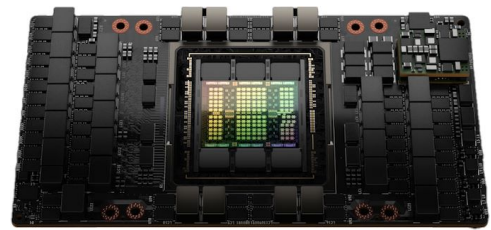
Arithmétique flottante et analyse d'erreur (AFAE)

Lectures 3 & 4: emulation, linear systems

Theo Mary (CNRS)
theo.mary@lip6.fr
https://perso.lip6.fr/Theo.Mary/

M2 course at Sorbonne Université, 2025–2026



Precision emulation

Linear systems

Block low-rank approximations

NVIDIA GPUs

	Accelerator/Co-Processor	Count	System Share (%)	Rmax (GFlops)	Rpeak (GFlops)	Cores
1	NVIDIA Tesla V100	34	6.8	160,351,840	258,591,394	3,042,576
2	NVIDIA A100	28	5.6	334,073,980	501,369,740	3,381,552
3	NVIDIA A100 SXM4 40 GB	17	3.4	182,940,000	256,548,320	1,863,032
4	NVIDIA A100 SXM4 80 GB	12	2.4	69,404,000	81,030,352	622,400
5	NVIDIA Tesla A100 80G	11	2.2	125,864,100	167,922,790	1,088,032
6	AMD Instinct MI250X	10	2	1,753,739,370	2,480,336,322	12,660,800
7	NVIDIA Tesla V100 SXM2	10	2	88,521,490	177,435,765	1,997,584
8	NVIDIA Tesla A100 40G	8	1.6	58,848,600	94,137,490	616,484
9	NVIDIA GH200 Superchip	7	1.4	472,151,000	663,723,750	2,401,488
10	NVIDIA H100	7	1.4	637,532,000	969,686,010	2,238,328
11	Nvidia H100 SXM5 94Gb	6	1.2	76,124,660	137,526,280	371,536
12	NVIDIA Tesla P100	5	1	42,902,620	60,652,584	839,200
13	NVIDIA Volta GV100	4	0.8	269,439,000	362,564,722	4,408,096
14	NVIDIA H100 SXM5 80GB	4	0.8	196,217,000	318,153,330	795,872
15	AMD Instinct MI300A	3	0.6	58,950,000	96,293,683	387,072
16	NVIDIA H100 80GB	2	0.4	12,966,000	20,143,060	45,568
17	Intel Data Center GPU Max 1550	2	0.4	23,334,690	59,843,110	199,872
18	NVIDIA A100 80GB	2	0.4	37,150,000	41,471,280	275,776
19	NVIDIA Tesla K40	2	0.4	7,154,000	12,263,680	145,600
20	Intel Data Center GPU Max	2	0.4	1,029,190,500	2,007,963,420	9,413,888

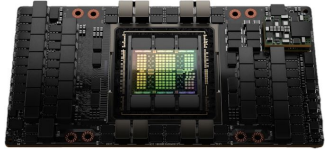
Top 20 accelerators in TOP500

NVIDIA stock price history

Speed vs precision on NVIDIA GPUs

Peak performance (TFLOPS)

	Pascal P100 2016	Volta V100 2018	Ampere A100 2020	Hopper GH200 SXM 2022	Blackwell GB200 2025
fp64	5	8	20	67	40
fp32	10	16	20	67	80
tfloat32	--	--	160	495	1,100
fp16/bfloat16	20	125	320	990	2,250
fp8/int8	--	--	--	2,000	4,500
fp4	--	--	--	--	9,000



NVIDIA Hopper (H100) GPU

fp64/fp8 speed ratio:

- Hopper (2022): 30×
- Blackwell (2025): 112×

The limitations of lower precisions

	Signif. bits	Exp. bits	Range ($f_{\max}/f_{\min}$)	Unit roundoff u
fp128	113	15	$2^{32766} \approx 10^{9863}$	$2^{-114} \approx 1 \times 10^{-34}$
fp64	52	11	$2^{2046} \approx 10^{616}$	$2^{-53} \approx 1 \times 10^{-16}$
fp32	23	8	$2^{254} \approx 10^{76}$	$2^{-24} \approx 6 \times 10^{-8}$
tfloat32	10	8	$2^{254} \approx 10^{76}$	$2^{-11} \approx 5 \times 10^{-4}$
fp16	10	5	$2^{30} \approx 10^{9}$	$2^{-11} \approx 5 \times 10^{-4}$
bfloat16	7	8	$2^{254} \approx 10^{76}$	$2^{-8} \approx 4 \times 10^{-3}$
fp8 (E4M3)	3	4	$2^{15} \approx 3 \times 10^{4}$	$2^{-4} \approx 6 \times 10^{-2}$
fp8 (E5M2)	2	5	$2^{30} \approx 10^{9}$	$2^{-3} \approx 1 \times 10^{-1}$
fp6 (E2M3)	3	2	$2^{3} \approx 8$	$2^{-4} \approx 6 \times 10^{-2}$
fp6 (E3M2)	2	3	$2^{7} \approx 128$	$2^{-3} \approx 0.125$
fp4 (E2M1)	1	2	$2^{3} \approx 8$	$2^{-2} \approx 0.25$

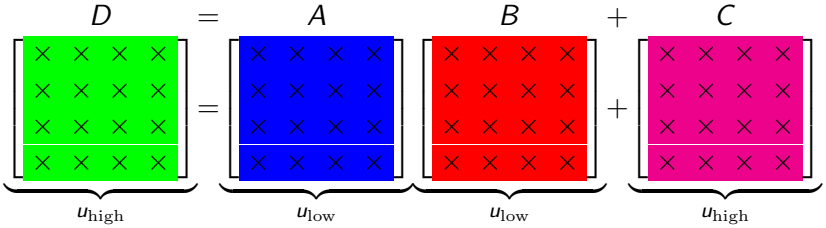
Lower precisions:

- ☺ Faster, consume less memory and energy
- ☹ Lower accuracy and narrower range
- ⇒ Mixed precision algorithms

Standard model of FPA:

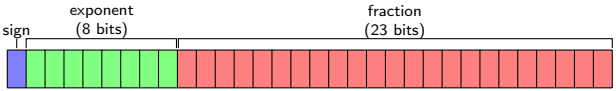
For any x such that $|x| \in [f_{\min}, f_{\max}]$
$$\mathrm{fl}(x) = x(1+\delta), \quad |\delta| \leq u$$

Tensor cores units available on NVIDIA GPUs carry out a mixed precision matrix multiply–accumulate ($u_{\text{high}} \equiv$ fp32 and $u_{\text{low}} \equiv$ fp16/fp8/fp4)



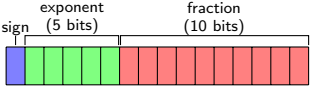
Element-wise multiplication of matrix A and B is performed with at least single precision. When .ctype or .dtype is .f32, **accumulation of the intermediate values is performed with at least single precision**. *When both .ctype and .dtype are specified as .f16, the accumulation is performed with at least half precision.*
The accumulation order, rounding and handling of subnormal inputs is unspecified.

Tensor core precisions



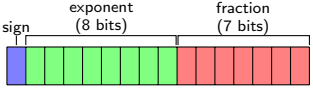
fp32

Range $10^{\pm 38}$, $u = 6 \times 10^{-8}$



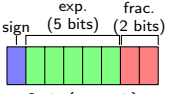
fp16

Range $10^{\pm 5}$, $u = 5 \times 10^{-4}$



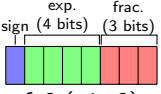
bfloat16

Range $10^{\pm 38}$, $u = 4 \times 10^{-3}$



fp8 (e5m2)

Range $10^{\pm 5}$, $u = 0.125$

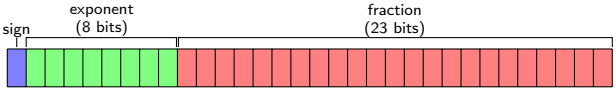


fp8 (e4m3)

Range $[0.015, 448]$, $u = 6 \times 10^{-2}$

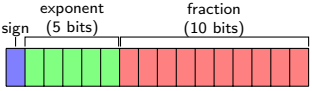
fp4 (e2m1)

Range $[1, 6]$, $u = 0.25$



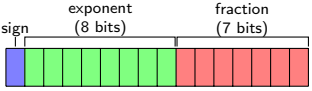
fp32

Range $10^{\pm 38}$, $u = 6 \times 10^{-8}$



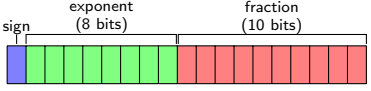
fp16

Range $10^{\pm 5}$, $u = 5 \times 10^{-4}$



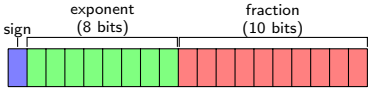
bfloat16

Range $10^{\pm 38}$, $u = 4 \times 10^{-3}$



tfloat32

Range $10^{\pm 38}$, $u = 5 \times 10^{-4}$



tfloat32

Range $10^{\pm 38}$, $u = 5 \times 10^{-4}$

$1 + 8 + 10 = 19$ bits, why is it called tfloat32 ??

> *TF32 uses the same 10-bit mantissa as the half-precision (FP16) math, shown to have more than sufficient margin for the precision requirements of AI workloads. And TF32 adopts the same 8-bit exponent as FP32 so it can support the same numeric range. The combination makes TF32 a great alternative to FP32 for crunching through single-precision math. To validate the accuracy of TF32, we used it to train a broad set of AI networks. All of them have the same convergence-to-accuracy behavior as FP32.*

source: blogs.nvidia.com

Matrix multiplication with tensor cores

This algorithm computes $C = AB$ using tensor cores, where $A, B, C \in \mathbb{R}^{n\times n}$, and returns C in precision u_{high}

$\widetilde{A} \leftarrow \text{fl}_{\text{low}}(A)$ and $\widetilde{B} \leftarrow \text{fl}_{\text{low}}(B)$ (if necessary)
for $i = 1\colon n/b_1$ **do**
  for $j = 1\colon n/b_2$ **do**
    $C_{ij} = 0$
    for $k = 1\colon n/b$ **do**
      Compute $C_{ij} = C_{ij} + \widetilde{A}_{ik}\widetilde{B}_{kj}$ using tensor cores
    end for
  end for
end for

Mixed precision MMA: model and error analysis

- We consider an MMA (matrix multiply–accumulate) unit that computes $C = AB$, $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times q}$, $C \in \mathbb{R}^{m\times q}$, as follows:
- First, we convert A and B to low precision:

$$\widetilde{A} = \mathrm{fl}_{\mathrm{low}}(A) = A + \Delta A, \quad |\Delta A| \le u_{\mathrm{low}}|A|,$$
$$\widetilde{B} = \mathrm{fl}_{\mathrm{low}}(B) = B + \Delta B, \quad |\Delta B| \le u_{\mathrm{low}}|B|.$$

- Second, we compute the product:

$$\begin{aligned}\widehat{C} &= \widetilde{A}\widetilde{B} + \Delta C, \qquad |\Delta C| \lesssim n u_{\mathrm{high}}|\widetilde{A}||\widetilde{B}|,\\ &= AB + \Delta AB + A\Delta B + \Delta A\Delta B + \Delta C\\ &= AB + E, \qquad |E| \le \big(\underbrace{2u_{\mathrm{low}} + u_{\mathrm{low}}^2}_{\text{Conversion}} + \underbrace{n u_{\mathrm{high}}}_{\text{Accumulation}}\big)|A||B|\end{aligned}$$

Mixed precision MMA: model and error analysis

- We consider an MMA (matrix multiply–accumulate) unit that computes $C = AB$, $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times q}$, $C \in \mathbb{R}^{m\times q}$, as follows:
- First, we convert A and B to low precision:

$$\widetilde{A} = \mathrm{fl}_{\mathrm{low}}(A) = A + \Delta A, \quad |\Delta A| \le u_{\mathrm{low}}|A|,$$
$$\widetilde{B} = \mathrm{fl}_{\mathrm{low}}(B) = B + \Delta B, \quad |\Delta B| \le u_{\mathrm{low}}|B|.$$

- Second, we compute the product:

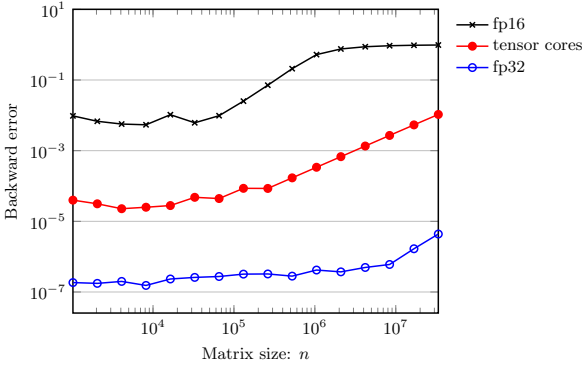
$$\begin{aligned}\widehat{C} &= \widetilde{A}\widetilde{B} + \Delta C, \qquad |\Delta C| \lesssim n u_{\mathrm{high}}|\widetilde{A}||\widetilde{B}|,\\ &= AB + \Delta AB + A\Delta B + \Delta A\Delta B + \Delta C\\ &= AB + E, \qquad |E| \le \big(\underbrace{2u_{\mathrm{low}} + u_{\mathrm{low}}^2}_{\text{Conversion}} + \underbrace{n u_{\mathrm{high}}}_{\text{Accumulation}}\big)|A||B|\end{aligned}$$

Evaluation method	Bound
Standard in precision u_{low}	nu_{low}
Tensor cores	$2u_{\mathrm{low}} + nu_{\mathrm{high}}$

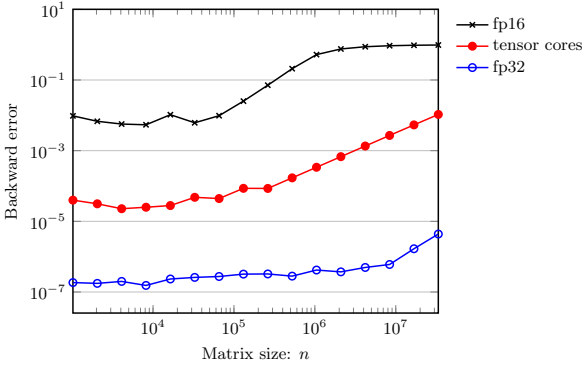
[Blanchard, Higham, Lopez, M., Pranesh, SISC 2020]

$\Rightarrow$ reduction by a factor $\min\big(n/2, u_{\mathrm{low}}/u_{\mathrm{high}}\big)$

Matrix multiplication with tensor cores



Matrix multiplication with tensor cores



Warning!

