UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Data parallelism: GPU computing

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# OpenACC Directives

# What is OpenACC ?

- The OpenACC Application Programming Interface provides a set of
  - compiler directives (pragmas)
  - library routines and
  - environment variables
- that can be used to write data parallel Fortran, C and C++ programs that run on accelerator devices including GPUs and CPUs
- Similar, in spirit, to OpenMP: it’s an implicit accelerated HPC (not necessarily CUDA/GPU) programming framework

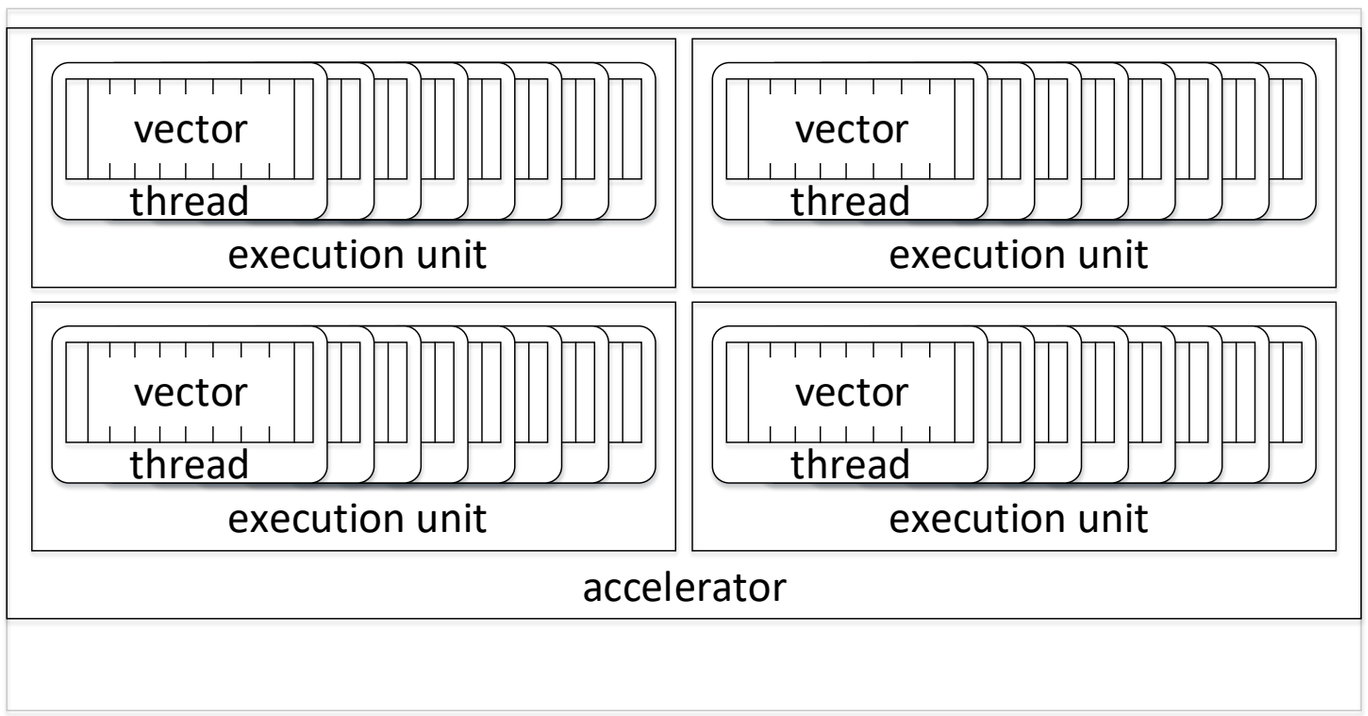
# What is OpenACC ?

