MODULE EIGHT: INTEROPERABILITY

Speaker, Date

OpenACC

More Science, Less Programming

MODULE OVERVIEW

Topics to be covered

- Types of interoperability
- Share data between OpenACC and CUDA
- Add CUDA or accelerated libraries to an OpenACC application
- Add OpenACC to an existing accelerated application

INTEROPERABILITY

MODULE BACKGROUND

- This module concentrates primarily on non-shared memory systems
- Non-shared memory systems have host and device copies of data objects
- Shared memory systems have a single data object for both the host and the device
- It is possible to have hybrid memory systems with some shared memory and some non-shared memory – programming these systems is beyond the scope of this module

TYPES OF INTEROPERABILITY

- OpenACC can be self contained but it has to live in a complex world
- OpenACC can be used as the primary programming model and augmented by device-specific kernels, i.e. CUDA kernels and tuned math libraries.
- OpenACC can be used as a way to quickly add functionality to a CUDA program by working directly on CUDA data within OpenACC compute regions
- OpenACC can call CUDA device functions
- CUDA kernels can call OpenACC device functions

THE KEY TO INTEROPERABILITY

The data

Data sharing

- Using data on the device between programming models

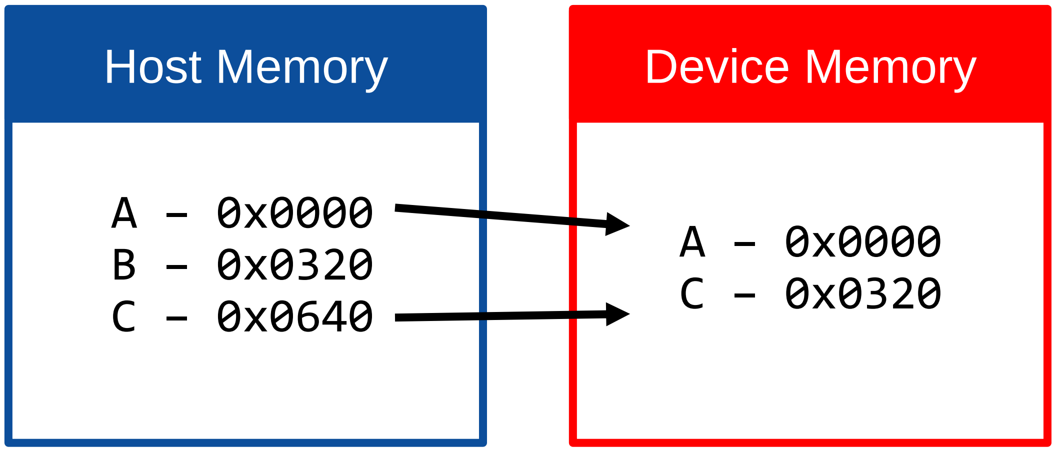
Kernel sharing

- Calling kernels on data allocated by another programming model

SHARING DATA WITH CUDA

SHARING DATA WITH CUDA

- OpenACC does not require the program to maintain different pointers for Host and Device memory
- When data is allocated on the device, a mapping between the host address and device address is created

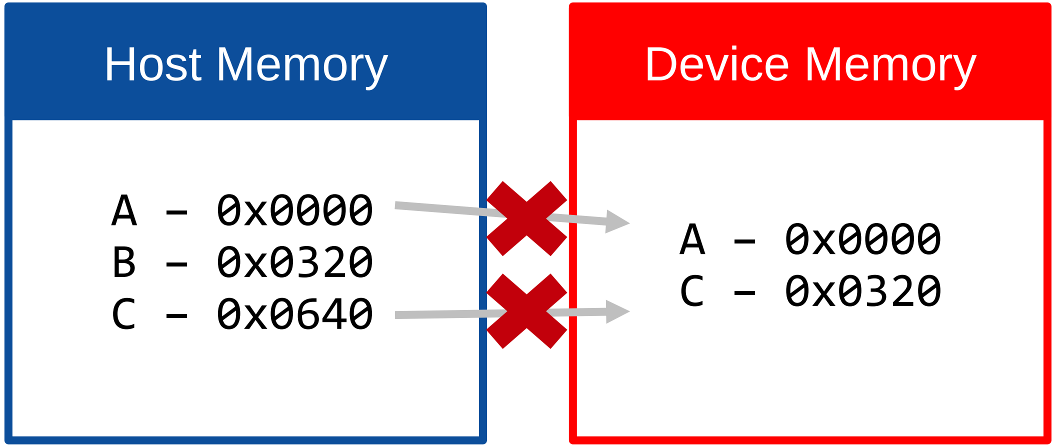


- Which memory pool is used is dependent on the context in which the pointer is referenced
- If you reference the pointer in host code, the host address will be used
- If you reference the pointer in device code (e.g. a parallel loop), the device address will be used

SHARING DATA WITH CUDA

- When using CUDA, this mapping does not exist, and you must explicitly use device addresses

- Meaning that if you want to launch a CUDA kernel from within our OpenACC code using data that we allocated with OpenACC, we must ensure that we are giving CUDA the device addresses, and not the host addresses



- The **host_data** directive is used to expose the OpenACC device address.

- The **host_data** directive specifies that the **device pointer** should be used instead of the **host pointer**

SHARING DATA WITH CUDA

Using the HOST_DATA directive

```
int x[100], y[100];

#pragma acc data copy(x,y)
{
// x and y are host pointers


#pragma acc host_data use_device(x,y)
{
 // x and y are device pointers
}
// x and y are host pointers



}
```

Allocate some data in host memory

We can use the host_data directive to expose the mapped device address

the host address

OpenACC

EXAMPLE CUDA CODE: SAXPY

- To demonstrate a use for the host_data directive, we will use it to launch this sample CUDA kernel
- In order for OpenACC + CUDA interoperability to work, we need to pass device pointers to the CUDA kernel

```
__global__
void saxpy_kernel(int n, float a,
      float *x, float *y)
{
  int i = blockIdx.x*blockDim.x + threadIdx.x;
  if (i < n) y[i] = a*x[i] + y[i];
}

void saxpy(int n, float a,
      float *dx, float *dy)
{
  // Launch CUDA Kernel
  saxpy_kernel<<<4096,256>>>(n, a, dx, dy);
}
```

HOST_DATA EXAMPLE

C (main program)

```
extern void saxpy(int n, float a,
                  float *dx, float *dy);
void main() {
   integer n = 2<<20;
   float x[n], y[n];
   float a = 2.0;
   for ( int i = 0; i < N; i++ ) {
      x[i] = 1.0;
      y[i] = 0.0;
#pragma acc data copy(y[0:n]) copyin(x[0:n])
{
...
  #pragma acc host_data use_device(x,y)
  {
     saxpy(n, a, x, y);
  }
}
}
```

- #pragma acc data copy(y) copyin(x) create copies of data on the device
- Calling the saxpy function will result in an error, because it will use host addresses instead of device addresses
- #pragma acc host_data use_device(x,y) – tells the runtime to send the device addresses of x and y to the saxpy function
- Any other call added within the OpenACC **host_data** region will recieve device addresses as well

CUBLAS LIBRARY & OPENACC

We can also use this strategy to interface with existing GPU-optimized libraries (from C/C++ or Fortran).

This includes…

- CUBLAS
- Libsci_acc
- CUFFT
- MAGMA
- CULA
- Thrust (more on this later!)

OpenACC Main Calling CUBLAS

```
int N = 1<<20;
float *x, *y
// Allocate & Initialize X & Y
...
cublasInit();
#pragma acc data copyin(x[0:N]) copy(y[0:N])
{
  #pragma acc host_data use_device(x,y)
  {
    // Perform SAXPY on 1M elements
    cublasSaxpy(N, 2.0, x, 1, y, 1);
  }
}
cublasShutdown();
```

OpenACC

SHARING CUDA DATA

SHARING CUDA DATA

Using CUDA addresses in OpenACC compute constructs

- Now lets look at the reverse situation…
- We can use CUDA-allocated device memory in our OpenACC code
- In order to use CUDA addresses, we must mark them as special addresses

CUDA address markup takes two forms:

- **Telling the OpenACC to use a *device pointer***
- **Associating a device pointer with an object in the OpenACC runtime**

