oneAPI DPC++ 编译器和运行时架构设计

简介

本文档描述了 DPC++ 编译器和运行时库的体系结构。有关 DPC++ 规范，请参阅规范.

DPC++ 编译器架构

DPC++应用程序编译流程：

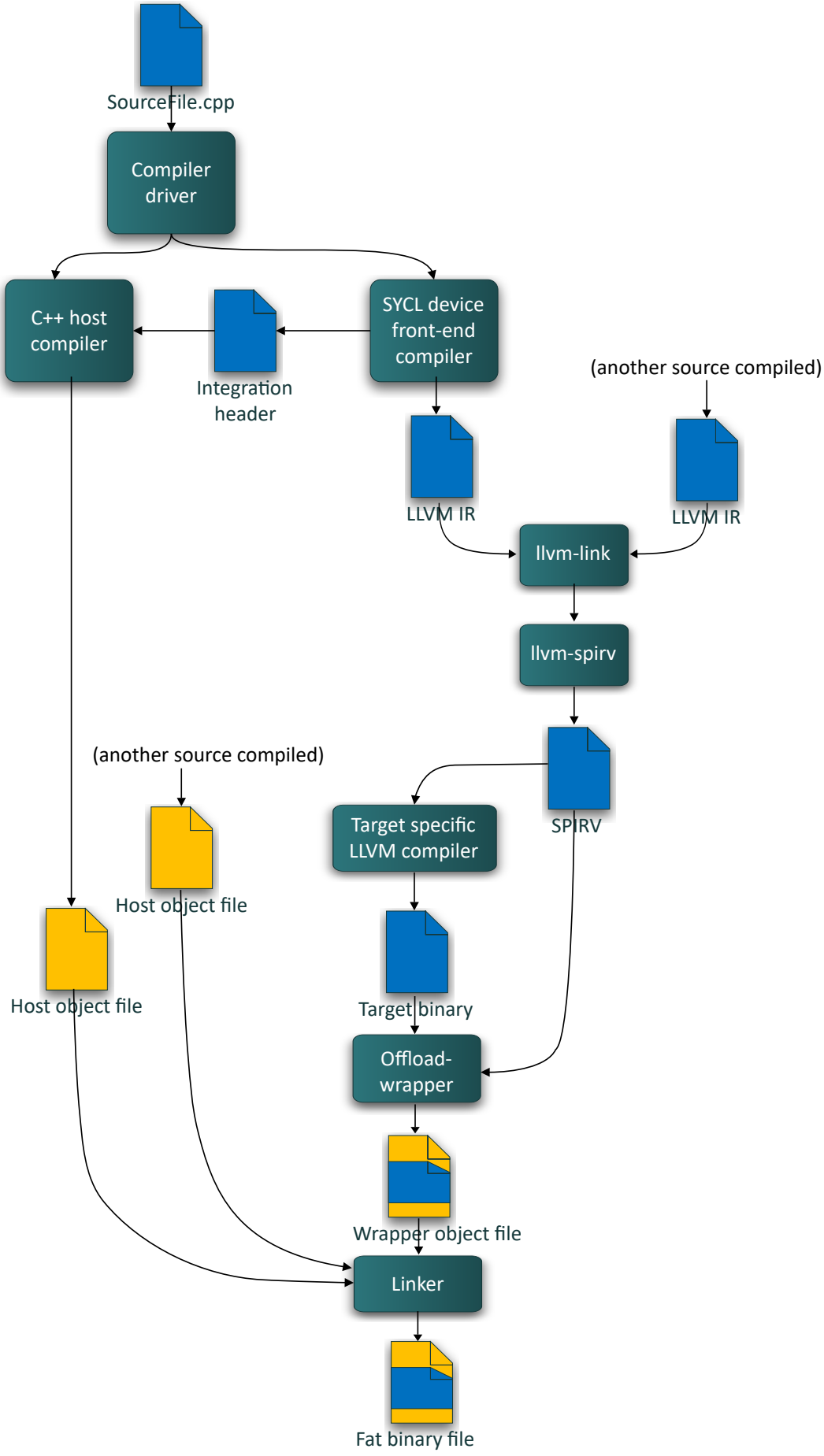

SourceFile.cpp
Compiler driver
C++ host compiler
Integration header
SYCL device front-end compiler
(another source compiled)
LLVM IR
LLVM IR
llvm-link
llvm-spirv
SPIRV
(another source compiled)
Target specific LLVM compiler
Host object file
Host object file
Target binary
Offload-wrapper
Wrapper object file
Linker
Fat binary file

图 1. 应用程序构建流程。

DPC++ 编译器在逻辑上可以分为主机编译器和多个设备编译器——每个编译器支持一个目标。Clang 驱动程序编排编译过程，它将为每个有目标的请求调用一次设备编译器，然后它将调用主机编译器来编译 SYCL 源代码的主机部分。在最简单的情况下，当编译和链接在一次编译器驱动程序调用中完成时，一旦编译完成，设备对象文件（实际上是 LLVM IR 文件）将被`llvm-link`工具链接。然后使用`llvm-spirv`工具 将生成的 LLVM IR 模块转换为 SPIR-V 模块， 并使用`clang-offload-wrapper`工具将模块包装到一个主机对象文件中。一旦所有主机对象文件和带有设备代码的包装对象准备就绪，驱动程序就会调用通用平台链接器，并生成名为“fat binary”的最终可执行文件。这是一个主机可执行文件或库，其中包含了在命令行中指定的每个目标的嵌入链接镜像文件。

根据用户是否选择执行以下一项或多项操作，编译过程有许多变化：

- 与链接分开执行编译
- 为一个或多个目标提前编译设备 SPIR-V 模块
- 执行设备代码拆分，以便设备代码分布在多个模块中，而不是包含在单个模块中
- 执行静态设备库的链接 下面的部分提供了有关其中一些场景的更多详细信息。

SYCL 源代码也可以编译为常规 C++ 代码，在这种模式下，代码没有“设备部分”——一切都在主机上执行。

设备编译器进一步分为以下主要组件：

- **前端** - 解析输入源，“概述”代码的设备部分，对设备代码应用额外的限制（例如，没有异常或虚拟调用），仅为设备代码生成 LLVM IR 和提供运行时库的内核名称、参数顺序和数据类型的“集成头文件”。
- **中端** - 转换初始 LLVM IR 以供后端使用。现在的中端转换只包括几个过程：
- 可选：地址空间推理传递
    - 待定：潜在中端优化器可以运行任何 LLVM IR 转换，但只有一个限制：后端编译器应该能够处理转换后的 LLVM IR。
    - 可选：LLVM IR → SPIR-V 转换器。
- **后端** - 生成本机“设备”代码。 它在图 1 中显示为“特定于目标的 LLVM 编译器”框。它在编译时（在提前编译场景中）或在运行时（在即时编译场景中）被调用。

