
CUDA Fortran
SC11
Dr. Justin Luitjens, NVIDIA Corporation
NVIDIA

CUDA Fortran

- Co-defined by NVIDIA and PGI, implemented in the PGI Fortran compiler
  - Separate from PGI Accelerator
    - Directive-based, OpenMP-like interface to CUDA
- Cuda-like minimalistic design
  - Same semantics
  - Most features are have a one-to-one mapping to CUDA C
- Also supports a kernel loop directive
- I will only be covering the basics of CUDA Fortran for more details see the documentation

http://www.pgroup.com/resources/cudafortran.htm

CUDA C to CUDA Fortran

- A CUDA kernel is equivalent to a Fortran subroutine
  - Must either place inside a module or declare an explicit interface
- Use attributes() to specify qualifiers on variables, functions, and subroutines

C Qualifier	Fortran Attribute
__global__	global
__host__	host
__device__	device
__shared__	shared

C Built-in	Fortran Built-in
threadIdx.{xyz}	threadIdx%{xyz}
blockIdx.{xyz}	blockIdx%{xyz}
blockDim.{xyz}	blockDim%{xyz}
gridDim.{xyz}	gridDim%{xyz}
__syncthreads()	call syncthreads()

threadIdx & blockIdx are 1-based

CUDA C to CUDA Fortran

- The location of memory (host or device) and size are part of the data type in CUDA Fortran
- This allows implicit data transfers

C:
```
cudaMemcpy(d_x, h_x ,size, cudaMemcpyHostToDevice);
```

Fortran:
```
d_x = h_x
```

- Explicit memory copy routines also exist:

```
cudaMemcpy(d_x, h_x, size)
```

- Kernel Launches are a call to a subroutine with <<< >>>

C:
```
mykernel<<<numBlocks,numThreads>>>(…);
```

Fortran:
```
call mykernel<<<numBlocks,numThreads>>>(…)
```

```
Fortran parameters are passed by reference by default!
```

Kernel Loop Directive

- CUDA Fortran can automatically translate some loop constructs to kernels
  - Automatically creates and calls the kernel
  - Data must already be on the device

```
!$cuf kernel do(n) <<< grid, block >>
```

```
!$cuf kernel do(2) <<< *, * >>>
do j = 1, m
  do i = 1, n
    d_a(i,j) = d_b(i,j) + d_c(i,j)
  end do
end do
```

Reduction using CUF Kernels

- Compiler recognizes use of scalar reduction and generates one result

```
...
sum = 0.0
!$cuf kernel do <<<*,*>>>
do i = 1, n
  sum = sum + a_d(i)
end do
...
```

Compilation

- **Source-to-source compilation (generates CUDA C)**
  - pgfortran - PGI's Fortran compiler
  - All source code with .cuf or .CUF is compiled as CUDA Fortran enabled automatically
  - Flag to target architecture (e.g. -Mcuda=cc20)
    - -Mcuda=emu specifies emulation mode
  - Flag to target toolkit version (e.g. -Mcuda=cuda4.0)
  - -Mcuda=fastmath enables faster intrinsics (__sinf())
  - -Mcuda=nofma turns off fused multiply-add
  - -Mcuda=maxregcount:<n> limits register use per thread
  - -Mcuda=ptxinfo prints memory usage per kernel

Example: Hello World

Fortran

```
module hello
contains
  subroutine mykernel()
  end subroutine
end module

program gpu_example
  use hello

  print *, "Hello World!"
  call mykernel()
end program
```

Cuda Fortran

```
module hello
contains
  attributes(global) subroutine mykernel()
  end subroutine
end module

program gpu_example
  use hello
  use cudafor
  print *, "Hello World!"
  call mykernel<<<1,1>>>()
end program
```

Example: Vector Addition

Fortran

```
program vector_addition
  use vectorAdd

  integer, allocatable:: a(:), b(:), c(:)

  integer:: N
  N=1000000
  allocate( a(N) , b(N) , c(N) )
  call initializeVector(a, N)
  call initializeVector(b, N)



  call add(N,a,b,c)


  deallocate( a(N), b(N), c(N) )
end program
```

Cuda Fortran

```
program vector_addition
  use vectorAdd
  use cudafor
  integer, allocatable:: a(:), b(:), c(:)
  integer, device, allocatable:: d_a(:), d_b(:), d_c(:)
  integer:: N
  N=1000000
  allocate( a(N) , b(N) , c(N), d_a(N), d_b(N), d_c(N) )
  call initializeVector(a, N)
  call initializeVector(b, N)

  d_a = a
  d_b = b
  call add<<<CEILING(N/512.0),512>>>(N,d_a,d_b,d_c)
  c = d_c

 deallocate( a(N , b(N), c(N), d_a(N), d_b(N), d_c(N) )
end program
```

Example: Vector Addition

Fortran

```
module vectorAdd
contains
  subroutine add(N, a, b, c)
    integer:: N
    integer:: a(N), b(N), c(N)
    integer:: idx


    do idx = 1, N
      c(idx) = a(idx) + b(idx)
    end do

end subroutine
end module
```

