Softwarové nástroje

Tomáš Oberhuber
tomas.oberhuber@fjfi.cvut.cz

10. března 2024

Videa na Youtube

TNL, Template Numerical Library

TNL = Template Numerical Library - **www.tnl-project.org**

- numerická knihovna pro moderní paralelní architektury
- nabízí jednotné rozhraní pro vícejádrová CPU a GPU
- vyvíjena v C++ a využívající moderních vlastností C++17
- $\approx$ 300,000 řádek kódu a dokumentace
- dostupná pod MIT licencí
- afiliovaný projekt sdružení Numfocus (www.numfocus.org)

NUMFOCUS [AFFILIATED PROJECT]

TEMPLATE NUMERICAL LIBRARY

TNL, Template Numerical Library

- Knihovna je dostupná na Gitlabu
  - https://gitlab.com/tnl-project/tnl
- `tnlcxx` je nástroj pro jednodušší kompilaci s TNL
  - https://gitlab.com/tnl-project/tnlcxx
- Knihovna obsahuje několik modulů:
  - TNL-MHFEM - Mixed-Hybrid Finite Element Method
  - TNL-LBM - Lattice Boltzmann method
  - TNL-SPH - Smoothed-Particle Hydrodynamics

TNL, pole a vektory

Správa paměti je v TNL řešena hlavně pomocí polí a vektorů.

Pole je reprezentováno šablonovou třídou `Array`:

```
1 template< typename Value = double,
2           typename Device = TNL::Devices::Host,
3           typename Index = int >
4 class TNL::Containers::Array { ... };
```

kde

- `Value` je typ prvků ukládaných v poli
- `Device` říká, kde má být pole alokováno
  - `TNL::Devices::Host` pro CPU a systeomovu paměť
  - `TNL::Devices::Cuda` pro GPU kompatibilní s CUDA
  - `TNL::Devices::Hip` pro GPU kompatibilní s HIP
- `Index` je typ pro indexování prvků pole
- ukázka kódu

TNL, pole a vektory

- Vektor je reprezentovaný třídou `Vector` se stejnými šablonovými parametry.
- Podporuje *výrazové šablony* (expression templates, ET) pro algebraické vektorové výrazy.

Např. výraz

$$\vec{x} = \vec{a} + 2\vec{b} + 3\vec{c}$$

lze vyhodnotit s pomocí Blas/Cublas pomocí kódu:

```
cublasHandle_t c_h;
cublasSaxpy(c_h,N,1.0,a,1,x,1); // -> x = a
cublasSaxpy(c_h,N,2.0,b,1,x,1); // -> x = x + 2 * b
cublasSaxpy(c_h,N,3.0,c,1,x,1); // -> x = x + 3 * c
```

S pomocí ET v TNL je to mnohem jednodušší

```
x = a + 2 * b + 3 * c;
```

... a **efektivnější !**

ET pro vektory

- ET je proxy objekt pro vektorový výraz
- umožňuje tzv. **lazy evaluation**
- $i-$tý prvek výrazu je vypočítaný, až když je přiřazován, tj.

```
ET[ i ] ≡ a[ i ] + 2 * b[ i ] + 3 * c[ i ]
```

- na CPU by pak vyhodnocení výrazyu proběhlo

```
for( int i = 0; i < n; i++ )
x[ i ] = a[ i ] + 2 * b[ i ] + 3 * c[ i ];
//        ^        = ET[ i ]              ^
```

- to vyžaduje pouze **jeden zápis** do $\vec{x}$ místo **třech zápisů a dvou čtení** jako u Blasu

ET pro vektory

Test sčítání vektorů: `x += a + b + c`.

Size	CPU			GPU		
	BLAS	TNL		cuBLAS	TNL	
	BW	BW	Speed-up	BW	BW	Speed-up
100k	19.3	41.5	**2.2**	194.7	236.5	**1.21**
200k	19.7	41.7	**2.1**	228.3	277.6	**1.21**
400k	17.3	35.9	**2.1**	218.3	330.9	**1.51**
800k	11.7	19.3	**1.6**	233.3	370.6	**1.58**
1.6M	10.4	17.0	**1.6**	249.6	403.4	**1.61**
3.2M	10.2	17.3	**1.7**	266.6	444.8	**1.66**
6.4M	10.2	17.3	**1.7**	276.6	471.3	**1.70**

BW = efektivní datová propustnost v GB/s,
testováno na GPU Nvidia P100 (16 GB HBM2 @ 732 GB/s, 3584 CUDA jader)
a Intel Core i7-5820K (3.3GHz, 16MB cache).

