Acceleration of Dense-Matrix Tableau Simplex Method with parallel and re-configurable computing

Duo Wang
2nd year BEng in Electronic and Information Engineering
Department of Electrical and Electronic Engineering
Imperial College London

1. Introduction

Simplex method is a very popular method for finding optimal solutions of Linear Programming problems. One of the characteristics of simplex method, particularly tableau simplex method, is that a large part of the algorithm are massively parallel data processing. Therefore parallel computation using accelerators like GPU and FPGA will tremendously accelerate this algorithm.

In this implementation, CPU, GPU and FPGA are used together with the help of newly released Altera OpenCL Standard Software Development Kit(SDK) for its FPGAs. This SDK allows programmers to program the FPGA very similarly to programming the GPUs with OpenCL. Two heterogeneous platforms, one with GPU and the other with FPGA, are used to compare the performance of this particular implementation on each hardware.

2. Tableau Simplex Method

The simplex method discussed in this section is the tableau simplex method. Detailed explanation of this method can be found either online or in the book "Linear programming and network flows"(1).

2.1 Simple One-Phase simplex method

The tableau simplex method puts all matrices into one single tableau matrix. For every iteration, a set of elementary row operations called "pivoting" is done on the tableau. In this implementation, a 2-phase tableau simplex method with variable boundary conditions is used to solve the most general cases of Linear Programming problems. Next the algorithm for 1-phase tableau simplex method is first described in pseudo code because 2-phase method is built on this 1-phase method:

```
Tableau=readProblem(int problem_num);
```

```
While(not found)
{
    search for minimum Ck-Zk;
    if (all Ck-Zk<0) found=1;
    else
    {
        search for row r giving minimum(b_r/y_rk: y_rk>0);
        if all y_rk<=0 return (no solution found);
        else
            pivoting(tableau);
    }
}
```

2.2 Two-Phase Simplex Method

The 1-phase implementation can only solve the linear programming problems with constraints in the form of Ax≤B and B≥0 (referred to as simple LP cases in later discussion). For the general case of $B_l \leq Ax \leq B_u$, 2-phase method is an efficient way of finding an initial basic feasible solution to start the simplex method without having to calculate B^{-1}. The 2-phase method, in its first phase, introduces some artificial variables for all constraints violating the simple LP cases. The algorithm then eliminate these artificial variables by executing the simplex method once with a cost function having "-1" as coefficient for every artificial variable and "0" as coefficient for all other variables. This first phase will eliminate artificial variables and reach a feasible point to start the simplex method. If artificial variables cannot be eliminated, the algorithms then exit with no feasible solution found.

The pseudo code for 2-phase simplex method:

```
Tableau=readProblem(int problem_num);
For all constraint
    If (relop is ">=" or "=") Add artificial Variable for that constraint
Endfor
Initialize cost vector
Feasible=Phase I of Simplex Method(&tableau)
If (feasible)
    Resize Tableau;
    Phase II of Simplex Method();
endif
```

In the above pseudo code, the function "Phase I" and "Phase II" basically contain the iteration loop in pseudo code of 1-phase simplex method together with the

judgment on whether a feasible solution is found. “Phase I” returns whether all artificial variables are eliminated from the basis. If they are, proceed to “Phase II”.

2.3 Bounded Two-Phase simplex method

The simplex method discussed previously only assumes that all variables are non-negative. However, in some practical implementation of simplex method, the variables have upper and lower boundaries. That means we need to add the further constraints of “L<=X<=U”. The selections of k_{th} column and r_{th} row need to be modified to achieve this.

