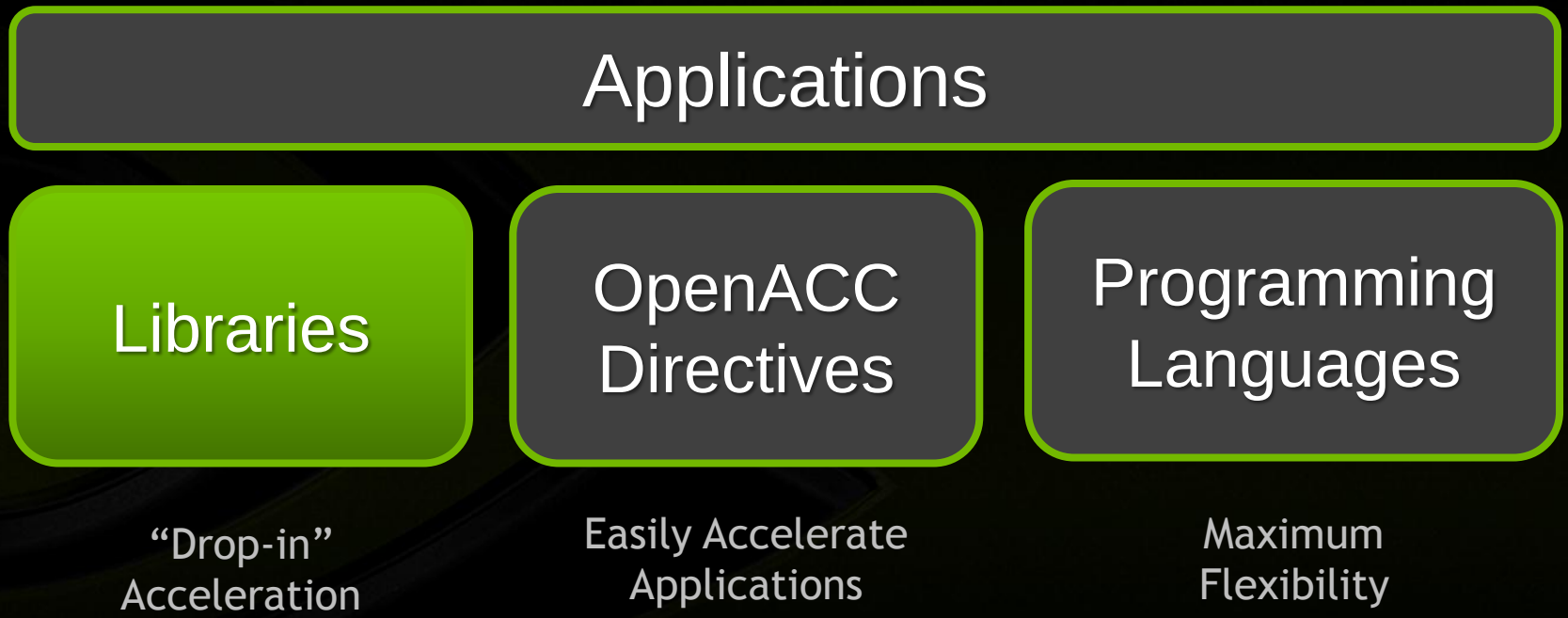
Fast GPU Development with CUDA Libraries

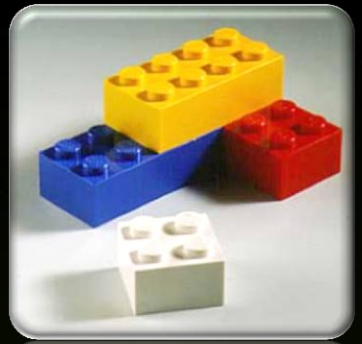
Jeff Larkin

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

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

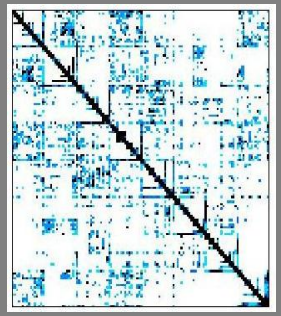
Free download @ www.nvidia.com/getcuda

Linear Algebra

Dense

cuBLAS

Sparse



cuSPARSE

Drop-In Acceleration

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: y[]=a*x[]+y[]
saxpy(N, 2.0, d_x, 1, d_y, 1);
```

Drop-In Acceleration (Step 1)

```
int N = 1 << 20;
```

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

Add “cublas” prefix and use device variables

Drop-In Acceleration (Step 2)

```
int N = 1 << 20;
cublasInit();
```

Initialize CUBLAS

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cublasShutdown();
```

Shut down CUBLAS

Drop-In Acceleration (Step 3)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void*)&d_y);
```

Allocate device vectors

```
// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);
```

```
cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Deallocate device vectors

Drop-In Acceleration (Step 4)

```
int N = 1 << 20;
cublasInit();
cublasAlloc(N, sizeof(float), (void**)&d_x);
cublasAlloc(N, sizeof(float), (void**)&d_y);

cublasSetVector(N, sizeof(x[0]), x, 1, d_x, 1);
cublasSetVector(N, sizeof(y[0]), y, 1, d_y, 1);

// Perform SAXPY on 1M elements: d_y[]=a*d_x[]+d_y[]
cublasSaxpy(N, 2.0, d_x, 1, d_y, 1);

cublasGetVector(N, sizeof(y[0]), d_y, 1, y, 1);

cublasFree(d_x);
cublasFree(d_y);
cublasShutdown();
```

