NAME

llvmopenmp - LLVM/OpenMP 15.0.7

NOTE:

This document is a work in progress and most of the expected content is not yet available. While you can expect changes, we always welcome feedback and additions. Please contact, e.g., through **openmp-dev@lists.llvm.org**.

OpenMP impacts various parts of the LLVM project, from the frontends (*Clang* and Flang), through middle-end *optimizations*, up to the multitude of available *OpenMP runtimes*.

A high-level overview of OpenMP in LLVM can be found *here*.

OPENMP IN LLVM --- DESIGN OVERVIEW

Resources

- OpenMP Booth @ SC19: "OpenMP clang and flang Development" *https://youtu.be/6yOa-hRi63M*

LLVM/OpenMP Runtimes

There are four distinct types of LLVM/OpenMP runtimes: the host runtime *LLVM/OpenMP Host Runtime (libomp)*, the target offloading runtime *LLVM/OpenMP Target Host Runtime (libomptarget)*, the target offloading plugin *LLVM/OpenMP Target Host Runtime Plugins (libomptarget.rtl.XXXX)*, and finally the target device runtime *LLVM/OpenMP Target Device Runtime (libomptarget-ARCH-SUBARCH.bc)*.

For general information on debugging OpenMP target offloading applications, see *LIBOMPTARGET_INFO* and *Debugging*

LLVM/OpenMP Host Runtime (libomp)

An *early (2015) design document* for the LLVM/OpenMP host runtime, aka. *libomp.so*, is available as a *pdf*.

Environment Variables

OMP_CANCELLATION

Enables cancellation of the innermost enclosing region of the type specified. If set to **true**, the effects of the cancel construct and of cancellation points are enabled and cancellation is activated. If set to **false**, cancellation is disabled and the cancel construct and cancellation points are effectively ignored.

NOTE:

Internal barrier code will work differently depending on whether cancellation is enabled. Barrier

code should repeatedly check the global flag to figure out if cancellation has been triggered. If a thread observes cancellation, it should leave the barrier prematurely with the return value 1 (and may wake up other threads). Otherwise, it should leave the barrier with the return value 0.

Enables (**true**) or disables (**false**) cancellation of the innermost enclosing region of the type specified.

Default: false

OMP_DISPLAY_ENV

Enables (**true**) or disables (**false**) the printing to **stderr** of the OpenMP version number and the values associated with the OpenMP environment variables.

Possible values are: **true**, **false**, or **verbose**.

Default: false

OMP_DEFAULT_DEVICE

Sets the device that will be used in a target region. The OpenMP routine **omp_set_default_device** or a device clause in a parallel pragma can override this variable. If no device with the specified device number exists, the code is executed on the host. If this environment variable is not set, device number 0 is used.

OMP_DYNAMIC

Enables (**true**) or disables (**false**) the dynamic adjustment of the number of threads.
Default: false

OMP_MAX_ACTIVE_LEVELS

The maximum number of levels of parallel nesting for the program.
Default: 1

OMP_NESTED

WARNING:

Deprecated. Please use **OMP_MAX_ACTIVE_LEVELS** to control nested parallelism

Enables (**true**) or disables (**false**) nested parallelism.
Default: false

OMP_NUM_THREADS

Sets the maximum number of threads to use for OpenMP parallel regions if no other value is specified in the application.

The value can be a single integer, in which case it specifies the number of threads for all parallel regions. The value can also be a comma-separated list of integers, in which case each integer specifies the number of threads for a parallel region at that particular nesting level.

The first position in the list represents the outer-most parallel nesting level, the second position represents the next-inner parallel nesting level, and so on. At any level, the integer can be left out of the list. If the first integer in a list is left out, it implies the normal default value for threads is used at the outer-most level. If the integer is left out of any other level, the number of threads for that level is inherited from the previous level.
Default: The number of processors visible to the operating system on which the program is executed.
Syntax: OMP_NUM_THREADS=value[,value]*
Example: OMP_NUM_THREADS=4,3

OMP_PLACES

Specifies an explicit ordered list of places, either as an abstract name describing a set of places or as an explicit list of places described by non-negative numbers. An exclusion operator, **!**, can also be used to exclude the number or place immediately following the operator.

For **explicit lists**, an ordered list of places is specified with each place represented as a set of non-negative numbers. The non-negative numbers represent operating system logical processor numbers and can be thought of as an OS affinity mask.

Individual places can be specified through two methods. Both the **examples** below represent the same place.

- An explicit list of comma-separated non-negatives numbers **Example: {0,2,4,6}**

- An interval with notation **<lower-bound>:<length>[:<stride>]**. **Example: {0:4:2}**. When **<stride>** is omitted, a unit stride is assumed. The interval notation represents this set of numbers:

  <lower-bound>, <lower-bound> + <stride>, ..., <lower-bound> + (<length> - 1) * <stride>

  A place list can also be specified using the same interval notation: **{place}:<length>[:<stride>]**. This represents the list of length **<length>** places determined by the following:

{place}, {place} + <stride>, ..., {place} + (<length>-1)*<stride>
Where given {place} and integer N, {place} + N = {place with every number offset by N}
Example: {0,3,6}:4:1 represents {0,3,6}, {1,4,7}, {2,5,8}, {3,6,9}

Examples of explicit lists: These all represent the same set of places

OMP_PLACES="{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
OMP_PLACES="{0:4},{4:4},{8:4},{12:4}"
OMP_PLACES="{0:4}:4:4"

NOTE:
When specifying a place using a set of numbers, if any number cannot be mapped to a processor on the target platform, then that number is ignored within the place, but the rest of the place is kept intact. If all numbers within a place are invalid, then the entire place is removed from the place list, but the rest of place list is kept intact.

The **abstract names** listed below are understood by the run-time environment:

- **threads:** Each place corresponds to a single hardware thread.
- **cores:** Each place corresponds to a single core (having one or more hardware threads).
- **sockets:** Each place corresponds to a single socket (consisting of one or more cores).
- **numa_domains:** Each place corresponds to a single NUMA domain (consisting of one or more cores).
- **ll_caches:** Each place corresponds to a last-level cache (consisting of one or more cores).

The abstract name may be appended by a positive number in parentheses to denote the length of the place list to be created, that is **abstract_name(num-places)**. If the optional number isn't specified, then the runtime will use all available resources of type **abstract_name**. When requesting fewer places than available on the system, the first available resources as determined by **abstract_name** are used. When requesting more places than available on the system, only the available resources are used.

Examples of abstract names:

OMP_PLACES=threads
OMP_PLACES=threads(4)

OMP_PROC_BIND (Windows, Linux)

Sets the thread affinity policy to be used for parallel regions at the corresponding nested level. Enables (**true**) or disables (**false**) the binding of threads to processor contexts. If enabled, this is the same as specifying **KMP_AFFINITY=scatter**. If disabled, this is the same as specifying **KMP_AFFINITY=none**.

Acceptable values: true, **false**, or a comma separated list, each element of which is one of the following values: **master**, **close**, **spread**, or **primary**.

Default: false

WARNING:

master is deprecated. The semantics of **master** are the same as **primary**.

If set to **false**, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and **proc_bind** clauses on parallel constructs are ignored. Otherwise, the execution environment should not move OpenMP threads between OpenMP places, thread affinity is enabled, and the initial thread is bound to the first place in the OpenMP place list.

If set to **primary**, all threads are bound to the same place as the primary thread.

If set to **close**, threads are bound to successive places, near where the primary thread is bound.

If set to **spread**, the primary thread's partition is subdivided and threads are bound to single place successive sub-partitions.
Related environment variables: KMP_AFFINITY (overrides **OMP_PROC_BIND**).

OMP_SCHEDULE

Sets the run-time schedule type and an optional chunk size.
Default: static, no chunk size specified
Syntax: OMP_SCHEDULE="kind[,chunk_size]"

OMP_STACKSIZE

Sets the number of bytes to allocate for each OpenMP thread to use as the private stack for the thread. Recommended size is 16M.

Use the optional suffixes to specify byte units: **B** (bytes), **K** (Kilobytes), **M** (Megabytes), **G** (Gigabytes), or **T** (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit

is assumed to be **K** (Kilobytes).

This variable does not affect the native operating system threads created by the user program, or the thread executing the sequential part of an OpenMP program.

The **kmp_{set,get}_stacksize_s()** routines set/retrieve the value. The **kmp_set_stacksize_s()** routine must be called from sequential part, before first parallel region is created. Otherwise, calling **kmp_set_stacksize_s()** has no effect.
Default:

- 32-bit architecture: **2M**
- 64-bit architecture: **4M**

Related environment variables: KMP_STACKSIZE (overrides **OMP_STACKSIZE**).
Example: OMP_STACKSIZE=8M

OMP_THREAD_LIMIT

Limits the number of simultaneously-executing threads in an OpenMP program.

If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, the program can abort with an error message. If this limit is reached when an OpenMP parallel region begins, a one-time warning message might be generated indicating that the number of threads in the team was reduced, but the program will continue.

The **omp_get_thread_limit()** routine returns the value of the limit.
Default: No enforced limit
Related environment variable: KMP_ALL_THREADS (overrides **OMP_THREAD_LIMIT**).

OMP_WAIT_POLICY

Decides whether threads spin (active) or yield (passive) while they are waiting.
OMP_WAIT_POLICY=active is an alias for **KMP_LIBRARY=turnaround**, and
OMP_WAIT_POLICY=passive is an alias for **KMP_LIBRARY=throughput**.
Default: passive

NOTE:

Although the default is **passive**, unless the user has explicitly set **OMP_WAIT_POLICY**, there is a

small period of active spinning determined by **KMP_BLOCKTIME**.

KMP_AFFINITY (Windows, Linux)

Enables run-time library to bind threads to physical processing units.

You must set this environment variable before the first parallel region, or certain API calls including **omp_get_max_threads()**, **omp_get_num_procs()** and any affinity API calls.

Syntax: KMP_AFFINITY=[<modifier>,...]<type>[,<permute>][,<offset>]

modifiers are optional strings consisting of a keyword and possibly a specifier

- **respect** (default) and **norespect** - determine whether to respect the original process affinity mask.
- **verbose** and **noverbose** (default) - determine whether to display affinity information.
- **warnings** (default) and **nowarnings** - determine whether to display warnings during affinity detection.
- **reset** and **noreset** (default) - determine whether to reset primary thread's affinity after outermost parallel region(s)
- **granularity=<specifier>** - takes the following specifiers **thread**, **core** (default), **tile**, **socket**, **die**, **group** (Windows only). The granularity describes the lowest topology levels that OpenMP threads are allowed to float within a topology map. For example, if **granularity=core**, then the OpenMP threads will be allowed to move between logical processors within a single core. If **granularity=thread**, then the OpenMP threads will be restricted to a single logical processor.
- **proclist=[<proc_list>]** - The **proc_list** is specified by

Value	Description
<proc_list> :=	<proc_id> | { <id_list> | }
<id_list> :=	<proc_id> | <proc_id>,<id_list>

Where each **proc_id** represents an operating system logical processor ID. For example, **proclist=[3,0,{1,2},{0,3}]** with **OMP_NUM_THREADS=4** would place thread 0 on OS logical processor 3, thread 1 on OS logical processor 0, thread 2 on both OS logical processors 1 & 2, and thread 3 on OS logical processors 0 & 3.

type is the thread affinity policy to choose. Valid choices are **none**, **balanced**, **compact**, **scatter**, **explicit**, **disabled**

- type **none** (default) - Does not bind OpenMP threads to particular thread contexts; however, if the operating system supports affinity, the compiler still uses the OpenMP thread affinity interface to determine machine topology. Specify **KMP_AFFINITY=verbose,none** to list a machine topology map.

- type **compact** - Specifying compact assigns the OpenMP thread <n>+1 to a free thread context as close as possible to the thread context where the <n> OpenMP thread was placed. For example, in a topology map, the nearer a node is to the root, the more significance the node has when sorting the threads.

- type **scatter** - Specifying scatter distributes the threads as evenly as possible across the entire system. **scatter** is the opposite of **compact**; so the leaves of the node are most significant when sorting through the machine topology map.

