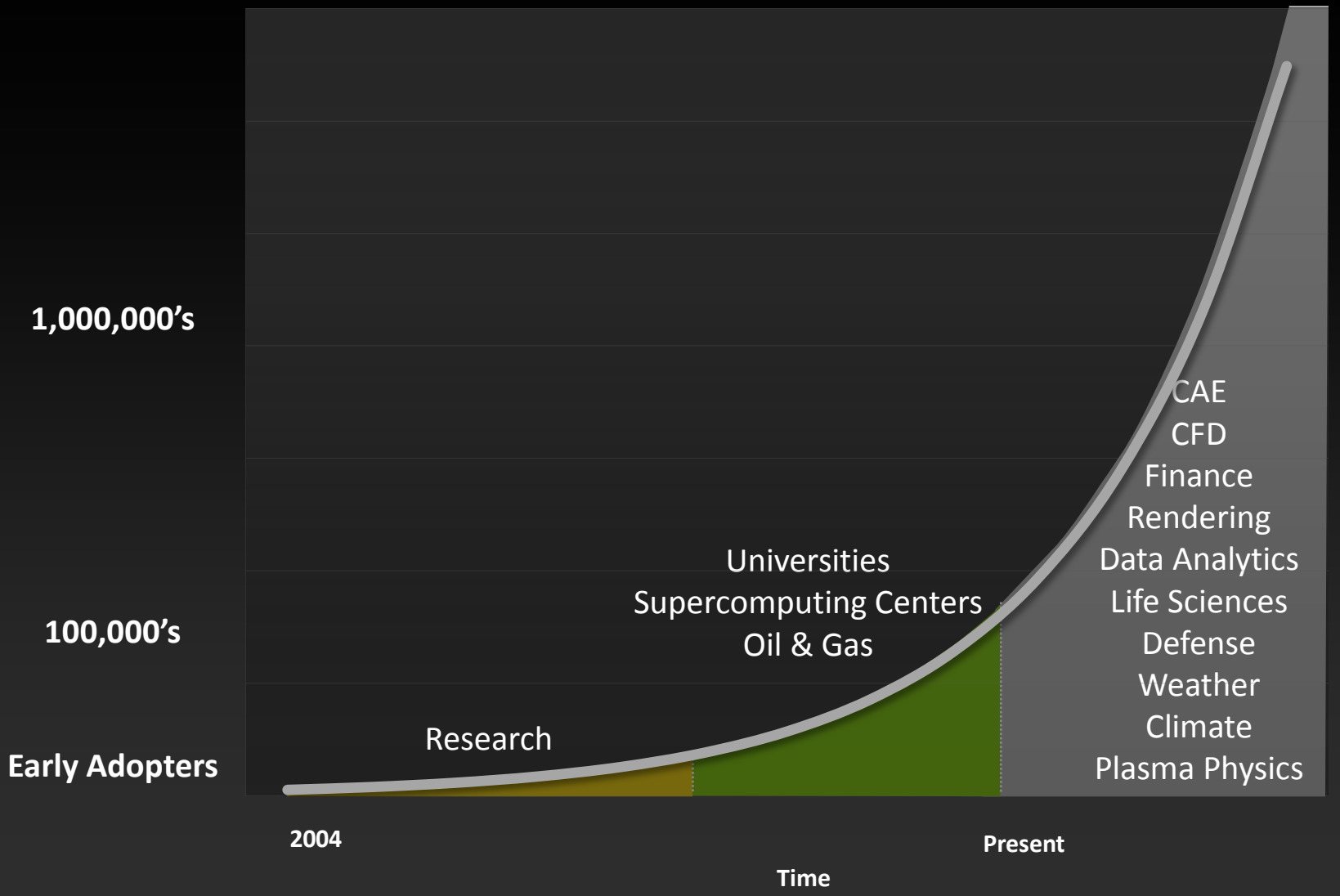
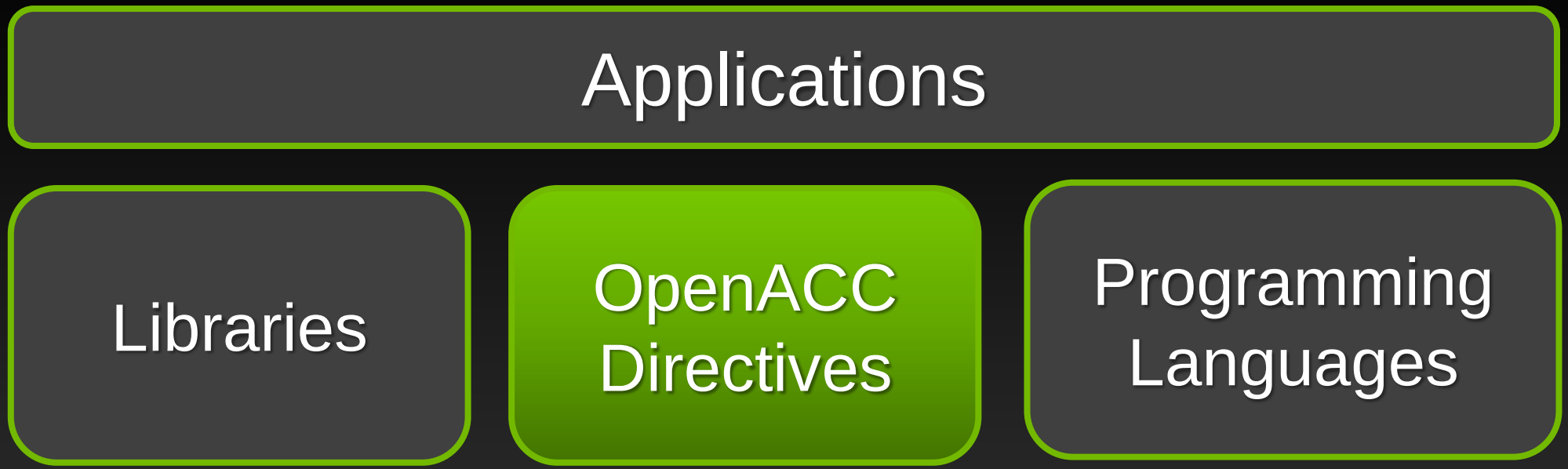

Getting Started with OpenACC
Jeff Larkin, NVIDIA
NVIDIA

GPUs Reaching Broader Set of Developers



3 Ways to Accelerate Applications



OpenACC
The Standard for GPU Directives

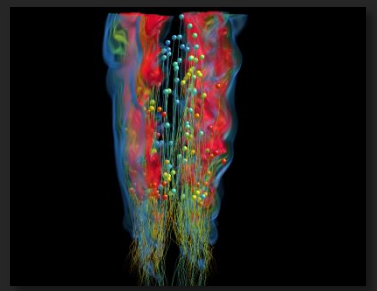
- **Simple:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive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

Programmer Focuses on Expressing Parallelism

(not on platform-specific optimization)

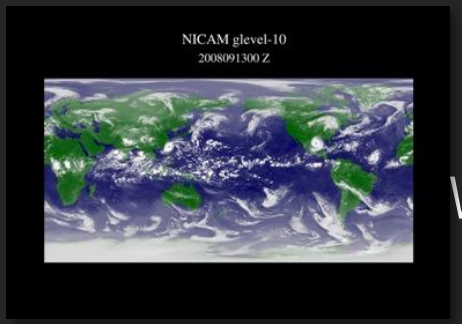
Example: Application tuning work using directives at ORNL and Tokyo Tech

(comparing CPU+GPU vs. dual-CPU nodes)



