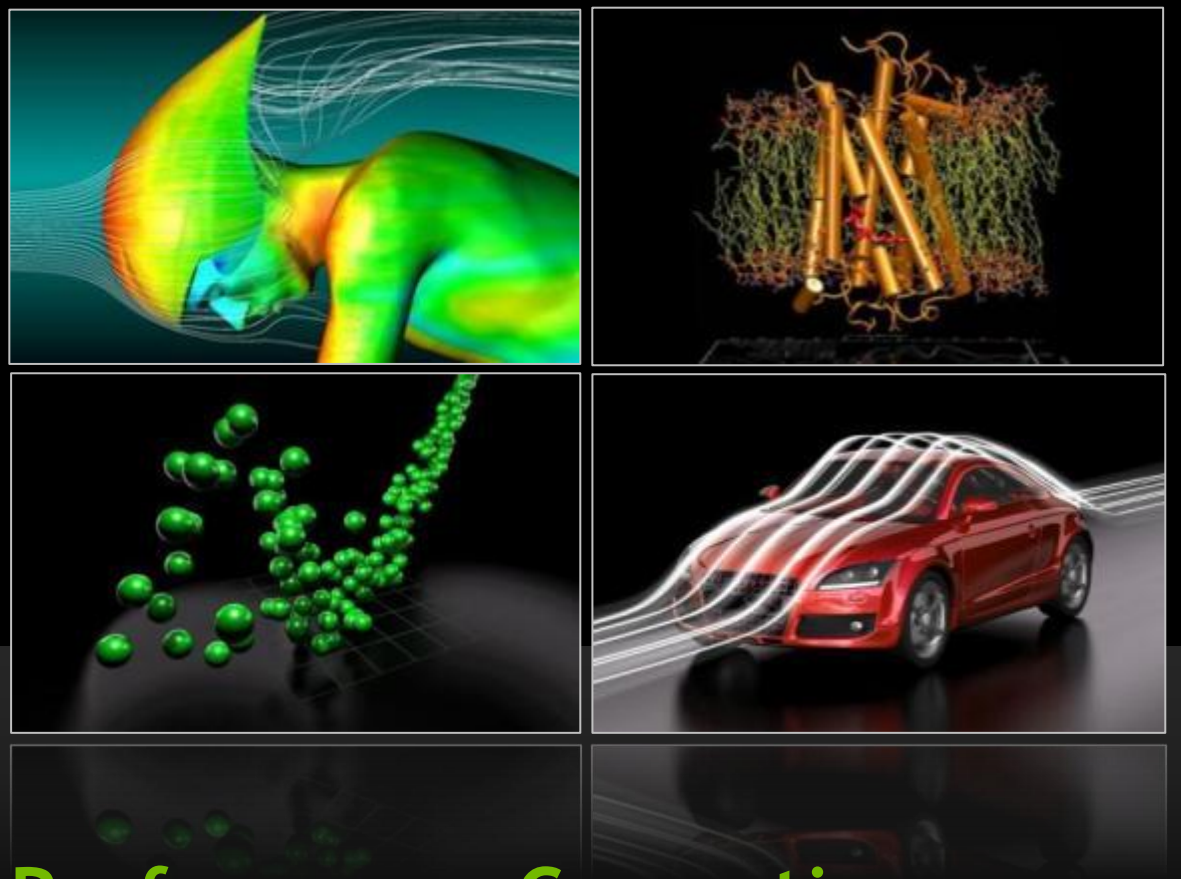
TESLA

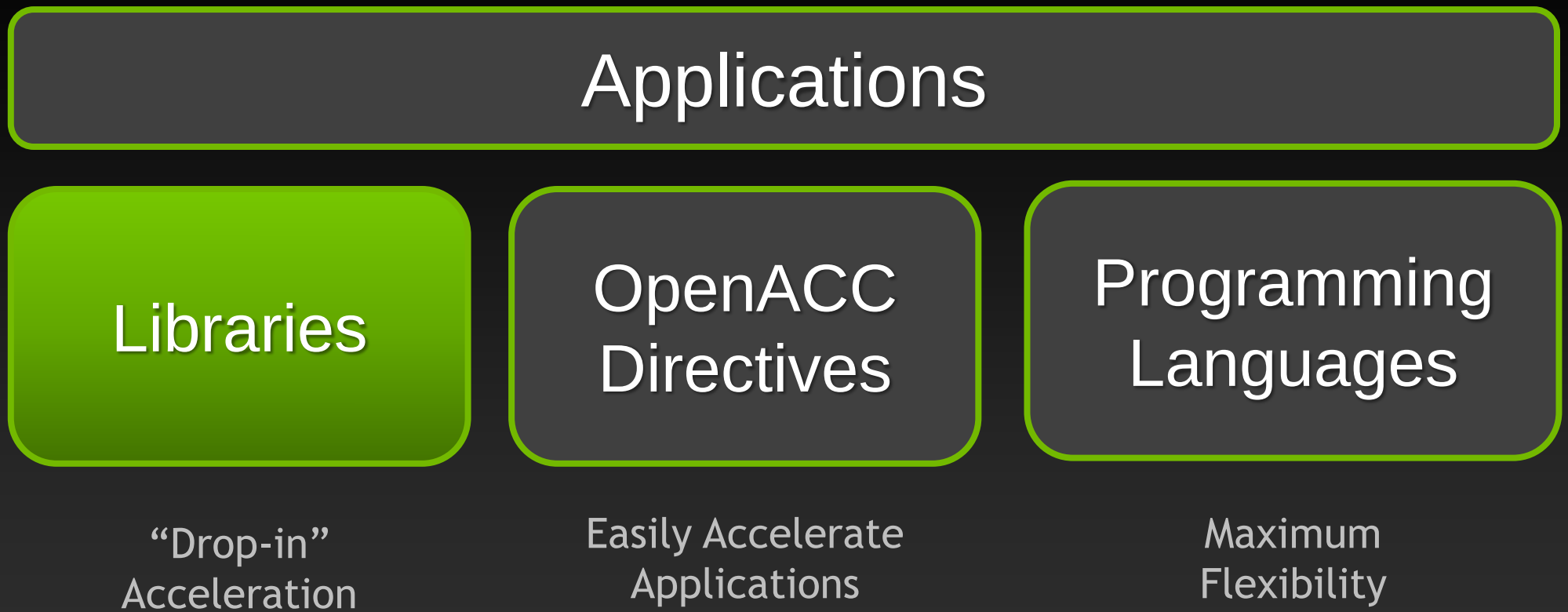
GPU Computing



Accelerating High Performance Computing

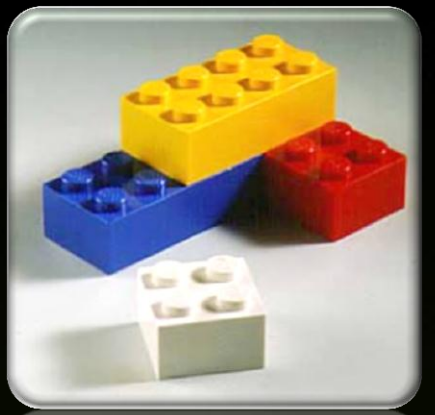
http://www.nvidia.com/tesla

3 Ways to Accelerate Applications



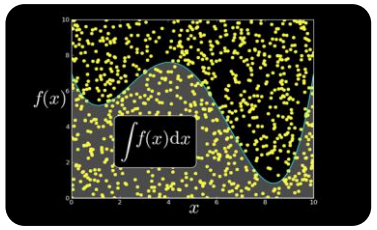
NVIDIA CUDA Library Approach

- Provide basic building blocks
- Make them easy to use
- Make them fast

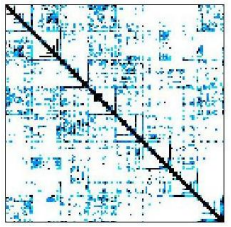


- Provides a quick path to GPU acceleration
- Enables developers to focus on their “secret sauce”
- Ideal for applications that use CPU libraries

NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE

NVIDIA NPP

GPU VSIPL

Vector Signal
Image Processing

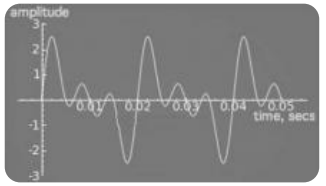
CULA tools

GPU Accelerated
Linear Algebra

MAGMA

ICL UT

Matrix Algebra on
GPU and Multicore



NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

CUSP

Sparse Linear
Algebra

libjacket

Building-block
Algorithms for CUDA

Thrust

C++ STL Features
for CUDA

GPU Accelerated Libraries

"Drop-in" Acceleration for Your Applications

CUDA Math Libraries

High performance math routines for your applications:

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

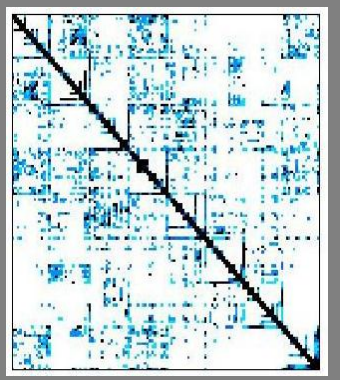
Included in the CUDA Toolkit **Free download @ www.nvidia.com/getcuda**

Linear Algebra

Dense

cuBLAS

Sparse



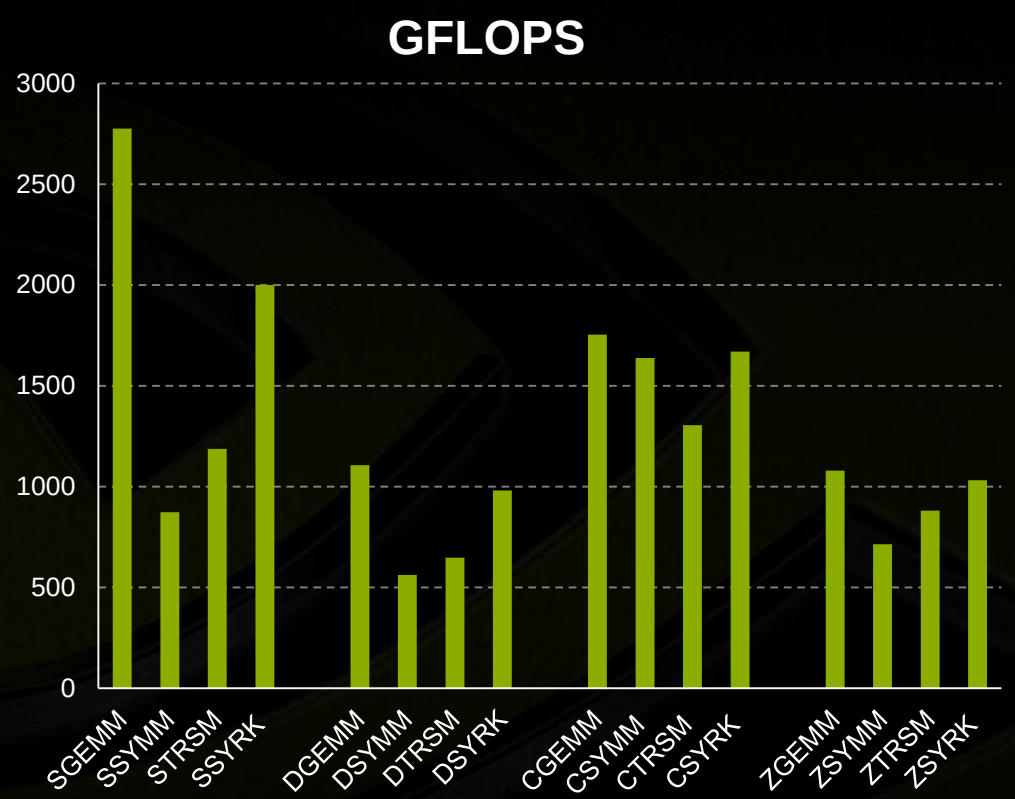
cuSPARSE

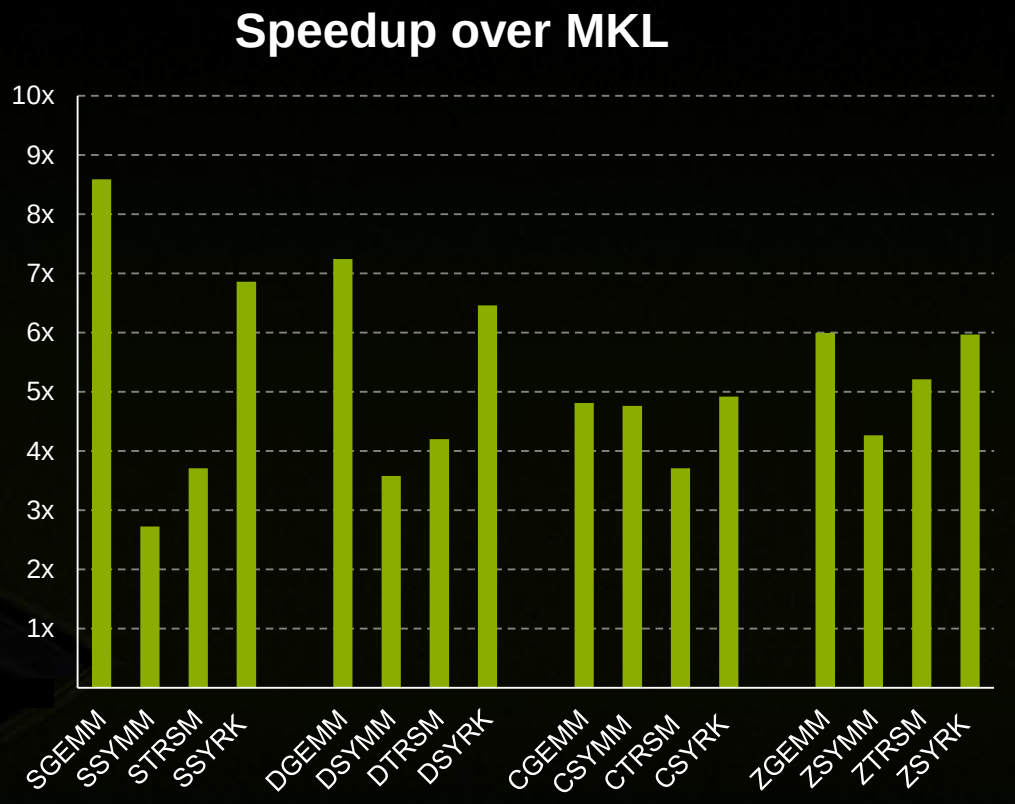
cuBLAS Interface

```
err = cublasIdamax(hdl, n, col, 1, size);

err = cublasDscal(hdl, n, val, row, 1);

cublas_dgemm('N','N',m,n,k,1.0,A,m,
      B,k,0.0,C,m)
```

```
err = idamax(n, col, 1, size);

err = dscal(n, val, row, 1);

dgemm('N','N',m,n,k,1.0,A,m,
      B,k,0.0,C,m)
```

cuBLAS Level 3: >1 TFLOPS double-precision





- MKL 10.3.6 on Intel SandyBridge E5-2687W @3.10GHz
- CUBLAS 5.0.30 on K20X, input and output data on device