- type **balanced** - Places threads on separate cores until all cores have at least one thread, similar to the **scatter** type. However, when the runtime must use multiple hardware thread contexts on the same core, the balanced type ensures that the OpenMP thread numbers are close to each other, which scatter does not do. This affinity type is supported on the CPU only for single socket systems.

- type **explicit** - Specifying explicit assigns OpenMP threads to a list of OS proc IDs that have been explicitly specified by using the **proclist** modifier, which is required for this affinity type.

- type **disabled** - Specifying disabled completely disables the thread affinity interfaces. This forces the OpenMP run-time library to behave as if the affinity interface was not supported by the operating system. This includes the low-level API interfaces such as **kmp_set_affinity** and **kmp_get_affinity**, which have no effect and will return a nonzero error code.

For both **compact** and **scatter**, **permute** and **offset** are allowed; however, if you specify only one integer, the runtime interprets the value as a permute specifier. **Both permute and offset default to 0.**

The **permute** specifier controls which levels are most significant when sorting the machine

topology map. A value for **permute** forces the mappings to make the specified number of most significant levels of the sort the least significant, and it inverts the order of significance. The root node of the tree is not considered a separate level for the sort operations.

The **offset** specifier indicates the starting position for thread assignment.
Default: noverbose,warnings,respect,granularity=core,none
Related environment variable: OMP_PROC_BIND (**KMP_AFFINITY** takes precedence)

NOTE:
On Windows with multiple processor groups, the norespect affinity modifier is assumed when the process affinity mask equals a single processor group (which is default on Windows). Otherwise, the respect affinity modifier is used.

NOTE:
On Windows with multiple processor groups, if the granularity is too coarse, it will be set to **granularity=group**. For example, if two processor groups exist across one socket, and **granularity=socket** the runtime will shift the granularity down to group since that is the largest granularity allowed by the OS.

KMP_ALL_THREADS
Limits the number of simultaneously-executing threads in an OpenMP program. If this limit is reached and another native operating system thread encounters OpenMP API calls or constructs, then the program may abort with an error message. If this limit is reached at the time an OpenMP parallel region begins, a one-time warning message may be generated indicating that the number of threads in the team was reduced, but the program will continue execution.
Default: No enforced limit.
Related environment variable: OMP_THREAD_LIMIT (**KMP_ALL_THREADS** takes precedence)

KMP_BLOCKTIME
Sets the time, in milliseconds, that a thread should wait, after completing the execution of a parallel region, before sleeping.

Use the optional character suffixes: **s** (seconds), **m** (minutes), **h** (hours), or **d** (days) to specify the units.

Specify infinite for an unlimited wait time.
Default: 200 milliseconds
Related Environment Variable: KMP_LIBRARY
Example: KMP_BLOCKTIME=1s

KMP_CPUINFO_FILE

Specifies an alternate file name for a file containing the machine topology description. The file must be in the same format as **/proc/cpuinfo**.

Default: None

KMP_DETERMINISTIC_REDUCTION

Enables (**true**) or disables (**false**) the use of a specific ordering of the reduction operations for implementing the reduction clause for an OpenMP parallel region. This has the effect that, for a given number of threads, in a given parallel region, for a given data set and reduction operation, a floating point reduction done for an OpenMP reduction clause has a consistent floating point result from run to run, since round-off errors are identical.
Default: false
Example: KMP_DETERMINISTIC_REDUCTION=true

KMP_DYNAMIC_MODE

Selects the method used to determine the number of threads to use for a parallel region when **OMP_DYNAMIC=true**. Possible values: (**load_balance** | **thread_limit**), where,

- **load_balance**: tries to avoid using more threads than available execution units on the machine;
- **thread_limit**: tries to avoid using more threads than total execution units on the machine.

Default: load_balance (on all supported platforms)

KMP_HOT_TEAMS_MAX_LEVEL

Sets the maximum nested level to which teams of threads will be hot.

NOTE:

A hot team is a team of threads optimized for faster reuse by subsequent parallel regions. In a hot team, threads are kept ready for execution of the next parallel region, in contrast to the cold team, which is freed after each parallel region, with its threads going into a common pool of threads.

For values of 2 and above, nested parallelism should be enabled.

Default: 1

KMP_HOT_TEAMS_MODE

Specifies the run-time behavior when the number of threads in a hot team is reduced. Possible values:

- **0** - Extra threads are freed and put into a common pool of threads.

- **1** - Extra threads are kept in the team in reserve, for faster reuse in subsequent parallel regions.

Default: 0

KMP_HW_SUBSET

Specifies the subset of available hardware resources for the hardware topology hierarchy. The subset is specified in terms of number of units per upper layer unit starting from top layer downwards. E.g. the number of sockets (top layer units), cores per socket, and the threads per core, to use with an OpenMP application, as an alternative to writing complicated explicit affinity settings or a limiting process affinity mask. You can also specify an offset value to set which resources to use. When available, you can specify attributes to select different subsets of resources.

An extended syntax is available when **KMP_TOPOLOGY_METHOD=hwloc**. Depending on what resources are detected, you may be able to specify additional resources, such as NUMA domains and groups of hardware resources that share certain cache levels.

Basic syntax: [num_units|*]ID[@offset][:attribute] [,[num_units|*]ID[@offset][:attribute]...]

Supported unit IDs are not case-insensitive.
S - socket
num_units specifies the requested number of sockets.

D - die
num_units specifies the requested number of dies per socket.

C - core
num_units specifies the requested number of cores per die - if any - otherwise, per socket.

T - thread
num_units specifies the requested number of HW threads per core.

NOTE:

num_units can be left out or explicitly specified as ***** instead of a positive integer meaning use all specified resources at that level. e.g., **1s,*c** means use 1 socket and all the cores on that socket

offset - (Optional) The number of units to skip.

attribute - (Optional) An attribute differentiating resources at a particular level. The attributes available to users are:

- **Core type** - On Intel architectures, this can be **intel_atom** or **intel_core**

- **Core efficiency** - This is specified as **eff***num* where *num* is a number from 0 to the number of core efficiencies detected in the machine topology minus one. E.g., **eff0**. The greater the efficiency number the more performant the core. There may be more core efficiencies than core types and can be viewed by setting **KMP_AFFINITY=verbose**

NOTE:
The hardware cache can be specified as a unit, e.g. L2 for L2 cache, or LL for last level cache.

Extended syntax when KMP_TOPOLOGY_METHOD=hwloc:

Additional IDs can be specified if detected. For example:

N - numa **num_units** specifies the requested number of NUMA nodes per upper layer unit, e.g. per socket.

TI - tile num_units specifies the requested number of tiles to use per upper layer unit, e.g. per NUMA node.

When any numa or tile units are specified in **KMP_HW_SUBSET** and the hwloc topology method is available, the **KMP_TOPOLOGY_METHOD** will be automatically set to hwloc, so there is no need to set it explicitly.

If you don’t specify one or more types of resource, such as socket or thread, all available resources of that type are used.

The run-time library prints a warning, and the setting of **KMP_HW_SUBSET** is ignored if:

- a resource is specified, but detection of that resource is not supported by the chosen topology detection method and/or

- a resource is specified twice. An exception to this condition is if attributes differentiate the resource.

- attributes are used when not detected in the machine topology or conflict with each other.

This variable does not work if **KMP_AFFINITY=disabled**.

Default: If omitted, the default value is to use all the available hardware resources.

Examples:

- **2s,4c,2t**: Use the first 2 sockets (s0 and s1), the first 4 cores on each socket (c0 - c3), and 2 threads per core.
- **2s@2,4c@8,2t**: Skip the first 2 sockets (s0 and s1) and use 2 sockets (s2-s3), skip the first 8 cores (c0-c7) and use 4 cores on each socket (c8-c11), and use 2 threads per core.
- **5C@1,3T**: Use all available sockets, skip the first core and use 5 cores, and use 3 threads per core.
- **1T**: Use all cores on all sockets, 1 thread per core.
- **1s, 1d, 1n, 1c, 1t**: Use 1 socket, 1 die, 1 NUMA node, 1 core, 1 thread - use HW thread as a result.
- **4c:intel_atom,5c:intel_core**: Use all available sockets and use 4 Intel Atom(R) processor cores and 5 Intel(R) Core(TM) processor cores per socket.
- **2c:eff0@1,3c:eff1**: Use all available sockets, skip the first core with efficiency 0 and use the next 2 cores with efficiency 0 and 3 cores with efficiency 1 per socket.
- **1s, 1c, 1t**: Use 1 socket, 1 core, 1 thread. This may result in using single thread on a 3-layer topology architecture, or multiple threads on 4-layer or 5-layer architecture. Result may even be different on the same architecture, depending on **KMP_TOPOLOGY_METHOD** specified, as hwloc can often detect more topology layers than the default method used by the OpenMP run-time library.
- ***c:eff1@3**: Use all available sockets, skip the first three cores of efficiency 1, and then use the rest of the available cores of efficiency 1.

To see the result of the setting, you can specify **verbose** modifier in **KMP_AFFINITY** environment variable. The OpenMP run-time library will output to **stderr** the information about the discovered hardware topology before and after the **KMP_HW_SUBSET** setting was applied.

KMP_INHERIT_FP_CONTROL

Enables (**true**) or disables (**false**) the copying of the floating-point control settings of the primary thread to the floating-point control settings of the OpenMP worker threads at the start of each parallel region.

Default: true

KMP_LIBRARY
Selects the OpenMP run-time library execution mode. The values for this variable are **serial**, **turnaround**, or **throughput**.
Default: throughput
Related environment variable: KMP_BLOCKTIME and **OMP_WAIT_POLICY**

KMP_SETTINGS
Enables (**true**) or disables (**false**) the printing of OpenMP run-time library environment variables during program execution. Two lists of variables are printed: user-defined environment variables settings and effective values of variables used by OpenMP run-time library.

Default: false

KMP_STACKSIZE
Sets the number of bytes to allocate for each OpenMP thread to use as its private stack.

Recommended size is **16M**.

Use the optional suffixes to specify byte units: **B** (bytes), **K** (Kilobytes), **M** (Megabytes), **G** (Gigabytes), or **T** (Terabytes) to specify the units. If you specify a value without a suffix, the byte unit is assumed to be K (Kilobytes).

Related environment variable: KMP_STACKSIZE overrides **GOMP_STACKSIZE**, which overrides **OMP_STACKSIZE**.

Default:

- 32-bit architectures: **2M**
- 64-bit architectures: **4M**

KMP_TOPOLOGY_METHOD
Forces OpenMP to use a particular machine topology modeling method.

Possible values are:

- **all** - Let OpenMP choose which topology method is most appropriate based on the platform and possibly other environment variable settings.

- **cpuid_leaf31** (x86 only) - Decodes the APIC identifiers as specified by leaf 31 of the cpuid instruction. The runtime will produce an error if the machine does not support leaf 31.

- **cpuid_leaf11** (x86 only) - Decodes the APIC identifiers as specified by leaf 11 of the cpuid instruction. The runtime will produce an error if the machine does not support leaf 11.

- **cpuid_leaf4** (x86 only) - Decodes the APIC identifiers as specified in leaf 4 of the cpuid instruction. The runtime will produce an error if the machine does not support leaf 4.

- **cpuinfo** - If **KMP_CPUINFO_FILE** is not specified, forces OpenMP to parse **/proc/cpuinfo** to determine the topology (Linux only). If **KMP_CPUINFO_FILE** is specified as described above, uses it (Windows or Linux).

- **group** - Models the machine as a 2-level map, with level 0 specifying the different processors in a group, and level 1 specifying the different groups (Windows 64-bit only).

  NOTE:
  Support for group is now deprecated and will be removed in a future release. Use all instead.

- **flat** - Models the machine as a flat (linear) list of processors.

- **hwloc** - Models the machine as the Portable Hardware Locality (hwloc) library does. This model is the most detailed and includes, but is not limited to: numa domains, packages, cores, hardware threads, caches, and Windows processor groups. This method is only available if you have configured libomp to use hwloc during CMake configuration.

  Default: all

KMP_VERSION

Enables (**true**) or disables (**false**) the printing of OpenMP run-time library version information during program execution.