Cuda Fortran

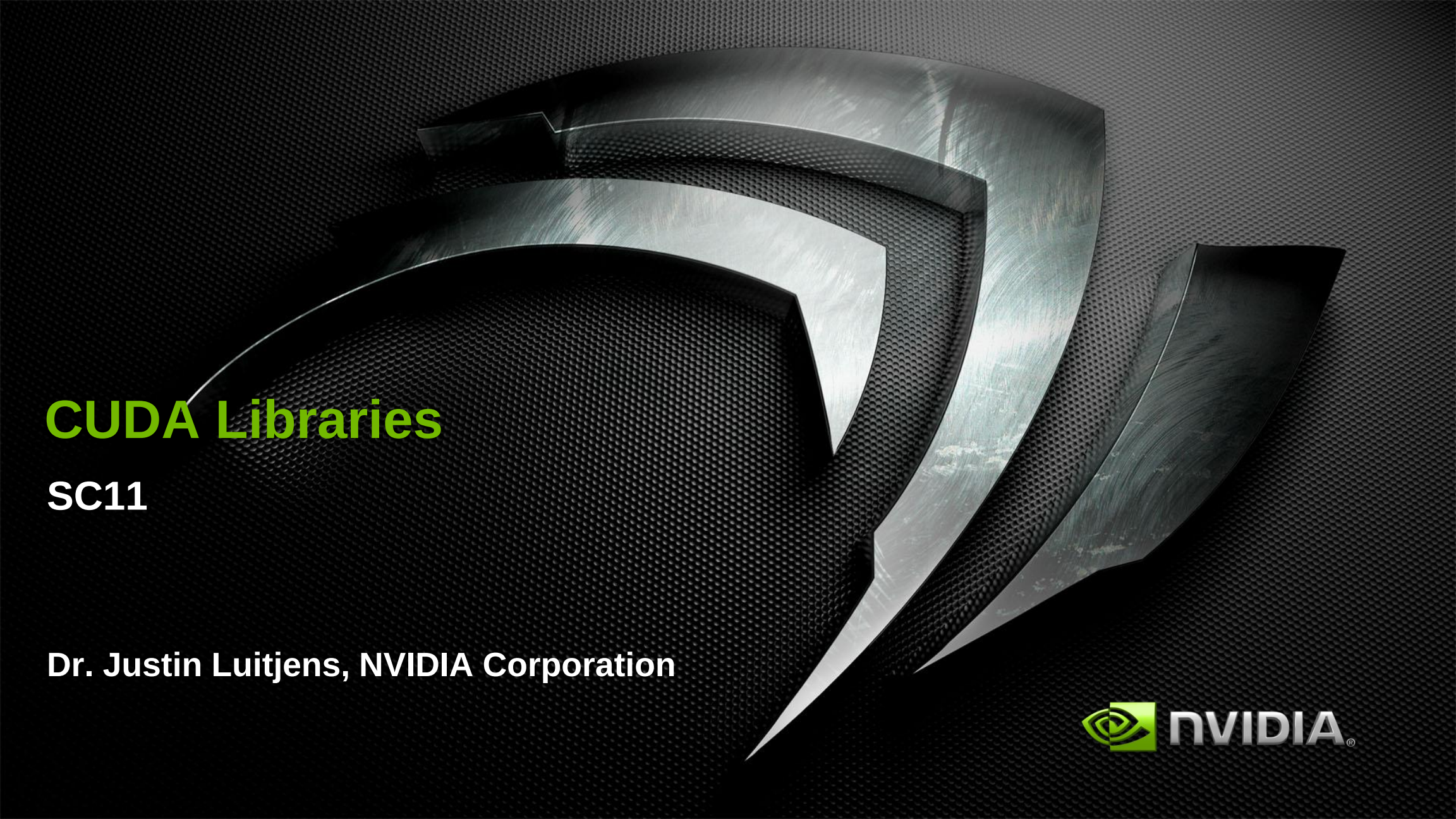
```
module vectorAdd
contains
  attributes(global) subroutine add(N, a, b, c)
    integer, value:: N
    integer:: a(N), b(N), c(N)
    integer:: idx

    idx = threadIdx%x+(blockIdx%x-1)*blockDim%x
    if ( idx <= N ) then
      c(idx) = a(idx) + b(idx)
    end if

  end subroutine
end module
```

Example: Cuda Fortran Stencil 1D Kernel

```
module stencil1d
  integer, parameter:: RADIUS = 3
  integer, parameter:: BLOCK_SIZE = 512
contains
  attributes(global) subroutine stencil_1d(N, in, out)
    integer, value:: N
    integer:: in(N), out(N)
    integer, shared:: temp(BLOCK_SIZE + 2 * RADIUS)
    integer:: gidx, lidx, sum, offset

    ! compute local and global index
    gidx = threadIdx%x + (blockIdx%x-1) * blockDim%x
    lidx = threadIdx%x + RADIUS

    ! load input into shared memory
    temp(lidx) = in(gidx)
    if (threadIdx%x <= RADIUS) then
      temp(lidx – RADIUS) = in(gidx – RADIUS)
      temp(lidx + BLOCK_SIZE) = in(gidx + BLOCK_SIZE)
    end if

    ! wait for all threads
    call syncthreads()

    ! sum out of shared memory
    sum = 0
    do offset = -RADIUS, RADIUS
      sum = sum + temp(lidx + offset)
    end do

    ! write the sum to global memory
    out(gidx) = sum
  end subroutine
end module
```

CUDA Fortran

- Cuda Fortran Programming Guide
  - http://www.pgroup.com/lit/whitepapers/pgicudaforug.pdf
- Download CUDA Fortran from PGI
  - 15 day free trial
  - Includes PGI Accelerator
    - Completely directive based approach to GPU programming
  - http://www.pgroup.com/resources/cudafortran.htm


CUDA Libraries
SC11
Dr. Justin Luitjens, NVIDIA Corporation
NVIDIA

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 Parallel Algorithms & Data Structures**
- **math.h - C99 floating-point Library**

- **Included in the CUDA Toolkit (free download)**
  - **www.nvidia.com/getcuda**

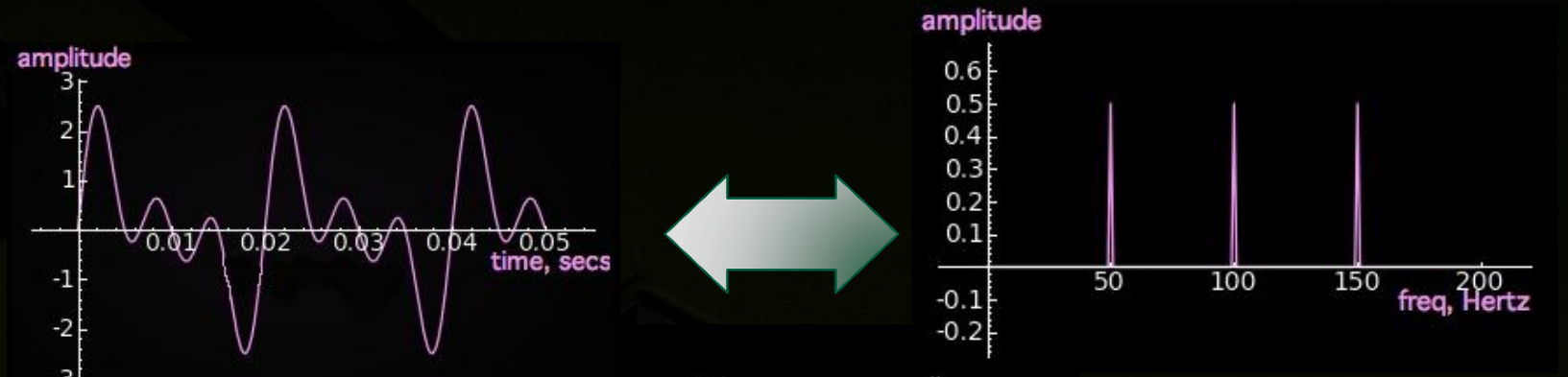
Today's Goal

- Provide an overview of various libraries and their features but not necessarily how to use them.
- For additional information please consult the library's documentation.

http://developer.nvidia.com/nvidia-gpu-computing-documentation

cuFFT Library

CUFFT is a GPU based Fast Fourier Transform library



$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

$$f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$

cuFFT Library Features

- Algorithms based on Cooley-Tukey ($n = 2^a \cdot 3^b \cdot 5^c \cdot 7^d$) and Bluestein
- Simple interface similar to FFTW
- 1D, 2D and 3D transforms of complex and real data
- Single precision (SP) and Double precision (DP) transforms
- Row-major order (C-order) for 2D and 3D data
- In-place and out-of-place transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- Streamed asynchronous execution
- Non normalized output: IFFT(FFT(A))=len(A)*A
- API is now thread-safe & callable from multiple host threads

cuFFT in 4 easy steps

Step 1 – Allocate space on GPU memory

Step 2 – Create plan specifying transform configuration like the size and type (real, complex, 1D, 2D and so on).

