Programmation hybride : une étape vers l'Exaflops

François Courteille | fcourteille@nvidia.com | Ateliers Lyon Calcul | 25 Février 2014

Outline

- Introduction to Accelerated computing
- NVIDIA architecture roadmap and current line-up
- Introduction to hybrid processing with GPU
- 4 ways to accelerate applications
  - Applications catalog
  - Libraries
  - OpenACC directives programming
    - Introduction
    - First Example : Porting a Jacobi solver
    - Hands-on : Tune an example application
      - Data motion optimization
      - Loop scheduling optimizations
      - Asynchronous execution
    - Interface OpenACC with libraries/CUDA
      - Interoperability with OpenGL
      - MPI communication
    - Summary
  - Programming languages : CUDA, PYTHON, MATLAB,….

World Leader in Visual Computing

GAMING

PRO VISUALIZATION

HPC & BIG DATA

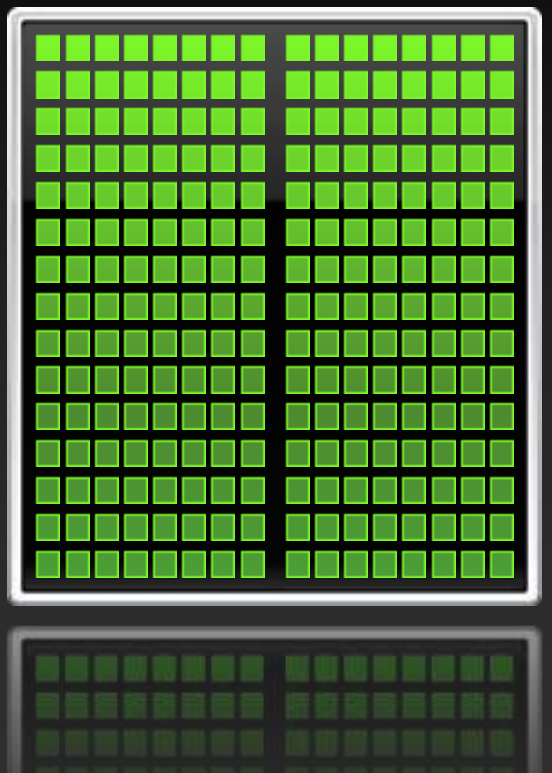
MOBILE COMPUTING

Accelerated Computing

10x Performance, 5x Energy Efficiency

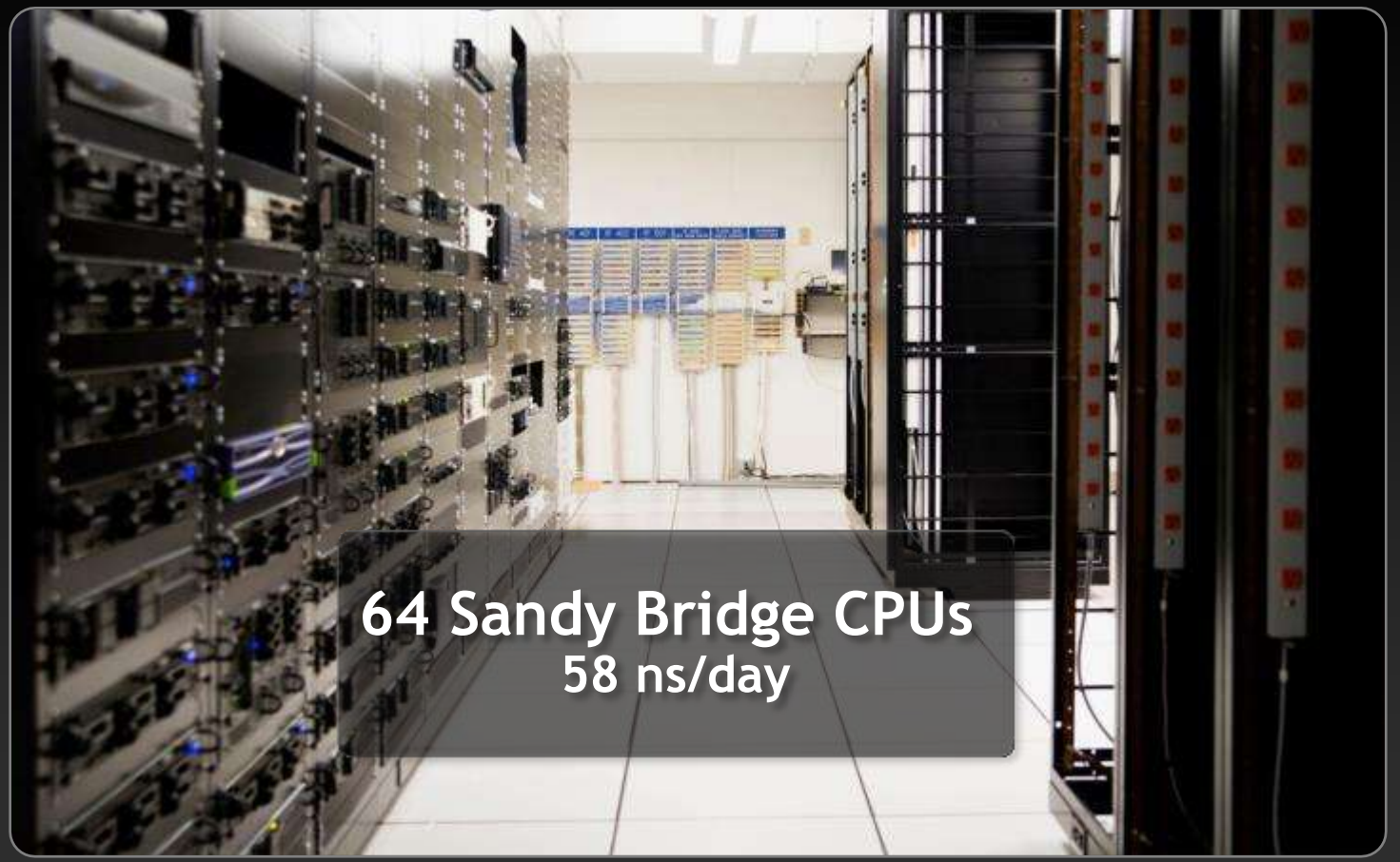
CPU
Optimized for
Serial Tasks

GPU Accelerator
Optimized for Many
Parallel Tasks



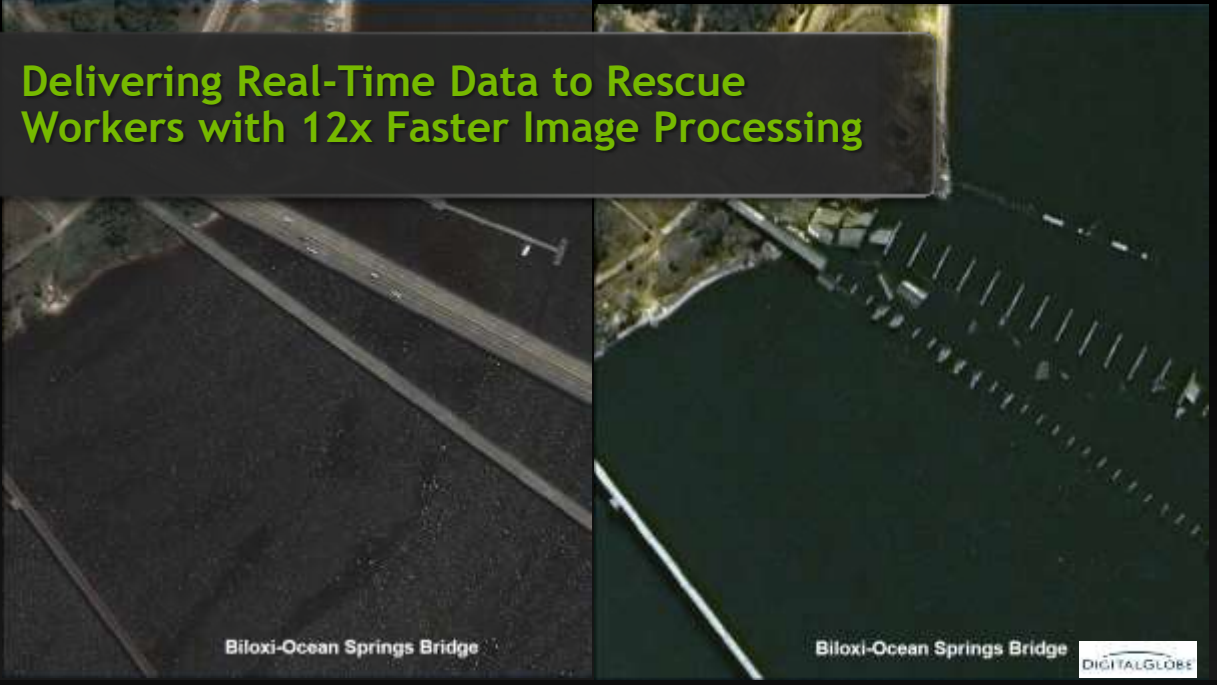
Unprecedented Value to Scientific Computing

AMBER Molecular Dynamics Simulation
DHFR NVE Benchmark

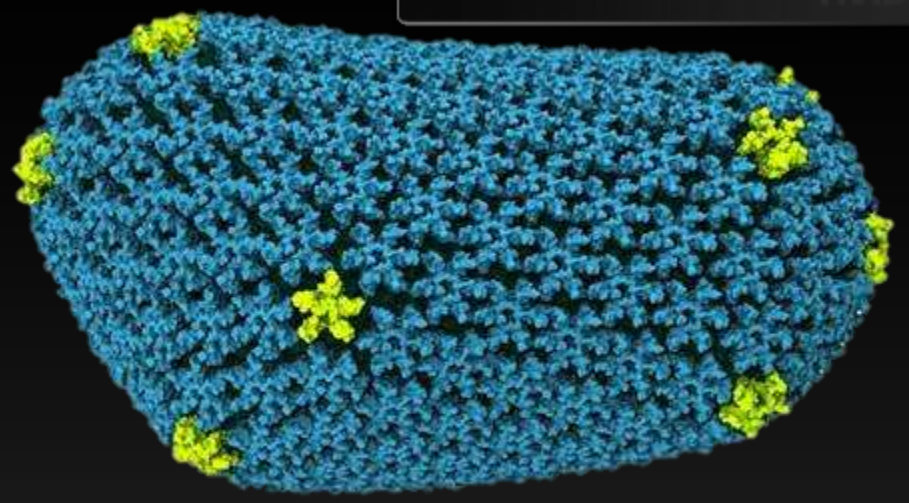


1 Tesla K40 GPU
102 ns/day

Delivering Real-Time Data to Rescue Workers with 12x Faster Image Processing



Accelerating Breakthrough in HIV Research

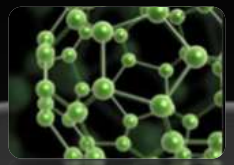


Powering the Fastest Supercomputers In USA and Europe

Accelerating Mainstream Datacenters

Oil & Gas

- Schlumberger
- BR PETROBRAS
- Chevron
- Eni
- Statoil
- Paradigm



Higher Ed

- Chinese Academy of Sciences
- Georgia Tech
- HARVARD School of Engineering and Applied Sciences
- STANFORD UNIVERSITY

Government

- Air Force Research Laboratory
- Raytheon
- NASA
- Naval Research Laboratory
- MITRE

Supercomputing

- CSCS Swiss National Supercomputing Centre
- NCSA
- Tokyo Institute of Technology
- OAK RIDGE National Laboratory

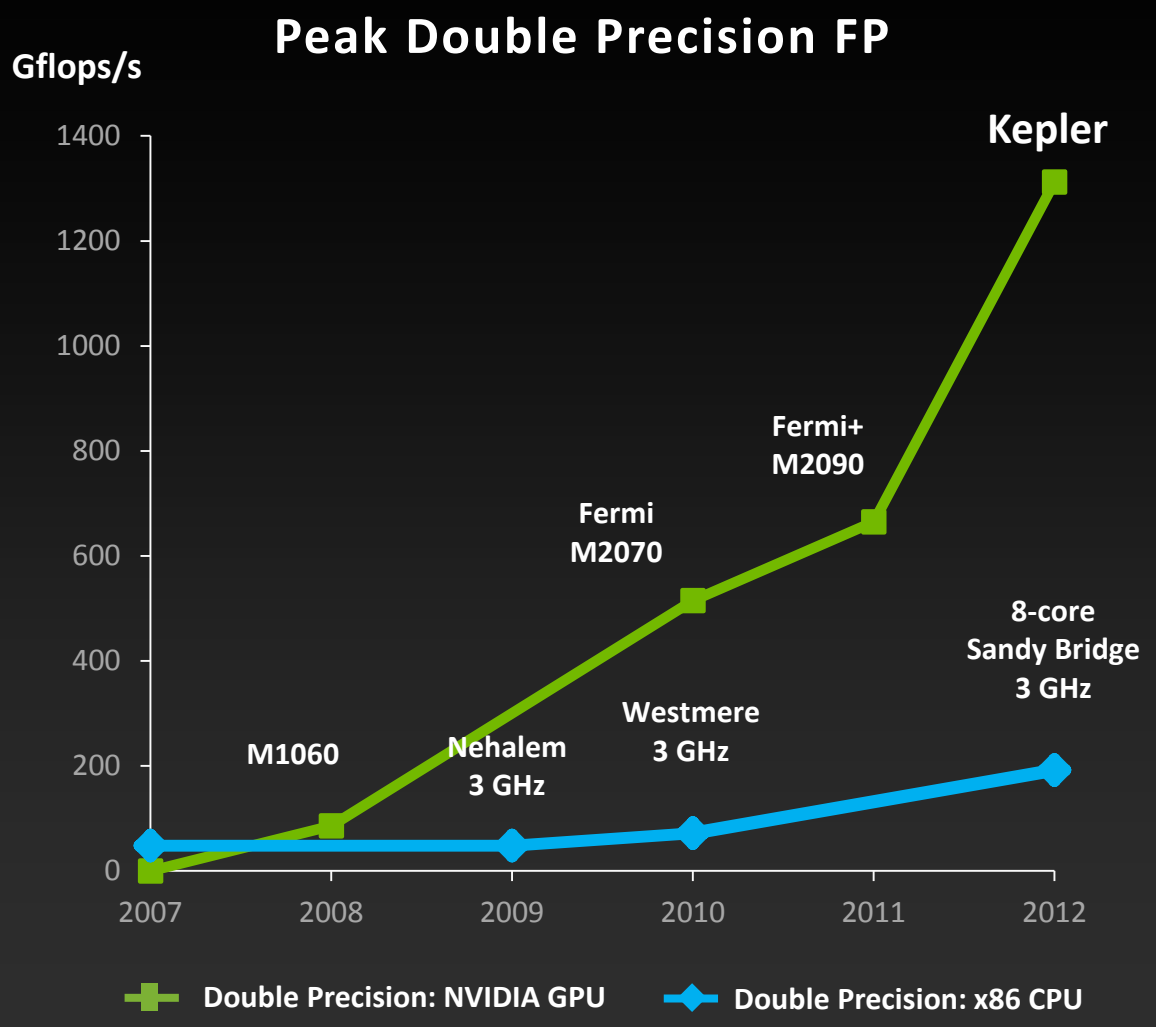
Finance

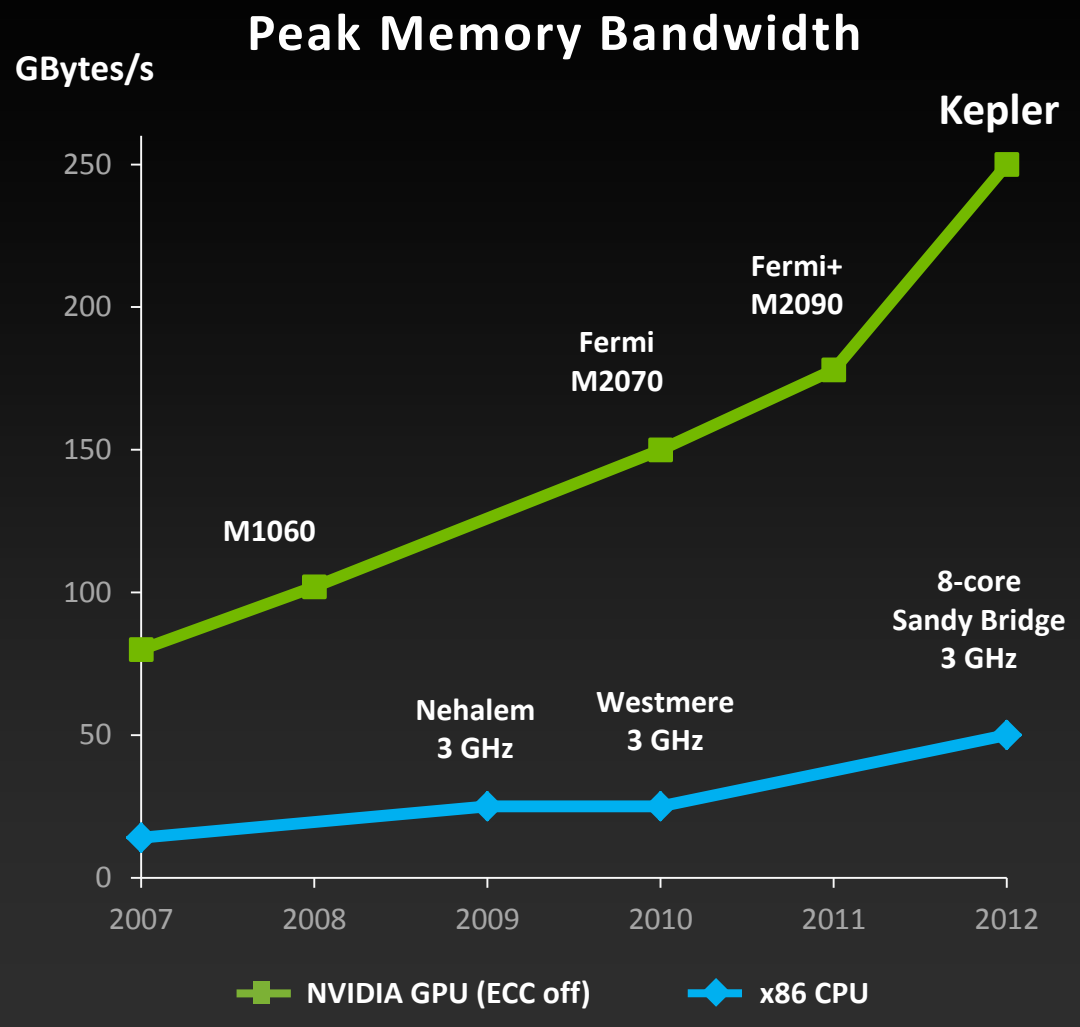
- J.P.Morgan
- BARCLAYS
- STANDARD LIFE
- BNP PARIBAS
- MUREX

Web 2.0

- Baidu 百度
- salesforce SOFTWARE
- SHAZAM
- amazon.com

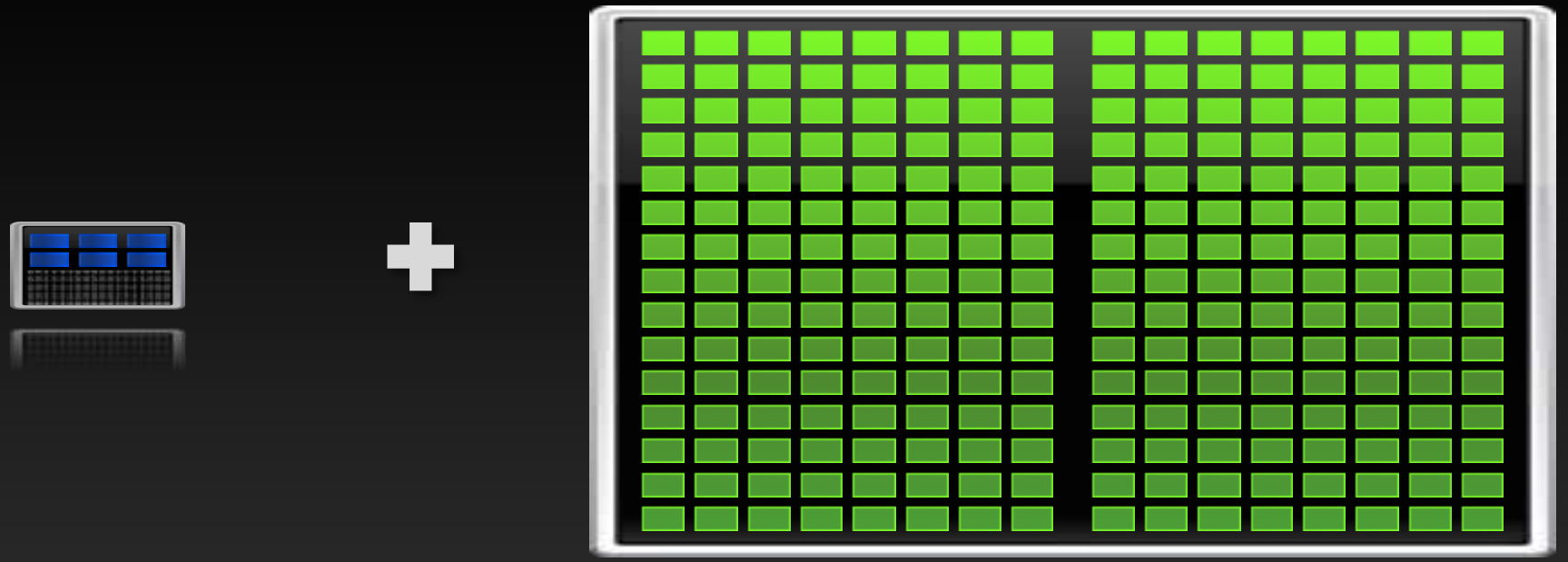
The March of GPUs





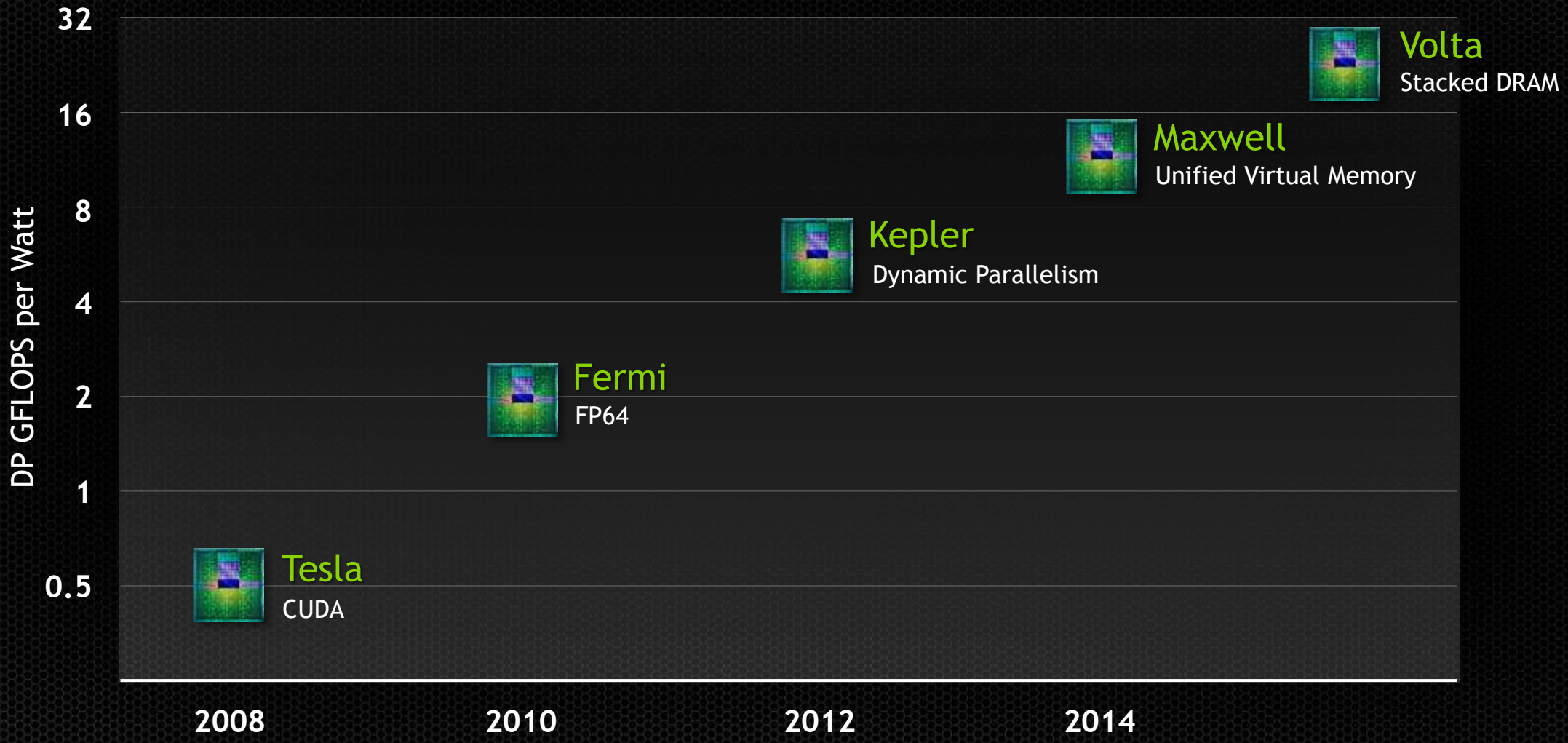
CPU

GPU



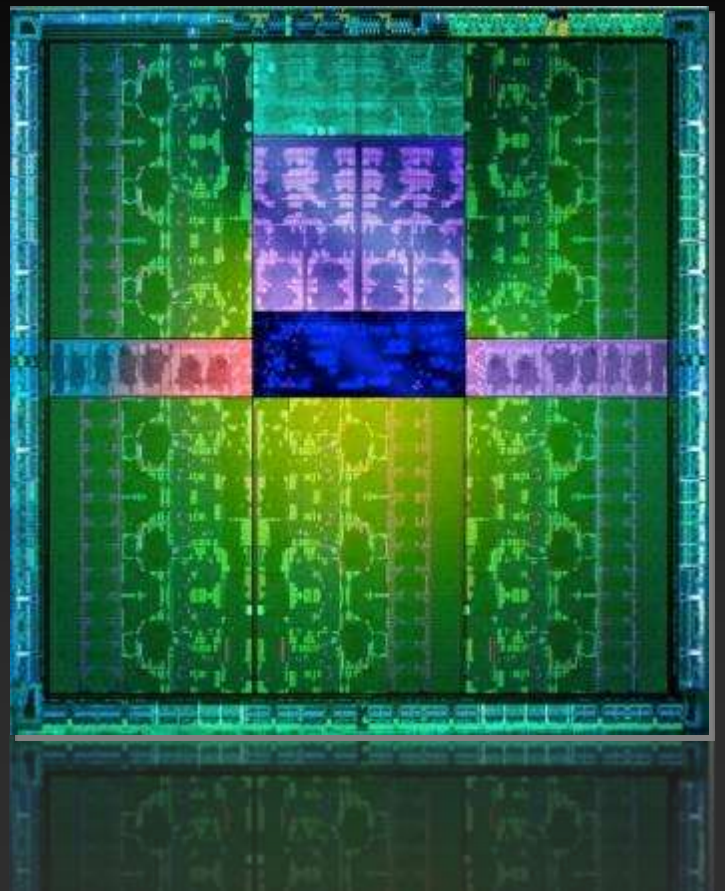
Add GPUs: Accelerate Applications

Strong CUDA GPU Roadmap



Kepler

Fastest, Most Efficient HPC Architecture Ever

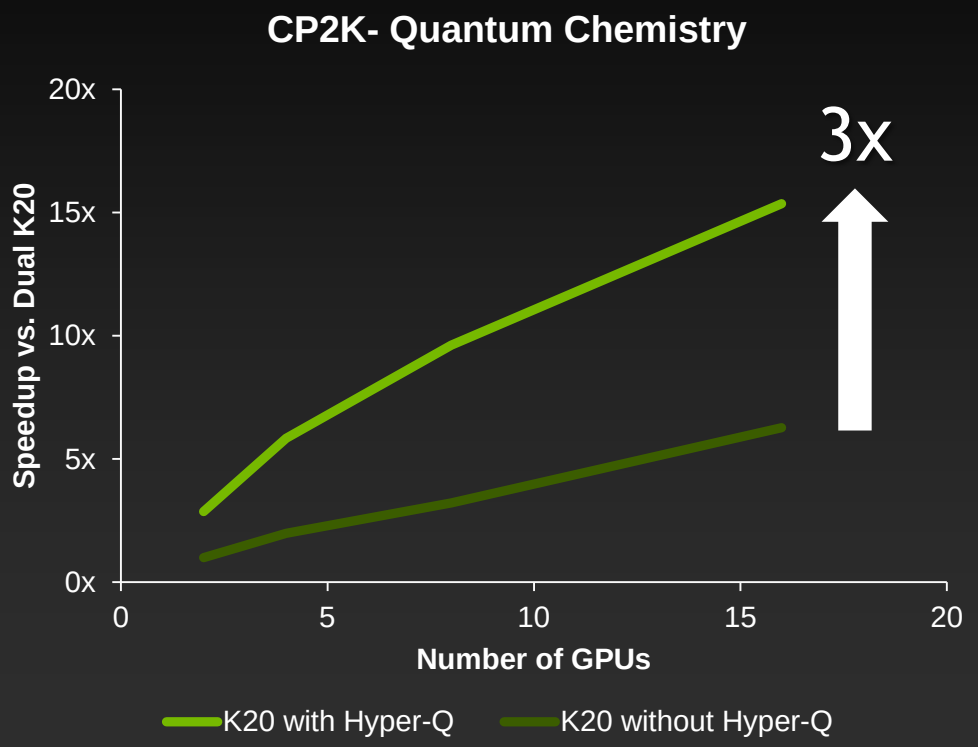


SMX	3x Performance per Watt
Hyper-Q	Easy Speed-up for Legacy MPI Apps
Dynamic Parallelism	Parallel Programming Made Easier than Ever

GPU Coding Made Easier & More Efficient

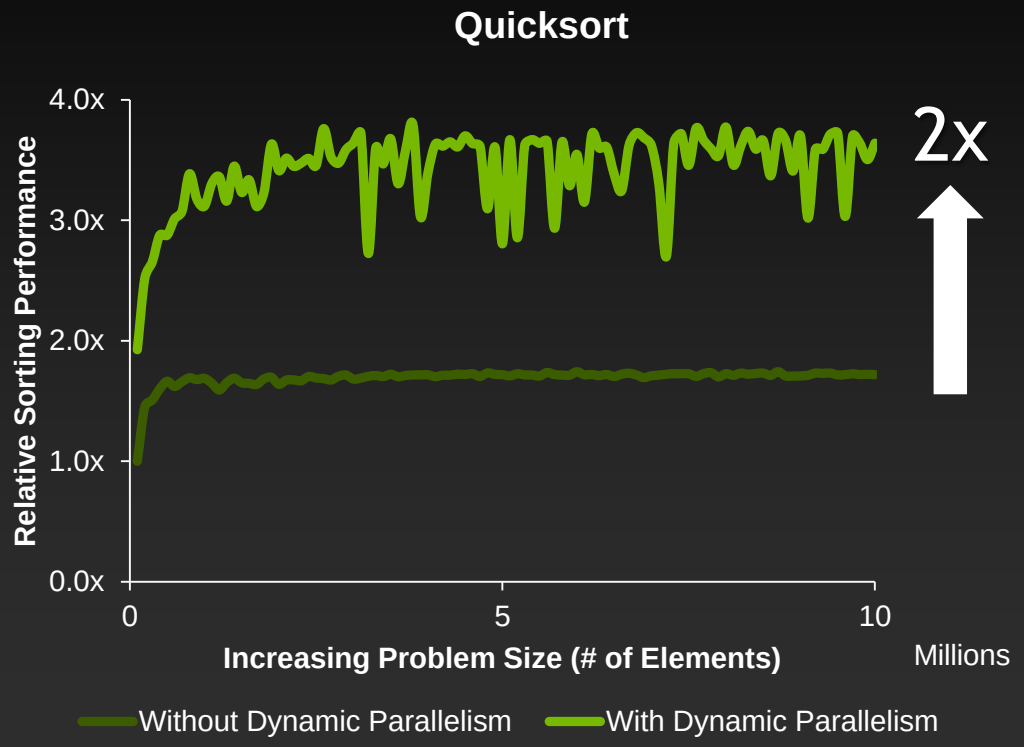
Hyper-Q: 32 MPI jobs per GPU

Easy Speed-up for Legacy MPI Apps



Dynamic Parallelism: GPU Generates Work

Less Effort, Higher Performance

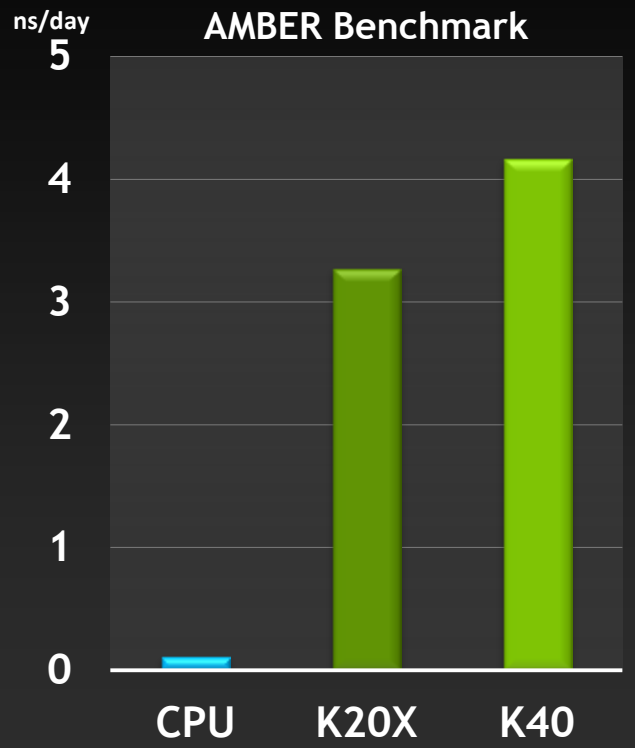


Tesla K40

World's Fastest Accelerator

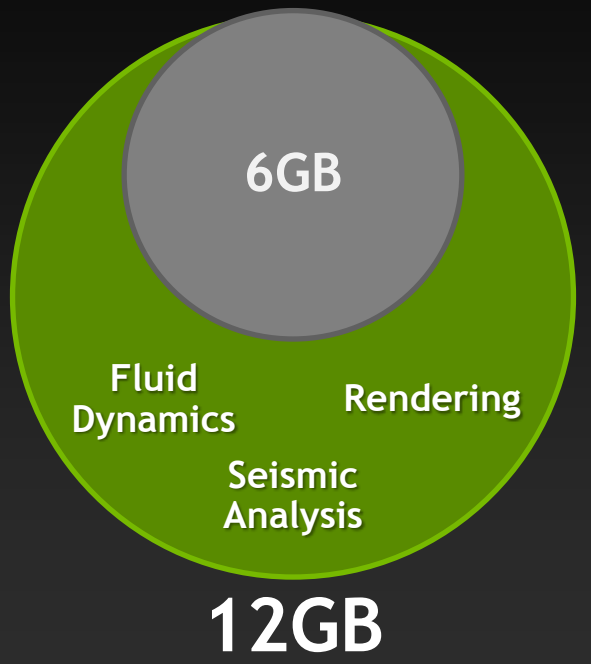
FASTER

1.4 TF| 2880 Cores | 288 GB/s



LARGER

2x Memory Enables More Apps



SMARTER

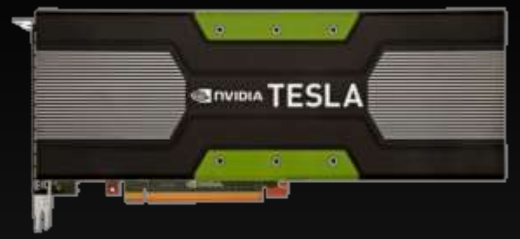
Unlock Extra Performance Using Power Headroom

GPU Boost

AMBER Benchmark: SPFP-Nucleosome
CPU: Dual E5-2687W @ 3.10GHz, 64GB System Memory, CentOS 6.2, GPU systems: Single Tesla K20X or Single Tesla K40

Tesla Kepler Family

World's Fastest and Most Efficient HPC Accelerators



	GPUs	Single Precision Peak *(SGEMM)*	Double Precision Peak *(DGEMM)*	Memory Size	Memory Bandwidth (ECC off)	PCIe Gen	System Solution
CFD, BioChemistry, Neural Networks, High Energy Physiscs, Graph analytics, Material Science, BioInformatics, M&E	**K40**	4.29 TF *(3.22TF)*	1.43 TF *(1.33 TF)*	12 GB	288 GB/s	Gen 3	Server + Workstation
Weather & Climate, Physics, BioChemistry, CAE, Material Science	**K20X**	3.95 TF *(2.90 TF)*	1.32 TF *(1.22 TF)*	6 GB	250 GB/s	Gen 2	Server only
	K20	3.52 TF *(2.61 TF)*	1.17 TF *(1.10 TF)*	5 GB	208 GB/s	Gen 2	Server + Workstation
Image, Signal, Video, Seismic	**K10**	4.58 TF	0.19 TF	8 GB	320 GB/s	Gen 3	Server only

Develop on GeForce, Deploy on Tesla

GeForce GPUs

Tesla K40/K20X/K20

Designed for Gamers & Developers

Available Everywhere

https://developer.nvidia.com/cuda-gpus

Designed for Cluster Deployment

ECC
24x7 Runtime
GPU Monitoring
Cluster Management
GPUDirect-RDMA
Hyper-Q for MPI
3 Year Warranty
Integrated OEM Systems, Professional Support

Introducing GeForce GTX TITAN

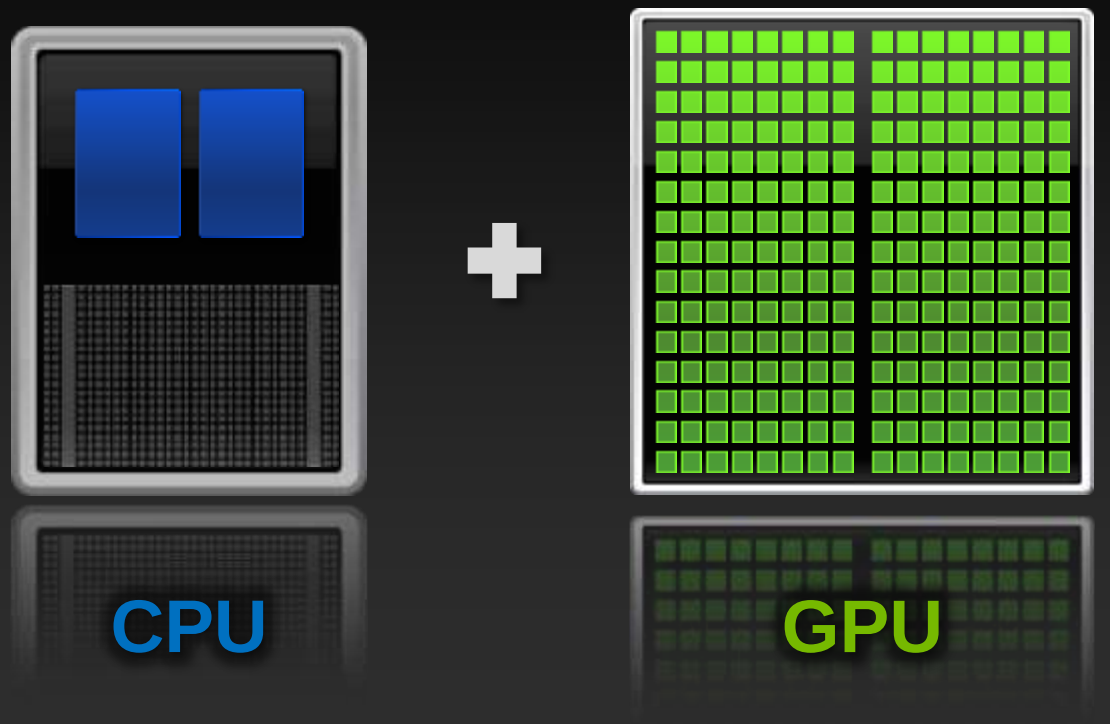
The Ultimate CUDA Development GPU

Personal Supercomputer on Your Desktop

2688	4.5	1.27	288
CUDA Cores	Teraflops Single Precision	Teraflops Double Precision	GB/s Memory Bandwidth

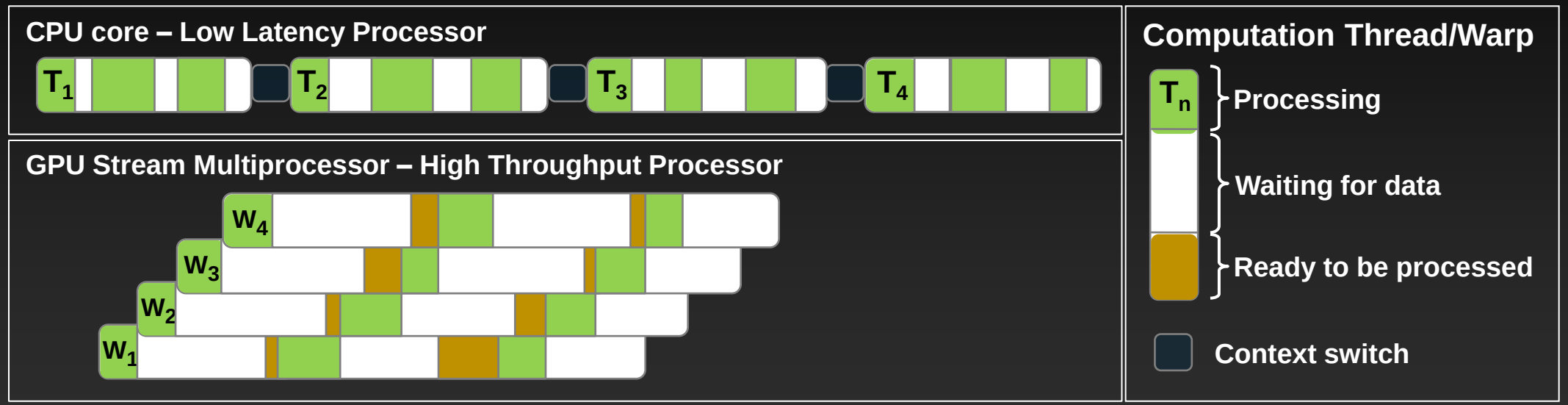
GPGPU Revolutionizes Computing

Latency Processor + Throughput processor



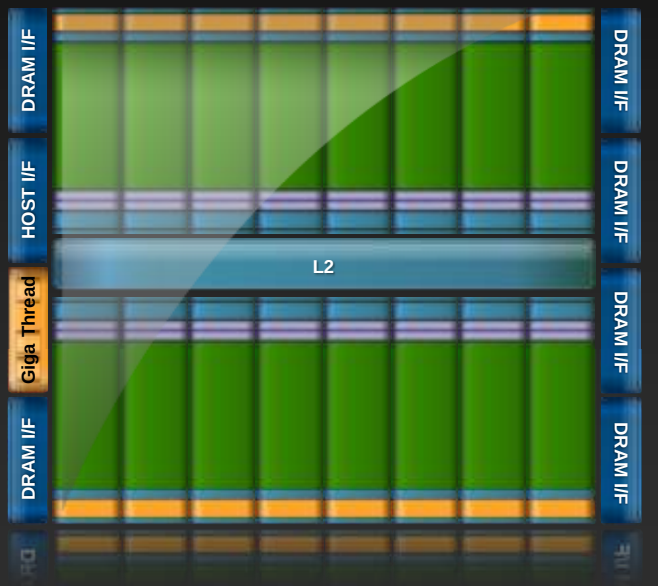
Low Latency or High Throughput?

