Boost.Compute

A parallel computing library for C++ based on OpenCL

Jakub Szuppe

What is Boost.Compute?

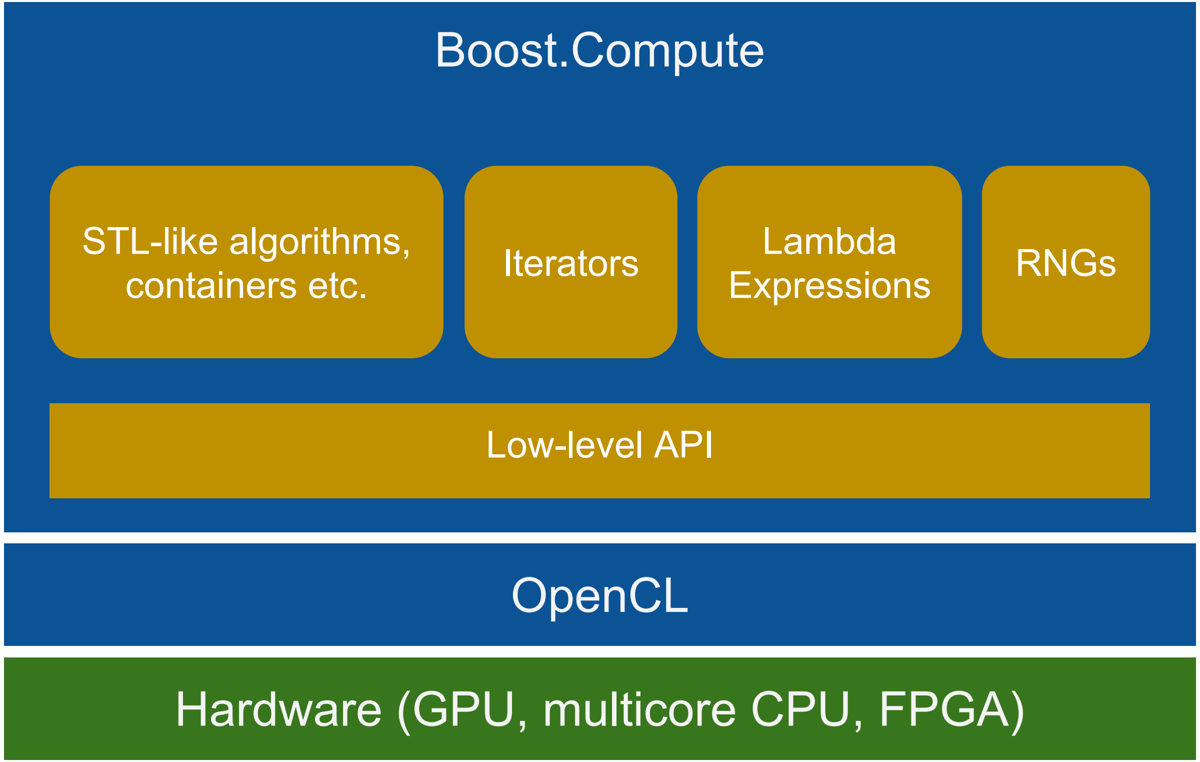
- C++ template library for parallel computing based on OpenCL
- Created by Kyle Lutz

- Accepted as an official Boost library in January 2015
- Available in Boost starting with version 1.61 (April 2016)

- https://github.com/boostorg/compute

Design

Library architecture


Boost.Compute
STL-like algorithms, containers etc.
Iterators
Lambda Expressions
RNGs
Low-level API
OpenCL
Hardware (GPU, multicore CPU, FPGA)

Library architecture

- Layered architecture
- High-level API is designed to resemble the Standard Template Library
  - Familiar API
  - Easy to use
  - Simplifies porting existing algorithms (std:: → boost::compute::)
- Low-level API provides C++ wrapper over the OpenCL API

Library architecture

- Library-based solution (header-only)
- OpenCL
  - Vendor-neutral
- Standard C++
  - No special compiler is required
  - No compiler extension is required
  - C++11 is not required

Low-level API

How Boost.Compute merges OpenCL into C++

OpenCL C++ wrapper

- Provides classes for OpenCL objects - `buffer`, `command_queue` etc.
- Takes care of:
  - Reference counting
  - Error checking
- Various utility functions that help integrate OpenCL into C++
  - Default device, context, command queue
  - Online and offline program caching

Low-level API example #1

```cpp
#include <boost/compute/core.hpp>

// let’s find device
auto device = boost::compute::system::find_device("my-device-name"s);

// create new OpenCL context for the device
auto ctx = boost::compute::context(device);

// create command queue for the device
auto queue = boost::compute::command_queue(ctx, device);

// do something
auto event = queue.enqueue_read_buffer_async(...)
```