ET pro vektory - příklady

ET v TNL podporuje následující operátory a funkce:

- operátory: `+ - * / += -= *= /=`
- funkce: `sign abs sqrt sin cos tan asin acos atan sinh cosh tanh asinh acosh atanh exp log pow`
- minima a maxima: `minimum  maximum`
- porovnávání: `== !=`
- porovnávání po složkách: `equalTo notEqualTo less lessEqual greater greaterEqual`
- redukce: `sum min max dot all any argMin argMax`
- ET a redukce:

Parallel for

Pro obecnější algoritmy je často potřeba použít funkci `parallelFor`:

```
namespace TNL::Algorithms {

template< typename Device >
void parallelFor( int begin, int end, Function f );}
```

Toto je zjednodušená deklarace `parallelFor`.

Lambda funkce v C++

Lambda funkce v C++ jsou užitečné pokud potřebujeme vložit

- kód,
- data a proměnné,

do existujícího algoritmu.

```
1  auto f = [ <data  and variables> ] ( int idx ) mutable { <piece of code> };
```

Lambda funkce v C++

```
1  template< typename LambdaFunction >
2  void forLoop(int begin,
3               int end,
4               LambdaFunction function )
5  {
6     for( int i = begin; i < end; i++)
7        function( i );
8  }
9
10 int main( int argc, char* argv[] )
11 {
12    int* data = new int[ 10 ];
13    auto f = [=] ( int idx ) mutable{
14       data[ i ] = i;
15    };
16    forLoop( 0, 10, f );
17    delete[] data;
18 }
```

Lambda funkce v C++

```
template< typename LambdaFunction >
void forLoop(int begin,
             int end,
             LambdaFunction function )
{
   for( int i = begin; i < end; i++)
      function( i );
}

int main( int argc, char* argv[] )
{
   int* data = new int[ 10 ];
   auto f = [=] ( int idx ) mutable{
      data[ i ] = i;
   };
   forLoop( 0, 10, f );
   delete[] data;
}
```

```
int main( int argc, char* argv[] )
{
   int* data = new int[ 10 ];
   for( int i = begin; i < end; i++ )
      data[ i ] = i;
   delete[] data;
}
```

