General purpose calculations on

HETEROGENEOUS COMPUTER SYSTEM (GPUs)

N. Shukla

High-Performance Computing Department CINECA

Casalecchio di Reno Bologna, Italy

Email: n.shukla@cineca.it

AGENDA

What is OpenACC?
Overview: Acceleration workflow

Data management
Controlling data movement on the host to device vice versa

Parallelism
Expressing parallelism

Checkpoints

What is OpenACC?

OPENACC IS…

a directives-based **parallel programming model** designed for **performance** and **portability**.

Add Simple Compiler Directive

```
main()
 {
  <serial code>
  #pragma acc kernels
  {
    <parallel code>
  }
}
```

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)

Why OpenACC?

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate mode of the code

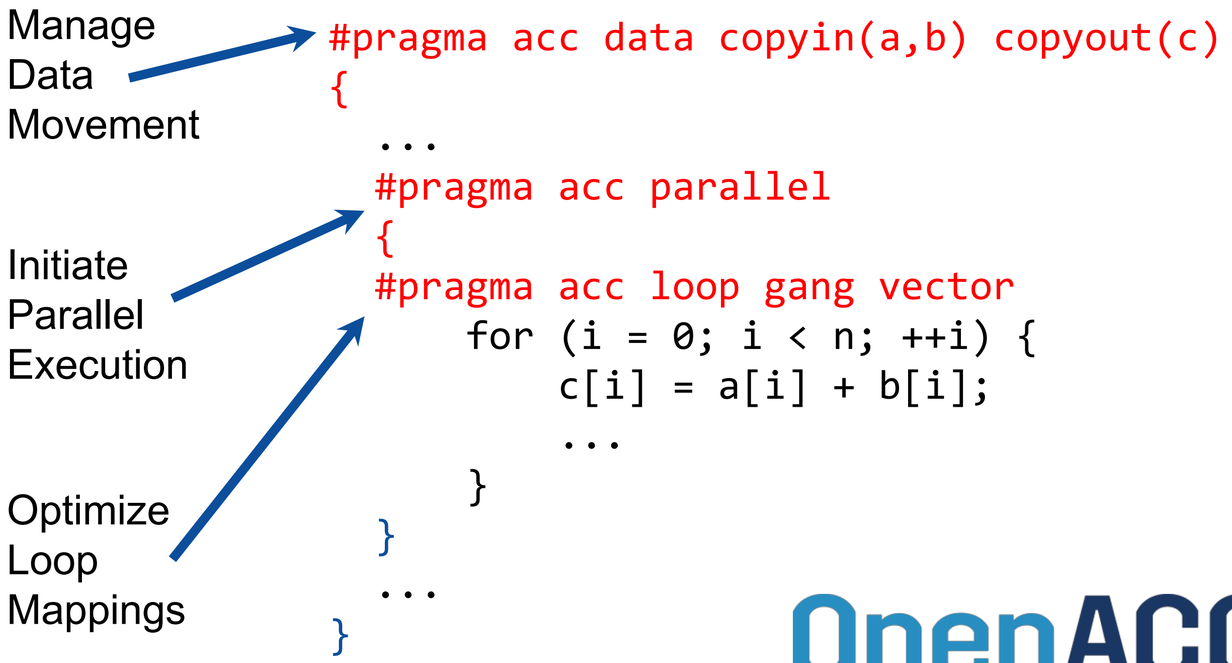
Single source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Low learning curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware

OpenACC Directives Syntax



OpenACC
Directives for Accelerators

- Incremental
- Single source
- Interoperable
- Performance portable
- CPU, GPU, Manycore

Why OpenACC?

Incremental

- Maintain existing sequential code
- Add annotations to expose parallelism
- After verifying correctness, annotate mode of the code

Enhance Sequential Code

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

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

Begin with a working sequential code

Parallelize it with OpenACC

Rerun the code to verify correctness and performance

Why OpenACC?

Single source

- Rebuild the same code on multiple architectures
- Compiler determines how to parallelize for the desired machine
- Sequential code is maintained

Enhance Sequential Code

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

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

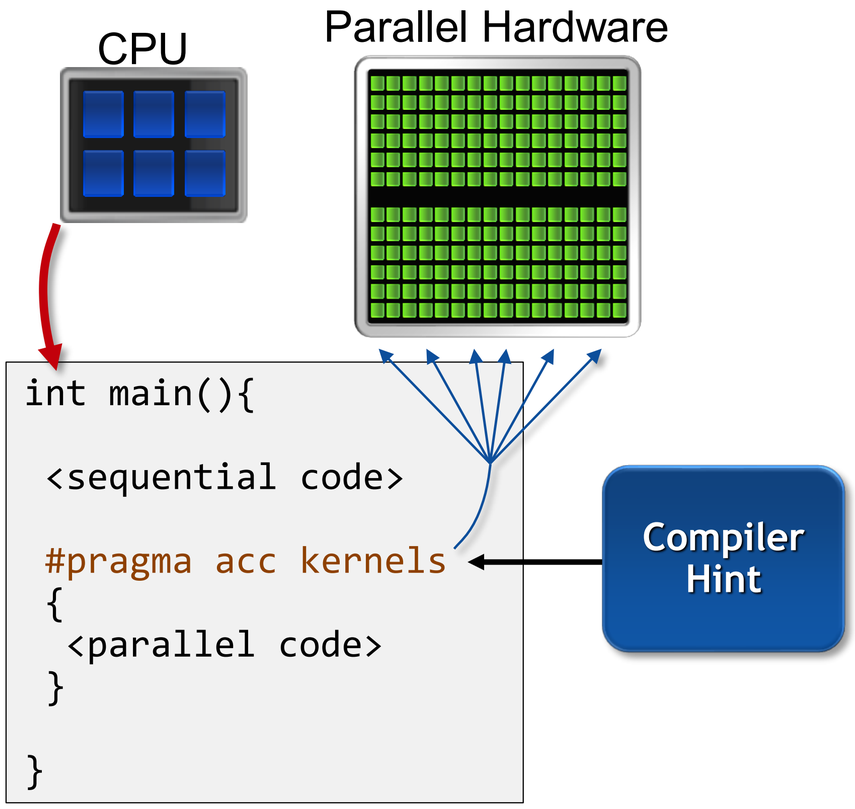
Begin with a working sequential code

Parallelize it with OpenACC

Rerun the code to verify correctness and performance

The compiler can **ignore** your OpenACC code additions, so the same code can be used for **parallel** or **sequential** execution.

