MI3.22
Advanced Programming for HPC
Master ICT, USTH, 2nd year

Aveneau Lilian

lilian.aveneau@univ-poitiers.fr
XLIM/SIC/IG, CNRS, Computer Science Department
University of Poitiers

Year 2019/2020

Lecture 1 – Thrust and first Parallel Patterns

- Thrust: high-level library
- First example
- Iterators
- Introduction to PRAM
- Parallel Patterns

Overview

- Thrust: high-level library
- First example
- Iterators
- Introduction to PRAM
- Parallel Patterns

Thrust: an introduction

Thrust is a high level parallel library:

- Resembles the STL (so it is a C++ library, with templates)
- Handles memory allocation and release using vectors
- Implements many parallel patterns:
    - MAP (known as transform in thrust)
    - GATHER
    - SCATTER
    - REDUCE
    - SEGMENTED REDUCE
    - SCAN
    - SEGMENTED SCAN
    - COMPACT (or compress)
    - PARTITION
    - SORT ...

Full documentation at: `http://thrust.github.io/doc/index.html`

Overview

First thrust example: addition of two vectors I

```
#include <iostream>

#include <thrust/device_vector.h>
#include <thrust/sequence.h>
#include <thrust/fill.h>

/// display a vector
void display(thrust::device_vector<float>& U, const std::string& name)
{
    // copy the data from GPU memory to CPU memory
    thrust::host_vector<float> V = U;
    // display each values
    for(int i=0; i<V.size(); ++i) {
        std::cout<< name << "[" << i << "]␣=␣" << V[i] << std::endl;
    }
}

/// function that adds two elements and return a new one
class AdditionFunctor : public thrust::binary_function<float,float,float> {
  public:
    __host__ __device__ float operator()(const float &x, const float &y) {
        return x + y;
    }
};
```

First thrust example: addition of two vectors II

```
/// creates two vectors on GPU, and do their addition
void doAddition(unsigned workSize)
{
    thrust::device_vector<float> result(workSize);
    thrust::device_vector<float> U(workSize);
    thrust::device_vector<float> V(workSize);

    /// initialization of two given arrays
    // fill U with 1, 2, 3 .... U.size()
    thrust::sequence(U.begin(), U.end(), 1.f);
    // fill V with 4, 4, .... 4
    thrust::fill(V.begin(), V.end(), 4.f);

    /// do a MAP (one-to-one operation)
    thrust::transform(U.begin(), U.end(),
                      V.begin(),
                      result.begin(),
                      AdditionFunctor());
    /// display the result (if not too big)
    if(workSize < 128) display(result, "result");
}

/// main function
int main(void)
{
    doAddition(10); // add "big" vectors (of size 10)
    return 0;
}
```

Overview

- Thrust: high-level library
- First example
- Iterators
- Introduction to PRAM
- Parallel Patterns

Iterator: a key for Thrust programming

Same meaning than in STL or other library:

- allows to traverse a given container

Properties

- Similar to Boost's iterators
- 5 kinds of iterators:

```
struct input_device_iterator_tag;         //   read access
struct output_device_iterator_tag;        // + write access
struct forward_device_iterator_tag;       // + increment
struct bidirectional_device_iterator_tag; // + decrement
struct random_access_device_iterator_tag; // + random access
```

- Some specialized Thrust's (random) iterators
    - constant
    - counting
    - fancy
    - zip
    - ...

Iterator: a key for thrust programming

Same program, excepting the `addition` function:

```
// creates two vectors on GPU, and do their addition
void doAdditionIterator(const unsigned workSize)
{
    thrust::device_vector<float> result(workSize);
    thrust::counting_iterator<float> U(1.f);
    thrust::constant_iterator<float> V(4.f);
    thrust::transform(U, U+workSize, V, result.begin(), AdditionFunctor());
    if(workSize < 128) display(result, "result");
}
```

We use far less memory, and so computations are done faster ;-)

Another example is the ZIP iterator: addition of three vectors:

```
// computes X + Y + Z, X, Y and Z being three vectors
thrust::transform(thrust::make_zip_iterator(thrust::make_tuple(X, Y, Z)),
                  thrust::make_zip_iterator(thrust::make_tuple(X+size, Y+size, Z+size)),
                  result.begin(),
                  AdditionFunctor3() );
```

where the functor is:

```
// function that adds three elements and returns a new one
typedef thrust::tuple<float,float,float> myFloat3;
class AdditionFunctor3 : public thrust::unary_function<myFloat3,float> {
  public:
    __device__ float operator()(const myFloat3& tuple) {
        return thrust::get<0>(tuple) + thrust::get<1>(tuple) + thrust::get<2>(tuple);
    }
};
```

Overview

Introduction

Why a theoretical model?

Sequential computation: TURING's machine

- Algorithm cost (Computability and Complexity)
- Polynomial problem or NP-complete

Parallelism:

- Communication cost? Simpler is to ignore it!
- Imperfect model, but useful to classify algorithms (their complexity)
- We will speak about maximal parallelism

PRAM model

Main memory is ∞, shared among all PE (∞ of them)

- CRCW: Concurrent Read Concurrent Write
  - Concurrent Writing: consistent, arbitrary or fusion modes
- CREW: Concurrent Read Exclusive Write
- EREW: Exclusive Read Exclusive Write

Overview

Goals

Writing a parallel algorithm is very complicated!

- Race condition, deadlock, ...
- Generally we start with a sequential solution

We need some well known parallel patterns:

- Pattern: software engineering solution to common problem
- Tools to write parallel algorithms efficiently

You should already know:

- MAP (or transform)
- GATHER and SCATTER
- REDUCE

The following is a catalog using PRAM and Thrust ...

