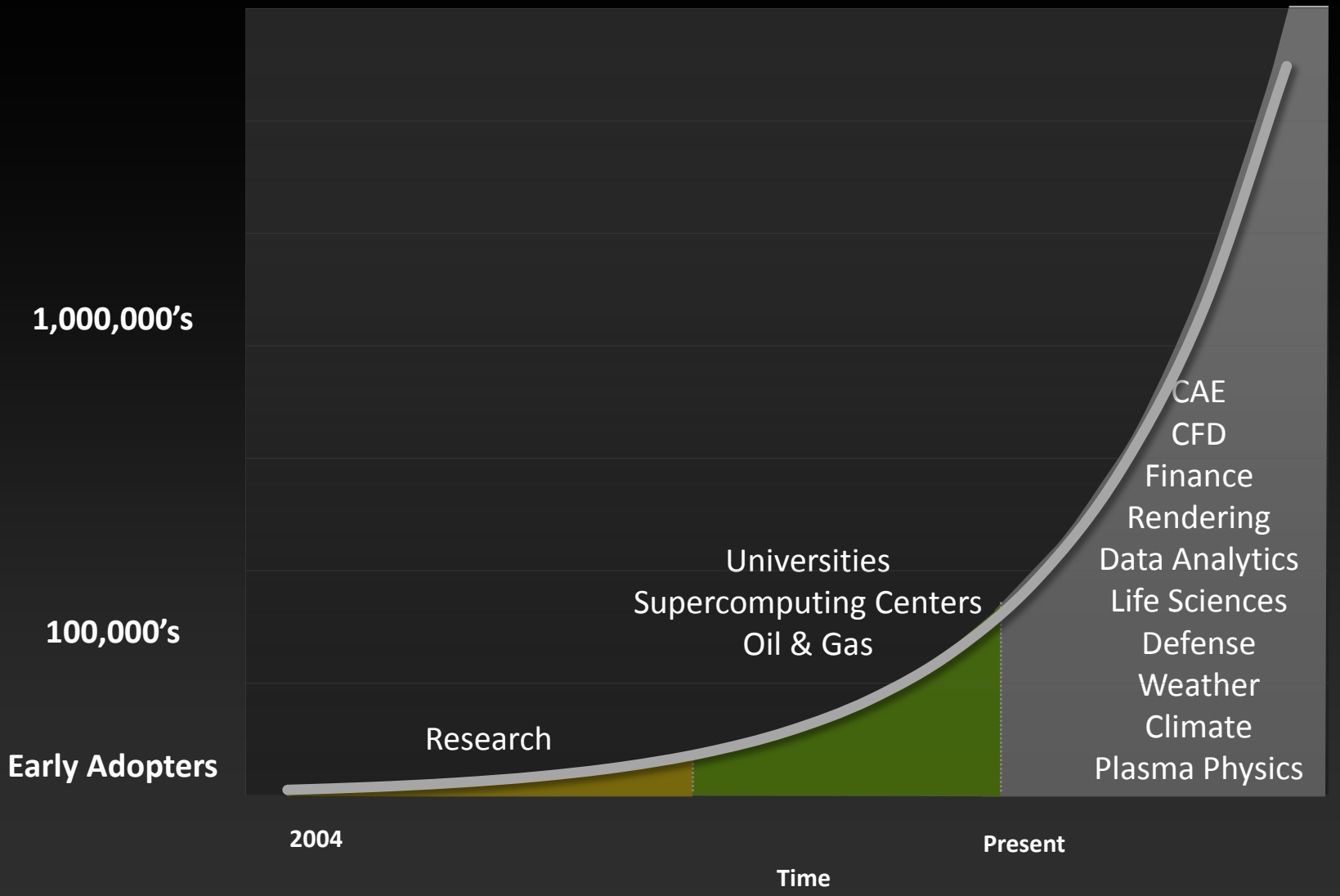
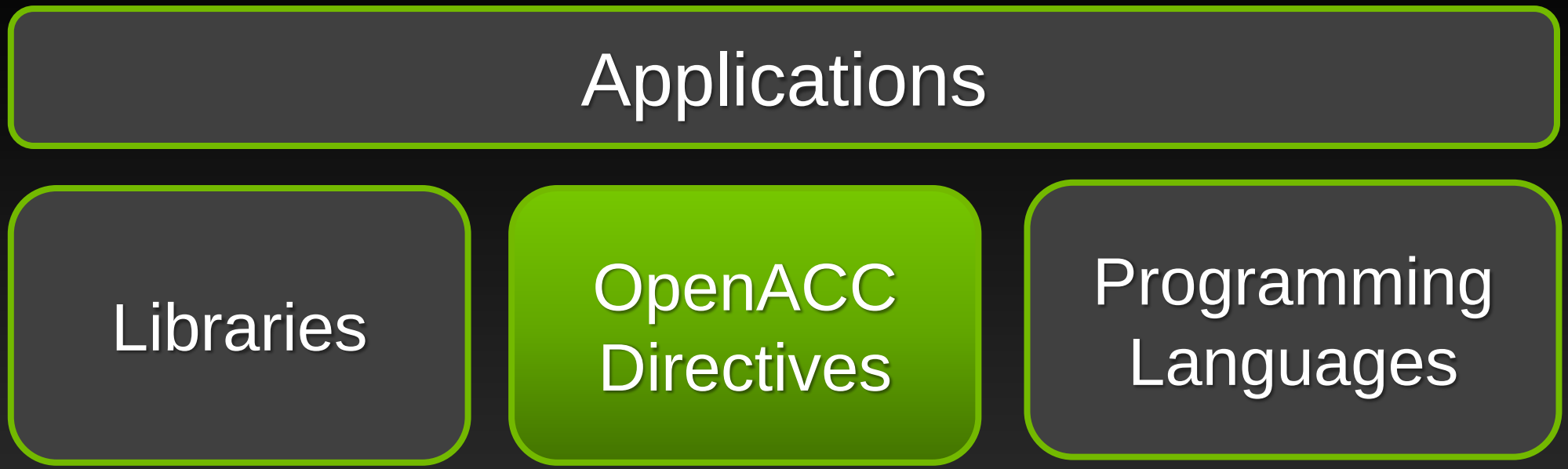

