Towards Numerical Benchmark for Half-Precision Floating Point Arithmetic

Piotr Luszczek
University of Tennessee

Jakub Kurzak
University of Tennessee

Ichitaro Yamazaki
University of Tennessee

Jack Dongarra
University of Tennessee
Oak Ridge National Laboratory
Manchester University

***Abstract*—With NVIDA Tegra Jetson X1 and Pascal P100 GPUs, NVIDIA introduced hardware-based computation on FP16 numbers also called half-precision arithmetic. In this talk, we will introduce the steps required to build a viable benchmark for this new arithmetic format. This will include the connections to established IEEE floating point standards and existing HPC benchmarks. The discussion will focus on performance and numerical stability issues that are important for this kind of benchmarking and how they relate to NVIDIA platforms.**

I. Introduction

Iterative refinement has been known for many decades as an effective tool for increasing accuracy of a solution of a set of simultaneous linear equations [1], [2], [3]. The requirement for improvement is accumulation of the residual $b - Ax$ in higher-precision arithmetic which is often possible through already available hardware or, absent that, software-based techniques.

If it is possible to customize the working precision, as is the case on FPGAs, then this scheme can be exploited by creating a custom floating-point units [4]. Alternatively, it is possible to exploit system matrix conditioning and use iterative refinement as a method of accessing faster hardware capabilities [5]. This allows a solver to run at the speed of lower-precision arithmetic and deliver working-precision answer. This may deliver two-fold speed up on majority of hardware platforms and occasionally as much as 10-fold increase on less common systems. Modern GPUs may offer an even greater benefit as some consumer grade gaming cards feature only a rudimentary support for double precision arithmetic that ends being 32 times slower than its single precision counter part.

The current state of silicon chip market, often referred as post-Dennard and post-Moore era, precipitated emergence of accelerators and specialized compute options such as FPGAs, SOCs, and TPUs. In the software layer this resulted in codesign efforts with representative mini-applications to drive the effort [6]. A regular linear system solve with a dense matrix and the optimal algorithm no longer looks uniform granted these hardware advances. Therefore, there is a need for more nuanced look and specialization of the algorithmic choices which we attempt in this writing.

II. Background

While solving a system of linear equations of the form:

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

where $A \in \mathbb{R}^{n \times n}$ and $x, b \in \mathbb{R}^n$. Taking advantage of the structure of A in Eq. (1) is commonly used in computational science. For dense matrices, switching between LU, Cholesky [7], and QR factorizations is an option, which depends on the numerical properties of the matrices. Further exploitation of rank structure of A may take form of $\mathcal{H}$-matrices [8], [9], [10], $\mathcal{H}^2$-matrices [11], [12], [13], quasi-separable matrices [14], [15], semiseparable matrices [16], [17], and multilevel low-rank structures [18]. Lacking such structure and not being able to exploit low-rank properties, a matrix may still offer an opportunity for faster solver if the condition number is low enough and the speed of lower-precision may be utilized by the available hardware. The exploitation of the condition number of the L and U factors to tackle least squares problems can also be taken advantage of as long as regularization is involved [19].

TABLE I
Summary of IEEE 754 (2008) floating point.

Precision	Mantissa	Exponent	Epsilon	Max
Quadruple	112	15	$O(10^{-34})$	$O(10^{4932})$
Extended	64	15	$O(10^{-19})$	$O(10^{308})$
Double	52	11	$O(10^{-16})$	$O(10^{308})$
Single	23	8	$O(10^{-7})$	$O(10^{38})$
Half†	10	5	$O(10^{-3})$	65504

† defined only for storage

Table I shows IEEE floating point standard formats that are currently supported by the hardware available on the market.

III. Iterative Algorithm and Its Numerical Analysis Properties

Note that LU contains multiple stages that can be targeted by appropriate precision based on the final precision bounds on the numerical properties. The pivoting in the LU factorization maintains L well conditioned (to the extent possible) and pushes the unbound pivot growth into U. Implementations commonly push this one step farther and produce U that is singular if A is singular in exact arithmetic, i.e., $\kappa(A) = \infty$, or only singular in the working precision: $\kappa\left(A^{(wp)}\right) = \texttt{Inf}$. In the least squares minimization context, note that it might be more preferable to numerically find $\min \|b - Ly\|_2^2$ rather than more computationally involved $\min \|b - Ay\|_2^2$ if the L factor is better conditioned than the original system matrix A [20]. The

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

Algorithm 1: Mixed precision iterative refinement with the 16-bit factorization and 64-bit corrections.