- NVIDIA tensor cores do not conform to the IEEE standard
- The additions in the product AB are performed with the **round-towards-zero** rounding mode Fasi, Higham, Mikaitis, Pranesh (2021)

fp32 emulation: a multiword approach

- Step a) compute the multiword decompositions

$$A \approx \sum_{i=0}^{s-1} A_i \quad \text{and} \quad B \approx \sum_{j=0}^{s-1} B_j$$

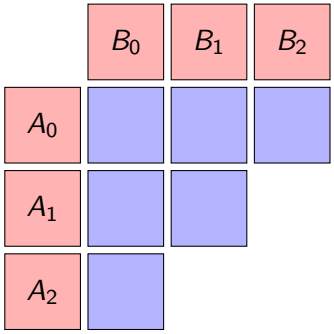
with A_i and B_j stored in precision u_{low}

- Step b) compute the $s(s+1)/2$ leading products

$$C = \sum_{i+j<s} A_i B_j$$

with a mixed precision MMA with accumulation precision u_{high}

Example: **fp32 emulation** with bfloat16 tensor cores (28× speed ratio on Blackwell)
$u_{\text{low}} \equiv$ fp16, $u_{\text{high}} \equiv$ fp32, $s = 3$



Step a) build

$$A_i = \mathrm{fl}_{\mathrm{low}}\left(A - \sum_{k=0}^{i-1} A_k\right), \quad B_j = \mathrm{fl}_{\mathrm{low}}\left(B - \sum_{k=0}^{j-1} B_k\right).$$

to obtain

$$A = \sum_{i=0}^{s-1} A_i + \Delta A, \qquad |\Delta A| \leq u_{\mathrm{low}}^{s}|A|,$$

$$B = \sum_{j=0}^{s-1} B_j + \Delta B, \qquad |\Delta B| \leq u_{\mathrm{low}}^{s}|B|.$$

Step a) build

$$A_i = \mathrm{fl}_{\mathrm{low}}\left(A - \sum_{k=0}^{i-1} A_k\right), \quad B_j = \mathrm{fl}_{\mathrm{low}}\left(B - \sum_{k=0}^{j-1} B_k\right).$$

to obtain

$$A = \sum_{i=0}^{s-1} A_i + \Delta A, \qquad |\Delta A| \leq u_{\mathrm{low}}^{s}|A|,$$

$$B = \sum_{j=0}^{s-1} B_j + \Delta B, \qquad |\Delta B| \leq u_{\mathrm{low}}^{s}|B|.$$

Step b) compute the s^2 products A_iB_j by chaining calls to the MMA:

$$\widehat{C} = \sum_{i=0}^{s-1}\sum_{j=0}^{s-1} A_iB_j + \Delta C, \quad |\Delta C| \lesssim (n+s^2)u_{\mathrm{high}}|A||B|.$$

Step a) build

$$A_i = \mathrm{fl}_{\mathrm{low}}\left(A - \sum_{k=0}^{i-1} A_k\right), \quad B_j = \mathrm{fl}_{\mathrm{low}}\left(B - \sum_{k=0}^{j-1} B_k\right).$$

to obtain

$$A = \sum_{i=0}^{s-1} A_i + \Delta A, \qquad |\Delta A| \leq u_{\mathrm{low}}^s |A|,$$

$$B = \sum_{j=0}^{s-1} B_j + \Delta B, \qquad |\Delta B| \leq u_{\mathrm{low}}^s |B|.$$

Step b) compute the s^2 products $A_i B_j$ by chaining calls to the MMA:

$$\widehat{C} = \sum_{i=0}^{s-1}\sum_{j=0}^{s-1} A_i B_j + \Delta C, \quad |\Delta C| \lesssim (n + s^2) u_{\mathrm{high}} |A||B|.$$

Overall

$$\widehat{C} = AB + E, \qquad |E| \lesssim \left(2u_{\mathrm{low}}^s + u_{\mathrm{low}}^{2s} + (n + s^2) u_{\mathrm{high}}\right)|A||B|.$$

Dropping terms

$A_i = \mathrm{fl}_{\mathrm{low}}\left(A - \sum_{k=0}^{i-1} A_k\right)$ is the approximation residual from the first $i-1$ words

$$|A_i| \leq u_{\mathrm{low}}^{i}(1+u_{\mathrm{low}})|A|$$
$$|B_j| \leq u_{\mathrm{low}}^{j}(1+u_{\mathrm{low}})|B|$$
$$|A_i||B_j| \leq u_{\mathrm{low}}^{i+j}(1+u_{\mathrm{low}})^2|A||B|$$

Dropping terms

$A_i = \mathrm{fl}_{\mathrm{low}}\left(A - \sum_{k=0}^{i-1} A_k\right)$ is the approximation residual from the first $i-1$ words

$$|A_i| \leq u_{\mathrm{low}}^i (1+u_{\mathrm{low}})|A|$$
$$|B_j| \leq u_{\mathrm{low}}^j (1+u_{\mathrm{low}})|B|$$
$$|A_i||B_j| \leq u_{\mathrm{low}}^{i+j} (1+u_{\mathrm{low}})^2 |A||B|$$

$\Rightarrow$ Not all s^2 products $A_i B_j$ need be computed! Skipping any product $A_i B_j$ such that $i+j \geq s$ only adds $O(u_{\mathrm{low}}^s)$ error terms: $\widehat{C} = AB + E$,

$$|E| \leq \left(2u_{\mathrm{low}}^s + u_{\mathrm{low}}^{2s} + (n+s^2)u_{\mathrm{high}} + \sum_{i=0}^{s-1}(s-i-1)u_{\mathrm{low}}^{s+i}(1+u_{\mathrm{low}})^2\right)|A||B|.$$

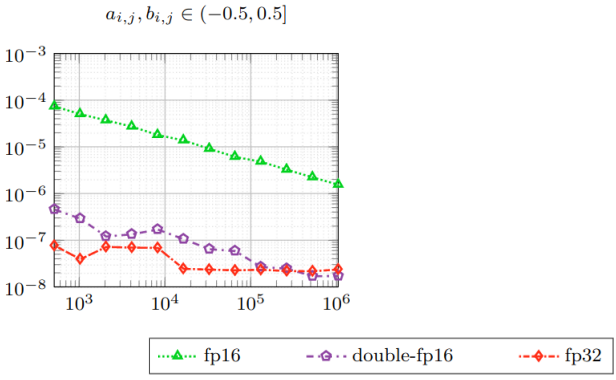
- number of products: $s^2 \to s(s+1)/2$
- error to order u_{low}^s: constant $2 \to s+1$
- further reducing the number of products is not useful (similar error as $s-1$ words)

Multiword MMA error bound (Fasi, Higham, Lopez, M., Mikaitis, SISC 2023)

The computed $\widehat{C}$ satisfies $|\widehat{C} - AB| \lesssim \left((s+1)u_{\text{low}}^s + nu_{\text{high}}\right)|A||B|$.

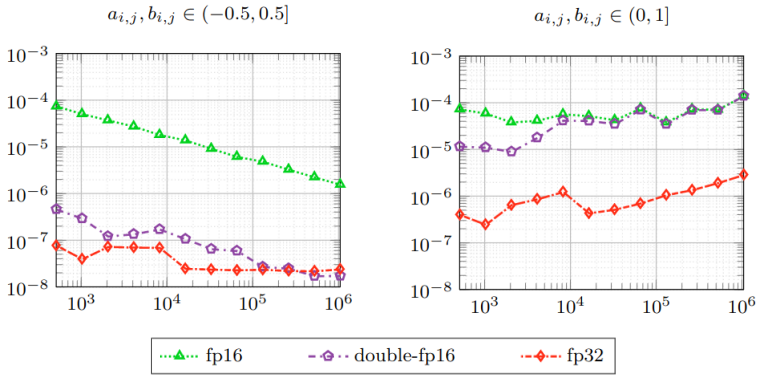
u_{high}	u_{low}		Error bound
2^{-24} (fp32)	2^{-11} (fp16)	$s = 1$	$2 \times 2^{-11} + n \times 2^{-24}$
		$s = 2$	$3 \times 2^{-22} + n \times 2^{-24}$
	2^{-8} (bfloat16)	$s = 1$	$2 \times 2^{-8} + n \times 2^{-24}$
		$s = 2$	$3 \times 2^{-16} + n \times 2^{-24}$
		$s = 3$	$n \times 2^{-24}$

From theory to practice: a word of warning



- Approach works OK for matrices with random $[-1, 1]$ uniform entries...

From theory to practice: a word of warning



- Approach works OK for matrices with random $[-1, 1]$ uniform entries...
- ... but not $[0, 1]$ entries!!

From theory to practice: a word of warning

The explanation: the culprit is round to zero (RZ)

- fp32 uses the standard RTN, but tensor cores only support RZ [Fasi, Higham, Mikaitis, Pranesh, PeerJ CS 2021]
- With data of nonzero mean and RZ, most rounding errors happen in the same direction

$\Rightarrow$ Worst-case bound nu_{32} is attained with RZ, whereas with RTN we can usually replace it by $\sqrt{n}u_{32}$ [Higham and M., SISC 2019, 2020]

- Same error *bound* $\neq$ same error !

From theory to practice: a word of warning

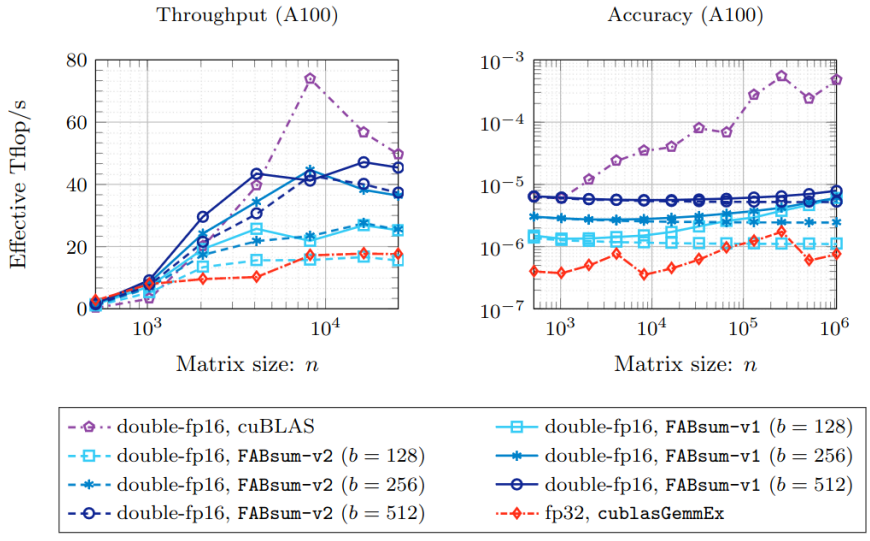
The explanation: the culprit is round to zero (RZ)

- fp32 uses the standard RTN, but tensor cores only support RZ [Fasi, Higham, Mikaitis, Pranesh, PeerJ CS 2021]
- With data of nonzero mean and RZ, most rounding errors happen in the same direction

$\Rightarrow$ Worst-case bound nu_{32} is attained with RZ, whereas with RTN we can usually replace it by $\sqrt{n}u_{32}$ [Higham and M., SISC 2019, 2020]

- Same error *bound* $\neq$ same error !
- Possible cures:
  - Since the worst-case accumulation bound nu_{32} is attained $\Rightarrow$ reduce the bound ! e.g., with blocked summation/FABsum [Blanchard, Higham, M., SISC 2020]
  - Alternatively, avoid using tensor cores for accumulation [Ootomo and Yokota, IJHPCA 2022]

FABsum cure



- Tradeoff between speed and accuracy

fp32 emulation with bfloat16-TC in cuBLAS

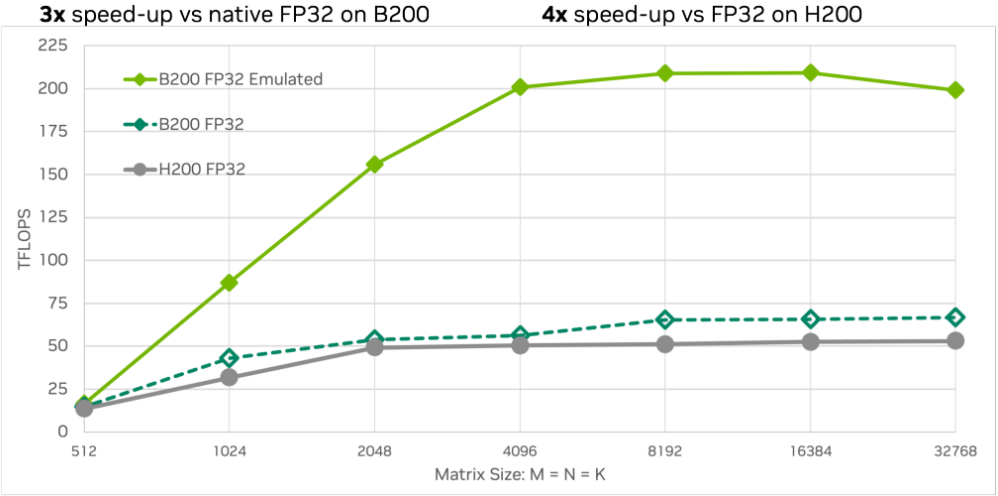


Figure courtesy of NVIDIA

fp64 emulation: Ozaki-I scheme

- Tensor cores do not provide fp64 accumulators. Can we nevertheless emulate fp64 accuracy?
- Ozaki scheme: decompose A and B such that $A_i B_j$ can be computed *exactly* [Ozaki, Ogita, Oishi, Rump, NumAlgs 2012]
- A particularly simple yet efficient approach is obtained by pushing this logic to the extreme: decompose A and B into *integers*! [Ootomo, Ozaki, Yokota, IJHPCA 2024]
  - Step 1: compute scaled integer approximations $A \approx D_A A'$ and $B \approx B' D_B$ (where A', B' have integer coefficients)
  - Step 2: compute $C' = A'B'$ with multiword integer arithmetic
  - Step 3: recover $C = D_A C' D_B$ (exact if scaling factors are powers of two)

Step 1: approximation error

Goal: find scalings D_A, D_B and integer A', B' such that $A \approx D_A A'$ and $B \approx B' D_B$

$\Rightarrow$ Block floating-point representations of A and B (rows of A and columns of B must share same exponent)

- Base-10 examples with 3-digit integers:

$$A = [1.234] = \underbrace{[10^{-2}]}_{D_A} \times \underbrace{[123]}_{A'} + \underbrace{[0.004]}_{\text{error}}$$

Step 1: approximation error

Goal: find scalings D_A, D_B and integer A', B' such that $A \approx D_A A'$ and $B \approx B' D_B$

$\Rightarrow$ Block floating-point representations of A and B (rows of A and columns of B must share same exponent)

- Base-10 examples with 3-digit integers:

$$A = \begin{bmatrix} 1.234 \\ 0.05678 \end{bmatrix} = \underbrace{\begin{bmatrix} 10^{-2} & \\ & 10^{-4} \end{bmatrix}}_{D_A} \times \underbrace{\begin{bmatrix} 123 \\ 568 \end{bmatrix}}_{A'} + \underbrace{\begin{bmatrix} 0.004 \\ -0.00002 \end{bmatrix}}_{\text{error}}$$

Step 1: approximation error

Goal: find scalings D_A, D_B and integer A', B' such that $A \approx D_A A'$ and $B \approx B' D_B$

$\Rightarrow$ Block floating-point representations of A and B (rows of A and columns of B must share same exponent)