GPU Computing with
OpenACC Directives
NVIDIA

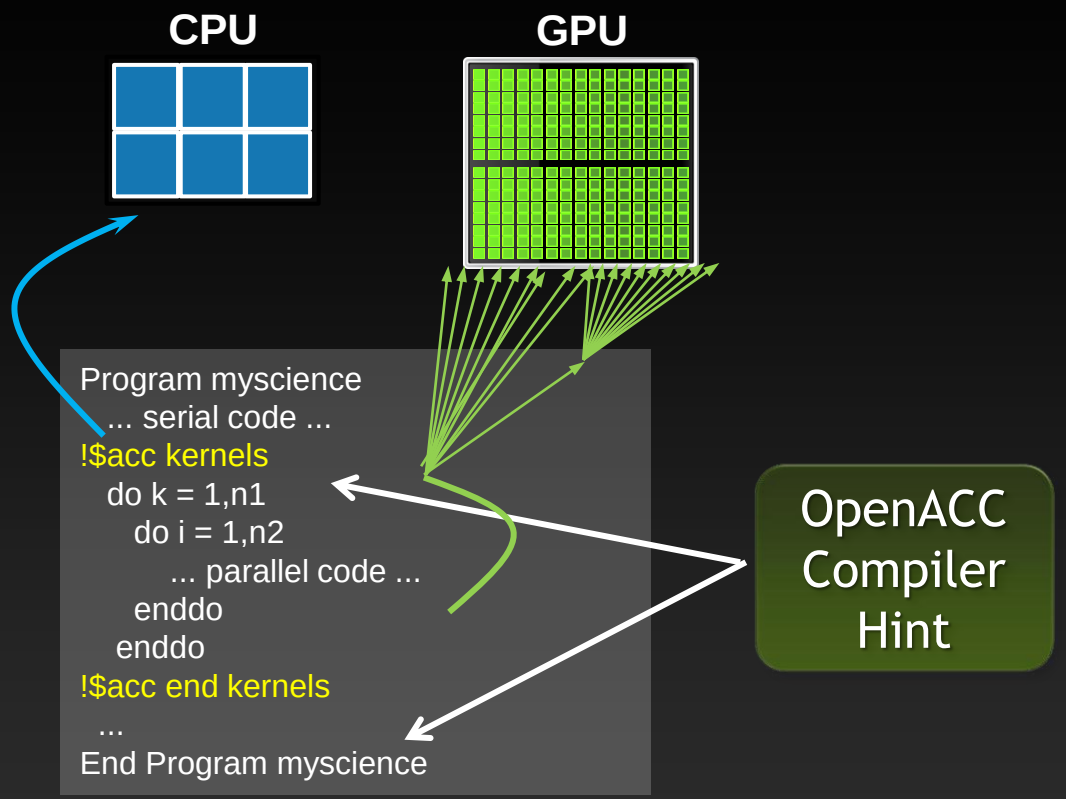
GPUs Reaching Broader Set of Developers



3 Ways to Accelerate Applications



OpenACC Directives



Your original Fortran or C code

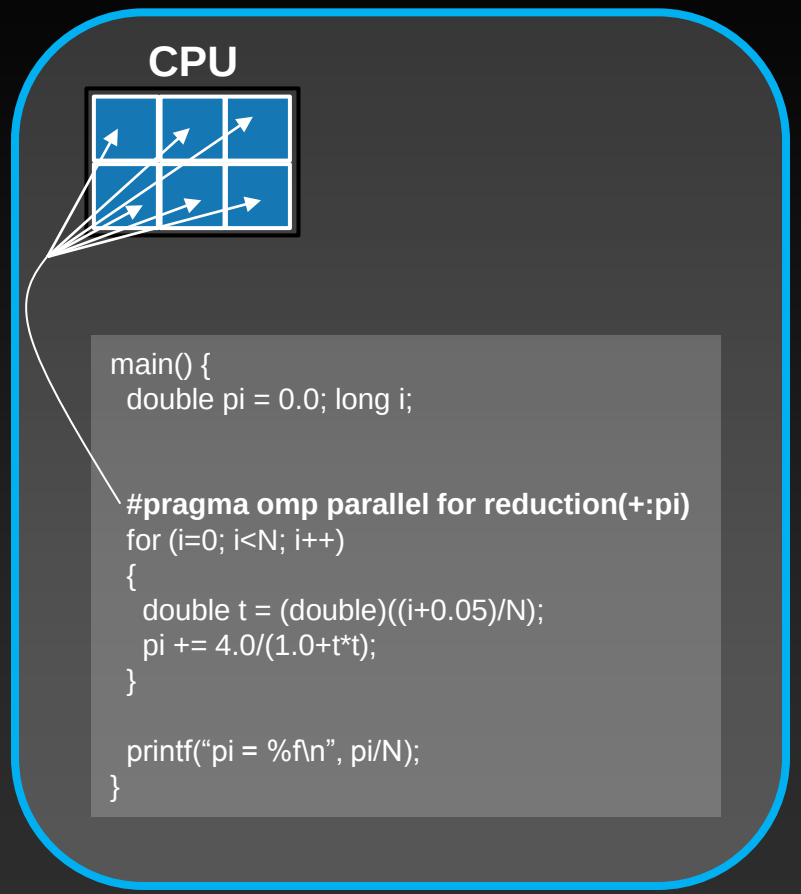
Simple Compiler hints

Compiler Parallelizes code

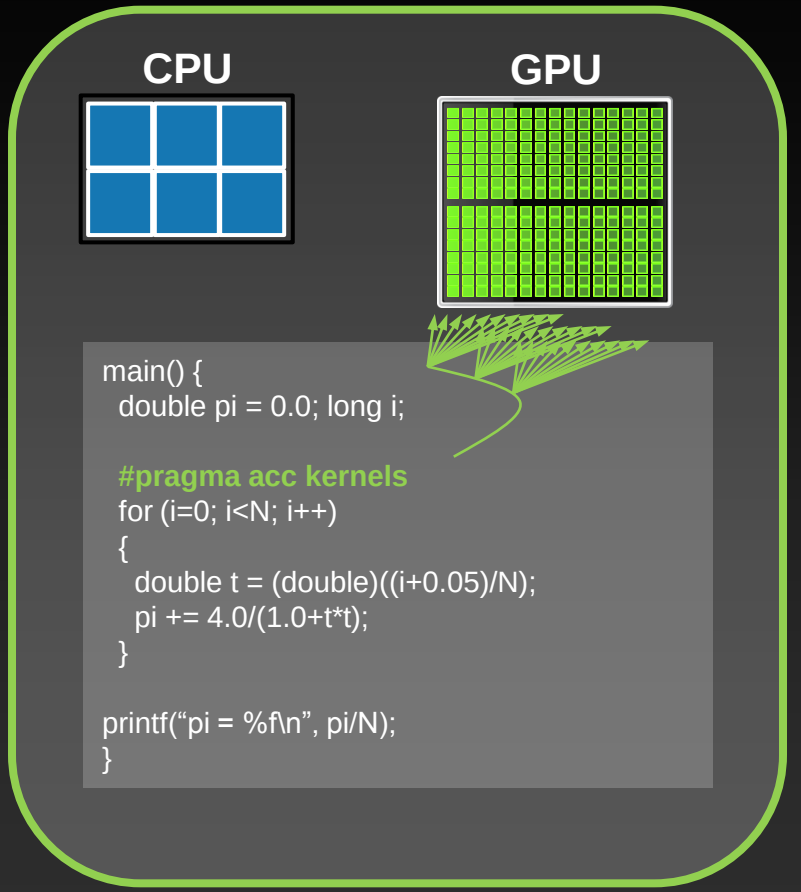
Works on many-core GPUs & multicore CPUs

Familiar to OpenMP Programmers

OpenMP



OpenACC



OpenACC

Open Programming Standard for Parallel Computing

"OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan."

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

"OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP."

--Michael Wong, CEO OpenMP Directives Board

OpenACC Standard

NVIDIA

CRAY THE SUPERCOMPUTER COMPANY

PGI

CAPS

OpenACC
The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

High-level, with low-level access

- Compiler directives to specify parallel regions in C, C++, Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems

5x in 40 Hours

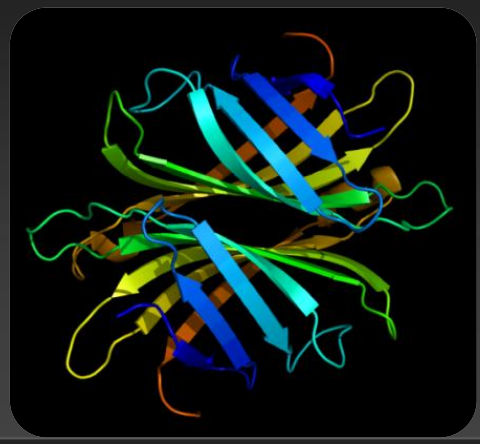
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 8 Hours

> "Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications."
>
> *-- Developer at the Global Manufacturer of Navigation Systems*

Small Effort. Real Impact.

Large Oil Company

3x in 7 days

Solving billions of equations iteratively for oil production at world's largest petroleum reservoirs

UNIVERSITY of HOUSTON
