Address Space Inference using ComputeCpp

Peter Žužek, ComputeCpp Product Owner

ISC HPC, Portable Heterogeneous Programming with SYCL (PHPS22)

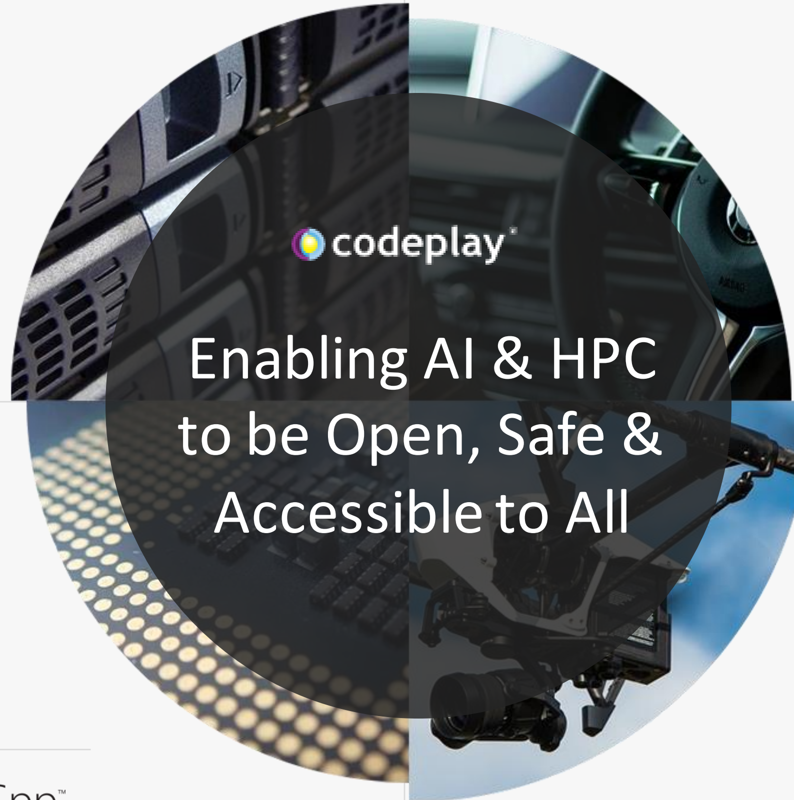
May 31st 2022

Company

Leaders in enabling high-performance software solutions for new AI processing systems

Enabling the toughest processors with tools and middleware based on open standards

Established 2002 in Scotland with ~80 employees



Partners

intel, BROADCOM, SYNOPSYS, CEVA, Imagination, RENESAS, KMC Kyoto Microcomputer Co., Ltd., NSI-TEXE, Berkeley Lab, Oak Ridge National Laboratory, Argonne National Laboratory

And many more!

Products

Integrates all the industry standard technologies needed to support a very wide range of AI and HPC

The heart of Codeplay's compute technology enabling OpenCL™, SPIR-V™, HSA™ and Vulkan™

C++ platform via the SYCL™ open standard, enabling vision & machine learning e.g. TensorFlow™

Markets

High Performance Compute (HPC)
Automotive ADAS, IoT, Cloud Compute
Smartphones & Tablets
Medical & Industrial

Technologies: Artificial Intelligence
Vision Processing
Machine Learning
Big Data Compute

© 2022 Codeplay Software Ltd.

Overview

- SYCL basics
- How to resolve address spaces
  - SYCL 1.2.1
  - SYCL 2020
- Generics
- Address Space Inference
- ComputeCpp device compiler

© 2022 Codeplay Software Ltd.

ComputeCpp 2.10 available

https://developer.codeplay.com

- Many SYCL 2020 features (subgroups algorithms, default placeholders, reversed ranged accessor iterators, ...)
- Big improvements to experimental device compiler (LLVM 15, faster compilation, better address space inference, ...)

