Programming paradigms for GPU devices

PTC course

@CINECA

15-17 April 2020

Sergio Orlandini

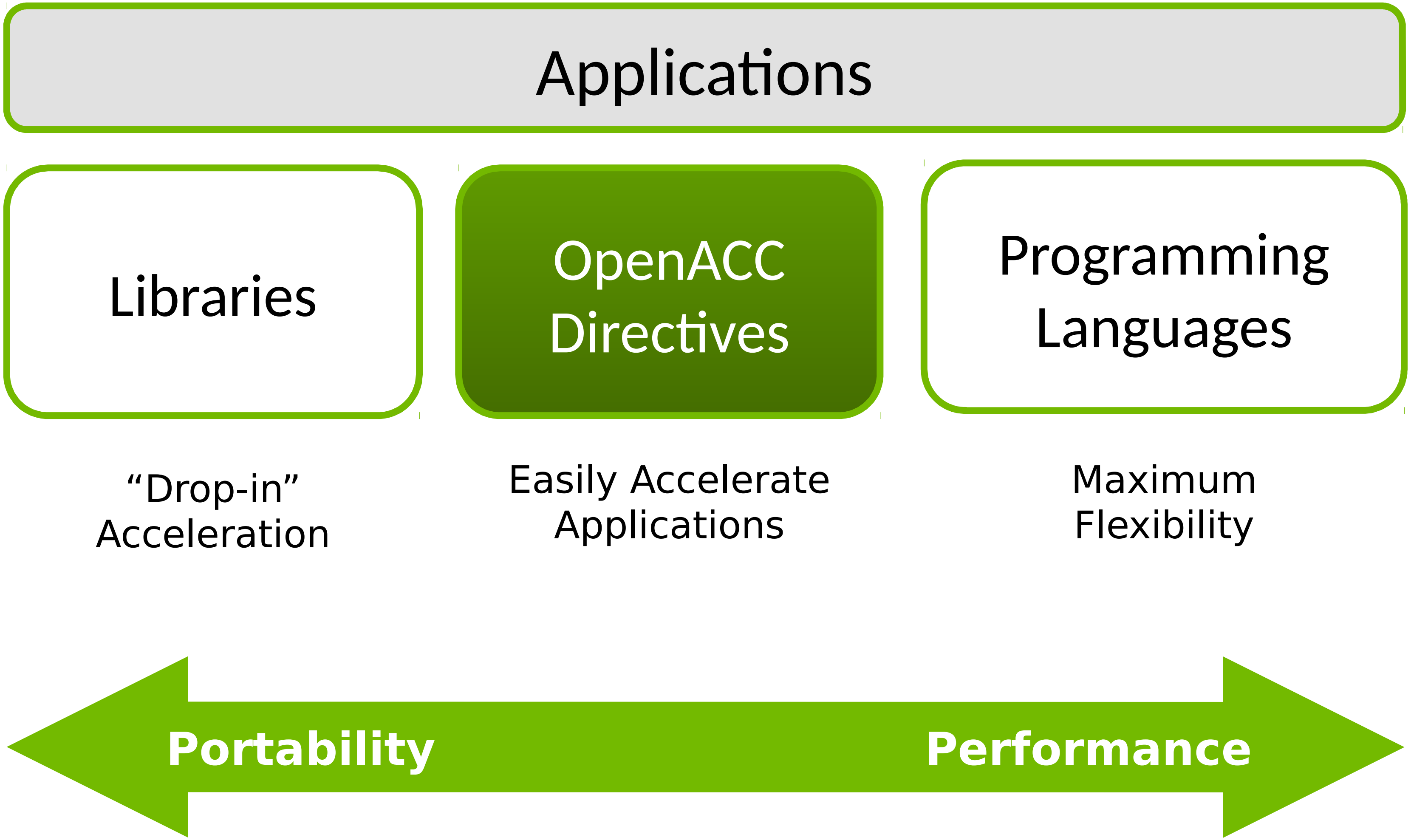
s.orlandini@cineca.it

Luca Ferraro

l.ferraro@cineca.it

- OpenACC introduction
  - *express parallelism*
  - *optimize data movements*
  - *practical examples*

3 Ways to Accelerate Applications



Applications

Libraries

OpenACC Directives

Programming Languages

"Drop-in" Acceleration

Easily Accelerate Applications

Maximum Flexibility

Portability

Performance

OpenACC Friendly Disclaimer

OpenACC Directives

Easily Accelerate Applications

OpenACC does not make GPU programming easy. (...)

