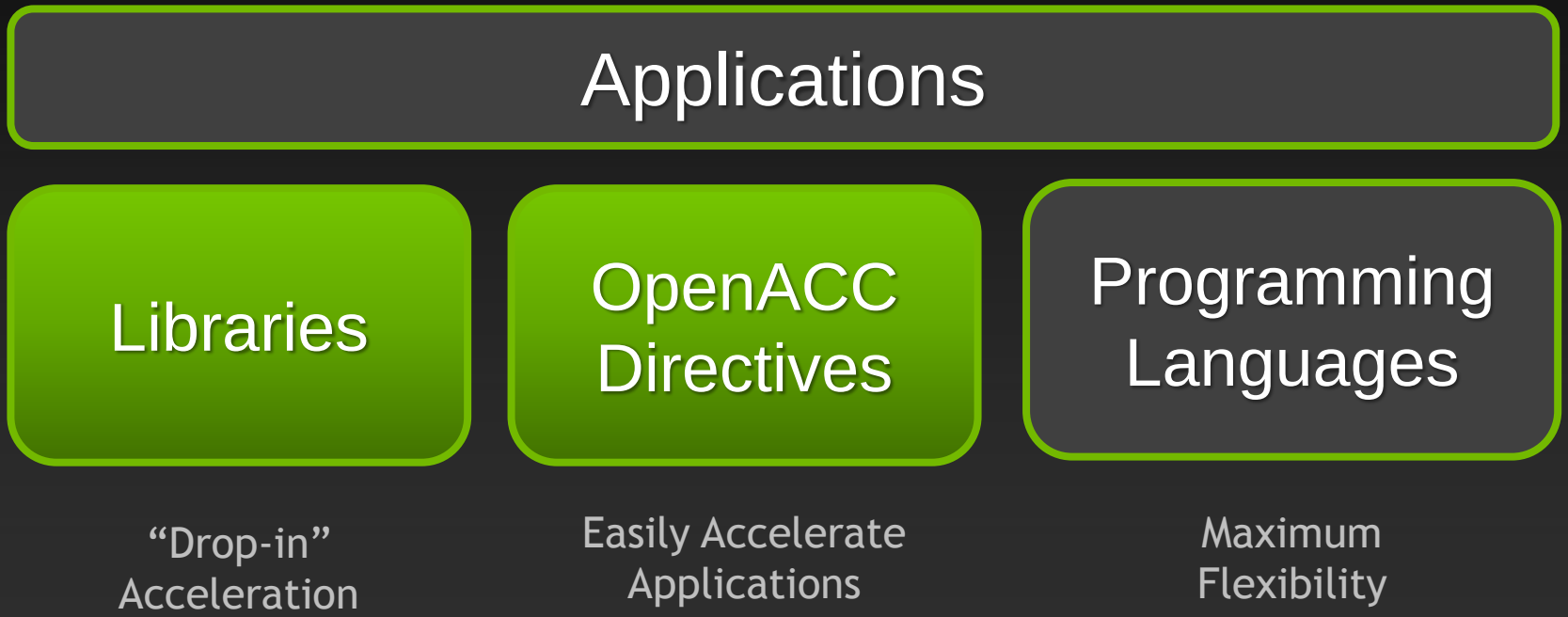
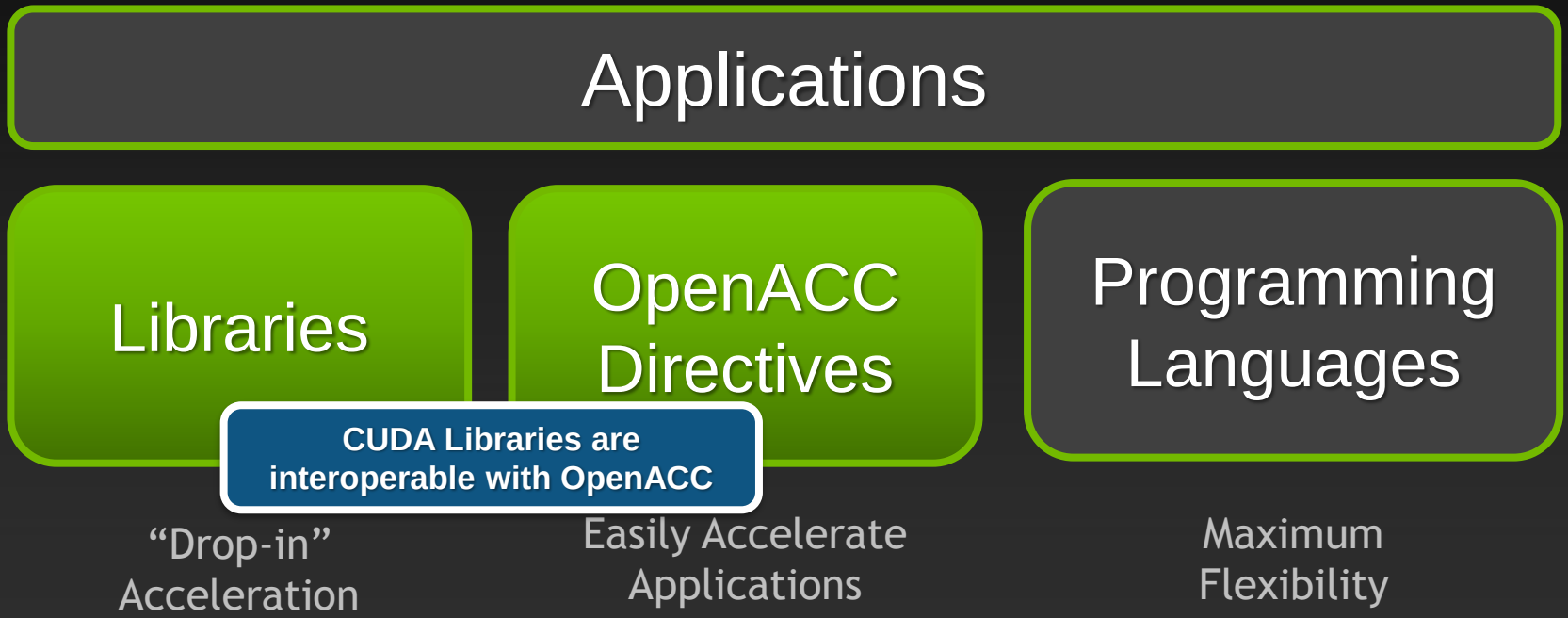
Using OpenACC With CUDA Libraries