https://developer.codeplay.com/products/computecpp/ce/guides/release-notes

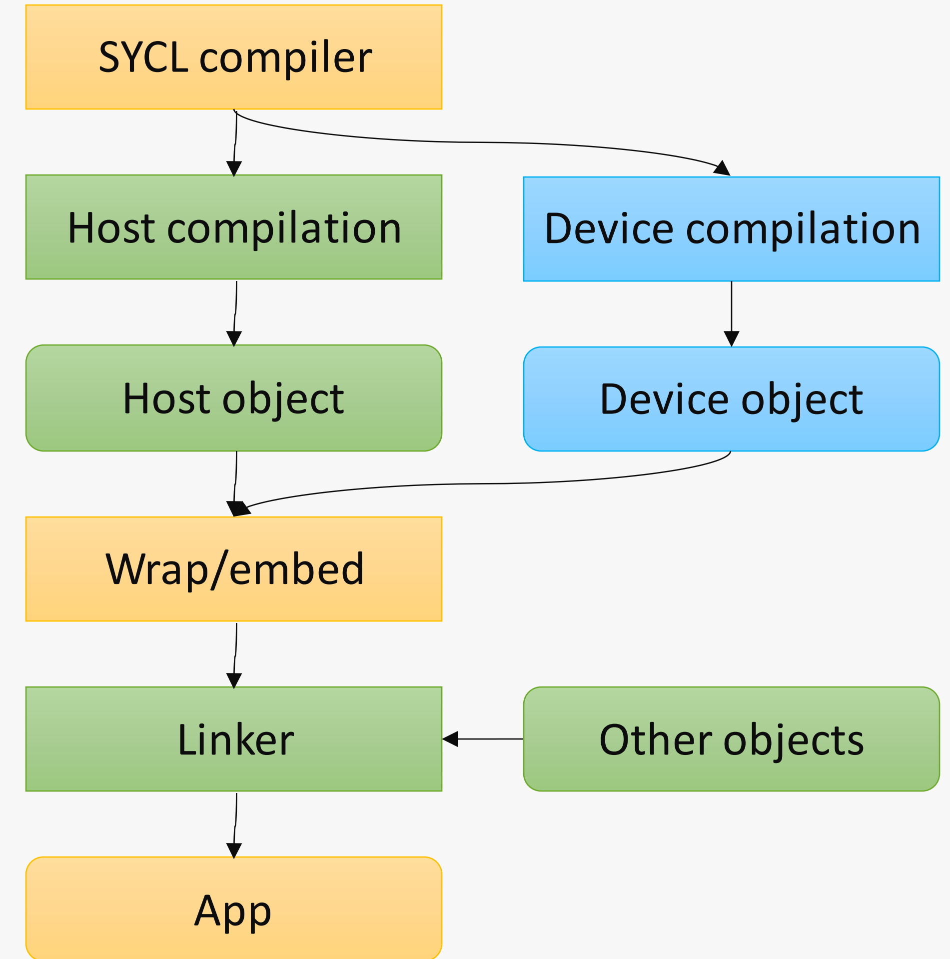
© 2022 Codeplay Software Ltd.

SYCL basics

© 2022 Codeplay Software Ltd.

