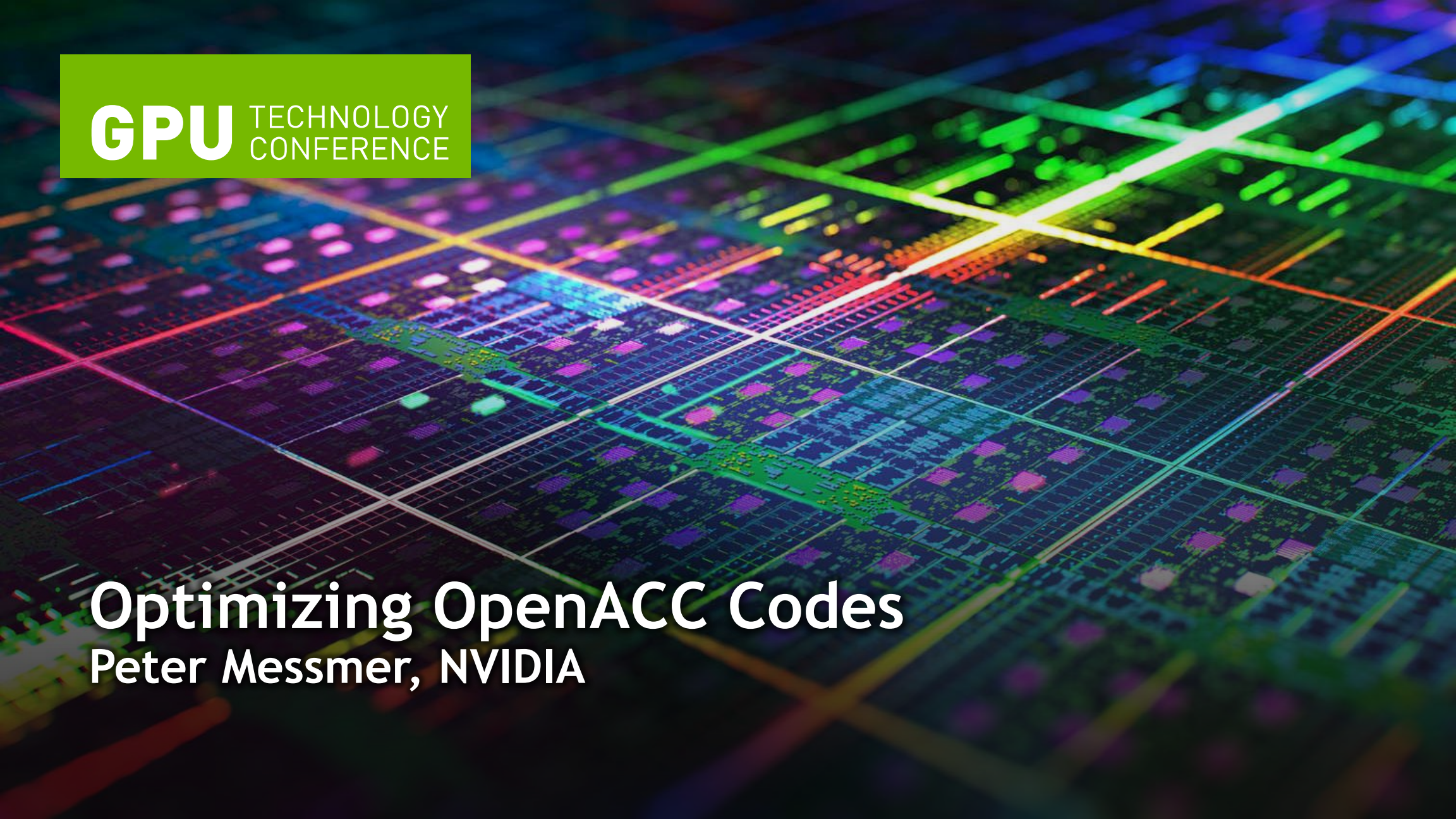
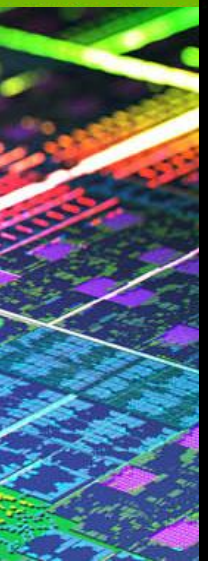


Optimizing OpenACC Codes

Peter Messmer, NVIDIA



Outline

- OpenACC in a nutshell
- Tune an example application
  - Data motion optimization
  - Asynchronous execution
  - Loop scheduling optimizations
- Interface OpenACC with libraries/CUDA
  - Interface with OpenGL
- Summary

3 Ways to Accelerate Applications

Applications

Libraries	OpenACC Directives	Programming Languages
"Drop-in" Acceleration	Easily Accelerate Applications	Maximum Flexibility

Directives for expressing parallelism

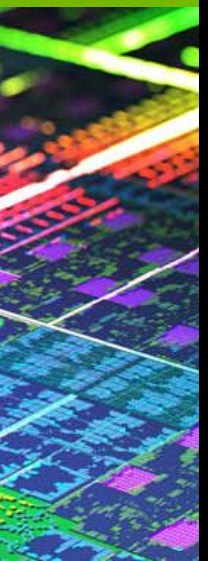
- Programmer identifies a loop as having parallelism, compiler generates a parallel kernel for that loop.

```
$!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`.



Directives for expressing parallelism

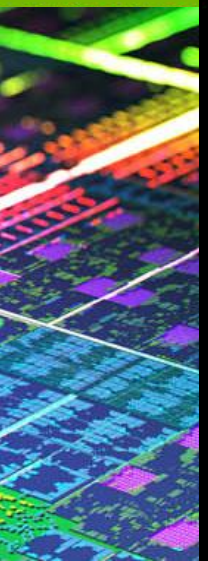
- The kernels construct expresses that a region may contain parallelism and the compiler determines what can safely be parallelized.

```
!$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.



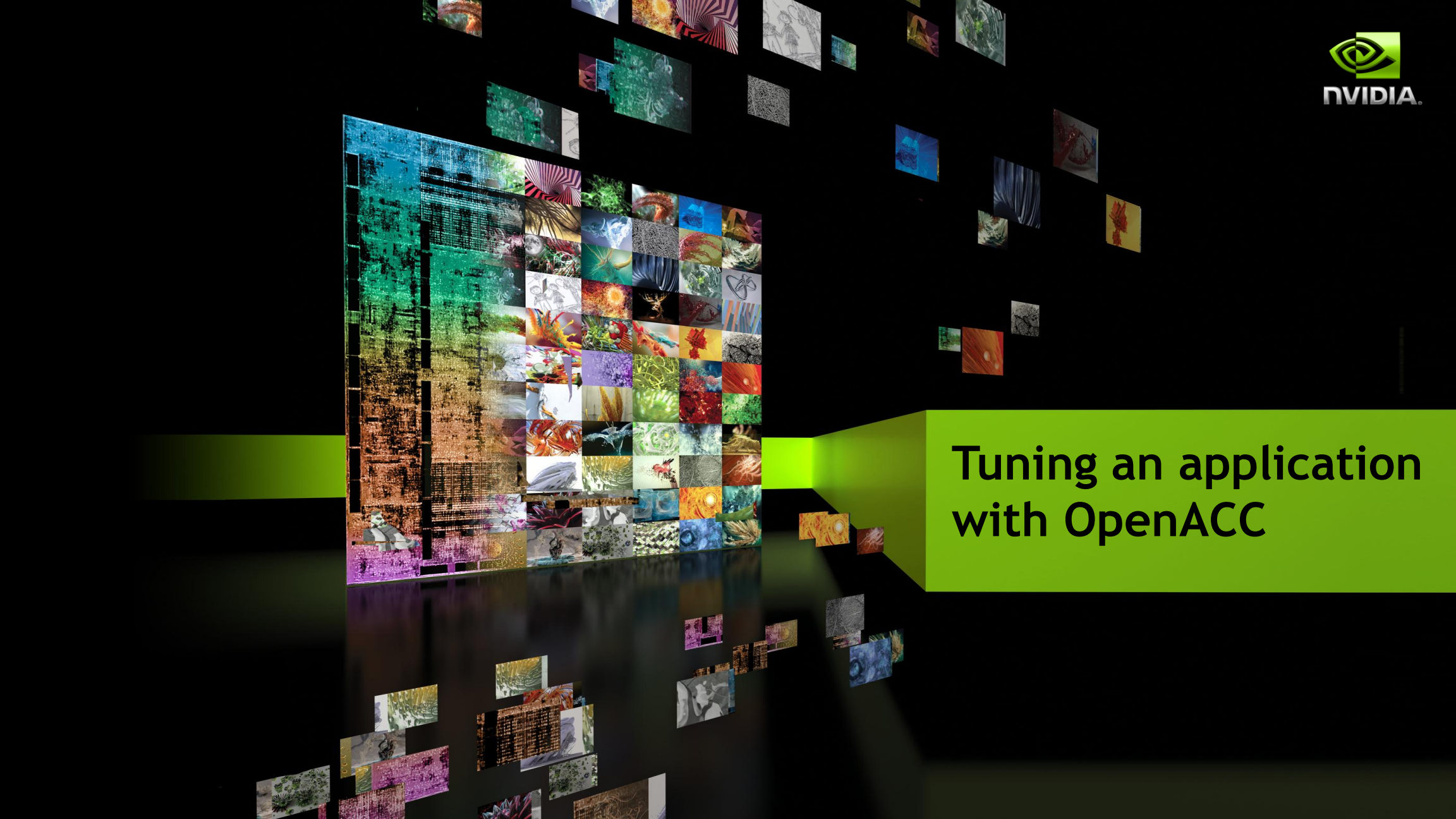
Directives for expressing data locality

- 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.

