



HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK



Budapest,
2025.03.26.

SZÁMÍTÓGÉP ARCHITEKTÚRÁK

Lokalitástudatos programozás

Horváth Gábor, Belső Zoltán

BME Hálózati Rendszerek és Szolgáltatások Tanszék
ghorvath@hit.bme.hu, belso@hit.bme.hu

- Hogyan lehet **lassú** programot írni?
 - Címezzünk össze-vissza a memóriában!
 - Sok cache-hiba lesz
 - Sok TLB hiba lesz
 - Sok laphiba lesz
 - Sokszor kell új sort nyitni a DRAM-ban
- Hogy lehet **gyors** programot írni?
 - Dolgozzunk a tárhierarchia keze alá!
 - Memóriahivatkozásaink legyenek
 - Térben lokálisak
 - Időben lokálisak
- Tartalomjegyzék:
 1. Megéri egyáltalán?
 2. Ha igen, mit tehetünk C programozóként?

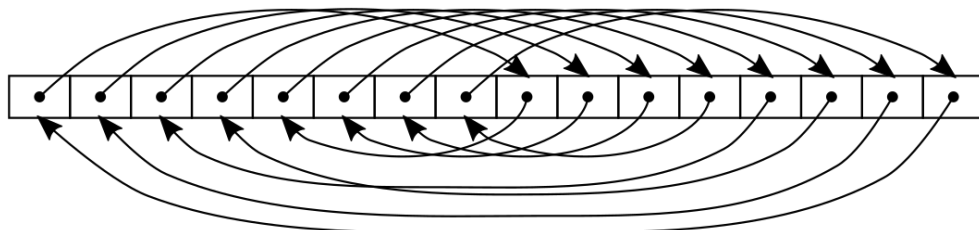


A lokális viselkedés hatásának számszerűsítése

Avagy: megéri ezzel a témával foglalkozni?

- Az időbeli lokalitás hatását
 - Mennyit számít, hogy a program minél hamarabb visszatér ugyanarra a memóriacímre?
- A térbeli lokalitás hatását
 - Mennyit számít, hogy a program a memóriát szépen sorban, szabályosan járja be?

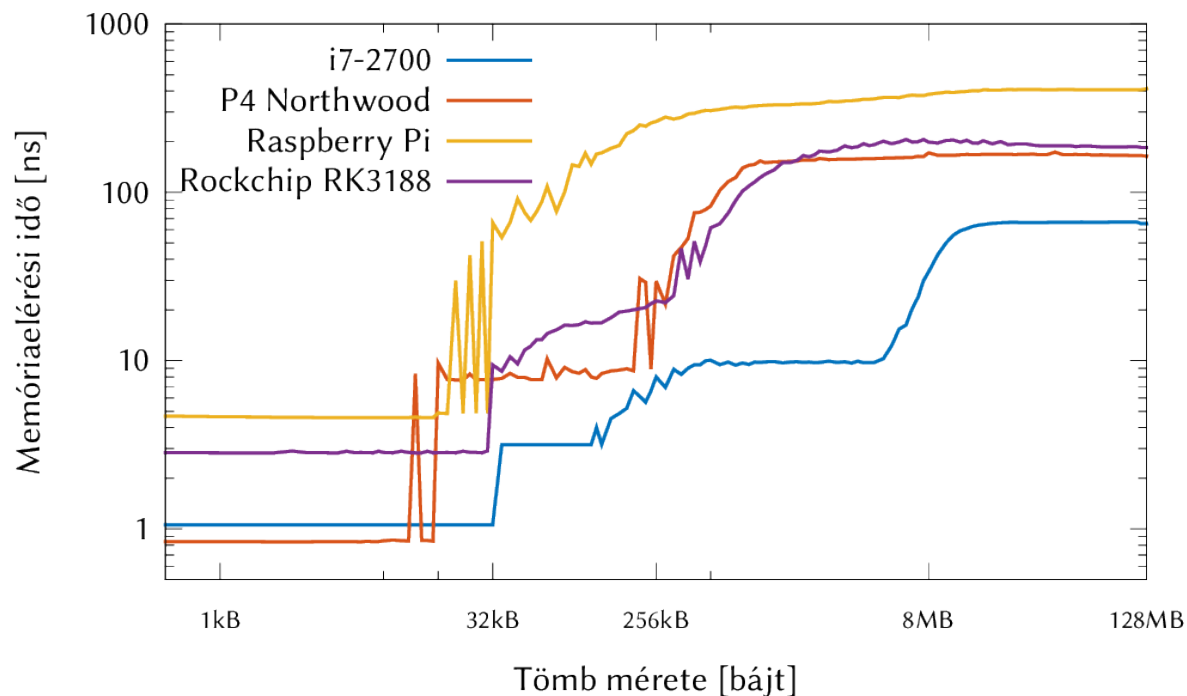
- A mérés módja:
 - Vegyünk egy tömböt (N)
 - Tömbelemek: pointerok a tömb más elemeire
 - A pointerlánc minden elemet érintsen, de legyen összevissza



- Mérjük a futási időt N függvényében

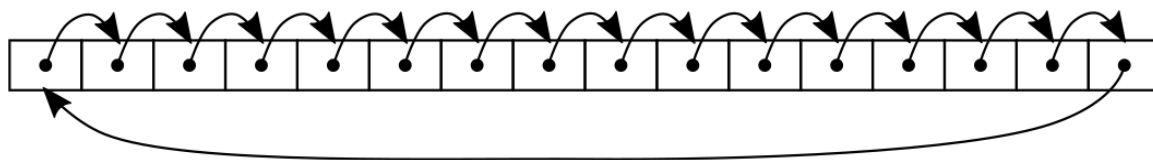
```
for (int i=0; i<iterations/100; i++) {  
    p = *p;  
    p = *p;  
    ...  
    p = *p;  
}
```