- Base-10 examples with 3-digit integers:

$$A = \begin{bmatrix} 1.234 \\ 0.05678 \end{bmatrix} = \underbrace{\begin{bmatrix} 10^{-2} & \\ & 10^{-4} \end{bmatrix}}_{D_A} \times \underbrace{\begin{bmatrix} 123 \\ 568 \end{bmatrix}}_{A'} + \underbrace{\begin{bmatrix} 0.004 \\ -0.00002 \end{bmatrix}}_{\text{error}}$$

$$A = \begin{bmatrix} 1.234 & 0.05678 \end{bmatrix} = \underbrace{\begin{bmatrix} 10^{-2} \end{bmatrix}}_{D_A} \times \underbrace{\begin{bmatrix} 123 & 006 \end{bmatrix}}_{A'} + \underbrace{\begin{bmatrix} 0.004 & -0.00322 \end{bmatrix}}_{\text{error}}$$

Step 1: approximation error

Goal: find scalings D_A, D_B and integer A', B' such that $A \approx D_A A'$ and $B \approx B' D_B$

$\Rightarrow$ Block floating-point representations of A and B (rows of A and columns of B must share same exponent)

- Base-10 examples with 3-digit integers:

$$A = \begin{bmatrix} 1.234 \\ 0.05678 \end{bmatrix} = \underbrace{\begin{bmatrix} 10^{-2} & \\ & 10^{-4} \end{bmatrix}}_{D_A} \times \underbrace{\begin{bmatrix} 123 \\ 568 \end{bmatrix}}_{A'} + \underbrace{\begin{bmatrix} 0.004 \\ -0.00002 \end{bmatrix}}_{\text{error}}$$

$$A = \begin{bmatrix} 1.234 & 0.05678 \end{bmatrix} = \underbrace{\begin{bmatrix} 10^{-2} \end{bmatrix}}_{D_A} \times \underbrace{\begin{bmatrix} 123 & 006 \end{bmatrix}}_{A'} + \underbrace{\begin{bmatrix} 0.004 & -0.00322 \end{bmatrix}}_{\text{error}}$$

- For integers bounded by p, the error $|e_{ij}|$ is less than $p^{-1} \times \max_j |a_{ij}|$ for A (conversely, $p^{-1} \times \max_i |b_{ij}|$ for B), A simpler but weaker normwise bound is

$$\|AB - D_A A' B' D_B\| \leq c(n) \times p^{-1} \times \|A\| \|B\|$$

Step 1: approximation error

Goal: find scalings D_A, D_B and integer A', B' such that $A \approx D_A A'$ and $B \approx B' D_B$

- For integers bounded by p, the error $|e_{ij}|$ is less than $p^{-1} \times \max_j |a_{ij}|$ for A (conversely, $p^{-1} \times \max_i |b_{ij}|$ for B), A simpler but weaker normwise bound is

$$\|AB - D_A A' B' D_B\| \leq c(n) \times p^{-1} \times \|A\| \|B\|$$

$\Rightarrow$ Is this satisfactory?

- Argument for YES: several common algorithms can only guarantee normwise accuracy (underflow, Strassen, compression...)
- Argument for NO: standard algorithm guarantees componentwise accuracy (relative to $|A||B|$)

Step 1: extra precision and adaptive choice of p

- Since error $|e_{ij}|$ is less than $p^{-1} \times \max_j |a_{ij}|$ for A and $p^{-1} \times \max_i |b_{ij}|$ for B, use extra digits! May need up to $p \leftarrow \max(\kappa_A, \kappa_B) \times p$, where

$$\kappa_A = \max_i \frac{\max_j |a_{ij}|}{\min_j |a_{ij}|}, \qquad \kappa_B = \max_j \frac{\max_i |b_{ij}|}{\min_i |b_{ij}|}$$

  [Abdelfattah, Dongarra, Fasi, Mikaitis, Tisseur, 2025]

- This is a very pessimistic bound. Can be sharpened in several ways:
- Use different integer sizes:
  - for A and B, $p_A = \kappa_A \times p$ and $p_B = \kappa_B \times p$
  - for different rows of A (columns of B), e.g., $p_A^{(i)} = \frac{\max_j |a_{ij}|}{\min_j |a_{ij}|} \times p$
  - for block-columns of A (block-rows of B)
- More importantly, a large relative error on $|a_{ij}|$ is only problematic if there exists a large $|b_{jk}|$ in front of it $\Rightarrow$ given a row x^T of A and a column y of B, compute $z = x \circ y$, $\kappa = \frac{\max |x| \max |y|}{\max |z|}$, and set $p \leftarrow p \times \kappa$ [NVIDIA, 2025]

Step 2: accumulation error

Goal: compute $C' = A'B'$, $A' \in \mathbb{Z}^{m \times n}$, $B' \in \mathbb{Z}^{n \times q}$, coefficients bounded by p

- Step 2a) compute the decompositions

$$A' = \sum_{i=0}^{s-1} \gamma^{s-i-1} A_i \quad \text{and} \quad B' = \sum_{j=0}^{s-1} \gamma^{s-j-1} B_j$$

with coefficients A_i and B_j bounded by $\gamma = \lceil p^{1/s} \rceil$

$\Rightarrow$ exact if $\gamma \leq 2^{\sigma} - 1$, where σ is the storage bitsize

- Step 2b) compute each individual product:

$$C_{ij} = A_i B_j$$

in integer arithmetic

$\Rightarrow$ exact if $n\gamma^2 \leq 2^{\tau} - 1$, where τ is the accumulator bitsize

- Step 2c) accumulate all the products in fp64:

$$C = \sum_{i,j} \gamma^{2s-i-j-2} C_{ij}$$

$\Rightarrow$ fp64 accumulation errors
+ dropping errors if we skip computing C_{ij} when $i + j \geq s$

Step 2: further details and optimizations

- Which of $\gamma \leq 2^\sigma - 1$ and $n\gamma^2 \leq 2^\tau - 1$ is the limiting condition?

$\Rightarrow$ Depends!

`uniform (e.g., fp32)`	$\sigma = \tau$		$\Rightarrow$ τ `is limiting`
`fp16-TC`	$\sigma = 11$	$\tau = 24$	$\Rightarrow$ τ `is limiting for` $n > 4$
`bfloat16-TC`	$\sigma = 8$	$\tau = 24$	$\Rightarrow$ τ `is limiting for` $n > 2^8$
`int8-TC`	$\sigma = 7$	$\tau = 31$	$\Rightarrow$ σ `is limiting for` $n < 2^{17}$

- When τ is limiting, use blocking to replace n with block size: $b\gamma^2 < 2^\tau - 1$

$$A_i B_j = \begin{bmatrix} A_i^{(1)} & \dots & A_i^{(n/b)} \end{bmatrix} \begin{bmatrix} B_j^{(1)} \\ \vdots \\ B_j^{(n/b)} \end{bmatrix} \quad \Rightarrow \quad C = \sum_{i,j} \gamma^{2s-i-j-2} \sum_{k=1}^{n/b} A_i^{(k)} B_j^{(k)}$$

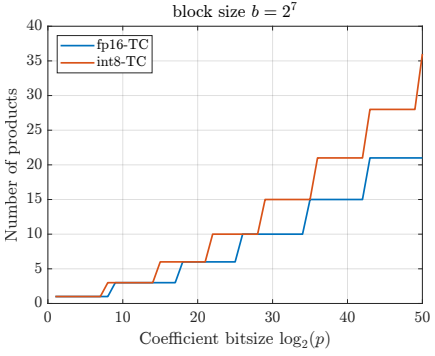
- When σ is limiting, the number of fp64 additions can be reduced by summing C_{ij} for fixed $i + j$ together in integer arithmetic [Uchino, Ozaki, Imamura, IJHPCA 2025]

Number of products vs precision — multiword

- Precision determined by integer bitsize of A' and B':

$$\log_2(p) \approx \log_2(\gamma^s) \approx s \times \min\left(\sigma, \frac{\tau - \log_2 b}{2}\right)$$

- Number of products equal to $\frac{s(s+1)}{2}$ (with dropping) $\Rightarrow$ quadratic in s



fp64 emulation with int8-TC in cuBLAS

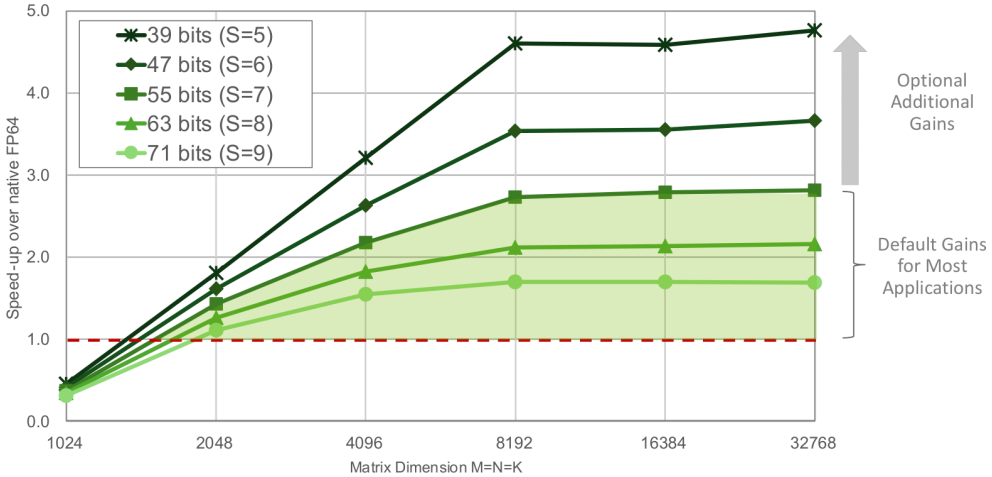


Figure courtesy of NVIDIA

Ozaki-II scheme

Ozaki-II scheme is based on multimodular matrix multiplication [Ozaki, Uchino, Imamura, 2025]

- Step 1: compute scaled integer approximations $A \approx D_A A'$ and $B \approx B' D_B$ (where A', B' have integer coefficients)
- Step 2: compute $C' = A'B'$ with multimodular integer arithmetic
- Step 3: recover $C = D_A C' D_B$ (exact if scaling factors are powers of two)

Chinese remainder theorem

- Let $m_1, \ldots, m_s$ be pairwise coprime integers (called moduli) and let $M = \prod_{i=1}^{s} m_i$. For given remainders $a_1 < m_1, \ldots, a_s < m_s$, there exists a unique $x < M$ such that $x \equiv a_i \bmod m_i$ for $i = 1, \ldots, s$. Moreover x is given by

$$x = \sum_{i=1}^{s} a_i M_i N_i \bmod M, \quad \text{with } M_i = M/m_i \text{ and } M_i N_i \equiv 1 \bmod m_i.$$

- In other words, if $x \bmod m_i$ is known for sufficiently many m_i (for sufficiently large $M = \prod_{i=1}^{s} m_i$), then x can be recovered.

- Key observation: $C = AB \bmod m_i = (A \bmod m_i)(B \bmod m_i) \bmod m_i$, and the coefficients of $A \bmod m_i$ and $B \bmod m_i$ are less than m_i! $\Rightarrow$ use modular reductions to reduce the coefficient sizes

Step 2 with multimodular approach

Goal: compute $C' = A'B'$, $A' \in \mathbb{Z}^{m \times n}$, $B' \in \mathbb{Z}^{n \times q}$, coefficients bounded by p

- Step 2a) compute the modular reductions
  $A_i = A' \bmod m_i$ and $B_i = B' \bmod m_i$
  with coefficients A_i and B_i bounded by $\gamma = \max_i m_i$

  $\Rightarrow$ exact if $\gamma \leq 2^\sigma - 1$, where σ is the storage bitsize

- Step 2b) compute each individual product:
  $$C_i = A_i B_i$$
  in integer arithmetic

  $\Rightarrow$ exact if $n\gamma^2 \leq 2^\tau - 1$, where τ is the accumulator bitsize

- Step 2c) recover the true product in fp64:
  $$C = \sum_{i=1}^{s} C_i M_i N_i \bmod M$$

  $\Rightarrow$ fp64 accumulation errors
  + CRT condition:
  $M > np^2 \Rightarrow \gamma^s \gtrsim np^2$
  ($n \leftarrow b$ with blocking)

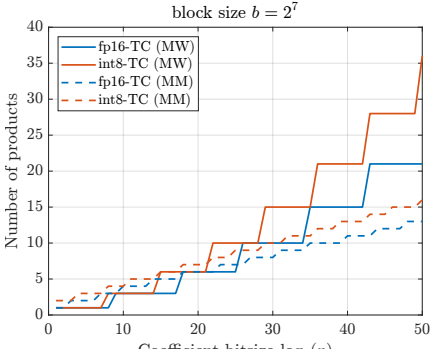
Number of products vs precision — multimodular

- Precision determined by integer bitsize of A' and B':

$$\log_2(p) \approx \frac{\log_2(\gamma^s) - \log_2(b)}{2} \approx \frac{s \times \min\left(\sigma, \frac{\tau - \log_2 b}{2}\right) - \log_2(b)}{2}$$

- Number of products equal to $s \Rightarrow$ linear in s

$\Rightarrow$ Compared to multiword for fixed s: less precision (by about a factor of 2) but less products (by about a factor $s/2$) $\Rightarrow$ cutoff around $s_{\mathrm{MW}} \approx 4$ (10 products)



Precision emulation

Linear systems

Block low-rank approximations

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 $\kappa(A) = \|A\|\|A^{-1}\|$

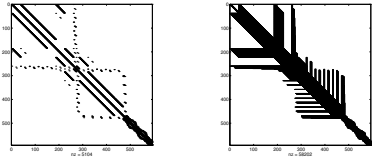
Cost for dense $n \times n$ matrices:

- $2n^3/3$ flops
- n^2 entries (if A is overwritten with $L + U$)

Sparse matrices

- Sparse matrices satisfy $\mathrm{nnz}(A) \ll \mathrm{nnz}(LU)$

Original matrix *Factorized matrix*



- For example, for 3D cubic problems,
  - $\mathrm{nnz}(A) = O(n)$
  - $\mathrm{nnz}(LU) = O(n^{4/3})$
  - flops $= O(n^2)$

$r_0 = b - Ax_0$
$\beta = \|r_0\|$
$q_0 = r_0/\beta$
repeat
  $v_k = Aq_k$
  for $j = 1 \colon k$ **do**
    $h_{jk} = q_j^T v_k$
    $v_k = v_k - h_{jk} q_j$
  end for
  $h_{k+1,k} = \|v_k\|$
  $q_{k+1} = v_k / h_{k+1,k}$
  Solve $\min_{y_k} \|r_k\| = \|\beta e_1 - H y_k\|$.
until $\|r_k\|$ is small enough
$x_k = x_0 + Q_k y_k$

Krylov method to solve $Ax = b$: build orthogonal basis Q_k of subspace $\mathcal{K}_k = \{r_0, Ar_0, A^2 r_0, \ldots, A^k r_0\}$

At iteration k:

- Add $q_k = A^{k-1} r_0 = Aq_{k-1}$ to the basis
- Arnoldi relation: $AQ_k = Q_{k+1} H_k$
- Find $x_k \in x_0 + \mathcal{K}_k$ minimizing $r_k = Ax_k - b$
  - $x_k \in x_0 \mathcal{K}_k \Rightarrow x_k = x_0 + Q_k y_k$
  - $r_k = b - Ax_k = b - A(x_0 + Q_k y_k) = r_0 - AQ_k y_k = \beta q_1 - Q_{k+1} H_k y_k = Q_{k+1}(\beta e_1 - H_k y_k)$
- Stop if $\|r_k\|$ is small enough

MGS-GMRES is backward stable Paige, Rozloznik, Strakos (2006)

Theorem

If unpreconditioned GMRES run in precision u, there exists an iteration $k \leq n$ at which the iterate $\widehat{x}_k$ satisfies

$$(A + \Delta A)\widehat{x}_k = b, \quad \|\Delta A\| \leq O(u)\|A\|$$

and so

$$\|\widehat{x}_k - x\| \lesssim \kappa(A)u\|x\|.$$

This is an existence result: no guarantee that k will be small (might be as large as n!)

