OpenMP offload infrastructure in LLVM

Samuel Antao (IBM)

Carlo Bertolli (IBM)

Andrey Bokhanko (Intel)

Alexandre Eichenberger (IBM)

Hal Finkel (Argonne National Laboratory)

Sergey Ostanevich (Intel)

Eric Stotzer (Texas Instruments)

Guansong Zhang (AMD)

1. OpenMP 4.0 offloading overview

The OpenMP 4.0 specification introduces offloading directives that can be used to take advantage of accelerators or *devices*, in OpenMP terminology (see the OpenMP 4.0 specification at www.openmp.org).

- **Device**: an implementation-defined logical execution unit. The execution model is host-centric such that a *host device* offloads code and data to *target devices.*
- **Target** regions are structured code blocks that execute on a target device. This is conditional on the run-time availability of a device, the ability of the compiler to generate device code, and other factors.
- **Target declarations** are used to specify mapping of global variables to a device and create device specific versions of functions that can be called from target regions.
- **Device data environments** contain the variables that are currently present on the target device.
- A **mapped variable** is a variable in a (host) data environment with a corresponding variable in a device data environment.

The **target** directive creates both a device data environment and a target region. It may have associated clauses to specify further details, like the exact device to use if more than one is present in the system (**device** clause) or whether the data should be moved to/from the device or only allocated in the device memory (**map** clause). The **declare target** directive declares that enclosed global variables and functions should have corresponding device versions.

Example 1 illustrates how offloading can be expressed using the available set of directives.

```
// foo() will be implemented for the host and device
#pragma omp declare target
int foo(int[1000]);
#pragma omp end declare target


// All declarations outside a declare target region will NOT be implemented
// for the device
...
int device_count = omp_get_num_devices();
int device_no;
int *red = malloc(device_count * sizeof(int));
int c[1000];

// Several host threads are going to be spawned to execute the structured
// block associated with the parallel region (this is unrelated with the
// offloading support )
#pragma omp parallel for
for (i = 0; i < 1000; i++) {

  device_no = i % device_count;

  // The target directive specifies that the execution of associated
  // structured block (target region) should be transferred to the device.
  //
  // The device clause states that device whose ID is device_no should be
  // used.
  //
  // The first map clause specifies that an instance of c has to be allocated
  // in the device and updated with the host content of c prior the execution
  // of the target region (to:).
  //
  // The second map clause specifies that an instance of red[i] has to be
  // allocated in the device and updated with the host content prior the
  // execution of the target region and that the host instance should be
  // updated with the content of the device instance after the target region
  // execution is complete.
  //
  // If for some reason the device is not available, a host version of the
  // target region is executed instead.
  #pragma omp target device(device_no) map(to:c) map(red[i])
  {
    // This code is going to be executed on the device(s)
    red[i] += foo(c);
  }

  // The execution of the target region is blocking, therefore the dedicated
  // host thread will wait for the device to complete execution.

}

for (i = 0; i< device_count; i++)
  total_red = red[i];
```

Example 1

2. The goal of the design

- The offload infrastructure should support multiple target device types at runtime and be extensible in the future with minimal or no changes.
- The infrastructure should determine the availability of target devices at runtime and make a decision to offload depending on the availability and load of a target device.

3. Model of use

The proposed offload mechanism implements the OpenMP 4.0 “target data”, “target” and “declare target” constructs. The compiler generates calls to the runtime library whenever a “target data” or “target” directive is encountered. The “declare target” construct will result in the generation of appropriate *target code* for the target device.

The target code is stored inside the host binaries as additional ELF sections with an appropriate naming convention. The target code is either target assembly in binary form (ELF, PE, etc.) or a higher-level intermediate representation (IR) such as LLVM IR or any other type of IR. If the target code is stored as IR, an implementation can support on-the-fly compilation into target assembly. Note that ELF terminology is being used throughout the document only for illustrative purposes – the implementation should accommodate any other object format supported by LLVM.

A target-independent offload runtime library named `libomptarget.so` supports multiple target device types. The `libomptarget.so` library utilizes device-specific target run-time libraries (RTLs). At the start of host code execution `libomptarget.so` will do the following:

1. Search for target RTLs according to a naming convention.
2. Verify target RTL interface compliance.
3. Add target RTLs into a list of available target device types.