GPU programming and parallel programming is not easy. It cannot be made easy. However, GPU programming need not be difficult, and certainly can be made straightforward, once you know how to program and know enough about the GPU architecture to optimize your algorithms and data structures to make effective use of the GPU for computing. OpenACC is designed to fill that role.

(Michael Wolfe, The Portland Group)

OpenACC History

- OpenACC is a high-level specification with **compiler directives** for expressing parallelism for accelerators
  - **Portable** to a wide range of accelerators
  - One specification for **Multiple Vendors** and **Multiple Devices**
- OpenACC specification was released in November 2011.
  - Original members: **CAPS**, **Cray**, **Nvidia**, **Portland Group**
- OpenACC 2.0 was released in June 2013
  - More functionality
  - Improve portability
- OpenACC 2.5 in November 2015
- OpenACC 2.7 in November 2018
- OpenACC had more than 10 member organizations

OpenACC Info & Vendors

- **http://www.openacc.org**
- Novelty in OpenACC 2.0 are significant
  - OpenACC 1.0 maybe not very mature…
- Some changes are inspired by the development of CUDA programming model
  - but the standard is not limited to NVIDIA GPUs: one of its pros is the **interoperability** between platforms
- Standard implementation
  - CRAY provides full OpenACC 2.0 support in CCE 8.2
  - PGI support to OpenACC 2.5 is almost complete (starting from version 15.1)
    - Suppurt for OpanACC 2.0 starting from 14.1
  - GNU implementation effort ongoing
    - partial implementation in the 5.1 release
    - a dedicated branch for 7.1 realease for implementation of the OpenACC 2.0
    - Complete support of OpenACC 2.5 from 9,1 release
- We will focus on **PGI** compiler
  - all-in-one compiler, easy usage
  - sometimes the compiler tries to help you…
  - but also a constraint on the compiler to use

Directive Based Approach

- **Directives** are added to serial source code
  - Manage loop parallelization
  - Manage data transfer between CPU and GPU memory

- Directives are formatted as **comments**
  - They don't interfere with serial execution

- Maintaines **portability** of original code

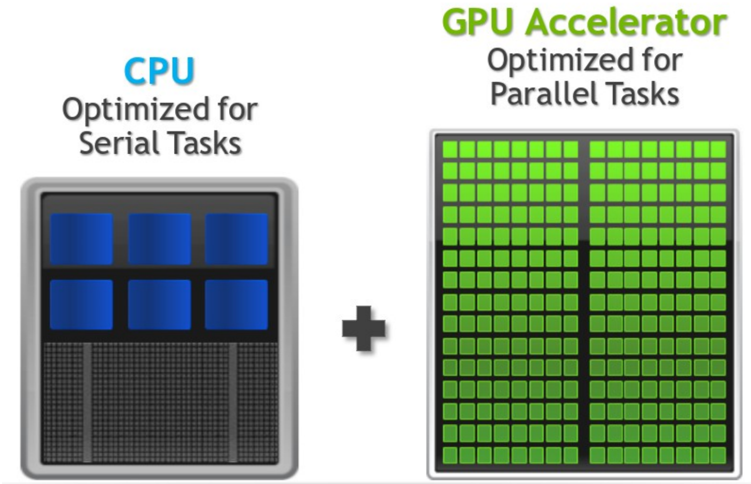
- Works with C/C++ or Fortran
  - Can be combined with explicit CUDA C/Fortran usage

OpenACC – Simple, Powerful, Portable

```
main()
{

   <serial code>

   #pragma acc kernels
   //automatically runs on GPU
   {
      <parallel code>
   }
}
```



