OpenMP Offloading Features in LLVM 15

July 29th 2022
Joseph Huber
LLVM Release & Talk Overview

- LLVM 15 has been forked and contains several new features for offloading, incl.
  - A new compiler driver
  - Multi-Architecture binaries
  - Link Time Optimization
  - Static Library Support
  - OpenMP and CUDA / HIP interoperability
  - Extra flags improving offloading performance
Building LLVM with OpenMP offload

• Single command often suffices to configure:
  cmake llvm-project/llvm -DLLVM_ENABLE_PROJECTS='clang;lld' -DLLVM_ENABLE_RUNTIMES='openmp'
  make install -j

• Useful options include:
  CMAKE_BUILD_TYPE={Release,Asserts,…}
  LLVM_ENABLE_ASSERTIONS={ON,OFF}
  LLVM_CCACHE_BUILD={ON,OFF}
  -G Ninja

• Various resources available online! Start here:
  https://llvm.org/docs/GettingStarted.html
  https://openmp.llvm.org/SupportAndFAQ.html
Compilation Phases w/ the New Driver

Compilation

Device Codegen

C/C++ and OpenMP

Clang

Dev IR

LLVM

Dev ASM

Dev Obj

Vendor tools

LLVM

Host IR

LLVM

Host Obj

Linking

Linker Wrapper

Host Obj

Exe

libomptarget (host, plugins)
Embedding Device Code

C/C++ and OpenMP

Host IR

Device Object

Binary Packager

Active Toolchains

Section Table

Host Sections

@.llvm.embedded.object = private constant [N x i8] c"...", section ".llvm.offloading"

Section Contents

<Triple and Arch>

<Bitcode or Object>

Contains kernels and globals to register
Linker Wrapper

Extracts and links each target in the fat binary.

Embeds the linked device image and create code to register it.

Device Linking

- Dev BC
- LTO
- Dev Obj
- Vendor Linker
- Dev Exe

Wrapper

- Host Object
- Exe

Linked objects flow through the diagram:

- Linker Input
- Extract
- Dev Obj
- Wrapper
- Exe

Wrapped Linker job
Multi-architecture Binaries

- LLVM now supports compiling for many architectures
  - Allows the same binary to run on several machines
- Without --fopenmp-targets we will try to infer the triples

```bash
$ clang app.c -fopenmp -fopenmp-targets=nvptx64,admgcn -c \
  -Xopenmp-target=nvptx64 --offload-arch=sm_80 \
  -Xopenmp-target=amdgcn --offload-arch=gfx90a
$ clang app.c -fopenmp --offload-arch=sm_80 --offload-arch=gfx90a -c
$ llvm-readelf -S app.o
Section Headers:
[Nr] Name              Type            Address    Off  Size   ES Flg Lk Inf Al
[11] .llvm.offloading LLVM_OFFLOADING 000000000002058 002058 0024c0 00 E 0 0 8
[12] omp_offloading_entries PROGBITS 000000000005048 004048 000020 00 A 0 0 8
```
Multi-architecture Binaries

- Can inspect the embedded device code with binary utils

```bash
$ clang app.c -fopenmp --offload-arch=sm_80 --offload-arch=gfx90a -o app
$ llvm-objdump --offloading ./app
OFFLOADING IMAGE [0]:
  kind elf
  arch gfx90a
  triple amdgcn-amd-amdhsa
  producer openmp

OFFLOADING IMAGE [1]:
  kind elf
  arch sm_80
  triple nvptx64-nvidia-cuda
  producer openmp
```
Link Time Optimization (LTO)

• Compilers normally optimize a single translation unit (TU) at a time
  – LTO allows the compiler to optimize the whole program
• LLVM now supports LTO for the device
• Currently needs to be specified for both

$ clang app.c -fopenmp -fopenmp-targets=nvptx64 -foffload-lto -O3 -c
$ clang app.o -fopenmp -fopenmp-targets=nvptx64 -foffload-lto -O3
LTO Performance Improvement (A100 Nvidia GPU)
Static Library Support

- LLVM now completely supports static libraries
  - Any method of creating static libraries should work now
- The linker only imports used symbols from static libraries
  - Somewhat inherit this behaviour for multi-architecture binaries

```
$ clang foo.c -fopenmp --offload-arch=sm_70 --offload-arch=sm_80 --offload-arch=gfx908 -c
$ llvm-ar rcs libfoo.a foo.o
$ clang app.c -fopenmp --offload-arch=sm_70 -lfoo -o app
$ llvm-objdump --offloading
OFFLOADING IMAGE [0]:
kind elf
arch sm_70
triple nvptx64-nvidia-cuda
producer openmp
```
Static Library Support

- Can use this to create generic libraries
  - LTO can create zero overhead libraries

```c
#pragma omp begin declare target device_type(nohost)

#pragma omp begin declare variant match(...) void foo() {}
#pragma omp end declare variant

#pragma omp end declare target
```

```
$ clang device.c -fopenmp --offload-arch=sm_52,sm_70,sm_80,gfx908,gfx90a,gfx90c -O3 \
    -offload-lto -fvisibility=hidden -fopenmp-cuda-mode
$ llvm-ar rcs libdevice.a device.o
$ clang app.c -fopenmp --offload-arch=sm_80 -offload-lto -ldevice
```
CUDA / HIP Interoperability

- The new driver can compile both CUDA and HIP
  - Requires explicitly using the new Driver
- LLVM now supports CUDA compilation in RDC-mode
  - Previously required external build systems

```
$ clang++ cuda.cu util.cu -fgpu-rdc --offload-arch=sm_70 --offload-new-driver -c
$ clang++ cuda.o util.o --offload-link -lcudart -o app
$ ./a.out
```
Linker Wrapper

* Device Images will be distinct
CUDA / HIP Interoperability

- OpenMP interoperability with CUDA / HIP objects
  - Caveat: Global state is not yet shared
  - Would require having all state registered by OpenMP Or CUDA

```c
void openmp() { printf("Hello from OpenMP\n"); }
#pragma omp declare target device_type(nohost) to(openmp)
__device__ cuda() { printf("Hello from CUDA\n"); }
```

```
$ clang++ cuda.cu -fgpu-rdc --offload-arch=sm_70 --offload-new-driver -c
$ clang++ openmp.cpp -fopenmp --offload-arch=sm_70 -c
$ clang++ cuda.o openmp.o -fopenmp -fopenmp-targets=nvptx64 -lcudart
./a.out
Hello from OpenMP
Hello from CUDA
```
Device Only Compilation

• Device only compilation output the device result
  – Caveat: Can only output a single architecture currently
• Mainly useful for inspecting output

$ clang app.c -fopenmp --offload-arch=sm_70 -S -emit-llvm --offload-device-only -o -
  < LLVM IR >
Mandatory Offloading

• OpenMP offloading supports host-fallback by default
• This requires emitting each device function on the host
• Can be disabled using a command line flag
  – Makes interoperability with CUDA easier.

$ clang app.c -fopenmp --offload-arch=sm_70 -fopenmp-offload-mandatory
Passing Arguments to the Device Linker

- The linker wrapper links many devices in a single invocation
- Extra arguments can be forwarded to the device linker if needed

```
$ clang app.c -fopenmp --offload-arch=sm_70 -Xoffload-linker -g
$ clang app.c -fopenmp --offload-arch=sm_70 -Xoffload-linker-nvptx64-nvidia-cuda -g
```
No Thread State Assertions

- The OpenMP GPU runtime supports standard OpenMP
  - Some features are difficult to optimize out and costly
- We provide a flag to manually disable thread state
  - Used for features like nested parallelism and tasking
  - Should hopefully not be required once we have more advanced optimizations
- Should improve application performance

$ clang app.c -fopenmp --offload-arch=gfx90a -fopenmp-assume-no-thread-state
Questions?