OpenMP®

SC25 OpenMP Tech Talk Series

OpenMP Interop: Combining GPU Vendor Math Libraries with OpenMP

Tobias Burnus
BayLibre

About Me & BayLibre

Tobias Burnus, tburnus@baylibre.com

- Contributor to the **OpenMP specification**
- Compiler: **GCC** contributor related to GPU offloading and OpenMP
  (OpenMP & Fortran co-maintainer)
- *Background in numerical/computational physics (time-dependent density-functional theory, then related to transition-metal oxides). — Contribution to GCC's gfortran started back then.*

Baylibre baylibre.com/blog/

Service company doing work related to

- Compilers, linker, debugger, …
- Linux kernel, Zephyr, Android, Yocto, …

with staff and customers on three continents

Starting Point

Typical large programs:

- MPI plus thread parallelization (by OpenMP)
- GPU offload of compute heavy calculation
  Either *host* → *offload* → *host* — or asynchronously with host calculations

GPU vendors provide optimized GPU libraries, callable from the host

- such as for FFT, (sparse)BLAS, LAPACK, random-number generation

→ Desire to use those

Issues To Be Solved — With the Help of Interop

OpenMP to be used for better portability, device-side initialization etc. — or also for heavy lifting. Vendor libs for performance and convenience.

OpenMP features may help with:

- OpenMP & library operating on the same device
- Asynchronous calc ⟶ dependency requirements
- Code abstraction – reducing vendor dependency – host vs. device code differences

Host Call To A Vendor Library — Without OpenMP

Let's start with an example for $Y = \alpha X + Y$, i.e. scalar times vector plus vector (BLAS' DaXpY)

```
   // cuBLAS Example
   ...
   // Allocate X, Y on the host, initialize Y
   // Initialize X:
 1 for (size_t i = 0; i < N; i++)
 2   X[i] = 1.0;

   // Make it available on the device:
 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); delete[] X;
12 cudaMemcpy (Y, d_Y, N * sizeof(*X), cudaMemcpyDeviceToHost);
14 cudaFree (d_Y);
```

Full source code, use PDF slides

```
// cuBLAS Example
#include <print>  // C++23
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

int main()
{
 constexpr int N = 1024;
 constexpr double alpha = 2./3.;
 double *d_X, *d_Y; // device ptr
 cudaStream_t stream;
 cublasHandle_t handle;

 double *X = new double[N];
 double *Y = new double[N];
 for (size_t i = 0; i < N; i++)
   Y[i] = static_cast<double>( i );

 // Allocate X, Y on the host, initialize Y
 // Initialize X:
 for (size_t i = 0; i < N; i++)
   X[i] = 1.0;

 // Make it available on the device:
 cudaMalloc ((void**) &d_X, N * sizeof(*X));
 cudaMemcpy (d_X, X, N * sizeof(*X), cudaMemcpyHostToDevice);
 // ...
 cudaMalloc ((void**) &d_Y, N * sizeof(*Y));
 cudaMemcpy (d_Y, Y, N * sizeof(*Y), cudaMemcpyHostToDevice);

 // Create streaming object and handle
 cudaStreamCreate (&stream);
 cublasCreate (&handle);
 cublasSetStream (handle, stream);

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

 cublasDestroy (handle);
 cudaStreamDestroy (stream);

 // Copy Y - free X as no longer needed
 cudaFree (d_X); delete[] X;
 cudaMemcpy (Y, d_Y, N * sizeof(*X), cudaMemcpyDeviceToHost);
 cudaFree (d_Y);

 for (size_t i = 0; i < 10; i++)
   std::println ("{}: {}", i, Y[i]);
 delete[] Y;
}
```

Host Call To A Vendor Library — Without OpenMP

Let's start with an example for $Y = \alpha X + Y$, i.e. scalar times vector plus vector (BLAS' DaXpY)

```
   // cuBLAS Example
   ...
   // Allocate X, Y on the host, initialize Y
   // Initialize X:
 1 for (size_t i = 0; i < N; i++)
 2   X[i] = 1.0;

   // Make it available on the device:
 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); delete[] 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.

Full source code, use PDF slides