Low-level API example #2

```
#include <boost/compute/core.hpp>

// get default compute device and print its name
auto device = boost::compute::system::default_device();

// get default OpenCL context
auto ctx = boost::compute::system::default_context();

// get default command queue
auto queue = boost::compute::system::default_queue();

// do something
auto event = queue.enqueue_read_buffer_async(...)
```

The default device is selected based on a set of heuristics and can be influenced using one of the following environment variables:

- `BOOST_COMPUTE_DEFAULT_DEVICE` - name of the compute device (e.g. "AMD Radeon")
- `BOOST_COMPUTE_DEFAULT_DEVICE_TYPE` - type of the compute device (e.g. "GPU" or "CPU")
- `BOOST_COMPUTE_DEFAULT_PLATFORM` - name of the platform (e.g. "NVIDIA CUDA")
- `BOOST_COMPUTE_DEFAULT_VENDOR` - name of the device vendor (e.g. "Intel")

OpenCL program caching

- Reduces OpenCL just-in-time compilation overhead

- Online caching using `program_cache` class
- Offline cache ensures one-per-system compilation
  - Built into program build process
  - `BOOST_COMPUTE_USE_OFFLINE_CACHE`

Kernel example

- Command queue is the only additional argument compared to pure C++ saxpy function

```
#include <boost/compute/core.hpp>
#include <boost/compute/container/vector.hpp>
#include <boost/compute/utility/program_cache.hpp>

void saxpy(const boost::compute::vector<float>& x,
           const boost::compute::vector<float>& y,
           const float a,
           boost::compute::command_queue& queue)
{
  boost::compute::context context = queue.get_context();
  // kernel source code
  std::string source =
    "__kernel void saxpy(__global float *x,"
    "                    __global float *y,"
    "                    const float a)"
    "{"
    "    const uint i = get_global_id(0);"
    "    y[i] = a * x[i] + y[i];"
    "}";
```

Kernel example

- Command queue is the only additional argument compared to pure C++ saxpy function
- Users can create their own program caches or use the global cache
- Program compilation, online and offline cache in one method

```cpp
  // get global cache (online)
  boost::shared_ptr<boost::compute::program_cache> global_cache =
    boost::compute::program_cache::get_global_cache(context);

  // set compilation options and cache key
  std::string options;
  std::string key = "__iwocl16_saxpy";

  // get compiled program from online cache,
  // load binary (offline caching) or compile it
  boost::compute::program program =
    global_cache->get_or_build(key, options, source, context);

  // create the saxpy kernel
  boost::compute::kernel kernel = program.create_kernel("saxpy");
  // set arguments (C++11 variadic templates) and run the kernel
  kernel.set_args(x.get_buffer(), y.get_buffer(), a);
  queue.enqueue_1d_range_kernel(kernel, 0, y.size(), 0);
}
```

Kernel example

- Command queue is the only additional argument compared to pure C++ saxpy function

- Users can create their own program caches or use the global cache
- Program compilation, online and offline cache in one method

- Less code
- Users do not need to:
  - Check for errors
  - Store compiled program objects
  - Compile each time function is called

```
#include <boost/compute/core.hpp>
#include <boost/compute/container/vector.hpp>
#include <boost/compute/utility/program_cache.hpp>

void saxpy(const boost::compute::vector<float>& x,
           const boost::compute::vector<float>& y,
           const float a,
           boost::compute::command_queue& queue)
{
  boost::compute::context context = queue.get_context();
  // kernel source code
  std::string source =
    "__kernel void saxpy(__global float *x,"
    "                    __global float *y,"
    "                    const float a)"
    "{"
    "    const uint i = get_global_id(0);"
    "    y[i] = a * x[i] + y[i];"
    "}";

  // get global cache (online)
  boost::shared_ptr<boost::compute::program_cache> global_cache =
    boost::compute::program_cache::get_global_cache(context);

  // set compilation options and cache key
  std::string options;
  std::string key = "__iwocl16_saxpy";

  // get compiled program from online cache, load binary (offline caching) or compile it
  boost::compute::program program =
    global_cache->get_or_build(key, options, source, context);

  // create the saxpy kernel, set argument and run it
  boost::compute::kernel kernel = program.create_kernel("saxpy");
  kernel.set_args(x.get_buffer(), y.get_buffer(), a);
  queue.enqueue_1d_range_kernel(kernel, 0, y.size(), 0);
}
```

High-level API

“Parallel Standard Template Library”

