codeplay®
Connecting AI to Silicon

Improved address space inference for SYCL programs

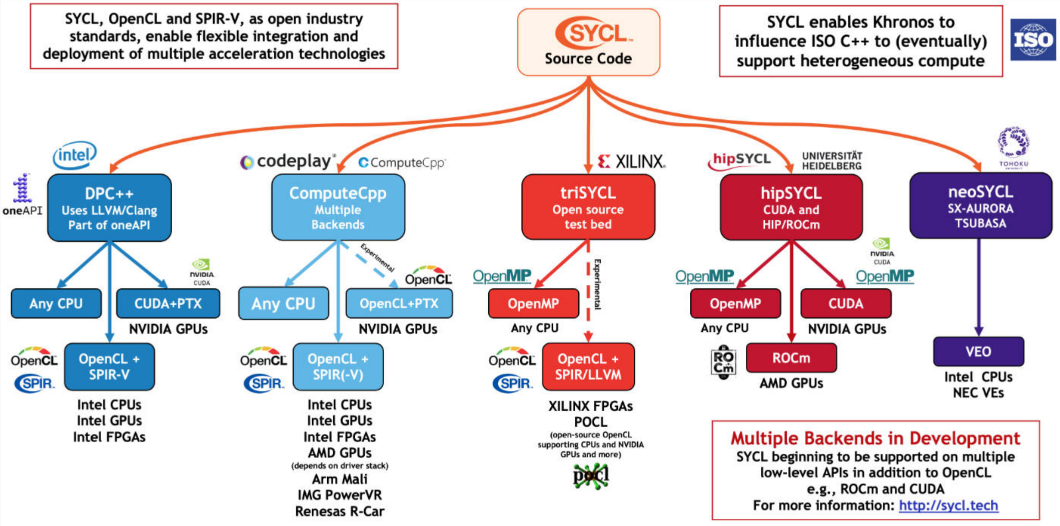
Ross Brunton Victor Lomüller

May 2, 2022

Outline

- General presentations
  - SYCL
  - Address space
- Address space inference in SYCL 1.2.1 and 2020
- New approach
- Conclusion

SYCL

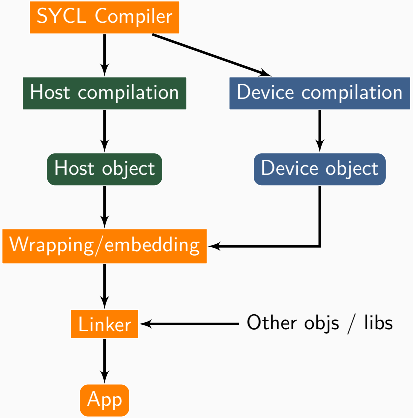

SYCL, OpenCL and SPIR-V, as open industry standards, enable flexible integration and deployment of multiple acceleration technologies
SYCL Source Code
SYCL enables Khronos to influence ISO C++ to (eventually) support heterogeneous compute
ISO
intel
oneAPI
DPC++
Uses LLVM/Clang
Part of oneAPI
Any CPU
CUDA+PTX
NVIDIA CUDA
NVIDIA GPUs
OpenCL
SPIR
OpenCL + SPIR-V
Intel CPUs
Intel GPUs
Intel FPGAs
codeplay
ComputeCpp
ComputeCpp
Multiple Backends
Experimental
OpenCL
Any CPU
OpenCL+PTX
NVIDIA GPUs
OpenCL
SPIR
OpenCL + SPIR(-V)
Intel CPUs
Intel GPUs
Intel FPGAs
AMD GPUs
(depends on driver stack)
Arm Mali
IMG PowerVR
Renesas R-Car
XILINX
triSYCL
Open source test bed
OpenMP
Experimental
OpenMP
Any CPU
OpenCL
SPIR
OpenCL + SPIR/LLVM
XILINX FPGAs
POCL
(open-source OpenCL supporting CPUs and NVIDIA GPUs and more)
pocl
hipSYCL
UNIVERSITÄT HEIDELBERG
hipSYCL
CUDA and HIP/ROCm
NVIDIA CUDA
OpenMP
OpenMP
OpenMP
CUDA
Any CPU
NVIDIA GPUs
ROCm
ROCm
AMD GPUs
TOHOKU
neoSYCL
SX-AURORA TSUBASA
VEO
Intel CPUs
NEC VEs
Multiple Backends in Development
SYCL beginning to be supported on multiple low-level APIs in addition to OpenCL
e.g., ROCm and CUDA
For more information: http://sycl.tech

