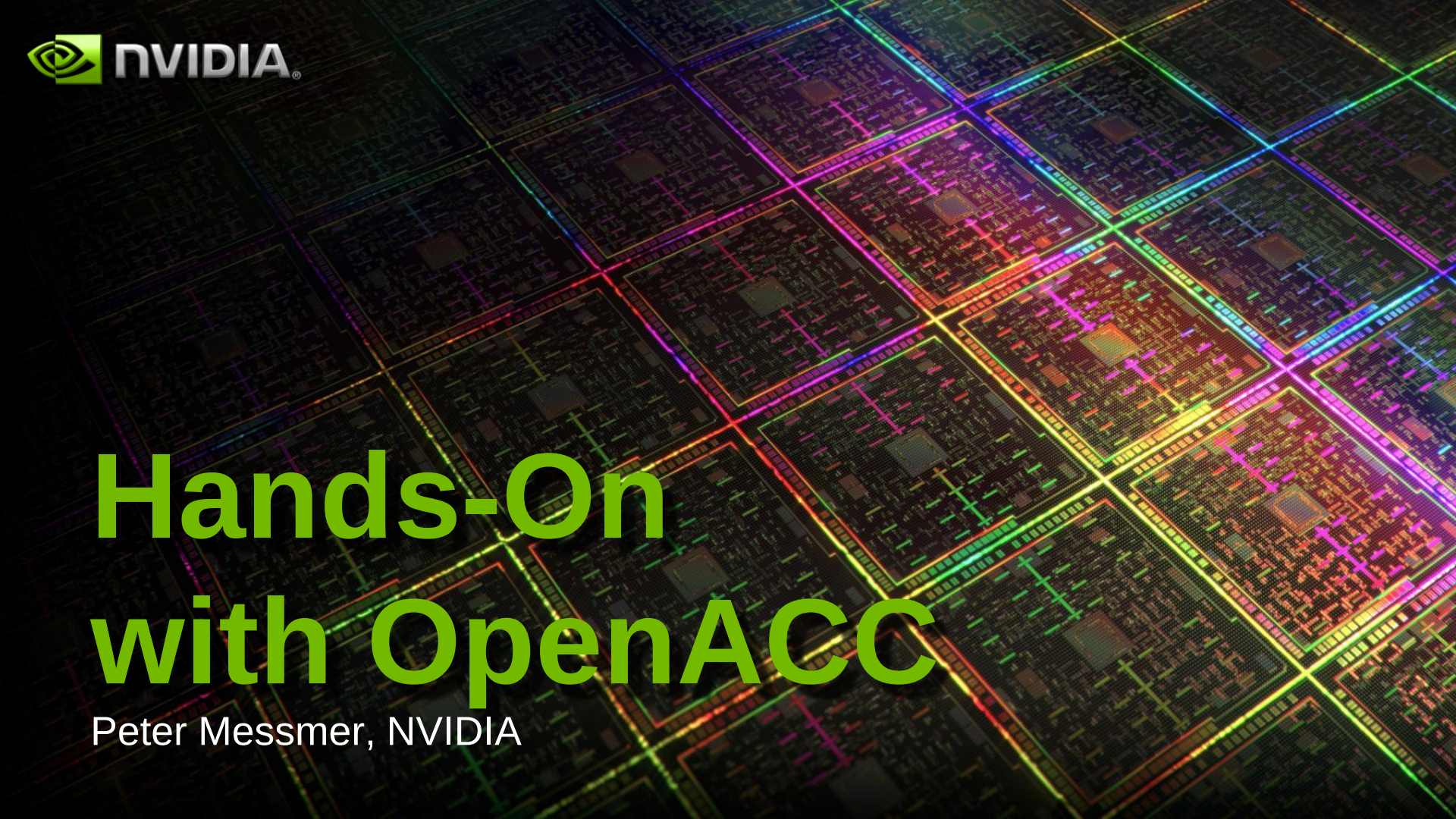


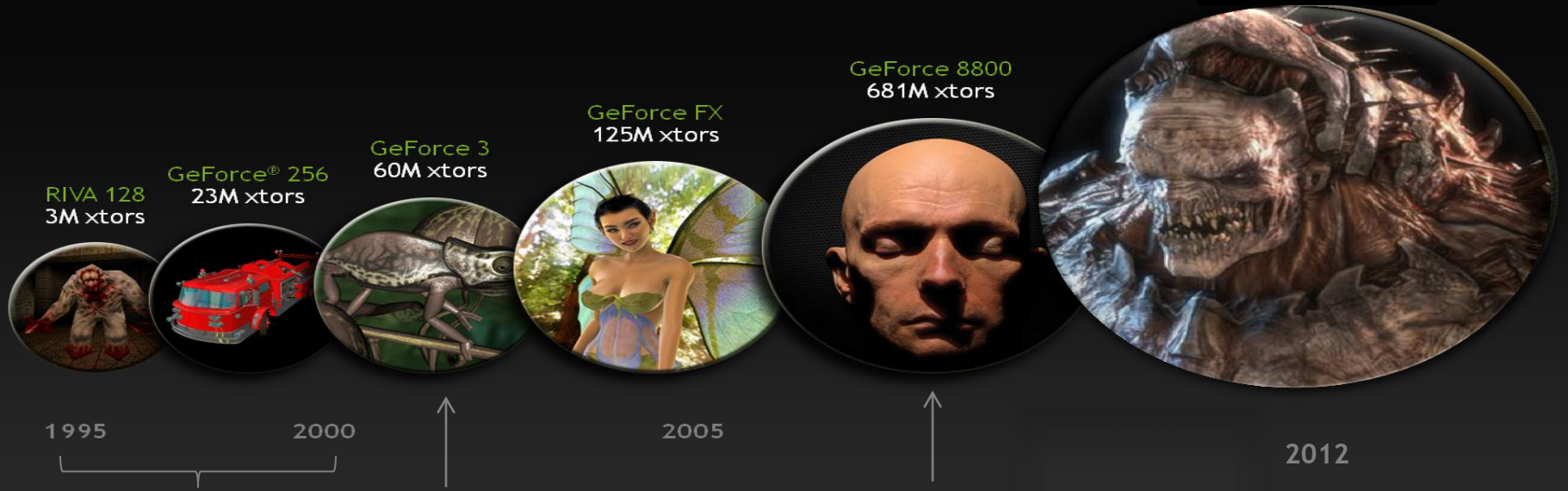
Hands-On with OpenACC

Peter Messmer, NVIDIA

Agenda

2:00 – 3:00	OpenACC Intro & Lab setup
3:00 – 4:00	OpenACC by Example
4:00 – 4:30	Break
4:30 – 6:00	Optimization and Libraries Interoperability

Evolution of GPUs

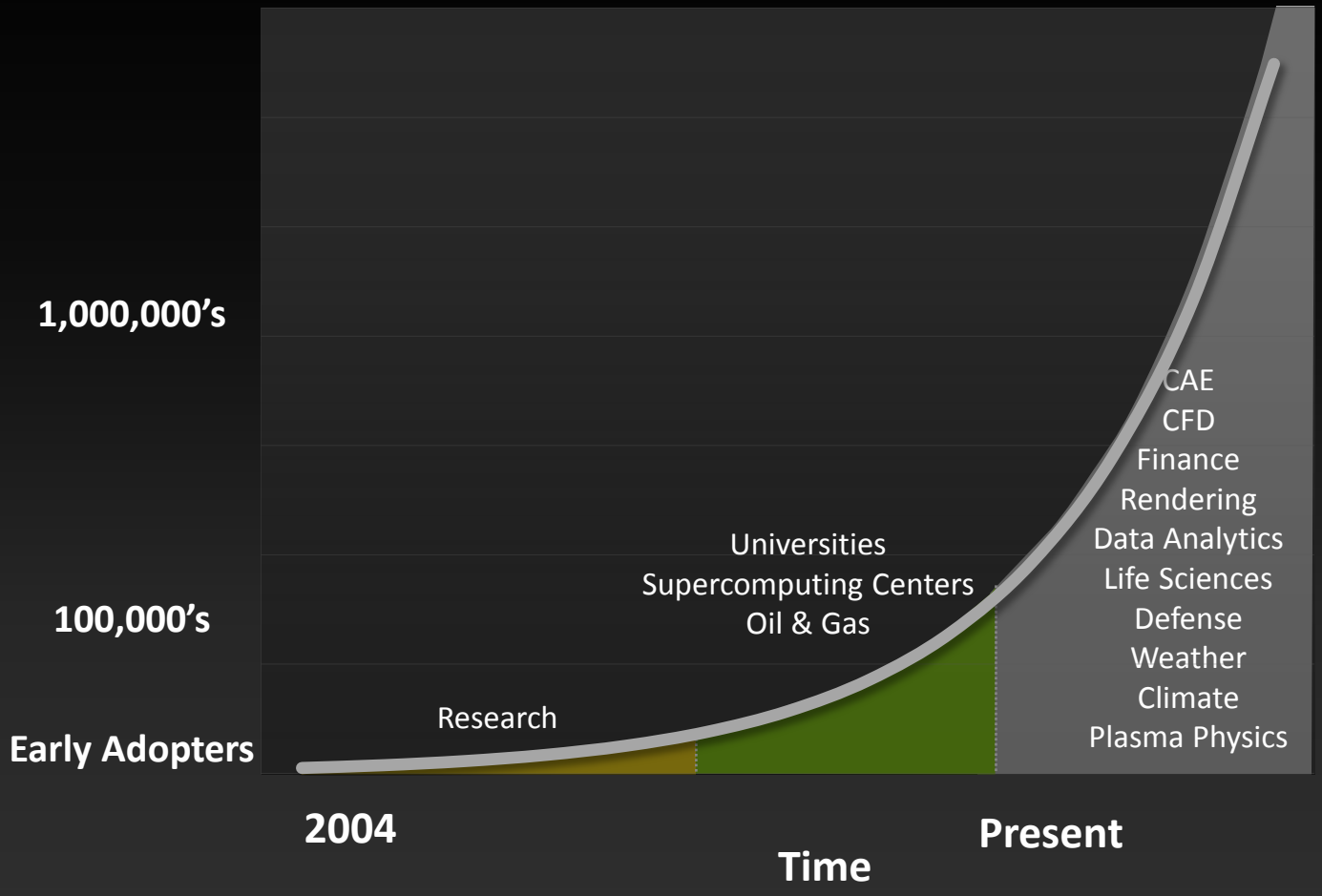


Fixed function

Programmable pixel and vertex shaders
Geforce 3 series
2001

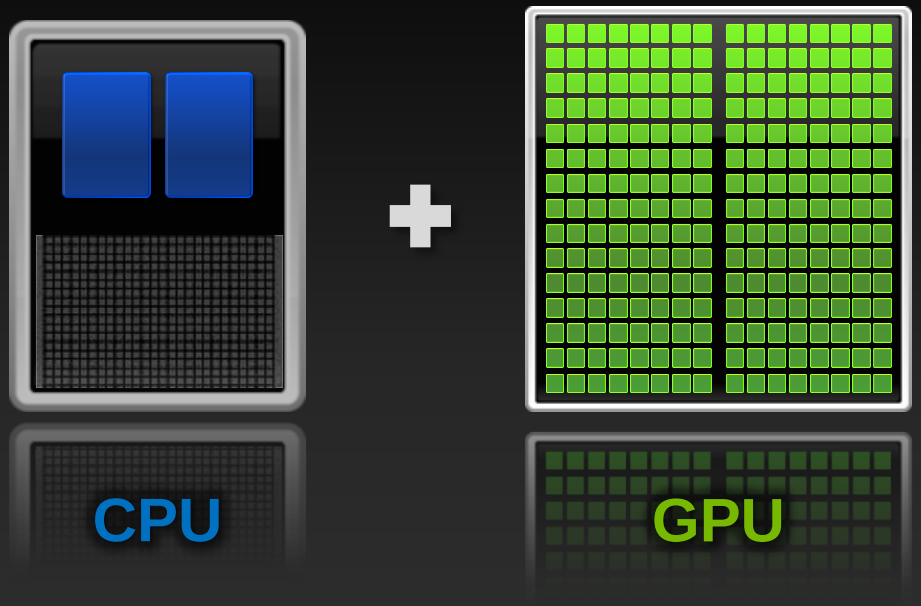
CUDA capable
Geforce 8 series
2006

GPUs Reaching Broader Set of Developers



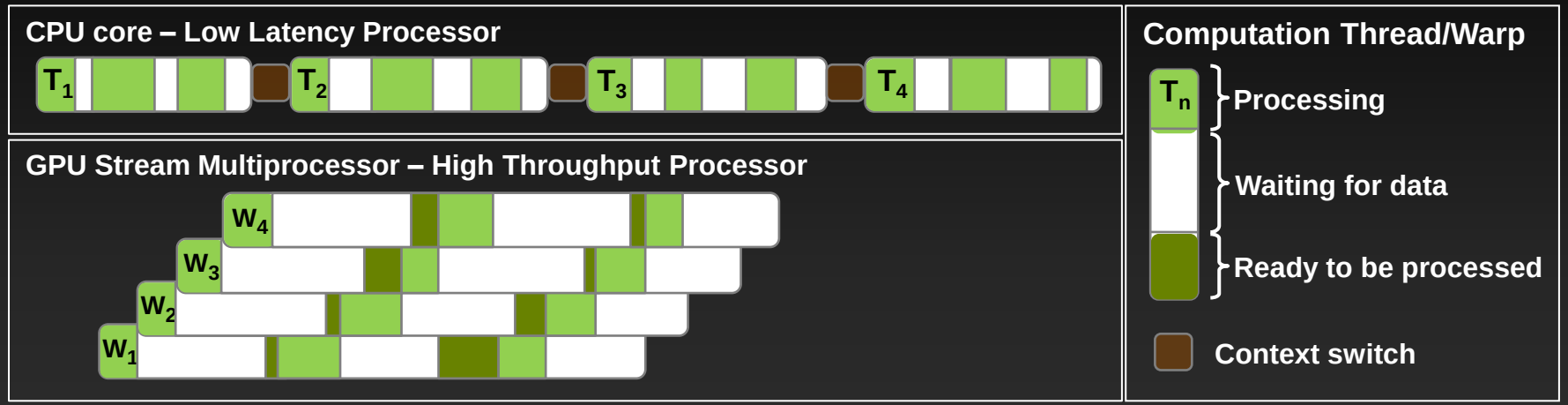
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor



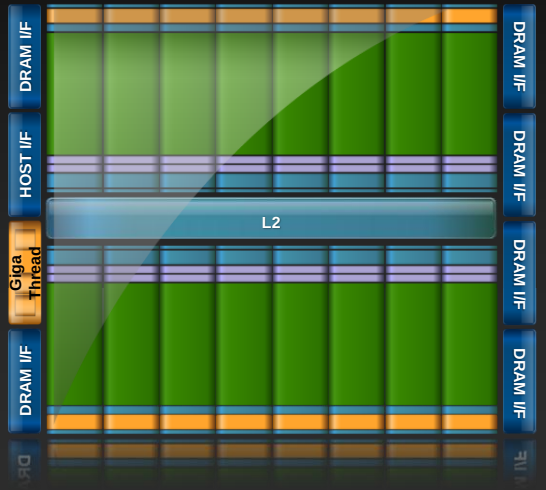
Low Latency or High Throughput?