Step 3 – Execute the plan as many times as required, providing the pointer to the GPU data created in Step 1.

Step 4 – Destroy plan, free GPU memory

Example cuFFT Program

```
#include <stdlib.h>
#include <stdio.h>
#include “cufft.h”

#define NX 256
#define NY 128

int main() {
  cufftHandle plan;
  cufftComplex *idata, *odata;
  int i;

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

  /* initialize idata  */
  …
```

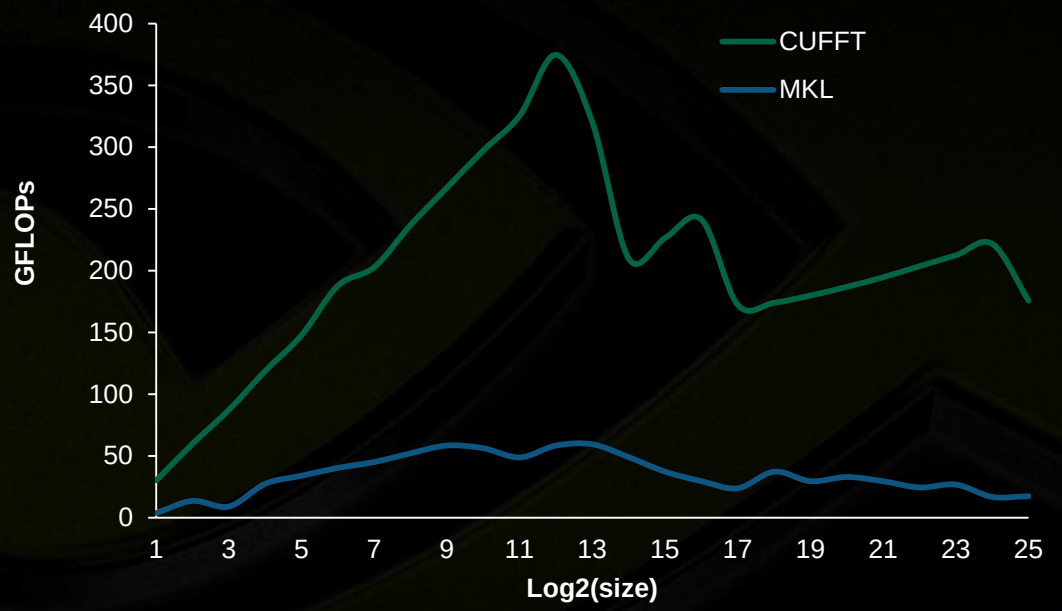
```
  /* create a 2D FFT plan */
  cufftPlan2d(&plan, NX, NY, CUFFT_C2C);

  /* Use the CUFFT plan to transform the signal out of place.
   *  Note:  idata != odata indicates an out of place
   *  transformation to CUFFT at execution time. */
   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);

  return 0
}
```

Check out the CUDA SDK for more examples

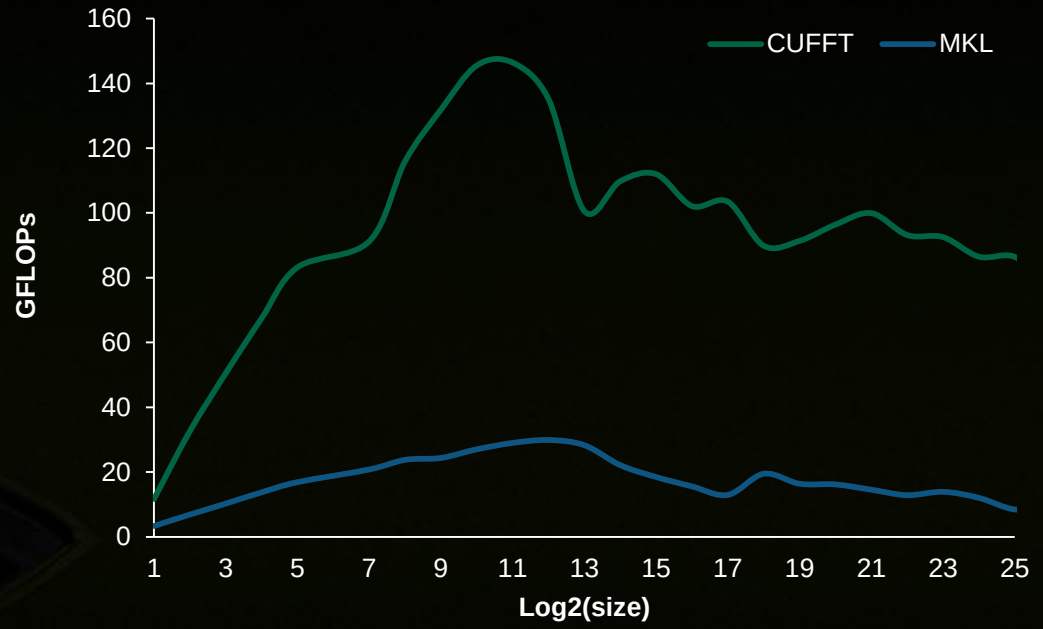
FFTs up to 10x Faster than MKL

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

cuFFT-Single Precision



cuFFT-Double Precision



- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015Xeon x5680 @ 3.33 GHz

cuBLAS Library

- Implementation of BLAS (Basic Linear Algebra Subprograms)
  - Self-contained at the API level
- Supports all the BLAS functions
  - Level1 (vector,vector): O(N)
    - AXPY : y = alpha.x + y
    - DOT : dot = x.y
  - Level 2(matrix,vector): $O(N^{2})$
    - Vector multiplication by a General Matrix : GEMV
    - Triangular solver : TRSV
  - Level3(matrix,matrix): $O(N^{3})$
    - General Matrix Multiplication : GEMM
    - Triangular Solver : TRSM
- Following BLAS convention, CUBLAS uses column-major storage

http://developer.nvidia.com/nvidia-gpu-computing-documentation

Using cuBLAS

- Support of 4 types
  - Float, Double, Complex, Double Complex
  - Respective Prefixes : S, D, C, Z
- Function naming convention: cublas + BLAS name
  - Example: cublasSGEMM
- Interface to CUBLAS library is in cublas.h
- Helper functions:
  - Memory allocation, data transfer

