Compiling CUDA with clang

Introduction

This document describes how to compile CUDA code with clang, and gives some details about LLVM and clang's CUDA implementations.

This document assumes a basic familiarity with CUDA. Information about CUDA programming can be found in the CUDA programming guide.

Compiling CUDA Code

Prerequisites

CUDA is supported since llvm 3.9. Clang currently supports CUDA 7.0 through 11.5. If clang detects a newer CUDA version, it will issue a warning and will attempt to use detected CUDA SDK it as if it were CUDA 11.5.

Before you build CUDA code, you'll need to have installed the CUDA SDK. See NVIDIA's CUDA installation guide for details. Note that clang maynot support the CUDA toolkit as installed by some Linux package managers. Clang does attempt to deal with specific details of CUDA installation on a handful of common Linux distributions, but in general the most reliable way to make it work is to install CUDA in a single directory from NVIDIA's *.run* package and specify its location via *-cuda-path=...* argument.

CUDA compilation is supported on Linux. Compilation on MacOS and Windows may or may not work and currently have no maintainers.

Invoking clang

Invoking clang for CUDA compilation works similarly to compiling regular C++. You just need to be aware of a few additional flags.

You can use this program as a toy example. Save it as `axpy.cu`. (Clang detects that you're compiling CUDA code by noticing that your filename ends with `.cu`. Alternatively, you can pass `-x cuda`.)

To build and run, run the following commands, filling in the parts in angle brackets as described below:

```
$ clang++ axpy.cu -o axpy --cuda-gpu-arch=<GPU arch> \
    -L<CUDA install path>/<lib64 or lib>             \
    -lcudart_static -ldl -lrt -pthread
$ ./axpy
y[0] = 2
y[1] = 4
y[2] = 6
y[3] = 8
```

On MacOS, replace *`-lcudart_static`* with *`-lcudart`*; otherwise, you may get "CUDA driver version is insufficient for CUDA runtime version" errors when you run your program.

- `<CUDA install path>` – the directory where you installed CUDA SDK. Typically, `/usr/local/cuda`.

  Pass e.g. `-L/usr/local/cuda/lib64` if compiling in 64-bit mode; otherwise, pass e.g. `-L/usr/local/cuda/lib`. (In CUDA, the device code and host code always have the same pointer widths, so if you're compiling 64-bit code for the host, you're also compiling 64-bit code for the device.) Note that as of v10.0 CUDA SDK no longer supports compilation of 32-bit applications.

- `<GPU arch>` – the compute capability of your GPU. For example, if you want to run your program on a GPU with compute capability of 3.5, specify `--cuda-gpu-arch=sm_35`.

  Note: You cannot pass `compute_XX` as an argument to `--cuda-gpu-arch`; only `sm_XX` is currently supported. However, clang always includes PTX in its binaries, so e.g. a binary compiled with `--cuda-gpu-arch=sm_30` would be forwards-compatible with e.g. `sm_35` GPUs.

  You can pass `--cuda-gpu-arch` multiple times to compile for multiple archs.

The *`-L`* and *`-l`* flags only need to be passed when linking. When compiling, you may also need to pass `--cuda-path=/path/to/cuda` if you didn't install the CUDA SDK into `/usr/local/cuda` or `/usr/local/cuda-X.Y`.

Flags that control numerical code

If you're using GPUs, you probably care about making numerical code run fast. GPU hardware allows for more control over numerical operations than most CPUs, but this results in more compiler options for you to juggle.

Flags you may wish to tweak include:

- `-ffp-contract={on,off,fast}` (defaults to `fast` on host and device when compiling CUDA) Controls whether the compiler emits fused multiply-add operations.
  - `off`: never emit fma operations, and prevent ptxas from fusing multiply and add instructions.
  - `on`: fuse multiplies and adds within a single statement, but never across statements (C11 semantics). Prevent ptxas from fusing other multiplies and adds.
  - `fast`: fuse multiplies and adds wherever profitable, even across statements. Doesn't prevent ptxas from fusing additional multiplies and adds.

  Fused multiply-add instructions can be much faster than the unfused equivalents, but because the intermediate result in an fma is not rounded, this flag can affect numerical code.
- `-fcuda-flush-denormals-to-zero` (default: off) When this is enabled, floating point operations may flush denormal inputs and/or outputs to 0. Operations on denormal numbers are often much slower than the same operations on normal numbers.
- `-fcuda-approx-transcendentals` (default: off) When this is enabled, the compiler may emit calls to faster, approximate versions of transcendental functions, instead of using the slower, fully IEEE-compliant versions. For example, this flag allows clang to emit the ptx `sin.approx.f32` instruction.

