Using SYCL as an Implementation Framework for HPX.Compute

Marcin Copik [1]
Hartmut Kaiser [2]

[1] RWTH Aachen University
mcopik@gmail.com
[2] Louisiana State University
Center for Computation and Technology
The STE||AR Group

May 16, 2017

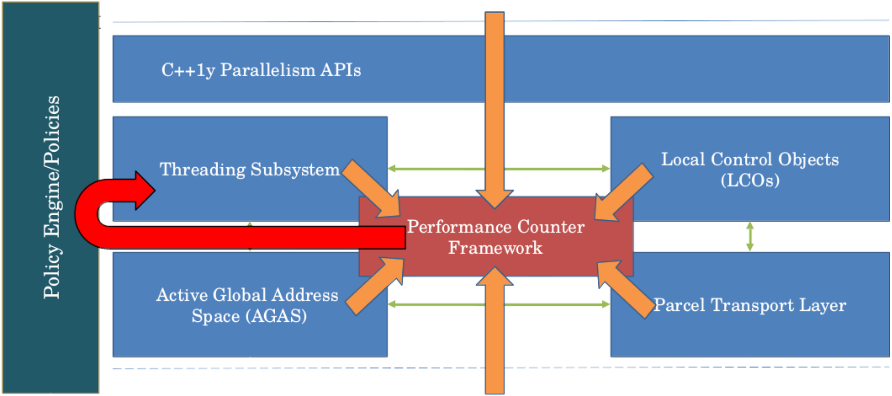
Plan

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
[2] *A Task Based Programming Model in a Global Address Space* - H. Kaiser et al - PGAS, 2014

What is HPX?


Policy Engine/Policies
C++1y Parallelism APIs
Threading Subsystem
Local Control Objects (LCOs)
Performance Counter Framework
Active Global Address Space (AGAS)
Parcel Transport Layer

HPX and C++ standard

HPX implements and even extends:

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

[3] *Segmented Iterators and Hierarchical Algorithms*-Austern, Matthew H. - Generic Programming: International Seminar on Generic Programming, 2000

HPX and C++ standard

HPX implements and even extends:

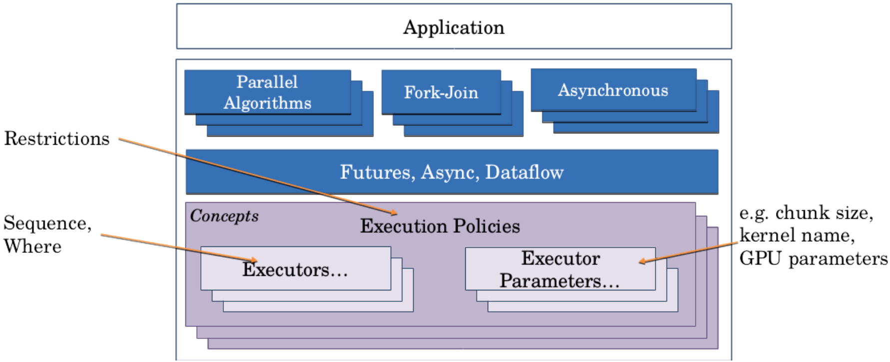
- Concurrency TS, N4107
- Extended async, N3632
- Task block, N4411
- **Parallelism TS, N4105**
- **Executor, N4406**

Another components

- partitioned vector
- segmented algorithms[3]

[3] *Segmented Iterators and Hierarchical Algorithms*-Austern, Matthew H. - Generic Programming: International Seminar on Generic Programming, 2000

Overview


Application
Parallel Algorithms
Fork-Join
Asynchronous
Restrictions
Futures, Async, Dataflow
Concepts
Execution Policies
Sequence, Where
Executors...
Executor Parameters...
e.g. chunk size, kernel name, GPU parameters

Execution policy

Puts restriction on execution, ensuring thread-safety

C++17

- sequential
- parallel
- parallel unsequenced

HPX

- asynchronous sequential
- asynchronous parallel

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

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

Plan

HPX.Compute

- a unified model for heterogeneous programming
- platform and vendor independent
- interface based on C++17 and further extensions to C++ standard

Backends for:

- host
- CUDA
- HCC[4]
- SYCL

[4] unfinished

HPX.Compute

- a unified model for heterogeneous programming
- platform and vendor independent
- interface based on C++17 and further extensions to C++ standard

Three major concepts:

- target
- allocator
- executor

[4] unfinished

Target

- an abstract type expressing data locality and place of execution
- variety of represented hardware requires a simplified interface

Target interface:

```
//Blocks until target is ready
void synchronize();
//Future is ready when all tasks allocated on target have been
    finished
hpx::future<void> get_future() const;
```

Target

- an abstract type expressing data locality and place of execution
- variety of represented hardware requires a simplified interface

SYCL implementation of target

- communicaties with device through `sycl::queue`
- multiple targets may represent the same device
- requires additional measures for asynchronous communication