- CPU architecture must minimize latency within each thread
- GPU architecture hides latency with computation from other thread warps

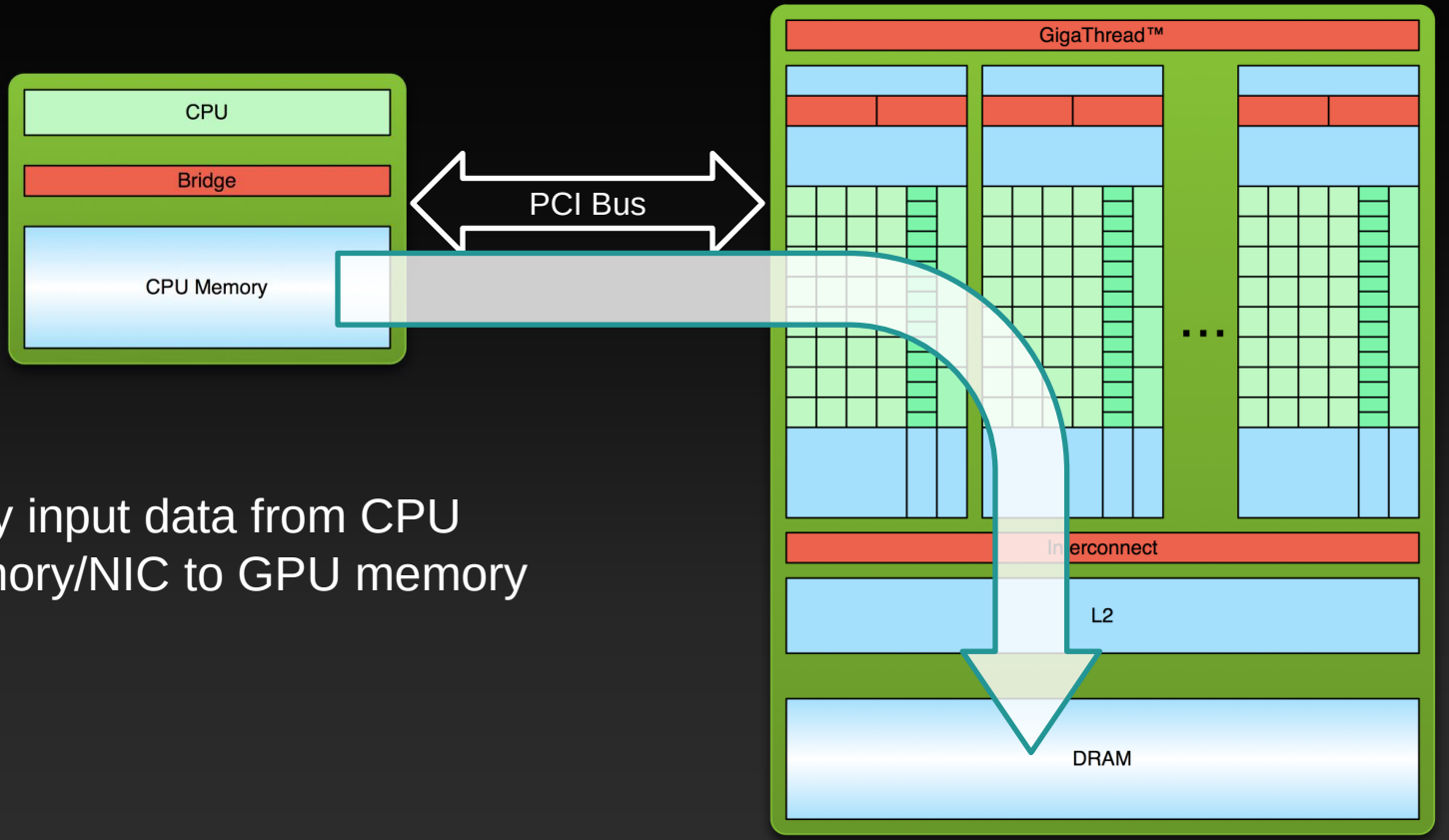


GPU Architecture: Two Main Components

- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB per GPU
  - Bandwidth currently up to ~250 GB/s (Tesla products)
  - ECC on/off (Quadro and Tesla products)
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches

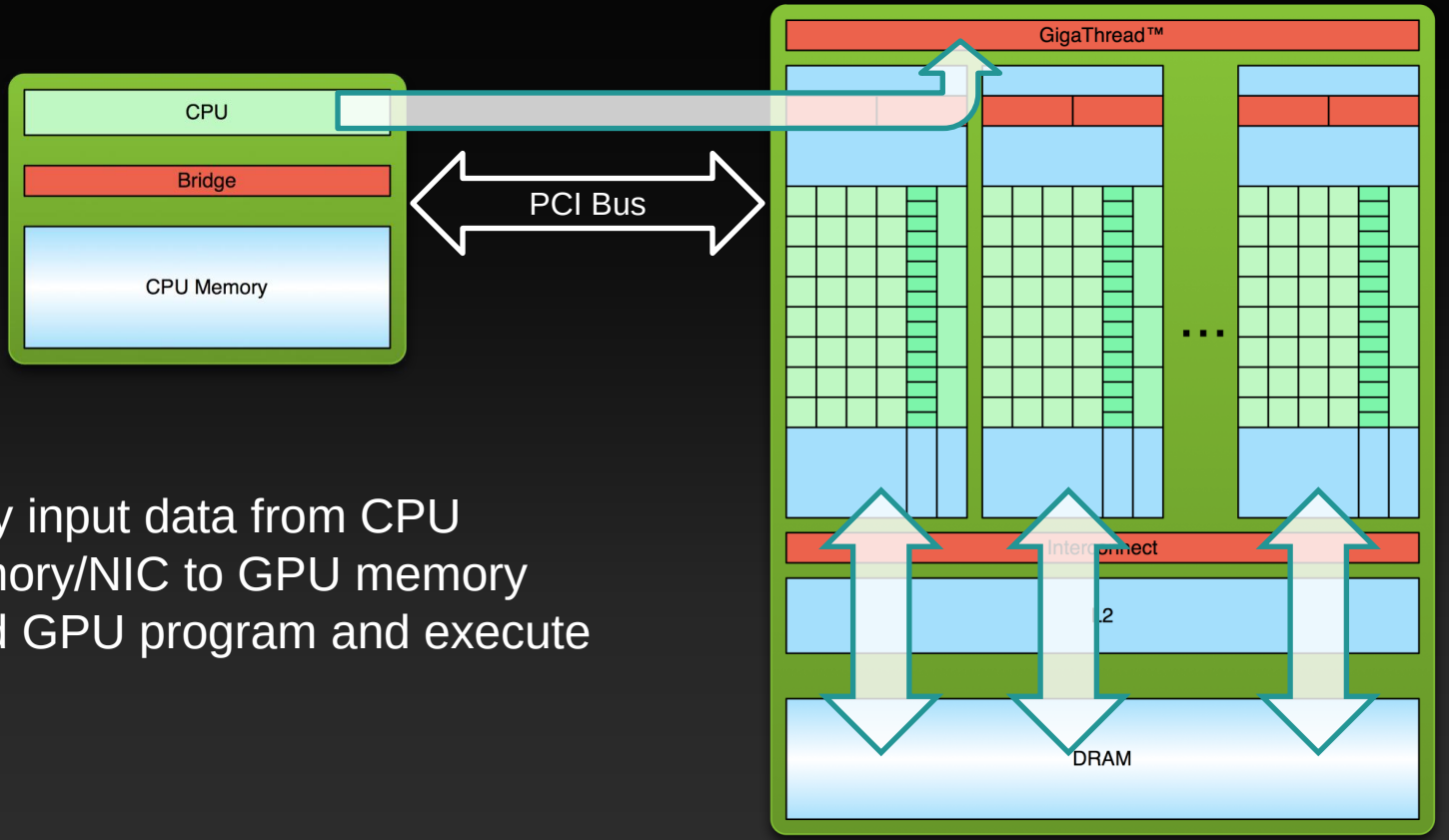


Simple Processing Flow



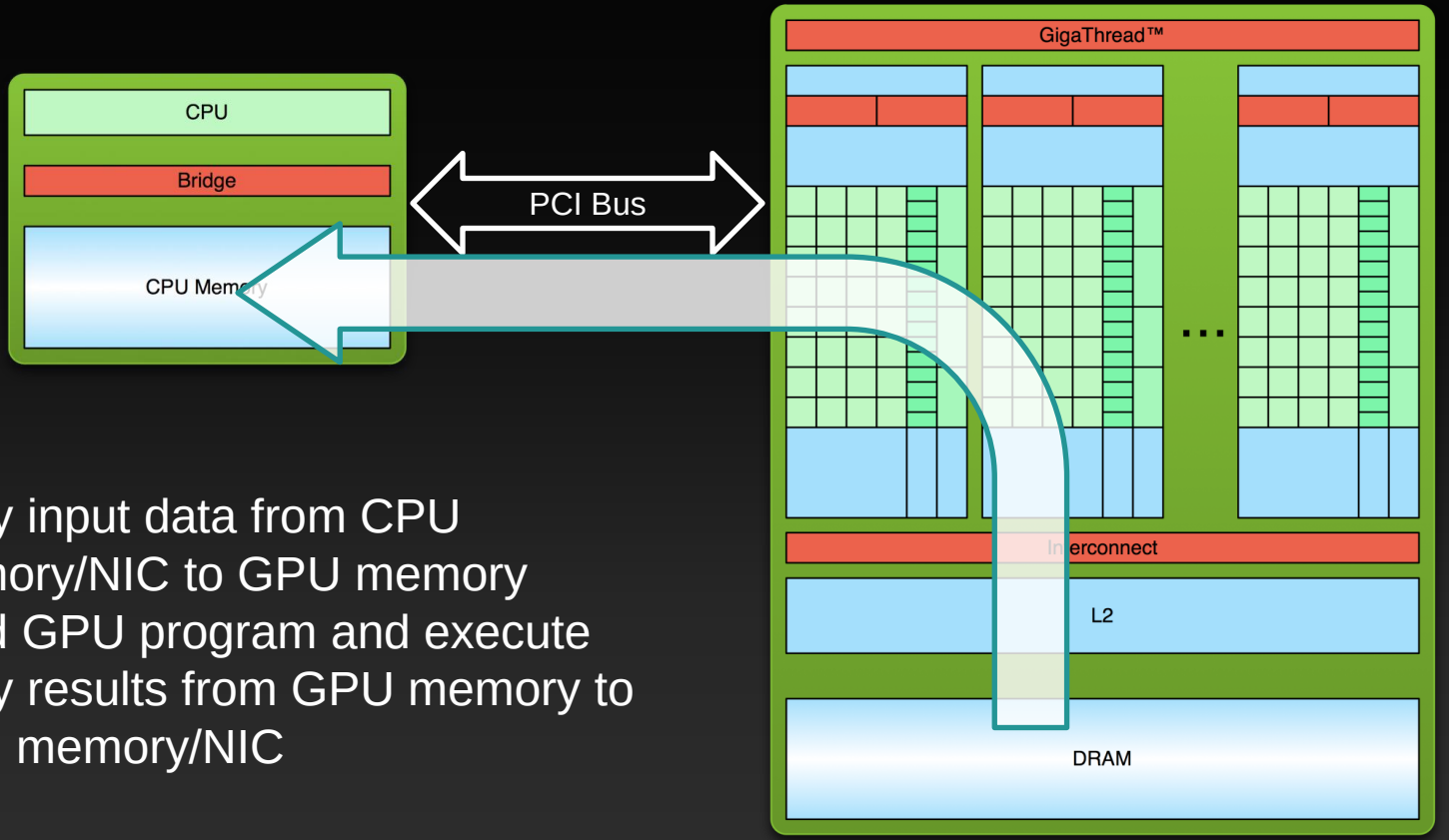
1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute

Simple Processing Flow

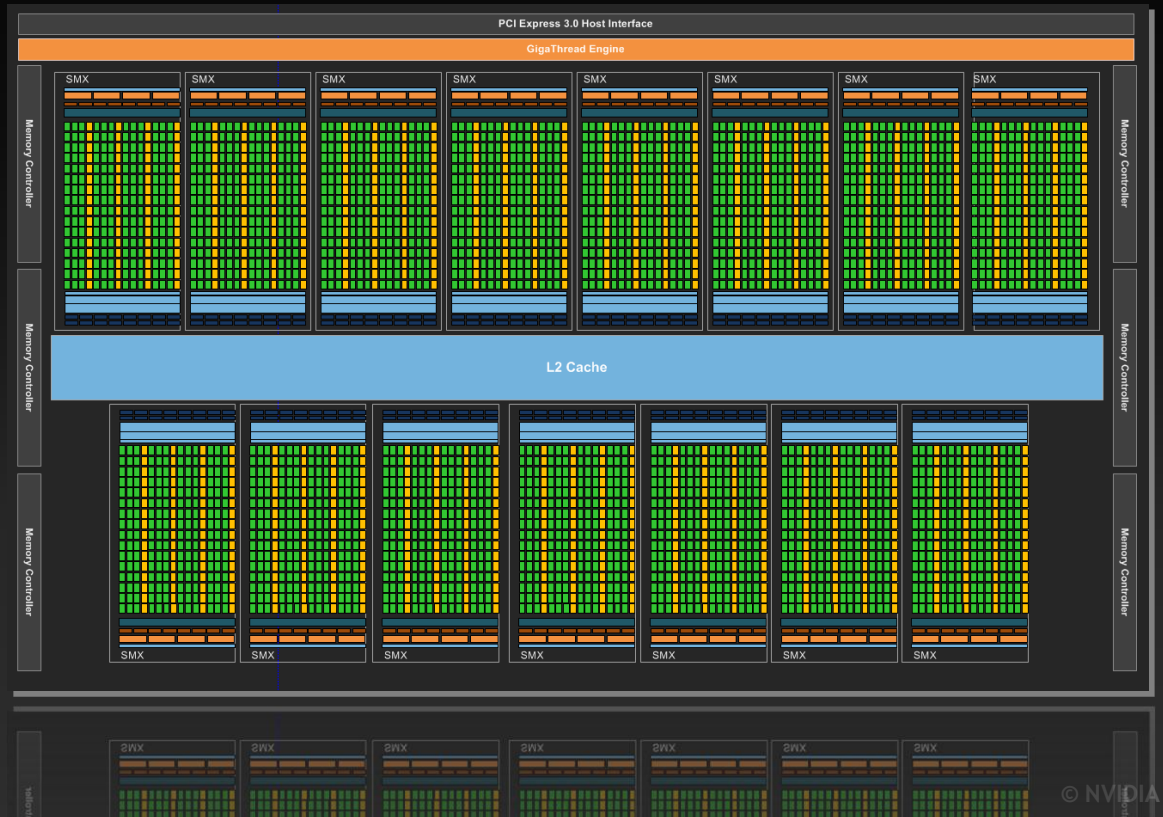


1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

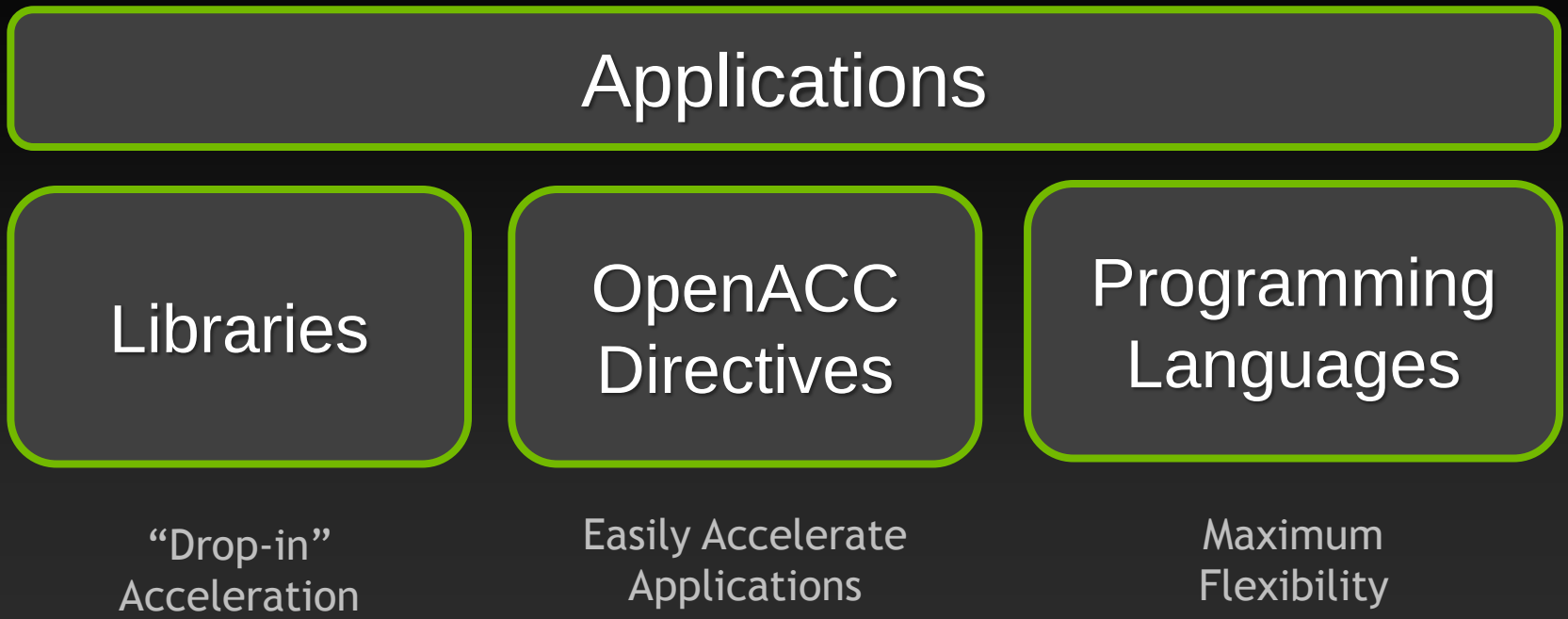
Kepler GK110 Block Diagram

Architecture

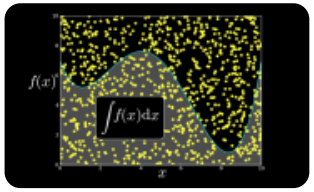
- 7.1B Transistors
- 15 SMX units
- > 1 TFLOP FP64
- 1.5 MB L2 Cache
- 384-bit GDDR5



3 Ways to Accelerate Applications



NVIDIA cuBLAS



NVIDIA cuRAND

NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal
Image Processing

CULA tools

GPU Accelerated
Linear Algebra

MAGMA

Matrix Algebra on
GPU and Multicore

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

CUSP

Sparse Linear
Algebra

libjacket

Building-block
Algorithms for CUDA

Thrust

C++ STL Features
for CUDA

GPU Accelerated Libraries

"Drop-in" Acceleration for Your Applications

CUDA Math Libraries

High performance math routines for your applications:

- **cuFFT – Fast Fourier Transforms Library**
- **cuBLAS – Complete BLAS Library**
- **cuSPARSE – Sparse Matrix Library**
- **cuRAND – Random Number Generation (RNG) Library**
- **NPP – Performance Primitives for Image & Video Processing**
- **Thrust – Templated C++ Parallel Algorithms & Data Structures**
- **math.h - C99 floating-point Library**

Included in the CUDA Toolkit

Free download @ www.nvidia.com/getcuda

CUDA for C : C with a few keywords

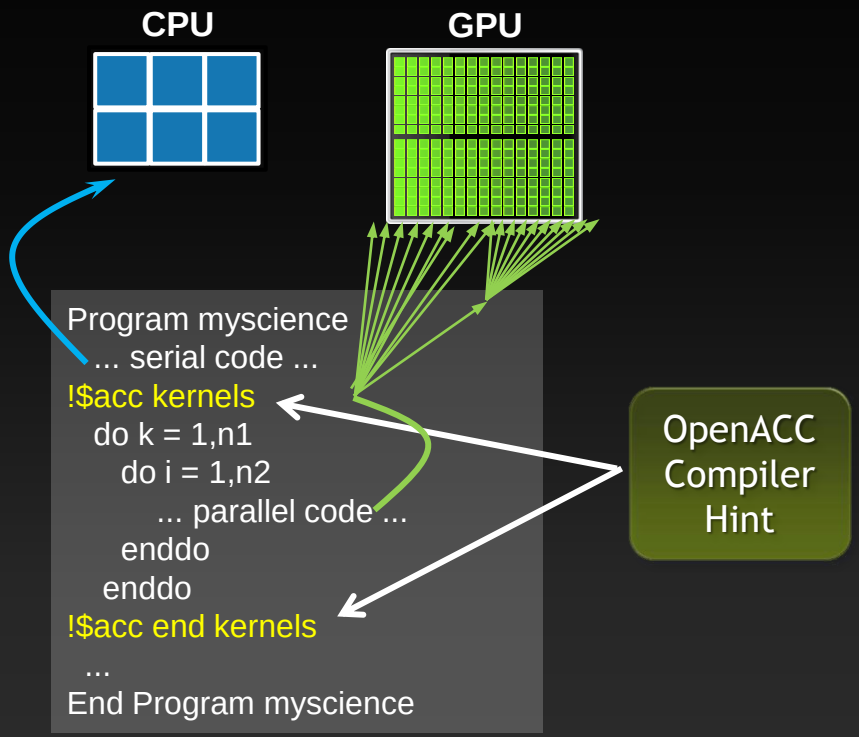
```
void saxpy_serial(int n, float a, float *x, float *y)
{
    for (int i = 0; i < n; ++i)
        y[i] = a*x[i] + y[i];
}
// Invoke serial SAXPY kernel
saxpy_serial(n, 2.0, x, y);
```

Standard C Code

```
__global__ void saxpy_parallel(int n, float a, float *x, float *y)
{
    int i = blockIdx.x*blockDim.x + threadIdx.x;
    if (i < n)  y[i] = a*x[i] + y[i];
}
// Invoke parallel SAXPY kernel with 256 threads/block
int nblocks = (n + 255) / 256;
saxpy_parallel<<<nblocks, 256>>>(n, 2.0, x, y);
```

Parallel C Code

OpenACC Directives



Simple Compiler hints

Compiler Parallelizes code

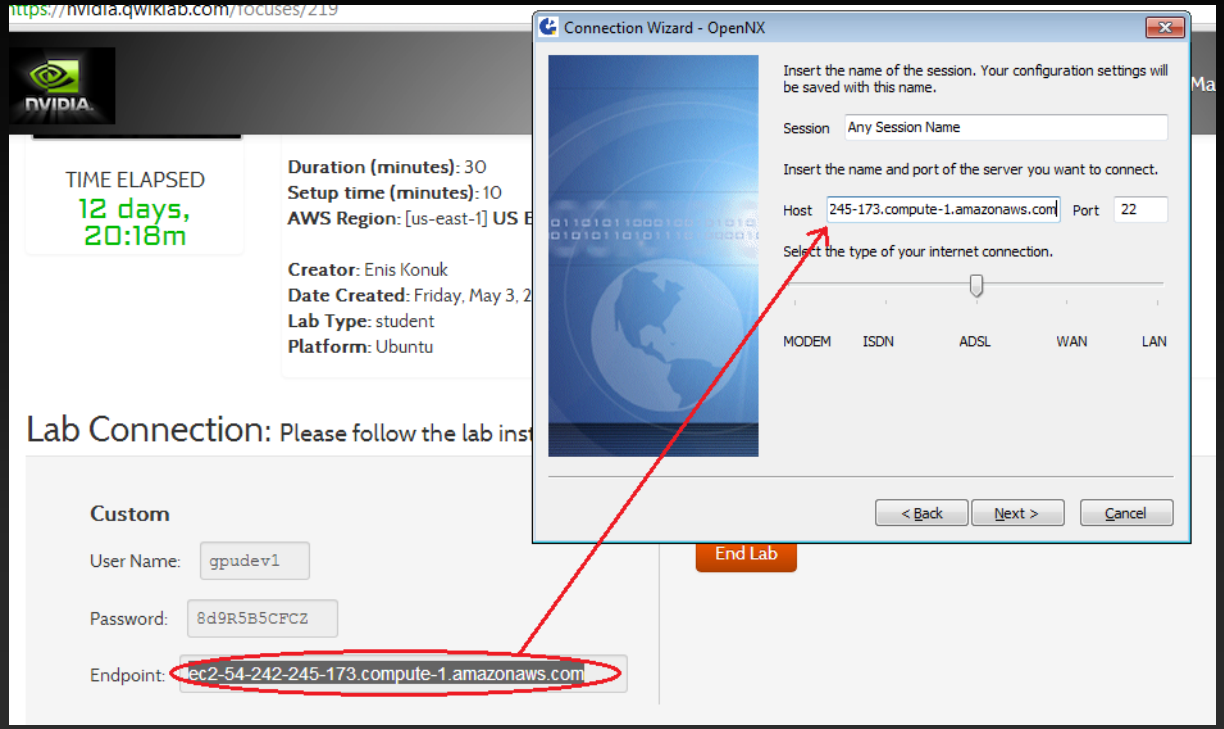
Works on many-core GPUs & multicore CPUs

Lab Setup

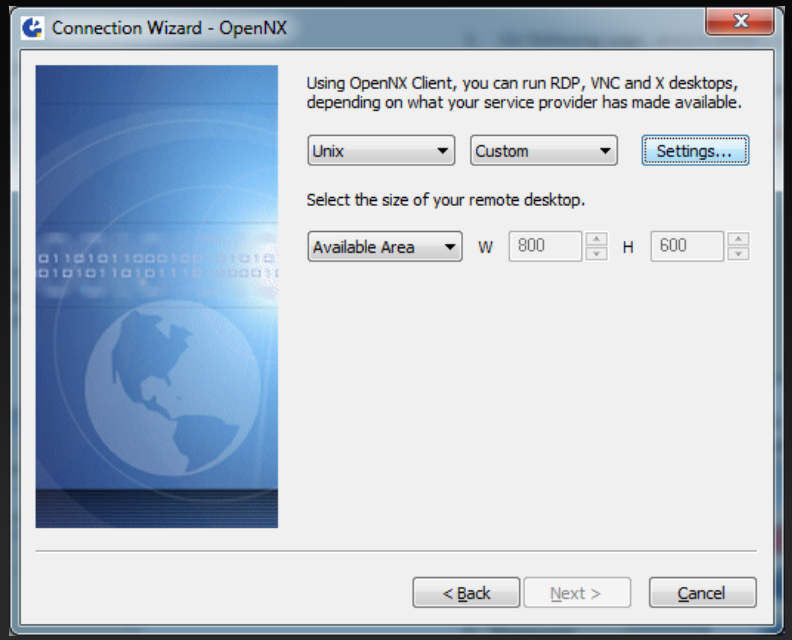
How to connect to Amazon EC2?

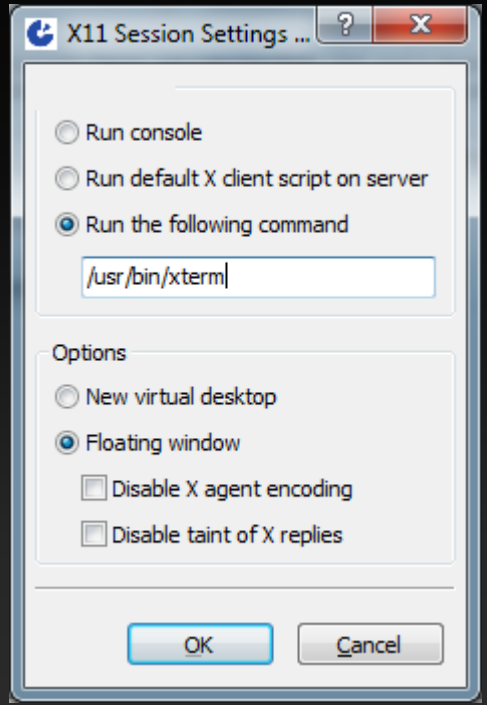
1. If you have not done already:
   - Download and install OpenNX
   - Instructions: https://developer.nvidia.com/cuda-cloud
2. Surf to https://nvidia.qwiklab.com/
   1. Sign in and select appropriate class
   2. Find the correct lab‘s listing, and once enabled press Start Lab
   3. Enter the Token number provided when prompted
3. Setting up OpenNX Client Connection

Setting up OpenNX Client Connection I (demo)

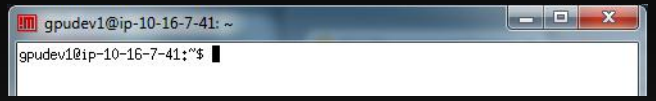


Setting up OpenNX Client Connection II (demo)

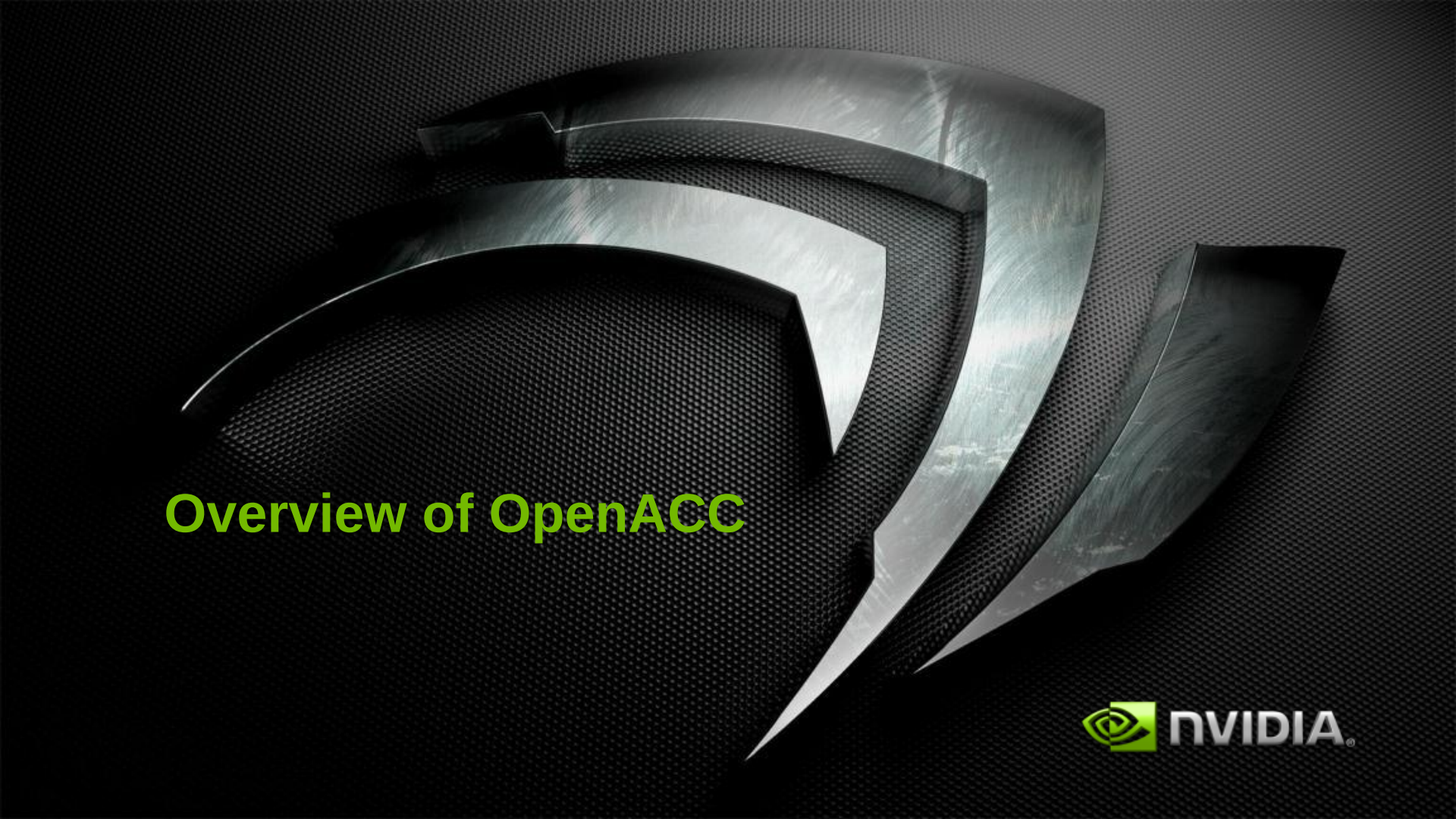




Setting up OpenNX Client Connection III (demo)



Overview of OpenACC



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

High-level

- Compiler directives to specify parallel regions in C & Fortran
  - Offload parallel regions
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization
  - Without explicit data or program transfers between host and accelerator

High-level… with low-level access

- Programming model allows programmers to start simple
- Compiler gives additional guidance
  - Loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems

5x in 40 Hours

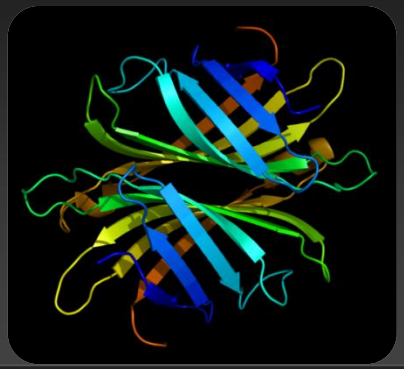
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 8 Hours

"Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications."

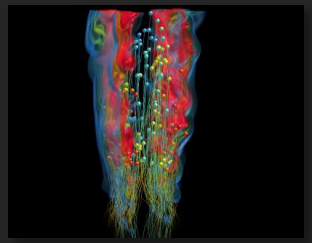
-- Developer at the Global Manufacturer of Navigation Systems

Focus on Exposing Parallelism

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

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

S3D
Research more efficient combustion with next-generation fuels



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

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

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

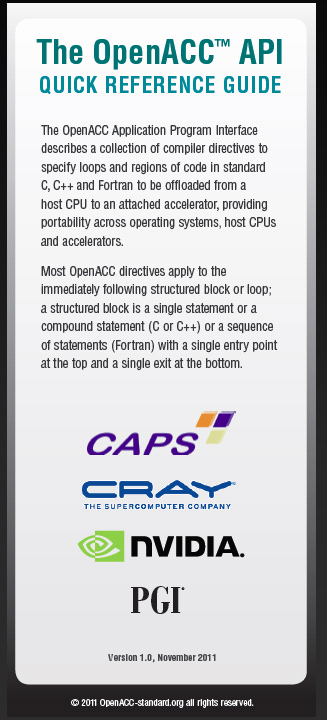
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online

www.openacc.org

- Quick reference card also available

- Compilers available now from PGI, Cray, and CAPS



SAXPY – Single precision A*X Plus Y

SAXPY in C

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

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

...
// Perform SAXPY on N elements
saxpy(N, 3.0, x, y);
...
```

SAXPY in Fortran

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

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

end subroutine saxpy


...
! Perform SAXPY on N elements
call saxpy(N, 3.0, x, y)
...
```

C tip: the `restrict` keyword

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

```
float *restrict ptr
```

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

- **Limits the effects of pointer aliasing**
- **OpenACC compilers often require `restrict` to determine independence**
  - **Otherwise the compiler can't parallelize loops that access `ptr`**
  - **Note: if programmer violates the declaration, behavior is undefined**

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

SAXPY – Single prec A*X Plus Y

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
saxpy(N, 3.0, x, y);
...
```

Support your compiler!

SAXPY in Fortran

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

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

end subroutine saxpy


...
! Perform SAXPY on N elements
call saxpy(N, 3.0, x, y)
...
```

SAXPY – Single prec A*X Plus Y in 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 N elements
saxpy(N, 3.0, x, y);
...
```

SAXPY in Fortran

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


...
! Perform SAXPY on N elements
call saxpy(N, 3.0, x, y)
...
```

SAXPY – Single prec A*X Plus Y in OpenACC

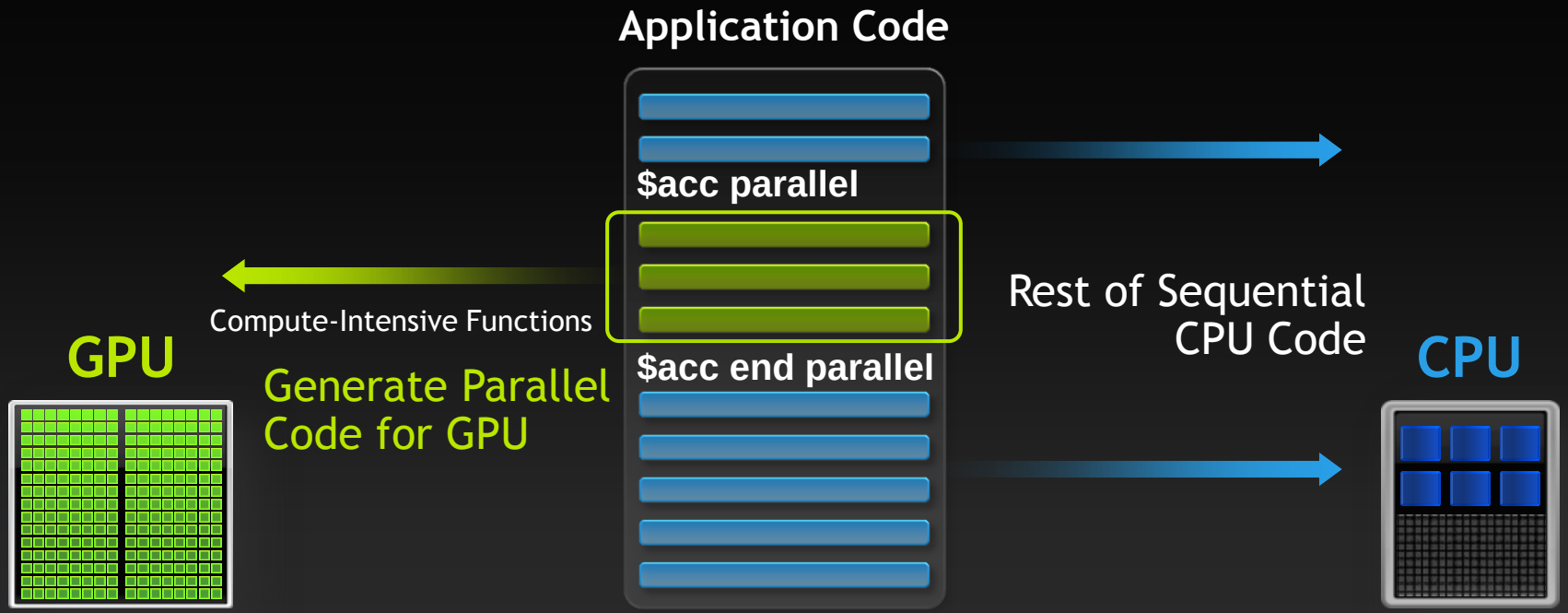
SAXPY in C

```
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 N elements
saxpy(N, 3.0, x, y);
...
```

SAXPY in Fortran

```
subroutine saxpy(n, a, x, y)
  real :: x(*), y(*), 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 N elements
call saxpy(N, 3.0, x, y)
...
```

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 parallel function that runs on the GPU

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

Compile (PGI)

- C:

```
pgcc -acc [-Minfo=accel] -o task1 task1.c
```

- Fortran:

```
pgf90 -acc [-Minfo=accel] -o task1 task1.f90
```

- Compiler output:

```
pgcc -fast -acc -Minfo=accel task1.c
saxpy:
      5, Accelerator kernel generated
          6, #pragma acc loop gang, vector(256) /* blockIdx.x threadIdx.x */
      5, 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
```

Getting basic accelerator information (PGI)

- The PGI compiler provides automatic instrumentation when `PGI_ACC_NOTIFY=<1,2,3>` at runtime

```
% export PGI_ACC_NOTIFY=1
% task1
launch CUDA kernel  file=~/task1.f90 function=saxpy
        line=7 device=0 grid=12 block=256
```

Getting basic accelerator information (PGI)

- The PGI compiler provides automatic instrumentation when `PGI_ACC_NOTIFY=<1,2,3>` at runtime

```
% export PGI_ACC_NOTIFY=3
% task1
upload CUDA data  file=~/task1.f90 function=saxpy line=7 device=0
       variable=x    bytes=12000
upload CUDA data  file=~/task1.f90 function=saxpy line=7 device=0
       variable=y    bytes=12000

launch CUDA kernel  file=~/task1.f90 function=saxpy line=7
      device=0 grid=12    block=256

download CUDA data  file=~/task1.f90 function=saxpy line=11
       device=0 variable=y bytes=12000
```

Run (PGI)

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

```
Accelerator Kernel Timing data
~/isc/openacc/task1.f90
  saxpy  NVIDIA  devicenum=0
    time(us): 65
    7: compute region reached 1 time
        7: data copyin reached 2 times
             device time(us): total=34 max=25 min=9 avg=17
        7: kernel launched 1 time
            grid: [12]  block: [256]
             device time(us): total=21 max=21 min=21 avg=21
            elapsed time(us): total=33 max=33 min=33 avg=33
        11: data copyout reached 1 time
             device time(us): total=10 max=10 min=10 avg=10
```

Task 1: Build SAXPY, compile & Run

- To do:
  - Complete set up the cloud infrastructure, editor..
  - Build demo at ~/openACC/task1
    - Direct invocation or use make
    - Express parallelism via “parallel loop” or “kernels” directive
  - Solutions in ~/openACC/task1.solution

- What do we learn?
  - Know how to compile with PGI OpenACC
  - First experience with kernels, parallel loop directive

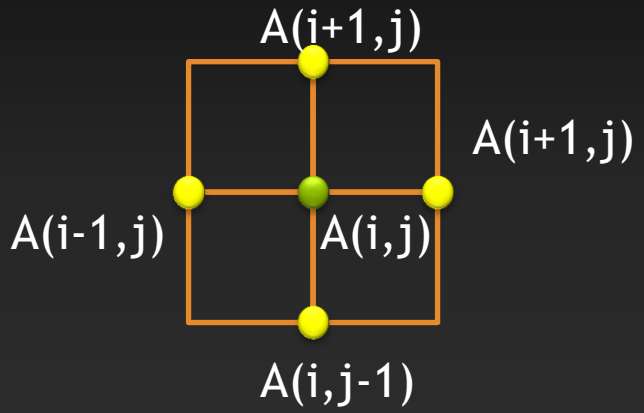
- Optional for C: Investigate effect of “restrict” keyword
  - How does the compiler output change?