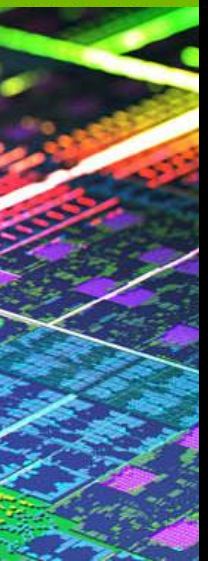
```
!$acc data
      !$acc parallel loop
       …

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

Data Region

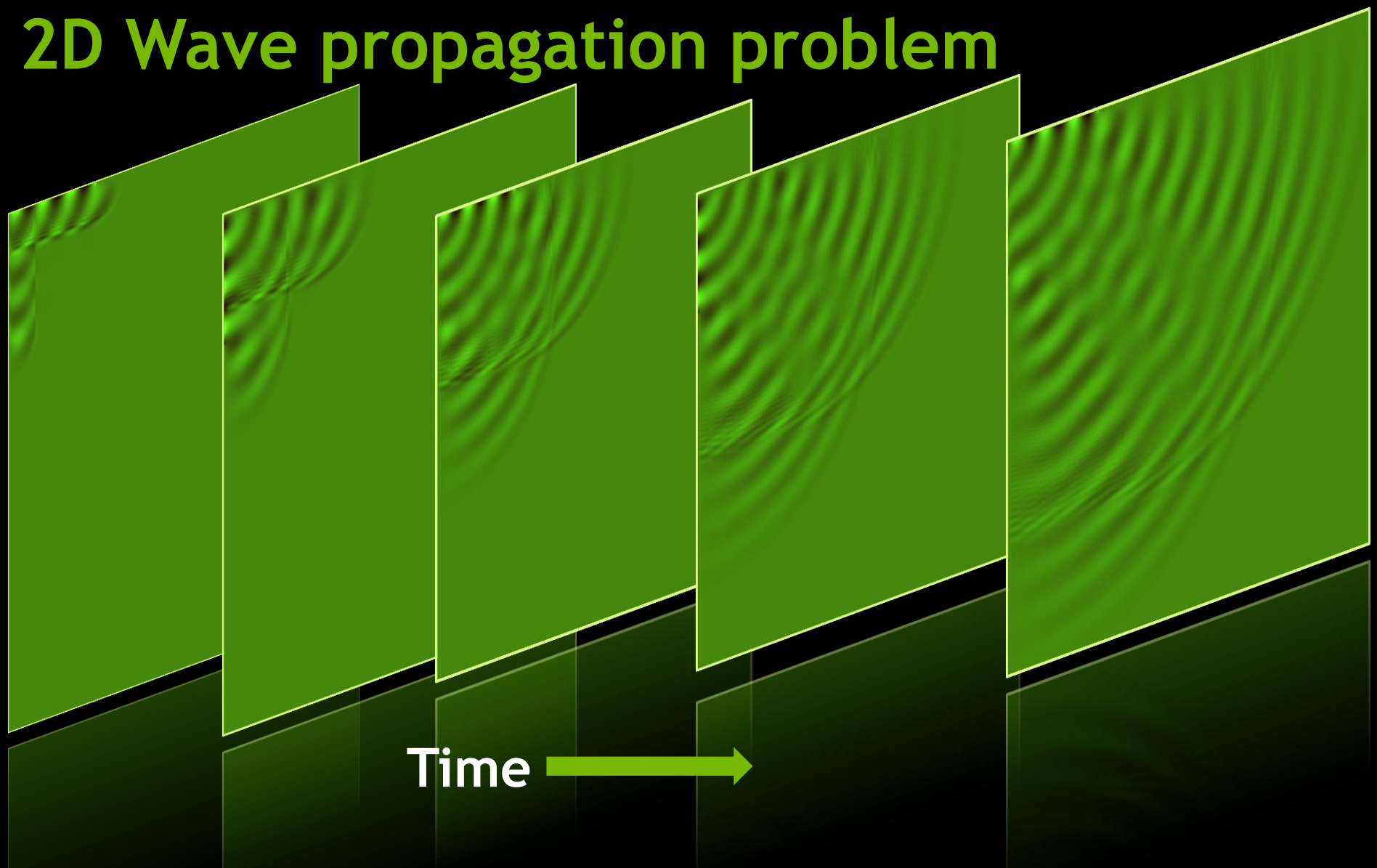
Arrays used within the data region will remain on the GPU until the end of the data region.



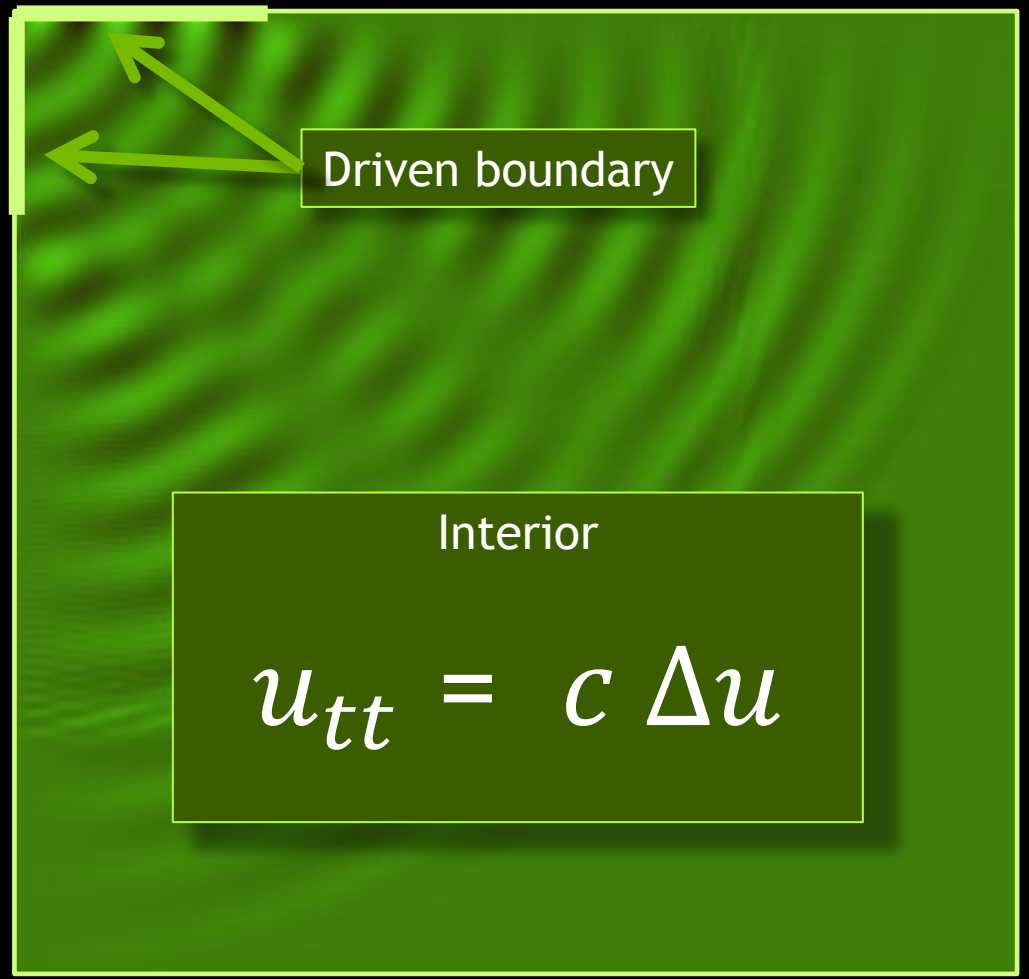
Tuning an application with OpenACC



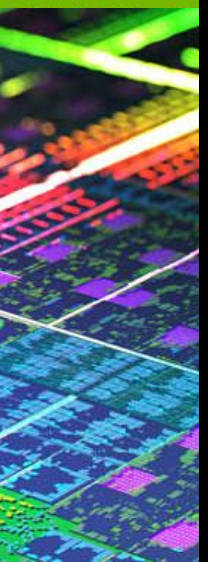
2D Wave propagation problem

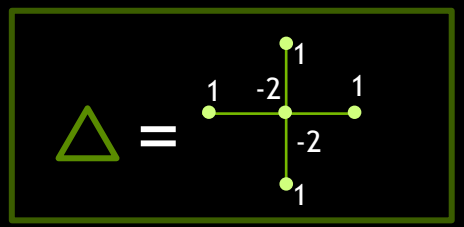


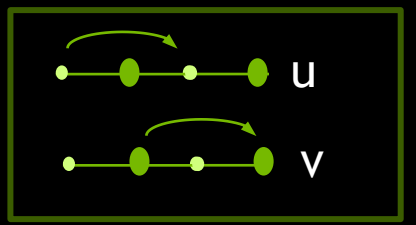
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$

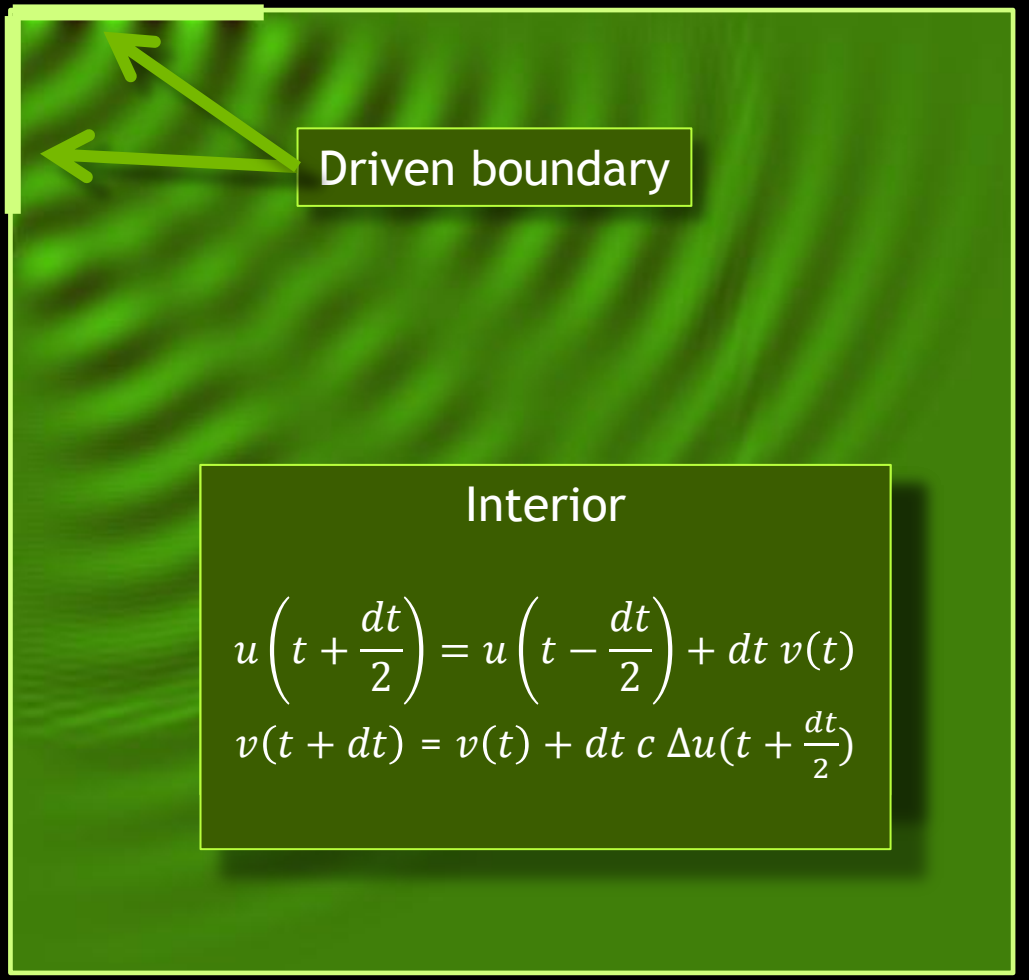


A basic 2D scalar wave solver: $u_{tt} = c\Delta u$

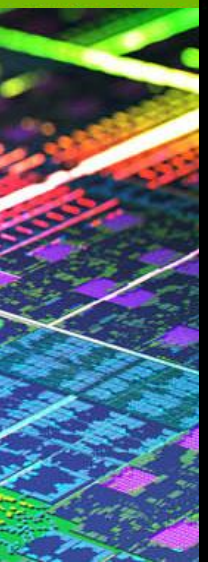


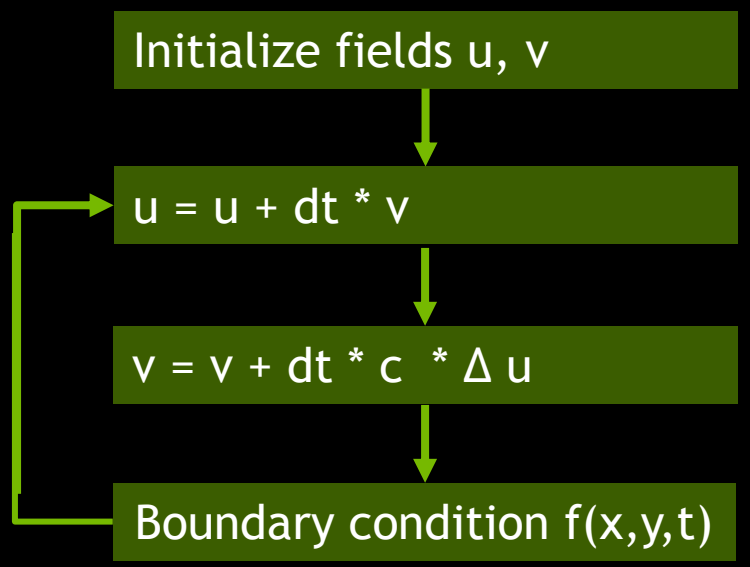


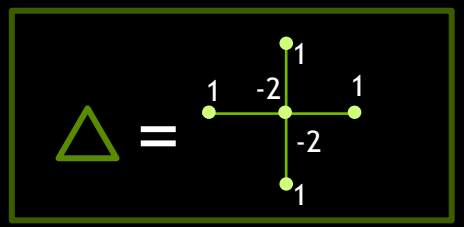


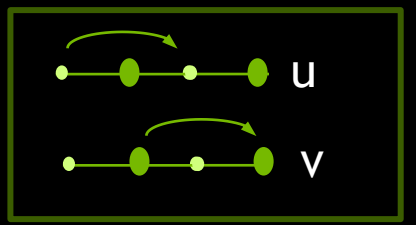


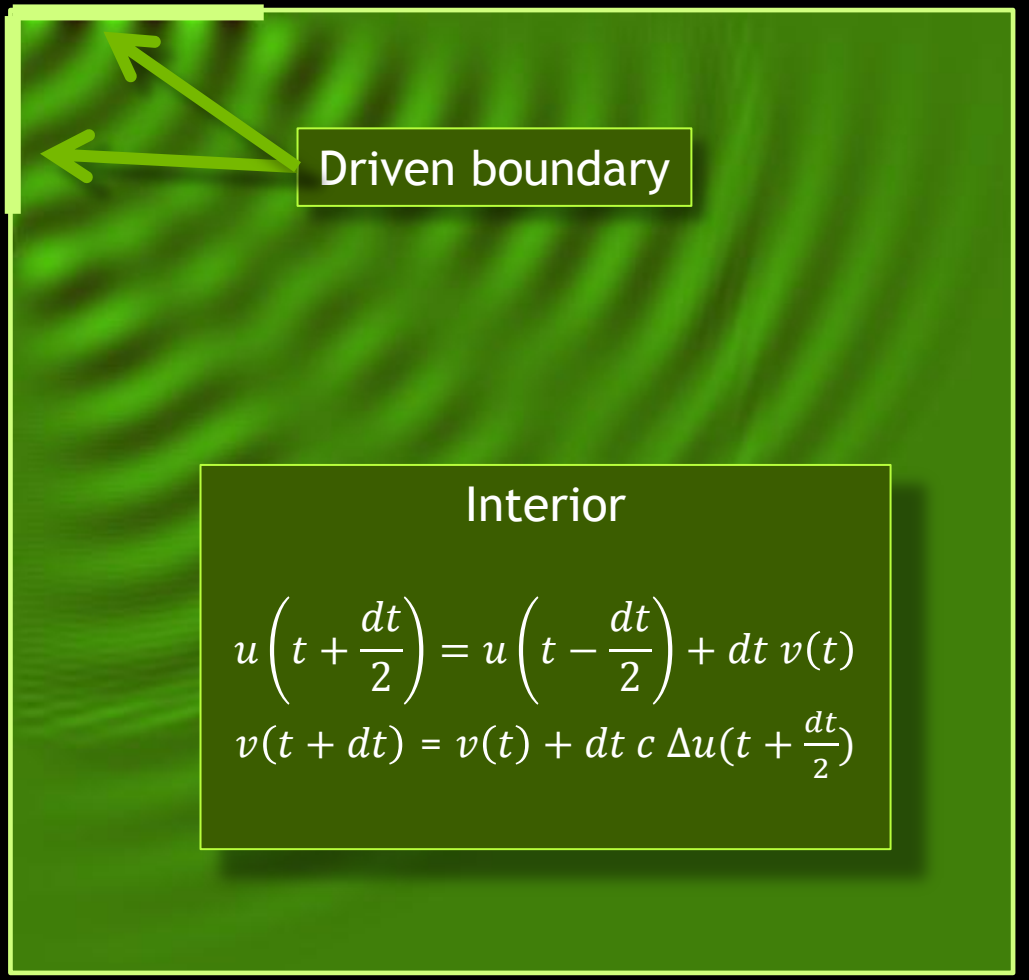
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$



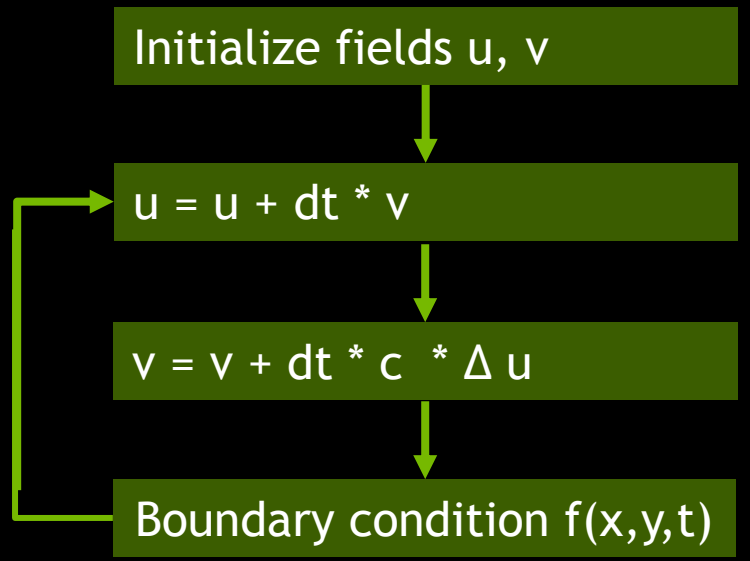






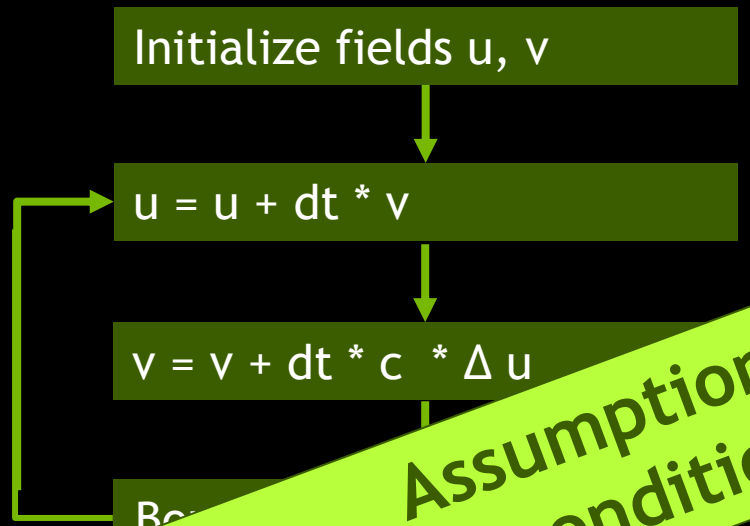


Sample: Basic Euler Solver




```
do t = 1, 1000
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
        do x=2,nx-1
                  v(x,y) = v(x,y) + dt * c * (  &
                        (u(x,y+1) - 2*u(x,y) + u(x,y-1)) / dx2 +  &
                        (u(x+1,y) - 2*u(x,y) + u(x-1,y)) / dy2 )
        end do
   end do

   call BoundaryConditon(u)
end do
```

Sample: Basic Euler Solver




```
do t = 1, 1000
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
        do x
                                    dt * c * (  &
                                (u(x,y+1) - 2*u(x,y) + u(x,y-1)) / dx2 +  &
                                (u(x+1,y) - 2*u(x,y) + u(x-1,y)) / dy2 )
        end do
   end do

   call BoundaryConditon(u)
end do
```

Assumption for the next few slides:
Boundary condition can only be computed on CPU

Initial attempt with OpenACC

```
do t = 1, 1000
!$acc kernels copy(u, v)
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
        do x=2,nx-1
                v(x,y) = v(x,y) + dt * c * (  &
                        (u(x,y+1) - 2*u(x,y) + u(x,y-1)) / dx2 +  &
                        (u(x+1,y) - 2*u(x,y) + u(x-1,y)) / dy2 )
        end do
   end do
!$acc end kernels
   call BoundaryConditon(u)
end do
```

Kernel 1

Kernel 2

Boundary condition on CPU

Compilation with OpenACC turned on

pgf90 -acc -Minfo=acc euler.F90

```
euler:
     29, Generating copy(v(:,:))
         Generating copy(u(:,:))
     33, Generating present_or_copy(u(:,:))
         Generating present_or_copy(v(:,:))
     34, Loop is parallelizable
         Accelerator kernel generated
         34, !$acc loop gang ! blockidx%y
             !$acc loop gang, vector(128) ! blockidx%x threadidx%x
     36, Loop is parallelizable
     37, Loop is parallelizable
         Accelerator kernel generated
         36, !$acc loop gang ! blockidx%y
         37, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

Profile your application

Use compiler output to determine how loops were mapped onto the accelerator

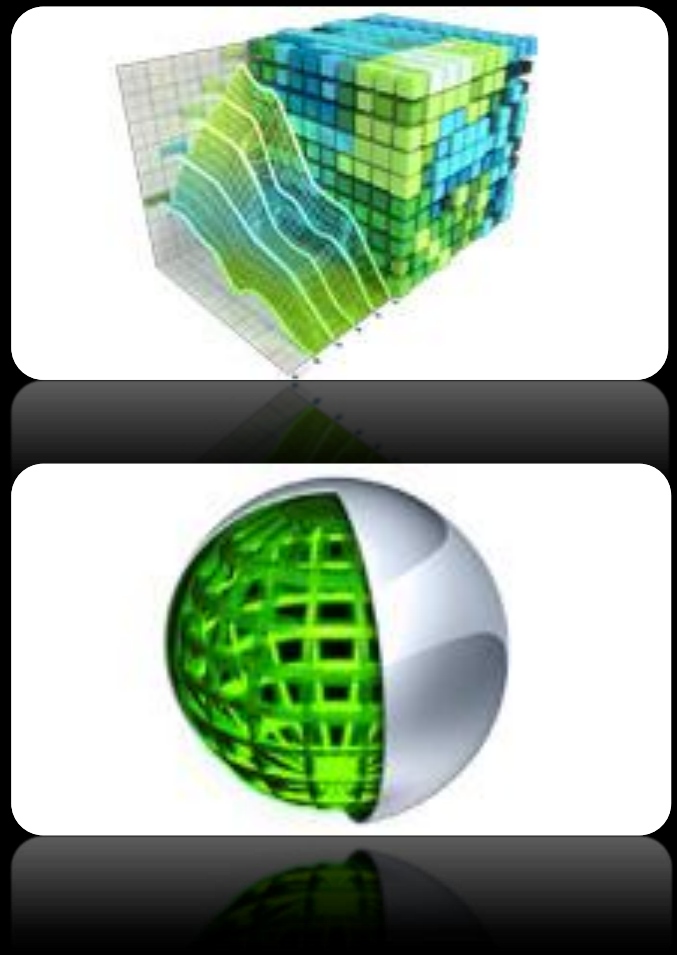
- Not exactly “profiling”, but it’s helpful information that a GPU-aware profiler would also have given you

PGI: Use PGI_ACC_TIME option to learn where time is being spent

NVIDIA Visual Profiler

3rd-party profiling tools that are CUDA-aware

- (But those are outside the scope of this talk)



Our test environment

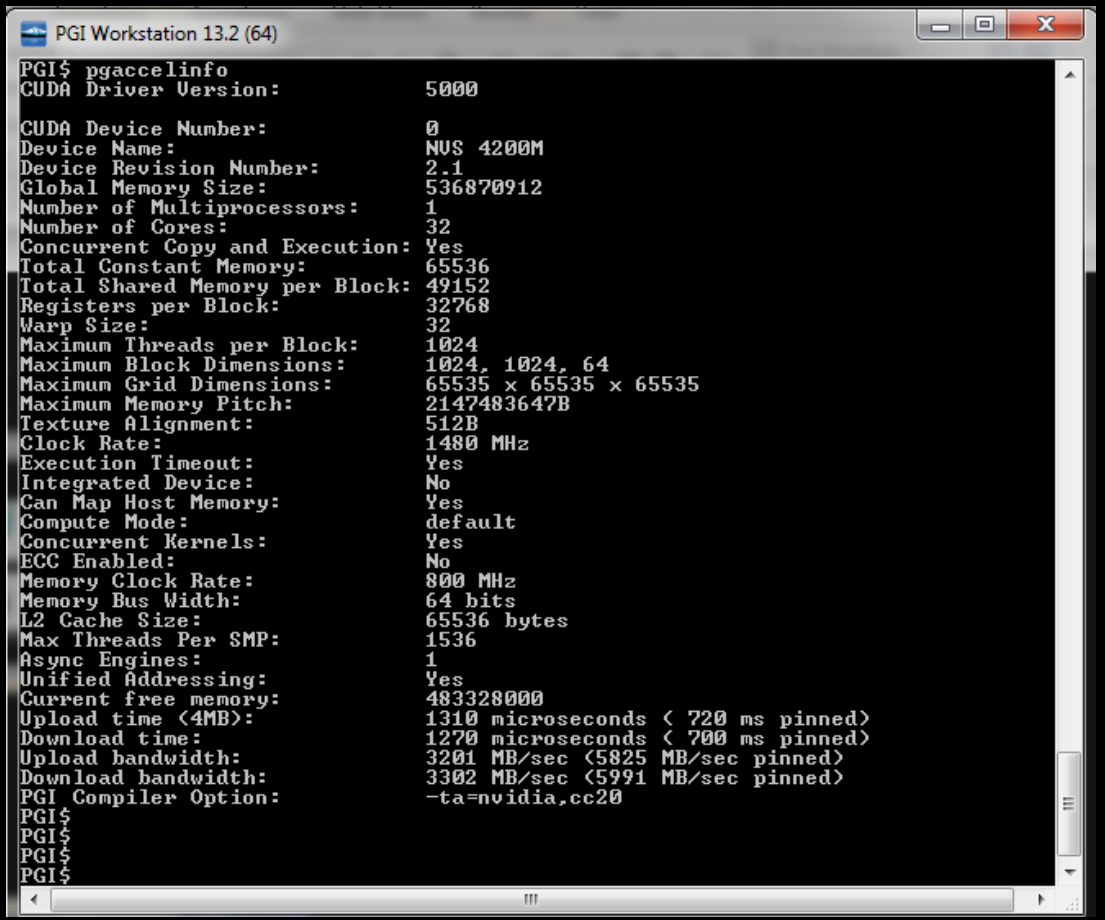
OpenACC is cross-platform!
Run on laptop with GPU

Fermi Generation GPU

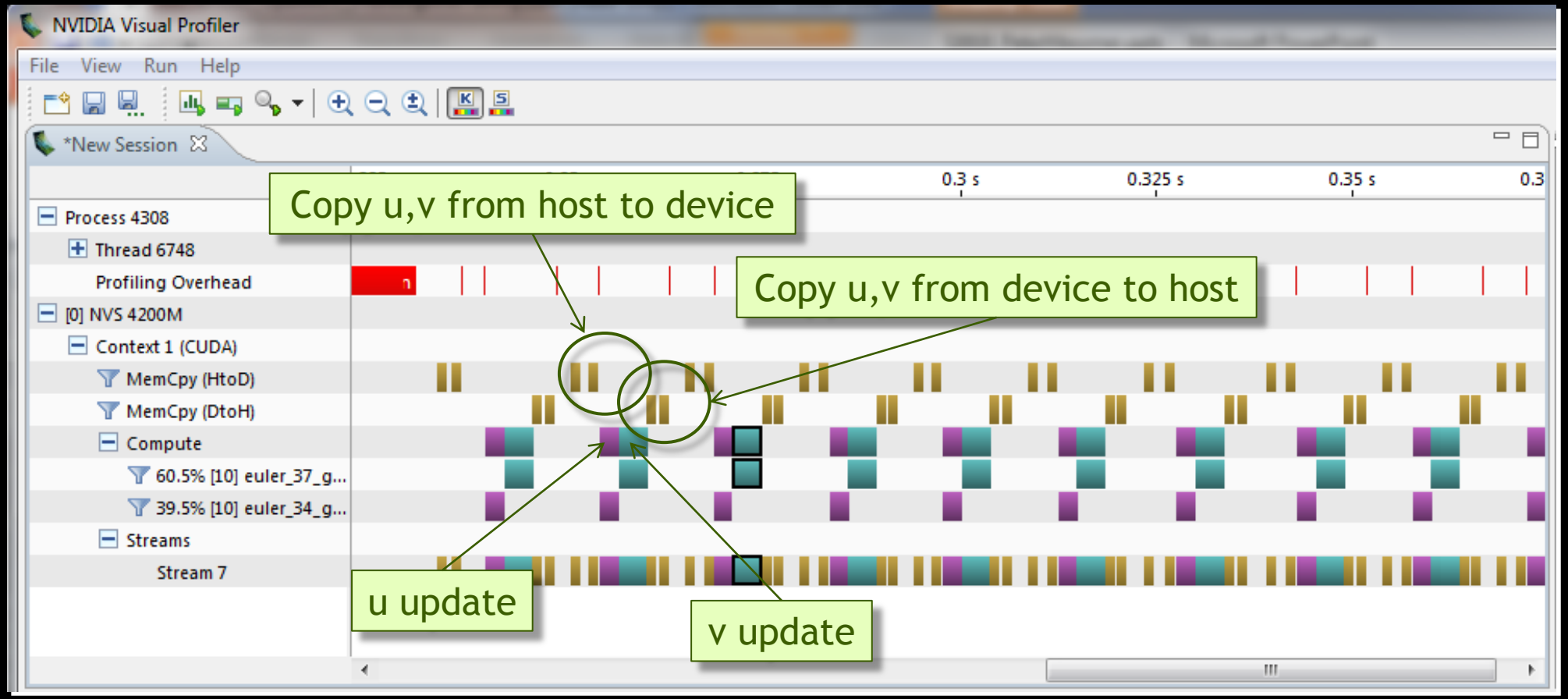
Single Multi-Processor

Up to 48 warps/SM

Up to 8 blocks per SM



Profiling via NVVP



Optimization: Keep data on GPU

```
!$acc data copy(u, v)
do t = 1, 1000
  !$acc kernels
   u(:,:) = u(:,:) + dt  * v(:,:)

   do y=2, ny-1
         do x=2,nx-1
                  v(x,y) = v(x,y) + dt * c * …
         end do
    end do
  !$acc end kernels
   call BoundaryConditon(u)
end do
!$acc end data
```

u,v on GPU for the duration of the loop

Optimization: Keep data on GPU

```
!$acc data copy(u, v)
do t = 1, 1000
  !$acc kernels
   u(:,:) = u(:,:) + dt  * v(:,:)

   do y=2, ny-1
         do x=2,nx-1
                     v(
         end do
   end do
  !$acc end
   call BoundaryC
end do
!$acc end data
```

WRONG!!!!

u,v on GPU for the duration of the loop

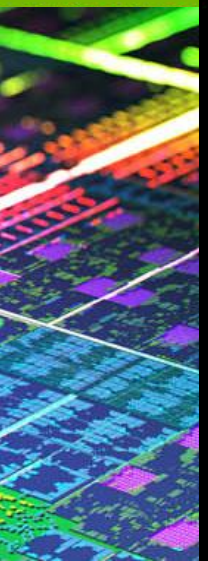
Beware of the separate address spaces!

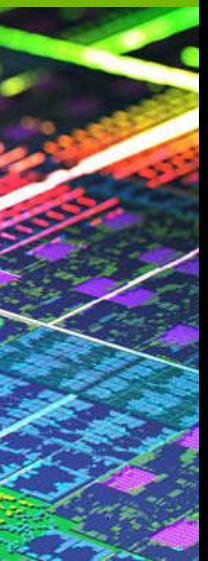
```
!$acc data copy(u, v)
do t = 1, 1000
  !$acc kernels
   u(:,:) = u(:,:) + dt  * v(:,:)

   do y=2, ny-1
         do x=2,nx-1
                  v(x,y) = v(x,y) + dt * c * …
         end do
   end do
  !$acc end kernels
   call BoundaryConditon(u)
end do
!$acc end data
```

Wave launcher along x and y

Modifies u,v on GPU

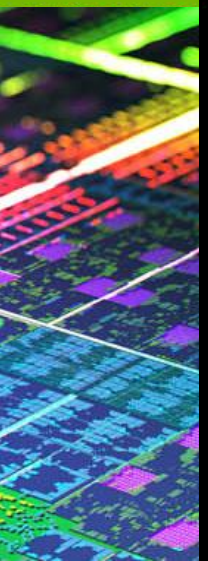
Modifies u on CPU





Always have a sign of quality/checksum!!

- Annotating code with OpenACC is simple
  - Bad use can lead to wrong results
- Always validate the results of OpenACC optimizations
- Ideally a simple, yet comprehensive test
  - Total energy in the system
  - Total number of non-zero values
  - …



OpenACC update Directive

Programmer specifies an array (or partial array) that should be refreshed within a data region.

```
do_something_on_device()

!$acc update host(a)

do_something_on_host()

!$acc update device(a)
```

Copy “a” from GPU to CPU

Copy “a” from CPU to GPU

The programmer may choose to specify only part of the array to update.

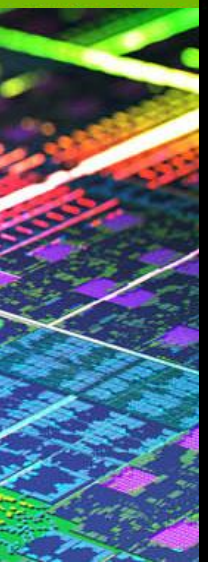
update directive for boundaries

```
!$acc data copy(u, v)
do t = 1, 1000
  !$acc kernels
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
        do x=2,nx-1
                  v(x,y) = v(x,y) + dt * c * …
        end do
  end do
  !$acc end kernels
  !$acc update host(u(1:nx/4,1:2))
  call BoundaryConditon(u)
  !$acc update device(u(1:nx/4, 1:2)
end do
!$acc data copy(u, v)
```

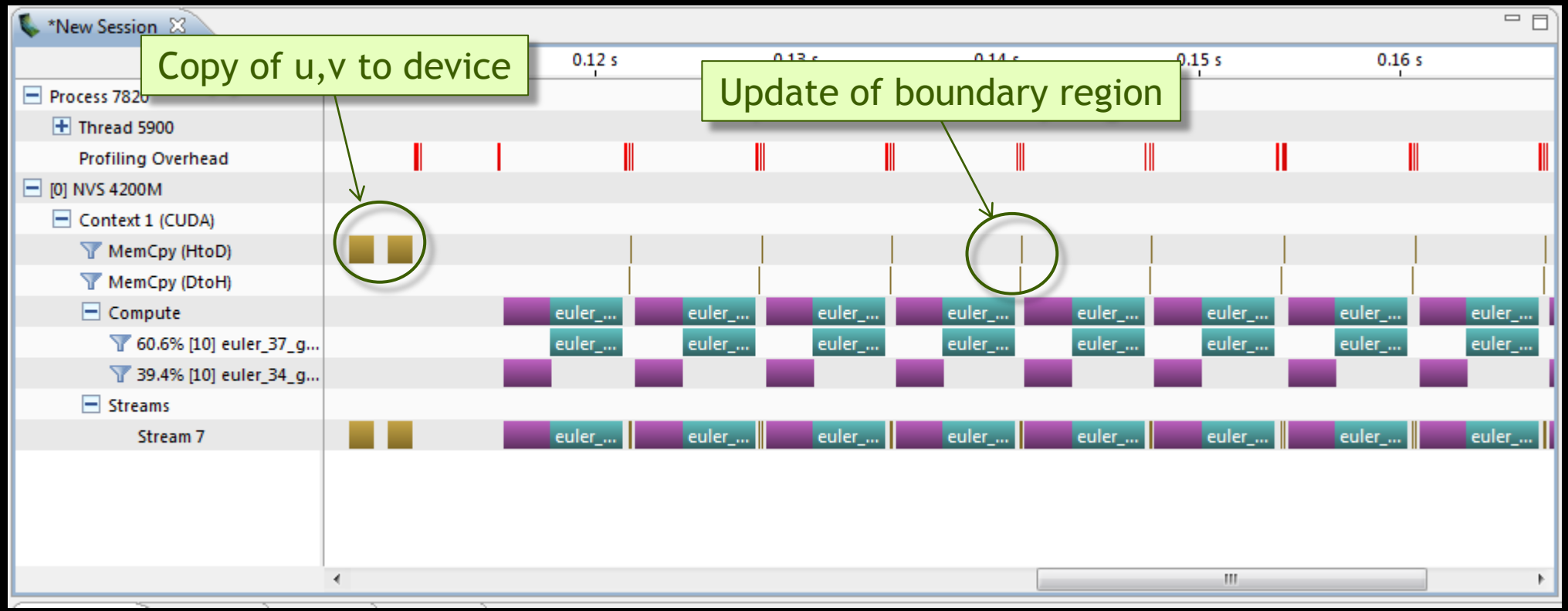
Modifies GPU version of u,v

Pull fraction of u back to CPU

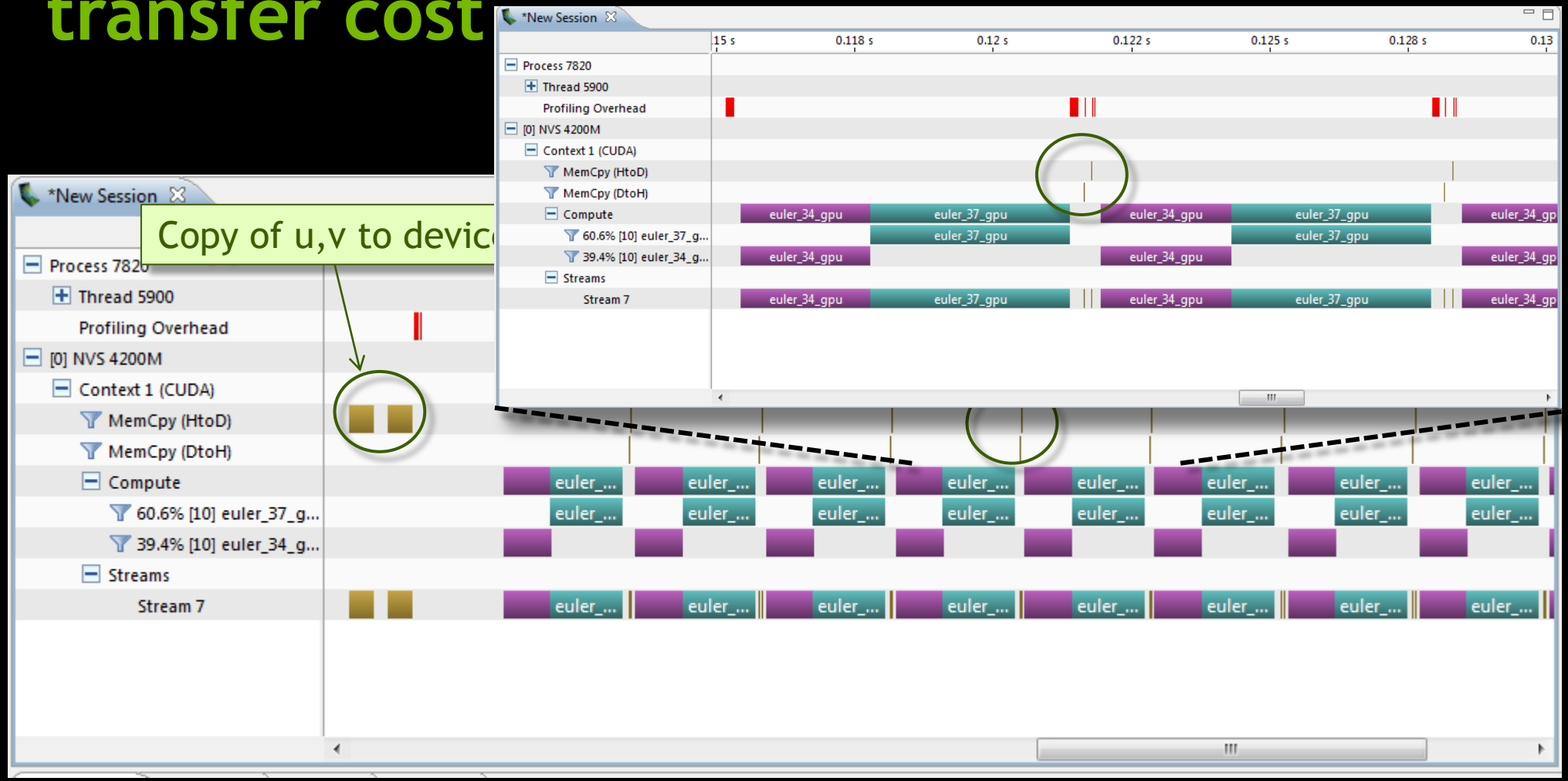
Push fraction of u back to GPU



`update` significantly reduces data transfer cost



`update` significantly reduces data transfer cost



update directive for boundaries

```
!$acc data copy(u, v)
do t = 1, 1000
 !$acc kernels
  u(:,:) = u(:,:) + dt  * v(:,:)

  do y=2, ny-1
        do x=2,nx-1
                  v(x,y) = v(x,y) + dt * c * …
        end do
  end do
 !$acc end kernels
 !$acc update host(u(1:nx/4,1:2))
  call BoundaryConditon(u)
 !$acc update device(u(1:nx/4, 1:2)
end do
!$acc data copy(u, v)
```