1 $L^{(16)}, U^{(16)}, P \leftarrow \mathsf{lu}\left(A^{(16)}\right)$ $\quad O\left(n^3\right) \times 16$ bits
2 $v^{(16)} \leftarrow P \backslash b^{(16)}$ $\quad O\left(n^2\right) \times 16$ bits
3 $y^{(16)} \leftarrow L^{(16)} \backslash v^{(16)}$ $\quad O\left(n^2\right) \times 16$ bits
4 $x_0^{(16)} \leftarrow U^{(16)} \backslash y^{(16)}$ $\quad O\left(n^2\right) \times 16$ bits
5 **for** $k = 1, 2, \ldots$ **do**
6 $\quad r_k^{(64)} \leftarrow b^{(64)} - A^{(64)} x_{k-1}^{(64)}$ $\quad O\left(n^2\right) \times \boxed{64}$ bits
7 $\quad s_k^{(16)} \leftarrow P \backslash r_k^{(16)}$ $\quad O\left(n^2\right) \times 16$ bits
8 $\quad t_k^{(16)} \leftarrow L^{(16)} \backslash s_k^{(16)}$ $\quad O\left(n^2\right) \times 16$ bits
9 $\quad z_k^{(16)} \leftarrow U^{(16)} \backslash t_k^{(16)}$ $\quad O\left(n^2\right) \times 16$ bits
10 $\quad x_k^{(64)} \leftarrow x_{k-1}^{(64)} + z_k^{(64)}$ $\quad O(n) \times \boxed{64}$ bits

conditioning of the system matrix A depends on the originating application but in a benchmarking context, $\kappa(A)$ can be a controlled quantity. A known results states that $\kappa(A) \approx \sqrt{n}$ if the entries of A are normally distributed around zero: $x_{ij} = \mathcal{N}(0, 1)$ [21] and this can be guaranteed in practice through the correct choice of the pseudo random number generator (PRNG) for the system matrix A. In fact, the High Performance LINPACK benchmark uses a uniform distribution around zero: $\mathcal{U}\left(0, \frac{1}{2}\right)$ [22] which is sufficient in practice to bound the condition number at reasonable level while still force excessive pivot growth when at least partial pivoting is not used by a non-conforming implementation.

The Algorithm 1 shows an iterative refinement adopted from the 32/64 bit formulation [23], [24], [25] to the 16/64 scenario. The algorithm solves the system from Eq. (1) by introducing representation of matrices and vectors in fixed-precision arithmetic:

$$A^{(64)} x^{(64)} = b^{(64)} \tag{2}$$

where $A^{(64)}$, $b^{(64)}$, $x^{(64)}$ represent A, b, x of Eq. (1) but in 64-bit precision floating-point arithmetic. To describe the application of an inverse of a matrix we use the *backslash* notation with the symbol "\" borrowed from the programming systems for computational algebra such as Julia, MATLAB, Octave, and SciLab. While in infinite precision arithmetic $x \leftarrow A^{-1} b$ is equivalent to $x \leftarrow A \backslash b$, in finite precision floating-point representation the latter is more accurate because it does not explicitly form A^{-1} which results in larger round-off error than the decompositional approach of triangular factorization $PA = LU$ followed by triangular system solves with L and U, respectively. And with our triangular factors represented in the much lower precision, minimizing the round-off errors is essential.