AZ IDŐBELI LOKALITÁS HATÁSÁNAK MÉRÉSE



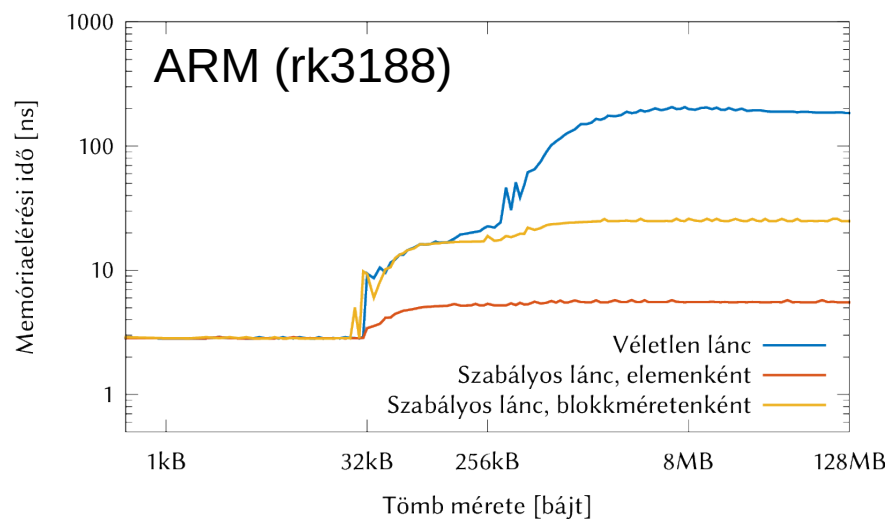
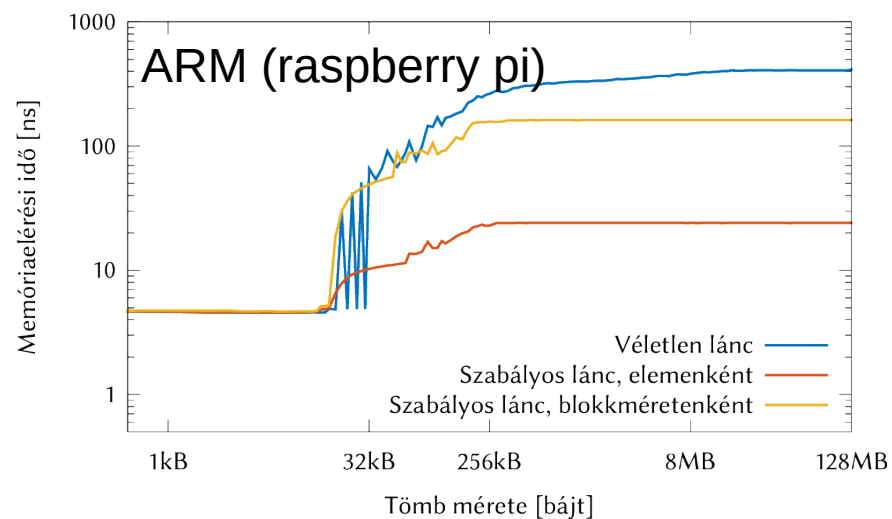
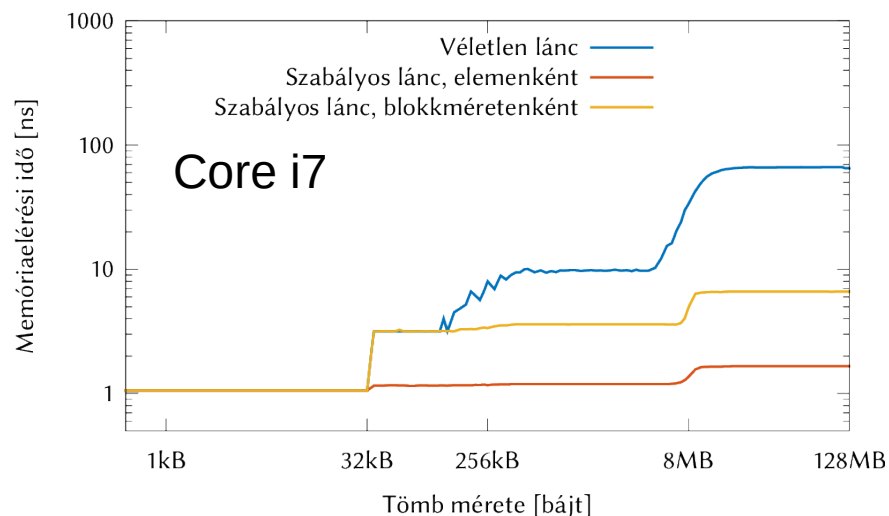
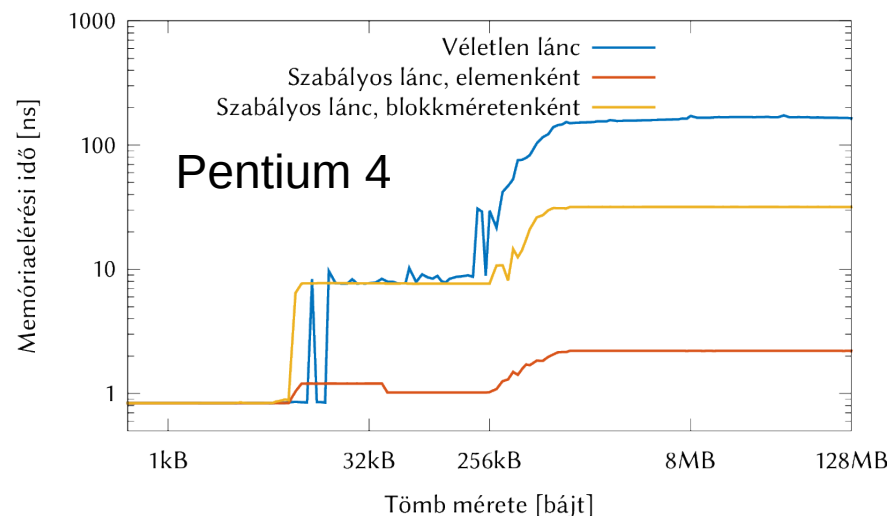
- Értékelés:
 - Behatárolható a tárhierarchia szintjeinek mérete!
- Üzenet:
 - Az időbeli lokalitás nagyon sokat számít
 - **Különbség akár 200-szoros!!!**

