



**GPU Laboratórium**

## 4. Funkcionális primitívek GPU-n

# GPU API emlékeztető

Jelenleg a következő eszközök állnak rendelkezésre GPU-kódok futtatására:

- ▶ DirectX vagy OpenGL vagy Vulkan Compute Shader
  - ▶ Ez grafikai célokra van kitalálva, nem tudományos számításokra
  - ▶ Nincs teljes körű kontroll a memória elrendezés felett (nem lehet átcsofolni memóriát), nincs garantálva a pontosság minden IEEE lebegőpontos műveletre
- ▶ Microsoft C++ AMP - leálltak a fejlesztéssel, csak egy demonstráció volt
- ▶ OpenMP, OpenACC és társai:
  - ▶ pragma alapú párhuzamosítás a for ciklusok előtt, nem túl elegáns...
- ▶ OpenCL: A C kernelnyelv nem skálázik, összetett dolgokra és generikusságra nem alkalmas, a 2.1-be tervezett C++ kernelnyelv még sehol nincs implementálva...

# GPU API emlékeztető

Jelenleg a következő eszközök állnak rendelkezésre GPU-kódok futtatására:

- ▶ Clang / GCC is tud C/C++ kódokat a megfelelő köztesekre átfordítani
- ▶ AMD HCC stack: csak AMD eszközökön működő, a C++ AMP szabvány továbbfejlesztésére épülő nyelvi kiterjesztés

Mi az alábbiakkal fogunk részletesebben foglalkozni:

- ▶ **NVIDIA CUDA:**
  - ▶ C++ nyelvi kiterjesztés, aktívan fejlesztik, azonban csak NVIDIA GPUkon fut (bár vannak kísérletek ennek független eszközökkel való átfordítására)
- ▶ **KHRONOS SYCL standard és annak a ComputeCpp implementációja**
  - ▶ Még eléggé béta állapotú, de szintén aktívan fejlesztett könyvtár (NEM nyelvi kiterjesztés!!!)
  - ▶ Előnye, hogy elvileg minden OpenCL-t támogató szabványos implementáció fölött képes futni (a gyakorlatban ez jelenleg Intel és AMD eszközöket jelent)

# CUDA vs SYCL

# CUDA vs SYCL

## Hasonlóságok:

- ▶ Mindkét rendszer lehetővé teszi a C++ nyelvi eszközök használatát a GPU-kon.
  - ▶ Nyilván azokat a részeket, amiket az eszköz nem támogat, nem lehet használni, ebben a közös probléma az exception handling, ezért lényegében a C++ STL-je nem használható GPUkon.
- ▶ Mindkét rendszer alól lehet a grafikai elemeket használni, így textúrákat, grafikus interopot, vektor regisztereket, stb.
- ▶ Mindkét rendszer single-source, azaz közös forrásfájlban található az eszköz és a gazda oldali kód (ellentétben pl. az OpenCL-el).

# CUDA vs SYCL

## Különbségek:

- ▶ A CUDA támogat dinamikus allokációt az eszközoldali kódban, a SYCL nem.
- ▶ A CUDA fordítója az nvcc, amely PTX köztest állít elő
- ▶ A SYCL forráskód először C++ kódra és SPIR / SPIRV köztesre fordul, majd a C++ kódot egy gazda oldali fordító fordítja binárissá, amely a megfelelő OpenCL hívásokkal betölti a köztest.

# CUDA vs SYCL

## Különbségek:

- ▶ A CUDA egy **nyelvi kiterjesztés**, a függvényeket annotálni kell, hogy az eszközön, a gazdaoldalon, vagy mindkettőn lesznek majd meghívva. SYCL-ben ilyen megkülönböztetés nincs.

//Ez csak host oldalon hívható:

```
__global__ int sq(int x){ return x*x; }
```

//Ez csak az eszközön:

```
__device__ int sq(int x){ return x*x; }
```

//Ez mindkettőn:

```
__global__ __device__ int sq(int x){ return x*x; }
```

# CUDA vs SYCL

## Különbségek:

- ▶ SYCL-ben nincs megkülönböztetve a kód, nem kell a függvényeket és lambdákat annotálni. A fordító ismeri fel, hogy melyik függvény az, ami majd az eszköz hívás belépési pontja lesz és az abban hivatkozott függvényeket megpróbálja az eszközre is lefordítani.
- ▶ Amíg nem használunk olyan funkcionalitást az eszközoldali kódon belül, amit nem lehet a GPUra lefordítani, addig a fordító szó nélkül feldolgozza a tetszőleges (akár nem is GPUra szánt) kódokat.



# CUDA vs SYCL

Jelenlegi állapot:

- ▶ CUDA 10 (2018 szeptember óta)
  - ▶ C++ 14 támogatás az eszközoldali kódban is, helyenként limitációkkal
- ▶ SYCL 1.2.1 konform GPUs implementáció:  
ComputeCpp 1.0.2 (2018 október)
  - ▶ Clang 6.0 alapú, így a legtöbb c++17 szolgáltatás elérhető benne

# CUDA vs SYCL

Példakódok, SDK:

A CUDA példakódokat a [CUDA telepítő teszi fel](#).

