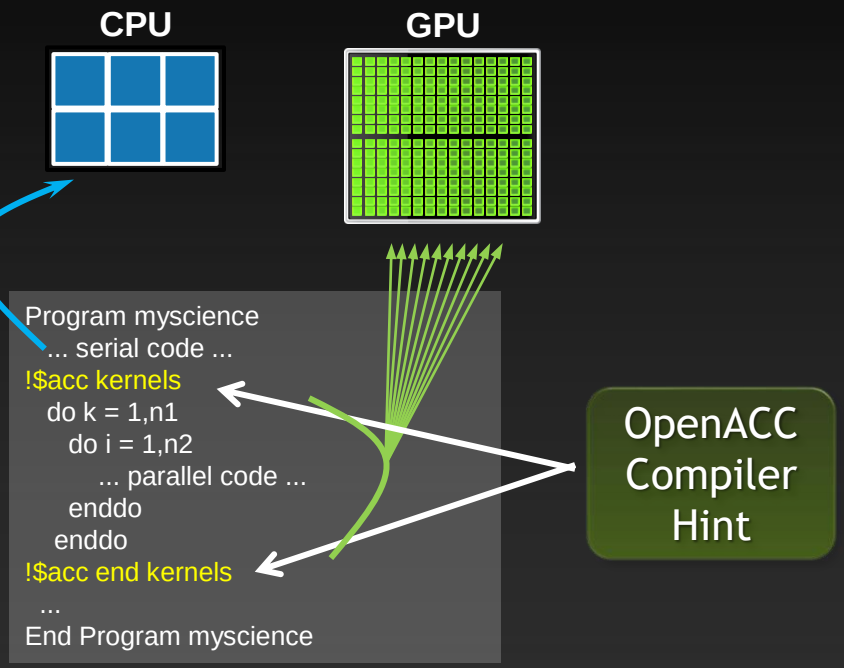
Introduction to OpenACC

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

What is OpenACC?

It is a directive based standard to allow developers to take advantage of accelerators such as GPUs from NVIDIA and AMD, Intel's Xeon Phi, FPGAs, and even DSP chips.

Directives



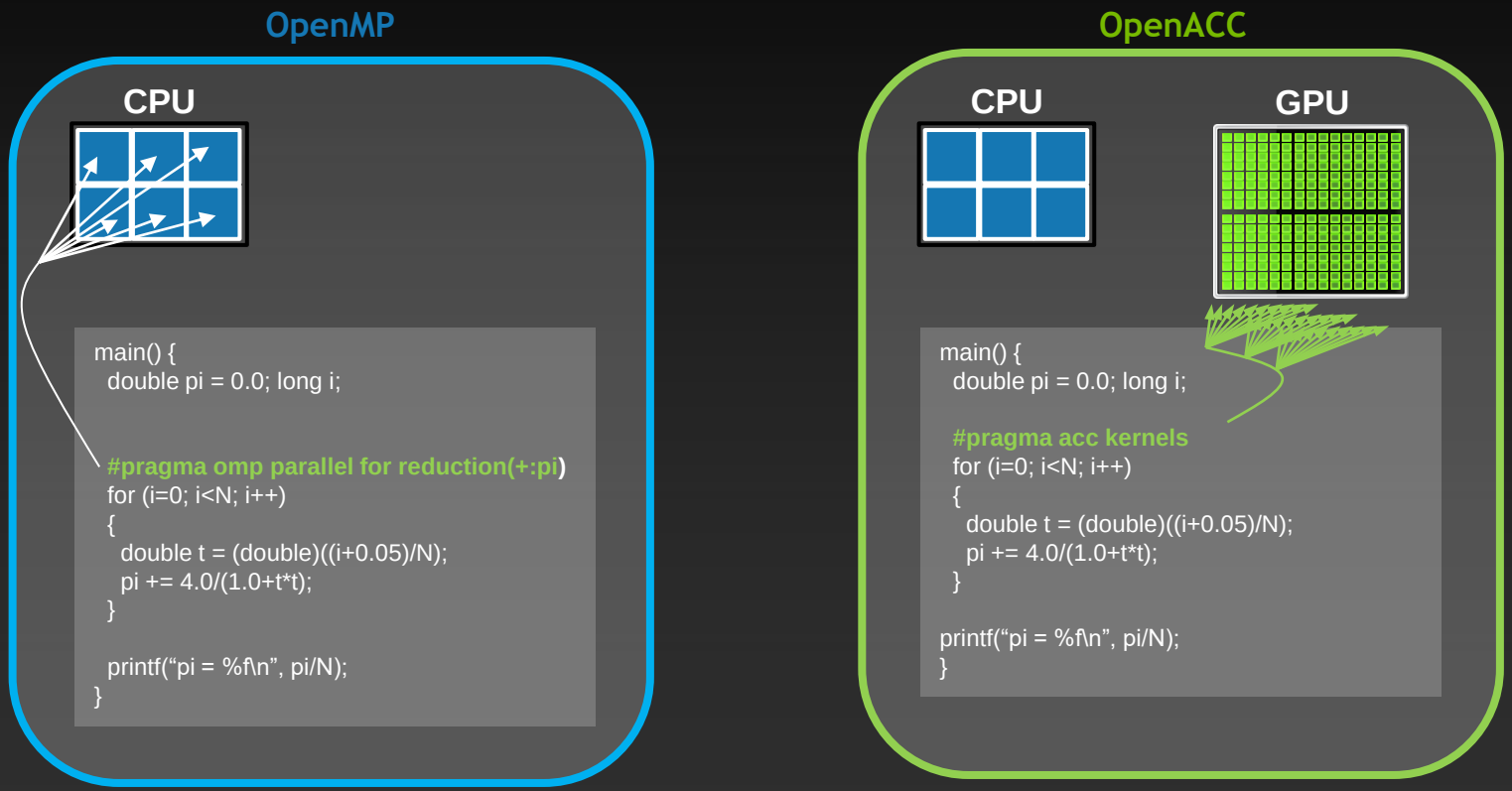
Your original
Fortran or C code

Simple compiler hints from coder.

Compiler generates parallel threaded code.