- A mérés módja
 - Mint eddig, de most láncoljuk az elemeket szépen sorba



- Várt eredmény: alacsonyabb késleltetés
 - Csökken a cache hiba-arány:
 - Ha egy cache blokkra léptünk, végigmegyünk a blokk többi elemén is
 - Ha van prefetch algoritmus, rá tud tanulni, előbetölti a blokkokat
 - Csökken a TLB hiba-arány:
 - Ha egy lapra léptünk, a lap végigjárásához ugyanaz a laptábla bejegyzés kell

A TÉRBELI LOKALITÁS HATÁSÁNAK MÉRÉSE



- Tanulság:
 - Kifizetődő a sorrendi bejárás
 - Nagy tömbmérettel kb. 40-80-szoros különbség



Lokalitásbarát ciklusszervezés

Avagy: mit tehet egy C programozó?

Eredeti C kód:

```
for (i=0; i<N; i++)  
    b[i] = c * a[i] + x;  
sum = 0;  
for (i=0; i<N; i++)  
    sum += b[i];  
for (i=0; i<N; i++)  
    d[i] = a[i] + b[i];
```

Ciklusegyesítéssel:

```
sum = 0;  
for (i=0; i<N; i++) {  
    b[i] = c * a[i] + x;  
    sum += b[i];  
    d[i] = a[i] + b[i];  
}
```

- Cache hiba-analízis: (felt.: 8 double fér egy blokkba, N nagy)
- Eredeti kód:
 - Első ciklus: $2N$ hivatkozás, $2N/8$ cache hiba
 - Második ciklus: N hivatkozás, $N/8$ hiba
 - Harmadik ciklus: $3N$ hivatkozás, $3N/8$ hiba
- Összesen: $6N$ hivatkozás, $6N/8$ cache hiba
→ **Cache hiba-arány: $1/8 = 12,5\%$**

Eredeti C kód:

```
for (i=0; i<N; i++)  
    b[i] = c * a[i] + x;  
sum = 0;  
for (i=0; i<N; i++)  
    sum += b[i];  
for (i=0; i<N; i++)  
    d[i] = a[i] + b[i];
```

Ciklusegyesítéssel:

```
sum = 0;  
for (i=0; i<N; i++) {  
    b[i] = c * a[i] + x;  
    sum += b[i];  
    d[i] = a[i] + b[i];  
}
```

- Ciklusegyesítéssel:
 - Ciklus első sora: $2N$ hivatkozás, $2N/8$ cache hiba
 - Második sor: N hivatkozás, 0 cache hiba!
 - Harmadik sor: $3N$ hivatkozás, $N/8$ hiba ($d[i]$ miatt)
- Összesen: $6N$ hivatkozás, $3N/8$ cache hiba
→ **Cache hiba-arány: $1/16 = 6,25\%$**

- Tanulság:
 - Kerüljük a tömbök ismételt bejárását
 - Amit lehet, vonjunk be közös ciklusba
- Mérési eredmények:
 - $N=2^{22}$

	i7-2600	p4	Rasp. Pi	RK3188
Eredeti algoritmus	16,533 ms	109,974 ms	698,450 ms	115,354 ms
Ciklusegyesítéssel	8,469 ms	84,917 ms	203,755 ms	97,126 ms

Bejárás sor-folytonosan:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        sum += a[i][j];
```

Bejárás oszlop-folytonosan:

```
for (j=0; j<N; j++)  
    for (i=0; i<N; i++)  
        sum += a[i][j];
```

- C nyelv: tömbök tárolása sor-folytonos
- Feltételezés: 8 double/cache blokk, N nagy
- Cache hiba-analízis:
- Sor-folytonos bejárás:
 - Memóriában folytonosan végigmegy a tömb minden elemén
 - Láttuk, hogy ez milyen előnyös
 - Cache hiba: minden 8. elem elérésekor
 - **Cache hiba-arány: $1/8 = 12,5\%$**

Bejárás sor-folytonosan:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        sum += a[i][j];
```

Bejárás oszlop-folytonosan:

```
for (j=0; j<N; j++)  
    for (i=0; i<N; i++)  
        sum += a[i][j];
```

- Oszlop-folytonos bejárás:
 - Minden lépésben N elemnyit ugrunk
 - Ha van cache prefetch, az rátanul, és előbetölt
 - Ha nincs, és $N > \text{cache méret}$:
 - Mire j-t lépteti, a blokk kiszorul a cache-ből
 - Minden elérés cache-hibával jár!
 - **Cache hiba-arány: 100%**

