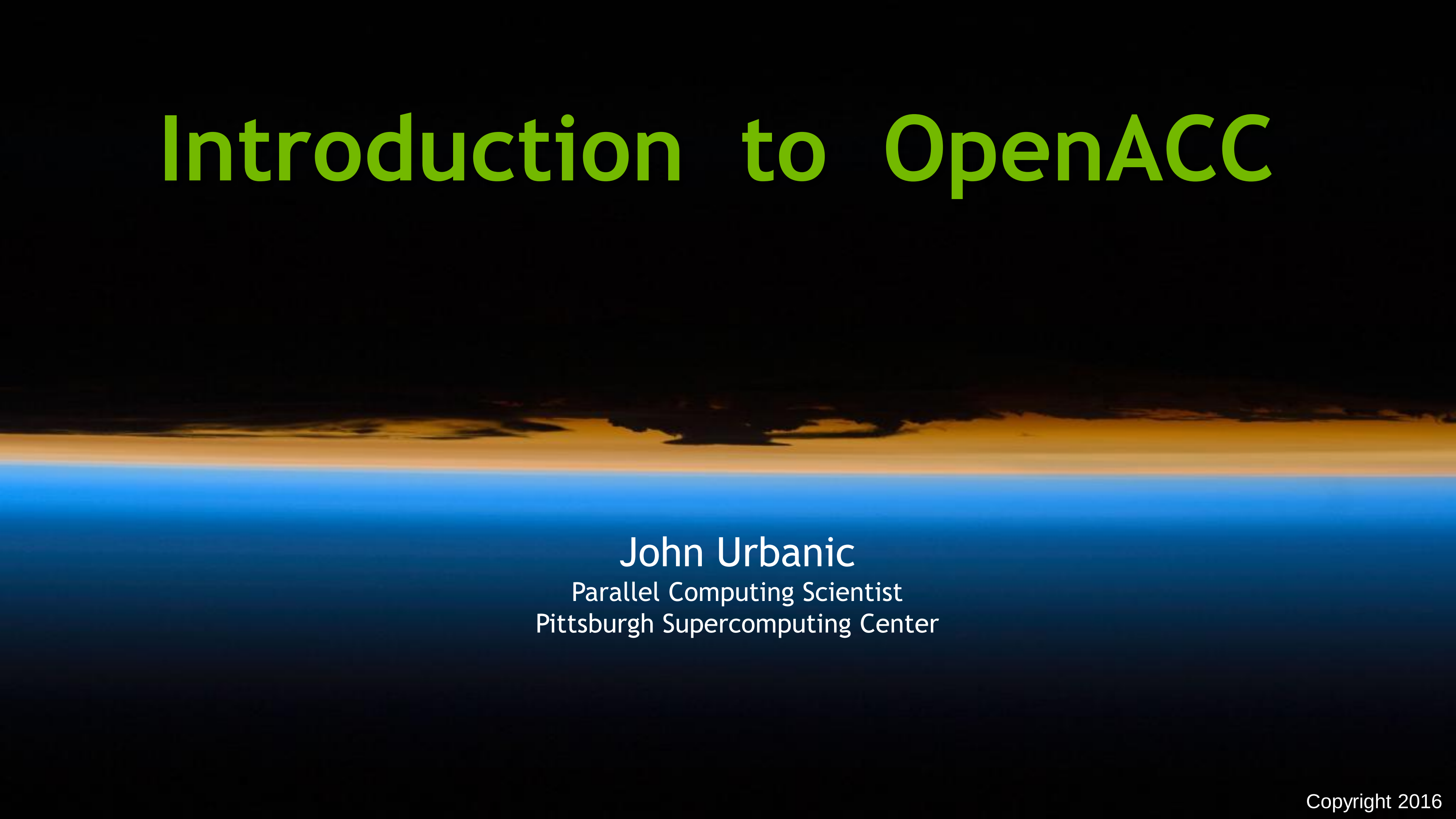


Introduction to OpenACC

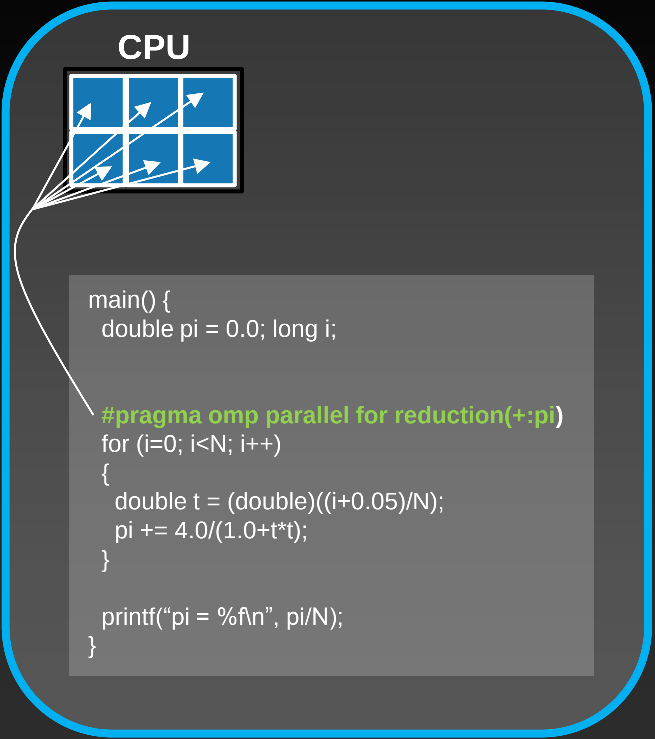
John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

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.