Algorithms

accumulate()
adjacent_difference()
adjacent_find()
all_of()
any_of()
binary_search()
copy()
copy_if()
copy_n()
count()
count_if()
equal()
equal_range()
exclusive_scan()
fill()
fill_n()
find()
find_end()
find_if()
find_if_not()
for_each()
gather()
generate()
generate_n()
includes()
inclusive_scan()
inner_product()
inplace_merge()
iota()
is_partitioned()
is_permutation()
is_sorted()
lower_bound()
lexicographical_compare()
max_element()
merge()
min_element()
minmax_element()
mismatch()
next_permutation()
none_of()
nth_element()
partial_sum()
partition()
partition_copy()
partition_point()
prev_permutation()
random_shuffle()
reduce()
reduce_by_key()
remove()
remove_if()
replace()
replace_copy()
reverse()
reverse_copy()
rotate()
rotate_copy()
scatter()
search()
search_n()
set_difference()
set_intersection()
set_symmetric_difference()
set_union()
sort()
sort_by_key()
stable_partition()
stable_sort()
stable_sort_by_key()
swap_ranges()
transform()
transform_reduce()
unique()
unique_copy()
upper_bound()

Containers

- array<T, N>
- dynamic_bitset<T>
- flat_map<Key, T>
- flat_set<T>
- **mapped_view<T>**
- stack<T>
- string
- valarray<T>
- **vector<T>**

Random Number Generators

- bernoulli_distribution
- default_random_engine
- discrete_distribution
- linear_congruential_engine
- mersenne_twister_engine
- threefry_engine
- normal_distribution
- uniform_int_distribution
- uniform_real_distribution

Iterators

- **buffer_iterator<T>**
- constant_buffer_iterator<T>
- constant_iterator<T>
- counting_iterator<T>
- discard_iterator
- function_input_iterator<Function>
- permutation_iterator<Elem, Index>
- strided_iterator<Iterator>
- **transform_iterator<Iterator, Function>**
- **zip_iterator<IteratorTuple>**

Sort example

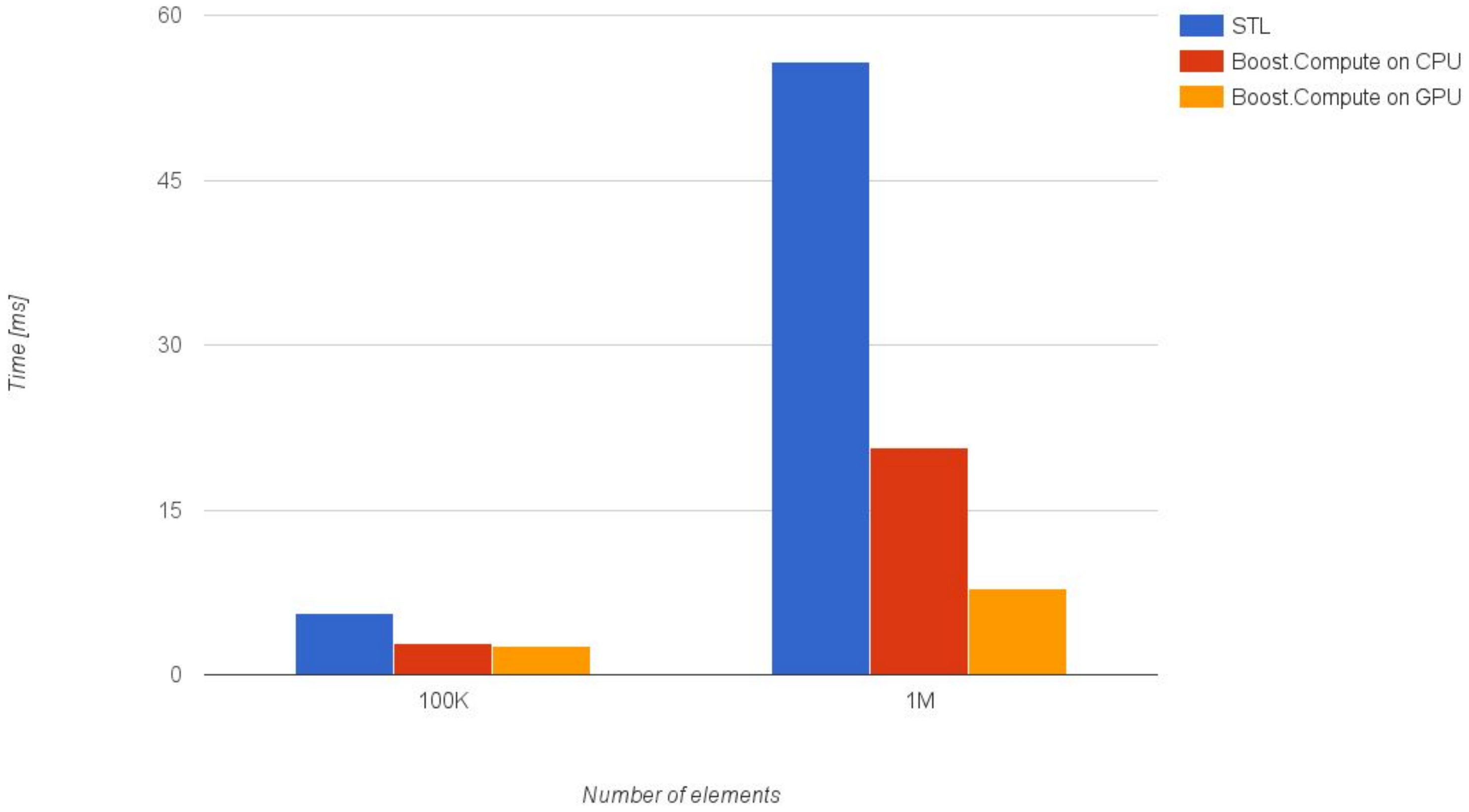
Standard Template Library

```cpp
#include <algorithm>
#include <vector>


std::vector<int> v;
// Fill the vector with some data
std::sort(v.begin(), v.end());
```

