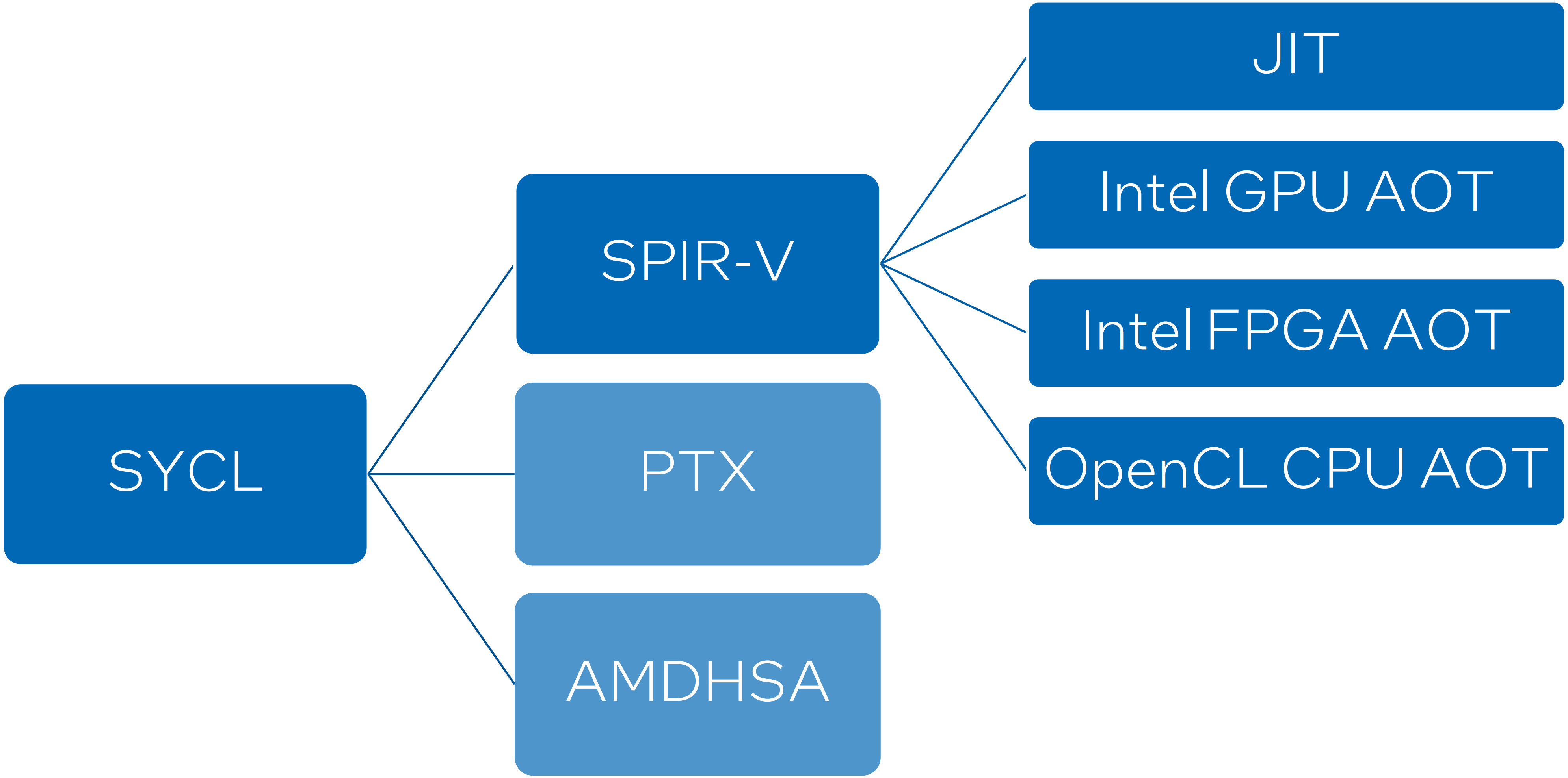
EuroLLVM’24

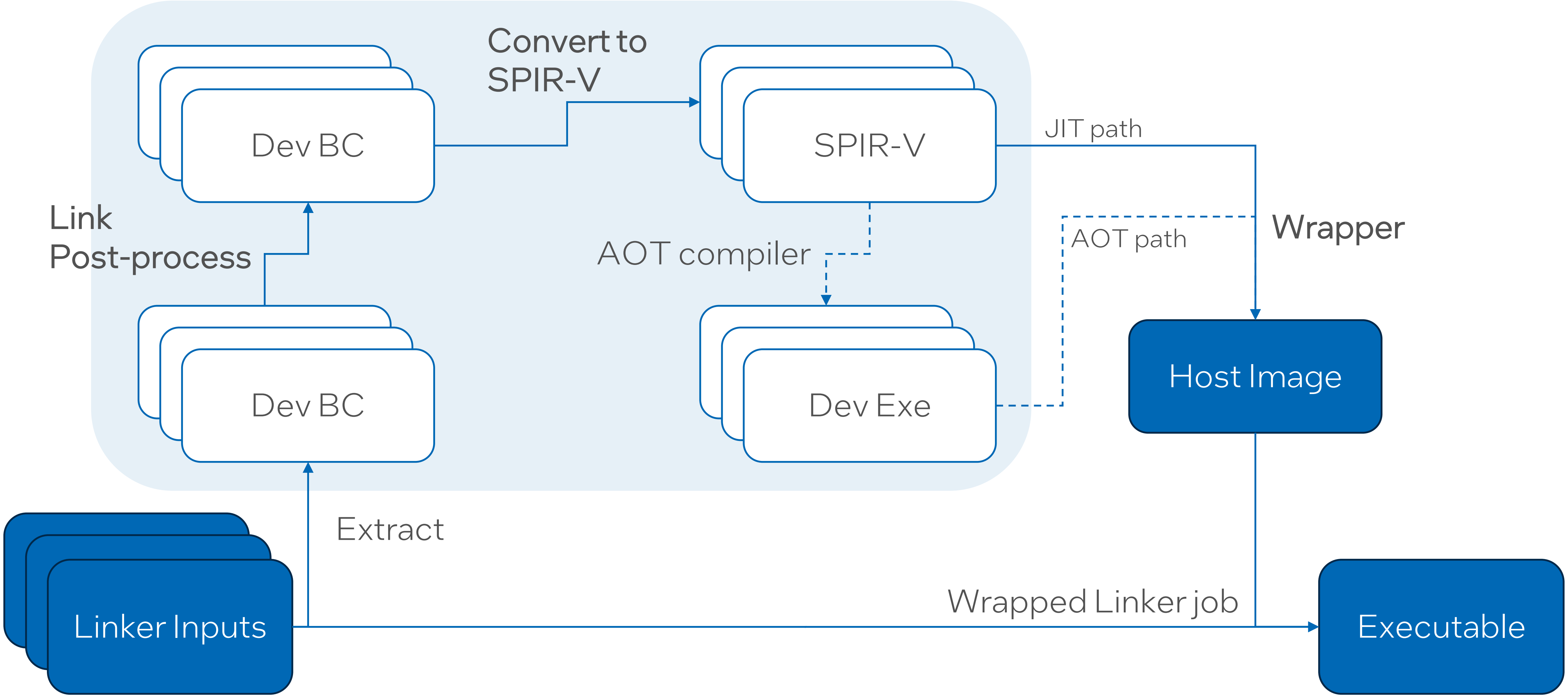
Enhancing clang-linker-wrapper to support SYCL/DPC++

Alexey Sachkov

Supported targets



SYCL offloading flow (simplified)



Wrapping (clang-*offload*-wrapper)

OpenMP as a starting point

```
struct __tgt_device_image {
  void *ImageStart;
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

Wrapping (clang-*offload*-wrapper)

What we have ended up with (simplified)

```
struct __tgt_sycl_device_image {
  uint8_t OffloadKind;
  uint8_t Format;
  const char *DeviceTargetSpec;
  const char *CompileOptions;
  const char *LinkOptions;
  __property_set *PropertiesBegin;
  __property_set *PropertiesEnd;
  void *ImageStart;
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

Wrapping (clang-*offload*-wrapper)

Device image is a black box

No reverse-engineering of 3rd-party toolchains

No dependencies on implementation details of other components

```
struct __tgt_sycl_device_image {
  uint8_t OffloadKind; // SYCL vs OpenMP
  uint8_t Format; // AOT vs SPIR-V
  const char *DeviceTargetSpec; // Triple
  const char *CompileOptions;
  const char *LinkOptions;
  __property_set *PropertiesBegin;
  __property_set *PropertiesEnd;
  void *ImageStart;
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

intel/llvm, commit 0738099b, Mar 2019

Wrapping (clang-*offload*-wrapper)

JIT compiler should follow host compiler

If an app is compiled with `-O0`, device code should be JIT-compiled with `-O0` as well.

