OpenMP device constructs

Mandes Schönherr (Cray Inc. Germany)
mandes.schoenherr@cray.com

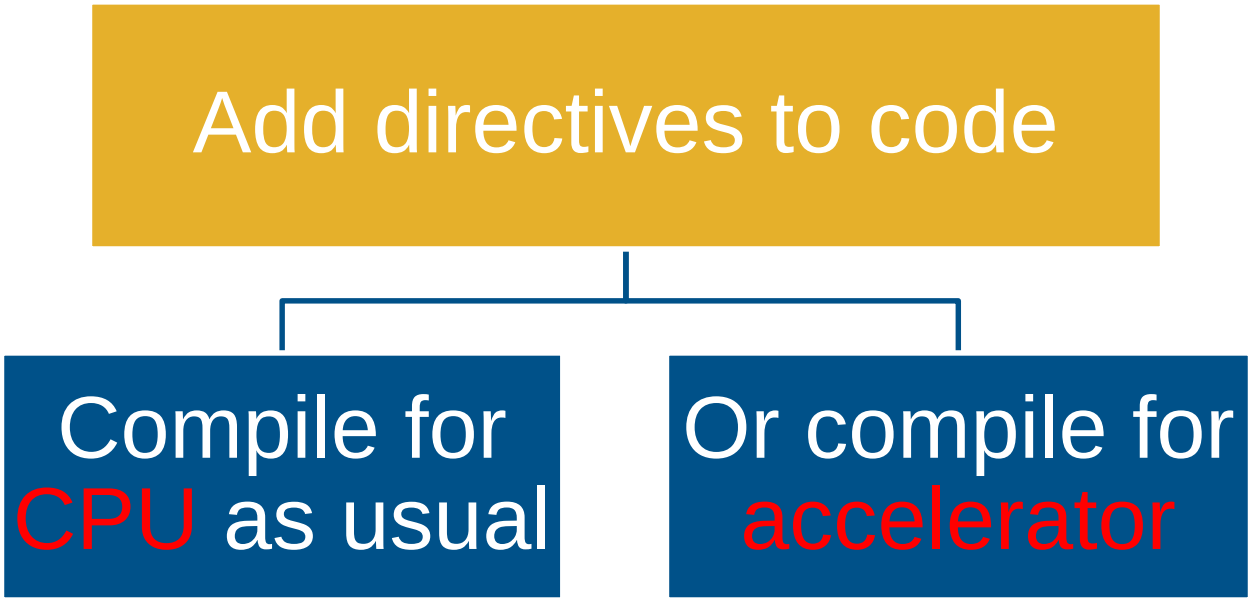
Alistair Hart
(Cray UK Ltd.)

Content

- **Why directive based accelerator models?**
- **Introduction to OpenMP and OpenACC**
  - First kernel
  - data scoping
  - data regions

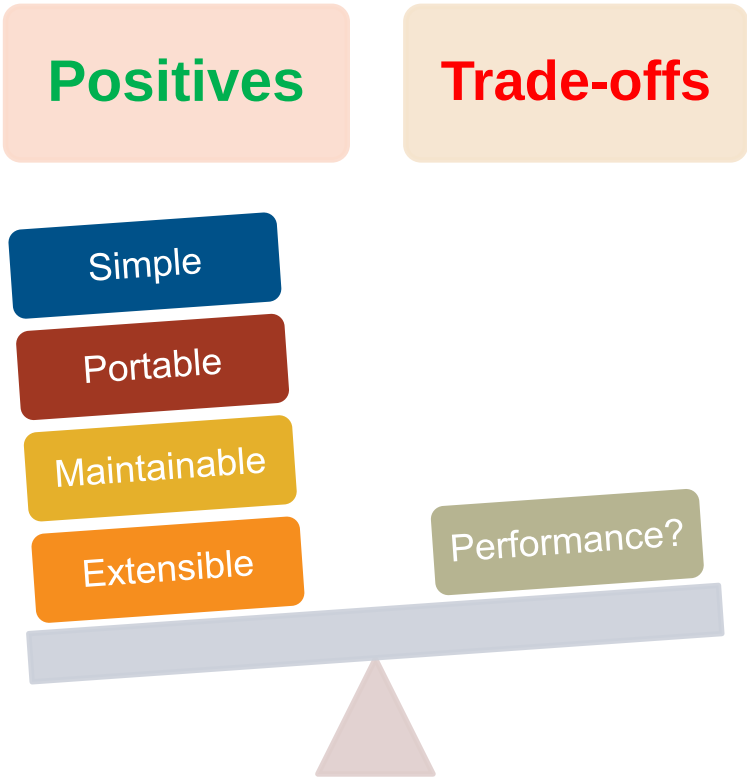
Directive based programming

- **Directives are comments in the code, which can be used as hints by the compiler**



Why use accelerator directive models?