Transfer data to GPU

Read data back GPU

cuBLAS Interface

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

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

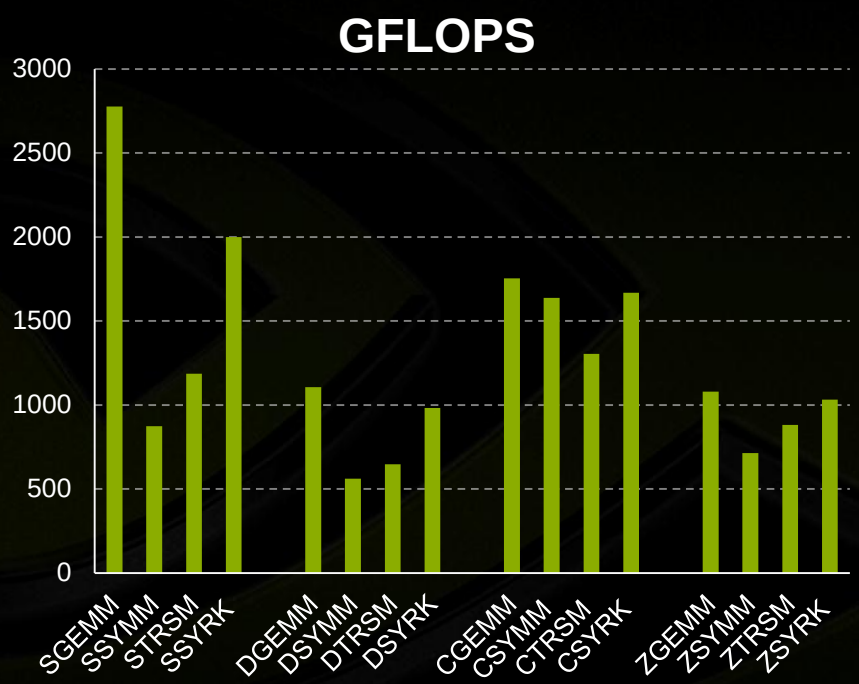
```
dgemm('N','N',m,n,k,A,m,
      B,k,0.0,C,m)
```

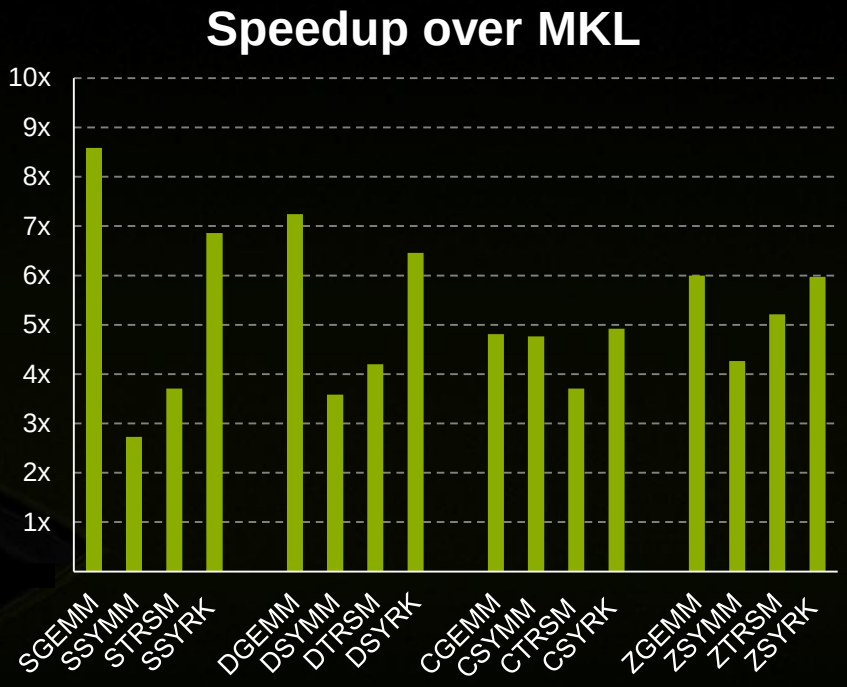
```
err = cublasIdamax(hdl, n, col, 1, size);
```