Note that only line 1 of Algorithm 1 has complexity $O\left(n^3\right)$ and this is the key to taking advantage of the available hardware and performing this step with the fastest method. In our case, this is done through the use of 16-bit floating-point arithmetic that is the fastest on the hardware where it is available. To be more precise, with higher-order terms only, the time to solution may be stated analytically in the form:

$$t(n) \propto \left(\frac{2}{3} n^3 + O\left(n^2\right)\right) \times \tau^{(16)} + O\left(n^2\right) \times \tau^{(64)} \tag{3}$$

where $\tau^{(16)}$ and $\tau^{(64)}$ are the products of throughput rates of the execution units and the clock cycle length for 16- and 64-bit arithmetic, respectively. On most of the hardware platforms currently available on the market the ratio of the two quantities is 4: $\frac{\tau^{(16)}}{\tau^{(64)}} = 4$. However, due to the importance of 16-bit arithmetic, this is projected to be much higher. For example, the announced NVIDIA Volta GPU accelerators will have this ratio as high as 10 which makes this hardware so much more important from the perspective of raw performance and performance-per-watt perspectives. Furthermore, the Volta cards are announced to include 16-bit arithmetic with 32-bit accumulation – a critical feature for numerical issues surrounding limited precision formats. This kind of intermediate increase in precision is already used in hardware-based implementations of the *fused-multiply-add* (FMA) instruction across almost all commercially available computing platforms as speicified by the IEEE 754 floating point standard [26]. This intermediate accumulation at higher precision may be used in in handling some of the quantities of the algorithims, for example, $r_k^{(64)}$ and $r_k^{(16)}$ as well as $r_k^{(64)}$ and $r_k^{(16)}$.

The iterative process from Algorithm 1 is called a Richardson iteration and due to round-off error may be considered a form of Newton method. The iteration count in line 5 is left unspecified and can be established experimentally or through the analysis of the Newton's algorithm that is known to double the number of significant digits in the solution as long as the initial guess x_0 is in the *basin of quadratic convergence* [27]. Consequently, we conducted relevant experiments to establish numerically the convergence properties with respect to three precisions and matrix sizes. Figure 1 shows the resulting convergence charts with the iteration count on the horizontal axis and the residual 2-norm $\|b - Ax\|_\infty$ on the vertical axis. The figure clearly indicates that convergence rate for 16-bit starting vector and the L/U factors is much slower than the corresponding rates for the 32-bit format. This cannot be explained by the condition number alone that we keep constant for a given matrix size n and small enough to guarantee existence of non-singular L and U factors when partial pivoting is used during the factorization. Instead, the convergence is slowed down by the underflow rounding due to limited representation range of the half-precision numbers especially when residual norm becomes small. Figure 2 shows extension of Figure 1 for the matrix when the rounding causes the iteration count to increase above 60.

IV. Implementation Notes

The code implementing algorithms in this paper relies on the support of various floating-point precisions in the compiler. The half precision is not supported in the C/C++ family of

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

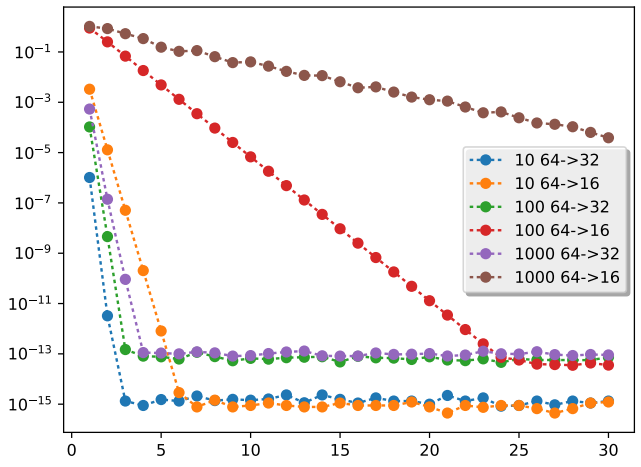

Fig. 1. Convergence rate of iterative refinement for various matrix sizes and precisions. The horizontal axis indicates iteration step and the vertical axis shows $\|b - Ax\|_\infty$.

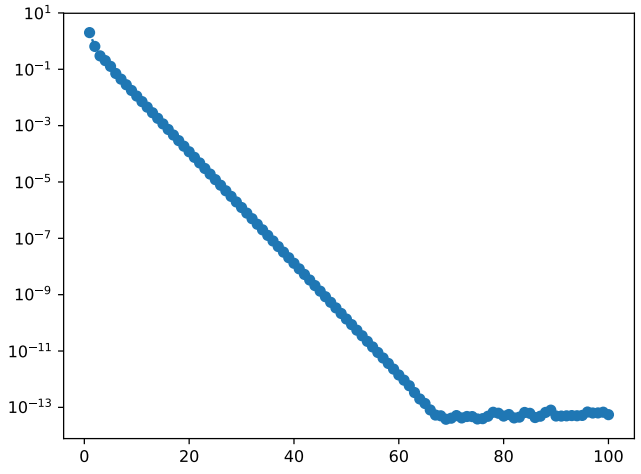

Fig. 2. Convergence rate of iterative refinement for $n = 1000$ and 16/64 refinement precision. The horizontal axis indicates iteration step and the vertical axis shows $\|b - Ax\|_\infty$.

