OpenACC CUDA Interoperability

JSC OpenACC Course 2024

October 2024 | Kaveh Haghighi Mood, Andreas Herten | Forschungszentrum Jülich

Contents

OpenACC is a team player!

OpenACC can interplay with

- CUDA
- GPU-enabled libraries and applications

Contents

OpenACC is a team player!

OpenACC can interplay with

- CUDA
- GPU-enabled libraries and applications

Motivation

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!

Motivation

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

Motivation

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

The Keyword

OpenACC's Rosetta Stone

`host_data use_device`

The Keyword

OpenACC's Rosetta Stone

`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

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

The `host_data` Construct

Fortran

- Usage example

```fortran
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
```

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

C

- 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):

```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):

```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: *S*AXPY, *D*GEMV, *Z*GEMM

  → `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

FORTRAN

- Nvidia 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()`
  - nvfortran 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://docs.nvidia.com/hpc-sdk/compilers/pdf/hpc209cudaint.pdf

Task 1

Vector Addition with cuBLAS

TASK | C | FORTRAN

- Use cuBLAS for vector addition

Task 1: OpenACC+cuBLAS

- Location of code:
  `5-Interoperability/Tasks/{C,Fortran}/Tasks/Task1`
- Work on TODOs in `vecAddRed.{c,F03}`
  - Use `host_data use_device` to provide correct pointer
  - Check cuBLAS documentation for details on `cublasDaxpy()`
- Compile: `make`
- Submit to the batch system: `make run`

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 $\rightarrow$ Block $\rightarrow$ Grid
    Total number of threads should map to your problem; threads are alway 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`

$\rightarrow$ http://docs.nvidia.com/cuda/

Task 2

TASK
C

Vector Addition with CUDA Kernel: C

- CUDA kernel for vector addition, rest OpenACC
- Marrying CUDA **C** and OpenACC:
  - No need to use wrappers! OpenACC and CUDA directly supported by nvc++

Task 2: OpenACC+CUDA

- Change to `5-Interoperability/Tasks/C/Tasks/Task2` directory
- Work on TODOs in `vecAddRed.c`
  - Use `host_data use_device` to provide correct pointer
  - Implement computation in kernel, implement call of kernel
- Compile: `make`; Submit to the batch system: `make run`

Task 2

TASK
FORTRAN

Vector Addition with CUDA Kernel: Fortran

- CUDA kernel for vector addition, rest OpenACC
- 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

Task 2: OpenACC+CUDA

- Change to `5-Interoperability/Tasks/Fortran/Tasks/Task2` directory
- Work on TODOs in `vecAddRed.F03`
  - Use `host_data use_device` to provide correct pointer
  - Implement computation in kernel, implement call of kernel
- Compile: `make`; Submit to the batch system: `make run`

Tasks

Task 3

Thrust

Iterators! Iterators everywhere! 🚀

- $\frac{\text{Thrust}}{\text{CUDA}} = \frac{\text{STL}}{\text{C++}}$
- Template library
- Based on iterators, but also works with plain C
- Data-parallel primitives (`scan()`, `sort()`, `reduce()`, …); algorithms

→ `http://thrust.github.io/`
`http://docs.nvidia.com/cuda/thrust/`

Thrust

Code example

```
int a = 42;
int n = 10;
thrust::host_vector<float> x(n), y(n);
// fill x, y

thrust::device_vector d_x = x, d_y = y;

using namespace thrust::placeholders;
thrust::transform(d_x.begin(), d_x.end(), d_y.begin(), d_y.begin(), a * _1 + _2);

x = d_x;
```

Task 3

Vector Addition with Thrust: C

TASK

C

- Use Thrust for reduction, everything else of vector addition with OpenACC

Task 3: OpenACC+Thrust

- Change to directory `5-Interoperability/Tasks/C/Tasks/Task3`
- Work on TODOs in `vecAddRed.c`
  - Use `host_data use_device` to provide correct pointer
  - Implement call to `thrust::reduce` using `c_ptr`
- Compile: `make`
- Submit to the batch system: `make run`

