LU Factorization with Integer Arithmetic

Yaohung Mike Tsai
Piotr Luszczek
Jack Dongarra
Innovative Computing Laboratory
University of Tennessee Knoxville

SIAM Conference on Parallel Processing for Scientific Computing (PP20)
February 12 - 15, 2020

THE UNIVERSITY OF TENNESSEE KNOXVILLE

https://icl.utk.edu/

Motivation

Integer arithmetic is available on most hardware architectures. FPGA is usually more capable in integer operations and might not have floating-point number arithmetic units. New application-specific integrated circuits (ASICs) for deep learning inference are also moving toward using mostly integer arithmetic in quantized neural networks. This motivated this study to look at the fundamental numerical linear algebra operation: Gaussian elimination (LU factorization) with partial pivoting using integer arithmetic to solve linear system Ax=b. The goal is to have a **low accuracy but fast solution and factorization** for later preconditioned iterative refinement in mixed precision algorithms.

Number Representations

Format	Range	Accuracy	Sign bit	Exponent bits	Fraction bits (Mantissa)
Double Precision (FP64)	$2e^{-308}$ to $2e^{308}$	$2^{-53} \approx 1e^{-16}$		11	52
Single Precision (FP32)	$1e^{-38}$ to $3e^{38}$	$2^{-24} \approx 6e^{-8}$		8	23
Half Precision (FP16)	$6e^{-8}$ to 65504	$2^{-53} \approx 0.0005$		5	11
BFloat16	$1e^{-38}$ to $3e^{38}$	$2^{-8} \approx 0.004$		8	8
INT32	-2^{31} to 2^{31}-1	1/2			31
INT16	-32768 to 32767	1/2			15

Related Mixed Precision Work

Haidar et al. [1] achieved 4x speed up by utilizing the half-precision tensor core from NVIDIA Volta architecture for solving linear system. The matrix multiplication operation in tensor core is accumulating in single precision so the result is better than pure half precision in previous architectures. Carson and Higham [2,3] provided error analysis for varying the precision in different part of the mixed precision algorithm as well as the condition number of the input matrix A. Higham et al. [4] proposed a method to deal with very limited range of half precision format (FP16).

HPL-AI Mixed Precision Benchmark

The HPL-AI benchmark seeks to highlight the emerging convergence of high-performance computing (HPC) and artificial intelligence (AI) workloads. While traditional HPC focused on simulation runs for modeling phenomena in physics, chemistry, biology, and so on, the mathematical models that drive these computations require, for the most part, 64-bit accuracy. On the other hand, the machine learning methods that fuel advances in AI achieve desired results at 32-bit and even lower floating-point precision formats. This lesser demand for accuracy fueled a resurgence of interest in new hardware platforms that deliver a mix of unprecedented performance levels and energy savings to achieve the classification and recognition fidelity afforded by higher-accuracy formats.

FIND OUT MORE AT
https://icl.bitbucket.io/hpl-ai/

Fixed-point Representation

The basic idea is to scale down the numbers and use a **fixed-point number representation**: $i/2^{32} \times 2^0$ where i is in 32-bit integer. The exponent won't change under addition or multiplication so can be ignored. The addition under is form is simply integer addition. Multiplication becomes: $i/2^{32} \times j/2^{32} = i \times j/2^{64} = (i \times j/2^{32})/2^{32}$. The computation of $i \times j/2^{32}$ can be done by multiplying 32-bit integers and returning the high 32 bits in the 64 bits result. Note that this operation has native instruction support on modern CPU instruction set architectures (ISAs) including x64 and ARM. Table 1 summarizes the proposed fixed-point number representation.

Storage format	i in 32 bits integer
Represented real number	$R(i) = i/2^{32} \times 2^0$
Range	$[-0.5, 0.5)$
Conversion from `double` precision number α	$i \leftarrow \texttt{int32}(\alpha \times 2^{32})$
Conversion to `double` precision number α	$\alpha \leftarrow \texttt{double}(i)/2^{32}$
Addition	$R(i) + R(j) = i/2^{32} + j/2^{32} = (i+j)/2^{32} = R(i+j)$
Multiplication	$R(i) \times R(j) = i/2^{32} \times j/2^{32} = (i \times j)/2^{64}$ $= (i \times j/2^{32})/2^{32} = R(i \times j/2^{32})$
Division	$R(i) \div R(j) = (i/2^{32}) \div (j/2^{32}) = i \div j$ $= (i \div j \times 2^{32})/2^{32} = R(i \div j \times 2^{32})$