- Many advantages compared to accelerator programming languages (e.g. CUDA or OpenCL)



Motivating Example: Reduction

- Sum elements of an array
- Original Fortran code

```
a = 0.0

do i = 1,n
  a = a + b(i)
end do
```

2.0 GFlops

The reduction code in simple CUDA

```
__global__ void reduce0(int *g_idata, int *g_odata)
{
extern __shared__ int sdata[];

unsigned int tid = threadIdx.x;
unsigned int i = blockIdx.x*blockDim.x + threadIdx.x;
sdata[tid] = g_idata[i];
__syncthreads();

for(unsigned int s=1; s < blockDim.x; s *= 2) {
if ((tid % (2*s)) == 0) {
sdata[tid] += sdata[tid + s];
}
__syncthreads();
}

if (tid == 0) g_odata[blockIdx.x] = sdata[0];
}


extern "C" void reduce0_cuda_(int *n, int *a, int *b)
{
int *b_d,red;
const int b_size = *n;


cudaMalloc((void **) &b_d , sizeof(int)*b_size);
cudaMemcpy(b_d, b, sizeof(int)*b_size,
cudaMemcpyHostToDevice);
```

```
dim3 dimBlock(128, 1, 1);
dim3 dimGrid(2048, 1, 1);
dim3 small_dimGrid(16, 1, 1);

int smemSize = 128 * sizeof(int);
int *buffer_d, *red_d;
int *small_buffer_d;

cudaMalloc((void **) &buffer_d , sizeof(int)*2048);
cudaMalloc((void **) &small_buffer_d , sizeof(int)*16);
cudaMalloc((void **) &red_d , sizeof(int));


reduce0<<< dimGrid, dimBlock, smemSize >>>(b_d, buffer_d);

reduce0<<< small_dimGrid, dimBlock, smemSize >>>(buffer_d,
small_buffer_d);

reduce0<<< 1, 16, smemSize >>>(small_buffer_d, red_d);

cudaMemcpy(&red, red_d, sizeof(int),
cudaMemcpyDeviceToHost);

*a = red;

cudaFree(buffer_d);
cudaFree(small_buffer_d);
cudaFree(b_d);
}
```

41 lines CUDA
1.74 GFlops

The reduction code in optimized CUDA

```
template<class T>
struct SharedMemory
{
    __device__ inline operator       T*()
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }

    __device__ inline operator const T*() const
    {
        extern __shared__ int __smem[];
        return (T*)__smem;
    }
};

template <class T, unsigned int blockSize, bool nIsPow2>
__global__ void
reduce6(T *g_idata, T *g_odata, unsigned int n)
{
    T *sdata = SharedMemory<T>();

    unsigned int tid = threadIdx.x;
    unsigned int i = blockIdx.x*blockSize*2 + threadIdx.x;
    unsigned int gridSize = blockSize*2*gridDim.x;

    T mySum = 0;
    while (i < n)
    {
        mySum += g_idata[i];
        if (nIsPow2 || i + blockSize < n)
            mySum += g_idata[i+blockSize];
        i += gridSize;
    }
sdata[tid] = mySum;
    __syncthreads();

    if (blockSize >= 512) { if (tid < 256) { sdata[tid] = mySum = mySum
+ sdata[tid + 256]; } __syncthreads(); }
    if (blockSize >= 256) { if (tid < 128) { sdata[tid] = mySum = mySum
+ sdata[tid + 128]; } __syncthreads(); }
    if (blockSize >= 128) { if (tid <  64) { sdata[tid] = mySum = mySum
+ sdata[tid +  64]; } __syncthreads(); }
```

```
  if (tid < 32)
   {
      volatile T* smem = sdata;
      if (blockSize >=  64) { smem[tid] = mySum = mySum + smem[tid + 32];  }
      if (blockSize >=  32) { smem[tid] = mySum = mySum + smem[tid + 16];  }
      if (blockSize >=  16) { smem[tid] = mySum = mySum + smem[tid + 8];  }
      if (blockSize >=   8) { smem[tid] = mySum = mySum + smem[tid + 4];  }
      if (blockSize >=   4) { smem[tid] = mySum = mySum + smem[tid + 2];  }
      if (blockSize >=   2) { smem[tid] = mySum = mySum + smem[tid + 1];  }
   }

   if (tid == 0)
      g_odata[blockIdx.x] = sdata[0];
}
extern "C" void reduce6_cuda_(int *n, int *a, int *b)
{
   int *b_d;
   const int b_size = *n;

   cudaMalloc((void **) &b_d , sizeof(int)*b_size);
   cudaMemcpy(b_d, b, sizeof(int)*b_size, cudaMemcpyHostToDevice);

   dim3 dimBlock(128, 1, 1);
   dim3 dimGrid(128, 1, 1);
   dim3 small_dimGrid(1, 1, 1);
   int smemSize = 128 * sizeof(int);
   int *buffer_d;
   int small_buffer[4],*small_buffer_d;

   cudaMalloc((void **) &buffer_d , sizeof(int)*128);
   cudaMalloc((void **) &small_buffer_d , sizeof(int));
   reduce6<int,128,false><<< dimGrid, dimBlock, smemSize >>>(b_d,buffer_d, b_size);
   reduce6<int,128,false><<< small_dimGrid, dimBlock, smemSize
>>>(buffer_d, small_buffer_d,128);
   cudaMemcpy(small_buffer, small_buffer_d, sizeof(int),
cudaMemcpyDeviceToHost);

   *a = *small_buffer;

   cudaFree(buffer_d);
   cudaFree(small_buffer_d);
   cudaFree(b_d);
}
```

