Clang CUDA前端分析

软件所智能软件中心PLCT实验室 邹小芳

2023/06/10

目录

01 CUDA 介绍

- **CUDA**（Compute Unified Device Architecture的缩写）

  是由Nvidia创建的并行计算平台和应用程序编程接口（API）。

  它允许软件开发人员使用支持CUDA的图形处理单元（GPU）进行通用处理，这种方法称为GPGPU（general-purpose computing on graphics processing units）。

  CUDA附带了一个软件环境，允许开发人员使用C++作为高级编程语言。CUDA C/C++程序的标准编译包括将运行在设备上的函数编译成一个虚拟的ISA格式，称为PTX。然后，PTX代码被编译到称为SASS（Shader ASSembler）的低级机器指令集，该指令集在NVIDIA GPU硬件上执行。

02 基于LLVM的CUDA编译流程

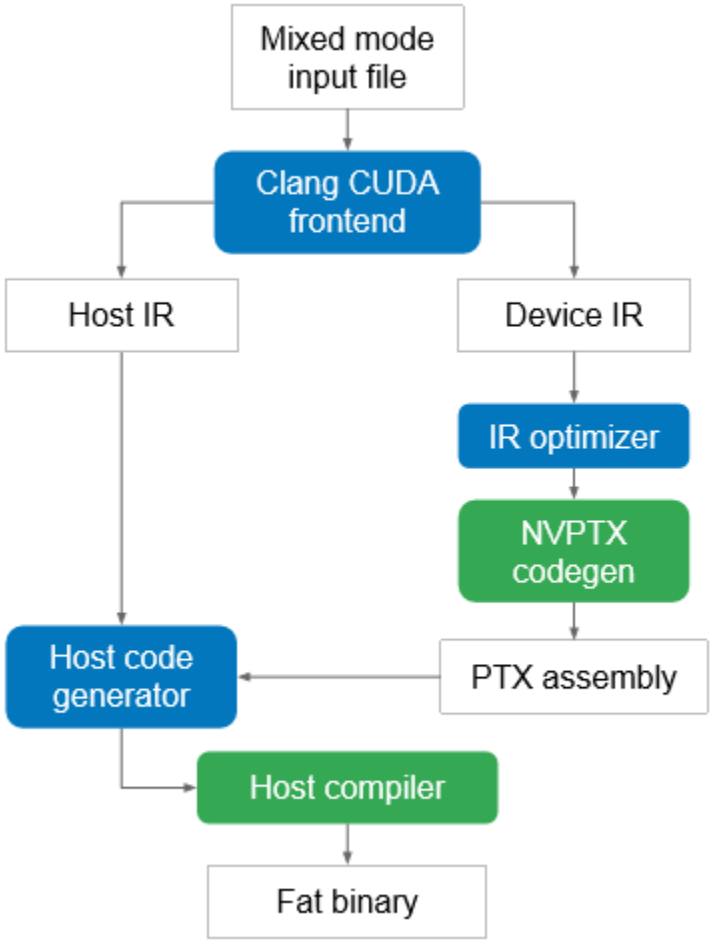


Image from: http://wujingyue.github.io/docs/gpucc-talk.pdf

03 Compiling CUDA code

$clang++ axpy.cu -o axpy -L<CUDA install path>/<lib64 or lib> --cuda-gpu-arch=<GPU arch> -lcudart_static -ldl -lrt –pthread

