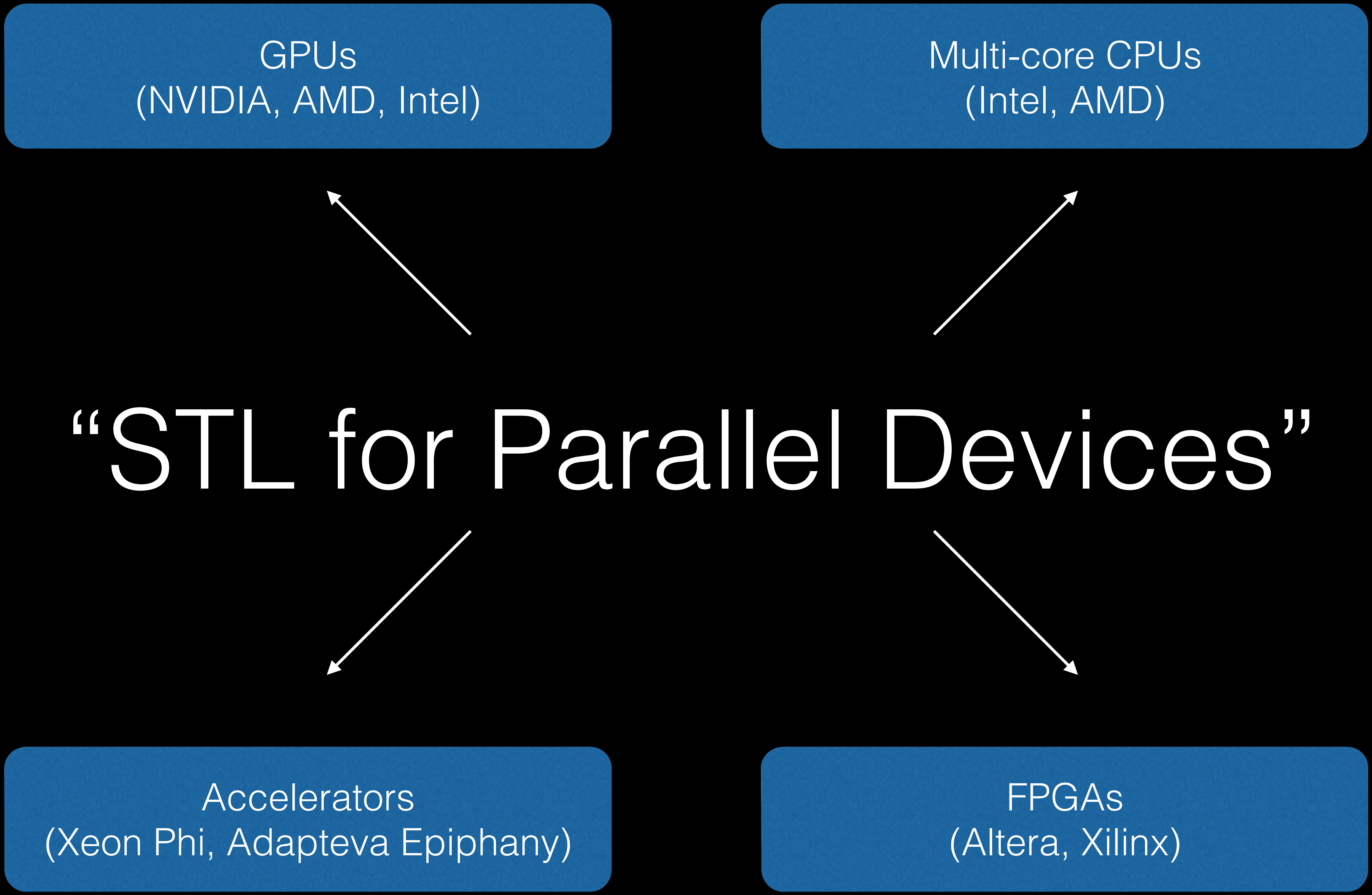
# Boost.Compute

A C++ library for GPU computing

Kyle Lutz



“STL for Parallel Devices”

GPUs
(NVIDIA, AMD, Intel)

Multi-core CPUs
(Intel, AMD)

Accelerators
(Xeon Phi, Adapteva Epiphany)

FPGAs
(Altera, Xilinx)