Calling cuBLAS from C

```
#include <stdlib.h>
#include <stdio.h>
#include “cublas.h”

int main() {
  float *a, *b, *c, *d_a, *d_b, *d_c;
  int n=4096;
  int num_bytes=n*n*sizeof(float);

  /* initialize cublas  */
  cublasInit();

  /* allocate memory  */
  a=(float*)malloc(num_bytes);
  b=(float*)malloc(num_bytes);
  c=(float*)malloc(num_bytes);
  cublasAlloc(n*n, sizeof(float), (void**) & d_a);
  cublasAlloc(n*n, sizeof(float), (void**) & d_b);
  cublasAlloc(n*n, sizeof(float), (void**) & d_c);

   /* initialize matrices a and b on host */
    …

  /* copy matrices to device */
  cublasSetMatrix(n,n,sizeof(float), a, n, d_a, n);
  cublasSetMatrix(n,n,sizeof(float), b, n, d_b, n);

  /* perform multiplication */
  cublasSgemm(‘N’, ‘N’, n, n, n, 1.0, d_a, n, d_b, n, 0.0, d_c, n);

  /* copy result back to the host */
  cublasGetMatrix( n, n, sizeof(float), d_c, n, c, n);

  /* free memory */
  cublasFree(d_a); cublasFree(d_b); cublasFree(d_c);
  free(a); free(b); free(c);

  /* shutdown cublas  */
  cublasShutdown();
  return 0;
}
```

Calling cuBLAS from FORTRAN

- **Two interfaces:**
  - **Thunking**
    - Allows interfacing to existing applications without any changes
    - During each call, the wrappers allocate GPU memory, copy source data from CPU memory space to GPU memory space, call CUBLAS, and finally copy back the results to CPU memory space and deallocate the GPGPU memory
    - Intended for light testing due to call overhead
  - **Non-Thunking** (default)
    - Intended for production code
    - Substitute device pointers for vector and matrix arguments in all BLAS functions
    - Existing applications need to be modified slightly to allocate and deallocate data structures in GPGPU memory space (using CUBLAS_ALLOC and CUBLAS_FREE) and to copy data between GPU and CPU memory spaces (using CUBLAS_SET_VECTOR, CUBLAS_GET_VECTOR, CUBLAS_SET_MATRIX, and CUBLAS_GET_MATRIX)

SGEMM example (Thunking)

```
program example_sgemm
  ! Define  3  single precision matrices A, B, C
  real, dimension(:,:),allocatable:: A(:,:),B(:,:),C(:,:)
  integer:: n=16
  allocate (A(n,n),B(n,n),C(n,n))
  ! Initialize A, B and C
  …
  #ifdef CUBLAS
    ! Call SGEMM in CUBLAS library using THUNKING interface (library takes care of
    ! memory allocation on device and data movement)
    call cublas_SGEMM('n','n', n,n,n,1.,A,n,B,n,1.,C,n)
  #else
    ! Call SGEMM in  host BLAS library
    call SGEMM('n','n', n,n,n,1.,A,n,B,n,1.,C,n)
  #endif
end program example_sgemm
```

To use the host BLAS routine:

g95 –O3 code.f90 –L/usr/local/lib -lblas

To use the CUBLAS routine (fortran_thunking.c is included in the toolkit /usr/local/cuda/src):

nvcc -O3 -c fortran_thunking.c

g95 -O3 -DCUBLAS code.f90 fortran_thunking.o -L/usr/local/cuda/lib64 -lcudart -lcublas

SGEMM example (Non-thunking)

```
program example_sgemm
  real, dimension(:,:),allocatable:: A(:,:),B(:,:),C(:,:)
  integer*8:: devPtrA, devPtrB, devPtrC
  integer:: n=16, size_of_real=16
  allocate (A(n,n),B(n,n),C(n,n))
  call cublas_Alloc(n*n ,size_of_real, devPtrA)
  call cublas_Alloc(n*n, size_of_real, devPtrB)
  call cublas_Alloc(n*n, size_of_real, devPtrC)
  ! Initialize A, B and C
  …
  ! Copy data to GPU
  call cublas_Set_Matrix(n, n, size_of_real, A, n, devPtrA, n)
  call cublas_Set_Matrix(n, n, size_of_real, B, n ,devPtrB, n)
  call cublas_Set_Matrix(n, n, size_of_real, C, n, devPtrC, n)
  ! Call SGEMM in CUBLAS library
  call cublas_SGEMM('n', 'n', n, n, n, 1., devPtrA, n, devPtrB, n, 1., devPtrC, n)
  ! Copy data from GPU
  call cublas_Get_Matrix( n, n, size_of_real, devPtrC, n, C, n)

  call cublas_Free( devPtrA )
  call cublas_Free( devPtrB )
  call cublas_Free( devPtrC )
end program example_sgemm
```

To use the CUBLAS routine (fortran.c is included in the toolkit /usr/local/cuda/src):

```
nvcc -O3 -c  fortran.c
g95 -O3  code.f90 fortran.o -L/usr/local/cuda/lib64 -lcudart  -lcublas
```