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

```
cublasDgemm(hdl,'N','N',m,n,k,d_A,m,
      d_B,k,0.0,d_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**

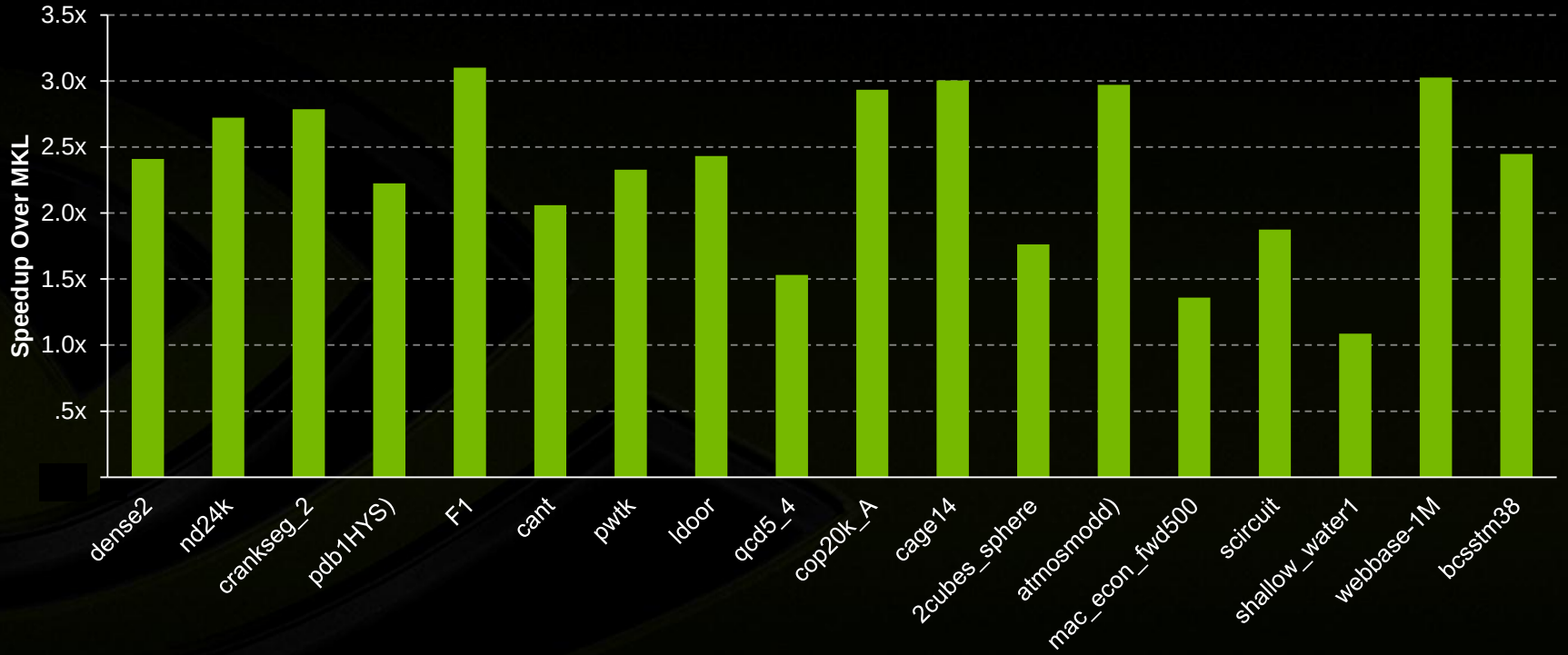
cuSPARSE Interface

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

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

cuSPARSE performance

Sparse Matrix x Dense Vector

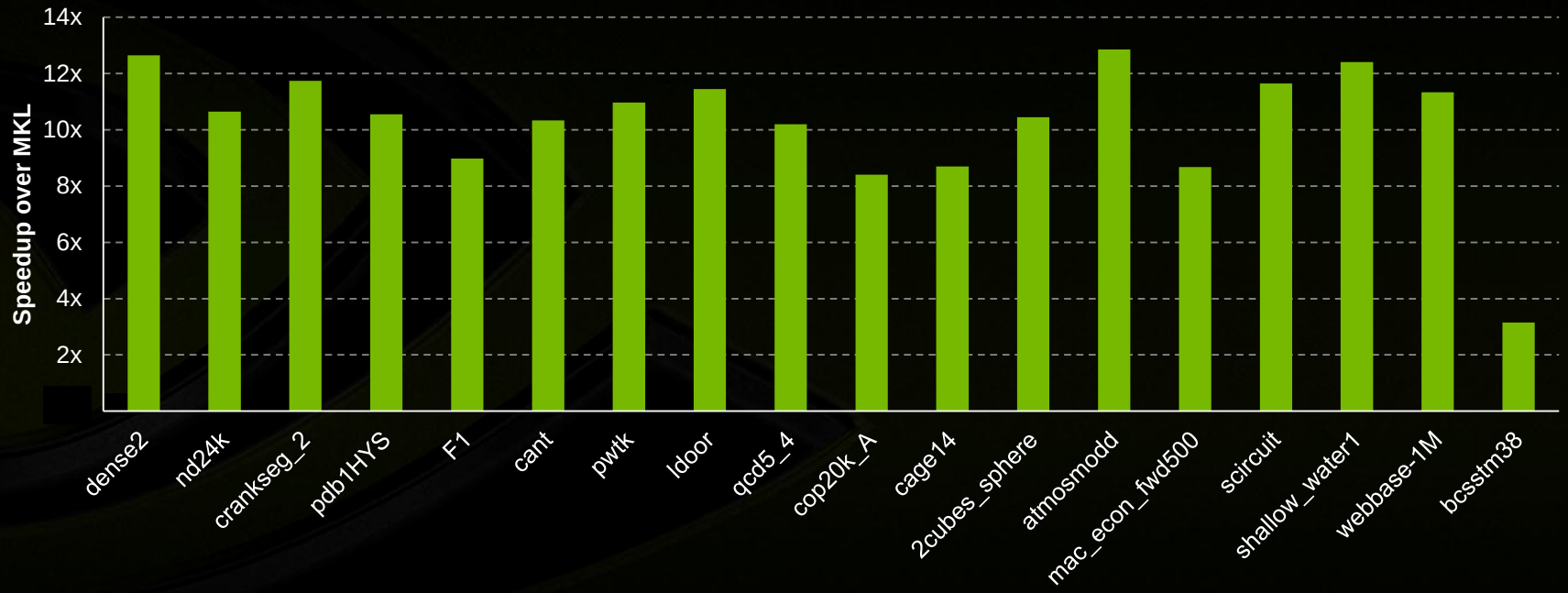


- Average of s/d/c/z routines
- cuSPARSE 5.0 on K20X, input and output data on device
- MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

Performance may vary based on OS version and motherboard configuration

cuSPARSE: up to 12x Faster than MKL