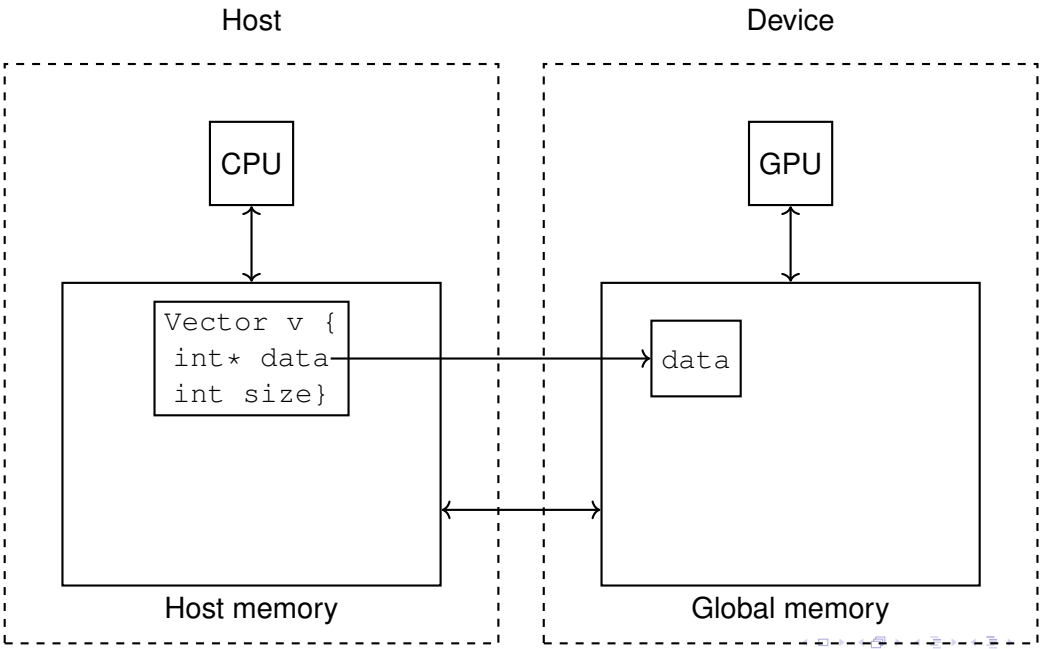
Lambda funkce v C++

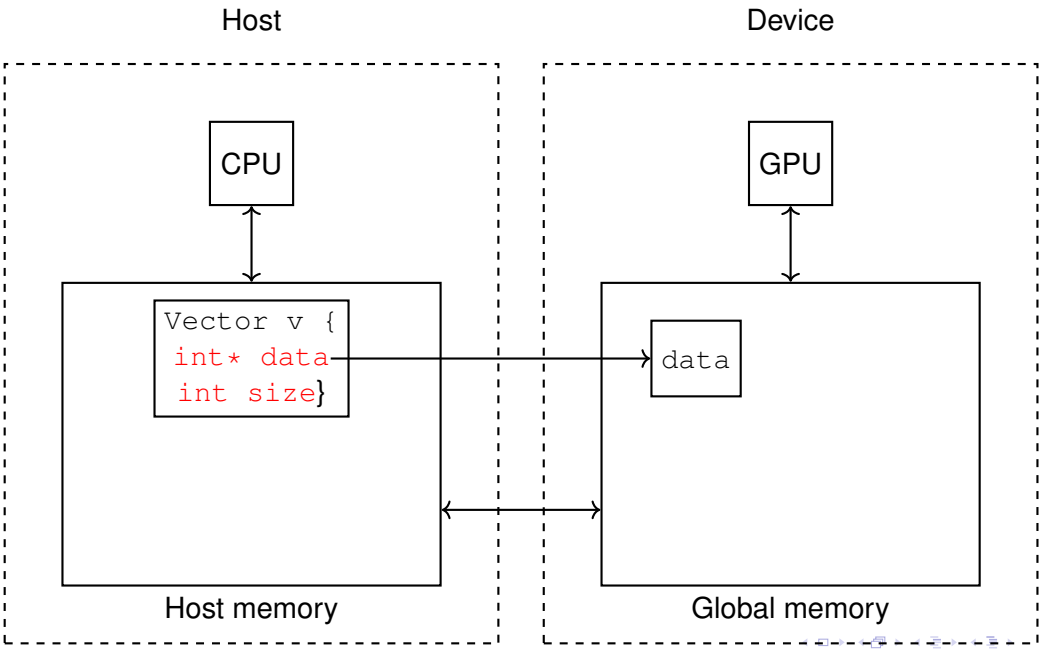
```
template< typename LambdaFunction >
void forLoop( int begin, int end, LambdaFunction function )
{
    for( int i = begin; i < end; i++ ) function( i );
}

int main( int argc, char* argv[] ) {
    TNL::Containers::Vector< int > v( 10 );
    std::cout << "Capturing by value:" << std::endl;
    auto f_value = [=] ( int idx ) mutable {
        v[ i ] = 0;
    };
    forLoop( 0, 10, f_value );
    std::cout << v << std::endl;

    std::cout << "Capturing by reference:" << std::endl;
    auto f_reference = [&] ( int idx ) mutable {
        v[ i ] = 0;
    };
    forLoop( 0, 10, f_reference );
    std::cout << v << std::endl;
}
```