S3D
Combustion

- Tuned top 3 kernels for GPUs (90% of runtime)
- End result: 2.2X faster with K20X vs. dual AMD node
  - Kepler is 6X faster than Fermi
- ***Improved performance of CPU-only version by 50%***



NICAM
Weather/Climate

- Tuned top kernels using CUDA, then OpenACC
- CUDA result: 3.1x faster on GPU vs. CPU node
- OpenACC result (*preliminary*) = 69-77% of CUDA
  - More portable, more maintainable
  - Full OpenACC port in progress

Results from Cray/ORNL and Tokyo Tech

OpenACC is not *GPU Programming.*

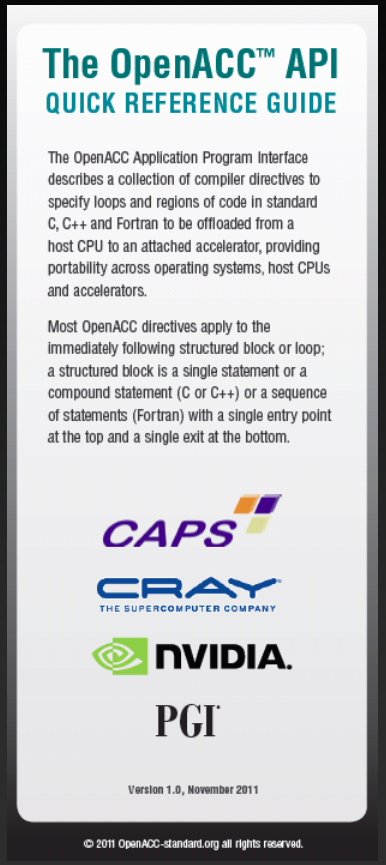
OpenACC is *Expressing Parallelism* in your code.

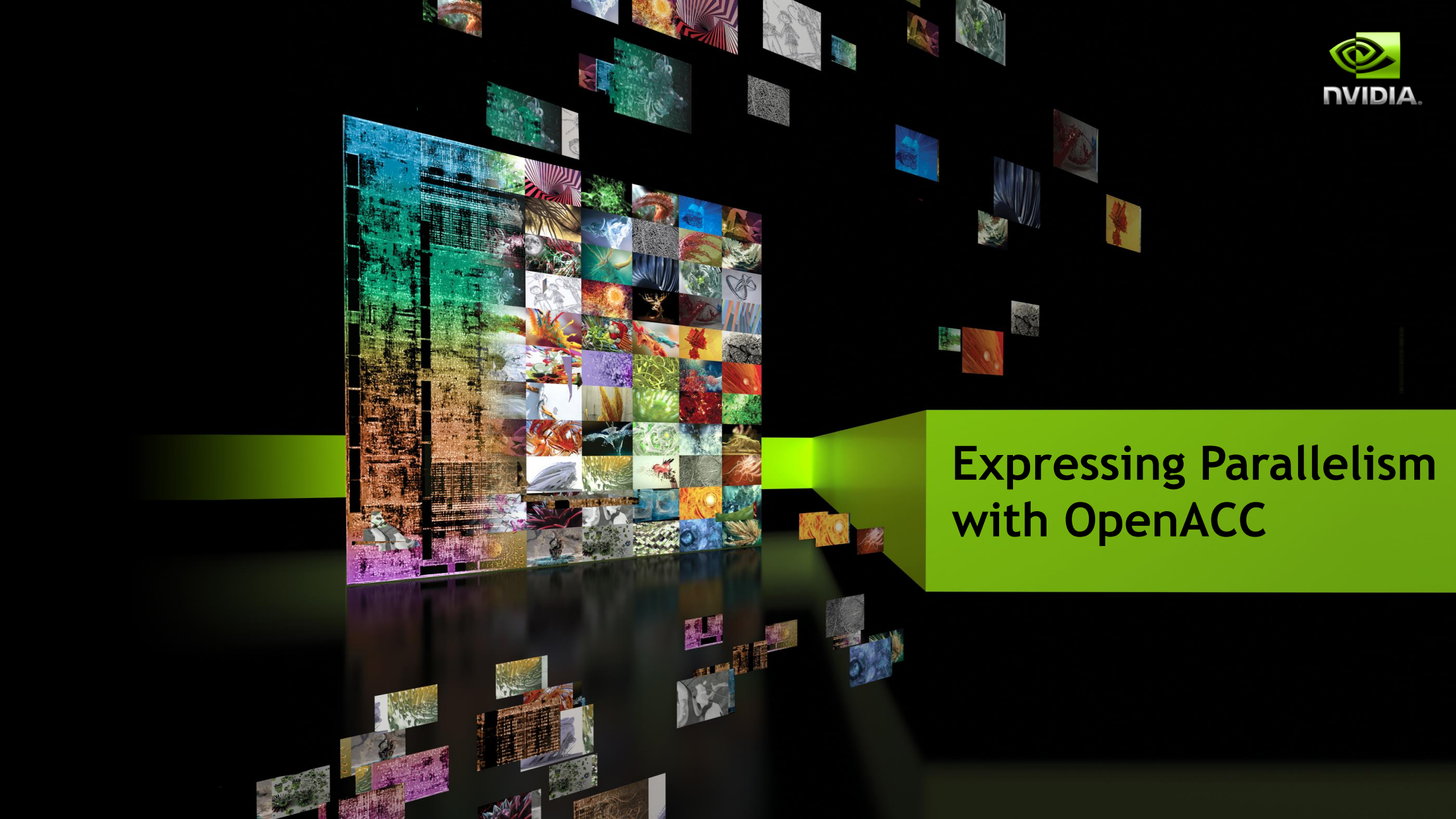
OpenACC Specification and Website

- Full OpenACC 2.0 Specification available online

www.openacc.org

- Quick reference card also available
- Compilers available now from PGI, Cray, and CAPS





Expressing Parallelism with OpenACC

A Very Simple Exercise: SAXPY

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{

  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(n), y(n), a
  integer :: n, i


  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo

end subroutine saxpy

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

A Very Simple Exercise: SAXPY OpenMP

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma omp parallel for
  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(n), y(n), a
  integer :: n, i

!$omp parallel do
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$omp end parallel do
end subroutine saxpy

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

A Very Simple Exercise: SAXPY OpenACC

SAXPY in C

```
void saxpy(int n,
           float a,
           float *x,
           float *restrict y)
{
#pragma acc parallel loop
  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);
...
```

