WIGNER GPU Lab

8. fejezet

Optimalizációk

Grafikus Processzorok Tudományos Célú Programozása

Berényi Dániel – Nagy-Egri Máté Ferenc

Optimalizáció – Mit? Mikor? Hogyan?

WIGNER GPU Lab

Optimalizálni valamilyen paramétereket lehet, hogy egy cél költségfüggvényt minimalizáljunk...

- Munkaidőre: minél tovább tartson
- Kódsorra: minél több sortörés
- Beágyazott rendszer: minél kevesebb memória
- Nagy-teljesítményű számolások: idő...

Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Optimalizálni valamilyen paramétereket lehet, hogy egy cél költségfüggvényt minimalizáljunk...

- Munkaidőre: minél tovább tartson
- Kódsorra: minél több sortörés
- Beágyazott rendszer: minél kevesebb memória
- Nagy-teljesítményű számolások: idő...

Optimalizáció – Mit? Mikor? Hogyan?

Idő…

Optimalizáció – Mit? Mikor? Hogyan?

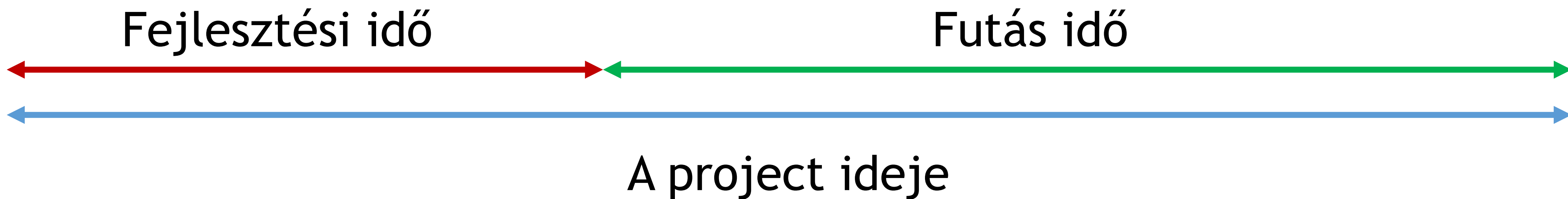
Idő...

A project ideje

Optimalizáció – Mit? Mikor? Hogyan?

WIGNER GPU Lab

Idő…

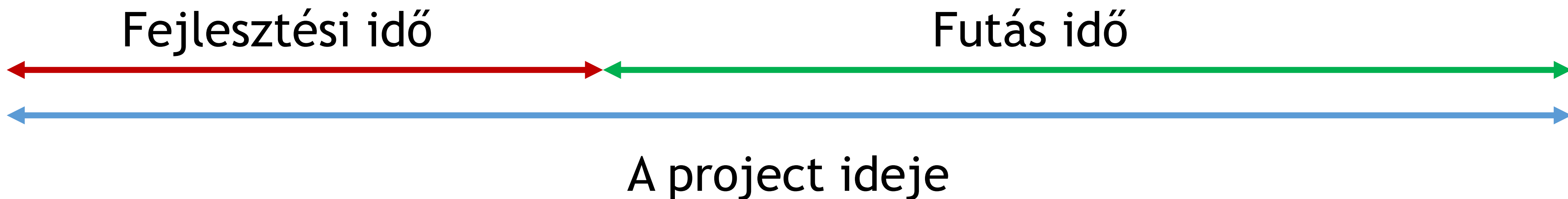


Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Idő…

Kérdés: hogy aránylik a kettő egymáshoz?



Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Időre optimalizálás, de melyikre?

Csak akkor érdemes belemerülni a végrehajtási idő optimalizálásába, ha:

- Várhatóan sokkal tovább fog futni, mint a fejlesztési idő
  (pl.: hetekig, hónapokig egy több száz magos klaszteren)
- Vagy amikor a gyorsaság (low latency) fontos
  (valós idejű adatfeldolgozás)

Optimalizáció – Mit? Mikor? Hogyan?

Akkor kezdjünk is neki!

Optimalizáció – Mit? Mikor? Hogyan?

Akkor kezdjünk is neki!

- Első lépés:

Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***

Optimalizáció – Mit? Mikor? Hogyan?

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***

Az idő előtti optimalizáció még a megosztott változóknál is rosszabb!!!

Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***
    - Ellenőrizzük a kód helyességét

Optimalizáció – Mit? Mikor? Hogyan?

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***
  - Ellenőrizzük a kód helyességét
    - a memória kezelés helyességét (allokáció, deallokáció)
    - a határeseteket (ciklusok, elágazások)
    - az inicializálatlan változókat (konstruktorok, lista-inicializáció)
    - kivétel kezelés, null pointerek
    - A lebegőpontos műveleteket (túlcsordulás, pontosságvesztés, INF, NAN)

Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***
  - Ellenőrizzük a kód helyességét...
    - Debug és nem-Debug (Release) módokban
    - 32 és 64 biten
    - különböző fordítókkal
    - kapcsoljuk be a figyelmeztetéseket (warnings)
    - futtassunk statikus kódanalízist

Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***
  - Ellenőrizzük a fordítási környezetet:
    - A környezeti változókat, amik befolyásolhatják a fordítást
    - A headereket és definíciókat, pláne ha több fordítási egység van
    - A linkelt külső könyvtárakat
(különös tekintettel a verzióra és 32-64 bit szélességre)
    - Ha egy másik nyelvvel interfészelünk, akkor az argumentum átadást (sorrend, érték vagy pointer szerinti átadás)

Optimalizáció – Mit? Mikor? Hogyan?

Wigner GPU Lab

Akkor kezdjünk is neki!

- Első lépés: ***Ne optimalizáljunk!***
  - Ellenőrizzük a könyvtárak helyes használatát!!!

EGYSZER az életben olvassuk el, hogy hogyan kell helyesen használni az adott könyvtárat, vagy programozási interfészt (API-t)

Optimalizáció – Mit? Mikor? Hogyan?

Akkor kezdjünk is neki!

- Második lépés:

Optimalizáció – Mit? Mikor? Hogyan?

WIGNER GPU Lab

Akkor kezdjünk is neki!

- Második lépés:

  Ismételjük az első lépést addig,
  ameddig biztosak nem vagyunk, hogy minden korrekt!

Optimalizáció – Mit? Mikor? Hogyan?

Akkor kezdjünk is neki!

- Harmadik lépés:
  - Készítsünk teszteket, amik a program helyes működését ellenőrzik
    (pl.: adott bemenetekre ismert kimeneteket produkálnak)
  - Készítsünk a működő kódról biztonsági mentést!!!
    - Azért, hogy a későbbi változatokkal összehasonlítható legyen

Optimalizáció – Mit? Mikor? Hogyan?

WIGNER GPU Lab

Akkor kezdjünk is neki!

- Harmadik lépés:
  - Készítsünk teszteket, amik a program helyes működését ellenőrzik
    (pl.: adott bemenetekre ismert kimeneteket produkálnak)
  - Készítsünk a működő kódról biztonsági mentést!!!
    - Azért, hogy a későbbi változatokkal összehasonlítható legyen

    NEM (csak) a sebesség szempontjából, hanem hogy továbbra is korrekten működnek-e!

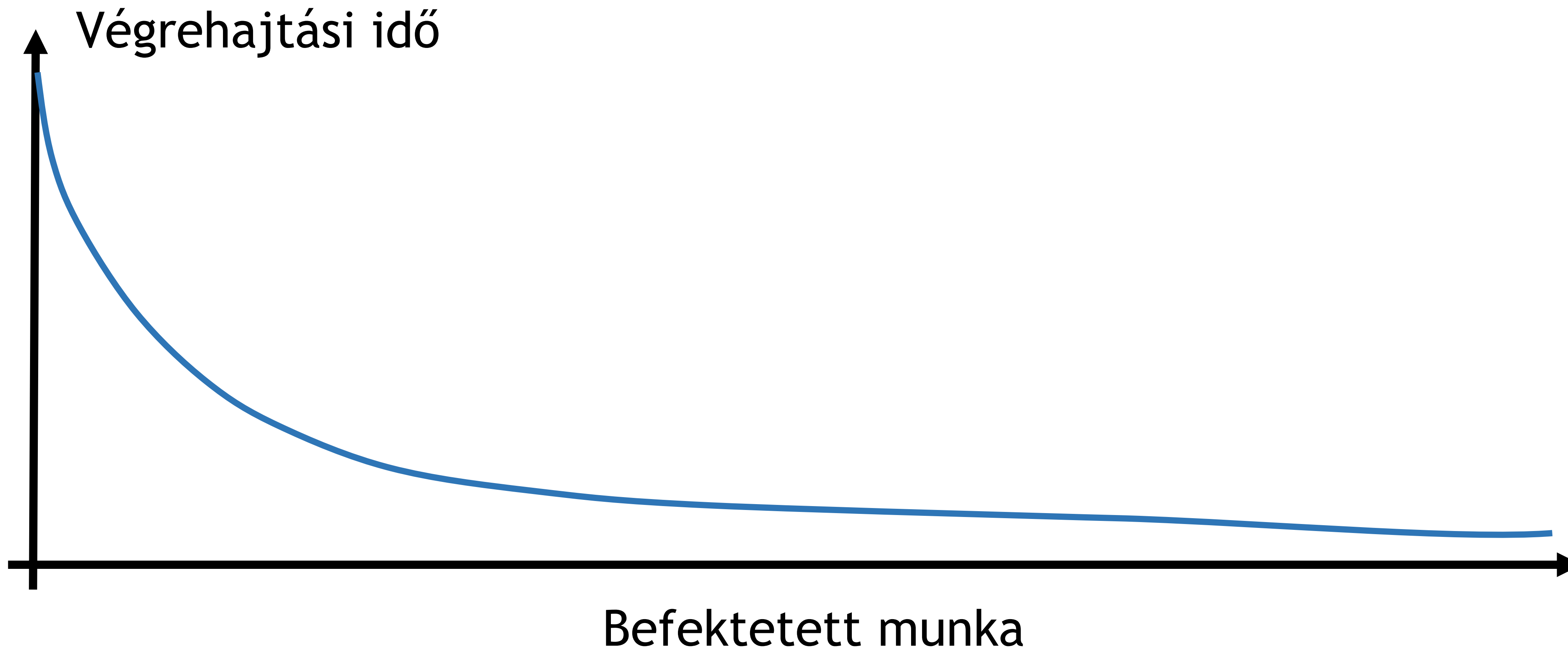
Optimalizáció – Mit? Mikor? Hogyan?

Tipikus optimalizációs görbe:

Optimalizáció – Mit? Mikor? Hogyan?

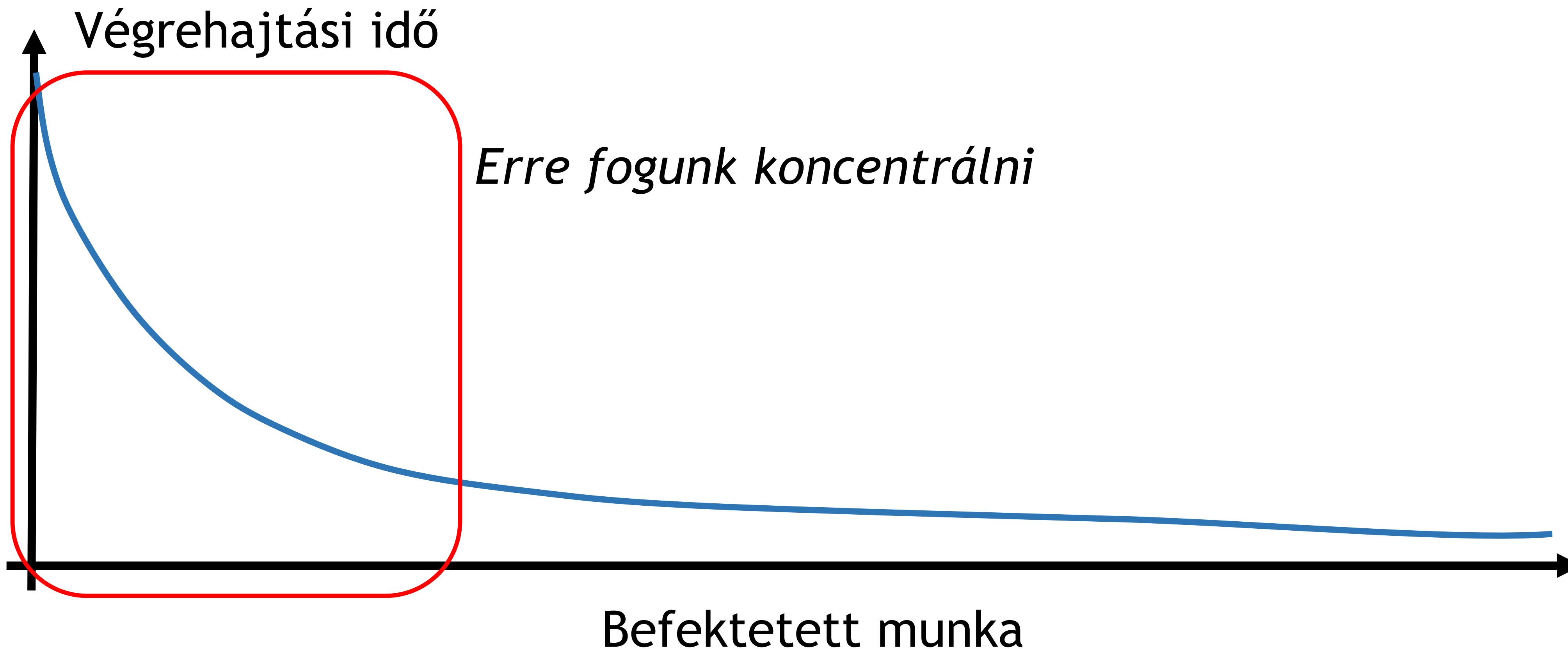
Wigner GPU Lab

Tipikus optimalizációs görbe:



Optimalizáció – Mit? Mikor? Hogyan?

Tipikus optimalizációs görbe:



Időmérés

Milyen gyors a kód?

Időmérés

Milyen gyors a kód?

Végre van sztenderd módszer az időmérésre, használjuk!

Időmérés

```
#include <chrono>
auto tmark()
{
        return std::chrono::high_resolution_clock::now();
}

template<typename T1, typename T2>
auto delta_time( T1&& t1, T2&& t2 )
{
    return
std::chrono::duration_cast<std::chrono::nanoseconds>(t2-t1)
                                            .count()/1000.0;
}
```

Időmérés

```
auto t0 = tmark();

//a kód, amit szeretnénk mérni

auto t1 = tmark();

std::cout << "My calculation took: "
          << delta_time(t0, t1) << " usecs.\n";
```

Időmérés

A mért idők értelmezése azonban trükkös feladat...

- A rendszer hívások idejei sok tényezőtől függhetnek, például a rendszer terheltségétől
- Az időmérés maga is egy rendszer hívás

Időmérés

A mért idők értelmezése azonban trükkös feladat...

