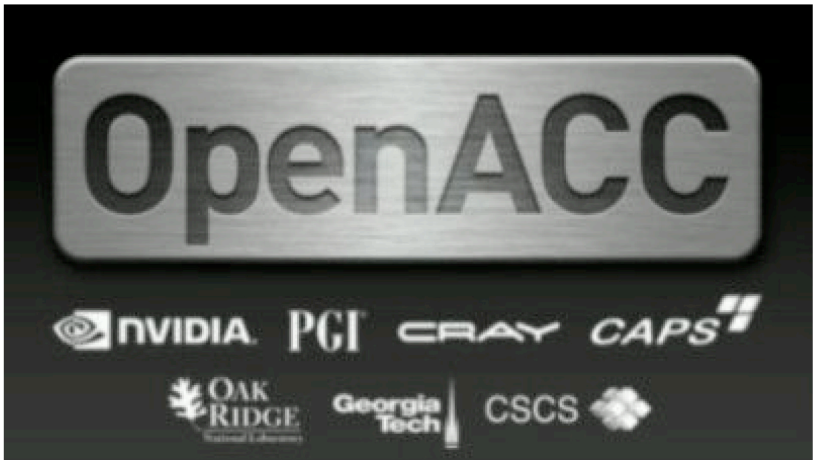


Intro to OpenACC

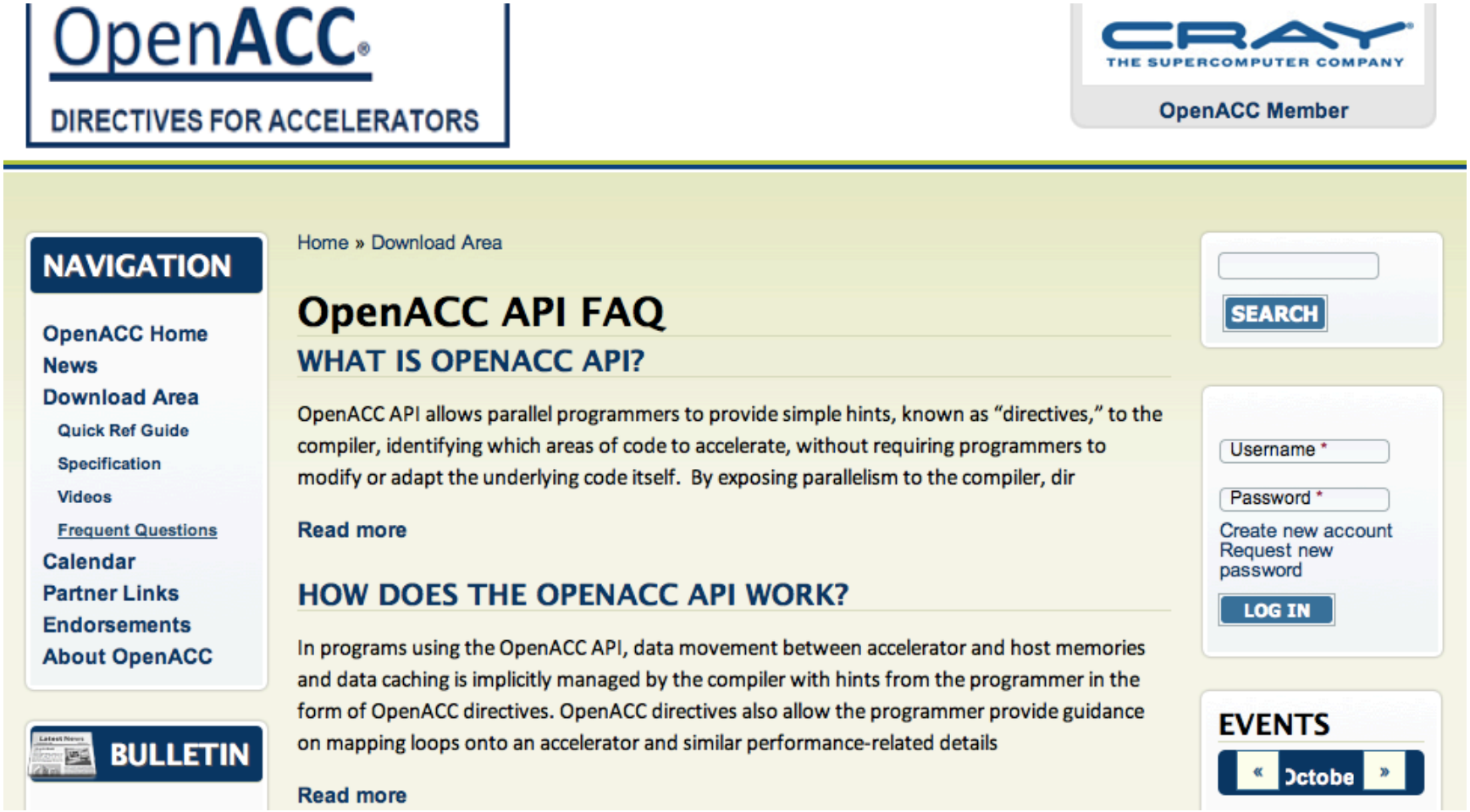
Heterogeneous Computing Using Accelerators at ORNL