# Why Parallel?

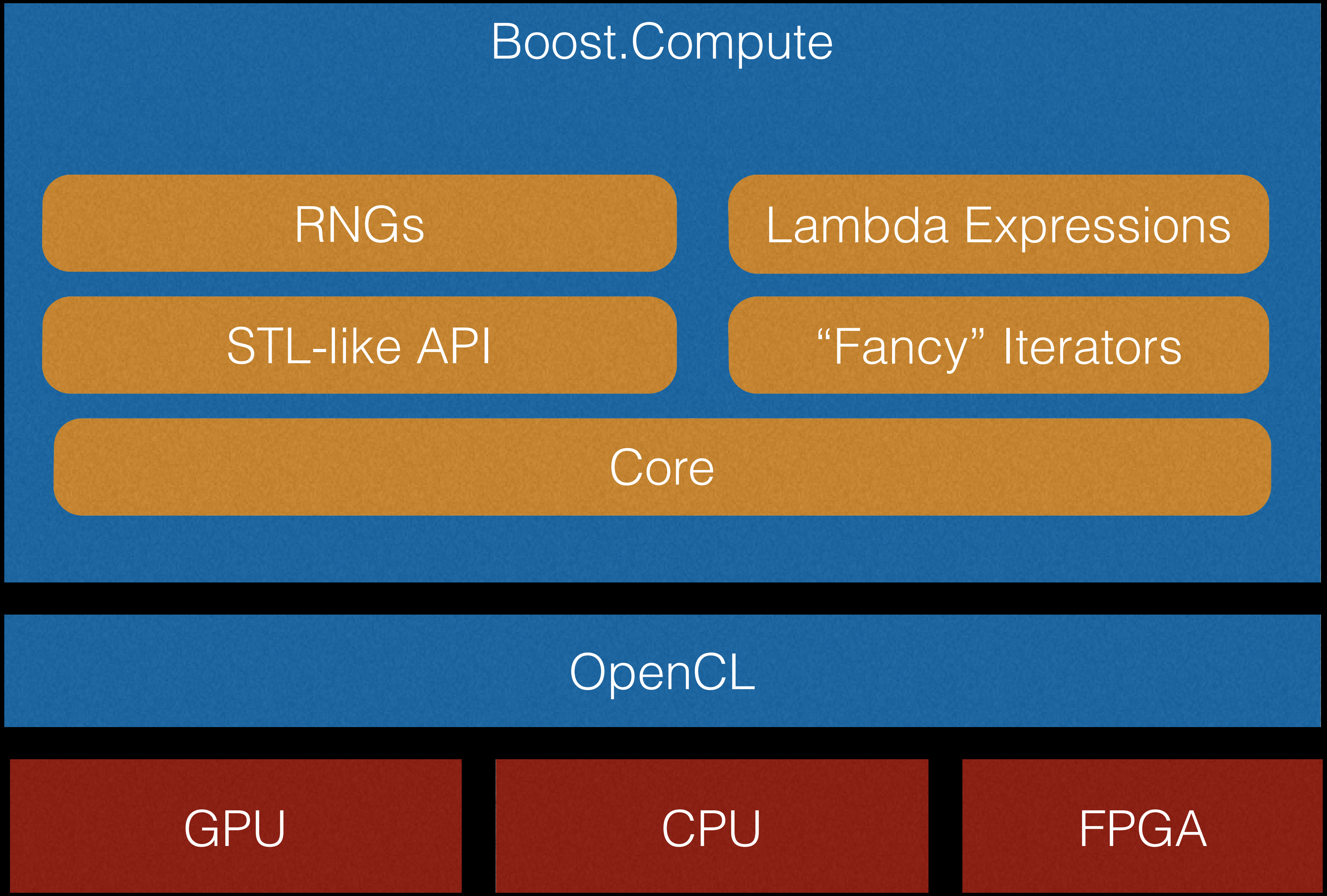
- “The free lunch is over”
- Single-threaded execution is a small percentage of total compute power in a modern machine

# Why STL?

- Familiar to most C++ programmers (or should be)
- Simplifies porting existing applications to parallel architectures

# Design

# Library Architecture



# Why OpenCL?

(or why not CUDA/Thrust/Bolt/SYCL/OpenACC/OpenMP/C++AMP?)

- Standard C++ (no special compiler or compiler extensions)
- Library-based solution (no special build-system integration)
- Vendor-neutral, open-standard

# Low-level API

# Low-level API

- Provides classes to wrap OpenCL objects such as buffer, context, program, and command_queue.
- Takes care of reference counting and error checking
- Also provides utility functions for handling error codes or setting up the default device

# Low-level API

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

// lookup default compute device
auto gpu = boost::compute::system::default_device();

// create opencl context for the device
auto ctx = boost::compute::context(gpu);

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

// print device name
std::cout << “device = “ << gpu.name() << std::endl;
```

```
for(auto& device : boost::compute::system::devices()){
    std::cout << "device: " << device.name() << std::endl;
}
```

→

```
// query number of opencl platforms
cl_uint num_platforms = 0;
cl_int ret = clGetPlatformIDs(0, NULL, &num_platforms);
if(ret != CL_SUCCESS){
    std::cerr << "failed to query platforms: " << ret << std::endl;
    return -1;
}