GPU-Callable Libraries

New in CUDA 5.0

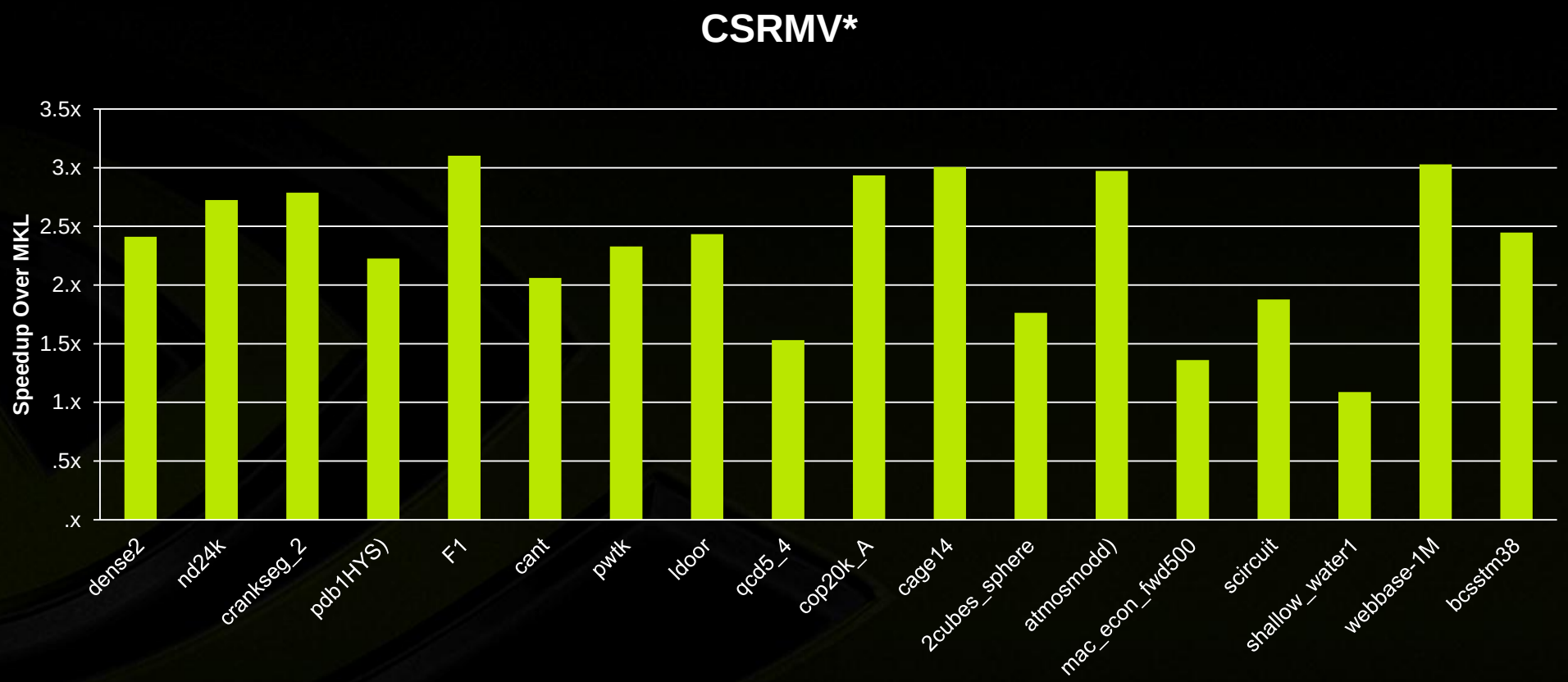
- **Call cuBLAS library function from GPU code**
- **Supported on K20 and K20X only**
- **Encourages third party libraries**

cuSPARSE Interface

```
err = cusparseDcsrmv(hdl, transa, m, k,
                     nnz, alpha, desrc,
                     val, indx, col,
                     x, beta, y);
```

```
mkl_dcsrmv(transa, m, k,
           alpha, descr,
           val, indx, pntrb, pntre,
           x, beta, y);
```

cuSPARSE Performance



* Average of single, double, complex and double complex

- MKL 10.3.6 on Intel SandyBridge E5-2687W @3.10GHz
- CUBLAS 5.0.30 on K20X, input and output data on device

Different Approaches to Linear Algebra

- CULA tools (dense, sparse)
  - LAPACK based API
  - Solvers, Factorizations, Least Squares, SVD, Eigensolvers
  - Sparse: Krylov solvers, Preconditioners, support for various formats

```
culaSgetrf(M, N, A, LDA, IPIV, INFO)
```

EM Photonics

- ArrayFire
  - Array container object
  - Solvers, Factorizations, SVD, Eigensolvers

```
array out = lu(A)
```

AccelerEyes

Different Approaches to Linear Algebra (cont.)

- MAGMA
  - LAPACK conforming API
  - Magma BLAS and LAPACK
  - High performance by utilizing both GPU and CPU

```
magma_sgetrf(M, N, A, LDA, IPIV, INFO)
```

ICL

- LibFlame
  - LAPACK compatibility interface
  - Infrastructure for rapid linear algebra algorithm development

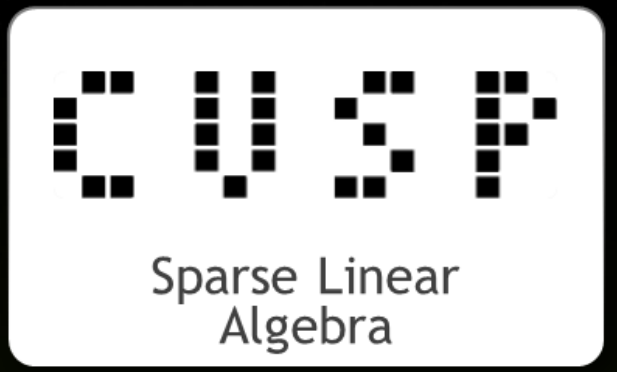
```
FLASH_LU_piv(A, p)
```

UT-Austin

Different Approaches to Linear Algebra (cont.)

- CUSP
  - Sparse matrix operations
  - Open source
  - Supports COO, CSR, ELL, DIA, hybrid, etc.
  - Solvers, monitors, preconditioners, etc.

```
cusp::krylov::cg(A, x, b);
```



Toolkits are increasingly supporting GPUs

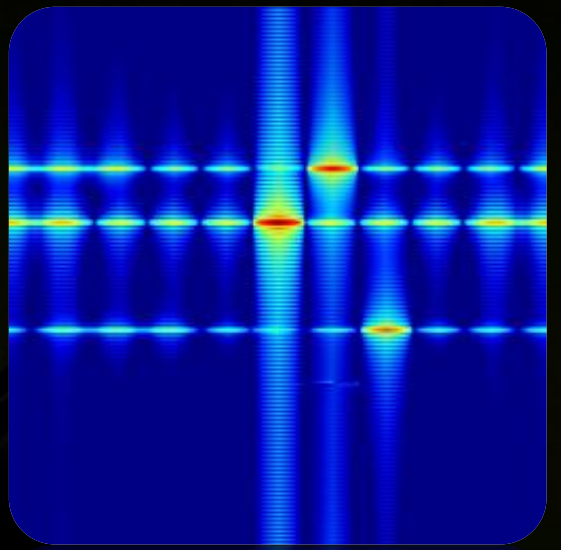
- PETSc
  - GPU support via extension to Vec and Mat classes
  - Partially dependent on CUSP
  - MPI parallel, GPU accelerated solvers

- Trilinos
  - GPU support in KOKKOS package
  - Used through vector class Tpetra
  - MPI parallel, GPU accelerated solvers

Common Tasks in Signal Processing

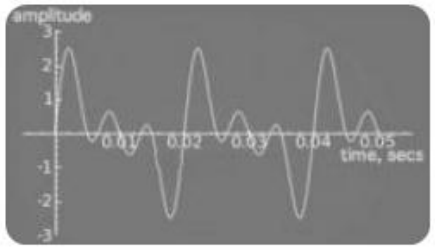
Filtering

Correlation



Segmentation

Libraries for GPU Accelerated Signal Processing



NVIDIA cuFFT

NVIDIA NPP

GPU VSIPL

Vector Signal Image Processing

MATLAB

Parallel Computing Toolbox

ArrayFire Matrix Computations

GPULib

GPU Accelerated Data Analysis

cuFFT

- Interface modeled after FFTW

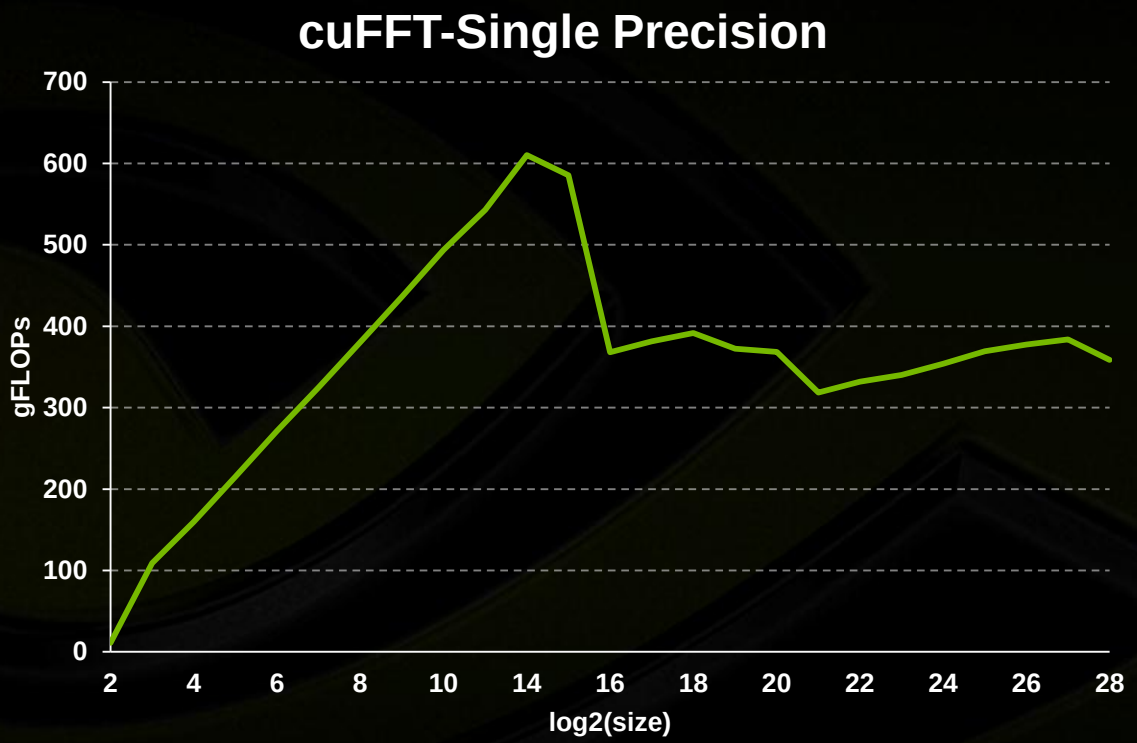
```
cufftPlan2d PlanA;

cufftCreatePlan(N, M, &PlanA,
CUFFT_C2C);

cufftExecC2C(PlanA, d_data, d_data,
             CUFFT_FORWARD);
```

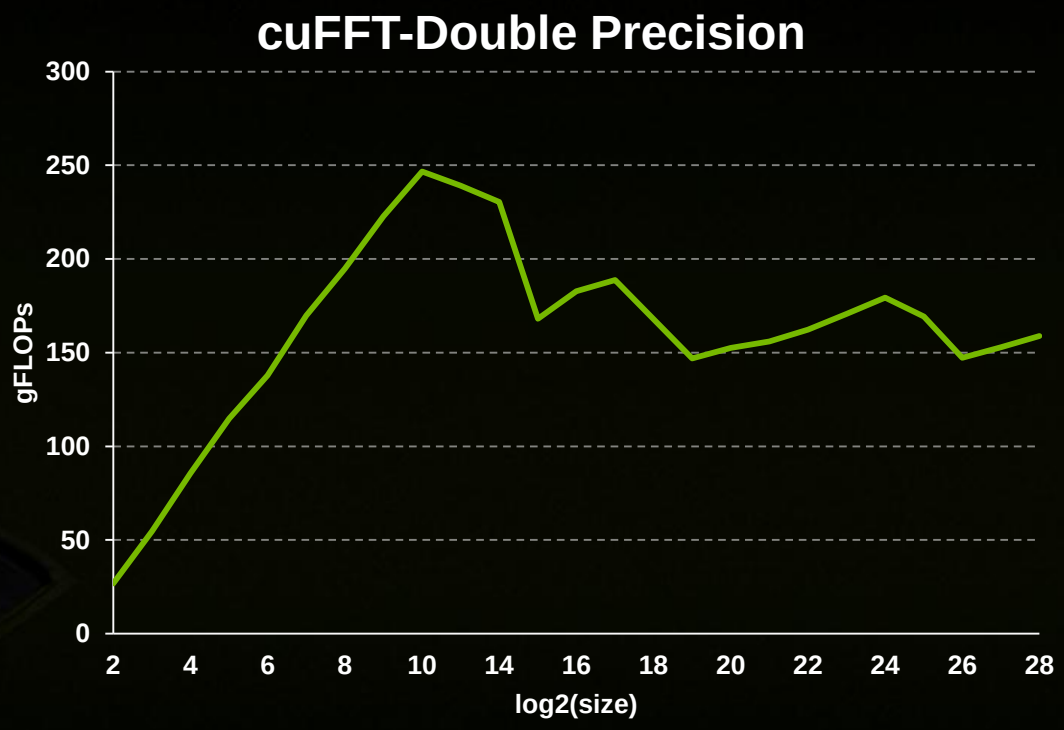
```
fftw_plan PlanA;

fftw_plan_dft_2d(N, M, &PlanA,
         data, data, FFT_FORWARD)

fftw_execute_dft(PlanA, data,
                 data);
```

- Supports streams and batching (2 and 3-D, too!) for better performance

CUFFT: up to 600 GFLOPS

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





- CUFFT 5.0.30 on K20X, input and output data on device

Basic concepts of NPP

- Collection of high-performance GPU processing
  - Non-linear data transforms (point-by-point mult, sqrt, etc.)
  - Support for multi-channel integer and float data

- C API => name disambiguates between data types, flavor

```
nppiAdd_32f_C1R (…)
```

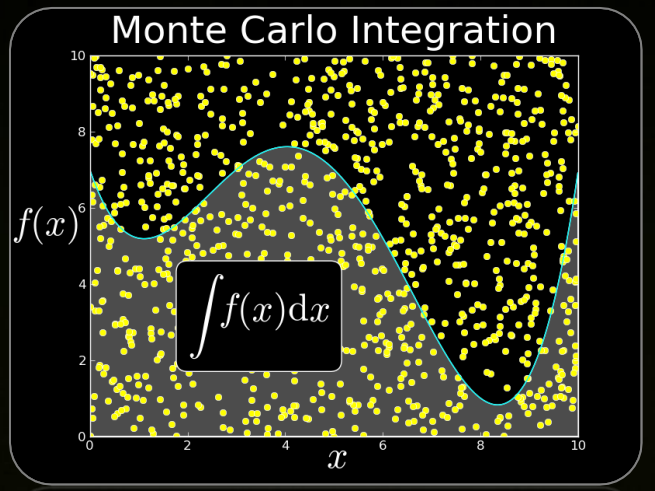
  - "Add" two single channel ("C1") 32-bit float ("32f") images, possibly masked by a region of interest ("R")