Look Familiar?

OpenMP

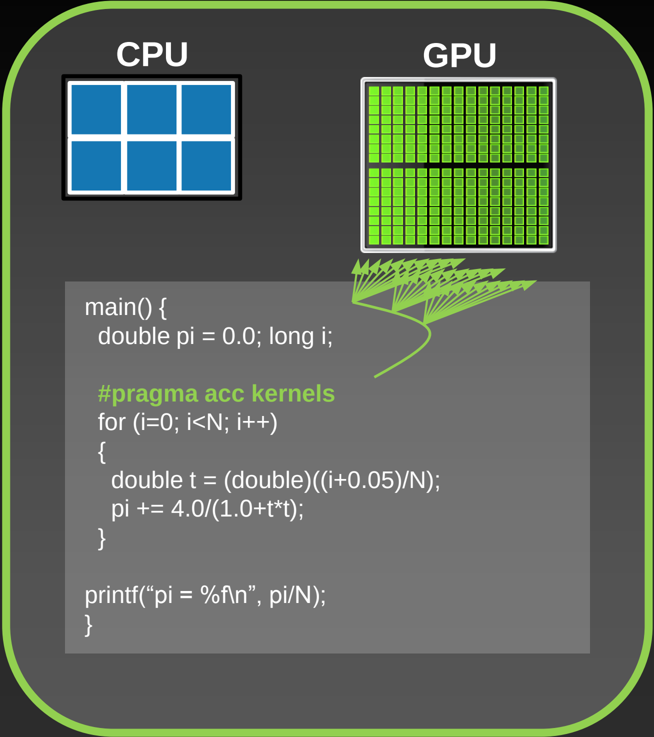


```
main() {
  double pi = 0.0; long i;

  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf("pi = %f\n", pi/N);
}
```

OpenACC



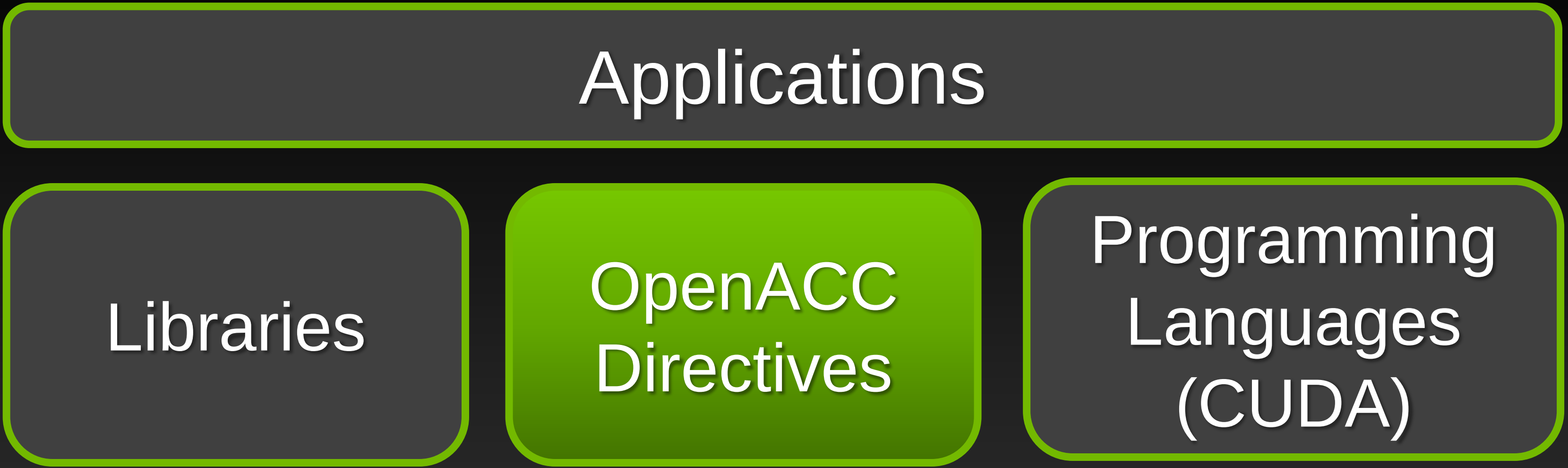
```
main() {
  double pi = 0.0; long i;

  #pragma acc kernels
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

printf("pi = %f\n", pi/N);
}
```

A few important differences!

How Else Would We Accelerate Applications?



“Drop-in” Acceleration

Incrementally Accelerate Applications

Maximum Flexibility

Similar 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.*

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

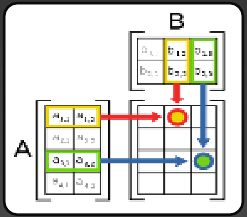
Opel Blue, Ltd

Few hours

70x faster

Matrix Matrix Multiply

Independent Research Scientist



4 directives

6.4x faster

A Champion Case

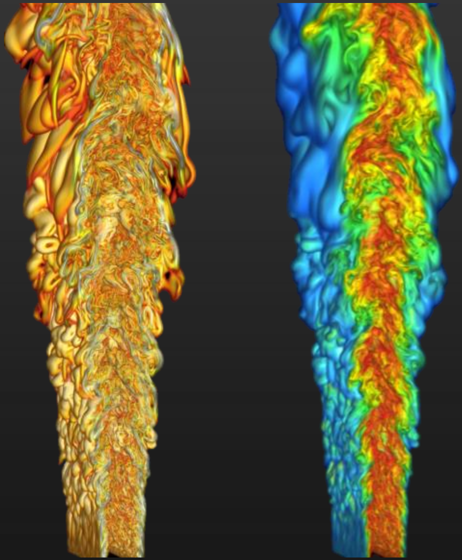
4x Faster

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

Broad Accelerator Support

- Xeon Phi support already in CAPS. Demonstrated and soon to be release for PGI.
- AMD line of accelerated processing units (APUs) as well as the AMD line of discrete GPUs for preliminary PGI support.
- Carma - a hybrid platform based on ARM Cortex-A9 quad core and an NVIDIA Quadro® 1000M GPU.
- NVIDIA...