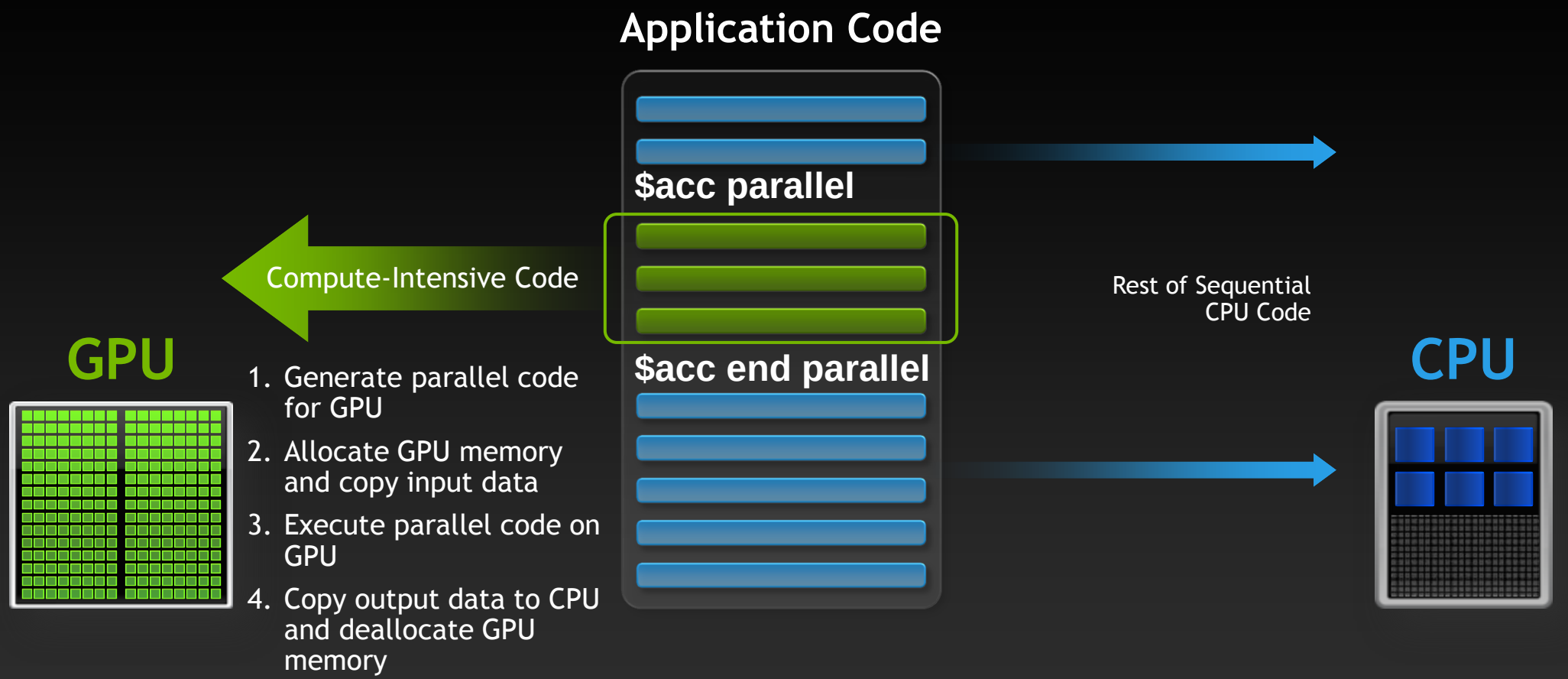
SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(n), y(n), a
  integer :: n, i

!$acc parallel loop
  do i=1,n
    y(i) = a*x(i)+y(i)
  enddo
!$acc end parallel loop
end subroutine saxpy

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

OpenACC Execution Model



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

- Common Clauses

  `if(condition), async(handle)`

OpenACC parallel Directive

Programmer identifies a loop as having parallelism, compiler generates a parallel kernel for that loop.

```
$!acc parallel loop
   do i=1,n
     y(i) = a*x(i)+y(i)
   enddo
$!acc end parallel loop
```

Parallel kernel

Kernel:
A function that runs in parallel on the GPU

*Most often `parallel` will be used as `parallel loop`.

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 parallel loop
for (int i = 0; i < n; ++i)
    y[i] = a * x[i] + y[i];
}
```

```
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 (PGI)

- C:

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

- Fortran:

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

- Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
     11, Accelerator kernel generated
         13, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
     11, Generating present_or_copyin(x[0:n])
         Generating present_or_copy(y[0:n])
         Generating NVIDIA code
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
         Generating compute capability 3.0 binary
```

Run

- The PGI compiler provides automatic instrumentation when `PGI_ACC_TIME=1` at runtime