OLCF Fall User Training 2012

OpenACC Info

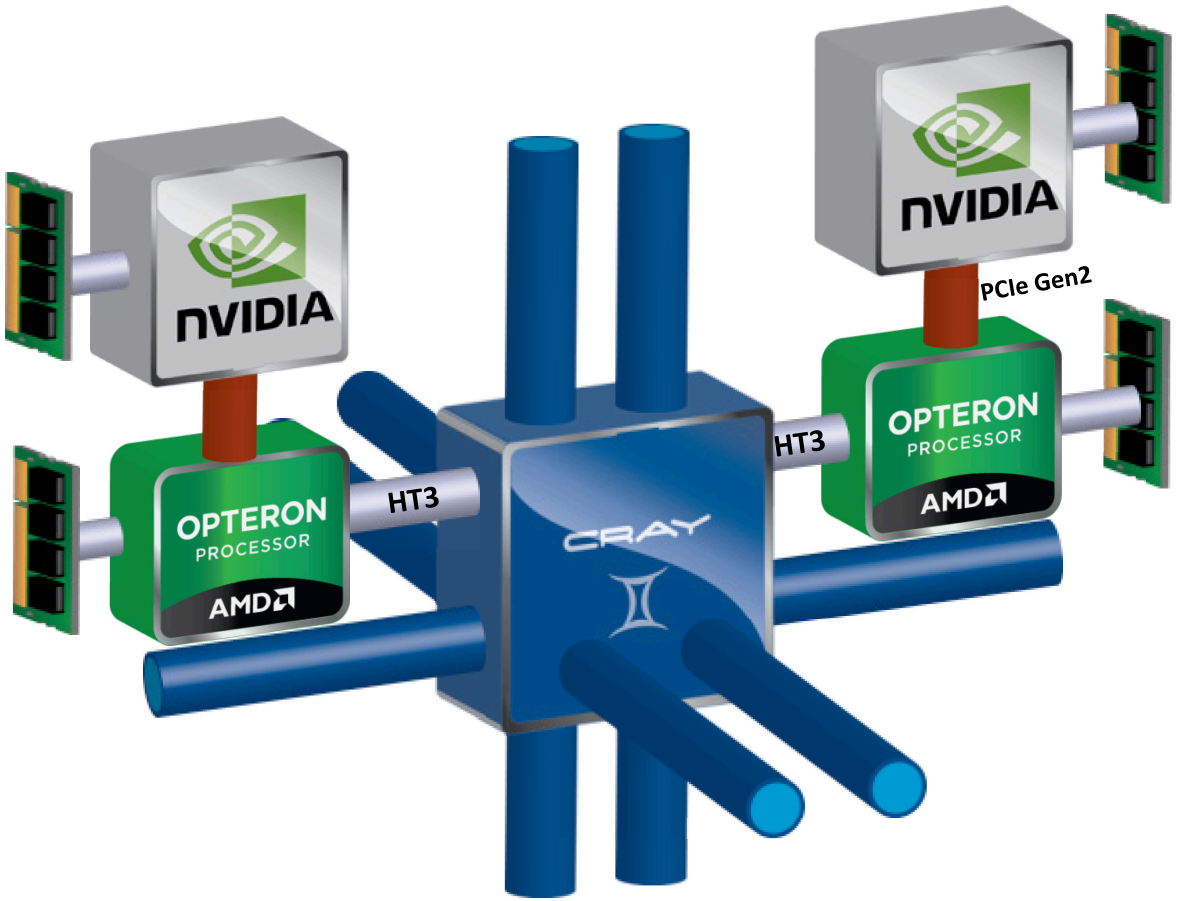
- OpenACC was developed by PGI, Cray, CAPS and Nvidia
- Specification 1.0 released Nov 2011