John Urbanic
with NVIDIA
Pittsburgh Supercomputing Center

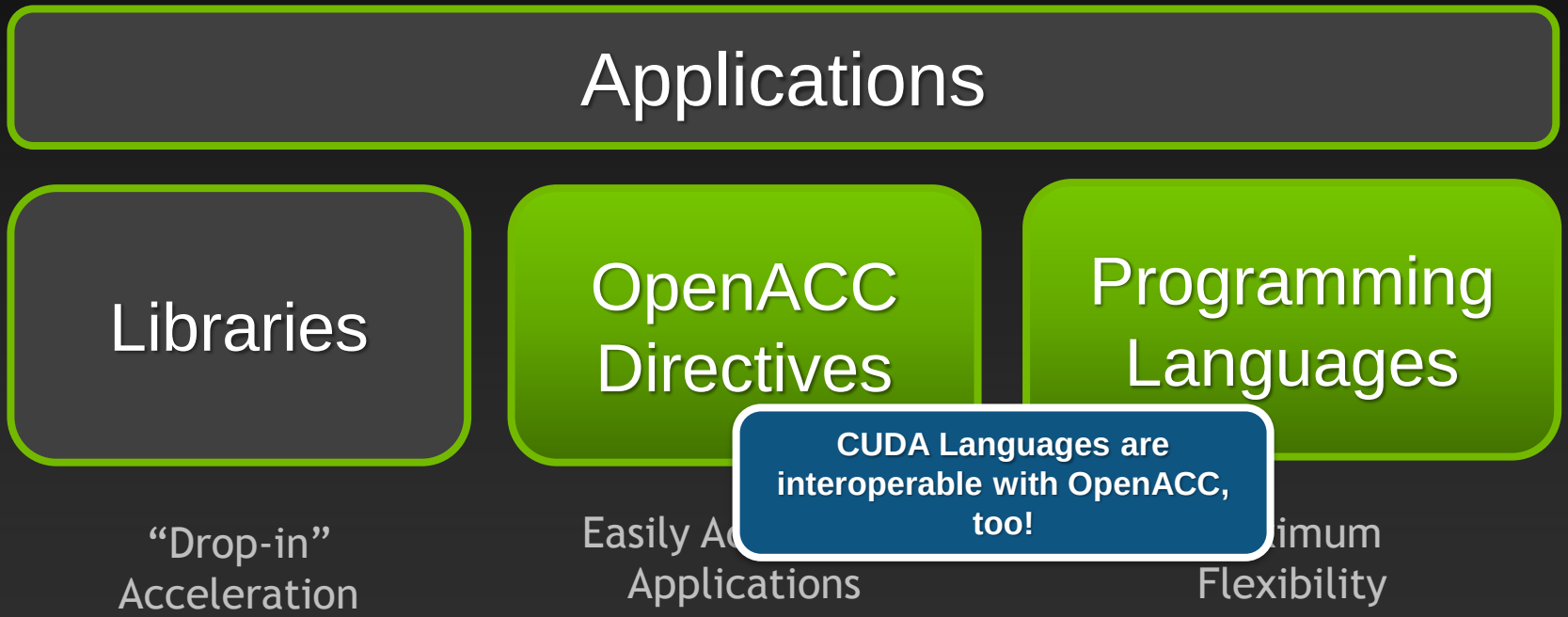
3 Ways to Accelerate Applications



3 Ways to Accelerate Applications

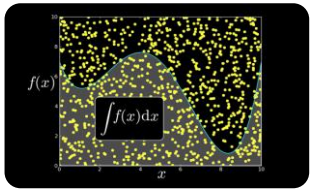


3 Ways to Accelerate Applications

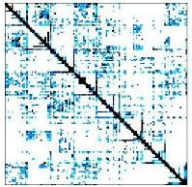


CUDA Libraries Overview

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

open source initiative

NVIDIA cuFFT

ROGUE WAVE SOFTWARE

IMSL Library

libjacket

Building-block Algorithms for CUDA

CUSP

Sparse Linear Algebra

open source initiative

Thrust

C++ STL Features for CUDA

open source initiative

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

Sharing data with libraries

- CUDA libraries and OpenACC both operate on device arrays
- OpenACC provides mechanisms for interop with library calls
  - deviceptr data clause
  - host_data construct
- These same mechanisms are useful for interoperating with custom CUDA C, C++ and Fortran code.

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

Makes the address of device data available on the host.

`use_device( list )` Tells the compiler to use the device address for any variable in *list*. Variables in the list must be present in device memory due to data regions that contain this construct

Example

C

```
#pragma acc host_data use_device(d_input)
```

Fortran

```
$!acc host_data use_device(d_input)
```

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/Cufft-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;
        }
    }
}
```

Note: The PGI C compiler does not currently support structs in OpenACC loops, so 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
```

Summary

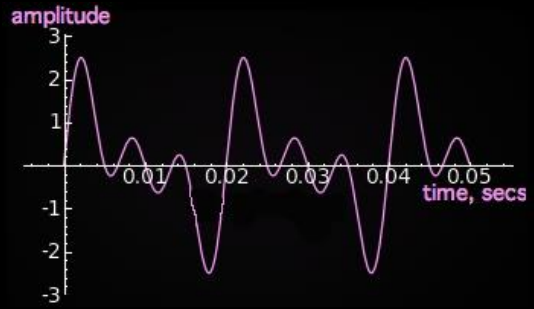
- Use deviceptr data clause to pass pre-allocated device data to OpenACC regions and loops
- Use host_data to get device address for pointers inside acc data regions
- The same techniques shown here can be used to share device data between OpenACC loops and
  - Your custom CUDA C/C++/Fortran/etc. device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

Appendix

Compelling Cases For Various Libraries Of Possible Interest To You

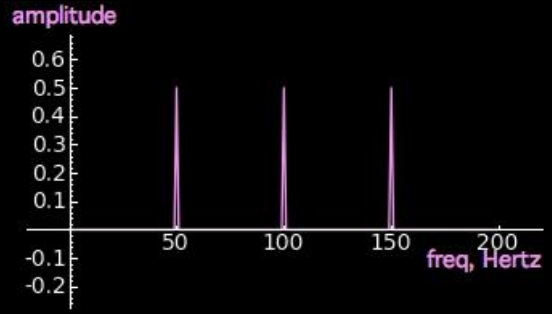
cuFFT: Multi-dimensional FFTs

- 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



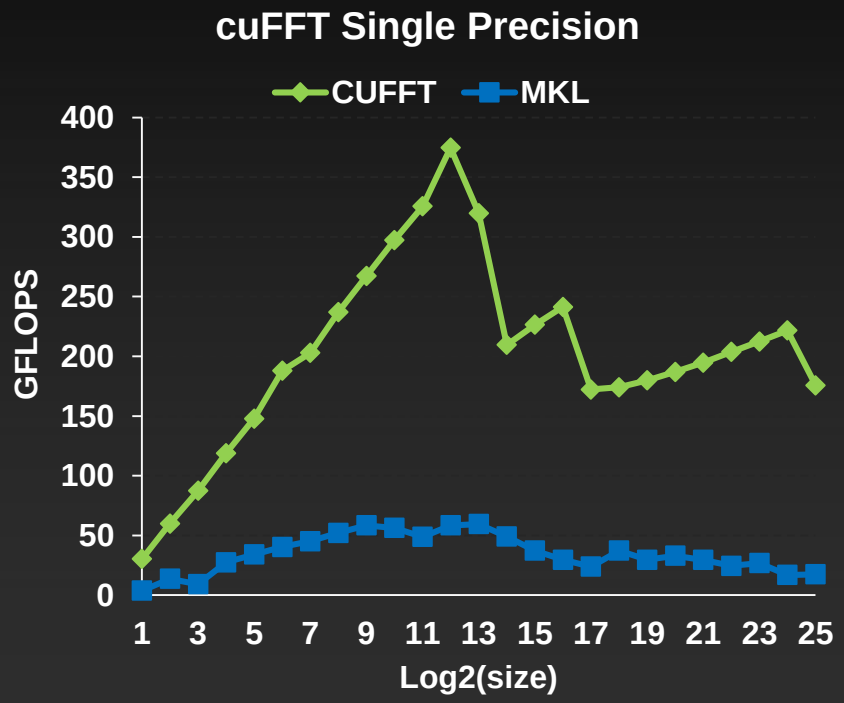
$$F(x) = \sum_{n=0}^{N-1} f(n) e^{-j2\pi(x\frac{n}{N})}$$