```
// cuBLAS Example
#include <print>  // C++23
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

int main()
{
 constexpr int N = 1024;
 constexpr double alpha = 2./3.;
 double *d_X, *d_Y; // device ptr
 cudaStream_t stream;
 cublasHandle_t handle;

 double *X = new double[N];
 double *Y = new double[N];
 for (size_t i = 0; i < N; i++)
   Y[i] = static_cast<double>( i );

 // Allocate X, Y on the host, initialize Y
 // Initialize X:
 for (size_t i = 0; i < N; i++)
   X[i] = 1.0;

 // Make it available on the device:
 cudaMalloc ((void**) &d_X, N * sizeof(*X));
 cudaMemcpy (d_X, X, N * sizeof(*X), cudaMemcpyHostToDevice);
 // ...
 cudaMalloc ((void**) &d_Y, N * sizeof(*Y));
 cudaMemcpy (d_Y, Y, N * sizeof(*Y), cudaMemcpyHostToDevice);

 // Create streaming object and handle
 cudaStreamCreate (&stream);
 cublasCreate (&handle);
 cublasSetStream (handle, stream);

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

 cublasDestroy (handle);
 cudaStreamDestroy (stream);

 // Copy Y - free X as no longer needed
 cudaFree (d_X); delete[] X;
 cudaMemcpy (Y, d_Y, N * sizeof(*X), cudaMemcpyDeviceToHost);
 cudaFree (d_Y);

 for (size_t i = 0; i < 10; i++)
   std::println ("{}: {}", i, Y[i]);
 delete[] Y;
}
```

Use OpenMP For Initialization And Memory Handling

Let's start with an example for $Y = \alpha X + Y$, i.e. scalar times vector plus vector (BLAS' DaXpY)

```
   // cuBLAS Example
   // Initialize X + copy to the device
 1 #pragma omp target enter data map(alloc: X[:N]) map(to:Y[:N])
 2 #pragma omp target
 3   for (size_t i = 0; i < N; i++)
 4     X[i] = 1.0;
   //  Get device ptr:
 5 d_X = (double*) omp_get_mapped_ptr (X,
                                       omp_get_default_device());
   ...
   // Create streaming object and handle
 6 cudaStreamCreate (&stream);
 7 cublasCreate (&handle);
 8 cublasSetStream (handle, stream);

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

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

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

```
// cuBLAS Example
#include <print>  // C++23
#include <omp.h>
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

int main()
{
 constexpr int N = 1024;
 constexpr double alpha = 2./3.;
 double *d_X, *d_Y; // device ptr
 cudaStream_t stream;
 cublasHandle_t handle;

 double *X = new double[N];
 double *Y = new double[N];
 for (size_t i = 0; i < N; i++)
   Y[i] = static_cast<double>( i );

 // Allocate X, Y on the host, initialize Y
 // Initialize X:
 #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());
 d_Y = (double*) omp_get_mapped_ptr (Y, omp_get_default_device());

 // Create streaming object and handle
 cudaStreamCreate (&stream);
 cublasCreate (&handle);
 cublasSetStream (handle, stream);

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

 cublasDestroy (handle);
 cudaStreamDestroy (stream);

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

 for (size_t i = 0; i < 10; i++)
   std::println ("{}: {}", i, Y[i]);
 delete[] Y;
}
```

Use OpenMP For Initialization And Memory Handling

Let's start with an example for $Y = \alpha X + Y$, i.e. scalar times vector plus vector (BLAS' DaXpY)

```
   // cuBLAS Example
   // Initialize X + copy to the device
 1 #pragma omp target enter data map(alloc: X[:N]) map(to:Y[:N])
 2 #pragma omp target
 3   for (size_t i = 0; i < N; i++)
 4     X[i] = 1.0;
   //  Get device ptr:
 5 d_X = (double*) omp_get_mapped_ptr (X,
                                       omp_get_default_device());
   ...
   // Create streaming object and handle
 6 cudaStreamCreate (&stream);
 7 cublasCreate (&handle);
 8 cublasSetStream (handle, stream);

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

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

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

OpenMP memory allocation + data handling

Variants:

- Unified shared memory (or self_maps) `#pragma omp requires self_maps`
- `omp_target_alloc`
- Associating, e.g. `cuda…` API allocated host/device memory with OpenMP → `omp_target_associate_ptr`
- OpenMP allocators – e.g. user defined (pinned, device accessible, …) or predefined ones, e.g. GCC's `ompx_gnu_managed_mem_{alloc,space}`