http://www.openacc.org

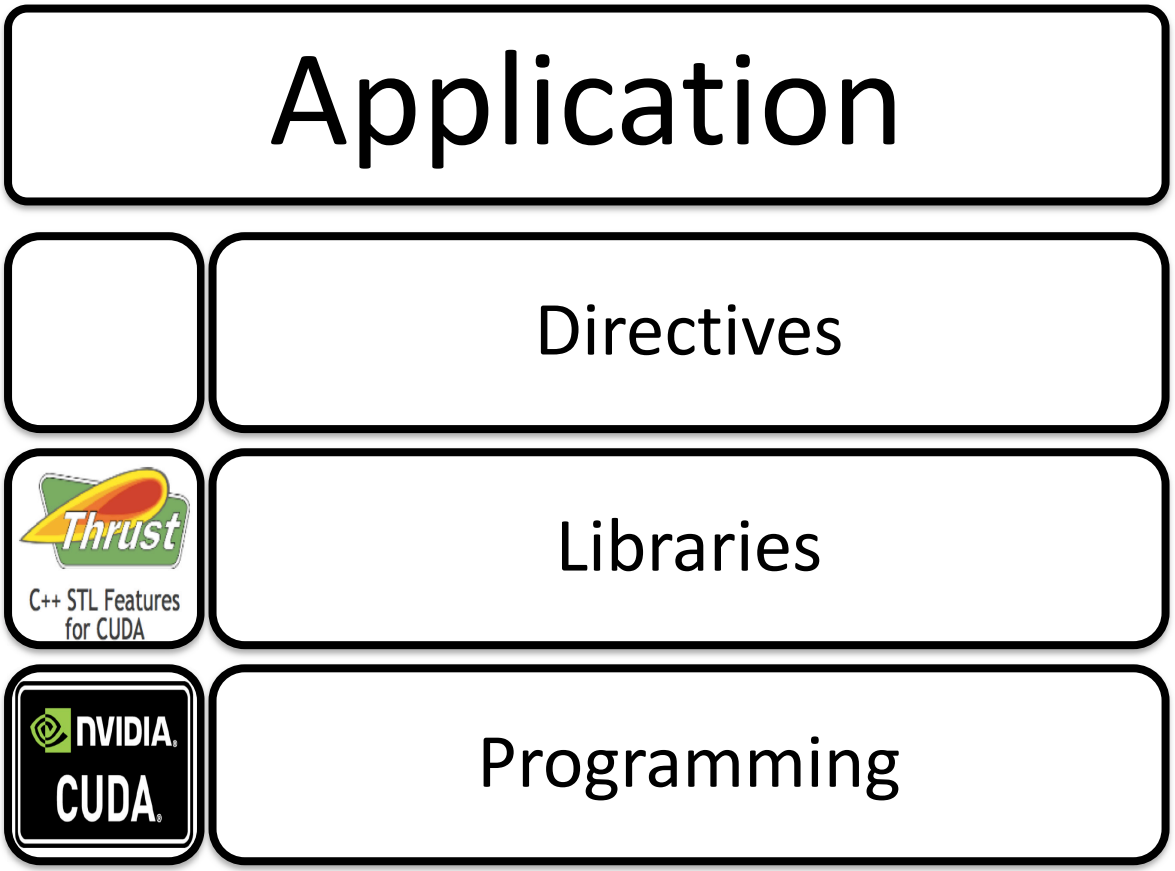


Heterogeneous Computing

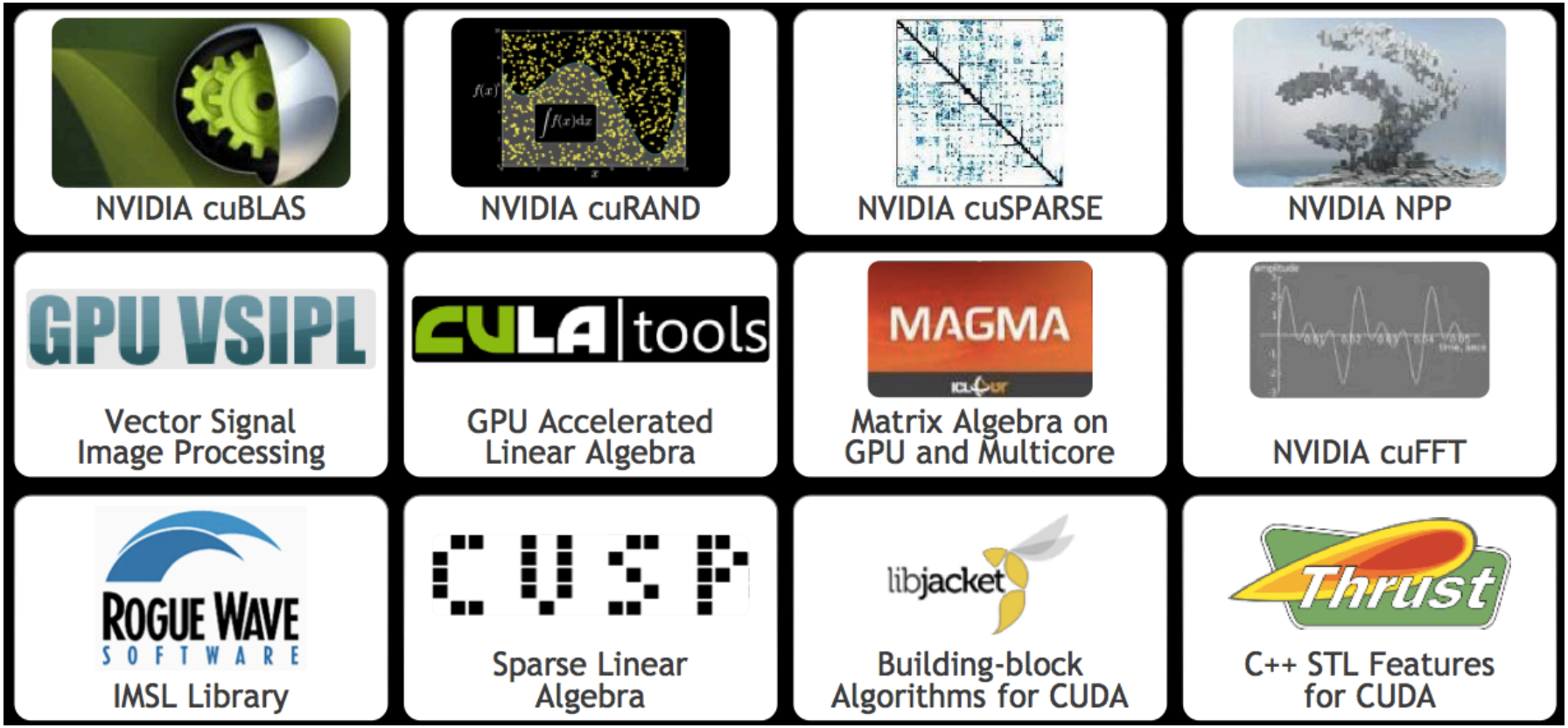
CPUs: designed to multitask.
GPUs: designed to single task



How to Accelerate Applications



Libraries


NVIDIA cuBLAS
NVIDIA cuRAND
NVIDIA cuSPARSE
NVIDIA NPP
GPU VSIPL
Vector Signal Image Processing
CULA tools
GPU Accelerated Linear Algebra
MAGMA
Matrix Algebra on GPU and Multicore
NVIDIA cuFFT
ROGUE WAVE SOFTWARE
IMSL Library
CUSP
Sparse Linear Algebra
libjacket
Building-block Algorithms for CUDA
Thrust
C++ STL Features for CUDA

CUDA kernel

```
__global__ void PPI (float *d_image, float *d_random, int *d_res_partial,
int num_lines, int num_samples, int num_bands)
{

int idx = blockDim.x * blockIdx.x+threadIdx.x;
float pemax; // Maximum value of dot product
float pemin; // Minimum value of dot product
float pe;    // Scalar product

int v,d; int imax = 0; int imin = 0; pemax = MIN_INT; pemin = MAX_INT;

__shared__ float s_pixels[Tam_Vector]; float l_rand[224];

//Copy a skewer from GPU global memory to GPU registers
for (int k=0; k < num_bands; k++){
      l_rand[k] = d_random[idx*num_bands+k];
}

for (int it = 0; it < num_lines*num_samples/N_Pixels; it++){

      //Copy N_Pixels pixels to shared memory
      if (threadIdx.x < N_Pixels){
      for (int j=0; j<num_bands; j++){
            s_pixels[threadIdx.x+N_Pixels*j] =
            d_imagen[(it*N_Pixels+threadIdx.x)+(num_lines*num_samples*j)];
      }
}

__syncthreads();

//For each pixel
for (v=0; v < N_Pixels; v++){

      //Calculate dot product
      pe = 0;
      for (d = 0; d < num_bands; d++){
            pe = pe + l_rand[d]*s_pixels[v+N_Pixels*d];
```