1. **Simple**:
   - Simple compiler **directives**
   - Directives are the easy path to accelerate compute intensive applications
   - Compiler parallelizes code
2. **Open**:
   - OpenACC is an open GPU directives **standard**, making GPU programming straightforward and portable across parallel and multi-core processors
3. **Portable**:
   - Works on many-core GPUs and multi-core CPUs
4. **Powerful**:
   - GPU Directives allow complete access to the massive parallel power of a GPU

Directive Syntax

- C/C++

  ```
  #pragma acc directive [clause [,] clause] …]
  ```

  Often followed by a structured code block

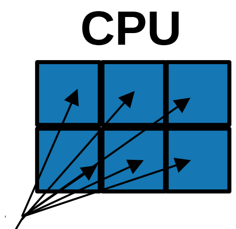
- Fortran

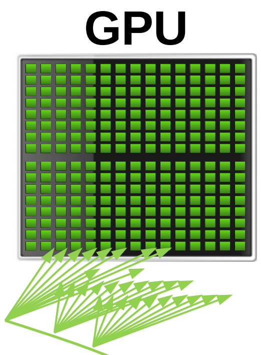
  ```
  !$acc directive [clause [,] clause] …]
  ```

  Often paired with a matching end directive surrounding a structured code block

  ```
  !$acc end directive
  ```

Familiar to OpenMP Programmers





OpenMP

```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf(“pi = %f\n”, pi/N);
}
```

OpenACC

```
main() {
  double pi = 0.0; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

printf(“pi = %f\n”, pi/N);
}
```

OpenMP 4.0/4.5 alternative

- **OpenMP** 4.0/4.5 supports heterogeneous systems (accelerators/devices)
- What's new in OpenMP 4.x for support accelerator model
  - **Target regions**
    - Structured and unstructured target data regions
      - `omp target [clause[[,] clause],…]`
      - `omp declare target`
    - **Asynchronous** execution (`nowait`) and **data dependency** (`depend`)
  - Manage device data environment
    - **Data mapping** APIs
      - `map ([map-type:] list)`
    - **Data regions**
      - `omp target data [clause[[,] clause], …]`
      - `omp target enter/exit data [clause[[,] clause], …]`
  - **Parallelism & Workshare** for devices
    - `omp teams [clause[[,] clause],…]`
    - `omp distribute [clause[[,] clause],…]`
  - **SIMD** parallelism

OpenMP 4.0/4.5 alternative

```
main()
{

   <serial code>

   #pragma omp target map(to:u) map(from:v)
   #pragma omp parallel for collapse(2)
   for ( i = 0; i < NUM_I; i++ ) {
      for ( j = 0; j < NUM_J; j++ ) {
         v[i][j] = u[j][i+1] + u[j][i-1] + u[j-1][i] + u[j+1][i]);
      }
   }

   <serial code>

}
```

Porting to OpenACC

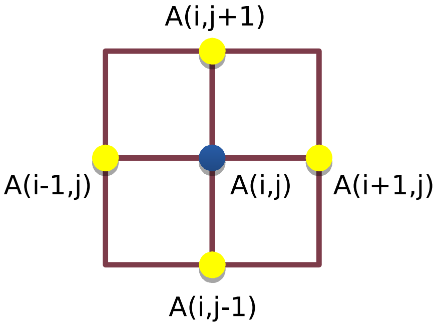
1. **Identify** available parallelism
2. Express **parallelism**
3. Express **data movement**
4. **Optimize** loop performance

Example: Jacobi Iteration

- Jacobi method is an iterative method to solve a system of linear equation
- It is a very common and useful algorithm
- Example: Jacobi method can be used to solve the Laplace differential equation in two variables (2D)
- The Laplace equation models the steady state of a function ***f*** defined in a physical 2D space where ***f*** is a physical quantity (e.g. Temperature)

$$\nabla^2 f(x,y) = 0$$

- Iteratively converges to correct value by computing new values at each point from the average of neighboring points



$$A_{k+1}(i,j) = \frac{A_k(i-1,j) + A_k(i+1,j) + A_k(i,j-1) + A_k(i,j+1)}{4}$$

Jacobi Iteration: C/C++ Code

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

1. Iterate until converged
2. Iterate across matrix elements
3. Calculate new value from neighbors
4. Compute max error for convergence
5. Swap input/output arrays

