OpenACC:
Directive-based Programming
For
Hardware Accelerators

https://www.openacc.org/

Programming Hardware Accelerators

Scientific Applications

Software Libraries	Compiler directives	Programming Languages
Drop-in Replacement of Relevant Functions	Easy Acceleration of Custom Code	• Flexibility of implementation • Steep learning curve • Maximum performance • Error Prone
• Easy of use • Performance guarantee • Fixed interface	• Portable • Full assistance of the compiler	• Not for beginners

OpenACC in a Nutshell

- Directives in
  - pragmas (C, C++) or
  - comments Fortran
- Multiple parallelism levels
- Portable across
  - Accelerators
    - NVIDIA GPUs
    - Sunway
    - PEZY-SC
    - Including CPU target
  - Compilers
    - GNU GCC 9 (OpenACC 2.6)
    - PGI, …
- Performance
  - Often close to “native” efficiency

```c
int main(void) {
    // host code
    #pragma acc kernels
    {
      // accelerated code
    }
    return 0;
}
```

OpenACC Main Features

- Incremental
  - Sequential code preserved
  - Annotations limited to important parts of the code
    - Profile-based performance tuning
  - Correctness verified as needed without massive code changes
- Single Source
  - Same code for sequential and accelerated code
  - Recompile on a new platform for performance
  - Sequential code unaffected and developed in unison
- Ease of use
  - Gentle learning curve
  - Works for HPC languages: C, C++, Fortran
  - No low-level hardware experience or details required

Jacobi Iteration: C Code

```
while ( err > tol && iter < iter_max ) {
   err=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]);

         err = max(err, 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 convergence

Iterate across matrix elems.

Calculate new value from neighbors

Compute max-error for convergence

Swap input/output arrays

OpenACC Syntax Overview

```c
#pragma acc <directive> <clause_1>
{
  // block of code to accelerate
}
```

```fortran
!$acc <directive> <clause_1>
! block of code to accelerate
!$acc end <directive>
```

- A **pragma** in C and C++
  - Instructs the compiler how to compile the code
  - Compilers may ignore pragma they don't understand
- Specially formatted comment is required in Fortran
  - Comments can be ignored by the compiler
- **acc** starts OpenACC directives and clauses
- **Directives** are commands for OpenACC compiler to alter the following code
- **Clauses** are augment directives with extra details and functionality

OpenACC Directives

Manage Data Movement

Initiate Parallel Execution

Optimize Loop Mappings

```
#pragma acc data copyin(x,y) copyout(z)
{
    /* … */
    #pragma acc parallel
    {
    #pragma acc loop gang vector
        for (i = 0; i < n; ++i) {
            z[i] = x[i] + y[i];
            /* … */
        }
    }
    /* … */
}
```

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Xeon Phi

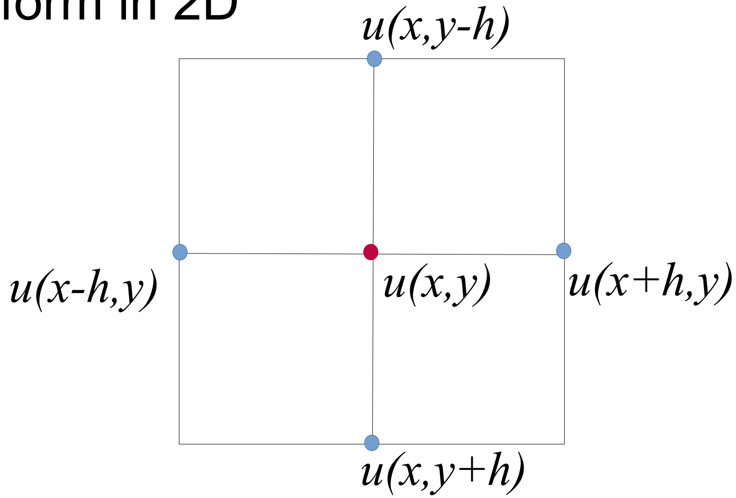
OpenACC
Directives for Accelerators

OpenACC Example: Stencil Code with DATA and KERNELS

