, allowing kernels to run at peak performance (Lindtjorn *et al*., 2011).

Additionally, the MAX3 acceleration card is also attached with a compiling platform named MaxCompiler provided by Maxeler Technologies. Using MaxCompiler, we only need to program the algorithm of Jacobi method in Java and the MaxCompiler will compile and build the Java code into a configurable bitstream file through the Vendor's CAD software that are integrated in MaxCompiler. The generated bitstream file can be loaded into Maxeler acceleration card at runtime.

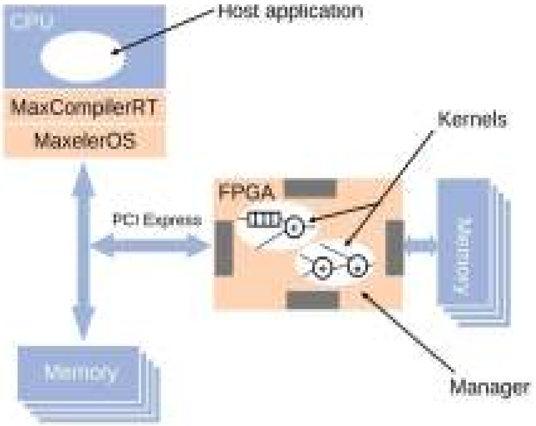

Fig. 1: Architecture of maxeler acceleration card

Design of FPGA-based Jacobi solver: In this section, we design a pipeline-friendly algorithm to eliminate the data dependence between iteration steps. We divide the complete large scale linear systems into blocks, in which one block consists of a small number of linear equations. All equations in one block will be simultaneously solved when the pipeline-friendly algorithm finishes iterations.

We can see from Line 9 in Algorithm 1 that updating the δ value is one important step of Jacobi method. This step needs all components x_j (j = 1, 2, ⋯, n) that have been calculated in previous iteration. If we just simply migrate the step of updating δ to FPGA hardware logic, the computing process will fail because the calculation for x_i needs to pass through a long pipeline stages in FPGA. When a new iteration step starts, a number of x_j (j = k, k + 1, k + 2, ⋯, n) are still non-value. It will lead to an invalid updating for δ.

In order to solve the above problem, we reconstruct the classical Jacobi method to a pipeline-friendly Jacobi method which is shown in Algorithm 2. The main improvement of using block is solving a number of equations simultaneously during one iteration step, while the original method is only solving a single equation during one iteration step.

We assume T FPGA clock cycles are needed before all components $(x_1, x_2, \dots, x_n)^T$ are figured out for all the linear equations in a block. Before updating the δ value, we calculate every x_i in parallel as following steps:

- **Step 1:** In t_1 cycle, the algorithm starts to calculate x_i for the 1st equation in block.
- **Step 2:** In t_2 cycle, the algorithm starts to calculate x_i for the 2nd equation in block. Meanwhile, it still process rest calculation on xi for the 1st equation in block.
- **Step i:** In t_i cycle, the algorithm starts to calculate x_i for the ith equation in block. Meanwhile, it still process rest calculation on x_i for previous i equations.

Algorithm 2: The Pipeline-Friendly Jacobi Method:

```
Choose an initial guess solutions $x_0^1$, $x_0^2$,…, $x_0^T$ to all equations in block;
k = 0;
check if all equations in block have reached convergence:
while all equations in block have reached convergence do
  for t : = 1 step until T do
     $\delta_t$ = 0;
  end for
  for i : = 1 step until n do
    for t : = 1 step until T do
      for j : = 1 step until n do
        if j ≠ i then
           $\delta_t = \delta_t + a_{ij}\, x_{tj}^{(k)}$;
        end if
      end for
      $x_{ti}^{(k+1)} = (b_{ti} - \delta_t)/\, a_{ii}$;
    end for
   end for
 check whether T equations in current block have reached convergence;
 k = k + 1;
end while
```

After T FPGA clock cycles elapse, all components $(x_1, x_2, \dots, x_n)^T$ for the 1st equation are figured out. Therefore the updating for the first equation's δ can be correctly performed in new iteration. When the second cycle starts in new iteration, all components $(x_1, x_2, \dots, x_n)^T$ for the 2nd equation are also figured out since it also has elapsed T FPGA clock cycles when calculation on xi for the 2nd equation starts, so updating δ for the 2nd equation can also be correctly performed. The process is the same for updating δ for other equations in block when following cycles start in new iteration.

