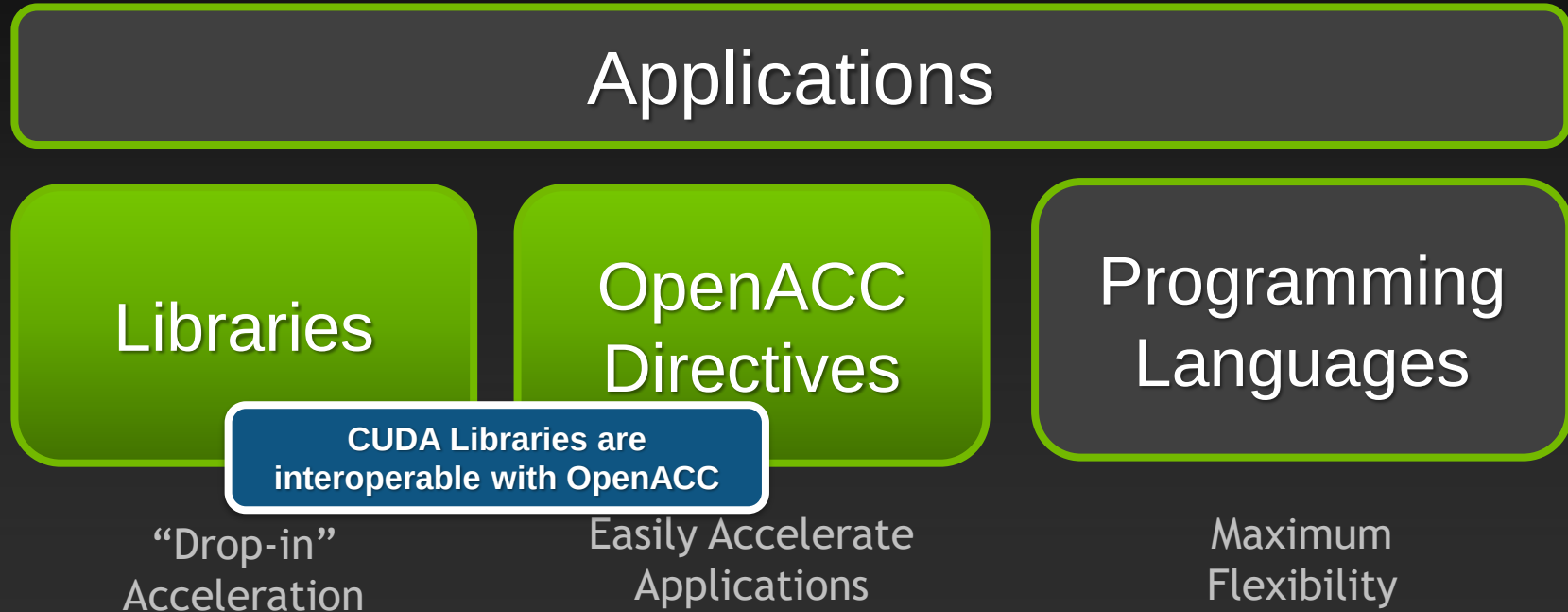


Using OpenACC With CUDA Libraries

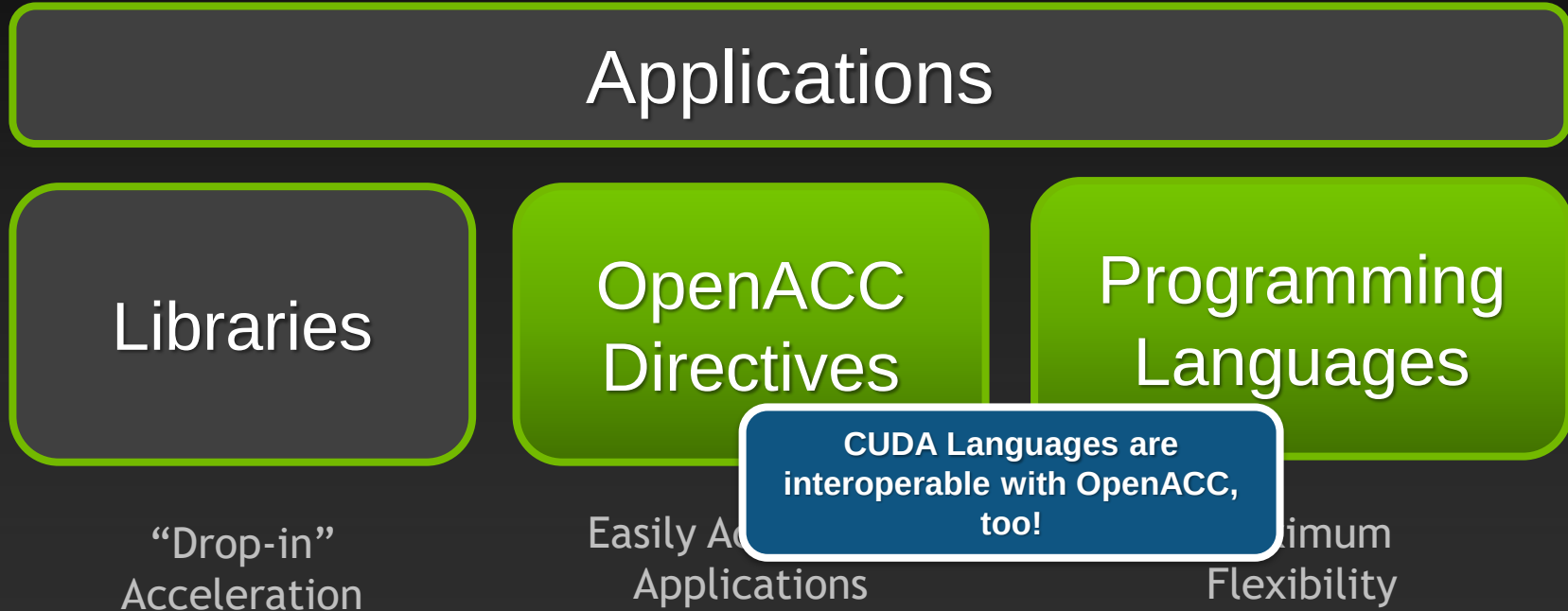
John Urbanic
with NVIDIA

Pittsburgh Supercomputing Center

3 Ways to Accelerate Applications

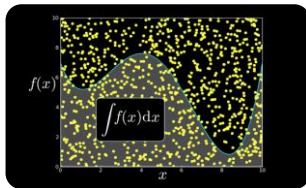


3 Ways to Accelerate Applications

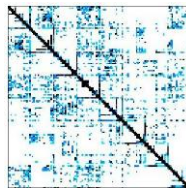




NVIDIA cuBLAS



NVIDIA cuRAND



NVIDIA cuSPARSE



NVIDIA NPP

GPU VSIPL

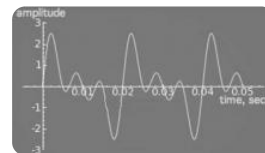
Vector Signal
Image Processing

CULA | tools

GPU Accelerated
Linear Algebra



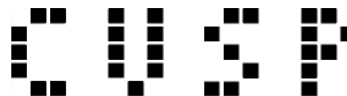
Matrix Algebra on
GPU and Multicore



NVIDIA cuFFT



Building-block
Algorithms for CUDA



Sparse Linear
Algebra



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

Always more available at NVIDIA Developer site.

How To Use CUDA Libraries With OpenACC

CUDA data in OpenACC

You have to allocate data memory on the host and device with `alloc/cudaMalloc`. `deviceptr()` lets OpenACC know that has happened.

```
float *a;
...
err = cudaMalloc(&a, sizeof(float)*n);
kernel<<<n/32,32>>>(a,...);
...
incr(a,n);

void incr(float* x, int n){
    #pragma acc parallel loop deviceptr(x)
    for (int i = 0; i < n; ++i)
        x[i] += 1.0f;
}
```

deviceptr Data Clause

`deviceptr( list )` Declares that the pointers in *list* refer to device pointers that need not be allocated or moved between the host and device for this pointer.

Example:

C

```
#pragma acc data deviceptr(d_input)
```

Fortran

```
$!acc data deviceptr(d_input)
```


host_data Construct

If the data is on the device - say it has been *create()*ed - then `host_data use_device()` allows us to grab that device pointer on the host so that we can pass it along to some CUDA routine elsewhere.

```
a = (float*)malloc(sizeof(float)*n);
#pragma acc data create(a[0:n])
{
    #pragma acc host_data use_device(a)
    {
        incr(a,n);
    }
}

----- separate file with CUDA code -----
__global__ inckernel(float* x, int n){ ... }

void incr(float* x, int n){
    inckernel<<<n/32,n>>>(x,n);
}
```

Example: 1D convolution using CUFFT

- Perform convolution in frequency space
 1. Use CUFFT to transform input signal and filter kernel into the frequency domain
 2. Perform point-wise complex multiply and scale on transformed signal
 3. Use CUFFT to transform result back into the time domain