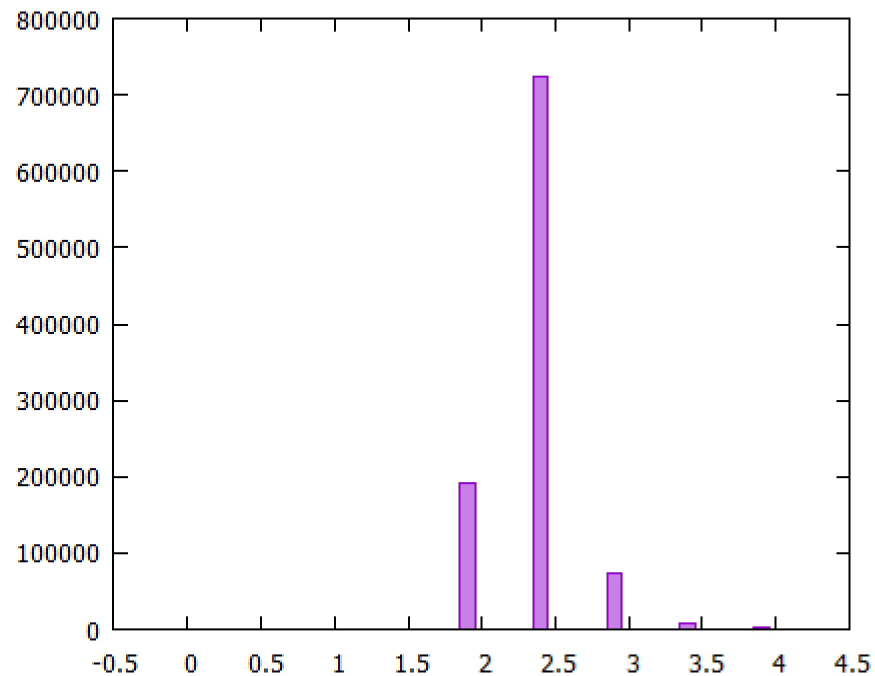
- A rendszer hívások idejei sok tényezőtől függhetnek, például a rendszer terheltségétől
- Az időmérés maga is egy rendszer hívás

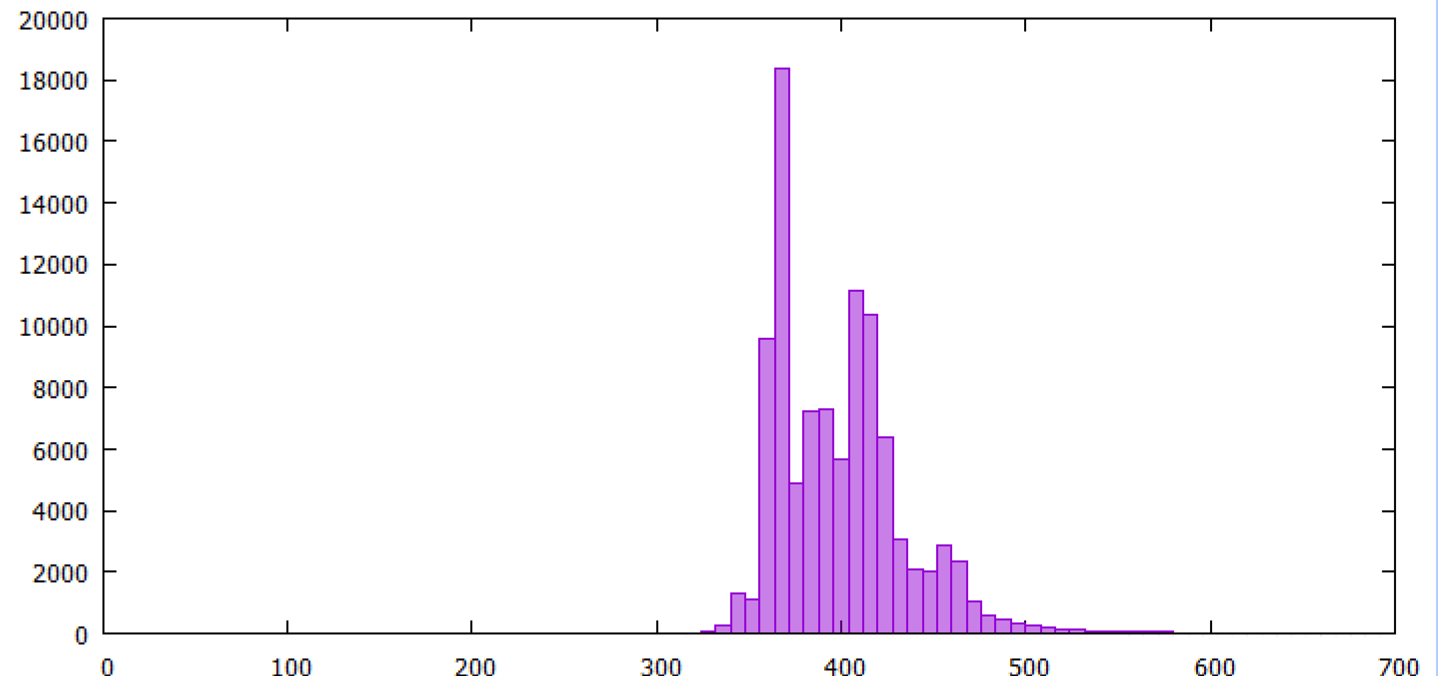
- A szálkezelő félre teheti a szálat, de a mért időket nem korrigálja
- A szálkezelő átteheti egyik magról a másikra a szálat
  a mért időkben kitör a káosz

- Maga a hardveres óra is csinálhat fura dolgokat, főleg többmagos rendszereken

Időmérés

A mért idők eloszlása közel sem triviális:





Trigonometrikus összeg (50 tag) [μs]

Memória allokáció (300 kB) [μs]

Időmérés

Tehát:

- Érdemes többször (10x-100x) lemérni a kódot
  (lehetőleg terheletlen rendszeren)

- Ha összehasonlítást akarunk tenni, akkor venni a minimumot:
  - mert a hardveres és szoftveres megszakítások ideális esetben csak növelik a futás időt

- Ezért a minimum robosztusabb, és jobban jellemzi a kód hatékonyságát, mint az átlag.

Optimalizációk

Legegyszerűbb optimalizációk:

Optimalizációk

Legegyszerűbb optimalizációk:

Ismerjük meg a fordítót!

Emlékeztetőül a párhuzamosság szintjei:

- Bit szint
- Utasítás szint
- Vector szint
- Feladat/Eszköz szint
- Folyamat/Klaszter szint

Ezekért a *fordító* és a processzor felel! (Bit, Utasítás, Vector szint)

Optimalizációk

WIGNER GPU Lab

Legegyszerűbb optimalizációk:

- Fordító optimalizációk! Csak egy kapcsolón múlik...
  - Kapcsoljuk be őket (-O3)
  - Gyors lebegőpontos műveletek és kevésbé szigorú átalakítások (azt állítjuk ekkor, hogy nem lesznek NaN-ok Inf-ek, -fast-math)
  - Engedélyezzük a kiterjesztett utasításkészleteket (SSE, AVX, ...)
  - Engedélyezzük / hangoljuk a vektorizációt és a ciklus optimalizációt

Nézzük meg a fordító leírását!!!

Optimalizációk

Legegyszerűbb optimalizációk:

- Fordító optimalizációk! Csak egy kapcsolón múlik...
  - Kapcsoljuk be őket (-O3)
  - Gyors lebegőpontos műveletek és kevésbé szigorú átalakítások (azt állítjuk ekkor, hogy nem lesznek NaN-ok Inf-ek, -fast-math)
  - Engedélyezzük a kiterjesztett utasításkészleteket (SSE, AVX, ...)
  - Engedélyezzük / hangoljuk a vektorizációt és a ciklus optimalizációt

Nézzük meg a fordító leírását!!!

És mindegyik bekapcsolása után ellenőrizzük a program helyes működését!!!

Optimalizációk

Wigner GPU Lab

Kevésbé egyszerű optimalizációk (itt már gondolkozni is kell☺):

Optimalizációk

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

Optimalizációk

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja

Optimalizációk - formulák

WIGNER GPU Lab

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

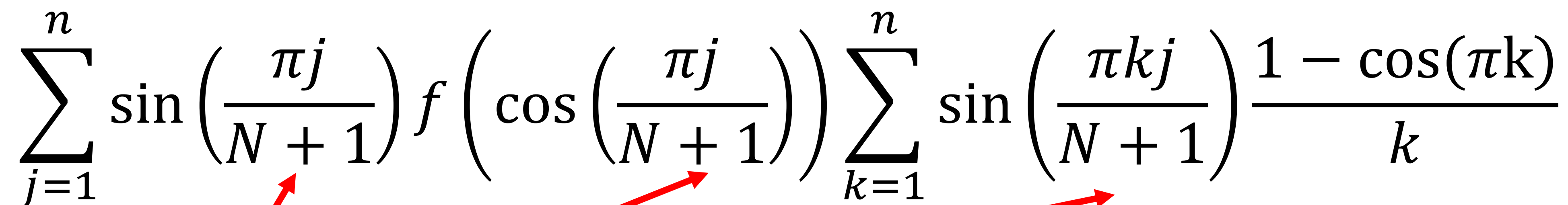
- A felírt formulák konkrét alakja – Példa: a matematika ezt dobta:

$$\sum_{j=1}^{n} \sin\left(\frac{\pi j}{N+1}\right) f\left(\cos\left(\frac{\pi j}{N+1}\right)\right) \sum_{k=1}^{n} \sin\left(\frac{\pi k j}{N+1}\right) \frac{1-\cos(\pi \mathrm{k})}{k}$$

Optimalizációk - formulák

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja – Példa: a matematika ezt dobta:



$$\sum_{j=1}^{n} \sin\left(\frac{\pi j}{N+1}\right) f\left(\cos\left(\frac{\pi j}{N+1}\right)\right) \sum_{k=1}^{n} \sin\left(\frac{\pi k j}{N+1}\right) \frac{1-\cos(\pi \mathrm{k})}{k}$$

Ugyan az a faktor előbukkan 3x, emeljük ki!

A fordító lehet, hogy felismeri, de általános esetben erre nem alapozhatunk

Optimalizációk - formulák

WIGNER GPU Lab

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja – Példa: a matematika ezt dobta:

Legyen $q_j = \frac{\pi j}{N+1}$

$$\sum_{j=1}^{n} \sin(q_j)\, f\left(\cos(q_j)\right) \sum_{k=1}^{n} \sin(k q_j) \frac{1-\cos(\pi k)}{k}$$

Optimalizációk - formulák

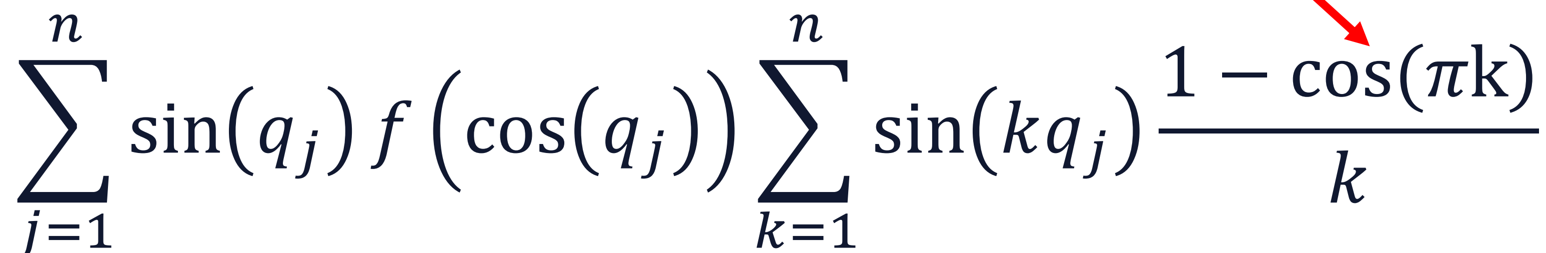
A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja – Példa: a matematika ezt dobta:

Emlékezzünk, a cos költséges művelet, és itt csak π egész többszöröseinél van kiértékelve

Legyen $q_j = \frac{\pi j}{N+1}$

$\frac{1-\cos(\pi k)}{k}$ = (k%2==1 ? 2.0/k : 0.0)



$$\sum_{j=1}^{n} \sin(q_j) f\left(\cos(q_j)\right) \sum_{k=1}^{n} \sin(kq_j) \frac{1-\cos(\pi k)}{k}$$

Optimalizációk

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja
- A memória használat

Optimalizációk

Wigner GPU Lab

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja
- A memória használat
- A memória használat

Optimalizációk

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja
- A memória használat
- A memória használat
- A memória használat
- A memória használat

Optimalizációk

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja
- A memória használat
- A memória használat
- A memória használat
- A memória használat
- A memória használat
- A memória használat

Optimalizációk

A nagy számításigényű (tudományos) kódok teljesítményét legjobban a következő faktorok befolyásolják:

- A felírt formulák konkrét alakja
- A memória használat
- A memória használat
- A memória használat
- A memória használat
- A memória használat
- A memória használat
- A memória használat

A memória használat
A memória használat
A memória használat
A memória használat
A memória használat
A memória használat
A memória használat
A memória használat
A memória használat

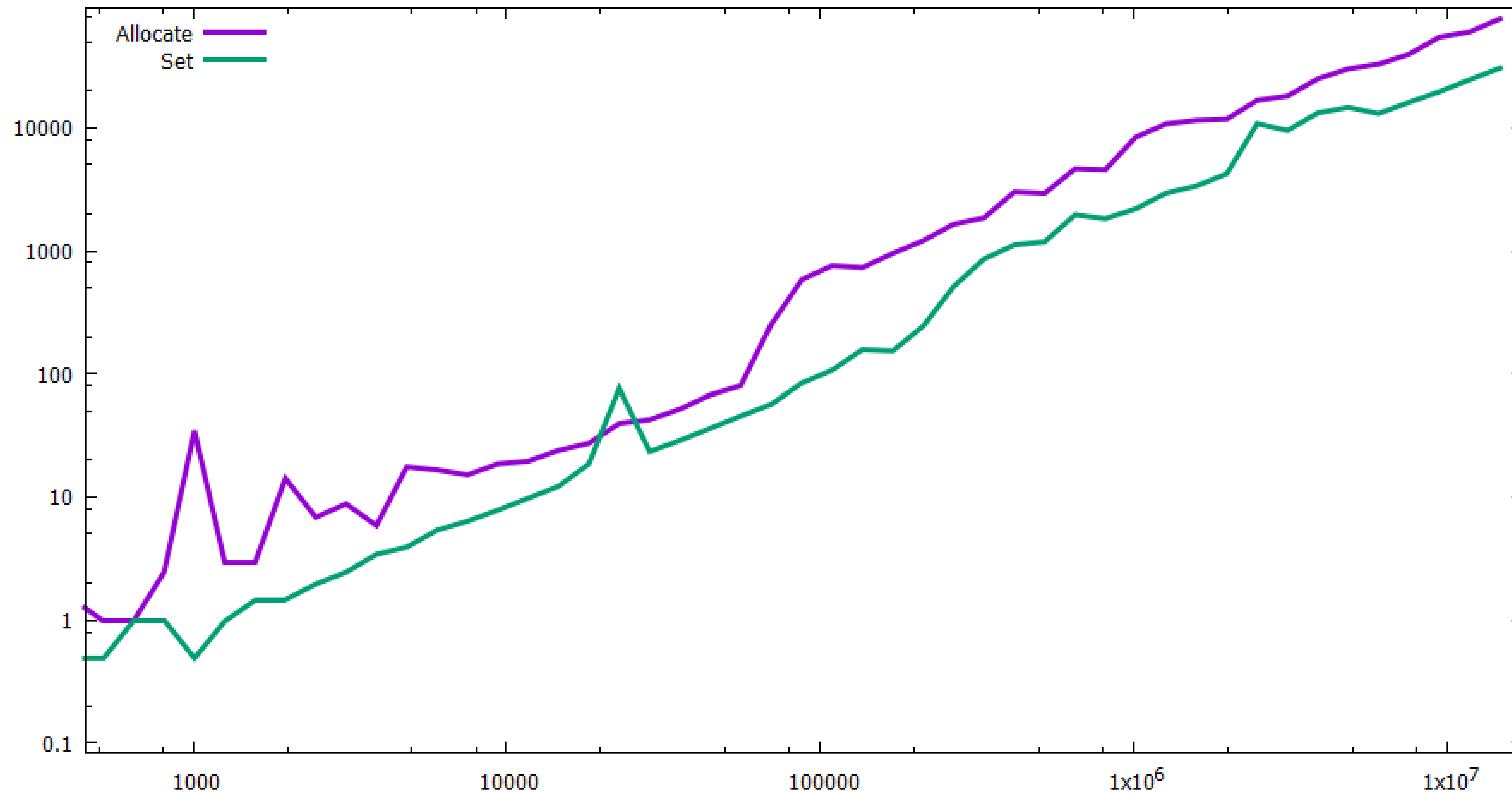
Optimalizációk - memória

Az optimalizációk jelentős része (mondjuk 80%-a) az adathozzáféréssel és a memória elrendezéssel kapcsolatos

Ennek megfelelően a teljesítmény problémák 80%-a is ezekből ered.

Optimalizációk - memória

Például: a dinamikus allokáció drága dolog!