```
Accelerator Kernel Timing data
/home/jlarkin/kernels/saxpy/saxpy.c
  saxpy  NVIDIA  devicenum=0
        time(us): 3,256
        11: data copyin reached 2 times
             device time(us): total=1,619 max=892 min=727 avg=809
        11: kernel launched 1 times
            grid: [4096]  block: [256]
             device time(us): total=714 max=714 min=714 avg=714
            elapsed time(us): total=724 max=724 min=724 avg=724
        15: data copyout reached 1 times
             device time(us): total=923 max=923 min=923 avg=923
```

Another approach: `kernels` construct

- The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```
!$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

The compiler identifies 2 parallel loops and generates 2 kernels.

OpenACC parallel vs. kernels

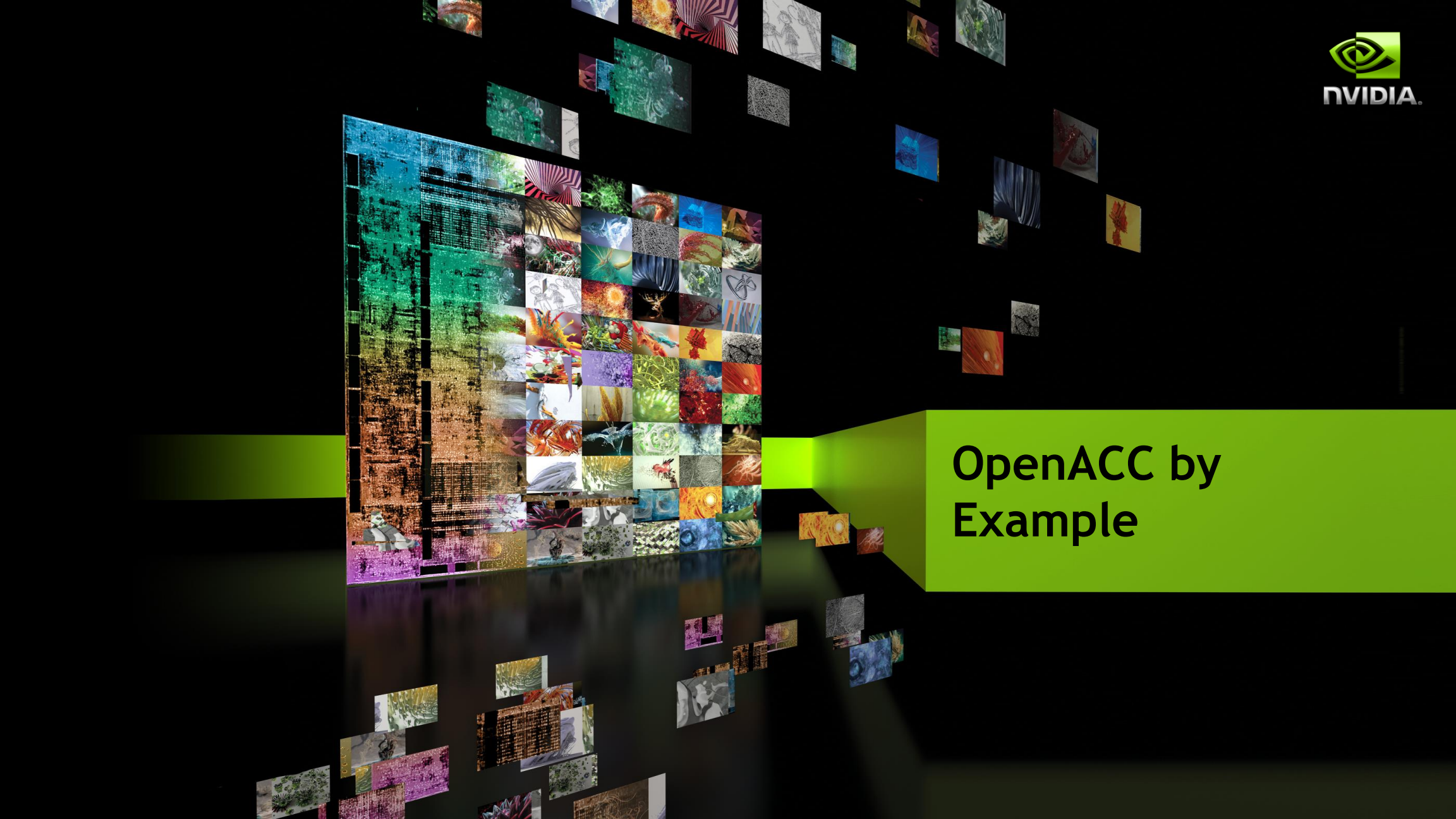
PARALLEL

- Requires analysis by programmer to ensure safe parallelism
- Straightforward path from OpenMP

KERNELS

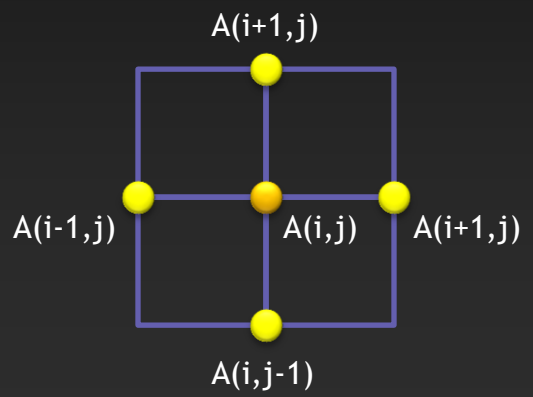
- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover larger area of code with single directive

Both approaches are equally valid and can perform equally well.


NVIDIA
OpenACC by
Example

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 ( err > tol && iter < iter_max ) {
  err=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]);

      err = max(err, 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: OpenMP C Code

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

#pragma omp parallel for shared(m, n, Anew, A) reduction(max:err)
  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]);

      err = max(err, 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

Jacobi Iteration: OpenACC C Code

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

#pragma acc parallel loop reduction(max:err)
  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]);

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

#pragma acc parallel loop
  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 nest on GPU

Parallelize loop nest on GPU

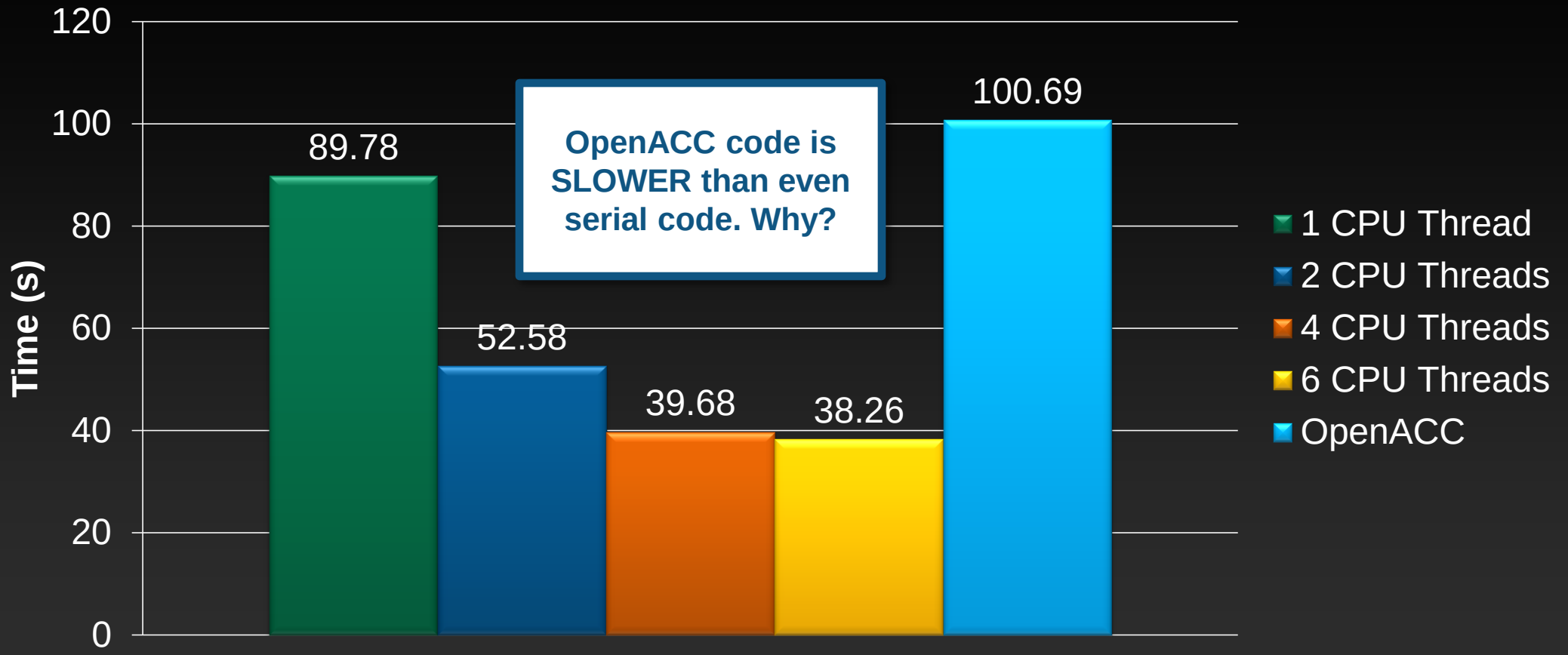
PGI Accelerator Compiler output (C)


```
pgcc -Minfo=all -ta=nvidia:5.0,cc3x -acc -Minfo=accel -o laplace2d_acc laplace2d.c
main:
     56, Accelerator kernel generated
         57, #pragma acc loop gang /* blockIdx.x */
         59, #pragma acc loop vector(256) /* threadIdx.x */
     56, Generating present_or_copyin(A[0:][0:])
         Generating present_or_copyout(Anew[1:4094][1:4094])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     59, Loop is parallelizable
     68, Accelerator kernel generated
         69, #pragma acc loop gang /* blockIdx.x */
         71, #pragma acc loop vector(256) /* threadIdx.x */
     68, Generating present_or_copyout(A[1:4094][1:4094])
         Generating present_or_copyin(Anew[1:4094][1:4094])
         Generating NVIDIA code
         Generating compute capability 3.0 binary
     71, Loop is parallelizable
```

Execution Time (lower is better)

CPU: Intel i7-3930K
6 Cores @ 3.20GHz

GPU: NVIDIA Tesla K20



What went wrong?

- Set `PGI_ACC_TIME` environment variable to '1'