SYCL

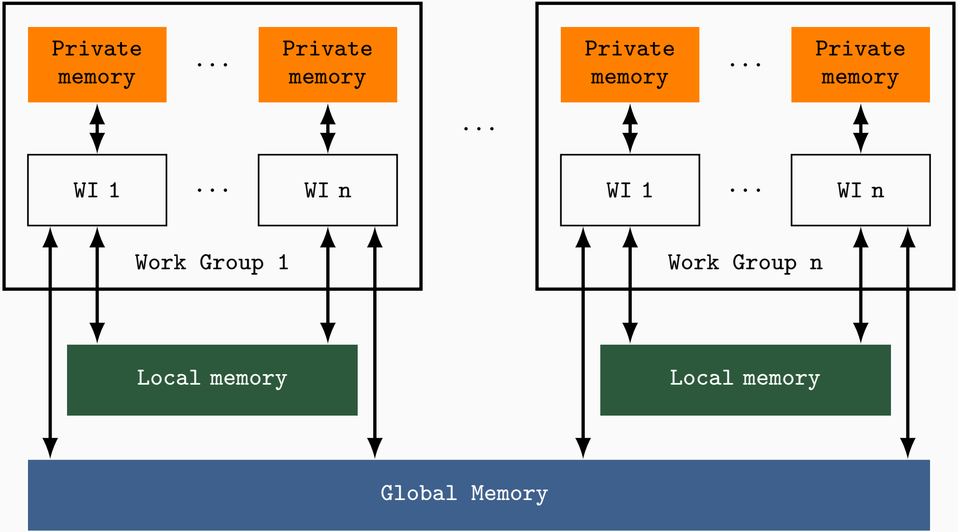
```
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  queue mQ;

  int *buf = malloc_shared<int>(1024, mQ);

  mQ.parallel_for(1024, [=](id<1> idx) {
    buf[idx] = idx;
  });
  mQ.wait();

  return 0;
}
```

SYCL

```cpp
#include <sycl/sycl.hpp>
using namespace sycl;

int main() {
  queue mQ;

  int *buf = malloc_shared<int>(1024, mQ);

  mQ.parallel_for(1024, [=](id<1> idx) {
    buf[idx] = idx;
  });
  mQ.wait();

  return 0;
}
```



Memory model



Why inference ?

- Generic can have a cost in terms of performance
- Some (potential) SYCL backends do not support generic:
    - OpenCL: unknown in 1.2, optional in 3.0
    - Vulkan

Inference in SYCL 1.2.1 / 2020: call graph duplication

- Address space inference in SYCL works in tandem with a `call graph duplication`.
  - This allows us to ensure the inference doesn't stop at function boundaries

```
void foo(int*);
[...]
__global int* i = /*...*/; // i deduced to __global
foo(i);
__local int* j = /*...*/; // j deduced to __local
foo(j);
```

Inference in SYCL 1.2.1 / 2020: call graph duplication

```
void foo(int*);
void foo(__global int*); // compiler generated
void foo(__local int*); // compiler generated
[...]
__global int* i;
foo(i);
__local int* j;
foo(j);
```

Inference in SYCL 1.2.1 / 2020: call graph duplication

```
void bar(int*);

void foo(int* p) { bar(p); }
void foo(__global int*) { bar(p); } // compiler generated
void foo(__local int*){ bar(p); } // compiler generated
[...]
__global int* i;
foo(i);
__local int* j;
foo(j);
```

Inference in SYCL 1.2.1 / 2020: call graph duplication

```
void bar(int*);
void bar(__global int*); // compiler generated
void bar(__local int*); // compiler generated

void foo(int* p) { bar(p); }
void foo(__global int*) { bar(p); } // compiler generated
void foo(__local int*){ bar(p); } // compiler generated
[...]
__global int* i;
foo(i);
__local int* j;
foo(j);
```

