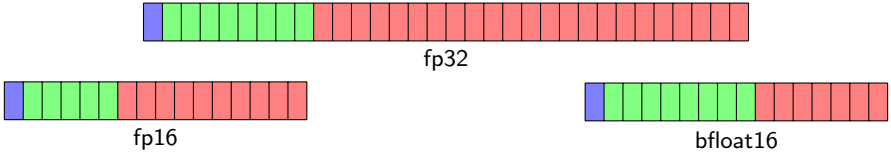
Mixed precision algorithms: an overview

Theo Mary
Sorbonne Université, CNRS, LIP6

NHR PerfLab Seminar Series, 12 December 2023

Slides available at
https://bit.ly/NHRmixed



Floating-point arithmetics

		number of bits			
		signif. (t)	exp.	range	$u = 2^{-t}$
fp128	quadruple	113	15	$10^{\pm 4932}$	1×10^{-34}
fp64	double	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	single	24	8	$10^{\pm 38}$	6×10^{-8}
fp16	half	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16		8	8	$10^{\pm 38}$	4×10^{-3}
fp8 (e4m3)	quarter	4	4	$10^{\pm 2}$	6×10^{-2}
fp8 (e5m2)		3	5	$10^{\pm 5}$	1×10^{-1}

The unit roundoff $u = 2^{-t}$ determines the relative accuracy any number in the representable range can be approximated with:

If $x \in \mathbb{R}$ belongs to $[e_{\min}, e_{\max}]$, then $\mathrm{fl}(x) = x(1+\delta), \quad |\delta| \leq u$

Moreover the **standard model of arithmetic** is

$\mathrm{fl}(x \operatorname{op} y) = (x \operatorname{op} y)(1+\delta), \quad |\delta| \leq u$, for $\operatorname{op} \in \{+, -, \times, \div\}$

Pros and cons of lower precisions

☺ Storage, data movement and communications are all proportional to total number of bits (mantissa + exponent)
lower precision ⇒ lighter computations

☺ Speed of computations also generally proportional

- on most architectures, fp32 is 2× faster than fp64
- on some architectures, fp16/bfloat16 up to 16× faster than fp32

lower precision ⇒ faster computations

☺ Power consumption is proportional to the square of the number of mantissa bits

- fp16 (11 bits) consumes 5× less energy than fp32 (24 bits)
- bfloat16 (8 bits) consumes 9× less energy than fp32

lower precision ⇒ greener computations

☹ Errors are proportional to the unit roundoff
lower precision ⇒ lower accuracy

Mixed precision algorithms

Mix several precisions in the same code with the goal of

- Getting the **performance benefits of low precisions**
- While preserving the **accuracy and stability of the high precision**

Terminology varies: Mixed precision, Multiprecision, Adaptive precision, Variable precision, Transprecision, Dynamic precision, ...

Mixed precision algorithms

Mix several precisions in the same code with the goal of

- Getting the **performance benefits of low precisions**
- While preserving the **accuracy and stability of the high precision**

Terminology varies: Mixed precision, Multiprecision, Adaptive precision, Variable precision, Transprecision, Dynamic precision, ...

How to select the right precision for the right variable/operation

- **Precision tuning:** autotuning based on the source code
  - ☺ Does not need any understanding of what the code does
  - ☹ Does not have any understanding of what the code does
- **In linear algebra:** **exploit as much as possible the knowledge we have about the code**

Survey

Acta Numerica (2022), pp. 347–414
doi:10.1017/S0962492922000022

Mixed precision algorithms in numerical linear algebra

Nicholas J. Higham
Department of Mathematics, University of Manchester, Manchester, M13 9PL, UK
E-mail: nick.higham@manchester.ac.uk

Theo Mary
Sorbonne Université, CNRS, LIP6, Paris, F-75005, France
E-mail: theo.mary@lip6.fr

https://bit.ly/mixed-survey

CONTENTS

Linear systems

Solution of $Ax = b$:

- **Direct methods**
  - Robust, black box solvers
  - High time and memory cost for factorization of A
- **Iterative methods**
  - Low time and memory per-iteration cost
  - Convergence is application dependent

Linear systems

Solution of $Ax = b$:

- **Direct methods**
  - Robust, black box solvers
  - High time and memory cost for factorization of A

  $\Rightarrow$ Need fast factorization

- **Iterative methods**
  - Low time and memory per-iteration cost
  - Convergence is application dependent

  $\Rightarrow$ Need good preconditioner

Linear systems

Solution of $Ax = b$:

- **Direct methods**
  - Robust, black box solvers
  - High time and memory cost for factorization of A

  $\Rightarrow$ Need fast factorization

- **Iterative methods**
  - Low time and memory per-iteration cost
  - Convergence is application dependent

  $\Rightarrow$ Need good preconditioner

$\Rightarrow$ **Mixed precision / approximate factorizations** bridge the gap
  - as approximate fast direct methods
  - as high quality preconditioners

Direct solver with LU factorization

Standard method to solve $Ax = b$:

1. Factorize $A = LU$, where L and U are lower and upper triangular
2. Solve $Ly = b$ and $Ux = y$

In uniform precision u, the computed $\widehat{x}$ satisfies

- Backward error $\frac{\|A\widehat{x}-b\|}{\|A\|\|\widehat{x}\|+\|b\|} \leq f(n)\rho_n u = O(u)$
- Forward error $\frac{\|\widehat{x}-x\|}{\|x\|} \leq f(n)\rho_n \kappa(A) u = O(\kappa(A)u)$,
  with $\kappa(A) = \|A\|\|A^{-1}\|$

LU-based refinement (LU-IR)

Factorize $A = LU$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$
repeat
    $r_i = b - Ax_i$
    Solve $Ad_i = r_i$ via $d_i = U^{-1}(L^{-1}r_i)$
    $x_{i+1} = x_i + d_i$
until converged

LU-based refinement (LU-IR)

Factorize $A = LU$ in precision $\mathbf{u}$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$ in precision $\mathbf{u}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}^2$
  Solve $Ad_i = r_i$ via $d_i = U^{-1}(L^{-1}r_i)$ in precision $\mathbf{u}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

Wilkinson (1948) Moler (1967)

Assuming $\kappa(A)\mathbf{u} < 1$:

- Backward error $\frac{\|A\hat{x}-b\|}{\|A\|\|\hat{x}\|+\|b\|} = O(\mathbf{u})$
- Forward error $\frac{\|\hat{x}-x\|}{\|x\|} = O(\mathbf{u})$

LU-IR with fp32 LU

Factorize $A = LU$ in precision $\mathbf{u_f}$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$ in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}$
  Solve $Ad_i = r_i$ via $d_i = U^{-1}(L^{-1}r_i)$ in precision $\mathbf{u_f}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

with $\mathbf{u_f} \equiv$ fp32 and $\mathbf{u} \equiv$ fp64

Langou et al (2006) Buttari et al (2007) Baboulin et al (2009)