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

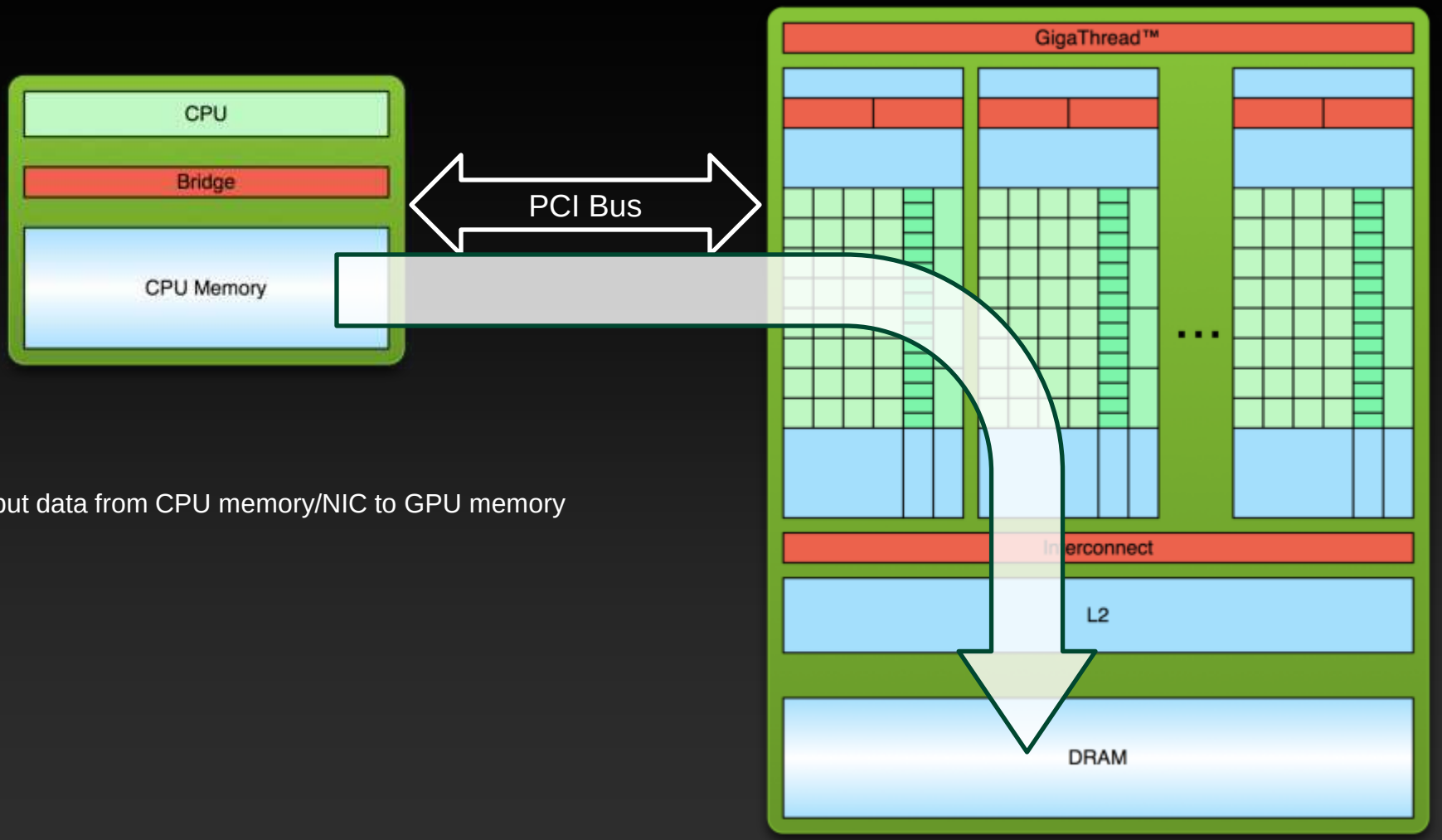


GPU Architecture: Two Main Components

- **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

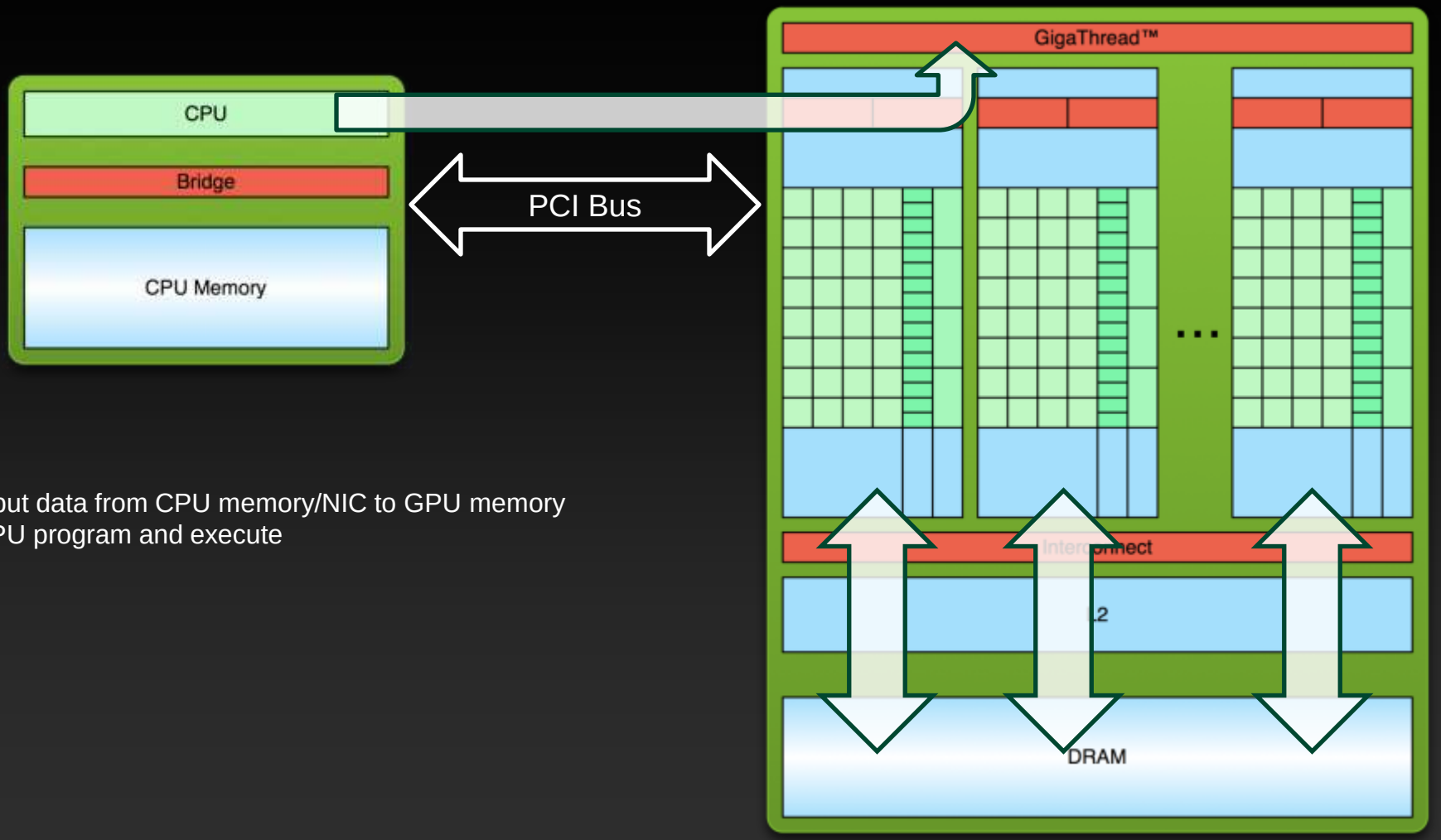


Simple Processing Flow



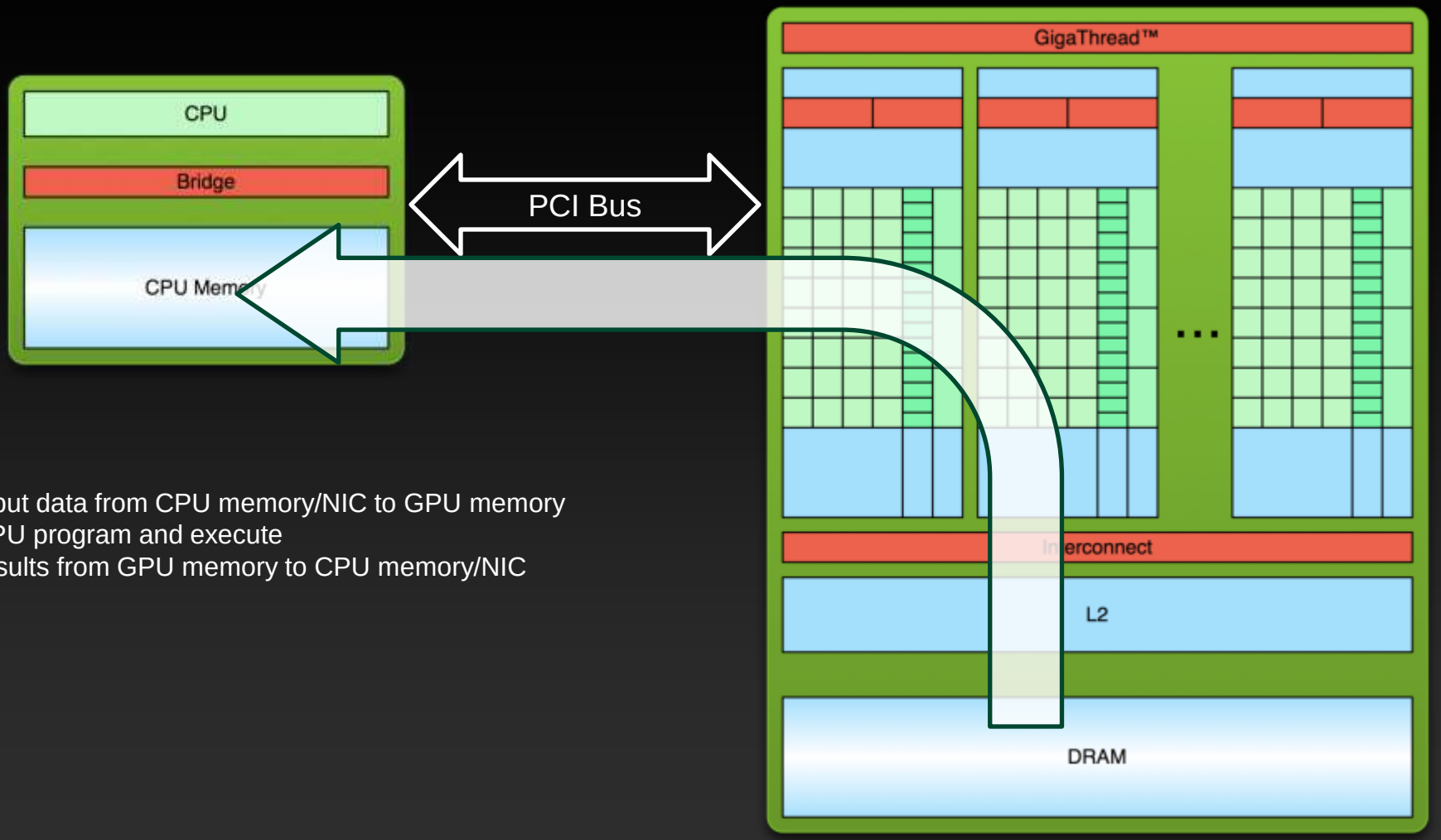
1. Copy input data from CPU memory/NIC to GPU memory

Simple Processing Flow



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

Simple Processing Flow



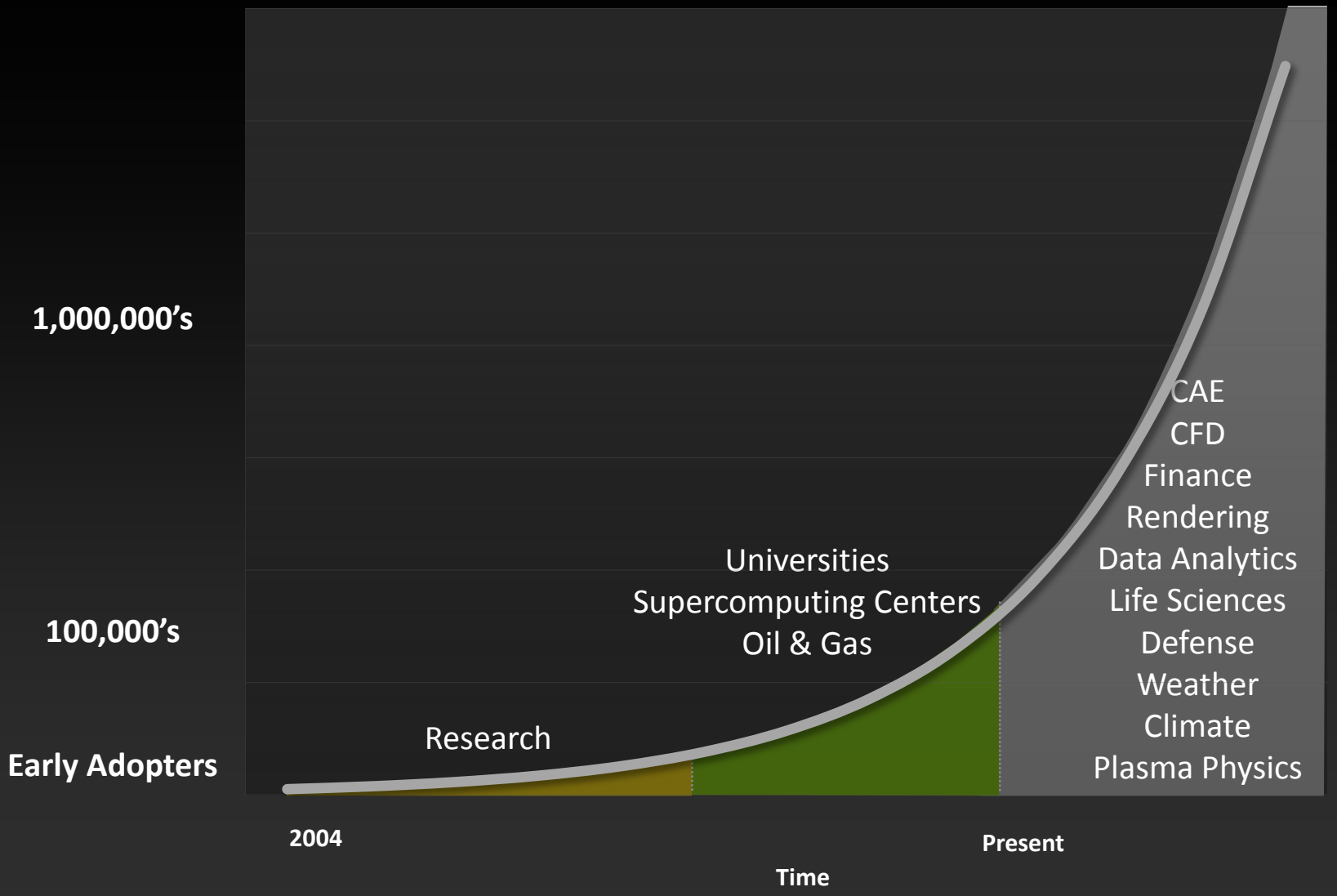
1. Copy input data from CPU memory/NIC to GPU memory
2. Load GPU program and execute
3. Copy results from GPU memory to CPU memory/NIC

Kepler GK110 Block Diagram

Architecture

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

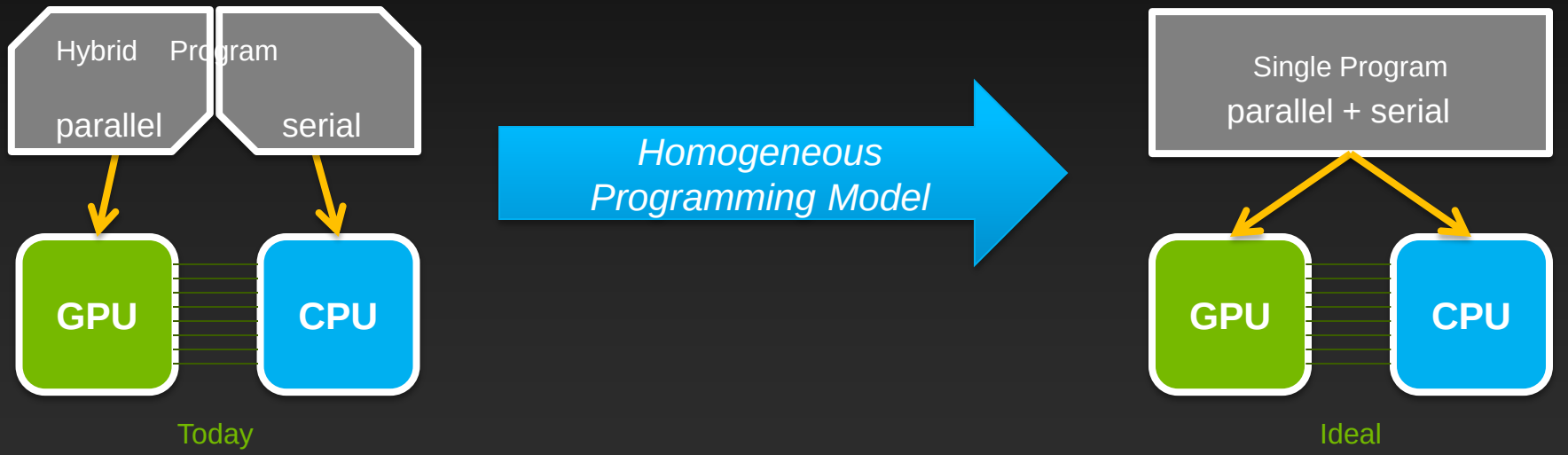
GPUs Reaching Broader Set of Developers



Programming Heterogeneous Systems

We want: *homogeneous* program, *heterogeneous* execution

- Unified programming model includes parallelism in language
- Scalable programs must be designed for parallelism



CUDA Parallel Computing Platform

www.nvidia.com/getcuda

Programming Approaches

- **Libraries** — "Drop-in" Acceleration
- **OpenACC Directives** — Easily Accelerate Apps
- **Programming Languages** — Maximum Flexibility

Development Environment

Nsight IDE
Linux, Mac and Windows
GPU Debugging and Profiling

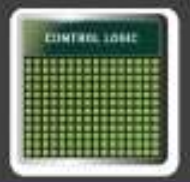
CUDA-GDB debugger
NVIDIA Visual Profiler

Open Compiler Tool Chain

Enables compiling new languages to CUDA platform, and CUDA languages to other architectures

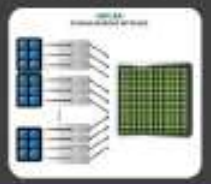
Hardware Capabilities

SMX



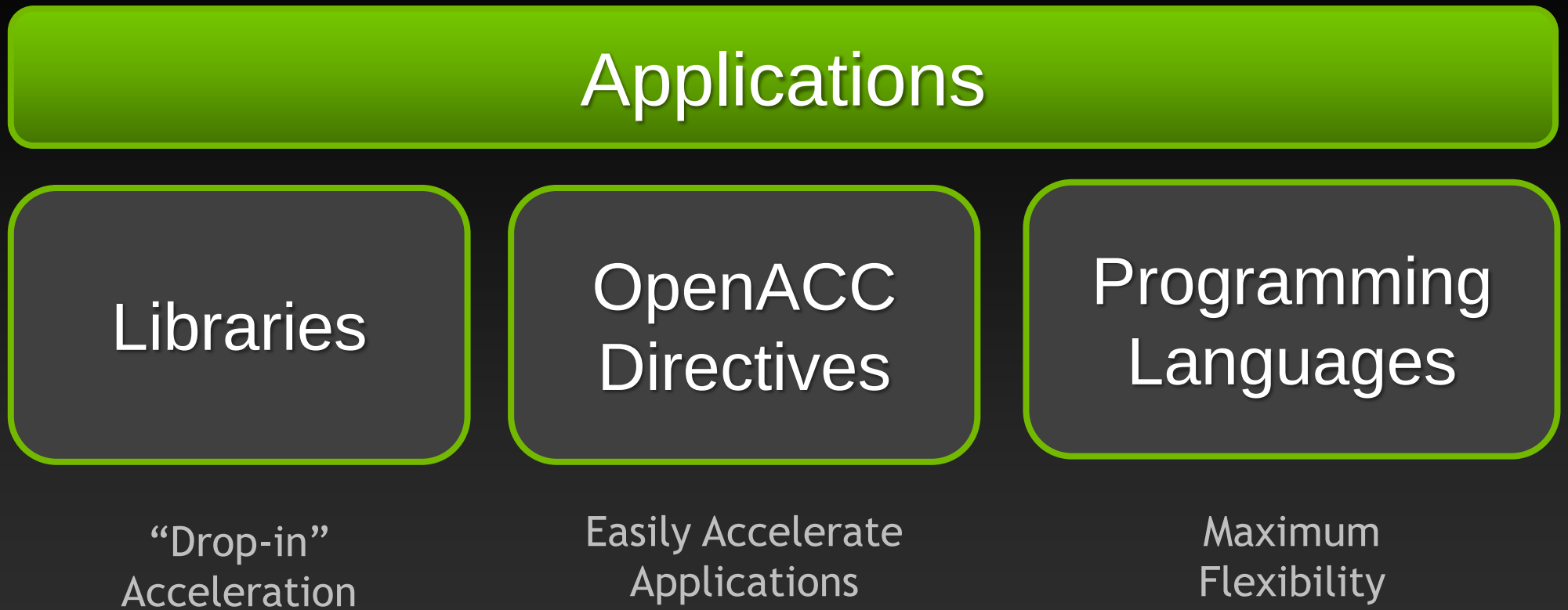
Dynamic Parallelism

HyperQ

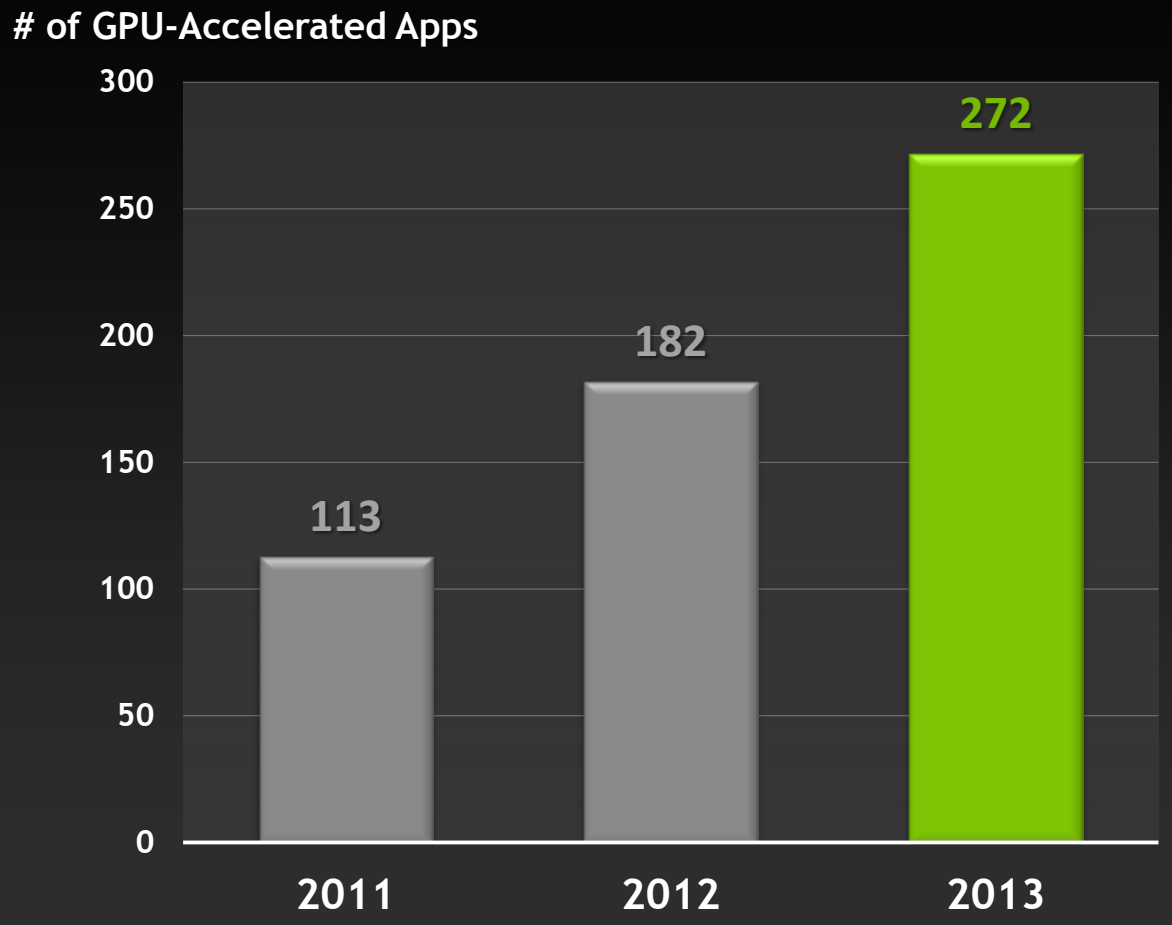


GPUDirect

4 Ways to Accelerate Applications



Solid Growth of GPU Accelerated Apps



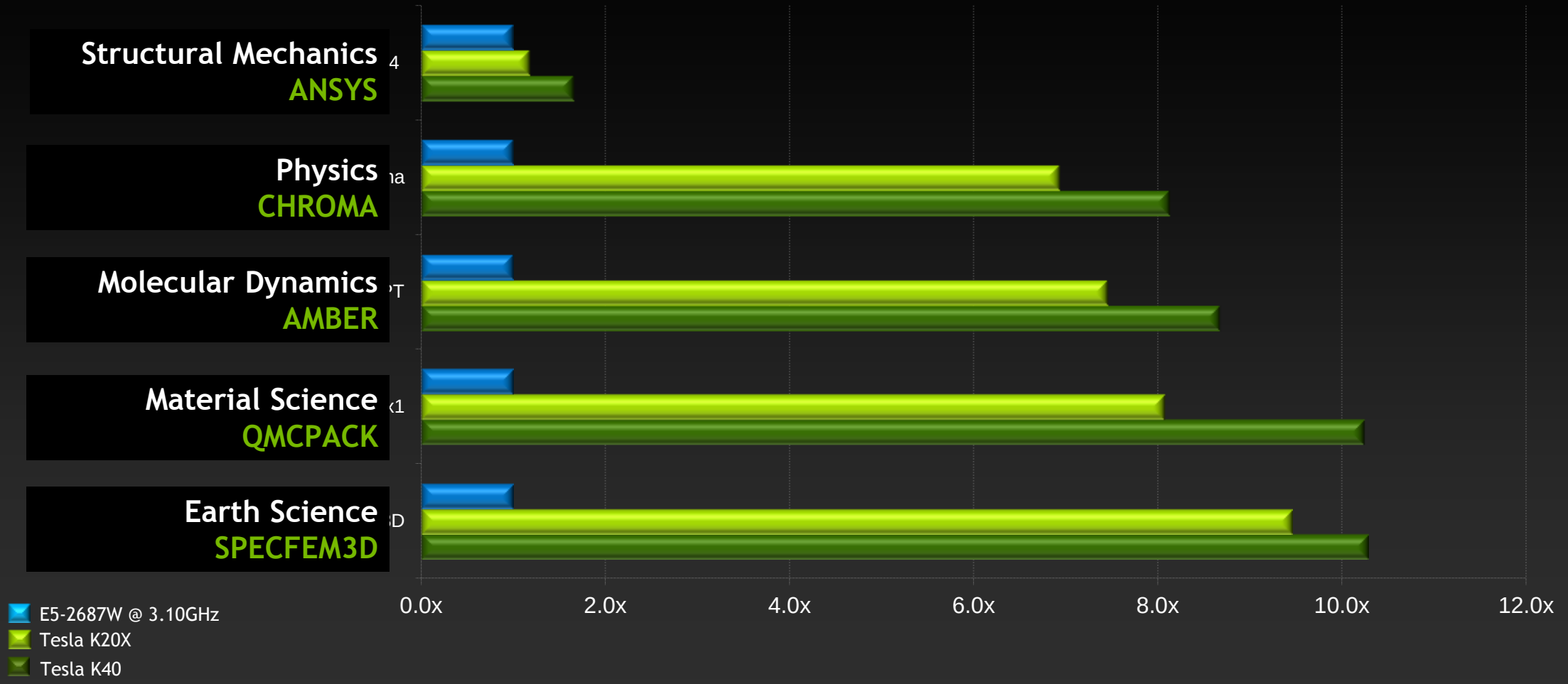
Top HPC Applications

Molecular Dynamics	AMBER CHARMM DESMOND	GROMACS LAMMPS NAMD
Quantum Chemistry	Abinit Gaussian	GAMESS NWChem
Material Science	CP2K QMCPACK	Quantum Espresso VASP
Weather & Climate	COSMO GEOS-5 HOMME	CAM-SE NEMO NIM WRF
Lattice QCD	Chroma	MILC
Plasma Physics	GTC	GTS
Structural Mechanics	ANSYS Mechanical LS-DYNA Implicit MSC Nastran	OptiStruct Abaqus/Standard
Fluid Dynamics	ANSYS Fluent	Culises (OpenFOAM)

Accelerated, In Development


NVIDIA
POPULAR GPU-ACCELERATED APPLICATIONS
02 Research: Higher Education and Supercomputing
06 Defense and Intelligence
07 Computational Finance
08 Manufacturing: CAD and CAE
10 Media and Entertainment
14 Oil and Gas
Research: Higher Education and Supercomputing
COMPUTATIONAL CHEMISTRY AND BIOLOGY
Bioinformatics
Molecular Dynamics
Defense and Intelligence
Computational Finance
Manufacturing: CAD and CAE
Media and Entertainment
Oil and Gas
272 GPU-Accelerated Applications
www.nvidia.com/appscatalog

Performance on Leading Scientific Applications



Ways to Accelerate Applications

Applications

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

GPU Accelerated Libraries

"Drop-in" Acceleration for your Applications

Linear Algebra
FFT, BLAS,
SPARSE, Matrix

NVIDIA cuFFT, cuBLAS, cuSPARSE

CULA tools

MAGMA

CUSP

Numerical & Math
RAND, Statistics

IMSL Fortran Numerical Library

NVIDIA Math Lib

ArrayFire

NVIDIA cuRAND

Data Struct. & AI
Sort, Scan, Zero Sum

Thrust

GPU AI – Board Games

ARCHIA

GPU AI – Path Finding

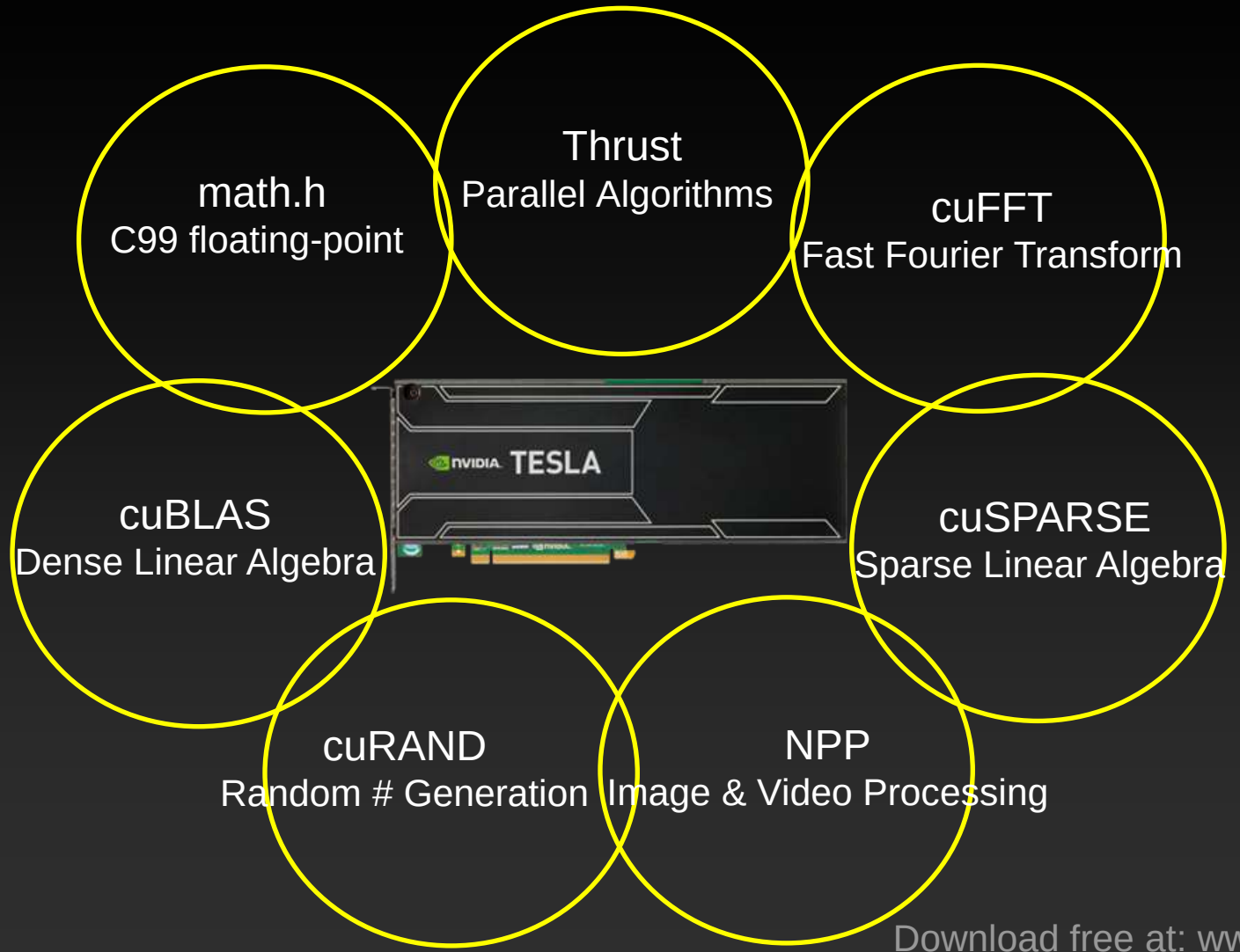
Visual Processing
Image & Video

NVIDIA NPP

NVIDIA Video Encode

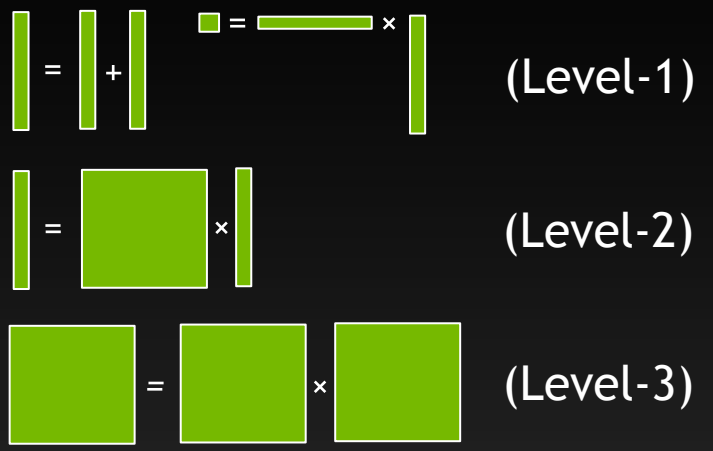
Sundog Software

CUDA Accelerated Compute Libraries



cuBLAS Library

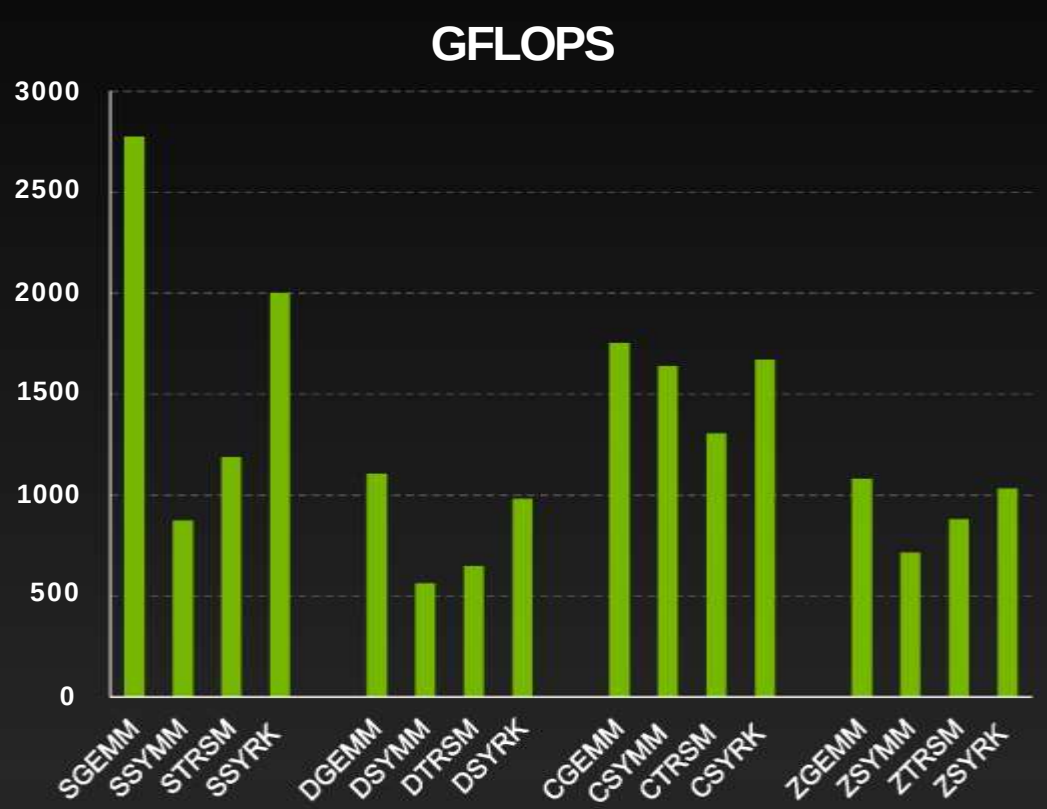
- Dense Linear Algebra
  - Single and double precision
  - Real and complex data types
  - Vector- and matrix-vector operations
  - Matrix-matrix operations

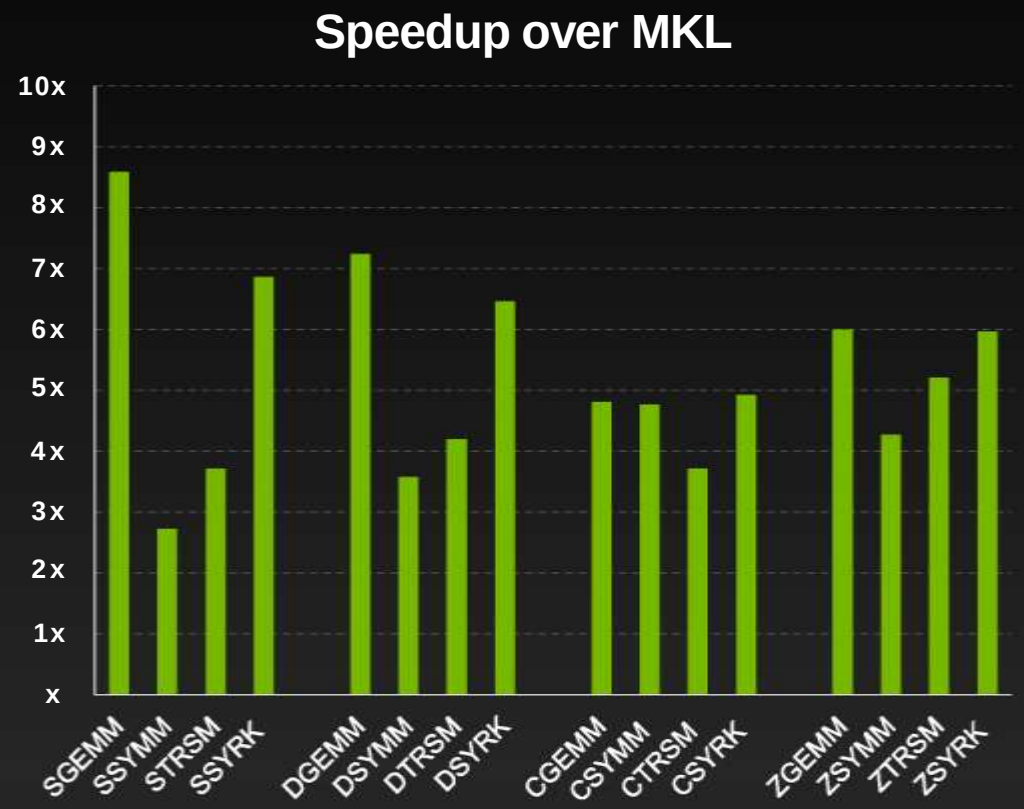


- Interface
  - Similar to Basic Linear Algebra Subprograms (BLAS)
  - Supports dynamic parallelism (on K20)

CUBLAS Level 3 Performance

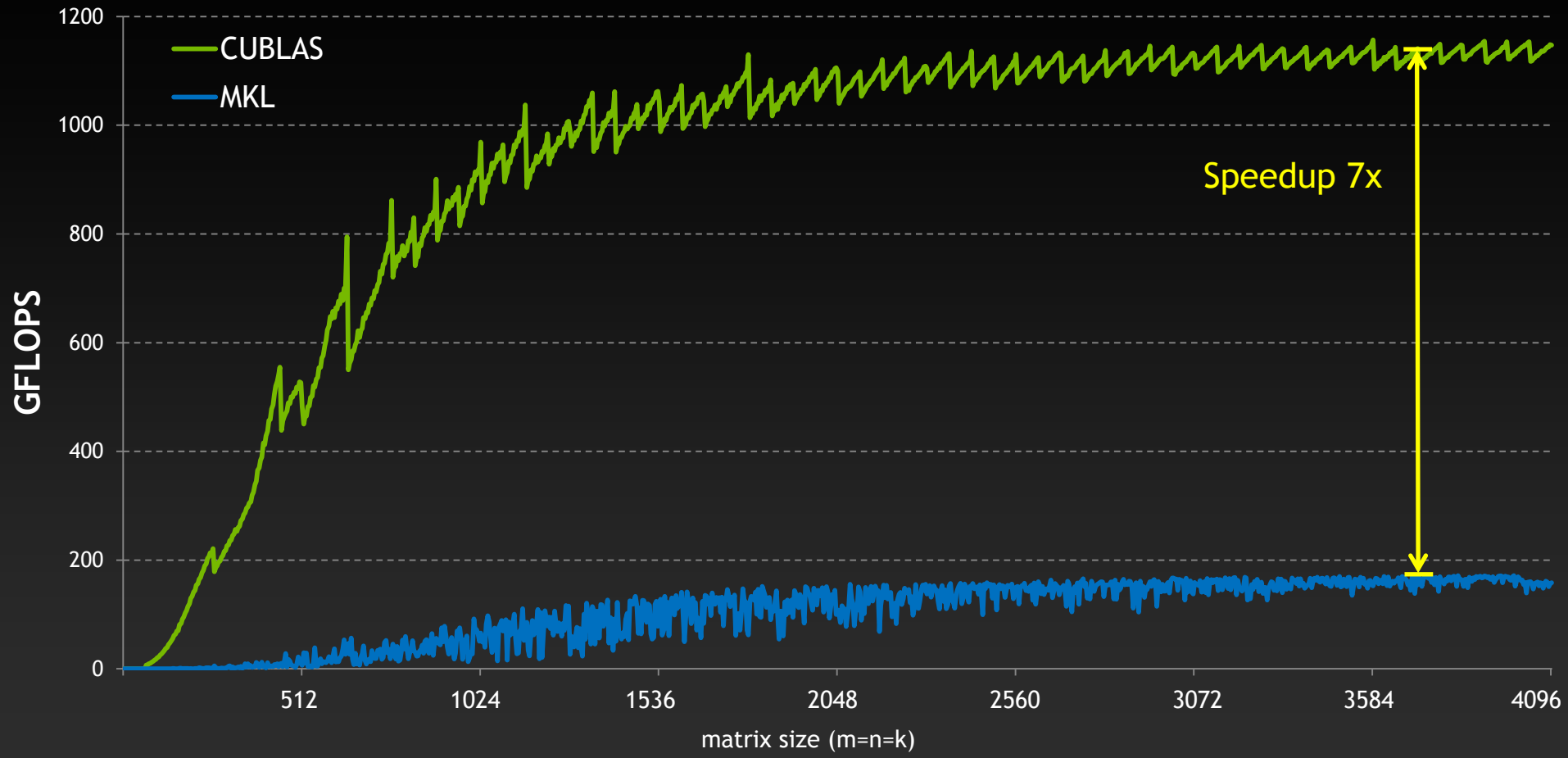
Up to 1 TFLOPS sustained performance and 7x faster than Intel MKL





- cuBLAS 5.0 on K20X, input and output data on device
- MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

DGEMM Performance



- cuBLAS 5.0 on K20X, input and output data on device
- MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

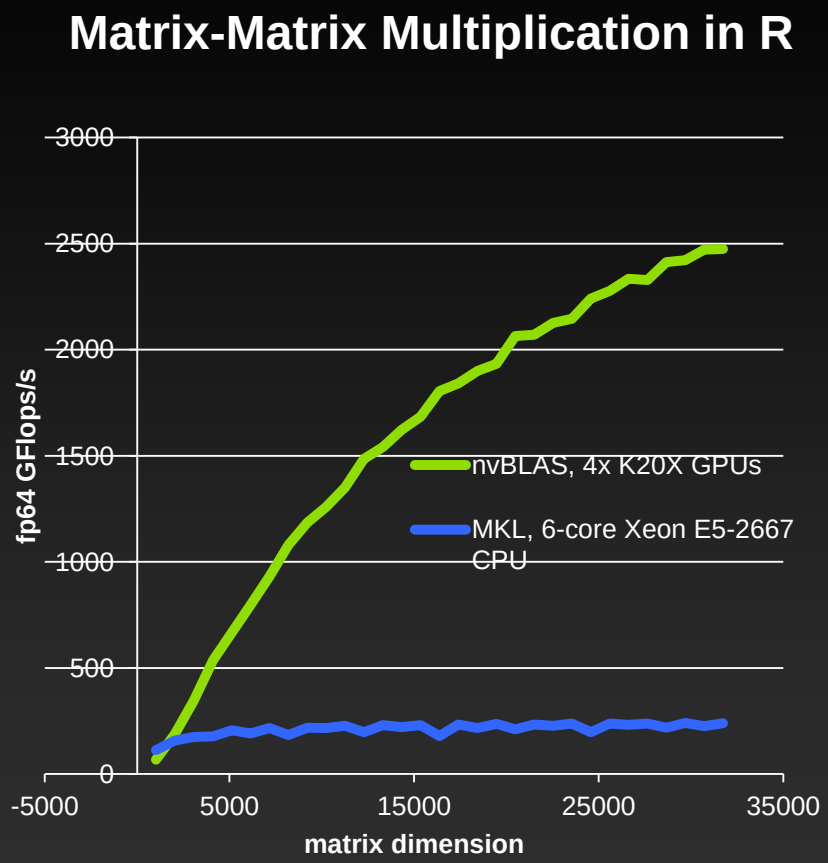
Performance may vary based on OS version and motherboard configuration

New Drop-in NVBLAS Library

- Drop-in replacement for CPU-only BLAS
  - Automatically route BLAS3 calls to cuBLAS
- Example: Drop-in Speedup for R

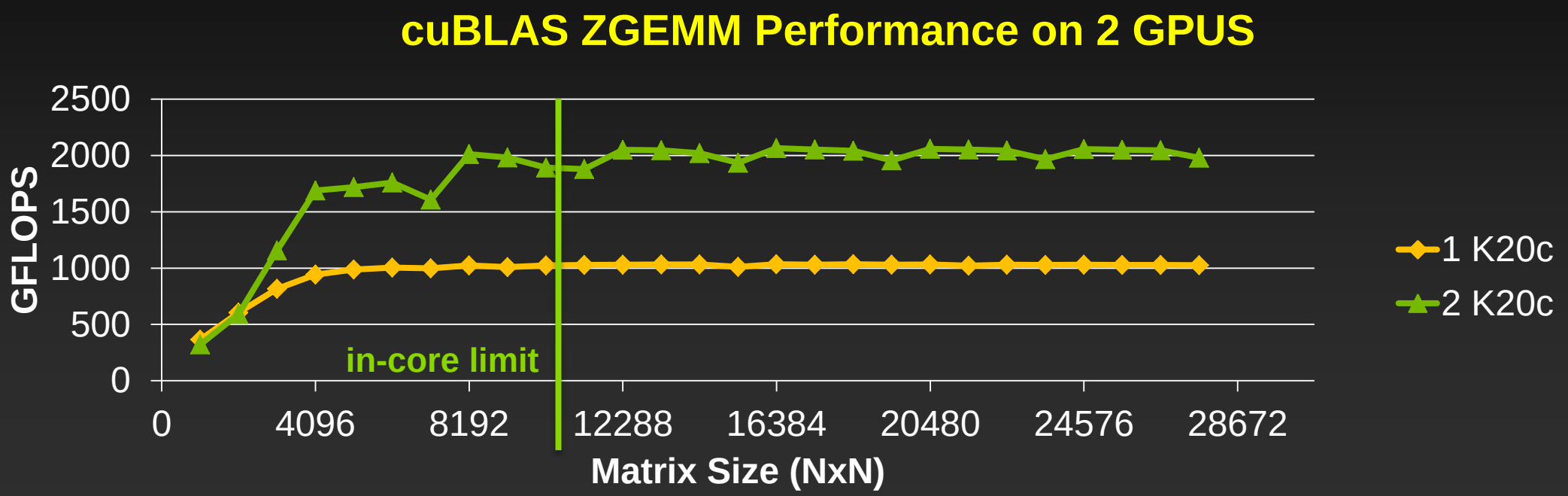
```
> LD_PRELOAD=/usr/local/cuda/lib64/libnvblas.so R
> A <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> B <- matrix(rnorm(4096*4096), nrow=4096, ncol=4096)
> system.time(C <- A %*% B)
   user  system elapsed
  0.348   0.142   0.289
```

- Use in any app that uses standard BLAS3
  - Octave, Scilab, etc.



Multi-GPU cuBLAS

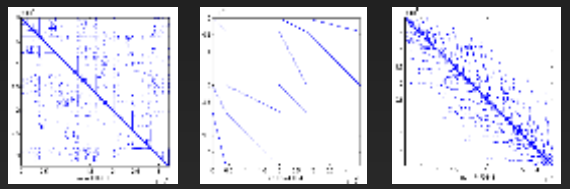
- Single function call automatically spreads work across two GPUs
- Source and result data in system memory
- Supports matrices > size of memory (out-of-core)
- All BLAS Level-3 routines



cuSPARSE : Sparse linear algebra routines

- Features
  - Format conversion (dense, CSR, block-CSR, HYB, COO, CSC...)
  - Sparse-dense (matrix-vector multiply and triangular solve)
  - Sparse-sparse (matrix-matrix add and multiply)
  - Preconditioners (incomplete-LU & Cholesky, tridiagonal, ...)