For non-negative boundary conditions, the selections of k is:

K=j for Maximum(Zj-Cj)

For bounded case, each variable can be either increased or decreased, for decreasing variable we are looking for Maximum(Zj-Cj) while for increasing variable we are looking for maximum(Cj-Zj). Therefore the new condition is:

K=j for Maximum(Maximum(Zj-Cj):$j \in R_1$, Maximum(Cj-Zj):$j \in R_2$)

Where R_1 is the set of decreasing variables and R_2 the set of increasing variables.
For non-negative boundary condition, the selection of r is:

R=i for $\text{minimum}(\frac{b_r}{y_{rk}}: y_{rk}>0)$

In bounded cases, the boundary is no longer **0** and there are both upper and lower boundaries. There are three possible boundary conditions for this case:

1.A basic variable drops to its lower bound
2.A basic variable reaches its upper bound
3.X_k itself reaches its upper bound

The selection of r can be summarized as:

$$r = i:\ for\ minimum \begin{cases} \min\limits_{1\le i\le m}(\frac{b_i - l_i}{y_{ik}}: y_{ik} > 0)\ if\ y_k > 0 \\ \min\limits_{1\le i\le m}(\frac{b_i - u_i}{y_{ik}}: y_{ik} < 0)\ if\ y_k < 0 \\ u_k - l_k \end{cases}$$

Furthermore, the tableau needs to be pre-updated before pivoting with entering and leaving variables that does not go down to zero. For a more detailed explanation, please see Chapter 5.2 of book “Linear programming and network flows”(1).

3. Heterogeneous Implementation with OpenCL

3.1 Parallelizable parts of the algorithm

As mentioned in the introduction, Simplex method is a massively data-parallel algorithm that can take advantage of parallel accelerators such as GPU and FPGAs. There are three parts of the algorithm that can be implemented in parallel:

1. Selection of entering variables (column k)
2. Selection of leaving variables (column r)
3. Pivoting

Implementing the "pivoting" operation with parallel accelerators will accelerate the algorithm as long as the tableau is not extremely small and simple. However, the implementation of the first two parts in parallel incurs both benefits and costs. Both the first and second part is developed from an OpenCL algorithm (2) that can find the minimum/maximum of an n-length array in O (log n) time. The algorithm is discussed in details in the article (2).

This algorithm returns the minimum value of a given array. However the objective of first part (and the second part) is to find the index k giving largest cost vector cost[k]. Therefore the algorithm needs to be modified to return the index of the minimum value, rather than the minimum value itself. The original algorithm does the following:

```
int id=get_local_id();
If (scratch[id]<scratch[id+offset]) scratch[id]=scratch[id+offset];
```

In our new algorithm of finding the column k, we first introduce an array of "pointers", sequentially pointing to the array of actual data:

```
Local_index[id]=id;
```

The data values are then compared by dereferencing the pointer array. The index of the larger value, instead of the value itself, in each comparison is propagated to lower positions of the index array:

```
int this_index=local_index[id];
int other_index=local_index[id+offset];
If (scratch[this_index]<scratch[other_index]) local_index[id]=other_index;
```

Therefore the returned value is the index of the maximum value in the array.

The second part, which finds the row r that gives $\text{minimum}(\frac{b_r}{y_{rk}}: y_{rk}>0)$, is

modified from the first part by recording $\frac{b_i - bound_i}{y_{ik}}$ for local array's element scratch[i].

The benefit of using this algorithm is that it can find minimum/maximum in O(log n) time. However, the input array has to be zero padded to have length of 2^m,where m is a positive integer. Furthermore, implementing the first and second part in accelerators incurs overhead such as transferring data to and from the accelerators and formatting the data for transfer. For an input array of small size, it is actually faster to implement the first and second part using the sequential approach on CPU. The exact size by which the parallel approach will surpass the sequential approach is not yet determined as there are further complications involved in using parallel approach, which will be explained in the next section.

3.2 A host-controlled parallel computation approach

Implementing all of the three parts of computation has one further benefit: the tableau of data only needs to be written to and read from the accelerator once. During the computation the host only pass parametric variables to and from the accelerator and synchronize the parallel computation.

