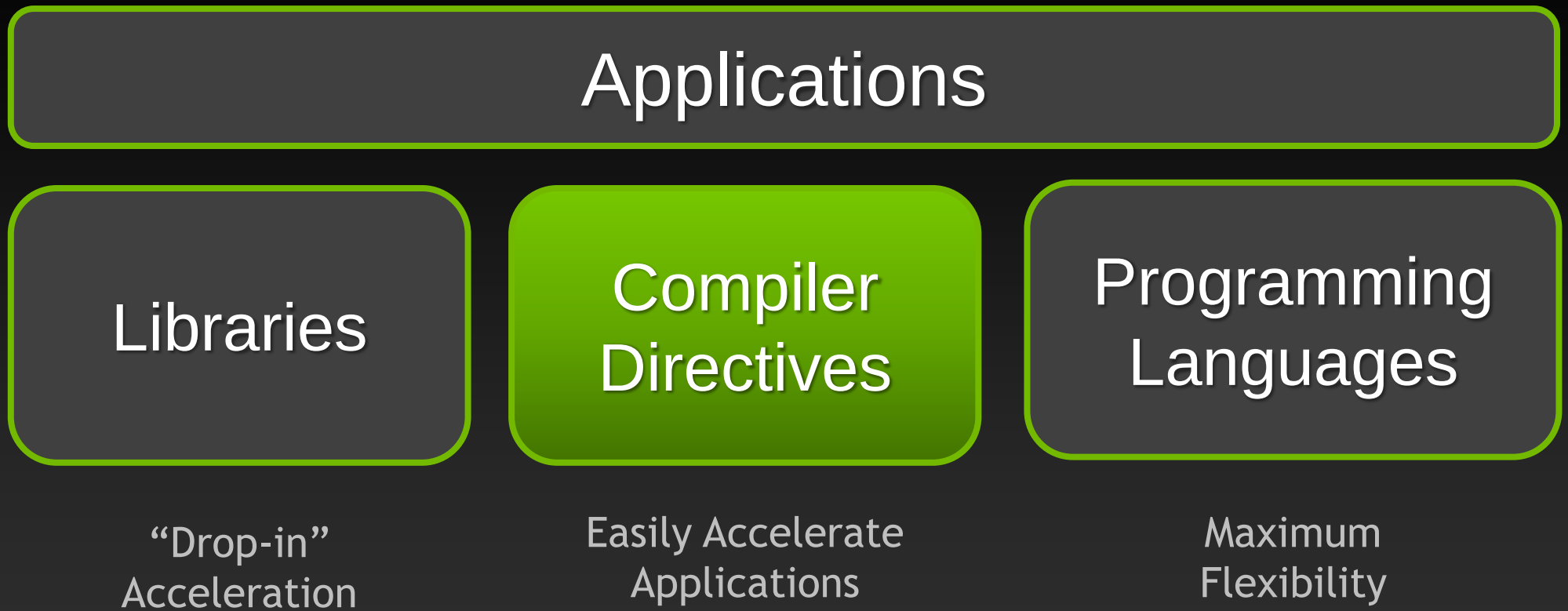
Introduction to OpenACC

Peng Wang
HPC Developer Technology, NVIDIA
penwang@nvidia.com

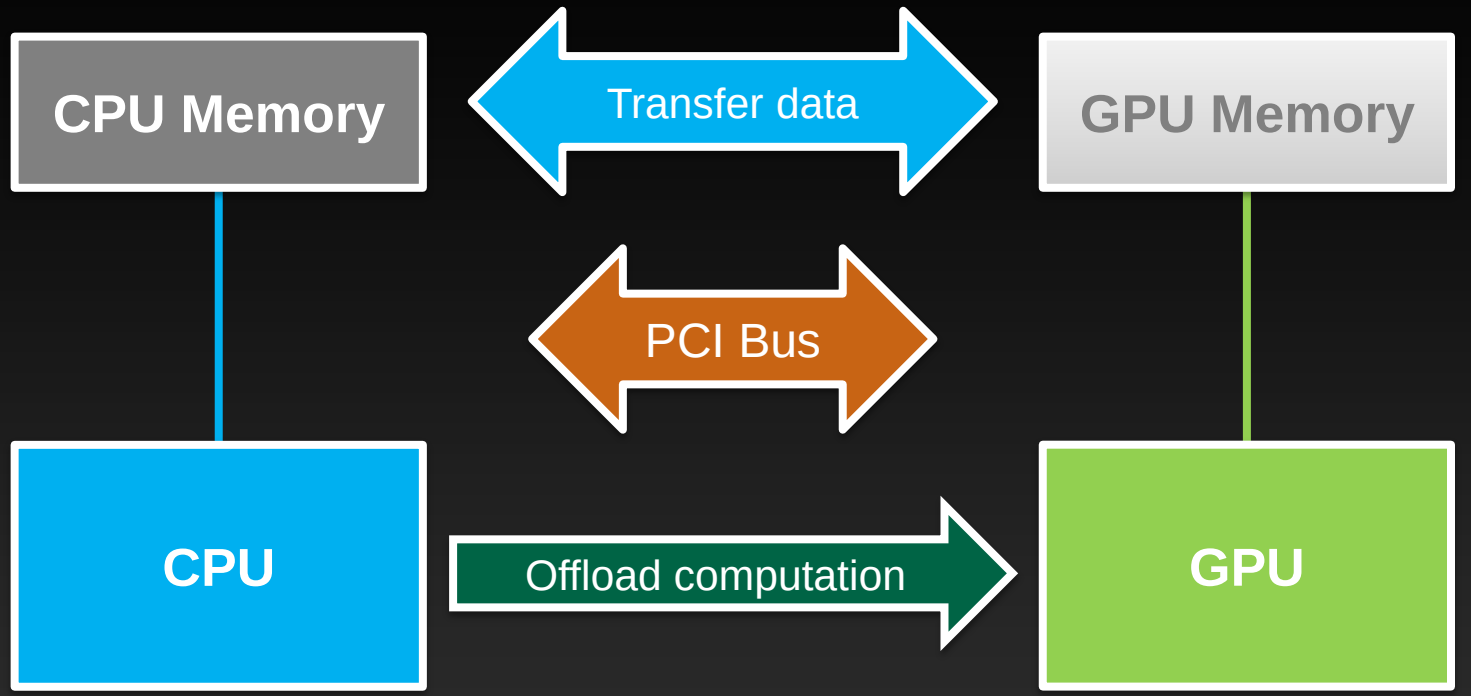
Outline

- Introduction of directive-based parallel programming
- Basic parallel construct
- Data management
- Controlling parallelism
- Interoperability
- Unified memory
- *“Directive”-based programming in CUDA*

