Технології графічного процесінгу

(Масивно-паралельні обчислення на графічних прискорювачах
…
Massively Parallel Computing on Graphic Processing Units - GPUs)

Lecture 7. CUDA Specialized Libraries and Development Tools

Yuri G. Gordienko

(NTUU-KPI, 2021)

(on the basis of materials by NVIDIA, W.Hwu, D.Kirk, S.Tomow, A.Klockner)

From the previous lecture:
Parallel Patterns

- Think at a higher level than individual CUDA kernels

- Specify **what** to compute, not **how** to compute it

- Let programmer worry about algorithm

- Defer pattern implementation to someone else

From the previous lecture:
Parallel Computing Scenarios

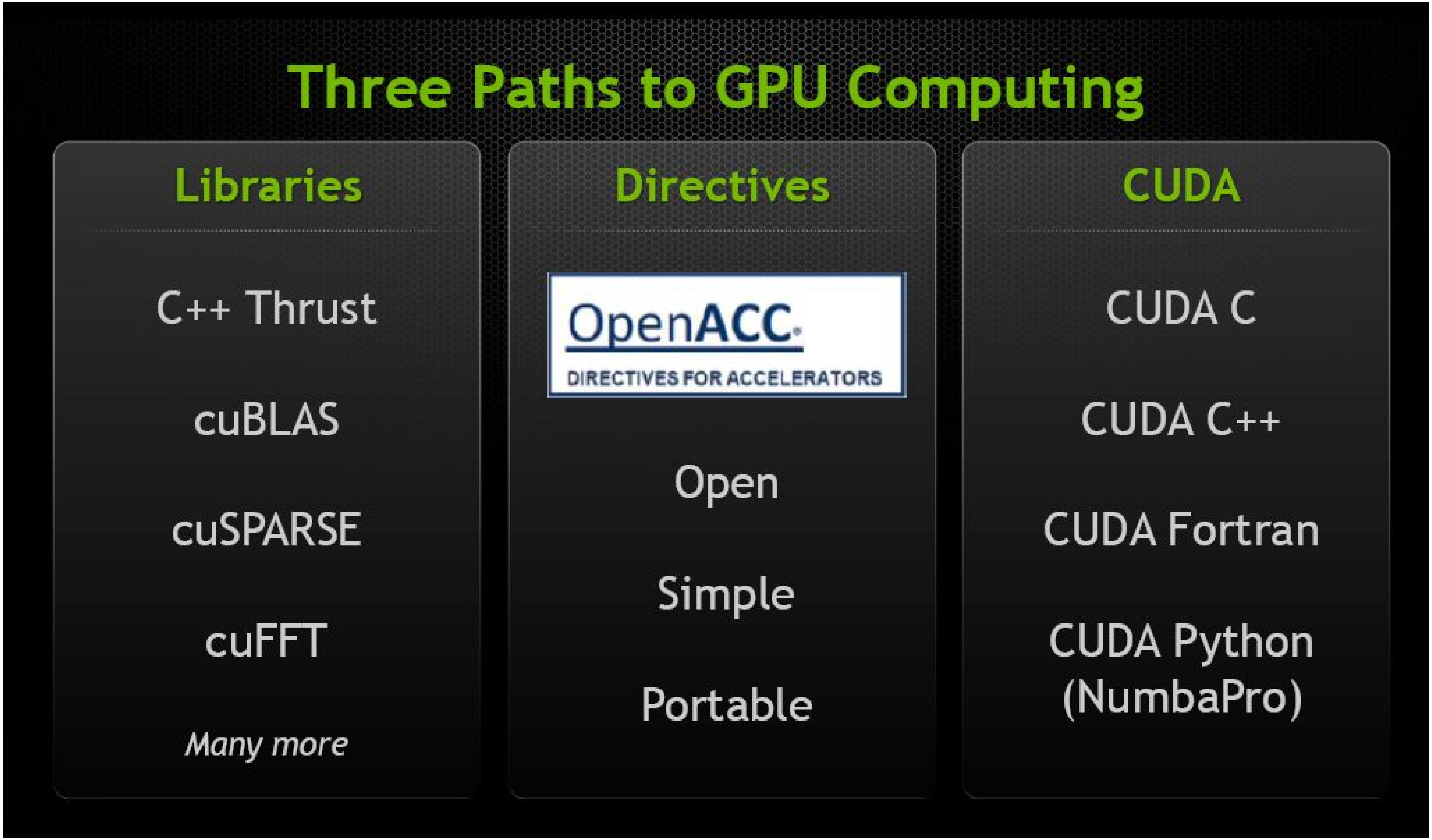
- Many parallel threads need to generate a single result
  - **Reduce**

- Many parallel threads need to partition data
  - **Split**

- Many parallel threads produce variable output / thread
  - **Compact / Expand**

From the previous lecture:

Current trends in GPU programming



Three Paths to GPU Computing

Libraries

C++ Thrust

cuBLAS

cuSPARSE

cuFFT

Many more

Directives

OpenACC

DIRECTIVES FOR ACCELERATORS

Open

Simple

Portable

CUDA

CUDA C

CUDA C++

CUDA Fortran

CUDA Python (NumbaPro)

Parallel Computing Algorithms: CUDA Libraries –> **Thrust**

What is Thrust?

- High-Level Parallel Algorithms Library
- Parallel Analog of the C++ Standard Template Library (STL)
- Performance-Portable Abstraction Layer
- Productive way to program CUDA

Code Example: Magically Simple!

```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>

int main(void)
{
    // generate 16M random numbers on the host
    thrust::host_vector<int> h_vec(1 << 24);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);
    // transfer data to the device
    thrust::device_vector<int> d_vec = h_vec;
    // sort data on the 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());
    return 0;
}
```

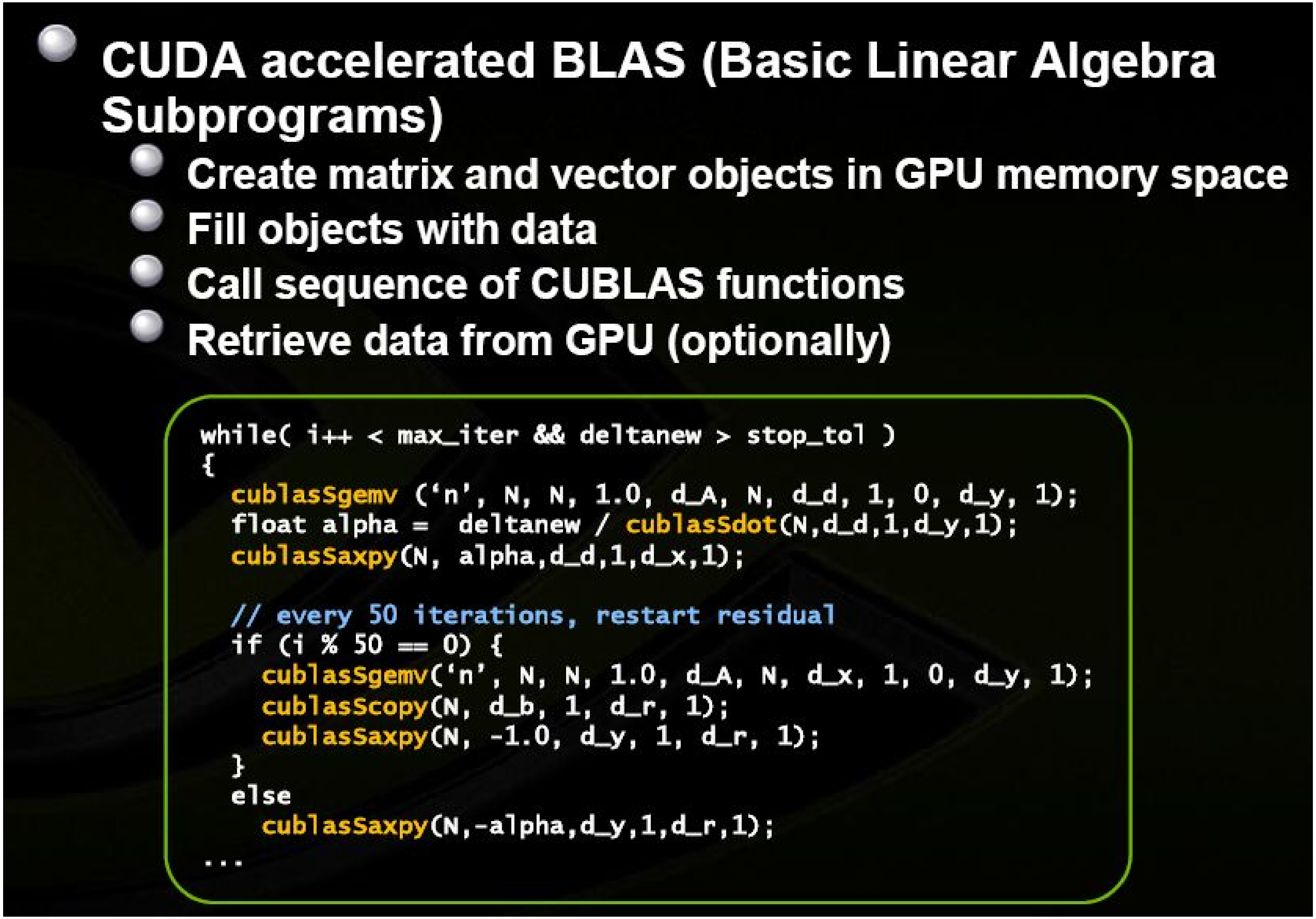
Other
CUDA Specialized Libraries

CUDA Specialized Libraries: CUBLAS

CUDA Specialized Libraries: CUBLAS

- **Cuda Based Linear Algebra Subroutines**
- SAXPY, conjugate gradient, linear solvers.
- 3D reconstruction of planetary nebulae example

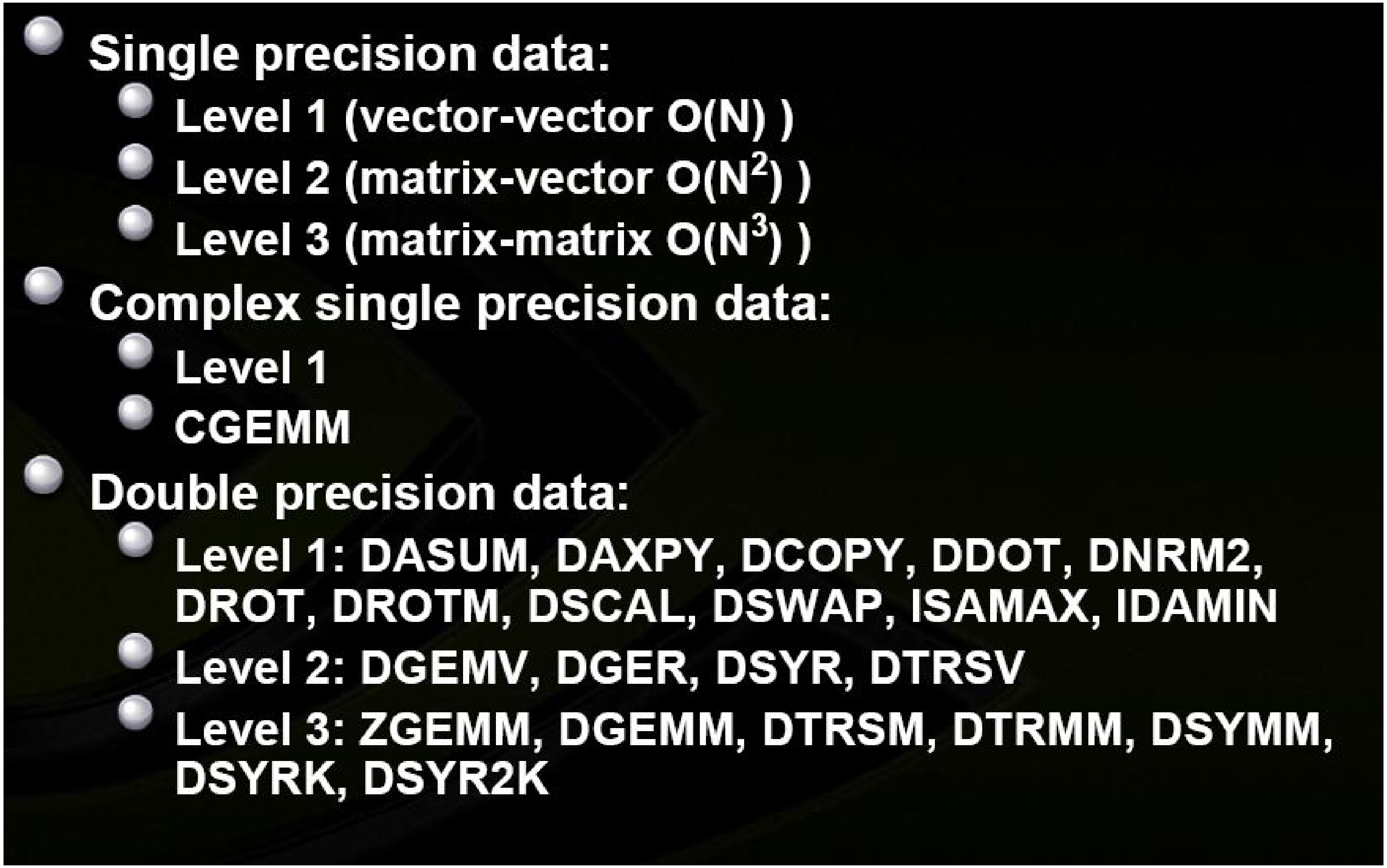
CUBLAS



- CUDA accelerated BLAS (Basic Linear Algebra Subprograms)
  - Create matrix and vector objects in GPU memory space
  - Fill objects with data
  - Call sequence of CUBLAS functions
  - Retrieve data from GPU (optionally)

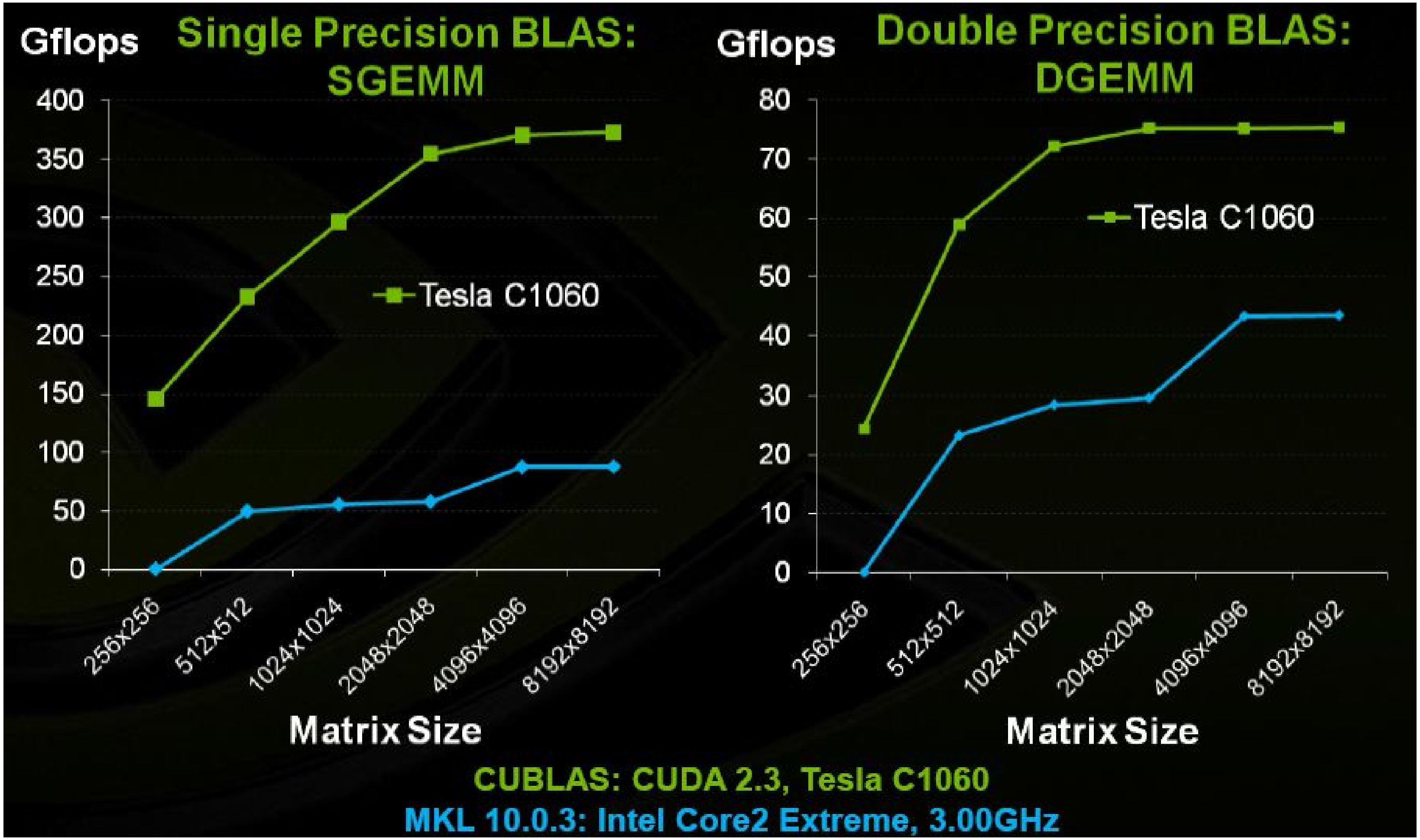
```
while( i++ < max_iter && deltanew > stop_tol )
{
  cublasSgemv (‘n’, N, N, 1.0, d_A, N, d_d, 1, 0, d_y, 1);
  float alpha =  deltanew / cublasSdot(N,d_d,1,d_y,1);
  cublasSaxpy(N, alpha,d_d,1,d_x,1);

  // every 50 iterations, restart residual
  if (i % 50 == 0) {
    cublasSgemv(‘n’, N, N, 1.0, d_A, N, d_x, 1, 0, d_y, 1);
    cublasScopy(N, d_b, 1, d_r, 1);
    cublasSaxpy(N, -1.0, d_y, 1, d_r, 1);
  }
  else
    cublasSaxpy(N,-alpha,d_y,1,d_r,1);
...
```

CUBLAS Features



- Single precision data:
  - Level 1 (vector-vector O(N))
  - Level 2 (matrix-vector O(N^2))
  - Level 3 (matrix-matrix O(N^3))
- Complex single precision data:
  - Level 1
  - CGEMM
- Double precision data:
  - Level 1: DASUM, DAXPY, DCOPY, DDOT, DNRM2, DROT, DROTM, DSCAL, DSWAP, ISAMAX, IDAMIN
  - Level 2: DGEMV, DGER, DSYR, DTRSV
  - Level 3: ZGEMM, DGEMM, DTRSM, DTRMM, DSYMM, DSYRK, DSYR2K