CULLEN COLLEGE of ENGINEERING

Univ. of Houston

Prof. M.A. Kayali

20x in 2 days

Studying magnetic systems for innovations in magnetic storage media and memory, field sensors, and biomagnetism

THE UNIVERSITY OF MELBOURNE

Uni. Of Melbourne

Prof. Kerry Black

65x in 2 days

Better understand complex reasons by lifecycles of snapper fish in Port Phillip Bay

УГАТУ

Ufa State Aviation

Prof. Arthur Yuldashev

7x in 4 Weeks

Generating stochastic geological models of oilfield reservoirs with borehole data

COMPUTING FOR SCIENCE fs

GAMESS-UK

Dr. Wilkinson, Prof. Naidoo

10x

Used for various fields such as investigating biofuel production and molecular sensors.

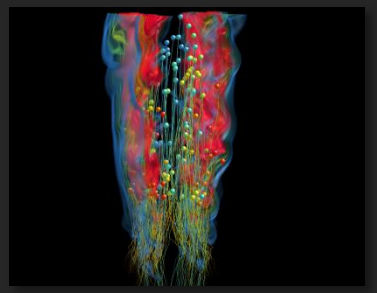
* Achieved using the PGI Accelerator Compiler

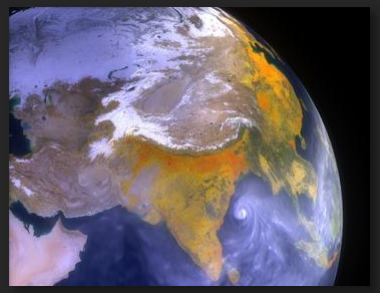
Focus on Exposing Parallelism

With Directives, tuning work focuses on *exposing parallelism*, which makes codes inherently better

Example: Application tuning work using directives for new Titan system at ORNL

S3D
Research more efficient combustion with next-generation fuels





CAM-SE
Answer questions about specific climate change adaptation and mitigation scenarios

- **Tuning top 3 kernels (90% of runtime)**
- ***3 to 6x faster on CPU+GPU vs. CPU+CPU***
- **But also improved all-CPU version by 50%**

- **Tuning top key kernel (50% of runtime)**
- **6.5x faster on CPU+GPU vs. CPU+CPU**
- **Improved performance of CPU version by 100%**

