An Integer Arithmetic-Based Sparse Linear Solver Using GMRES and Iterative Refinement

Presentation Discourse

Giulia Lionetti Alberto Taddei

January 27, 2026

Slide 1: Introduction

Good morning everyone. Today we are going to present a paper by Iwashita, Suzuki, and Fukaya that tackles what might seem like an impossible challenge: solving large sparse linear systems to double-precision accuracy using only integer arithmetic in the computational core.

Slide 2: Roadmap

Our presentation will follow five main sections. First, we'll explore the motivation—why would anyone want to abandon floating-point arithmetic when it's been the foundation of scientific computing for decades? Second, we'll identify the core problems that arise when you remove floating-point from iterative solvers. Third, we'll dive into the authors' elegant three-layer solution architecture. Fourth, we'll examine their experimental results to see if this actually works in practice. And finally, we'll discuss the limitations and broader implications of this work.

Slide 3: Is Moore's Law Dead?

A provocative question: Is Moore's Law dead? Well, perhaps it's more accurate to say it's evolving in unexpected ways. Three major trends are converging in the hardware landscape.

First, energy efficiency improvements are slowing dramatically. We're approaching fundamental physical limits—transistors can't get much smaller without quantum effects dominating, and power density is already a major constraint.

Second, novel architectures are emerging that challenge our assumptions. Superconducting single flux quantum circuits, for example, operate at cryogenic temperatures and use incredibly low power, but they're naturally suited to integer operations. Neuromorphic chips inspired by biological neurons similarly favor simple, integer-based computations. These aren't just lab curiosities—they represent serious bets by hardware researchers on post-CMOS computing.

Third, and most critically, these emerging technologies may only support integer arithmetic efficiently. Why? Because floating-point units are complex beasts. They require sophisticated circuitry to handle the mantissa, exponent, rounding, special values like infinity and NaN, and subnormal numbers. A 64-bit floating-point multiply-add might consume 50 picojoules of energy, while an equivalent-width integer operation uses only 5 picojoules—a 10x difference. In energy-constrained environments, that factor matters enormously.

This raises a fundamental question: Can we do real scientific computing—solving PDEs, running simulations, analyzing massive datasets—without floating-point hardware? That's what this paper sets out to explore.

Slide 4: The Problem

The specific challenge is this: solve the sparse linear system $A\hat{x} = \hat{b}$ to double precision accuracy using only integer arithmetic in the main computational loop.

Why is this hard? Three fundamental reasons. First, integers have a fixed range—overflow is an ever-present danger when you're multiplying large values or accumulating many terms. Second, bit shifts lose precision. Unlike floating-point where the exponent adjusts automatically, with integers you must manually shift bits to keep values in range, and every right shift permanently discards the least significant bits. Third, there's no dynamic exponent to handle the enormous range of magnitudes that can appear in scientific computations—we're talking about values that might span 20 or 30 orders of magnitude in typical PDE discretizations.

Yet despite these challenges, the requirements remain stringent: we need GMRES-like convergence behavior, double-precision accuracy in the final result, and robustness to ill-conditioned matrices that are common in real applications.

Slide 5: What We Lose Without Floating-Point

Floating-point numbers use the representation $\pm m \times 2^e$, where the exponent e provides automatic range management. If you multiply two numbers and the result is huge, the exponent just increases. If you add numbers of vastly different magnitudes, the hardware handles alignment automatically.

With fixed-point integers, you have a fixed number of bits, period. The algorithm must manually manage the range through shifts. This dramatically increases overflow risk—one poorly scaled intermediate result can ruin your computation. On the other hand, the energy cost drops by nearly an order of magnitude, and hardware complexity plummets.

This is simultaneously the challenge and the opportunity. The challenge is making it work algorithmically. The opportunity is enabling scientific computing on ultra-low-power hardware that can't afford floating-point units.

Slide 6: State of the Art

Before diving into the solution, let's acknowledge what's already been done. Mixed-precision iterative refinement is a well-established technique, pioneered by people like

Göddeke in 2007 and extensively studied by Carson and Higham around 2018. The idea is simple but powerful: solve your system approximately in low precision (say FP16 or FP32), then compute the residual and correction in high precision (FP64). This gives you FP64 accuracy with much of the computational work done in cheaper arithmetic.

This works beautifully and has been deployed successfully on GPUs and other modern hardware. But—and this is the critical point—it still requires floating-point hardware. You're just using different precisions of it.

The gap that this paper addresses is stark: nobody had built Krylov subspace solvers that work without floating-point kernels at all.

Slide 7: Three-Layer Architecture

Now we get to the heart of the solution: a clever three-layer architecture that addresses different aspects of the integer arithmetic challenge.

Layer 1 is iterative refinement, which handles the gross range management. You compute the residual in floating-point, scale it appropriately before sending it to the integer solver, and accumulate corrections. This is your outer control loop.