Restarted GMRES

$x = x_0$
$r = b - Ax$ and $\beta = \|r\|$
while β is not small enough **do**
  $q_0 = r/\beta$
  for $k = 1 : m$ **do**
    $v_k = Aq_k$
    for $j = 1 : k$ **do**
      $h_{jk} = q_j^T v_k$
      $v_k = v_k - h_{jk} q_j$
    end for
    $h_{k+1,k} = \|v_k\|$
    $q_{k+1} = v_k / h_{k+1,k}$
    Solve $\min_{y_k} \|r_k\| = \|\beta e_1 - H y_k\|$.
  end for
  $x = x + Q_k y_k$
  $r = b - Ax$ and $\beta = \|r\|$
end while

- Cost of SpMV: $\mathrm{nnz}(A)$ per iteration
- Cost of building $\mathcal{K}_m$:
  - $O(nm^2)$ flops
  - $O(nm)$ storage

  $\Rightarrow$ unaffordable as m increases
- Restarted GMRES: limit size of Krylov basis to small m
  - Stop after m inner iterations
  - Update x and restart
  - Repeat until $\|r\|$ is small enough
- Slower convergence but bounded cost per iteration

Preconditioned GMRES

- Convergence of GMRES strongly depends on matrix $\Rightarrow$ preconditioning is needed
- Preconditioned GMRES: apply GMRES to

$$MAx = Mb \qquad \text{(left preconditioning)}$$
$$AMy = b, \quad My = x \qquad \text{(right preconditioning)}$$

  where $M \approx A^{-1}$
- Some examples of preconditioners:
  - Jacobi: $M = \text{diag}(A)^{-1}$
  - Low precision LU preconditioner: $M = U^{-1}L^{-1}$ $\Rightarrow$ requires triangular solves
  - Approximate LU: incomplete LU, BLR LU, ...

Direct vs iterative

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

Direct vs iterative

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

Direct vs iterative

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

An algorithm to refine the solution: **iterative refinement** (IR)

Solve $Ax_1 = b$ via $x_1 = U^{-1}(L^{-1}b)$
while Not converged **do**
  $r_i = b - Ax_i$
  Solve $Ad_i = r_i$
  $x_{i+1} = x_i + d_i$
end while

Many variants over the years, depending on choice of precisions and solver for $Ad_i = r_i$

Error analysis of general IR

Carson and Higham (2018) analyze the most general version of IR to date:

Choose an initial x_1
for $i = 1$: *nsteps* **do**
  $r_i = b - Ax_i$ in precision $\mathbf{u_r}$
  Solve $Ad_i = r_i$ such that $\|\widehat{d}_i - d_i\| \leq \phi_i \|d_i\|$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
end for

Theorem (simplified from Carson and Higham, 2018)

Under the condition $\phi_i < 1$, the forward error is reduced at each step by a factor ϕ_i until it reaches its limiting value

$$\frac{\|\widehat{x} - x\|}{\|x\|} \lesssim 2n\kappa(A)\mathbf{u_r} + \mathbf{u}$$

Proof

Assuming

- $\widehat{r}_i = b - A\widehat{x}_i + e_i, \quad \|e_i\| \leq n\mathbf{u_r}(\|A\|\|\widehat{x}_i\| + \|b\|)$
- $\widehat{d}_i = d_i + f_i, \quad \|f_i\| \leq \phi_i\|d_i\|$
- $\widehat{x}_{i+1} = \widehat{x}_i + \widehat{d}_i + g_i, \quad \|g_i\| \leq \mathbf{u}\|\widehat{x}_{i+1}\|$

we have $\widehat{x}_{i+1} = x + A^{-1}e_i + f_i + g_i$ and thus

$$\|x - \widehat{x}_{i+1}\| \lesssim (2n\kappa(A)\mathbf{u_r} + \mathbf{u})\|x\| + \phi_i\|x - \widehat{x}_i\|$$

Choice of solver determines ϕ_i:

- **LU solver:** $d_i = U^{-1}L^{-1}r_i$

$$\frac{\|\widehat{d_i} - d_i\|}{\|\widehat{d_i}\|} \lesssim f(n)\||A^{-1}||\widehat{L}||\widehat{U}\|\mathbf{u_f} \lesssim f(n)\rho_n\kappa(A)\mathbf{u_f}$$

  where ρ_n is the growth factor: the maximum magnitude of the elements appearing during LU factorization

- For stable pivoting strategies (e.g., partial pivoting), ρ_n is almost always $O(1)$

$\Rightarrow$ $\phi_i < 1 \Leftrightarrow \kappa(A)\mathbf{u_f} \ll 1$

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)

- Convergence speed: $\phi = O(\kappa(A)\mathbf{u})$
- Attainable accuracy: $O(\kappa(A)\mathbf{u}^2) = O(\mathbf{u})$

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)

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

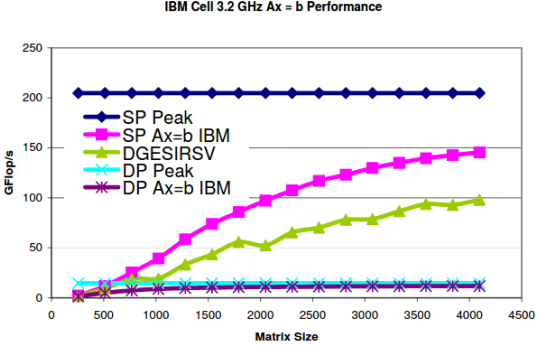
with $\mathbf{u_f} \equiv$ fp32 and $\mathbf{u} \equiv$ fp64

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
- Convergence speed: $\phi = O(\kappa(A)\mathbf{u_f})$
- Attainable accuracy: $O(\kappa(A)\mathbf{u})$

LU-IR with CELL processor

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



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)

- Convergence speed: $\phi = O(\kappa(A)\mathbf{u_f})$
- Attainable accuracy: $O(\mathbf{u} + \kappa(A)\mathbf{u_r})$
- Three-precision LU-IR is as general (as modular) as possible

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**

      $A_{ij} \leftarrow A_{ij} - \widetilde{L}_{ik}\widetilde{U}_{kj}$
    end for
  end for
end for

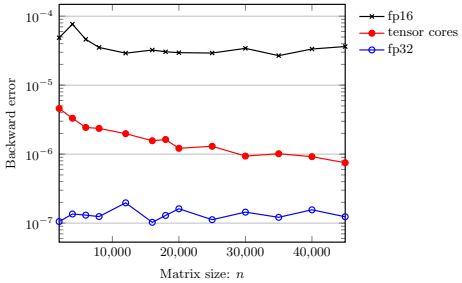
Block LU factorization with tensor cores

- 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

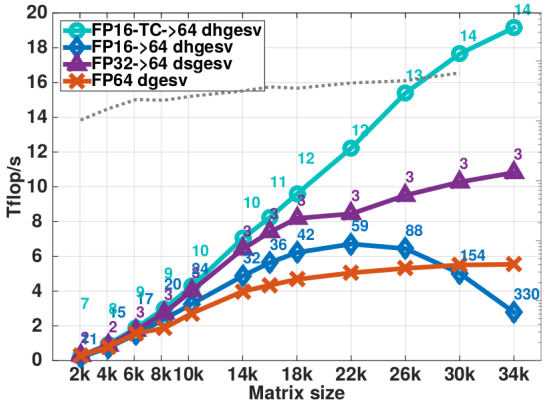
Error analysis for LU follows from matrix multiplication analysis and gives same bounds to first order Blanchard et al. (2020)

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



Impact on iterative refinement

Results from Haidar et al. (2018)

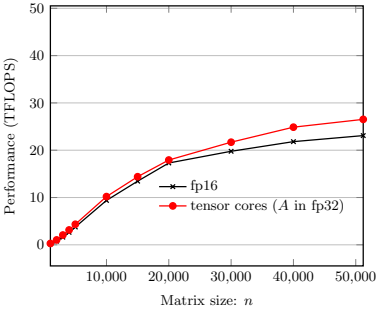


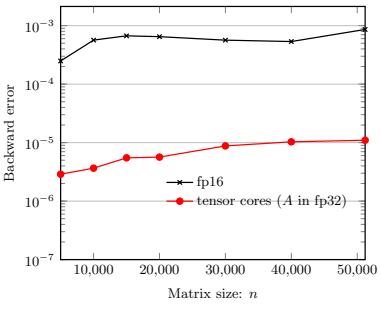
- TC accuracy boost can be critical!
- TC performance suboptimal here ⇒ why?

LU factorization is memory bound

- LU factorization is traditionally a compute-bound operation...
- With Tensor Cores, flops are $O(10\times)$ faster
- Matrix is stored in fp32 $\Rightarrow$ **data movement is unchanged**

$\Rightarrow$ LU with tensor cores becomes memory-bound !

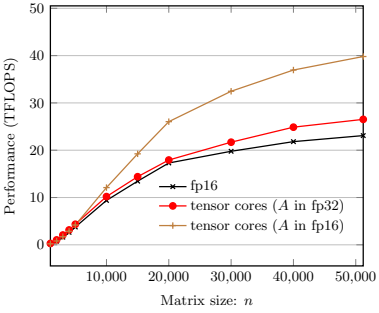


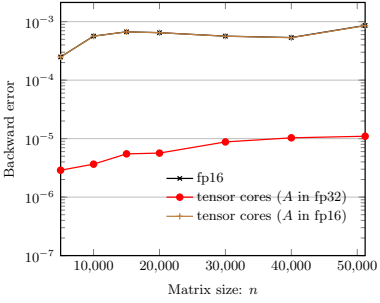


LU factorization is memory bound

- LU factorization is traditionally a compute-bound operation...
- With Tensor Cores, flops are $O(10\times)$ faster
- Matrix is stored in fp32 $\Rightarrow$ **data movement is unchanged**

$\Rightarrow$ 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

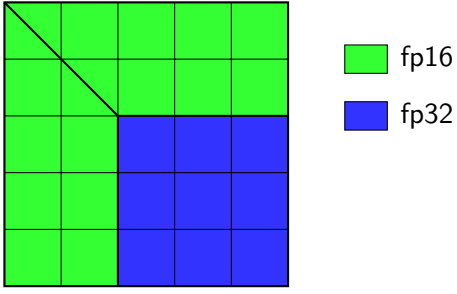
Two ingredients to reduce data movement with no accuracy loss:

Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

1. Mixed fp16/fp32 representation

Matrix after 2 steps:

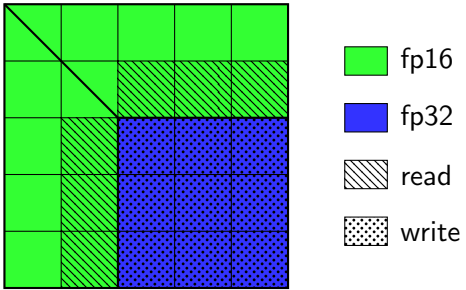


Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

1. Mixed fp16/fp32 representation

Matrix after 2 steps:



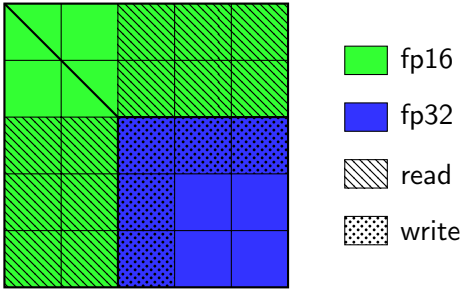
for $k = 1$ **to** p **do**
  FACTOR:
  Compute the LU fac. $L_{kk}U_{kk} = A_{kk}$.
  for $i = k + 1$ **to** p **do**
    Solve $L_{ik}U_{kk} = A_{ik}$ for L_{ik}.
    Solve $L_{kk}U_{ki} = A_{ki}$ for U_{ki}.
  end for
  UPDATE:
  for $i = k + 1$ **to** p **do**
    for $j = k + 1$ **to** p **do**
      $A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}$
    end for
  end for
end for

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

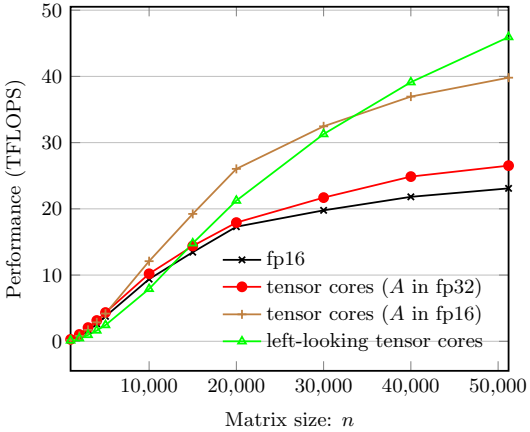
Matrix after 2 steps:

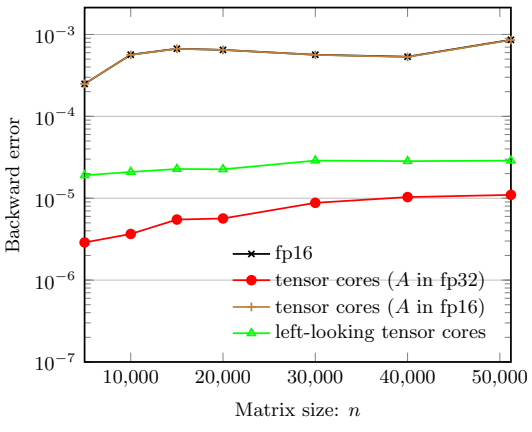


for $k = 1$ **to** p **do**
  UPDATE:
  $A_{kk} \leftarrow A_{kk} - \sum_{j=1}^{k-1} L_{kj} U_{jk}$.
  for $i = k+1$ **to** p **do**
    $A_{ik} \leftarrow A_{ik} - \sum_{j=1}^{k-1} L_{ij} U_{jk}$
    $A_{ki} \leftarrow A_{ki} - \sum_{j=1}^{k-1} L_{kj} U_{ji}$
  end for
  FACTOR:
  Compute the LU fac. $L_{kk} U_{kk} = A_{kk}$.
  for $i = k+1$ **to** p **do**
    Solve $L_{ik} U_{kk} = A_{ik}$ for L_{ik}.
    Solve $L_{kk} U_{ki} = A_{ki}$ for U_{ki}.
  end for
end for

