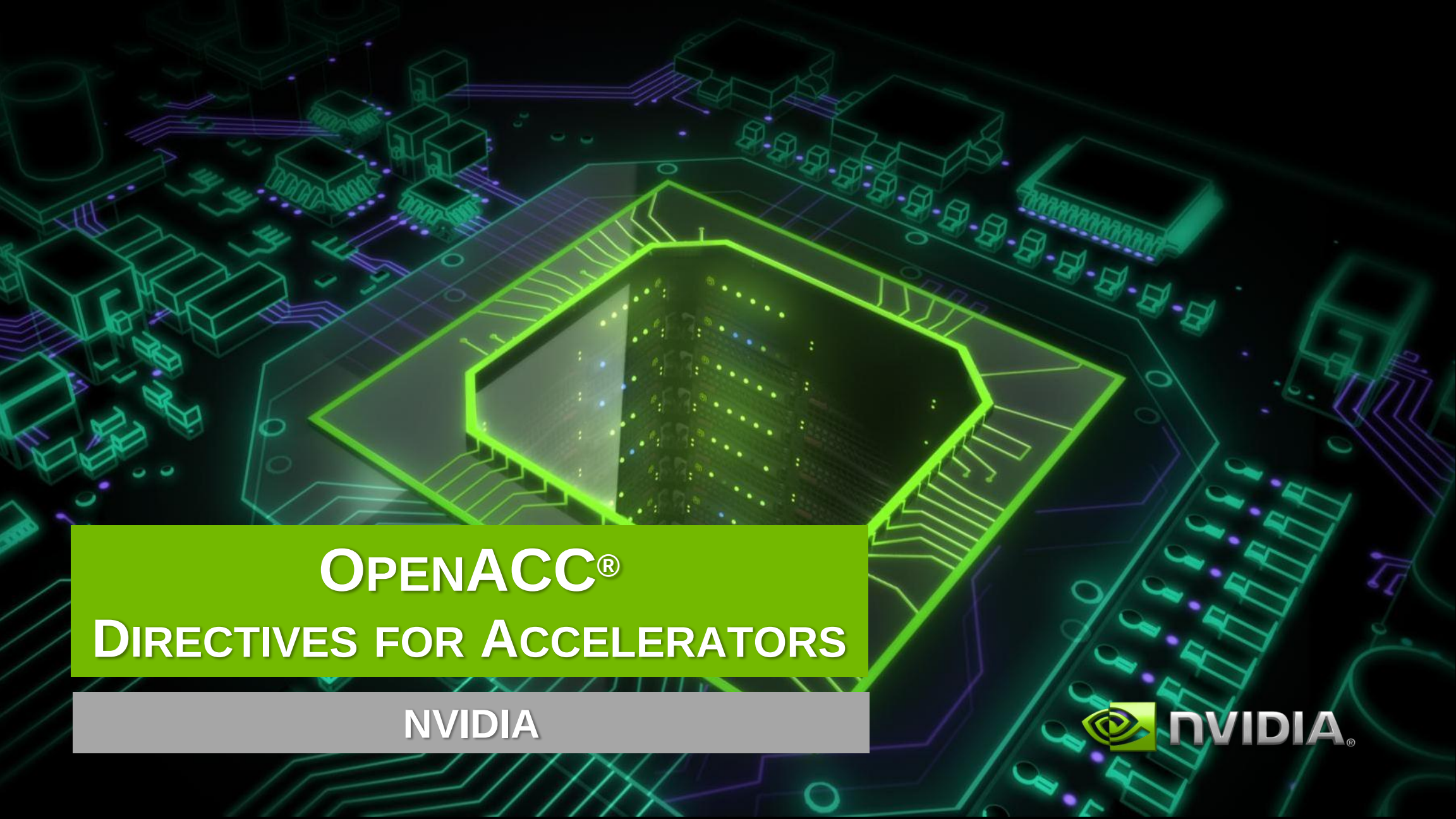
