Main functionalities needed for offloading

- Mapping data
  - allocation on the device
  - copy data to and from the device
- Execute kernels
  - libraries
    - BLAS/LAPACK
    - cuFFT
    - ...
  - Single Block on Multiple DATA
    - loops
    - array operations in Fortran
  - Synchronization functionalies ##

slide: https://hackmd.io/@pietrodelugas/OpenACC1

OpenACC

OpenACC API collects compiler directives and library routines to offload
code regions of standard HPC codes in C, C++, and Fortran

- Mapping data with data constructs
  - Structured data regions
  - Unstructured data regions
- Kernels:
  - routine directive for device routines
  - Compute constructs to offload parallelizable regions of code in host routines
    - Kernels construct
    - Parallel construct
    - Serial construct
- Managing mapped data
  - host_data construct

- update directive

AXPY example

```
#include<vector>
#include<cmath>
#include<iomanip>
#include<openacc.h>
using namespace std;

int main(){
  constexpr size_t N = 100000;
  constexpr double  alpha = M_PI/double(N);
  vector<double> a(N), b(N), c(N);
  fill(a.begin(), a.end(), 1.0);
  fill(b.begin(), b.end(), 2.0);
  fill(c.begin(), c.end(), 0.0);
  double* buff_a = a.data();
  double* buff_b = b.data();
  double* buff_c = c.data();
  #pragma acc parallel copyout(buff_c[0:N])
  #pragma acc loop vector
  for (size_t i=0; i < N; ++i){
    buff_c[i] = buff_c[i] + i * alpha *  (buff_a[i] + buff_b[i]);
  }
#if defined(_OPENACC)
  cout << acc_is_present(buff_c,N) << endl;
#endif
  cout << scientific << setprecision(6);
  for (size_t  i =0, start=0, stop = start + 5 < N ? start +5: N; i<4; ++i){
    for (size_t j=start; j < stop; ++j){
      cout << setw(16) << c[j] << " ";
    }
    cout<< endl;
    start = start + 5;
    stop = start + 5 < N ? start + 5: N;
  }
}
```

---

looking how it works

- We are using NVidia compilers
  - C++: `nvc++`
  - Fortran: `nvfortran`
- to see how the compiler offloads use `-Minfo=accel` of `-Minfo`
  - `nvfortran -acc -Minfo prova.f90`

- `nvc++ -acc -Minfo=accel prova.cpp`

```
18, Generating copyout(buff_c[:100000]) [if not already present]
       Generating implicit copyin(buff_b[:100000]) [if not already present]
       Generating NVIDIA GPU code
       21, #pragma acc loop vector(128) /* threadIdx.x */
    18, Generating implicit copyin(buiff_a[:100000]) [if not already present]
    21, Loop is parallelizable
```

---

looking how it works II

- The compiler generates kernels parallelizing the loops
  - gang –> blocks
  - vector –> threads
- Data movement is generated automatically, only output data are copied out
- No data remains resident on the GPU in this way (unless already present)
  - No need to free data on GPUs

---

looking how it works III (the kernels construct)

- The kernels construct instructs the compiler to run the following code block in the device, but it leave to the compiler to decide if it is save to parallelize it or execute it serially

- if we replace the parallel construct with the kernels we have:

```
    18, Generating implicit copyin(buff_b[:100000]) [if not already present]
        Generating implicit copy(buff_c[:100000]) [if not already present]
        Generating implicit copyin(buff_a[:100000]) [if not already present]
    20, Complex loop carried dependence of buff_b->,buff_a-> prevents parallelizа
        Loop carried dependence of buff_c-> prevents parallelization
        Loop carried backward dependence of buff_c-> prevents vectorization
        Accelerator serial kernel generated
        Generating NVIDIA GPU code
        20, #pragma acc loop seq
    20, Complex loop carried dependence of buff_a-> prevents parallelization
        Loop carried backward dependence of buff_c-> prevents vectorization
```

- with pointers the compiler cannot say if there is a race condition that preempts loop parallelization, it is up to the programmer check it and instruct the compiler:
  - use parallel loop instead of kernel

- use the `__restrict__` keyword when declaring the pointers (e.g.)

```
auto* __restrict__ a_ptr=a.data();
auto* __restrict__ b_ptr=b.data();
auto* __restrict__ c_ptr=c.data();
```

