

Harnessing inexactness in scientific computing

Lecture 15: numerical computing on GPUs

Theo Mary (CNRS)

theo.mary@lip6.fr

<https://perso.lip6.fr/Theo.Mary/>

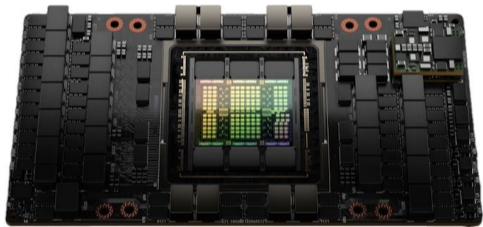
Elisa Riccietti (ENS Lyon)

elisa.riccietti@ens-lyon.fr

<https://perso.ens-lyon.fr/elisa.riccietti/>

M2 course at ENS Lyon, 2024–2025

Slides available on course webpage



Introduction

Rounding error analysis

Iterative refinement

Multiword matrix multiplication

Randomized LRA

Introduction

Rounding error analysis

Iterative refinement

Multiword matrix multiplication

Randomized LRA

	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

		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}
bf16		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}

Peak performance (TFLOPS)					
	Pascal 2016	Volta 2018	Ampere 2020	Hopper 2022	Blackwell 2025
fp64	5	8	20	67	40
fp32	10	16	20	67	80
tfloat32	--	--	160	495	2,200
fp16/bfloat16	20	125	320	990	4,500
fp8	--	--	--	2,000	9,000
fp4	--	--	--	--	18,000

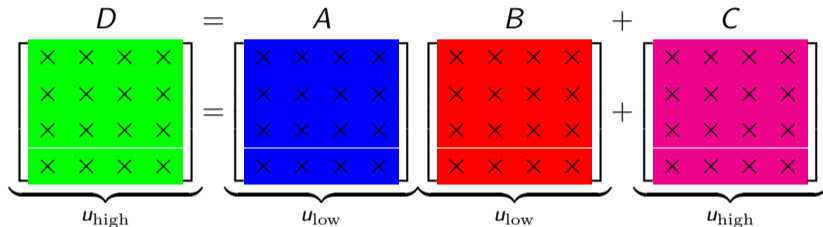


NVIDIA Hopper (H100) GPU

fp64/fp16 speed ratio:

- Hopper (2022): 15×
- Blackwell (2025): 112×

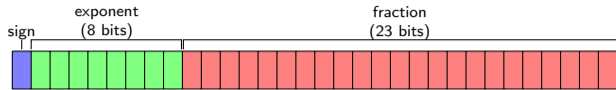
Tensor cores units available on NVIDIA GPUs carry out a mixed precision matrix multiply-accumulate ($u_{\text{high}} \equiv \text{fp32}$ and $u_{\text{low}} \equiv \text{fp16}/\text{fp8}/\text{fp4}$)



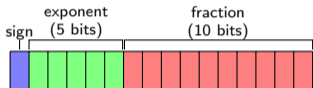
Element-wise multiplication of matrix A and B is performed with at least single precision. When `.ctype` or `.dtype` is `.f32`, accumulation of the intermediate values is performed with at least single precision. When both `.ctype` and `.dtype` are specified as `.f16`, the accumulation is performed with at least half precision.

The accumulation order, rounding and handling of subnormal inputs is unspecified.

Tensor core precisions



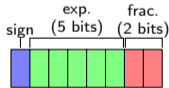
fp32
Range $10^{\pm 38}$, $u = 6 \times 10^{-8}$



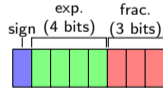
fp16
Range $10^{\pm 5}$, $u = 5 \times 10^{-4}$



bfloat16
Range $10^{\pm 38}$, $u = 4 \times 10^{-3}$



fp8 (e5m2)
Range $10^{\pm 5}$, $u = 0.125$



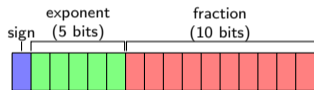
fp8 (e4m3)
Range $[0.015, 448]$, $u = 6 \times 10^{-2}$



fp4 (e2m1)
Range $[1, 6]$, $u = 0.25$



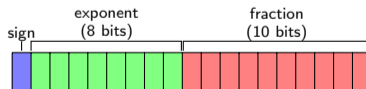
fp32
Range $10^{\pm 38}$, $u = 6 \times 10^{-8}$



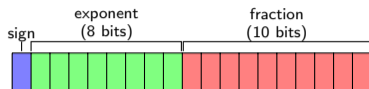
fp16
Range $10^{\pm 5}$, $u = 5 \times 10^{-4}$



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

Introduction

Rounding error analysis

Iterative refinement

Multiword matrix multiplication

Randomized LRA

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}

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

First, we convert A and B to low precision:

$$\tilde{A} = \text{fl}_{\text{low}}(A) = A + \Delta A, \quad |\Delta A| \leq u_{\text{low}}|A|,$$

$$\tilde{B} = \text{fl}_{\text{low}}(B) = B + \Delta B, \quad |\Delta B| \leq u_{\text{low}}|B|.$$

Second, we compute the product:

$$\begin{aligned}\hat{C} &= \tilde{A}\tilde{B} + \Delta C, & |\Delta C| &\lesssim nu_{\text{high}}|\tilde{A}||\tilde{B}|, \\ &= AB + \Delta AB + A\Delta B + \Delta A\Delta B + \Delta C \\ &= AB + E, & |E| &\lesssim \left(\underbrace{2u_{\text{low}}}_{\text{Conversion}} + \underbrace{nu_{\text{high}}}_{\text{Accumulation}} \right) |A||B|\end{aligned}$$

Matrix multiplication: error analysis

First, we convert A and B to low precision:

$$\begin{aligned}\tilde{A} &= \text{fl}_{\text{low}}(A) = A + \Delta A, & |\Delta A| &\leq u_{\text{low}}|A|, \\ \tilde{B} &= \text{fl}_{\text{low}}(B) = B + \Delta B, & |\Delta B| &\leq u_{\text{low}}|B|.\end{aligned}$$

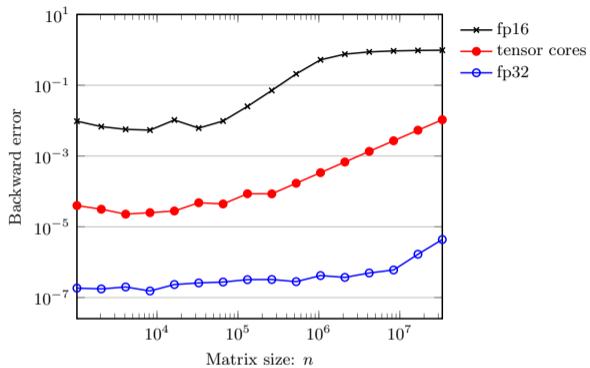
Second, we compute the product:

$$\begin{aligned}\hat{C} &= \tilde{A}\tilde{B} + \Delta C, & |\Delta C| &\lesssim nu_{\text{high}}|\tilde{A}||\tilde{B}|, \\ &= AB + \Delta AB + A\Delta B + \Delta A\Delta B + \Delta C \\ &= AB + E, & |E| &\lesssim \left(\underbrace{2u_{\text{low}}}_{\text{Conversion}} + \underbrace{nu_{\text{high}}}_{\text{Accumulation}} \right) |A||B|\end{aligned}$$

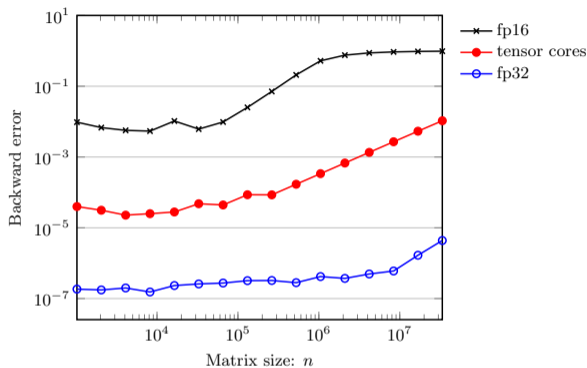
Evaluation method	Bound
Standard in precision u_{low}	nu_{low}
Standard in precision u_{high}	nu_{high}
Tensor cores	$2u_{\text{low}} + nu_{\text{high}}$

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

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 \(2021\)](#)

Introduction

Rounding error analysis

Iterative refinement

Multiword matrix multiplication

Randomized LRA

Factorize $A = LU$ in precision \mathbf{u}_f
 Solve $Ax_0 = b$ via $x_0 = U^{-1}(L^{-1}b)$ 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 $\mathbf{u}_f \equiv \text{fp16}$, $\mathbf{u} \equiv \text{fp32}$, and $\mathbf{u}_r \equiv \text{fp64}$
 or $\mathbf{u}_f \equiv \text{fp16}$, $\mathbf{u} \equiv \text{fp64}$, and $\mathbf{u}_r \equiv \text{fp128}$

 Carson and Higham (2018)

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

- 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  
       $A_{ij} \leftarrow A_{ij} - L_{ik}U_{kj}$   
    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  $U_{ki}$   
     $\tilde{L}_{ik} \leftarrow \text{fl}_{16}(L_{ik})$  and  $\tilde{U}_{ki} \leftarrow \text{fl}_{16}(U_{ki})$   
  end for  
  for  $i = k + 1: n/b$  do  
    for  $j = k + 1: n/b$  do  
       $A_{ij} \leftarrow A_{ij} - \tilde{L}_{ik}\tilde{U}_{kj}$  using tensor cores  
    end for  
  end for  
end for
```

LU factorization with tensor cores

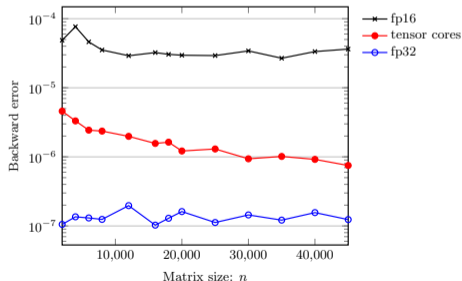
Error analysis for LU follows from matrix multiplication analysis and gives same bounds to first order [Blanchard, Higham, Lopez, M., Pranesh \(2020\)](#)

Standard fp16	Tensor cores	Standard fp32
---------------	--------------	---------------

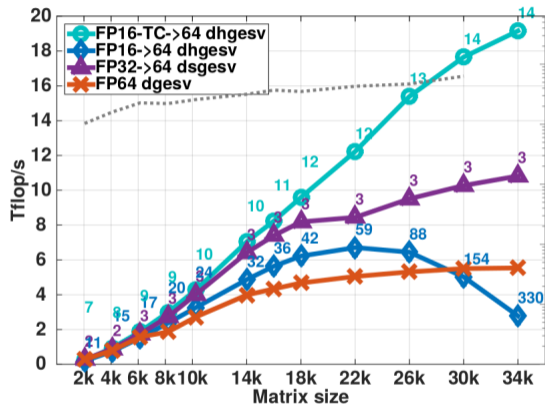
nu_{16}

$2u_{16} + nu_{32}$

nu_{32}



Results from [Haidar et al. \(2018\)](#)

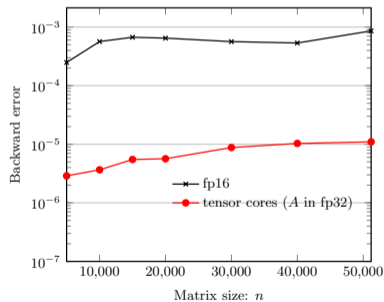
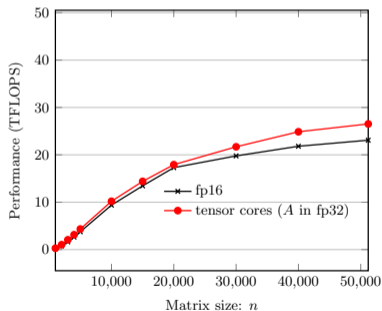


- TC accuracy boost can be critical!
- TC performance suboptimal here \Rightarrow why?

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**

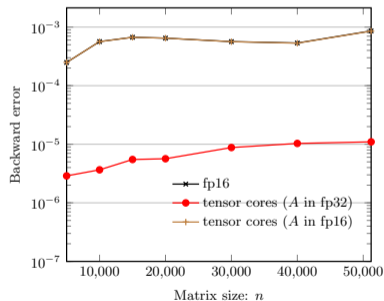
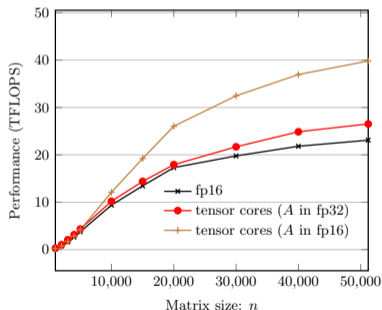
\Rightarrow LU with tensor cores becomes memory-bound !



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 !



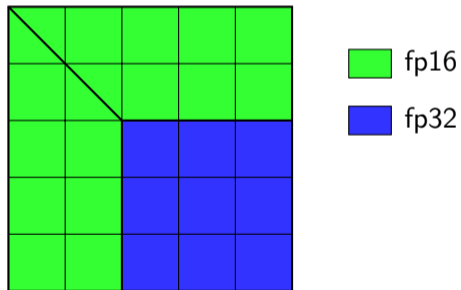
- Idea: **store matrix in fp16**
- Problem: **huge accuracy loss**, tensor cores accuracy boost completely negated

Two ingredients to **reduce data movement with no accuracy loss**:

Two ingredients to **reduce data movement with no accuracy loss**:

1. Mixed fp16/fp32 representation

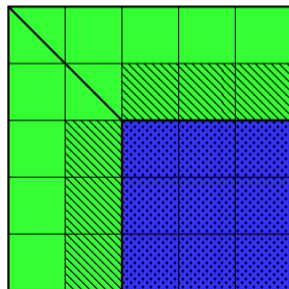
Matrix after 2 steps:



Two ingredients to **reduce data movement with no accuracy loss**:


1. Mixed fp16/fp32 representation

Matrix after 2 steps:



 fp16

 fp32

 read

 write

```
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
```

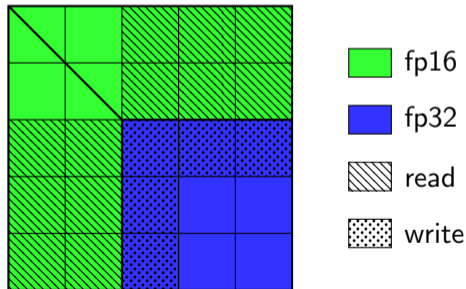
```
  end for
```

```
end for
```

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} .

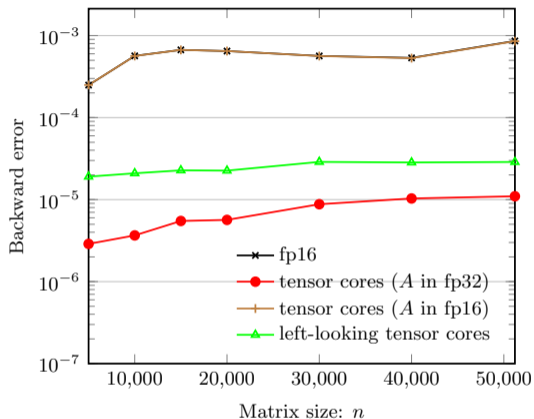
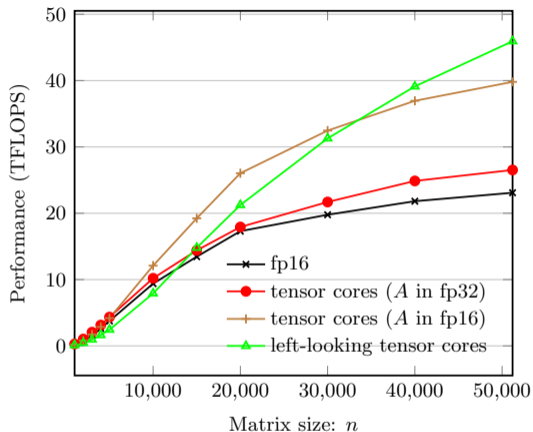
Solve $L_{kk} U_{ki} = A_{ki}$ for U_{ki} .

end for

end for

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

Experimental results

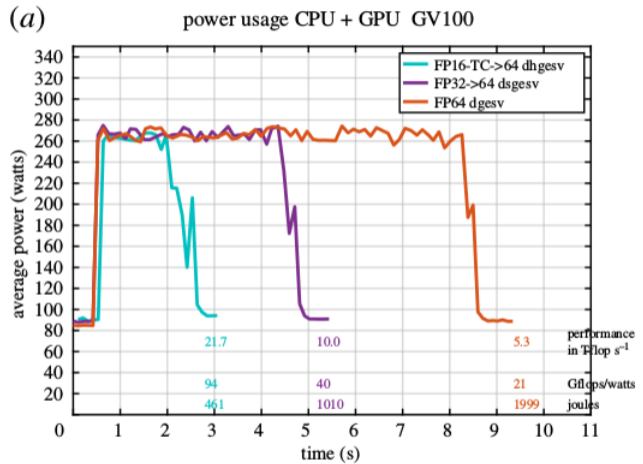



- Nearly **50 TFLOPS** without significantly impacting accuracy

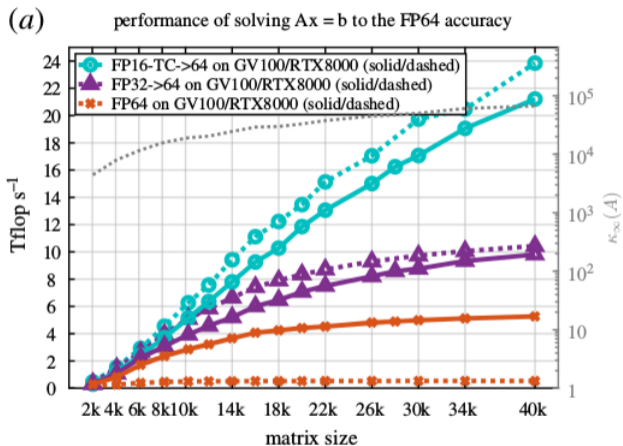
[Lopez and M. \(2023\)](#)

- Even more critical on A100:

50 TFLOPS (A in fp32) \rightarrow **175 TFLOPS** (A in fp16+left-looking)



 Haidar et al. (2020)



 Haidar et al. (2020)

Introduction

Rounding error analysis

Iterative refinement

Multiword matrix multiplication

Randomized LRA

- [Markidis et al. \(2018\)](#) propose “precision refinement”:

$$A_{32}B_{32} \approx A_{16}B_{16} + A_{16}R_B + R_A B_{16} + R_A R_B$$

where $R_A = \text{fp16}(A_{32} - A_{16})$ and $R_B = \text{fp16}(B_{32} - B_{16})$

- Reminiscent of multiword arithmetic (ex: double-double)
 - Represent high precision number as the unevaluated sum of lower precision numbers
 - **Double-double** arithmetic:

$$x = \underbrace{x_1}_{\text{fp64}} + \underbrace{x_2}_{\text{fp64}}$$

⇒ x has up to $2 \times 53 = 106$ significant bits $\approx 10^{-32}$ precision

- Less than fp128 (113 significant bits), but much faster, because computations rely on fp64 arithmetic
- Need for error-free transformations makes it much slower than fp64 ⇒ double-single arithmetic not meaningful on most processors

	Signif. bits	Exp. bits	Range	Unit roundoff u
fp32	24	8	$10^{\pm 38}$	6×10^{-8}
fp16	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16	8	8	$10^{\pm 38}$	4×10^{-3}

Let $x \in \mathbb{R}$ and $u_s = 2^{-24}$

$$x = \underbrace{x_1}_{\text{fp16}} + \underbrace{x_2}_{\text{fp16}} + \epsilon \quad |\epsilon| \leq 4u_s$$

$$x = \underbrace{x_1}_{\text{bfloat16}} + \underbrace{x_2}_{\text{bfloat16}} + \underbrace{x_3}_{\text{bfloat16}} + \epsilon \quad |\epsilon| \leq u_s$$

Double-half arithmetic with tensor cores

Apply this elementwise to $A \in \mathbb{R}^{m \times n}$ and $B \in \mathbb{R}^{n \times p}$:

$$A = A_1 + A_2, \quad B = B_1 + B_2$$

and compute $C = AB$ as

$$C \approx \sum_{i,j} A_i B_j \quad \text{using tensor cores}$$

GPU tensor cores provide a new perspective:

- Intermediate computations are done in fp32 \Rightarrow no need for error-free transformations!
- Double-fp16 \Rightarrow 4 \times more flops (can be reduced to 3 \times)
- Triple-bfloat16 \Rightarrow 9 \times more flops (can be reduced to 6 \times)
- Tensor cores 8 \times –16 \times faster than fp32

27/51 \Rightarrow **Multiword half arithmetic potentially faster at same accuracy!**

Step 1: build

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

to obtain

$$A = \sum_{i=1}^p A_i + \Delta A, \quad |\Delta A| \leq u_{\text{low}}^p |A|,$$

$$B = \sum_{j=1}^p B_j + \Delta B, \quad |\Delta B| \leq u_{\text{low}}^p |B|.$$

Step 2: compute the p^2 products $A_i B_j$ by chaining calls to the block FMA:

$$\widehat{C} = C + \Delta C, \quad |\Delta C| \lesssim (n + p^2) u_{\text{high}} |A| |B|.$$

Overall

$$\widehat{C} = AB + E, \quad |E| \lesssim (2u_{\text{low}}^p + u_{\text{low}}^{2p} + (n + p^2) u_{\text{high}}) |A| |B|.$$

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

$$|A_i| \leq u_{\text{low}}^{i-1} (1 + u_{\text{low}}) |A|$$

$$|B_j| \leq u_{\text{low}}^{j-1} (1 + u_{\text{low}}) |B|$$

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

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

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

- number of products: $p^2 \rightarrow p(p + 1)/2$
- error to order u_{low}^p : constant $2 \rightarrow p + 1$

$$\widehat{C} = AB + E, \quad |E| \lesssim ((p+1)u_{\text{low}}^p + nu_{\text{high}})|A||B|.$$

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

 Fasi, Higham, Lopez, M., Mikaitis (2023)

$$\widehat{C} = AB + E, \quad |E| \lesssim ((p+1)u_{\text{low}}^p + nu_{\text{high}})|A||B|.$$

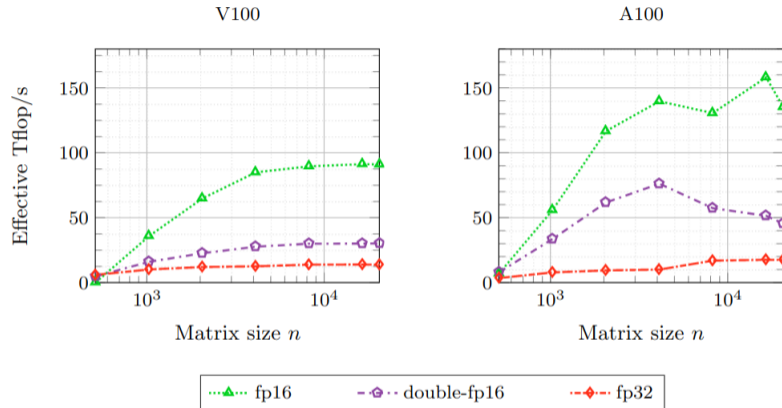
u_{high}	u_{low}		Error bound
2^{-24} (fp32)	2^{-11} (fp16)	$p = 1$	$2 \times 2^{-11} + n \times 2^{-24}$
		$p \geq 2$	$n \times 2^{-24}$
2^{-24} (fp32)	2^{-8} (bfloat16)	$p = 1$	$2 \times 2^{-8} + n \times 2^{-24}$
		$p = 2$	$3 \times 2^{-16} + n \times 2^{-24}$
		$p \geq 3$	$n \times 2^{-24}$

 Fasi, Higham, Lopez, M., Mikaitis (2023)

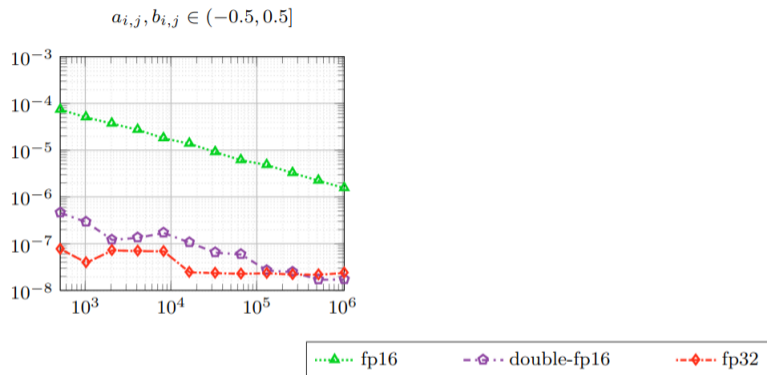
In the following we focus on **double-fp16** arithmetic:

$$AB \approx A_1B_1 + A_1B_2 + A_2B_1$$

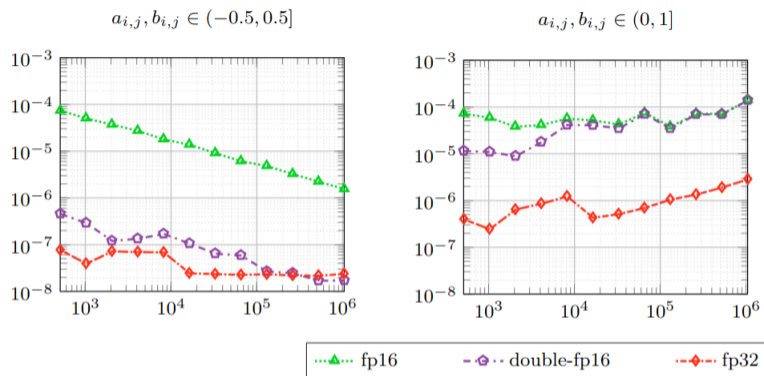
computed via three tensor core products



- Double-fp16 2–7 \times faster than fp32



- Double-fp16 2–7× faster than fp32
- Similar backward error for matrices with random $[-1, 1]$ uniform entries (decreasing error is expected)



- Double-fp16 2–7× faster than fp32
- Similar backward error for matrices with random $[-1, 1]$ uniform entries (decreasing error is expected)
- $[0, 1]$ uniform entries!!

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

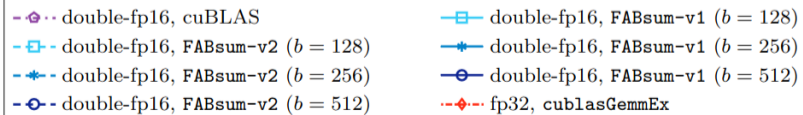
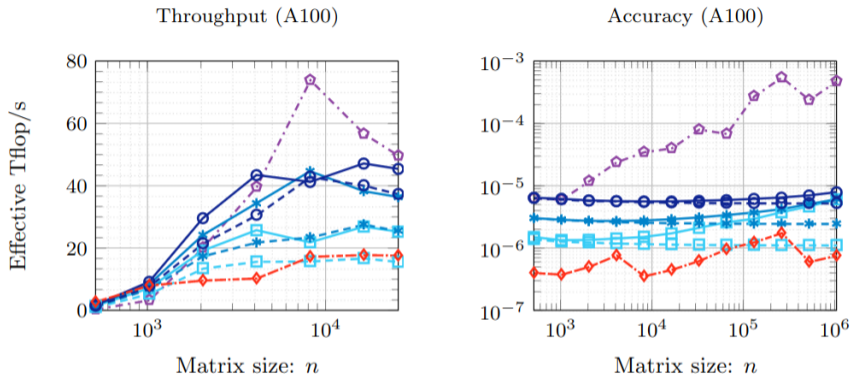
- fp32 uses the standard RTN, but tensor cores only support RZ [Fasi et al. \(2020\)](#)
 - With data of **nonzero mean** and RZ, most rounding errors happen in the **same direction**
- ⇒ Worst-case bound nu_{32} is attained with RZ, whereas with RTN we can usually replace it by $\sqrt{n}u_{32}$
- Same error bound \neq same error !

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

- fp32 uses the standard RTN, but tensor cores only support RZ [📄 Fasi et al. \(2020\)](#)
- With data of **nonzero mean** and RZ, most rounding errors happen in the **same direction**

⇒ Worst-case bound nu_{32} is attained with RZ, whereas with RTN we can usually replace it by $\sqrt{n}u_{32}$

- Same error bound \neq same error !
- A possible cure? The worst-case accumulation bound nu_{32} is attained \Rightarrow need to reduce the bound \Rightarrow **use blocked summation!**



Introduction

Rounding error analysis

Iterative refinement

Multiword matrix multiplication

Randomized LRA

- The following algorithm computes a rank- k LRA by using mainly matrix products
[📄 Halko, Martinsson, Tropp \(2011\)](#)

Input: $A \in \mathbb{R}^{m \times n}$, k , p

Output: $X \in \mathbb{R}^{m \times k}$, $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$

$\Omega \leftarrow \text{randn}(n, k + p)$

$B \leftarrow A\Omega$

$Q \leftarrow \text{qr}(B)$

$C \leftarrow Q^T A$

$ZY^T \leftarrow \text{LRA}(C, k)$

$X \leftarrow QZ$

- In the following, we consider $p = 0$ which simplifies the GPU implementation by replacing the last two lines by $X = Q$ and $Y = C^T$

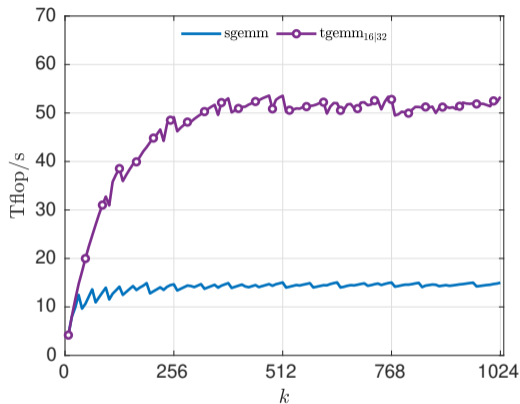
randLRA on GPU with `tgemm16|32`.

Input: $A_{32} \in \mathbb{R}^{m \times n}$, the target rank k .

Output: $X_{16} \in \mathbb{R}^{m \times k}$ and $Y_{16} \in \mathbb{R}^{n \times k}$ such that $A_{32} \approx X_{16} Y_{16}^T$.

- 1: $\Omega_{16} = \text{randn}(n, k)$
 - 2: $A_{16} = \text{fp16}(A_{32})$
 - 3: $B_{32} = \text{tgemm}_{16|32}(A_{16}, \Omega_{16})$
 - 4: $Q_{32} = \text{qr}(B_{32})$
 - 5: $X_{16} = \text{fp16}(Q_{32})$
 - 6: $Y_{32} = \text{tgemm}_{16|32}(A_{16}^T, X_{16})$
 - 7: $Y_{16} = \text{fp16}(Y_{32})$
-

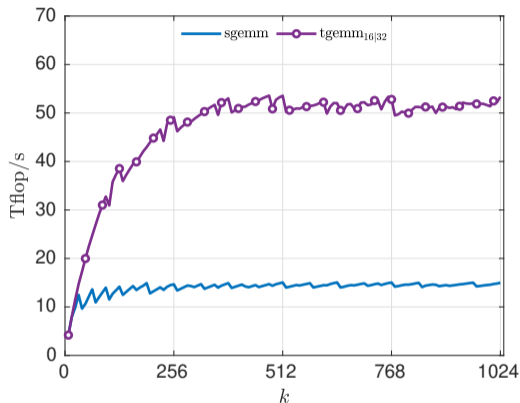
Performance of randomized LRA with tensor cores



Performance of randLRA on A100, $m = n = 35840$

- randLRA with tgemm_{16|32} is up to $\times 3$ faster than with sgemm.
- Yet the theoretical peak is 312 TFlop/s.

Performance of randomized LRA with tensor cores



Performance of randLRA on A100, $m = n = 35840$

- randLRA with tgemm_{16|32} is up to $\times 3$ faster than with sgemm.
- Yet the theoretical peak is 312 Tflop/s. \Rightarrow Why is it not attained?

randLRA on GPU with `tgemm16|32`.

Input: $A_{32} \in \mathbb{R}^{m \times n}$, the target rank k

Output: $X_{16} \in \mathbb{R}^{m \times k}$ and $Y_{16} \in \mathbb{R}^{n \times k}$ such that $A_{32} \approx X_{16} Y_{16}^T$.

- 1: $\Omega_{16} = \text{randn}(n, k)$
 - 2: $A_{16} = \text{fp16}(A_{32})$
 - 3: $B_{32} = \text{tgemm}_{16|32}(A_{16}, \Omega_{16})$
 - 4: $Q_{32} = \text{qr}(B_{32}) \leftarrow \text{Bottleneck now!}$
 - 5: $X_{16} = \text{fp16}(Q_{32})$
 - 6: $Y_{32} = \text{tgemm}_{16|32}(A_{16}^T, X_{16})$
 - 7: $Y_{16} = \text{fp16}(Y_{32})$
-

Householder QR :



Stable.



Slow, especially on GPU.

Cholesky QR :



Based on GEMM \Rightarrow very efficient on GPU.



Unstable, loss of orthogonality $\propto \kappa(B) = \kappa(A)^2$.

Cholesky QR kernel

Input: $A \in \mathbb{R}^{m \times n}$.

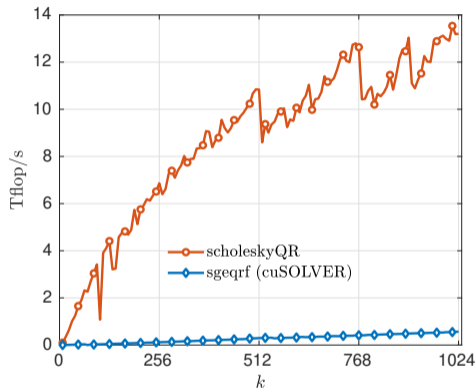
Output: Orthonormal factor $Q \in \mathbb{R}^{m \times n}$ of A .

1: $B = A^T A$

2: $R = \text{chol}(B)$

3: $Q = AR^{-1}$

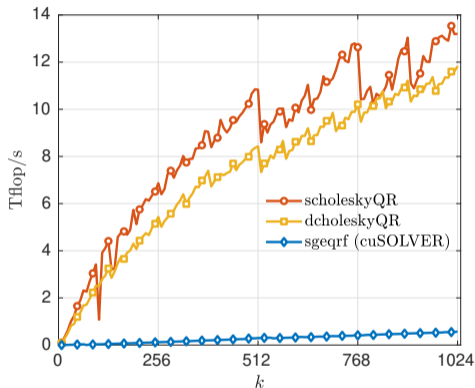
Performance of Cholesky QR



- Cholesky QR vs. Householder QR (sgeqrf):

- 👍 much faster.
- 👍 same accuracy.
- 👎 14% of breakdowns in fp32.

Performance of Cholesky QR



- Cholesky QR vs. Householder QR (sgeqrf):
 - 👍 much faster.
 - 👍 same accuracy.
 - 👎 14% of breakdowns in fp32.
- Switching from fp32 to fp64 **removes breakdowns** and maintains almost the **same performance**.
- Some variants of Cholesky QR without breakdown exist (see Lecture 11), but on A100 fp64 and fp32 achieve the same performance peak \Rightarrow using fp64 is more efficient

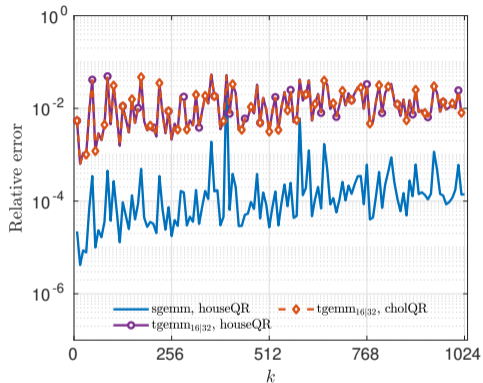
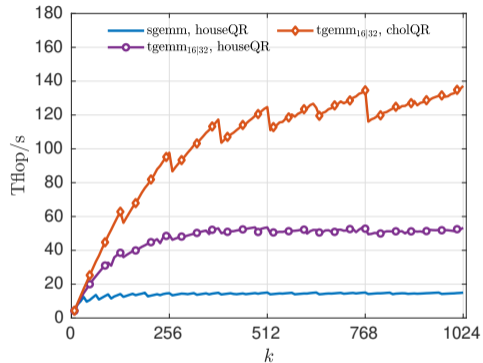
randLRA on GPU with `tgemm16|32` and Cholesky QR.

Input: $A_{32} \in \mathbb{R}^{m \times n}$, the target rank k .

Output: $X_{16} \in \mathbb{R}^{m \times k}$ and $Y_{16} \in \mathbb{R}^{n \times k}$ such that $A_{32} \approx X_{16} Y_{16}^T$.

- 1: $\Omega_{16} = \text{randn}(n, k)$
 - 2: $A_{16} = \text{fp16}(A_{32})$
 - 3: $B_{32} = \text{tgemm}_{16|32}(A_{16}, \Omega_{16})$
 - 4: $B_{64} = \text{fp64}(B_{32})$
 - 5: $C_{64} = B_{64}^T B_{64}$
 - 6: $R_{64} = \text{chol}(C_{64})$
 - 7: $Q_{64} = B_{64} R_{64}^{-1}$
 - 8: $X_{16} = \text{fp16}(Q_{64})$
 - 9: $Y_{32} = \text{tgemm}_{16|32}(A_{16}^T, X_{16})$
 - 10: $Y_{16} = \text{fp16}(Y_{32})$
-

Performance and accuracy of randomized LRA



Very fast, but only $\sim 10^{-2}$ accuracy \Rightarrow how can we improve it?

Three approaches to combine mixed precision and low-rank approximations:

- **Adaptive precision:** adapt the precision to the matrix at hand, taking advantage of the possibly rapid decay of singular values
- **Multiword arithmetic:** use multiword arithmetic to accelerate matrix products, by combining randomized methods with fast hardware such as NVIDIA tensor cores
- **Iterative refinement:** compute a low precision LRA and refine its accuracy iteratively, drawing inspiration from IR for $Ax = b$

Input: $A \in \mathbb{R}^{m \times n}$, k , p

Output: $X \in \mathbb{R}^{m \times k}$, $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$

$\Omega \leftarrow \text{randn}(n, k + p)$

Compute the MW decomp. $A \approx A_1 + A_2$.

Compute the MW decomp. $\Omega \approx \Omega_1 + \Omega_2$.

$B \leftarrow A_1 \Omega_1 + A_2 \Omega_1 + A_1 \Omega_2$

$Q \leftarrow \text{qr}(B)$

Compute the MW decomp. $Q \approx Q_1 + Q_2$.

$C \leftarrow A_1^T Q_1 + A_2^T Q_1 + A_1^T Q_2$

$ZY^T \leftarrow \text{LRA}(C, k)$

$X \leftarrow QZ$

- If the fp16/fp32 speed ratio is s , then for $m \rightarrow \infty$ this algorithm is $s/3$ faster compared with the uniform fp32 one.

- The expectation of the approximation error remains unchanged if $\Omega \sim \mathcal{N}(0, \sigma)$ as long as $\sigma \approx 1$. [📄 Ootomo and Yokota \(2023\)](#)
- Generating Ω in a t -bit arithmetic yields $\sigma \approx 1 + 2^{-t} \Rightarrow$ can store Ω in low precision and reduce the cost of the $A\Omega$ product!

Input: $A \in \mathbb{R}^{m \times n}$, k , p

Output: $X \in \mathbb{R}^{m \times k}$, $Y \in \mathbb{R}^{n \times k}$ such that $A \approx XY^T$

$\Omega_1 \leftarrow \text{randn}(n, k + p)$ in fp16.

Compute the MW decomp. $A \approx A_1 + A_2$.

$B \leftarrow A_1\Omega_1 + A_2\Omega_1 + A_1\Omega_2$

$Q \leftarrow \text{qr}(B)$

Compute the MW decomp. $Q \approx Q_1 + Q_2$.

$C \leftarrow A_1^T Q_1 + A_2^T Q_1 + A_1^T Q_2$

$ZY^T \leftarrow \text{LRA}(C, k)$

$X \leftarrow QZ$

If the fp16/fp32 speed ratio is s , then for $m \rightarrow \infty$ this algorithm is $2s/5$ faster compared with the uniform fp32 one.

Can we refine a low precision LRA into a higher precision one?

1. Apply method to input **in low precision**
2. Compute residual error **in high precision**
3. Apply method to residual error **in low precision**
4. Combine result of (1) and (3) to obtain refined result **in high precision**

Can we refine a low precision LRA into a higher precision one?

Input: a matrix A

Output: its low-rank factors $X_1 Y_1^T$

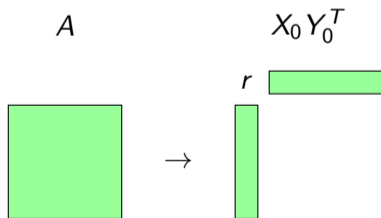
- 1: Compute LRA $X_0 Y_0^T \approx A$ in precision u_{low} .
- 2: Compute $E = A - X_0 Y_0^T$ in precision u_{high} .
- 3: Compute LRA $X_E Y_E^T \approx E$ in precision u_{low} .
- 4: $X_1 = [X_0 \ X_E]$ and $Y_1 = [Y_0 \ Y_E]$.

Can we refine a low precision LRA into a higher precision one?

Input: a matrix A

Output: its low-rank factors $X_1 Y_1^T$

- 1: Compute LRA $X_0 Y_0^T \approx A$ in precision u_{low} .
- 2: Compute $E = A - X_0 Y_0^T$ in precision u_{high} .
- 3: Compute LRA $X_E Y_E^T \approx E$ in precision u_{low} .
- 4: $X_1 = [X_0 \ X_E]$ and $Y_1 = [Y_0 \ Y_E]$.



Can we refine a low precision LRA into a higher precision one?

Input: a matrix A

Output: its low-rank factors $X_1 Y_1^T$

- 1: Compute LRA $X_0 Y_0^T \approx A$ in precision u_{low} .
- 2: Compute $E = A - X_0 Y_0^T$ in precision u_{high} .
- 3: Compute LRA $X_E Y_E^T \approx E$ in precision u_{low} .
- 4: $X_1 = [X_0 \ X_E]$ and $Y_1 = [Y_0 \ Y_E]$.

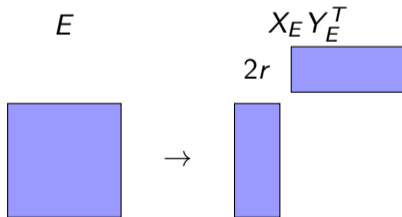
$$\begin{array}{ccccc} E & & A & & X_0 Y_0^T \\ & & & & r \text{ } \color{green}{\boxed{}} \\ \color{blue}{\boxed{}} & = & \color{green}{\boxed{}} & - & \color{green}{\boxed{}} \\ \text{rank}(E) & \leq & \text{rank}(A) & + & r \end{array}$$

Can we refine a low precision LRA into a higher precision one?

Input: a matrix A

Output: its low-rank factors $X_1 Y_1^T$

- 1: Compute LRA $X_0 Y_0^T \approx A$ in precision u_{low} .
- 2: Compute $E = A - X_0 Y_0^T$ in precision u_{high} .
- 3: **Compute LRA $X_E Y_E^T \approx E$ in precision u_{low} .**
- 4: $X_1 = [X_0 \ X_E]$ and $Y_1 = [Y_0 \ Y_E]$.

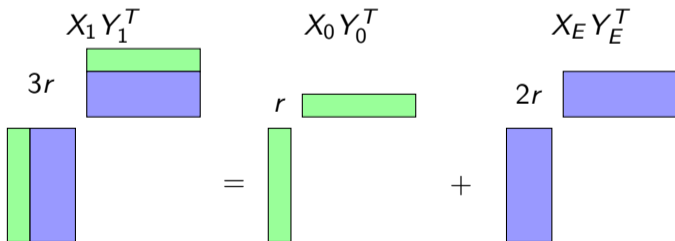


Can we refine a low precision LRA into a higher precision one?

Input: a matrix A

Output: its low-rank factors $X_1 Y_1^T$

- 1: Compute LRA $X_0 Y_0^T \approx A$ in precision u_{low} .
- 2: Compute $E = A - X_0 Y_0^T$ in precision u_{high} .
- 3: Compute LRA $X_E Y_E^T \approx E$ in precision u_{low} .
- 4: $X_1 = [X_0 \ X_E]$ and $Y_1 = [Y_0 \ Y_E]$.



Can we refine a low precision LRA into a higher precision one?

Input: a matrix A

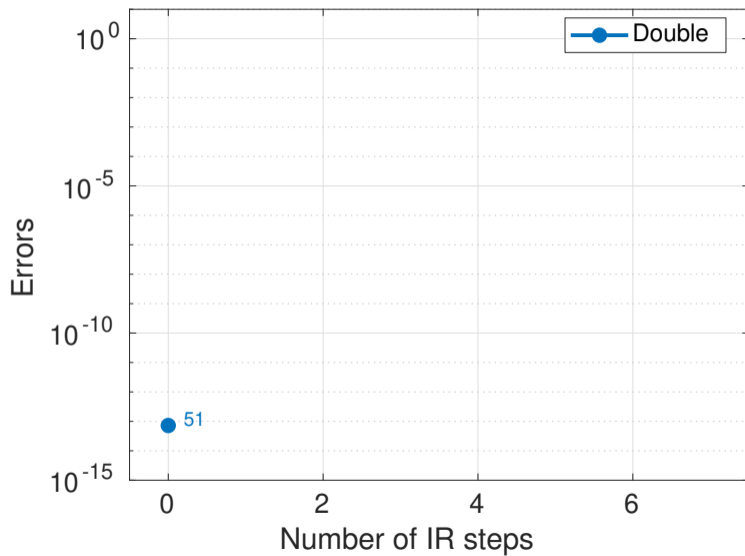
Output: its low-rank factors $X_1 Y_1^T$

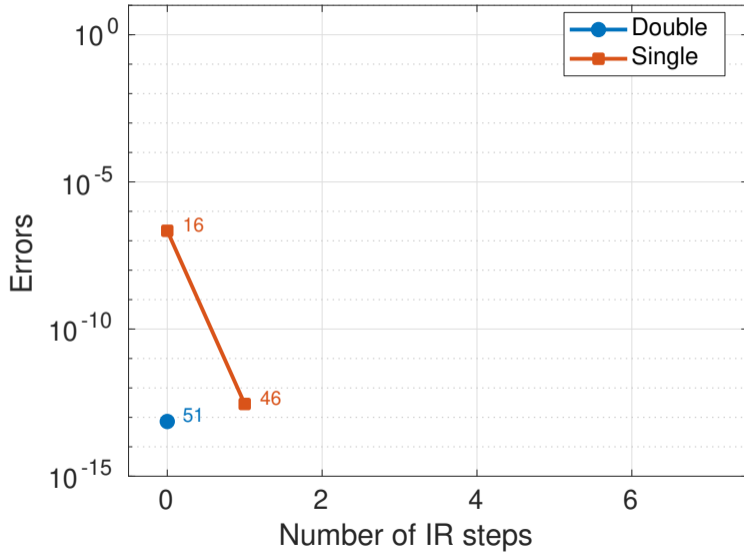
- 1: Compute LRA $X_0 Y_0^T \approx A$ in precision u_{low} .
- 2: Compute $E = A - X_0 Y_0^T$ in precision u_{high} .
- 3: Compute LRA $X_E Y_E^T \approx E$ in precision u_{low} .
- 4: $X_1 = [X_0 \ X_E]$ and $Y_1 = [Y_0 \ Y_E]$.

- Can recompress $X_1 Y_1^T$ from rank $3r$ to rank r
- Achieves u_{low}^2 accuracy with most of the work done in precision u_{low}
- Can repeat process: after i iterations, the computed $X_i Y_i^T$ satisfies

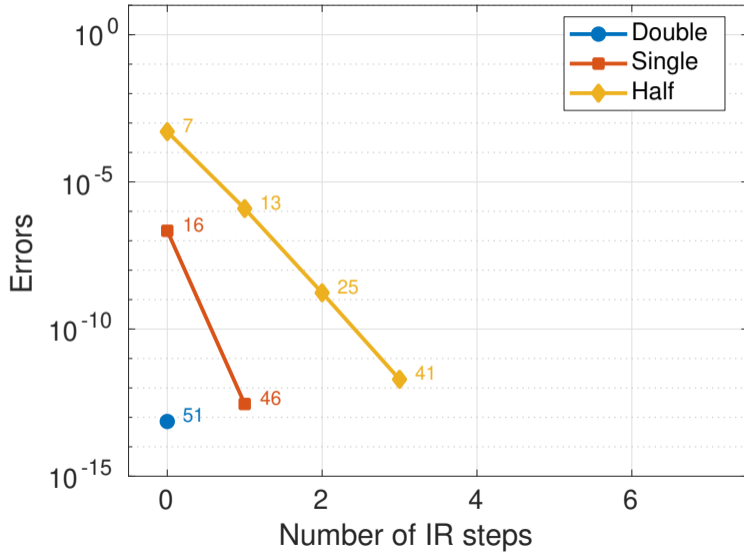
$$\|A - X_i Y_i^T\| \leq (\phi^{i+1} + \xi + O(u_{\text{low}} u_{\text{high}})) \|A\|$$

- $\phi = O(u_{\text{low}})$ is the convergence speed
- $\xi = O(u_{\text{high}})$ is the attainable accuracy

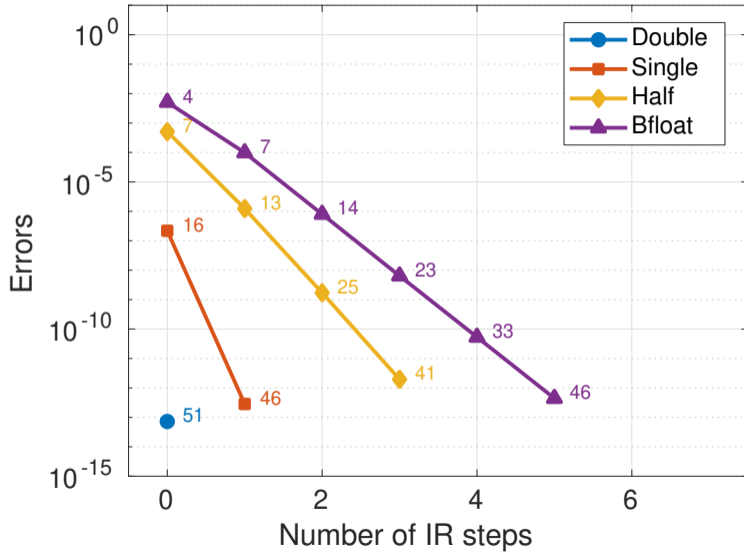




LRA-IR experiments (MATLAB)



LRA-IR experiments (MATLAB)



IR faster than standard high precision LRA in two scenarios:

- If the ranks r_i at the early iterations are much smaller than the final rank: $r_i \ll r \Rightarrow$ requires rapid decay of singular values
- If the low precision is much faster than high precision, such as with GPU tensor cores!

LRA-IR therefore bridges the gap between adaptive precision LRA and multiword LRA!

- Large singular values are computed with low precision but high accuracy, as in the multiword approach
- Small singular values are computed with low precision and low accuracy as in the adaptive precision approach

randLRA with iterative refinement.

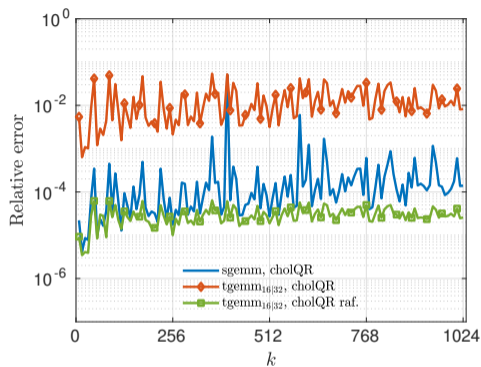
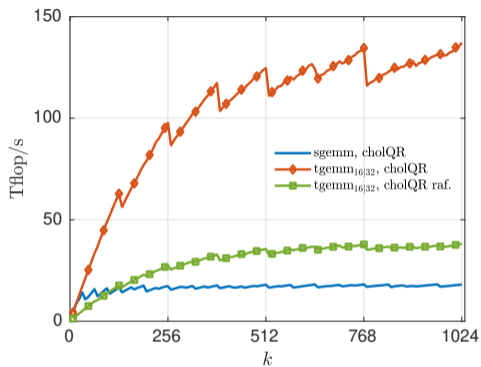
Input: $A_{32} \in \mathbb{R}^{m \times n}$, the target rank k .

Output: $X_{16} \in \mathbb{R}^{m \times 3k}$ and $Y_{16} \in \mathbb{R}^{n \times 3k}$ such that $A_{32} \approx X_{16} Y_{16}^T$.

- 1: $[X_{16}, Y_{16}] = \text{randLRA}(A_{32}, k)$
 - 2: $E_{32} = A_{32} - \text{tcgemm}_{16|32}(X_{16}, Y_{16}^T)$
 - 3: $[X'_{16}, Y'_{16}] = \text{randLRA}(E_{32}, 2k)$
 - 4: $X_{16} = [X_{16}, X'_{16}]$
 - 5: $Y_{16} = [Y_{16}, Y'_{16}]$
-

- Since input of `tcgemm` is already in fp16, can use tensor cores to compute E_{32} with fp32 accuracy!

Performance and accuracy of randLRA with refinement



- randLRA with tgemm and refinement is $\times 2.1$ faster than with SGEMM.
- Also, more accurate than with SGEMM.

[Baboulin, Donfack, Kaya, M., Robeyns \(2024\)](#)

- **Adaptive precision LRA**
 - ▲ Can be applied to a wide range of LRA methods
 - ▲ Rigorous control of the accuracy via adaptive criterion
 - ▼ Performance gains conditioned on rapid decay of singular values
- **Randomized LRA with multiword arithmetic**
 - ▲ Conceptually very simple and transparent for the user
 - ▲ Rigorous control of the accuracy via emulation
 - ▼ Restricted to randomized methods and to fast “tensor core” hardware
- **Iterative refinement for LRA**
 - ▲ Unifies both previous methods: can take advantage of *both* fast hardware and rapid decay of singular values
 - ▲ Rigorous control of the accuracy via refinement
 - ▼ Requires more flops than either of the two previous methods

Take-away: mixed precision, low-rank approximations, and randomization synergize well together!