```
#pragma acc data copy(b[0:n][0:m]) create(a[0:n][0:m])
{
for (iter = 1; iter <= p; ++iter){
   #pragma acc kernels
   {
   for (i = 1; i < n-1; ++i){
      for (j = 1; j < m-1; ++j){
         a[i][j]=w0*b[i][j]+
                 w1*(b[i-1][j]+b[i+1][j]+
                     b[i][j-1]+b[i][j+1])+
                 w2*(b[i-1][j-1]+b[i-1][j+1]+
                     b[i+1][j-1]+b[i+1][j+1]);
   }
   for( i = 1; i < n-1; ++i )
      for( j = 1; j < m-1; ++j )
         b[i][j] = a[i][j];
}
```

Example: Jacobi Iteration for Poisson Equation

- Poisson equation has a simple form in 2D
  - $u_{xx} + u_{yy} = f(x,y)$
- Applications include
  - Electricity
  - Magnetism
  - Gravity
  - Heat distribution
  - Fluid flow
  - Torsion



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

- When f(x,y)=0 we call it Laplace equation

$$u_{xx} = \frac{\partial^2 u}{\partial x \partial x} \approx \frac{u(x+h, y) - 2u(x, y) + u(x-h, y)}{h^2}$$

Jacobi Iteration: C Code

```
while ( err > tol && iter < iter_max ) {
    err=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]);

            err = max(err, 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 convergence

Iterate across matrix elems.

Calculate new value from neighbors

Compute max-error for convergence

Swap input/output arrays

Looking for Parallelism

```
while ( err > tol && iter < iter_max ) {
    err=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]);

            err = max(err, 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 dependence between iterations

Independent loop iterations

Max reduction required

Independent loop iterations

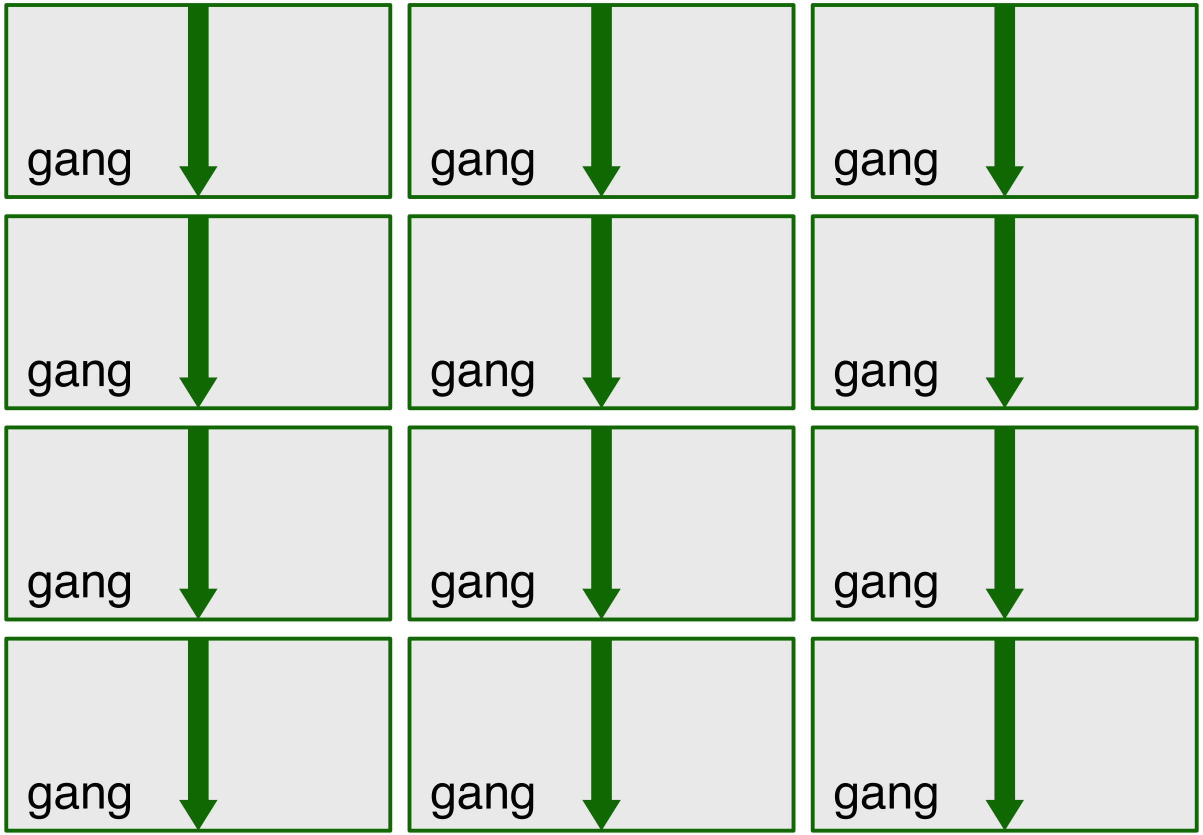
OpenACC Parallel Directive

Generates parallelism

