[EMDC OS]

Intel DPC++의 팻-바이너리 구조 분석과 동적 조립/재조립을 위한 도구 사용법

버전	내용
v1.0	동적 조립/재조립을 위한 도구 사용법 및 초안 작성
v0.2	동적 조립/재조립을 위한 팻-바이너리 구조 작성
v0.1	개요 및 Intel DPC++ compilation flow 와 팻-바이너리 구조 작성

성균관대학교 컴파일러 및 시스템 연구실

1. 개요

본 문서는 엣지 마이크로 데이터 센터에서 이종 컴퓨팅 가속기의 핵심자원으로서 추상화를 위해 Intel DPC++[1]의 팻-바이너리 구조 분석, 동적 조립/재조립을 위한 구조 정의, 도구 사용법을 설명한다. 이종 컴퓨팅 가속기 자원을 추상화하기 위해서 Khronos group에서 표준화한 SYCL[2]을 Intel에서 구현한 DPC++를 대상으로 컴파일 과정, 팻-바이너리 구조를 분석한다.

2. 배경 지식

1) SYCL

SYCL은 크로스-플랫폼 추상화 계층으로 ISO C++ 표준으로 작성된 호스트와 커널 코드가 하나의 파일에 작성된 애플리케이션 코드를 여러 이종 가속기에서 실행할 수 있도록 KHRONOS GROUP에서 정의한 표준이다[2].

2) Intel DPC++

Intel Data Parallel C++(DPC++)은 data parallel 프로그래밍을 위한 고급 언어이며, Intel의 SYCL implementation을 포함한다. Intel CPU, GPU, FPGA 뿐만 아니라 NVIDIA GPU(CUDA)도 지원하며 디바이스 코드가 여러 가속기에서 실행될 수 있도록 multiple target backend 옵션을 제공한다[1].

3) Intel DPC++ compilation flow

Intel DPC++ 컴파일러는 LLVM 기반으로 C++ 호스트 컴파일러, SYCL 디바이스 프론트-엔드 컴파일러, 가속기 타겟별 LLVM 컴파일러 등으로 구성된다. 그림 1 은 DPC++ 컴파일 과정으로 크게 호스트 코드 컴파일과 디바이스 코드 컴파일 과정으로 나뉜다.

호스트 코드는 호스트 C++ 컴파일러가 호스트 오브젝트 파일로 컴파일한다. 람다 또는 커널 형태로 작성된 디바이스 코드는 SYCL 디바이스 프론트-엔드 컴파일러를 거쳐 LLVM IR로 변환된다. 이후 컴파일 타겟에 따라서 Intel CPU, GPU, FPGA 타겟은 llvm-spirv가 SPIRV로 변환하고 NVIDIA CUDA pass는 llvm-spirv를 거치지않고 LLVM IR이 타겟 컴파일러로 전달된다. SPIRV/LLVM IR은 이종 가속기 타겟 컴파일러에 의해 타겟 바이너리로 컴파일되고 offload-wrapper가 wrapper 오브젝트로 변환한다. 호스트 오브젝트와 wrapper 오브젝트를 링킹하여 팻-바이너리 형태로 결과물을 도출한다.