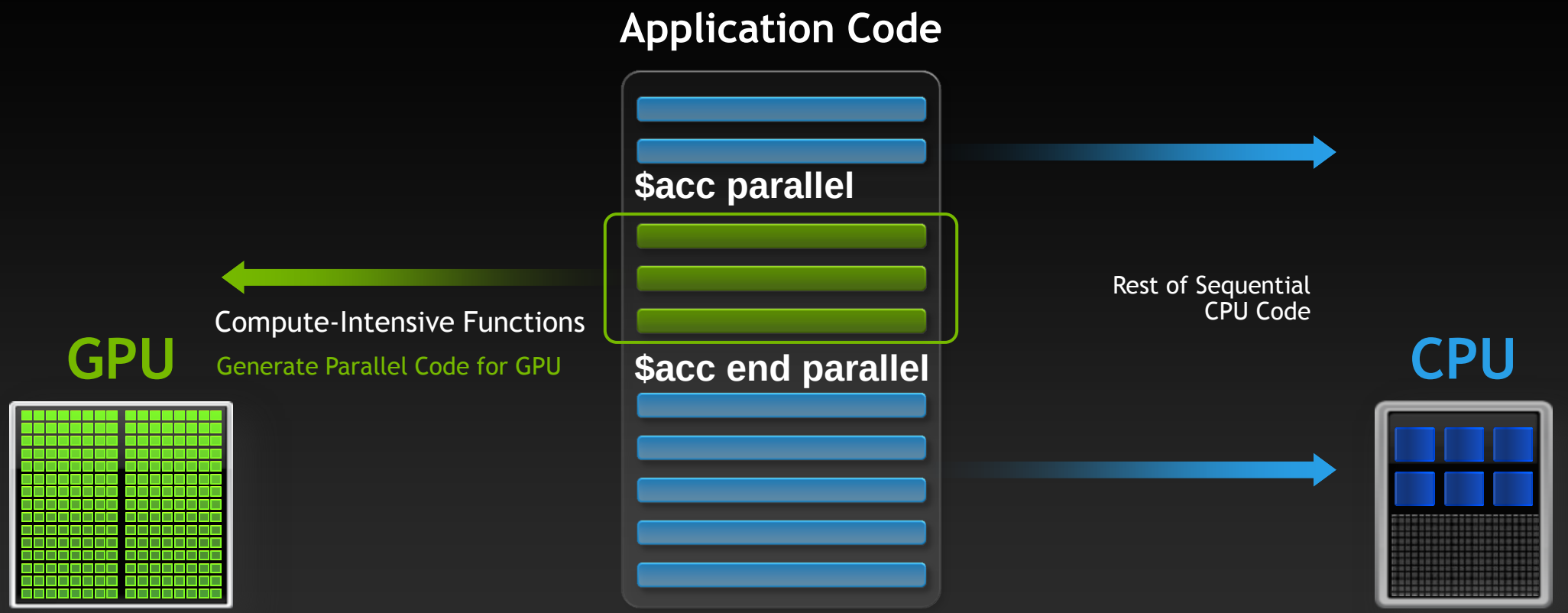
3 Ways to Accelerate Applications



GPU Computing: Basic Concepts



OpenACC Execution Model



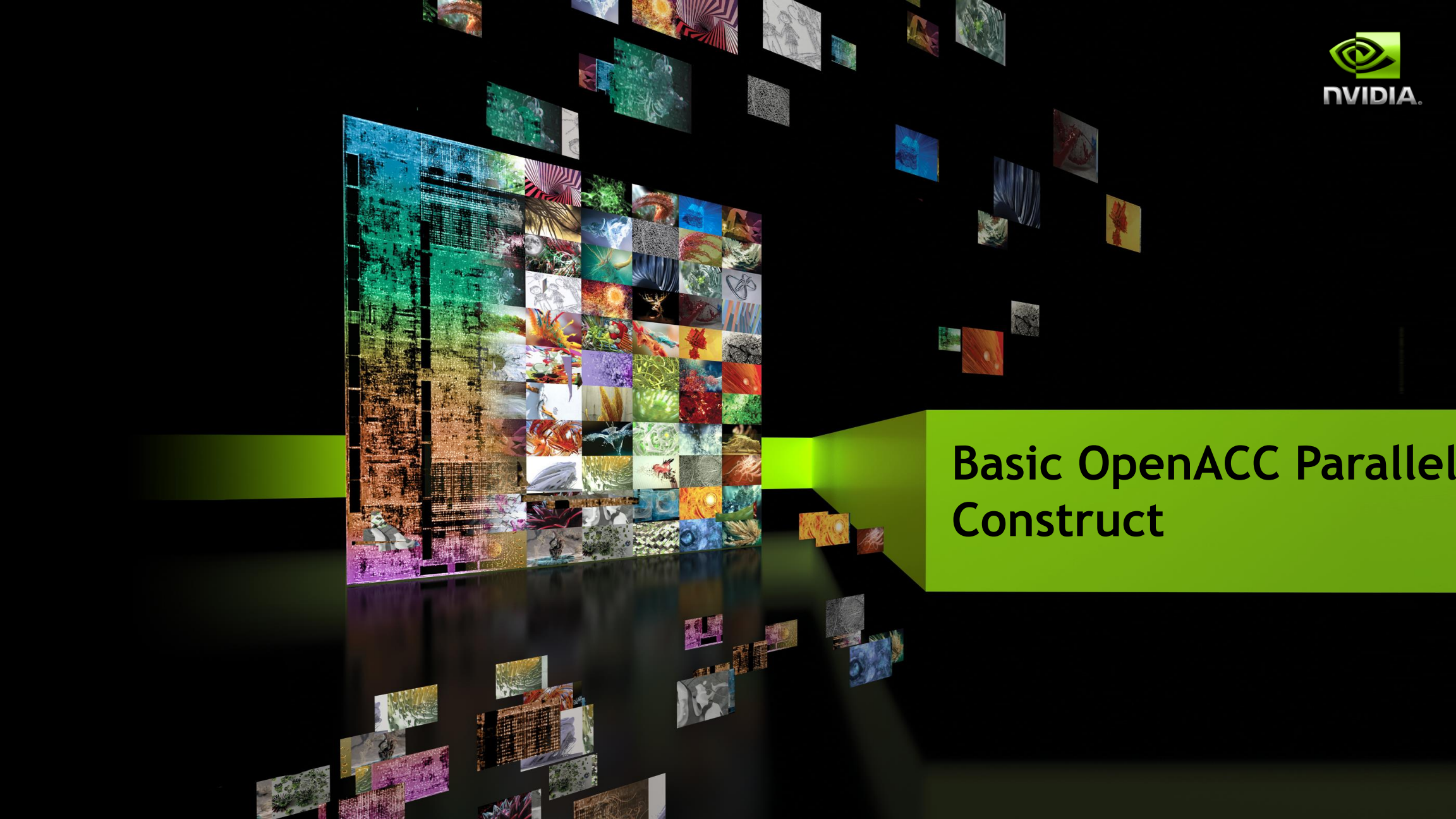
Directive-based Parallel Programming


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

- Suitable for manifestly parallel algorithms
- Portable across OSes, host CPUs, accelerators, and compilers
- Optimization no different from CUDA: unified host code doesn't mean unified hardware
  - With directives, tuning work focuses on exposing parallelism in the *CPU* code
    - E.g. enough loop parallelism, use SOA for better memory coalescing
    - Makes code inherently better for any platforms
  - Some optimizations techniques may not be available in OpenACC: explicit control of texture, constant, shared memory

You Should Learn All 3 Ways

- Which technique is most suitable for the given task?
- CUDA
  - Ensuring best performance
  - Tricky algorithm
- OpenACC
  - Code with manifestly massive parallelism
  - Quick prototype
  - Single source code for different archs
- Library
  - Quick prototype
  - Easy maintenance
  - Good performance
- Mixing all 3 ways



Basic OpenACC Parallel Construct

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)
```

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];
}

...
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

...
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];
}

...
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

...
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];
}

...
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

...
call saxpy(2**20, 2.0, x_d, y_d)
...
```

OpenACC parallel Directive

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

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

CUDA kernel

Kernel:
A parallel function that runs on the GPU

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

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`
- Always use "-Minfo=accel" and pay attention to the output!
- 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
```

Complete SAXPY example code

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

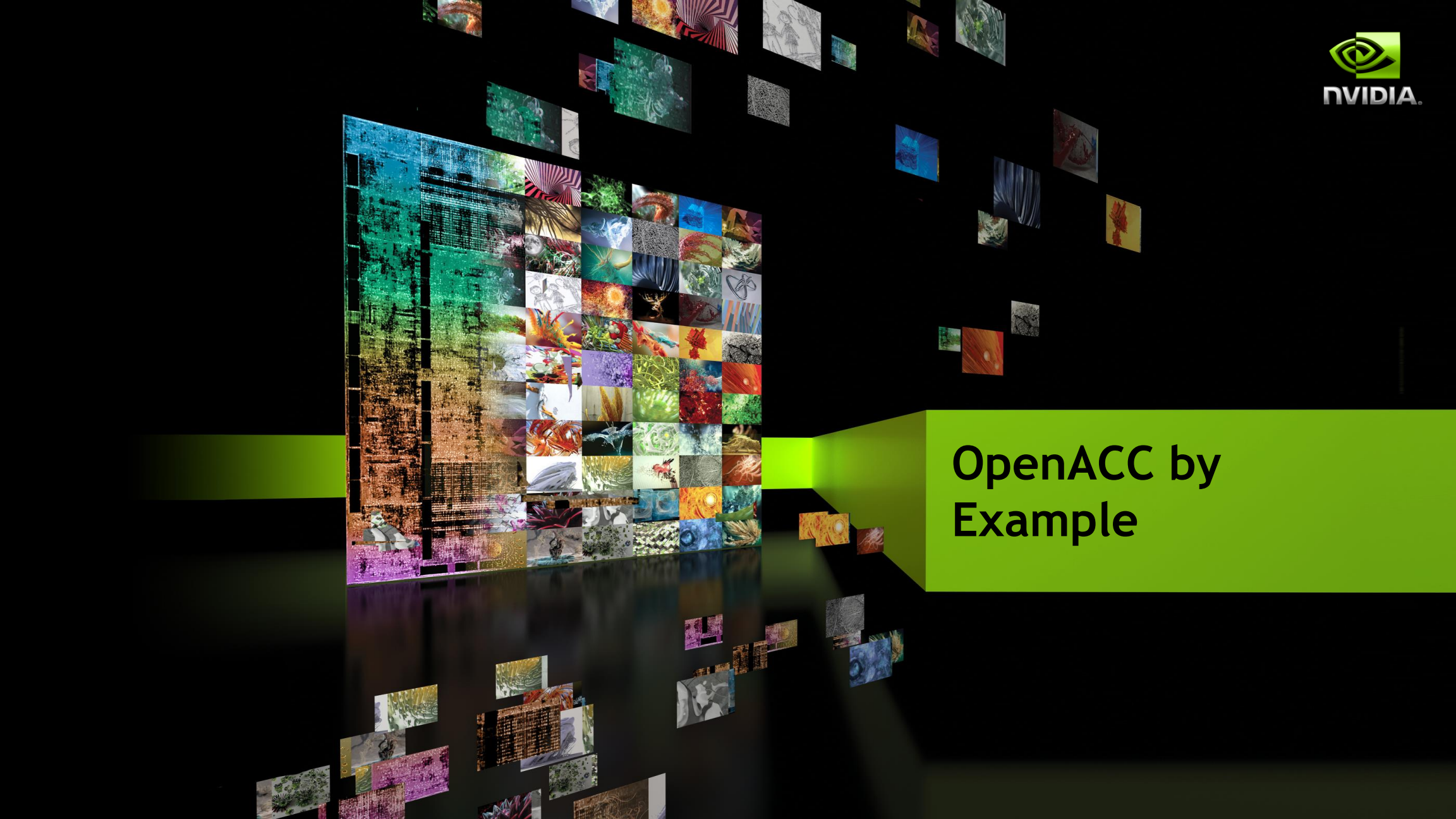
```
#include <stdlib.h>

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;
}
```

Concurrency

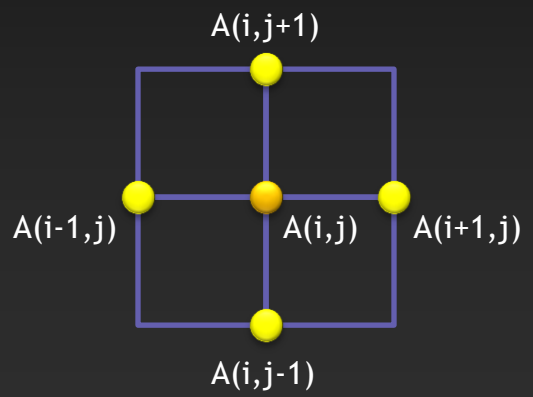
- OpenACC loops are blocking by default
- Use “async” & “wait” for asynchrounous kernel execution

```
#pragma acc parallel loop async(1)
for (int i = 0; i < n; i++)
    a[i] = i;

for (int i = 0; i < n; ++i)
    b[i] = i;

#pragma acc wait(1)
```


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=0.0f;

    for(int i = 1; i < nx-1; i++) {
      Anew[i] = 0.5f * (A[i+1] + A[i-1]);
      err = fmax(err, fabs(Anew[i] - A[i]));
    }

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

    iter++;
}
```

We will start with 1D. 2D will be discussed latter.

Jacobi Iteration: OpenMP C Code


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

    err=0.0f;

    #pragma omp parallel for shared(nx, Anew, A) reduction(max:err)
    for(int i = 1; i < nx-1; i++) {
      Anew[i] = 0.5f * (A[i+1] + A[i-1]);
      err = fmax(err, fabs(Anew[i] - A[i]));
    }

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

    iter++;
}
```

Jacobi Iteration: OpenACC C Code


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

    err=0.0f;

    #pragma acc parallel loop reduction(max:err)
    for(int i = 1; i < nx-1; i++) {
      Anew[i] = 0.5f * (A[i+1] + A[i-1]);
      err = fmax(err, fabs(Anew[i] - A[i]));
    }

    #pragma acc parallel loop
    for( int i = 1; i < nx-1; i++ ) {
      A[i] = Anew[i];
    }

    iter++;
}
```

PGI Accelerator Compiler output (C)


```
$ pgCC -acc -Minfo=accel -ta=nvidia,cc35 –O3 jacobi.cpp
    48, Accelerator kernel generated
        48, Max reduction generated for err
        50, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    48, Generating copyout(Anew[1:nx-2])
        Generating copyin(A[:nx])
        Generating Tesla code
    53, Accelerator kernel generated
        55, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
    53, Generating copyout(A[1:nx-2])
        Generating copyin(Anew[1:nx-2])
        Generating Tesla code
```

Performance

nx=512x512x512,
iter=100

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	42.5	1.0x
CPU 6 OpenMP threads	12.7	3.4x
OpenACC GPU	45.5	0.3x FAIL!

vs. 1 CPU core

vs. 6 CPU cores

CPU: Intel Core i7-3930K @ 3.2 GHz, "-mp –O3"