NPP features a large set of functions

- Arithmetic and Logical Operations
  - Point-by-point ops, clamp, threshold, etc.
- Geometric transformations
  - Rotate, Warp, Interpolate
- Compression
  - jpeg de/compression
- Image processing
  - Filter, histogram, statistics

Random Number Generation on GPU

- Generating high quality random numbers in parallel is hard
  - Don’t do it yourself, use a library!
- Large suite of generators and distributions
  - XORWOW, MRG323ka, MTGP32, (scrambled) Sobol
  - uniform, normal, log-normal
  - Single and double precision



- Two APIs for cuRAND
  - Called from CPU: Ideal when generating large batches of RNGs on GPU
  - Called from GPU: Ideal when RNGs need to be generated inside a kernel

cuRAND: Host vs Device API

- **CPU API**

```
#include "curand.h"
curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);
curandGenerateUniform(gen, d_data, n);
```

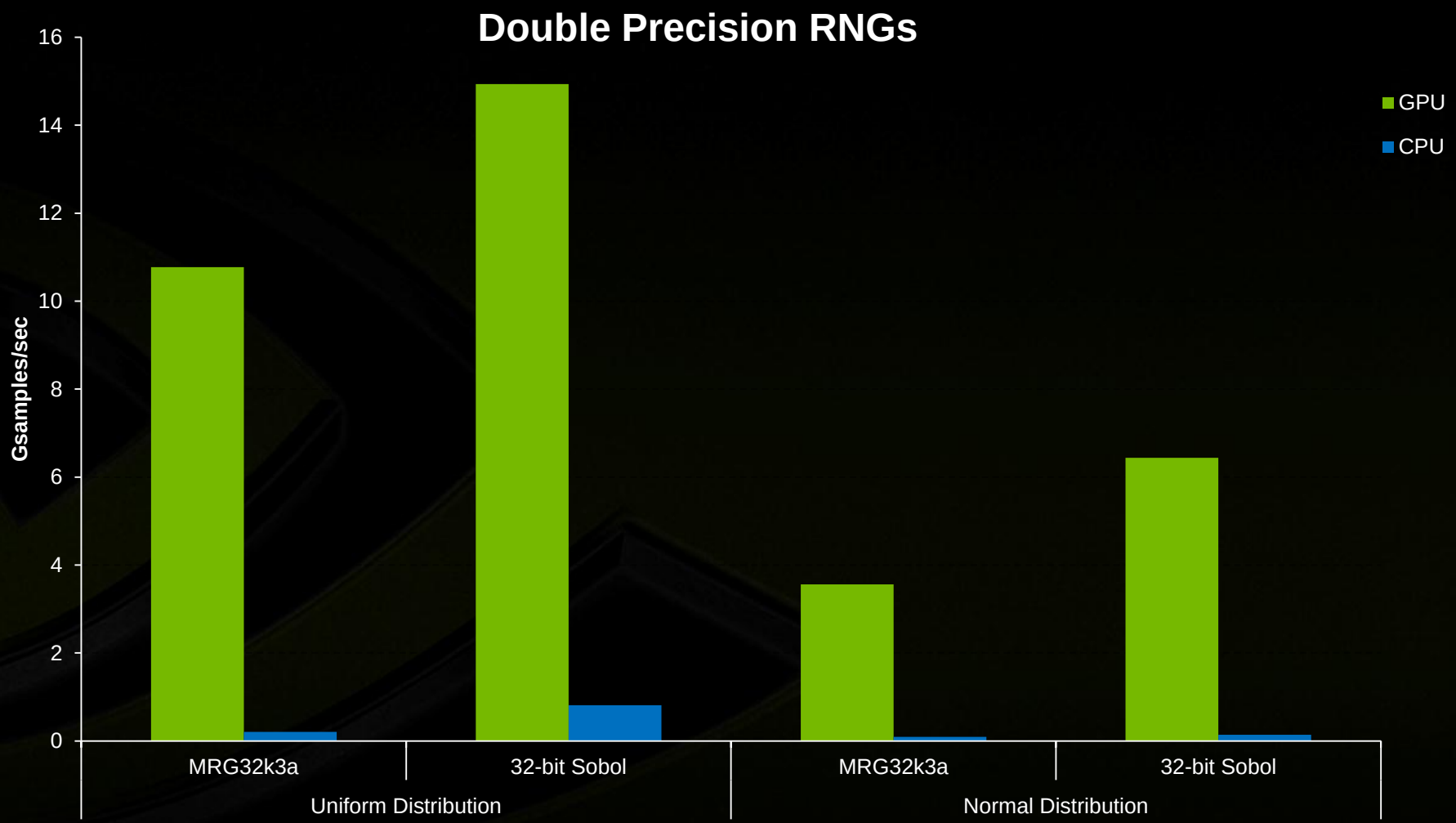
Generate set of random numbers at once

- **GPU API**

```
#include "curand_kernel.h"
__global__ void generate_kernel(curandState *state) {
        int id = threadIdx.x + blockIdx.x * 64;
        x = curand(&state[id]);
}
```

Generate random numbers per thread

cuRAND Performance



Thrust: STL-like CUDA Template Library

- **GPU(device) and CPU(host) vector class**

```
thrust::host_vector<float> H(10, 1.f);
thrust::device_vector<float> D = H;
```

- **Iterators**

```
thrust::fill(D.begin(), D.begin()+5, 42.f);
float* raw_ptr = thrust::raw_pointer_cast(D);
```

- **Algorithms**
  - **Sort, reduce, transformation, scan, ..**

```
thrust::transform(D1.begin(), D1.end(), D2.begin(), D2.end(),
 thrust::plus<float>());   // D2 = D1 + D2
```

CUDA libs with OpenACC


```
cufftExecPlan(plan, d_signal, d_signal)
…
#pragma acc data deviceptr(d_signal)
#pragma acc loop independent
for(i=0; i<n; i++) d_signal[i] = 2 * d_signal[i];
```

Expanded presentation @ GTC On Demand

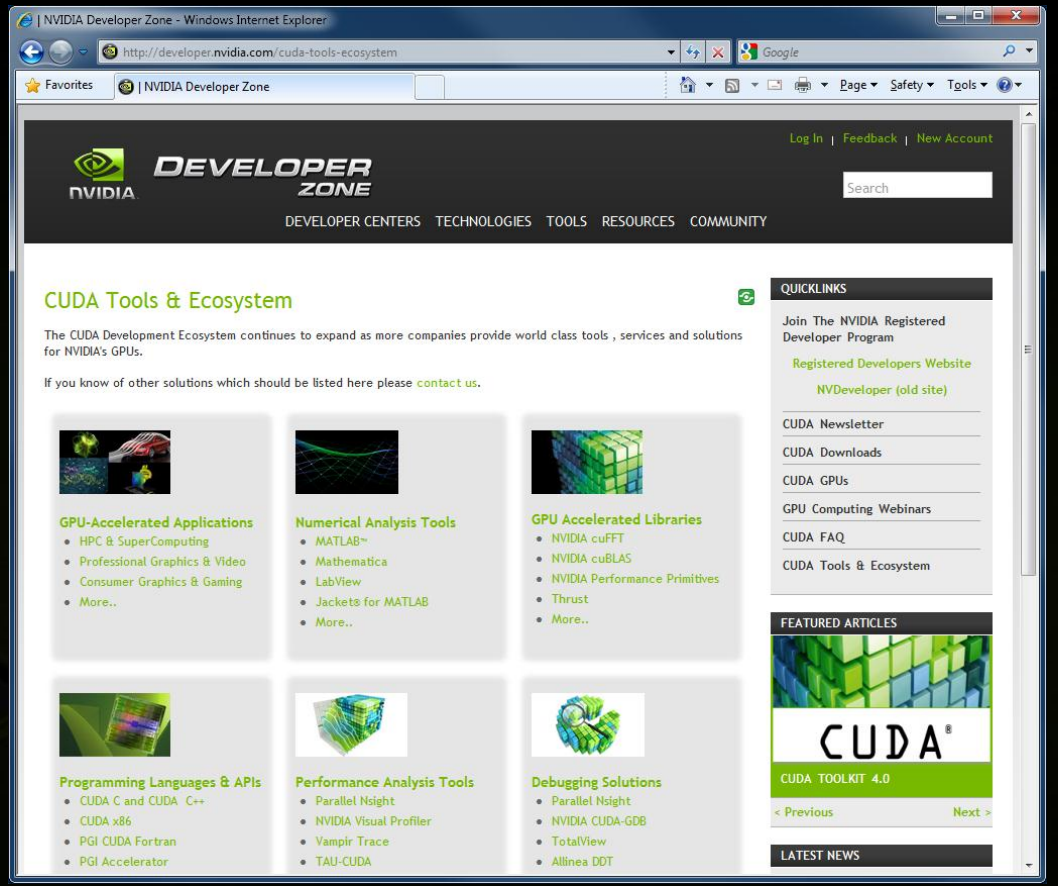
- CUDA Accelerated GPU Libraries, Peter Messmer S0629
  - nvidia.fullviewmedia.com/gtc2012/0514-A5-S0629.html
- More at www.gputechconf.com/gtcnew/on-demand-gtc.php

Explore the CUDA (Libraries) Ecosystem

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

  developer.nvidia.com/

- Get a taste of CUDA libs
  - Test drive **your app** on the new K20
- See the demos
  - CUDA/CUDA library experts on call

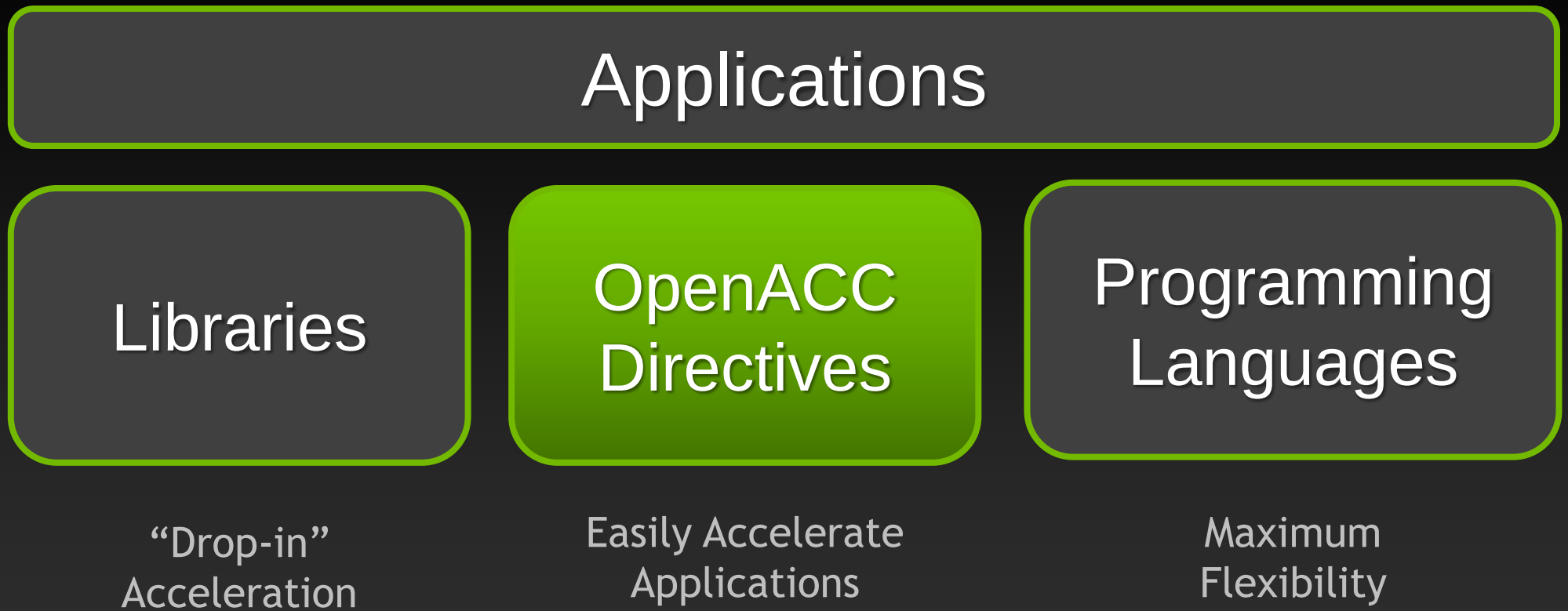


Summary

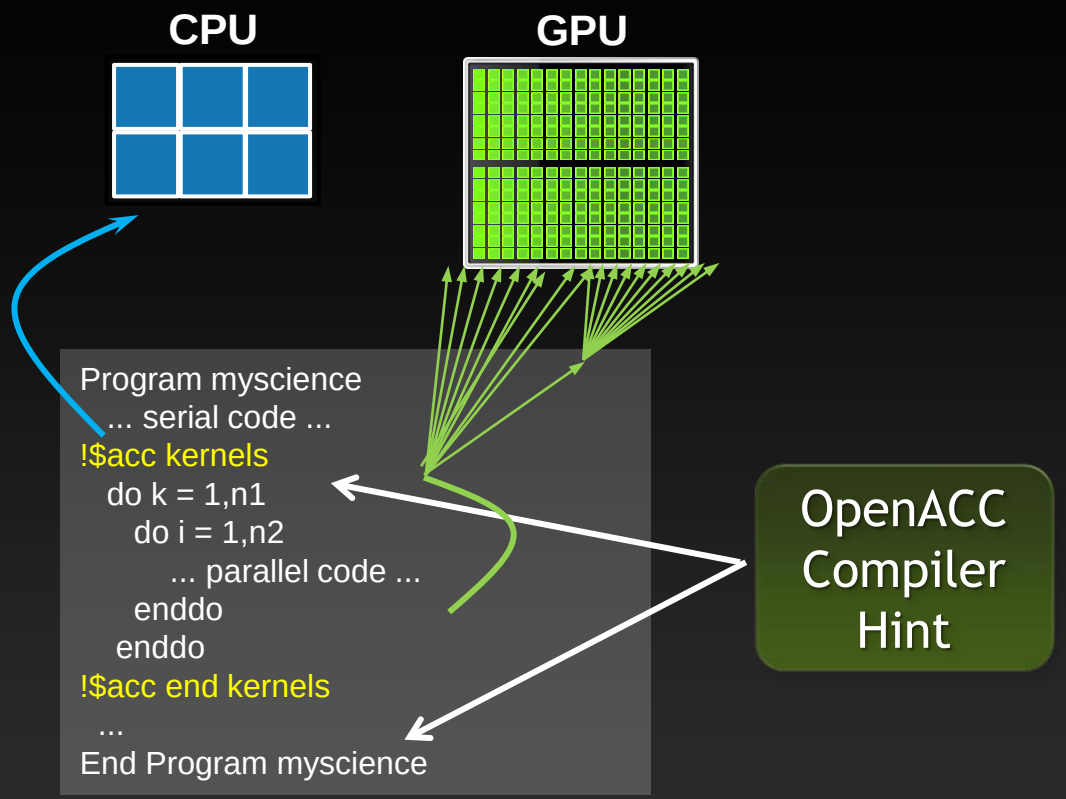
- CUDA libraries offer high performance for minimal effort
- Robust community of 3rd party libraries
- Familiar interfaces make porting legacy code easy (“drop-in”)
- Enables focus on core IP

