HPX and GPU-parallelized STL

Marcin Copik

Louisiana State University
Center for Computation and Technology
The STEllAR Group
mcopik@gmail.com

May 13, 2016
Project: "Integrate a C++AMP Kernel with HPX"
Mentor: Hartmut Kaiser
Plan

HPX
Parallelism in C++
  Concepts
GPU in HPX
  Execution
  Data placement
GPU standards for C++

C++AMP
Khronos SYCL
Compilers

Results
Implementation
STREAM benchmark

Goals
What is HPX?

- High Performance ParalleX $^{1,2}$
- Runtime for parallel and distributed applications
- Written purely in C++, with large usage of Boost
- Unified and standard-conforming C++ API

---

$^1$ ParalleX an advanced parallel execution model for scaling-impaired applications - H. Kaiser et al - ICPPW, 2009

What is HPX?
HPX and C++ standard

HPX implements and even extends:

- Concurrency TS, N4107
- Extended async, N3632
- Task block, N4411
- Parallelism TS, N4105
- Executor, N4406

---


5 of 65
HPX and C++ standard

HPX implements and even extends:

- Concurrency TS, N4107
- Extended async, N3632
- Task block, N4411
- Parallelism TS, N4105
- Executor, N4406

Another components

- partitioned vector
- segmented algorithms

---

Plan

HPX
Parallelism in C++
  Concepts
GPU in HPX
  Execution
  Data placement
GPU standards for C++

C++AMP
Khronos SYCL
Compilers

Results
Implementation
STREAM benchmark

Goals
Execution policy

Puts restriction on execution, ensuring thread-safety

Parallelism TS
- sequential
- parallel
- parallel_vector

HPX
- asynchronous sequential
- asynchronous parallel
Execution policy

Extended API for algorithms:

```cpp
template< typename ExecutionPolicy, typename InputIt, typename UnaryFunction >
void for_each(ExecutionPolicy&& policy, InputIt first, InputIt last, UnaryFunction f)
```
Executor

Platform and vendor independent abstraction for launching work

- generic sequential and parallel executor
- core
- NUMA domain
- cluster node
- accelerator
Executor API

Requires only one function:

```cpp
template <typename F>
hpx::future<
    typename hpx::util::result_of<
        typename hpx::util::decay<F>::type()
    >::type>
async_execute(F && f)
{
    return hpx::async(launch::async, std::forward<F>(f));
}
```

Synchronous execution and bulk overload may be provided through `executor_traits`
Algorithm example

```cpp
std::vector<std::size_t> c(n);
std::iota(boost::begin(c), boost::end(c), std::rand());
/** Semantically same as std::for_each **/
hpx::parallel::for_each(hpx::parallel::seq, boost::begin(c), boost::end(c), [](std::size_t & v) { v = 42;});
/** Parallelize for_each **/
hpx::parallel::for_each(hpx::parallel::par, boost::begin(c), boost::end(c), [](std::size_t & v) { v = 43;});
```
Executor parameters

Provide specific launch parameters

- chunk size controls scheduling, similar to OpenMP

Bind executor with parameter

```cpp
hpx::parallel::for_each(
    par.with(hpx::parallel::static_chunk_size(100)),
    ...)
```

Bind executor with tasking and parameter

```cpp
hpx::parallel::for_each(
    par.on(hpx::parallel::task).with(hpx::parallel::
        static_chunk_size(100)),
    ...)
```
Asynchronous execution

Future

- represents result of an unfinished computation
- enables sending off operations to another thread
- TS allows for concurrent composition of different algorithms
- explicit depiction of data dependencies

Compose different operations

```cpp
future<type> f1 = for_each(par_task, ...);
auto f2 = f1.then(
    [](future<type> f1) {
        for_each(par_task, ...);
    }
);
```
Plan

HPX
Parallelism in C++
  Concepts
GPU in HPX
  Execution
  Data placement
GPU standards for C++

C++AMP
Khronos SYCL
Compilers

Results
Implementation
STREAM benchmark

Goals
GPU execution policy

Why a separate policy?