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

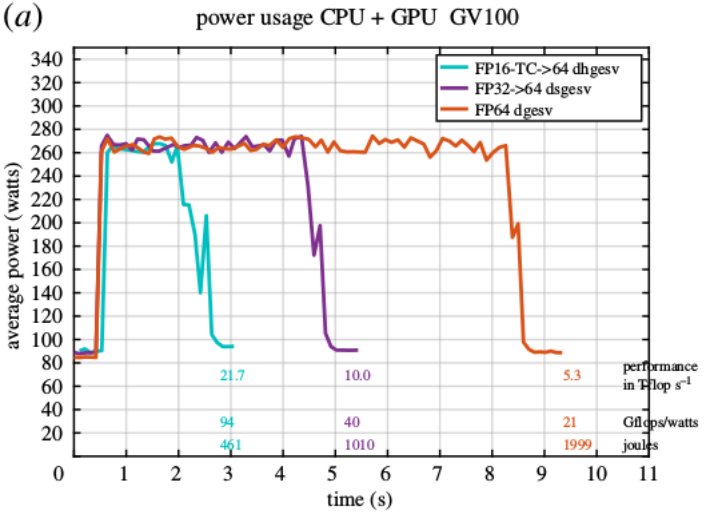
Experimental results





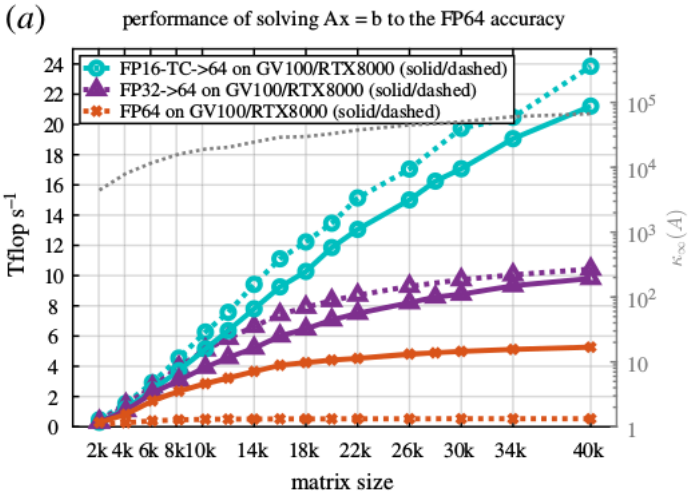
- Nearly **50 TFLOPS** without significantly impacting accuracy
  Lopez and M. (2020)
- Even more critical on A100:
  50 TFLOPS (A in fp32) $\rightarrow$ **175 TFLOPS** (A in fp16+left-looking)

Power consumption of IR



Haidar et al (2020)

IR on GPUs with little fp64 support



Haidar et al (2020)

Scaling and shifting 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 ($D = \mathrm{diag}(A)$, $\sigma = O(u_{16})$)

GMRES-based IR (GMRES-IR)

Choose an initial x_1
repeat
  $r_i = b - Ax_i$ in precision $\mathbf{u}_r$
  Solve $Ad_i = r_i$ with GMRES in precision $\mathbf{u_g}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

- GMRES is stable $\Rightarrow \phi = \kappa(A)\mathbf{u_g}$
- Can be interpreted as mixed precision restarted GMRES
- Inner GMRES is unpreconditioned $\Rightarrow$ might take too many iterations!

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 $\mathbf{u_g}$
  $x_{i+1} = x_i + d_i$ in precision $\mathbf{u}$
until converged

Rationale for replacing LU solver by preconditioned GMRES:

- GMRES can be asked to converge to accuracy $\mathbf{u_g} \ll \mathbf{u_f}$
- $\kappa(\widetilde{A}) = \kappa(U^{-1}L^{-1}A)$ often smaller than $\kappa(A)$
- If $\widetilde{A}d_i = \widetilde{r}_i$ were solved with accuracy $\phi_i = \kappa(\widetilde{A})\mathbf{u_g}$, convergence condition would be improved from $\kappa(A)\mathbf{u_f} < 1$ to $\kappa(\widetilde{A})\mathbf{u_g} < 1 \ldots$
- . . . but there is a catch!

Stability of preconditioned GMRES

- As mentioned previously unpreconditioned GMRES is stable...
- ...but what about preconditioned GMRES? Two key differences:
  - The system being solved is $\widehat{U}^{-1}\widehat{L}^{-1}Ax = \widehat{U}^{-1}\widehat{L}^{-1}b \Rightarrow$ bound becomes of order $\kappa(\widetilde{A})\mathbf{u_g}$.

Stability of preconditioned GMRES

- As mentioned previously unpreconditioned GMRES is stable...
- ...but what about preconditioned GMRES? Two key differences:
  - The system being solved is $\widehat{U}^{-1}\widehat{L}^{-1}Ax = \widehat{U}^{-1}\widehat{L}^{-1}b \Rightarrow$ bound becomes of order $\kappa(\widetilde{A})\mathbf{u_g}$.
  - The matrix–vector products are performed with $\widehat{U}^{-1}$, $\widehat{L}^{-1}$, and A separately, not directly with $\widetilde{A}$ (which is never formed)
    - $y = \widetilde{A}x \Rightarrow \|\widehat{y} - y\| \leq n\mathbf{u_g}\|\widetilde{A}\|\|x\|$
    - $y = \widehat{U}^{-1}\widehat{L}^{-1}Ax \Rightarrow \|\widehat{y} - y\| \leq f(n)u\|A\|\|\widehat{U}^{-1}\|\|\widehat{L}^{-1}\|\|x\| \lesssim \kappa(A)f(n)\mathbf{u_g}\|\widetilde{A}\|\|x\|$

    $\Rightarrow$ extra $\kappa(A)$ term appears, it is as if GMRES was run in "precision" $\kappa(A)\mathbf{u_g}$

- Overall: $\phi_i = \kappa(\widetilde{A})\kappa(A)\mathbf{u_g}$

$\Rightarrow$ Potentially better than $\phi = \kappa(A)\mathbf{u_f}$ if $\kappa(\widetilde{A})$ is small

- We have the (pessimistic) bound $\kappa(\widetilde{A}) \leq \mathbf{u_f}^2\kappa(A)^2$

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

- Perform matvecs with $\widetilde{A}$ in precision $\mathbf{u_p} \leq \mathbf{u_g}$ to reduce $\kappa(A)$ dependence
- Convergence speed: $\phi = O(\kappa(\widetilde{A})(\mathbf{u_g} + \kappa(A)\mathbf{u_p})) = O(\kappa(A)^2\mathbf{u_f}^2(\mathbf{u_g} + \kappa(A)\mathbf{u_p}))$
- Attainable accuracy: $O(\mathbf{u} + \kappa(A)\mathbf{u_r})$
- Modular error analysis (parameterize every line by independent precisions) reveals the numerical structure of the algorithm!

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

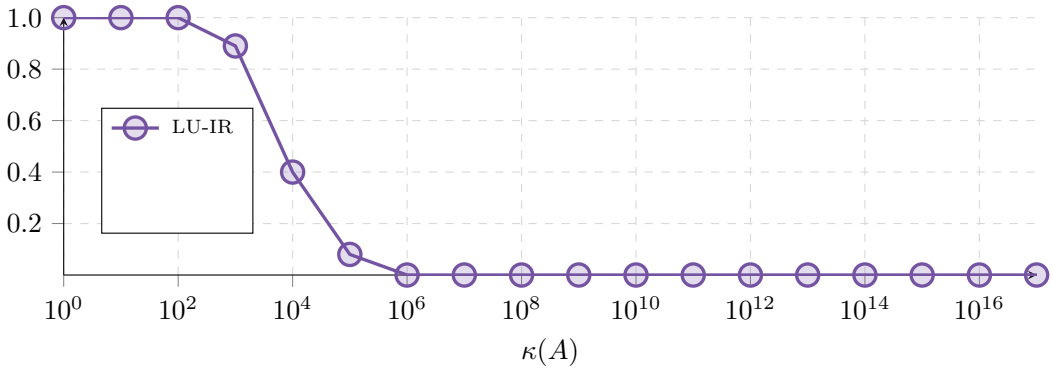
Meaningful combinations of GMRES-IR5 for $\mathbf{u_f} \equiv$ fp16 and $\mathbf{u} \equiv$ fp64

$\mathbf{u_g}$	$\mathbf{u_p}$	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}

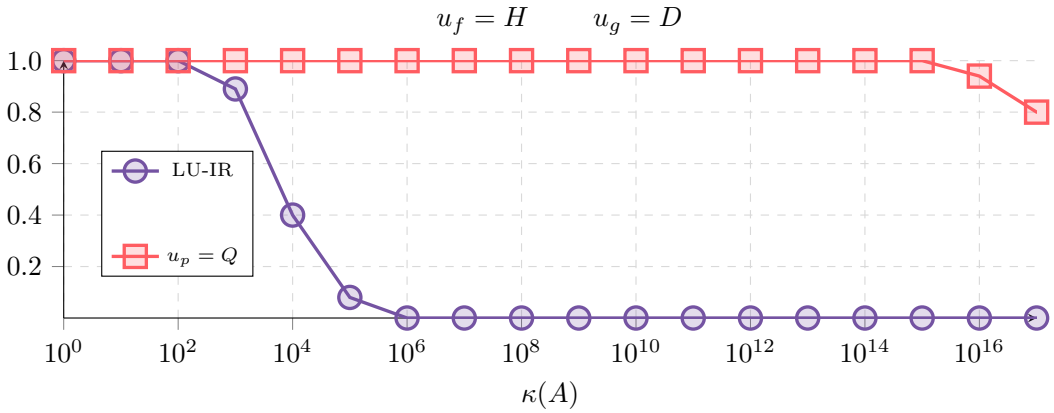
Six meaningful combinations $\Rightarrow$ **flexible** precisions choice to fit at best the **hardware constraints** and the **problem difficulty**.

Take 100 random matrices with specified $\kappa(A)$ and measure the success rate: the percentage of matrices for which GMRES-IR5 converges to a small forward error

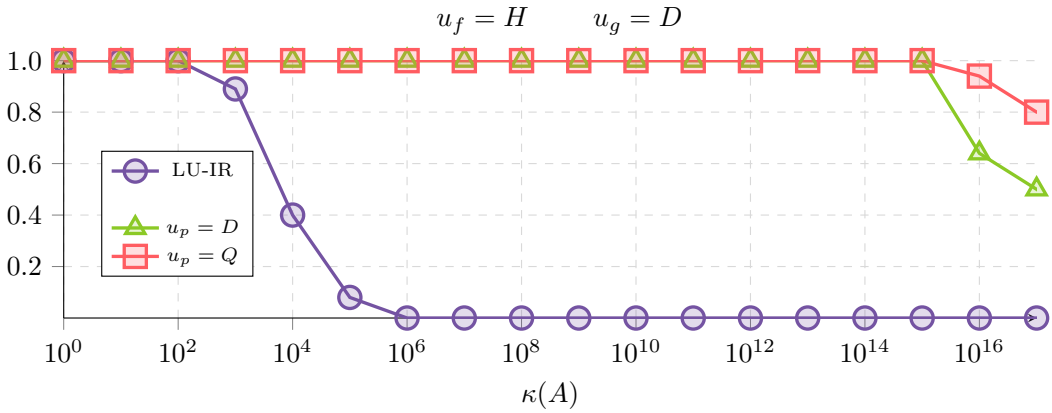
$u_f = H \qquad u_g = D$



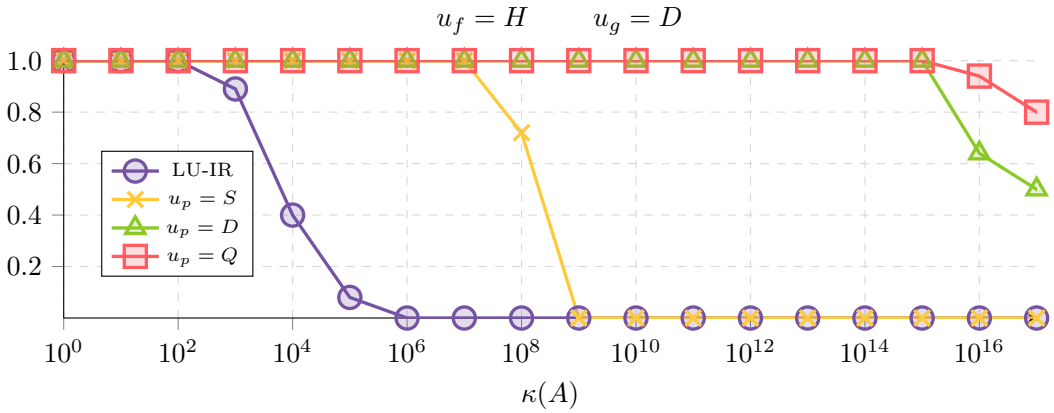
Take 100 random matrices with specified $\kappa(A)$ and measure the success rate: the percentage of matrices for which GMRES-IR5 converges to a small forward error



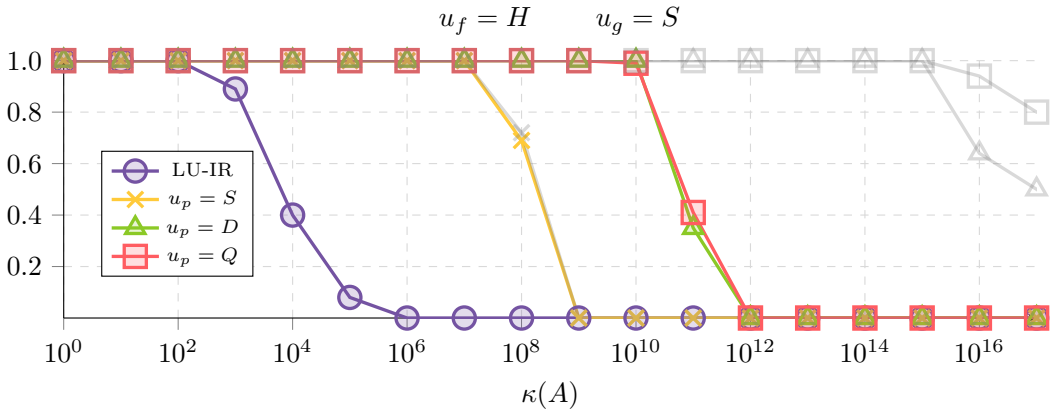
Take 100 random matrices with specified $\kappa(A)$ and measure the success rate: the percentage of matrices for which GMRES-IR5 converges to a small forward error



