Wigner GPU Lab

10. fejezet

GPU Példakódok

Grafikus Processzorok Tudományos Célú Programozása

Berényi Dániel – Nagy-Egri Máté Ferenc

Mátrix szorzás OpenCL-ben

Feladat:

- Csináljunk mátrix szorzást OpenCL-ben,
- hasonlítsuk össze a CPU-s változattal!

Mátrix szorzás OpenCL-ben

Host oldali kód feladatai:

- Platform, device, context, queue, program, kernel inicializálása
- Bufferek létrehozása és feltöltése
- Kernel argumentumok beállítása
- Kernelek futtatása
- Bufferek visszamásolása

Mátrix szorzás OpenCL-ben

```
int main()
{
    //Az inicializáció olyan,
    //mint a múlt órán mutatott...
    //Csak az eltéréseket emeljük ki
}
```

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

Szeretnénk időt mérni a GPU oldalon is,
ehhez a Command Queue-nél be kell kapcsolni a profiling-ot:

```
cl_command_queue_properties cqps = CL_QUEUE_PROFILING_ENABLE;

cl_queue_properties qps[] = { CL_QUEUE_PROPERTIES, cqps, 0 };

auto queue =
clCreateCommandQueueWithProperties(context, device, &qps[0], &status);
```

Mátrix szorzás OpenCL-ben

A példakódban generálunk 2 db 1024 x 1024-es random mátrixot, amiket 3 féle képpen szorzunk össze:

- A naív CPU-s implementációval
- A naív GPU-s implementációval
- Egy kicsit szofisztikáltabb GPU-s implementációval

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

- A mátrixok generálása:

```
constexpr int size = 1024;
std::vector<double> A(size*size), B(size*size);
std::random_device rnd_device;
std::mt19937 rnd_engine(rnd_device());
std::uniform_real_distribution<double> dist(-1.0, 1.0);
auto gen = [&]() mutable { return dist(rnd_engine); };
std::generate(A.begin(), A.end(), gen );
std::generate(B.begin(), B.end(), gen );
```

Mátrix szorzás OpenCL-ben

WIGNER GPU Lab

- A naív CPU-s implementáció:

```
auto tp1 = std::chrono::high_resolution_clock::now();
for(int i=0; i<size; ++i)
{
    for(int j=0; j<size; ++j)
    {
        auto acc = 0.0;
        for(int k=0; k<size; ++k){ acc += A[i*size+k] * B[k*size+j]; }
        E[i*size+j] = acc;
    }
}
auto tp2 = std::chrono::high_resolution_clock::now();
auto tnaive =
        std::chrono::duration_cast<std::chrono::microseconds>(tp2-tp1).count()/1000.0;
```

Mátrix szorzás OpenCL-ben

WIGNER GPU Lab

- A naív GPU-s implementáció:

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

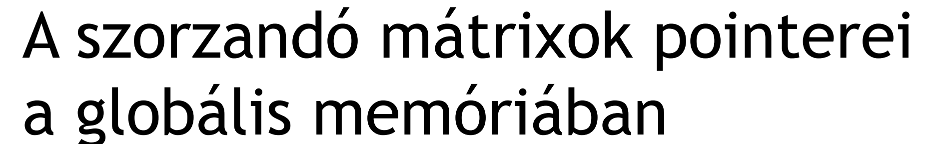
- A naív GPU-s implementáció:

```
__kernel void matmul0(__global double* A, __global double* B,
                      __global double* C, int size)
{
   int thx = get_global_id(0);
   int thy = get_global_id(1);
   double acc = 0.0;
   for(int i=0; i<size; ++i)
   {
      acc += A[thy*size+i] * B[i*size+thx];
   }
   C[thy * size + thx] = acc;
}
```

Mátrix szorzás OpenCL-ben

- A naív GPU-s implementáció:

```
__kernel void matmul0(__global double* A, __global double* B,
                      __global double* C, int size)
{
    int thx = get_global_id(0);
    int thy = get_global_id(1);
    double acc = 0.0;
    for(int i=0; i<size; ++i)
    {
        acc += A[thy*size+i] * B[i*size+thx];
    }
    C[thy * size + thx] = acc;
}
```



Mátrix szorzás OpenCL-ben

- A naív GPU-s implementáció:

```
__kernel void matmul0(__global double* A, __global double* B,
                      __global double* C, int size)
{
   int thx = get_global_id(0);
   int thy = get_global_id(1);
   double acc = 0.0;
   for(int i=0; i<size; ++i)
   {
      acc += A[thy*size+i] * B[i*size+thx];
   }
   C[thy * size + thx] = acc;
}
```

Mátrix szorzás OpenCL-ben

WIGNER GPU Lab

- A naív GPU-s implementáció:

```
__kernel void matmul0(__global double* A, __global double* B,
                      __global double* C, int size)
{
   int thx = get_global_id(0);
   int thy = get_global_id(1);
   double acc = 0.0;
   for(int i=0; i<size; ++i)
   {
      acc += A[thy*size+i] * B[i*size+thx];
   }
   C[thy * size + thx] = acc;
}
```

Szálazaonosítók 2D-ben, az első az első ciklust váltja ki, a második a másodikat a naív CPU-s implementációval összehasonlítva

Mátrix szorzás OpenCL-ben

GPU Lab

- A naív GPU-s implementáció:

```
__kernel void matmul0(__global double* A, __global double* B,
                      __global double* C, int size)
{
   int thx = get_global_id(0);
   int thy = get_global_id(1);
   double acc = 0.0;
   for(int i=0; i<size; ++i)
   {
      acc += A[thy*size+i] * B[i*size+thx];
   }
   C[thy * size + thx] = acc;
}
```

Csak a legbelső ciklus marad meg egy-egy szálnak, ez egy skaláris szorzatot számol ki, ami egy elem lesz az eredmény mátrixban

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

- A naív GPU-s implementáció a host oldalon:

```
auto bufferA = clCreateBuffer(...);
auto bufferB = clCreateBuffer(...);
auto bufferC = clCreateBuffer(...);

clSetKernelArg(kernel1, 0, sizeof(bufferA), &bufferA);
clSetKernelArg(kernel1, 1, sizeof(bufferB), &bufferB);
clSetKernelArg(kernel1, 2, sizeof(bufferC), &bufferC);
clSetKernelArg(kernel1, 3, sizeof(int), &size);
```

Mátrix szorzás OpenCL-ben

- A kernelek futásidejének lemérése:

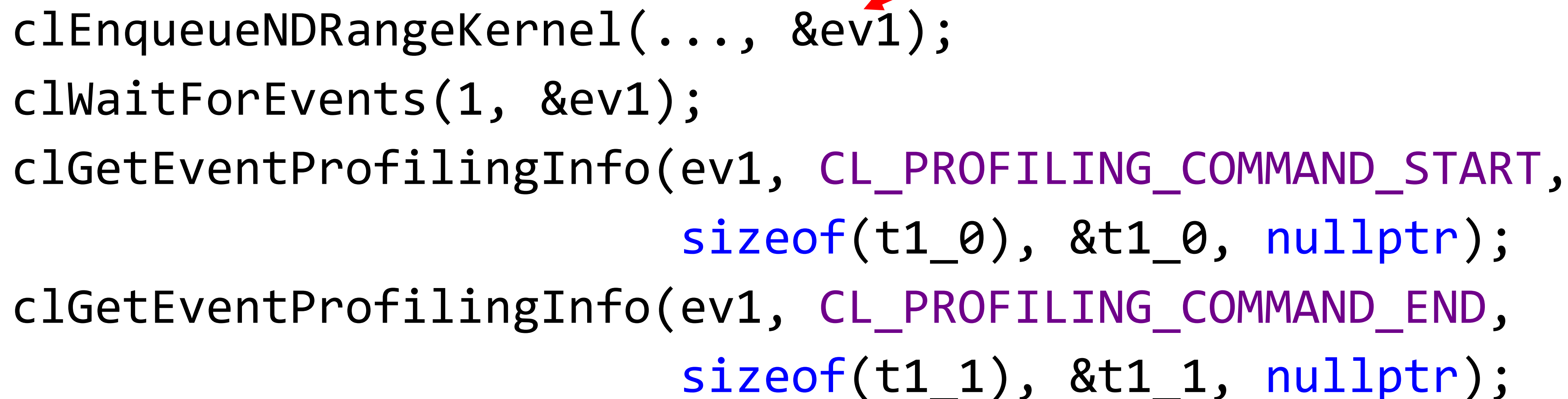
```
cl_event ev1;
cl_ulong t1_0, t1_1;

clEnqueueNDRangeKernel(..., &ev1);
clWaitForEvents(1, &ev1);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_START,
                        sizeof(t1_0), &t1_0, nullptr);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_END,
                        sizeof(t1_1), &t1_1, nullptr);
```

Wigner GPU Lab

- A kernelek futásidejének lemérése:

Az ismert módon elindítjuk a kernelt, DE most az utolsó argumentum segítségével eltároljuk az eventet, ami azonosítja ezt az elindított számolást!

```
cl_event ev1;
cl_ulong t1_0, t1_1;
```

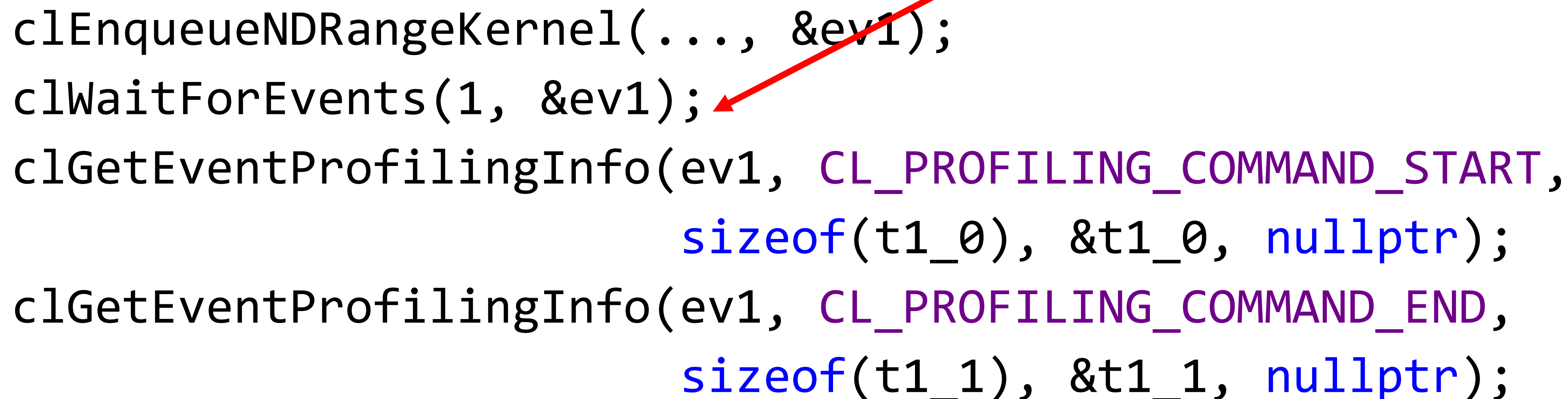


```
clEnqueueNDRangeKernel(..., &ev1);
clWaitForEvents(1, &ev1);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_START,
                        sizeof(t1_0), &t1_0, nullptr);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_END,
                        sizeof(t1_1), &t1_1, nullptr);
```

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

- A kernelek futásidejének lemérése:



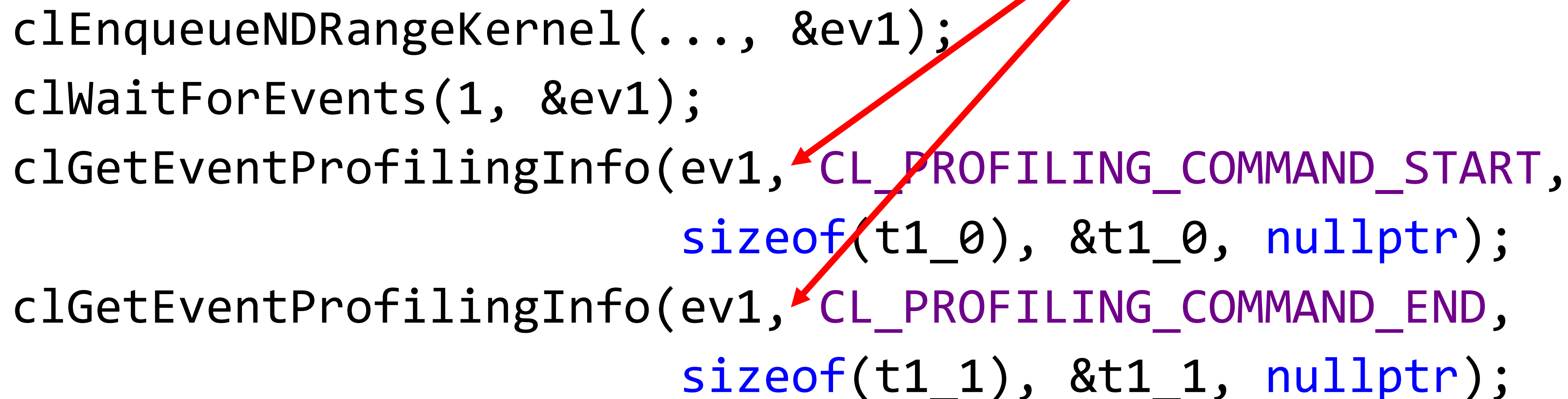

```
cl_event ev1;
cl_ulong t1_0, t1_1;

clEnqueueNDRangeKernel(..., &ev1);
clWaitForEvents(1, &ev1);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_START,
                        sizeof(t1_0), &t1_0, nullptr);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_END,
                        sizeof(t1_1), &t1_1, nullptr);
```

Mátrix szorzás OpenCL-ben

- A kernelek futásidejének lemérése:

A befejeződött event-től meg lehet kérdezni időpontokat, amik különbsége nanosec-ben van mérve.

```
cl_event ev1;
cl_ulong t1_0, t1_1;
```



```
clEnqueueNDRangeKernel(..., &ev1);
clWaitForEvents(1, &ev1);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_START,
                        sizeof(t1_0), &t1_0, nullptr);
clGetEventProfilingInfo(ev1, CL_PROFILING_COMMAND_END,
                        sizeof(t1_1), &t1_1, nullptr);
```

Mátrix szorzás OpenCL-ben

Kevésbé naív GPU-s implementáció:

A trükk ugyan az, mint CPU-n: blokkosítás

A blokkokat a lokális memóriában tároljuk el!

Mátrix szorzás OpenCL-ben

```
__kernel void matmul1(__global double* A,
                      __global double* B,
                      __global double* C,
                               int     size,
                               int     blocksize,
                      __local  double* Ablock,
                      __local  double* Bblock)
{ ... }
```

Mátrix szorzás OpenCL-ben


```
__kernel void matmul1(__global double* A,
                      __global double* B,
                      __global double* C,
                               int     size,
                               int     blocksize,
                      __local  double* Ablock,
                      __local  double* Bblock)
{ ... }
```

A lokális memóriát itt expliciten „átadjuk” a kernelnek, ebbe fogjuk az alblokkokat tárolni.

Mátrix szorzás OpenCL-ben

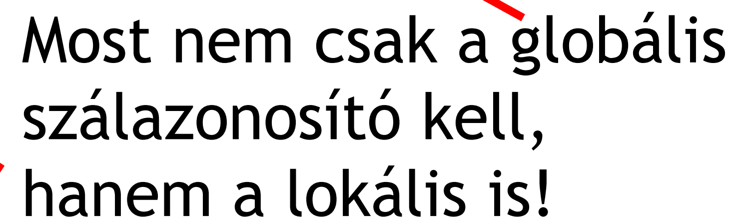
A szálak látszólag továbbra is egyetlen elemet számolnak ki, csak az változik, hogyan teszik ezt!

```
int gx = get_global_id(0); int gy = get_global_id(1);
int lx = get_local_id(0);  int ly = get_local_id(1);

int steps = size / blocksize;
double acc = 0.0;
for(int s=0; s<steps; s=s+1)
{

}
C[gy * size + gx] = acc;
```

Mátrix szorzás OpenCL-ben

A szálak látszólag továbbra is egyetlen elemet számolnak ki, csak az változik, hogyan teszik ezt!

```
int gx = get_global_id(0); int gy = get_global_id(1);
int lx = get_local_id(0);  int ly = get_local_id(1);

int steps = size / blocksize;
double acc = 0.0;
for(int s=0; s<steps; s=s+1)
{

}
C[gy * size + gx] = acc;
```



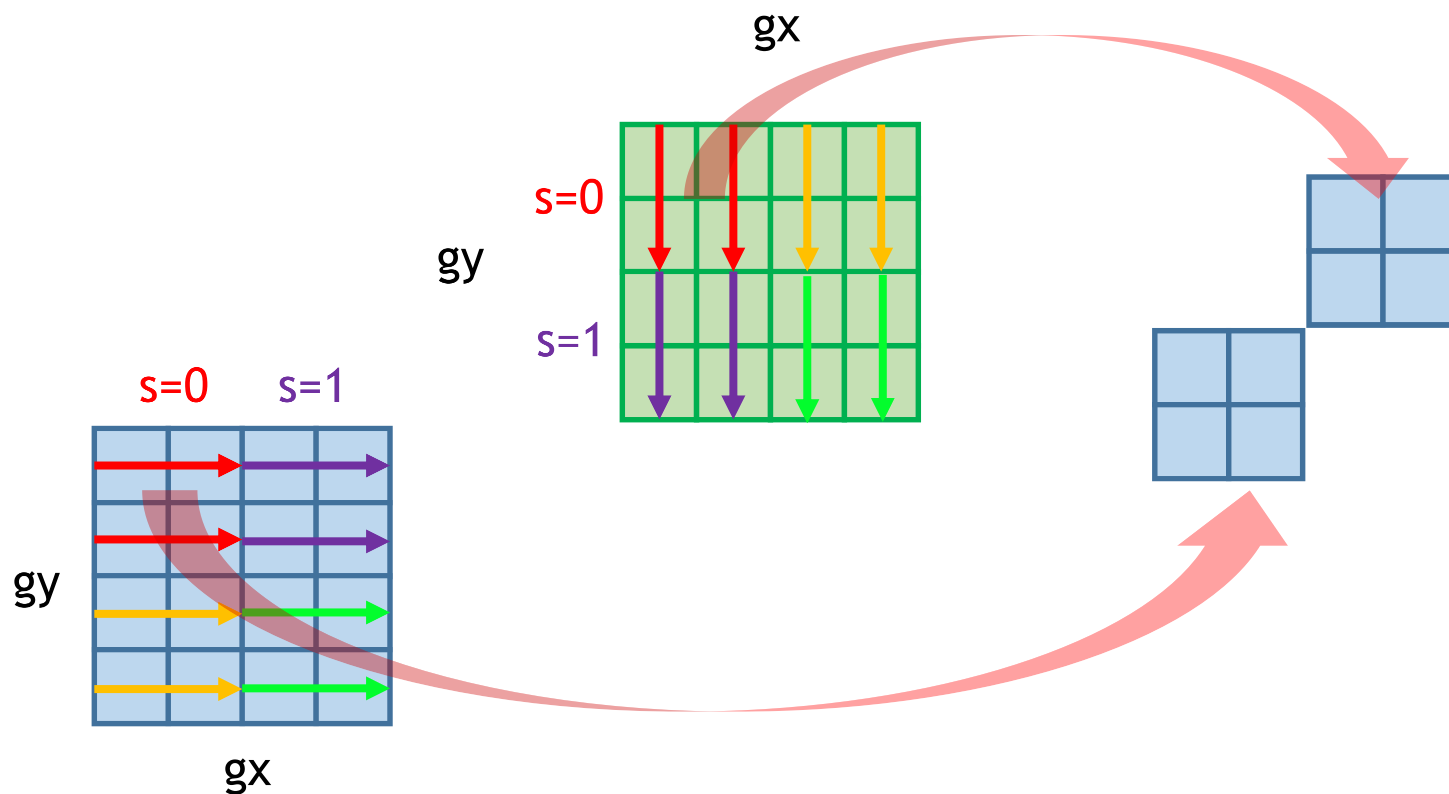
Mátrix szorzás OpenCL-ben

A szálak látszólag továbbra is egyetlen elemet számolnak ki, csak az változik, hogyan teszik ezt!

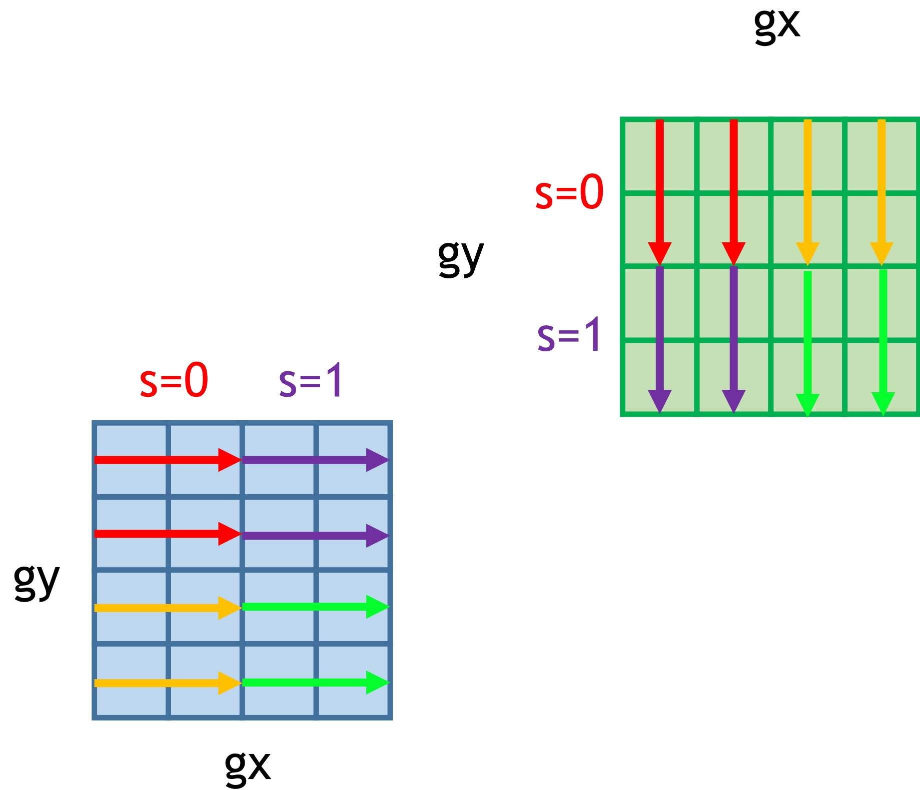
```
int gx = get_global_id(0); int gy = get_global_id(1);
int lx = get_local_id(0);  int ly = get_local_id(1);

int steps = size / blocksize;
double acc = 0.0;
for(int s=0; s<steps; s=s+1)
{

}
C[gy * size + gx] = acc;
```

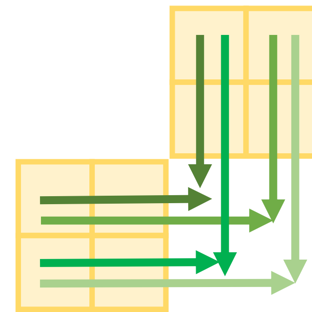
Mátrix szorzás OpenCL-ben

A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába, ezt akarjuk kifejezni:



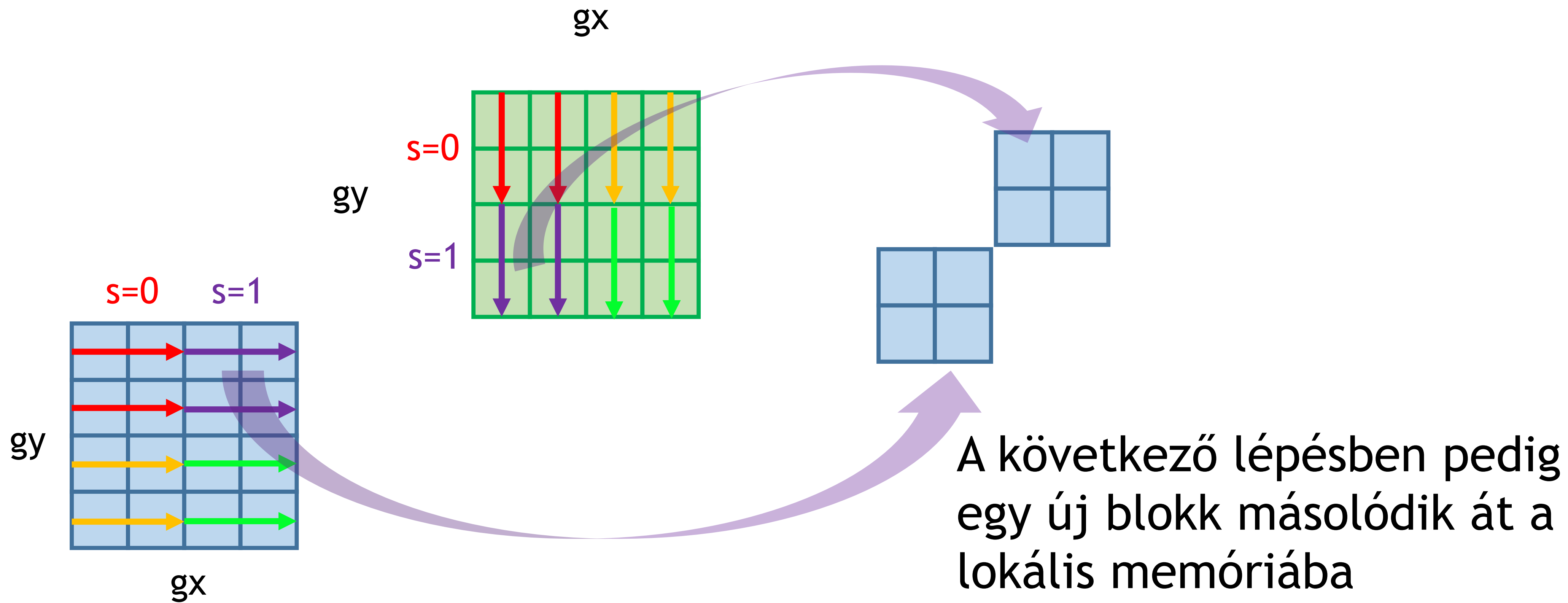
Mátrix szorzás OpenCL-ben





A másolás után minden szál kiszámolja az őt érintő skaláris szorzatot a lokális memóriából olvasva

Mátrix szorzás OpenCL-ben



A következő lépésben pedig egy új blokk másolódik át a lokális memóriába

Mátrix szorzás OpenCL-ben

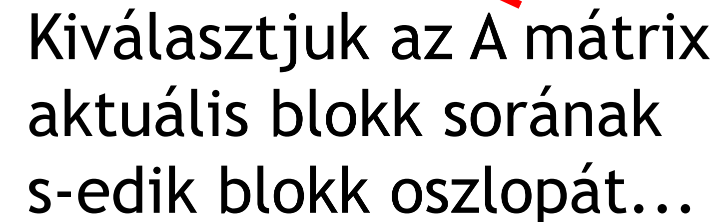
A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába:

```
for(int s=0; s<steps; s=s+1)
{
   int Aoffset      = gy * size + s * blocksize + lx;
   int Boffset      = (s * blocksize + ly) * size + gx;
   int Ablockoffset = ly * blocksize + lx;
   int Bblockoffset = lx * blocksize + ly;
   Ablock[Ablockoffset] = A[Aoffset];
   Bblock[Bblockoffset] = B[Boffset];
   barrier(CLK_LOCAL_MEM_FENCE);
   for(int i = 0; i < blocksize; ++i){...}
}
```

Mátrix szorzás OpenCL-ben

A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába:

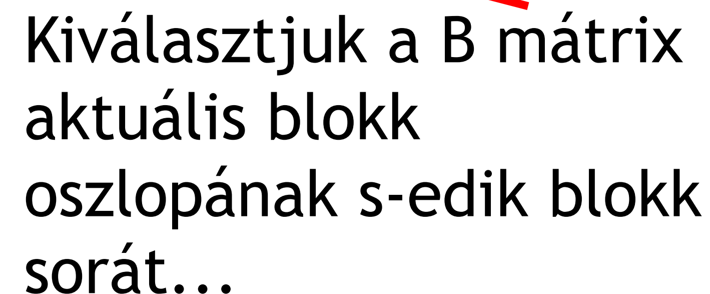
```
for(int s=0; s<steps; s=s+1)
{
    int Aoffset      = gy * size + s * blocksize + lx;
    int Boffset      = (s * blocksize + ly) * size + gx;
    int Ablockoffset = ly * blocksize + lx;
    int Bblockoffset = lx * blocksize + ly;
    Ablock[Ablockoffset] = A[Aoffset];
    Bblock[Bblockoffset] = B[Boffset];
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = 0; i < blocksize; ++i){...}
}
```



Mátrix szorzás OpenCL-ben

A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába:

```
for(int s=0; s<steps; s=s+1)
{
   int Aoffset      = gy * size + s * blocksize + lx;
   int Boffset      = (s * blocksize + ly) * size + gx;
   int Ablockoffset = ly * blocksize + lx;
   int Bblockoffset = lx * blocksize + ly;
   Ablock[Ablockoffset] = A[Aoffset];
   Bblock[Bblockoffset] = B[Boffset];
   barrier(CLK_LOCAL_MEM_FENCE);
   for(int i = 0; i < blocksize; ++i){...}
}
```



Mátrix szorzás OpenCL-ben

A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába:

```
for(int s=0; s<steps; s=s+1)
{
   int Aoffset      = gy * size + s * blocksize + lx;
   int Boffset      = (s * blocksize + ly) * size + gx;
   int Ablockoffset = ly * blocksize + lx;
   int Bblockoffset = lx * blocksize + ly;
   Ablock[Ablockoffset] = A[Aoffset];
   Bblock[Bblockoffset] = B[Boffset];
   barrier(CLK_LOCAL_MEM_FENCE);
   for(int i = 0; i < blocksize; ++i){...}
}
```

... és a kiválasztott globális blokkokat átmásoljuk a lokális memóriába a lokális szálazonosítók által adott helyre.

Mátrix szorzás OpenCL-ben

A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába:

```
for(int s=0; s<steps; s=s+1)
{
   int Aoffset      = gy * size + s * blocksize + lx;
   int Boffset      = (s * blocksize + ly) * size + gx;
   int Ablockoffset = ly * blocksize + lx;
   int Bblockoffset = lx * blocksize + ly;
   Ablock[Ablockoffset] = A[Aoffset];
   Bblock[Bblockoffset] = B[Boffset];
   barrier(CLK_LOCAL_MEM_FENCE);
   for(int i = 0; i < blocksize; ++i){...}
}
```

Itt még annyi trükköt csinálunk, hogy a B blokkot meg is transzponáljuk, hogy a skaláris szorzás hatékonyabb legyen

Mátrix szorzás OpenCL-ben

A belső ciklusban először átmásolják a szálak az almátrixot a lokális memóriába:

```
for(int s=0; s<steps; s=s+1)
{
   int Aoffset      = gy * size + s * blocksize + lx;
   int Boffset      = (s * blocksize + ly) * size + gx;
   int Ablockoffset = ly * blocksize + lx;
   int Bblockoffset = lx * blocksize + ly;
   Ablock[Ablockoffset] = A[Aoffset];
   Bblock[Bblockoffset] = B[Boffset];
   barrier(CLK_LOCAL_MEM_FENCE);
   for(int i = 0; i < blocksize; ++i){...}
}
```

Szinkronizálni kell, hogy biztosan minden szál másolása befejeződjön, mielőtt használni akarják egymás adatait!!!

Mátrix szorzás OpenCL-ben

A blokk cikluson belüli ciklusban a skaláris szorzás törénik:

```
for(int s=0; s<steps; s=s+1)
{
    ...
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = 0; i < blocksize; ++i)
    {
        double fA = Ablock[ly*blocksize+i];
        double fB = Bblock[lx*blocksize+i];
        acc += fA * fB;
    }
}
```

Végül a szálak a blokkon belül egy blokkméretnyi skaláris szorzatot számolnak ki!

Mátrix szorzás OpenCL-ben

A blokk cikluson belüli ciklusban a skaláris szorzás törénik:

```
for(int s=0; s<steps; s=s+1)
{
    ...
    barrier(CLK_LOCAL_MEM_FENCE);
    for(int i = 0; i < blocksize; ++i)
    {
        double fA = Ablock[ly*blocksize+i];
        double fB = Bblock[lx*blocksize+i];
        acc += fA * fB;
    }
}
```

Mivel B-t megtranszponáltuk, mind két lokális memóriát sorfolytonosan tudjuk olvasni!

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

Lokális memória beállítása a host oldalon:

Ez egyszerű:

- Nem kell buffert létrehozni
- A kernel argumentumnál csak méretet adunk meg és pointert nem:

```
static const int blocksize = 8;
clSetKernelArg(kernel2, 5, blocksize*blocksize*sizeof(double), nullptr);
clSetKernelArg(kernel2, 6, blocksize*blocksize*sizeof(double), nullptr);
```

Eredmények

Mátrix szorzás OpenCL-ben

Wigner GPU Lab

Eredmények

Blokk méret	CPU Idő [s]	GPU1 Idő [s]	GPU2 Idő [s]
1	66	19.5	10.9
2	16.1	3.55	2.15
4	5.9	0.76	0.47
8	2.4	**0.21**	**0.11**
16	2.9		
32	2.7		
64	**2.3**		
128	2.6		
256	2.8		
512	15.2		
1024 (ez a naiv)	36.4	0.49	0.64

Mátrix szorzás OpenCL-ben

Ez még nem a legoptimálisabb kód

Tovább lehet játszani a blokk méretekkel,

több adatot beolvasni a lokális memóriába,

vektorosan olvasni a lokális memóriát,

a lokális memóriából privátba átvenni több adatot

nem egyetlen elemet számolni, hanem többet belőlük,

B-t nem transzponálni?

stb.

Mátrix szorzás SYCL-ben

Wigner GPU Lab

Feladat:

- Írjuk meg most a mátrix szorzást SYCL-ben!

Mátrix szorzás SYCL-ben

Egy függvényt írunk erre, ami a mátrix méretet, a blokk méretet, a két szorzandó-, valamint az eredmény tömböt várja:

```
template <typename T>
void matmul_kernel(int size, int blocksize,
                   std::vector<T> const& mA,
                   std::vector<T> const& mB,
                   std::vector<T>& mC)
{ ... }
```

Mátrix szorzás SYCL-ben

A SYCL kernel nevét pedig vezessük be ugyanitt:

```
template <typename T> class MatulKernel;

template <typename T>
void matmul_kernel(int size, int blocksize,
                   std::vector<T> const& mA,
                   std::vector<T> const& mB,
                   std::vector<T>& mC)
{ ... }
```

Mátrix szorzás SYCL-ben


```
cl::sycl::queue deviceQueue{cl::sycl::gpu_selector()};

cl::sycl::buffer<T, 1> ba(mA.data(), size*size);
cl::sycl::buffer<T, 1> bb(mB.data(), size*size);
cl::sycl::buffer<T, 1> bc(mC.data(), size*size);
```

Mátrix szorzás SYCL-ben

Létrehozunk egy alap queue-t az első GPU eszközre

```
cl::sycl::queue deviceQueue{cl::sycl::gpu_selector()};

cl::sycl::buffer<T, 1> ba(mA.data(), size*size);
cl::sycl::buffer<T, 1> bb(mB.data(), size*size);
cl::sycl::buffer<T, 1> bc(mC.data(), size*size);
```

Mátrix szorzás SYCL-ben


```
cl::sycl::queue deviceQueue{cl::sycl::gpu_selector()};

cl::sycl::buffer<T, 1> ba(mA.data(), size*size);
cl::sycl::buffer<T, 1> bb(mB.data(), size*size);
cl::sycl::buffer<T, 1> bc(mC.data(), size*size);
```

Létrehozunk buffereket a három tömbre

Mátrix szorzás SYCL-ben

Wigner GPU Lab

Nyitunk egy scope-ot, ebben fog lefutni a GPU-s rész,
a queue-ba pedig egy lambdát küldünk...

```
{
   deviceQueue.submit([&](cl::sycl::handler& cgh)
                                   { ... });
}
```

Mátrix szorzás SYCL-ben

GPU Lab

A queue lambda belsejében hozzáféréseket kell kérnünk minden memóriához, amit használni szeretnénk (értsünk ide most egy `using namespace cl::sycl;` -t):

```
auto A = ba.template get_access<access::mode::read>(cgh);
auto B = bb.template get_access<access::mode::read>(cgh);
auto C = bc.template get_access<access::mode::write>(cgh);

auto local_range = range<1>(blocksize * blocksize);
accessor<T, 1, access::mode::read_write,
               access::target::local> Ablock(local_range, cgh);
accessor<T, 1, access::mode::read_write,
               access::target::local> Bblock(local_range, cgh);
```

Mátrix szorzás SYCL-ben

Wigner GPU Lab

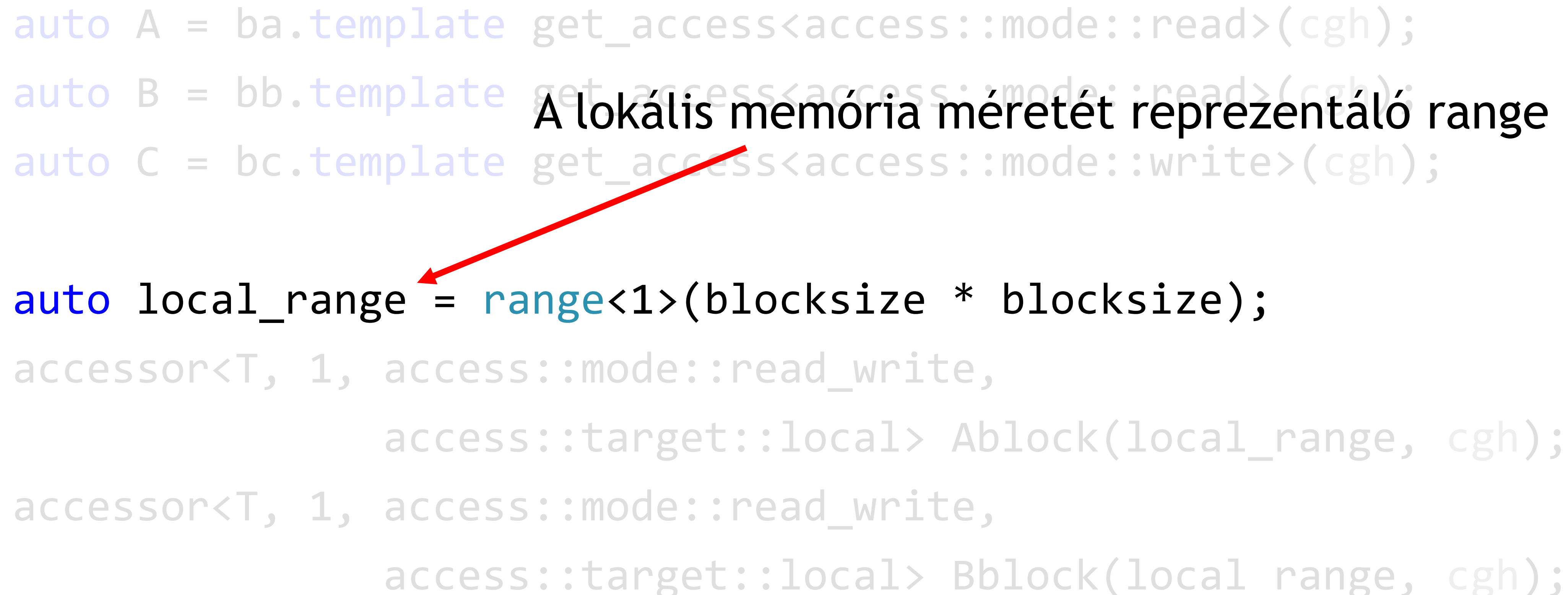
A queue lambda belsejében hozzáféréseket kell kérnünk minden memóriához, amit használni szeretnénk (értsünk ide most egy `using namespace cl::sycl;` -t):

```
auto A = ba.template get_access<access::mode::read>(cgh);
auto B = bb.template get_access<access::mode::read>(cgh);
auto C = bc.template get_access<access::mode::write>(cgh);
```

```
auto local_range = range<1>(blocksize * blocksize);
accessor<T, 1, access::mode::read_write,
               access::target::local> Ablock(local_range, cgh);
accessor<T, 1, access::mode::read_write,
               access::target::local> Bblock(local_range, cgh);
```