LU-IR with fp32 LU

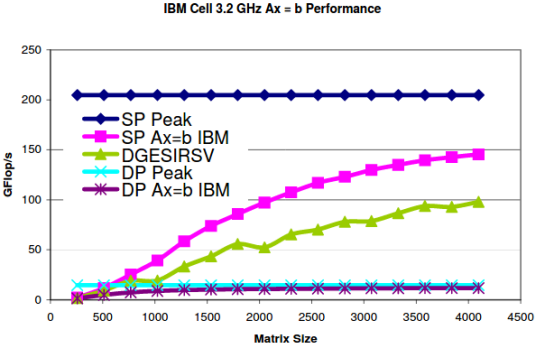
Factorize $A = LU$ in precision $\mathbf{u_f}$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$ in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}$
  Solve $Ad_i = r_i$ via $d_i = U^{-1}(L^{-1}r_i)$ in precision $\mathbf{u_f}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

with $\mathbf{u_f} \equiv$ fp32 and $\mathbf{u} \equiv$ fp64

Langou et al (2006) Buttari et al (2007) Baboulin et al (2009)

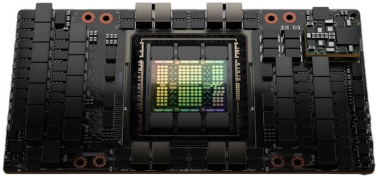
- For $n \times n$ matrices:
  - $O(n^3)$ flops in fp32
  - $O(n^2)$ flops per iteration in fp64
- Assuming $\kappa(A)\mathbf{u_f} < 1$:
  - Backward error $\frac{\|A\hat{x}-b\|}{\|A\|\|\hat{x}\|+\|b\|} = O(\mathbf{u})$
  - Forward error $\frac{\|\hat{x}-x\|}{\|x\|} = O(\kappa(A)\mathbf{u})$

LU-IR with CELL processor



CELL processor (2006–2008)
fp64 peak: 21 GFLOPS
fp32 peak: 205 GFLOPS
$\Rightarrow$ 10$\times$ speedup!

NVIDIA GPUs



NVIDIA Hopper (H100) GPU

Peak performance (TFLOPS)

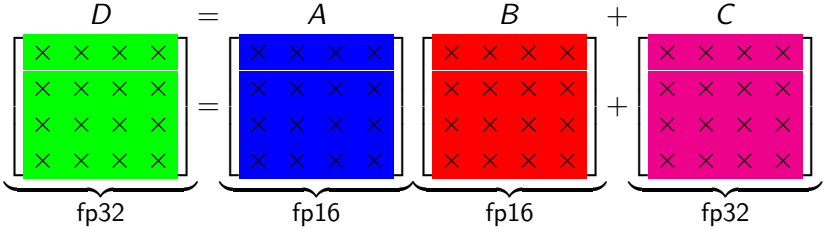
	P100 2016	V100 2018	A100 2020	H100 2022
fp64	5	8	10 → 20	33 → 67
fp32	10	16	20	67
tfloat32	--	--	160	495
fp16	20	125	40 → 320	134 → 990
bfloat16	--	--	40 → 320	134 → 990
fp8	--	--	--	1979

with tensor cores

 Since A100, 16-bit arithmetic is 16× faster than 32-bit

NVIDIA GPU tensor cores

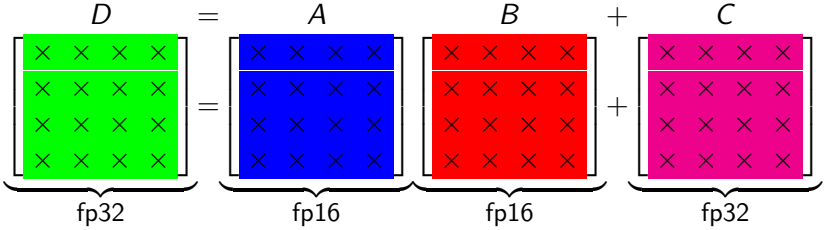
Tensor cores units available on NVIDIA GPUs carry out a fixed size (e.g., 4×4) matrix multiplication :



- **Performance boost vs fp32**: 8–16$\times$ speedup vs fp32

NVIDIA GPU tensor cores

Tensor cores units available on NVIDIA GPUs carry out a fixed size (e.g., 4×4) matrix multiplication :



- **Performance boost vs fp32**: 8–16× speedup vs fp32
- **Accuracy boost vs fp16**: let $C = AB$, with $A \in \mathbb{R}^{m \times n}$, $B \in \mathbb{R}^{n \times p}$, the computed $\widehat{C}$ satisfies

$$|\widehat{C} - C| \lesssim c_n |A||B|, \quad c_n = \begin{cases} nu_{16} & \text{(fp16)} \\ 2u_{16} + nu_{32} & \text{(tensor cores)} \\ nu_{32} & \text{(fp32)} \end{cases}$$

Blanchard, Higham, Lopez, M., Pranesh (2020)

Block LU factorization

- Block version to use matrix–matrix operations

```
for k = 1: n/b do
    Factorize L_kk U_kk = A_kk    (with unblocked alg.)
    for i = k + 1: n/b do
        Solve L_ik U_kk = A_ik and L_kk U_ki = A_ki for L_ik and U_ki
    end for
    for i = k + 1: n/b do
        for j = k + 1: n/b do

            A_ij ← A_ij − L̃_ik Ũ_kj
        end for
    end for
end for
```

Block LU factorization with tensor cores

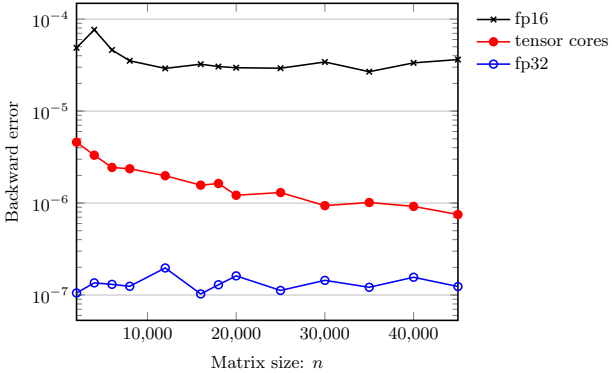
- Block version to use matrix–matrix operations
- $O(n^3)$ part of the flops done with tensor cores

for $k = 1 : n/b$ **do**
  Factorize $L_{kk} U_{kk} = A_{kk}$ (with unblocked alg.)
  for $i = k+1 : n/b$ **do**
    Solve $L_{ik} U_{kk} = A_{ik}$ and $L_{kk} U_{ki} = A_{ki}$ for L_{ik} and U_{ki}
  end for
  for $i = k+1 : n/b$ **do**
    for $j = k+1 : n/b$ **do**
      $\widetilde{L}_{ik} \leftarrow \mathrm{fl}_{16}(L_{ik})$ and $\widetilde{U}_{ki} \leftarrow \mathrm{fl}_{16}(U_{ki})$
      $A_{ij} \leftarrow A_{ij} - \widetilde{L}_{ik} \widetilde{U}_{kj}$ using tensor cores
    end for
  end for
end for