- Interface
  - C API with Fortran wrappers

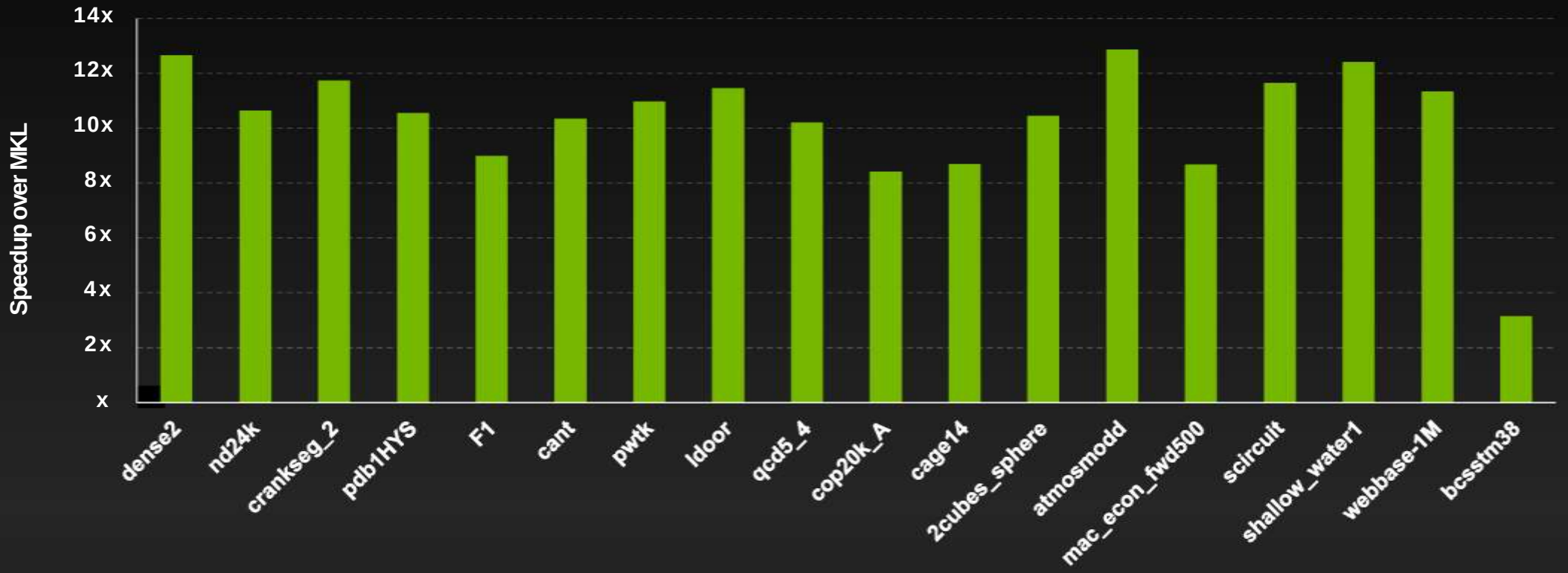


different matrix sparsity patterns

cuSPARSE: up to 12x Faster than MKL

Sparse Matrix x 6 Dense Vectors

(useful for block iterative solvers)

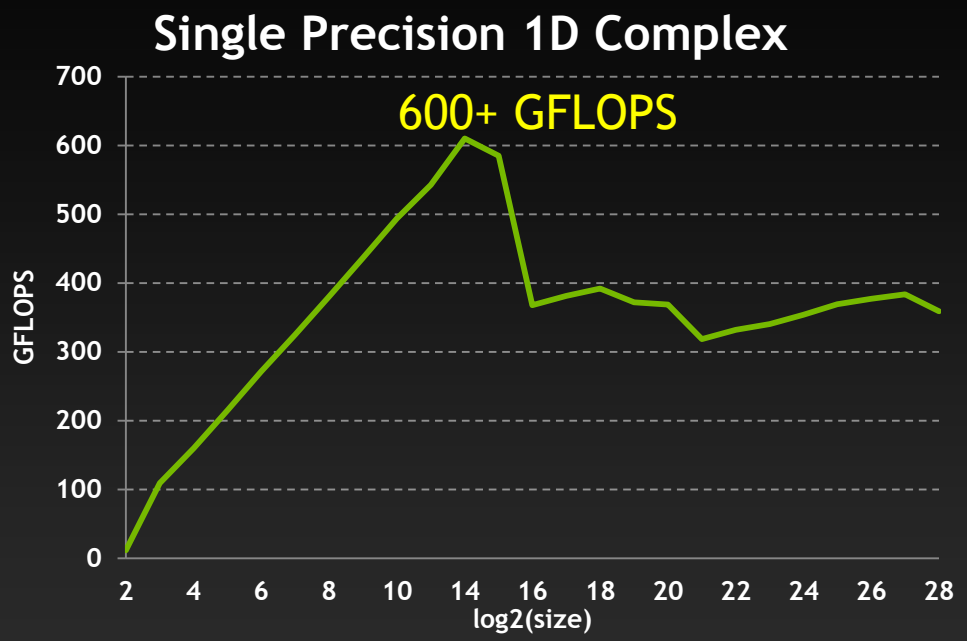


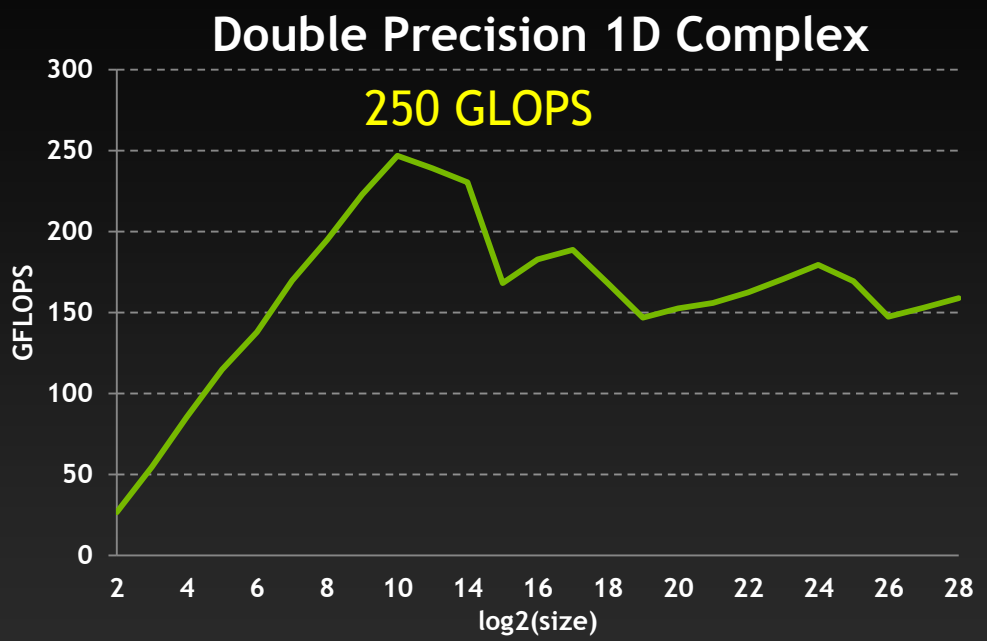
- Average of s/d/c/z routines
- CUSPARSE 5.0 on K20X, input and output data on device

MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

cuFFT Performance

1D used in audio processing and as a foundation for 2D and 3D FFTs



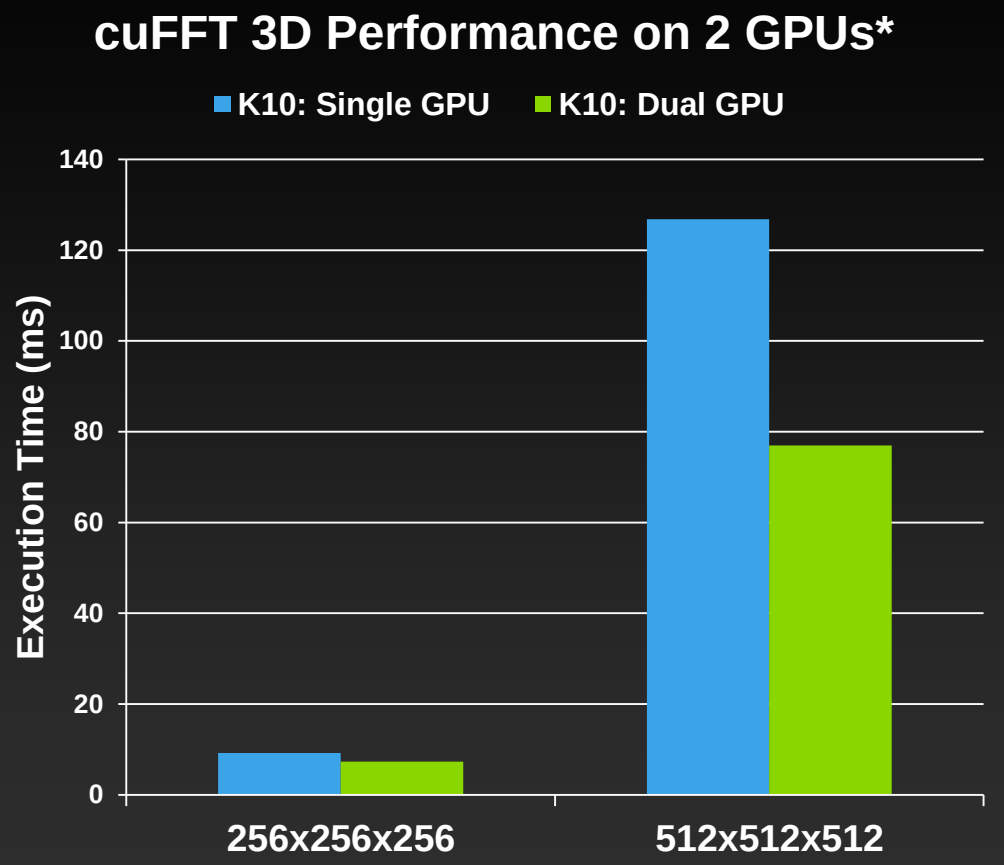


Performance may vary based on OS version and motherboard configuration

• cuFFT 5.0 on K20X, input and output data on device

Multi-GPU cuFFT

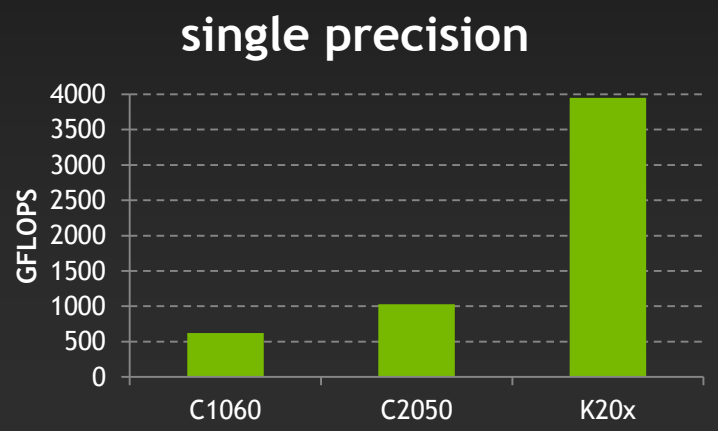
- Single & Batch Transforms across multiple GPUs (max 2 in CUDA 6)
- Tuned for multi-GPU cards (K10)
  - Better scaling for larger transforms

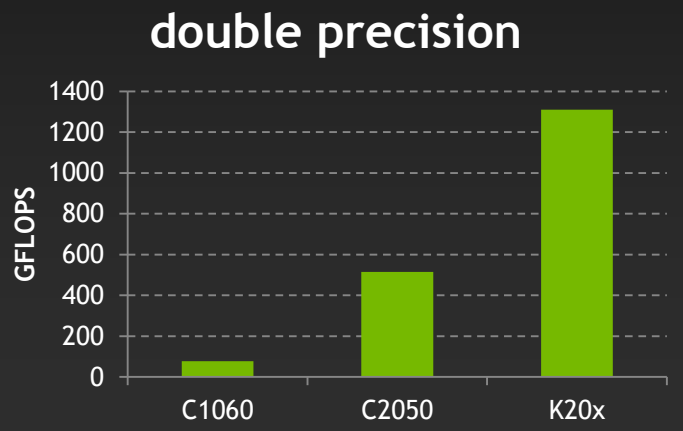


*Does not include memcpy time

math.h Library

- C99 floating point operations + extras
  - IEEE-754 accurate single and double (+, *, fma, /, ...)
  - Exponential (exp, log, ...)
  - Trigonometric (sin, cos, tan, ...)
  - Special (lgamma, tgamma, erf, erfc, ...)
- Peak (Theoretical) Performance



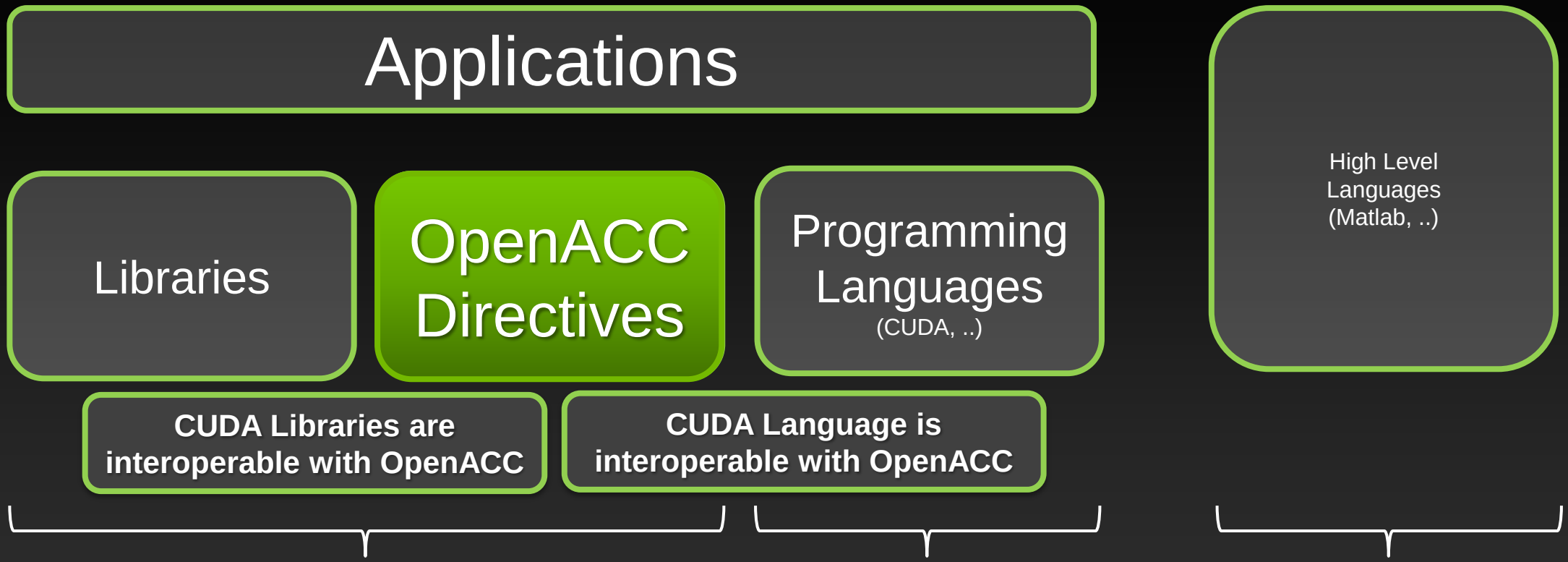


Explore the CUDA (Libraries) Ecosystem

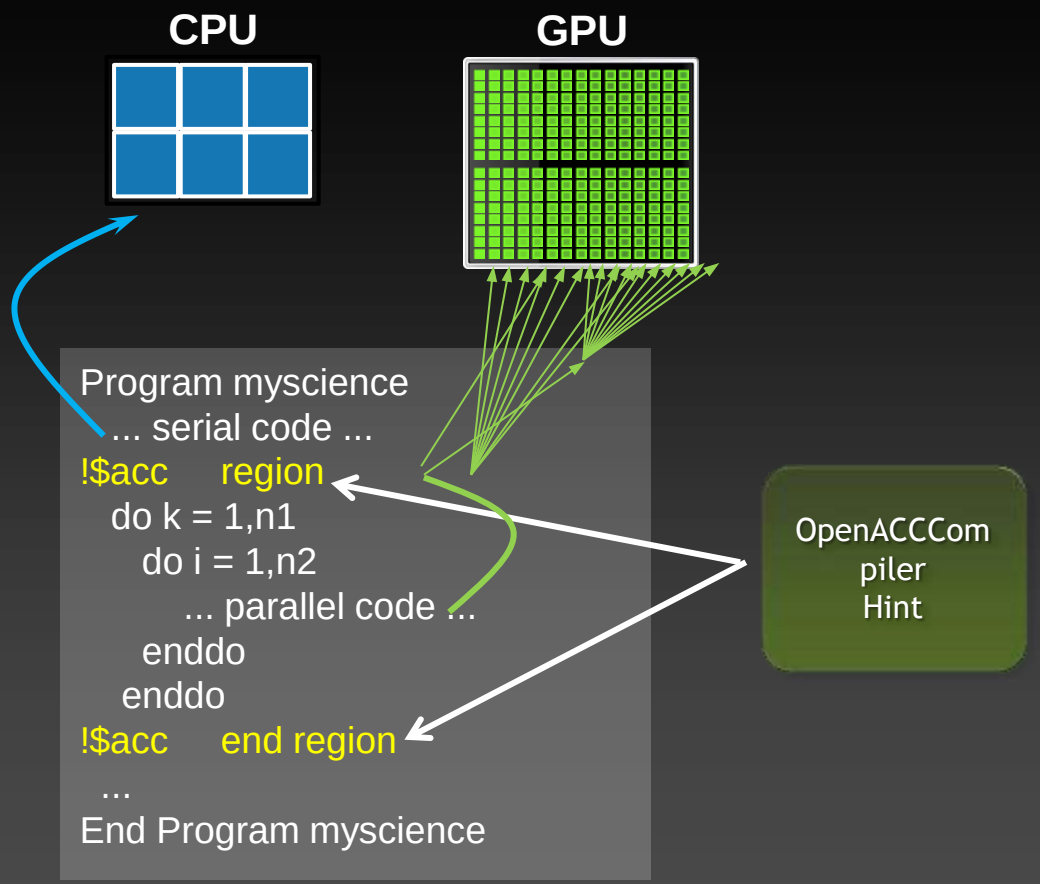
- CUDA Tools and Ecosystem described in detail on NVIDIA Developer Zone:

  developer.nvidia.com/cuda-tools-ecosystem

Ways to Accelerate Applications



OpenACC Directives



Your original Fortran or C code

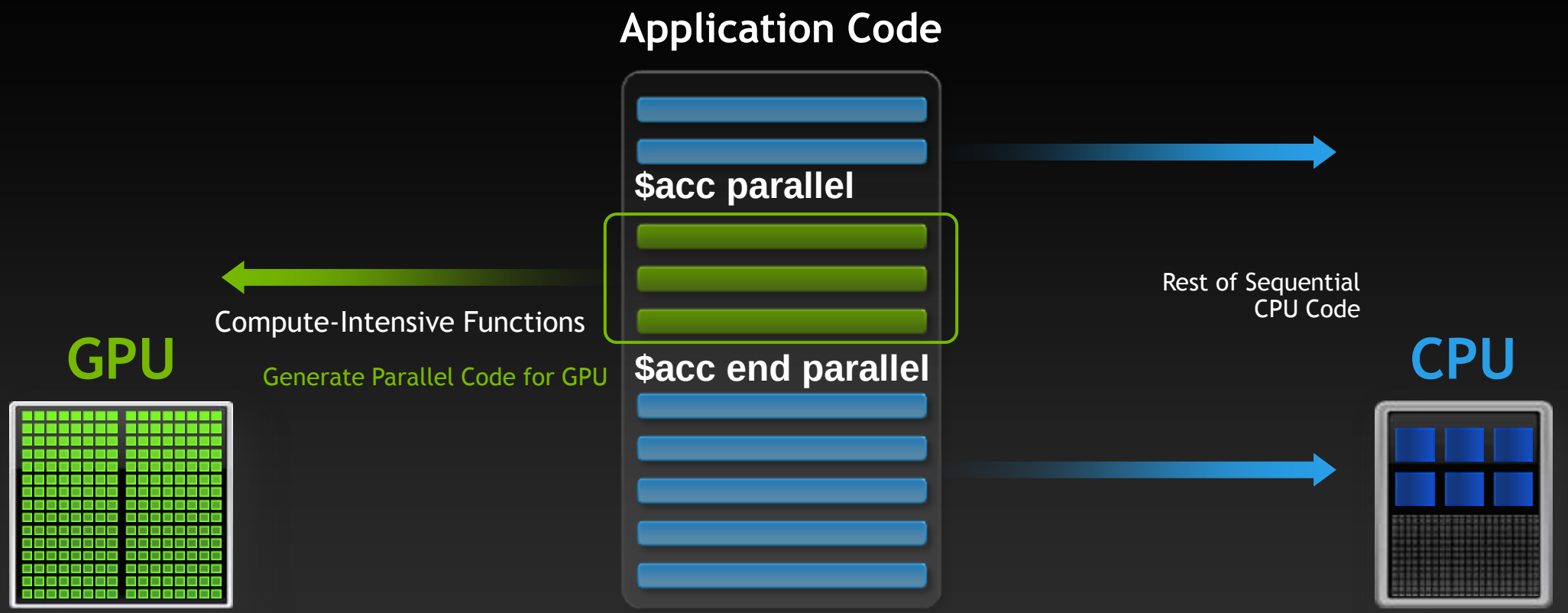
http://www.openacc.org

PGI CAPS CRAY NVIDIA allinea OAK RIDGE Sandia National Laboratories ROGUE WAVE SOFTWARE CSCS TECHNISCHE UNIVERSITÄT DRESDEN

Easy, Open, Powerful

- Simple Compiler hints
- Works on multicore CPUs & many core GPUs
- Compiler Parallelizes code
- Future Integration into OpenMP standard planned

OpenACC Execution Model

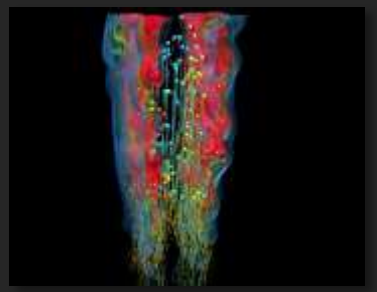


Focus on Exposing Parallelism

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



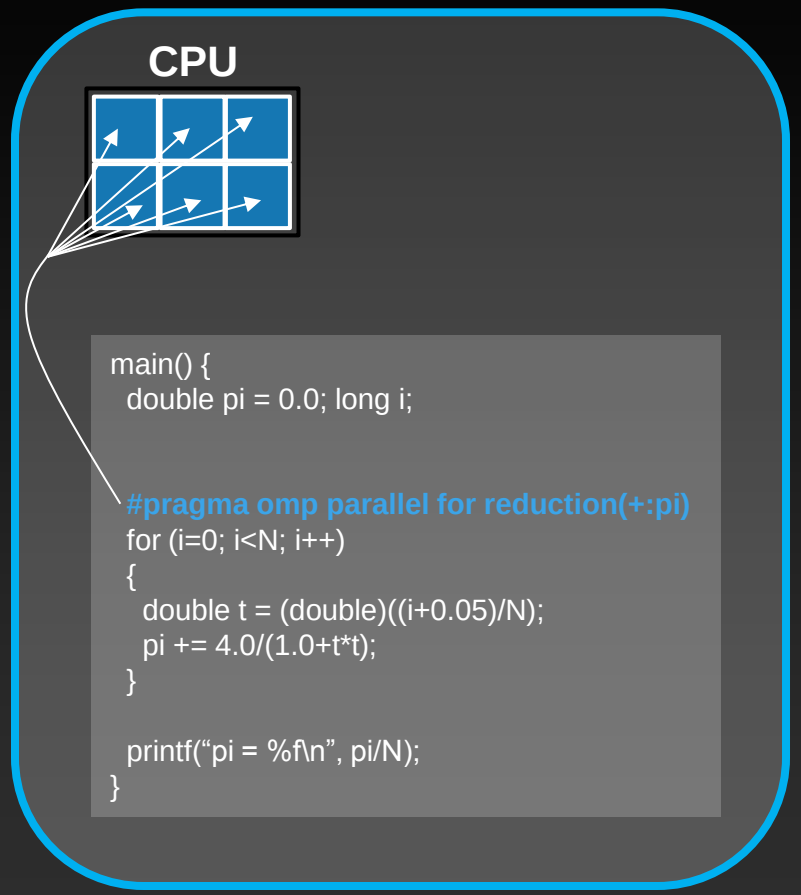
CAM-SE
Answer questions about specific climate change adaptation and mitigation scenarios

- **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%**

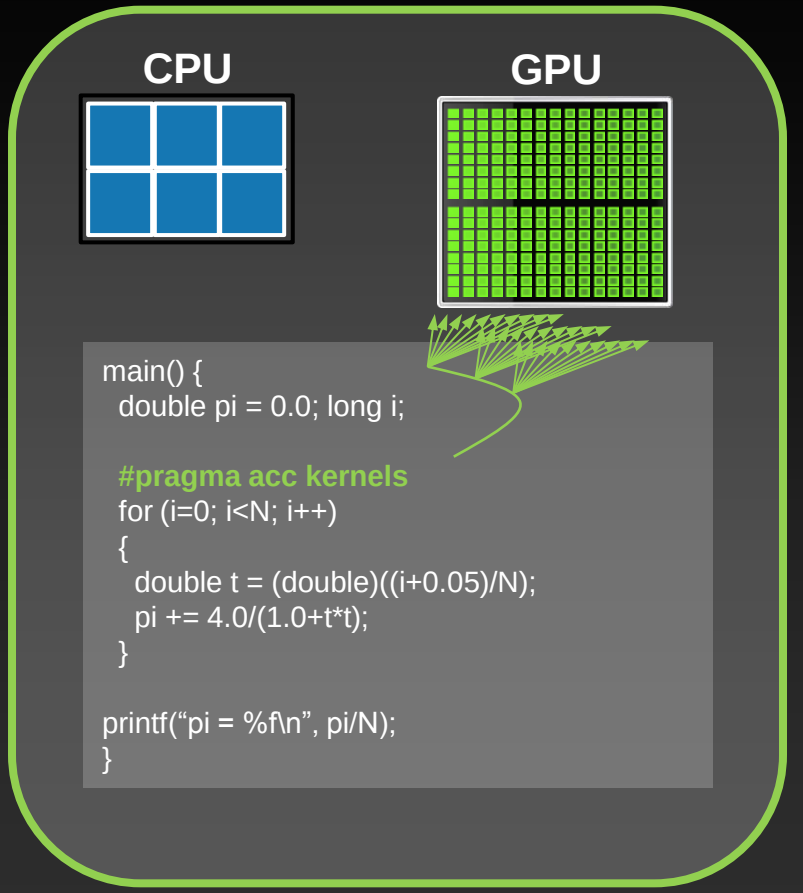
- **Tuning top key kernel (50% of runtime)**
- **6.5x faster on CPU+GPU vs. CPU+CPU**
- **Improved performance of CPU version by 100%**

Familiar to OpenMP Programmers

OpenMP



OpenACC



Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems

5x in 40 Hours

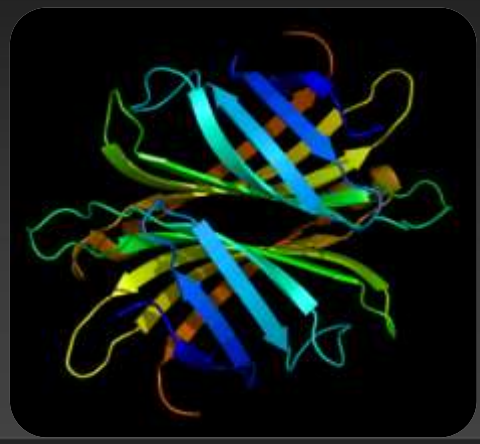
Valuation of Stock Portfolios using Monte Carlo

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

OpenACC Accelerates Science

Weather Prediction

Swiss National Weather Agency

COSMO (Physics)

4.2x

Chemistry Research

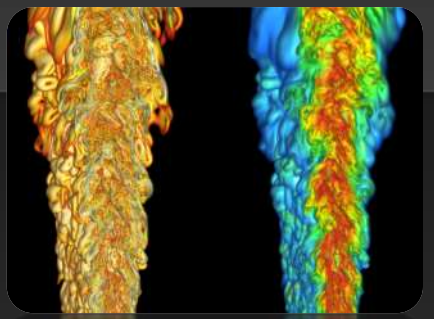
Blue Waters @ NCSA

GAMESS CCSD

3.1x

Fuel Efficiency

National Renewable Energy Lab

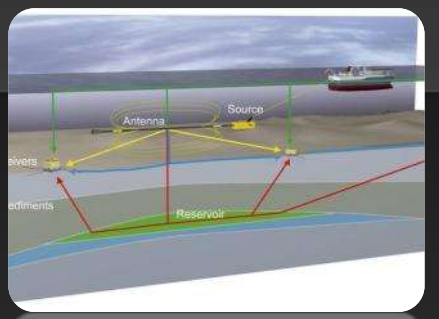


S3D

4.2x

Oil Exploration

EMGS



ELAN

3.2x

Performance Speed-up:
2x AMD6274 CPUs vs 1 AMD6274 CPU + 1 Tesla K20X

A Champion Case

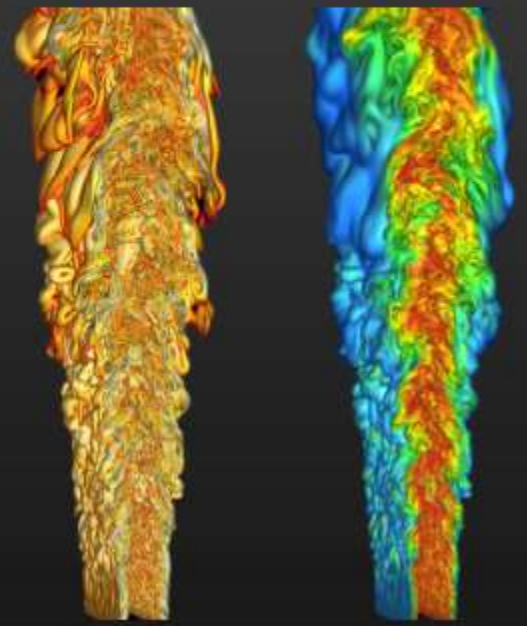
4x Faster

Jaguar	Titan
42 days	10 days

Modified <1%
Lines of Code

Was recently the fastest scientific simulation ever done. 15 PF!

Design alternative fuels with up to 50% higher efficiency

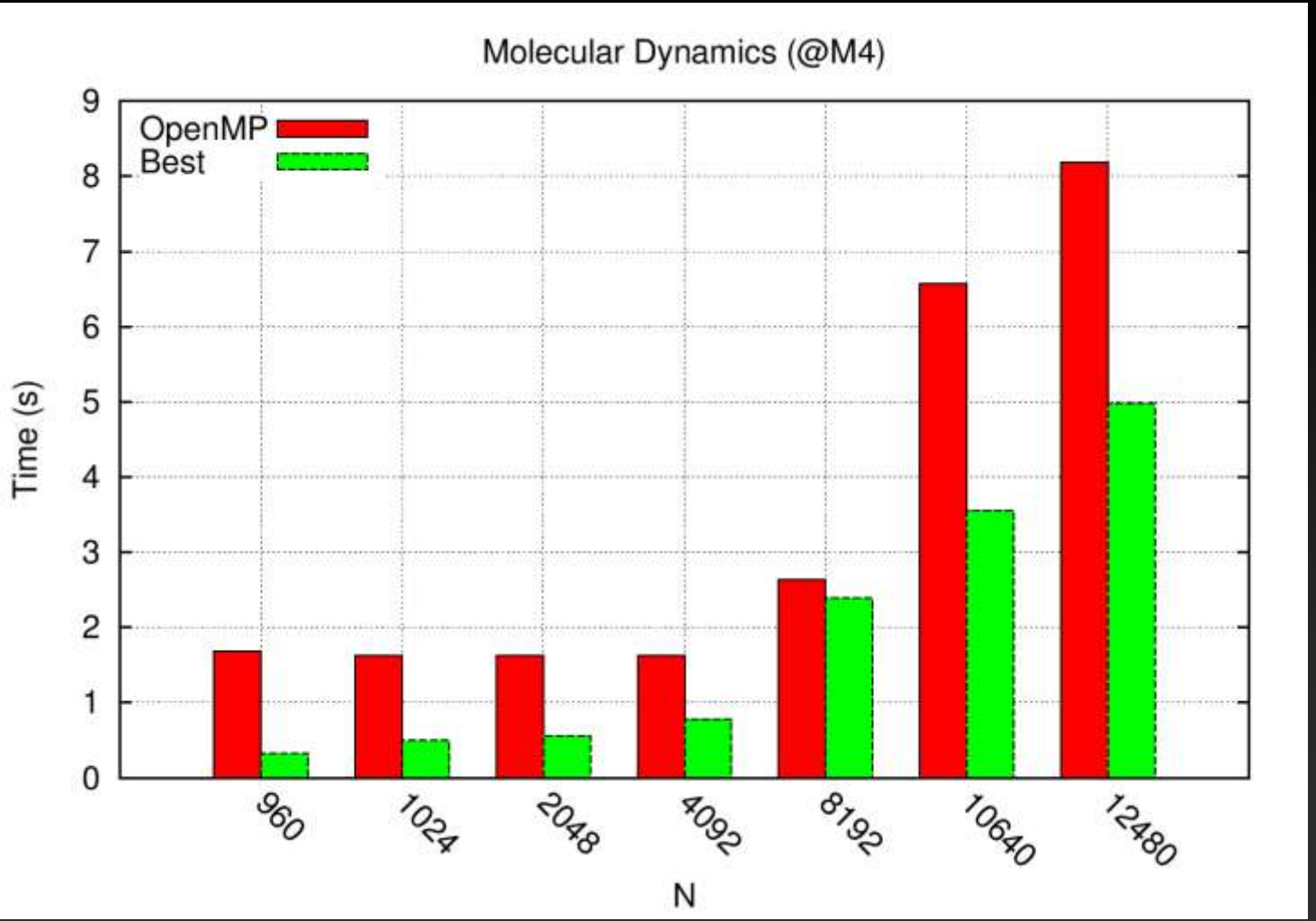


S3D: Fuel Combustion

PITTSBURGH SUPERCOMPUTING CENTER

ACCULL The OpenACC research implementation

(http://accull.wordpress.com/2012/05/)



OpenACC
DIRECTIVES FOR ACCELERATORS

Linux GCC Compiler to Support GPU Accelerators

Open Source

GCC Efforts by Samsung & Mentor Graphics

Pervasive Impact

Free to all Linux users

Mainstream

Most Widely Used HPC Compiler

> *Incorporating OpenACC into GCC is an excellent example of open source and open standards working together to make accelerated computing broadly accessible to all Linux developers.*

Oscar Hernandez
Oak Ridge National Laboratories

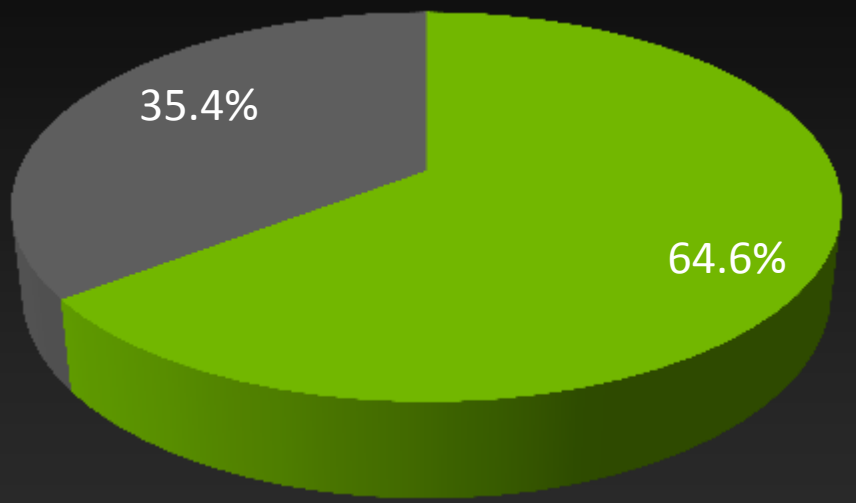
OAK RIDGE National Laboratory

OpenACC Directives for Accelerators

OpenACC Survey

Survey Says Easy to Program, Great Performance

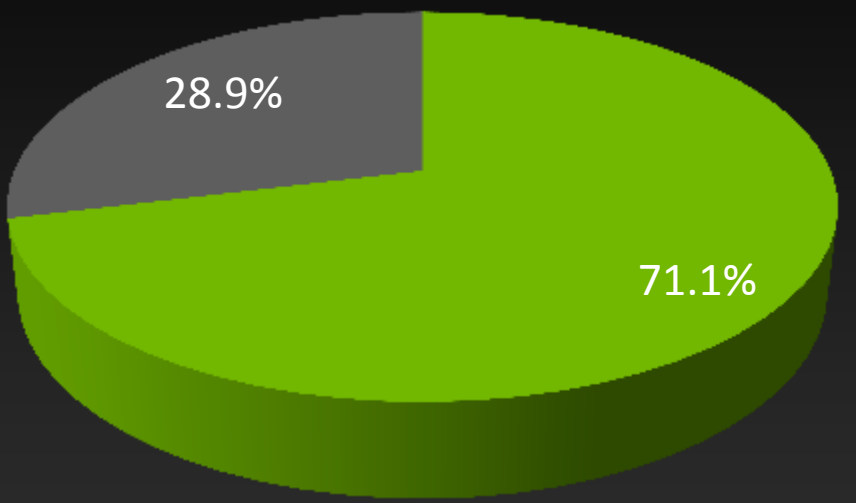
Application Acceleration



2X or Higher Acceleration

< 2X Acceleration

Easy to Program



Somewhat/Extremely Easy

Somewhat/Extremely Difficult

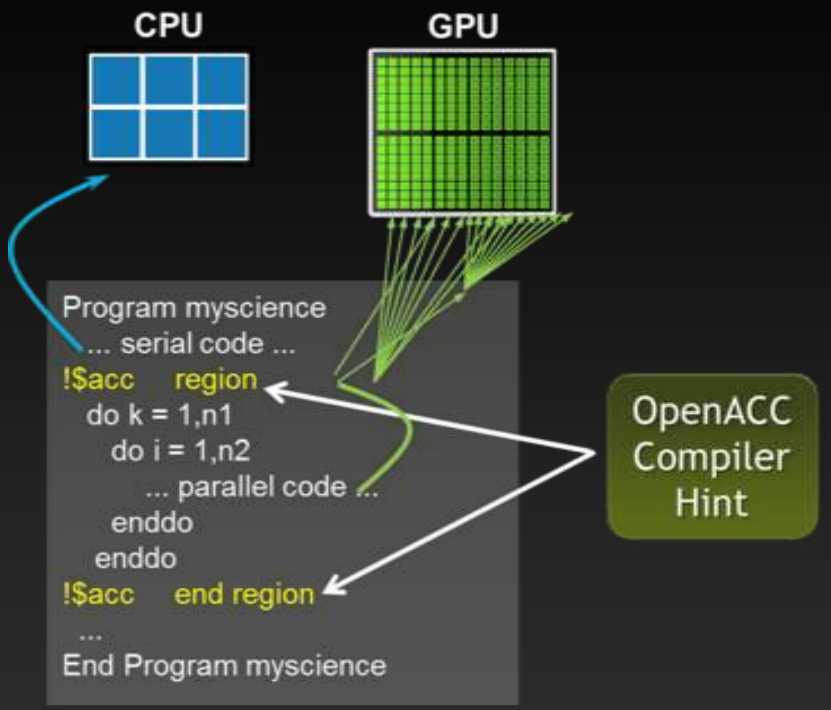
Survey done in May 2013
Over 160 Responses

Additions for OpenACC 2.0

OpenACC
DIRECTIVES FOR ACCELERATORS
NVIDIA

- Procedure calls
- Separate compilation
- Nested parallelism
- Device-specific tuning, multiple devices
- Data management features and global data
- Multiple host thread support
- Loop directive additions
- Asynchronous behavior additions
- New API routines for target platforms

(CUDA, OpenCL, Intel Coprocessor Offload Infrastructure)



See http://www.openacc.org/sites/default/files/OpenACC-2.0-draft.pdf

OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online

  http://www.openacc-standard.org

- Quick reference card also available

- Beta implementations available now from PGI, Cray, and CAPS

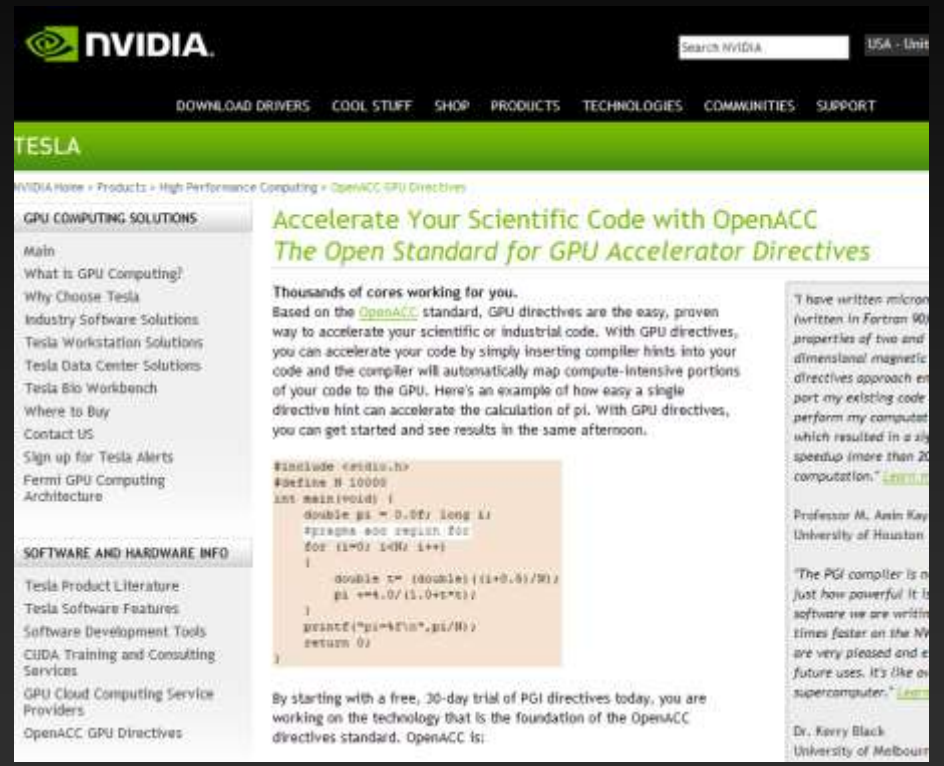
Start Now with OpenACC Directives

Sign up for a free trial of the directives compiler now!

Free trial license to PGI Accelerator

Tools for quick ramp

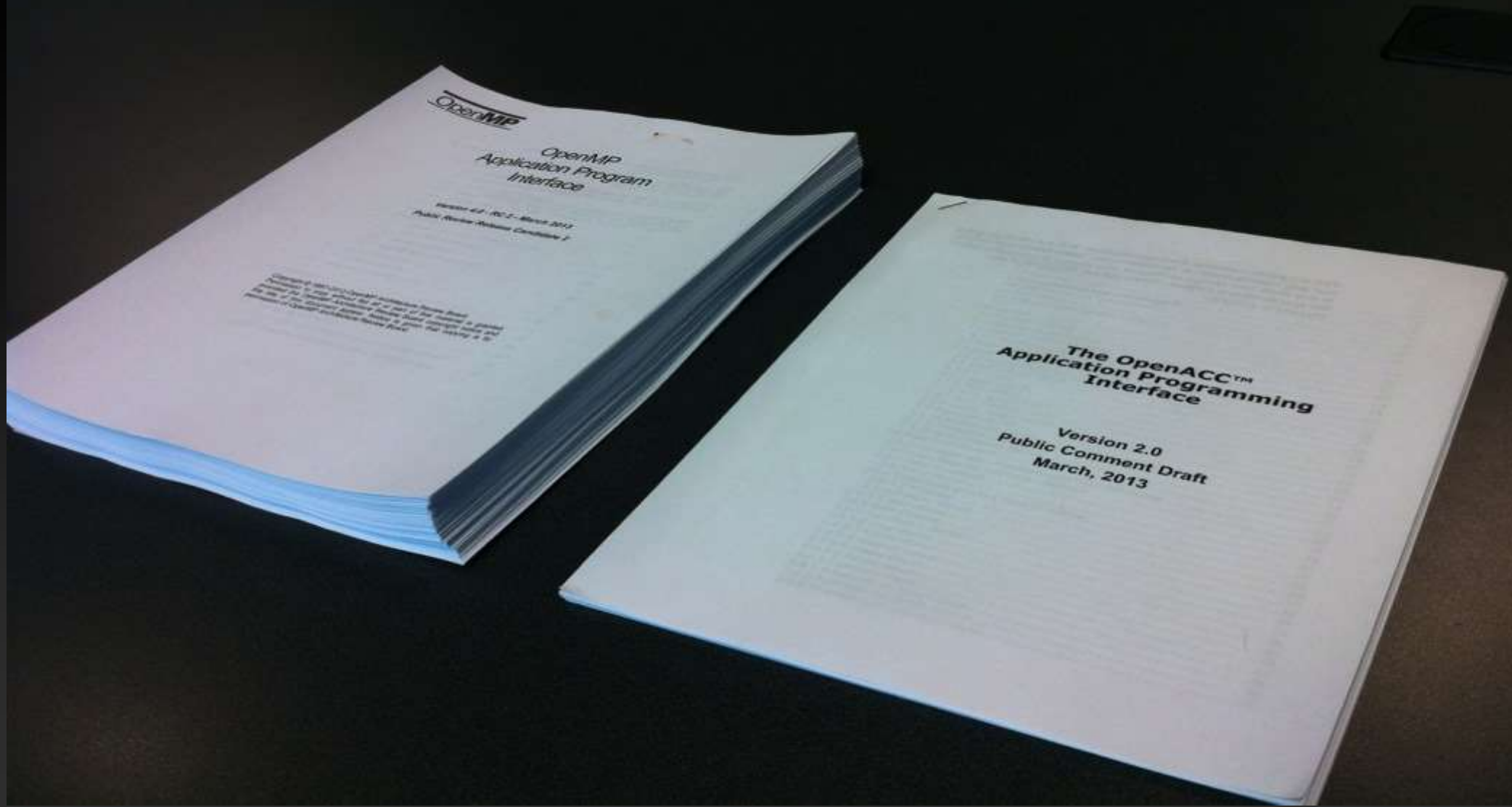
www.nvidia.com/gpudirectives



NVIDIA

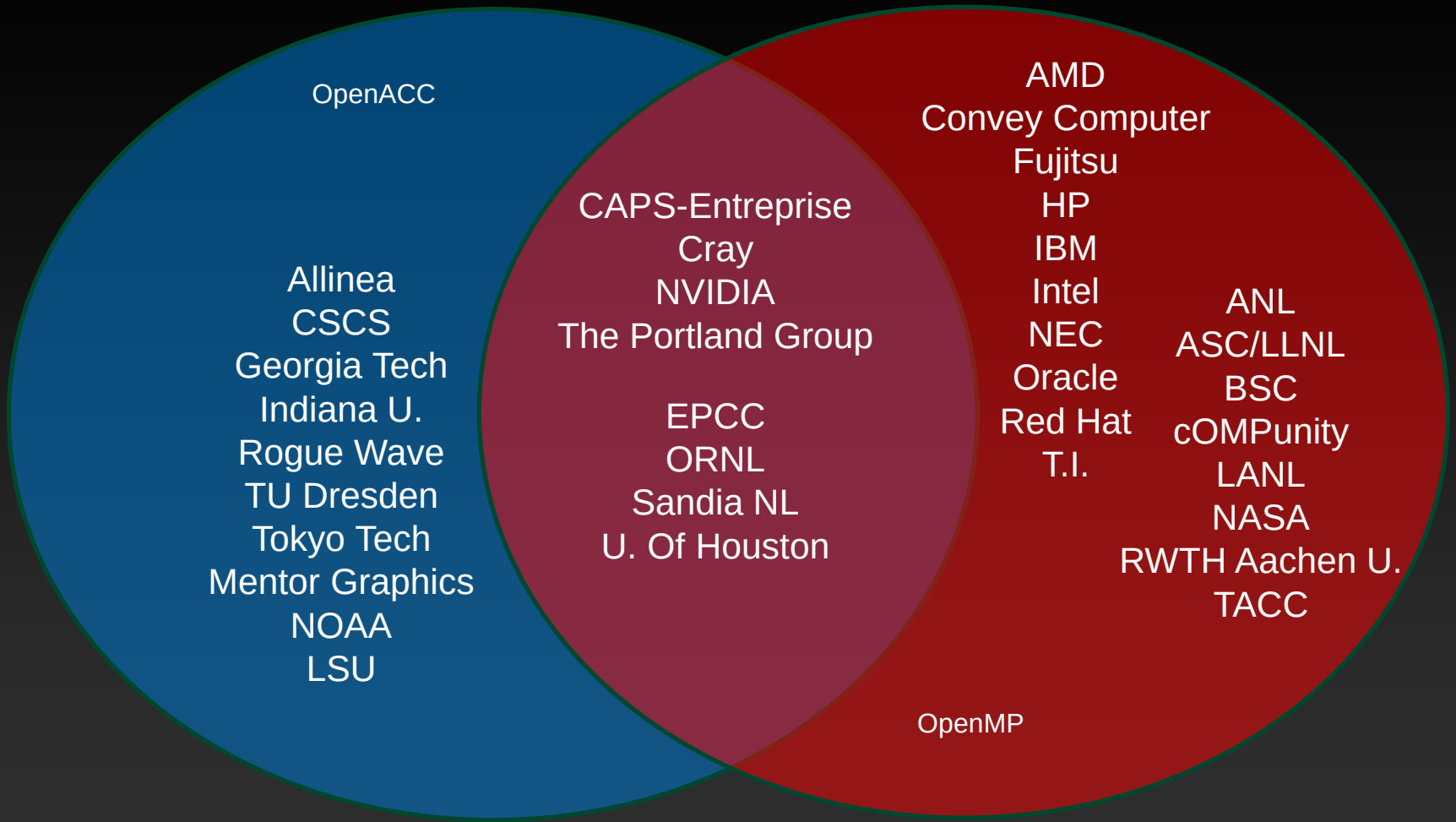
PGI®

OpenACC and OpenMP



Most OpenACC members also in OpenMP

OpenACC
Directives for Accelerators

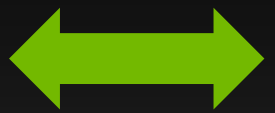


NVIDIA Contributes *Actively to OpenMP 4.0 and beyond*

OpenACC
Directives for Accelerators

```
a = 0.0
!$omp target update to(a)
!$omp target
!$omp parallel
!$omp do reduction(+:a)
do i = 1,n
   a= a+b(i)
end do
!$omp end do
!$omp end parallel
!$omp end target
```

Works well on Phi



```
a = 0.0
!$omp target update to(a)
!$omp target
!$omp team
!$omp distribute reduction(+:a)
do i = 1,n
   a= a+b(i)
end do
!$omp end do
!$omp end distribute
!$omp end team
!$omp end target
```

Works OK on GPU
Missing *many* optimizations currently available with OpenACC

Source: Cray Inc.