[CUDA dokumentáció](#)

[SYCL specifikáció](#)

Compute Cpp SDK: sok hasznos példakód:

<https://github.com/codeplaysoftware/computecpp-sdk>

# CUDA vs SYCL

Hibakezelés:

CUDA:

Error kódokat használ

```
__host__ __device__ const char* cudaGetErrorName ( cudaError_t error )  
Returns the string representation of an error code enum name.
```

```
__host__ __device__ const char* cudaGetErrorString ( cudaError_t error )  
Returns the description string for an error code.
```

```
__host__ __device__ cudaError_t cudaGetLastError ( void )  
Returns the last error from a runtime call.
```

```
__host__ __device__ cudaError_t cudaPeekAtLastError ( void )  
Returns the last error from a runtime call.
```

# CUDA vs SYCL

Hibakezelés:

SYCL:

Exception-ök

```
int main(){
    try
    {
        auto plats = cl::sycl::platform::get_platforms();
        if (plats.empty()) throw std::runtime_error{ "No OpenCL platform found." };
    }
    catch (cl::sycl::exception e){ }
    catch (std::exception e){}
    return 0;
}
```

# Funkcionális primitívek GPU-n

A három legfontosabb funkcionális primitív, a map, zip, reduce hatékonyan párhuzamosítható, így ezeket implementáljuk először a GPU-kra.

# CUDA map

A teljes kód itt érhető el:

<https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/map.cu>

# CUDA map

Az eszközoldali kód törzse:

```
template<typename F, typename T>
__global__ void map(F f, T* src, T* dst)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    dst[id] = f(src[id]);
}
```

# CUDA map

Az eszközoldali kód törzse:

Ez a belépési pont a gazdaoldalról lesz hívva.

```
template<typename F, typename T>
__global__ void map(F f, T* src, T* dst)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    dst[id] = f(src[id]);
}
```




# CUDA map

Az eszközoldali kód törzse:

```
template<typename F, typename T>
__global__ void map(F f, T* src, T* dst)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    dst[id] = f(src[id]);
}
```

Az eszközoldalon beépített változók a számítási tartomány paraméterei (mint openCL-ben)



# CUDA map

Az eszközoldali kód törzse:

```
template<typename F, typename T>
__global__ void map(F f, T* src, T* dst)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    dst[id] = f(src[id]);
}
```

A meghívott f függvénynek **\_\_device\_\_** annotáltnak kell lennie!

# CUDA map

A gazdaoldali kód törzse:

```
int main()
{
    using T = double;    using P = T*;

    int n = 16;    size_t sz = n*sizeof(T);
    std::vector<T> h_src(n), h_dst(n);
    P d_src, d_dst;

    cudaMalloc( &d_src, sz );
    cudaMalloc( &d_dst, sz );

    cudaMemcpy( d_src, h_src.data(), sz, cudaMemcpyHostToDevice);
    cudaMemcpy( d_dst, h_dst.data(), sz, cudaMemcpyHostToDevice);
}
```

# CUDA map

A gazdaoldali kód törzse:

```
int main()
{
    using T = double;    using P = T*;

    int n = 16;    size_t sz = n*sizeof(T);
    std::vector<T> h_src(n), h_dst(n);
    P d_src, d_dst;

    cudaMalloc( &d_src, sz );
    cudaMalloc( &d_dst, sz );

    cudaMemcpy( d_src, h_src.data(), sz, cudaMemcpyHostToDevice);
    cudaMemcpy( d_dst, h_dst.data(), sz, cudaMemcpyHostToDevice);
}
```

Memória foglalás az eszközön



# CUDA map

A gazdaoldali kód törzse:

```
int main()
{
    using T = double;    using P = T*;

    int n = 16;    size_t sz = n*sizeof(T);
    std::vector<T> h_src(n), h_dst(n);
    P d_src, d_dst;

    cudaMalloc( &d_src, sz );
    cudaMalloc( &d_dst, sz );
    cudaMemcpy( d_src, h_src.data(), sz, cudaMemcpyHostToDevice);
    cudaMemcpy( d_dst, h_dst.data(), sz, cudaMemcpyHostToDevice);
}
```

Memória másolás az eszközre



# CUDA map

A gazdaoldali kód törzse:

```
int blockSize = 4;
int gridSize = (int)ceil((float)n/blockSize);

auto sq = [] __device__ (auto const& x){ return x*x; };

map<<<gridSize, blockSize>>>(sq, d_src, d_dst);
```

# CUDA map

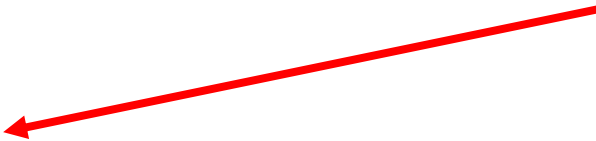
A gazdaoldali kód törzse:

```
int blockSize = 4;
int gridSize = (int)ceil((float)n/blockSize);

auto sq = [] __device__ (auto const& x){ return x*x; };

map<<<gridSize, blockSize>>>(sq, d_src, d_dst);
```

A teljes elemszámot felosztjuk  
4-es blokkokra



# CUDA map

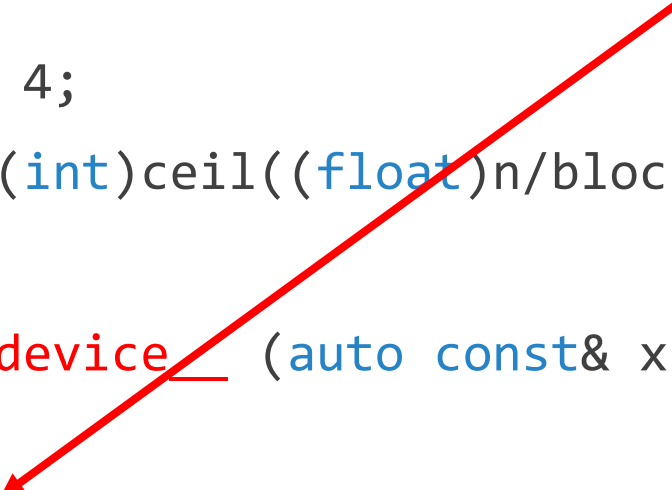
A gazdaoldali kód törzse:

Itt adjuk át a méreteket az eszköznek: a blokkok számát és az egy blokkon belüli szálak számát

```
int blockSize = 4;
int gridSize = (int)ceil((float)n/blockSize);

auto sq = [] __device__ (auto const& x){ return x*x; };

map<<<gridSize, blockSize>>>(sq, d_src, d_dst);
```





# CUDA map

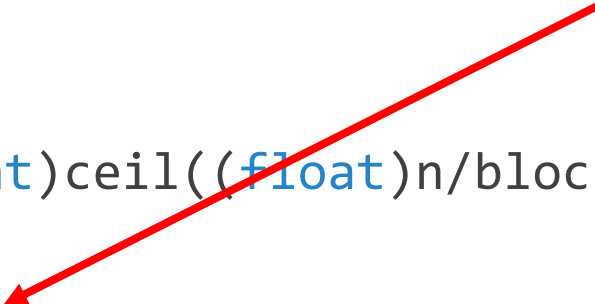
A gazdaoldali kód törzse:

Az eszköz oldalon alkalmazandó lambda

```
int blockSize = 4;
int gridSize = (int)ceil((float)n/blockSize);

auto sq = [] __device__ (auto const& x){ return x*x; };

map<<<gridSize, blockSize>>>(sq, d_src, d_dst);
```



# CUDA map

A gazdaoldali kód törzse:

```
cudaMemcpy( h_dst.data(), d_dst, sz, cudaMemcpyDeviceToHost );
```

```
cudaFree(d_src);
```

```
cudaFree(d_dst);
```

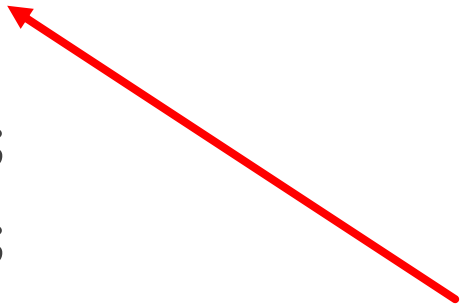
# CUDA map

A gazdaoldali kód törzse:

```
cudaMemcpy( h_dst.data(), d_dst, sz, cudaMemcpyDeviceToHost );
```

```
cudaFree(d_src);
```

```
cudaFree(d_dst);
```



Az eszközoldali hívás után a memóriát vissza kell másolni a gazda oldalra

# CUDA map

A gazdaoldali kód törzse:

```
cudaMemcpy( h_dst.data(), d_dst, sz, cudaMemcpyDeviceToHost );
```

```
cudaFree(d_src);
```

```
cudaFree(d_dst);
```



Ha végeztünk az eszköz oldali memóriát  
fel kell szabadítani

# CUDA map


Fordítás:

- ▶ Ha CMAKE-el szeretnénk fordítani, nézzük meg az [itteni](#) példát.
- ▶ Paransorból kézzel:
  - ▶ `nvcc -std=c++14 --expt-extended-lambda map.cu`

# CUDA map

Fordítás:

- ▶ Ha CMAKE-el szeretnénk fordítani, nézzük meg az [itteni](#) példát.
- ▶ Paransorból kézzel:
  - ▶ `nvcc -std=c++14 --expt-extended-lambda map.cu`



A lambdák annotálásához szükséges  
kiterjesztés engedélyezése

# CUDA zip

A teljes kód itt érhető el:

<https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/zip.cu>

# CUDA zip

A zip teljesen hasonló a map-hoz:

```
template<typename F, typename T>
__global__ void zip(F f, T* src1, T* src2, T* dst)
{
    int id = blockIdx.x * blockDim.x + threadIdx.x;
    dst[id] = f(src1[id], src2[id]);
}
```



# És most ugyan ez SYCL-ben!

# SYCL map

A teljes kód itt érhető el:

[https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/map\\_sycl.cpp](https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/map_sycl.cpp)

# SYCL map

A sycl használatához be kell includolni a sycl.hpp-t:

```
// SYCL include  
#include <CL/sycl.hpp>
```

# SYCL map

A map függvény belsejét külön függvényben írtuk meg:

```
template<typename F, typename RA, typename WA>  
void map(F f, RA src, WA dst, cl::sycl::id<1> id)  
{  
    dst[id] = f(src[id]);  
}
```

# SYCL map

A map függvény belsejét külön függvényben írtuk meg:

```
template<typename F, typename RA, typename WA>  
void map(F f, RA src, WA dst, cl::sycl::id<1> id)  
{  
    dst[id] = f(src[id]);  
}
```

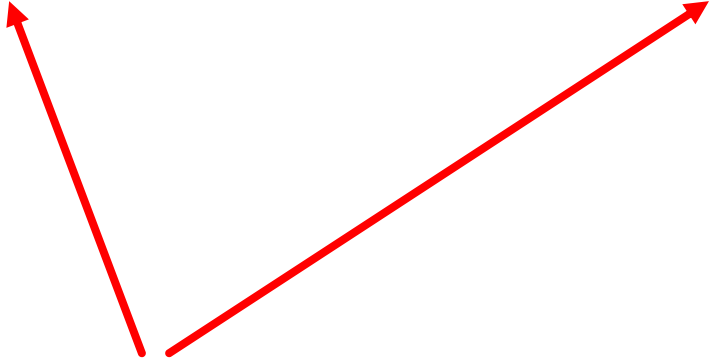


Nincs semmilyen extra annotáció,  
szemben a CUDA-val

# SYCL map

A map függvény belsejét külön függvényben írtuk meg:

```
template<typename F, typename RA, typename WA>  
void map(F f, RA src, WA dst, cl::sycl::id<1> id)  
{  
    dst[id] = f(src[id]);  
}
```