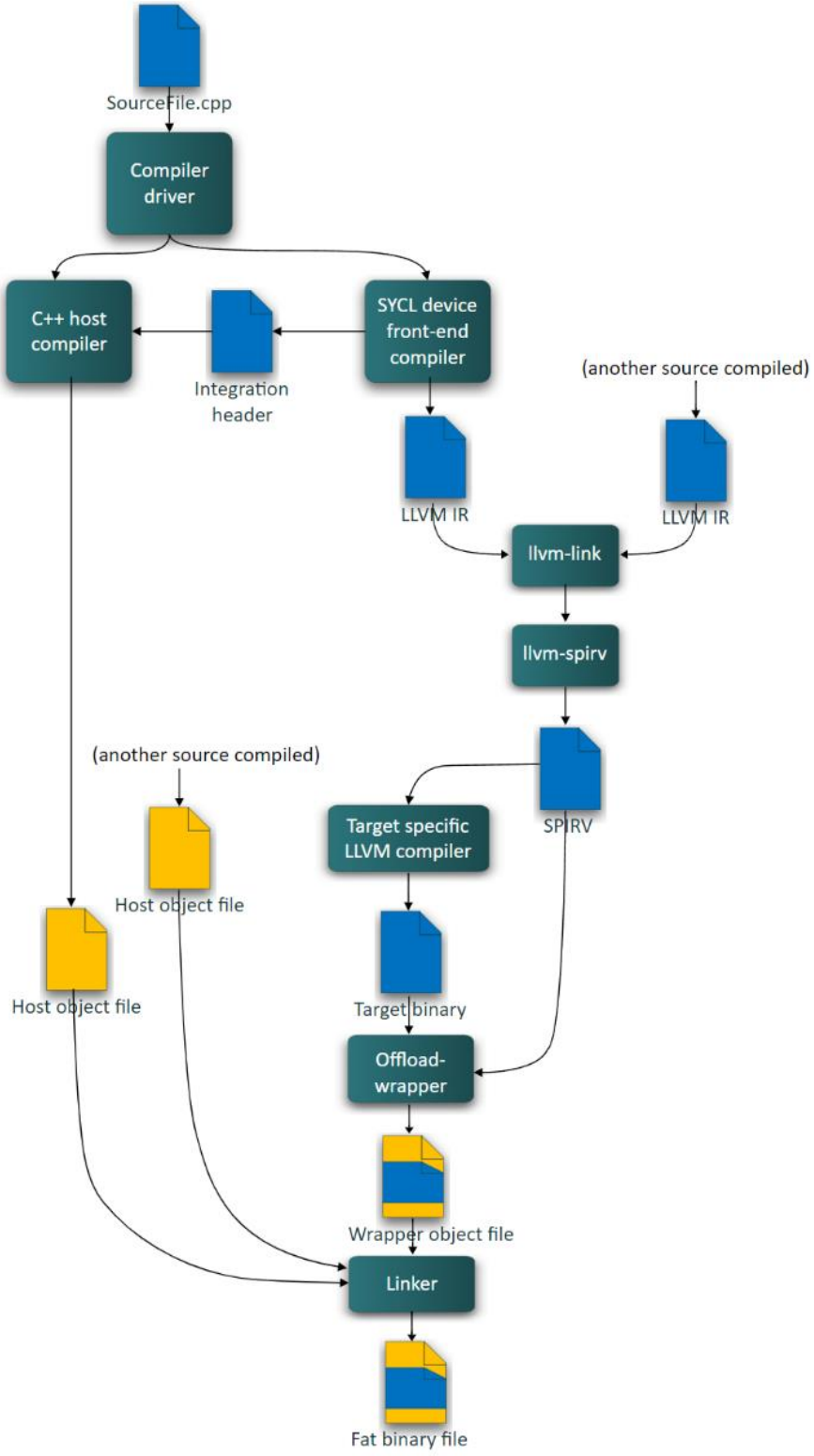


그림 1 Intel DPC++ compilation flow[3]