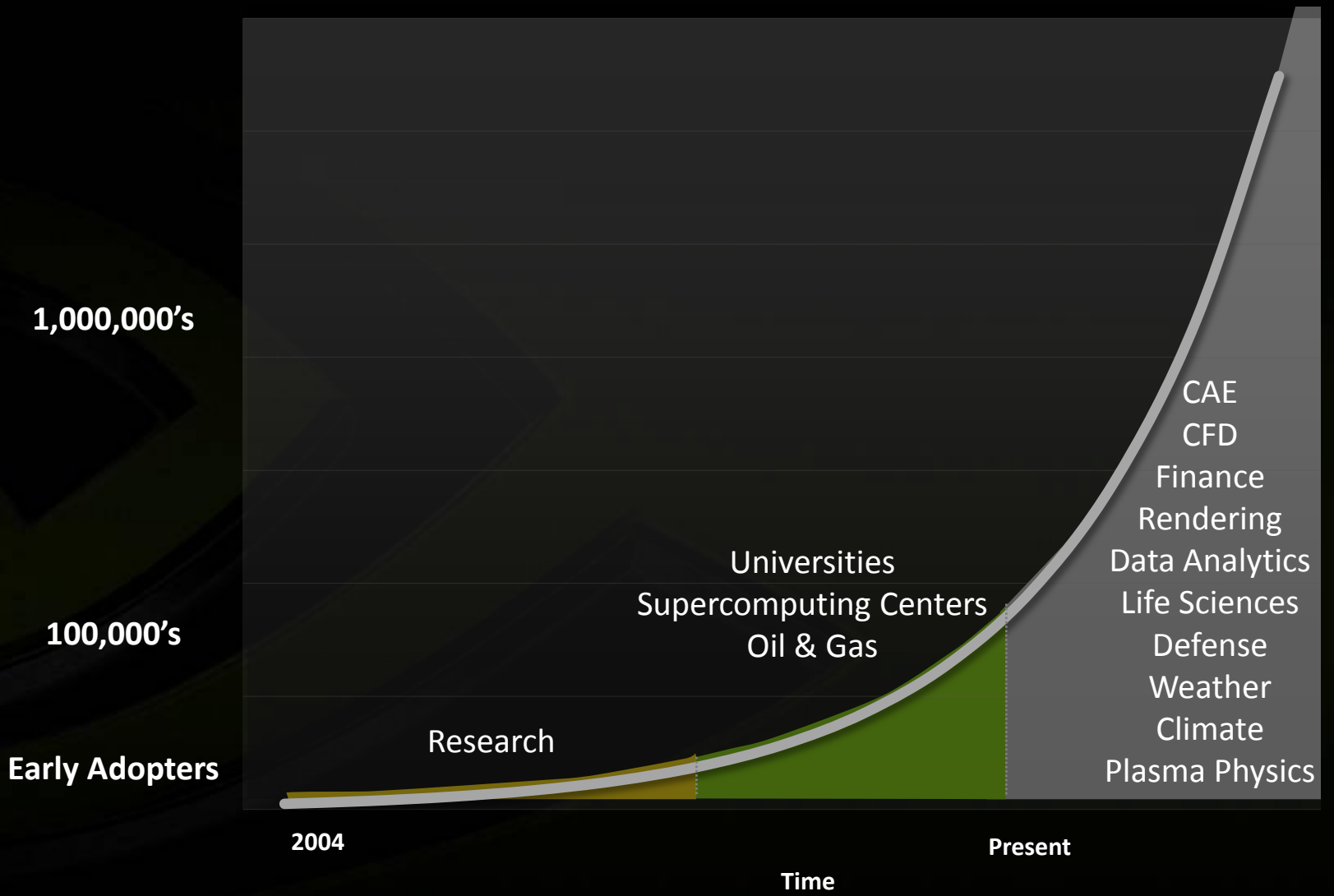