```
Accelerator Kernel Timing data
/home/jlarkin/openacc-workshop/exercises/001-laplace2D-kernels/laplace2d.c
  main  NVIDIA  devicenum=0
        time(us): 93,201,190
        56: data copyin reached 1000 times
             device time(us): total=23,049,452 max=28,928 min=22,761 avg=23,049
        56: kernel launched 1000 times
            grid: [4094]  block: [256]
             device time(us): total=2,609,928 max=2,812 min=2,593
            elapsed time(us): total=2,872,585 max=3,022 min=2,642
        56: reduction kernel launched 1000 times
            grid: [1]  block: [256]
             device time(us): total=19,218 max=724 min=16 avg=19
            elapsed time(us): total=29,070 max=734 min=26 avg=29
        68: data copyin reached 1000 times
             device time(us): total=23,888,588 max=33,546 min=23,378 avg=23,888
        68: kernel launched 1000 times
            grid: [4094]  block: [256]
             device time(us): total=2,398,101 max=2,961 min=2,137 avg=2,398
            elapsed time(us): total=2,407,481 max=2,971 min=2,146 avg=2,407
        68: data copyout reached 1000 times
             device time(us): total=20,664,362 max=27,788 min=20,511 avg=20,664
        77: data copyout reached 1000 times
             device time(us): total=20,571,541 max=24,837 min=20,521 avg=20,571
```

23 seconds

2.6 seconds

0.19 seconds

23.9 seconds

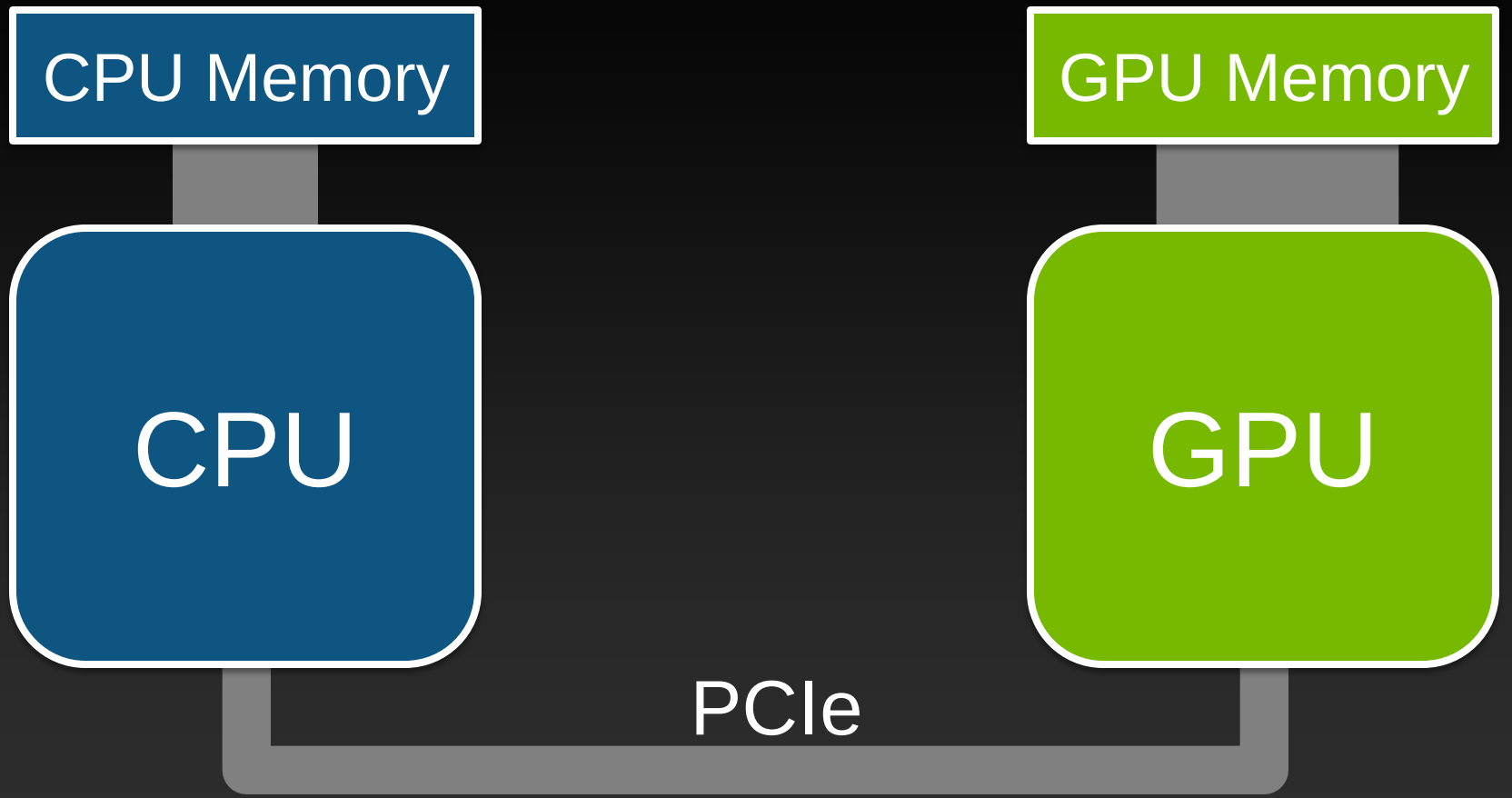
2.4 seconds

27.8 seconds

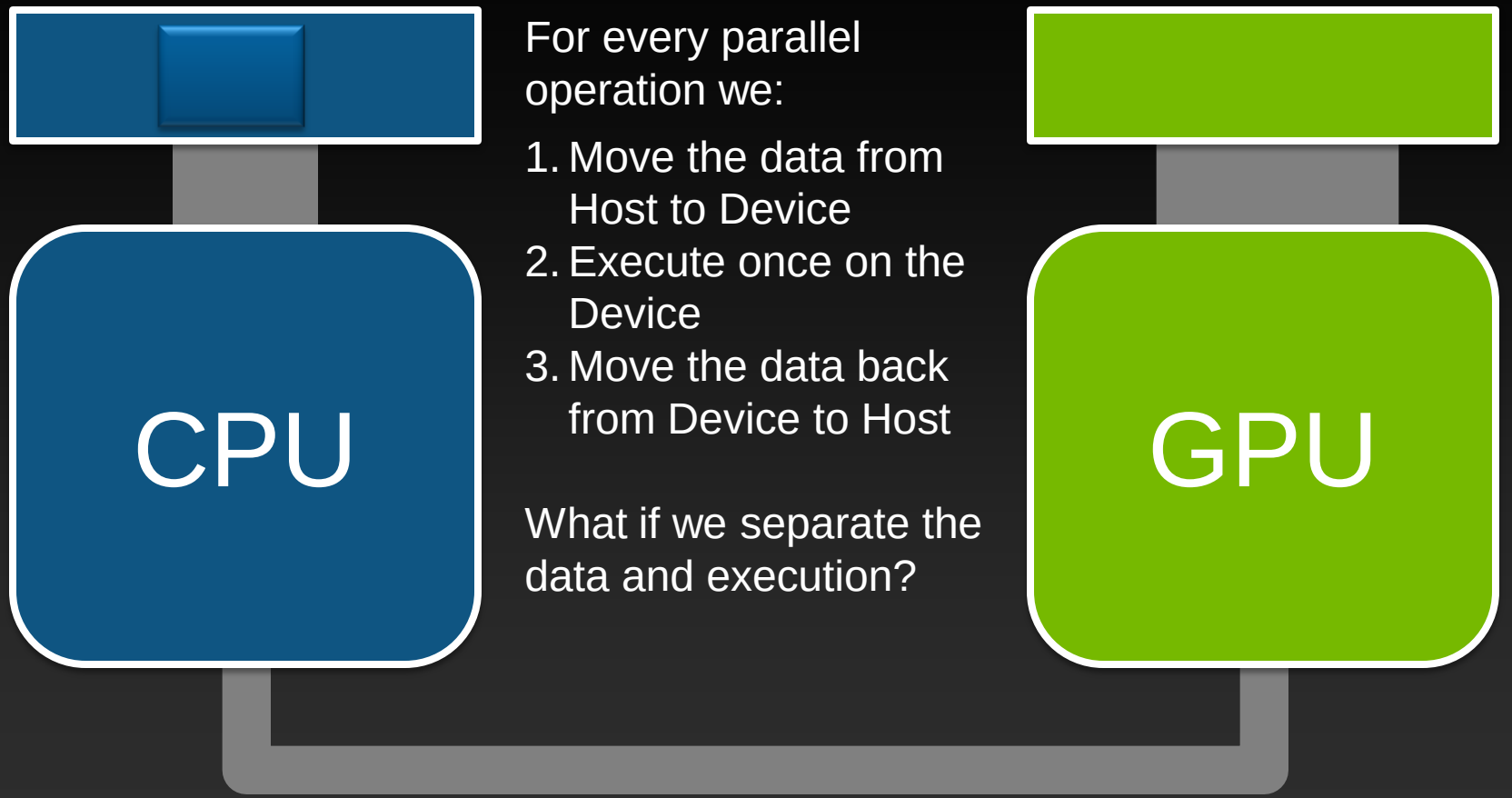
24.8 seconds

Huge Data Transfer Bottleneck!
Computation: 5.19 seconds
Data movement: 74.7 seconds

Offloading a Parallel Kernel



Offloading a Parallel Kernel

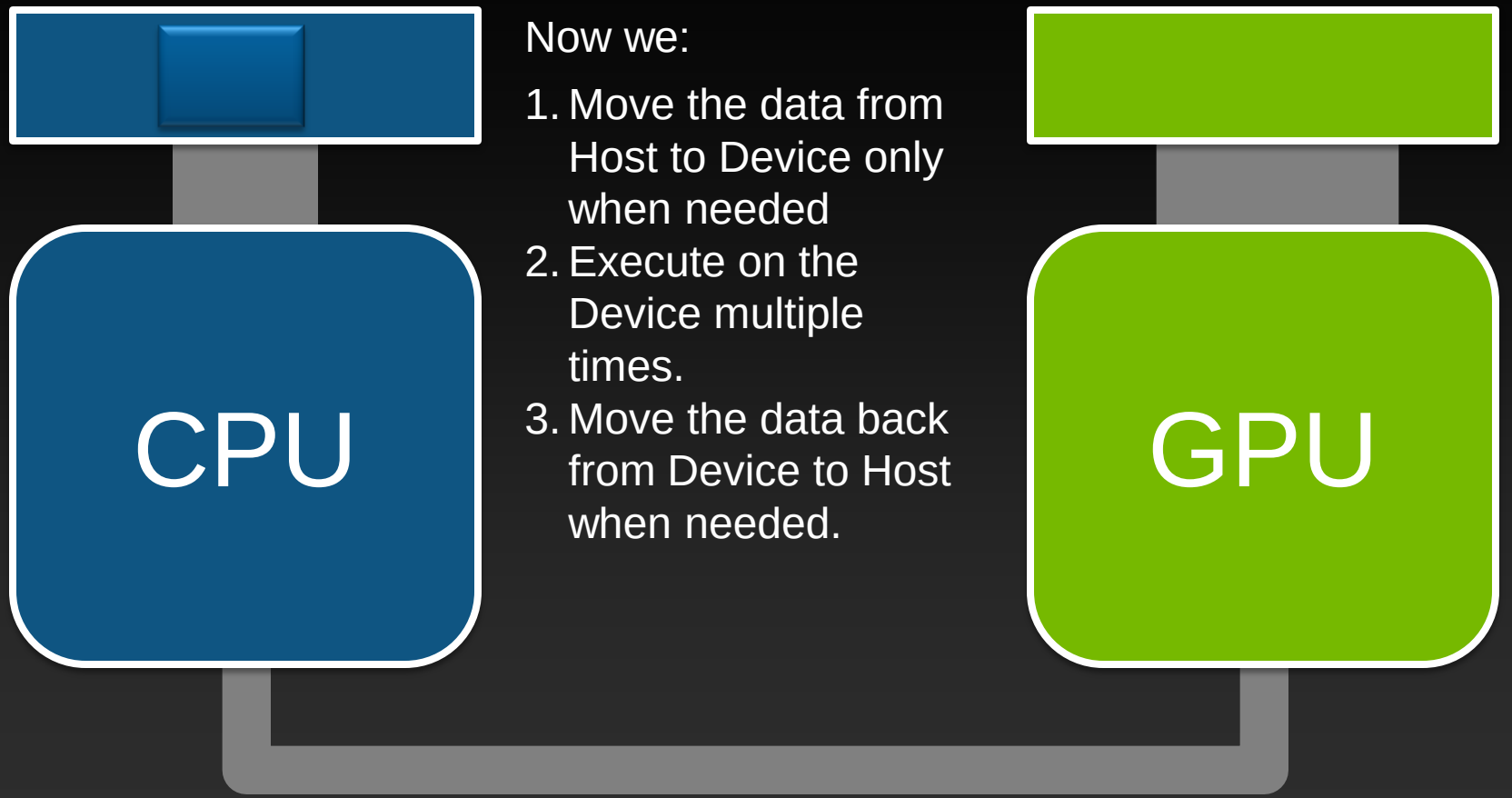


For every parallel operation we:

1. Move the data from Host to Device
2. Execute once on the Device
3. Move the data back from Device to Host

What if we separate the data and execution?

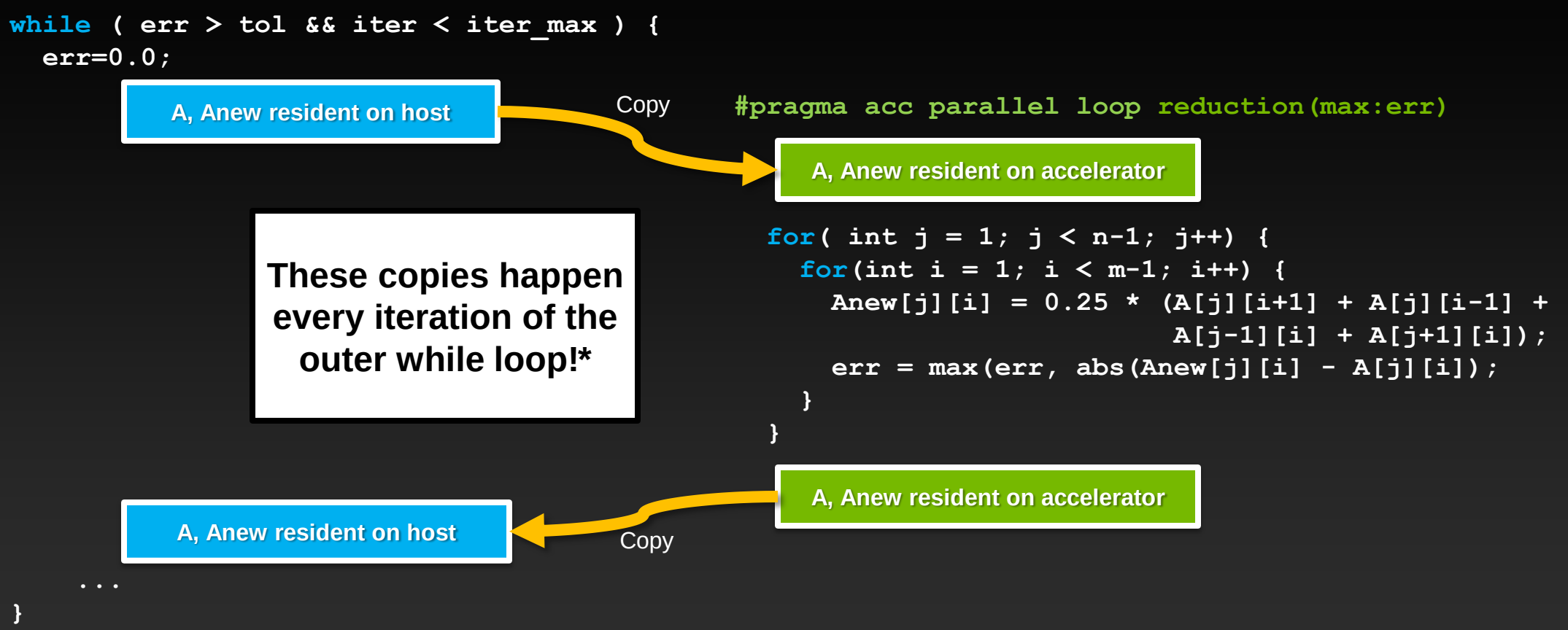
Separating Data from Computation



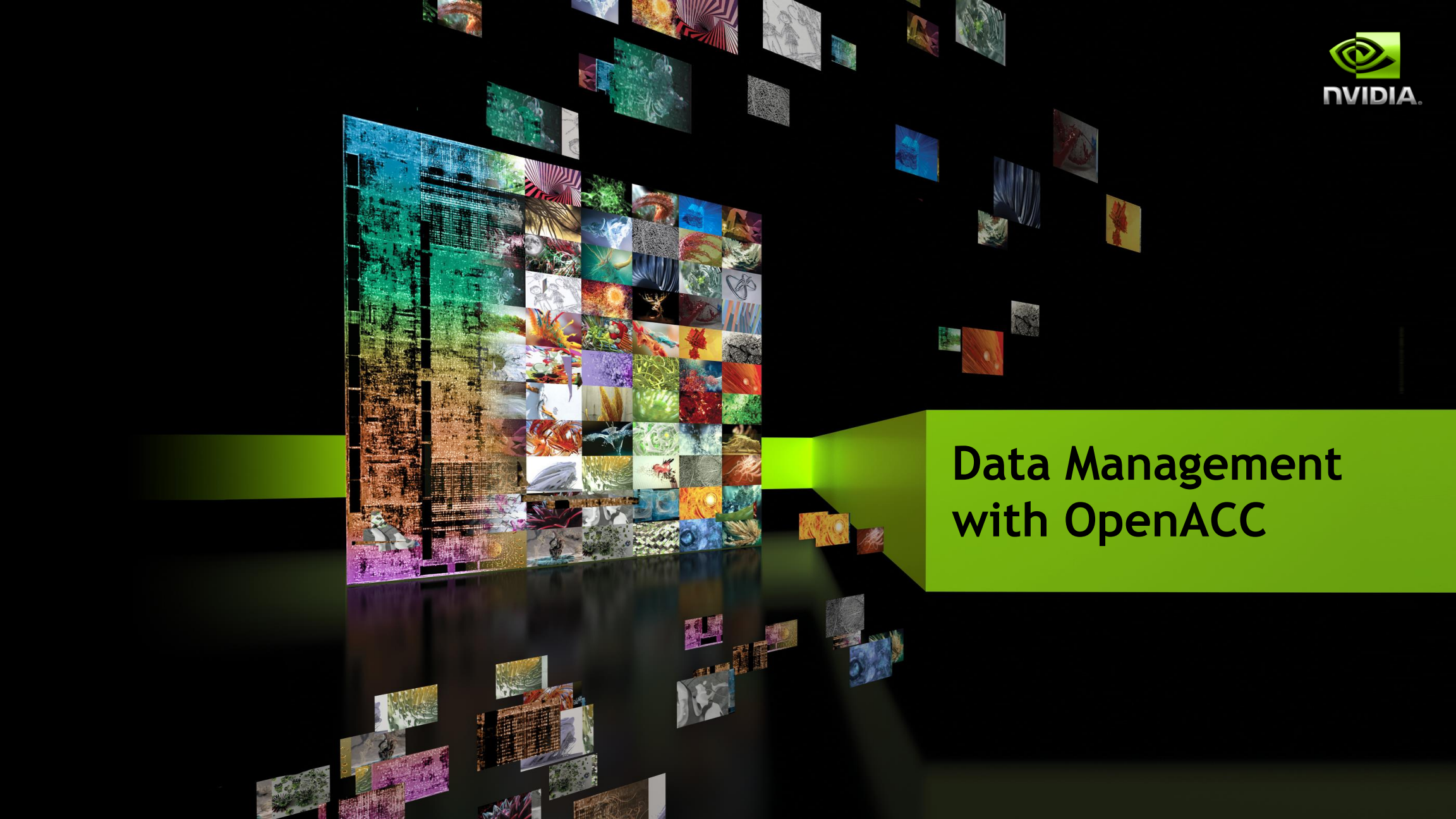
Now we:

1. Move the data from Host to Device only when needed
2. Execute on the Device multiple times.
3. Move the data back from Device to Host when needed.

Excessive Data Transfers



And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!


NVIDIA
Data Management
with OpenACC

Defining data regions

- The `data` construct defines a region of code in which GPU arrays remain on the GPU and are shared among all kernels in that region.