Using this pipeline-friendly Jacobi method, we can overlap multiple computing stages for updating δ and results in eliminating the data dependency between consequent iteration steps. This method is easy to be implemented by FPGA hardware logic.

Since Algorithm 2 needs to solve T equations simultaneously during the iterations, we change the terms δ, x_i, x_j, b_i in Algorithm 1 to δ_t, x_{ti}, x_{tj} and b_{ti} in Algorithm 2 respectively, where:

- δ_t is the corresponding δ for t_{th} equation in block.
- x_{ti} and x_{tj} are the corresponding component x_i and x_j respectively for t_{th} equation in block.
- b_{ti} is the corresponding b_i for t_{th} equation in block.

RESULTS AND DISSCUSSION

We implement the pipeline-friendly Jacobi method on Maxeler MAX3 acceleration card with one Vitex-6

Table 1: Performance comparison (solved equations per second)

Data set ID	Equation dimension	Single-thread CPU version	Multi-thread CPU version	MPI version	FPGA version
1	2	2,031,034	3,100,000	271,428	6,544,444
2	4	589,000	1,840,625	205,944	3,926,666
3	8	313,297	879,104	146,517	2,945,000
4	16	85,985	252,789	65,299	1,840,625
5	32	29,129	81,578	47,576	1,682,385
6	64	7,968	22,750	36,335	950,000
7	128	1,786	4,869	28,816	463,779
8	200	871	2,574	23,179	297,474

Table 2: Speedup result (FPGA-based solution V.S. CPU-based solutions)

Data set ID	Equation dimension	FPGA v.s. single-thread CPU version	FPGA v.s. multi-thread CPU version	FPGA v.s. MPI version
1	2	3x	2x	24x
2	4	6x	2x	19x
3	8	9x	3x	20x
4	16	21x	7x	28x
5	32	57x	20x	35x
6	64	119x	41x	26x
7	128	259x	95x	16x
8	200	341x	115x	12x

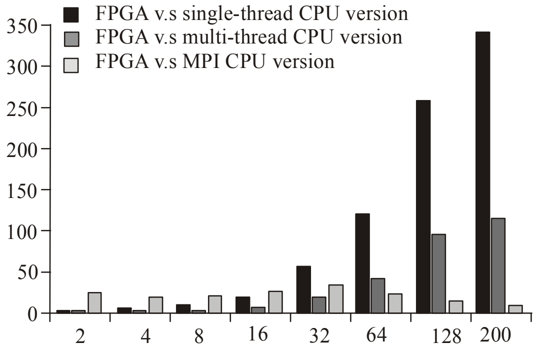

Fig. 2: Speedup of FPGA v.s. other CPU versions

SX475T FPGA. Using MaxCompiler software tool suites provided by Maxeler Technologies, we only need to program the pipeline-friendly Jacobi method in Java and the MaxCompiler will compile and build the Java code into a configurable bitstream file through the CAD software integrated in MaxCompiler. The generated bitstream file can be loaded into Maxeler acceleration card at runtime.

For the purpose of comparison, three different CPU versions of Jacobi method including a single-thread CPU version, a multi-thread CPU version and a MPI CPU version are also developed and tested. All these versions are fully optimized in the C program language and compiled with the Intel Compiler configured at the O3 optimization level.

The single-thread CPU version and multi-thread CPU version are deployed in a PC workstation with one 2.93 GHz quad-core Intel i7 CPU and a 4GB DDR3 memory. The MPI version is deployed over a 16-nodes cluster with Intel MPI Library configured at highest optimization level and each node powered by a 2.93 GHz quad-core Intel i7 CPU and a 4 GB DDR3 memory. Moreover, the MPI version is performed with 64 processes. Here we run the MPI-based implementation in a limited number of processes only for simple reference purpose, rather than discussing the scalability of Jacobi method. The FPGA-based implementation is deployed over a Maxeler MAX3 acceleration card with a Vitex-6 SX475T FPGA running at 175 MHz and 24 GB DDR3 onboard memory.

All above four implementations of Jacobi method are tested according to eight datasets, which are generated randomly with different linear dimension from small to large. The linear dimension of eight datasets is 2, 4, 8, 16, 32, 64, 128 and 200, respectively. Limited by the quantity of hardware resources provided by Vitex-6 SX475T FPGA on one Maxeler MAX3 acceleration card, the maximum linear dimension that Jacobi Sovler can solve is 200. If more hardware resources are available, the maximum linear dimension can be extended.