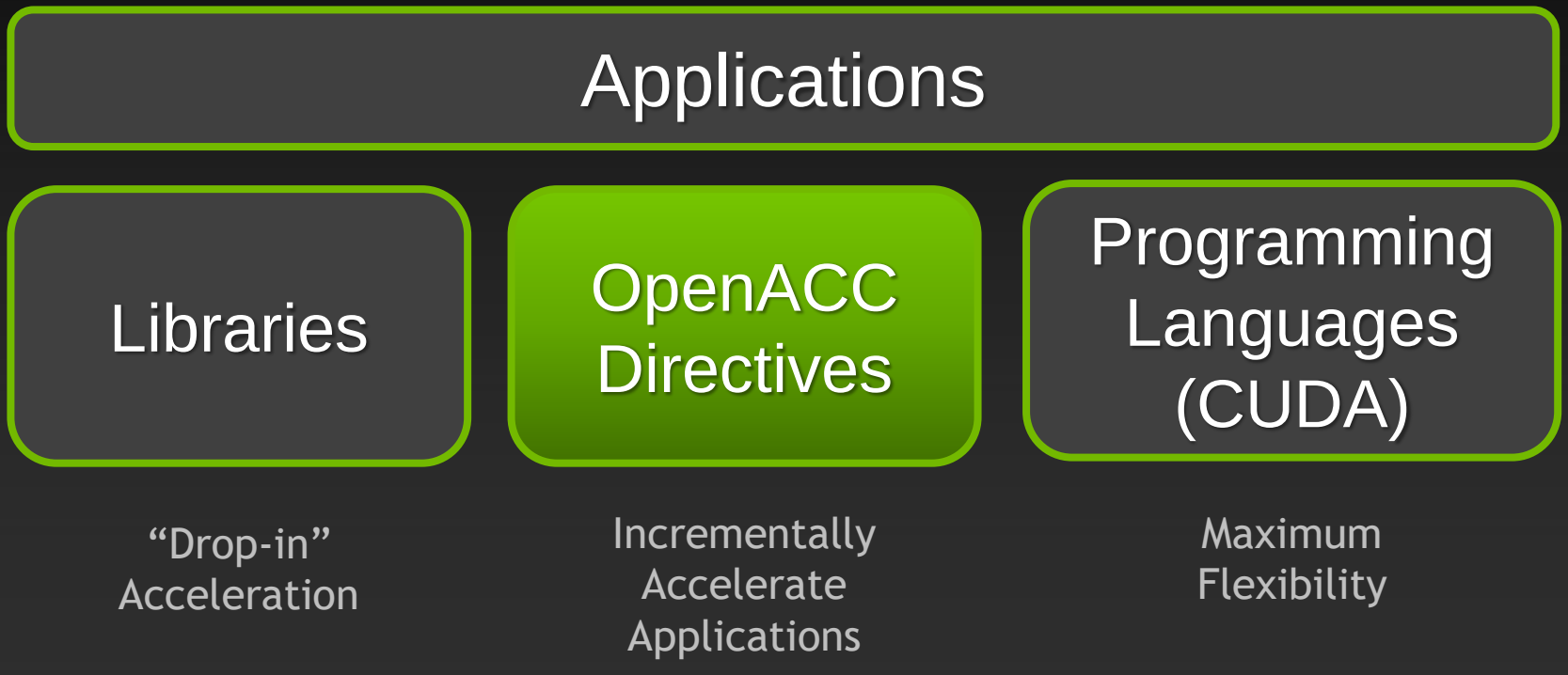
Ignorant compiler just sees some comments.

Familiar to OpenMP Programmers



More on this later!

How Else Would We Accelerate Applications?



Key Advantages Of This Approach

- High-level. No involvement of OpenCL, CUDA, etc.
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial; non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms.
- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be *quick*.

True Standard

- Full OpenACC specifications (now on 3.2) available online

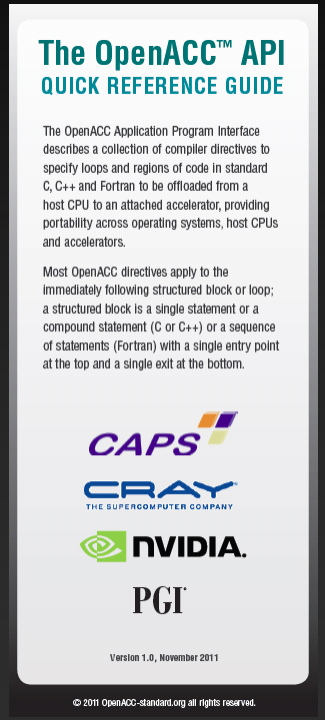
  http://www.openacc-standard.org

- Quick reference card also available and useful

- Implementations available now from PGI, Cray, CAPS and GCC.

- GCC version of OpenACC started in 5.x, but use at least 10.x

- Best free option is very probably PGI Community version:
  http://www.pgroup.com/products/community.htm



OPENACC Resources

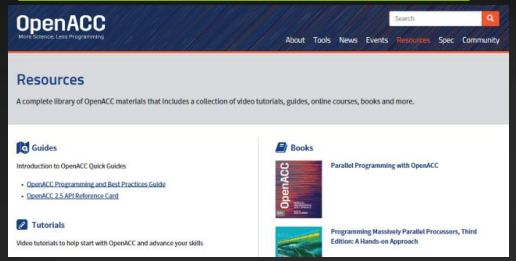
Guides ● Talks ● Tutorials ● Videos ● Books ● Spec ● Code Samples ● Teaching Materials ● Events ● Success Stories ● Courses ● Slack ● Stack Overflow

FREE Compilers

PGI® Community EDITION

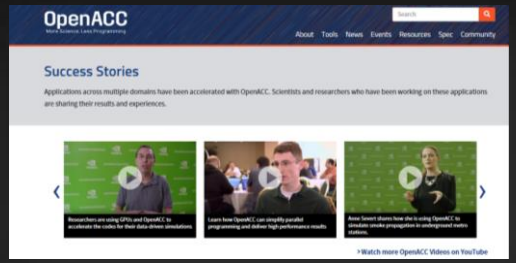
Resources

https://www.openacc.org/resources



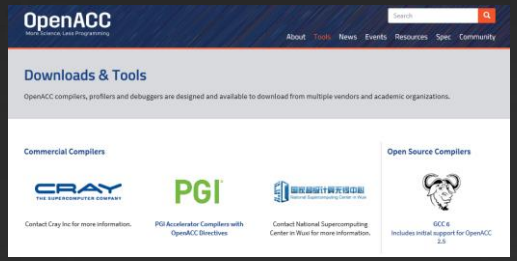
Success Stories

https://www.openacc.org/success-stories



Compilers and Tools

https://www.openacc.org/tools



Events

https://www.openacc.org/events

PITTSBURGH SUPERCOMPUTING CENTER

Serious Adoption

NEW PLATFORMS

OpenPOWER™

Sunway TaihuLight
Built around
OpenACC

GROWING COMMUNITY

- 6,000+ enabled developers
- Hackathons constantly
- Diverse online community

PORTING SUCCESS

- Five of 13 CAAR codes using OpenACC
- Gaussian ported to Tesla with OpenACC
- FLUENT using OpenACC in R18 production release

A Few Cases

Reading DNA nucleotide sequences

Shanghai JiaoTong University

4 directives

16x faster

Designing circuits for quantum computing

UIST, Macedonia

1 week

40x faster

Extracting image features in real-time

Aselsan

3 directives

4.1x faster

HydroC- Galaxy Formation

PRACE Benchmark Code, CAPS

1 week

3x faster

Real-time Derivative Valuation

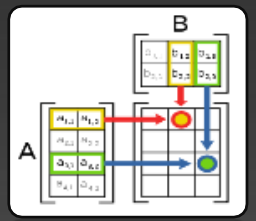
Opel Blue, Ltd

Few hours

70x faster

Matrix Matrix Multiply

Independent Research Scientist



4 directives

6.4x faster

A Champion Case

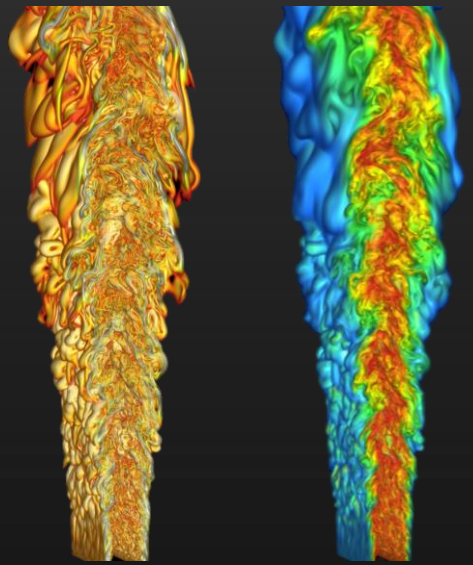
4x Faster

Jaguar	Titan
42 days	10 days

Modified <1%
Lines of Code

15 PF! One of fastest simulations ever!

Design alternative fuels with up to 50% higher efficiency



S3D: Fuel Combustion

PITTSBURGH SUPERCOMPUTING CENTER

A Simple Example: SAXPY

SAXPY in C

```
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];
}

...
// Somewhere in main
// call SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(:), y(:), a
  integer :: n, i
!$acc kernels
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end kernels
end subroutine saxpy


...
$ From main program
$ call SAXPY on 1M elements
call saxpy(2**20, 2.0, x_d, y_d)
...
```