- The OpenACC Application Programming Interface provides a set of
  - compiler directives (pragmas)
  - library routines and
  - environment variables

Requires a compiler that can accept the directives.
E.g. PGI compiler provided in NVIDIA HPC package

- that can be used to write data parallel Fortran, C and C++ programs that run on accelerator devices including GPUs and CPUs
- Similar, in spirit, to OpenMP: it's an implicit accelerated HPC (not necessarily CUDA/GPU) programming framework

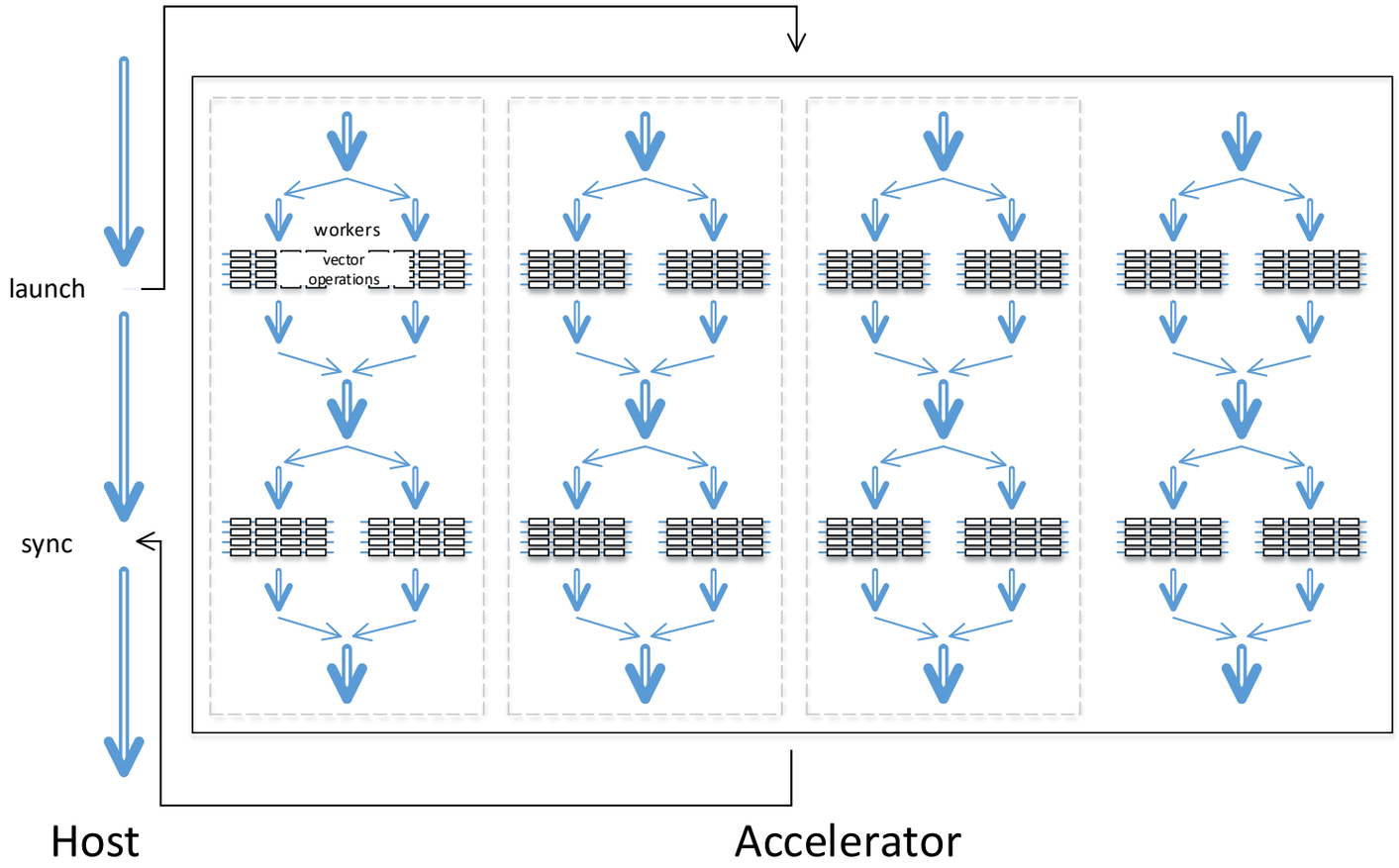
# OpenACC Device Model




- The model deals with accelerators that contain a number of execution units. Threads can execute vector instructions.