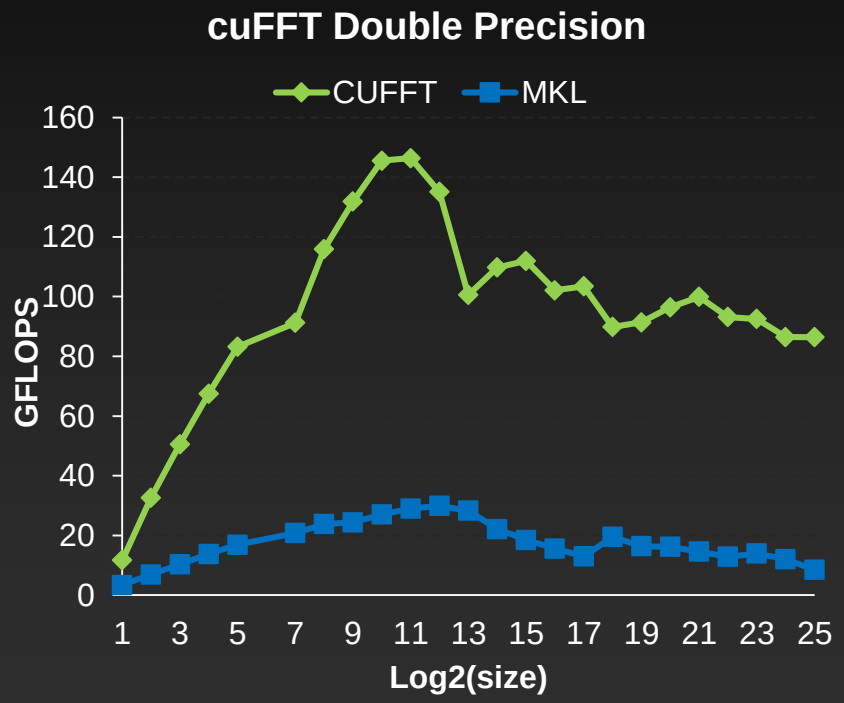
$$f(n) = \frac{1}{N} \sum_{n=0}^{N-1} F(x) e^{j2\pi(x\frac{n}{N})}$$



FFTs up to 10x Faster than MKL

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



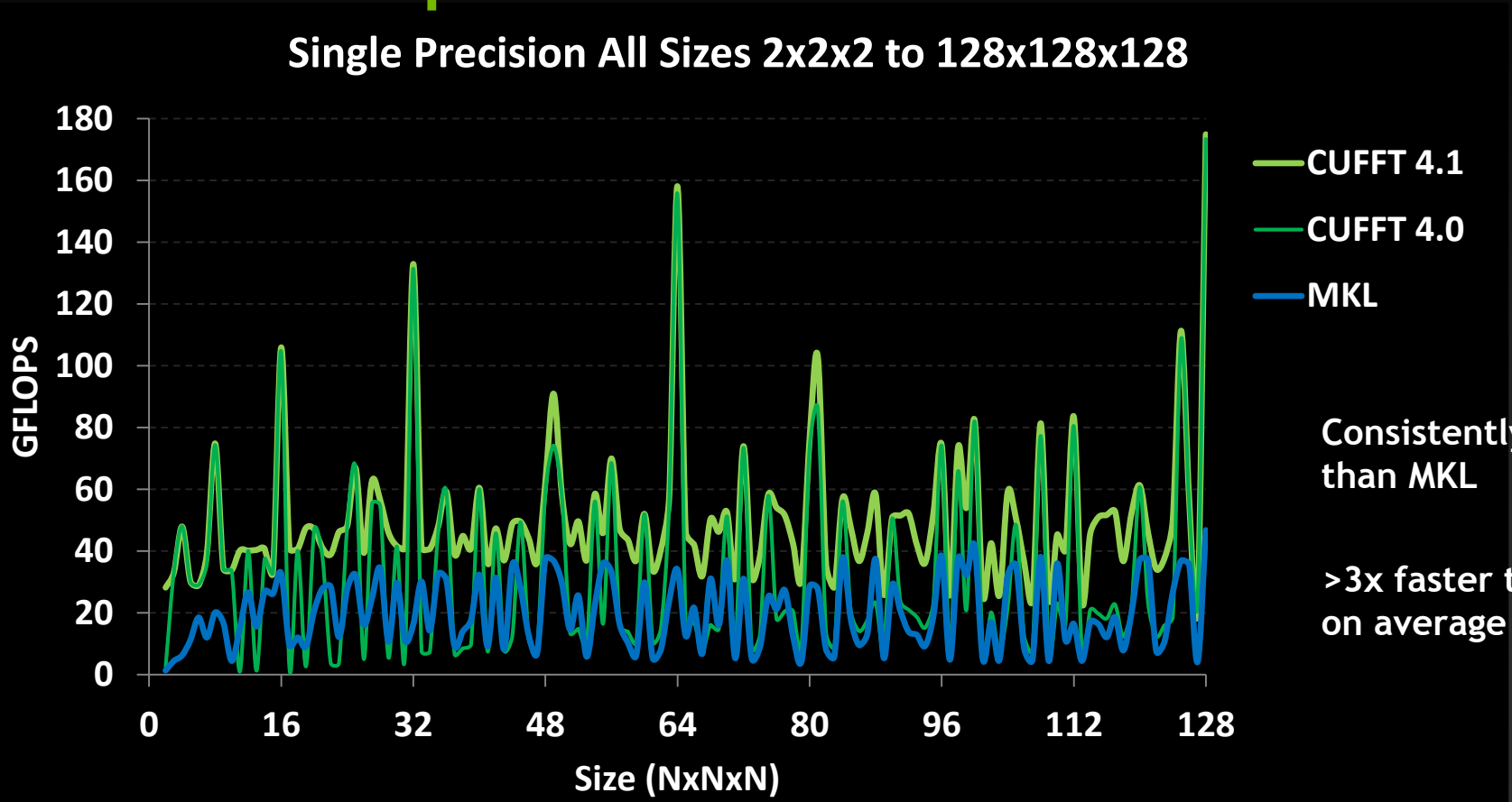


- Measured on sizes that are exactly powers-of-2
- cuFFT 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz

Performance may vary based on OS version and motherboard configuration

PITTSBURGH SUPERCOMPUTING CENTER

CUDA 4.1 optimizes 3D transforms



Consistently faster than MKL

>3x faster than 4.0 on average

- 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

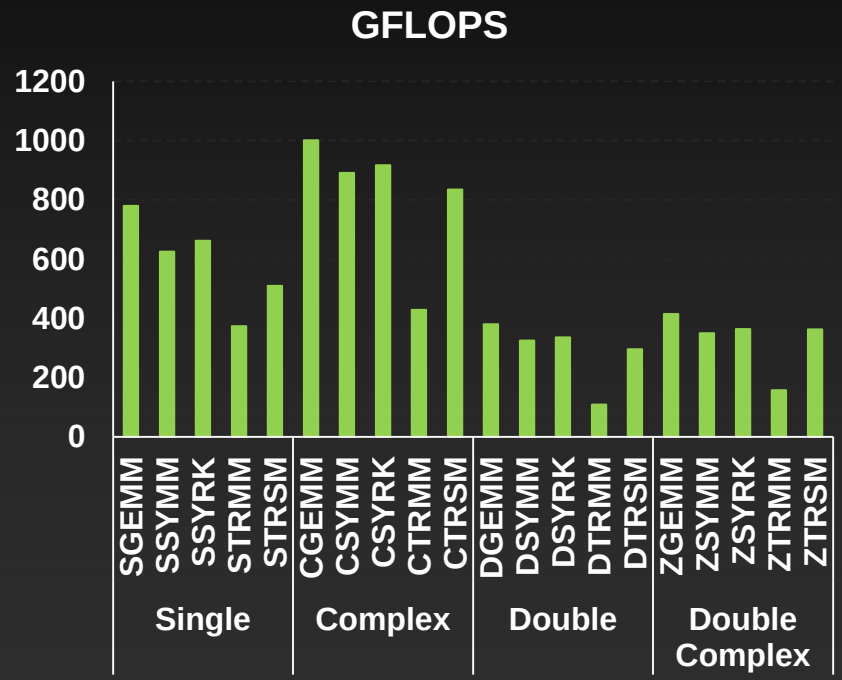
cuBLAS: Dense Linear Algebra on GPUs

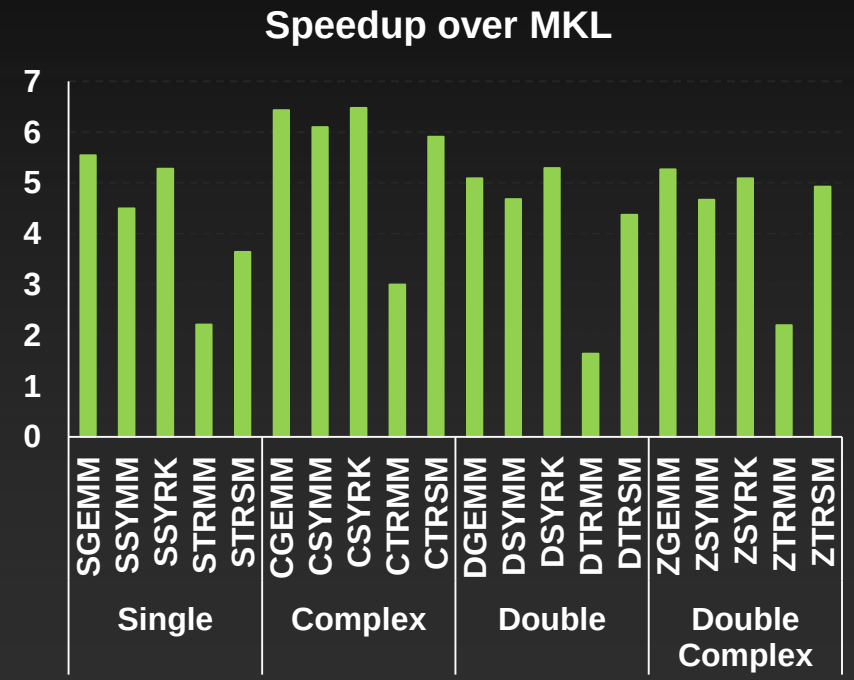
- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex

- 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



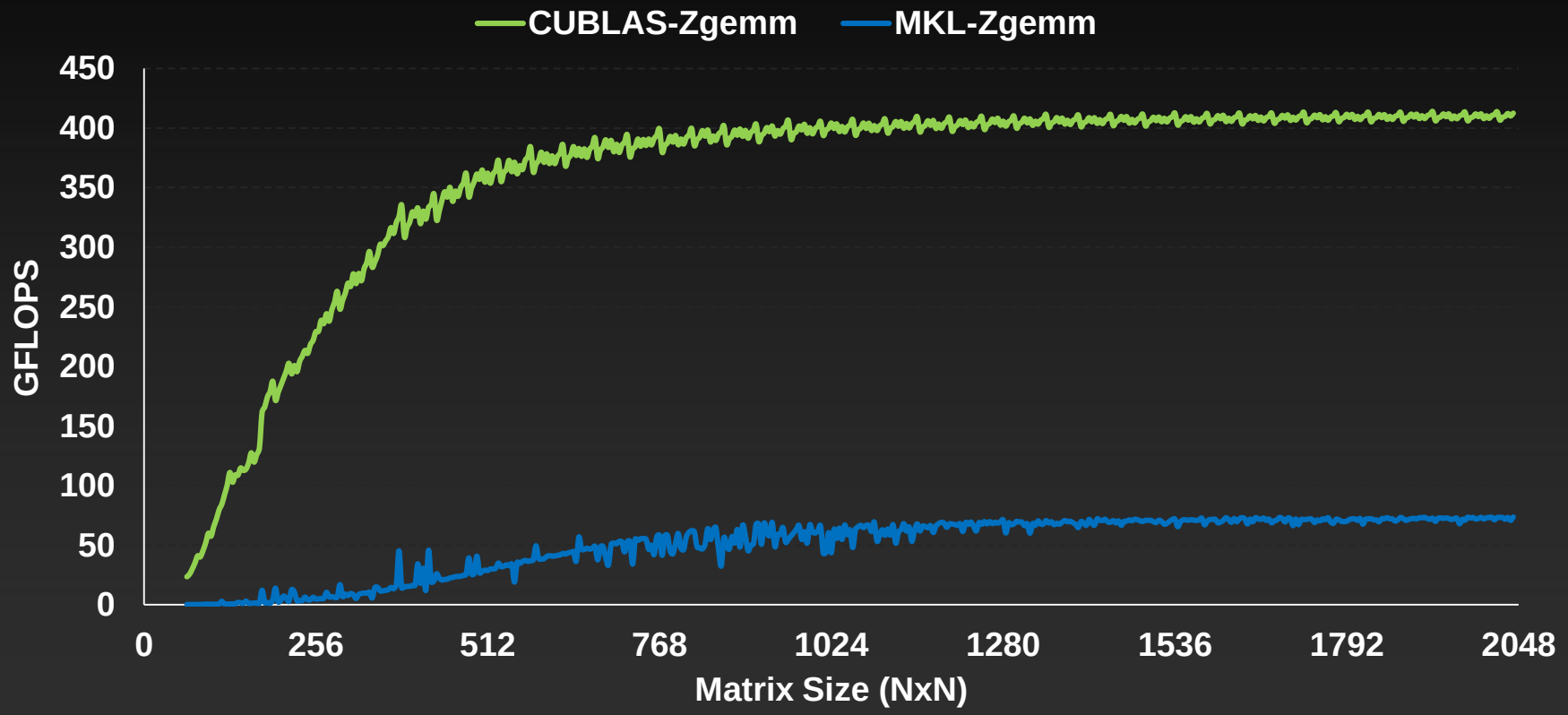


- 4Kx4K matrix size
- cuBLAS 4.1, Tesla M2090 (Fermi), ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @

Performance may vary based on OS version and motherboard configuration

PITTSBURGH SUPERCOMPUTING CENTER

ZGEMM Performance vs Intel MKL

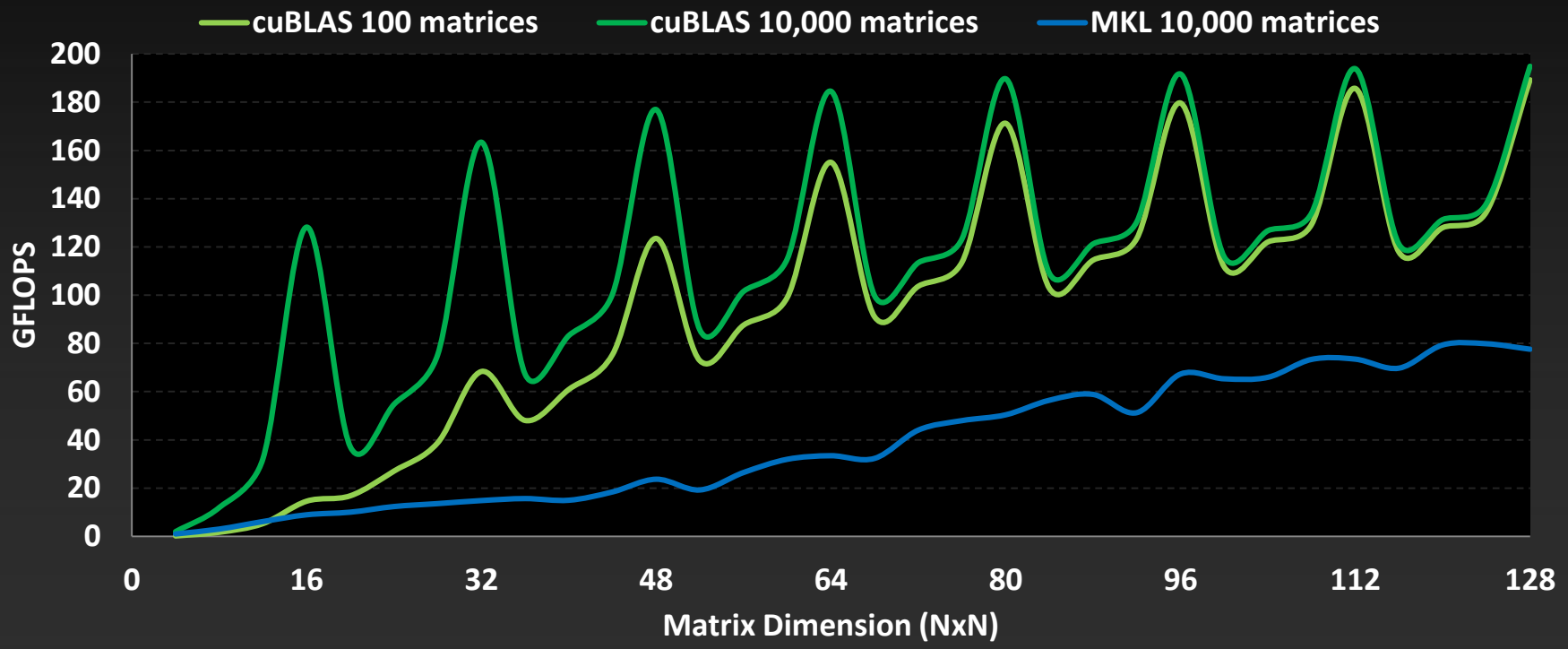


Performance may vary based on OS version and motherboard configuration

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

PITTSBURGH SUPERCOMPUTING CENTER

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

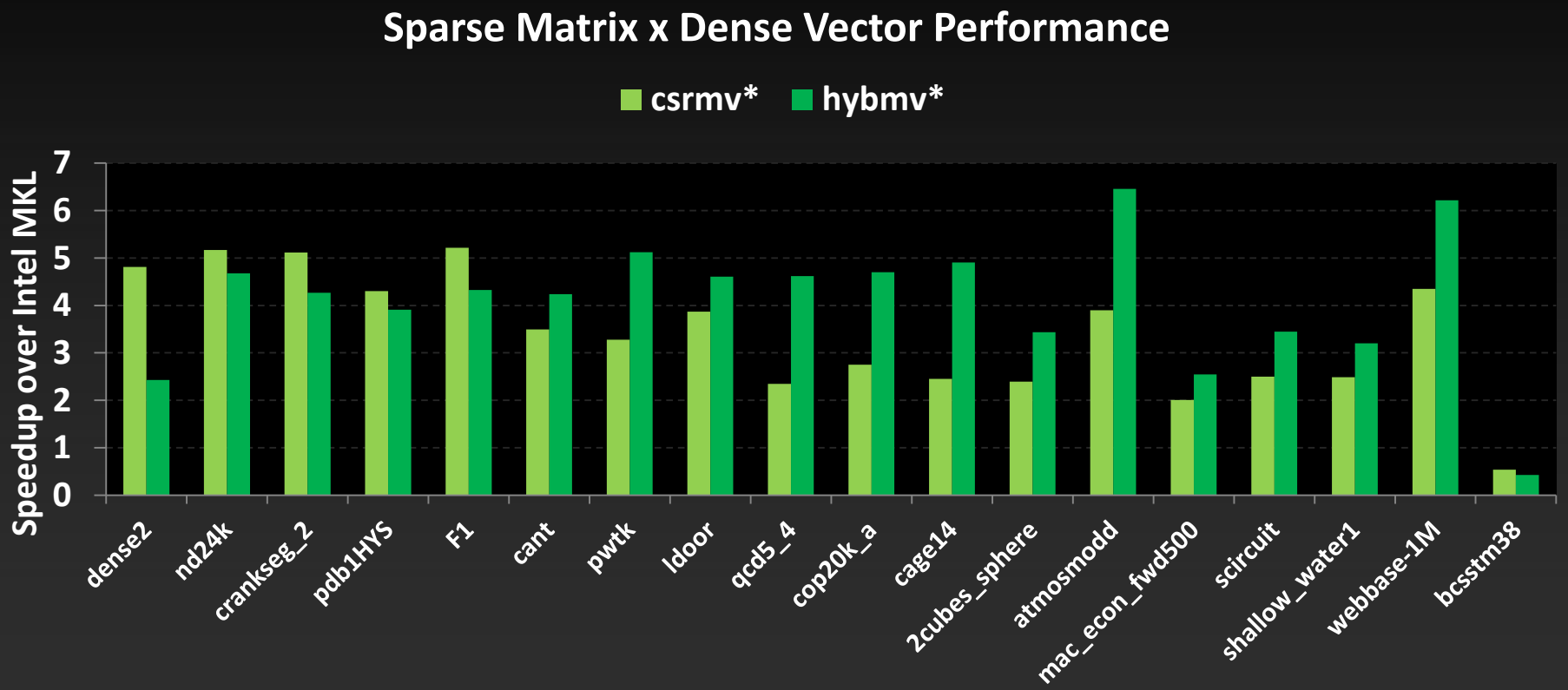
PITTSBURGH SUPERCOMPUTING CENTER

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



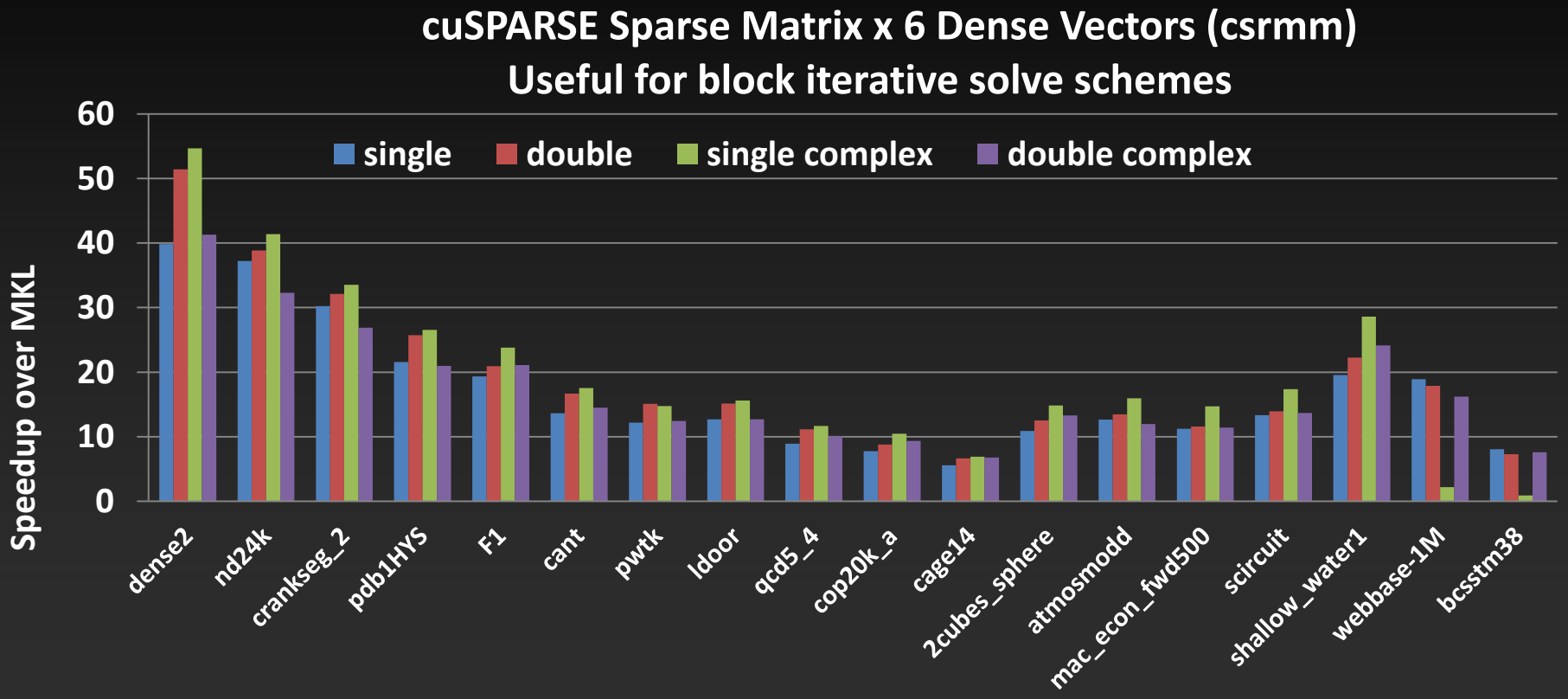
**Average speedup over single, double, single complex & double-complex*