Comparing OpenACC with OpenMP 4.0 on NVIDIA and Phi

- OpenMP 4.0 for Intel Xeon Phi

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
 for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

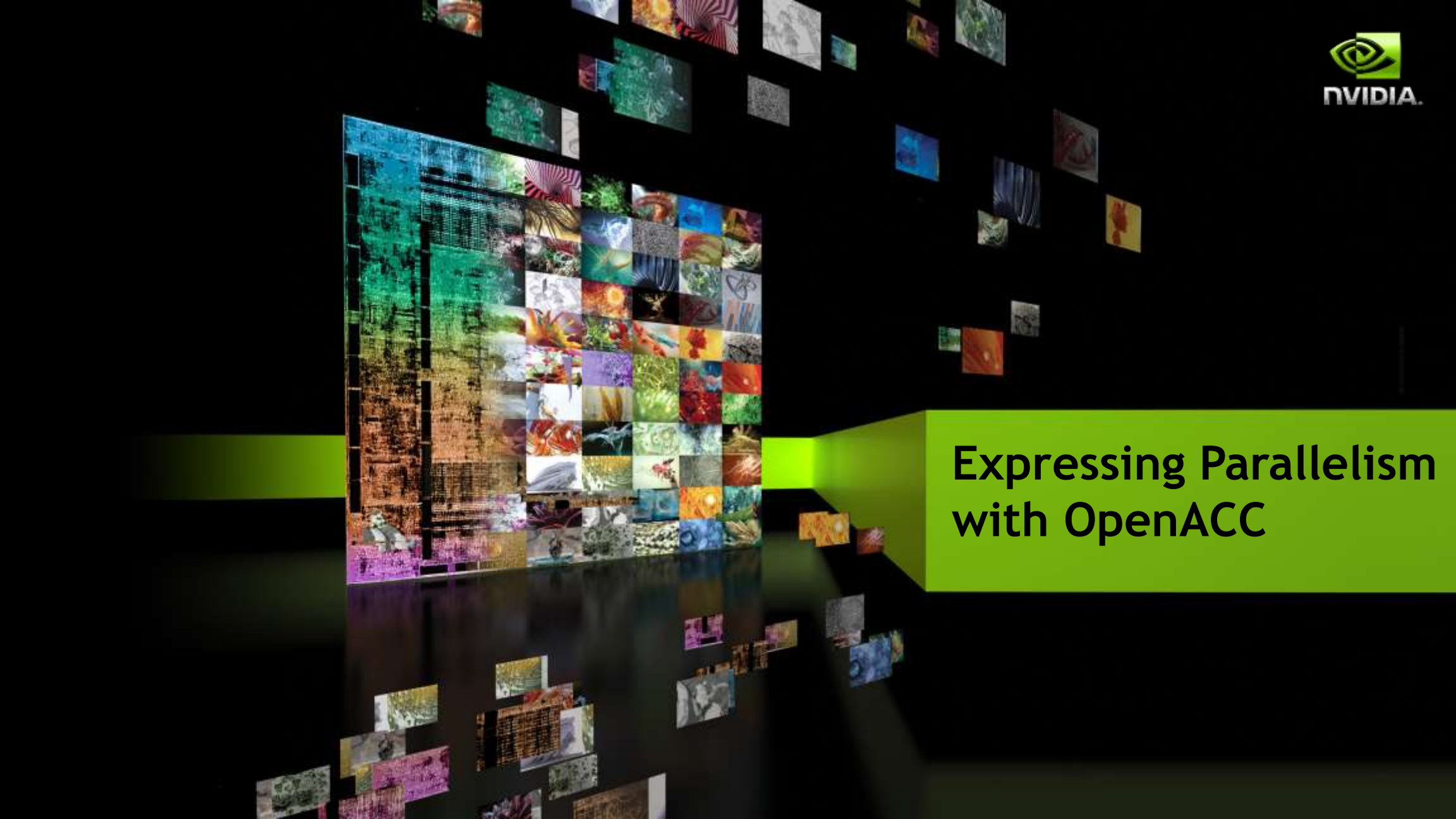
- OpenMP 4.0 for NVIDIA GPU

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
 for (i=0; i<N; i += num_blocks)
#pragma omp parallel for
      for (b = i; b < i+num_blocks; b++)
            B[b] += sin(B[b]);
```

- OpenACC for NVIDIA GPU

```
#pragma acc kernels
for (i=0; i<N; ++i)
B[i]+= sin(B[i]);
```

- First two examples
- Courtesy Christian Terboven


NVIDIA.
Expressing Parallelism
with OpenACC

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];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, 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, 2.0, x, y)
...
```

SAXPY - Single prec A*X Plus Y in OpenMP

SAXPY in C

```
void saxpy(int n, float a,
           float *x, float *y)
{
#pragma omp parallel for
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, 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, 2.0, x, y)
...
```

SAXPY - Single prec A*X Plus Y in OpenACC

SAXPY in C

```
void saxpy(int n, float a,
           float *x, float *y)
{
#pragma acc parallel loop
  for (int i = 0; i < n; ++i)
    y[i] = a*x[i] + y[i];
}

int N = 1<<20;



// Perform SAXPY on 1M elements
saxpy(N, 2.0, 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, 2.0, x, y)
...
```

A Very Simple Exercise: SAXPY

SAXPY in C

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

...
// Perform SAXPY on 1M elements
saxpy(1<<20, 2.0, x, y);
...
```

SAXPY in Fortran

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


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

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

kernels: Your first OpenACC Directive

Each loop executed as a separate *kernel* on the GPU.

```
!$acc kernels
      do i=1,n
         a(i) = 0.0
         b(i) = 1.0
         c(i) = 2.0
      end do

      do i=1,n
         a(i) = b(i) + c(i)
      end do
!$acc end kernels
```

kernel 1

kernel 2

Kernel:
A parallel function that runs on the GPU

Kernels Construct

Fortran

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

C

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

Clauses

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

Also, any data clause (more later)

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

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

*restrict:
"I promise y does not alias x"

```
int main(int argc, char **argv)
{
  int N = 1<<20; // 1 million floats

  if (argc > 1)
    N = atoi(argv[1]);

  float *x = (float*)malloc(N * sizeof(float));
  float *y = (float*)malloc(N * sizeof(float));

  for (int i = 0; i < N; ++i) {
    x[i] = 2.0f;
    y[i] = 1.0f;
  }

  saxpy(N, 3.0f, x, y);

  return 0;
}
```

Compile and run

- C:

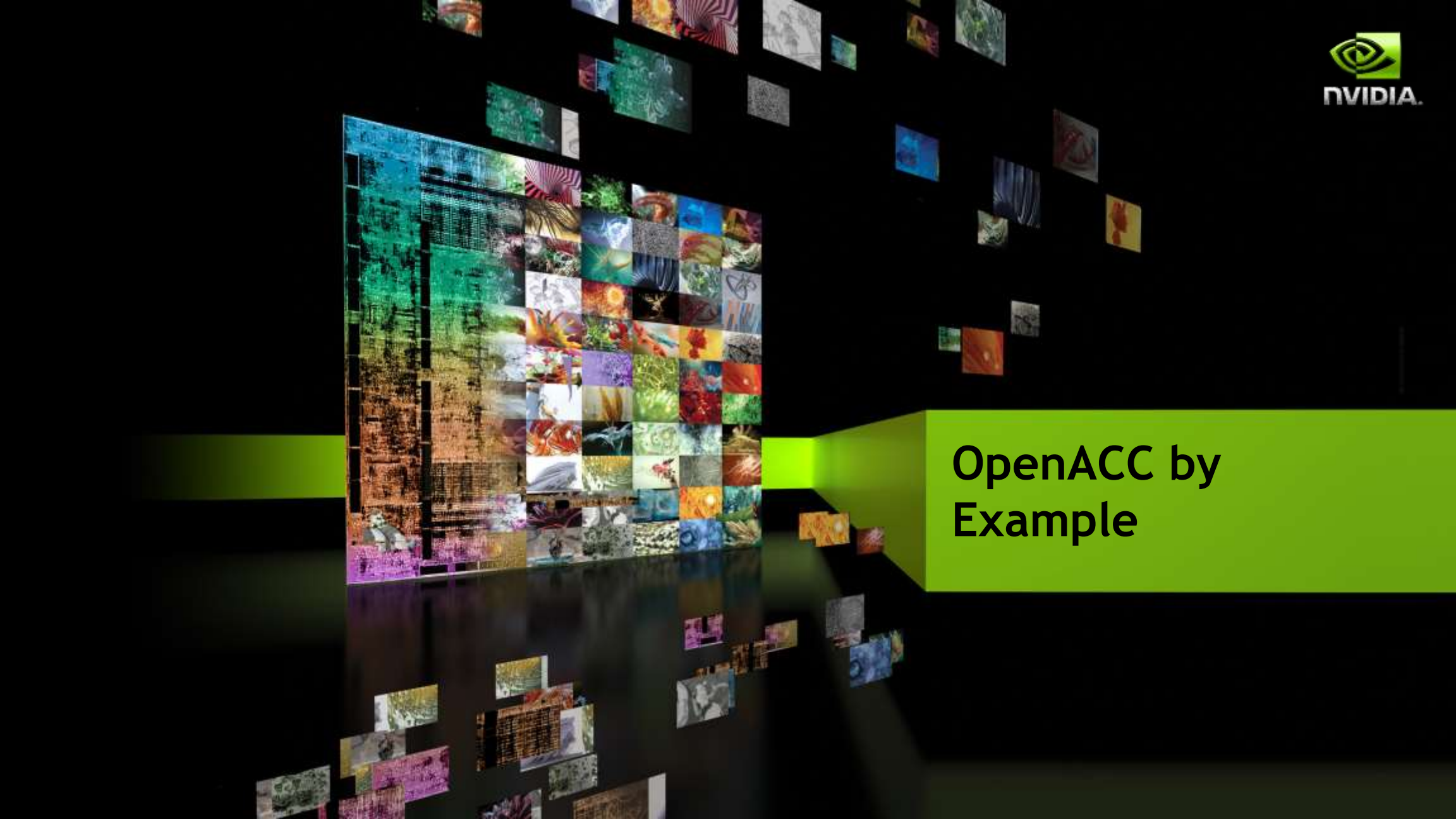
```
pgcc -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.c
```

- Fortran:

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

- Compiler output:

```
pgcc -acc -Minfo=accel -ta=nvidia -o saxpy_acc saxpy.c
saxpy:
      8, Generating copyin(x[:n-1])
         Generating copy(y[:n-1])
         Generating compute capability 1.0 binary
         Generating compute capability 2.0 binary
      9, Loop is parallelizable
         Accelerator kernel generated
          9, #pragma acc loop worker, vector(256) /* blockIdx.x threadIdx.x */
             CC 1.0 : 4 registers; 52 shared, 4 constant, 0 local memory bytes; 100% occupancy
             CC 2.0 : 8 registers; 4 shared, 64 constant, 0 local memory bytes; 100% occupancy
```

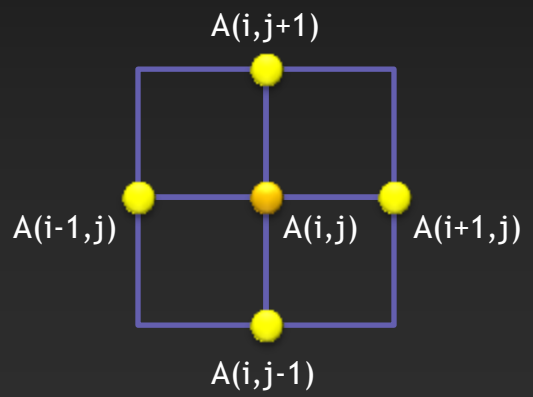

NVIDIA.
OpenACC by
Example

Our Foundation Exercise: Jacobi Iteration

- It is a simulation problem, not rigged for OpenACC.
- In this most basic form, it solves the Laplace equation: $\Delta u = 0$
- In our workshop example it is the Steady State Heat Equation.
- Students start with a realistic, normal serial code and parallelize it themselves.

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


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

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

  iter++;
}
```

Iterate until converged

Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

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

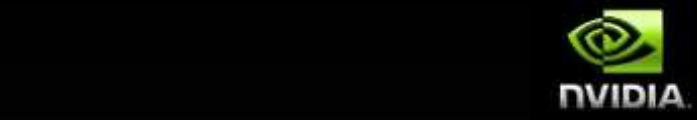
Iterate across matrix elements

Calculate new value from neighbors

Compute max error for convergence

Swap input/output arrays

OpenMP C Code



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

#pragma omp parallel for shared(m, n, Anew, A)
  for( int j = 1; j < n-1; j++) {
    for(int i = 1; i < m-1; i++) {

      Anew[j][i] = 0.25 * (A[j][i+1] + A[j][i-1] +
                           A[j-1][i] + A[j+1][i]);

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

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

  iter++;
}
```

Parallelize loop across CPU threads

Parallelize loop across CPU threads

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 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

  iter = iter +1
end do
```

Parallelize loop across CPU threads

Parallelize loop across CPU threads

Exercises: General Instructions (compiling)

- Exercises are in “exercises” directory in your home directory
  - Solutions are in “solutions” directory

- To compile, use one of the provided makefiles

```
> cd exercises/001-laplace2D
```

C:

```
> make
```

Fortran:

```
> make -f Makefile_f90
```

- Remember these compiler flags:

```
-acc -ta=nvidia -Minfo=accel (PGI)
```

Exercises: General Instructions (running)

To run, use qsub with one of the provided job files

```
> qsub laplace_acc.job
> qstat          # prints qsub status
```

Output is placed in laplace_acc.job.o<job#> when finished.

OpenACC job file looks like this

```
#!/bin/ksh
#PBS -l walltime=3:00
export HMPPRT_PATH= your compile directory (HMPP CAPS)
./laplace2d_acc
```

The OpenMP version specifies number of cores to use

```
#!/bin/csh
#PBS -l walltime=3:00
setenv OMP_NUM_THREADS 6
./laplace2d_omp
```

Edit this to control the number of cores to use

GPU startup overhead

- If no other GPU process running, GPU driver may be swapped out
  - Linux specific
  - Starting it up can take 1-2 seconds
- Two options
  - Run nvidia-smi in persistence mode (requires root permissions)
  - Run “nvidia-smi -q -l 30” in the background
- If your running time is off by ~2 seconds from results in these slides, suspect this
  - nvidia-smi should be running in persistent mode for these exercises

Exercise 1: Jacobi Kernels

- Task: use `acc kernels` to parallelize the Jacobi loop nests
- Edit laplace2D.c or laplace2D.f90 (your choice)
  - In the `001-laplace2D-kernels` directory
  - Add directives where it helps
  - Figure out the proper compilation command (similar to SAXPY example)
    - Compile both with and without OpenACC parallelization
    - Optionally compile with OpenMP (original code has OpenMP directives)
  - Run OpenACC version with `laplace_acc.job`, OpenMP with `laplace_omp.job`
- Q: can you get a speedup with just kernels directives?
  - Versus 1 CPU core? Versus 6 CPU cores?

Exercise 1 Solution: OpenACC C


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

#pragma acc kernels
  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]);

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

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

  iter++;
}
```

Execute GPU kernel for loop nest

Execute GPU kernel for loop nest

Exercise 1 Solution: OpenACC Fortran


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

!$acc kernels
  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 kernels

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

Generate GPU kernel for loop nest

Generate GPU kernel for loop nest

Exercise 1 Solution: C Makefile


```
CC       = pgcc
CCFLAGS  =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN =  laplace2d_omp laplace2d_acc

all: $(BIN)

laplace2d_acc: laplace2d.c
        $(CC) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_omp: laplace2d.c
        $(CC) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
        $(RM) $(BIN)
```

Exercise 1 Solution: Fortran Makefile


```
F90        = pgf90
CCFLAGS    =
ACCFLAGS = -acc -ta=nvidia, -Minfo=accel
OMPFLAGS = -fast -mp -Minfo

BIN =  laplace2d_f90_omp laplace2d_f90_acc

all: $(BIN)

laplace2d_f90_acc: laplace2d.f90
        $(F90) $(CCFLAGS) $(ACCFLAGS) -o $@ $<

laplace2d_f90_omp: laplace2d.f90
        $(F90) $(CCFLAGS) $(OMPFLAGS) -o $@ $<

clean:
        $(RM) $(BIN)
```

Exercise 1: Compiler output (C)


```
pgcc  -acc -ta=nvidia -Minfo=accel -o laplace2d_acc laplace2d.c
main:
     57, Generating copyin(A[:4095][:4095])
         Generating copyout(Anew[1:4094][1:4094])
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     58, Loop is parallelizable
     60, Loop is parallelizable
         Accelerator kernel generated
         58, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
         60, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
             Cached references to size [18x18] block of 'A'
             CC 1.3 : 17 registers; 2656 shared, 40 constant, 0 local memory bytes; 75% occupancy
             CC 2.0 : 18 registers; 2600 shared, 80 constant, 0 local memory bytes; 100% occupancy
         64, Max reduction generated for error
     69, Generating copyout(A[1:4094][1:4094])
         Generating copyin(Anew[1:4094][1:4094])
         Generating compute capability 1.3 binary
         Generating compute capability 2.0 binary
     70, Loop is parallelizable
     72, Loop is parallelizable
         Accelerator kernel generated
         70, #pragma acc loop worker, vector(16) /* blockIdx.y threadIdx.y */
         72, #pragma acc loop worker, vector(16) /* blockIdx.x threadIdx.x */
             CC 1.3 : 8 registers; 48 shared, 8 constant, 0 local memory bytes; 100% occupancy
             CC 2.0 : 10 registers; 8 shared, 56 constant, 0 local memory bytes; 100% occupancy
```

Exercise 1: Performance

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	162.16	0.24x FAIL

Speedup vs. 1 CPU core

Speedup vs. 6 CPU cores

What went wrong?

- Add `-ta=nvidia,time` to compiler command line

```
Accelerator Kernel Timing data
/usr/users/6/harrism/openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c
  main
    69: region entered 1000 times
        time(us): total=77524918 init=240 region=77524678
                  kernels=4422961 data=66464916
        w/o init: total=77524678 max=83398 min=72025 avg=77524
        72: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=4422961 max=4543 min=4345 avg=4422
/usr/users/6/harrism/openacc-workshop/solutions/001-laplace2D-kernels/laplace2d.c
  main
    57: region entered 1000 times
        time(us): total=82135902 init=216 region=82135686
                  kernels=8346306 data=66775717
        w/o init: total=82135686 max=159083 min=76575 avg=82135
        60: kernel launched 1000 times
            grid: [256x256]  block: [16x16]
            time(us): total=8201000 max=8297 min=8187 avg=8201
        64: kernel launched 1000 times
            grid: [1]  block: [256]
            time(us): total=145306 max=242 min=143 avg=145
acc_init.c
  acc_init
    29: region entered 1 time
        time(us): init=158248
```

4.4 seconds

66.5 seconds

8.3 seconds

66.8 seconds

Huge Data Transfer Bottleneck!
Computation: 12.7 seconds
Data movement: 133.3 seconds

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

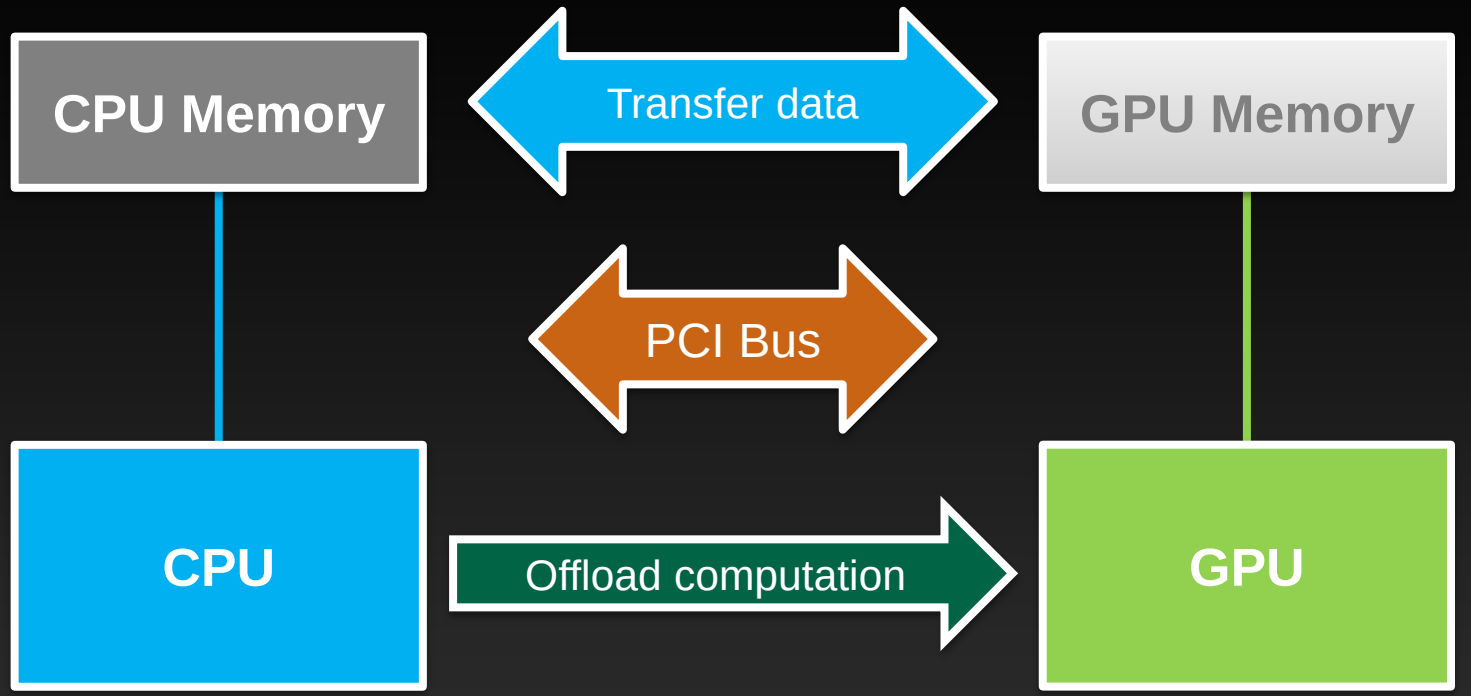
```
% export PGI_ACC_NOTIFY=3
% task1
upload CUDA data  file=~/task1.f90 function=saxpy line=7 device=0
              variable=x      bytes=12000
upload CUDA data  file=~/task1.f90 function=saxpy line=7 device=0
              variable=y      bytes=12000

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

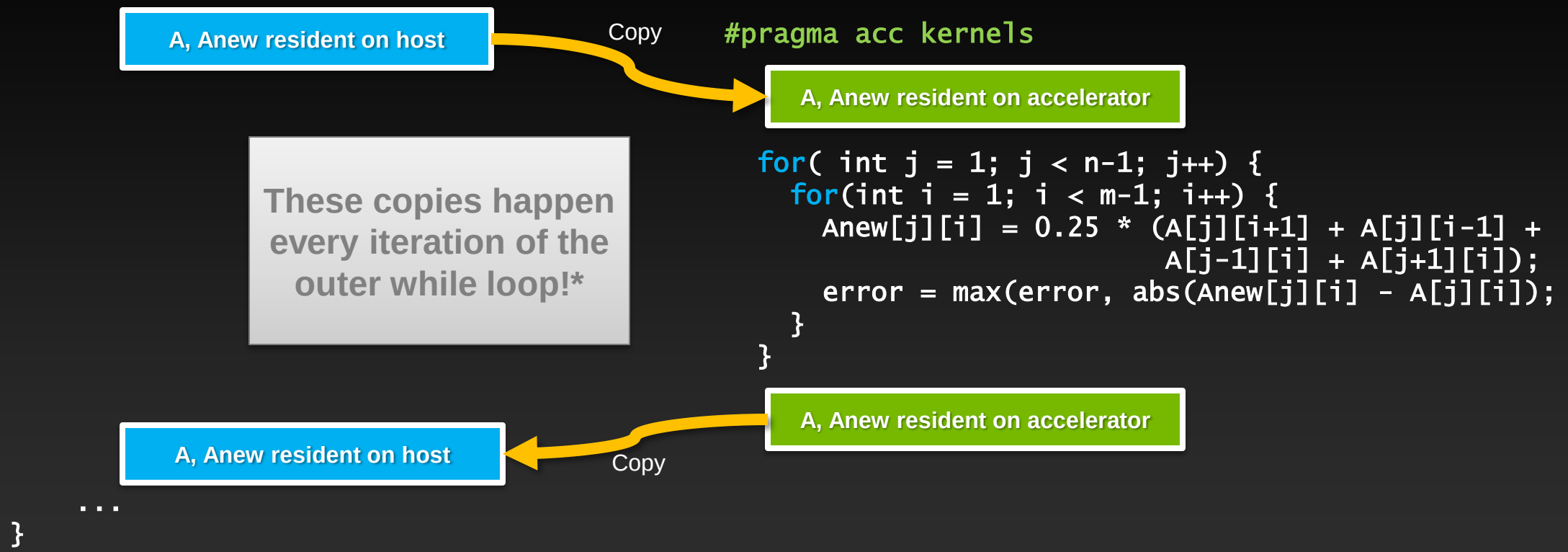
```
Accelerator Kernel Timing data
~/isc/openacc/task1.f90
  saxpy  NVIDIA  devicenum=0
    time(us): 65
    7: compute region reached 1 time
        7: data copyin reached 2 times
             device time(us): total=34 max=25 min=9 avg=17
        7: kernel launched 1 time
            grid: [12]  block: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

Basic Concepts

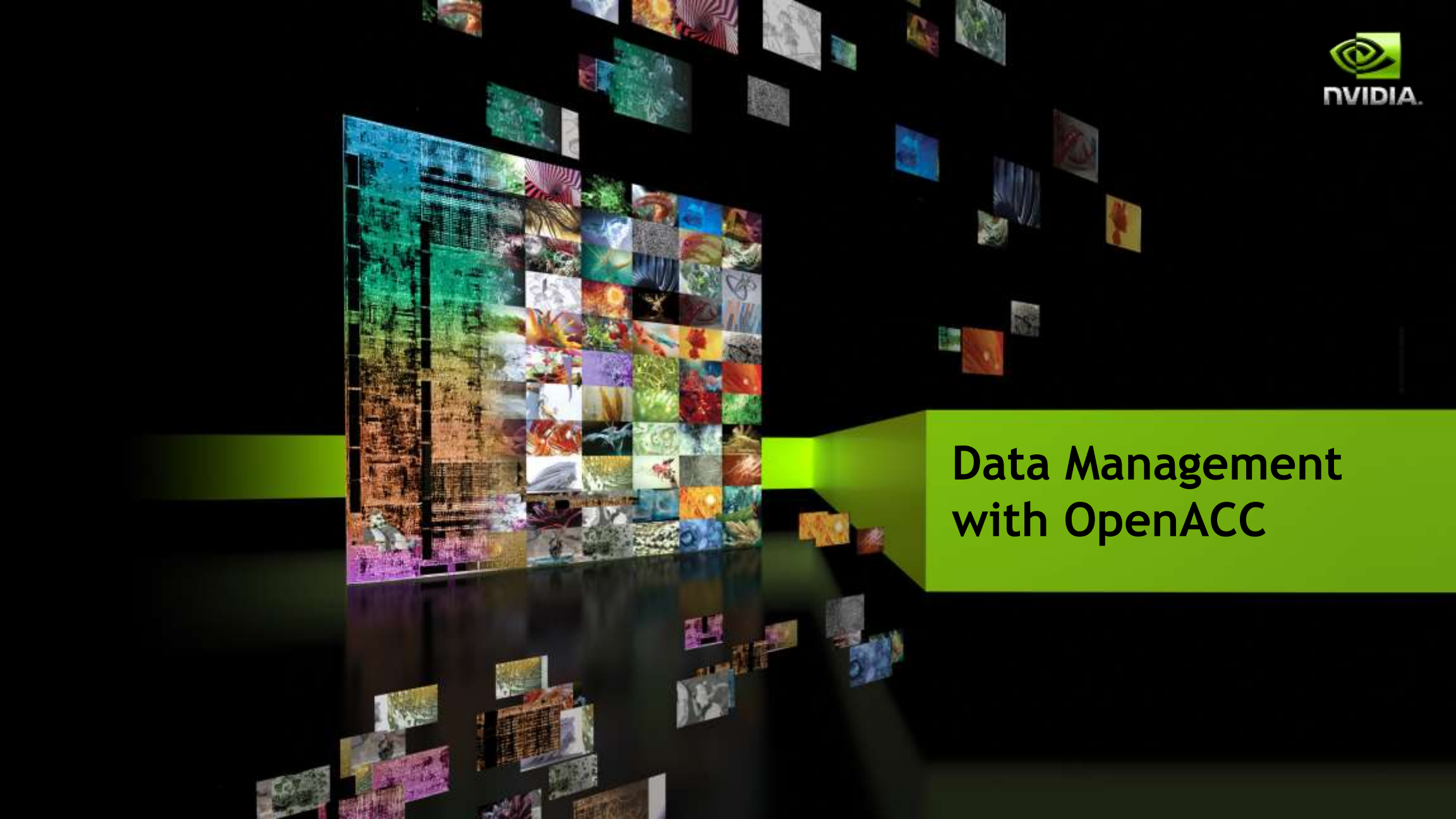


For efficiency, decouple data movement and compute off-load

Excessive Data Transfers



*Note: there are two #pragma acc kernels, so there are 4 copies per while loop iteration!


NVIDIA.
Data Management
with OpenACC

Defining data regions

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

```
!$acc data
    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 Construct

Fortran

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

C

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

General Clauses

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

Manage data movement. Data regions may be nested.

Data Clauses

`copy ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region and copies data to the host when exiting region.
`copyin ( list )`	Allocates memory on GPU and copies data from host to GPU when entering region.
`copyout ( list )`	Allocates memory on GPU and copies data to the host when exiting region.
`create ( list )`	Allocates memory on GPU but does not copy.
`present ( list )`	Data is already present on GPU from another containing data region.

and `present_or_copy[in|out]`, `present_or_create`, `deviceptr`.

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-1]), copyout(b[s/4:3*s/4])
```

- Fortran

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

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

Update Construct

Fortran

```
!$acc update [clause …]
```

C

```
#pragma acc update [clause …]
```

Clauses

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

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

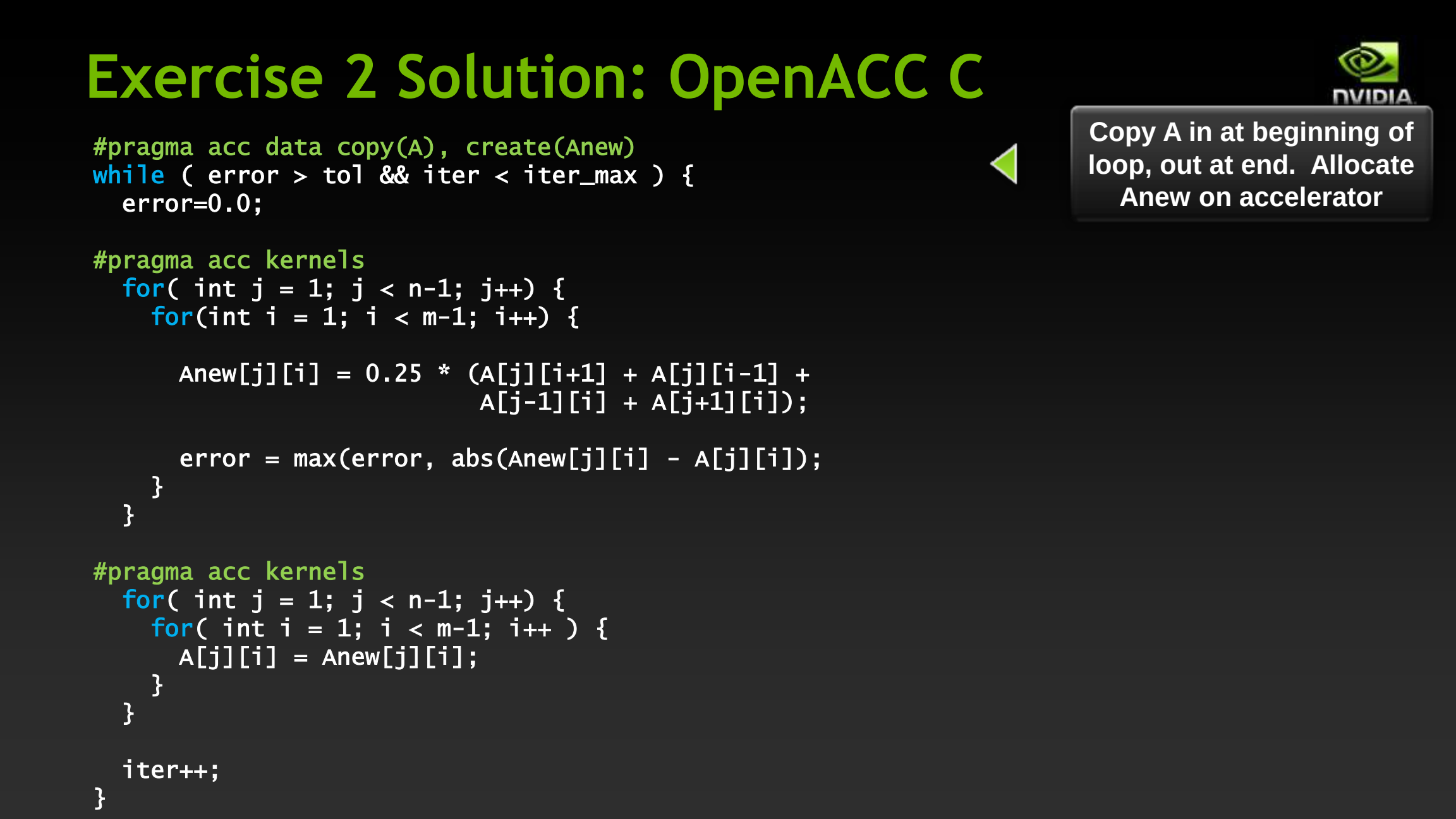
Used to update existing data after it has changed in its corresponding copy (e.g. update device copy after host copy changes)

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

Exercise 2: Jacobi Data Directives

- Task: use `acc data` to minimize transfers in the Jacobi example
- Start from given laplace2D.c or laplace2D.f90 (your choice)
  - In the `002-laplace2d-data` directory
  - Add directives where it helps (hint: [do] while loop)
- Q: What speedup can you get with data + kernels directives?
  - Versus 1 CPU core? Versus 6 CPU cores?

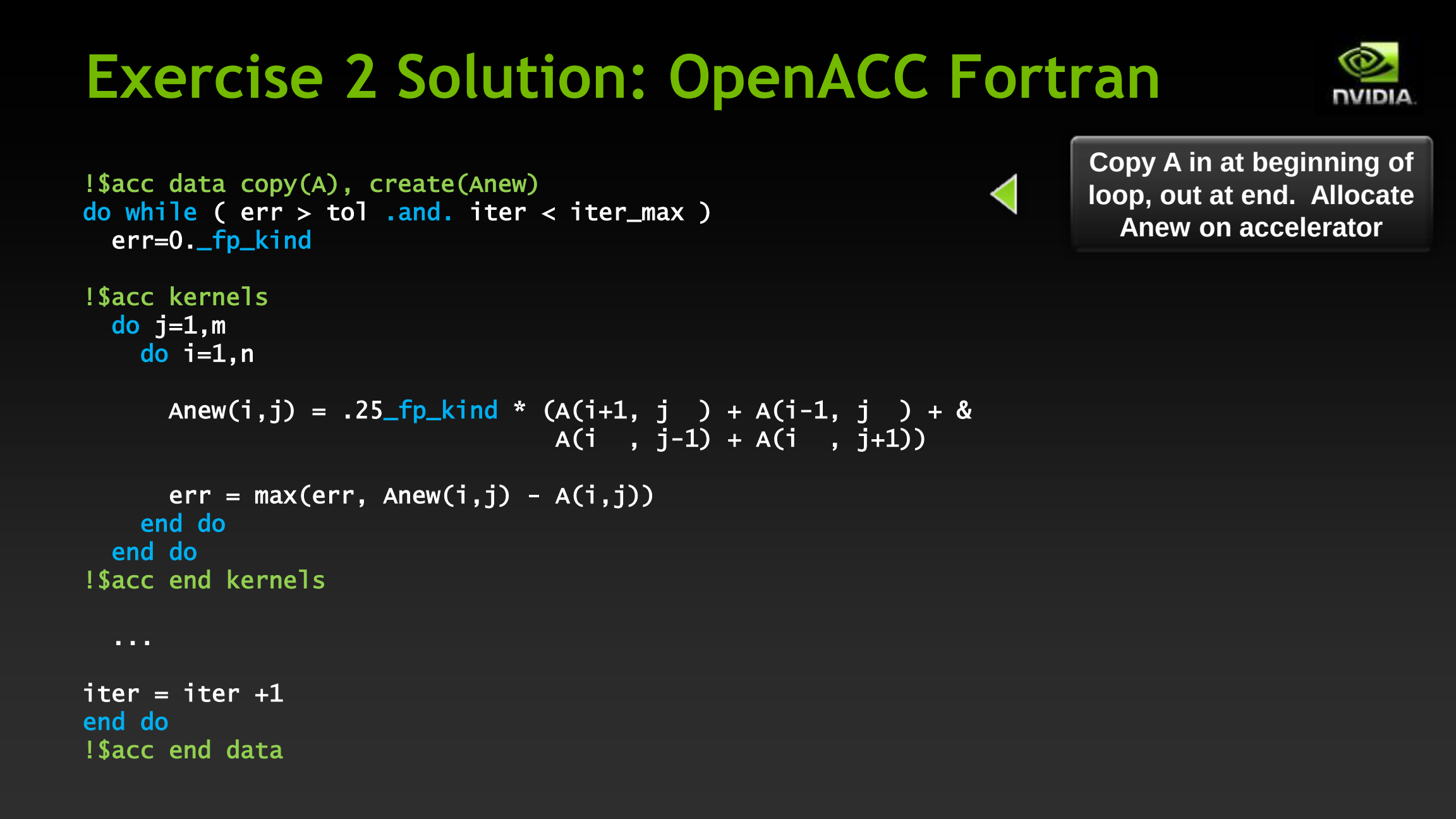
Exercise 2 Solution: OpenACC C



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

#pragma acc kernels
  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]);

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

#pragma acc kernels
  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

Exercise 2 Solution: OpenACC Fortran



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

!$acc kernels
  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 kernels

  ...

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

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

Exercise 2: Performance

CPU: Intel Xeon X5680
6 Cores @ 3.33GHz

GPU: NVIDIA Tesla M2070

Execution	Time (s)	Speedup
CPU 1 OpenMP thread	69.80	--
CPU 2 OpenMP threads	44.76	1.56x
CPU 4 OpenMP threads	39.59	1.76x
CPU 6 OpenMP threads	39.71	1.76x
OpenACC GPU	13.65	2.9x

Speedup vs. 1 CPU core

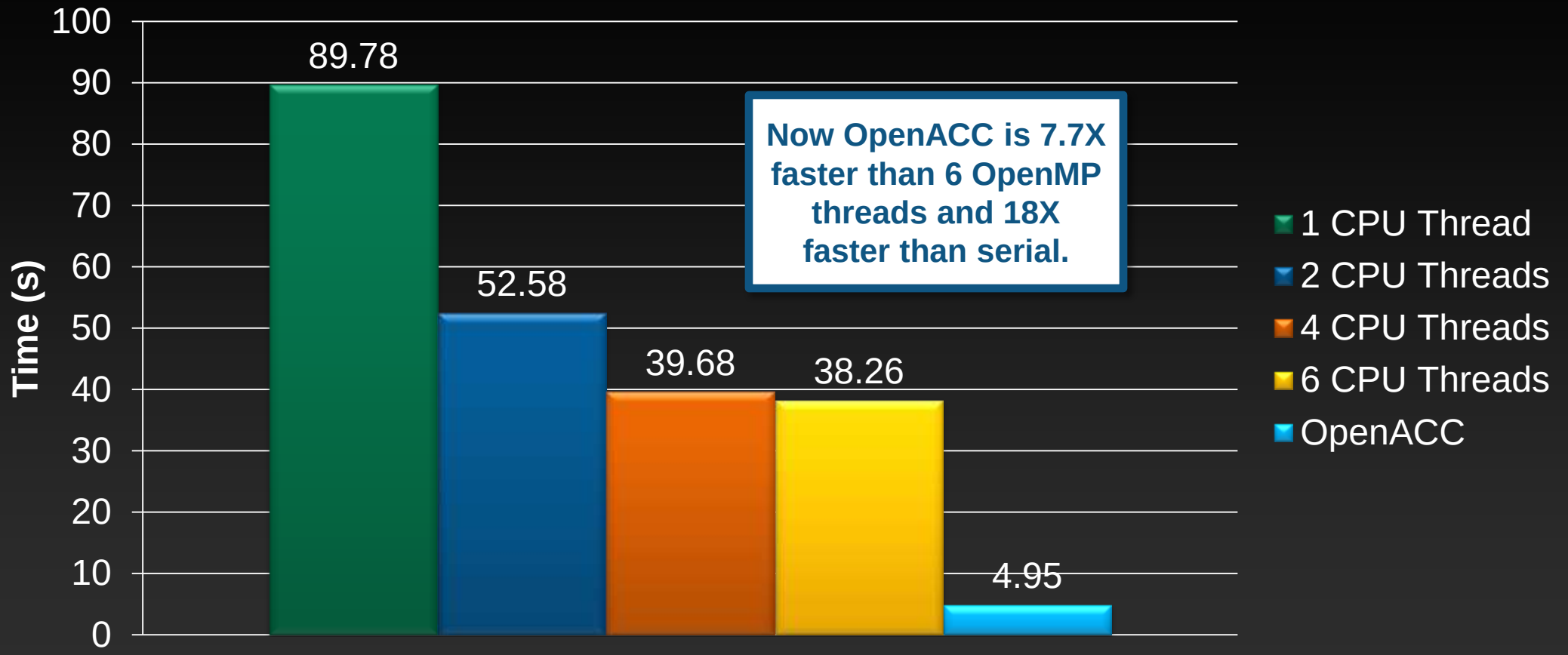
Speedup vs. 6 CPU cores

Note: same code runs in 9.78s on NVIDIA Tesla M2090 GPU

Execution Time (lower is better)

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

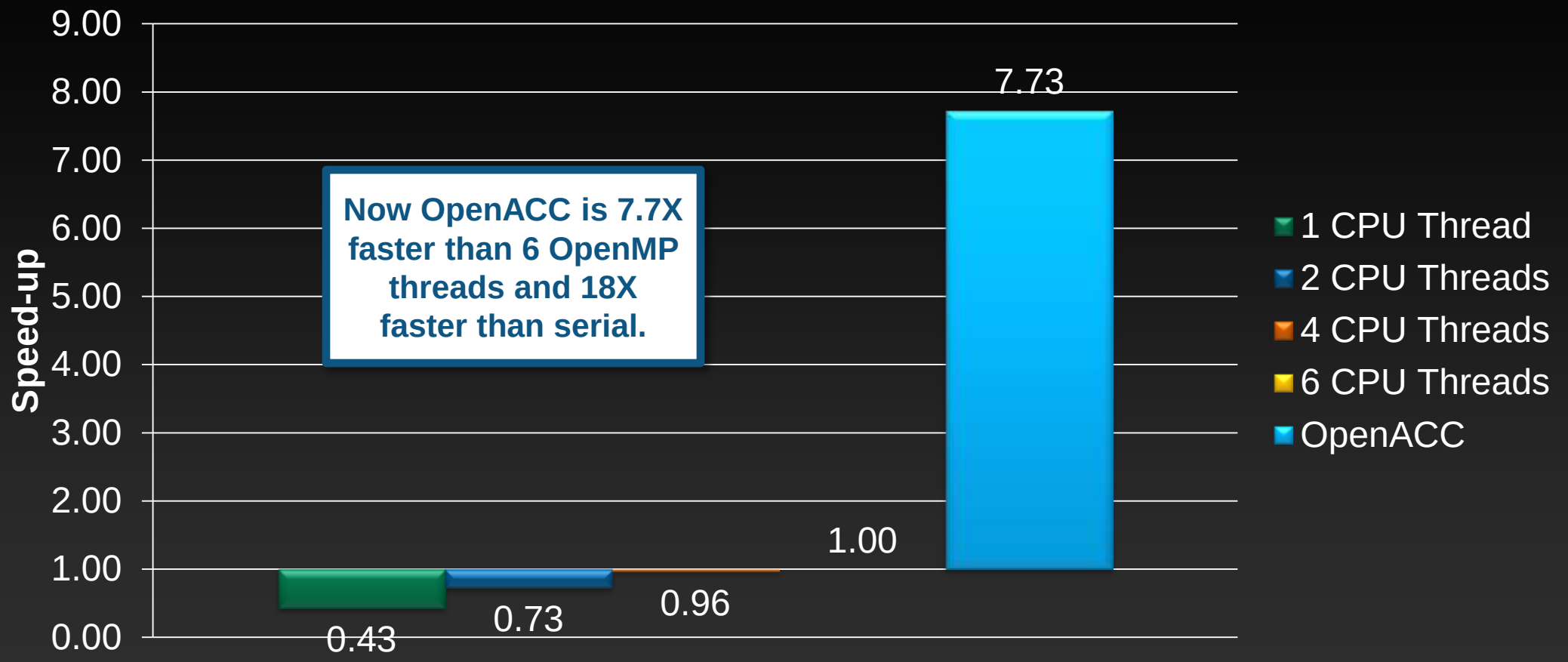
GPU: NVIDIA Tesla K20



Relative Speed-up

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

GPU: NVIDIA Tesla K20



Further speedups

- OpenACC gives us more detailed control over parallelization
  - Via `gang`, `worker`, and `vector` clauses
- By understanding more about 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 : Finding more parallelism

- 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

OpenACC Learning Resources

- OpenACC info, specification, FAQ, samples, and more
  - http://openacc.org
- PGI OpenACC resources
  - http://www.pgroup.com/resources/accel.htm

Complete OpenACC API

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

Kernels Construct

Fortran

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

C

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

Clauses

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

Also any data clause

Kernels Construct

Each loop executed as a separate kernel on the GPU.

