# ICNPG 2023

No terminé de leer la clase 6. Trata de cómo usar la librería thrust de la mejor forma posible

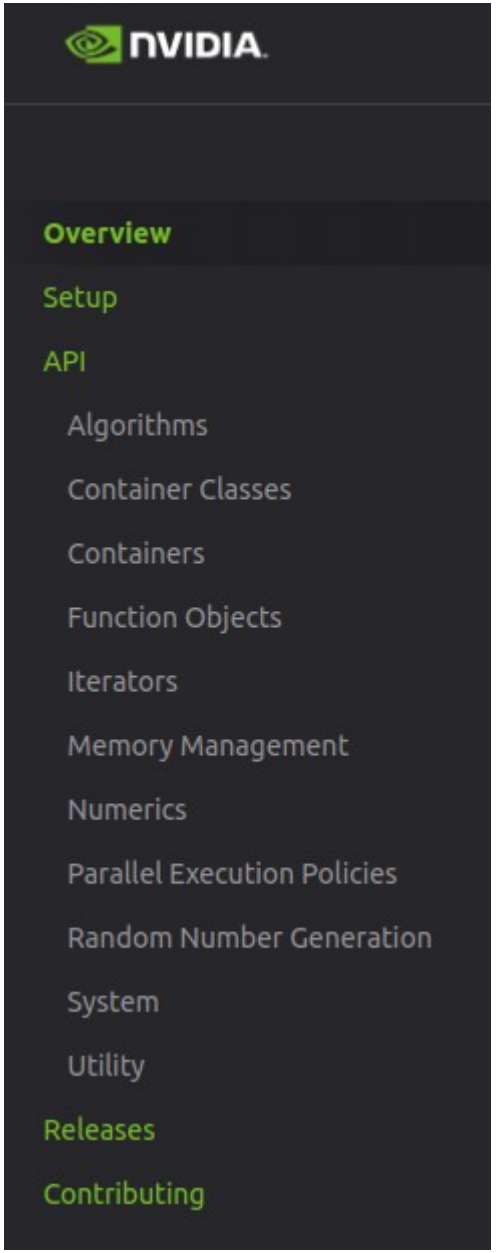
## Thrust



# ¿ Que es Thrust ?

**Thrust: The C++ Parallel Algorithms Library**


https://nvidia.github.io/thrust/



# Portabilidad: device backends

*Mismo programa, distintos ejecutables*

Serial en CPU: solo copiar los headers de thrust y usar el compilador de siempre

```
g++ -fopenmp -lgomp device_backends.cpp -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CPP -o cpp.out -I...
```

Paralelo en CPU: solo copiar los headers de thrust y usar el compilador de siempre

```
g++ -fopenmp -lgomp device_backends.cpp -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_OMP -o omp.out -I...
```

Paralelo en GPU: usar nvcc y gpu de nvidia...

```
nvcc device_backends.cu -DTHRUST_DEVICE_SYSTEM=THRUST_DEVICE_SYSTEM_CUDA -o cuda.out
```

Experimentar con device_backends.cu

**/share/apps/icnpg/clases/Clases_Thrust/clase_B/device_backends**

```
thrust::device_vector<float> dx=hx;

cronosin.tic();
float init=dx[0];
float suma = thrust::reduce(dx.begin(),dx.end(),init,thrust::minimum<float>());
```

# Execution policies: drop in replacement

```
#include <thrust/system/omp/execution_policy.h>
#include <thrust/sort.h>
#include <cstdlib>
#include <algorithm>
#include <iostream>
#include <vector>

int main(void)
{
  // serially generate 1M random numbers
  std::vector<int> vec(1 << 20);
  std::generate(vec.begin(), vec.end(), rand);

  // sort data in parallel with OpenMP by specifying its execution policy
  thrust::sort(thrust::omp::par, vec.begin(), vec.end());

  // report the largest number
  std::cout << "Largest number is " << vec.back() << std::endl;
}
```

# System-specific vector

```cpp
#include <thrust/system/omp/vector.h>
#include <thrust/sort.h>
#include <cstdlib>
#include <algorithm>
#include <iostream>

int main(void)
{
  // serially generate 1M random numbers
  thrust::omp::vector<int> vec(1 << 20);
  std::generate(vec.begin(), vec.end(), rand);

  // sort data in parallel with OpenMP
  thrust::sort(vec.begin(), vec.end());

  // no need to transfer data back to host

  // report the largest number
  std::cout << "Largest number is " << vec.back() << std::endl;

  return 0;
}
```

# Programación híbrida: system-specific vector

**/share/apps/icnpg/clases/Clases_Thrust/clase_B/omp_vs_cuda/ompvscuda.cu**

**Un solo Programa**

Paralelizacion para GPU

Paralelizacion para CPU multicore

**Simultáneamente...**

# C++ Standard Library

*Contributors to Wikimedia projects*

11-14 minutes

---

In the C++ programming language, the **C++ Standard Library** is a collection of classes and functions, which are written in the core language and part of the C++ ISO Standard itself.[1]

Un subconjunto de la std

Un laboratorio...

## Boost (C++ libraries)

*Contributors to Wikimedia projects*

5-6 minutes

---

Boost


**Boost** is a set of libraries for the C++ programming language that provides support for tasks and structures such as linear algebra, pseudorandom number generation, multithreading, image processing, regular expressions, and unit testing. It contains 164 individual libraries (as of version 1.76).[3]

# Standard Template Library

*Contributors to Wikimedia projects*

17-22 minutes

---

For other uses, see STL.