Proposed Fixed-point Number Representation

Proposed Algorithm

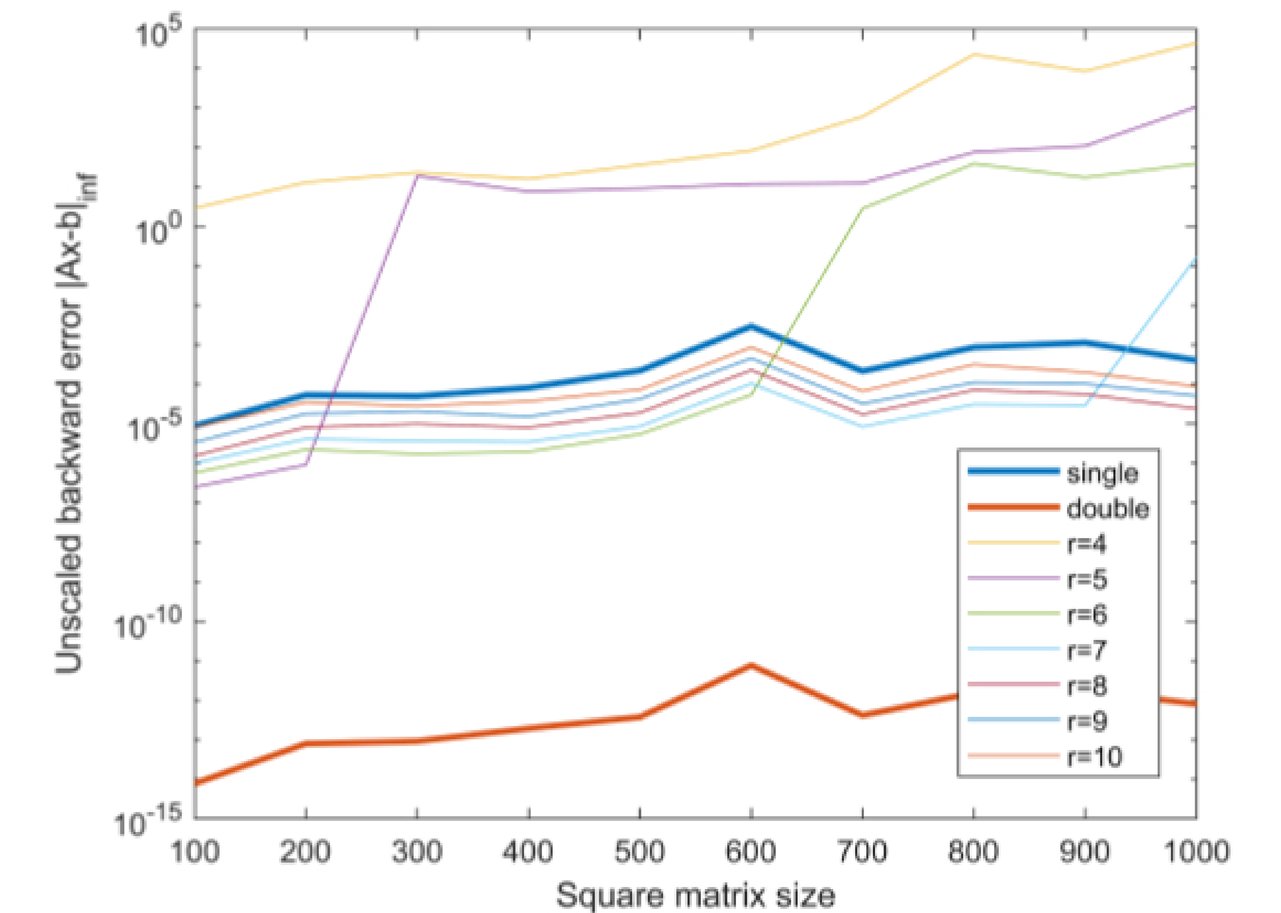
1: **Input**: n by n matrix A in double precision.
Integer r for the range while normalizing A.
2: Declare identity matrix P as permutation matrix.
3: $m \leftarrow \max(A) \times 2^r$; $A \leftarrow A/m$ ▷ Normalize A into $[-2^{-r}, 2^{-r}]$
4: $A_{int} \leftarrow \text{int32}(A \times 2^{32})$ ▷ Convert A into proposed fixed-point representation.
5: **for** $i = 1 \ldots n$ **do** ▷ Main loop over columns
6: pivot $\leftarrow (\arg\max |A_{int}[i{:}n, i]|) + i - 1$ ▷ Find the pivot index.
7: swap $(A_{int}[i,:], A_{int}[\text{pivot},:])$ ▷ Swap rows.
8: swap $(P[i,:], P[\text{pivot},:])$
9: $\alpha \leftarrow \text{int64}(2^{32})/A[i,i]$ ▷ Find the scale with integer division.
10: $A_{int}[i{:}n, i] \leftarrow \alpha A_{int}[i{:}n, i]$ ▷ Scale the column.
11: $A_{int}[i+1{:}n, i+1{:}n] \leftarrow A_{int}[i+1{:}n, i+1{:}n] - A_{int}[i+1{:}n, i] \times A_{int}[i, i+1{:}n]/2^{32}$
12: ▷ Integer rank-1 update with a division using integer shift.
13: **end for**
14: $L \leftarrow$ lower triangular part of $\text{double}(A)/2^{32}$ with unit diagonal.
15: $U \leftarrow$ upper triangular part of $\text{double}(A)/2^{32}$ including diagonal.
16: **Return**: P, L, U as the result of factorization such that $P(A/m) = LU$

