

OpenACC CUDA Interoperability

OpenACC Course 2016

Contents

OpenACC is a team player!

- OpenACC can interplay with CUDA
- OpenACC can interplay with GPU-enabled libraries

Contents

OpenACC is a team player!

- OpenACC can interplay with CUDA
- OpenACC can interplay with GPU-enabled libraries

Motivation

The Keyword
Tasks

Task 1

Task 2

Task 3

Task 4

Usually, three reasons for mixing OpenACC with others

1 Libraries!

- A lot of hard problems have already been solved by others
- Make use of this!

Usually, three reasons for mixing OpenACC with others

1 Libraries!

- A lot of hard problems have already been solved by others
- Make use of this!

2 Existing environment

- You build up on other's work
- Part of code is already ported (e.g. with CUDA), the rest should follow
- OpenACC is a good first step in porting, CUDA a possible next

Usually, three reasons for mixing OpenACC with others

1 Libraries!

- A lot of hard problems have already been solved by others
- Make use of this!

2 Existing environment

- You build up on other's work
- Part of code is already ported (e.g. with CUDA), the rest should follow
- OpenACC is a good first step in porting, CUDA a possible next

3 OpenACC coverage

- Sometimes, OpenACC does not support specific *part* needed (very well)
- Sometimes, more fine-grained manipulation needed

```
host_data use_device
```

host_data use_device

- Background
 - GPU and CPU are different devices, have different memory
 - Distinct address spaces
 - OpenACC hides handling of addresses from user
 - For every chunk of accelerated data, **two** addresses exist
 - One for CPU data, one for GPU data
 - OpenACC uses appropriate address in accelerated kernel
 - **But:** Automatic handling not working when out of OpenACC (OpenACC will default to host address)
- **host_data use_device** uses the address of the GPU device data for scope

The host_data Construct

C

- Usage:

```
double* foo = new double[N];           // foo on Host
#pragma acc data copyin(foo[0:N])      // foo on Device
{
    ...
    #pragma acc host_data use_device(foo)
    some_lfunc(foo);                    // Device: OK!
    ...
}
```

- Directive can be used for structured block as well

The host_data Construct

Fortran

- Usage example

```
real(8) :: foo(N)           ! foo on Host
!$acc data copyin(foo)      ! foo on Device
...
!$acc host_data use_device(foo)
call some_func(foo);        ! Device: OK!
!$acc end host_data
...
!$acc end data
```

- Directive can be used for structured block as well

The Inverse: `deviceptr`

When CUDA is involved

- For the inverse case:
 - Data has been copied by CUDA or a CUDA-using library
 - Pointer to data residing on devices is returned
 - Use this data in OpenACC context
- `deviceptr` clause declares data to be on device

The Inverse: deviceptr

When CUDA is involved

- For the inverse case:
 - Data has been copied by CUDA or a CUDA-using library
 - Pointer to data residing on devices is returned
 - Use this data in OpenACC context
- deviceptr clause declares data to be on device
- Usage (C):

```
float * n;  
int n = 4223;  
cudaMalloc((void**)&x, (size_t)n*sizeof(float));  
// ...  
#pragma acc kernels deviceptr(x)  
for (int i = 0; i < n; i++) {  
    x[i] = i;  
}
```

The Inverse: deviceptr

When CUDA is involved

- For the inverse case:
 - Data has been copied by CUDA or a CUDA-using library
 - Pointer to data residing on devices is returned
 - Use this data in OpenACC context
- deviceptr clause declares data to be on device
- Usage (Fortran):

```
integer, parameter :: n = 4223
real, device, dimension(N) :: x  ! automatically on device
integer :: i

// ...
!$acc kernels deviceptr(x)
do i=1, n
    x(i) = i
end do
!$acc end kernels
```

Tasks

Tasks

Task 1

Task 1

Introduction to BLAS

- Use case: Anything linear algebra
- **BLAS**: Basic Linear Algebra Subprograms
 - Vector-vector, vector-matrix, matrix-matrix operations
 - Specification of routines
 - Examples: SAXPY, DGEMV, ZGEMM
 - <http://www.netlib.org/blas/>
- **cuBLAS**: NVIDIA's linear algebra routines with BLAS interface, readily accelerated
 - <http://docs.nvidia.com/cuda/cublas/>
- **Task 1**: Use cuBLAS for vector addition, everything else with OpenACC