LU factorization with tensor cores

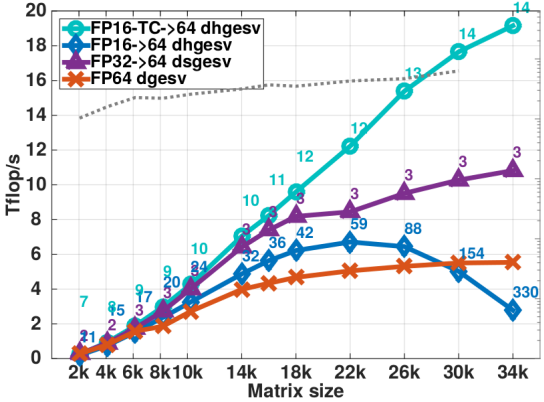
Error analysis for LU follows from matrix multiplication analysis and gives same bounds to first order Blanchard et al. (2020)

Standard fp16	Tensor cores	Standard fp32
nu_{16}	$2u_{16} + nu_{32}$	nu_{32}



Impact on iterative refinement

Results from Haidar et al. (2018)

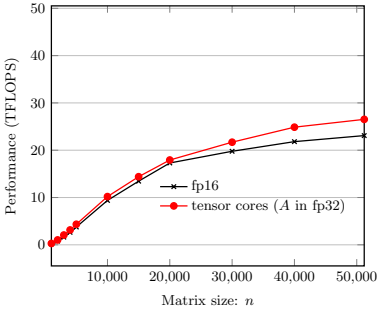


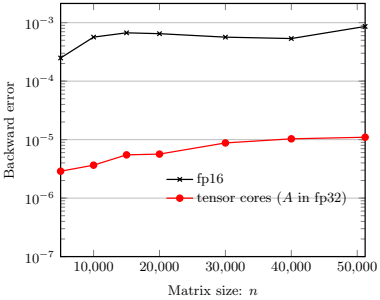
- TC accuracy boost can be critical!
- TC performance suboptimal here ⇒ why?

LU factorization is memory bound

- LU factorization is traditionally a compute-bound operation...
- With Tensor Cores, flops are 8–16× faster
- Matrix is stored in fp32 ⇒ **data movement is unchanged**

⇒ LU with tensor cores becomes memory-bound !

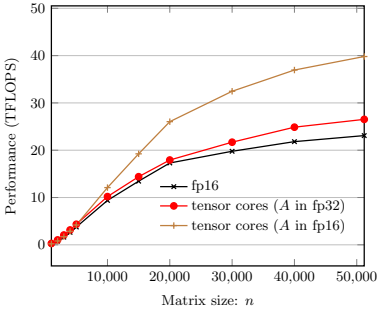


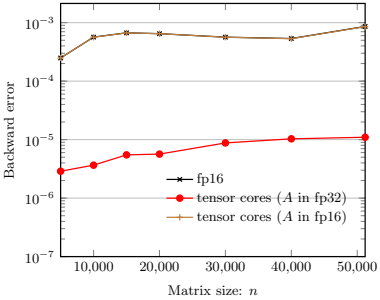


LU factorization is memory bound

- LU factorization is traditionally a compute-bound operation...
- With Tensor Cores, flops are 8–16× faster
- Matrix is stored in fp32 ⇒ **data movement is unchanged**

⇒ LU with tensor cores becomes memory-bound !





- Idea: store matrix in fp16
- Problem: huge accuracy loss, tensor cores accuracy boost completely negated

Reducing data movement

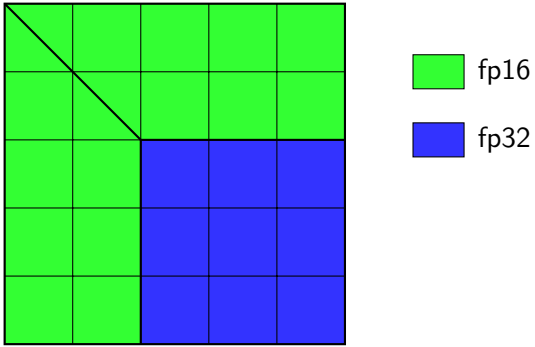
Two ingredients to reduce data movement with no accuracy loss:

Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

1. Mixed fp16/fp32 representation

Matrix after 2 steps:

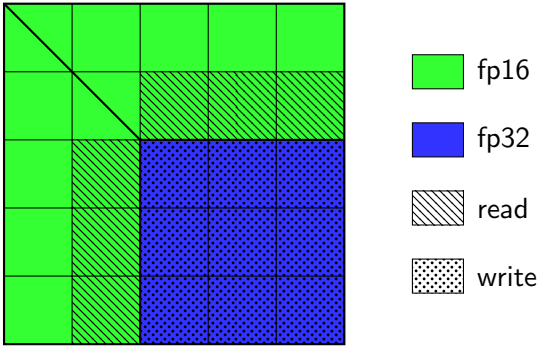


Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

1. Mixed fp16/fp32 representation

Matrix after 2 steps:

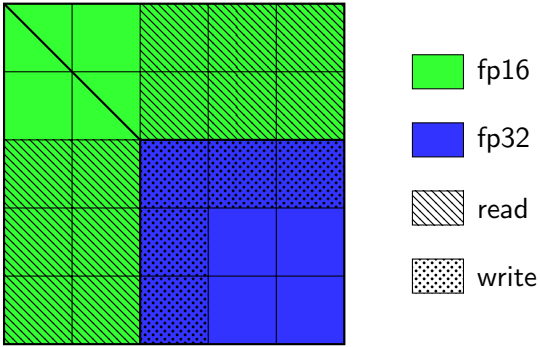


Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

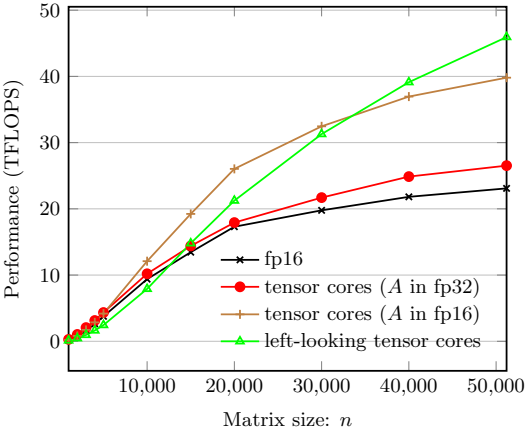
1. Mixed fp16/fp32 representation
2. Right-looking $\rightarrow$ left-looking factorization

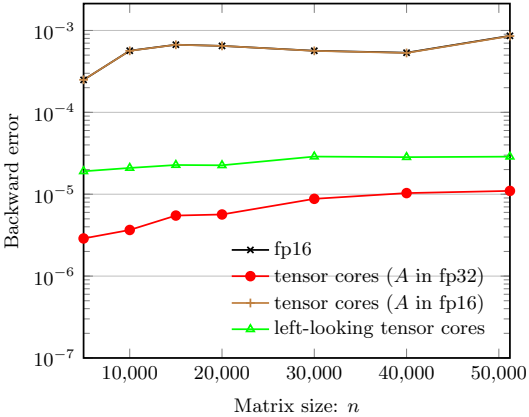
Matrix after 2 steps:



$O(n^3)$ fp32 $+$ $O(n^2)$ fp16 $\rightarrow$ $O(n^2)$ fp32 $+$ $O(n^3)$ fp16

Experimental results





- Nearly **50 TFLOPS** without significantly impacting accuracy
  Lopez and M. (2020)
- Even more critical on A100:
  50 TFLOPS (A in fp32) → **175 TFLOPS** (A in fp16+left-looking)

Challenges of fp16 LU

Use of fp16 presents two risks:

- Overflow/underflow in the LU factors
  - $\||L||U|\| \leq f(n)\rho_n\|A\| \Rightarrow$ even if A fits in the range, its LU factors may not
  - Higham, Pranesh, Zounon (2019) : two-sided diagonal scaling $A' \leftarrow D_r A D_c$ so that $\|A\| \leq c$
  - To minimize underflow and better utilize the range of fp16, helpful to take c as close as possible to maximum safe value
  - Zounon et al. (2020) : appearance of subnormal numbers (in fp32) can lead to slowdowns if they are not flushed to zero
- Loss of positive definiteness
  - Rounding a posdef A to fp16 might make it indefinite $\Rightarrow$ Cholesky factorization breaks down
  - Higham & Pranesh (2021) : factorize $A + \sigma D$ instead ($D = \mathrm{diag}(A)$, $\sigma = O(u_{16})$)

Three-precision LU-IR

Factorize $A = LU$ in precision $\mathbf{u_f}$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$ in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}_r$
  Solve $Ad_i = r_i$ via $d_i = U^{-1}(L^{-1}r_i)$ in precision $\mathbf{u_f}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

e.g., with $\mathbf{u_f} \equiv$ fp16, $\mathbf{u} \equiv$ fp32, and $\mathbf{u_r} \equiv$ fp64

Carson and Higham (2018)

Three-precision LU-IR

Factorize $A = LU$ in precision $\mathbf{u_f}$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$ in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}_r$
  Solve $Ad_i = r_i$ via $d_i = U^{-1}(L^{-1}r_i)$ in precision $\mathbf{u_f}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

e.g., with $\mathbf{u_f} \equiv$ fp16, $\mathbf{u} \equiv$ fp32, and $\mathbf{u_r} \equiv$ fp64

Carson and Higham (2018)

Assuming $\kappa(A)\mathbf{u_f} < 1$:

- Backward error $\frac{\|A\widehat{x}-b\|}{\|A\|\|\widehat{x}\|+\|b\|} = O(\mathbf{u})$
- Forward error $\frac{\|\widehat{x}-x\|}{\|x\|} = O(\mathbf{u} + \kappa(A)\mathbf{u_r})$

Three-precision LU-IR is as general (as modular) as possible

GMRES-based IR (GMRES-IR)

Factorize $A = LU$ in precision $\mathbf{u_f}$
Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$ in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}_r$
  Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ with GMRES in precision $\mathbf{u}$
    with products with $U^{-1}L^{-1}A$ in precision $\mathbf{u}^2$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

Carson and Higham (2017)

- **Replace LU solver by preconditioned GMRES:**
  - GMRES can be asked to converge to accuracy $\mathbf{u} \ll \mathbf{u_f}$
  - $\kappa(\widetilde{A})$ often smaller than $\kappa(A)$

  $\Rightarrow$ $\widetilde{A}d_i = \widetilde{r}_i$ is solved with accuracy $\phi_i = \kappa(\widetilde{A})\mathbf{u}$
  - Convergence condition improved from $\kappa(A)\mathbf{u_f} < 1$ to $\kappa(\widetilde{A})\mathbf{u} < 1$
- **The catch**: the matrix–vector products are with $\widetilde{A} = U^{-1}L^{-1}A$, introduce an extra $\kappa(A)$ unless performed in higher precision

LU-IR vs GMRES-IR

	$\mathbf{u_f}$	$\mathbf{u}$	$\mathbf{u_r}$	max $\kappa(A)$	Forward error
LU-IR	fp32	fp64	fp128	10^8	10^{-16}
GMRES-IR	fp32	fp64	fp128	10^{16}	10^{-16}
LU-IR	fp16	fp64	fp128	10^3	10^{-16}
GMRES-IR	fp16	fp64	fp128	10^{11}	10^{-16}

GMRES-IR can handle much more ill-conditioned matrices.

LU-IR vs GMRES-IR

	$\mathbf{u_f}$	$\mathbf{u}$	$\mathbf{u_r}$	max $\kappa(A)$	Forward error
LU-IR	fp32	fp64	fp128	10^8	10^{-16}
GMRES-IR	fp32	fp64	fp128	10^{16}	10^{-16}
LU-IR	fp16	fp64	fp128	10^3	10^{-16}
GMRES-IR	fp16	fp64	fp128	10^{11}	10^{-16}

GMRES-IR can handle much more ill-conditioned matrices.
However: LU solves are performed in precision $\mathbf{u}^2$ instead of $\mathbf{u_f}$
$\Rightarrow$ **practical limitation**

Rethinking GMRES-IR

- Goal: solve $Ad_i = r_i$ with GMRES and bound $\phi_i = \|\widehat{d_i} - d_i\|/\|d_i\|$
  - In what precision do we really need to run GMRES?
  - How much extra precision is really needed in the matvec products?

Solve $Ax_1 = b$ by LU factorization in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u_r}$
  Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ with GMRES in precision $\mathbf{u}$
    except products with $U^{-1}L^{-1}A$ in precision $\mathbf{u}^2$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

Rethinking GMRES-IR

- Goal: solve $Ad_i = r_i$ with GMRES and bound $\phi_i = \|\widehat{d_i} - d_i\|/\|d_i\|$
  - In what precision do we really need to run GMRES?
  - How much extra precision is really needed in the matvec products?

Solve $Ax_1 = b$ by LU factorization in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u_r}$
  Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ with GMRES in precision $\mathbf{u}$
    except products with $U^{-1}L^{-1}A$ in precision $\mathbf{u}^2$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

Rethinking GMRES-IR

- Goal: solve $Ad_i = r_i$ with GMRES and bound $\phi_i = \|\widehat{d_i} - d_i\|/\|d_i\|$
  - In what precision do we really need to run GMRES?
  - How much extra precision is really needed in the matvec products?

> Solve $Ax_1 = b$ by LU factorization in precision $\mathbf{u_f}$
> **repeat**
> $r_i = b - Ax_i$ in precision $\mathbf{u_r}$
> Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ with GMRES in precision $\mathbf{u_g}$
> except products with $U^{-1}L^{-1}A$ in precision $\mathbf{u_p}$
> $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
> **until** converged

Relax the requirements on the GMRES precisions: run at precision $\mathbf{u_g} \leq \mathbf{u}$ with matvecs at precision $\mathbf{u_p} \leq \mathbf{u}^2$

$\Rightarrow$ **FIVE precisions** in total!

What can we say about the convergence of this GMRES-IR5?