The general flow of the algorithm, shown as divided into host side and accelerator side implementation, is:

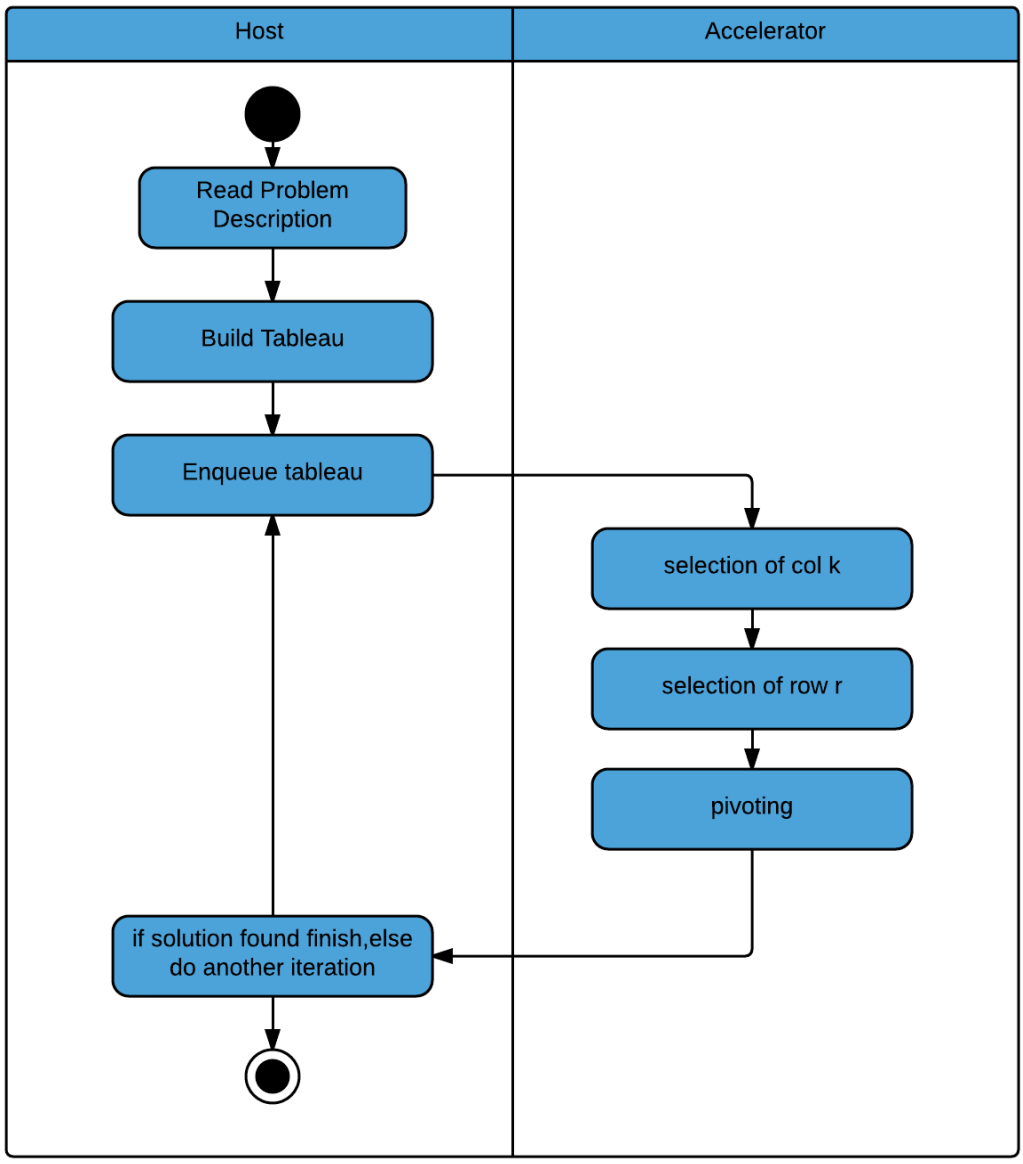

Fig.1 Flow-chart of the accelerator intensive approach

This accelerator intensive approach is ideal for problem of very large size of

which the memory access time is longer than the overhead of enqueuing two more kernels. However, the threshold size and complexity of the problem has to be determined by further experiments.