Task 1

cuBLAS OpenACC Interaction

- cuBLAS routine used:

```
cublasDaxpy(cublasHandle_t handle, int n,  
            const double          *alpha,  
            const double          *x, int incx,  
            double                *y, int incy)
```

- handle capsules GPU auxiliary data, needs to be created and destroyed with `cublasCreate` and `cublasDestroy`
- `x` and `y` point to addresses on **device**!
- cuBLAS library needs to be linked with `-lcublas`

Task 1

cuBLAS on Fortran

- PGI offers bindings to cuBLAS out of the box

```
integer(4) function cublasdaxpy_v2(h, n, a, x, incx, y, incy)
  type(cublasHandle) :: h
  integer :: n
  real(8) :: a
  real(8), device, dimension(*) :: x, y
  integer :: incx, incy
```

- Usage: `use cublas` in code; add `-Mcuda -Lcublas` during compilation
 - Notes
 - Legacy (v1) cuBLAS bindings (no handle) also available, i.e. `cublasdaxpy()`
 - PGI's Fortran allows to omit `host_data use_device`, but not recommended
 - Module `openacc_cublas` exists, specifically designed for usage with OpenACC (no need for `host_data use_device`)
- ⇒ Both not part of training

→ <https://www.pgroup.com/doc/pgicudaint.pdf>

Task 1

Vector Addition with cuBLAS

- Location of code: Interoperability/tasks/{C, Fortran}/task1
- Parts of task:
Go through `vecAddRed.{c, F03}`, work on TODOs
 - Use `host_data` `use_device` to provide correct pointer
 - Check [cuBLAS documentation](#) for details on `cublasDaxpy()`
- Compile with `make`

Task 1

Vector Addition with cuBLAS

- Location of code: Interoperability/tasks/{C, Fortran}/task1
- Parts of task:
Go through `vecAddRed.{c, F03}`, work on TODOs
 - Use `host_data` `use_device` to provide correct pointer
 - Check [cuBLAS documentation](#) for details on `cublasDaxpy()`
- Compile with `make`

JURECA Getting Started

```
module load PGI CUDA
salloc --reservation=openacc --partition=gpus --nodes=1 --time=1:30:00
↪ --gres=mem128,gpu:4
make
srun ./vecAddRed.bin
```

Tasks

Task 2

Task 2

CUDA Need-to-Know

- Use case:
 - Working on legacy code
 - Need the *raw* power (/flexibility) of CUDA
 - CUDA need-to-knows:
 - Thread → Block → Grid
Total number of threads should map to your problem; threads are always given per block
 - A kernel is called from every thread on GPU device
Number of kernel threads: *triple chevron syntax*
`kernel<<<nBlocks, nThreads>>>(arg1, arg2, ...)`
 - Kernel: Function with `__global__` prefix
Aware of its index by global variables, e.g. `threadIdx.x`
- <http://docs.nvidia.com/cuda/>

Task 2

Vector Addition with CUDA Kernel: C

- **Task 2:** Use a CUDA kernel for vector addition, everything else with OpenACC
- Location of code: Interoperability/tasks/C/task2
- Marrying CUDA C and OpenACC:
 - All direct CUDA interaction wrapped in wrapper file `cudaWrapper.cu`, compiled with `nvcc` to object file (`-c`)
 - `vecAddRed.c` calls external function from `cudaWrapper.cu` (**extern**)
 - ↪ `vecAddRed.c:main() → cudaWrapper.cu:cudaVecAddWrapper() → cudaWrapper.cu:cudaVecAdd() → CUDA`
- Parts of task:
Go through `vecAddRed.c` and `cublasWrapper.cu`, work on TODOs
 - Use `host_data use_device` to provide correct pointer
 - Implement computation in kernel, implement call of kernel
- Again, use `make` to compile