EXAMPLE OPENACC CODE: SAXPY

OpenACC Version

```
void saxpy(int n, float a,
    float *x, float *y)
{

#pragma acc parallel loop default(present)
    for(int i = 0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }

}
```

- This time we will use an OpenACC version of the saxpy function
- Normally, we would give this function host pointers, which the OpenACC runtime would translate to the mapped device pointers
- But, when using CUDA device memory, only device pointers are available
- We will use the **deviceptr** clause to tell the OpenACC runtime that X and Y are already device pointers, and no translation is neccessary

DEVICEPTR CLAUSE

The **deviceptr** clause informs the compiler that an object is already on the device, so no translation is necessary.

- deviceptr can be used in either the parallel, kernels, or data directive

```
cudaMallocManaged((void*)&x,(size_t)n*sizeof(float));
cudaMallocManaged((void*)&y,(size_t)n*sizeof(float));

void saxpy(int n, float a, float *x, float *y) {
 #pragma acc parallel loop deviceptr(x,y)
    for(int i = 0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }
}
```

Do not translate x and y, they are already device pointers!

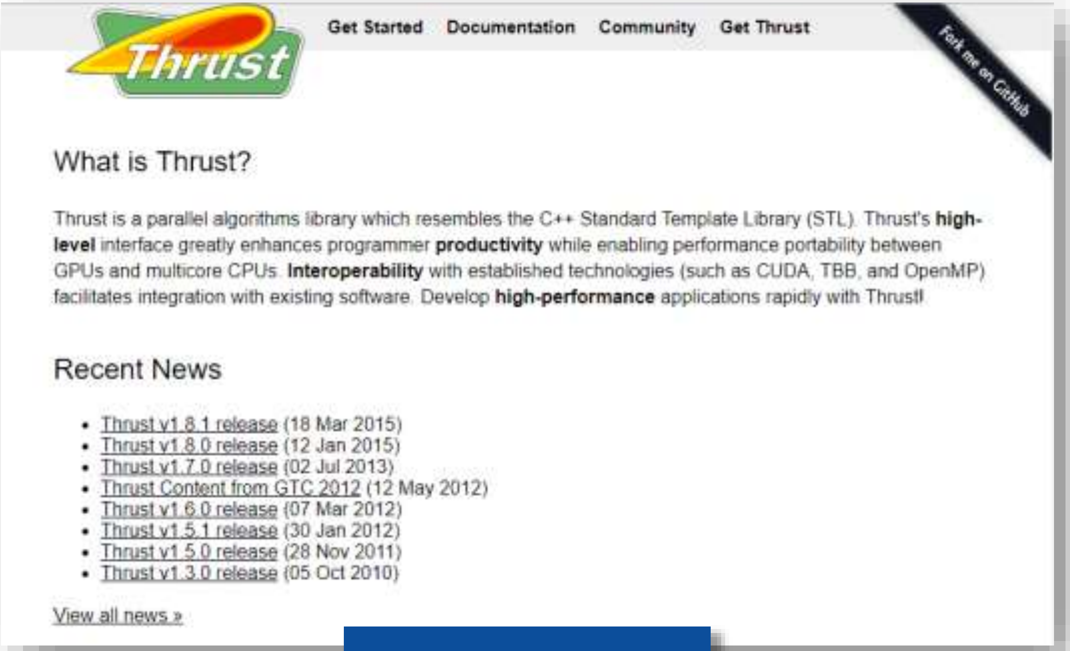
OPENACC & THRUST

Thrust is a STL-like library for C++ on accelerators.

- High-level interface
- Host/Device container classes
- Common parallel algorithms

It's possible to cast Thrust vectors to device pointers for use with OpenACC

```
void saxpy(int n, float a, float *x, float *y)
{
#pragma acc parallel loop deviceptr(x,y)
    for(int i = 0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }

}
```



thrust.github.io

OPENACC & THRUST

Thrust is a STL-like library for C++ on accelerators.

- High-level interface
- Host/Device container classes
- Common parallel algorithms

It's possible to cast Thrust vectors to device pointers for use with OpenACC

```
void saxpy(int n, float a, float *x, float *y)
{
#pragma acc parallel loop deviceptr(x,y)
    for(int i = 0; i < n; i++) {
        y[i] = a*x[i] + y[i];
    }

}
```

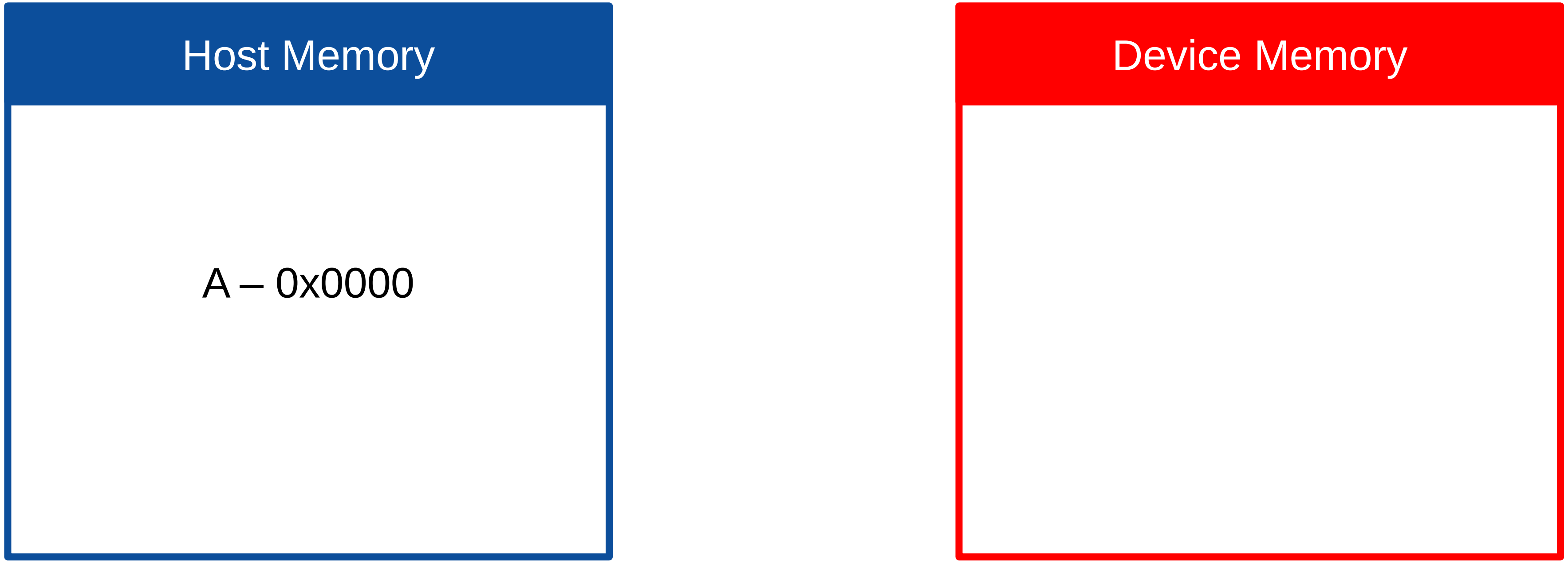
```
int N = 1<<20;
thrust::host_vector<float> x(N), y(N);
for(int i=0; i<N; i++)
{
  x[i] = 1.0f;
  y[i] = 0.0f;
}

// Copy to Device
thrust::device_vector<float> d_x = x;
thrust::device_vector<float> d_y = y;

saxpy(N,2.0, d_x.data().get(),
             d_y.data().get());

// Copy back to host
y = d_y;
```

MAPPING DEVICE POINTERS

- As mentioned previously, OpenACC creates a mapping between host and device memory
- CUDA, on the other hand, does not
- To further connect OpenACC and CUDA, we can allocate device data with CUDA, and then use OpenACC to manually map it to host pointers
- This allows the programmer to take a more hands-on approach to memory mapping

