Error-Explicit Low-Precision Factorizations for Iterative Refinement

Joseph McLaughlin
Department of Computer Science
University of Oregon
Eugene, Oregon, USA
jmclaug2@uoregon.edu

Jee Whan Choi
Department of Computer Science
University of Oregon
Eugene, Oregon, USA
jeec@uoregon.edu

***Abstract*—Recent NVIDIA GPUs have prioritized low-precision throughput while native FP64 performance has stagnated or declined, with peak FP64 throughput falling from 37 TFLOP/s on B200 to 1.25 TFLOP/s on B300. Mixed-precision iterative refinement (MPIR) exploits this imbalance by factorizing in FP32, but still retains the FP64 matrix A to evaluate residuals, requiring $12n^2$ bytes of persistent storage. We present two new refinement algorithms that remove this dependence on native FP64 matrix operations. Split-A iterative refinement (SA-IR) represents A as two FP32 words and evaluates residual products on low-precision units. Error-explicit iterative refinement (R-IR) instead stores the FP32 factorization residual $R = PA - LU$ and discards A after setup, reducing persistent storage to $8n^2$ bytes. We show that R-IR inherits MPIR's first-order convergence condition and that, in the tested configurations, its accuracy is limited by residual-evaluation precision rather than FP32 storage of R. Across five GPUs with peak FP32:FP64 ratios ranging from 2 to 64, both methods outperform native and emulated FP64 on GPUs with large ratios. On B300 at $n = 32{,}768$, SA-IR is $25\times$ faster than emulated FP64 and $1.4\times$ faster than vendor refinement for a single right-hand side, while R-IR is $24\times$ faster than vendor refinement for 8192 right-hand sides. A streamed construction of R gives a peak footprint of $8.24n^2$–$8.57n^2$ bytes, enabling solves to $n = 184{,}320$ in 287 GB where $12n^2$ exceeds device memory.**

I. Introduction

The rapid growth of AI has reshaped data-center GPU architectures around high-throughput low-precision arithmetic. While computational throughput for AI-oriented low-precision formats has increased rapidly across recent GPU generations, native FP64 performance has largely stagnated and, in some architectures, even decreased. For example, NVIDIA's Blackwell Ultra B300 provides only 1.25 TFLOP/s of native FP64 throughput, a substantial decrease relative to earlier data-center GPUs. Although the newer Rubin architecture restores native FP64 throughput to 33 TFLOP/s, this is slightly lower than the 34 TFLOP/s provided by the Hopper H100 when it was introduced in 2022 [1]. This stagnation in FP64 performance, contrasted with the rapid growth of low-precision throughput, creates an opportunity to redesign scientific algorithms to exploit increasingly abundant low-precision hardware while retaining FP64-class accuracy.

In this work, we explore this opportunity for dense linear solves through mixed-precision iterative refinement, using LU factorization as a first case study. We introduce two complementary approaches. *Split-A Iterative Refinement (SA-IR)* represents the FP64 matrix using two FP32 components, allowing computationally expensive operations to exploit high-throughput low-precision units while retaining FP64-class accuracy. *Error-Explicit Iterative Refinement (R-IR)* instead retains the factorization residual $R = PA - LU$ in FP32, eliminating the need to preserve the original high-precision matrix and reducing persistent operator storage from $12n^2$ to $8n^2$ bytes, at the cost of a one-time construction of R. Together, *SA-IR* and *R-IR* exploit the widening gap between low-precision and FP64 throughput to improve performance, while *R-IR* additionally reduces the persistent memory footprint to alleviate limited device-memory capacity. Three factors determine which method to use. Hardware determines whether low-precision refinement is worthwhile at all, memory determines which representations fit, and factorization reuse determines whether constructing R repays its cost.

Conventional MPIR nevertheless retains the FP64 matrix A for residual evaluation, requiring $12n^2$ bytes of persistent operator storage and repeated high-precision access to A. Our methods target these remaining computation and storage costs. *The key contributions of this work are:*

1) **Split-A Iterative Refinement (SA-IR).** SA-IR represents A using two FP32 words and evaluates high-accuracy residual products using low-precision hardware, eliminating native FP64 matrix operations during refinement.
2) **Error-Explicit Iterative Refinement (R-IR).** R-IR stores the FP32 factorization residual $R = PA - LU$ instead of A, reducing persistent storage from $12n^2$ to $8n^2$ bytes; a streamed construction enables problems beyond the capacity of A-retaining MPIR.
3) **Numerical characterization of R-IR.** We show that R-IR inherits MPIR's first-order convergence condition, characterize its finite-precision error sources, and demonstrate that FP32 storage of R does not limit accuracy in the tested configurations.
4) **Numerical and performance evaluation.** Across five GPUs and single- and multiple-right-hand-side workloads, we characterize accuracy, performance, and memory usage against native FP64, FP64 emulation, and vendor refinement, reaching $53.5\times$ and $20.5\times$ speedups

for SA-IR and R-IR at $n = 98{,}304$ on B300.

II. Background and Related Work

A. Mixed-Precision Iterative Refinement

Iterative refinement improves the solution of a linear system $Ax = b$ by repeatedly computing a residual and applying a correction using an existing factorization [2], [3]. Mixed-precision iterative refinement (MPIR) reduces the cost of this process by computing the factorization at lower precision while evaluating the residual at higher precision, thereby recovering FP64-class accuracy without requiring an FP64 factorization [4], [5]. GPU implementations further accelerate the factorization using low-precision tensor cores [6], [7], and MPIR is now available in vendor libraries and widely used mixed-precision benchmarks [8], [9].

For an FP32 LU factorization with partial pivoting,

$$PA_{32} \approx LU, \qquad A_{32} = \mathrm{fl}_{32}(A). \tag{1}$$

An initial solution x_0 is obtained using the FP32 factors, after which classical correction-form refinement repeatedly evaluates, for $m = 0, 1, \cdots$,

$$r_m = b - Ax_m, \qquad x_{m+1} = x_m + U^{-1}L^{-1}Pr_m. \tag{2}$$

Under standard assumptions, the convergence of FP32-factorized MPIR is governed to first order by the relationship between the condition number of A and the FP32 unit roundoff, commonly expressed as $\kappa(A)u_{32} \ll 1$ for rapid convergence [5], [10], [11].

Although MPIR avoids an FP64 factorization, conventional implementations still retain the original high-precision matrix A to evaluate the residual. For an $n \times n$ dense system, the FP32 factors require $4n^2$ bytes, while the retained FP64 matrix requires another $8n^2$ bytes, yielding a persistent operator footprint of $12n^2$ bytes. The FP64 representation of A is therefore both the largest persistent allocation and the high-precision operator repeatedly accessed during refinement. Existing MPIR methods and analyses generally assume that the high-precision operator remains available during refinement [4]–[7], [11], [12]. R-IR revisits this assumption by asking whether the original matrix must remain resident after factorization.