The **Standard Template Library** (**STL**) is a software library for the C++ programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms*, *containers*, *functions*, and *iterators*.[1]

# Standard Template Library

## *más abstracción sin menos eficiencia*

From Wikipedia, the free encyclopedia

- The Standard Template Library (STL) **is a software library for the C++** programming language that influenced many parts of the C++ Standard Library. It provides four components called *algorithms, containers, functional, and iterators*.

- The STL provides a set of common classes for C++, such as containers and associative arrays, that can be used with *any built-in type and with any user-defined type* that supports some elementary operations (such as copying and assignment).

- STL *algorithms are independent of containers*, which significantly reduces the complexity of the library.

- The STL achieves its results through the *use of templates*. This approach provides compile-time polymorphism that is often more efficient than traditional run-time polymorphism. Modern C++ compilers are tuned to minimize abstraction penalty arising from heavy use of the STL.

- The STL was created as *the first library of generic algorithms and data structures for C++,* with four ideas in mind: *generic programming,* ***abstractness without loss of efficiency***, the Von Neumann computation model, and value semantics...

# Thrust API (inspirada en STL)

**http://thrust.github.io/doc/modules.html**

thrust

Main Page | Modules | Namespaces ▾ | Classes ▾ | Files ▾

thrust

abs
acos
acosh
addressof
adjacent_difference
adjacent_difference
adjacent_difference
adjacent_difference
advance
all_of
all_of
any_of
any_of
arg
asin
asinh

◆ find() [2/2]

```
template<typename InputIterator , typename T >
InputIterator thrust::find ( InputIterator  first,
                             InputIterator  last,
                             const T &      value
                           )
```

`find` returns the first iterator `i` in the range `[first, last)` such that `*i == value` or `last` if no such iterator exists.

**Parameters**

**first** Beginning of the sequence to search.

**last** End of the sequence to search.

**value** The value to find.

**Returns**

The first iterator `i` such that `*i == value` or `last`.

# Thrust API “Snippets”

**http://thrust.github.io/doc/modules.html**

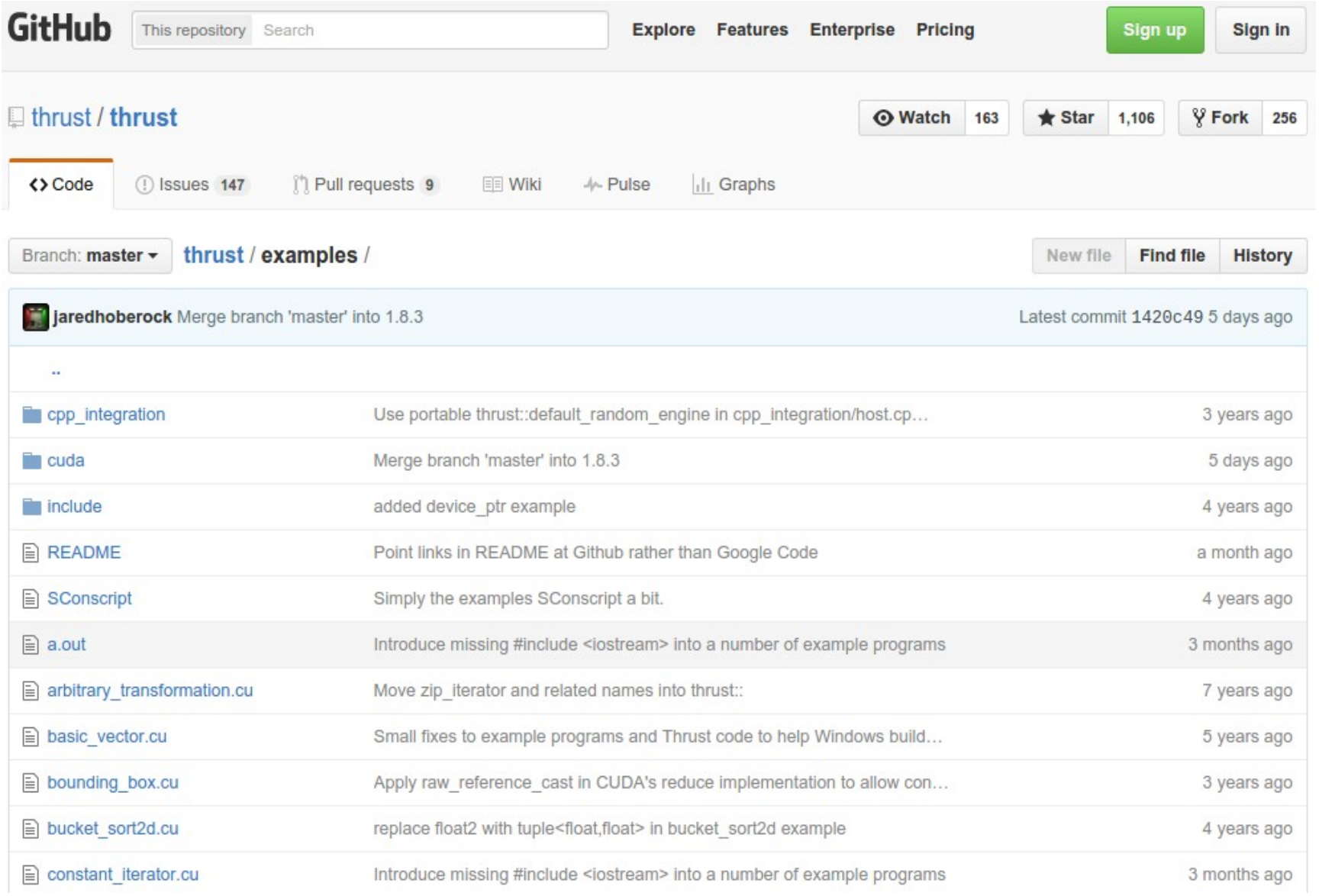
```
#include <thrust/find.h>
#include <thrust/device_vector.h>
...
thrust::device_vector<int> input(4);

input[0] = 0;
input[1] = 5;
input[2] = 3;
input[3] = 7;

thrust::device_vector<int>::iterator iter;

iter = thrust::find(input.begin(), input.end(), 3); // returns input.first() + 2
iter = thrust::find(input.begin(), input.end(), 5); // returns input.first() + 1
iter = thrust::find(input.begin(), input.end(), 9); // returns input.end()
```