// check that at least one platform was found
if(num_platforms == 0){
    std::cerr << "found 0 platforms" << std::endl;
    return 0;
}

// get platform ids
cl_platform_id *platforms = new cl_platform_id[num_platforms];
clGetPlatformIDs(num_platforms, platforms, NULL);

// iterate through each platform and query its devices
for(cl_uint i = 0; i < num_platforms; i++){
    cl_platform_id platform = platforms[i];

    // query number of opencl devices
    cl_uint num_devices = 0;
    ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, 0, NULL, &num_devices);
    if(ret != CL_SUCCESS){
        std::cerr << "failed to lookup devices for platform " << i << std::endl;
        continue;
    }

    // print number of devices found
    std::cout << "platform " << i << " has " << num_devices << " devices:" << std::endl;

    // get device ids for the platform
    cl_device_id *devices = new cl_device_id[num_devices];
    ret = clGetDeviceIDs(platform, CL_DEVICE_TYPE_ALL, num_devices, devices, NULL);
    if(ret != CL_SUCCESS){
        std::cerr << "failed to query platform devices" << std::endl;
        delete[] devices;
        continue;
    }

    // iterate through each device on the platform and print its name
    for(cl_uint j = 0; j < num_devices; j++){
        cl_device_id device = devices[j];

        // get length of the device name string
        size_t name_length = 0;
        ret = clGetDeviceInfo(device, CL_DEVICE_NAME, 0, NULL, &name_length);
        if(ret != CL_SUCCESS){
            std::cerr << "failed to query device name length for device " << j << std::endl;
            continue;
        }

        // get the device name string
        char *name = new char[name_length];
        ret = clGetDeviceInfo(device, CL_DEVICE_NAME, name_length, name, NULL);
        if(ret != CL_SUCCESS){
            std::cerr << "failed to query device name string for device " << j << std::endl;
            delete[] name;
            continue;
        }

        // print out the device name
        std::cout << "  device: " << name << std::endl;

        delete[] name;
    }
    delete[] devices;
}
delete[] platforms;
```

# High-level API

# 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()
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()
swap_ranges()
transform()
transform_reduce()
unique()
unique_copy()
upper_bound()

## Containers

```
array<T, N>
dynamic_bitset<T>
flat_map<Key, T>
flat_set<T>
stack<T>
string
valarray<T>
vector<T>
```

## Iterators

```
buffer_iterator<T>
constant_buffer_iterator<T>
constant_iterator<T>
counting_iterator<T>
discard_iterator
function_input_iterator<Function>
permutation_iterator<Elem, Index>
transform_iterator<Iter, Function>
zip_iterator<IterTuple>
```

## Random Number Generators

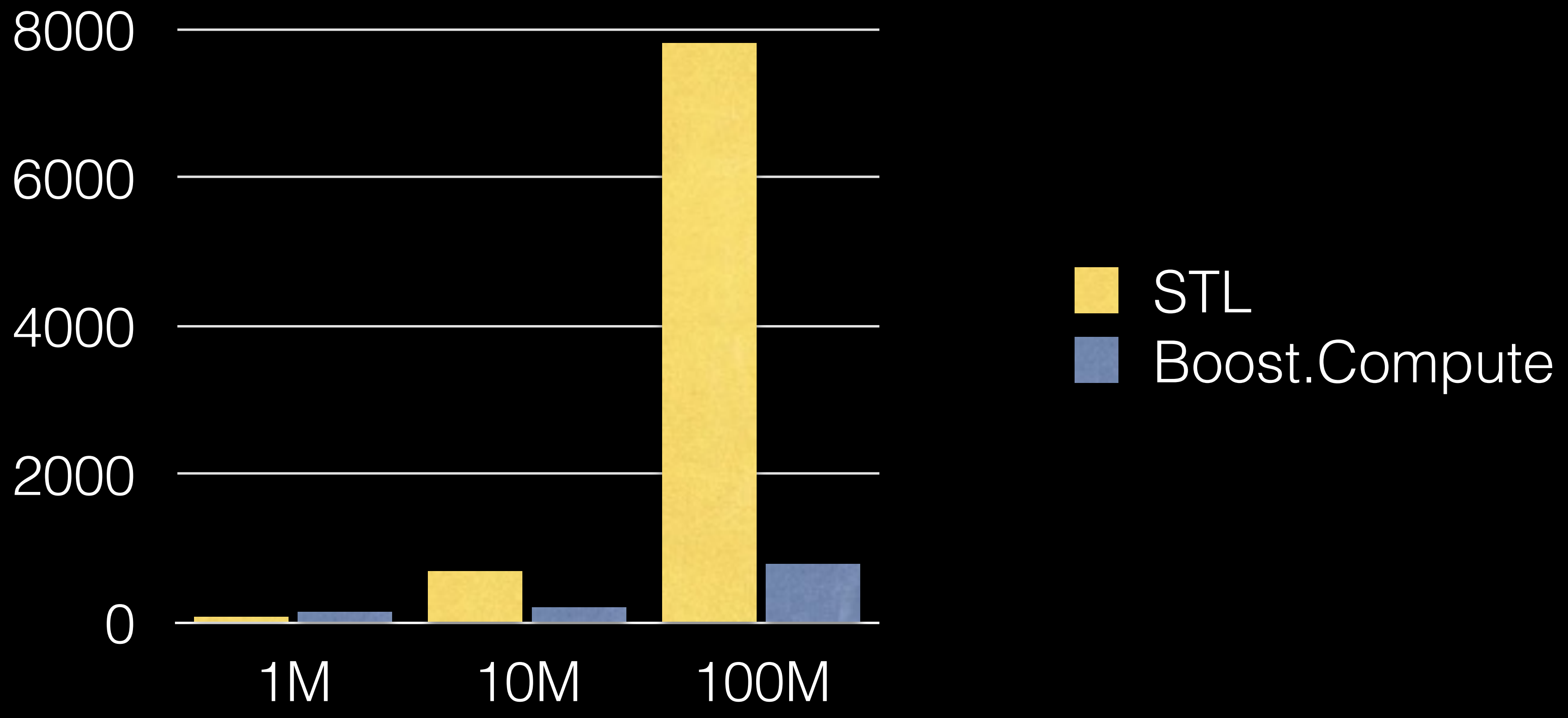
```
bernoulli_distribution
default_random_engine
discrete_distribution
linear_congruential_engine
mersenne_twister_engine
normal_distribution
uniform_int_distribution
uniform_real_distribution
```

# Sort Host Data

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

std::vector<int> vec = { ... };

std::sort(vec.begin(), vec.end());
```