Proposed LU Factorization with Integer Arithmetic

The input and output matrices of this algorithm are still in double precision to be comparable with the reference factorization from LAPACK. The input integer **r** determines how many bits we are actually using **(32-r)** while converting **A** into 32-bit integer. Because the matrix would grow during the factorization and we do not dynamically scale, it might hit the range and overflow at some point. To avoid it, we first scale the matrix into **$[-2^{-r}, 2^{-r}]$**. The higher **r** is, the more room we will have from the range. But less accurate the input matrix would be after being converted into `int32`.

The computation inside the loop is mainly 32-bit integer arithmetic. Line 9 requires 64-bit integer division but only once per column. The scaling at line 10 will remain in `int32` range because the pivot has larger magnitude then other elements in the column. The update in line 11 is 32-bit integer multiply but we only need the high 32 bits in 64 bits result. Other than mentioned lines, the algorithm mimics the standard LU factorization with partial pivoting. Partial pivoting also controls the growth rate to be at most 2. Otherwise the overflow could happen easily.

Numerical Results



The figure shows the unscaled backward error $|\mathbf{Ax\text{-}b}|_{inf}$ vs. input matrix size. The algorithm is implemented in MATLAB R2018b. Each element of the matrix is generated from uniform random distribution: uniform(-1,1). The results from single and double precision LU factorization are also reported as reference. While **r≤7**, overflow occurs and the algorithm failed. The higher r is, the smaller range input matrix A will be normalized into. There will be more room from overflowing so it could work with larger matrix. However, the backward error grows with r since the input is truncated more. Nevertheless, when **r=10** it is still using 32-10=22 bits and the result is still better than single precision which is using 23 mantissa bits.

Conclusion and Future Work

We have demonstrated that it is feasible to use 32-bit integer arithmetic perform LU factorization with partial pivoting and achieve better accuracy than single precision. The main issue for this approach is the possibility of overflow during factorization. To tackle the problem, we would like to continue the research on following directions:

- **Dynamically scale the column during factorization.** While finding the pivot, if it's too close to overflow, we can further scale down the column and remaining unfactored matrix. It can also be done in paneled factorization and do the check once for each panel.
- **Other representation formats.** There are other less common number representation like blocked floating-point numbers. Although they might not have native hardware support, they could fit our need better in the algorithm.
- **Error analysis.** We would like to perform error analysis to have a better understand about behavior of algorithm and incorporate the findings to improve the algorithm.
- **Smaller datatype including `int16` and `int8`.** The goal of this project is to find a fast solver using smaller datatype. The computational complexity of factorization is $O(n^3)$ while the later iterative refinement is $O(n^2)$. So the factorization is critical to overall performance of mixed-precision solver. Shorter datatype might give us another chance for speedup while the desired accuracy can still be obtained with preconditioned GMRES or other refinement schemes.

REFERENCES

[1] Haidar, A., Tomov, S., Dongarra, J., & Higham, N. J. (2018, November). Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis* (pp. 603-613). IEEE.

[2] Carson, E., & Higham, N. J. (2017). A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems. *SIAM Journal on Scientific Computing*, *39*(6), A2834-A2856.

[3] Carson, E., & Higham, N. J. (2018). Accelerating the solution of linear systems by iterative refinement in three precisions. *SIAM Journal on Scientific Computing*, *40*(2), A817-A847.

[4] Higham, N. J., Pranesh, S., & Zounon, M. (2019). Squeezing a matrix into half precision, with an application to solving linear systems. *SIAM Journal on Scientific Computing*, *41*(4), A2536-A2551.

This research was supported by the Exascale Computing Project (17-SC-20-SC), a collaborative effort of the U.S. Department of Energy Office of Science and the National Nuclear Security Administration, under prime contract #DE-AC05-00OR22725, and UT Battelle subaward #4000152412