Allocator

- allocate and deallocate larger chunks of data on target
- data allocation is trivial on backends where memory is accessed with pointers (host, CUDA)

SYCL implementation of allocator

- create `sycl::buffer` objects
- not possible to tie a buffer to given device

Executor

- execute code on device indicated by data location
- usual GPU-related restrictions on allowed C++ operations
- marking device functions not required

Interface of an executor

```
struct default_executor : hpx::parallel::executor_tag
{
  template <typename F, typename Shape, typename ... Ts>
  void bulk_launch(F && f, Shape const& shape, Ts &&... ts)
      const;

  template <typename F, typename Shape, typename ... Ts>
  std::vector<hpx::future<void>> bulk_async_execute(F && f,
      Shape const& shape, Ts &&... ts) const;
};
```

Plan

HPX
- Concepts

HPX.Compute

Challenges

Benchmarking

Summary
- Goals

Device accessors

Capturing data buffers in SYCL

- a host iterator can only store `sycl::buffer` and position
- a separate device iterator has to be created in command group scope
- `sycl::global_ptr` represents an iterator type on device, but `std::iterator_traits` specialization or related typedefs are missing in SYCL standard

Comparision with other backends:

- an additional static conversion function is necessary
- distinct iterator types on host and device
- requires templated function objects or C++14 generic lambda

Data movement

Problem: copy data from a device to a given memory block on host, with a selection of an offset and size?

- `host_accessor` - an intermediate copy in SYCL runtime, no flexibility, may lead to deadlocks if a host accessor is not destroyed
- `set_final_data` - applicable only for buffer destruction, no flexibility
- range-based subbufer - can emulate offset and size for `host_accessor`
- `map_allocator` - data is copied to a pointer defined by the SYCL user, but it can not be changed

Further issues

- no ability to synchronize with data transfer

Data movement

Suggested extension for SYCL

```
// copy all contents of buffer
template<typename T, int N, typename OutIter>
sycl::event copy(const sycl::buffer<T,N> & src, OutIter dest);

// copy range [begin, end) to buffer, fully replacing its
    contents
template<typename InIter, T, int N>
sycl::event copy(InIter begin, InIter end, sycl::buffer<T,N> &
     dest);
```

Data movement

Suggested extension for SYCL

```
// write range to buffer starting at 'pos'
template<typename T, int N, typename InIter>
sycl::event sycl::buffer<T,N>::write(
  std::size_t pos, InIter begin, InIter end
);

// read 'size' elements starting at 'pos'
template<typename T, int N, typename OutIter>
sycl::event sycl::buffer<T,N>::read(
  size_t pos, size_t size, OutIter dest
);
```

Asynchronous execution

What SYCL offers for synchronization?

- blocking wait for tasks in queue
- blocking wait for enqueued kernels with `sycl::event`
- SYCL API does not cover OpenCL callbacks

Competing solutions

- stream callbacks in CUDA
- an extended future in C++AMP/HCC

Asynchronous execution

Use SYCL-OpenCL interoperability for callbacks

```
// future_data is a shared state of hpx::future
cl::sycl::queue queue = ...;
future_data * ptr = ...;
cl_event marker;
clEnqueueMarkerWithWaitList(queue.get(), 0, nullptr, &marker);
clSetEventCallback(marker, CL_COMPLETE,
  [](cl_event, cl_int, void * ptr) {
    marker_callback(static_cast<future_data*>(ptr));
  }, ptr);
```

Downside

- not applicable for SYCL host device

Non-standard layout datatypes

An example: standard C++ tuple

- common `std::tuple` implementations, such as in libstdc++ or libc++, are not C++11 standard layout due to multiple inheritance
- adding a non-standard implementation requires complex changes in existing codebase

Approaches for other types

- refactor current solution to be C++ standard layout
- manually deconstruct the object and construct again in kernel scope
- add serialization and deserialization interface to problematic types
- automatic serialization by the compiler - technique used in HCC

Kernel naming

- two-tier compilation needs to link kernel code and invocation
- name has to be unique
- breaks the standard API for STL algorithms
- different extensions to C++ may solve this problem[5]

[5] *Khronos's OpenCL SYCL to support Heterogeneous Devices for C++* - Wong, M. et al. - P0236R0

Named execution policy

- execution policy contains the name
- use the type of function object if no name is provided
- used in ParallelSTL project[6]

A SYCL named execution policy

```
struct DefaultKernelName {};

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

[6] https://github.com/KhronosGroup/SyclParallelSTL/

Named execution policy

- execution policy contains the name
- use the type of function object if no name is provided
- used in ParallelSTL project[6]

Cons:

- no logical connection between execution policy and kernel name
- duplicating `std::par` execution policy

[6] https://github.com/KhronosGroup/SyclParallelSTL/

Named execution policy

Our solution: executor parameters

- an HPX extension to proposed concepts for executors
- a set of configuration options to control execution
- control settings which are independent from the actual executor type
- example: OpenMP-like chunk sizes

Pass kernel name as a parameter