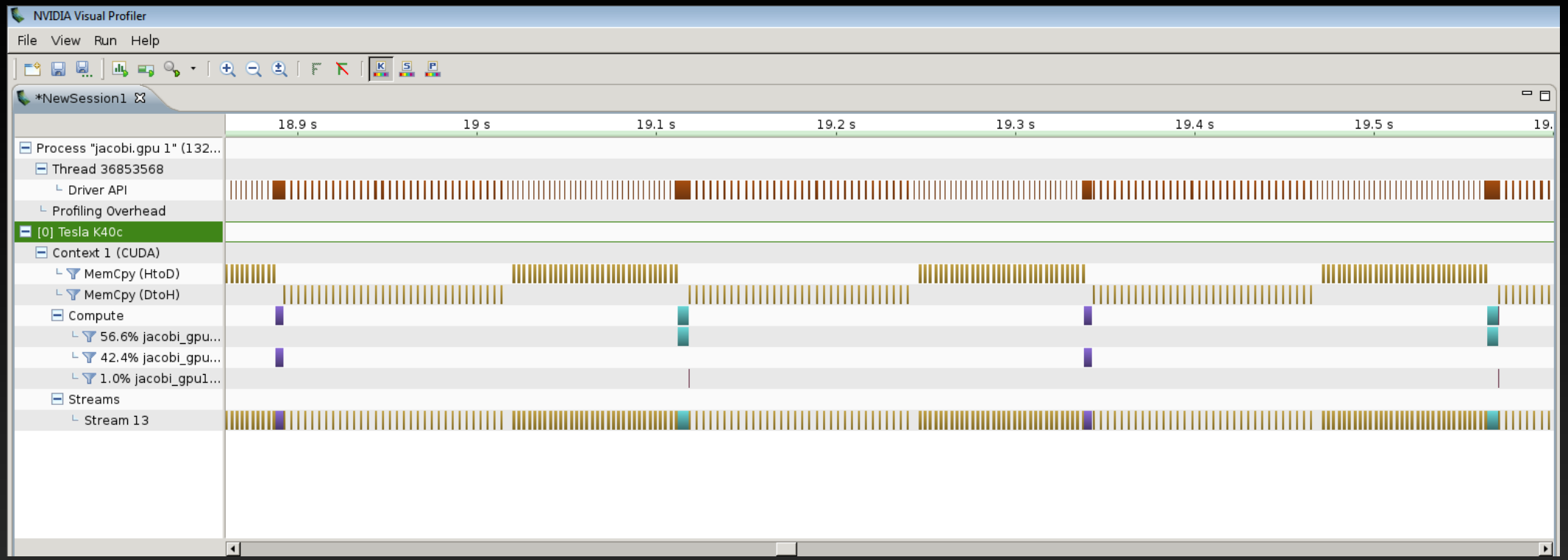
GPU: NVIDIA Tesla K40c @ 875 MHz, "**-acc -ta=nvidia,cc35 –O3**"

What went wrong

```
$ nvprof ./a.out
Time(%)      Time     Calls       Avg       Min       Max  Name
 48.45%  9.18953s      6500  1.4138ms     736ns  1.5423ms  [CUDA memcpy HtoD]
 45.79%  8.68473s      6700  1.2962ms  2.0480us  1.6118ms  [CUDA memcpy DtoH]
  3.26%  618.08ms       100  6.1808ms  6.1669ms  6.2061ms  jacobi_gpu1__FPfT1i_48_gpu
  2.44%  463.02ms       100  4.6302ms  4.6275ms  4.6327ms  jacobi_gpu1__FPfT1i_53_gpu
  0.06%  10.693ms       100  106.93us  106.63us  107.43us  jacobi_gpu1__FPfT1i_48_gpu_red
```

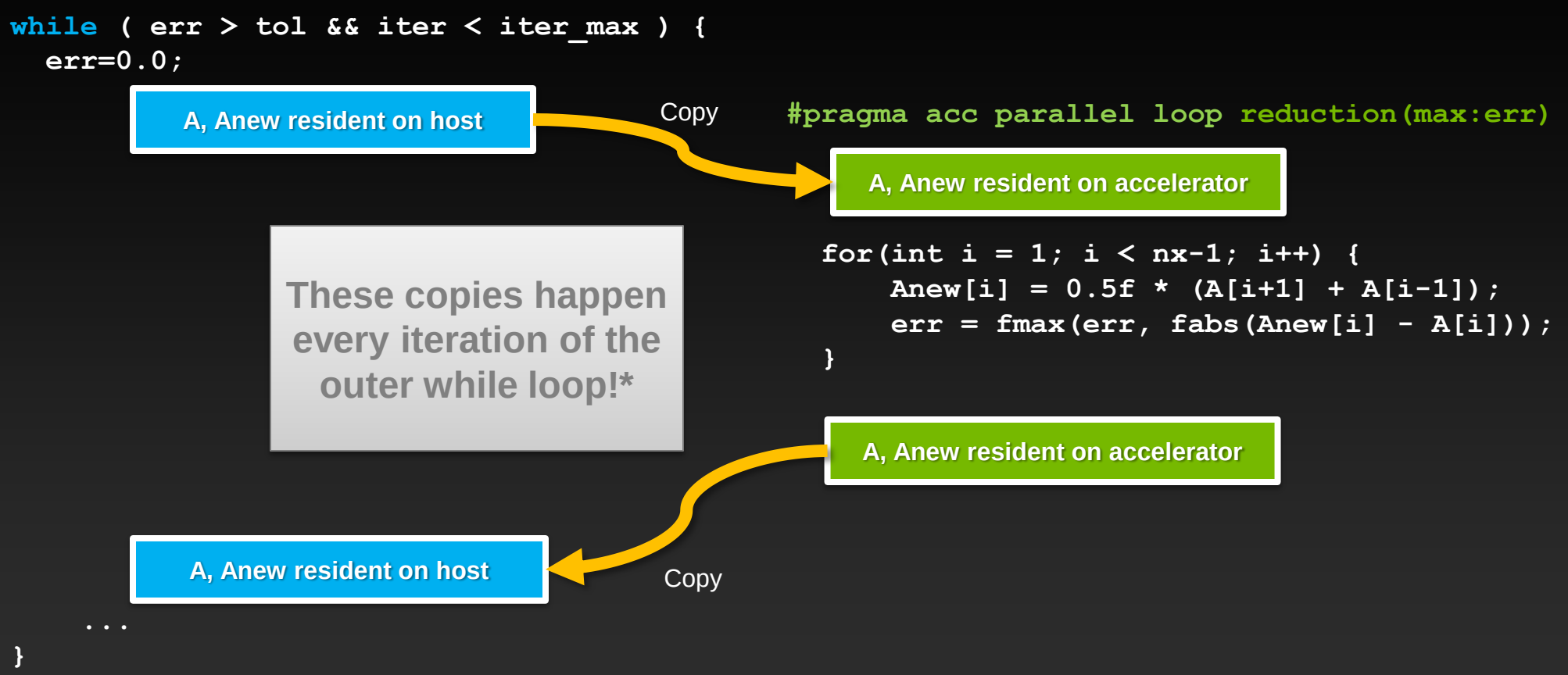
- ~94% of GPU time in CUDA memcpy
  - where is the memcpy
- CUDA memcpy time ~ 18 sec vs Elapsed time ~ 45 sec
  - why the difference?

NVIDIA Visual Profiler (nvvp)



- Where is the memcpy: between kernel calls

Excessive Data Transfers



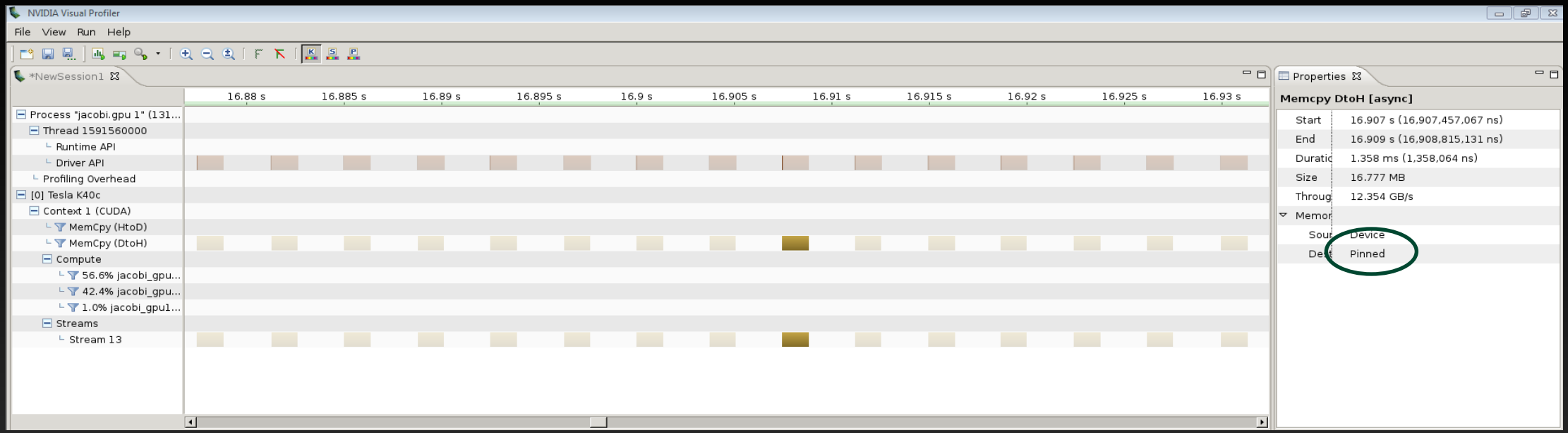
And note that there are two #pragma acc parallel, so there are 4 copies per while loop iteration!

Compiler: I Told You So

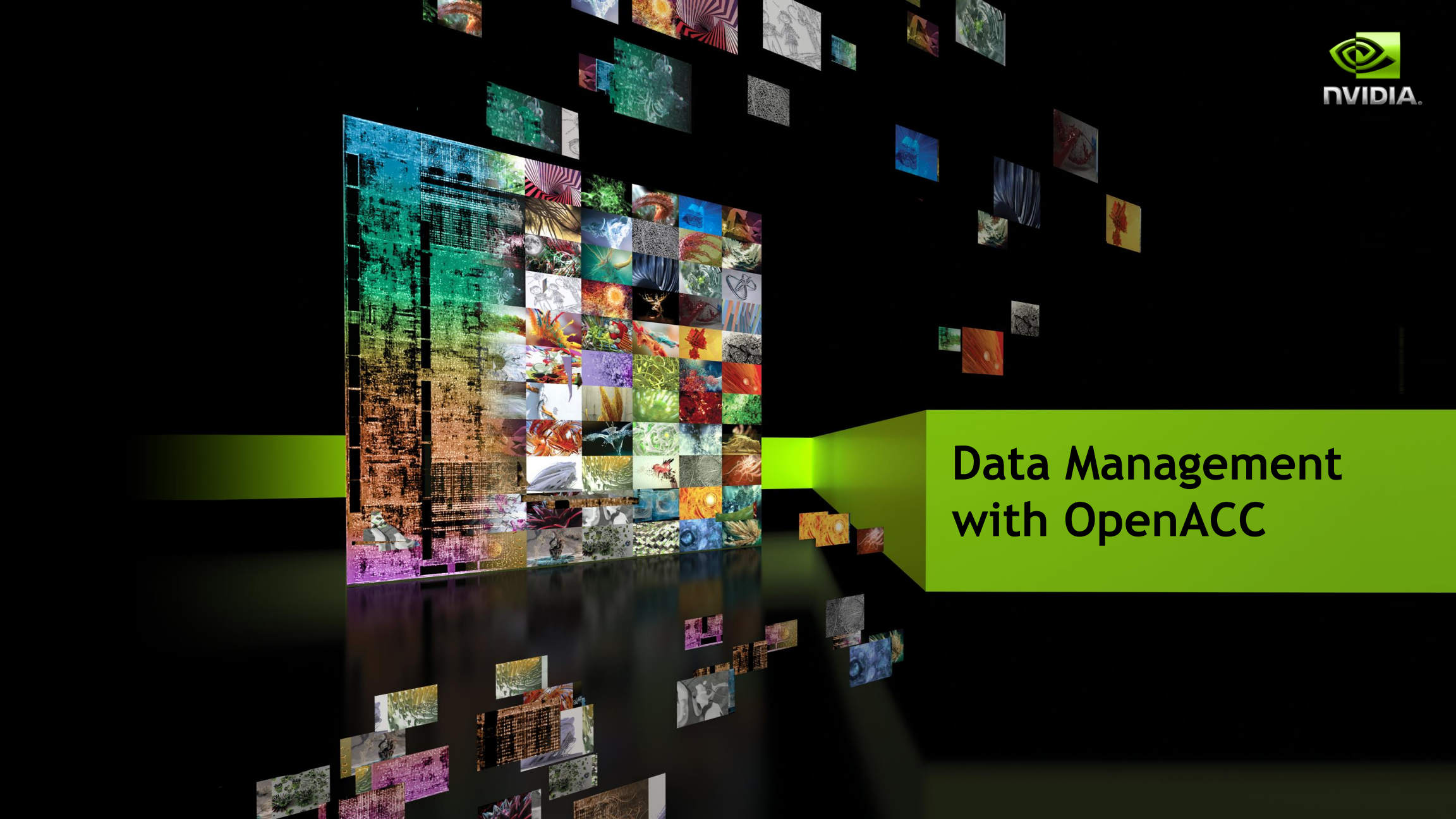

```
$ pgCC -acc -Minfo=accel -ta=nvidia,cc35 –O3 jacobi.cpp
     48, Accelerator kernel generated
         48, Max reduction generated for err
         50, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     48, Generating copyout(Anew[1:nx-2])
         Generating copyin(A[:nx])
         Generating Tesla code
     53, Accelerator kernel generated
         55, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     53, Generating copyout(A[1:nx-2])
         Generating copyin(Anew[1:nx-2])
         Generating Tesla code
```