Jacobi Iteration: Fortran Code

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

  do j=1,m
    do i=1,n

     Anew(i,j) = .25 * (A(i+1,j  ) + A(i-1,j  ) + &
                        A(i  ,j-1) + A(i  ,j+1))


      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do

  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do

  iter = iter +1
end do
```

1. Iterate until converged
2. Iterate across matrix elements
3. Calculate new value from neighbors
4. Compute max error for convergence
5. Swap input/output arrays

1. Identify parallelism

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Data dependency between iterations

Independent loop iterations

Independent loop iterations

`parallel` construct

- **Programmer identifies** a block of code suitable for parallelization
- and guarantees that **no dependency** occurs across iterations
- Compiler generates parallel instructions for that loop
  - e.g., a parallel CUDA kernel for a GPU

```
#pragma acc parallel loop
for (int j=1;j<n-1;j++) {
   for (int i=1;i<n-1;i++) {
       A[j][i] = B[j][i] + C[j][i]
   }
}
```

`kernels` construct

- The **`kernels`** construct expresses that a region **may** contain parallelism and the **compiler** determines what can be safely parallelized

C/C++

```
#pragma acc kernels [clause …]
   {structured block}
```

```
!$acc kernels
  do i=1,n
     a(i) = 0.0
     b(i) = 1.0
     c(i) = 2.0
  end do

  do i=1,n
     a(i) = b(i) + c(i)
  end do
!$acc end kernels
```

kernel 1

kernel 2

- The compiler identifies 2 parallel loops and generate 2 kernels

parallel vs kernels

parallel

- Requires analysis by **programmer** to ensure safe parallelism
- Straightforward path from **OpenMP**
- Mandatory to fully control the different levels of parallelism
- Implicit barrier at the end of the parallel region

kernels

- **Compiler** performs parallel analysis and parallelizes what it believes safe
- Can cover **larger** area of code with a single directive
- Please, write clean codes and add directives to help the compiler
- Implicit barrier at the end and between each loop

C tip: the restrict keyword

- Declaration of intent given by the programmer to the compiler

  Applied to a pointer, e.g.

  ```
  float *restrict ptr
  ```

  Meaning: “for the lifetime of ptr, only it or a value directly derived from it (such as ptr + 1) will be used to access the object to which it points”
- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence between the iterations of a loop
  - Crucial when adopting `kernels` directive, but also for other optimizations
  - Note: if the programmer violates the declaration, the behavior is undefined

SAXPY example code

- Use restrict to help the compiler when adopting **kernels**
  - Apply a **loop** directive
- Be careful: **restrict** is C99 but not C++ standard

```
#include <stdlib.h>

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

```
int main(int argc, char **argv)
{
  int N = 1<<20; // 1 million floats

  if (argc > 1)
    N = atoi(argv[1]);

  float *x = (float*)malloc(N*sizeof(float));
  float *y = (float*)malloc(N*sizeof(float));

  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }

  saxpy(N, 3.0f, x, y);

  return 0;
}
```

restrict
“I promise y does not alias x“

`loop` construct

- Applies to a loop which must immediately follow this directive
- Describes:
  - type of parallelism
  - loop-private variables, arrays, and reduction operations
- We already encountered it combined with the parallel directive
  - combining kernels and loop is also possible but limits the capability of kernels construct (i.e. extending to wide regions of code)

C/C++

```
#pragma acc loop [clause …]
    { for block }
```

Fortran

```
!$acc loop [clause …]
    { do block }
```

independent clause

- In a **kernels** construct, the **independent loop** clause helps the compiler in guaranteeing that the iterations of the loop are independent with each other
- E.g., consider m>n

```
#pragma acc kernels
#pragma acc loop independent
for(int i;i<n;i++)
    c[i] = 2.*c[m+i];
```

