Introduction to CUDA and Libraries

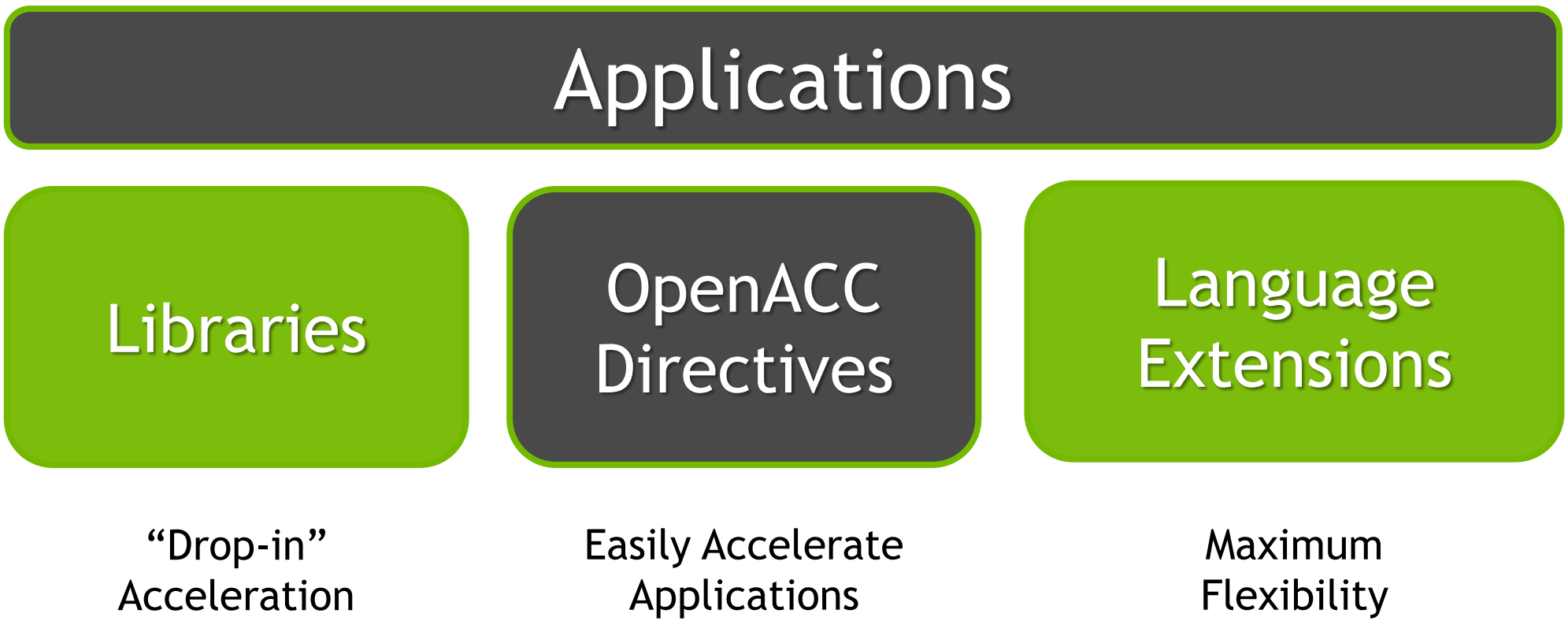
Jeremy Appleyard, September 2015

In this talk

- CUDA Programming
- GPU Libraries

Ask questions at any point!

Three Ways to Accelerate Applications

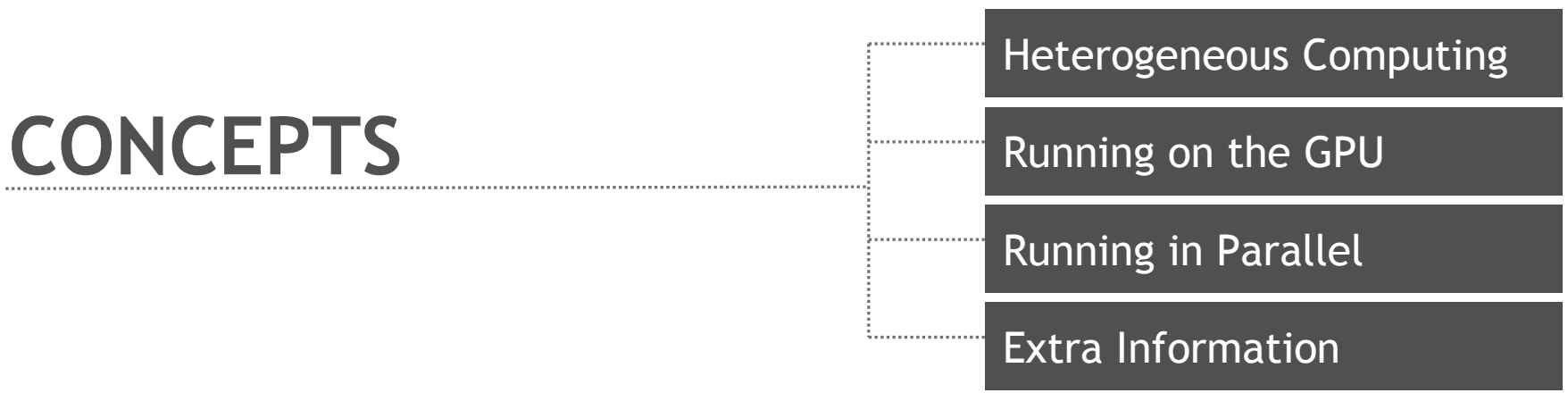


CUDA Programming

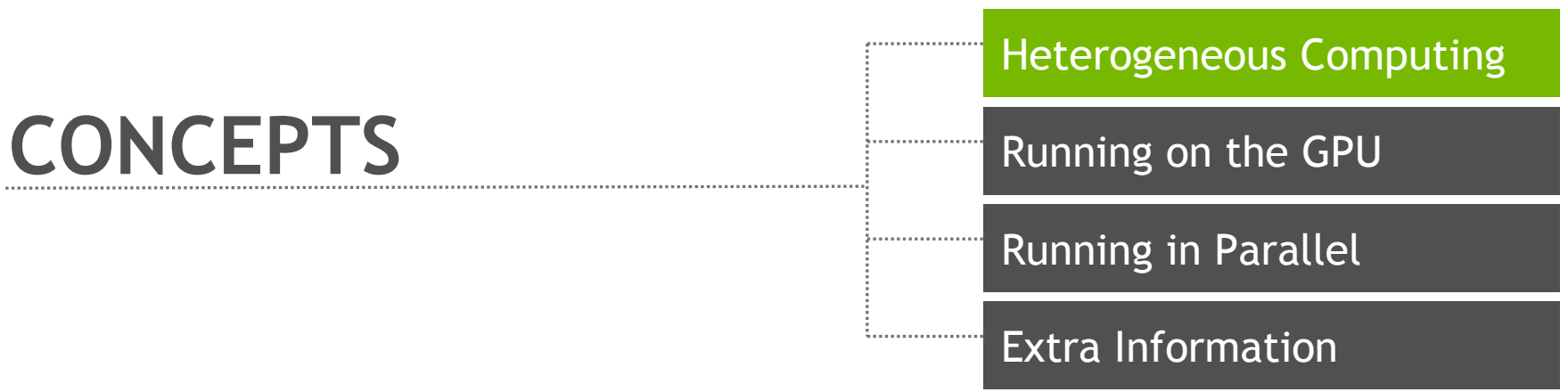
GPU Language Extensions
CUDA

- CUDA is available through C/C++, Fortran, Python, Matlab, …
- CUDA Fortran
  - Based on industry-standard Fortran
  - Small set of extensions to enable heterogeneous programming
  - Straightforward APIs to manage devices, memory etc.