Mátrix szorzás SYCL-ben

A queue lambda belsejében hozzáféréseket kell kérnünk minden memóriához, amit használni szeretnénk (értsünk ide most egy `using namespace cl::sycl;` -t):



Mátrix szorzás SYCL-ben

A queue lambda belsejében hozzáféréseket kell kérnünk minden memóriához, amit használni szeretnénk (értsünk ide most egy `using namespace cl::sycl;` -t):

```
auto A = ba.template get_access<access::mode::read>(cgh);
auto B = bb.template get_access<access::mode::read>(cgh);
auto C = bc.template get_access<access::mode::write>(cgh);
```

```
auto local_range = range<1>(blocksize * blocksize);
accessor<T, 1, access::mode::read_write,
               access::target::local> Ablock(local_range, cgh);
accessor<T, 1, access::mode::read_write,
               access::target::local> Bblock(local_range, cgh);
```

Mátrix szorzás SYCL-ben

Wigner GPU Lab

A számolást a mátrix méretén a blokkméret csoportokban indítjuk el:

```
auto global_range = nd_range<2>{range<2>(size, size),
                                range<2>(blocksize, blocksize)};

cgh.parallel_for<class MatulKernel<T>>(global_range,
                                       [=](nd_item<2> i){ ... } );
```

Mátrix szorzás SYCL-ben

Wigner GPU Lab

Maga a kernel ezekután majdnem tökéletes másolata az OpenCL-es változatnak:

```
[=](cl::sycl::nd_item<2> i){
  int lx = i.get_local(0);  int ly = i.get_local(1);
  int gx = i.get_global(0); int gy = i.get_global(1);
  int steps = size / blocksize; T acc = 0.0;
  for(int s=0; s<steps; s=s+1){
    int Ablockoffset = ly * blocksize + lx; int Aoffset = gy * size + s * blocksize + lx;
    int Bblockoffset = lx * blocksize + ly; int Boffset = (s * blocksize + ly) * size + gx;
    Ablock[Ablockoffset] = A[Aoffset]; Bblock[Bblockoffset] = B[Boffset];
    i.barrier(cl::sycl::access::fence_space::local_space);
    for(int i=0; i<blocksize; ++i)
    { T fA = Ablock[ly*blocksize+i]; T fB = Bblock[lx*blocksize+i]; acc += fA * fB; }
    i.barrier(cl::sycl::access::fence_space::local_space);
  }
  C[gy * size + gx] = acc;
}
```

Mátrix szorzás SYCL-ben

Pl.: a szálazonosítót a lambda argumentumából,
a `cl::sycl::nd_item`-ből, vagy `cl::sycl::id`-ből tudjuk lekérni:

SYCL-ben:

```
int gx = i.get_global(0);
int lx = i.get_local(0);
```

OpenCL-ben:

```
int gx = get_global_id(0);
int lx = get_local_id(0);
```

Mátrix szorzás SYCL-ben

Wigner GPU Lab

Pl.: hasonlóan a lokális memóriát érintő csoportszinkronizációt is a `cl::sycl::nd_item`-ből, vagy `cl::sycl::id`-ből kell hívni:

SYCL-ben:

```
i.barrier(cl::sycl::access::fence_space::local_space);
```

OpenCL-ben:

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Mátrix szorzás

A példakódok itt érhetőek el:

- OpenCL
- SYCL

Gravitációs N test

OpenCL módra

Gravitációs N test

- Korábban láttuk, hogy milyen megfontolásokkal javíthatunk egy naiv C++ kódon, hogy az ~2X gyorsabb legyen.
- A részekre ható eredő erő teljesen függetlenül számolható
  - Az egyetlen „függőségünk” az optimális adat hozzáférés lesz
- Egyes optimalizációk működnek GPU-n is, mások nem annyira

Típus definíciók

- Eddig már megszoktuk, hogy a gazda és az eszköz oldali kód nem csak elválik egymástól, hanem konkrétan más nyelven íródnak
- Az első újdonság most a típus definíciók különbözősége
- Eltérő módokon definiálhatunk egy objektumot a gazda és eszköz nyelvén
  - Vagy egy forrás fájlba rakjuk őket és makró mágiával választjuk szét a két esetet
  - Vagy külön forrás fájlba írjuk őket
  - Mi az utóbbit szorgalmazzuk

Memória elrendezés

particle.hpp

```
struct particle
{
  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct
{
  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

Memória elrendezés

particle.hpp

```
struct particle
{
  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct
{
  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

A két nyelv eltérő típus neveket használ ugyanazokra a típusokra

Memória elrendezés

particle.hpp

```
struct particle
{
  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct
{
  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

Gazda oldalon, ha egy API saját típusokat szállít, akár olyan dolgokra is amiket egyébként ismerünk, akkor is érdemes ezeket használni.
Általában nagyon jó okuk van új típusokat definiálni

Memória elrendezés

particle.hpp

```
struct                  particle
{
  particle(const input_particle& p) :
    mass(in.mass),
    pos(in.pos),
    v(in.v),
    f{ 0, 0, 0 } {}

  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct
{
  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

Kényelmi konstruktor fájl beolvasáshoz

Memória elrendezés

particle.hpp

Kedvenc C++ fordítónk

```
struct                  particle
{
  particle(const input_particle& p) :
    mass(in.mass),
    pos(in.pos),
    v(in.v),
    f{ 0, 0, 0 } {}

  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct

{




  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

Memória elrendezés

GPU Lab

particle.hpp

```
struct              particle
{
  particle(const input_particle& p) :
    mass(in.mass),
    pos(in.pos),
    v(in.v),
    f{ 0, 0, 0 } {}

  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl OpenCL C fordító

```
typedef struct

{








  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

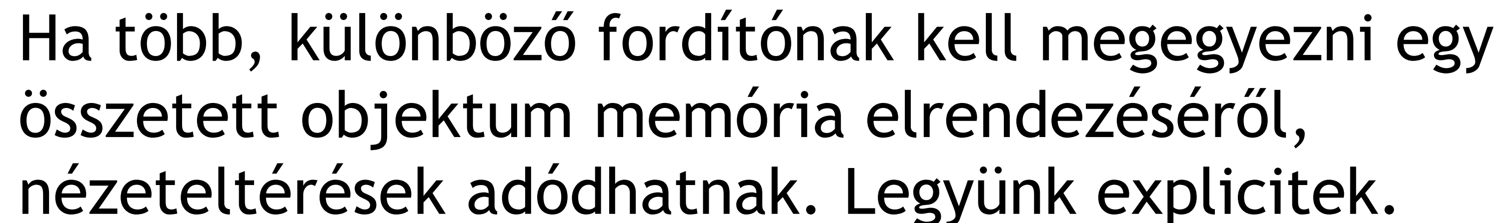
Memória elrendezés

particle.hpp

```
struct alignas(16) particle
{
  particle(const input_particle& p) :
    mass(in.mass),
    pos(in.pos),
    v(in.v),
    f{ 0, 0, 0 } {}

  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct __attribute__
((aligned(16)))
{
  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```



Memória elrendezés

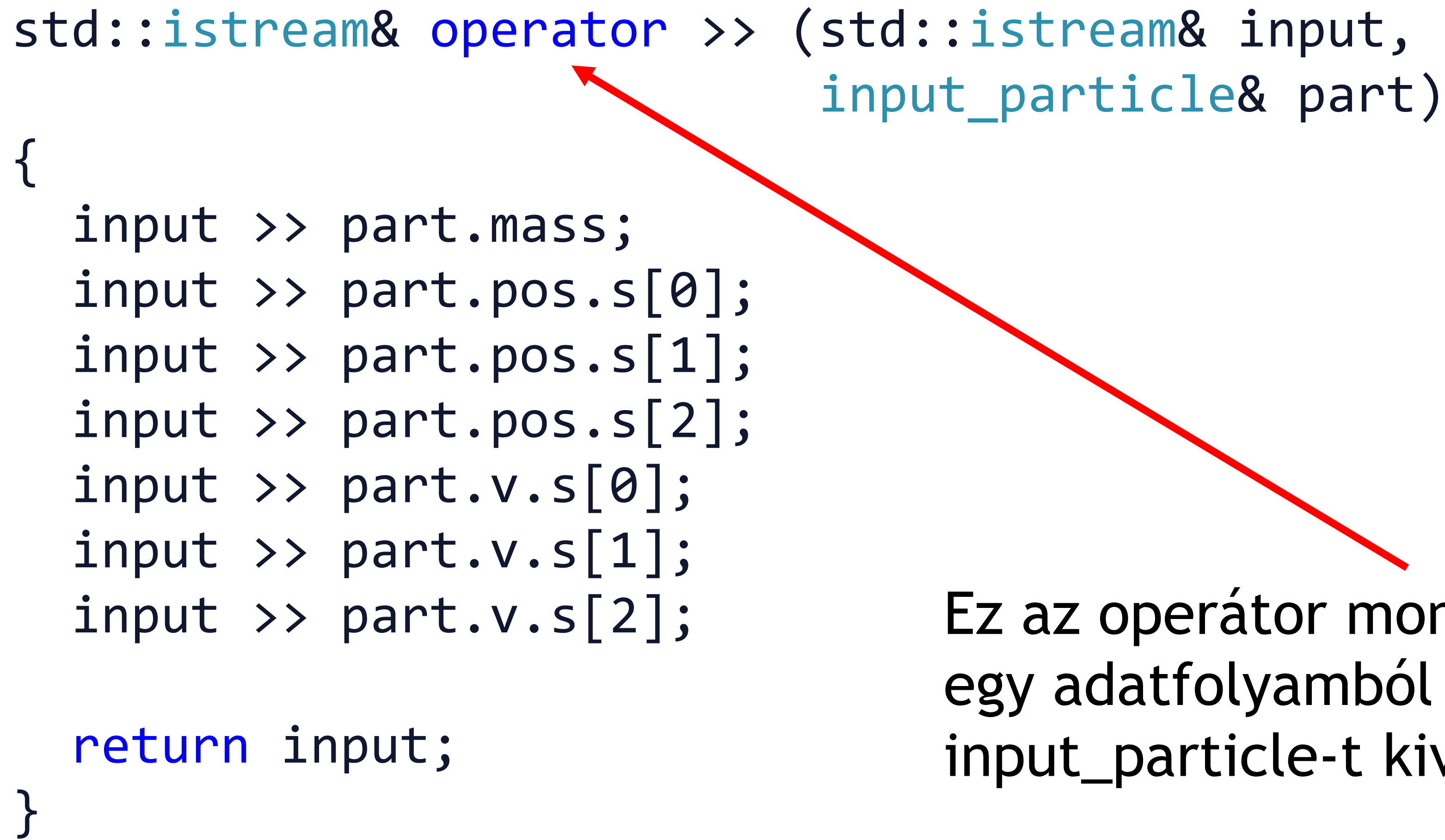
particle.hpp

```
struct alignas(16) particle
{
  particle(const input_particle& p) :
    mass(in.mass),
    pos(in.pos),
    v(in.v),
    f{ 0, 0, 0 } {}

  cl_float3 pos;
  cl_float3 v;
  cl_float3 f;
  cl_float mass;
};
```

particle.cl

```
typedef struct __attribute__
((aligned(16)))
{



  float3 pos;
  float3 v;
  float3 f;
  float mass;
} particle;
```