Why OpenACC?



The programmer will give hints to the compiler.

The compiler parallelizes the code

Low learning curve

- OpenACC is meant to be easy to use, and easy to learn
- Programmer remains in familiar C, C++, or Fortran
- No reason to learn low-level details of the hardware

Building and running enabled OpenACC code

COMPILER	COMPILER FLAGS	ADDITIONAL FLAGS
PGI	`-acc`	`-ta=`target architecture `-Minfo=accel`
GCC	`-fopenacc`	`-foffload=`offload target
OpenUH	Compile: `-fopenacc` Link: `-lopenacc`	`-Wb,-accarch:`target architecture
Cray	C/C++: `-h pragma=acc` Fortran: `-h acc,noomp`	`-h msgs`

OpenACC Directives Syntax

C and C++

```
# pragma acc directive [ clause…]
  // line code
```

Fortran

```
!$acc directive[ clause…]
   # line code
!$acc end directive [ clause…]
```

- A **pragma** in C/C++ gives instructions to the compiler on how to compile the code. Compilers that do not understand a particular pragma can freely ignore it
- A ***directive*** in Fortran is a specially formatted comment that likewise instructions the compiler in it compilation of the code and can be freely ignored
- "***acc***" informs the compiler that what will come is an OpenACC directive
- ***Directives*** are commands in OpenACC for altering our code
- ***Clauses*** are specifiers or additions to directives

Sample code StreamTriad: DAXPY

Compute a*x+y, where x and y are vectors, and a is a scalar

DAXPY in C

```
int main(int argc, char **argv){
  int N = 1024;
  float A = 16.0f;
  double x[n], y[n];
  for (size_t i=0; i<n; i++)
  {
          x[i] = 1.0;
          y[i] = 2.0;
  }

  for ( size_t i=0; i<n; i++ )
         d[i] = a*x[i] + y[I];
}
```

DAXPY in Fortran

```
program main
 integer :: n=1000, i
 real :: a=3.0
 real, allocatable :: x(:), y(:)
 allocate(x(n),y(n))
 x(1:n)=2.0
 y(1:n)=1.0

 do i=1,n
  d(i) = a * x(i) + y(i)
 enddo

end program main
```

Offloads execution and associated data from the CPU and to the GPU

Programming GPUs can be thought into two parts

Parallelisim	**Data movement**
Expose to the GPU environment	Optimising data locality

Compute directives

- kernels
- parallel
- loop

Data directives

- data
- enter data
- exit data

Offloads execution and associated data from the CPU and to the GPU

Programming GPUs can be thought into two parts

Parallelisim	**Data movement**
Expose to the GPU environment	Optimising data locality

Compute directives

- kernels
- parallel
- loop

Data directives

- data
- enter data
- exit data

Automatic parallelization using Kernels compute directives

Sample code StreamTriad: DAXPY

Compute a*x+y, where x and y are vectors, and a is a scalar

DAXPY in C

```
int main(int argc, char **argv){
  int N = 1024;
  float A = 16.0f;
  double x[n], y[n];
  for (size_t i=0; i<n; i++)
  {
          x[i] = 1.0;
          y[i] = 2.0;
  }
    #pragma acc kernels
  for ( size_t i=0; i<n; i++ )
         d[i] = a*x[i] + y[I];
}
```

DAXPY in Fortran

```
program main
 integer :: n=1000, i
 real :: a=3.0
 real, allocatable :: x(:), y(:)
 allocate(x(n),y(n))
 x(1:n)=2.0
 y(1:n)=1.0

  !$ acc kernels
 do i=1,n
  d(i) = a * x(i) + y(i)
 enddo
!$ acc end kernels
end program main
```

Another approach: a parallel routine to run on the parallel hardware

Compiler identifies 2 parallel loops and generate 2 kernels

DAXPY in C

```
#pragma acc kernels
{
   for (int i=0; i<n; i++)
   {
     x[i] =  1.0;
     y[i] = 2.0;
   }

   for ( int i=0; I<ARRAY_SIZE; i++)
   {
      d[i] = a*x[I] + y[I];
   }
}
```

Kernels 1

Kernels 2

DAXPY in Fortran

```
  // Initization …
!$acc kernels
  do I = 1, N
    D(i) = 0
    X(i) = 1
    Y(i) = 2
  end do

 Do I = 1, N
      D(I) = A*X(I) + Y(I);
  end do
!$acc end kernels
```