- In **parallel** construct the independent clause is implied on all **loop** directives without a **seq** clause

seq and collapse

- The **seq** clause specifies that the associated loops have to be executed sequentially on the accelerator
- Beware: the **loop** directive applies to the immediately following loop

```
#pragma acc parallel
#pragma acc loop collapse(2) // independent is automatically
                             // enforced
for(int i;i<n;i++)
for(int k;k<n;k++)
#pragma acc loop seq
for(int j;j<n;j++)
       c[i][j][k] = 2.*c[i][j+1][k];
```

- **collapse(<n_loops>)** clause allows for extending **loop** to tightly nested loops
  - but the compiler may decide to collapse loops anyway, check the report!

Loop reductions

- The **`reduction`** clause on a **`loop`** specifies a reduction operator on one or more scalar variables
  - For each variable, a private copy is created for each thread executing the associated loops
  - At the end of the loop, the values for each thread are combined using the reduction clause

- Reductions may be defined even at **`parallel`** level (advanced topic)

- Common operators are supported:

```
+   *   max   min   &&   || ....
```

2. Express parallelism

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
{
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
}

  iter++;
}
```

2. Express parallelism

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n

     Anew(i,j) = .25 * (A(i+1,j  ) + A(i-1,j  ) + &
                        A(i  ,j-1) + A(i  ,j+1))


      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do

  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels

  iter = iter +1
end do
```

2. Express parallelism

```
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc parallel loop reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

2. Express parallelism

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc parallel loop reduction(max:error)
  do j=1,m
    do i=1,n

     Anew(i,j) = .25 * (A(i+1,j  ) + A(i-1,j  ) + &
                        A(i  ,j-1) + A(i  ,j+1))


      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do
!$acc end parallel loop

!$acc parallel loop
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end parallel loop

  iter = iter +1
end do
```

Compiling and running (PGI)

```
pgcc -acc -ta=tesla -Minfo=all -acc=noautopar -o laplace2d.x laplace2d.c
```

```
main:
     34, Loop not vectorized: may not be beneficial
         Unrolled inner loop 4 times
         Generated 3 prefetches in scalar loop
     44, Loop not vectorized/parallelized: potential early exits
     48, Generating copyout(Anew[1:4094][1:4094])
         Generating copyin(A[:][:])
         Generating copyout(A[1:4094][1:4094])
     50, Loop is parallelizable
     51, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         50, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         51, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         54, Max reduction generated for error
     58, Loop is parallelizable
     59, Loop is parallelizable
         Accelerator kernel generated
         Generating Tesla code
         58, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         59, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
```

- For the hands-on, compile using the makefile and run by typing

```
$ make pgi
$ ./laplace2d N  (N is the GPU number to use, 0 1 2 ...)
```

- Laplace 2D
  - Compile serial code for reference
  - Accelerate serial code with OpenACC
  - Use **`kernels`** construct
  - Use **`parallel`** construct
  - Performance

Performance

Resources	Time / sec. - PGI
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz
GPU Nvidia Tesla K80
PCI-e 3.0

Performance

Resources	Time / sec. - PGI
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	**9.2**

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz
GPU Nvidia Tesla K80
PCI-e 3.0

What is going wrong?

```
Accelerator Kernel Timing data
laplace2d.c
  main
    69: region entered 1000 times
        time(us): total=77524918 init=240 region=77524678
                  kernels=4422961 data=66464916
        w/o init: total=77524678 max=83398 min=72025 avg=77524
        72: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=4422961 max=4543 min=4345 avg=4422
laplace2d.c
  main
    57: region entered 1000 times
        time(us): total=82135902 init=216 region=82135686
                  kernels=8346306 data=66775717
        w/o init: total=82135686 max=159083 min=76575 avg=82135
        60: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=8201000 max=8297 min=8187 avg=8201
        64: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=145306 max=242 min=143 avg=145
acc_init.c
  acc_init
    29: region entered 1 time
        time(us): init=158248
```

