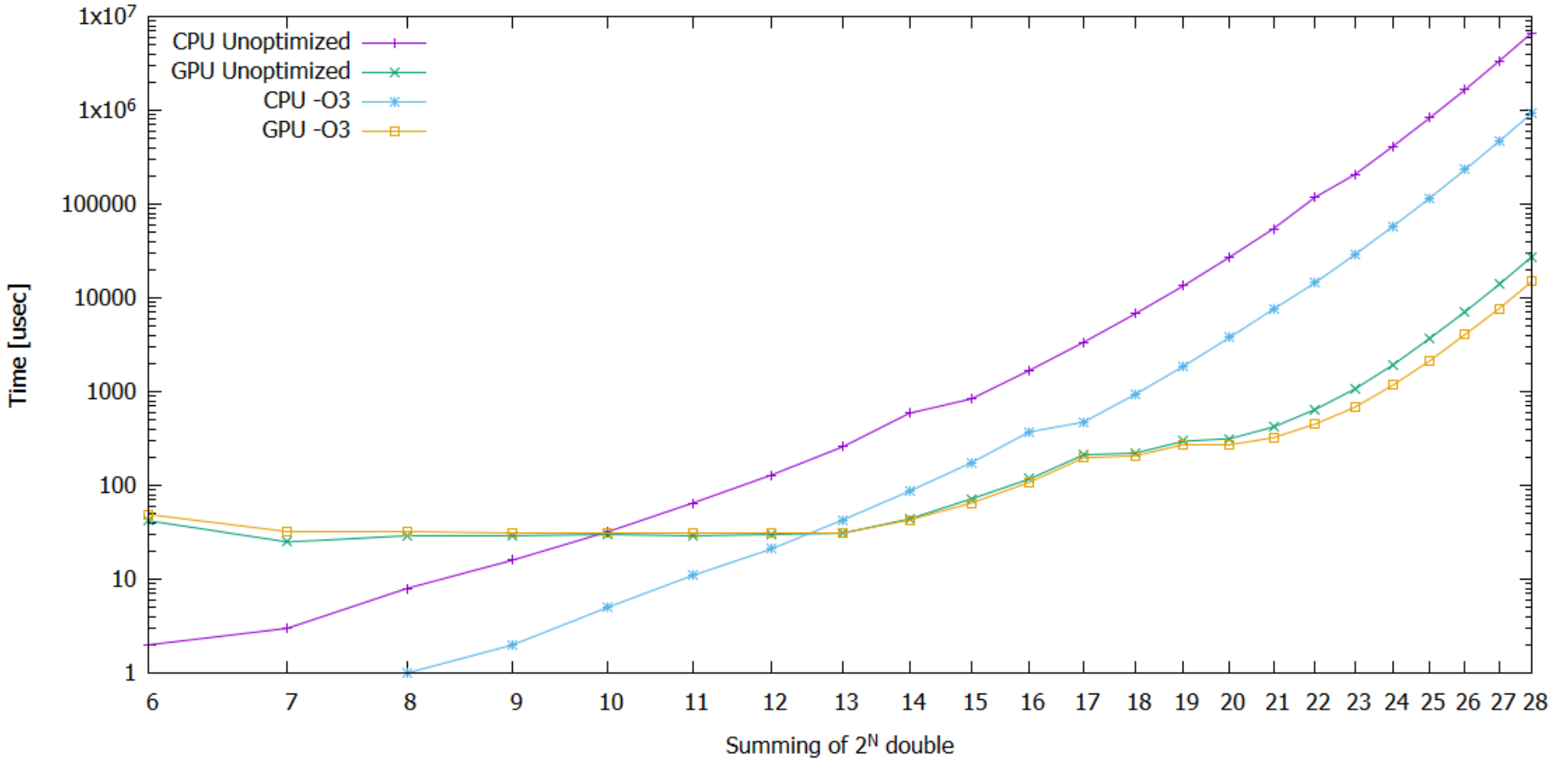
5. Parametrikus számítások GPUn

Redukció CPU-n és GPU-n

CUDA verzió, GeForce GTX 980:



ODE Megoldás

CUDA verziók (Lotka-Volterra, Van der Pol oszcillátor)
Adaptív lépésköz vezérlés, PI control, l. még Numerical Recipes:

$$h' = s \cdot h \cdot (err^{-\alpha}) \cdot \left(err2^{\beta}\right)$$

h az aktuális lépésköz

err a mostani lépésben elkövetett hiba

err2 az előző lépésben elkövetett hiba

s egy biztonsági faktor

h' az új lépésköz

$$\alpha = \frac{1}{k} - 0.75\beta$$

$$\beta = \frac{0.4}{k}$$

ahol k a léptető rendje

ODE Megoldás

Skálázott hiba számolás a két becsült állapot között:

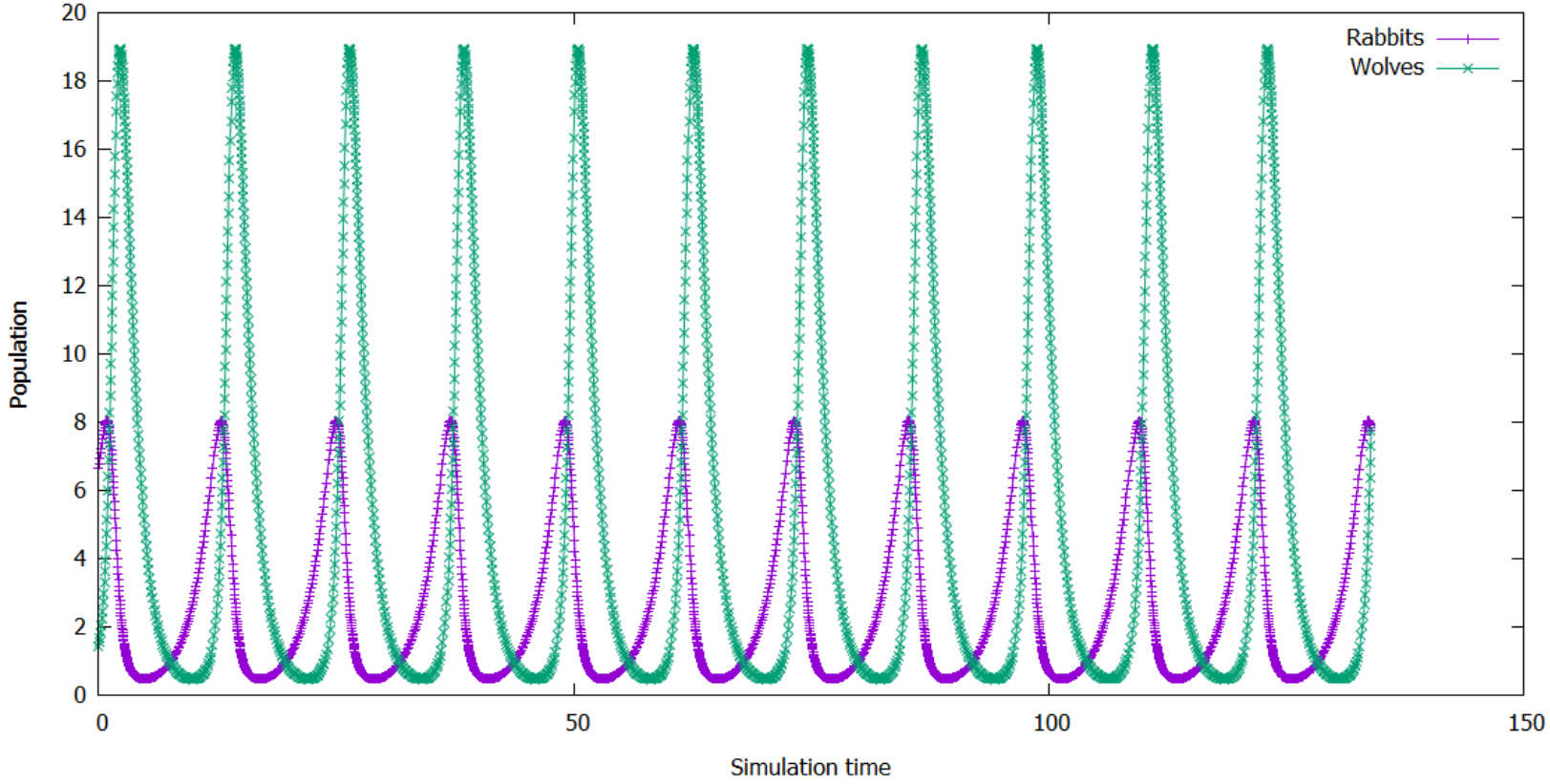
Zip ezzel a függvénnyel:

```
const auto rel_diff = [&] __device__ (auto x, auto y)
{
  auto scale = atol + rtol * max(x, y);
  return (x-y)*(x-y)/scale/scale;
};
```

ahol atol az abszolút tolerancia, rtol a relatív.