Kernels 1

Kernels 2

Take control of your parallelization by using Parallel compute directives

Parallelism identified by programmer

runs same code on multiple threads redundantly

DAXPY in C

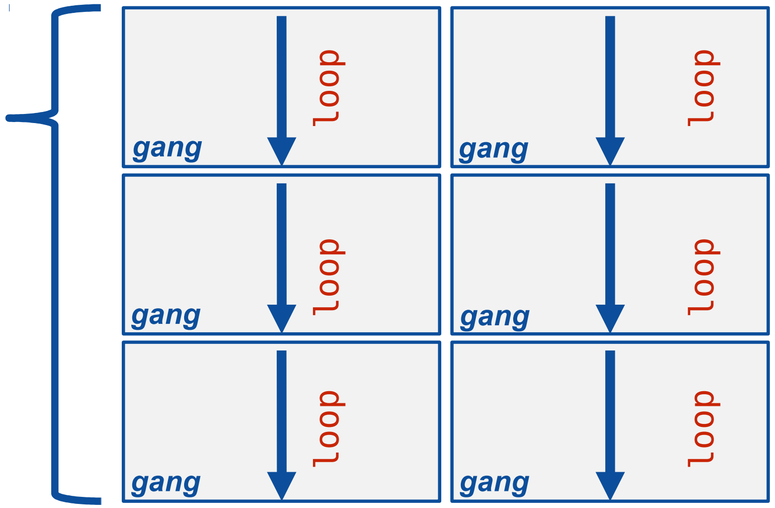
```
#pragma acc parallel
{
   for (int i=0; i<ARRAY_SIZE; i++)
   {
      D[i] = 1.0;
      X[i] = 2.0;
      Y[i] = 0.0;
   }

   for ( int i=0; I<ARRAY_SIZE; i++)
   {
      D[i] = a*X[I] + Y[i];
   }
}
```

Kernels 1

Kernels 2

Compiler generates 1 or more parallel gangs



This means that each ***gang*** will execute the entire loop

Parallel vs kernels

Kernels compute directives

- Compiler's leeway parallelising and optimising a loop
- Only provide by **OpenACC**
- Compiler guarantees correctness
- Or at least it does two things:
  - Fuse the two loop nests into one loop nest
  - Generates two kernels
- Implicit barrier at the end and between each loop

Parallel compute directives

- Programmer's responsibility to identify region for parallelization
- Programmer guarantees correctness
- Parallel just run the same code on multiple threads redundantly
- Guarantees that no dependency occurs iterations
- Implicit barrier at the end of the parallel region

When fully optimised both will give similar performance

C tip: use restrict to avoid pointer aliasing

- 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 to”

- 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

C tip: use restrict to avoid pointer aliasing

```
int main(int argc, char **argv)
{
    int n = 1<<20; // 1 million floats
    double  *x = (double*)malloc(n * sizeof(double));
    double  *y = (double*)malloc(n * sizeof(double));

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

    daxpy(n, 16.0f, x, y);

    return 0;
}
```

```
#include <stdlib.h>

void daxpy( size_t n,
            float a,
            double *x,
            double *restrict y )
{
      #pragma acc kernels
      for ( int i = 0; i < n; i++)
            y[I] = a*x[i] + y[I];

}
```

"I promise y does not alias x"

Parallel directive must combine with one or more loop directives

Loop directives to specify where to apply work sharing, relatively useless by itself

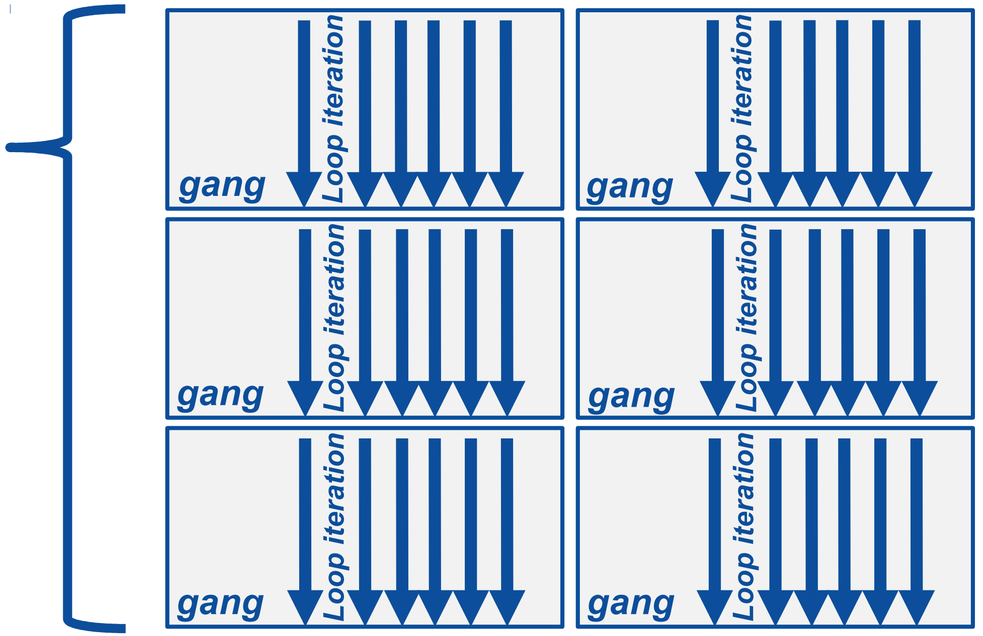
DAXPY in C

```
#pragma acc parallel
{
   #pragma acc loop
   for (int i=0; i<ARRAY_SIZE; i++)
   {
      D[i] = 1.0;
      X[i] = 2.0;
      Y[i] = 0.0;
   }
   #pragma acc loop
   for (int i=0; i<ARRAY_SIZE; i++)
   {
      D[i] = a*X[I] + Y[i];
   }
}
```

