ICHEC
Irish Centre for High-End Computing

Modern GPU Programming With CUDA and Thrust

Gilles Civario (ICHEC)
gilles.civario@ichec.ie

An Roinn Post, Fiontar agus Nuálaíochta
Department of Jobs, Enterprise and Innovation

sfi
science foundation ireland
fondúireacht eolaíochta éireann

AN ROINN
OIDEACHAIS AGUS SCILEANNA
DEPARTMENT OF
EDUCATION AND SKILLS

HEA
Higher Education Authority
An tÚdarás um Ard-Oideachas

Let me introduce myself

ID: Gilles Civario
Gender: Male
Age: 40
Affiliation: ICHEC
Specialty: HPC
Mission: CUDA teaching

And ICHEC…

- Irish Centre for High-End Computing
  - Based in Ireland
  - Computing centre
  - Specialised in HPC

Plan

- Quick overview and rational for GPU computing
- Example of "legacy" CUDA code
- Thrust in a nutshell
- Hands on session
- Conclusion

What is GPU computing?

- What about you tell me?

- A few milestones
  - GPGPU (2002 – 2005): pioneers using DirectX or OpenGL
  - CUDA 1.0 (2006) , now version 5.0 (or 5.5)
  - OpenCL 1.0 (2009), now version 1.2
  - OpenACC 1.0 (2011), now version 2.0 in draft
  - OpenMP 4.0 (expected)

Why GPU computing?

- To make it fast!
- To make it faster!
- To make it even faster…

- Because it's fun
- To solve today your tomorrow's problems

  (which interestingly enough, gives you double task to deal with today's problems too…)

Is that a silver bullet?

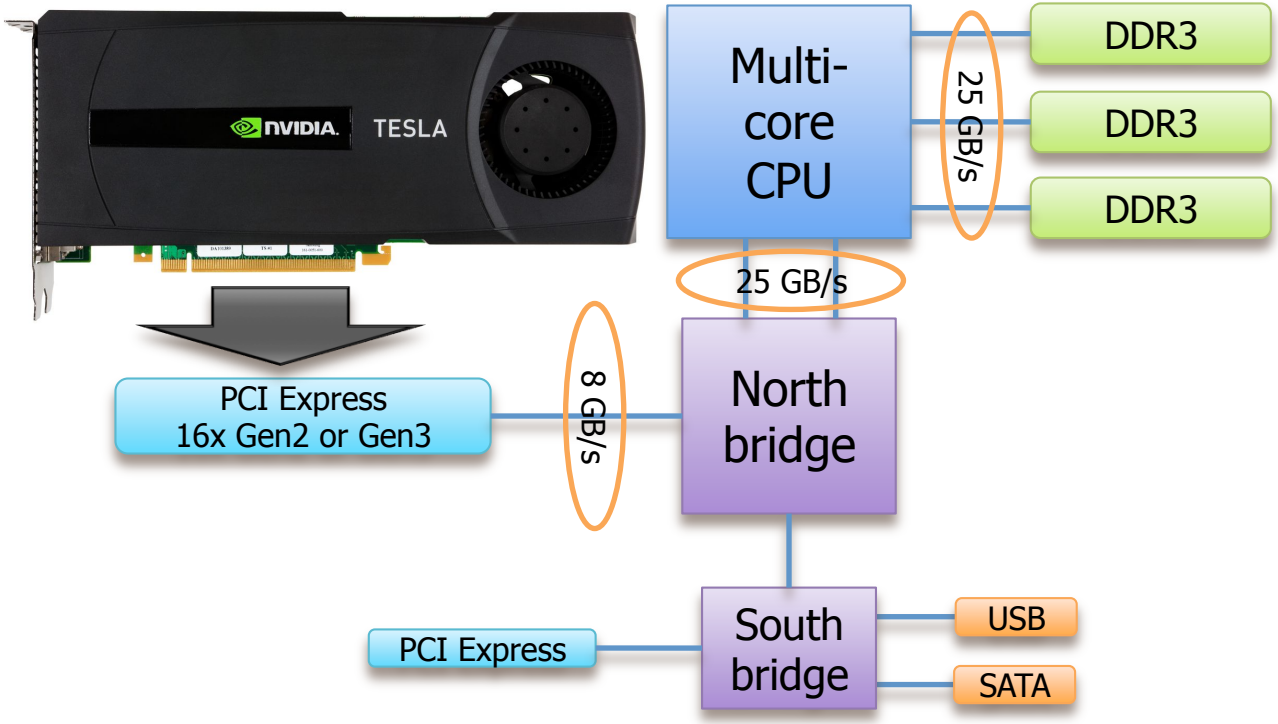
- YES: it is the answer to the *Ultimate question of Life, The Universe and Everything*