Inference in SYCL 1.2.1 / 2020

- Greedy approach
- Update pointer/reference type of a decl based on the type of its initializer

```
// deduce to global
int* P = accessor_to_buffer.get_ptr();
```

- If no initializer, then it defaults to private

```
int* P; // default to private
P = accessor_to_buffer.get_ptr();
```

- `sycl::multi_ptr` to be used to specify an address space or work around limitation

Why the update

- Works fine in 1.2.1
  - seriously limited compared to using generic
  - Does not play well with USM
- Greedy approach and defaulting rules are the core problems
- Rules are ambiguous in some places

Why the update: USM

```
struct Kernel {
  int *buf; // <- default to private
  void operator()(id<1> idx) const {
    buf[idx] = idx; // buf should be a pointer to global
  }
};

int *buf = malloc_shared<int>(1024, mQ);
Kernel K{buf};

mQ.parallel_for(1024, K);
```

Overview of the new approach

- Put fewer constrains on users
- Allow support for USM
- Will not guarantee that a program compiling with generic will also compile in inferred mode
    - But gives better chances it will

Overview of the new approach

- Uses a type inference technique
- Stop defaulting to private if no constraints
  - Yields a type variable, to be resolved depending on how the variable is used
- Structure can be duplicated as well

Inference working principle: example 1

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

Inference working principle: example 1

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

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

Inference working principle: example 2

```
struct Data {
  int *f1;
  int *f2;
};

void foo(Data);
```

Inference working principle: example 2

```
<A, B>
struct Data {
  int A *f1;
  int B *f2;
};

void foo(Data<A1, B1>, Data<A2, B2>*);
```

Inference working principle: unifying example 1

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

Inference working principle: unifying example 1

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

Inference working principle: unifying example 1

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

<A>
void bar(int A* a, int __global* b) {
  int* t; // Internally, t is seen as 'int T*'
  t = foo(a, b);
}
```

Inference working principle: unifying example 1

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

void bar(int __global* a, int __global* b) {
  int __global* t;
  t = foo(a, b);
}
```

Inference working principle: raising error

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

void error(int __local* a, int __global* b) {
  foo(a, b); // Error
}
```

Points to consider for a well defined SYCL specification

- no longer defaulting to private
  - => undeduced is dangerous

Points to consider for a well defined SYCL specification

```
struct Kernel {

  int** in;
  int** out;

  void operator()() {
    **out = **in;
  }
};
```

Points to consider for a well defined SYCL specification

```
<A, B>
struct Kernel {

  int A* __global* in;
  int B* __global* out;

  void operator()() {
    **out = **in;
  }
};
```

Points to consider for a well defined SYCL specification

```
struct Kernel {

  int __global* __global* in;
  int __global* __global* out;

  void operator()() {
    **out = **in;
  }
};
```

Points to consider for a well defined SYCL specification

- no longer defaulting to private
  - => undeduced is dangerous
- handling cast

Points to consider for a well defined SYCL specification

```
T1 * t = /*...*/;
T2 * t_cast = reinterpret_cast<T2*>(t);

struct Foo1 {
 void* f;
};

struct Foo2 {
 int* f;
};

Foo1 * f1 = /*...*/;
Foo2 * f2 = reinterpret_cast<Foo2*>(f1);
```

Points to consider for a well defined SYCL specification

```
int * p = /*...*/;
intptr_t ip = reinterpret_cast<intptr_t>(p);
```

Points to consider for a well defined SYCL specification

- no longer defaulting to private
  - => undeduced is dangerous
- handling cast
  - function boundary of cast

Points to consider for a well defined SYCL specification

```
void my_memcpy(void* dst, void* src, size_t s) {
 /* ... */
}

void call() {
  Str t1 = /*...*/;
  Str t2;
  my_memcpy(&t1, &t2, sizeof (Str));
}
```

Points to consider for a well defined SYCL specification

- no longer defaulting to private
  - => undeduced is dangerous
- handling cast
  - function boundary of cast
  - Standard library exceptions

Conclusion

- Allow more rich programs
- Working principles are not significantly more difficult to understand
  - New rules are more complex than the current ones
  - But less complex than what can be found with languages like Haskell
  - Possible to handle complex code using USM
- Current and future work:
  - Make it production ready!
    - Error reporting and performance
  - Extension to be written