OpenACC

3 Ways to Accelerate Applications



OpenACC Directives



Your original
Fortran or C code

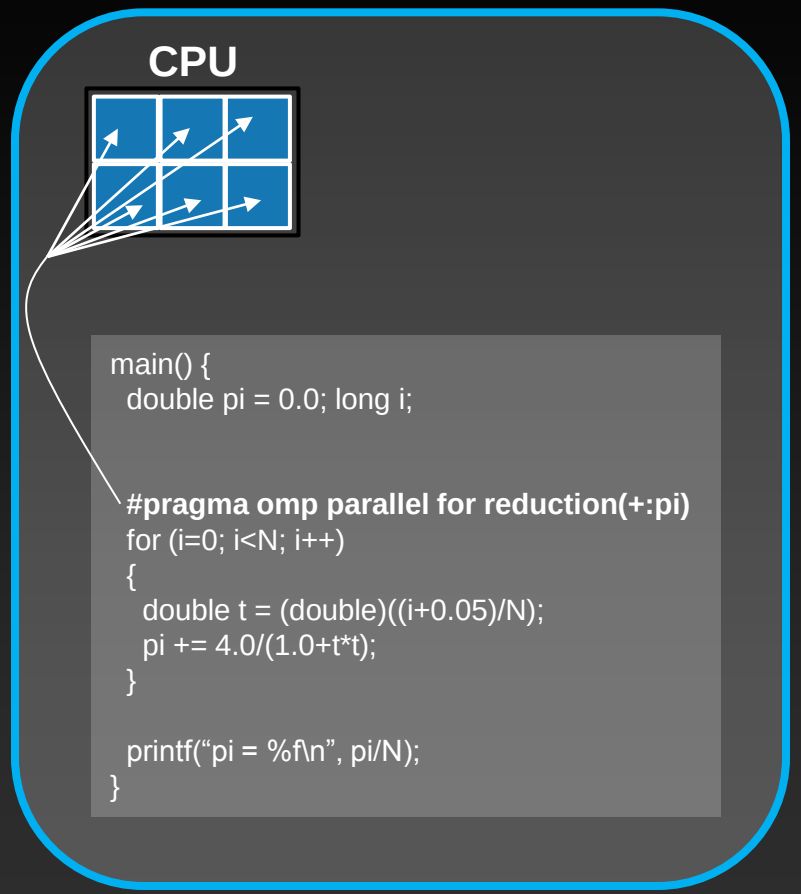
Simple Compiler hints

Compiler Parallelizes code

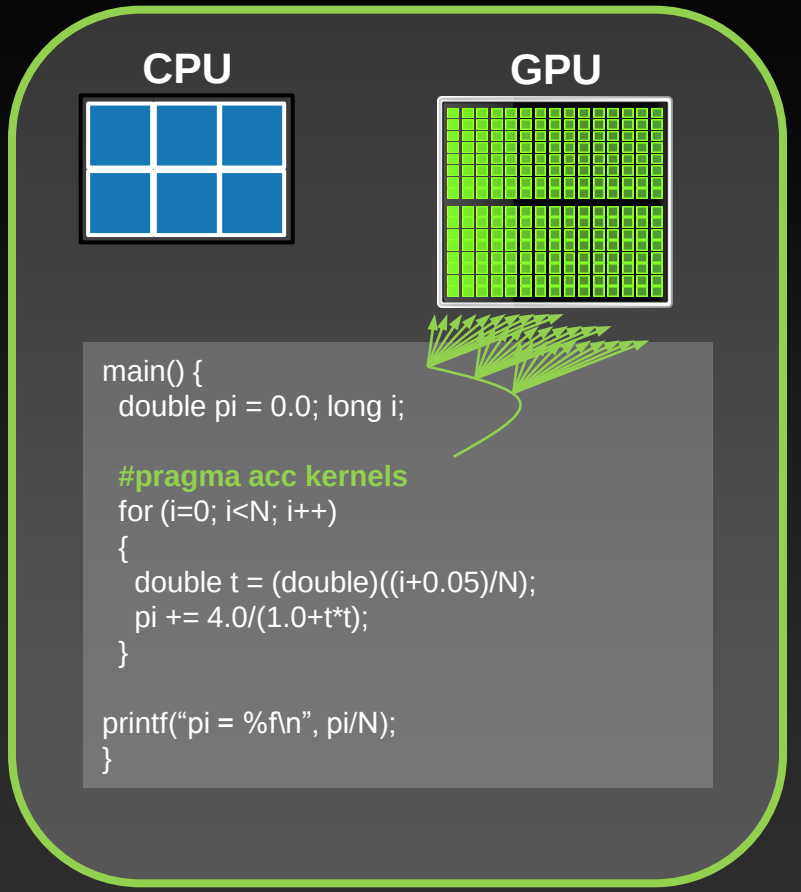
Works on many-core GPUs & multicore CPUs

Familiar to OpenMP Programmers

OpenMP



OpenACC



OpenACC

Open Programming Standard for Parallel Computing

"OpenACC will enable programmers to easily develop portable applications that maximize the performance and power efficiency benefits of the hybrid CPU/GPU architecture of Titan."

--Buddy Bland, Titan Project Director, Oak Ridge National Lab

"OpenACC is a technically impressive initiative brought together by members of the OpenMP Working Group on Accelerators, as well as many others. We look forward to releasing a version of this proposal in the next release of OpenMP."

--Michael Wong, CEO OpenMP Directives Board

OpenACC Standard

NVIDIA

CRAY THE SUPERCOMPUTER COMPANY

PGI

CAPS

OpenACC
The Standard for GPU Directives

- **Easy:** Directives are the easy path to accelerate compute intensive applications
- **Open:** OpenACC is an open GPU directives standard, making GPU programming straightforward and portable across parallel and multi-core processors
- **Powerful:** GPU Directives allow complete access to the massive parallel power of a GPU

High-level, with low-level access

- Compiler directives to specify parallel regions in C, C++, Fortran
  - OpenACC compilers offload parallel regions from host to accelerator
  - Portable across OSes, host CPUs, accelerators, and compilers
- Create high-level heterogeneous programs
  - Without explicit accelerator initialization,
  - Without explicit data or program transfers between host and accelerator
- Programming model allows programmers to start simple
  - Enhance with additional guidance for compiler on loop mappings, data location, and other performance details
- Compatible with other GPU languages and libraries
  - Interoperate between CUDA C/Fortran and GPU libraries
  - e.g. CUFFT, CUBLAS, CUSPARSE, etc.

Directives: Easy & Powerful

Real-Time Object Detection

Global Manufacturer of Navigation Systems

5x in 40 Hours

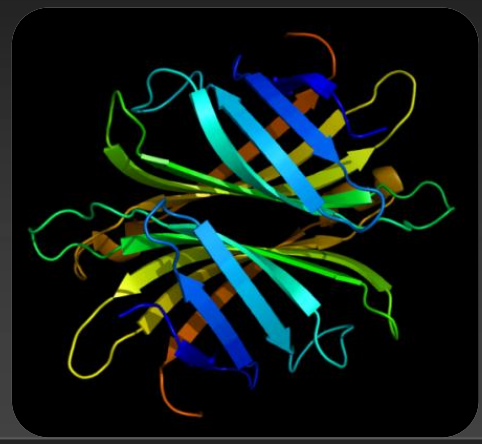
Valuation of Stock Portfolios using Monte Carlo

Global Technology Consulting Company

2x in 4 Hours

Interaction of Solvents and Biomolecules

University of Texas at San Antonio



5x in 8 Hours

> "Optimizing code with directives is quite easy, especially compared to CPU threads or writing CUDA kernels. The most important thing is avoiding restructuring of existing code for production applications."
>
> *-- Developer at the Global Manufacturer of Navigation Systems*

Small Effort. Real Impact.

Large Oil Company

3x in 7 days

Solving billions of equations iteratively for oil production at world's largest petroleum reservoirs

UNIVERSITY of HOUSTON
CULLEN COLLEGE of ENGINEERING

Univ. of Houston

Prof. M.A. Kayali

20x in 2 days

Studying magnetic systems for innovations in magnetic storage media and memory, field sensors, and biomagnetism

THE UNIVERSITY OF MELBOURNE

Uni. Of Melbourne

Prof. Kerry Black

65x in 2 days

Better understand complex reasons by lifecycles of snapper fish in Port Phillip Bay

УГАТУ

Ufa State Aviation

Prof. Arthur Yuldashev

7x in 4 Weeks

Generating stochastic geological models of oilfield reservoirs with borehole data

COMPUTING FOR SCIENCE fs

GAMESS-UK

Dr. Wilkinson, Prof. Naidoo

10x

Used for various fields such as investigating biofuel production and molecular sensors.

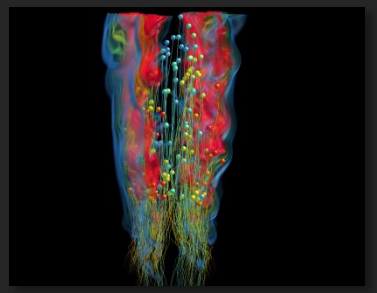
* Achieved using the PGI Accelerator Compiler

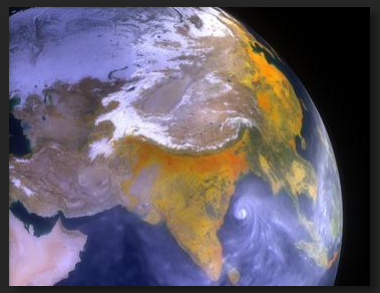
Focus on Exposing Parallelism

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

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

S3D
Research more efficient combustion with next-generation fuels





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

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

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

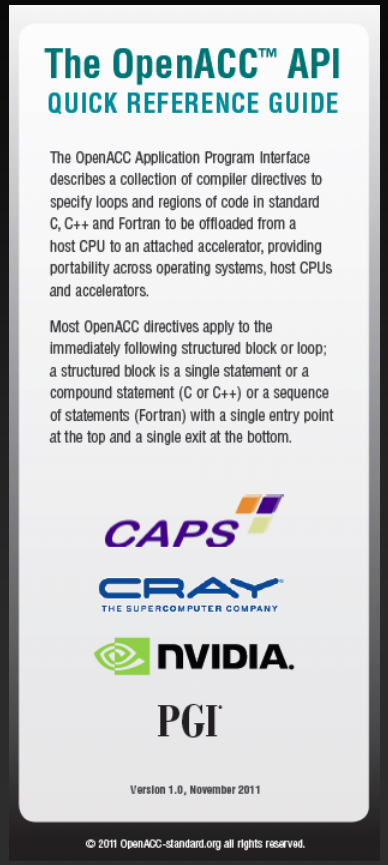
OpenACC Specification and Website

- Full OpenACC 1.0 Specification available online

  http://www.openacc-standard.org

- Quick reference card also available

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



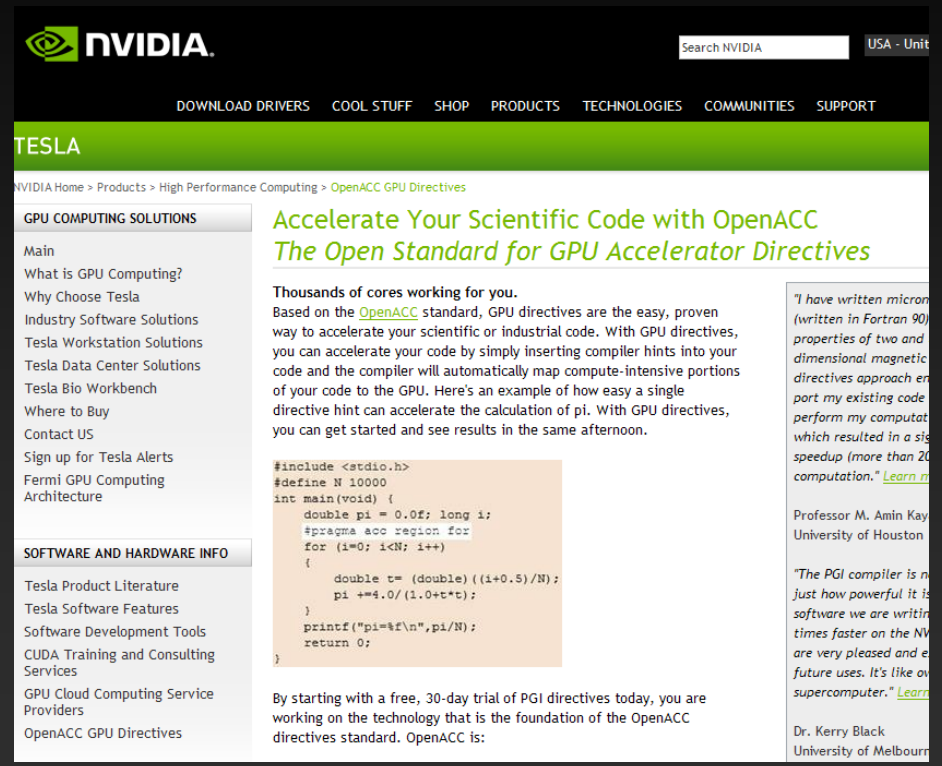
Start Now with OpenACC Directives

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®

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:

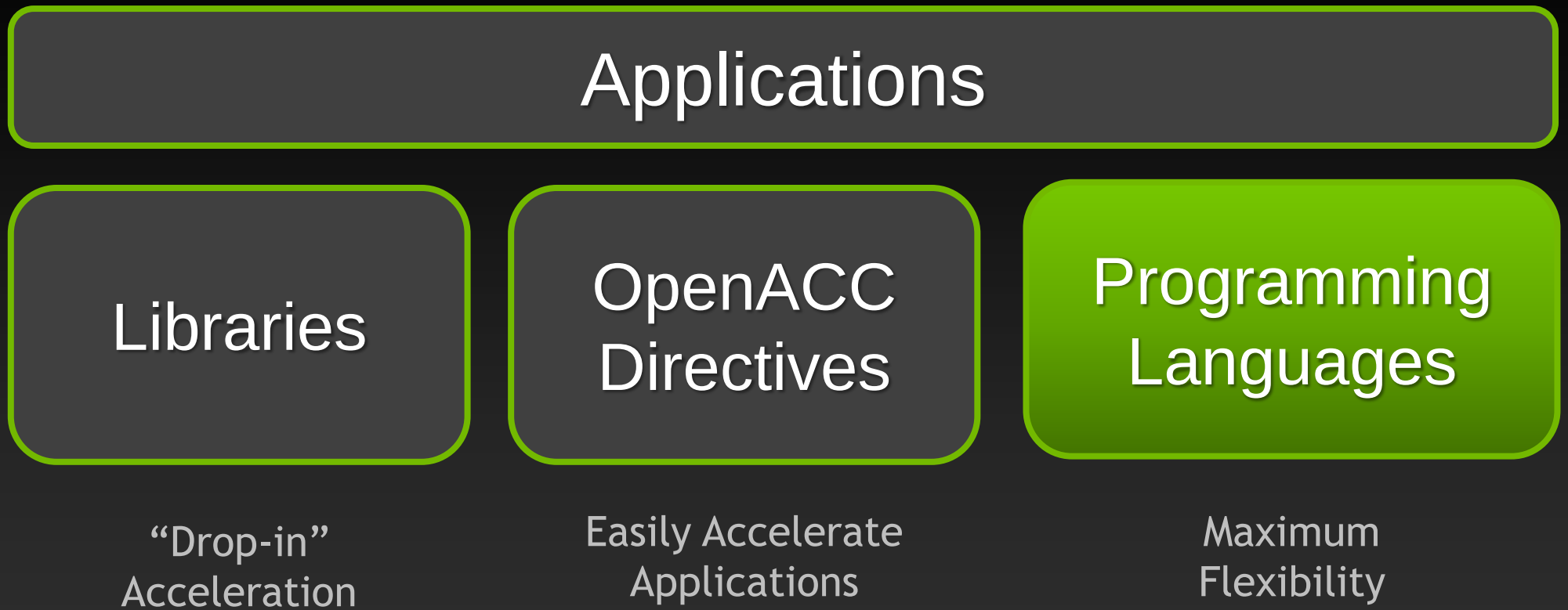
```
pgf90 -acc -ta=nvidia -Minfo=accel -o saxpy_acc saxpy.f90
```

- 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
```

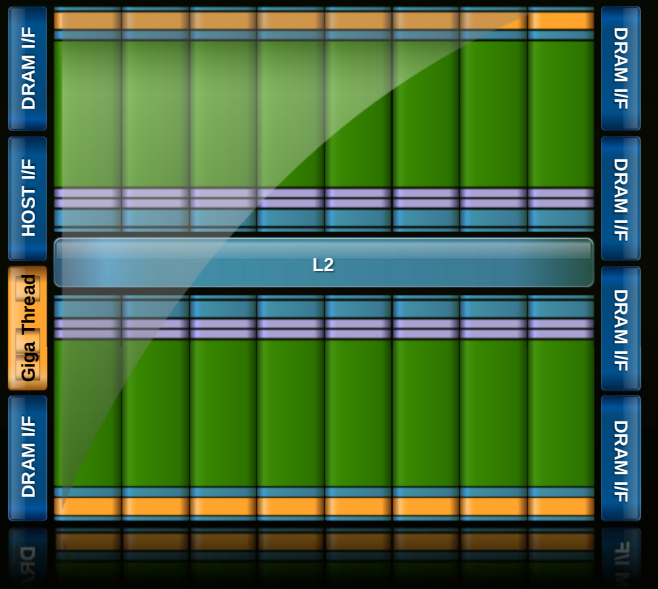

CUDA
NVIDIA

3 Ways to Accelerate Applications



GPU Architecture: Two Main Components

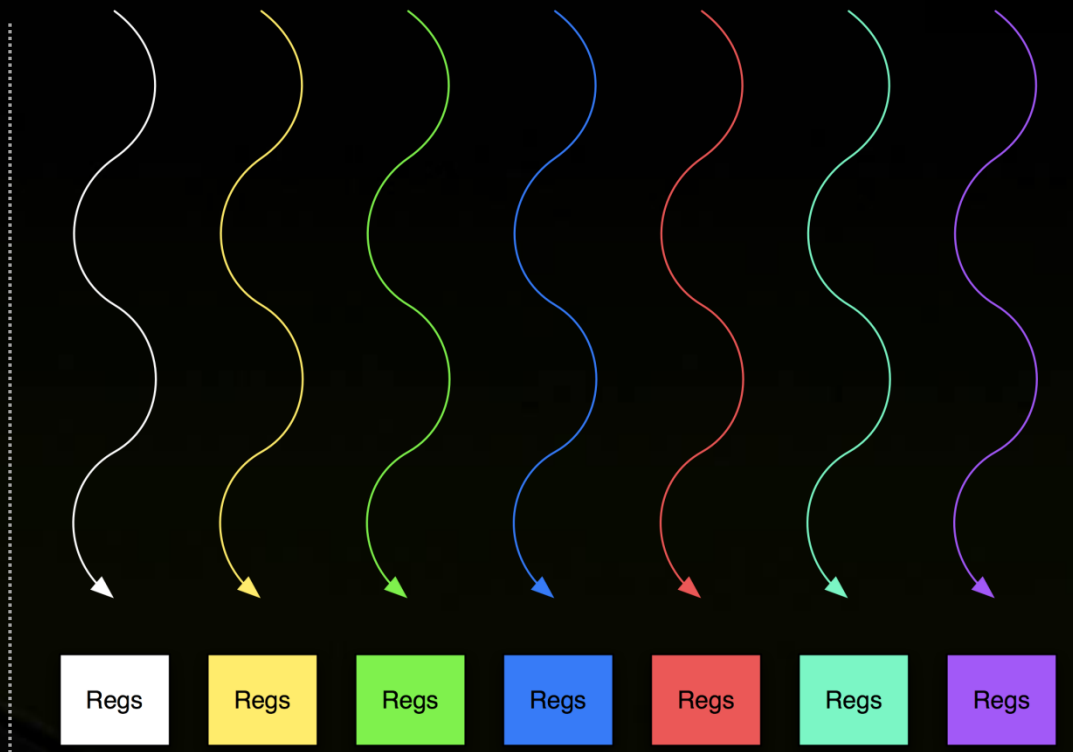
- **Global memory**
  - Analogous to RAM in a CPU server
  - Accessible by both GPU and CPU
  - Currently up to 6 GB
  - Bandwidth currently up to 150 GB/s for Quadro and Tesla products
  - ECC on/off option for Quadro and Tesla products
- **Streaming Multiprocessors (SMs)**
  - Perform the actual computations
  - Each SM has its own:
    - Control units, registers, execution pipelines, caches



MEMORY SYSTEM

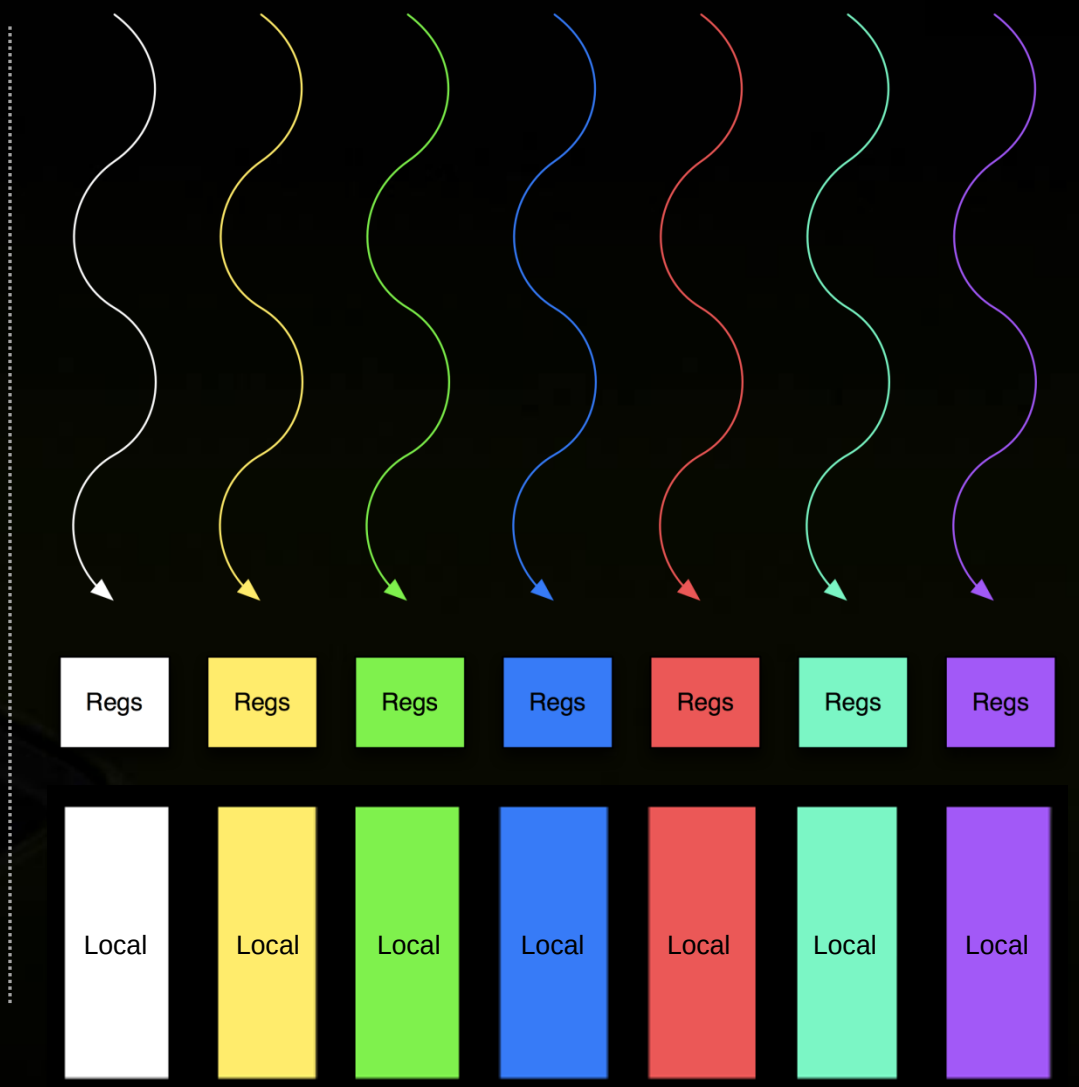
Memory hierarchy

- **Thread:**
  - **Registers**



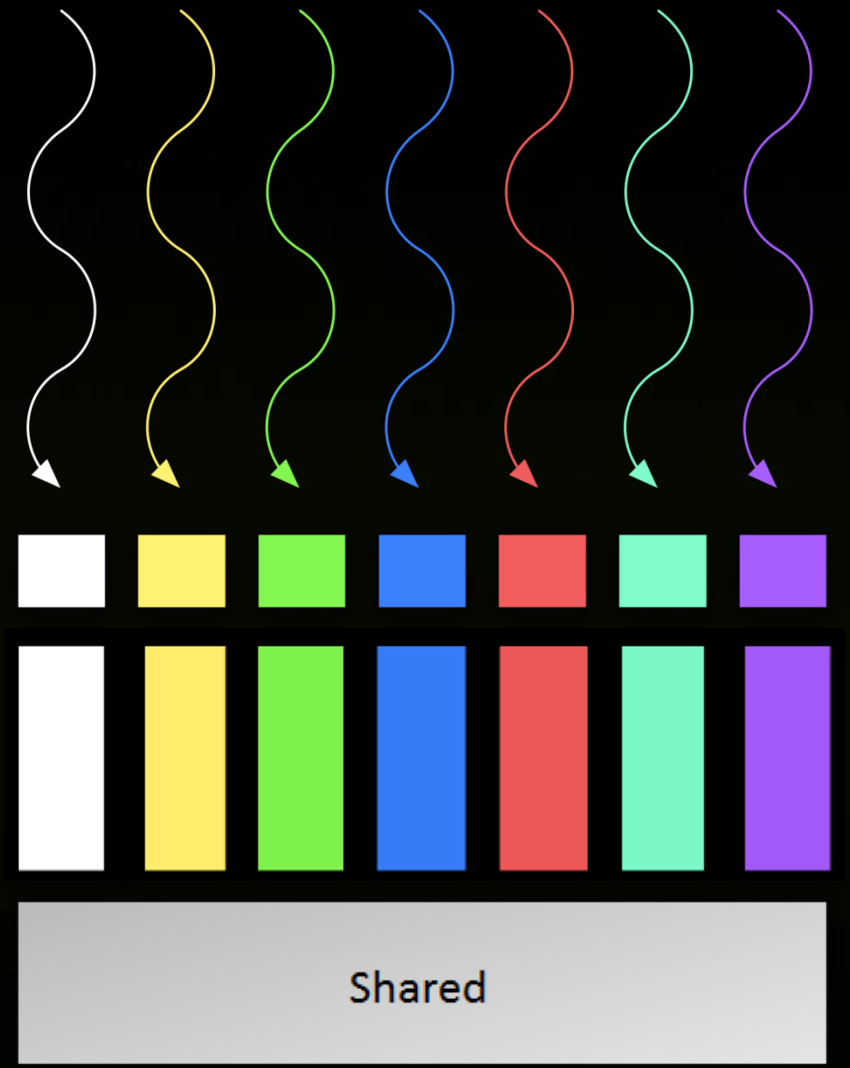
Memory hierarchy

- **Thread:**
  - **Registers**
  - **Local memory**



Memory hierarchy

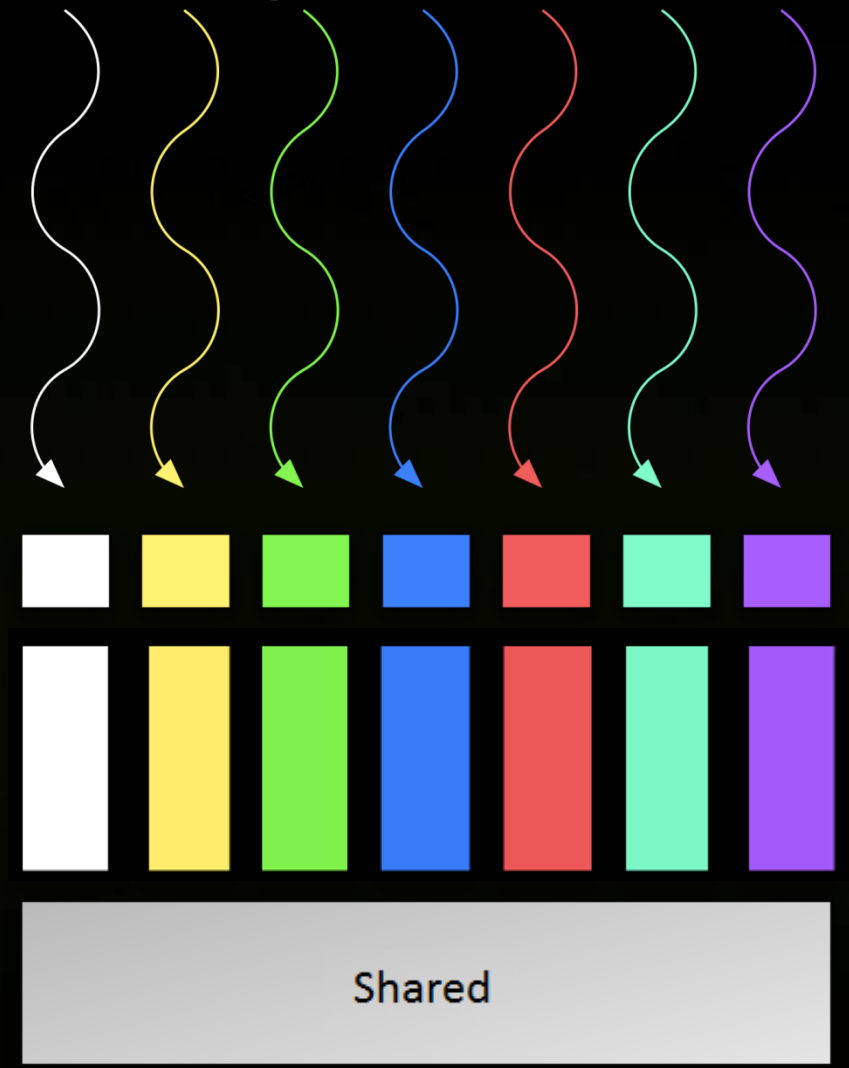
- **Thread:**
  - **Registers**
  - **Local** memory
- **Block of threads:**
  - **Shared** memory



Memory hierarchy : Shared memory

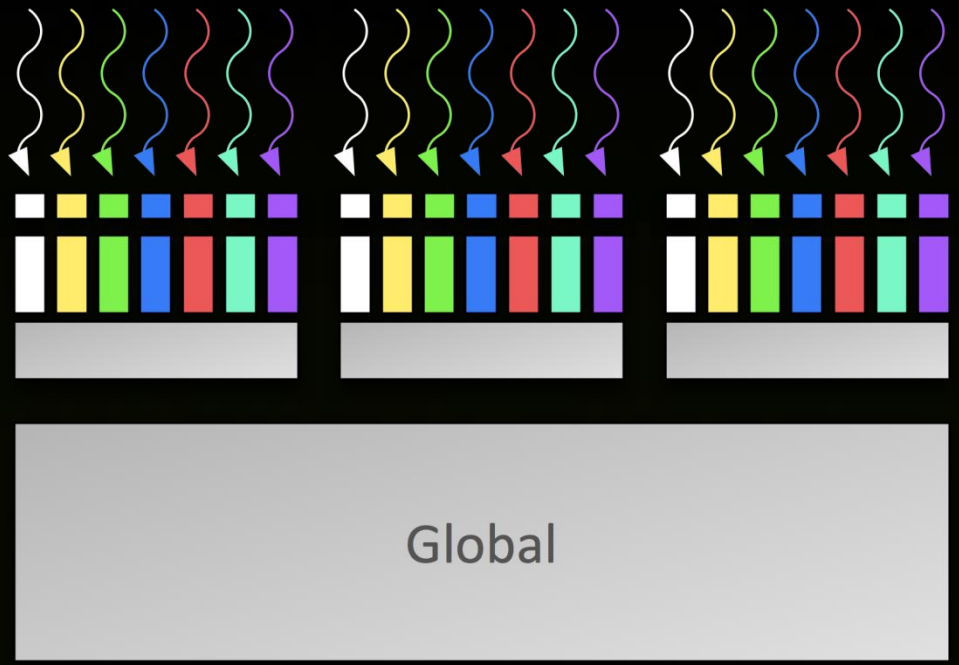
__shared__ int a[SIZE];

- Allocated per thread block, same lifetime as the block