Az itteni megvalósításban a bemenő argumentumok accessorok, és a szárazonosító is argumentumként jön a függvénybe (nincsenek beépített változók, mint CUDA-ban)

# SYCL map

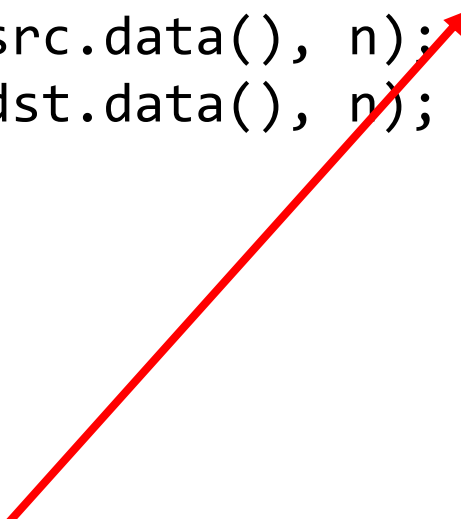
A SYCL kód gazda oldali része:

```
{  
    cl::sycl::queue queue{ cl::sycl::gpu_selector() };  
    cl::sycl::buffer<T, 1> b_src(h_src.data(), n);  
    cl::sycl::buffer<T, 1> b_dst(h_dst.data(), n);  
    cl::sycl::range<1> r(n);  
  
}
```

# SYCL map

A SYCL kód gazda oldali része:

```
{  
    cl::sycl::queue queue{ cl::sycl::gpu_selector() };  
    cl::sycl::buffer<T, 1> b_src(h_src.data(), n);  
    cl::sycl::buffer<T, 1> b_dst(h_dst.data(), n);  
    cl::sycl::range<1> r(n);  
}
```



Parancslista létrehozása, ami most az első GPU eszközt próbálja meg kiválasztani

```
}
```



# SYCL map

A SYCL kód gazda oldali része:

```
{  
    cl::sycl::queue queue{ cl::sycl::gpu_selector() };  
    cl::sycl::buffer<T, 1> b_src(h_src.data(), n);  
    cl::sycl::buffer<T, 1> b_dst(h_dst.data(), n);  
    cl::sycl::range<1> r(n);  
}
```

↑  
Bufferek létrehozása az adatokhoz

# SYCL map

A SYCL kód gazda oldali része:

```
{
    auto sq = [](auto const& x){ return x*x; };
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)
        {
            map(sq, a_src, a_dst, i); });
    });
}
```

# SYCL map

A SYCL kód gazda oldali része:

Semmilyen annotáció nem  
kell a lambdához sem

```
{
    auto sq = [](auto const& x){ return x*x; };
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)
        {
            map(sq, a_src, a_dst, i); });
    });
}
```

# SYCL map

A SYCL kód gazda oldali része:

```
{
    auto sq = [](auto const& x){ return x*x; };
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)
        {
            map(sq, a_src, a_dst, i); });
    });
}
```

Hozzáférést kérünk  
a bufferekhez



# SYCL map

A SYCL kód gazda oldali része:

```
{
    auto sq = [](auto const& x){ return x*x; };
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)
        {
            map(sq, a_src, a_dst, i); });
    });
}
```

Egy parallel\_for hívást hajtunk végre a megadott kernel névvel...

# SYCL map

A SYCL kód gazda oldali része:

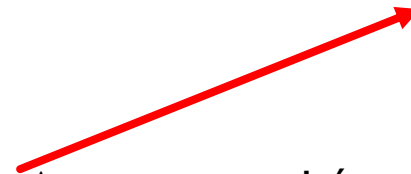
```
{
    auto sq = [](auto const& x){ return x*x; };
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)
        {
            map(sq, a_src, a_dst, i); });
    });
}
```

↑  
Átveszünk mindent  
érték szerint,  
ideértve az  
accessorokat is

# SYCL map

A SYCL kód gazda oldali része:

```
{
    auto sq = [](auto const& x){ return x*x; };
    queue.submit([&](cl::sycl::handler& cgh)
    {
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)
        {
            map(sq, a_src, a_dst, i); });
    });
}
```



Argumentumként  
kapjuk a  
szálazonosítót

# SYCL map

A SYCL kód gazda oldali része:

```
{  
    auto sq = [](auto const& x){ return x*x; };  
    queue.submit([&](cl::sycl::handler& cgh)  
    {  
        auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);  
        auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);  
        cgh.parallel_for<class Map>(r, [=](cl::sycl::id<1> i)  
        {  
            map(sq, a_src, a_dst, i); });  
    });  
}
```

Meghívjuk a  
korábban megírt  
függvényünket.



# SYCL fordítás

A ComputeCpp-vel, CMAKE-el történő fordításhoz, nézzük meg ezt a példakódot:

<https://github.com/Wigner-GPU-Lab/Teaching/tree/master/GPGPU1/SYCL>

Amihez szükség van a FindComputeCpp CMAKE modulra, aminek a legfrissebb verziója innen tölthető le:

<https://github.com/codeplaysoftware/computecpp-sdk/tree/master/cmake/Modules>

# SYCL fordítás

Ha nem akarunk CMAKE-el fordítani, akkor parancssorból kb. így fest a dolog:

Először a compute++ fordító kiszedi a cpp fájlból a SYCL specifikus részeket és legenerál egy úgynevezett „integration headert” (esetünkben main.cpp.sycl):

```
/opt/Codeplay/ComputeCpp/latest/bin/compute++ -std=c++14 -O2 -mllvm -inline-threshold=1000  
-sycl -emit-llvm -intelspirmetadata -isystem /opt/Codeplay/ComputeCpp/latest/include/  
-I/opt/Codeplay/ComputeCpp/latest/include/ -I/usr/include -o main.cpp.sycl -c main.cpp
```