For smaller problems, the selection of column k and row r can be done on the host side, which means for each iteration the host CPU has to read the tableau from accelerator after pivoting for the next selection of k and r. This incurs additional memory access time. However when the problem is sufficiently small, the additional memory access time may be shorter than the overhead time of enqueuing two kernels, justifying the use of implementing selection of k and r on host side

6. Causes of inaccuracy in computation

One of the major causes of inaccuracy in computed result is the round-off error typically occurring in computation of very large data size. Round-off errors occur when the real value of the variable has more decimal places than what the data type can represent. The error is infinitesimal for one computation, however the error can be significant for millions steps of computation.

For simplex method, the round-off error is even more problematic because sometimes the error can be large enough to affect the selection of entering and leaving variables. This will lead the algorithm into wrong directions, causing large deviations in computed result.

The round-off error has been a more immediate problem in the previous buggy version of the solver. In the algorithm, there are conditional statements with relational conditions in the form “x<0”,”x>0” or “x=0”. However, round-off errors could make terms of zero value non-zero. Thus the conditional logic will be the opposite, therefore leading the algorithm away from the correct path.

A possible solution is to add a small error term ε into the conditions. For example when error is set as 0.0001, the condition statement will become “x<-0.0001”,”x>0.0001” and “-0.0001<x<0.0001”, accommodating the round-off errors. However the acceptable range of change in the threshold, which is “0” before, varies from problem to problem. Now the solver is using “0.0001” as the threshold. Future updates on the solver may explore ways of dynamically determining this threshold value.

7. Performance of different approaches

I have implemented four different approaches based on the variations discussed above. Two of them are less parallel approaches, with the computation of k and r occurring on host side, and pivoting on accelerator side. They are short-named GPU_host and FPGA_host as they are more host intensive methods. The other two are accelerator intensive approaches with all computations occurring on the accelerators. They are short-named GPU_device and FPGA_device.

Due to time constraints, the latter two approaches have only been implemented with a single workgroup, meaning that the maximum matrix width cannot exceed the maximum workgroup size of the accelerator. Implementing a multi-workgroup approaches requires re-writing most of the kernel codes. The multi-workgroup version will be studied and released in future updates.

The test cases used in performance testing all come from netlib(www.netlib.org/lp). Below is the performance testing result.