  This is implied by `-ffast-math`.

Standard library support

In clang and nvcc, most of the C++ standard library is not supported on the device side.

`<math.h>` and `<cmath>`

In clang, `math.h` and `cmath` are available and pass tests adapted from libc++'s test suite.

In nvcc `math.h` and `cmath` are mostly available. Versions of `::foof` in namespace std (e.g. `std::sinf`) are not available, and where the standard calls for overloads that take integral arguments, these are usually not available.

```
#include <math.h>
#include <cmath.h>

// clang is OK with everything in this function.
__device__ void test() {
  std::sin(0.); // nvcc - ok
  std::sin(0);  // nvcc - error, because no std::sin(int) override is available.
  sin(0);       // nvcc - same as above.

  sinf(0.);       // nvcc - ok
```

```
  std::sinf(0.);  // nvcc - no such function
}
```

<std::complex>

nvcc does not officially support `std::complex`. It's an error to use `std::complex` in `__device__` code, but it often works in `__host__ __device__` code due to nvcc's interpretation of the "wrong-side rule" (see below). However, we have heard from implementers that it's possible to get into situations where nvcc will omit a call to an `std::complex` function, especially when compiling without optimizations.

As of 2016-11-16, clang supports `std::complex` without these caveats. It is tested with libstdc++ 4.8.5 and newer, but is known to work only with libc++ newer than 2016-11-16.

<algorithm>

In C++14, many useful functions from `<algorithm>` (notably, `std::min` and `std::max`) become constexpr. You can therefore use these in device code, when compiling with clang.

Detecting clang vs NVCC from code

Although clang's CUDA implementation is largely compatible with NVCC's, you may still want to detect when you're compiling CUDA code specifically with clang.

This is tricky, because NVCC may invoke clang as part of its own compilation process! For example, NVCC uses the host compiler's preprocessor when compiling for device code, and that host compiler may in fact be clang.

When clang is actually compiling CUDA code – rather than being used as a subtool of NVCC's – it defines the `__CUDA__` macro. `__CUDA_ARCH__` is defined only in device mode (but will be defined if NVCC is using clang as a preprocessor). So you can use the following incantations to detect clang CUDA compilation, in host and device modes:

```
#if defined(__clang__) && defined(__CUDA__) && !defined(__CUDA_ARCH__)
// clang compiling CUDA code, host mode.
#endif

#if defined(__clang__) && defined(__CUDA__) && defined(__CUDA_ARCH__)
// clang compiling CUDA code, device mode.
#endif
```

Both clang and nvcc define `__CUDACC__` during CUDA compilation. You can detect NVCC specifically by looking for `__NVCC__`.

Dialect Differences Between clang and nvcc

There is no formal CUDA spec, and clang and nvcc speak slightly different dialects of the language. Below, we describe some of the differences.

This section is painful; hopefully you can skip this section and live your life blissfully unaware.

Compilation Models

Most of the differences between clang and nvcc stem from the different compilation models used by clang and nvcc. nvcc uses *split compilation*, which works roughly as follows:

- Run a preprocessor over the input `.cu` file to split it into two source files: `H`, containing source code for the host, and `D`, containing source code for the device.
- For each GPU architecture `arch` that we're compiling for, do:
  - Compile `D` using nvcc proper. The result of this is a `ptx` file for `P_arch`.
  - Optionally, invoke `ptxas`, the PTX assembler, to generate a file, `S_arch`, containing GPU machine code (SASS) for `arch`.
- Invoke `fatbin` to combine all `P_arch` and `S_arch` files into a single "fat binary" file, `F`.
- Compile `H` using an external host compiler (gcc, clang, or whatever you like). `F` is packaged up into a header file which is force-included into `H`; nvcc generates code that calls into this header to e.g. launch kernels.

clang uses *merged parsing*. This is similar to split compilation, except all of the host and device code is present and must be semantically-correct in both compilation steps.

- For each GPU architecture `arch` that we're compiling for, do:
  - Compile the input `.cu` file for device, using clang. `__host__` code is parsed and must be semantically correct, even though we're not generating code for the host at this time.