```
#pragma acc parallel
{
```

When encountering the parallel directive, the compiler will generate 1 or more parallel gangs, which execute redundantly.

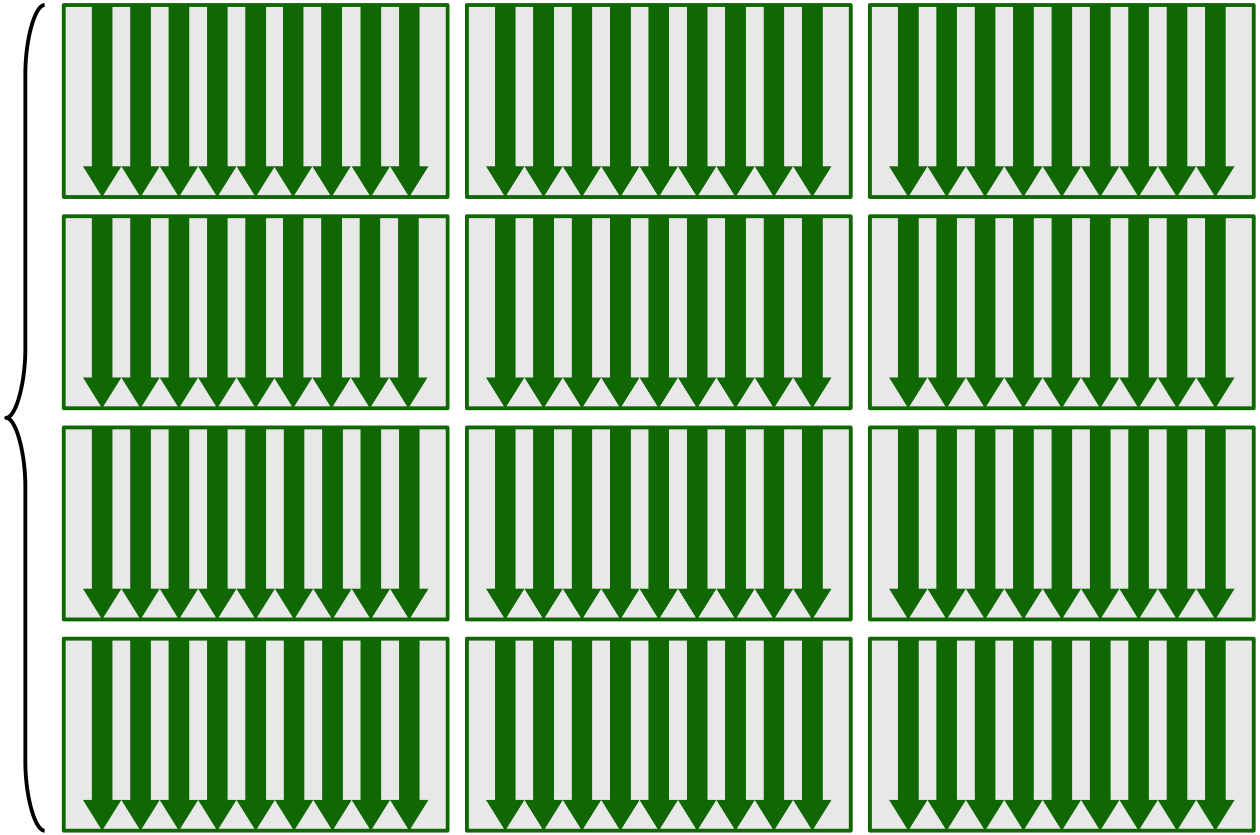
```
}
```



OpenACC Loop Directive

Identifies loops to run in parallel

```
#pragma acc parallel
{
   #pragma acc loop
   for (i=0; i<N; ++i)
   {
   }
}
```

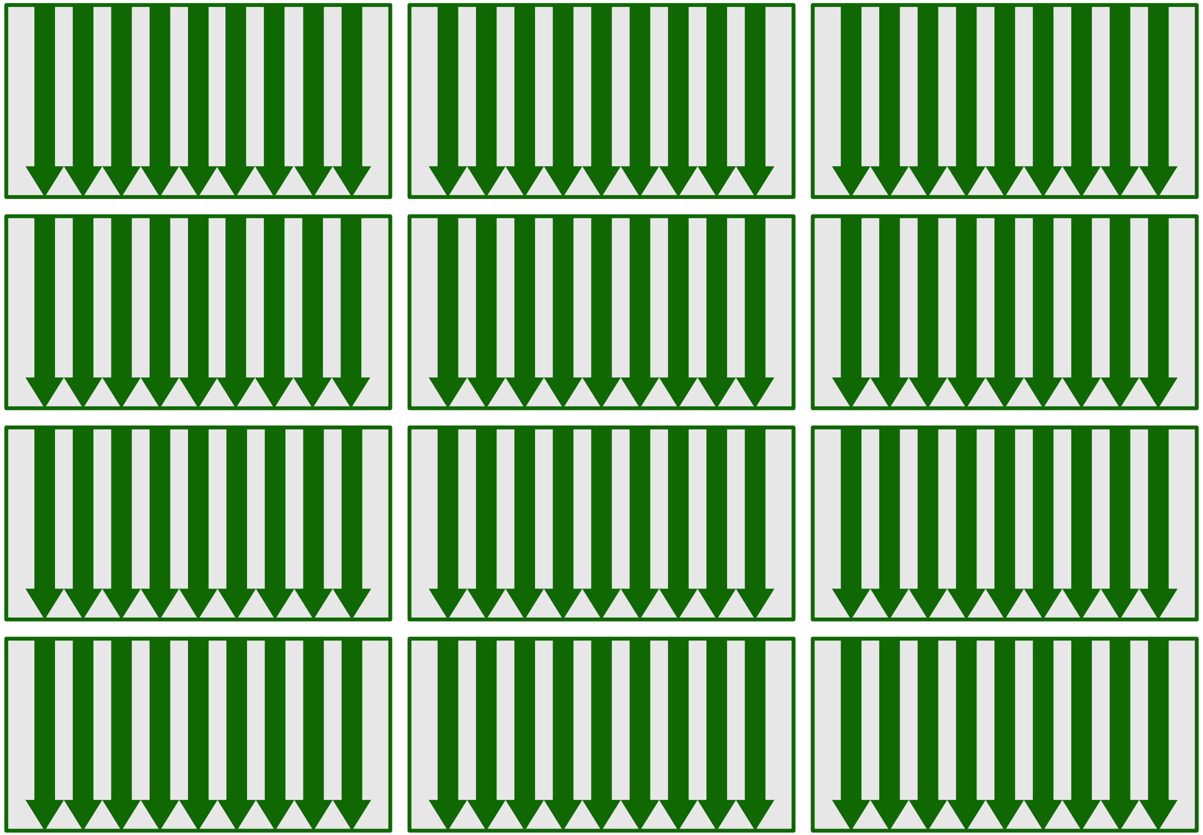


The loop directive informs the compiler which loops to parallelize.

OpenACC Parallel Loop Directive

Generates parallelism and identifies loop in one directive

```
#pragma acc parallel loop
   for (i=0; i<N; ++i)
   {
   }
```



The parallel and loop directives are frequently combined into one.

Looking for Parallelism