Fájl beolvasás



```
std::istream& operator >> (std::istream& input,
                           input_particle& part)
{
  input >> part.mass;
  input >> part.pos.s[0];
  input >> part.pos.s[1];
  input >> part.pos.s[2];
  input >> part.v.s[0];
  input >> part.v.s[1];
  input >> part.v.s[2];

  return input;
}
```

Ez az operátor mondja meg, hogyan kell egy adatfolyamból egy darab input_particle-t kivenni.

Fájl beolvasás

Wigner GPU Lab

```
std::vector<particle> read_particle_file(const std::string& filename)
{
  std::vector<particle> result;
  std::ifstream input(filename, std::ios::in);

  if (!input.is_open())
    std::cerr << "Couldn’t open file " << filename << std::endl;

  std::copy(std::istream_iterator<input_particle>{ input },
            std::istream_iterator<input_particle>{},
            std::back_inserter(result));

  return result;
}
```

Fájl beolvasás

```
std::vector<particle> read_particle_file(const std::string& filename)
{
  std::vector<particle> result;
  std::ifstream input(filename, std::ios::in);

  if (!input.is_open())
    std::cerr << "Couldn't open file " << filename << std::endl;

  std::copy(std::istream_iterator<input_particle>{ input },
            std::istream_iterator<input_particle>{},
            std::back_inserter(result));

  return result;
}
```

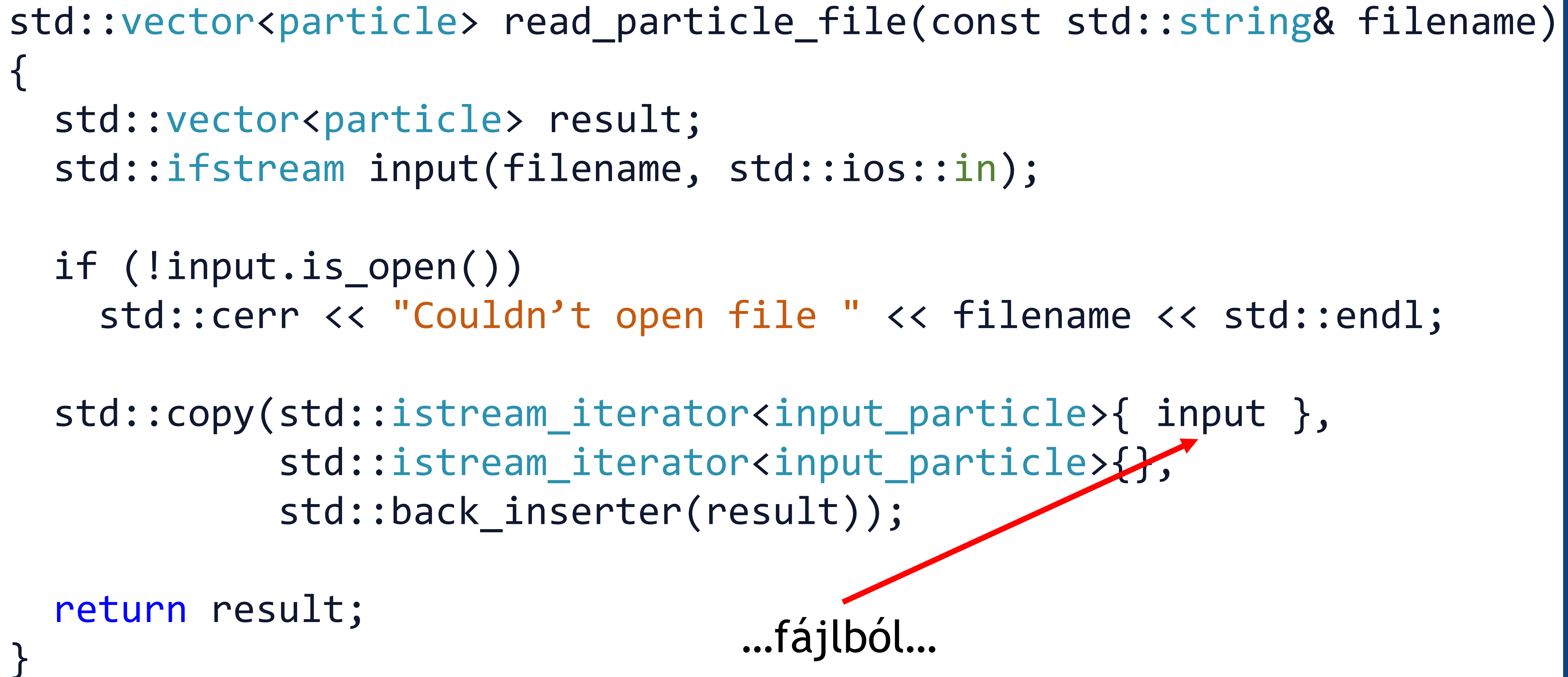
Létrehozunk egy bemeneti folyamot.

Fájl beolvasás

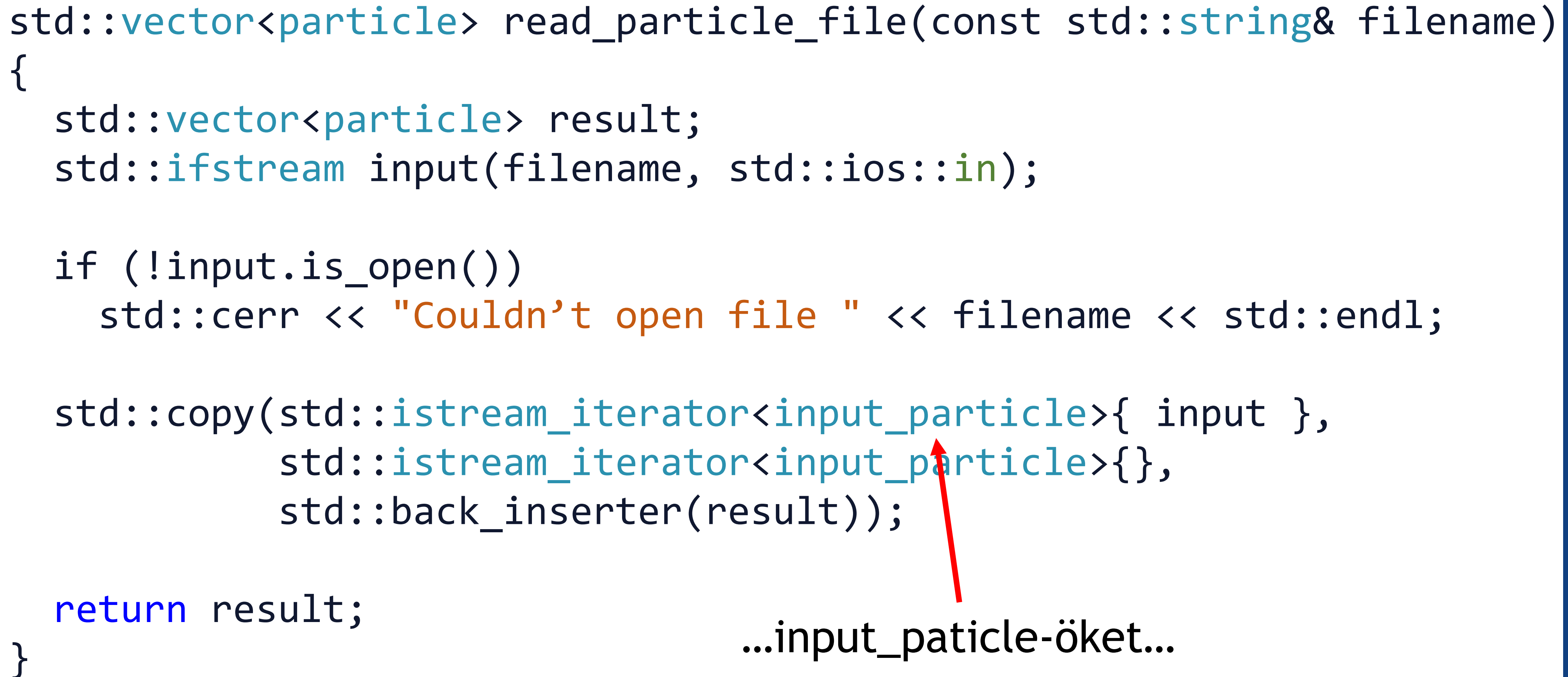
```
std::vector<particle> read_particle_file(const std::string& filename)
{
  std::vector<particle> result;
  std::ifstream input(filename, std::ios::in);

  if (!input.is_open())
    std::cerr << "Couldn’t open file " << filename << std::endl;

  std::copy(std::istream_iterator<input_particle>{ input },
            std::istream_iterator<input_particle>{},
            std::back_inserter(result));

  return result;
}
```