B. High Accuracy from Low-Precision Arithmetic

A higher-precision quantity can be represented and manipulated using multiple lower-precision components. Floating-point splitting and error-free transformations provide the classical foundation for such representations [13], [14]. The Ozaki scheme extends this principle to matrix multiplication by decomposing operands into multiple narrow low-precision slices, evaluating the resulting products using high-throughput matrix operations, and combining them to recover substantially higher accuracy [15].

Recent work has mapped these techniques to FP16, TF32, and integer tensor-core operations [16]–[19], enabling high-accuracy matrix multiplication without relying on native FP64 arithmetic. Related techniques are now incorporated into NVIDIA's floating-point emulation support in cuBLAS and cuSOLVER [20], [21]. The robustness and required number of slices for general operands remain active areas of research [22], [23].

We use these established techniques as an arithmetic substrate rather than proposing a new splitting or emulation scheme. SA-IR applies a two-word FP32 representation to the high-accuracy operations required by MPIR, whereas R-IR uses high-accuracy low-precision products to construct its error-explicit representation and evaluate the final residual.

C. Relation to Prior Work

Existing dense LU-based MPIR methods and analyses generally retain or repeatedly access the high-precision operator A for residual evaluation [4]–[7], [11], [12]. To our knowledge, prior work has not replaced this persistent operator with the precision-matched factorization residual $R = PA - LU$ and used the resulting representation to discard A. Closest in spirit, Higham and Mary [24] treat the error of a low-accuracy LU factorization as a numerical object and exploit low-rank approximations of it as a preconditioner. They do not materialize the factorization error at full density or use it as a replacement for the original operator. The splitting iteration underlying R-IR is classical; our contribution is the persistent error-explicit representation, its finite-precision analysis, and a high-performance GPU implementation.

III. Split-A Iterative Refinement

In mixed-precision iterative refinement, the residual evaluation is the operation that anchors FP64 arithmetic in the solver. Each refinement pass evaluates $r_m = b - Ax_m$, in which both terms are of magnitude $O(\|A\|\|x\|)$ while their difference decreases toward the size of the remaining solution error; as the leading digits cancel, the subtraction must be carried out at a precision substantially exceeding the working precision u. Conventional MPIR therefore retains an FP64 copy of A and evaluates the residual in native FP64 arithmetic. Since the requirement is accuracy rather than FP64 hardware, the natural alternative is to store A itself in split form and evaluate the residual by error-free low-precision products [13], [15].

A. Split Representation

SA-IR stores A as an unevaluated sum of two FP32 words,

$$A_{hi} = \mathrm{fl}_{32}(A), \qquad A_{lo} = \mathrm{fl}_{32}(A - A_{hi}), \tag{3}$$

in which A_{hi} carries the leading 24 bits of each entry and A_{lo} carries the following 24 bits. The pair consequently represents A to roughly 48 significant bits, with representation error $O(u_{32}^2\|A\|) \approx 2^{-48}\|A\|$ [13]. The refinement residual is then evaluated term by term as

$$r_m = b - A_{hi}x_m - A_{lo}x_m, \tag{4}$$

where each product is computed by Ozaki-scheme splitting on tensor cores and accumulated at high precision. The resulting method is classical correction-form refinement in which no FP64 matrix kernel appears: the factorization is performed

by FP32 `sgetrf`, the residual products execute on low-precision tensor cores, and the triangular solves are carried out in FP32. SA-IR preserves the memory layout of classical MPIR, requiring $4n^2$ bytes for the FP32 factors and $8n^2$ bytes for the two-word representation of A, for a persistent footprint of $12n^2$ bytes.

B. Convergence

SA-IR only modifies the arithmetic used to evaluate the residual, so its convergence behavior follows directly from the classical analysis of mixed-precision iterative refinement. The computed FP32 factors do not reproduce PA exactly; the discrepancy $PA - LU$ is the backward error of the factorization, and for a numerically stable FP32 factorization it is of magnitude $O(u_{32}\|A\|)$ [25]. For the error $e_m = x_m - x$, correction-form refinement satisfies the recurrence

$$e_{m+1} = -(LU)^{-1}\,(PA - LU)\,e_m + \delta_m, \tag{5}$$

in which δ_m collects the rounding errors incurred in the residual evaluation and the triangular solves. The homogeneous part of Eq 5 contracts when $\rho\big((LU)^{-1}(PA - LU)\big) < 1$. For a numerically stable FP32 factorization, $\|(LU)^{-1}(PA - LU)\| \lesssim c\,\kappa(A)\,u_{32}$ for a modest constant c, which yields the familiar first-order condition $\kappa(A)\,u_{32} \ll 1$ for rapid convergence [5], [10].

In the multi-precision framework of [5], SA-IR instantiates factorization precision $u_f = u_{32}$ and an effective residual precision $u_r \approx 2^{-48}$, the latter set jointly by the split representation (3) and the accuracy of the Ozaki products. The rate is governed by u_f and the attainable accuracy by u_r, so the iteration is expected to stagnate near a normalized residual of order 2^{-48}, five bits short of u_{64}; u_r is an effective precision for the split products rather than an exact bound, since their error also depends on dimension, scaling, and slice count. That floor lies far below the factorization precision that governs the rate, so it is the contraction condition $\kappa(A)\,u_{32} \ll 1$, not the representation, which determines the overall accuracy.

C. Complexity

The key advantage of SA-IR over conventional MPIR in FP32/TF32-rich environments is that it removes the last dependence on FP64 matrix units without changing the asymptotic work. Both methods factor in FP32 and perform a constant number of $O(n^2k)$ refinement passes for k right-hand sides; they differ in where the residual products execute and, because each Ozaki product expands into several low-precision products, in how many hardware operations they issue. Conventional MPIR issues them as FP64 matrix operations, so its recurring per-pass cost scales with the device's FP64 throughput, whereas SA-IR issues two $n \cdot n \cdot k$ split products at tensor-core rates. On devices where the FP32:FP64 throughput ratio is large, this shifts the residual off the scarcest units on the machine (Tab. II).

The same imbalance separates SA-IR from a direct FP64 solve, whose $\frac{2}{3}n^3$ operations execute at the FP64 rate rather than the FP32 rate; at fixed k that advantage approaches the throughput ratio, while for k growing with n the refinement work is no longer negligible and the measured behavior of Sec. V-C governs. The splitting itself is a one-time $O(n^2)$ pass, asymptotically lower order than the factorization.

IV. R-IR: Error-Explicit Iterative Refinement

R-IR retains the FP32 factors together with their factorization residual, defined below, in place of any representation of the original matrix A. The motivation is that SA-IR's persistent state is not minimal. Both A_{hi} and the computed factors approximate the same operator to FP32 accuracy, since $\|PA - LU\| = O(u_{32}\|A\|)$, and the factors are the one object that every refinement method must retain in any case. The $12n^2$-byte footprint therefore carries two FP32-accurate approximations of the operator, and devotes a further $4n^2$ bytes to A_{lo} solely to correct the one that is redundant. R-IR removes that redundancy.