NVIDIA Rules

or writes the rules. They have been the foremost supporter of GPU computing for much of the past decade, and have earned the focus of this workshop. We are using NVIDIA GPUs as our platform and our touchstone because:

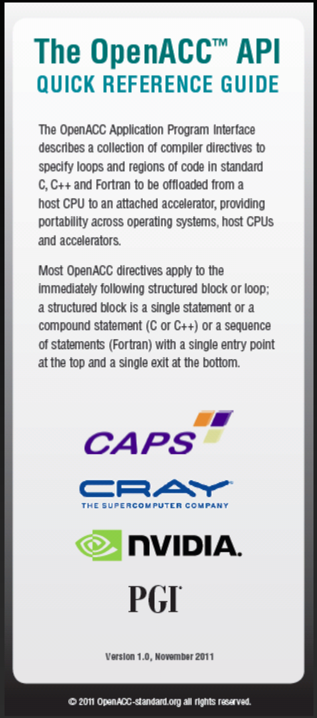
- They are proven
- Well understood
- Best bang for buck if you want to buy an accelerator
- Excellent support by vendor and community
- It is the basis for our leading edge platform, Keeneland
- It will not be going obsolete any time soon
- NVIDIA recently acquired PGI. That gave us a slight preference for the PGI compiler over the Cray one. Both are available on Blue Waters.

True Standard

- Full OpenACC 1.0 and 2.0 and now 2.5 Specifications available online

  http://www.openacc-standard.org

- Quick reference card also available
- Implementations available now from PGI, Cray, PathScale and CAPS.
- GCC version of OpenACC now in 5.x, better in 6.1.
- Several other open source versions as well. See openacc.org.


The OpenACC™ API
QUICK REFERENCE GUIDE

The OpenACC Application Program Interface describes a collection of compiler directives to specify loops and regions of code in standard C, C++ and Fortran to be offloaded from a host CPU to an attached accelerator, providing portability across operating systems, host CPUs and accelerators.

Most OpenACC directives apply to the immediately following structured block or loop; a structured block is a single statement or a compound statement (C or C++) or a sequence of statements (Fortran) with a single entry point at the top and a single exit at the bottom.

CAPS
CRAY THE SUPERCOMPUTER COMPANY
NVIDIA.
PGI

Version 1.0, November 2011

© 2011 OpenACC-standard.org all rights reserved.

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)
...
```

PITTSBURGH SUPERCOMPUTING CENTER

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.

PITTSBURGH SUPERCOMPUTING CENTER

Complete SAXPY Example Code

```c
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;
}
```

```c
#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)"

PITTSBURGH SUPERCOMPUTING CENTER

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: `cc -acc -Minfo=accel saxpy.c`
- Fortran: `ftn -acc -Minfo=accel saxpy.f90`

Compiler Output

```
cc -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
         Accelerator kernel generated
          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: `aprun a.out`

PITTSBURGH SUPERCOMPUTING CENTER

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 );
}
```

PITTSBURGH SUPERCOMPUTING CENTER

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

```c
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

```c
#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];
}
```

PITTSBURGH SUPERCOMPUTING CENTER

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 variables 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?

déjà vu: This looks 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

New and exciting!

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

Data Dependencies

Very much unlike OpenMP, if the compiler even ***suspects*** that there is a data dependency, it will, for the sake of correctness, refuse to parallelize that loop.

```
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

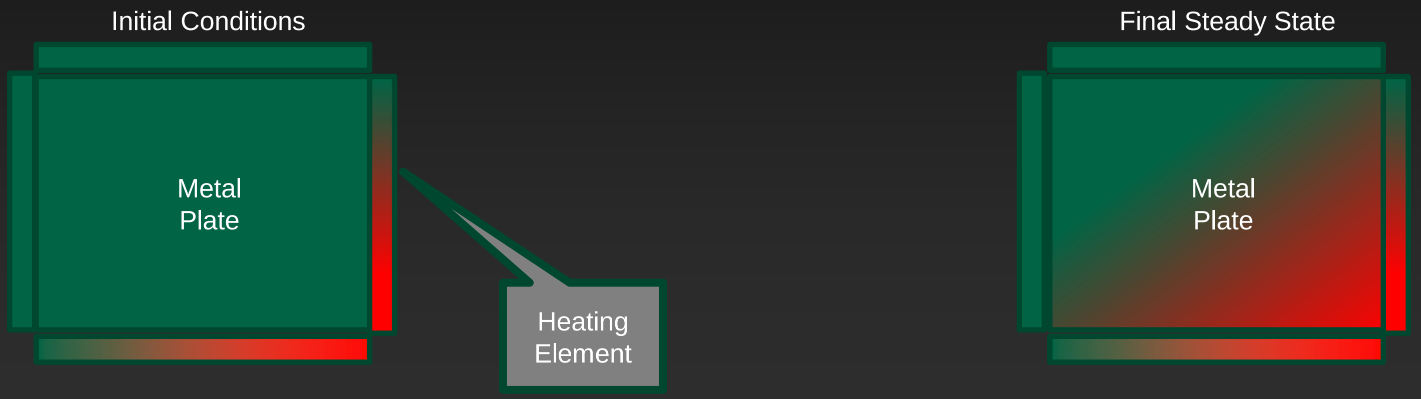
- Rearrange your code to make it more obvious to the compiler that there is not really a data dependency.