Sparse Matrix x 6 Dense Vectors
(useful for block iterative solvers)

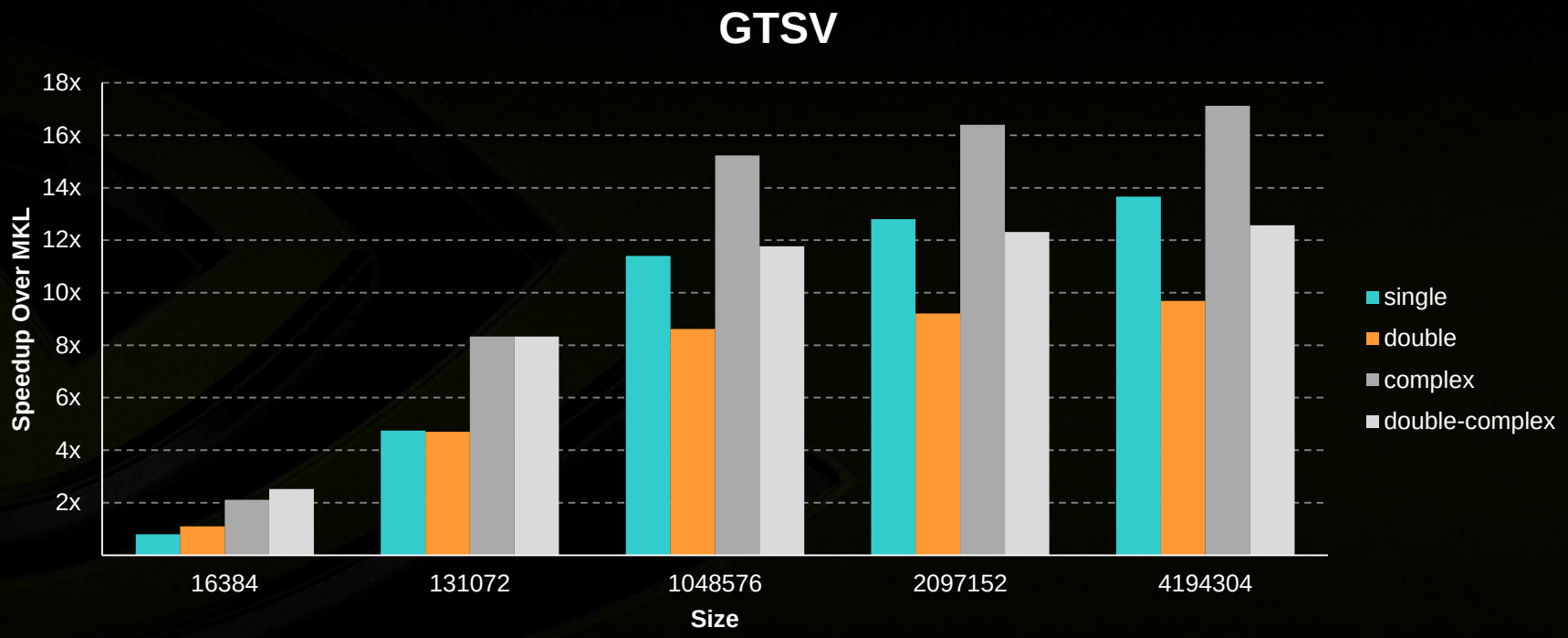


- Average of s/d/c/z routines
- cuSPARSE 5.0 on K20X, input and output data on device
- MKL 10.3.6 on Intel SandyBridge E5-2687W @ 3.10GHz

Performance may vary based on OS version and motherboard configuration

Tri-diagonal Solver Performance vs. MKL

Speedup for Tri-Diagonal solver (gtsv)



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

Performance may vary based on OS version and motherboard configuration

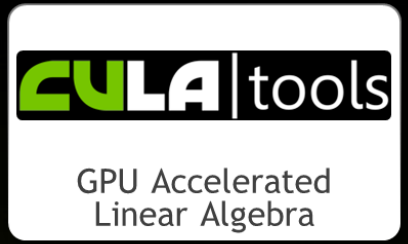
Different Approaches to Linear Algebra

- 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)
```