Difference in GPU time

Zoom into memcpy calls



- CUDA memcpy is broken up into many small chunks
- Compiler is trying to stage the memcpy in pinned host buffer
  - The gap is probably copying from pinned buffer to user memory on CPU


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.

```
#pragma acc data \
   create(b[0:n]) copyout(a[0:n])
{
      for (i=0;i<n;i++) {
         a[i] = 0.0;
         b[i] = 1.0;
      }
      for (i=0;i<n;i++) {
         a[i] = a[i] + b[i];
      }
}
```

Data Region

Arrays a and b 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/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

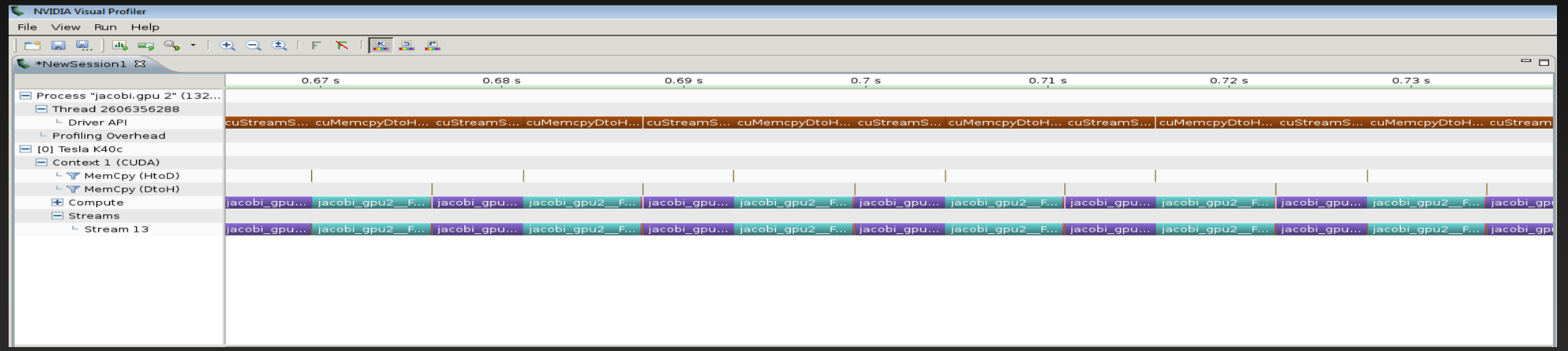
- Task: use `acc data` to minimize transfers in the Jacobi example

Jacobi Iteration: OpenACC C Code


```
#pragma acc data copyin(A[0:nx]) copyout(Anew[0:nx])
while ( err > tol && iter < iter_max ) {

    err=0.0f;

    #pragma acc parallel loop reduction(max:err)
    for(int i = 1; i < nx-1; i++) {
      Anew[i] = 0.5f * (A[i+1] + A[i-1]);
      err = fmax(err, fabs(Anew[i] - A[i]));
    }

    #pragma acc parallel loop
    for( int i = 1; i < nx-1; i++ ) {
      A[i] = Anew[i];
    }

    iter++;
}
```

Did it help?

```
$ nvprof ./a.out
Time(%)      Time     Calls       Avg       Min       Max  Name
 52.07%  647.62ms       100  6.4762ms  6.4607ms  6.5025ms  jacobi_gpu2__FPfT1i_69_gpu
 39.94%  496.72ms       100  4.9672ms  4.9624ms  4.9723ms  jacobi_gpu2__FPfT1i_74_gpu
  3.63%  45.199ms       132  342.41us     736ns  1.4128ms  [CUDA memcpy HtoD]
  3.50%  43.520ms       133  327.22us  1.7280us  1.3548ms  [CUDA memcpy DtoH]
  0.86%  10.700ms       100  107.00us  106.69us  107.52us  jacobi_gpu2__FPfT1i_69_gpu_red
```



- 18 sec -> 0.09 sec in CUDA memcpy
- No gaps between GPU calls

Double Check with Compiler

Previous version

```
48, Accelerator kernel generated
      48, Max reduction generated for err
      50, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
   48, Generating copyout(Anew[1:nx-2])
      Generating copyin(A[:nx])
      Generating Tesla code
   53, Accelerator kernel generated
      55, #pragma acc loop gang, vector(128) /* blockIdx.x
threadIdx.x */
   53, Generating copyout(A[1:nx-2])
      Generating copyin(Anew[1:nx-2])
      Generating Tesla code
```

New version

```
68, Generating copyin(A[:nx])
      Generating copyout(Anew[:nx])
   69, Accelerator kernel generated
      69, Max reduction generated for err
      71, #pragma acc loop gang, vector(128)
          /* blockIdx.x threadIdx.x */
   69, Generating Tesla code
   74, Accelerator kernel generated
      76, #pragma acc loop gang, vector(128)
       /* blockIdx.x threadIdx.x */
   74, Generating Tesla code
```

Performance

nx=512x512x512,
iter=100

Execution	Time (s)	Speedup	
CPU 1 OpenMP thread	42.5	--	
CPU 6 OpenMP threads	12.7	3.4x	**vs. 1 CPU core**
OpenACC GPU-V1	45.5	0.3x FAIL!	**vs. 6 CPU cores**
OpenACC GPU-V2	1.7	7.5X	**vs. 6 CPU cores**

CPU: Intel Core i7-3930K @ 3.2 GHz, “-mp –O3”

GPU: NVIDIA Tesla K40c @ 875 MHz, “**-acc -ta=nvidia,cc35 –O3**”

Unstructured Data Region

```
void foo(float* a, float* b, int n) {
  #pragma acc data create(b[0:n])\
     copyout(a[0:n])
  {
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
      a[i] = i;
      b[i] = i;
    }
    #pragma acc parrallel loop
    for (int i = 0; i < n; ++i)
      a[i] = a[i] + b[i];
  }
}

foo(a, b, n);
```

```
void foo(float* a, float* b, int n) {

  for (int i = 0; i < n; i++) {
    a[i] = i;
    b[i] = i;
  }
}
void bar(float *a, float *b, int n) {

  for (int i = 0; i < n; ++i)
    a[i] = a[i] + b[i];
}


foo(a, b, n);
bar(a, b, n);
```

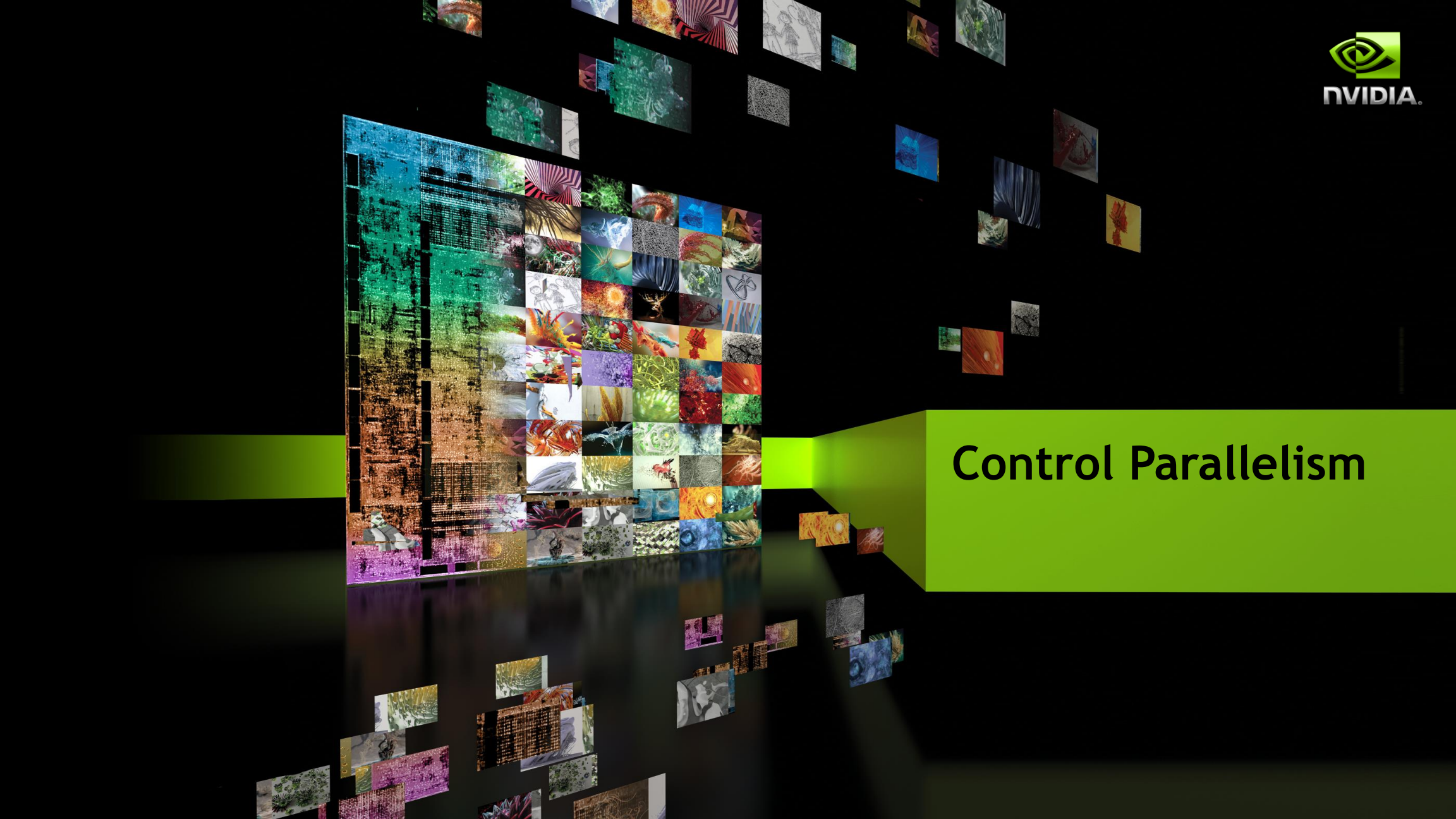
Use of data region is restricted to a single function.
What if the two loops are in two different functions?