```
struct __tgt_sycl_device_image {
  uint8_t OffloadKind;
  uint8_t Format;
  const char *DeviceTargetSpec;
  const char *CompileOptions; // to be passed
  const char *LinkOptions; // to JIT compiler
  __property_set *PropertiesBegin;
  __property_set *PropertiesEnd;
  void *ImageStart;
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

intel/llvm, commit 47a94115, Feb 2020

Wrapping (clang-*offload*-wrapper)

Flexible mechanism to communicate with runtime

Device image can be extended with new information without breaking ABI.

```
struct __tgt_sycl_device_image {
  uint8_t OffloadKind;
  uint8_t Format;
  const char *DeviceTargetSpec;
  const char *CompileOptions;
  const char *LinkOptions;
  __property_set *PropertiesBegin;
  __property_set *PropertiesEnd;
  void *ImageStart;
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

intel/llvm, commit 29abe37d, Mar 2020

Wrapping (clang-*offload*-wrapper)

Flexible mechanism to communicate with runtime

Groups of key-value pairs.

Generated from named metadata left by passes.

```
struct __property_set {
  char *Name;
  __device_image_property *Begin;
  __device_image_property *End;
}

struct __device_image_property {
  char *Name;
  void *Value;
  // Type is uint32 or byte array
  uint32_t Type;
  uint64_t ValueSize;
}
```

intel/llvm, commit 29abe37d, Mar 2020

Device image properties

Example usage: optional kernel features

Each device image is accompanied by a property set listing device requirements:

```
[SYCL/device requirements]
  aspects=list of sycl::aspect values
```

```
queue q;
// selected device *does not* support fp64
assert(!q.get_device().has(aspect::fp64));

q.single_task([=]() {
  // kernel uses fp64
  double pi = 3.14;
});
// single_task is expected to throw
// feature_not_supported exception
```

Wrapping (clang-linker-wrapper)

__tgt_device_image: expectation

```
struct __tgt_device_image {
  // Pointer to the target code start
  void *ImageStart;
  // Pointer to the target code end
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

Wrapping (clang-linker-wrapper)

__tgt_device_image: reality

```
struct __tgt_device_image {
  // Pointer to OffloadingImage start
  void *ImageStart;
  // Pointer to OffloadingImage end
  void *ImageEnd;
  __tgt_offload_entry *EntriesBegin;
  __tgt_offload_entry *EntriesEnd;
};
```

```
struct OffloadingImage {
  // LLVM BC, PTX, Object, etc.
  ImageKind TheImageKind;
  // OpenMP, CUDA, etc.
  OffloadKind TheOffloadKind;
  uint32_t Flags;
  // equivalent of device image properties
  MapVector<StringRef, StringRef> StringData;
  // actual target code
  std::unique_ptr<MemoryBuffer> Image;
};
```

llvm/llvm-project, commit e471ba3d, Apr 2022

Looks like everything is in place, right?