  Oops, no, that's

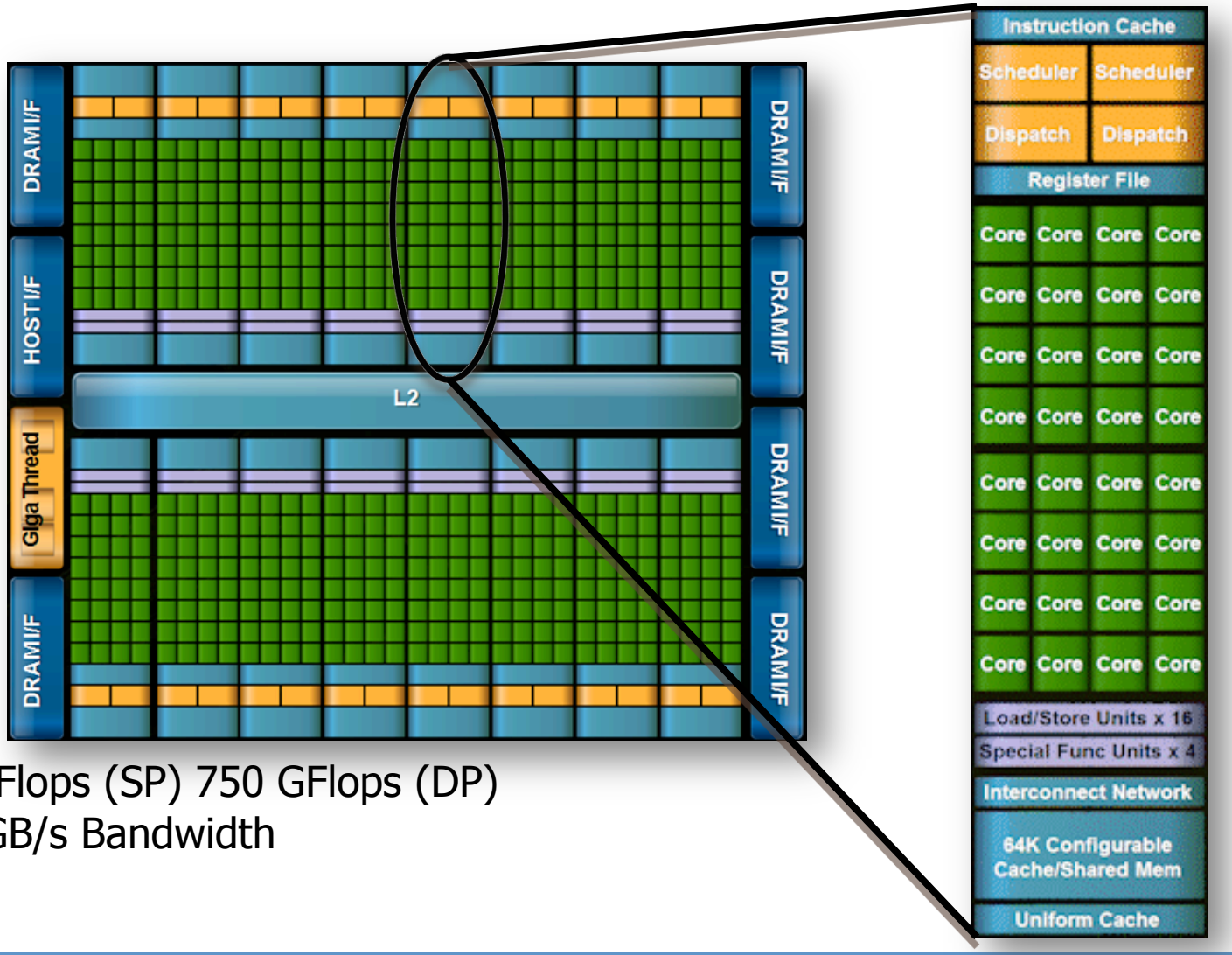
  sorry. But don't panic…

- It needs a few things to work properly:
  - Parallelism: ideally massive parallelism
  - A GPU card: from NVIDIA for **CUDA** obviously
  - Ideally to be able to avoid moving data back and forth between CPU and GPU

What does the HW look like?



