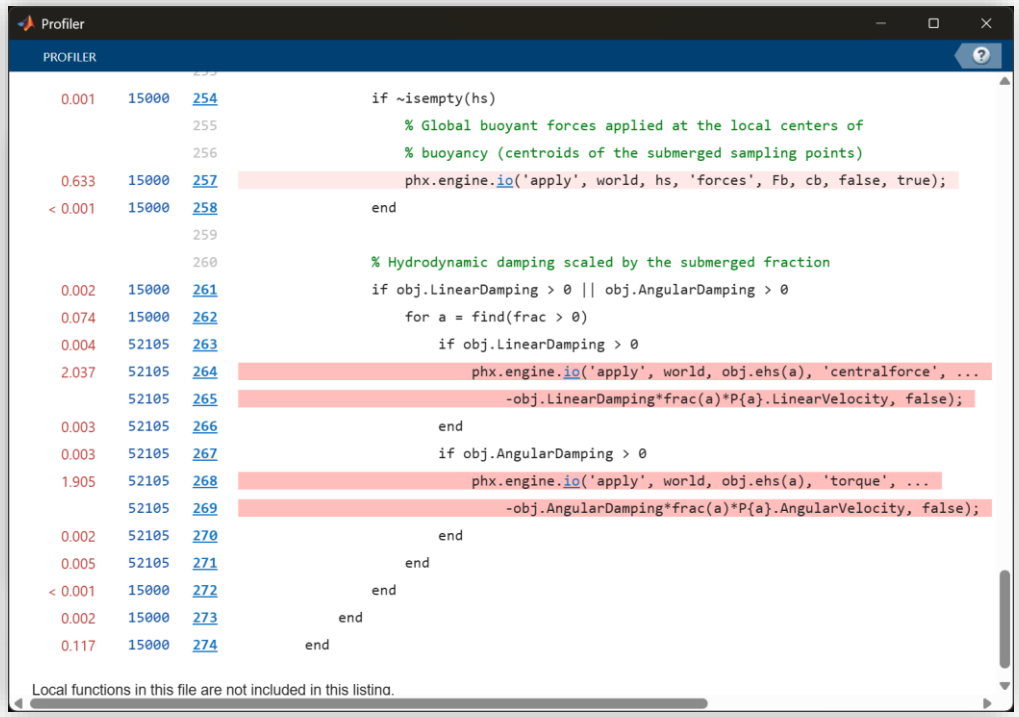
Optimalizace výkonu kódu v MATLABu



Lubor Zháňal

zhanal@humusoft.cz

www.humusoft.cz

Proč se tím zabývat

- stejný výsledek, jiný čas
  - často o řád, občas o tři
- kde se čas ztrácí nejčastěji
  - cyklus tam, kde stačí jedna operace
  - kopie, o které nevíte
  - volání navíc na každém prvku
- většina zrychlení je pár řádků, ne přepis

1052×

největší rozdíl, na jaký jsme při měření narazili

Nejdřív měřit, pak optimalizovat

špatně

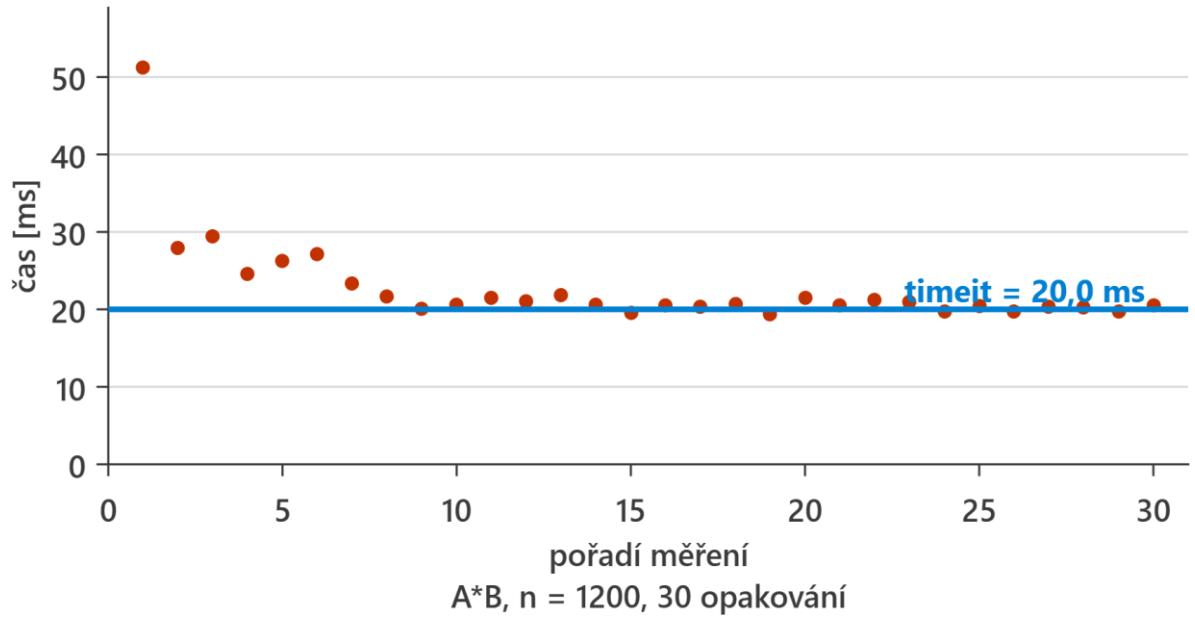
```
tic
vysledek = vypocet(data);
toc
```

dobře

```
t = timeit(@() vypocet(data));
```

- první běh měří warm-up, ne kód
- timeit opakuje, vrací medián, odečte režii měření

Všechna čísla v této přednášce: MATLAB R2026a Update 5, AMD Ryzen AI MAX+ 395 (16 jader), 24 GB RAM, Windows 11.

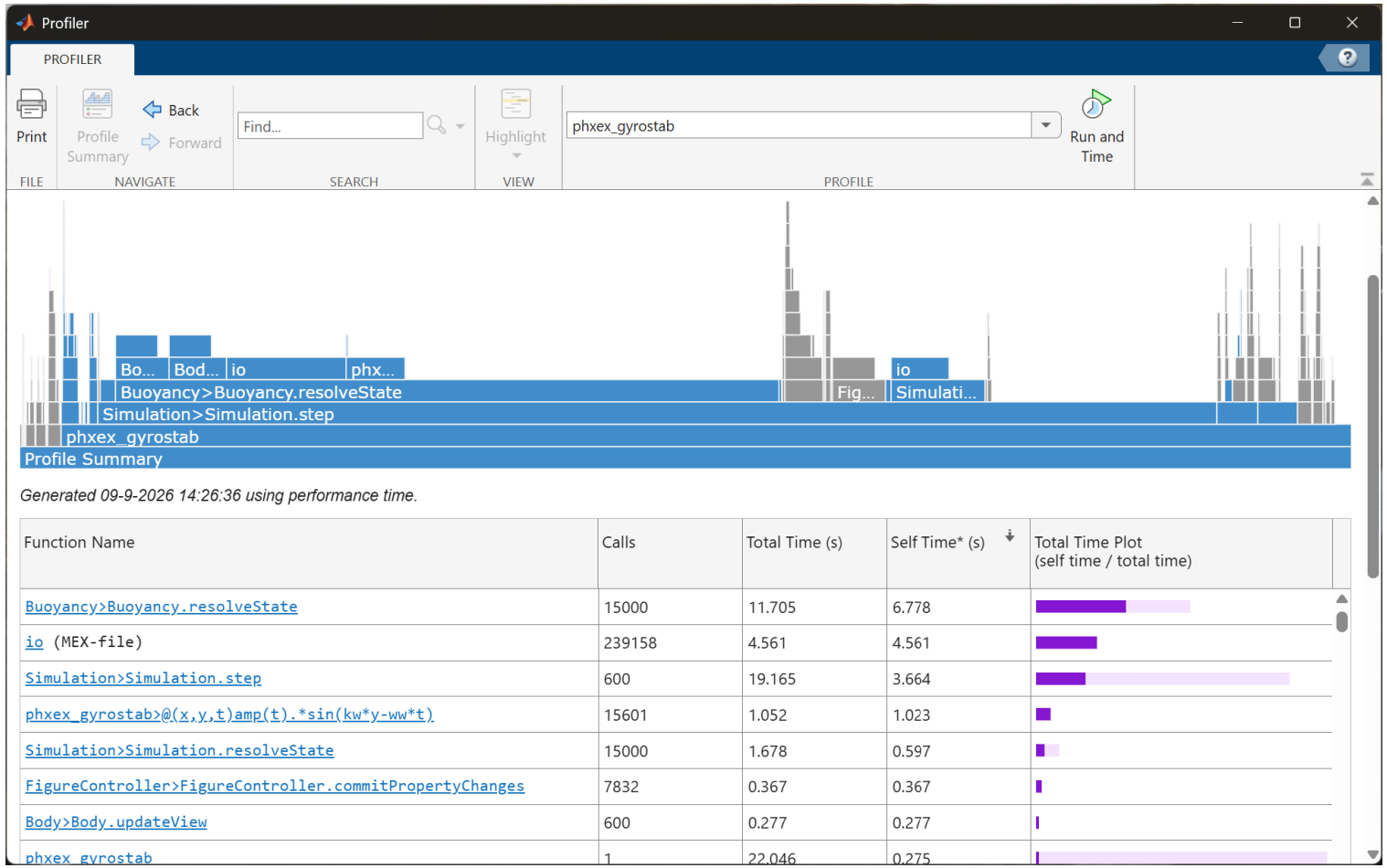


Profiler ukáže, kde ten čas je

```
profile on
mujProgram;
profile viewer
```

- v Editoru tlačítko Run and Time
- vlastní čas vs. celkový čas
- optimalizujte přednostně to, co trvá nejdéle (vlastní čas)