Two-precision GMRES

- **Unpreconditioned** GMRES in precision $\mathbf{u_g}$ for $Ax = b$:
  - Backward error of order $\mathbf{u_g}$ Paige, Rozloznik, Strakos (2006)
  - Forward error of order $\kappa(A)\mathbf{u_g}$

- **Two-precision preconditioned** GMRES for $\widetilde{A}x = b$:
  - Backward error of order $\kappa(A)\mathbf{u_p} + \mathbf{u_g}$
  - Forward error of order $\kappa(\widetilde{A})\big(\kappa(A)\mathbf{u_p} + \mathbf{u_g}\big)$
  - $\kappa(\widetilde{A}) \leq (1 + \kappa(A)\mathbf{u_f})^2$

Side-result: generalization of the backward stability of GMRES to a preconditioned two-precision GMRES

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2021)

Five-precision GMRES-IR

Solve $Ax_1 = b$ by LU factorization in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u_r}$
  Solve $U^{-1}L^{-1}Ad_i = U^{-1}L^{-1}r_i$ with GMRES in precision $\mathbf{u_g}$
    except products with $U^{-1}L^{-1}A$ in precision $\mathbf{u_p}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

Theorem (convergence of GMRES-IR5)

Under the condition $(\mathbf{u_g} + \kappa(A)\mathbf{u_p})\kappa(A)^2\mathbf{u_f}^2 < 1$, the forward error converges to its limiting accuracy

$$\frac{\|\widehat{x} - x\|}{\|x\|} \leq \mathbf{u_r}\kappa(A) + \mathbf{u}$$

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2021)

Meaningful combinations

With five arithmetics (fp16, bfloat16, fp32, fp64, fp128) there are over **3000 different combinations** of GMRES-IR5!

They are not all relevant !

Meaningful combinations: those where none of the precisions can be lowered without worsening either the limiting accuracy or the convergence condition.

Filtering rules

- $\mathbf{u}^2 \leq \mathbf{u_r} \leq \mathbf{u} \leq \mathbf{u_f}$
- $\mathbf{u_p} \leq \mathbf{u_g}$
- $\mathbf{u_p} < \mathbf{u_f}$
- $\mathbf{u_p} < \mathbf{u}$, $\mathbf{u_p} = \mathbf{u}$, $\mathbf{u_p} > \mathbf{u}$ all possible
- $\mathbf{u_g} \geq \mathbf{u}$
- $\mathbf{u_g} < \mathbf{u_f}$, $\mathbf{u_g} = \mathbf{u_f}$, $\mathbf{u_g} > \mathbf{u_f}$ all possible

Performance–robustness tradeoff

Meaningful combinations of GMRES-IR5 for $\mathbf{u_f} \equiv$ fp16 and $\mathbf{u} \equiv$ fp64

$\mathbf{u_g}$	$\mathbf{u_p}$	Convergence Condition $\max(\kappa(A))$
LU-IR		2×10^3
bfloat16	fp32	3×10^4
fp16	fp32	4×10^4
fp16	fp64	9×10^4
fp32	fp64	8×10^6
fp64	fp64	3×10^7
fp64	fp128	2×10^{11}

Five combinations between LU-IR and Carson & Higham's GMRES-IR $\Rightarrow$ More **flexible** precisions choice to fit at best the **hardware constraints** and the **problem difficulty**.

Right-preconditioned / flexible GMRES

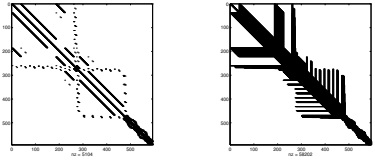
- What about solving $AU^{-1}L^{-1}y = b$ with **flexible GMRES** ?
- Stability not provable as easily/unconditionally as in the left-preconditioned case
  Arioli and Duff (2009) Carson and Daužickaitė (2023)
  ...but works very well in practice

$\Rightarrow$ Can relax the need to apply preconditioner in higher precision, at the cost of using flexible GMRES

Sparse matrices

Fill-in: $\mathrm{nnz}(A) \ll \mathrm{nnz}(LU)$

Original matrix *Factorized matrix*



- ☺ Memory footprint of high-precision copy of A negligible $\Rightarrow$ modern IR saves memory (**not** the case for dense systems!)
- ☺ Matvecs with A cheap compared with LU solves $\Rightarrow$ can afford to compute residual very accurately!
- ☹ Relative weight of refinement is higher: $O(n^2)$ vs $O(n^3)$ for dense $\Rightarrow$ $O(n^{4/3})$ vs $O(n^2)$ for sparse (at best, even worse for 2D problems) $\Rightarrow$ less room to amortize iterations
- ☹ Higher weight of symbolic operations

Sparse LU-IR vs GMRES-IR

Results with the MUMPS solver

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2023)

Matrix	time (s)			memory (GB)		
	fp64	fp32 +LU-IR	fp32 +GMRES-IR	fp64	fp32 +LU-IR	fp32 +GMRES-IR
ElectroPhys10M	265.2	**154.0**	166.5	272.0	**138.0**	171.3
Bump_2911	205.4	**129.3**	144.5	135.7	**68.4**	77.8
DrivAer6M	91.8	**67.6**	77.9	81.6	**41.7**	52.9
Queen_4147	284.2	**165.2**	184.7	178.0	**89.8**	114.5
tminlet3M	294.5	**136.2**	157.9	241.1	**121.0**	169.9
perf009ar	**46.1**	57.5	52.0	55.6	**28.9**	38.1
elasticity-3d	156.7	—	**118.6**	153.0	—	**103.6**
lfm_aug5M	536.2	**254.5**	269.3	312.0	**157.0**	187.5
Long_Coup_dt0	67.2	**46.6**	49.0	52.9	**26.7**	33.1
CarBody25M	**62.9**	—	109.8	77.6	—	**54.3**
thmgaz	97.6	**65.4**	79.8	192.0	**97.7**	141.7

- Up to 2× **time and memory reduction**, even for ill-conditioned problems
- GMRES-IR usually more expensive than LU-IR, but more robust

Cheaper preconditioners than LU

Compute $M^{-1} \approx A^{-1}$ and $x_1 = M^{-1}b$ in precision $\mathbf{u_f}$
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u_r}$
  Solve $M^{-1}Ad_i = M^{-1}r_i$ with GMRES in precision $\mathbf{u_g}$
    except products with $M^{-1}A$ in precision $\mathbf{u_p}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

- Replace $U^{-1}L^{-1}$ with general M^{-1}
- **Equivalent to restarted GMRES**

Cheaper preconditioners than LU

Initialize x_1
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u_r} \equiv \mathbf{u}_{\text{high}}$
  Solve $Ad_i = r_i$ with GMRES in precision $\mathbf{u_g} \equiv \mathbf{u}_{\text{low}}$

  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u} \equiv \mathbf{u}_{\text{high}}$
until converged

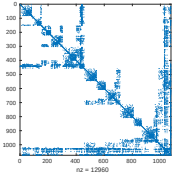
- Replace $U^{-1}L^{-1}$ with general M^{-1}
- **Equivalent to restarted GMRES**
- No preconditioner ($M = I$): mixed precision inner–outer scheme

  Turner and Walker (1992) Buttari et al. (2008)
  Lindquist et al. (2020) Loe et al. (2021)

Sparsification (dropping)