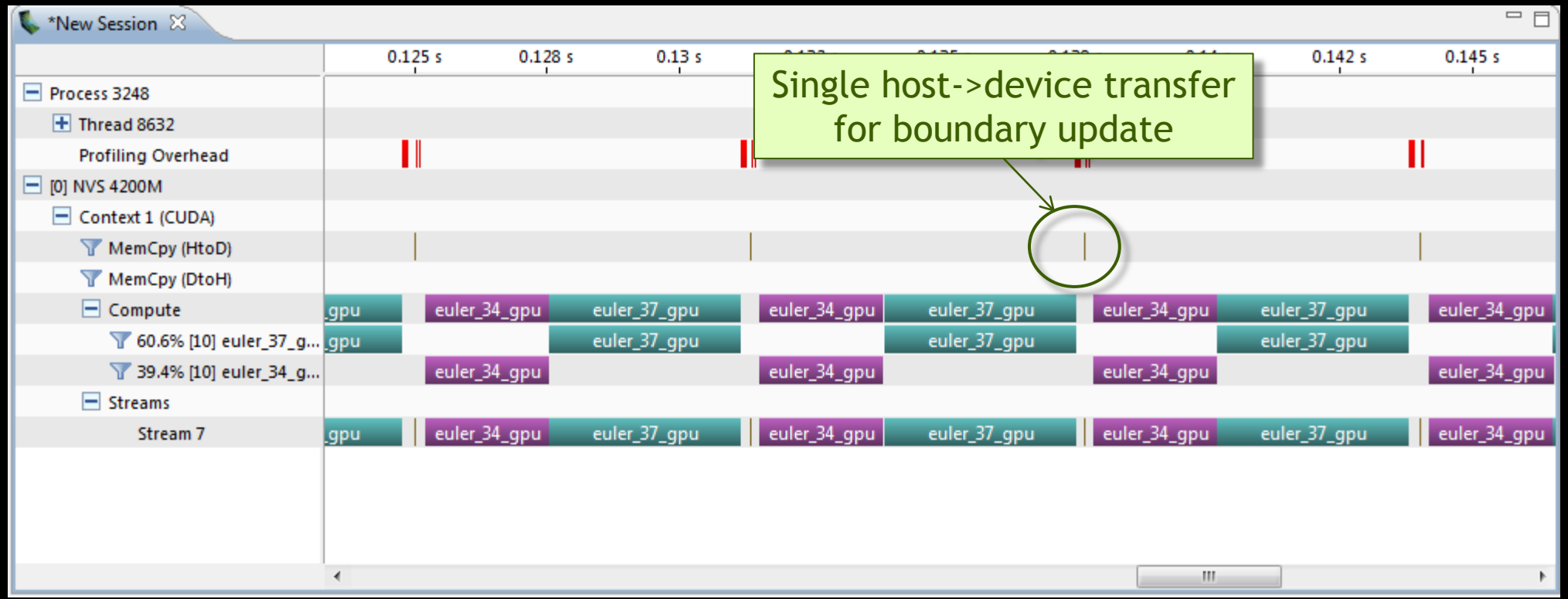
Unnecessary for "set" boundaries (Dirichlet)

ion of u,v

Pull fraction of u back to CPU

Push fraction of u back to GPU

Reducing data transfer even further



Summary of Data Motion Optimizations

- Basic OpenACC annotation

```
!$acc kernels
!$acc parallel loop
```

- Resident GPU data

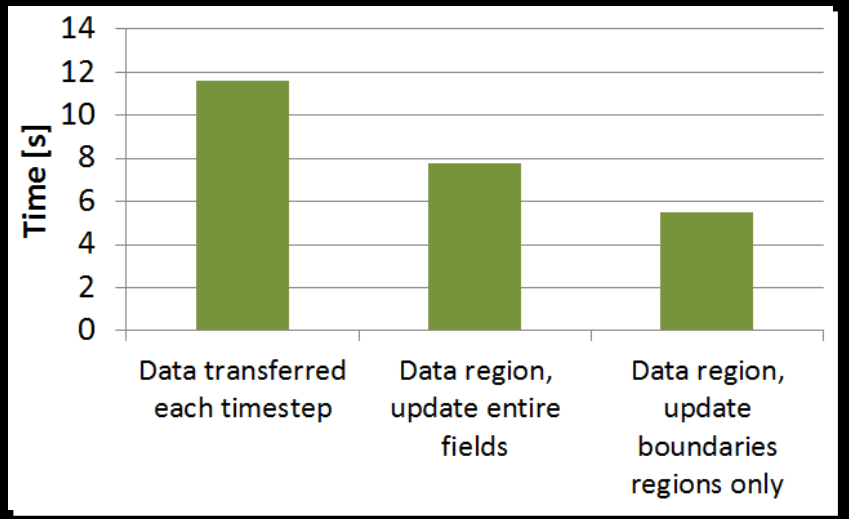
```
!$acc data copy(a)
```

- Partial variable update

```
!$acc update host(a) device(b)
```

- Subarray specification

```
!$acc update host(a(1:nx/4,1))
```



Assumption:
Boundaries computed on CPU

What if we compute boundaries on GPU?

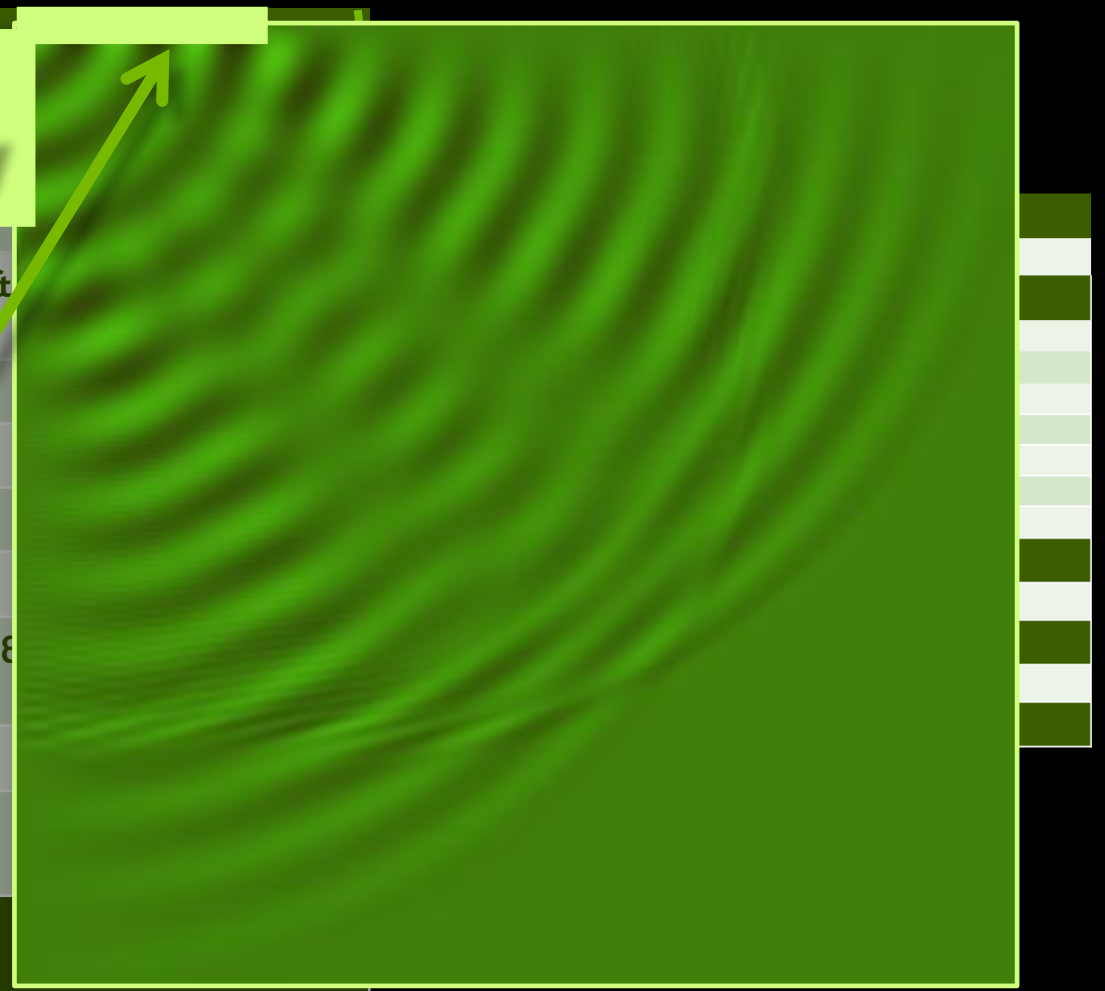
```
!$acc kernels
do y=1, 2
   do x=1, nx/4
         u(x,y) = sin(6.28*real(t) + real(x)/6.28) * &
                  cos(6.28*real(t) + real(y)/6.28)
    end do
end do
do y=1, ny/4
    do x=1, 2
         u(x,y) = (sin(6.28*real(t) + real(x)/6.28) * &
                   cos(6.28*real(t) + real(y)/6.28))
   end do
 end do

!$acc end kernels
```

```
!$acc data copy(u, v)
do t = 1, 1000
 !$acc kernels
 u(:,:) = u(:,:) + dt  * v(:,:)

 do y=2, ny-1
        do x=2,nx-1
                    v(x,y) = v(x,y) + dt * c * ...
        end do
 end do
 !$acc end kernels
 call BoundaryConditon(u)
 !$acc update device(u(1:nx/4, 1:2)
end do
!$acc data copy(u, v)
```

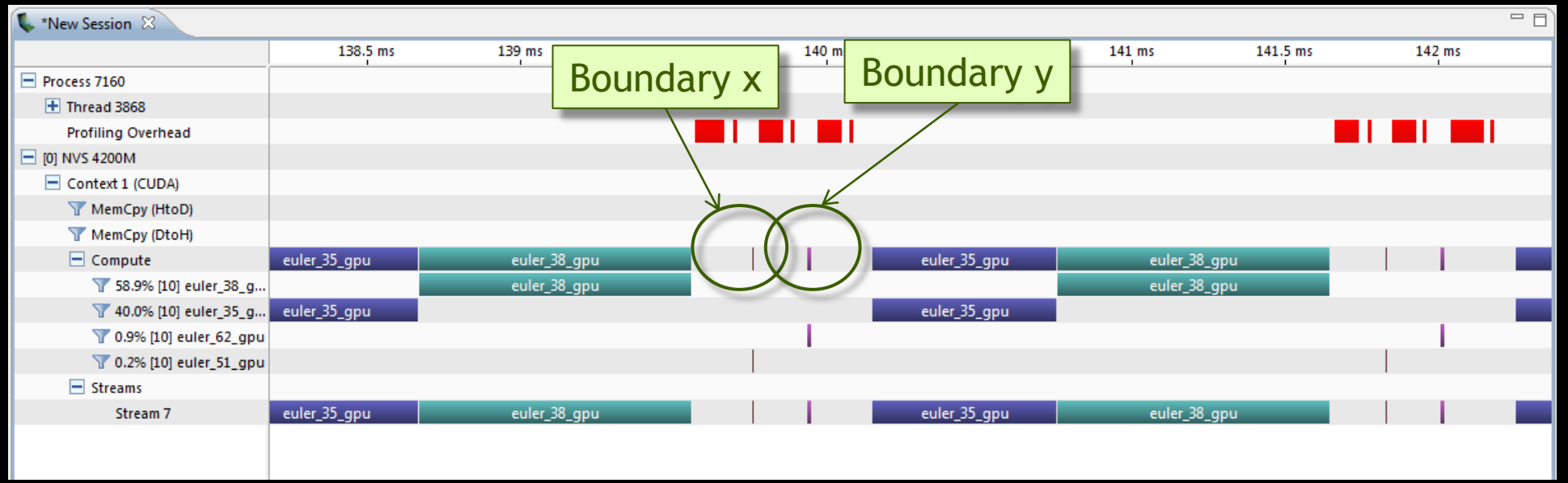
Wave launcher along x and y

What if we compute boundaries on GPU?

```
!$acc kernels
do y=1, 2
   do x=1, nx/4
      u(x,y) = sin(6.28*real(t) + real(x)/6.28) * &
               cos(6.28*real(t) + real(y)/6.28)
   end do
end do
do y=1, ny/4
   do x=1, 2
      u(x,y) = (sin(6.28*real(t) + real(x)/6.28) * &
               cos(6.28*real(t) + real(y)/6.28))
   end do
end do
!$acc end kernels
```



Boundary time is minimal .. But could we optimize it even further?



OpenACC async clause

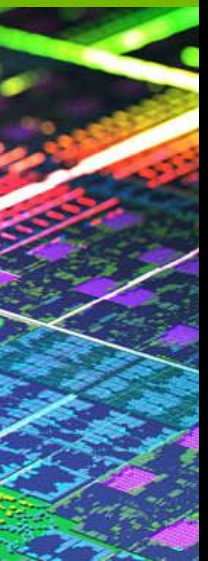
```
!$acc kernels async(1)
…
!$acc end kernels

do_something_on_host()

!$acc parallel async(2)
…
!$acc end parallel


!$acc wait(2)
!$acc wait
```

> The `async` clause is optional on the `parallel` and `kernels` constructs; when there is no `async` clause, the host process will wait until the parallel or kernels region is complete before executing any of the code that follows the construct. When there is an `async` clause,

Do not wait for kernel completion

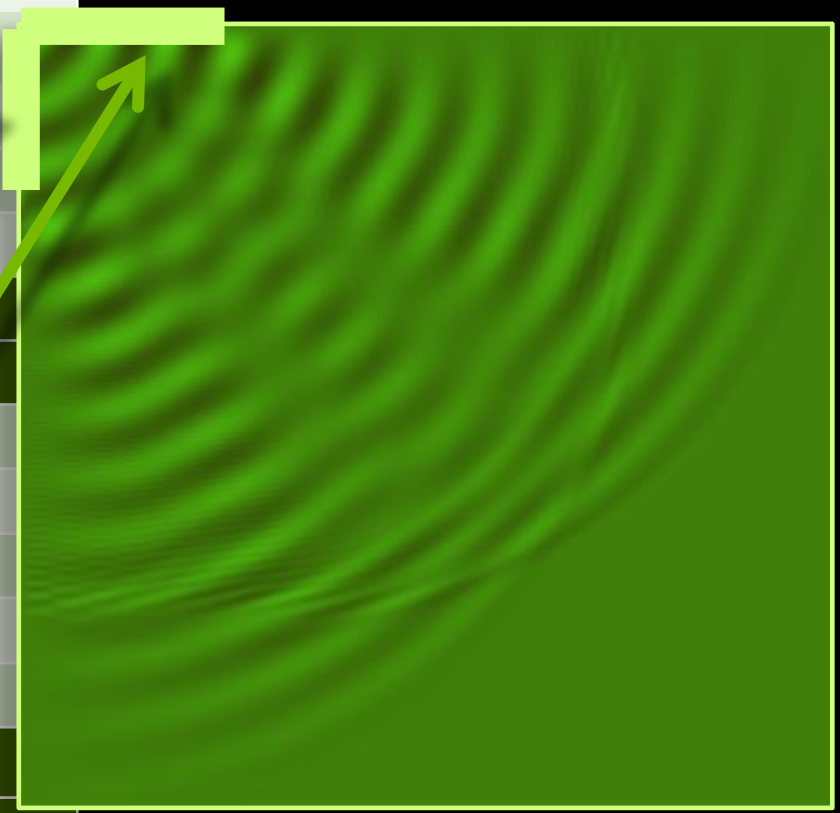
Executes concurrently with kernels

(potentially) executes concurrently with kernels

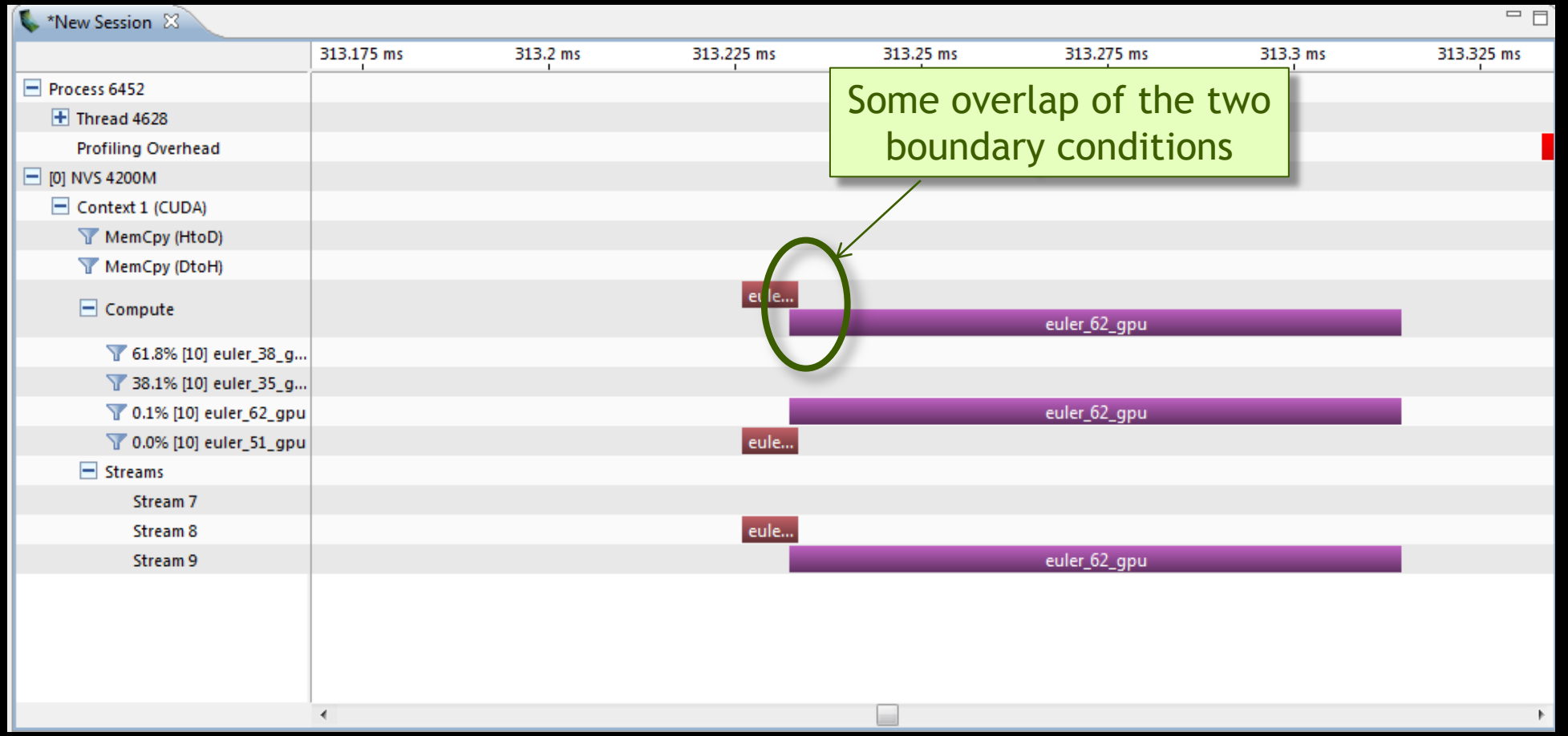


Set Boundary Regions concurrently

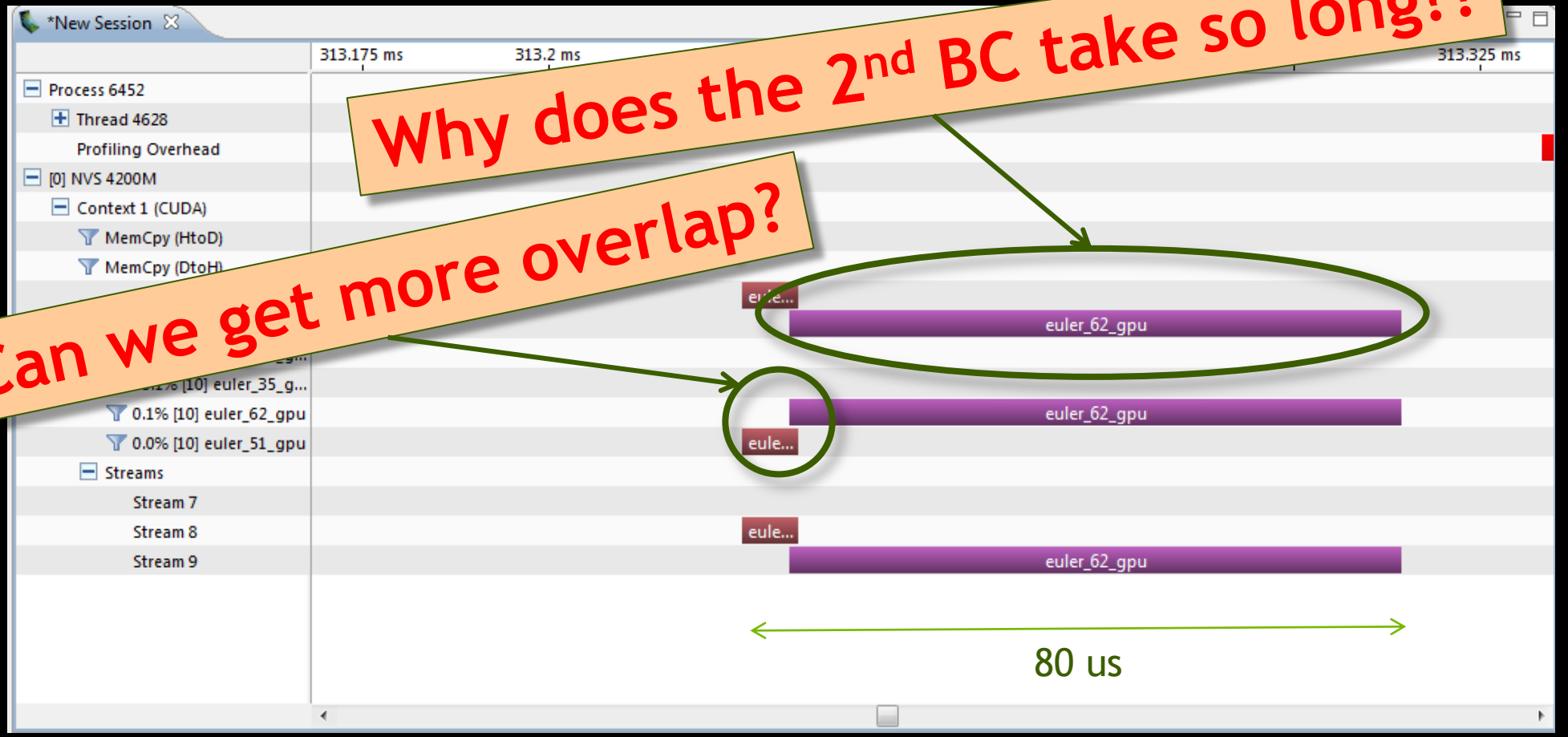
```
!$acc kernels async(1)
do y=1, 2
   do x=1, nx/4
        u(x,y) = sin(6.28*real(t) + real(x)/6.28) ..
    end do
end do
!$acc end kernels
!$acc kernels async(2)
do y=1, ny/4
    do x=1, 2
        u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
   end do
 end do
!$acc end kernels
!$acc wait
```



Async clause leads to overlapping kernels



Async clause leads to overlapping kernels

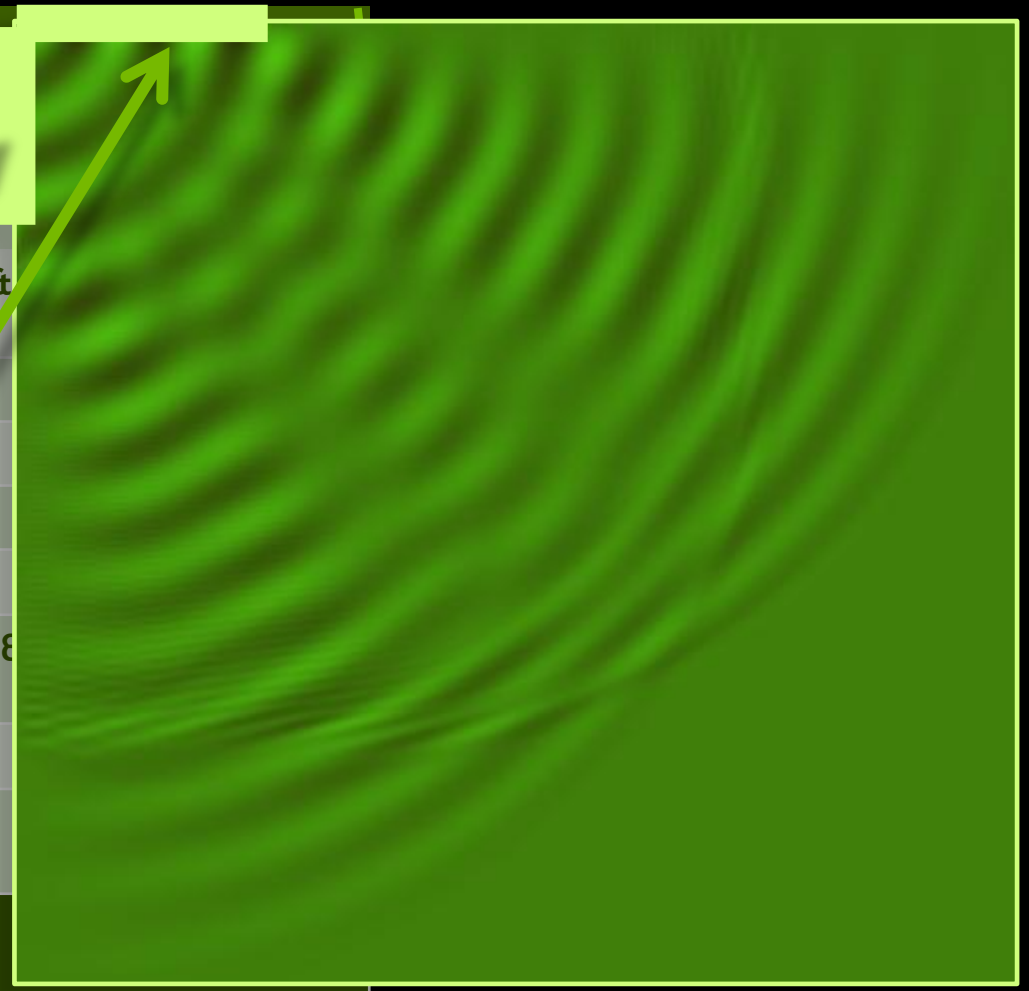


Why does the 2nd BC take so much longer?

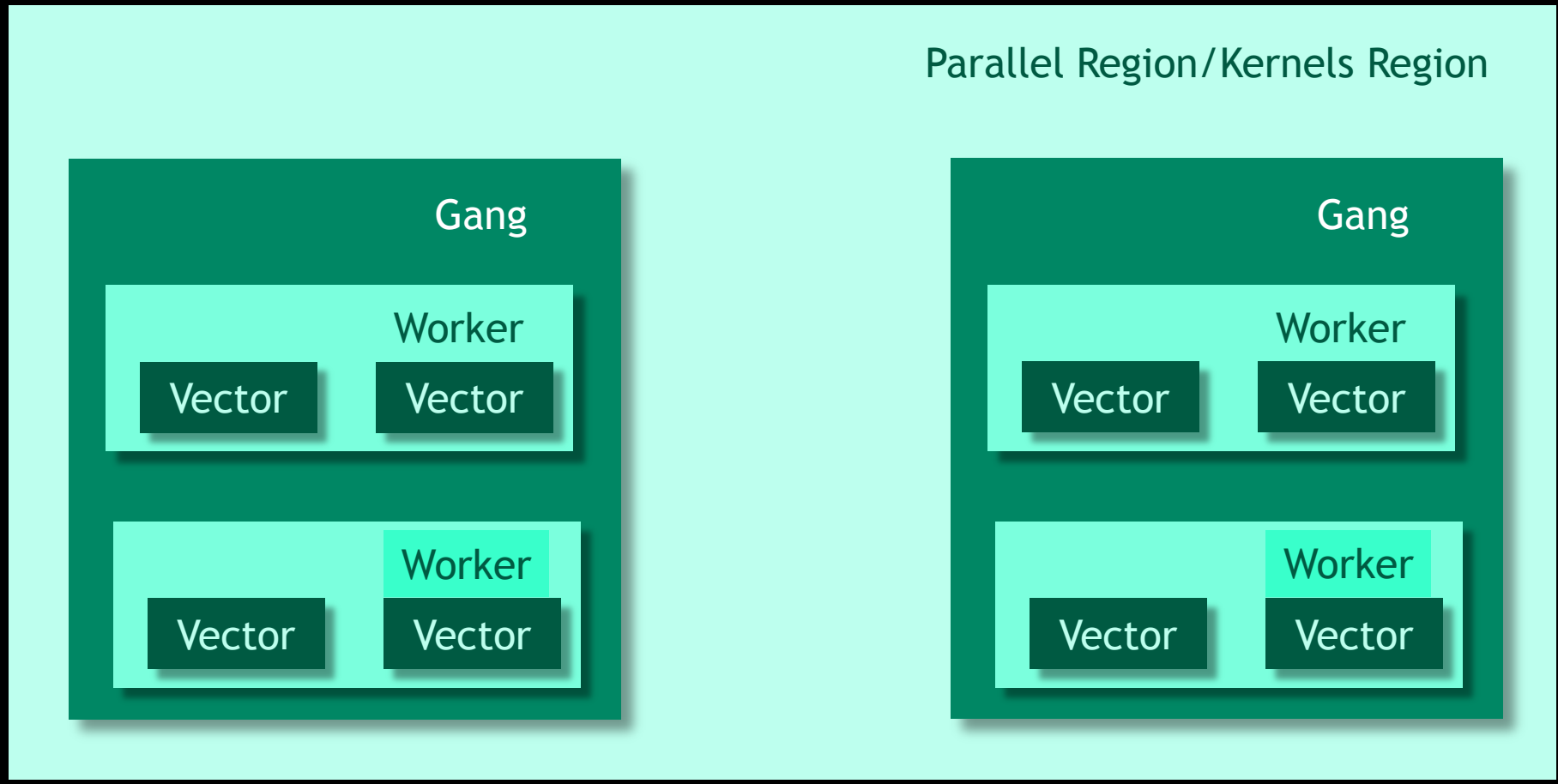
```
!$acc kernels
do y=1, 2
   do x=1, nx/4
      u(x,y) = sin(                + real(x)/6.28) * &
               co                  (t) + real(y)/6.28)
   end do
end do
do y=1, ny/4
   do x=1, 2
      u(x,y) = (si                 (t) + real(x)/6.28) * &
                                   al(t) + real(y)/6.28))
   end do
end do
!$acc end kernels
```