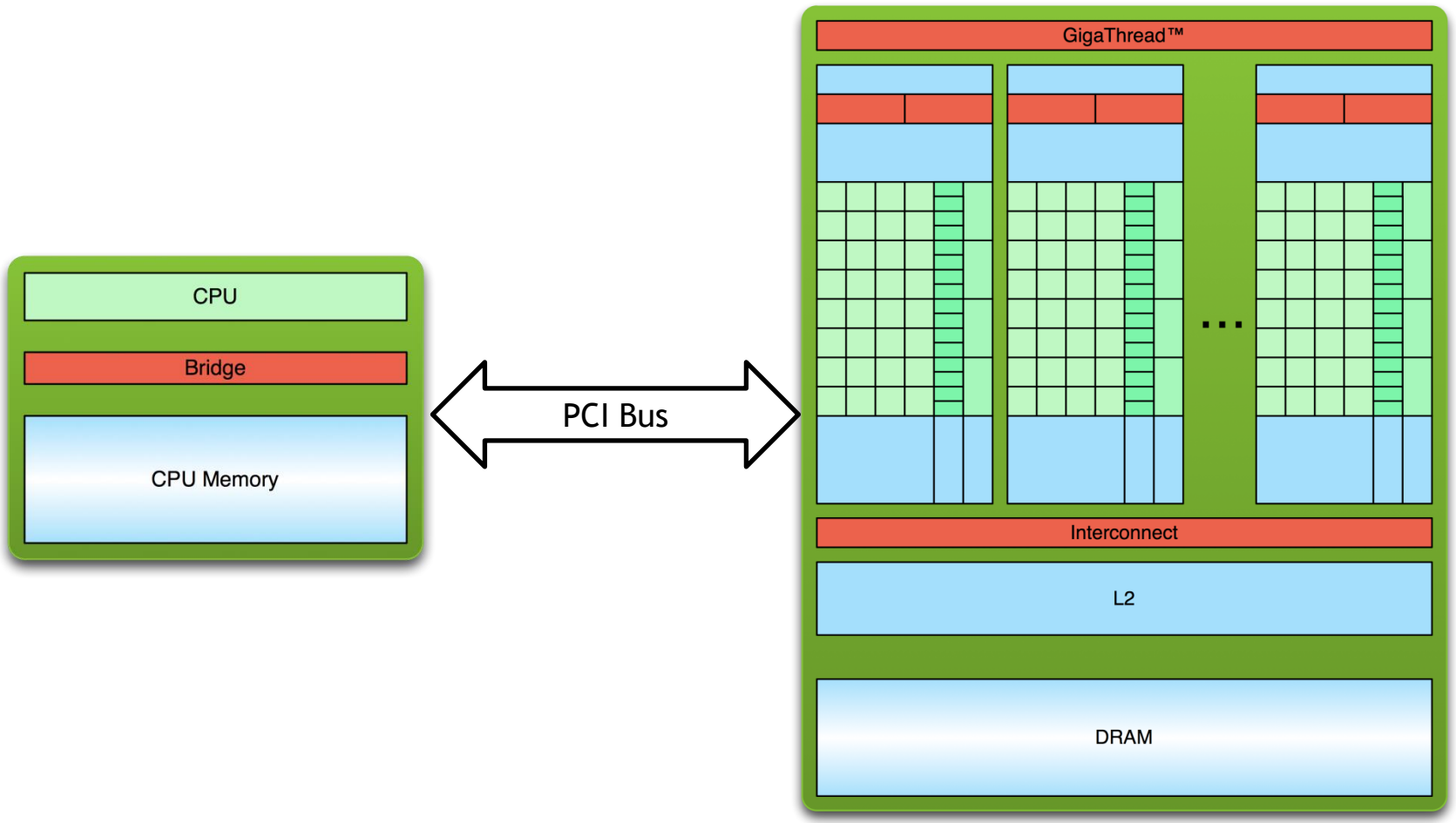
CONCEPTS


Heterogeneous Computing
Running on the GPU
Running in Parallel
Extra Information

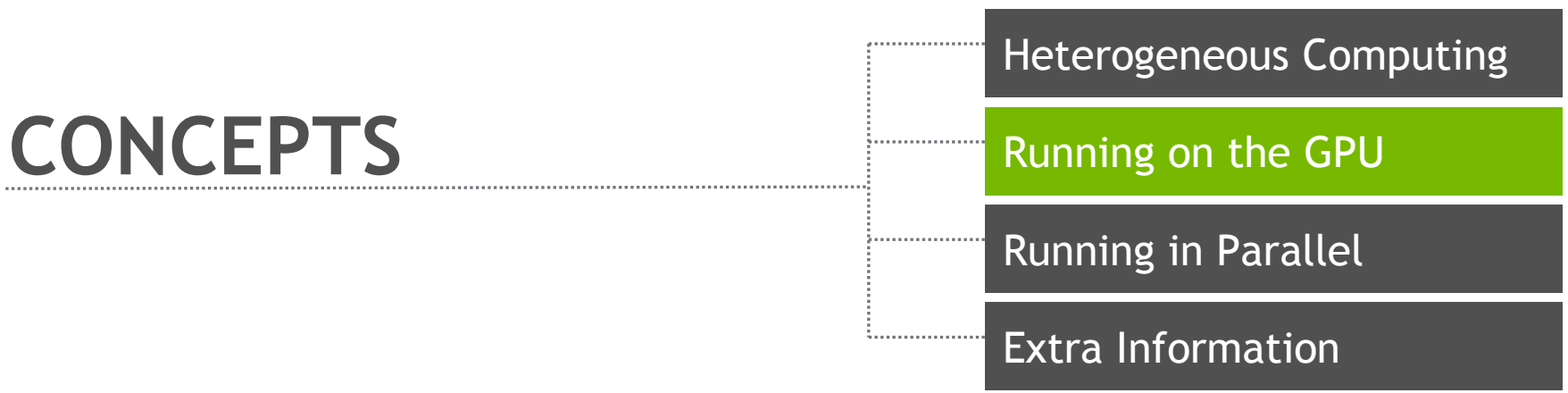
CONCEPTS


Heterogeneous Computing
Running on the GPU
Running in Parallel
Extra Information

Heterogeneous Computing



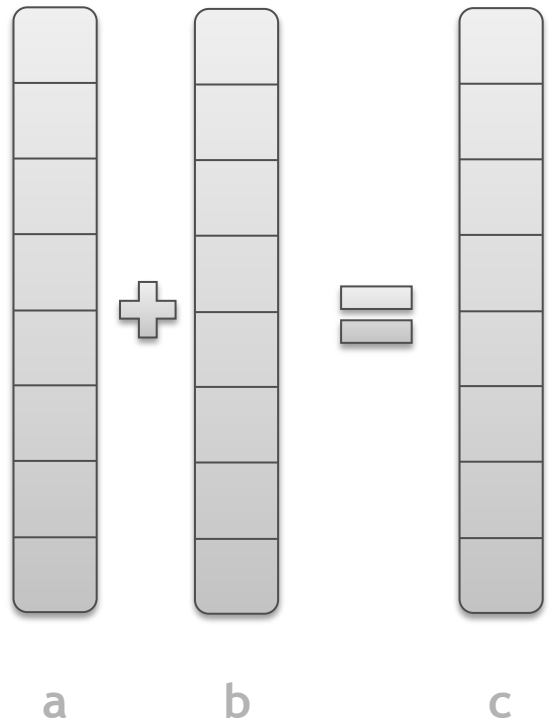
CONCEPTS


Heterogeneous Computing
Running on the GPU
Running in Parallel
Extra Information

Parallel Programming in CUDA

Example

- GPU computing is about massive parallelism
- Vector addition is a simple case where one can parallelise over the number of elements
- More generically this operation is called axpy