75 lines CUDA

10.5 GFlops

STORE

The reduction code in OpenMP

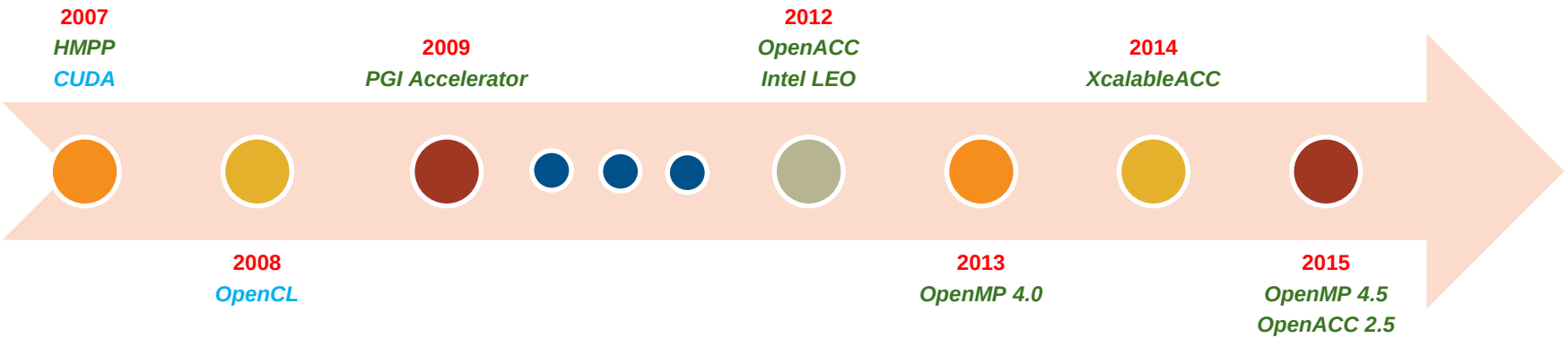
- **Compiler does the work:**
  - Identifies parallel loops within the region
  - Splits the code into accelerator and host portions
  - Workshares loops running on accelerator
    - Uses MIMD and SIMD parallelism
  - Data movement
    - allocates/frees GPU memory at start/end of region
    - moves data to/from GPU

```
! Assume outer data region has
! placed array b on accelerator
a = 0.0

!$omp target teams distribute &
!$omp         reduction(+:a)
do i = 1,n
  a = a + b(i)
end do
!$omp end target teams distribute
```

8.32 GFlops

Accelerator directives are not new



OpenMP

- **OpenMP 1.0 in Oct. 1997**
  - 8 members in OpenMP ARB
  - 2014 OpenMP ARB consits of 25 members
  - Standard available at openmp.org
- **July 2013 OpenMP 4.0 specifications released**
  - Including device constructs
  - API for Fortran, C/C++ for shared-memory parallel programming
- **Nov 2015 OpenMP 4.5 specifications released**
  - Futher constructs for devices
  - Target: CCE 8.5, full 8.6

OpenACC Directives for Accelerators

CRAY

- **Announced at SC11 conference**
  - Drawn up by: NVIDIA, Cray, PGI, CAPS
  - Works for Fortran, C, C++
    - Standard available at openacc.org
    - Initially implementations targeted at NVIDIA GPUs
- **Compiler support: all now complete**
  - Cray CCE: complete OpenACC 2.0 in v8.2
  - PGI Accelerator: v12.6 onwards
  - gcc: work started in late 2013
  - Various other compilers in development

CAPS CRAY THE SUPERCOMPUTER COMPANY NVIDIA The Portland Group

OpenMP vs. OpenACC: model approach

- **OpenMP**
  - aims for programmability
  - More general definition of pragmas
  - Prescriptive approach to parallel programming

- **OpenACC**
  - aims for portable performance
  - Focus on directives for accelerators
  - Descriptive approach to parallel programming

OpenMP or OpenACC?