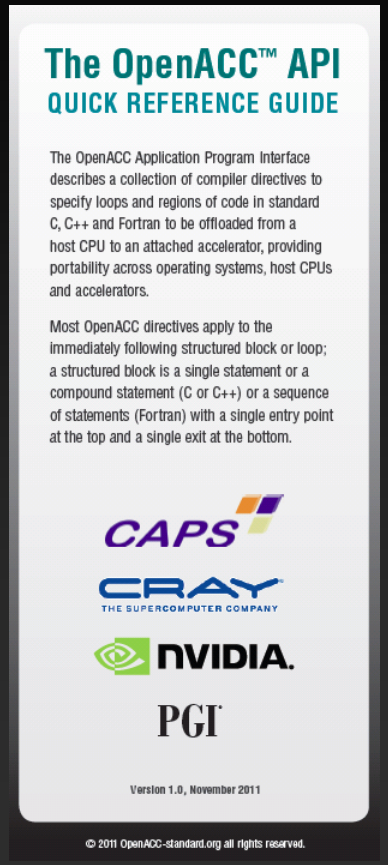
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online

  http://www.openacc-standard.org

- Quick reference card also available

- Beta implementations available now from PGI, Cray, and CAPS



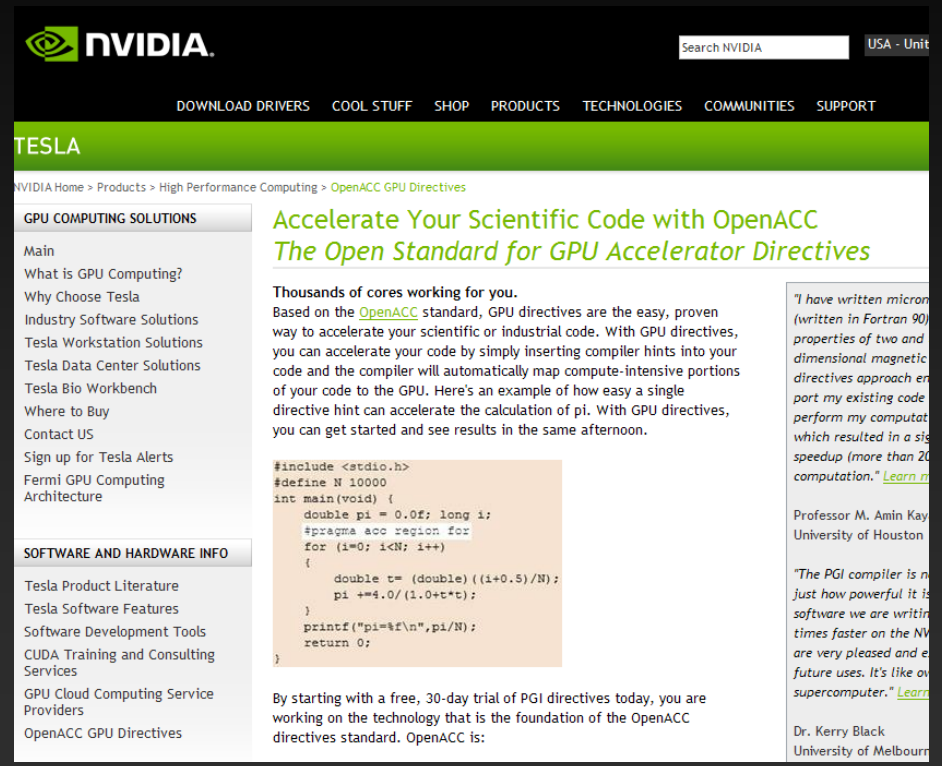
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator

Tools for quick ramp

www.nvidia.com/gpudirectives



NVIDIA

PGI®

A Very Simple Exercise: 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];
}

...
// Perform 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


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

Directive Syntax

- Fortran

  `!$acc directive [clause [,] clause] …]`

  Often paired with a matching end directive surrounding a structured code block

  `!$acc end directive`

- C

  `#pragma acc directive [clause [,] clause] …]`

  Often followed by a structured code block

kernels: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```
!$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 function that runs on the GPU

Kernels Construct

Fortran

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

C

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

Clauses

```
if( condition )
async( expression )
```

Also, any data clause (more later)

C tip: the `restrict` keyword

- Declaration of intent given by the programmer to the compiler
  - Applied to a pointer, e.g.

```
float *restrict ptr
```

  - Meaning: “for the lifetime of `ptr`, only it or a value directly derived from it (such as `ptr + 1`) will be used to access the object to which it points”*
- 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

http://en.wikipedia.org/wiki/Restrict

Complete SAXPY example code

- Trivial first example
  - Apply a loop directive
  - Learn compiler commands

```
#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];
}
```

*restrict:
"I promise y does not alias x"

```
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;
}
```

Compile and run

- C:

```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
```

- Fortran:

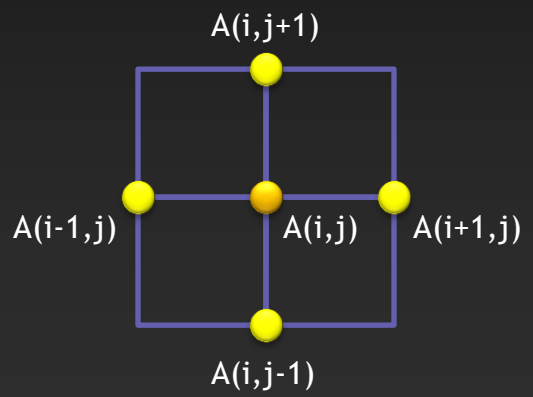
```
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90
```

- Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc 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
```

Example: Jacobi Iteration

- Iteratively converges to correct value (e.g. Temperature), by computing new values at each point from the average of neighboring points.
  - Common, useful algorithm
  - Example: Solve Laplace equation in 2D: $\nabla^2 f(x,y) = 0$



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

Jacobi Iteration C Code

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

  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {


      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);


      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

Jacobi Iteration Fortran Code


```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

  do j=1,m
    do i=1,n


      Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                 A(i  , j-1) + A(i  , j+1))