Boost.Compute

```cpp
#include <boost/compute/algorithm.hpp>
//#include <boost/compute/algorithm/sort.hpp>
#include <vector>

std::vector<int> v;
// Fill the vector with some data
boost::compute::sort(v.begin(), v.end(), queue);
```

Sort example



Kernel generation

```
boost::compute::vector<int> input = { ... };
boost::compute::vector<int> output(...);

boost::compute::transform(
    input.begin(),
    input.end(),
    output.begin(),
    boost::compute::abs<int>(),
    command_queue
);
```

- Boost.Compute algorithms generate OpenCL kernels, compile them and run on the device
- Internally Boost.Compute uses `meta_kernel` class for specifying kernel templates
- `meta_kernel` class is responsible for:
  - Generating final kernel
  - Online and offline caching
  - Enabling required OpenCL extensions

Kernel generation

```
boost::compute::vector<int> input = { ... };
boost::compute::vector<int> output(...);

boost::compute::transform(
    input.begin(),
    input.end(),
    output.begin(),
    boost::compute::abs<int>(),
    command_queue
);
```

Generated kernel

```
__kernel void transform(__global int * __buf1,
                        __global int * __buf2,
                        const uint count)
{
  uint gid = get_global_id(0);
  if(gid < count) {
    __buf2[gid] = abs(__buf1[gid]);
  }
}
```

Iterator adaptors

- Enhance the abilities of algorithms
- Can lead to more performant code

Iterator adaptors example

```
boost::compute::vector<float> result(5, context);
boost::compute::iota(
    result.begin(), result.end(),
    1.5f, command_queue
);
boost::compute::transform(
    result.begin(), result.end(), result.begin(),
    boost::compute::floor<float>(), command_queue
);
boost::compute::exclusive_scan(
    result.begin(), result.end(), result.begin(),
    command_queue
);
// result = {0.0, 1.0f, 3.0f, 6.0f, 10.0f}
```