OpenACC Directives

- Compiler directives specify parallel regions
- OpenACC compilers handle data between host and accelerators
- Intent is to be Portable (Ind of OS, CPU/accelerators vendor)
- High-level programming: accelerator and data transfer abstraction
- Will merge with OpenMP (at some point)

Syntax

C

#pragma acc *directive [clause [[,] clause]...] new-line*

Fortran

!$acc *directive [clause [[,] clause]...]*

Parallel Construct

#pragma acc parallel [clause [[,] clause]...] new-line

Data Constructs

#pragma acc data [clause [[,] clause]...] new-line

Loop Constructs

#pragma acc loop [clause [[,] clause]...]new-line

Calculate Pi using OpenACC

```
1  program picalc
2        implicit none
3        integer, parameter :: n=1000000
4        integer :: i
5        real(kind=8) :: t, pi
6        pi = 0.0
7     !$acc parallel loop
8        do i=0, n-1
9          t = (i+0.5)/n
10         pi = pi + 4.0/(1.0 + t*t)
11       end do
12    !$acc end parallel loop
13       print *, 'pi=', pi/n
14 end program picalc
15
```

OpenMP Example

```
1  /* matrix-omp.c */
2  #define SIZE 1000
3  float a[SIZE][SIZE];
4  float b[SIZE][SIZE];
5  float c[SIZE][SIZE];
6
7  int main()
8  {
9    int i,j,k;
10
11   // Initialize matrices.
12   for (i = 0; i < SIZE; ++i) {
13     for (j = 0; j < SIZE; ++j) {
14       a[i][j] = (float)i + j;
15       b[i][j] = (float)i - j;
16       c[i][j] = 0.0f;
17     }
18   }
19
20   // Compute matrix multiplication.
21 #pragma omp parallel for default(none) shared(a,b,c) private(i,j,k)
22   for (i = 0; i < SIZE; ++i) {
23     for (j = 0; j < SIZE; ++j) {
24       for (k = 0; k < SIZE; ++k) {
25     c[i][j] += a[i][k] * b[k][j];
26       }
27     }
28   }
29   return 0;
30 }
```

OpenACC Example

```
1  /* matrix-acc.c */
2  #define SIZE 1000
3  float a[SIZE][SIZE];
4  float b[SIZE][SIZE];
5  float c[SIZE][SIZE];
6
7  int main()
8  {
9    int i,j,k;
10
11   // Initialize matrices.
12   for (i = 0; i < SIZE; ++i) {
13     for (j = 0; j < SIZE; ++j) {
14       a[i][j] = (float)i + j;
15       b[i][j] = (float)i - j;
16       c[i][j] = 0.0f;
17     }
18   }
19
20   // Compute matrix multiplication.
21 #pragma acc kernels copyin(a,b) copy(c)
22   for (i = 0; i < SIZE; ++i) {
23     for (j = 0; j < SIZE; ++j) {
24       for (k = 0; k < SIZE; ++k) {
25     c[i][j] += a[i][k] * b[k][j];
26       }
27     }
28   }
29   return 0;
30 }
```

Easy Enough Right?

```c
/* matrix-acc.c */
#define SIZE 1000
float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];

int main()
{
  int i,j,k;

  // Initialize matrices.
  for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
      a[i][j] = (float)i + j;
      b[i][j] = (float)i - j;
      c[i][j] = 0.0f;
    }
  }

  // Compute matrix multiplication.
#pragma acc kernels copyin(a,b) copy(c)
  for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
      for (k = 0; k < SIZE; ++k) {
    c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return 0;
}
```

```c
/* matrix-omp.c */
#define SIZE 1000
float a[SIZE][SIZE];
float b[SIZE][SIZE];
float c[SIZE][SIZE];

int main()
{
  int i,j,k;

  // Initialize matrices.
  for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
      a[i][j] = (float)i + j;
      b[i][j] = (float)i - j;
      c[i][j] = 0.0f;
    }
  }

  // Compute matrix multiplication.
#pragma omp parallel for default(non
  for (i = 0; i < SIZE; ++i) {
    for (j = 0; j < SIZE; ++j) {
      for (k = 0; k < SIZE; ++k) {
    c[i][j] += a[i][k] * b[k][j];
      }
    }
  }
  return 0;
}
```