OpenACC®
Directives for Accelerators

NVIDIA

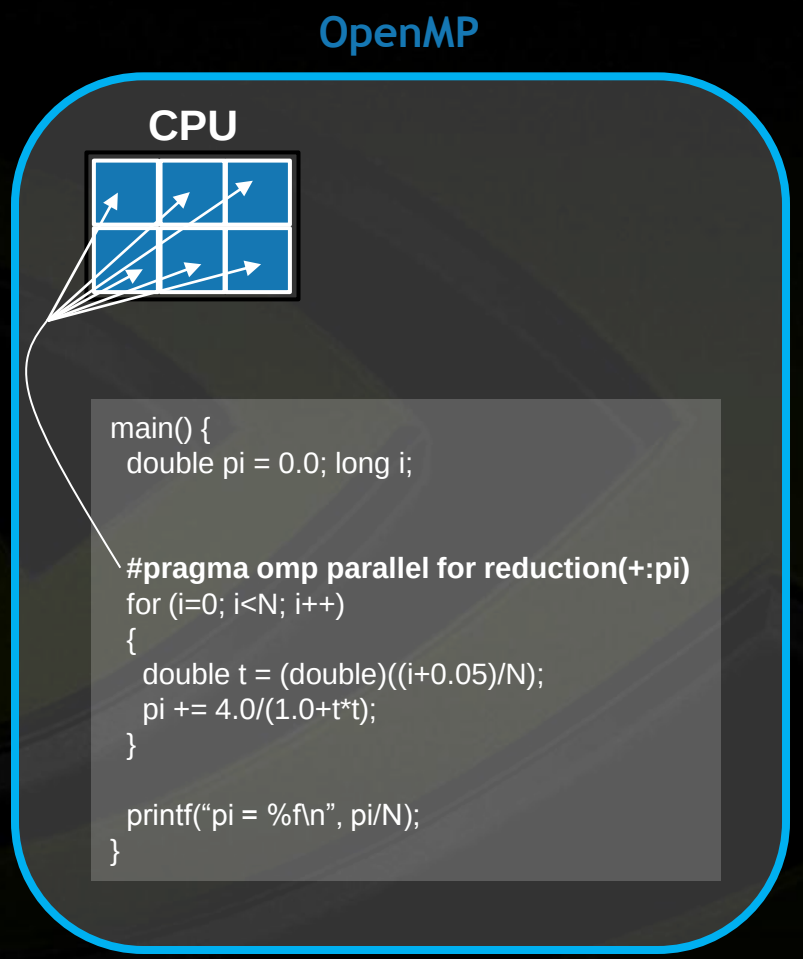
NVIDIA®

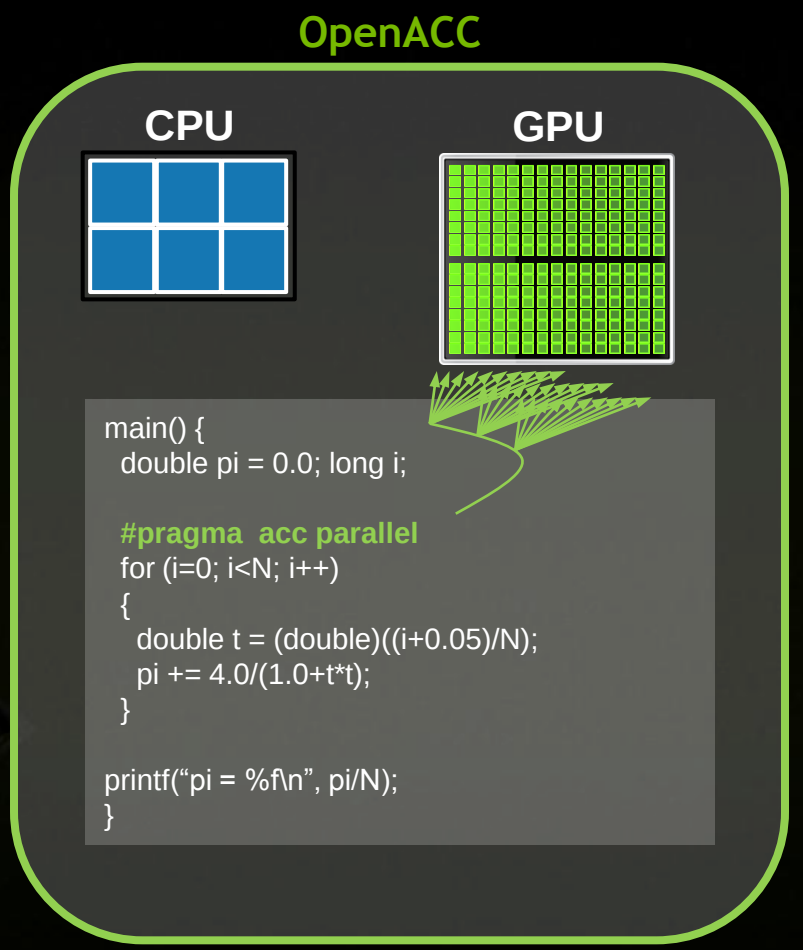
GPUs Reaching Broader Set of Developers



3 Ways to Accelerate Applications with GPUs

Directives: Add A Few Lines of Code





OpenACC:
Open Programming Standard for Parallel Computing

NVIDIA

NVIDIA | CRAY THE SUPERCOMPUTER COMPANY | PGI | CAPS

"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

Easy, Fast, Portable

"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

http://www.openacc-standard.org/

OpenACC

- 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

OpenACC Specification and Website

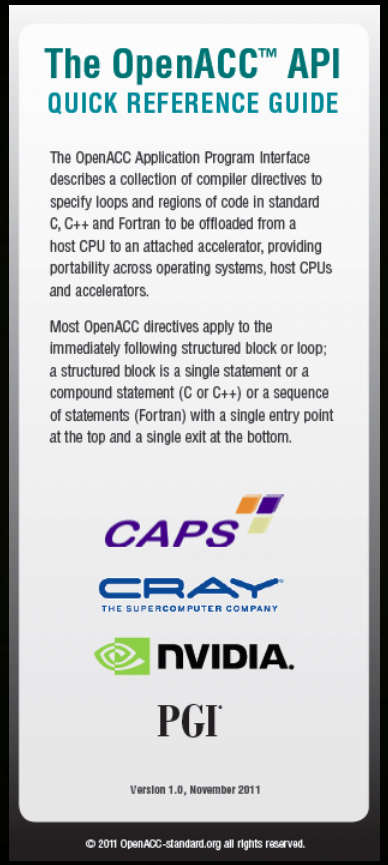
- Full OpenACC 1.0 Specification available online

  http://www.openacc-standard.org

- Quick reference card also available

- First implementations to appear mid-2012

- Current PGI, Cray, and CAPS compilers all have accelerator directives (precursors to OpenACC)



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.

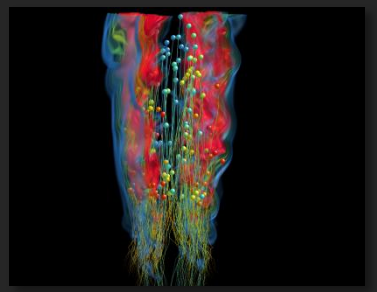
* Achieved using the PGI Accelerator Compiler

Focus on Exposing Parallelism

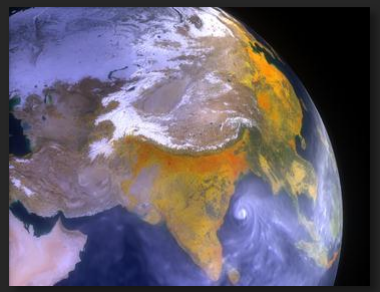
With Directives, tuning work focuses on *exposing parallelism*, which makes codes inherently better

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

S3D
Research more efficient combustion with next-generation fuels



- 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%



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

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

A simple example (Jacobi relaxation)

```
while ( error > tol && iter < iter_max ) {
    error=0.f;

    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25f * (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 elements of matrix

Calculate new value from neighbours

OpenMP CPU Implementation


```
while ( error > tol && iter < iter_max ) {
    error=0.f;
#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] = 0.25f * (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++;
```

Parallelise loops across CPU cores

OpenACC GPU Implementation

```
#pragma acc data copy(A, Anew)
while ( error > tol && iter < iter_max ) {
    error=0.f;
#pragma acc parallel
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
                                  + A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]);
        }
    }

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

Copy arrays into GPU within region

Parallelize loops with GPU kernels

Data automatically copied back to at end of region

Speed Up

Implementation	Device	Time (s)	Speedup
Single-threaded CPU	Xeon X5550	33.42	--
Multi-threaded CPU (8 threads)	Xeon X5550	17.81	1.88x
GPU accelerated	Tesla M2090	8.27	2.15x

Over 2x speed compared to 8-Core OpenMP code with just 3 directives.
Note: tested with PGI Accelerator directives

Jacobi Relaxation (Fortran)

Iterate until converged

Iterate across elements of matrix

Calculate new value from neighbours

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

  iter = iter + 1
  err = 0.0

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

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

OpenMP CPU Implementation (Fortran)