- We will perform step 2 using OpenACC
- Code highlights follow. Code available with exercises in:
`Exercises/OpenACC/Cuffft-acc`

Source Excerpt

Allocating Data

```
// Allocate host memory for the signal and filter
Complex *h_signal = (Complex *)malloc(sizeof(Complex) * SIGNAL_SIZE);
Complex *h_filter_kernel = (Complex *)malloc(sizeof(Complex) * FILTER_KERNEL_SIZE);
.
.
.

// Allocate device memory for signal
Complex *d_signal;
checkCudaErrors(cudaMalloc((void **)&d_signal, mem_size));
// Copy host memory to device
checkCudaErrors(cudaMemcpy(d_signal, h_padded_signal, mem_size, cudaMemcpyHostToDevice));

// Allocate device memory for filter kernel
Complex *d_filter_kernel;
checkCudaErrors(cudaMalloc((void **)&d_filter_kernel, mem_size));
```

Source Excerpt

Sharing Device Data (d_signal, d_filter_kernel)

```
// Transform signal and kernel
error = cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal, CUFFT_FORWARD);
error = cufftExecC2C(plan, (cufftComplex *)d_filter_kernel, (cufftComplex *)d_filter_kernel, CUFFT_FORWARD);

// Multiply the coefficients together and normalize the result
printf("Performing point-wise complex multiply and scale.\n");
complexPointwiseMulAndScale(new_size, (float *restrict)d_signal, (float *restrict)d_filter_kernel);

// Transform signal back
error = cufftExecC2C(plan, (cufftComplex *)d_signal, (cufftComplex *)d_signal, CUFFT_INVERSE);
```

OpenACC
Routine

CUDA
Routines

OpenACC Convolution Code

```
void complexPointwiseMulAndScale(int n, float *restrict signal,  
                                float *restrict filter_kernel)  
{  
    // Multiply the coefficients together and normalize the result  
    #pragma acc data deviceptr(signal, filter_kernel)  
    {  
        #pragma acc kernels loop independent  
        for (int i = 0; i < n; i++) {  
            float ax = signal[2*i];  
            float ay = signal[2*i+1];  
            float bx = filter_kernel[2*i];  
            float by = filter_kernel[2*i+1];  
            float s = 1.0f / n;  
            float cx = s * (ax * bx - ay * by);  
            float cy = s * (ax * by + ay * bx);  
            signal[2*i] = cx;  
            signal[2*i+1] = cy;  
        }  
    }  
}
```

Implementation note: We cast the Complex* pointers to float* pointers and use interleaved indexing