Time: 20 minutes

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

Jacobi Iteration: Fortran Code

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


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

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

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


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

  iter = iter +1
end do
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

Jacobi Iteration: OpenMP Fortran Code

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

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

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

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

Parallelize loop across CPU threads

Parallelize loop across CPU threads

Jacobi Iteration: OpenACC Fortran Code

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

!$acc parallel loop reduction(max:err)
  do j=1,m
    do i=1,n

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

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

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
```

Performance

CPU: AMD IL-16
@ 2.2 GHz

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	109.7	--
CPU 2 OpenMP threads	71.6	1.5x
CPU 4 OpenMP threads	53.7	2.0x
CPU 8 OpenMP threads	65.5	1.7x
CPU 16 OpenMP threads	66.7	1.6x
OpenACC GPU	180.9	0.6x FAIL!

GPU: NVIDIA Tesla K20X

Speedup vs.
1 CPU core

Speedup vs.
4 OpenMP Threads

What went wrong?

- Set `PGI_ACC_TIME` environment variable to '1'

```
Accelerator Kernel Timing data

/laplace2d.c
  main
    69: region entered 1000 times
                total=109,998,808 init=262 region=109,998,546
                kernels=1,748,221 data=109,554,793
        w/o init: total=109,998,546 max=110,762 min=109,378 avg=109,998
        69: kernel launched 1000 times
            grid: [4094]  block: [256]
            time(us): total=1,748,221 max=1,820 min=1,727

/laplace2d.c
  main
    57: region entered 1000 times
                total=71,790,531 init=491,553 region=71,2
                kernels=2,369,807 data=68,968,929
        w/o init: total=71,298,978 max=75,603 min=70,486 avg=71,298
        ..
```

1.7 seconds

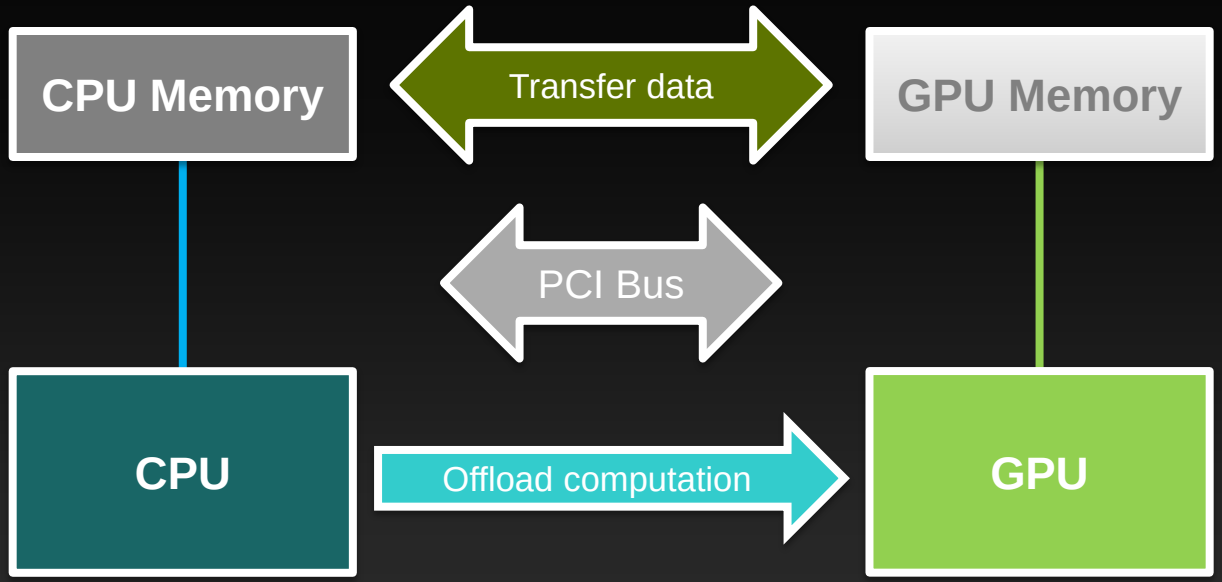
109.5 seconds

2.4 seconds

68.9 seconds

Huge Data Transfer Bottleneck!
Computation: 4.1 seconds
Data movement: 178.4 seconds

Basic Concepts



For efficiency, decouple data movement and compute off-load

Excessive Data Transfers

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

A, Anew resident on host

Copy

```
#pragma acc parallel loop reduction(max:err)
```

A, Anew resident on accelerator

```
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]);
  }
}
```

These copies happen every iteration of the outer while loop!*

A, Anew resident on accelerator

A, Anew resident on host

Copy

```
    ...
}
```

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

Data Management

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
    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 data
```

Data Region

Arrays a, b, and c 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

Jacobi Iteration: OpenACC Fortran Code

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

!$acc parallel loop reduction(max:err)
  do j=1,m
    do i=1,n

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

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

  ...

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

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

Did it help?

```
Accelerator Kernel Timing data
/laplace2d.c
  main
    69: region entered 1000 times
        time(us): total=1,791,050 init=217 region=1,790,833
                  kernels=1,742,066
        w/o init: total=1,790,833 max=1,950 min=1,773 avg=1,790
        69: kernel launched 1000 times
            grid: [4094]  block: [256]
            time(us): total=1,742,066 max=1,809 min=1,725 avg=1,742
/laplace2d.c
  main
    57: region entered 1000 times
        time(us): total=2,710,902 init=182 region=2,710,720
                  kernels=2,361,193
        w/o init: total=2,710,720 max=4,163 min=2,697 avg=2,710
        57: kernel launched 1000 times
            grid: [4094]  block: [256]
            time(us): total=2,339,800 max=3,709 min=2,334 avg=2,339
        58: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=21,393 max=1,321 min=19 avg=21
/laplace2d.c
  main
    51: region entered 1 time
        time(us): total=5,063,688 init=489,133 region=4,574,555
                  data=68,993
        w/o init: total=4,574,555 max=4,574,555 min=4,574,555 avg=4,574,555
```

0.69 seconds

Performance

CPU: AMD IL-16
@ 2.2 GHz

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	109.7	--
CPU 2 OpenMP threads	71.6	1.5x
CPU 4 OpenMP threads	53.7	2.0x
CPU 8 OpenMP threads	65.5	1.7x
CPU 16 OpenMP threads	66.7	1.6x
OpenACC GPU	4.96	10.8x

GPU: NVIDIA Tesla K20X

Speedup vs.
1 CPU core

Speedup vs.
4 OpenMP Threads

Further speedups

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

Tips and Tricks

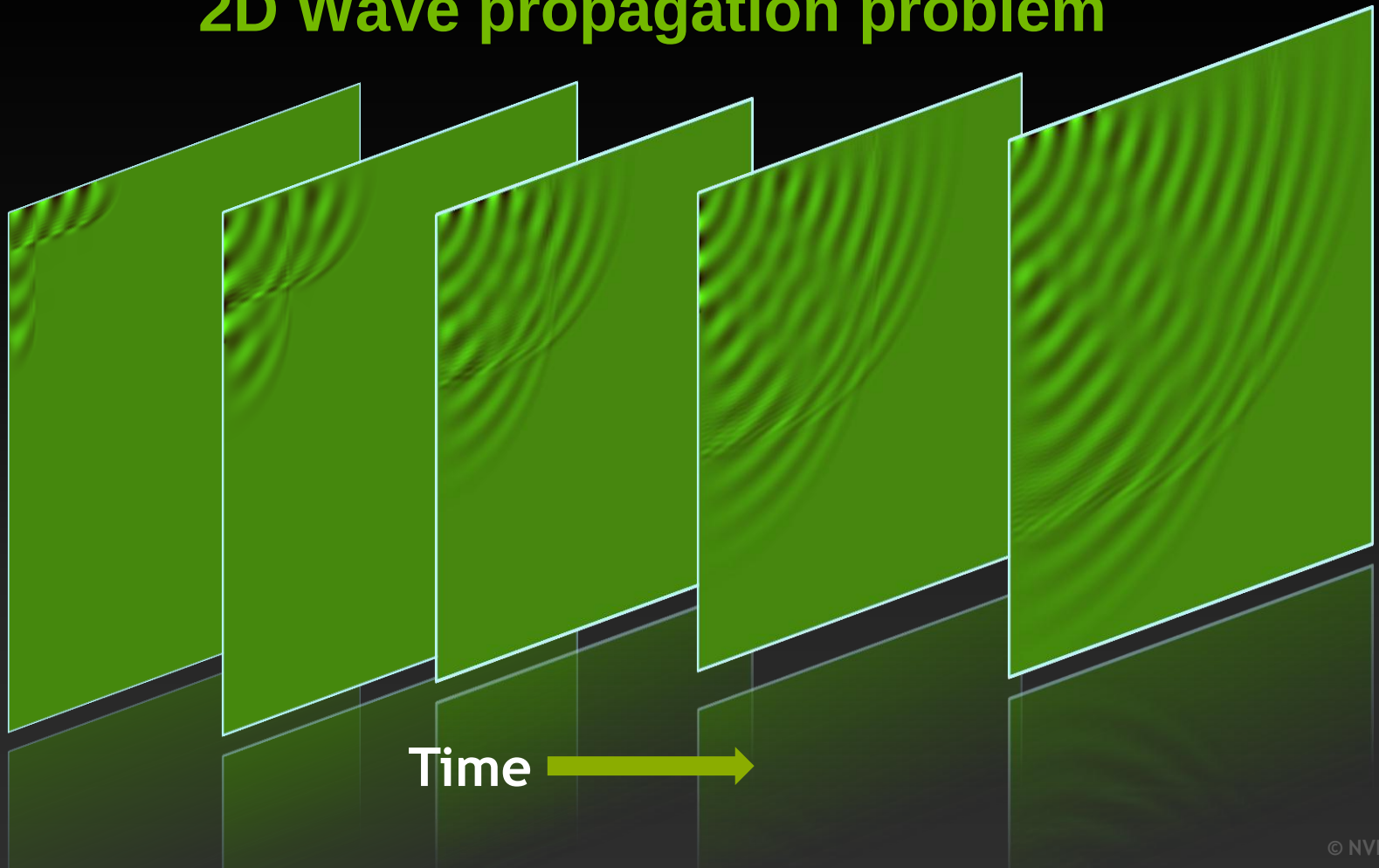
- Nested loops are best for parallelization
  - Large loop counts (1000s) needed to offset GPU/memcpy overhead
- Iterations of loops must be independent of each other
  - To help compiler: use `restrict` keyword in C
- Compiler must be able to figure out sizes of data regions
  - Can use directives to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

Tips and Tricks (cont.)