Unstructured Data Region

```
void foo(float* a, float* b, int n) {
  #pragma acc data create(b[0:n])\
     copyout(a[0:n])
  {
    #pragma acc parallel loop
    for (int i = 0; i < n; i++) {
      a[i] = i;
      b[i] = i;
    }
    #pragma acc parrallel loop
    for (int i = 0; i < n; ++i)
      a[i] = a[i] + b[i];
  }
}

foo(a, b, n);
```

```
void foo(float* a, float* b, int n) {
  #pragma acc parallel loop
  for (int i = 0; i < n; i++) {
    a[i] = i;
    b[i] = i;
  }
}
void bar(float *a, float *b, int n) {
  #pragma acc parrallel loop
  for (int i = 0; i < n; ++i)
    a[i] = a[i] + b[i];
}

#pragma acc enter data \
     create(a[0:n], b[0:n])
foo(a, b, n);
bar(a, b, n);
#pragma acc exit data delete(b[0:n]) \
     copyout(a[0:n])
```

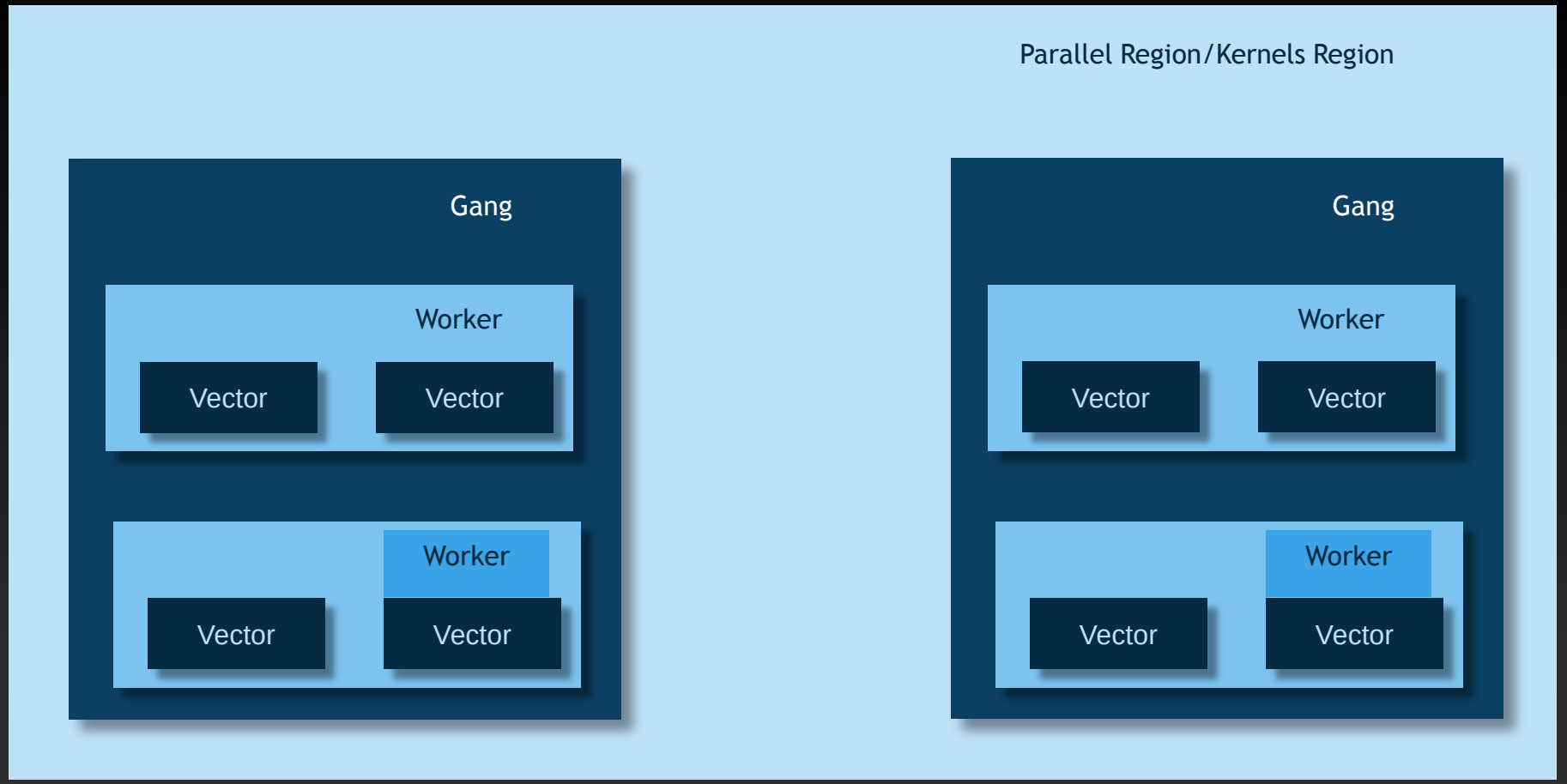
Use of data region is restricted to a single function.
What if the two loops are in two different functions?


NVIDIA
Control Parallelism

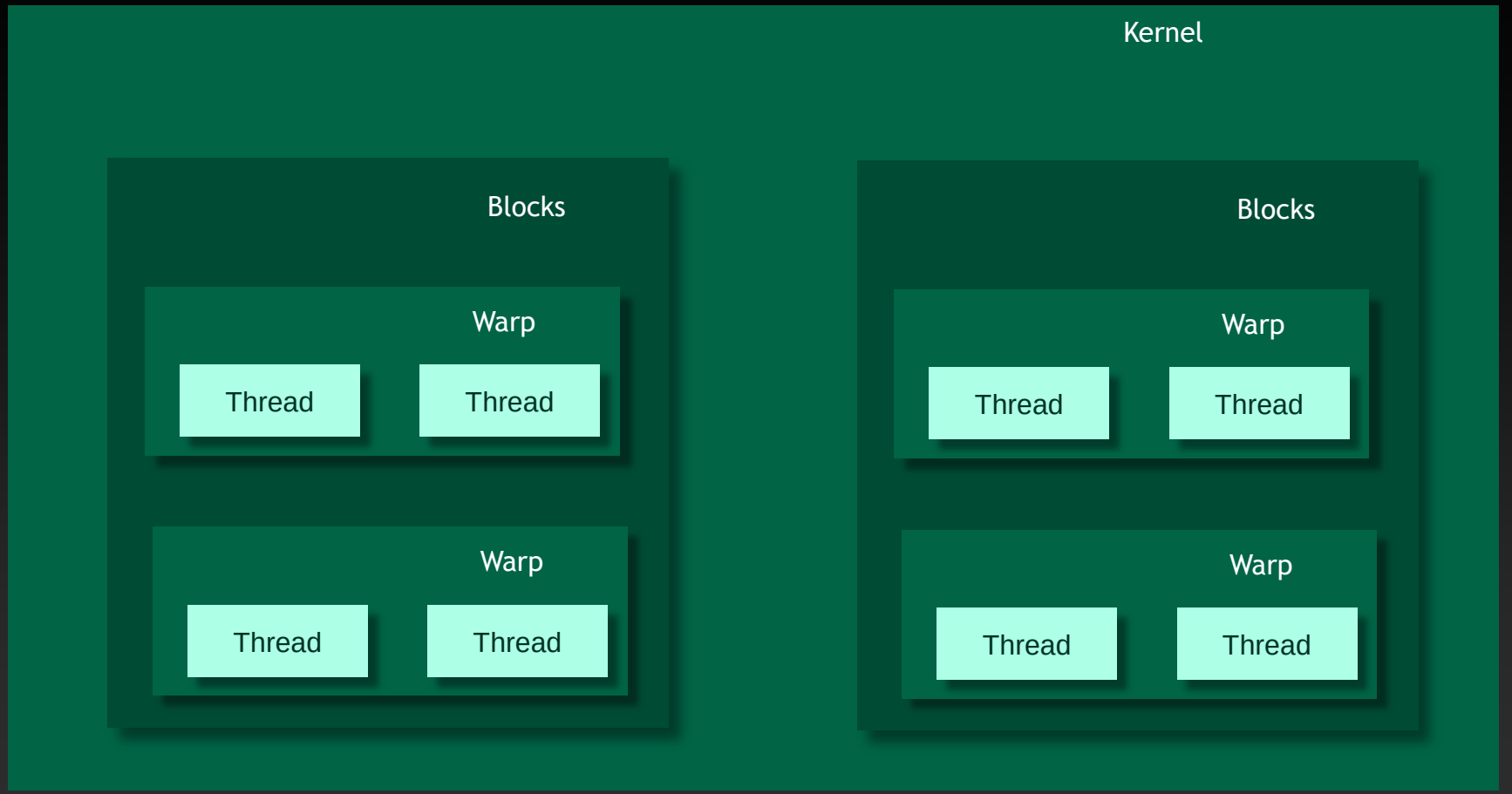
Controlling Parallelization Scheme

- By understanding more about OpenACC execution model and GPU hardware organization, we can get higher speedups on this code
- OpenACC gives us more detailed control over parallelization
  - Via `gang`, `worker`, and `vector` clauses