# Sort Host Data

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

std::vector<int> vec = { ... };

boost::compute::sort(vec.begin(), vec.end(), queue);
```



# Parallel Reduction

```
#include <boost/compute/algorithm/reduce.hpp>
#include <boost/compute/container/vector.hpp>

boost::compute::vector<int> data = { ... };

int sum = 0;

boost::compute::reduce(
    data.begin(), data.end(), &sum, queue
);

std::cout << “sum = “ << sum << std::endl;
```

# Algorithm Internals

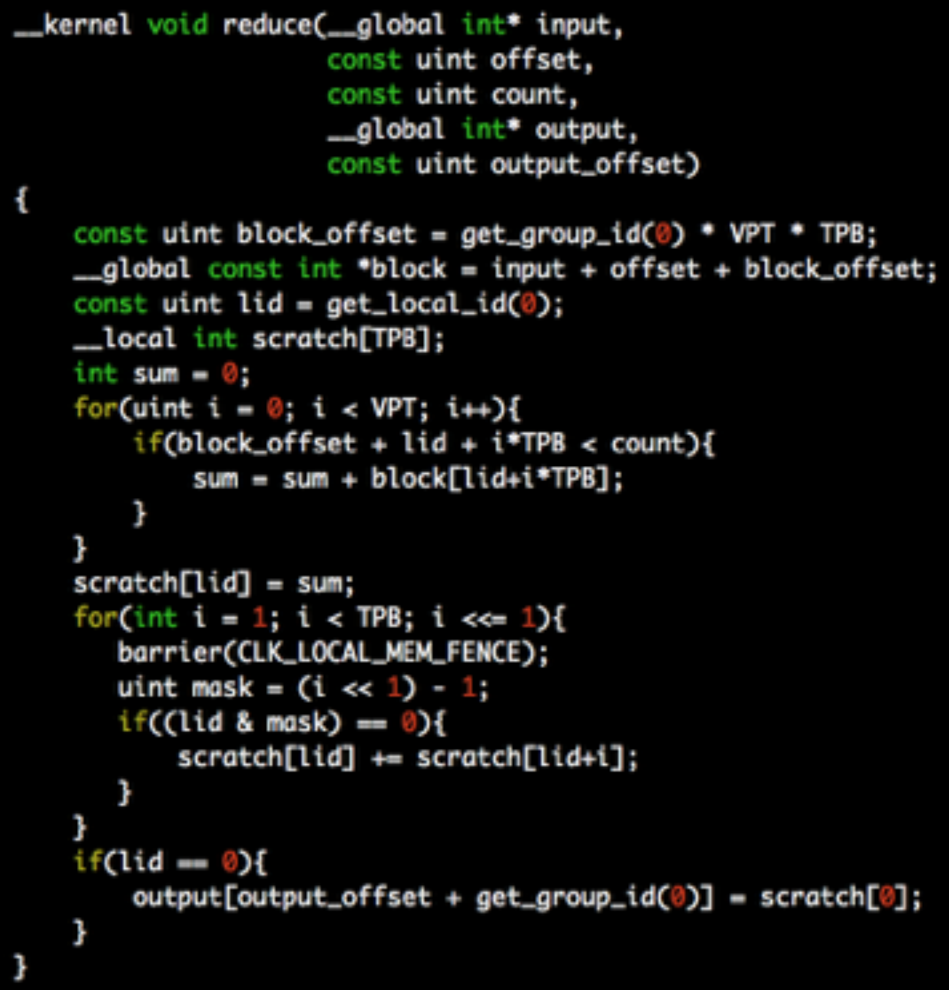
- Fundamentally, STL-like algorithms produce OpenCL kernel objects which are executed on a compute device.

C++