Linking CUFFT

- `#include "cufft.h"`
- Compiler command line options:

```
CUDA_PATH = /opt/pgi/13.10.0/linux86-64/2013/cuda/5.0  
CCFLAGS = -I$(CUDA_PATH)/include -L$(CUDA_PATH)/lib64  
          -lcudart -lcufft
```

Must use
PGI-provided
CUDA toolkit paths

Must link libcudart
and libcufft

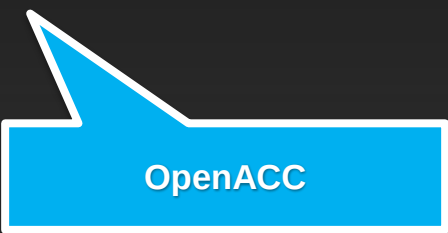
Result

```
instr009@nid27635:~/Cufft> aprun -n 1 cufft_acc  
Transforming signal cufftExecC2C  
Performing point-wise complex multiply and scale.  
Transforming signal back cufftExecC2C  
Performing Convolution on the host and checking correctness
```

```
Signal size: 500000, filter size: 33  
Total Device Convolution Time: 6.576960 ms (0.186368 for point-wise convolution)  
Test PASSED
```



CUFFT + cudaMemcpy



OpenACC

Summary

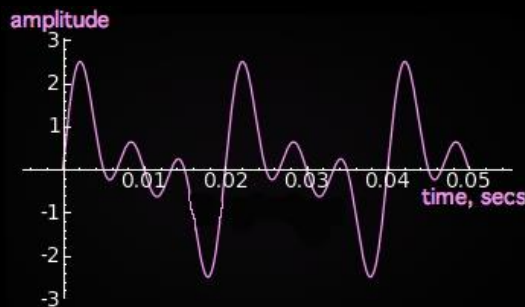
- Use deviceptr data clause to pass pre-allocated device data to OpenACC regions and loops
- Use host_data to get device address for pointers inside acc data regions
- The same techniques shown here can be used to share device data between OpenACC loops and
 - Your custom CUDA C/C++/Fortran/etc. device code
 - Any CUDA Library that uses CUDA device pointers

Appendix

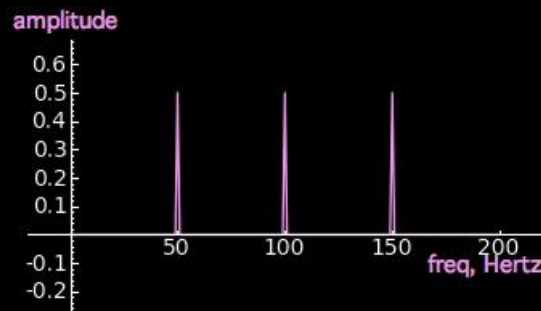
Compelling Cases For Various Libraries
Of Possible Interest To You

cuFFT: Multi-dimensional FFTs

- New in CUDA 4.1
 - Flexible input & output data layouts for all transform types
 - Similar to the FFTW “Advanced Interface”
 - Eliminates extra data transposes and copies
 - API is now thread-safe & callable from multiple host threads
 - Restructured documentation to clarify data layouts



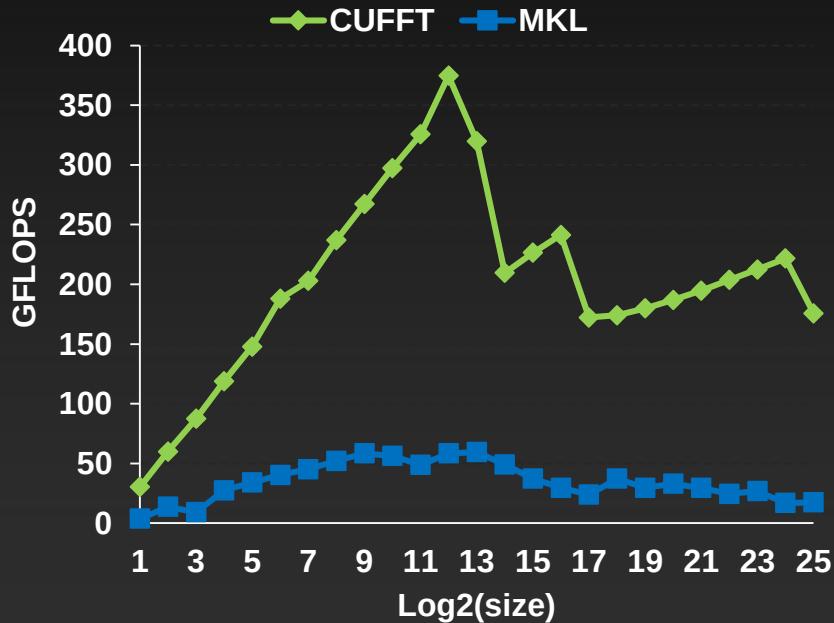
$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$
$$f(n) = \frac{1}{N} \sum_{x=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$