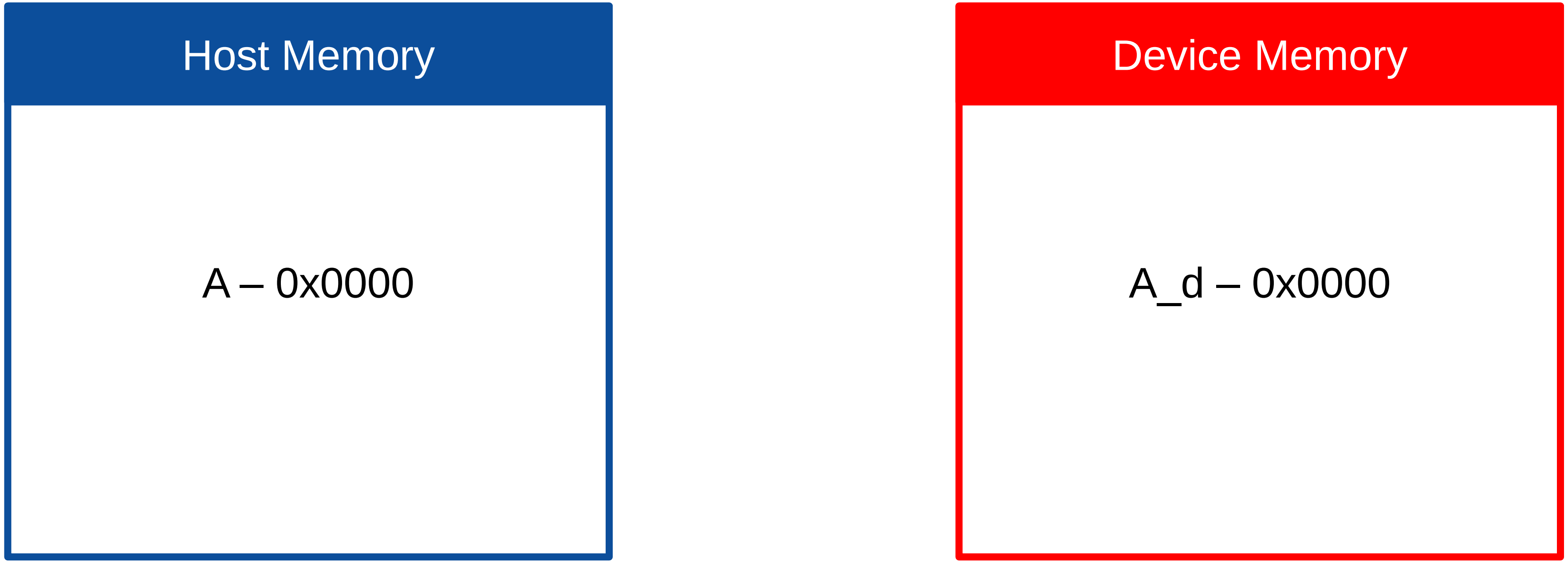
OPENACC ACC_MAP_DATA FUNCTION

```
float A[100];
```