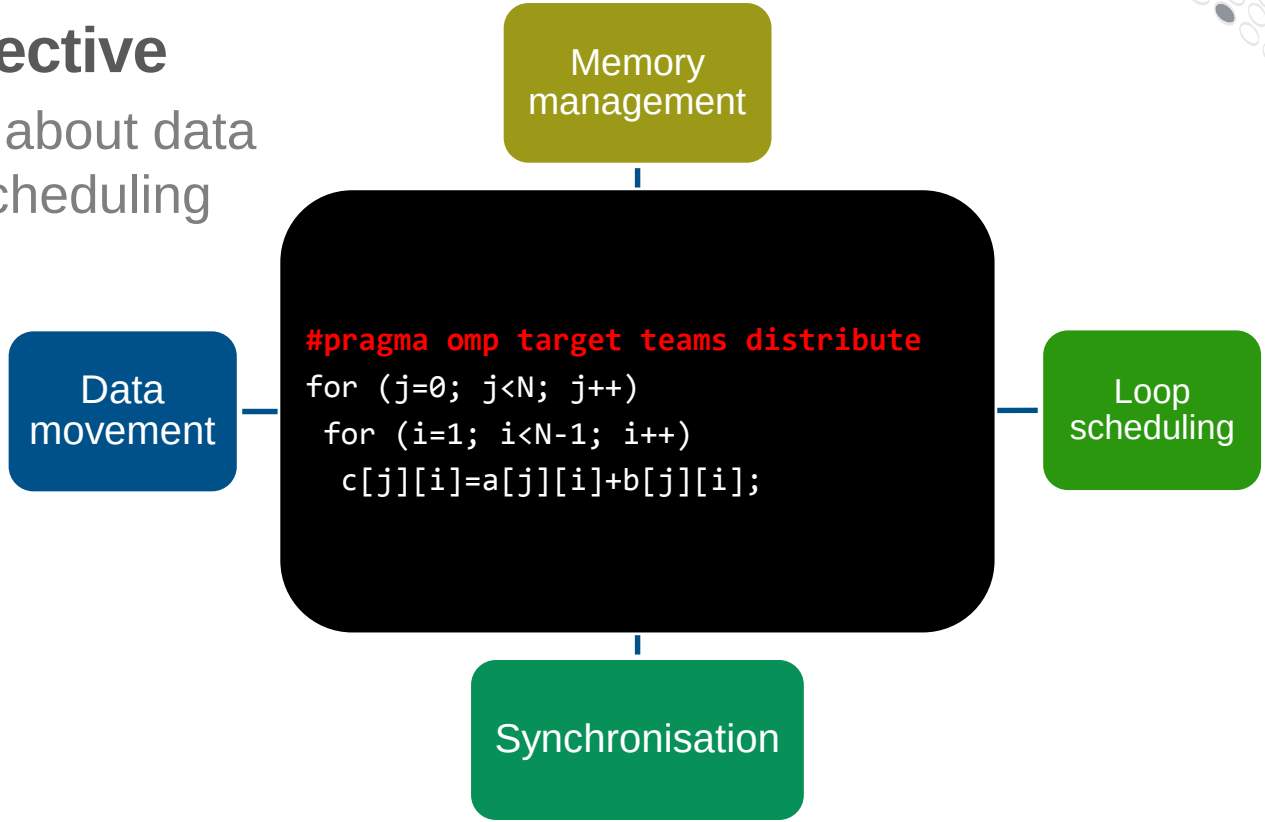
- **OpenMP and OpenACC are very similar ways of exposing information to the compiler**
  - information about data locality
  - information about parallelism (within and between loopnests)
- **The hard work is getting that information**
  - Not in choosing the way to express it
- **It is straightforward to migrate from OpenACC to OpenMP**
  - or even have both in the same code (choose which to compile)

So, to repeat, OpenACC or OpenMP?

- **OpenMP is likely to be more widely supported**
  - Wider range of target architectures (accelerators) and of compilers
  - Now leading the accelerator directive feature discussions
- **OpenMP is maturing rapidly**
  - CCE support 4.0 since Sep. 2015,
  - CCE support 4.5 since Jun. 2016 (few exceptions) (fully starting with CCE8.6)
  - 4.5 bridges the functionality gap to OpenACC
- **Advice on getting started**
  - First understand data locality and parallelism

A simple example

- **One simple directive**
  - OpenMP cares about data handling and scheduling



A First Program

- First loopnest initialises array
- Second loopnest modifies array

- Each loopnest should become a kernel
  - Executed on accelerator
  - Loop iterations "partitioned" over threads on the accelerator

- Separating kernels makes a barrier
  - Only way to get global sync. (with GPUs)

```
PROGRAM main

  <stuff>


! Start kernel
  DO i = 1,N
   a(i) = i
  ENDDO
! End kernel

! Start kernel
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
! End kernel


  <stuff>

END PROGRAM main
```