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

PITTSBURGH SUPERCOMPUTING CENTER

Up to 40x faster with 6 CSR Vectors

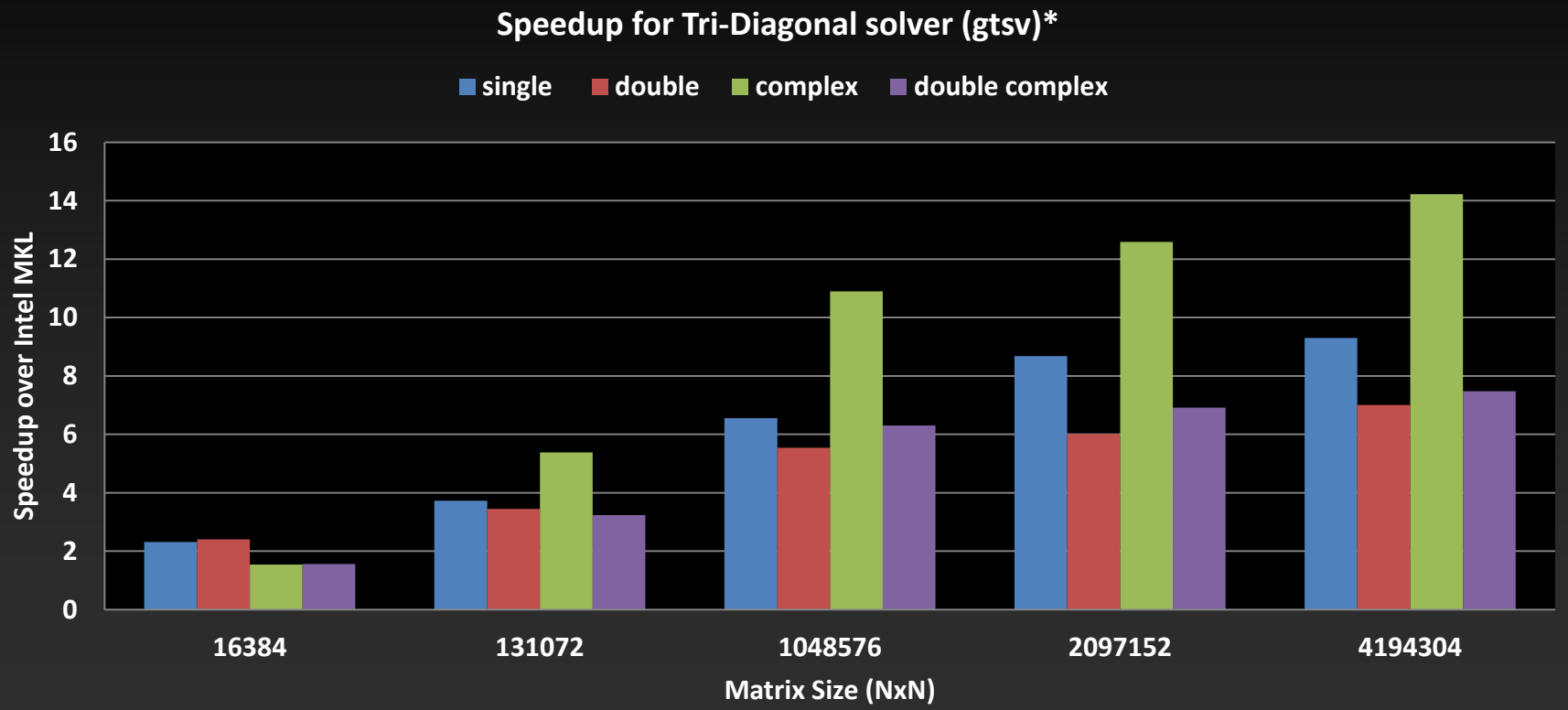


Performance may vary based on OS version and motherboard configuration

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

PITTSBURGH SUPERCOMPUTING CENTER

Tri-diagonal solver performance vs. MKL



Parallel GPU implementation does not include pivoting

Performance may vary based on OS version and motherboard configuration

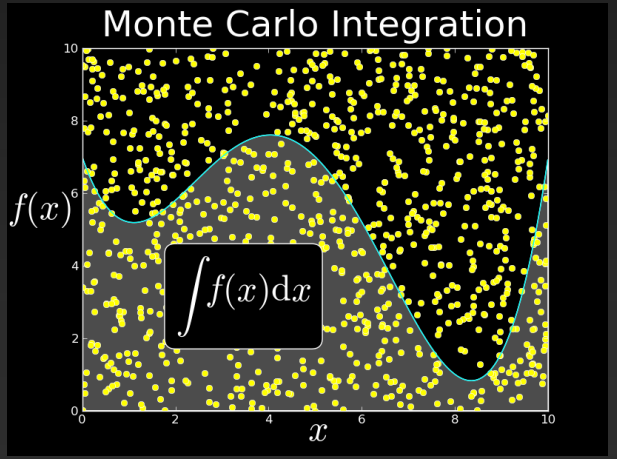
- cuSPARSE 4.1, Tesla M2090 (Fermi), ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @

PITTSBURGH SUPERCOMPUTING CENTER

cuRAND: Random Number Generation

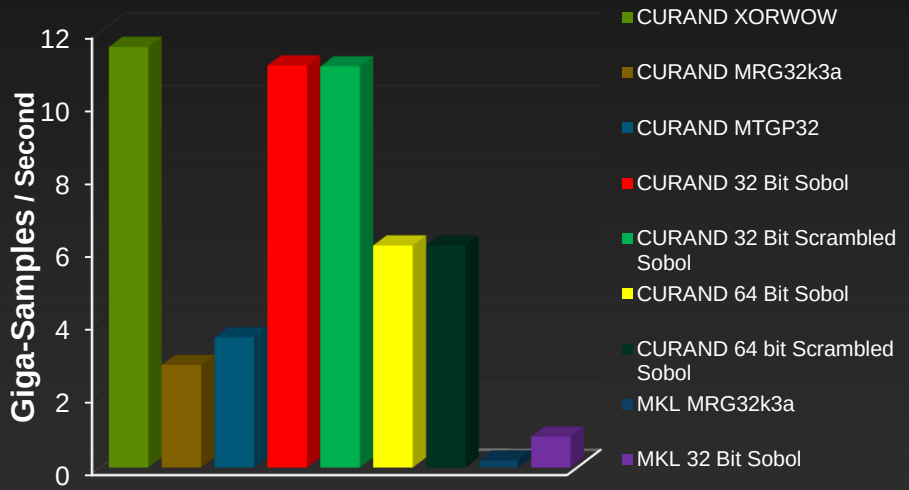
- Pseudo- and Quasi-RNGs
- Supports several output distributions
- Statistical test results reported in documentation

- New commonly used RNGs in CUDA 4.1
  - MRG32k3a RNG
  - MTGP11213 Mersenne Twister RNG

