Implementing ParallelSTL using SYCL

Monday 11\textsuperscript{th} May, 2015

Ruyman Reyes ruyman@codeplay.com

Codeplay Research
Who am I

Ruyman Reyes, Ph.D

- R&D Runtime Engineer, Codeplay
  → SYCL runtime and spec. contributor.
- > 5 years expr. HPC
  → BSC, EPCC among others
- Research in P.M. for Heterogeneous Computing
  → OpenMP, OMPSs, OpenAcc

✉️ ruyman@codeplay.com
🐦 @Ruyk
Overview

1 Background

2 The N4409 Proposal

3 A SYCL-based Parallel STL
Background
STL - C++ Standard Template Library

- Widely used algorithm/container library
- Template-based to enable compile-time optimizations
- Any container can be used on any algorithm via iterators
Using the STL

Using vectors, lists and algorithms

```cpp
std::vector<int> myVector = { 8, 9, 1, 4 };  
std::list<int> myList;  
// Insert something at the end of the vector  
myVector.push_back(33);  
// Sort the vector  
std::sort(myVector.begin(), myVector.end());  
// Check that myVector is sorted  
if (std::is_sorted(std::begin(myVector), std::end(myVector)))  
{
    std::cout << " Data is sorted! " << endl;
}
// Copy elements larger than 10  
std::copy_if(std::begin(myVector), std::end(myVector),  
            std::back_inserter(myList),  
            [](int i){ return (i>10); });  
// The list should only have one element  
std::cout << " List size :" << myList.size() << std::endl;
```
Parallel libraries in Heterogeneous computing

- Each vendor has its own parallel-algorithm library
- Interface resembles STL but different
- Makes code platform specific!
The N4409 Proposal
Parallel STL: Democratizing Parallelism in C++

- Group of software engineers from Intel, Microsoft and Nvidia
  → Based on TBB (Intel), PPL/C++AMP (MS), Thrust (Nvidia)
- Working draft for C++17 N4409
  → Previously: N3554, N3850, N3960, N4071
  → Describes an interface to algorithms with parallel execution
  → Perform parallel operations on generic containers
- Extends the current STL interface with policies allowing parallel execution.
Parallel STL: Democratizing Parallelism in C++

- Group of software engineers from Intel, Microsoft and Nvidia
  → Based on TBB (Intel), PPL/C++AMP (MS), Thrust (Nvidia)

- Working draft for C++17 N4409
  → Previously: N3554, N3850, N3960, N4071
  → Describes an interface to algorithms with parallel execution
  → Perform parallel operations on generic containers

- Extends the current STL interface with policies allowing parallel execution.

This is still not part of the C++ standard

This is a work-in-progress proposal for the upcoming C++17
A sequential sort

```cpp
std::vector<int> data = { 8, 9, 1, 4 };  
std::sort(data.begin(), data.end());  
if (std::is_sorted(data)) {
    cout << " Data is sorted! " << endl;
}
```
A parallel sort

```cpp
std::vector<int> data = { 8, 9, 1, 4 };
std::experimental::parallel::sort(par, data.begin(), data.end());
if (std::is_sorted(data)) {
    cout << " Data is sorted! " << endl;
}
```

- `par` is an *Execution Policy*
- Both the execution policy and sort itself are defined on the `experimental::parallel` namespace
  → Will be in `std::` if approved
The Execution Policy

- Enables a standard algorithm to be potentially executed in parallel
  → Parallel execution is not guaranteed
- User is responsible of providing valid parallel code.
- Different libraries may have different policies
The Execution Policy

- Enables a standard algorithm to be potentially executed in parallel
  → Parallel execution is not guaranteed
- User is responsible of providing valid parallel code.
- Different libraries may have different policies

Why not just a parallel implementation?

- Parallelism will not fit by default all problems
- Different algorithms may be better for different platforms
- Developer may want to use custom policies
Default policies

Standard policy classes

- sequential_policy: Never do parallel
- parallel_execution_policy: Do default parallelism
- vector_execution_policy: Use vectorisation if possible
- execution_policy: Dynamic execution policy

Standard policy objects

extern const sequential_execution_policy seq;
extern const parallel_execution_policy par;
extern const vector_execution_policy vec;
using namespace std::experimental::parallel;
std::vector<int> vec = ...;
size_t threshold = 100u;

execution_policy exec = seq;

if (vec.size() > threshold) {
    exec = par;
}

sort(exec, vec.begin(), vec.end());
Current implementations

- The Microsoft Codeplex Parallel STL
  [https://parallelstl.codeplex.com/](https://parallelstl.codeplex.com/)
- Thrust: Implement policies for algorithms
  [https://thrust.github.io/](https://thrust.github.io/)

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

- Some other implementations from the public avail. on github.
  → Search for Parallel STL!
Algorithms

- The proposal slightly modifies some algorithms
  → for_each does not return the Functor again
- Some new algorithms are added
  → for_each_n
  → inclusive_scan, exclusive_scan
- Many of the already defined STL algorithms shall have the ExecutionPolicy overloads.
  → copy_if, copy_n, remove, ...
A SYCL-based Parallel STL
Codeplay SYCL STL implementation

- Khronos Open Source License
- Available on Github: https://github.com/KhronosGroup/SyclParallelSTL
- Current basic implementation:
  - Policy mechanism in place
  - sort (bitonic if size is power of 2, seq on gpu otherwise)
  - parallel `transform`
  - parallel `for_each`

Plenty of opportunities for your contribution!