- Accessible by any thread in the block
- Latency: a few cycles
- High aggregate bandwidth:
  - 14 * 32 * 4 B * 1.15 GHz / 2 = 1.03 TB/s
- Several uses:
  - Sharing data among threads in a block
  - User-managed cache (reducing gmem accesses)



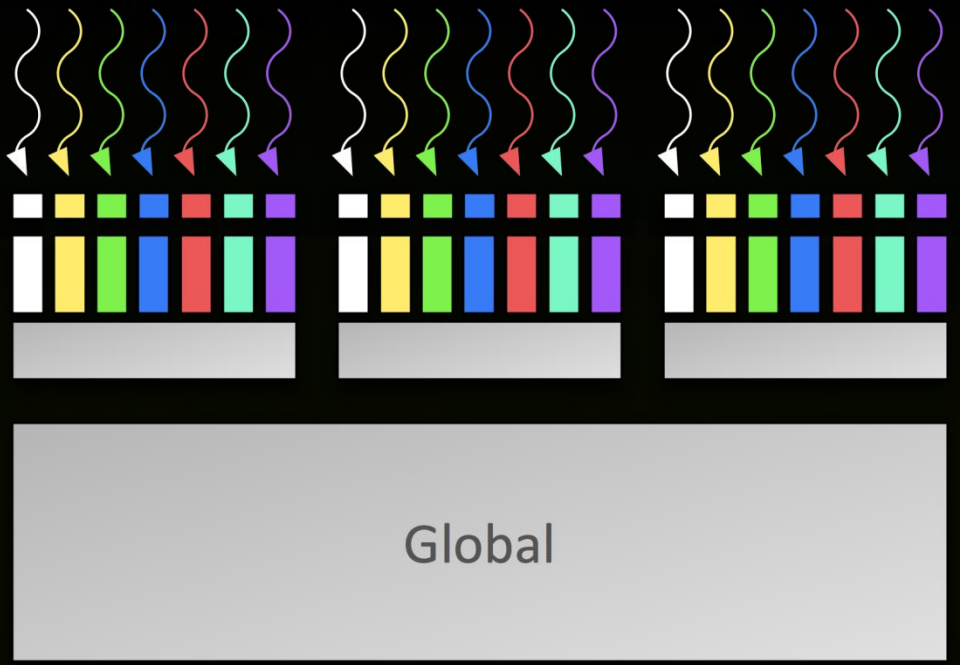
Memory hierarchy

- Thread:
  - Registers
  - Local memory
- Block of threads:
  - Shared memory
- All blocks:
  - Global memory



Memory hierarchy : Global memory

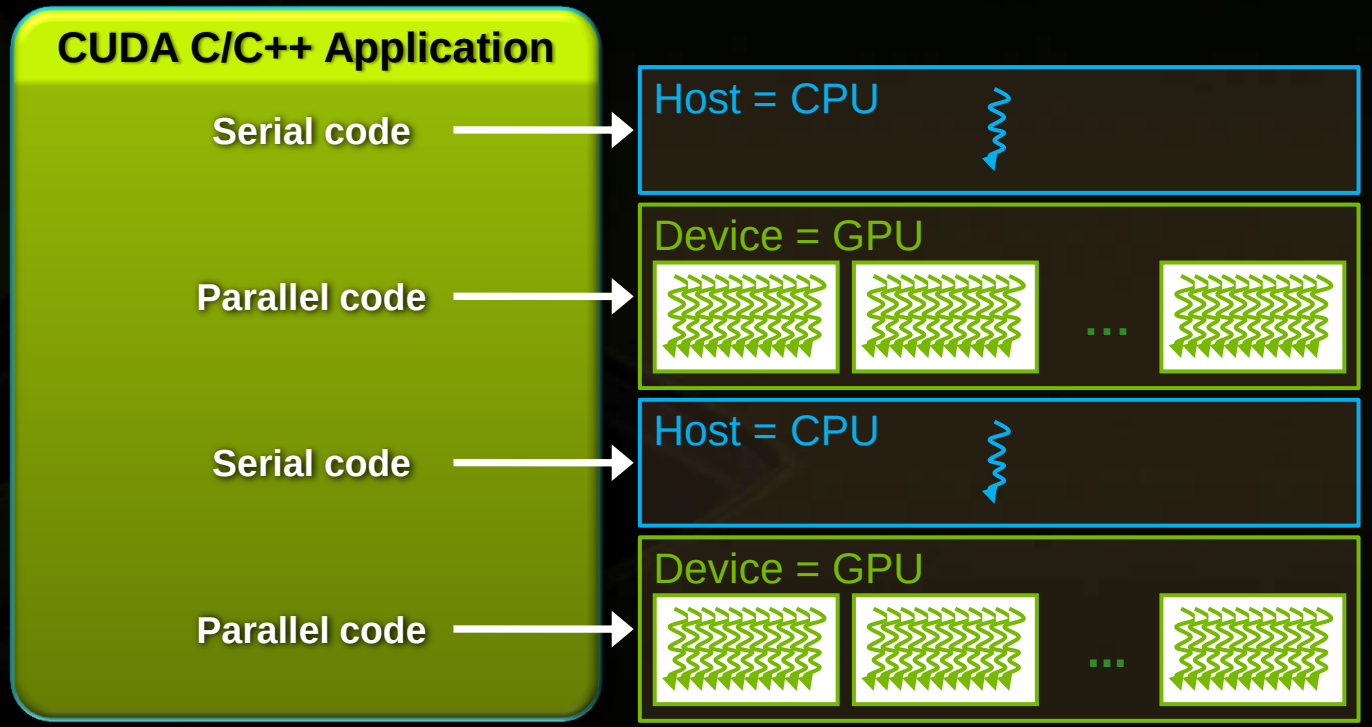
- Accessible by all threads of any kernel
- Data lifetime: from allocation to deallocation by host code
  - cudaMalloc (void ** pointer, size_t nbytes)
  - cudaMemset (void * pointer, int value, size_t count)
  - cudaFree (void* pointer)
- Latency: 400-800 cycles
- Bandwidth: 156 GB/s
  - Note: requirement on access pattern to reach peak performance



CUDA PROGRAMMING MODEL

Anatomy of a CUDA C/C++ Application

- **Serial** code executes in a **Host** (CPU) thread
- **Parallel** code executes in many **Device** (GPU) threads across multiple processing elements



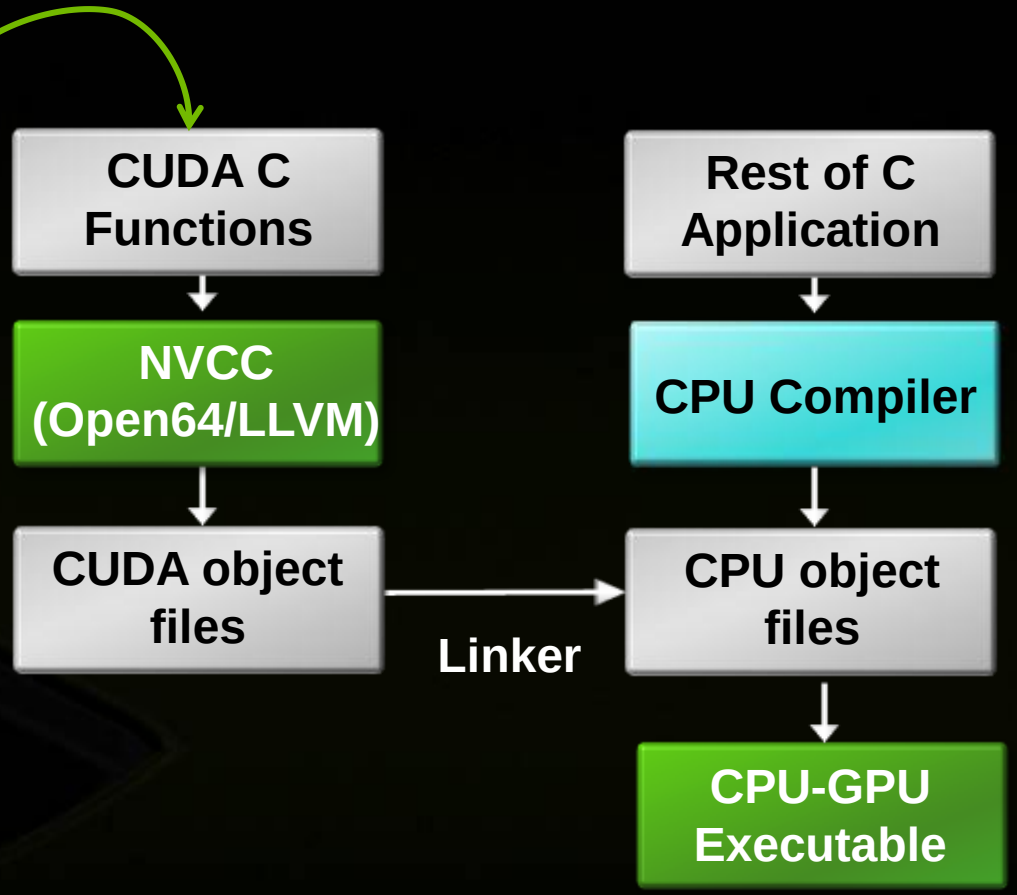
Compiling CUDA C Applications