Kernels 1

Kernels 2

Loop iterations are distributed to the gangs and runs in parallel



Parallelising many loop

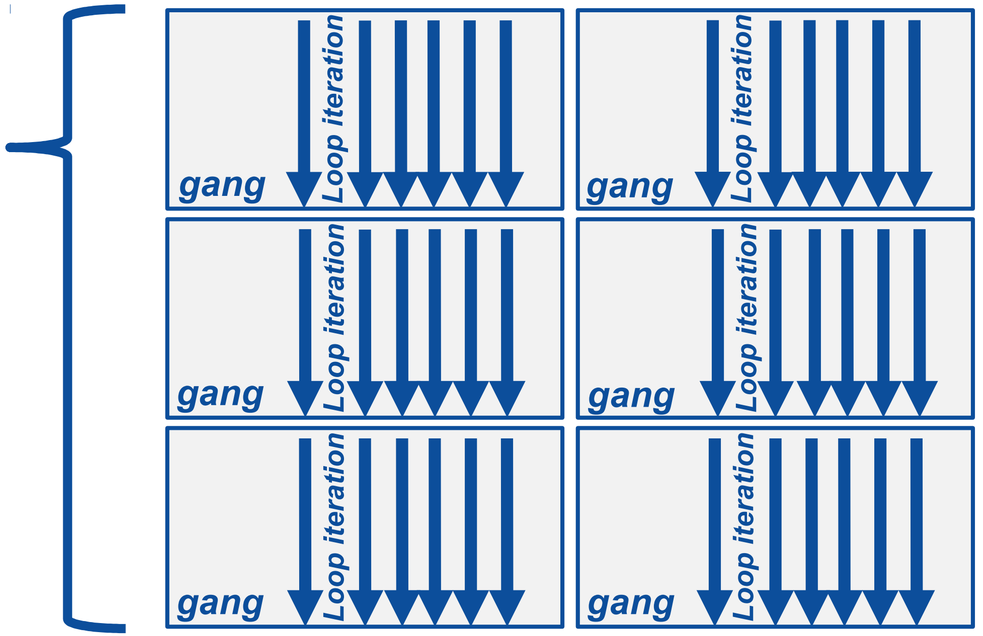
DAXPY in C

```
#pragma acc parallel loop
{
    for (int i=0; i<ARRAY_SIZE; i++)
    {
       D[i] = 1.0;
       X[i] = 2.0;
       Y[i] = 0.0;
    }
}
#pragma acc parallel loop
{
    for (int i=0; i<ARRAY_SIZE; i++)
    {
       D[i] = a*X[I] + Y[i];
    }
}
```

Kernels 1

Kernels 2

Loop iterations are distributed to the gangs and runs in parallel



Compile and Run

Compiling with PGI compiler

- Compile a C code : pgcc
- Compile a C++ code : pgc++
- Compile a Fortran code : pgfortran
- The -fast flag instructs the compiler to optimize the code to the best of its abilities

```
$ pgcc –fast -o binary main.c
$ pgc++ –fast -o binary main.cpp
$ pgfortran –fast -o binary main.f90
```

Compiling with PGI compiler

- Minfo flag

  - The -Minfo flag will instruct the compiler to print feedback about the compiled code
  - -Minfo=accel will give us information about what parts of the code were accelerated via OpenACC
  - -Minfo=opt will give information about all code optimizations
  - -Minfo=all will give all code feedback, whether positive or negative

```
$ pgcc –fast -Minfo=accel -o binary main.c
$ pgc++ –fast -Minfo=accel -o binary main.cpp
$ pgfortran –fast -Minfo=accel -o binary main.f90
```

Compiling with PGI compiler

- ta flag

- The -ta flag enables building OpenACC code for a “Target Accelerator” (TA)
- -ta=multicore – Build the code to run across threads on a multicore CPU
- -ta=tesla:managed – Build the code for an NVIDIA (Tesla) GPU and manage the data movement for me

```
$ pgcc –fast -Minfo=accel -ta=tesla:managed -o binary main.c
$ pgc++ –fast -Minfo=accel -ta=tesla:managed -o binary main.cpp
$ pgfortran –fast -Minfo=accel -ta=tesla:managed -o binary main.f90
```

-acc=noautopar:

noautopar avoids automatically parallelization of loops

Compile and analysis of the compiling output

```
$ make pgi_c
pgcc -O3 -ta=multicore -Minfo=all -acc=noautopar -o binary daxpy.c
daxpygpu:
     12, Loop not vectorized: data dependency
         Loop unrolled 4 times
         FMA (fused multiply-add) instruction(s) generated
main:
     12, FMA (fused multiply-add) instruction(s) generated
     44, Loop not vectorized: data dependency
         Loop unrolled 4 times
     53, daxpygpu inlined, size=7 (inline) file daxpy.c (9)
          12, Loop not vectorized: data dependency
              Loop unrolled 4 times
     58, Loop not vectorized/parallelized: potential early exits
```

-ta=multicore

```
$ make pgi_c
pgcc -O3 -ta=tesla:managed -Minfo=all -acc=noautopar -o binary daxpy.c
daxpygpu:
      13, Generating Tesla code
          13, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      13, Generating implicit copyout(D[:n]) [if not already present]
          Generating implicit copyin(Y[:n],X[:n]) [if not already present]
main:
      41, Loop not vectorized: data dependency
          Loop unrolled 4 times
      55, Loop not vectorized/parallelized: potential early exits
```

- Accelerator kernel is generated.
- The loop computation is offloaded to (Tesla) GPU and is parallelized.
- We will mention these later.

  The keywords copy and copyin are involved with data transfer.

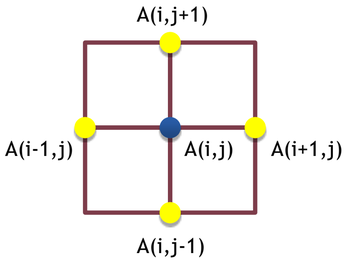
  The keywords gang and vector are involved with tasks granularity.