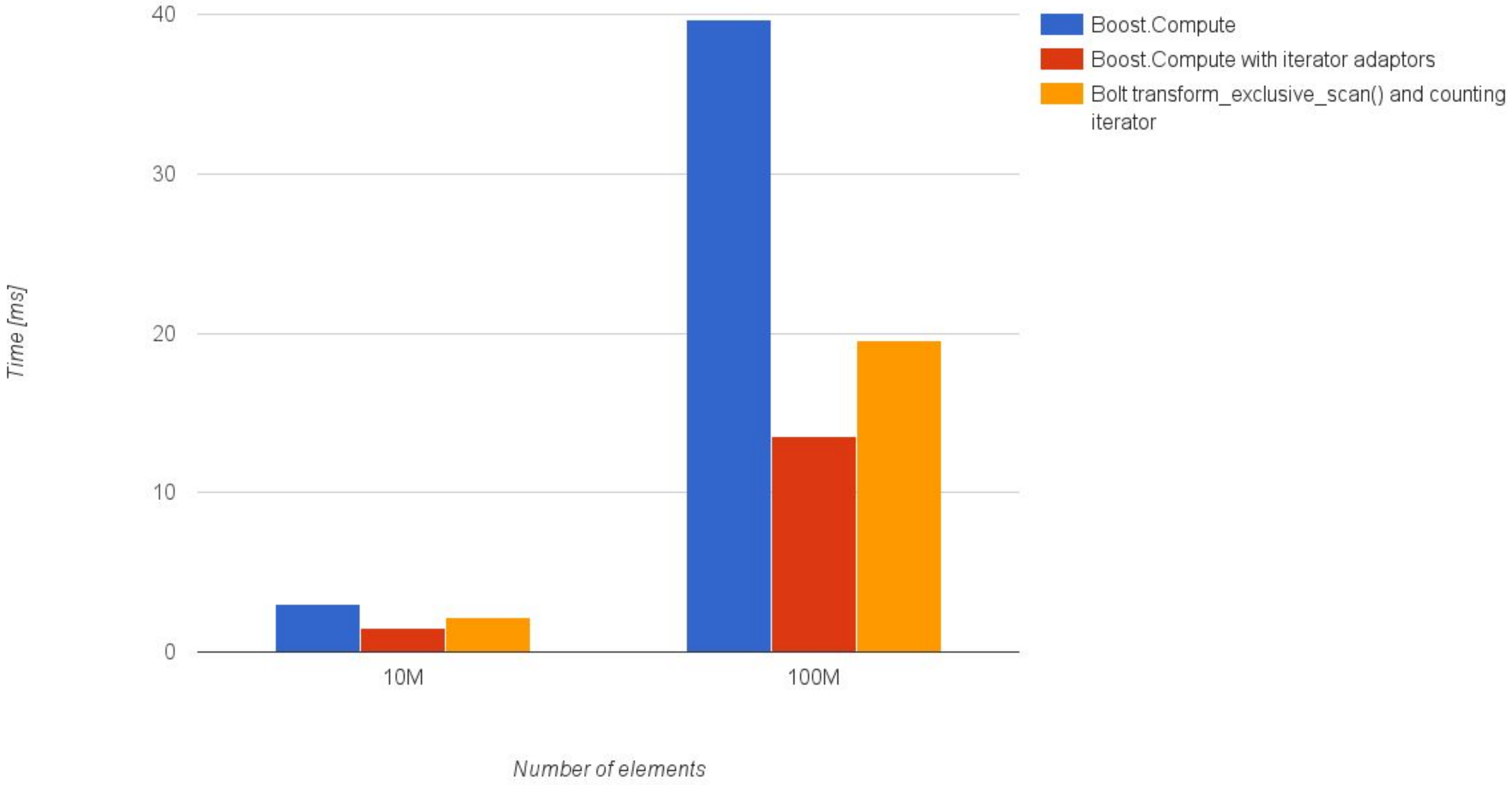
- Generate data
- Transform it
- Perform an exclusive scan operation

Iterator adaptors example

```
boost::compute::vector<float> result(5, context);
boost::compute::exclusive_scan(
    boost::compute::make_transform_iterator(
        boost::compute::make_counting_iterator<float>(1.5f),
        boost::compute::floor<float>()
    ),
    boost::compute::make_transform_iterator(
        boost::compute::make_counting_iterator<float>(6.5f),
        boost::compute::floor<float>()
    ),
    result.begin(),
    command_queue
);
// result = {0.0, 1.0f, 3.0f, 6.0f, 10.0f}
```

- Perform an exclusive scan operation

Iterator adaptors example



Custom functions

```
BOOST_COMPUTE_FUNCTION(int, add_three, (int x),
{
    return x + 3;
});

boost::compute::vector<int> vector = { ... };
boost::compute::transform(
    vector.begin(),
    vector.end(),
    vector.begin(),
    add_three,
    queue
);
```

- `BOOST_COMPUTE_FUNCTION()` macro produces Boost.Compute function object
- Based on function signature Boost.Compute verifies:
  - Function arity
  - Return type
  - Arguments’ types
- Body of the function is checked during OpenCL compilation

Closures

```
int y = 2;
BOOST_COMPUTE_CLOSURE(int, add_y, (int x), (y),
{
    return x + y;
});

boost::compute::vector<int> vector = { 1, 1, 1, 1, ... };
boost::compute::transform(vector.begin(), vector.end(), vector.begin(), add_y, queue);
// vector = { 3, 3, 3, 3, ... }

y = 3;
boost::compute::transform(vector.begin(), vector.end(), vector.begin(), add_y, queue);
// vector = { 6, 6, 6, 6, ... }
```

- Allow capturing of in-scope C++ variables

Lambda expressions

- Easy way for specifying custom function for algorithms
- Fully type-checked by the C++ compiler

```cpp
// placeholder
using boost::compute::_1;

boost::compute::vector<int> vec1 = { ... };
boost::compute::transform(vec1.begin(), vec1.end(), vec1.begin(), _1 + 3, queue);
```