A. Error-Explicit Residual

SA-IR suggests a way to view the storage that remains. Its accurate representation of A consists of a leading FP32 approximation together with a lower-order correction, $A \approx A_{hi} + A_{lo}$. After factorization, the stored factors provide another approximation to the operator, $LU \approx PA$, and the corresponding additive correction is exactly the *factorization residual*

$$R = PA - LU, \tag{6}$$

so that $PA = LU + R$. This is the quantity that governed the convergence analysis of Sec. III-B, where it appeared only as an analytical bound; here it becomes a stored object. Rather than retaining an independent two-word representation of A alongside the factors, R-IR uses the factors themselves as the leading approximation and stores only the error they leave behind. Because the factors are required for the correction solves in any case, this changes the additional operator storage from the $8n^2$ bytes of (A_{hi}, A_{lo}) to the $4n^2$ bytes of R, reducing persistent storage from $12n^2$ to $8n^2$ bytes. R-IR thus plays the same approximation-plus-correction game as SA-IR, but reuses the factorization as the approximation.

B. Storage Precision

The storage reduction rests on retaining R in FP32 rather than FP64. Doing so is safe only if R is small relative to A, as measured by $\mu \equiv \|R\|_F/\|A\|_F$. FP32 rounding perturbs R by approximately $u_{32}\|R\|$, so its effect on the represented operator diminishes as μ decreases. The classical worst-case bound [25] gives $\|R\| \le \gamma_n\|L\|\|U\|$ with $\gamma_n = nu_{32}/(1 - nu_{32})$, a bound that carries the factor n together with any element growth. At $n = 10^5$ and modest growth, it permits μ as large as 10^{-3}.

In practice we find μ to be far smaller, between 2×10^{-8} and 1.1×10^{-7} across the tested matrices. The gap between the bound and the measurement is expected under probabilistic rounding-error assumptions, as the worst-case factor n is replaced by $O(\sqrt{n \log n})$ for LU factorization [26], and on the tested matrices element growth remains modest, leaving

the measured μ within a small constant of u_{32}. At $\mu \approx 10^{-7}$, FP32 storage perturbs the operator by only $u_{32}\mu \approx 2^{-48}$, five bits short of FP64's 2^{-53} and of the same scale as SA-IR's split representation error.

C. Convergence

Given $R = PA - LU$, R-IR applies the fixed-point iteration

$$x_{m+1} = (LU)^{-1}\,(Pb - Rx_m)\,. \tag{7}$$

In exact arithmetic this is correction-form refinement, rewritten in terms of the factorization residual. Substituting $R = PA - LU$ gives $Pb - Rx_m = LUx_m + (Pb - PAx_m)$, and thus $x_{m+1} = x_m + (LU)^{-1}(Pb - PAx_m)$. For $e_m = x_m - x$, using $Pb = PAx$, we obtain

$$e_{m+1} = -(LU)^{-1} R\, e_m, \tag{8}$$

which, by Eq. (6), coincides with the homogeneous part of the SA-IR recurrence (Eq. 5). In exact arithmetic R-IR therefore contracts under the same condition as SA-IR and classical MPIR, to first order when $\kappa(A)\,u_{32} \ll 1$. The implemented iteration applies a constructed and rounded $\widehat{R} = R + \Delta R$, so its iteration matrix carries an additional $(LU)^{-1}\Delta R$ term, second order when $\|\Delta R\|/\|A\| = O(u_{32}^2)$. Equality of contraction behavior is thus a first-order argument supported by the measurements of Sec. V-A, not a property of the finite-precision algorithm. With exact R the contraction rate is likewise inherited, since the iteration matrix depends only on the factors and their error and not on how the residual is represented; in finite precision the additional $(LU)^{-1}\Delta R$ perturbation modifies that rate only at higher order under the assumptions above.

When we move to finite precision we see that classical refinement evaluates $Pb - PAx_m$, a difference of two $O(\|A\|\|x\|)$ terms that shrinks as the iterate converges, so the subtraction demands high-precision evaluation and access to the high-precision matrix. Evaluating $Pb - Rx_m$ instead removes both requirements. Since $\|R\| = O(u_{32}\|A\|)$, the product Rx_m is already smaller than the terms of the classical residual by approximately u_f, and the subtraction is no longer at risk of catastrophic cancellation. The FP32 rounding error of Rx_m, of magnitude $O(u_{32}\|R\|\|x\|) = O(u_{32}^2\|A\|\|x\|)$, lies at the two-word representation scale. We evaluate Rx_m in FP32 while retaining Pb in FP64; the difference $b - Ax$ is never formed in low precision, and A does not appear in the refinement phase. Thus the high-accuracy products that SA-IR performs at every iteration are instead performed once, during the construction of R.

In practice the solver iterates until convergence or until successive updates cease to contract, and then applies one higher-accuracy final correction. Algorithm 1 summarizes the precision used in the major operations.

D. Accuracy Floor

Whereas SA-IR's floor is set by a single quantity, *i.e.*, the residual precision u_r, R-IR introduces two additional candidate limits. These are the error incurred in constructing R and the error arising from storing R in FP32. Writing ΔR_{build} for the error of the Ozaki build of R and ΔR_{store} for its FP32 storage rounding, the idealized converged limit satisfies

$$(LU + \widehat{R})\,x = Pb, \quad \widehat{R} = R + \Delta R_{\text{build}} + \Delta R_{\text{store}}, \tag{9}$$

so the residual against $PA = LU + R$ is $\Delta R\,x$ with $\Delta R = \Delta R_{\text{build}} + \Delta R_{\text{store}}$. The reported normalized residual η_F then satisfies

$$\eta_F \;\lesssim\; \varepsilon_{\text{res}} \;+\; \|\Delta R_{\text{build}}\|/\|A\| \;+\; u_{32}\,\mu, \tag{10}$$

where ε_{res} is the accuracy of the final residual evaluation. Eq. 10 is an error budget for the represented operator rather than an accuracy bound for the finite-precision algorithm; it omits the FP32 triangular solves, the iteration itself, and the final correction, and perturbations may in any case cancel or act only weakly on x.

Should element growth enlarge R, the storage term $u_{32}\mu$ grows accordingly, and μ is known before any solve is attempted. Sec. V-A probes the storage and residual-evaluation terms and identifies which of them binds in the tested configurations.

Algorithm 1: R-IR for $Ax = b$ in error-explicit form.