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

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

Parallelise code inside region

Close off region

OpenACC GPU Implementation (Fortran)

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

Copy arrays into GPU memory within region

Parallelise code inside region

Close off parallel region

Close off data region, copy data back

Finding Parallelism in your code

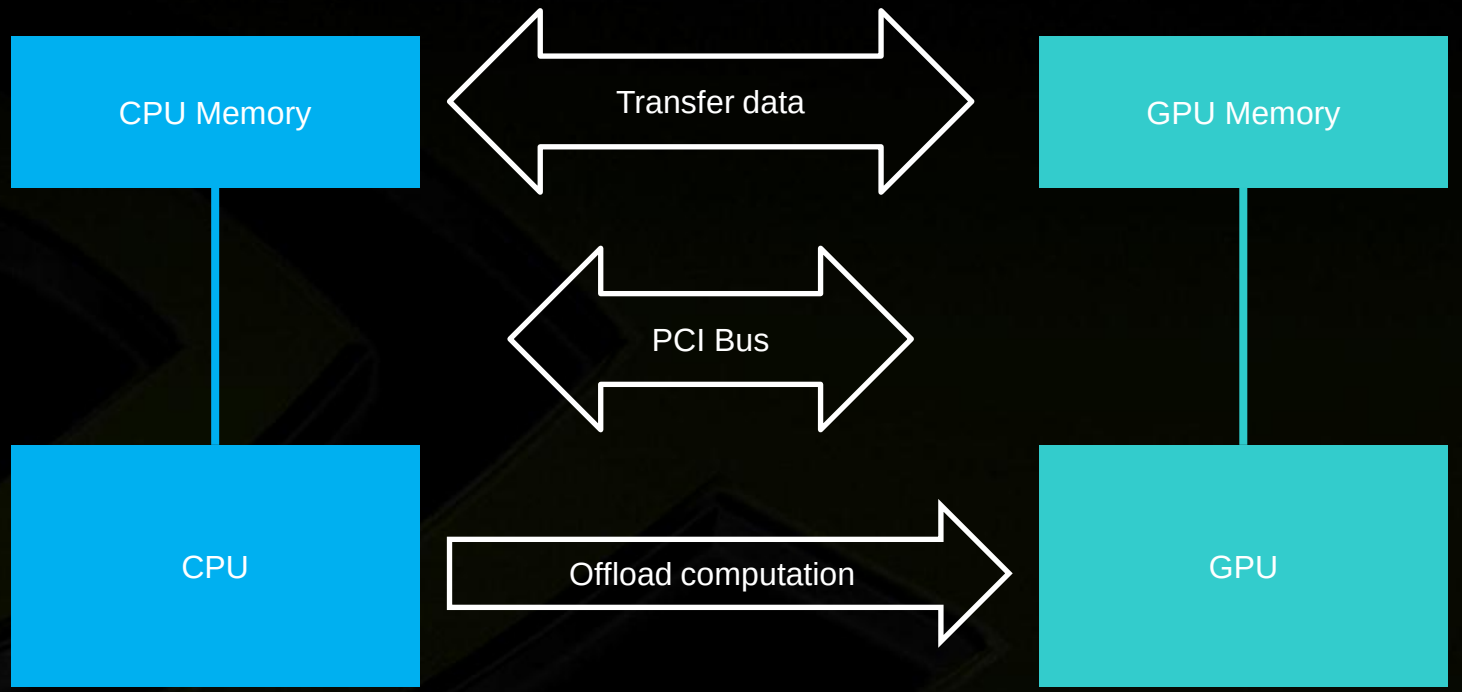
- (Nested) for loops are best for parallelization
- Large loop counts needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Pointer arithmetic should 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
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with _OPENACC macro

OpenACC Concepts

Basic Concepts



For efficiency, decouple data movement and compute off-load

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

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`.