```
boost::compute::reduce(
    data.begin(), data.end(), &sum, queue
);
```

→

OpenCL



```
__kernel void reduce(__global int* input,
                     const uint offset,
                     const uint count,
                     __global int* output,
                     const uint output_offset)
{
    const uint block_offset = get_group_id(0) * VPT * TPB;
    __global const int *block = input + offset + block_offset;
    const uint lid = get_local_id(0);
    __local int scratch[TPB];
    int sum = 0;
    for(uint i = 0; i < VPT; i++){
        if(block_offset + lid + i*TPB < count){
            sum = sum + block[lid+i*TPB];
        }
    }
    scratch[lid] = sum;
    for(int i = 1; i < TPB; i <<= 1){
       barrier(CLK_LOCAL_MEM_FENCE);
       uint mask = (i << 1) - 1;
       if((lid & mask) == 0){
           scratch[lid] += scratch[lid+i];
       }
    }
    if(lid == 0){
        output[output_offset + get_group_id(0)] = scratch[0];
    }
}
```

# Custom Functions

```
BOOST_COMPUTE_FUNCTION(int, plus_two, (int x),
{
    return x + 2;
});

boost::compute::transform(
    v.begin(), v.end(), v.begin(), plus_two, queue
);
```

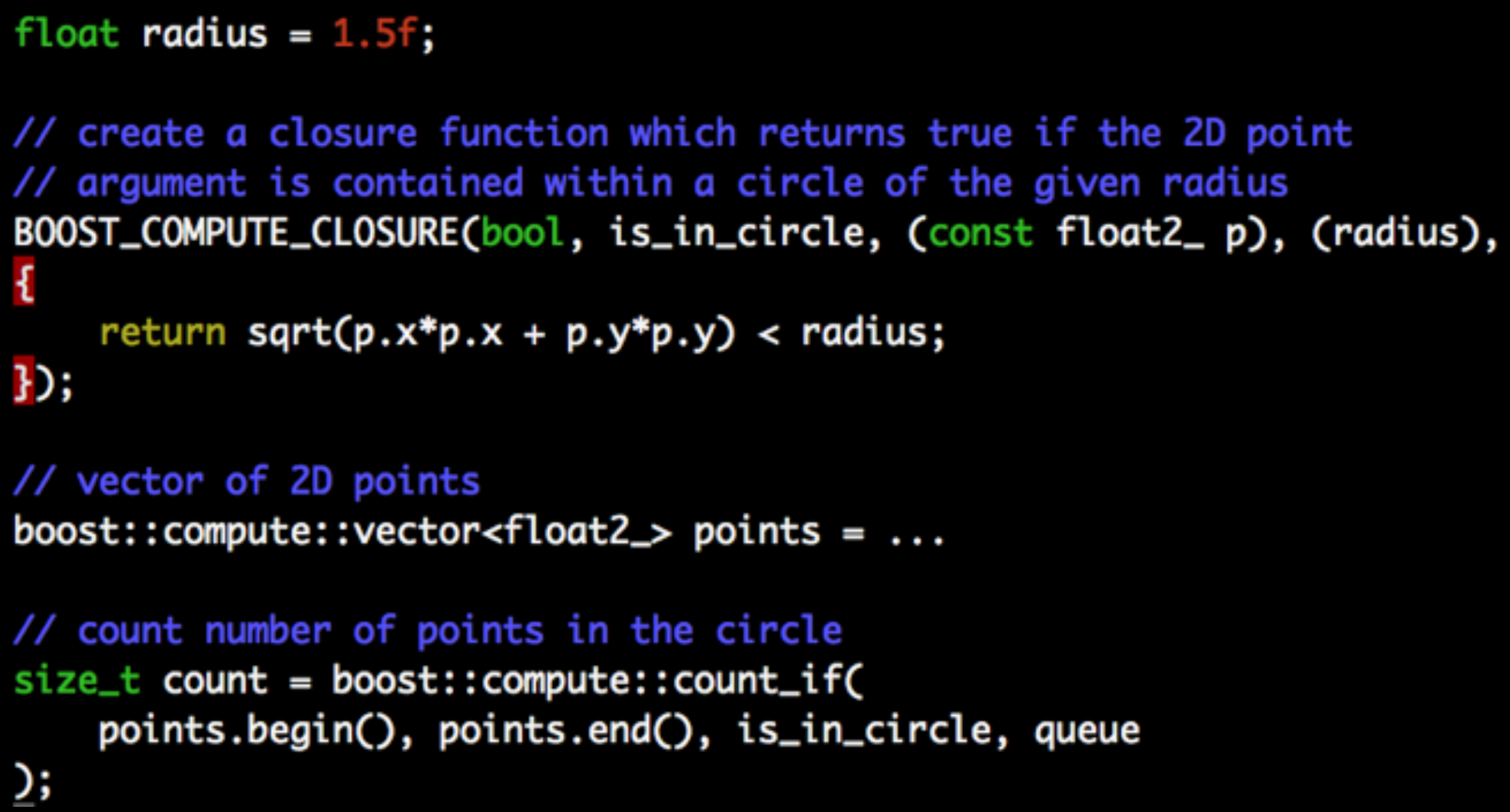
# Lambda Expressions

- Offers a concise syntax for specifying custom operations
- Fully type-checked by the C++ compiler