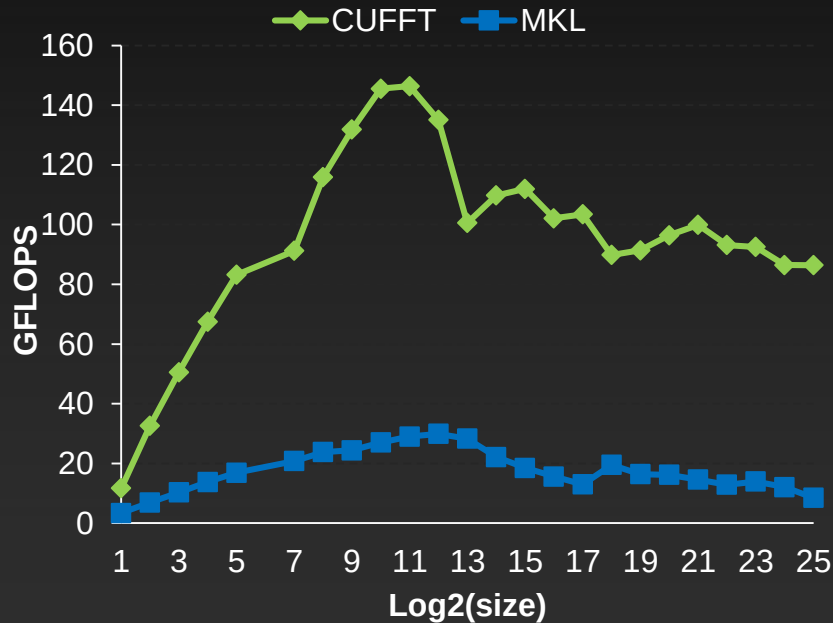
FFTs up to 10x Faster than MKL

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

cuFFT Single Precision



cuFFT Double Precision

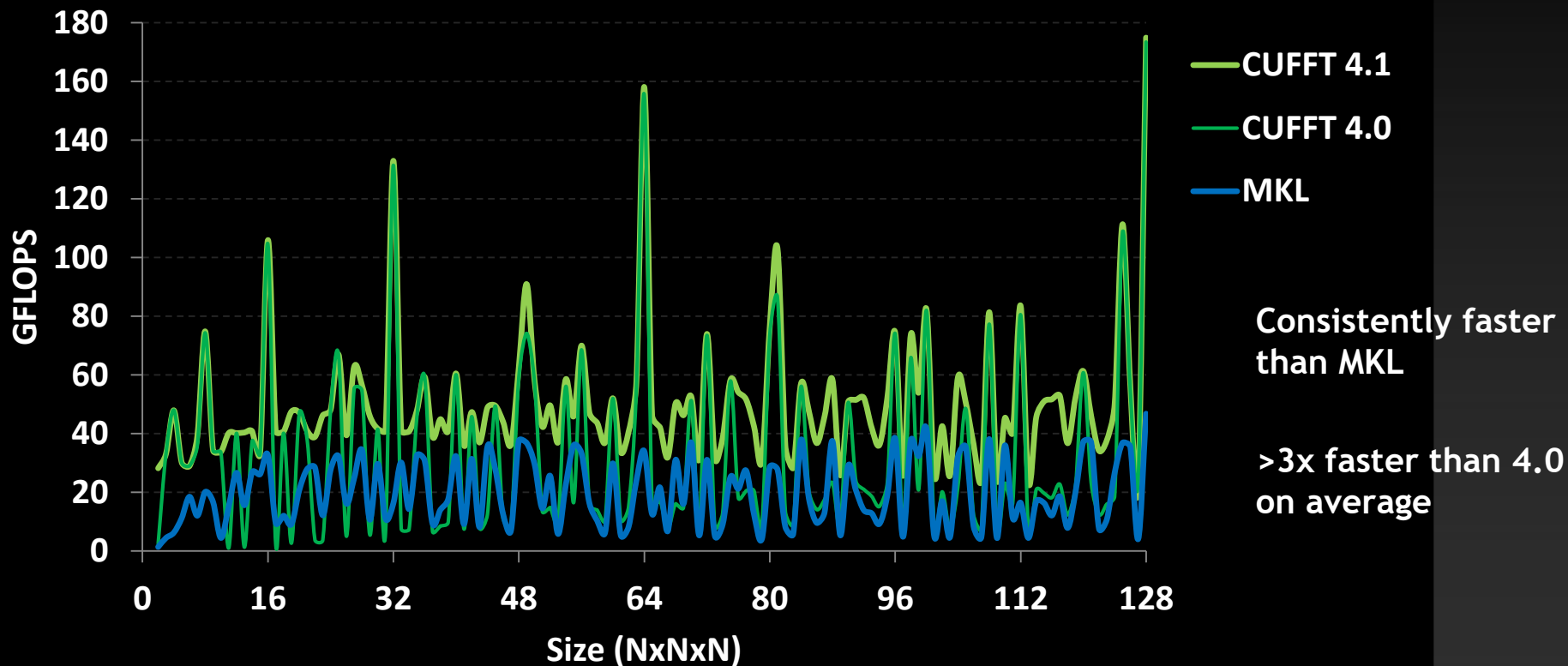


- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

Performance may vary based on OS version and motherboard configuration

CUDA 4.1 optimizes 3D transforms

Single Precision All Sizes 2x2x2 to 128x128x128



• cuFFT 4.1 on Tesla M2090, ECC on