Calling cuBLAS from CUDA Fortran

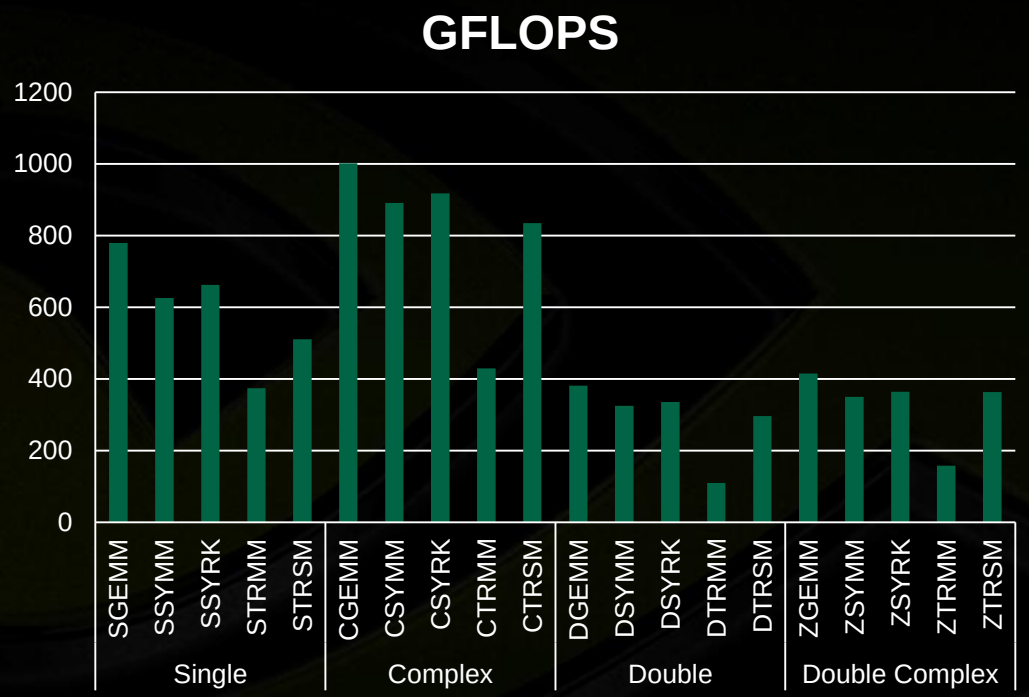
- Module which defines interfaces to CUBLAS from CUDA Fortran
  - `use cublas`
- Interfaces in three forms
  - Overloaded BLAS interfaces that take device array arguments
    - `call saxpy(n, a_d, x_d, incx, y_d, incy)`
  - Legacy CUBLAS interfaces
    - `call cublasSaxpy(n, a_d, x_d, incx, y_d, incy)`
  - Multi-GPU version (CUDA 4.0) that utilizes a handle `h`
    - `istat = cublasSaxpy_v2(h, n, a_d, x_d, incx, y_d, incy)`
- Mixing the three forms is allowed

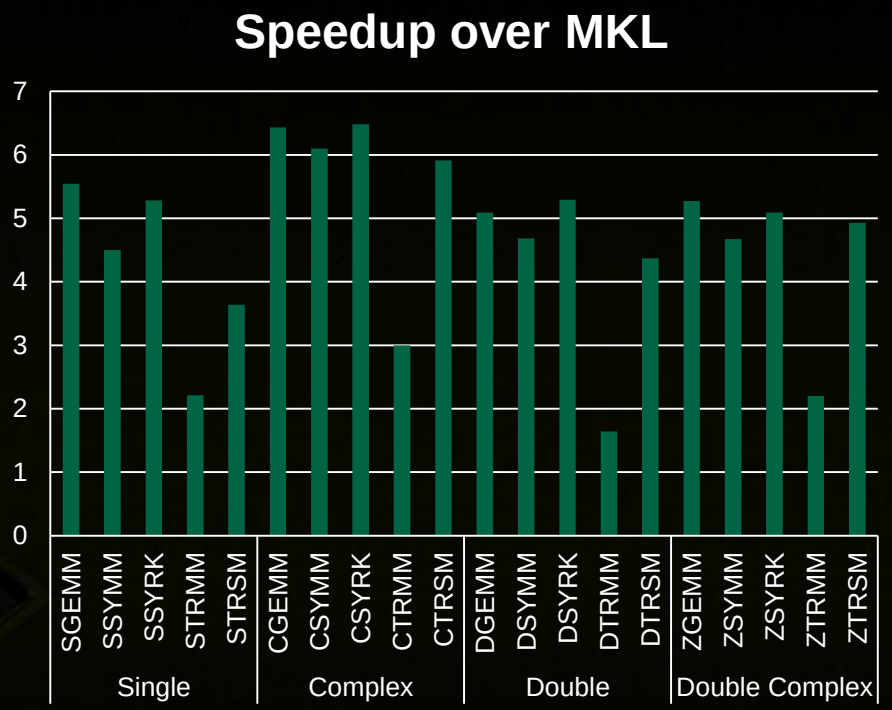
Calling cuBLAS from CUDA Fortran

```
program cublasTest
  use cublas
  implicit none

  real, allocatable :: a(:,:),b(:,:),c(:,:)
  real, device, allocatable :: a_d(:,:),b_d(:,:),c_d(:,:)
  integer :: k=4, m=4, n=4
  real :: alpha=1.0, beta=2.0

  allocate(a(m,k), b(k,n), c(m,n), a_d(m,k), b_d(k,n), c_d(m,n))

  a = 1; a_d = a
  b = 2; b_d = b
  c = 3; c_d = c

  call cublasSgemm('N', 'N', m , n, k, alpha, a_d, m, b_d, k, beta, c_d, m)

  c=c_d

  deallocate(a, b, c, a_d, b_d, c_d)

end program cublasTest
```

cuBLAS Level 3 Performance

Up to ~800GFLOPS and **>6x** speedup over MKL





- 4Kx4K matrix size
- cuBLAS 4.1, Tesla M2090 (Fermi), ECC on
- MKL 10.2.3, TYAN FT72-B7015Xeon x5680 @ 3.33 GHz

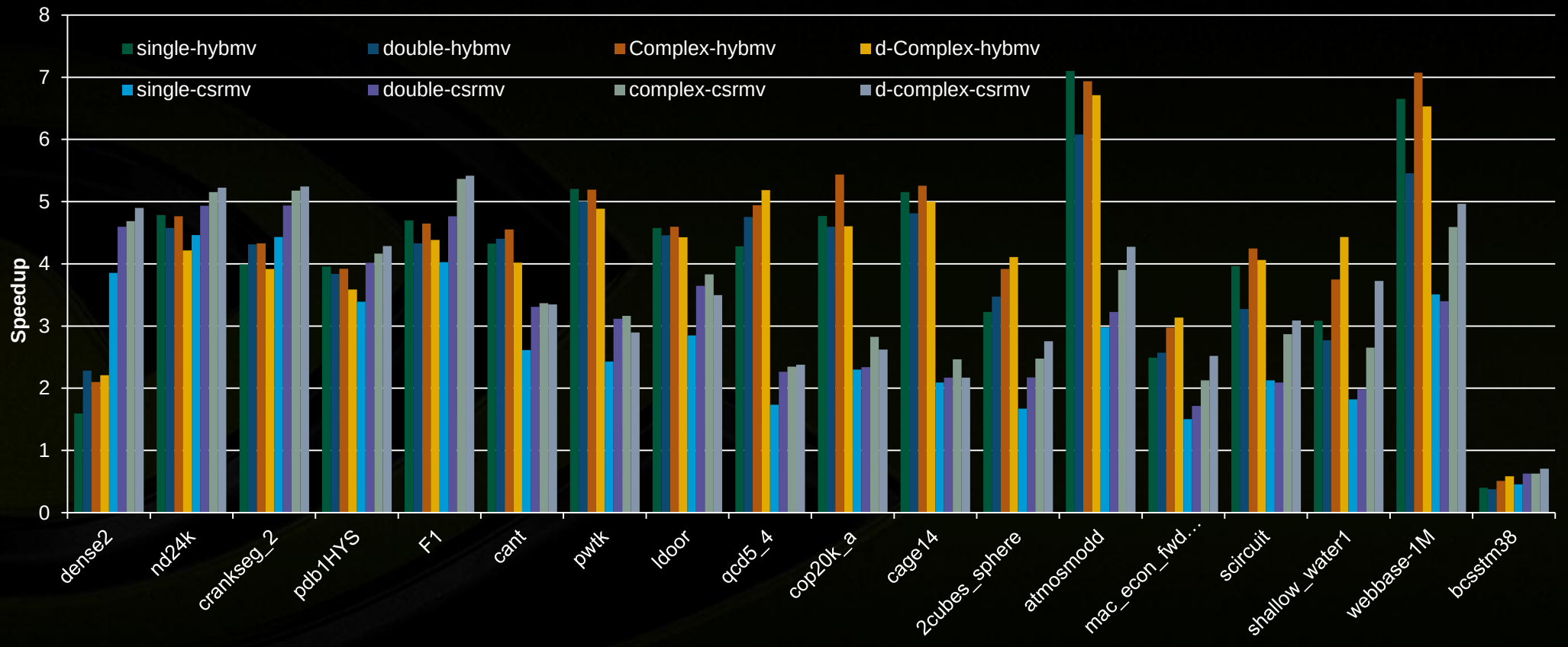
cuSPARSE: Sparse Linear Algebra Routines