$ ./axpy

y[0] = 2

y[1] = 4

y[2] = 6

y[3] = 8

- ✓ <CUDA install path> CUDA SDK安装目录
- ✓ <GPU arch> GPU的计算能力。例如，如果要在计算能力为8的GPU上运行程序，请指定--cuda-gpu-arch=sm_80。

```cpp
// axpy.cu
#include <iostream>
__global__ void axpy(float a, float* x, float* y) {
  y[threadIdx.x] = a * x[threadIdx.x];
}
int main(int argc, char* argv[]) {
  const int kDataLen = 4;

  float a = 2.0f;
  float host_x[kDataLen] = {1.0f, 2.0f, 3.0f, 4.0f};
  float host_y[kDataLen];

  // Copy input data to device.
  float* device_x;
  float* device_y;
  cudaMalloc(&device_x, kDataLen * sizeof(float));
  cudaMalloc(&device_y, kDataLen * sizeof(float));
  cudaMemcpy(device_x, host_x, kDataLen * sizeof(float),
             cudaMemcpyHostToDevice);

  // Launch the kernel.
  axpy<<<1, kDataLen>>>(a, device_x, device_y);

  // Copy output data to host.
  cudaDeviceSynchronize();
  cudaMemcpy(host_y, device_y, kDataLen * sizeof(float),
             cudaMemcpyDeviceToHost);

  // Print the results.
  for (int i = 0; i < kDataLen; ++i) {
    std::cout << "y[" << i << "] = " << host_y[i] << "\n";
  }

  cudaDeviceReset();
  return 0;
}
```

03 Compiling CUDA code

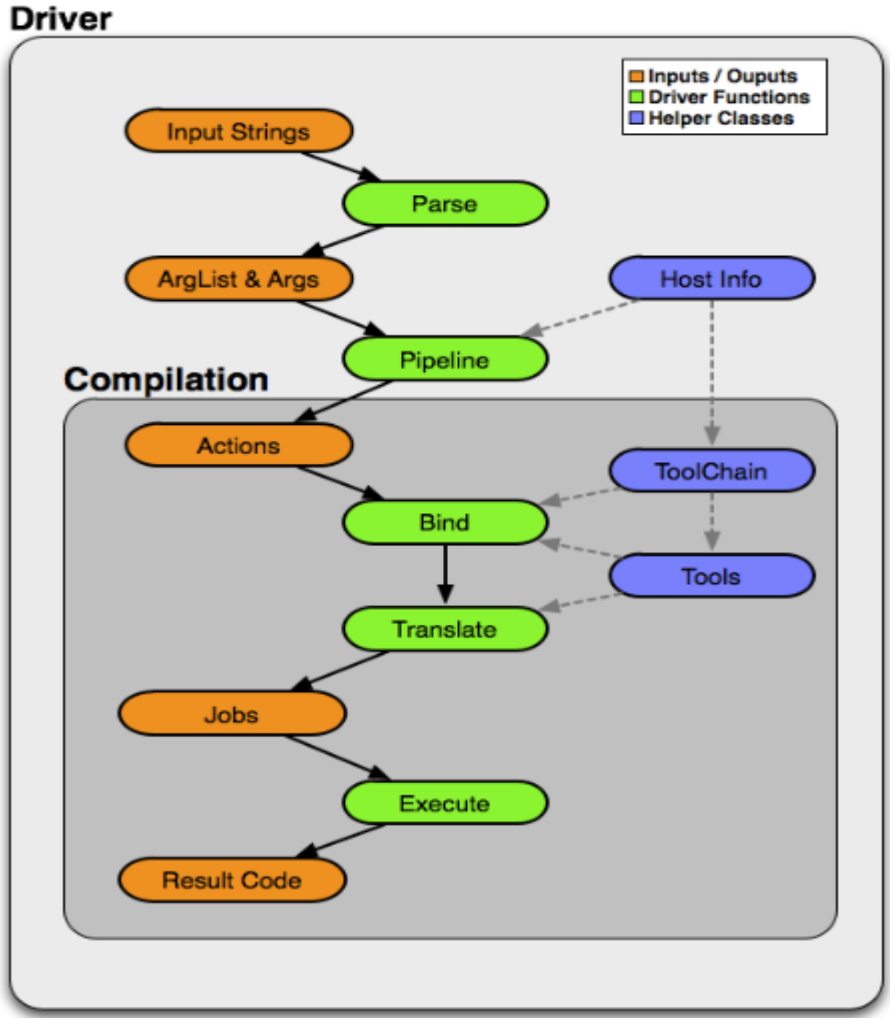
- **完整编译步骤如下：**

1. 使用clang，处理.cu源文件，生成ptx文件。
2. 使用ptxas来处理上一步输出的ptx文件，生成SASS代码。
3. 使用fatbinary来处理步骤1和2输出的文件，生成一个单独的fat binary 文件。
4. 使用clang再次处理.cu源文件，生成host代码。步骤3得到的fat binary也会作为输入传给clang，作为一个特殊的 ELF section。
5. 使用gnu链接器，步骤4的输出作为输入，链接生成可执行文件。