CUBLAS: Performance – CPU vs GPU



CUBLAS

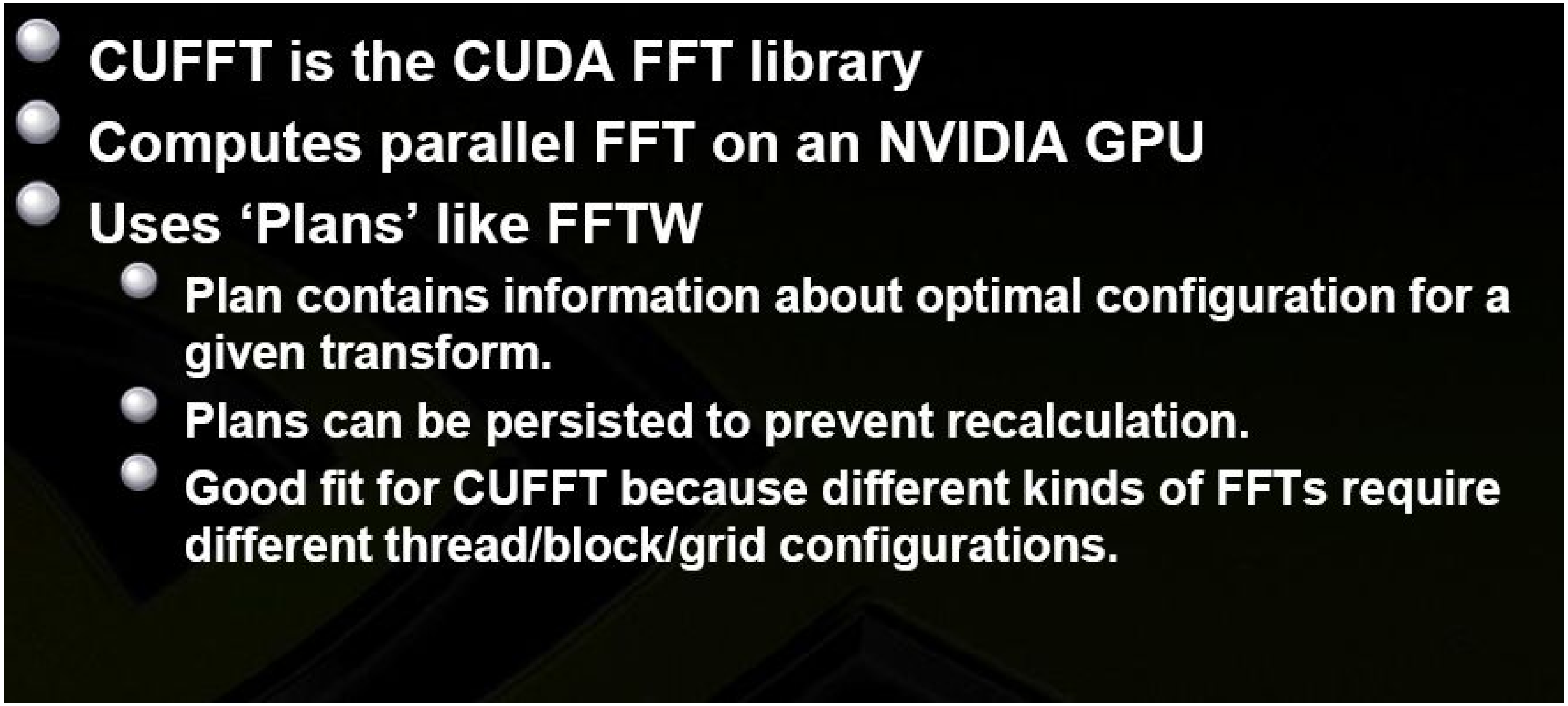
- GPU variant **100 times faster** than CPU version
- **Matrix size is unlimited** (limited by graphics card memory and texture size)
- Although taking advantage of sparce matrices will help reduce memory consumption, sparse matrix storage is not implemented by CUBLAS.

CUDA Specialized Libraries: CUFFT

CUDA Specialized Libraries: CUFFT

- **Cu**da Based **F**ast **F**ourier **T**ransform Library.
- The FFT is a divide-and-conquer algorithm for efficiently computing discrete Fourier transforms of complex or real-valued data sets,
- One of the most important and widely used numerical algorithms, with applications that include computational physics and general signal processing

CUFFT

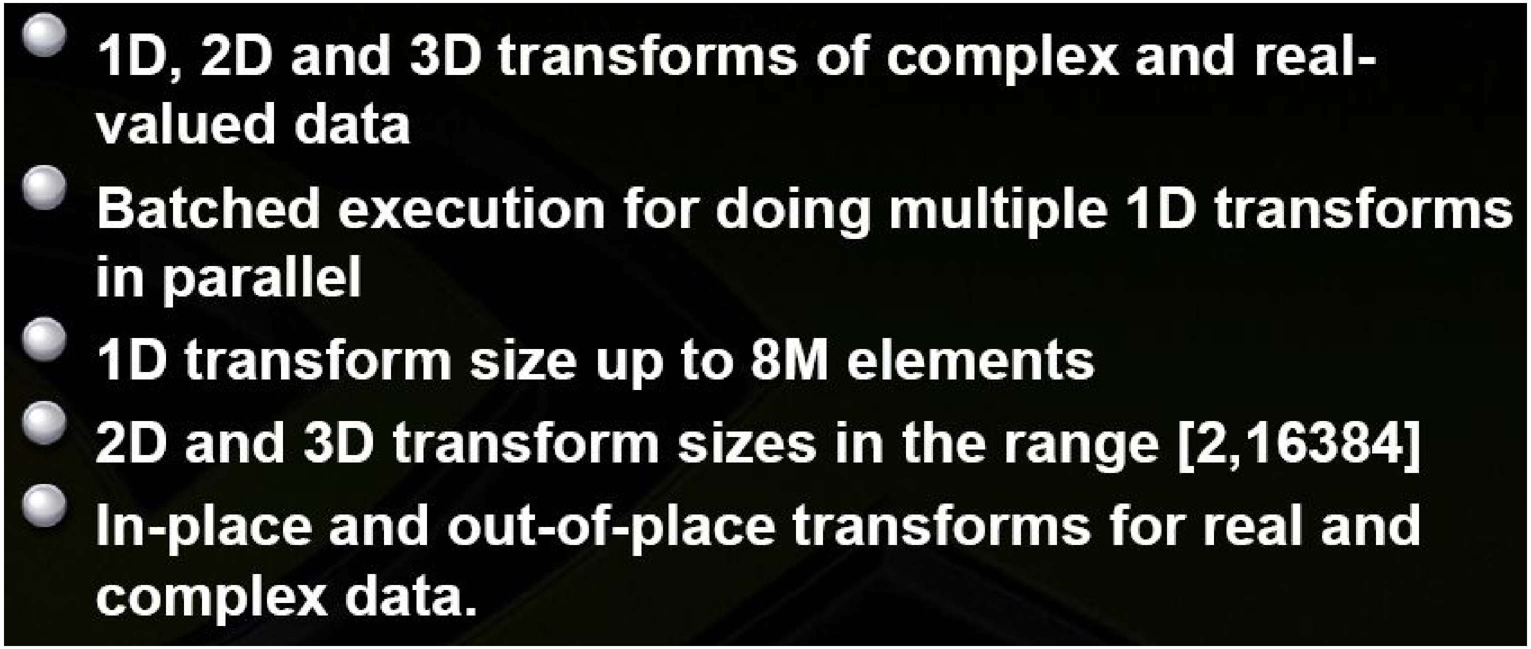


- CUFFT is the CUDA FFT library
- Computes parallel FFT on an NVIDIA GPU
- Uses ‘Plans’ like FFTW
  - Plan contains information about optimal configuration for a given transform.
  - Plans can be persisted to prevent recalculation.
  - Good fit for CUFFT because different kinds of FFTs require different thread/block/grid configurations.

CUFFT

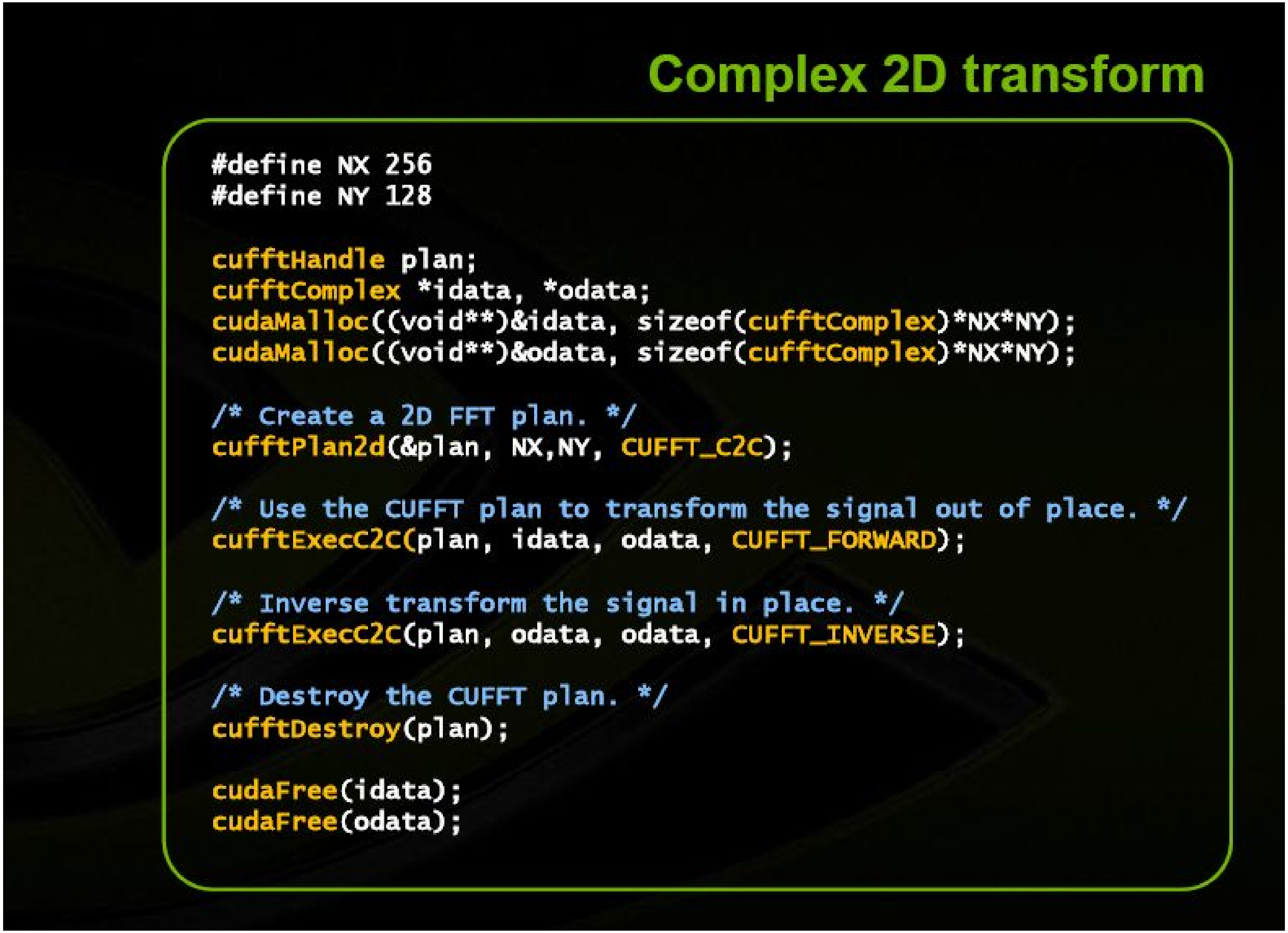
- If number of elements <8192, that it is **slower** than parallel **fftw**
- If >8192, **5x speedup** over threaded **fftw**

and **10x speedup** over serial fftw.



- **1D, 2D and 3D transforms of complex and real-valued data**
- **Batched execution for doing multiple 1D transforms in parallel**
- **1D transform size up to 8M elements**
- **2D and 3D transform sizes in the range [2,16384]**
- **In-place and out-of-place transforms for real and complex data.**

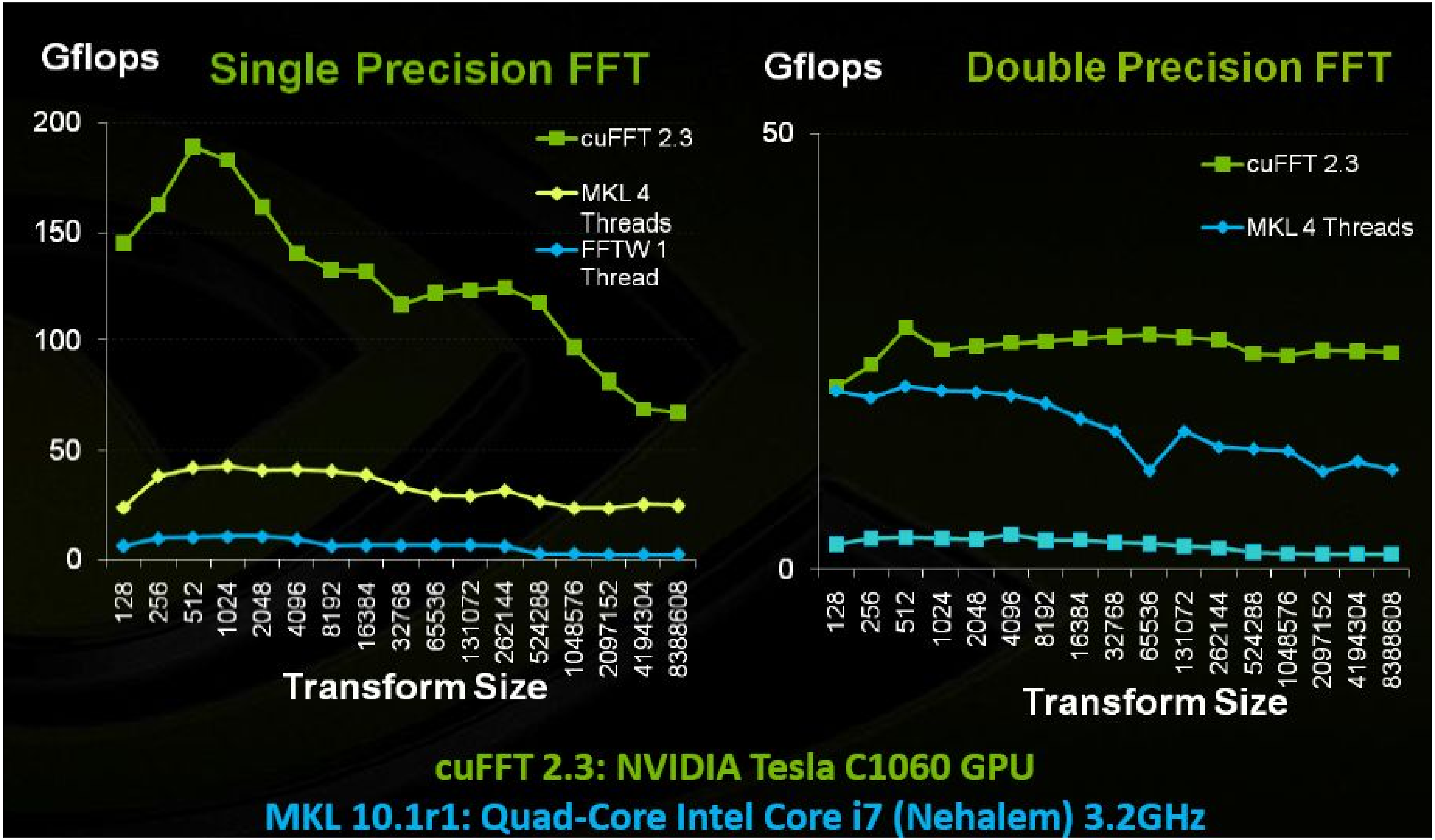
CUFFT: Example



Complex 2D transform

```
#define NX 256
#define NY 128

cufftHandle plan;
cufftComplex *idata, *odata;
cudaMalloc((void**)&idata, sizeof(cufftComplex)*NX*NY);
cudaMalloc((void**)&odata, sizeof(cufftComplex)*NX*NY);

/* Create a 2D FFT plan. */
cufftPlan2d(&plan, NX,NY, CUFFT_C2C);

/* Use the CUFFT plan to transform the signal out of place. */
cufftExecC2C(plan, idata, odata, CUFFT_FORWARD);

/* Inverse transform the signal in place. */
cufftExecC2C(plan, odata, odata, CUFFT_INVERSE);

/* Destroy the CUFFT plan. */
cufftDestroy(plan);

cudaFree(idata);
cudaFree(odata);
```

CUFFT: Performance – CPU vs GPU



cuFFT 2.3: NVIDIA Tesla C1060 GPU

MKL 10.1r1: Quad-Core Intel Core i7 (Nehalem) 3.2GHz

CUDA Specialized Libraries: MAGMA

CUDA Specialized Libraries: MAGMA

- Matrix Algebra on GPU and Multicore Architectures
- MAGMA aims to develop a dense linear algebra library **similar to LAPACK**, but for **heterogeneous/hybrid** architectures

  like the current "Multicore+GPU" systems.