Optimalizációk - memória

Például: a dinamikus allokáció drága dolog!
Mégis gyakran látni ilyeneket:

```
for(int i=0; i<N; ++i)
{









}
```

Optimalizációk - memória

Például: a dinamikus allokáció drága dolog!
Mégis gyakran látni ilyeneket:

```
for(int i=0; i<N; ++i)
{
      for(int j=0; j<N; ++j)
      {




      }
}
```

Optimalizációk - memória

Például: a dinamikus allokáció drága dolog!
Mégis gyakran látni ilyeneket:

```
for(int i=0; i<N; ++i)
{
    for(int j=0; j<N; ++j)
    {
        for(int k=0; k<N; ++k)
        {

        }
    }
}
```

Wigner GPU Lab

Optimalizációk - memória

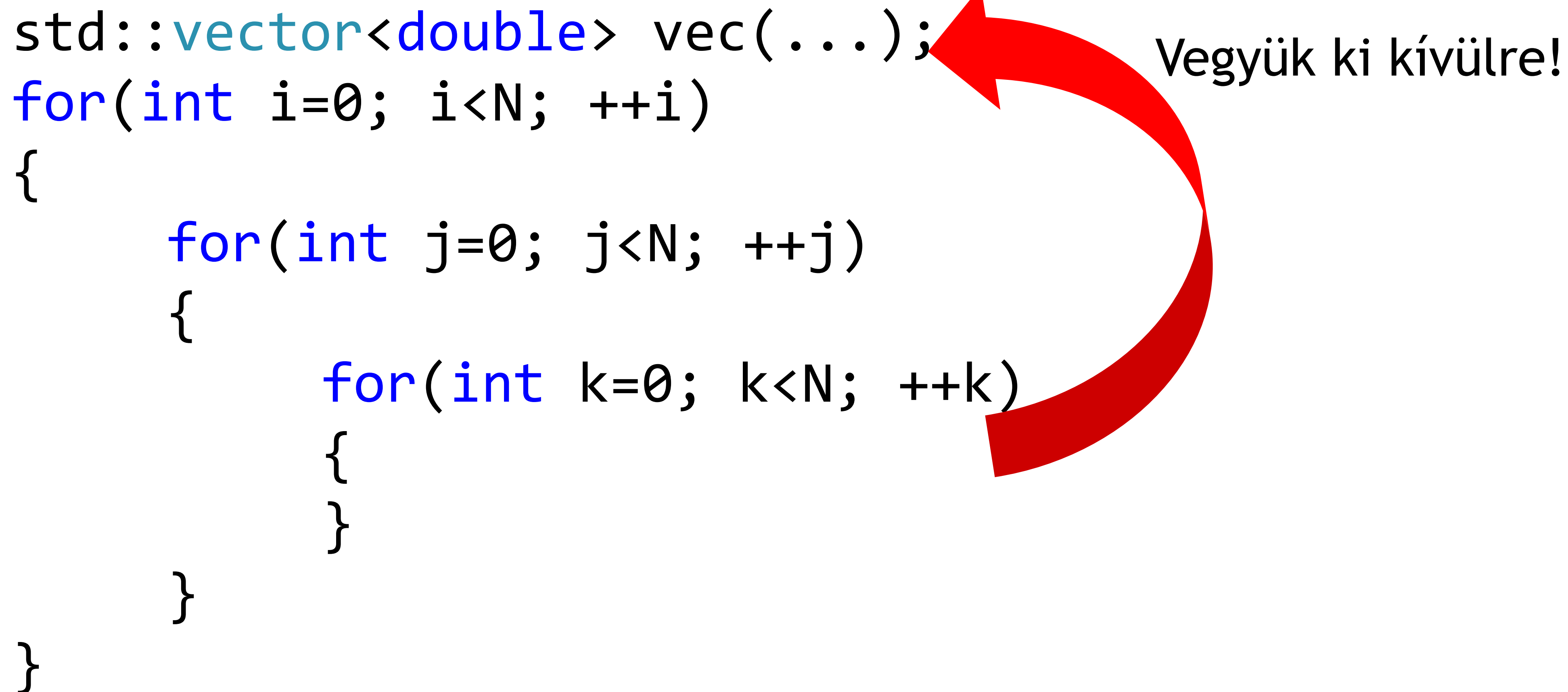
Például: a dinamikus allokáció drága dolog!
Mégis gyakran látni ilyeneket:

```
for(int i=0; i<N; ++i)
{
        for(int j=0; j<N; ++j)
        {
                for(int k=0; k<N; ++k)
                {
                        std::vector<double> vec(...);
                }
        }
}
```

Optimalizációk - memória

Például: a dinamikus allokáció drága dolog!

```
std::vector<double> vec(...);
for(int i=0; i<N; ++i)
{
	for(int j=0; j<N; ++j)
	{
		for(int k=0; k<N; ++k)
		{
		}
	}
}
```



Optimalizációk - memória

A legtöbb C++ tároló alapból dinamikus memória allokátorokat használ:

- `std::vector, std::list, std::set, ...`

Az egyetlen tároló, ami nem ilyen:

- `std::array<T, n>`

Optimalizációk - memória

Ha mégis muszáj dinamikusan allokálni, akkor:

- Ezt tegyük meg a komplex számolások elején / előtt
- Ha nem tudjuk pontosan mennyi kell, próbáljuk meg megbecsülni és lefoglalni

  `std::vector::reserve(size_t)` függvény!

Optimalizációk - memória

Wigner GPU Lab

Gyakran használt átméretezhető tárolók:

```
std::vector, std::list
```

Hasonlítsuk őket össze!

Optimalizációk - memória

Gyakran használt átméretezhető tárolók:

`std::vector`, `std::list`

Hasonlítsuk őket össze!

Sorozatos push_back:

#double	std::vector [ms]	std::list [ms]
10k	0.3	1.1
1M	32	90
10M	300	920
100M	2700	9300

Optimalizációk - memória

Gyakran használt átméretezhető tárolók:

`std::vector`, `std::list`

Hasonlítsuk őket össze!

Sorozatos push_back:

#double	std::vector [ms]	std::list [ms]	Egyszeri allokáció [ms]
10k	0.3	1.1	0.05
1M	32	90	5
10M	300	920	50
100M	2700	9300	500

Optimalizációk - memória

Wigner GPU Lab

Gyakran használt átméretezhető tárolók:

`std::vector`, `std::list`

Mások eredményeit is alapul véve (link, link), arra juthatunk, hogy:

Az egyetlen eset, amikor az `std::list` gyorsabb, ha nagy mennyiségű adatot kell az adatsor elejére vagy közepére beszúrni.

Optimalizációk - memória

Ha mindenképp az allokáció sebessége a limitáló tényező:

- Analizáljuk az allokált méretek gyakoriságát
- Ennek a statisztikának megfelelően írjunk saját memória kezelőt
- Terheljük túl a `new`/`delete` operátorokat, hogy a saját memória kezelőnket hívják és ne a `malloc`-ot.

Vagy:

- Használjunk előre megírt memória kezelőket (tcmalloc, jemalloc)
- Vannak más eszközök is, amik képesek átirányítani a C könyvtárhívásait és sokkal hatékonyabbak gyakori kis allokációkra, mint a `malloc`.

Optimalizációk - memória

Adathozzáférés!

Optimalizációk - memória

Wigner GPU Lab

Adathozzáférés!

Szekvenciális (folytonos)
vagy kihagyásos (pl. minden n-dik elem)

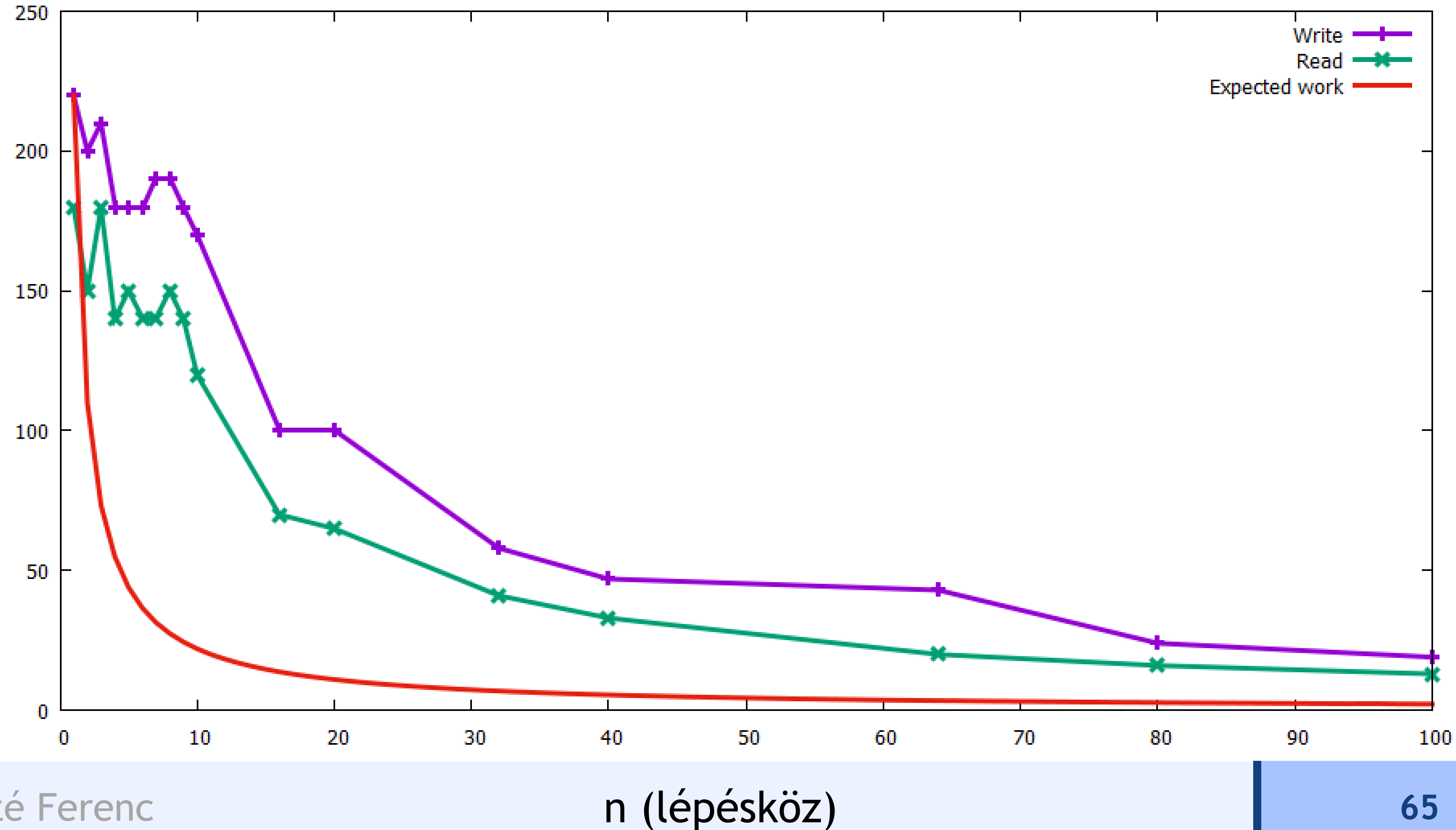
Optimalizációk - memória

Adathozzáférés!

Szekvenciális
folytonos

vs

Kihagyásos
csak n-dik elemek



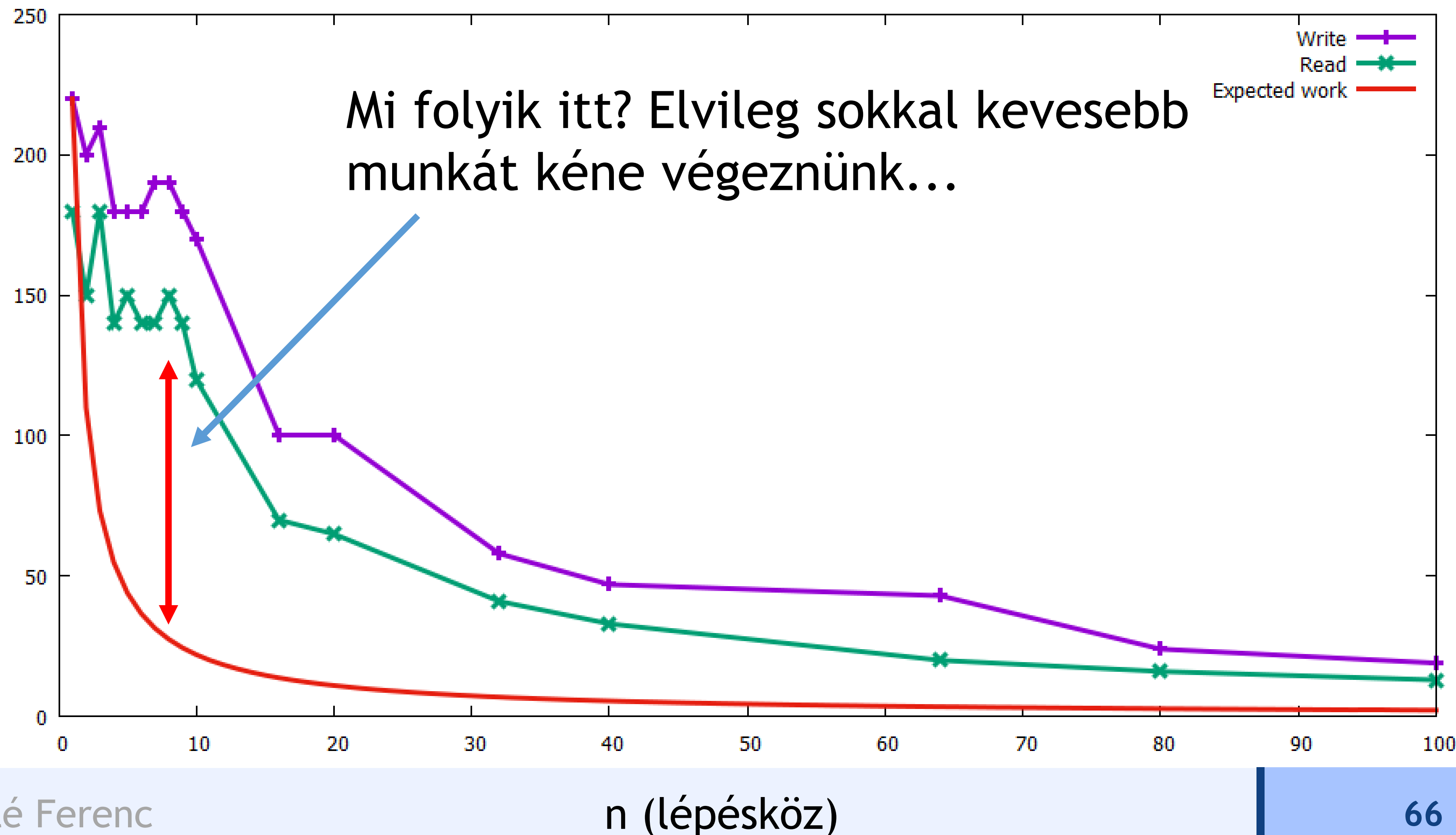
Optimalizációk - memória

Adathozzáférés!

Szekvenciális
folytonos

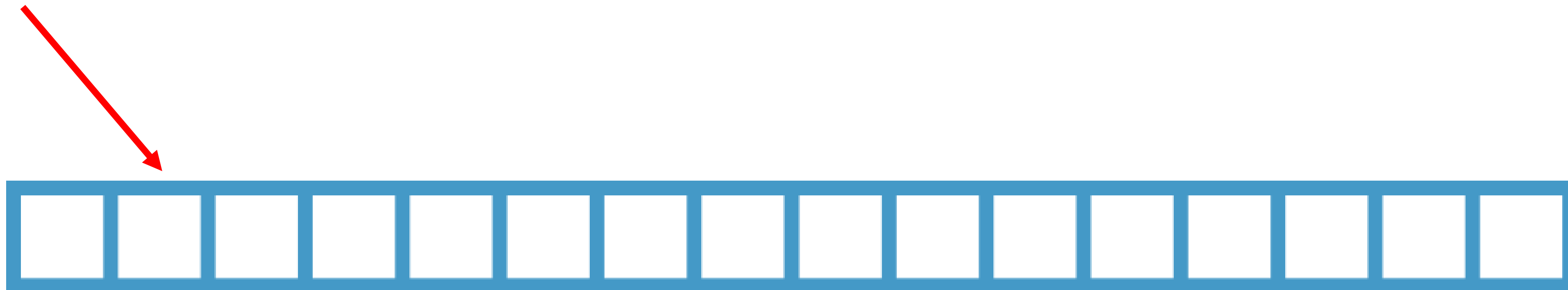
vs

Kihagyásos
csak n-dik elemek



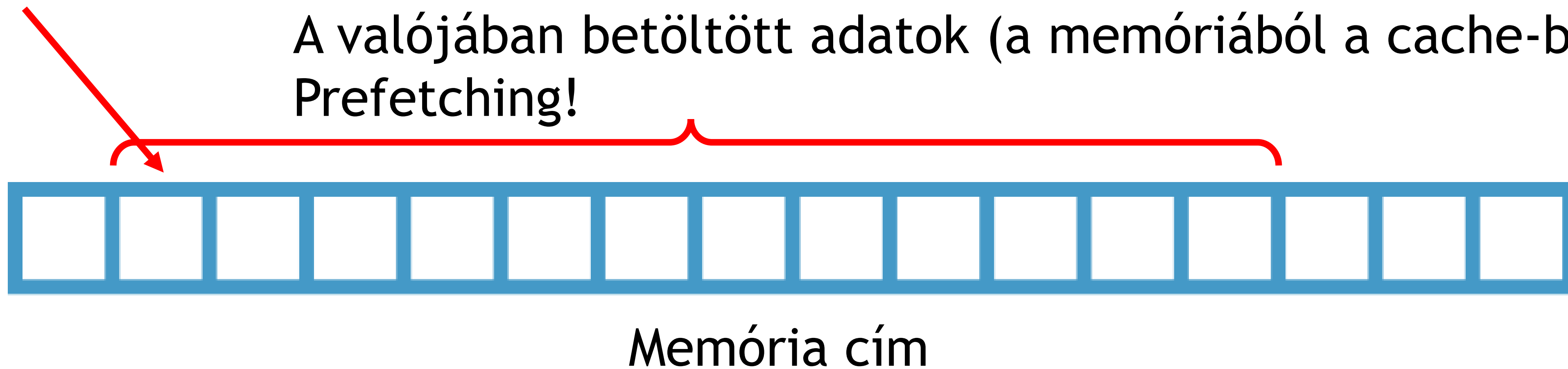
Optimalizációk - memória

A válasz a memória vezérlő és a cache működésében rejlik:



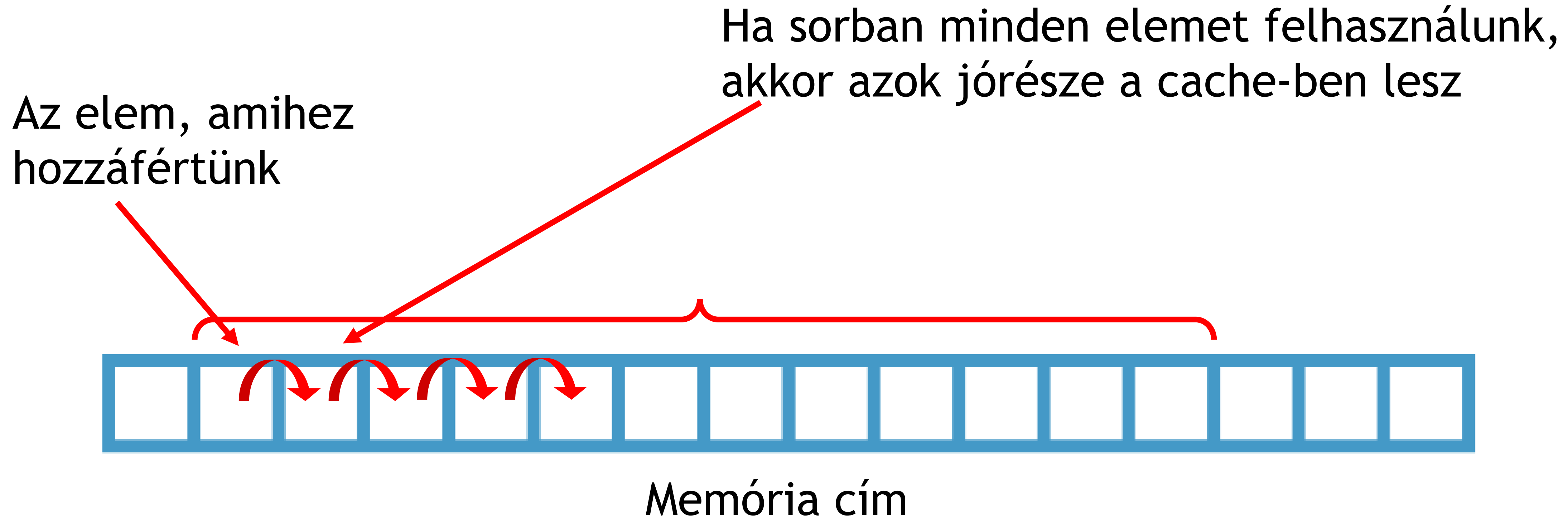
Optimalizációk - memória

A válasz a memória vezérlő és a cache működésében rejlik:



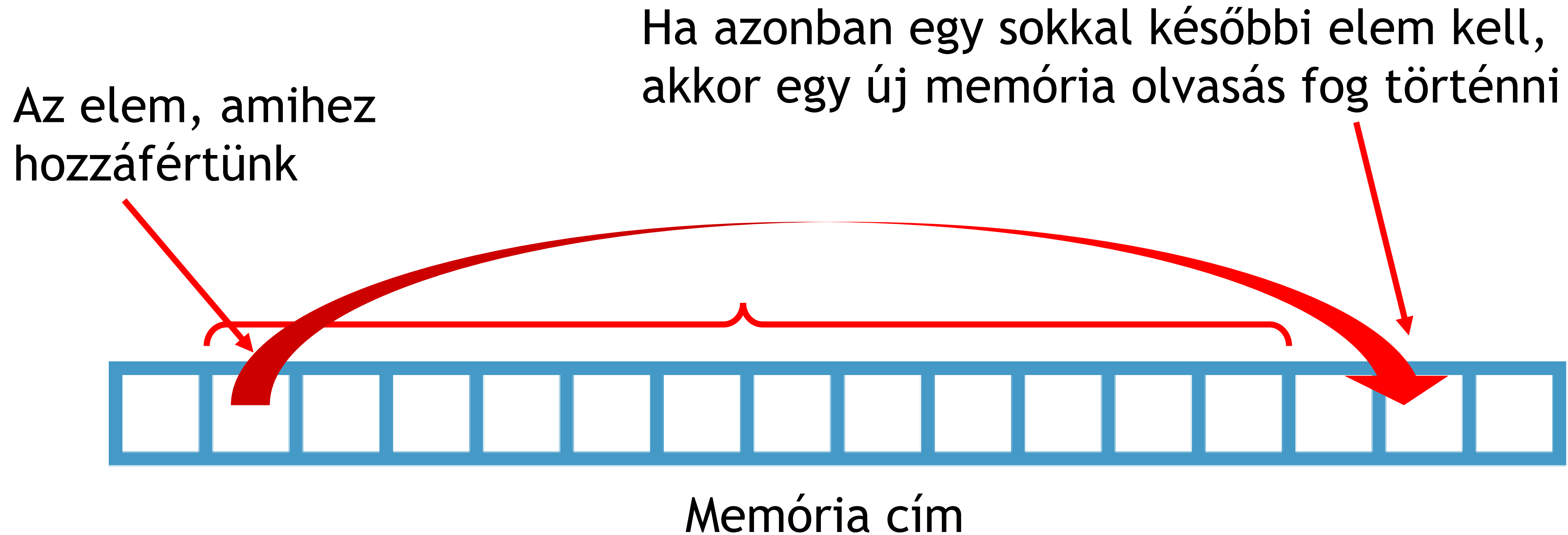
Optimalizációk - memória

A válasz a memória vezérlő és a cache működésében rejlik:



Optimalizációk - memória

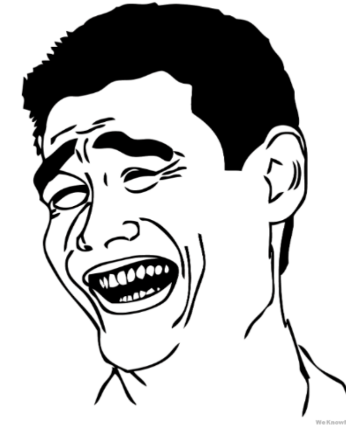
A válasz a memória vezérlő és a cache működésében rejlik:

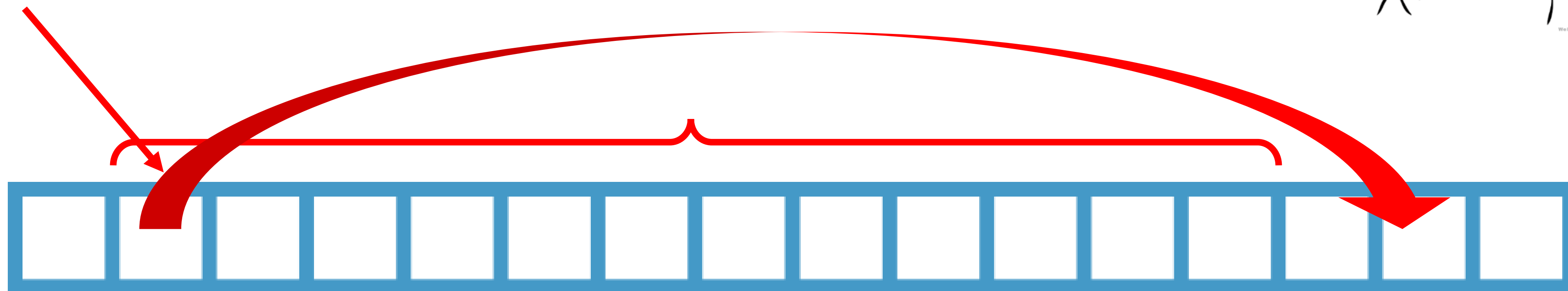


Optimalizációk - memória

A válasz a memória vezérlő és a cache működésében rejlik:

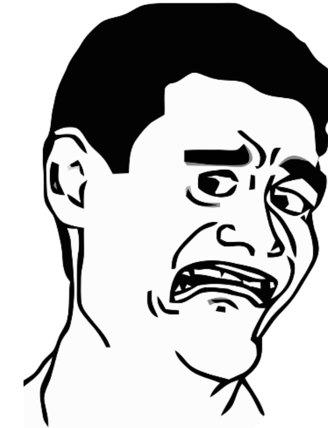
Ugyan már … ki ír ilyen kódot?

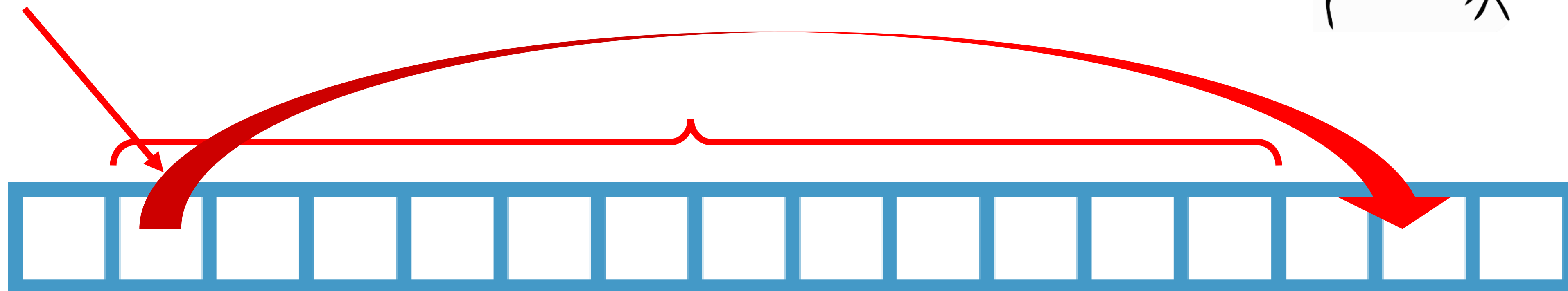




Optimalizációk - memória

A válasz a memória vezérlő és a cache működésében rejlik:





Optimalizációk - memória

A kihagyásos memóriahozzáférés tipikus esetei:

- Egy struktúrákból álló tömb csak egyes tagváltozóinak használata
- Többdimenziós tömbök

Optimalizációk - memória

Wigner GPU Lab

A kihagyásos memóriahozzáférés tipikus esetei:

- Egy struktúrákból álló tömb csak egyes tagváltozóinak használata
- Többdimenziós tömbök
- És a fenti kettő egyszerre

Optimalizációk - memória

Példa struktúra, és a belőle létrehozott tömb:

```
struct Particle
{
	std::array<double, 3> position;
	std::array<double, 3> velocity;
	std::array<double, 3> total_force;
	double mass, radius;
};
std::vector<Particle> particles;
```

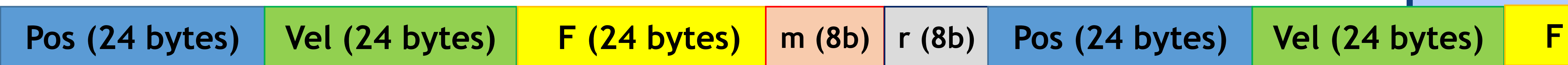
Optimalizációk - memória

Struktúra részleges használata:

```
struct Particle
{
        std::array<double, 3> position;
        std::array<double, 3> velocity;
        std::array<double, 3> total_force;
        double mass, radius;
};
std::vector<Particle> particles;
```

Ha erőt akarunk számolni, ahhoz általában csak a pozíció és a tömeg kell…

Csak ezeket használjuk



Particle

Particle

Optimalizációk - memória

Wigner GPU Lab

A megoldás, hogy transzponáljuk a struktúrát: megcseréljük, hogy a tömbök belül legyenek:

```
struct Particles
{
      std::vector< std::array<double, 3> > positions;
      std::vector< std::array<double, 3> > velocities;
      std::vector< std::array<double, 3> > total_forces;
      std::vector<double> masses;
      std::vector<double> radii;
};
Particles particles;
```

Optimalizációk - memória

A megoldás, hogy transzponáljuk a struktúrát: megcseréljük, hogy a tömbök belül legyenek:

```
struct Particles
{
        std::vector< std::array<double, 3> > positions;
        std::vector< std::array<double, 3> > velocities;
        std::vector< std::array<double, 3> > total_forces;
        std::vector<double> masses;
        std::vector<double> radii;
};
Particles particles;
```

Optimalizációk - memória

Több dimenziós tömbök!

- Példa: naiv mátrix szorzás (800x800)
  különböző index sorrenddel
  minden mátrix sorfolytonosan van tárolva:

$$C_{ik} = A_{ij} B_{jk}$$
$$D_{ik} = A_{ij} B_{kj}$$
$$E_{ik} = A_{ji} B_{jk}$$
$$F_{ki} = A_{ji} B_{jk}$$

Optimalizációk - memória

Wigner GPU Lab

Több dimenziós tömbök!

- Példa: naiv mátrix szorzás (800x800)
  különböző index sorrenddel
  minden mátrix sorfolytonosan van tárolva:

$$C_{ik} = A_{ij} B_{jk} \quad 1.82 \text{ sec}$$
$$D_{ik} = A_{ij} B_{kj} \quad 0.69 \text{ sec}$$
$$E_{ik} = A_{ji} B_{jk} \quad 6.45 \text{ sec}$$
$$F_{ki} = A_{ji} B_{jk} \quad 6.77 \text{ sec}$$

Optimalizációk - memória

Wigner GPU Lab

Több dimenziós tömbök!

- Példa: naiv 3-as tenzor – mátrix szorzás (400x400x400) különböző index sorrenddel
  most is sorfolytonos a tárolás:

$$C_{ijk} = A_{ijl} B_{kl}$$
$$D_{ijk} = A_{ijl} B_{lk}$$
$$E_{ijk} = A_{ilj} B_{kl}$$
$$F_{ijk} = A_{ilj} B_{lk}$$
$$G_{ijk} = A_{lij} B_{kl}$$

Optimalizációk - memória

Wigner GPU Lab

Több dimenziós tömbök!

- Példa: naiv 3-as tenzor – mátrix szorzás (400x400x400)
  különböző index sorrenddel
  most is sorfolytonos a tárolás:

$$C_{ijk} = A_{ijl} B_{kl}$$ 15 sec

$$D_{ijk} = A_{ijl} B_{lk}$$ 55 sec

$$E_{ijk} = A_{ilj} B_{kl}$$ 51 sec

$$F_{ijk} = A_{ilj} B_{lk}$$ 92 sec

$$G_{ijk} = A_{lij} B_{kl}$$ 64 sec

Optimalizációk

A továbbiakban két példán részletesen szemléltetjük az optimalizációs szempontokat:

- Mátrix-mátrix szorzás
- N-test szimuláció

Mátrix-mátrix szorzás

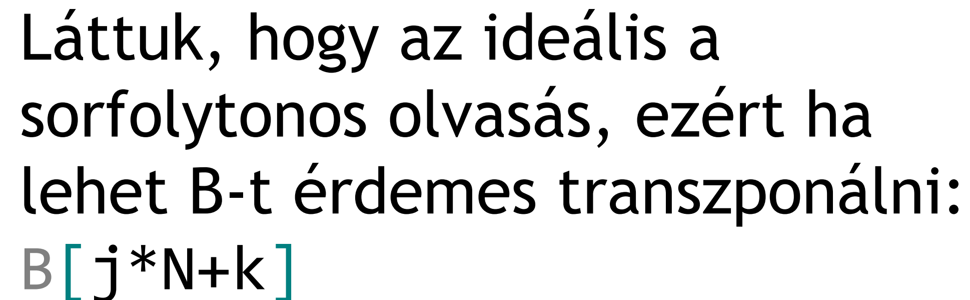
A naiv implementáció:

```
for( int i=0; i<N; ++i )
{
	for( int j=0; j<N; ++j )
	{
		double sum = 0.0;
		for( int k=0; k<N; ++k )
		{
			sum += A[i*N+k] * B[k*N+j];
		}
		C[i*N+j] = sum;
	}
}
```

Mátrix-mátrix szorzás

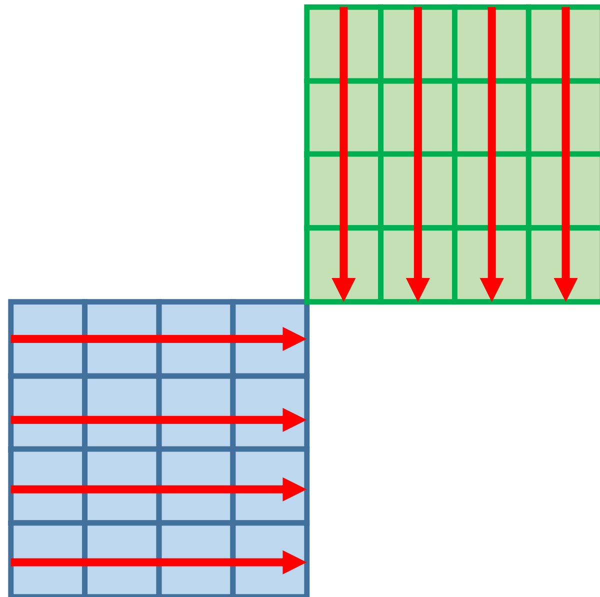
A naiv implementáció:

```
for( int i=0; i<N; ++i )
{
      for( int j=0; j<N; ++j )
      {
            double sum = 0.0;
            for( int k=0; k<N; ++k )
            {
                  sum += A[i*N+k] * B[k*N+j];
            }
            C[i*N+j] = sum;
      }
}
```