- pozor na režii profilingu



Základy

co se vyplatí vždycky

Prealokace

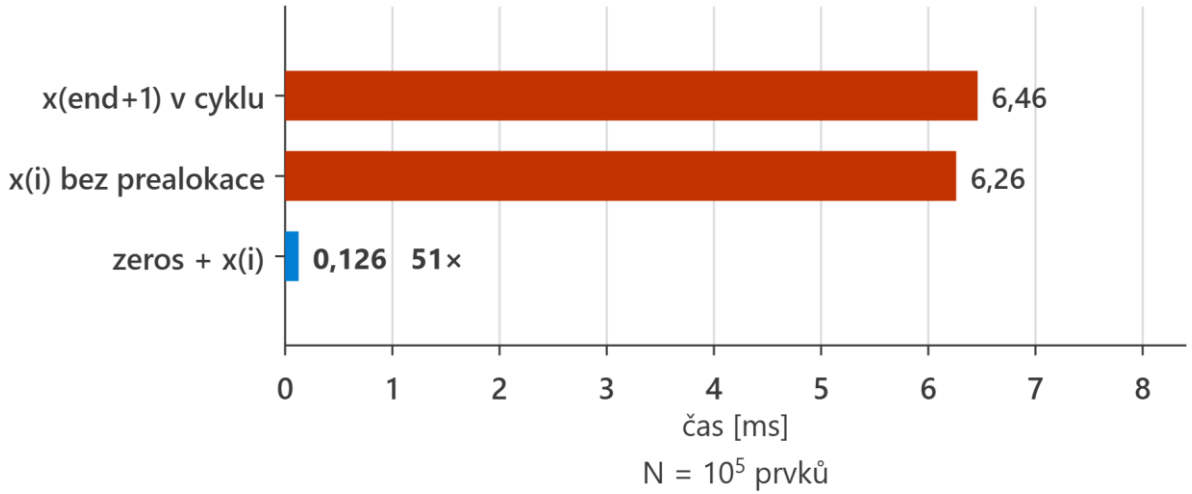
```
n = 100000;              % počet prvků
```

špatně

```
x = [];
for i = 1:n
    x(end+1) = f(i);
end
```

dobře

```
x = zeros(1, n);
for i = 1:n
    x(i) = f(i);
end
```



- když rozměry předem neznán
  - prealokovat velkoryse více, pak oříznout
  - zvětšovat pole po blocích během výpočtu

Vektorizace

```
x = rand(1, 3000);      % řádkový vektor
y = rand(1, 3000);      % řádkový vektor
% chceme D(i,j) = |x(i) - y(j)|, tedy 3000×3000
```

špatně

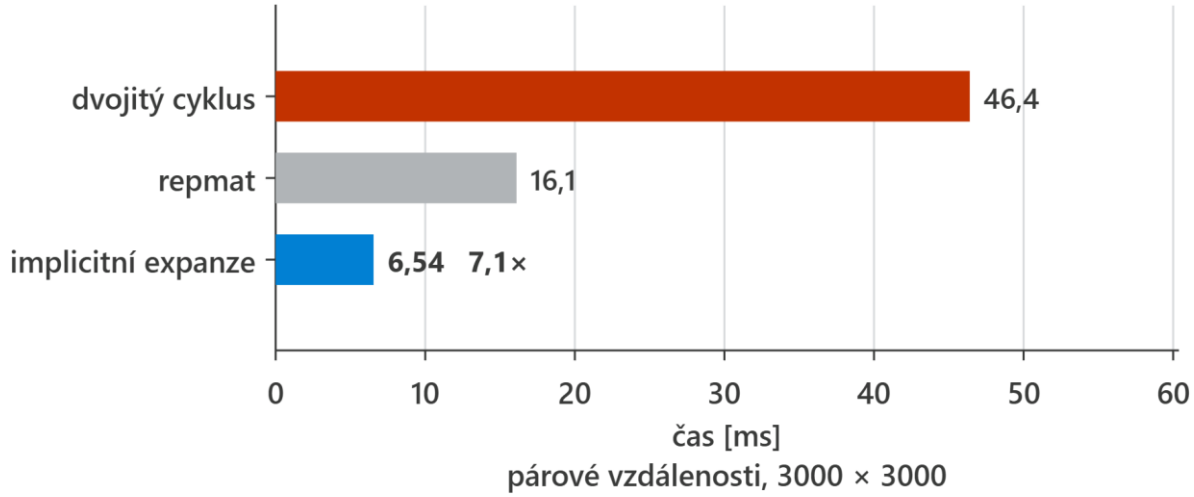
```
for i = 1:n
    for j = 1:n
        D(i,j) = abs(x(i) - y(j));
    end
end
```

lépe

```
D = abs(repmat(x', 1, 3000) - repmat(y, 3000, 1));
```

dobře

```
D = abs(x' - y);
```



- repmat obě matice fyzicky vyrobí, implicitní expanze ne

Kdy naopak vyhraje cyklus

```
x = rand(1, 1e7);       % řádkový vektor
prah = 0.999;
```

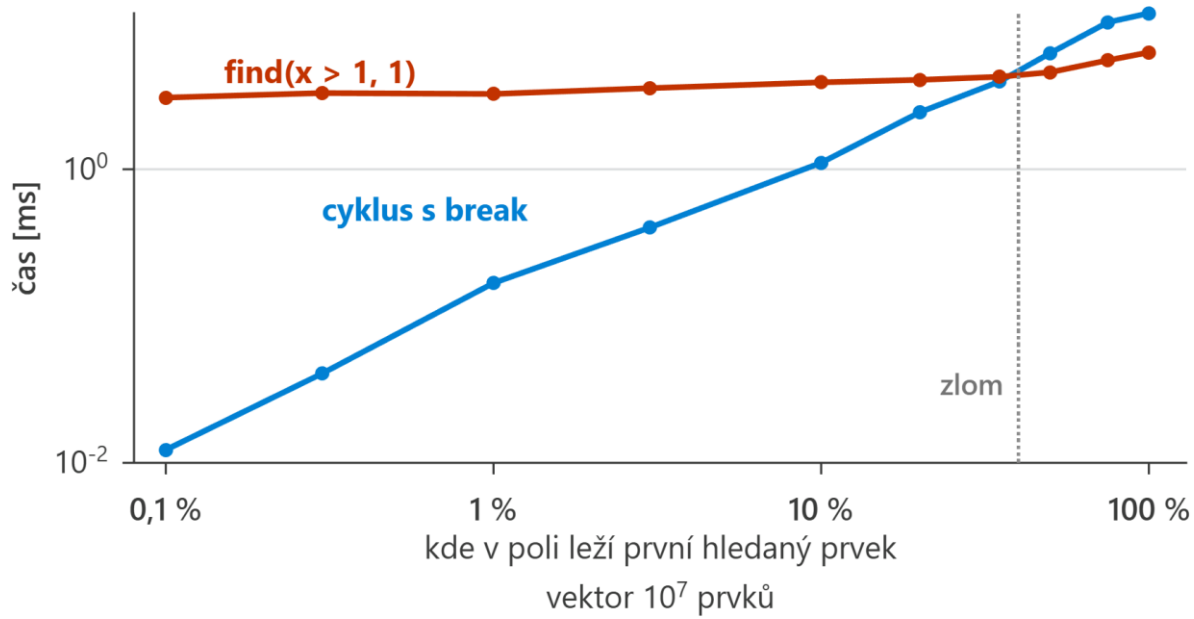
vektorizace spočítá vždycky všechno

```
i = find(x > prah, 1);
```

cyklus může přestat

```
for i = 1:numel(x)
    if x(i) > prah, break; end
end
```