OpenMP

```
PROGRAM main

  <stuff>


!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target teams distribute
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute


  <stuff>

END PROGRAM main
```

- Each loopnest is target region
- teams creates threads
  - divided into a "league of teams"
  - Like CUDA "grid of threadblocks"
- distribute partitions loop
  - iterations divided over threads

OpenMP with C/C++

```
int main() {

  <stuff>


#pragma omp target teams distribute
  for {int i=0; i<N; i++) {
    a[i] = i;
  }


#pragma omp target teams distribute
  for {int i=0; i<N; i++) {
    a[i] = 2*a[i];
  }



  <stuff>

}
```

- No end directive needed
  - pragma applies to structured block

```
PROGRAM main

  <stuff>


!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop

!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop


  <stuff>

END PROGRAM main
```

OpenACC

- Each loopnest is parallel region
- parallel creates threads
  - divided into a "gang of workers/vectors"
  - Like CUDA "grid of threadblocks"
- loop partitions loop
  - iterations divided over threads

OpenACC with C/C++

```
int main() {

  <stuff>

#pragma acc parallel loop
  for {int i=0; i<N; i++) {
    a[i] = i;
  }


#pragma acc parallel loop
  for {int i=0; i<N; i++) {
    a[i] = 2*a[i];
  }

  <stuff>

}
```

- No end directive needed
  - pragma applies to structured block

Data scoping - Motivation

- **Now regarding new code execution**
  - in parallel, rather than serial
  - on accelerator, in separate memory
- **Must ensure that code executes correctly**
  - data is appropriately shared (or not)
    - `i` is different for each loop iteration
    - `a` should be shared between loop iterations
  - data is appropriately synchronised between memory spaces
    - First kernel: `a` is used in write-only fashion
    - Second kernel: `a` is used in read-write fashion
- **Data "scoping" ensures this**

```
PROGRAM main

  <stuff>

! Threaded part
! Start kernel
for (i=0; i<N; i++) {
   t = a[i];
   t++;
   b[i] = 2*t;
}
! End kernel
! End threaded part

  <stuff>

END PROGRAM main
```

Data scoping

- **Codes process data, using other data to do this**
  - all this data is held in structures, such as arrays or scalars

- **In a serial code (or pure MPI), there are no complications**
- **In a thread-parallel code, things are more complicated:**
  - Some data will be the same for each thread (e.g. the main data array)
    - The threads can (and usually should) share a single copy of this data
  - Some data will differ between threads (e.g. loop index values)
    - Each thread will need it's own private copy of this data

- **Data scoping ensures get same answer in parallel as in serial.**
  - It is done: automatically (by the compiler) or explicitly (by the programmer)

- **If the data scoping is incorrect, we get:**
  - incorrect (and inconsistent) answers ("race conditions"), and/or
  - a memory footprint that is too large to run

Understanding data scoping

- **Variables are declared to be shared or private**
  - shared
    - all loop iterations process the same version of the variable
    - variable could be a scalar or an array
    - **a** and **b** are shared arrays in this example
  - private
    - each loop iteration uses the variable separately
    - again, variable could be a scalar or an array
    - **t** is a private scalar in this example
    - loop index variables (like **i**) are also private
  - firstprivate: a variation on private
    - each thread's copy set to same initial value
    - loop limits (like **N**) should be firstprivate

```
for (i=0; i<N; i++) {
   t = a[i];
   t++;
   b[i] = 2*t;
}
```

Reduction variables

- **Reduction variables are a special case of shared variables**
  - where we will need to combine values across loop iterations
  - e.g. sum, max, min, logical-and etc. acting on a shared array

- **We need to tell the compiler to treat these appropriately**
  - In OpenMP-host, use the reduction clause on parallel region
  - Examples:
    - sum: use clause reduction(+:t)
      - Note sum could involve adding and/or subtracting
    - max: use clause reduction(max:u)

```
DO i = 1,N
   t = t + a(i) - b(i)
   u = MAX(u,a(i))
ENDDO
```

Data scoping with accelerators

- **Data clauses are used to express the scoping**
  - Applied to parallel regions and/or data regions
- **In OpenMP-host, we have exactly these data clauses**
  - shared, private, firstprivate, reduction
- **With accelerators, shared variables are more complicated**
  - we also need to think about data movements to/from accelerator
- **Sub-classify shared variables by how data is used on accelerator**
  - read-only: data need only be copied to accelerator at start
  - write-only: data need only be copied to accelerator at end
  - read-write: data should be copied to/from accelerator at start/end
  - scratch: no data needs to be copied