Default: false

KMP_WARNINGS

Enables (**true**) or disables (**false**) displaying warnings from the OpenMP run-time library during program execution.

Default: true

LLVM/OpenMP Target Host Runtime (libomptarget)

Environment Variables

libomptarget uses environment variables to control different features of the library at runtime. This allows the user to obtain useful runtime information as well as enable or disable certain features. A full list of supported environment variables is defined below.

- **LIBOMPTARGET_DEBUG=<Num>**
- **LIBOMPTARGET_PROFILE=<Filename>**
- **LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD=<Num>**
- **LIBOMPTARGET_INFO=<Num>**
- **LIBOMPTARGET_HEAP_SIZE=<Num>**
- **LIBOMPTARGET_STACK_SIZE=<Num>**
- **LIBOMPTARGET_SHARED_MEMORY_SIZE=<Num>**
- **LIBOMPTARGET_MAP_FORCE_ATOMIC=[TRUE/FALSE] (default TRUE)**

LIBOMPTARGET_DEBUG

LIBOMPTARGET_DEBUG controls whether or not debugging information will be displayed. This feature is only availible if **libomptarget** was built with **-DOMPTARGET_DEBUG**. The debugging output provided is intended for use by **libomptarget** developers. More user-friendly output is presented when using **LIBOMPTARGET_INFO**.

LIBOMPTARGET_PROFILE

LIBOMPTARGET_PROFILE allows **libomptarget** to generate time profile output similar to Clang's **-ftime-trace** option. This generates a JSON file based on *Chrome Tracing* that can be viewed with **chrome://tracing** or the *Speedscope App*. Building this feature depends on the *LLVM Support Library* for time trace output. Using this library is enabled by default when building using the CMake option **OPENMP_ENABLE_LIBOMPTARGET_PROFILING**. The output will be saved to the filename specified by the environment variable. For multi-threaded applications, profiling in **libomp** is also needed. Setting the CMake option **OPENMP_ENABLE_LIBOMP_PROFILING=ON** to enable the feature. Note that this will turn **libomp** into a C++ library.

LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD

LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD sets the threshold size for which the

libomptarget memory manager will handle the allocation. Any allocations larger than this threshold will not use the memory manager and be freed after the device kernel exits. The default threshold value is **8KB**. If **LIBOMPTARGET_MEMORY_MANAGER_THRESHOLD** is set to **0** the memory manager will be completely disabled.

LIBOMPTARGET_INFO

LIBOMPTARGET_INFO allows the user to request different types of runtime information from **libomptarget**. **LIBOMPTARGET_INFO** uses a 32-bit field to enable or disable different types of information. This includes information about data-mappings and kernel execution. It is recommended to build your application with debugging information enabled, this will enable filenames and variable declarations in the information messages. OpenMP Debugging information is enabled at any level of debugging so a full debug runtime is not required. For minimal debugging information compile with *-gline-tables-only*, or compile with *-g* for full debug information. A full list of flags supported by **LIBOMPTARGET_INFO** is given below.

- Print all data arguments upon entering an OpenMP device kernel: **0x01**
- Indicate when a mapped address already exists in the device mapping table: **0x02**
- Dump the contents of the device pointer map at kernel exit: **0x04**
- Indicate when an entry is changed in the device mapping table: **0x08**
- Print OpenMP kernel information from device plugins: **0x10**
- Indicate when data is copied to and from the device: **0x20**

Any combination of these flags can be used by setting the appropriate bits. For example, to enable printing all data active in an OpenMP target region along with **CUDA** information, run the following **bash** command.

```
$ env LIBOMPTARGET_INFO=$((0x1 | 0x10)) ./your-application
```

Or, to enable every flag run with every bit set.

```
$ env LIBOMPTARGET_INFO=-1 ./your-application
```

For example, given a small application implementing the **ZAXPY** BLAS routine, **Libomptarget** can provide useful information about data mappings and thread usages.

```
#include <complex>

using complex = std::complex<double>;

void zaxpy(complex *X, complex *Y, complex D, std::size_t N) {
#pragma omp target teams distribute parallel for
  for (std::size_t i = 0; i < N; ++i)
    Y[i] = D * X[i] + Y[i];
}

int main() {
  const std::size_t N = 1024;
  complex X[N], Y[N], D;
#pragma omp target data map(to:X[0 : N]) map(tofrom:Y[0 : N])
  zaxpy(X, Y, D, N);
}
```

Compiling this code targeting **nvptx64** with all information enabled will provide the following output from the runtime library.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O3 -gline-tables-only zaxpy.cpp -o zaxpy
$ env LIBOMPTARGET_INFO=-1 ./zaxpy

Info: Entering OpenMP data region at zaxpy.cpp:14:1 with 2 arguments:
Info: to(X[0:N])[16384]
Info: tofrom(Y[0:N])[16384]
Info: Creating new map entry with HstPtrBegin=0x00007fff0d259a40,
    TgtPtrBegin=0x00007fdba5800000, Size=16384, RefCount=1, Name=X[0:N]
Info: Copying data from host to device, HstPtr=0x00007fff0d259a40,
    TgtPtr=0x00007fdba5800000, Size=16384, Name=X[0:N]
Info: Creating new map entry with HstPtrBegin=0x00007fff0d255a40,
    TgtPtrBegin=0x00007fdba5804000, Size=16384, RefCount=1, Name=Y[0:N]
Info: Copying data from host to device, HstPtr=0x00007fff0d255a40,
    TgtPtr=0x00007fdba5804000, Size=16384, Name=Y[0:N]
Info: OpenMP Host-Device pointer mappings after block at zaxpy.cpp:14:1:
Info: Host Ptr          Target Ptr        Size (B) RefCount Declaration
Info: 0x00007fff0d255a40 0x00007fdba5804000 16384    1        Y[0:N] at zaxpy.cpp:13:17
Info: 0x00007fff0d259a40 0x00007fdba5800000 16384    1        X[0:N] at zaxpy.cpp:13:11
Info: Entering OpenMP kernel at zaxpy.cpp:6:1 with 4 arguments:
Info: firstprivate(N)[8] (implicit)
```

```
Info: use_address(Y)[0] (implicit)
Info: tofrom(D)[16] (implicit)
Info: use_address(X)[0] (implicit)
Info: Mapping exists (implicit) with HstPtrBegin=0x00007fff0d255a40,
    TgtPtrBegin=0x00007fdba5804000, Size=0, RefCount=2 (incremented), Name=Y
Info: Creating new map entry with HstPtrBegin=0x00007fff0d2559f0,
    TgtPtrBegin=0x00007fdba5808000, Size=16, RefCount=1, Name=D
Info: Copying data from host to device, HstPtr=0x00007fff0d2559f0,
    TgtPtr=0x00007fdba5808000, Size=16, Name=D
Info: Mapping exists (implicit) with HstPtrBegin=0x00007fff0d259a40,
    TgtPtrBegin=0x00007fdba5800000, Size=0, RefCount=2 (incremented), Name=X
Info: Mapping exists with HstPtrBegin=0x00007fff0d255a40,
    TgtPtrBegin=0x00007fdba5804000, Size=0, RefCount=2 (update suppressed)
Info: Mapping exists with HstPtrBegin=0x00007fff0d2559f0,
    TgtPtrBegin=0x00007fdba5808000, Size=16, RefCount=1 (update suppressed)
Info: Mapping exists with HstPtrBegin=0x00007fff0d259a40,
    TgtPtrBegin=0x00007fdba5800000, Size=0, RefCount=2 (update suppressed)
Info: Launching kernel __omp_offloading_10305_c08c86__Z5zaxpyPSt7complexIdES1_S0_m_l6
    with 8 blocks and 128 threads in SPMD mode
Info: Mapping exists with HstPtrBegin=0x00007fff0d259a40,
    TgtPtrBegin=0x00007fdba5800000, Size=0, RefCount=1 (decremented)
Info: Mapping exists with HstPtrBegin=0x00007fff0d2559f0,
    TgtPtrBegin=0x00007fdba5808000, Size=16, RefCount=1 (deferred final decrement)
Info: Copying data from device to host, TgtPtr=0x00007fdba5808000,
    HstPtr=0x00007fff0d2559f0, Size=16, Name=D
Info: Mapping exists with HstPtrBegin=0x00007fff0d255a40,
    TgtPtrBegin=0x00007fdba5804000, Size=0, RefCount=1 (decremented)
Info: Removing map entry with HstPtrBegin=0x00007fff0d2559f0,
    TgtPtrBegin=0x00007fdba5808000, Size=16, Name=D
Info: OpenMP Host-Device pointer mappings after block at zaxpy.cpp:6:1:
Info: Host Ptr          Target Ptr        Size (B) RefCount Declaration
Info: 0x00007fff0d255a40 0x00007fdba5804000 16384    1        Y[0:N] at zaxpy.cpp:13:17
Info: 0x00007fff0d259a40 0x00007fdba5800000 16384    1        X[0:N] at zaxpy.cpp:13:11
Info: Exiting OpenMP data region at zaxpy.cpp:14:1 with 2 arguments:
Info: to(X[0:N])[16384]
Info: tofrom(Y[0:N])[16384]
Info: Mapping exists with HstPtrBegin=0x00007fff0d255a40,
    TgtPtrBegin=0x00007fdba5804000, Size=16384, RefCount=1 (deferred final decrement)
Info: Copying data from device to host, TgtPtr=0x00007fdba5804000,
    HstPtr=0x00007fff0d255a40, Size=16384, Name=Y[0:N]
```

```
Info: Mapping exists with HstPtrBegin=0x00007fff0d259a40,
    TgtPtrBegin=0x00007fdba5800000, Size=16384, RefCount=1 (deferred final decrement)
Info: Removing map entry with HstPtrBegin=0x00007fff0d255a40,
    TgtPtrBegin=0x00007fdba5804000, Size=16384, Name=Y[0:N]
Info: Removing map entry with HstPtrBegin=0x00007fff0d259a40,
    TgtPtrBegin=0x00007fdba5800000, Size=16384, Name=X[0:N]
```

From this information, we can see the OpenMP kernel being launched on the CUDA device with enough threads and blocks for all **1024** iterations of the loop in simplified *SPMD Mode*. The information from the OpenMP data region shows the two arrays **X** and **Y** being copied from the host to the device. This creates an entry in the host-device mapping table associating the host pointers to the newly created device data. The data mappings in the OpenMP device kernel show the default mappings being used for all the variables used implicitly on the device. Because **X** and **Y** are already mapped in the device's table, no new entries are created. Additionally, the default mapping shows that **D** will be copied back from the device once the OpenMP device kernel region ends even though it isn't written to. Finally, at the end of the OpenMP data region the entries for **X** and **Y** are removed from the table.

The information level can be controlled at runtime using an internal libomptarget library call **__tgt_set_info_flag**. This allows for different levels of information to be enabled or disabled for certain regions of code. Using this requires declaring the function signature as an external function so it can be linked with the runtime library.

```
extern "C" void __tgt_set_info_flag(uint32_t);

extern foo();

int main() {
  __tgt_set_info_flag(0x10);
#pragma omp target
  foo();
}
```

Errors:

libomptarget provides error messages when the program fails inside the OpenMP target region. Common causes of failure could be an invalid pointer access, running out of device memory, or trying to offload when the device is busy. If the application was built with debugging symbols the error messages will additionally provide the source location of the OpenMP target region.

For example, consider the following code that implements a simple parallel reduction on the GPU. This

code has a bug that causes it to fail in the offloading region.