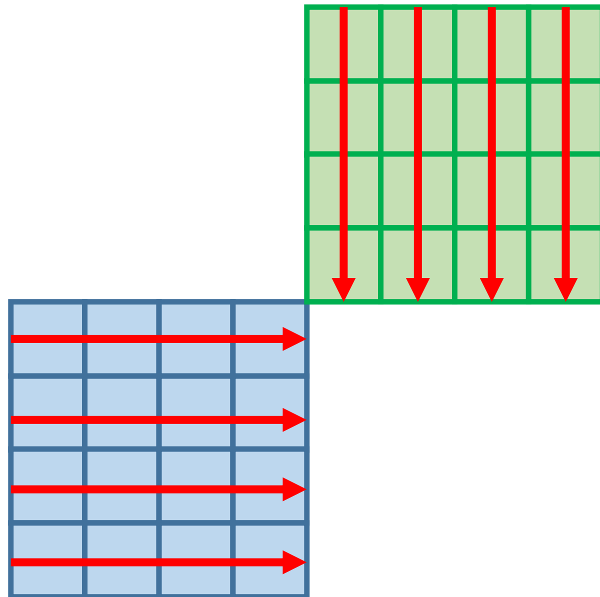
Mátrix-mátrix szorzás

A naiv megoldás hozzáférési mintázata ilyen:

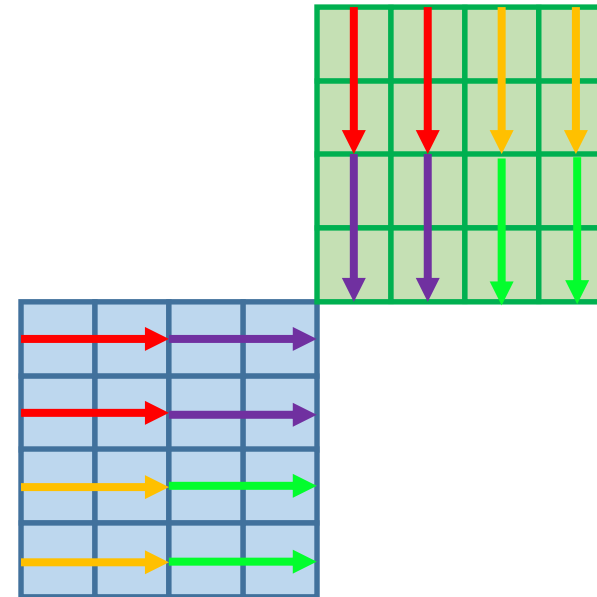


Mátrix-mátrix szorzás

A naiv megoldás hozzáférési mintázata ilyen:



Azonban a blokkosított elérés sokkal hatékonyabb:



Wigner GPU Lab

Mátrix-mátrix szorzás


```
for(int bi=0; bi<Bs; ++bi){ //block index 1
  for(int bj=0; bj<Bs; ++bj){ //block index 2
    for(int bk=0; bk<Bs; ++bk){ //block index 3
      auto i0 = bi * b; auto j0 = bj * b; auto k0 = bk * b;
      for(int i=0; i<b; ++i ){ auto ii = i0 + i;
        for(int j=0; j<b; ++j ){ auto jj = j0 + j; double sum = 0.0;
          for(int k=0; k<b; ++k ){
            sum += A[i*N+k0+k] * B[(k0+k)*N+j];
          }
          C[ii*N+jj] += sum;
        }
      }
    }
  }
}
```

Mátrix-mátrix szorzás

Időzítések: windows, clang, N = 1024

$$C_{ik} = A_{ij} B_{jk} \qquad 36.4 \text{ sec}$$

$$C_{ik} = A_{ij} B_{kj} \qquad 1.49 \text{ sec}$$

Wigner GPU Lab

Mátrix-mátrix szorzás

Időzítések:
windows, clang,
N = 1024

Blokk méret	$C_{ik} = A_{ij}B_{jk}$ Idő [s]	$C_{ik} = A_{ij}B_{kj}$ Idő [s]
1	66	6.7
2	16.1	1.7
4	5.9	1.8
8	2.4	1.6
16	2.9	1.3
32	2.7	1.01
64	**2.3**	0.93
128	2.6	0.75
256	2.8	**0.69**
512	15.2	1.49
1024 (ez a naiv)	36.4	1.48

Mátrix-mátrix szorzás

Időzítések: windows, msvc, N = 1024, 1023, 1025

Mátrix-mátrix szorzás

Wigner GPU Lab

Időzítések: windows, msvc, N = 1024, 1023, 1025

N = 1024 Blokk méret	Idő [s]
1	74
2	17.1
4	5.8
8	2.7
16	2.9
32	2.7
64	**2.48**
128	2.6
256	2.6
512	15.5
1024	36.1

N = 1023 Blokk méret	Idő [s]
1	10.4
3	2.8
11	1.8
13	**1.7**
33	1.8
93	2.7
341	2.9
1023	6.8

N = 1025 Blokk méret	Idő [s]
1	10.9
5	2.1
25	1.8
41	**1.7**
205	2.6
1025	8.5

Mátrix-mátrix szorzás

Mi folyik itt?

Miért tart tovább lényegesen az 1024-es méretű mátrix szorzás?

Mátrix-mátrix szorzás

Mi folyik itt?

Miért tart tovább lényegesen az 1024-es méretű mátrix szorzás?

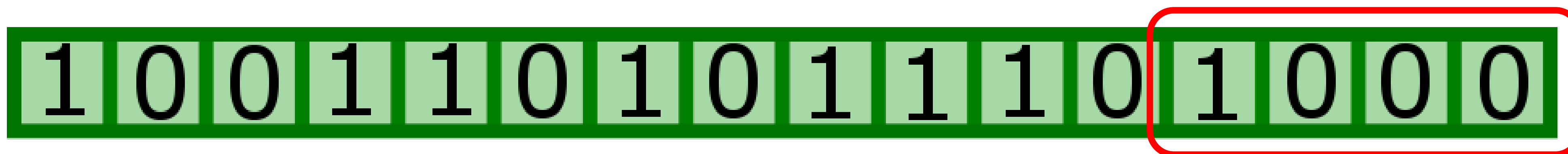
A válasz a cache működésében rejlik...

Mátrix-mátrix szorzás

- A cache nem egy sima random hozzáférésű memória, line-okra van osztva. Egy line pl. 64 byte
- Amikor az adat tárolásra kerül a cache-ben, a címének végéből képződik az index, hogy melyik line-ra kerül

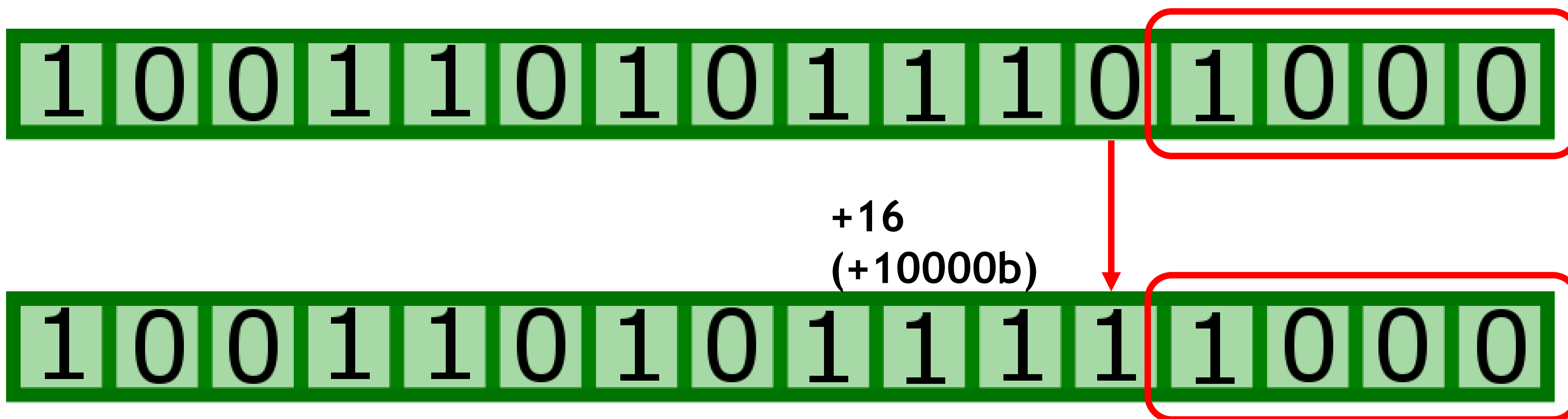
Mátrix-mátrix szorzás

- A cache nem egy sima random hozzáférésű memória, line-okra van osztva. Egy line pl. 64 byte
- Amikor az adat tárolásra kerül a cache-ben, a címének végéből képződik az index, hogy melyik line-ra kerül



Mátrix-mátrix szorzás

- Ha két olyan cím érkezik egymás után a cache-be, amik között az eltérés nem látszik az alsó pár biten, akkor azok ugyan arra a line-ra kerülnek:



Mátrix-mátrix szorzás

Wigner GPU Lab

- Ha két olyan cím érkezik egymás után a cache-be, amik között az eltérés nem látszik az alsó pár biten, akkor azok ugyan arra a line-ra kerülnek

- Az új adat annak ellenére írja felül a régit, hogy bőven lenne hely másik lineon…

További olvasnivaló

Optimalizációk - memória

További tippek optimalizációkra a memóriával kapcsolatban

Optimalizációk - memória

WIGNER GPU Lab

Ha lehet, fordítás időben ismert méreteket használjunk!

Példa: 3 dimenziós tömb indexelés

offset = ((((i*Nj)+j)*Nk)+k)

Ha Nj és Nk nem ismert fordítási időben, akkor plussz műveletek kellenek, hogy betöltsük őket.

Szemben, ha bele vannak fordítva a kódba, akkor sokkal olcsóbbak!

Optimalizációk - memória

Ha lehet, fordítás időben ismert méreteket használjunk!

Példa: 3 dimenziós tömb indexelés

offset = ((((i*Nj)+j)*Nk)+k)

Ni = 256, Nj = 256, Nk = 1024, elemek összege 3-as ciklussal:

Statikus méretek esetén: 172 ms

Dinamikus méretekkel: 389 ms

MSVC, windows

Optimalizációk - memória

Ha nagyon sokdimenziós adatunk van, és nagyon sokszor kell újraolvasni, tipikusan a csúszó ablakos eljárásoknál, pl.:

- Konvolúció
- Véges differencia sémák
- Szomszédokra hivatkozó rács számolások

Akkor érdemes lehet rész blokkokat kimásolni egy átmeneti tömbbe, hogy a stride méret csökkenjen a dimenziók mentén

Optimalizációk - memória

Ha bonyodalmas az index számítás, akkor amit lehet végezzünk el minél kintebbi ciklusban:

```
for(int i=0; i<b; ++i )
{
  auto ii = i0 + i;
  for(int j=0; j<b; ++j )
  {
    auto jj = j0 + j; double sum = 0.0;
    for(int k=0; k<b; ++k )
    {
      sum += A[i*N+k0+k] * B[(k0+k)*N+j];
    }
    C[ii*N+jj] += sum;
  }
}
```

Optimalizációk - memória

Ha bonyodalmas az index számítás, akkor amit lehet végezzünk el minél kintebbi ciklusban:

```
for(int i=0; i<b; ++i )
{
  auto ii = i0 + i;
  for(int j=0; j<b; ++j )
  {
    auto jj = j0 + j; double sum = 0.0;
    for(int k=0; k<b; ++k )
    {
      sum += A[i*N+k0+k] * B[(k+k0)*N+j];
    }
    C[ii*N+jj] += sum;
  }
}
```

Ezek változatlanok a belső ciklusban

Pár % nyerhető ezzel is
(minél komplexebb a számítás, annál több)

N-test szimuláció

A fizikában igen gyakoriak párkölcsönhatású N-test szimulációk

Itt is van mit optimalizálni☺

N-test szimuláció

Wigner GPU Lab

Két tömegpont gravitációs kölcsönhatása:

$$\vec{F} = \frac{-Gm_1 m_2(\vec{r}_1 - \vec{r}_2)}{|\vec{r}_1 - \vec{r}_2|^3}$$

Ezt fel kell összegezni az eredő erőhöz és később a lépétetőben (itt most egy egyszerű Euler) használjuk:

$$\vec{r}_{n+1} = \vec{r}_n + \vec{v}_n \Delta t + \frac{1}{2} \vec{a}_n \Delta t^2$$

N-test szimuláció

Wigner GPU Lab

Megint kell egy részecske osztály:

```
struct particle
{
    double mass;
    std::array<double, 3> pos;
    std::array<double, 3> v;
    std::array<double, 3> f;//ez lesz az eredő erő
};
```

N-test szimuláció

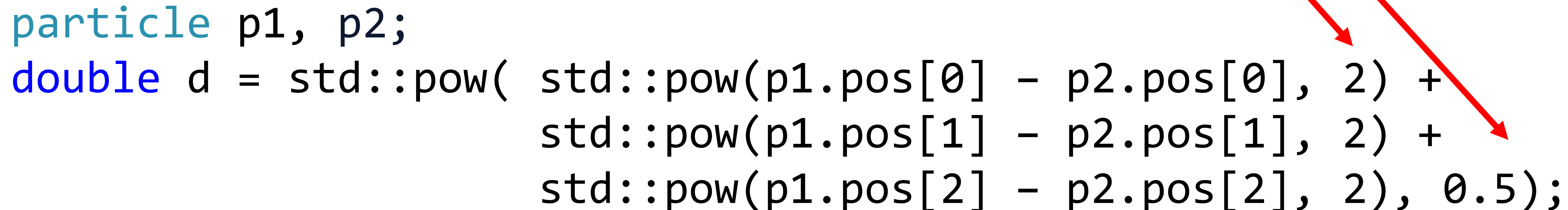
Az erőhöz kell távolság.
A leggyakoribb, ahogy naivan megírná az ember:

```
particle p1, p2;
double d = std::pow( std::pow(p1.pos[0] – p2.pos[0], 2) +
                     std::pow(p1.pos[1] – p2.pos[1], 2) +
                     std::pow(p1.pos[2] – p2.pos[2], 2), 0.5);
```

N-test szimuláció

Az erőhöz kell távolság.
A leggyakoribb, ahogy naivan megírná az ember:

Bár az ember elvárná, de nem minden fordító optimalizálja ki a speciális kitevőket…



```
particle p1, p2;
double d = std::pow( std::pow(p1.pos[0] - p2.pos[0], 2) +
                     std::pow(p1.pos[1] - p2.pos[1], 2) +
                     std::pow(p1.pos[2] - p2.pos[2], 2), 0.5);
```

N-test szimuláció

Az erőhöz kell távolság.

Érdemes bevezetni a négyzetre emelés függvényt:

```
template<typename T>
auto sq( T const& x ){ return x*x; }

particle p1, p2;
double d = std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) );
```

N-test szimuláció

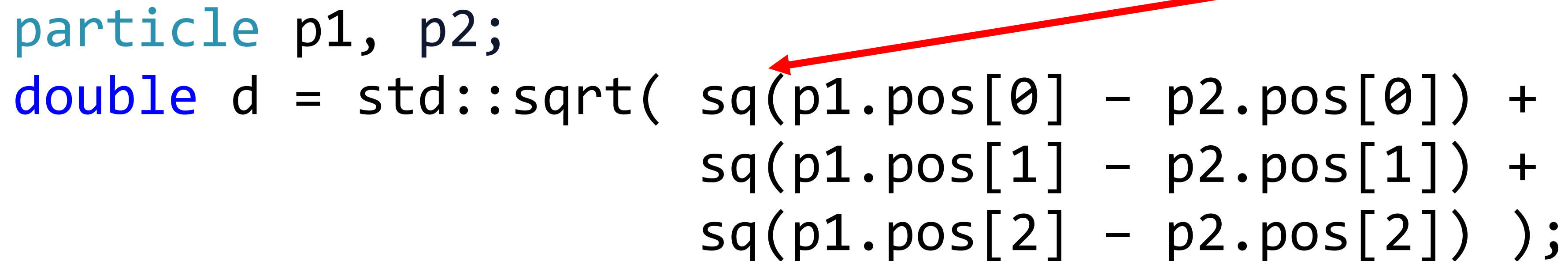
WIGNER GPU Lab

Az erőhöz kell távolság.