      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do

  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do

  iter = iter +1
end do
```

Iterate until converged

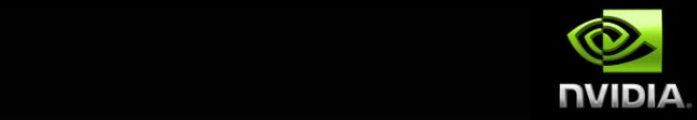
Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

OpenMP C Code



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

#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Parallelize loop across CPU threads

Parallelize loop across CPU threads

OpenMP Fortran Code

```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$omp parallel do shared(m,n,Anew,A) reduction(max:err)
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                 A(i  , j-1) + A(i  , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do

!$omp parallel do shared(m,n,Anew,A)
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do

  iter = iter +1
end do
```

Parallelize loop across CPU threads

Parallelize loop across CPU threads

Exercises: General Instructions (compiling)

- Exercises are in “exercises/openacc” directory in your home directory
  - Solutions are in “exercise_solutions/openacc” directory

- To compile, use one of the provided makefiles

  C:

  ```
  > make
  ```

  Fortran:

  ```
  > make -f Makefile_f90
  ```

- Remember these compiler flags:

  ```
  -acc -ta=nvidia -Minfo=accel
  ```

Exercises: General Instructions (running)

To run, use qsub with one of the provided job files

```
> qsub laplace_acc.job
> qstat          # prints qsub status
```

Output is placed in laplace_acc.job.o<job#> when finished.

OpenACC job file looks like this

```
#!/bin/csh
#PBS -l walltime=3:00
./laplace2d_acc
```

The OpenMP version specifies number of cores to use

```
#!/bin/csh
#PBS -l walltime=3:00
setenv OMP_NUM_THREADS 6
./laplace2d_omp
```

Edit this to control the number of cores to use

GPU startup overhead

- If no other GPU process running, GPU driver may be swapped out
  - Linux specific
  - Starting it up can take 1-2 seconds
- Two options
  - Run nvidia-smi in persistence mode (requires root permissions)
  - Run “nvidia-smi -q -l 30” in the background
- If your running time is off by ~2 seconds from results in these slides, suspect this
  - Nvidia-smi should be running in persistent mode for these exercises

Exercise 1: Jacobi Kernels

- Task: use `acc kernels` to parallelize the Jacobi loop nests

- Edit laplace2D.c or laplace2D.f90 (your choice)
  - In the `001-laplace2D-kernels` directory
  - Add directives where it helps
  - Figure out the proper compilation command (similar to SAXPY example)
    - Compile both with and without OpenACC parallelization
    - Optionally compile with OpenMP (original code has OpenMP directives)
  - Run OpenACC version with `laplace_acc`, OpenMP with `laplace_omp`

- Q: can you get a speedup with just kernels directives?
  - Versus 1 CPU core? Versus 6 CPU cores?

Exercise 1 Solution: OpenACC C

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

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest

Exercise 1 Solution: OpenACC Fortran


```
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                 A(i  , j-1) + A(i  , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

!$acc kernels
  do j=1,m-2
    do i=1,n-2
      A(i,j) = Anew(i,j)
    end do
  end do
!$acc end kernels
  iter = iter +1
end do
```

Generate GPU kernel for loop nest

Generate GPU kernel for loop nest

Exercise 1 Solution: C Makefile


```
CC       = pgcc
CCFLAGS  =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN =  laplace2d_omp laplace2d_acc

all: $(BIN)

laplace2d_acc: laplace2d.c
        $(CC) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_omp: laplace2d.c
        $(CC) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
        $(RM) $(BIN)
```

Exercise 1 Solution: Fortran Makefile


```
F90        = pgf90
CCFLAGS    =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN =  laplace2d_f90_omp laplace2d_f90_acc

all: $(BIN)

laplace2d_f90_acc: laplace2d.f90
        $(F90) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_f90_omp: laplace2d.f90
        $(F90) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
        $(RM) $(BIN)
```

Exercise 1: Compiler output (C)


```
pgcc  -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:
     57, Generating copyin(A[:4095][:4095])
         Generating copyout(Anew[1:4094][1:4094])
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     58, Loop is parallelizable
     60, Loop is parallelizable
         Accelerator kernel generated
         58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
         60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
             Cached references to size [18x18] block of 'A'
             CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
             CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
         64, Max reduction generated for error
     69, Generating copyout(A[1:4094][1:4094])
         Generating copyin(Anew[1:4094][1:4094])
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     70, Loop is parallelizable
     72, Loop is parallelizable
         Accelerator kernel generated
         70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
         72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
             CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
             CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy
```

Exercise 1: Performance

CPU: Intel Xeon X5690
6 Cores @ 3.47GHz

GPU: NVIDIA Tesla M2090

Execution (4096x4096)	Time (s)	Speedup
CPU 1 OpenMP thread	105.6	--
CPU 2 OpenMP threads	62.5	1.69x
CPU 4 OpenMP threads	43.3	2.44x
CPU 6 OpenMP threads	41.2	2.56x
OpenACC GPU	180.0	0.23x FAIL

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

What went wrong?

- Add `-ta=nvidia,time` to compiler command line