```
using boost::compute::lambda::_1;

boost::compute::transform(
    v.begin(), v.end(), v.begin(), _1 + 2, queue
);
```

# Closures

- Similar to BOOST_COMPUTE_FUNCTION()
- Additionally allow capturing of in-scope C++ variables



```
float radius = 1.5f;

// create a closure function which returns true if the 2D point
// argument is contained within a circle of the given radius
BOOST_COMPUTE_CLOSURE(bool, is_in_circle, (const float2_ p), (radius),
{
    return sqrt(p.x*p.x + p.y*p.y) < radius;
});

// vector of 2D points
boost::compute::vector<float2_> points = ...

// count number of points in the circle
size_t count = boost::compute::count_if(
    points.begin(), points.end(), is_in_circle, queue
);
```

# Iterator Adaptors

- Augment abilities of existing algorithms
- Leads to more performant code

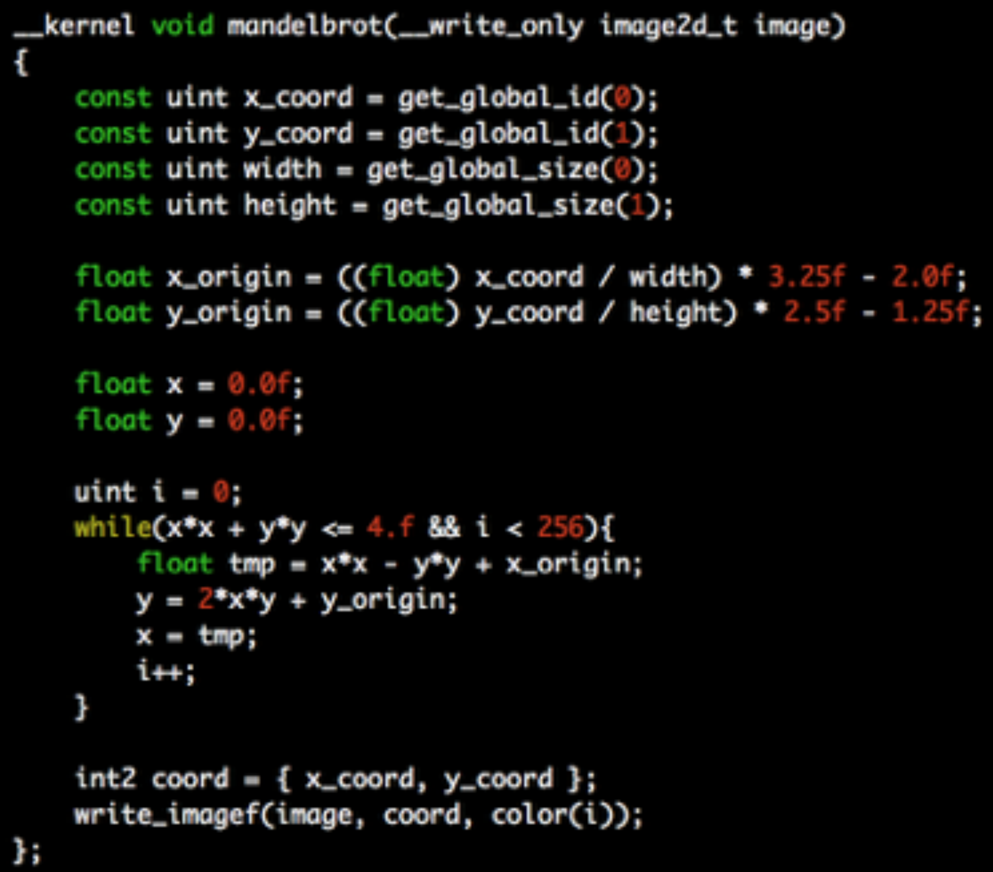
```
boost::compute::vector<int> v = ...;