Task 2

Vector Addition with CUDA Kernel: Fortran

- **Task 2:** Use a CUDA kernel for vector addition, everything else with OpenACC
- Location of code: Interoperability/tasks/Fortran/task2
- Marrying CUDA **Fortran** and OpenACC:
 - No need to use wrappers!
 - OpenACC and CUDA Fortran directly supported in same source
 - Having a dedicated module file could make sense anyway
- Parts of task:
Go through `vecAddRed.F03` and work on TODOs
 - Use `host_data use_device` to provide correct pointer
 - Implement computation in kernel, implement call of kernel
- Again, use `make` to compile

Tasks

Task 3

Task 3

Vector Addition with Thrust: C

- **Thrust**
 - Template library for CUDA C/C++ (similar to STL)
 - Offers many pre-made algorithms for popular computing tasks
 - Usually works with C++ iterators, but understands C arrays as well
 - <http://thrust.github.io/>
- **Task 3:** Use Thrust for reduction, everything else of vector addition with OpenACC
- Location of code: Interoperability/tasks/C/task3
- Parts of task:
Go through `vecAddRed.c` and `thrustWrapper.cu`, work on TODOs
 - Use `host_data` `use_device` to provide correct pointer
 - Implement call to `thrust::reduce` using `c_ptr`
- Use make for compilation

Task 3

Vector Addition with Thrust: Fortran

- **Thrust**
 - Template library for CUDA C/C++ (similar to STL)
 - Offers many pre-made algorithms for popular computing tasks
 - Usually works with C++ iterators, but understands C arrays as well
- <http://thrust.github.io/>
- **Task 3:** Use Thrust for reduction, everything else of vector addition with OpenACC
- Location of code Interoperability/tasks/Fortran/task3
- Parts of task:
Go through `vecAddRed.F09`, `thrustWrapper.cu` and `fortranthrust.F03`, work on TODOs
 - Thrust used via `ISO_C_BINDING` (*one more wrapper*) → familiarize yourself with setup
 - Use `host_data` `use_device` to provide correct pointer
 - Implement call to `thrust::reduce` using `c_ptr`
- Use make for compilation

Tasks

Task 4

Task 4

Stating the Problem

- We want to solve the Poisson equation

$$\Delta\Phi(x, y) = -\rho(x, y)$$

with periodic boundary conditions in x and y

- Needed, e.g., for finding electrostatic potential Φ for a given charge distribution ρ
- Model problem

$$\begin{aligned}\rho(x, y) &= \cos(4\pi x) \sin(2\pi y) \\ (x, y) &\in [0, 1)^2\end{aligned}$$

- Analytically known: $\Phi(x, y) = \Phi_0 \cos(4\pi x) \sin(2\pi y)$
- Let's solve the Poisson equation with a Fourier Transform!

Task 4

Introduction to Fourier Transforms

- Discrete Fourier Transform and Re-Transform:

$$\hat{f}_k = \sum_{j=0}^{N-1} f_j e^{-\frac{2\pi i k}{N} j} \quad \Leftrightarrow \quad f_j = \sum_{k=0}^{N-1} \hat{f}_k e^{\frac{2\pi i j}{N} k}$$

- Time for all \hat{f}_k : $\mathcal{O}(N^2)$
- Fast Fourier Transform: Recursively splitting $\rightarrow \mathcal{O}(N \log(N))$
- Find derivatives in Fourier space:

$$f'_j = \sum_{k=0}^{N-1} i k \hat{f}_k e^{\frac{2\pi i j}{N} k}$$

It's just multiplying by $i k$!