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

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



EM Photonics

AccelerEyes

Different Approaches to Linear Algebra (cont.)

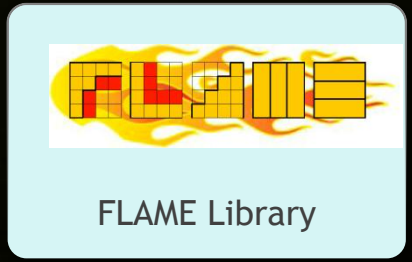
- MAGMA
  - LAPACK conforming API
  - Magma BLAS and LAPACK
  - High performance by utilizing both GPU and CPU

```
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)
```

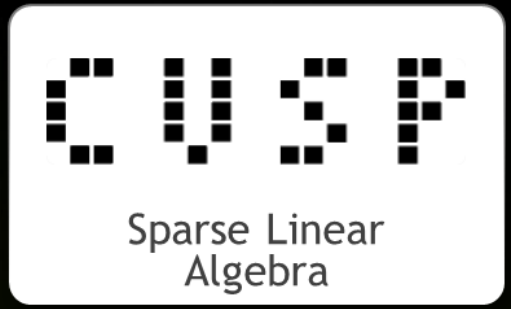


UT-Austin

Different Approaches to Linear Algebra (cont.)

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

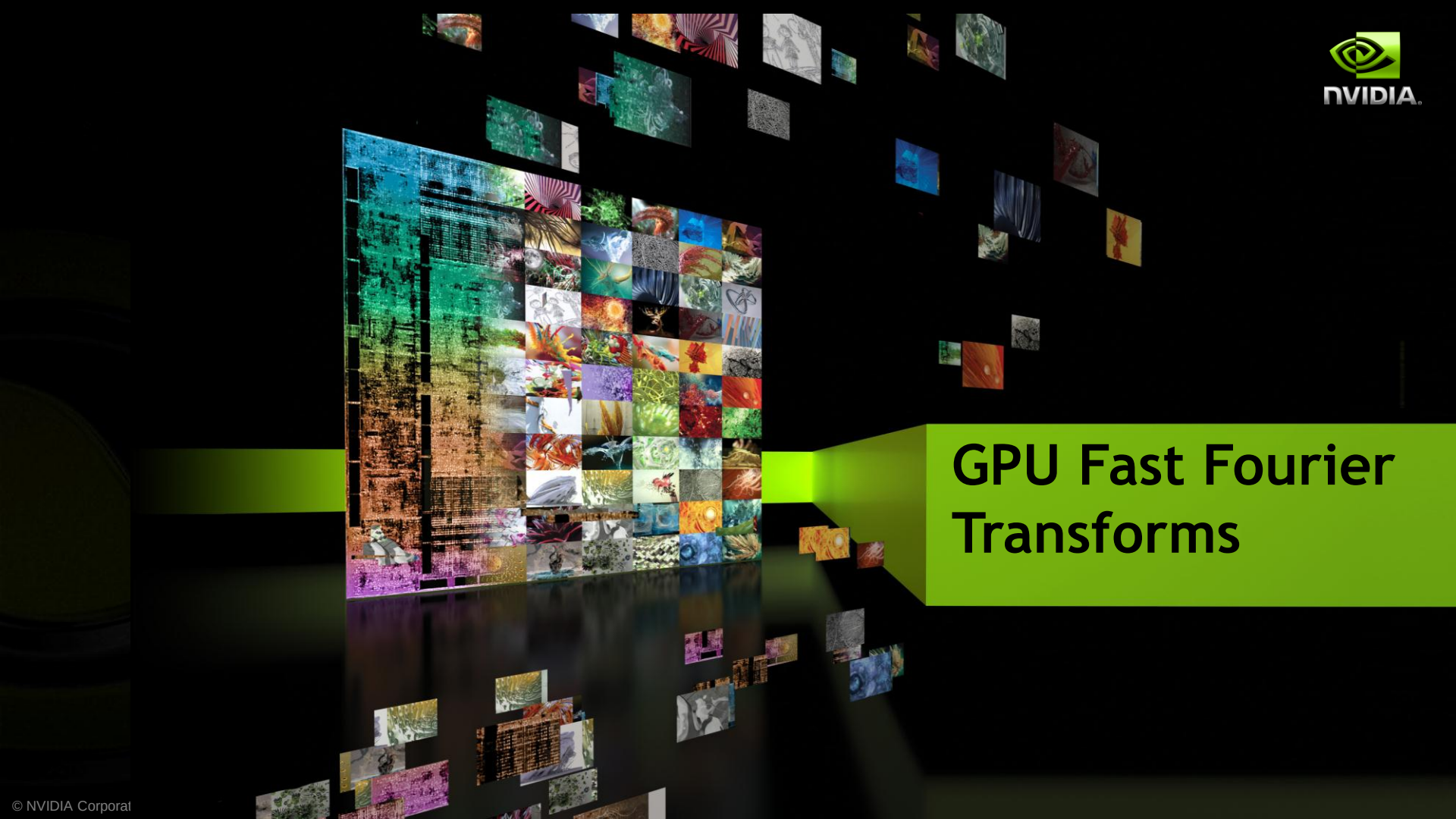
```
cusp::krylov::cg(A, x, b);
```



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



cuFFT

- Interface modeled after FFTW

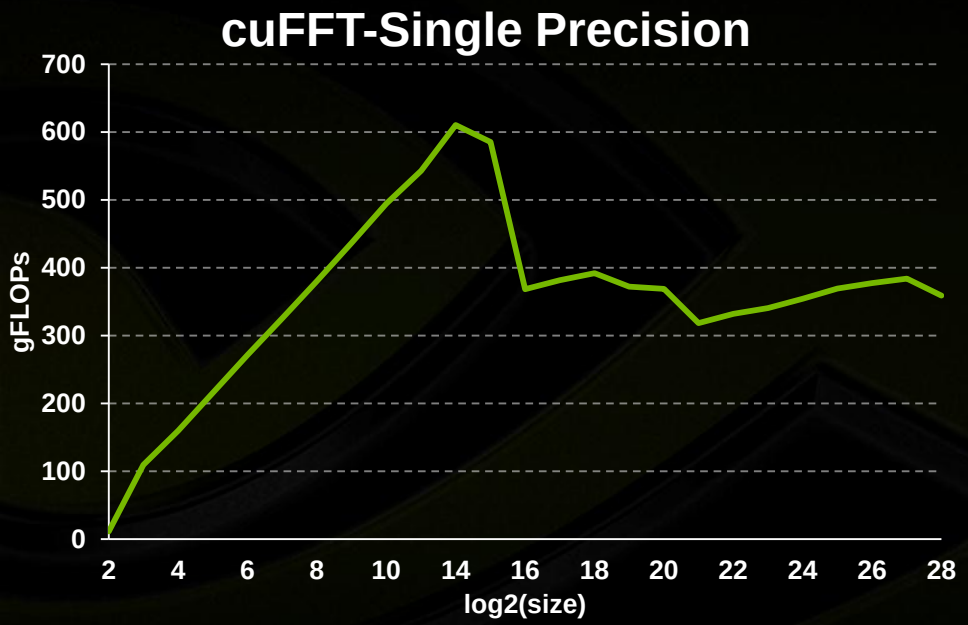
```
fftw_plan PlanA;

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

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

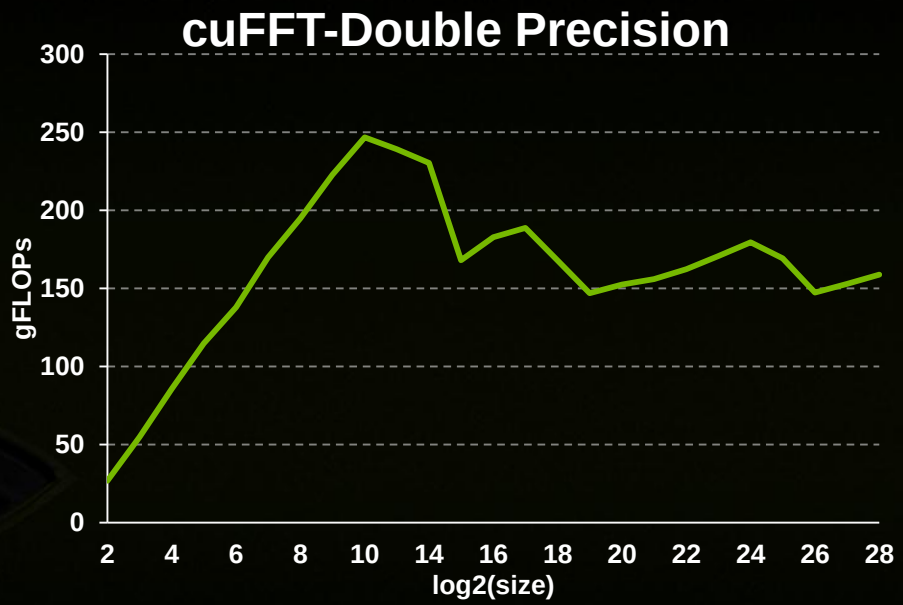
```
cufftPlan2d PlanA;