- Implement more functions
- Improve algorithms
- Suggestions on the interface (e.g. Device-side vector?)
A sequential sort

```cpp
vector<int> data = { 8, 9, 1, 4 };  
sort(data.begin(), data.end());  
if (is_sorted(data)) {  
    cout << " Data is sorted! " << endl;  
}
```
```cpp
vector<int> data = { 8, 9, 1, 4 };  
sycl::sort(sycl_policy, v.begin(), v.end());  
if (is_sorted(data)) {
    cout << " Data is sorted! " << endl;
}
```
Sorting with the STL

Sorting on the GPU!

```cpp
vector<int> data = { 8, 9, 1, 4 };
sycl::sort(sycl_policy, v.begin(), v.end());
if (is_sorted(data)) {
    cout << " Data is sorted! " << endl;
}
```

- `sycl_policy` is an *Execution Policy*
- `data` is an standard stl::vector
- Technically will use the device returned by *default_selector*
The SYCL Policy

```
template <typename KernelName = DefaultKernelName>
class sycl_execution_policy {
    public:

    using kernelName = KernelName;

    sycl_execution_policy() = default;
    sycl_execution_policy(cl::sycl::queue q);
    cl::sycl::queue get_queue() const;
};
```

- Indicates algorithm will be executed using a SYCL-device
- Can optionally take a queue
  → Re-use device-selection
  → Asynchronous data copy-back
  → ...

Why the KernelName template?

Two separate calls can generate different kernels!

```cpp
transform(par, v.begin(), v.end(), [=](int& val){ val++; });
transform(par, v.begin(), v.end(), [=](int& val){ val--; });
```
Why the KernelName template?

Two separate calls can generate different kernels!

```cpp
transform(par, v.begin(), v.end(), [=](int& val){ val++; });
transform(par, v.begin(), v.end(), [=](int& val){ val--; });
```

Why is this?

```cpp
auto f = [vectorSize, &bufI, &bufO, op](cl::sycl::handler &h) mutable {
...
    auto aI = bufI.template get_access<access::mode::read>(h);
    auto aO = bufO.template get_access<access::mode::write>(h);
    h.parallel_for< /* The Kernel Name */>(r,
        [aI, aO, op](cl::sycl::id<1> id) {
            aO[id.get(0)] = UserFunctor(aI[id.get(0)]);
        });
};
```
Using named policies and queues

```cpp
using namespace cl::sycl;
using namespace experimental::parallel::sycl;

std::vector<int> v = ...;
// Transform
default_selector ds;
{
    queue q(ds);
    sort(sycl_execution_policy(q), v.begin(), v.end());
    sycl_execution_policy<class myName> sepn1(q);
    transform(sepn1, v2.begin(), v2.end(),
             v2.begin(), [=](int i) { return i + 1;});
}
```

- Only required for lambdas, not functors
- Device selection and queue are re-used
- Data is copied in/out in each call!
Avoiding data-copies using buffers

```cpp
using namespace cl::sycl;
using namespace experimental::parallel::sycl;

std::vector<int> v = ...;
default_selector h;
{
    buffer<int> b(v.begin(), v.end());
b.set_final_data(v.data());
{
    cl::sycl::queue q(h);
sort(sycl_execution_policy(q), begin(b), end(b));
sycl_execution_policy sepn1(q);
transform(sepn1, v2.begin(), v2.end(),
    v2.begin(), [=](int i) { return i + 1;});
}
}
```

- Buffer is constructed from STL containers
- Data will be copied back to the container when buffer is done
  → Note the additional copy from vec to buffer and viceversa
Using device-only data

```cpp
using namespace experimental::parallel::sycl;
default_selector h;
{
    buffer<int> b(range<1>(size));
    b.set_final_data(v.data());
    {
        cl::sycl::queue q(h);
        {
            auto hostAcc = b.get_access<mode::read_write,
                target::host_buffer>();
            for (auto & : hostAcc) {
                *i = read_data_from_file(...);
            }
        }
        sort(sycl_execution_policy(q), begin(b), end(b));
        transform(sycl_policy, begin(b), end(b), begin(b),
            std::negate<int>());
    }
}
```

- Data is initialized in the host using a host accessor
- After host accessor is done, data is on the device
SYCL/STL interop functions

**Defined in SYCL**

- Buffer constructor taking `begin/end`
- `set_final_data` taking `begin/end`

**Extended by SYCL STL**

- Host and Buffer iterators
- `begin/end` functions

Remember: You need an *accessor* to read data from buffers!
Final Remarks

If you have any questions or comments...

@ sycl@codeplay.com

@ ruyman@codeplay.com

@Ruyk