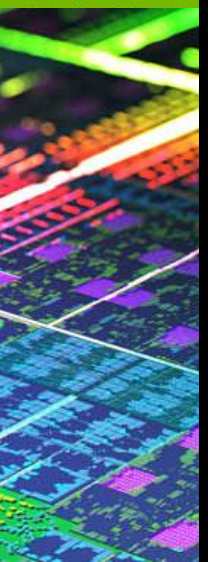
Fast

Slow

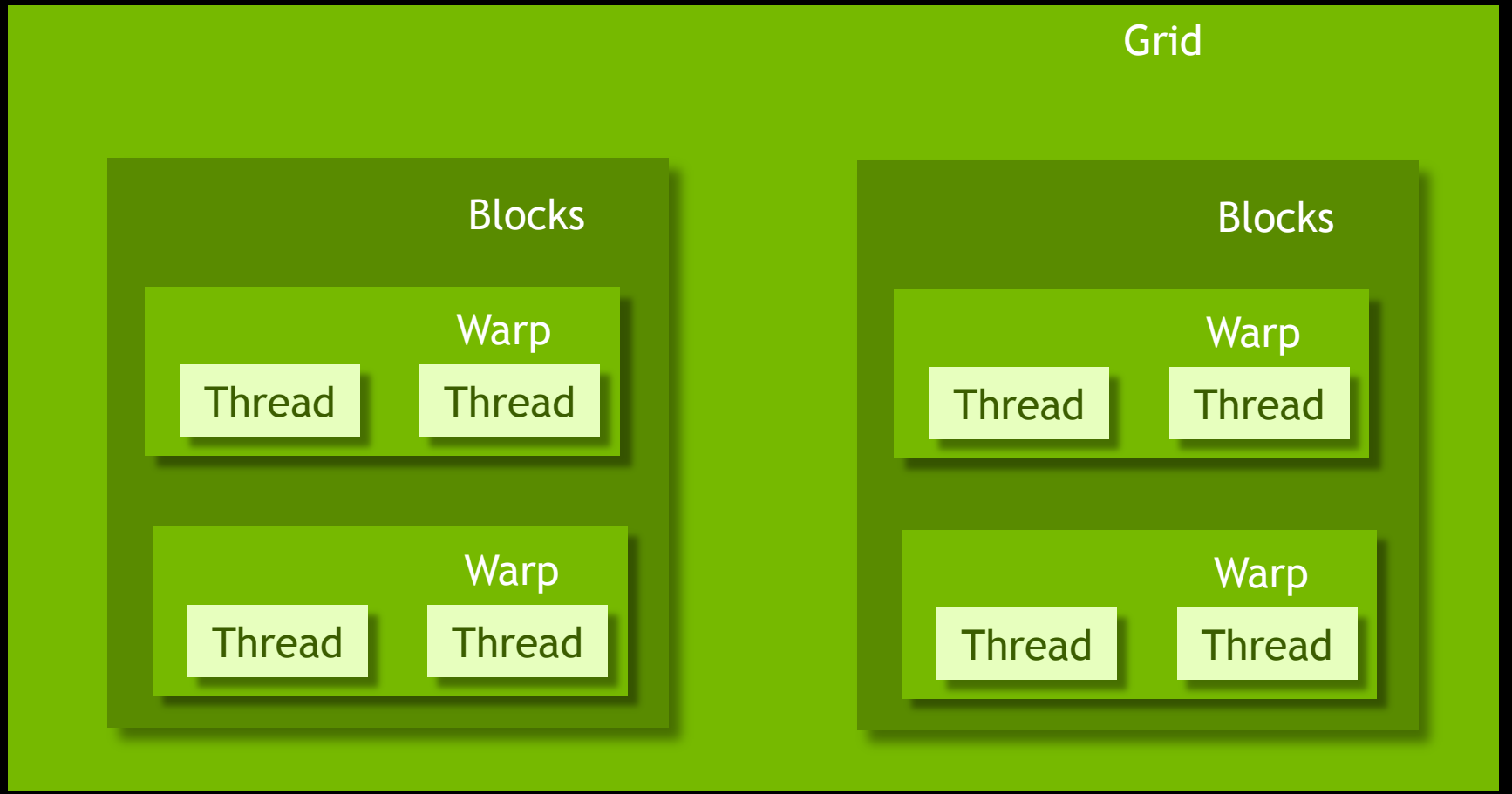


OpenACC's three levels of parallelism

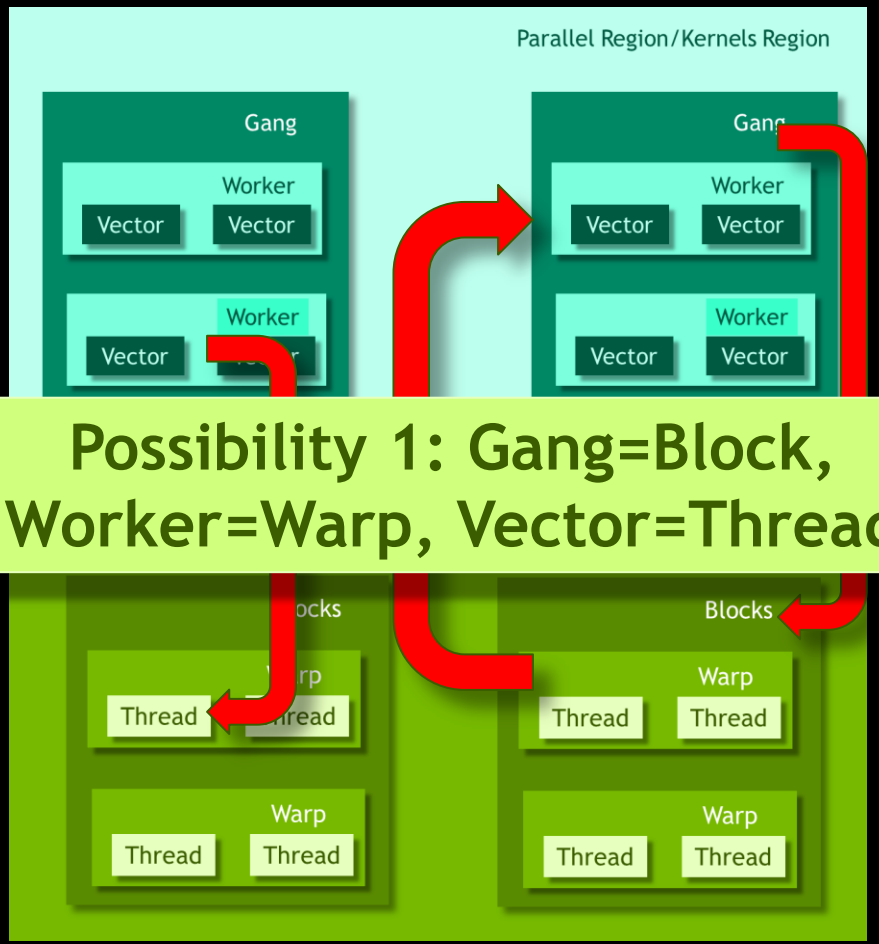




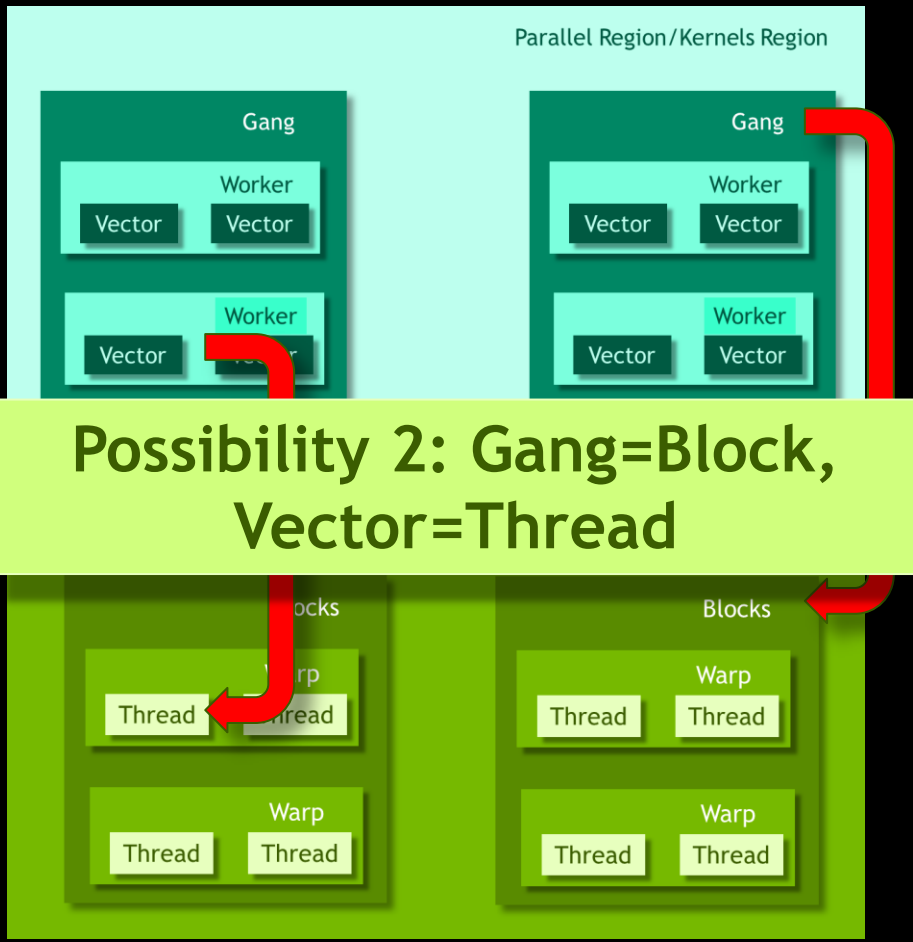
The CUDA execution model



Mapping OpenACC parallelism to CUDA is up to compiler



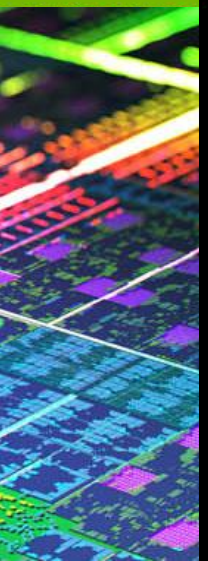
Possibility 1: Gang=Block, Worker=Warp, Vector=Thread



Possibility 2: Gang=Block, Vector=Thread

Compiler provides information about mapping

```
34, !$acc loop gang, vector(128) ! blockidx%x threadidx%
    45, Generating present_or_copy(u(:,:))
    46, Loop is parallelizable
    47, Loop is parallelizable
        Accelerator kernel generated
        46, !$acc loop gang ! blockidx%y
        47, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
    54, Generating present_or_copy(u(:,:))
    55, Loop is parallelizable
    56, Loop is parallelizable
        Accelerator kernel generated
        55, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
        56, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

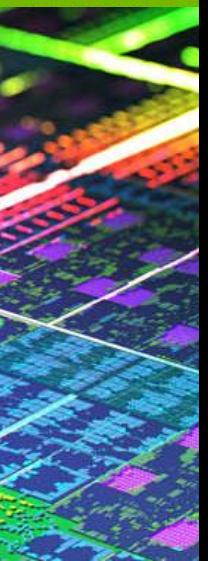



Compiler provides information about mapping

```
34, !$acc loop gang, vector(128) ! blockidx%x threadidx%
    present_or_copy(u(:,:))
    rallelizable
47, Loop is parallelizable
    Accelerator kernel generated
    46, !$acc loop gang ! blockidx%y
    47, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
54, Generating present_or_copy(u(:,:))
55, Loop is parallelizable
56, Loop is parallelizable
    Accelerator kernel generated
    55, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
    56, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```

OpenACC

CUDA

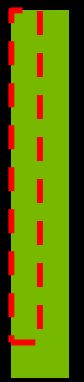


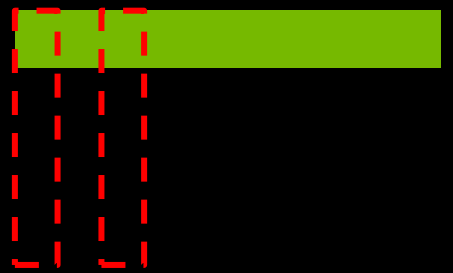
Mapping of boundary kernels to CUDA

```
!$acc kernels async(1)
do y=1, 2
   do x=1, nx/4
        u(x,y) = sin(6.28*real(t) + real(x)/6.28) ..
    end do
end do
!$acc end kernels
!$acc kernels async(2)
do y=1, ny/4
    do x=1, 2
        u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
   end do
 end do
!$acc end kernels
!$acc wait
```

```
46, !$acc loop gang ! blockidx%y
47, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
```

```
55, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
56, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```



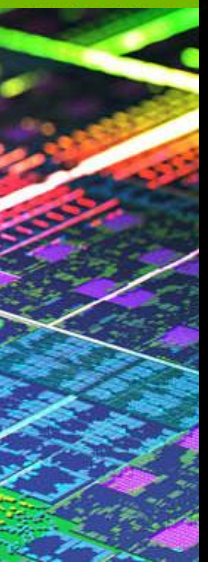


Mapping of boundary kernels to CUDA

32 threads in x, 30 threads idle

```
!$acc kernels async(2)
do y=1, ny/4
  do x=1, 2
    u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
  end do
end do
!$acc end kernels
!$acc wait
```

```
55, !$acc loop gang, vector(4) ! blockidx%y threadidx%y
56, !$acc loop gang, vector(32) ! blockidx%x threadidx%x
```



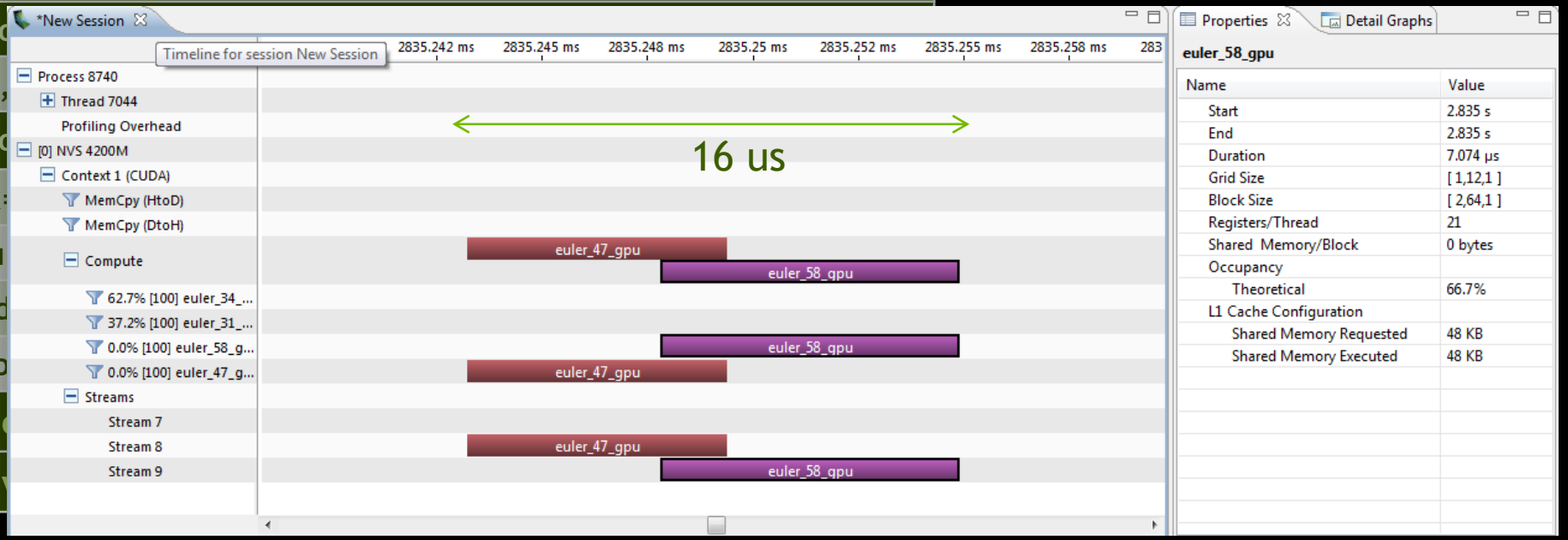
Loop scheduling via `vector` clause

```
!$acc kernels async(2)
!$acc loop vector (64)
do y=1, ny/4
!$acc loop vector (2)
   do x=1, 2
      u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
  end do
 end do
!$acc end kernels
!$acc wait
```

```
56, Loop is parallelizable
    58, Loop is parallelizable
        Accelerator kernel generated
        56, !$acc loop gang, vector(64) ! blockidx%y threadidx%y
        58, !$acc loop gang, vector(2) ! blockidx%x threadidx%x
```

Manual loop scheduling accelerates y-Boundary

```
!$acc kernels async(2)
!$acc lo
do y=1,
!$acc lo
do x=
u
end d
end do
!$acc
!$acc
```