Input: A (streamed or generated during setup), b (FP64)
Output: x

Setup
- $A_{32} \leftarrow \mathrm{fl}_{32}(A)$
- $L, U, P \leftarrow$ `sgetrf`(A_{32})
- $R \leftarrow PA - LU$ `// Ozaki product; store FP32`
- $\mu \leftarrow \|R\|_F/\|A\|_F$
- discard A

Solve
- $x_0 \leftarrow U^{-1}L^{-1}Pb$ `// FP32 solves`
- **repeat**
  - $d \leftarrow Pb - Rx$ `// FP32 product`
  - $x \leftarrow U^{-1}L^{-1}d$ `// FP32 solves`
- **until** *converged or stalled*
- $t \leftarrow Ux$
- $d \leftarrow Pb - Lt - Rx$ `// Ozaki FP64 accumulation`
- $x \leftarrow x + U^{-1}L^{-1}d$ `// FP32 triangular solves`

E. Complexity

Constructing R contributes $n^3/3$ logical multiply-accumulate operations at $\sim$48-bit product accuracy, each expanded into several low-precision products, a second $\Theta(n^3)$ pass where SA-IR's splitting is only $O(n^2)$. Each refinement pass then reads $8n^2$ bytes against SA-IR's $12n^2$, and a single FP32 Rx product replaces two Ozaki split products. Persistent storage is likewise $8n^2$ against $12n^2$, so for a fixed memory budget R-IR accommodates problems approximately $\sqrt{12/8} = 1.22\times$ larger in n, provided A and the completed representation are never simultaneously resident. The following subsection removes this proviso.

The setup cost is therefore $\Theta(n^3)$ and paid once, while the saving is $\Theta(n^2k)$ and accrues with every solve. R-IR is the more expensive method for a single solve and the cheaper one once the factorization is reused sufficiently often, with the crossover determined by the ratio of the two. Relative to direct FP64, R-IR requires the same hardware imbalance as SA-IR to be profitable but clears it later, since its two

TABLE I: Operator storage (bytes). N is the number of column blocks in the streamed build.

Method	Persistent	Transient	Peak
Classical MPIR	$12n^2$	—	$12n^2$
SA-IR	$12n^2$	$1.5n^2$ (solve)	$13.5n^2$
R-IR, A-resident	$8n^2$	$4n^2/N$ (build)	$8n^2{+}4n^2/N$
R-IR, A-streamed	$8n^2$	$\leq 1.5n^2$ (solve)	$\leq 9.5n^2$

low-precision $\Theta(n^3)$ passes remain cheaper than one FP64 factorization only where the FP32:FP64 throughput ratio is large enough to absorb the additional pass.

Tab. I accounts for both. Constructing R requires access to A, so the build streams or generates A one block at a time, forming the corresponding portion of R and discarding the block thereafter; the peak footprint then never exceeds the persistent $8n^2$ by more than staging workspace. If A is already resident the build may use it directly, at a peak of approximately $12n^2$ plus one transient block. A-retaining methods can stream A as well, but must repeat that access at every residual evaluation, whereas R-IR pays it once during setup. Because PA and LU nearly cancel, the build requires roughly 48-bit product accuracy so that the difference survives rounding to FP32; we obtain this from Ozaki splitting, exploiting the triangular structure of L and U.

In the A-resident build, each block of A is consumed and freed as the corresponding block of R is produced, so the two never coexist in full and the peak stays at $12n^2$ plus one transient block rather than the $16n^2$ that simultaneous residency would require.

V. Results

Platforms and workloads. We evaluate on five NVIDIA GPUs spanning multiple generations and a wide range of native FP64 capability (Tab. II), using CUDA 13.2 on a single GPU. Matrix sizes range from $n = 4096$ to 184,320 and right-hand-side counts from $k = 1$ to 8192. Performance experiments use a diagonally dominant timing family, with $n = 32{,}768$ as the reference size. Block MRHS workloads solve all k right-hand sides simultaneously as $AX = B$, enabling BLAS-3 operations and cross-RHS reuse; sequential-stream results combine separately measured $k = 1$ factorization and per-solve times.

Baselines. We compare against three cuSOLVER baselines: native-FP64 direct solve (cu-Dir), FP64-emulated direct solve (cu-Emu), and iterative refinement (cu-IR). We verify FP64 emulation through an output-level diagnostic and, unless otherwise stated, report direct-solver speedups against the faster of cu-Dir and cu-Emu.

Matrices and measurements. Numerical experiments use generated matrices with prescribed spectra, including positive and signed variants. Execution times are measured with CUDA events after one discarded warm-up and reported as medians of five runs, with refinement iteration counts determined by convergence. Memory-capacity experiments report the largest completed solve on each GPU. Sec. V-A uses controlled-condition-number matrices at $n = 32{,}768$ for convergence histories and $n = 8192$ for accuracy and sensitivity experiments.

TABLE II: Experimental platforms. FP64 rates and ratios are vendor peak specifications.

GPU	Memory	Peak FP64 (TFLOP/s)	Peak FP32:FP64 ratio
H100	80 GB	34.0	2.0
B200	183 GB	37.0	2.0
B300	287 GB	1.25	60
RTX PRO 6000	96 GB	1.95	64
RTX 5090	32.6 GB	1.64	64

Numerical accuracy is measured identically for all methods using the aggregate normalized residual

$$\eta_F = \frac{\|PB - PAX\|_F}{\|PA\|_F \|X\|_F}. \tag{11}$$

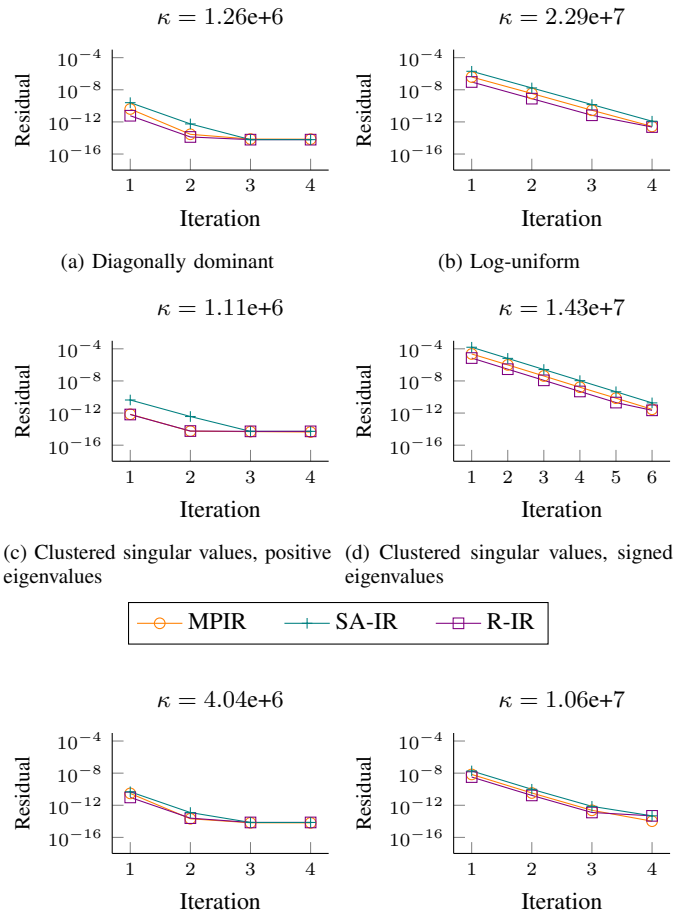
This quantity omits the $\|B\|$ term of the Rigal–Gaches normwise backward error [27] and therefore upper-bounds it, so we report it as a normalized residual rather than as a backward error. Where solution accuracy itself is at issue we report the relative difference from an FP64 direct solution x_{64}; at the condition numbers tested x_{64} is not exact, so this quantity measures agreement with a reference rather than absolute forward error.