- Currently OpenACC does not expose threads synchronization

# OpenACC Execution Model




- The programming model assumes that the program execution will begin on a host CPU which may offload execution and data to an accelerator device.

- Since offloading computation may also require copying of data, which can be time consuming when the host and accelerator have physically separate memories, OpenACC also provides a means for controlling how data is moved between the host and device and how it is shared between different offloaded regions.

# OpenACC parallelism

- OpenACC exposes three levels of parallelism on the accelerator device:
  - **Gangs** are fully independent execution units, where no two gangs may synchronize nor may they exchange data, except through the globally accessible memory. Since gangs work completely independently of each other, the programmer can make no assumptions about the order in which gangs will execute or how many gangs will be executed simultaneously.
    Note the similarity between gangs and CUDA thread blocks.
  - Each gang contains one or more **workers**. Workers have access to a shared cache memory and may be synchronized by the compiler to ensure correct behavior.
    Note the similarity between workers and CUDA threads.
  - Workers operate on **vectors** of work. A vector is an operation that is computed on multiple data elements in the same instruction; the number of elements calculated in the instruction is referred to as the vector length.

# Pragmas (and comments)

- –In C and C++, the `#pragma` directive is the method to provide to the compiler information that is not specified in the standard language.

- These pragmas extend the  base language

- In FORTRAN comments are used instead of `#pragma`

# Pragmas (and comments)

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

  Often paired with a matching end directive surrounding a structured code block
  ```
  !$acc end directive
  ```

- C / C++
  ```
  #pragma acc directive [clause [,] clause] …]
  ```

  Often followed by a structured code block {}

# Motivation

- OpenACC programmers can often start with writing a sequential version and then annotate their sequential program with OpenACC directives.
  - leave most of the details in generating a kernel, memory allocation, and data transfers to the OpenACC compiler.
- OpenACC code can be compiled by non-OpenACC compilers by ignoring the pragmas.

# `kernels` directive

Each loop executed as a separate *kernel* on the GPU. Places the responsibility on the compiler to identify which loops can be safely parallelized and how to do so.

- `#pragma acc kernels [clause …]`
  `{ structured block }`
- Clauses
  - `if( condition )`
  - `async( expression )`
  - any data clause (more later)

# A simple example

```
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];
}



...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

# A simple example

```
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];
}

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
```

- Declaration of intent given by the programmer to the compiler (added in C99)

Applied to a pointer, e.g.