Másolni szeretnénk…

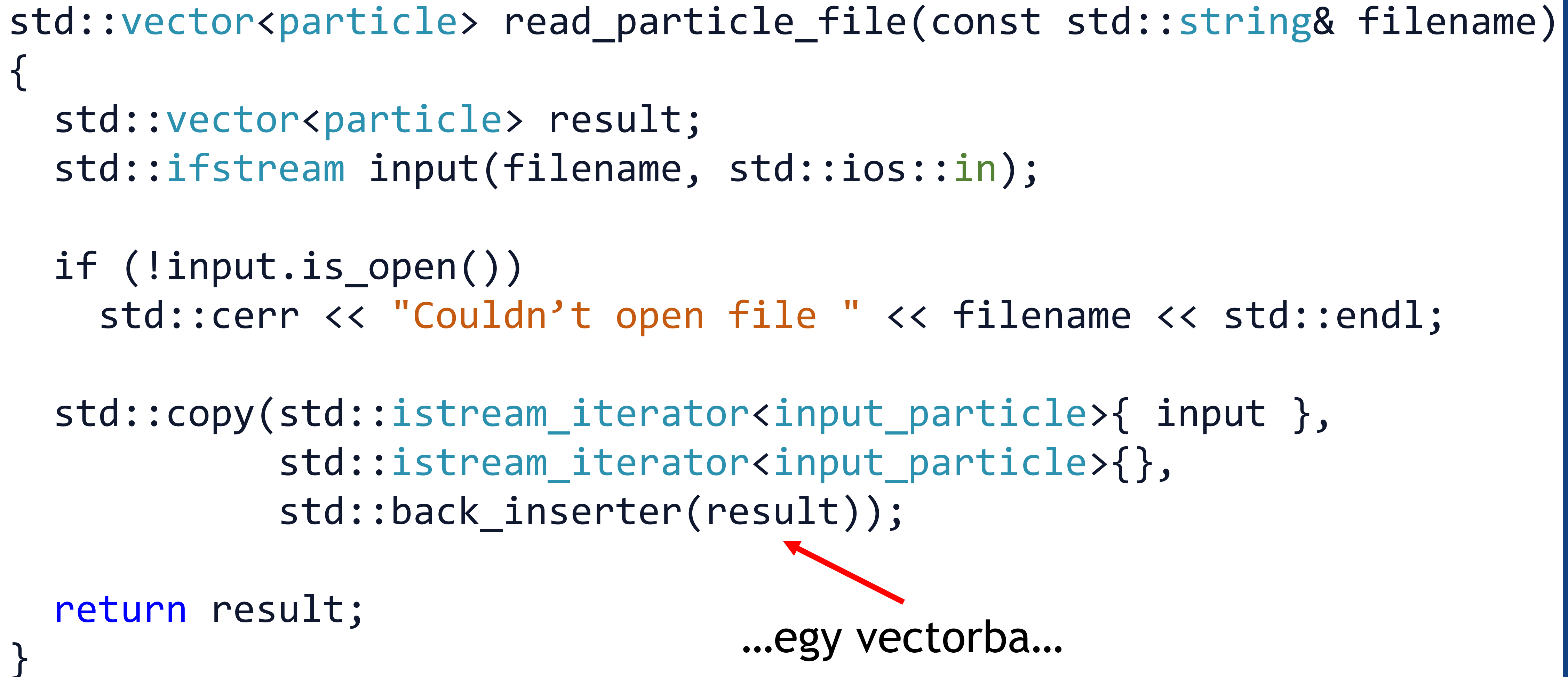
Fájl beolvasás



Fájl beolvasás



Fájl beolvasás



Fájl beolvasás

Wigner GPU Lab

```
std::vector<particle> read_particle_file(const std::string& filename)
{
  std::vector<particle> result;
  std::ifstream input(filename, std::ios::in);

  if (!input.is_open())
    std::cerr << "Couldn’t open file " << filename << std::endl;

  std::copy(std::istream_iterator<input_particle>{ input },
            std::istream_iterator<input_particle>{},
            std::back_inserter(result));

  return result;
}
```

Az istream_iterator minden léptetéskor (operator++) meghívja az általunk írt operator<<-t. Egy particle pedig tudja hogyan kell létrehozni magát input_particle-ből.

Et cetera

- Ezektől eltekintve semmi újdonság nincs gazda oldalon
    - Parancssori kapcsolók olvasása (ha akarunk)
    - Platform, eszközök lekérdezése, választás közülük
    - Kontextus, parancslisták, bufferek létrehozása
    - Program betöltése, kernelek fordítása

Eszköz oldali kód

```
__kernel void interaction( __global particle* particles )
{
  int gid = get_global_id(0), gsi = get_global_size(0);

  particle my_particle = particles[gid];
  float3 force = (float3)(0.0, 0.0, 0.0);

  for (int i = 0 ; i < gsi ; ++i) {
    particle temp = particles[i];

    if (gid != i)
      force += burning_calculate_force(&my_particle, &temp);
  }
  particles[gid].f = force;
}
```

Eszköz oldali kód

```
__kernel void interaction( __global particle* particles )
{
  int gid = get_global_id(0), gsi = get_global_size(0);

  particle my_particle = particles[gid];
  float3 force = (float3)(0.0, 0.0, 0.0);

  for (int i = 0 ; i < gsi ; ++i) {
    particle temp = particles[i];

    if (gid != i)
      force += burning_calculate_force(&my_particle, &temp);
  }
  particles[gid].f = force;
}
```

A naiv C++-os esetben két ciklusunk volt. A külső választott ki egy részecskét írásra, a belső olvasott és felösszegezte az eredő erőt. 1 dimenziós kernel indexeléssel egy ciklust hámozunk le.

Wigner GPU Lab

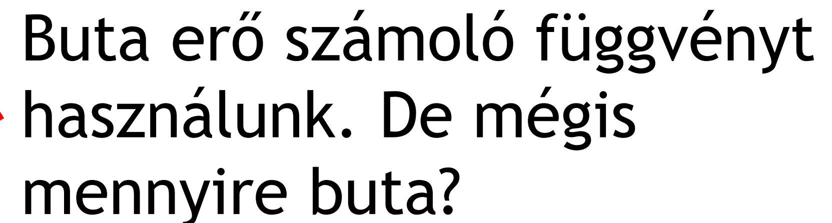
Eszköz oldali kód

```
__kernel void interaction( __global particle* particles )
{
  int gid = get_global_id(0), gsi = get_global_size(0);

  particle my_particle = particles[gid];
  float3 force = (float3)(0.0, 0.0, 0.0);

  for (int i = 0 ; i < gsi ; ++i) {
    particle temp = particles[i];

    if (gid != i)
      force += burning_calculate_force(&my_particle, &temp);
  }
  particles[gid].f = force;
}
```

Futásidejű index határokat használunk.

Eszköz oldali kód

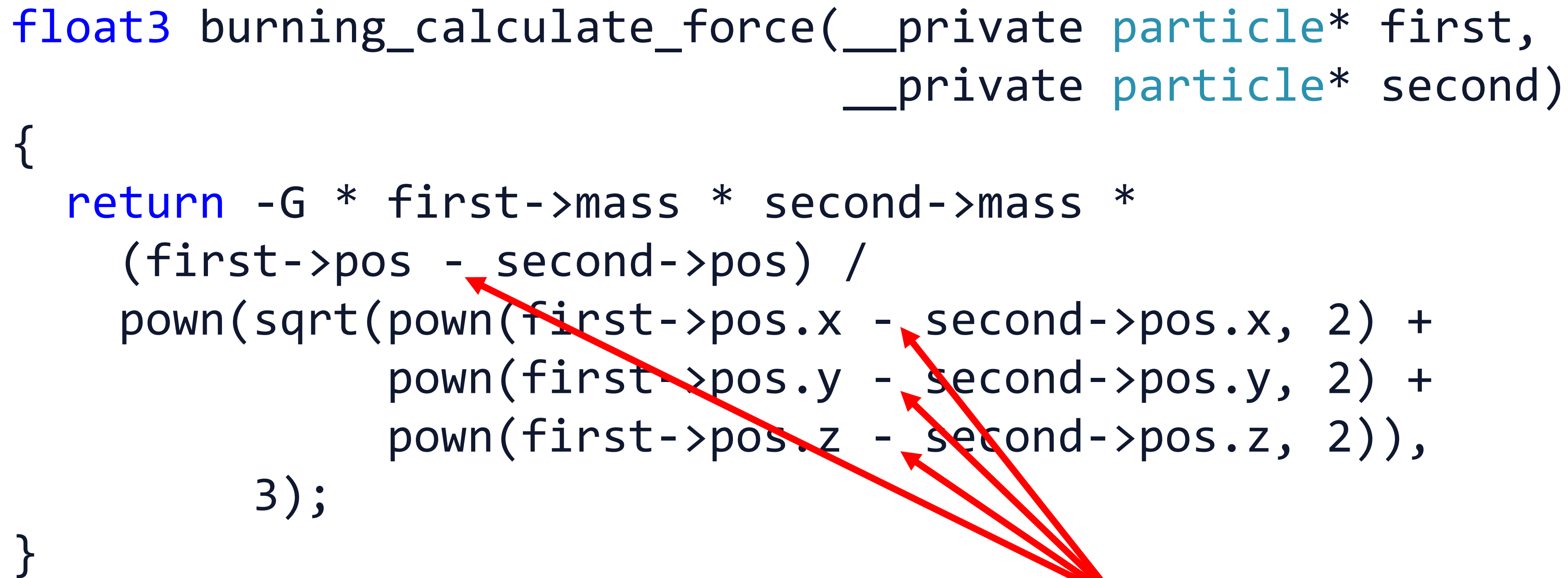
```
__kernel void interaction( __global particle* particles )
{
  int gid = get_global_id(0), gsi = get_global_size(0);

  particle my_particle = particles[gid];
  float3 force = (float3)(0.0, 0.0, 0.0);

  for (int i = 0 ; i < gsi ; ++i) {
    particle temp = particles[i];

    if (gid != i)
      force += burning_calculate_force(&my_particle, &temp);
  }
  particles[gid].f = force;
}
```



Eszköz oldali kód


```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(sqrt(pown(first->pos.x - second->pos.x, 2) +
              pown(first->pos.y - second->pos.y, 2) +
              pown(first->pos.z - second->pos.z, 2)),
         3);
}
```

Eszköz oldali kód



```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(sqrt(pown(first->pos.x - second->pos.x, 2) +
              pown(first->pos.y - second->pos.y, 2) +
              pown(first->pos.z - second->pos.z, 2)),
         3);
}
```

A koordináta különbségeket valószínűleg kétszer számoljuk.