Érdemes bevezetni a négyzetre emelés függvényt:

```
template<typename T>
auto sq( T const& x ){ return x*x; }
```

Így a kód is jobban olvasható



```
particle p1, p2;
double d = std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) );
```

N-test szimuláció

De… Igazából nem is ez kell, hanem: $\frac{\vec{r}_1-\vec{r}_2}{|\vec{r}_1-\vec{r}_2|^3}$

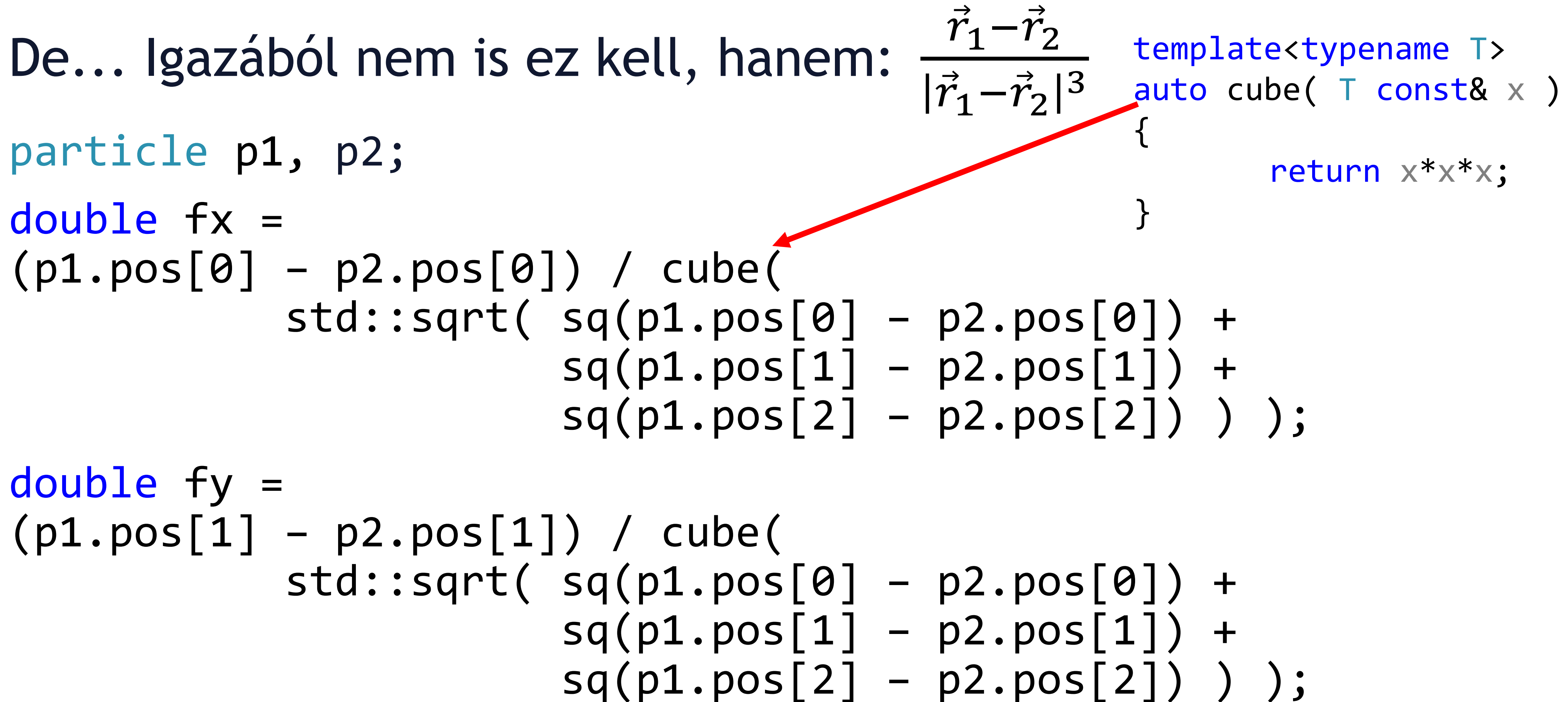
```
particle p1, p2;

double fx =
(p1.pos[0] – p2.pos[0]) / cube(
            std::sqrt( sq(p1.pos[0] – p2.pos[0]) +
                       sq(p1.pos[1] – p2.pos[1]) +
                       sq(p1.pos[2] – p2.pos[2]) ) );

double fy =
(p1.pos[1] – p2.pos[1]) / cube(
            std::sqrt( sq(p1.pos[0] – p2.pos[0]) +
                       sq(p1.pos[1] – p2.pos[1]) +
                       sq(p1.pos[2] – p2.pos[2]) ) );
```

N-test szimuláció

De… Igazából nem is ez kell, hanem: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$



```
template<typename T>
auto cube( T const& x )
{
        return x*x*x;
}
```

```
particle p1, p2;

double fx =
(p1.pos[0] - p2.pos[0]) / cube(
           std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) ) );

double fy =
(p1.pos[1] - p2.pos[1]) / cube(
           std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) ) );
```

N-test szimuláció

De... Igazából nem is ez kell, hanem: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

```
particle p1, p2;

double fx =
(p1.pos[0] – p2.pos[0]) / cube(
             std::sqrt( sq(p1.pos[0] – p2.pos[0]) +
                        sq(p1.pos[1] – p2.pos[1]) +
                        sq(p1.pos[2] – p2.pos[2]) ) );

double fy =
(p1.pos[1] – p2.pos[1]) / cube(
             std::sqrt( sq(p1.pos[0] – p2.pos[0]) +
                        sq(p1.pos[1] – p2.pos[1]) +
                        sq(p1.pos[2] – p2.pos[2]) ) );
```

N-test szimuláció

Ezt számoljuk: $\frac{\vec{r}_1-\vec{r}_2}{|\vec{r}_1-\vec{r}_2|^3}$

```
particle p1, p2;
double factor = cube(
            std::sqrt( sq(p1.pos[0] – p2.pos[0]) +
                       sq(p1.pos[1] – p2.pos[1]) +
                       sq(p1.pos[2] – p2.pos[2]) ) );
double fx = (p1.pos[0] – p2.pos[0]) / factor;
double fy = (p1.pos[1] – p2.pos[1]) / factor;
double fz = (p1.pos[2] – p2.pos[2]) / factor;
```

N-test szimuláció

Ezt számoljuk: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

```
particle p1, p2;

double factor = cube(
           std::sqrt( sq(p1.pos[0] - p2.pos[0]) +
                      sq(p1.pos[1] - p2.pos[1]) +
                      sq(p1.pos[2] - p2.pos[2]) ) );
double fx = (p1.pos[0] - p2.pos[0]) / factor;
double fy = (p1.pos[1] - p2.pos[1]) / factor;
double fz = (p1.pos[2] - p2.pos[2]) / factor;
```

N-test szimuláció

Ezt számoljuk: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

```
particle p1, p2;
double dx = p1.pos[0] – p2.pos[0];
double dy = p1.pos[1] – p2.pos[1];
double dz = p1.pos[2] – p2.pos[2];
double factor = cube( std::sqrt( sq(dx) + sq(dy) + sq(dz) ) );
double fx = dx / factor;
double fy = dy / factor;
double fz = dz / factor;
```

N-test szimuláció

Wigner GPU Lab

Ezt számoljuk: $\frac{\vec{r}_1-\vec{r}_2}{|\vec{r}_1-\vec{r}_2|^3}$

```
particle p1, p2;
double dx = p1.pos[0] – p2.pos[0];
double dy = p1.pos[1] – p2.pos[1];
double dz = p1.pos[2] – p2.pos[2];
double factor = cube( std::sqrt( sq(dx) + sq(dy) + sq(dz) ) );
double fx = dx / factor;
double fy = dy / factor;
double fz = dz / factor;
```

N-test szimuláció

Ezt számoljuk: $\frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

```
particle p1, p2;

double dx = p1.pos[0] - p2.pos[0];
double dy = p1.pos[1] - p2.pos[1];
double dz = p1.pos[2] - p2.pos[2];
double factor = 1. / cube( std::sqrt( sq(dx)+sq(dy)+sq(dz) ) );
double fx = dx * factor;
double fy = dy * factor;
double fz = dz * factor;
```

Más lesz a kerekítés!

N-test szimuláció

Maga az erő még pár faktorral több: $-Gm_1 m_2 \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

```
particle p1, p2;
double dx = p1.pos[0] – p2.pos[0];
double dy = p1.pos[1] – p2.pos[1];
double dz = p1.pos[2] – p2.pos[2];
double factor = 1. / cube( std::sqrt( sq(dx)+sq(dy)+sq(dz) ) );
double fx = dx * factor;
double fy = dy * factor;
double fz = dz * factor;
```

N-test szimuláció

Maga az erő még pár faktorral több: $-Gm_1 m_2 \frac{\vec{r}_1 - \vec{r}_2}{|\vec{r}_1 - \vec{r}_2|^3}$

Ezek is közös faktorok mindegyik koordinátához

```
particle p1, p2;
double dx = p1.pos[0] - p2.pos[0];
double dy = p1.pos[1] - p2.pos[1];
double dz = p1.pos[2] - p2.pos[2];
double factor = -G * p1.mass * p2.mass
                  / cube( std::sqrt( sq(dx)+sq(dy)+sq(dz) ) );
double fx = dx * factor;
double fy = dy * factor;
double fz = dz * factor;
```

N-test szimuláció

Az erő számolás végül kb. így nézhet ki:

```
std::array<double, 3> calculate_force(const particle& p1,
                                      const particle& p2)
{
   std::array<double, 3> dr
   { p1.pos[0] - p2.pos[0], p1.pos[1] - p2.pos[1],
     p1.pos[2] - p2.pos[2] };

   double d = std::sqrt( sq(dr[0]) + sq(dr[1]) + sq(dr[2]) );
   double f = -G * p1.mass * p2.mass / cube(d);
   return { f * dr[0], f * dr[1], f * dr[2] };
}
```

N-test szimuláció

Teljesen hasonló gondolatmenettel az időléptetés:

```
void forward_euler(particle& part, const double& dt){
   auto dt_per_m = dt / part.mass;
   auto half_dt_sq_per_m = 0.5 * dt_per_m * dt;

   part.pos[0] += part.v[0] * dt + part.f[0] * half_dt_sq_per_m;
   part.pos[1] += part.v[1] * dt + part.f[1] * half_dt_sq_per_m;
   part.pos[2] += part.v[2] * dt + part.f[2] * half_dt_sq_per_m;

   part.v[0] += part.f[0] * dt_per_m;
   part.v[1] += part.f[1] * dt_per_m;
   part.v[2] += part.f[2] * dt_per_m;
   part.f = { 0.0, 0.0, 0.0 };
}
```

N-test szimuláció

Teljesen hasonló gondolatmenettel az időléptetés:

```
void forward_euler(particle& part, const double& dt){
    auto dt_per_m = dt / part.mass;
    auto half_dt_sq_per_m = 0.5 * dt_per_m * dt;

    part.pos[0] += part.v[0] * dt + part.f[0] * half_dt_sq_per_m;
    part.pos[1] += part.v[1] * dt + part.f[1] * half_dt_sq_per_m;
    part.pos[2] += part.v[2] * dt + part.f[2] * half_dt_sq_per_m;

    part.v[0] += part.f[0] * dt_per_m;
    part.v[1] += part.f[1] * dt_per_m;
    part.v[2] += part.f[2] * dt_per_m;
    part.f = { 0.0, 0.0, 0.0 };
}
```

Figyelem!

Gyakori hiba!

Amikor módosítunk egy értéket a léptetéskor, ellenőrizzük, hogy egy későbbi komponens nem használja-e a RÉGI értéket!

Pl. ha itt most először a sebességeket léptetnénk és utána a pozíciókat, az hibás lenne!

N-test szimuláció

Megjegyzések:

- Az itt bemutatott átrendezéseket a fordító vagy megcsinálja, vagy nem.
  Ha eleve átrendezve írjuk fel, akkor valószínűleg jobb eredményt kapunk (továbbra sem 100%)

- Az átrendezések megváltoztatják a kerekítést!
  Az esetek eltérhetnek az eredmények utolsó tizedeseiben
  Mindig legyen elképzelésünk a számok nagyságrendjéről, szélsőséges értékeiről!

N-test szimuláció

Az erőket fel kell összegezni az összes részecske párra...

Legyen nekünk egy:

```
std::vector< particle > particles;
```

N-test szimuláció

Naiv megoldás: (az erők a léptetésnél ki vannak nullázva!)

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  for(auto it = particles.cbegin(); it != particles.cend(); ++it)
  {
    if (IT != it)//önkölcsönhatás nincs!
    {
      auto f = calculate_force(*IT, *it);
      IT->f[0] += f[0];
      IT->f[1] += f[1];
      IT->f[2] += f[2];
    }
  }
}
```

N-test szimuláció

Wigner GPU Lab

Naiv megoldás: (az erők a léptetésnél ki vannak nullázva!)

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  for(auto it = particles.cbegin(); it != particles.cend(); ++it)
  {
    if (IT != it)//önkölcsönhatás nincs!
    {
      auto f = calculate_force(*IT, *it);
      IT->f[0] += f[0];
      IT->f[1] += f[1];
      IT->f[2] += f[2];
    }
  }
}
```

Probléma: állandóan olvassuk és írjuk a távoli objektumot a heap-en!

Ez lassú...

N-test szimuláció

Wigner GPU Lab

Kevésbé naiv megoldás: (az erők a léptetésnél ki vannak nullázva!)

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != particles.cend(); ++it)
  {
    if (IT != it)//önkölcsönhatás nincs!
    {
       auto f = calculate_force(*IT, *it);
       force[0] += f[0];
       force[1] += f[1];
       force[2] += f[2];
    }

  }
  IT->f = force;
}
```

Használjunk egy lokális összegző változót a stack-en!

N-test szimuláció

Kevésbé naiv megoldás: (az erők a léptetésnél ki vannak nullázva!)

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != particles.cend(); ++it)
  {
    if (IT != it)//önkölcsönhatás nincs!
    {
       auto f = calculate_force(*IT, *it);
       force[0] += f[0];
       force[1] += f[1];
       force[2] += f[2];
    }

  }
  IT->f = force;
}
```

Nem szerencsés ciklusban elágazni...

N-test szimuláció


```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != IT; ++it)
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  for(auto it = std::next(IT); it != particles.cend(); ++it )
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  IT->f = force;
}
```

N-test szimuláció

```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != IT; ++it)
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  for(auto it = std::next(IT); it != particles.cend(); ++it )
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  IT->f = force;
}
```

Ketté választottuk a belső ciklust

N-test szimuláció


```
for(auto IT = particles.begin(); IT != particles.end(); ++IT)
{
  std::array<double, 3> force{ 0.0, 0.0, 0.0 };
  for(auto it = particles.cbegin(); it != IT; ++it)
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  for(auto it = std::next(IT); it != particles.cend(); ++it )
  {
    auto f = calculate_force( *IT, *it );
    force[0] += f[0]; force[1] += f[1]; force[2] += f[2];
  }
  IT->f = force;
}
```

Itt lépjük át az önkölcsönhatást

N-test szimuláció

Ez már nem annyira rossz.

De...

A cache-t pont annyira rosszul használjuk, mint a naiv mátrix szorzásnál...

A megoldás is ugyan az: blokkosítás!

N-test szimuláció

Wigner GPU Lab

```
size_t N = 1024;//block size
for( auto b = particles.begin(); b != particles.end(); b += N ){
  for( auto IT = particles.begin(); IT != particles.end(); ++IT ){
    auto before = intersect( particles.begin(), IT, b, b + N );
    auto after  = intersect( std::next( IT ), particles.end(), b, b + N );

    std::array<double, 3> force{ 0.0, 0.0, 0.0 };

    for( auto it = before.first; it != before.second; ++it )
    { /*calculate_force( *IT, *it ); accumulate to force*/ }
    for ( auto it = after.first; it != after.second; ++it )
    { /*calculate_force( *IT, *it ); accumulate to force*/ }

    IT->f[0] += force[0]; IT->f[1] += force[1]; IT->f[2] += force[2];
  }
}
```

N-test szimuláció

```
size_t N = 1024;//block size
for( auto b = particles.begin(); b != particles.end(); b += N ){
  for( auto IT = particles.begin(); IT != particles.end(); ++IT ){
    auto before = intersect( particles.begin(), IT, b, b + N );
    auto after  = intersect( std::next( IT ), particles.end(), b, b + N );

    std::array<double, 3> force{ 0.0, 0.0, 0.0 };

    for( auto it = before.first; it != before.second; ++it )
    { /*calculate_force( *IT, *it ); accumulate to force*/ }

    for ( auto it = after.first; it != after.second; ++it )
    { /*calculate_force( *IT, *it ); accumulate to force*/ }


    IT->f[0] += force[0]; IT->f[1] += force[1]; IT->f[2] += force[2];
  }
}
```

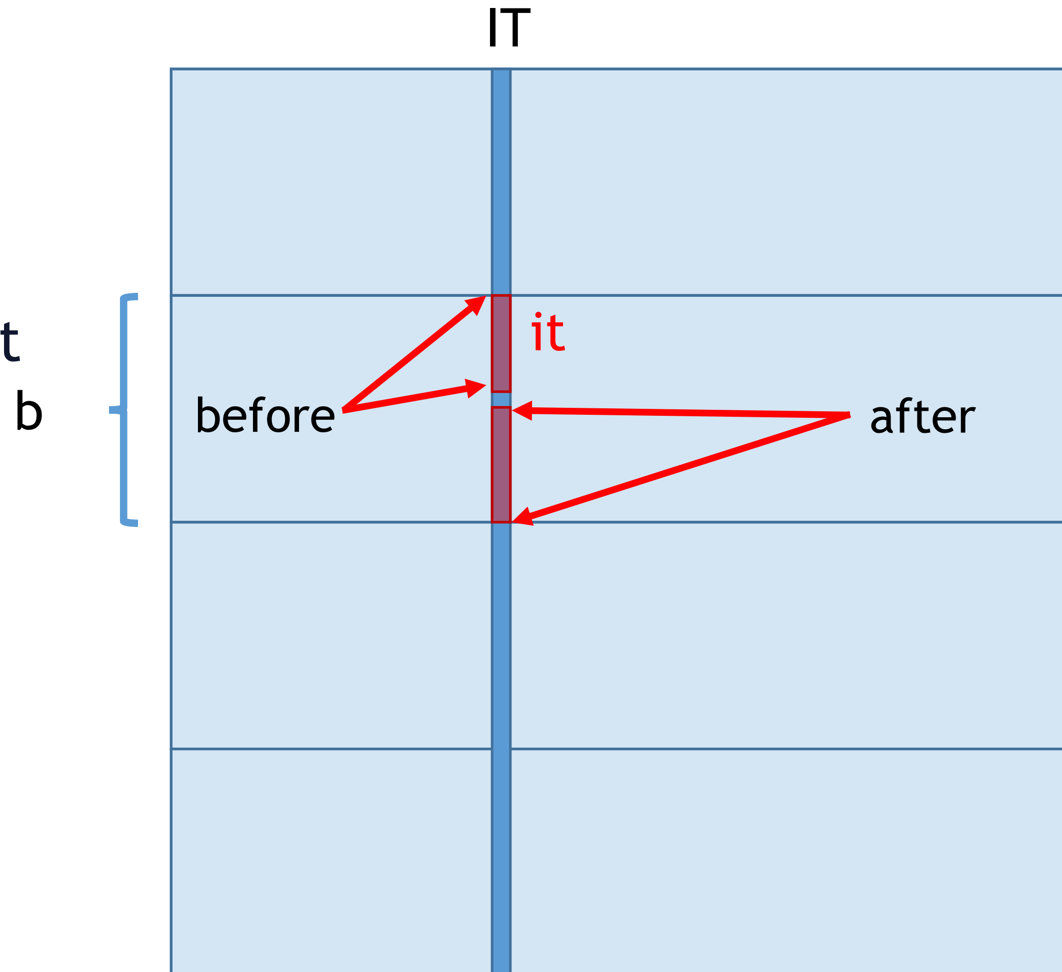
Itt vágjuk el az önkölcsönhatásnál a belső ciklust

N-test szimuláció

A bejárási mintázat tehát ilyesmi:

Minden IT részecskéhez a pár másik felét csak a b blokkon belülről választjuk, átugorva az önkölcsönhatást.