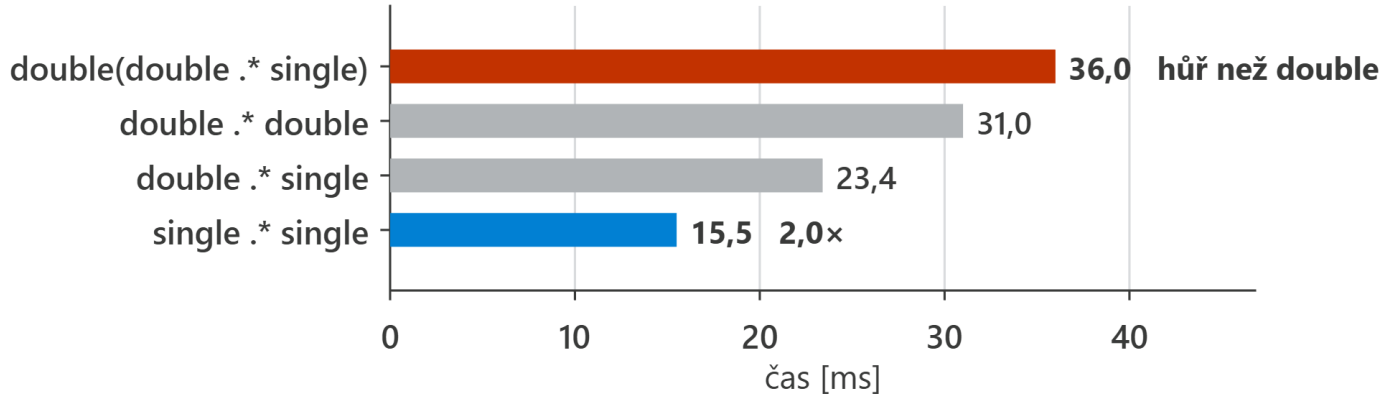
- vyplatí se, dokud hledaný prvek leží v první třetině pole
  - hned na začátku 210×, na konci naopak 1,8× hůř
- totéž u drahé práce jen na pár prvcích
  - cyklus ji přeskočí, maska ji spočítá a pak zahodí
- rehabilitace cyklů: čitelnost je také hodnota



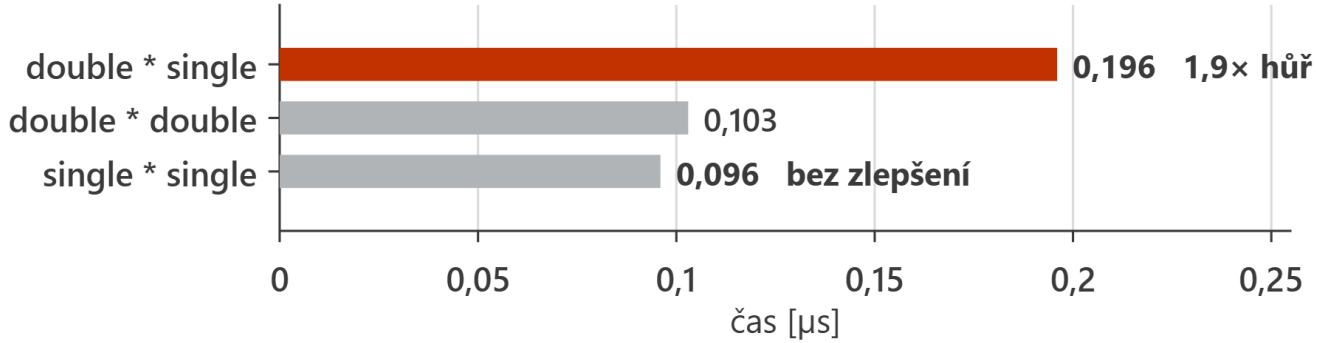
Volba číselného typu

```
d = rand(5e7, 1);    % 400 MB
s = single(d);       % 200 MB
```

- single je 2× rychlejší a zabere polovinu paměti
  - zisk se objeví až od matic asi 16×16
- převod výsledku zpět na double úsporu smaže
  - double(d.*s) je pomalejší než rovnou d.*d
- u malých matic nezáleží na typu, ale na míchání
- cena: 7 platných číslic místo 16



vektor $5 \cdot 10^7$ prvků, minimum ze 7 měření



matice 4×4, 10^5 opakování v cyklu

Po sloupcích, ne po řádcích

```
A = rand(5000);         % matice 5000×5000
```

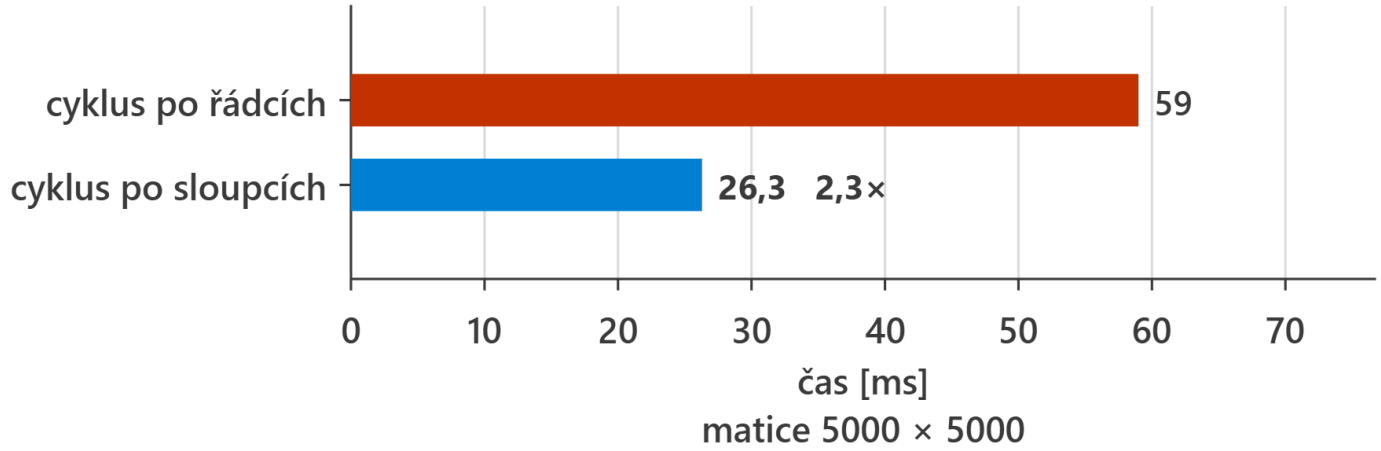
špatně

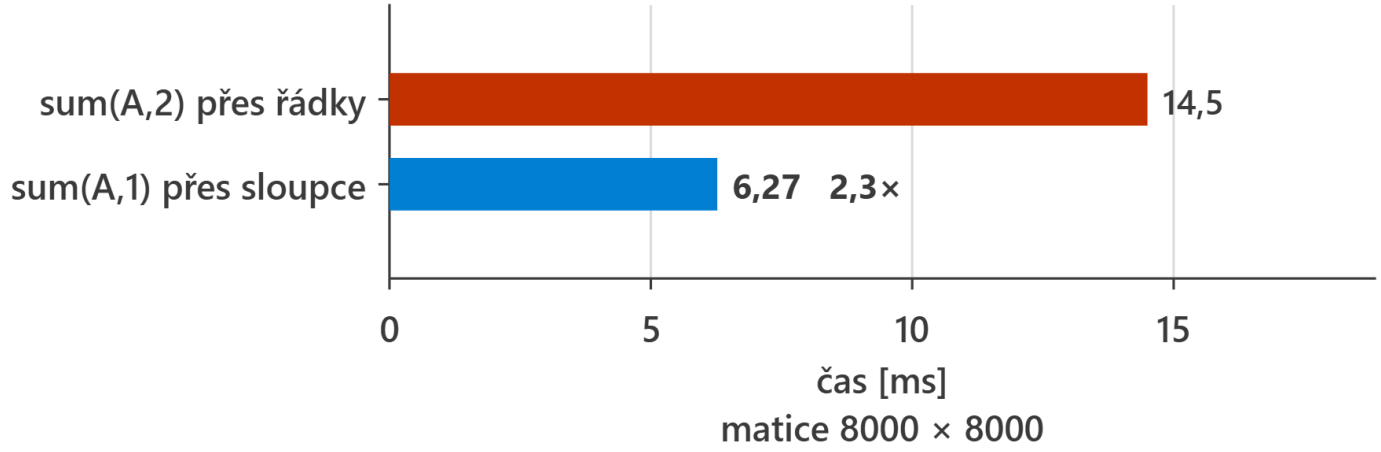
```
for i = 1:size(A,1)
    for j = 1:size(A,2)
        s = s + A(i,j);
    end
end
```

dobře

```
for j = 1:size(A,2)
    for i = 1:size(A,1)
        s = s + A(i,j);
    end
end
```

- MATLAB ukládá matice po sloupcích
- projeví se, až data přetečou cache

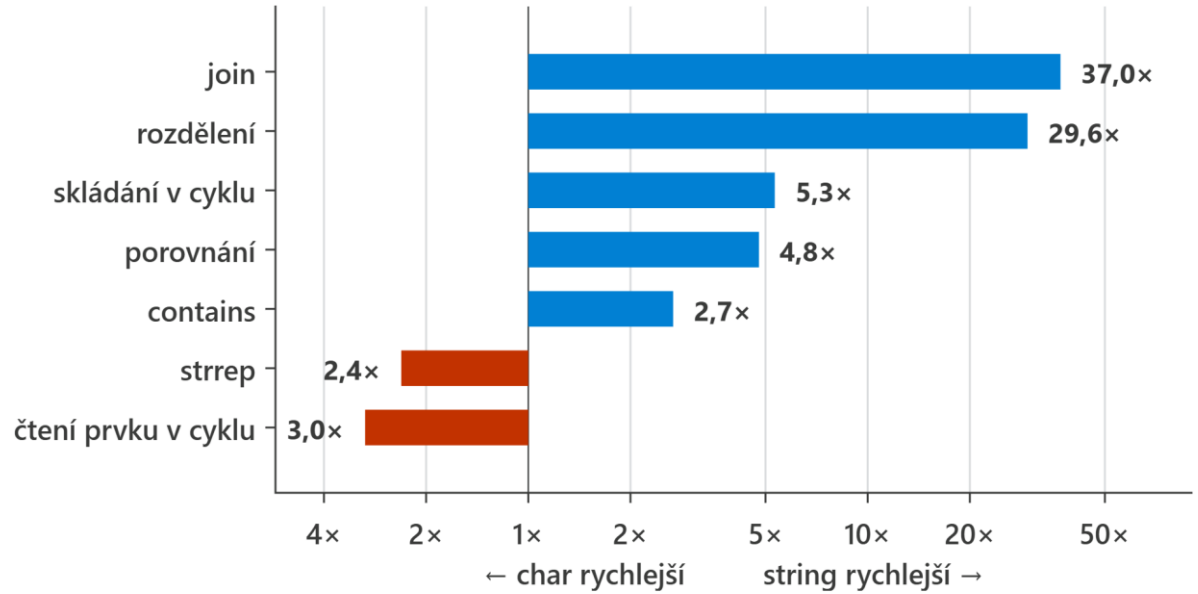




char nebo string?

```
c = cellstr(...);   % 100 000 charů v cell array
s = string(...);    % string array 1×100 000
```

- string vyhrává nad celým polem najednou
  - join 37×, rozdělení 30×, porovnání 4,8×
- char vyhrává při čtení prvku po prvku v cyklu
  - 3× — a u strrep dokonce i vektorově, 2,4×
- závisí na verzi MATLABu



Past dynamického jména

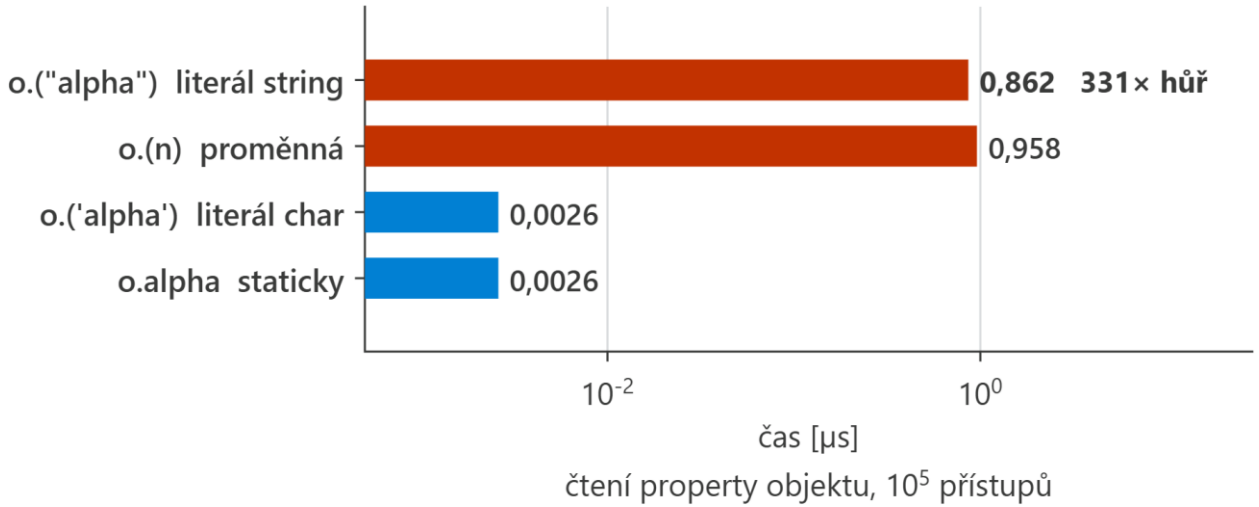
zdarma — MATLAB to přeloží na statický přístup

```
o.('alpha')
```

331× dráž — literál se nepřeloží

```
o.("alpha")
```

- s apostrofy zadarmo, s uvozovkami plná cena
- jakmile je jméno v proměnné, platíte vždycky
  - char může být o něco rychlejší
- u struktur je dopad menší než u objektů

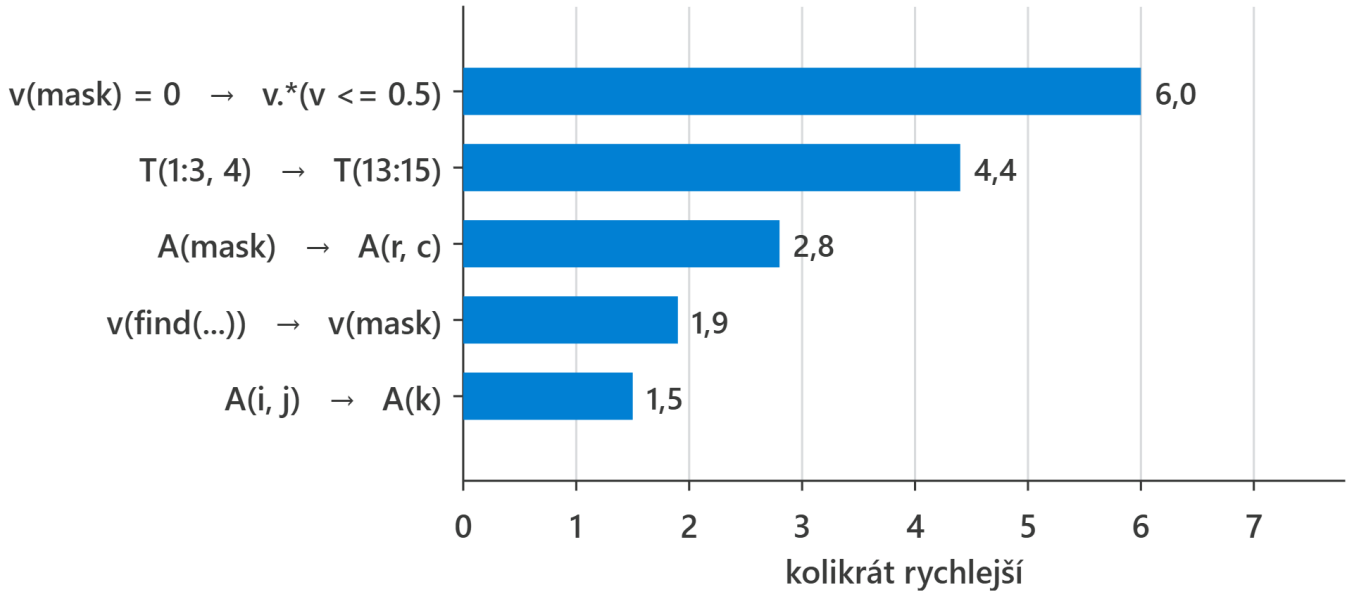


Souřadnicové, lineární, nebo logické?

```
T = eye(4);          % transformační matice 4×4
A = rand(2000);      % matice 2000×2000
v = rand(1, 1e7);    % řádkový vektor

% táž translace, dva zápisy
p = T(1:3, 4);       % souřadnicové
p = T(13:15);        % lineární
```

- pár prvků nebo skalár: vyhrává lineární
  - ale jen při čtení, u zápisu je o 8 % pomalejší
- souvislý blok: naopak souřadnicové
  - je 2,8× rychlejší a hned správného rozměru
- výběr podle podmínky: logická maska, ne find
- nejrychlejší bývá neindexovat vůbec :)
  - vybrat až na konci výpočtu



Kopie, o které nevíte

```
A = rand(4000);  % matice 4000×4000, 128 MB
```

bez kopie

```
function s = jenCte(A)
    s = A(1);
```

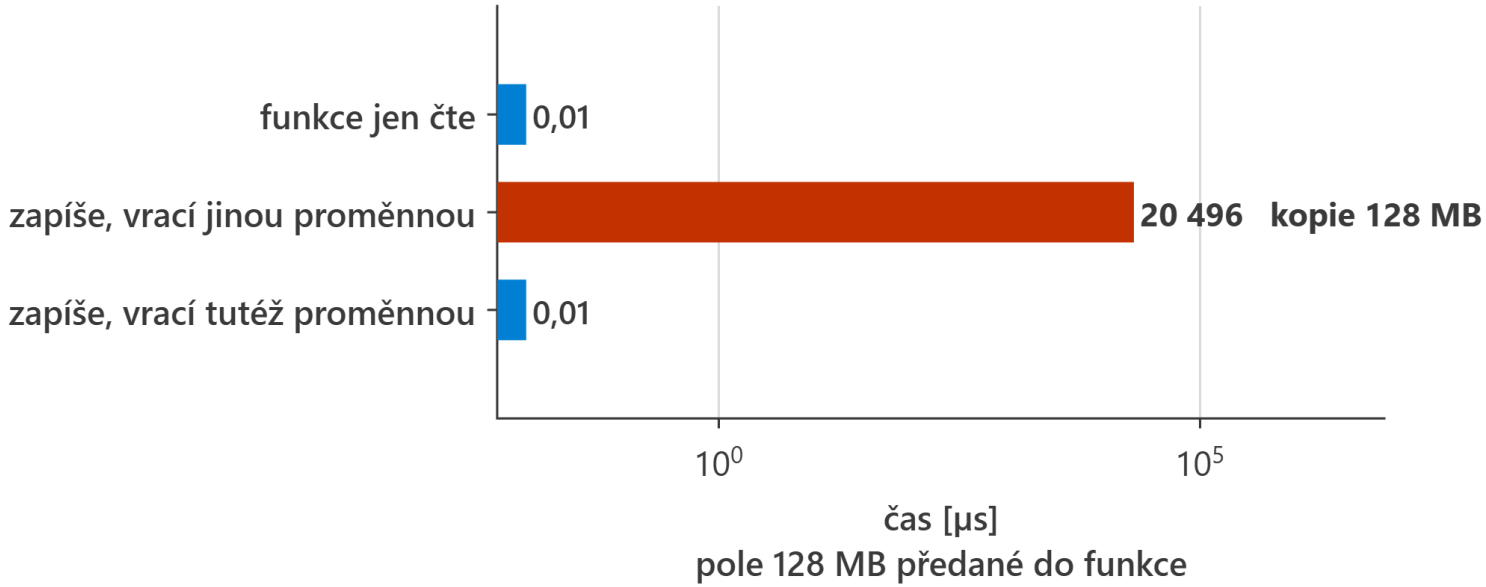
kopie celého pole

```
function s = zapise(A)
    A(1) = 0;
```

taky bez kopie

```
function A = zapise(A)
    A(1) = 0;
% volat jako   A = zapise(A);
```

- kopii nedělá zápis, ale to, že si necháte i původní hodnotu
- musí sedět hlava funkce i místo volání
  - B = zapise(A) kopíruje i tehdy, když už A nikdo nepotřebuje



Nechte práci MATLABu

vestavěné funkce jsou vícevláknové

Soustavy rovnic

```
A = rand(2000);         % matice 2000×2000
b = rand(2000, 1);      % sloupcový vektor
B = rand(2000, 100);    % 100 pravých stran
```

špatně

```
x = inv(A)*b;
```

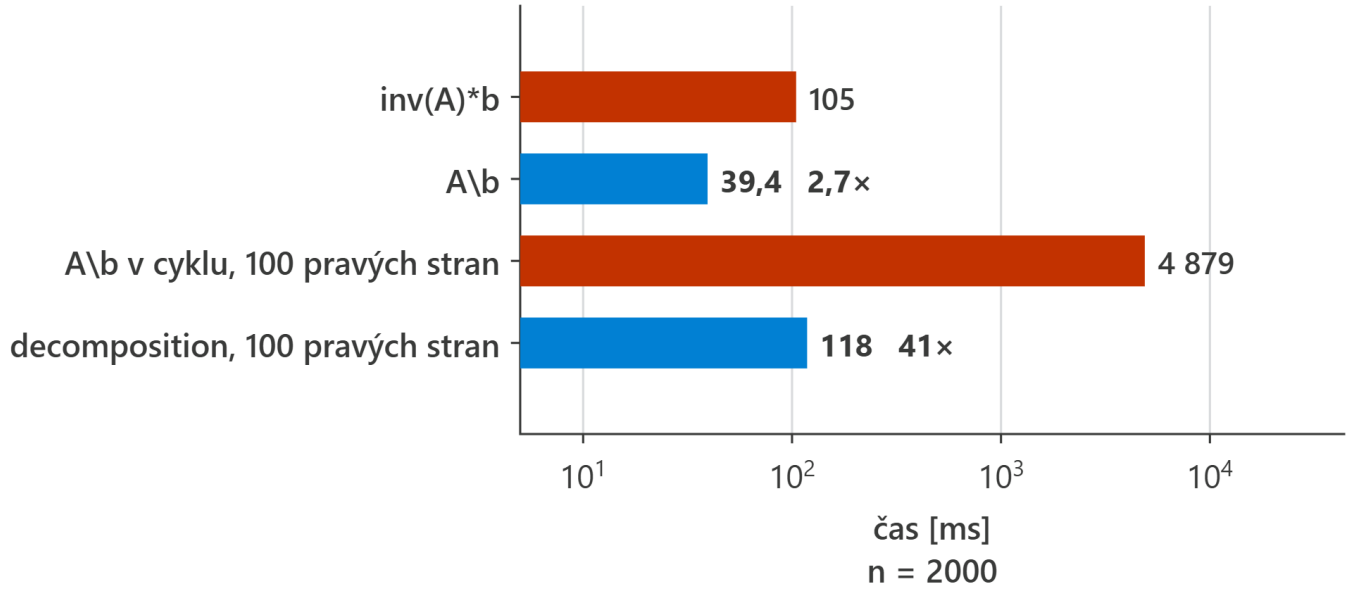
dobře

```
x = A\b;
```

opakované řešení s toutéž maticí

```
dA = decomposition(A);
for k = 1:100
    X(:,k) = dA\B(:,k);
end
```

- rozklad se spočítá jednou, ne stokrát
  - rozklad stojí 39 ms, samotné řešení 0,65 ms



Nevolejte funkci na každý prvek

```
keys = randperm(1e6, 50000);  % 50 000 klíčů
q = keys(randperm(50000));    % dotazy, 1×50 000
```

špatně

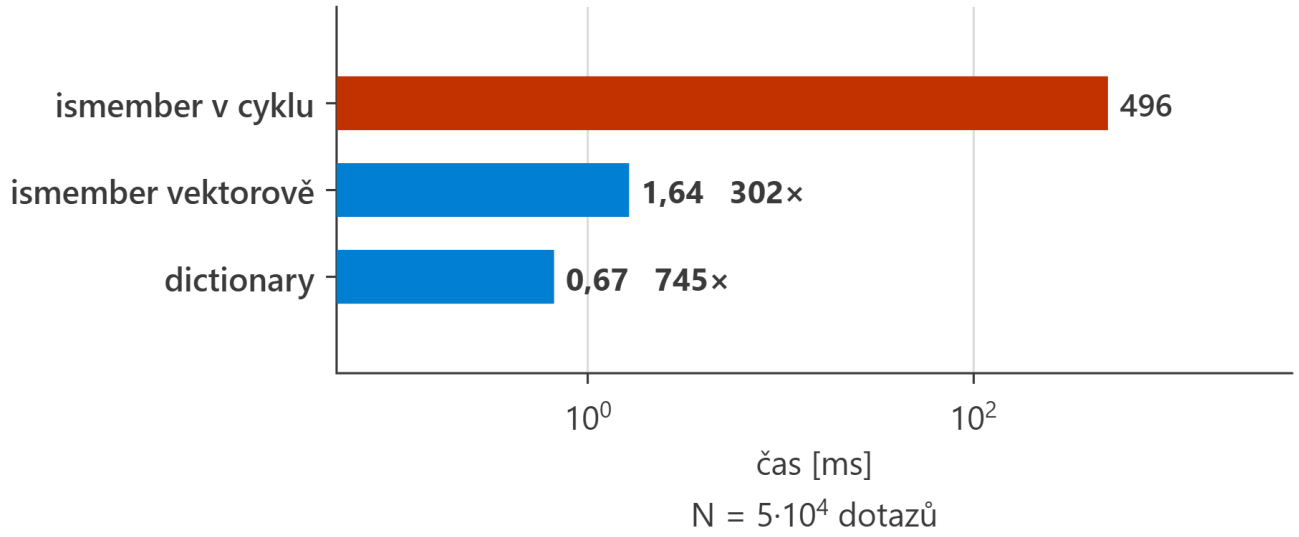
```
for i = 1:n
    tf(i) = ismember(q(i), keys);
end
```

dobře

```
tf = ismember(q, keys);
```

na opakované dotazy

```
d = dictionary(keys, 1:n);
v = d(q);
```



cellfun a arrayfun nejsou zrychlení

```
c = repmat({rand(1, 10)}, 1, 1e5);  % 100 000 buněk
```

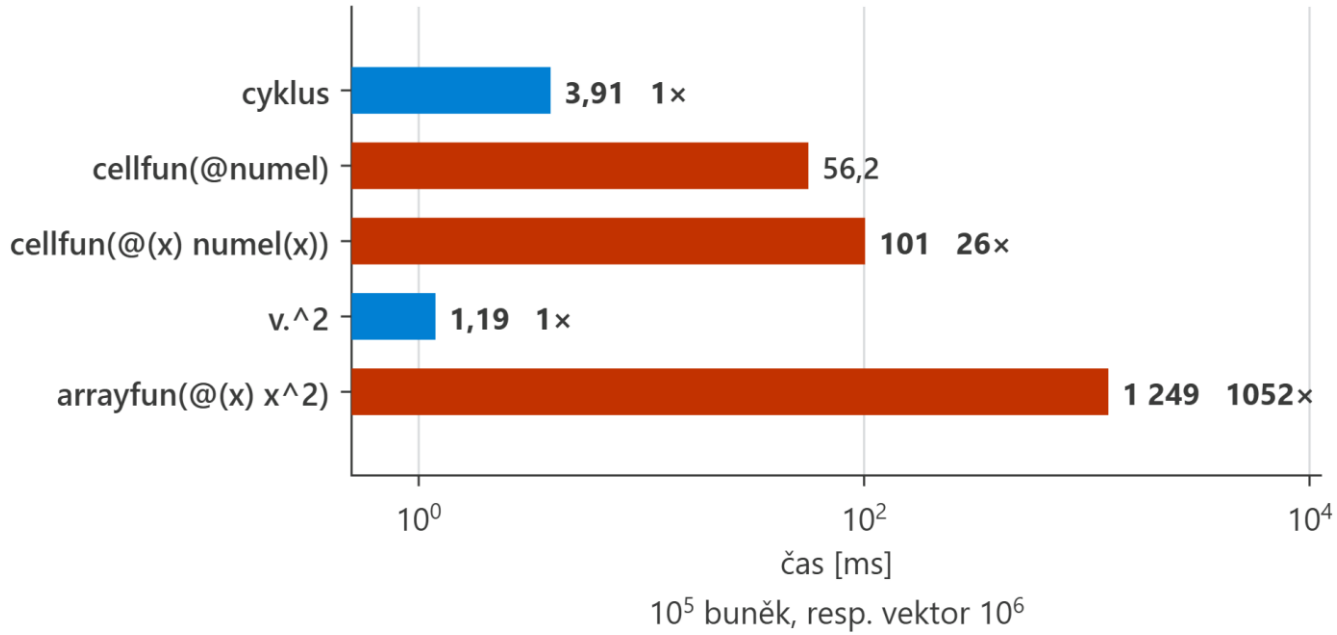
špatně

```
n = cellfun(@(x) numel(x), c);
```

dobře

```
n = zeros(1, numel(c));
for i = 1:numel(c)
    n(i) = numel(c{i});
end
```

- jsou to nástroje na čitelnost, ne na výkon
- anonymní funkce cenu ještě zdvojnásobí



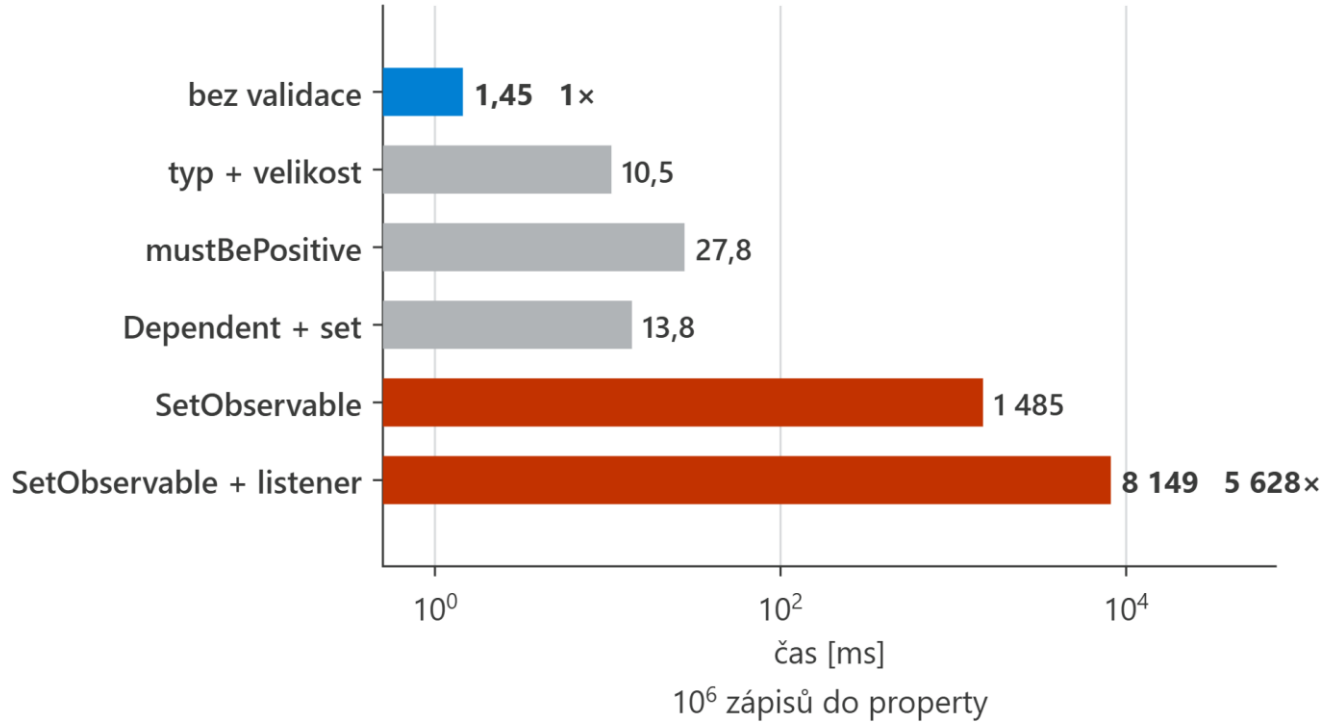
Objekty

kde se v OOP kódu ztrácí čas

Validace se platí při každém zápisu

```
properties
    Prop (1,1) double {mustBePositive}
end
```

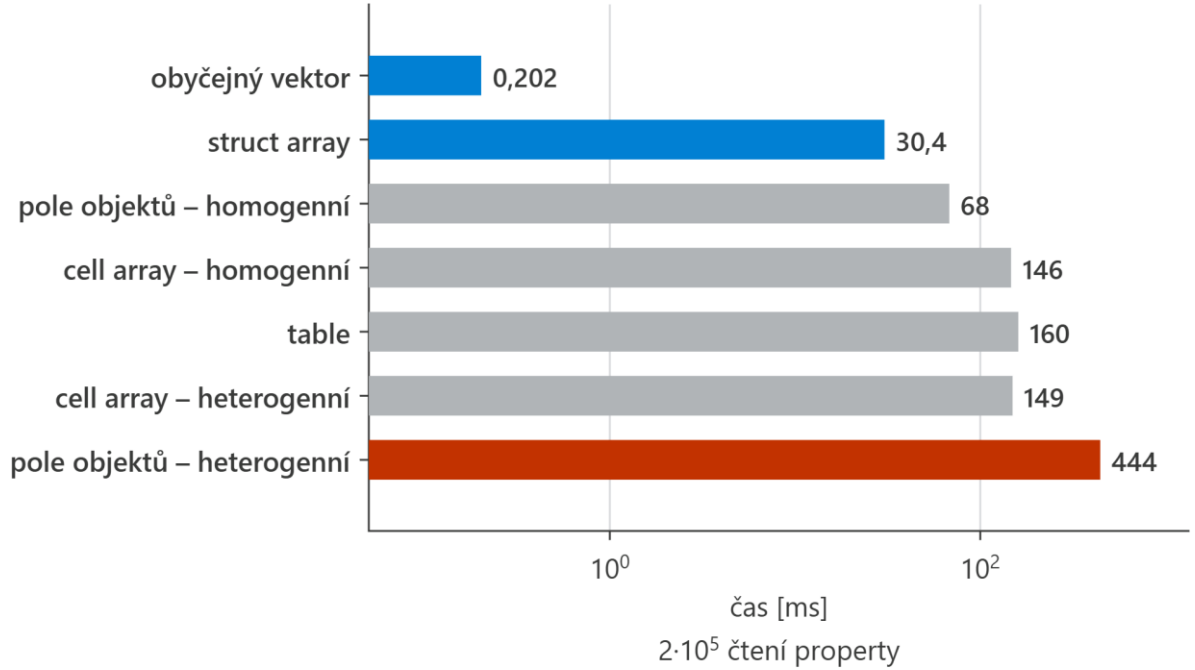
- validátor běží při každém zápisu
  - na čtení nestojí nic
- Dependent stojí i na čtení, 5×
- SetObservable s listenerem 5 628×
- v těsné smyčce si hodnotu načtěte do lokální proměnné
  - 3,6× rychlejší čtení
- validace má smysl zejména pro public properties



Jak držet kolekci objektů

```
% 200 000 objektů, tytéž ve dvou kontejnerech
objArr(i).Prop          % pole objektů
objCell{i}.Prop         % cell array
```

- pole objektů vyhrává jen u homogenní kolekce
  - a zejména umožňuje vektorizaci metod
- cell array vyhrává, jakmile jsou třídy různé
  - a vždycky při volání metod, 2×
- když nutně nepotřebuji objekty, tak struct array je rychlejší než obojí



Nejdražší je objekty vyrobit

```
n = 200000;
pos = rand(3, n);      % 3×n souřadnic částic
```

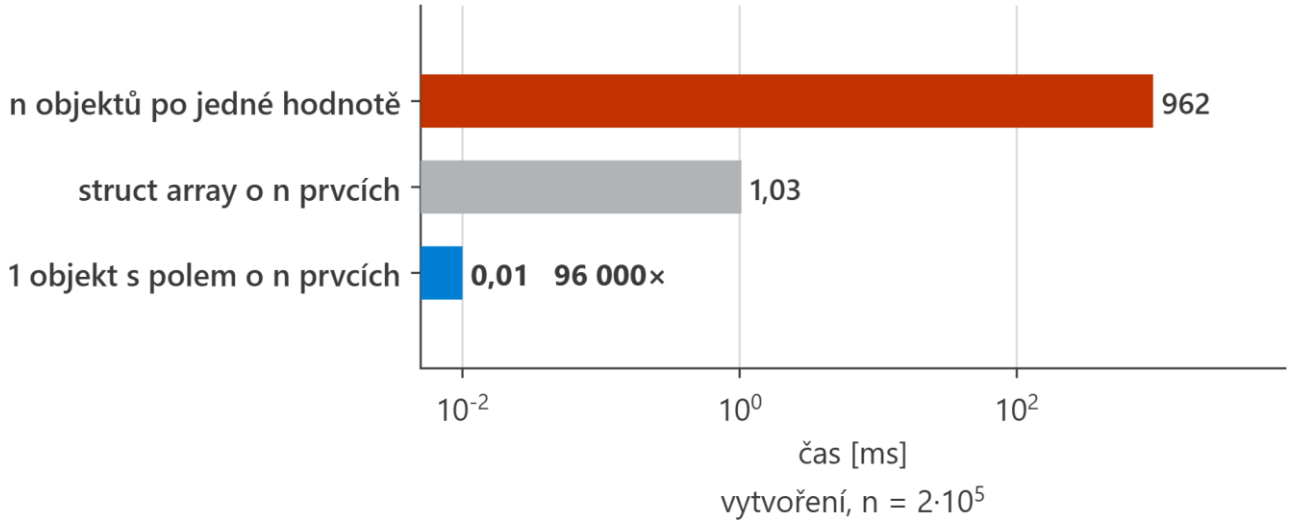
špatně — objekt na každou částici

```
for i = 1:n
    castice{i} = Castice(pos(:,i));
end
```

dobře — jeden objekt, pole uvnitř

```
system = CasticovySystem(pos);
```

- konstruktor se platí za každý objekt zvlášť
- objekt na entitu, ne na každý datový prvek
  - ale také záleží na komplexnosti toho prvku



Když to pořád nestačí

paralelizace a cesta ven z MATLABu

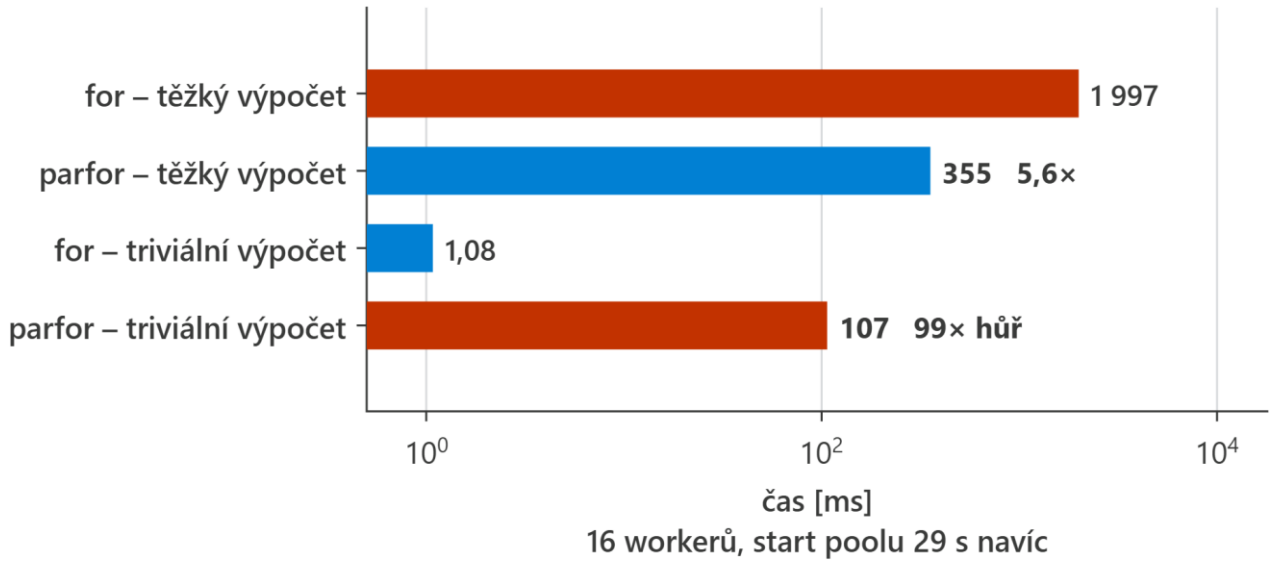
parfor

```
y = zeros(1, 32);       % předalokováno
parfor i = 1:32
    y(i) = tezkyVypocet(i);
end
```

- 16 workerů dá 5,6×, ne 16×
- start poolu 29 s, jednorázově

- na triviální tělo cyklu 99× hůř než for

- vyplatí se, až když jedna iterace trvá desítky ms



MEX a režie volání

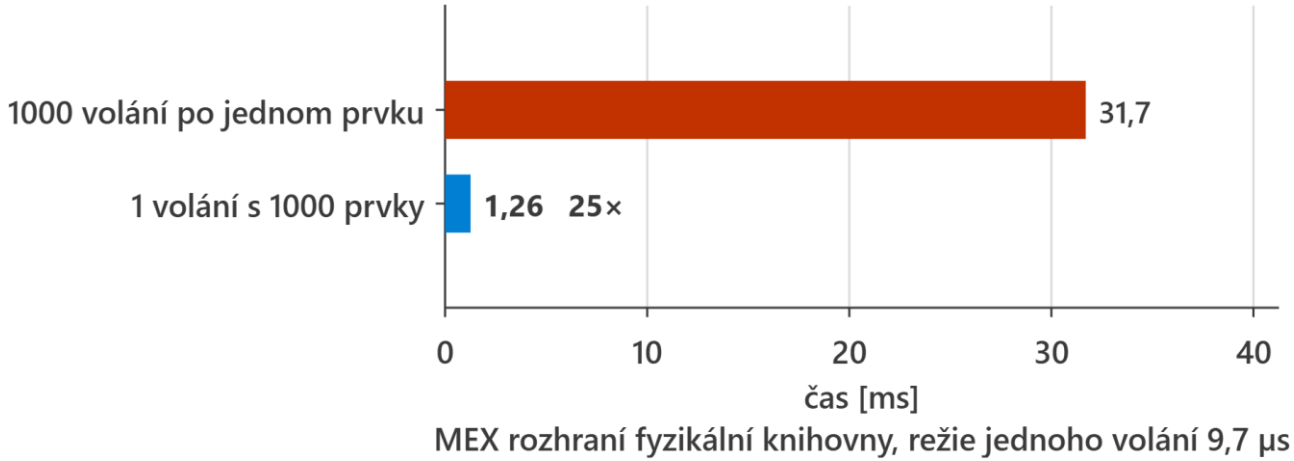
```
x = rand(3, 1000);     % 3×1000, tisíc trojic
% mexFunkce je MEX: každé volání
% překročí hranici mezi MATLABem a C
```

špatně — volání na každý prvek

```
for i = 1:1000
    y(:,i) = mexFunkce(x(:,i));
end
```

dobře — jedno volání na celou dávku

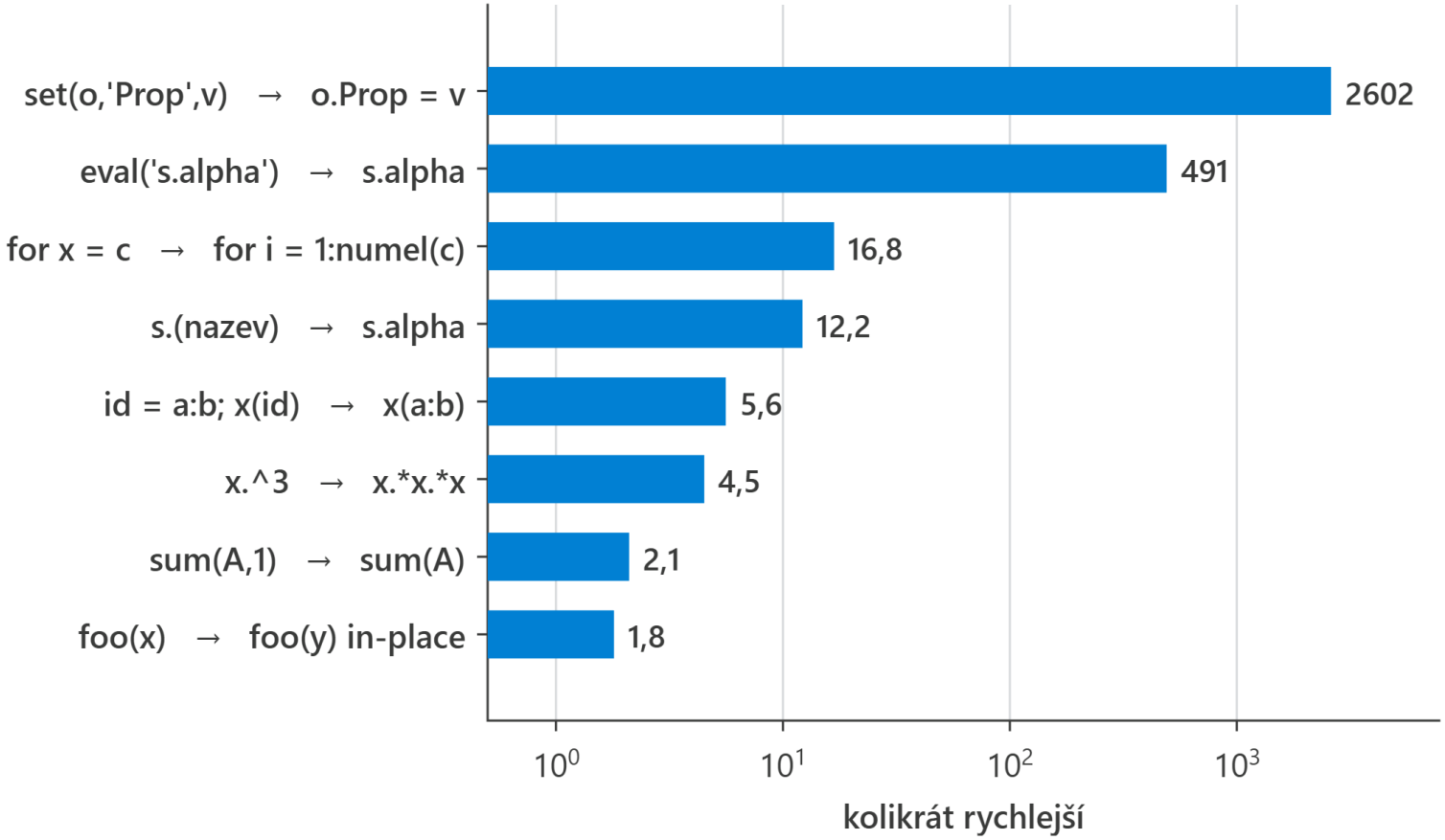
```
y = mexFunkce(x);
```



MEX rozhraní fyzikální knihovny, režie jednoho volání 9,7 µs

- pevná režie volání ≈ 10 µs, ať předáte cokoli
- optimalizujte počet volání, ne kód uvnitř
  - totéž platí pro Python, .NET i pro čtení souboru

Drobnosti, které se sčítají



- pevná režie na hranici volání, viditelná při mnoha voláních s málo daty
- neplatí univerzálně: x.^2 → x.*x nedá nic, mean(A,1) je dokonce rychlejší

Načítání dat ze souboru

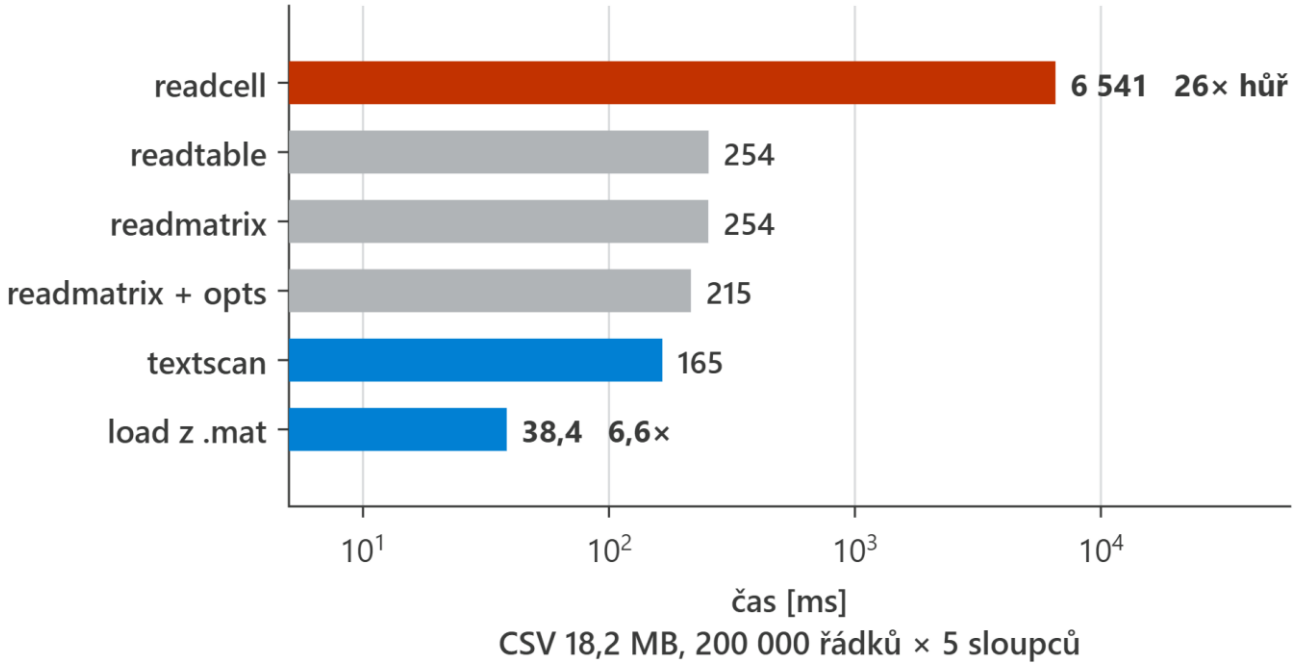
opakované čtení téhož

```
save("data.mat", "M");
load("data.mat", "M");
```

opakované čtení stejného formátu

```
opts = detectImportOptions("data.csv");
M = readmatrix("data.csv", opts);
```

- readcell platí alokaci za každou buňku



Shrnutí — v pořadí úsilí a zisku

- změřte
  - timeit, **Profiler**, ne odhad
- napište to správně
  - předalokace, vektorizace, průchod po sloupcích, žádné zbytečné kopie
- nechte práci vestavěným funkcím
  - jsou vícevláknové a napsané v C
- v objektech ubírejte
  - méně objektů, méně validace, data v maticích
- teprve pak parfor, GPU a MEX
  - největší úsilí, nejmenší jistota

Kdy přestat

- optimalizujte to, co je v profileru nahoře
- čitelnost je taky výkon, jen se projeví jinde
- po každé změně změřte znovu

- zajímavost pro pokročilé — novinka v R2026a
  - namespaceFunctions, namespaceClasses, innerNamespaces
  - dají se jimi najít interní funkce, třeba matlab.graphics.internal.drawnow.startUpdate
  - nedokumentované, mezi verzemi se mění bez náhrady

Děkuji za pozornost

Lubor Zháňal

zhanal@humusoft.cz

www.humusoft.cz