kernels: Our first OpenACC Directive

We request that each loop execute as a separate *kernel* on the GPU. This is an incredibly powerful directive.

```
!$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

Kernel:
A parallel routine to run on the GPU

General Directive Syntax and Scope

Fortran

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

C

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

I may indent the directives at the natural code indentation level for readability. It is a common practice to always start them in the first column (ala #define/#ifdef). Either is fine with C or Fortran 90 compilers.

Complete SAXPY Example Code

```
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;
}
```

```
#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];
}
```

"I promise y is not aliased by Anything else (esp. x)"

C Detail: the `restrict` keyword

- Standard C (as of C99).
- Important for optimization of serial as well as OpenACC and OpenMP code.
- Promise given by the programmer to the compiler for a pointer

```
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

Compile and Run

- C: `nvc -acc -Minfo=accel saxpy.c`
- Fortran: `nvfortran -acc -Minfo=accel saxpy.f90`

Compiler Output

```
nvc -acc -Minfo=accel saxpy.c
saxpy:
      8, Generating copyin(x[:n-1])
         Generating copy(y[:n-1])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      9, Loop is parallelizable
         Generating Tesla code
          9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
             CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

- Run: `a.out`

Compare: Partial CUDA C SAXPY Code

Just the subroutine

```
__global__ void saxpy_kernel( float a, float* x, float* y, int n ){
  int i;
  i = blockIdx.x*blockDim.x + threadIdx.x;
  if( i <= n ) x[i] = a*x[i] + y[i];
}

void saxpy( float a, float* x, float* y, int n ){
  float *xd, *yd;
  cudaMalloc( (void**)&xd, n*sizeof(float) );
  cudaMalloc( (void**)&yd, n*sizeof(float) ); cudaMemcpy( xd, x, n*sizeof(float),
                    cudaMemcpyHostToDevice );
  cudaMemcpy( yd, y, n*sizeof(float),
                    cudaMemcpyHostToDevice );
  saxpy_kernel<<< (n+31)/32, 32 >>>( a, xd, yd, n );
  cudaMemcpy( x, xd, n*sizeof(float),
                    cudaMemcpyDeviceToHost );
  cudaFree( xd ); cudaFree( yd );
}
```

Compare: Partial CUDA Fortran SAXPY Code

Just the subroutine

```
module kmod
 use cudafor
contains
 attributes(global) subroutine saxpy_kernel(A,X,Y,N)
  real(4), device :: A, X(N), Y(N)
  integer, value :: N
  integer :: i
  i = (blockidx%x-1)*blockdim%x + threadidx%x
  if( i <= N ) X(i) = A*X(i) + Y(i)
 end subroutine
end module

 subroutine saxpy( A, X, Y, N )
  use kmod
  real(4) :: A, X(N), Y(N)
  integer :: N
  real(4), device, allocatable, dimension(:):: &
                Xd, Yd
  allocate( Xd(N), Yd(N) )
  Xd = X(1:N)
  Yd = Y(1:N)
  call saxpy_kernel<<<(N+31)/32,32>>>(A, Xd, Yd, N)
  X(1:N) = Xd
  deallocate( Xd, Yd )
 end subroutine
```

Again: Complete SAXPY Example Code

Main Code

```
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;
}
```

Entire Subroutine

```
#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];
}
```

Big Difference!

- With CUDA, we changed the structure of the old code. Non-CUDA programmers can't understand new code. It is not even ANSI standard code.
- We have separate sections for the host code and the GPU code. Different flow of code. Serial path now gone forever.
- Where did these "32"s and other mystery numbers come from? This is a clue that we have some hardware details to deal with here.
- Exact same situation as assembly used to be. How much hand-assembled code is still being written in HPC now that compilers have gotten so efficient?

This looks easy! Too easy…

- If it is this simple, why don't we just throw ***kernel*** in front of every loop?
- Better yet, why doesn't the compiler do this for me?

The answer is that there are two general issues that prevent the compiler from being able to just automatically parallelize every loop.

- Data Dependencies in Loops
- Data Movement

The compiler needs your higher level perspective (in the form of directive hints) to get correct results and reasonable performance.

Data Dependencies

Most directive based parallelization consists of splitting up big do/for loops into independent chunks that the many processors can work on simultaneously.

Take, for example, a simple for loop like this:

```
for(index=0; index<1000000; index++)
        Array[index] = 4 * Array[index];
```

When run on 1000 processors, it will execute something like this…

No Data Dependency

Processor 0

```
for(index=0, index<999,index++)
   Array[index] = 4*Array[index];
```

Processor 1

```
for(index=1000, index<1999,index++)
   Array[index] = 4*Array[index];
```

Processor 2

```
for(index=2000, index<2999,index++)
   Array[index] = 4*Array[index];
```

Processor 3

```
for(index=3000, index<3999,index++)
   Array[index] = 4*Array[index];
```

Processor 4

```
for(index=4000, index<4999,index++)
   Array[index] = 4*Array[index];
```

Data Dependency

But what if the loops are not entirely independent?

Take, for example, a similar loop like this:

```
for(index=1; index<1000000; index++)
        Array[index] = 4 * Array[index] – Array[index-1];
```

This is perfectly valid serial code.

Data Dependency

Now Processor 1, in trying to calculate its first iteration…

```
for(index=1000; index<1999; index++)
      Array[1000] = 4 * Array[1000] – Array[999];
```

needs the result of Processor 0's last iteration. If we want the correct ("same as serial") result, we need to wait until processor 0 finishes. Likewise for processors 2, 3, …

Data Dependencies

That is a data dependency. If the compiler even ***suspects*** that there is a data dependency, it will, for the sake of correctness, refuse to parallelize that loop with *kernels*.