Standard Fortran

One loop

```
subroutine vecAdd_CPU(c, a, b, n)
   INTEGER, intent(in) :: n
   REAL, intent(in)  :: a(n), b(n)
   REAL, intent(out) :: c(n)

   INTEGER :: i

   do i=1,n
      c(i) = a(i) + b(i)
   end do
end subroutine vecAdd_CPU

...

call vecAdd_CPU(c, a, b, n)
```

CUDA Fortran

Executing on the GPU

- attributes(global)
  - Tells the compiler we want to compile for GPU
- value
  - Pass n by value
- <<< 1, 1 >>>
  - GPU launch command

```
attributes(global) subroutine vecAdd_GPU(c, a, b, n)
   INTEGER, value :: n
   REAL, device, intent(in)  :: a(n), b(n)
   REAL, device, intent(out) :: c(n)

   INTEGER :: i

   do i=1,n
      c(i) = a(i) + b(i)
   end do
end subroutine vecAdd_GPU

...

call vecAdd_GPU <<< 1, 1 >>> (c, a, b, n)
```

Memory Management

Host and device

- Host and device memory are physically separate entities
- Two ways of declaring GPU readable memory:
  - Managed attribute (GPU and CPU readable)
  - Device attribute (GPU only readable)

Memory Management

Managed

- Managed memory is very easy to use
- Requires synchronization after the kernel call before the CPU can access

```
program main
   use kernelMod
   use cudafor
   integer, parameter :: n = 1000000
   real, managed :: a(n), b(n), c(n)
   integer :: istat

   ! Initialise a, b on the CPU

   ...

   call vecAdd_GPU <<< 1, 1 >>> (c, a, b, n)

   istat = cudaDeviceSynchronize()

   ...
```

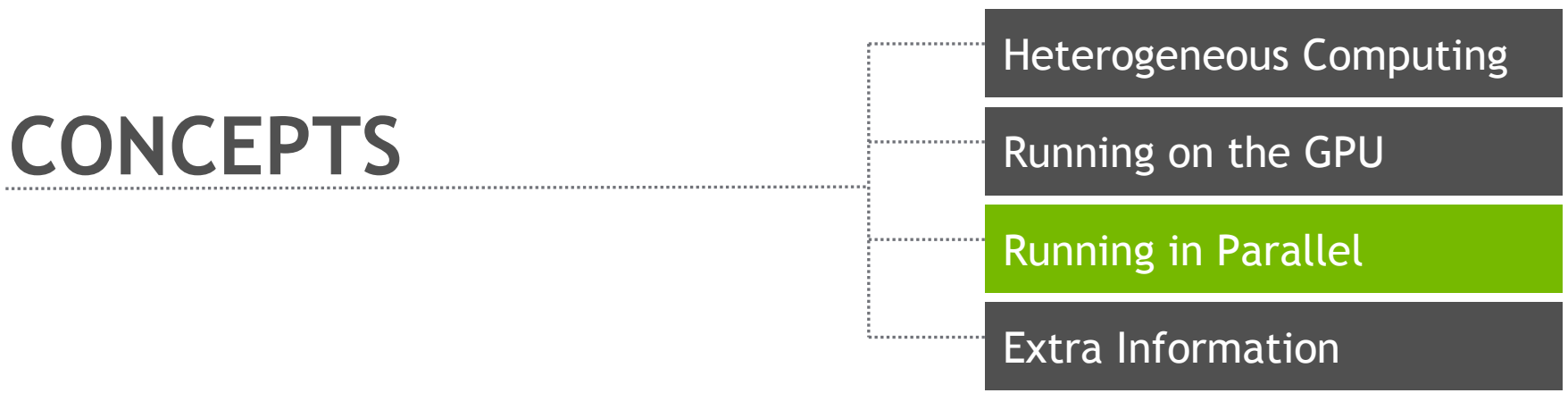
Memory Management

Device

- Device memory gives the programmer more control
  - In charge of moving memory
  - Potentially faster
- More complex code

```
program main
   use kernelMod
   integer, parameter :: n = 1000000
   real, device :: a_d(n), b_d(n), c_d(n)
   real         :: a_h(n), b_h(n), c_h(n)

   ! Initialise a_h, b_h on the CPU

   ...

   ! Copy to the GPU
   a_d = a_h
   b_d = b_h

   call vecAdd_GPU <<< 1, 1 >>> (c_d, a_d, b_d, n)

   ! Copy the result back from the GPU
   c_h = c_d

   ...
```

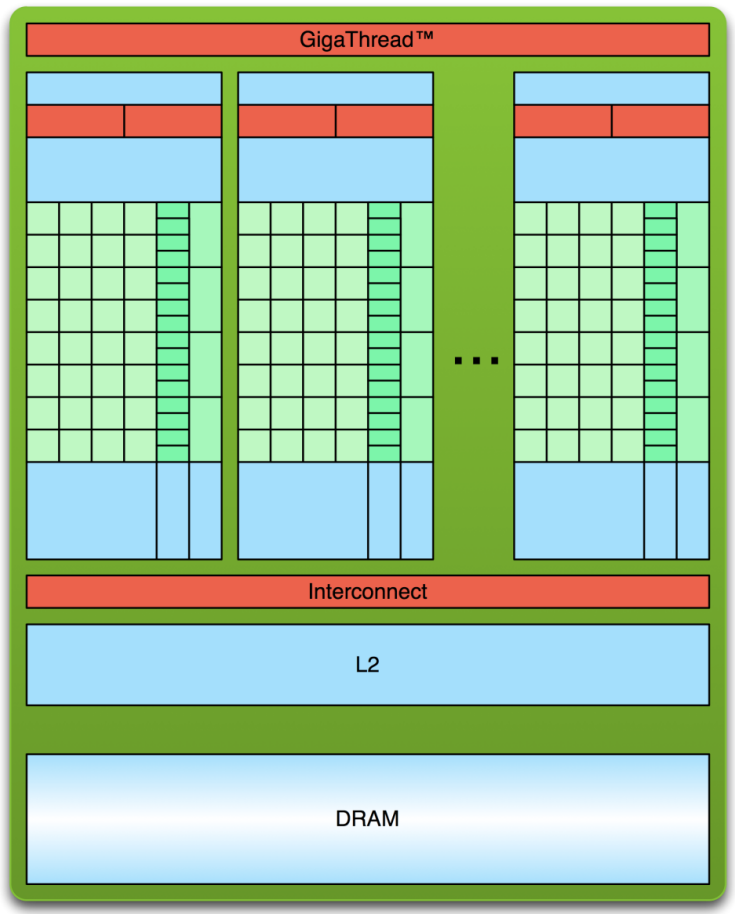
CONCEPTS


Heterogeneous Computing
Running on the GPU
Running in Parallel
Extra Information

GPU Hardware Layout

Streaming Multiprocessors

- GPUs are split into many streaming multiprocessors (SMs)
- Each SM contains many cores
- Each SM also has its own private memory