- allows to specialize algorithms behaviour
- explicit offloading of computation to a device
- wraps a default type of executor
GPU execution policy

Why a separate policy?
- allows to specialize algorithms behaviour
- explicit offloading of computation to a device
- wraps a default type of executor

Code does not depend on executor

```cpp
#if defined(HPX_WITH_AMP)
  typedef parallel::gpu_amp_executor executor_type;
#else
  typedef parallel::gpu_sycl_executor executor_type;
#endif
... 
_gpu::executor_type my_exec;
```
GPU executor

- implements functions for synchronous and asynchronous execution
- currently provides interface for data allocation

GPU executors:

- C++AMP
- SYCL
- CUDA
- probably HC in future
Data placement on device

Scheme of execution on GPU:

- transfer data from host to device
- submit kernel
- wait for finish
- transfer data back from device
Data placement on device

Scheme of execution on GPU:
- transfer data from host to device
- submit kernel
- wait for finish
- transfer data back from device

**Solution:** algorithm automatically transfers data to GPU

- user is not aware of data transfer
- algorithms API does not change
Data placement on device

Scheme of execution on GPU:

- transfer data from host to device
- submit kernel
- wait for finish run more kernels
- transfer data back from device
Data placement on device

Scheme of execution on GPU:
- transfer data from host to device
- submit kernel
- wait for finish run more kernels
- transfer data back from device

Solution: algorithm automatically transfers data to GPU

+ user is not aware of data transfer
+ algorithms API does not change
− unnecessary data transfers for operations over the same data
Data placement on device

**Solution:** GPU iterator

- use executor API to place data on GPU
- run many algorithms using iterator defined in executor
- synchronize data on GPU with host when it’s needed

```cpp
std::vector<int> vec(10);
auto buffer = exec.create_buffers(vec.begin(), vec.end());
hpx::parallel::for_each(hpx::parallel::gpu, buffer.begin(), buffer.end(), ...);
buffer.synchronize();
```
Data placement on device

**Solution:** GPU iterator

- use executor API to place data on GPU
- run many algorithms using iterator defined in executor
- synchronize data on GPU with host when it’s needed

**Solution:** GPU iterator

+ optimized data transfer
+ algorithms API does not change
- explicit dependency on a GPU executor
Data placement on device

**Solution:** GPU iterator

- use executor API to place data on GPU
- run many algorithms using iterator defined in executor
- synchronize data on GPU with host when it’s needed

**Solution:** GPU iterator

+ optimized data transfer
+ algorithms API does not change
- explicit dependency on a GPU executor → **GPU-aware data structure**
Plan

HPX
Parallelism in C++
  Concepts
GPU in HPX
  Execution
  Data placement
GPU standards for C++

C++AMP
Khronos SYCL
Compilers

Results
Implementation
STREAM benchmark

Goals
open specification proposed by Microsoft

designed primarily for implementation based on DirectX

allows for scheduling C++ kernels on *accelerators*

current version 1.2, released in March 2013

**warning:** Microsoft specific extensions, code may not be portable
Accelerators

**accelerator**
- abstract view of a computing device (e.g. CPU, GPU)
- provides necessary information, e.g. amount of memory
- standard specifies only two accelerator types: cpu and a default one
- may be used to control on which device place data and run job

**accelerator_view**
- a queue for submitting jobs
- enables synchronization through wait method
- multiple views are safe in a multi-threaded application
Data placement

array

- N-dimensional generic container for data on device
- type restrictions: proper alignment of compound types and at least for 4 bytes per fundamental type
- bound to a specific accelerator

array_view

- cached view of data with an implicit copy
- useful for containers or pointers
Kernel submission

parallel_for_each

- uses separate *extent* structure to specify number of threads created on device
- function executed on GPU has exactly one argument - index specifying location inside thread grid
- call should be synchronous, but may be optimized by compiler and return earlier
- first copy of data out of device enforces synchronization

extent

- creates N-dimensional grid of threads
- dimensionality known at compile time
Kernel submission