04 Clang Driver

- Clang Driver 是一个产品级的编译器驱动程序，提供对Clang编译器和工具的访问，并具有与gcc驱动程序兼容的命令行接口。有如下特征：
  - GCC Compatibility
  - Flexible
  - Low Overhead
  - Simple

- Clang Dirver功能阶段：
  - Parse: Option Parsing
  - Pipeline: Compilation Action Construction
  - Bind: Tool & Filename Selection
  - Translate: Tool Specific Argument Translation
  - Execute



04 Clang Driver

命令行选项

- **CUDA相关**

-Xarch_host <arg>： Pass <arg> to the CUDA/HIP host compilation

-Xarch_device <arg>：Pass <arg> to the CUDA/HIP device compilation

-Xcuda-fatbinary <arg>：Pass <arg> to fatbinary invocation

-Xcuda-ptxas <arg>：Pass <arg> to the ptxas assembler

--cuda-device-only： Compile CUDA code for device only

--cuda-host-only： Compile CUDA code for host only. Has no effect on non-CUDA

--cuda-compile-host-device：Compile CUDA code for both host and device (default). Has no effect on non-CUDA compilations.

--cuda-include-ptx=<value>：Include PTX for the following GPU architecture (e.g. sm_35) or 'all'. May be specified more than once.

--no-cuda-include-ptx=<value>：Do not include PTX for the following GPU architecture (e.g. sm_35) or 'all'. May be specified more than once.

--offload-arch=<value>：CUDA offloading device architecture (e.g. sm_35), or HIP offloading target ID in the form of a device architecture followed by target ID features delimited by a colon.

--cuda-gpu-arch=<value>: --offload-arch=<value> alias

……

参考文件：clang/include/clang/Driver/Options.td

04 Clang Driver

Driver type

- **Driver type 定义**

  TYPE(NAME，ID，PP_TYPE，TEMP_SUFFIX，FLAGS)

  NAME：字符串形式的类型名

  ID：类型ID，会产生一个 clang::driver::types::TY_XX枚举常量

  PP_TYPE：该类型的预处理输入的类型id

  TEMP_SUFFIX：当创建此typel临时文件时使用的后缀

  FLAGS：每种类型的phases列表

- **CUDA相关**

  TYPE("cuda-cpp-output", PP_CUDA, INVALID, "cui", phases::Compile, phases::Backend, phases::Assemble, phases::Link)

  TYPE("cuda", CUDA, PP_CUDA, "cu", phases::Preprocess, phases::Compile, phases::Backend, phases::Assemble, phases::Link)

  TYPE("cuda", CUDA_DEVICE，PP_CUDA, "cu", phases::Preprocess, phases::Compile, phases::Backend, phases::Assemble, phases::Link)