Note that the amount of computation to be performed within Jacobi method is proportional to linear dimension. Therefore we keep the amount of linear equations at 1 million in every dataset for performance comparison.

We present the number of processed equations per second for different versions in Table 1 and the speedup of FPGA version over other versions in Table 2. For simplification, we plot the Fig. 2 to compare the speedup in detail.

As shown in Table 1, the average throughput of FPGA-based implementation is high. For two dimensions scenario, it can solve 6,544,444 equations in one second. For two hundred dimensions scenario, it still can solve about 297,474 equations in one second. These peak performance results are much higher than the other three CPU versions.

As shown in Table 2, with the increasing of dimension in each dataset, the speedup is obviously keeping rising except MPI version. In our test cases, most of the best speedups are achieved on the largest dataset (i.e., two hundred dimensions scenario) with

341x speedup over the single-thread CPU version, 115x speedup over the M-CPU version. Comparing with the MPI-based solution, FPGA solution achieves maximum 35x speedup when dimension is 32. When dimension increases to 200, the FPGA solution only achieves 12x faster than MPI-based solution. The main reason for this result is the time saved in computation process is greater than time consumed in the overhead of forking 64 MPI processes. Our studies in this case are still limited to the number of MAX3 card. If more MAX3 cards are available, we can compare multi-FPGA version with MPI version further.

The pipeline-friendly Jacobi method enables us to deploy sufficient FPGA functional units and is easy to be implemented in deep pipeline FPGA hardware logic. Furthermore, if more hardware resources on FPGA platform being available, we can deploy more than one pipeline for this pipeline-friendly Jacobi method in FPGA which will result in further acceleration.

CONCLUSION

This study presents Jacobi Solver, a fast FPGA-based design of Jacobi Method. It is useful to solve large-scale linear systems. We design a pipeline-friendly Jacobi method, which eliminates the data dependency between iterations and make Jacobi method not only successfully be implemented with FPGA hardware logic, but also achieve significant acceleration. Our experimental results show that Jacobi Solver is more efficient than other CPU versions significantly.

REFERENCES

Barrachina, S., M. Castillo, F. Igual, R. Mayo and E. Quintana-Orti, 2008. Solving dense linear systems on graphics processors. Proceedings of the 14th International Euro-Par Conference on Parallel Processing. Springer-Verlag Berlin, Heidelberg, pp: 739-748.

Bronshtein, I., K. Semendiaev and K. Hirsch, 1985. Handbook of Mathematics. Van Nostrand Reinhold, New York.

Bru, R. and R. Fuster, 1990. Parallel chaotic extrapolated jacobi method. Appl. Math. Lett., 3(4): 65-69.

Cavallaro, J. and F. Luk, 1988. Cordic arithmetic for an svd processor. J. Parallel. Distr. Com., 5(3): 271-290.

Lindtjorn, O., R. Clapp, O. Pell, H. Fu, M. Flynn and H. Fu, 2011. Beyond traditional microprocessors for geoscience high-performance computing applications. IEEE Micro, 31(2): 41-49.

Morris, G. and V. Prasanna, 2005. An fpga-based floating-point jacobi iterative solver. Proceeding of the 8th IEEE International Symposium on Parallel Architectures, Algorithms and Networks, pp: 8-17.

O'Leary, D.P. and R.E. White, 1985. Multi-splittings of matricesand parallel solution of linear systems. SIAM J. Algebr. Discrete Methods, 6(4): 630-640.

Saad, Y. and H.A. Van Der Vorst, 2000. Iterative solution of linear systems in the 20th century. J. Comput. Appl. Math., 123(1-2): 1-33.

Wang, T., Y. Yao, L. Han, D. Zhang, and Y. Zhang, 2009. Implementation of jacobi iterative method on graphics processorunit. Proceeding of IEEE International Conference on Intelligent Computing and Intelligent Systems, 3: 324-327.

Weaver, N., Y. Markovskiy, Y. Patel and J. Wawrzynek, 2003. Post-placement c-slow retiming for the xilinx virtex fpga. Proceeding of the 11th International Symposium on Field Programmable Gate Arrays. ACM, Monterey CA, Feb. 23-25, pp: 185-194.

Young, D., 2003. Iterative Solution of Large Linear Systems. Courier Dover Publications, Mineola, N.Y., pp: 570.