The `libomptarget.so` provides the host with an API to map variables and initiate execution of target regions on a target device. After the `libomptarget.so` has verified that suitable target code is present and that a target RTL is ready to execute a target region, the target RTL is invoked via API routines (described below) to execute the region.

This scheme provides flexibility for generating code for multiple heterogeneous device types. For example, the target region code in Example 1 can be executed in a Phi™ coprocessor if it is present on user’s system or on a GPU system if it is present (see Figure 1). The design of the offloading interface does not limit the number and type of devices associated to a host processor or the ability to use these at the same time. If both devices are present, different iterations of the loop can be executed on both devices at the same time.

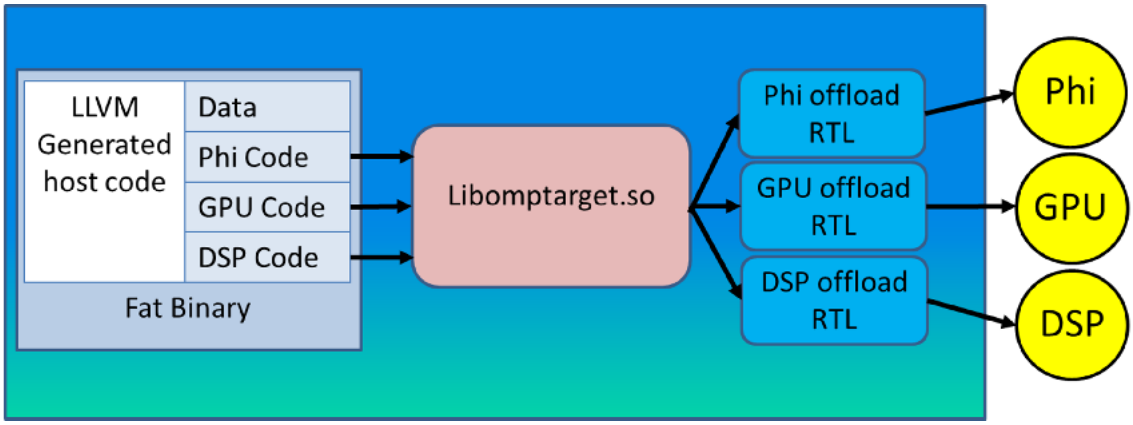

Figure 1

4. Generation of fat binaries

To make code generation consistent and straightforward the following scheme is proposed:

1. For each source file provided, the driver spawns the execution of preprocessor, compiler and assembler for the host and each available target device type. This results in the generation of an object file for each target device type. The toolchain of a given target may be modified so that it uses the same definitions (header files) as the host toolchain if that suits the system constraints.
2. Target linkers combine dedicated target objects into target shared libraries – one for each target device type. The commands passed to the target frontend by the compiler driver always assume the creation of a shared library even if the commands passed to the driver by the user specify otherwise. The driver performs the translation of the host frontend commands to target frontend commands to assure that a target shared library is generated.
3. The host linker combines host object files into an executable/shared library and incorporates each target shared libraries *as is* (no actual linking is done between host and target objects) into its own separate section within the host binary. The format of a binary section for offloading to a specific device is target-dependent and will be thereafter handled by the target RTL at runtime.
4. A new driver command-line option `–omptargets=T1,…,Tn` where `Ti` are valid target triples that specify which target device types the user wants to support in the execution of OpenMP target regions. An example, of the invocation of the compiler would be:

   ```
   clang –fopenmp –target powerpc64-ibm-linux-gnu –omptargets=nvptx64-
   nvidia-cuda,x86-pc-linux-gnu foo.c bar.c –o foobar.bin
   ```

   for a hypothetical system where the host is a PowerPC processor and the available target device types are an NVIDIA GPU and x86 processor.