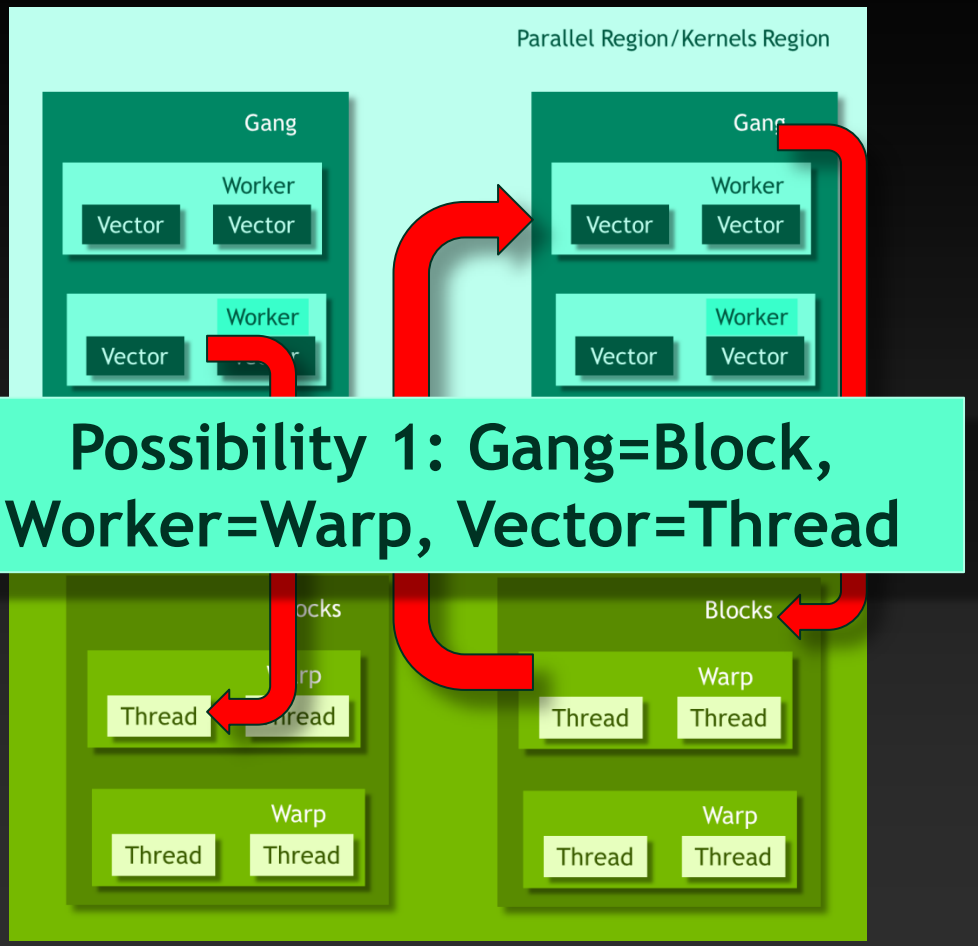
OpenACC's three levels of parallelism



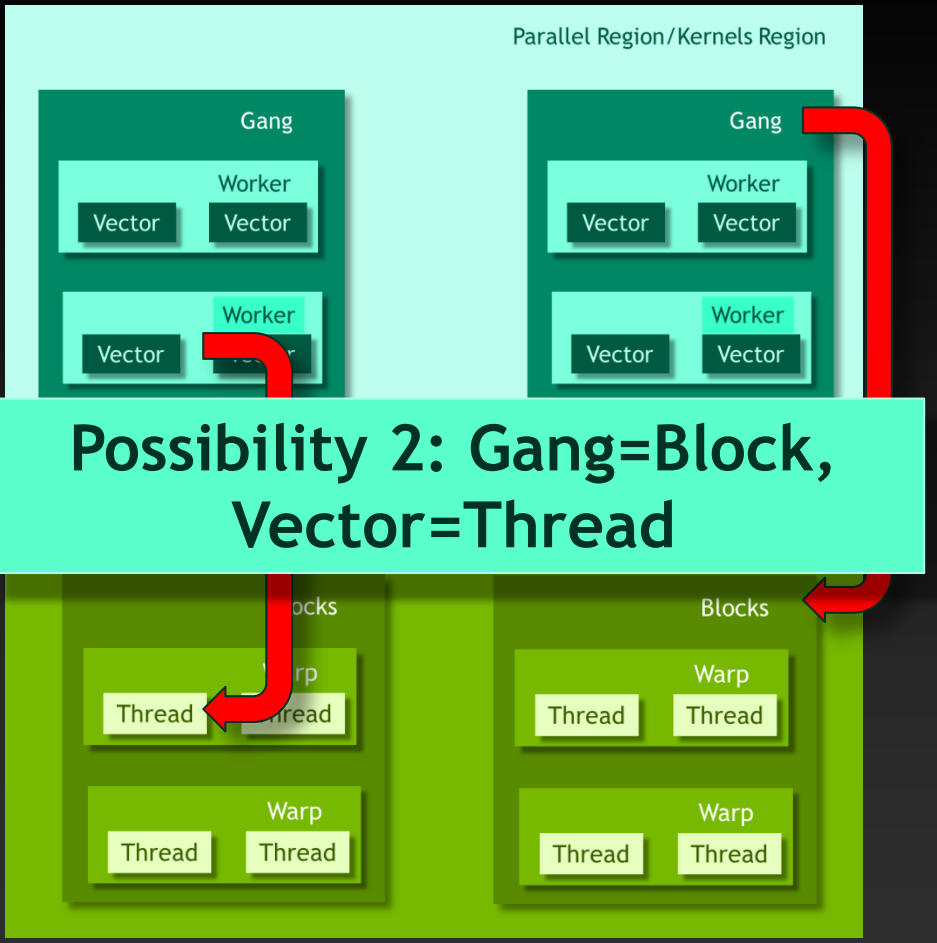
The CUDA execution model



Mapping is up to compiler!



Possibility 1: Gang=Block, Worker=Warp, Vector=Thread



Possibility 2: Gang=Block, Vector=Thread

Compiler shows the mapping


```
$ pgCC -acc -Minfo=accel -ta=nvidia,cc35 –O3 jacobi.cpp
     48, Accelerator kernel generated
         48, Max reduction generated for err
         50, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     48, Generating copyout(Anew[1:nx-2])
         Generating copyin(A[:nx])
         Generating Tesla code
     53, Accelerator kernel generated
         55, #pragma acc loop gang, vector(128) /* blockIdx.x threadIdx.x */
     53, Generating copyout(A[1:nx-2])
         Generating copyin(Anew[1:nx-2])
         Generating Tesla code
```

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
- Some clause requires parallel, e.g. reduction

KERNELS

- Compiler performs parallel analysis and parallelizes what it believes safe
- Can cover more than one loops with single directive

Example

```
void foo(float* a, float* b, int n) {
  #pragma acc kernels
  {
    for (int i = 0; i < n; i++) {
      a[i] = i;
      b[i] = i;
    }
    for (int i = 0; i < n; ++i)
      a[i] = a[i] + b[i];
  }
}
```

```
23, Complex loop carried dependence of 'a->' prevents parallelization
      Loop carried dependence of 'b->' prevents parallelization
      Loop carried backward dependence of 'b->' prevents vectorization
      Accelerator scalar kernel generated
28, Complex loop carried dependence of 'b->' prevents parallelization
      Loop carried dependence of 'a->' prevents parallelization
      Loop carried backward dependence of 'a->' prevents vectorization
      Accelerator scalar kernel generated
```

- Problem: compiler cannot decide it can safely parallelize the loops
  - What if b=a+1
- Solution: some way to tell compiler not to worry about this

Solutions

Solution 1

```
void foo(float* restrict a,
         float* restrict b,
         int n) {
  #pragma acc kernels
  {
    for (int i = 0; i < n; i++) {
      a[i] = i;
      b[i] = i;
    }
    for (int i = 0; i < n; ++i)
      a[i] = a[i] + b[i];
  }
}
```

Solution 2

```
void foo(float* a, float* b, int n) {
  #pragma acc kernels
  {
    #pragma acc loop independent
    for (int i = 0; i < n; i++) {
      a[i] = i;
      b[i] = i;
    }
    #pragma acc loop independent
    for (int i = 0; i < n; ++i)
      a[i] = a[i] + b[i];
  }
}
```

Solution 3: compiler option “-Mnodepchk”.

- Don’t check for data dependencies.
- May result in incorrect code. Not recommended.

Jacobi 2D

```
#pragma acc data copyin(A[0:nx*ny]) copyout(Anew[0:nx*ny])
  while ( err > TOL && iter < ITER_MAX ) {
    err=0.0f;
    #pragma acc parallel loop reduction(max:err)
    for (int j = 1; j < ny-1; j++) {
      #pragma acc loop independent
      for(int i = 1; i < nx-1; i++) {
        int idx = j*nx + i;
        Anew[idx] = 0.25f * (A[idx+1] + A[idx-1] +
                             A[idx+nx] + A[idx-nx]);
        err = fmax(fabs(Anew[idx]-A[idx]), err);
      }
    }
    #pragma acc parallel loop
    for (int j = 1; j < ny-1; j++) {
      #pragma acc loop independent
      for( int i = 1; i < nx-1; i++ ) {
        int idx = j*nx + i;
        A[idx] = Anew[idx];
      }
    }

    iter++;
  }
```

- Need copyin/copyout
  - “Accelerator restriction: size of the GPU copy of Anew,A is unknown”
- Need loop independent
  - “Complex loop carried dependence of A->,Anew-> prevents parallelization”
- Pay attention to the compiler info

Atomics

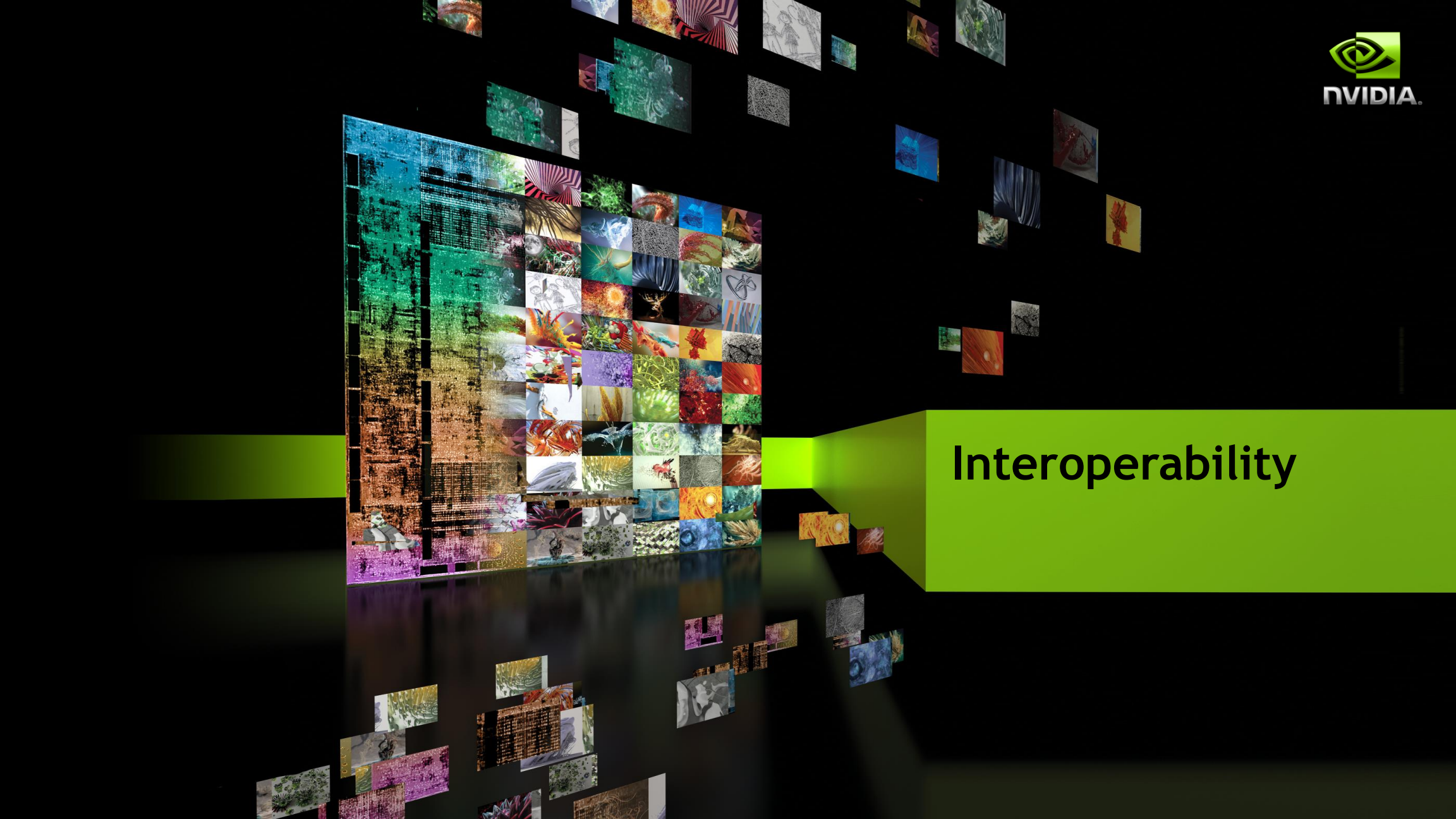
- Resolve write conflict

```
#pragma acc parallel loop
for (i=0;i<n;i++) {
  j = idx(i);
  #pragma acc atomic update
  a[j] = a[j]+b[i];
}
```