Lambda expressions example

```
size_t n = 100;
boost::compute::vector<float> x(n, 1.0f, queue);
boost::compute::vector<float> y(n, 2.0f, queue);
boost::compute::vector<float> k(n, context);
boost::compute::vector<float> l(n, context);

float alpha = 1.5f;
float beta = -1.0f;

auto X = boost::compute::lambda::get<0>(boost::compute::_1);
auto Y = boost::compute::lambda::get<1>(boost::compute::_1);
auto K = boost::compute::lambda::get<2>(boost::compute::_1);
auto L = boost::compute::lambda::get<3>(boost::compute::_1);

using boost::compute::lambda::cos;
using boost::compute::lambda::sin;
```

```
boost::compute::for_each(
    boost::compute::make_zip_iterator(
        boost::make_tuple(
            x.begin(), y.begin(), k.begin(), l.begin()
        )
    ),
    boost::compute::make_zip_iterator(
        boost::make_tuple(
            x.end(), y.end(), k.end(), l.end()
        )
    ),
    boost::compute::lambda::make_tuple(
        K = alpha * sin(X) - cos(Y),
        L = beta * sin(Y) - cos(X)
    ),
    queue
);
```

Auto-tuning

```
C:\...\BoostCompute\build [master ≡]> .\perf\Release\perf_sort.exe 262144
size: 262144
device: Intel(R) HD Graphics 530
time: 39.6195 ms

C:\...\BoostCompute\build [master ≡]> .\perf\Release\perf_sort.exe 262144 --tune
size: 262144
device: Intel(R) HD Graphics 530
time: 27.9715 ms

C:\...\BoostCompute\build [master ≡]> .\perf\Release\perf_sort.exe 262144
size: 262144
device: Intel(R) HD Graphics 530
time: 27.8637 ms
```