```
!$acc data
      !$acc parallel loop
       …

      !$acc parallel loop
       …
!$acc end data
```

Data Region

Arrays used within the data region will remain on the GPU until the end of the data region.

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]), copyout(b[s/4:3*s/4])
```

- Fortran

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

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

Jacobi Iteration: Data Directives

- Task: use `acc data` to minimize transfers in the Jacobi example

Jacobi Iteration: OpenACC C Code


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

#pragma acc parallel loop reduction(max:err)
  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]);

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

#pragma acc parallel loop
  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

Did it help?

- Set `PGI_ACC_TIME` environment variable to '1'

```
Accelerator Kernel Timing data
/home/jlarkin/openacc-workshop/exercises/001-laplace2D-kernels/laplace2d.c
  main  NVIDIA  devicenum=0
        time(us): 4,802,950
        51: data copyin reached 1 times
             device time(us): total=22,768 max=22,768 min=22,768 avg=22,768
        57: kernel launched 1000 times
            grid: [4094]  block: [256]
             device time(us): total=2,611,387 max=2,817 min=2,593 avg=2,611
            elapsed time(us): total=2,620,044 max=2,900 min=2,601 avg=2,620
        57: reduction kernel launched 1000 times
            grid: [1]  block: [256]
             device time(us): total=18,083 max=842 min=16 avg=18
            elapsed time(us): total=27,731 max=852 min=25 avg=27
        69: kernel launched 1000 times
            grid: [4094]  block: [256]
             device time(us): total=2,130,162 max=2,599 min=2,112 avg=2,130
            elapsed time(us): total=2,139,919 max=2,712 min=2
        83: data copyout reached 1 times
             device time(us): total=20,550 max=20,550 min=20,550 avg=20,550
```