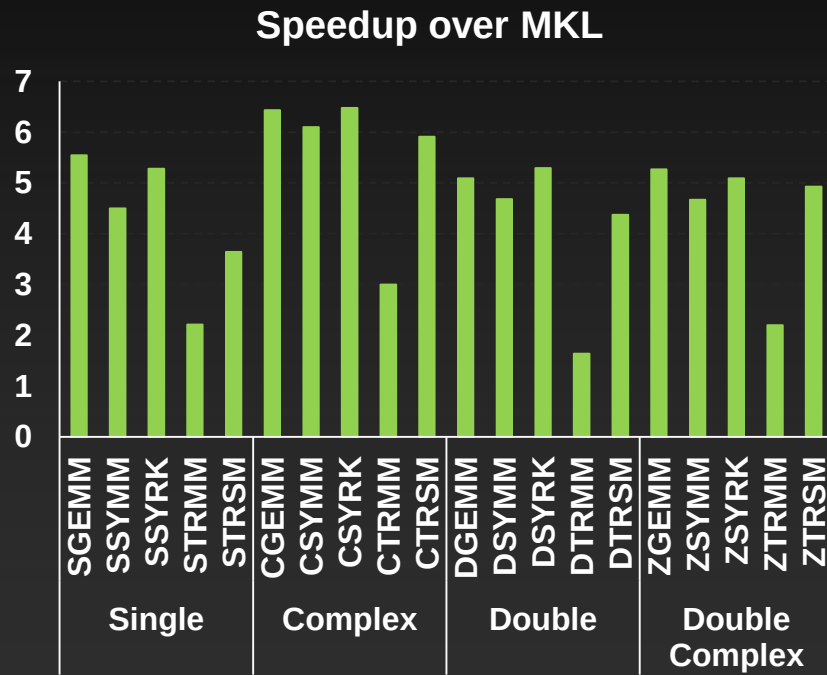
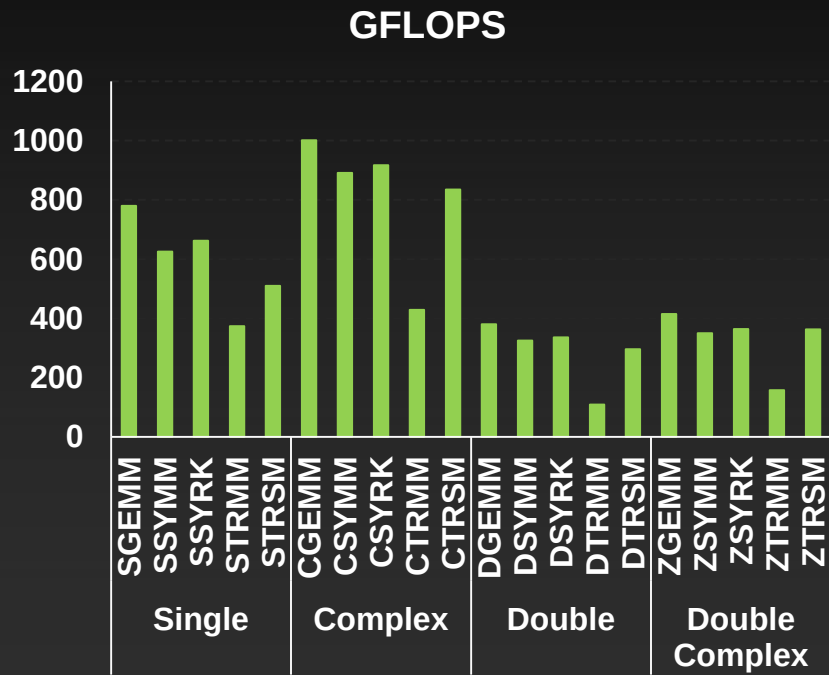
• MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

cuBLAS: Dense Linear Algebra on GPUs

- Complete BLAS implementation plus useful extensions
 - Supports all 152 standard routines for single, double, complex, and double complex
- New in CUDA 4.1
 - New batched GEMM API provides >4x speedup over MKL
 - Useful for batches of 100+ small matrices from 4x4 to 128x128
 - 5%-10% performance improvement to large GEMMs