5. For each source file passed to the driver a unique object file is created for each target device type. The naming convention for a target object file is to append a suffix `tgt-<some-target-triple>` to the host object file name. At link time the driver forwards the target object files to the corresponding target toolchain. This mechanism underlies the compiler support for separate compilation. For example:

   ```
   clang –fopenmp –target powerpc64-ibm-linux-gnu –omptargets=nvptx64-
   nvidia-cuda,x86-pc-linux-gnu foo.c -c
   ```

produces the files `foo.o`, `foo.o.tgt-nvptx64-nvidia-cuda` and `foo.o.tgt-x86-pc-linux-gnu`. Then the command

```
clang –fopenmp –target powerpc64-ibm-linux-gnu –omptargets=nvptx64-nvidia-cuda,x86-pc-linux-gnu foo.o –o foo.bin
```

instructs the compiler to locate and forward the files `foo.o.tgt-nvptx64-nvidia-cuda` and `foo.o.tgt-x86-pc-linux-gnu.o` *to the nvptx64-nvidia-cuda* and *x86-pc-linux-gnu* toolchains, respectively, so they can be properly linked before being embedded in the host binary.

The resulting host executable/shared library will depend on the offload runtime library – `libomptarget.so`. This library will handle the initialization of target RTLs and translate the offload interface from compiler-generated code to the target RTL during program execution.

5. The libomptarget.so interface

The offload library implements the OpenMP 4.0 user-level runtime library routines:

- `void omp_set_default_device(int device_num)`
- `int omp_get_default_device(void)`
- `int omp_get_num_devices(void)`
- `int omp_is_initial_device(void)`

The offload library implements the following compiler-level runtime library routines:

- `void __tgt_register_lib(__tgt_bin_desc* desc)`

  Register the `libomptarget.so` library and initialize target state (i.e. global variables and target entry points) for the current host shared library/executable and the corresponding target execution images that have those entry points implemented. This does not trigger any execution in any target as any real work with the target device can be postponed until the first target region is encountered during execution. This function is expected to appear only once per host shared library/executable in the .init section and is called before any constructors or static initializers are called for the host. The name of the caller of __tgt_register_lib follows the same pattern of the C++ initializers in Clang and is set to `_GLOBAL__A_000000_OPENMPTGT`.

- `void __tgt_target_data_begin(int32_t device_id, int32_t num_args, void** args_base, void** args, int64_t *args_size, int32_t *args_maptype)`

  Initiate a device data environment. It maps variables from the host data environment to the device data environment by recording the mapping between the associated variables' references used in the host and target into the `libomptarget.so` internal structures. The associated variables in the target device data environment are initialized according to the map-type. In the event a given associated variable has already been mapped in other enclosing device data environment, no action is taken for that variable.

- `void __tgt_target_data_end(int32_t device_id, int32_t num_args, void** args_base, void** args, int64_t *args_size, int32_t *args_maptype)`

  Close a device data environment. It removes mapped variables from the current device data environment, releases target memory and destroy the mappings created by the

__tgt_target_data_begin() call that initiated the current device data environment. It assigns host variables the value of the corresponding device data environment variable according to the map-type.

- void __tgt_target_data_update(int32_t device_id, int32_t num_args, void** args_base, void** args, int64_t *args_size, int32_t *args_maptype)

  Make the value of a set of variables consistent between the host device and a target device. If a variable's map type is 'from', use the value of the variable on the target device. If a variable's map type is 'to', use the value of the variable on the host device.

- int32_t __tgt_target(int32_t device_id, void *host_addr, int32_t num_args, void** args_base, void** args, int64_t *args_size, int32_t *args_maptype)

  Perform the same actions as __tgt_target_data_begin in case arg_num is non-zero and launch the execution of the target region on the target device; if arg_num is non-zero after the region execution is done it also performs the same action as __tgt_data_data_end above. If the offloading fails, an error code is returned, which notifies the caller to transfer execution to the appropriate host region. The return code can be used as an error code which will give the compiler and run-time the freedom to implement optimized behaviors.

- int32_t __tgt_target_teams(int32_t device_id, void *host_addr, int32_t num_args, void** args_base, void** args, int64_t *args_size, int32_t *args_maptype, int32_t num_teams, int32_t thread_limit)

  This is an extension of __tgt_target where the caller is able to specify the (maximum) number of teams and threads in each team that libomptarget.so should start. It reflects the common nesting of the pragma target with the teams one, in OpenMP 4.0.