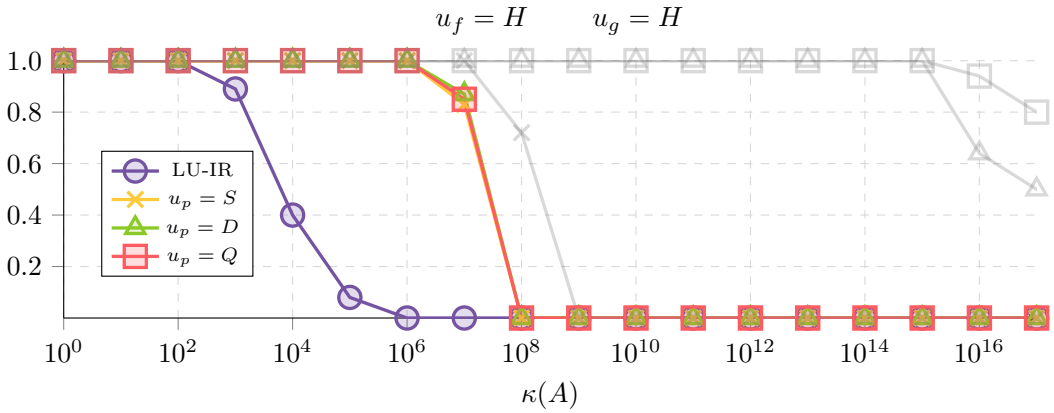
Take 100 random matrices with specified $\kappa(A)$ and measure the success rate: the percentage of matrices for which GMRES-IR5 converges to a small forward error



Take 100 random matrices with specified $\kappa(A)$ and measure the success rate: the percentage of matrices for which GMRES-IR5 converges to a small forward error



Take 100 random matrices with specified $\kappa(A)$ and measure the success rate: the percentage of matrices for which GMRES-IR5 converges to a small forward error



$r_0 = b - Ax_0$
$\beta = \|r_0\|$
$q_0 = r_0/\beta$
repeat
  $v_k = Aq_k$
  for $j = 1\colon k$ **do**
    $h_{jk} = q_j^T v_k$
    $v_k = v_k - h_{jk} q_j$
  end for
  $h_{k+1,k} = \|v_k\|$
  $q_{k+1} = v_k / h_{k+1,k}$
  Solve $\min_{y_k} \|r_k\| = \|\beta e_1 - Hy_k\|$.
until $\|r_k\|$ is small enough
$x_k = x_0 + Q_k y_k$

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

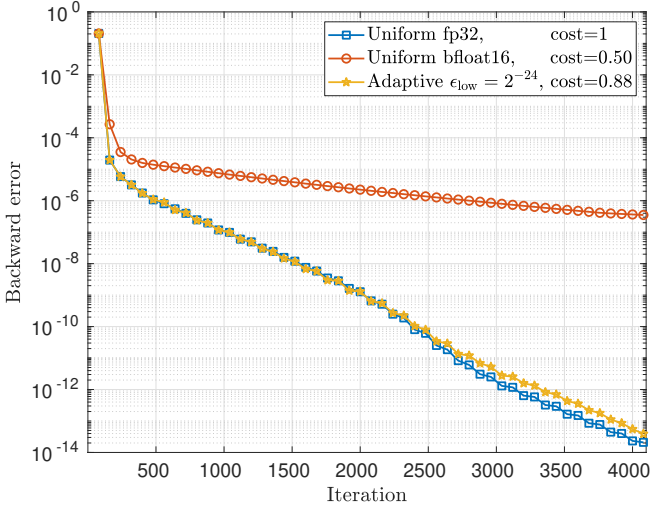
GMRES-IR with adaptive precision SpMV

$r_0 = b - Ax_0$
$\beta = \|r_0\|$
$q_0 = r_0/\beta$
repeat
  $v_k = Aq_k \rightarrow \epsilon_{\text{low}}$
  for $j = 1\colon k$ **do**
    $h_{jk} = q_j^T v_k$
    $v_k = v_k - h_{jk} q_j$
  end for
  $h_{k+1,k} = \|v_k\|$
  $q_{k+1} = v_k / h_{k+1,k}$
  Solve $\min_{y_k} \|r_k\| = \|\beta e_1 - Hy_k\|$.
until $\|r_k\|$ is small enough
$x_k = x_0 + Q_k y_k$

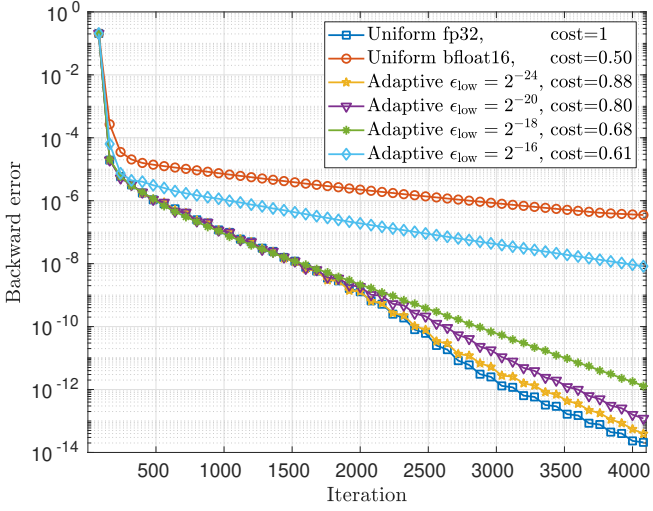
GMRES-IR

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

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



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



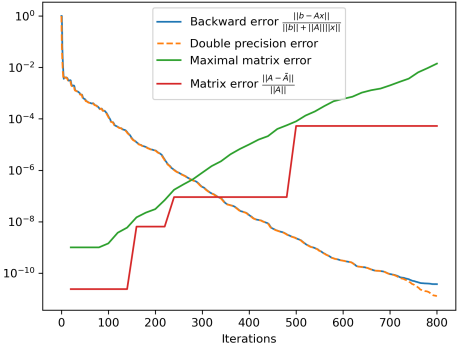
Relaxed GMRES (Giraud, Gratton, Langou, 2009)

Stability up to $O(\varepsilon)$ is maintained if, at each iteration k, the matvec is performed with $\widetilde{A}_k = A + E_k$ such that

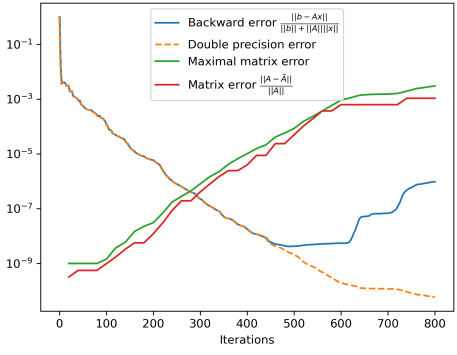
$$\frac{\|E_k\|}{\|A\|} \leq \frac{1}{n\kappa(A)} \frac{\|b\|}{\|r_{k-1}\|} \varepsilon$$

- Matvec precision can be reduced to be inversely proportional to the residual norm $\Rightarrow$ lower and lower precision as iterations progress

Relaxed GMRES with adaptive SpMV

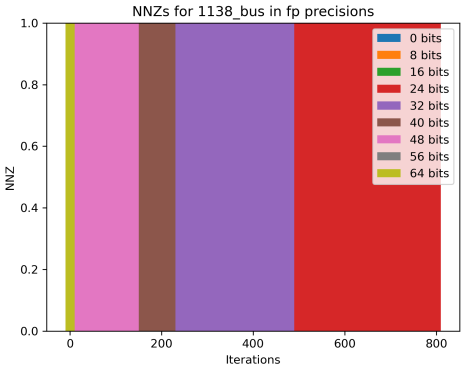


Relaxed

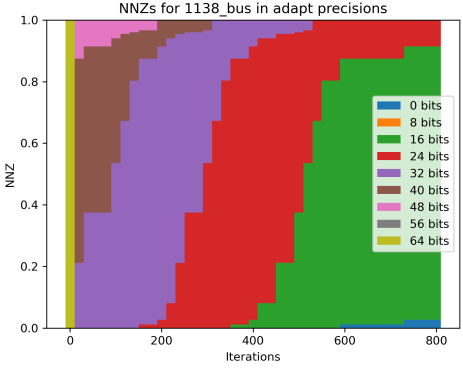


Relaxed+adaptive

Relaxed GMRES with adaptive SpMV



Relaxed



Relaxed+adaptive

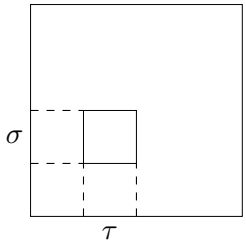
Precision emulation

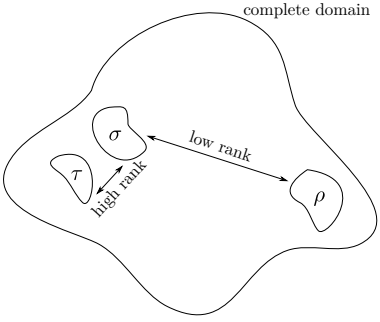
Linear systems

Block low-rank approximations

Data sparse matrices

Data sparse matrices generalize numerically sparse matrices: approximate entire blocks rather than single entries



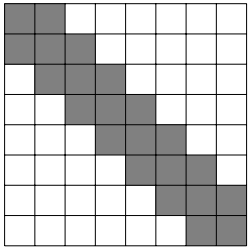


A block B represents the interaction between two subdomains σ and τ.

Large distance $\Leftrightarrow$ low numerical rank,

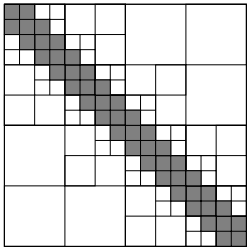
even for small ϵ!

Many different block partitionings possible



BLR matrix

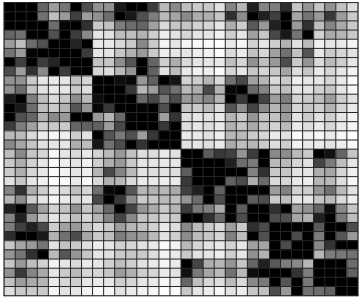
- Simple, flat structure
- Superlinear complexity



$\mathcal{H}$-matrix

- Complex, hierarchical structure
- Near-optimal loglinear complexity

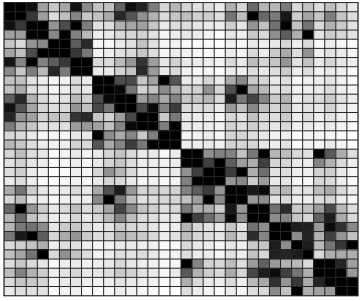
BLR matrices use a flat 2D block partitioning Amestoy et al. (2019)



Example of a BLR matrix (Schur complement of a 64^3 Poisson problem with block size 128)

- Diagonal blocks are full rank
- Off-diagonal ones are stored in low rank form if their ϵ-rank is small enough
- $\epsilon = 10^{-15} \rightarrow 50\%$ entries kept

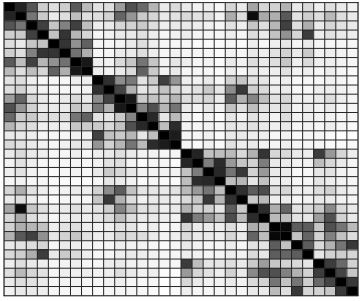
BLR matrices use a flat 2D block partitioning Amestoy et al. (2019)



Example of a BLR matrix (Schur complement of a 64^3 Poisson problem with block size 128)

- Diagonal blocks are full rank
- Off-diagonal ones are stored in low rank form if their ϵ-rank is small enough
- $\epsilon = 10^{-15} \rightarrow 50\%$ entries kept
- $\epsilon = 10^{-12} \rightarrow 36\%$ entries kept

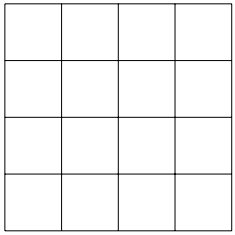
BLR matrices use a flat 2D block partitioning Amestoy et al. (2019)



Example of a BLR matrix (Schur complement of a 64^3 Poisson problem with block size 128)

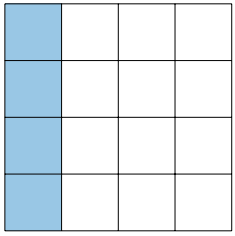
- Diagonal blocks are full rank
- Off-diagonal ones are stored in low rank form if their ϵ-rank is small enough
- $\epsilon = 10^{-15} \rightarrow 50\%$ entries kept
- $\epsilon = 10^{-12} \rightarrow 36\%$ entries kept
- $\epsilon = 10^{-9} \rightarrow 23\%$ entries kept

Standard BLR factorization



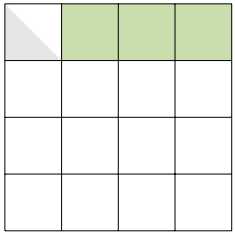
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



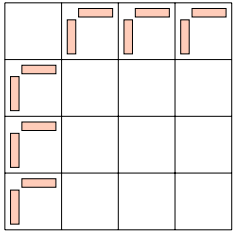
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



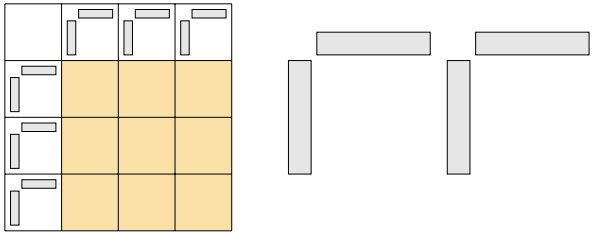
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



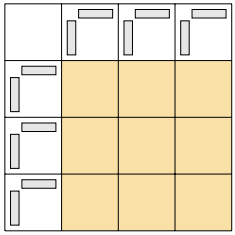
- Adapt blocked LU algorithm to exploit low-rank blocks

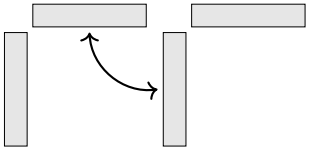
Standard BLR factorization



- Adapt blocked LU algorithm to exploit low-rank blocks

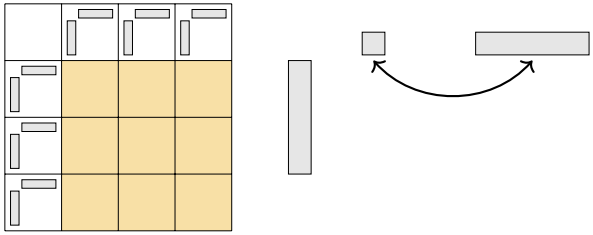
Standard BLR factorization





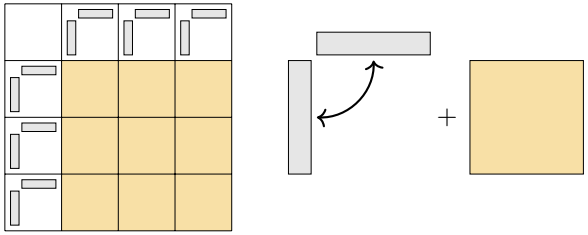
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



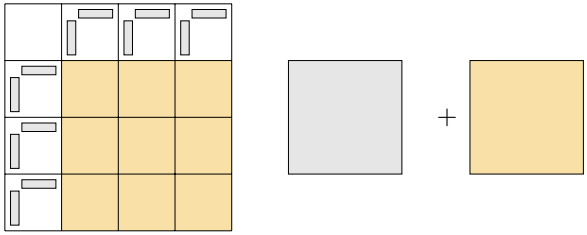
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



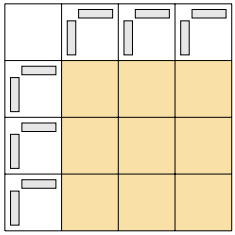
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization

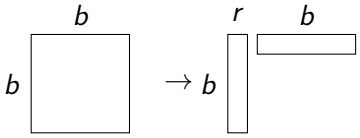


- Adapt blocked LU algorithm to exploit low-rank blocks
- Reduced LU factorization complexity Amestoy et al. (2017)

		Flops	Memory
Dense	Full-rank	$O(n^3)$	$O(n^2)$
Dense	BLR	$O(n^2)$	$O(n^{1.5})$
Sparse*	Full-rank	$O(n^2)$	$O(n^{\frac{4}{3}})$
Sparse*	BLR	$O(n^{\frac{4}{3}})$	$O(n \log n)$

* for 3D cubic problems

Challenges with data sparse algorithms



- Low granularities: low-rank matrices have much smaller granularities ($r \ll b$), making computations inefficient (BLAS-2 $\rightarrow$ BLAS-3, less data reuse)
- Memory/communication-boundness: multiplying two $b \times b$ dense matrices costs $2b^3$ flops, whereas multiplying two $b \times b$ rank-r matrices costs $4br^2$ flops. Hence
  - Flops ratio: $\frac{1}{2} \cdot \left(\frac{b}{r}\right)^2$ Ex: $r = b/10 \Rightarrow 50\times$ less flops
  - BUT storage ratio: $\frac{1}{2} \cdot \frac{b}{r}$ Ex: $r = b/10 \Rightarrow 5\times$ less storage

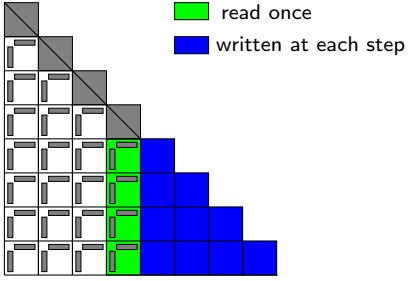
  $\Rightarrow$ relative weight of memory/communications much higher than for dense computations!