```
Accelerator Kernel Timing data
/home/jonathan.bentz/jlb_exercises/openacc/001-laplace2D-kernels/laplace2d.c
  main
    68: region entered 1000 times
        time(us): total=87778393 init=354 region=87778039
                  kernels=3010082 data=83805262
        w/o init: total=87778039 max=121635 min=87235 avg=87778
        71: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=3010082 max=3024 min=3002 avg=3010
/home/jonathan.bentz/jlb_exercises/openacc/001-laplace2D-kernels/laplace2d.c
  main
    56: region entered 1000 times
        time(us): total=92260070 init=223006 region=92037064
                  kernels=6265371 data=81807020
        w/o init: total=92037064 max=134358 min=91221 avg=92037
        59: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=6144256 max=6209 min=5777 avg=6144
        63: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=121115 max=129 min=120 avg=121
```

3.0 seconds

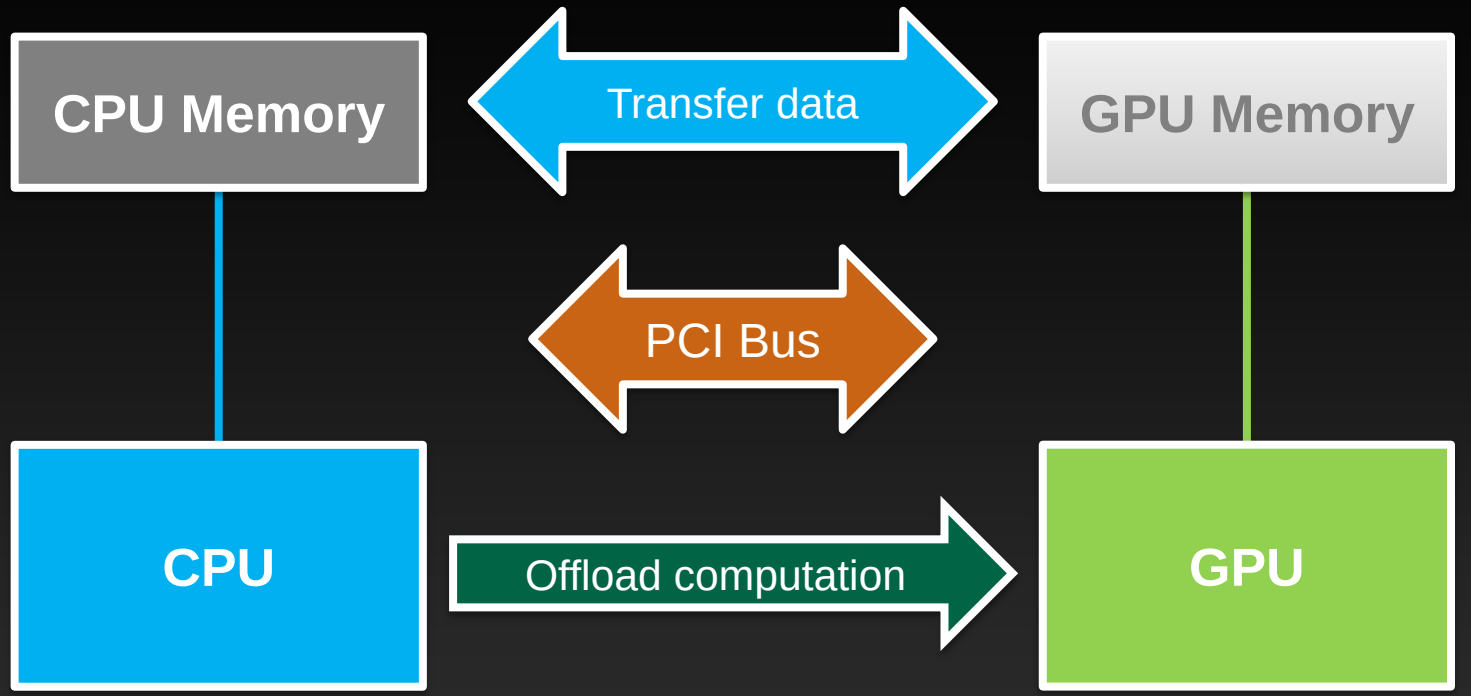
83.8 seconds

6.2 seconds

84.8 seconds

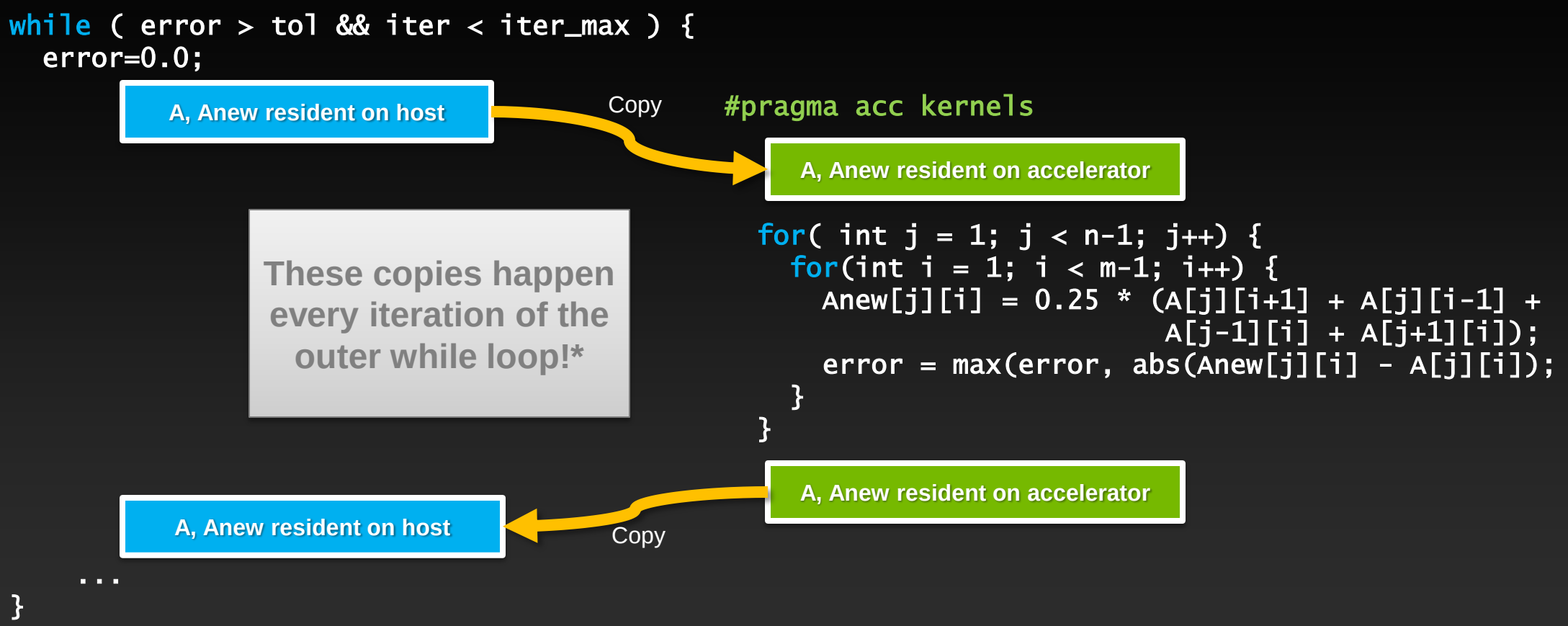
Huge Data Transfer Bottleneck!
Computation: 9.2 seconds
Data movement: 168.6 seconds

Basic Concepts



For efficiency, decouple data movement and compute off-load

Excessive Data Transfers



*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!

Data Management

Data Construct

Fortran

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

C

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

General Clauses