But what's inside?

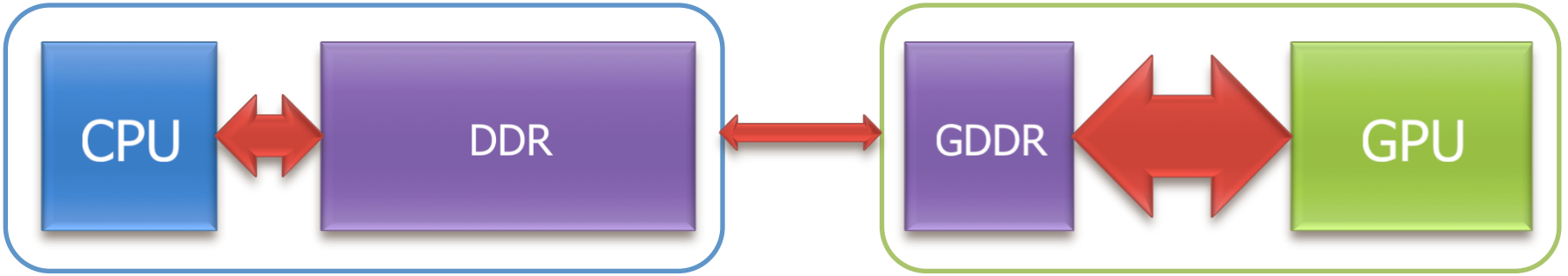


Fermi

- 1.5 TFlops (SP) 750 GFlops (DP)
- 190 GB/s Bandwidth

Could we make it simpler?

- Let's recap:
  - CPU is the "host"
  - It has its own memory
  - GPU is the "device"
  - It has its own private embedded memory
  - Host and device are connected through PCIe bus



But what's the programming model?

- The host manages everything
  - Memory allocation
  - Data transfers
  - Launching "kernels" on the GPU
- The expected speed-up comes from
  - Massive parallelisation over thousands of threads
  - Huge memory bandwidth on the device
- But is limited by Amdahl's law
  - If p is the accelerated portion of the code, speed-up upper limit is?

$$\frac{1}{1 - p}$$

OK, how do I code for that?

- That' s where CUDA comes in
  - Memory management: *cudaMalloc*, *cudaFree*, *cudaMemcpy*, *cudaMemset*, …
  - Various device memory definitions: *__device__*, *__shared__*, *texture*
  - A few function qualifiers: *__host__*, *__device__*, *__global__*
  - Organisation by threads and thread blocks on the device: *threadIdx.x*, *blockIdx.y*, *blockDim.z*, …
  - A way of running on the device: *kernel<<<…>>>()*

A CUDA "Hello world!"

```
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

__global__ void adder(float *z, const float *x, const float *y) {
    int tid = blockIdx.x * blockDim.x + threadIdx.x;
    z[tid] = x[tid] + y[tid];
}

int main() {
    float *x = (float*) malloc(SIZE*sizeof(float));
    float *y = (float*) malloc(SIZE*sizeof(float));
    float *z = (float*) malloc(SIZE*sizeof(float));
    float *d_x, *d_y, *d_z;
    int i;

    for (i=0; i<SIZE; i++) {
        x[i] = i;
        y[i] = .1f * i;
    }
```

A CUDA "Hello world!"

```
cudaMalloc(&d_x, SIZE*sizeof(float));
cudaMalloc(&d_y, SIZE*sizeof(float));
cudaMalloc(&d_z, SIZE*sizeof(float));

cudaMemcpy(d_x, x, SIZE*sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(d_y, y, SIZE*sizeof(float), cudaMemcpyHostToDevice);

adder<<<1, SIZE>>>(d_z, d_x, d_y);

cudaMemcpy(z, d_z, SIZE*sizeof(float), cudaMemcpyDeviceToHost);

cudaFree(d_x); cudaFree(d_y); cudaFree(d_z);
```

A CUDA "Hello world!"

```
    for (i=0; i<SIZE; i++)
        printf("%5.3f ", z[i]);
    printf("\n");

    free(x); free(y); free(z);

    return 0;
}
```

```
gcivario@chartreuse: ~/workdir/TRAINING/ISS13/ex
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$ nvcc -o hello hello.cu
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$ ./hello
0.000 1.100 2.200 3.300 4.400 5.500 6.600 7.700 8.800 9.900
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$
```

Let's step back...