MAGMA: Matrix Algebra on GPU and Multicore Architectures

MAGMA and LAPACK

- MAGMA - based on LAPACK, extended for heterogeneous systems
- MAGMA - similar to LAPACK in functionality, data storage, interface

Features

- Goal: easy porting from LAPACK to take advantage of the new GPU + multicore architectures
- Leverage: experience developing open source Linear Algebra software (LAPACK, ScaLAPACK, BLAS, ATLAS)
- Incorporate: newest numerical developments (e.g. communication avoiding algorithms) and experiences on homogeneous multicores (e.g. PLASMA)

MAGMA Developers

- University of Tennessee, Knoxville
- University of California, Berkeley
- University of Colorado, Denver
- Number of contributors from the LA community

© NVIDIA Corporation 2009

Portions of this slide courtesy Stan Tomov

MAGMA Release

MAGMA version 0.1 (08/04/09)

- One-sided factorizations [for linear solvers] in single and double precision arithmetic
- Hardware target: 1 core + 1 GPU (CUDA enabled)

MAGMA version 0.2 (11/14/09)

- One-sided factorizations in complex arithmetic
- Two-sided factorizations for eigenvalue solvers
- Linear solvers, including least squares and mixed precision iterative solvers
- MAGMA BLAS (gemm optimized for rectangular matrices, triangular solvers, gemv, etc)
- Hardware target:
  - 1 core + 1 GPU (all)
  - multicore + multi-GPU (one-sided factorizations)

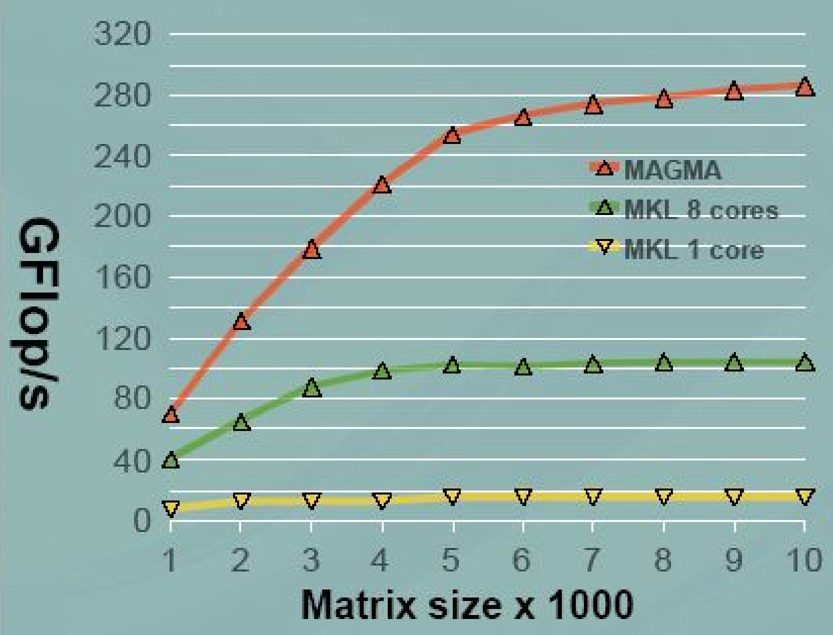
© NVIDIA Corporation 2009

Portions of this slide courtesy Stan Tomov

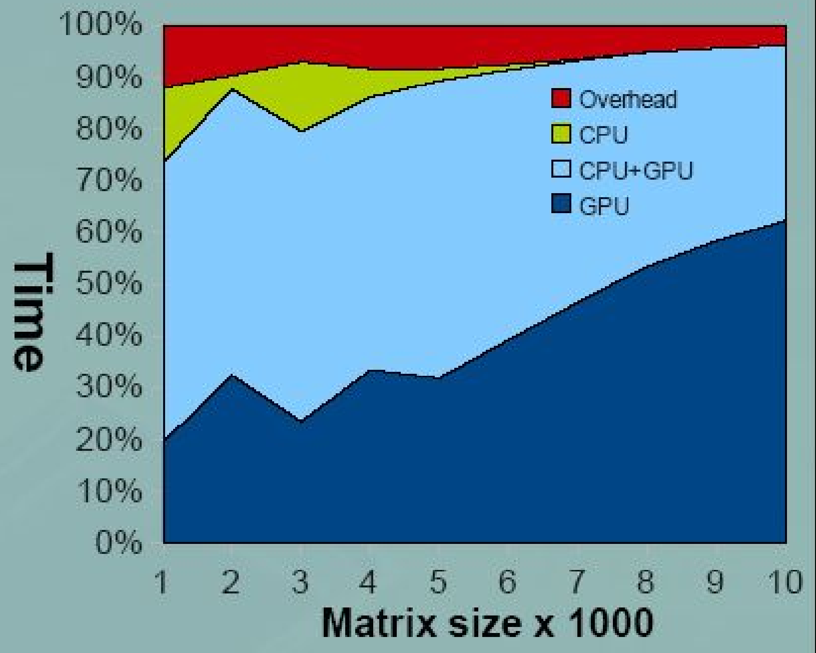
MAGMA Version 0.1 Performance

QR factorization in single precision arithmetic, CPU interface

Performance of MAGMA vs MKL



MAGMA QR time breakdown



GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

GPU BLAS : CUBLAS 2.2, sgemm peak: 375 GFlop/s
CPU BLAS : MKL 10.0 , sgemm peak: 128 GFlop/s

For more performance data, see http://icl.cs.utk.edu/magma

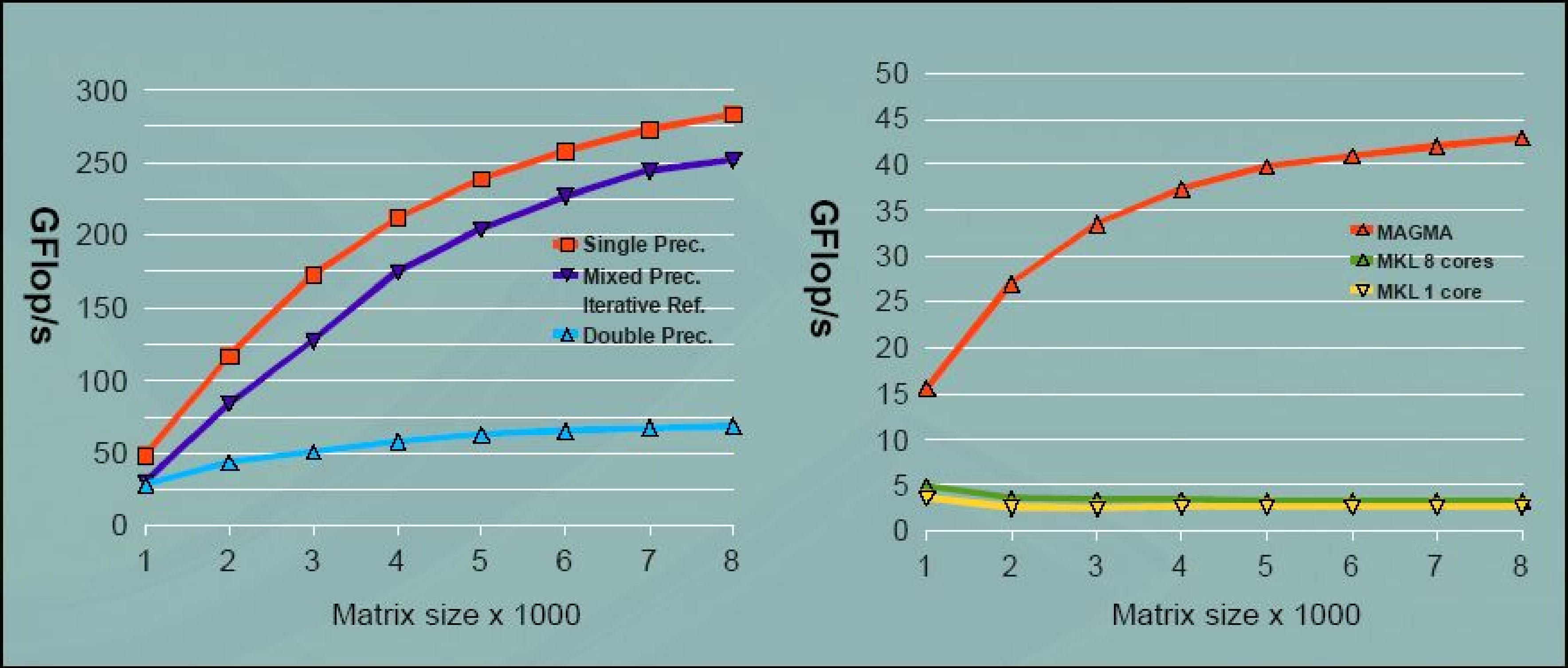
© NVIDIA Corporation 2009

Portions of this slide courtesy Stan Tomov

MAGMA Version 0.2 Performance

Linear Solvers
[e.g. A x = b using LU Factorization]

Hessenberg factorization
[e.g. double precision, CPU interface]



GPU : NVIDIA GeForce GTX 280 (240 cores @ 1.30GHz)
CPU : Intel Xeon dual socket quad-core (8 cores @2.33 GHz)

GPU BLAS : CUBLAS 2.2, dgemm peak: 75 GFlop/s
CPU BLAS : MKL 10.0   , dgemm peak: 65 GFlop/s

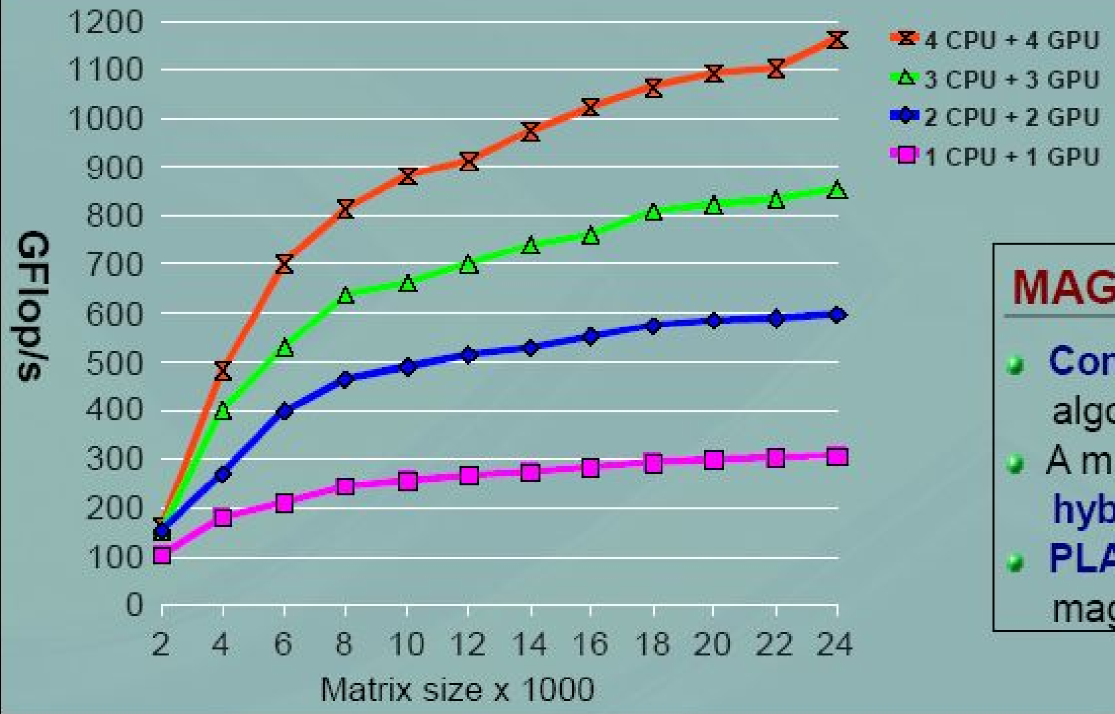
For more performance data, see http://icl.cs.utk.edu/magma

© NVIDIA Corporation 2009

Portions of this slide courtesy Stan Tomov

MAGMA Multi-GPU Performance

Cholesky factorization in single precision arithmetic
Performance and scalability on 4 GPUs



MAGNUM-tiles approach

- **Communication-avoiding/tile** type algorithms on large (magnum) tiles
- A magnum tile/task is defined for **hybrid 1 CPU + 1 GPU** computing
- **PLASMA scheduling** on the magnum tiles/tasks

GPU : NVIDIA Tesla C1070 (4 GPUs @1.44GHz)
CPU : AMD Opteron dual socket dual-core (4 cores @1.8 GHz)

For more performance data, see http://icl.cs.utk.edu/magma

© NVIDIA Corporation 2009

Portions of this slide courtesy Stan Tomov

CUDA Specialized Libraries: CULA

CUDA Specialized Libraries: CULA

- CULA is EM Photonics' GPU-accelerated numerical linear algebra library that contains a growing list of LAPACK functions.
- LAPACK stands for **L**inear **A**lgebra **PACK**age. It is an industry standard computational library that has been in development for over 20 years and provides a large number of routines for factorization, decomposition, system solvers, and eigenvalue problems.

CULA | tools EM Photonics NVIDIA

3rd Party Implementation of LAPACK interface from EM Photonics (www.culatools.com)

CULA | basic

- Six popular single/complex-single LAPACK functions
- Free!

Function Name	Description
getrf	LU decomposition
gesv	System solve
geqrf	QR factorization
gesvd	Singular value decomposition
gels	Least squares
gglse	Constrained least squares

CULA | premium

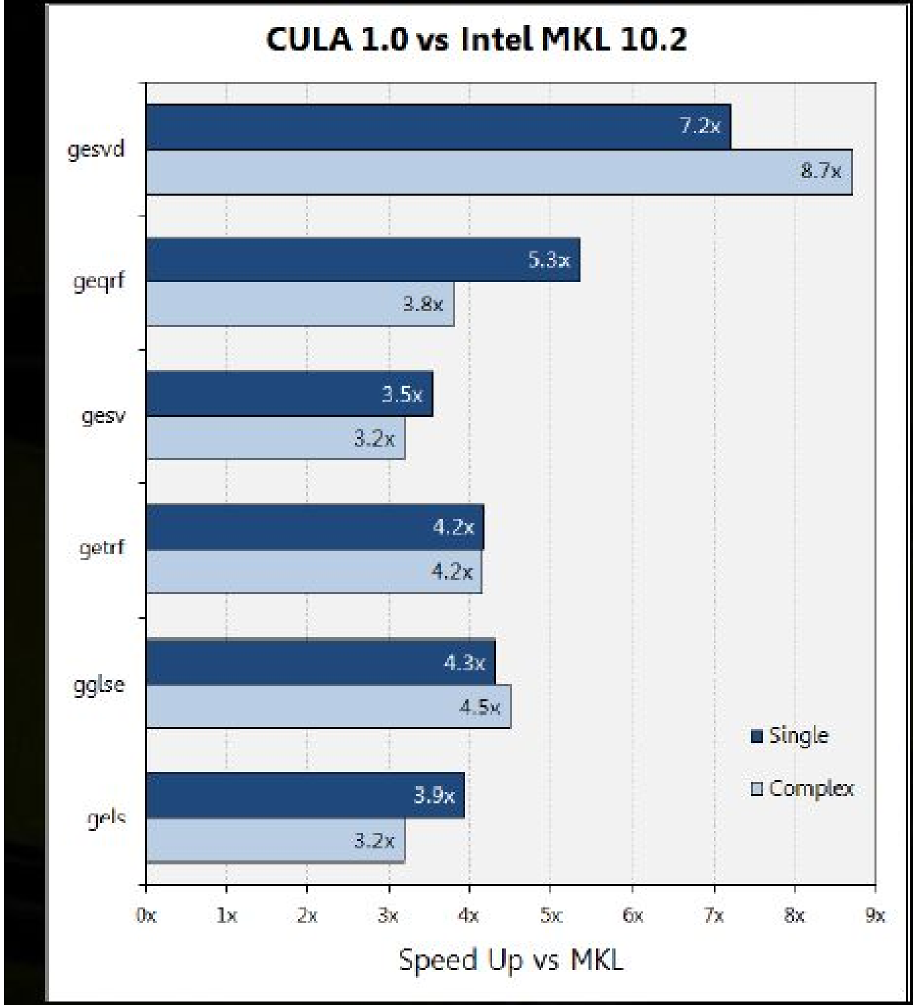
- Available for purchase
- Adds 18 more routines (and growing)
- Adds Double (D) / Double Complex (Z)

Function Name	Description
potrf	Cholesky factorization
gebrd	Bidiagonalization
getri	Matrix inversion
getrs	LU Backsolve
trtrs	Triangular solve
gelqf	LQ factorization
posv	Positive-definite system solve