How Hardware maps to CUDA

Hardware	Programming
One GPU	One CUDA grid
Many Streaming Multiprocessors (SMs)	Many CUDA blocks in a grid
Many cores per SM	Many CUDA threads in a block

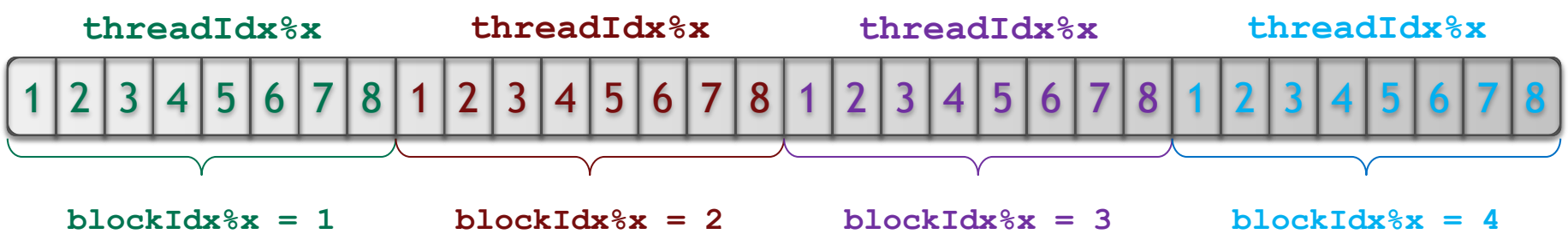
We need many blocks to map to many SMs

We need many threads to map to many cores per SM

We write CUDA from the perspective of a single thread

Indexing Arrays with Blocks and Threads

Consider indexing an array with one element per thread (8 threads/block)



With `blockDim%x` threads per block, a unique index for each thread is given by:

```
index = (blockIdx%x - 1) * blockDim%x + threadIdx%x;
```

CUDA Fortran

Parallel GPU code

- Routine changed so that one element is computed per thread
- This requires:
  - Indexing to get our unique index
  - Bounds checking

```
attributes(global) subroutine vecAdd_GPU(c, a, b, n)
   INTEGER, value :: n
   REAL, device, intent(in)  :: a(n), b(n)
   REAL, device, intent(out) :: c(n)

   INTEGER :: i

   i = (blockIdx%x - 1) * blockDim%x + threadIdx%x

   if (i <= n) then
      c(i) = a(i) + b(i)
   end if
end subroutine vecAdd_GPU
```

Running in Parallel

Host code

- Create a grid of 1024 blocks
- Each has 1024 threads
- Note: this is why we might want to do bounds checking, as n < 1024 * 1024.

```
program main
   use kernelMod
   use cudafor
   integer, parameter :: n = 1000000
   type(dim3) :: blckSz, grdSz

   ...

   blckSz = dim3(1024,1,1)
   grdSz = dim3(1024,1,1)

   call vecAdd_GPU <<< grdSz, blckSz >>> (c_d, a_d, b_d, n)

   ...
```

Performance

Kepler K20x

- Kernel runtime with 1 thread, 1 block:

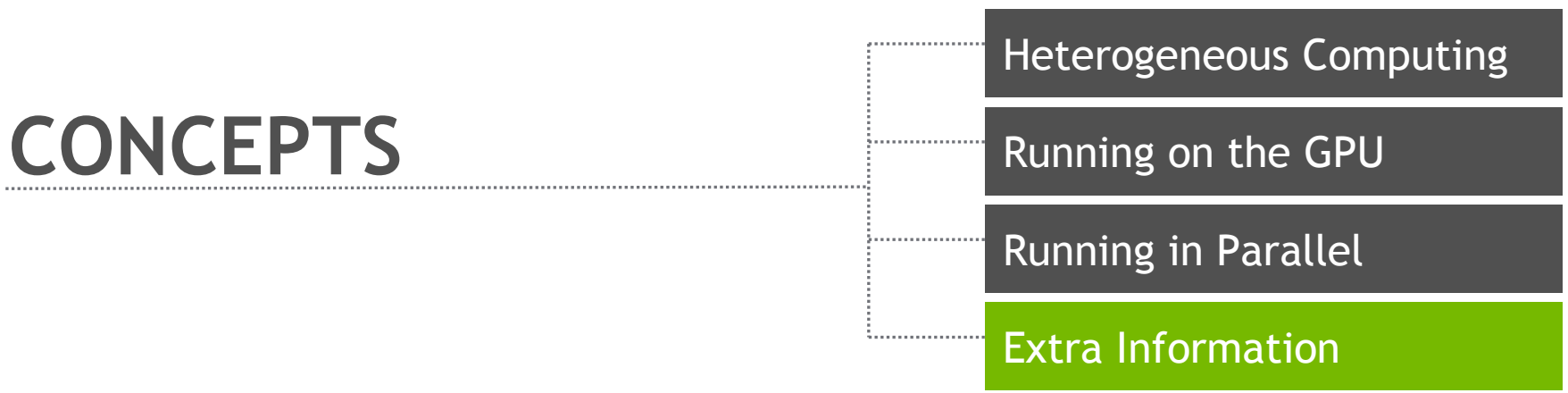
  241.26**ms**

- Kernel runtime with 1024 threads, 1024 blocks:

  66.18**us**

- 4000x faster

CONCEPTS


Heterogeneous Computing
Running on the GPU
Running in Parallel
Extra Information

Shared Memory

- A SM has a small (~50k) region of memory which is know as shared memory
- Shared memory is fast and accessible by all threads within a block
- Allows block-wide co-operation
  - Each thread in a block has an efficient way of passing data to other threads
- Synchronization primitives prevent race conditions (`__syncthreads()`)

Divergence and branching

Performance tip

GPUs work fastest if the following criteria are met:

- A group of 32 threads access a contiguous 32/64/128 byte memory space in the same instruction
  - Good: `a(threadIdx%x)`. Thread 2 will access an adjacent element to thread 1, etc.
  - Bad: `a(threadIdx%x * 1000)`. There will be a stride of 1000 between accesses.
- If the code branches a group of 32 threads should (ideally) take the same path
  - Small branching is usually ok
  - Large branching may not be