- 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 Returns

- 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:



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

PITTSBURGH SUPERCOMPUTING CENTER

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

Exercises: General Instructions for Compiling

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

Activate OpenACC directives

```
pgcc -acc laplace.c
pgf90 -acc laplace.f90
```

- This will generate the executable a.out

PITTSBURGH SUPERCOMPUTING CENTER

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.

```
instr009@h2ologin2:~/Test> cc -acc -Minfo=accel laplace_bad_acc.c
main:
     71, Generating present_or_copyout(Temperature[1:1000][1:1000])
         Generating present_or_copyin(Temperature_old[0:][0:])
         Generating NVIDIA code
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     72, Loop is parallelizable
     73, Loop is parallelizable
         Accelerator kernel generated
         72, #pragma acc loop gang /* blockIdx.y */
         73, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     82, Generating present_or_copyin(Temperature[1:1000][1:1000])
         Generating present_or_copy(Temperature_old[1:1000][1:1000])
         Generating NVIDIA code
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     83, Loop is parallelizable
     84, Loop is parallelizable
         Accelerator kernel generated
         83, #pragma acc loop gang /* blockIdx.y */
         84, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     85, Max reduction generated for dt
```

Special Instructions for Running on the GPUs

As I mentioned, we on Bridges you generally only have to use the queueing system when you want to. However, as we have 300+ of you, and only 60 GPUs at this phase of building the machine, we will have to use it here to prevent anarchy.

You do not have to be in an interactive shell (indeed should not) to use the GPU queues. We can all edit and compile from the Bridges login nodes. Remember, they have a command prompt like “fred@br003$”.

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

```
fred@br003$ 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.

PITTSBURGH SUPERCOMPUTING CENTER

Exercise 1: Using `kernels` to parallelize the main loops
(About 45 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. `pgcc -acc -Minfo=accel laplace_acc.c` or
      `pgf90 -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`
   2. Wait a minute and then look at the slurm output file for your results.
   3. Compare to the serial version to see how you are doing.

PITTSBURGH SUPERCOMPUTING CENTER

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

PITTSBURGH SUPERCOMPUTING CENTER

Exercise 1: Compiler output (C)

```
instr009@h2ologin2:~/Update> cc -acc -Minfo=accel laplace_bad_acc.c
main:
     62, Generating present_or_copyout(Temperature[1:1000][1:1000])
         Generating present_or_copyin(Temperature_last[0:][0:])
         Generating NVIDIA code
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     63, Loop is parallelizable
     64, Loop is parallelizable
         Accelerator kernel generated
         63, #pragma acc loop gang /* blockIdx.y */
         64, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     73, Generating present_or_copyin(Temperature[1:1000][1:1000])
         Generating present_or_copy(Temperature_last[1:1000][1:1000])
         Generating NVIDIA code
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
     74, Loop is parallelizable
     75, Loop is parallelizable
         Accelerator kernel generated
         74, #pragma acc loop gang /* blockIdx.y */
         75, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     76, Max reduction generated for dt
```

Compiler was able to parallelize

Compiler was able to parallelize

PITTSBURGH SUPERCOMPUTING CENTER

Exercise 1: Performance

3372 steps to convergence

Execution	Time (s)	Speedup
CPU Serial	18	--
CPU 2 OpenMP threads	9.4	1.99
CPU 4 OpenMP threads	4.7	3.98
CPU 8 OpenMP threads	2.5	7.48
CPU 16 OpenMP threads	1.4	13.4
CPU 28 OpenMP threads	0.9	21.5
OpenACC GPU	29	0.6x

PITTSBURGH SUPERCOMPUTING CENTER

What went wrong?

export PGI_ACC_TIME=1 to activate profiling and run again:

```
Accelerator Kernel Timing data
/mnt/a/u/training/instr009/Update/laplace_bad_acc.c
  main  NVIDIA  devicenum=0
    time(us): 22,902,870
    62: compute region reached 3372 times
        62: data copyin reached 3372 times
             device time(us): total=4,561,531 max=1,362 min=1,350 avg=1,352
        64: kernel launched 3372 times
            grid: [8x1000]  block: [128]
             device time(us): total=441,105 max=268 min=129 avg=130
            elapsed time(us): total=487,585 max=282 min=141 avg=144
        70: data copyout reached 3372 times
             device time(us): total=4,063,246 max=1,230 min=1,202 avg=1,204
    73: compute region reached 3372 times
        73: data copyin reached 6744 times
             device time(us): total=9,135,367 max=1,428 min=1,346 avg=1,354
        75: kernel launched 3372 times
            grid: [8x1000]  block: [128]
             device time(us): total=546,820 max=296 min=155 avg=162
            elapsed time(us): total=593,424 max=309 min=171 avg=175
        75: reduction kernel launched 3372 times
            grid: [1]  block: [256]
             device time(us): total=91,638 max=161 min=25 avg=27
            elapsed time(us): total=136,871 max=174 min=38 avg=40
        82: data copyout reached 3372 times
             device time(us): total=4,063,163 max=1,259 min=1,202 avg=1,204
```

4.5 seconds

0.5 seconds

4.0 seconds

9.1 seconds

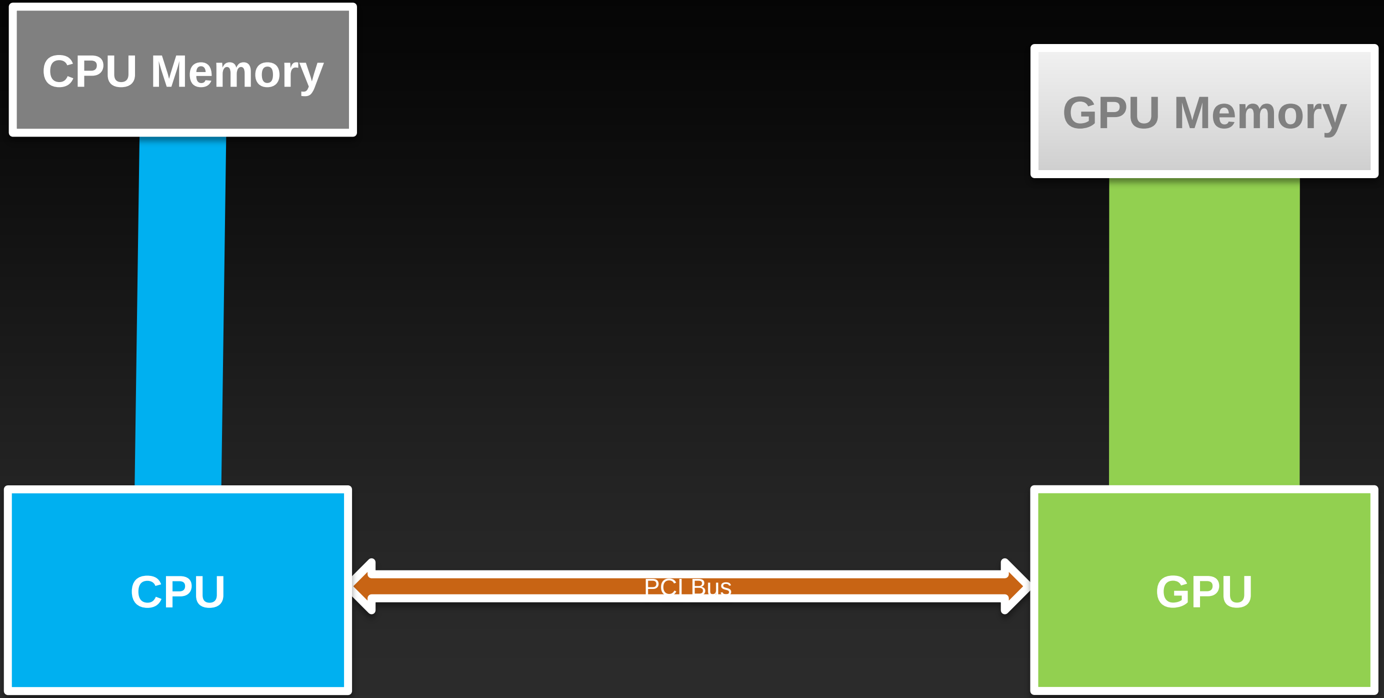
0.6 seconds

0.1 seconds

4.0 seconds

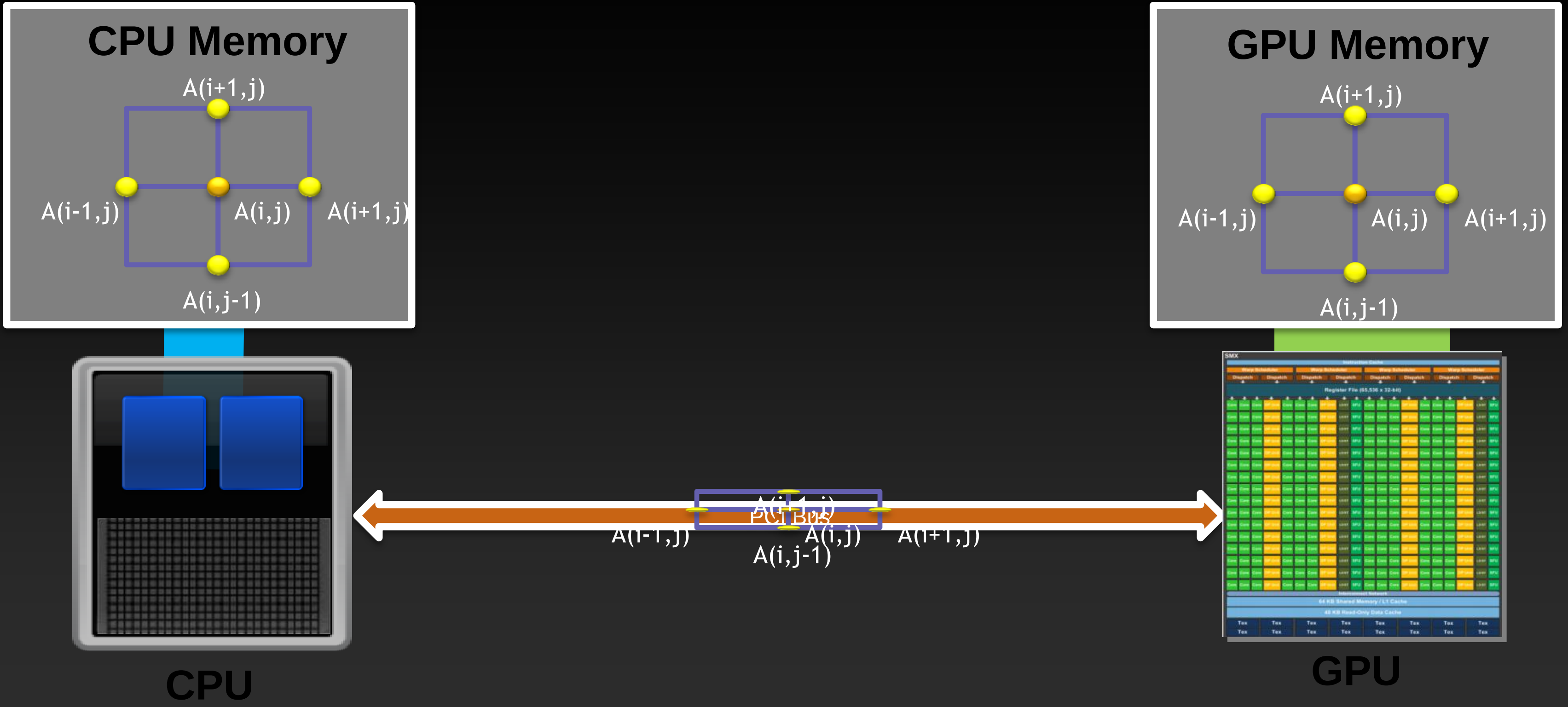
Basic Concept

Simplified, but sadly true

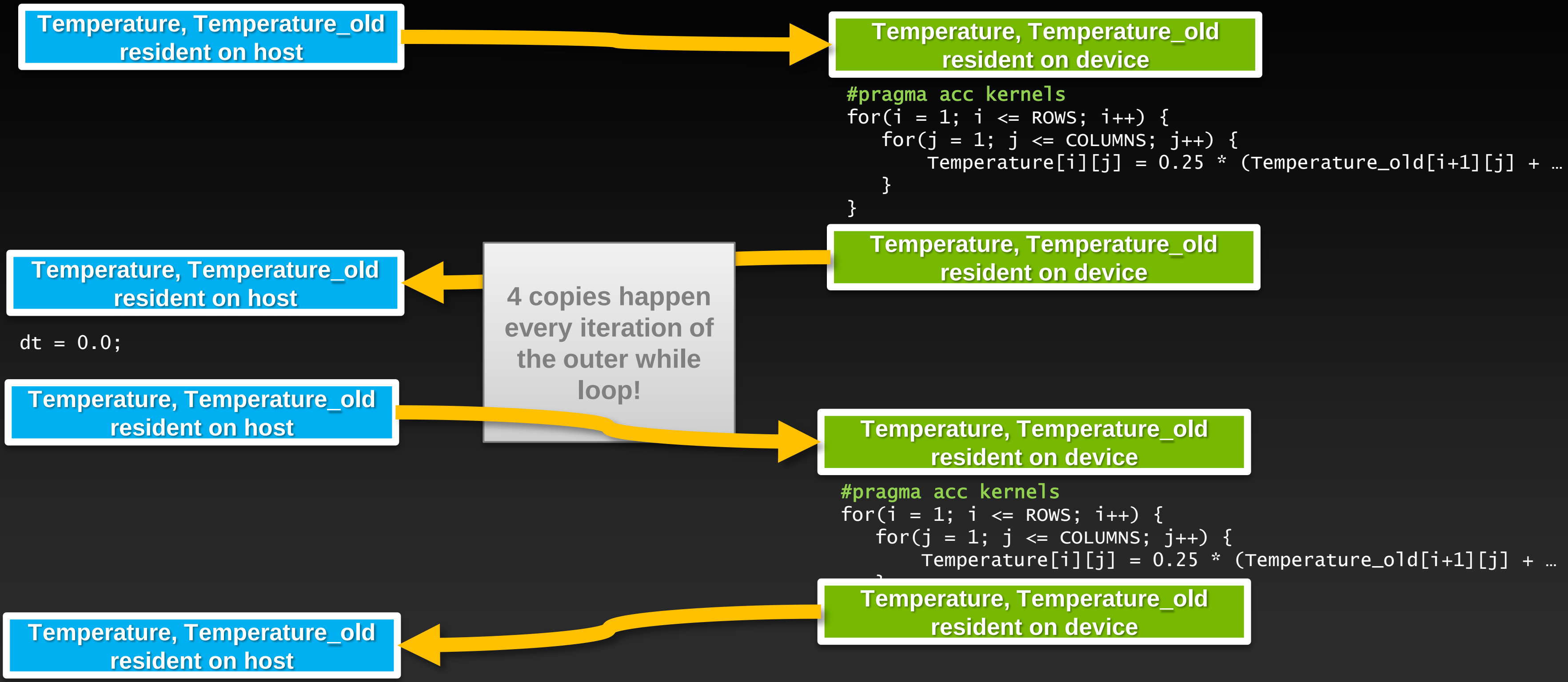


PITTSBURGH SUPERCOMPUTING CENTER

Multiple Times Each Iteration



Excessive Data Transfers



Temperature, Temperature_old resident on host
Temperature, Temperature_old resident on device
Temperature, Temperature_old resident on device
Temperature, Temperature_old resident on host
4 copies happen every iteration of the outer while loop!
Temperature, Temperature_old resident on host
Temperature, Temperature_old resident on device
Temperature, Temperature_old resident on device
Temperature, Temperature_old resident on host