```
11, Loop carried dependence of 'Array' prevents parallelization
    Loop carried backward dependence of 'Array' prevents vectorization
```

As large, complex loops are quite common in HPC, especially around the most important parts of your code, the compiler will often balk most when you most need a kernel to be generated. What can you do?

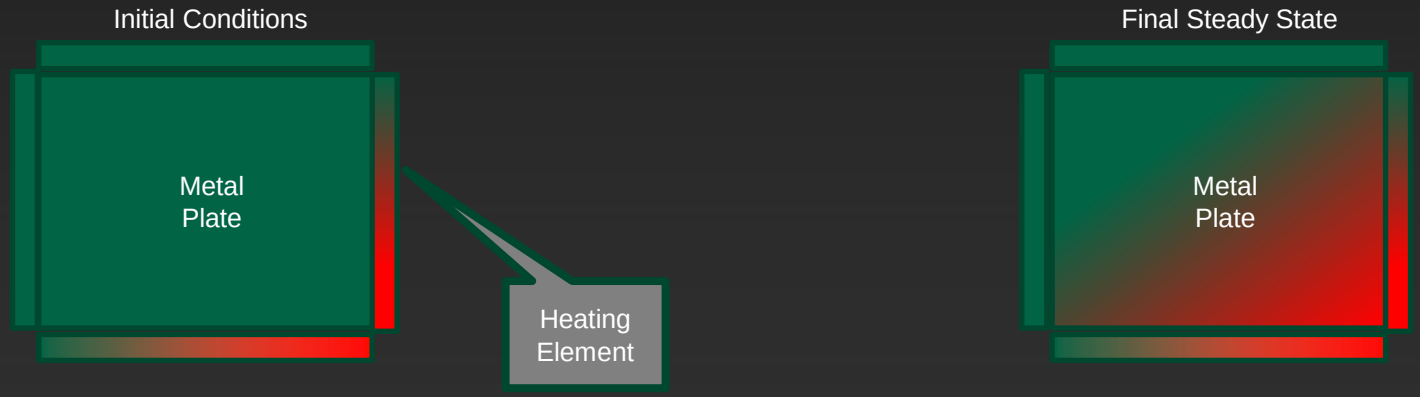
Data Dependencies

- Rearrange your code to make it more obvious to the compiler that there is not really a data dependency.
- Eliminate a real dependency by changing your code.
  - There is a common bag of tricks developed for this as this issue goes back 40 years in HPC. Many are quite trivial to apply.
  - The compilers have gradually been learning these themselves.
- Override the compiler's judgment (`independent` clause) at the risk of invalid results. Misuse of `restrict` has similar consequences.

Our Foundation Exercise: Laplace Solver

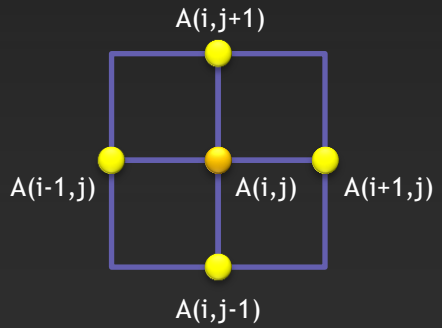
- I've been using this for MPI, OpenMP and now OpenACC. It is a great simulation problem, not rigged for OpenACC.
- In this most basic form, it solves the Laplace equation: $\nabla^2 f(x, y) = 0$
- The Laplace Equation applies to many physical problems, including:
  - Electrostatics
  - Fluid Flow
  - Temperature

- For temperature, it is the Steady State Heat Equation:



Exercise Foundation: Jacobi Iteration

- The Laplace equation on a grid states that each grid point is the average of it's neighbors.
- We can iteratively converge to that state by repeatedly computing new values at each point from the average of neighboring points.
- We just keep doing this until the difference from one pass to the next is small enough for us to tolerate.



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

Serial Code Implementation

```
for(i = 1; i <= ROWS; i++) {
     for(j = 1; j <= COLUMNS; j++) {
          Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                      Temperature_last[i][j+1] + Temperature_last[i][j-1]);
     }
}
```

```
do j=1,columns
     do i=1,rows
          temperature(i,j)= 0.25 * (temperature_last(i+1,j)+temperature_last(i-1,j) + &
                                    temperature_last(i,j+1)+temperature_last(i,j-1) )
     enddo
enddo
```

Serial C Code (kernel)

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

     for(i = 1; i <= ROWS; i++) {
          for(j = 1; j <= COLUMNS; j++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

     for(i = 1; i <= ROWS; i++){
          for(j = 1; j <= COLUMNS; j++){
               dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
               Temperature_last[i][j] = Temperature[i][j];
          }
     }

     if((iteration % 100) == 0) {
          track_progress(iteration);
     }

     iteration++;

}
```

Done?

Calculate

Update temp array and find max change

Output

Serial C Code Subroutines

```
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}
```

BCs could run from 0 to ROWS+1 or from 1 to ROWS. We chose the former.

```
void track_progress(int iteration) {

  int i;

  printf("-- Iteration: %d --\n", iteration);
  for(i = ROWS-5; i <= ROWS; i++) {
     printf("[%d,%d]: %5.2f ", i, i,Temperature[i][i]);
  }
  printf("\n");
}
```

Whole C Code

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <sys/time.h>

// size of plate
#define COLUMNS    1000
#define ROWS       1000

// largest permitted change in temp (This value takes about 3400 steps)
#define MAX_TEMP_ERROR 0.01

double Temperature[ROWS+2][COLUMNS+2];      // temperature grid
double Temperature_last[ROWS+2][COLUMNS+2]; // temperature grid from last iteration

//   helper routines
void initialize();
void track_progress(int iter);

int main(int argc, char *argv[]) {

    int i, j;                                            // grid indexes
    int max_iterations;                                  // number of iterations
    int iteration=1;                                     // current iteration
    double dt=100;                                       // largest change in t
    struct timeval start_time, stop_time, elapsed_time;  // timers

    printf("Maximum iterations [100-4000]?\n");
    scanf("%d", &max_iterations);

    gettimeofday(&start_time,NULL); // Unix timer

    initialize();                   // initialize Temp_last including boundary conditions

    // do until error is minimal or until max steps
    while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

        // main calculation: average my four neighbors
        for(i = 1; i <= ROWS; i++) {
            for(j = 1; j <= COLUMNS; j++) {
                Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                            Temperature_last[i][j+1] + Temperature_last[i][j-1]);
            }
        }

        dt = 0.0; // reset largest temperature change

        // copy grid to old grid for next iteration and find latest dt
        for(i = 1; i <= ROWS; i++){
            for(j = 1; j <= COLUMNS; j++){
              dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
              Temperature_last[i][j] = Temperature[i][j];
            }
        }

        // periodically print test values
        if((iteration % 100) == 0) {
            track_progress(iteration);
        }

        iteration++;
    }

    gettimeofday(&stop_time,NULL);
    timersub(&stop_time, &start_time, &elapsed_time); // Unix time subtract routine

    printf("\nMax error at iteration %d was %f\n", iteration-1, dt);
    printf("Total time was %f seconds.\n", elapsed_time.tv_sec+elapsed_time.tv_usec/1000000.0);

}

// initialize plate and boundary conditions
// Temp_last is used to to start first iteration
void initialize(){

    int i,j;

    for(i = 0; i <= ROWS+1; i++){
        for (j = 0; j <= COLUMNS+1; j++){
            Temperature_last[i][j] = 0.0;
        }
    }

    // these boundary conditions never change throughout run

    // set left side to 0 and right to a linear increase
    for(i = 0; i <= ROWS+1; i++) {
        Temperature_last[i][0] = 0.0;
        Temperature_last[i][COLUMNS+1] = (100.0/ROWS)*i;
    }