A. Accuracy and Convergence

We first evaluate whether replacing the retained FP64 matrix A with the FP32 factorization residual R changes the numerical behavior of iterative refinement. We compare R-IR against conventional MPIR using the same FP32 LU factorization, with MPIR retaining the FP64 matrix A for residual evaluation. We then evaluate the final accuracy of both R-IR and SA-IR, determine whether FP32 storage of R limits R-IR's accuracy, and test whether stagnation is detected in cases where refinement fails to contract.

1) Convergence behavior: Condition numbers are prescribed in the 2-norm and verified against measured κ_2 within 4×10^{-5}; Fig. 1 reports the corresponding measured κ_∞. Across all six matrix families at $n = 32{,}768$, SA-IR, R-IR, and conventional MPIR require the same number of refinement iterations, from four on the easier families to six on the most difficult case. R-IR tracks MPIR closely throughout, reaching final residuals within 20% on four of the six families. SA-IR generally begins at a higher residual but contracts at the same rate, reaching the same floor on four families and trailing by roughly one iteration on the other two. All three methods share the same FP32 factorization and exact-arithmetic iteration matrix, but their residual arithmetic differs; the agreement in measured contraction rates is therefore empirical. These results support the analysis of Sec. IV-C: replacing A with either the split or error-explicit representation does not change the observed convergence behavior across the tested matrix families.

2) Attainable accuracy: Tab. III reports normalized residuals and differences from the FP64 reference solution at prescribed $\kappa(A) = 10^6$. All five methods attain $\eta_F \leq 4 \times 10^{-14}$



(a) Diagonally dominant

(b) Log-uniform

(c) Clustered singular values, positive eigenvalues

(d) Clustered singular values, signed eigenvalues

(e) Arithmetic distribution of singular values, positive eigenvalues

(f) Arithmetic distribution of singular values, signed eigenvalues

Fig. 1: Convergence of SA-IR, R-IR and FP64-A MPIR for different synthetic matrix types, each at $n = 32{,}768$ with a prescribed $\kappa = 10^6$.

across the six families. The ordering among refinement methods is family-dependent: SA-IR and R-IR outperform cu-IR on two families and trail it by up to about 2.3 orders of magnitude on the others, with the largest gaps on the log-uniform and clustered-signed families. Differences from the FP64 reference are larger, as expected under conditioning, and each of SA-IR and R-IR matches or improves on cu-IR on three of the six families. Residual and reference-solution accuracy do not always track one another; for example, on the clustered-signed family SA-IR attains a smaller residual than R-IR, but a larger difference from the FP64 reference.

3) Sensitivity to the precision of the stored residual: Across the tested matrices, the factorization residual is small relative to the original matrix, with $\mu = \|R\|_F/\|A\|_F$ ranging from 2×10^{-8} to 1.1×10^{-7}. To determine whether rounding R to FP32 limits the final accuracy, we perturb the stored residual by increasing relative amounts. A perturbation of 10^{-7} changes η_F by only 0.7%, while even a 10^{-6} perturbation, approximately $17u_{32}$, increases η_F only from 5.1×10^{-16} to 8.3×10^{-16}. Perturbations substantially larger than FP32 storage rounding are therefore required before the effect becomes

TABLE III: Accuracy by matrix family, prescribed $\kappa(A) = 10^6$

	cu-Dir	cu-Emu	cu-IR	SA-IR	R-IR
normalized residual η_F					
Diagonally dominant	2.1e-17	2.1e-17	3.0e-18	1.2e-17	2.5e-16
Log-uniform σ	1.1e-18	1.2e-18	4.3e-17	5.4e-15	3.7e-15
Clustered σ, pos.	1.8e-19	2.6e-19	9.0e-16	1.5e-17	2.1e-16
Clustered σ, signed	2.0e-19	3.1e-19	4.2e-18	2.3e-17	8.8e-16
Arithmetic σ, pos.	3.2e-19	3.1e-19	2.3e-15	3.5e-14	1.7e-14
Arithmetic σ, signed	1.9e-16	2.6e-16	6.6e-15	8.0e-16	7.5e-16
relative difference from FP64 reference $\|x - x_{64}\|/\|x_{64}\|$					
Diagonally dominant	—	3.0e-16	3.9e-15	4.3e-15	4.6e-15
Log-uniform σ	—	3.7e-13	9.2e-12	6.0e-12	3.8e-12
Clustered σ, pos.	—	1.6e-15	8.0e-12	9.4e-12	7.1e-12
Clustered σ, signed	—	1.8e-13	4.2e-12	2.1e-11	6.1e-12
Arithmetic σ, pos.	—	1.2e-13	2.9e-10	2.9e-10	5.8e-10
Arithmetic σ, signed	—	5.3e-11	1.1e-09	9.2e-11	4.3e-11

significant.

In this configuration the observed accuracy floor is instead determined by residual evaluation. On the RTX 5090, replacing the default residual product with the higher-accuracy Ozaki cascade reduces η_F from 8.8×10^{-16} to 4.2×10^{-16}, comparable to SA-IR's 3.8×10^{-16}, at a 24% increase in solve time. These results indicate that residual-evaluation precision, rather than FP32 storage of R, sets the observed accuracy floor.

4) Runtime diagnostics: At setup, $\mu = \|R\|_F/\|A\|_F$ measures whether the factorization residual is small enough for FP32 storage, and during refinement the solver monitors whether successive updates continue to contract. We evaluate these diagnostics on seven SuiteSparse matrices, materialized in dense form as numerical stress tests. Refinement stagnates on `nasa2910` and `human_gene2`, where the normalized residual degrades to as high as 4.4×10^{-12}.

Per-column backward-error measurements confirm that the aggregate Frobenius-norm metric does not mask poorly converged right-hand sides. At the reference problem size, the worst per-column residual is within $1.15\times$ of the median for every method, and per-column evaluation does not change the relative conclusions on the two stress cases.

B. Single Right-Hand Side Performance

We next evaluate complete solve time relative to the cuSOLVER baselines. Reported times include factorization and solution, and for R-IR also include the one-time construction of the factorization residual R. We first compare across GPU generations at a fixed problem size, then examine scaling with problem size on B300.

1) Dependence on native FP64 throughput: Fig. 2 compares factorization-plus-solve time for a single right-hand side at $n = 32{,}768$ across the three datacenter GPUs. Against the fastest cuSOLVER baseline among cu-Dir, cu-Emu, and cu-IR, SA-IR achieves speedups of $1.01\times$, $1.45\times$, and $1.44\times$ on H100, B200, and B300, respectively. R-IR does not outperform the fastest cuSOLVER baseline in this single-RHS configuration because the one-time construction of R cannot be amortized.