- This example is based on a C-like code

- This exam

```c
#include <stdio.h>
#include <stdlib.h>

#define SIZE 10

void adder(float* z, float *x, float *y, int n) {
    int i;
    for (i=0; i<n; i++)
        z[i] = x[i] + y[i];
}

int main() {
    float *x = (float*) malloc(SIZE*sizeof(float));
    float *y = (float*) malloc(SIZE*sizeof(float));
    float *z = (float*) malloc(SIZE*sizeof(float));
    int i;

    for (i=0; i<SIZE; i++) {
        x[i] = i;
        y[i] = .1f * i;
    }

    adder(z, x, y, SIZE);

    for (i=0; i<SIZE; i++)
        printf("%5.3f ", z[i]);
    printf("\n");

    free(x); free(y); free(z);

    return 0;
}
```

Let's step back…

- This example is based on a C-like code
  - Not a particularly clever one…
- But how could we rewrite it in a C++ STL based style?

```cpp
#include <iostream>
#include <iterator>
#include <vector>
#include <algorithm>
#include <functional>

int main() {
    const int SIZE=10;
    std::vector<float> x(SIZE), y(SIZE), z(SIZE);

    for (int i=0; i<x.size(); i++) {
        x[i] = i;
        y[i] = .1f * i;
    }

    std::transform(x.begin(), x.end(), y.begin(), z.begin(), std::plus<float>());

    std::copy(z.begin(), z.end(), std::ostream_iterator<float>(std::cout, " "));
    std::cout << std::endl;

    return 0;
}
```

Could we do the same for CUDA?

- Well, actually we can!
  - The answer is called Thrust
  - It comes with CUDA
  - It mimics the STL philosophy and constructs
  - It hides all the complexity of memory management
  - It provides effective algorithms for the most common operations

Let's revisit our example

```
#include <iostream>
#include <iterator>
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sequence.h>
#include <thrust/transform.h>
#include <thrust/functional.h>
#include <algorithm>

int main() {
    const int SIZE=10;
    thrust::host_vector<float> x(SIZE), y(SIZE);

    thrust::sequence(x.begin(), x.end());
    thrust::sequence(y.begin(), y.end(), 0.f, .1f);
```

Let's revisit our example

```
    thrust::device_vector<float> d_x = x, d_y = y, d_z(SIZE);

    thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_z.begin(),
                      thrust::plus<float>());

    thrust::host_vector<float> z = d_z;

    std::copy(z.begin(), z.end(), std::ostream_iterator<float>(std::cout, " "));
    std::cout << std::endl;

    return 0;
}
```

```
gcivario@chartreuse: ~/workdir/TRAINING/ISS13/examples
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$ nvcc -o hello_thrust hello_thrust.cu
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$ ./hello_thrust
0 1.1 2.2 3.3 4.4 5.5 6.6 7.7 8.8 9.9
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$
```

What is thrust good at?

- Hiding the complexity
- Improving your productivity
- Keeping code readable and maintainable
- Incrementally porting parts of your applications
  - Allows keeping a working version
  - Slowly shifting the hot spots

But is sufficient?

- It depends
- Thrust comes with pre-defined algorithms
  - If your problem match them, it might be enough
  - If not, you might need more developments
  - But some other libraries exist
    - cuBLAS, cuFFT, cuSPARSE, CURAND
    - CUB: CUDA UnBound

Ultimately, you decide when faster is fast enough

Philosophy

- Same as the C++ STL
- Based on iterators applying on containers
- Provides the containers and some main algorithms
  - Building block for more advanced processing
- Allows applying user-defined functions through "functors"
  - Typically a class implementing a function through the *operator()* method

Example: saxpy

- Computes $Y \leftarrow A*X + Y$
- First method: using temporary storage

```
void saxpy(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y) {
    thrust::device_vector<float> temp(X.size());

    // temp <- A
    thrust::fill(temp.begin(), temp.end(), A);

    // temp <- A * X
    thrust::transform(X.begin(), X.end(),
                      temp.begin(), temp.begin(),
                      thrust::multiplies<float>());

    // Y <- A * X + Y
    thrust::transform(temp.begin(), temp.end(),
                      Y.begin(), Y.begin(),
                      thrust::plus<float>());
}
```