```
// cuBLAS Example
#include <print>  // C++23
#include <omp.h>
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

int main()
{
 constexpr int N = 1024;
 constexpr double alpha = 2./3.;
 double *d_X, *d_Y; // device ptr
 cudaStream_t stream;
 cublasHandle_t handle;

 double *X = new double[N];
 double *Y = new double[N];
 for (size_t i = 0; i < N; i++)
   Y[i] = static_cast<double>( i );

 // Allocate X, Y on the host, initialize Y
 // Initialize X:
 #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());
 d_Y = (double*) omp_get_mapped_ptr (Y, omp_get_default_device());

 // Create streaming object and handle
 cudaStreamCreate (&stream);
 cublasCreate (&handle);
 cublasSetStream (handle, stream);

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

 cublasDestroy (handle);
 cudaStreamDestroy (stream);

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

 for (size_t i = 0; i < 10; i++)
   std::println ("{}: {}", i, Y[i]);
 delete[] Y;
}
```

Interop Directive — Target

Previous page: Code assumed that OpenMP's default device and CUDA's current device is the same.
Let's fix this – **OpenMP's 'interop' directive** can be obtain this

Interop Directive — Target

Previous page: Code assumed that OpenMP's default device and CUDA's current device is the same.

Let's fix this – **OpenMP's 'interop' directive** can be obtain this

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

 1 int main() {
 2   omp_interop_t obj;

 3   #pragma omp interop init(target : obj) \
                  device(omp_get_default_device())

     /* Uses default type, manually, e.g.
 4   #pragma omp interop \
       init(target, prefer_type("cuda", "cuda_driver") : obj)  */

     // If successful, we should have got the following:
 5   if (obj != omp_interop_none /* N/A, e.g. host.  */
 6       && omp_ifr_cuda == omp_get_interop_int (obj,
                                omp_ipr_fr_id, nullptr))
     {
 7     std::println("CUDA device no {}: Runtime '{}'",
 8       omp_get_interop_int (obj, omp_ipr_device_num, nullptr),
 9       omp_get_interop_str (obj, omp_ipr_fr_name, nullptr));
       // Prints, e.g.: CUDA device no 0: Runtime 'cuda'
     }

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

Interop Directive — Target

Previous page: Code assumed that OpenMP's default device and CUDA's current device is the same.

Let's fix this – **OpenMP's 'interop' directive** can be obtain this

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

int main() {
  omp_interop_t obj;

  #pragma omp interop init(target : obj) \
                  device(omp_get_default_device())

  /* Uses default type, manually, e.g.
  #pragma omp interop \
    init(target, prefer_type("cuda", "cuda_driver") : obj)  */

  // If successful, we should have got the following:
  if (obj != omp_interop_none /* N/A, e.g. host.  */
      && omp_ifr_cuda == omp_get_interop_int (obj,
                                 omp_ipr_fr_id, nullptr))
  {
    std::println("CUDA device no {}: Runtime '{}'",
      omp_get_interop_int (obj, omp_ipr_device_num, nullptr),
      omp_get_interop_str (obj, omp_ipr_fr_name, nullptr));
    // Prints, e.g.: CUDA device no 0: Runtime 'cuda'
  }

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

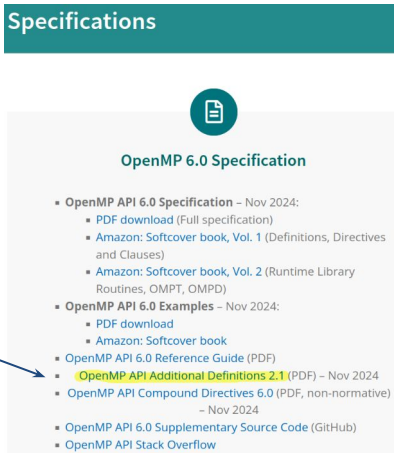
Availability of foreign runtime depends on the compiler & the GPU hardware.

List of foreign runtimes and data types:
→ cuda, cuda_driver, opencl, sycl, hip, level_zero, hsa
Defined in the "Additional Definitions for the OpenMP API Specification" document, www.openmp.org/specifications

For GCC:

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

gcc.gnu.org/onlinedocs/libgomp/Offload-Target-Specifics.html



Interop Directive — Targetsync

Back to the first example — Replace the ‘cudaCreateStream’ + fix device-num issue:

Interop Directive — Targetsync

Back to the first example — Replace the ‘cudaCreateStream’ + fix device-num issue:

```
1   #pragma omp target enter data map(alloc: X[:N]) map(to:Y[:N])
2   #pragma omp target
3   for (size_t i = 0; i < N; i++)
4     X[i] = 1.0;

5   d_X = (double*) omp_get_mapped_ptr (X,
                                        omp_get_default_device());
    ...
    // Get stream obj (sorry, no error handling)
6   omp_interop_t obj;
7   #pragma omp interop init(targetsync, prefer_type("cuda"): obj)
8   cudaStream_t stream = (cudaStream_t)
       omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
9   cublasCreate (&handle);
10  cublasSetStream (handle, stream);

11  cublasDaxpy (handle, N, &alpha, d_X, 1, d_Y, 1);

12  cublasDestroy (handle);
13  #pragma omp interop destroy(obj)

14  #pragma omp target exit data map(release: X) map(from: Y[:N])
```

```
#include <print>  // C++23
#include <omp.h>
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

int main()
{
 constexpr int N = 1024;
 constexpr double alpha = 2./3.;
 double *d_X, *d_Y; // device ptr
 cublasHandle_t handle;

 double *X = new double[N];
 double *Y = new double[N];
 for (size_t i = 0; i < N; i++)
   Y[i] = static_cast<double>( i );

 // Allocate X, Y on the host
 // Initialize Y on the host - copy X to device and init there
 #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());
 d_Y = (double*) omp_get_mapped_ptr (Y,
omp_get_default_device());

 // Get stream obj (sorry, no error handling)
 omp_interop_t obj;
 #pragma omp interop \
   init(targetsync, prefer_type("cuda", "cuda_driver") : obj)
 cudaStream_t stream = (cudaStream_t) omp_get_interop_ptr (
                                      obj, omp_ipr_targetsync, nullptr);
 cublasCreate (&handle);
 cublasSetStream (handle, stream);

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

 cublasDestroy (handle);
 #pragma omp interop destroy(obj)

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

 for (size_t i = 0; i < 10; i++)
   std::println ("{}: {}", i, Y[i]);
 delete[] Y;
}
```

Interop Directive — Targetsync

Back to the first example — Replace the ‘cudaCreateStream’ + fix device-num issue:

```
1   #pragma omp target enter data map(alloc: X[:N]) map(to:Y[:N])
2   #pragma omp target
3   for (size_t i = 0; i < N; i++)
4     X[i] = 1.0;

5   d_X = (double*) omp_get_mapped_ptr (X,
                                        omp_get_default_device());
    ...
    // Get stream obj (sorry, no error handling)
6   omp_interop_t obj;
7   #pragma omp interop init(targetsync, prefer_type("cuda"): obj)
8   cudaStream_t stream = (cudaStream_t)
       omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
9   cublasCreate (&handle);
10  cublasSetStream (handle, stream);

11  cublasDaxpy (handle, N, &alpha, d_X, 1, d_Y, 1);

12  cublasDestroy (handle);
13  #pragma omp interop destroy(obj)

14  #pragma omp target exit data map(release: X) map(from: Y[:N])
```

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 *

```
#include <print>  // C++23
#include <omp.h>
#include <cuda_runtime_api.h>
#include <cublas_v2.h>

int main()
{
 constexpr int N = 1024;
 constexpr double alpha = 2./3.;
 double *d_X, *d_Y; // device ptr
 cublasHandle_t handle;

 double *X = new double[N];
 double *Y = new double[N];
 for (size_t i = 0; i < N; i++)
   Y[i] = static_cast<double>( i );

 // Allocate X, Y on the host
 // Initialize Y on the host - copy X to device and init there
 #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());
 d_Y = (double*) omp_get_mapped_ptr (Y,
omp_get_default_device());

 // Get stream obj (sorry, no error handling)
 omp_interop_t obj;
 #pragma omp interop \
   init(targetsync, prefer_type("cuda", "cuda_driver") : obj)
 cudaStream_t stream = (cudaStream_t) omp_get_interop_ptr (
                                  obj, omp_ipr_targetsync, nullptr);
 cublasCreate (&handle);
 cublasSetStream (handle, stream);

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

 cublasDestroy (handle);
 #pragma omp interop destroy(obj)

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

 for (size_t i = 0; i < 10; i++)
   std::println ("{}: {}", i, Y[i]);
 delete[] Y;
}
```

Going Asynchronous …

Concurrency on host and device and in-background operations

→ Faster, but dependency has to be taken into account → OpenMP's 'depend' and 'nowait' clauses

```
#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
```

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

Interop Directive — Fortran & HIP Example

Interop with Fortran (since OpenMP 6) – Example using AMD's HIP (for AMD and Nvidia GPUs)

```
1   use iso_c_binding, only: c_ptr, c_int
2   use omp_lib
3   use hipfort

5   integer(omp_interop_rc_kind) :: res
6   integer(omp_interop_kind) :: obj
7   integer(omp_interop_fr_kind) :: fr
8   integer(c_int) :: hip_dev
9   type(c_ptr) :: hip_ctx, hip_sm

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

13  fr = omp_get_interop_int (obj, omp_ipr_fr_id, res)
14  if (res /= omp_irc_success) error stop 1
15  if (fr /= omp_ifr_hip) error stop 1

17  hip_dev = omp_get_interop_int (obj, omp_ipr_device, res)
18  hip_ctx = omp_get_interop_ptr (obj, omp_ipr_device_context, res)
19  hip_sm  = omp_get_interop_ptr (obj, omp_ipr_targetsync, res)
    ...
21  !$omp interop destroy(obj)
```

Example taken from GCC's testsuite, `libgomp/testsuite/libgomp.fortran/interop-hip-{amd,nvidia}*.F90`

Requires **hipfort** module from, https://github.com/ROCm/hipfort (compiled for Nvidia [CUDA] or AMD GPU)

HIP & C/C++: `#include <hip/hip_runtime_api.h>` requires that __HIP_PLATFORM_AMD__ or __HIP_PLATFORM_NVIDIA__ is defined before including it *(albeit defaults to a value if the AMD or Nvidia compiler is used)*.

Compiler Support

‘interop’ in the **OpenMP Specification**

- Added in OpenMP 5.1
- 5.2 tiny changes in additions
- 6.0: ‘hsa’, attributes with ‘prefer_type’, Fortran API routines, ‘dispatch’ extensions (later)

Compiler support

- **GCC 15** added most (6.0’s adjust_args changes partially, schedl. for GCC 16)
- Intel Compiler
- HPE’s CCE → https://cpe.ext.hpe.com/docs/latest/cce/man7/intro_openmp.7.html
- ?

Compiler Support

'interop' in the **OpenMP Specification**

- Added in OpenMP 5.1
- 5.2 tiny changes in additions
- 6.0: 'hsa', attributes with 'prefer_type', Fortran API routines, 'dispatch' extensions (later)

Compiler support

- **GCC 15** added most (6.0's adjust_args changes partially, schedl. for GCC 16)
- Intel Compiler
- HPE's CCE → https://cpe.ext.hpe.com/docs/latest/cce/man7/intro_openmp.7.html
- ?

Intel example for OpenMP Interop with SYCL

```
// Create an interop object with SYCL queue access.
#pragma omp interop init(prefer_type(omp_ifr_sycl), targetsync : obj)
  if (omp_ifr_sycl != omp_get_interop_int(obj, omp_ipr_fr_id, nullptr)){
    fprintf(stderr,
            "ERROR: Failed to create interop with SYCL queue access\n");
    exit(1);
  }

  // Access SYCL queue returned by OpenMP interop.
  auto *q = static_cast<sycl::queue *>(
      omp_get_interop_ptr(obj, omp_ipr_targetsync, nullptr));
```

https://www.intel.com/content/www/us/en/docs/oneapi/optimization-guide-gpu/2025-2/openmp-interop-with-sycl.html

Take Away — Part 1

- Memory management in OpenMP largely
  avoids need of vendor/device-specific code

- Interop permits dependency handling
  and reduces device-dependent code

Useful way to mix fast GPU-vendor device libraries with OpenMP

Make Library Calls Easier: Declare Variant

Part 2

Declare Variant

OpenMP's 'declare variant' permits to call function variants under certain conditions (context). Example:

```
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() {
 int arr[] = {1,2,3,4,5};
 constexpr int n = 5;

 base_fn (n, arr); // Calls base function
 use_variant = true;
 base_fn (n, arr); // Calls variant function 'variant_fn'
}
```

Use Declare Variant for Host And GPU Library Version of “daxpy”

Let’s use declare variant for daxpy

```
cublasStatus_t cublasDaxpy (cublasHandle_t, int, const double*, const double*, int, double*, int);

// Assume this C function (wrapper) around NetLib's BLAS subroutine:
void daxpy (int n, double *da, double *dx, int incx, double *dx, int incy);
```

ISSUE: Calling the vendor functions has a different ABI – and requires a handle and device pointer

Use Declare Variant for Host And GPU Library Version of “daxpy”

Let’s use declare variant for daxpy

```
cublasStatus_t cublasDaxpy (cublasHandle_t, int, const double*, const double*, int, double*, int);

// Assume this C function (wrapper) around NetLib's BLAS subroutine:
void daxpy (int n, double *da, double *dx, int incx, double *dx, int incy);
```

ISSUE: Calling the vendor functions has a different ABI – and requires a handle and device pointer → wrapper

```
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);
```

Avoid Hard-Coding ‘use_device_ptr’

Issue 1: Requires device pointer – hardcoded in the call

```
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);

void test() { /* ... */
 daxpy (N, &alpha, d_X, 1, d_Y, 1);  // Calls NetLib version
 // some context that fulfills the condition
    daxpy (N, &alpha, d_X, 1, d_Y, 1);  // Calls device_daxpy
```

need_device_ptr & dispatch

Issue 1: Requires device pointer – hardcoded in the call

```
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( construct={dispatch} )  adjust_args(need_device_ptr: dx, dy)
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);  // Calls NetLib version
 #pragma omp dispatch
    daxpy (N, &alpha, d_X, 1, d_Y, 1);  // Calls device_daxpy
```

Note: adjust_args requires 'dispatch'.
Reduces chance of surprises if dx and dy do not exist on the device

Dispatch Clauses

Dispatch Examples

```
#pragma omp declare variant(device_daxpy) match( construct={dispatch} )  adjust_args(need_device_ptr: dx, dy)
...
void test() { /* ... */
 #pragma omp dispatch
    daxpy (N, &alpha, d_X, 1, d_Y, 1);  // Convert dx + dy to device pointer

 #pragma omp dispatch is_device_ptr(dev_X, dev_Y)
    daxpy (N, &alpha, dev_X, 1, dev_Y, 1);  // No need to convert

// No context = 'construct(dispatch)' not matched
// * n <  256 → call daxpy
// * n >= 256 → call device_daxpy
 #pragma omp dispatch nocontext(n < 256)
    daxpy (N, &alpha, dev_X, 1, dev_Y, 1);
```

- For C++ and esp. Fortran also '**need_device_addr**' and '**has_device_addr**'
  (OpenMP 6.0, already implemented in multiple compilers)
- '**novariants**(*cond*)' clause disables all variant replacements, not only those using 'construct'
- '**depend**' and '**nowait**' are also supported

Fix Device-Number Dependency & Stream Creation

Issue 2: Used device hardcode via the streaming object → vendor version used