```
void serial_function(… ) {
  ...
}
void other_function(int ... ) {
  ...
}
```

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

```
void main( ) {
  float x;
  saxpy_serial(..);
  ...
}
```

Modify into Parallel CUDA C code



CUDA C : C with a few keywords

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

Standard C Code

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

Parallel C Code

CUDA C : C with a few keywords

- Kernel: function called by the host that executes on the GPU
  - Can only access GPU memory
  - No variable number of arguments
  - No static variables
  - No recursion
- Functions must be declared with a qualifier:
  - __global__ : GPU kernel function launched by CPU, must return void
  - __device__ : can be called from GPU functions
  - __host__ : can be called from CPU functions (default)
  - __host__ and __device__ qualifiers can be combined

CUDA Kernels

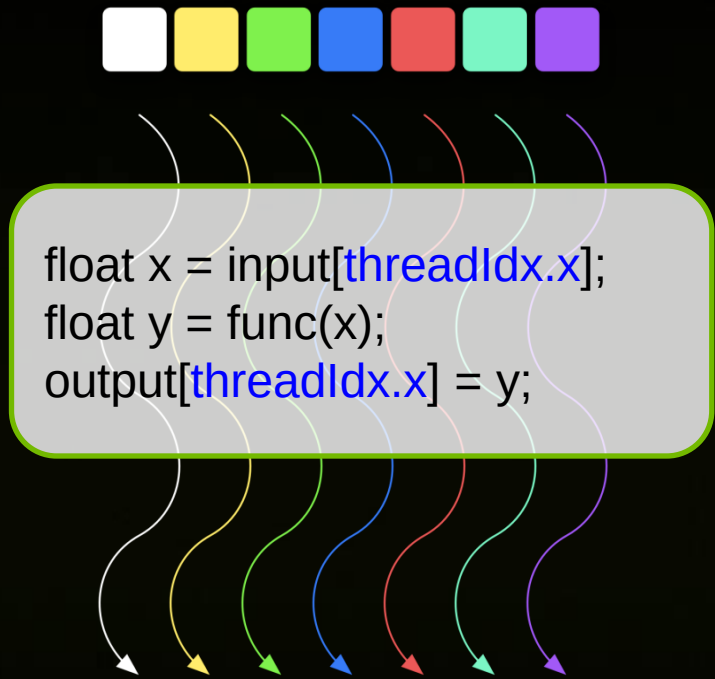
- Parallel portion of application: execute as a kernel
  - Entire GPU executes kernel, many threads

- CUDA threads:
  - Lightweight
  - Fast switching
  - 1000s execute simultaneously

CPU	Host	Executes functions
GPU	Device	Executes kernels

CUDA Kernels: Parallel Threads

- A **kernel** is a function executed on the GPU as an array of threads in parallel
- All threads execute the same code, can take different paths
- Each thread has an ID
  - Select input/output data
  - Control decisions

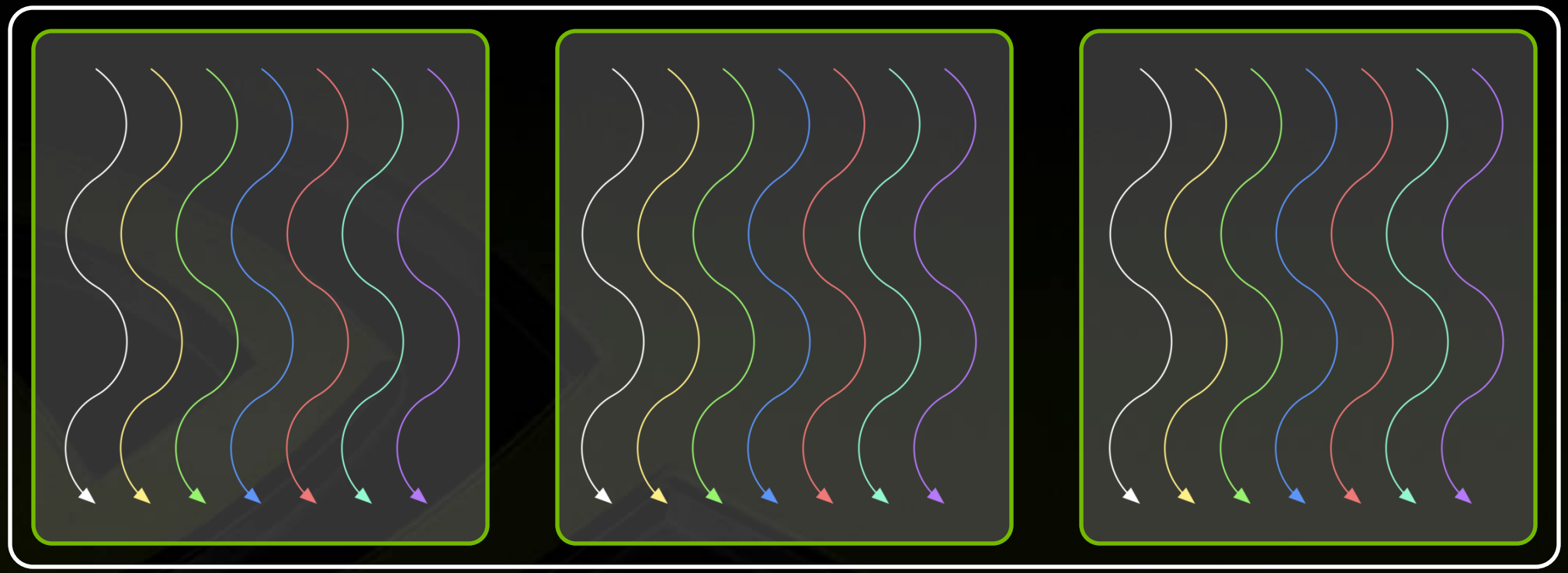


CUDA Kernels: Subdivide into Blocks

CUDA Kernels: Subdivide into Blocks

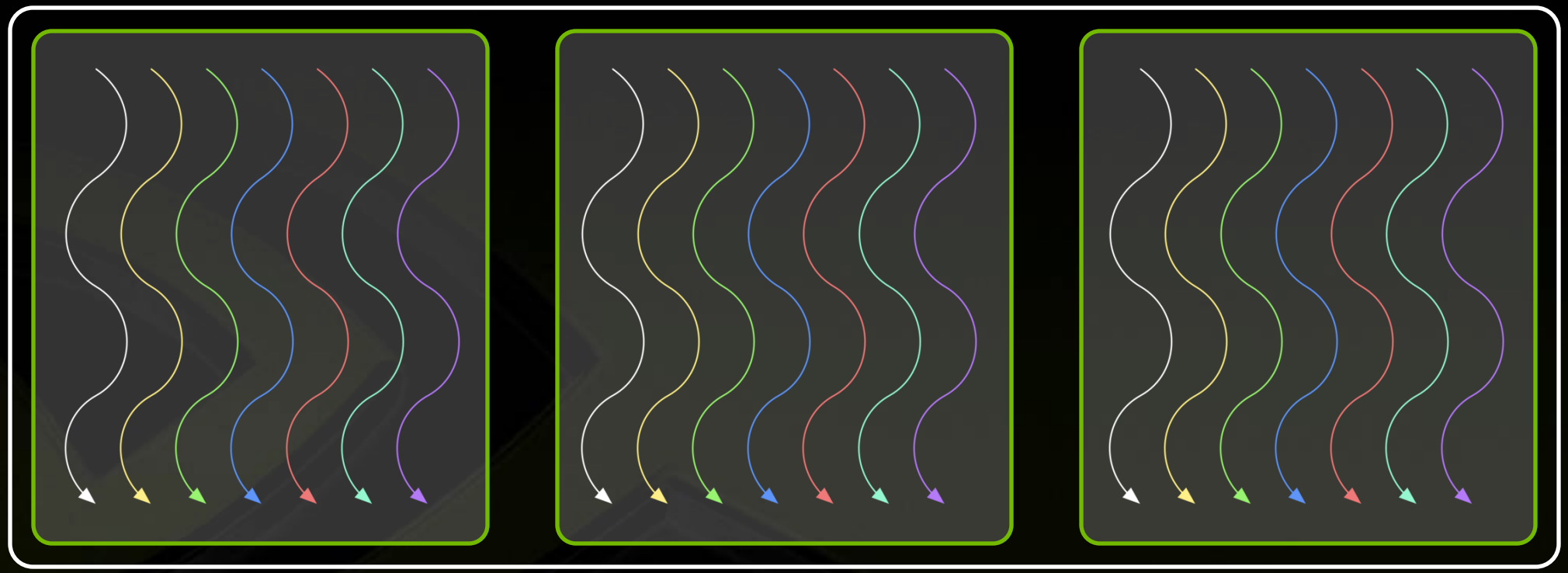
- Threads are grouped into blocks

CUDA Kernels: Subdivide into Blocks



- Threads are grouped into blocks
- Blocks are grouped into a grid

CUDA Kernels: Subdivide into Blocks

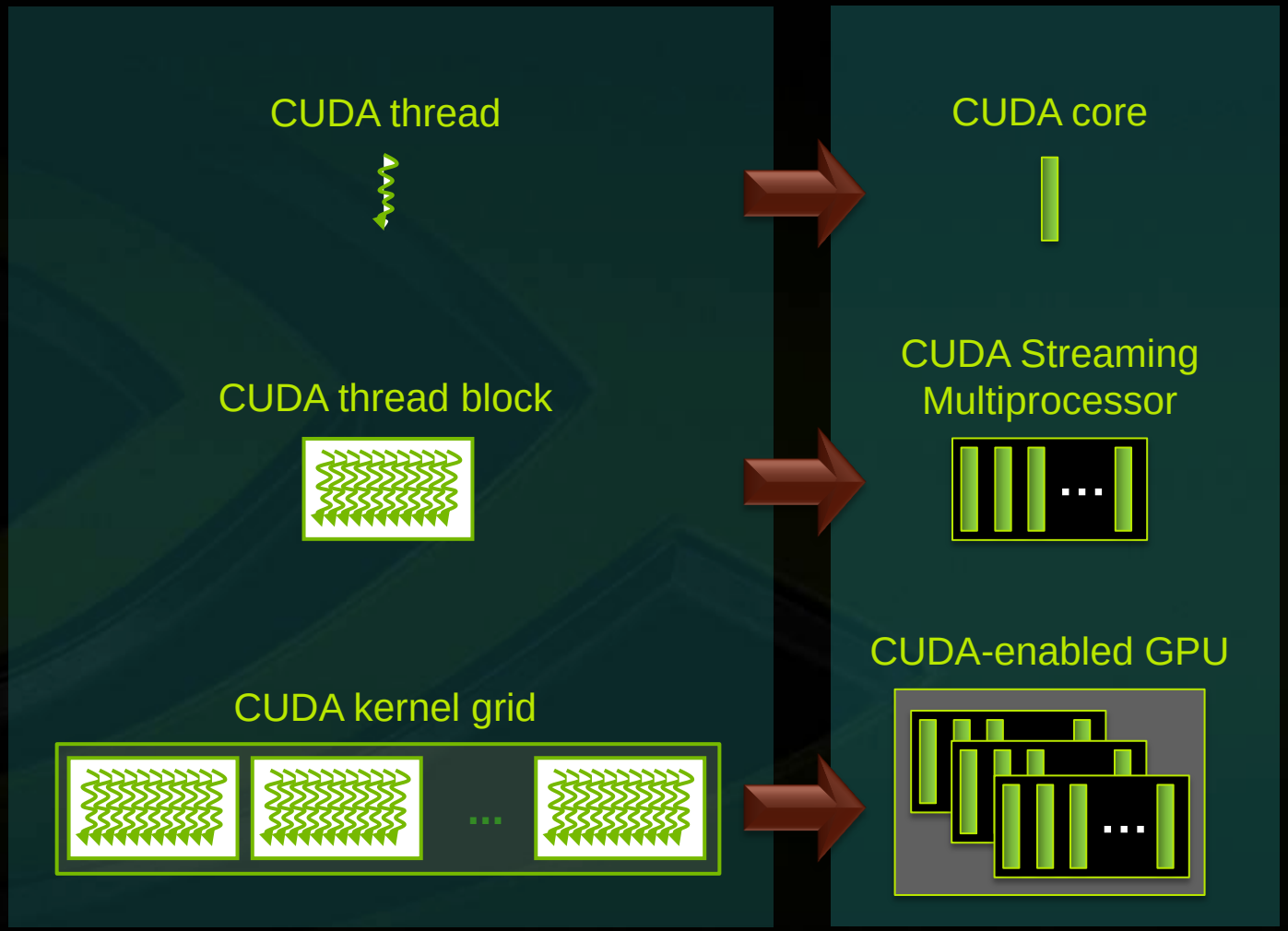


- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

CUDA Kernels: Subdivide into Blocks

- Threads are grouped into blocks
- Blocks are grouped into a grid
- A kernel is executed as a grid of blocks of threads

Kernel Execution



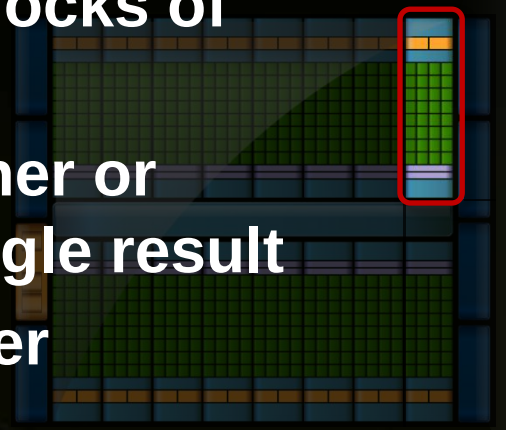
- Each thread is executed by a core

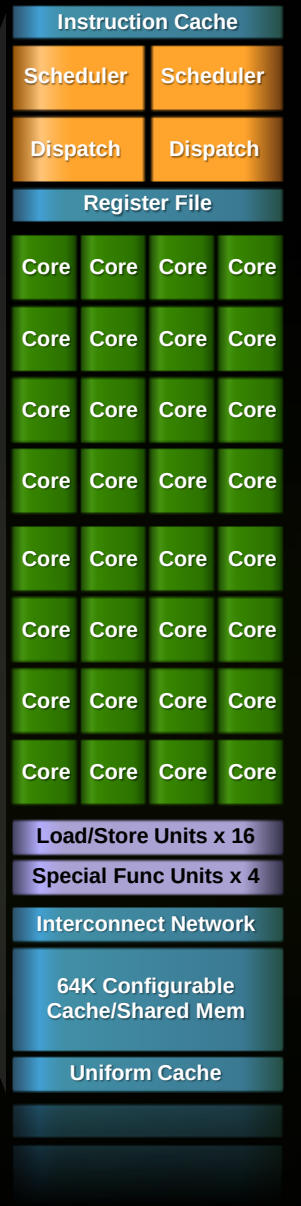
- Each block is executed by one SM and does not migrate
- Several concurrent blocks can reside on one SM depending on the blocks' memory requirements and the SM's memory resources

- Each kernel is executed on one device
- Multiple kernels can execute on a device at one time

Thread blocks allow cooperation

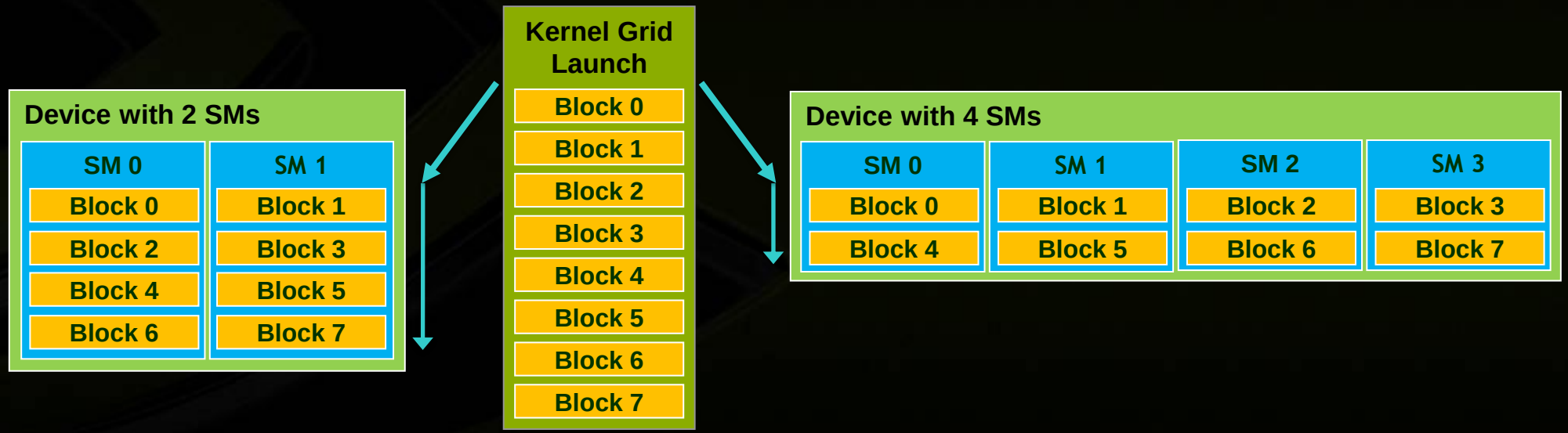
- Threads may need to cooperate:
  - Cooperatively load/store blocks of memory all will use
  - Share results with each other or cooperate to produce a single result
  - Synchronize with each other





Thread blocks allow scalability

- Blocks can execute in any order, concurrently or sequentially
- This independence between blocks gives scalability:
  - A kernel scales across any number of SMs




Intro to CUDA C
Memory Management
NVIDIA

Memory Spaces

CPU and GPU have separate memory spaces

- Data is moved across PCIe bus
- Use functions to allocate/set/copy memory on GPU
  - Very similar to corresponding C functions

Pointers are just addresses

- Can’t tell from the pointer value whether the address is on CPU or GPU
  - Must use **cudaPointerGetAttributes(…)**
- Must exercise care when dereferencing:
  - Dereferencing CPU pointer on GPU will likely crash
  - Dereferencing GPU pointer on CPU will likely crash

GPU Memory Allocation / Release

Host (CPU) manages device (GPU) memory

- cudaMalloc (void ** pointer, size_t nbytes)
- cudaMemset (void * pointer, int value, size_t count)
- cudaFree (void* pointer)

```
int n = 1024;
int nbytes = 1024*sizeof(int);
int * d_a = 0;
cudaMalloc( (void**)&d_a,  nbytes );
cudaMemset( d_a, 0, nbytes);
cudaFree(d_a);
```

Note: Device memory from GPU point of view is also referred to as global memory.

Data Copies

cudaMemcpy(void *dst, void *src, size_t nbytes,
  enum cudaMemcpyKind direction);

- returns after the copy is complete
- blocks CPU thread until all bytes have been copied
- doesn’t start copying until previous CUDA calls complete

enum cudaMemcpyKind

- cudaMemcpyHostToDevice
- cudaMemcpyDeviceToHost
- cudaMemcpyDeviceToDevice

Non-blocking memcopies are provided

Code Walkthrough 1

- Allocate CPU memory for *n* integers
- Allocate GPU memory for *n* integers
- Initialize GPU memory to 0s
- Copy from GPU to CPU
- Print the values

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );
```