    The output of this step is a `ptx` file `P_arch`.
  - Invoke `ptxas` to generate a SASS file, `S_arch`. Note that, unlike nvcc, clang always generates SASS code.
- Invoke `fatbin` to combine all `P_arch` and `S_arch` files into a single fat binary file, `F`.
- Compile `H` using clang. `__device__` code is parsed and must be semantically correct, even though we're not generating code for the device at this time.

  `F` is passed to this compilation, and clang includes it in a special ELF section, where it can be found by tools like `cuobjdump`.

(You may ask at this point, why does clang need to parse the input file multiple times? Why not parse it just once, and then use the AST to generate code for the host and each device architecture?

Unfortunately this can't work because we have to define different macros during host compilation and during device compilation for each GPU architecture.)

clang's approach allows it to be highly robust to C++ edge cases, as it doesn't need to decide at an early stage which declarations to keep and which to throw away. But it has some consequences you should be aware of.

Overloading Based on `__host__` and `__device__` Attributes

Let "H", "D", and "HD" stand for "`__host__` functions", "`__device__` functions", and "`__host__ __device__` functions", respectively. Functions with no attributes behave the same as H.

nvcc does not allow you to create H and D functions with the same signature:

```
// nvcc: error - function "foo" has already been defined
__host__ void foo() {}
__device__ void foo() {}
```

However, nvcc allows you to "overload" H and D functions with different signatures:

```
// nvcc: no error
__host__ void foo(int) {}
__device__ void foo() {}
```

In clang, the `__host__` and `__device__` attributes are part of a function's signature, and so it's legal to have H and D functions with (otherwise) the same signature:

```
// clang: no error
__host__ void foo() {}
__device__ void foo() {}
```

HD functions cannot be overloaded by H or D functions with the same signature:

```
// nvcc: error - function "foo" has already been defined
// clang: error - redefinition of 'foo'
__host__ __device__ void foo() {}
__device__ void foo() {}

// nvcc: no error
// clang: no error
__host__ __device__ void bar(int) {}
__device__ void bar() {}
```

When resolving an overloaded function, clang considers the host/device attributes of the caller and callee. These are used as a tiebreaker during overload resolution. See IdentifyCUDAPreference for the full set of rules, but at a high level they are:

- D functions prefer to call other Ds. HDs are given lower priority.
- Similarly, H functions prefer to call other Hs, or `__global__` functions (with equal priority). HDs are given lower priority.
- HD functions prefer to call other HDs.

  When compiling for device, HDs will call Ds with lower priority than HD, and will call Hs with still lower priority. If it's forced to call an H, the program is malformed if we emit code for this HD function. We call this the "wrong-side rule", see example below.

  The rules are symmetrical when compiling for host.

Some examples:

```
__host__ void foo();
__device__ void foo();

__host__ void bar();
```

```
__host__ __device__ void bar();

__host__ void test_host() {
  foo();  // calls H overload
  bar();  // calls H overload
}

__device__ void test_device() {
  foo();  // calls D overload
  bar();  // calls HD overload
}

__host__ __device__ void test_hd() {
  foo();  // calls H overload when compiling for host, otherwise D overload
  bar();  // always calls HD overload
}
```

Wrong-side rule example:

```
__host__ void host_only();

// We don't codegen inline functions unless they're referenced by a
// non-inline function.  inline_hd1() is called only from the host side, so
// does not generate an error.  inline_hd2() is called from the device side,
// so it generates an error.
inline __host__ __device__ void inline_hd1() { host_only(); }  // no error
inline __host__ __device__ void inline_hd2() { host_only(); }  // error

__host__ void host_fn() { inline_hd1(); }
__device__ void device_fn() { inline_hd2(); }

// This function is not inline, so it's always codegen'ed on both the host
// and the device.  Therefore, it generates an error.
__host__ __device__ void not_inline_hd() { host_only(); }
```

For the purposes of the wrong-side rule, templated functions also behave like `inline` functions: They aren't codegen'ed unless they're instantiated (usually as part of the process of invoking them).

clang's behavior with respect to the wrong-side rule matches nvcc's, except nvcc only emits a warning for `not_inline_hd`; device code is allowed to call `not_inline_hd`. In its generated code, nvcc may omit `not_inline_hd`'s call to `host_only` entirely, or it may try to generate code for `host_only` on the device. What you get seems to depend on whether or not the compiler chooses to inline `host_only`.

