FOSDEM'26

HPC, Big Data
& Data Science

Using OpenMP's interop for calling GPU-vendor libs with GCC

Tobias Burnus

About Me & Baylibre

Contributor to the **OpenMP specification**

Contributor to **GCC** related to
GPU offloading, OpenMP, Fortran, …
(+ co-maintainer/reviewer)

Background in numerical/computational **physics** (time-dependent density-functional theory, then related to transition-metal oxides). *Contribution to GCC's gfortran started back then.*

Doing contract work related to
– open-source compilers, linker, debugger, …
– Linux kernel, Zephyr, Android, Yocto, …

with staff and customers on three continents

baylibre.com/blog/

tburnus@baylibre.com

GCC — OpenMP and Offloading

- Support for **OpenMP** and **OpenACC**
- Most OpenMP **5.2** features
  (but no OMPT and OMPD, yet)
  gcc.gnu.org/projects/gomp/
  gcc.gnu.org/onlinedocs/libgomp/
- Offloading to **Nvidia** and **AMD GPUs**
  gcc.gnu.org/wiki/Offloading
  For build + install infos; Linux distro ship Offload support in optional packages.
- Use **GCC 15** (released April 2025),
  gcc.gnu.org/gcc-15/
- **GCC 16** (April 2026?)
  gcc.gnu.org/gcc-16/changes.html

OpenMP **`interop`**

- Since GCC 15 [CUDA, HIP, HSA] with most OpenMP 6.0 additions (e.g. Fortran support)

WIP/near-term plan (a bit for 16, most GCC 17)

- Reduce missing bits for OpenMP 5.x + bug fixes
- Offload-performance improvements
- OMPT

And minor 6.x additions

Contributors welcome – code, documentation, bug reports
GSoC 2026 ?

Acknowledgement: This research used resources of the Oak Ridge Leadership Computing Facility, which is a DOE Office of Science User Facility supported under Contract DE-AC05-00OR22725

`interop` – Starting Point

- Existing code – possibly with OpenMP for host and/or GPU parallelization
- System with a GPU
- Hot code using FFT, (sparse)BLAS, LAPACK, random-number generation

  → To be calculated by vendor lib on the GPU

OpenMP's interop + variant functions:

- Make it easier to hide some complexity while coding
- Improve portability a bit
- Permit dependency handling of OpenMP code vs. vendor-lib code

Vendor Lib Call Example

Example: CUDA call for $Y = \alpha X + Y$ [scalar times vector + vector; BLAS' DaXpY]

```
   // Alloc device mem + copy data
 3 cudaMalloc ((void**) &d_X, N * sizeof(*X));
 4 cudaMemcpy (d_X, X, N * sizeof(*X), cudaMemcpyHostToDevice);
   …
   // Create streaming object and handle
 5 cudaStreamCreate (&stream);
 6 cublasCreate (&handle);
 7 cublasSetStream (handle, stream);

   // Y = alpha * X + Y
 8 cublasDaxpy (handle, N, &alpha, d_X, 1, d_Y, 1);

 9 cublasDestroy (handle);
10 cudaStreamDestroy (stream);

  // Copy Y - free X as no longer needed
11 cudaFree (d_X);
12 cudaMemcpy (Y, d_Y, N * sizeof(*X), cudaMemcpyDeviceToHost);
14 cudaFree (d_Y);
```

Vendor Lib Call Example

Example: CUDA call for $Y = \alpha X + Y$ [scalar times vector + vector; BLAS' DaXpY]

```
   // Alloc device mem + copy data
 3 cudaMalloc ((void**) &d_X, N * sizeof(*X));
 4 cudaMemcpy (d_X, X, N * sizeof(*X), cudaMemcpyHostToDevice);
   …
   // Create streaming object and handle
 5 cudaStreamCreate (&stream);
 6 cublasCreate (&handle);
 7 cublasSetStream (handle, stream);

   // Y = alpha * X + Y
 8 cublasDaxpy (handle, N, &alpha, d_X, 1, d_Y, 1);

 9 cublasDestroy (handle);
10 cudaStreamDestroy (stream);

  // Copy Y - free X as no longer needed
11 cudaFree (d_X);
12 cudaMemcpy (Y, d_Y, N * sizeof(*X), cudaMemcpyDeviceToHost);
14 cudaFree (d_Y);
```

First step: Use OpenMP for memory allocation, host-device transfer, and on-device initialization

→ reduce/localize vendor specific code.