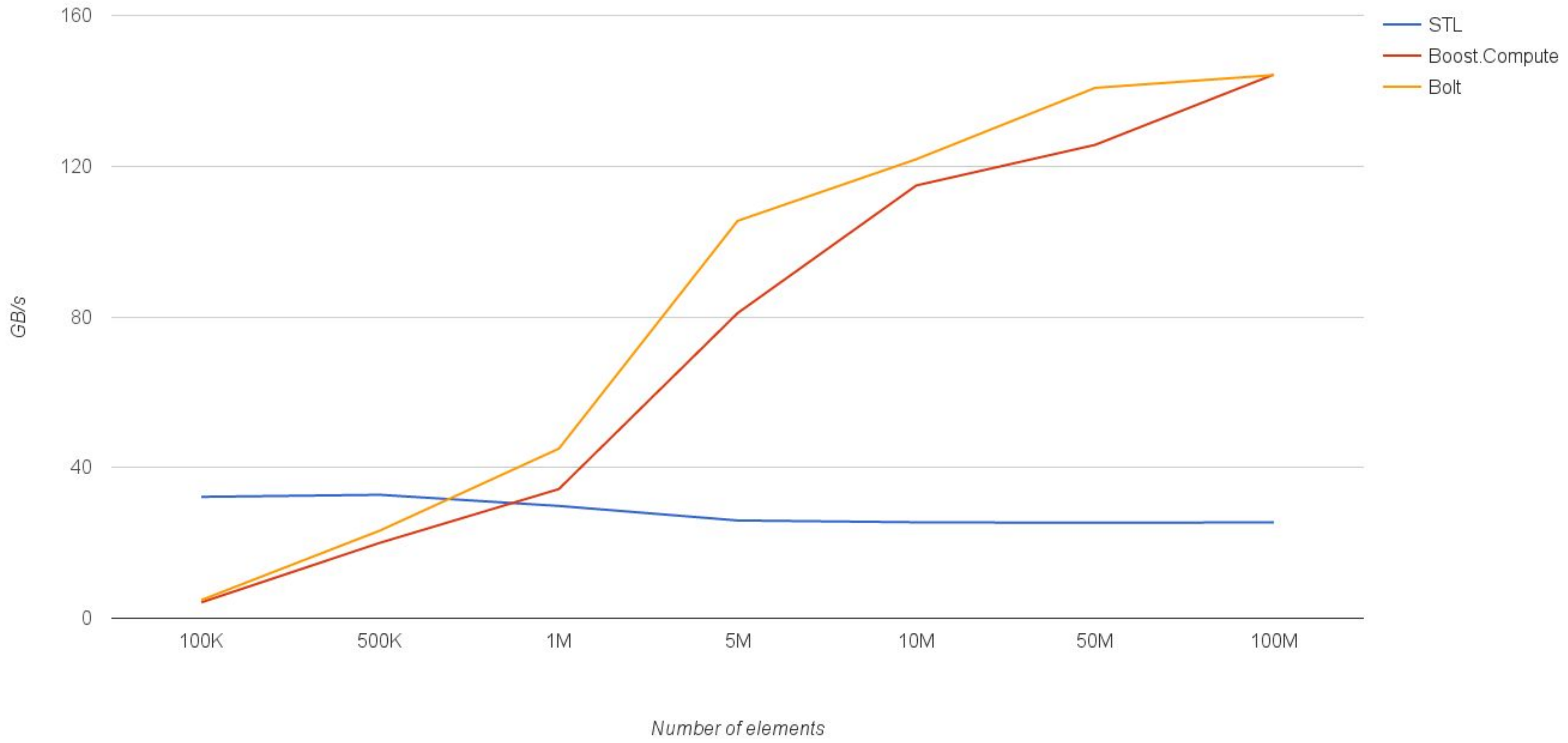
Performance

Benchmarks

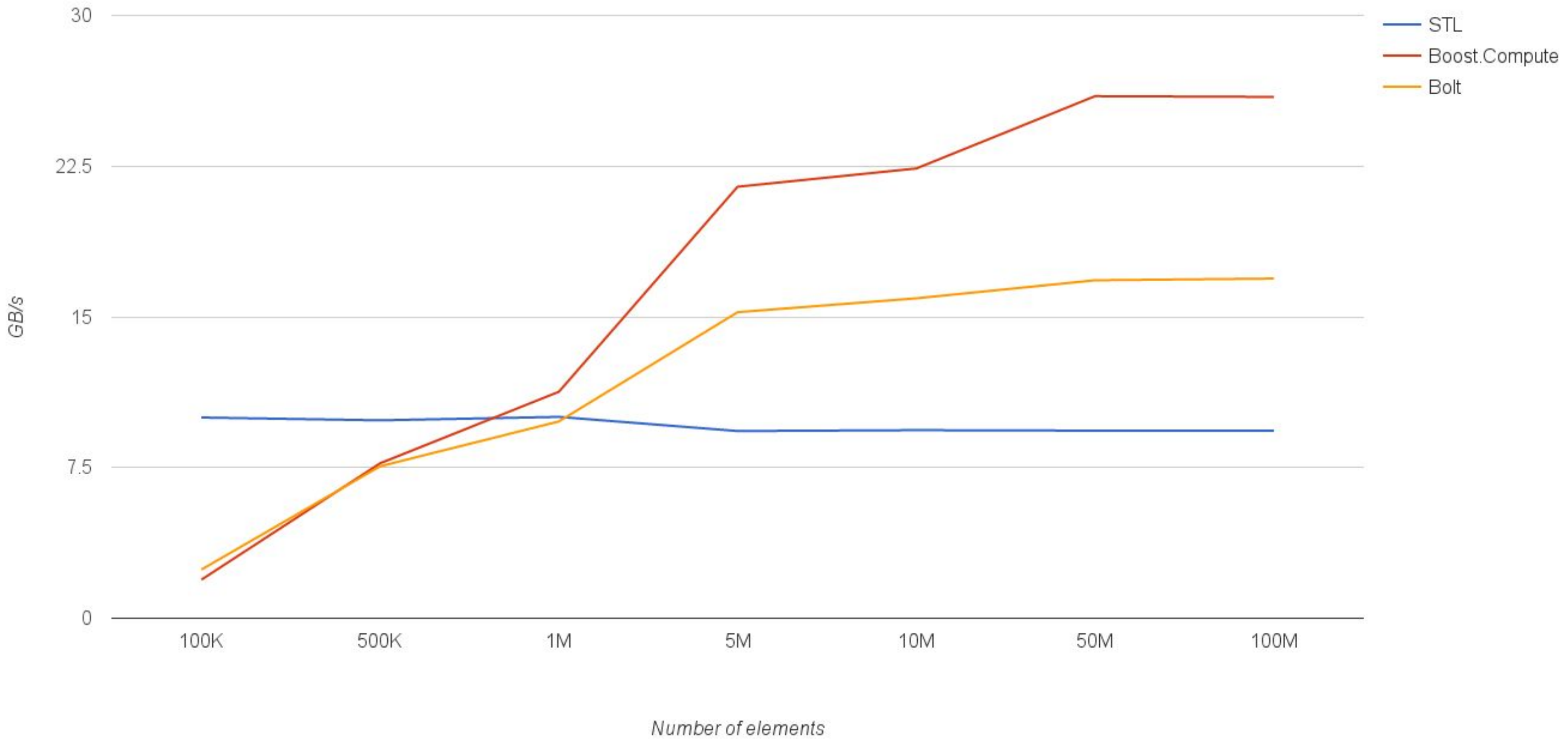
- Standard Template Library
- Boost.Compute
- Bolt C++ Template Library