```
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_old[i+1][j] + …
       }
    }

    dt = 0.0;

    #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] + …

}
```

Data Management

The First, Most Important, and possibly Only OpenACC Optimization

PITTSBURGH SUPERCOMPUTING CENTER

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 kernels 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 will be combined with (intelligently) initialized parallel copies at end.

This explicitly declares the reduction.

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

That the compiler recognizes this and does a reduction is a wonderful thing. Indeed, we can get too sophisticated for it to happen automatically.

Data Construct Syntax and Scope

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.

PITTSBURGH SUPERCOMPUTING CENTER

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];

  }
}
```

```
pgcc -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
```

Smarter

Smartest

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

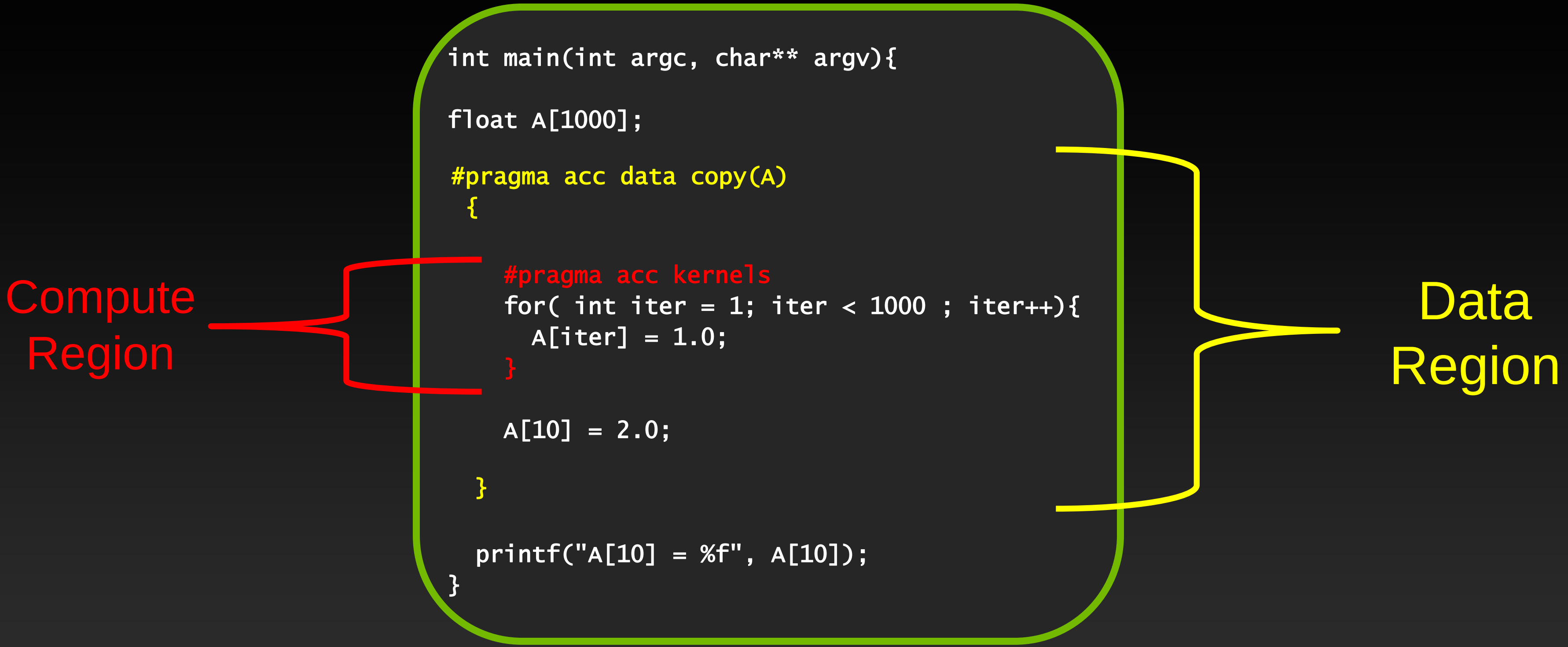
Still Runs On Host

A[] Copied To Host

```
Output:
            A[10] = 1.0
```

PITTSBURGH SUPERCOMPUTING CENTER

Data Regions Are Different Than Compute Regions



```
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]);
}
```

```
Output:
          A[10] = 1.0
```

PITTSBURGH SUPERCOMPUTING CENTER

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)  //Gets host a current copy`

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.