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

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

- 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

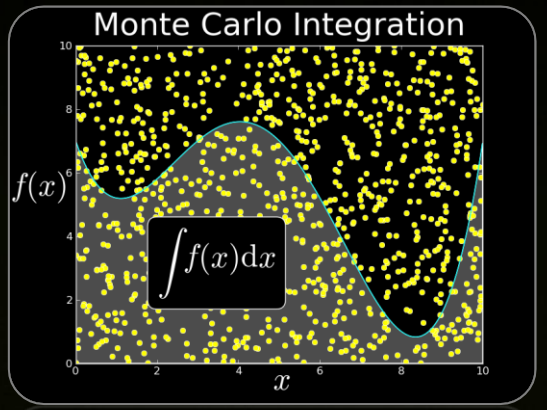




- CUFFT 5.0.30 on K20X, input and output data on device

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);
```

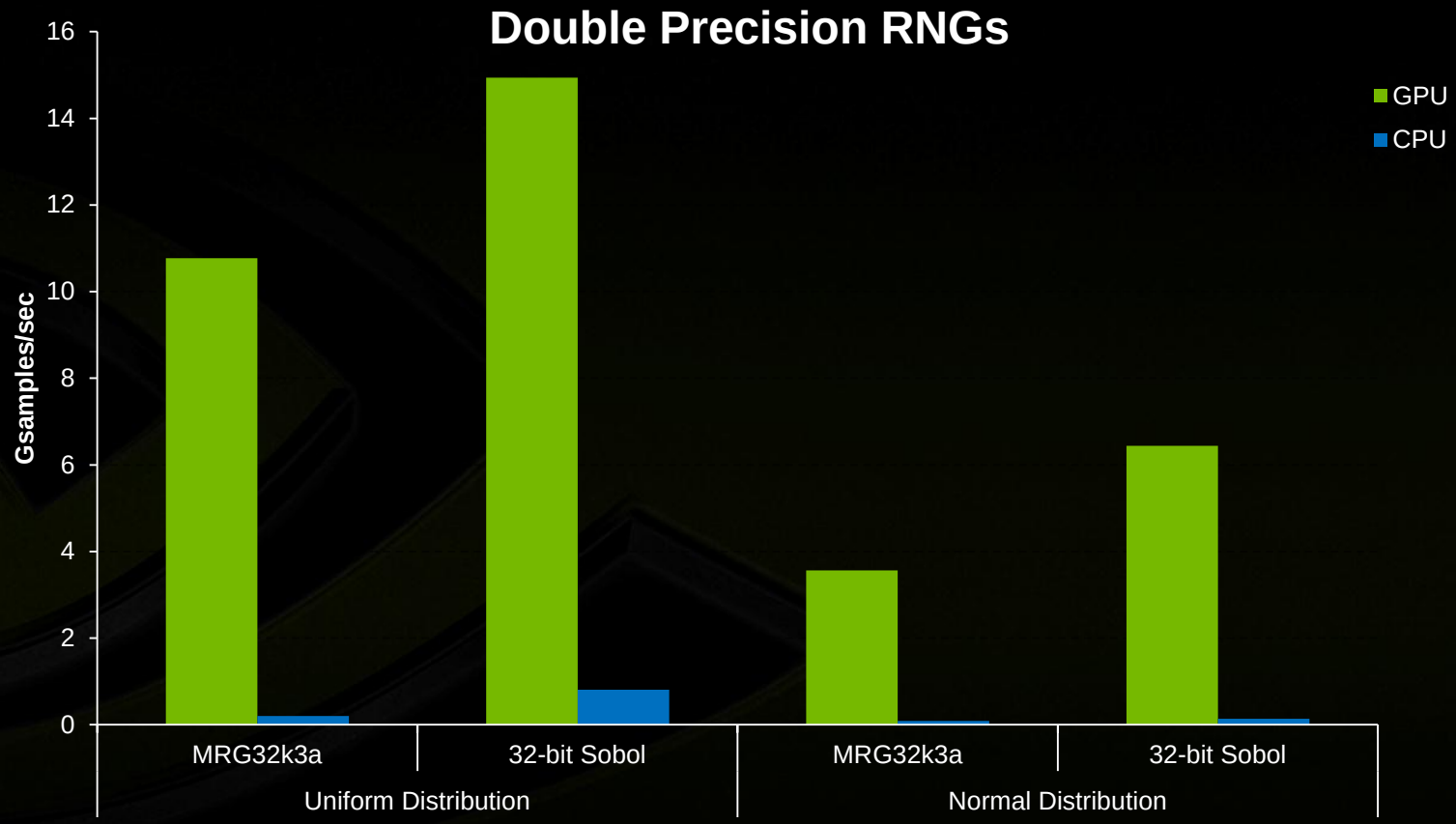
Generate set of random numbers at once

- **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**

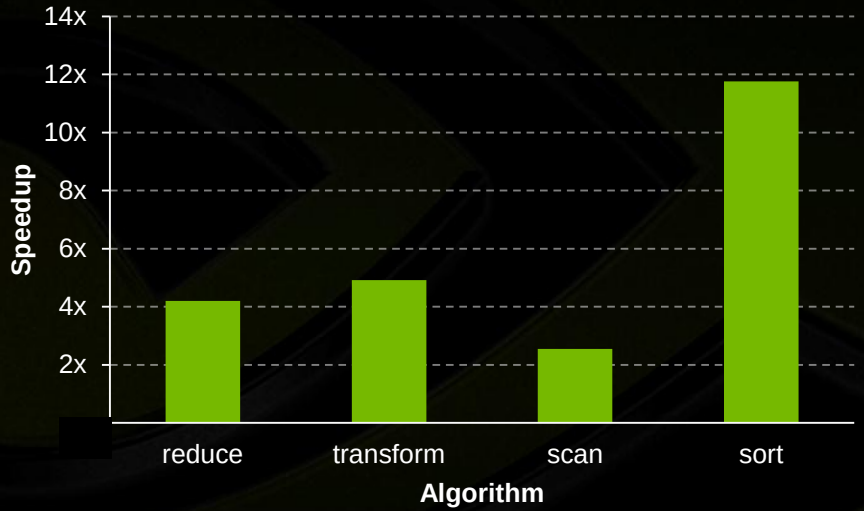
```
thrust::fill(D.begin(), D.begin()+5, 42.f);
float* raw_ptr = thrust::raw_pointer_cast(D);
```

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

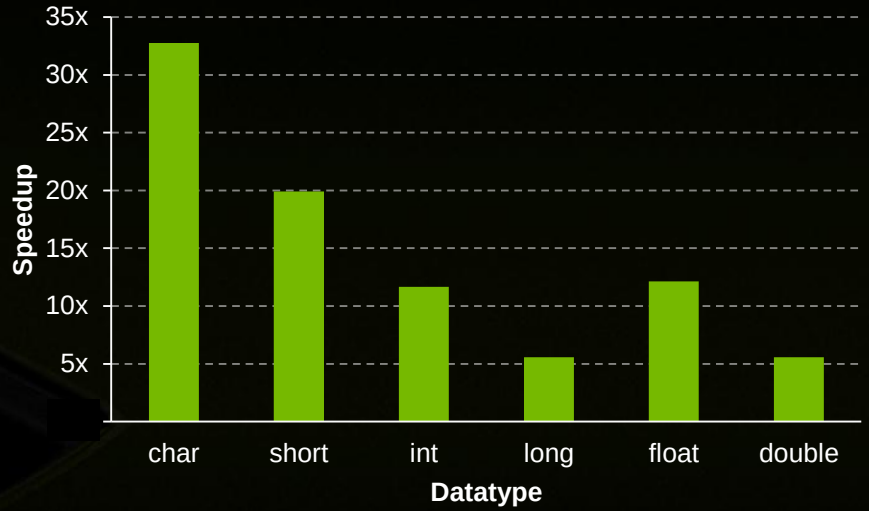
```
thrust::transform(D1.begin(), D1.end(), D2.begin(), D2.end(),
thrust::plus<float>());   // D2 = D1 + D2
```

Thrust Performance

Various Algorithms (32M int.) Speedup over TBB



Sort (32M samples) Speedup over TBB



- Thrust 5.0 on K20X, input and output data on device
- TBB 4.1 on Intel SandyBridge E5-2687W @3.10GHz

Performance may vary based on OS version and motherboard configuration

math.h: C99 floating-point library + extras

CUDA math.h is industry proven, high performance, accurate

- 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 CUDA 5.0
  - sincospi[f]() and normcdf[inv][f]()
  - sin(), cos() and erfcinvf() more accurate and faster
  - Full list of new features and optimizations:

    http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#math

    http://docs.nvidia.com/cuda/cuda-toolkit-release-notes/index.html#math-performance-improvements

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];
```