设计说明：在当前设计中，我们使用 SYCL 设备前端编译器来生成集成头文件，原因有两个。首先，必须可以使用任何主机编译器来生成 SYCL 异构应用程序。其次，即使主机编译使用相同的 clang 编译器，集成头文件中提供的信息也会被 SYCL 运行时实现使用（包含），因此该头文件必须在主机编译开始之前可用。

Clang 前端中的 SYCL 支持

Clang 前端中的 SYCL 支持可以分为以下组件：

- 设备代码概述。该组件负责识别和概述单一来源中的“设备代码”。
- SYCL 内核函数对象（仿函数或 lambda）底层化。该组件为 SYCL 内核创建了一个 OpenCL 内核函数接口。
- 设备代码诊断。该组件对设备代码实施语言限制。
- 集成头文件生成。该组件发出的信息是使用 OpenCL API 绑定SYCL代码的主机和设备部分所需要的。

设备代码概述

下面是一个演示编译器概述工作的 SYCL 程序的代码示例：

```
int foo(int x) { return ++x; }
int bar(int x) { throw std::exception{"CPU code only!"}; }
...
using namespace cl::sycl;
queue Q;
buffer<int, 1> a{range<1>{1024}};
Q.submit([&](handler& cgh) {
      auto A = a.get_access<access::mode::write>(cgh);
      cgh.parallel_for<init_a>(range<1>{1024}, [=](id<1> index) {
        A[index] = index[0] * 2 + foo(42);
      });
    }
...
```

在此示例中，编译器需要为设备编译传递给`cl::sycl::handler::parallel_for`方法的 lambda 表达式，以及从 lambda 表达式中调用的`foo`函数。

当编译单个源代码文件中的设备部分时，编译器必须忽略`bar`函数，因为它在源代码的设备部分中未使用（传递给`cl::sycl::handler::parallel_for`的 lambda 表达式的内容以及从此 lambda 表达式调用的任何函数）。

当前的方法是使用运行时的 SYCL 内核属性将传递给`cl::sycl::handler::parallel_for`的代码标记为“内核函数”。运行时库不能将 `foo` 标记为“设备”代码——这是编译器的工作：遍历内核函数中可访问的所有符号，并将它们添加到“设备部分”的代码中，并用新的 SYCL 设备属性标记它们。

lambda 函数对象和命名函数对象底层化

主机和设备之间共享的所有 SYCL 内存对象（缓冲区/图像，这些对象映射到 OpenCL 缓冲区和图像）必须通过特殊 `accessor`类访问。“设备”端实现包含指向设备内存的指针的类。由于在 OpenCL 中无法将内部带有指针的结构作为内核参数传递，因此主机和设备之间共享的所有内存对象都必须作为原始指针传递给内核。

SYCL 也有一种特殊的机制，用于将内核参数从主机传递到设备。在 OpenCL 中，内核参数是通过 `clSetKernelArg` 函数来设置的，而在 SYCL 中，所有内核参数都是“SYCL 内核函数”的字段，可以定义为 lambda 函数或命名函数对象并作为参数传递给 SYCL 函数用于调用内核（例如`parallel_for`或 `single_task`）。例如，在前面的代码片段中，`accessor A`就是一个这样捕获的内核参数。

为了促进 SYCL 内核数据成员到 OpenCL 内核参数的映射并克服 OpenCL 限制，我们在编译器中添加了 OpenCL 内核函数的生成。OpenCL 内核函数包含 SYCL 内核函数的主体，接收类似 OpenCL 的参数，并额外执行一些操作以使用这些参数初始化 SYCL 内核数据成员。在一些伪代码中，上述代码片段的 OpenCL 内核函数如下所示：

```
// SYCL kernel is defined in SYCL headers:
template <typename KernelName, typename KernelType/*, ...*/>
__attribute__((sycl_kernel)) void sycl_kernel_function(KernelType KernelFuncObj) {
  // ...
  KernelFuncObj();
}

// Generated OpenCL kernel function
```

```
__kernel KernelName(global int* a) {
  KernelType KernelFuncObj; // Actually kernel function object declaration
  // doesn't have a name in AST.
  // Let the kernel function object have one captured field - accessor A.
  // We need to init it with global pointer from arguments:
  KernelFuncObj.A.__init(a);
  // Body of the SYCL kernel from SYCL headers:
  {
    KernelFuncObj();
  }
}
```

OpenCL 内核函数由编译器内部的 Sema 使用 AST 节点生成。内核参数传递的更多细节可以在SYCL 内核参数处理和数组支持文档中找到。 .

驱动程序（编译器）中的 SYCL 支持

驱动程序中的 SYCL 转存支持是基于 clang 驱动的概念和定义：

- 目标三元组和每个目标的本机工具链（包括"虚拟"目标，如 SPIR-V）。
- 基于通用转存操作的 SYCL 转存。

SYCL 编译管道与其他编译场景相比有一个特点——管道中的某些操作可能会输出多个"集群"文件，它们稍后会被其他操作使用。例如，每个设备二进制文件可能伴随着一个符号表和一个专门的常量映射 - 这些附加信息会被SYCL 运行时库使用 - 它需要由转存包装工具存储到设备二进制描述符中。启用设备代码拆分功能后，可以有多个这样的文件集（集群）——每个单独的设备二进制文件一个文件集。

当前的 clang 驱动程序设计不允许对其建模，即：

1. 动作图（action graph）中的多个输入/输出。
2. 多个输入/输出的逻辑分组。例如，一个输入或输出可以由多对文件组成，其中每一对代表单个设备代码模块的信息：[带有设备代码的文件，带有导出符号的文件]。

为了支持这一点，SYCL 引入了`file-table-tform`工具。此工具可以按照作为输入参数传递的命令来转换文件表。表中的每一行代表一个文件簇，每一列 —— 与一个簇相关联的一种类型的数据。该工具可以替换和提取列。例如， `sycl-post-link`工具可以输出两个文件簇以及引用簇中所有文件的文件表：

```
[Code|Symbols|Properties]
a_0.bc|a_0.sym|a_0.props
a_1.bc|a_1.sym|a_1.props
```

当参与动作图时，此工具输入文件表（`TY_Tempfiletable`clang 输入类型）和/或文件列表（`TY_Tempfilelist`），执行转换请求并输出文件表或列表。从 clang 设计的角度来看，仍然有单一的输入和输出，尽管实际上有多个。

例如，根据编译选项，上面"代码"列中的文件可能需要在设备代码拆分步骤之后进行 AOT 编译，执行 `sycl-post-link`工具是作为代码转换序列的一部分。然后驱动程序会：

- 使用`file-table-tform`提取代码文件并生成文件列表：

```
a_0.bc
a_1.bc
```

- 将此文件列表和 AOT 编译命令传递给`llvm-foreach`工具，以在列表中的每个文件上调用该命令。这将产生另一个文件列表

```
a_0.bin
a_1.bin
```

- 然后`file-table-tform`再次被调用，并使用`.bin`替换文件表中的`.bc`以获取新的文件表：

```
  [Code|Symbols|Properties]
  a_0.bin|a_0.sym|a_0.props
  a_1.bin|a_1.sym|a_1.props
```