Handson: Jacobi Iteration

- Jacobi method is an iterative method to solver a system of linear equations
- it is a very common and useful algorithm (very famous in HPC tutorials)
- Example: Jacobi method can be used to solve the Laplace differential equation in two variables (a 2D propagation problem)

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

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



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

Tasks-1: 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++;
}
```

- Iterate until converged
- Iterate across matrix element
- Calculate new value from neighbours
- Compute max error for convergence
- Swap input/output arrays

Accelerate serial code with OpenACC

- Add Kernels
- Parallel Loop
- Save execution time

OpenACC execution model has three level of parallelism Gangs, workers and vectors

Gang:

Finest granularity

Threads work in lockstep (SIMD/SIMT parallelism)

Worker:

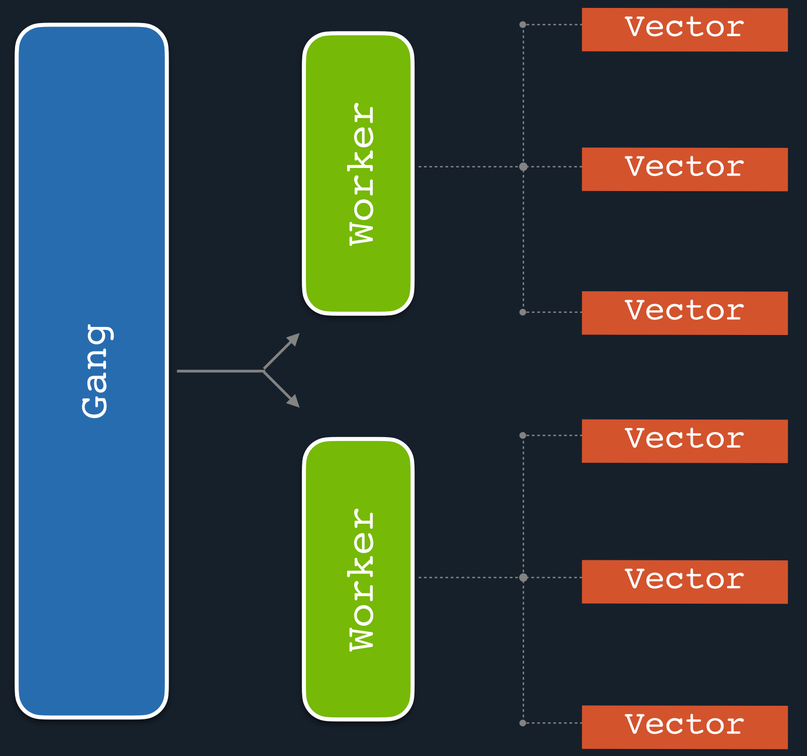
Compute a vector

Intermediate level between the low-level parallelism implemented in vector and group of threads

Vector parallelism:

Finest granularity

Threads work in lockstep (SIMD/SIMT parallelism)



OpenACC Gang, Worker, Vector Clauses

- The developer can instruct the compiler which levels of parallelism to use on given loops by adding clauses: distribution
- **gangs -** Mark this loop for gang parallelism
- **worker -** Mark this loop for worker parallelism
- **vector -** Mark this loop for vector parallelism

These can be combined on the same loop.

```
clock_t begin = clock();
#pragma acc parallel loop gang
for (i=0; i<rows; i++)
        #pragma acc loop worker
        for (k=0; k<rows; k++)
                #pragma acc loop vector
                for (j=0; j<cols; j++)
                        c[i][j] = a[i][k]*b[k][j];
clock_t end = clock();
```

Controlling the size of Gang, Worker and Vectors

The compiler will choose a number of gangs, workers, and a vector length for you, but you can change it with clauses

- **num_gangs(N)** - Generate N gangs for this parallel region
- **num_workers(M)** - Generate M workers for this parallel region
- **vector_length (P)** - Use a vector length of P for this parallel region

```
#pragma acc parallel num_gangs(2) num_workers(2) \
vector_length(32)
{
  #pragma acc loop gang worker
  for (int i = 0; i<4; i++) {
    #pragma acc loop vector
    for (int j = 0; j<32; j++) {
            c[I] [j]++;
    }
  }
}
```

Best practices for gang, workers and vectors

- Use gangs for outer loops and vector for inner loops
- Do not set manually the number of gangs but let the compiler do it
- Start running a loop by setting number of vectors: most of the time this enough
- Vector should access data consecutively in order to exploit locality
- **Vector number should be multiple of 32 on nVidia GPUs (in order to map wraps)**
- Worker loop is used early on GPUs

The collapse clause

collapse(n):

- Combine the next N tightly nested loops
- Can tune a multidimensional loop nest into a single-dimension loop
- Extremely useful for increasing memory locality, as well as creating larger loops to expose more parallelism

```
clock_t begin = clock();

#pragma acc parallel
#pragma acc loop collapse(2)
for (i=0; i<size; i++)
                for (j=0; j<size; j++)
                        c[i][j] = a[i][j] + b[i][j];

clock_t end = clock();
```

The compiler may decide to collapse loops anyway, check the report!!

The SEQ and collapse clause

- The SEQ clause (short for sequential) will tell the compiler to run the loop sequentially
- In the sample code, the compiler will parallelize the outer loops across the parallel threads, but each thread will run the inner-most loop sequentially
- The compiler may automatically apply the SEQ clause to loops as well

```
clock_t begin = clock();
#pragma acc parallel loop
for (i=0; i<size; i++)
        #pragma acc loop
        for (k=0; k<size; k++)
                #pragma acc loop seq
                for (j=0; j<size; j++)
                        c[i][j] += a[i][k] + b[k][j];
clock_t end = clock();

double time_spent = (end - begin);
```

The reduction clause

- The reduction clause takes many values and "reduces" them to a single value, such as in a sum or maximum
- Variable will be operated on locally for each thread
- Added to a kernels, parallel, or loop directives

```
double t_start = getClock();
#pragma acc parallel loop reduction(+:sum)
for ( size_t i=0; i<ARRAY_SIZE; i++ )
{
        D[i] = A*X[i] + Y[i];
        sum += D[i];
}

double t_end = getClock();
```

Loop directive clauses

Independent

- Use carefully
- Overrides compiler analysis for dependence

private(list)

- Private data for each iteration of the loop

scheduling

- Auto, SEQ, vector, worker, gang,

Tasks-2: 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++;
}
```