Explore the CUDA (Libraries) Ecosystem

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

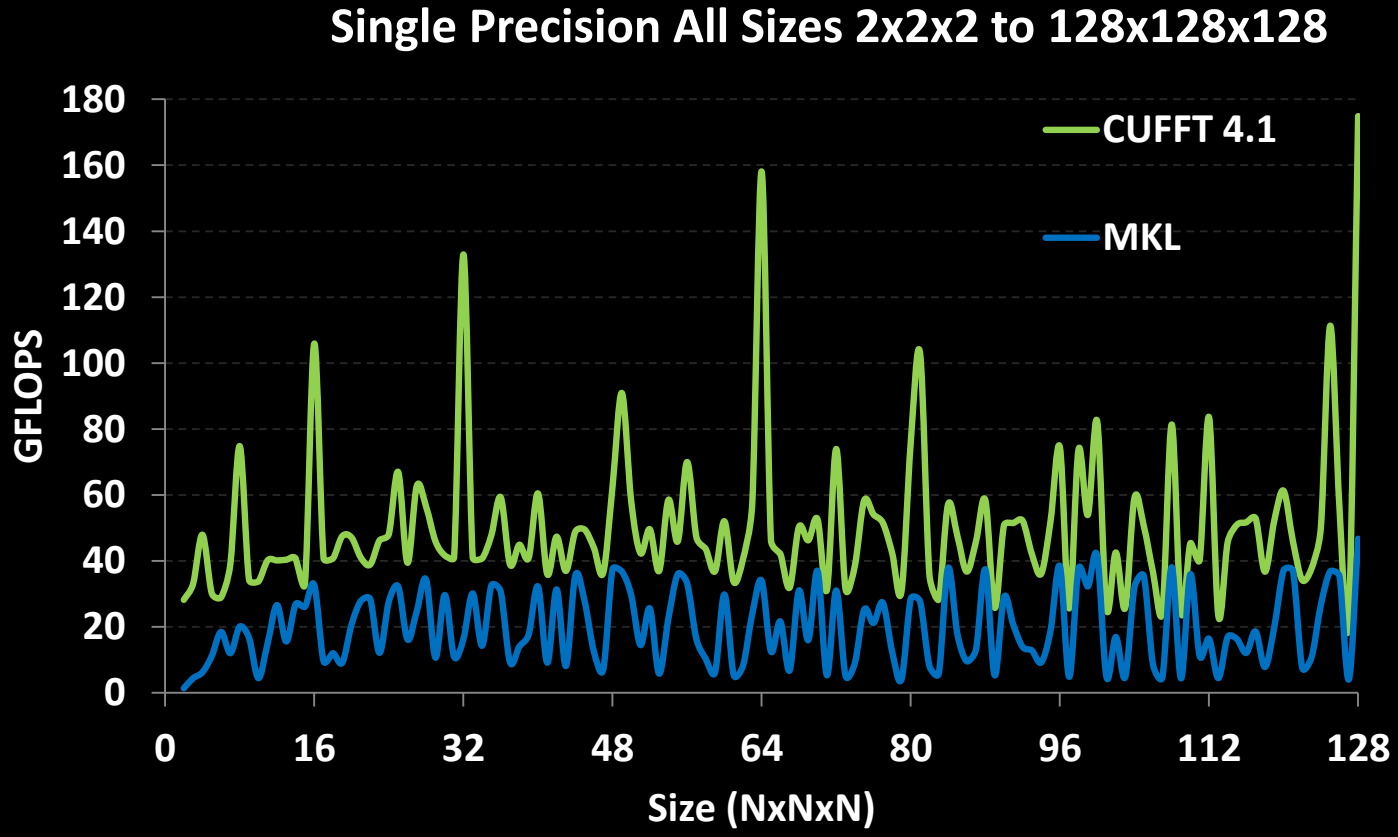
developer.nvidia.com/cuda-tools-ecosystem

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


Thank you
NVIDIA

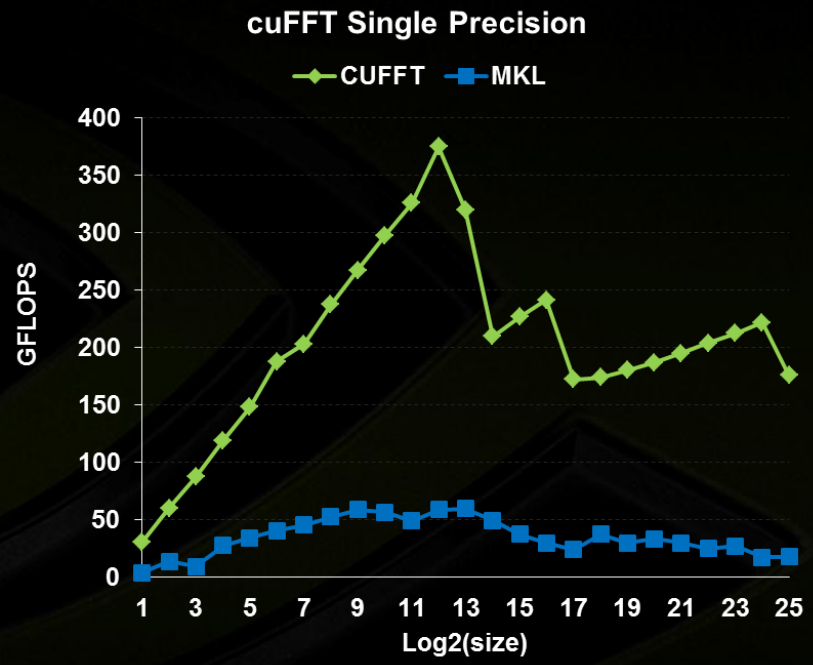
Optimized 3D transforms

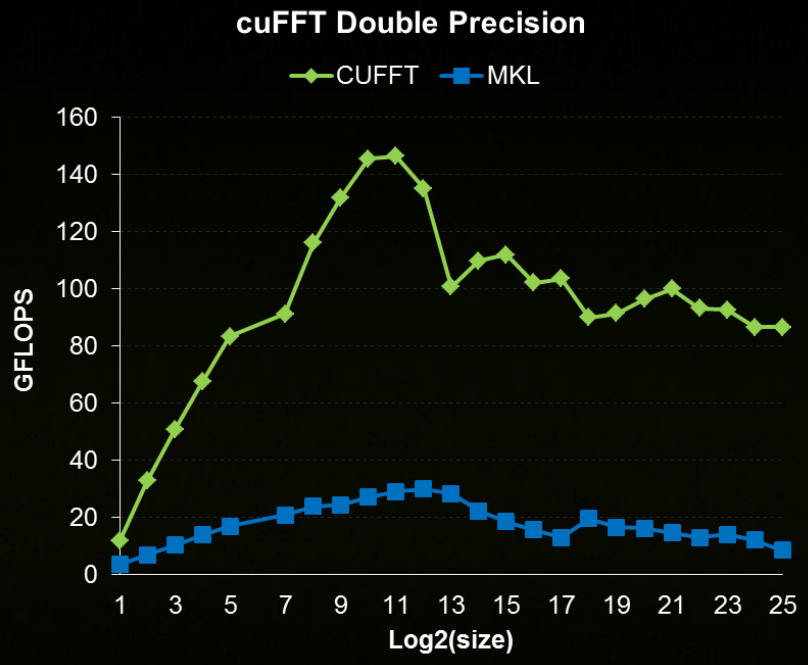


Performance may vary based on OS version and motherboard configuration

- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

cuFFT Performance -1D





Performance may vary based on OS version and motherboard configuration

- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz