How fast can you drive your computer?

Paolo Bientinesi
Umeå Universitet

Kunskapsveckan, Umeå
Oct 28, 2020

About me

Professor in High-Performance Computing

High Performance and Automatic Computing

HPC2N

High-Performance Computing Center North

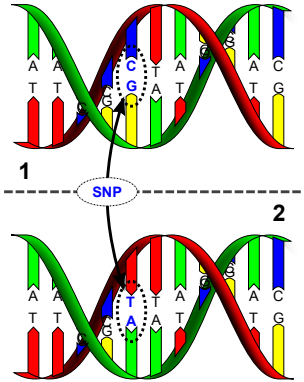
About me

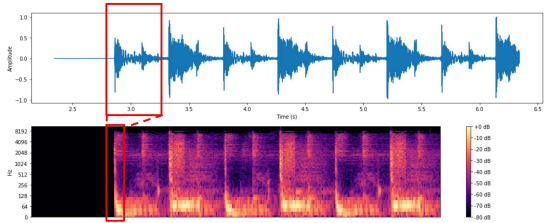
Professor in High-Performance Computing

High-Performance Computing Center North

"High-Performance Computing" ?

Computing


1
SNP
2

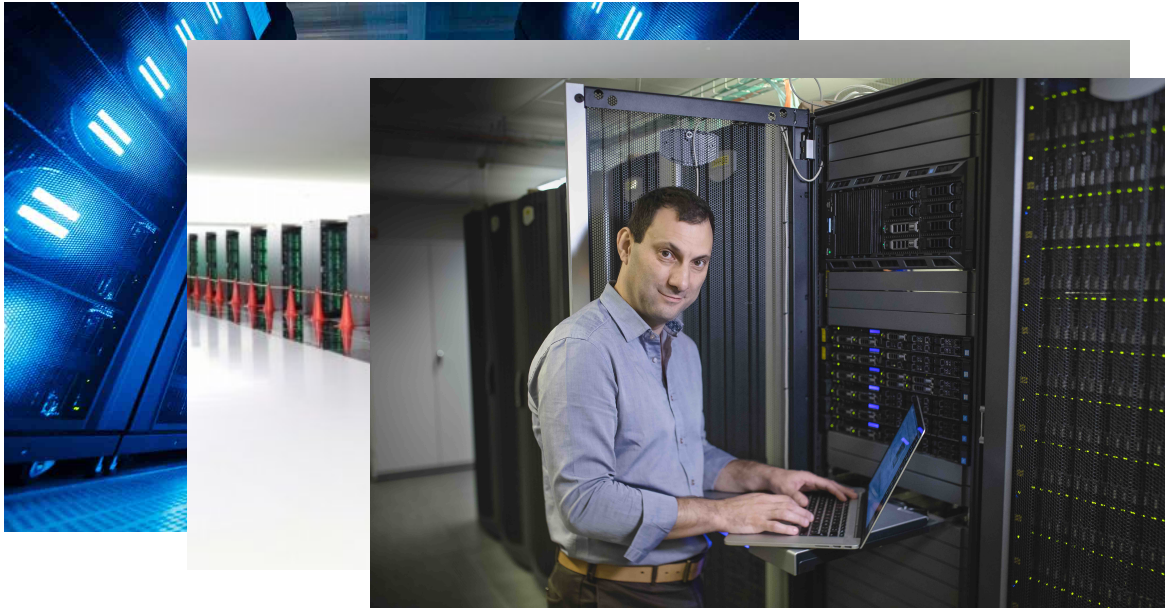

Amplitude
Time (s)
Hz

High-performance

High-performance $\rightarrow$ Supercomputers

High-performance $\rightarrow$ Supercomputers

High-performance



In actuality

Every computer is a supercomputer, even laptops!

Analogy: Computers = Fast cars

Analogy: Computers = Fast cars

speed

Km/h

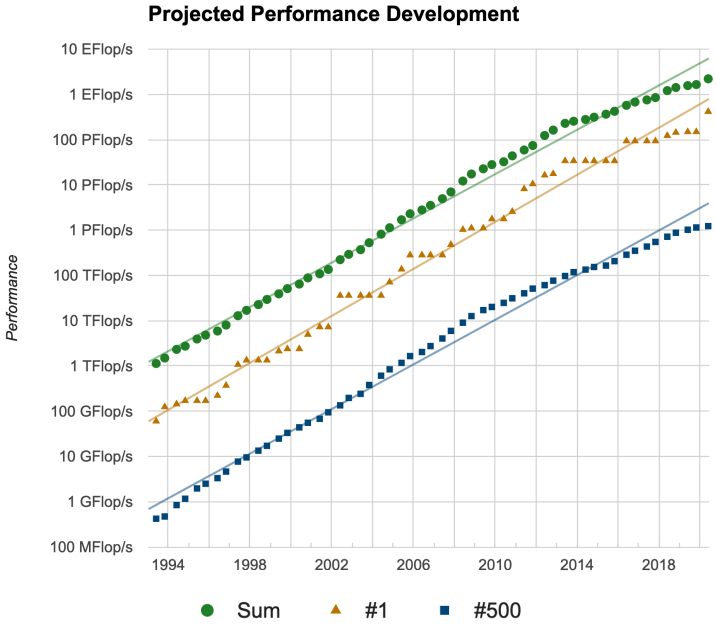
Analogy: Computers = Fast cars

speed

FLOPS/sec

Km/h

Fastest computers in the world



My laptop

"Peak performance" = Top speed

My laptop

"Peak performance" = Top speed =

Freq $\times$ #cores $\times$ #flops/cycle =

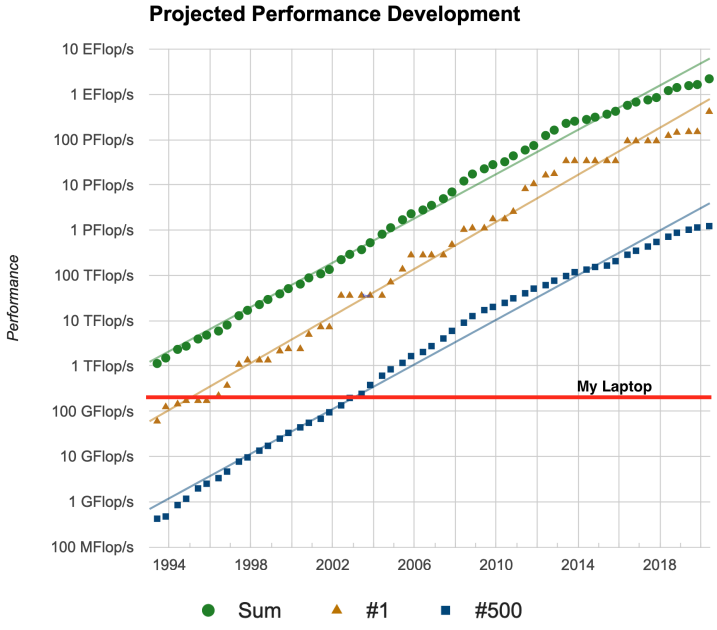
My laptop

"Peak performance" = Top speed =

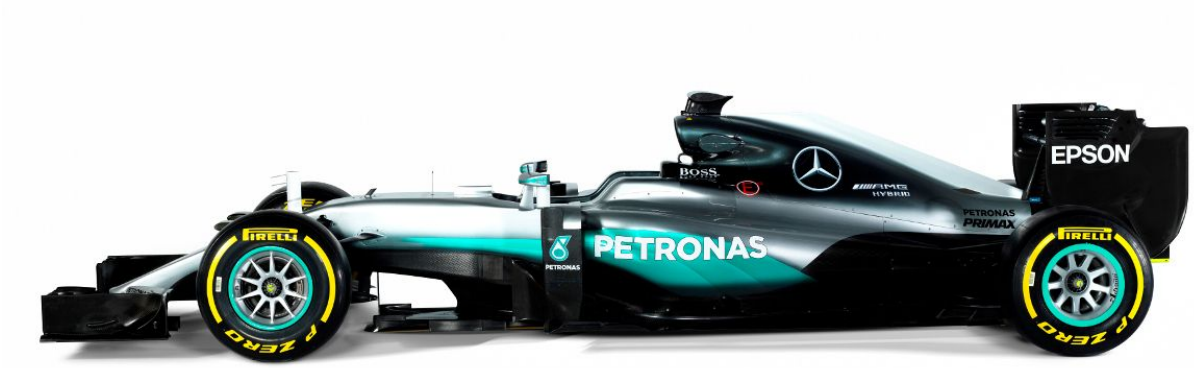
Freq $\times$ #cores $\times$ #flops/cycle =

2.8×10^9 Ghz $\times$ 4 $\times$ 16 ops/cycle = 179.2×10^9 ops/sec = **179 GFlops/sec**

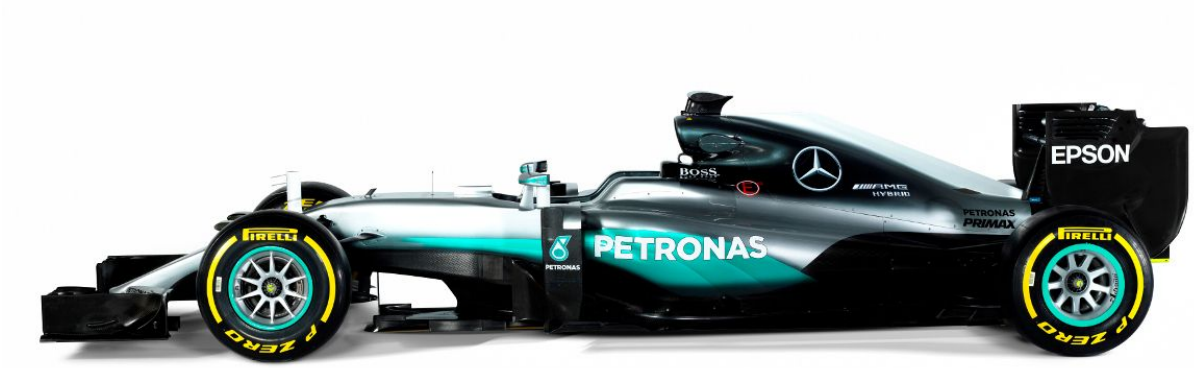
My laptop > #1 supercomputer of 1996 !!!



We all drive one of these



We all drive one of these



but . . . do we know how to drive that?

Do we drive it like this?

. . . like this?

or . . . ?

How to “drive” computers?

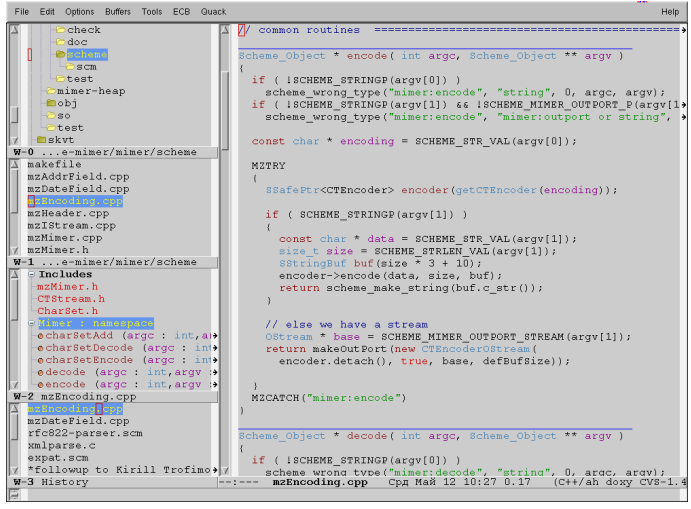
How to “drive” computers?

Programming language

How to “drive” computers?

Programming language

Program



Program $\mathcal{P}$

- $[[\mathcal{P}]]$ = *semantics* of $\mathcal{P}$ = "what does the program do?"

- Perf($\mathcal{P}$) = *performance* of $\mathcal{P}$ = "how fast does the program run?"

Program $\mathcal{P}$

- $[[\mathcal{P}]]$ = *semantics* of $\mathcal{P}$ = "what does the program do?"

  Whose responsibility? Programmer

- Perf($\mathcal{P}$) = *performance* of $\mathcal{P}$ = "how fast does the program run?"

  Whose responsibility? Language/compiler/interpreter

Program $\mathcal{P}$

- $[[\mathcal{P}]]$ = *semantics* of $\mathcal{P}$ = "what does the program do?"

  Whose responsibility? Programmer

- Perf($\mathcal{P}$) = *performance* of $\mathcal{P}$ = "how fast does the program run?"

  Whose responsibility? Language/compiler/interpreter $\rightarrow$ Wishful thinking

Program $\mathcal{P}$

- $[[\mathcal{P}]]$ $=$ *semantics* of $\mathcal{P}$ $=$ "what does the program do?"

  Whose responsibility? Programmer

- Perf($\mathcal{P}$) $=$ *performance* of $\mathcal{P}$ $=$ "how fast does the program run?"

  Whose responsibility? Language/compiler/interpreter $\rightarrow$ Wishful thinking

  Why is this important?

Example

Refresher: Matrices

$$\begin{cases} 3x - 2y + z + 2w = -10 \\ 2x + 2y - 8z - w = 0 \\ -x - y + 3.5z = -2 \\ 5x - 3y + 3z - 2w = 16 \end{cases}$$

Example

Refresher: Matrices

$$\begin{cases} 3x - 2y + z + 2w = -10 \\ 2x + 2y - 8z - w = 0 \\ -x - y + 3.5z = -2 \\ 5x - 3y + 3z - 2w = 16 \end{cases}$$

$$A = \begin{pmatrix} 3 & -2 & 1 & 2 \\ 2 & 2 & -8 & -1 \\ -1 & -1 & 3.5 & 0 \\ 5 & -3 & 3 & -2 \end{pmatrix} \quad k = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \quad b = \begin{pmatrix} 10 \\ 0 \\ -2 \\ 16 \end{pmatrix}$$

Example

Refresher: Matrices

$$\begin{cases} 3x - 2y + z + 2w = -10 \\ 2x + 2y - 8z - w = 0 \\ -x - y + 3.5z = -2 \\ 5x - 3y + 3z - 2w = 16 \end{cases}$$

$$A = \begin{pmatrix} 3 & -2 & 1 & 2 \\ 2 & 2 & -8 & -1 \\ -1 & -1 & 3.5 & 0 \\ 5 & -3 & 3 & -2 \end{pmatrix} \quad k = \begin{pmatrix} x \\ y \\ z \\ w \end{pmatrix} \quad b = \begin{pmatrix} 10 \\ 0 \\ -2 \\ 16 \end{pmatrix}$$

$$Ak = b$$

A seemingly elementary example: Matrix-Matrix product

Essential operation at the core of most matrix computations

$C := AB + C$

$$\forall i \; \forall j \quad C_{ij} := C_{ij} + \sum_k A_{ik} B_{kj}$$

A seemingly elementary example: Matrix-Matrix product

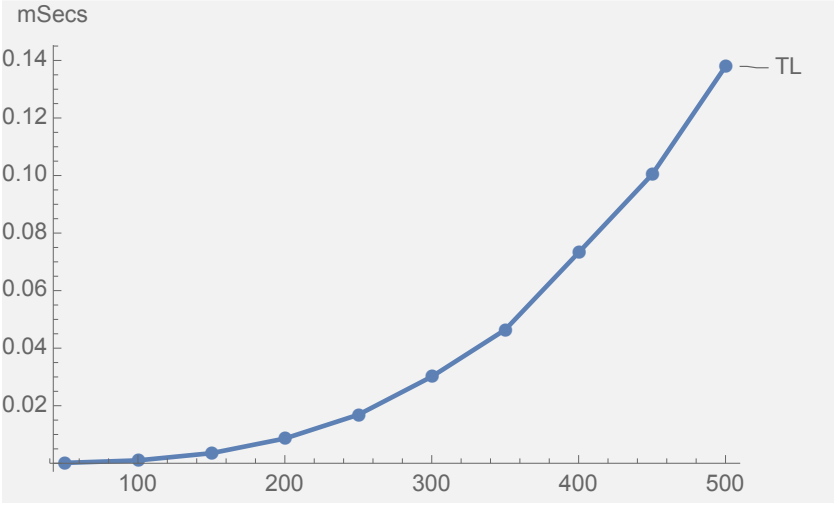
Essential operation at the core of most matrix computations

$C := AB + C$

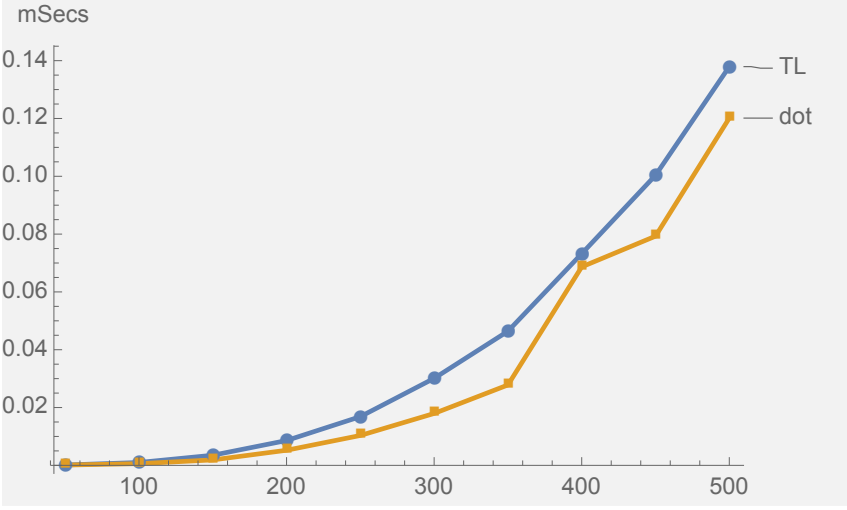
$$\forall i \; \forall j \quad C_{ij} := C_{ij} + \sum_k A_{ik} B_{kj}$$

```
for( i=0; i<m; i++ )
   for( j=0; j<n; j++ )
      for( l=0; l<k; l++ )
         C[i+j*m] += A[i+l*m] * B[l+j*k];
```

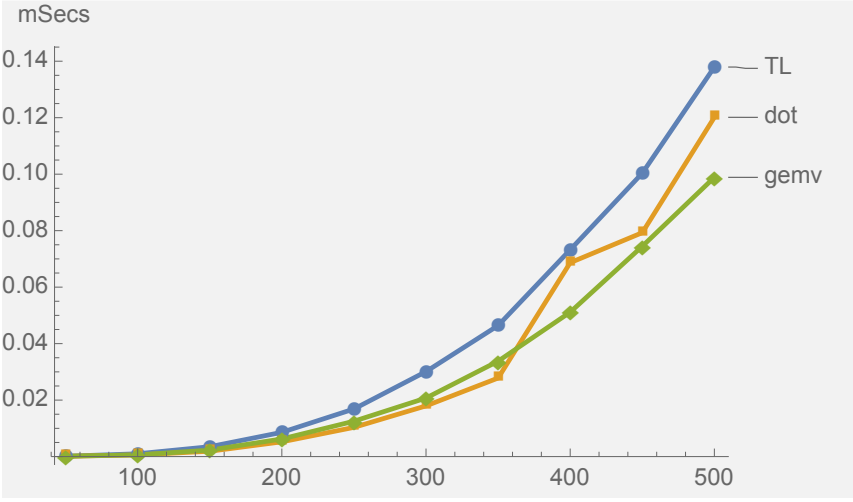
Algorithm #1: a few lines of code



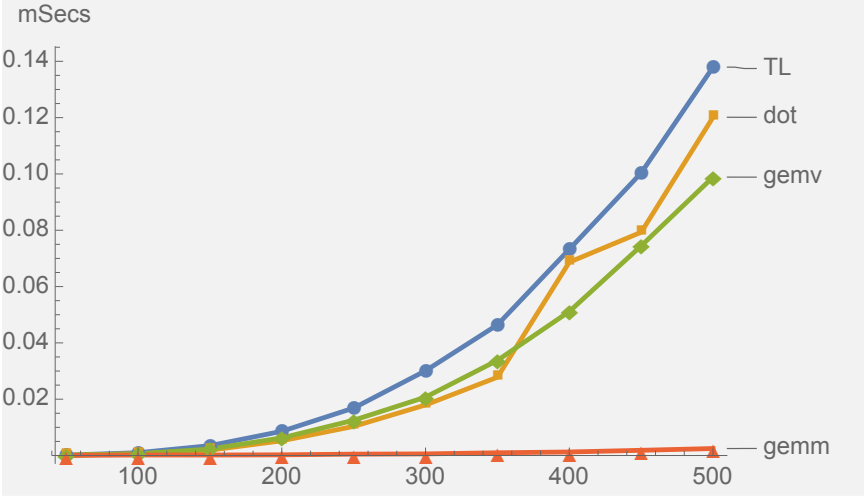
Algorithm #2: a few lines of code



Algorithm #3: a few lines of code



Algorithm #4: thousands of lines of code



Another analogy: Parallelism (concurrency)

Another analogy: Parallelism (concurrency)

Morale

"Driving" computers fast is more and more challenging

Compilers: smart and powerful (with numbers)

Compilers: smart and powerful (with numbers)

- Simplifications

$$\alpha := \beta * \gamma * \beta^{-1} \quad \rightarrow \quad \alpha := \gamma$$

Compilers: smart and powerful (with numbers)

- Simplifications

$$\alpha := \beta * \gamma * \beta^{-1} \quad \rightarrow \quad \alpha := \gamma$$

- Multiplication times 0 or 1

Compilers: smart and powerful (with numbers)

- Simplifications

$$\alpha := \beta * \gamma * \beta^{-1} \quad \rightarrow \quad \alpha := \gamma$$

- Multiplication times 0 or 1
- Common expressions

$$\begin{array}{l} \alpha := \beta * \gamma \\ \delta := \epsilon * \beta * \gamma \end{array} \quad \rightarrow \quad \begin{array}{l} \alpha := \beta * \gamma \\ \delta := \epsilon * \alpha \end{array}$$

Compilers: smart and powerful (with numbers)

- Simplifications

$$\alpha := \beta * \gamma * \beta^{-1} \quad \rightarrow \quad \alpha := \gamma$$

- Multiplication times 0 or 1
- Common expressions

$$\begin{array}{l} \alpha := \beta * \gamma \\ \delta := \epsilon * \beta * \gamma \end{array} \quad \rightarrow \quad \begin{array}{l} \alpha := \beta * \gamma \\ \delta := \epsilon * \alpha \end{array}$$

- Code motion

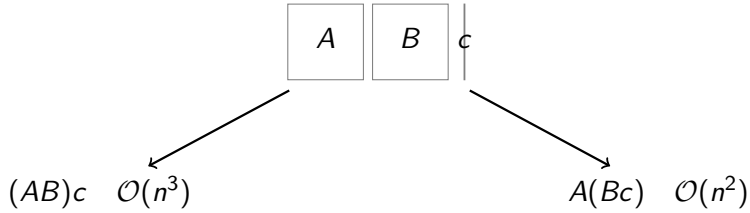
```
for( i=0; i<n; i++ ){
  a = 34 * 45;
  C[i] += A[i] * B[i];
}
```

```
a = 34 * 45;
for( i=0; i<n; i++ ){
  C[i] += A[i] * B[i];
}
```

- ...

Compilers: NOT so smart with matrices

Example: Parenthesisation



Product is associative, but its cost is not

Matrix computations

Signal Processing	$x := \left(A^{-T}B^{T}BA^{-1} + R^{T}LR\right)^{-1} A^{-T}B^{T}BA^{-1}y$	$R \in \mathbb{R}^{n-1\times n}$, UT; $L \in \mathbb{R}^{n-1\times n-1}$, DI
Kalman Filter	$K_k := P_k^b H^T (HP_k^b H^T + R)^{-1};\ x_k^a := x_k^b + K_k(z_k - Hx_k^b);\ P_k^a := (I - K_K H)\, P_k^b$	
Ensemble Kalman Filter	$X^a := X^b + \left(B^{-1} + H^T R^{-1} H\right)^{-1} \left(Y - HX^b\right)$	$B \in \mathbb{R}^{N\times N}$ SSPD; $R \in \mathbb{R}^{m\times m}$, SSPD
Ensemble Kalman Filter	$\delta X := \left(B^{-1} + H^T R^{-1} H\right)^{-1} H^T R^{-1} \left(Y - HX^b\right)$	
Ensemble Kalman Filter	$\delta X := XV^T \left(R + HX(HX)^T\right)^{-1} \left(Y - HX^b\right)$	
Image Restoration	$x_k := (H^T H + \lambda\sigma^2 I_n)^{-1}(H^T y + \lambda\sigma^2(v_{k-1} - u_{k-1}))$	
Image Restoration	$H^\dagger := H^T(HH^T)^{-1};\ y_k := H^\dagger y + (I_n - H^\dagger H)x_k$	
Rand. Matrix Inversion	$X_{k+1} := S(S^TAS)^{-1}S^T + (I_n - S(S^TAS)^{-1}S^TA)X_k(I_n - AS(S^TAS)^{-1}S^T)$	
Rand. Matrix Inversion	$X_{k+1} := X_k + WA^TS(S^TAWA^TS)^{-1}S^T(I_n - AX_k)$	$W \in \mathbb{R}^{n\times n}$, SPD
Rand. Matrix Inversion	$X_{k+1} := X_k + (I_n - X_kA^T)S(S^TA^TWAS)^{-1}S^TA^TW$	
Rand. Matrix Inversion	$\Lambda := S(S^TAWAS)^{-1}S^T;\ \Theta := \Lambda AW;\ M_k := X_kA - I$ $X_{k+1} := X_k - M_k\Theta - (M_k\Theta)^T + \Theta^T(AX_kA - A)\Theta$	

Matrix computations (2)

Generalized Least Squares	$b := (X^T M^{-1} X)^{-1} X^T M^{-1} y$	$n > m$; $M \in \mathbb{R}^{n \times n}$, SPD; $X \in \mathbb{R}^{n \times m}$; $y \in \mathbb{R}^{n \times 1}$
Stochastic Newton	$B_k := \frac{k}{k-1} B_{k-1}(I_n - A^T W_k((k-1)I_l + W_k^T A B_{k-1} A^T W_k)^{-1} W_k^T A B_{k-1})$	
Optimization	$x_f := WA^T(AWA^T)^{-1}(b - Ax)$; $x_o := W(A^T(AWA^T)^{-1}Ax - c)$	
Optimization	$x := W(A^T(AWA^T)^{-1}b - c)$	
Triangular Matrix Inv.	$X_{10} := L_{10}L_{00}^{-1}$; $X_{20} := L_{20} + L_{22}^{-1}L_{21}L_{11}^{-1}L_{10}$; $X_{11} := L_{11}^{-1}$; $X_{21} := -L_{22}^{-1}L_{21}$	
Tikhonov Regularization	$x := (A^T A + \Gamma^T \Gamma)^{-1} A^T b$	$A \in \mathbb{R}^{n \times m}$; $\Gamma \in \mathbb{R}^{m \times m}$; $b \in \mathbb{R}^{n \times 1}$
Tikhonov Regularization	$x := (A^T A + \alpha^2 I)^{-1} A^T b$	
Gen. Tikhonov Reg.	$x := (A^T PA + Q)^{-1}(A^T Pb + Qx_0)$	$P \in \mathbb{R}^{n \times n}$, SSPD; $Q \in \mathbb{R}^{m \times m}$, SSPD; $x_0 \in \mathbb{R}^{m \times 1}$
Gen. Tikhonov reg.	$x := x_0 + (A^T PA + Q)^{-1}(A^T P(b - Ax_0))$	
LMMSE estimator	$K_{t+1} := C_t A^T (AC_t A^T + C_z)^{-1}$; $x_{t+1} := x_t + K_{t+1}(y - Ax_t)$; $C_{t+1} := (I - K_{t+1}A)C_t$	
LMMSE estimator	$x_{\text{out}} = C_X A^T (AC_X A^T + C_Z)^{-1}(y - Ax) + x$	
LMMSE estimator	$x_{\text{out}} := (A^T C_Z^{-1} A + C_X^{-1})^{-1} A^T C_Z^{-1}(y - Ax) + x$	