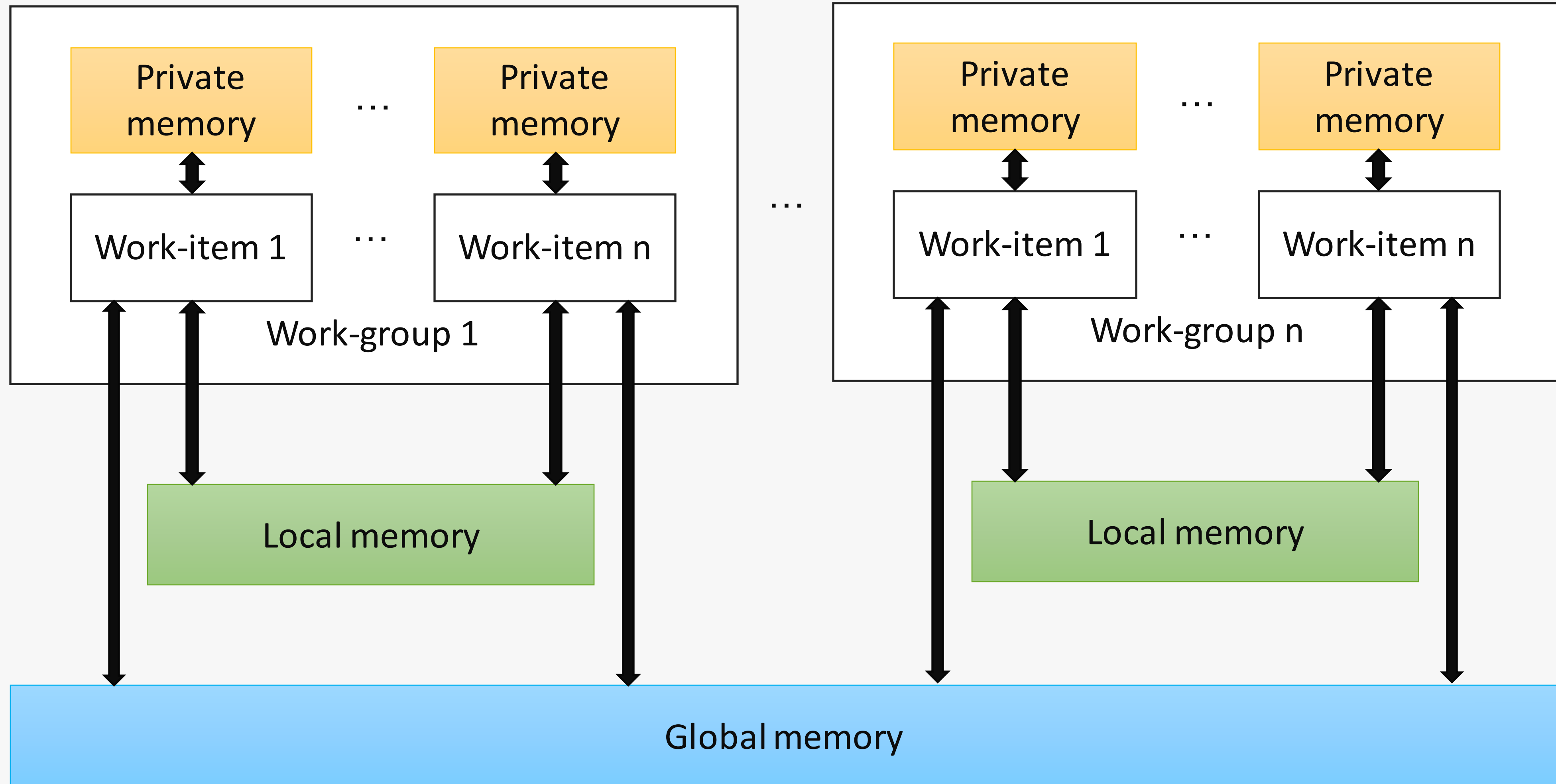
SYCL compilation model

```
#include <sycl/sycl.hpp>

using namespace sycl;

int main() {
  queue q;
  size_t numElems = 1024;

  int* ptr = sycl::malloc_shared<int>(numElems, q);

  q.parallel_for(numElems, [=](id<1> idx) {
    ptr[idx.get(0)] = idx.get(0);  // Kernel code
  });
  q.wait();
}
```



© 2022 Codeplay Software Ltd.

Memory model



© 2022 Codeplay Software Ltd.

Address Space Rules

© 2022 Codeplay Software Ltd.

Address spaces in C++

- Address spaces mark pointers in disjoint memory
- SYCL uses standard C++
  - But address spaces don't exist in standard C++
  - OpenCL: `__global`, `__local`, `__private`, …
- Need a way to determine address spaces within the rules of standard C++
  - SYCL user code should avoid writing them
  - Some pointers can be marked internally
    - But this is not enough
  - Device offloading part of SYCL compiler can do magic behind the scenes

© 2022 Codeplay Software Ltd.

Call graph duplication

- Functions get duplicated for a particular address space
  - Recursive process

```
void prepare_work(int*);



void do_work() {
  __global int* i = get_global();
  prepare_work(i);
  __local int* j = get_local();
  prepare_work(j);
}
```

```
void prepare_work(int*);
void prepare_work(__global int*);
void prepare_work(__local int*);

void do_work() {
  __global int* i = get_global();
  prepare_work(i);
  __local int* j = get_local();
  prepare_work(j);
}
```


© 2022 Codeplay Software Ltd.

SYCL 1.2.1: Deduction

- Address spaces treated as a language extension
  - As a type qualifier
  - Use initial values from some types and conditions
  - Use deduction rules
- Greedy approach
  - Address space deduced when pointer declared

```
auto globalPtr = globalAcc.get_pointer().get(); // __global

int* ptr; // Default to __private
ptr = globalAcc.get_pointer().get(); // Conflict
```

- `multi_ptr` can be used as a workaround

© 2022 Codeplay Software Ltd.

Deduction drawbacks

- Interferes with C++ semantic analysis
  - Creates inconsistencies in template deduction and instantiation in user code
  - compute++ requires modifications to Clang
- Very limited subset of code accepted
  - Does not play well with USM (`usm_wrapper` in ComputeCpp)
- Rules are ambiguous in some places

© 2022 Codeplay Software Ltd.