Routine

- Call routines within OpenACC loop

```
#pragma acc routine seq
void foo(float v) {
   v += 1;
}

#pragma acc parallel loop
for ( int i=0; i<n; ++i) {
  foo(v[i]);
}
```


NVIDIA
Interoperability

Interoperability

- OpenMP
  - “-acc -mp”
- CUDA

Mixing OpenACC and CUDA in C/C++

- Using OpenACC for CUDA managed data: device_ptr

```
float *a;
cudaMalloc(&a, n*sizeof(float));
ker<<<grid,block>>>(a);
#pragma acc parallel loop device_ptr(a)
for (i=0; i < n; i++)
  a[i] += 1.0f;
```

Mixing OpenACC and CUDA

- Using CUDA for OpenACC managed data: host_data

```
__global__ vec_scale_ker(float *a)
{
  int i = blockIdx.x*blockDim.x+threadIdx.x;
  a[i] *= 2;
}
void vec_scale(float *a,int n)
{
  vec_scale_ker<<<n/128,128>>>(a,n);
}
```

```
#pragma acc data copy(a[0:n-1]) copyin(b[0:n-1])
{
#pragma acc parallel loop
for (i=0;i<n;i++)
  a[i] = a[i]+b[i];
#pragma acc host_data use_device(a)
{
vec_scale(a);
}
}
```

If vec_scale is a library call, e.g. CUBLAS, it works in the same way.

Mixing OpenACC and CPU code

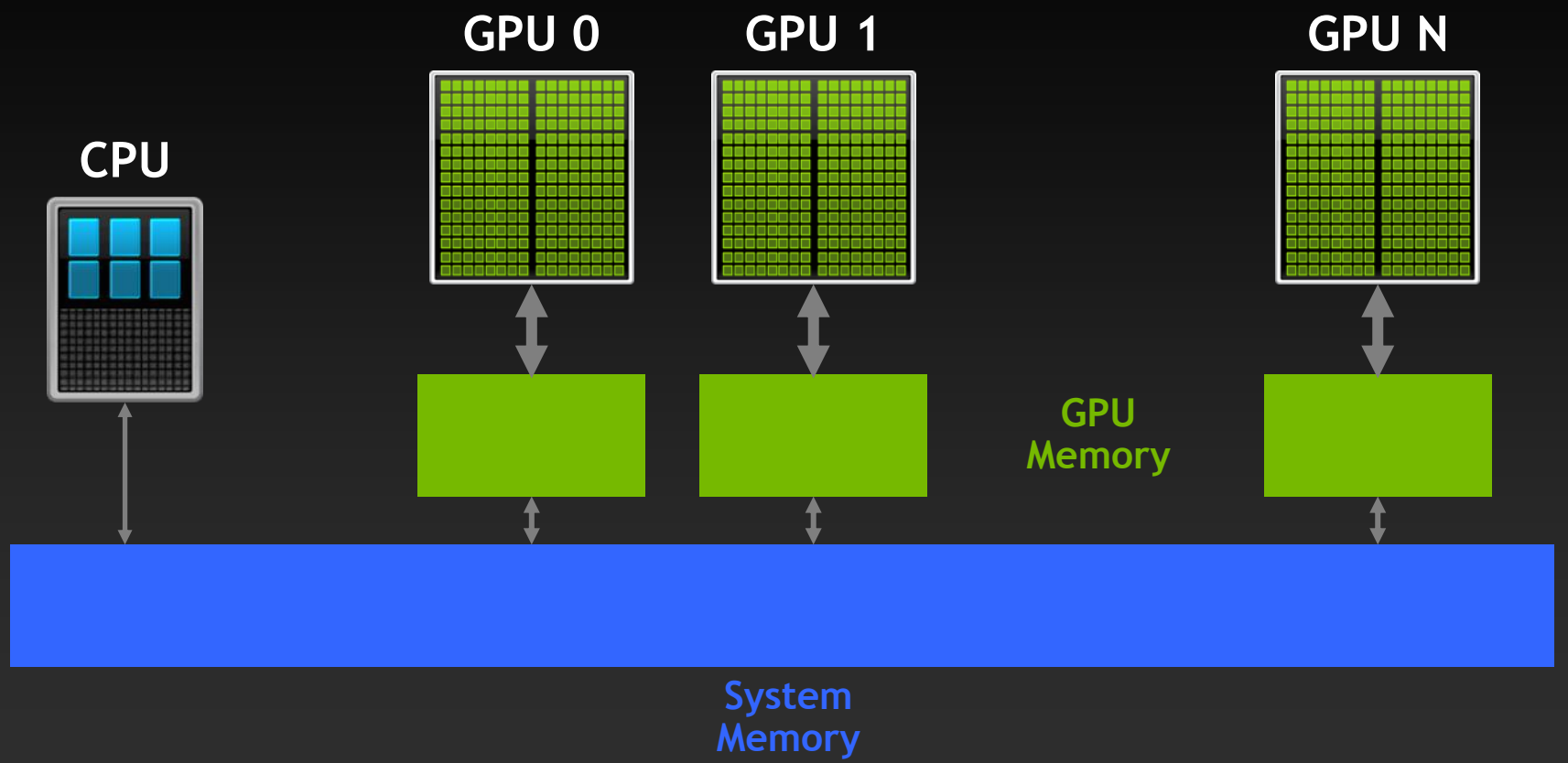
- Passing data explicitly between GPU and CPU: update

```
#pragma acc data copy(a[0:n-1]) copyin(b[0:n-1])
{
#pragma acc parallel loop
for (i=0;i<n;i++)
  a[i] = a[i]+b[i];
#pragma update host(a[0:n-1])
for (i=0;i<n;i++)
  a[i] -= 2;
#pragma update device(a[0:n-1])
…
}
```