04 Clang Driver

工具链

- **Clang & 工具链**

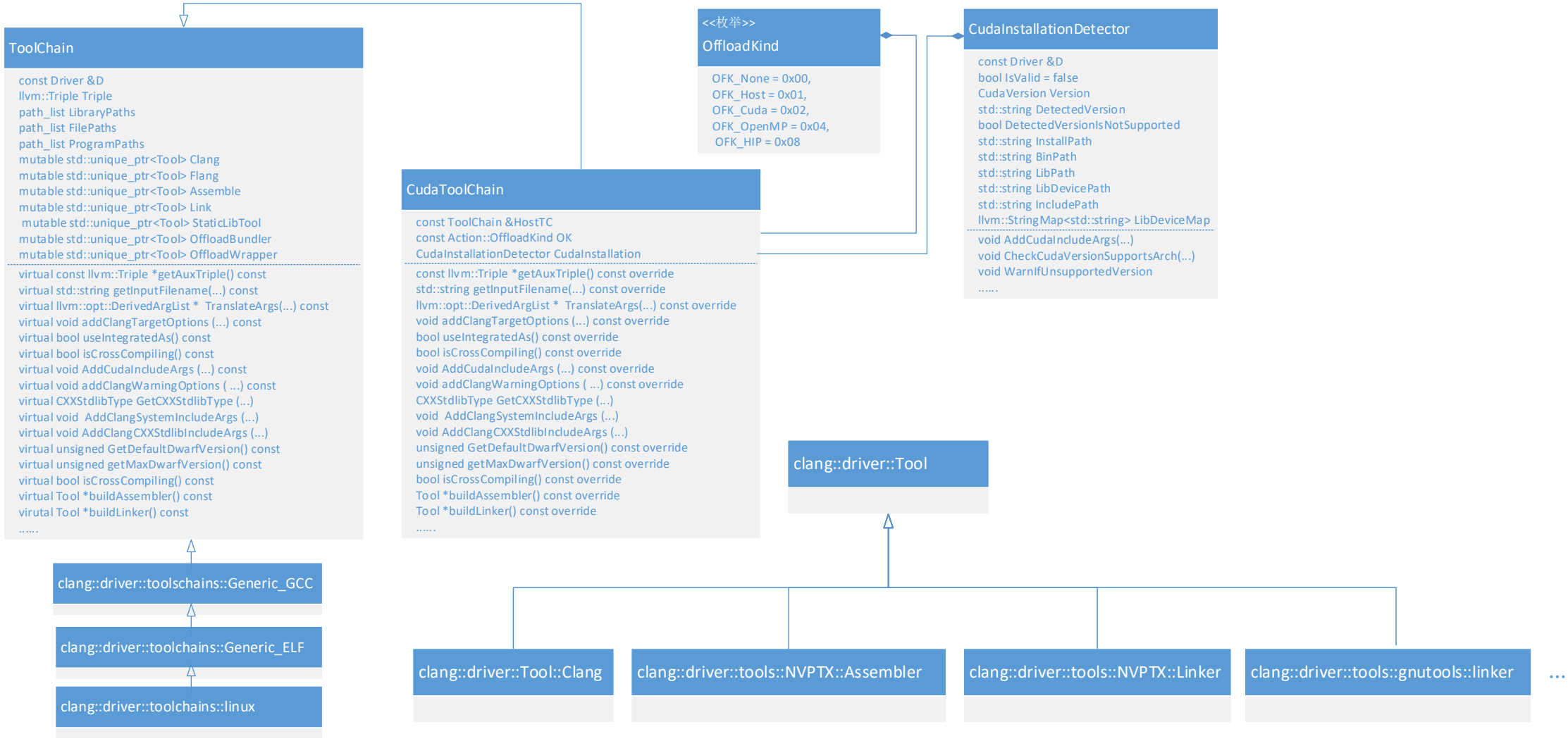
Clang只是C族程序设计语言完整工具链的一个组成部分。为了组装一个完整的工具链，需要额外的工具和运行库。Clang被设计为与目标平台的现有工具和库进行互操作，LLVM项目为这些组件提供了选择方案。

工具链负责选择要执行特定操作的工具。驱动程序与工具链交互以执行工具绑定。每个工具链都包含有关编译特定体系结构、平台和操作系统所需的所有工具的信息。

CUDA编译总是需要两个工具链，CUDA工具链和Host工具链。

04 Clang Driver

Tool & ToolChain



https://clang.llvm.org/doxygen/classclang_1_1driver_1_1ToolChain.html

05 Frontend

语言标准及宏定义

- **语言标准**

(clang/include/clang/Basic/LangStandards.def)
LANGSTANDARD(cuda, "cuda", CUDA, "NVIDIA CUDA(tm)", LineComment | CPlusPlus | Digraphs)

```
enum class Language : uint8_t {
  Unknown,
  Asm,
  LLVM_IR,
  C,
  CXX,
  ObjC,
  ObjCXX,
  OpenCL,
  CUDA,
  RenderScript,
  HIP
}
```