- Conversion routines for dense, COO, CSR and CSC formats
- Optimized sparse matrix-vector multiplication for CSR
- New Sparse Triangular Solve CUDA 4.0/4.1
  - API optimized for common iterative solve algorithms

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \alpha \begin{bmatrix} 1.0 & & & \\ 2.0 & 3.0 & & \\ & & 4.0 & \\ 5.0 & & 6.0 & 7.0 \end{bmatrix} \begin{bmatrix} 1.0 \\ 2.0 \\ 3.0 \\ 4.0 \end{bmatrix} + \beta \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix}$$

cuSPARSE is up to 6x Faster than MKL

Sparse Matrix x Dense Vector

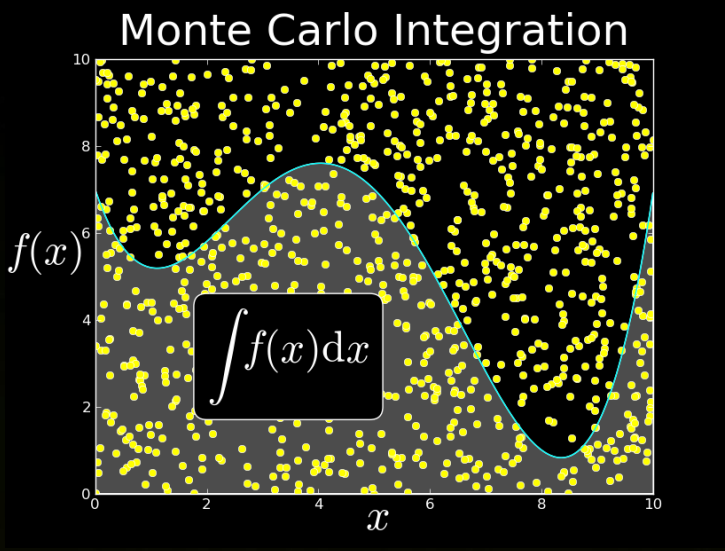


Performance may vary based on OS version and motherboard configuration

* cuSPARSE 4.1, NVIDIA C2050 (Fermi), ECC on
* MKL 10.2.3, TYAN FT72-B7015Xeon x5680 @ 3.33 GHz

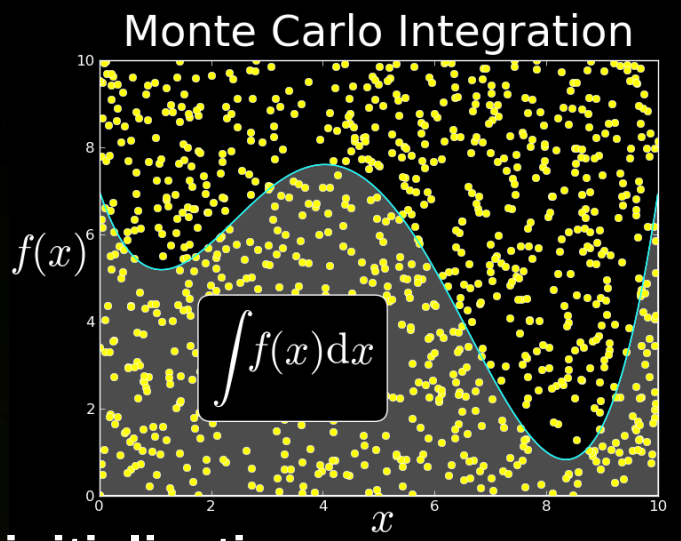
CURAND Library

- Library for generating random numbers



- Features:
  - XORWOW pseudo-random generator
  - Sobol' quasi-random number generators
  - Host API for generating random numbers in bulk
  - Inline implementation allows use inside GPU functions/kernels
  - Single- and double-precision, uniform, normal and log-normal distributions