Restrictions

- no virtual functions
- no RTTI
- no exceptions
- recursion allowed since AMP 1.2
- functions called on device must be visible at compile time
- keyword `restrict(amp)` must be used on functions executed on device
Kernel submission for iterators

```cpp
std::vector<int> v = { 'G', 'd', 'k', 'k', 'n', 31, 'v', 'n', 'q', 'k', 'c'};
Concurrency::extent<1> extent(v.size());
Concurrency::array<int> data(extent, v.begin(), v.end());

Concurrency::parallel_for_each(extent,
    [&data, lambda](index<1> idx) restrict(amp) {
        lambda(data[idx[0]]);
    });
Concurrency::copy(data, v.begin());
```
Heterogeneous Computing

- modification of C++AMP designed by AMD
- very novel idea, no formal specification yet
- uses concepts and design from AMP, but lifts some restrictions

Changes:

- keyword *restrict* is no longer necessary
- dynamic choice of extent dimensionality
- common address space for both host and device on HSA platforms
SYCL

- proposed by Khronos Group
- brings many concepts known from OpenCL
- version 1.2 of specification released in May 2015
- version 2.2 released in March 2016
- targets devices supporting different versions of OpenCL
Accelerators

- similar to OpenCL in design - platform, context, device, queue
- device selection through a separate selector: default, gpu, cpu, host
- non standard device selection through a custom selector
- kernel submission in a queue
Data placement

buffer

- N-dimensional generic container for data
- type restrictions: C++11 standard layout

buffer accessor

- data accessor on host or device
- doesn’t expose iterators, only index operator
- needs to be captured by lambda executed on device
- **device accessor can be created only in queue code**
Kernel submission for iterators

```cpp
std::vector<int> data{ 'G', 'd', 'k', 'k', 'n', 31, 'v', 'n', 'q', 'k', 'c'};
default_selector selector; queue myQueue(selector);
auto first = data.begin(); std::size_t size = data.size();

/** Create buffer with copy back**/
std::shared_ptr<int> buf_data{new int[size],
    [first, size](int * ptr) {
        std::copy(ptr, ptr + size, first);
        delete[] ptr;
    }
};
std::copy(data.begin(), data.end(), buf_data.get());
buffer<int, 1> buf(buf_data, cl::sycl::range<1>(size));
```
/** Send kernel **/
myQueue.submit([&](handler& cgh) {
    auto ptr = buf.get_access<access::mode::read_write>(cgh);
    auto lambda = [](int & v) { ++v; };
    cgh.parallel_for<class HelloWorld>(range<1>(data.size()),
        [=](id<1> idx) {
            lambda(ptr[idx[0]]);
        });
});
std::vector<int> data{ 'G', 'd', 'k', 'k', 'n', 31, 'v', 'n', 'q', 'k', 'c'};
default_selector selector; queue myQueue(selector);
buffer<int, 1> buf(data.begin(), data.end());
myQueue.submit([&](handler& cgh) {
    auto ptr = buf.get_access<access::mode::read_write>(cgh);
    auto lambda = [](int & v) { ++v; };
    cgh.parallel_for<class HelloWorld>(range<1>(data.size()), [=](id<1> idx) {
        lambda(ptr[idx[0]]);
    });
});
auto host_acc = buf.get_access<access::mode::read, access::target::host_buffer>();
std::copy(host_acc.get_pointer(), host_acc.get_pointer() + buf.get_count(), data.begin());
Kernel restrictions

- no virtual functions
- no exceptions
- no RTTI
- **no recursion**
- functions called on device must be visible at compile time
Kernel name

- two-tier compilation needs to link kernel code and invocation
- name has to be unique across whole program
- breaks the standard API for STL algorithms
- different extensions to C++ may solve this problem

\textsuperscript{5} Khronos’s OpenCL SYCL to support Heterogeneous Devices for C++ - Wong, M. et al. - P0236R0
Kernel name

- two-tier compilation needs to link kernel code and invocation
- name has to be unique across whole program
- breaks the standard API for STL algorithms
- different extensions to C++ may solve this problem

C++ code

```cpp
    cgh.parallel_for_each<class KernelName>(...);
```

---

*Khronos’s OpenCL SYCL to support Heterogeneous Devices for C++* - Wong, M. et al. - P0236R0
Kernel name