- **宏定义**

__CUDA__
__CUDA_ARCH__
__CLANG_CUDA_APPROX_TRANSCENDENTALS__

05 Frontend

词法分析 & 语法分析

- **词法分析**

  词法分析将字符序列转换成token流。

  CUDA相关部分：

  1. 分别把<<<和>>>字符分别转成tok::lesslessless和tok:: greatergreatergreater 。
  2. 允许f16 literals 以避免错误的警报。因为CUDA host和device可能有不同的_Float16支持。

- **语法分析**

  语法分析对Token流进行分析。

  CUDA相关部分：

  1. 分析tok::lesslessless和tok:: greatergreatergreater间的tokens，并调用其Sema成员的接口，生成CUDAKernelCallExpr语句。
  2. 解析CUDA特定语言的pragma。

05 Frontend

抽象语法树

- **抽象语法树（AST）**

抽象语法树是一种层次化的程序表示法，根据程序语言的语法来表示源代码结构，每个AST节点对应一个源代码项。clang中，有关翻译单元的AST的所有信息都捆绑在ASTContext类中。

通过-ast-dump选项dump AST，示例：

```
// test.cc
int addi(int a, int b) {
  int result = a+b;
  return result;
}
```

```
$ clang -Xclang -ast-dump -fsyntax-only test.cc
TranslationUnitDecl 0x565349ef6d78 <<invalid sloc>> <invalid sloc>
... cutting out internal declarations of clang ...
`-FunctionDecl 0x565349f38680 <./test.cc:1:1, line:4:1> line:1:5 addi 'int (int, int)'
  |-ParmVarDecl 0x565349f38528 <col:10, col:14> col:14 used a 'int'
  |-ParmVarDecl 0x565349f385a8 <col:17, col:21> col:21 used b 'int'
  `-CompoundStmt 0x565349f388e8 <col:24, line:4:1>
    |-DeclStmt 0x565349f38888 <line:2:3, col:19>
    | `-VarDecl 0x565349f38790 <col:3, col:18> col:7 used result 'int' cinit
    |   `-BinaryOperator 0x565349f38868 <col:16, col:18> 'int' '+'
    |     |-ImplicitCastExpr 0x565349f38838 <col:16> 'int' <LValueToRValue>
    |     | `-DeclRefExpr 0x565349f387f8 <col:16> 'int' lvalue ParmVar 0x565349f38528 'a' 'int'
    |     `-ImplicitCastExpr 0x565349f38850 <col:18> 'int' <LValueToRValue>
    |       `-DeclRefExpr 0x565349f38818 <col:18> 'int' lvalue ParmVar 0x565349f385a8 'b' 'int'
    `-ReturnStmt 0x565349f388d8 <line:3:3, col:10>
      `-ImplicitCastExpr 0x565349f388c0 <col:10> 'int' <LValueToRValue>
        `-DeclRefExpr 0x565349f388a0 <col:10> 'int' lvalue Var 0x565349f38790 'result' 'int'
```