Ezután a szokásos módon lefordítjuk a forrásfájlt, CSAK hozzá kell adni a generált headert includeként expliciten!:

```
g++ -isystem /opt/Codeplay/ComputeCpp/latest/include -O3 -DNDEBUG -include main.cpp.sycl  
-std=gnu++14 -o main.o -c main.cpp
```

Végül linkelünk (a `-Wl, -rpath` varázslásról [bővebben itt](#)):

```
g++ -O3 -DNDEBUG main.o -o SYCL_app -rdynamic /opt/Codeplay/ComputeCpp/latest/lib/libComputeCpp.so  
-lOpenCL -Wl,-rpath,/opt/Codeplay/ComputeCpp/latest/lib
```

# SYCL zip

A teljes kód itt:

[https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/zip\\_sycl.cpp](https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/zip_sycl.cpp)

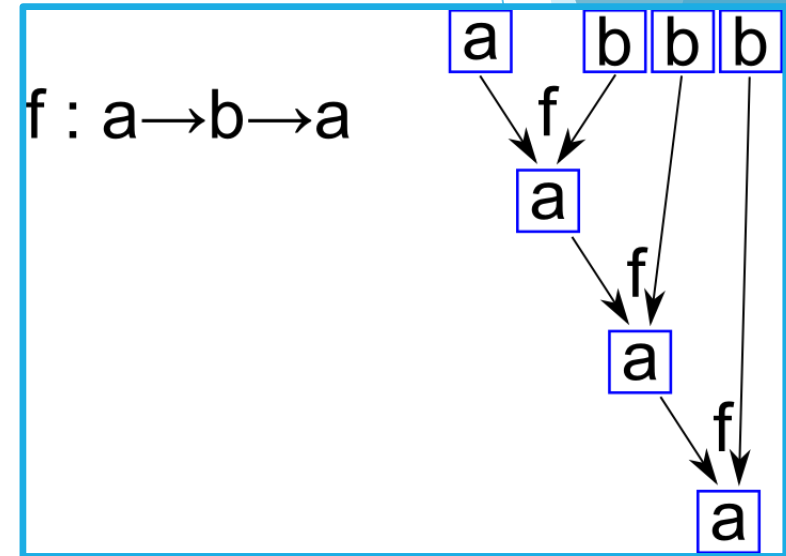
# Redukciók GPU-n

# Redukciók GPU-n

A korábban bevezetett primitívek között a foldl/r szerepelt mint redukciós primitív:

$\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow [b] \rightarrow a$

Ezzel az a probléma, hogy nem párhuzamosítható, mert explicit végrehajtási sorrendet ír elő.



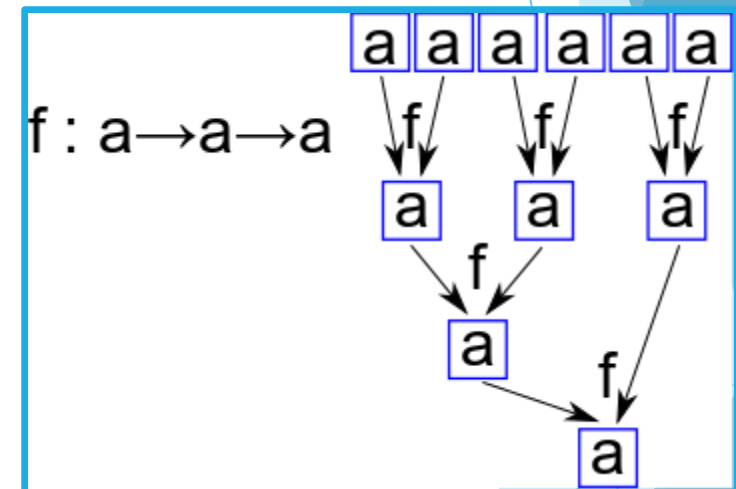
# Redukciók GPU-n

A korábban bevezetett primitívek között a foldl/r szerepelt mint redukciós primitív:

$\text{foldl} :: (a \rightarrow b \rightarrow a) \rightarrow a \rightarrow f \ b \rightarrow a$

Ha megváltoztatjuk a szignatúrát, és feltételezzük, hogy **a bináris művelet asszociatív és kommutatív**, akkor lehetőségünk nyílik átrendezni és párhuzamosítani a műveletet:

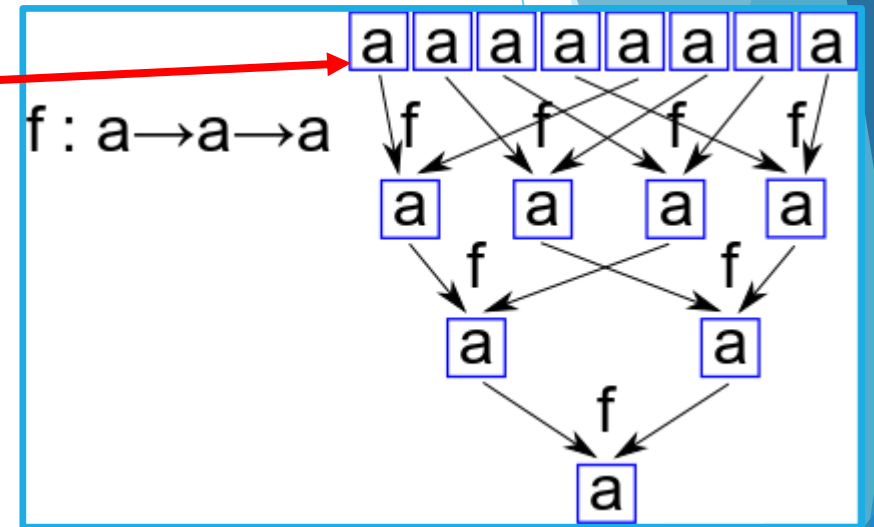
$\text{reduce} :: (a \rightarrow a \rightarrow a) \rightarrow f \ a \rightarrow a$