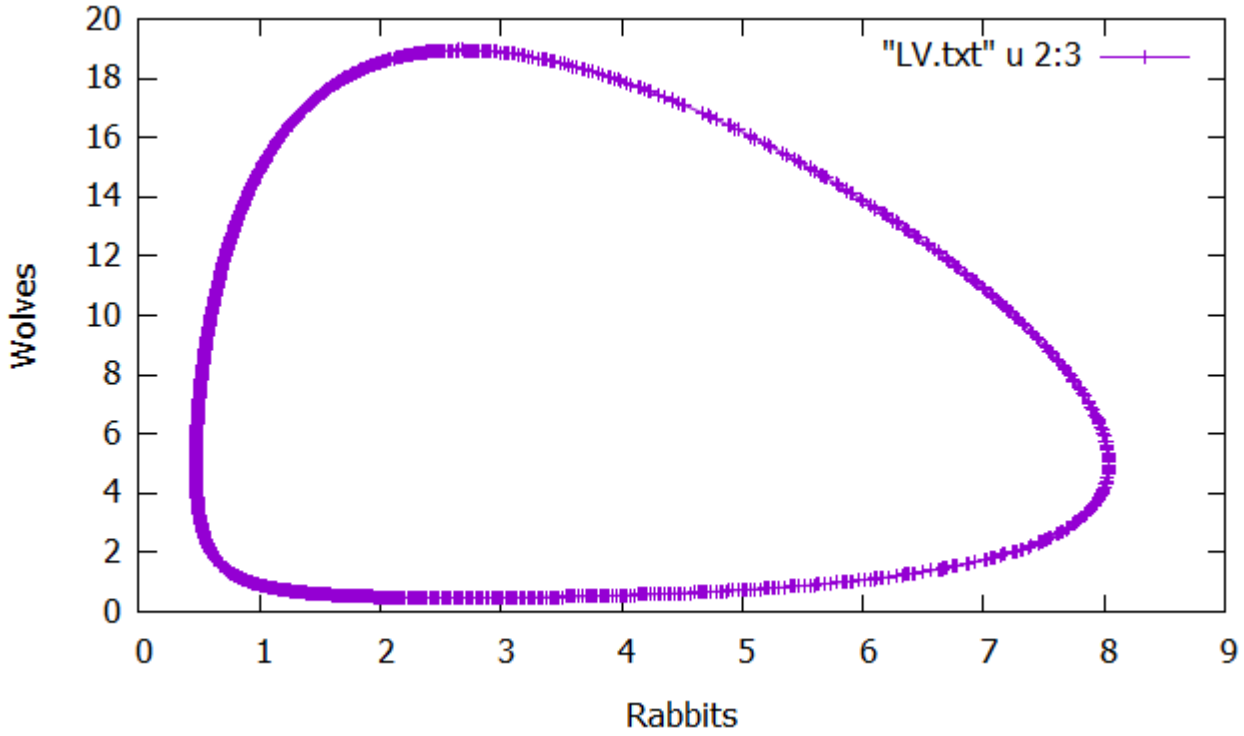
ODE Megoldás

CUDA, állapot kiírása minden sikeres lépésnél



ODE Megoldás

CUDA, állapot kiírása minden sikeres lépésnél



Csúszóablak műveletek

Tipikus példák:

- Véges differencia sémák
- Konvolúció

Csúszóablak műveletek

Véges differencia sémák:

Első derivált becslése:

$$\frac{df}{dx} = \frac{\frac{1}{2}\left(f(x+dx) - f(x-dx)\right)}{dx}$$

Második derivált becslése:

$$\frac{d^2f}{dx^2} = \frac{f(x+dx) - 2f(x) + f(x-dx)}{dx^2}$$

Csúszóablak műveletek

Véges differencia sémák:

Első derivált becslése:

```
auto diff1 = [=] (T y0, T y1, T y2)->T
{
    return 0.5*(y2-y0)/dx;
};
```

Második derivált becslése:

```
auto diff2 = [=] (T y0, T y1, T y2)->T
{
    return (y0-2.0*y1+y2)/(dx*dx);
};
```

Csúszóablak műveletek

Naív implementáció:

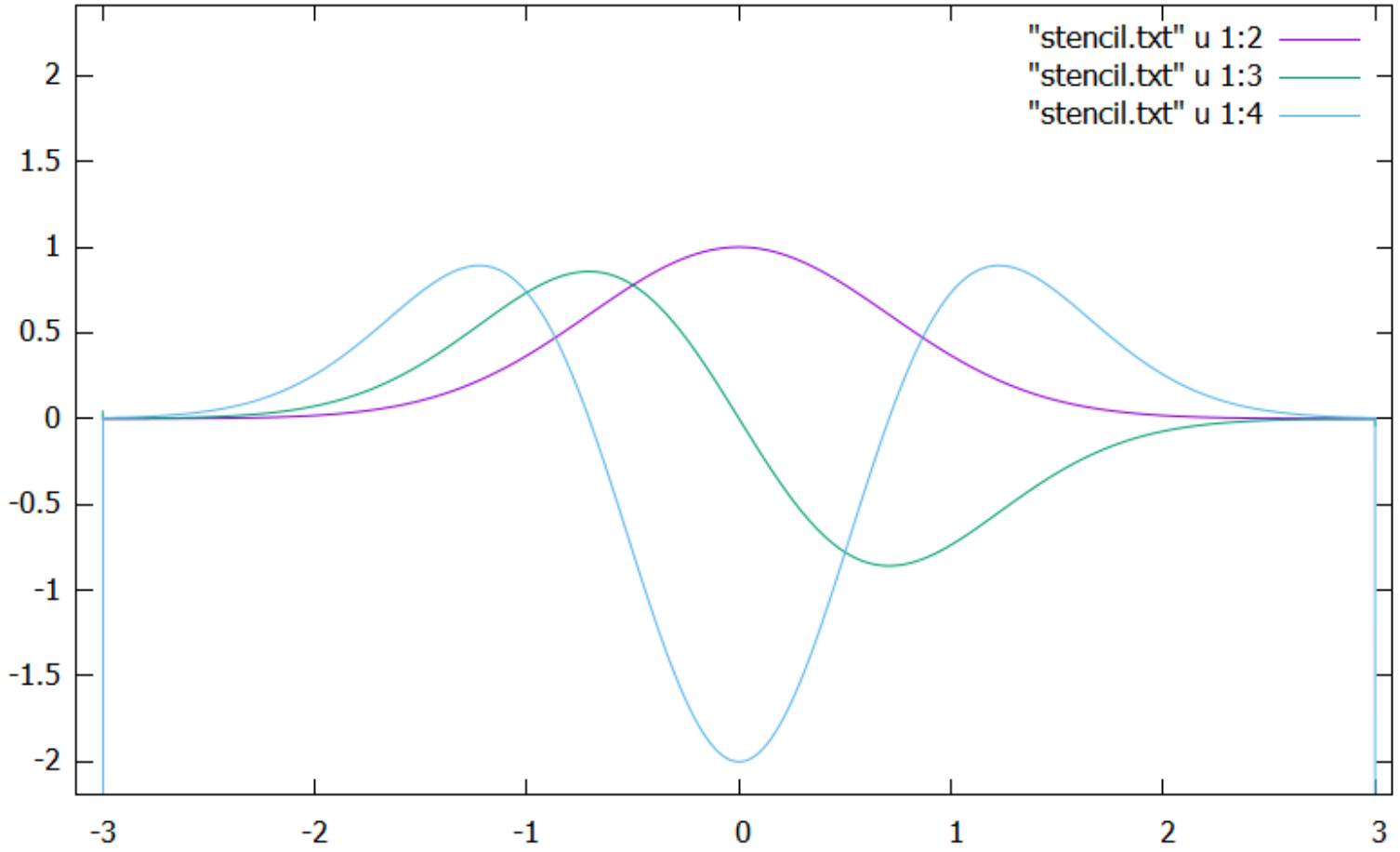
```
template<typename F, typename T, typename R>
__global__ void sliding_map_3_impl(F f, T left, T right, T* src, R* dst)
{
   int id = blockIdx.x*blockDim.x+threadIdx.x;
   int imax = blockDim.x * gridDim.x - 1;
   if     (id == 0   ){ dst[id] = f(left,        src[0   ], src[1   ]); }
   else if(id == imax){ dst[id] = f(src[imax-1], src[imax], right    ); }
   else               { dst[id] = f(src[id  -1], src[id  ], src[id+1]); }
}
```