- 与CUDA相关部分，涉及到类型，声明，表达式，打印信息等。

05 Frontend

语义分析

- **语义分析模块**

语义分析模块主要用于分析程序语义，做包括类型检查在内的各种语义检查、创建AST节点。这部分主要是要根据CUDA语言规范文档中的条款/描述来进行相应的语义检查。

- **CUDA相关：**

1. CUDA数据结构
2. 类型
3. 声明
4. 表达式
5. 语句
6. C++重载
7. C++模板
8. Lambda
9. CUDA语义
10. 异常处理

```
enum CUDAFunctionTarget {
  CFT_Device,
  CFT_Global,
  CFT_Host,
  CFT_HostDevice,
  CFT_InvalidTarget
};
```

```
enum class FunctionEmissionStatus {
  Emitted,
  CUDADiscarded,     // Discarded due to CUDA/HIP hostness
  OMPDiscarded,      // Discarded due to OpenMP hostness
  TemplateDiscarded, // Discarded due to uninstantiated templates
  Unknown,
};
```

```
enum CUDAFunctionPreference {
  CFP_Never,      // Invalid caller/callee combination.
  CFP_WrongSide,  // Calls from host-device to host or device function that do not match current compilation mode.
  CFP_HostDevice, // Any calls to host/device functions.
  CFP_SameSide,   // Calls from host-device to host or device function matching current compilation mode.
  CFP_Native,     // host-to-host or device-to-device calls.
};
```

05 Frontend

代码生成

- **中间代码生成模块**

中间代码生成模块用来将翻译单元（translation unit）的抽象语法树（AST）转换成LLVM中间表示（IR）。代码生成的主要流程是遍历AST节点（Decl、Expr、Stmt、Type等），针对每个节点生成对应LLVM IR。

- **CUDA相关（lib/CodeGen 下）：**
  1. 模块
  2. 函数
  3. 类型
  4. 声明
  5. 表达式
  6. 语句
  7. 函数调用约定
  8. Target信息
  9. 运行时
  10. 其他

```
%call4 = call i32 @__cudaPushCallConfiguration(i64 %7, i32 %9, i64 %13, i32 %15, i64 0, i8* null)
  %tobool = icmp ne i32 %call4, 0
  br i1 %tobool, label %kcall.end, label %kcall.configok

kcall.configok:                                   ; preds = %entry
  %16 = load float, float* %a, align 4
  %17 = load float*, float** %device_x, align 8
  %18 = load float*, float** %device_y, align 8
  call void @_Z19__device_stub__axpyfPfS_(float %16, float* %17, float* %18)
  br label %kcall.end

kcall.end:                                        ; preds = %kcall.configok, %entry
  %call5 = call i32 @cudaDeviceSynchronize()
  %arraydecay6 = getelementptr inbounds [4 x float], [4 x float]* %host_y, i64 0, i64 0
  %19 = bitcast float* %arraydecay6 to i8*
  %20 = load float*, float** %device_y, align 8
  %21 = bitcast float* %20 to i8*
  %call7 = call i32 @cudaMemcpy(i8* %19, i8* %21, i64 16, i32 2)
  store i32 0, i32* %i, align 4
  br label %for.cond
```

生成的代码片段

06 运行时头文件

- **Clang/lib/Headers下的运行时头文件**

  主要用来实现运行时头文件的封装，会被安装到工具链的固定路径下。

  有些头文件是提供给用户使用的，比如intrinsic API头文件，在写应用程序的时候，可以使用include将头文件包含进来；有些头文件是提供给编译器使用的，这些文件的名字一般是由下划线开头。

- **Cuda头文件**
  - ✓ lib/Headers/__clang_hip_runtime_wrapper.h
  - ✓ lib/Headers/__clang_cuda_builtin_vars.h
  - ✓ lib/Headers/__clang_cuda_libdevice_declares.h
  - ✓ lib/Headers/__clang_cuda_device_functions.h
  - ✓ lib/Headers/__clang_cuda_math.h
  - ✓ lib/Headers/__clang_cuda_cmath.h
  - ✓ lib/Headers/__clang_cuda_math_forward_declares.h
  - ✓ lib/Headers/__clang_cuda_intrinsics.h
  - ✓ lib/Headers/__clang_cuda_complex_builtins.h

07 参考资料

https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
http://wujingyue.github.io/docs/gpucc-talk.pdf
https://llvm.org/docs/CompileCudaWithLLVM.html
https://clang.llvm.org/docs/DriverInternals.html
https://clang.llvm.org/docs/IntroductionToTheClangAST.html

谢 谢

欢迎交流合作

2023/06/10