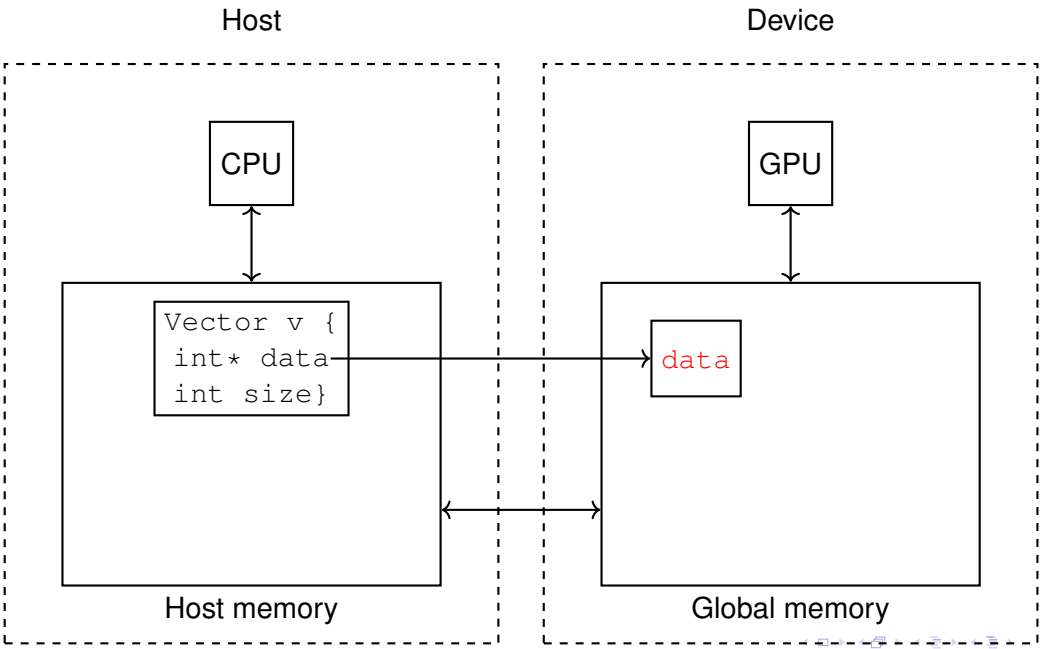
Lambda funkce v C++

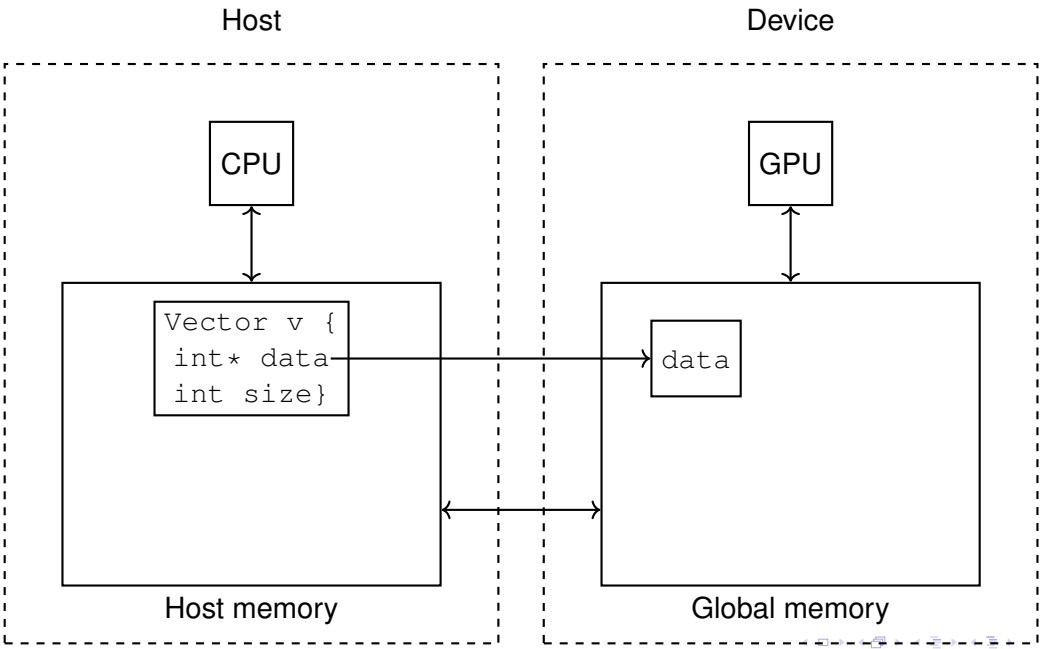
Předpokládejme nyní, že vektor v je alokovaný na GPU:

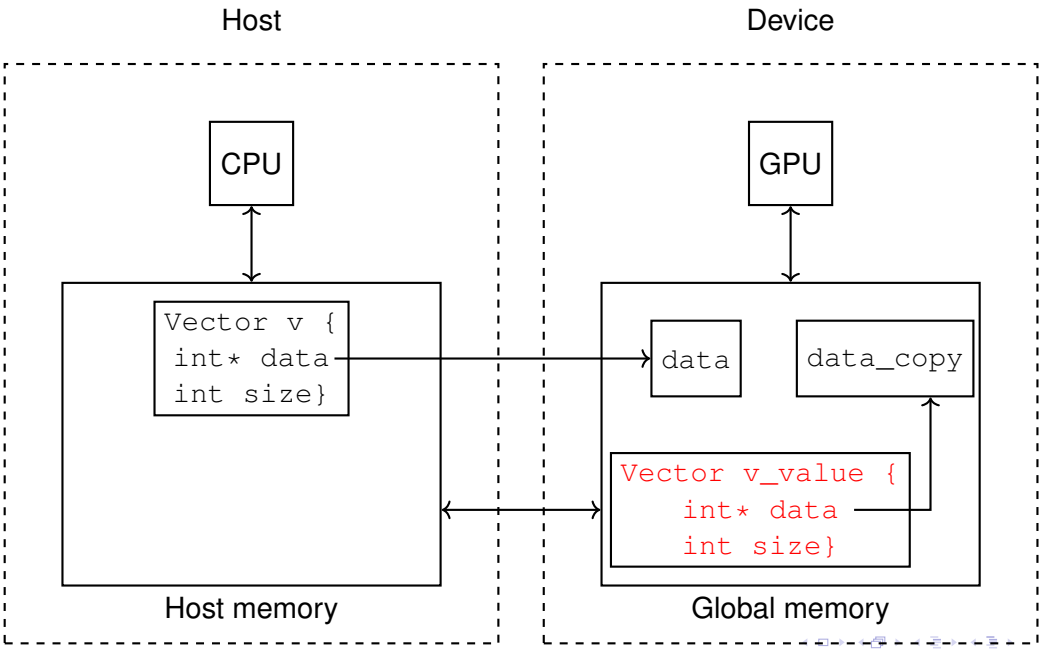
```
TNL::Containers::Vector< int, TNL::Devices::Cuda > v( 10 );
```