Right-looking Vs. Left-looking BLR

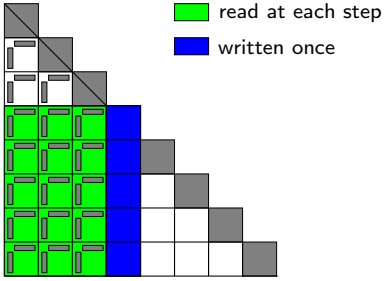
	FR time		BLR time	
	RL	LL	RL	LL
Update	338	336	110	67
Total	424	421	221	175

Right-looking Vs. Left-looking BLR

	FR time		BLR time	
	RL	LL	RL	LL
Update	338	336	110	67
Total	424	421	221	175



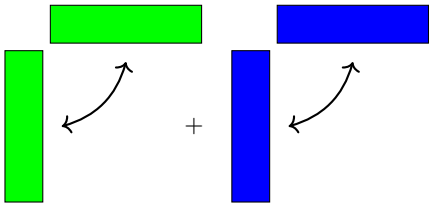
RL factorization

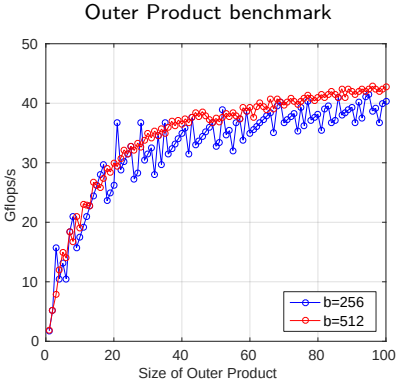


LL factorization

⇒ Lower volume of memory transfers in LL

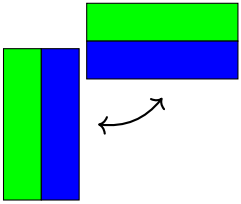
Low-rank update accumulation (LUA)

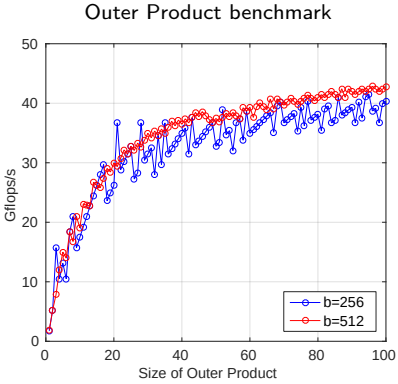




		LL
average size of Outer Product		16.5
time (s)	Outer Product	21
	Total	175

Low-rank update accumulation (LUA)





		LL	LUA
average size of Outer Product		16.5	61.0
time (s)	Outer Product	21	14
	Total	175	167

Impact of machine properties on BLR: roofline model

	specs		time (s) for BLR factorization		
	peak (GF/s)	bw (GB/s)	RL	LL	LUA
grunch (28 threads)	37	57	248	228	196
brunch (24 threads)	46	102	221	175	167

Impact of machine properties on BLR: roofline model

	specs		time (s) for BLR factorization		
	peak (GF/s)	bw (GB/s)	RL	LL	LUA
grunch (28 threads)	37	57	248	228	196
brunch (24 threads)	46	102	221	175	167

Arithmetic Intensity in BLR:

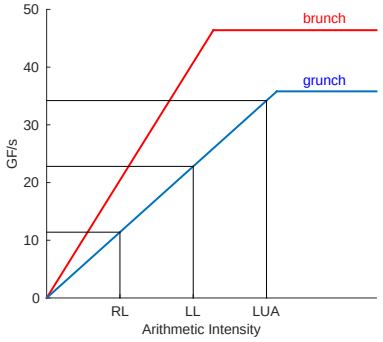
- LL $>$ RL (lower volume of memory transfers)
- LUA $>$ LL (higher granularities $\Rightarrow$ more efficient cache use)

Impact of machine properties on BLR: roofline model

	specs		time (s) for BLR factorization		
	peak (GF/s)	bw (GB/s)	RL	LL	LUA
grunch (28 threads)	37	57	248	228	196
brunch (24 threads)	46	102	221	175	167

Arithmetic Intensity in BLR:

- LL $>$ RL (lower volume of memory transfers)
- LUA $>$ LL (higher granularities $\Rightarrow$ more efficient cache use)

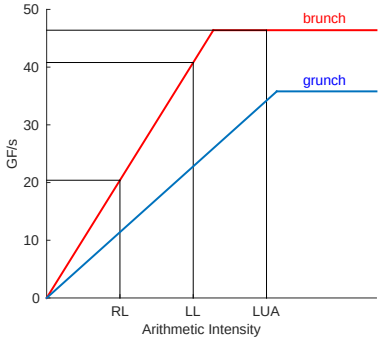


Impact of machine properties on BLR: roofline model

	specs		time (s) for BLR factorization		
	peak (GF/s)	bw (GB/s)	RL	LL	LUA
grunch (28 threads)	37	57	248	228	196
brunch (24 threads)	46	102	221	175	167

Arithmetic Intensity in BLR:

- LL $>$ RL (lower volume of memory transfers)
- LUA $>$ LL (higher granularities $\Rightarrow$ more efficient cache use)



Combining (data) sparsity and low precisions

Two approaches: coarse and fine grain mixed precision

- **Coarse grain mixed precision:** Run baseline algorithm in low precision, refine the result to high precision
  - ☺ Simple and efficient, can rely on optimized libraries
  - ☹ Extra work for the refinement, not always guaranteed to work
- **Fine grain mixed precision:** Adapt the precision of each instruction/operation to achieve a given accuracy target
  - ☺ Optimal use of low precision, with guaranteed target accuracy
  - ☹ Much more intrusive, may be less efficient

Iterative refinement with BLR LU (coarse grain)

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

Example on tminlet3M matrix
fp64 LU reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

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

Iterative refinement with BLR LU (coarse grain)

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

Example on tminlet3M matrix
fp64 LU reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

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

Iterative refinement with BLR LU (coarse grain)

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

Example on tminlet3M matrix
fp64 LU reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

	time (s)		memory (GB)	
	LU-IR	GMRES-IR	LU-IR	GMRES-IR
FR	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

Iterative refinement with BLR LU (coarse grain)

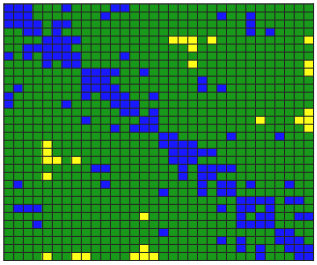
Error analysis: replace $\mathbf{u_f}$ by $\mathbf{u_f} + \epsilon$ in the convergence conditions

Example on tminlet3M matrix
fp64 LU reference: time $\rightarrow$ 295.5 memory $\rightarrow$ 241.1

	time (s)		memory (GB)	
	LU-IR	GMRES-IR	LU-IR	GMRES-IR
FR	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!

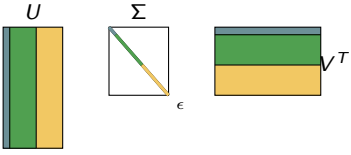
Adaptive precision BLR compression (fine grain)



fp64 fp32 fp16

- Off-diagonal blocks represent weak interactions, "less important"

$\Rightarrow$ Can we store some blocks in lower precisions ?



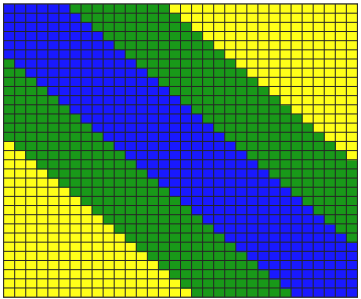
- Singular values decay rapidly, the first few are the "most important"

$\Rightarrow$ Can we switch some singular vectors to lower precisions ?

Adaptive precision BLR matrices

Idea: store blocks far away from the diagonal in lower precisions

Abdulah et al. (2019) Doucet et al. (2019) Abdulah et al. (2021)

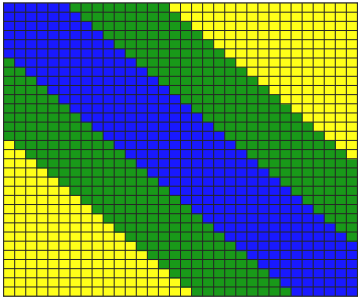


- **fp64**
- **fp32**
- **fp16**

Adaptive precision BLR matrices

Idea: store blocks far away from the diagonal in lower precisions

Abdulah et al. (2019) Doucet et al. (2019) Abdulah et al. (2021)



- **fp64**
- **fp32**
- **fp16**

Error analysis:

- Converting A_{ij} to precision u_{low} introduces an error $u_{\text{low}}\|A_{ij}\|$

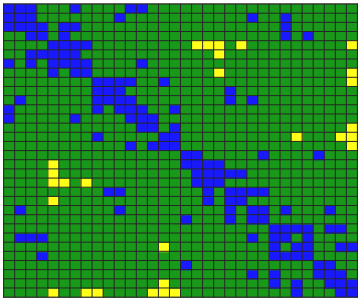
$\Rightarrow$ If $\|A_{ij}\| \leq \epsilon\|A\|/u_{\text{low}}$, block can be stored in precision u_{low}

Adaptive precision BLR matrices

Idea: store blocks far away from the diagonal in lower precisions

Abdulah et al. (2019) Doucet et al. (2019) Abdulah et al. (2021)

(Poisson, $\epsilon = 10^{-10}$)



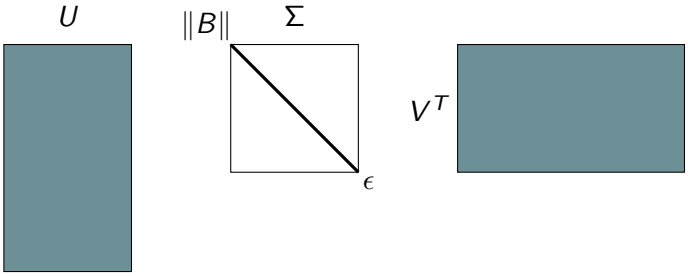
- **fp64**
- **fp32**
- **fp16**

Error analysis:

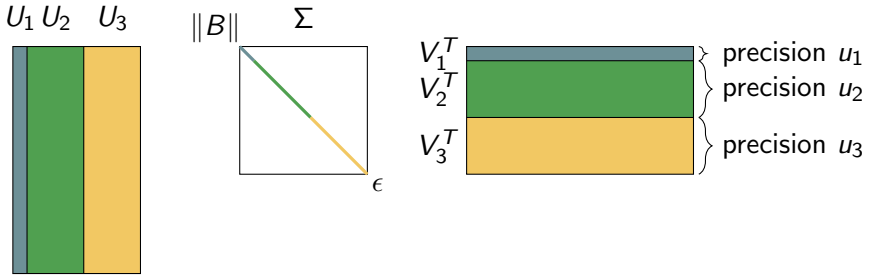
- Converting A_{ij} to precision u_{low} introduces an error $u_{\text{low}}\|A_{ij}\|$

$\Rightarrow$ If $\|A_{ij}\| \leq \epsilon\|A\|/u_{\text{low}}$, block can be stored in precision u_{low}

Adaptive precision low rank compression

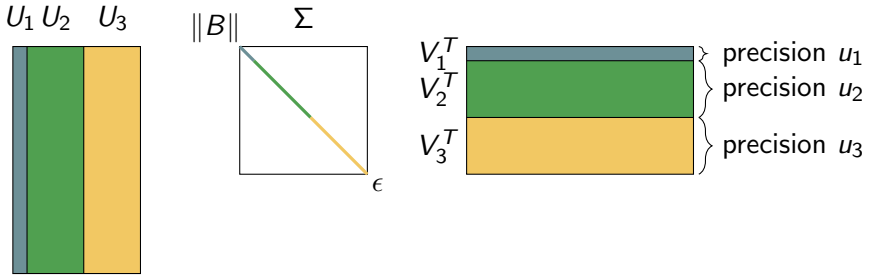


Adaptive precision low rank compression



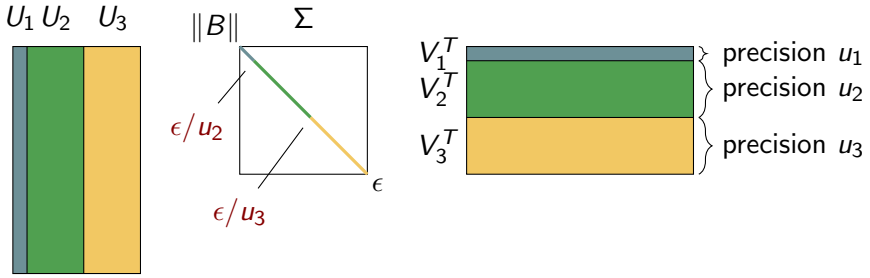
- **Adaptive precision compression:** partition U and V into q groups of decreasing precisions $u_1 \leq \epsilon < u_2 < \ldots < u_q$