Csúszóablak műveletek

Mi legyen a határokon???

- Vagy periodikus a tartomány,
- Vagy ha nem, akkor:
  - Vagy extra értékekkel körben ki van egészítve a tartomány
  - Vagy más fajta függvények hatnak a széleken
  - Vagy kisebb méretű lesz a kimenet, mint a bemenet!

Csúszóablak műveletek



Csúszóablak műveletek

Véges differencia sémák:

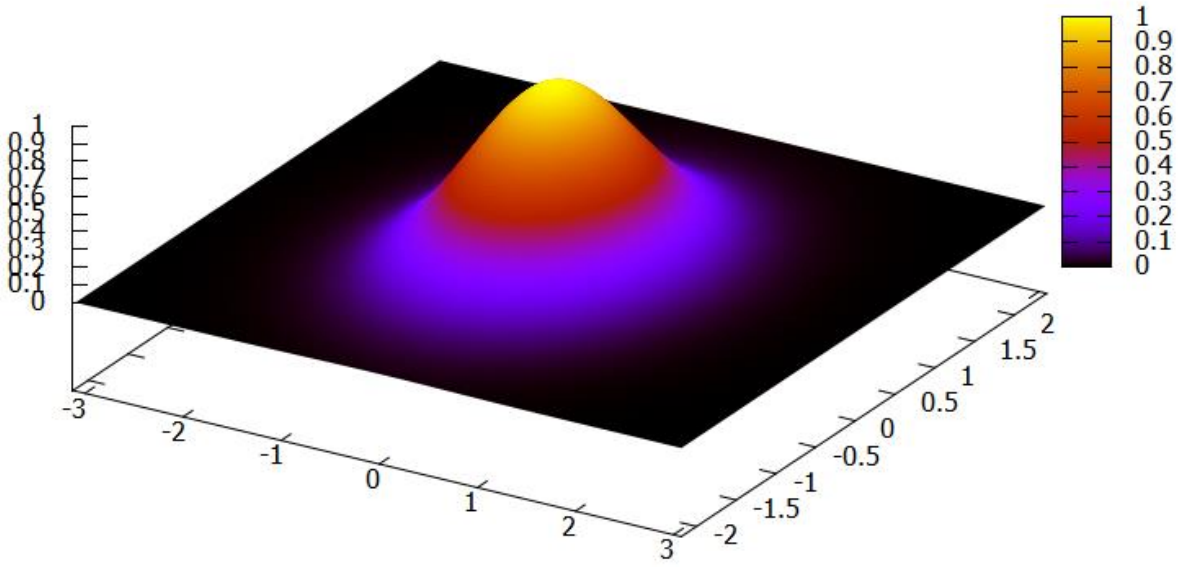
Derivált két változó szerint:

$$\frac{df}{dxdy} = \frac{\frac{1}{2}\left(f(x-dx, y-dy) - f(x+h, y-dy) + f(x+dx, y+dy) - f(x-dx, y+dy)\right)}{dx\, dy}$$

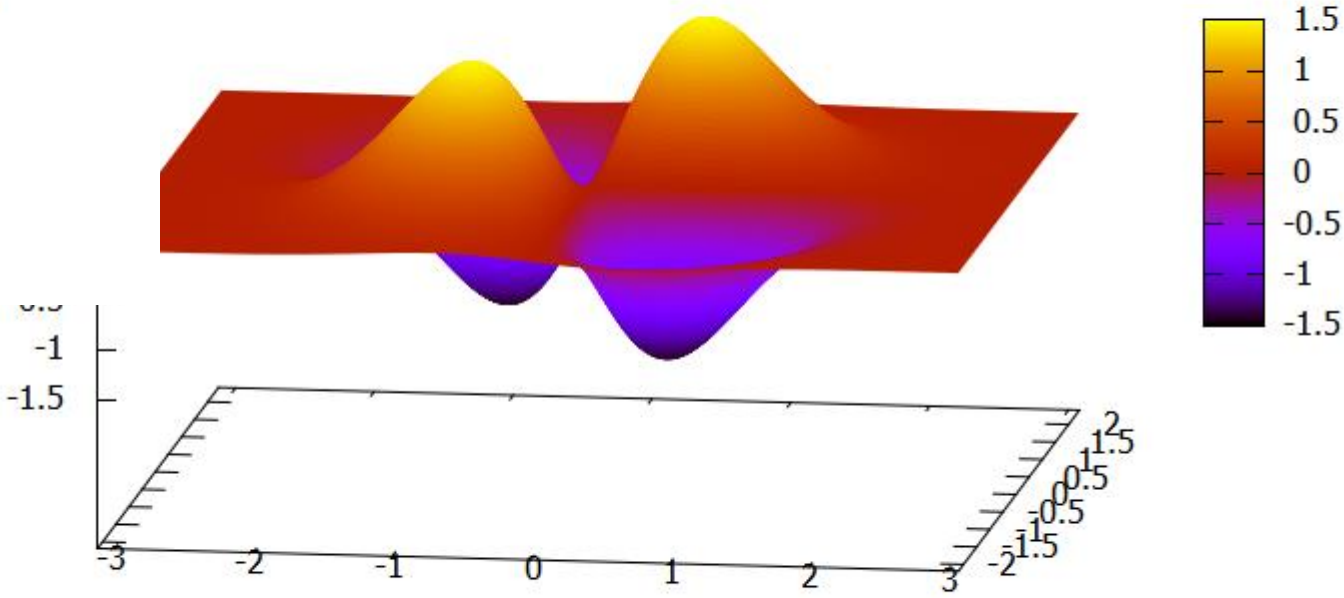
```
auto diffxy = [=] (T a00, T a01, T a02,
                   T a10, T a11, T a12,
                   T a20, T a21, T a22)->T
{
    return (a00-a20 + a22 - a02)/dx/dy/2.0;
};
```

Csúszóablak műveletek

Eredeti gauss függvény



Első derivált x és y szerint is:



Csúszóablak műveletek

Véges differencia sémák:

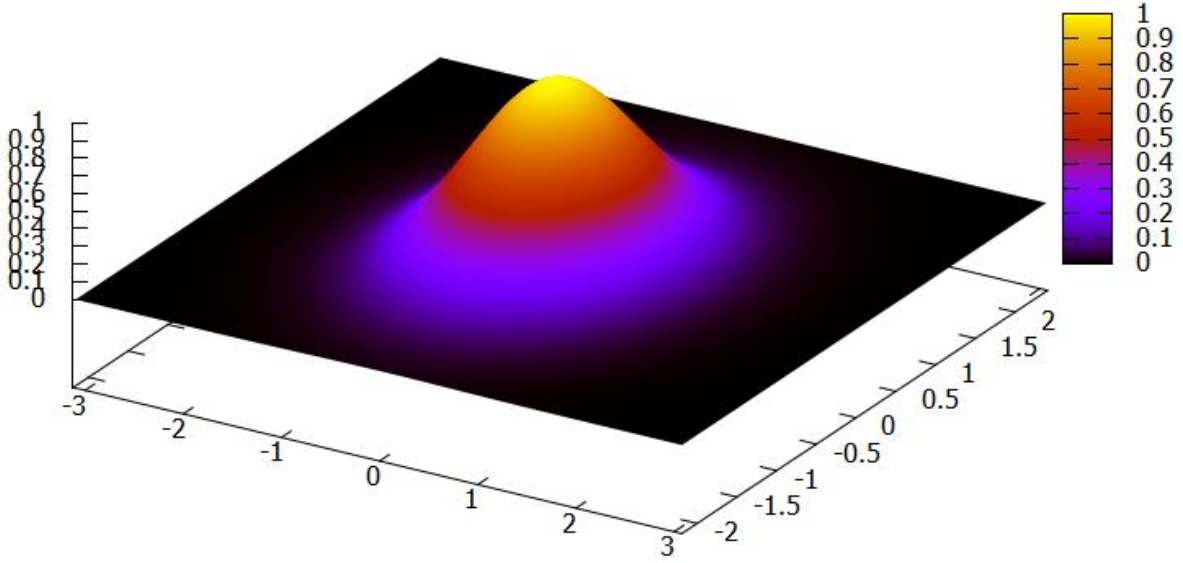
Laplace operátor:

$$\left(\frac{d^2}{dx^2}+\frac{d^2}{dy^2}\right)f(x,y)=\frac{1}{6}\frac{1}{dxdy}\cdot$$
$$f(x-dx,y-dy)+f(x-dx,y+dy)+f(x+dx,y-dy)+f(x+dx,y+dy)$$
$$+4[f(x,y-dy)+f(x,y+dy)+f(x-dx,y)+f(x+dx,y)]-20f(x,y)$$

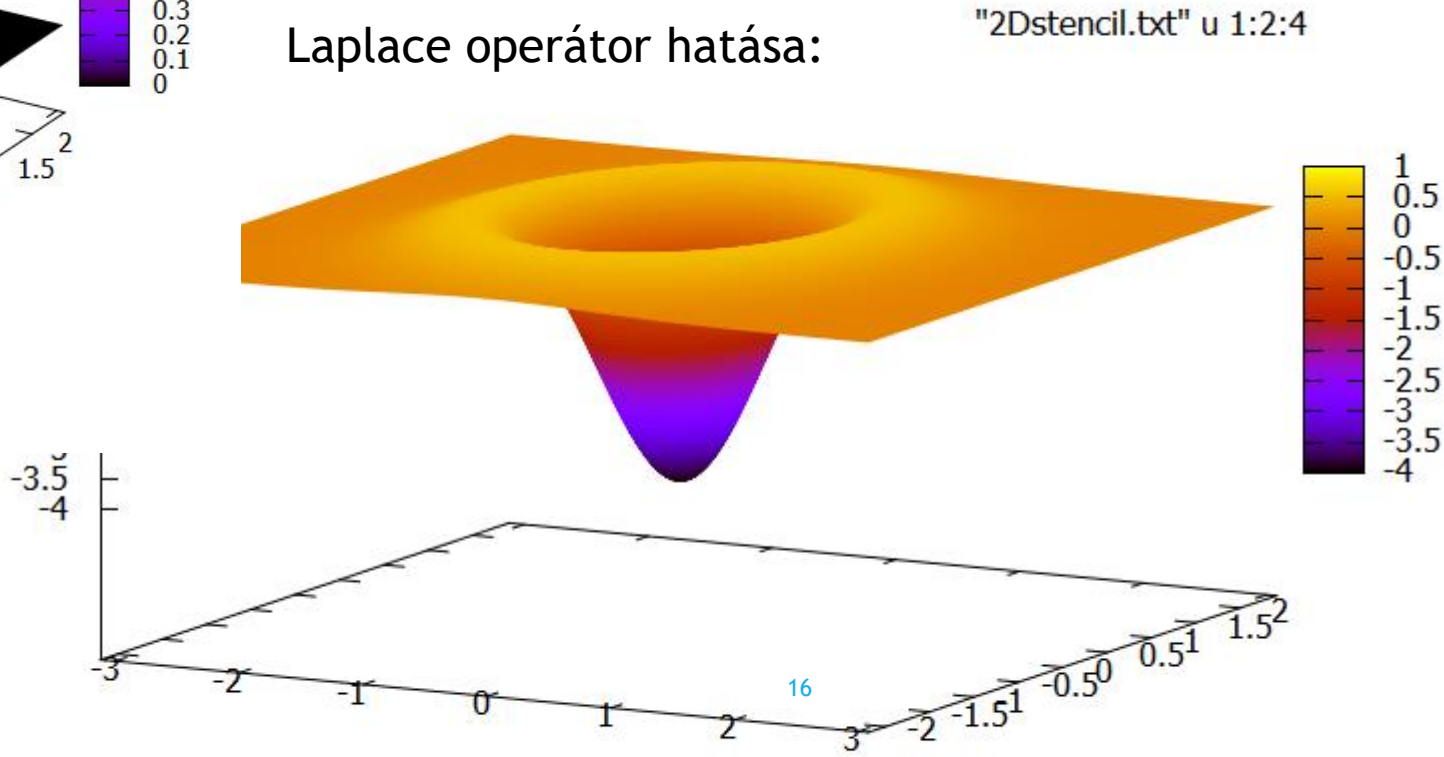
```
auto laplacian = [=] (T a00, T a01, T a02,
                      T a10, T a11, T a12,
                      T a20, T a21, T a22)->T
{
   return (a00+a02+a20+a22 + 4.0*(a10+a01+a12+a21)-20*a11)/dx/dy/6.0;
};
```

Csúszóablak műveletek

Eredeti gauss függvény



Laplace operátor hatása:



Monte-Carlo integrálás

- Szálankénti random szám generálás: kicsi állapot kell!
- Lehmer generátor:

```
double uniform_rnd(long& state)
{
    const long A = 48271;      /* multiplier*/
    const long M = 2147483647; /* modulus */
    const long Q = M / A;      /* quotient */
    const long R = M % A;      /* remainder */
    long t = A * (state % Q) - R * (state / Q);
    if (t > 0){ state = t;     }
    else      { state = t + M; }
    return ((double) state / M);
}
```

Monte-Carlo integrálás

A Monte-Carlo integrálás képlete alapján:

$$\int f(x)dx = V\frac{1}{N}\sum_{i}^{N} f(x_i)$$

Ahol x_i az integrálási tartományon dobott véletlen szám, és N mintát vettünk a tartományból.