```
void device_daxpy(int n, double *da, double *dx, int incx,
                  double *dx, int incy)
{
  cudaStreamCreate (&stream); // Use CUDA's current device
    cublasCreate (&handle);
    cublasSetStream (handle, stream);
      cublasDaxpy (handle, n, da, dx, incx, dy, incy);
    cublasDestroy (handle);
  cudaStreamDestroy (stream);
}
```

```
#pragma omp declare variant(device_daxpy) match( construct={dispatch} ) adjust_args(need_device_ptr: dx, dy)
```

Fix Device-Number Dependency & Stream Creation

Issue 2: Used device hardcode via the streaming object → interop used

```
void device_daxpy(int n, double *da, double *dx, int incx,
                  double *dx, int incy)
{
  cudaStreamCreate (&stream); // Use CUDA's current device
    cublasCreate (&handle);
    cublasSetStream (handle, stream);
      cublasDaxpy (handle, n, da, dx, incx, dy, incy);
    cublasDestroy (handle);
  cudaStreamDestroy (stream);
}
```

```
void device_daxpy(int n, double *da, double *dx, int incx,
                  double *dx, int incy)
{
// Interop variant - Use OpenmP's default device
omp_interop_t obj;
#pragma omp interop init(targetsync, prefer_type("cuda"): 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 interop destory
}
```

```
#pragma omp declare variant(device_daxpy) match( construct={dispatch} ) adjust_args(need_device_ptr: dx, dy)
```

Use Declare Variant to Create Steaming Object

Issue 2: Used device hardcode via the streaming object → Use **append_args**

```
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) match( construct={dispatch} ) adjust_args(need_device_ptr: dx, dy) \
            append_args(interop(targetsync))

void test() { /* ... */
 #pragma omp dispatch // Automatically obtain the interop obj
    daxpy (N, &alpha, d_X, 1, d_Y, 1);
 #pragma omp dispatch device(my_dev) // Specify a device to use
    daxpy (N, &alpha, d_X, 1, d_Y, 1);
 #pragma omp dispatch interop(my_interop_obj) // Use passed device object (OpenMP 6.0)
    daxpy (N, &alpha, d_X, 1, d_Y, 1);
```

Summary

OpenMP's `interop` & `declare variant` make it easier to combine OpenMP with GPU-Vendor Runtime Library

```
// Closing Example
void cuda_daxpy(…, omp_interop_t obj) {
  cudaStream_t stream = omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
}
void hip_daxpy(…, omp_interop_t obj) {
  hipStream_t stream = omp_get_interop_ptr (obj, omp_ipr_targetsync, nullptr);
}
#pragma omp declare variant(cuda_daxpy) match( construct={dispatch}, target_device={kind(nohost), arch("nvptx")}) \
            adjust_args(need_device_ptr: dx, dy) append_args(interop(targetsync, prefer_type("cuda", "cuda_driver") )
#pragma omp declare variant(hip_daxpy) match( construct={dispatch}, target_device={kind(nohost), arch("amdgcn")}) \
            adjust_args(need_device_ptr: dx, dy) append_args(interop(targetsync, prefer_type({ fr("hip")}, {fr("hsa")} ) ) // OpenMP 6 syntax
#pragma omp declare variant(sycl_daxpy) match( construct={dispatch}, target_device={kind(nohost), …}) \
            adjust_args(need_device_ptr: dx, dy) append_args(interop(targetsync, prefer_type({fr("sycl"), attr("ompx_inorder”} )

void test() { /* ... */
 #pragma omp dispatch device(my_dev) is_device_ptr(d_X) nocontext(n < 256)
    daxpy (N, &alpha, d_X, 1, d_Y, 1);
```

SYCL: execute tasks enqueued in either "in-order" or "out-of-order" (default); with in order, barriers can be avoided.
'attr' syntax supported by GCC 15, but currently not used. — Intel's compiler might (soon? already?) support some 'ompx…' for SYCL's in-order execution.

SC25 OpenMP Tech Talk Series

openmp.org — OpenMP API specs, forum, reference guides, and more

link.openmp.org/sc25talks — SC25 OpenMP Tech Talk videos and presentations