```
float *restrict ptr
```

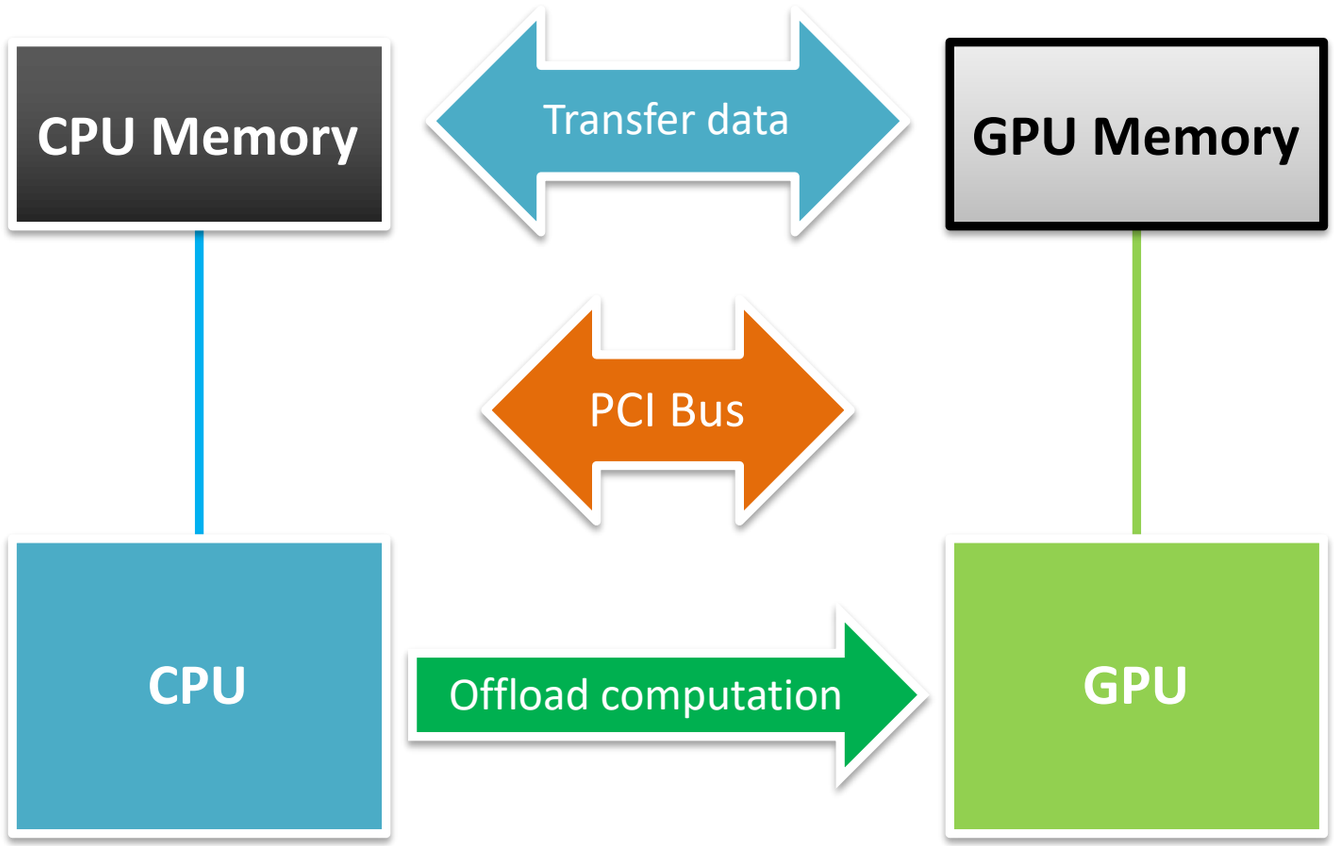
Meaning: "for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points"

- Limits the effects of pointer aliasing
- OpenACC compilers often require restrict to determine independence
  - Otherwise the compiler can't parallelize loops that access `ptr`
  - Note: if programmer violates the declaration, behavior is undefined
  - Note2: FORTRAN does not require it

# Issues

- Some OpenACC pragmas are hints to the OpenACC compiler, which may or may not be able to act accordingly
  - The performance of an OpenACC program depends heavily on the quality of the compiler.
  - It may be hard to figure out why the compiler cannot act according to your hints
- It's very easy to parallelize code and get worst performance than sequential version !

# Basic workflow




- For efficiency, decouple data movement and compute off-load
- Check that data is not copied back and forth CPU and GPU by the automatic implementation obtained using kernels and parallel directives.
  - Use specific data directives to reduce data transfer !

# `parallel` directive

- the `parallel` directive is used by the programmer to ask the compiler to generate parallelism and when combined with the `loop` directive, makes assertions about the feasibility of loops for acceleration without requiring detailed analysis by the compiler.

- Compilers are still required to determine the data requirements for the loops and make decisions about how best to parallelize the loop iterations to the targeted hardware

# `parallel` directive

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

- Clauses
  - `if( condition )`
  - `async( expression )`
  - `num_gangs( expression )` - Controls how many parallel gangs are created (CUDA `gridDim`).
  - `num_workers( expression )` - Controls how many workers are created in each gang (CUDA `blockDim`)
  - `vector_length( expression )` - Controls vector length of each worker (SIMD execution)
  - `private( list )`
  - `firstprivate( list )`
  - `reduction( operator:list )`
  - Also any data clause

# `loop` directive

- Detailed control of the parallel execution of the following loop
- `#pragma acc loop [clause …]`
  `{ loop }`
- Can be combined with `kernels` and `parallel`
- Loop clauses
  - `collapse( n )` - applies directive to the following n nested loops.
  - `seq` - executes the loop sequentially on the GPU.
  - `private( list )` - a copy of each variable in list is created for each iteration of the loop.
  - `reduction( operator:list )` - private variables combined across iterations.
- Loop clauses insieme `parallel` and `kernels` regions allow to share iterations across gangs, workers, and vectors

# `data` directive

- Manage data movement. Data regions may be nested.

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

- General Clauses
  - `if( condition )`
  - `async( expression )`

# `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`.

# `data` clauses

- Compiler sometimes cannot determine size of arrays

- Must specify explicitly using data clauses and array “shape”

- E.g.:

```
#pragma acc data copyin(a[0:size-1]),
    copyout(b[s/4:3*s/4])
```

# update directive

- Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)
- Move data from GPU to host, or host to GPU.
- Data movement can be conditional, and asynchronous.

- #pragma acc update [clause …]
- Clauses
  - host( list )
  - device( list )
  - if( expression )
  - async( expression )

UNIVERSITÀ
DEGLI STUDI
FIRENZE

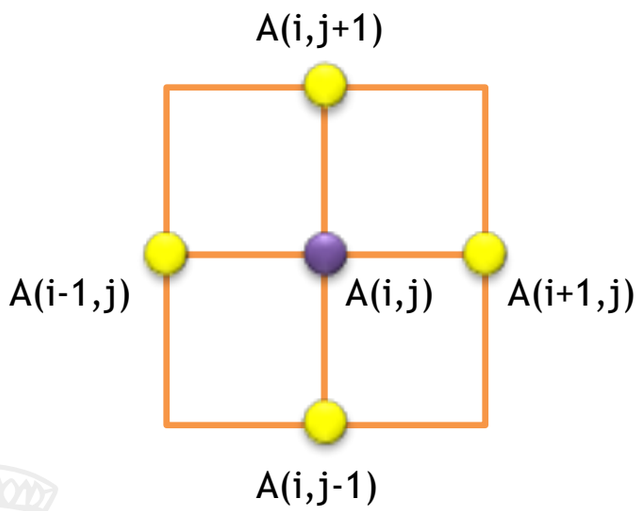
# A full example

# Example: Jacobi Method

- Iteratively converges to correct value (e.g. temperature), by computing new values at each point from the average of neighboring points.

- Common, useful algorithm

- Example: Solve Laplace equation in 2D: $\nabla^2 f(x,y)=0$




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

# Jacobi Iteration C 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++;

}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

# Jacobi Iteration OpenMP Code

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


#pragma omp parallel for shared(m, n, Anew, A)

  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 omp parallel for shared(m, n, Anew, A)

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

  iter++;

}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

# Jacobi Iteration Naïve OpenACC Code

```
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]);

    }
  }


#pragma acc kernels

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

  iter++;