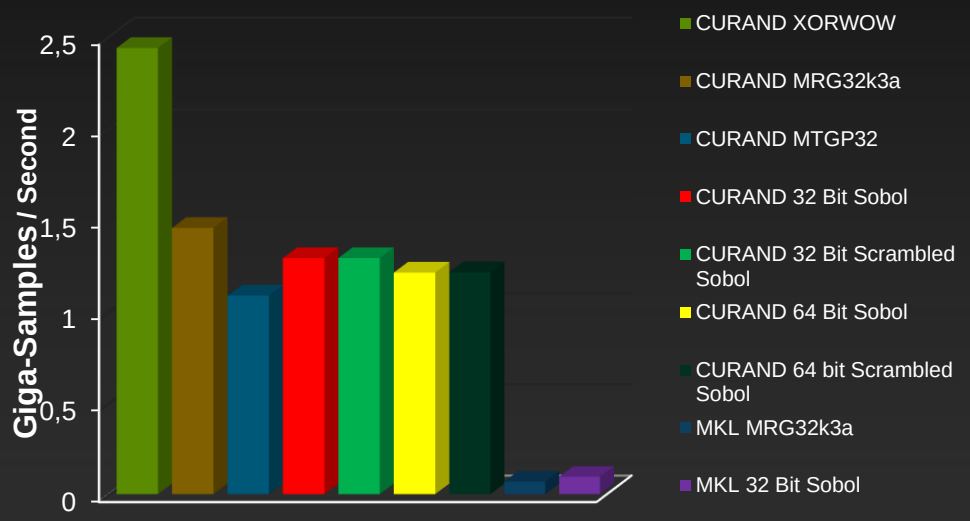


cuRAND Performance compared to Intel MKL

Double Precision Uniform Distribution



Double Precision Normal Distribution



Performance may vary based on OS version and motherboard configuration

•cuRAND 4.1, Tesla M2090 (Fermi), ECC
• MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 @

PITTSBURGH SUPERCOMPUTING CENTER

OpenMP 4.0 (now 4.5) for Accelerators

John Urbanic
Parallel Computing Scientist
Pittsburgh Supercomputing Center

OpenACC vs. OpenMP

- OpenMP has a very similar directive philosophy. This is no surprise as OpenACC was started by OpenMP members as an “accelerator development branch” with the idea of merging it back in.
- OpenMP assume(d) that memory movement isn’t an issue, but that thread startup overhead is. The available directives reflect that.
- OpenACC assumes threads are very lightweight, but that data movement onto and off of the accelerator are significant. The directives reflect that.
- But, they are both similar in approach and assume that you, the programmer, are responsible for designating parallelizable loops.
- They are also complementary and can be used together very well.

OpenMP Thread Control Philosophy

OpenMP was traditionally oriented towards controlling fully independent processors. In return for the flexibility to use those processors to their fullest extent, OpenMP assumes that you know what you are doing and does not recognize data dependencies in the same way as OpenACC.

While you override detected data dependencies in OpenACC (with the `independent` clause), there is no such thing in OpenMP. Everything is ***assumed*** to be independent. You must be the paranoid one, not the compiler.

OpenMP assumes that every thread has it's own synchronization control (barriers, locks). GPUs do not have that at all levels. For example, NVIDIA GPUs have synchronization at the Warp level, but not the Thread Block level. There are implications regarding this difference.

In general, you might observe that OpenMP was built when threads were limited and start up overhead was considerable (as it still is on CPUs). The design reflects the need to control for this. OpenACC starts with devices built around very, very lightweight threads.

Intel's MIC Approach

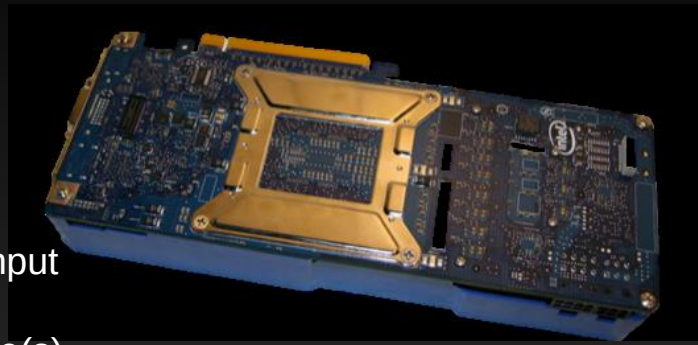
Since the days of RISC vs. CISC, Intel has mastered the art of figuring out what is important about a new processing technology, and saying "why can't we do this in x86?"

The Intel Many Integrated Core (MIC) architecture is about large die, simpler circuit, much more parallelism, in the x86 line.

Courtesy Dan Stanzione, TACC

What is MIC?

Basic Design Ideas:

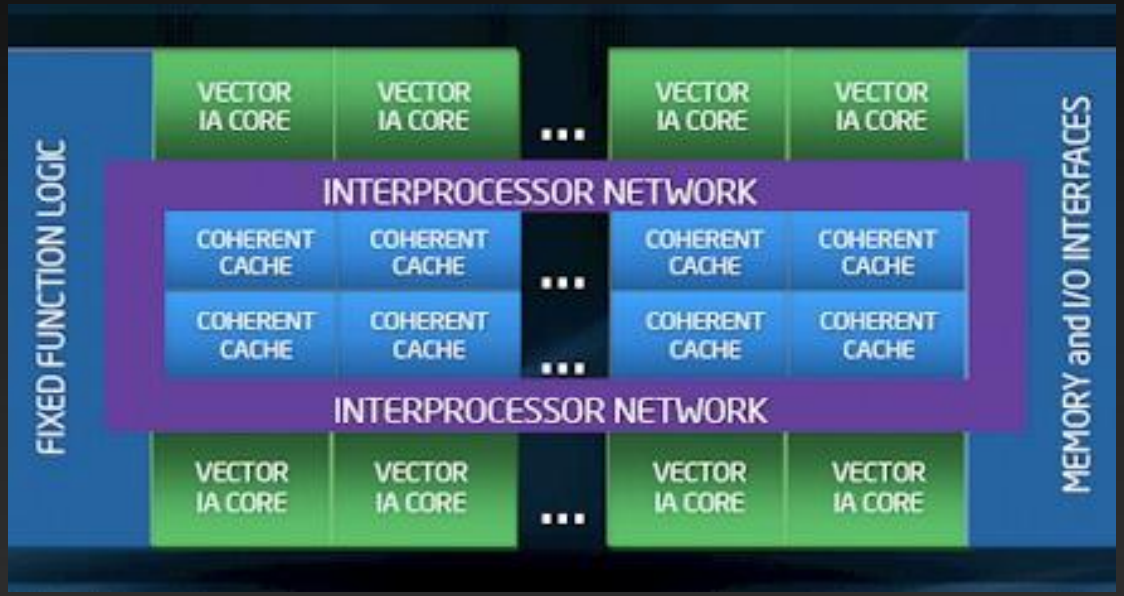


- Leverage x86 architecture (a CPU with many cores)
- Use x86 cores that are simpler, but allow for more compute throughput
- Leverage existing x86 programming models
- Dedicate much of the silicon to floating point ops., keep some cache(s)
- Keep cache-coherency protocol
- Increase floating-point throughput per core
- Implement as a separate device
- Strip expensive features (out-of-order execution, branch prediction, etc.)
- Widened SIMD registers for more throughput (512 bit)
- Fast (GDDR5) memory on card