languages unlike the 80/128-, 64-, and 32-bit formats as long double, double, float, respectively. Hence, while programming with the CUDA toolkit the cuda_fp16.h header is required to make the __half and vectorized __half2 data types available with the utility functions. Note, however, that they are only available in the device code and cause compilation error in the host code. A portable implementation could use the standard C library functions frexp(), frexpl(), and frexpf() to provide a fall-back for the half-precision data type in software including the compute and the necessary conversion routines. Open source implementations of half-precision arithmetic and conversions are also available in some packages including Eigen and OpenVZ.

In Fortran, half-precision data type has been available real*2 in addition to the more familiar real*4, real*8, and real*16. This support continues in the modern compilers. In fact, it is the real*2 that made the half-precision data type available in the MPI standard that had to be able to not just communicate all Fortran data types but also compute on the reductions that require support for all the standard arithmetic operators as well as the Fortran's intrinsic functions.

On the CPU side, the support for half-precision data type is very limited because most of the processor offer instructions only for conversion between 32-bit and 16-bit floating-point data types. ARM NEON VFP specification includes conversions in version 8.2-A as vld1_f16, vst1_f16, and vcvt_f16_f32 instructions. The x86 instruction set includes scalar and vector instructions for conversions to/from the 32-bit floating-point formats. They are available through the intrinsic functions declared in the x86intrin.h header file and include scalar _mm_cvtph_ps() and _mm_cvtps_ph() as well as vector _mm256_cvtph_ps() and _mm256_cvtps_ph().

The NVIDIA tool chain generates generates PTX instructions for the half-precision instructions as cvt.f16.* for conversions and fma.f16x2 vector FMA computation.

At the higher abstraction level, programming environments include support for half-precision. In Julia programming language, Float16 data type is available and many of the builtin libraries support computation on this type including dense linear algebra routines relevant to this paper. In Python's numerical extension package numpy, float16 data type is available and many functions in that package work as expected with this type. However, the dense linear algebra subpackage numpy.linalg does not support the objects of this type because it calls to LAPACK routines that do not have half-precision versions.

V. Performance Results

Algorithm 2 shows a blocked implementation of the LU factorization that is the basis of the code used in the LAPACK, ScaLAPACK, and MAGMA libraries. Given this formulation, there are multiple ways to get at high performance rates on NVIDIA accelerators for computations required for the LU factorization. All of the *Schur's complement* operations combined of the form $A_{2,2} \leftarrow A_{2,1} \times A_{1,1}^{-1} \times A_{1,2}$ require $O\left(n^3\right)$ operations are performed. It can accomplished with the following calls to the NVIDIA `cuBlas` library:

- cublasHgemm() – half-precision matrix-matrix multiply,
- cublasHgemmBatched() – half-precision batched matrix-matrix multiply,
- cublasHgemmStridedBatched – half-precision batched matrix-matrix multiply with a stride,
- cublasXgemm() – type-generic matrix-matrix multiply.

Based on our tests, the first one: cublasHgemm(), gave the based performance across a wide range of matrix sizes and block sizes for 16-bit precision. Next we performed an autotuning sweep across blocking sizes n_b and precisions. Figure 3 shows the results of all of these automating runs