```
while ( err > tol && iter < iter_max ) {
   err=0.0;
#pragma acc parallel loop reduction(max:err)
   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]);
         err = max(err, 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++;
}
```

Parallelize loop on accelerator

Parallelize loop on accelerator

A reduction means that all n*m values for err will be reduced to just one, the max.

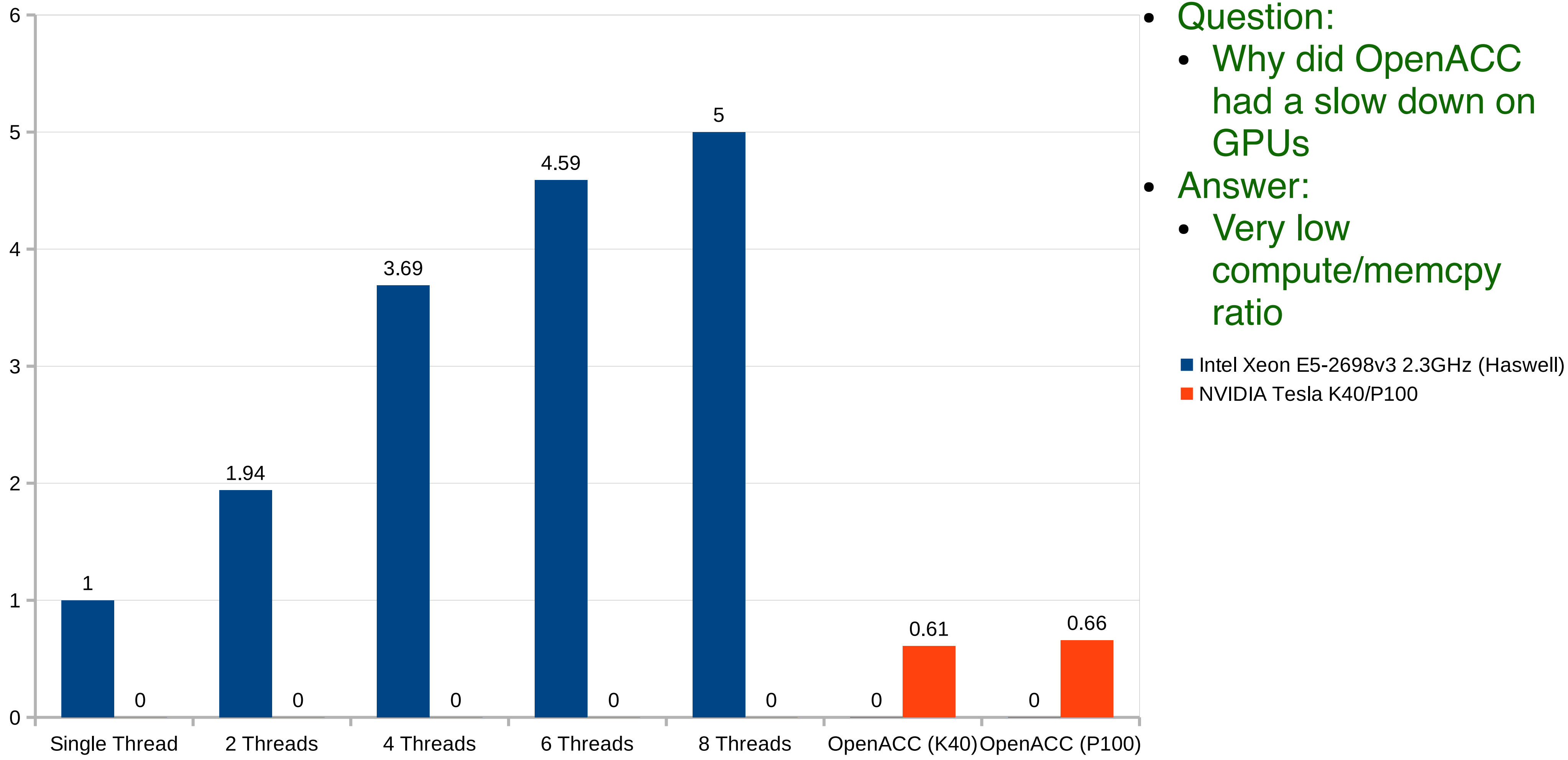
OpenACC Loop Directive: PRIVATE & REDUCTION

- The private and reduction clauses are not optimization clauses, the may be required for correctness.
- Private = a copy of the variable is made for each loop iteration
- Reduction = a reduction is performed on the listed variables
  - Supports: +, *, max, min, and logical operators

Building OpenACC Code with PGI Compiler