**See also**

find_if

mismatch

# https://github.com/thrust/thrust/tree/master/examples



# Componentes de Thrust

- Containers:
- Para manejar el contenido en memoria del host y del device.
- Simplifican las copias D → H, H → D.
- Mucho mas que un array, son colecciones *genéricas* de objetos.

- Iteradores:
- Actúan como punteros...
- Pero mantienen info del espacio de memoria.

- Algoritmos:
- Se aplican a rangos de containers, delimitados por iteradores.
- Son *genéricos*, flexibles, y portables.

# Operadores predefinidos

- **General types and operators**

```
#include <thrust/reduce.h>

// declare storage
device_vector<int>   i_vec = ...
device_vector<float> f_vec = ...

// sum of integers (equivalent calls)
reduce(i_vec.begin(), i_vec.end());
reduce(i_vec.begin(), i_vec.end(),    0, plus<int>());

// sum of floats (equivalent calls)
reduce(f_vec.begin(), f_vec.end());
reduce(f_vec.begin(), f_vec.end(), 0.0f, plus<float>());

// maximum of integers
reduce(i_vec.begin(), i_vec.end(),    0, maximum<int>());
```

# Operadores propios

- **General types and operators**

```
struct negate_float2
{
    __host__ __device__
    float2 operator()(float2 a)
    {
        return make_float2(-a.x, -a.y);
    }
};

// declare storage
device_vector<float2> input  = ...
device_vector<float2> output = ...

// create functor
negate_float2 func;

// negate vectors
transform(input.begin(), input.end(), output.begin(), func);
```

# Comparadores propios

## General types and operators

```
// compare x component of two float2 structures
struct compare_float2
{
    __host__ __device__
    bool operator()(float2 a, float2 b)
    {
        return a.x < b.x;
    }
};

// declare storage
device_vector<float2> vec = ...

// create comparison functor
compare_float2 comp;

// sort elements by x component
sort(vec.begin(), vec.end(), comp);
```



Ejemplo:

**struct** Persona{
    ...datos de una persona;
};

…..

Persona p;
p.inicializacion();

**device_vector**<p> Gente(40000000);

**sort**(Gente.begin(),Gente.end(),
comparador_de_edad);

Gente[0] → datos de
la persona mas joven;

Gente[40000000-1] → datos del
Mas viejo;

# Predicados propios

## Operators with State

```
// compare x component of two float2 structures
struct is_greater_than
{
    int threshold;

    is_greater_than(int t) { threshold = t; }

    __host__ __device__
    bool operator()(int x) { return x > threshold; }
};

device_vector<int> vec = ...

// create predicate functor (returns true for x > 10)
is_greater_than pred(10);

// count number of values > 10
int result = count_if(vec.begin(), vec.end(), pred);
```



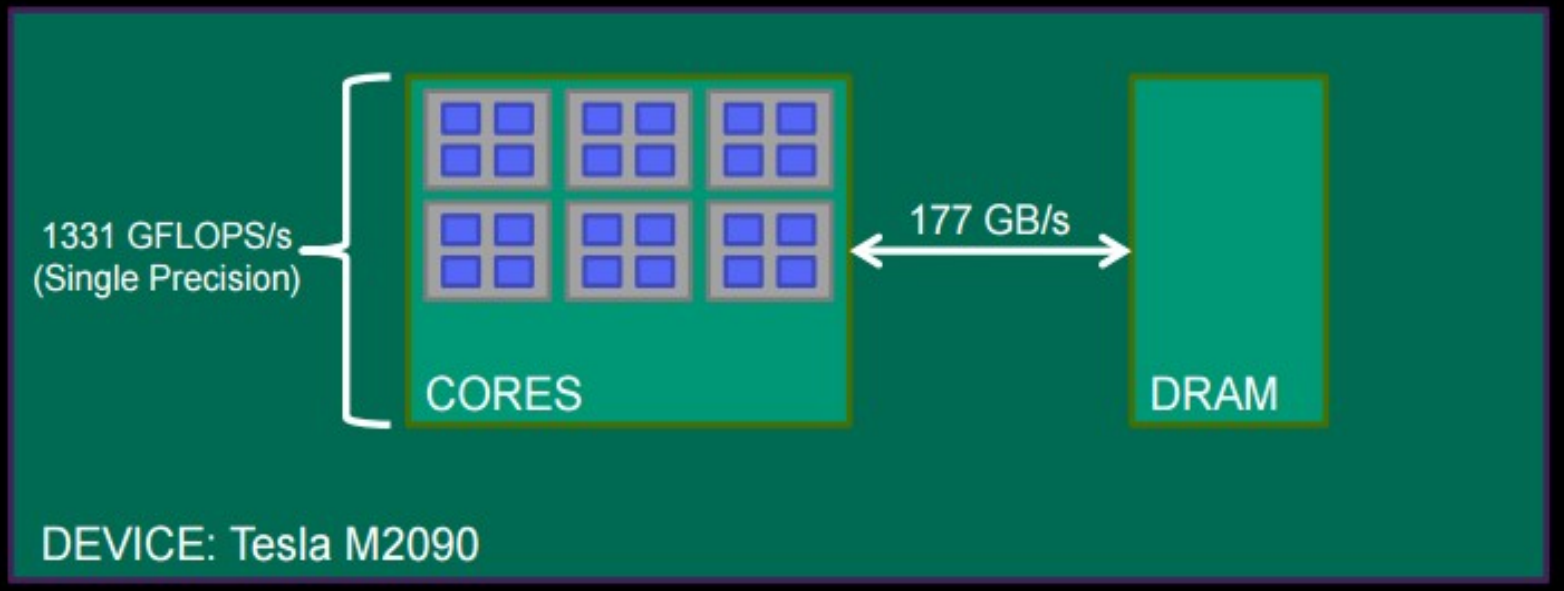
Ejemplo:

```
struct Persona{
        ...datos de una persona;
};

…..

Persona p;
p.inicializacion();

device_vector<p> Gente(40000000);

Int M =
count_if(Gente.begin(),Gente.end(),
vive_en_bariloche);

M ==
numero de personas que viven en Bche
```