Among the vendor baselines, cu-IR provides the closest algorithmic comparison because it likewise uses mixed-precision iterative refinement, whereas cu-Dir and cu-Emu expose the

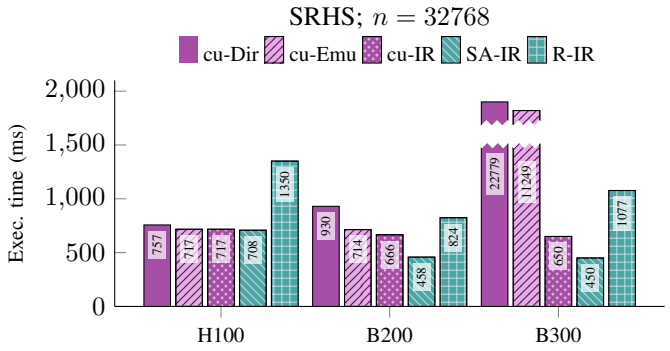


Fig. 2: End-to-end factorization-plus-solve time for a single right-hand side at $n = 32{,}768$ across three datacenter GPUs. SA-IR is competitive with the cuSOLVER baselines on H100 and becomes the fastest method on B200 and B300 as the relative cost of native FP64 increases. R-IR includes the additional one-time construction of R, which is not amortized in this single-RHS experiment.

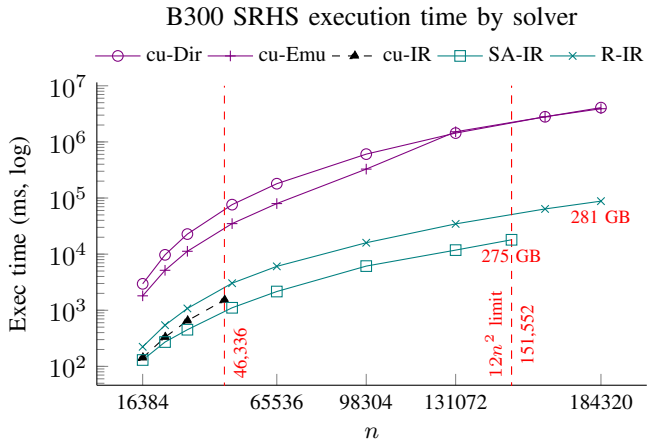


Fig. 3: End-to-end SRHS execution time on B300 as n increases. cu-IR does not complete beyond $n = 46{,}336$, while SA-IR is eventually limited by its $12n^2$ storage requirement. R-IR continues to $n = 184{,}320$ with its $8n^2$ persistent storage and outperforms cu-Dir and cu-Emu by $46.4\times$ and $44.4\times$, respectively.

effect of the changing balance between native FP64 and low-precision throughput. That comparison reveals a stronger architectural trend. H100 and B200 provide peak FP32:FP64 throughput ratios near $2\times$, whereas B300 raises this ratio to approximately $60\times$ by substantially reducing native FP64 throughput. Relative to the faster of cu-Dir and cu-Emu, SA-IR's speedup therefore grows from $1.01\times$ on H100 and $1.56\times$ on B200 to $25.0\times$ on B300. R-IR similarly moves from slower than the direct baselines on H100 and B200 to $10.4\times$ faster on B300, despite its additional construction of R. The benefit of replacing FP64 computation with low-precision arithmetic thus increases substantially as native FP64 is deprioritized.

2) Scaling with problem size: We next examine scaling with problem size on B300, which has the largest memory capacity among the evaluated GPUs. Fig. 3 reports end-to-end single-RHS execution time as n increases, and the advantage of both proposed methods grows throughout. At $n = 98{,}304$, SA-IR and R-IR complete in 6.1 s and 16.0 s, respectively, against 327.7 s for the fastest cuSOLVER baseline that completes at this size, corresponding to speedups of $53.5\times$ and $20.5\times$. SA-IR continues to $n = 151{,}552$, where its $12n^2$ storage reaches device capacity, whereas R-IR continues to $n = 184{,}320$ and completes in 88 s, $46.4\times$ and $44.4\times$ faster than cu-Dir and cu-Emu, respectively. cu-IR does not complete beyond $n = 46{,}336$, where n^2 exceeds the range of the 32-bit indexing used internally by the vendor solver.

C. Multiple Right-Hand Sides and Factorization Reuse

Fig. 4 reports execution time at $n = 32{,}768$ as k increases on B200 and B300, and the two devices separate sharply. On B300 at $k = 2048$, SA-IR and R-IR complete in 879 ms and 1351 ms against 13,388 ms for cu-Emu and 13,615 ms for cu-IR, speedups of $15.2\times$ and $9.9\times$ over the faster direct baseline. The gap widens with k as cu-IR grows from 649 ms at $k = 8$ to 52,367 ms at $k = 8192$, while R-IR grows only from 1072 ms to 2184 ms, a $24.0\times$ advantage at the largest k. The difference stems from residual evaluation, since cu-IR forms a product with the retained high-precision operator on every refinement iteration whereas SA-IR and R-IR perform the corresponding products in low precision.

1) Comparison of SA-IR and R-IR: The two proposed methods also separate from each other, in the direction predicted by Sec. IV-E. SA-IR is faster at moderate k because it avoids constructing R, leading by $1.5\times$ at $k = 2048$ on B300. Because each refinement pass reads $8n^2$ bytes for R-IR against $12n^2$ for SA-IR and replaces two split products with a single FP32 product, the gap closes as the number of passes grows, and at $k = 8192$ R-IR is the faster of the two (2184 ms against 2401 ms). The construction of R is thus a fixed cost that is repaid by a cheaper refinement pass, and the crossover falls between $k = 4096$ and $k = 8192$ at this problem size.

2) Sequential reuse: Block workloads exploit cross-RHS reuse, but some applications instead solve a sequence of right-hand sides one at a time. At $n = 8192$ on B300, R-IR requires 53.6 ms for factorization and setup and 7.81 ms per subsequent solve. SA-IR has a lower initial setup cost but a

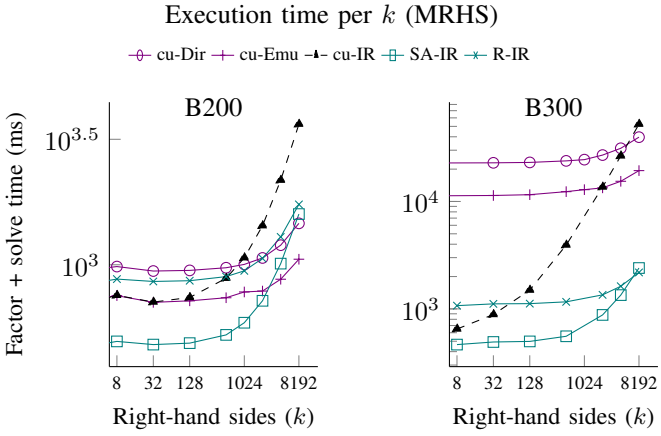


Fig. 4: End-to-end execution time as the number of right-hand sides k increases. On B300, SA-IR and R-IR scale much more slowly with k than cu-IR, whose residual evaluation continues to involve the retained high-precision operator. The B200 results show a smaller separation because native FP64 remains substantially stronger.