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

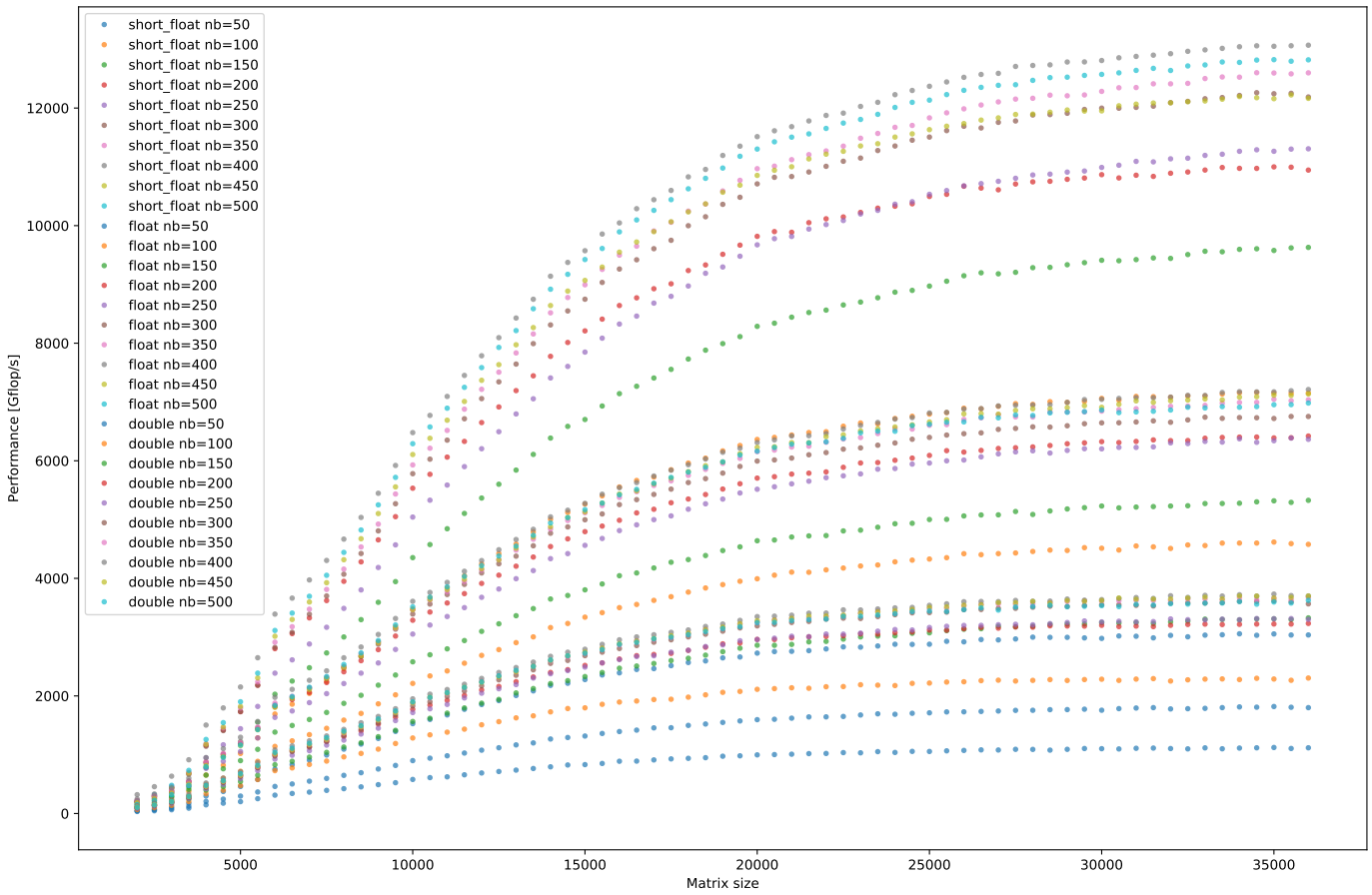

Fig. 3. Performance of the LU factorization in three precisions with for all block sizes using cublasHgemm().

Algorithm 2: Blocked implementation of the LU factorization in terms of Level 3 BLAS calls to trsm() and gemm().

$P \leftarrow I$ // start with identity permutation
for i **to** $n - n_b$ **by** n_b **do**
  // 2-by-2 partition
  $\begin{bmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{bmatrix} \equiv$
  $\begin{bmatrix} A_{i\ldots i+n_b,i\ldots i+n_b} & A_{i\ldots i+n_b,i\ldots n} \\ A_{i+n_b\ldots n,i\ldots i+n_b} & A_{i+n_b\ldots n,i+n_b\ldots n} \end{bmatrix}$
  getf2($[A_{1,1}A_{2,1}]^T, P_{i\ldots n,i\ldots n}$) // panel LU
  if $i \geq n - n_b$ **then**
    break
  laswp($[A_{1,2}A_{2,2}]^T, P_{i\ldots n,i\ldots n}$) // apply pivot
  trsm($A_{1,1}, A_{1,2}$) // triangular solve
  gemm($A_{2,1}A_{1,2}A_{2,2}$) // Schur's complement
return A, P