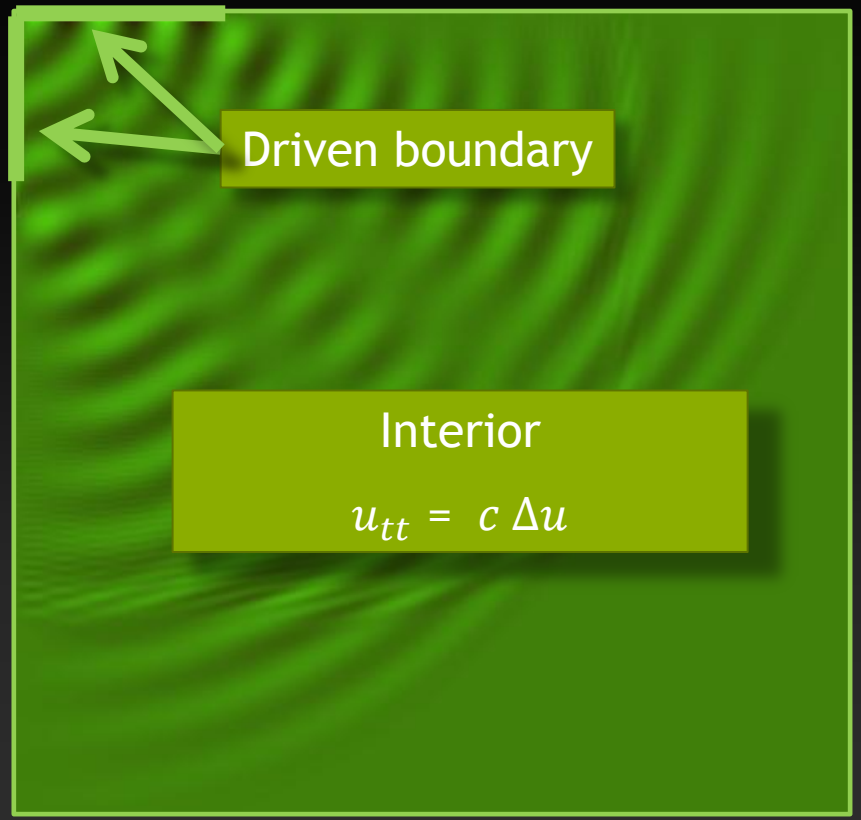
- Use time option to learn where time is being spent
  - (PGI) `PGI_ACC_TIME=1` (runtime environment variable)
  - (Cray) `CRAY_ACC_DEBUG=<1,2,3>` (runtime environment variable)
  - (CAPS) `HMPPRT_LOG_LEVEL=info` (runtime environment variable)
- Pointer arithmetic should be avoided if possible
  - Use 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

Hands-On Experiments

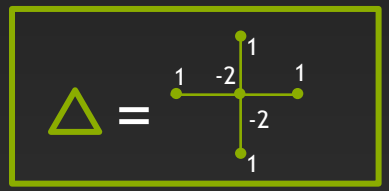
2D Wave propagation problem

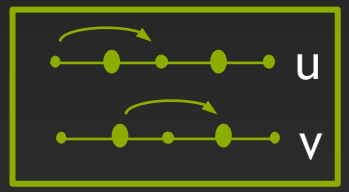


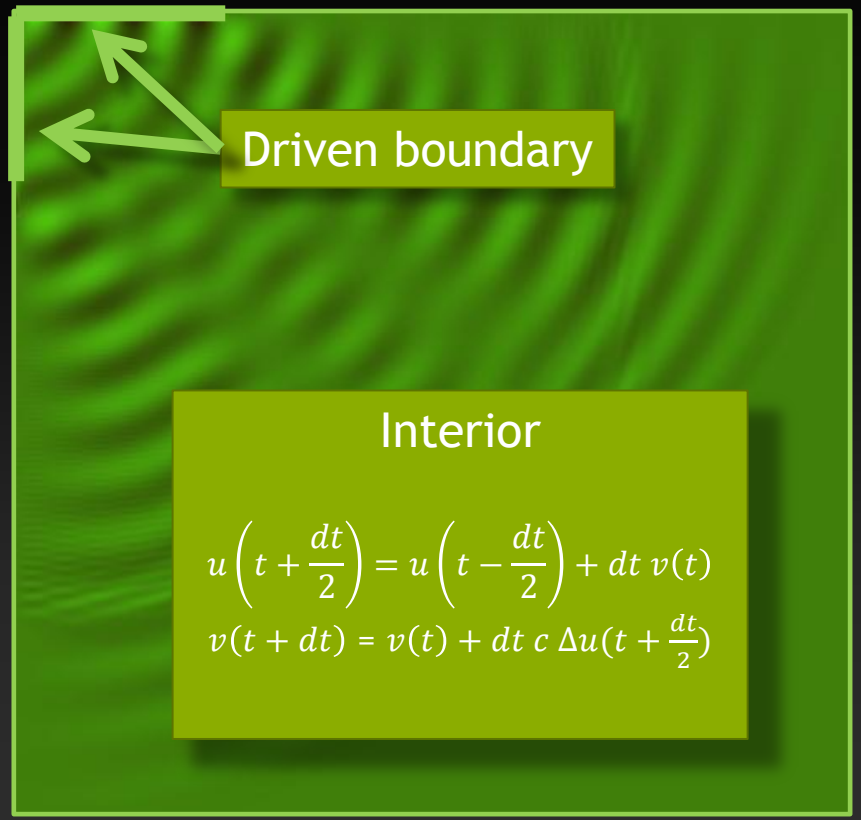
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$



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







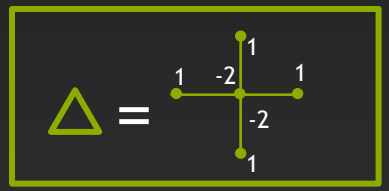
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$

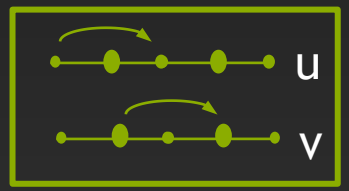
Initialize fields u, v

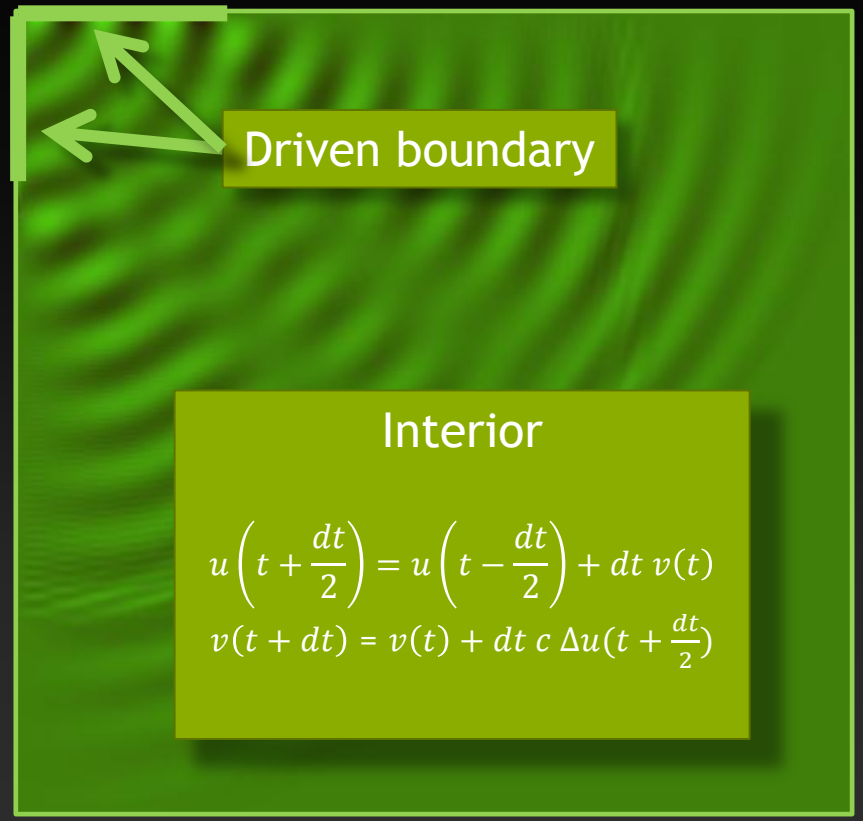
u = u + dt * v

v = v + dt * c * Δ u

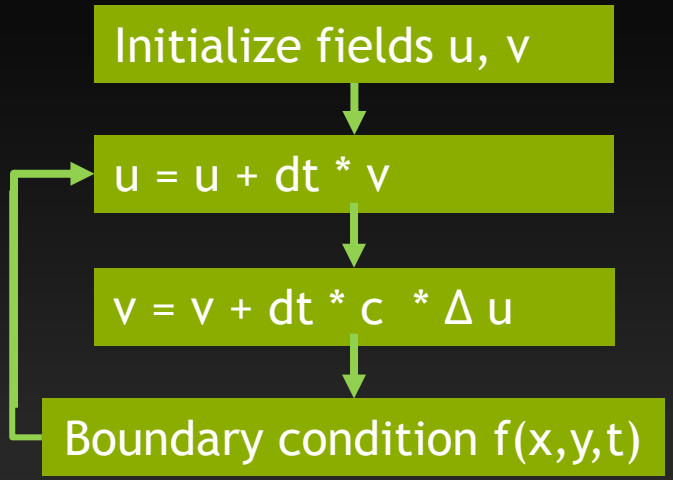
Boundary condition f(x,y,t)





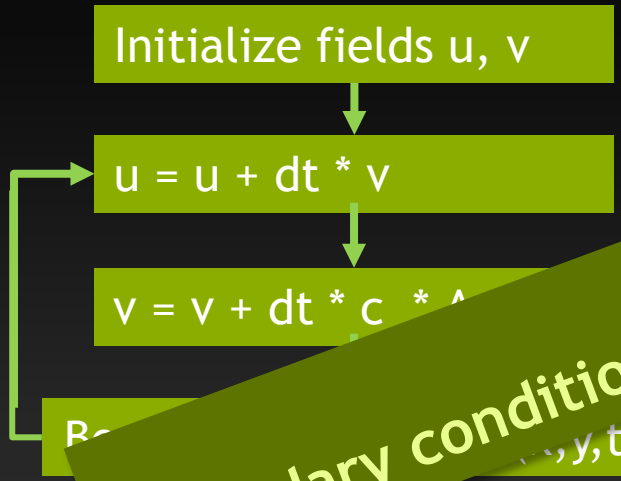


Sample: Basic Euler Solver




```
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=2,nx-1

                                  (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
```

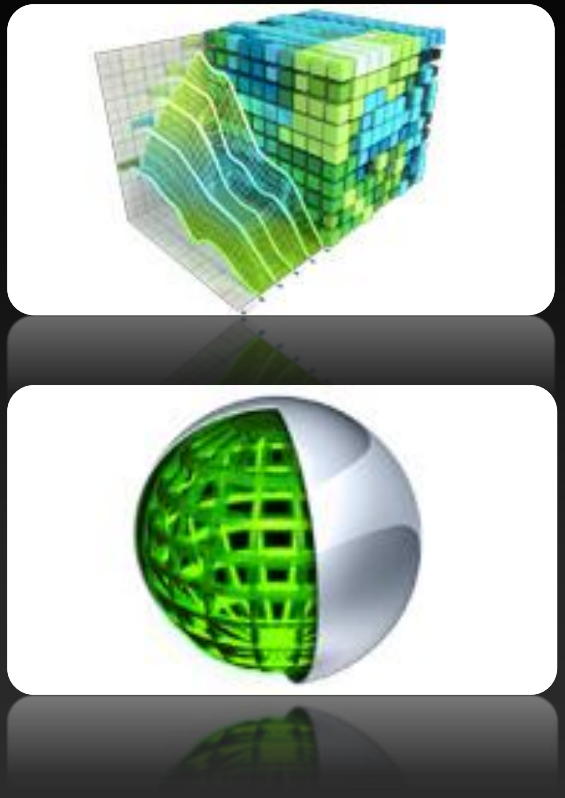
Task 2: Use OpenACC for u, v update

- To do:
  - Code walk through
  - Express parallelism via kernels or parallel directive
    - update_u and update_v only
    - Transfer data for each kernel
    - In C, remember to properly declare the array sizes
- What do we learn?
  - Use directives in a more complex application
  - copy clause, array declaration in C
  - Proper use of restrict
- Optional:
  - Analyze data flow

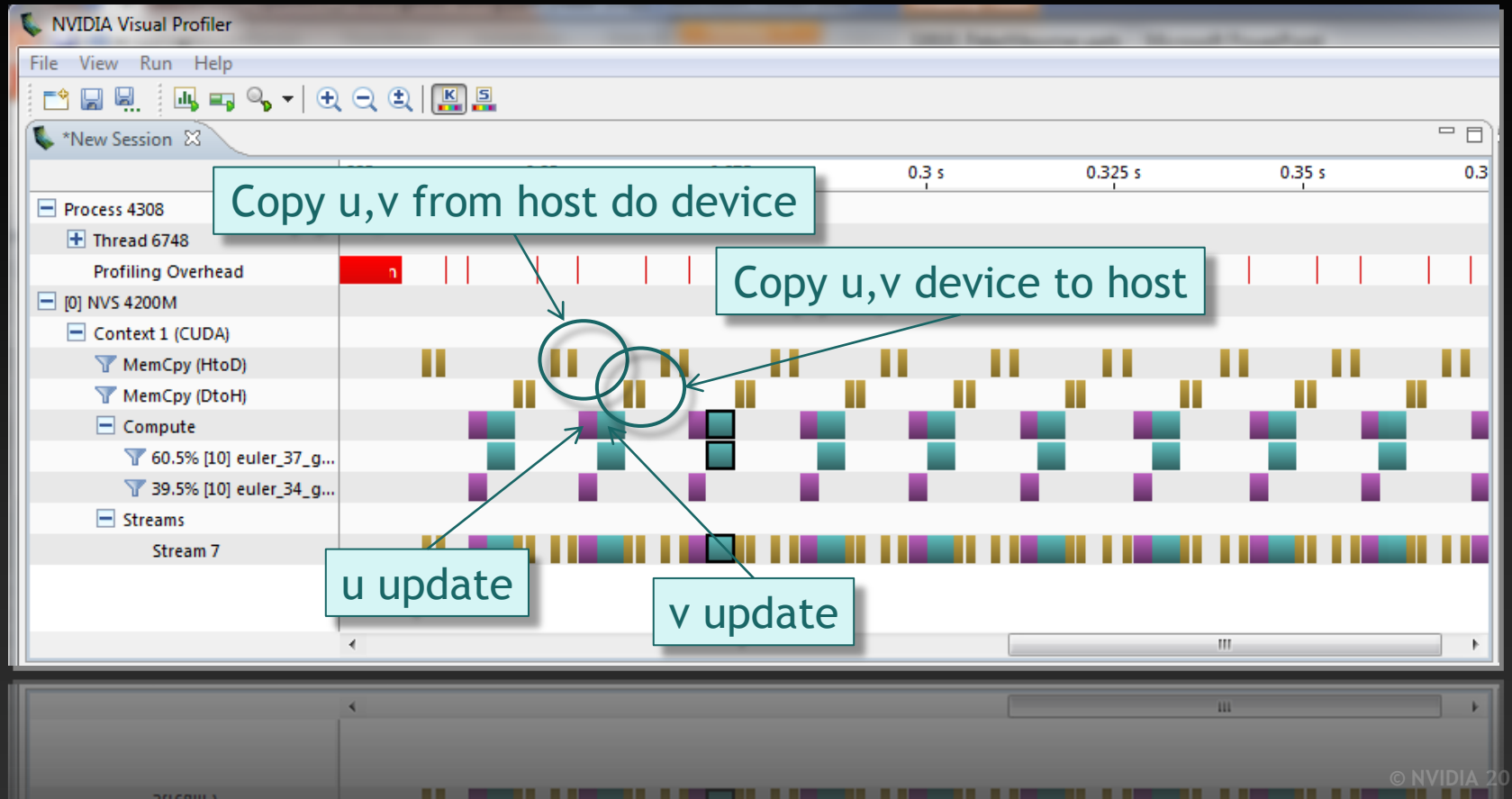
Time: 20 minutes

Profile your application

- 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
  - Similar env variables for Cray, Caps
- NVIDIA Visual Profiler
- 3rd-party profiling tools that are CUDA-aware
  - (But those are outside the scope of this talk)