- Iterate until converged
- Iterate across matrix element
- Calculate new value from neighbours
- Compte max error for convergence
- Swap input/output arrays

Accelerate serial code with OpenACC

- Add collapse and reduction clauses
- Save execution time

Solution

```
#pragma acc parallel loop reduction(max:error) collapse(2)
        for( j = 1; j < n-1; j++) {
            for( 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 = fmax( error, fabs(Anew[j][i] - A[j][i]));
            }
        }

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

        if(iter % 10 == 0) printf("%5d, %0.8lf\n", iter, error);
        iter++;
    }
```

Performance

Execution	Time (s) PGI
CPU1 OpenMP Thread	28.44
CPU2 OpenMP Thread	14.23
CPU4 OpenMP Thread	7.17
CPU8 OpenMP Thread	3.61
CPU16 OpenMP Thread	1.92
OpenACC GPU	5.67

OpenACC - porting strategy

When using OpenACC, follow this

1. **Analyze**

   Identify the compute intensive loops

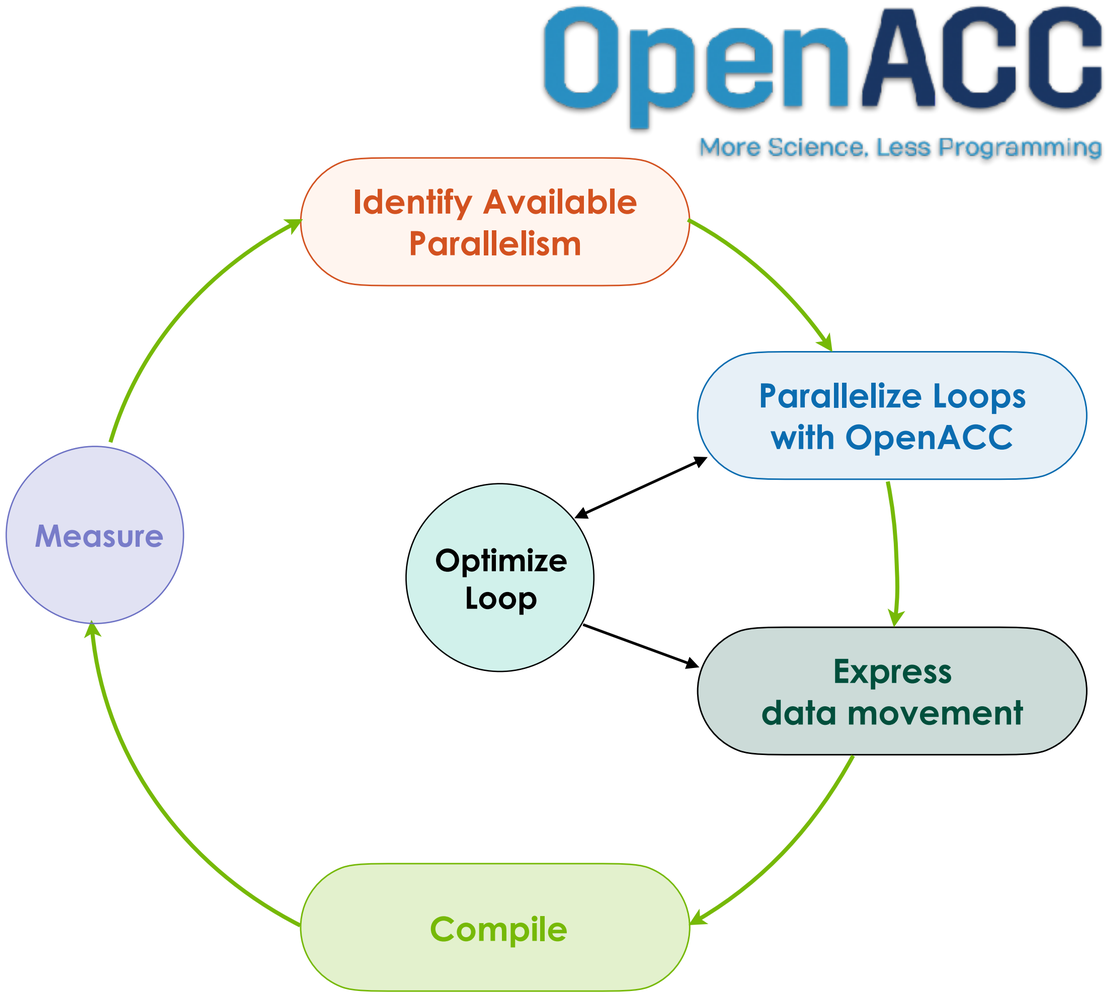
2. **Parallelization**

   Add OpenACC directives

3. **Optimizations**

   Optimize data transfers and loops

Repeat steps 1 to 3 until everything is on GPU



What is going wrong here?

Set export PGI_ACC_TIME=1

```
main  NVIDIA  devicenum=0
  time(us): 868,358
  49: compute region reached 100 times
      49: kernel launched 100 times
          grid: [65535]  block: [128]
          elapsed time(us): total=41,983 max=429 min=417 avg=419
      49: reduction kernel launched 100 times
          grid: [1]  block: [256]
          elapsed time(us): total=12,431 max=128 min=122 avg=124
  49: data region reached 200 times
      49: data copyin transfers: 900
           device time(us): total=217,776 max=289 min=12 avg=241
      55: data copyout transfers: 900
           device time(us): total=215,972 max=284 min=9 avg=239
  58: compute region reached 100 times
      58: kernel launched 100 times
          grid: [65535]  block: [128]
          elapsed time(us): total=40,998 max=414 min=408 avg=409
  58: data region reached 200 times
      58: data copyin transfers: 800
           device time(us): total=220,002 max=305 min=264 avg=275
      62: data copyout transfers: 800
           device time(us): total=214,608 max=283 min=261 avg=268
```

0.041983 seconds

0.2177 seconds

0.215 seconds

0.041 seconds

0.22 seconds

0.214 seconds

Data transfer Bottleneck!