- 最后，将此文件表传递给`clang-offload-wrapper`工具以构造一个嵌入所有这些文件的包装对象。

请注意，当更多的行（簇）或列（例如“清单”文件）被添加到表中时，图形不会改变。

启用 SYCL 转存

要启用SYCL 规范中描述的单源多编译器扫描 (SMCP) 技术的编译，必须将一个特殊选项传递给 clang 驱动程序：

`-fsycl`

指定此选项后，驱动程序将为`-fsycl-targets`选项中指定的目标调用主机编译器和许多 SYCL 设备编译器 。如果`-fsycl-targets`未指定，则假定为单个 SPIR-V 目标，并为此目标调用单个设备编译器。

选项`-sycl-std`允许指定将使用哪个版本的 SYCL 标准进行编译。此选项的默认值为`1.2.1`。

提前（AOT）编译

提前编译是在编译时调用后端以生成最终二进制文件的过程，与将最终代码生成推迟到应用程序运行时进行的即时 (JIT) 编译相反。

AOT 编译通过跳过 JIT 编译来减少应用程序执行时间，并且可以在部署之前测试最终的设备代码。

JIT 编译提供设备代码的可移植性和特定的目标优化。

本机目标列表

提前编译模式意味着必须有一种方法来指定一组目标架构并为其编译设备代码。默认情况下，编译器生成 SPIR-V，OpenCL 设备 JIT 编译器生成本地目标二进制文件。

要生成二进制文件就需要使用`triple1`和`triple2`来设置目标三元组来指定目标的体系结构 ，请使用以下 SYCL 编译器选项：

```
-fsycl-targets=triple1,triple2
```

将从 SYCL 内核为两个目标三元组指定的设备生成二进制文件。这基本上告诉驱动程序必须调用哪些设备编译器来编译 SYCL 内核代码。默认情况下，假定采用 JIT 编译方法，并且为单个目标三元组编译设备代码 —— [spir,spir64]--`。

设备代码格式

每个目标可以支持多种代码形式，每个设备编译器定义并理解指定特定代码形式的助记符，例如“`visa:3.3`”可以用来指定目标为英特尔 GPU （Gen 架构）的虚拟 ISA 版本 3.3。用户可以使用目标特定的选项机制指定所需的代码格式，类似于 OpenMP。

```
-Xsycl-target-backend=<triple> "arg1 arg2 ..."
```

例如，要支持转存到 Gen9/vISA3.3，将使用以下选项：

```
-fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device skl"
```

驱动程序直接将`-device skl`参数传递给 Gen 设备后端编译器，而编译器`ocloc`不对其进行解析。

`ocloc`还能够离线编译多个 ISA 版本/Gen 架构。例如，要使设备二进制文件与所有 Intel Gen9 GPU 平台兼容，可以使用：

```
-fsycl -fsycl-targets=spir64_gen -Xsycl-target-backend "-device gen9"
```

有关支持的平台和参数语法的更多详细信息，请通过检测本地按照的`ocloc`并 运行`ocloc compile --help`来参考 GPU 离线编译器手册。

分开的编译和链接

编译器支持以下功能

- 在生成要输送到后端的最终 SPIR-V 之前，从不同源文件获得链接的设备代码。
- 将应用程序构建拆分为分开的编译和链接步骤。

与上图 1 中所示的相比，总体构建流程变化如下。

编译步骤使用转存捆绑器以及同一异构源生成的每个 <host object, device code IR> 对来生成所谓的“胖对象（fat object）”。胖目标文件是编译的结果，类似于使用通常的非转存编译器的目标文件。

链接步骤首先将输入的胖对象分解为其组成部分，然后继续以与图 1 相同的方式 - 分别链接主机代码和设备代码，最后生成“胖二进制文件（ fat binary）”。

下图说明了构建流程中的变化。转存捆绑器/取消捆绑器操作基本上插入在`llvm-link`和`linker`调用之间，如图 1 所示。

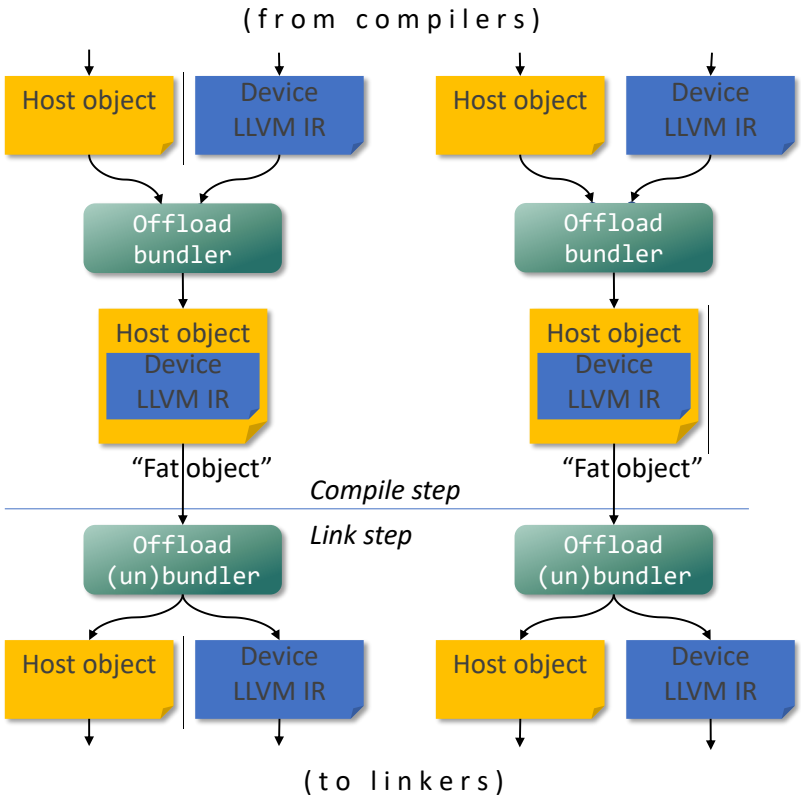


图 2. 拆分编译和链接。

当前实现使用 LLVM IR 作为`fat objects`的默认设备二进制格式，并将“链接的 LLVM IR”转换为 SPIR-V。这个决定的原因之一是 SPIR-V 不支持链接模板函数，它可以在多个模块中定义，并且链接器必须解析多个定义。LLVM IR 使用函数属性来满足“单一定义规则”，这在 SPIR-V 中没有对应的部分。

胖二进制（Fat binary）创建细节

“胖二进制文件”是最终主机链接步骤的结果 —— 这是一个嵌入了设备二进制文件的主机二进制文件。运行时，它会自动在 SYCL 运行时库中注册所有可用的设备二进制文件。本节介绍如何实现这一点。

胖二进制文件是使用通用的链接器创建的 - 例如Linux 的`ld` 和 Windows 上的`link.exe`。为了让链接器能够嵌入设备二进制文件，它们首先被“包装”到称为“包装器对象”的主机对象文件中。然后这个包装对象与其余的普通主机对象和/或库进行链接。