4.4 seconds → kernels=4422961

66.5 seconds → data=66464916

8.3 seconds → kernels=8346306

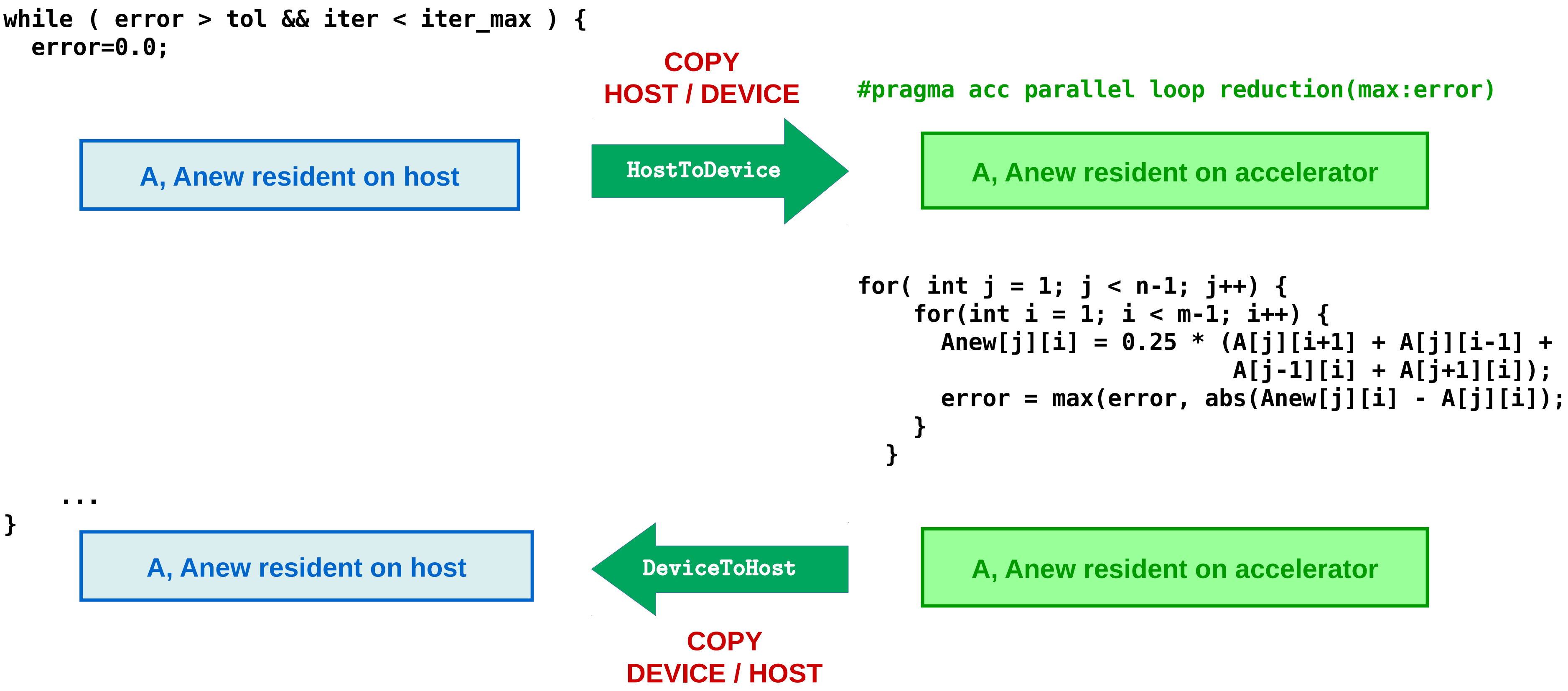
66.8 seconds → data=66775717

Huge Data Transfer Bottleneck!

Computation: 12.7 seconds

Data movement: 133.3 seconds

Excessive data transfers



```
while ( error > tol && iter < iter_max ) {
  error=0.0;
```

COPY
HOST / DEVICE

```
#pragma acc parallel loop reduction(max:error)
```

A, Anew resident on host

HostToDevice

A, Anew resident on accelerator

```
for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {
      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }
```

```
    ...
}
```

A, Anew resident on host

DeviceToHost

A, Anew resident on accelerator

COPY
DEVICE / HOST

These copies are performed every iteration of the while loop.