Names in template methods

```cpp
template<typename FloatingType>
void solve_pde(vector<FloatingType> & in, vector<FloatingType> & out)
{
    // ...
    cgh.parallel_for_each<class FDSolver>(...);
}

/* ... */
if (user_wants_less_precision)
    solve_pde(float_data, float_result);
else
    solve_pde(double_data, double_result);
```
Kernel name

Names in template methods

template<typename FloatingType>
void solve_pde(vector<FloatingType> & in, vector<FloatingType> & out)
{
    // ...
    cgh.parallel_for_each<class FDSolver>(...);
}

/* ... */
if (user_wants_less_precision)
    solve_pde(float_data, float_result);
else
    solve_pde(double_data, double_result);

error: definition with same mangled name as another definition
Names in template methods

template<
typename FloatingType,
typename SolverName>
void solve_pde(vector<FloatingType> & in, vector<FloatingType> & out)
{
    // ...
    cgh.parallel_for_each<SolverName>(...);
}
/* ... */
if (user_wants_less_precision)
    solve_pde<class FloatSolver>(float_data, float_result);
else
    solve_pde<class DoubleSolver>(double_data, double_result);
Named execution policy

- execution policy contains the name
- use the type of functor if no name is provided
- used in prototype implementation of ParallelSTL done by Khronos

```
struct DefaultKernelName {}

template <class KernelName = DefaultKernelName>
class sycl_execution_policy {
    ...
};
```

---

HCC - Heterogeneous Computing Compiler

- started as Clamp for C++AMP, renamed later to Kalmar
- since November 2015 development supported by AMD
- LLVM-based compiler, two passes over source code
- requires libc++

Frontends
- C++AMP
- HC

Backends
- OpenCL C
- OpenCL SPIR
- HSAIL
- AMD Native GCN ISA
HCC - Heterogeneous Computing Compiler

- started as Clamp for C++AMP, renamed later to Kalmar
- since November 2015 developed supported by AMD
- open source and LLVM-based compiler, two passes over code
- requires libc++

Frontends
- C++AMP
- HC

Backends
- OpenCL-C
- OpenCL-SPIR
- HSAIL
- AMD Native GCN ISA
ComputeCPP

- SYCL device compiler developed by Codeplay
- closed source, LLVM-based compiler
- no official release candidate (yet)
ComputeCPP

- SYCL **device** compiler developed by Codeplay
- closed source, LLVM-based compiler
- no official release candidate (yet)
- one backend: OpenCL SPIR
ComputeCPP

- SYCL `device` compiler developed by Codeplay
- closed source, LLVM-based compiler
- no official release candidate (yet)
- one backend: OpenCL SPIR → no support on NVIDIA GPUs
ComputeCPP

- SYCL **device** compiler developed by Codeplay
- closed source, LLVM-based compiler
- no official release candidate (yet)
- one backend: OpenCL SPIR → **no support on NVIDIA GPUs**

**compute.cpp**
- device code
- `.sycl` header for C++

**cxx**
- host code
- includes kernel header
Plan

HPX
Parallelism in C++
  Concepts
GPU in HPX
  Execution
  Data placement
GPU standards for C++

C++AMP
Khronos SYCL
Compilers

Results
  Implementation
  STREAM benchmark

Goals
Integration with HCC

- HPX needs to be compiled and linked with libc++
- HCC becomes the `CMAKE_CXX_COMPILER`
- expect that it may not always work out of the box

+ easy integration with existing build system
- increased time and memory usage for compilation, even for non-GPU source code
Executor

- two phase compilation requires pseudo-dependencies on targets in CMake
- `CMAKE_CXX_COMPILER` doesn’t change

+ no change in environment - same compiler, same implementation of C++
+ it is possible to apply new compiler only to files with GPU-code
  - may be tricky to get it right with different build systems
Implementation of for_each_n

Current parallel implementation:

```
template <typename ExPolicy, typename F, 
    typename Proj = util::projection_identity>
static typename detail::algorithm_result<ExPolicy, Iter>::type
    parallel(ExPolicy policy, Iter first, std::size_t count, 
        F && f, Proj && proj)
{
    if (count != 0)
    {
        return foreach_n_partitioner<ExPolicy>::call(policy, 
            first, count, [f, proj](Iter begin, std::size_t size) {
            loop_n(begin, size, [=](Iter const & curr) {
                invoke(f, invoke(proj, *curr));
            }); });
    }
    return detail::algorithm_result<ExPolicy, Iter>::get( 
        std::move(first));
}
Implementation of for_each_n

How do we implement synchronous bulk execution?