包装器对象由`clang-offload-wrapper`工具创建，或者简称为“转存包装器”。创建的包装对象有两个主要组件：

1. 全局符号 - 转存描述符 —— 指向放入对象数据节的特殊数据结构。它包含有关包装的设备二进制文件的所有所需信息——二进制文件的数量、每个二进制文件定义的符号等——以及二进制文件本身。
2. 注册/注销函数。前者放在一个特殊的节，以便在运行时将父胖二进制文件加载到进程中调用它，后者放在另一个节中，以便在卸载父胖二进制文件时调用。注册函数基本上采用指向转存描述符的指针并调用 SYCL 运行时库的注册函数，将其作为参数传递。

头文件`pi.h`中描述了转存描述符类型层次结构。顶层结构是`pi_device_binaries_struct`。

设备链接

`-fsycl-link` 标志指示编译器在不完全链接主机代码的情况下完全链接设备代码。这种编译的结果是一个胖对象，其中包含一个完全链接的设备二进制文件。此流程的主要动机是让用户能够在更改仅影响主机的代码时节省重新编译时间。在生成设备映像需要很长时间（例如 FPGA）的情况下，这种节省可能非常显着。

例如，如果用户将源代码分成四个文件：dev_a.cpp、dev_b.cpp、host_a.cpp和host_b.cpp，其中只有dev_a.cpp和dev_b.cpp包含设备代码，那么编译过程将可以分为三个步骤：

1. 设备链接：dev_a.cpp dev_b.cpp -> dev_image.o（包含设备镜像）
2. 主机编译（c）：host_a.cpp -> host_a.o；host_b.cpp -> host_b.o
3. 链接：dev_image.o host_a.o host_b.o -> 可执行文件

对于某些目标，步骤 1 可能需要数小时。但如果用户希望在仅修改 host_a.cpp 和 host_b.cpp 后重新编译，他们可以简单地运行步骤 2 和 3，而无需重新运行昂贵的步骤 1。

编译器负责检验用户是否向设备链接步骤提供了所有相关的文件。有2种情况需要检查：

1. 设备链接步骤中存在的内核引用的缺失符号（例如，已知内核调用的函数或已知内核使用的全局变量）。
2. 缺少内核。

通过扫描已知内核，可以在设备二进制生成阶段（步骤 1）中识别案例 1。情况 2 必须由驱动程序通过在最终链接阶段检查新引入的内核来验证（步骤 3）。

`llvm-no-spir-kernel` 工具可以方便的检查驱动程序中的第2种情况。它检测模块是否包含内核并按如下方式调用：

```
llvm-no-spir-kernel host.bc
```

如果不存在内核，则返回 0，否则返回 1。

设备代码链接后的步骤

在链接时，所有设备代码始终链接到单个 LLVM IR 模块。另外，这个 LLVM IR 模块移交给转存包装器之前，`sycl-post-link`工具会对其执行许多最终转换。其中包括：

- 设备代码拆分
- 符号表生成
- 专有常数底层化

根据选项，`sycl-post-link`可以输出单个 LLVM IR 文件，或多个文件以及引用所有文件的文件表。有关文件表的整体描述，请参阅“驱动程序中的 SYCL 支持”部分。下图显示了从单个链接的 LLVM IR 模块到创建包装器对象的编译过程，是clang可能的操作图。该图是多种可能的变体之一，具体取决于：