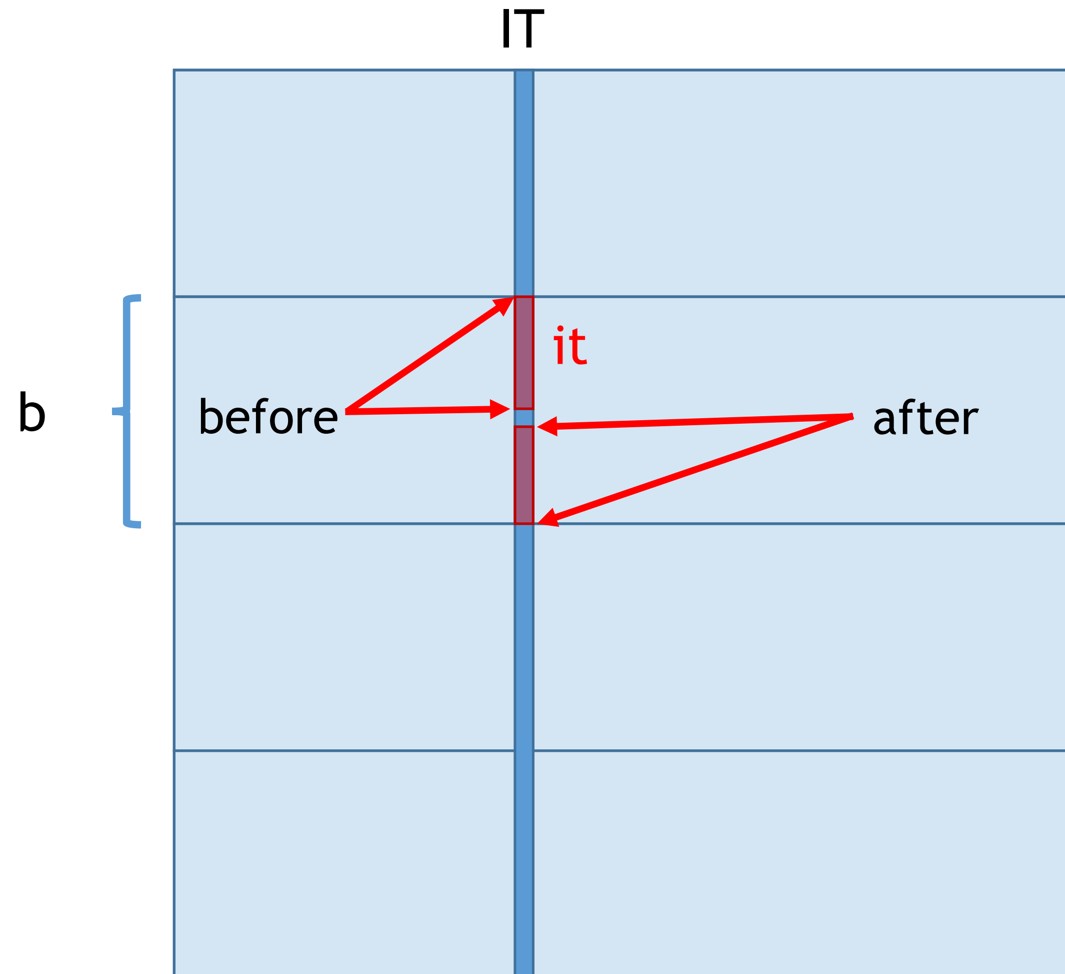
Így nem csak az IT részecske adatai lesznek a cache-ben, hanem nagy valószínűséggel a b blokkon belüliek is, hiszen b-t léptetjük a legritkábban.



N-test szimuláció

A bejárási mintázat tehát ilyesmi:

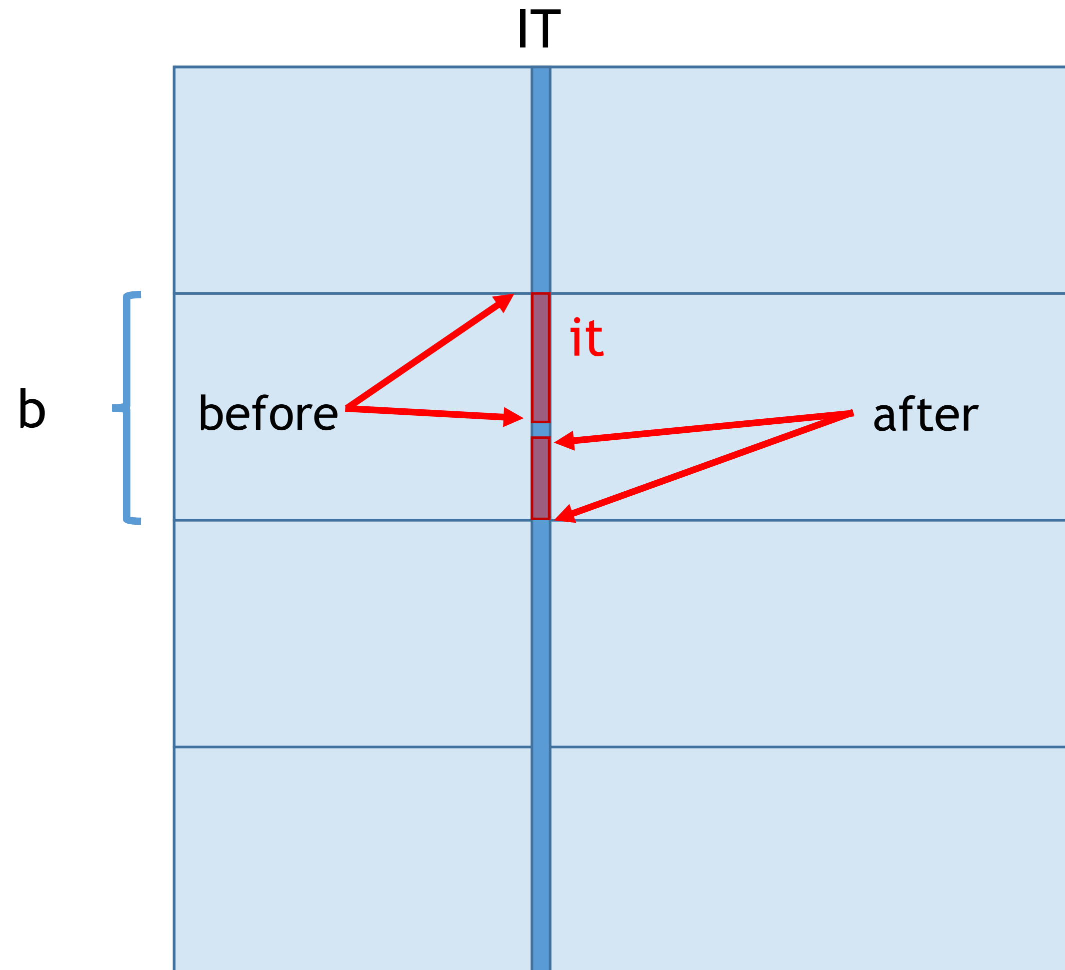
1 IT-vel később:



N-test szimuláció

A bejárási mintázat tehát ilyesmi:

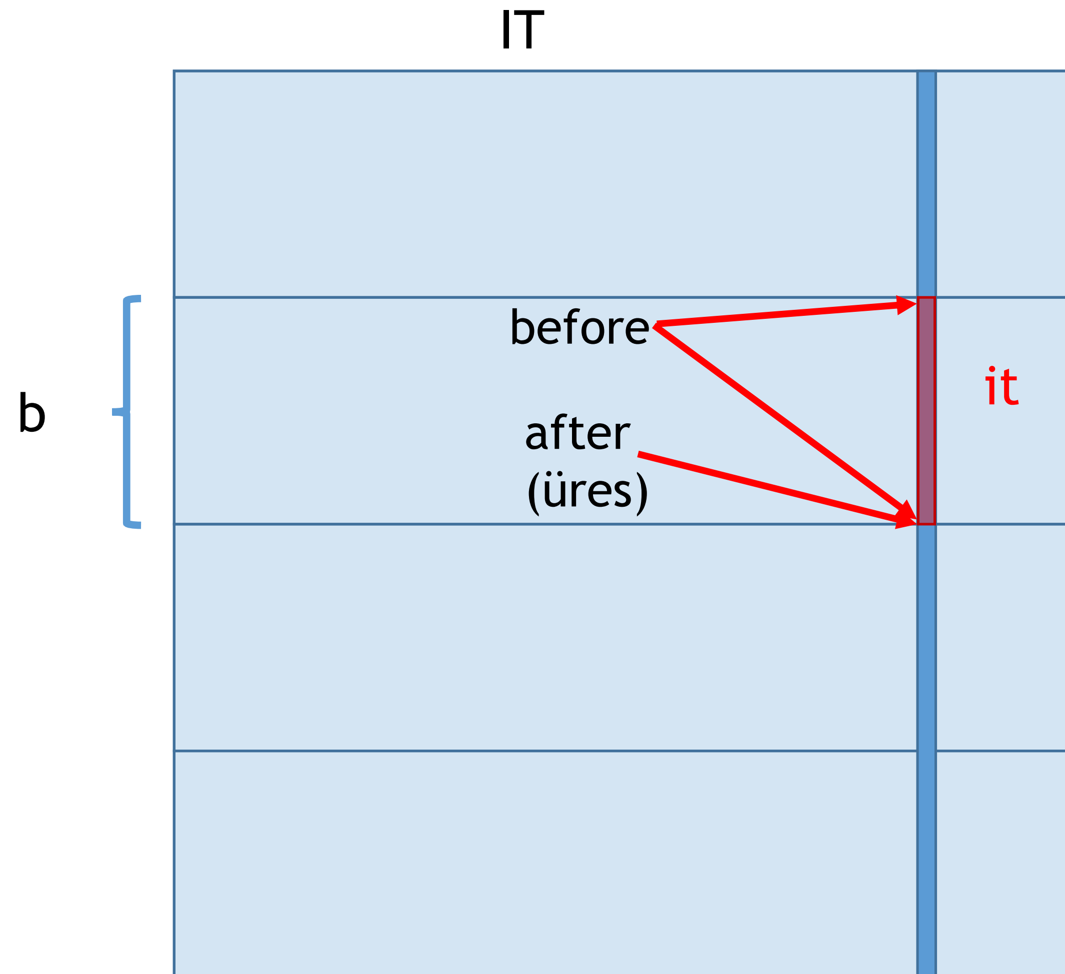
2 IT-vel később:



N-test szimuláció

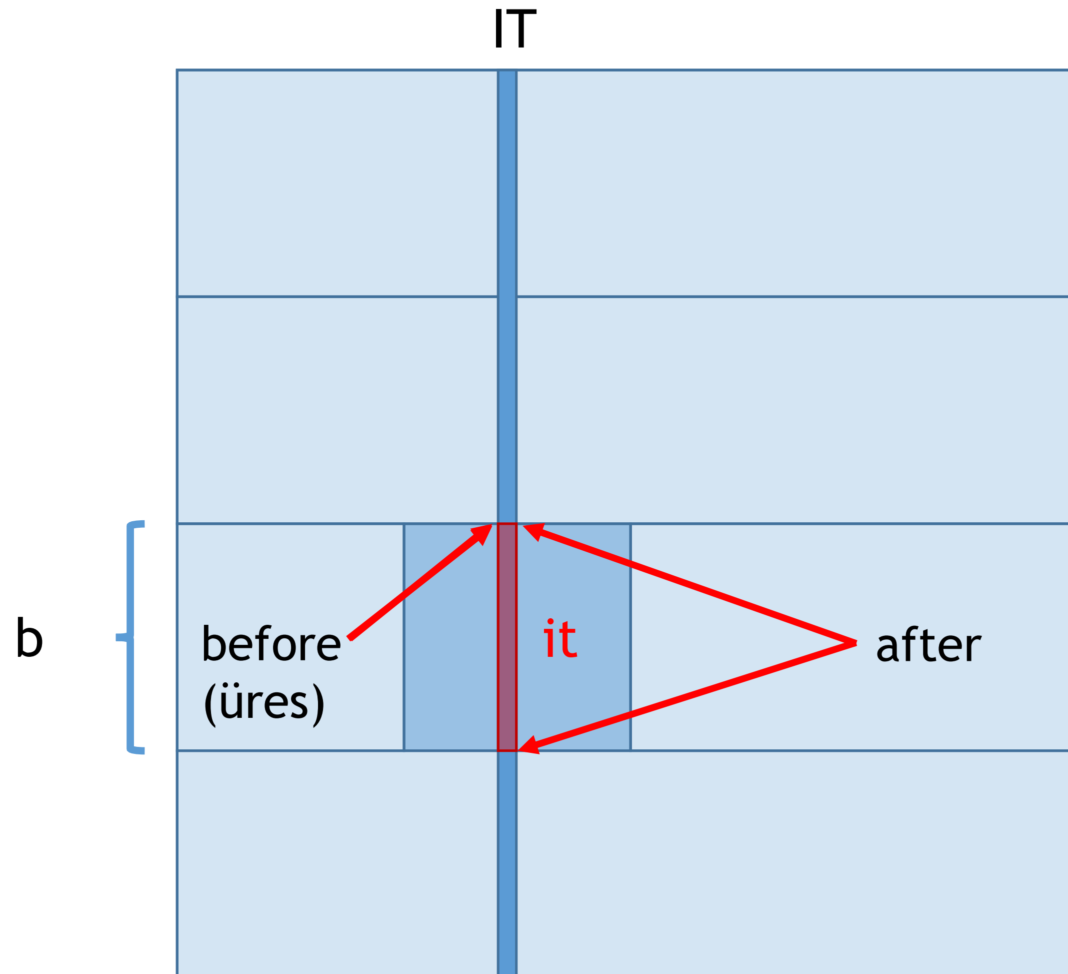
A bejárási mintázat tehát ilyesmi:

sok IT-vel később:



N-test szimuláció

Egy másik pillanatkép
egy blokkal később:



N-test szimuláció

Az `intersect` függvény:

```
template <typename RndIt>
std::pair<RndIt, RndIt> intersect( RndIt first1, RndIt last1,
                                   RndIt first2, RndIt last2 )
{  //ha nem fed át a két tartomány...
   if ( ((last2 < first1) || (last1 < first2)) )
   {
      return std::make_pair( first1, first1 );//üres tartomány
   }
   else{//egyébként a két tartomány metszete az eredmény:
     return std::make_pair( std::max( first1, first2 ),
                            std::min( last1 , last2  )  ); }
}
```

N-test szimuláció

Továbbá:

az erő szimmetrikus a részecskékben, elég egyszer számolni minden párt!