Layer 2 is matrix decomposition. They represent the matrix A as $A_0 + 2^{-\alpha_1}A_1 + 2^{-\alpha_2}A_2 + \ldots$, where each A_i contains only integers and the weights are powers of two (so multiplication by them is just bit shifts). This gives you progressive accuracy—you can start with just A_0 for early iterations and add finer terms as you need more precision.

Layer 3 is integer-GMRES itself—the full Arnoldi orthogonalization, Givens rotations, and matrix-vector products, all implemented in fixed-point arithmetic. This is where the computational work happens, and critically, it stays bounded because of the scaling from Layer 1.

The brilliance is how these layers work together: the integer loop stays numerically bounded while floating-point operations at the boundaries recover full accuracy.

Slide 8: Layer 1 – Iterative Refinement

Let's dig deeper into Layer 1. The standard iterative refinement formula is $x \leftarrow x + \gamma^{(k)}x^{(k)}$, where $x^{(k)}$ solves a scaled version of the residual system.

Here's why this matters for integer arithmetic: the residual $b' = b - Ax$ shrinks with each iteration. As you converge, the residual gets smaller and smaller. Before sending this to the integer solver, they scale it by $1/\gamma$ where $\gamma = \max|b'_i|$. This normalization ensures the integer solver always sees values with magnitudes near 1.

This is automatic range control. Without refinement, the integer solver would have to handle wildly varying input magnitudes across iterations, requiring different shift strategies and risking overflow. With refinement providing normalized inputs, the integer arithmetic can operate in a predictable, stable regime. It's an elegant exploitation of the refinement framework for a purpose beyond just accuracy—it provides numerical stability for the lower layer.

Slide 9: Layer 2 – Matrix Decomposition

The matrix decomposition layer addresses another fundamental challenge: representing real-valued matrices using integers without losing too much information.

Their approach decomposes A into multiple integer matrices with power-of-two weights: $A = A_0 + 2^{-\alpha_1}A_1 + 2^{-\alpha_2}A_2 + \ldots$ Each A_i contains only integers, capturing progressively finer detail about A.

A_0 captures the most significant digits—the gross structure of the matrix. A_1 captures the next level of detail, weighted by $2^{-\alpha_1}$, and so on. Because the weights are powers of two, applying them is just bit shifting, which is extremely cheap.

This gives them progressive accuracy: in early refinement iterations when you don't need full precision, you can use just A_0. As you get closer to convergence and need more accuracy, you add A_1, then A_2, and so on. This adaptivity helps manage the precision-versus-overflow tradeoff intelligently.

Slide 10: Layer 3 – Integer-GMRES

Now we reach the computational core: integer-GMRES. The structure follows standard GMRES—Arnoldi orthogonalization to build the Krylov subspace, Givens rotations to solve the least squares problem incrementally, and matrix-vector products at each step.

The key difference is that all the inner kernels use fixed-point arithmetic: matvecs, dot products, norms, and Givens rotations all operate on integers. Floating-point is only used for the initial residual computation, the final least squares solve, and the solution update—the parts that touch the outer refinement loop.

But here's the rub: dot products and norms are dangerous in fixed-point. When you compute $\sum x_i y_i$ with n terms, each product might be large, and summing n of them can easily overflow even 64-bit integers for moderately sized problems. This is the technical heart of the challenge.

Slide 11: Fixed-Point Arithmetic

Let us clarify what fixed-point arithmetic means here. They use the Qdm.df format: a sign bit, dm bits for the integer part, and df bits for the fractional part. With a 64-bit word and df=30, you have roughly 33 bits for the integer part (including sign).

The overflow problem is fundamental. If you multiply two numbers each using, say, 30 bits, the product needs 60 bits. If you're not careful, overflow immediately. The standard solution is to shift right by β bits before or after multiplication: result $= (t_1 \gg \beta_1) \cdot (t_2 \gg \beta_2) \gg (df - \beta_1 - \beta_2)$.

But this creates a terrible dilemma. More shift means safer (less overflow risk) but you lose bits of precision. Less shift means risky (overflow danger) but you keep more bits. How do you navigate this trade-off? That's where algorithmic insight becomes crucial.

Slide 12: Smart Shifting

Here's where the authors' deep understanding of GMRES pays dividends. They exploit the mathematical structure of GMRES to minimize necessary shifts:

First, Krylov vectors are normalized to unit length. This means they have many leading zeros in their fixed-point representation—they're safely bounded away from overflow.

Second, Givens rotation coefficients satisfy $|c|, |s| \leq 1$ by construction—they're inherently safe to manipulate.

Third, dot products between normalized vectors are bounded by the Cauchy-Schwarz inequality: $|\langle u, v\rangle| \leq \|u\|\|v\| = 1$ when both are unit vectors.