Adaptive precision low rank compression



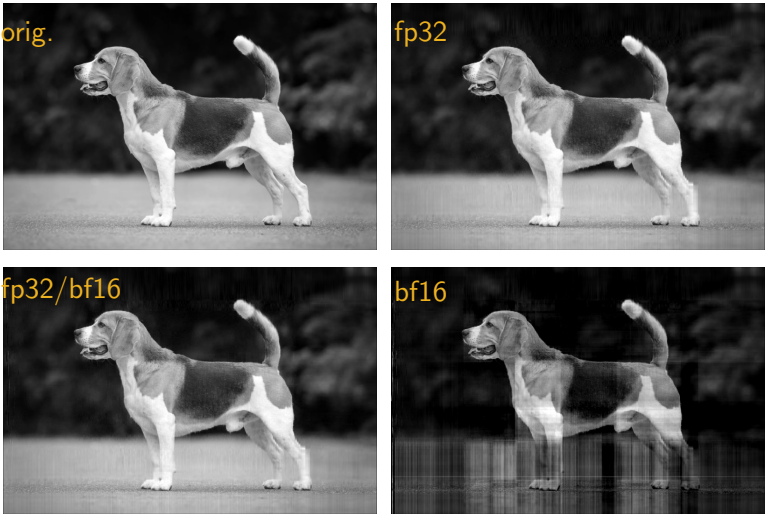
- **Adaptive precision compression:** partition U and V into q groups of decreasing precisions $u_1 \leq \epsilon < u_2 < \ldots < u_q$
- Why does it work? $B = \mathbf{B_1} + \mathbf{B_2} + \mathbf{B_3}$ with $|B_i| \leq O(\|\Sigma_i\|)$

Adaptive precision low rank compression



- **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 $|B_i| \leq O(\|\Sigma_i\|)$
- With p precisions and a partitioning such that $\|\Sigma_k\| \leq \epsilon\|B\|/u_k$, $\|B - \widehat{U}_\epsilon \Sigma_\epsilon \widehat{V}_\epsilon\| \lesssim (2p-1)\epsilon\|B\|$

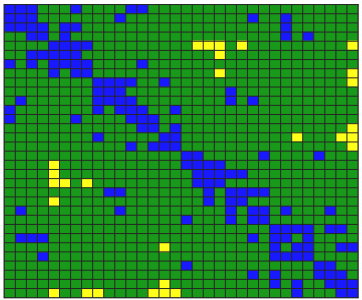
Experiments: Julia, image compression



With $\varepsilon = 0.04$ the rank is 191 but only 13 steps are done in fp32 and the rest in bf16 (original size is 1057×1600)

Back to adaptive precision BLR matrices

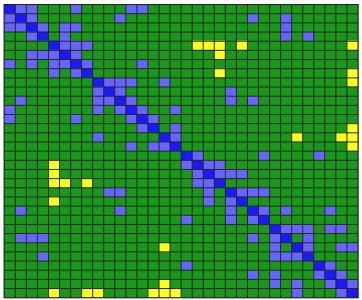
(Poisson, $\epsilon = 10^{-10}$)



- **fp64**
- **fp32**
- **fp16**

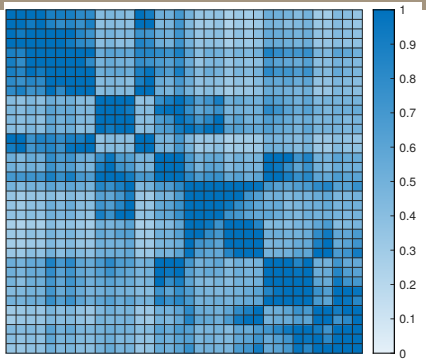
Back to adaptive precision BLR matrices

(Poisson, $\epsilon = 10^{-10}$)

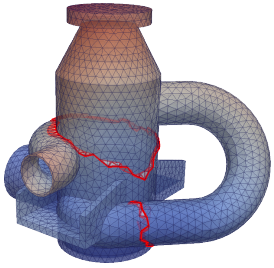


- **fp64** $\Rightarrow$ **fp64**, **fp64/fp32/fp16**
- **fp32** $\Rightarrow$ **fp32/fp16**
- **fp16**

Adaptive precision BLR compression



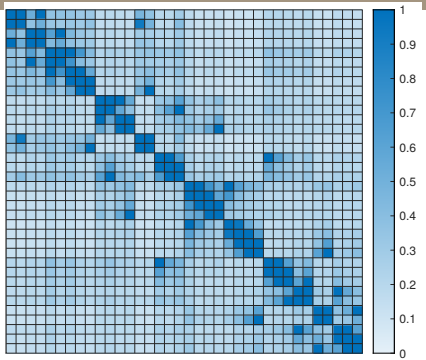
Normalized storage cost of each block



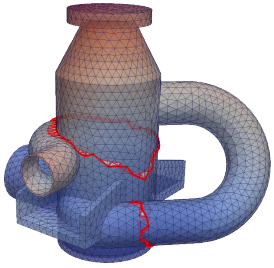
Matrix perf009d
(RIS pump from EDF)

100% entries in fp64

Adaptive precision BLR compression



Normalized storage cost of each block



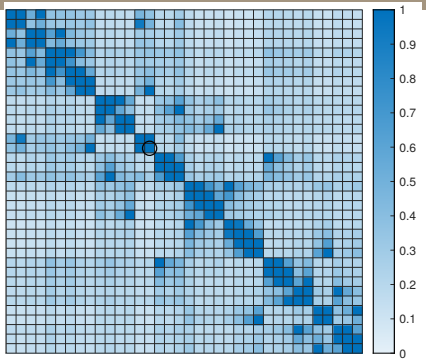
Matrix perf009d
(RIS pump from EDF)

100% entries in fp64

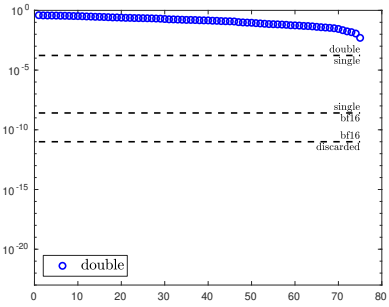
$\rightarrow$ 13% in fp64
53% in fp32
33% in bfloat16

$\Rightarrow 2\times$ storage reduction

Adaptive precision BLR compression



Normalized storage cost of each block



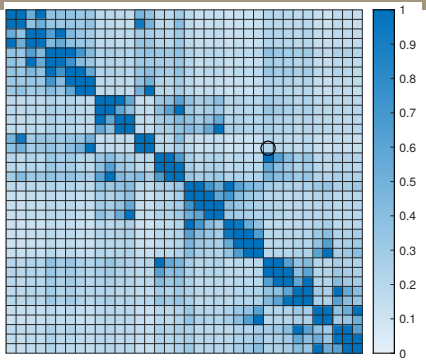
block (15,15)

100% entries in fp64

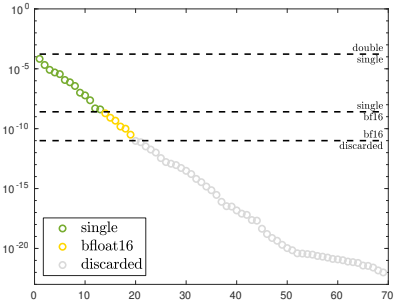
$$\rightarrow \begin{cases} 13\% \text{ in fp64} \\ 53\% \text{ in fp32} \\ 33\% \text{ in bfloat16} \end{cases}$$

$\Rightarrow 2\times$ storage reduction

Adaptive precision BLR compression



Normalized storage cost of each block



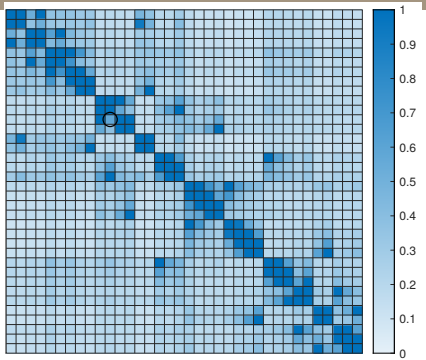
block (15,27)

100% entries in fp64

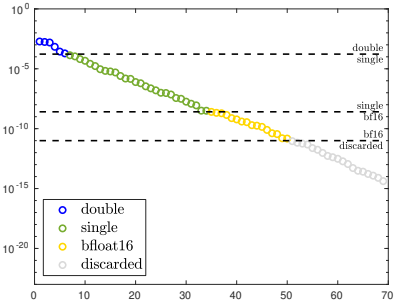
$$\rightarrow \begin{cases} 13\% \text{ in fp64} \\ 53\% \text{ in fp32} \\ 33\% \text{ in bfloat16} \end{cases}$$

$\Rightarrow 2\times$ storage reduction

Adaptive precision BLR compression



Normalized storage cost of each block



block (12,11)

100% entries in fp64

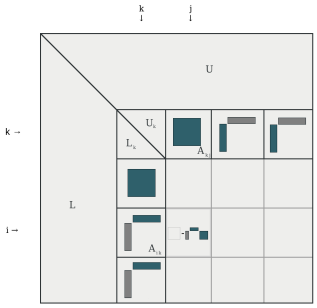
$\rightarrow \begin{cases} 13\% \text{ in fp64} \\ 53\% \text{ in fp32} \\ 33\% \text{ in bfloat16} \end{cases}$

$\Rightarrow 2\times$ storage reduction

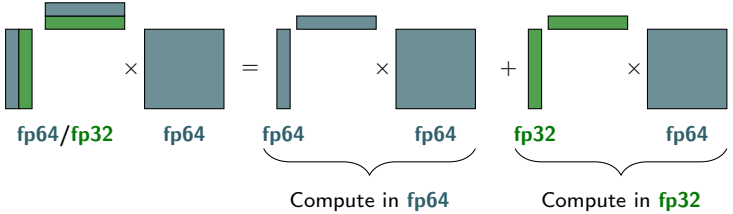
Adaptive precision BLR LU factorization

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

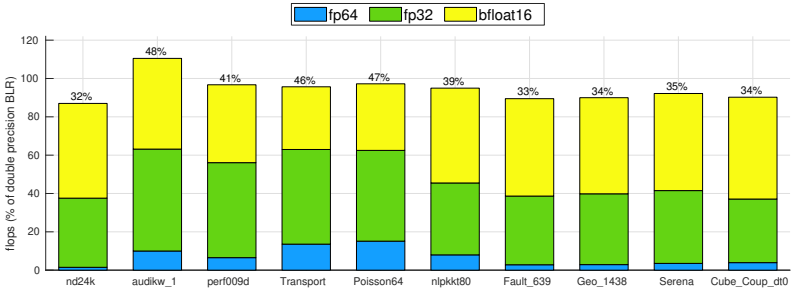
- Stability of LU factorization: $\widehat{L}\widehat{U} = A + \Delta A$
  - **Standard LU** : $\|\Delta A\| \lesssim 3n^3\rho_n u_1 \|A\|$
  - **BLR LU** : $\|\Delta A\| \lesssim (c_1\epsilon + c_2\rho_n u_1)\|A\|$
  - **Adaptive prec. BLR LU** : $\|\Delta A\| \lesssim (c_1'\epsilon + c_2'\rho_n u_1)\|A\|$



Example of kernel: LR × matrix multiplication:



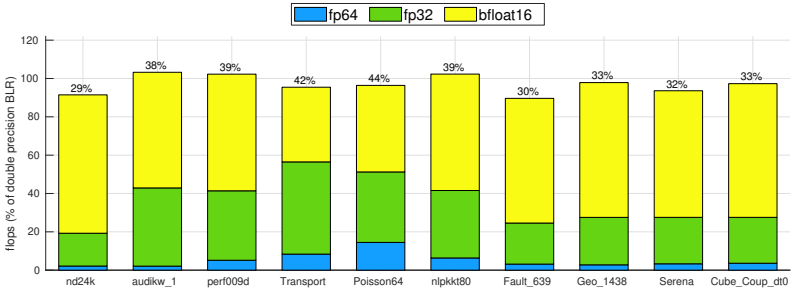
Adaptive precision BLR LU factorization



$\varepsilon = 10^{-12}$

Top of the bars: cost w.r.t. fp64 BLR, assuming
1 flop(fp64) = 2 flops(fp32) = 4 flops(bfloat16)

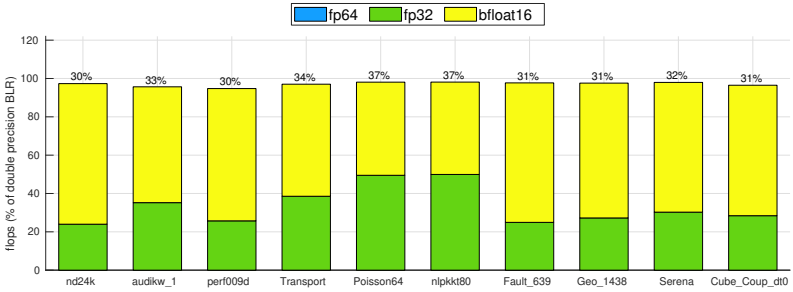
Adaptive precision BLR LU factorization



$\varepsilon = 10^{-9}$

Top of the bars: cost w.r.t. fp64 BLR, assuming
1 flop(fp64) = 2 flops(fp32) = 4 flops(bfloat16)

Adaptive precision BLR LU factorization

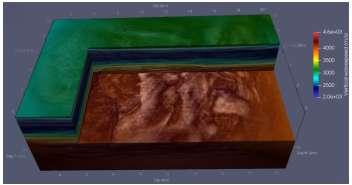


$\varepsilon = 10^{-6}$

Top of the bars: cost w.r.t. fp64 BLR, assuming
1 flop(fp64) = 2 flops(fp32) = 4 flops(bfloat16)

Performance impact illustration

- Adastra MUMPS4FWI project led by WIND team [Operto et al., The Leading Edge 2023]
- Application: Gorgon Model, reservoir 23km x 11km x 6.5km, grid size 15m, Helmholtz equation, 25-Hz
- Complex matrix, 531 Million dofs, storage(A)=220 GBytes;
- FR cost: flops for one LU factorization= 2.6×10^{18};
  Estimated storage for LU factors= 73 TBytes



(25-Hz Gorgon FWI velocity model)

FR (Full-Rank); BLR with $\varepsilon = 10^{-5}$; 48 000 cores (500 MPI $\times$ 96 threads/MPI)
FR: fp32; Adaptive precision BLR: 3 precisions (32bits, 24bits, 16bits) for storage

LU size (TBytes)			Flops		Time BLR + Mixed (sec)			Scaled Resid.
FR	BLR	+adapt.	FR	BLR+adapt.	Analysis	Facto	Solve	BLR+adapt.
73	34	26	2.6×10^{18}	0.5×10^{18}	446	5500	27	7×10^{-4}

Efficiency on 48 000 cores?

- Theoretical peak: 3686 TFLOPS (48000 $\times$ 2.4GHz $\times$ 2 (fp32) $\times$ 16 flop/cycle)
- Speed w.r.t. BLR flops: 364 TFLOPS (10% of the peak) ($0.5 \times 10^{18} \times 4$ (complex)$/5500/10^{12}$)