# Thrust Best Practices

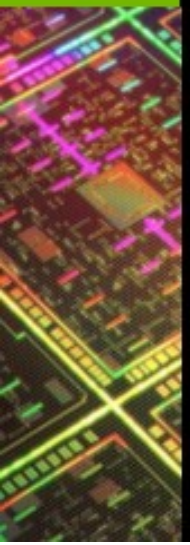


## Simplified View of a GPU



# Thrust Best Practices





## Best Practices

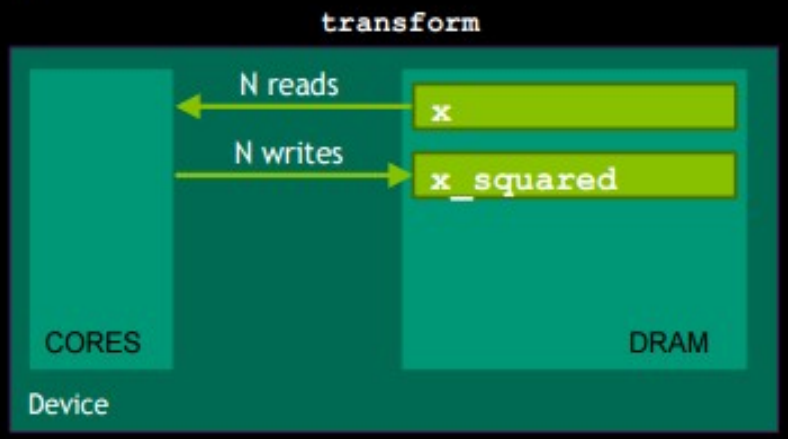
- In general
  - Many applications are limited by memory bandwidth
- Best Practices
  - Fusion
    - Combined related operations together
  - Structure of Arrays
    - Ensure memory coalescing
  - Implicit sequences
    - Eliminate memory accesses and storage

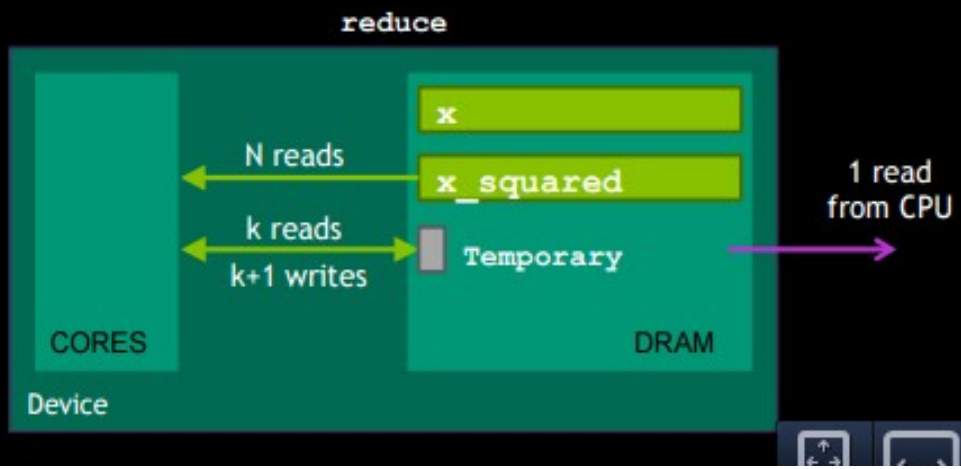
# Thrust Best Practices



## Fusion: Sum of squares $\sum x_i^2$

```
struct square { __device__ __host__ float operator()(float xi) { return xi*xi; } };

float sum_of_squares(const thrust::device_vector<float> &x)
{
  size_t N = x.size();
  thrust::device_vector<float> x_squared(N); // Temporary storage: N elements.

  // Compute x^2: N reads + N writes.
  thrust::transform(x.begin(), x.end(), x_squared.begin(), square());

  // Compute the sum of x^2s: N + k reads + k+1 writes (k is a small constant).
  return thrust::reduce(x_squared.begin(), x_squared.end());
}
```