De kell V is! Ha van egy befoglaló intervallumunk V_0, és egy maszkoló függvényünk $C(x)$: `T -> {0, 1}`, és K-szor próbálkozunk:

$$V = V_0\frac{1}{K}\sum_{i}^{K} C(x)$$

Monte-Carlo integrálás

Viszont, mivel a függvényt csak akkor tudjuk kiértékelni, ha C 1-et ad, ezért:

$$N = \sum_{i}^{K} C(x)$$

Ami visszaírva:

$$\int f(x)dx = \frac{V_0}{K} \sum_{i,C(x_i)=1}^{K} f(x_i)$$

Monte-Carlo integrálás

SYCL példakód:

```
cgh.parallel_for<class MCIKernel>(r, [=](cl::sycl::nd_item<1> id)
{
  auto g = id.get_group().get(0);
  auto bs = id.get_local_range().get(0);
  auto l = id.get_local().get(0);

  long q0 = (long)(15348919 ^ (7+g));
  long state = (long)(2147483647 * uniform_rnd(q0));
  double t = 0.0; size_t n = 0;
  while(n < thcount){
      double x = uniform_rnd(state) * (x1-x0) + x0;
      double y = uniform_rnd(state) * (y1-y0) + y0;
      if(mask(x, y)){ t += f(x, y); }
      n += 1;
  }
  loc[l] = t / N;
  //Innentől a szoksásos redukció...
```

Monte-Carlo integrálás

Összehasonlítás: 2^35 minta, 2D kör integrálja négyzeten belül

C++ Mersenne Twister CPU-n:

38 perc, eltérés: 0.0000038401290197

Lehmer CPU-n:

7.6 perc, eltérés: 0.0000000162166743

Lehmer, SYCL GPU-n, 16384 szál:
15 sec, eltérés: 0.0000113260987873

Ising model

Adott egy négyzetrácsunk, rajta $\sigma = \{-1, +1\}$ spinekkel és a következő Hamiltonnal:

$$\mathcal{H} = J \sum_{\{ij\}} \sigma_i \sigma_j$$

ahol ij minden rácspontra megy, j pedig a legközelebbi szomszédokra (4 db). J a spinek közötti kölcsönhatás erőssége.

Határozzuk meg az egyensúlyi eloszlását a spineknek!

Ising model

Ez is egy monte-carlo szimuláció, metropolis eljárással:

Referencia módszer: 1 spin változtatása egy időben

1. Válasszunk ki egy random spint
2. Számoljuk ki az energiáját az aktuális helyzetében és megfordítva az állapotát:
   ```
   auto dE = J * (Eflip - Ecurrent);
   ```
3. Fogadjuk el az új állapotot, ha a megfordítás előnyös, vagy ha az átmenet valószínűségének megfelelő számot dobtunk:

```
if( dE <= 0.0 || exp(-dE*beta) > uniform_rnd(state) )
{
   grid[y1*N+x1] = (T)1 - c0;
}
```

Ising model

Megfigyelhető mennyiségek:

Mágnesezettség (átlagos spin):

$$\langle M \rangle = \frac{J}{N} \sum_i \sigma_i$$

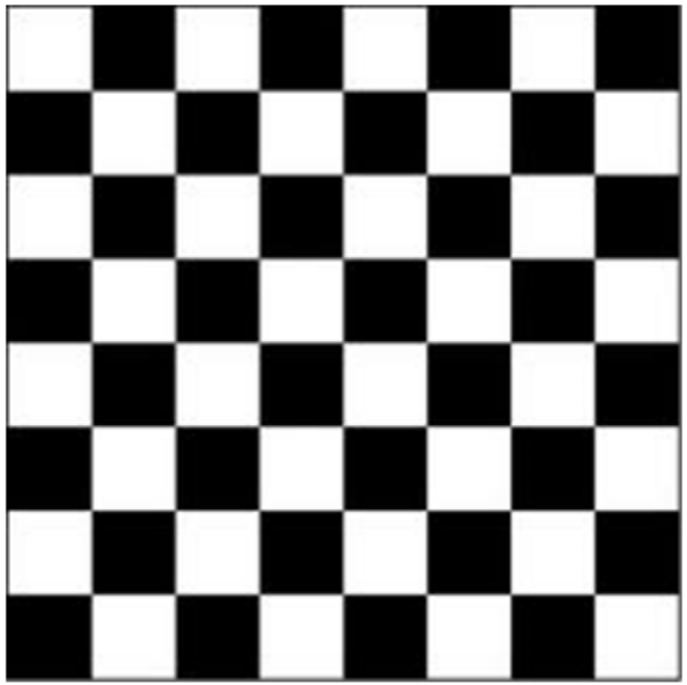
Átlagos energia:

$$\langle U \rangle = -\frac{J}{2N} \sum_{\{ij\}} \sigma_i \sigma_j$$

Ising model

GPU-s változat:

Az egy spin változtatás nem párhuzamosítható, fordítsunk több spint egyszerre. A lehető legtöbbet->sakktábla update:



Egy lépésben az összes azonos színű spint tudjuk frissíteni, mert az ő értékük csak a másik színű spinektől függ!

Ising model

```
size_t x = id.get(0), y0 = id.get(1), xlo, xhi, ylo, yhi; long pstate = rng[y0*N+x];

for(int eo0=0; eo0<1; ++eo0){
  auto eo = (even_odd + eo0) % 2;
  auto y = y0 * 2 + ((int)(x+eo) % 2); //checkerboard on global level
  make_periodic(x, N, xlo, xhi); make_periodic(y, N, ylo, yhi);

  for(int k=0; k<nsteps; ++k){

    float sum = src[xlo][y] + src[xhi][y] + src[x][ylo] + src[x][yhi];
    float cell = src[x][y];
    float Ecurrent = -cell * sum, Eflip = +cell * sum;
    auto dE = J * (Eflip - Ecurrent);
    if( dE <= 0.0 || cl::sycl::exp(-dE*beta) > uniform_rnd(pstate) )
    { src[x][y] = -cell; }
  }

}
```

Ising model

```
size_t x = id.get(0), y0 = id.get(1), xlo, xhi, ylo, yhi; long pstate = rng[y0*N+x];

for(int eo0=0; eo0<1; ++eo0){
  auto eo = (even_odd + eo0) % 2;
  auto y = y0 * 2 + ((int)(x+eo) % 2); //checkerboard on global level
  make_periodic(x, N, xlo, xhi); make_periodic(y, N, ylo, yhi);

  for(int k=0; k<nsteps; ++k){

    float sum = src[xlo][y] + src[xhi][y] + src[x][ylo] + src[x][yhi];
    float cell = src[x][y];
    float Ecurrent = -cell * sum, Eflip = +cell * sum;
    auto dE = J * (Eflip - Ecurrent);
    if( dE <= 0.0 || cl::sycl::exp(-dE*beta) > uniform_rnd(pstate) )
    { src[x][y] = -cell; }
  }

}
```

Itt történik a sakktábla létrehozása!