```
if( condition )
async( expression )
```

Manage data movement. Data regions may be nested.

Data Clauses

`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to the host when exiting region.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

Array Shaping

- Compiler sometimes cannot determine size of arrays
  - Must specify explicitly using data clauses and array “shape”
- C

```
#pragma acc data copyin(a[0:size-1]), copyout(b[s/4:3*s/4])
```

- Fortran

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

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

Update Construct

Fortran

```
!$acc update [clause …]
```

C

```
#pragma acc update [clause …]
```

Clauses

```
host( list )
device( list )
```

```
if( expression )
async( expression )
```

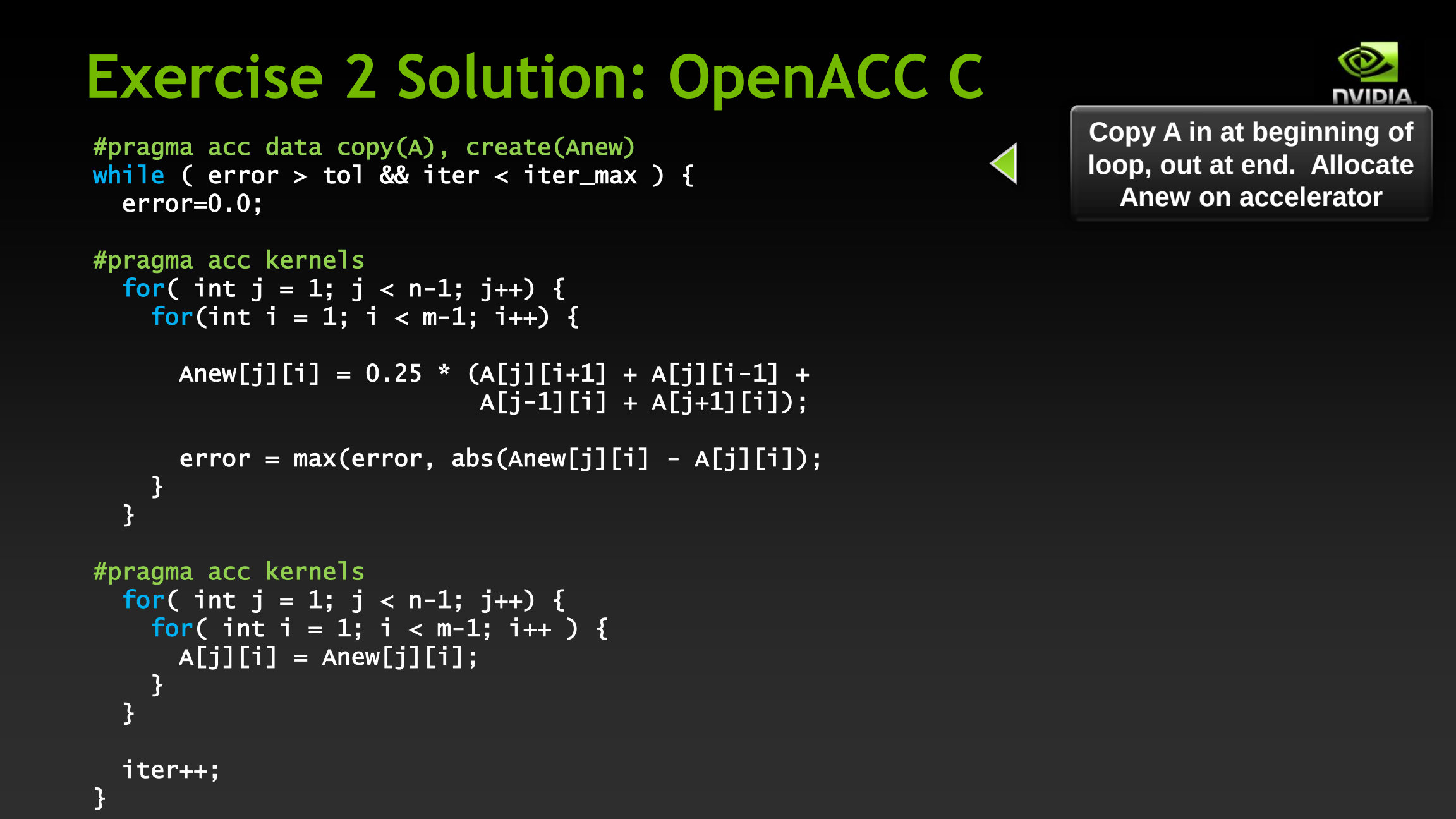
Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

Move data from GPU to host, or host to GPU.
Data movement can be conditional, and asynchronous.

Exercise 2: Jacobi Data Directives

- Task: use `acc data` to minimize transfers in the Jacobi example
- Start from given laplace2D.c or laplace2D.f90 (your choice)
  - In the `002-laplace2d-data` directory
  - Add directives where it helps (hint: [do] while loop)
- Q: What speedup can you get with data + kernels directives?
  - Versus 1 CPU core? Versus 6 CPU cores?

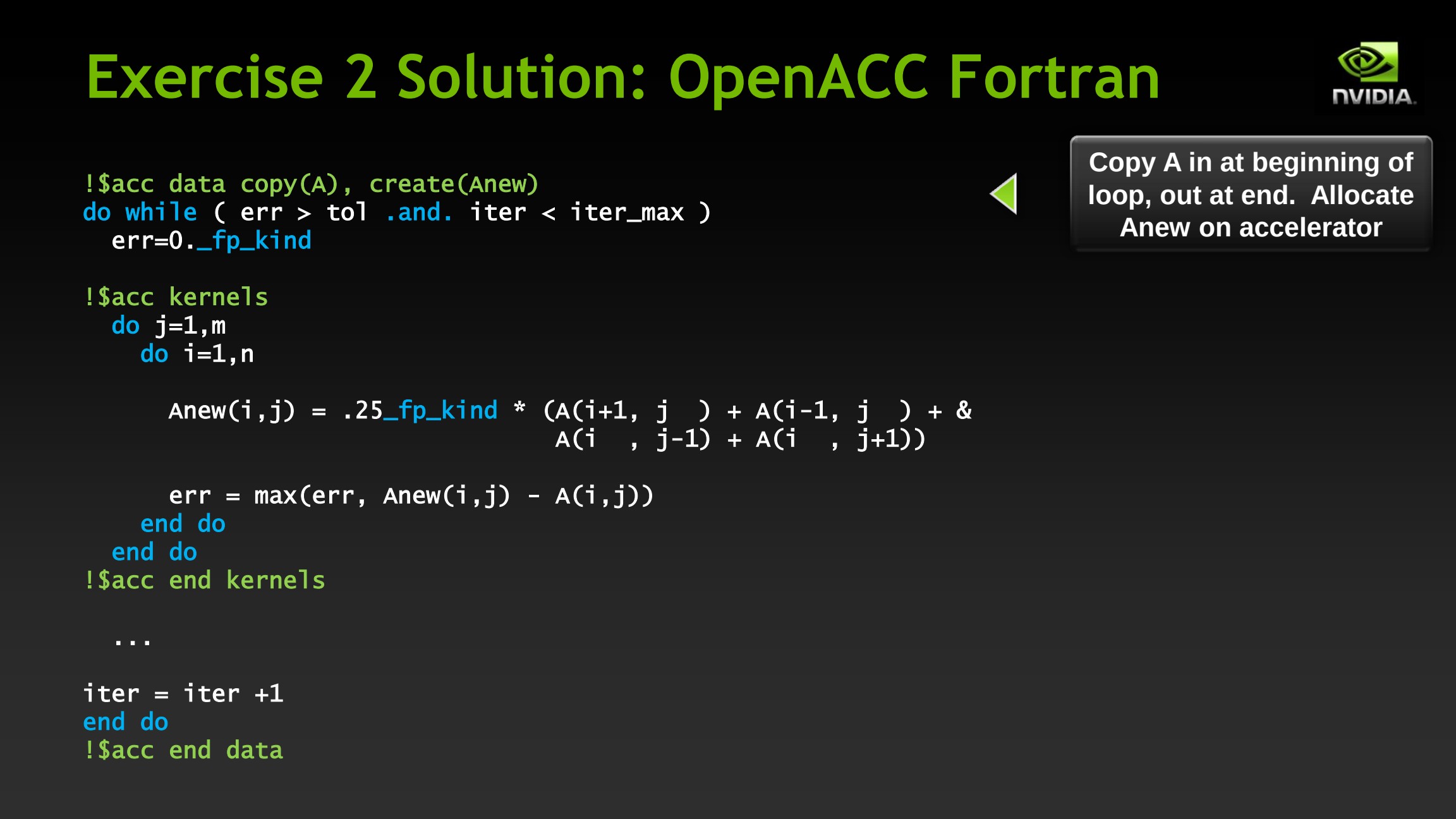
Exercise 2 Solution: OpenACC C