```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
40, Loop not fused: function call before adjacent loop
    Generated vector sse code for the loop
51, Loop not vectorized/parallelized: potential early exits
55, Accelerator kernel generated
    55, Max reduction generated for error
    56, #pragma acc loop gang /* blockIdx.x */
    58, #pragma acc loop vector(256) /* threadIdx.x */
55, Generating copyout(Anew[1:4094][1:4094])
    Generating copyin(A[:][:])
    Generating Tesla code
58, Loop is parallelizable
66, Accelerator kernel generated
    67, #pragma acc loop gang /* blockIdx.x */
    69, #pragma acc loop vector(256) /* threadIdx.x */
66, Generating copyin(Anew[1:4094][1:4094])
    Generating copyout(A[1:4094][1:4094])
    Generating Tesla code
69, Loop is parallelizable
```

Performance Results: Speedup (Higher is Better)



- Question:
  - Why did OpenACC had a slow down on GPUs
- Answer:
  - Very low compute/memcpy ratio

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

A, Anew resident on host

copy

A, Anew resident on accelerator

```
#pragma acc parallel loop
   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]);
         err = max(err,
        abs(Anew[j][i] - A[j][i]));
      }
   }
```

These copies happen every iteration of the outer while loop!

A, Anew resident on accelerator

copy

A, Anew resident on host

```
}
```

Looking for Parallelism

```
while ( err > tol && iter < iter_max ) {
   err=0.0;
#pragma acc parallel loop reduction(max:err)
   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]);
         err = max(err, 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++;
}
```

Does CPU need data between these loop nests?

Does CPU need data between iterations of the convergence loop?

Data Regions

The data directive defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
#pragma acc data
{
#pragma acc parallel loop
/* . . . */

#pragma acc parallel loop
/* . . . */
}
```

Data region

Arrays used within the data region will remain on the GPU until the end of the data region.

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.
- `deviceptr( list )`
  - The variable is a device pointer (e.g. CUDA) and can be used directly on the device.

Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”
- C/C++
  - `#pragma acc data copyin(a[0:nelem]) copyout(b[s/4:3*s/4])`
- Fortran
  - `!$acc data copyin(a(1:end)) copyout(b(s/4:3*s/4))`
- Note: data clauses can be used on data, parallel, or kernels

Data Regions Have Real Consequences

```
int main(int argc, char** argv) {
   float A[1000];

   #pragma acc kernels
   for( int iter = 1; iter < 1000;
        ++iter){
      A[iter] = 1.0;
   }

  A[10] = 2.0;

 printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

A[ ] Copied To Host

Runs on Host

Output: A[10] = 2.0

```
int main(int argc, char** argv) {
   float A[1000];
#pragma acc data copy(A)
{
   #pragma acc kernels
   for( int iter = 1; iter < 1000;
        ++iter){
      A[iter] = 1.0;
   } // "kernels" end here
   A[10] = 2.0;
}

  printf("A[10] = %f", A[10]);
}
```

A[ ] Copied To GPU

Still Runs on Host

A[ ] Copied To Host

Output: A[10] = 1.0

Looking for Parallelism

```
#pragma acc data copy(A) create(Anew)
while ( err > tol && iter < iter_max ) {
   err=0.0;
#pragma acc parallel loop reduction(max:err)
   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]);
         err = max(err, 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++;
}
```

Copy A to/from the accelerator only when needed.

Create Anew as a device temporary.

Building the New OpenACC Code with PGI Compiler

```
$ pgcc -fast -acc -ta=tesla -Minfo=all laplace2d.c
main:
40, Loop not fused: function call before adjacent loop
    Generated vector sse code for the loop
51, Generating copy(A[:][:])
    Generating create(Anew[:][:])
    Loop not vectorized/parallelized: potential early exits
56, Accelerator kernel generated
    56, Max reduction generated for error
    57, #pragma acc loop gang /* blockIdx.x */
    59, #pragma acc loop vector(256) /* threadIdx.x */
56, Generating Tesla code
59, Loop is parallelizable
67, Accelerator kernel generated
    68, #pragma acc loop gang /* blockIdx.x */
    70, #pragma acc loop vector(256) /* threadIdx.x */
67, Generating Tesla code
70, Loop is parallelizable
```

Performance Results: Speedup (Higher is Better)