```
!$acc kernels
      do i=1,n
         a(i) = 0.0
         b(i) = 1.0
         c(i) = 2.0
      end do

      do i=1,n
         a(i) = b(i) + c(i)
      end do
!$acc end kernels
```

kernel 1

kernel 2

Parallel Construct

Fortran

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

C

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

Clauses

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

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

Also any data clause

Parallel Clauses

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

Loop Construct

Fortran

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

C

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

Combined directives

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

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

Detailed control of the parallel execution of the following loop.

Loop Clauses

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

Loop Clauses Inside parallel Region

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

Loop Clauses Inside kernels Region

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

Parallel vs. Kernels

- **parallel and kernels regions look very similar**
- both define a region to be accelerated
- different heritage
- different levels of obligation for the compiler
- **parallel**
- prescriptive (like OpenMP programming model)
- uses a single accelerator kernel to accelerate region
- compiler *will* accelerate region
- even if this leads to incorrect results
- **kernels**
- descriptive (like PGI Accelerator programming model)
- uses one or more accelerator kernels to accelerate region
- compiler *may* accelerate region
- may not if not sure that loop iterations are independent
- For more info: http://www.pgroup.com/lit/articles/insider/v4n2a1.htm

Parallel vs. Kernels

- **Which to use (my opinion)**
- parallel offers greater control
- kernels better for initially exploring parallelism
- there should not be a performance difference
  with same compiler, using same loop scheduling

Other Syntax

Other Directives

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

Runtime Library Routines

Fortran

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

```
acc_get_num_devices
acc_set_device_type
acc_get_device_type
acc_set_device_num
acc_get_device_num
acc_async_test
acc_async_test_all
```

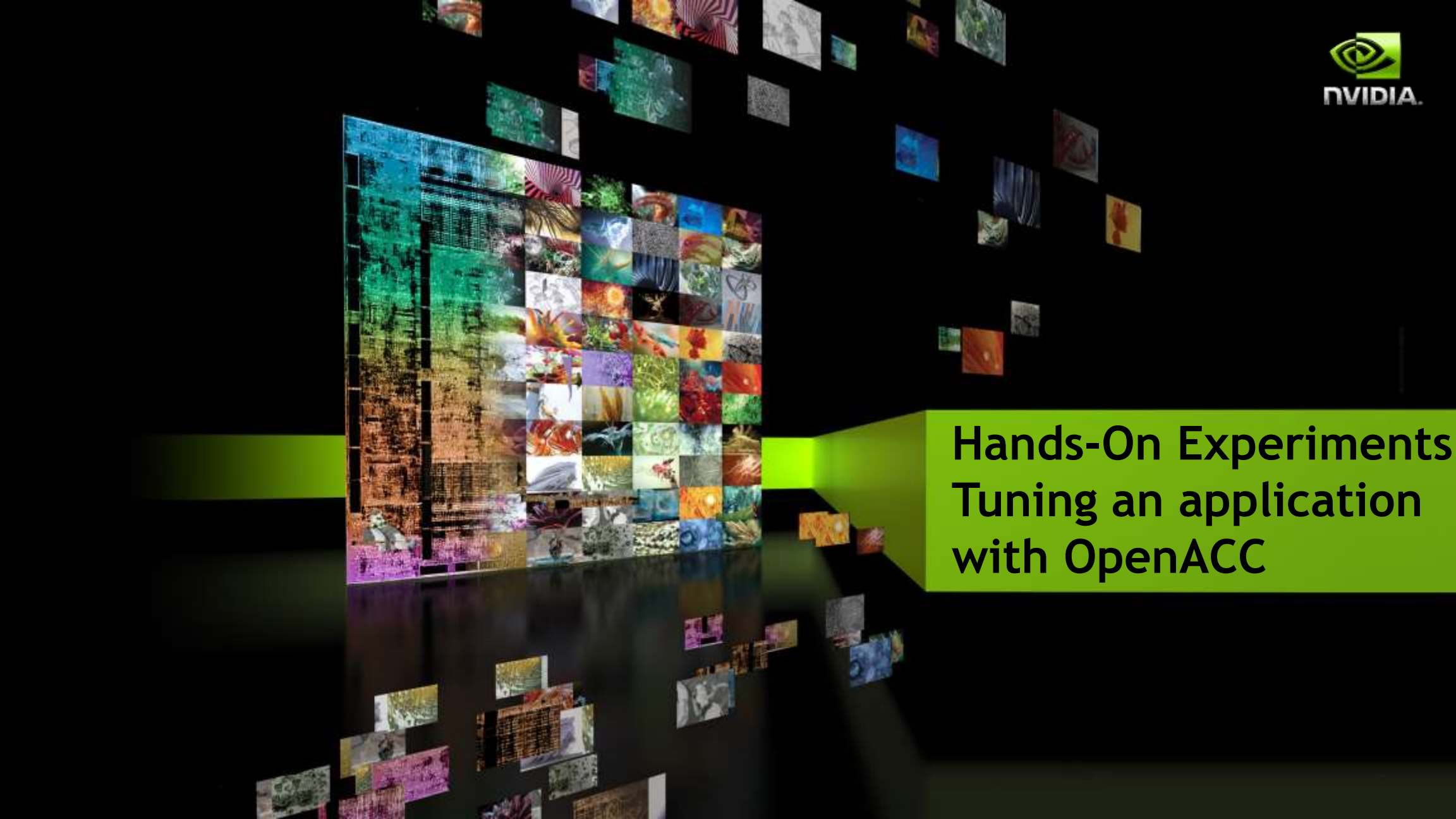
C

```
#include "openacc.h"
```

```
acc_async_wait
acc_async_wait_all
acc_shutdown
acc_on_device
acc_malloc
acc_free
```

Environment and Conditional Compilation

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


NVIDIA.
Hands-On Experiments
Tuning an application
with OpenACC

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
```

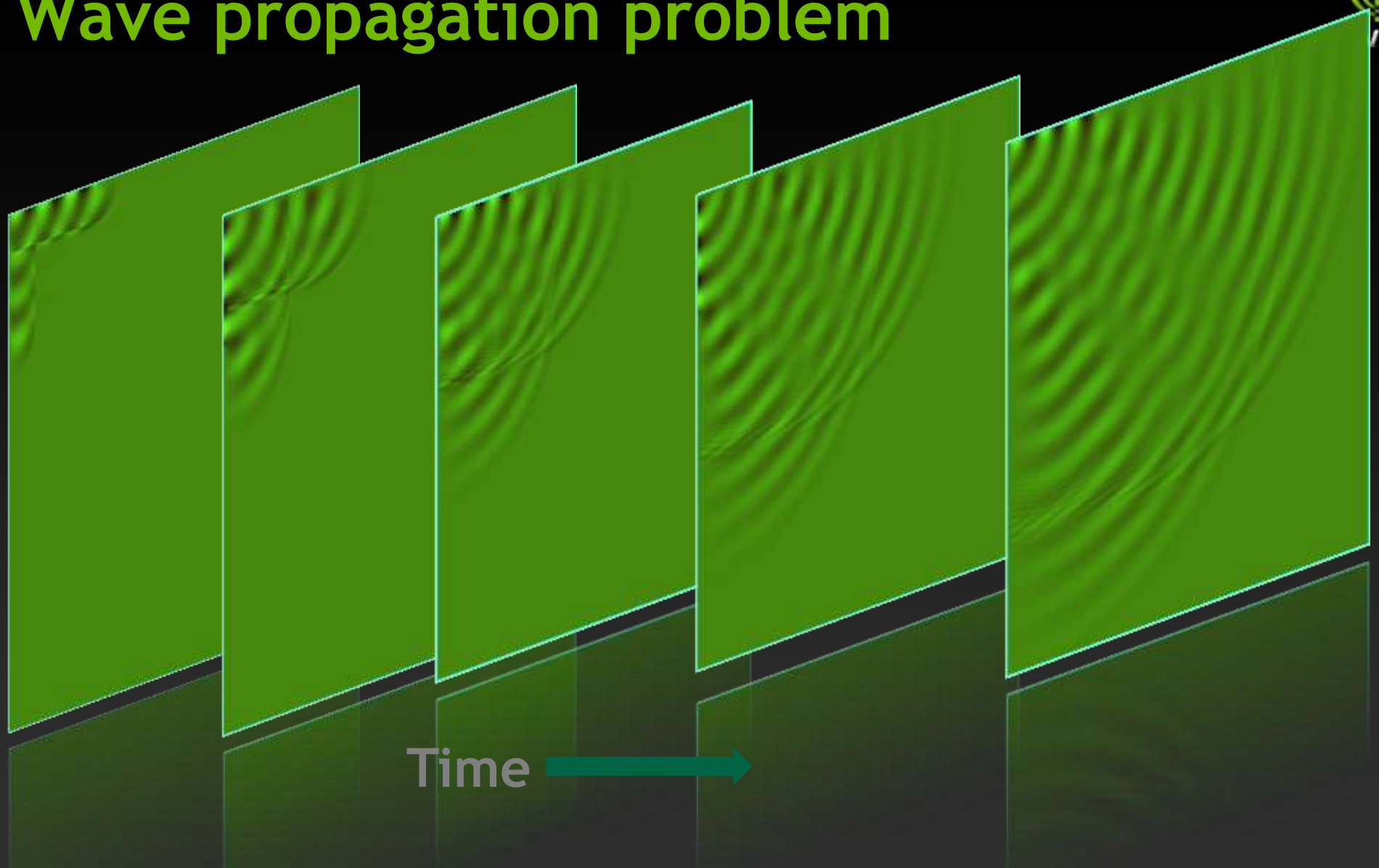
Data Region

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

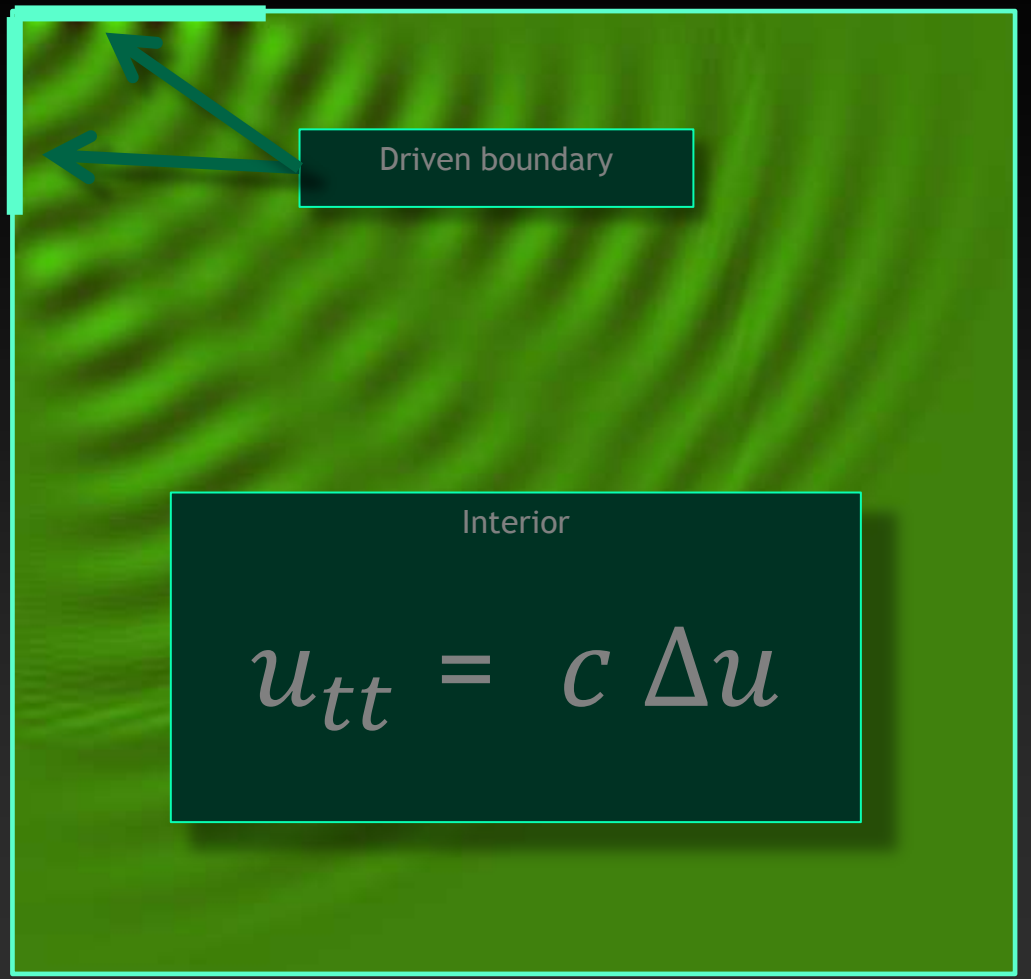
Outline

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

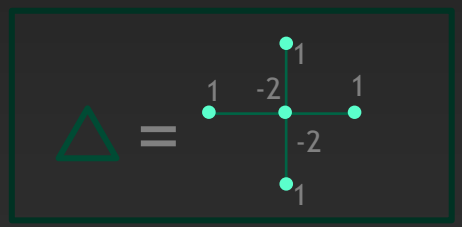
2D Wave propagation problem

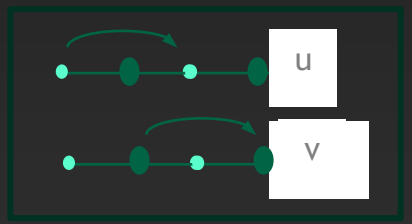


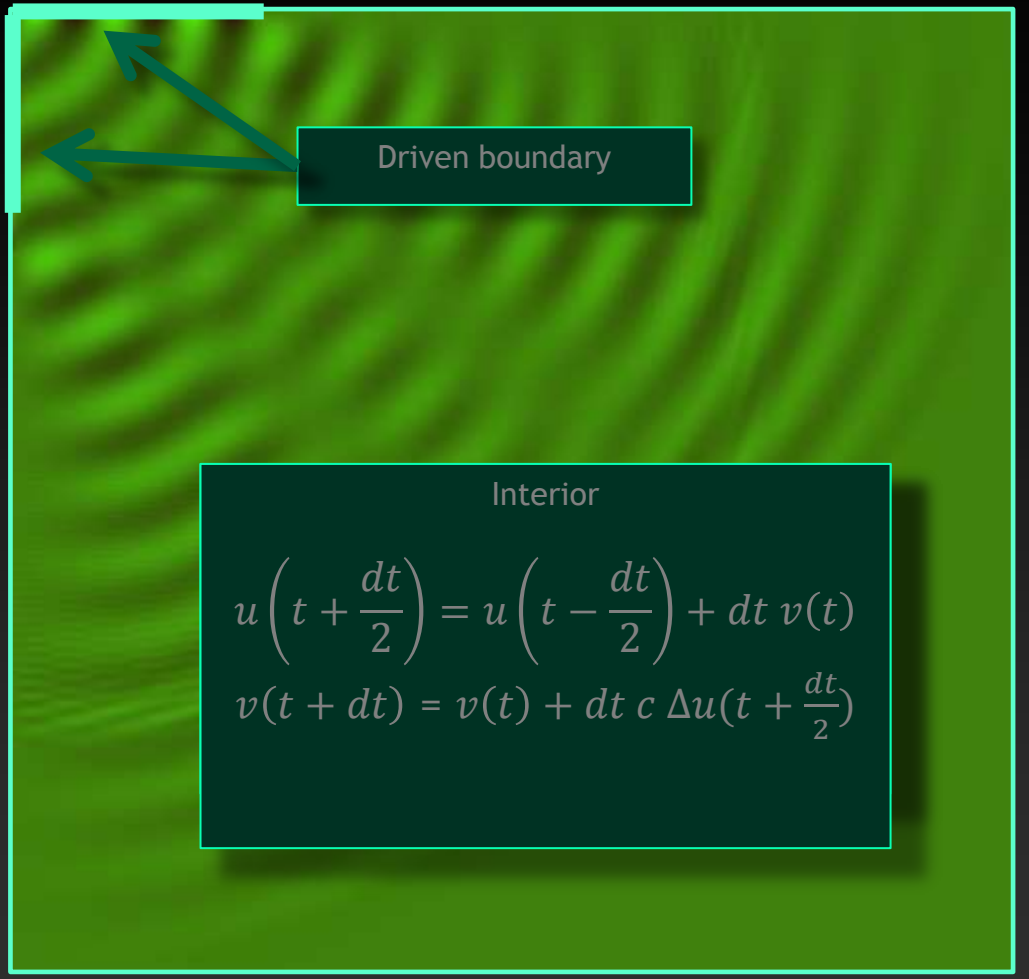
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$



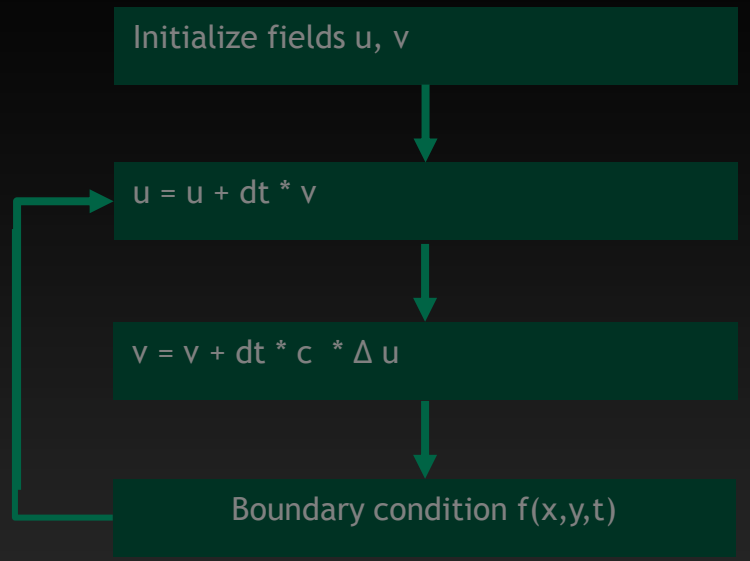
A basic 2D scalar wave solver: $u_{tt} = c\Delta u$

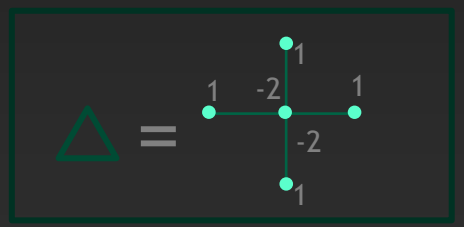


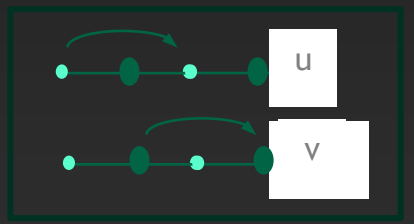


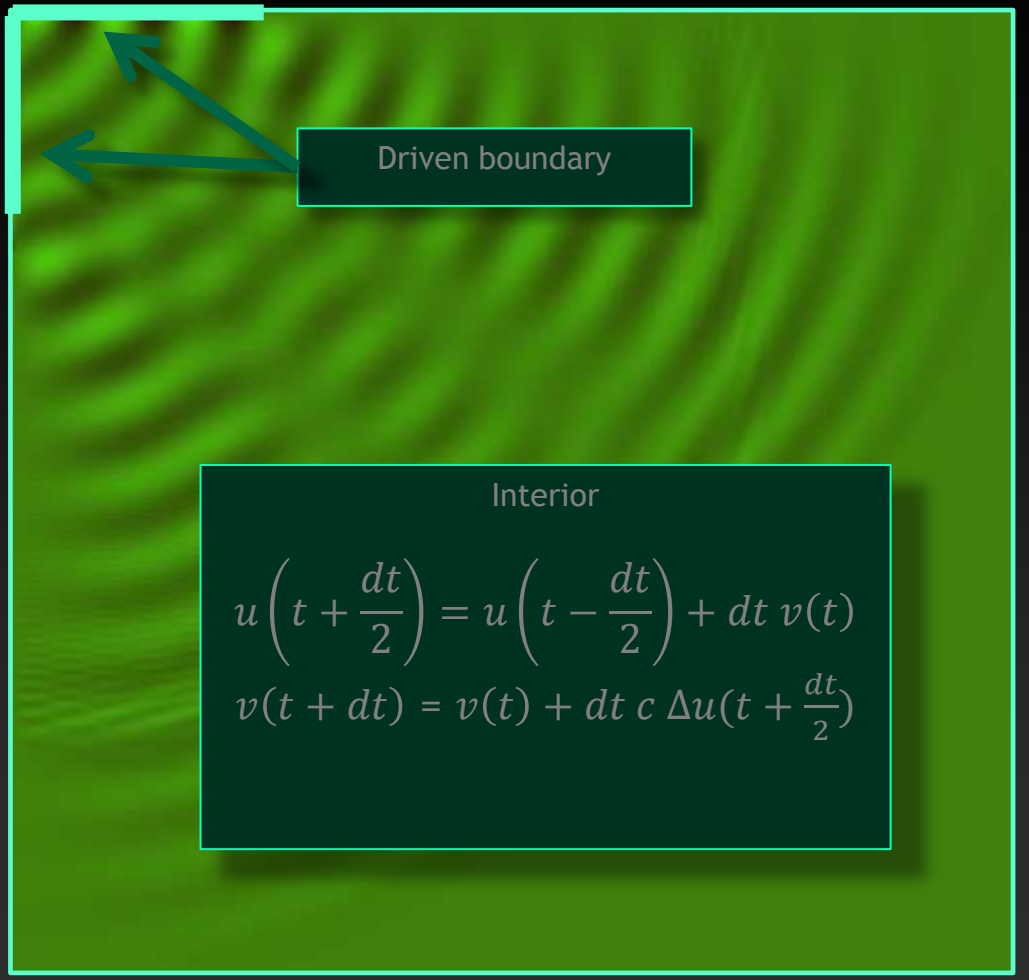


A basic 2D scalar wave solver: $u_{tt} = c\Delta u$

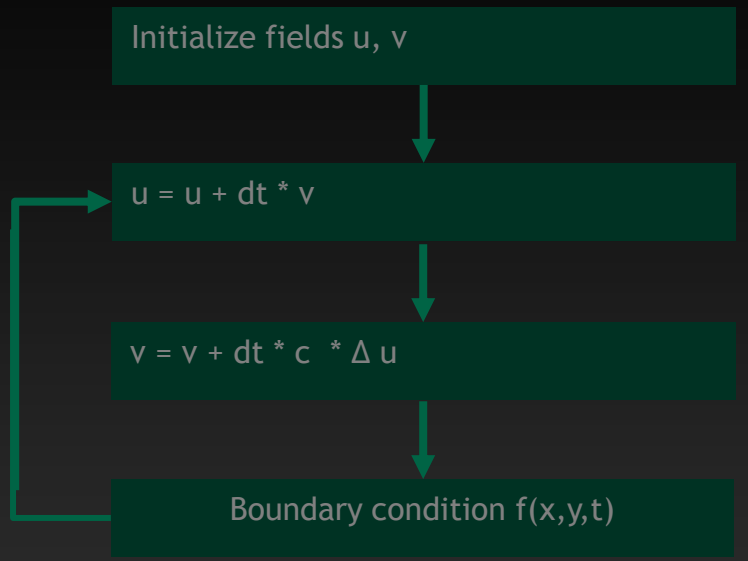






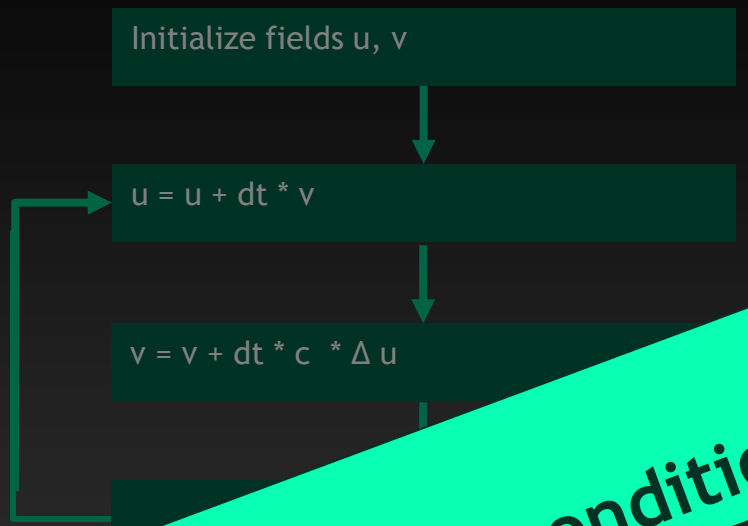


Sample: Basic Euler Solver




```
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:
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

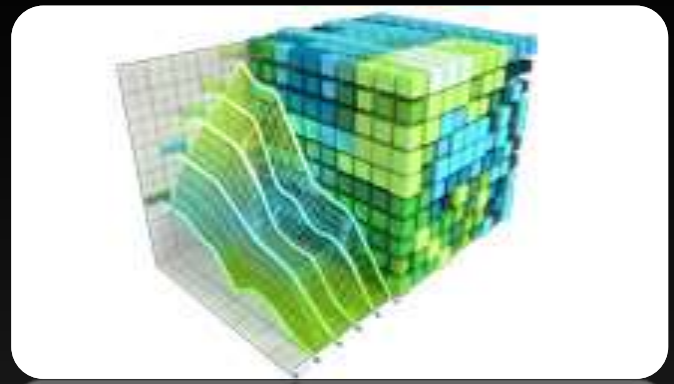
- Not exactly “profiling”, but it’s helpful information that a GPU-aware profiler would also have given you

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

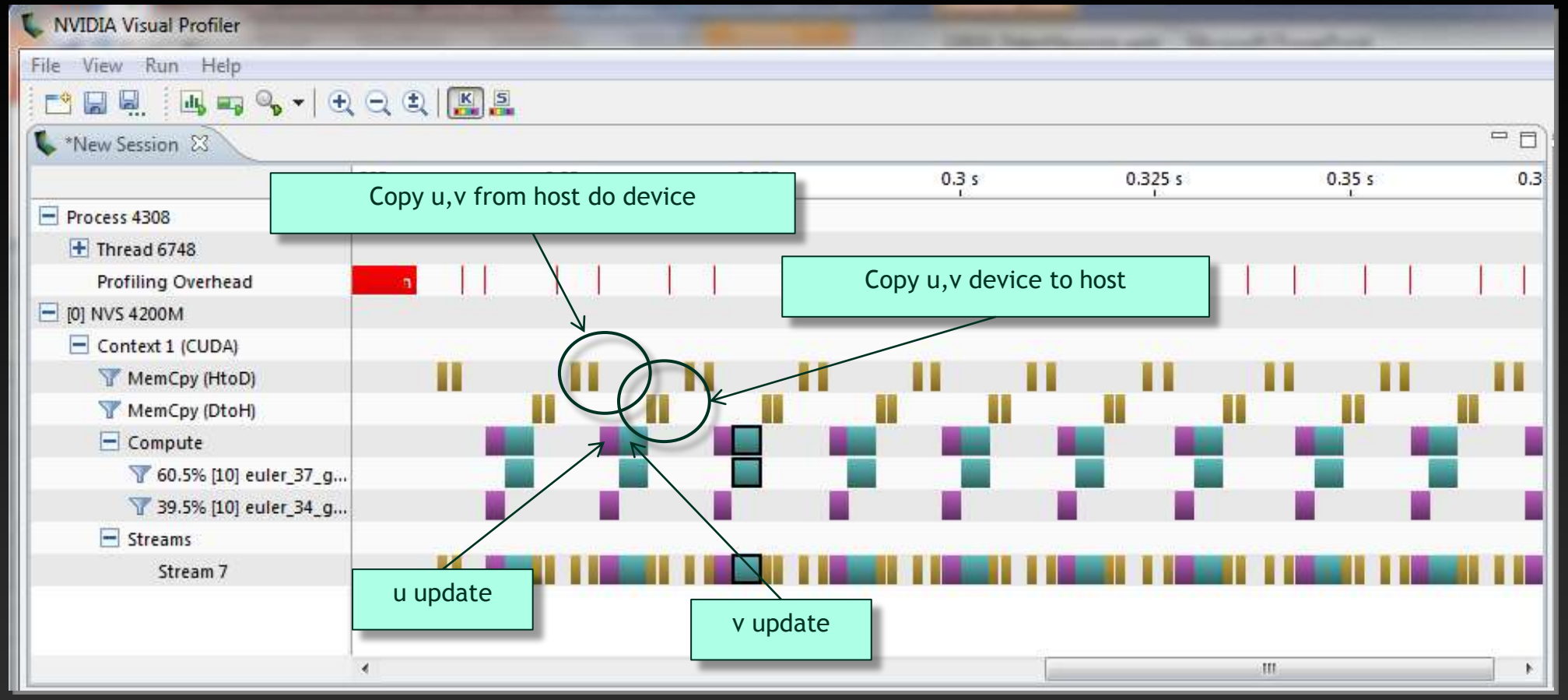
NVIDIA Visual Profiler

3rd-party profiling tools that are CUDA-aware

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



Profiling via NVVP



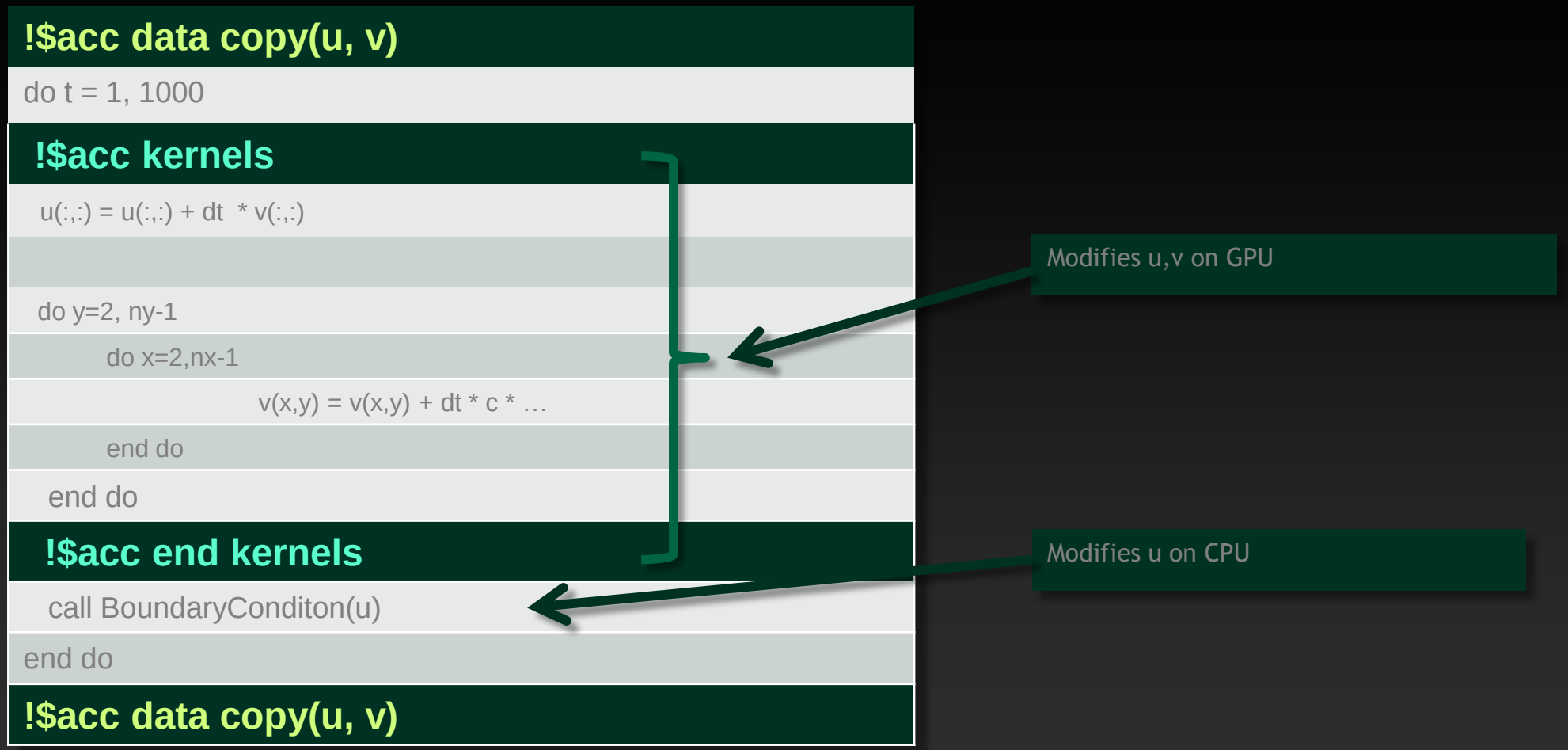
Optimization: Keep data on GPU

```
!$acc data copy(u, v)
do t = 1, 1000
  !$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 data copy(u, v)
```

u,v on GPU for the duration of the loop

Optimization: Keep data on GPU

Beware of the different address spaces!



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

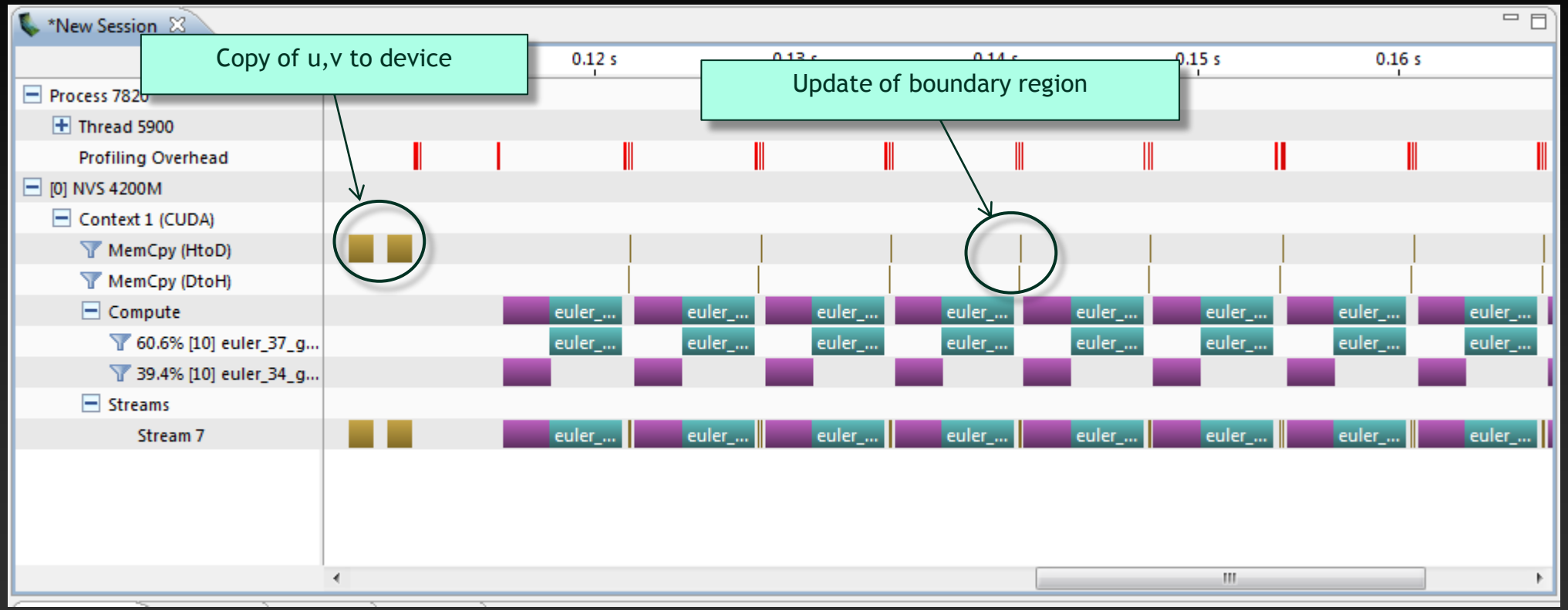
```
!$acc data copy(u, v)
do t = 1, 1000
  !$acc kernels
  u(:,:) = u(:,:) + dt  * v(:,:)

  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

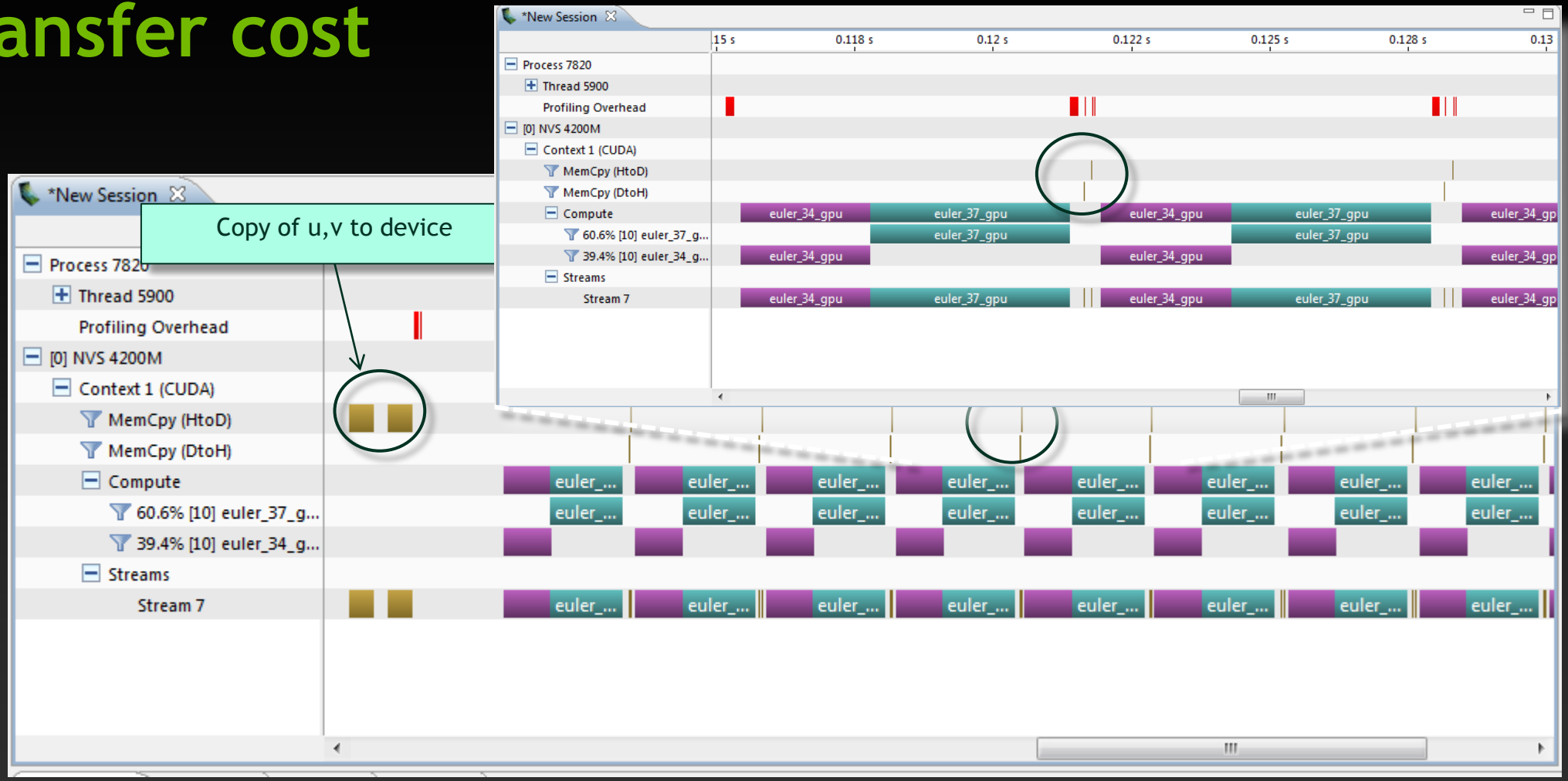
Pull fraction of u back to CPU

Push fraction of u back to GPU

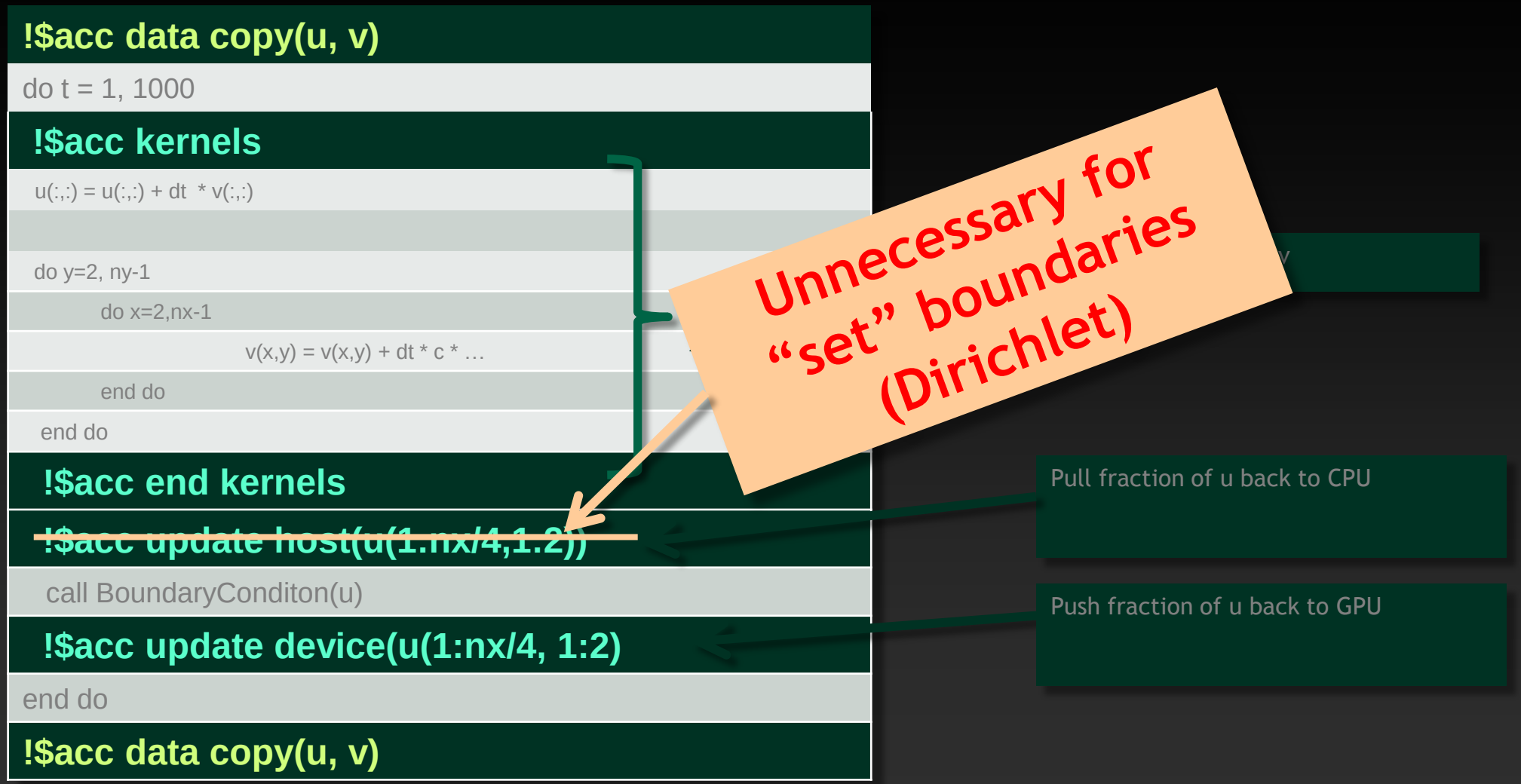
`update` significantly reduces data transfer cost



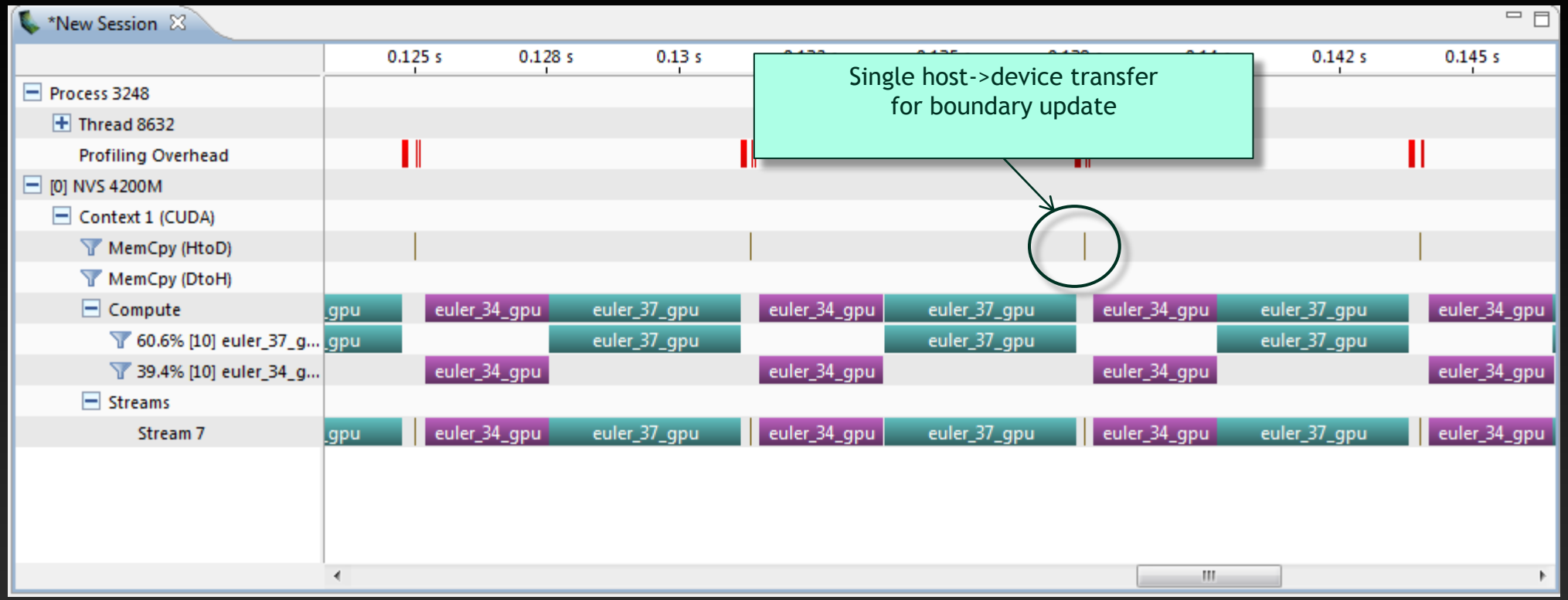
`update` significantly reduces data transfer cost



update directive for boundaries



Reducing data transfer even further



Summary of Data Motion Optimizations

- Basic OpenACC annotation

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

- Resident GPU data

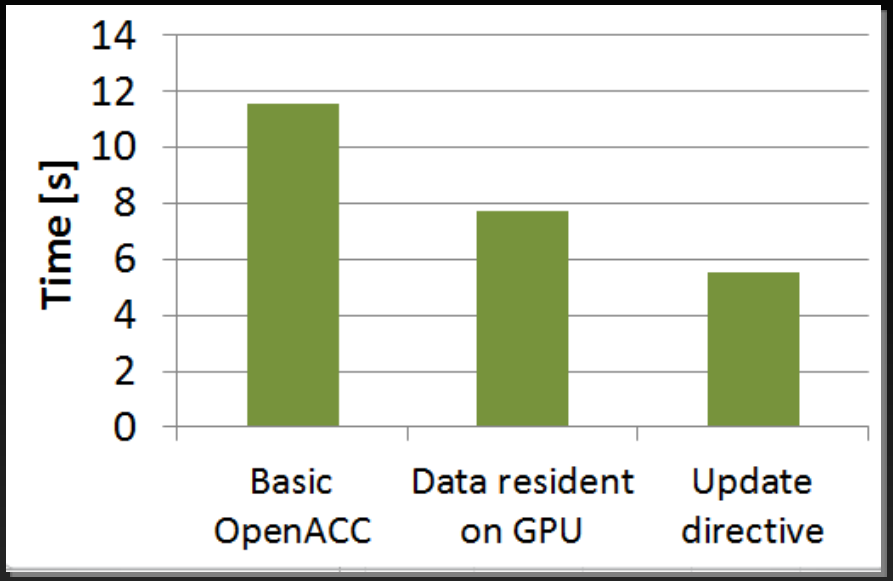
```
!$acc data copy(a)
```

- 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?