```
#include <cstdio>

double sum(double *A, std::size_t N) {
  double sum = 0.0;
#pragma omp target teams distribute parallel for reduction(+:sum)
  for (int i = 0; i < N; ++i)
    sum += A[i];

  return sum;
}

int main() {
  const int N = 1024;
  double A[N];
  sum(A, N);
}
```

If this code is compiled and run, there will be an error message indicating what is going wrong.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O3 -gline-tables-only sum.cpp -o sum
$ ./sum

CUDA error: an illegal memory access was encountered
Libomptarget error: Copying data from device failed.
Libomptarget error: Call to targetDataEnd failed, abort target.
Libomptarget error: Failed to process data after launching the kernel.
Libomptarget error: Consult https://openmp.llvm.org/design/Runtimes.html for debugging options.
sum.cpp:5:1: Libomptarget error 1: failure of target construct while offloading is mandatory
```

This shows that there is an illegal memory access occuring inside the OpenMP target region once execution has moved to the CUDA device, suggesting a segmentation fault. This then causes a chain reaction of failures in **libomptarget**. Another message suggests using the **LIBOMPTARGET_INFO** environment variable as described in *Environment Variables*. If we do this it will print the sate of the host-target pointer mappings at the time of failure.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O3 -gline-tables-only sum.cpp -o sum
$ env LIBOMPTARGET_INFO=4 ./sum
```

```
info: OpenMP Host-Device pointer mappings after block at sum.cpp:5:1:
info: Host Ptr          Target Ptr        Size (B) RefCount Declaration
info: 0x00007ffc058280f8 0x00007f4186600000 8       1        sum at sum.cpp:4:10
```

This tells us that the only data mapped between the host and the device is the **sum** variable that will be copied back from the device once the reduction has ended. There is no entry mapping the host array **A** to the device. In this situation, the compiler cannot determine the size of the array at compile time so it will simply assume that the pointer is mapped on the device already by default. The solution is to add an explicit map clause in the target region.

```
double sum(double *A, std::size_t N) {
  double sum = 0.0;
#pragma omp target teams distribute parallel for reduction(+:sum) map(to:A[0 : N])
  for (int i = 0; i < N; ++i)
    sum += A[i];

  return sum;
}
```

LIBOMPTARGET_STACK_SIZE

This environment variable sets the stack size in bytes for the CUDA plugin. This can be used to increase or decrease the standard amount of memory reserved for each thread's stack.

LIBOMPTARGET_HEAP_SIZE

This environment variable sets the amount of memory in bytes that can be allocated using **malloc** and **free** for the CUDA plugin. This is necessary for some applications that allocate too much memory either through the user or globalization.

LIBOMPTARGET_SHARED_MEMORY_SIZE

This environment variable sets the amount of dynamic shared memory in bytes used by the kernel once it is launched. A pointer to the dynamic memory buffer can be accessed using the **llvm_omp_target_dynamic_shared_alloc** function. An example is shown in *Dynamic Shared Memory*.

OpenMP in LLVM --- Offloading Design

OpenMP Target Offloading --- SPMD Mode

OpenMP Target Offloading --- Generic Mode

LIBOMPTARGET_MAP_FORCE_ATOMIC

The OpenMP standard guarantees that map clauses are atomic. However, the this can have a drastic performance impact. Users that do not require atomic map clauses can disable them to potentially recover lost performance. As a consequence, users have to guarantee themselves that no two map

clauses will concurrently map the same memory. If the memory is already mapped and the map clauses will only modify the reference counter from a non-zero count to another non-zero count, concurrent map clauses are supported regardless of this option. To disable forced atomic map clauses use "false"/"FALSE" as the value of the **LIBOMPTARGET_MAP_FORCE_ATOMIC** environment variable. The default behavior of LLVM 14 is to force atomic maps clauses, prior versions of LLVM did not.

LLVM/OpenMP Target Host Runtime Plugins (libomptarget.rtl.XXXX)

Remote Offloading Plugin:

The remote offloading plugin permits the execution of OpenMP target regions on devices in remote hosts in addition to the devices connected to the local host. All target devices on the remote host will be exposed to the application as if they were local devices, that is, the remote host CPU or its GPUs can be offloaded to with the appropriate device number. If the server is running on the same host, each device may be identified twice: once through the device plugins and once through the device plugins that the server application has access to.

This plugin consists of **libomptarget.rtl.rpc.so** and **openmp-offloading-server** which should be running on the (remote) host. The server application does not have to be running on a remote host, and can instead be used on the same host in order to debug memory mapping during offloading. These are implemented via gRPC/protobuf so these libraries are required to build and use this plugin. The server must also have access to the necessary target-specific plugins in order to perform the offloading.

Due to the experimental nature of this plugin, the CMake variable **LIBOMPTARGET_ENABLE_EXPERIMENTAL_REMOTE_PLUGIN** must be set in order to build this plugin. For example, the rpc plugin is not designed to be thread-safe, the server cannot concurrently handle offloading from multiple applications at once (it is synchronous) and will terminate after a single execution. Note that **openmp-offloading-server** is unable to remote offload onto a remote host itself and will error out if this is attempted.

Remote offloading is configured via environment variables at runtime of the OpenMP application:

- **LIBOMPTARGET_RPC_ADDRESS=<Address>:<Port>**
- **LIBOMPTARGET_RPC_ALLOCATOR_MAX=<NumBytes>**
- **LIBOMPTARGET_BLOCK_SIZE=<NumBytes>**
- **LIBOMPTARGET_RPC_LATENCY=<Seconds>**

LIBOMPTARGET_RPC_ADDRESS

The address and port at which the server is running. This needs to be set for the server and the application, the default is **0.0.0.0:50051**. A single OpenMP executable can offload onto multiple remote hosts by setting this to comma-seperated values of the addresses.

LIBOMPTARGET_RPC_ALLOCATOR_MAX
After allocating this size, the protobuf allocator will clear. This can be set for both endpoints.

LIBOMPTARGET_BLOCK_SIZE
This is the maximum size of a single message while streaming data transfers between the two endpoints and can be set for both endpoints.

LIBOMPTARGET_RPC_LATENCY
This is the maximum amount of time the client will wait for a response from the server.

LLVM/OpenMP Target Device Runtime (libomptarget-ARCH-SUBARCH.bc)
The target device runtime is an LLVM bitcode library that implements OpenMP runtime functions on the target device. It is linked with the device code's LLVM IR during compilation.

Dynamic Shared Memory
The target device runtime contains a pointer to the dynamic shared memory buffer. This pointer can be obtained using the **llvm_omp_target_dynamic_shared_alloc** extension. If this function is called from the host it will simply return a null pointer. In order to use this buffer the kernel must be launched with an adequate amount of dynamic shared memory allocated. Currently this is done using the **LIBOMPTARGET_SHARED_MEMORY_SIZE** environment variable. An example is given below.

```
void foo() {
  int x;
#pragma omp target parallel map(from : x)
  {
    int *buf = llvm_omp_target_dynamic_shared_alloc();
#pragma omp barrier
    if (omp_get_thread_num() == 0)
      *buf = 1;
#pragma omp barrier
    if (omp_get_thread_num() == 1)
      x = *buf;
  }
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 shared.c
```

```
$ env LIBOMPTARGET_SHARED_MEMORY_SIZE=256 ./shared
```

Debugging

The device runtime supports debugging in the runtime itself. This is configured at compile-time using the flag **-fopenmp-target-debug=<N>** rather than using a separate debugging build. If debugging is not enabled, the debugging paths will be considered trivially dead and removed by the compiler with zero overhead. Debugging is enabled at runtime by running with the environment variable **LIBOMPTARGET_DEVICE_RTL_DEBUG=<N>** set. The number set is a 32-bit field used to selectively enable and disable different features. Currently, the following debugging features are supported.

- Enable debugging assertions in the device. **0x01**
- Enable OpenMP runtime function traces in the device. **0x2**
- Enable diagnosing common problems during offloading . **0x4**

```
void copy(double *X, double *Y) {
#pragma omp target teams distribute parallel for
  for (std::size_t i = 0; i < N; ++i)
    Y[i] = X[i];
}
```

Compiling this code targeting **nvptx64** with debugging enabled will provide the following output from the device runtime library.

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -fopenmp-target-debug=3
$ env LIBOMPTARGET_DEVICE_RTL_DEBUG=3 ./zaxpy

Kernel.cpp:70: Thread 0 Entering int32_t __kmpc_target_init()
Parallelism.cpp:196: Thread 0 Entering int32_t __kmpc_global_thread_num()
Mapping.cpp:239: Thread 0 Entering uint32_t __kmpc_get_hardware_num_threads_in_block()
Workshare.cpp:616: Thread 0 Entering void __kmpc_distribute_static_init_4()
Parallelism.cpp:85: Thread 0 Entering void __kmpc_parallel_51()
  Parallelism.cpp:69: Thread 0 Entering <OpenMP Outlined Function>
    Workshare.cpp:575: Thread 0 Entering void __kmpc_for_static_init_4()
    Workshare.cpp:660: Thread 0 Entering void __kmpc_distribute_static_fini()
Workshare.cpp:660: Thread 0 Entering void __kmpc_distribute_static_fini()
Kernel.cpp:103: Thread 0 Entering void __kmpc_target_deinit()
```

OpenACC support is under development for both Flang and Clang. For this purpose, LLVM's OpenMP runtimes are being extended to serve as OpenACC runtimes. In some cases, Clang supports *OpenMP extensions* to make the additional functionality also available in OpenMP applications.

OPENACC SUPPORT

OpenACC support is under development for both Flang and Clang. For this purpose, LLVM's OpenMP runtimes are being extended to serve as OpenACC runtimes.

OpenMP Extensions for OpenACC

OpenACC provides some functionality that OpenMP does not. In some cases, Clang supports OpenMP extensions to provide similar functionality, taking advantage of the runtime implementation already required for OpenACC. This section documents those extensions.

By default, Clang recognizes these extensions. The command-line option **-fno-openmp-extensions** can be specified to disable all OpenMP extensions, including those described in this section.

Motivation

There are multiple benefits to exposing OpenACC functionality as LLVM OpenMP extensions:

- OpenMP applications can take advantage of the additional functionality.
- As LLVM's implementation of these extensions matures, it can serve as a basis for including these extensions in the OpenMP standard.
- Source-to-source translation from certain OpenACC features to OpenMP is otherwise impossible.
- Runtime tests can be written in terms of OpenMP instead of OpenACC or low-level runtime calls.
- More generally, there is a clean separation of concerns between OpenACC and OpenMP development in LLVM. That is, LLVM's OpenMP developers can discuss, modify, and debug LLVM's extended OpenMP implementation and test suite without directly considering OpenACC's language and execution model, which are handled by LLVM's OpenACC developers.

ompx_hold Map Type Modifier

Example

```
#pragma omp target data map(ompx_hold, tofrom: x) // holds onto mapping of x throughout region
{
  foo(); // might have map(delete: x)
```

```
  #pragma omp target map(present, alloc: x) // x is guaranteed to be present
  printf("%d\n", x);
}
```

The **ompx_hold** map type modifier above specifies that the **target data** directive holds onto the mapping for **x** throughout the associated region regardless of any **target exit data** directives executed during the call to **foo**. Thus, the presence assertion for **x** at the enclosed **target** construct cannot fail.

Behavior

- Stated more generally, the **ompx_hold** map type modifier specifies that the associated data is not unmapped until the end of the construct. As usual, the standard OpenMP reference count for the data must also reach zero before the data is unmapped.

- If **ompx_hold** is specified for the same data on lexically or dynamically enclosed constructs, there is no additional effect as the data mapping is already held throughout their regions.

- The **ompx_hold** map type modifier is permitted to appear only on **target** constructs (and associated combined constructs) and **target data** constructs. It is not permitted to appear on **target enter data** or **target exit data** directives because there is no associated statement, so it is not meaningful to hold onto a mapping until the end of the directive.

- The runtime reports an error if **omp_target_disassociate_ptr** is called for a mapping for which the **ompx_hold** map type modifier is in effect.

- Like the **present** map type modifier, the **ompx_hold** map type modifier applies to an entire struct if it's specified for any member of that struct even if other **map** clauses on the same directive specify other members without the **ompx_hold** map type modifier.

- **ompx_hold** support is not yet provided for **defaultmap**.

Implementation

- LLVM uses the term *dynamic reference count* for the standard OpenMP reference count for host/device data mappings.

- The **ompx_hold** map type modifier selects an alternate reference count, called the *hold reference count*.

- A mapping is removed only once both its reference counts reach zero.

- Because **ompx_hold** can appear only constructs, increments and decrements of the hold reference count are guaranteed to be balanced, so it is impossible to decrement it below zero.

- The dynamic reference count is used wherever **ompx_hold** is not specified (and possibly cannot be specified). Decrementing the dynamic reference count has no effect if it is already zero.