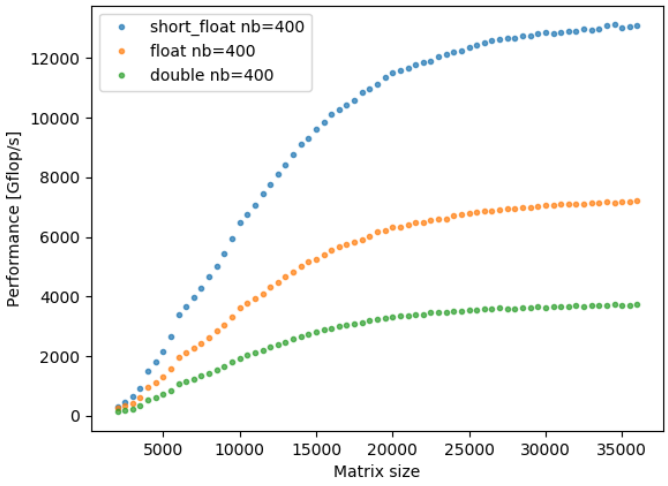

Fig. 4. Performance of the LU in three precisions factorization with optimal block size $n_b = 400$ using cublasHgemm().

and Figure 4 shows the optimal performance for the three precisions when $n_b = 400$. The maximum performance reached was approximately 3900 Gflop/s, 7550 Gflop/s, and 13760 Gflop/s for double precision, single precision and half precision, respectively. This corresponds to 96% efficiency of going from double to single precision and 88% efficiency

978-1-5386-3472-1/17/$31.00 ©2017 IEEE

when going from double to half precision. These numbers are expected to improve with newer version of NVIDIA CUDA toolkit, cuBlas library, and our own autotuned kernels.

VI. Conclusions and Future Work

We have shown in this paper the numerical and implementation building blocks of benchmarking procedures capable of assessing the potential of the half-precision data type on modern accelerator hardware.

Our future will focus on the implementation details on a Jetson X1 board that takes advantage of the specialized shared memory available in the system and the ARM processor optimizations with respect to the routines that can be performed on the host. Similarly, an implementation targeting IBM Minsky platform is of great interest with support for both the Pascal card and the IBM POWER8 processor directly connected with the NVLink interface for much higher bandwidth and lower latency when compared to GPU cards connected through the PCIexpress bus.

In a more distant future, we would like to focus on NVIDIA Volta and its new instruction set additions referred to as Tensor Core which give an additional performance advantage to the half-precision performance.

Another interesting extension is to see the potential of the half precision complex variant of the code that can achieve even higher performance levels due to the higher floating-point intensity.

Acknowledgements

This research was supported by the National Science Foundation through Award 1439052. Also, 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.

References

[1] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*. Princeton, NJ, USA: Prentice-Hall, 1963.

[2] ——, *The Algebraic Eigenvalue Problem*. London, UK: Oxford University Press, 1965.

[3] G. Peters and J. H. Wilkinson, "On the stability of Gauss-Jordan elimination with pivoting," *Communications of the ACM*, vol. 18, pp. 20–24, 1975.

[4] J. Lee, G. D. Peterson, R. J. Harrison, and R. J. Hinde, "Mixed precision dense linear system solvers for high performance reconfigurable computing," ser. 2009 Symposium on Application Accelerators in High-Performance Computing (SAAHPC'09), University of Illinois at Urbana-Champaign, USA, 2009.

[5] A. Buttari, J. Dongarra, J. Langou, J. Langou, P. Luszczek, and J. Kurzak, "Mixed precision iterative refinement techniques for the solution of dense linear systems," Manchester Institute for Mathematical Sciences, School of Mathematics, The University of Manchester, Tech. Rep. MIMS EPrint: 2007.124, 2007, iSSN 1749-9097, Reports available from: http://www.manchester.ac.uk/mims/eprints.

[6] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, September 2009.