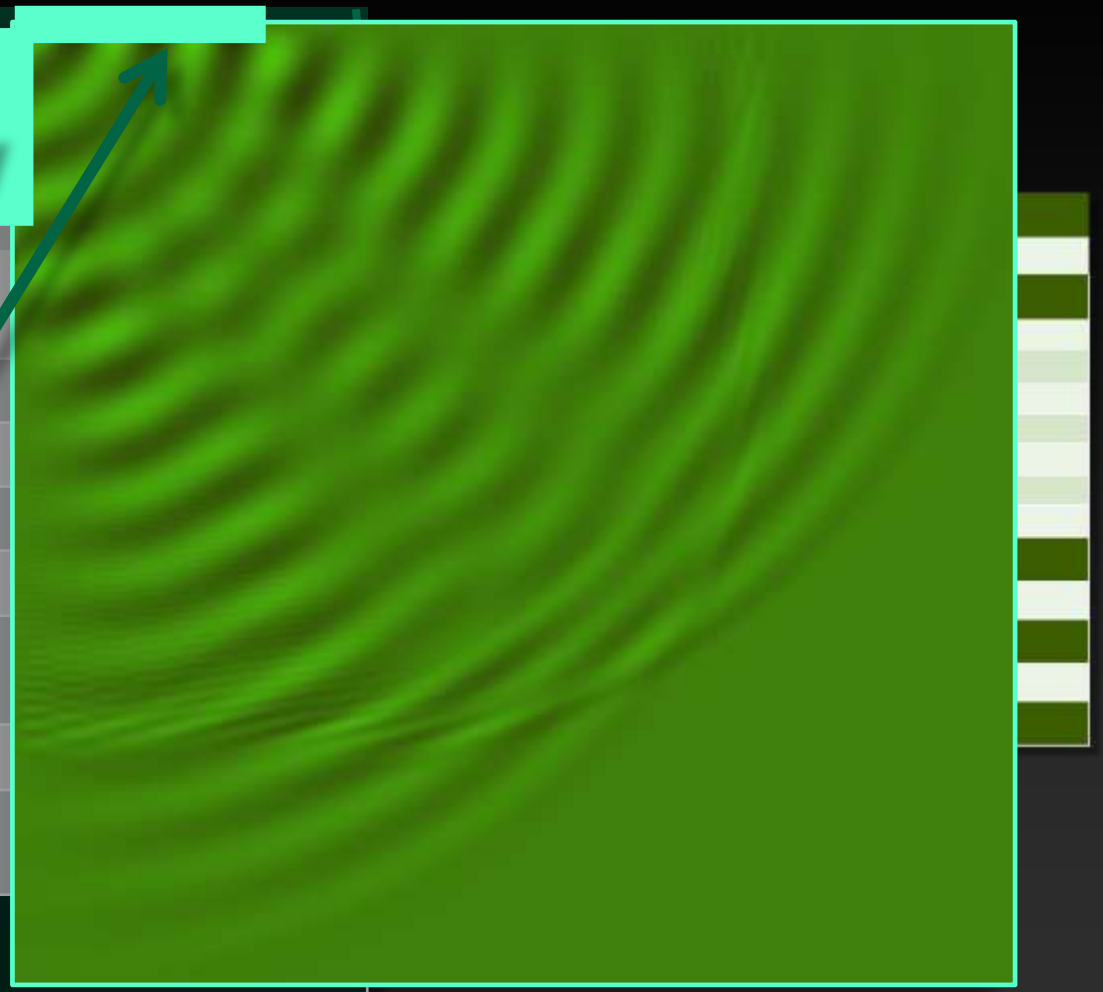
```
!$acc kernels
do y=1, 2
   do x=1, nx/4
         u(x,y) = sin(6.28*real(t) + real(x)/6.28) * &
                  cos(6.28*real(t) + real(y)/6.28)
    end do
end do
do y=1, ny/4
    do x=1, 2
         u(x,y) = (sin(6.28*real(t) + real(x)/6.28) * &
                  cos(6.28*real(t) + real(y)/6.28))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

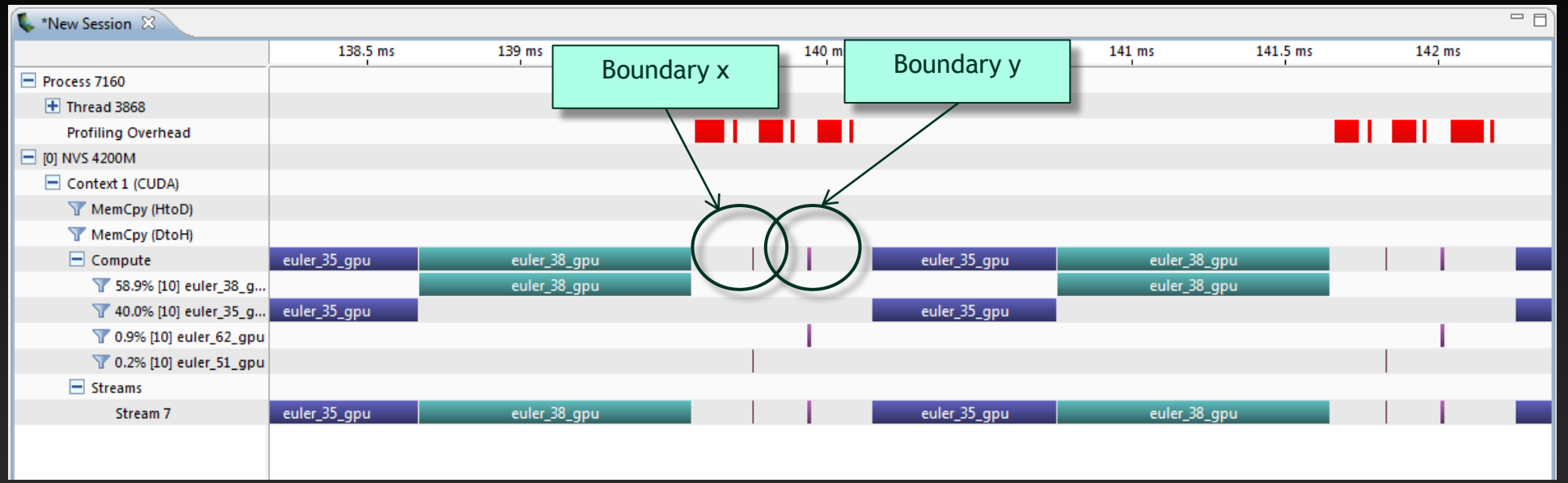
Wave launcher along
x and y

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,

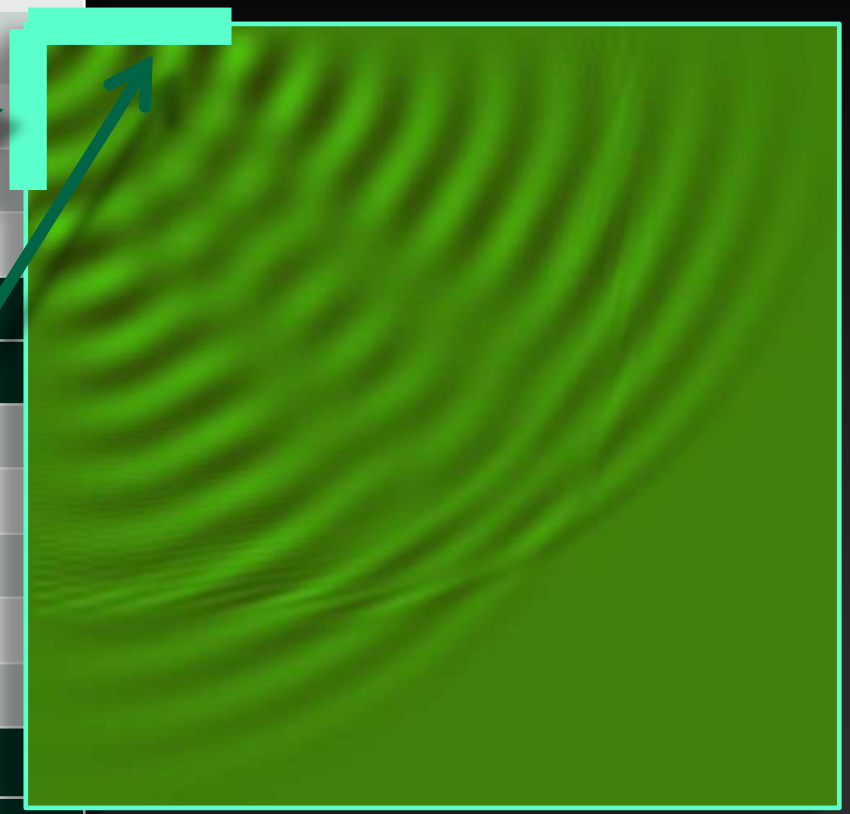
Do not wait for kernel completion

Executes concurrently with kernels

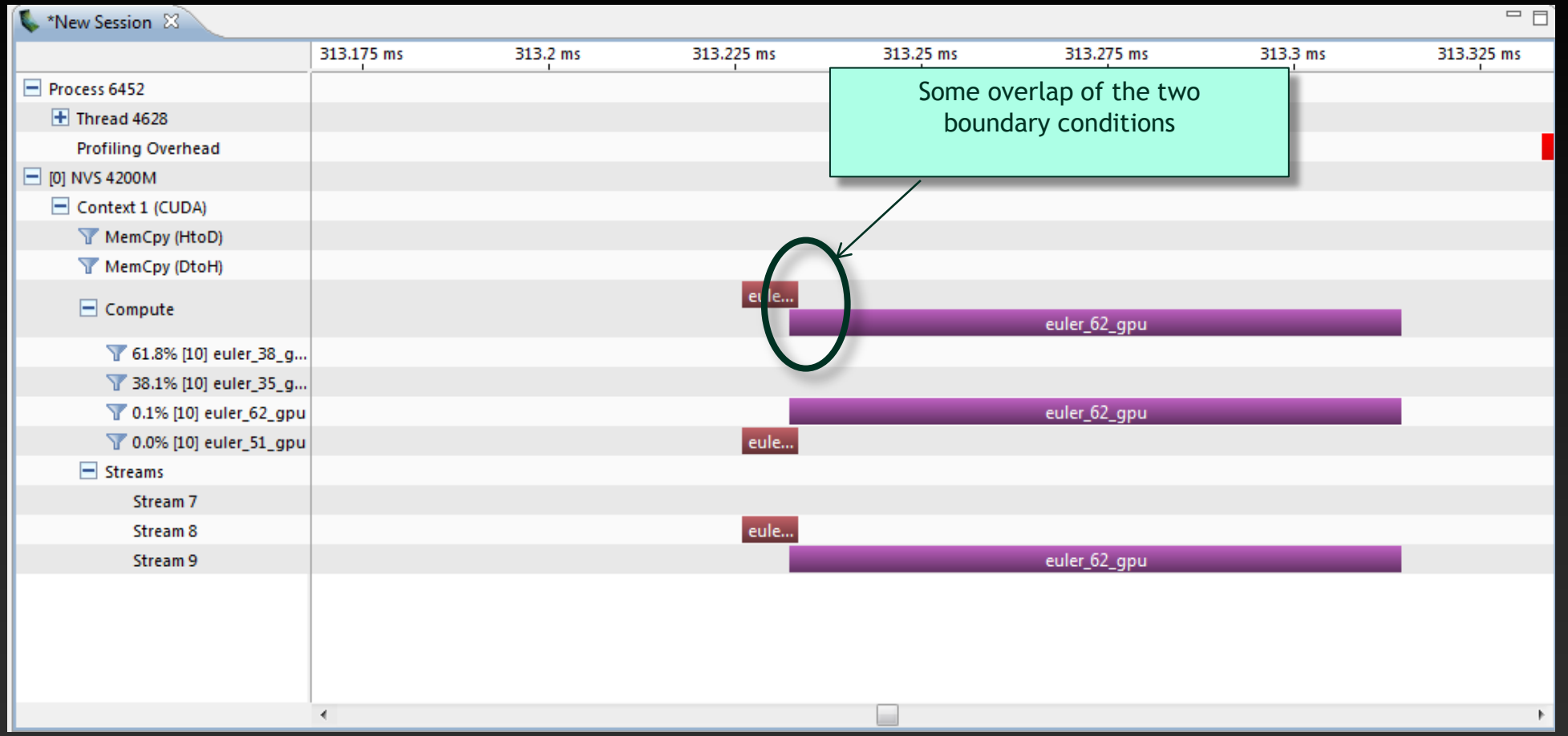
(potentially) executes concurrently with kernels

Set Boundary Regions concurrently

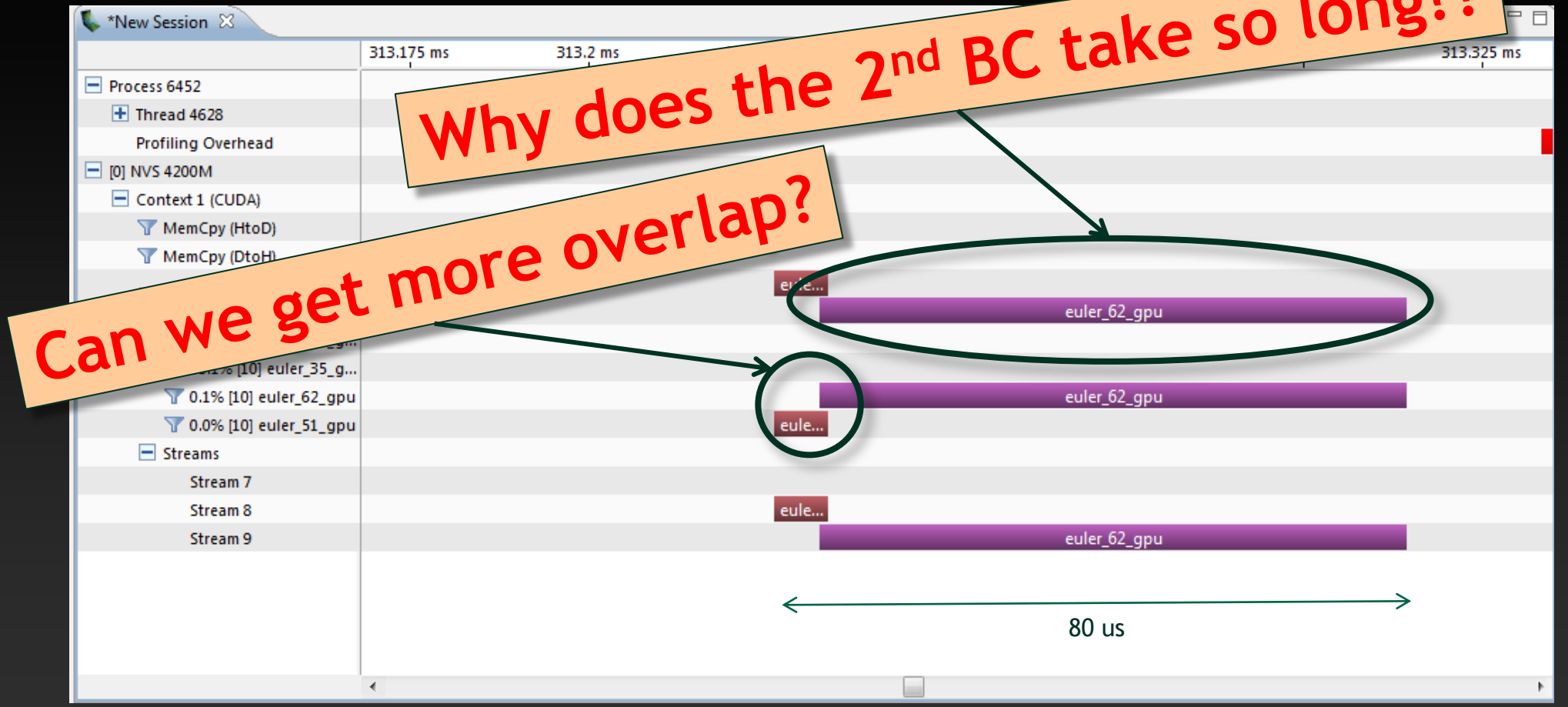
```
!$acc kernels async(1)
do y=1, 2
   do x=1, nx/4
        u(x,y) = sin(6.28*real(t) + real(x)/6.28) ..
    end do
end do
!$acc end kernels
!$acc kernels async(2)
do y=1, ny/4
    do x=1, 2
        u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
   end do
 end do
!$acc end kernels
!$acc wait
```



Async clause leads to overlapping kernels



Async clause leads to overlapping kernels

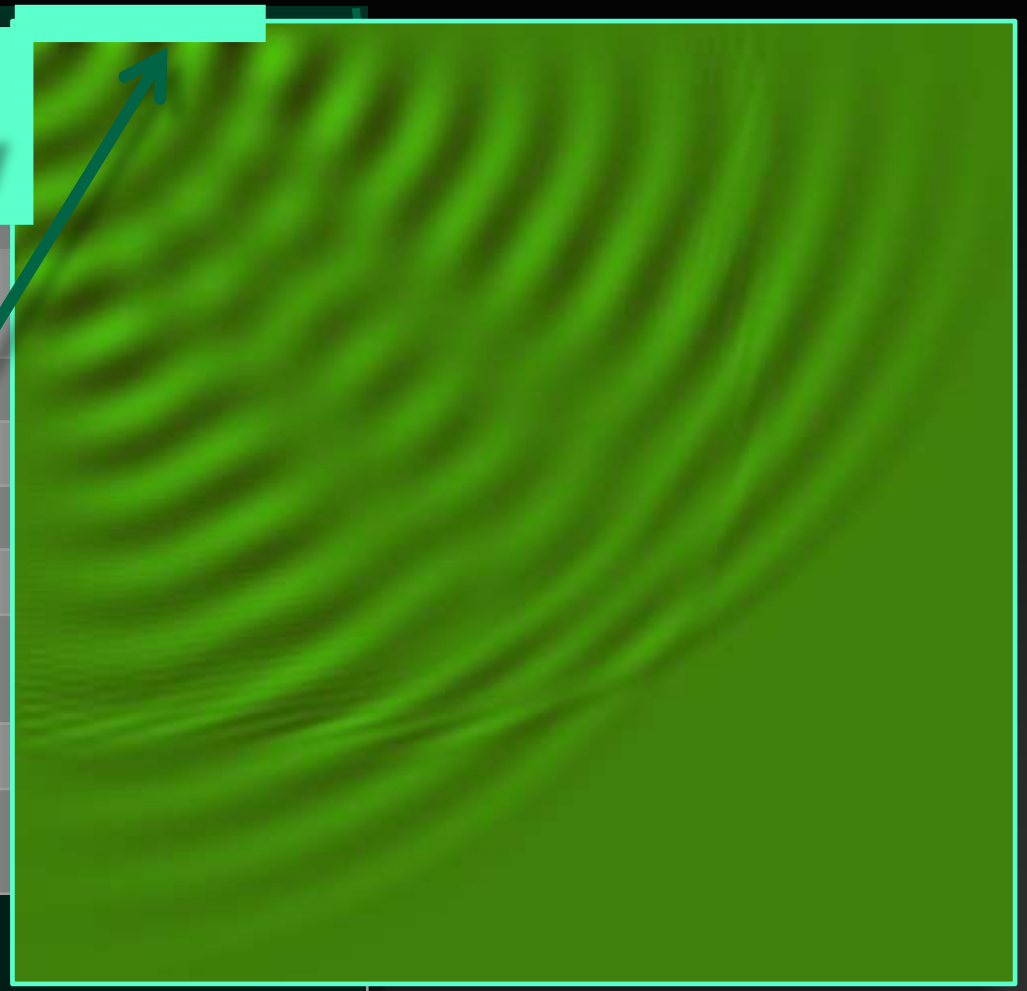


Why does the 2nd BC take so much longer?

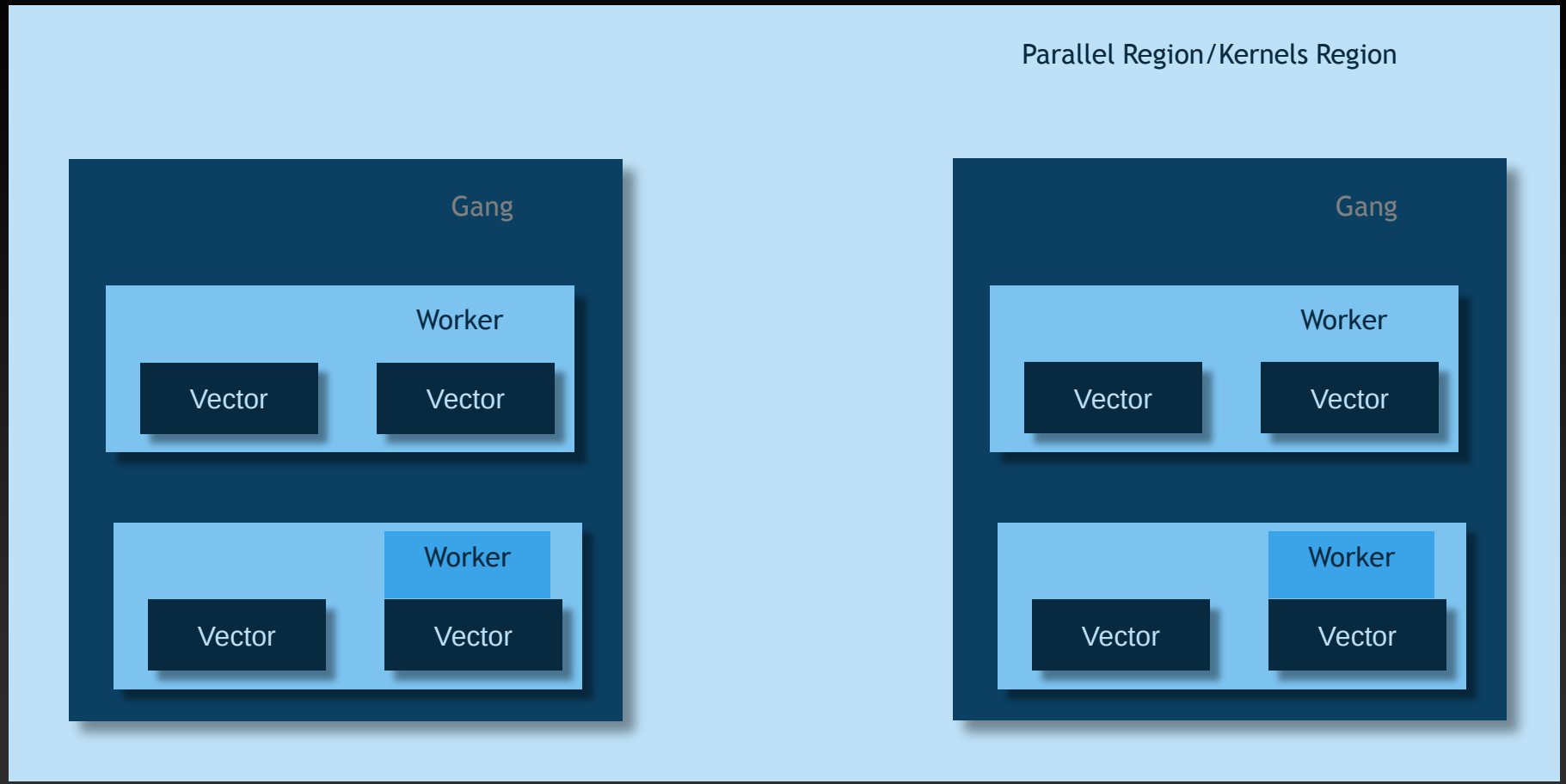
```
!$acc kernels
do y=1, 2
   do x=1, nx/4
      u(x,y) = sin(6      rea (x)/6.28) * &
                  cos(    ) + real(y)/6.28)
   end do
end do
do y=1, ny/4
   do x=1, 2
      u(x,y) = (sin(      ) + real(x)/6.28) * &
                  cos(t) + real(y)/6.28))
   end do
end do
!$acc end kernels
```

Fast

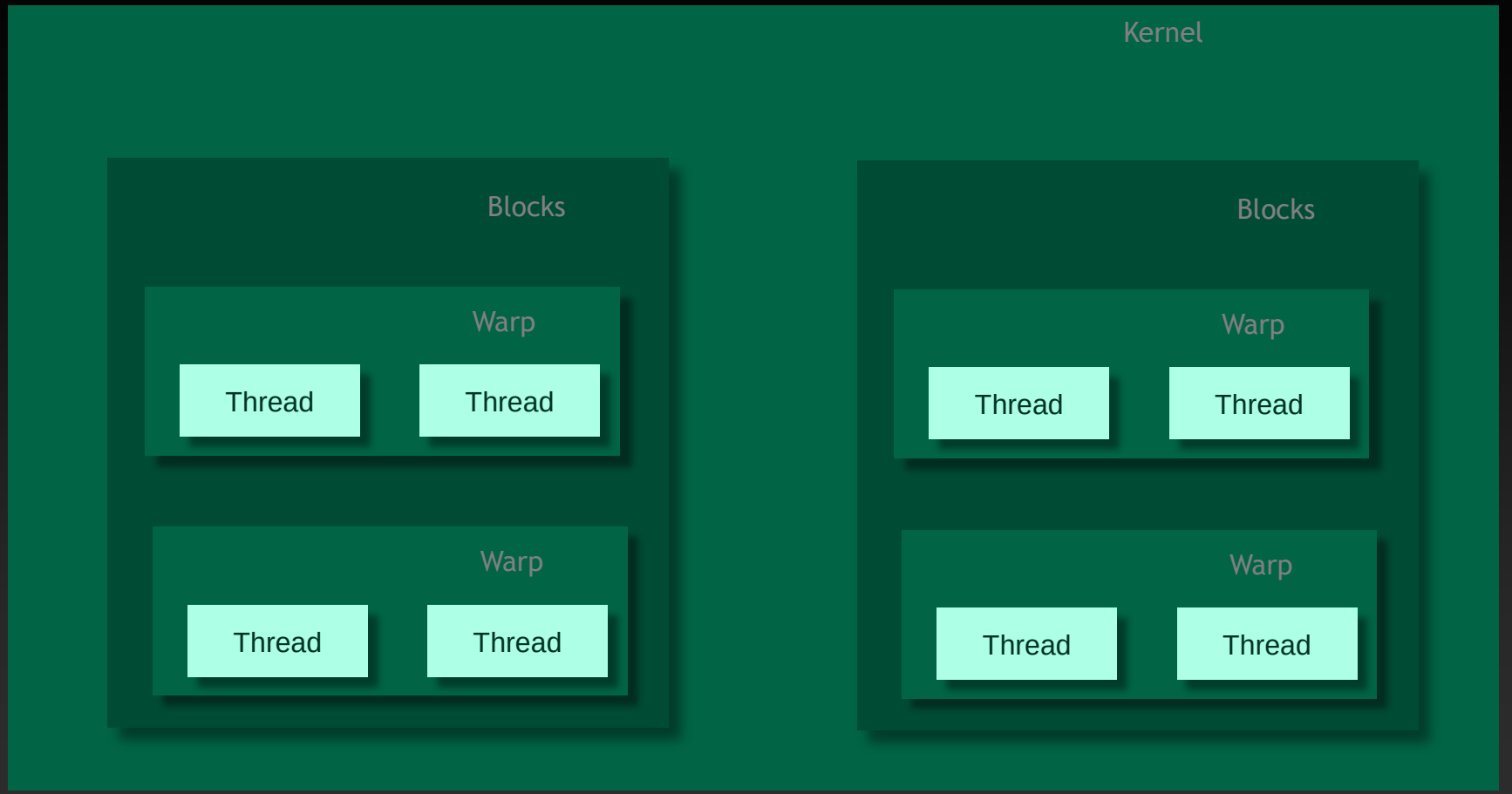
Slow



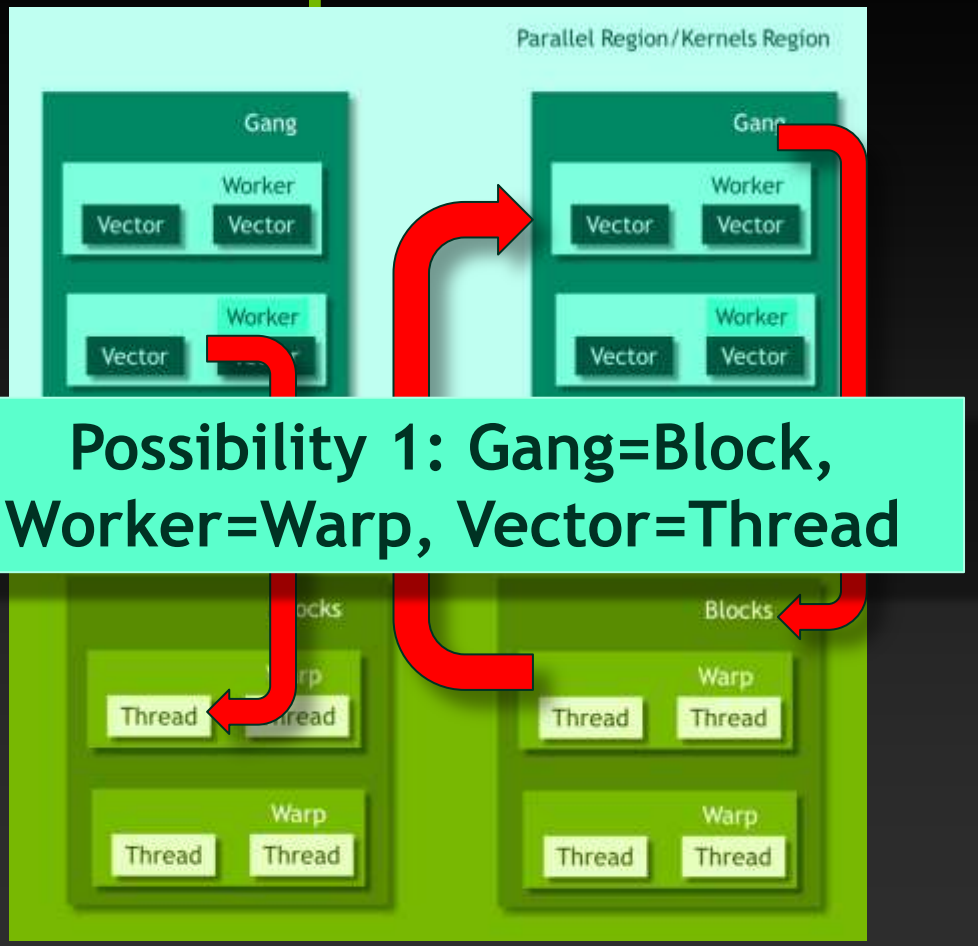
OpenACC's three levels of parallelism



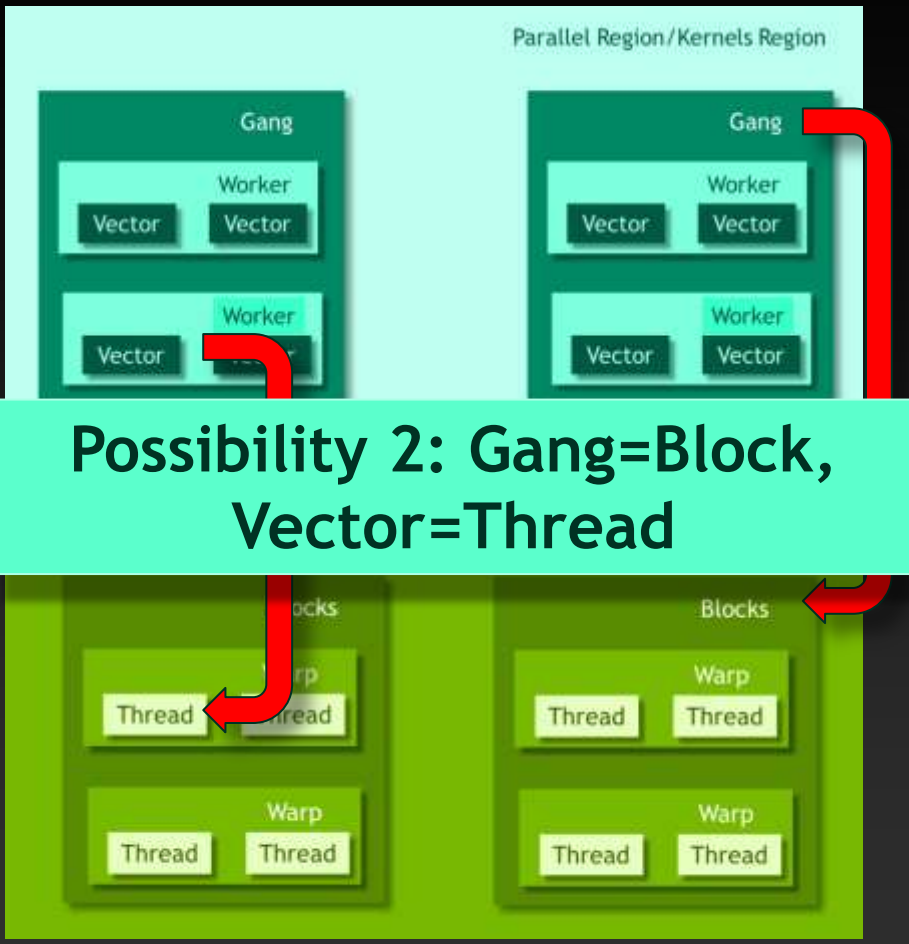
The CUDA execution model



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

OpenACC

CUDA

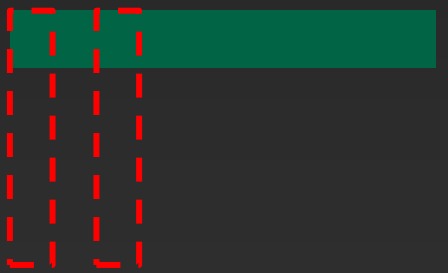
```
     ng, vector(128) ! blockidx%
     ng present_or_copy(u(:,:))
46, Loop is parallelizable
47, Loop is parallelizable
    Accelerator kernel generated
    46, !$acc loop gang ! blockidx%y
    47, !$acc loop gang, vector(128) ! blockidx%x threadidx%x
54, Generating present_or_copy(u(:,:))
55, Loop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

Mapping of boundary kernels to CUDA

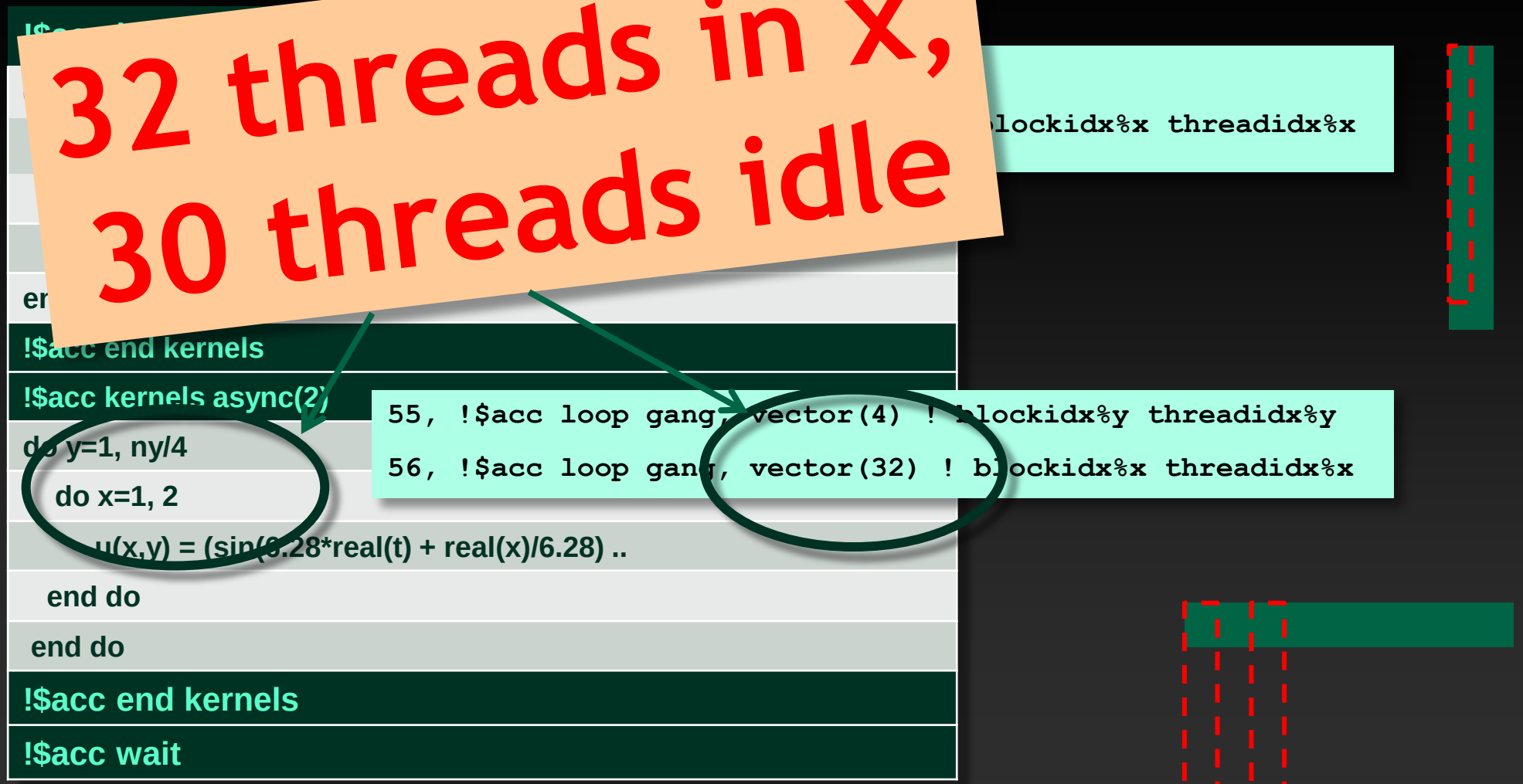
```
!$acc kernels async(1)
do y=1, 2
  do x=1, nx/4
      u(x,y) = sin(6.28*real(t) + real(x)/6.28) ..
  end do
end do
!$acc end kernels
!$acc kernels async(2)
do y=1, ny/4
  do x=1, 2
      u(x,y) = (sin(6.28*real(t) + real(x)/6.28) ..
  end do
end do
!$acc end kernels
!$acc wait
```

```
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



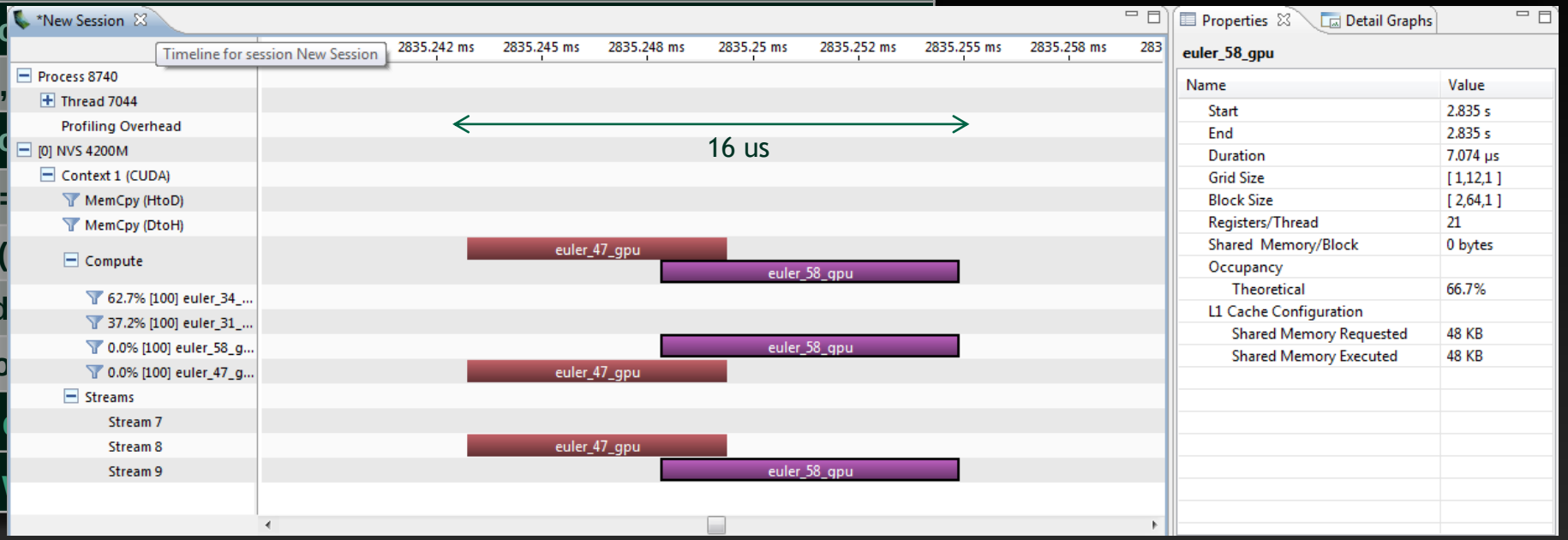
Loop scheduling via `vector` clause

```
!$acc kernels async(2)
!$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?

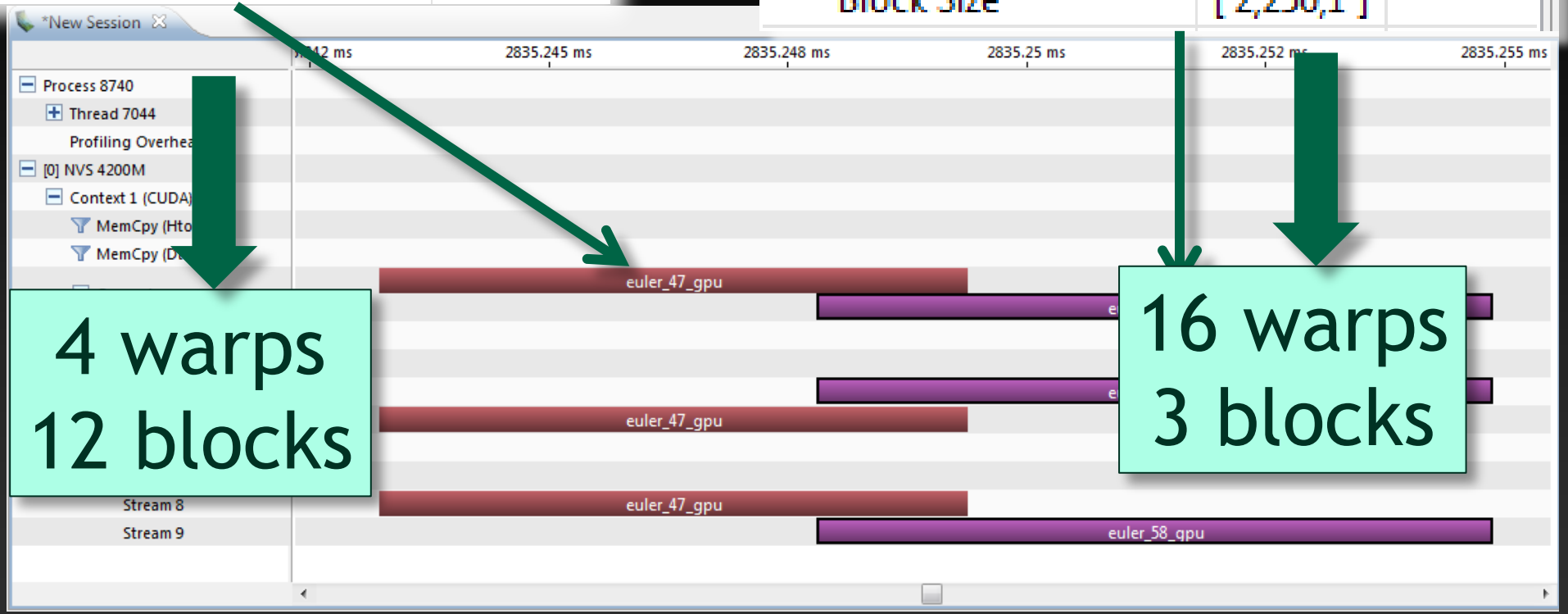
- Overlap requires sufficient resource for both kernels
- Our GPU: 1 SM, 8 Blocks, 48 warps
- What resources do the kernels use?

1 Fermi SM; 8 blocks; 48 warps

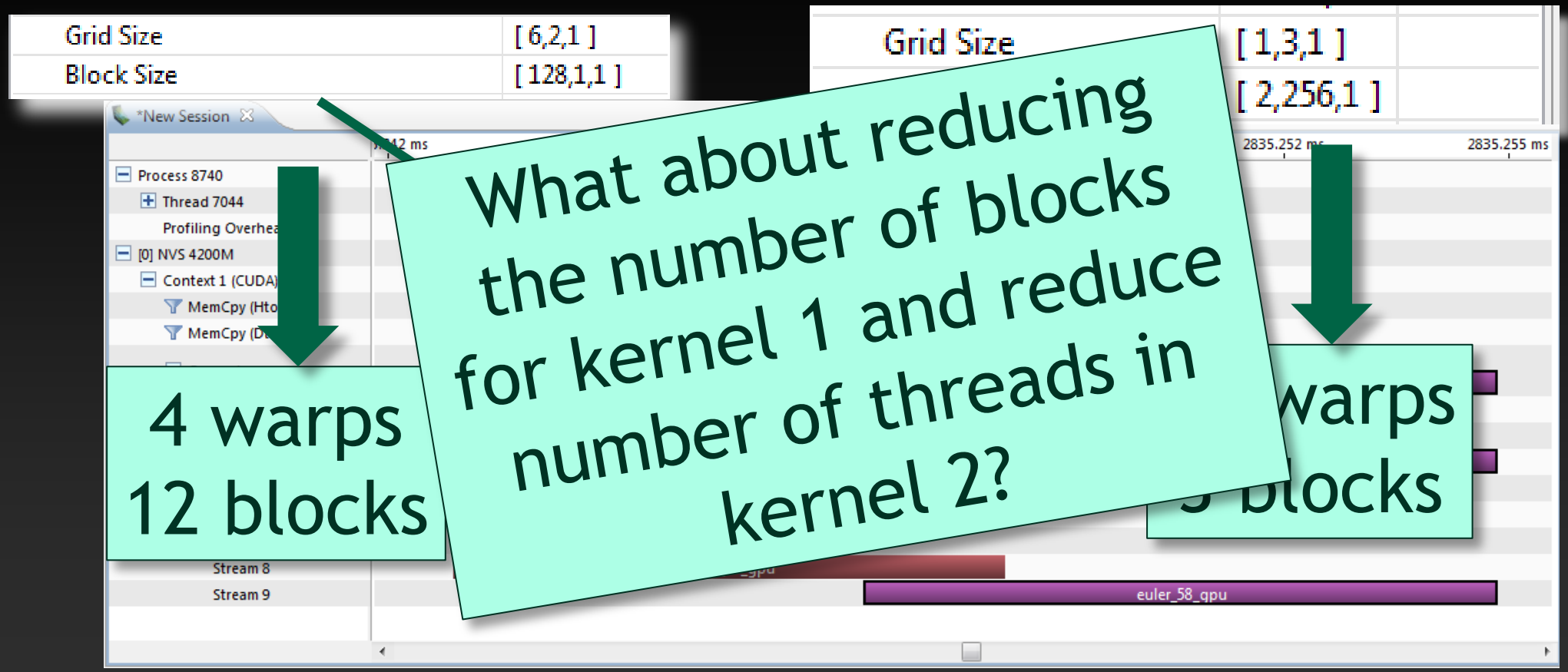
Can we achieve better overlap?

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

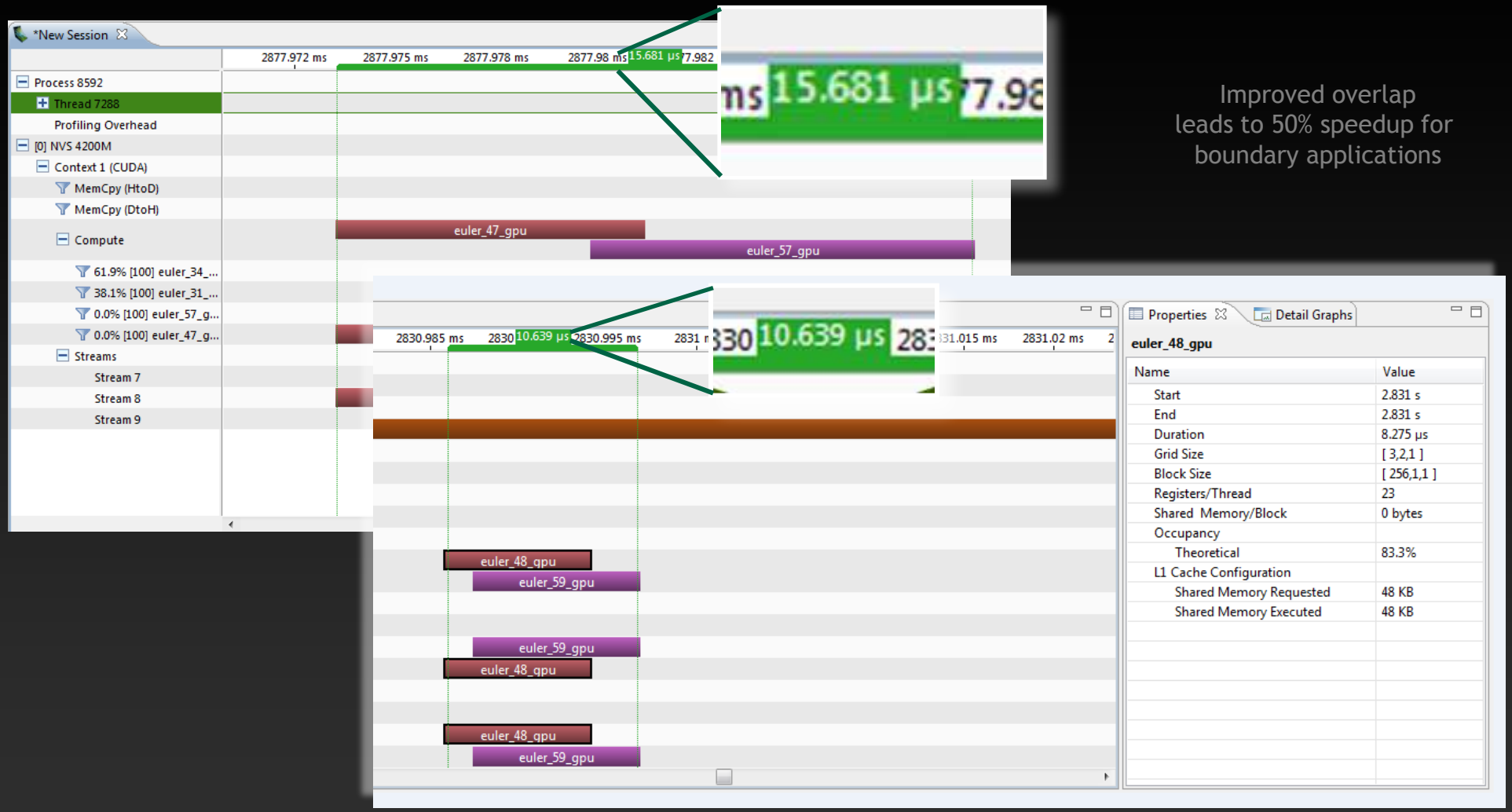
Grid Size	[1,3,1]
Block Size	[2,256,1]



Can we achieve better overlap?



Can we achieve better overlap?