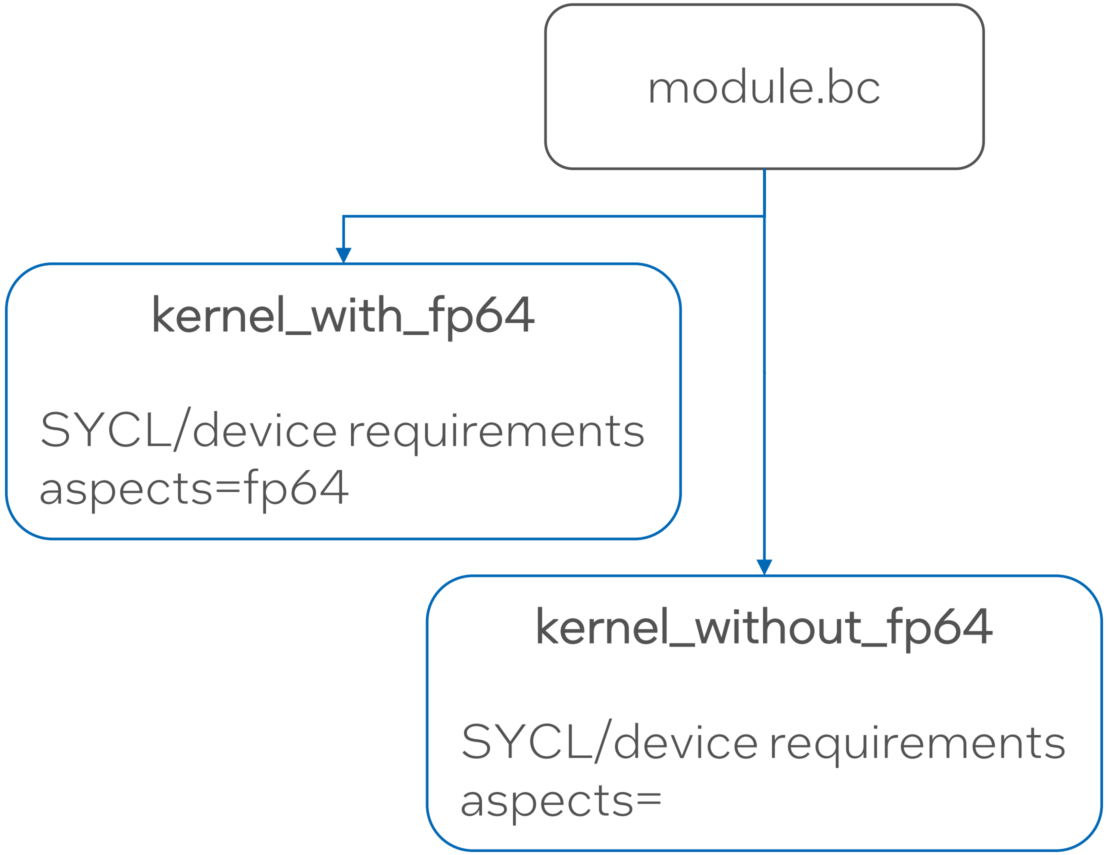
Almost

The fact that `__tgt_device_image` contains extra metadata is not documented.

`libomptarget`'s `DeviceImage` constructor discards that extra metadata, making it unavailable to target plugins.

Link & Post-process

Device code split



```
queue q;

if (q.get_device().has(aspect::fp64))
  q.single_task<kernel_with_fp64>([=]() {
    // kernel uses fp64
    double pi = 3.14;
  });
else
  q.single_task<kernel_without_fp64>([=]()
    // kernel *does not* use fp64
    float pi = 3.14f;
  });
```

Link & Post-process

Device code split

Per used optional features

- For SYCL 2020 conformance

Per kernel

- To reduce JIT overhead

```
queue q;

if (q.get_device().has(aspect::fp64))
  q.single_task<kernel_with_fp64>([=]() {
    // kernel uses fp64
    double pi = 3.14;
  });
else
  q.single_task<kernel_without_fp64>([=]()
    // kernel *does not* use fp64
    float pi = 3.14f;
  });
```

What’s next?

Stay tuned for PRs and possibly RFCs

SYCL offloading kind is coming to `clang-linker-wrapper`.

Let’s make `__tgt_device_image` content more obvious and better documented.

Is anyone else interested in device code split to make it generic?

References

- GitHub intel/llvm repository
- [RFC] Add Full Support for the SYCL Programming Model
    - LLVM Discourse
- [RFC] Offloading design for SYCL offload kind and SPIR targets
    - LLVM Discourse
- What exactly is stored in `__tgt_device_image` struct?
    - LLVM Discourse
- EuroLLVM'19, A. Savonichev "SYCL compiler: zero-cost abstraction and type safety for heterogeneous computing"
    - YouTube

intel®