Example: saxpy

- Second method: using a functor

```
struct saxpy_functor {
    const float a;

    saxpy_functor(float _a) : a(_a) {}

    __host__ __device__ float operator()(const float& x, const float& y) const {
        return a * x + y;
    }
};

void saxpy(float A, thrust::device_vector<float>& X, thrust::device_vector<float>& Y) {

    thrust::transform(X.begin(), X.end(), Y.begin(), Y.begin(), saxpy_functor(A));

}
```

Performance remarks

- Both methods are equally valid, but
  - Saxpy is memory bound, not compute bound
    - So the number of memory transactions drives its performance
  - The first one needs 4N loads and 3N stores
    - N stores for fill()
    - 2N loads and N stores per transform()
  - The second one needs 2N loads and N stores
    - Only one call to transform()

➔ The second version should be ~2.3x faster (on GPU)

Performance comparison

```
int main() {
    const int N=1000000;
    thrust::host_vector<float> tmp(N);
    thrust::generate(tmp.begin(), tmp.end(), myrand);

    thrust::device_vector<float> X=tmp, Y=tmp;
    float etime1, etime2;

    // warming up
    TIME_GPU( saxpy1(10.f, X, Y), etime1, 10);
    // actual runs
    TIME_GPU( saxpy1(10.f, X, Y), etime1, 10);
    TIME_GPU( saxpy2(10.f, X, Y), etime2, 10);
```

```
gcivario@chartreuse: ~/workdir/TRAINING/ISS13/examples
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$ nvcc -o saxpy saxpy.cu
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$ ./saxpy
Time for method 1 is 2.96682 ms, and method 2 is 1.07359 ms for vectors of 1000000 elements.
Speed-up is 2.76345 for method 2 vs. method 1.
gcivario@chartreuse:~/workdir/TRAINING/ISS13/examples$
```

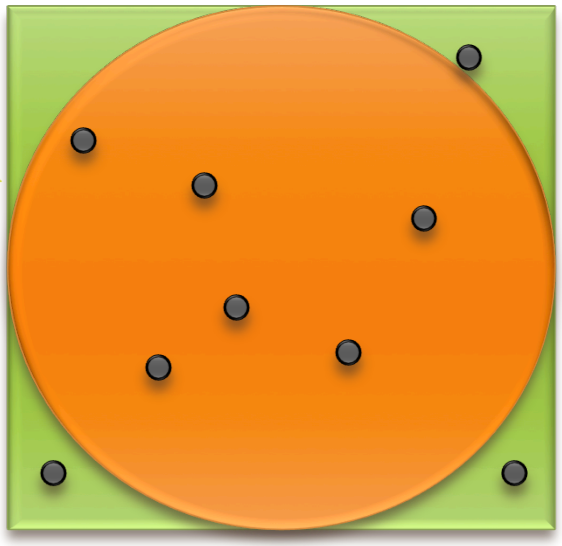
Thrust main features

- Memory management:
  - *host_vector* and *device_vector*
  - *copy()*, *sequence()*, *fill()*
- Algorithms:
  - *transform()*, *reduce()*, *transform_reduce()*
  - *inclusive_scan()* and *exclusive_scan()*
  - *gather()* and *scatter()*
  - *sort()*, *merge()*
  - Algebra of sets
  - Iterators…

Let's get our hands dirty

- Goal: computing π
  - With the most ineffective method ever, based on Monte-Carlo method

    And on a blind gunner



$$\lim_{try \to \infty} \frac{hit}{try} = \frac{\pi}{4}$$

What you've got to do

- The lab is split into steps
  - Progress one step at a time
  - Read instructions into the readme.txt files
  - Modify or adapt the source codes
- To compile
  - No makefile… That'd be too easy
    - C++: g++ -o *bin file.cc*
    - CUDA: nvcc -arch sm_20 -o *bin file.cu*
- May the force be with you young padawans

Doc: sequence

```
template<typename ForwardIterator >
void thrust::sequence ( ForwardIterator first,
                        ForwardIterator last
                      )
```

sequence fills the range `[first, last)` with a sequence of numbers.

For each iterator `i` in the range `[first, last)`, this version of sequence performs the assignment `*i = (i - first)`.

Parameters:

first	The beginning of the sequence.
last	The end of the sequence.

Template Parameters:

ForwardIterator	is a model of Forward Iterator, and `ForwardIterator` is mutable, and if `x` and `y` are objects of `ForwardIterator`'s `value_type`, then `x + y` is defined, and if `T` is `ForwardIterator`'s `value_type`, then `T(0)` is defined.