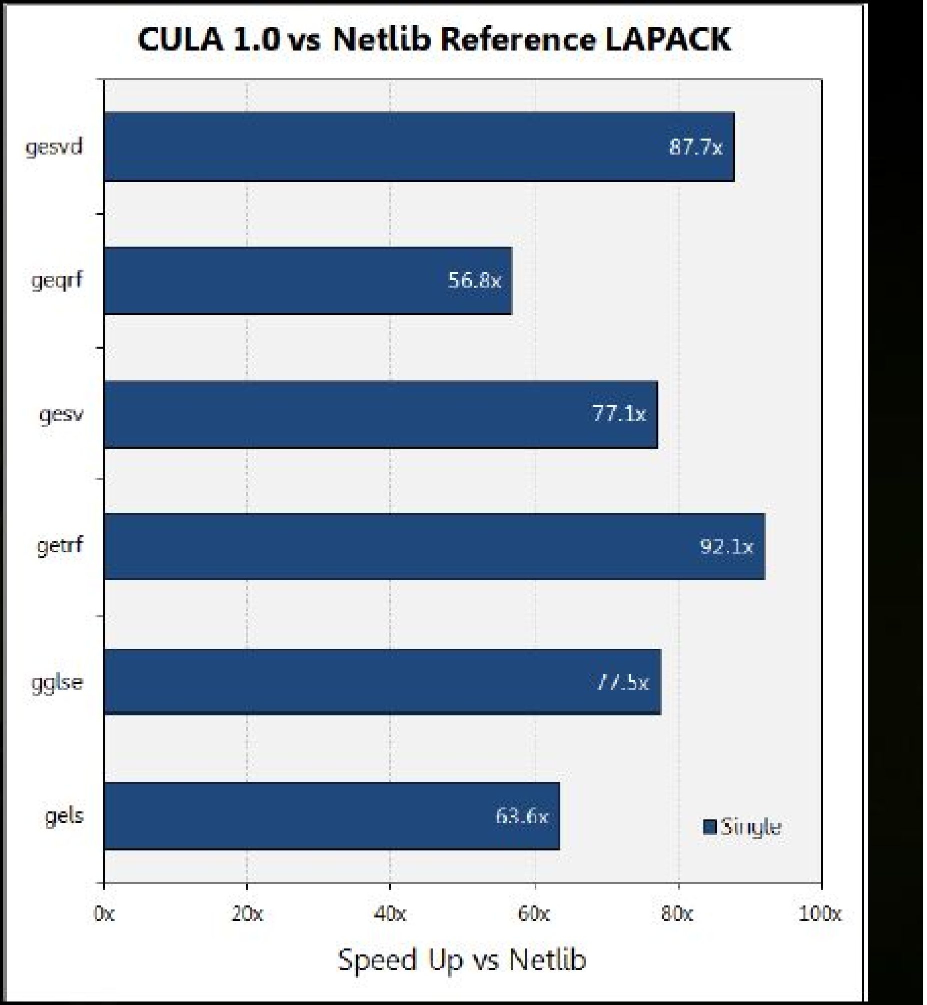
© NVIDIA Corporation 2009

Courtesy EM Photonics

CULA Performance



CULA 1.0 vs Intel MKL 10.2



CULA 1.0 vs Netlib Reference LAPACK

Tesla C1060 vs Intel Core i7, matrix size ~10,000x10,000

© NVIDIA Corporation 2009

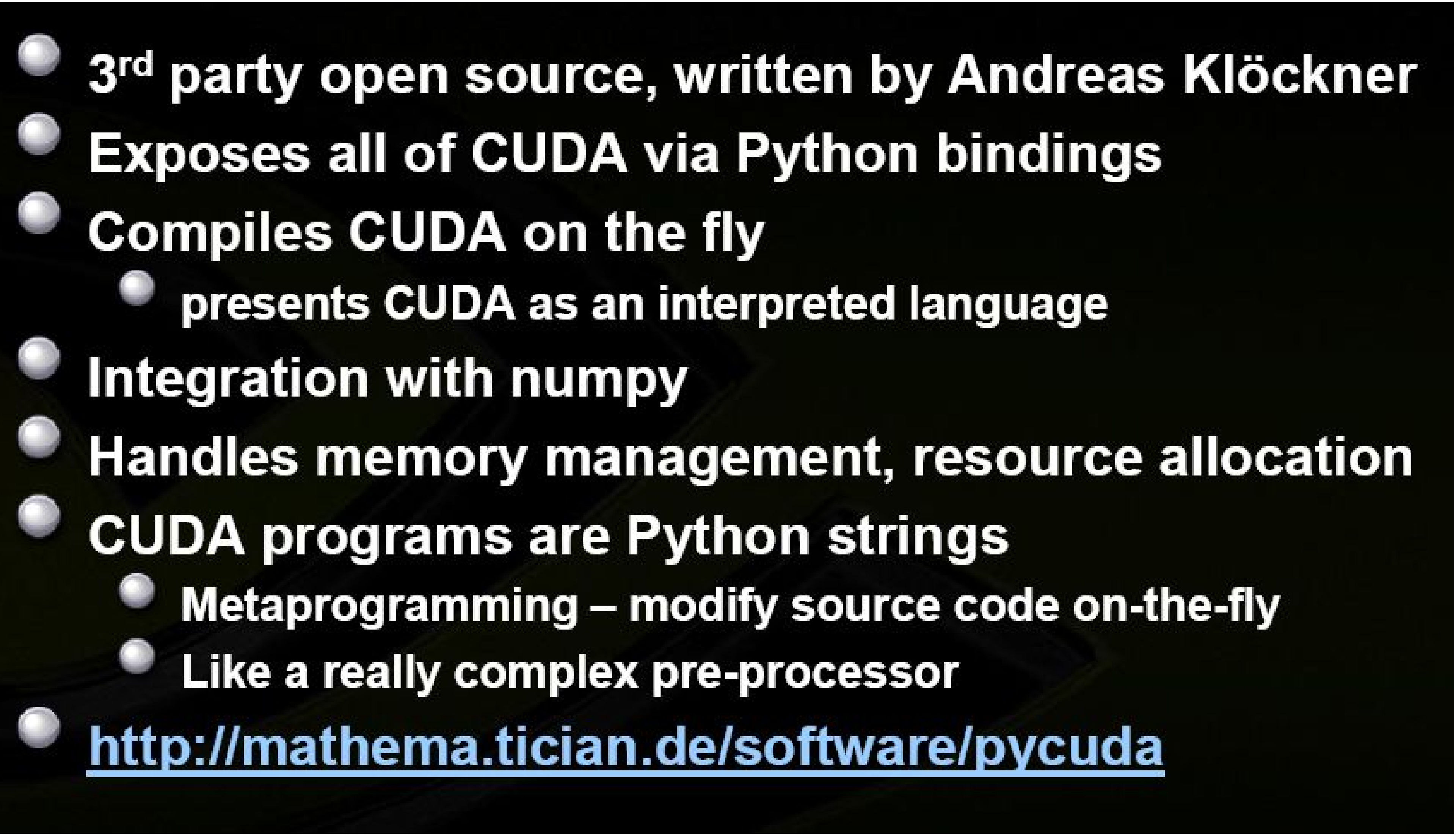
Courtesy EM Photonics

CUDA Specialized Libraries: PyCUDA

CUDA Specialized Libraries: PyCUDA

- **PyCUDA** – **Py**thon **CUDA**
- It lets you access Nvidia CUDA parallel computation API from Python

PyCUDA

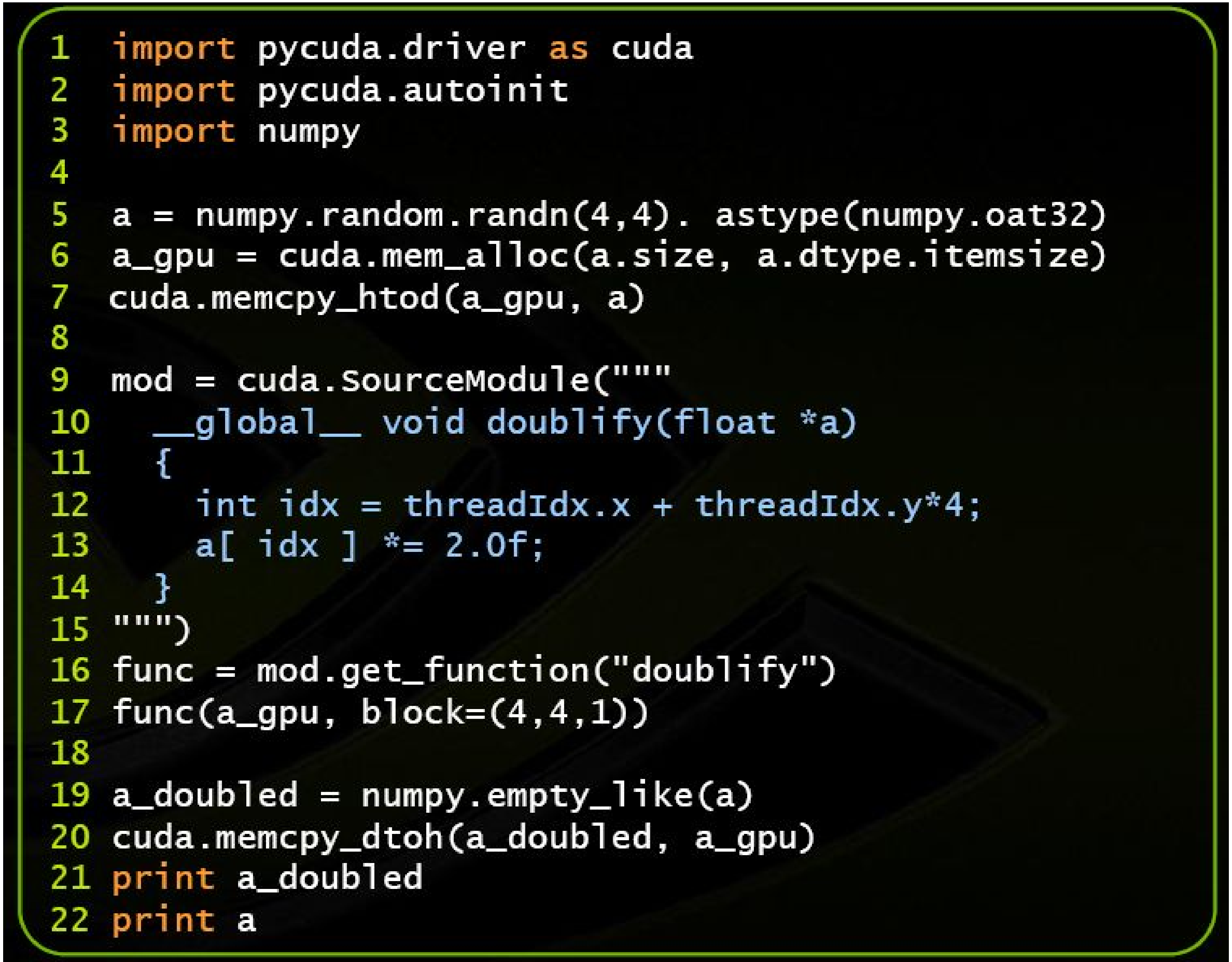


- 3rd party open source, written by Andreas Klöckner
- Exposes all of CUDA via Python bindings
- Compiles CUDA on the fly
  - presents CUDA as an interpreted language
- Integration with numpy
- Handles memory management, resource allocation
- CUDA programs are Python strings
  - Metaprogramming – modify source code on-the-fly
  - Like a really complex pre-processor
- http://mathema.tician.de/software/pycuda

PyCUDA - Differences

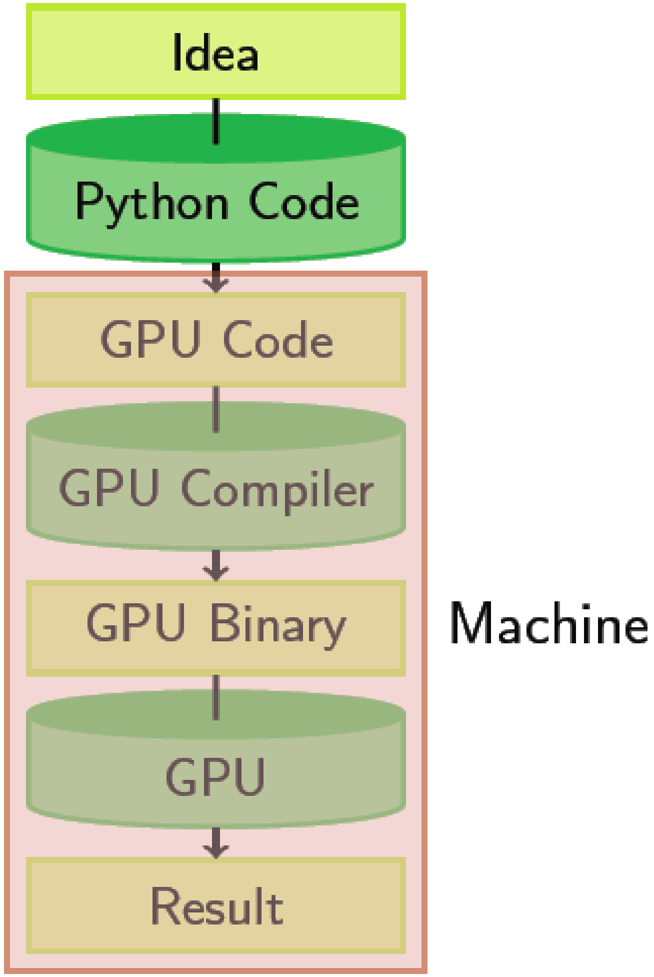
- Object cleanup tied to lifetime of objects. This idiom, often called RAII in C++, makes it much easier to write correct, leak- and crash-free code. PyCUDA knows about dependencies, too, so (for example) it won't detach from a context before all memory allocated in it is also freed.
- Convenience. Abstractions like pycuda.driver.SourceModule and pycuda.gpuarray.GPUArray make CUDA programming even more convenient than with Nvidia's C-based runtime.
- Completeness. PyCUDA puts the full power of CUDA's driver API at your disposal, if you wish.
- Automatic Error Checking. All CUDA errors are automatically translated into Python exceptions.
- Speed. PyCUDA's base layer is written in C++, so all the niceties above are virtually free.

PyCUDA - Example



```
import pycuda.driver as cuda
import pycuda.autoinit
import numpy

a = numpy.random.randn(4,4). astype(numpy.oat32)
a_gpu = cuda.mem_alloc(a.size, a.dtype.itemsize)
cuda.memcpy_htod(a_gpu, a)

mod = cuda.SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[ idx ] *= 2.0f;
  }
""")
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```

Metaprogramming



In GPU scripting, GPU code does *not* need to be a compile-time constant.

(Key: Code is data–it *wants* to be reasoned about at run time)

CUDA Specialized Libraries: CUDPP

CUDA Specialized Libraries: CUDPP

- **CUDPP**: CUDA Data Parallel Primitives Library
- CUDPP is a library of data-parallel algorithm primitives such as
  - parallel prefix-sum (”scan”)
  - parallel sort
  - parallel reduction

CUDPP – Design Goals

- **Performance**: aims to provide best-of-class performance for simple primitives.
- **Modularity**: primitives easily included in other applications.
  - CUDPP is provided as a library that can link against other applications.
  - CUDPP calls run on the GPU on GPU data. Thus they can be used as standalone calls on the GPU (on GPU data initialized by the calling application) and, more importantly, as GPU components in larger CPU/GPU applications

CUDPP – Layers

CUDPP is implemented as 4 layers:

- **Public Interface** is the external library interface, which is the entry point for most applications. It calls into the **Application-Level API**.
- **Application-Level API** comprises functions callable from CPU code. These functions execute code jointly on the CPU (host) and the GPU by calling into the **Kernel-Level API** below them.
- **Kernel-Level API** comprises functions that run entirely on the GPU across an entire grid of thread blocks. They may call the CTA-Level API.
- **CTA (Cooperative Thread Array)-Level API** comprises functions that run entirely on the GPU within a single CTA (thread) block. They are low-level functions that implement core data-parallel algorithms, typically by processing data within CUDA shared memory.

CUDPP

CUDPP_DLL CUDPPResult cudppSparseMatrixVectorMultiply(CUDPPHandle *sparseMatrixHandle*, void * *d_y*,const void * *d_x*)

Perform matrix-vector multiply y = A*x for arbitrary sparse matrix A and vector x.

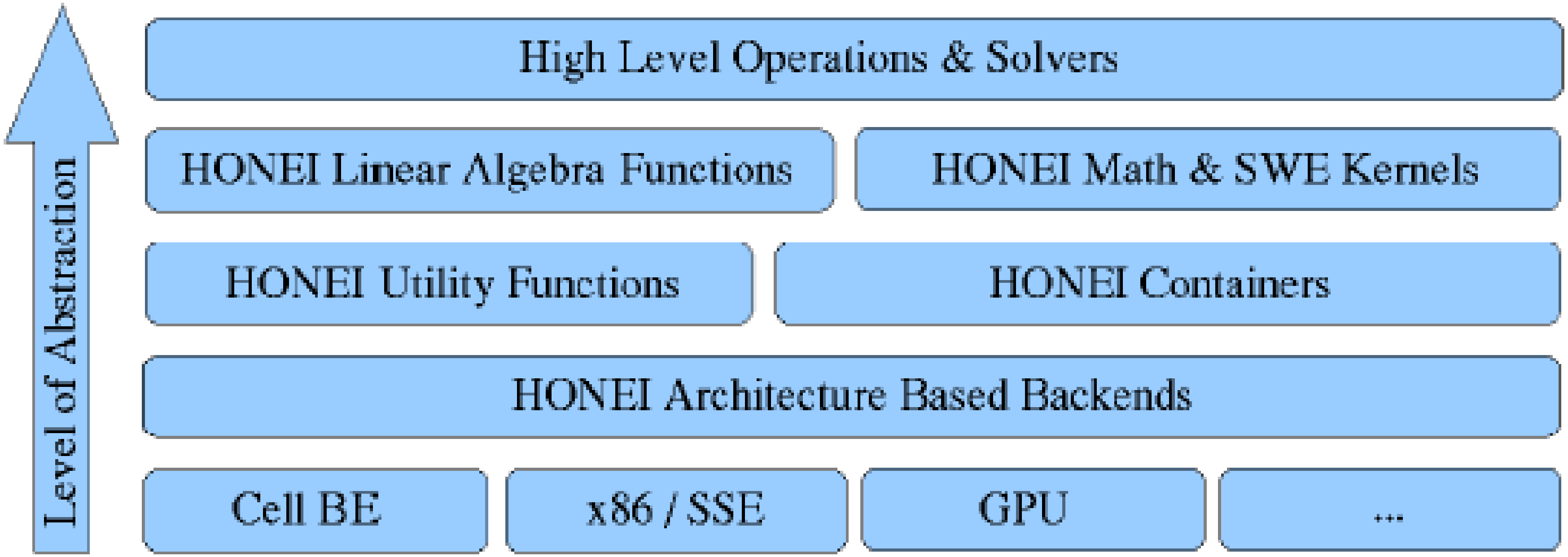
CUDPP - Example