Data clauses in OpenMP-device

- **private variable clauses**
  - private
  - firstprivate
  - reduction
- **shared variable clauses:**
  - read-only: map(to:)
  - write-only: map(from:)
  - read-write: map(tofrom:)
  - scratch: map(alloc:)
- **Default scoping rules (applied by compiler)**
  - arrays: shared (compiler automatically subclassifies)
  - loop variables/limits: private or firstprivate
  - scalars: shared (4.0) / firstprivate (4.5)

```
PROGRAM main

  <stuff>


!$omp target teams distribute &
!$omp    map(from:a)
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target teams distribute &
!$omp    map(tofrom:a)
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute


  <stuff>

END PROGRAM main
```

OpenMP

- Compiler does automatic scoping
  - of variables used in kernels
  - see compiler feedback for the results
- We can also use clauses explicitly for one or more variables
  - to replicate the compiler's choices, or
  - to over-ride the compiler's choices
- Unspecified variables continue to be scoped automatically

Data clauses in OpenACC

- **private variable clauses**
  - private
  - firstprivate
  - reduction

- **shared variable clauses:**
  - read-only: pcopyin
  - write-only: pcopyout
  - read-write: pcopy
  - scratch: pcreate

- **Default scoping rules (applied by the compiler)**
  - arrays: shared (compiler automatically subclassifies)
  - loop variables/limits: private or firstprivate
  - scalars: private
    - ATTENTION: this is different from OpenMP!

```
PROGRAM main

  <stuff>


!$acc parallel loop &
!$acc    pcopyout(a)
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop

!$acc parallel loop &
!$acc    pcopy(a)
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop


  <stuff>

END PROGRAM main
```

OpenACC

- Compiler does automatic scoping
  - of variables used in kernels
  - see compiler feedback for the results
- We can also use explicit clauses for one or more variables
  - to replicate the compiler's choices, or
  - to over-ride the compiler's choices
- Unspecified variables continue to be scoped automatically

Data regions

- **Data is sloshing to/from accelerator between kernels**
  - that is, needless movement of an array `a(:)`
- **Want to define larger data regions**
  - A lexical code region
  - Data remains on accelerator for entire region
  - Used by multiple enclosed kernels
  - Can also span host code
- **Two copies of arrays inside data region**
  - One on host
  - One on accelerator
- **Copies of arrays independent**
  - Synchronise at data region boundaries
    - as specified by user using data clauses
  - No automatic synchronisation of copies within data region
- **Unspecified data still moved by each kernel**
  - No automatic scoping for data regions
- **Data regions can also be nested**

```
PROGRAM main

  <stuff>
!$omp target data map(from:a)

!$omp target teams distribute &
!$omp    map(from:a)
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target teams distribute &
!$omp    map(tofrom:a)
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute

!$omp end target data
  <stuff>

END PROGRAM main
```

OpenMP

- Perspective of data region
  - a(:) is used in write-only fashion
  - write-only + read-write = write-only
- Enclosed kernels have data clauses
  - explicit or implicit
- Data clauses all have "present test"
- First check if specified data already on accelerator (because of outer data region)
  - yes: use data already there
  - no: move data as specified by clause
- So you can leave data clauses on kernels

```
PROGRAM main

  <stuff>
!$acc data pcopyout(a)

!$acc parallel loop &
!$acc    pcopyout(a)
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop

!$acc parallel loop &
!$acc    pcopy(a)
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop

!$acc end data
  <stuff>

END PROGRAM main
```

OpenACC

- Perspective of data region
  - a(:) is used in write-only fashion
  - write-only + read-write = write-only
- Enclosed kernels have data clauses
  - explicit or implicit
- Data clauses all have "present test"
- First check if specified data already on accelerator (because of outer data region)
  - yes: use data already there
  - no: move data as specified by clause
- So you can leave data clauses on kernels

```
PROGRAM main

  <stuff>
! Start data region, a used write-only

! Start kernel
  DO i = 1,N
   a(i) = i
  ENDDO
! End kernel

! Update a to host, e.g. to print checksum
  <stuff>

! Start kernel
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
! End kernel

! End data region
  <stuff>

END PROGRAM main
```

data synchronisation

- Data only synchronised between host, device memory at start/end of:
  - explicit data region in code
    - according to data clauses
  - implicit data region of a kernel
    - according to explicit or implicit data clauses
- Can add additional sync. points in data regions using update directives
  - e.g. for I/O (incl. debugging), or
  - for communication (e.g. MPI)

```
PROGRAM main

 <stuff>
!$omp target data map(from:a)
!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute

!$omp target update from(a)
  PRINT *,SUM(a)

!$omp target teams distribute
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$omp end target teams distribute
!$omp end target data
 <stuff>

END PROGRAM main
```

OpenMP

- target update directive
- clauses specify direction
  - to: update to device from host
  - from: update from device to host

```
PROGRAM main

<stuff>

!$acc data pcopyout(a)
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop

!$acc update self(a)
  PRINT *,SUM(a)

!$acc parallel loop
  DO i = 1,N
   a(i) = 2*a(i)
  ENDDO
!$acc end parallel loop
!$acc end data

<stuff>

END PROGRAM main
```