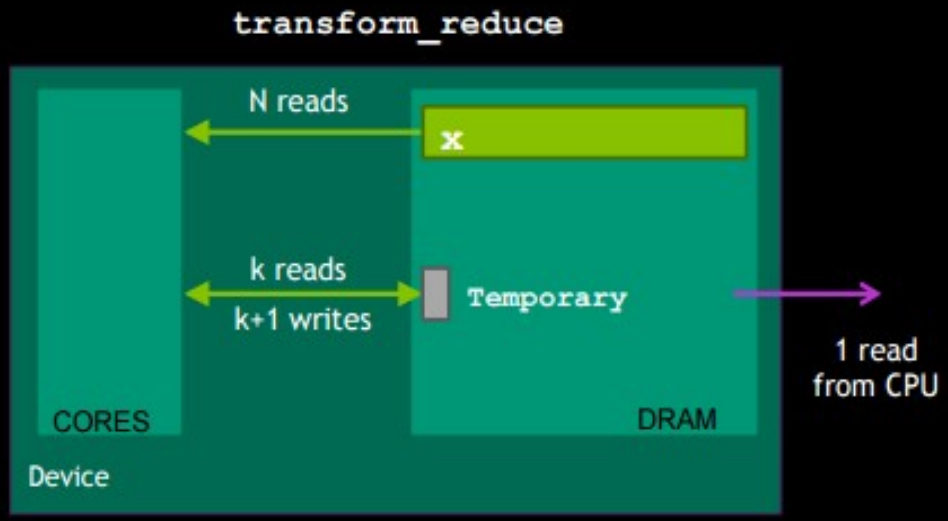
# Thrust Best Practices



## Fusion

- Combined related operations together

```
float fused_sum_of_squares(const thrust::device_vector<float> &x)
{
  // Compute the x^2s and their sum: N + k reads + k+1 writes (k is a small constant).
  return thrust::reduce(
    thrust::make_transform_iterator(x.begin(), square()),
    thrust::make_transform_iterator(x.end(),   square()));
}
```




We save:

- N temporary storage (`x_squared`)
- N writes (to `x_squared`)
- N reads (from `x_squared`)

# Fancy Iterators

## thrust

Main Page | Modules | Namespaces | Classes | Files

### Fancy Iterators

Iterators

### Classes

| | |
|---|---|
| class | thrust::constant_iterator< Value, Incrementable, System > |
| class | thrust::counting_iterator< Incrementable, System, Traversal, Difference > |
| class | thrust::discard_iterator< System > |
| class | thrust::iterator_adaptor< Derived, Base, Value, System, Traversal, Reference, Difference > |
| class | thrust::iterator_facade< Derived, Value, System, Traversal, Reference, Difference > |
| class | thrust::iterator_core_access |
| class | thrust::permutation_iterator< ElementIterator, IndexIterator > |
| class | thrust::reverse_iterator< BidirectionalIterator > |
| class | thrust::transform_iterator< AdaptableUnaryFunction, Iterator, Reference, Value > |
| class | thrust::zip_iterator< IteratorTuple > |

# Fancy **Iterators**

- constant_iterator
- counting_iterator
- transform_iterator
- permutation_iterator
- zip_iterator

# Fancy Iterators

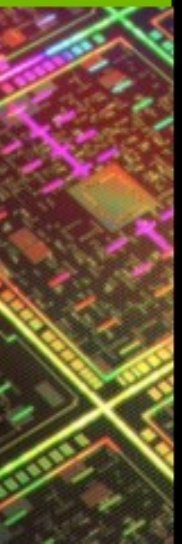
## transform_iterator

- Representa un puntero a un rango de valores después de una transformación por una función.
- Util para crear un rango lleno con el resultado de aplicar una operacion sobre otro rango sin guardarlo explícitamente en memoria ni ejecutando la transformación...
- Facilita la “fusión de kernels”, pateando la ejecución de la transformación hasta que el valor transformado es necesitado, minimizando asi memoria y ancho de banda.

```
...
int main(void)
{
  thrust::device_vector<float> v(4);
  v[0] = 1.0f; v[1] = 4.0f; v[2] = 9.0f; v[3] = 16.0f;

  typedef thrust::device_vector<float>::iterator FloatIterator;

  thrust::transform_iterator<square_root, FloatIterator> iter(v.begin(), square_root());

  *iter;   // returns 1.0f
  iter[0]; // returns 1.0f;
  iter[1]; // returns 2.0f;
  iter[2]; // returns 3.0f;
  iter[3]; // returns 4.0f;

  // iter[4] is an out-of-bounds error
}
```

# Thrust Best Practices





## Implicit Sequences

- Often we need ranges following a sequential pattern
  - Constant ranges
    - `[1, 1, 1, 1, ...]`
  - Incrementing ranges
    - `[0, 1, 2, 3, ...]`
- Implicit ranges require no storage
  - `thrust::constant_iterator`
  - `thrust::counting_iterator`

# x={0,1,2,...} → f(x)={f(0),f(1),...}

```
...

int main(int arch, char **argv)
{
   srand(13); int N=10000000;

   thrust::device_vector<float> x(N); // abscisa
   thrust::device_vector<float> y(N); // coordenada
   thrust::sequence(x.begin(),x.end()); // x={0,1,2,3,...N-1}

   float a=1.0; mifuncion op(a);

   thrust::transform(x.begin(),x.end(), y.begin(),op);
   std::cout << y[2] << " vs " << sin(a*2.0) << std::endl;
}
```

*ineficiente*

# Fancy Iterators

## counting_iterator

- Representa un puntero a un rando de valores constantes.
- Util para crear una secuencia constante sin usar memoria.
- Si uso economiza memoria y ancho de banda.

```
...
int main(int arch, char **argv)
{
    int N=10000000;
    thrust::device_vector<float> y(N); // coordenada

    float a=1.0; mifuncion op(a);

    thrust::transform(thrust::make_counting_iterator(0),
                      thrust::make_counting_iterator(N),
                      y.begin(),op);

}
```