SYCL 2020: New approach

- Motivation
  - Put fewer restraints on user
  - Allow support for USM
- Main change: don’t always default to private
  - Resolve address space based on how a pointer is used
  - Better support for templates
  - Allows struct duplication
- Two ways to achieve this
  - Generic pointers
  - Address Space Inference

© 2022 Codeplay Software Ltd.

Generics

© 2022 Codeplay Software Ltd.

Generic address space

- Some compilers place unannotated pointers into generic address space
  - No special qualifier needed
  - Additional instructions inserted to resolve the pointer

```
auto globalPtr = get_global();    // __global
auto localPtr = get_local();      // __local
auto privatePtr = get_private();  // __private

int* ptr;
ptr = globalPtr;  // legal
ptr = localPtr;   // legal
ptr = privatePtr; // legal

// The other way round requires an explicit address space cast
```


© 2022 Codeplay Software Ltd.

Drawbacks of generics

- Runtime overhead
  - Observed up to 10% performance hit in internal benchmarks
- Target device must support them
  - Not in OpenCL 1.2 or 3.0
  - Embedded and automotive devices usually don’t support generics
  - Need well-defined address spaces

© 2022 Codeplay Software Ltd.

Address Space Inference

© 2022 Codeplay Software Ltd.

Inference instead of deduction

- Run extra LLVM pass
  - When address space is determined (solved) for a type, propagate it back to all types related to the solved one
    - Type unification based on usage
    - Can detect mismatches

© 2022 Codeplay Software Ltd.

Inference example

```
int* prepare_work(int* a, int* b);

<A, B, C>
int C* prepare_work(int A* a, int B* b);

<A, B, C>
int C* prepare_work(int A* a, int B* b)
{
  return (*a % 2) ? a : b;
}

<A>
int A* prepare_work(int A* a, int A* b)
{
  return (*a % 2) ? a : b;
}
```

```
<A>
void do_work(int A* a,
             __global* b) {
  int* t; // int T*
  t = prepare_work(a, b);
}

void do_work(int __global* a,
             int __global* b) {
  int __global* t;
  t = prepare_work(a, b);
}
```


© 2022 Codeplay Software Ltd.

Inference benefits

- Lets developers target a wider range of more power efficient hardware compared to generics
  - Can target many embedded devices
  - No overhead from determining address spaces at runtime
  - Unlocks more optimization opportunities compared to generics
    - Alias analysis
    - Interprocedural (optimizations don't affect result)
- Does not interfere with C++ type system
  - Accepts a lot more code than address space deduction
  - Enables USM and struct duplication

© 2022 Codeplay Software Ltd.

Drawbacks of inference

- Does not support as much code as generics do
  - Still covers a lot more than the deduction approach
  - Should be enough for the vast majority of cases
  - Inference can improve
- Some breaking changes compared to current compiler
  - But more in line with other SYCL implementations

© 2022 Codeplay Software Ltd.

Future direction

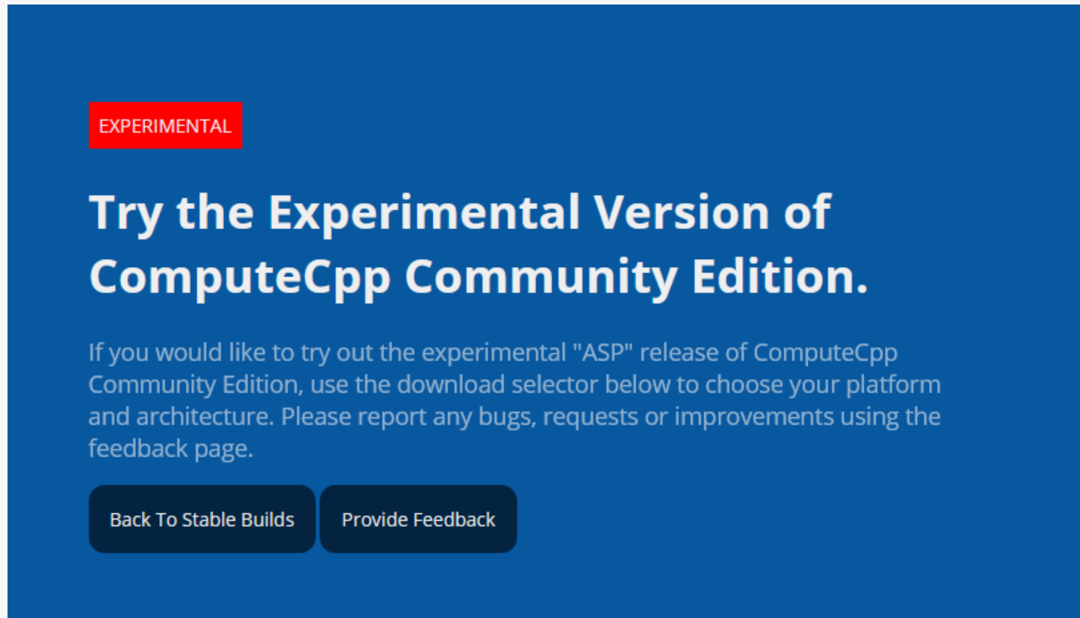
- Improve address space inference implementation
- Improve SYCL specification
- Hybrid inference/generics approach?