OpenACC

- update directive
- clauses specify direction
  - self: from device to host
  - device: from host to device
  - host: same as self (deprecated)

Array sections

- **Data clauses and update clauses**
  - take list of arrays or array sections
- **Array sections specified using ":" notation**
  - Fortran: start:end
    - 1:N transfers the first N elements of the array
  - C/C++: start:length
    - 0:N also transfers the first N elements of the array (NOT 0:N-1)
- **Advice: be careful and don't make mistakes!**
  - Use profiler, runtime commentary to measure data moved
  - Avoid non-contiguous array slices for performance

Unstructured data regions

- **Data regions so far must start and end in same routine**
  - and data to be held on accelerator must be in scope
- **This does not work for more modular/OO codes**
  - data structures may have creator/destructor routines or methods
- **Requires a more flexible data region**
  - ability to separate the start/end directives

Unstructured data region clauses

- **Start/end directives take different data clauses:**
  - Need to split the synchronisation tasks
    - e.g. read-write variable is read at the start and write at the end
  - start:
    - possibly allocate space on device memory
    - possibly copy date to device from host
  - end:
    - possible copy data from device to host
    - possibly free space on device memory
  - only do these if a "present test" for the variable fails

Unstructured data regions in OpenMP (v4.5)

- **target enter data**
  - map(to:) allocates and copies from host to device
  - map(alloc:) allocates space on accelerator without copy
- **target exit data**
  - map(from:) copies from device to host and frees
  - map(release:) frees space on accelerator without copy

Unstructured data regions in OpenACC

- **enter data**
  - pcopyin allocates and copies from host to device
  - pcreate allocates space on accelerator without copy
- **exit data**
  - copyout copies from device to host and frees
  - delete frees space on accelerator without copy
- **Note: clauses depend on OpenACC version**

Sharing GPU Data Between Subprograms

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$omp target data map(from:a)
!$omp target teams distribute
  DO i = 1,N
   a(i) = i
  ENDDO
!$omp end target teams distribute
  CALL double_array(a)
!$omp end target data
  <stuff>
END PROGRAM main
```

```
SUBROUTINE double_array(b)
  INTEGER :: b(N)
!$omp target teams distribute map(tofrom:b)
  DO i = 1,N
   b(i) = double_scalar(b(i))
  ENDDO
!$omp end target teams distribute
END SUBROUTINE double_array
```

```
INTEGER FUNCTION double_scalar(c)
  INTEGER :: c
  double_scalar = 2*c
END FUNCTION double_scalar
```

data clause optional

- Real applications have a call tree, **which we must preserve**
- One of the kernels now in subroutine (maybe in separate file)
  - Compiler supports function calls inside kernels
- Array `b(:)` will be scoped as read-write [automatically or explicitly]
  - data clause includes present test, so uses data already placed on accelerator by data region

Sharing GPU Data Between Subprograms

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data pcopyout(a)
!$acc parallel loop
  DO i = 1,N
   a(i) = i
  ENDDO
!$acc end parallel loop
  CALL double_array(a)
!$acc end data
  <stuff>
END PROGRAM main
```

```
SUBROUTINE double_array(b)
  INTEGER :: b(N)
!$acc parallel loop pcopy(b)
  DO i = 1,N
   b(i) = double_scalar(b(i))
  ENDDO
!$acc end parallel loop
END SUBROUTINE double_array
```

```
INTEGER FUNCTION double_scalar(c)
  INTEGER :: c
  double_scalar = 2*c
END FUNCTION double_scalar
```

- Real applications have a call tree, **which we must preserve**
- One of the kernels now in subroutine (maybe in separate file)
  - Compiler supports function calls inside kernels
- Array `b(:)` will be scoped as read-write [automatically or explicitly]
  - data clause includes present test, so uses data already placed on accelerator by data region

Exposing device memory pointers

- **May want to separately process device data, e.g.**
  - use an optimised CUDA kernel
  - use a GPU library (e.g. cuBLAS, cuFFT)
  - use GPUdirect communication libraries(e.g. G2G MPI)
- **Requires address of data in device memory**
- **This is not usually exposed**
  - all addresses are in host memory
  - If data is also on the accelerator, the runtime knows the mapping to the corresponding address in device memory
- **A special directive exposes the device address**

CUDA Interoperability in OpenMP (v4.5)

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$omp target data map(from:a)
! <Populate a(:) on device as before>
!$omp target data use_device_ptr(a)
  CALL dbl_cuda(a)
!$omp end target data
!$omp end target data
  <stuff>
END PROGRAM main
```