Compilers: NOT so smart with matrices

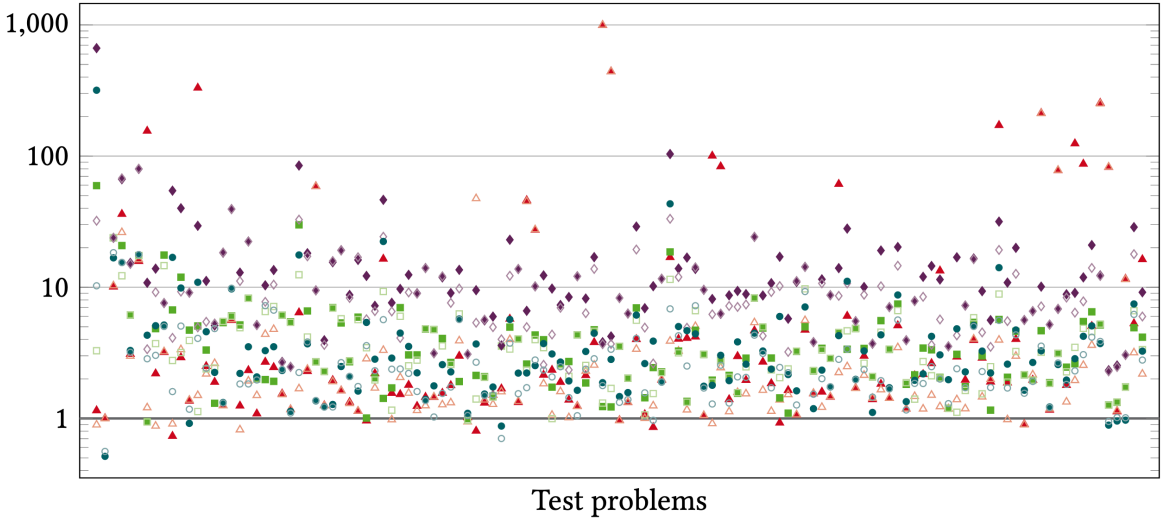
- Parenthesization
- Matrix properties
- Properties might propagate as computation unfolds
- Common subexpressions may/may not help
- Space needed for intermediate results
- ...

Domain-specific compiler

Henrik Barthels, Christos Psarras

- Linear algebra knowledge (from textbooks)
  - properties
  - equalities
  - theorems
- High-performance computing expertize (from experience)

Linnea speedups



Linnea vs. Julia, Armadillo, Eigen, Matlab, Octave, Python, R

Conclusion

- Despite 60 years of development, when it comes to matrix computations, compilers still have a long way to go
- Modern computers: It is increasingly more difficult to use their full potential
- Modern programming languages: They offer great productivity, but often at the cost of efficiency
- Linnea: our own compiler specialized in matrix computations

Thank you for your attention