Vendor Lib Call Example

OpenMP – memory handling (+ on-device var initialization)

```
// Initialize X + copy Y to the device
#pragma omp target enter data map(alloc: X[:N]) map(to:Y[:N])
#pragma omp target
  for (size_t i = 0; i < N; i++)
    X[i] = 1.0;
//  Get device ptr:
d_X = (double*) omp_get_mapped_ptr (X, omp_get_default_device());
…
cublasDaxpy (handle, N, &alpha, d_X, 1, d_Y, 1);
…

// Copy Y - free X as no longer needed
#pragma omp target exit data map(release: X) map(from: Y[:N])
```

Vendor Lib Call Example

OpenMP – memory handling (+ on-device var initialization)

```
// Initialize X + copy Y to the device
#pragma omp target enter data map(alloc: X[:N]) map(to:Y[:N])
#pragma omp target
  for (size_t i = 0; i < N; i++)
    X[i] = 1.0;
//  Get device ptr:
d_X = (double*) omp_get_mapped_ptr (X, omp_get_default_device());
…
cublasDaxpy (handle, N, &alpha, d_X, 1, d_Y, 1);
…

// Copy Y - free X as no longer needed
#pragma omp target exit data map(release: X) map(from: Y[:N])
```

Variants:

- Use unified shared memory
- `omp_target_alloc`
  use directly or associate with host variable
- OpenMP allocators
  pinned, device-accessible, managed memory, …

OpenMP→CUDA device number via `interop`

*Previous slides: assumption they are the same. Proper way + **`interop`** intro*

```
1   omp_interop_t obj;

2   #pragma omp interop init(target, prefer_type("cuda") : obj) \
                        device(omp_get_default_device())

    // Check result: got an interop object – and for CUDA:
5   if (obj != omp_interop_none
6       && omp_ifr_cuda == omp_get_interop_int (obj, omp_ipr_fr_id, nullptr))
    {
7     int dev_num = omp_get_interop_int (obj, omp_ipr_device_num, nullptr);
          // e.g. → 0
9     char* str = omp_get_interop_str (obj, omp_ipr_fr_name, nullptr));
          // → "cuda"
    }

10  #pragma omp interop destroy(obj)
```

OpenMP→CUDA device number via `interop`

*Previous code assume they are the same. Proper way + **`interop`** intro*

```
1   omp_interop_t obj;

2   #pragma omp interop init(target, prefer_type("cuda") : obj) \
                         device(omp_get_default_device())

    // Check result: got an interop object – and for CUDA:
5   if (obj != omp_interop_none
6       && omp_ifr_cuda == omp_get_interop_int (obj, omp_ipr_fr_id, nullptr))
    {
7     int dev_num = omp_get_interop_int (obj, omp_ipr_device_num, nullptr);
          // e.g. → 0
9     char* str = omp_get_interop_str (obj, omp_ipr_fr_name, nullptr));
          // → "cuda"
    }

10  #pragma omp interop destroy(obj)
```

Available *foreign runtimes:*
→ depends on device/GPU + compiler

For GCC:

- Nvidia GPUs: **cuda**, cuda_driver, hip
- AMD GPUs: **hip**, hsa

→ OpenMP.org for list + assoc data types

OpenMP→ Streaming Object

cudaStreamCreate + cudaStreamDestroy → OpenMP interop (targetsync)

```
   omp_interop_t obj;
   #pragma omp interop init(targetsync, prefer_type("cuda"): obj)
   cudaStream_t stream = (cudaStream_t)
      omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
   …
   cublasSetStream (handle, stream);
   …
13 #pragma omp interop destroy(obj)
```

OpenMP→ Streaming Object

`cudaStreamCreate` + `cudaStreamDestroy` → OpenMP `interop` (`targetsync`)

```
   omp_interop_t obj;
   #pragma omp interop init(targetsync, prefer_type("cuda"): obj)
   cudaStream_t stream = (cudaStream_t)
      omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
   …
   cublasSetStream (handle, stream);
   …
13 #pragma omp interop destroy(obj)
```

foreign-runtime-ids — data types

id	name	targetsync
1	cuda	`cudaStream_t`
2	cuda_driver	`CUstream`
3	opencl	`cl_queue`
4	sycl	`cl::sycl::queue`
5	hip	`hipStream_t`
6	level_zero	`ze_command_queue_handle_t`
7	hsa	`hsa_queue_t *`

Less library/compiler specific code – but also:

- Same device a OpenMP
- Dependency handling with asynchronous execution supported

→ *next page*

Interop with ‘depend’ and ‘nowait’ | HIP + Fortran

```
#pragma omp target nowait depend(out: x[0:N]) map(from: x[0:N])
 myVectorSet(N,  1.0, x);

#pragma omp task depend(out: y[0:N])
 myVectorSet(N, -1.0, y);

#pragma omp interop init(targetsync: obj) \
                    depend(in: x[0:N]) \
                   depend(inout: y[0:N])
...
#pragma omp interop destroy(obj) nowait \
                   depend(out: y[0:N])

#pragma omp target depend(inout: x[0:N])
 myDscal(N, scalar, x);

#pragma omp taskwait
```

```
integer(omp_interop_kind) :: obj
integer(omp_interop_fr_kind) :: fr
type(c_ptr) :: hip_sm

!$omp interop init(target, targetsync, prefer_type("hip"): obj)

fr = omp_get_interop_int (obj, omp_ipr_fr_id, res)
hip_sm  = omp_get_interop_ptr (obj, omp_ipr_targetsync, res)
...
!$omp interop destroy(obj)
```

Full example with explanations: OpenMP Examples document
www.openmp.org/specifications

