Appendix - File Format

Executable and Linkable Format [1], abbreviated to ELF, is a standard file format typically used on Unix and Unix-like systems. A variant of this format is also used by NVIDIA software in order to package low-level GPU code. An ELF file consists of four components: the ELF header, the section header table which describes each of the ELF file's sections, the program header table which defines the memory segments, and finally the various sections.

In this section we describe the file format of CUDA applications, which is vital when trying to modify existing code. We performed all of our experimentation on Linux (Ubuntu and openSUSE), and so information concerning the CPU executable may not be applicable for operating systems such as Windows. Information concerning the GPU ELF, however, is applicable regardless of the operating system.

GPU ELF

Every CUDA program has one or more executable ELF files embedded inside of it, which contain the GPU code. Notably, this nested ELF can have some differences from typical ELF files, such as a unique version number, which may cause problems for existing programs and libraries when trying to analyze it.

Every ELF's header has an attribute named e_version, which is usually set to 1; other values are typically considered invalid. Though the embedded GPU ELF files created by some versions of nvcc match this, with others this value is set to the compiler version. For example, with nvcc version 6.5, e_version is set to 65, and with nvcc version 8.0, e_version is set to 80. The ELF header is otherwise as expected, though the e_machine attribute - which indicates the architecture - is set to 190 to indicate CUDA.

The compiler will usually generate mangled names for each kernel function. For example, a kernel with the name "foo" and one integer parameter will likely have the mangled name "_Z3fooi". The ELF will use this name instead of the original, unmangled version.

For each kernel it describes, the ELF will contain a section called ".text.func", where func is the mangled name of the kernel, containing the function's binary code. The highest eight bits of the INFO attribute for this section control the number of registers which will be allocated per-thread, and the lowest eight bits hold this section's index in the symbol table.

For a kernel function which uses shared memory, the ELF will also contain a section named ".nv.shared.func" for each kernel "func" - the size of this section is the number of bytes of shared memory which the GPU will allocate per thread-block for the associated kernel function. Similarly, it can contain sections named ".nv.constantX.func" for different values of X, allocating (and possibly initializing) constant memory for the kernel functions. Each .nv... section's INFO value is equal to the index of the associated .text... section's index in the section header table.

Several of the sections have an entry in the symbol table of the GPU ELF. Additionally, the symbol table can have entries for subroutines within the kernel functions, with an st_info value of 34.These have a st_shndx value equal to the section number of the associated kernel function's .text... section, and an st_value value equal to their offset inside the kernel function's code.

.nv.info The ELF also contains sections named ".nv.info.func" for each kernel "func", and also a ".nv.info" section, containing various pieces of metadata. For example, the amount of local memory allocated per-thread is controlled by the MIN_STACK_SIZE, FRAME_SIZE, and MAX_STACK_SIZE attributes inside of .nv.info.

The .nv.info sections contain one or more attributes. An attribute starts with a byte indicating the attribute format, and then a byte with the attribute ID. If the attribute format is 1 (NVAL), then there is no associated value. If the attribute format is 2 (BVAL), then the following byte contains the attribute's value. If the attribute format is 3 (HVAL), then the following two bytes contain the attribute's value. Finally, if the attribute format is 4 (SVAL), then the following two bytes contain some value, n, and then the next n bytes contain the attribute's values.

For example, for FRAME_SIZE, the attribute type will be 4 (SVAL), the next two bytes after the attribute ID will contain the size 8, the next four bytes contain the associated kernel function's index in the symbol table, and the remaining four bytes contain the actual frame size value.

In Table 1, we list all attribute IDs we are aware of that can appear in the .nv.info sections, with their human-readable name and their number in the actual binary. Some attributes are only compatible with more recent versions of the CUDA SDK.

CPU Executable

With older versions of nvcc, the GPU ELF described above was stored inside the .rodata section of the CPU ELF. But starting with version 5.0 released in 2012, the compiler creates an executable containing dedicated sections for GPU code.

Most important is the .nv_fatbin section. It is split into an arbitrary number of distinct regions, each of which contains one or more GPU ELF files, PTX code files, and/or cubin files. Each region begins with a 16 byte header: the first 8 bytes are the .nv_fatbin magic number, and the remaining eight bytes contain the size of the rest of the region. The rest of the region alternates between detailed headers and the embedded file (ELF, PTX, or cubin) which the detailed header describes.