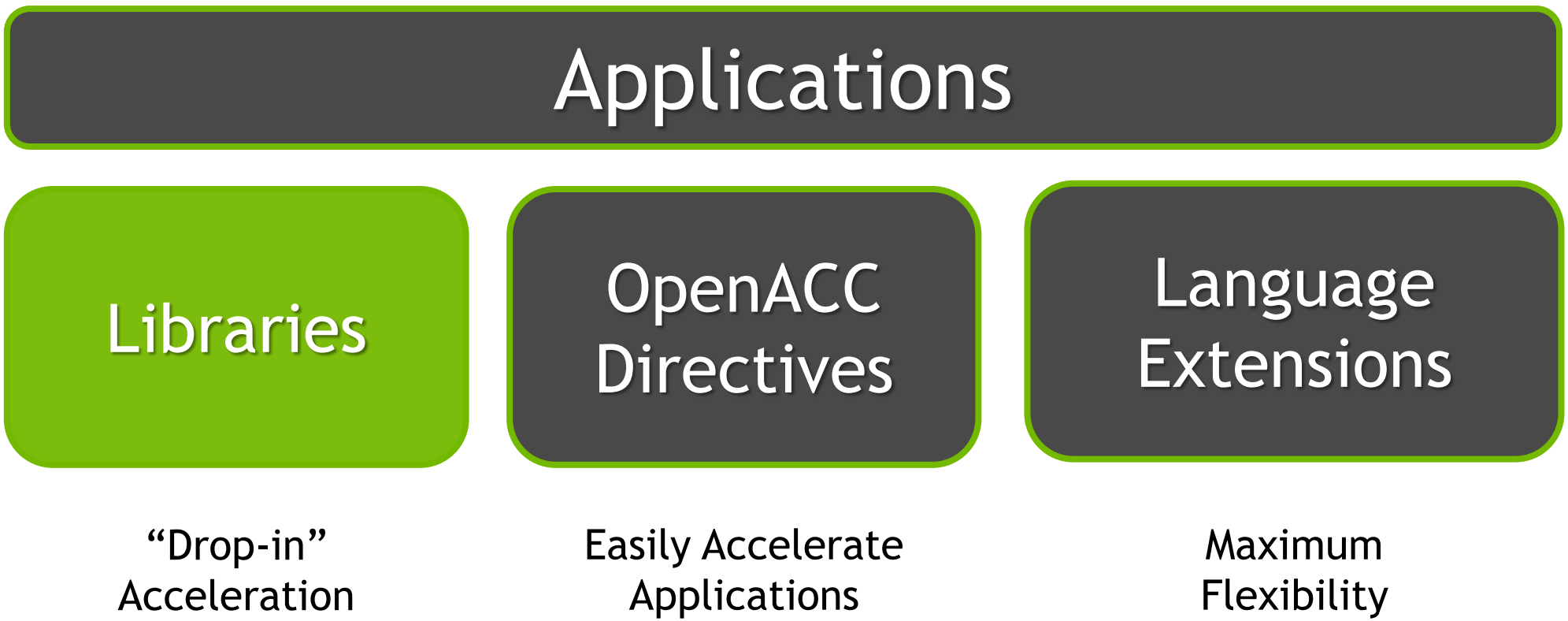
Only the tip of the iceberg

More resources available online

- There are more advanced concepts which have not been covered here.
- CUDA Fortran Programming Guide:
  - http://www.pgroup.com/doc/pgicudaforug.pdf
- A good reference is the CUDA C programming guide:
  - http://docs.nvidia.com/cuda/cuda-c-programming-guide