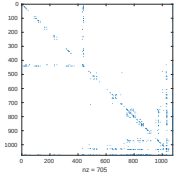
Dropping: replace with zero any value sufficiently small

$$|a_{ij}| \leq \epsilon \|A\| \quad \Rightarrow \quad a_{ij} \leftarrow 0$$



sparse A

$\xrightarrow{drop}$

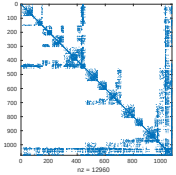


sparser A

Sparsification (dropping)

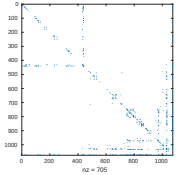
Dropping: replace with zero any value sufficiently small

$$|a_{ij}| \leq \epsilon \|A\| \quad \Rightarrow \quad a_{ij} \leftarrow 0$$

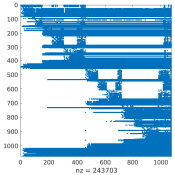


sparse A

$\xrightarrow{drop}$

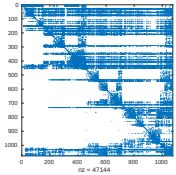


sparser A



LU factors

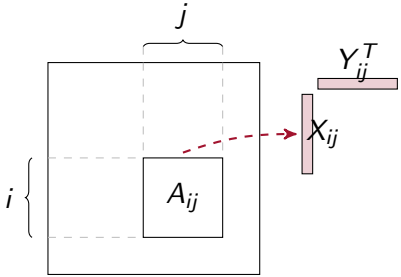
$\xrightarrow{drop}$

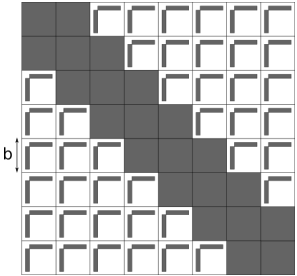


incomplete LU

Data sparsification (low-rank approximations)

Low-rank compression: given $A = U\Sigma V^T$, if we truncate singular vectors associated with $\sigma_i \leq \epsilon$, we obtain $\widetilde{A}$ such that $\|\widetilde{A} - A\| \leq \epsilon$





Block Low Rank

Amestoy et al. (2015, 2017, 2019)

Compress A_{ij} such that $\|\widetilde{A}_{ij} - A_{ij}\| \leq \epsilon\|A\|$:

- If $\|A_{ij}\| \leq \epsilon\|A\| \Rightarrow A_{ij} \leftarrow 0$ (drop block)
- otherwise replace A_{ij} with $\widetilde{A}_{ij} = X_{ij}Y_{ij}^T$

BLR + IR

Error analysis: replace $\mathbf{u_f}$ by $\mathbf{u_f} + \epsilon$ in the convergence conditions

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2023)

Example on tminlet3M matrix

fp64 MUMPS reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

	time (s)		memory (GB)	
	LU-IR	GMRES-IR	LU-IR	GMRES-IR
fp32 MUMPS	136.2	157.9	121.0	169.9

BLR + IR

Error analysis: replace $\mathbf{u_f}$ by $\mathbf{u_f} + \epsilon$ in the convergence conditions

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2023)

Example on tminlet3M matrix

fp64 MUMPS reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

	time (s)		memory (GB)	
	LU-IR	GMRES-IR	LU-IR	GMRES-IR
fp32 MUMPS	136.2	157.9	121.0	169.9
$\epsilon = 10^{-8}$	149.7	165.3	114.0	161.9

BLR + IR

Error analysis: replace $\mathbf{u_f}$ by $\mathbf{u_f} + \epsilon$ in the convergence conditions

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2023)

Example on tminlet3M matrix

fp64 MUMPS reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

	time (s)		memory (GB)	
	LU-IR	GMRES-IR	LU-IR	GMRES-IR
fp32 MUMPS	136.2	157.9	121.0	169.9
$\epsilon = 10^{-8}$	149.7	165.3	114.0	161.9
$\epsilon = 10^{-6}$	**88.3**	98.8	82.4	93.8

BLR + IR

Error analysis: replace $\mathbf{u_f}$ by $\mathbf{u_f} + \epsilon$ in the convergence conditions

Amestoy, Buttari, Higham, L'Excellent, M., Vieublé (2023)

Example on tminlet3M matrix

fp64 MUMPS reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

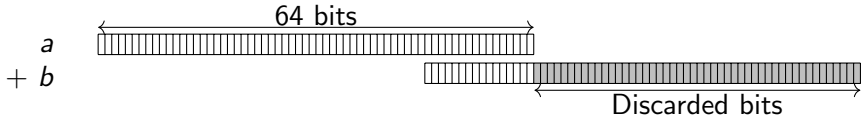
	time (s)		memory (GB)	
	LU-IR	GMRES-IR	LU-IR	GMRES-IR
fp32 MUMPS	136.2	157.9	121.0	169.9
$\epsilon = 10^{-8}$	149.7	165.3	114.0	161.9
$\epsilon = 10^{-6}$	**88.3**	98.8	82.4	93.8
$\epsilon = 10^{-4}$	—	105.6	—	**70.9**

- GMRES-IR allows to push BLR further!

Adaptive precision sparsification

- Sparsification only deals in absolutes:
  either we keep the data at full accuracy, or we discard it completely!
- We need **a new paradigm** that uses multiple, gradual levels of approximation

⇒ Adaptive precision sparsification



Adapt the precisions to the data at hand by storing and computing "less important" (usually meaning smaller) data in lower precision

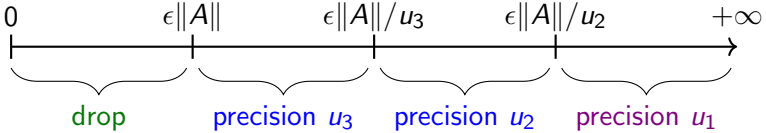
Adaptive precision SpMV

- **Goal:** compute the SpMV $y = Ax$ with accuracy ϵ using q precisions $u_1 \leq \epsilon < u_2 < \ldots < u_q$
- Split elements a_{ij} on each row i into q buckets $B_{i1}, \ldots, B_{iq}$, where bucket B_{ik} uses precision u_k

Adaptive precision SpMV

- **Goal:** compute the SpMV $y = Ax$ with accuracy ϵ using q precisions $u_1 \leq \epsilon < u_2 < \ldots < u_q$
- Split elements a_{ij} on each row i into q buckets $B_{i1}, \ldots, B_{iq}$, where bucket B_{ik} uses precision u_k
- How should we build the buckets?