    // set top to 0 and bottom to linear increase
    for(j = 0; j <= COLUMNS+1; j++) {
        Temperature_last[0][j] = 0.0;
        Temperature_last[ROWS+1][j] = (100.0/COLUMNS)*j;
    }
}

// print diagonal in bottom right corner where most action is
void track_progress(int iteration) {

    int i;

    printf("---------- Iteration number: %d ------------\n", iteration);
    for(i = ROWS-5; i <= ROWS; i++) {
        printf("[%d,%d]: %5.2f  ", i, i, Temperature[i][i]);
    }
    printf("\n");
}
```

Serial Fortran Code (kernel)

```
do while ( dt > max_temp_error .and. iteration <= max_iterations)

  do j=1,columns
     do i=1,rows
        temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                               temperature_last(i,j+1)+temperature_last(i,j-1) )
     enddo
  enddo

  dt=0.0

  do j=1,columns
     do i=1,rows
        dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
        temperature_last(i,j) = temperature(i,j)
     enddo
  enddo

  if( mod(iteration,100).eq.0 ) then
     call track_progress(temperature, iteration)
  endif

  iteration = iteration+1

enddo
```

Done?

Calculate

Update temp array and find max change

Output

Serial Fortran Code Subroutines

```
subroutine initialize( temperature_last )
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,j

      double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

      temperature_last = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature_last(i,0) = 0.0
         temperature_last(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature_last(0,j) = 0.0
         temperature_last(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize
```

```
subroutine track_progress(temperature, iteration)
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: temperature

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                  rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
      print *
```

Whole Fortran Code

```
program serial
      implicit none

      !Size of plate
      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      double precision, parameter    :: max_temp_error=0.01

      integer                        :: i, j, max_iterations, iteration=1
      double precision               :: dt=100.0
      real                           :: start_time, stop_time

      double precision, dimension(0:rows+1,0:columns+1) :: temperature, temperature_last

      print*, 'Maximum iterations [100-4000]?'
      read*,   max_iterations

      call cpu_time(start_time)      !Fortran timer

      call initialize(temperature_last)

      !do until error is minimal or until maximum steps
      do while ( dt > max_temp_error .and. iteration <= max_iterations)

         do j=1,columns
            do i=1,rows
               temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                      temperature_last(i,j+1)+temperature_last(i,j-1) )
            enddo
         enddo

         dt=0.0

         !copy grid to old grid for next iteration and find max change
         do j=1,columns
            do i=1,rows
               dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
               temperature_last(i,j) = temperature(i,j)
            enddo
         enddo

         !periodically print test values
         if( mod(iteration,100).eq.0 ) then
            call track_progress(temperature, iteration)
         endif

         iteration = iteration+1

      enddo

      call cpu_time(stop_time)

      print*, 'Max error at iteration ', iteration-1, ' was ',dt
      print*, 'Total time was ',stop_time-start_time, ' seconds.'

end program serial
```

```
! initialize plate and boundery conditions
! temp_last is used to to start first iteration
subroutine initialize( temperature_last )
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,j

      double precision, dimension(0:rows+1,0:columns+1) :: temperature_last

      temperature_last = 0.0

      !these boundary conditions never change throughout run

      !set left side to 0 and right to linear increase
      do i=0,rows+1
         temperature_last(i,0) = 0.0
         temperature_last(i,columns+1) = (100.0/rows) * i
      enddo

      !set top to 0 and bottom to linear increase
      do j=0,columns+1
         temperature_last(0,j) = 0.0
         temperature_last(rows+1,j) = ((100.0)/columns) * j
      enddo

end subroutine initialize

!print diagonal in bottom corner where most action is
subroutine track_progress(temperature, iteration)
      implicit none

      integer, parameter             :: columns=1000
      integer, parameter             :: rows=1000
      integer                        :: i,iteration

      double precision, dimension(0:rows+1,0:columns+1) :: temperature

      print *, '---------- Iteration number: ', iteration, ' ---------------'
      do i=5,0,-1
         write (*,'("("i4,",",i4,"):",f6.2,"  ")',advance='no'), &
                    rows-i,columns-i,temperature(rows-i,columns-i)
      enddo
      print *
end subroutine track_progress
```

Exercises: General Instructions for Compiling

- Exercises are in the “Exercises/OpenACC” directory in your home directory
- Solutions are in the “Solutions” subdirectory
- To compile

```
nvc -acc laplace.c
nvfortran -acc laplace.f90
```

- This will generate the executable a.out

Exercises: Very useful compiler option

Adding -Minfo=accel to your compile command will give you some very useful information about how well the compiler was able to honor your OpenACC directives.

```
[urbanic@gpu017 Solutions]$ nvc -acc -Minfo=accel laplace_acc.c
main:
     59, Generating create(Temperature[:][:]) [if not already present]
         Generating copy(Temperature_last[:][:]) [if not already present]
     64, Loop is parallelizable
     65, Loop is parallelizable
         Generating Tesla code
         64, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         65, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     75, Loop is parallelizable
     76, Loop is parallelizable
         Generating Tesla code
         75, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         76, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         77, Generating implicit reduction(max:dt)
     85, Generating update self(Temperature[:][:])
```

Special Instructions for Running on the GPUs (during this workshop)

As mentioned, on Bridges2 you generally only have to use the queueing system when you want to. However, as we have hundreds of you wanting quick turnaround, we will have to use it today.

Once you have an a.out that you want to run, you should use the simple job that we have already created (in Exercises/OpenACC) for you to run:

```
fred@bridges2-login011$ sbatch gpu.job
```

Output From Your Batch Job

The machine will tell you it submitted a batch job, and you can await your output, while will come back in a file with the corresponding number as a name:

slurm-138555.out

As everything we are doing this afternoon only requires a few minutes at most (and usually just seconds), you could just sit there and wait for the file to magically appear. At which point you can "more" it or review it with your editor.

Changing Things Up

If you get impatient, or want to see what the machine us up to, you can look at the situation with squeue.

You might wonder what happened to the interaction count that the user is prompted for. I stuck a reasonable default (4000 iterations) into the job file. You can edit it if you want to. The whole job file is just a few lines.

Congratulations, you are now a Batch System veteran. Welcome to supercomputing.

Exercise 1: Using kernels to parallelize the main loops

(About 20 minutes)

Q: Can you get a speedup with just the kernels directives?

1. Edit *laplace_serial.c*/f90
   1. Maybe copy your intended OpenACC version to *laplace_acc.c* to start
   2. Add directives where it helps

2. Compile with OpenACC parallelization
   1. nvc -acc -Minfo=accel laplace_acc.c or
      nvfortran -acc -Minfo=accel laplace_acc.f90
   2. Look at your compiler output to make sure you are having an effect

3. Run
   1. sbatch gpu.job (Leave it at 4000 iterations if you want a solution that converges to current tolerance)
   2. Look at output in file that returns (something like slurm-138555.out)
   3. Compare the serial and your OpenACC version for performance difference

Exercise 1 C Solution

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
            dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
            Temperature_last[i][j] = Temperature[i][j];
        }
    }

    if((iteration % 100) == 0) {
        track_progress(iteration);
    }

    iteration++;
}
```