GPU Libraries

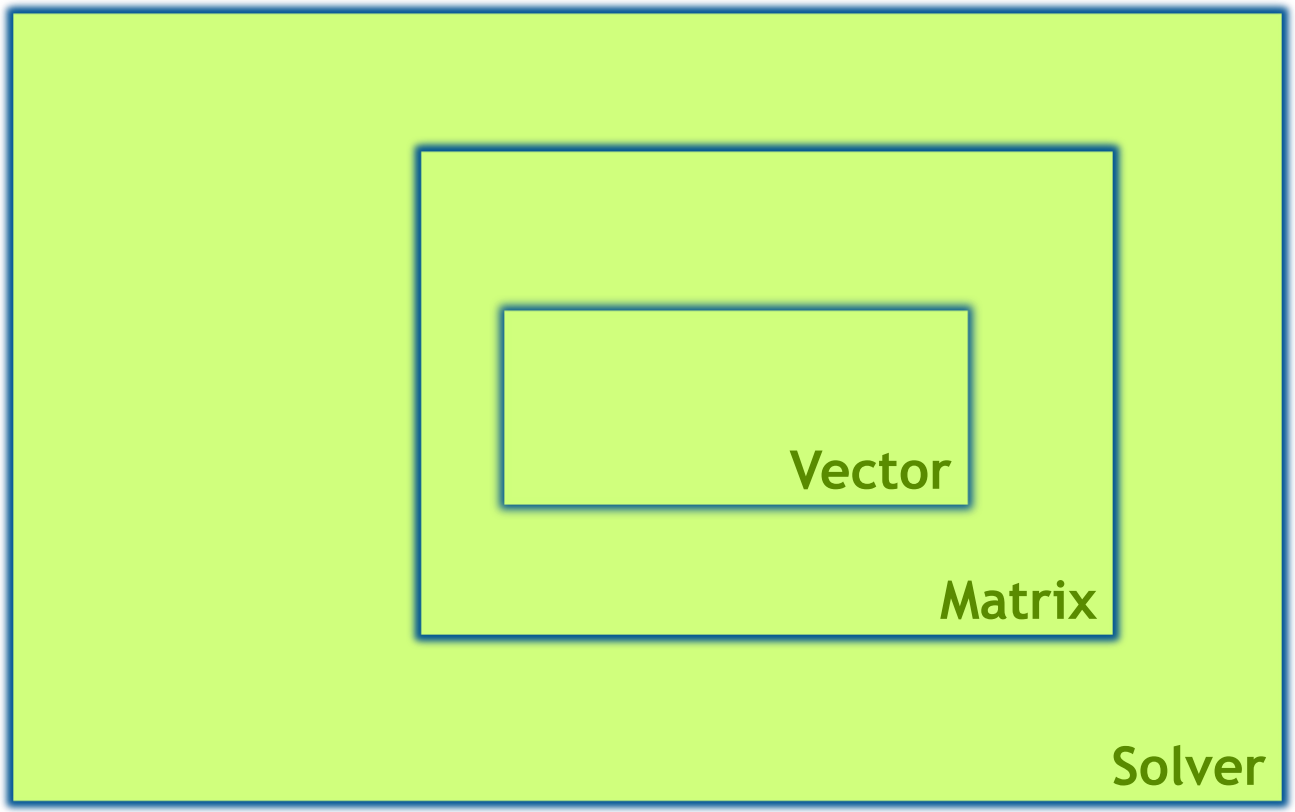
Three Ways to Accelerate Applications



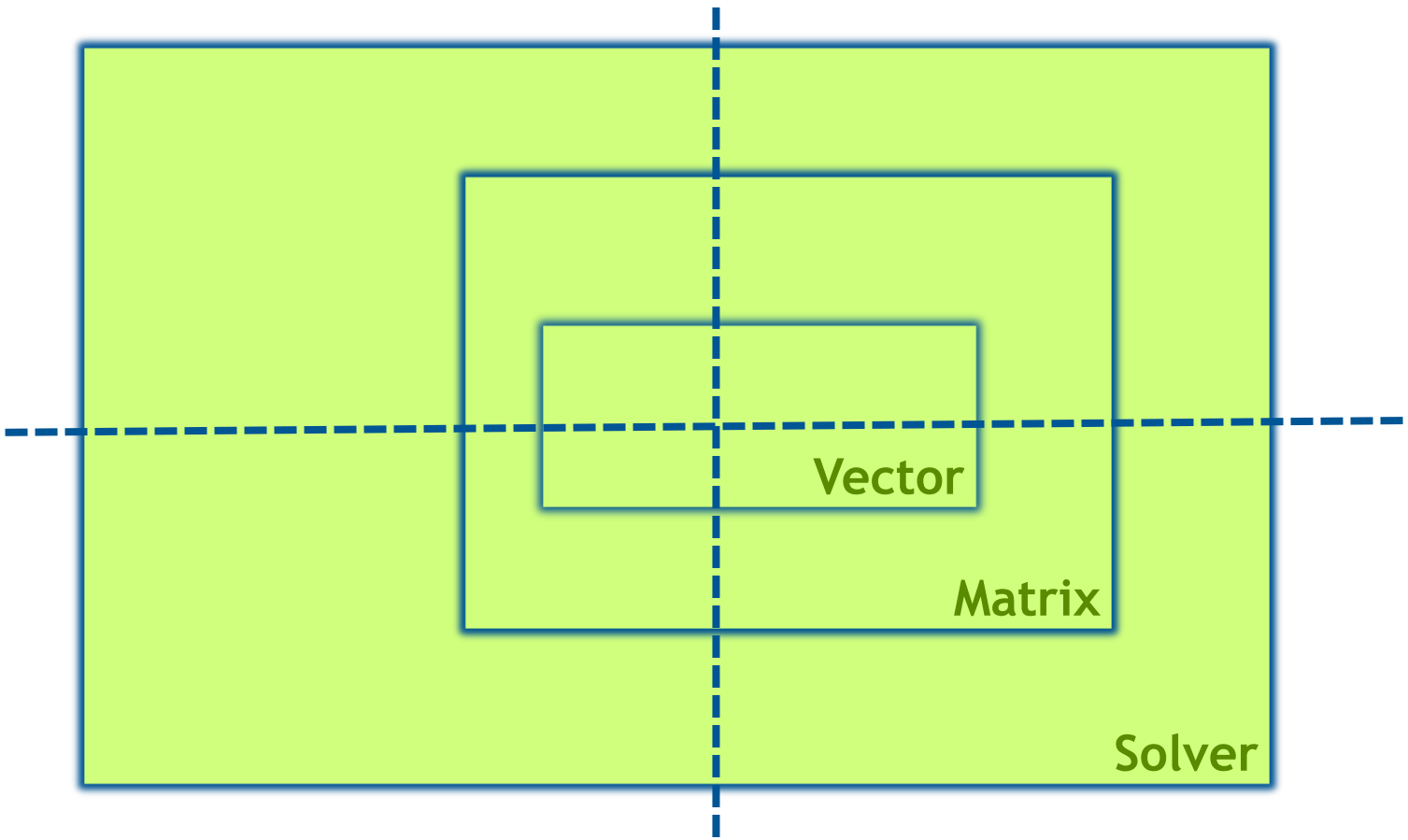
Libraries: Easy, High-Quality Acceleration

- **Ease of use:** Using libraries enables GPU acceleration without in-depth knowledge of GPU programming
- **"Drop-in":** Many GPU-accelerated libraries follow standard APIs, thus enabling acceleration with minimal code changes
- **Quality:** Libraries offer high-quality implementations of functions encountered in a broad range of applications
- **Performance:** NVIDIA libraries are tuned by experts