Ising model

Fontos dolgok:

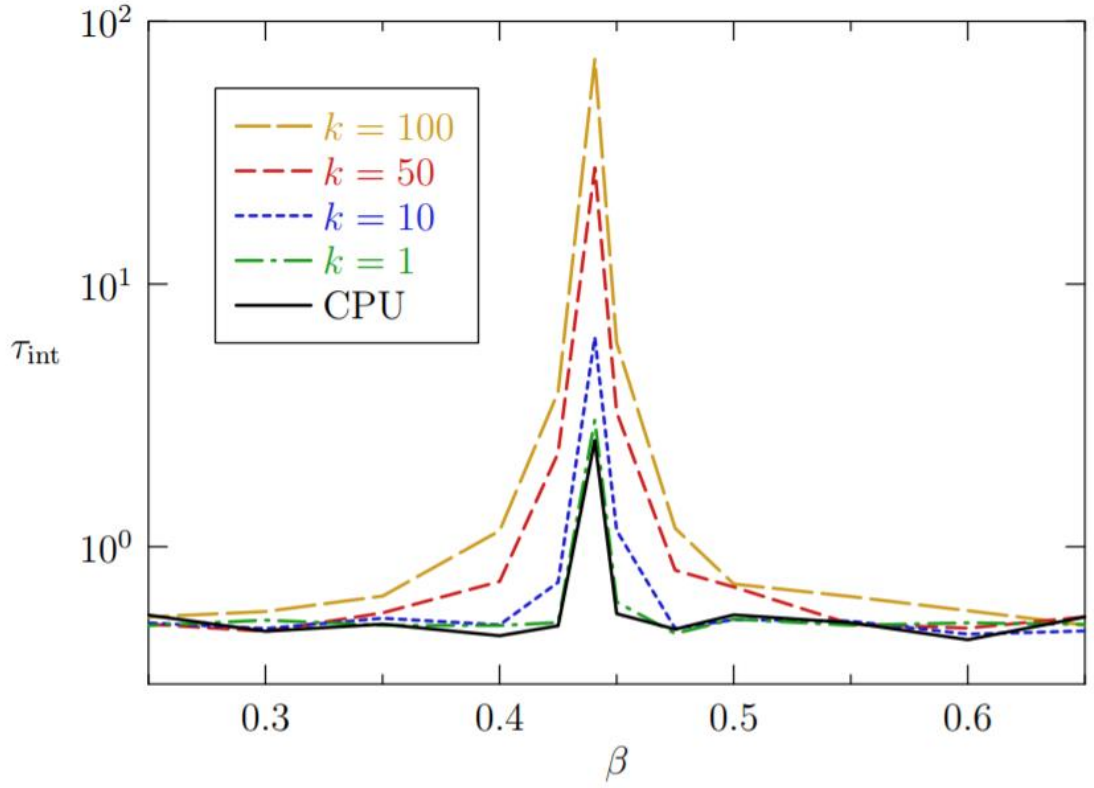
- N/2 db szál dolgozik
- Minden alrácson (fél sakktáblán) több updatet is végzünk
- Majd áttérünk a másik rácsra

- Ez a módszer elrontja a részletes egyensúlyt! Az csak átlagban teljesül! Ezért így érdemes léptetni és mérni (ahol M a mérés A és B a két alrács update-je):

AAAA(M)AAAABBBB(M)BBBBAAAA(M)AAAABBBB(M)BBBB

Ising model

A kritikus hőmérséklet közelében ez a módszer sokkal lassabban állítja elő az egyensúlyi eloszlást!



Forrás illetve további olvasni való!

Ising model

Teljesítmény (nanoszekundum / spin-flip):

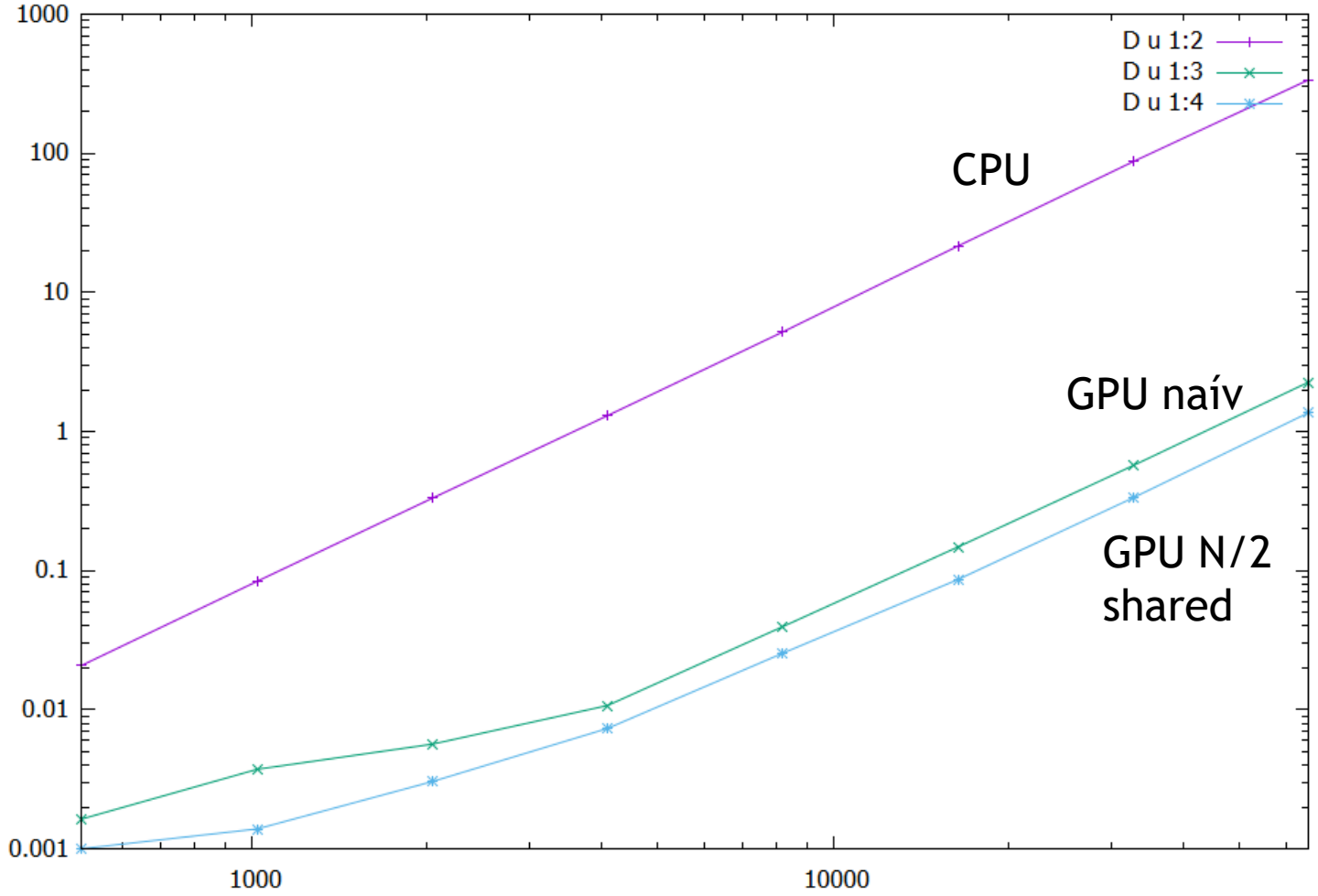
méret	CPU	GPU
32	29	23
64	30	5.7
128	35	1.7
256	37	1.2
512	41	0.96
1024	52	0.88

N test szimuláció

- Parametrikus kölcsönhatás, de feltételezzük, hogy szimmetrikus
- Csak erőt számolunk (az a költséges)
- CPU referencia változat
- GPU naív implementáció (itt nehéz az N/2!)
- GPU lokális memóriás változat