NB: in case of two **#pragma acc parallel** there are 4 copies per while loop iteration.

data construct

C/C++

```
#pragma acc data [clause …]
    { structured block }
```

Fortran

```
!$acc data [clause …]
    structured block
!$acc end data
```

- Manages explicitely data movements between host and device
- Crucial to handle GPU data persistence
- Allows for decoupling the scope of GPU variables from that of the accelerated regions
- May be nested
- Data clauses define different possible behaviours

`data` clauses

`copy ( list )` Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.

`copyin ( list )` Allocates memory on GPU and copies data from host to GPU when entering region.

`copyout ( list )` Allocates memory on GPU and copies data to the host when exiting region.

`create ( list )` Allocates memory on GPU but does not copy.

`present ( list )` Data is already present on GPU from another containing data region.

and `present_or_copy[in|out], present_or_create, deviceptr`.

Array shaping

- The compiler sometimes cannot determine the sizes of arrays
  - you must specify them by using data clauses and array “shape”
  - you may need just a section of an array
  - sub-array syntax is allowed, in Fortran it is language-native
- C/C++

```
#pragma acc data copyin(a[1:size]), copyout(b[s/4:3*s/4+1])
```

- Fortran

```
 !$pragma acc data copyin(a(1:size)), copyout(b(s/4:s))
```

- Data clauses can be used on `data`, `kernels` or `parallel`

- Laplace 2D
  - Use `data` construct
  - Performance

3. Express data movement

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
{
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }
}

  iter++;
}
```

3. Express data movement

```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n

     Anew(i,j) = .25 * (A(i+1,j  ) + A(i-1,j  ) + &
                        A(i  ,j-1) + A(i  ,j+1))


      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do

  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels

  iter = iter +1
end do
!$acc end data
```

3. Express data movement

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc parallel loop reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma acc parallel loop
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

3. Express data movement

```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc parallel loop reduction(max:error)
  do j=1,m
    do i=1,n

     Anew(i,j) = .25 * (A(i+1,j  ) + A(i-1,j  ) + &
                        A(i  ,j-1) + A(i  ,j+1))

      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do
!$acc end parallel loop

!$acc parallel loop
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end parallel loop
  iter = iter +1
end do
!$acc end data
```

Performance

Resources	Time / sec. - PGI
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz
GPU Nvidia Tesla K80
PCI-e 3.0

Performance

Resources	Time / sec. - PGI
CPU 1 OpenMP thread	22.5
CPU 2 OpenMP threads	11.5
CPU 4 OpenMP threads	6.5
CPU 8 OpenMP threads	3.9
CPU 16 OpenMP threads	2.5
OpenACC 1GPU	**0.6**

2 eight-core Intel(R) Xeon(R) CPU E5-2687W @ 3.10GHz
GPU Nvidia Tesla K80
PCI-e 3.0

4. Optimize loop performance

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc parallel loop collapse(2) reduction(max:error)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma acc parallel loop collapse(2)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

4. Optimize loop performance

```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc parallel loop collapse(2) reduction(max:error)
  do j=1,m
    do i=1,n

     Anew(i,j) = .25 * (A(i+1,j  ) + A(i-1,j  ) + &
                        A(i  ,j-1) + A(i  ,j+1))

      err = max(err, Anew(i,j) - A(i,j))

    end do
  end do
!$acc end parallel loop

!$acc parallel loop collapse(2)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end parallel loop
  iter = iter +1
end do
!$acc end data
```

4. Optimize loop performance

```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels loop gang(32), vector(16)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16), vector(32)
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);
      error = max(error, abs(Anew[j][i] - A[j][i]);

    }
  }

#pragma acc kernels loop gang(16), vector(32)
  for( int j = 1; j < n-1; j++) {
#pragma acc loop
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Rights & Credits

Slides and examples were authored by:

Isabella Baccarelli, Luca Ferraro, Sergio Orlandini