}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

# Naïve OpenACC problem

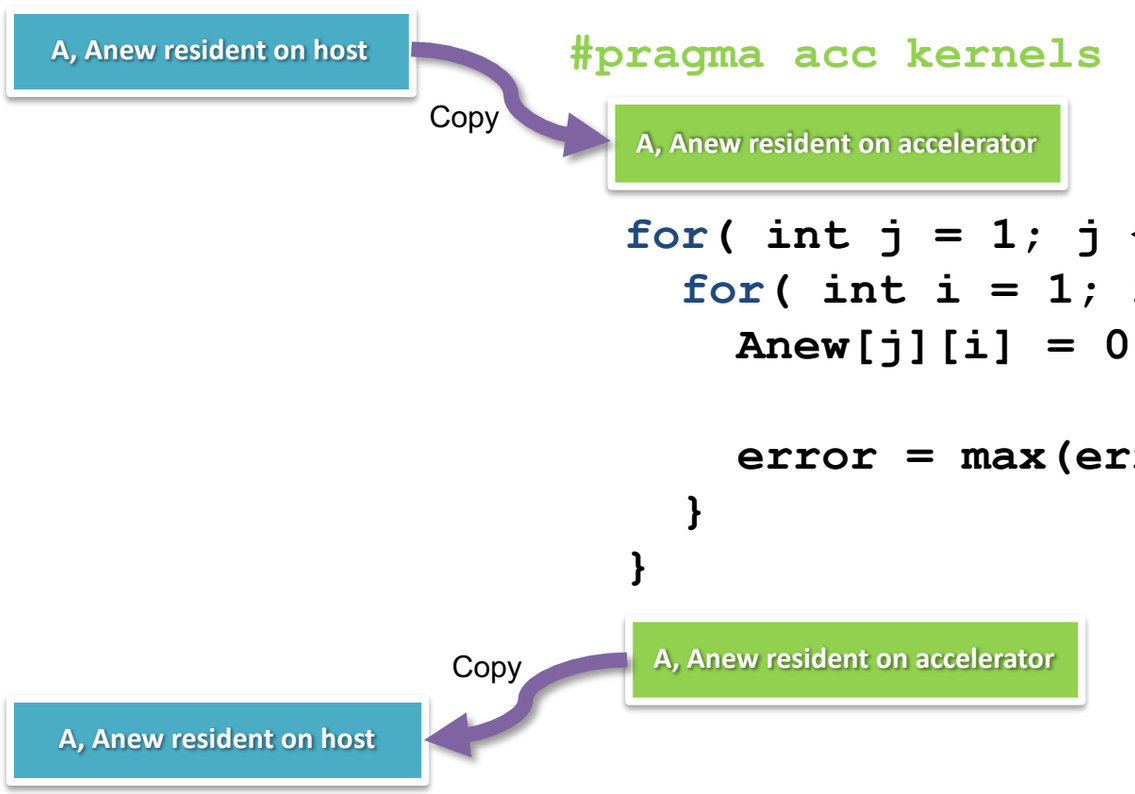
- The OpenACC problem will be much slower than a sequential version !

- Instead, OpenMP will easily get a ~1.5-2× speedup

- The issue is due to excessive data transfer: the OpenACC compiler will automatically copy back A and Anew arrays from host to device back and forth for each `kernels` directive

# Excessive data transfer

```
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]);
  }
}

...
}
```

These copies happen every iteration of the outer while loop!
There are two `#pragma acc kernels`, so there are 4 copies per while loop iteration!

# Jacobi Iteration Better OpenACC Code

```
#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]);

    }
  }

#pragma acc kernels

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

  iter++;

}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

# Jacobi Iteration Better OpenACC Code

```
#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]);

    }
  }

#pragma acc kernels

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

  iter++;

}
```

**Copy A in at beginning of loop, out at end. Allocate Anew on accelerator**

**Iterate until converged**

**Iterate across matrix elements**

**Calculate new value from neighbors**

**Compute max error for convergence**

**Swap input/output arrays**

Managing data transfer allows to greatly improve performance and obtain substantial speedup over OpenMP

# Tricks

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Use `data` regions to avoid excessive memory transfers
- Use profiling, e.g. PGI compiler has specific options to get it
- Iterations of loops must be independent of each other
  - To help compiler: restrict keyword (C), independent clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.

# Credits

- These slides report material from:
  - NVIDIA GPU Teaching Kit
  - M. Harris, NVIDIA

# Books

- Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 2nd edition - Chapt. 15

or

Programming Massively Parallel Processors: A Hands-on Approach, D. B. Kirk and W-M. W. Hwu, Morgan Kaufman - 3rd edition - Chapt. 19