- 具体目标要求
- 设备代码拆分
- AOT 编译

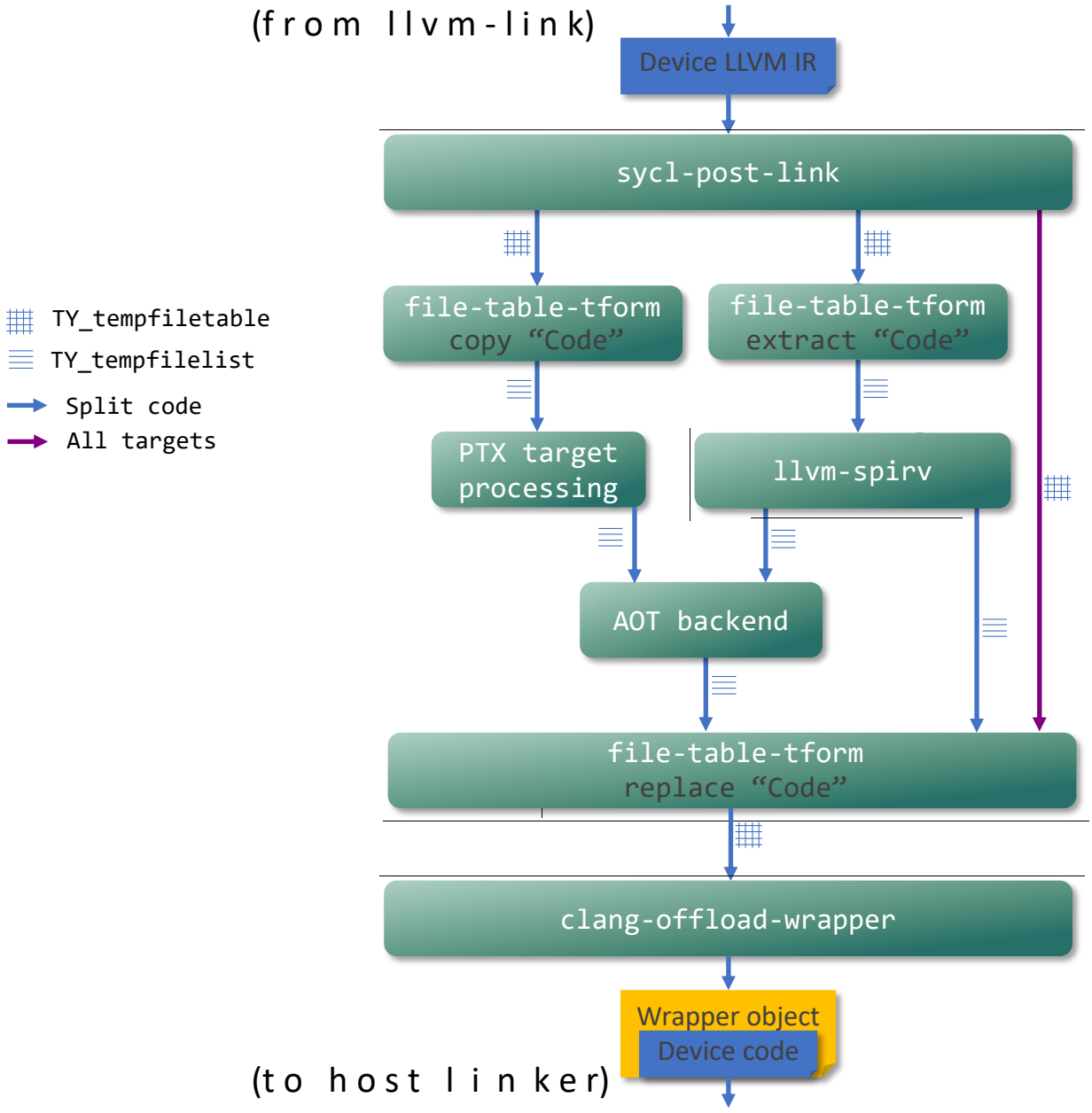


图 3. 设备代码链接流程。

图表边缘的颜色显示根据上述因素采用的路径。每条边也用输入/输出文件类型进行注释。为清楚起见，该图未显示`llvm-foreach`工具的调用。该工具在文件列表中的每个文件上调用给定的命令行。在上图中，只要输入/输出类型为`TY_tempfilelist`且目标不是 PTX，该工具就会应用于`llvm-spirv`和AOT 后端。在此之后，`file-table-tform`需要两个输入 - 文件表和来自`llvm-spirv`或来自 AOT 后端的文件列表。PTX 的情况目前只接受单个输入文件进行处理，因此 `file-table-tform`用于从文件表中提取代码文件，然后由 “PTX 目标处理”步骤处理. 生成设备的二进制文件并插入回文件表中的`file-table-tform`代替被提取的代码文件。

设备代码拆分

将所有设备代码放入单个 SPIR-V 模块在以下情况下效果不佳：

1. 定义了数千个内核，其中只有一小部分在运行时使用。将它们全部放在一个 SPIR-V 模块中会显著增加 JIT 时间。
2. 设备代码可以专门用于不同的设备。例如，只有在 FPGA 上才能执行的内核应该仅能够适用于 FPGA 的扩展。即使从未在其他设备上调用此特定内核，这也会导致其他设备上的 JIT 编译失败。

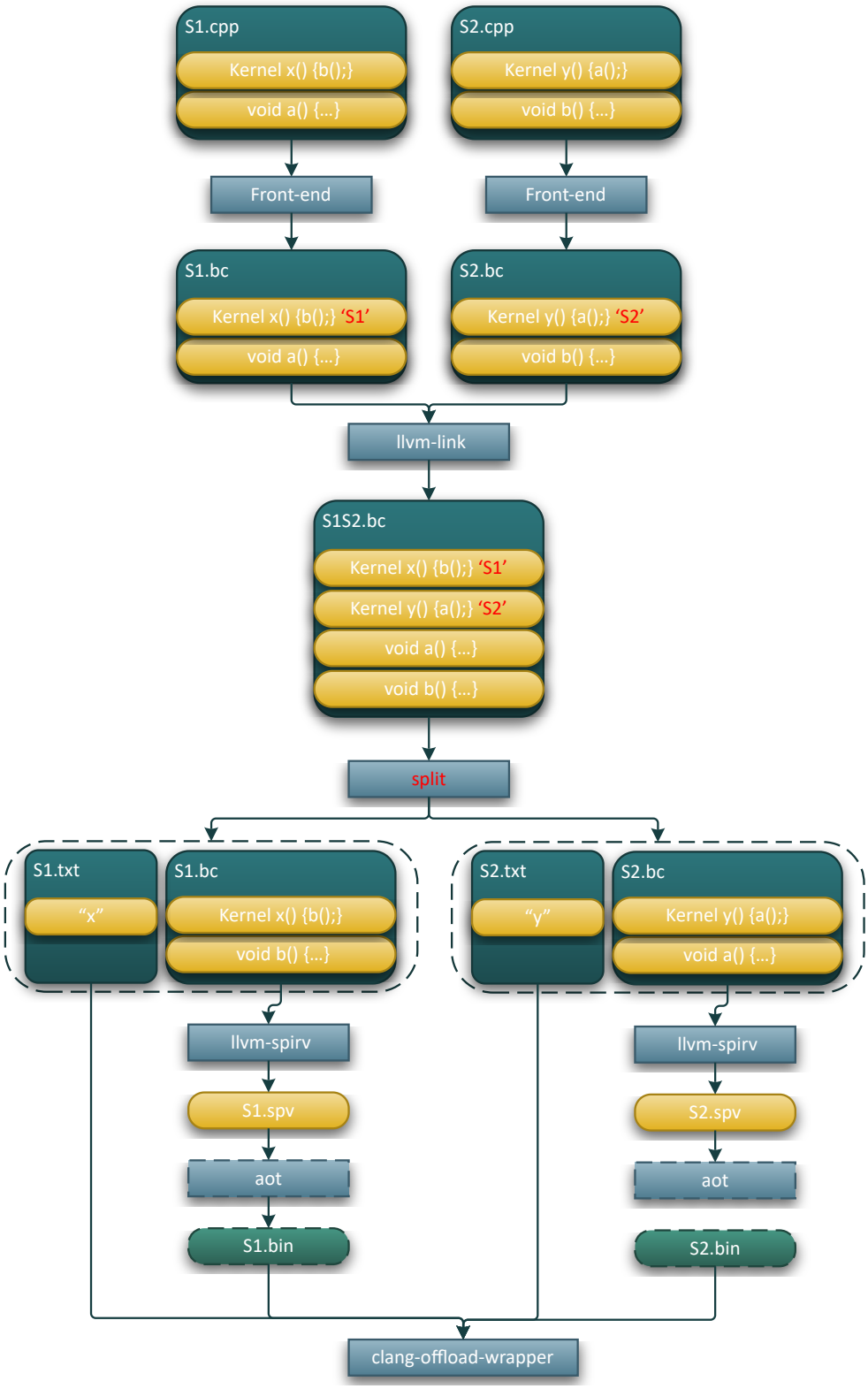
为了解决这些问题，编译器可以将单个模块拆分为更小的模块。以下是支持的功能：

- 根据源代码生成单独的模块 （翻译单元）
- 根据单个内核生成单独的模块

目前的做法是：

- 为 SYCL 前端的每个内核生成带有翻译单元 ID 的特殊元数据。此 ID 将用于按翻译单元对内核进行分组
- 使用 llvm-link 链接所有设备 LLVM 模块
- 在完全链接的模块上执行拆分
- 为每个生成的设备模块生成一个符号表（内核列表），以便在运行时正确选择模块
- 分别对每个生成的模块执行 SPIR-V 翻译和 AOT 编译（如果需要）
- 将有关展示内核的信息添加到每个设备镜像的包装对象

设备代码拆分流程：


S1.cpp
Kernel x() {b();}
void a() {...}
S2.cpp
Kernel y() {a();}
void b() {...}
Front-end
Front-end
S1.bc
Kernel x() {b();} 'S1'
void a() {...}
S2.bc
Kernel y() {a();} 'S2'
void b() {...}
llvm-link
S1S2.bc
Kernel x() {b();} 'S1'
Kernel y() {a();} 'S2'
void a() {...}
void b() {...}
split
S1.txt
"x"
S1.bc
Kernel x() {b();}
void b() {...}
S2.txt
"y"
S2.bc
Kernel y() {a();}
void a() {...}
llvm-spirv
llvm-spirv
S1.spv
S2.spv
aot
aot
S1.bin
S2.bin
clang-offload-wrapper