© 2022 Codeplay Software Ltd.

ComputeCpp device compiler

© 2022 Codeplay Software Ltd.

Experimental Device Compiler



https://developer.codeplay.com/experimental

- Implements new Address Space Inference
- Will follow LLVM upstream closely
- Will become default compiler
- ComputeCpp 2.10 ships very stable experimental compiler
  - Still some limitations on user code
  - Error reporting needs improvements

Still experimental, please try it and report any issues

© 2022 Codeplay Software Ltd.

Exclusive Features: Experimental Compiler

- Unified Shared Memory
  - Used to require usm_wrapper class, no longer needed
    - Most other functionality available since 2.2
  - Relies heavily on Address Space Inference
- Unnamed kernel lambdas
  - Host compiler must support __builtin_sycl_unique_stable_name
    - Must match device compiler behavior
    - Works best with –sycl-driver switch
- Customer specific SPIR-V extensions

```
auto ptr = sycl::malloc_device<float>(count, testQueue);
testQueue.submit([&](handler& cgh) {
  cgh.parallel_for(range{count}, [=](auto itemID) {
    auto i = itemID.get_linear_id();
    ptr[i] = static_cast<float>(i);         }); });
```


© 2022 Codeplay Software Ltd.

ComputeCpp capabilities

- Profiling support
- Custom instructions
  - `[[computecpp::builtin]]`
- Buffer creation policies
- Offline kernel compilation
- Multiple device targets in single binary
- Checking accessor bounds
- Host device
- Configuration file
- Kernel performance inspector
- computecpp_info

```
********************************************************************************

ComputeCpp Info (RC 2.8.0 2021/12/06)

SYCL 1.2.1 revision 3

********************************************************************************


Device Info:

Discovered 1 devices matching:
  platform    : <any>
  device type : <any>

--------------------------------------------------------------------------------
Device 0:

  Device is supported                     : UNTESTED - Untested OS
  Bitcode targets                         : spir64 spirv64
  CL_DEVICE_NAME                          : Intel(R) Core(TM) i7-1065G7 CPU @ 1.30GHz
  CL_DEVICE_VENDOR                        : Intel(R) Corporation
  CL_DRIVER_VERSION                       : 2020.10.6.0.04
  CL_DEVICE_TYPE                          : CL_DEVICE_TYPE_CPU

********************************************************************************
```


© 2022 Codeplay Software Ltd.

Download ComputeCpp Today

https://developer.codeplay.com

https://developer.codeplay.com/experimental

- Download the latest releases
- The brand new compiler is available at the experimental link
- Remember to give us your feedback

sycl@codeplay.com

© 2022 Codeplay Software Ltd.

We're Hiring!

codeplay.com/careers/

Enabling AI to be Open, Safe & Accessible to All

@codeplaysoft

info@codeplay.com

codeplay.com

Template instantiation changes

```
template <typename T>
void act(T *t) {
  t[0] = 1.0;
};
template<>
void act(float __attribute__((opencl_global))* t) {
  t[0] = 2.0;
}
...
// buf points to `float value`
accessor acc{buf, cgh};
cgh.single_task<class kernel>([=]() {
  // get() returns `__global float*`
  float* as_pointer = acc.get_multi_ptr().get();
  act(as_pointer);
});
...
std::cout << "value: " << value << "\n";
```

- Current compiler prints 2.0
  - `as_pointer` deduced to `__global float*`
- Experimental compiler prints 1.0
  - Generic template selected before inference happens

© 2022 Codeplay Software Ltd.