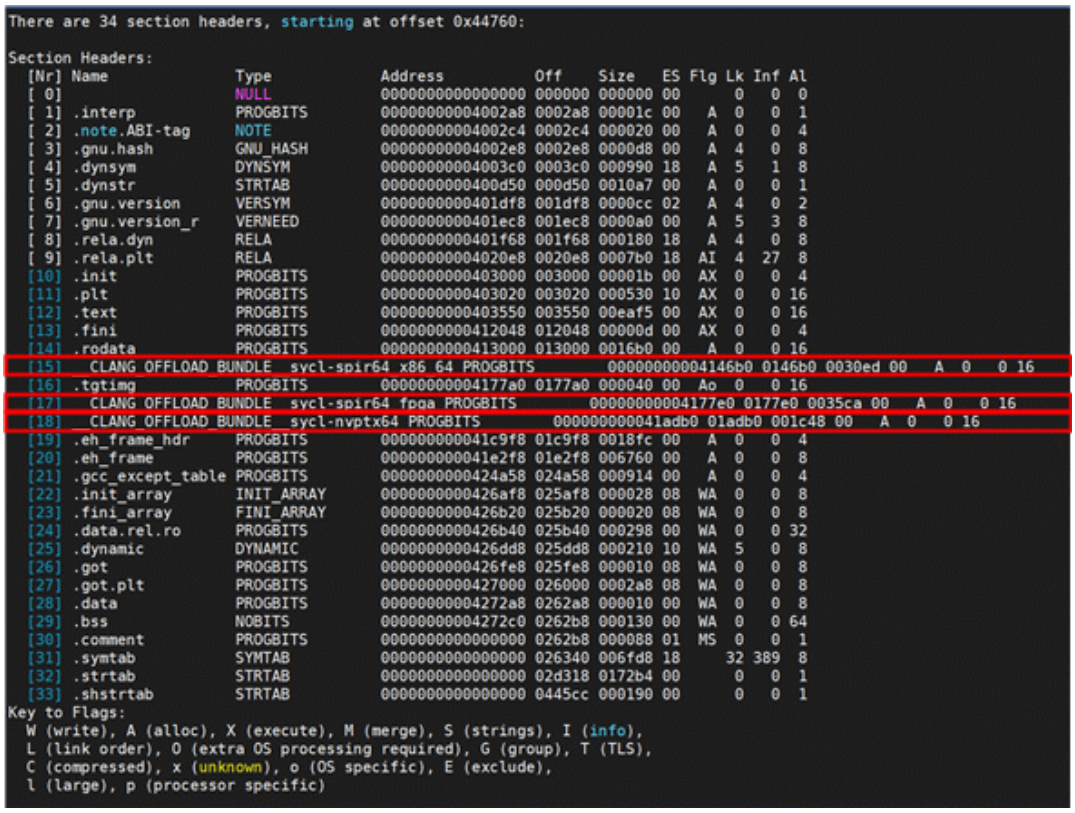
3. 팻-바이너리 구조 분석

1) ELF

Executable and Linkable Format(ELF)은 ELF header, Program(segment) header table, section entries, Section header table로 구성된다.

2) Intel DPC++ 팻-바이너리 구조

Intel DPC++는 디바이스 코드를 여러 타겟으로 여러 번 컴파일한 후 타겟 바이너리를 wrapper 오브젝트 형태로 변환하여 각각 다른 section으로 관리한다. 그림 2는 ComputeCPP SDK의 scan 애플리케이션을 Intel CPU, FPGA emulator, NVIDIA GPU(CUDA)를 타겟으로 컴파일한 팻 바이너리의 section headers이다. 컴파일된 타겟의 바이너리들은 wrapper 오브젝트로 각각 다른 __CLANG_OFFLOAD_BUNDLE__ section으로 나뉘어 구성되는 것을 확인할 수 있다.



```
There are 34 section headers, starting at offset 0x44760:

Section Headers:
  [Nr] Name              Type            Address          Off    Size   ES Flg Lk Inf Al
  [ 0]                   NULL            0000000000000000 000000 000000 00      0   0  0
  [ 1] .interp           PROGBITS        00000000004002a8 0002a8 00001c 00   A  0   0  1
  [ 2] .note.ABI-tag     NOTE            00000000004002c4 0002c4 000020 00   A  0   0  4
  [ 3] .gnu.hash         GNU_HASH        00000000004002e8 0002e8 0000d8 00   A  4   0  8
  [ 4] .dynsym           DYNSYM          00000000004003c0 0003c0 000990 18   A  5   1  8
  [ 5] .dynstr           STRTAB          0000000000400d50 000d50 0010a7 00   A  0   0  1
  [ 6] .gnu.version      VERSYM          0000000000401df8 001df8 0000cc 02   A  4   0  2
  [ 7] .gnu.version_r    VERNEED         0000000000401ec8 001ec8 0000a0 00   A  5   3  8
  [ 8] .rela.dyn         RELA            0000000000401f68 001f68 000180 18   A  4   0  8
  [ 9] .rela.plt         RELA            00000000004020e8 0020e8 0007b0 18  AI  4  27  8
  [10] .init             PROGBITS        0000000000403000 003000 00001b 00  AX  0   0  4
  [11] .plt              PROGBITS        0000000000403020 003020 000530 10  AX  0   0 16
  [12] .text             PROGBITS        0000000000403550 003550 00eaf5 00  AX  0   0 16
  [13] .fini             PROGBITS        0000000000412048 012048 00000d 00  AX  0   0  4
  [14] .rodata           PROGBITS        0000000000413000 013000 0016b0 00   A  0   0 16
  [15]   CLANG OFFLOAD BUNDLE  sycl-spir64_x86_64 PROGBITS        00000000004146b0 0146b0 0030ed 00   A  0   0 16
  [16] .tgtimg           PROGBITS        00000000004177a0 0177a0 000040 00  Ao  0   0 16
  [17]   CLANG OFFLOAD BUNDLE  sycl-spir64_fpga PROGBITS        00000000004177e0 0177e0 0035ca 00   A  0   0 16
  [18]   CLANG OFFLOAD BUNDLE  sycl-nvptx64 PROGBITS        000000000041adb0 01adb0 001c48 00   A  0   0 16
  [19] .eh_frame_hdr     PROGBITS        000000000041c9f8 01c9f8 0018fc 00   A  0   0  4
  [20] .eh_frame         PROGBITS        000000000041e2f8 01e2f8 006760 00   A  0   0  8
  [21] .gcc_except_table PROGBITS        0000000000424a58 024a58 000914 00   A  0   0  4
  [22] .init_array       INIT_ARRAY      0000000000426af8 025af8 000028 08  WA  0   0  8
  [23] .fini_array       FINI_ARRAY      0000000000426b20 025b20 000020 08  WA  0   0  8
  [24] .data.rel.ro      PROGBITS        0000000000426b40 025b40 000298 00  WA  0   0 32
  [25] .dynamic          DYNAMIC         0000000000426dd8 025dd8 000210 10  WA  5   0  8
  [26] .got              PROGBITS        0000000000426fe8 025fe8 000010 08  WA  0   0  8
  [27] .got.plt          PROGBITS        0000000000427000 026000 0002a8 08  WA  0   0  8
  [28] .data             PROGBITS        00000000004272a8 0262a8 000010 00  WA  0   0  8
  [29] .bss              NOBITS          00000000004272c0 0262b8 000130 00  WA  0   0 64
  [30] .comment          PROGBITS        0000000000000000 0262b8 000088 01  MS  0   0  1
  [31] .symtab           SYMTAB          0000000000000000 026340 006fd8 18     32 389  8
  [32] .strtab           STRTAB          0000000000000000 02d318 0172b4 00      0   0  1
  [33] .shstrtab         STRTAB          0000000000000000 0445cc 000190 00      0   0  1
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings), I (info),
  L (link order), O (extra OS processing required), G (group), T (TLS),
  C (compressed), x (unknown), o (OS specific), E (exclude),
  l (large), p (processor specific)
```