PITTSBURGH SUPERCOMPUTING CENTER

Exercise 2 C 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++;
}
```

No data movement in this block.

Except once in a while here.

PITTSBURGH SUPERCOMPUTING CENTER

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 GPI

Extra efficient:

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

Except bring back a copy here

PITTSBURGH SUPERCOMPUTING CENTER

Exercise 2: Performance

3372 steps to convergence

Execution	Time (s)	Speedup
CPU Serial	18	--
CPU 2 OpenMP threads	9.4	1.99
CPU 4 OpenMP threads	4.7	3.98
CPU 8 OpenMP threads	2.5	7.48
CPU 16 OpenMP threads	1.4	13.4
CPU 28 OpenMP threads	0.9	21.5
OpenACC GPU	1.5	12

PITTSBURGH SUPERCOMPUTING CENTER

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 very realistic, but 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 will indeed 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

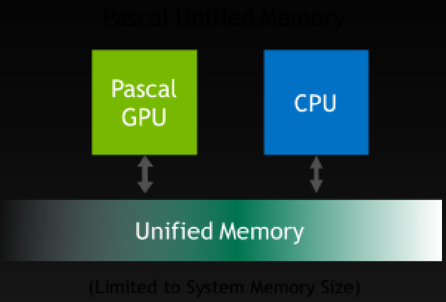
Obvious cusp for core scaling appears

10K x 10K Problem Size

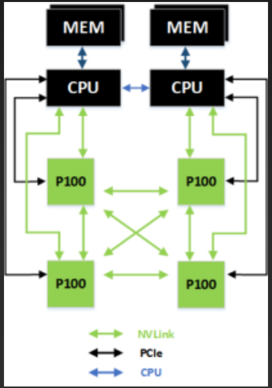
PITTSBURGH SUPERCOMPUTING CENTER

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!

General Principles: Finding Parallelism In Code

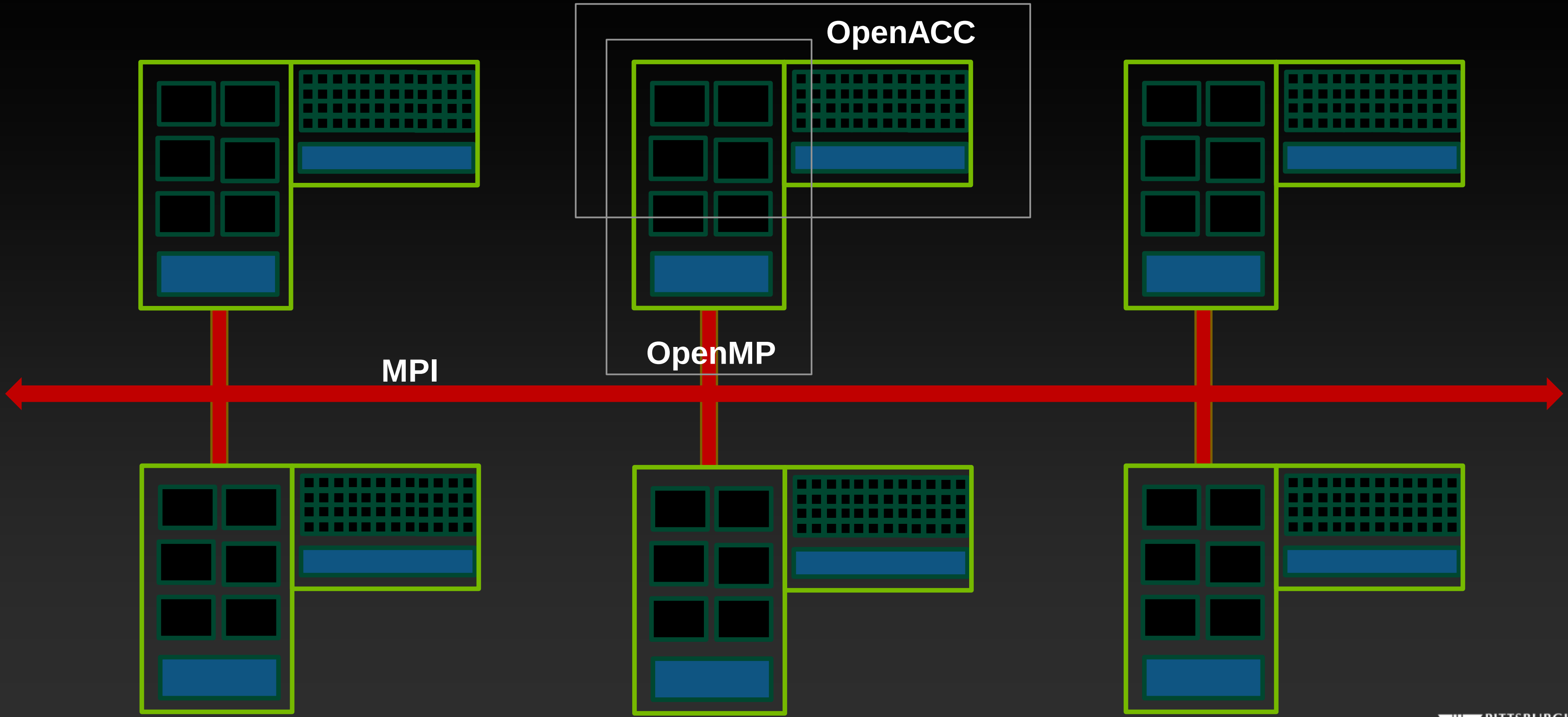
- Nested for/do loops are best for parallelization
  - Large loop counts are best
- Iterations of loops must be <u>independent</u> of each other
  - To help compiler: `restrict` keyword (C), `independent` clause
  - Use subscripted arrays, rather than pointer-indexed arrays (C)
- Data regions should avoid wasted transfers
  - If applicable, could use directives to explicitly control sizes
- Various other annoying things can interfere with accelerated regions
  - IO
  - Limitations on function calls and nested parallelism (relaxed much in 2.0)

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 show 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.*

PITTSBURGH SUPERCOMPUTING CENTER

In Conclusion…


OpenACC
OpenMP
MPI