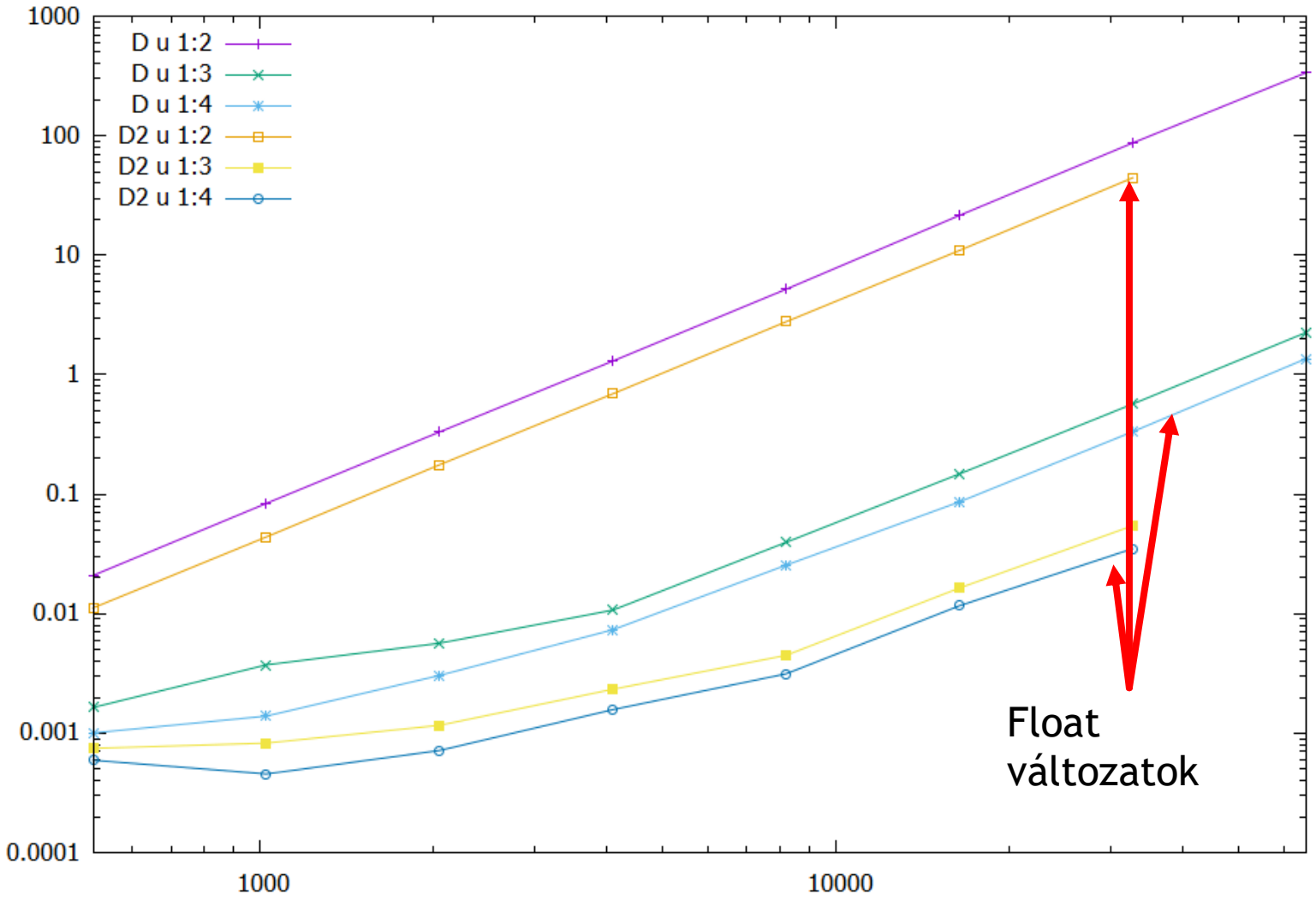
N test szimuláció

Double precision



N test szimuláció

Számít, hogy float, vagy double!

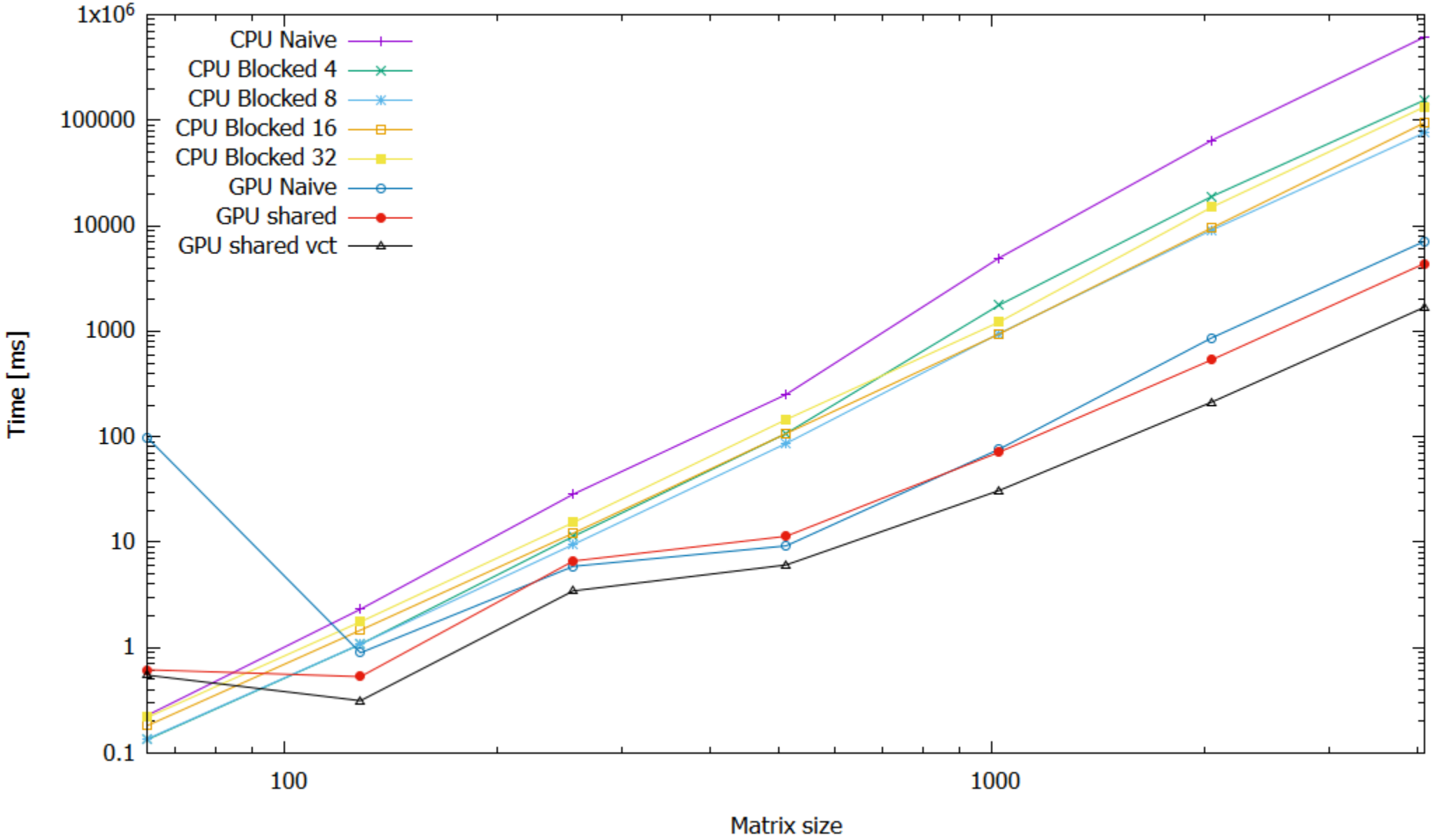


Mátrix-vektor szorzás

Naív és különböző blokkosítású változatok.