그림 2 Intel DPC++ 팻 바이너리 section headers

4. 동적 조립/재조립을 위한 구조 및 도구 사용법

1) 동적 조립/재조립을 위한 팻-바이너리 구조

엣지 마이크로 데이터 센터에서 작업 스케줄러가 태스크에 가속기 자원을 할당한 이후에 실행을 위해 필요한 타겟 바이너리만 동적으로 조립하고, 다른 가속기로 재할당되는 경우에 동적으로 재조립하고자 한다. 이를 위해서는 앞선 팻-바이너리 구조에서 특정 offload section을 추가/삭제가 가능해야 한다.

하지만, 현재 ELF 구조에서는 새로운 offload section을 추가하기 위해서는 mapping이 필요하여 매우 복잡하고 불안정하고, 제거하기 위해서는 .plt의 내용이 .got 또는 .got.plt의 주소를 가르키고 있기 때문에 중간에 위치한 offload section을 실제 바이너리 파일에서 제거할 수 없다.

이런 문제들을 해결하기 위해서 데이터 센터에서 가용한 모든 가속기에 대한 offload section들을 가장 끝 entry에 위치하도록 만들고, 빈 파일로 업데이트하여 삭제와 유사한 효과를 내는 구조를 제안한다.

2) 링커 도구 사용법

링커(ld)를 이용하여 offload section의 시작 주소를 명시하여 offload section들이 가장 끝에 위치하도록 조정한다. 링커 옵션 중 --section-start SECTION=ADDRESS를 이용하며 이 링커 옵션은 -Xlinker <arg> 형태로 컴파일러에 전달할 수 있고 아래 예시처럼 사용할 수 있다.

예시) clang++ … -Xlinker --section-start -Xlinker __CLANG_OFFLOAD_BUNDLE__sycl-spir64_fpga=0x600000 …

3) 바이너리 도구 사용법

GNU BIN Utility(objcopy, llvm-objcopy, strip 등)를 이용하여 특정 offload section을 추가, 제거한다. 추가, 제거 기능은 실제로는 update 옵션을 이용한다. objcopy 도구를 예로 들면, --update-section <name>=<file> 형태로 사용할

수 있고, 아래 예시처럼 CPU 타겟 section을 제거할 수 있다.

예시) objcopy --update-section __CLANG_OFFLOAD_BUNDLE__sycl-spir64_x86_64=/dev/null

4) 동적 조립 순서

① offload section들이 가장 마지막에 위치하도록 링커 옵션을 이용하여 컴파일

② 각 offload section을 백업(objcopy --dump-section 옵션 사용)

③ 불필요한 offload section 제거(update 옵션 사용)

참고문헌

[1] Intel DPC++

[2] SYCL

[3] Intel DPC++ compilation flow