All the __tgt_target… calls presented above perform an initial check to understand if the target specified by device_id was already initialized, and if not, triggers that initialization. The information registered by __tgt_register_lib is used to accomplish that.

5.1. Arguments for the libomptarget.so calls

The following arguments are used:

__tgt_bin_desc* desc points to a constant data struct statically defined by the compiler:

```
struct __tgt_bin_desc{
  uint32_t              NumDevices;
  __tgt_device_image   *DeviceImages;
  __tgt_offload_entry  *EntriesBegin;
  __tgt_offload_entry  *EntriesEnd;
};
```

NumDevices is the number of device types whose execution image was generated by the compiler in order to implement some of the offloading entry points. The device types are specified by the user during the invocation of the compiler by passing the flag –omptargets=T1,…,Tn where Ti is the triple of a target the user wants to support. NumDevices equals n in this case. DeviceImages is a pointer to an array of NumDevices elements, whose element type is

```
struct __tgt_device_image{
  void   *ImageStart;
  void   *ImageEnd;
  __tgt_offload_entry  *EntriesBegin;
  __tgt_offload_entry  *EntriesEnd;
};
```

where `ImageStart` and `ImageEnd` contain the addresses where a target image associated to the current host executable/shared library for a given device type starts and ends, respectively. `ImageEnd` is non-inclusive, i.e. it points to the byte immediately after the target image ends. `EntriesBegin` and `EntriesEnd` point to the first and last element of an array that contains the information of each global variable and target entry point that require a map between host and target. These pointers are present in both `__tgt_bin_desc` and `__tgt_device_image` so that the target dependent runtime (see section 5) can use this information as well to more easily retrieve the entries from the target image (e.g. to retrieve the symbol names of the entries – see below). Each element of the array pointed by `EntriesBegin` has type

```
struct __tgt_offload_entry{
  void      *addr;
  char      *name;
  int64_t    size;
};
```

where `addr` is the address of that global variable or entry point in the host, `name` is the name of the symbol that refers to that global variable or entry point, and `size` is the size in bytes of the global variable or zero if it is an entry point. If the address field is set to `NULL`, the corresponding entry in the table is not supported by the target device associated to this table. `libomptarget.so` has to be able to map the host entries to the corresponding device entries. There are different strategies that can be used for mapping the entries. The simplest one is to use the names of the entries for the mnapping by performing a name-based search using the `name` field of the `__tgt_offload_entry`. The field `name` is useful for targets whose runtime requires access to the symbol names in order to locate the correspondent address in the target image. The array that starts at `EntriesBegin` is built by the compiler in conjunction with the linker, which forwards sequences of entries of different compilation units to the same binary section. In case target and host toolchains can provide strict ordering for both target and host tables – then the mapping can be done by the sequence number of the entry. The frontend ensures that the global variables/entries follow the same order they appear in the source file. Entries associated with static initializers and global destructors are appended to the end of the entries array (i.e. after the global variables themselves and the entries associated with target regions) in the same order they are required in the program. E.g. in the sample program in Example 2, the order of the entries will be: global variable `a`, target region 1, global variable `b`, target region 2, caller of the constructor of `a` and `b` (`a` and `b` have the same priority), caller of the destructors of `a` and `b`. The callers of the constructors and destructors are always launched with a single thread and team. If the toolchain of a given target does not preserve the order, that target runtime may consult the host entries and obtain the same order of the symbols based on the name.

`int32_t device_id` is an integer that uniquely identifies a given target. In the first call of `__tgt_register_lib` the `libomptarget.so` library detects the available target dependent RTLs in the system and uses them to query the number of devices of each type that are ready to be used. If `A` devices of type `T1` and `B` devices of type `T2` are found, where `T1` and `T2` are the types specified

by the user with `-omptargets=T1,T2`, `device_id [0,A[` will map to devices of type `T1` and `device_id [A,A+B[` will map to devices of type `T2`. If `device_id` is greater or equal than `A+B`, the call where it is used will fail and no action will be taken by `libomptarget.so`. On top of the positive values used for `device_id`, the compiler also employs three reserved values:

- `device_id = -1`: informs the runtime that the user has not specified any device ID, and therefore the default must be used, which may be specified through an environment variables as specified in the OpenMP 4.0 specification.
- `device_id = -2`: informs the runtime that the target action must be performed on all available devices and can be delayed until the first time the device action is invoked with a device ID greater or equal to -1. This is mainly used to call C++ global initializers, which only need to be called if the device is eventually used for executing at least one target region.
- `device_id = -3`: informs the runtime that the target action must be performed on all available devices that were ever used in the current library. This is mainly used to call C++ destructors, which are only required if that device was used before.