Variant Functions

```
bool use_variant = false;

void variant_fn (int n, int *A) { /* … */ }

#pragma omp declare variant(variant_fn) match( user={condition(use_variant)} )
// #pragma omp declare variant(variant_fn) match( construct={parallel} )
// #pragma omp declare variant(variant_fn) match( target_device={kind(gpu)} )

void base_fn (int n, int *A) { /* … */ }

void test() {
 …
 base_fn (n, arr); // Calls base function
 use_variant = true;
 base_fn (n, arr); // Calls variant function ‘variant_fn’
}
```

Variant Function for Device Variant

Assume base function:

```
void daxpy (int n, double *da, double *dx, int incx, double *dx, int incy);
```

And device wrapper function – issues: device hard coded, device-ptr assumptions

```
void device_daxpy (int n, double *da, double *dx, int incx, double *dx, int incy) {
 cudaStreamCreate (&stream)
 cublasCreate (&handle)
 cublasSetStream (handle, stream);
 #pragma omp target data use_device_ptr(dx, dy)
   cublasDaxpy (handle, n, da, dx, incx, dy, incy);
 cublasDestroy (handle); cudaStreamDestroy (stream);
}

#pragma omp declare variant(device_daxpy) match( ... )
void daxpy (int n, double *da, double *dx, int incx, double *dx, int incy);
```

Variant Function for Device Variant

Autoconvert host→device pointer + create streaming object

```
void device_daxpy(int n, double *da, double *dx, int incx,
                  double *dx, int incy, omp_interop_t obj)
{
  cudaStream_t stream = omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
  cublasSetStream (handle, stream);
  cublasDaxpy (handle, n, da, dx, incx, dy, incy);
  cublasDestroy (handle);
}
#pragma omp declare variant(device_daxpy) adjust_args(need_device_ptr: dx, dy) \
          append_args(interop(targetsync)) match( construct={dispatch} )
void daxpy (int n, double *da, double *dx, int incx, double *dx, int incy);

void test() { /* ... */
 daxpy (N, &alpha, d_X, 1, d_Y, 1);  // Base function
 #pragma omp dispatch device(my_dev) // Specify a device to use
    // Variant function called with device ptr for dx/dy + 'targetsync' interop object
    daxpy (N, &alpha, d_X, 1, d_Y, 1);
```

Sneak preview:
Low-Overhead OpenMP

GPU kernel

Simple & automatically — vs. — explicit but faster

- CUDA + HIP (as language): Low-overhead + fast code

- OpenMP: Flexible, multiple design patterns, e.g. nested 'parallel'
  no need to specify number of teams, threads,
  stateful (internal control variables, ICV),

For hot code, faster code would be better → *OpenMP* 6.x … →

OpenMP's Kernel-Language-Like Features

CUDA/HIP

- Team-private variables: `__SHARED__`
  – for static vars
  – with *size* specified at kernel-launch time

- ```
  dim3 nblocks(N/256, N/256, N/256);
  dim3 nthreads(256, 256, 256);
  kernel<<<nblocks, nthreads>>>(...)
  ```
  — *and* —
  ```
  int x = blockIdx.x * blockDim.x +
  threadIdx.x;
  ```

- *By construction: low launch overhead*

OpenMP

- `omp` **`groupprivate`**`(x)` *[OpenMP **6.0**]*
- `omp target` **`dyn_groupprivate`**`(size)`
  + `omp_get_dyn_groupprivate_ptr()`
  *[TR14 → OpenMP **6.1**]*

- *[WIP, **6.1?**]* `omp target teams num_teams(dims(3):N/256,N/256,N/256)`
  — *and* —
  ```
  int x = omp_get_team_num_dim(1) *
  omp_get_num_teams_dim(1) + …
  ```

- *[**6.1/7?**]* optionally, disable features to reduce launch/on-device overhead
  ```

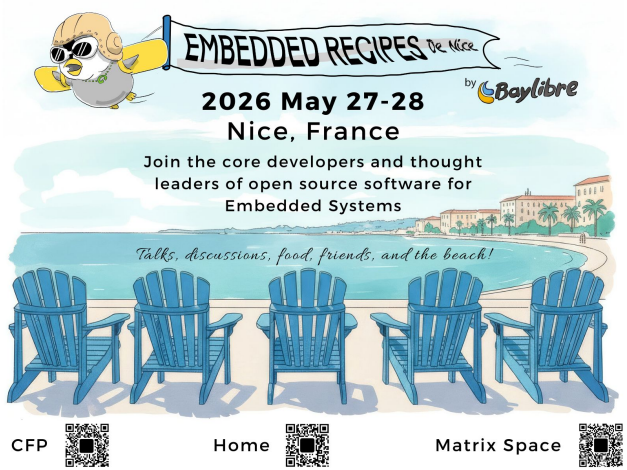
Covered Topics

- GCC's OpenMP / offload support
- `interop` – easier access to vendor libs
- OpenMP CUDA/HIP-like features ('kernel language')

More to the last two → SC25 booth/tech talk → link.openmp.org/sc25talks

Thanks for listening! — Questions, comments?

Tobias Burnus <tburnus@baylibre.com>



https://embedded-recipes.org/

Venture capital for opensource projects https://commit.fund/