```cpp
static typename detail::bulk_execute_result<F, Shape>::type
bulk_execute(F && f, Shape const & shape)
{
    // Shape elements are tuples with iterator, data count and
    // chunk size
    typedef typename Shape::value_type tuple_t;
    for (auto const & elem : shape) {

        auto iter = hpx::util::get<0>(elem);
        std::size_t data_count = hpx::util::get<1>(elem);
        std::size_t chunk_size = hpx::util::get<2>(elem);

        std::size_t threads_to_run = data_count / chunk_size;
        std::size_t last_thread_chunk = data_count -
        (threads_to_run - 1)*chunk_size;
    }
```
Implementation of for_each

How do we implement it?

```cpp
Concurrency::extent<1> e(threads_to_run);
Concurrency::parallel_for_each(e, [=](Concurrency::index<1> idx) restrict(amp)
{
    std::size_t part_size =
    idx[0] != static_cast<int>(threads_to_run - 1) ?
        chunk_size : last_thread_chunk;
    auto it = iter;
    it.advance(idx[0]*chunk_size);
    tuple_t tuple(it, 0, part_size);
    f(tuple);
});
accelerator_view.wait();
```
Implementation of for_each_n

How do we call it?

```cpp
std::vector<int> c(n);
std::iota(boost::begin(c), boost::end(c), std::rand());

auto buffer = hpx::parallel::gpu.executor().create_buffers(c.
    begin(), c.end());

hpx::parallel::for_each(hpx::parallel::gpu,
    buffer.begin(), buffer.end(),
    [](int& v) {
        v = 400;
    });

buffer.sync();
```
Implementation of transform

What is an unary transform?

typedef hpx::util::zip_iterator<FwdIter, output_iterator> zip_iterator;
typedef typename zip_iterator::reference reference;
for_each_n<zip_iterator>().call(policy,
    hpx::util::make_zip_iterator(first, dest),
    std::distance(first, last),
    [f, proj](reference t)
    {
        using hpx::util::get; get<1>(t) = f(get<0>(t));
    });

What is a binary transform?
Same idea, just three iterators.
Implementation of transform

Wouldn’t it be great if it worked immediately on GPUs?

gpu_AMP_executor?
Implementation of transform

Wouldn’t it be great if it worked immediately on GPUs?

gpu_amp_executor?
yes, I can do that!
Implementation of transform

Wouldn’t it be great if it worked immediately on GPUs?

gpu_sycl_executor?
Implementation of transform

Wouldn’t it be great if it worked immediately on GPUs?

```
gpu_sycl_executor?
```

error: cannot capture object ptr of type ’class cl::sycl::accessor[...]’ in a SYCL kernel, because it is a non standard-layout type
Implementation of transform

Wouldn’t it be great if it worked immediately on GPUs?

gpu_sycl_executors?

error: can not capture object ptr of type ’class cl::sycl::accessor[...]' in a SYCL kernel, because it is a non standard-layout type
error: class std::tuple is not standard layout, because multiple classes among its base classes declare non-static fields
Executor parameters

Chunk size

- parallel algorithm exposes dynamic, static, guided, auto
- most of them doesn't make sense on GPU, where there is a certain overhead of launching small jobs
- GPU executor takes a static chunk size

Also:

- kernel name
- tiling size (local work size) in future
Name as an executor parameter

- name is still tied to an executor
- same API calls for both AMP and SYCL

```cpp
#include <hpx/include/parallel_executor_parameters.hpp>

hpx::parallel::transform(hpx::parallel::gpu.with(
    hpx::parallel::static_chunk_size(32),
    hpx::parallel::kernel_name<class Add>(()), ... ));
```
Naming the kernel

- light wrapper around the kernel
- name is tied directly to the executed function
- not applicable for algorithms without user-defined operator