int abs_sum = boost::compute::accumulate(
    make_transform_iterator(v.begin(), abs<int>()),
    make_transform_iterator(v.end(), abs<int>()),
    0,
    queue
);
```

# Additional Features

# OpenGL Interop

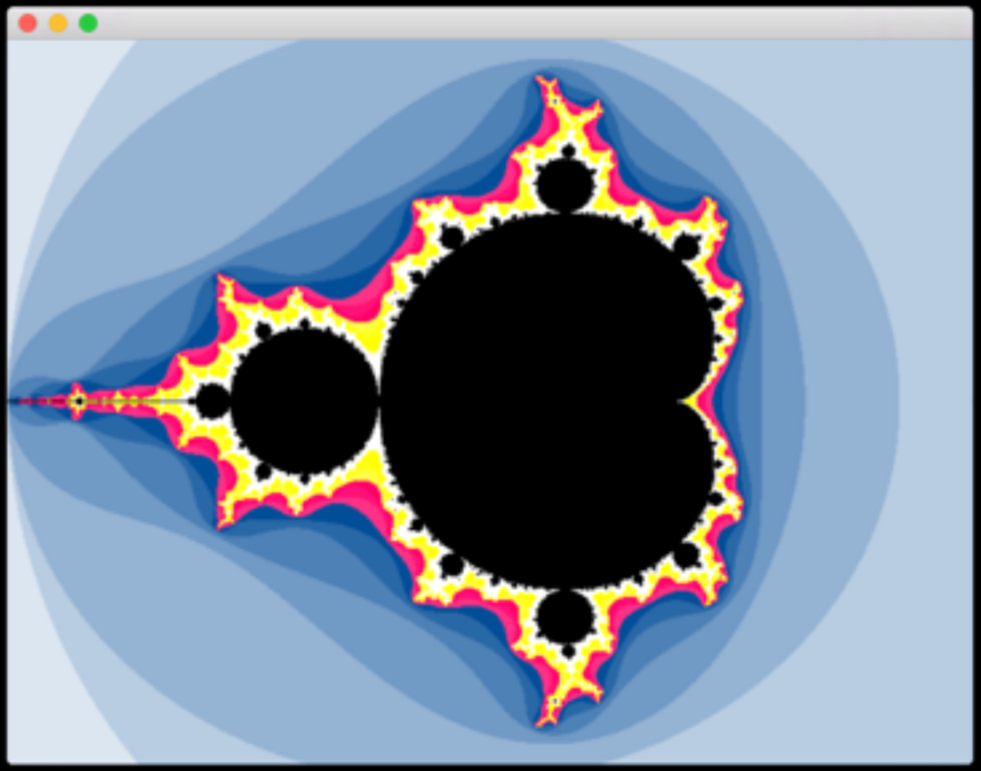
- OpenCL provides mechanisms for synchronizing with OpenGL to implement direct rendering on the GPU
- Boost.Compute provides easy to use functions for interacting with OpenGL in a portable manner.

## OpenCL



```
__kernel void mandelbrot(__write_only image2d_t image)
{
    const uint x_coord = get_global_id(0);
    const uint y_coord = get_global_id(1);
    const uint width = get_global_size(0);
    const uint height = get_global_size(1);

    float x_origin = ((float) x_coord / width) * 3.25f - 2.0f;
    float y_origin = ((float) y_coord / height) * 2.5f - 1.25f;

    float x = 0.0f;
    float y = 0.0f;

    uint i = 0;
    while(x*x + y*y <= 4.f && i < 256){
        float tmp = x*x - y*y + x_origin;
        y = 2*x*y + y_origin;
        x = tmp;
        i++;
    }