```
CUDPPScanConfig config;
    config.direction = CUDPP_SCAN_FORWARD; config.exclusivity =
    CUDPP_SCAN_EXCLUSIVE; config.op = CUDPP_ADD;
    config.datatype = CUDPP_FLOAT; config.maxNumElements = numElements;
    config.maxNumRows = 1;
    config.rowPitch = 0;
cudppInitializeScan(&config);
cudppScan(d_odata, d_idata, numElements, &config);
```

CUDA Specialized Libraries: HONEI

CUDA Specialized Libraries: HONEI

A collection of libraries for numerical computations targeting multiple processor architectures



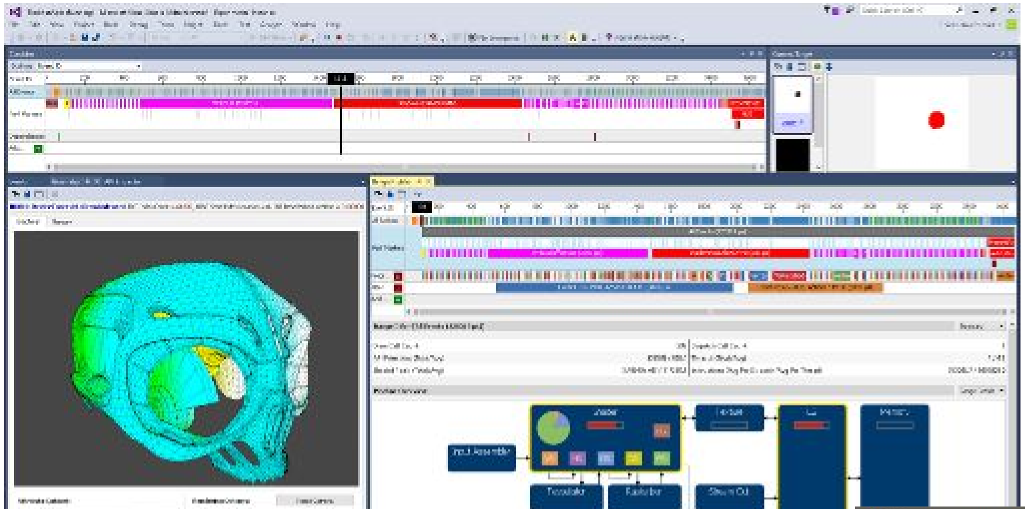
HONEI

- **HONEI**, an open-source collection of libraries offering a hardware oriented approach to numerical calculations.
- **HONEI** abstracts the hardware, and applications written on top of HONEI can be executed on a wide range of computer architectures such as CPUs, GPUs and the Cell processor.
  - The most important frontend library is **libhoneila**, HONEI's linear algebra library. It provides templated container classes for different matrix and vector types.
  - The numerics and math library **libhoneimath** contains high performance kernels for iterative linear system solvers as well as other useful components like interpolation and approximation.

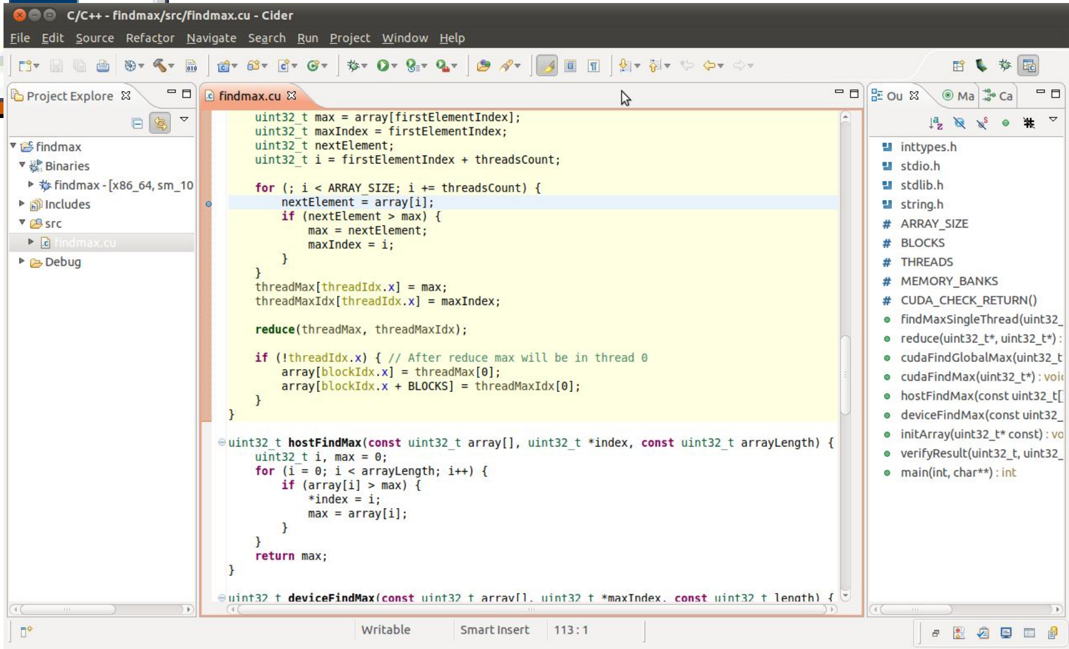
CUDA Development Tools

CUDA Development Tools:
NVIDIA Nsight

Nsight in Various Integrated Development Environment



Nsight
Visual Studio Edition


C/C++ - findmax/src/findmax.cu - Cider
File Edit Source Refactor Navigate Search Run Project Window Help
Project Explore
findmax
Binaries
findmax - [x86_64, sm_10
Includes
src
findmax.cu
Debug
findmax.cu
Ou
Ma
Ca
inttypes.h
stdio.h
stdlib.h
string.h
ARRAY_SIZE
BLOCKS
THREADS
MEMORY_BANKS
CUDA_CHECK_RETURN()
findMaxSingleThread(uint32_
reduce(uint32_t*, uint32_t*) :
cudaFindGlobalMax(uint32_t
cudaFindMax(uint32_t*) : voi
hostFindMax(const uint32_t[
deviceFindMax(const uint32_
initArray(uint32_t* const) : vo
verifyResult(uint32_t, uint32_
main(int, char**) : int
Writable
Smart Insert
113 : 1

```
    uint32_t max = array[firstElementIndex];
    uint32_t maxIndex = firstElementIndex;
    uint32_t nextElement;
    uint32_t i = firstElementIndex + threadsCount;

    for (; i < ARRAY_SIZE; i += threadsCount) {
        nextElement = array[i];
        if (nextElement > max) {
            max = nextElement;
            maxIndex = i;
        }
    }
    threadMax[threadIdx.x] = max;
    threadMaxIdx[threadIdx.x] = maxIndex;

    reduce(threadMax, threadMaxIdx);

    if (!threadIdx.x) { // After reduce max will be in thread 0
        array[blockIdx.x] = threadMax[0];
        array[blockIdx.x + BLOCKS] = threadMaxIdx[0];
    }
}

uint32_t hostFindMax(const uint32_t array[], uint32_t *index, const uint32_t arrayLength) {
    uint32_t i, max = 0;
    for (i = 0; i < arrayLength; i++) {
        if (array[i] > max) {
            *index = i;
            max = array[i];
        }
    }
    return max;
}

uint32_t deviceFindMax(const uint32_t array[], uint32_t *maxIndex, const uint32_t length) {
```

Nsight
Eclipse Edition

Nsight

- New project templates and integration with CUDA SDK samples make getting started quick and easy
- CUDA code highlighting makes it easy to navigate heterogeneous CUDA code
- Dynamic HLSL shader editing

CUDA Development Tools:

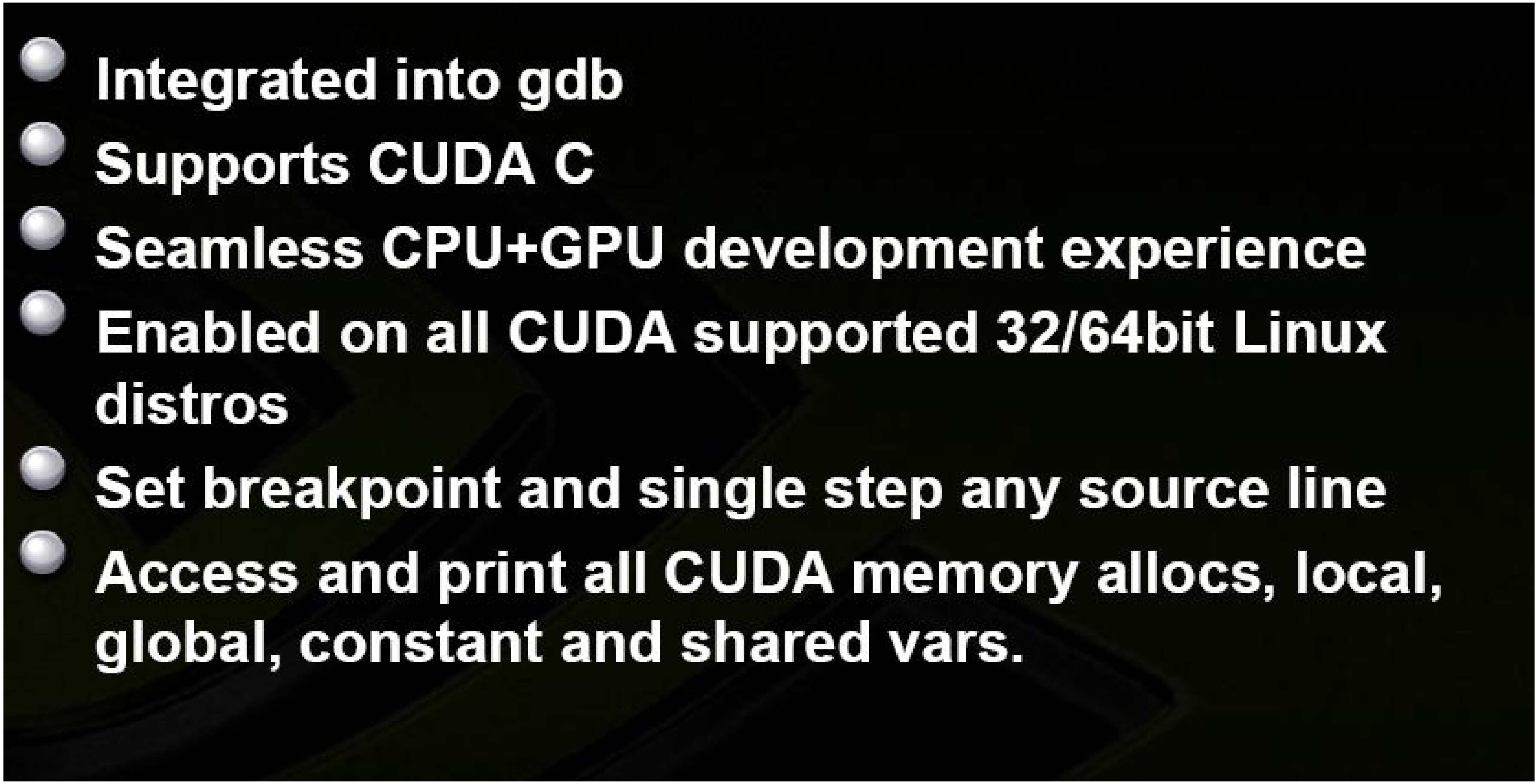
CUDA-gdb

vs.

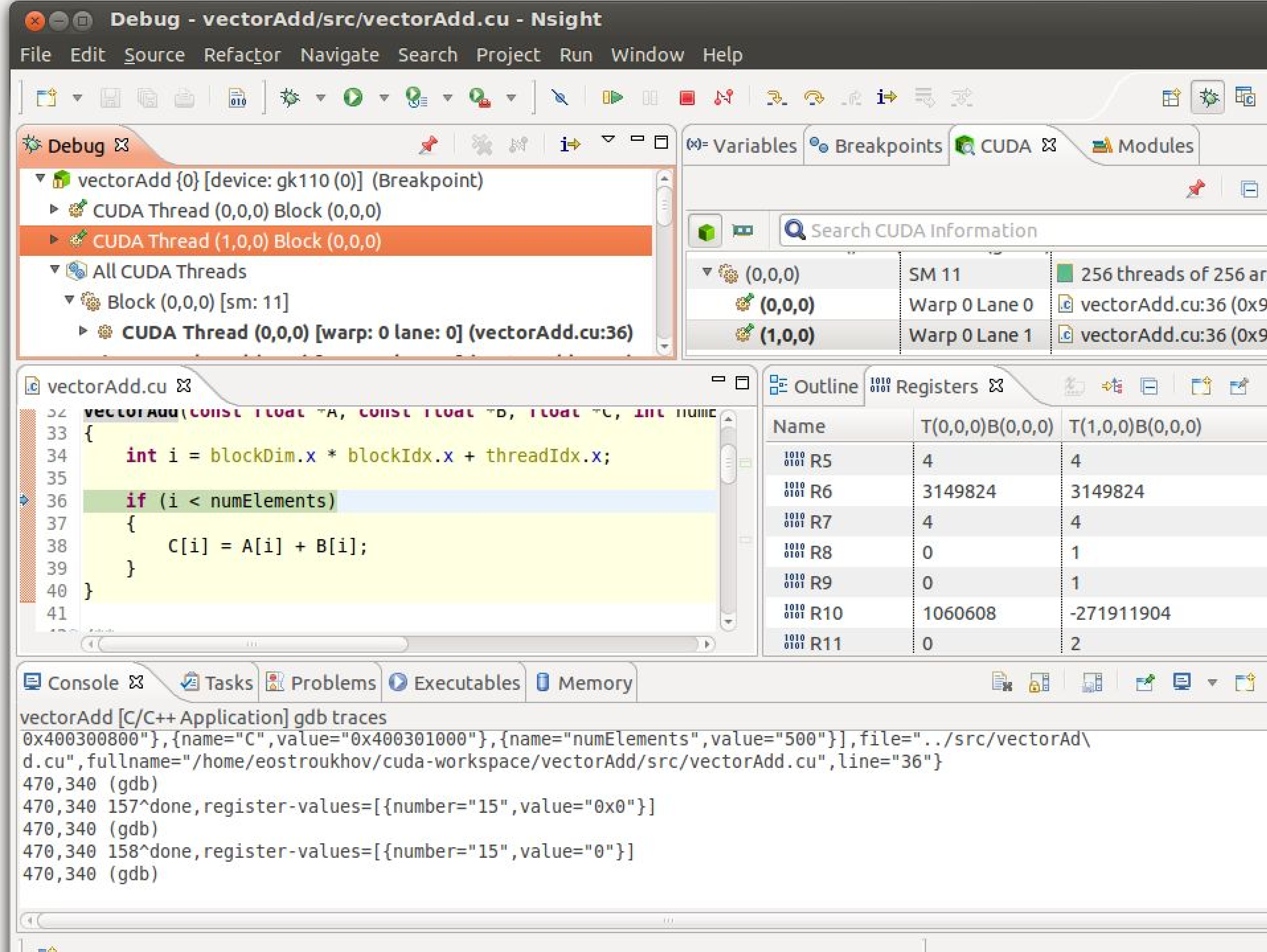
Nsight Debug tools

CUDA-gdb

Simple Debugger integrated into gdb



- Integrated into gdb
- Supports CUDA C
- Seamless CPU+GPU development experience
- Enabled on all CUDA supported 32/64bit Linux distros
- Set breakpoint and single step any source line
- Access and print all CUDA memory allocs, local, global, constant and shared vars.



vectorAdd {0} [device: gk110 (0)] (Breakpoint)
CUDA Thread (0,0,0) Block (0,0,0)
CUDA Thread (1,0,0) Block (0,0,0)
All CUDA Threads
Block (0,0,0) [sm: 11]
CUDA Thread (0,0,0) [warp: 0 lane: 0] (vectorAdd.cu:36)

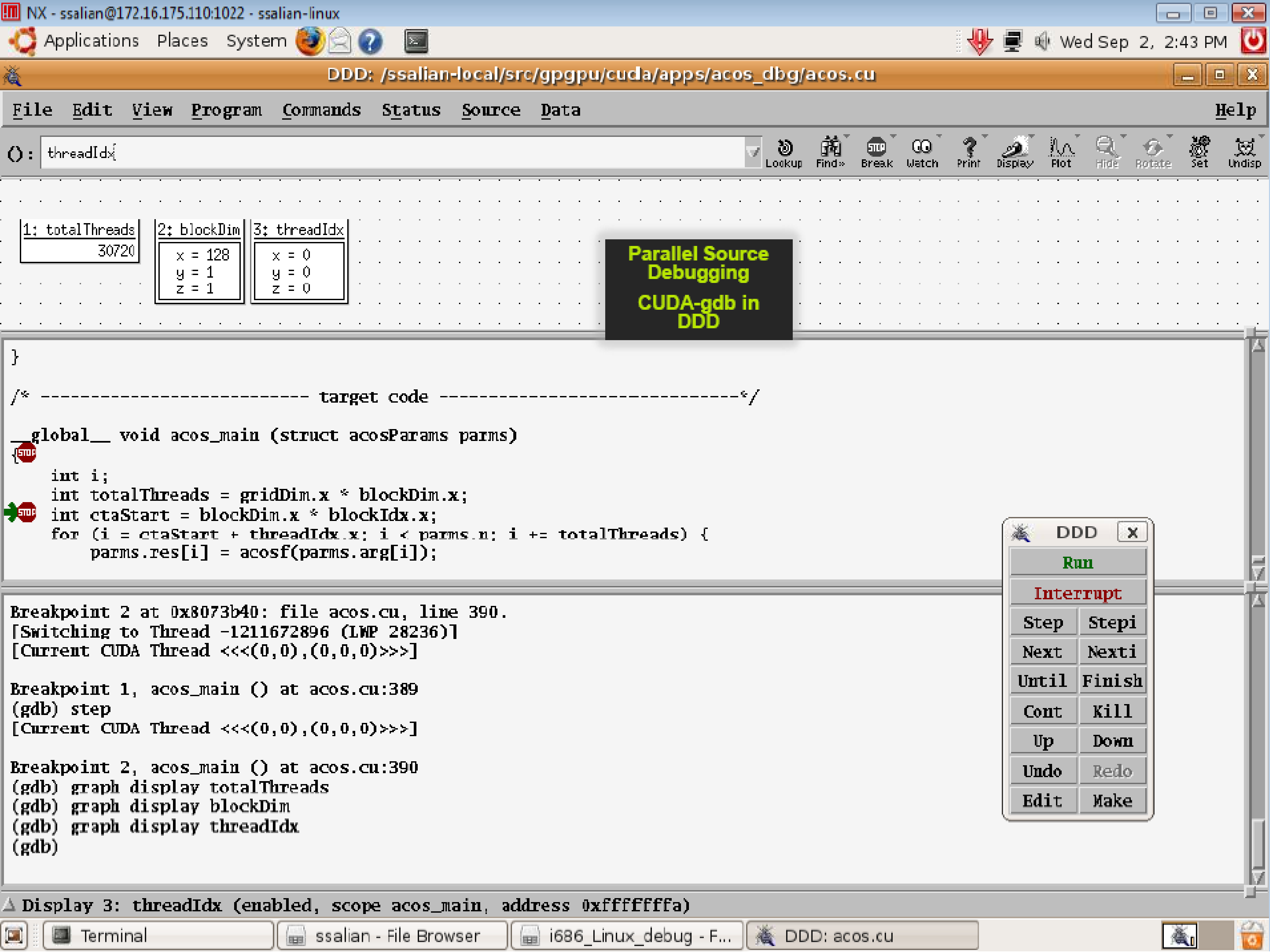
(0,0,0)	SM 11	🟩 256 threads of 256 ar
(0,0,0)	Warp 0 Lane 0	vectorAdd.cu:36 (0x9
(1,0,0)	Warp 0 Lane 1	vectorAdd.cu:36 (0x9

```
vectorAdd(const float *A, const float *B, float *C, int nume
{
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements)
    {
        C[i] = A[i] + B[i];
    }
}
```

Name	T(0,0,0)B(0,0,0)	T(1,0,0)B(0,0,0)
R5	4	4
R6	3149824	3149824
R7	4	4
R8	0	1
R9	0	1
R10	1060608	-271911904
R11	0	2

vectorAdd [C/C++ Application] gdb traces

```
0x400300800"},{name="C",value="0x400301000"},{name="numElements",value="500"}],file="../src/vectorAd\
d.cu",fullname="/home/eostroukhov/cuda-workspace/vectorAdd/src/vectorAdd.cu",line="36"}
470,340 (gdb)
470,340 157^done,register-values=[{number="15",value="0x0"}]
470,340 (gdb)
470,340 158^done,register-values=[{number="15",value="0"}]
470,340 (gdb)
```



Parallel Source Debugging

CUDA-gdb in DDD

```
1: totalThreads
30720

2: blockDim
x = 128
y = 1
z = 1

3: threadIdx
x = 0
y = 0
z = 0
```

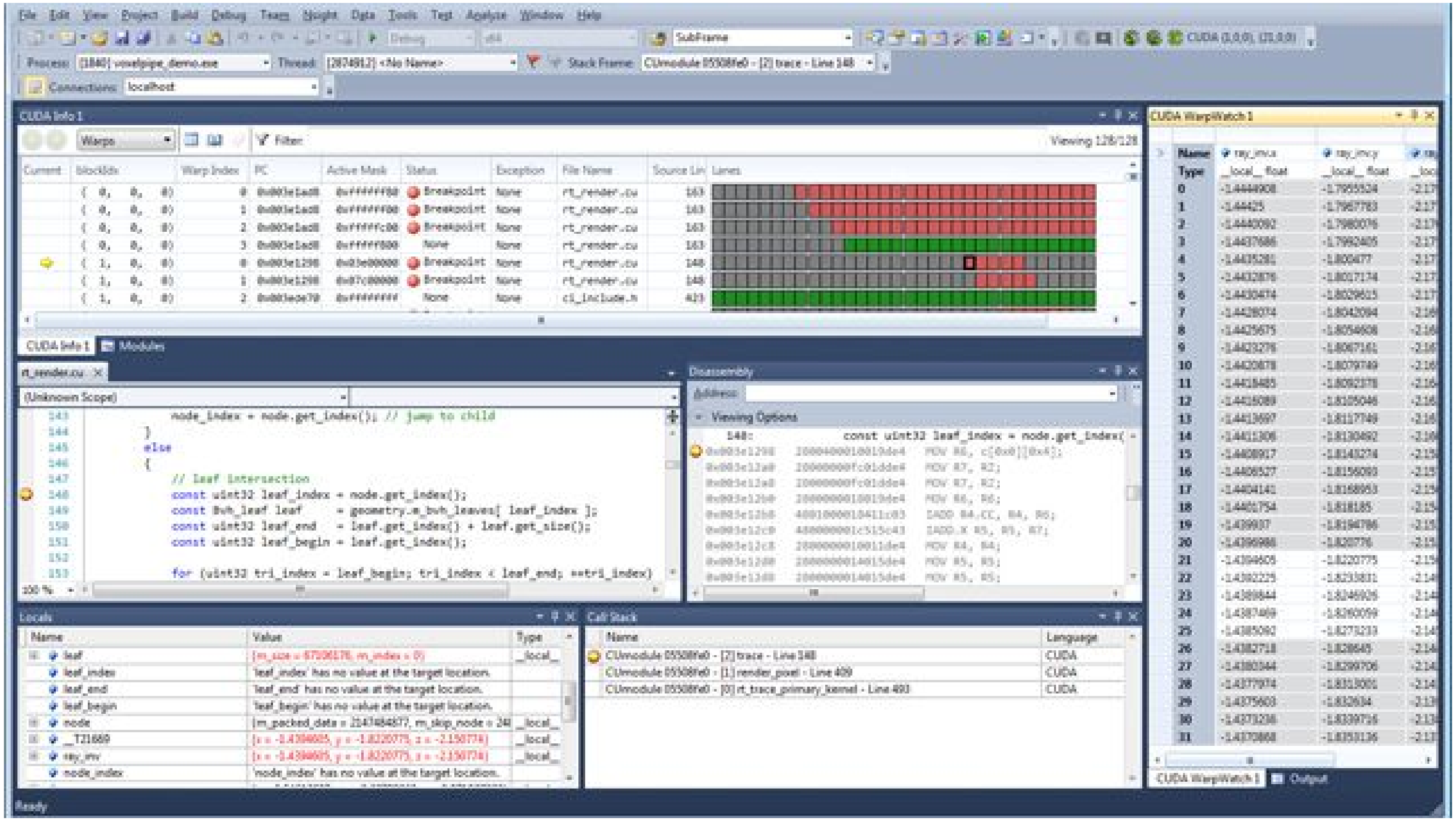
```
}

/* ------------------------- target code ----------------------------*/

__global__ void acos_main (struct acosParams parms)
{
    int i;
    int totalThreads = gridDim.x * blockDim.x;
    int ctaStart = blockDim.x * blockIdx.x;
    for (i = ctaStart + threadIdx.x; i < parms.n; i += totalThreads) {
        parms.res[i] = acosf(parms.arg[i]);
```

```
Breakpoint 2 at 0x8073b40: file acos.cu, line 390.
[Switching to Thread -1211672896 (LWP 28236)]
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 1, acos_main () at acos.cu:389
(gdb) step
[Current CUDA Thread <<<(0,0),(0,0,0)>>>]

Breakpoint 2, acos_main () at acos.cu:390
(gdb) graph display totalThreads
(gdb) graph display blockDim
(gdb) graph display threadIdx
(gdb)
```

```
Display 3: threadIdx (enabled, scope acos_main, address 0xfffffffa)
```

Nsight - Debug

- Debug CPU and GPU code simultaneously and seamlessly
- Debug shaders as they are being executed on the GPU
- Real-time inspection of Direct3D 9/10/11 API calls



Process: [1840] voxelpipe_demo.exe
Thread: [2874912] <No Name>
Stack Frame: CUmodule 05508fe0 - [2] trace - Line 148
Connections: localhost

Current	blockIdx	Warp Index	PC	Active Mask	Status	Exception	File Name	Source Lin	Lanes
	{ 0, 0, 0}	0	0x003e1ad8	0xffffff80	Breakpoint	None	rt_render.cu	163	
	{ 0, 0, 0}	1	0x003e1ad8	0xffffff00	Breakpoint	None	rt_render.cu	163	
	{ 0, 0, 0}	2	0x003e1ad8	0xfffffc00	Breakpoint	None	rt_render.cu	163	
	{ 0, 0, 0}	3	0x003e1ad8	0xfffff800	None	None	rt_render.cu	163	
⇨	{ 1, 0, 0}	0	0x003e1298	0x03e00000	Breakpoint	None	rt_render.cu	148	
	{ 1, 0, 0}	1	0x003e1298	0x07c00000	Breakpoint	None	rt_render.cu	148	
	{ 1, 0, 0}	2	0x003e0e70	0xffffffff	None	None	ci_include.h	423	

```
            node_index = node.get_index(); // jump to child
        }
        else
        {
            // leaf intersection
            const uint32 leaf_index = node.get_index();
            const Bvh_leaf leaf     = geometry.m_bvh_leaves[ leaf_index ];
            const uint32 leaf_end   = leaf.get_index() + leaf.get_size();
            const uint32 leaf_begin = leaf.get_index();

            for (uint32 tri_index = leaf_begin; tri_index < leaf_end; ++tri_index)
```

Address:

```
    148:                    const uint32 leaf_index = node.get_index(
0x003e1298   2000400001019de4   MOV R6, c[0x0][0x4];
0x003e12a0   20000000fc01dde4   MOV R7, R2;
0x003e12a8   20000000fc01dde4   MOV R7, R2;
0x003e12b0   2000000018019de4   MOV R6, R6;
0x003e12b8   4801000018411c03   IADD R4.CC, R4, R6;
0x003e12c0   480000001c515c43   IADD.X R5, R5, R7;
0x003e12c8   2000000010011de4   MOV R4, R4;
0x003e12d0   2000000014015de4   MOV R5, R5;
0x003e12d8   2000000014015de4   MOV R5, R5;
```

Locals

Name	Value	Type
leaf	{m_size = 67108176, m_index = 0}	_local_
leaf_index	'leaf_index' has no value at the target location.	
leaf_end	'leaf_end' has no value at the target location.	
leaf_begin	'leaf_begin' has no value at the target location.	
node	{m_packed_data = 2147484877, m_skip_node = 24(	_local_
__T21680	{x = -1.4394605, y = -1.8220775, z = -2.150774}	_local_
ray_inv	{x = -1.4394605, y = -1.8220775, z = -2.150774}	_local_
node_index	'node_index' has no value at the target location.	

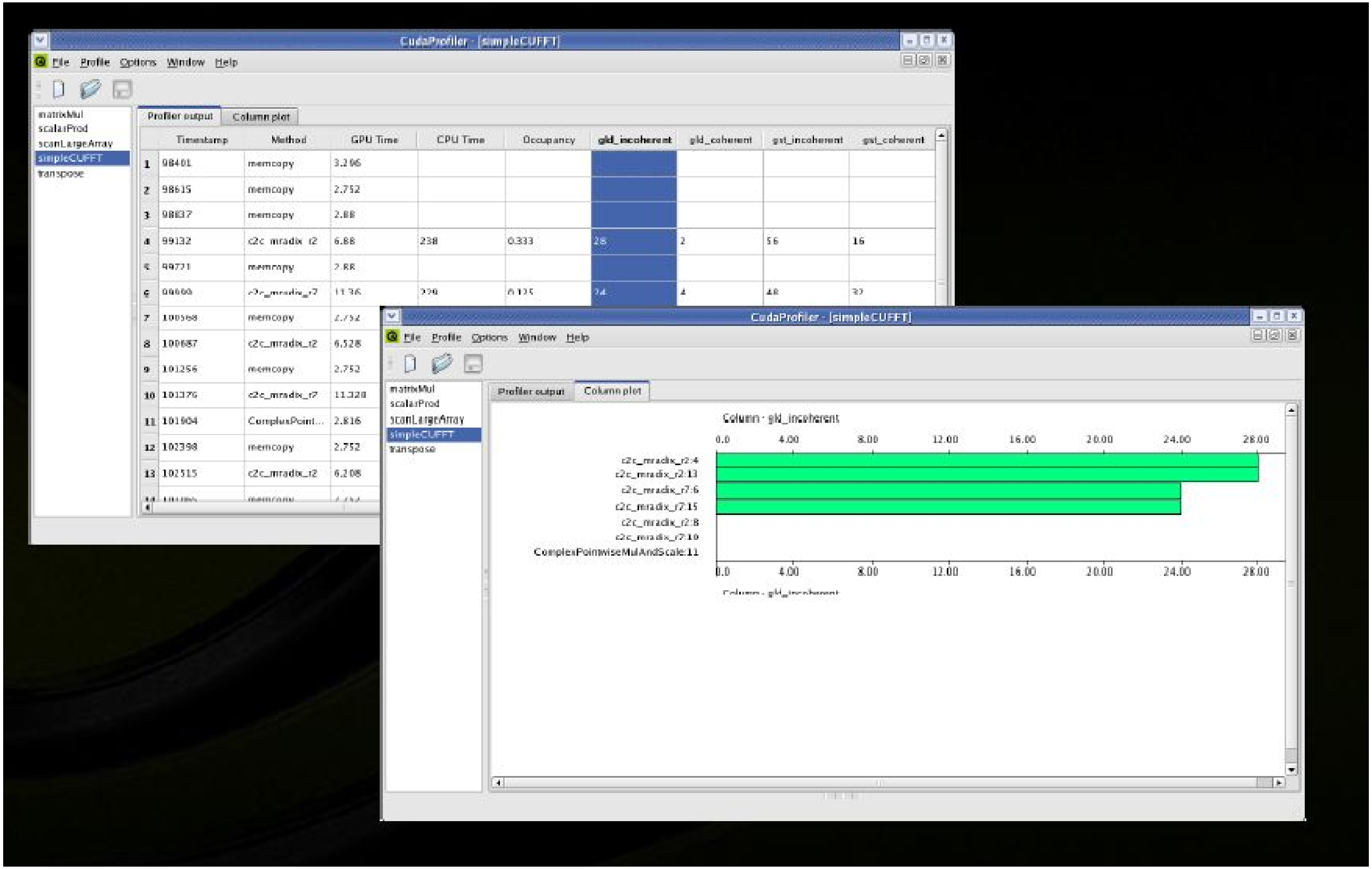
Call Stack

Name	Language
CUmodule 05508fe0 - [2] trace - Line 148	CUDA
CUmodule 05508fe0 - [1] render_pixel - Line 409	CUDA
CUmodule 05508fe0 - [0] rt_trace_primary_kernel - Line 493	CUDA

Name	ray_inv.x	ray_inv.y	ra
Type	_local_ float	_local_ float	_loc
0	-1.4444908	-1.7955524	-2.17
1	-1.44425	-1.7967783	-2.17
2	-1.4440092	-1.7980076	-2.17
3	-1.4437686	-1.7992405	-2.17
4	-1.4435281	-1.800477	-2.17
5	-1.4432876	-1.8017174	-2.17
6	-1.4430474	-1.8029615	-2.17
7	-1.4428074	-1.8042094	-2.16
8	-1.4425675	-1.8054608	-2.16
9	-1.4423278	-1.8067161	-2.16
10	-1.4420878	-1.8079749	-2.16
11	-1.4418485	-1.8092378	-2.16
12	-1.4416089	-1.8105046	-2.16
13	-1.4413697	-1.8117749	-2.16
14	-1.4411306	-1.8130492	-2.16
15	-1.4408917	-1.8143274	-2.15
16	-1.4406527	-1.8156093	-2.15
17	-1.4404141	-1.8168953	-2.15
18	-1.4401754	-1.818185	-2.15
19	-1.439937	-1.8194786	-2.15
20	-1.4396988	-1.820776	-2.15
21	-1.4394605	-1.8220775	-2.15
22	-1.4392225	-1.8233831	-2.14
23	-1.4389844	-1.8246926	-2.14
24	-1.4387469	-1.8260059	-2.14
25	-1.4385092	-1.8273233	-2.14
26	-1.4382718	-1.828645	-2.14
27	-1.4380344	-1.8299706	-2.14
28	-1.4377974	-1.8313001	-2.14
29	-1.4375603	-1.832634	-2.13
30	-1.4373236	-1.8339716	-2.13
31	-1.4370868	-1.8353136	-2.13

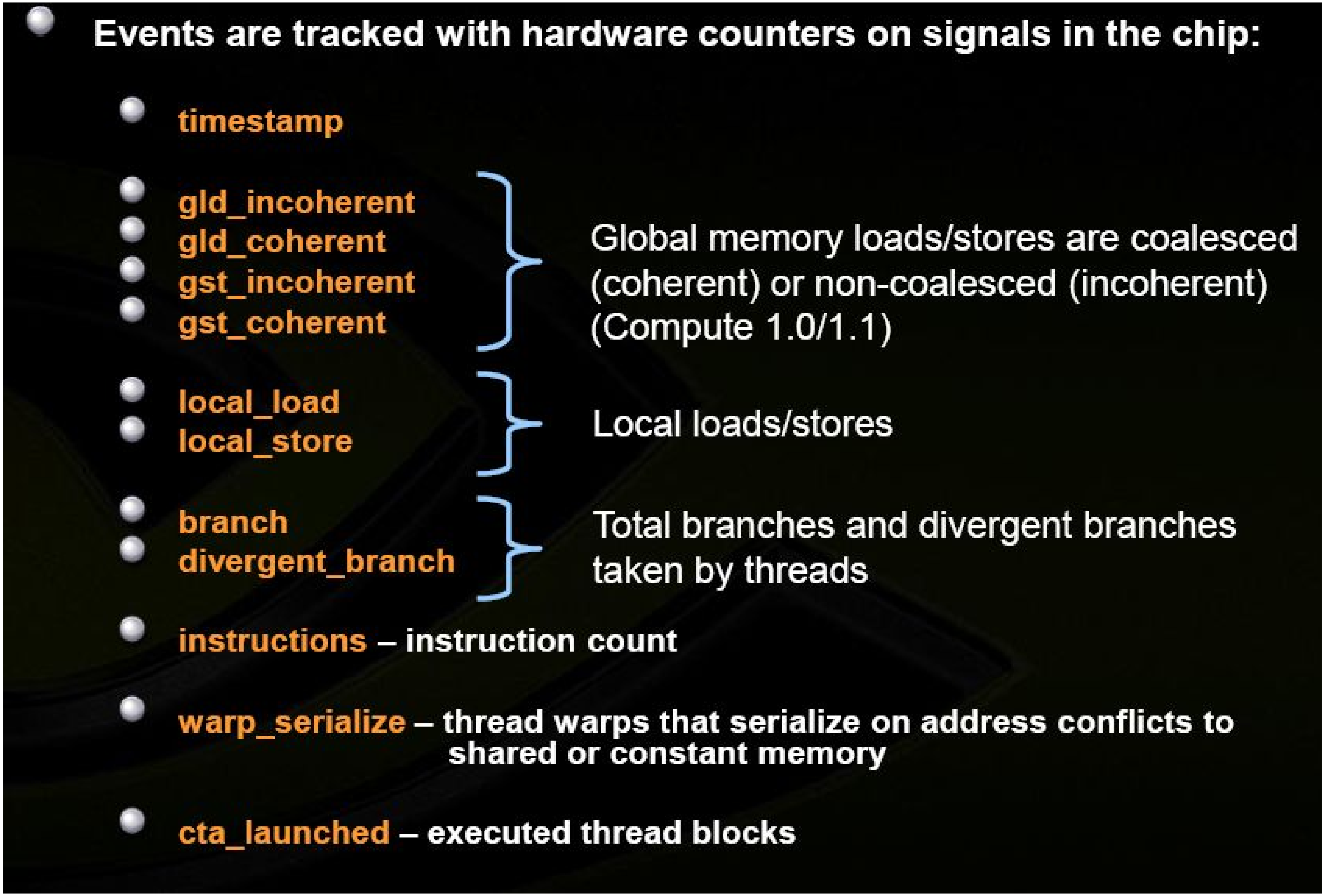
CUDA Development Tools:
Visual Profiler

CUDA Visual Profiler



	Timestamp	Method	GPU Time	CPU Time	Occupancy	gld_incoherent	gld_coherent	gst_incoherent	gst_coherent
1	98401	memcopy	3.296						
2	98615	memcopy	2.752						
3	98837	memcopy	2.88						
4	99132	c2c_mradix_r2	6.88	238	0.333	28	2	56	16
5	99721	memcopy	2.88						
6	99600	c2c_mradix_r7	11.36	229	0.125	24	4	48	32
7	100568	memcopy	2.752						
8	100687	c2c_mradix_r2	6.528						
9	101256	memcopy	2.752						
10	101376	c2c_mradix_r7	11.328						
11	101904	ComplexPoint...	2.816						
12	102398	memcopy	2.752						
13	102515	c2c_mradix_r2	6.208						
14	[illegible]	memcopy	2.752						

CUDA Visual Profiler



- Events are tracked with hardware counters on signals in the chip:
  - timestamp
  - gld_incoherent
  - gld_coherent
  - gst_incoherent
  - gst_coherent

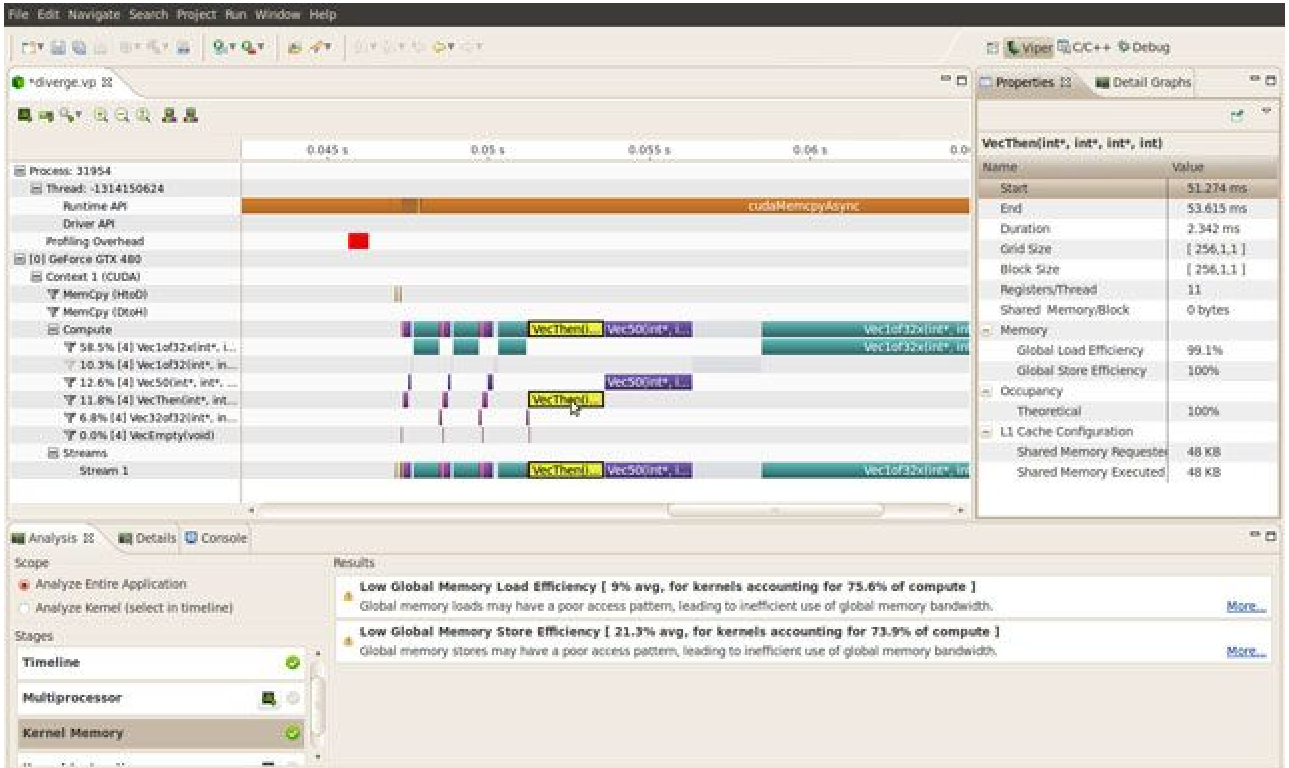
    Global memory loads/stores are coalesced (coherent) or non-coalesced (incoherent) (Compute 1.0/1.1)
  - local_load
  - local_store

    Local loads/stores
  - branch
  - divergent_branch

    Total branches and divergent branches taken by threads
  - instructions – instruction count
  - warp_serialize – thread warps that serialize on address conflicts to shared or constant memory
  - cta_launched – executed thread blocks

Nsight - Profile

- Easily identify performance bottlenecks using a unified CPU and GPU trace of application activity
- In-session kernel replay mode for more accurate profiling
- Profile frames and automatically measure GPU bottlenecks
- Visualizing concurrency of execution



VecThen(int*, int*, int*, int)

Name	Value
Start	51.274 ms
End	53.615 ms
Duration	2.342 ms
Grid Size	[256,1,1]
Block Size	[256,1,1]
Registers/Thread	11
Shared Memory/Block	0 bytes
Memory	
Global Load Efficiency	99.1%
Global Store Efficiency	100%
Occupancy	
Theoretical	100%
L1 Cache Configuration	
Shared Memory Requeste	48 KB
Shared Memory Executed	48 KB

Scope

● Analyze Entire Application

○ Analyze Kernel (select in timeline)

Stages

Results

Low Global Memory Load Efficiency [9% avg, for kernels accounting for 75.6% of compute]
Global memory loads may have a poor access pattern, leading to inefficient use of global memory bandwidth.

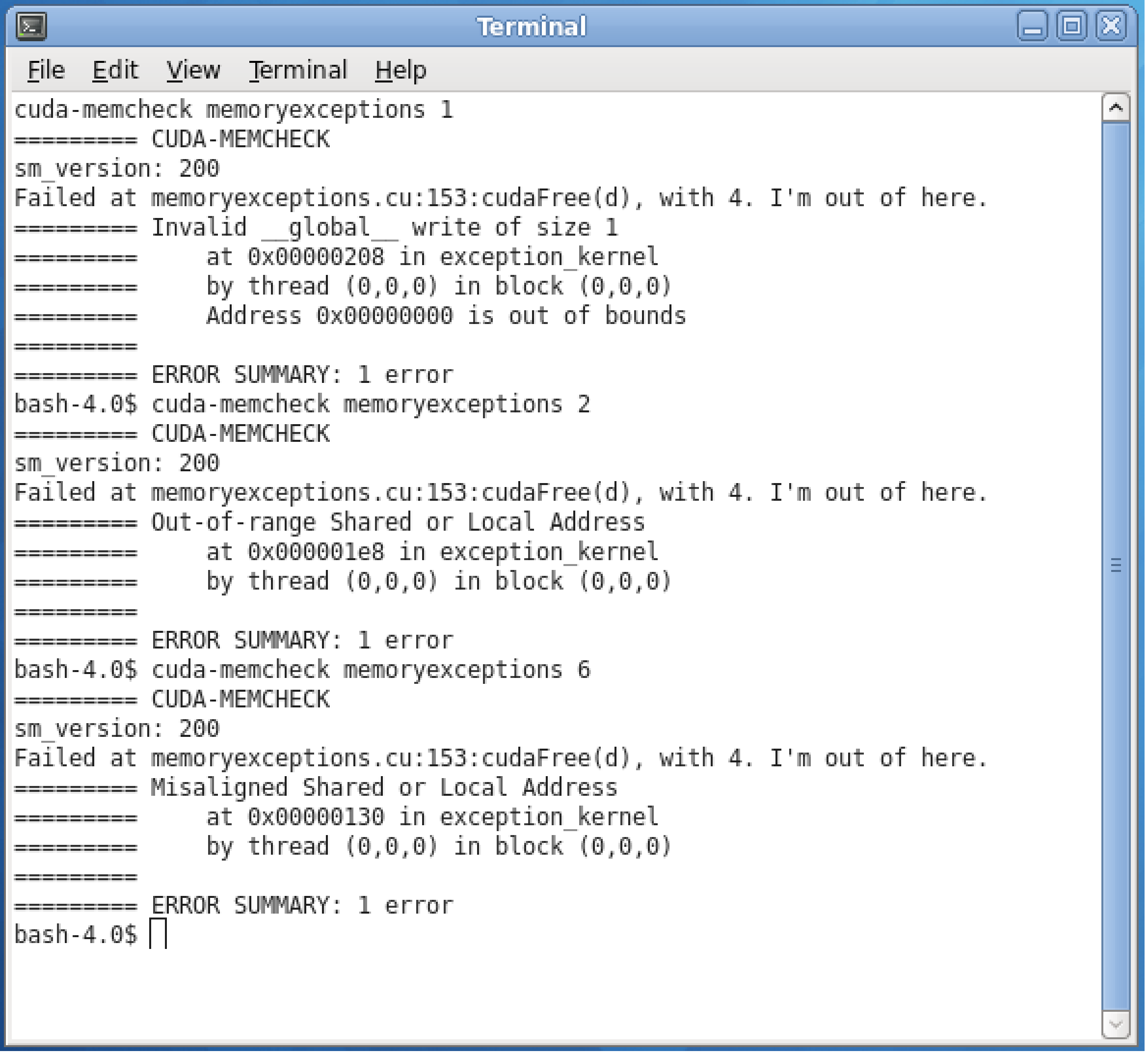
Low Global Memory Store Efficiency [21.3% avg, for kernels accounting for 73.9% of compute]
Global memory stores may have a poor access pattern, leading to inefficient use of global memory bandwidth.

CUDA Development Tools: MemCheck

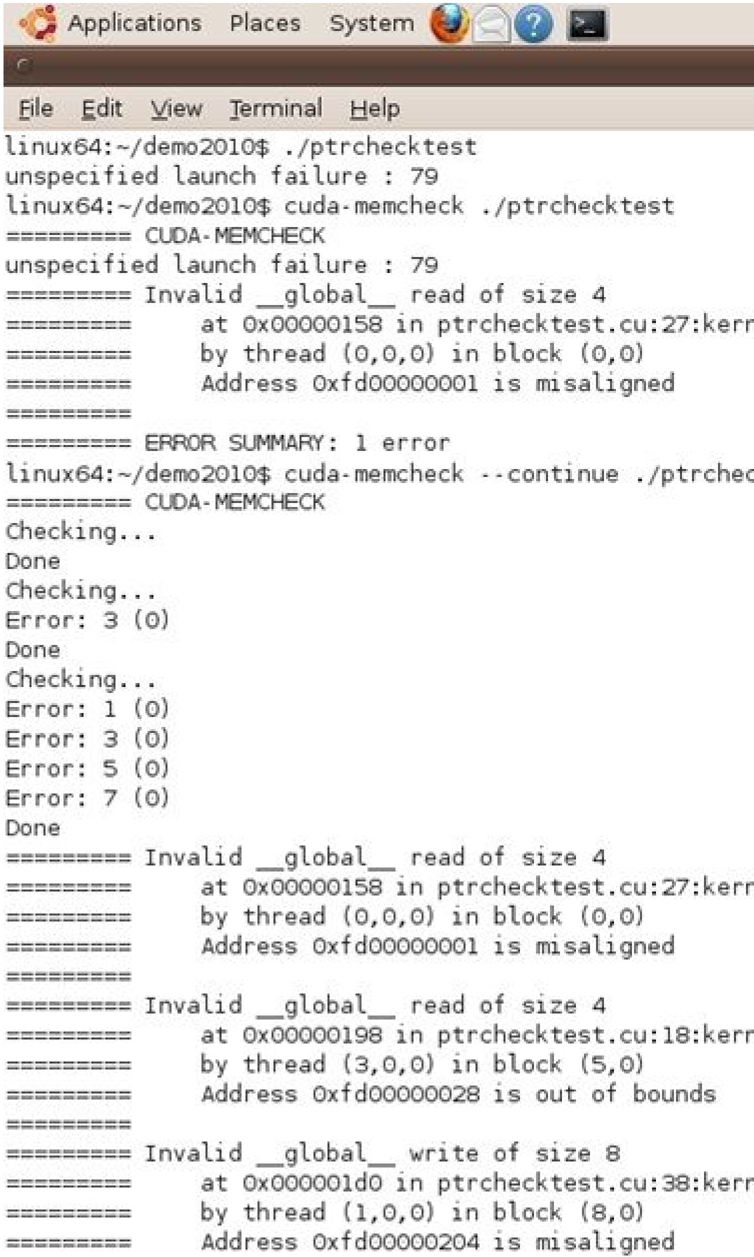
CUDA-MemCheck

- Coming with CUDA 3.0 Release
- Track out of bounds and misaligned accesses
- Supports CUDA C
- Integrated into the CUDA-GDB debugger
- Available as standalone tool on all OS platforms.

© NVIDIA Corporation 2009