In the detailed header, the first 4-byte word contains the embedded file's type and ptxas flags; the lower two bytes

Table 1. Known .nv.info attributes.

Attribute (EIATTR)	ID
ERROR	0x00
PAD	0x01
IMAGE_SLOT	0x02
JUMPTABLE_RELOCS	0x03
CTAIDZ_USED	0x04
MAX_THREADS	0x05
IMAGE_OFFSET	0x06
IMAGE_SIZE	0x07
TEXTURE_NORMALIZED	0x08
SAMPLER_INIT	0x09
PARAM_CBANK	0x0a
SMEM_PARAM_OFFSETS	0x0b
CBANK_PARAM_OFFSETS	0x0c
SYNC_STACK	0x0d
TEXID_SAMPID_MAP	0x0e
EXTERNS	0x0f
REQNTID	0x10
FRAME_SIZE	0x11
MIN_STACK_SIZE	0x12
SAMPLER_FORCE_UNNORMALIZED	0x13
BINDLESS_IMAGE_OFFSETS	0x14
BINDLESS_TEXTURE_BANK	0x15
BINDLESS_SURFACE_BANK	0x16
KPARAM_INFO	0x17
SMEM_PARAM_SIZE	0x18
CBANK_PARAM_SIZE	0x19
QUERY_NUMATTRIB	0x1a
MAXREG_COUNT	0x1b
EXIT_INSTR_OFFSETS	0x1c
S2RCTAID_INSTR_OFFSETS	0x1d
CRS_STACK_SIZE	0x1e
NEED_CNP_WRAPPER	0x1f
NEED_CNP_PATCH	0x20
EXPLICIT_CACHING	0x21
ISTYPEP_USED	0x22
MAX_STACK_SIZE	0x23
SUQ_USED	0x24
LD_CACHEMOD_INSTR_OFFSETS	0x25
LOAD_CACHE_REQUEST	0x26
ATOM_SYS_INSTR_OFFSETS	0x27
COOP_GROUP_INSTR_OFFSETS	0x28
COOP_GROUP_MASK_REGIDS	0x29
SW1850030_WAR	0x2a
WMMA_USED	0x2b

have a value of 2 for GPU ELF files. The second word is the offset of the embedded file, relative to the start of this detailed header. The dword comprising the third and fourth words holds the size of the embedded file. The seventh word is the code version, which is dependent on the compiler. The eighth word contains the target architecture - a value of 20 for compute capability 2.0, a value of 35 for compute capability 3.5, etcetera. The rest of the detailed header contains less important metadata, such as the operating system or the source code's filename.

Another section of the CPU ELF that is unique to CUDA programs is called .nvFatBinSegment. It contains metadata about the .nv_fatbin section, such as the starting addresses of its regions. Its size is a multiple of six words (24 bytes), where the third word in each group of six is an address inside of the .nv_fatbin section. If we modify the .nv_fatbin, then these addresses need to be changed to match it. (TODO figure out exactly how the rest of this section works - possibly the first word is a magic number, and the second word is the number of six-word groups remaining including the current one?)

Side Effects of Modification

If we modify the size of the GPU kernel in a way that requires us to change the size of the .nv_fatbin section, then adjusting the parts described above are insufficient to keep the executable working. There are other changes that need to be made to prevent the program from simply crashing.

Increasing the size of the GPU code shifts the addresses of various parts of the executable. As such, several changes need to be made to the CPU executable. The offsets for several sections need to be fixed in the program header section. We also scan the CPU assembly code for any addresses which point anywhere after the changed part of the .nv_fatbin section, and increment them appropriately in the binary. Similarly, we fix such addresses inside several sections including any symbol tables (indicated by a type of SHT_SYMTAB or SHT_DYNSYM), relocation tables (with type SHT_RELA), dynamic tables (with type SHT_DYNAMIC), and global offset tables (with name ".got").

While we find that these fixes seem to work in practice, it is difficult to guarantee that they will be successful. For example, cases may arise where other data is mistakenly treated as an address and incremented, creating errors. As such, whenever possible, it is best to prepare the executable in such a way that modifications will not require extra space for additional code.

References

[1] Committee, T., et al. Tool interface standard (tis) executable and linking format (elf) specification version 1.2. *TIS Committee* (1995).