“拆分”框是由专用工具`sycl-post-link`的功能实现的 。该工具运行一组 LLVM 扫描以拆分输入模块，并为每个生成的设备模块生成一个符号表（内核列表）。

要启用设备代码拆分，必须将一个特殊选项传递给 clang 驱动程序：

`-fsycl-device-code-split=<value>`

此选项有三个可能的值：

- `per_source` -可以为每个源代码（翻译单元）生成一个单独的模块
- `per_kernel` - 允许为每个内核生成一个单独的模块
- `off` - 禁用设备代码拆分

符号表生成

待定

专有常数底层化

请参阅相应的文档

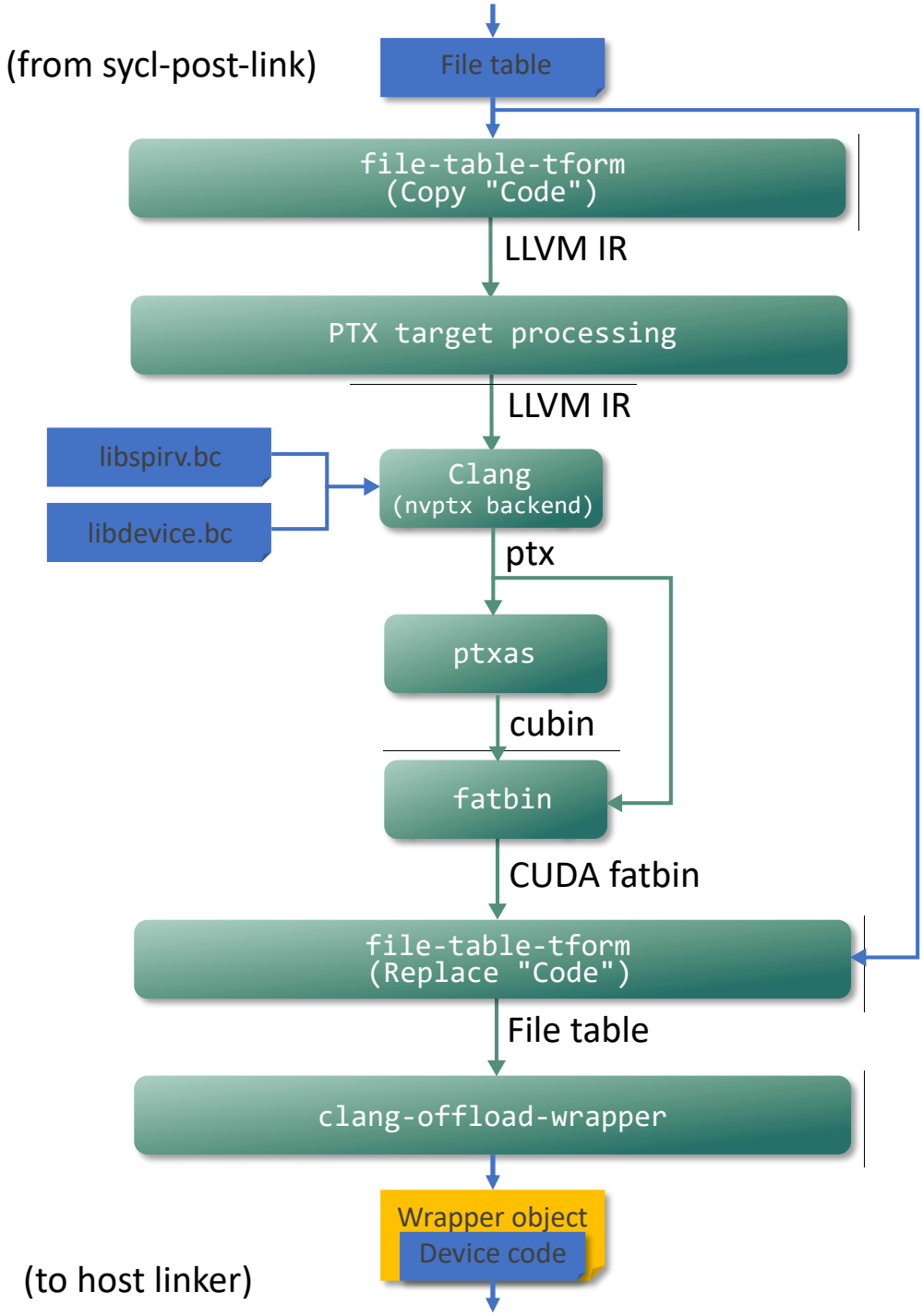
CUDA 支持

当`nvptx64-nvidia-cuda`传递给`-fsycl-targets`时，驱动程序支持编译成 NVPTX。

与其他 AOT 目标不同，从中间编译对象链接的位码模块永远不会通过 SPIR-V。相反，它以位码形式直接传递到 NVPTX 后端。所有生成的位码都依赖于两个库， `libdevice.bc`（由 CUDA SDK 提供）和`libspirv-nvptx64--nvidiacl.bc`变体（由 libclc 项目构建）。`libspirv-nvptx64--nvidiacl.bc`不直接使用。相反，它用于生成重组变体 `remangled-l64-signed_char.libspirv-nvptx64--nvidiacl.bc`和 `remangled-l32-signed_char.libspirv-nvptx64--nvidiacl.bc`来处理 Linux 和 Windows 之间的原始类型差异。

CUDA 的设备代码链接后步骤

在设备代码链接后的步骤中的“PTX 目标处理”部分，以CUDA 为目标的 llvm 位码对象在通用 `llvm-link`步骤期间链接在一起，然后使用`sycl-post-link`工具进行拆分。对于每个临时位码文件，都会调用 clang 以使临时文件链接 `libspirv-nvptx64--nvidiacl.bc`和`libdevice.bc`，并且使用 NVPTX 后端将链接生成的模块编译为 PTX。然后使用该`ptxas`工具（CUDA SDK 的一部分）将生成的 PTX 文件组装到一个 cubin 中。再使用`fatbinary`工具将PTX 文件和 cubin 组装在一起生成 CUDA fatbin。生成的 CUDA fatbins 会替换由`sycl-post-link` 生成文件表中的 llvm 位码文件。最后的结果表被传递给转存包装器工具。



检查编译器是否针对 NVPTX

当 SYCL 编译器处于设备模式并以 NVPTX 后端为目标时，编译器定义`__SYCL_DEVICE_ONLY__`和`__NVPTX__`宏。这个宏组合可以安全地用在 SYCL 内核中去启用 NVPTX 特定的代码路径。