Eszköz oldali kód

WIGNER GPU Lab

```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(sqrt(pown(first->pos.x - second->pos.x, 2) +
              pown(first->pos.y - second->pos.y, 2) +
              pown(first->pos.z - second->pos.z, 2)),
         3);
}
```

Hatványozni pow-val még mindig nem a legjobb. Itt legalább van integrális változat (pown) amit fordítási idejű konstanssal hívunk. Amelyik fordító ezt nem ismeri fel, azt nem szabad használni.

„New is always better” – Barney Stinson

Intrinsics FTW

Wigner GPU Lab

```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(sqrt(pown(first->pos.x - second->pos.x, 2) +
              pown(first->pos.y - second->pos.y, 2) +
              pown(first->pos.z - second->pos.z, 2)),
         3);
}
```

Intrinsics FTW


```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(
                                                    ,
         3);
}
```

Intrinsics FTW

Wigner GPU Lab

```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(distance(first->pos, second->pos),



         3);
}
```

Van nekünk egy distance nevű beépített függvényünk. Talán érdemes használni.

Intrinsics FTW

```
float3 burning_calculate_force(__private particle* first,
                               __private particle* second)
{
  return -G * first->mass * second->mass *
    (first->pos - second->pos) /
    pown(distance(first->pos, second->pos), 3);
}
```

Aliasing / elágazás

```
__kernel void interaction( __global particle* particles )
{
  int gid = get_global_id(0), gsi = get_global_size(0);

  particle my_particle = particles[gid];
  float3 force = (float3)(0.0, 0.0, 0.0);

  for (int i = 0 ; i < gsi ; ++i) {
    particle temp = particles[i];

    if (gid != i)
      force += burning_calculate_force(&my_particle, &temp);
  }
  particles[gid].f = force;
}
```

Említettük, hogy ciklus hasában elágazni nem szexi, és ugye GPU-n elágazni pedig fáj. Vegyük ki!

Aliasing / elágazás

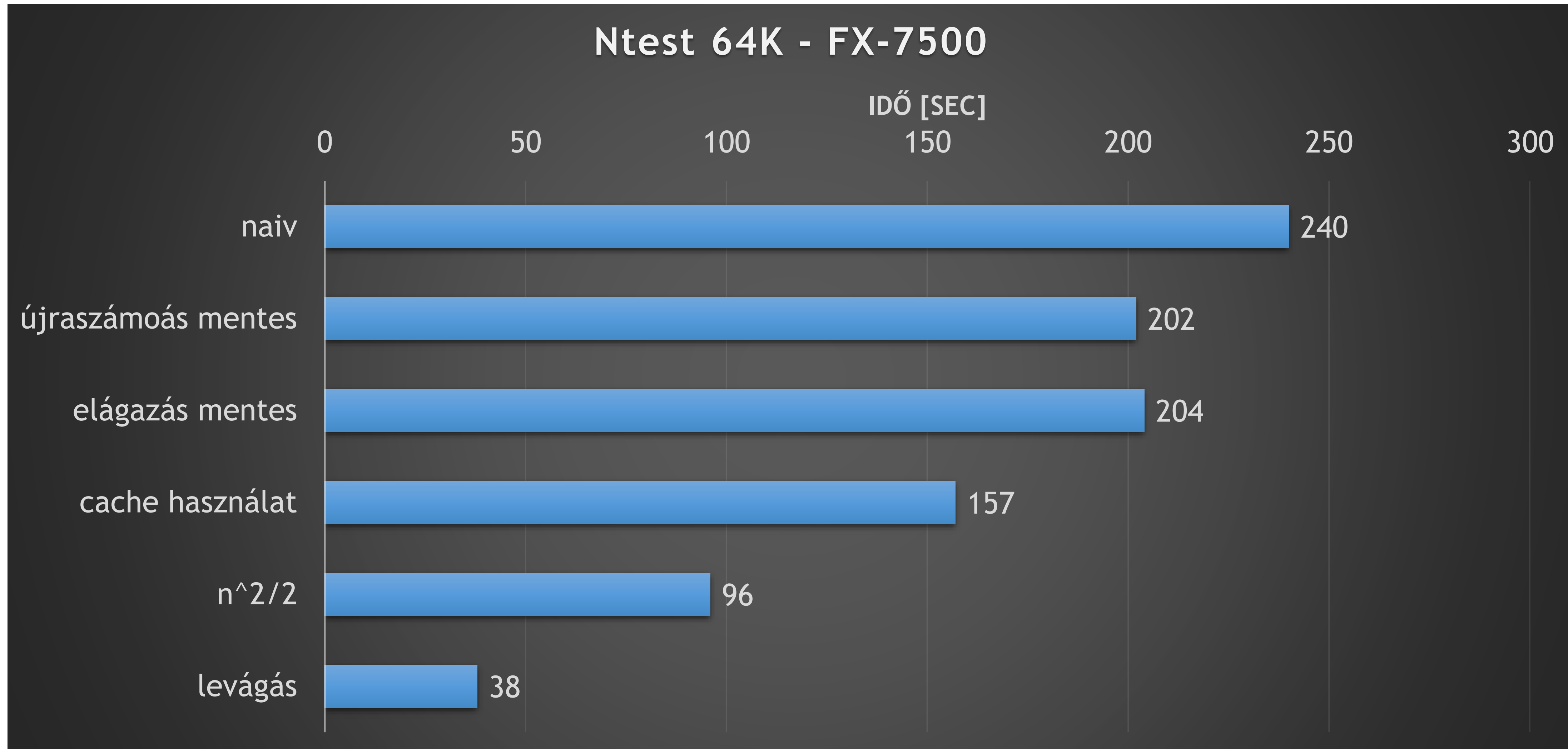
```
for (int i = 0 ; i < gsi ; ++i)
{
  particle temp = particles[i];

  if (gid != i)
    force += burning_calculate_force(&my_particle, &temp);
}
```


Wigner GPU Lab

Aliasing / elágazás

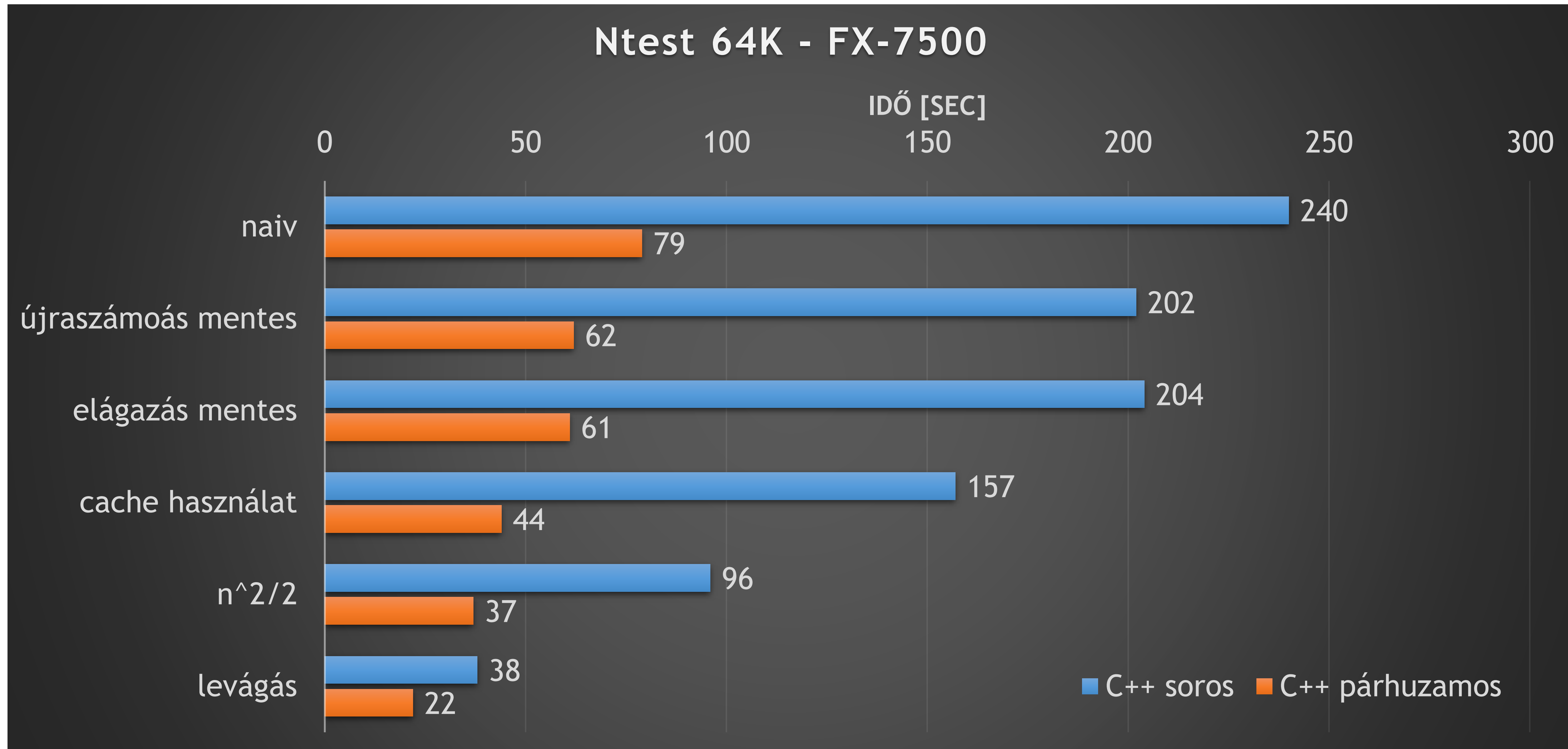
```
for (int i = 0 ; i < gid ; ++i)
{
  particle temp = particles[i];

  force += calculate_force(&my_particle, &temp);
}

for (int i = gid + 1 ; i < gsi ; ++i)
{
  particle temp = particles[i];

  force += calculate_force(&my_particle, &temp);
}
```

Időzítések



Wigner GPU Lab

Időzítések



Időzítések