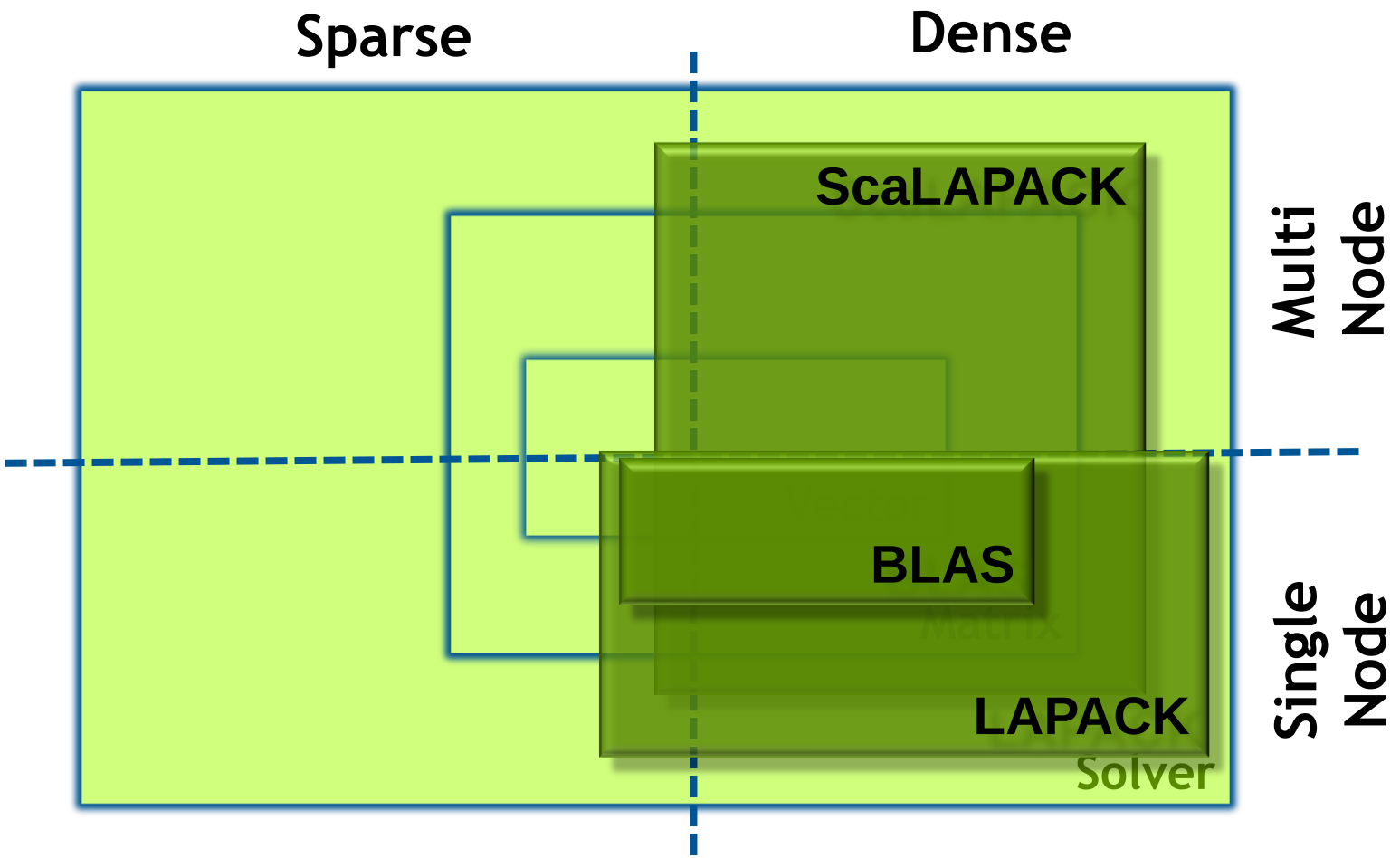
A Birds Eye View on Linear Algebra



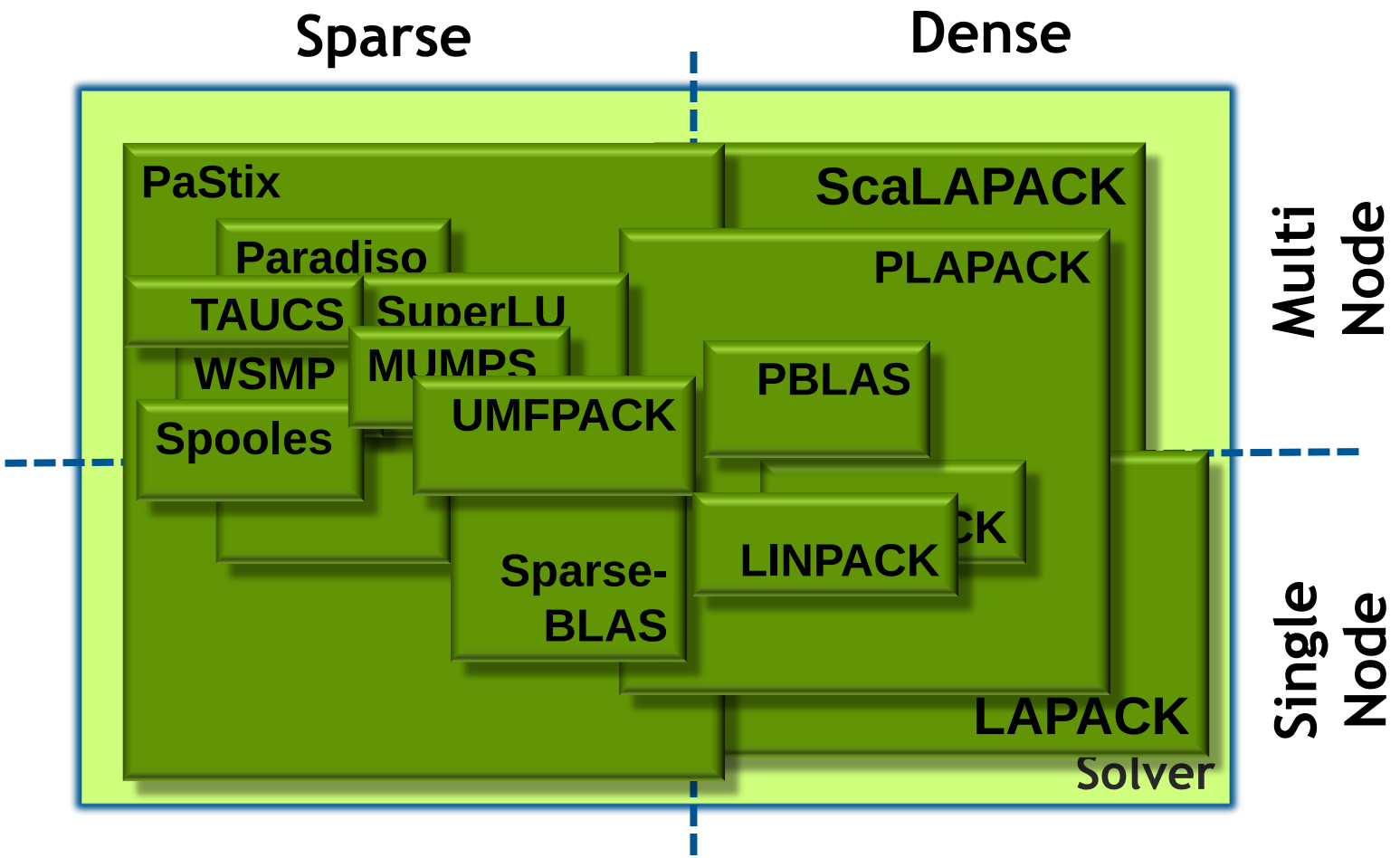
A Birds Eye View on Linear Algebra



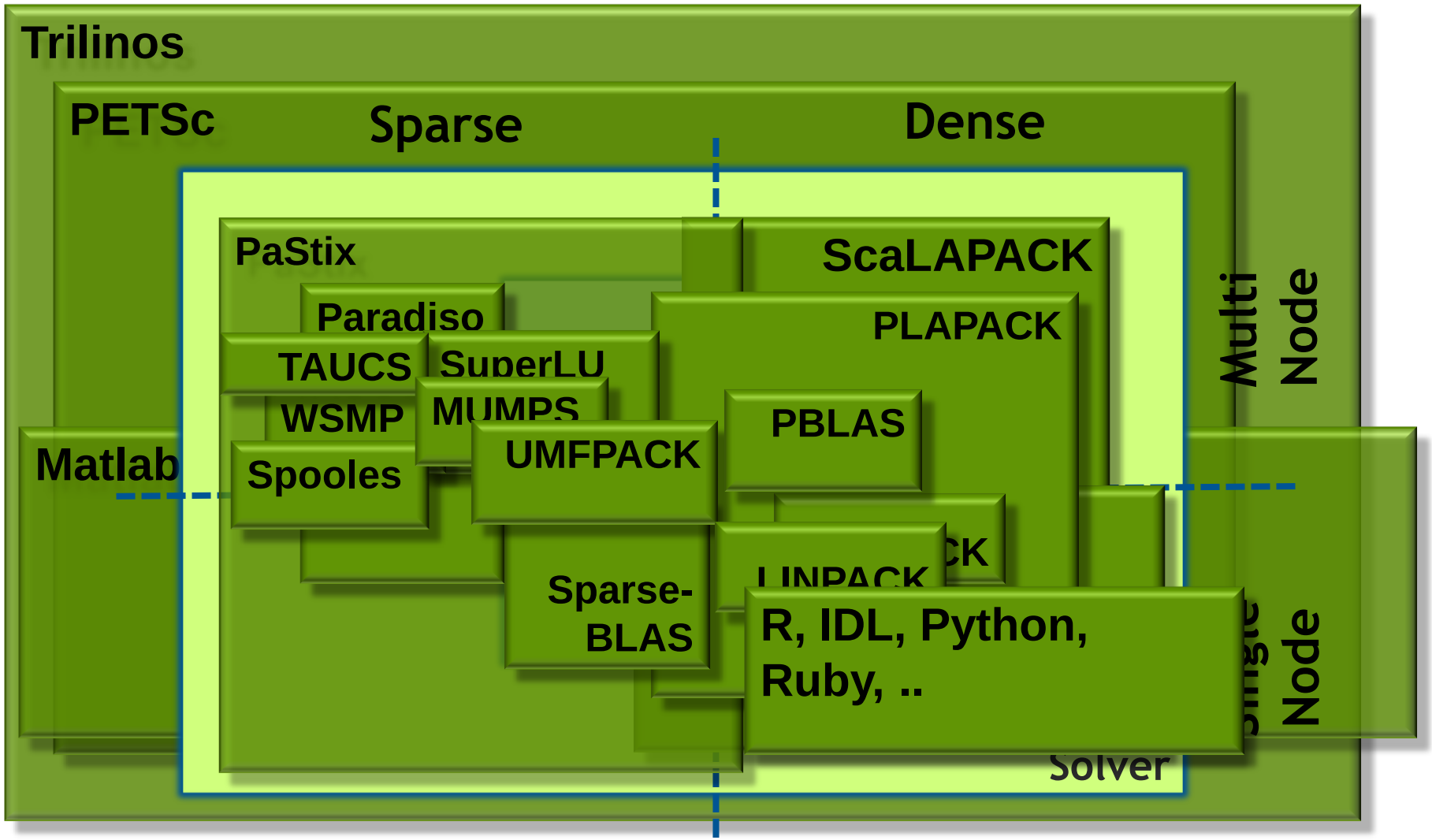
Sometimes it seems as if there's only three …



.. but there is more …

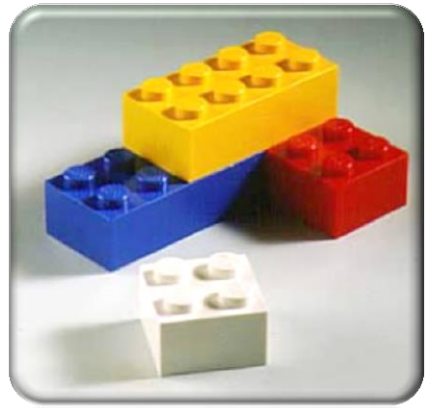


… and even more ..