- The runtime determines that the **ompx_hold** map type modifier is *in effect* (see *Behavior* above) when the hold reference count is greater than zero.

Relationship with OpenACC

OpenACC specifies two reference counts for tracking host/device data mappings. Which reference count is used to implement an OpenACC directive is determined by the nature of that directive, either dynamic or structured:

- The *dynamic reference count* is always used for **enter data** and **exit data** directives and corresponding OpenACC routines.

- The *structured reference count* is always used for **data** and compute constructs, which are similar to OpenMP's **target data** and **target** constructs.

  Contrast with OpenMP, where the dynamic reference count is always used unless the application developer specifies an alternate behavior via our map type modifier extension. We chose the name *hold* for that map type modifier because, as demonstrated in the above *example*, *hold* concisely identifies the desired behavior from the application developer's perspective without referencing the implementation of that behavior.

  The hold reference count is otherwise modeled after OpenACC's structured reference count. For example, calling **acc_unmap_data**, which is similar to **omp_target_disassociate_ptr**, is an error when the structured reference count is not zero.

  While Flang and Clang obviously must implement the syntax and semantics for selecting OpenACC reference counts differently than for selecting OpenMP reference counts, the implementation is the same at the runtime level. That is, OpenACC's dynamic reference count is OpenMP's dynamic reference count, and OpenACC's structured reference count is our OpenMP hold reference count extension.

atomic Strictly Nested Within teams

Example

OpenMP 5.2, sec. 10.2 "teams Construct", p. 232, L9-12 restricts what regions can be strictly nested within a **teams** region. As an extension, Clang relaxes that restriction in the case of the **atomic** construct so that, for example, the following case is permitted:

```
#pragma omp target teams map(tofrom:x)
#pragma omp atomic update
x++;
```

Relationship with OpenACC

This extension is important when translating OpenACC to OpenMP because OpenACC does not have the same restriction for its corresponding constructs. For example, the following is conforming OpenACC:

```
#pragma acc parallel copy(x)
#pragma acc atomic update
x++;
```

LLVM, since *version 11* (12 Oct 2020), has an *OpenMP-Aware optimization pass* as well as the ability to *perform "scalar optimizations" across OpenMP region boundaries*.

In-depth discussion of the topic can be found *here*.

OPENMP OPTIMIZATIONS IN LLVM

LLVM, since *version 11* (12 Oct 2020), has an *OpenMP-Aware optimization pass* as well as the ability to *perform "scalar optimizations" across OpenMP region boundaries*.

OpenMP-Aware Optimizations

LLVM, since *version 11* (12 Oct 2020), supports an *OpenMP-Aware optimization pass*. This optimization pass will attempt to optimize the module with OpenMP-specific domain-knowledge. This pass is enabled by default at high optimization levels (O2 / O3) if compiling with OpenMP support enabled.

OpenMPOpt

- *OpenMP Runtime Call Deduplication*
- *Globalization*

OpenMPOpt contains several OpenMP-Aware optimizations. This pass is run early on the entire Module, and later on the entire call graph. Most optimizations done by OpenMPOpt support

remarks. Optimization remarks can be enabled by compiling with the following flags.

```
$ clang -Rpass=openmp-opt -Rpass-missed=openmp-opt -Rpass-analysis=openmp-opt
```

OpenMP Runtime Call Deduplication

The OpenMP runtime library contains several functions used to implement features of the OpenMP standard. Several of the runtime calls are constant within a parallel region. A common optimization is to replace invariant code with a single reference, but in this case the compiler will only see an opaque call into the runtime library. To get around this, OpenMPOpt maintains a list of OpenMP runtime functions that are constant and will manually deduplicate them.

Globalization

The OpenMP standard requires that data can be shared between different threads. This requirement poses a unique challenge when offloading to GPU accelerators. Data cannot be shared between the threads in a GPU by default, in order to do this it must either be placed in global or shared memory. This needs to be done every time a variable may potentially be shared in order to create correct OpenMP programs. Unfortunately, this has significant performance implications and is not needed in the majority of cases. For example, when Clang is generating code for this offloading region, it will see that the variable *x* escapes and is potentially shared. This will require globalizing the variable, which means it cannot reside in the registers on the device.

```
void use(void *) { }

void foo() {
  int x;
  use(&x);
}

int main() {
#pragma omp target parallel
  foo();
}
```

In many cases, this transformation is not actually necessary but still carries a significant performance penalty. Because of this, OpenMPOpt can perform and inter-procedural optimization and scan each known usage of the globalized variable and determine if it is potentially captured and shared by another thread. If it is not actually captured, it can safely be moved back to fast register memory.

Another case is memory that is intentionally shared between the threads, but is shared from one

thread to all the others. Such variables can be moved to shared memory when compiled without needing to go through the runtime library. This allows for users to confidently declare shared memory on the device without needing to use custom OpenMP allocators or rely on the runtime.

```
static void share(void *);

static void foo() {
  int x[64];
#pragma omp parallel
  share(x);
}

int main() {
  #pragma omp target
  foo();
}
```

These optimizations can have very large performance implications. Both of these optimizations rely heavily on inter-procedural analysis. Because of this, offloading applications should ideally be contained in a single translation unit and functions should not be externally visible unless needed. OpenMPOpt will inform the user if any globalization calls remain if remarks are enabled. This should be treated as a defect in the program.

Resources

- 2021 OpenMP Webinar: "A Compiler's View of OpenMP" *https://youtu.be/eIMpgez61r4*
- 2020 LLVM Developers' Meeting: "(OpenMP) Parallelism-Aware Optimizations" *https://youtu.be/gtxWkeLCxmU*
- 2019 EuroLLVM Developers' Meeting: "Compiler Optimizations for (OpenMP) Target Offloading to GPUs" *https://youtu.be/3AbS82C3X30*

OpenMP-Unaware Optimizations
Resources

- 2018 LLVM Developers' Meeting: "Optimizing Indirections, using abstractions without remorse" *https://youtu.be/zfiHaPaoQPc*
- 2019 LLVM Developers' Meeting: "The Attributor: A Versatile Inter-procedural Fixpoint Iteration

Framework" *https://youtu.be/CzWkc_JcfS0*

LLVM has an elaborate ecosystem around *analysis and optimization remarks* issues during compilation. The remarks can be enabled from the clang frontend *[1] [2]* in various formats *[3] [4]* to be used by tools, i.a., *opt-viewer* or *llvm-opt-report* (dated).

The OpenMP optimizations in LLVM have been developed with remark support as a priority. For a list of OpenMP specific remarks and more information on them, please refer to *OpenMP Optimization Remarks*.

- *[1] https://clang.llvm.org/docs/UsersManual.html#options-to-emit-optimization-reports*
- *[2] https://clang.llvm.org/docs/ClangCommandLineReference.html#diagnostic-flags*
- *[3] https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang-foptimization-record-file*
- *[4] https://clang.llvm.org/docs/ClangCommandLineReference.html#cmdoption-clang1-fsave-optimization-record*

OPENMP OPTIMIZATION REMARKS

The *OpenMP-Aware optimization pass* is able to generate compiler remarks for performed and missed optimisations. To emit them, pass these options to the Clang invocation: **-Rpass=openmp-opt -Rpass-analysis=openmp-opt -Rpass-missed=openmp-opt**. For more information and features of the remark system, consult the clang documentation:

- *Clang options to emit optimization reports*
- *Clang diagnostic and remark flags*
- The *-foptimization-record-file flag* and the *-fsave-optimization-record flag*

OpenMP Remarks

Potentially unknown OpenMP target region caller *[OMP100]*

A function remark that indicates the function, when compiled for a GPU, is potentially called from outside the translation unit. Note that a remark is only issued if we tried to perform an optimization which would require us to know all callers on the GPU.

To facilitate OpenMP semantics on GPUs we provide a runtime mechanism through which the code that makes up the body of a parallel region is shared with the threads in the team. Generally we use the

address of the outlined parallel region to identify the code that needs to be executed. If we know all target regions that reach the parallel region we can avoid this function pointer passing scheme and often improve the register usage on the GPU. However, If a parallel region on the GPU is in a function with external linkage we may not know all callers statically. If there are outside callers within target regions, this remark is to be ignored. If there are no such callers, users can modify the linkage and thereby help optimization with a *static* or *__attribute__((internal))* function annotation. If changing the linkage is impossible, e.g., because there are outside callers on the host, one can split the function into an external visible interface which is not compiled for the target and an internal implementation which is compiled for the target and should be called from within the target region.

Parallel region is used in unknown / unexpected ways. Will not attempt to rewrite the state machine. [OMP101]

An analysis remark that indicates that a parallel region has unknown calls.

Parallel region is not called from a unique kernel. Will not attempt to rewrite the state machine. [OMP102]

This analysis remark indicates that a given parallel region is called by multiple kernels. This prevents the compiler from optimizing it to a single kernel and rewrite the state machine.

Moving globalized variable to the stack. [OMP110]

This optimization remark indicates that a globalized variable was moved back to thread-local stack memory on the device. This occurs when the optimization pass can determine that a globalized variable cannot possibly be shared between threads and globalization was ultimately unnecessary. Using stack memory is the best-case scenario for data globalization as the variable can now be stored in fast register files on the device. This optimization requires full visibility of each variable.

Globalization typically occurs when a pointer to a thread-local variable escapes the current scope. The compiler needs to be pessimistic and assume that the pointer could be shared between multiple threads according to the OpenMP standard. This is expensive on target offloading devices that do not allow threads to share data by default. Instead, this data must be moved to memory that can be shared, such as shared or global memory. This optimization moves the data back from shared or global memory to thread-local stack memory if the data is not actually shared between the threads.

Examples

A trivial example of globalization occurring can be seen with this example. The compiler sees that a pointer to the thread-local variable **x** escapes the current scope and must globalize it even though it is not actually necessary. Fortunately, this optimization can undo this by looking at its usage.

```
void use(int *x) { }
```

```
void foo() {
  int x;
  use(&x);
}

int main() {
#pragma omp target parallel
  foo();
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 omp110.cpp -O1 -Rpass=openmp-opt
omp110.cpp:6:7: remark: Moving globalized variable to the stack. [OMP110]
  int x;
      ^
```

A less trivial example can be seen using C++'s complex numbers. In this case the overloaded arithmetic operators cause pointers to the complex numbers to escape the current scope, but they can again be removed once the usage is visible.

```
#include <complex>

using complex = std::complex<double>;

void zaxpy(complex *X, complex *Y, const complex D, int N) {
#pragma omp target teams distribute parallel for firstprivate(D)
  for (int i = 0; i < N; ++i)
    Y[i] = D * X[i] + Y[i];
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 omp110.cpp -O1 -Rpass=openmp-opt
In file included from omp110.cpp:1:
In file included from /usr/bin/clang/lib/clang/13.0.0/include/openmp_wrappers/complex:27:
/usr/include/c++/8/complex:328:20: remark: Moving globalized variable to the stack. [OMP110]
      complex<_Tp> __r = __x;
                   ^
/usr/include/c++/8/complex:388:20: remark: Moving globalized variable to the stack. [OMP110]
      complex<_Tp> __r = __x;
                   ^
```

Diagnostic Scope

OpenMP target offloading optimization remark.

Replaced globalized variable with X bytes of shared memory. [OMP111]

This optimization occurs when a globalized variable's data is shared between multiple threads, but requires a constant amount of memory that can be determined at compile time. This is the case when only a single thread creates the memory and is then shared between every thread. The memory can then be pushed to a static buffer of shared memory on the device. This optimization allows users to declare shared memory on the device without using OpenMP's custom allocators.

Globalization occurs when a pointer to a thread-local variable escapes the current scope. If a single thread is known to be responsible for creating and sharing the data it can instead be mapped directly to the device's shared memory. Checking if only a single thread can execute an instruction requires that the parent functions have internal linkage. Otherwise, an external caller could invalidate this analysis but having multiple threads call that function. The optimization pass will make internal copies of each function to use for this reason, but it is still recommended to mark them as internal using keywords like **static** whenever possible.