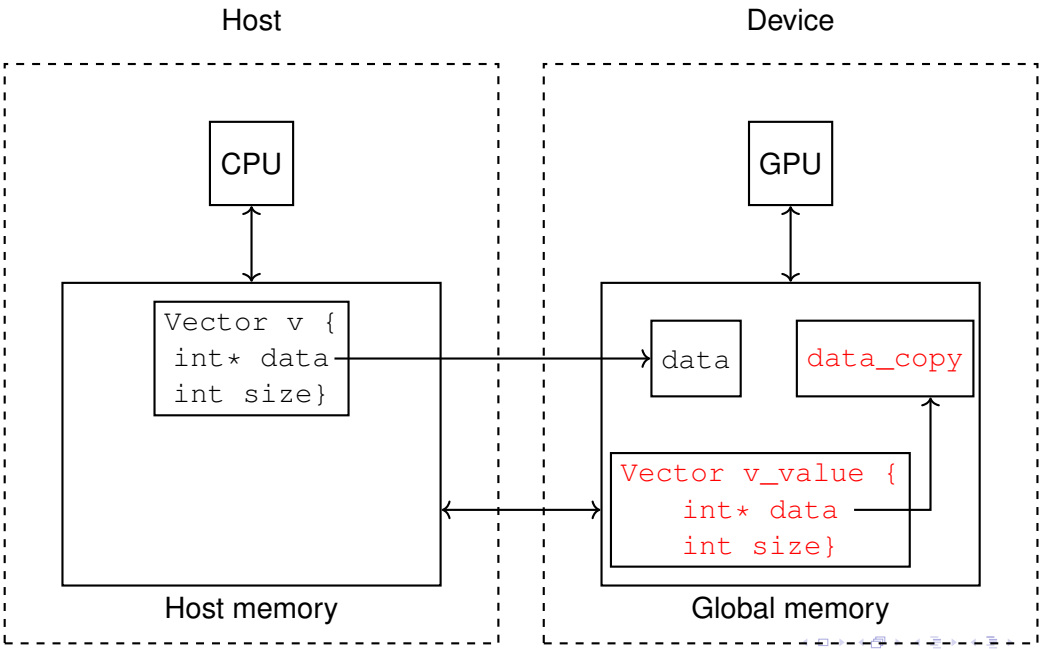

Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
Host memory
Global memory

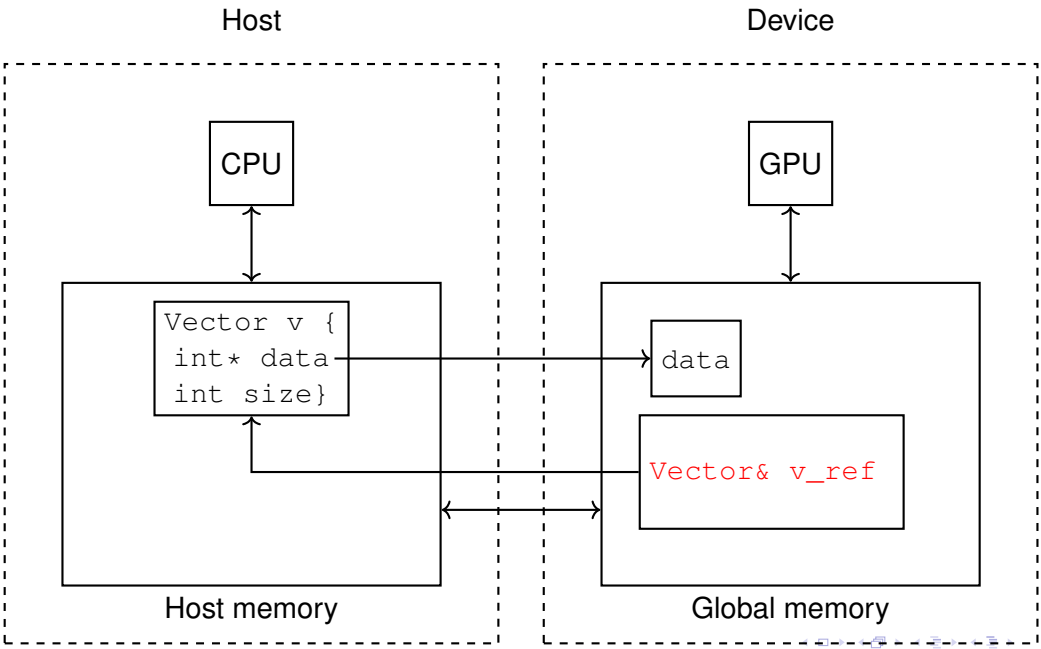

Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
Host memory
Global memory


Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
Host memory
Global memory


Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
Host memory
Global memory


Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
data_copy
Vector v_value {
int* data
int size}
Host memory
Global memory

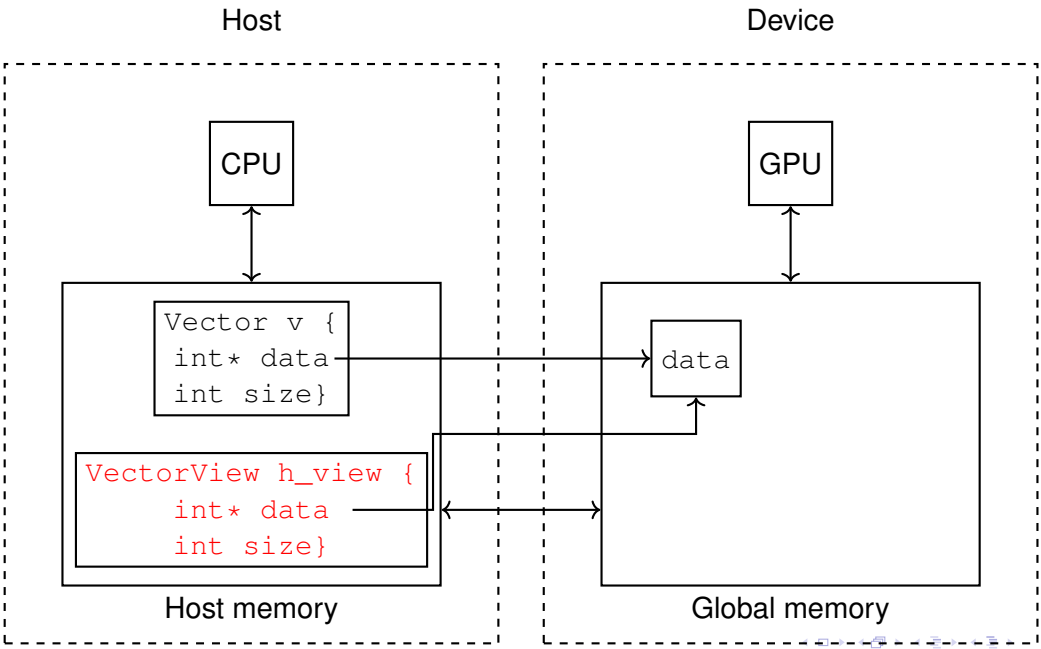

Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
data_copy
Vector v_value {
int* data
int size}
Host memory
Global memory