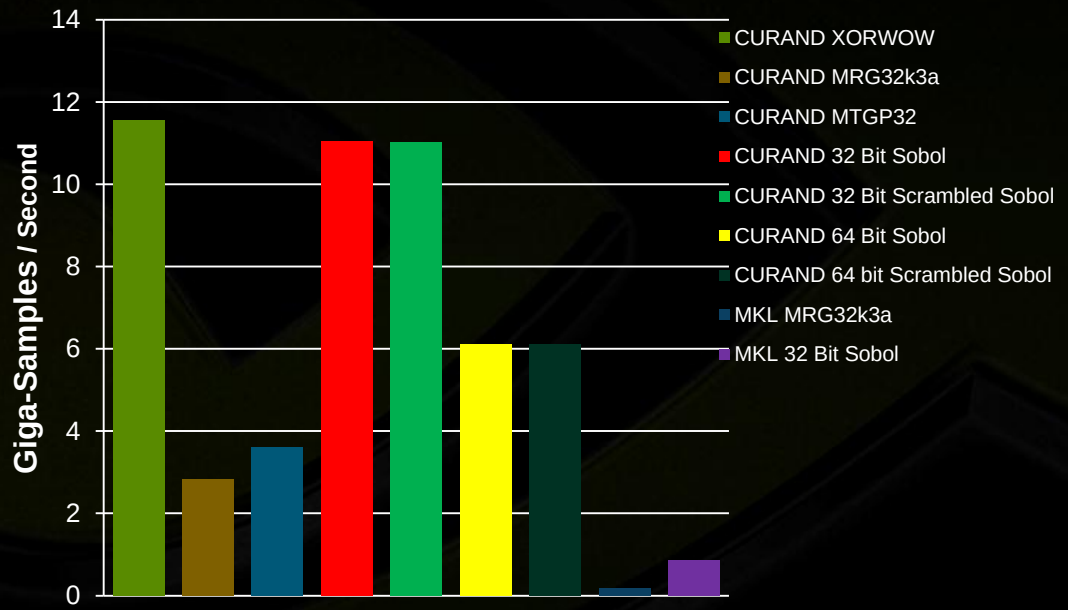
cuRAND: Random Number Generation



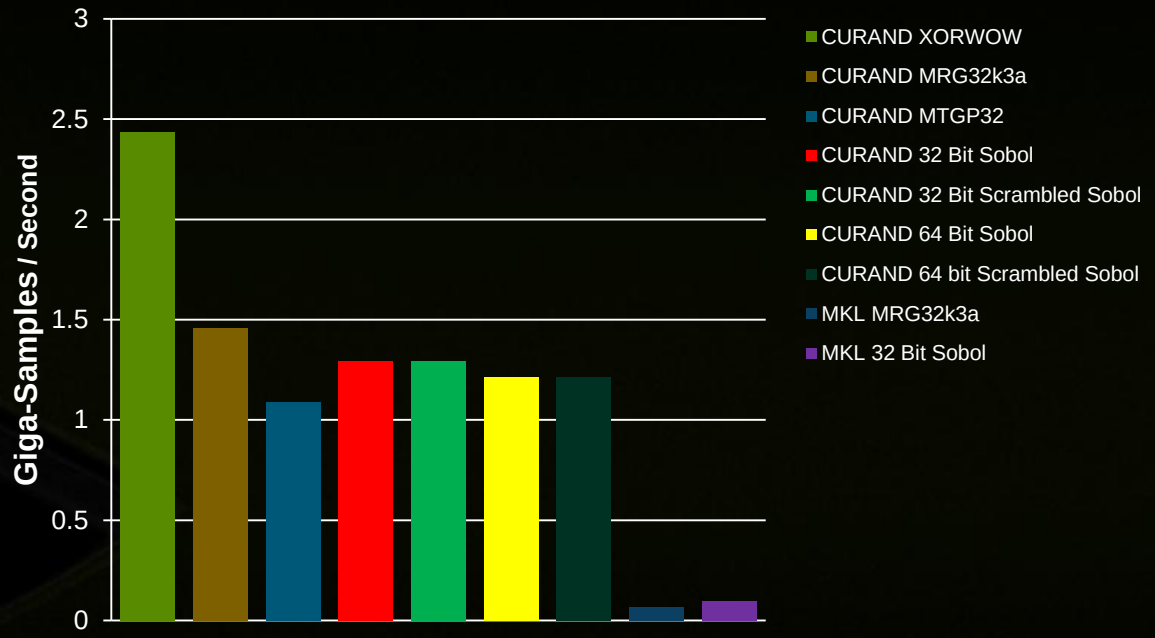
- New in CUDA 4.0/4.1
  - Scrambled and 64-bit Sobol'
  - Log-normal distribution
  - New parallel ordering supports faster XORWOW initialization
  - Results of cuRAND generators against standard statistical test batteries are reported in documentation

cuRAND Performance

cuRAND vs MKL
Double Precision, Uniform Distribution



cuRAND vs MKL
Double Precision, Normal Distribution



Performance may vary based on OS version and motherboard configuration

•cuRAND 4.1, Tesla M2090 (Fermi), ECC on
• MKL 10.2.3, TYAN FT72-B7015Xeon x5680 @ 3.33 GHz

Thrust: CUDA C++ Template Library

- Template library for CUDA mimics the C++ STL
  - Optimized algorithms for sort, reduce, scan, etc.
  - OpenMP backend for portability
- Allows applications and prototypes to be built ***quickly***
- New in 4.1: Boost-style placeholders allow inline functors
  - Example: *saxpy* in 1 line:

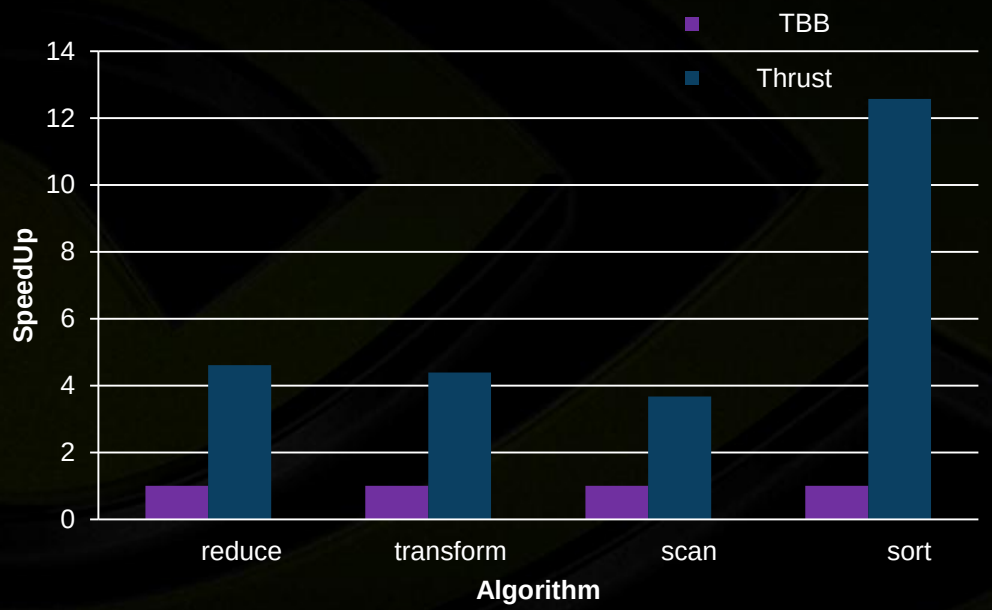
```
thrust::transform(x.begin(), x.end(), y.begin(), y.begin(), a * _1 + _2);
```