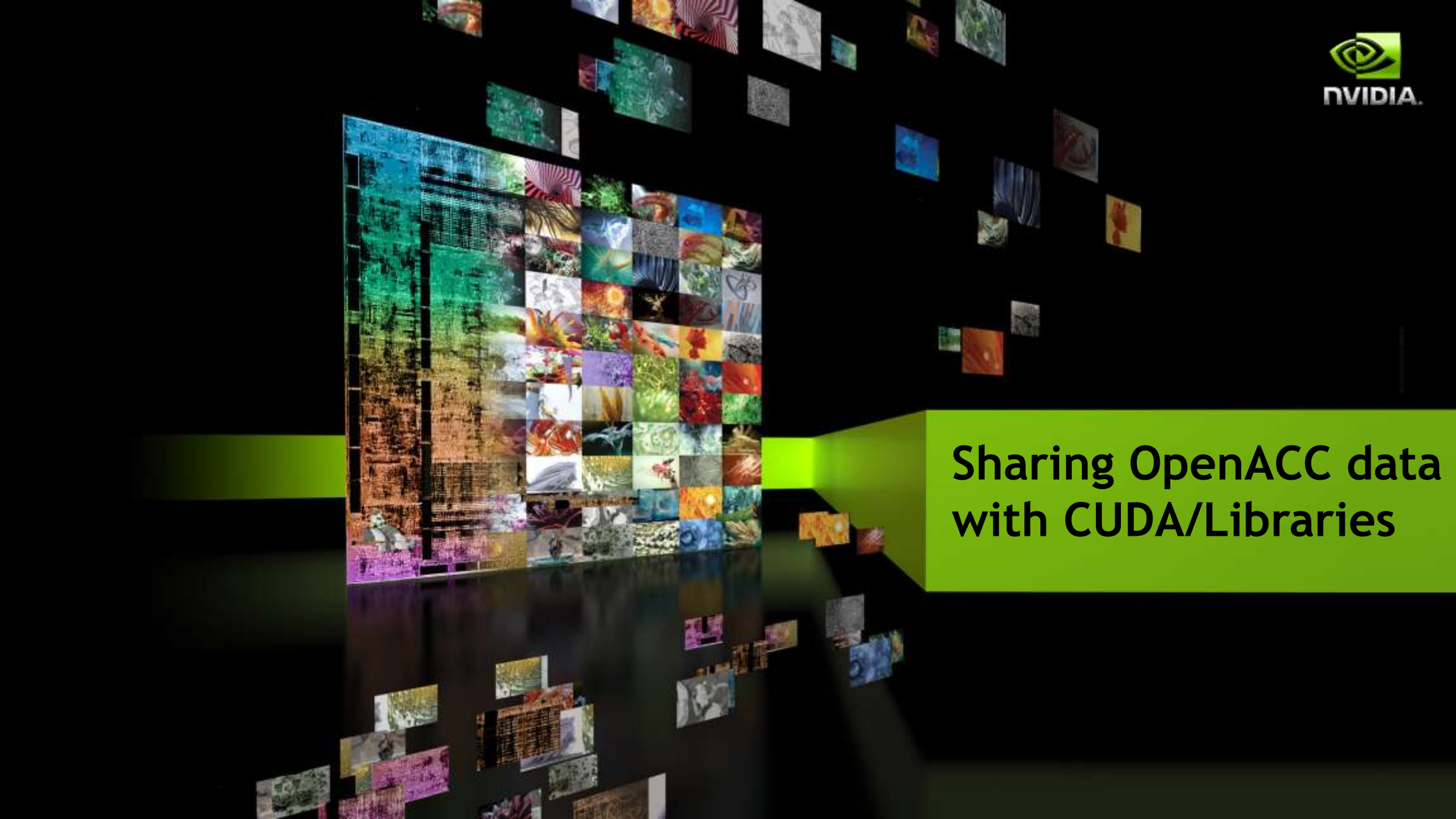


Improved overlap
leads to 50% speedup for
boundary applications

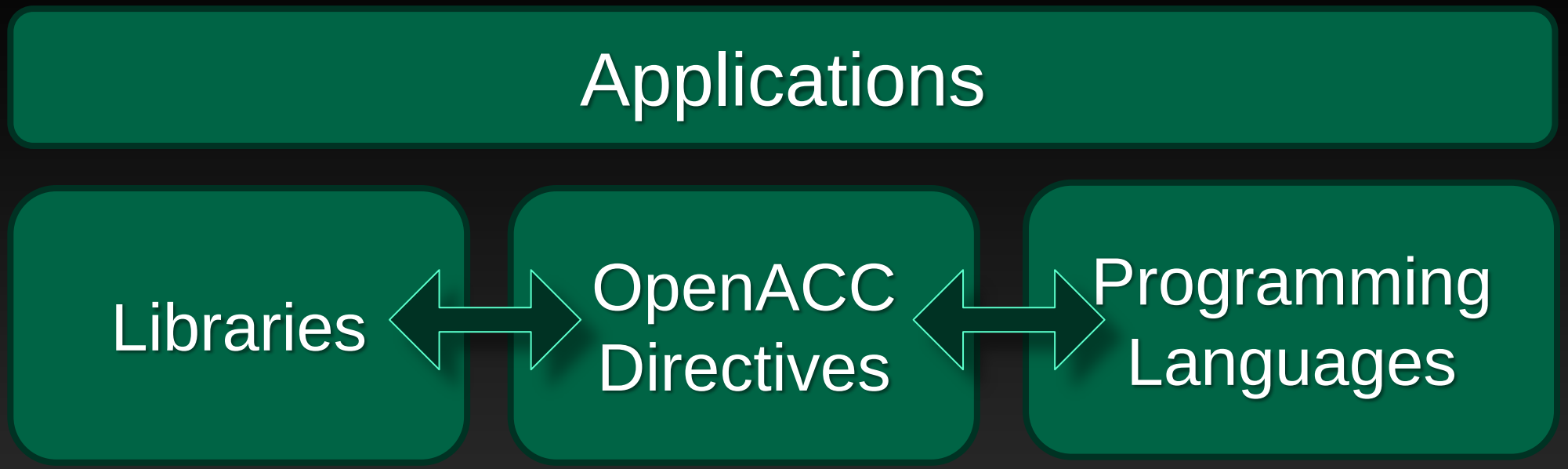
Summary of Optimization Steps

- 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

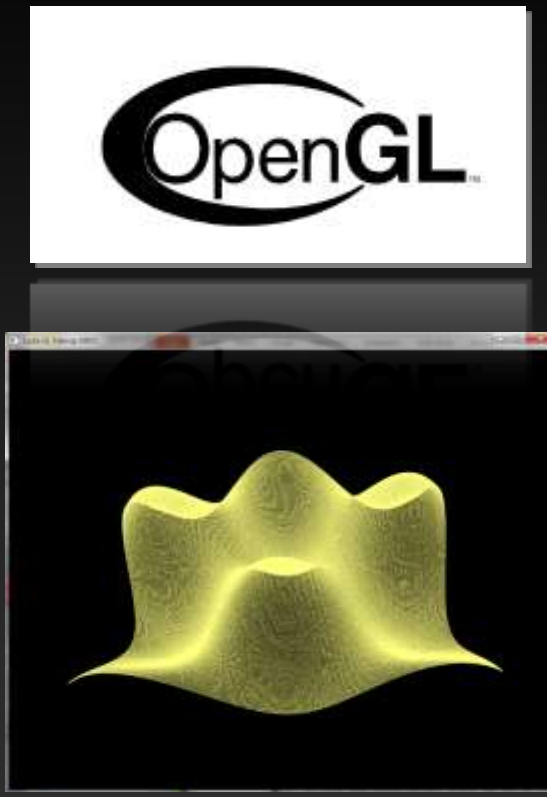
Task 5: Compute field magnitude using cublas

- 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

`deviceptr` example: Interop with OpenGL

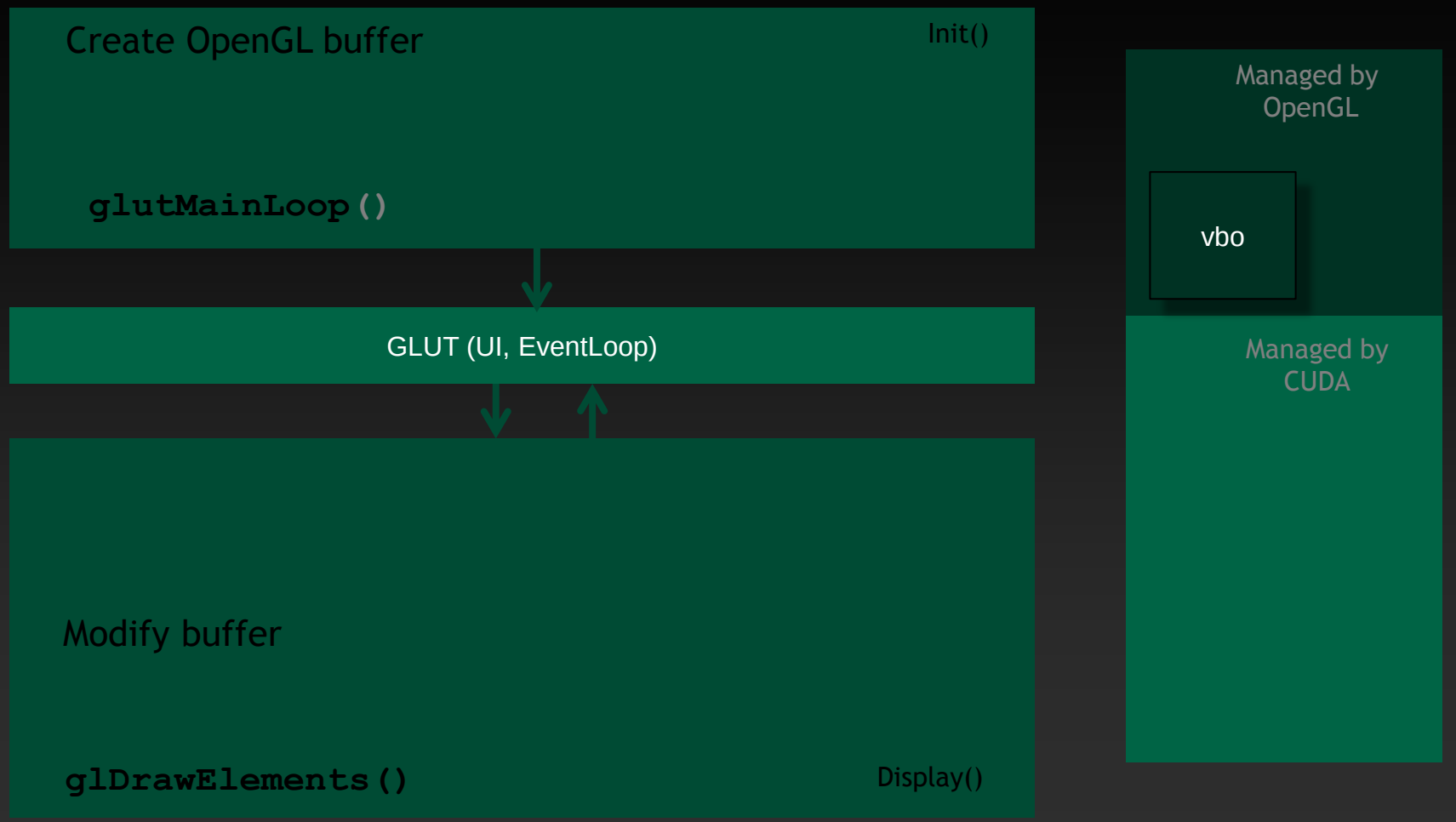
- OpenGL: Programmers interface to Graphics Hardware
- API to specify
  - Primitives: points, lines, polygons
  - Properties: colors, lighting, textures, ..
  - View: camera position and perspective



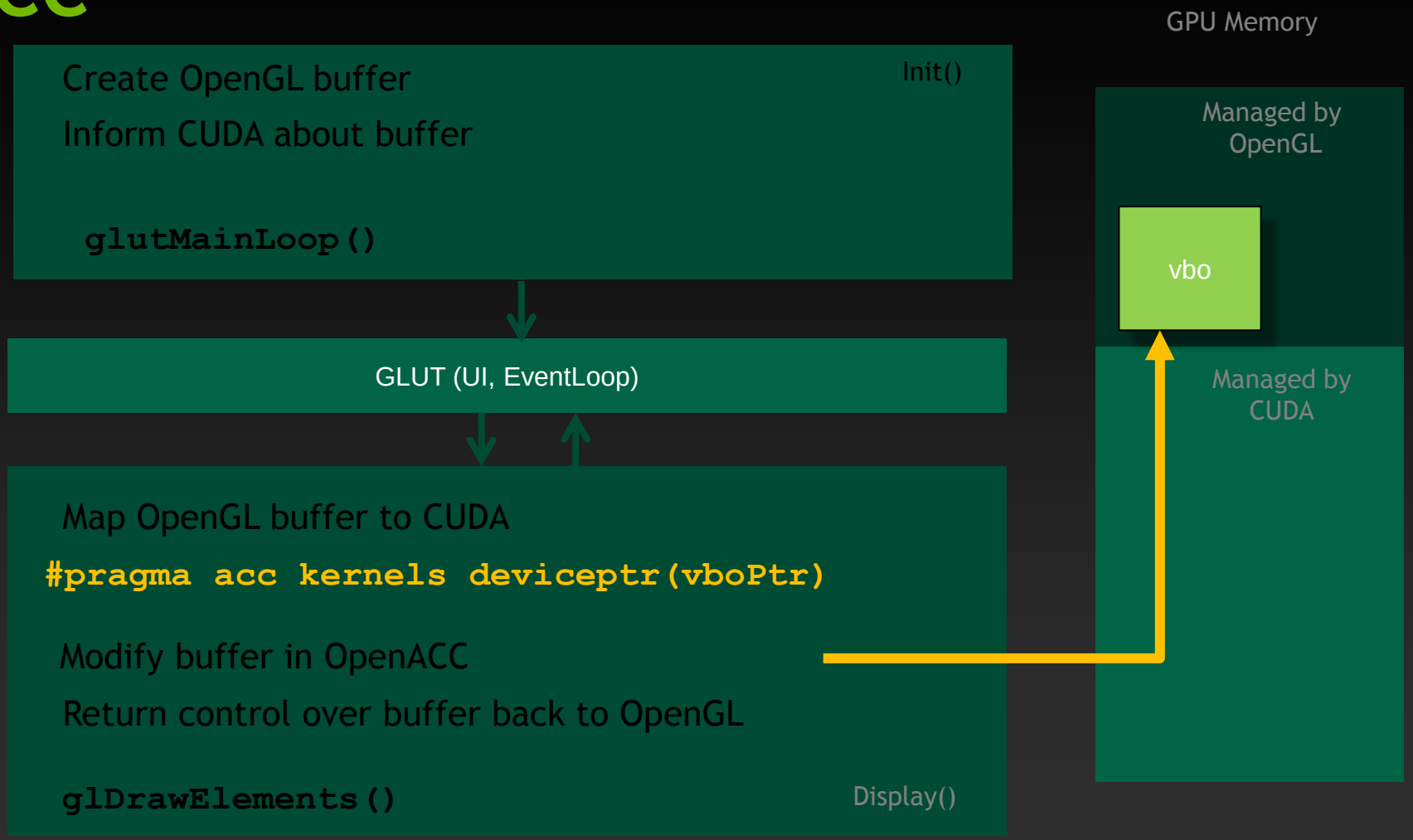
- Manages data on GPU

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

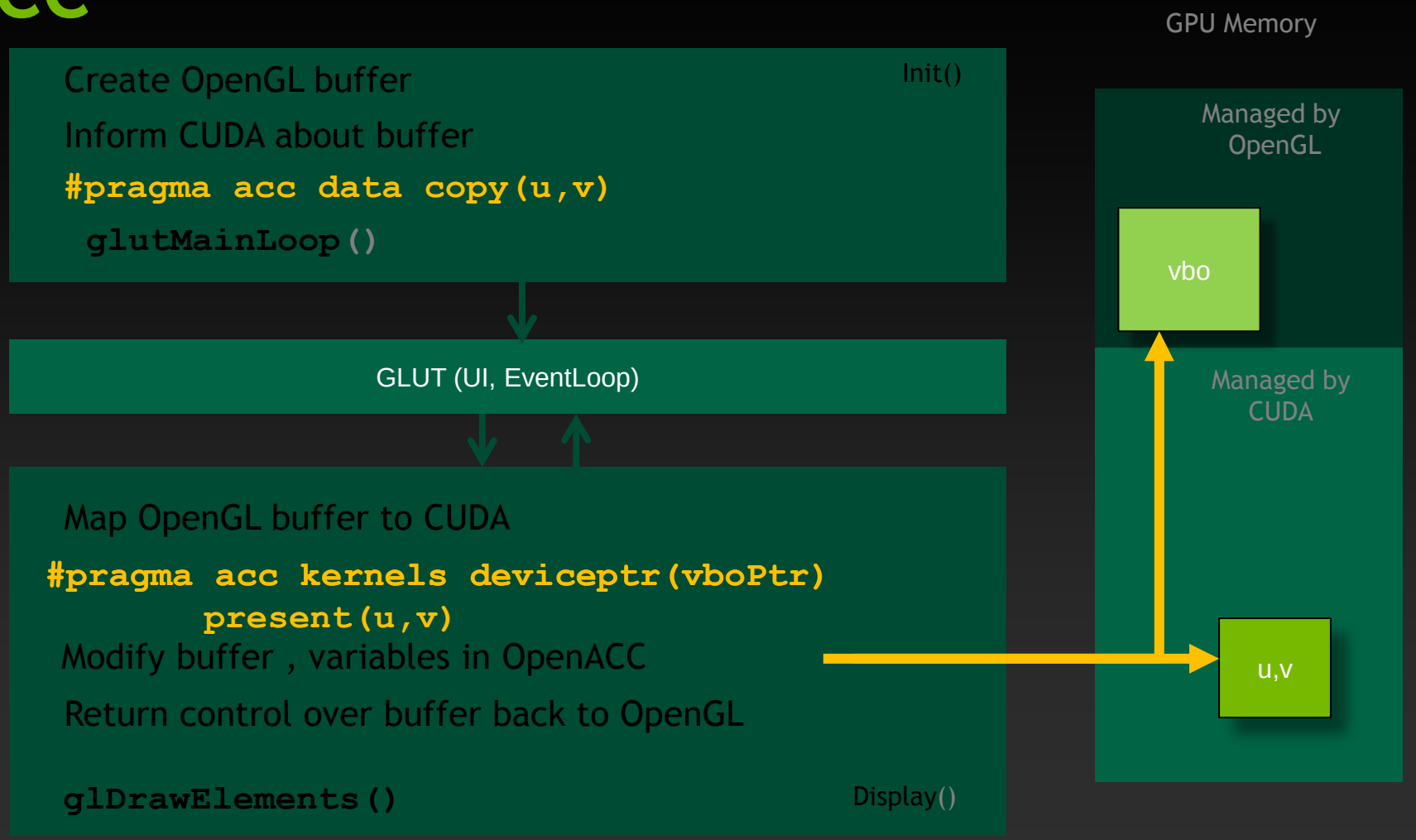
Basic components of an OpenGL application



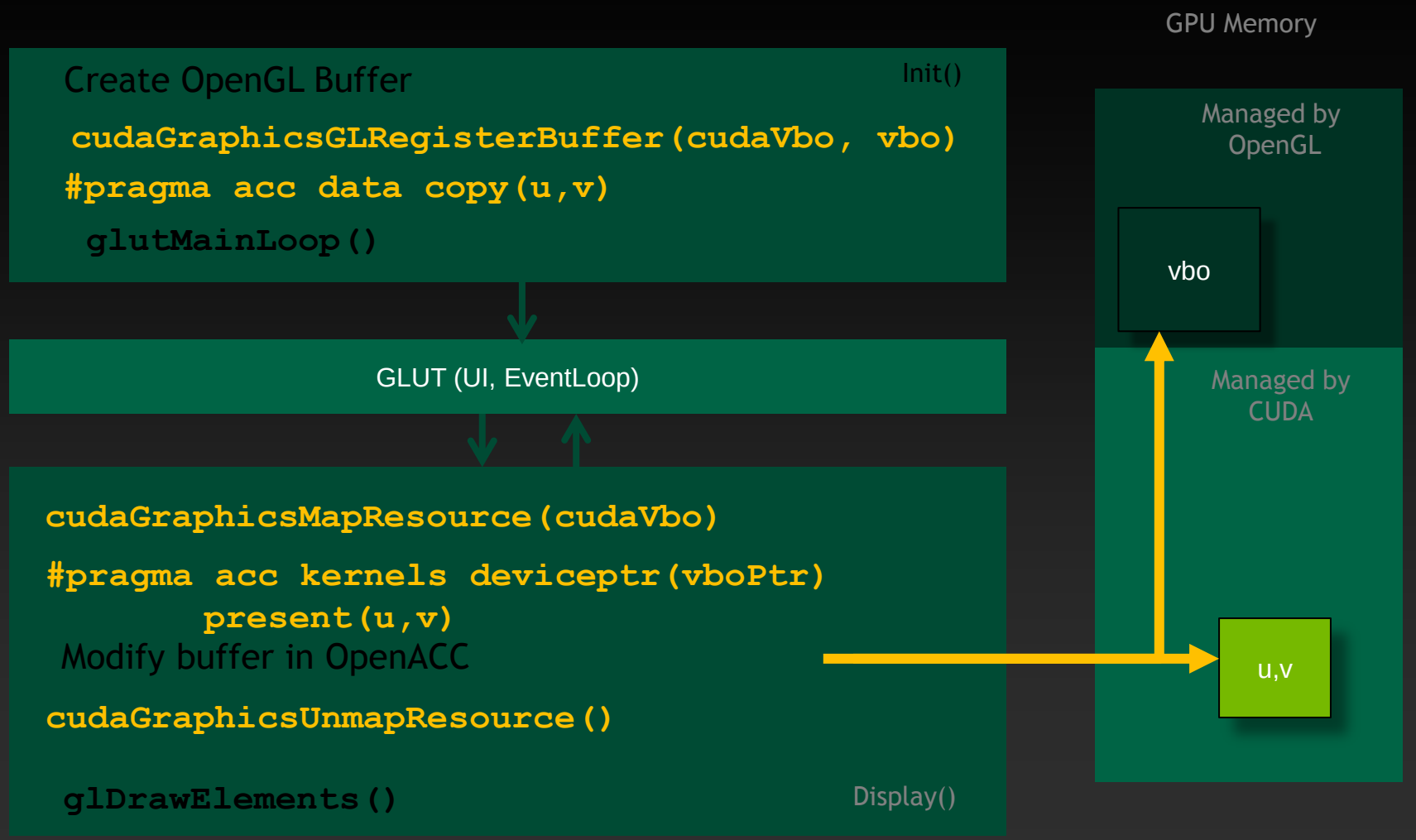
OpenGL Buffer Modification with OpenACC



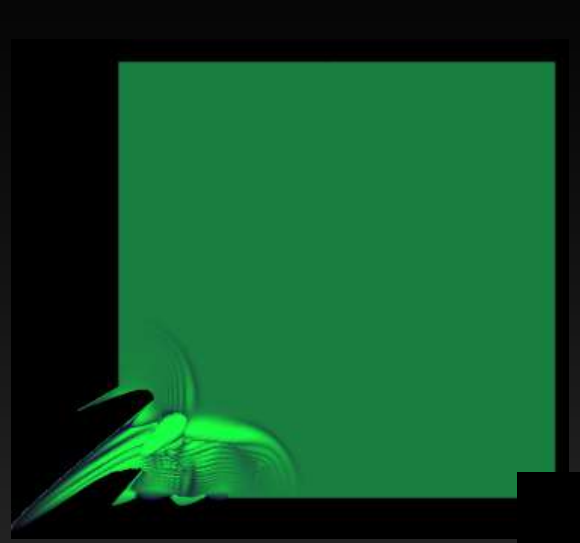
OpenGL Buffer Modification with OpenACC

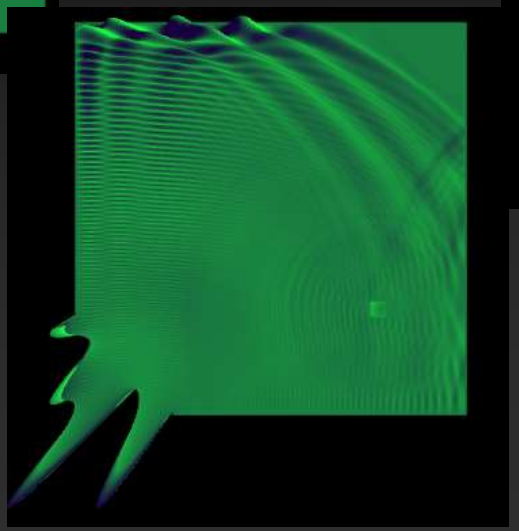


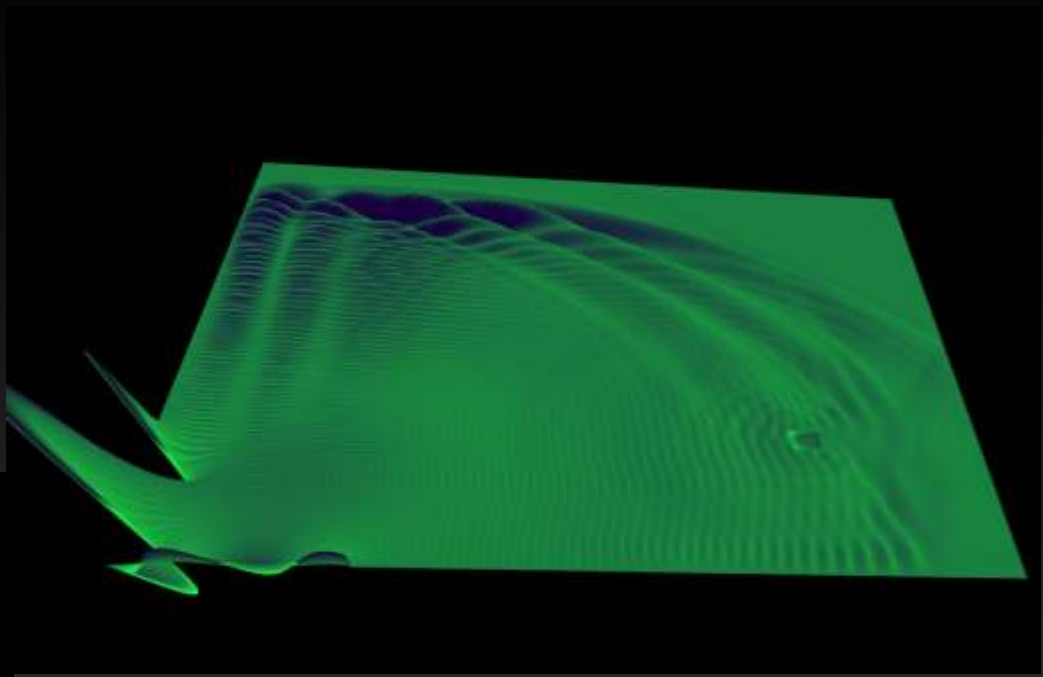
Some relevant Interop API calls



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







DEMO
(These are just backup pictures)


NVIDIA.
Communication & IO
with OpenACC

Calling MPI with OpenACC (Standard MPI)


```
!$acc data copy(A)
!$acc parallel loop
do i=1,N
  …
enddo
!$acc end parallel loop

call neighbor_exchange(A)

!$acc parallel loop
do i=1,N
  …
enddo
!$acc end parallel loop
!$acc end data
```

Array "A" resides in GPU memory.

Routine contains MPI and requires "A."

Array "A" returns to CPU here.

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.

Calling MPI with OpenACC (Standard MPI)


```
!$acc data copy(A)
!$acc parallel loop
do i=1,N
 …
enddo
!$acc end parallel loop
!$acc update host(A)
call neighbor_exchange(A)
!$acc update device(A)
!$acc parallel loop
do i=1,N
 …
enddo
!$acc end parallel loop
!$acc end data
```

Copy “A” to CPU for MPI.

Return “A” after MPI to GPU.

OpenACC host_data Directive

Programmer specifies that host arrays should be used within this section, unless specified with `use_device`. This is useful when calling libraries that expect GPU pointers.

```
!$acc host_data use_device(a)
call MPI_Sendrecv(a,…)
!$acc end host_data
```

Pass the device copy of "a" to subroutine.

```
#pragma host_data use_device(a)
{
cublasDgemm(…,a,…);
}
```

Pass the device copy of "a" to function.

This directive allows interoperability with a variety of other technologies, CUDA, accelerated libraries, OpenGL, etc.

Calling MPI with OpenACC (GPU-aware MPI)


```
!$acc data copy(A)
!$acc parallel loop
do i=1,N
 …
enddo
!$acc end parallel loop
!$acc host_data use_device(A)
call neighbor_exchange(A)
!$acc end host_data
!$acc parallel loop
do i=1,N
 …
enddo
!$acc end parallel loop
!$acc end data
```

Pass device "A" directly to a GPU-aware MPI library called in neighbor_exchange.

*More information about GPU-aware MPI libraries is available in other sessions, please see your agenda.

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)

GPU TECHNOLOGY CONFERENCE

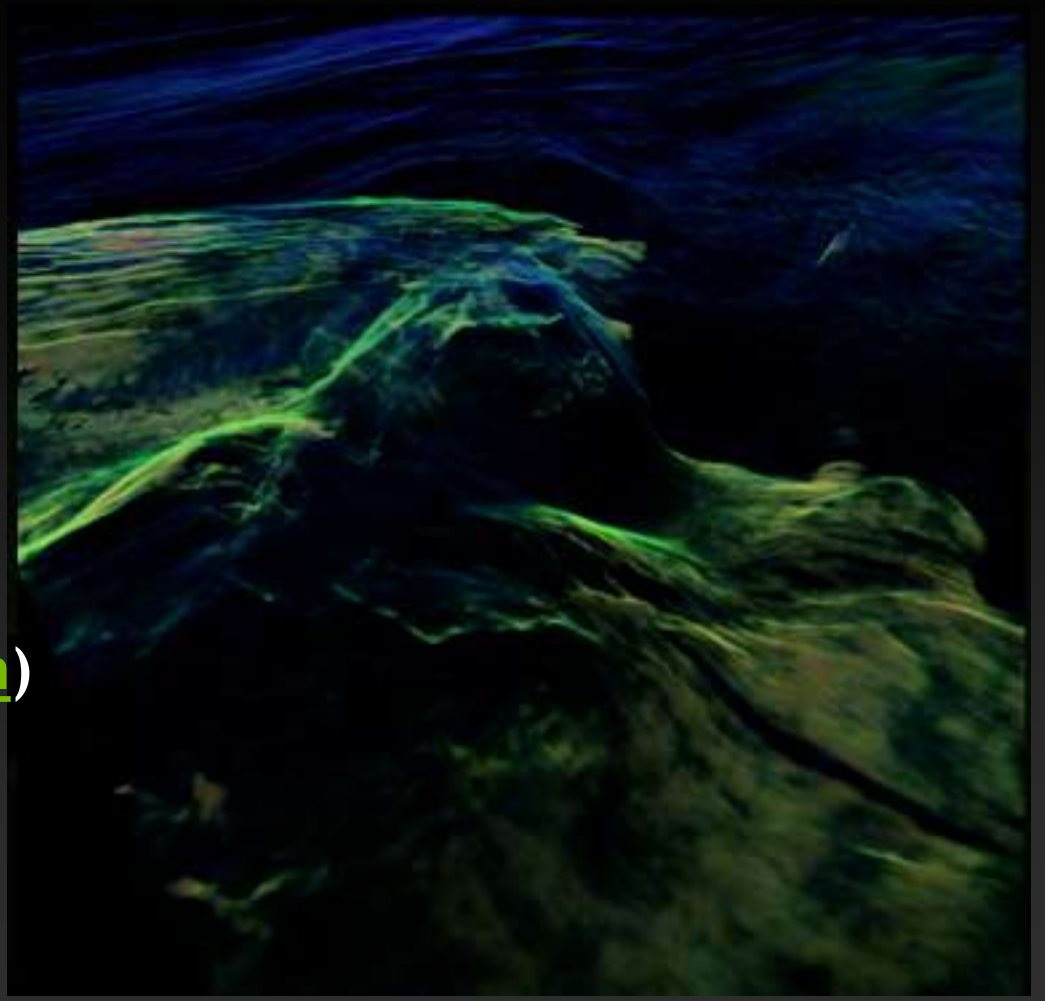
March 24-27, 2014 | San Jose, CA

LEARN | NEWS | CONNECT | FUN

Thank You

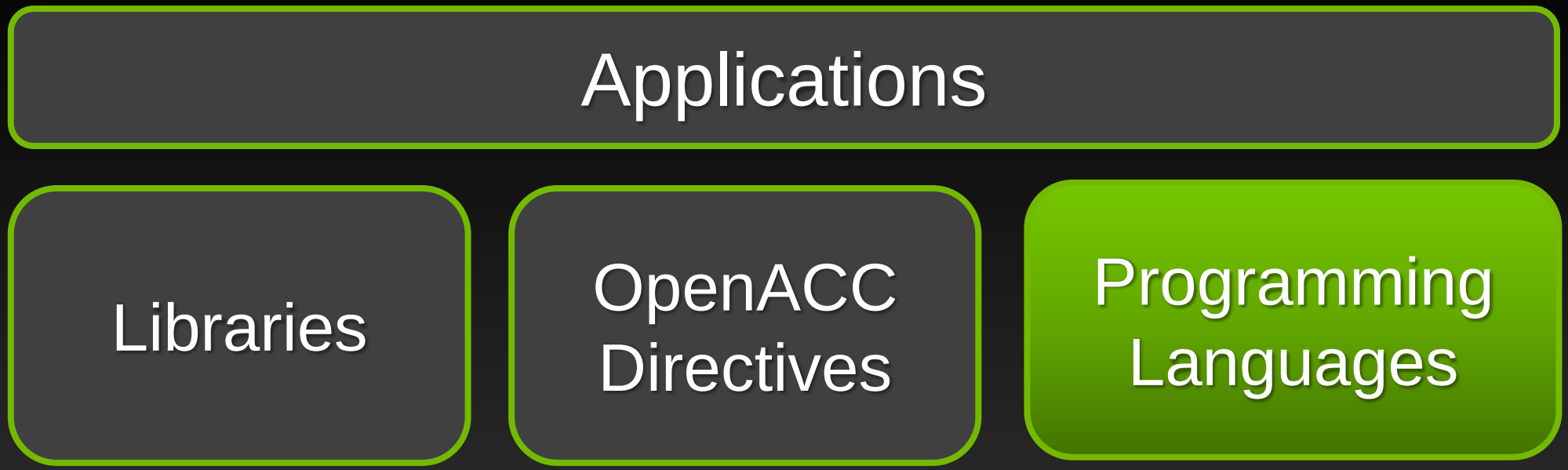
NVIDIA

Questions?



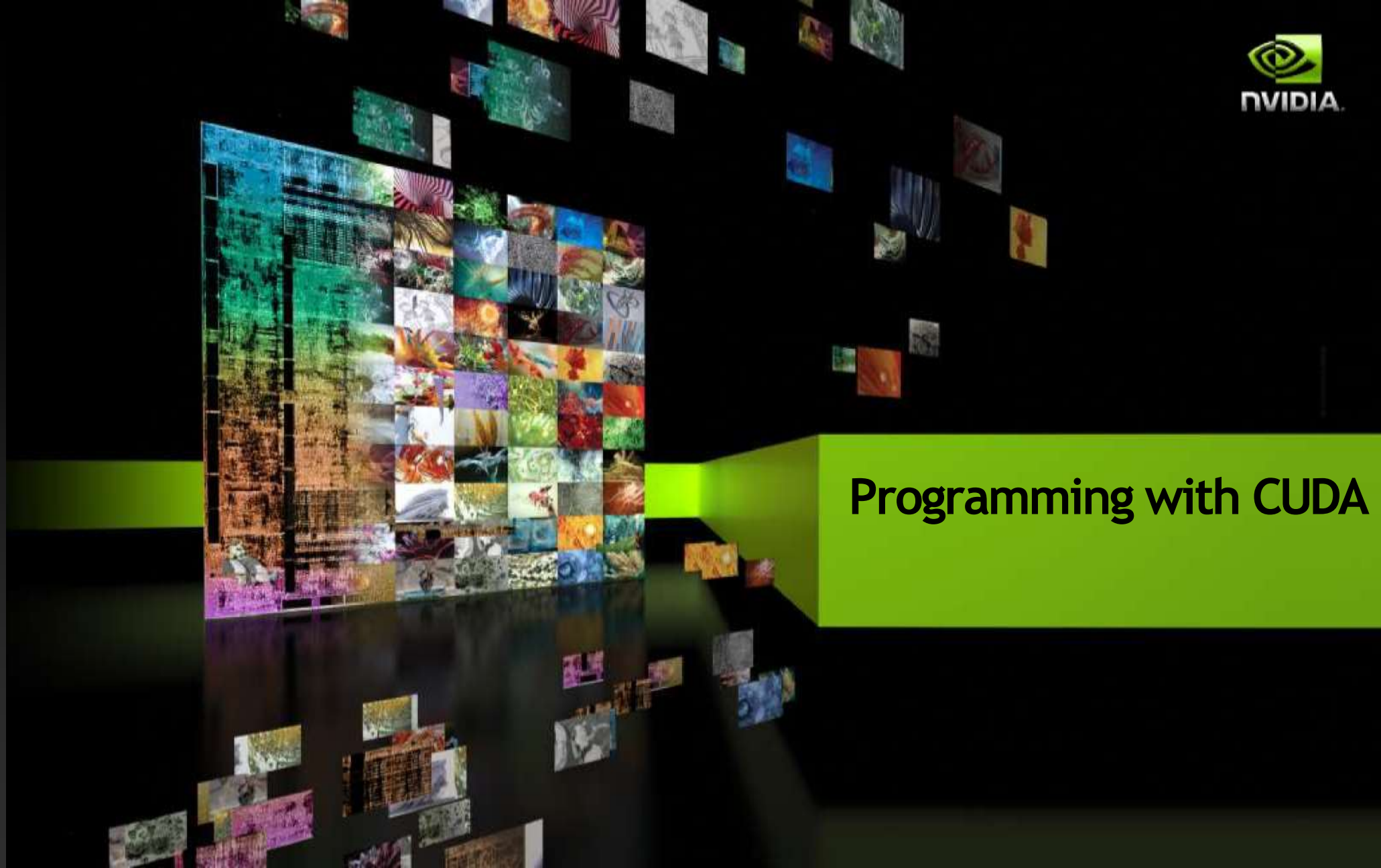
François Courteille (fcourteille@nvidia.com)
NVIDIA Corporation

Ways to Accelerate Applications



GPU Programming Languages

Numerical analytics	MATLAB, Mathematica, LabVIEW
Fortran	OpenACC, CUDA Fortran
C	OpenACC, CUDA C
C++	Thrust, CUDA C++
Python	PyCUDA, Copperhead, NumbaPro (Continuum Analytics)
C#	GPU.NET, Hybridizer(AltiMesh)


NVIDIA
Programming with CUDA

CUDA: World's Most Pervasive Parallel Programming Model

14,000 Institutions with CUDA Developers

2,000,000 CUDA Downloads

487,000,000 CUDA GPUs Shipped

700+ University Courses
In 62 Countries

CUDA for C : Standard 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

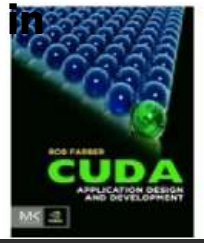
CUDA C++: Develop Generic Parallel Code

- CUDA C++ features enable sophisticated and flexible applications and middleware
  - Class hierarchies
  - __device__ methods
  - Templates
  - Operator overloading
  - Functors (function objects)
  - Device-side new/delete
  - More...

```
template <typename T>
struct Functor {
  __device__ Functor(_a) : a(_a) {}
  __device__ T operator(T x) { return
  a*x; } T a;
}

template <typename T, typename Oper>
__global__ void kernel(T *output, int
n) { Oper op(3.7);
  output = new T[n]; // dynamic allocation
  int i = blockIdx.x*blockDim.x +
  threadIdx.x; if (i < n)
    output[i] = op(i); // apply functor
}
```

If you know C++, you are already programming GPUs!

- First two examples in




```
//seqSerial.cpp
#include
<iostream>
#include <vector>
using namespace
std;


int main() {

  const int N=50000;

  // task 1: create the array
  vector<int> a(N);

  // task 2: fill the array
  for(int i=0; i < N; i++) a[i]=i;

  // task 3: calculate the sum of the array int
  sumA=0;
  for(int i=0; i < N; i++) sumA += a[i];

  // task 4: calculate the sum of 0 .. N-1
  int sumCheck=0;
   for(int i=0; i < N; i++) sumCheck += i;

  // task 5: check the results agree
  if(sumA == sumCheck) cout << "Test Succeeded!" <<
  endl; else {cerr << "Test FAILED!" << endl;
  return(1);}
 return(0);
}
```

```
//

#include<iostream>
using namespace std;
#include <thrust/reduce.h>
#include <thrust/sequence.h>
#include <thrust/host_vector.h>
#include
<thrust/device_vector.h>
int main() {

  const int N=50000;

  // task 1: create the array
  thrust::device_vector<int> a(N);

  // task 2: fill the array
  thrust::sequence(a.begin(), a.end(), 0);

  // task 3: calculate the sum of the array
  int sumA= thrust::reduce(a.begin(),a.end(), 0);

  // task 4: calculate the sum of 0 .. N-1
   int sumCheck=0;
  for(int i=0; i < N; i++) sumCheck += i;

  // task 5: check the results agree
  if(sumA == sumCheck) cout << "Test Succeeded!" <<
  endl; else { cerr << "Test FAILED!" << endl;
  return(1);}
 return(0)
};
```

Rapid Parallel C++ Development

- Resembles C++ STL
- High-level interface
  - Enhances developer productivity
  - Enables performance portability between GPUs and multicore CPUs
- Flexible
  - CUDA, OpenMP, and TBB backends
  - Extensible and customizable
  - Integrates with existing software
- Open source

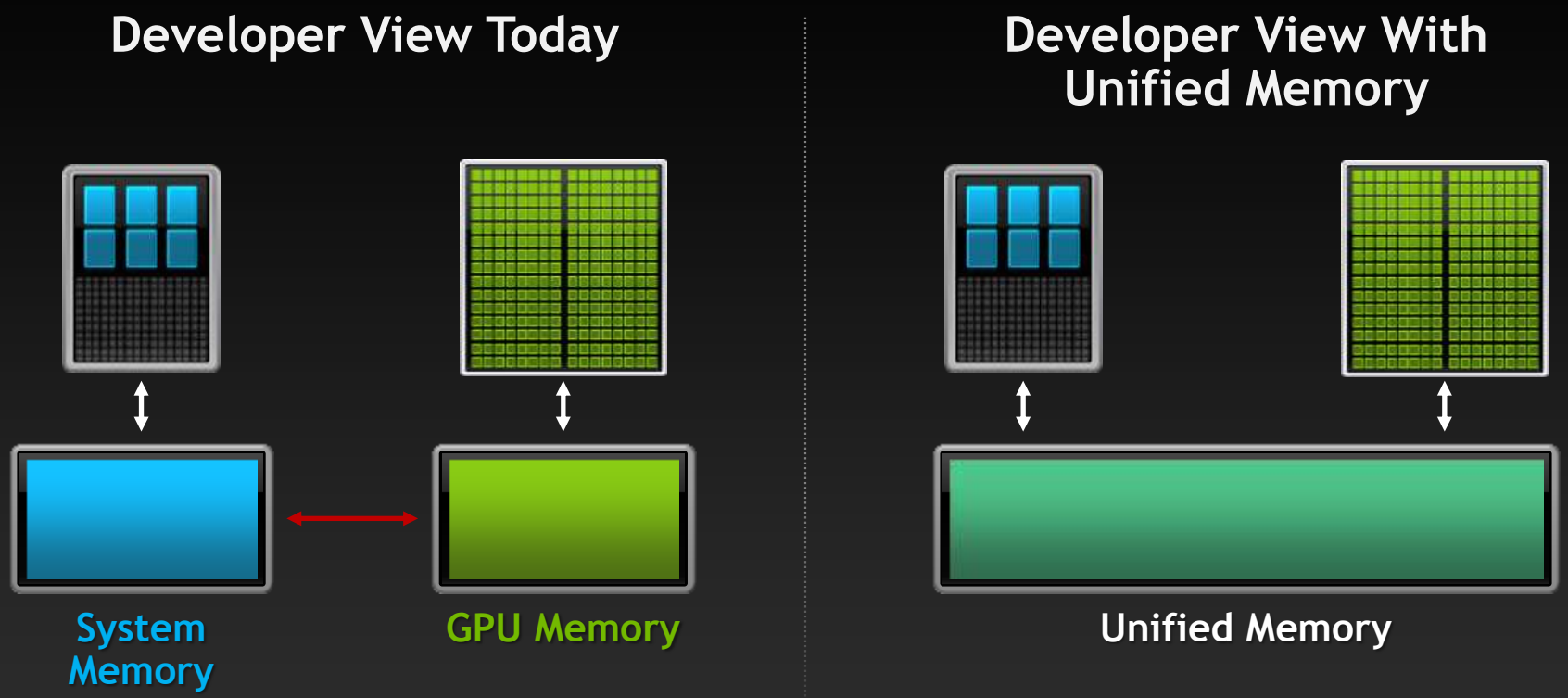

```
// generate 32M random numbers on host
thrust::host_vector<int> h_vec(32 << 20);
thrust::generate(h_vec.begin(),
                 h_vec.end(),
                 rand);

// transfer data to device (GPU)
thrust::device_vector<int> d_vec = h_vec;

// sort data on device
thrust::sort(d_vec.begin(), d_vec.end());

// transfer data back to host
thrust::copy(d_vec.begin(),
             d_vec.end(),
             h_vec.begin());
```

ANNOUNCING
CUDA 6

Dramatically Simplifies Parallel Programming with Unified Memory

Unified Memory

Dramatically Lower Developer Effort



What Is Unified Memory?

1. Simpler Programming & Memory Model
   - Single pointer to data, accessible anywhere
   - Eliminate need for *cudaMemcpy()*
   - Greatly simplifies code porting

2. Performance Through Data Locality
   - Migrate data to accessing processor
   - Guarantee global coherency
   - Still allows *cudaMemcpyAsync()* hand tuning

Just Three Additions To CUDA

New API: *cudaMallocManaged()*

- Drop-in replacement for cudaMalloc() allocates managed memory
- Returns pointer accessible from both Host and Device

New API: *cudaStreamAttachMemAsync()*

- Manages concurrency in multi-threaded CPU applications

New keyword: *__managed__*

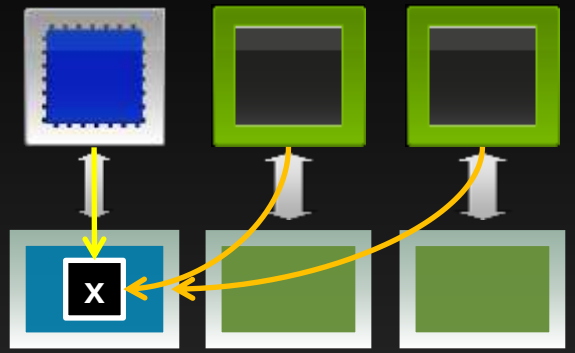
- Global variable annotation combines with __device__
- Declares global-scope migratable device variable
- Symbol accessible from both GPU and CPU code

System Requirements

GPU	Kepler & Maxwell	(GK10x+ or GM10x+, i.e. SM3.0+)
Operating System	64-bit required	(Android allows 32-bit)
Linux	Kernel 2.6.18+	(all CUDA-supported distros, not ARM)
Windows	Win7 or Win8	(WDDM & TCC no XP/Vista)
Android	Logan, iGPU	(with r19 driver, no discrete GPU)
Mac OSX	Not supported in CUDA 6.0	
Linux on ARM	Not supported until ARM64	

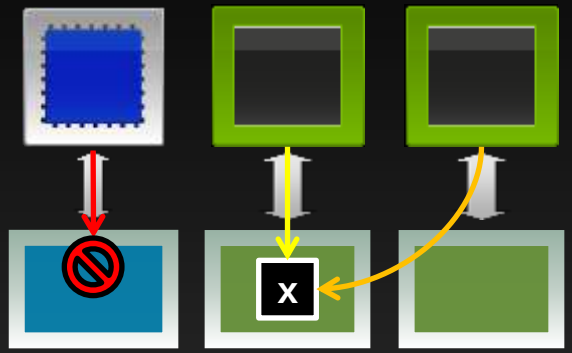
CUDA Memory Types

Zero-Copy
`cudaMallocHost(&x, 4)`



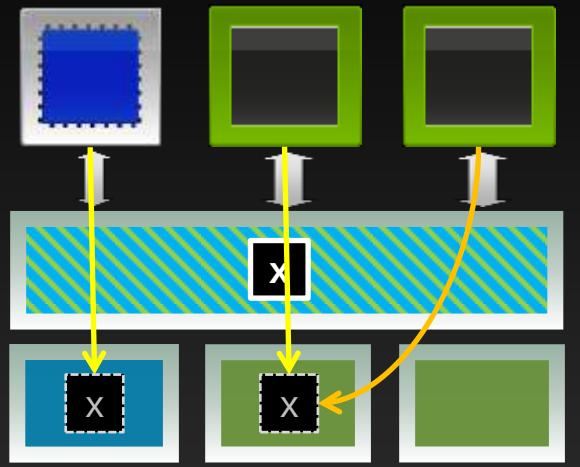
- Allocation fixed in CPU mem
- PCIe access for all GPUs
- Local access for CPU

Unified Virtual Addressing
`cudaMalloc(&x, 4)`



- Allocation fixed in GPU mem
- Local access for home GPU
- No CPU access
- PCIe access for other GPUs

Unified Memory (6.0)
`cudaMallocManaged(&x, 4)`



- On-access CPU/GPU migration
- Local access for home GPU
- Local access for CPU
- PCIe access for other GPUs

CUDA 5 without Unified Memory

Memory Management is Functional Requirement

Call Sort on CPU

```
void sortfile(FILE *in, FILE *out, int N)
{
    char *data = (char *)malloc(N);
    fread(data, 1, N, in);

    sort(data, N);

    fwrite(data, 1, N, out);
    free(data);
}
```

Call Sort on GPU

```
void sortfile(FILE *in, FILE *out, int N)
{
    char *data = (char *)malloc(N);
    fread(data, 1, N, in);

    char *gpu_data;
    cudaMalloc(&gpu_data, N);
    cudaMemcpy(gpu_data, data, N, ...);

    parallel_sort<<< ... >>>(data, N);
    cudaDeviceSynchronize();

    cudaMemcpy(data, gpu_data, N, ...);
    fwrite(data, 1, N, out););
    free(data);
    cudaFree(gpu_data);
}
```

CUDA 6 with Unified Memory on Kepler

Memory Management Becomes Performance Optimization

Call Sort on CPU

```
void sortfile(FILE *in, FILE *out, int N)
{
    char *data = (char *)malloc(N);
    fread(data, 1, N, in);

    sort(data, N);

    fwrite(data, 1, N, out);
    free(data);
}
```

Call Sort on GPU with UM - Kepler

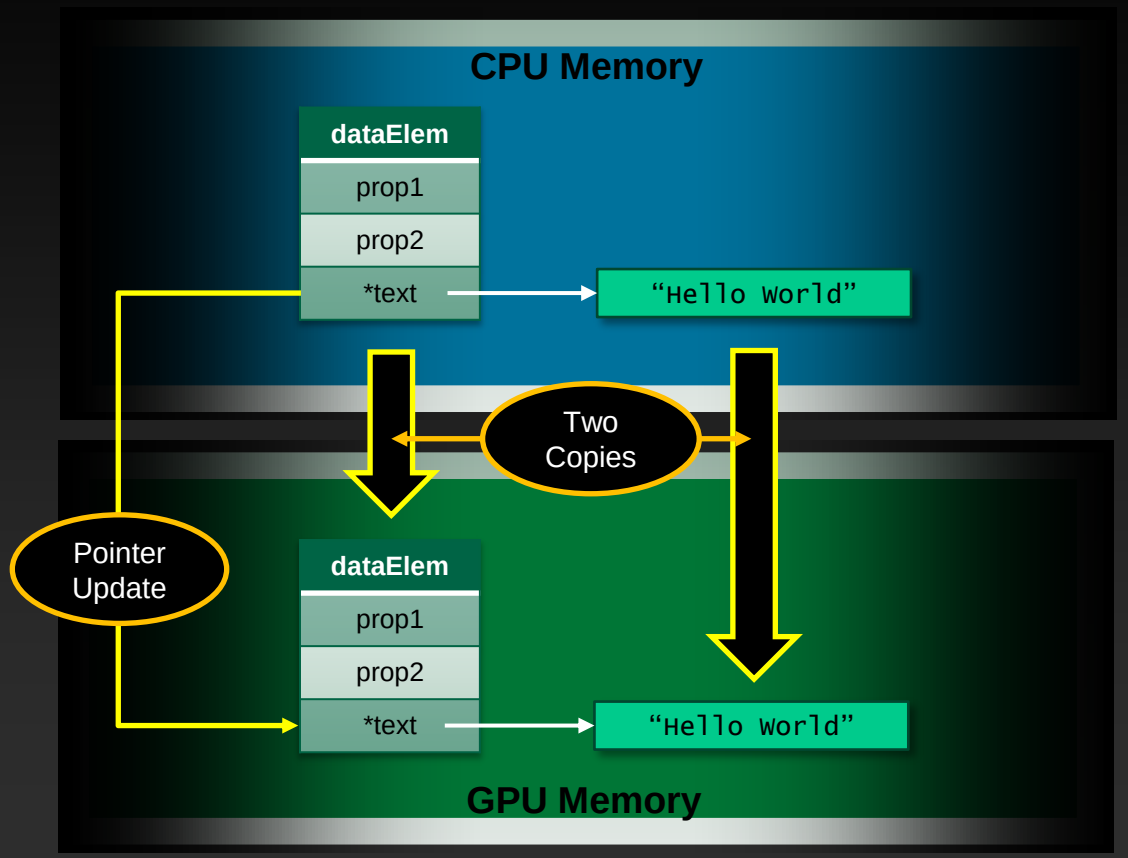
```
void sortfile(FILE *in, FILE *out, int N)
{
    char *data = (char *)cudaMallocManaged(N);
    fread(data, 1, N, in);

    parallel_sort<<< ... >>>(data, N);

    fwrite(data, 1, N, out);
    cudaFree(gpu_data);
}
```

Specially-allocated memory automatically migrated to GPU before kernels if touched by CPU. CPU can migrate data over on access.