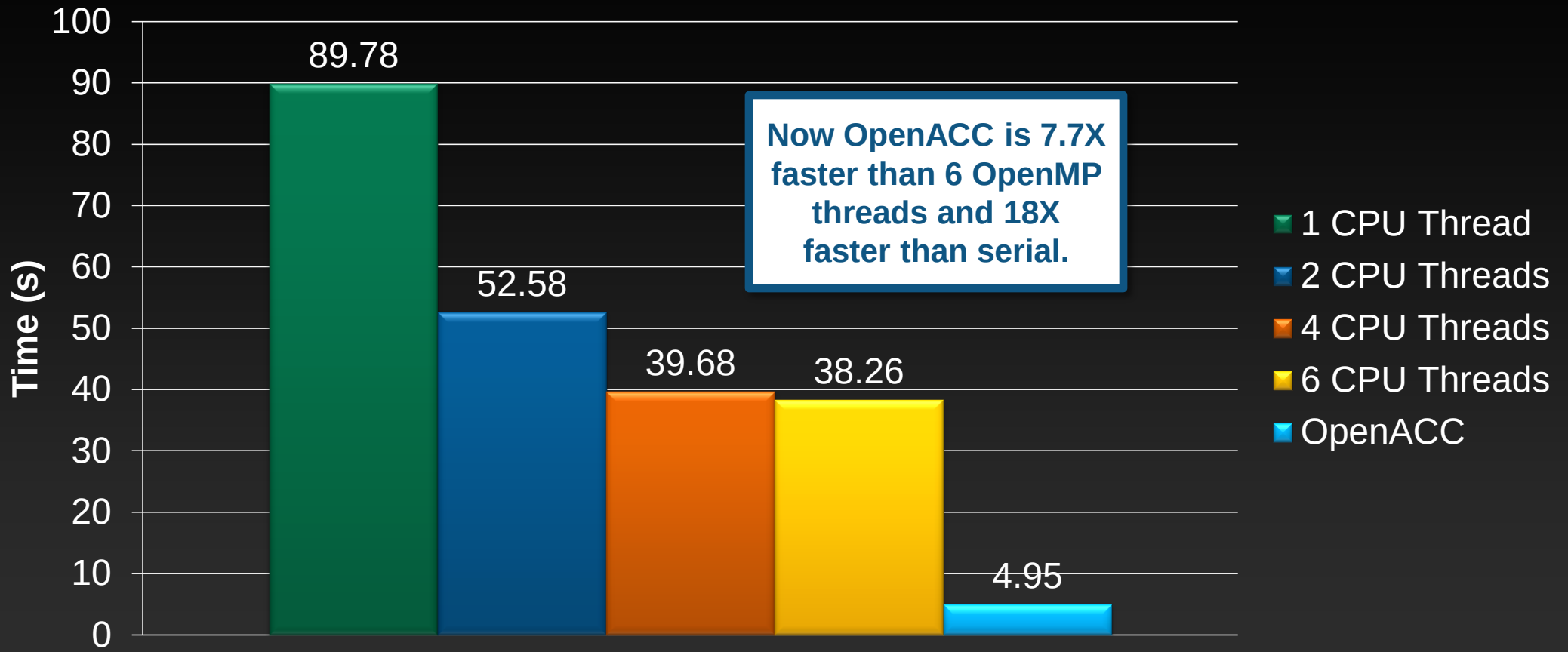
0.23 seconds

0.24 seconds

Execution Time (lower is better)

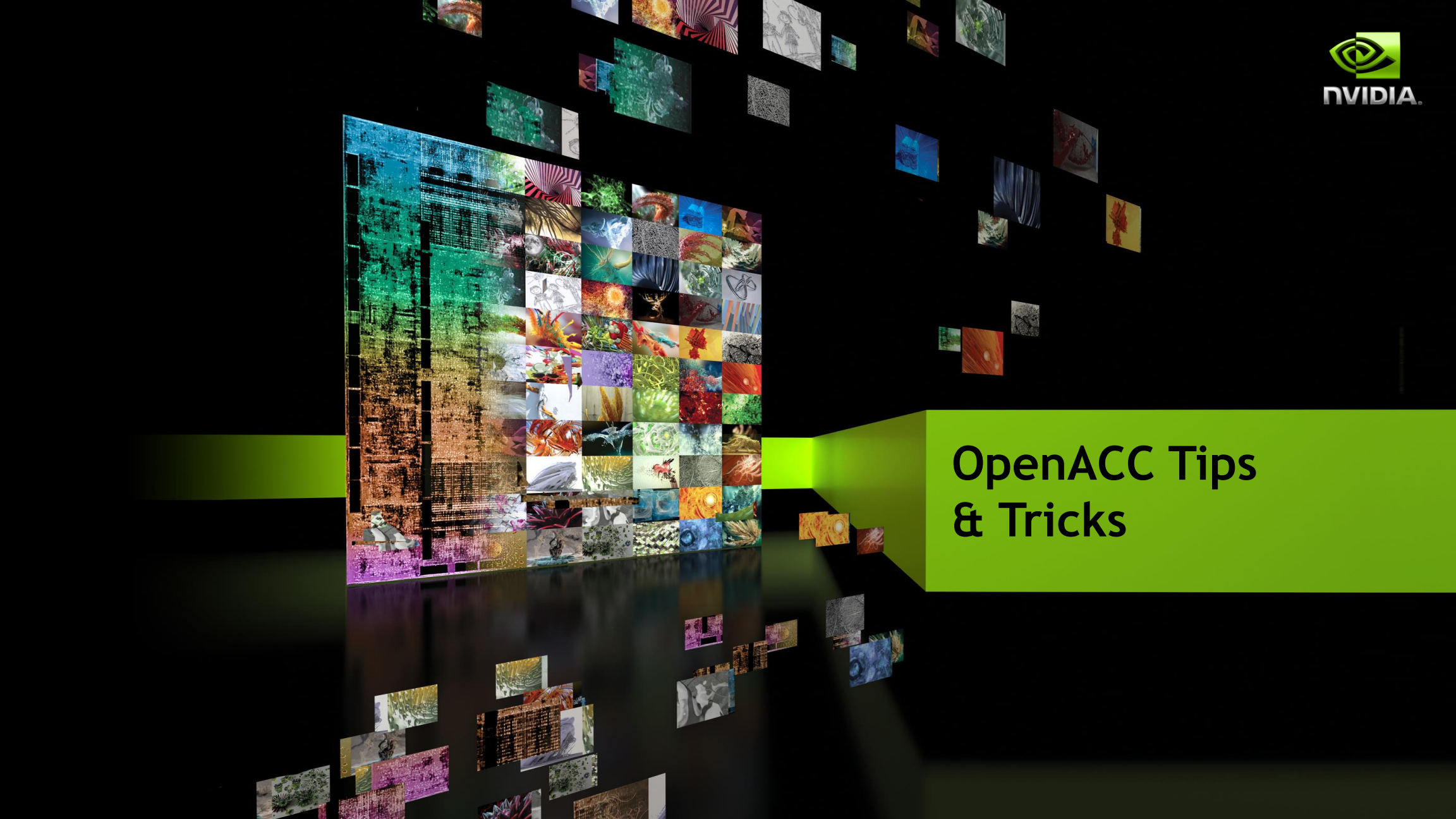
CPU: Intel i7-3930K
6 Cores @ 3.20GHz

GPU: NVIDIA Tesla K20



Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via `gang`, `worker`, and `vector` clauses
- By understanding more about the specific GPU on which you're running, using these clauses may allow better performance.
- By understanding bottlenecks in the code via profiling, we can reorganize the code for even better performance


NVIDIA
OpenACC Tips
& Tricks

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
- Compilers often require `restrict` to determine independence (true for OpenACC, OpenMP, and vectorization)
  - 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

Tips and Tricks

- Nested loops are best for parallelization
  - Large loop counts (1000s) needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: use `restrict` keyword in C
- Compiler must be able to figure out sizes of data regions
  - Can use directives to explicitly control sizes
- Inline function calls in directives regions
  - (PGI): `-Minline` or `-Minline=levels:<N>`
  - (Cray): `-hpl=<dir/>`
  - This has been improved in OpenACC 2.0

Tips and Tricks (cont.)

- Use time option to learn where time is being spent
  - (PGI) `PGI_ACC_TIME=1` (runtime environment variable)
  - (Cray) `CRAY_ACC_DEBUG=<1,2,3>` (runtime environment variable)
  - (CAPS) `HMPPRT_LOG_LEVEL=info` (runtime environment variable)
- Pointer arithmetic should be avoided if possible
  - Use subscripted arrays, rather than pointer-indexed arrays.
- Use contiguous memory for multi-dimensional arrays
- Use data regions to avoid excessive memory transfers
- Conditional compilation with `_OPENACC` macro


Thank you
NVIDIA