TABLE IV: Largest completed streamed R-IR solve on each GPU.

GPU	Memory (GB)	Largest n	Peak R-IR footprint	$12n^2$ at same n (GB)
H100	80	96,256	$8.36n^2$	111
B200	183	147,456	$8.24n^2$	261
B300	287	184,320	$8.26n^2$	408
RTX PRO 6000	96	106,496	$8.35n^2$	136
RTX 5090	32.6	56,320	$8.57n^2$	38

higher per-RHS application cost of 8.95 ms, so R-IR recovers its additional setup cost after roughly 17 solves. Relative to a direct FP64 solve the tradeoff is reversed, since R-IR's factorization is much cheaper but each subsequent triangular-solve phase is more expensive, so sufficiently long sequential streams can eventually favor the direct solver.

D. Maximum Solvable Problem Size

We next examine whether R-IR's reduced persistent storage translates into a larger maximum solvable problem size. Conventional A-retaining refinement and SA-IR require $12n^2$ bytes of persistent operator storage, whereas R-IR requires only $8n^2$ bytes for the FP32 factors and factorization residual R.

1) Measured capacity: Tab. IV reports the largest completed R-IR solve on each device using the streamed construction of R. The measured peak footprint remains between $8.24n^2$ and $8.57n^2$ bytes across all five GPUs, confirming that the streamed construction does not reintroduce a transient $12n^2$ peak. On every device, R-IR completes a problem for which a $12n^2$ representation would exceed the available device memory. R-IR reaches $n = 147{,}456$ on the 183 GB B200 and $n = 184{,}320$ on the 287 GB B300, whereas a $12n^2$ representation would require 261 GB and 408 GB, respectively.

2) Performance at extended problem sizes: Fig. 5 compares R-IR with cu-Dir and cu-Emu at a large problem size on each GPU. At these sizes SA-IR exceeds device-memory capacity because of its $12n^2$ requirement, while cu-IR does not complete beyond $n = 46{,}336$ in our experiments, leaving R-IR as the only refinement-based method in the comparison.

The hardware-dependent trend observed earlier persists. On B300, R-IR completes in 88 s against 4082 s for cu-Dir and 3905 s for cu-Emu, speedups of 46.4× and 44.4×. On the RTX PRO 6000 the corresponding speedups are 4.5× and 4.1×, and on the RTX 5090 3.1× and 2.0×. R-IR is modestly faster on B200 but trails both direct solvers by approximately 2.1× on H100, where native FP64 remains strong.

E. Energy Efficiency

Fig. 6 reports end-to-end energy at $n = 32{,}768$ and $k = 2048$. Energy broadly follows execution time: on B300, SA-IR and R-IR reduce energy by up to 11.3× and 6.4×, respectively, relative to the direct baselines, and both consume less energy than cu-IR. Against cu-IR, SA-IR also reduces energy on the RTX PRO 6000 and RTX 5090, while R-IR does so on the RTX PRO 6000 but not the RTX 5090, where its longer runtime offsets the power advantage. The benefit decreases as native FP64 becomes stronger, with both methods consuming more energy than the direct baselines on H100.

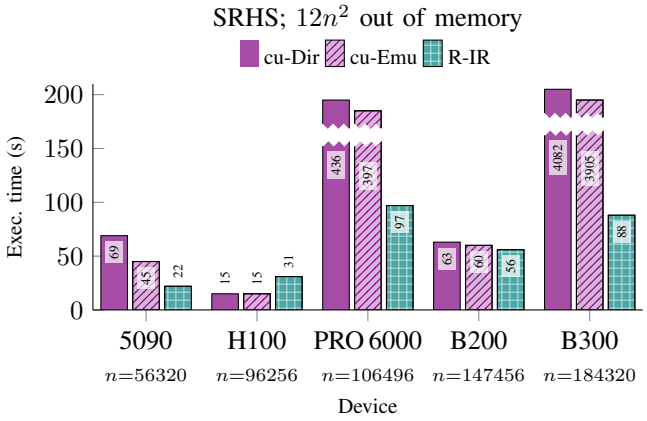


Fig. 5: End-to-end SRHS execution time at a large problem size on each GPU. SA-IR exceeds memory capacity at these sizes, while cu-IR does not complete beyond $n = 46{,}336$. Each GPU uses its largest completed R-IR matrix size; GPUs are ordered by increasing n.

VI. Conclusion

We presented two low-precision iterative-refinement algorithms that remove native FP64 matrix operations from the refinement process. SA-IR represents A using two FP32 words, while R-IR stores the factorization residual $R = PA - LU$ and discards A after setup. R-IR inherits MPIR's first-order convergence condition, and in the tested configurations its accuracy is limited by residual-evaluation precision rather than FP32 storage of R.

The two representations expose a complementary tradeoff between performance and memory. SA-IR is generally faster when its $12n^2$ representation fits in memory and setup cost matters, whereas R-IR reduces persistent storage to $8n^2$ and benefits from factorization reuse. On B300, SA-IR reaches 25× speedup over emulated FP64 for a single right-hand side, while R-IR reaches 24× over vendor refinement for 8192 right-hand sides. Streamed R-IR requires only $8.24n^2$–$8.57n^2$ bytes in practice and solves problems that a $12n^2$ representation cannot fit. These results suggest that storing factorization error rather than the original operator may benefit

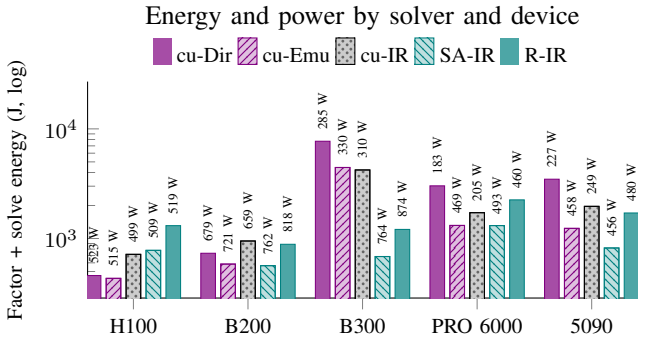


Fig. 6: End-to-end energy per solve at $n = 32{,}768$ and $k = 2048$ computed from GPU board power. Energy savings broadly follow the performance trends: SA-IR and R-IR are particularly energy efficient on B300, where native FP64 throughput is strongly deprioritized.

other factorization-based solvers as low-precision throughput continues to outpace FP64 on modern GPUs.