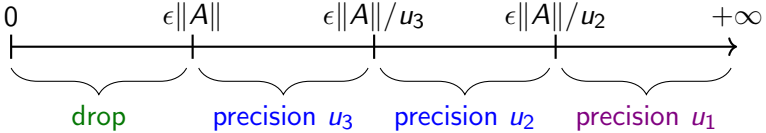
$$\begin{cases} |a_{ij}| \leq \epsilon \|A\| & \Rightarrow \text{ drop} \\ |a_{ij}| \in [\epsilon\|A\|/u_{k+1}, \epsilon\|A\|/u_k) & \Rightarrow \text{ place in } B_{ik} \\ |a_{ij}| > \epsilon\|A\|/u_2 & \Rightarrow \text{ place in } B_{i1} \end{cases}$$



Adaptive precision SpMV

- **Goal:** compute the SpMV $y = Ax$ with accuracy ϵ using q precisions $u_1 \leq \epsilon < u_2 < \ldots < u_q$
- Split elements a_{ij} on each row i into q buckets $B_{i1}, \ldots, B_{iq}$, where bucket B_{ik} uses precision u_k
- How should we build the buckets?

$$\begin{cases} |a_{ij}| \leq \epsilon \|A\| & \Rightarrow \text{ drop} \\ |a_{ij}| \in [\epsilon\|A\|/u_{k+1}, \epsilon\|A\|/u_k) & \Rightarrow \text{ place in } B_{ik} \\ |a_{ij}| > \epsilon\|A\|/u_2 & \Rightarrow \text{ place in } B_{i1} \end{cases}$$



- **Theorem**: the computed $\widehat{y}$ satisfies $\|\widehat{y} - y\| \leq c\epsilon\|A\|\|x\|$

Graillat, Jézéquel, M., Molina (2022)

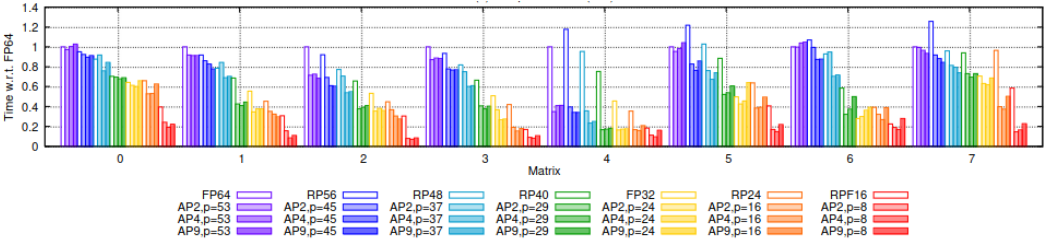
Adaptive precision SpMV: results

- Experimental results on 36-core computer
- Adaptive algorithm uses fp64, fp32, and dropping and $\epsilon = 2^{-53}$
- Comparison vs uniform fp64

Matrix	adaptive storage (% of fp64)	adaptive time (% of fp64)	backward error	
			fp64	adaptive
Hook_1498	97%	99%	3e-16	7e-16
Flan_1565	78%	82%	3e-16	3e-16
Long_Coup_dt0	75%	83%	5e-16	2e-15
imagesensor	15%	56%	2e-16	9e-16
power9	16%	37%	1e-16	4e-16
nv2	19%	26%	2e-16	2e-15

Seven-precision SpMV

Emulated formats				
	Bits			
Format	Signif.(t)	Exponent	Range	$u = 2^{-t}$
bf16	8	8	$10^{\pm 38}$	4×10^{-3}
fp24	16	8	$10^{\pm 38}$	2×10^{-5}
fp32	24	8	$10^{\pm 38}$	6×10^{-8}
fp40	29	11	$10^{\pm 308}$	2×10^{-9}
fp48	37	11	$10^{\pm 308}$	8×10^{-12}
fp56	45	11	$10^{\pm 308}$	3×10^{-14}
fp64	53	11	$10^{\pm 308}$	1×10^{-16}



Graillat, Jézéquel, M., Molina, Mukunoki (2023)

GMRES-based iterative refinement

GMRES

$r = b - Ax_0$
$\beta = \|r\|_2$
$q_1 = r/\beta$
for $k = 1, 2, \ldots$ **do**
  $y = Aq_k$
  for $j = 1 : k$ **do**
    $h_{jk} = q_j^T y$
    $y = y - h_{jk} q_j$
  end for
  $h_{k+1,k} = \|y\|_2$
  $q_{k+1} = y / h_{k+1,k}$
  Solve $\min_{c_k} \|Hc_k - \beta e_1\|_2$.
  $x_k = x_0 + Q_k c_k$
end for

GMRES-IR

for $i = 1, 2, \ldots$ **do**
  $r_i = b - Ax_{i-1}$
  Solve $Ad_i = r_i$ by GMRES
  $x_i = x_{i-1} + d_i$
end for

GMRES-based iterative refinement

GMRES

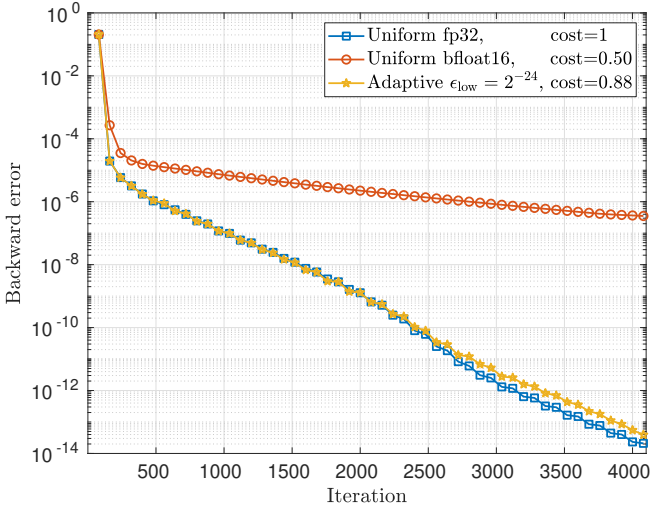
$r = b - Ax_0$
$\beta = \|r\|_2$
$q_1 = r/\beta$
for $k = 1, 2, \ldots$ **do**
  $y = Aq_k \rightarrow \epsilon_{\text{low}}$
  for $j = 1 : k$ **do**
    $h_{jk} = q_j^T y$
    $y = y - h_{jk} q_j$
  end for
  $h_{k+1,k} = \|y\|_2$
  $q_{k+1} = y / h_{k+1,k}$
  Solve $\min_{c_k} \|Hc_k - \beta e_1\|_2$.
  $x_k = x_0 + Q_k c_k$
end for

GMRES-IR

for $i = 1, 2, \ldots$ **do**
  $r_i = b - Ax_{i-1} \rightarrow \epsilon_{\text{high}}$
  Solve $Ad_i = r_i$ by GMRES
  $x_i = x_{i-1} + d_i$
end for

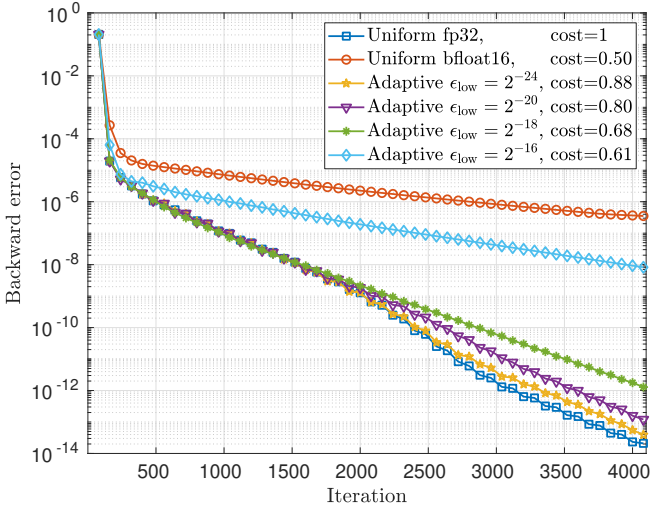
GMRES-IR with adaptive precision SpMV

ML_Laplace ($\epsilon_{\text{high}} = 2^{-53}$, restart $= 80$, Jacobi preconditioner)
3 precisions (fp64, fp32, bfloat16) + dropping