Profiling via NVVP



Task 3: Reduce data transfer via data region

- To do:
  - Use data region to keep u, v resident on device
    - update_u and update_v only
    - Transfer data for each kernel

- What do we learn?
  - Data regions
  - Present-or-copy directives
  - Reminder: host memory space is NOT device memory space

- Optional:
  - Use nvvp to analyze timeline

Time: 20 minutes

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

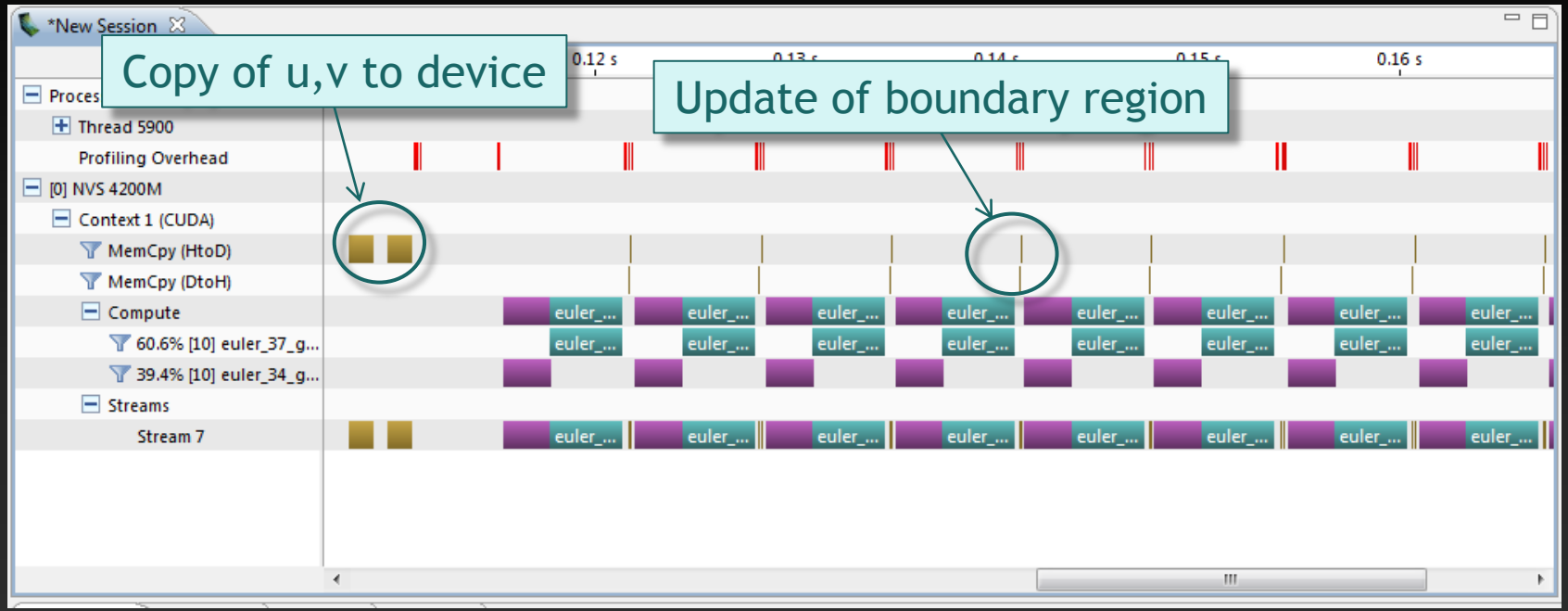
Copy “a” from CPU to GPU

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

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
  - …

`update` significantly reduces data transfer cost



Summary of Data Motion Optimizations

- Basic OpenACC annotation

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

- Resident GPU data

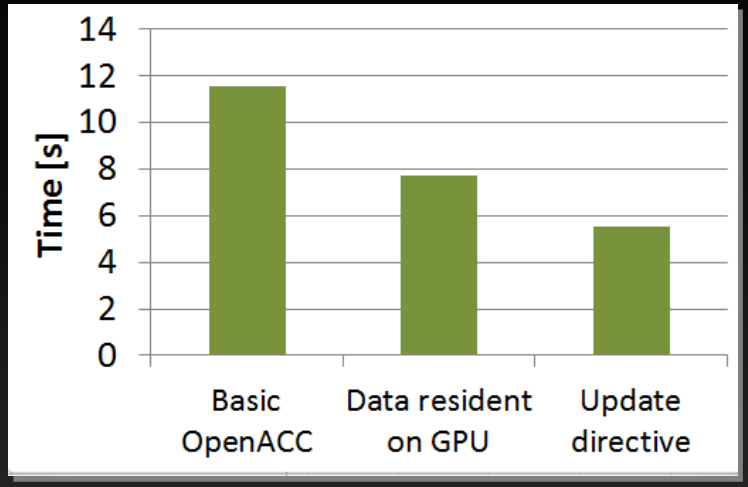
```
!$acc data copy(a)
```

- Variable update

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

- Subarray specification

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



Assumption:
Boundaries computed on CPU

Optimizations

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

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,

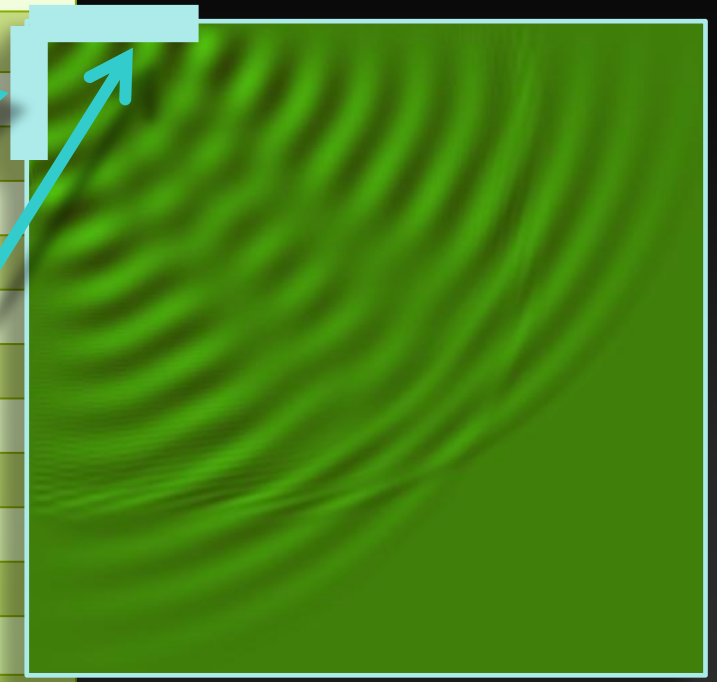
Do not wait for kernel completion

Executes concurrently with kernels

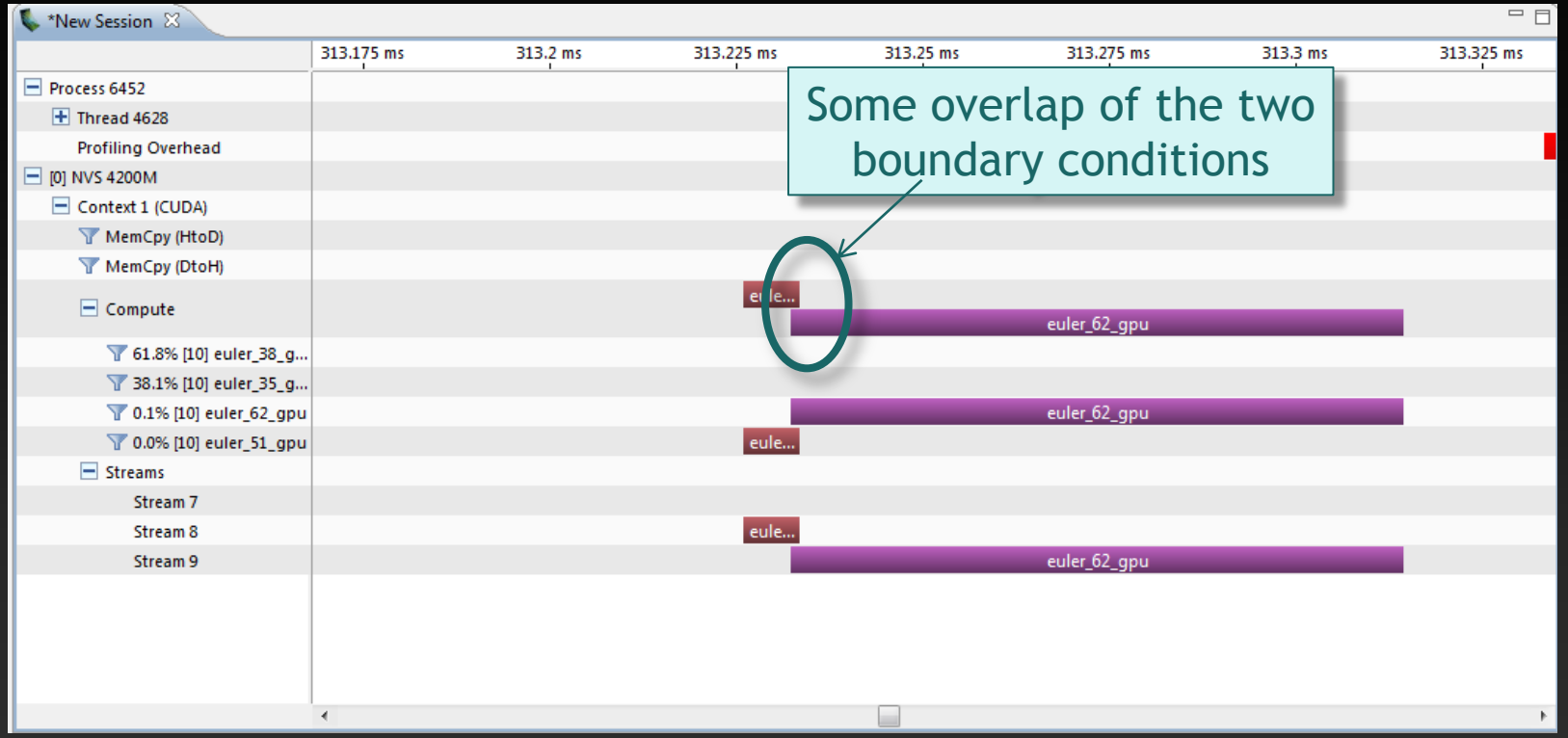
(potentially) executes concurrently with kernels

Set Boundary Regions concurrently

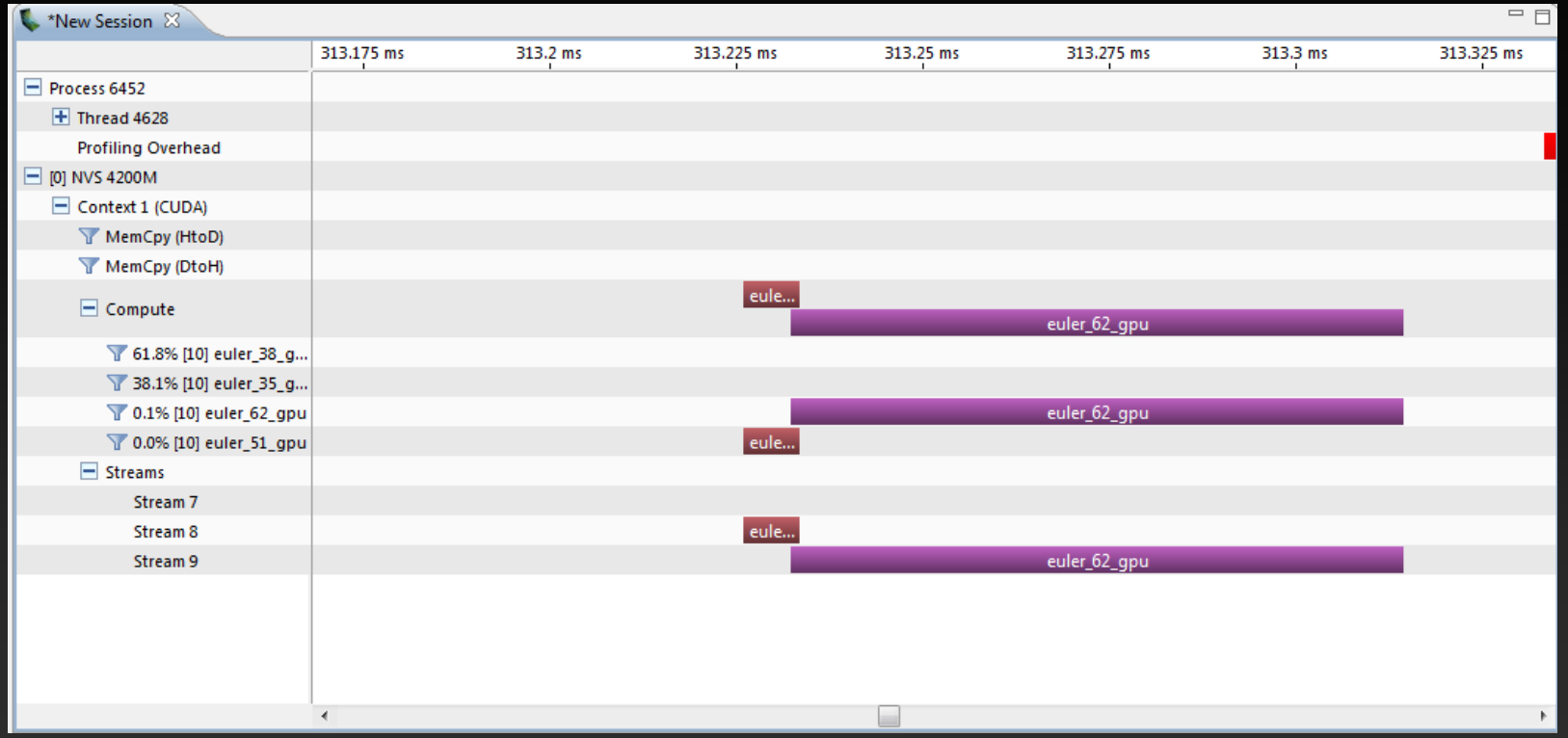
```
!$acc kernels async(1)
do y=1, 2
   do x=1, nx/4
        u(x,y) = sin(6.28*real(t) + real(x)/6.28) ..
    end do
end do
!$acc end kernels
!$acc kernels async(2)
do y=1, ny/4
    do x=1, 2
        u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
   end do
 end do
!$acc end kernels
!$acc wait
```