    int2 coord = { x_coord, y_coord };
    write_imagef(image, coord, color(i));
};
```

## OpenGL



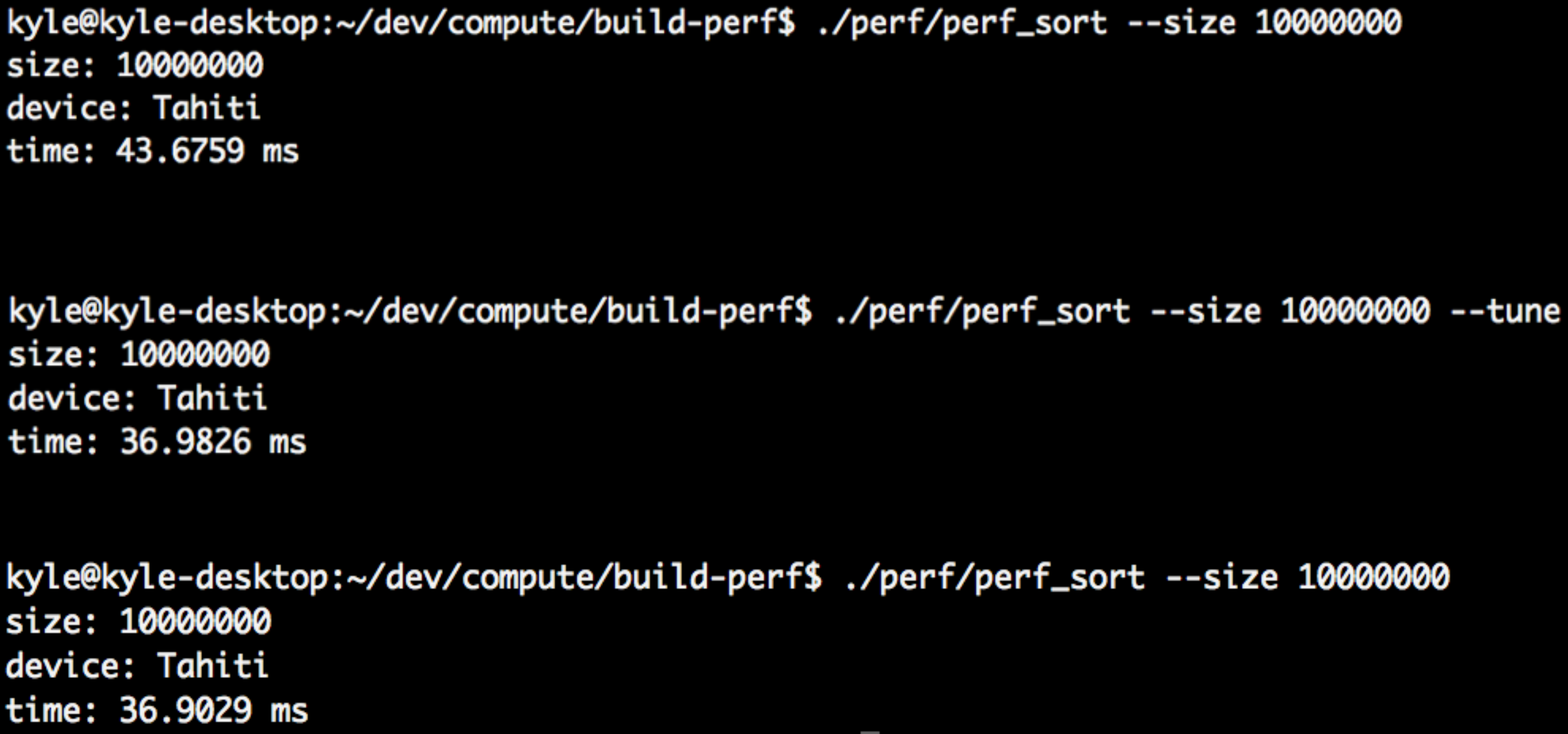
# Program Caching

- Helps mitigate run-time kernel compilation costs
- Frequently-used kernels are stored and retrieved from the global cache
- Offline cache reduces this to one compilation per system

# Auto-tuning

- OpenCL supports a wide variety of hardware with diverse execution characteristics
- Algorithms support different execution parameters such as work-group size, amount of work to execute serially
- These parameters are tunable and their results are measurable
- Boost.Compute includes benchmarks and tuning utilities to find the optimal parameters for a given device

# Auto-tuning



```
kyle@kyle-desktop:~/dev/compute/build-perf$ ./perf/perf_sort --size 10000000
size: 10000000
device: Tahiti
time: 43.6759 ms
```

```
kyle@kyle-desktop:~/dev/compute/build-perf$ ./perf/perf_sort --size 10000000 --tune
size: 10000000
device: Tahiti
time: 36.9826 ms
```

```
kyle@kyle-desktop:~/dev/compute/build-perf$ ./perf/perf_sort --size 10000000
size: 10000000
device: Tahiti
time: 36.9029 ms
```

# Recent News

# Coming soon to Boost

- Went through Boost peer-review in December 2014
- Accepted as an official Boost library in January 2015
- Should be packaged in a Boost release this year (1.59)

# Thank You

Source

https://github.com/boostorg/compute

Documentation

http://boostorg.github.io/compute