Example

This optimization should apply to any variable declared in an OpenMP target region that is then shared with every thread in a parallel region. This allows the user to declare shared memory without using custom allocators. A simple stencil calculation shows how this can be used.

```
void stencil(int M, int N, double *X, double *Y) {
#pragma omp target teams distribute collapse(2) \
  map(to : X [0:M * N]) map(tofrom : Y [0:M * N])
  for (int i0 = 0; i0 < M; i0 += MC) {
    for (int j0 = 0; j0 < N; j0 += NC) {
      double sX[MC][NC];

#pragma omp parallel for collapse(2) shared(sX) default(firstprivate)
      for (int i1 = 0; i1 < MC; ++i1)
        for (int j1 = 0; j1 < NC; ++j1)
          sX[i1][j1] = X[(i0 + i1) * N + (j0 + j1)];

#pragma omp parallel for collapse(2) shared(sX) default(firstprivate)
      for (int i1 = 1; i1 < MC - 1; ++i1)
        for (int j1 = 1; j1 < NC - 1; ++j1)
          Y[(i0 + i1) * N + j0 * j1] = (sX[i1 + 1][j1] + sX[i1 - 1][j1] +
                                        sX[i1][j1 + 1] + sX[i1][j1 - 1] +
                                        -4.0 * sX[i1][j1]) / (dX * dX);
```

```
    }
  }
}
```

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O1 -Rpass=openmp-opt -fopenmp-version=51 omp111.cpp
omp111.cpp:10:14: remark: Replaced globalized variable with 8192 bytes of shared memory. [OMP111]
  double sX[MC][NC];
         ^
```

The default mapping for variables captured in an OpenMP parallel region is **shared**. This means taking a pointer to the object which will ultimately result in globalization that will be mapped to shared memory when it could have been placed in registers. To avoid this, make sure each variable that can be copied into the region is marked **firstprivate** either explicitly or using the OpenMP 5.1 feature **default(firstprivate)**.

Diagnostic Scope

OpenMP target offloading optimization remark.

Found thread data sharing on the GPU. Expect degraded performance due to data globalization. [OMP112]

This missed remark indicates that a globalized value was found on the target device that was not either replaced with stack memory by *OMP110* or shared memory by *OMP111*. Globalization that has not been removed will need to be handled by the runtime and will significantly impact performance.

The OpenMP standard requires that threads are able to share their data between each-other. However, this is not true by default when offloading to a target device such as a GPU. Threads on a GPU cannot shared their data unless it is first placed in global or shared memory. In order to create standards complaint code, the Clang compiler will globalize any variables that could potentially be shared between the threads. In the majority of cases, globalized variables can either be returns to a thread-local stack, or pushed to shared memory. However, in a few cases it is necessary and will cause a performance penalty.

Examples

This example shows legitimate data sharing on the device. It is a convoluted example, but is completely complaint with the OpenMP standard. If globalization was not added this would result in different results on different target devices.

```
#include <omp.h>
#include <cstdio>
```

```
#pragma omp declare target
static int *p;
#pragma omp end declare target

void foo() {
  int x = omp_get_thread_num();
  if (omp_get_thread_num() == 1)
    p = &x;

#pragma omp barrier

  printf ("Thread %d: %d\n", omp_get_thread_num(), *p);
}

int main() {
#pragma omp target parallel
  foo();
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O1 -Rpass-missed=openmp-opt omp112.cpp
omp112.cpp:9:7: remark: Found thread data sharing on the GPU. Expect degraded performance
due to data globalization. [OMP112] [-Rpass-missed=openmp-opt]
int x = omp_get_thread_num();
    ^
```

A less convoluted example globalization that cannot be removed occurs when calling functions that aren’t visible from the current translation unit.

```
extern void use(int *x);

void foo() {
  int x;
  use(&x);
}

int main() {
#pragma omp target parallel
  foo();
}
```

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O1 -Rpass-missed=openmp-opt omp112.cpp
omp112.cpp:4:7: remark: Found thread data sharing on the GPU. Expect degraded performance
due to data globalization. [OMP112] [-Rpass-missed=openmp-opt]
int x;
  ^
```

Diagnostic Scope

OpenMP target offloading missed remark.

Could not move globalized variable to the stack. Variable is potentially captured in call. Mark parameter as *__attribute__((noescape))* to override. [OMP113]

This missed remark indicates that a globalized value could not be moved to the stack because it is potentially captured by a call to a function we cannot analyze. In order for a globalized variable to be moved to the stack, copies to its pointer cannot be stored. Otherwise it is considered captured and could potentially be shared between the threads. This can be overridden using a parameter level attribute as suggested in the remark text.

Globalization will occur when a pointer to a thread-local variable escapes the current scope. In most cases it can be determined that the variable cannot be shared if a copy of its pointer is never made. However, this remark indicates a copy of the pointer is present or that sharing is possible because it is used outside the current translation unit.

Examples

If a pointer to a thread-local variable is passed to a function not visible in the current translation unit we need to assume a copy is made of it that can be shared between the threads. This prevents *OMP110* from triggering, which will result in a performance penalty when executing on the target device.

```
extern void use(int *x);

void foo() {
  int x;
  use(&x);
}

int main() {
#pragma omp target parallel
  foo();
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass-missed=openmp-opt omp113.cpp
```

```
missed.cpp:4:7: remark: Could not move globalized variable to the stack. Variable is
potentially captured in call. Mark parameter as ‘__attribute__((noescape))‘ to
override. [OMP113]
  int x;
      ^
```

As the remark suggests, this behaviour can be overridden using the **noescape** attribute. This tells the compiler that no reference to the object the pointer points to that is derived from the parameter value will survive after the function returns. The user is responsible for verifying that this assertion is correct.

```
extern void use(__attribute__((noescape)) int *x);

void foo() {
  int x;
  use(&x);
}

int main() {
#pragma omp target parallel
  foo();
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass=openmp-opt omp113.cpp
missed.cpp:4:7: remark: Moving globalized variable to the stack. [OMP110]
int x;
    ^
```

Diagnostic Scope

OpenMP target offloading missed remark.

Transformed generic-mode kernel to SPMD-mode [OMP120]

This optimization remark indicates that the execution strategy for the OpenMP target offloading kernel was changed. Generic-mode kernels are executed by a single thread that schedules parallel worker threads using a state machine. This code transformation can move a kernel that was initially generated in generic mode to SPMD-mode where all threads are active at the same time with no state machine. This execution strategy is closer to how the threads are actually executed on a GPU target. This is only possible if the instructions previously executed by a single thread have no side-effects or can be guarded. If the instructions have no side-effects they are simply recomputed by each thread.

Generic-mode is often considerably slower than SPMD-mode because of the extra overhead required to separately schedule worker threads and pass data between them.This optimization allows users to use generic-mode semantics while achieving the performance of SPMD-mode. This can be helpful when defining shared memory between the threads using *OMP111*.

Examples

Normally, any kernel that contains split OpenMP target and parallel regions will be executed in generic-mode. Sometimes it is easier to use generic-mode semantics to define shared memory, or more tightly control the distribution of the threads. This shows a naive matrix-matrix multiplication that contains code that will need to be guarded.

```
void matmul(int M, int N, int K, double *A, double *B, double *C) {
#pragma omp target teams distribute collapse(2) \
  map(to:A[0: M*K]) map(to:B[0: K*N]) map(tofrom:C[0 : M*N])
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
      double sum = 0.0;

#pragma omp parallel for reduction(+:sum) default(firstprivate)
      for (int k = 0; k < K; k++)
        sum += A[i*K + k] * B[k*N + j];

      C[i*N + j] = sum;
    }
  }
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -fopenmp-version=51 -O2 -Rpass=openmp-opt omp120.cpp
omp120.cpp:6:14: remark: Replaced globalized variable with 8 bytes of shared memory. [OMP111]
    double sum = 0.0;
           ^
omp120.cpp:2:1: remark: Transformed generic-mode kernel to SPMD-mode. [OMP120]
#pragma omp target teams distribute collapse(2) \
^
```

This requires guarding the store to the shared variable **sum** and the store to the matrix **C**. This can be thought of as generating the code below.

```
void matmul(int M, int N, int K, double *A, double *B, double *C) {
#pragma omp target teams distribute collapse(2) \
```

```
  map(to:A[0: M*K]) map(to:B[0: K*N]) map(tofrom:C[0 : M*N])
  for (int i = 0; i < M; i++) {
    for (int j = 0; j < N; j++) {
    double sum;
#pragma omp parallel default(firstprivate) shared(sum)
    {
    #pragma omp barrier
    if (omp_get_thread_num() == 0)
      sum = 0.0;
    #pragma omp barrier

#pragma omp for reduction(+:sum)
      for (int k = 0; k < K; k++)
        sum += A[i*K + k] * B[k*N + j];

    #pragma omp barrier
    if (omp_get_thread_num() == 0)
      C[i*N + j] = sum;
    #pragma omp barrier
    }
    }
  }
}
```

Diagnostic Scope

OpenMP target offloading optimization remark.

Value has potential side effects preventing SPMD-mode execution. Add *__attribute__((assume("ompx_spmd_amenable")))* to the called function to override. [OMP121]

This analysis remarks indicates that a potential side-effect that cannot be guarded prevents the target region from executing in SPMD-mode. SPMD-mode requires that each thread is active inside the region. Any instruction that cannot be either recomputed by each thread independently or guarded and executed by a single thread prevents the region from executing in SPMD-mode.

This remark will attempt to print out the instructions preventing the region from being executed in SPMD-mode. Calls to functions outside the current translation unit will prevent this transformation from occurring as well, but can be overridden using an assumption stating that it contains no calls that prevent SPMD execution.

Examples

Calls to functions outside the current translation unit may contain instructions or operations that cannot be executed in SPMD-mode.

```
extern int work();

void use(int x);

void foo() {
#pragma omp target teams
  {
    int x = work();
#pragma omp parallel
      use(x);

  }
}
```

```
$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass-analysis=openmp-opt omp121.cpp
omp121.cpp:8:13: remark: Value has potential side effects preventing SPMD-mode
execution.  Add ‘__attribute__((assume("ompx_spmd_amenable")))‘ to the called function
to override. [OMP121]
int x = work();
        ^
```

As the remark suggests, the problem is caused by the unknown call to the external function **work**. This can be overridden by asserting that it does not contain any code that prevents SPMD-mode execution.

```
__attribute__((assume("ompx_spmd_amenable"))) extern int work();

void use(int x);

void foo() {
#pragma omp target teams
  {
    int x = work();
#pragma omp parallel
      use(x);

  }
```

```
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass=openmp-opt omp121.cpp
omp121.cpp:6:1: remark: Transformed generic-mode kernel to SPMD-mode. [OMP120]
#pragma omp target teams
^
```

Diagnostic Scope

OpenMP target offloading analysis remark.

Removing unused state machine from generic-mode kernel. [OMP130]

This optimization remark indicates that an unused state machine was removed from a target region. This occurs when there are no parallel regions inside of a target construct. Normally, a state machine is required to schedule the threads inside of a parallel region. If there are no parallel regions, the state machine is unnecessary because there is only a single thread active at any time.

Examples

This optimization should occur on any target region that does not contain any parallel work.

```
void copy(int N, double *X, double *Y) {
#pragma omp target teams distribute map(tofrom: X[0:N]) map(tofrom: Y[0:N])
  for (int i = 0; i < N; ++i)
    Y[i] = X[i];
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass=openmp-opt omp130.cpp
omp130.cpp:2:1: remark: Removing unused state machine from generic-mode kernel. [OMP130]
#pragma omp target teams distribute map(tofrom: X[0:N]) map(tofrom: Y[0:N])
^
```

Diagnostic Scope

OpenMP target offloading optimization remark.

Rewriting generic-mode kernel with a customized state machine. [OMP131]

This optimization remark indicates that a generic-mode kernel on the device was specialized for the given target region. When offloading in generic-mode, a state machine is required to schedule the work between the parallel worker threads. This optimization specializes the state machine in cases where there is a known number of parallel regions inside the kernel. A much simpler state machine can be used if it is known that there is no nested parallelism and the number of regions to schedule is a static amount.