# Redukciók GPU-n

A redukciót a következő logikával végezzük el:

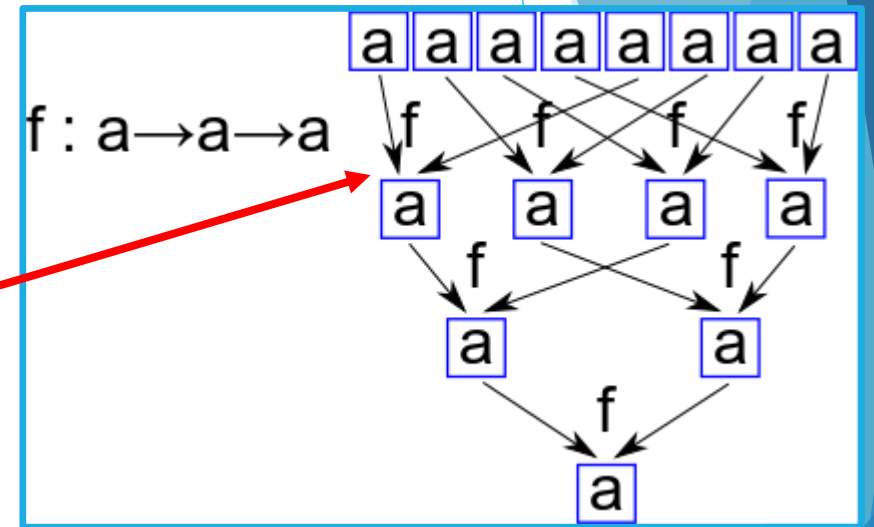
- ▶ Az első globális memóriából történő olvasásnál két elemet olvasunk be, eltolva a blokk mérettel (**koaleszkált memória hozzáférés**)



# Redukciók GPU-n

A redukciót a következő logikával végezzük el:

- ▶ Az első globális memóriából történő olvasásnál két elemet olvasunk be, eltolva a blokk mérettel (**koaleszkált memória hozzáférés**)
- ▶ ezeket azonnal redukáljuk és letároljuk a lokális (shared) memóriában.

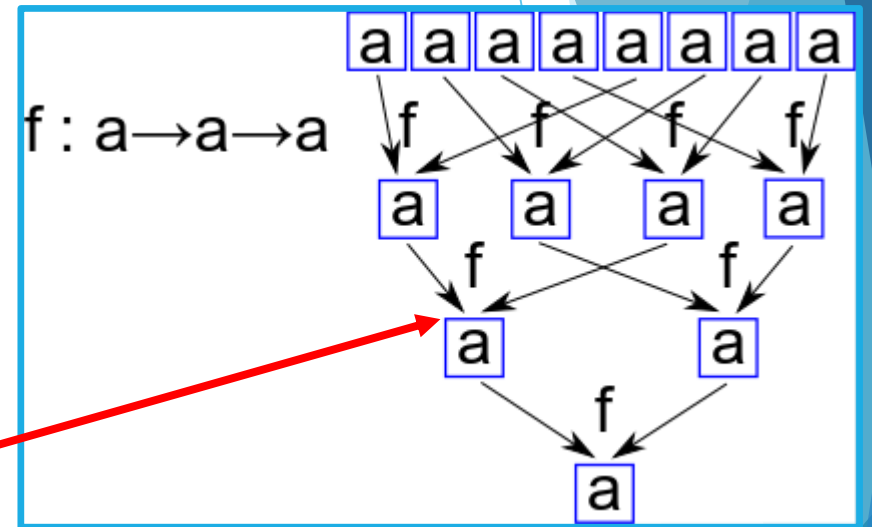




# Redukciók GPU-n

A redukciót a következő logikával végezzük el:

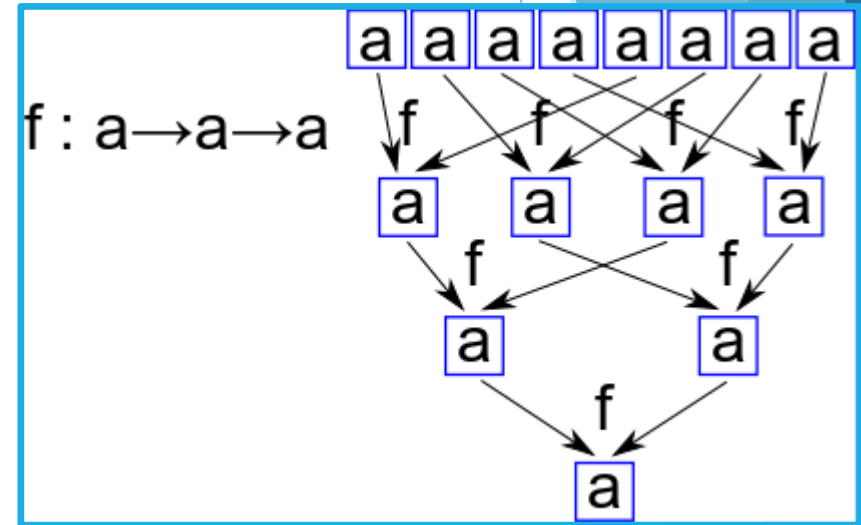
- ▶ Az első globális memóriából történő olvasásnál két elemet olvasunk be, eltolva a blokk mérettel (**koaleszkált memória hozzáférés**)
- ▶ ezeket azonnal redukáljuk és letároljuk a lokális (shared) memóriában.
- ▶ A további redukciót is ugyan itt végezzük, minden lépésben felezve az eltolást (**a megosztott memória bank-conflictjainak elkerülésére**)