REFERENCES

[1] K. Aubrey, "Inside the NVIDIA Vera Rubin platform: Six new chips, one AI supercomputer," NVIDIA Technical Blog, Jan. 2026, published January 5, 2026. [Online]. Available: https://developer.nvidia.com/blog/inside-the-nvidia-rubin-platform-six-new-chips-one-ai-supercomputer/

[2] J. H. Wilkinson, *Rounding Errors in Algebraic Processes*, ser. Notes on Applied Science No. 32. London: Her Majesty's Stationery Office, 1963, reprinted by Dover, New York, 1994.

[3] C. B. Moler, "Iterative refinement in floating point," *Journal of the ACM*, vol. 14, no. 2, pp. 316–321, 1967.

[4] 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 *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC '06)*. Tampa, FL: ACM, 2006, p. 113.

[5] E. Carson and N. J. Higham, "Accelerating the solution of linear systems by iterative refinement in three precisions," *SIAM Journal on Scientific Computing*, vol. 40, no. 2, pp. A817–A847, 2018.

[6] A. Haidar, S. Tomov, J. Dongarra, and N. J. Higham, "Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC '18)*. Dallas, TX: IEEE, 2018, pp. 47:1–47:11.

[7] A. Haidar, H. Bayraktar, S. Tomov, J. Dongarra, and N. J. Higham, "Mixed-precision iterative refinement using tensor cores on GPUs to accelerate solution of linear systems," *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 476, no. 2243, p. 20200110, 2020.

[8] NVIDIA Corporation, "cuSOLVER library documentation: Iterative refinement solver (IRS) functions, cusolverDnIRSXgesv," https://docs.nvidia.com/cuda/cusolver/index.html, 2026, CUDA Toolkit documentation; accessed 2026-07-29.

[9] J. Dongarra and P. Luszczek, "HPL-MxP benchmark: Mixed-precision algorithms, iterative refinement, and scalable data generation," *The International Journal of High Performance Computing Applications*, vol. 40, no. 1, 2026, also available as arXiv:2509.19618.

[10] E. Carson and N. J. Higham, "A new analysis of iterative refinement and its application to accurate solution of ill-conditioned sparse linear systems," *SIAM Journal on Scientific Computing*, vol. 39, no. 6, pp. A2834–A2856, 2017.

[11] N. J. Higham and T. Mary, "Mixed precision algorithms in numerical linear algebra," *Acta Numerica*, vol. 31, pp. 347–414, 2022.

[12] A. Abdelfattah, H. Anzt, E. G. Boman, E. Carson, T. Cojean, J. Dongarra, A. Fox, M. Gates, N. J. Higham, X. S. Li, J. Loe, P. Luszczek, S. Pranesh, S. Rajamanickam, T. Ribizel, B. F. Smith, K. Swirydowicz, S. Thomas, S. Tomov, Y. M. Tsai, and U. M. Yang, "A survey of numerical linear algebra methods utilizing mixed-precision arithmetic," *The International Journal of High Performance Computing Applications*, vol. 35, no. 4, pp. 344–369, 2021.

[13] T. J. Dekker, "A floating-point technique for extending the available precision," *Numerische Mathematik*, vol. 18, no. 3, pp. 224–242, 1971.

[14] T. Ogita, S. M. Rump, and S. Oishi, "Accurate sum and dot product," *SIAM Journal on Scientific Computing*, vol. 26, no. 6, pp. 1955–1988, 2005.

[15] K. Ozaki, T. Ogita, S. Oishi, and S. M. Rump, "Error-free transformations of matrix multiplication by using fast routines of matrix multiplication and its applications," *Numerical Algorithms*, vol. 59, no. 1, pp. 95–118, 2012.

[16] D. Mukunoki, K. Ozaki, T. Ogita, and T. Imamura, "DGEMM using Tensor Cores, and its accurate and reproducible versions," in *High Performance Computing (ISC High Performance 2020)*, ser. Lecture Notes in Computer Science, vol. 12151. Springer, 2020, pp. 230–248.

[17] H. Ootomo and R. Yokota, "Recovering single precision accuracy from Tensor Cores while surpassing the FP32 theoretical peak performance," *The International Journal of High Performance Computing Applications*, vol. 36, no. 4, pp. 475–491, 2022.

[18] H. Ootomo, K. Ozaki, and R. Yokota, "DGEMM on integer matrix multiplication unit," *The International Journal of High Performance Computing Applications*, vol. 38, no. 4, pp. 297–313, 2024.

[19] Y. Uchino, K. Ozaki, and T. Imamura, "Performance enhancement of the Ozaki Scheme on integer matrix multiplication unit," *The International Journal of High Performance Computing Applications*, vol. 39, no. 3, pp. 462–476, 2025.

[20] NVIDIA Corporation, "Boosting matrix multiplication speed and flexibility with NVIDIA cuBLAS 12.9," NVIDIA Technical Blog, https://developer.nvidia.com/blog/boosting-matrix-multiplication-speed-and-flexibility-with-nvidia-cublas-12-9/, 2025, accessed 2026-07-29. Introduces the floating-point emulation building blocks in cuBLAS 12.9.

[21] C. Brower, J. Gunnels, and G. Lopez, "Unlocking tensor core performance with floating point emulation in cuBLAS," NVIDIA Technical Blog, https://developer.nvidia.com/blog/unlocking-tensor-core-performance-with-floating-point-emulation-in-cublas/, Oct. 2025, accessed 2026-07-29. Ozaki-scheme FP64 emulation on Blackwell INT8 tensor cores in cuBLAS (CUDA 13.0 Update 2): up to 2.3× over native FP64 on GB200 NVL72 and up to 13× on RTX PRO 6000 Blackwell Server Edition.

[22] A. Schwarz, A. Anders, C. Brower, H. Bayraktar, J. Gunnels, K. Clark, R. G. Xu, S. Rodriguez, S. Cayrols, P. Tabaszewski, and V. Podlozhnyuk, "Guaranteed DGEMM accuracy while using reduced precision tensor cores through extensions of the Ozaki scheme," arXiv preprint arXiv:2511.13778, 2025.

[23] A. Abdelfattah, J. Dongarra, M. Fasi, M. Mikaitis, and F. Tisseur, "Analysis of floating-point matrix multiplication computed via integer arithmetic," arXiv preprint arXiv:2506.11277, 2025.

[24] N. J. Higham and T. Mary, "A new preconditioner that exploits low-rank approximations to factorization error," *SIAM Journal on Scientific Computing*, vol. 41, no. 1, pp. A59–A82, 2019.

[25] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*, 2nd ed. Philadelphia, PA: Society for Industrial and Applied Mathematics, 2002.

[26] N. J. Higham and T. Mary, "A new approach to probabilistic rounding error analysis," *SIAM Journal on Scientific Computing*, vol. 41, no. 5, pp. A2815–A2835, 2019.

[27] J.-L. Rigal and J. Gaches, "On the compatibility of a given solution with the data of a linear system," *Journal of the ACM*, vol. 14, no. 3, pp. 543–548, 1967.