Examples

This optimization should occur on any generic-mode kernel that has visibility on all parallel regions, but cannot be moved to SPMD-mode and has no nested parallelism.

```
#pragma omp declare target
int TID;
#pragma omp end declare target

void foo() {
#pragma omp target
{
 TID = omp_get_thread_num();
 #pragma omp parallel
 {
   work();
 }
}
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass=openmp-opt omp131.cpp
omp131.cpp:8:1: remark: Rewriting generic-mode kernel with a customized state machine. [OMP131]
#pragma omp target
^
```

Diagnostic Scope

OpenMP target offloading optimization remark.

Generic-mode kernel is executed with a customized state machine that requires a fallback. [OMP132]

This analysis remark indicates that a state machine rewrite occurred, but could not be done fully because of unknown calls to functions that may contain parallel regions. The state machine handles scheduling work between parallel worker threads on the device when operating in generic-mode. If there are unknown parallel regions it prevents the optimization from fully rewriting the state machine.

Examples

This will occur for any generic-mode kernel that may contain unknown parallel regions. This is typically coupled with the *OMP133* remark.

```
extern void setup();

void foo() {
```

```
#pragma omp target
{
 setup();
 #pragma omp parallel
 {
  work();
 }
}
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass-analysis=openmp-opt omp132.cpp
omp133.cpp:4:1: remark: Generic-mode kernel is executed with a customized state machine
that requires a fallback. [OMP132]
#pragma omp target
^
```

Diagnostic Scope

OpenMP target offloading analysis remark.

Call may contain unknown parallel regions. Use *__attribute__((assume("omp_no_parallelism")))* to override. [OMP133]

This analysis remark identifies calls that prevented *OMP131* from providing the generic-mode kernel with a fully specialized state machine. This remark will identify each call that may contain unknown parallel regions that caused the kernel to require a fallback.

Examples

This will occur for any generic-mode kernel that may contain unknown parallel regions. This is typically coupled with the *OMP132* remark.

```
extern void setup();

void foo() {
#pragma omp target
{
 setup();
 #pragma omp parallel
 {
  work();
 }
}
```

```
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass-analysis=openmp-opt omp133.cpp
omp133.cpp:6:5: remark: Call may contain unknown parallel regions. Use
‘__attribute__((assume("omp_no_parallelism")))‘ to override. [OMP133]
setup();
^
```

The remark suggests marking the function with the assumption that it contains no parallel regions. If this is done then the kernel will be rewritten with a fully specialized state machine.

```
__attribute__((assume("omp_no_parallelism"))) extern void setup();


void foo() {
#pragma omp target
{
 setup();
 #pragma omp parallel
 {
  work();
 }
}
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O2 -Rpass=openmp-opt omp133.cpp
omp133.cpp:4:1: remark: Rewriting generic-mode kernel with a customized state machine. [OMP131]
#pragma omp target
^
```

Diagnostic Scope

OpenMP target offloading analysis remark.

Could not internalize function. Some optimizations may not be possible. [OMP140]

This analysis remark indicates that function internalization failed for the given function. Internalization occurs when a call to a function that ordinarily has external visibility is replaced with a call to a copy of that function with only internal visibility. This allows the compiler to make strong static assertions about the context a function is called in. Without internalization this analysis would always be invalidated by the possibility of someone calling the function in a different context outside of the current translation unit. This is necessary for optimizations like *OMP111* and *OMP120*. If a function

failed to be internalized it most likely has linkage that cannot be copied. Internalization is currently only enabled by default for OpenMP target offloading.

Examples

This will occur for any function declaration that has incompatible linkage.

```
__attribute__((weak)) void setup();

void foo() {
#pragma omp target
{
  setup();
  #pragma omp parallel
  {
    work();
  }
}
}

$ clang++ -fopenmp -fopenmp-targets=nvptx64 -O1 -Rpass-analysis=openmp-opt omp140.cpp
omp140.cpp:1:1: remark: Could not internalize function. Some optimizations may not
be possible. [OMP140]
__attribute__((weak)) void setup() {
^
```

Diagnostic Scope

OpenMP analysis remark.

Parallel region merged with parallel region at <location>. [OMP150]

This optimization remark indicates that a parallel region was merged with others into a single parallel region. Parallel region merging fuses consecutive parallel regions to reduce the team activation overhead of forking and increases the scope of possible OpenMP-specific optimizations within merged parallel regions. This optimization can also guard sequential code between two parallel regions if applicable.

Example

This optimization should apply to any compatible and consecutive parallel regions. In this case the sequential region between the parallel regions will be guarded so it is only executed by a single thread in the new merged region.

```
void foo() {
#pragma omp parallel
  parallel_work();

  sequential_work();

#pragma omp parallel
  parallel_work();
}

$ clang++ -fopenmp -O2 -Rpass=openmp-opt -mllvm -openmp-opt-enable-merging omp150.cpp
omp150.cpp:2:1: remark: Parallel region merged with parallel region at merge.cpp:7:1. [OMP150]
#pragma omp parallel
^
```

Diagnostic Scope

OpenMP optimization remark.

Removing parallel region with no side-effects. [OMP160]

This optimization remark indicates that a parallel region was deleted because it was not found to have any side-effects. This can occur if the region does not write any of its results to memory visible outside the region. This optimization is necessary because the barrier between sequential and parallel code typically prevents dead code elimination from completely removing the region. Otherwise there will still be overhead to fork and merge the threads with no work done.

Example

This optimization occurs whenever a parallel region was not found to have any side-effects. This can occur if the parallel region only reads memory or is simply empty.

```
void foo() {
#pragma omp parallel
  { }
#pragma omp parallel
  { int x = 1; }
}
}

$ clang++ -fopenmp -O2 -Rpass=openmp-opt omp160.cpp
omp160.cpp:4:1: remark: Removing parallel region with no side-effects. [OMP160] [-Rpass=openmp-opt]
#pragma omp parallel
```

```
^
delete.cpp:2:1: remark: Removing parallel region with no side-effects. [OMP160] [-Rpass=openmp-opt]
#pragma omp parallel
^
^
```

Diagnostic Scope

OpenMP optimization remark.

OpenMP runtime call <call> deduplicated. [OMP170]

This optimization remark indicates that a call to an OpenMP runtime call was replaced with the result of an existing one. This occurs when the compiler knows that the result of a runtime call is immutable. Removing duplicate calls is done by replacing all calls to that function with the result of the first call. This cannot be done automatically by the compiler because the implementations of the OpenMP runtime calls live in a separate library the compiler cannot see.

Example

This optimization will trigger for known OpenMP runtime calls whose return value will not change.

```
void foo(int N) {
  double *A = malloc(N * omp_get_thread_limit());
  double *B = malloc(N * omp_get_thread_limit());

#pragma omp parallel
  work(&A[omp_get_thread_num() * N]);
#pragma omp parallel
  work(&B[omp_get_thread_num() * N]);
}

$ clang -fopenmp -O2 -Rpass=openmp-opt omp170.c
ompi170.c:2:26: remark: OpenMP runtime call omp_get_thread_limit deduplicated. [OMP170]
double *A = malloc(N * omp_get_thread_limit());
                       ^
```

Diagnostic Scope

OpenMP optimization remark.

Replacing OpenMP runtime call <call> with <value>.

This optimization remark indicates that analysis determined an OpenMP runtime calls can be replaced with a constant value. This can occur when an OpenMP runtime call that queried some internal state

was found to always return a single value after analysis.

Example

This optimization will trigger for most target regions to simplify the runtime once certain constants are known. This will trigger for internal runtime functions so it requires enabling verbose remarks with *-openmp-opt-verbose-remarks* (prefixed with *-mllvm* for use with clang).

```
void foo() {
#pragma omp target parallel
  { }
}

$ clang test.c -fopenmp -fopenmp-targets=nvptx64 -O1 -Rpass=openmp-opt \
  -mllvm -openmp-opt-verbose-remarks
remark: Replacing runtime call __kmpc_is_spmd_exec_mode with 1. [OMP180] [-Rpass=openmp-opt]
remark: Replacing runtime call __kmpc_is_spmd_exec_mode with 1. [OMP180] [-Rpass=openmp-opt]
remark: Replacing runtime call __kmpc_parallel_level with 1. [OMP180] [-Rpass=openmp-opt]
remark: Replacing runtime call __kmpc_parallel_level with 1. [OMP180] [-Rpass=openmp-opt]
```

Diagnostic Scope

OpenMP optimization remark.

Redundant barrier eliminated. (device only)

This optimization remark indicates that analysis determined an aligned barrier in the device code to be redundant. This can occur when state updates that have been synchronized by the barrier were eliminated too. See also "Co-Designing an OpenMP GPU Runtime and Optimizations for Near-Zero Overhead Execution", IPDPS’22.

Example

This optimization will trigger for most target regions if state initialization was removed as a consequence of "state forwarding". This will trigger for internal runtime functions so it requires enabling verbose remarks with *-openmp-opt-verbose-remarks* (prefixed with *-mllvm* for use with clang).

Diagnostic Scope

OpenMP optimization remark.

Diagnostics Number	Diagnostics Kind	Diagnostics Description

OMP100	Analysis	Potentially unknown OpenMP target region caller.
OMP101	Analysis	Parallel region is used in unknown / unexpected ways. Will not attempt to rewrite the state machine.
OMP102	Analysis	Parallel region is not called from a unique kernel. Will not attempt to rewrite the state machine.
OMP110	Optimization	Moving globalized variable to the stack.
OMP111	Optimization	Replaced globalized variable with X bytes of shared memory.
OMP112	Missed	Found thread data sharing on the GPU. Expect degraded performance due to data globalization.
OMP113	Missed	Could not move globalized variable to the stack. Variable is potentially captured in call. Mark parameter as *__attribute__((noescape))* to override.
OMP120	Optimization	Transformed generic-mode kernel to SPMD-mode.
OMP121	Analysis	Value has potential side effects preventing SPMD-mode execution. Add *__attribute__((assume("ompx_spmd_amenable")))* to the called function to override.
OMP130	Optimization	Removing unused state machine from generic-mode kernel.
OMP131	Optimization	Rewriting generic-mode kernel with a customized state machine.
OMP132	Analysis	Generic-mode kernel is executed with a

		|customized state machine that requires a fallback.
OMP133	Analysis	Call may contain unknown parallel regions. Use *__attribute__((assume("omp_no_parallelism")))* to override.
OMP140	Analysis	Could not internalize function. Some optimizations may not be possible.
OMP150	Optimization	Parallel region merged with parallel region at <location>.
OMP160	Optimization	Removing parallel region with no side-effects.
OMP170	Optimization	OpenMP runtime call <call> deduplicated.
OMP180	Optimization	Replacing OpenMP runtime call <call> with <value>.
OMP190	Optimization	Redundant barrier eliminated. (device only)

Dealing with OpenMP can be complicated. For help with the setup of an OpenMP (offload) capable compiler toolchain, its usage, and common problems, consult the *Support and FAQ* page.

We also encourage everyone interested in OpenMP in LLVM to *get involved*.

SUPPORT, GETTING INVOLVED, AND FAQ

Please do not hesitate to reach out to us via *openmp-dev@lists.llvm.org* or join one of our *regular calls*. Some common questions are answered in the *FAQ*.

Calls

OpenMP in LLVM Technical Call

- Development updates on OpenMP (and OpenACC) in the LLVM Project, including Clang, optimization, and runtime work.

- Join *OpenMP in LLVM Technical Call*.
- Time: Weekly call on every Wednesday 7:00 AM Pacific time.
- Meeting minutes are *here*.
- Status tracking *page*.

OpenMP in Flang Technical Call

- Development updates on OpenMP and OpenACC in the Flang Project.
- Join *OpenMP in Flang Technical Call*
- Time: Weekly call on every Thursdays 8:00 AM Pacific time.
- Meeting minutes are *here*.
- Status tracking *page*.

FAQ

NOTE:

The FAQ is a work in progress and most of the expected content is not yet available. While you can expect changes, we always welcome feedback and additions. Please contact, e.g., through **openmp-dev@lists.llvm.org**.

Q: How to contribute a patch to the webpage or any other part?

All patches go through the regular *LLVM review process*.

Q: How to build an OpenMP GPU offload capable compiler?