Doc: transform

```
template<typename InputIterator , typename OutputIterator , typename UnaryFunction >
OutputIterator thrust::transform ( InputIterator  first,
                                   InputIterator  last,
                                   OutputIterator result,
                                   UnaryFunction  op
                                 )
```

This version of `transform` applies a unary function to each element of an input sequence and stores the result in the corresponding position in an output sequence. Specifically, for each iterator `i` in the range `[first, last)` the operation `op(*i)` is performed and the result is assigned to `*o`, where `o` is the corresponding output iterator in the range `[result, result + (last - first) )`. The input and output sequences may coincide, resulting in an in-place transformation.

Parameters:

first	The beginning of the input sequence.
last	The end of the input sequence.
result	The beginning of the output sequence.
op	The tranformation operation.

Returns:

The end of the output sequence.

Template Parameters:

InputIterator	is a model of Input Iterator and `InputIterator's value_type` is convertible to `UnaryFunction's argument_type`.
OutputIterator	is a model of Output Iterator.
UnaryFunction	is a model of Unary Function and `UnaryFunction's result_type` is convertible to `OutputIterator's value_type`.

Doc: reduce

```
template<typename InputIterator >
thrust::iterator_traits<InputIterator>::value_type thrust::reduce ( InputIterator first,
                                                                    InputIterator last
                                                                  )
```

reduce is a generalization of summation: it computes the sum (or some other binary operation) of all the elements in the range [first, last). This version of reduce uses 0 as the initial value of the reduction. reduce is similar to the C++ Standard Template Library's std::accumulate. The primary difference between the two functions is that std::accumulate guarantees the order of summation, while reduce requires associativity of the binary operation to parallelize the reduction.

Note that reduce also assumes that the binary reduction operator (in this case operator+) is commutative. If the reduction operator is not commutative then thrust::reduce should not be used. Instead, one could use inclusive_scan (which does not require commutativity) and select the last element of the output array.

Parameters:

first The beginning of the sequence.

last The end of the sequence.

Returns:

The result of the reduction.

Template Parameters:

InputIterator is a model of Input Iterator and if x and y are objects of InputIterator's value_type, then x + y is defined and is convertible to InputIterator's value_type. If T is InputIterator's value_type, then T(0) is defined.

Doc: counting_iterator

thrust::counting_iterator< Incrementable, System, Traversal, Difference > Class Template Reference

Fancy Iterators

```
#include <counting_iterator.h>
```

List of all members.

Detailed Description

template<typename Incrementable, typename System = use_default, typename Traversal = use_default, typename Difference = use_default>
class thrust::counting_iterator< Incrementable, System, Traversal, Difference >

`counting_iterator` is an iterator which represents a pointer into a range of sequentially changing values. This iterator is useful for creating a range filled with a sequence without explicitly storing it in memory. Using `counting_iterator` saves memory capacity and bandwidth.

Doc: transform_reduce

```
template<typename InputIterator , typename UnaryFunction , typename OutputType , typename BinaryFunction >
OutputType thrust::transform_reduce ( InputIterator    first,
                                      InputIterator    last,
                                      UnaryFunction    unary_op,
                                      OutputType       init,
                                      BinaryFunction   binary_op
                                    )
```

`transform_reduce` fuses the `transform` and `reduce` operations. `transform_reduce` is equivalent to performing a transformation defined by `unary_op` into a temporary sequence and then performing `reduce` on the transformed sequence. In most cases, fusing these two operations together is more efficient, since fewer memory reads and writes are required.

`transform_reduce` performs a reduction on the transformation of the sequence `[first, last)` according to `unary_op`. Specifically, `unary_op` is applied to each element of the sequence and then the result is reduced to a single value with `binary_op` using the initial value `init`. Note that the transformation `unary_op` is not applied to the initial value `init`. The order of reduction is not specified, so `binary_op` must be both commutative and associative.

Parameters:

first	The beginning of the sequence.
last	The end of the sequence.
unary_op	The function to apply to each element of the input sequence.
init	The result is initialized to this value.
binary_op	The reduction operation.

Returns:

The result of the transformed reduction.

Template Parameters:

InputIterator	is a model of Input Iterator, and `InputIterator`'s `value_type` is convertible to `UnaryFunction`'s `argument_type`.
UnaryFunction	is a model of Unary Function, and `UnaryFunction`'s `result_type` is convertible to `OutputType`.
OutputType	is a model of Assignable, and is convertible to `BinaryFunction`'s `first_argument_type` and `second_argument_type`.
BinaryFunction	is a model of Binary Function, and `BinaryFunction`'s `result_type` is convertible to `OutputType`.