By carefully analyzing each operation in GMRES, they determine the minimum shifts needed to guarantee safety. This isn't just heuristic—it uses algorithm invariants (properties that are mathematically guaranteed to hold) to maximize effective precision.

The key insight is moving from "shift enough to be safe in the worst case" to "shift exactly what the algorithm structure guarantees we need." This precision management through algorithmic knowledge is beautiful.

Slide 13: Core Algorithm – Visual Overview

Let us walk through the algorithm visually. At the boundary, you compute the initial residual r_0 and first Krylov vector v_1 in floating-point. You immediately cast v_1 to fixed-point—this is your entry into the integer world.

Now the integer loop begins. You do matrix-vector multiplication, dot products for orthogonalization, norm computations, and Givens rotations—all in fixed-point. The Arnoldi process builds up your Krylov basis orthogonally. Givens rotations update the least squares problem incrementally.

Throughout this loop, the arithmetic stays bounded. Values don't explode because of the normalization from the refinement layer and the careful shift management within the GMRES structure.

Finally, when the loop finishes, you solve the small least squares problem in floating-point and update the solution. Then you're back in floating-point land for the next refinement iteration.

Slide 14: Core Algorithm – Pseudocode

Looking at the pseudocode makes this concrete. Line 1 computes r_0 and v_1 in floating-point—this is the refinement layer preparing inputs. Line 2 casts v_1 to fixed-point—the entry into integer arithmetic.

The loop from lines 3-11 is pure integer operations. Line 4: matrix-vector multiply with the integer matrix representation. Lines 5-7: the Arnoldi orthogonalization using integer dot products and subtractions. Line 9: norm and normalization in integer arithmetic. Line 10: Givens rotations to triangularize the Hessenberg matrix, also in integers.

Notice that each operation is annotated with “INT” to emphasize: no floating-point happens here. Every dot product, every vector update, every rotation uses the carefully designed fixed-point arithmetic with minimal, structure-aware shifts.

Line 12 returns to floating-point for the least squares solve and solution update. This hybrid design—integer for the bulk computation, floating-point for precision recovery—is the architectural insight that makes everything work.

Slide 15: Preconditioning is Critical

Now, preconditioning. In standard GMRES, preconditioning accelerates convergence, which is nice for performance. But in integer-GMRES, preconditioning serves a second, equally important role: it reduces overflow risk.

Here’s the logic chain. Better conditioning means the matrix is closer to well-scaled, so eigenvalues aren’t spread over huge ranges. This means Krylov vectors don’t develop extremely disparate components, which means intermediate values stay smaller in magnitude. Smaller values mean fewer bits shifts needed to avoid overflow. Fewer shifts mean higher effective precision. Higher precision means better convergence.

So preconditioning isn’t just about convergence speed—it’s about numerical feasibility. They use ILU(0), which is an incomplete LU factorization with no fill-in. Critically, they implement ILU entirely in integer arithmetic, maintaining the no-floating-point discipline even in the preconditioner construction.

This makes preconditioning essential for the integer approach to work robustly.

Slide 16: Experimental Setup

Let’s turn to the experiments. They test on 10 sparse matrices from the SuiteSparse collection, which is the standard benchmark repository for sparse matrix algorithms. These matrices come from real applications—structural mechanics, circuit simulation, computational fluid dynamics.

The target is a relative residual less than 10^{-8}, measured in full double precision. This is real double-precision accuracy, not some relaxed tolerance.

For the fixed-point format, they use word length WL=64 bits total, with df=30 fractional bits. This gives about 33 bits for the integer part, which is a reasonable middle ground.

They compare iteration counts between double-precision GMRES (the gold standard) and their integer-GMRES. Note carefully: this paper measures convergence, not performance. They’re asking “does it converge to the right answer in a reasonable number of iterations?” They’re not measuring wall-clock time or energy consumption because they’re running on conventional hardware. To measure actual speedups and energy savings, you’d need the target integer-only hardware, which doesn’t exist yet. This is foundational algorithmic work proving feasibility.

Slide 17: Without Preconditioning

Here are results without preconditioning. Looking at the table, we see mixed outcomes. For atmosmodj and atmosmodl, the iteration counts are identical—2100 and 420 respectively. Integer-GMRES matches double-GMRES exactly.

But for wang3, integer-GMRES needs 630 iterations versus 510 for double—that's 24% more iterations. For cage14, it's worse: 60 iterations versus 30, a full 2x slowdown.

What's happening? These latter matrices are more challenging numerically. Without preconditioning, the integer solver has to use aggressive shifts to prevent overflow, which loses precision. That precision loss slows convergence—you need more iterations to compensate for the arithmetic quality degradation.

This variability shows that naive integer arithmetic isn't a drop-in replacement for floating-point. Some problems work fine; others struggle. This motivates the need for preconditioning.