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Task 3

Vector Addition with Thrust: Fortran

TASK
FORTRAN

- Use Thrust for reduction, everything else of vector addition with OpenACC
- Thrust used via `ISO_C_BINDING` (*one more wrapper*)

Task 3: OpenACC+Thrust

- Change to directory `5-Interoperability/Tasks/Fortran/Tasks/Task3`
- Work on TODOs in `vecAddRed.F03`, `thrustWrapper.cu` and `fortranthrust.F03`
  - Familiarize yourself with setup of Thrust called via `ISO_C_BINDING`
  - Use `host_data use_device` to provide correct pointer
  - Implement call to `thrust::reduce` using `c_ptr`
- Compile: `make`
- Submit to the batch system: `make run`

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 ik!

Task 4

Plan for FFT Poisson Solution

Start with charge density ρ

1. Fourier-transform ρ
   $\hat{\rho} \leftarrow \mathcal{F}(\rho)$
2. *Integrate* ρ in Fourier space twice
   $\hat{\phi} \leftarrow -\hat{\rho} / \left(k_x^2 + k_y^2\right)$
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
   $\hat{\rho} \leftarrow \mathcal{F}(\rho)$
2. *Integrate* ρ in Fourier space twice
   $\hat{\phi} \leftarrow -\hat{\rho} / \left(k_x^2 + k_y^2\right)$
3. Inverse Fourier-transform $\hat{\phi}$
   $\phi \leftarrow \mathcal{F}^{-1}(\hat{\phi})$

cuFFT

OpenACC

cuFFT

Task 4

cuFFT: C

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: NVIDIA 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: NVIDIA 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

C

Synchronizing cuFFT: C

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

$\Rightarrow$ Force cuFFT on OpenACC stream

```c
#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

FORTRAN

Synchronizing cuFFT: Fortran

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

$\Rightarrow$ Force cuFFT on OpenACC stream

```fortran
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

TASK
C
FORTRAN

- Use case: Fourier transforms
- Use cuFFT and OpenACC to solve Poisson's Equation

Task 4: OpenACC+cuFFT

- Change to `Interoperability/Tasks/{C,Fortran}/Tasks/Task4` directory
- Work on TODOs in `poisson.{c,F03}`

  `solveRSpace` Force cuFFT on correct stream; implement data handling with `host_data use_device`

  `solveKSpace` Implement data handling and parallelism
- Compile: `make`
- Submit to the batch system: `make run`

Summary & Conclusion

- If needed, OpenACC can play team with
    - GPU-accelerated libraries
    - Plain CUDA code
- For Fortran, `ISO_C_BINDING` might be needed

Summary & Conclusion

- If needed, OpenACC can play team with
  - GPU-accelerated libraries
  - Plain CUDA code
- For Fortran, `ISO_C_BINDING` might be needed

Thank you
for your attention!

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Appendix

Glossary

References

List of Tasks

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE

Glossary I

CUDA Computing platform for GPUs from NVIDIA. Provides, among others, CUDA C/C++. 2, 3, 4, 5, 6, 11, 12, 13, 21, 22, 23, 34, 35, 36, 37, 39, 40, 42

NVIDIA US technology company creating GPUs. 16, 34, 35, 43

OpenACC Directive-based programming, primarily for many-core machines. 1, 2, 3, 4, 5, 6, 7, 8, 11, 12, 13, 16, 17, 18, 19, 22, 23, 27, 28, 32, 33, 36, 37, 38, 39, 40, 42

Thrust A parallel algorithms library for (among others) GPUs. See `https://thrust.github.io/`. 25, 26, 27, 28, 42

References: Images, Graphics I

[1] Chester Alvarez. *Untitled*. Freely available at Unsplash. URL: `https://unsplash.com/photos/bphc6kyobMg`.

JÜLICH Forschungszentrum | JÜLICH SUPERCOMPUTING CENTRE