```
56, Loop is parallelizable
    58, Loop is parallelizable
        Accelerator kernel generated
        56, !$acc loop gang, vector(64) ! blockidx%y threadidx%y
        58, !$acc loop gang, vector(2) ! blockidx%x threadidx%x
```

Can we achieve better overlap?

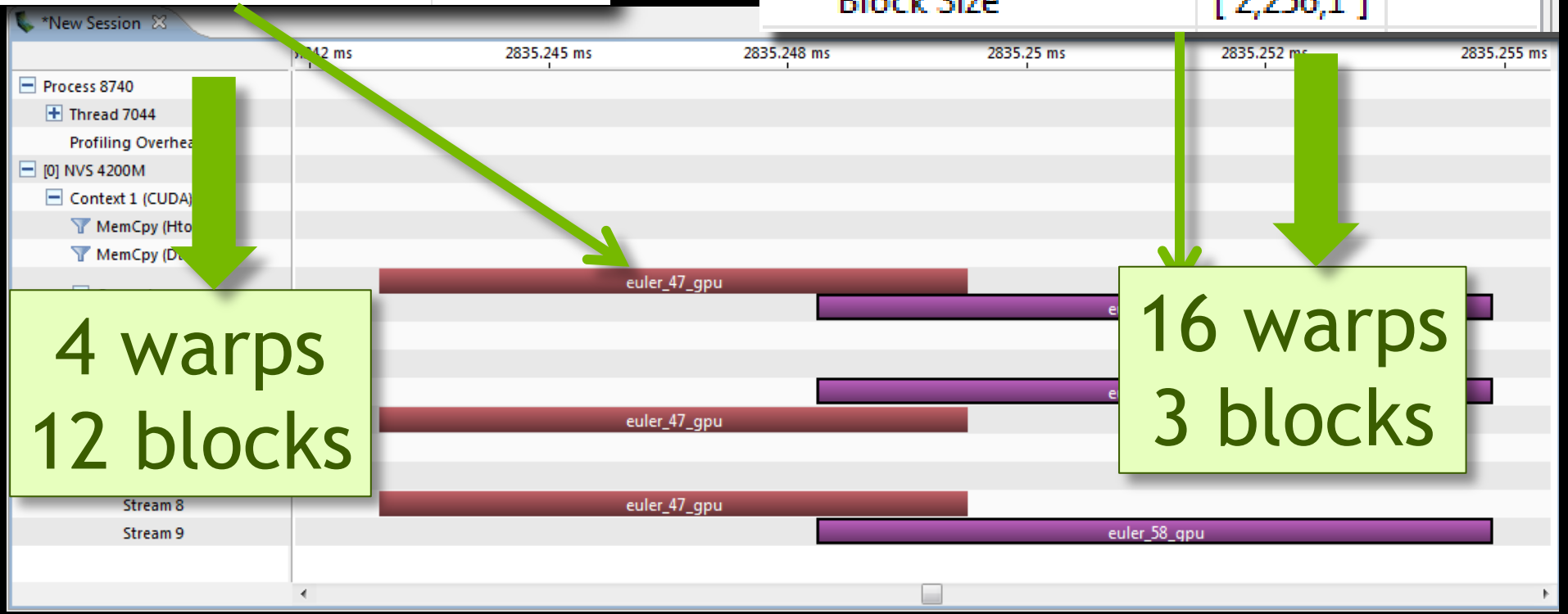
- Overlap requires sufficient resource for both kernels
- Our GPU: 1 SM, 8 Blocks, 48 warps
- What resources do the kernels use?

1 Fermi SM; 8 blocks; 48 warps

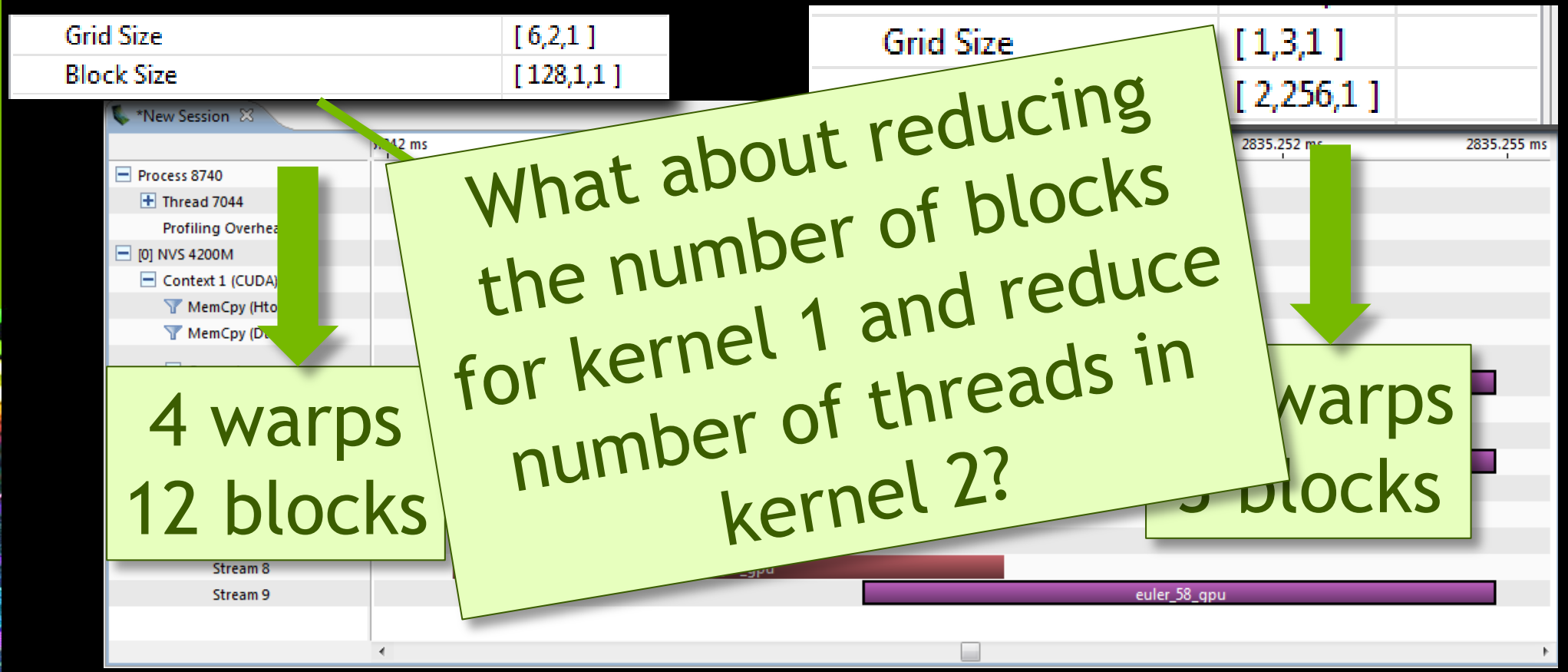
Can we achieve better overlap?

Grid Size	[6,2,1]
Block Size	[128,1,1]

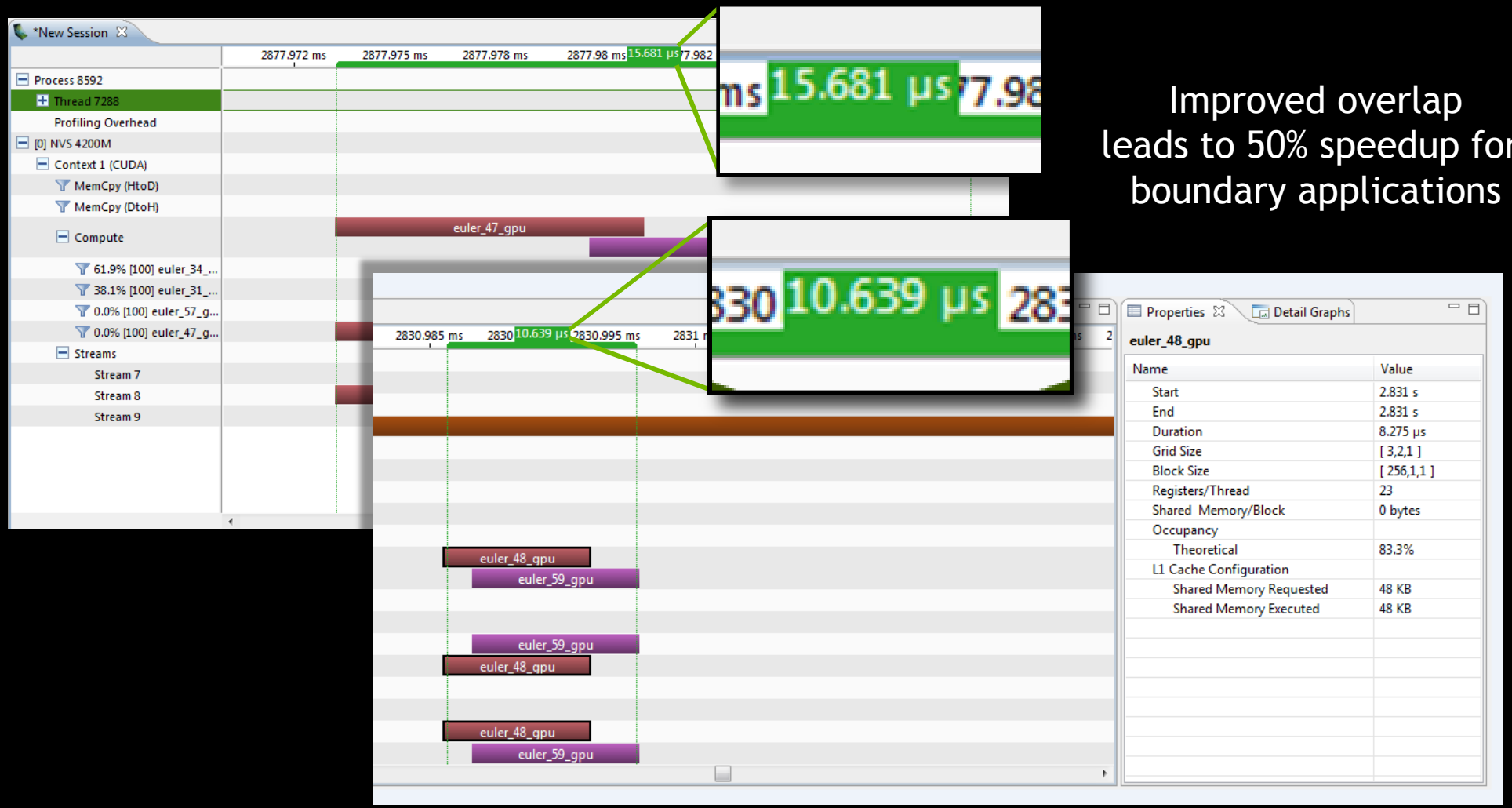
Grid Size	[1,3,1]
Block Size	[2,256,1]



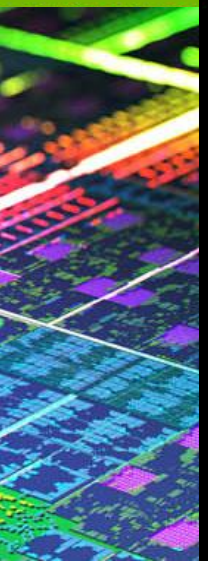
Can we achieve better overlap?



Can we achieve better overlap?



Improved overlap leads to 50% speedup for boundary applications



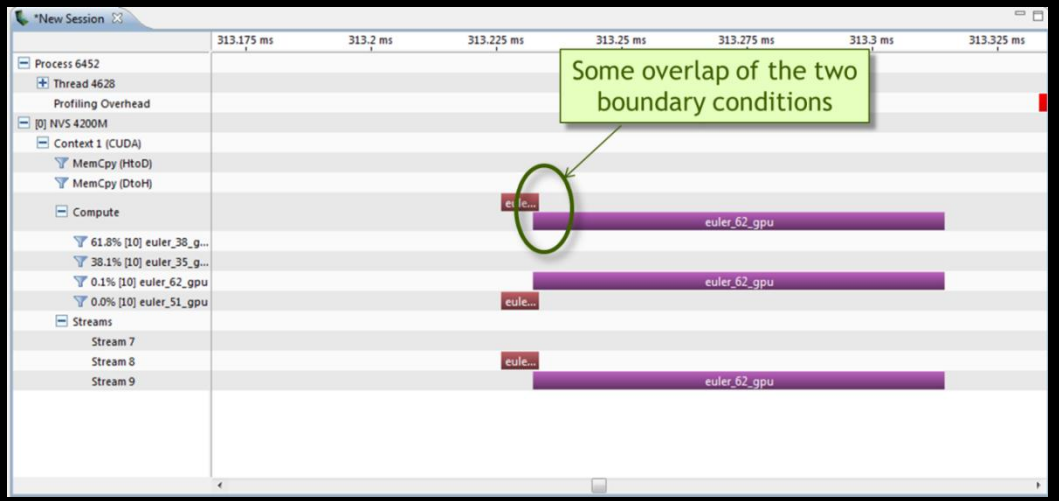
Some Comments on Manual Scheduling

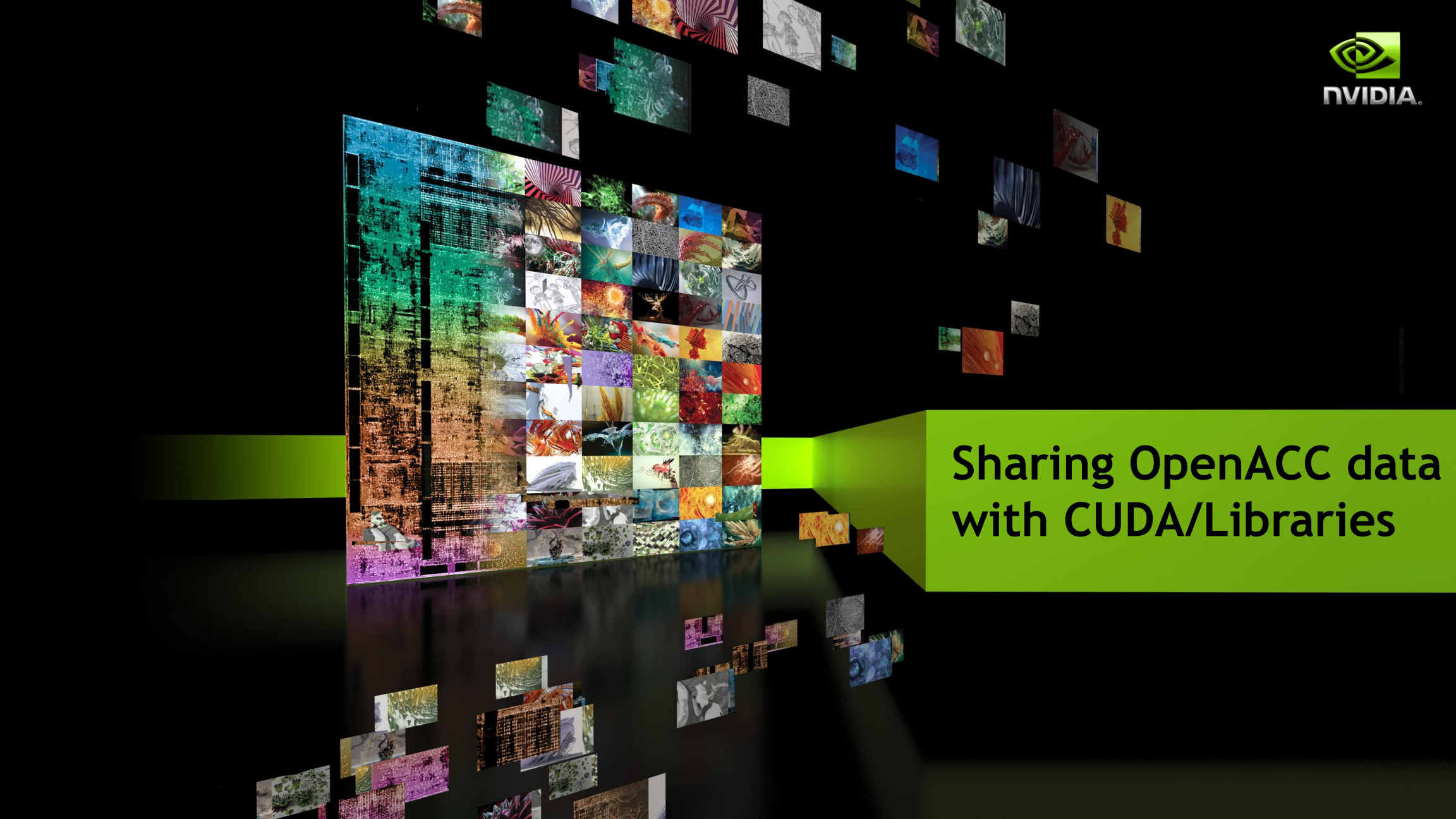
- Wide range of mechanisms to influence scheduling
  - Or num_gangs(), num_workers(), vector_length(),..
- Impact is compiler dependent
  - Standard leaves mapping up to the compiler
- Only for fine-tuning
  - Use only where needed!
- Also look at Independent clause, Collapse clause

Summary of Optimization Steps

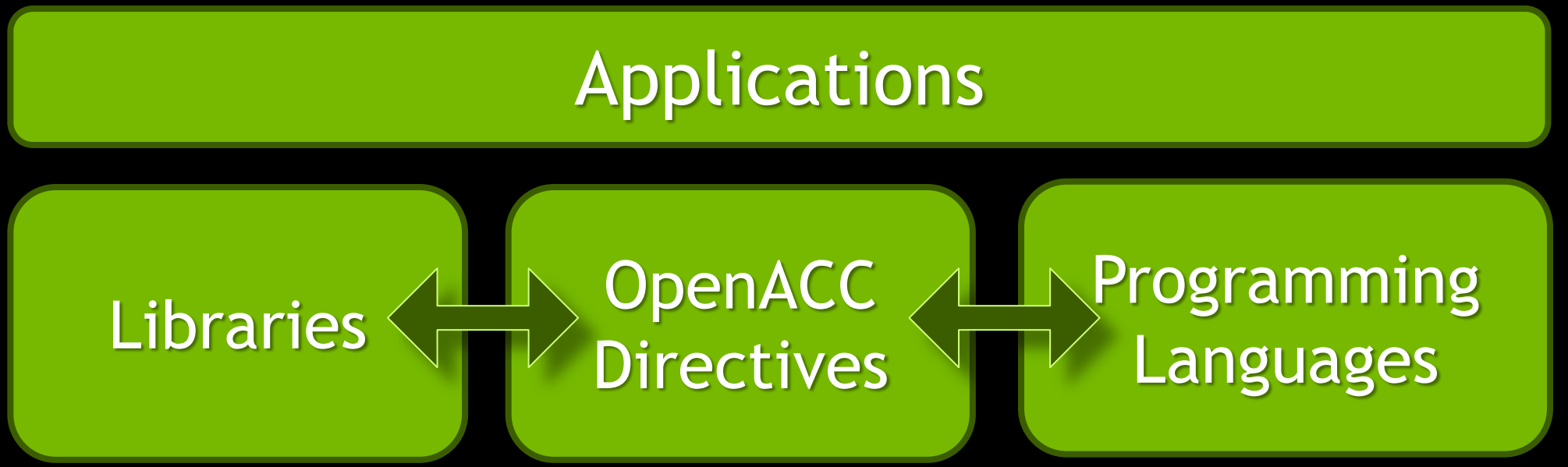
- Minimize data transfer (update)
- Optimize overlap (async)
- Manual scheduling (gang, worker, vector)

- Use profiler
- Use compiler feedback




NVIDIA
Sharing OpenACC data
with CUDA/Libraries

Interfacing OpenACC with Libraries, CUDA



"Drop-in" Acceleration

Easily Accelerate Applications

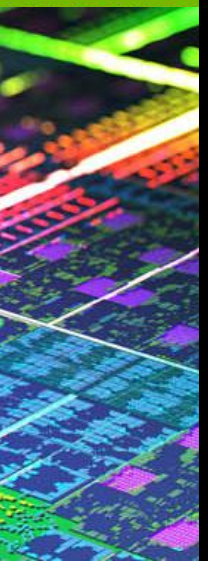
Maximum Flexibility

host_data use_device(): Obtain OpenACC Device Pointers

- Use data managed by OpenACC in libraries or CUDA code

```
!$acc data copy(a)
..
!$acc host_data use_device(a)
call myfunction(a)
!$acc end host_data
..
!$acc end data
```

From here on, host uses the device pointer for variable a



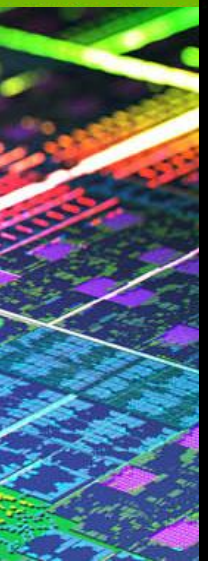
A different Approach to Obtain a Device Pointer

```
use isc_c_binding
real*8, dimension(1024) :: a
integer*8, dimension(1) :: a_ptr