*eficiente*

# Fancy Iterators

## counting_iterator

- Representa un puntero a un rando de valores constantes.
- Util para crear una secuencia constante sin usar memoria.
- Si uso economiza memoria y ancho de banda.

```
...
int main(int arch, char **argv)
{
        int N=10000000;
        thrust::device_vector<float> y(N); // coordenada
        float a=1.0; mifuncion op(a);

        thrust::counting_iterator<int> first(0);
        thrust::counting_iterator<int> last(N);
        thrust::transform(first,last,y.begin(),op);
}
```

*eficiente*

# Fancy Iterators

## constant_iterator

- Representa un puntero a un rango de valores constantes.
- Útil para crear una secuencia constante sin usar memoria.
- Su uso economiza memoria y ancho de banda.

```
  include <thrust/iterator/constant_iterator.h>

...
// create iterators
thrust::constant_iterator<int> first(10);
thrust::constant_iterator<int> last = first + 3;
first[0]   // returns 10
first[1]   // returns 10
first[100] // returns 10
// sum of [first, last)
thrust::reduce(first, last);   // returns 30 (i.e. 3 * 10)
```

# Fancy Iterators

## constant_iterator

- Representa un puntero a un rango de valores constantes.
- Útil para crear una secuencia constante sin usar memoria.
- Su uso economiza memoria y ancho de banda.

```
...
int main(void)
{
    thrust::device_vector<int> data(4);
    data[0] = 3; data[1] = 7;
    data[2] = 2; data[3] = 5;

    // add 10 to all values in data
    thrust::transform(data.begin(), data.end(),thrust::constant_iterator<int>(10),
                      data.begin(), thrust::plus<int>());
    // data is now [13, 17, 12, 15]

    // print result
    thrust::copy(data.begin(), data.end(),
             std::ostream_iterator<int>(std::cout, "\n"));

    return 0;
}
```

# Permutaciones

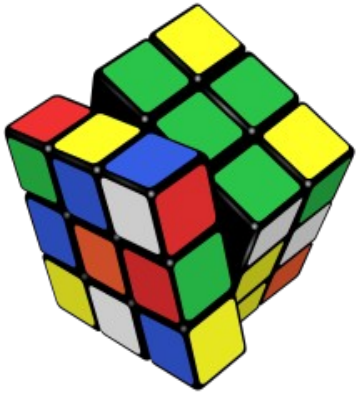


Imagen “encriptada”

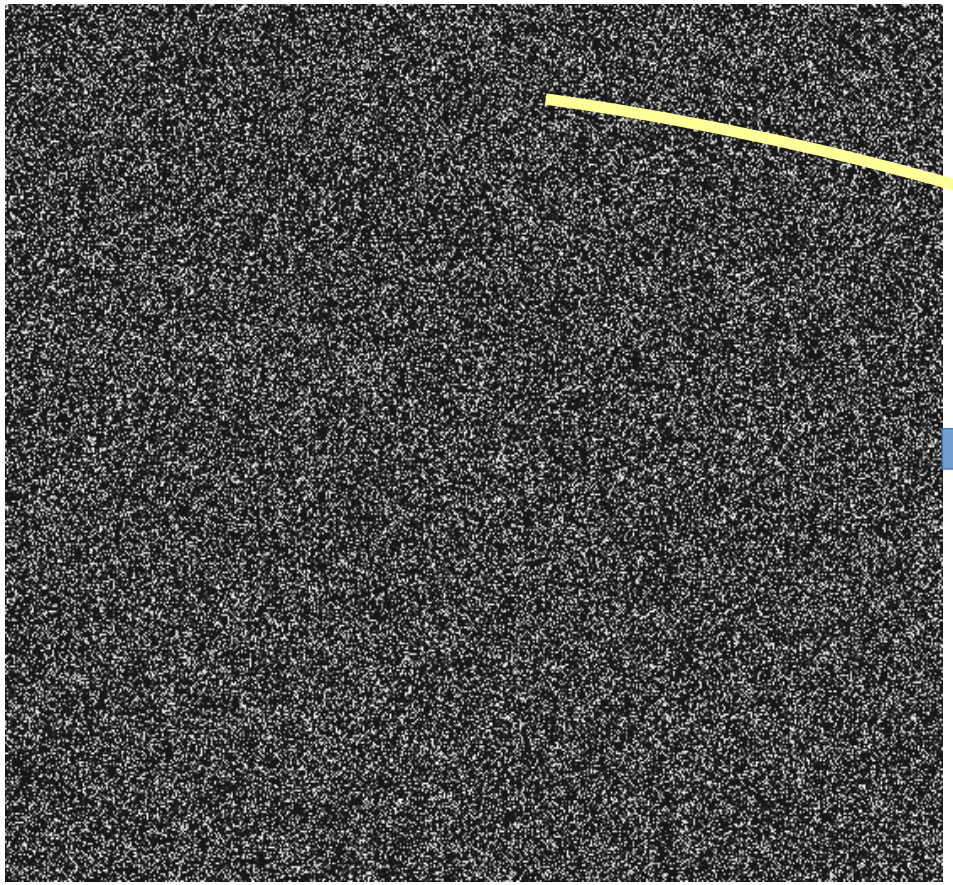
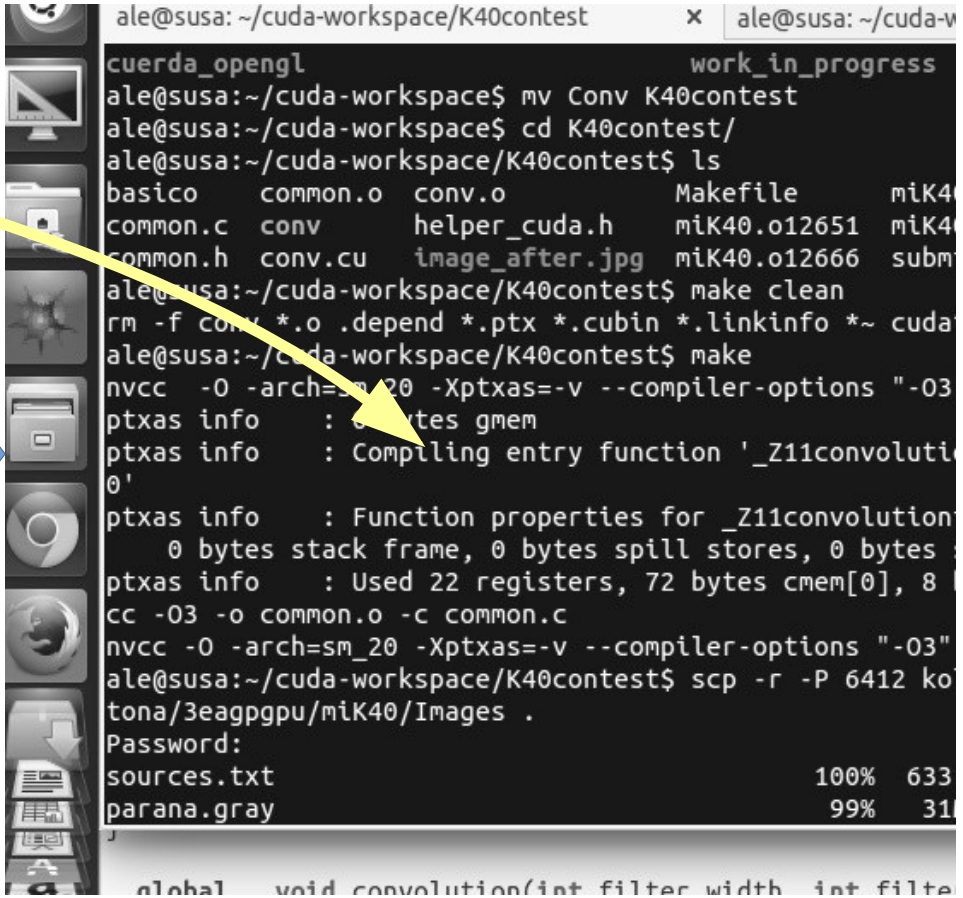