Async clause leads to overlapping kernels



Async clause leads to overlapping kernels

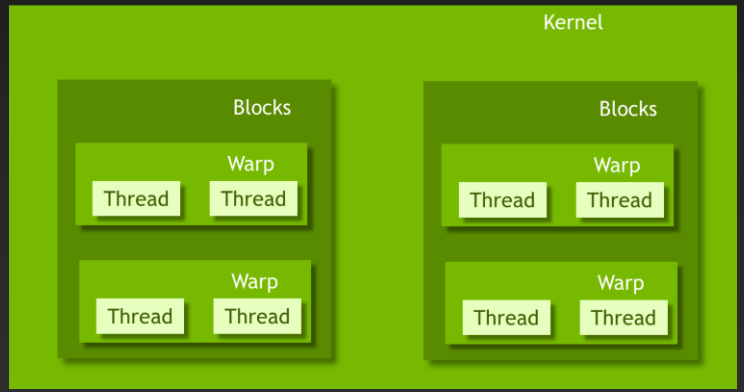


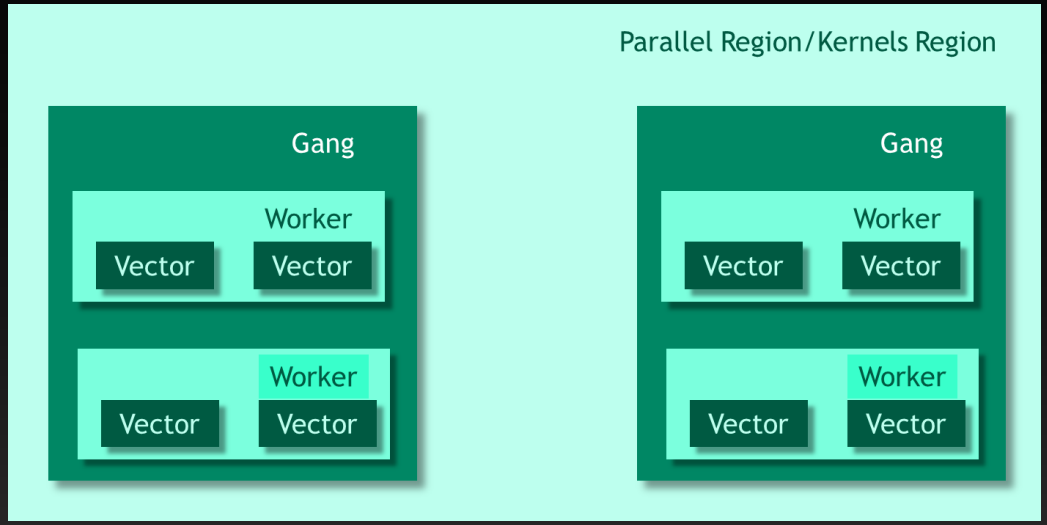
3 Levels of Parallelism

`gang`	“Loose” parallelism, where gangs can work independently without synchronization.
`worker`	“Tighter” parallelism, where workers may share data and/or coordinate within a common gang.
`vector`	“Tightest” parallelism, where a vector instruction may be used across multiple data.

OpenACC parallelism maps to CUDA..

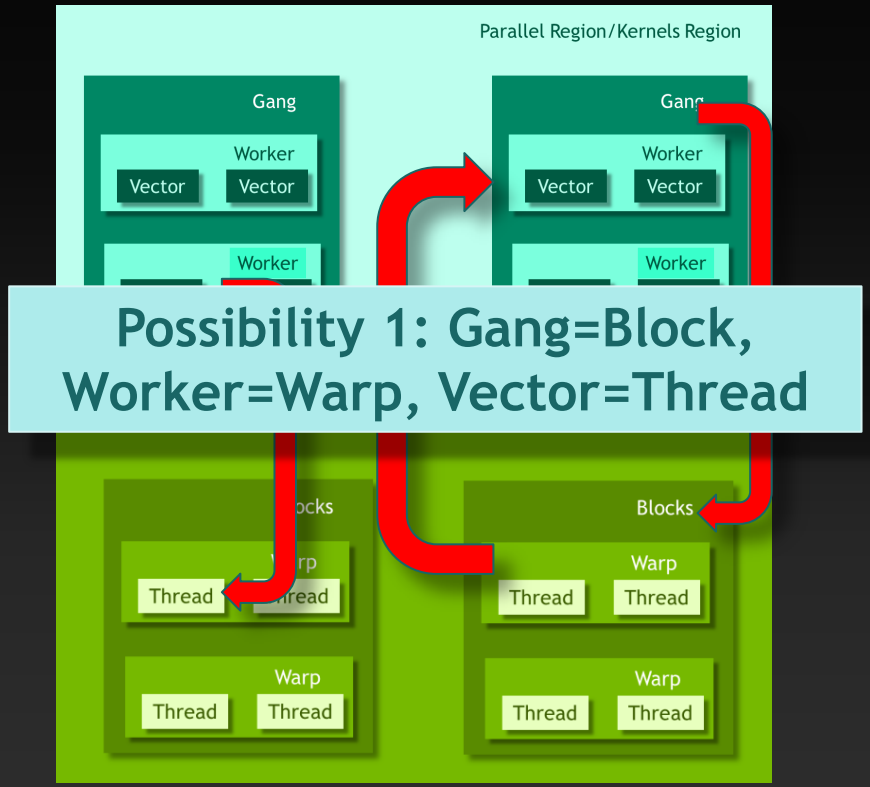
CUDA



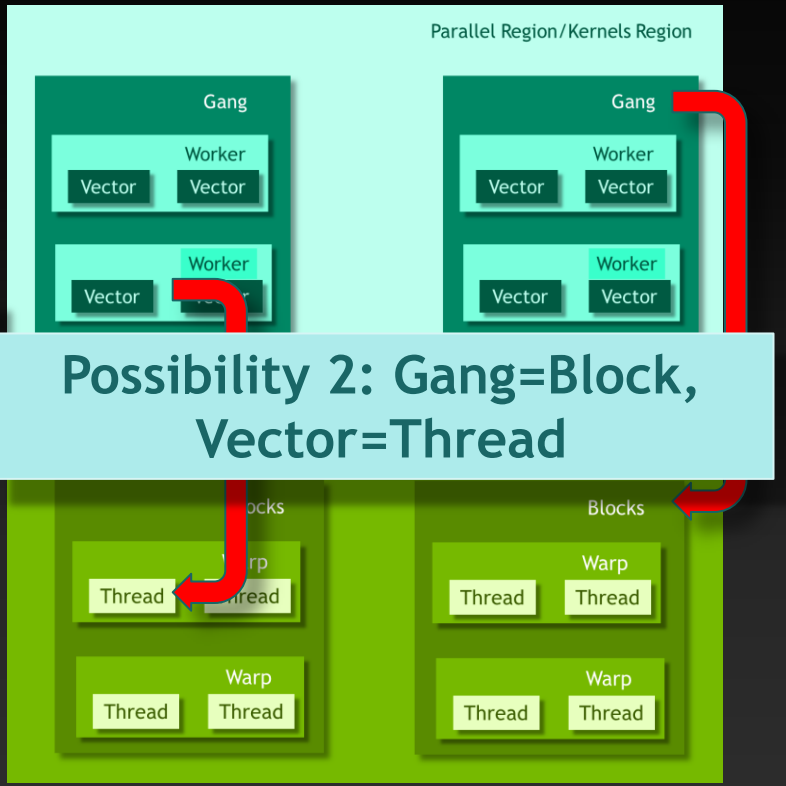


OpenACC

.. but mapping is left to the compiler



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



Possibility 2: Gang=Block, Vector=Thread

Additional `loop` Clauses

`independent`	Loops within a kernels region are independent and may be overlapped.
`collapse(n)`	This loop should be merged with the next n loops to expose additional parallelism.
`reduction()`	Loop contains a parallel reduction, care must be taken by the compiler
`private()`	Each iteration needs a private copy of the listed variables
`vector_length()`	
`num_workers()`	
`num_gangs()`	explicitly set the degree of parallelism at each level

Task 4: Minimize data transfer

- To do:
  - Create kernel for boundary condition
    - Transfer data back for diagnostic output
  - Determine optimal vector length to minimize execution time

- What do we learn?
  - Kernel execution over sub-arrays
  - Manual loop scheduling

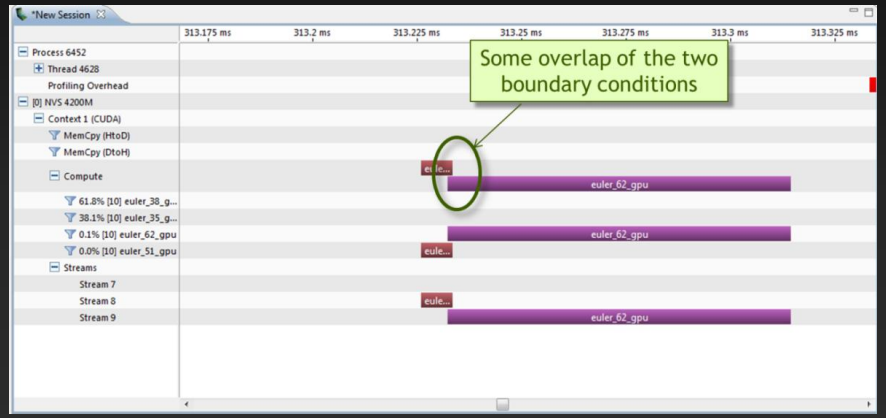
- Optional:
  - Use nvvp to analyze timeline

Time: 20 minutes

Summary of optimization steps

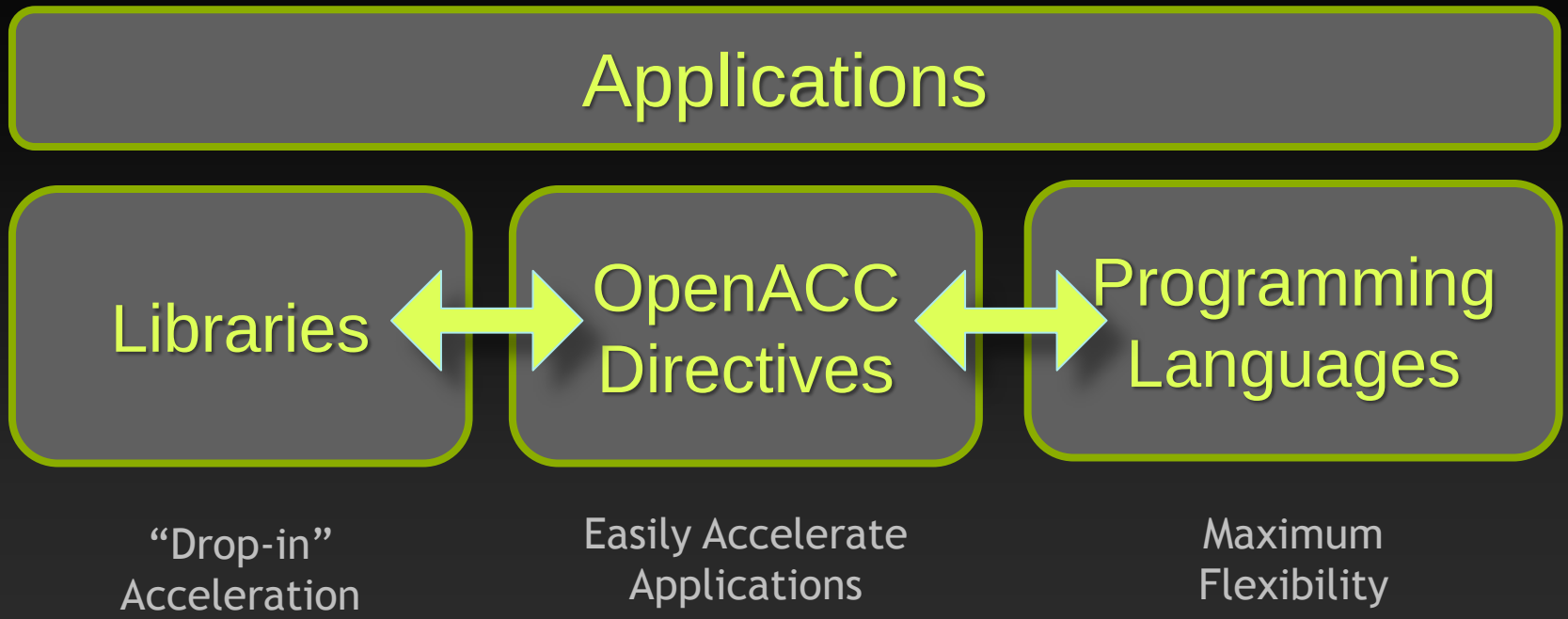
- Minimize data transfer (update)
- Optimize overlap (async)
- Manual scheduling (gang, worker, vector)

- Use profiler
- Use compiler feedback



Sharing OpenACC data with Libraries

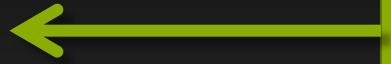
Interfacing OpenACC with Libraries, CUDA



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 iso_c_binding
real*8, dimension(1024) :: a
integer*8, dimension(1) :: a_ptr

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

Task 5: Compute field magnitude using cublas

- To do:
  - Use host_data to obtain device pointers
    - Transfer data back for diagnostic output
  - Use cublasDasum(n, x, incx) to compute sum
- What do we learn?
  - Using host_data
- Optional:
  - Use cublas for update of u

Time: 20 minutes

Summary

- OpenACC expresses parallelism and locality
  - Kernels, Parallel regions
- 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)


Thank you!
NVIDIA