```cpp
#include <hpx/parallel/executors/parallel_executor_parameters.hpp>
#include <hpx/parallel/kernel_name.hpp>

hpx::parallel::for_each(
    hpx::parallel::gpu.with(hpx::parallel::kernel_name<class FalseName>(),
    d.begin(), d.end(),
    hpx::parallel::make_kernel<class CorrectName>([](int & v) {
        v = 42;
    }));
```
Known problems

HCC
- problems with correct linking of kernel (HPX only)
- known bugs in OpenCL backend which most likely won’t be fixed

ComputeCPP
- incorrect capture of const integers in device lambda (HPX only)
- unfriendly build scripts
STREAM benchmark

STREAM benchmark consists of:

1 scalar k, 3 input arrays a, b, c and 4 operations

---

STREAM benchmark consists of:

1 scalar k, 3 input arrays a, b, c and 4 operations

- **copy**
  \[ c = a \]
STREAM benchmark consists of:

1 scalar k, 3 input arrays a, b, c and 4 operations

- **copy**
  \[ c = a \]

- **scale**
  \[ b = k \times c \]
STREAM benchmark consists of:

1 scalar k, 3 input arrays a, b, c and 4 operations

- **copy**
  
  \[ c = a \]

- **scale**
  
  \[ b = k \times c \]

- **add**
  
  \[ c = b + a \]
STREAM benchmark consists of:

1 scalar k, 3 input arrays $a, b, c$ and 4 operations

- **copy**
  
  $c = a$

- **scale**
  
  $b = k \times c$

- **add**
  
  $c = b + a$

- **triad**
  
  $a = b + k \times c$

---

Benchmarking hardware

C++AMP

- **GPU**: AMD Radeon R9 Fury Nano
- **OpenCL**: AMD APP SDK 3.02
- **HSA**: AMD ROCm 1.0

Khronos SYCL

- **GPU**: AMD Radeon R9 Fury Nano
- **ComputeCPP**: 15.10
- **OpenCL**: AMD APP SDK 3.02

GPU-STREAM has been used to measure OpenCL and HSA performance: [https://github.com/UoB-HPC/GPU-STREAM](https://github.com/UoB-HPC/GPU-STREAM)
How does AMP perform?

Performance in STREAM benchmark

Bandwidth [MB/s]

Number of double-precision floating point elements

- C++AMP/HSA
- HSA
- C++AMP/OpenCL
- OpenCL
How does AMP perform?

Performance in STREAM benchmark - CUDA, Tesla K80

Bandwidth [MB/s]

C++AMP/OpenCL
CUDA

Number of double-precision floating point elements
How does SYCL perform?

Performance in STREAM benchmark - average bandwidth

Bandwidth [MB/s]

Number of double-precision floating point elements

SYCL/OpenCL SPIR
OpenCL
Overhead

HCC

- Compilation of HPX is approximately \(2.4x\) slower
- Compilation of benchmark example increased from 20 to 48 seconds, \(2.4x\) slower
- Peak memory usage of compiler and binary size are both comparable

ComputeCPP

- For benchmark example, the overhead of device is compiler is 12 seconds to 20 seconds required by g++, slowing the compilation \(1.6\) times.
- Peak memory usage of compiler and binary size are both comparable
Plan

HPX
Parallelism in C++
  Concepts
GPU in HPX
  Execution
  Data placement
GPU standards for C++

C++AMP
Khronos SYCL
Compilers

Results
  Implementation
  STREAM benchmark

Goals
Data placement revised

How and when to place data?

- in the current implementation algorithm is responsible for data allocation
- different types of memory on GPUs ⇒ executor should know \textit{where} execute kernel, not \textit{how} to place data
- STL: algorithms and containers ⇒ container with special allocator
- we want to support multiple GPUs ⇒ a partitioned vector with segmented algorithms
Algorithms
hp::compute

- ongoing work to provide standard compliant GPU algorithms in an "STL way"
- includes AMP/SYCL backends presented here
- includes existing and developed support for CUDA and OpenCL
- focused on distributed computing
Thanks for your attention

mcopik@gmail.com

github.com/mcopik/