Computation: 0.83 seconds
Data movement: 0.88 seconds

Offloads execution and associated data from the CPU and to the GPU

Programming GPUs can be thought into two parts

Parallelisim	**Data movement**
Expose to the GPU environment	Optimising data locality

Compute directives

- kernels
- parallel
- loop

Data directives

- data
- enter data
- exit data

Offloads execution and associated data from the CPU and to the GPU

Programming GPUs can be thought into two parts

Parallelisim	**Data movement**
Expose to the GPU environment	Optimising data locality

Compute directives

- kernels
- parallel
- loop

Data directives

- data
- enter data
- exit data

Only Optimization in OpenACC

The data movement is a bottleneck

Optimize data locality

- Minimize memory copy from host to device or vice versa
- Avoid duplication if possible

  Only copy from or to Host what is valuable

  Maximise reuse of the copied data
- Copy only what's needed
- Contiguity in memory is your friend

  Compilers like it, increases cache reuse

Data clause

These clauses can be used with the kernels, parallel, data and declare constructs:

`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.

Principal use: For many important data structures in your code, this is a logical default to input, modify and return the data.

`copyin( list )`

Allocates memory on GPU and copies data from host to GPU when entering region.

Principal use: Think of this like an array that you would use as just an input to a subroutine.

Data clause

These clauses can be used with the kernels, parallel, data and declare constructs:

`copyout( list )`

Allocates memory on GPU and copies data to the host when exiting region.

Principal use: A result that isn't overwriting the input data structure.

`create( list )`

Allocates memory on GPU but does not copy.

Principal use: Temporary arrays.

present (list)

Data is already present on GPU from another containing data region

No allocation or copy required.

present_or_copy[in|out], present_or_create, deviceptr

Compiler often makes a good guess

DAXPY in C

```
#pragma acc parallel loop
 for (size_t i=0; i<ARRAY_SIZE; i++)
 {
        D[i] = 0.0;
        X[i] = 1.0;
        Y[i] = 2.0;
 }

 start_time = omp_get_wtime();
 #pragma acc parallel loop
 for ( size_t i=0; i<ARRAY_SIZE; i++ )
        D[i] = A*X[i] + Y[i];
 end_time = omp_get_wtime();
```

Compiler output

```
pgcc -mp -acc -O3 -ta=tesla -Minfo=all -acc=noautopar -o binary daxpy.c
main:
     32, Generating implicit
copyout(X[:500000000],Y[:500000000],D[:500000000]) [if not already
present]
     33, Loop is parallelizable
         Generating Tesla code
         33, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
     41, Complex loop carried dependence of X->,Y-> prevents
parallelization
         Loop carried dependence of D-> prevents parallelization
         Loop carried backward dependence of D-> prevents
vectorization
         Accelerator serial kernel generated
         Generating Tesla code
         41, #pragma acc loop seq
     52, Loop not vectorized/parallelized: potential early exits
```

Data construct syntax and scope

Data region constructs

- A data region is the dynamic scope of a structured block associated with an implicit or explicit data construct.
- Facilities the sharing of data between multiple parallel regions (kernels, parallel, loop *etc*)
- Must start and end in the scope of the same function or subroutine – it's a structure construct

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.

Syntax:

C:

```
#pragma acc data [clause]
{
    code region ...
    including compute related pragmas
}
```

Fortran:

```
!$acc data
    code region ...
    including compute related pragmas
!$acc end data
```

Data construct syntax and scope

DAXPY in C

```
#pragma acc data
{
   #pragma acc parallel loop
   for (int i=0; i<ARRAY_SIZE; i++)
   {
      D[i] = 1.0;
      X[i] = 2.0;
      Y[i] = 0.0;
   }
   #pragma acc parallel loop
   for (int i=0; i<ARRAY_SIZE; i++)
   {
      D[i] = a*X[I] + Y[i];
   }
}
```

Kernels 1

Kernels 2

Data Region

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

Shaping arrays

Structured data

Compiler sometimes cannot determine size of arrays

- Must specify explicitly start/end point
- Memory only exits within the data region
- Must be within a single function

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))
```

- Fortran uses *start:end* and C uses *start:count*
- Data clauses can be used on data, kernels or parallel

Note: data clauses can be used on data, parallel, or kernels

Explicitly copying structured data on the gpu

```
start_time = omp_get_wtime();
 #pragma acc data copyin( X[0:n], Y[0:n] ) copyout(D[0:n])
{
#pragma acc parallel loop
for ( size_t i=0; i<n; i++ )
        D[i] = A*X[i] + Y[i];
}
end_time = omp_get_wtime();
```

Compiler often makes a good guess

DAXPY in C

```
#pragma acc data create( D[0:ARRAY_SIZE], Y[0:ARRAY_SIZE] ) copyin(A) copyout(D[0:ARRAY_SIZE])
 {
#pragma acc parallel loop
 for (size_t i=0; i<ARRAY_SIZE; i++)
 {
        D[i] = 0.0;
        X[i] = 1.0;
        Y[i] = 2.0;
 }

 start_time = omp_get_wtime();
 #pragma acc parallel loop
 for ( size_t i=0; i<ARRAY_SIZE; i++ )
        D[i] = A*X[i] + Y[i];
 end_time = omp_get_wtime();
 }
```

Compute region

Data region

Copy data clause helps compiler to know about it's array size

Compiler output