Imagen “desencriptada”




Los pixels están mezclados usando un mapeo dado...

[FINAL ICNPG2014]

# Fancy Iterators

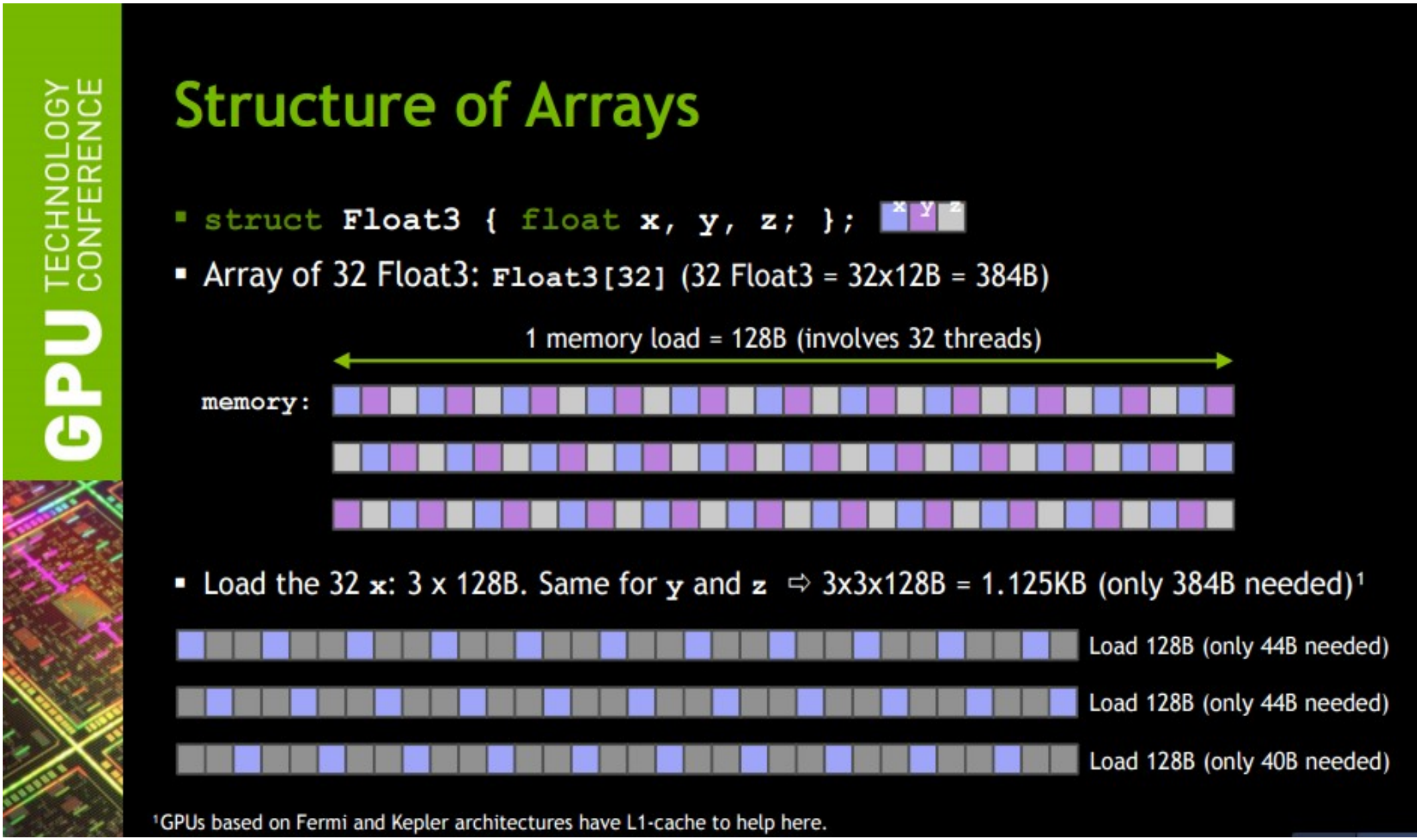
## permutation_iterator

- Representa un puntero a una vision reordenada de un rango.
- Util para fusionar las operaciones de “scatter” y “gather” con otros algoritmos.

```
int main(void)
{
    // gather locations
    thrust::device_vector<int> map(4);
    map[0] = 3; map[1] = 1; map[2] = 0; map[3] = 5;

    // array to gather from
    thrust::device_vector<int> source(6);
    source[0] = 10;
    source[1] = 20;
    source[2] = 30;
    source[3] = 40;
    source[4] = 50;
    source[5] = 60;

    // fuse gather with reduction:
    //   sum = source[map[0]] + source[map[1]] + ...
    int sum = thrust::reduce(thrust::make_permutation_iterator(source.begin(), map.begin()),
                             thrust::make_permutation_iterator(source.begin(), map.end()));
    return 0;
}
```

```
sum = source[map[0]] +
    source[map[1]]+ ...
```

# Thrust Best Practices



# Thrust Best Practices



## Structure of Arrays

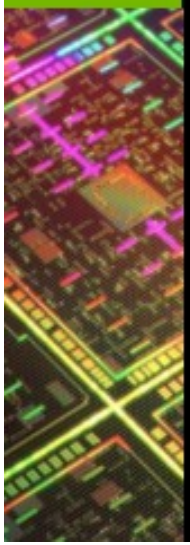
- Group xs, ys and zs

```
struct StructOfFloats
{
  thrust::device_vector<float> x;
  thrust::device_vector<float> y;
  thrust::device_vector<float> z;
};
```

- Load x: 1 x 128B. Same for y and z ⇨ 3x128B = 384B (all needed)

# Thrust Best Practices





## Structure of Arrays

- Example: Scale a sequence of Float3

```
struct scale
{
  typedef thrust::tuple<float, float, float> Float3;
  float s;
  scale(float s) : s(s) {}
  __host__ __device__ Float3 operator()(Float3 t)
  {
    float x = thrust::get<0>( t );
    float y = thrust::get<1>( t );
    float z = thrust::get<2>( t );
    return thrust::make_tuple( s*x, s*y, s*z );
  }
};

thrust::transform(
  thrust::make_zip_iterator(thrust::make_tuple(x.begin(), y.begin(), z.begin())),
  thrust::make_zip_iterator(thrust::make_tuple(x.end(),   y.end(),   z.end())),
  thrust::make_zip_iterator(thrust::make_tuple(x.begin(), y.begin(), z.begin())),