Simpler Memory Model

- Eliminate Deep Copying

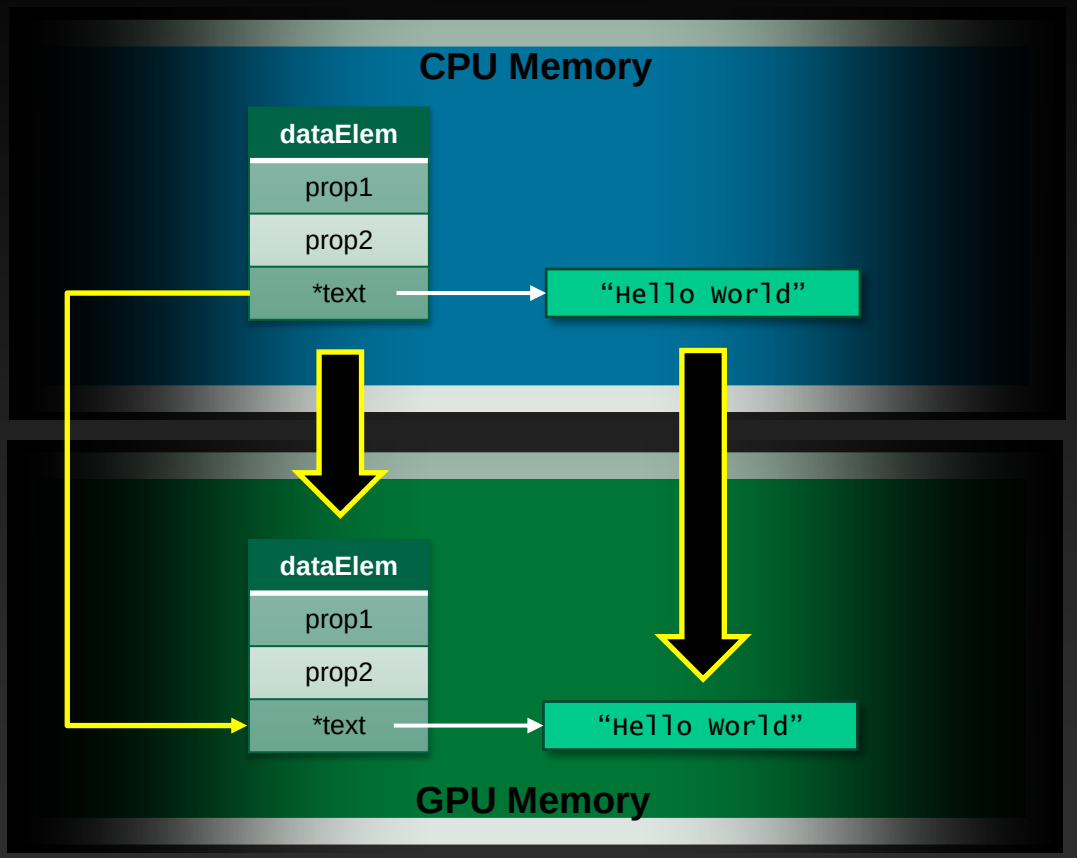
```
struct dataElem {
    int prop1;
    int prop2;
    char *text;
};
```



Simpler Memory Model

- Eliminate Deep Copying

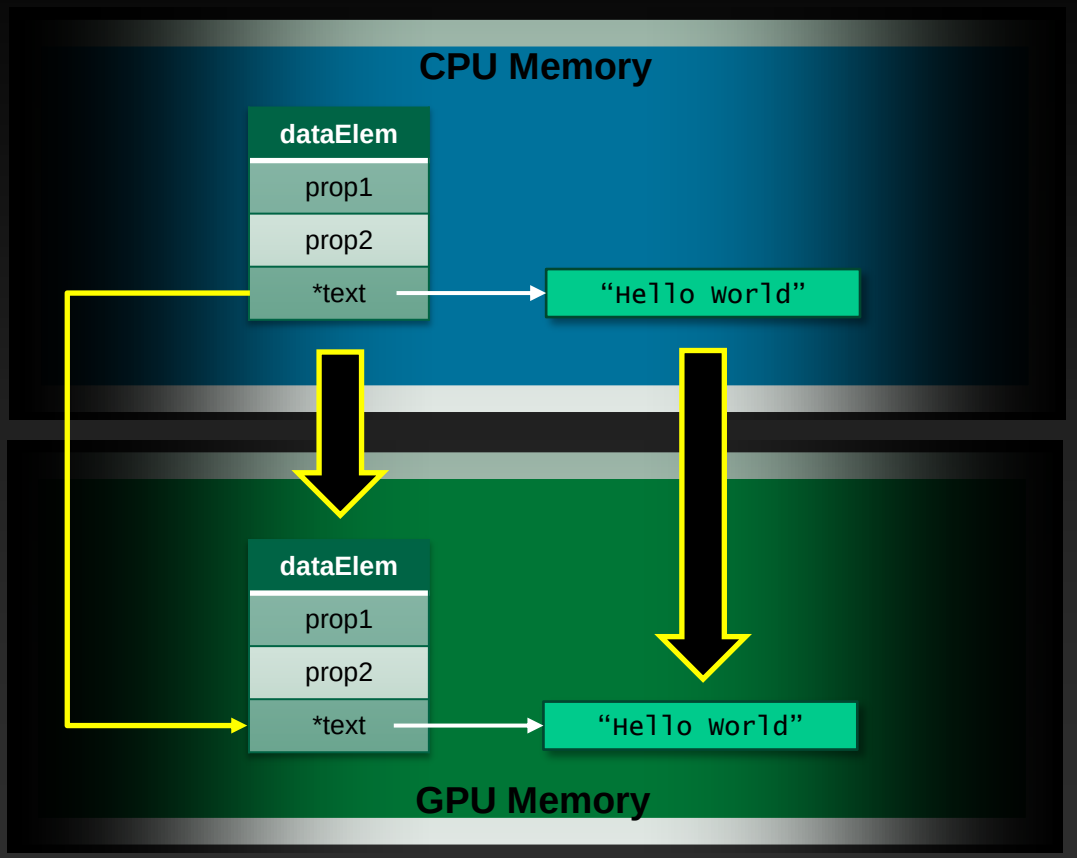
```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new "text" pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text,
                                sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU & GPU use different copies of "elem"
    kernel<<< ... >>>(g_elem);
}
```



Simpler Memory Model

- Eliminate Deep Copying

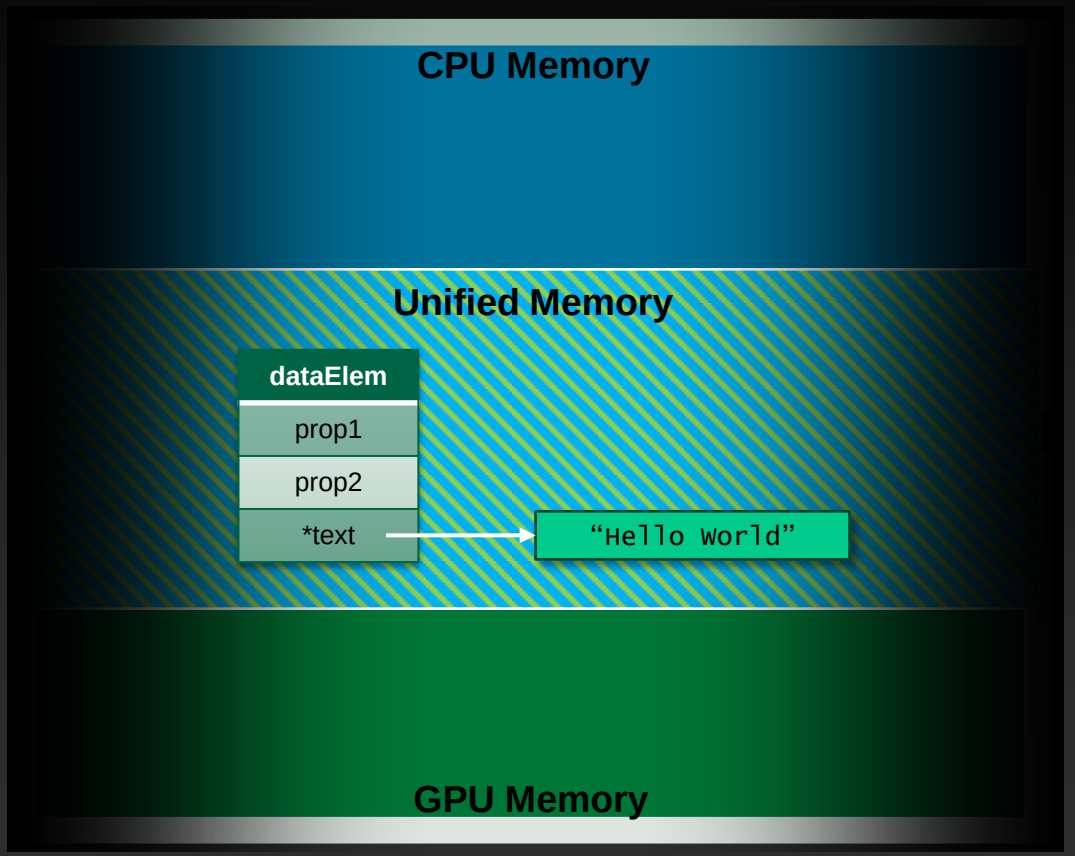
```
void launch(dataElem *elem) {
    dataElem *g_elem;
    char *g_text;

    int textlen = strlen(elem->text);

    // Allocate storage for struct and text
    cudaMalloc(&g_elem, sizeof(dataElem));
    cudaMalloc(&g_text, textlen);

    // Copy up each piece separately, including
    // new "text" pointer value
    cudaMemcpy(g_elem, elem, sizeof(dataElem));
    cudaMemcpy(g_text, elem->text, textlen);
    cudaMemcpy(&(g_elem->text), &g_text,
                                sizeof(g_text));

    // Finally we can launch our kernel, but
    // CPU & GPU use different copies of "elem"
    kernel<<< ... >>>(g_elem);
}
```



Simpler Memory Model

- Eliminate Deep Copying

```
void launch(dataElem *elem) {
    kernel<<< ... >>>(elem);
}
```



Unified Memory with C++

C++ objects migrate easily when allocated on managed heap

- Overload *new* operator* to use C++ in unified memory region

```
class Managed {
    void *operator new(size_t len) {
        void *ptr;
        cudaMallocManaged(&ptr, len);
        return ptr;
    }

    void operator delete(void *ptr) {
        cudaFree(ptr);
    }
};
```

NOTE: Unified Memory can only manage objects in heap - stack objects are unmanaged

NOTE: cudaMallocManaged() not supported in device code for CUDA 6.0

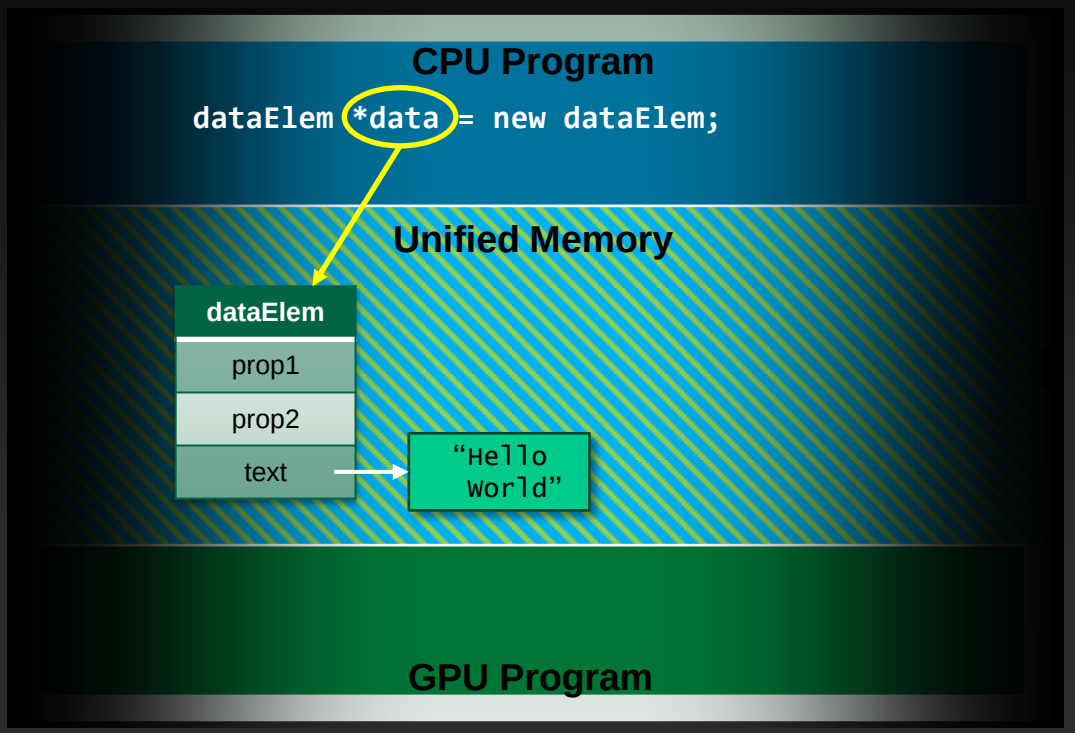
* (or use placement-new)

Unified Memory with C++

Combination of C++ and Unified Memory is very powerful

- Concise and explicit: let C++ handle deep copies
- Pass by-value or by-reference without memcpy shenanigans

```
// Note "managed" on this class, too.
// C++ now handles our deep copies
class dataElem : public Managed {
            int prop1;
            int prop2;
            String text;
};
```

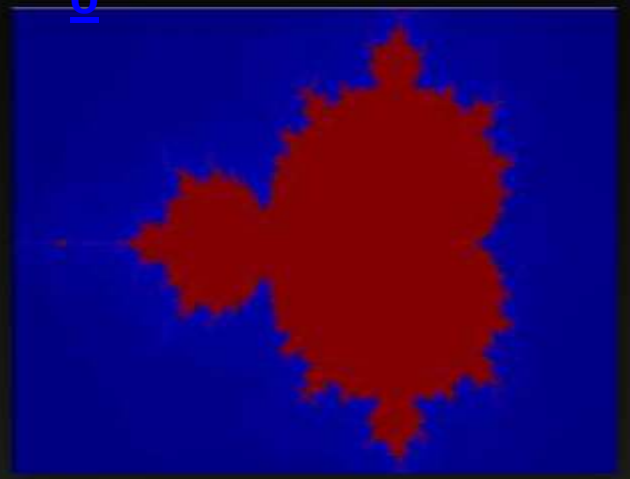


CUDA Fortran with Managed Data

```
program pgi14x
  use cudafor
  integer, parameter :: n = 100000
  integer, managed :: a(n), b(n), c(n)
  a = [(i,i=1,n)]
  b = 1
!$cuf kernel do <<< *, * >>>
  do i = 1, n
    c(i) = a(i) + b(i)
  end do
  if (sum(c).ne.n*(n+1)/2+n) then
    print *,c(1),c(n)
  end if
end
```

CUDA Python with NumbaPro

- http://continuum.io

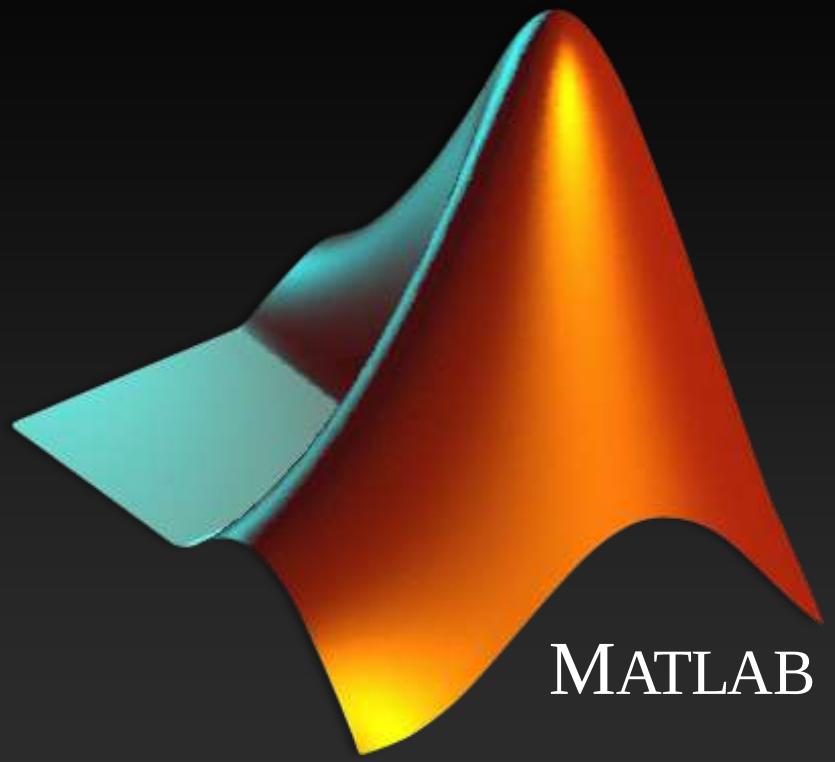


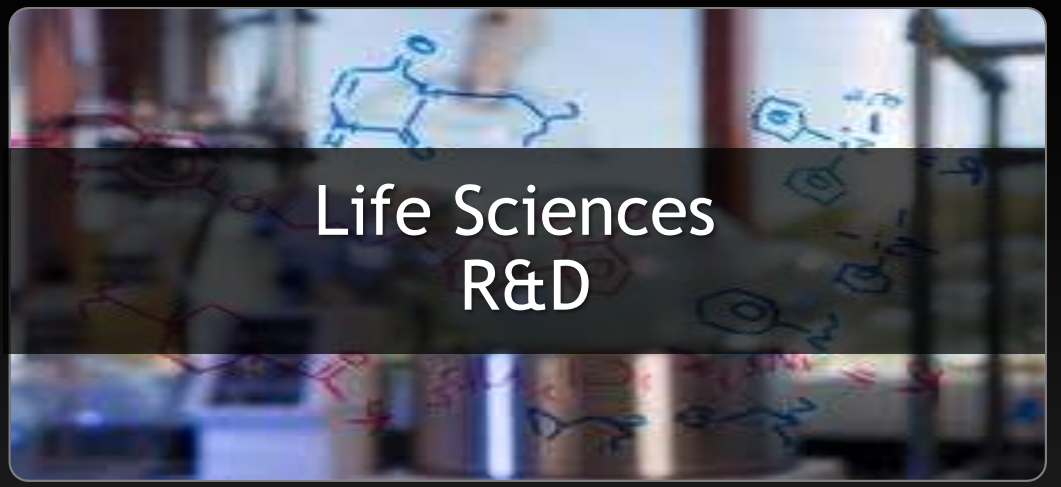
```
@cuda.jit(restype=uint8, argtypes=[f8, f8, uint32], device=True)
def mandel(x, y, max_iters):
zr, zi = 0.0, 0.0
  for i in range(max_iters):
  newzr = (zr*zr-zi*zi)+x zi
  = 2*zr*zi+y
    zr = newzr
    if (zr*zr+zi*zi) >= 4:
    return i
  return 255


@cuda.jit(argtypes=[uint8[:,:], f8, f8, f8, f8, uint32])
def mandel_kernel(img, xmin, xmax ymin, ymax, iters):
  x, y = cuda.grid(2)
  if x < img.shape[0] and y < img.shape[1]:
    img[y, x] = mandel(min_x+x*((max_x-min_x)/img.shape[0]),
    min_y+y*((max_y-min_y)/img.shape[1]), iters)

gimage = np.zeros((1024, 1024), dtype = np.uint8)
d_image = cuda.to_device(gimage)
mandel_kernel[(32,32), (32,32)](d_image, -2.0, 1.0, -1.0, 1.0, 20)
d_image.to_ host(
)
```

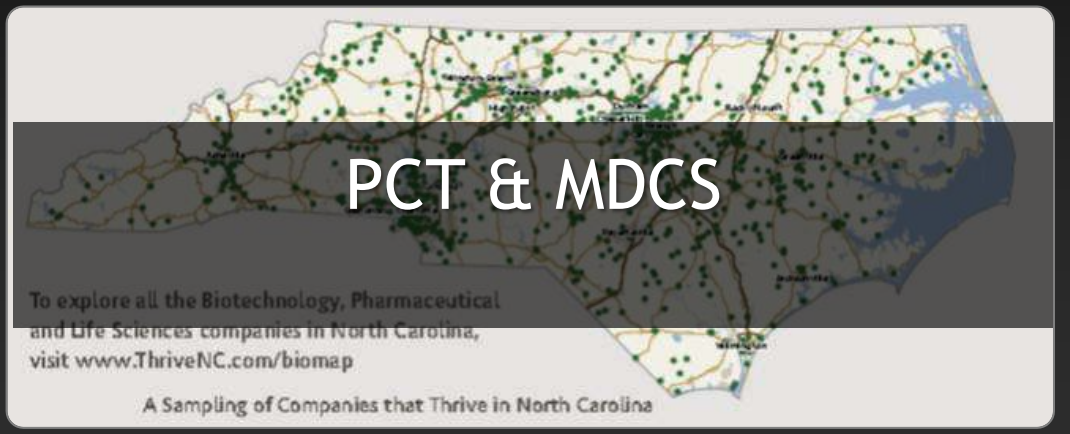
CUDA Programming, Python Syntax

	1024^2 Mandelbrot Time	Speedup v. Pure Python
Pure Python	4.85s	--
NumbaPro(CPU)	0.11s	44x
CUDA Python (K20)	.	

MATLAB

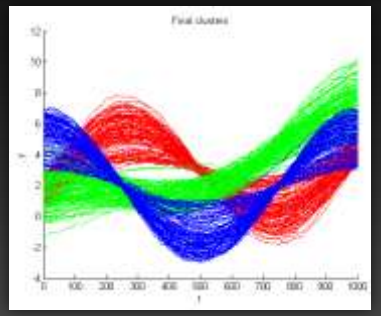

MATLAB


Life Sciences
R&D

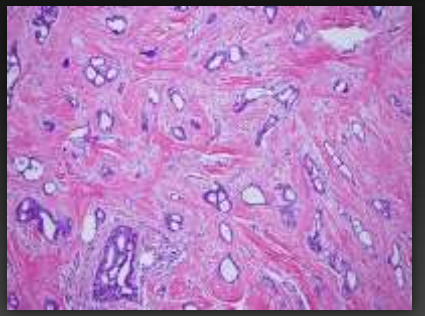

PCT & MDCS
To explore all the Biotechnology, Pharmaceutical
and Life Sciences companies in North Carolina,
visit www.ThriveNC.com/biomap
A Sampling of Companies that Thrive in North Carolina

GPU-Accelerated MATLAB Results

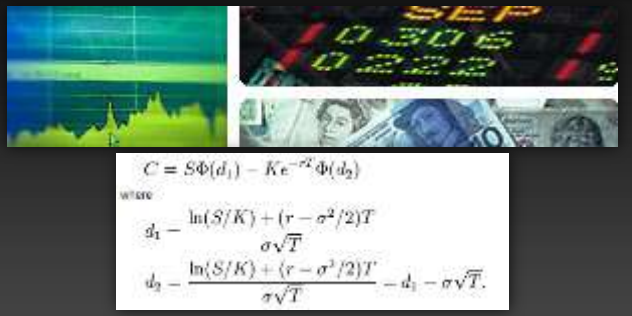
MathWorks



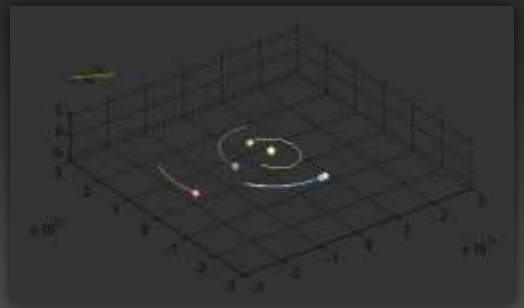
10x speedup in data clustering via K-means clustering algorithm



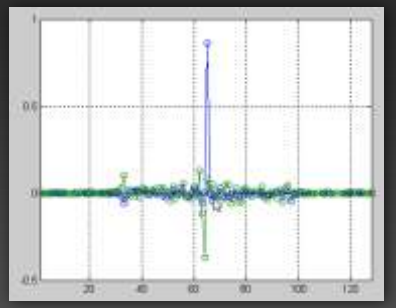
14x speedup in template matching routine (part of cancer cell image analysis)



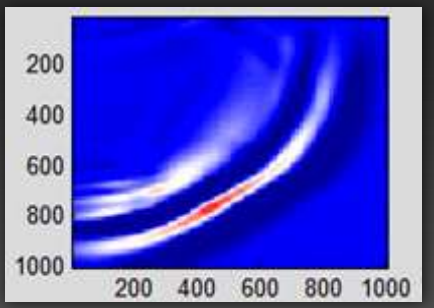
3x speedup in estimating 7.6 million contract prices using Black-Scholes model



17x speedup in simulating the movement of 3072 celestial objects

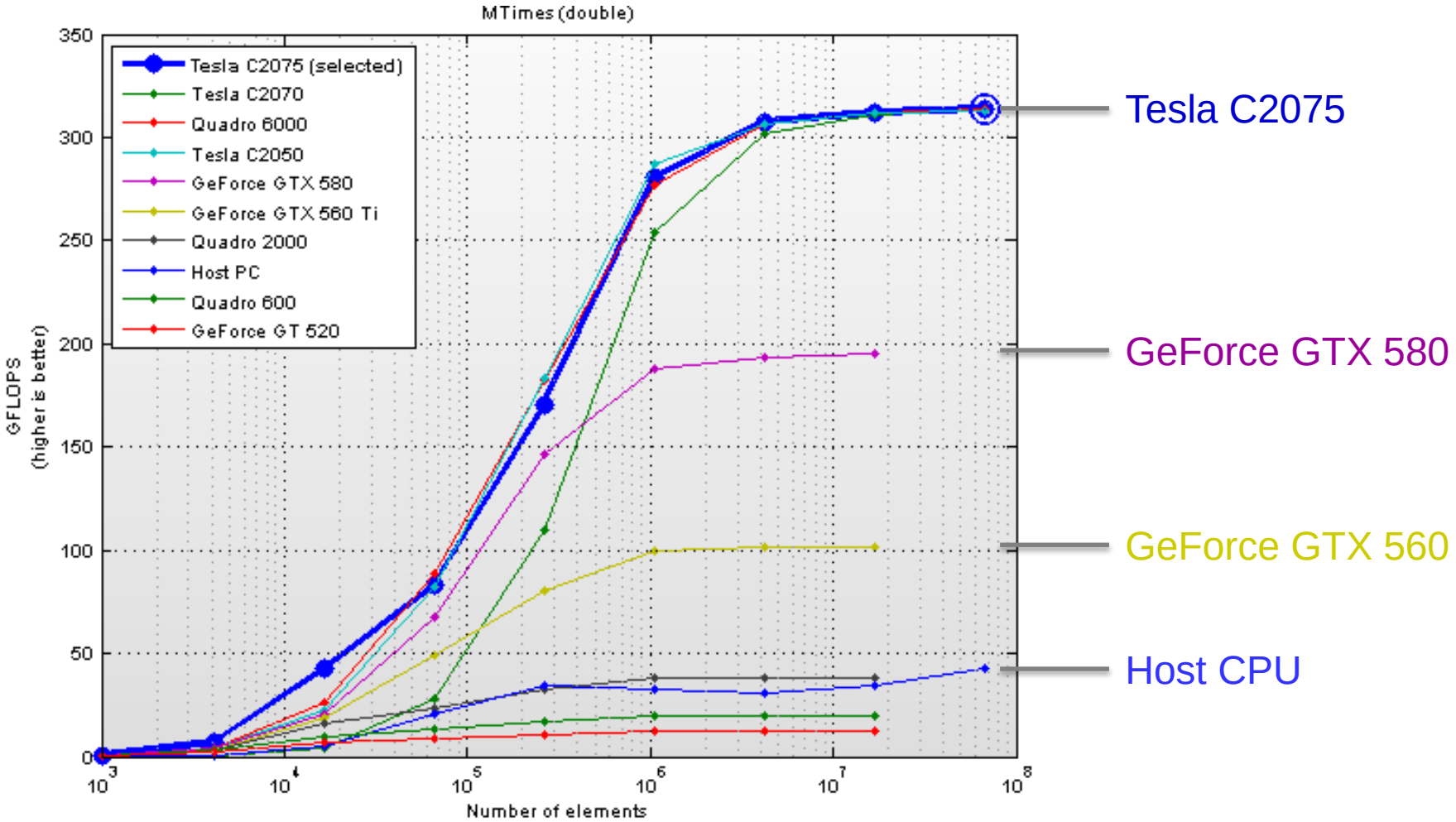


4x speedup in adaptive filtering routine (part of acoustic tracking algorithm)



4x speedup in wave equation solving (part of seismic data processing algorithm)

GPU Value in MATLAB - Bigger is Better



Double precision MTimes performance as measured by GPUBench – available from MATLAB Central File Exchange

Get Started with GPU Programming

Watch

bit.ly/GPUGetStarted

Explore

NVIDIA. DEVELOPER ZONE

- Get CUDA
- Access Tools
- Learn with Tutorials
- Join the Community

developer.nvidia.com/get-started-parallel-computing

Get Started Today

These languages are supported on all CUDA-capable GPUs.

You might already have a CUDA-capable GPU in your laptop or desktop PC!

CUDA C/C++
http://developer.nvidia.com/cuda-toolkit

Thrust C++ Template Library
http://developer.nvidia.com/thrust

CUDA Fortran
http://developer.nvidia.com/cuda-toolkit

PyCUDA (Python)
http://mathema.tician.de/software/pycuda

GPU.NET
http://tidepowerd.com

MATLAB
http://www.mathworks.com/discovery/matlab-gpu.html

Mathematica
http://www.wolfram.com/mathematica/new-in-8/cuda-and-opencl-support/

Easiest Way to Learn CUDA

50k Enrolled

127 Countries

coursera

Heterogeneous Parallel Programming
www.coursera.org

UDACITY

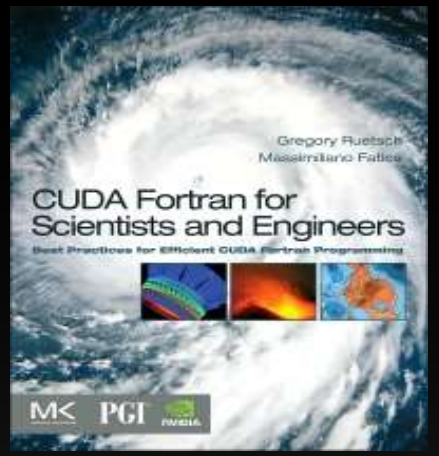
Introduction to Parallel Programming
www.udacity.com

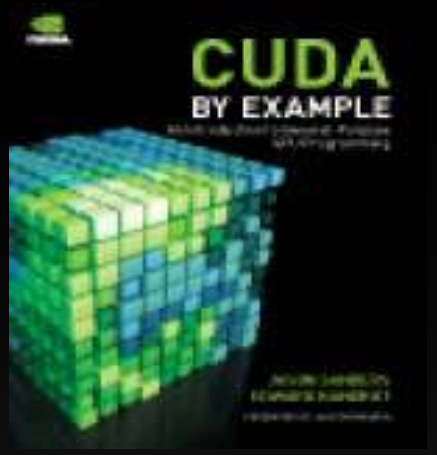
Learn from the Best
- Prof. John Owens – UC Davis
- Dr. David Luebke – NVIDIA Research
- Prof. Wen-mei W. Hwu – U of Illinois

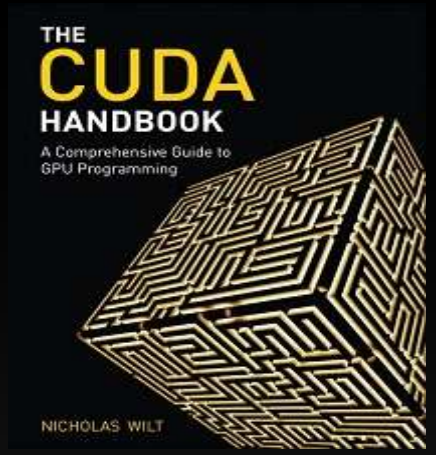
Anywhere, Any Time
- Online
- Worldwide
- Self Paced

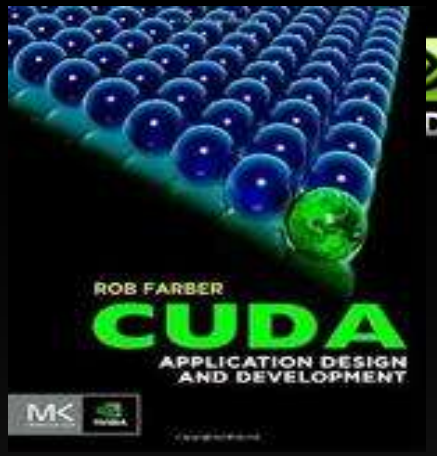
It's Free!
- No Tuition
- No Hardware
- No Books

Engage with an Active Community
- Forums and Meetups
- Hands-on Projects





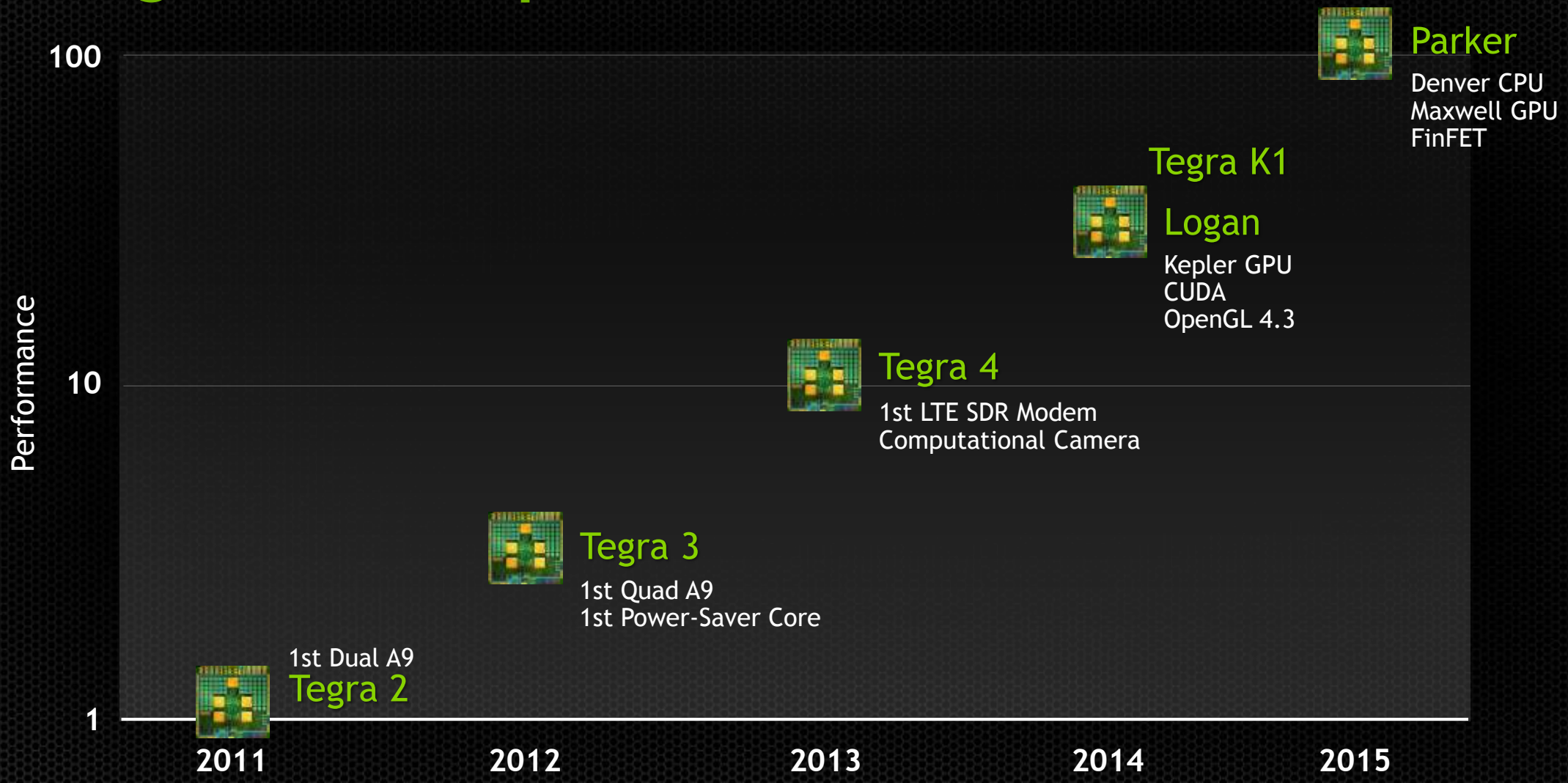




[CUDACasts]
[Parallelforall blog]
[New Parallel Forall blog]
[GTC Express Archives]
[Udacity]
[Coursera]
[CUDA Docs]
[CUDA Spotlights]
[CUDA Newsletter]

New self-paced learning pages!
https://developer.nvidia.com/how-to-numerical-analysis
https://developer.nvidia.com/how-to-cuda-python
https://developer.nvidia.com/how-to-cuda-libraries
https://developer.nvidia.com/how-to-openacc
https://developer.nvidia.com/how-to-cuda-c-cpp

Tegra Roadmap



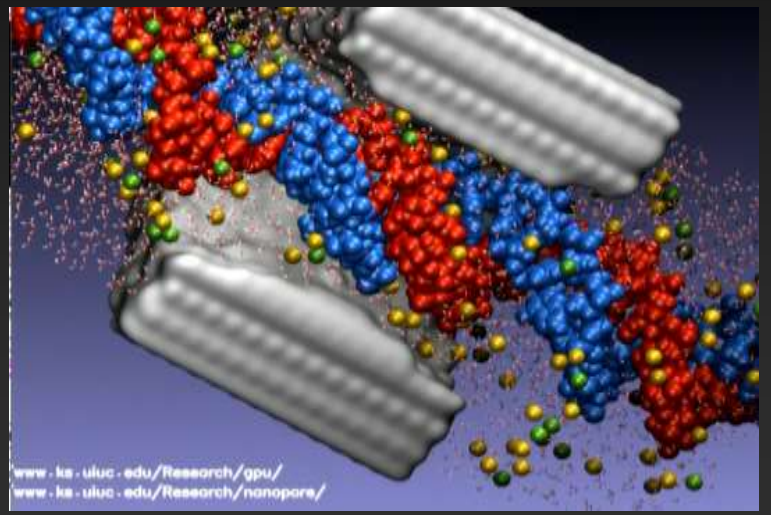
KAYLA

CUDA Fully Supported on ARM

CUDA 5 | OpenGL 4.3

Kickstarts ARM + CUDA Eco-system

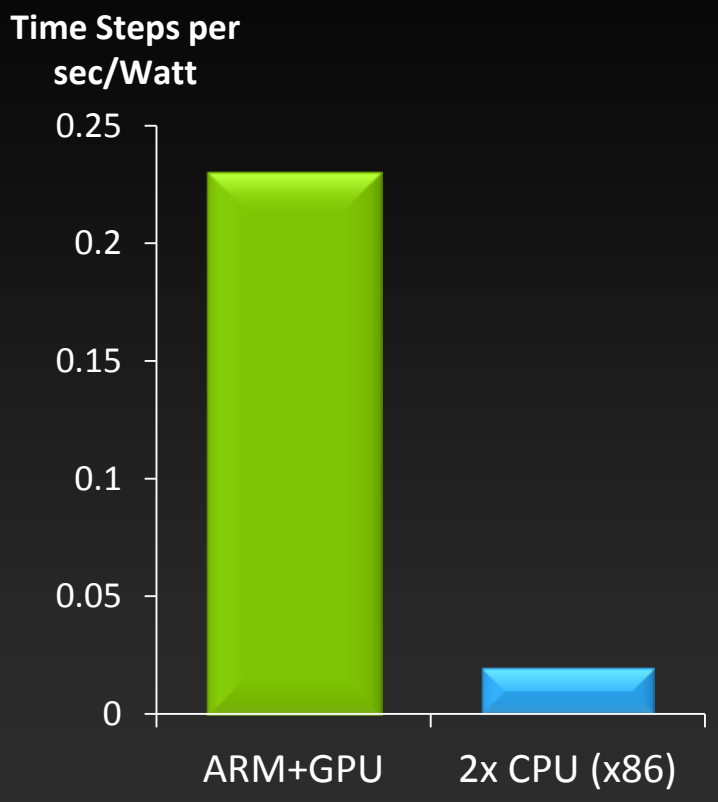
NAMD Ported in 2 Days

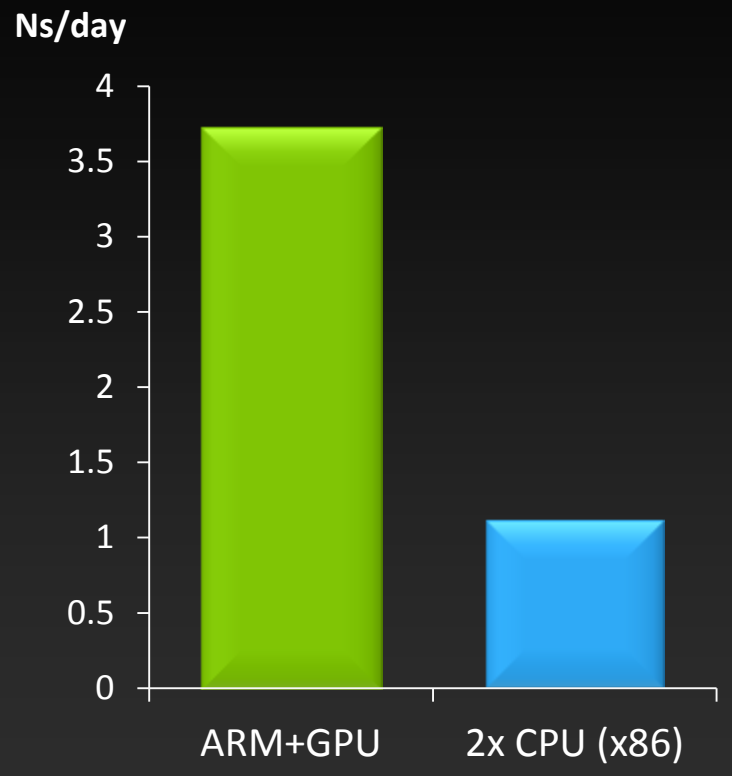


Quad ARM + Small CUDA GPU

Quad ARM + Any CUDA GPU

HPC Ecosystem Getting Ready for ARM



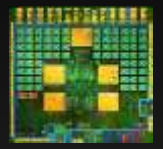
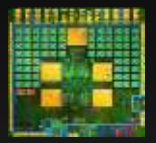


Applications Porting to ARM + GPU

- ACEMD
- AMBER
- GROMACS
- HOOMD-Blue
- NAMD
- NWChem
- OpenMM

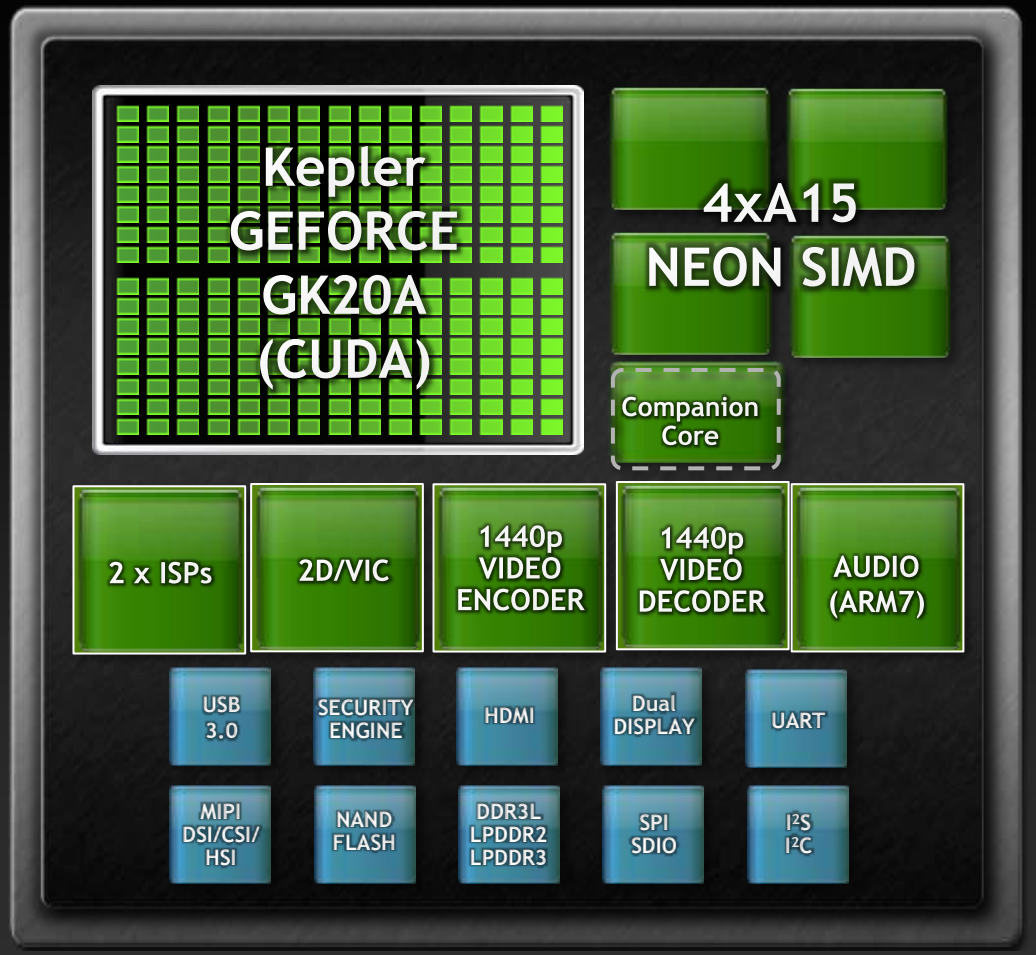
HOOMD-blue (LJ-liquid (N=64000), GeForce GT 640 + Tegra3 vs. 2x E5-2667 (12 Sandy Bridge cores)
AMBER (Cellulose NPT (408,609 atoms), GeForce GTX 680 + Tegra3 vs. 2x E5-2670 (16 Sandy Bridge cores)

The Evolution of Heterogeneous Solutions

2013	Q1 2014	QX 2014
Tegra4	Tegra K1 (Logan)	Logan-64
		
Traditional GPU cores	192 x DX11 CUDA cores	192 x DX11 CUDA cores
4+1 ARM A15 cores	4+1 ARM A15 cores	NVIDIA Denver 64b ARM cores

http://www.anandtech.com/show/7622/nvidia-tegra-k1

Logan (T124)



- SM 3.2 GPU Kepler architecture
- Full CUDA-C implementation
  - Fast flexible parallel programming
  - No dynamic parallelism nor Hyper-Q
- Truly heterogeneous programming with Unified memory. Shares memory accesses between ARM (NEON) and GPU (CUDA); UVM-Lite

X-Gene™ XC-1 Evaluation Board

Specifications

- **Mini-ITX form factor with power supply**
- **1 STORM CPU – 8 core 2.4 GHz**
- **2 Memory Channels – DDR3-1600; 16GB**
- **1x PCIe Gen3 x8**
- **4x SATA 3.0**
- **2x USB 3.0**
- **Serial Port: 1 DB-9**
- **Network**
  - 1 x 10GbE SFP+
  - 2 x 1GbE SGMII RJ-45, 1 x 1GbE RGMII RJ-45
- **ARM Debug Headers**
- **SD-Card: boot device**
- **SPI NOR Flash: BIOS/UEFI**