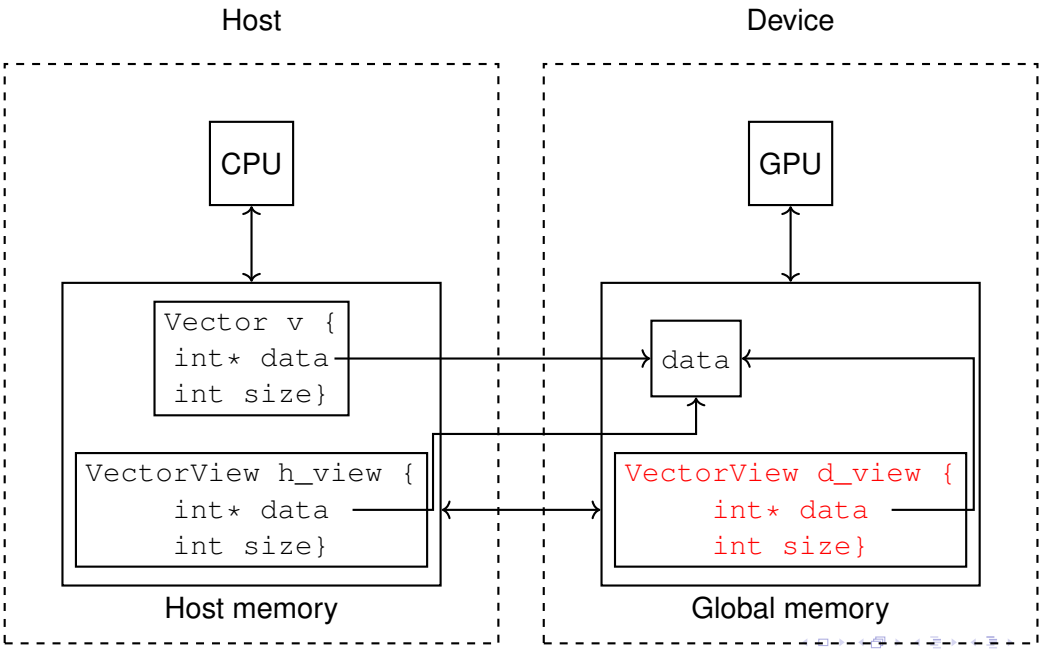

Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
Vector& v_ref
Host memory
Global memory

Řešením je použití `VectorView`:

```
1 template< typename Value = double,
2           typename Device = TNL::Devices::Host,
3           typename Index = int >
4 class TNL::Containers::VectorView { ... };
```

- `VectorView` (resp. `ArrayView`) má téměř identické rozhrání jako `Vector` (resp. `Array`).
- Neprovádí alokaci a dealokaci paměti, spravovaná data jsou sdílená.
- Dají se použít pro:
  - obalení dat alokovaných mimo knihovnu TNL,
  - rozdělení jednoho vektoru do více menších,
  - předávání dat mezi CPU a GPU.


Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
VectorView h_view {
int* data
int size}
Host memory
Global memory


Host
Device
CPU
GPU
Vector v {
int* data
int size}
data
VectorView h_view {
int* data
int size}
VectorView d_view {
int* data
int size}
Host memory
Global memory

Řešením je použití `VectorView`:

```
template< typename Value = double,
          typename Device = TNL::Devices::Host,
          typename Index = int >
class TNL::Containers::VectorView { ... };
```

- `VectorView` (resp. `ArrayView`) má téměř identické rozhrání jako `Vector` (resp. `Array`).
- Neprovádí alokaci a dealokaci paměti, spravovaná data jsou sdílená.
- Dají se použít pro:
  - obalení dat alokovaných mimo knihovnu TNL,
  - rozdělení jednoho vektoru do více menších,
  - předávání dat mezi CPU a GPU.

Podobné knihovny

- Thrust, rocThrust - C++ knihovny inspirované STL knihovnou pro programování GPU
- Kokkos - C++ programovací model pro vývoj portovatelných aplikací pro různé HPC architektury
- SYCL - je standard navržený skupinou Khronos Group definující abstrakci pro vývoj aplikací nad různými HPC platformami

AXPY příklady

`axpy` je rutina knihovny Blas počítající výraz

$$\vec{y} = \alpha\vec{x} + \vec{y}.$$

Ukážeme se její implementaci v různých knihovnách:

- CUDA
- Thrust
- Kokkos
- SYCL