  scale( 2.0f ));
```

# Hands On

/share/apps/icnpg/clases/Clases_Thrust/clase_B/SoAvsAoS

*Array de estructuras (AoS)*

```
thrust::device_vector<MyStruct> structures(N);
thrust::sort(structures.begin(), structures.end());
```

Necesito comparador custom

```
struct MyStruct
{
  int key;
  float value;

  __host__ __device__
    bool operator<(const MyStruct other) const
    {
      return key < other.key;
    }
};
```

**vs**

*Estructuras de arrays (SoA)*

```
thrust::device_vector<int>   keys(N);
thrust::device_vector<float> values(N);
thrust::sort_by_key(keys.begin(), keys.end(), values.begin());
```

Comparador default de int

# Ejemplo de Kernel fusion

$$\langle x \rangle = \sum_i x_i / N$$

$$\langle (x - \langle x \rangle)^2 \rangle = \sum_i (x_i - \langle x \rangle)^2 / N$$

- Calculo la media y la varianza lanzando dos kernel reduce
- ¿Las puedo calcular en uno solo kernel reduce?

```
thrust::transform_reduce
(
    d_vec.begin(), d_vec.end(),
    [=] __device__ (double x)
    {
        return thrust::make_tuple(x,x*x);
    },
    thrust::make_tuple(0.,0.),
    [=] __device__ (auto b, auto a){
        return
        thrust::make_tuple(
          thrust::get<0>(a)+thrust::get<0>(b),
          thrust::get<1>(a)+thrust::get<1>(b)
        );
    }
);
```

- *Entra d_vec, sale la media y la varianza*
- *La tupla me permite agrupar variables distintas*
- *Defino transformacion sobre la tupla (identidad o elevar al cuadrado)*
- *Defino operacion de reduccion sobre la tupla (suma de sus respectivos elementos)*

# Ejercicio

$$S = \frac{\sum_{i=0}^{n}(x_i + y_i)}{\sqrt{\sum_{i=0}^{n}(x_i + y_i)^2}}$$

# Aplicación: Modelo de Epidemias

**Tasa de contagio**

$$\frac{dS}{dt} = -\beta SI,$$

$$\frac{dI}{dt} = \beta SI - \gamma I,$$

$$\frac{dR}{dt} = \gamma I.$$

*Método de Euler*

$$S(t+\Delta t) = S(t) + \Delta t[-\beta I(t)S(t)]$$

$$I(t+\Delta t) = I(t) + \Delta t[\beta I(t)S(t) - \gamma I(t)]$$

$$R(t+\Delta t) = R(t) + \Delta t[\gamma I(t)]$$

**S**uceptible, **I**nfectado, **R**ecuperado