# Redukciók GPU-n

A redukciót a következő logikával végezzük el:

- ▶ Az első globális memóriából történő olvasásnál két elemet olvasunk be, eltolva a blokk mérettel (**koaleszkált memória hozzáférés**)
- ▶ ezeket azonnal redukáljuk és letároljuk a lokális (shared) memóriában.
- ▶ A további redukciót is ugyan itt végezzük, minden lépésben felezve az eltolást (**a megosztott memória bank-conflictjainak elkerülésére**)
- ▶ Végeredményben minden munkacsoport (blokk) egyetlen értéket fog előállítani. Ennek a redukálását vagy gazda oldalon végezzük, vagy egy külön kernel indításban.



# CUDA reduce

A teljes kód itt:

<https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/reduce.cu>

# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto t = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[t] = f( src[i], src[i+blockDim.x] );
    __syncthreads();

    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(t < s){ tmp[t] = f(tmp[t], tmp[t + s]); }
        __syncthreads();
    }
    if(t == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();

    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(l < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Lokális memória bevezetése,  
a mérete a hívó oldalról jön.




# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();

    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(l < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Lokális (blokkon belüli) szálindex



# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();

    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(l < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Globális szárazonosító, de kétszeresére felhízalt blokkmérettel az eltolás miatt



# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();
    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(l < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Első redukció a globális olvasáskor, a két elem között egy blokkméret eltolás van



# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();
    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(l < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Lokális memóriát írtunk,  
és most olvasni fogjuk,  
ezért szinkronizálni kell  
a szálakat!

# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();

    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(1 < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Egyre csökkenő  
eltolással további  
redukció a lokális  
memóriában

Minden lépés végén  
szinkronizáció!

# CUDA reduce

```
template<typename F, typename T>
__global__ void reduce(F f, T* src, T* dst)
{
    extern __shared__ T tmp[];
    auto l = threadIdx.x;
    auto i = blockIdx.x * (2*blockDim.x) + threadIdx.x;
    tmp[l] = f( src[i], src[i+blockDim.x] );
    __syncthreads();

    for(auto s=blockDim.x/2; s > 0; s >>= 1)
    {
        if(l < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        __syncthreads();
    }
    if(l == 0){ dst[blockIdx.x] = tmp[0]; }
}
```

Végül egyetlen szál visszaírja a lokális memória elejéről a kimenő tömbbe a blokk eredményét

# CUDA reduce

Gazdaoldali felparaméterezés:

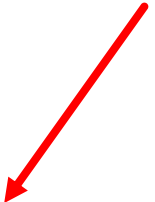
```
int n = 256;  
int blockSize = 16;  
size_t gridSize = (size_t)ceil((float)n/2/blockSize);  
reduce<<<gridSize, blockSize, blockSize*sizeof(T)>>>(sum, d_src, d_dst);
```

# CUDA reduce

Gazdaoldali felparaméterezés:

Minden szál 2 értéket olvas a globális memóriából, ezért a teljes méretet itt felezni kell!

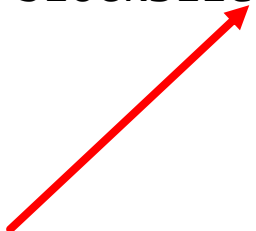
```
int n = 256;  
int blockSize = 16;  
size_t gridSize = (size_t)ceil((float)n/2/blockSize);  
reduce<<<gridSize, blockSize, blockSize*sizeof(T)>>>(sum, d_src, d_dst);
```



# CUDA reduce

Gazdaoldali felparaméterezés:

```
int n = 256;  
int blockSize = 16;  
size_t gridSize = (size_t)ceil((float)n/2/blockSize);  
reduce<<<gridSize, blockSize, blockSize*sizeof(T)>>>(sum, d_src, d_dst);
```



Itt kell megadni a lokális memória méretét  
byte-okban

# SYCL reduce

Hogy fest ugyanez SYCL-ben?

A teljes kód itt:

[https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/reduce\\_sycl.cpp](https://github.com/Wigner-GPU-Lab/Teaching/blob/master/GPGPU2/reduce_sycl.cpp)

# SYCL reduce

```
template<typename F, typename RA, typename RWA, typename WA>
void reduce(F f, RA src, RWA tmp, WA dst, 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);
    auto i = g * bs * 2 + l;
    tmp[l] = f( src[i], src[i+bs] );

    id.barrier(cl::sycl::access::fence_space::local_space);

    for(auto s=bs/2; s > 0; s >>= 1)
    {
        if(1 < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        id.barrier(cl::sycl::access::fence_space::local_space);
    }
    if(l == 0){ dst[g] = tmp[0]; }
}
```



# SYCL reduce

```
template<typename F, typename RA, typename RWA, typename WA>
void reduce(F f, RA src, RWA tmp, WA dst, 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);
    auto i = g * bs * 2 + l;
    tmp[l] = f( src[i], src[i+bs] );

    id.barrier(cl::sycl::access::fence_space::local_space);

    for(auto s=bs/2; s > 0; s >>= 1)
    {
        if(1 < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        id.barrier(cl::sycl::access::fence_space::local_space);
    }
    if(l == 0){ dst[g] = tmp[0]; }
}
```

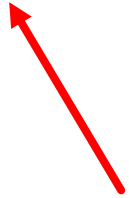
A lokális memóriát is accessorral érjük el,  
mint a buffereket!