Mátrix-mátrix szorzás

Naív, shared és vektorizált változat



További funkcionális primitívek

Számos esetben kell olyan ciklusokat használnunk, amelyekről nem tudjuk, hogy hányszor fognak lefutni:

- Runge-Kutta
  - Léptetés a megállási feltételig
  - Úlyapróbálkozás, ha az aktuális lépés hibája túl nagy
- Iteratív/adaptív módszerek
  - Finomítás, amíg egy hibahatáron belülre kerülünk

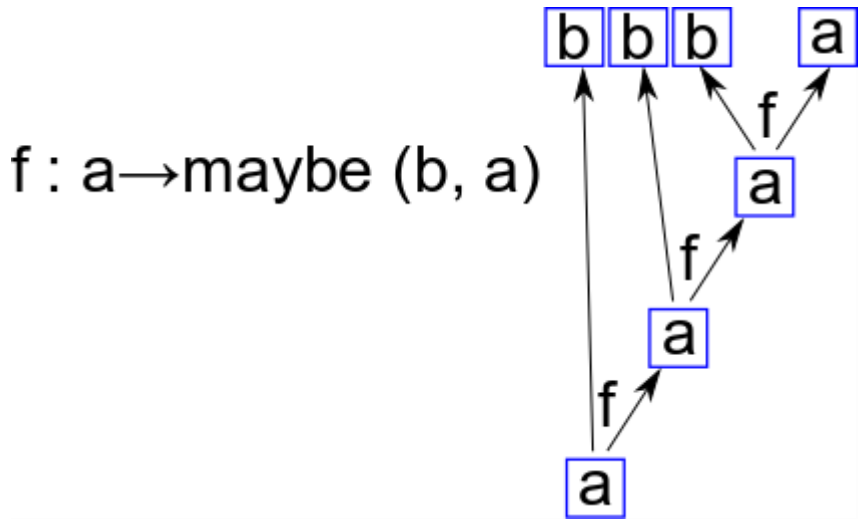
Ezeket általában while ciklusokkal oldjuk meg... Ez nem komponálható!

Nincs erre valami funkcionális konstrukció?

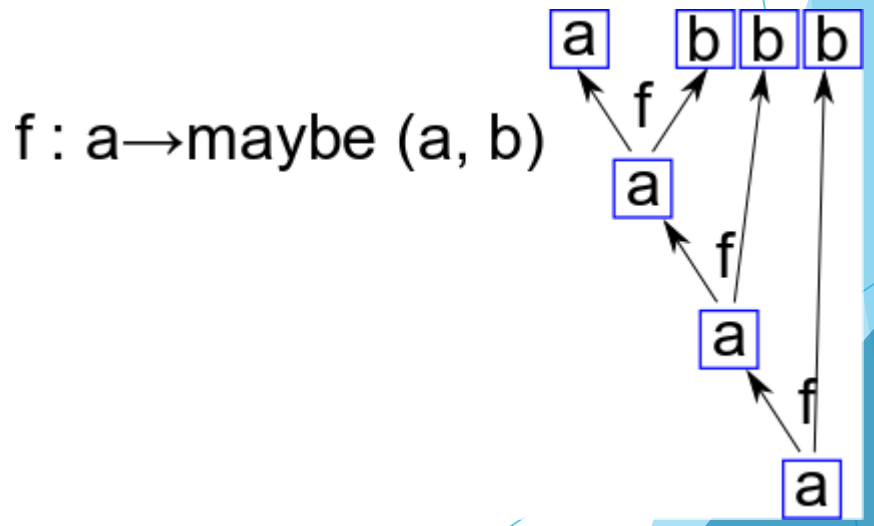
További funkcionális primitívek

While ciklusok... Nincs erre valami funkcionális konstrukció?

```
unfoldr :: (a->maybe (b, a)) -> a -> f b
```




```
unfoldl :: (a->maybe (a, b)) -> a -> f b
```



További funkcionális primitívek

Az unfoldokal szekvenciálisan állíthatunk elő (járhatunk be) kollekciókat.

```
unfoldr :: (a->maybe (b, a)) -> a -> f b

unfoldl :: (a->maybe (a, b)) -> a -> f b
```

Példák: fájlból elemek beolvasása, rekurzív kiértékelés (fibbonacci), egyik tárolóból a másikba átalakítás, scan, stb.

További funkcionális primitívek

Egy unfold és egy fold együtt nagyon hasznos konstrukciót eredményez:

A hylomorfizmust:

```
unfoldl :: (a->maybe (a, b)) -> a -> f b
foldr :: (b->c->c) -> c -> f b -> c

hylo :: (a->maybe (b, a)) -> (b->c->c) -> a -> c -> c
```

További funkcionális primitívek

A hylomorfizmus: `hylo :: (a->maybe (b, a)) -> (b->c->c) -> a -> c -> c`

```
template<typename UF, typename FF, typename S, typename Z>
auto hylo( UF uf, FF ff, S seed, Z zero ){
    auto maybe_val_and_seed = uf( seed );
    Z acc = zero;
    while( maybe_val_and_seed ){
        acc = ff( acc, maybe_val_and_seed.value.l );
        maybe_val_and_seed = uf( maybe_val_and_seed.value.r );
    }
    return acc;
}
```

További funkcionális primitívek

Alkalmazás: Newton iteráció

```
template<typename F, typename dF, typename S, typename T>
auto NewtonIterator( F f, dF df, S const& start, T const& tolerance, int nmaxit )
{
  return hylo([=](Pair<T, int> const& xn_n)
        {
          auto xnn = xn_n.l - f(xn_n.l)/df(xn_n.l);
          int n = xn_n.r + 1;
          return maybe(cl::sycl::fabs(xnn-xn_n.l)*2 > tolerance && n < nmaxit,
                       [=]{ return makePair(makePair(xnn, n), makePair(xnn, n)); });
        }, [](auto xn, auto xnn){ return xnn; }, start, makePair(0.0, 0) );
}
```

SYCL – OpenGL Interop

Amire figyelni kell:
Létrehozásnál OpenCL-APIval hozzuk létre a contextet, beállítva az OpenGL használatot, ahogy tavaly láttuk, majd ebből kell a SYCL objektumokat létrehozni:

```
auto glContext = wglGetCurrentContext();

auto gldc      = wglGetCurrentDC();

cl_context_properties cps[] =
{ CL_GL_CONTEXT_KHR,   (cl_context_properties) glContext,
  CL_WGL_HDC_KHR,      (cl_context_properties) gldc,
  CL_CONTEXT_PLATFORM, (cl_context_properties) platform,  0 };

cl_context context = clCreateContext(cps, 1, &device, 0, 0, &status);

cl::sycl::context sycl_context(context);

queue = clCreateCommandQueueWithProperties(context, device, nullptr, &status);

cl::sycl::queue sycl_queue(queue, sycl_context);
```

SYCL – OpenGL Interop

Amire figyelni kell:
Létrehozásnál OpenCL-APIval hozzuk létre a contextet, beállítva az OpenGL használatot, ahogy tavaly láttuk, majd ebből kell a SYCL objektumokat létrehozni:

```
auto glContext = wglGetCurrentContext();

auto gldc      = wglGetCurrentDC();

cl_context_properties cps[] =
{ CL_GL_CONTEXT_KHR,   (cl_context_properties) glContext,
  CL_WGL_HDC_KHR,      (cl_context_properties) gldc,
  CL_CONTEXT_PLATFORM, (cl_context_properties) platform,  0 };

cl_context context = clCreateContext(cps, 1, &device, 0, 0, &status);

cl::sycl::context sycl_context(context);

queue = clCreateCommandQueueWithProperties(context, device, nullptr, &status);

cl::sycl::queue sycl_queue(queue, sycl_context);
```

Linuxon ezek mások!

SYCL – OpenGL Interop

Amire figyelni kell:
A megosztott buffereket először OpenGL-ben hozzuk létre, majt utána szintén az OpenCL APIval csinálunk cl_buffert, majd abból SYCL buffert:

```
template<typename T>
auto create_from_gl_buffer(cl_mem_flags const& flags, GLuint glid )
{
   cl_int status;
   auto clid = clCreateFromGLBuffer(context, CL_MEM_READ_WRITE, glid, &status );
   return cl::sycl::buffer<T, 1>(clid, sycl_queue, cl::sycl::event());
}
```

Vizsgára

- Fogalmak a diasorokról
- Lambda kalkulus számítások, hány eleme van egy összetett típusnak (összeg, szorzat, hatvány/függvény típuskombinátorok)
- Matematikai formulák átírása funkcionális primitívekre (map, fold, zip)
- SYCL – CUDA összehasonlítás
- Példakódok működése
- Optimalizációk (shared memória, vektorizáció) működése
- Szinkronizációk kifejezése CUDA-ban, SYCL-ben