`

managing the data movement with the data construct

When we have more kernels it is important to avoid that the data move back and forth from host to device:

```
int main(){
  constexpr size_t N = 100000;
  constexpr double  alpha = M_PI/double(N);
  vector<double> a(N), b(N), c(N);
  fill(a.begin(), a.end(), 1.0);
  fill(b.begin(), b.end(), 2.0);
  fill(c.begin(), c.end(), 0.0);
  double* buff_a = a.data();
  double* buff_b = b.data();
  double* buff_c = c.data();
  // we first compute a + b
  axpy(N, 1.0, buff_a, buff_b);
#if defined(_OPENACC)
  // we check whether the data are present in the Device 0 no, 1 yes
  cout << acc_is_present(buff_b,N) << endl;
#endif
  // we now compute  c[i] = c[i] + i * alpha * b[i]
  #pragma acc parallel
  #pragma acc loop vector
  for (size_t i=0; i < N; ++i){
    buff_c[i] = buff_c[i] + i * alpha *  buff_b[i];
  }
// check again data presence on device
#if defined(_OPENACC)
  cout << acc_is_present(buff_c,N) << endl;
#endif
  cout << scientific << setprecision(6);
  for (size_t  i =0, start=0, stop = start + 5 < N ? start +5: N; i<4; ++i){
    for (size_t j=start; j < stop; ++j){
      cout << setw(16) << c[j] << " ";
    }
    cout<< endl;
    start = start + 5;
    stop = start + 5 < N ? start + 5: N;
  }
}
```

Using a structured data construct

```
#pragma acc data copyin(buff_a[:N], buff_b[:N]) copy(buff_c[:N])
  {
    axpy(N, 1.0, buff_a, buff_b);
    cout << b[4] << setw(5) << buff_b[4] << endl;
#if defined(_OPENACC)
    cout << acc_is_present(buff_b,N) << endl;
#endif

    #pragma acc parallel
    #pragma acc loop vector
    for (size_t i=0; i < N; ++i){
      buff_c[i] = buff_c[i] + i * alpha *  buff_b[i];
    }
#if defined(_OPENACC)
    cout << acc_is_present(buff_c,N) << endl;
#endif
  }
  auto start = c.begin();
  cout << scientific << setprecision(6);
  for (int  i =0; i<4; ++i){
    for (double val: vector<double>(start, start+5)){
      cout << setw(16) << val << " ";
    }
    cout<< endl;
    start = start + 5;
  }
```

- The structured data construct defines a code region on wich data are present on the device
- at the end of the execution of the region data are removed or their reference is decreased (if they were already present they remain on the device)
- according to the mapping claused data may be copied back to host or completely discarded.

Orphaned data regions

Host routines can be called from inside data regions; in this case the data region inherits the data mapped from the calling data region, e.g:

```
#include<curand.h>
int  fill_random_vector(double* buff, size_t N, long seed){
  curandGenerator_t gen;
  auto status = curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
  status = curandSetPseudoRandomGeneratorSeed(gen, seed);
#pragma acc data present_or_copyout(buff[:N])
  {
#pragma acc host_data use_device(buff)
    {
      status = curandGenerateUniformDouble(gen, buff, N);
    }
  }
  return status == CURAND_STATUS_SUCCESS;
}

```

- the routine can receive either data already present on the device or host variables that are mapped in the device during the execution of its code region.
- if present data are not copied out to the device but remain in the device
- notice here the `host_data use device` directive. We are using a cuda library routine that expects a device memory pointer; this directive instructs to pass it when it's name is used.

Unstructured data construct

- when the lifetime of the device data is not restricted to specific code regions but depend on runtime conditions it is possible to map data using the unstructured data constructs:
  - `acc enter data create(...) copyin(...)` maps data to the device memory.
  - `acc exit data delete(...) copyout(...)` maps data out from the device memory.

---

using cuda libraries

- cuda provides various libraries for GPUs with Host APIs
- they are executed from host code regions
- they expect data allocated on the device in examples usually these data are allocated with cudaMalloc but it is possible to use also data mapped with openACC
  - you can use the `host_data usedevice(...)` construct, within this regions the device address associated with these buffers is passed to called routines.
- for linking and accessing the includes with nv compilers (nvc, nvc++, nvfortran) use the option `-cudalib:[curand][cublas][cufft][cusolver]`

Double loops reductions, gangs and vectors:

Nested loops can express bidimensional data or sometime different levels of parallelism. A simple example it the GAXPY operation that performs \(Y = A\cdot X+Y\).

```
int main(){
  // initialize the pseudo-random-number gererator of curand
  curandGenerator_t generator;
  auto status = curandCreateGenerator(&generator, CURAND_RNG_PSEUDO_DEFAULT);
    if (status != CURAND_STATUS_SUCCESS){
      cerr << "Curand Create Generator failed"<< endl;
      return 0;
    }
  status = curandSetPseudoRandomGeneratorSeed(generator, 241202);
  if (status != CURAND_STATUS_SUCCESS){
    cerr << "Curand set seed failed"<< endl;
    return 0;
  }
  const size_t  N=10000, N2=N*N;
  vector<double>  A(N2), X(N), Y(N);
  double* A_data = A.data();
  double* X_data = X.data();
  double* Y_data = Y.data();
  fill(X.begin(), X.end(),0.);
  fill(Y.begin(), Y.end(),0.);
#pragma acc data create(A_data[:N2], X_data[:N]) copyout(Y_data[:N])
  {
//fill X and Y with random numbers using curand
#pragma acc host_data use_device(X_data,Y_data)
     {
      auto statusX  = curandGenerateUniformDouble(generator, X_data, N);
      auto statusY  = curandGenerateUniformDouble(generator, Y_data, N);
     }
#pragma acc update self(X_data[:10])
    write_vector_start(X, 6);
//generates matrix A
#pragma acc parallel loop
    for (size_t i=0; i < N2; ++i){
      A_data[i] = 1.0/(i%N+1 + i/N);
    }
#pragma acc update self(A_data[:N2])
    write_matrix_block(A, 6, N);
// perform y = AX + y
  double dotproduct;
#pragma acc parallel
#pragma acc loop gang
  for (int i=0;i<N;++i){
    dotproduct=0;
#pragma acc loop vector reduction(+:dotproduct)
    for (int j=0; j<N; ++j){
      dotproduct += A_data[j*N+i]*X_data[j];
    }
    Y_data[i] +=dotproduct;
  }

  }//ends data region
  write_vector_start(Y, 6);
}
```

- the GAXPY calculation involves two loops:
  - the exterior loop runs over the components of the Y vector ($\mathcal{O}(N)$)

- to each component of \(Y\) is added the dotproduct of a row of \(A\) times \(X\), again \(\mathcal{O}(N)\)
  - We notice that here we are performing a reduction, they are ordinarily detected by the compiler but is safer to indicate them in the directive.
  - Important thing to note:
    - scalars are firstprivate by default
    - arrays are share by default, if private arrays are needed the need to be added in the clause.

Kernel routines with OpenACC, the routine construct.

- The second loop above can also be cast as a reusable routine, but if we want to call it inside a device compute region it must be a device routine:

```
#pragma acc routine vector
double dotproduct(int dim, double* V1, int inc1, double* V2, int inc2){
  double somma=0.0;
#pragma acc loop reduction(+:somma)
  for (int i =0; i< dim; ++i){
    somma +=  V1[i*inc1]*V2[i*inc2];
  }
  return somma;
}
```

In this case the double loop of GAXPY can be written as:

```
#pragma acc parallel
#pragma acc loop gang
  for (int i=0;i<N;++i){
    Y_data[i] += dotproduct(N, A_data+i, N, X_data, 1);
  }
```

GAXPY directly from cublas:

we can obviously use the `dgemv` of cublas to execute the gaxpy:

```
    cublasHandle_t handle;
    cublasCreate(&handle);
    const double one=1.0;
#pragma acc host_data use_device(A_data, X_data, Y_data)
    {
      cublasDgemv(handle, CUBLAS_OP_N, N, N, &one, A_data, N, X_data, 1, &one
    }
```

- creating a cublas handle can be timeconsuming, should be initilized at the beginning of the execution and stored for the rest of the program.

An example with MPI

- MPI data passing can be done directly from device to device using GPU-aware MPI
  - The GPU-AWARE library is activated simply using device pointers.
  - Use the openmpi provided with nvhpc-sdk or other versions compiled with GPU aware enabled.

```
    if (rank == 2) {
       #pragma acc host_data use_device(mybuff)
       MPI_Reduce(MPI_IN_PLACE, mybuff, N, MPI_DOUBLE, MPI_SUM, 2, MPI_COMM_WC
     } else {
         #pragma acc host_data use_device(mybuff)
         MPI_Reduce(mybuff, nullptr, N, MPI_DOUBLE, MPI_SUM, 2, MPI_COMM_WORLI
     }
```

When uning MPI, it is important to avoid that all ranks use the same GPU. Distribute the GPUs of the node uniformly on all ranks is up to the programmer.

- This can also be done using the cuda runtime APIs:

```

  MPI_Init(&argc, &argv);
   int rank;
   int mpisize;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &mpisize);
   int num_of_gpus = acc_get_num_devices(acc_device_nvidia);
   acc_set_device_num(rank%mpisize, acc_device_nvidia);
   int mydevice = acc_get_device_num(acc_device_nvidia);
   cout << "hello, rank " << rank << "  is using device " << mydevice << endl;

```

Exercises:

- Write a routine that performs matrix multiplication using `dgemm` from the cublas library
- use openacc to accelerate the kernels of the MPI parallel Jacobi solver.
- compare times and scaling of the CPU and GPU versions of your solver.