Thrust Example

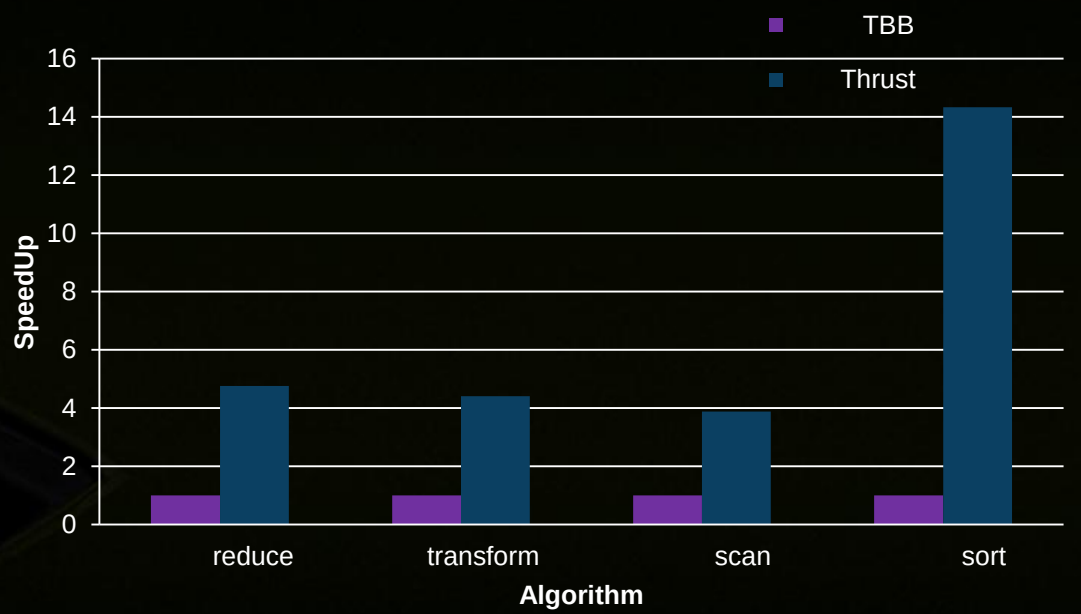
```
#include <thrust/host_vector.h>
#include <thrust/device_vector.h>
#include <thrust/sort.h>
#include <cstdlib.h>

int main(void)
{
    // generate 32M random numbers on the host
    thrust::host_vector<int> h_vec(32 << 20);
    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 (846M keys per sec on GeForce GTX 480)
    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;
}
```

Thrust Algorithm Performance

Various Algorithms (32M int.) Speedup compared to Intel TBB



Various Algorithms (32M float.) Speedup compared to Intel TBB

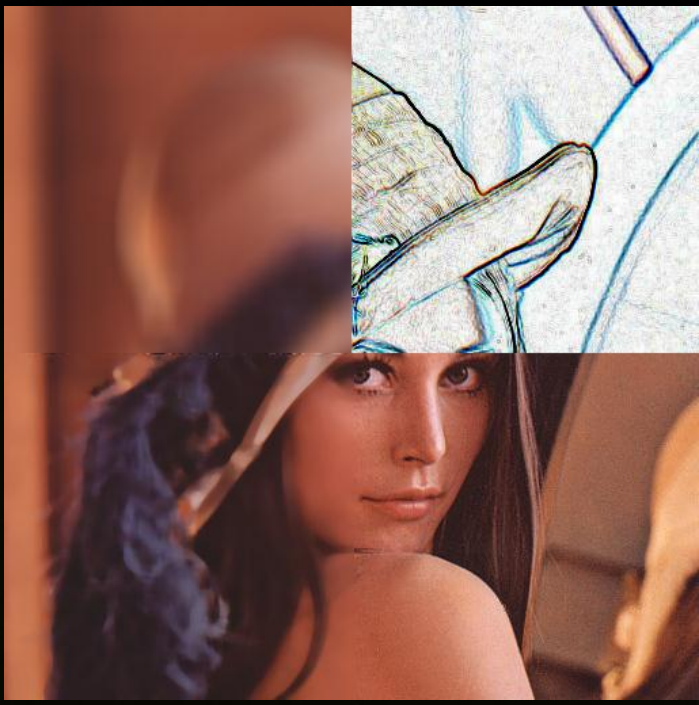


- Thrust 4.1, cuBLAS 4.1, Tesla M2090 (Fermi), ECC on
- TYAN FT72-B7015Xeon x5680 @ 3.33 GHz

NVIDIA Performance Primitives

Up to **40x** speedups

- Arithmetic, Logic, Conversions, Filters, Statistics, etc.
- Majority of primitives 5x to 10x faster than analogous routines in Intel IPP
- 1,000+ new image primitives in 4.1



* NPP 4.1, NVIDIA M2090 (Fermi)
* IPP 6.1, TYAN FT72-B7015Xeon x5680 @ 3.33 GHz

Image Processing Primitives

- Data exchange & initialization
  - Set, Convert, CopyConstBorder, Copy, Transpose, SwapChannels
- Arithmetic & Logical Ops
  - Add, Sub, Mul, Div, AbsDiff
- Threshold & Compare Ops
  - Threshold, Compare
- Color Conversion
  - RGB To YCbCr (& vice versa), ColorTwist, LUT_Linear
- Filter Functions
  - FilterBox, Row, Column, Max, Min, Dilate, Erode, SumWindowColumn/Row
- Geometry Transforms
  - Resize , Mirror, WarpAffine/Back/Quad, WarpPerspective/Back/Quad
- Statistics
  - Mean, StdDev, NormDiff, MinMax, Histogram, SqrIntegral, RectStdDev
- Segmentation
  - Graph Cut

math.h: C99 floating-point library + extras

- Basic: +, *, /, 1/, sqrt, FMA (all IEEE-754 accurate for float, double, all rounding modes)
- Exponentials: exp, exp2, log, log2, log10, ...
- Trigonometry: sin, cos, tan, asin, acos, atan2, sinh, cosh, asinh, acosh, ...
- Special functions: lgamma, tgamma, erf, erfc
- Utility: fmod, remquo, modf, trunc, round, ceil, floor, fabs, ...
- Extras: rsqrt, rcbrt, exp10, sinpi, sincos, cospi, erfinv, erfcinv, ...

- **New in 4.1**
  - **Bessel functions: j0, j1, jn, y0, y1, yn**
  - **Scaled complementary error function: erfcx**
  - **Average and rounded average: __{u}hadd, __{u}rhadd**

NVIDIA CUDA Libraries

CUDA Toolkit includes several libraries:

- **cuFFT: Fourier transforms**
- **cuBLAS: Dense Linear Algebra**
- **cuSPARSE : Sparse Linear Algebra**
- **cuRAND: Pseudo-random and Quasi-random numbers**
- **Thrust : STL-Like Primitives Library**
- **NPP: Image and Signal Processing**
- **LIBM: Standard C Math library**

Several open source and commercial* libraries:

- **MAGMA: Linear Algebra**
- **CULA Tools*: Linear Algebra**
- **CUSP: Sparse Linear Solvers**
- **CUDPP: Parallel Primitives Library**
- **And many more…**

- **OpenVidia: Computer Vision**
- **OpenCurrent: CFD**
- **NAG*: Computational Finance**


Questions?
NVIDIA