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. The goal of the design

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

2. Model of use

The proposed offload mechanism is expected to be used in accordance with the OpenMP 4.0 “target data”, “target” and “declare target” constructs. The compiler should generate calls to the runtime library whenever “target data” or a “target” is met, while the “declare target” construct will only cause generation of appropriate code for the target platform.

The target code generated by LLVM can be stored inside the host binaries as additional ELF sections with an appropriate naming convention. This code can be either target assembly in binary form, or it can be a LLVM IR so that the target runtime implementation can support on-the-fly compilation into target assembly. The latter can have some IP implications since LLVM IR is known to be easily reversed to high level language.

To enable multiple target platforms we propose to use a target-independent runtime library named `libtarget.so` which in turn relies on particular platform dynamic libraries (target plugins) presence. At the start of host code execution `libtarget.so` will search for target RTLs according to a naming convention. It will also verify plugin interface compliance and add this plugin into the list of possible targets. During the execution of host code there can appear a call to the `libtarget.so` interface to initiate execution of the target region on a target device. At this point `libtarget.so` verifies whether suitable target code is present and whether a target plugin is ready to perform the execution (depending on presence of HW and its load) – if so, calls to the plugin’s interface with the appropriate data are prepared.

The scheme above provides flexibility so that a software vendor can generate code for multiple target platforms at the same time and leverage performance of all targets present amongst his users, even on the same system.

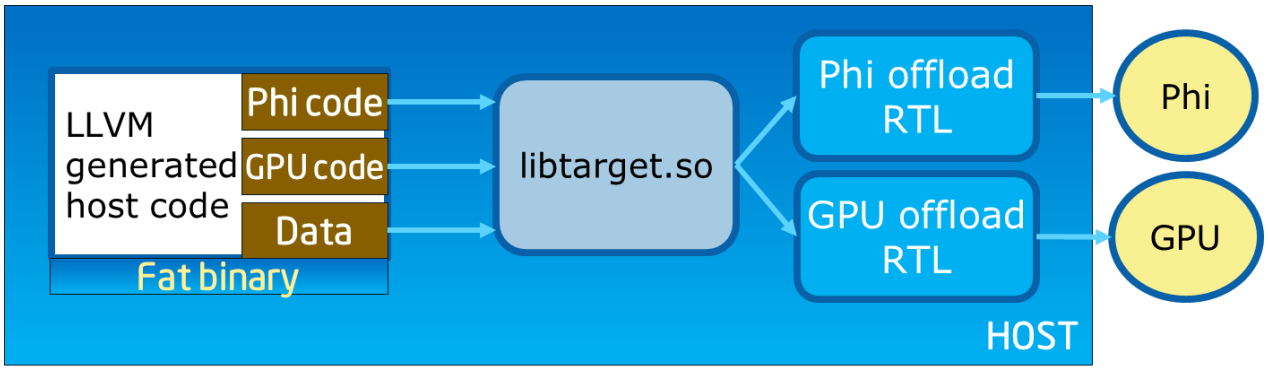

Figure 1

The following code:

```
#pragma omp declare target
int foo(int[1000]);
#pragma omp end declare target
...
int device_count = omp_get_num_devices();
int device_no;
int *red = malloc(device_count * sizeof(int));
#pragma omp parallel
for (i = 0; i < 1000; i++) {
  device_no = i % device_count;
  #pragma omp target device(device_no) map(to:c) map(red[i])
  {
    red[i] += foo(c);
  }
}

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

can be executed both on Intel® Xeon Phi™ coprocessor if it is present on user's system or it can be executed on a GPU system if it is present (see Figure 1). If both devices are present, different iterations of the loop can be executed on both devices simultaneously.

3. Generation of fat binaries

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

1. The driver called on a source code should spawn a number of front-end executions for each available target. We assume no restriction on a compiler used for target – it can be a user-defined 3rd party compiler. This should generate a set of object files for each target
2. Target linkers combine dedicated target objects into target shared libraries – one for each target
3. The host linker combines host object files into an executable/shared library and incorporates shared libraries for each target into a separate section within host binary. This process and format is target-dependent and will be thereafter handled by the target RTL at runtime

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

4. The libtarget.so interface

First of all the library should provide the OpenMP 4.0 user-level interfaces related to device part according to spec:

- `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 libtarget.so provides the following interface to the compiler generated code:

- `void __tgt_register_lib(tgt_bin_desc*)` – adds the target shared libraries to the target execution images descriptors within the libtarget.so. This is done to postpone the real work with target device until first target region is encountered during execution. This function is expected to appear only once per host shared library/executable in the .init section. The libtarget.so will have a list of target libraries that need initialization on target device
- `__tgt_target_data_begin(int device_id, int arg_num, void** arg_host_ptr, int* arg_size, tgt_map_type* arg_types)` – tests whether target image initialization is done, creates host to the target data mapping, store it in the libtarget.so internal structure (an entry in a stack of data maps) and passes the data to the device
- `__tgt_target_data_end(int device_id, int arg_num, void** arg_host_ptr, int* arg_size, tgt_map_type* arg_types)` – passes data from the target, release target memory and destroys the host-target mapping created by the corresponding `__tgt_target_data_begin()` call
- `__tgt_target_data_update(int device_id, int arg_num, void** arg_host_ptr, int* arg_size, tgt_map_type* arg_types)` – passes data to/from the target
- `__tgt_target(int device_id, void* host_address, int arg_num, void** arg_host_ptr, int* arg_size, tgt_map_type* arg_types)` – performs the same actions as `__tgt_target_data_begin` in case `arg_num` is non-zero and initiates run of offloaded region on target platform; if `arg_num` is non-zero after the region execution is done it also performs the same action as `data_update` and `data_end` above
- `__tgt_target_teams(int device_id, void* host_address, int arg_num, void** arg_host_ptr, int* arg_size, tgt_map_type* arg_types, int num_teams, int 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 libtarget.so should start. It reflects the common nesting of the pragma target with the teams one, in OpenMP 4.0.

  The following types are used:

```
struct __tgt_bin_desc{
  unsigned              NumDevices;
  tgt_device_image   *DeviceImages;
  tgt_offload_entry  *EntriesBegin
```

```
  tgt_offload_entry  *EntriesEnd;
};

struct tgt_offload_entry {
  void* tgt_offload_entry_address;
  int tgt_offload_entry_size;
}

struct tgt_device_image{
  void   *ImageStart;
  void   *ImageEnd;
};

enum tgt_map_type {
  tgt_map_alloc,
  tgt_map_to,
  tgt_map_from,
  tgt_map_tofrom
}
```

`libtarget.so` should initialize internal structures for tracking the offload targets availability, number of devices for each target, loaded shared libraries and corresponding address tables of the target regions. At startup host executable or shared library should call `__tgt_register_lib()` to initiate a search through the executable image for the appropriate section containing the targets binary. The `tgt_bin_desc` struct contains the information required for each target stored into an array of `tgt_device_image` structs. For each entry of this array, the `ImageStart` and `ImageEnd` fields identify the appropriate section and they are target dependent: each RTL is free to interpret the content of the characterized section as needed. For instance, for a specific device these two fields identify the start and end addresses of a section containing the implementation of the outlined target functions on the related device. On a different device, the section may contain additional information to optimize function retrieval and data access, along with the device-specific target function implementations.

When initialization of the host executable and all its libraries is done the libtarget.so will have a list of target binaries that should be transferred to the target device using the target RTL. Target RTL should provide an address table for the target regions present in each binary so that `libtarget.so` can map each region to the corresponding device's host address in the range from `EntriesBegin` to `EntriesEnd` in the `tgt_bin_desc` struct. In order for this to work, it is required that the symbols in the target image that represent entry points are retrieved in the same order the host entries are stored in that range. The target RTL may return NULL entries in this table for specific entry points that are not supported by the target.

`__tgt_target_data_begin()` – `libtarget.so` should check if data structure descriptor is present in `libtarget.so` for the relevant target and data. If no such structure descriptor is present it should call the target RTL interface to create appropriate data structure on target device and create a descriptor that reflects host-target address mapping. After the structure is present the transfer of the data or initialization is done according to the arguments passed.

`__tgt_target_data_end()` – `libtarget.so` should perform a call to the target RTL interface to pass data from the corresponding target data structure to the host. After that a call to the target RTL should free memory on target device and libtarget.so should remove the descriptor of the data structure from internal structure.

__tgt_target_data_update() – libtarget.so performs transfer to/from target device using RTL interface without update to data structure descriptor.

At the time of the call to __tgt_target() libtarget.so determines that

1. the target device is supported, which means that offload RTL is present and available for use
2. the requested address of the target region is present in the address table

After successful test of these conditions libtarget.so performs target region address translation and calls to the corresponding offload RTL interface to initiate the execution of translated address on the device. If the __tgt_target() or __tgt_target_team() call fails to pass the test it will return a integer different from zero. It is task of 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.

5. Target RTL interface

As can be derived from the previous chapter the target RTL should provide at least the following capabilities:

- int __tgt_rtl_device_type(int device_id) – returns a descriptor of the device type; it should match with target binaries descriptor
- int __tgt_rtl_number_of_devices() – returns number of available devices of the type supported by the target RTL
- void __tgt_init_device(int device_id) – initializes the device
- tgt_target_table* __tgt_rtl_load_binary(int device_id, void *target_image, int target_image_size) – to pass the binary to the particular target device and prepare an address table of target entries
- void* __tgt_rtl_data_alloc(int device_id, int size) – to allocate data on the particular target device according to the arguments and provide its target address to store in the libtarget.so host to target data mapping structures – they will be used further to generate a target variables address table to pass to __tgt_rtl_run_region()
- __tgt_rtl_data_submit(int device_id, void *target_ptr, void *host_ptr, int size) – to pass the data content to the target device using the target address
- __tgt_rtl_data_retrieve(int device_id, void *host_ptr, void *target_ptr, int size) – to retrieve the data content from the target device using its address
- __tgt_rtl_data_delete(int device_id, void *target_ptr) – to de-allocate the data at the target device according to its address
- __tgt_rtl_run_target_region(int device_id, void *target_entry_ptr, void *target_vars_ptr) – to transfer the control to the offloaded entry on target device; the 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
- __tgt_rtl_run_target_team_region(int device_id, void *target_entry_ptr, void *target_vars_ptr) – to transfer the control to the offloaded entry on target device; the 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