# SYCL reduce

```
template<typename F, typename RA, typename RWA, typename WA>
void reduce(F f, RA src, RWA tmp, WA dst, 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);
    auto i = g * bs * 2 + l;
    tmp[l] = f( src[i], src[i+bs] );

    id.barrier(cl::sycl::access::fence_space::local_space);

    for(auto s=bs/2; s > 0; s >>= 1)
    {
        if(1 < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        id.barrier(cl::sycl::access::fence_space::local_space);
    }
    if(1 == 0){ dst[g] = tmp[0]; }
}
```



Mivel most lokális index is van,  
`nd_item`-et használunk

# SYCL reduce

```
template<typename F, typename RA, typename RWA, typename WA>
void reduce(F f, RA src, RWA tmp, WA dst, 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);
    auto i = g * bs * 2 + l;
    tmp[l] = f( src[i], src[i+bs] );

    id.barrier(cl::sycl::access::fence_space::local_space);

    for(auto s=bs/2; s > 0; s >>= 1)
    {
        if(1 < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        id.barrier(cl::sycl::access::fence_space::local_space);
    }
    if(1 == 0){ dst[g] = tmp[0]; }
}
```

Ez az indexelés pontosan a  
CUDA-s megvalósítást másolja

minden méretet és indexet az  
`nd_item`-ből tudunk lekérni

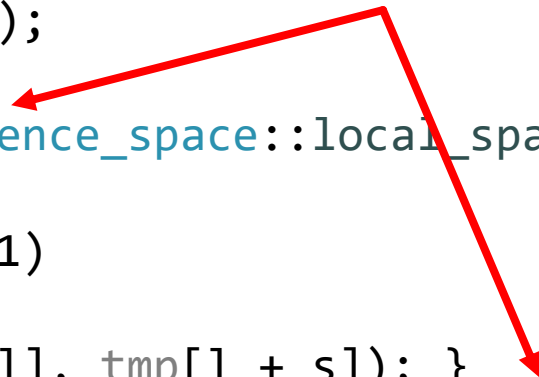
# SYCL reduce

```
template<typename F, typename RA, typename RWA, typename WA>
void reduce(F f, RA src, RWA tmp, WA dst, 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);
    auto i = g * bs * 2 + l;
    tmp[l] = f( src[i], src[i+bs] );

    id.barrier(cl::sycl::access::fence_space::local_space);

    for(auto s=bs/2; s > 0; s >>= 1)
    {
        if(1 < s){ tmp[l] = f(tmp[l], tmp[l + s]); }
        id.barrier(cl::sycl::access::fence_space::local_space);
    }
    if(1 == 0){ dst[g] = tmp[0]; }
}
```

Szintén az `nd_item`-en keresztül történik a szinkronizáció is.



# SYCL reduce

Gazdaoldali kód:

```
size_t n = 256, local_count = 16;
cl::sycl::nd_range<1> r(n/2, local_count);
queue.submit([&](cl::sycl::handler& cgh)
{
    auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
    auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
    cl::sycl::accessor<T, 1, cl::sycl::access::mode::read_write,
                      cl::sycl::access::target::local>
    a_tmp(cl::sycl::range<1>(local_count), cgh);

    cgh.parallel_for<class Reduce>(r, [=](cl::sycl::nd_item<1> i)
    { reduce(sum, a_src, a_tmp, a_dst, i); });
});
```

# SYCL reduce

Itt hozzuk létre az `nd_range`-t,  
ami a globális méretet és a  
blokkméretet írja le.

Gazdaoldali kód:

```
size_t n = 256, local_count = 16;  
cl::sycl::nd_range<1> r(n/2, local_count);  
queue.submit([&](cl::sycl::handler& cgh)  
{  
    auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);  
    auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);  
    cl::sycl::accessor<T, 1, cl::sycl::access::mode::read_write,  
                        cl::sycl::access::target::local>  
    a_tmp(cl::sycl::range<1>(local_count), cgh);  
  
    cgh.parallel_for<class Reduce>(r, [=](cl::sycl::nd_item<1> i)  
    { reduce(sum, a_src, a_tmp, a_dst, i); });  
});
```



# SYCL reduce

Gazdaoldali kód:

```
size_t n = 256, local_count = 16;
cl::sycl::nd_range<1> r(n/2, local_count);
queue.submit([&](cl::sycl::handler& cgh)
{
    auto a_src = b_src.get_access<cl::sycl::access::mode::read>(cgh);
    auto a_dst = b_dst.get_access<cl::sycl::access::mode::write>(cgh);
    cl::sycl::accessor<T, 1, cl::sycl::access::mode::read_write,
                      cl::sycl::access::target::local>
    a_tmp(cl::sycl::range<1>(local_count), cgh);

    cgh.parallel_for<class Reduce>(r, [=](cl::sycl::nd_item<1> i)
    { reduce(sum, a_src, a_tmp, a_dst, i); });
});
```

Ez az accessor a lokális memóriához ad hozzáférést.

A típusa a használatát, a konstruktora a méretet adja meg.

# GPU Reduce

A redukció még sokkal tovább optimalizálható a belső for ciklus kifejtésével.

Érdemes elolvasni [ezt](#) a redukció optimalizálásának gondolatmenetéről.



# GPU Reduce

A most ismertetett konstrukcióban kihasználtuk, hogy a bináris művelet kommutatív és asszociatív!

- ▶ Ha nem asszociatív, akkor csak foldolni tudunk.
- ▶ Ha asszociatív, de nem kommutatív, akkor párhuzamosíthatunk, de a memóriahozzáférés nem lesz ideális