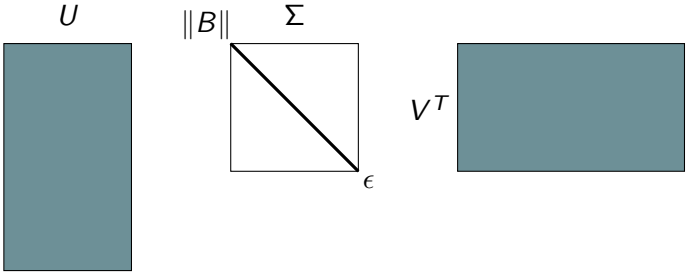


GMRES-IR with adaptive precision SpMV

ML_Laplace ($\epsilon_{\text{high}} = 2^{-53}$, restart $= 80$, Jacobi preconditioner)
3 precisions (fp64, fp32, bfloat16) + dropping

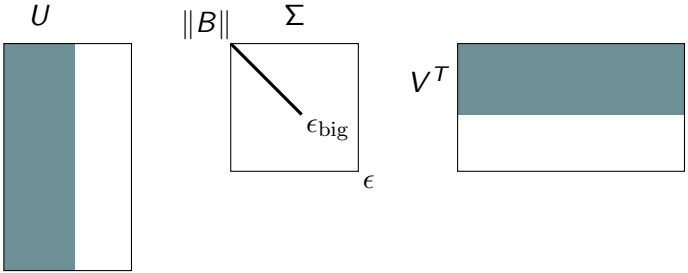


Adaptive precision low rank compression



How to increase low-rank compression?

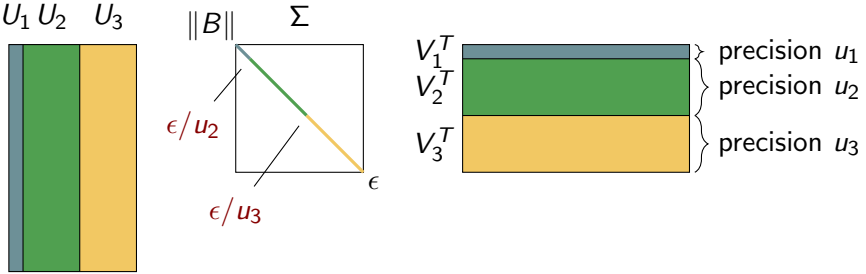
Adaptive precision low rank compression



How to increase low-rank compression?

- Standard approach: increase ϵ to discard more vectors

Adaptive precision low rank compression



How to increase low-rank compression?

- Standard approach: increase ϵ to discard more vectors
- **Adaptive precision compression:** partition U and V into q groups of decreasing precisions $u_1 \leq \epsilon < u_2 < \ldots < u_q$
- Why does it work? $B = \mathbf{B_1} + \mathbf{B_2} + \mathbf{B_3}$ with $|B_i| \leq O(\|\Sigma_i\|)$

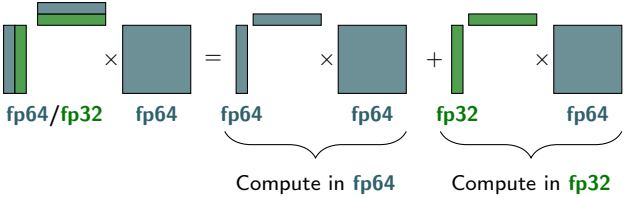
  Amestoy, Boiteau, Buttari, Gerest, Jézéquel, L'Excellent, M. (2021)

Adaptive precision BLR LU factorization

Stability of LU factorization: $\widehat{L}\widehat{U} = A + \Delta A$

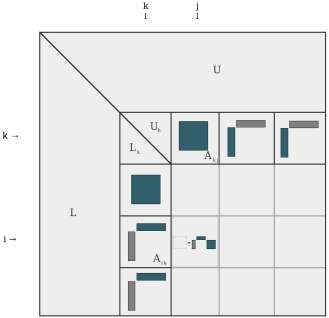
- **Standard LU** (Wilkinson) :
  $\|\Delta A\| \lesssim 3n^3 \rho_n u_1 \|A\|$
- **BLR LU** (Higham & M.) :
  $\|\Delta A\| \lesssim (c_1 \epsilon + c_2 \rho_n u_1)\|A\|$ Higham and M. (2021)
- **Adaptive prec. BLR LU** (this work) :
  $\|\Delta A\| \lesssim (c_1' \epsilon + c_2' \rho_n u_1)\|A\|$

Example of kernel: LR $\times$ matrix multiplication:



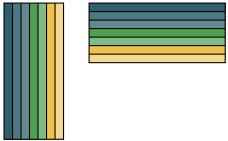
Step k:

- Compute $L_{kk} U_{kk} = A_{kk}$
- Update
  $A_{ij} \leftarrow A_{ij} - (A_{ik} U_{kk}^{-1}) \times (L_{kk}^{-1} A_{kj})$



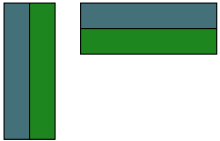
Adaptive precision BLR implementation in MUMPS

1) **Toward storage gains**
   - A large number of precisions (current version: **7 formats**)
   - For storage only



   - A preliminary version has been completed

2) **Toward time gains**
   - Small number of precisions
   - Chosen according to availability in hardware
   - For computations



   - Ongoing development

Adaptive precision BLR: results in MUMPS

Matrix		LU factors size	Total memory	Backward error
thmgaz	fp64 MUMPS	141	194	
	BLR double	95	120	6.4e-14
	BLR mixed	59	86	6.5e-14
knuckle8M	fp64 MUMPS	235	547	
	BLR double	117	281	1.6e-10
	BLR mixed	71	236	7.7e-09
perf009ar	fp64 MUMPS	38	58	
	BLR double	26	36	1.3e-10
	BLR mixed	20	25	1.4e-10

Memory consumption reduced by up to

- 1.7× (LU factors size)
- 1.4× (total memory, including working arrays)

Conclusion

A plausible scenario for solving $Ax = b$ in mixed precision:

Compute adaptive precision BLR LU factorization $A \approx LU$ **at accuracy ϵ_f**

Solve $Ax_1 = b$ with adaptive precision BLR LU solves **at accuracy ϵ_f**

repeat

Compute $r_i = b - Ax_i$ with adaptive precision SpMV **at accuracy ϵ_r**

Solve $Ad_i = r_i$ with adaptive precision GMRES preconditioned by BLR LU factors, using

adaptive precision SpMV **at accuracy ϵ_a**,

adaptive precision BLR LU solves **at accuracy ϵ_p**,

and **precision u_g** for the rest

$x_{i+1} = x_i + d_i$ **(in uniform precision u!)**

until converged

Refinement **Modularity** Adaptivity

Slides

https://bit.ly/NHRmixed

Survey

https://bit.ly/mixed-survey

Thanks! Questions?