NVIDIA
Unified memory

Heterogeneous architectures

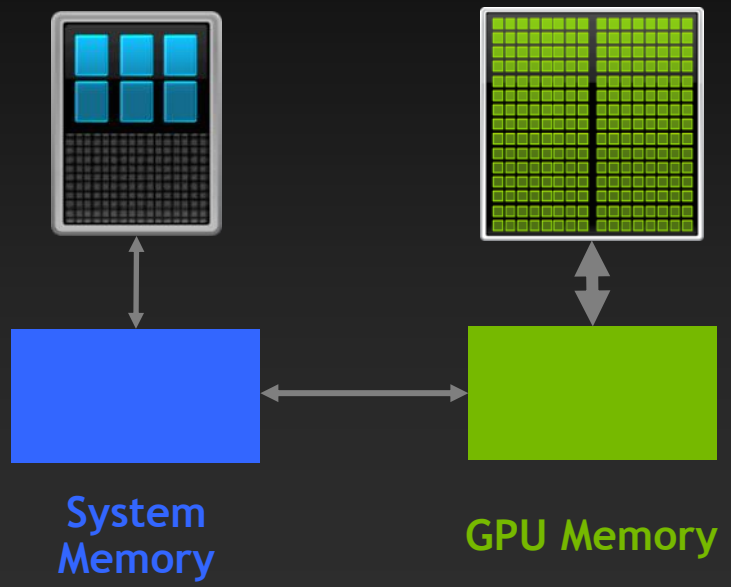
Memory hierarchy



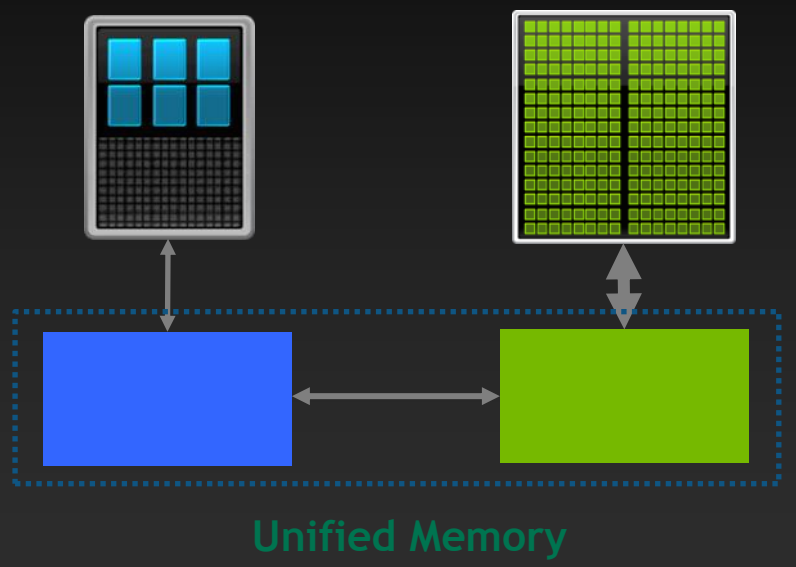
Unified Memory

Starting with Kepler and CUDA 6

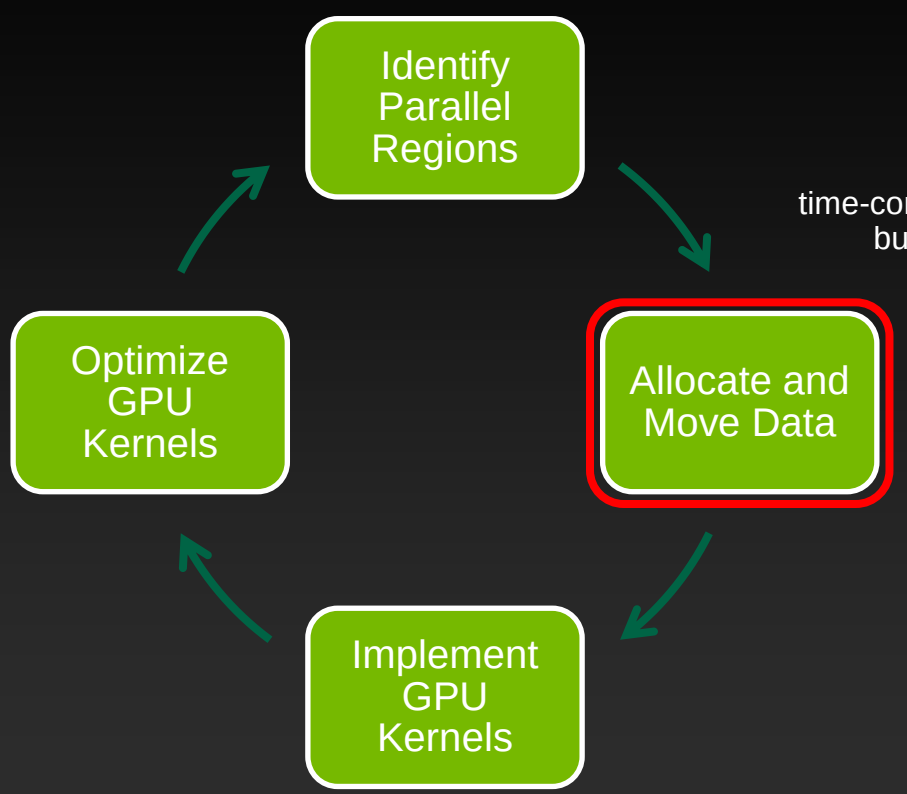
Custom Data Management



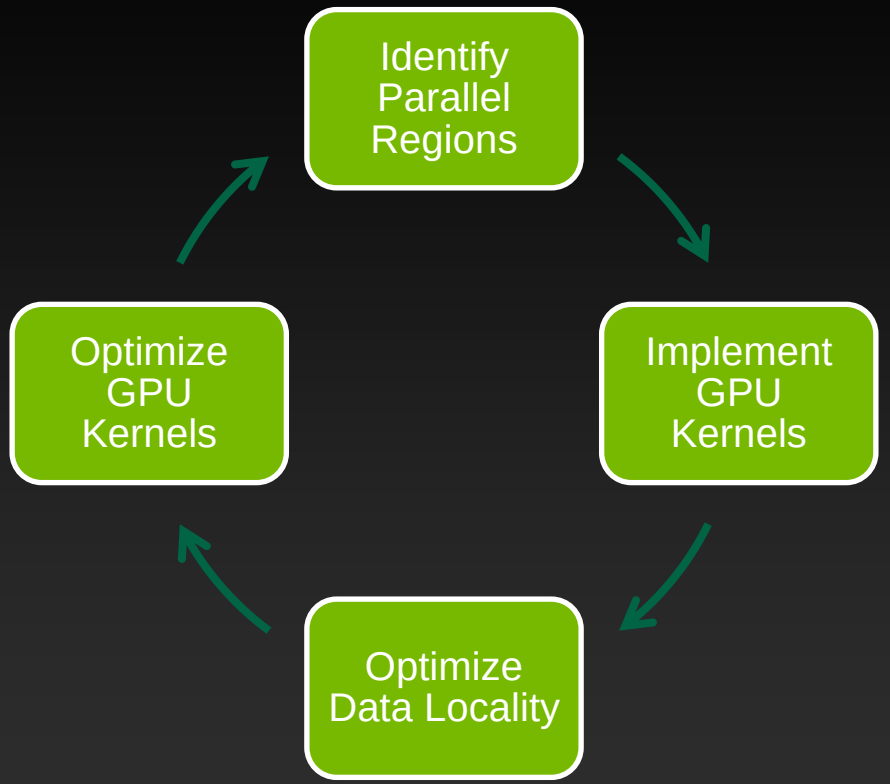
Developer View With Unified Memory



Unified memory improves dev cycle



Often time-consuming and bug-prone



W/o UM

```
module.F90:
module particle_array
real,dimension(:,:),allocatable :: zelectron
!$acc declare create(zelectron)

setup.F90:
allocate(zelectron(6,me))
call init(zelectron)
!$acc update device(zelectron)

pushe.F90:
real mpgc(4,me)
!$acc declare create(mpgc)
!$acc parallel loop
do i=1,me
   zelectron(1,m)=mpgc(1,m)
enddo
!$acc end parallel

!$acc update host(zelectron)
call foo(zelectron)
```

W/ UM

```
module.F90:
module particle_array
real,dimension(:,:),allocatable :: zelectron

setup.F90:
allocate(zelectron(6,me))
call init(zelectron)

pushe.F90:
real mpgc(4,me)
!$acc parallel loop
do i=1,me
   zelectron(1,me)=...
enddo
!$acc end parallel
call foo(zelectron)
```

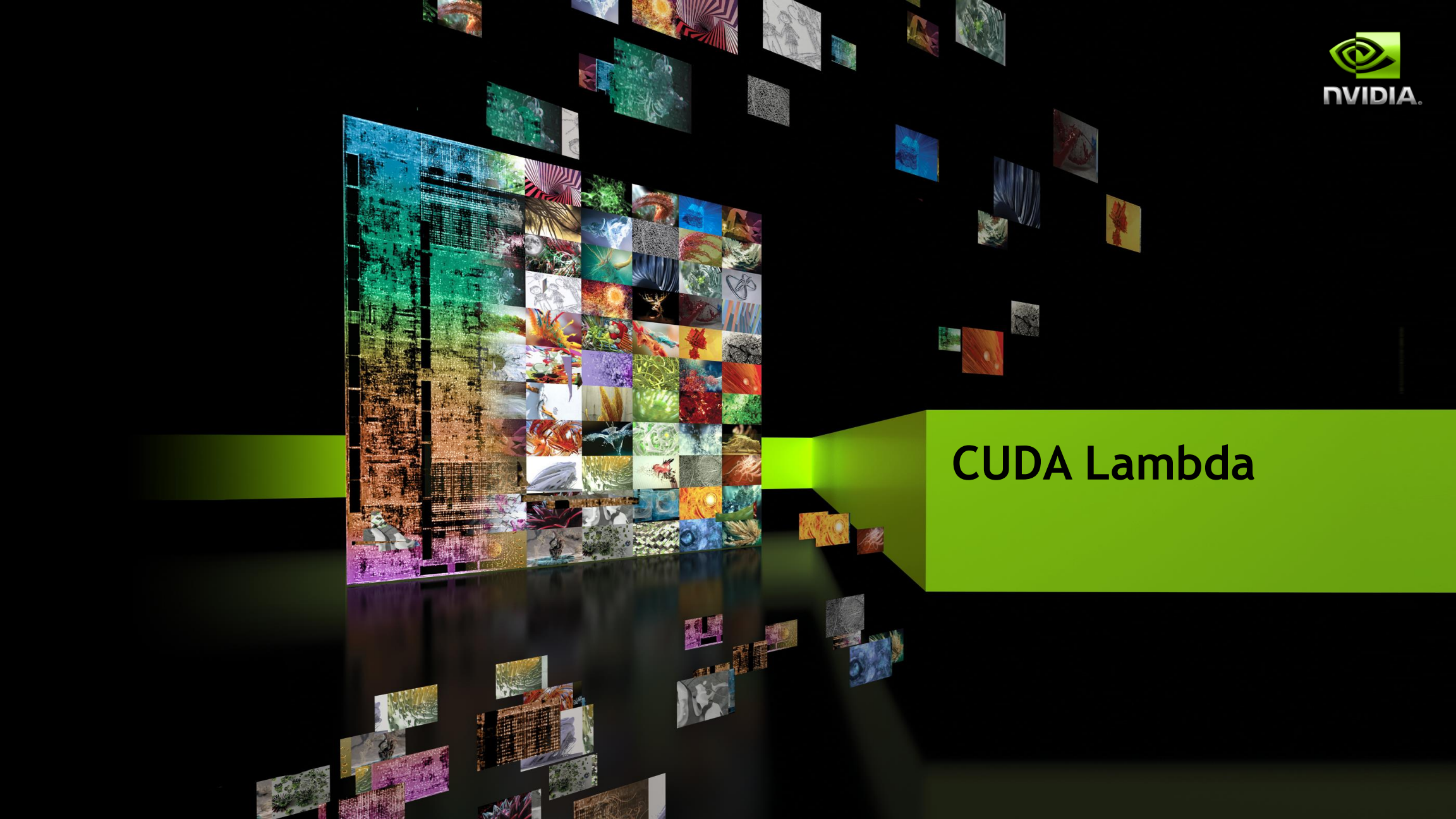
- Remove all data directives. Add "-ta=managed" to compile
- Only works for dynamically allocated memory on Pascal/Maxwell/Kepler

Summary

- Parallel loop
  - parallel loop, kernels
    - gang, worker, vector, reduction
  - routine
  - atomics
- Data:
  - data: copyin, copyout, create, present
  - enter/exit data: unstructured data region
  - host_data/dev_ptr: sharing data between CUDA and OpenACC
  - update: CPU-GPU

References

- OpenACC tutorials in GTC
  - http://www.gputechconf.com/gtcnew/on-demand-gtc.php
- http://www.openacc-standard.org/
- PGI document
  - http://www.pgroup.com/resources/accel.htm
- Forum
  - http://www.pgroup.com/userforum/viewforum.php?f=12&sid=12f0eb27e65b00cf7ea87d5d85600474


NVIDIA
CUDA Lambda

Lambda

- C++11 feature, CUDA 7.5+
- Concise syntax for defining anonymous functions
- Write CUDA in “directive-like” way
  - No need to define kernel for every loop
  - Unified CPU and GPU loops

```
struct saxpy_functor : public thrust::binary_function<float, float, float>
{
  float a;
  saxpy_functor(float a) : a(a) {}
  __host__ __device__
  float operator()(float x, float y)
  {
    return a * x + y;
  }
};

thrust::transform(X.begin(), X.end(),  Y.begin(), Y.begin(),
      saxpy_functor(a));
```

```
thrust::transform(X.begin(), X.end(),  Y.begin(), Y.begin(),
                  [=] __device__ (float x, float y) { return a * x + y; });
```

No need to define explicit kernel

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

C++

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

vec_add<<<(n+127)/128, 128>>>(d_x, d_y, d_z, a, n);
```

CUDA

```
for_each(counting_iterator<int>(0), counting_iterator<int>(n), [=] __device__ (int i) {
    d_z[i] = a * d_x[i] + d_y[i];
  });
```

CUDA+Lambda

Unified cpu and gpu loops

With UM, pointer is unified

```
#ifdef USE_GPU
for_each(counting_iterator<int>(0), counting_iterator<int>(n), [=] __device__ (int i) {
#else
for (int i = 0; i < n; i++) {
#endif
      z[i] = a * x[i] + y[i];
    });
```

UM+Lambda

CUDA 7.5, on today's Kepler GPU

nvcc -arch=sm_35 lambda.cu -std=c++11 --expt-extended-lambda

```
#include <stdio.h>
#include <thrust/iterator/counting_iterator.h>
#include <thrust/for_each.h>

using namespace thrust;

int main(void)
{
  float a = 2.0f;
  const int n = 4;
  float *x, *y, *z;
  cudaMallocManaged(&x, n*sizeof(float));
  cudaMallocManaged(&y, n*sizeof(float));
  cudaMallocManaged(&z, n*sizeof(float));

  for (int i = 0; i < n; i++) {
    x[i] = i;
    y[i] = 1;
  }

  for_each(counting_iterator<int>(0), counting_iterator<int>(n), [=] __device__ (int i) {
      z[i] = a * x[i] + y[i];
    });
  cudaDeviceSynchronize();

  for (int i = 0; i < n; i++)
    printf("%g*%g+%g=%g\n", a, x[i], y[i], z[i]);

  cudaFree(x);
  cudaFree(y);
  cudaFree(z);
  return 0;
}
```