`int32_t num_args` is the number of data pointers that require a mapping.

`void** args` is a pointer to an array with `num_args` arguments whose elements point to the first byte of the array section that needs to be mapped.

`int64_t* args_size` is a pointer to an array with `num_args` arguments whose elements contain the size in bytes of the array section to be mapped.

`void** args_base` is a pointer to an array with `num_args` arguments whose elements point to the base address of the declaration the mapping refers to. `args_base` differs from `args` if an array section does not start at zero. `libomptarget.so` needs to know the base addresses in order to relate mapped data with target region arguments that are dereferenced in the target region body.

```
#pragma omp declare target
class C{
    C() {// ctor of C}
    ~C() {//dtor of C}
};
C a;
#pragma omp end declare target

foo(){
    #pragma target
    { //target region 1 }
}

#pragma omp declare target
C b;
#pragma omp end declare target

bar(){
    #pragma target
    { //target region 2 }
}
```

Example 2

`int32_t *args_maptype` is a pointer to an array with `num_args` arguments whose elements contain the required map attributes as specified in the enum:

```
enum tgt_map_type {
  OMP_TGT_MAPTYPE_ALLOC   = 0x0000,
  OMP_TGT_MAPTYPE_TO      = 0x0001,
  OMP_TGT_MAPTYPE_FROM    = 0x0002,
  OMP_TGT_MAPTYPE_ALWAYS  = 0x0004,
  OMP_TGT_MAPTYPE_RELEASE = 0x0008,
  OMP_TGT_MAPTYPE_DELETE  = 0x0018,
  OMP_TGT_MAPTYPE_POINTER = 0x0020
}
```

The attributes OMP_TGT_MAPTYPE_ALLOC to OMP_TGT_MAPTYPE_DELETE follow the map types in the OpenMP 4.1 specification. OMP_TGT_MAPTYPE_POINTER informs the runtime the map is based on a pointer instead of a constant array. OMP_TGT_MAPTYPE_POINTER will cause the runtime to allocate the memory of the pointer and to initialize it with the address of the memory allocated on the device. The following aims at clarifying the use of this flag. Table 1 shows the content of the arrays passed to `__tgt_target_data_begin()` and `__tgt_target` as result of the omp target data and omp target pragmas. The interface is used in two different ways depending on the variable that is being mapped:

- Scalars and Arrays: `args_base` will contain references to the variables that are being mapped. For scalars, `args` will point to the variables that are being mapped. For arrays, `args` will point to the beginning of the section that the user specified (the user may require only part of the array to be used in the device). `args_size` will contain the size of the variable or section to be mapped and `args_maptype` will contain flags to control the data movement between host and device.
- Pointers: the first time a pointer is mapped, the region it is pointing to is also mapped, and the mapped address should be used to initialize the mapped pointer. When a pointer is being mapped, `args_base` will contain the reference to a pointer but `args` will contain the start address of the pointee's section that has to be mapped.

A and pA are also passed to `__tgt_target` because they are captured in the body of the target region and are therefore arguments to the target region. However, the `libomptarget.so` implementation will detect they were mapped before and will not take any action to map these variables again.