OPENACC ACC_MAP_DATA FUNCTION

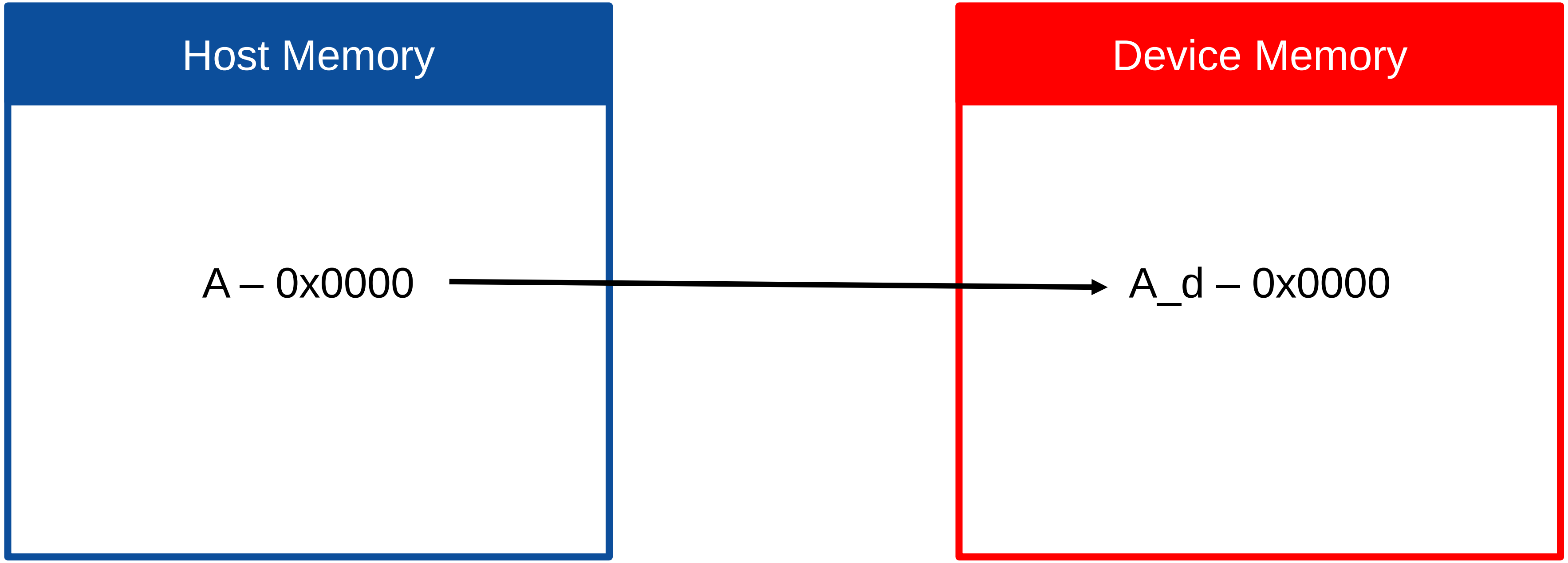
```
float A[100];
cudaMalloc((void*)&A_d,(size_t)100*sizeof(float));
```



OpenACC

OPENACC ACC_MAP_DATA FUNCTION

```
float A[100];
cudaMalloc((void*)&A_d,(size_t)100*sizeof(float));
acc_map_data(x, x_d, 100*sizeof(float));
```



OPENACC ACC_MAP_DATA FUNCTION

- The **acc_map_data** (**acc_unmap_data**) maps (unmaps) an existing device allocation to an OpenACC variable
- To map the data, both pointers must be valid
- Additionally, you must unmap the data before deallocating the memory

```
float x[n], y[n];

cudaMalloc((void*)&x_d,(size_t)n*sizeof(float));
cudaMalloc((void*)&y_d,(size_t)n*sizeof(float));

acc_map_data(x, x_d, n*sizeof(float));
acc_map_data(y, y_d, n*sizeof(float));

#pragma acc parallel loop
for(int i = 0; i < n; i++) {
    x[i] = x[i] * y[i];
}
```

Because of the mapping, this parallel loop will automatically translate x and y to their respective device addresses

OpenACC

USING DEVICE ROUTINES

CUDA DEVICE ROUTINES AND OPENACC

CUDA Code

```
extern "C" __device__ void
f1dev(float* a, float* b, int i)
{
  a[i] = .... b[i] .... ;
}
```

Even CUDA **__device__** functions can be called from OpenACC if declared with **acc routine**.

OpenACC Code

```
#pragma acc routine seq
extern "C" void f1dev( float*, float* int );
...
#pragma acc parallel loop \
  present( a[0:n], b[0:n] )
for( int i = 0; i < n; ++i )
{
  f1dev( a, b, i );
}
```

WRAP UP

KEY CONCEPTS

In this module we discussed…

- OpenACC data used in CUDA kernels
- CUDA data used in OpenACC regions
- Calling device functions from inside of OpenACC compute regions.

FURTHER EXAMPLES

- If you would like some more full code examples of OpenACC interoperability, follow the github link below to view a repository that contains many of the codes discussed in this module.
- If you would like to read some additional information about the concepts covered today, follow the second link to an NVIDIA devblog about OpenACC interoperability.

https://github.com/jefflarkin/openacc-interoperability
https://devblogs.nvidia.com/3-versatile-openacc-interoperability-techniques/

THANK YOU

OpenACC
More Science, Less Programming