```
#pragma acc data copy(A), create(Anew)
while ( error > tol && iter < iter_max ) {
  error=0.0;

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

      error = max(error, abs(Anew[j][i] - A[j][i]);
    }
  }

#pragma acc kernels
  for( int j = 1; j < n-1; j++) {
    for( int i = 1; i < m-1; i++ ) {
      A[j][i] = Anew[j][i];
    }
  }

  iter++;
}
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

Exercise 2 Solution: OpenACC Fortran



```
!$acc data copy(A), create(Anew)
do while ( err > tol .and. iter < iter_max )
  err=0._fp_kind

!$acc kernels
  do j=1,m
    do i=1,n

      Anew(i,j) = .25_fp_kind * (A(i+1, j  ) + A(i-1, j  ) + &
                                 A(i  , j-1) + A(i  , j+1))

      err = max(err, Anew(i,j) - A(i,j))
    end do
  end do
!$acc end kernels

  ...

iter = iter +1
end do
!$acc end data
```

Copy A in at beginning of loop, out at end. Allocate Anew on accelerator

Exercise 2: Performance

CPU: Intel Xeon X5690
6 Cores @ 3.47GHz

GPU: NVIDIA Tesla M2090

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	105.6	--
CPU 2 OpenMP threads	62.5	1.69x
CPU 4 OpenMP threads	43.3	2.44x
CPU 6 OpenMP threads	41.2	2.56x
OpenACC GPU	10.50	3.92x

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

Further speedups

- OpenACC gives us 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
- Will tackle these in later exercises

Finding Parallelism in your code

- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: `restrict` keyword (C), `independent` clause
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Function calls within accelerated region must be inlineable.

Tips and Tricks

- (PGI) Use time option to learn where time is being spent
  - -ta=nvidia,time
- Eliminate pointer arithmetic
- Inline function calls in directives regions
  - (PGI): -inline or -inline,levels(<N>)
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with _OPENACC macro

OpenACC Learning Resources

- OpenACC info, specification, FAQ, samples, and more
  - http://openacc.org
- PGI OpenACC resources
  - http://www.pgroup.com/resources/accel.htm