```
cuda-memcheck memoryexceptions 1
========= CUDA-MEMCHECK
sm_version: 200
Failed at memoryexceptions.cu:153:cudaFree(d), with 4. I'm out of here.
========= Invalid __global__ write of size 1
=========     at 0x00000208 in exception_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========     Address 0x00000000 is out of bounds
=========
========= ERROR SUMMARY: 1 error
bash-4.0$ cuda-memcheck memoryexceptions 2
========= CUDA-MEMCHECK
sm_version: 200
Failed at memoryexceptions.cu:153:cudaFree(d), with 4. I'm out of here.
========= Out-of-range Shared or Local Address
=========     at 0x000001e8 in exception_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========
========= ERROR SUMMARY: 1 error
bash-4.0$ cuda-memcheck memoryexceptions 6
========= CUDA-MEMCHECK
sm_version: 200
Failed at memoryexceptions.cu:153:cudaFree(d), with 4. I'm out of here.
========= Misaligned Shared or Local Address
=========     at 0x00000130 in exception_kernel
=========     by thread (0,0,0) in block (0,0,0)
=========
========= ERROR SUMMARY: 1 error
bash-4.0$
```



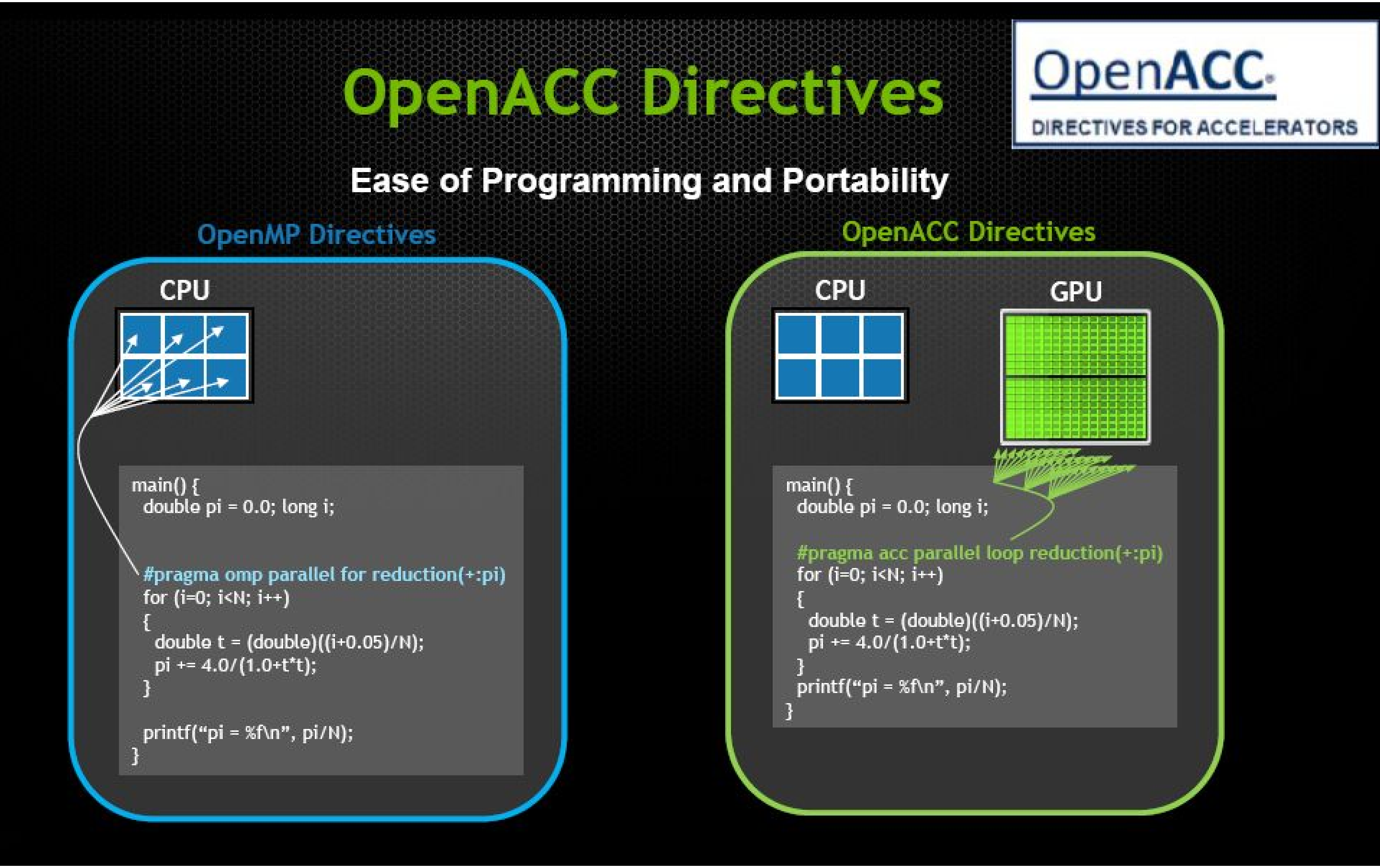
```
linux64:~/demo2010$ ./ptrchecktest
unspecified launch failure : 79
linux64:~/demo2010$ cuda-memcheck ./ptrchecktest
========= CUDA-MEMCHECK
unspecified launch failure : 79
========= Invalid __global__ read of size 4
=========     at 0x00000158 in ptrchecktest.cu:27:kerr
=========     by thread (0,0,0) in block (0,0)
=========     Address 0xfd00000001 is misaligned
=========
========= ERROR SUMMARY: 1 error
linux64:~/demo2010$ cuda-memcheck --continue ./ptrchec
========= CUDA-MEMCHECK
Checking...
Done
Checking...
Error: 3 (0)
Done
Checking...
Error: 1 (0)
Error: 3 (0)
Error: 5 (0)
Error: 7 (0)
Done
========= Invalid __global__ read of size 4
=========     at 0x00000158 in ptrchecktest.cu:27:kerr
=========     by thread (0,0,0) in block (0,0)
=========     Address 0xfd00000001 is misaligned
=========
========= Invalid __global__ read of size 4
=========     at 0x00000198 in ptrchecktest.cu:18:kerr
=========     by thread (3,0,0) in block (5,0)
=========     Address 0xfd00000028 is out of bounds
=========
========= Invalid __global__ write of size 8
=========     at 0x000001d0 in ptrchecktest.cu:38:kerr
=========     by thread (1,0,0) in block (8,0)
=========     Address 0xfd00000204 is misaligned
```

GPU programming –
CUDA –
OpenACC standard for directives

OpenACC - Overview

- New standard for parallel computing developed by compiler makers (2012) - http://www.openacc-standard.org/
- OpenACC works somewhat like OpenMP
- Goal is to provide simple directives to the compiler which enable it to accelerate the application on the GPU
- The tool is aimed at developers aiming to quickly speed up their code without extensive recoding in CUDA
- As tool is very new and this course focuses on CUDA, only a brief demo of OpenACC follows

OpenACC - Principle



OpenACC Directives

OpenACC DIRECTIVES FOR ACCELERATORS

Ease of Programming and Portability

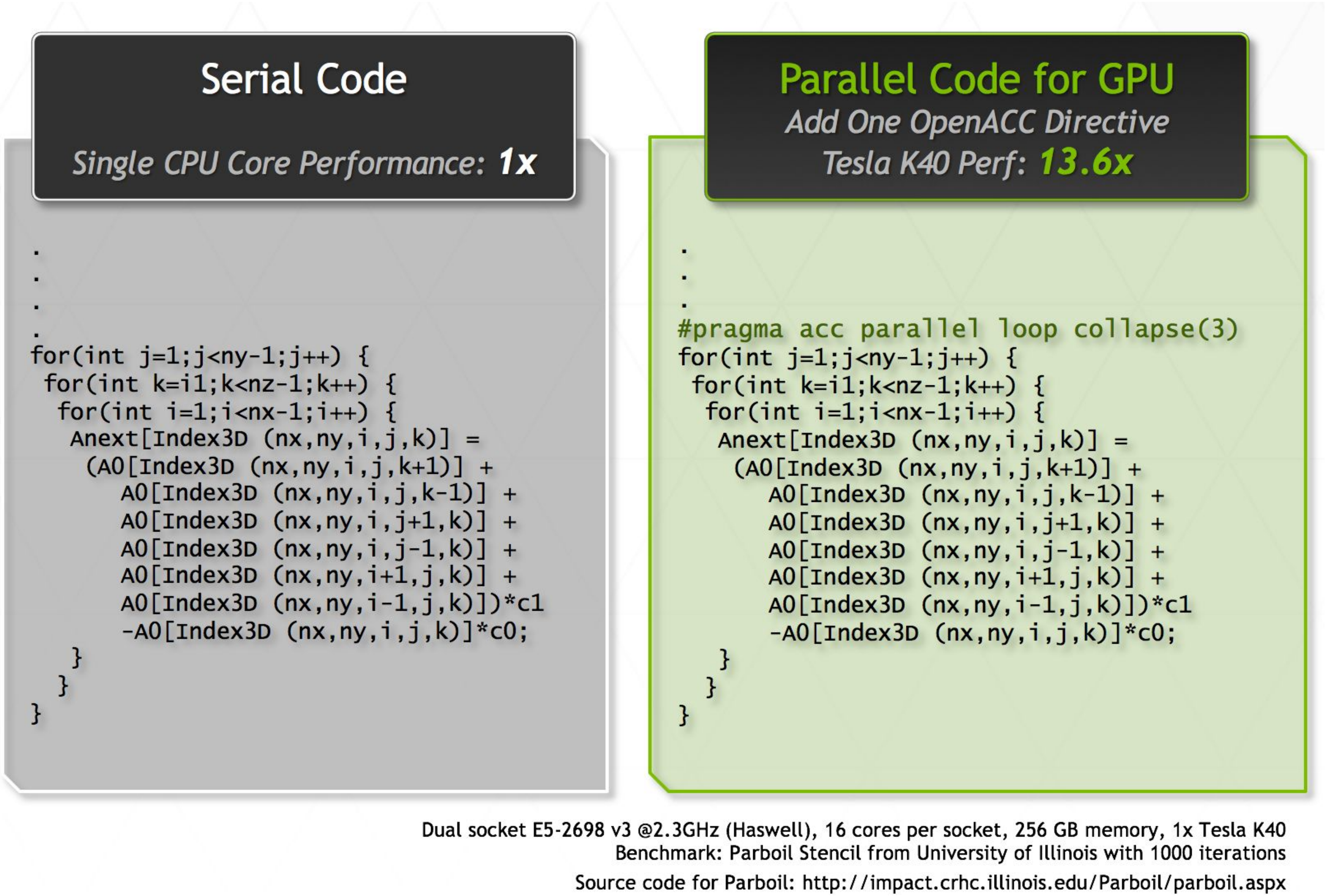
OpenMP Directives