A CUDA specific pattern: BLOCKING

Problems:

- DRAM accesses must be coalescent for efficiency!
- DRAM latency is high … avoid it for reading and writing!

BLOCKING solution:

- Logically partition data in well-sized blocks
  - Small enough to be staged into limited shared memory
- Use a sufficient number of blocks
- In each block, threads load data into shared memory
- Do the computation, using shared memory only
- Threads write the final result to DRAM

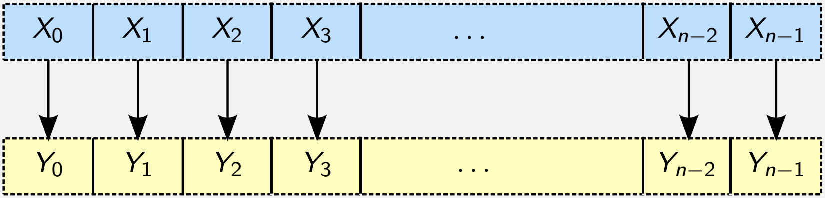
In each block, a thread is responsible to load/write a piece of memory in a coalescent way!

MAP

MAP is a one-to-one operation:

- For a given function f, an input array X of size n ...
- It computes the array Y of size n such that:

$$Y_i = f(X_i),\ \forall i \in [0, \ldots, n[$$



In Thrust, use the `thrust::transform` functions:

- Some for `thrust::unary_function`
- And some for `thrust::binary_function`

Exercise: Add vectors

Just try the previous examples (slides 4, 5 and 7), by doing it yourself ...

1. For two vectors containing data on the host (with at least 2^{16} values)
2. For two vectors of the same size, but using counting and constant iterators
3. For three vectors (on CPU) of the same size

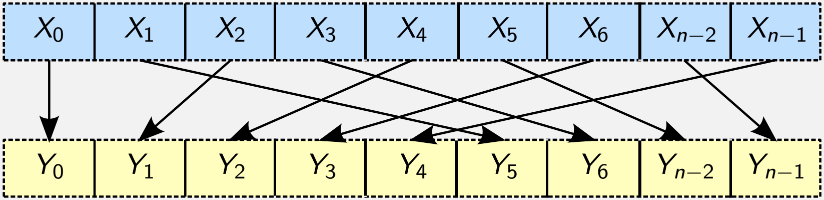
For each question, check its computation time (launch the calculation two or three times in a loop before to measure the time, since GPU takes some time to wake-up).

GATHER

Read data from arbitrary location to write contiguous values:

- Let X be an entry of size n, and map(i) a function returning a new index $\in [0 \ldots n[$
- The result is a set of contiguous values Y of size n such that:

$$Y_i = X_{\mathrm{map}(i)},\ \forall i \in [0, \ldots, n[$$



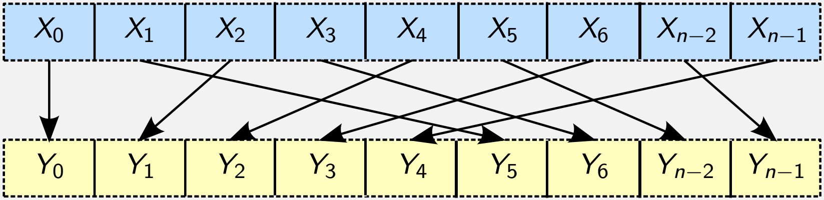
- In this figure: the map indicating from where output data come from is $\{0, 2, 4, 6, 8, 1, 3, 5, 7\}$
- In Thrust, it is implemented by functions `thrust::gather`
- Important to retain: the results are consecutively filled ...

SCATTER

Read consecutive data to write values in arbitrary locations:

- Let X be an entry of size n, and $map(i)$ a function returning an index into $[0, \dots, n[$
- The result is a set of values Y of size n such that:

$$Y_{\mathrm{map}(i)} = X_i,\ \forall i \in [0, \dots, n[$$



X_0 X_1 X_2 X_3 X_4 X_5 X_6 X_{n-2} X_{n-1}
Y_0 Y_1 Y_2 Y_3 Y_4 Y_5 Y_6 Y_{n-2} Y_{n-1}

- In this figure, the map is $\{0, 5, 1, 6, 2, 7, 3, 8, 4\}$
- In Thrust, it is implemented by functions `thrust::scatter`
- Difference with GATHER? Consecutive Read, not Write ...

MAP, GATHER & SCATTER using PRAM

MAP in PRAM

```
MAP($X$ , $f$ )
    { $X$ is an array of size N, $f$ a unary function }
    FOR each PE $i$ in parallel
        $Y_i \leftarrow f(X_i)$
    RETURN Y
```

GATHER in PRAM

```
GATHER($X$ , $Map$ )
    { $X$ and $Map$ are two arrays of size N }
    FOR each PE $i$ in parallel
        $Y_i \leftarrow X_{Map_i}$
    RETURN Y
```

SCATTER in PRAM

```
SCATTER($X$ , $Map$ )
    { $X$ and $Map$ are two arrays of size N }
    FOR each PE $i$ in parallel
        $Y_{Map_i} \leftarrow X_i$
    RETURN Y
```

Exercise: Odd-Even

Separate the numbers by odd and even index ...

1. Write two functions that takes as input a large vector of integers, and that separates and returns the same vector containing first the data at even indexes and then the ones at odd indexes. First function uses GATHER, second SCATTER.

   e.g.: $\{7, 5, 14, 10, 21, 15, 28\}$ becomes $\{7, 14, 21, 28, 5, 10, 15\}$.

   Hint: `transform_iterator` and `counting_iterator` may help you.
2. Do the same but for more heavy objects (records containing some useless data).

For each exercise, check its computation time (launch the calculation two or three times in a loop before to measure the time, since GPU takes some time to wake-up).