cuBLAS Level 3 Performance

Up to 1 TFLOPS sustained performance and **>6x** speedup over Intel MKL

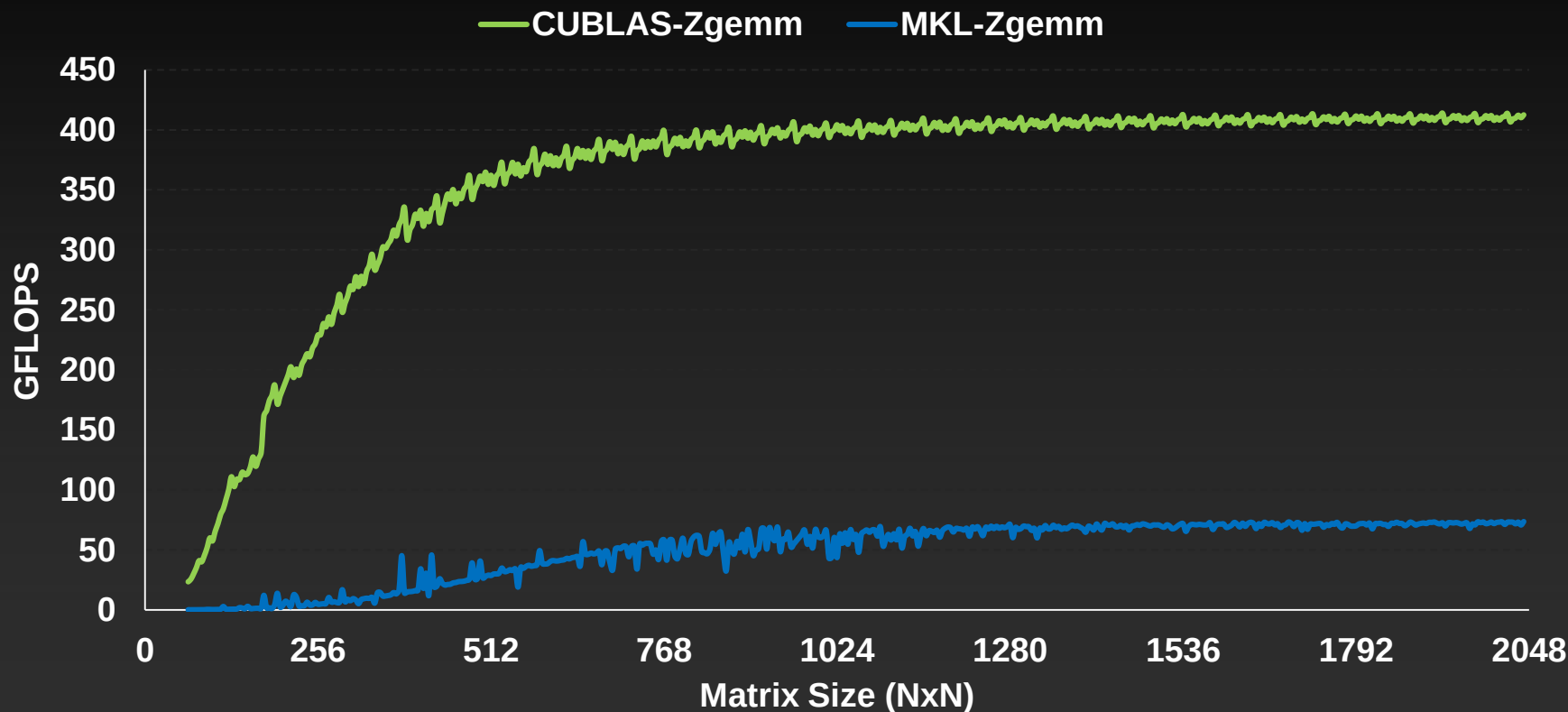


• 4Kx4K matrix size

• cuBLAS 4.1, Tesla M2090 (Fermi), ECC on

• MKL 10.2.3, TYAN FT72-R7015 Xeon x5680 Six-Core @

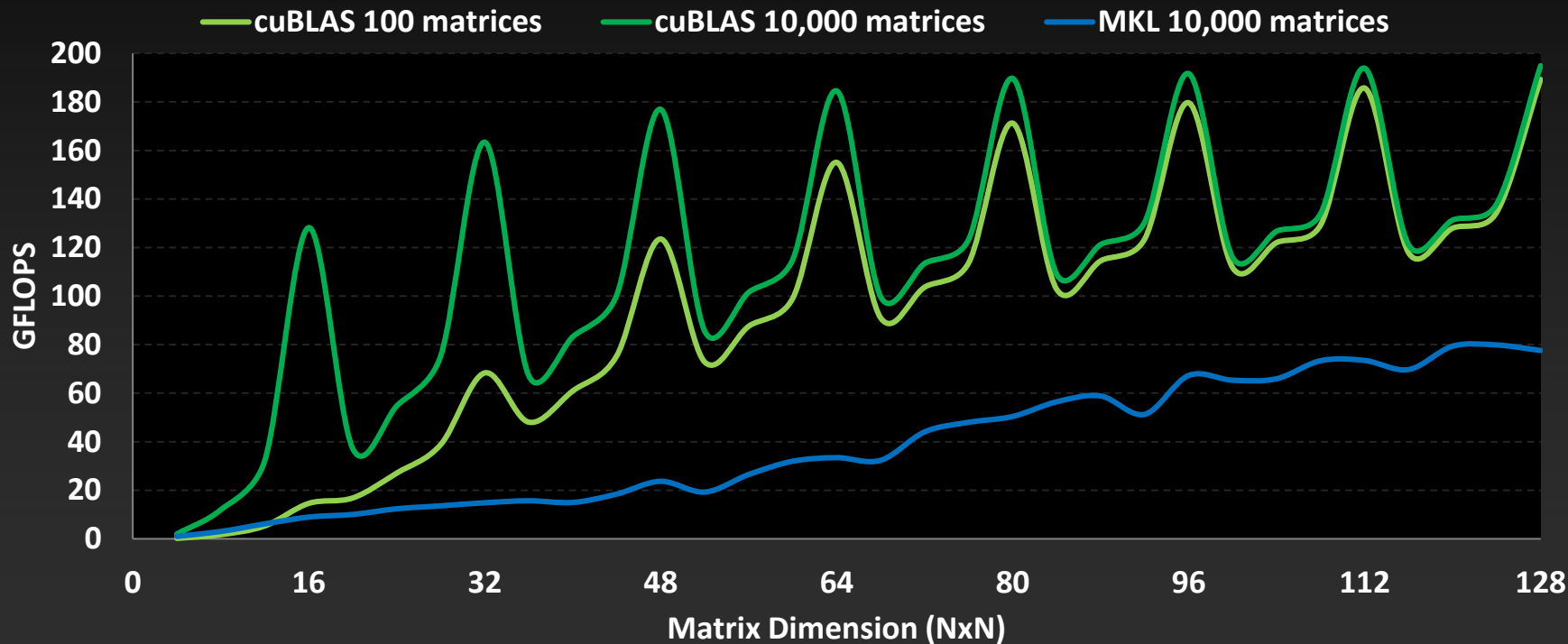
ZGEMM Performance vs Intel MKL



Performance may vary based on OS version and motherboard configuration

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

cuBLAS Batched GEMM API improves performance on batches of small matrices



Performance may vary based on OS version and motherboard configuration

• cuBLAS 4.1 on Tesla M2090, ECC on

• MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