Generate a GPU kernel

Generate a GPU kernel

Exercise 1 Fortran Solution

```
do while ( dt > max_temp_error .and. iteration <= max_iterations)

    !$acc kernels
    do j=1,columns
       do i=1,rows
          temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                 temperature_last(i,j+1)+temperature_last(i,j-1) )
       enddo
    enddo
    !$acc end kernels

    dt=0.0

    !$acc kernels
    do j=1,columns
       do i=1,rows
          dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
          temperature_last(i,j) = temperature(i,j)
       enddo
    enddo
    !$acc end kernels

    if( mod(iteration,100).eq.0 ) then
       call track_progress(temperature, iteration)
    endif

    iteration = iteration+1

enddo
```

Generate a GPU kernel

Generate a GPU kernel

Exercise 1: Compiler output (C)

```
[urbanic@gpu017 Solutions]$ nvc -acc -Minfo=accel laplace_acc.c
main:
     59, Generating create(Temperature[:][:]) [if not already present]
         Generating copy(Temperature_last[:][:]) [if not already present]
     64, Loop is parallelizable
     65, Loop is parallelizable
         Generating Tesla code
         64, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         65, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
     75, Loop is parallelizable
     76, Loop is parallelizable
         Generating Tesla code
         75, #pragma acc loop gang, vector(4) /* blockIdx.y threadIdx.y */
         76, #pragma acc loop gang, vector(32) /* blockIdx.x threadIdx.x */
         77, Generating implicit reduction(max:dt)
     85, Generating update self(Temperature[:][:])
```

Compiler was able to parallelize

Compiler was able to parallelize

First, about that “reduction”

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
     #pragma acc kernels
     for(i = 1; i <= ROWS; i++) {
          for(j = 1; j <= COLUMNS; j++) {
               Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                           Temperature_last[i][j+1] + Temperature_last[i][j-1]);
          }
     }

     dt = 0.0;

     #pragma acc loop reduction (max:dt)
     for(i = 1; i <= ROWS; i++){
          for(j = 1; j <= COLUMNS; j++){
               dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
               Temperature_last[i][j] = Temperature[i][j];
          }
     }

     .
     .
     iteration++;
}
```

This explicitly declares the reduction.

Exiting this loop, each processor has a different idea of what the max dt is.

With *kernel* the compiler recognizes this and does a reduction, a very convenient thing. We can get too sophisticated for this *autoscoping* to happen.

Exercise 1: Performance

3372 steps to convergence

Execution	Time (s)	Speedup
CPU Serial	20.6	--
CPU 2 OpenMP threads	10.3	2.0
CPU 4 OpenMP threads	5.2	4.0
CPU 8 OpenMP threads	2.6	7.9
CPU 16 OpenMP threads	1.4	14.7
CPU 32 OpenMP threads	0.80	25.7
CPU 64 OpenMP threads	0.72	28.6
CPU 128 OpenMP threads	1.4	14.7
OpenACC GPU	32.4	0.6x

Using NVHPC 21.2 on a V100

What's with the OpenMP?

- We can compare our GPU results to the best the multi-core CPUs can do.

- If you are familiar with OpenMP, or even if you are not, you can compile and run the OpenMP enabled versions in your OpenMP directory as:

```
nvc -mp laplace_omp.c     or    nvfortran -mp laplace_omp.f90
```

then to run on 8 threads do:

```
export OMP_NUM_THREADS=8
a.out
```

- Note that you probably only have 8 real cores if you are still on a GPU node. Do something like "interact -n28" if you want a full node of cores.

What went wrong?

`export PGI_ACC_TIME=1` to activate profiling and run again:

```
Accelerator Kernel Timing data
/home/urbanic/laplace_bad_acc.c
  main  NVIDIA  devicenum=0
    time(us): 12,095,531
    62: compute region reached 3372 times
        64: kernel launched 3372 times
            grid: [32x250]  block: [32x4]
             device time(us): total=127,989 max=48 min=37 avg=37
            elapsed time(us): total=241,221 max=1,407 min=61 avg=71
    62: data region reached 6744 times
        62: data copyin transfers: 3372
             device time(us): total=2,446,765 max=972 min=712 avg=725
        70: data copyout transfers: 3372
             device time(us): total=2,098,635 max=835 min=616 avg=622
    73: compute region reached 3372 times
        73: data copyin transfers: 3372
             device time(us): total=32,465 max=71 min=6 avg=9
        75: kernel launched 3372 times
            grid: [32x250]  block: [32x4]
             device time(us): total=179,342 max=63 min=52 avg=53
            elapsed time(us): total=294,686 max=407 min=76 avg=87
        75: reduction kernel launched 3372 times
            grid: [1]  block: [256]
             device time(us): total=50,490 max=23 min=14 avg=14
            elapsed time(us): total=137,910 max=549 min=34 avg=40
        75: data copyout transfers: 3372
             device time(us): total=60,080 max=266 min=13 avg=17
    73: data region reached 6744 times
        73: data copyin transfers: 6744
             device time(us): total=5,004,411 max=1,005 min=716 avg=742
        82: data copyout transfers: 3372
             device time(us): total=2,095,354 max=854 min=616 avg=621
```

0.2 seconds

0.3 seconds

0.1 seconds

2.4 seconds

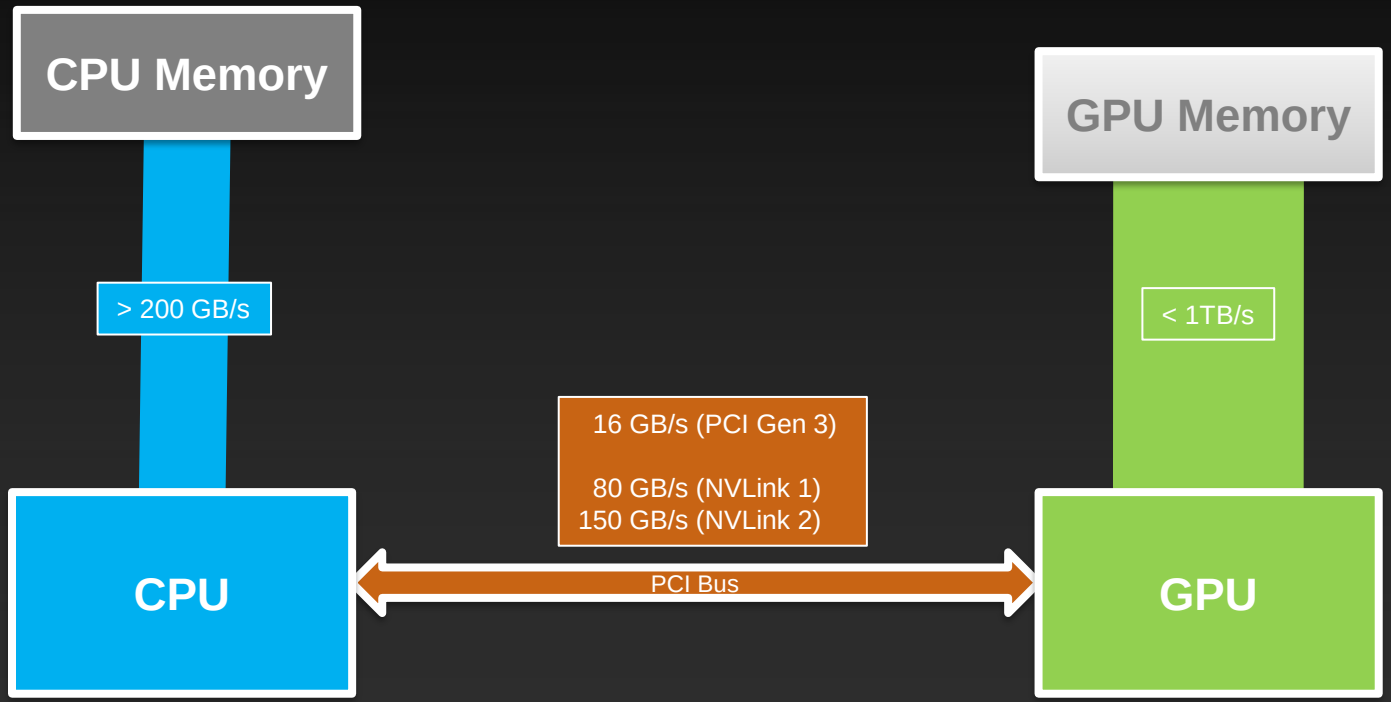
2.0 seconds

5.0 seconds

2.0 seconds

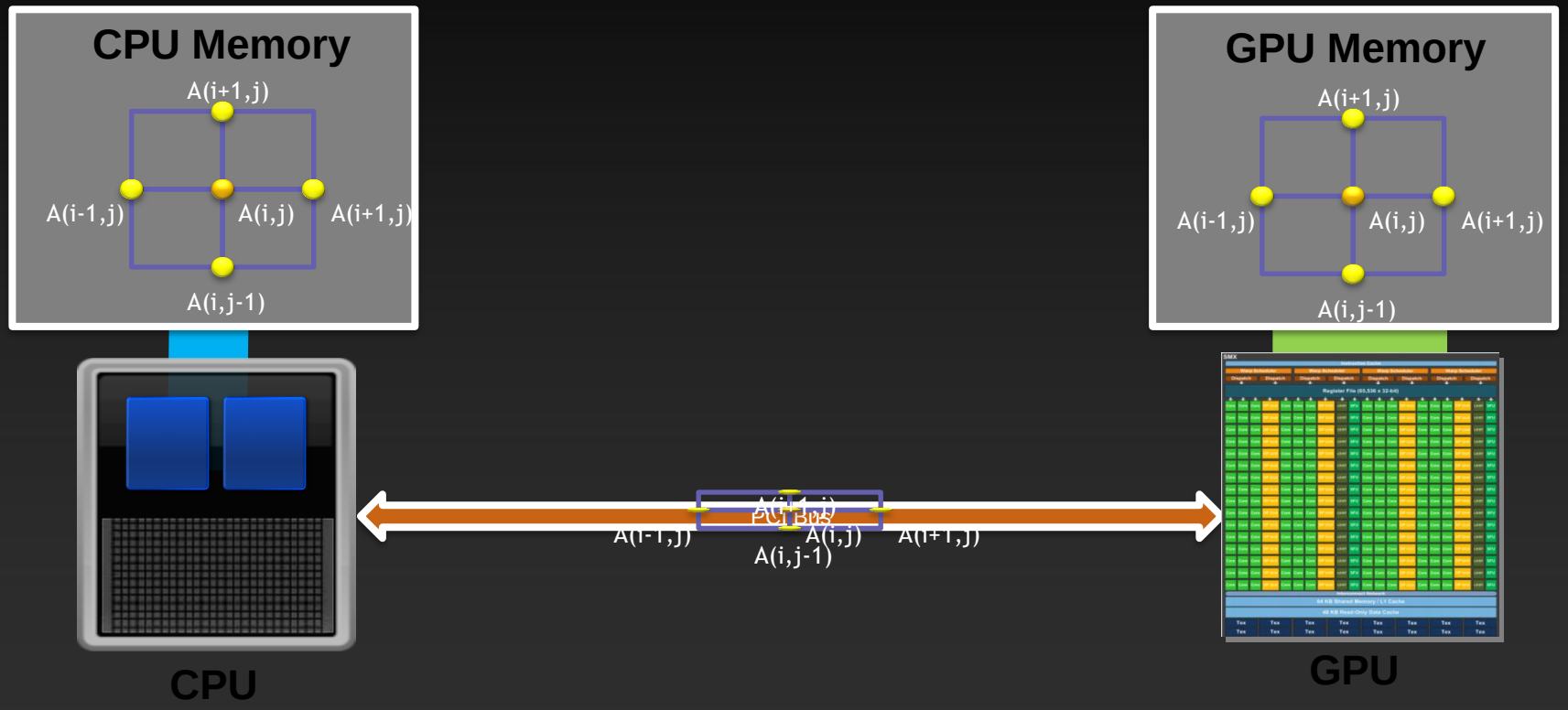
Basic Concept

Simplified, but sadly true



All bandwidths one-direction.

Multiple Times Each Iteration



Excessive Data Transfers

```
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {
```

Temperature, Temperature_old resident on host

Temperature, Temperature_old resident on device

```
#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
   for(j = 1; j <= COLUMNS; j++) {
       Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + …
   }
}
```

Temperature, Temperature_old resident on device

Temperature, Temperature_old resident on host

4 copies happen every iteration of the outer while loop!

```
dt = 0.0;
```

Temperature, Temperature_old resident on host

Temperature, Temperature_old resident on device

```
#pragma acc kernels
for(i = 1; i <= ROWS; i++) {
   for(j = 1; j <= COLUMNS; j++) {
       Temperature[i][j] = 0.25 * (Temperature_old[i+1][j] + …
```

Temperature, Temperature_old resident on device

Temperature, Temperature_old resident on host

```
}
```

Data Management

The First, Most Important, and Possibly Only OpenACC Optimization

Scoped Data Construct Syntax

Fortran

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

C

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

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. 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.
`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.

Array Shaping

- Compilers sometimes cannot determine the size of arrays, so we must specify explicitly using data clauses with an array “shape”. The compiler will let you know if you need to do this. Sometimes, you will want to for your own efficiency reasons.

- C

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

- Fortran

```
!$acc data copyin(a(1:size)), copyout(b(s/4:3*s/4))
```

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

Compiler will (increasingly) often make a good guess…

```
int main(int argc, char *argv[]) {

  int i;
  double A[2000], B[1000], C[1000];
    .
    .
    .
  #pragma acc kernels
  for (i=0; i<1000; i++){

    A[i] = 4 * i;
    B[i] = B[i] + 2;
    C[i] = A[i] + 2 * B[i];

  }
    .
    .
    .
}
```

Smarter

Smartest

```
nvc -acc -Minfo=accel loops.c
main:
   6, Generating present_or_copyout(C[:])
      Generating present_or_copy(B[:])
      Generating present_or_copyout(A[:1000])
      Generating NVIDIA code
   7, Loop is parallelizable
      Accelerator kernel generated
```

Data Regions Have Real Consequences

Simplest Kernel

```
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

With Global Data Region

```
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;
    }

    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

Data Regions Are Different Than Compute Regions

Compute Region

```
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;
   }

   A[10] = 2.0;

 }

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

Data Region

Output:

A[10] = 1.0

Data Movement Decisions

- Much like loop data dependencies, sometime the compiler needs your human intelligence to make high-level decisions about data movement. Otherwise, it must remain conservative - sometimes at great cost.