Improved OpenACC GPU Implementation


```
#pragma acc data copyin(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.f;
#pragma acc parallel
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
                                  + A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]);
        }
    }

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

Improved OpenACC GPU Implementation


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

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

Update Directive

Fortran

```
!$acc update [clause …]
```

C

```
#pragma acc update [clause …]
```

Clauses

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

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

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

Work Management

Parallel Construct

Fortran

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

C

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

Clauses

```
if( condition )
async( expression )
num_gangs( expression )
num_workers( expression )
vector_length( expression )
```

```
private( list )
firstprivate( list )
reduction( operator:list )
```

Any data clause

Parallel Clauses

`num_gangs ( expression )`	Controls how many parallel gangs are created (CUDA `gridDim`).
`num_workers ( expression )`	Controls how many workers are created in each gang (CUDA `blockDim`).
`vector_length ( list )`	Controls vector length of each worker (SIMD execution).
`private( list )`	A copy of each variable in list is allocated to each gang.
`firstprivate ( list )`	`private` variables initialized from host.
`reduction( operator:list )`	`private` variables combined across gangs.

More Parallelism (C)


```
#pragma acc data copyin(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.f;
#pragma acc parallel reduction( max:error )
    for( int j = 1; j < n-1; j++) {
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
                                  + A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]);
        }
    }

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

Find maximum over all iterations

More Parallelism (Fortran)

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

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

Find maximum over all iterations

Add second parallel region inside data region

Kernels Construct

Fortran

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

C

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

Clauses

`if( condition )`

`async( expression )`

Any data clause

Kernels Construct

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

Loop Construct

Fortran

```
!$acc loop [clause …]
    loop
!$acc end loop
```

C

```
#pragma acc loop [clause …]
    { loop }
```

Combined directives

```
!$acc parallel loop [clause …]
!$acc kernels loop [clause …]
```

```
!$acc parallel loop [clause …]
!$acc kernels loop [clause …]
```

Detailed control of the parallel execution of the following loop.

Loop Clauses

`collapse( n )`	Applies directive to the following `n` nested loops.
`seq`	Executes the loop sequentially on the GPU.
`private( list )`	A copy of each variable in list is created for each iteration of the loop.
`reduction( operator:list )`	`private` variables combined across iterations.

Loop Clauses Inside parallel Region

`gang`	Shares iterations across the gangs of the parallel region.
`worker`	Shares iterations across the workers of the gang.
`vector`	Execute the iterations in SIMD mode.

Loop Clauses Inside kernels Region

`gang [( num_gangs )]`	Shares iterations across across at most *num_gangs* gangs.
`worker [( num_workers )]`	Shares iterations across at most *num_workers* of a single gang.
`vector [( vector_length )]`	Execute the iterations in SIMD mode with maximum *vector_length*.
`independent`	Specify that the loop iterations are independent.

More Performance (C)

```
#pragma acc data copyin(A), create(Anew)
while ( error > tol && iter < iter_max ) {
    error=0.f;
#pragma acc kernels loop reduction( max:error ), gang(32), worker(16)
    for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16), worker(32)
        for(int i = 1; i < m-1; i++) {
            Anew[j][i] = 0.25f * (A[j][i+1] + A[j][i-1]
                                  + A[j-1][i] + A[j+1][i]);
            error = max(error, abs(Anew[j][i] - A[j][i]);
    }}
#pragma acc kernels loop
    for( int j = 1; j < n-1; j++) {
#pragma acc loop gang(16), worker(32)
        for( int i = 1; i < m-1; i++ ) {
            A[j][i] = Anew[j][i];
    }}
    iter++;  }
```

Combined 65% faster than default schedule

More Performance (Fortran)

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

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

30% faster than default schedule

Other Syntax

Other Directives

`cache` construct	Cache data in software managed data cache (CUDA shared memory).
`host_data` construct	Makes the address of device data available on the host.
`wait` directive	Waits for asynchronous GPU activity to complete.
`declare` directive	Specify that data is to allocated in device memory for the duration of an implicit data region created during the execution of a subprogram.

Runtime Library Routines

Fortran

```
use openacc
#include "openacc_lib.h"

acc_get_num_devices
acc_set_device_type
acc_get_device_type
acc_set_device_num
acc_get_device_num
acc_async_test
acc_async_test_all
```

C

```
#include "openacc.h"

acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```

Environment and Conditional Compilation

`ACC_DEVICE` *`device`*	Specifies which device type to connect to.
`ACC_DEVICE_NUM` *`num`*	Specifies which device number to connect to.
`_OPENACC`	Preprocessor directive for conditional compilation.