- Intel Core i5-6600K
- AMD Radeon R9 380 4GB

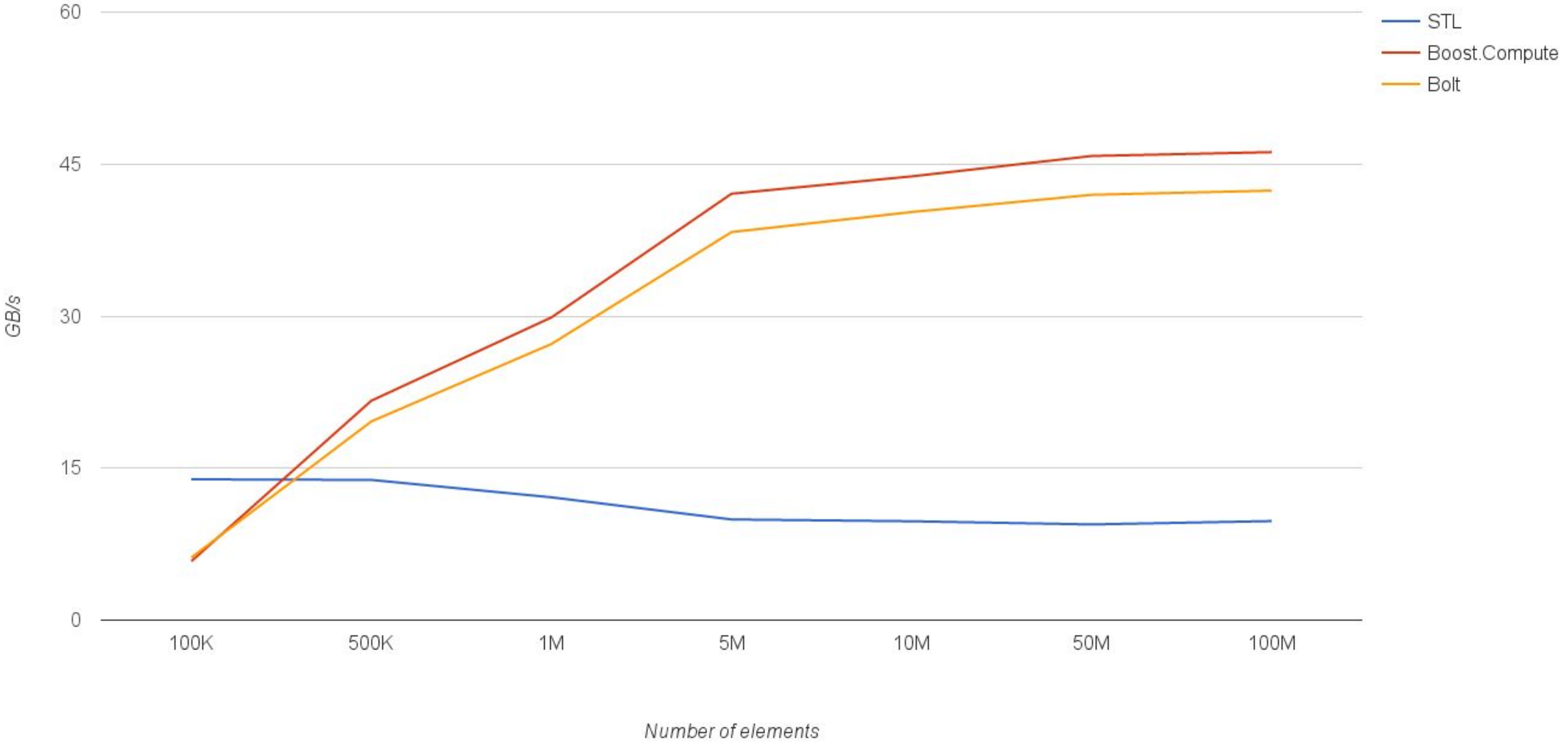
Reduce


160
120
80
40
0
GB/s
100K
500K
1M
5M
10M
50M
100M
Number of elements
STL
Boost.Compute
Bolt

Scan


30
22.5
15
7.5
0
GB/s
100K
500K
1M
5M
10M
50M
100M
Number of elements
STL
Boost.Compute
Bolt

Saxpy (transform)



Test details

- Test methodology:
    - 100 trials
    - Minimal execution time (best result)
- Test environment:
    - Intel Core i5-6600K @ 3.5GHz
        - Platform: Intel OpenCL
        - Driver version: 5.2.0.10094
        - Windows 10
    - AMD Radeon R9 380 4GB
        - Platform: AMD Accelerated Parallel Processing v3.0
        - Driver version: Crimson Edition 15.12 (15.302)
        - Linux

Questions

- Jakub Szuppe
  - E-mail: j.szuppe@gmail.com
  - Twitter: @JSZPP
- Boost.Compute:
  - https://github.com/boostorg/compute
  - http://boostorg.github.io/compute/