To build an *effective* OpenMP offload capable compiler, only one extra CMake option, *LLVM_ENABLE_RUNTIMES="openmp"*, is needed when building LLVM (Generic information about building LLVM is available *here*.). Make sure all backends that are targeted by OpenMP to be enabled. By default, Clang will be built with all backends enabled. When building with *LLVM_ENABLE_RUNTIMES="openmp"* OpenMP should not be enabled in *LLVM_ENABLE_PROJECTS* because it is enabled by default.

For Nvidia offload, please see *Q: How to build an OpenMP NVidia offload capable compiler?*. For AMDGPU offload, please see *Q: How to build an OpenMP AMDGPU offload capable compiler?*.

NOTE:

The compiler that generates the offload code should be the same (version) as the compiler that builds the OpenMP device runtimes. The OpenMP host runtime can be built by a different compiler.

Q: How to build an OpenMP NVidia offload capable compiler?

The Cuda SDK is required on the machine that will execute the openmp application.

If your build machine is not the target machine or automatic detection of the available GPUs failed, you should also set:

- *CLANG_OPENMP_NVPTX_DEFAULT_ARCH=sm_XX* where *XX* is the architecture of your GPU, e.g, 80.

- *LIBOMPTARGET_NVPTX_COMPUTE_CAPABILITIES=YY* where *YY* is the numeric compute capacity of your GPU, e.g., 75.

Q: How to build an OpenMP AMDGPU offload capable compiler?

A subset of the *ROCm* toolchain is required to build the LLVM toolchain and to execute the openmp application. Either install ROCm somewhere that cmake's find_package can locate it, or build the required subcomponents ROCt and ROCr from source.

The two components used are ROCT-Thunk-Interface, roct, and ROCR-Runtime, rocr. Roct is the userspace part of the linux driver. It calls into the driver which ships with the linux kernel. It is an implementation detail of Rocr from OpenMP's perspective. Rocr is an implementation of *HSA*.

```
SOURCE_DIR=same-as-llvm-source # e.g. the checkout of llvm-project, next to openmp
BUILD_DIR=somewhere
INSTALL_PREFIX=same-as-llvm-install

cd $SOURCE_DIR
git clone git@github.com:RadeonOpenCompute/ROCT-Thunk-Interface.git -b roc-4.2.x \
  --single-branch
git clone git@github.com:RadeonOpenCompute/ROCR-Runtime.git -b rocm-4.2.x \
  --single-branch

cd $BUILD_DIR && mkdir roct && cd roct
cmake $SOURCE_DIR/ROCT-Thunk-Interface/ -DCMAKE_INSTALL_PREFIX=$INSTALL_PREFIX \
  -DCMAKE_BUILD_TYPE=Release -DBUILD_SHARED_LIBS=OFF
make && make install
```

```
cd $BUILD_DIR && mkdir rocr && cd rocr
cmake $SOURCE_DIR/ROCR-Runtime/src -DIMAGE_SUPPORT=OFF \
 -DCMAKE_INSTALL_PREFIX=$INSTALL_PREFIX -DCMAKE_BUILD_TYPE=Release \
 -DBUILD_SHARED_LIBS=ON
make && make install
```

IMAGE_SUPPORT requires building rocr with clang and is not used by openmp.

Provided cmake's find_package can find the ROCR-Runtime package, LLVM will build a tool **bin/amdgpu-arch** which will print a string like **gfx906** when run if it recognises a GPU on the local system. LLVM will also build a shared library, libomptarget.rtl.amdgpu.so, which is linked against rocr.

With those libraries installed, then LLVM build and installed, try:

```
clang -O2 -fopenmp -fopenmp-targets=amdgcn-amd-amdhsa example.c -o example && ./example
```

Q: What are the known limitations of OpenMP AMDGPU offload?

LD_LIBRARY_PATH or rpath/runpath are required to find libomp.so and libomptarget.so

There is no libc. That is, malloc and printf do not exist. Libm is implemented in terms of the rocm device library, which will be searched for if linking with '-lm'.

Some versions of the driver for the radeon vii (gfx906) will error unless the environment variable 'export HSA_IGNORE_SRAMECC_MISREPORT=1' is set.

It is a recent addition to LLVM and the implementation differs from that which has been shipping in ROCm and AOMP for some time. Early adopters will encounter bugs.

Q: What are the LLVM components used in offloading and how are they found?

The libraries used by an executable compiled for target offloading are:

- **libomp.so** (or similar), the host openmp runtime
- **libomptarget.so**, the target-agnostic target offloading openmp runtime
- plugins loaded by libomptarget.so:
  - **libomptarget.rtl.amdgpu.so**

- **libomptarget.rtl.cuda.so**
- **libomptarget.rtl.x86_64.so**
- **libomptarget.rtl.ve.so**
- and others

- dependencies of those plugins, e.g. cuda/rocr for nvptx/amdgpu

The compiled executable is dynamically linked against a host runtime, e.g. **libomp.so**, and against the target offloading runtime, **libomptarget.so**. These are found like any other dynamic library, by setting rpath or runpath on the executable, by setting **LD_LIBRARY_PATH**, or by adding them to the system search.

libomptarget.so has rpath or runpath (whichever the system default is) set to **$ORIGIN**, and the plugins are located next to it, so it will find the plugins without any environment variables set. If **LD_LIBRARY_PATH** is set, whether it overrides which plugin is found depends on whether your system treats **-Wl,-rpath** as RPATH or RUNPATH.

The plugins will try to find their dependencies in plugin-dependent fashion.

The cuda plugin is dynamically linked against libcuda if cmake found it at compiler build time. Otherwise it will attempt to dlopen **libcuda.so**. It does not have rpath set.

The amdgpu plugin is linked against ROCr if cmake found it at compiler build time. Otherwise it will attempt to dlopen **libhsa-runtime64.so**. It has rpath set to **$ORIGIN**, so installing **libhsa-runtime64.so** in the same directory is a way to locate it without environment variables.

In addition to those, there is a compiler runtime library called deviceRTL. This is compiled from mostly common code into an architecture specific bitcode library, e.g. **libomptarget-nvptx-sm_70.bc**.

Clang and the deviceRTL need to match closely as the interface between them changes frequently. Using both from the same monorepo checkout is strongly recommended.

Unlike the host side which lets environment variables select components, the deviceRTL that is located in the clang lib directory is preferred. Only if it is absent, the **LIBRARY_PATH** environment variable is searched to find a bitcode file with the right name. This can be overridden by passing a clang flag, **--libomptarget-nvptx-bc-path** or

--libomptarget-amdgcn-bc-path. That can specify a directory or an exact bitcode file to use.

Q: Does OpenMP offloading support work in pre-packaged LLVM releases?
For now, the answer is most likely *no*. Please see *Q: How to build an OpenMP GPU offload capable compiler?*.

Q: Does OpenMP offloading support work in packages distributed as part of my OS?
For now, the answer is most likely *no*. Please see *Q: How to build an OpenMP GPU offload capable compiler?*.

Q: Does Clang support <*math.h*> **and** <*complex.h*> **operations in OpenMP target on GPUs?**
Yes, LLVM/Clang allows math functions and complex arithmetic inside of OpenMP target regions that are compiled for GPUs.

Clang provides a set of wrapper headers that are found first when *math.h* and *complex.h*, for C, *cmath* and *complex*, for C++, or similar headers are included by the application. These wrappers will eventually include the system version of the corresponding header file after setting up a target device specific environment. The fact that the system header is included is important because they differ based on the architecture and operating system and may contain preprocessor, variable, and function definitions that need to be available in the target region regardless of the targeted device architecture. However, various functions may require specialized device versions, e.g., *sin*, and others are only available on certain devices, e.g., *__umul64hi*. To provide "native" support for math and complex on the respective architecture, Clang will wrap the "native" math functions, e.g., as provided by the device vendor, in an OpenMP begin/end declare variant. These functions will then be picked up instead of the host versions while host only variables and function definitions are still available. Complex arithmetic and functions are support through a similar mechanism. It is worth noting that this support requires *extensions to the OpenMP begin/end declare variant context selector* that are exposed through LLVM/Clang to the user as well.

Q: What is a way to debug errors from mapping memory to a target device?
An experimental way to debug these errors is to use *remote process offloading*. By using **libomptarget.rtl.rpc.so** and **openmp-offloading-server**, it is possible to explicitly perform memory transfers between processes on the host CPU and run sanitizers while doing so in order to catch these errors.

Q: Why does my application say "Named symbol not found" and abort when I run it?
This is most likely caused by trying to use OpenMP offloading with static libraries. Static libraries do not contain any device code, so when the runtime attempts to execute the target region it will not be found and you will get an an error like this.

```
CUDA error: Loading '__omp_offloading_fd02_3231c15__Z3foov_l2' Failed
CUDA error: named symbol not found
Libomptarget error: Unable to generate entries table for device id 0.
```

Currently, the only solution is to change how the application is built and avoid the use of static libraries.

Q: Can I use dynamically linked libraries with OpenMP offloading?

Dynamically linked libraries can be only used if there is no device code split between the library and application. Anything declared on the device inside the shared library will not be visible to the application when it's linked.

Q: How to build an OpenMP offload capable compiler with an outdated host compiler?

Enabling the OpenMP runtime will perform a two-stage build for you. If your host compiler is different from your system-wide compiler, you may need to set the CMake variable *GCC_INSTALL_PREFIX* so clang will be able to find the correct GCC toolchain in the second stage of the build.

For example, if your system-wide GCC installation is too old to build LLVM and you would like to use a newer GCC, set the CMake variable *GCC_INSTALL_PREFIX* to inform clang of the GCC installation you would like to use in the second stage.

Q: How can I include OpenMP offloading support in my CMake project?

Currently, there is an experimental CMake find module for OpenMP target offloading provided by LLVM. It will attempt to find OpenMP target offloading support for your compiler. The flags necessary for OpenMP target offloading will be loaded into the **OpenMPTarget::OpenMPTarget_<device>** target or the **OpenMPTarget_<device>_FLAGS** variable if successful. Currently supported devices are **AMDGPU** and **NVPTX**.

To use this module, simply add the path to CMake's current module path and call **find_package**. The module will be installed with your OpenMP installation by default. Including OpenMP offloading support in an application should now only require a few additions.

```
cmake_minimum_required(VERSION 3.13.4)
project(offloadTest VERSION 1.0 LANGUAGES CXX)

list(APPEND CMAKE_MODULE_PATH "${PATH_TO_OPENMP_INSTALL}/lib/cmake/openmp")

find_package(OpenMPTarget REQUIRED NVPTX)
```

```
add_executable(offload)
target_link_libraries(offload PRIVATE OpenMPTarget::OpenMPTarget_NVPTX)
target_sources(offload PRIVATE ${CMAKE_CURRENT_SOURCE_DIR}/src/Main.cpp)
```

Using this module requires at least CMake version 3.13.4. Supported languages are C and C++ with Fortran support planned in the future. Compiler support is best for Clang but this module should work for other compiler vendors such as IBM, GNU.

Q: What does 'Stack size for entry function cannot be statically determined' mean?

This is a warning that the Nvidia tools will sometimes emit if the offloading region is too complex. Normally, the CUDA tools attempt to statically determine how much stack memory each thread. This way when the kernel is launched each thread will have as much memory as it needs. If the control flow of the kernel is too complex, containing recursive calls or nested parallelism, this analysis can fail. If this warning is triggered it means that the kernel may run out of stack memory during execution and crash. The environment variable **LIBOMPTARGET_STACK_SIZE** can be used to increase the stack size if this occurs.

The current (in-progress) release notes can be found *here* while release notes for releases, starting with LLVM 12, will be available on *the Download Page*.

OPENMP 15.0.0 RELEASE NOTES

WARNING:

These are in-progress notes for the upcoming LLVM 15.0.0 release. Release notes for previous releases can be found on *the Download Page*.

Introduction

This document contains the release notes for the OpenMP runtime, release 15.0.0. Here we describe the status of OpenMP, including major improvements from the previous release. All OpenMP releases may be downloaded from the *LLVM releases web site*.

Non-comprehensive list of changes in this release

AUTHOR

unknown

COPYRIGHT