Slide 18: With ILU Preconditioning

Now add ILU(0) preconditioning, and the picture transforms dramatically. Atmosmodj: 300 iterations for both double and integer—identical. Atmosmodl: 120 each—identical. Wang3: 120 each—now identical, whereas it was 24% slower before. Cage14 is still 2x slower at 60 versus 30, but that's the outlier.

For most matrices, with preconditioning, integer-GMRES matches double-precision GMRES iteration-for-iteration. This is remarkable. It means the fixed-point arithmetic, when properly managed through smart shifts and good conditioning, doesn't degrade the convergence rate of the iterative solver.

This validates the core hypothesis: with careful algorithm design and preconditioning, integer arithmetic can substitute for floating-point in Krylov solvers without sacrificing solution quality or iteration count.

Slide 19: Convergence Curves

The left plot shows atmosmodj without preconditioning. The red line (integer-GMRES) and green line (double-GMRES) are literally on top of each other—the convergence curves are identical. The residual decreases smoothly and monotonically from 10^0 to below 10^{-8} over about 2100 iterations, and both solvers take exactly the same path.

The right plot shows wang3 without preconditioning. Here you can see the integer solver (red) lagging behind the double solver (green). It starts similarly, but around iteration 200 the gap opens up. The double solver reaches 10^{-8} by iteration 510, while the integer solver needs 630 iterations. The degradation is visible but not catastrophic—it's not failing, just slower.

This graphically illustrates the matrix-dependence of the integer approach's effectiveness when preconditioning isn't used.

Slide 20: With ILU – Wang3

Now here's wang3 with ILU preconditioning. The two curves are essentially indistinguishable again. Both solvers converge smoothly from 10^0 to 10^{-8} in exactly 120 iterations. The preconditioning has stabilized the integer arithmetic so completely that its convergence behavior matches floating-point.

This is the headline result: properly preconditioned integer-GMRES can deliver double-precision solutions with convergence rates competitive with double-precision GMRES, despite using only integer operations in the computational core.

Slide 21: What They Contributed

Let's step back and summarize the contributions. First, a working integer-GMRES implementation—this didn't exist before. It's not just a theoretical sketch; they actually built it and demonstrated it on real matrices.

Second, an iterative refinement framework specifically designed for integer arithmetic, using refinement not just for accuracy recovery but for range control.

Third, an operation-specific shift strategy that exploits GMRES's mathematical invariants to minimize precision loss. They didn't just apply generic fixed-point arithmetic; they tailored it to the algorithm structure.

Fourth, empirical evidence that ILU preconditioning is essential for robust integer solving. This is a practical insight that would guide anyone trying to implement these ideas.

The main conceptual insight is this: precision management moves from hardware to algorithm. In floating-point, the hardware manages exponents automatically. In integer arithmetic, the algorithm must explicitly manage range, normalization, and shifts. This is more work for the algorithm designer, but it enables computation on radically simpler, lower-power hardware.

Slide 22: Limitations

Of course, there are limitations. First, they only tested on moderately conditioned problems—matrices with condition numbers up to maybe 10^6 or 10^8. Really ill-conditioned problems (condition number 10^{12} or higher) might struggle. We don't know the boundaries yet.

Second, the approach has several tuning parameters: the number of fractional bits df, the shift amounts for different operations, the depth of the matrix decomposition. These currently require problem-specific tuning. There's no automatic, black-box way to set them optimally.

Third, crucially, there are no actual performance or energy measurements. They demonstrate algorithmic feasibility—that the iteration counts are competitive—but the claimed energy benefits are theoretical. To validate speedup and power savings, you'd need actual integer-only hardware (which doesn't exist yet) or at least cycle-accurate simulation with energy modeling.

Fourth, theoretical convergence guarantees are unclear. Classical GMRES convergence analysis assumes exact arithmetic. With fixed-point, you have bounded rounding errors at every operation. How do these accumulate? What's the worst-case? The error analysis is incomplete.

But despite these limitations, the work demonstrates feasibility. Integer Krylov solvers can work, which is a significant threshold result that opens up further research.

Slide 23: Conclusions

Let us conclude with the key takeaways. Integer-only solvers are viable if you manage range explicitly and carefully. The techniques demonstrated here—iterative refinement for range control, smart shifts exploiting algorithmic structure, progressive matrix decomposition, and aggressive preconditioning—provide a blueprint for making it work.

This opens a path to ultra-low-power scientific computing. Imagine solving large PDEs on neuromorphic chips or superconducting circuits that consume a tiny fraction of the power of conventional processors. That's the long-term vision this enables.

Of course, significant work remains. We need better error analysis, automatic parameter tuning, extensions to other Krylov methods and problem classes, and most importantly, validation on actual integer-only hardware. But this paper demonstrates that the core challenge—achieving double-precision accuracy without floating-point in the computation—is solvable.