Task 4

Plan for FFT Poisson Solution

Start with charge density ρ

- 1 Fourier-transform ρ

$$\hat{\rho} \leftarrow \mathcal{F}(\rho)$$

- 2 Derive ρ in Fourier space twice

$$\hat{\phi} \leftarrow -\hat{\rho} / (k_x^2 + k_y^2)$$

- 3 Inverse Fourier-transform $\hat{\phi}$

$$\phi \leftarrow \mathcal{F}^{-1}(\hat{\phi})$$

Task 4

Plan for FFT Poisson Solution

Start with charge density ρ

- 1** Fourier-transform ρ

$$\hat{\rho} \leftarrow \mathcal{F}(\rho)$$

cuFFT

- 2** Derive ρ in Fourier space twice

$$\hat{\phi} \leftarrow -\hat{\rho} / (k_x^2 + k_y^2)$$

OpenACC

- 3** Inverse Fourier-transform $\hat{\phi}$

$$\phi \leftarrow \mathcal{F}^{-1}(\hat{\phi})$$

cuFFT

Task 4

cuFFT: C

- cuFFT: NVIDIA's (Fast) Fourier Transform library
 - 1D, 2D, 3D transforms; complex and real data types
 - Asynchronous execution
 - Modeled after FFTW library (API)
 - Part of CUDA Toolkit
 - Fortran: PGI offers bindings with **use** cufft
- <https://developer.nvidia.com/cufft>

```
cufftDoubleComplex *src, *tgt;           // Device data!
cufftHandle plan;
// Setup 2d complex-complex trafo w/ dimensions (Nx, Ny)
cufftCreatePlan(plan, Nx, Ny, CUFFT_Z2Z);
cufftExecZ2Z(plan, src, tgt, CUFFT_FORWARD); // FFT
cufftExecZ2Z(plan, tgt, tgt, CUFFT_INVERSE); // iFFT
// Inplace trafo  ^----^
cufftDestroy(plan);           // Clean-up
```

Task 4

cuFFT: Fortran

- cuFFT: NVIDIA's (Fast) Fourier Transform library
 - 1D, 2D, 3D transforms; complex and real data types
 - Asynchronous execution
 - Modeled after FFTW library (API)
 - Part of CUDA Toolkit
 - Fortran: PGI offers bindings with **use** cufft
- <https://developer.nvidia.com/cufft>

```
double complex, allocatable :: src(:, :), tgt(:, :) ! Device
integer :: plan, ierr
! Setup 2d complex-complex trafo w/ dimensions (Nx, Ny)
ierr = cufftCreatePlan(plan, Nx, Ny, CUFFT_Z2Z)
ierr = cufftExecZ2Z(plan, src, tgt, CUFFT_FORWARD) ! FFT
ierr = cufftExecZ2Z(plan, tgt, tgt, CUFFT_INVERSE) ! iFFT
! Inplace trafo          ^-----^
ierr = cufftDestroy(plan)                                ! Clean-up
```

Task 4

Synchronizing cuFFT: C

- CUDA Streams enable interleaving of computational tasks
- cuFFT uses streams for asynchronous execution
- cuFFT runs in default CUDA stream;
OpenACC not → trouble

⇒ Force cuFFT on OpenACC stream

```
#include <openacc.h>
// Obtain the OpenACC default stream id
cudaStream_t accStream =
    (cudaStream_t) acc_get_cuda_stream(acc_async_sync) ;
// Execute all cuFFT calls on this stream
cufftSetStream(accStream);
```

Task 4

Synchronizing cuFFT: Fortran

- CUDA Streams enable interleaving of computational tasks
- cuFFT uses streams for asynchronous execution
- cuFFT runs in default CUDA stream;
OpenACC not → trouble

⇒ Force cuFFT on OpenACC stream

```
use openacc
```

```
integer :: stream
```

```
! Obtain the OpenACC default stream id
```

```
stream = acc_get_cuda_stream(acc_async_sync)
```

```
! Execute all cufft calls on this stream
```

```
ierr = cufftSetStream(plan, stream)
```

Task 4

OpenACC and cuFFT

- Use case: Fourier transforms
- **Task 4:** Use cuFFT and OpenACC to solve Poisson's Equation
- Location of code: Interoperability/tasks/{C, Fortran}/task4
- Parts of task:
Go through `poisson.{c, Fortran}` and work on TODOs
`solveRSpace` Force cuFFT on correct stream; implement data handling with `host_data use_device`
`solveKSpace` Implement data handling and parallelism
- Use make for compilation
- *Note for Fortran: Code not well-tested! Might contain errors.*

- If needed, OpenACC can play team with
 - GPU-accelerated libraries
 - Plain CUDA code
- Link externally compiled object (e.g. with `nvcc`) into PGI-compiled OpenACC program
Alternative: use `-ccbin=pgc++` as a `nvcc` flag
- For Fortran, `ISO_C_BINDING` might be needed