```
// uses default executor: par
hpx::parallel::for_each(
  hpx::parallel::par.with(
    hpx::parallel::kernel_name<class Name>()
  ),
  ...
);
```

Plan

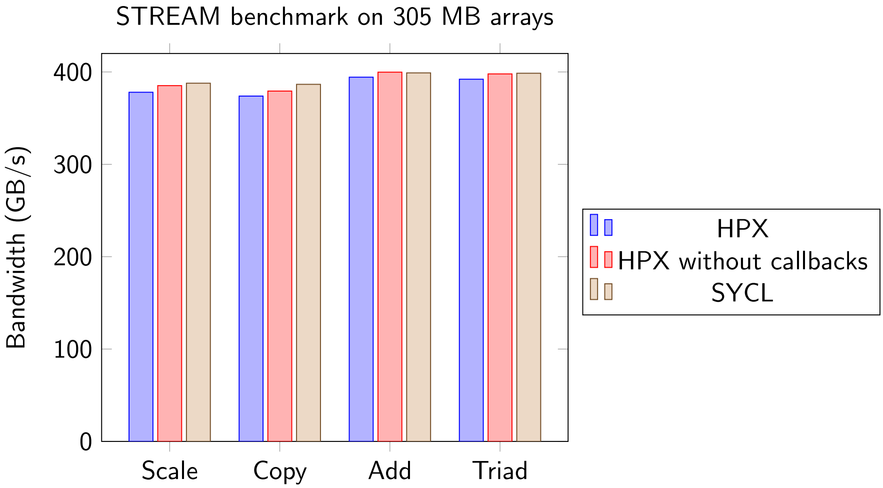
Benchmarking hardware for STREAM

Khronos SYCL

- **GPU:** AMD Radeon R9 Fury Nano
- **ComputeCPP:** CommunityEdition-0.1.1
- **OpenCL:** AMD APP SDK 2.9

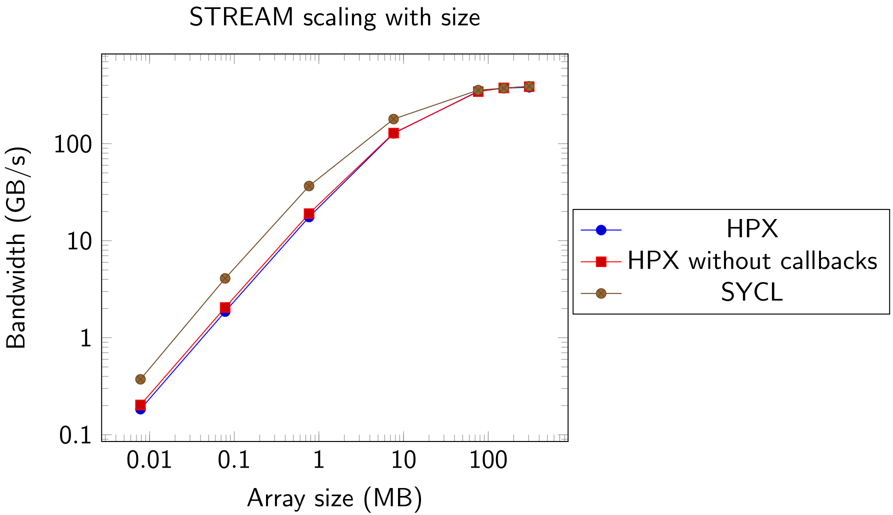
GPU-STREAM has been used to measure SYCL performance:
`https://github.com/UoB-HPC/GPU-STREAM`

STREAM



STREAM benchmark on 305 MB arrays
Bandwidth (GB/s)
400
300
200
100
0
Scale
Copy
Add
Triad
HPX
HPX without callbacks
SYCL

STREAM



STREAM scaling with size
Bandwidth (GB/s)
100
10
1
0.1
0.01
0.1
1
10
100
Array size (MB)
HPX
HPX without callbacks
SYCL

Plan

Summary

The Good

- performance and capabilities comparable with competing standards
- no requirement of marking functions capable of running on a device
- previous experiments revealed that an overhead of ComputeCpp, an offline device compiler for SYCL, is not severe during build process

Summary

The Bad

- kernel names appearing in standard interface
- troublesome capture of complex types storing SYCL buffers
- lack of explicit data movement
- limited support for SPIR on modern GPUs

Summary

The Ugly

- asynchronous callbacks work but with a slight overhead
- SYCL pointer types can not be treated as iterators
- troublesome capture of non-standard layout types

Future

Goals

- demonstrate a complex problem solved over host and GPU with our model and STL algorithms
- extend implementation with more algorithms

Challenges

- how to express on-chip/local memory through our model?
- try to reduce overhead for shorter kernels

Thanks for your attention

mcopik@gmail.com