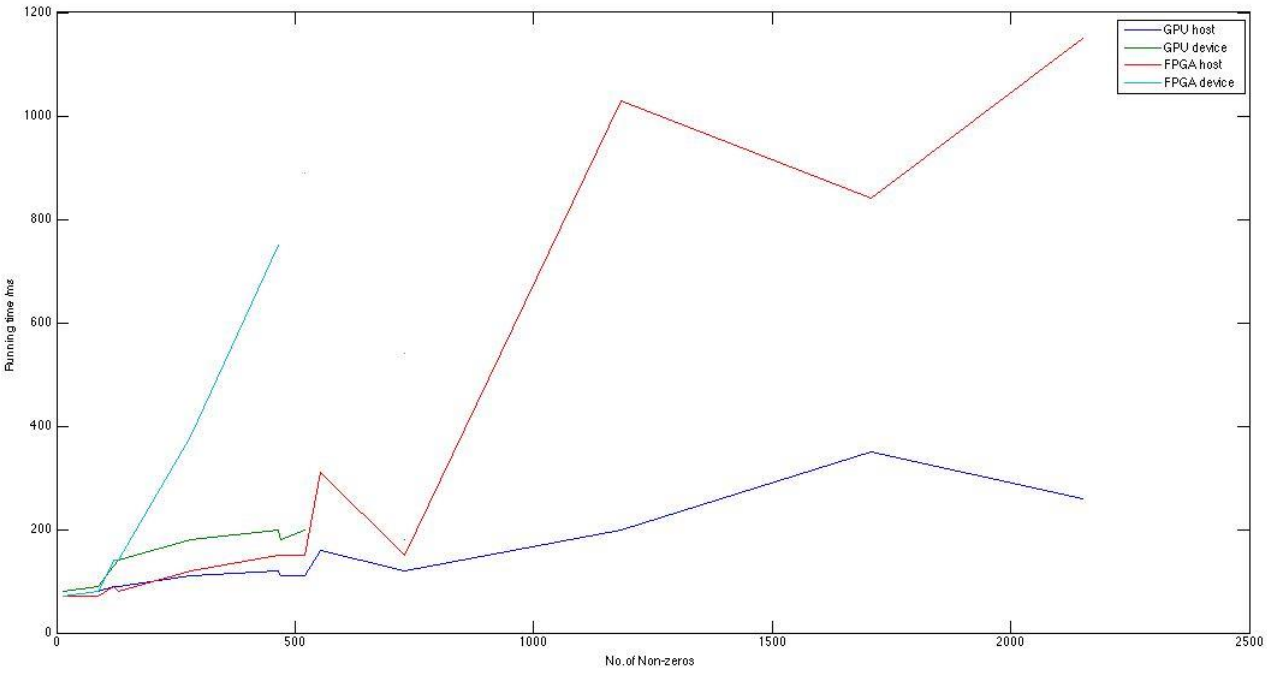

Fig.2 Performance testing result

The x-axis is the number of non-zeros present in the test cases. The y-axis is the execution CPU time in milliseconds. The performance is done with Intel i3-2130 CPU and AMD Firepro w5000 GPU and Altera Stratix V FPGA.

The performance result for accelerator-intensive approaches, as discussed above, stops at around 500 number of non-zeros because of single-workgroup limitation. However from the performance graph we can see that the

accelerator-intensive approaches are actually slower than host-intensive approaches, contrary to theoretical analysis. This is because the test cases that fits the single-workgroup limitations are too small. The overhead time of setting up two more kernels is comparatively large, making the parallel approach slower. In future version a multi-workgroup approaches will most probably show that the execution time of accelerator-intensive approaches for problems of very large size can be lower than that of host-intensive approaches.

Another observation is that approaches using GPU is generally faster than using FPGA when the test case get large. However both tests are done with the un-vectorized version of implementation for fair comparison. For the FPGA host-intensive approaches the Logic utilization is 38% and for accelerator-intensive approaches the Logic utilization is 55%. Therefore for FPGA there are rooms for increasing the number of compute units and the degree of SIMD vectorization. A maximum-performance FPGA approach will be release later.

8. Conclusion and future works

Tableau simplex method, in its nature, is a dense matrix method. This means that for most LP cases with a large number of zero entries, tableau simplex method is not an optimal solution. Sparse matrix methods have much shorter computation time for a very large LP problem. Therefore the tableau simplex method is only suitable for computation of various small LP test cases. This is exactly the motivation for this research, as a trial to solve mixed-integer programming problem by breaking up the problem into many small sub-problems requires this dense-matrix implementation.

This research is not yet complete due to time constraints. In the future, a FPGA implementation with maximum performance will be tested. However it can be said that with a problem of size smaller than the maximum workgroup size of the accelerator. The host-intensive approach with GPU yields the best result.

Reference:

(1) *"Linear Programming and network flows",* *Mokhtar S Bazaraa John J Jarvis; Hanif D Sherali Hoboken : Wiley 2011 4th ed.*

(2) *"OpenCL™ Optimization Case Study: Simple Reductions" url: http://developer.amd.com/resources/documentation-articles/articles-whitepapers/opencl-optimization-case-study-simple-reductions/*