- You must think about when data truly needs to migrate, and see if that is better than the default.

- Besides the scope-based data clauses, there are OpenACC options to let us manage data movement more intensely or asynchronously. We could manage the above behavior with the `update` construct:

Fortran :
```
!$acc update [host(), device(), …]
```

C:
```
#pragma acc update [host(), device(), …]
```

Ex: `#pragma acc update host(Temp_array)  //Get host a copy from device`

Exercise 2: Use acc data to minimize transfers

(about 40 minutes)

Q: What speedup can you get with data + kernels directives?

- Start with your Exercise 1 solution or grab laplace_bad_acc.c/f90 from the Solutions subdirectory. This is just the solution of the last exercise.

- Add *data* directives where it helps.
  - Think: when *should* I move data between host and GPU? Think how you would do it by hand, then determine which data clauses will implement that plan.
  - Hint: you may find it helpful to ignore the output at first and just concentrate on getting the solution to converge quickly (at 3372 steps). Then worry about *updating* the printout.

Exercise 2 C Solution

```
#pragma acc data copy(Temperature_last, Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature)
        track_progress(iteration);
    }

    iteration++;
}
```

No data movement in this block.

Except once in a while here.

Exercise 2, Slightly better solution

```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature)
        track_progress(iteration);
    }

    iteration++;
}
```

Temperature is purely temporary.

Slightly better still solution

```
#pragma acc data copy(Temperature_last), create(Temperature)
while ( dt > MAX_TEMP_ERROR && iteration <= max_iterations ) {

    // main calculation: average my four neighbors
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++) {
        for(j = 1; j <= COLUMNS; j++) {
            Temperature[i][j] = 0.25 * (Temperature_last[i+1][j] + Temperature_last[i-1][j] +
                                        Temperature_last[i][j+1] + Temperature_last[i][j-1]);
        }
    }

    dt = 0.0; // reset largest temperature change

    // copy grid to old grid for next iteration and find latest dt
    #pragma acc kernels
    for(i = 1; i <= ROWS; i++){
        for(j = 1; j <= COLUMNS; j++){
          dt = fmax( fabs(Temperature[i][j]-Temperature_last[i][j]), dt);
          Temperature_last[i][j] = Temperature[i][j];
        }
    }

    // periodically print test values
    if((iteration % 100) == 0) {
        #pragma acc update host(Temperature[ROWS-4:5][COLUMNS-4:5])
        track_progress(iteration);
    }

    iteration++;
}
```

Only need corner elements.

Exercise 2 Fortran Solution

```
!$acc data copy(temperature_last), create(temperature)
do while ( dt > max_temp_error .and. iteration <= max_iterations)

   !$acc kernels
   do j=1,columns
      do i=1,rows
         temperature(i,j)=0.25*(temperature_last(i+1,j)+temperature_last(i-1,j)+ &
                                temperature_last(i,j+1)+temperature_last(i,j-1) )
      enddo
   enddo
   !$acc end kernels

   dt=0.0

   !copy grid to old grid for next iteration and find max change
   !$acc kernels
   do j=1,columns
      do i=1,rows
         dt = max( abs(temperature(i,j) - temperature_last(i,j)), dt )
         temperature_last(i,j) = temperature(i,j)
      enddo
   enddo
   !$acc end kernels

   !periodically print test values
   if( mod(iteration,100).eq.0 ) then
      !$acc update host(temperature)
      call track_progress(temperature, iteration)
   endif

   iteration = iteration+1

enddo
!$acc end data
```

Keep these on GPU

Extra efficient:

!$acc update host(temperature(columns-5:columns,rows-5:rows))

Except bring back a copy here

Exercise 2: Performance

3372 steps to convergence

Execution	Time (s)	Speedup
CPU Serial	20.6	--
CPU 2 OpenMP threads	10.3	2.0
CPU 4 OpenMP threads	5.2	4.0
CPU 8 OpenMP threads	2.6	7.9
CPU 16 OpenMP threads	1.4	14.7
CPU 32 OpenMP threads	0.80	25.7
CPU 64 OpenMP threads	0.72	28.6
CPU 128 OpenMP threads	1.4	14.7
OpenACC GPU	0.62	33

OpenACC or OpenMP?

Don't draw any grand conclusions yet. We have gotten impressive speedups from both approaches. But our problem size is pretty small. Our main data structure is:

1000 x 1000 = 1M elements = 8MB of memory

We have 2 of these (temperature and temperature_last) so we are using roughly 16 MB of memory. Not very large. When divided over cores it gets even smaller and can easily fit into cache.

The algorithm is realistic, but the problem size is tiny and hence the memory bandwidth stress is very low.

OpenACC or OpenMP on Larger Data?

We can easily scale this problem up, so why don't I? Because it is nice to have exercises that finish in a few minutes or less.

We scale this up to 10K x 10K (1.6 GB problem size) for the hybrid challenge. These numbers start to look a little more realistic. But the serial code takes over 30 minutes to finish. That would have gotten us off to a slow start!

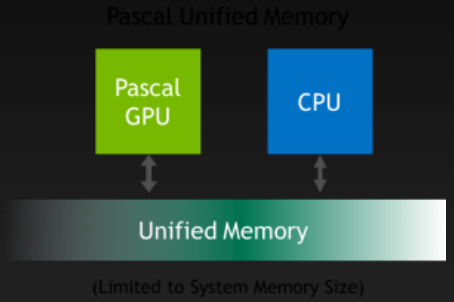
Execution	Time (s)	Speedup
CPU Serial	2187	--
CPU 16 OpenMP threads	183	12
CPU 28 OpenMP threads	162	13.5
OpenACC	103	21

Obvious cusp for core scaling appears

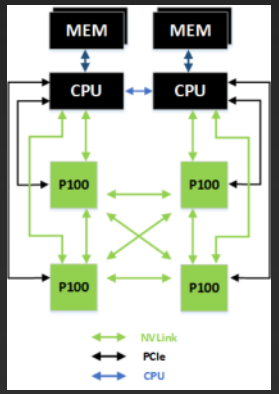
10K x 10K Problem Size

Latest Happenings In Data Management

- **Unified Memory**
  - Unified address space allows us to pretend we have shared memory
  - Skip data management, hope it works, and then optimize if necessary
  - For dynamically allocated memory can eliminate need for pointer clauses



- **NVLink**
  - One route around PCI bus (with multiple GPUs)



Further speedups

- OpenACC gives us even more detailed control over parallelization
  - Via gang, worker, and vector clauses
- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- By understanding bottlenecks in the code via profiling, we can reorganize the code for higher performance
- But you have already gained most of any potential speedup, and you did it with a few lines of directives!

Is OpenACC Living Up To My Claims?

- High-level. No involvement of OpenCL, CUDA, etc.
- Single source. No forking off a separate GPU code. Compile the same program for accelerators or serial; non-GPU programmers can play along.
- Efficient. Experience shows very favorable comparison to low-level implementations of same algorithms. **kernels** is magical!
- Performance portable. Supports GPU accelerators and co-processors from multiple vendors, current and future versions.
- Incremental. Developers can port and tune parts of their application as resources and profiling dictates. No wholesale rewrite required. Which can be *quick*.

In Conclusion…