cuSPARSE: Sparse linear algebra routines

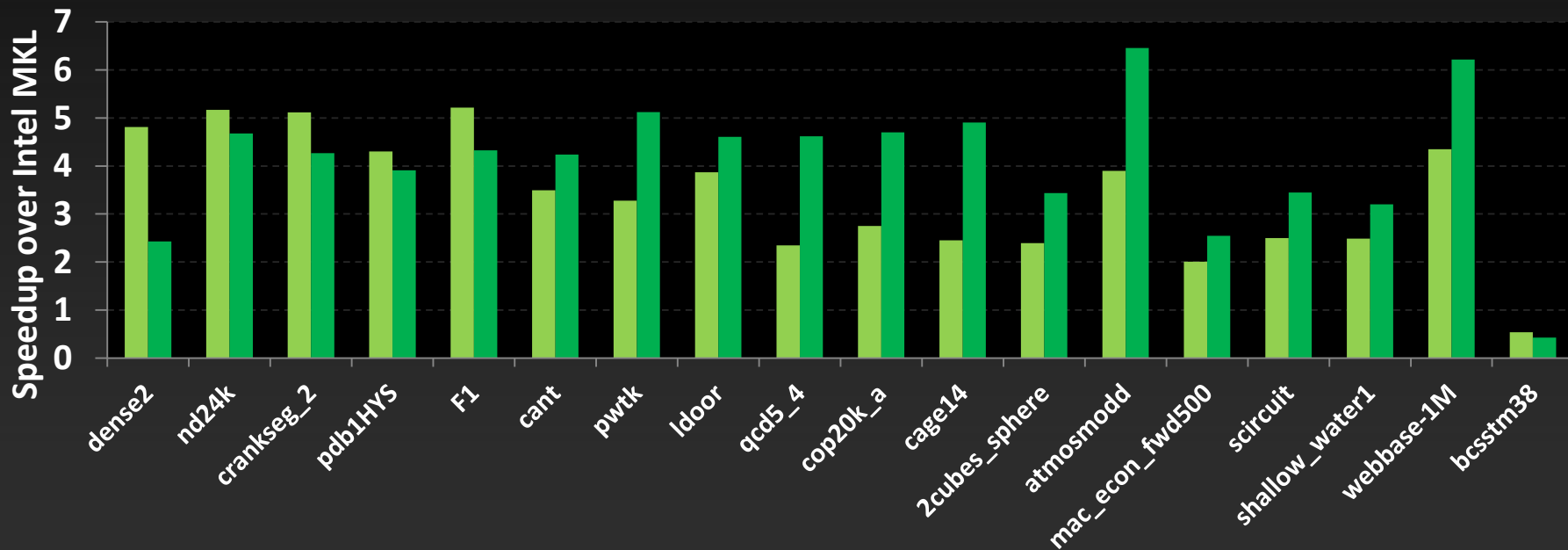
- Sparse matrix-vector multiplication & triangular solve
 - APIs optimized for iterative methods
- New in 4.1
 - Tri-diagonal solver with speedups up to 10x over Intel MKL
 - ELL-HYB format offers 2x faster matrix-vector multiplication

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

cuSPARSE is >6x Faster than Intel MKL

Sparse Matrix x Dense Vector Performance

■ csrmv* ■ hybmv*



*Average speedup over single, double, single complex & double-complex

Performance may vary based on OS version and motherboard configuration

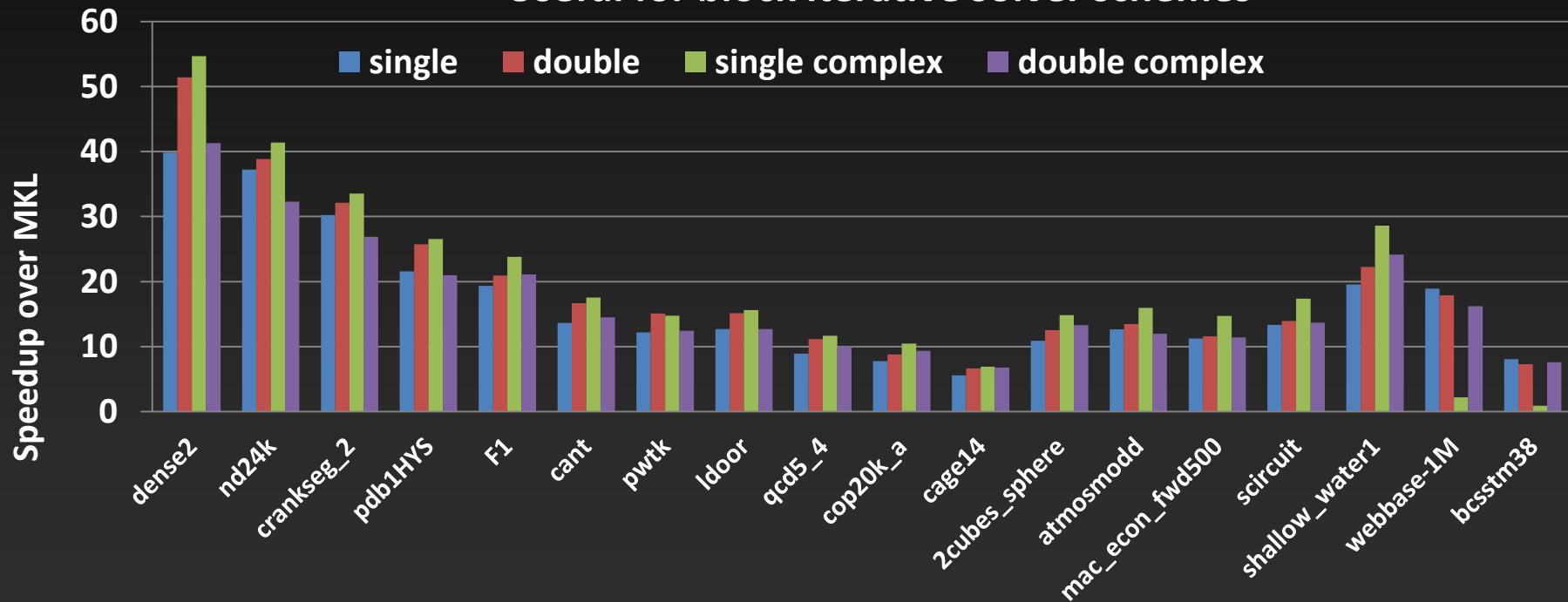
• cuSPARSE 4.1, Tesla M2090 (Fermi), ECC on

• MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core

Up to 40x faster with 6 CSR Vectors

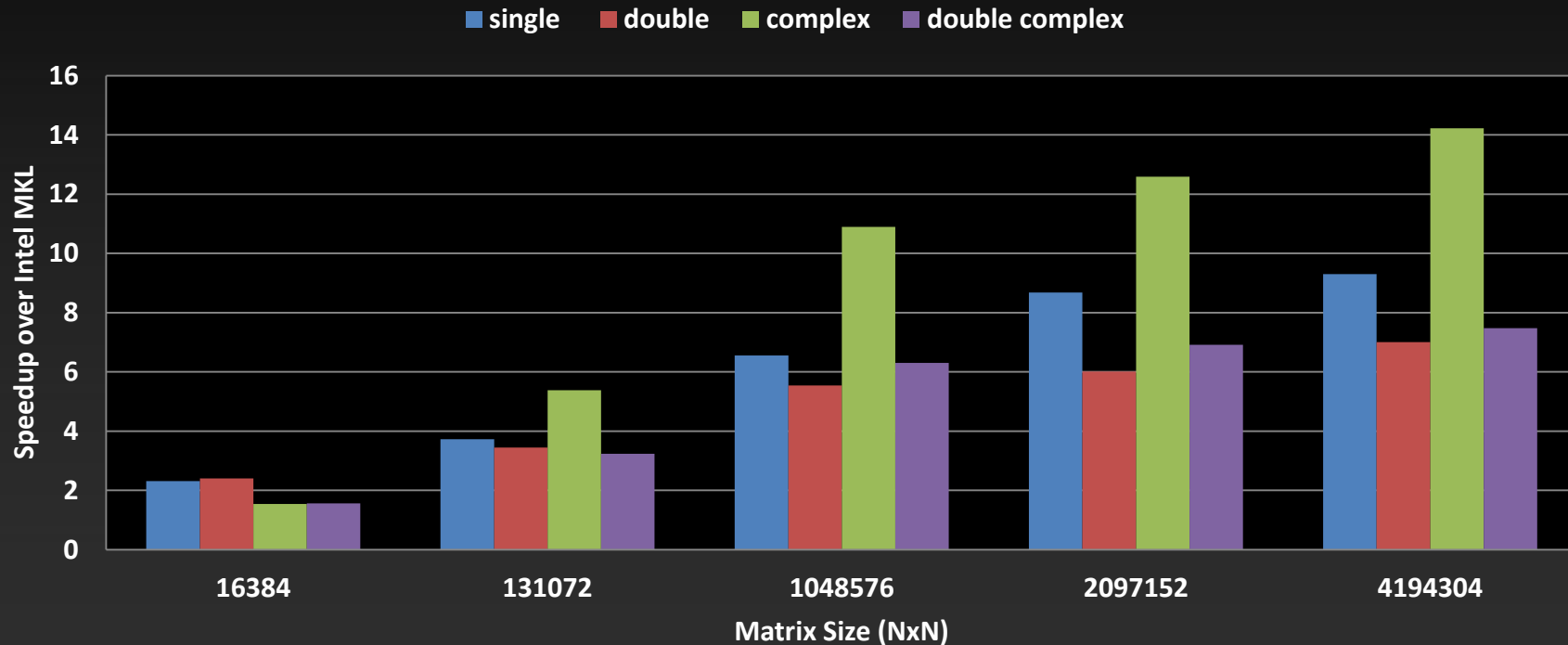
cuSPARSE Sparse Matrix x 6 Dense Vectors (csrmm)

Useful for block iterative solver schemes



Tri-diagonal solver performance vs. MKL

Speedup for Tri-Diagonal solver (gtsv)*



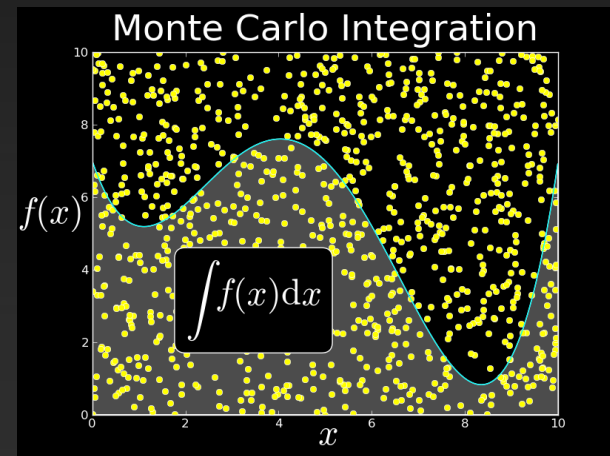
*Parallel GPU implementation does not include pivoting

Performance may vary based on OS version and motherboard configuration

• cuSPARSE 4.1, Tesla M2090 (Fermi), ECC on
• MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @

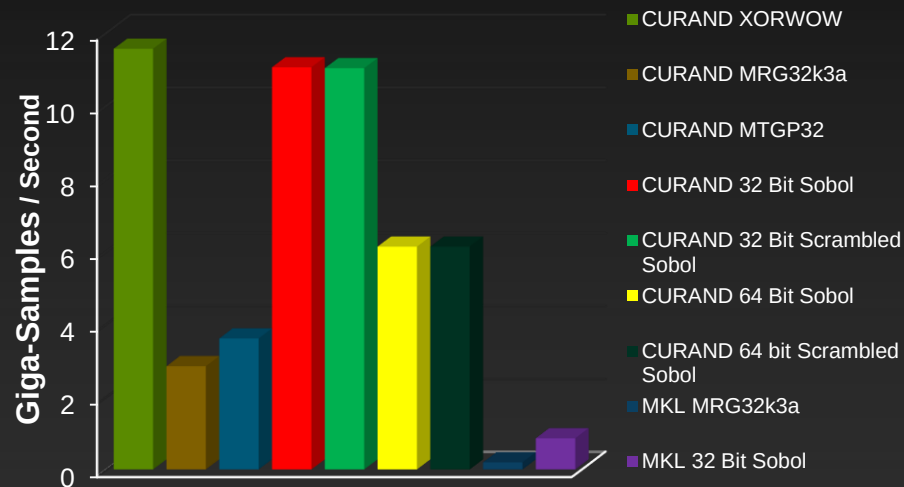
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
- Supports several output distributions
- Statistical test results reported in documentation
- New commonly used RNGs in CUDA 4.1
 - MRG32k3a RNG
 - MTGP11213 Mersenne Twister RNG



cuRAND Performance compared to Intel MKL

Double Precision Uniform Distribution



Double Precision Normal Distribution