```
main() {
  double pi = 0.0; long i;


  #pragma omp parallel for reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }

  printf(“pi = %f\n”, pi/N);
}
```

OpenACC Directives

```
main() {
  double pi = 0.0; long i;

  #pragma acc parallel loop reduction(+:pi)
  for (i=0; i<N; i++)
  {
    double t = (double)((i+0.05)/N);
    pi += 4.0/(1.0+t*t);
  }
  printf(“pi = %f\n”, pi/N);
}
```

OpenACC – Efficiency



Serial Code

Single CPU Core Performance: ***1x***

```
.
.
.
.
for(int j=1;j<ny-1;j++) {
 for(int k=i1;k<nz-1;k++) {
  for(int i=1;i<nx-1;i++) {
   Anext[Index3D (nx,ny,i,j,k)] =
    (A0[Index3D (nx,ny,i,j,k+1)] +
       A0[Index3D (nx,ny,i,j,k-1)] +
       A0[Index3D (nx,ny,i,j+1,k)] +
       A0[Index3D (nx,ny,i,j-1,k)] +
       A0[Index3D (nx,ny,i+1,j,k)] +
       A0[Index3D (nx,ny,i-1,j,k)])*c1
       -A0[Index3D (nx,ny,i,j,k)]*c0;
   }
  }
}
```

Parallel Code for GPU

Add One OpenACC Directive

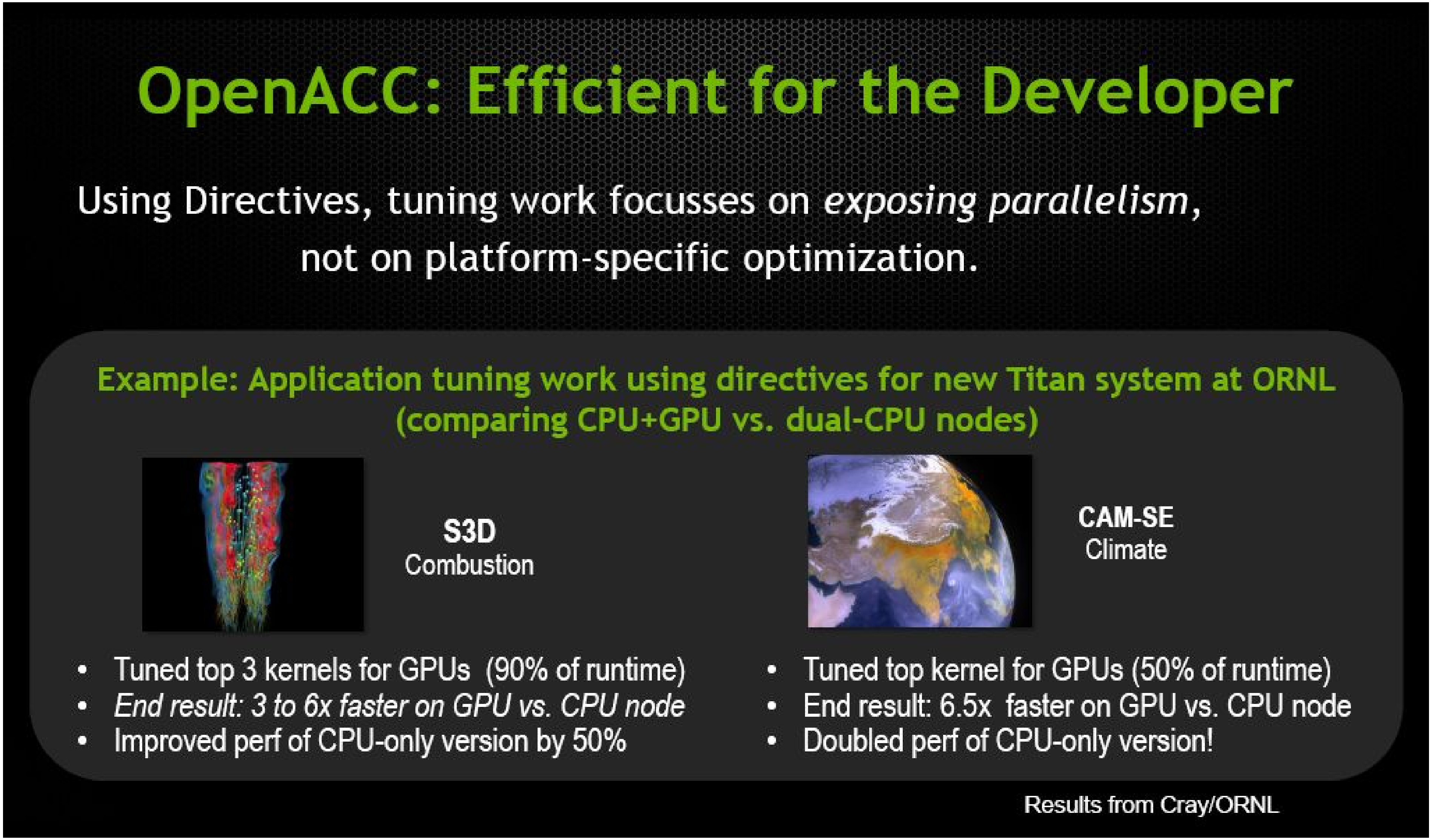
Tesla K40 Perf: ***13.6x***

```
.
.
.
#pragma acc parallel loop collapse(3)
for(int j=1;j<ny-1;j++) {
 for(int k=i1;k<nz-1;k++) {
  for(int i=1;i<nx-1;i++) {
   Anext[Index3D (nx,ny,i,j,k)] =
    (A0[Index3D (nx,ny,i,j,k+1)] +
       A0[Index3D (nx,ny,i,j,k-1)] +
       A0[Index3D (nx,ny,i,j+1,k)] +
       A0[Index3D (nx,ny,i,j-1,k)] +
       A0[Index3D (nx,ny,i+1,j,k)] +
       A0[Index3D (nx,ny,i-1,j,k)])*c1
       -A0[Index3D (nx,ny,i,j,k)]*c0;
   }
  }
}
```

Dual socket E5-2698 v3 @2.3GHz (Haswell), 16 cores per socket, 256 GB memory, 1x Tesla K40
Benchmark: Parboil Stencil from University of Illinois with 1000 iterations

Source code for Parboil: http://impact.crhc.illinois.edu/Parboil/parboil.aspx

OpenACC - Efficiency



OpenACC: Efficient for the Developer

Using Directives, tuning work focusses on *exposing parallelism*, not on platform-specific optimization.

Example: Application tuning work using directives for new Titan system at ORNL (comparing CPU+GPU vs. dual-CPU nodes)

S3D
Combustion

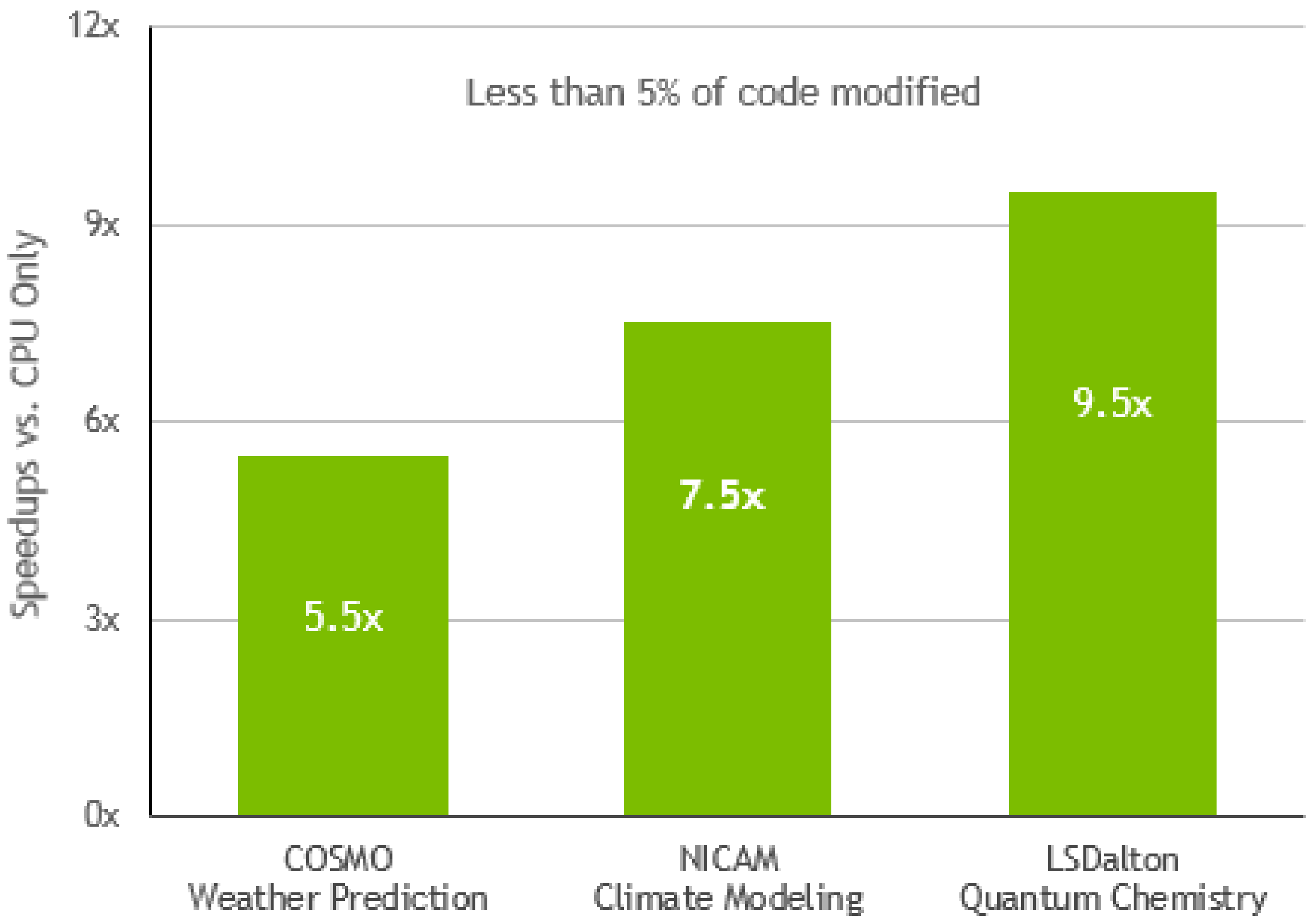
- Tuned top 3 kernels for GPUs (90% of runtime)
- *End result: 3 to 6x faster on GPU vs. CPU node*
- Improved perf of CPU-only version by 50%

CAM-SE
Climate

- Tuned top kernel for GPUs (50% of runtime)
- End result: 6.5x faster on GPU vs. CPU node
- Doubled perf of CPU-only version!

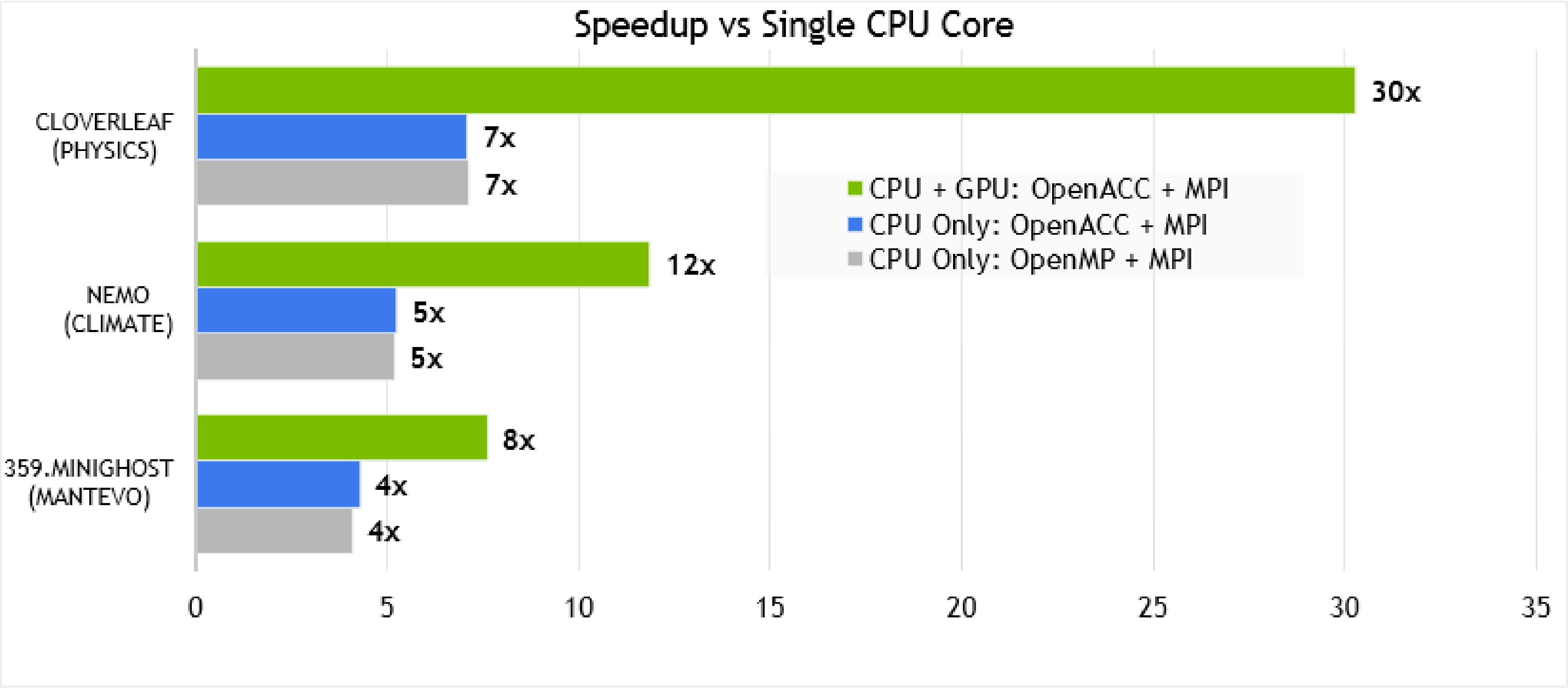
Results from Cray/ORNL

OpenACC - Efficiency



LS-DALTON: Benchmark on Oak Ridge Titan Supercomputer, AMD CPU vs Tesla K20X GPU. Test input: Alanine-3 on CCSD(T) module.
NICAM: Benchmark on TiTech TSUBAME 2.5, Westmere CPU vs. K20X.

OpenACC - Efficiency



359.miniGhost: CPU: Intel Xeon E5-2698 v3, 2 sockets, 32-cores total, GPU: Tesla K80 (single GPU)
NEMO: Each socket CPU: Intel Xeon E5--2698 v3, 16 cores; GPU: NVIDIA K80 both GPUs
CLOVERLEAF: CPU: Dual socket Intel Xeon CPU E5-2690 v2, 20 cores total, GPU: Tesla K80 both GPUs