Doc: curand_init

```
__device__ void
curand_init (
    unsigned long long seed, unsigned long long sequence,
    unsigned long long offset, curandState_t *state)
```

The `curand_init()` function sets up an initial state allocated by the caller using the given seed, sequence number, and offset within the sequence. Different seeds are guaranteed to produce different starting states and different sequences. The same seed always produces the same state and the same sequence. The state set up will be the state after $2^{67} \cdot$ `sequence` $+$ `offset` calls to `curand()` from the seed state.

Sequences generated with different seeds usually do not have statistically correlated values, but some choices of seeds may give statistically correlated sequences. Sequences generated with the same seed and different sequence numbers will not have statistically correlated values.

For the highest quality parallel pseudorandom number generation, each experiment should be assigned a unique seed. Within an experiment, each thread of computation should be assigned a unique sequence number. If an experiment spans multiple kernel launches, it is recommended that threads between kernel launches be given the same seed, and sequence numbers be assigned in a monotonically increasing way. If the same configuration of threads is launched, random state can be preserved in global memory between launches to avoid state setup time.

Doc: curand_uniform

```
__device__ float
curand_uniform (curandState_t *state)
```

This function returns a sequence of pseudorandom floats uniformly distributed between 0.0 and 1.0. It may return from 0.0 to 1.0, where 1.0 is included and 0.0 is excluded. Distribution functions may use any number of unsigned integer values from a basic generator. The number of values consumed is not guaranteed to be fixed.

But that's not all

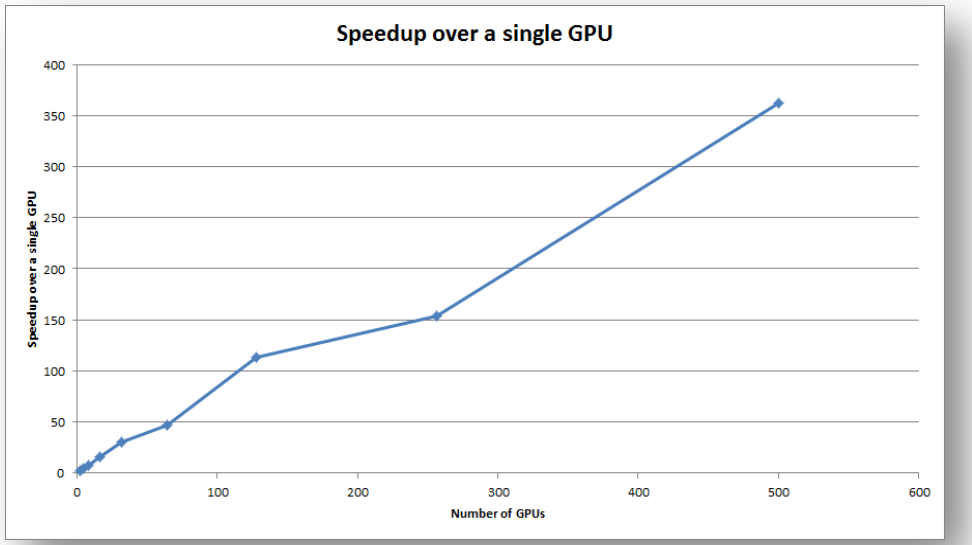
- Thrust is not only for CUDA
  - By default:
    - Host is a sequential CPU
    - Device is a CUDA-capable GPU
  - But you can adjust that with cpp macros
    - THRUST_HOST_BACKEND
    - THRUST_DEVICE_BACKEND
  - Supported targets
    - C++, CUDA, OpenMP, TBB

What more?

- Thrust comes with CUDA
  - No need to install anything more
- But it has its own development path
  - CUDA and Thrust are not always in sync
    - CUDA 5 → Thrust 1.5.3
    - CUDA 5.5rc → Thrust 1.7.0
  - You are welcome to install a newer version
    - New features and enhancements come regularly

And in real life?

- Well, it works
  - Straightforward if your code uses STL constructs
  - But not only
- It scales
  - It Peta-scales even
    (Graph courtesy R. Farber)



- It makes your life some much simpler

So…

- Use the Thrust, young padawans, use the Thrust