注意：这些宏仅在设备编译阶段定义。

NVPTX 内置插件

内置函数在 libclc 中的 OpenCL C 中实现。OpenCL C 将`long`类型视为 64 位并且没有`long long`类型，而 Windows DPC++ 将`long`类型视为 32 位整数，将类型`long long`视为64 位整数。原始类型之间的差异可能导致应用程序使用不兼容的 libclc 内置程序。重组器创建多个具有不同重组函数名称的 libspriv 文件以支持 Windows 和 Linux。在构建针对 CUDA 后端的 SYCL 应用程序时， 如果主机目标是 Windows，驱动程序将链接

`remangled-l32-signed_char.libspirv-nvptx64--nvidiacl.bc` 设备代码，或者如果主机目标是 Linux，它将链接`remangled-l64-signed_char.libspirv-nvptx64--nvidiacl.bc`设备代码。

当 SYCL 编译器处于设备模式并以 NVPTX 后端为目标时，编译器会公开 clang 支持的 NVPTX 内置函数。

注意：这会启用其他目标或主机无法支持的 NVPTX 特定功能。

例子：

```
double my_min(double x, double y) {
#if defined(__NVPTX__) && defined(__SYCL_DEVICE_ONLY__)
  // Only available if in device mode and
  // while compiling for the NVPTX target.
  return __nvvm_fmin_d(x, y);
#else
  return x < y ? x : y;
#endif
}
```

本地内存支持

在 CUDA 中，用户只能分配一块主机分配的共享内存（映射到 SYCL 的本地访问器）。这块内存被分配为一个数组`extern __shared__ <type> <name>[];`，LLVM 将其表示为 CUDA 共享内存地址空间的外部全局符号。NVPTX 后端然后将其底层化为`.extern .shared .align 4 .b8`PTX 指令。

在 SYCL 中，用户可以分配多个本地访问器并将它们作为内核参数传递。当 SYCL 前端将 SYCL 内核调用底层化为符合 OpenCL 的内核条目时，它会将本地访问器底层化为指向 OpenCL 本地内存（CUDA 共享内存）的指针，但这对于 CUDA 内核是不合法的。

为了使 CUDA 的 SYCL 底层化合法，SYCL 针对 CUDA 特殊扫描将执行以下操作：

- 为 CUDA 共享内存地址空间创建一个全局符号
- 将所有指向 CUDA 共享内存的指针转换为一个 32 位整数，表示使用全局符号的字节偏移量
- 将所有通过地址使用的转换后的指针替换为作为参数传递的全局符号偏移量的整数值

例如，以下内核：

```
define void @SYCL_generated_kernel(i64 addrspace(3)* nocapture %local_ptr, i32
%arg, i64 addrspace(3)* nocapture %local_ptr2) {
  %0 = load i64, i64 addrspace(3)* %local_ptr
  %1 = load i64, i64 addrspace(3)* %local_ptr2
}
```

以 CUDA 为目标时转换为此内核：

```
@SYCL_generated_kernel.shared_mem = external dso_local local_unnamed_addr
addrspace(3) global [0 x i8], align 4
```

```
define void @SYCL_generated_kernel(i32 %local_ptr_offset, i32 %arg, i32
%local_ptr_offset2) {
  %new_local_ptr = getelementptr inbounds [0 x i8], [0 x i8] addrspace(3)*
@SYCL_generated_kernel.shared_mem, i32 0, i32 %local_ptr_offset
  %new_local_ptr2 = getelementptr inbounds [0 x i8], [0 x i8] addrspace(3)*
@SYCL_generated_kernel.shared_mem, i32 0, i32 %local_ptr_offset2
  %0 = load i32, i32 addrspace(3)* %new_local_ptr
  %1 = load i64, i64 addrspace(3)* %new_local_ptr2
}
```

在运行时方面，当设置本地内存参数时，CUDA PI 实现会在内部将参数设置为相对于已用本地内存的累积大小的偏移量。这种方法保留了现有的 PI 接口。

全局偏移支持

CUDA API 本身不支持 SYCL 期望的全局偏移参数。

为了模拟这一点并使生成的内核兼容，引入了一个内部函数的`llvm.nvvm.implicit.offset`（clang builtin `__builtin_ptx_implicit_offset`），将这个隐式参数用于 NVPTX 后端。内在函数返回一个指向`i32`3 元素数组的指针。

每个在调用图中使用到隐式偏移量的非内核函数都增加了一个指向`i32`的类型指针的额外隐式参数。调用那些使用了此内部函数的函数的内核被克隆：

- 原始内核将一个拥有3个`i32` 的数组初始化为 0，并将指向该数组的指针传递给每个带有隐式参数的函数；
- 克隆函数类型增加了类型为 3个`i32`数组 的隐式参数。然后将指向该数组的指针传递给每个带有隐式参数的函数。

运行时将查询两个内核并根据以下逻辑调用适当的内核：

- 如果存在这2个版本，且全局偏移量为0，则调用原始内核，否则将调用克隆的内核并通过值传递偏移量；
- 如果仅存在 1 个函数，则假定内核未使用此参数，因此忽略它。

例如，以下代码：

```
declare i32* @llvm.nvvm.implicit.offset()

define weak_odr dso_local i64 @other_function() {
  %1 = tail call i32* @llvm.nvvm.implicit.offset()
  %2 = getelementptr inbounds i32, i32* %1, i64 2
  %3 = load i32, i32* %2, align 4
  %4 = zext i32 %3 to i64
  ret i64 %4
}

define weak_odr dso_local void @other_function2() {
  ret
```

```
}

define weak_odr dso_local void @example_kernel() {
entry:
  %0 = call i64 @other_function()
  call void @other_function2()
  ret void
}
```

转换为了这样：

```
define weak_odr dso_local i64 @other_function(i32* %0) {
  %2 = getelementptr inbounds i32, i32* %0, i64 2
  %3 = load i32, i32* %2, align 4
  %4 = zext i32 %3 to i64

  ret i64 %4
}

define weak_odr dso_local void @example_kernel() {
entry:
  %0 = alloca [3 x i32], align 4
  %1 = bitcast [3 x i32]* %0 to i8*
  call void @llvm.memset.p0i8.i64(i8* nonnull align 4 dereferenceable(12) %1, i8
0, i64 12, i1 false)
  %2 = getelementptr inbounds [3 x i32], [3 x i32]* %0, i32 0, i32 0
  %3 = call i64 @other_function(i32* %2)
  call void @other_function2()
  ret void
}

define weak_odr dso_local void @example_kernel_with_offset([3 x i32]* byval([3 x
i32]) %0) {
entry:
  %1 = bitcast [3 x i32]* %0 to i32*
  %2 = call i64 @other_function(i32* %1)
  call void @other_function2()
  ret void
}
```