Code Walkthrough 1

```
#include <stdio.h>

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a ) {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemset( d_a, 0, num_bytes );
    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

BASIC KERNELS AND EXECUTION

CUDA Programming Model revisited

- Parallel code (kernel) is launched and executed on a device by many threads
- Threads are grouped into thread blocks
- Parallel code is written for a thread
  - Each thread is free to execute a unique code path
  - Built-in thread and block ID variables

Thread Hierarchy

- Threads launched for a parallel section are partitioned into thread blocks
  - Grid = all blocks for a given launch
- Thread block is a group of threads that can:
  - Synchronize their execution
  - Communicate via shared memory

IDs and Dimensions

Threads

- 3D IDs, unique within a block

Blocks

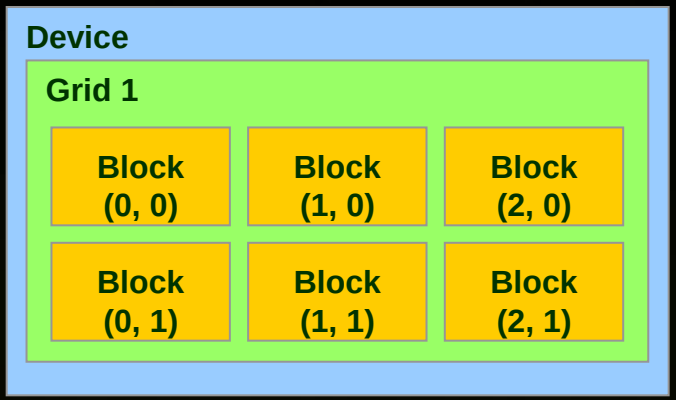
- 2D IDs, unique within a grid

Dimensions set at launch time

- Can be unique for each grid

Built-in variables

- threadIdx, blockIdx
- blockDim, gridDim



(Continued)

IDs and Dimensions

Threads

- 3D IDs, unique within a block

Blocks

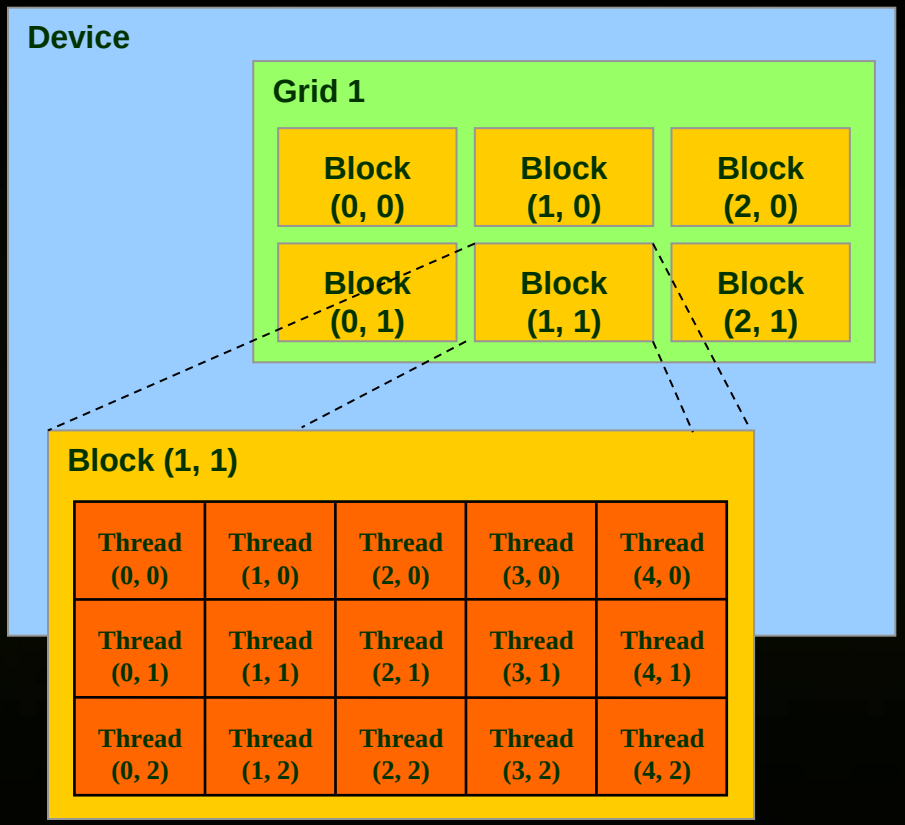
- 2D IDs, unique within a grid

Dimensions set at launch time

- Can be unique for each grid

Built-in variables

- threadIdx, blockIdx
- blockDim, gridDim



Code executed on GPU

C function with some restrictions:

- Can only access GPU memory (0-copy is the exception)
- No variable number of arguments
- No static variables
- No recursion

Must be declared with a qualifier:

- __global__ : launched by CPU, cannot be called from GPU must return void
- __device__ : called from other GPU functions, cannot be launched by the CPU
- __host__ : can be executed by CPU
- __host__ and __device__ qualifiers can be combined
  - Sample mathematical functions

Code Walkthrough 2

- Build on Walkthrough 1
- Write a kernel to initialize integers
- Copy the result back to CPU
- Print the values

Kernel Code (executed on GPU)

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Launching Kernels on GPU

Launch parameters (triple chevron <<<>>> notation)

- grid dimensions (up to 2D), dim3 type
- thread-block dimensions (up to 3D), dim3 type
- shared memory: number of bytes per block
  - for extern smem variables declared without size
  - Optional, 0 by default
- stream ID
  - Optional, 0 by default

```
dim3 grid(16, 16);
dim3 block(16,16);
kernel<<<grid, block, 0, 0>>>(...);
kernel<<<32, 512>>>(...);
```

```
#include <stdio.h>

__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}

int main()
{
    int dimx = 16;
    int num_bytes = dimx*sizeof(int);

    int *d_a=0, *h_a=0; // device and host pointers

    h_a = (int*)malloc(num_bytes);
    cudaMalloc( (void**)&d_a, num_bytes );

    if( 0==h_a || 0==d_a )  {
        printf("couldn't allocate memory\n"); return 1;
    }

    cudaMemset( d_a, 0, num_bytes );

      dim3 grid, block;
      block.x = 4;
      grid.x  = dimx / block.x;

      kernel<<<grid, block>>>( d_a );

    cudaMemcpy( h_a, d_a, num_bytes, cudaMemcpyDeviceToHost );

    for(int i=0; i<dimx; i++)
        printf("%d ", h_a[i] );
    printf("\n");

    free( h_a );
    cudaFree( d_a );

    return 0;
}
```

Kernel Variations and Output

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = 7;
}
```

Output: 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7 7

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = blockIdx.x;
}
```

Output: 0 0 0 0 1 1 1 1 2 2 2 2 3 3 3 3

```
__global__ void kernel( int *a )
{
    int idx = blockIdx.x*blockDim.x + threadIdx.x;
    a[idx] = threadIdx.x;
}
```

Output: 0 1 2 3 0 1 2 3 0 1 2 3 0 1 2 3

GPU Kernel Execution

How the HW executes kernels

- GPU consists of multiple cores (Multiprocessors, up to 30)
- Blocks are launched on SMs
- Each SM can have multiple concurrent blocks executing
- Once a block is started it will not migrate to another SM

Blocks Must Be Independent

Any possible interleaving of blocks should be valid

- presumed to run to completion without pre-emption
- can run in any order
- can run concurrently OR sequentially

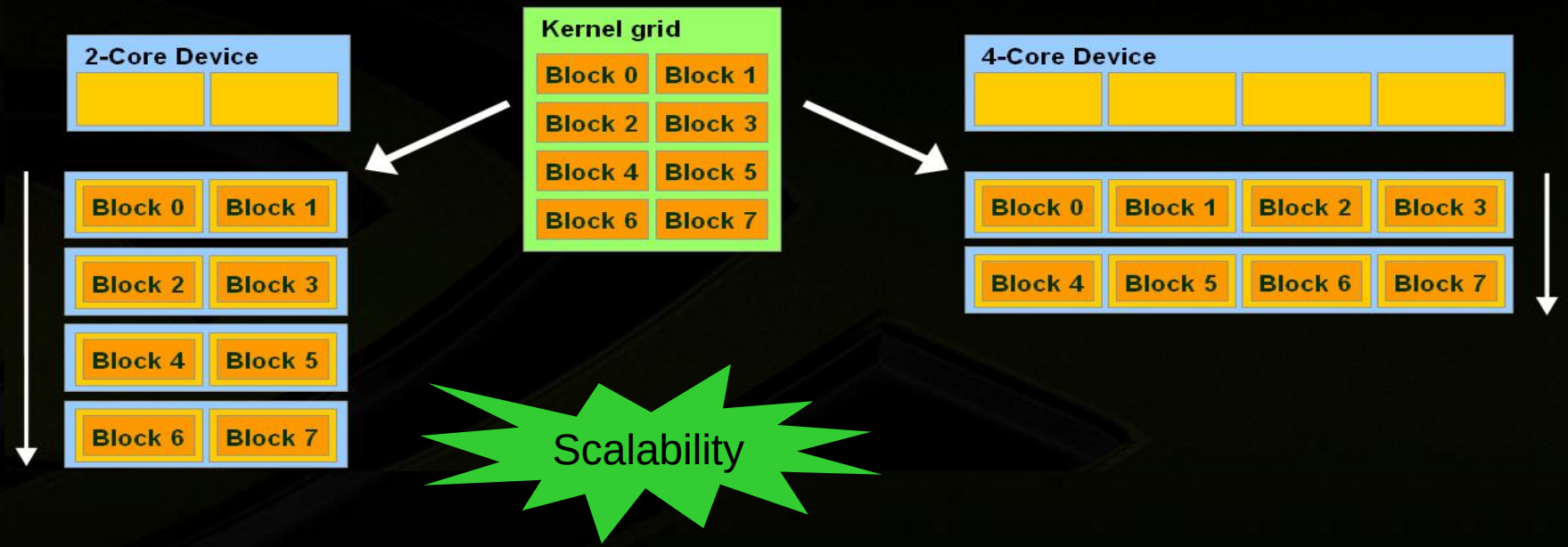
Blocks may coordinate but not synchronize

- shared queue pointer: OK
- shared lock: BAD … any dependence on order easily deadlocks

Independence requirement gives scalability

Blocks Must Be Independent

- Facilitates scaling of the same code across many devices



COORDINATING GPU AND GPU EXECUTION

Synchronizing GPU and CPU

- All kernel launches are asynchronous
  - control returns to CPU immediately
  - kernel starts executing once all previous CUDA calls have completed
- cudaMemcpy() is synchronous
  - control returns to CPU once the copy is complete
  - copy starts once all previous CUDA calls have completed
- cudaThreadSynchronize()
  - blocks until all previous CUDA calls complete
- Outlook: Asynchronous CUDA calls
  - non-blocking memcopies
  - ability to overlap memcopies and kernel execution

CUDA Error Reporting to CPU

- All CUDA calls return error code:
  - except kernel launches
  - cudaError_t type
- cudaError_t cudaGetLastError(void)
  - returns the code for the last error ("no error" has a code)
- char* cudaGetErrorString(cudaError_t code)
  - returns a null-terminated character string describing the error

```
printf("%s\n", cudaGetErrorString( cudaGetLastError() ) );
```

CUDA Event API

- Events are inserted (recorded) into CUDA call streams
- Usage scenarios:
  - measure elapsed time for CUDA calls (clock cycle precision)
  - query the status of an asynchronous CUDA call
  - block CPU until CUDA calls prior to the event are completed
  - asyncAPI sample in CUDA SDK

CUDA Event API

```
cudaEvent_t start, stop;
cudaEventCreate(&start);          cudaEventCreate(&stop);
cudaEventRecord(start, 0);
kernel<<<grid, block>>>(...);
cudaEventRecord(stop, 0);
cudaEventSynchronize(stop);
float et;
cudaEventElapsedTime(&et, start, stop);
cudaEventDestroy(start);                  cudaEventDestroy(stop);
```

Device Management

- CPU can query and select GPU devices
  - **cudaGetDeviceCount(int* count)**
  - **cudaSetDevice(int device)**
  - **cudaGetDevice(int *current_device)**
  - **cudaGetDeviceProperties(cudaDeviceProp* prop, int device)**
  - **cudaChooseDevice(int *device, cudaDeviceProp* prop)**
- **Outlook: Multi-GPU setup**
  - **device 0 is used by default**
  - **one CPU thread can control one GPU**
    - **multiple CPU threads can control the same GPU**
    - **SDK sample simpleMultiGPU**

Rich Toolchain & Ecosystem for Fast Ramp-up on GPUs

Numerical Packages

MATLAB
Mathematica
NI LabView
pyCUDA

Debuggers & Profilers

cuda-gdb
NV Visual Profiler
Parallel Nsight
Visual Studio
Allinea
TotalView

GPU Compilers

C
C++
Fortran
OpenCL
DirectCompute
Java
Python

Parallelizing Compilers

PGI Accelerator
CAPS HMPP
mCUDA
OpenMP

Libraries

BLAS
FFT
LAPACK
NPP
Sparse
Imaging
RNG

GPGPU Consultants & Training

acceleware
GASS
STONE RIDGE TECHNOLOGY
ANEO
GPU Tech
TECH
CAPS
HPC
EM Photonics
SCALABLE GRAPHICS
WIPRO

OEM Solution Providers

DELL
hp
IBM
CRAY
ASUS
SUPERMICRO
sgi
FUJITSU
BULL
APPRO
lenovo 联想
NEC

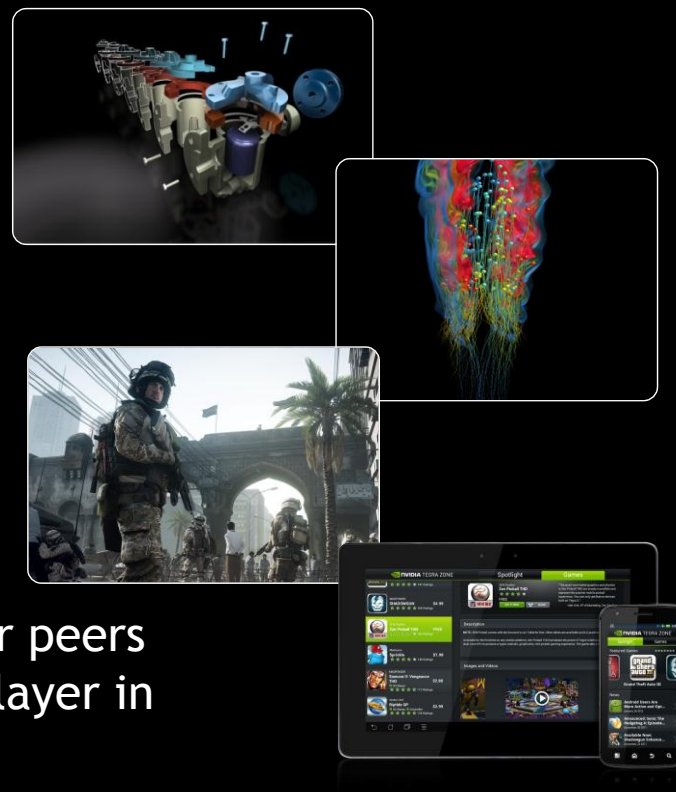
CUDA By the Numbers:

>375,000,000	CUDA Capable GPUs
>1,000,000	Toolkit Downloads
>120,000	Active Developers
>500	Universities Teaching CUDA
100%	OEMs offer CUDA GPU PCs

GPU Technology Conference 2013
March 18-21 | San Jose, CA

Why attend GTC?

GTC advances global awareness of the dramatic changes we're seeing in science and research, graphics, cloud computing, game development, and mobile computing, and how the GPU is central to innovation in all areas.



Ways to participate

- Submit a Research Poster – share your work and gain exposure as a thought leader
- Register – learn from the experts and network with your peers
- Exhibit/Sponsor – promote your organization as a key player in the GPU ecosystem

Visit www.gputechconf.com