Jacobi Relaxation

```
iter = 0
do while ( err .gt tol .and. iter .gt. iter_max )

  iter = iter + 1
  err = 0.0

  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * (A(i+1,j) + A(i-1,j) + A(i,j-1) + A(i, j+1)
      err = max( err, abs(Anew(i,j)-A(i,j)) )
    end do
  end do

  if( mod(iter,100).eq.0 .or. iter.eq.1 ) print*, iter, err
  A = Anew
end do
```

Iterate until converged

Iterate across elements of matrix

Calculate new value from neighbours

OpenMP CPU Implementation

```
iter = 0
do while ( err .gt tol .and. iter .gt. iter_max )

  iter = iter + 1
  err = 0.0
!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * (A(i+1,j) + A(i-1,j) + A(i,j-1) + A(i,  j+1)
      err = max( err, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
!$omp end parallel do
  if( mod(iter,100).eq.0 ) print*, iter, err
  A = Anew
end do
```

Parallelise code inside region

Close off region

OpenMP CPU Implementation

```
iter = 0
do while ( err .gt tol .and. iter .gt. iter_max )

  iter = iter + 1
  err = 0.0
!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * (A(i+1,j) + A(i-1,j) + A(i,j-1) + A(i,  j+1)
      err = max( err, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
!$omp end parallel do
  if( mod(iter,100).eq.0 ) print*, iter, err
  A = Anew
end do
```

Parallelise code inside region

Close off region

Improved OpenACC GPU Implementation

```
!$acc data copyin(A), copyout(Anew)
iter = 0
do while ( err .gt tol .and. iter .gt. iter_max )

  iter = iter + 1
  err = 0.0
!$acc parallel reduction( max:err )
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * ( A(i+1,j  ) + A(i-1,j  ) &
                           A(i,  j-1) + A(i,  j+1)
      err = max( err, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
!$acc end parallel
  if( mod(iter,100).eq.0 ) print*, iter, err
  A = Anew
end do
!$acc end data
```

Reduced data movement

Improved OpenACC GPU Implementation

```
!$acc data copyin(A), copyout(Anew)
iter = 0
do while ( err .gt tol .and. iter .gt. iter_max )

  iter = iter + 1
  err = 0.0
!$acc parallel reduction( max:err )
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * ( A(i+1,j  ) + A(i-1,j  ) &
                           A(i,  j-1) + A(i,  j+1)
      err = max( err, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
!$acc end parallel
  if( mod(iter,100).eq.0 ) print*, iter, err
  A = Anew
end do
!$acc end data
```

Reduced data movement

More Performance

```
!$acc data copyin(A), create(Anew)
iter = 0
do while ( err .gt. tol .and. iter .gt. iter_max )

  iter = iter + 1
  err = 0.0
!$acc kernels loop reduction( max:err ), gang(32), worker(8)
  do j=1,m
    do i=1,n
      Anew(i,j) = 0.25 * ( A(i+1,j  ) + A(i-1,j  ) &
                           A(i,  j-1) + A(i,  j+1) )
      err = max( err, abs(Anew(i,j)-A(i,j)) )
    end do
  end do
!$acc end kernels loop
  if( mod(iter,100).eq.0 ) print*, iter, err
!$acc parallel
  A = Anew
!$acc end parallel
end do
!$acc end data
```

30% faster than default schedule

Restrictions - Fortran

- Upper bound for the last dimension of an assumed-size dummy array must be specified.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- Variables or arrays with derived type are treated specially (see manual)
- Arrays must be contiguous

Restrictions – C/C++

- C and C++: the length for a dynamically allocated array must be explicitly specified.
- The compiler may pad dimensions of arrays on the accelerator to improve memory alignment and program performance.
- Variables or arrays of struct or class type are treated specially (see manual)

Summary

- Easy to use
- Obtain free trial of PGI compiler at

http://www.pgroup.com

Execution	Time (s)	Speedup vs. 1 CPU thread	Speedup vs. 4 CPU threads
CPU 1 thread	34.14	--	--
CPU 4 threads	21.16	1.61x	1.0x
GPU (OpenACC)	5.32	6.42x	3.98x