注意：内核命名目前还不是完全稳定的。

与 SPIR-V 格式集成

本节说明如何从 C++ 类和函数生成 SPIR-V 特定类型和操作。

将 SYCL C++ 程序转换为可在异构系统上执行的代码可以被认为是三个步骤：

1. 将 SYCL C++ 程序翻译成 LLVM IR
2. 从 LLVM IR 翻译到 SPIR-V 的

3. 从 SPIR-V 到机器码的翻译

LLVM-IR 到 SPIR-V 的转换由专用工具转换器 执行。该工具将大多数常规 LLVM IR 类型、操作等正确转换为 SPIR-V。

例如：

- 类型：`i32`→`OpTypeInt`
- 操作：`load`→`OpLoad`
- 调用： `call` → `OpFunctionCall`

SPIR-V 定义了在 LLVM IR 中没有对应等价的特殊内置类型和操作。例如

- 类型： ??? → `OpTypeEvent`
- 操作： ??? → `OpGroupAsyncCopy`

还支持从 LLVM IR 转换为特殊类型的 SPIR-V，但此类 LLVM IR 必须符合一些特殊要求。不幸的是，在 LLVM IR 中没有规范形式的特殊内置类型和操作，而且我们不能重用 OpenCL C 前端编译器生成的现有的表示。例如，以下是OpenCL C 前端编译器生成的`OpGroupAsyncCopy` 在LLVM IR 中的操作代码。

```
@_Z21async_work_group_copyPU3AS3fPU3AS1Kfjj(float addrspace(3)*, float
addrspace(1)*, i32, i32)
```

它是一个常规函数，可能与从 C++ 源代码生成的用户代码发生冲突。

DPC++ 编译器使用了以 OpenCL C++ 编译器为原型开发的修改解决方案：

- 编译器：https://github.com/KhronosGroup/SPIR/tree/spirv-1.1
- 头文件： https://github.com/KhronosGroup/libclcxx

我们的解决方案重用 OpenCL 数据类型，如采样器、事件、图像类型等，但我们使用不同的拼写来避免与 C++ 代码的潜在冲突。在 SYCL 模式下启用的 OpenCL 类型的拼写约定是：

```
__ocl_<OpenCL_type_name> // e.g. __ocl_sampler_t, __ocl_event_t
```

使用 OpenCL 类型的操作使用了本文档中描述的特殊命名约定。该解决方案使我们避免了 SPIR-V 转换器中的 SYCL 特化，并且充分利用了为 OpenCL 类型开发的 clang 基础设施。

没有 LLVM 等效项的 SPIR-V 操作在头文件中**声明**（但未定义）满足以下要求：

- 该操作在 C++ 中表示为不引发 C++ 异常的`extern`函数
- 该操作不能在 C++ 程序中具有实际定义

例如，以下 C++ 代码被成功识别并转换为 SPIR-V 操作`OpGroupAsyncCopy`：

```
template <typename dataT>
extern __ocl_event_t
__spirv_OpGroupAsyncCopy(int32_t Scope, __local dataT *Dest,
```

```
                          __global dataT *Src, size_t NumElements,
                          size_t Stride, __ocl_event_t E) noexcept;

__ocl_event_t e =
  __spirv_OpGroupAsyncCopy(cl::__spirv::Scope::Workgroup,
                           dst, src, numElements, 1, E);
```

关于使用 SPIR-V 特殊类型和操作的一些细节和协议

SPIR-V 特定的 C++ 枚举类型和类在文件中声明： `sycl/include/CL/__spirv/spirv_types.hpp`.

SPIR-V 特定的 C++ 函数声明位于文件中： `sycl/include/CL/__spirv/spirv_ops.hpp`.

在 SYCL 主机设备上 SPIR-V 特定函数的 实现： `sycl/source/spirv_ops.cpp`.

地址空间处理

SYCL 规范使用 C++ 类来表示指向加速器上不相交内存区域的指针，以支持使用标准 C++ 工具链和 SYCL 编译器工具链进行编译。

例如：

```
// check that SYCL mode is ON and we can use non-standard decorations
#if defined(__SYCL_DEVICE_ONLY__)
// GPU/accelerator implementation
template <typename T, address_space AS> class multi_ptr {
  // DecoratedType applies corresponding address space attribute to the type T
  // DecoratedType<T, global_space>::type == "__attribute__((opencl_global)) T"
  // See sycl/include/CL/sycl/access/access.hpp for more details
  using pointer_t = typename DecoratedType<T, AS>::type *;

  pointer_t m_Pointer;
  public:
  pointer_t get() { return m_Pointer; }
  T& operator* () { return *reinterpret_cast<T*>(m_Pointer); }
}
#else
// CPU/host implementation
template <typename T, address_space AS> class multi_ptr {
  T *m_Pointer; // regular undecorated pointer
  public:
  T *get() { return m_Pointer; }
  T& operator* () { return *m_Pointer; }
}
#endif
```

根据编译器模式决定`multi_ptr`是否使用地址空间属性装饰内部数据。

SYCL 模式与 OpenCL 的主要地址空间语义差异是 SYCL 不会将 OpenCL 通用地址空间分配给没有显式地址空间属性的声明类型。与其他基于单源 C++ 的 GPU 编程模式（如 OpenMP/CUDA/HIP）类似，SYCL 对没有地址空

间属性的类型使用 clang 的"默认"地址空间。在底层化到 LLVM IR 期间，默认地址空间被映射到 SPIR 通用地址空间。声明根据其声明上下文分配给相关的内存区域，并且指向它们的指针被强制转换为通用的。这种设计有两个重要特点：一方面保持类型系统与 C++ 一致，另一方面可以启用用于生成和 SPIR 内存模型（和其他 GPU 目标）对齐的设备代码的工具。

所以在函数内部，这个变量声明：

```
int var;
```

DPC++变成

```
VarDecl  var 'int'
```

OpenCL 编译器变成

```
VarDecl  var '__private int'
```

更改变量类型在 C++ 中具有巨大的破坏性影响。例如，以下代码不能在 OpenCL 模式的C++ 中编译：

```
template<typename T1, typename T2>
struct is_same {
    static constexpr int value = 0;
};

template<typename T>
struct is_same<T, T> {
    static constexpr int value = 1;
};

void foo(int p) {
    static_assert(is_same<decltype(p), int>::value, "int is not an int?"); //
Fails: p is '__private int' != 'int'
    static_assert(is_same<decltype(&p), int*>::value, "int* is not an int*?");  //
Fails: p is '__private int*' != '__generic int*'
}
```

为了利用现有 clang 的功能，我们在 SYCL 模式下重用以下 OpenCL 地址空间属性：

Address space attribute	SYCL address_space enumeration
`__attribute__((opencl_global))`	global_space, constant_space
`__attribute__((opencl_global_host))`	ext_intel_global_host_space
`__attribute__((opencl_global_device))`	ext_intel_global_device_space

Address space attribute	SYCL address_space enumeration
`__attribute__((opencl_local))`	local_space
`__attribute__((opencl_private))`	private_space
`__attribute__((opencl_constant))`	N/A

> **注意**：虽然 SYCL 设备编译器支持 `__attribute__((opencl_constant))`，但此属性的使用仅限于 SYCL 实现。OpenCL 常量指针不能转换为具有任何其他地址空间（包括默认值）的指针。

编译器/运行时接口

DPC++ 运行时架构

待定

SYCL 的 DPC++ 语言扩展

语言扩展列表可以在扩展中找到