Member functions, including constructors, may be overloaded using H and D attributes. However, destructors cannot be overloaded.

Using a Different Class on Host/Device

Occasionally you may want to have a class with different host/device versions.

If all of the class's members are the same on the host and device, you can just provide overloads for the class's member functions.

However, if you want your class to have different members on host/device, you won't be able to provide working H and D overloads in both classes. In this case, clang is likely to be unhappy with you.

```
#ifdef __CUDA_ARCH__
struct S {
  __device__ void foo() { /* use device_only */ }
  int device_only;
};
#else
struct S {
  __host__ void foo() { /* use host_only */ }
  double host_only;
};

__device__ void test() {
  S s;
  // clang generates an error here, because during host compilation, we
  // have ifdef'ed away the __device__ overload of S::foo().  The __device__
  // overload must be present *even during host compilation*.
  S.foo();
}
#endif
```

We posit that you don't really want to have classes with different members on H and D. For example, if you were to pass one of these as a parameter to a kernel, it would have a different layout on H and D, so would not work properly.

To make code like this compatible with clang, we recommend you separate it out into two classes. If you need to write code that works on both host and device, consider writing an overloaded wrapper function that returns different types on host and device.

```
struct HostS { ... };
struct DeviceS { ... };

__host__ HostS MakeStruct() { return HostS(); }
__device__ DeviceS MakeStruct() { return DeviceS(); }

// Now host and device code can call MakeStruct().
```

Unfortunately, this idiom isn't compatible with nvcc, because it doesn't allow you to overload based on the H/D attributes. Here's an idiom that works with both clang and nvcc:

```
struct HostS { ... };
struct DeviceS { ... };

#ifdef __NVCC__
  #ifndef __CUDA_ARCH__
    __host__ HostS MakeStruct() { return HostS(); }
  #else
    __device__ DeviceS MakeStruct() { return DeviceS(); }
  #endif
#else
  __host__ HostS MakeStruct() { return HostS(); }
  __device__ DeviceS MakeStruct() { return DeviceS(); }
```

```
#endif

// Now host and device code can call MakeStruct().
```

Hopefully you don't have to do this sort of thing often.

Optimizations

Modern CPUs and GPUs are architecturally quite different, so code that's fast on a CPU isn't necessarily fast on a GPU. We've made a number of changes to LLVM to make it generate good GPU code. Among these changes are:

- Straight-line scalar optimizations – These reduce redundancy within straight-line code.
- Aggressive speculative execution – This is mainly for promoting straight-line scalar optimizations, which are most effective on code along dominator paths.
- Memory space inference – In PTX, we can operate on pointers that are in a particular "address space" (global, shared, constant, or local), or we can operate on pointers in the "generic" address space, which can point to anything. Operations in a non-generic address space are faster, but pointers in CUDA are not explicitly annotated with their address space, so it's up to LLVM to infer it where possible.
- Bypassing 64-bit divides – This was an existing optimization that we enabled for the PTX backend.

  64-bit integer divides are much slower than 32-bit ones on NVIDIA GPUs. Many of the 64-bit divides in our benchmarks have a divisor and dividend which fit in 32-bits at runtime. This optimization provides a fast path for this common case.
- Aggressive loop unrolling and function inlining – Loop unrolling and function inlining need to be more aggressive for GPUs than for CPUs because control flow transfer in GPU is more expensive. More aggressive unrolling and inlining also promote other optimizations, such as constant propagation and SROA, which sometimes speed up code by over 10x.

  (Programmers can force unrolling and inline using clang's loop unrolling pragmas and `__attribute__((always_inline))`.)

Publication

The team at Google published a paper in CGO 2016 detailing the optimizations they'd made to clang/LLVM. Note that "gpucc" is no longer a meaningful name: The relevant tools are now just vanilla clang/LLVM.

gpucc: An Open-Source GPGPU Compiler
Jingyue Wu, Artem Belevich, Eli Bendersky, Mark Heffernan, Chris Leary, Jacques Pienaar, Bjarke Roune, Rob Springer, Xuetian Weng, Robert Hundt
Proceedings of the 2016 International Symposium on Code Generation and Optimization (CGO 2016)

Slides from the CGO talk

Tutorial given at CGO