!$acc data copy(a)
!$acc kernel copyout(a_ptr)
    .. do something with a ..
   a_ptr(1) = c_ptr(a(1))
!$acc end kernel
call myfunction(a_ptr(1))
!$acc end data
```

Return a_ptr to host

Store the address of a(1)

Use a_ptr as device pointer



Inverse to host_data: deviceptr clause

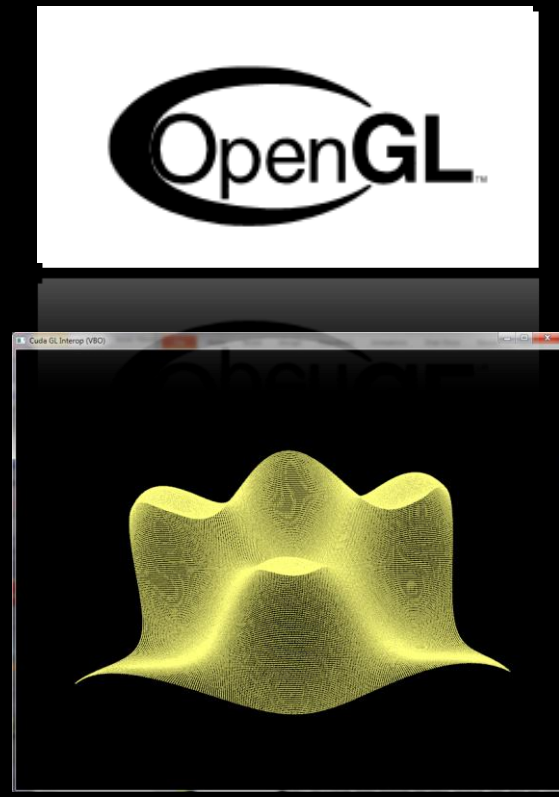
- Inform OpenACC that you have taken care of the device allocation

```
cudaMalloc(&myvar, N)

#pragma acc kernels deviceptr(myvar)
for(int i=0; i<1000;i++){
  myvar[i] = ..
}
```

- Useful when memory is managed by outside instance

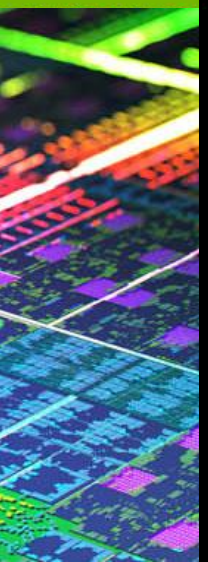
`deviceptr` example: Interop with OpenGL

- OpenGL: Programmers interface to Graphics Hardware
- API to specify
  - Primitives: points, lines, polygons
  - Properties: colors, lighting, textures, ..
  - View: camera position and perspective
- Manages data on GPU



See e.g. “What Every CUDA Programmer Should Know About OpenGL”
(http://www.nvidia.com/content/GTC/documents/1055_GTC09.pdf)

Basic components of an OpenGL application



Create OpenGL buffer

Init()

`glutMainLoop()`

GLUT (UI, EventLoop)

Modify buffer

`glDrawElements()`

Display()

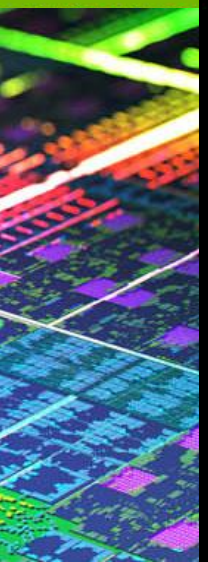
GPU Memory

Managed by OpenGL

vbo

Managed by CUDA

OpenGL Buffer Modification with OpenACC



Create OpenGL buffer
Inform CUDA about buffer

Init()

```
glutMainLoop()
```

GLUT (UI, EventLoop)

Map OpenGL buffer to CUDA

```
#pragma acc kernels deviceptr(vboPtr)
```

Modify buffer in OpenACC
Return control over buffer back to OpenGL

```
glDrawElements()
```

Display()

GPU Memory

Managed by OpenGL

vbo

Managed by CUDA

Initialization

```
GLuint vbo;

glGenBuffers(1, &vbo);
glBindBuffer(GL_ARRAY_BUFFER, vbo);

unsigned int size = mesh_size * 4 * sizeof(float);
glBufferData(GL_ARRAY_BUFFER, size, 0, GL_DYNAMIC_DRAW);
glBindBuffer(GL_ARRAY_BUFFER, 0);

struct cudaGraphicsResource* cuda_vbo_resource;
cudaGraphicsGLRegisterBuffer(&cuda_vbo_resource, vbo, vbo_res_flags);
```

Create buffer object

Initialize buffer object

Register buffer with CUDA

Display

```
cudaGraphicsMapResources(1, &cuda_vbo_resource, 0);
float *dptr; size_t num_bytes;
cudaGraphicsResourceGetMappedPointer((void **)&dptr,
                &num_bytes, cuda_vbo_resource);
#pragma acc data deviceptr(dptr)
{
  #pragma acc kernels loop independent
   for(int x=0; x<mesh_width; x++) {
       dptr[x] = ..
}
cudaGraphicsUnmapResources(1, &cuda_vbo_resource, 0);

glBindBuffer(GL_BUFFER_ARRAY, vbo);
glDrawArrays(GL_POINTS,..);
```

Map buffer to CUDA space

Obtain device pointer

Inform OpenACC about buffer and use it

Hand buffer back to OpenGL

Draw the points

Maintain OpenACC state

Create OpenGL buffer

Inform CUDA about buffer

```
#pragma acc data copy(u,v)
```

```
glutMainLoop()
```

Init()

GLUT (UI, EventLoop)

Map OpenGL buffer to CUDA

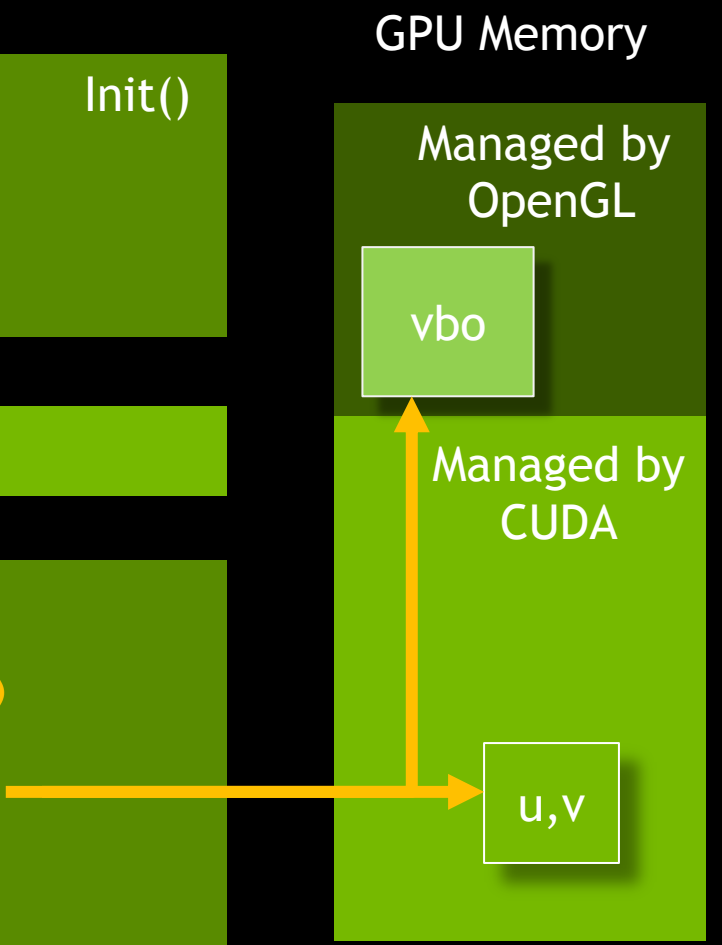
```
#pragma acc kernels deviceptr(vboPtr)
    present(u,v)
```

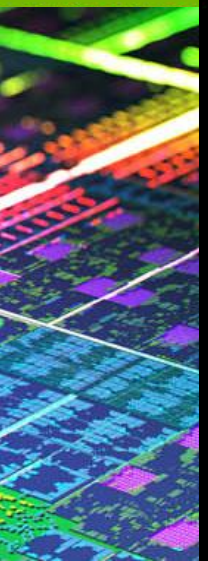
Modify buffer, variables in OpenACC

Return control over buffer back to OpenGL

```
glDrawElements()
```

Display()





Summary

- Optimizing data locality is key to OpenACC performance
  - Data regions, update directive
  - Use profiler and compiler feedback
- Further tuning possible
  - Async clause, scheduling clauses
- OpenACC integrates well into GPU ecosystem
  - host_data use_deivce
  - deviceptr
  - (in-situ viz/analysis in OpenGL/OpenACC)


NVIDIA
Thank you!
See you at GTC2014

Some Relevant Interop API Calls

Create OpenGL Buffer — Init()

```
cudaGraphicsGLRegisterBuffer(cudaVbo, vbo)
#pragma acc data copy(u,v)
glutMainLoop()
```

GLUT (UI, EventLoop)

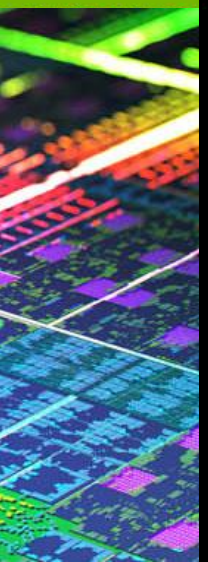
```
cudaGraphicsMapResource(cudaVbo)
#pragma acc kernels deviceptr(vboPtr)
    present(u,v)
```

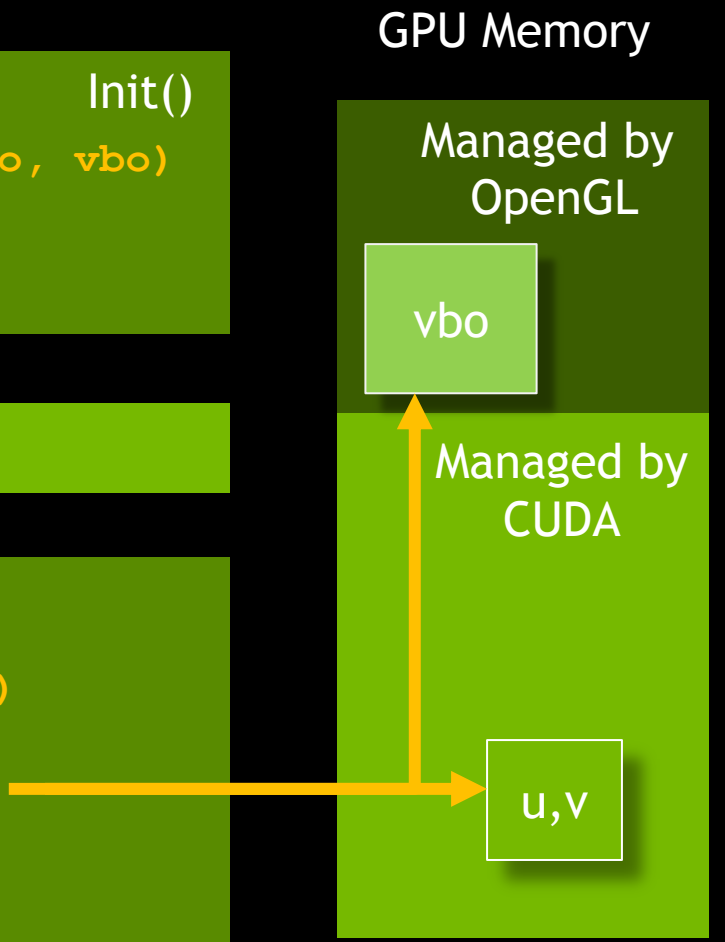
Modify buffer in OpenACC

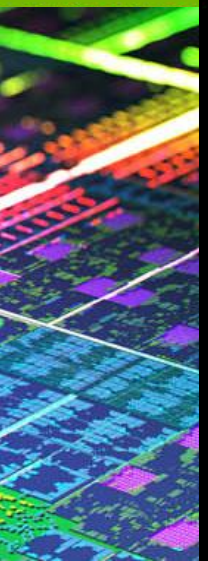
```
cudaGraphicsUnmapResource()
```

```
glDrawElements()
```

Display()



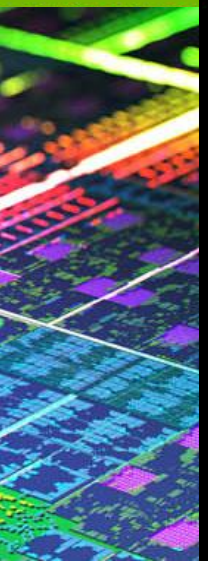




Bullet/Content Slide Title Here

- Try to limit the number of slides you use
- Keep text to a minimum
  - Instead, speak more to your audience (engage them with anecdotes/enthusiasm/eye contact)
  - Try not to read your points verbatim; bullet points should be used for key points only
  - Use images/graphics to help convey your message

This slide will be removed prior to presentation

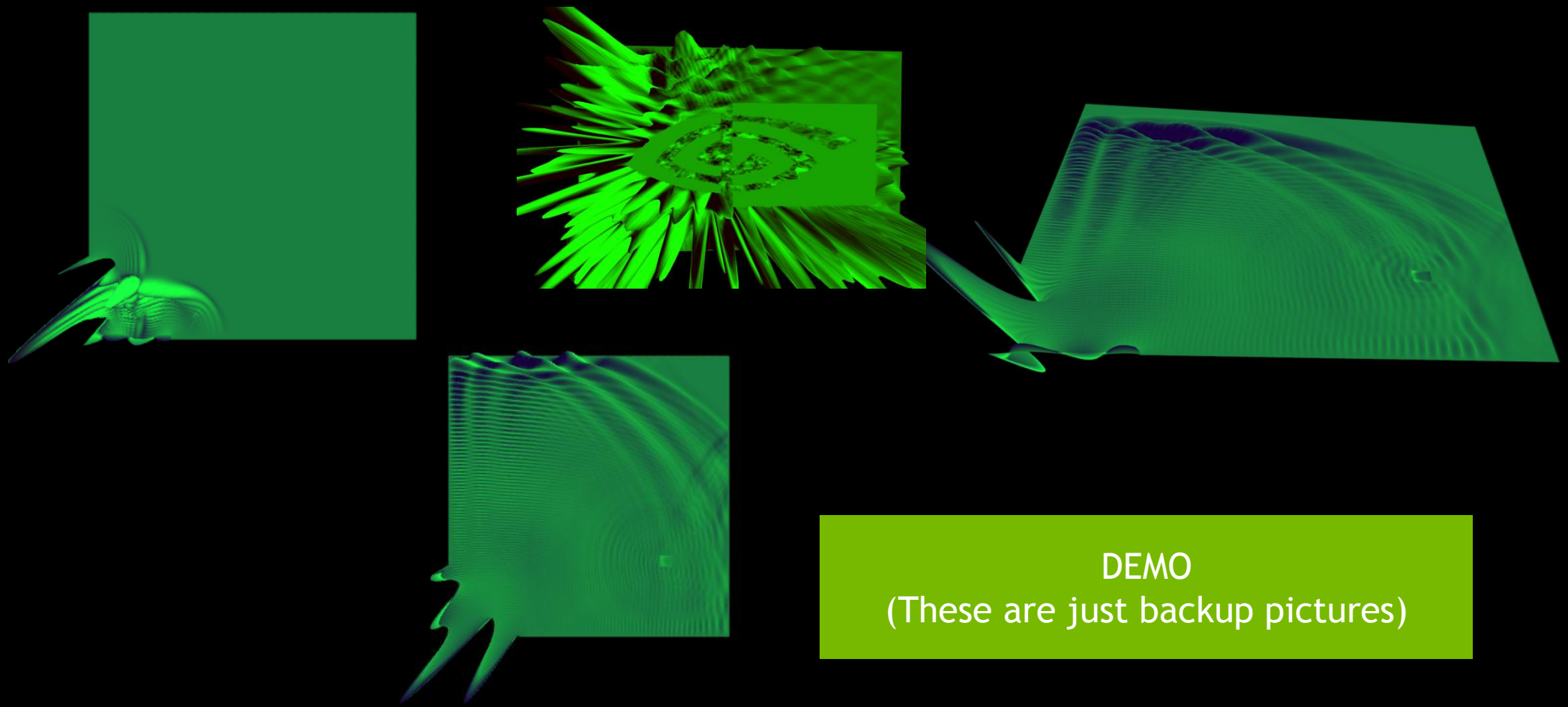


(Abstract from the agenda, just for reference)

OpenACC has quickly become the standard for accelerating large code bases with GPUs. Using directives, the programmer provides hints about data locality, data dependency and control flow that allows the compiler to automatically generate efficient GPU code. While the OpenACC model is well suited for a broad range of commonly encountered software patterns, it is sometimes necessary to fine-tune an application with advanced OpenACC directives or interface to an external CUDA code to take advantage of latest hardware features. The goal of this tutorial is to present the different strategies to tune OpenACC code and introduce mechanisms to interface OpenACC with other GPU code. Based on examples, we will first present different strategies to assess and optimize the performance of an OpenACC code, and will then focus on interfacing OpenACC code with CUDA and graphics libraries.

This slide will be removed prior to presentation

OpenACC interop with OpenGL: One step towards in-situ visualization..



DEMO
(These are just backup pictures)