```
__global__ void dbl_knl(int *c) {
  int i = \
       blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}
```

- **use_device_ptr exposes accelerator memory address**
  - within its own inner data region (nested inside outer region)
- **CUDA-C wrapper compiled with nvcc linked with CCE**
  - Must include cudaThreadSynchronize() before and after
  - CUDA kernel written as usual
  - Can use same method to call existing CUDA library or G2G-enabled MPI

CUDA Interoperability in OpenACC

```
PROGRAM main
  INTEGER :: a(N)
  <stuff>
!$acc data pcopyout(a)
! <Populate a(:) on device as before>
!$acc host_data use_device(a)
  CALL dbl_cuda(a)
!$acc end host_data
!$acc end data
  <stuff>
END PROGRAM main
```

```
__global__ void dbl_knl(int *c) {
  int i = \
       blockIdx.x*blockDim.x+threadIdx.x;
  if (i < N) c[i] *= 2;
}

extern "C" void dbl_cuda_(int *b_d) {
  cudaThreadSynchronize();
  dbl_knl<<<NBLOCKS,BSIZE>>>(b_d);
  cudaThreadSynchronize();
}
```

- **use_device exposes accelerator memory address**
  - within its own, special host_data region (nested inside outer region)
- **CUDA-C wrapper compiled with nvcc linked with CCE**
  - Must include cudaThreadSynchronize() before and after
  - CUDA kernel written as usual
  - Can use same method to call existing CUDA library or G2G-enabled MPI

Directive comparison

Accelerator Compute Constructs

OpenACC	OpenMP
!$acc parallel	!$omp target teams
!$acc parallel num_gangs(1)	!$omp target
	!$omp teams
!$acc loop gang	!$omp distribute
!$acc loop worker	!$omp parallel do
!$acc loop vector	!$omp simd
!$acc kernels	

Structured Data Constructs

OpenACC	OpenMP
!$acc data	!$omp target data
create(<list>) copyin(<list>) copyout(<list>) copy(<list>)	
present(<list>)	map(<list>)
present_or_create(<list>) present_or_copyin(<list>) present_or_copyout(<list>) present_or_copy(<list>)	map(alloc:<list>) map(to:<list>) map(from:<list>) map([tofrom:]<list>)
!$acc update self(<list>)	!$omp target update from(<list>)
!$acc update device(<list>)	!$omp target update to(<list>)
!$acc cache	

Unstructured Data Constructs

OpenACC	OpenMP
`!$acc enter data` `!$acc exit data`	`!$omp target enter data` `!$omp target exit data`
`delete(<list>)`	`map(delete:<list>)`
	`map(release:<list>)`
`deviceptr(<list>)`	`is_device_ptr(<list>)`
`!$acc host_data use_device(<list>)`	`!$omp target data use_device_ptr(<list>)`
`!$acc data pcreate(<list>) +` `!$acc update device/self(<list>)`	`map(always:<list>)`

- **Scalars are implicitly firstprivate in OpenMP 4.5**
  - `defaultmap` clause restores OpenMP 4.0 behavior

Separate Compilation

OpenACC	OpenMP
!$acc routine !$acc routine(<name>)	!$omp declare target !$omp declare target(<name>)
!$acc declare create(<list>)	!$omp declare target(<list>)
!$acc declare device_resident	
!$acc declare link(<list>)	!$omp declare target link(<list>)

Asynchronous Constructs

OpenACC	OpenMP
`!$acc parallel async(…) wait(…)`	`!$omp target nowait depend(…)`
`!$acc update async(…) wait(…)`	`!$omp target update nowait depend(…)`
`!$acc enter data async(…) wait(…)` `!$acc exit data async(…) wait(…)`	`!$omp target enter data nowait depend(…)` `!$omp target exit data nowait depend(…)`
`!$acc wait`	`!$omp taskwait OR !$omp end taskgroup`
`!$acc wait(…)`	`!$omp task if(.false.) depend(…)`
`!$acc wait(…) async(…)`	`!$omp target depend(…) nowait`

- **OpenACC model: async "streams" are serialized**
  - Use different streams to express independence
- **OpenMP model: nowait "tasks" are independent**
  - Use task dependences to express dependence

Accelerator Programming Strategy

1. **Offload time-intensive parallel loops**
   - Focus on functional correctness
   - Rely on “point-of-use” data transfers
2. **Optimize kernel computation**
   - Temporarily ignore data transfer overheads
3. **Optimize data transfers**
   - Trace transfers with env var CRAY_ACC_DEBUG=2
   - Add enclosing data regions and move up the call chain
   - Add updates where necessary
4. **Use device asynchronously**
   - Fill device “queue” with a “stream” of dependent work
   - Hide latency of data transfers
   - Execute multiple kernels in parallel

Conclusions

- Use directives specify accelerator task
- Let the compiler do the main porting tasks
- Easy to port OpenACC <-> OpenMP device constructs
- Main tasks is checking and defining data locality