```
pgcc -mp -acc -O3 -ta=tesla -Minfo=all -acc=noautopar -o binary daxpy.c
main:
     32, Generating copyout(D[:ARRAY_SIZE]) [if not already present]
         Generating copyin(A) [if not already present]
         Generating Tesla code
      34, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      32, Generating create(Y[:ARRAY_SIZE]) [if not already present]
          Generating implicit copyout(X[:500000000]) [if not already present]
      41, Generating Tesla code
      43, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
      41, Generating implicit copyin(X[:500000000]) [if not already present]
      54, Loop not vectorized/parallelized: potential early exits
```

Explicitly copying unstructured data on the gpu

unStructured data

- Can have multiple starting/ending points
- Memory exists until explicitly deallocated
- Can branch across multiple functions

```
start_time = omp_get_wtime();
 #pragma acc enter data copyin( X[0:n], Y[0:n] ) create(D[0:n])
{
#pragma acc parallel loop
for ( size_t i=0; i<n; i++ )
        D[i] = A*X[i] + Y[i];
}
end_time = omp_get_wtime();

#pragma acc exit data copyout(D[0:n]) delete(X,Y)
```

Update directive

- Some data synchronization is required between H and D while computation in progress.
  - E.g. Check-pointing or writing results on disk.
- update directive copies data allocated on device memory to the CPU memory or vice versa.
  - It may appear within a data region, implicit or explicit.

Syntax:

C :

```
#pragma acc update [clause]
```

Fortran:

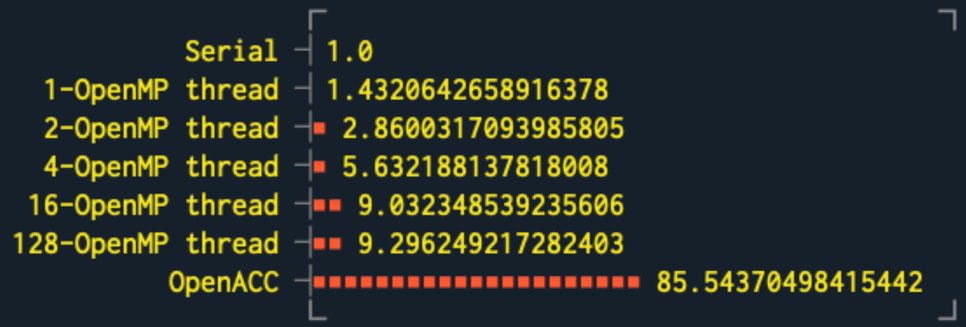
```
!$acc update [clause]
```

Example:

```
if (iteration % 100){
    #pragma acc update (x[0:end])
    write_output(x);
}
```

Triad program is a memory-intensive benchmark

```
julia> p
                Triad operation (A[i] = B[i] + constant * C[i])
```



The total memory allocated = 13.000 GB

Tasks-3: 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++;
}
```

- Iterate until converged
- Iterate across matrix element
- Calculate new value from neighbours
- Compte max error for convergence
- Swap input/output arrays

Accelerate serial code with OpenACC

- Use parallel loop and data construct
- Perfromance

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++;
}
```

```
$ make pgi_c
pgcc -O3 -ta=tesla -Minfo=all -acc=noautopar -o laplace2d_parallel_collapse_data laplace2d.c
main:
     34, Loop unrolled 8 times
     45, Generating create(Anew[:][:]) [if not already present]
         Generating copy(A[:][:]) [if not already present]
         Loop not vectorized/parallelized: potential early exits
     50, Generating implicit copy(error) [if not already present]
         Generating Tesla code
         50, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
             Generating reduction(max:error)
         51,   /* blockIdx.x threadIdx.x collapsed */
     59, Generating Tesla code
         59, #pragma acc loop gang, vector(128) collapse(2) /* blockIdx.x threadIdx.x */
         60,   /* blockIdx.x threadIdx.x collapsed */
     70, FMA (fused multiply-add) instruction(s) generated
```

Performance

Execution	Time (s) PGI
CPU1 OpenMP Thread	28.44
CPU2 OpenMP Thread	14.23
CPU4 OpenMP Thread	7.17
CPU8 OpenMP Thread	3.61
CPU16 OpenMP Thread	1.92
OpenACC GPU	0.13

Express data movement

```
#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++;
 }
```

```
$ make pgi_c
pgcc -O3 -ta=tesla -Minfo=all -acc=noautopar -o laplace2d_kernels_data_optimized laplace2d.c
main:
     34, Loop unrolled 8 times
     45, Generating create(Anew[:][:]) [if not already present]
         Generating copy(A[:][:]) [if not already present]
         Loop not vectorized/parallelized: potential early exits
     50, Loop is parallelizable
         Generating implicit copy(error) [if not already present]
     52, Loop is parallelizable
         Generating Tesla code
         50, #pragma acc loop gang(32), vector(16) /* blockIdx.y threadIdx.y */
             Generating implicit reduction(max:error)
         52, #pragma acc loop gang(16), vector(32) /* blockIdx.x threadIdx.x */
     60, Loop is parallelizable
     62, Loop is parallelizable
         Generating Tesla code
         60, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         62, #pragma acc loop gang(16), vector(32) /* blockIdx.x threadIdx.x */
     72, FMA (fused multiply-add) instruction(s) generated
```

Performance

Execution	Time (s) PGI
CPU1 OpenMP Thread	28.44
CPU2 OpenMP Thread	14.23
CPU4 OpenMP Thread	7.17
CPU8 OpenMP Thread	3.61
CPU16 OpenMP Thread	1.92
OpenACC GPU	0.17

Closing thoughts

- OpenACC is a directive based API to offload code sections on GPUs.
- Data dependencies and Data movement must be understood
- Kernel construct allows compiler to optimize data dependency and movement to the best of its understanding
- Data regions can be used to make data movement coarse grain.

Grazie Mille!!

Feel free to reach me out
n.shukla@cineca.it