- Tanulság:
 - Bejárásakor igazodjunk az adatszerkezet memóriabeli elhelyezkedéséhez
- Mérési eredmények:
 - $N=2048$

	i7-2600	p4	Rasp. Pi	RK3188
Sor-folytonos	6,312 ms	8,973 ms	605,757 ms	14,879 ms
Oszlop-folytonos	6,926 ms	160,78 ms	4363,13 ms	60,96 ms

(az i7-nek jó cache prefetch algoritmus van)

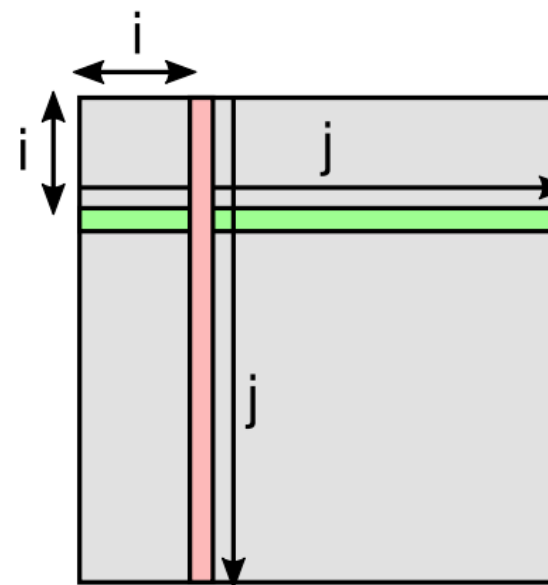
Eredeti C kód:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

Blokkos ciklusszervezéssel:

```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

- Mátrix transzponálás (kép forgatás, stb.)
- Feltételezés: 8 double/cache blokk, N nagy
- Cache hiba-analízis:
- Eredeti kód:
 - $a[i][j]$: sor-folytonos elérés
→ N^2 hivatkozás, $N^2/8$ cache hiba
 - $b[j][i]$: oszlop-folytonos elérés
→ N^2 hivatkozás, N^2 cache hiba
 - Összesen: $2N^2$ hivatkozás, $N^2/8 + N^2$ cache hiba
 - **Cache hiba-arány: $9/16 = 56,25\%$**



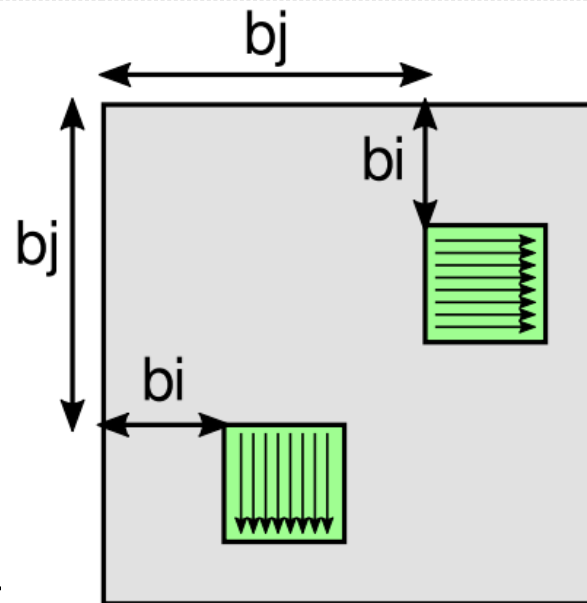
Eredeti C kód:

```
for (i=0; i<N; i++)
    for (j=0; j<N; j++)
        b[j][i] = a[i][j];
```

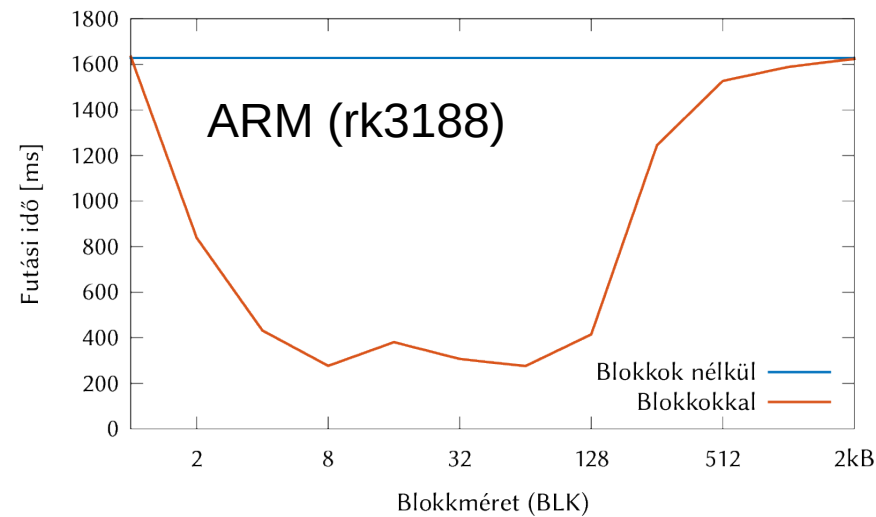
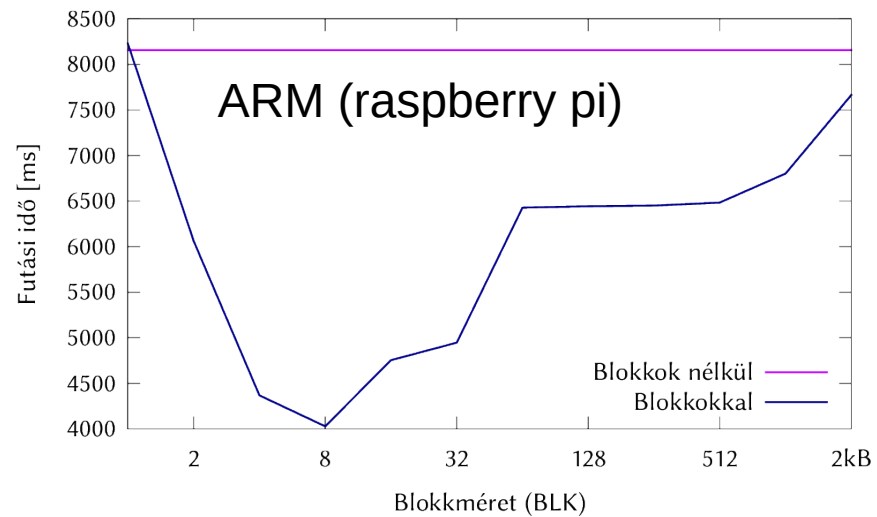
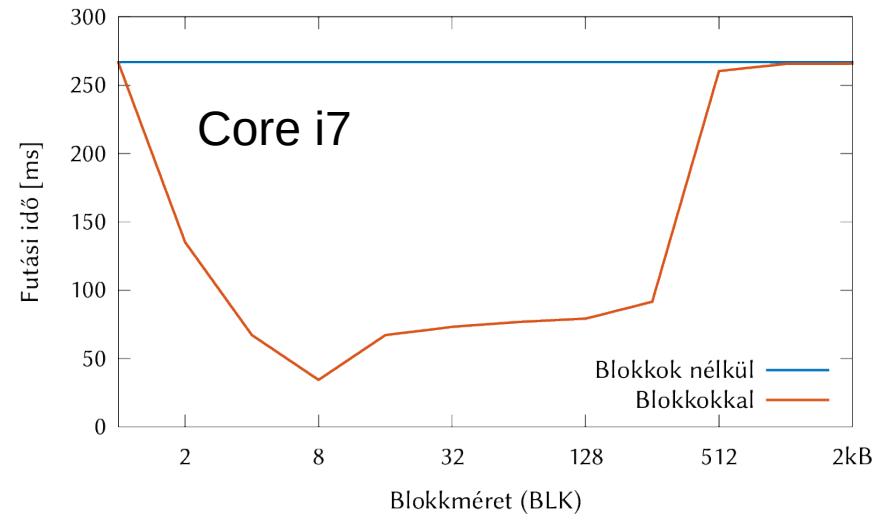
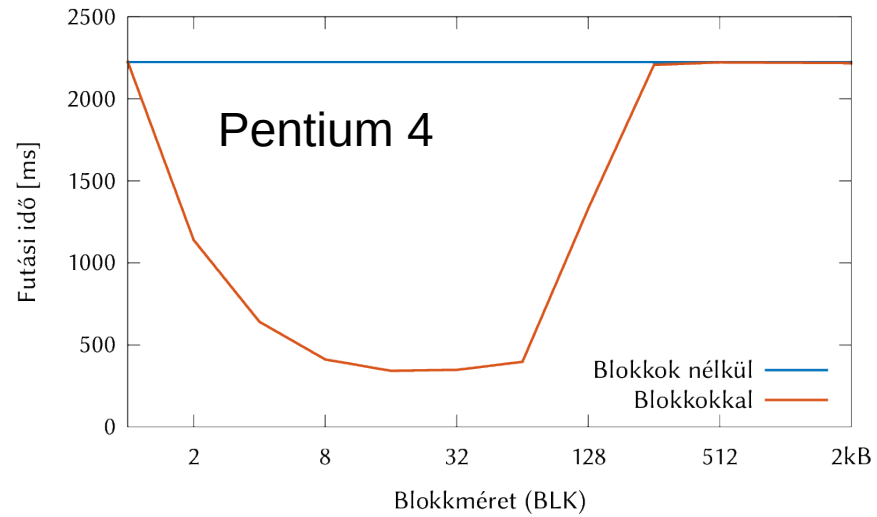
Blokkos ciklusszervezéssel:

```
for (bi=0; bi<=N-BLK; bi+=BLK)
    for (bj=0; bj<=N-BLK; bj+=BLK)
        for (i=bi; i<bi+BLK; i++)
            for (j=bj; j<bj+BLK; j++)
                b[j][i] = a[i][j];
```

- Blokkos ciklusszervezéssel:
 - Blokkonként haladunk
 - Ha jó a BLK, egy BLK x BLK méretű tömbdarab befér a cache-be
 - $a[i][j]$ és $b[j][i]$ is a cache memóriában lesz!
 - $a[i][j]$: sor-folytonos elérés, $\rightarrow N^2$ hivatkozás, $N^2/8$ cache hiba
 - $b[j][i]$: oszlop-folytonos elérés, $\rightarrow N^2$ hivatkozás, $N^2/8$ cache hiba
 - Összesen: $2N^2$ hivatkozás, $N^2/8 + N^2/8$ cache hiba
 - **Cache hiba-arány: $1/8 = 12,5\%$**



- Mi legyen a blokkméret?
 - Túl kicsi → olyan mintha nem lenne blokkszervezés
 - Túl nagy → olyan mintha nem lenne blokkszervezés
 - Architektúrafüggő!
- Mérési eredmények:
 - $N=2048$
 - BLK 1-től 2048





Esettanulmány: mátrixszorzás hatékonyan

- $C = A \cdot B$
 - $c_{ij} = \sum_k a_{ik} \cdot b_{kj}$
- Naiv algoritmus:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        for (k=0; k<N; k++)  
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- Kipróbáljuk a
 - ciklussorrend optimalizálását
 - blokkos ciklusszervezést

- (ijk) sorrend:

```
for (i=0; i<N; i++)  
    for (j=0; j<N; j++)  
        for (k=0; k<N; k++)  
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- $4N^3$ memóriaművelet:
 - $c[i][j]$ olvasás, $a[i][k]$ olvasás, $b[k][j]$ olvasás, $c[i][j]$ írás
- $c[i][j]$ olv.: sor-folytonos bejárás+belső ciklus ugyanazt az elemet éri el: $N^2/8$ cache hiba
- $a[i][k]$ olv.: sor-folytonos bejárás, minden sort N -szer: $N^3/8$ cache hiba
- $b[k][j]$ olv.: oszlop-folytonos bejárás N -szer: N^3 cache hiba
- $c[i][j]$ írás: soha nincs cache hiba, az olvasás behozta a cache-be
- Összegzés:
 - Aszimptotikus cache hiba-arány: $\lim_{N \rightarrow \infty} (N^2/8 + N^3/8 + N^3)/4N^3 =$
28,125%

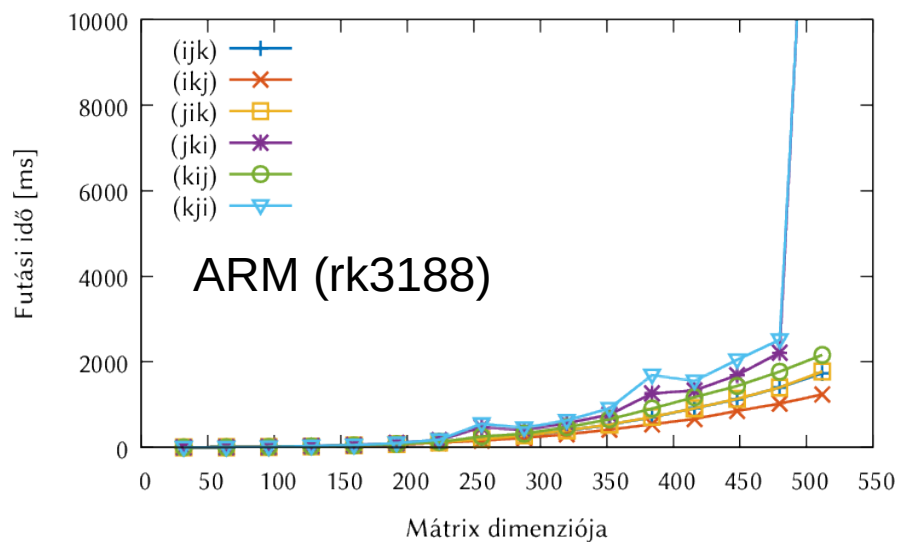
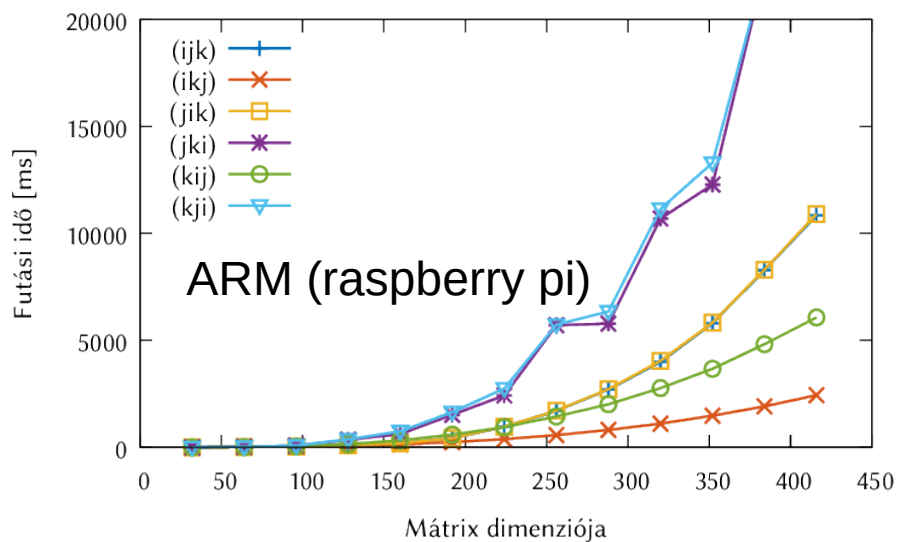
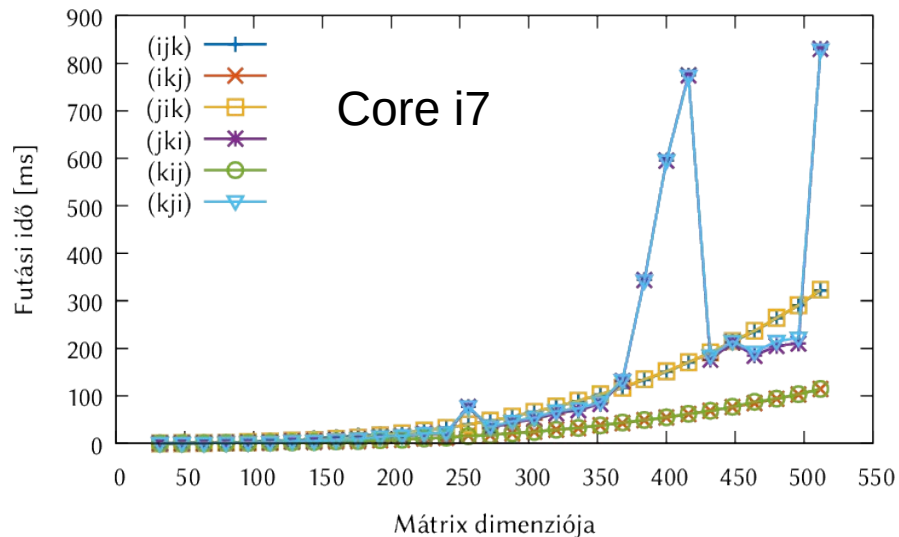
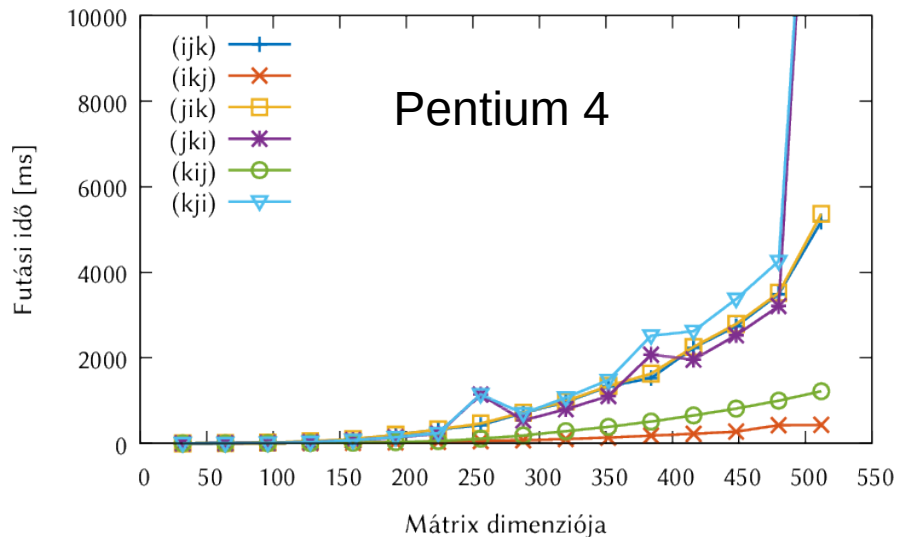
- (ikj) sorrend:

```
for (i=0; i<N; i++)  
    for (k=0; k<N; k++)  
        for (j=0; j<N; j++)  
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- $c[i][j]$ olv.: sor-folytonos bejárás, minden sort N -szer:
→ $N^3/8$ cache hiba
- $a[i][k]$ olv.: sor-folytonos bejárás+belső ciklus ugyanazt az elemet éri el: → $N^2/8$ cache hiba
- $b[k][j]$ olv.: sor-folytonos bejárás N -szer: → $N^3/8$ cache hiba
- $c[i][j]$ írás: soha nincs cache hiba, az olvasás behozta a cache-be
- Összegzés:
 - Aszimptotikus cache hiba-arány:
$$\lim_{N \rightarrow \infty} (N^3/8 + N^2/8 + N^3/8) / 4N^3 = \mathbf{6,25\%}$$

- 3 ciklus, 6-féle sorrend lehet
- Cache hiba-arányok:

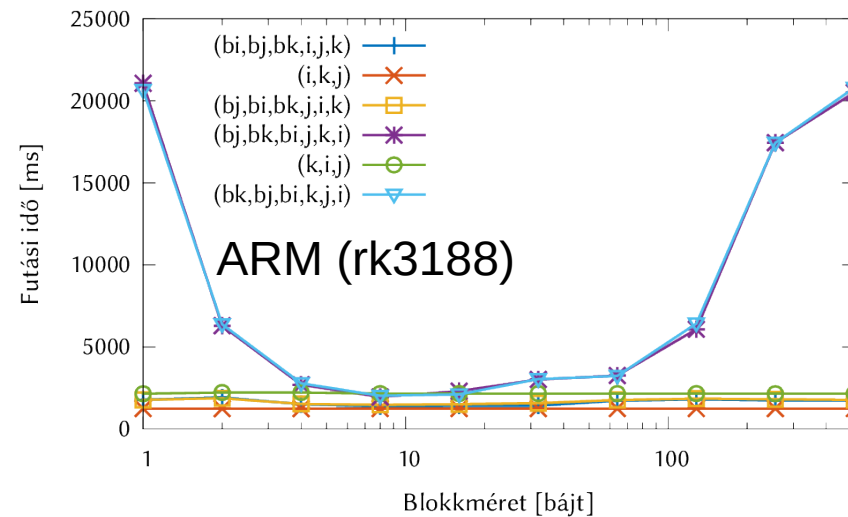
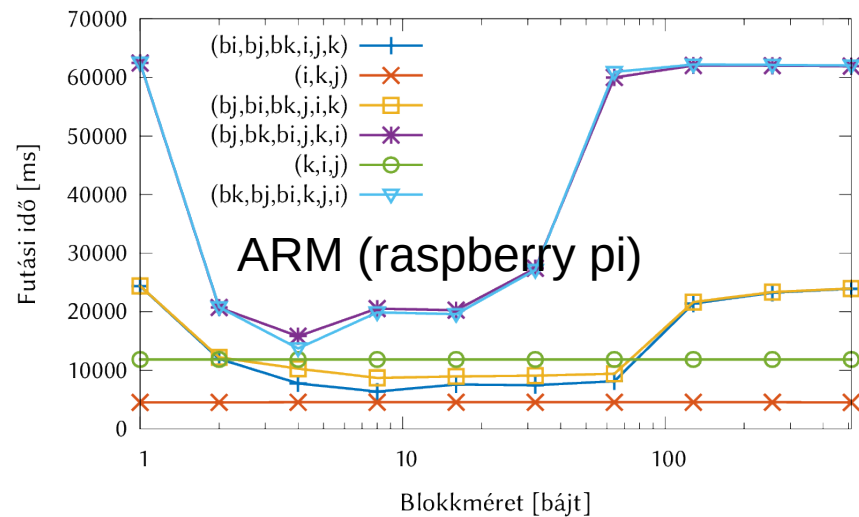
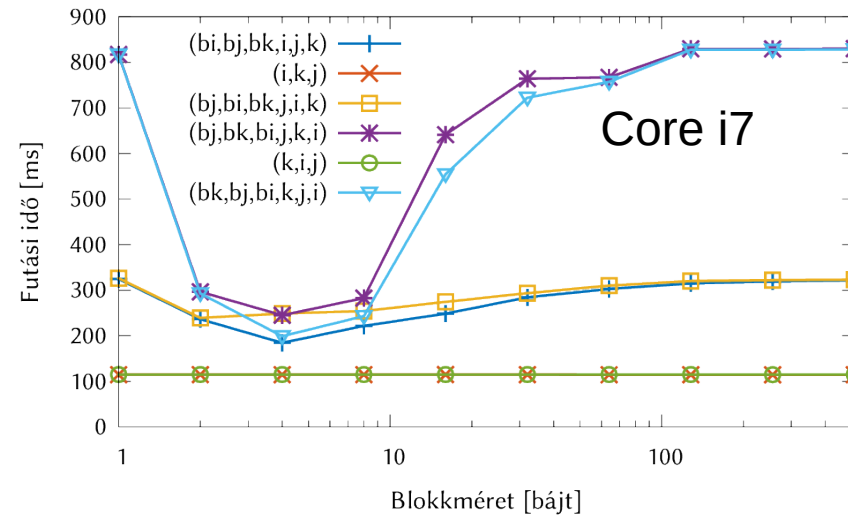
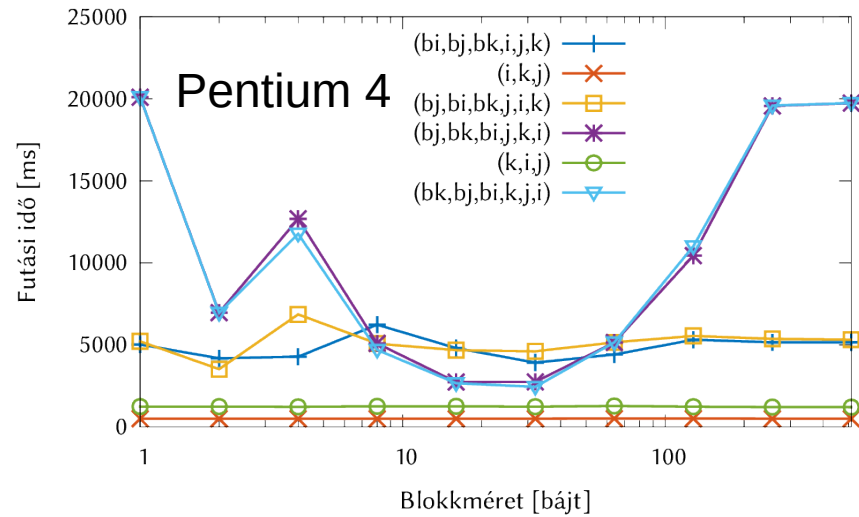
	$c[i][j]$ olv.	$a[i][k]$	$b[k][j]$	$c[i][j]$ ír.	Cache hiba-arány
(ijk)	$N^2/8$	$N^3/8$	N^3	0	28,125%
(ikj)	$N^3/8$	$N^2/8$	$N^3/8$	0	6,25%
(jik)	N^2	$N^3/8$	N^3	0	28,125%
(jki)	N^3	N^3	N^2	0	50%
(kij)	$N^3/8$	N^2	$N^3/8$	0	6,25%
(kji)	N^3	N^3	$N^2/8$	0	50%



- Mivel architektúrafüggő, csak a kísérletezés marad
- Lehet mindhárom ciklus blokkosított:

```
for (bi=0; bi<N; bi+=BLK)
  for (bj=0; bj<N; bj+=BLK)
    for (bk=0; bk<N; bk+=BLK)
      for (i=bi; i<bi+BLK; i++)
        for (j=bj; j<bj+BLK; j++)
          for (k=bk; k<bk+BLK; k++)
            c[i][j] = c[i][j] + a[i][k]*b[k][j];
```

- ... vagy csak kettő is.
- Mértünk
 - Az (ikj) és (kij) nem lett jobb a blokkoktól, a többi igen
 - A többenél az a legjobb, ha mindhárom ciklus blokkosított



- Konklúzió:
 - A naiv mátrixszorzó lassú
 - A blokkos szervezés sokat javít
 - De a legjobb ciklussorrend (ikj) anélkül is jobb
 - A legjobb ciklussorrend nem volt triviális, át kellett gondolni!

- Összegzés:
 - Sokszor lényegesen gyorsabb lesz a program, ha figyelünk a lokalitásra
 - Tanult technikák:
 - Ciklusegyesítés
 - Ciklussorrend optimalizálás
 - Blokkos ciklusszervezés
 - Több is van!
 - Vannak adatszerkezetek, melyek kifejezetten cache tudatosak
 - Rendezés: funnel sort
 - Mátrixműveletek: Morton reprezentáció
 - Stb.
- De!!!

Premature optimization is the root of all evil
(Donald Knuth)



HÁLÓZATI RENDSZEREK
ÉS SZOLGÁLTATÁSOK
TANSZÉK