N-test szimuláció

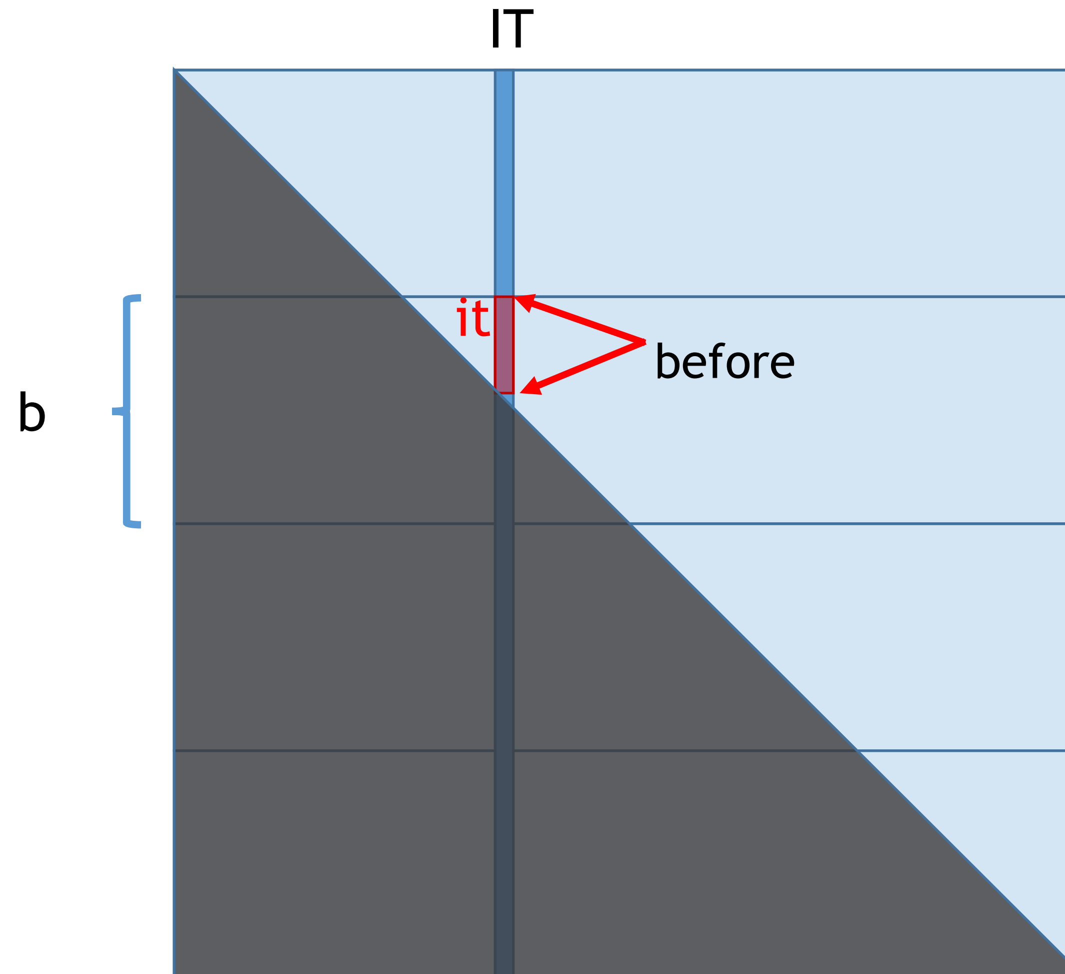

```
size_t N = 1024;//block size
for( auto b =  particles.begin();
          b != particles.end(); b += N ){

   for( auto IT = particles.begin(); IT != particles.end(); ++IT ){

     auto before = intersect( particles.begin(), IT, b, b + N );
      for( auto it = before.first; it != before.second; ++it )
      {
        auto force = calculate_force( *IT, *it );
        IT->f[0] += force[0]; IT->f[1] += force[1]; IT->f[2] += force[2];
        it->f[0] -= force[0]; it->f[1] -= force[1]; it->f[2] -= force[2];
      }
   }
}
```

N-test szimuláció

A bejárási mintázat most annyit változik, hogy az alsó háromszöget egyáltalán nem érintjük:



N-test szimuláció

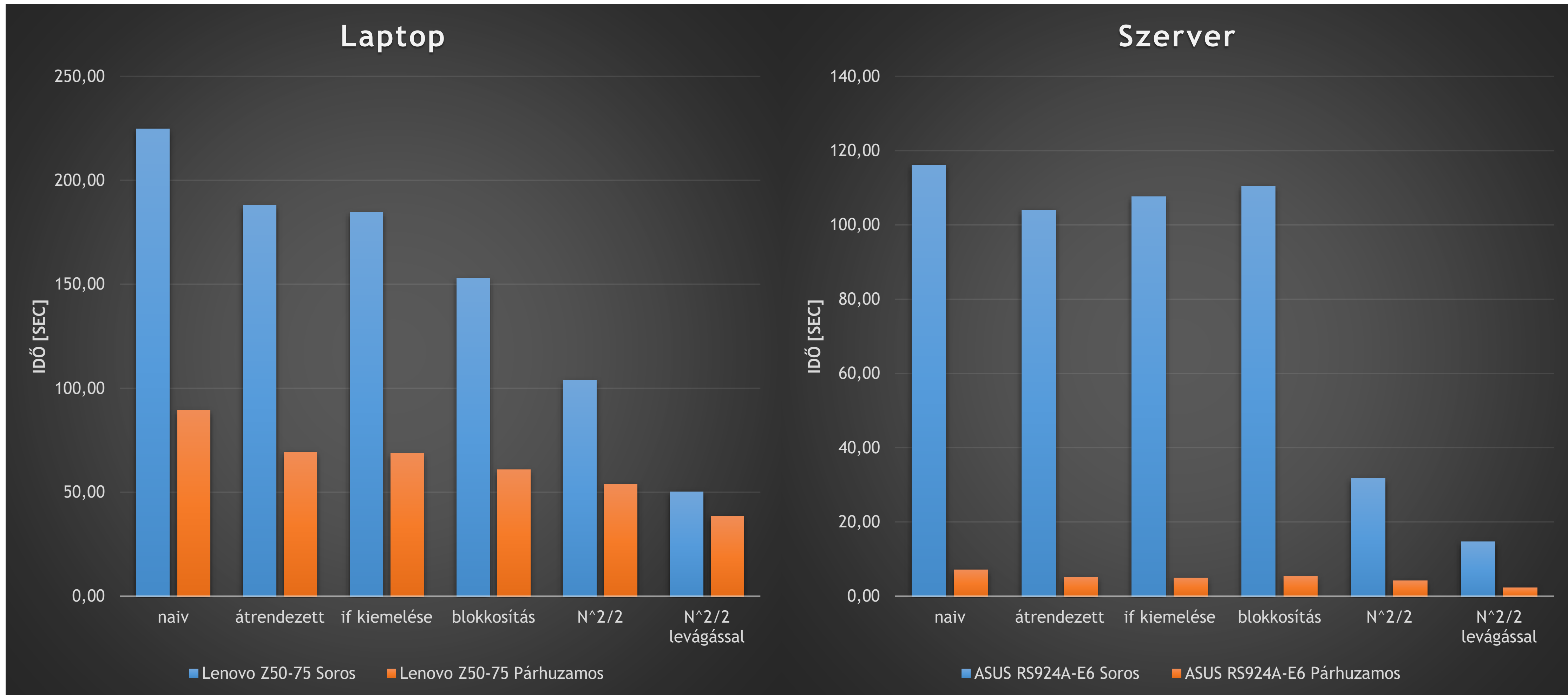
Fizikailag még el lehet hanyagolni egy küszöb távolságnál messzebbi párokat, így az erő számolás egy része megspórolható.

N-test szimuláció

Mennyit jelentenek ezek időben?

64k részecskére...

N-test szimuláció



N-test szimuláció

Optimalizáció	Lenovo Z50-75		ASUS RS924A-E6	
	Soros	Párhuzamos (4 szál)	Serial	Párhuzamos (64 szál)
naiv	225s	89s	116s	7s
átrendezett	188s	69s	104s	5s
if kiemelése	185s	69s	108s	5s
blokkosítás	153s	61s	110s	5s
N^2/2	104s	54s	32s	4s
levágással	50s	38s	15s	2s

N-test szimuláció

További optimalizációk az N-test esetére:

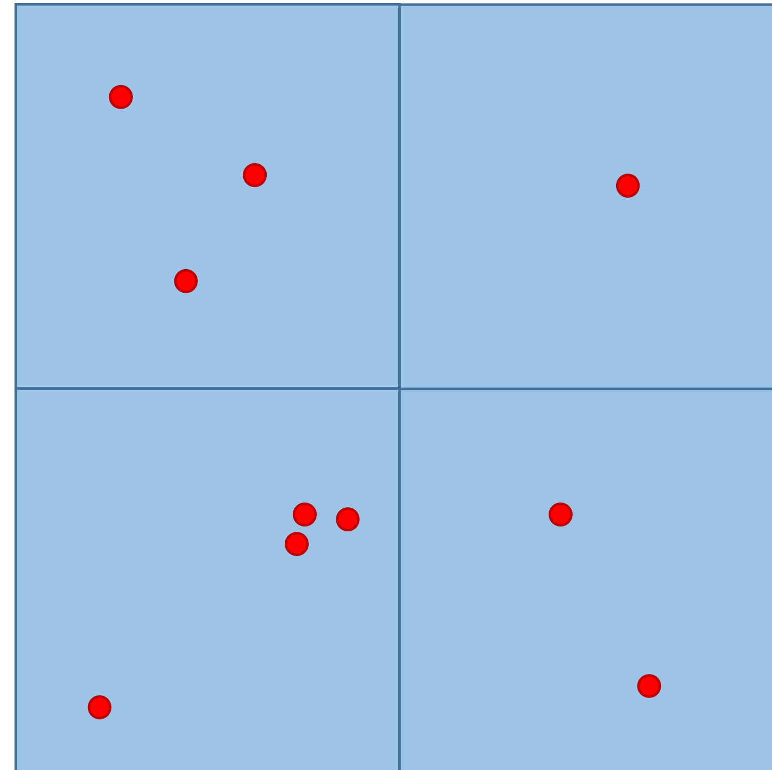
Térpartícionálás

N-test szimuláció

Wigner GPU Lab

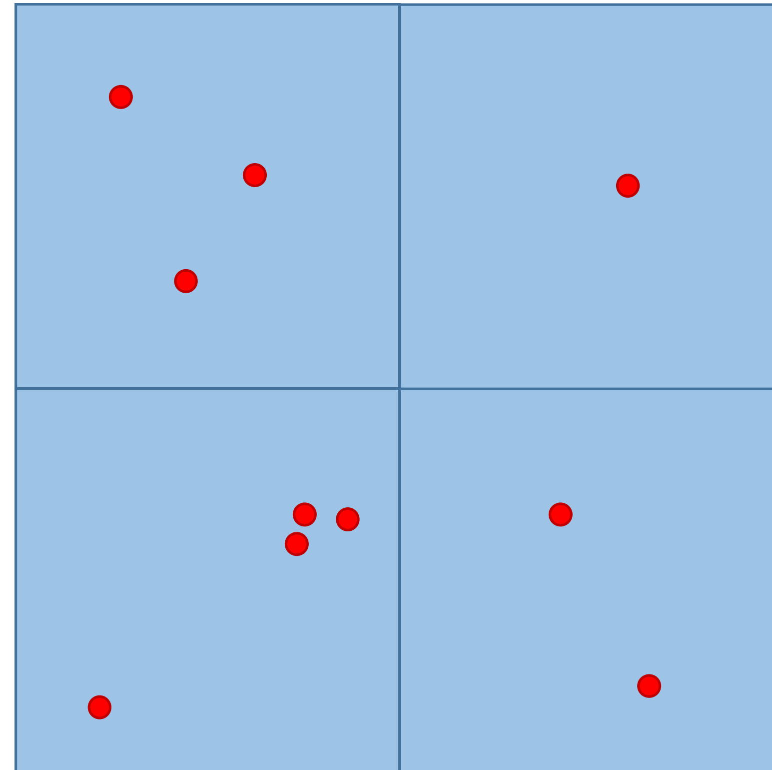
Térpartícionálás

- Osszuk fel a szimulációs teret blokkokra
- Minden blokkban van egy tömb, ami tárolja a blokkban levő részecskéket
- Ahogy a részecskék mozognak, frissíteni kell a blokkok tartalmát



N-test szimuláció

Térpartícionálás



Előny:

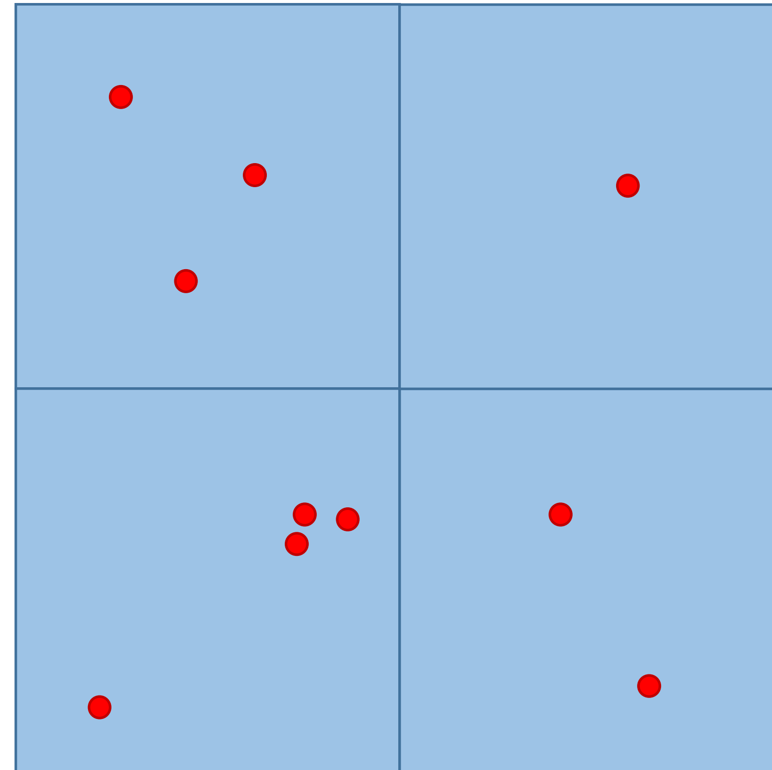
- Egzakt kölcsönhatás csak néhány blokk távolságig
- Távolabb csak a tömegközéppontok hatnak kölcsön (multipól sorfejtés)

Hátrány:

- Pontosság vesztés
- Meg is kell írni☺

N-test szimuláció

Térpartícionálás



További előnyök:

- A blokkokon belül nő az adatlokalitás, jobb lesz a cache kihasználtság
- Könnyű párhuzamosítani

N-test szimuláció

WIGNER GPU Lab

Egy szimulációs lépés (N_cell = 30^2, N_part = N_total / N_cell):

1. Részecskék beosztása a cellákba ($N_{cell}N_{part}$)
2. Cellán belüli erők ($N_{cell}N_{part}^2/2$)
3. A legközelebbi cellák között egzakt erők ($4N_{cell}N_{part}^2/2$)
4. Tömegközépponti átlagok számolása ($N_{cell}N_{part}$)
5. Cellák kölcsönhatása az átlagokon keresztül (N_{cell}^2)
6. Idő léptetés ($N_{cell}N_{part}$)

N-test szimuláció

Wigner GPU Lab

Egy szimulációs lépés (N_cell = 30^2, N_part = 25000 / N_cell):

1.	Részecskék beosztása a cellákba	1 ms (2%)
2.	Cellán belüli erők	5.5 ms (11%)
3.	A legközelebbi cellák között egzakt erők	33 ms (67%)
4.	Tömegközépponti átlagok számolása	1.3 ms (3%)
5.	Cellák kölcsönhatása az átlagokon keresztül	7.4 ms (15%)
6.	Idő léptetés	0.8 ms (1.7%)
	Összesen:	49 ms

Optimalizációk

Még pár apróság...

Optimalizációk

Érdemes figyelni a struktúrák elemeinek sorrendjére:

A nagy méretű elemek általában 16-32-64 byte határokra vannak igazítva. A kis méretű elemek ezt elrontják, a fordító paddingot rak be, nem lesz hatékony a memória és cache elérés…

ROSSZ

```
struct particle
{
    std::array<double, 3> pos;
    bool bActive
    std::array<double, 3> v;
    double mass;
};
```

Jó

```
struct particle
{
    std::array<double, 3> pos;
    std::array<double, 3> v;
    double mass;
    bool bActive
};
```

Optimalizációk

Wigner GPU Lab

Érdemes figyelni a struktúrák elemeinek sorrendjére:

Az objektum címe az első elemre mutat (általában), ezért célszerű a gyakran használt elemeket tenni előre, a ritkábban használtakat hátrébb, hogy valószínűbben legyenek cacheban a gyakran kellő részek.

ROSSZ

```
struct particle
{
    std::array<double, 3> pos;
    bool bActive
    std::array<double, 3> v;
    double mass;
};
```

Jó

```
struct particle
{
    std::array<double, 3> pos;
    std::array<double, 3> v;
    double mass;
    bool bActive
};
```

Optimalizációk

Wigner GPU Lab

- Ha több ÉS-elt feltételünk van és sejtjük, hogy melyik milyen gyakran igaz, akkor rendezzük őket úgy, hogy a leggyakrabban hamis legyen elöl (VAGY esetén fordítva)

Általában hamis (cond1) — Majdnem mindig igaz (cond3)

```
if( cond1() && cond2() && cond3() ){ ... }
```

Az ÉS és VAGY rövidrezártan értékelődik ki C++-ban:
Ha az első hamis, akkor a `cond2()`, `cond3()` meg sem hívódik!