[7] M. A. Saunders, "Major Cholesky would feel proud," *ORSA Journal on Computing*, vol. 6, pp. 23–27, 1994.

[8] W. Hackbusch, "A sparse matrix arithmetic based on H-matrices. Part I: Introduction to H-matrices," *Computing*, vol. 62, pp. 89–108, 1999.

[9] W. Hackbusch and B. N. Khoromskij, "A sparse H-matrix arithmetic. Part-II: Application to multi-dimensional problems," *Computing*, vol. 64, pp. 21–47, 2000.

[10] W. Hackbusch, L. Grasedyck, and S. Börm, "An introduction to hierarchical matrices," *Math. Bohem.*, vol. 127, pp. 229–241, 2002.

[11] S. Börm, L. Grasedyck, and W. Hackbusch, "Introduction to hierarchical matrices with applications," *Eng. Anal. Bound. Elem.*, vol. 27, pp. 405–422, 2003.

[12] S. Börm and W. Hackbusch, "Data-sparse approximation by adaptive H2-matrices," Max Planck Institute for Mathematics, Leipzig, Germany, Tech. Rep. 86, 2001.

[13] W. Hackbusch, B. N. Khoromskij, and S. A. Sauter, "On H2-matrices," *Lectures on Applied Mathematics*, vol. (Munich, 1999), pp. 9–29, 2000.

[14] T. Bella, Y. Eidelman, I. Gohberg, and V. Olshevsky, "Computations with quasiseparable polynomials and matrices," *Theoret. Comput. Sci.*, vol. 409, pp. 158–179, 2008.

[15] Y. Eidelman and I. Gohberg, "On a new class of structured matrices," *Integr. Equat. Operat. Theor.*, vol. 34, pp. 293–324, 1999.

[16] S. Chandrasekaran, P. Dewilde, M. Gu, T. Pals, X. Sun, A.-J. van der Veen, and D. White, "Some fast algorithms for sequentially semiseparable representations," *SIAM J. Matrix Anal. Appl.*, vol. 27, pp. 341–364, 2005.

[17] R. Vandebril, M. V. Barel, G. Golub, and N. Mastronardi, "A bibliography on semiseparable matrices," *Calcolo*, vol. 42, pp. 249–270, 2005.

[18] R. Li and Y. Saad, "Divide and conquer low-rank preconditioners for symmetric matrices," *SIAM J. Sci. Comput.*, vol. 35, no. 4, 2013, a2069A2095.

[19] G. W. Howell and M. Baboulin, "LU preconditioning for overdetermined sparse least squares problems," in *Proceedings of Parallel Processing and Applied Mathmetics (PPAM) 2015*, Lublin, Poland, 2015, to appear.

[20] G. Peters and J. H. Wilkinson, "The least-squares problem and pseudo-inverses," *Comput. J.*, vol. 13, pp. 309–316, 1970.

[21] A. Edelman, "Eigenvalues and condition numbers of random matrices," Ph.D. dissertation, Massachusetts Institute of Technology, May 1989.

[22] J. J. Dongarra, P. Luszczek, and A. Petitet, "The LINPACK benchmark: Past, present, and future," *Concurrency and Computation: Practice and Experience*, vol. 15, no. 9, pp. 803–820, August 10 2003, dOI: 10.1002/cpe.728.

[23] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra, "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy (revisiting iterative refinement for linear systems)," in *ACM/IEEE SC 2006 Conference (SC'06)*, Nov. 2006, p. 50.

[24] M. Baboulin, A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, "Accelerating scientific computations with mixed precision algorithms," *Computer Physics Communications*, vol. 180, pp. 2526–2533, 2009.

[25] A. Buttari, J. Dongarra, J. Kurzak, J. Langou, J. Langou, P. Luszczek, and S. Tomov, *High Performance Computing and Grids in Action*. IOS Press, Amsterdam, Nov. 2007, ch. Exploiting Mixed Precision Floating Point Hardware in Scientific Computations.

[26] A. S. 754-1985, "Standard for binary floating point arithmetic," Institute of Electrical and Electronics Engineers, Tech. Rep., 1985.

[27] R. D. Neidinger, "A classroom note: Newton's method doubles digit improvement," *Mathematics and Computer Education*, vol. 48, no. 1, January 1 2014.