Courtesy Dan Stanzione, TACC

MIC Architecture

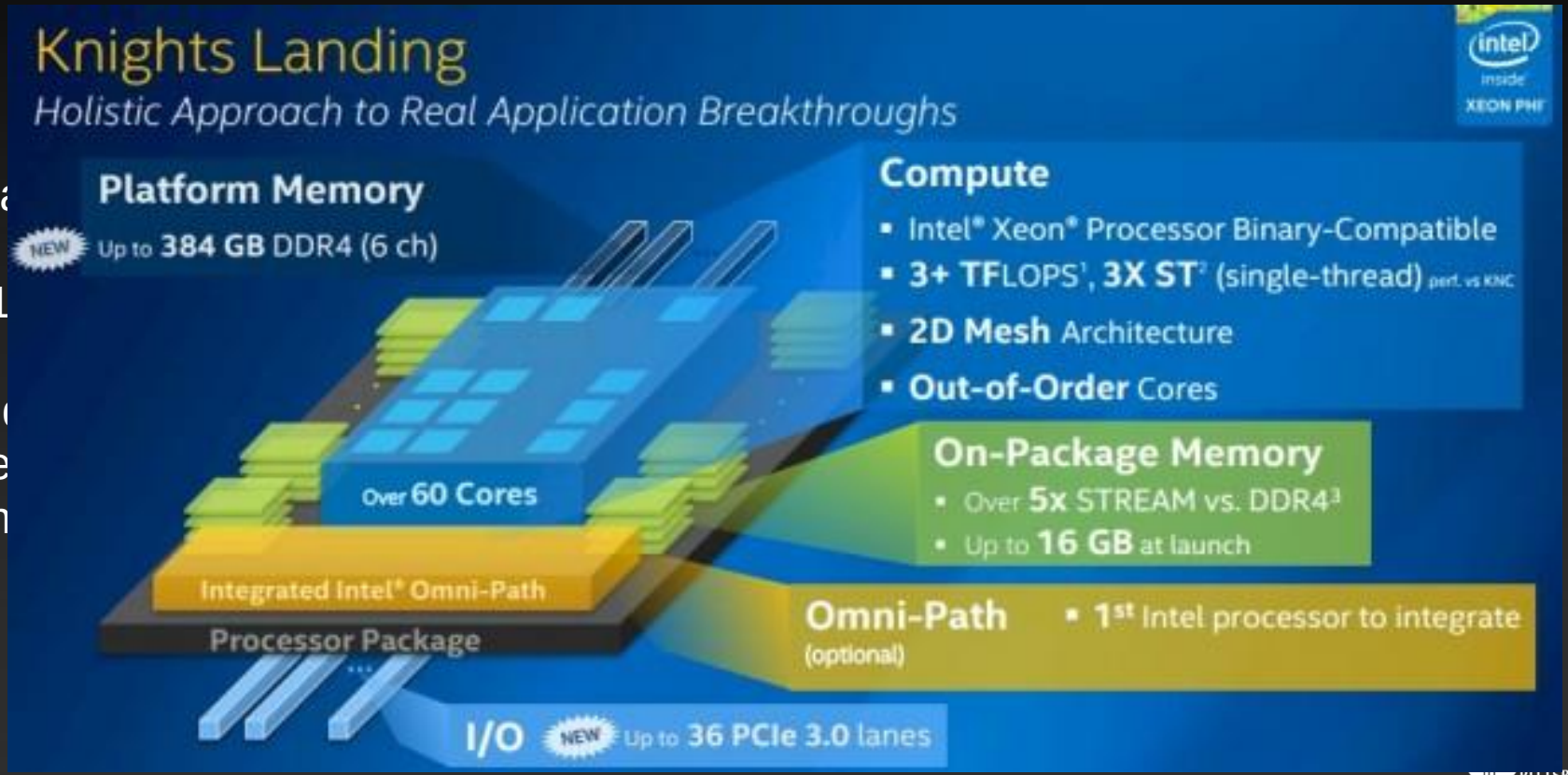
- Many cores on the die
- L1 and L2 cache
- Bidirectional ring network for **L2** Memory and PCIe connection



Courtesy Dan Stanzione, TACC

MIC Architecture

- Ma
- L1
- Bi
  ne
  an



Courtesy Dan Stanzione, TACC

OpenMP 4.0 Data Migration

OpenMP comes from an SMP multi-core background. The original idea was to avoid the pain of using Unix/Posix pthreads directly. As SMPs have no concept of different memory spaces, OpenMP has not been concerned with that until now. With OpenMP 4.0, that changes. We now have data migration control and related capability like data shaping.

```
#pragma omp target device(0) map(tofrom:B)
```

SAXPY in OpenMP 4.0 on NVIDIA

```
int main(int argc, const char* argv[]) {
    int n = 10240; floata = 2.0f; floatb = 3.0f;
    float*x = (float*) malloc(n * sizeof(float));
    float*y = (float*) malloc(n * sizeof(float));

// Run SAXPY TWICE inside data region
#pragma omp target data map(to:x)
{
#pragma omp target map(tofrom:y)
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
    for(inti = 0; i < n; ++i){
          y[i] = a*x[i] + y[i];
     }
#pragma omp target map(tofrom:y)
#pragma omp teams
#pragma omp distribute
#pragma omp parallel for
     for(inti = 0; i < n; ++i){
          y[i] = b*x[i] + y[i];
     }
}
}
```

Courtesy Christian Terboven

Comparing OpenACC with OpenMP 4.0 on NVIDIA & Phi

OpenMP 4.0 for Intel Xeon Phi

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp parallel for
for (i=0; i<N; i++)
    B[i] += sin(B[i]);
```

OpenMP 4.0 for NVIDIA GPU

```
#pragma omp target device(0) map(tofrom:B)
#pragma omp teams num_teams(num_blocks) num_threads(bsize)
#pragma omp distribute
for (i=0; i<N; i += num_blocks)
      #pragma omp parallel for
      for (b = i; b < i+num_blocks; b++)
            B[b] += sin(B[b]);
```

OpenACC for NVIDIA GPU

```
#pragma acc kernels
for (i=0; i<N; ++i)
     B[i] += sin(B[i]);
```

First two examples
Courtesy Christian Terboven

OpenMP 4.0 Across Architectures

```
#if defined FORCPU
#pragma omp parallel for simd
#elif defined FORKNC
#pragma omp target teams distribute parallel for simd
#elif defined FORGPU
#pragma omp target teams distribute parallel for \
   schedule(static,1)
#elif defined FORKNL
#pragma omp parallel for simd schedule(dynamic)
#endif
for( int j = 0; j < n; ++j )
   x[j] += a*y[j];
```

Courtesy Michael Wolfe

Which way to go?

While this might be an interesting discussion of the finer distinctions between these two standards and the future merging thereof, it is not. At the moment, there is a simpler reality:

- OpenMP 4.0 was ratified in July 2013, and it will be a while before it has the widespread support of OpenMP 3. It is currently fully implemented only on Intel compilers for Xeon Phi and partially now in GCC 5.x and better in GCC 6.1. LLVM Clang seems to be on way.

- OpenACC supports Phi with the CAPS compiler, but via an OpenCL back end. PGI has had something "coming" for a while. You would really have to have a good reason to not use the native Intel compiler OpenMP 4.0 at this time.

So, at this time…

- If you are using Phi, you are probably going to be using the Intel OpenMP release.
- If you are using NVIDIA GPU's, you are going to be using OpenACC.

Of course, there are other ways of programming both of these devices. You might treat Phi as MPI cores and use CUDA on NVIDIA , for example. But if the directive based approach is for you, then your path is clear. I don't attempt to discuss the many other types of accelerators here (AMD, DSPs, FPGAs, ARM), but these techniques apply there as well.

And as you should now suspect, even if it takes a while for these to merge as a standard, it is not a big jump for you to move between them.

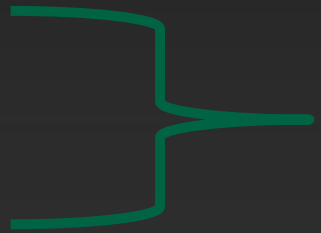
Going Hostless

Both Intel and NVIDIA are converging towards a hostless future.

- Intel
  - Plug a bunch of MICs (Knights Landing) into backplanes
  - Programming model doesn't really change
- NVIDIA
  - Expanding MIMD capability of hardware with each generation
  - CUDA evolving towards remote data access with each version
  - Adding CPU on board

Some things we did not mention

- OpenCL (Khronos Group)
  - Everyone supports, but not as a primary focus
  - Intel – OpenMP
  - NVIDIA – CUDA, OpenACC
  - AMD – now HSA (hUMA/APU oriented)
- DirectCompute (Microsoft)
  - Not HPC oriented
- C++ AMP (MS/AMD)
- TBB (Intel C++ template library)
- Cilk (Intel, now in a gcc branch)



Very C++ for threads