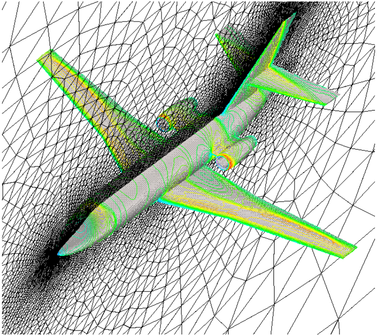
Approximate high performance computing

Theo Mary (CNRS)

HPCA course (2022 version)
Sorbonne Université



Challenges in HPC

- Increasingly large problems (10^7–10^9 unknowns)
- Increasingly parallel computers
- Heterogeneity in the computing units: CPUs, GPUs, other accelerators
- Increasing gap between speed of computations and communications
- Increasing power consumption

$\Rightarrow$ We will tackle these challenges by working with **approximations**

Sources of error in computing

1. **Model errors**

$$\frac{\partial u}{\partial t} = \Delta u$$

2. **Discretization errors**

$$\frac{u_j^{n+1} - u_j^n}{k} = \frac{u_{j+1}^{n+1} - 2u_j^{n+1} + u_{j-1}^{n+1}}{h^2}$$

$$\Rightarrow (1 + 2r)u_j^{n+1} - ru_{j-1}^{n+1} - ru_{j+1}^{n+1} = u_j^n$$

$$\Rightarrow Au^{n+1} = f(u^n)$$

3. **Rounding errors**

Floating-point numbers are represented by

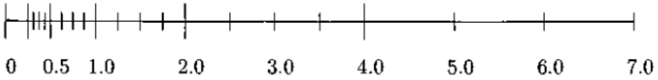
$$x = \pm m \times \beta^{e-t}, \quad m \in [0, \beta^t - 1]$$

A floating-point number system is thus charaterized by

- Base β (usually 2)
- Precision t
- Exponent range: $e \in [e_{\min}, e_{\max}]$

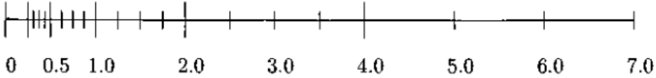
which are encoded with a finite number of bits assigned to the mantissa and exponent

$$\beta = 2, \quad t = 3, \quad [e_{\min}, e_{\max}] = [-1, 3]$$



Sources of error in computing

$$\beta = 2, \quad t = 3, \quad [e_{\min}, e_{\max}] = [-1, 3]$$



The unit roundoff $u = \beta^{1-t}/2$ ($= 2^{-t}$ in base 2) determines the relative accuracy any number in the representable range can be approximated with:

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

Moreover the **standard model of arithmetic** is

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

Double and single precision

	Number of bits		Range	Unit roundoff u
	Mantissa	Exponent		
fp64	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	24	8	$10^{\pm 38}$	6×10^{-8}

Double (fp64) and single (fp32) precision both widely supported in hardware

10^{-16} is tiny! Are rounding errors really significant?

Impact of rounding errors

Consider the computation

$$s = \sum_{i=1}^{n} x_i$$

In floating-point arithmetic, each addition produces a rounding error. The overall error E is bounded by

$$|E| \le n\kappa u, \qquad \kappa = \frac{\sum |x_i|}{|\sum x_i|}$$

E can be large when

- The unit roundoff u is large **(low precision)**
- The dimension n is large **(error accumulation)**
- The condition number κ is large **(error amplification)**

Historical perspective

- Backward error analysis was developed by James Wilkison in the 1960s
- At that time, $n = 100$ was huge! Solving linear systems of $n = O(10)$ equations would take days

$\Rightarrow$ n was considered a "constant"

*The **constant** terms in an error bound are the least important parts of error analysis. It is not worth spending much effort to minimize constants because the achievable improvements are usually insignificant.*

Nick Higham, ASNA 2ed (2002)

Hence traditional error analysis has focused on error amplification

Today: large problems and low precisions

Problems are getting larger and larger (ex: 21 million equations solved in 4 hours for the latest TOP500 ranking)

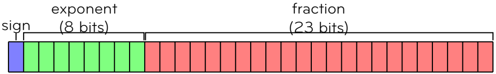
and

Precisions are getting lower and lower

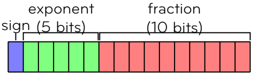
	Number of bits		Range	Unit roundoff u
	Mantissa	Exponent		
fp64	53	11	$10^{\pm 308}$	1×10^{-16}
fp32	24	8	$10^{\pm 38}$	6×10^{-8}
tfloat32	11	8	$10^{\pm 38}$	5×10^{-4}
fp16	11	5	$10^{\pm 5}$	5×10^{-4}
bfloat16	8	8	$10^{\pm 38}$	4×10^{-3}
fp8 (e4m3)	4	4	$10^{\pm 2}$	6×10^{-2}
fp8 (e5m2)	3	5	$10^{\pm 5}$	1×10^{-1}

- Half (16-bit) and quarter (8-bit) precision now in hardware, driven by AI

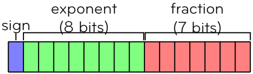
Lower precisions



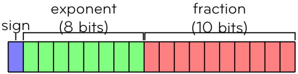
fp32
Range $10^{\pm 38}$, $u = 6 \times 10^{-8}$



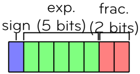
fp16
Range $10^{\pm 5}$, $u = 5 \times 10^{-4}$



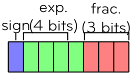
bfloat16
Range $10^{\pm 38}$, $u = 4 \times 10^{-3}$



tfloat32
Range $10^{\pm 38}$, $u = 5 \times 10^{-4}$



fp8 (e5m2)
Range $10^{\pm 5}$, $u = 1 \times 10^{-1}$



fp8 (e4m3)
Range $10^{\pm 2}$, $u = 6 \times 10^{-2}$

(Deliberately) approximate computing

Conclusion: today's computing is already approximate!

Since errors are part of HPC, let's embrace them

4. **Approximation errors**
    - Rounding errors from use of low precision arithmetic
    - Compression/sparsification errors
    - Errors from unstable algorithms
    - ...

5. **Silent errors** (bitflips)

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_0 = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_1 = \begin{pmatrix} 1 \\ 0.5 \\ 0 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_2 = \begin{pmatrix} 1.3 \\ -0.5 \\ -0.125 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_3 = \begin{pmatrix} 1.3 \\ 0.187 \\ 0.162 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_4 = \begin{pmatrix} 1.1125 \\ 0.09375 \\ -0.1625 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge: solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_5 = \begin{pmatrix} 1.05625 \\ 0.09375 \\ -0.1390625 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_6 = \begin{pmatrix} 1.05625 \\ 0.1523438 \\ -0.1320313 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_7 = \begin{pmatrix} 1.0914062 \\ 0.1699219 \\ -0.1320312 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_8 = \begin{pmatrix} 1.0914062 \\ 0.1699219 \\ -0.1320312 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_9 = \begin{pmatrix} 1.1019531 \\ 0.1589355 \\ -0.1377441 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_{10} = \begin{pmatrix} 1.0953613 \\ 0.1556396 \\ -0.1377441 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_{11} = \begin{pmatrix} 1.0933838 \\ 0.1556396 \\ -0.1369202 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:

solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_{12} = \begin{pmatrix} 1.0933838 \\ 0.1576996 \\ -0.1366730 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge: solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1}y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$$x_{13} = \begin{pmatrix} 1.0946198 \\ 0.1583176 \\ -0.1366730 \end{pmatrix}$$

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge: solve $M^{-1}Ax = M^{-1}b$

Two classes of methods to solve $Ax = b$

Direct methods

Gaussian elimination, based on LU factorization.

$$A = \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 1 & 0 & 8 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0.2 & 0.3 & 1 \end{pmatrix} \times \begin{pmatrix} 5 & -3 & 0 \\ 0 & 2 & -5 \\ 0 & 0 & 9.5 \end{pmatrix} = LU$$

$L(Ux) = b$ solved in two steps:

(i) $y = L^{-1} \begin{pmatrix} 5 \\ 1 \\ 0 \end{pmatrix} = \begin{pmatrix} 5 \\ 1 \\ -1.3 \end{pmatrix}$ then (ii) $x = U^{-1} y = \begin{pmatrix} 1.0947368 \\ 0.1578947 \\ -0.1368421 \end{pmatrix}$

Iterative methods

Build a sequence of iterates x_0, ..., x_k until $\|Ax_k - b\|$ small enough

Example (Jacobi): x_0, $x_{k+1} = x_k + D^{-1} \times (b - Ax_k)$, $D = diag(A)$

$x_{14} = \begin{pmatrix} 1.0946198 \\ 0.1583176 \\ -0.1366730 \end{pmatrix}$ $\quad \|Ax_{14} - b\| \approx 8.5 \times 10^{-4}$ $\quad$ ($\approx 10^{-15}$ with LU)

In practice, require a preconditioner $M^{-1} \approx A^{-1}$ to converge:
solve $M^{-1}Ax = M^{-1}b$

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

- **Iterative methods**
  - Low time and memory per-iteration cost
  - Convergence is application dependent
  - ⇒ Need good preconditioner

⇒ **Approximate factorizations...**
  - as approximate fast direct methods, if
    - low accuracy is sufficient, or
    - matrix is structured (data sparsity)
  - as high quality preconditioners otherwise

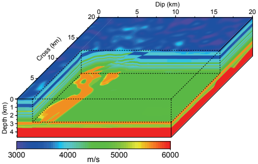
LU factorization

$A = LU \Leftrightarrow \forall i,j \quad a_{ij} = \sum_{k=1}^{\min(i,j)} \ell_{ik} u_{kj}$

```
for k = 1: n do
    u_kk = a_kk (ℓ_kk = 1)
    for i = k + 1: n do
        ℓ_ik = a_ik/u_kk and u_ki = a_ki
    end for
    for i = k + 1: n/b do
        for j = k + 1: n/b do
            a_ij ← a_ij − ℓ_ik u_kj
        end for
    end for
end for
```

$2n^3/3$ flops for unsymmetric A

Seismic imaging in geophysics



(3D EAGE/SEG overthrust model)

(credits: SEISCOPE project)

$\Longrightarrow$

Frequency domain FWI (Full-Wave Inversion)

Helmholtz equations

Complex Unsym. sparse matrix **A**

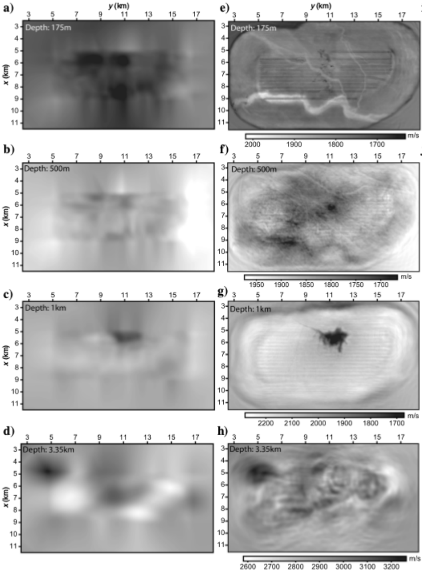
Multiple (very) sparse **B**

Required accuracy $< 10^{-4}$

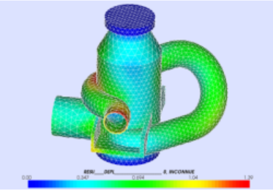
freq	flops LU	Factor Storage	Peak memory
2 Hz	9.0E+11	3 GB	4 GB
4 Hz	1.6E+13	22 GB	25 GB
8 Hz	5.8E+14	247 GB	283 GB
10 Hz	2.7E+15	728 GB	984 GB

Higher frequency leads to refined model

Seismic imaging in geophysics



Pump from nuclear reactor



A RIS pump (circuit d'injection de sécurité) under internal pressure

Real sym. **indefinite** sparse matrix **A**

One dense right-hand side **b**

Required accuracy $> 10^{-9}$

n	nnz	flops LU	LU Storage
5.4E+6	2.1E+8	1.8E+13	56 GB

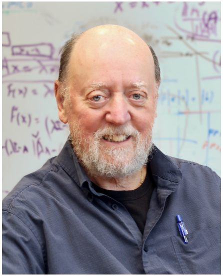
Number of delayed pivots = 79k

TOP500 ranking

Since the 1990s, the TOP500 list ranks the world's most powerful supercomputers based on how fast they can solve a dense linear system of equations $Ax = b$

Rank	System	Cores	Rmax (PFlop/s)	Rpeak (PFlop/s)	Power (kW)
1	**Frontier** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE DOE/SC/Oak Ridge National Laboratory United States	8,730,112	1,102.00	1,685.65	21,100
2	**Supercomputer Fugaku** - Supercomputer Fugaku, A64FX 48C 2.2GHz, Tofu interconnect D, Fujitsu RIKEN Center for Computational Science Japan	7,630,848	442.01	537.21	29,899
3	**LUMI** - HPE Cray EX235a, AMD Optimized 3rd Generation EPYC 64C 2GHz, AMD Instinct MI250X, Slingshot-11, HPE EuroHPC/CSC Finland	1,110,144	151.90	214.35	2,942

June 2022:
Frontier achieves
1.1 ExaFLOPS



Jack Dongarra
(Turing Award 2021)

Benefits of lowering the precision

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

- Speed of computations also generally proportional
  On most computers, fp32 is twice faster than fp64 (💬 why?)
  lower precision ⇒ faster computations

- Power consumption is proportional to the square of the number of mantissa bits. Thus:
  - fp16 and tfloat32 (11 bits) consume $5\times$ less energy than fp32 (24 bits)
  - bfloat16 (8 bits) consumes $2\times$ less energy than fp16/tfloat32 and $9\times$ less than fp32!

  lower precision ⇒ greener computations

CELL processor

A notable exception: CELL processor (2006–2008)
1 CELL = 1 PPE + $8\times$ SPE
PPE peak (GFLOPS): 6.4 (fp64) $\rightarrow$ 25.6 (fp32) $4\times$ speedup!
SPE peak (GFLOPS): 1.8 (fp64) $\rightarrow$ 25.6 (fp32) $14\times$ speedup!!

Paper from 2008: Kurzak et al (2008)

The PlayStation 3 for High-Performance Scientific Computing

Publisher: IEEE

Cite This

PDF

Jakub Kurzak; Alfredo Buttari; Piotr Luszczek; Jack Dongarra All Authors

Paper from 2008: Kurzak et al (2008)

The PlayStation 3 for High-Performance Scientific Computing

Publisher: IEEE Cite This PDF

Jakub Kurzak; Alfredo Buttari; Piotr Luszczek; Jack Dongarra **All Authors**

Condor Cluster (peak: 500 TFLOPS)
Made of 1760 PS3s !

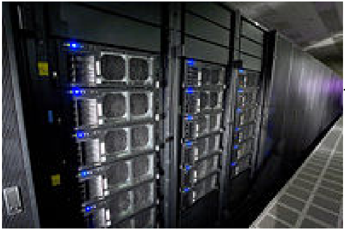
Paper from 2008: Kurzak et al (2008)

The PlayStation 3 for High-Performance Scientific Computing

Publisher: IEEE Cite This PDF

Jakub Kurzak; Alfredo Buttari; Piotr Luszczek; Jack Dongarra **All Authors**

Condor Cluster (peak: 500 TFLOPS)
Made of 1760 PS3s !



IBM Roadrunner (peak: 1.7 PFLOPS)
1st on TOP500 ranking in 2008
First computer to surpass 1 PFLOP on LINPACK benchmark!

NVIDIA GPUs

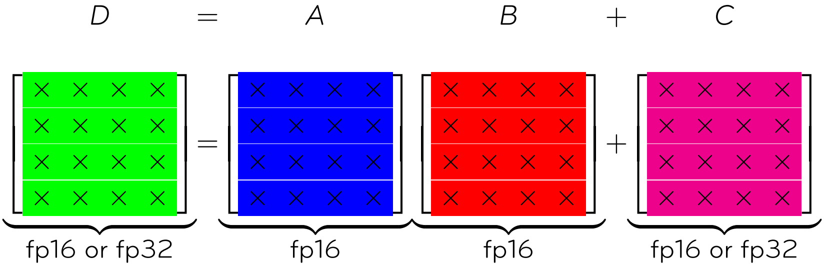
The exception is becoming the rule for half precision on modern hardware
NVIDIA Tesla GPUs

Peak performance (TFLOPS)

		fp64	fp32	tfloat32	fp16	bfloat16	fp8
Pascal	2016	5	9	–	19	–	–
Volta	2018	8	16	–	125	–	–
Ampere	2020	10	19	156	312	312	–
Hopper	2022	30	60	500	1000	1000	2000

NVIDIA GPU tensor cores

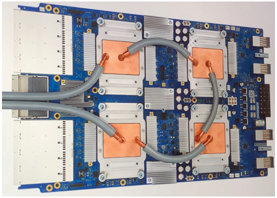
Tensor cores units available on NVIDIA GPUs V100 carry out a 4×4 matrix multiplication **in 1 clock cycle**:



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.

On A100, support for bfloat16 and tfloat32 was added

Other similar units



MXUs (matrix units) from Google TPUs (Tensor Processing Units) carry out a MAC (Multiply and Accumulate) on 256×256 or 128×128 matrices using bfloat16

ARMv8-A CPUs have vector instructions for bfloat16 with fp32 accumulation

Intel Cooper Lake CPUs have scalar FMAs with bfloat16 input and fp32 ouptut

Block FMA: a general framework

We consider the following framework Blanchard et al. (2020)

- $A \in \mathbb{R}^{b_1 \times b}$, $B \in \mathbb{R}^{b \times b_2}$, and $C \in \mathbb{R}^{b_1 \times b_2}$,

$$\underbrace{D}_{u_{\text{low}} \text{ or } u_{\text{high}}} = \underbrace{C}_{u_{\text{low}} \text{ or } u_{\text{high}}} + \underbrace{A}_{u_{\text{low}}} \underbrace{B}_{u_{\text{low}}}$$

- AB is computed internally in precision u_{high}

$$|\widehat{D} - D| \lesssim (b+1) u_{\text{high}} (|C| + |A||B|)$$

- Why "block FMA" ?
  - If $u_{\text{high}} = 0$, generalizes FMA to block of entries
  - With $u_{\text{high}} \neq 0$, it ressembles an FMA (no internal error to order $O(u_{\text{low}})$)

Examples of block FMA units

Year	Device	b_1	b	b_2	u_{low}	u_{high}
2016	Google TPU v2	128	128	128	bfloat16	fp32
2017	Google TPU v3	128	128	128	bfloat16	fp32
2020	Google TPU v4i	128	128	128	bfloat16	fp32
2017	NVIDIA V100	4	4	4	fp16	fp32
2018	NVIDIA T4	4	4	4	fp16	fp32
2020	NVIDIA A100	8	8	4	fp16	fp32
2020	NVIDIA A100	8	8	4	bfloat16	fp32
2020	NVIDIA A100	8	4	4	tfloat32	fp32
2020	NVIDIA A100	2	4	2	fp64	fp64
2019	ARMv8.6-A	2	4	2	bfloat16	fp32
2020	Intel Cooper Lake	1	1	1	bfloat16	fp32

More to come!

Matrix multiplication with block FMA

This algorithm computes $C = AB$ using a block FMA, 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 a block FMA
        end for
    end for
end for

Matrix multiplication: error analysis

First, we convert A and B to low precision:

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

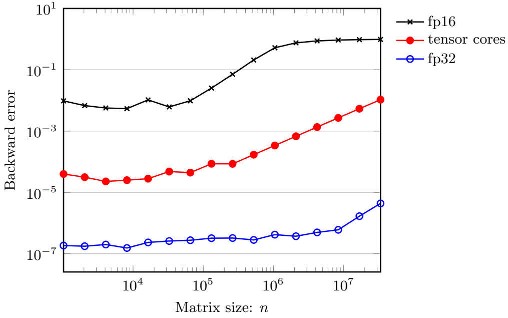
Second, we compute the product:

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

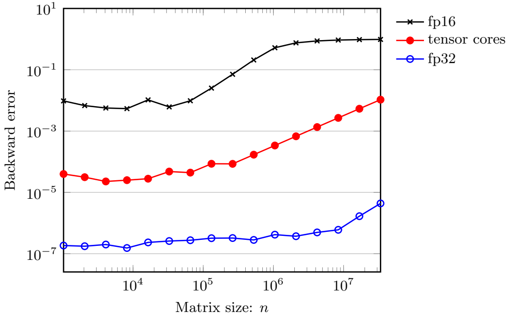
Evaluation method	Bound
Standard in precision u_{low}	$n u_{\mathrm{low}}$
Standard in precision u_{high}	$n u_{\mathrm{high}}$
Block FMA	$2u_{\mathrm{low}} + n u_{\mathrm{high}}$

$\Rightarrow$ 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, and exhibit the following special behaviors (not documented by NVIDIA!) Fasi et al. (2021)

- The additions in the product AB are performed with the **round-towards-zero** rounding mode.
- The order of the additions is variable and starts with the largest element.
- Tensor cores are non-monotonic!

Solving $Ax = b$

Standard method to solve $Ax = b$:

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

Solving $Ax = b$

Standard method to solve $Ax = b$:

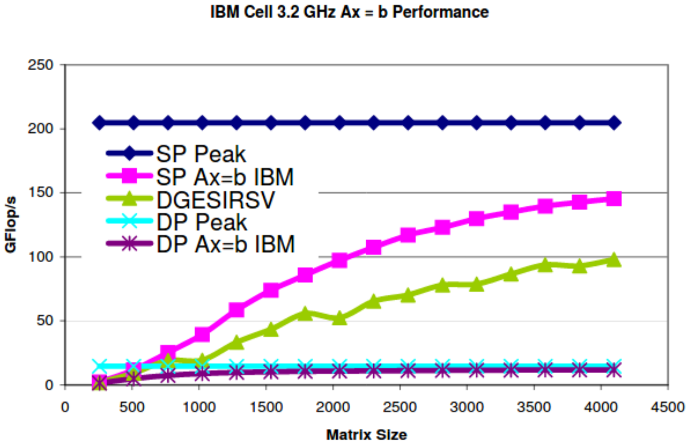
1. Factorize $A = LU$, where L and U are lower and upper triangular
2. Solve $Ly = b$ and $Ux = y$

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

```
Solve Ax_1 = b as above at precision u_low
for i = 1: nsteps do
    r_i = b − Ax_i at precision u_high
    Solve Ad_i = r_i via d_i = U^{-1}(L^{-1} r_i) at precision u_low
    x_{i+1} = x_i + d_i at precision u_high
end for
```

- Most of the flops in precision u_{low} (only $O(n^2)$ in precision u_{high})
- Convergence to u_{high} accuracy guaranteed as long as $\kappa(A)u_{\text{low}} < 1$

IR with fp32 LU (CELL processor)



Langou et al (2006)

LU factorization (Gaussian elimination)

- Objective: given $A \in \mathbb{R}^{n\times n}$, compute lower and upper triangular matrices L and U such that $A = LU$
- $\forall i, j \quad a_{ij} = \sum_{k=1}^{\min(i,j)} \ell_{ik} u_{kj}$

```
for k = 1: n do
    u_kk = a_kk (ℓ_kk = 1)
    for i = k + 1: n do
        ℓ_ik = a_ik / u_kk and u_ki = a_ki
    end for
    for i = k + 1: n/b do
        for j = k + 1: n/b do
            a_ij ← a_ij − ℓ_ik u_kj
        end for
    end for
end for
```

- $2n^3/3$ flops

Block LU factorization

- Block version to use matrix–matrix operations

for $k = 1\colon n/b$ **do**
    Factorize $L_{kk}U_{kk} = A_{kk}$ (with unblocked alg.)
    for $i = k+1\colon n/b$ **do**
        Solve $L_{ik}U_{kk} = A_{ik}$ and $L_{kk}U_{ki} = A_{ki}$ for L_{ik} and U_{ki}
    end for
    for $i = k+1\colon n/b$ **do**
        for $j = k+1\colon n/b$ **do**

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

        end for
    end for
end for

Block LU factorization with block FMA

- Block version to use matrix–matrix operations
- With a block FMA: $A \in \mathbb{R}^{n\times n}$ is given in precision u_{high}, and L and U are returned in precision u_{FMA}

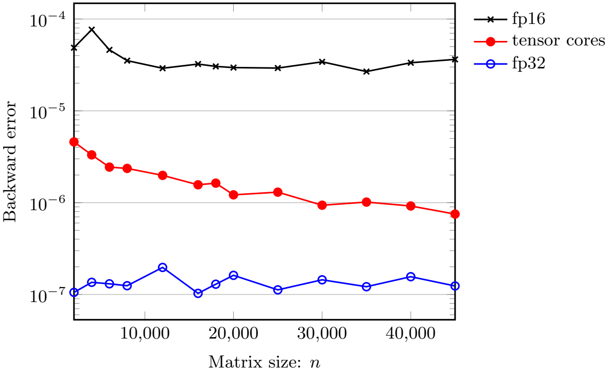
```
for k = 1: n/b do
    Factorize L_kk U_kk = A_kk    (with unblocked alg.)
    for i = k+1: n/b do
        Solve L_ik U_kk = A_ik and L_kk U_ki = A_ki for L_ik and U_ki
    end for
    for i = k+1: n/b do
        for j = k+1: n/b do
            L̃_ik ← fl_low(L_ik) and Ũ_ki ← fl_low(U_ki)
            A_ij ← A_ij − L̃_ik Ũ_kj using a block FMA
        end for
    end for
end for
```

- $O(n^3)$ part of the flops done with block FMA

LU factorization with tensor cores

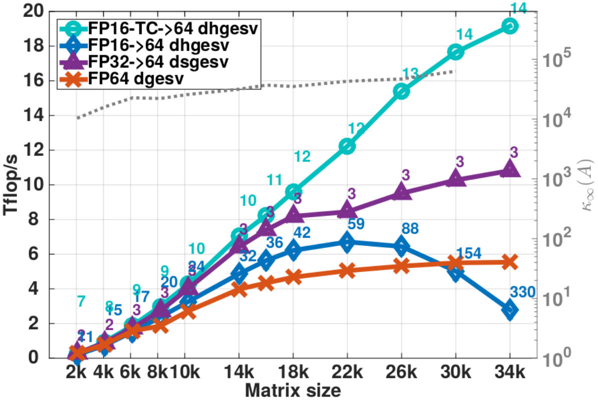
Error analysis for LU follows from matrix multiplication analysis and gives same bounds to first order Blanchard et al. (2020)

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



Impact on iterative refinement

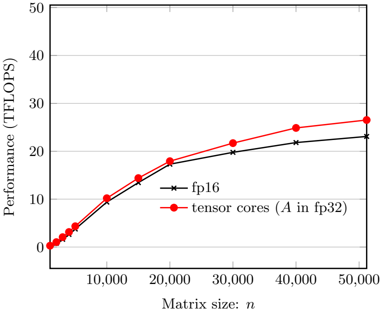
Results from Haidar et al. (2018)

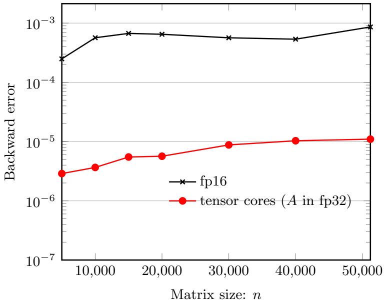


- TC accuracy boost can be critical!

- LU factorization is traditionally a compute-bound operation...
- With Tensor Cores, flops are $8\times$ faster
- Matrix is stored in fp32 $\Rightarrow$ **data movement is unchanged !**

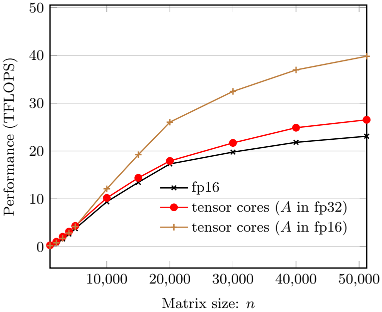
$\Rightarrow$ LU with tensor cores becomes memory-bound !

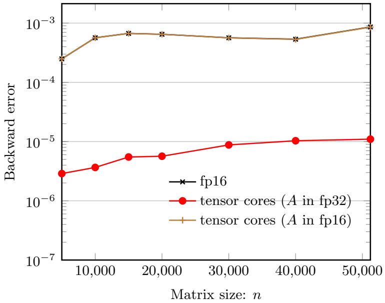




- LU factorization is traditionally a compute-bound operation...
- With Tensor Cores, flops are $8\times$ faster
- Matrix is stored in fp32 $\Rightarrow$ **data movement is unchanged !**

$\Rightarrow$ 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

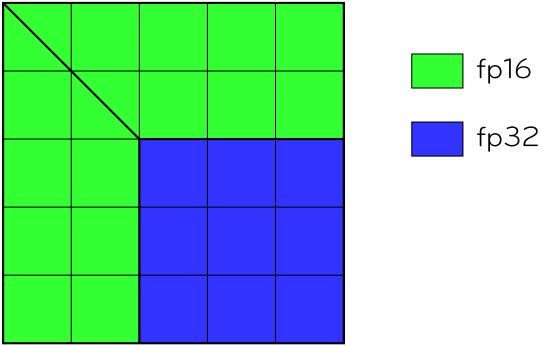
Two ingredients to reduce data movement with no accuracy loss:

Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

1. Mixed fp16/fp32 representation

Matrix after 2 steps:

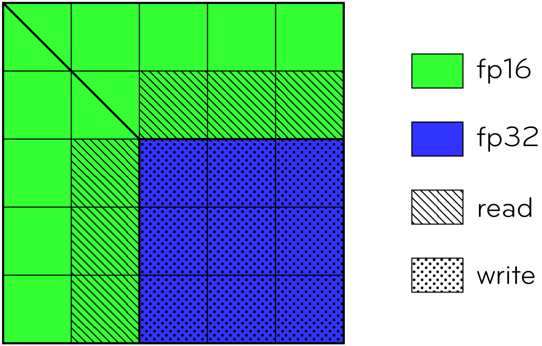


Reducing data movement

Two ingredients to reduce data movement with no accuracy loss:

1. Mixed fp16/fp32 representation

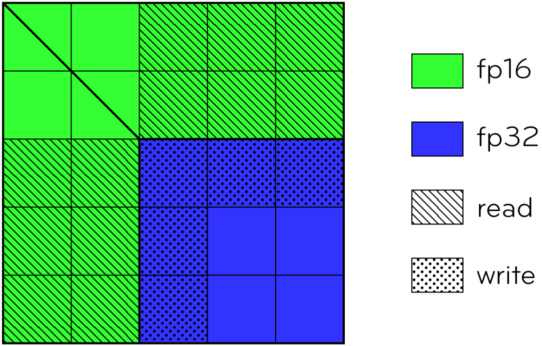
Matrix after 2 steps:



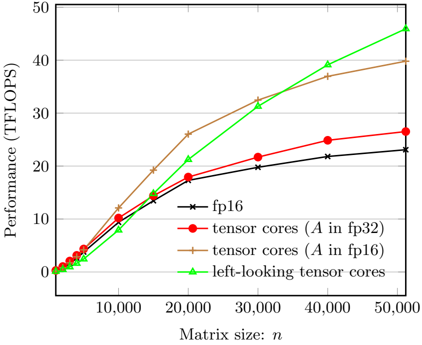
Two ingredients to reduce data movement with no accuracy loss:

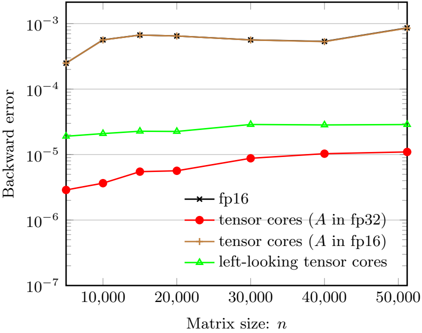
1. Mixed fp16/fp32 representation
2. Right-looking $\rightarrow$ left-looking factorization

Matrix after 2 steps:



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





Nearly **50 TFLOPS** without significantly impacting accuracy

Lopez and M. (2020)

The HPL-AI benchmark

- Standard TOP500 ranking based on the LINPACK benchmark: solve $Ax = b$, with A a dense unsym. matrix with no limitation of size, by using LU factorization, with all computations in 64-bit arithmetic
- The HPL-AI benchmark: solve $Ax = b$ to 64-bit accuracy, but use of lower precisions in intermediate computations is allowed

  The HPL-AI benchmark seeks to highlight the emerging convergence of high-performance computing (HPC) and artificial intelligence (AI) workloads

- Most implementations rely on fp16 or bfloat16 LU followed by IR in fp64

The HPL-AI benchmark

June 2022

Rank	Site	Computer	Cores	HPL-AI (Eflop/s)	TOP500 Rank	HPL Rmax (Eflop/s)	Speedup
1	DOE/SC/ORNL, USA	Frontier	8,730,112	6.861	1	1.102	6.2
2	RIKEN, Japan	Fugaku	7,630,848	2.000	2	0.4420	4.5
3	DOE/SC/ORNL, USA	Summit	2,414,592	1.411	4	0.1486	9.5
4	NVIDIA, USA	Selene	555,520	0.630	8	0.0630	9.9
5	DOE/SC/LBNL, USA	Perlmutter	761,856	0.590	7	0.0709	8.3
6	FZJ, Germany	JUWELS BM	449,280	0.470	11	0.0440	10.0
7	University of Florida, USA	HiPerGator	138,880	0.170	34	0.0170	9.9
8	SberCloud, Russia	Christofari Neo	98,208	0.123	47	0.0120	10.3
9	DOE/SC/ANL, USA	Polaris	259,840	0.114	14	0.0238	4.8
10	ITC, Japan	Wisteria	368,640	0.100	20	0.0220	4.5

The HPL-AI benchmark

June 2020

Rank	Site	Computer	Cores	HPL-AI (Eflop/s)	TOP500 Rank	HPL Rmax (Eflop/s)	Speedup
1	RIKEN, Japan	Fugaku	7,299,072	1.42	1	0.416	3.42

With mixed precision, exascale was reached in 2020!

IR for industrial problems

Matrix	time (s)		memory (GB)	
	fp64	fp32→fp64	fp64	fp32→fp64
ElectroPhys10M	265.2	**154.0**	272.0	**138.0**
Bump_2911	205.4	**129.3**	135.7	**68.4**
DrivAer6M	91.8	**67.6**	81.6	**41.7**
Queen_4147	284.2	**165.2**	178.0	**89.8**
tminlet3M	294.5	**136.2**	241.1	**121.0**
perf009ar	**46.1**	57.5	55.6	**28.9**
elasticity-3d	**156.7**	–	**153.0**	–
lfm_aug5M	536.2	**254.5**	312.0	**157.0**
Long_Coup_dt0	67.2	**46.6**	52.9	**26.7**
CarBody25M	**62.9**	–	**77.6**	–
thmgaz	97.6	**65.4**	192.0	**97.7**

- Up to $2\times$ time and memory reduction
- Convergence can be slow or impossible for ill-conditioned problems

Sparse matrices

$$\begin{pmatrix} 4 & 0 & 0 & 0 \\ 0 & 7 & 0 & 3 \\ -2 & 0 & 0 & 0 \\ 0 & 0 & 5 & 0 \end{pmatrix} \Rightarrow \quad ROW\_IND \begin{bmatrix} 1 \\ 2 \\ 4 \\ 5 \end{bmatrix} \quad COL\_IND \begin{bmatrix} 1 \\ 2 \\ 4 \\ 1 \\ 3 \end{bmatrix} \quad VAL \begin{bmatrix} 4 \\ 7 \\ 3 \\ -2 \\ 5 \end{bmatrix}$$

Gaussian elimination: $a_{ij} \leftarrow a_{ij} - a_{ik}a_{kj}$

$\Rightarrow$ a_{ij} becomes nonzero if a_{ik} and a_{kj} are nonzero: fill-in

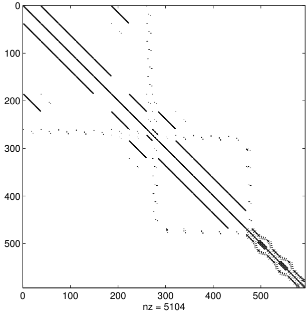
Interest of permuting a matrix:

$$\begin{pmatrix} X & X & X & X & X \\ X & X & 0 & 0 & 0 \\ X & 0 & X & 0 & 0 \\ X & 0 & 0 & X & 0 \\ X & 0 & 0 & 0 & X \end{pmatrix} \quad 1 \leftrightarrow 5 \quad \begin{pmatrix} X & 0 & 0 & 0 & X \\ 0 & X & 0 & 0 & X \\ 0 & 0 & X & 0 & X \\ 0 & 0 & 0 & X & X \\ X & X & X & X & X \end{pmatrix}$$

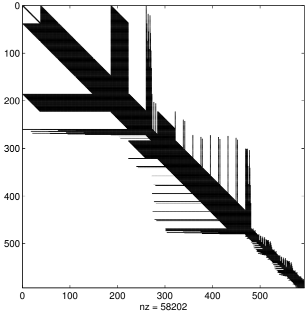
Sparse matrices

Example: dwt_592.rua, structural computing on a submarine.

Original matrix



Factorized matrix



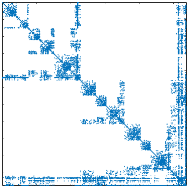
Computational savings from sparsity heavily dependent on matrix structure and permutation.

However, for regular 3D problems (e.g. PDE discretized on a cube): Flops: $O(n^3) \to O(n^2)$, Storage: $O(n^2) \to O(n^{4/3})$

Dropping approximations (sparsification)

Dropping: replace with zero any value sufficiently small

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

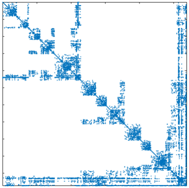


sparse A

Dropping approximations (sparsification)

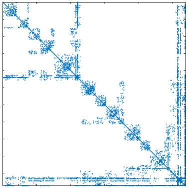
Dropping: replace with zero any value sufficiently small

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



sparse A

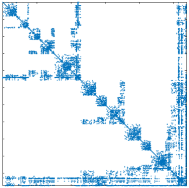
$$\xrightarrow{\textit{drop}}$$



sparser A

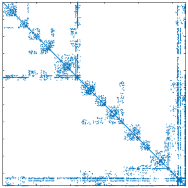
Dropping: replace with zero any value sufficiently small

$$\begin{cases} |\ell_{ij} u_{jj}| \leq \epsilon \|A\| & \Rightarrow \quad \ell_{ij} \leftarrow 0 \\ |u_{ij}| \leq \epsilon \|A\| & \Rightarrow \quad u_{ij} \leftarrow 0 \end{cases}$$

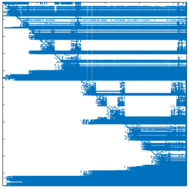


sparse A

drop $\longrightarrow$

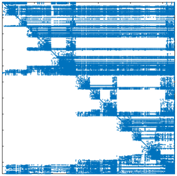


sparser A



LU factors

drop $\longrightarrow$



incomplete LU

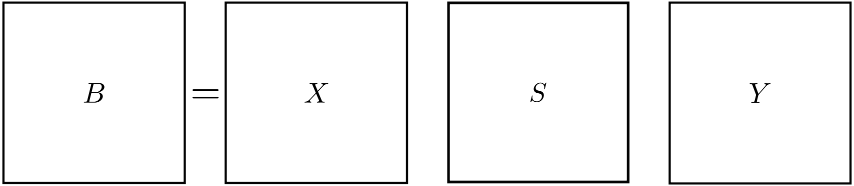
Incomplete LU factorization

for $k = 1\colon n$ **do**
  for $j = k\colon n$ **do**
    $u_{kj} = a_{kj} - \sum_{i=1}^{k-1} \ell_{ki} u_{ij}$
    if $|u_{kj}| \leq \epsilon \|A\|$ **then** $u_{kj} = 0$
  end for
  for $i = k+1\colon n$ **do**
    $\ell_{ik} = (a_{ik} - \sum_{j=1}^{k-1} \ell_{ij} u_{jk}) / u_{kk}$
    if $|\ell_{ik} u_{kk}| \leq \epsilon \|A\|$ **then** $\ell_{ik} = 0$
  end for
end for

- Incomplete factorization: drop entries $< \varepsilon$ from LU factors
- Alternatively, do not update $a_{ij} \leftarrow a_{ij} - a_{ik} a_{kj}$ if a_{ij} is zero (i.e., enforce same sparsity pattern for LU as for A)

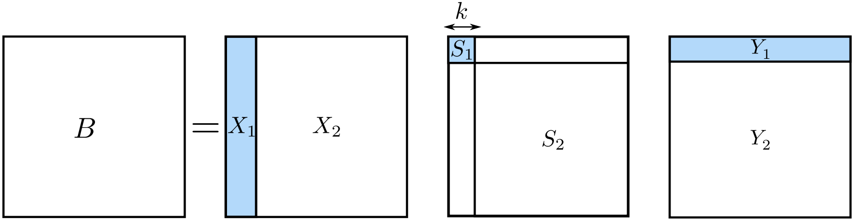
Data sparsity

Take a dense matrix B of size $b \times b$. Compute its SVD $B = XSY$:



Data sparsity

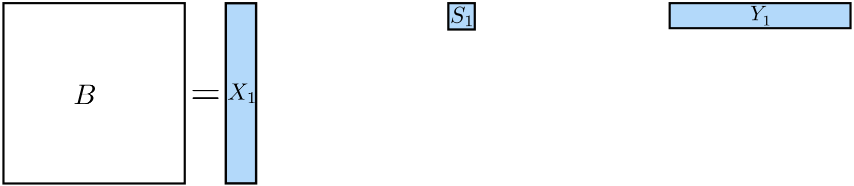
Take a dense matrix B of size $b \times b$. Compute its SVD $B = XSY$:



$k = \min\{k \leq b; \sigma_{k+1} \leq \varepsilon\}$ is the numerical rank at accuracy ε

Data sparsity

Take a dense matrix B of size $b \times b$. Compute its SVD $B = XSY$:

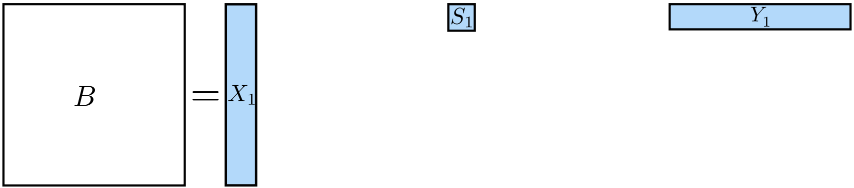


$k = \min \{k \leq b; \sigma_{k+1} \leq \varepsilon\}$ is the numerical rank at accuracy ε

$\tilde{B} = X_1 S_1 Y_1$ is a low-rank approximation to B: $\|B - \tilde{B}\|_2 \leq \varepsilon$

Data sparsity

Take a dense matrix B of size $b \times b$. Compute its SVD $B = XSY$:



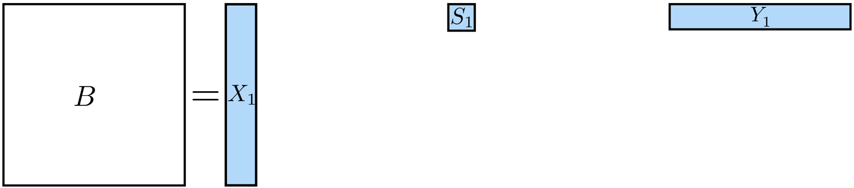
$k = \min\{k \leq b; \sigma_{k+1} \leq \varepsilon\}$ is the numerical rank at accuracy ε

$\tilde{B} = X_1 S_1 Y_1$ is a low-rank approximation to B: $\|B - \tilde{B}\|_2 \leq \varepsilon$

Storage savings: $b^2/2bk = b/2k$
Similar flops savings when used in most linear algebra kernels

Data sparsity

Take a dense matrix B of size $b \times b$. Compute its SVD $B = XSY$:



$k = \min\{k \leq b; \sigma_{k+1} \leq \varepsilon\}$ is the numerical rank at accuracy ε

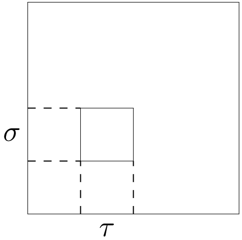
$\tilde{B} = X_1 S_1 Y_1$ is a low-rank approximation to B: $\|B - \tilde{B}\|_2 \leq \varepsilon$

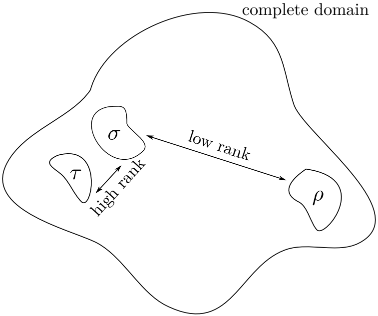
Storage savings: $b^2/2bk = b/2k$
Similar flops savings when used in most linear algebra kernels

In practice SVD is too expensive $\Rightarrow$ use other methods, e.g., randomized, comm avoiding

Data sparse matrices

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

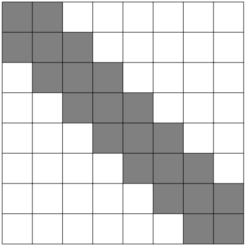




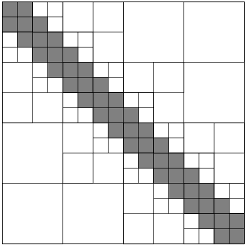
A block B represents the interaction
between two subdomains σ and τ.
Large distance $\Leftrightarrow$ low numerical rank,
even for small ε!

Data sparse matrices

Many different block partitionings possible



BLR matrix

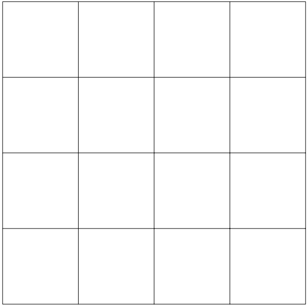


$\mathcal{H}$-matrix

- Simple, flat structure
- $O(n^3) \to O(n^2)$ flops (dense $\to$ data sparse)
- $O(n^2) \to O(n^{4/3})$ flops (sparse $\to$ sparse+data sparse)

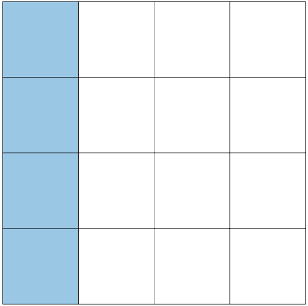
- Complex, hierarchical structure
- $O(n^3) \to O(n \log^2 n)$ flops (dense $\to$ data sparse)
- $O(n^2) \to O(n)$ flops (sparse $\to$ sparse+data sparse)

Standard BLR factorization



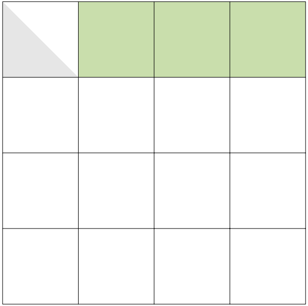
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



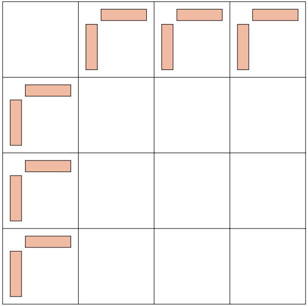
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



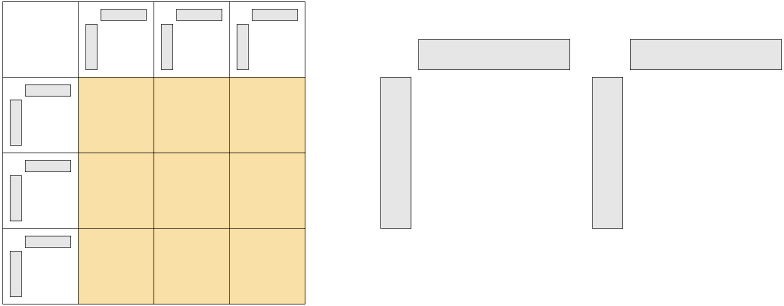
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



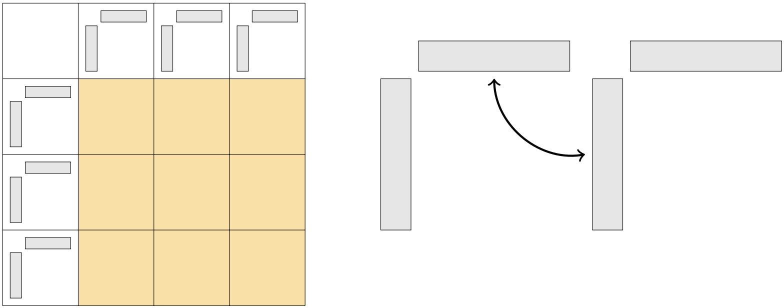
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



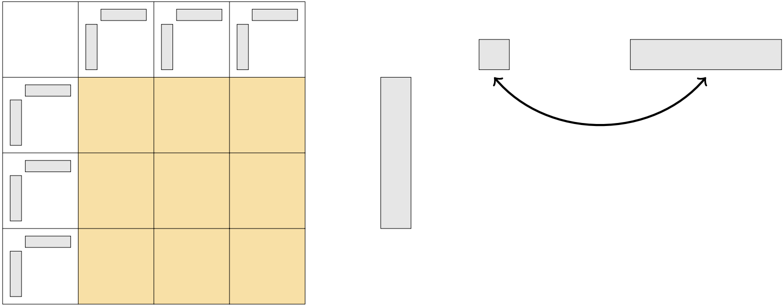
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



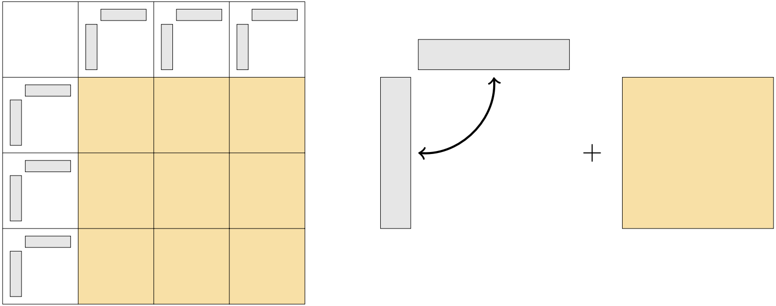
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



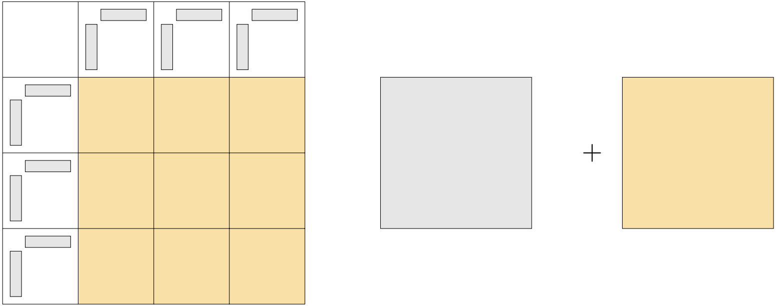
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



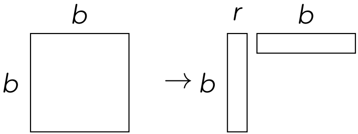
- Adapt blocked LU algorithm to exploit low-rank blocks

Standard BLR factorization



- Adapt blocked LU algorithm to exploit low-rank blocks

Challenges with data sparse algorithms



- Low granularities: low-rank matrices have much smaller granularities ($r \ll b$), making computations inefficient (BLAS-2 → 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

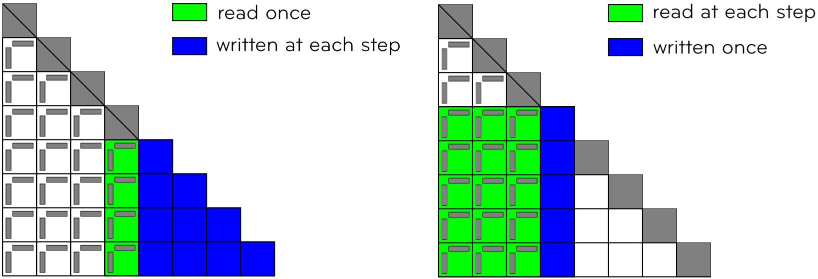
  $\Rightarrow$ relative weight of memory/communications much higher than for dense computations!

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

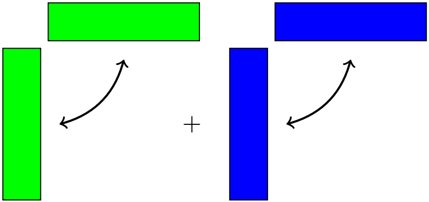
	FR time		BLR time	
	RL	LL	RL	LL
Update	338	336	110	67
Total	424	421	221	175

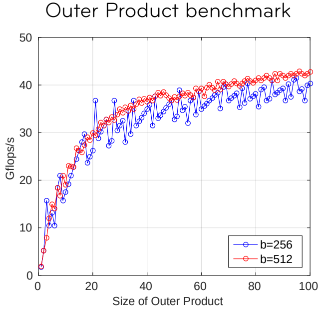


RL factorization LL factorization

⇒ Lower volume of memory transfers in LL

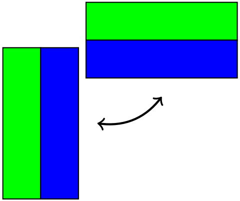
Low-rank update accumulation (LUA)

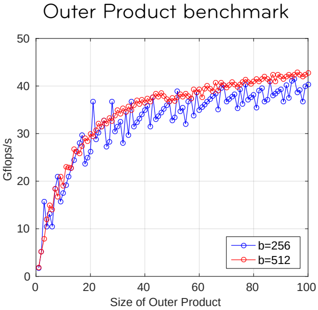




		LL
average size of Outer Product		16.5
time (s)	Outer Product	21
	Total	175

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

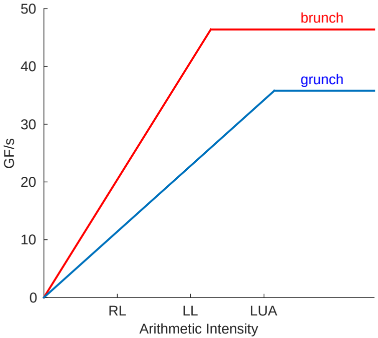
	specs		time (s) for BLR factorization		
	peak (GF/s)	bw (GB/s)	RL	LL	LUA
grunch (28 threads)	37	57	248	228	196
brunch (24 threads)	46	102	221	175	167

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 ⇒ more efficient cache use)

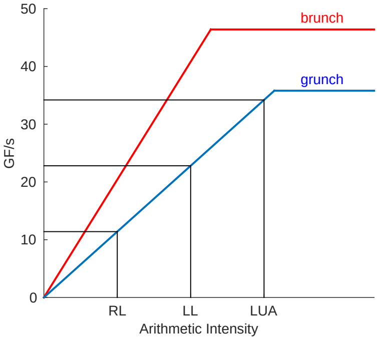


Impact of machine properties on BLR: roofline model

	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 ⇒ more efficient cache use)

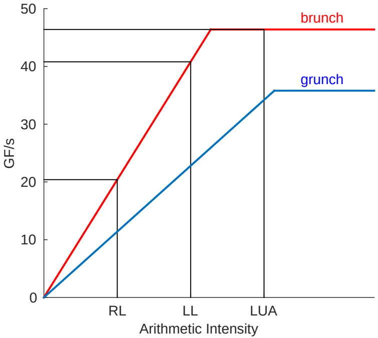


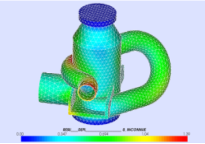
Impact of machine properties on BLR: roofline model

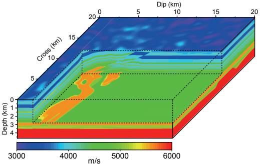
	specs		time (s) for BLR factorization		
	peak (GF/s)	bw (GB/s)	RL	LL	LUA
grunch (28 threads)	37	57	248	228	196
brunch (24 threads)	46	102	221	175	167

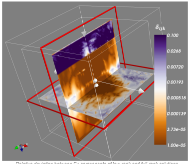
Arithmetic Intensity in BLR:

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









Structural mechanics
Matrix of order 8M
Required accuracy: 10^{-9}

Seismic imaging
Matrix of order 17M
Required accuracy: 10^{-3}

Electromagnetism
Matrix of order 30M
Required accuracy: 10^{-7}

Results on 900 cores:

	factorization time (s)			memory/proc (GB)		
application	before	after	ratio	before	after	gain
structural	289.3	104.9	2.5	7.9	5.9	25%
seismic	617.0	123.4	4.9	13.3	10.4	22%
electromag.	1307.4	233.8	5.3	20.6	14.4	30%

Conclusions

- Today's computing is full of errors ⇒ today's HPC should be approximate
  - Low precisions, iterative refinement
  - Data sparsisty, low-rank approximations
  - We will see how to combine them in AFAE
- Fundamental to develop **rigorous underlying theory** to know what and when to approximate! (more on this in AFAE)
- But also fundamental to go all the way to the **end-user application** to assess the true potential of the methods

⇒ Approximate HPC is a challenging but exciting field!

Sujets de stages et thèses

- **Sujet 1: méthodes itératives préconditionnées en précision mixte (LIP6, Paris)**
  - Quel choix de préconditionneur?
  - Comment mélanger les précisions?
  - Analyse théorique d'erreur pour répondre à ces questions
  - Evaluation sur applications industrielles d'IFP Energies Nouvelles
- **Sujet 2: formats BLR pour la taille extrême (LIP6, Paris)**
  - Du fait de sa complexité superlinéaire, le BLR atteint ses limites pour des problèmes de taille extrême (∼100M)
  - Représentations multiniveaux et/ou partagées pour aller plus loin
  - Développement et optimisation de ces nouveaux formats dans un logiciel libre mondialement reconnu (MUMPS)
  - Evaluation sur applications industrielles de Mumps Technologies
- **Sujet 3: approximations de rang faible randomisées en précision mixte (IRIT, Toulouse)**

Sujet de stages détaillés:
`https://bit.ly/stagesHPC`

Contact: `theo.mary@lip6.fr`