The arguments that are used to invoke the target kernel (see `void *target_vars_ptr` in section 5) consist of the mapped base address of all elements. In the example above the arguments would be `[&dB[0], &di, &dpB, &A[0], &dpA]`, where dX is the map of X in the device.

args_base	args	args_size	args_maptype
#pragma omp target data - __tgt_target_data_begin()			
&A[0]	&A[S]	M*sizeof(int)	OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM
&pA	&pA[S]	M*sizeof(int)	OMP_TGT_MAPTYPE_POINTER | OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM
#pragma omp target - __tgt_target()			
&B[0]	&B[0]	M*sizeof(int)	OMP_TGT_MAPTYPE_FROM
&i	&i	sizeof(int)	OMP_TGT_MAPTYPE_FROM
&pB	&pB[S]	M*sizeof(int)	OMP_TGT_MAPTYPE_POINTER | OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM
&A[0]	&A[0]	M*sizeof(int)	OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM |
&pA	&pA	M*sizeof(void*)	OMP_TGT_MAPTYPE_TO | OMP_TGT_MAPTYPE_FROM

Table 1

```
// N, M and S are constants

foo(){
    int A[N], B[M];
    int *pA, *pB;
    int i;

    pA = (int*)malloc(N*sizeof(int));
    pB = (int*)malloc(M*sizeof(int));
    #pragma omp target data map(A[S:M], pA[S:M])
    {
        #pragma omp target map(from:B,i) map(to:pB[S:M])
        {
            for (i=S; i<M; ++i){
                ++A[i];
                --pA[i];
                B[i-S] = pB[i] - A[i]*pA[i];
            }
        }
    }
}
```

Example 3

6. Target RTL interface

As it can be derived from the previous chapter, a target RTL must provide the following capabilities:

- `int32_t __tgt_rtl_device_type(int32_t device_id)` – return an integer that identifies the device type. The identifier must match the target binary descriptor.
- `int32_t __tgt_rtl_number_of_devices()` – return the number of available devices of the type supported by the target RTL.
- `int32_t __tgt_init_device(int32_t device_id)` – initialize the specified device. In case of success return 0; otherwise return an error code.
- `tgt_target_table* __tgt_rtl_load_binary(int32_t device_id, __tgt_device_image *image)` – pass an executable image section described by `image` to the specified device and prepare an address table of target entities. In case of error, return `NULL`. Otherwise, return a pointer to the built address table. Individual entries in the table may also be `NULL`, when the corresponding offload region is not supported on the target device (see previous chapter).
- `void* __tgt_rtl_data_alloc(int32_t device_id, int64_t size)` – allocate data on the particular target device, of the specified `size`. Return address of the allocated data on the target that will be stored in the `libomptarget.so` host to target data mapping structures. These mapping structures are used to generate a target variables address table to pass to `__tgt_rtl_run_region()`. The `__tgt_rtl_data_alloc()` returns `NULL` in case an error occurred on the target device.
- `int32_t __tgt_rtl_data_submit(int32_t device_id, void *target_ptr, void *host_ptr, int64_t size)` – pass the data content to the target device using the target address. In case of success, return zero. Otherwise, return an error code.
- `int32_t __tgt_rtl_data_retrieve(int32_t device_id, void *host_ptr, void *target_ptr, int64_t size)` – retrieve the data content from the target device using its address. In case of success, return zero. Otherwise, return an error code.
- `int32_t __tgt_rtl_data_delete(int32_t device_id, void *target_ptr)` – de-allocate the data referenced by `target_ptr` on the device. In case of success, return zero. Otherwise, return an error code.
- `int32_t __tgt_rtl_run_target_region(int32_t device_id, void *target_entry_ptr, void **target_vars_ptr, int32_t arg_num)` – transfer control to the offloaded entry on the target device; `target_vars_ptr` is a table to store the target addresses of all variables used in the target entry code. Entries in `target_vars_ptr` match the order of the variables passed in the `arg_host_ptr` argument passed to the target region. In case of success, return zero. Otherwise, return an error code.
- `int32_t __tgt_rtl_run_target_team_region(int32_t device_id, void *target_entry_ptr, void **target_vars_ptr, int32_t arg_num, int32_t num_teams, int32_t thread_limit)` – transfer control to the offloaded entry on the target device; `target_vars_ptr` is a table to store the target addresses of all variables used in the target team entry code. Entries in `target_vars_ptr` match the order of the variables passed in the `arg_host_ptr` argument passed to the target team region. In case of success, return zero. Otherwise, return an error code.

For each platform there can be a dedicated set of error numbers defined. `libomptarget.so` assumes that zero is returned in case of success, independently of the target device. If a given system supports shared memory, the target RTL implementation of the `__tgt_rtl_data_*()` can optionally not take any action, as host and target device can operate on the same data.