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

Many available

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
- … and more!

cuBLAS

Dense Linear Algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex
  - Host and device-callable interface
- XT Interface for Level 3 BLAS
  - Distributed computations across multiple GPUs
  - Out-of-core streaming to GPU, no upper limit on matrix size
  - “Drop-in” BLAS intercepts CPU BLAS calls, streams to GPU

cuBLAS Interface

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

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

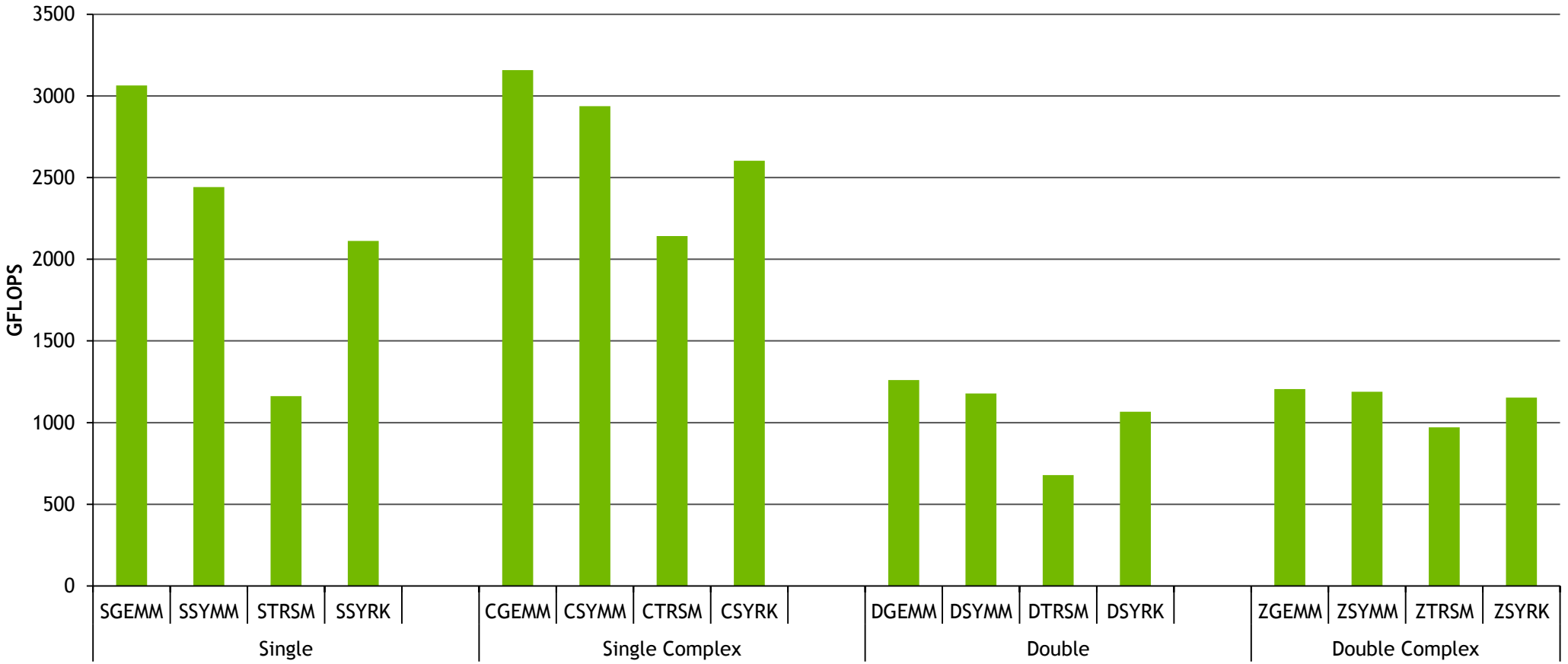
```
dgemm('N','N',m,n,k,A,m,
      B,k,0.0,C,m)
```

```
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)
```

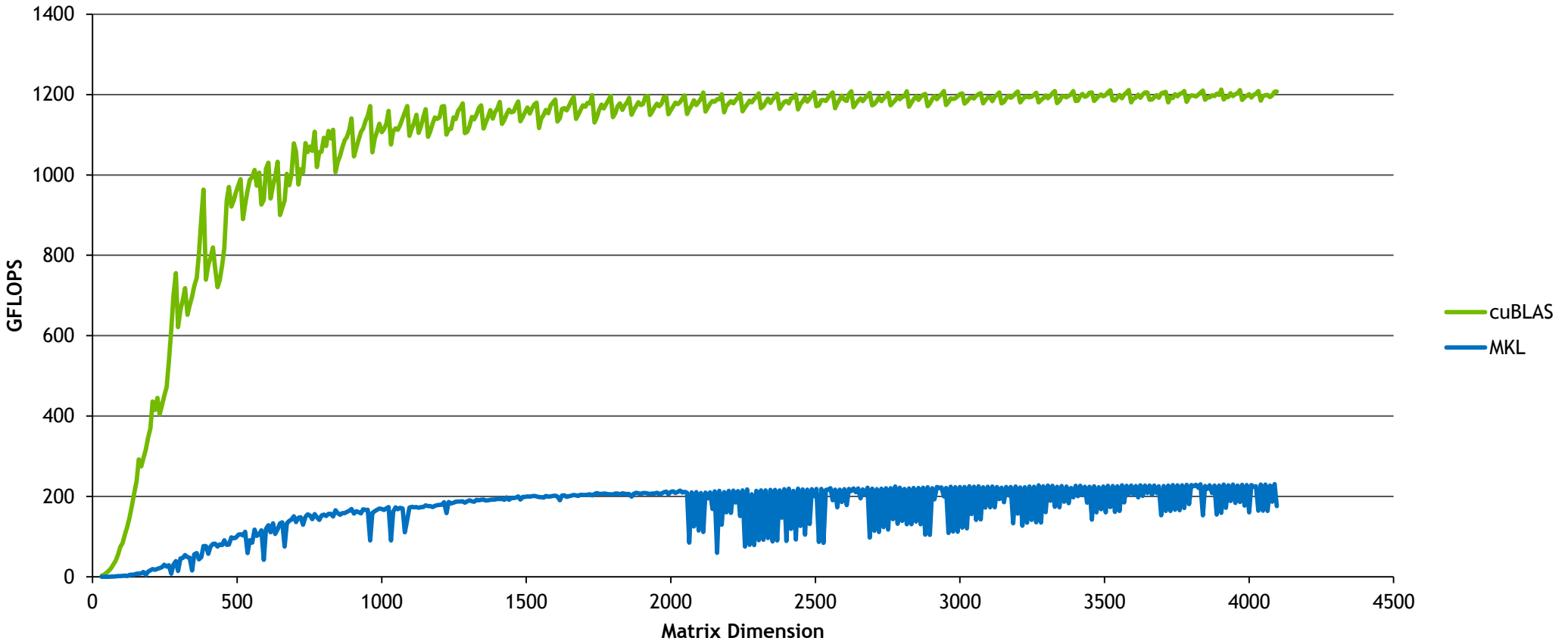
cuBLAS: >3 TFLOPS single-precision
>1 TFLOPS double-precision



Performance may vary based on OS version and motherboard configuration

• cuBLAS 6.0 on K40c, input and output data on host

cuBLAS: ZGEMM 6x Faster than MKL



- cuBLAS 6.0 on K40c, input and output data on device
- MKL 11.0.4 on Intel IvyBridge 12-core E5-2697 v2 @ 2.70GHz

Performance may vary based on OS version and motherboard configuration

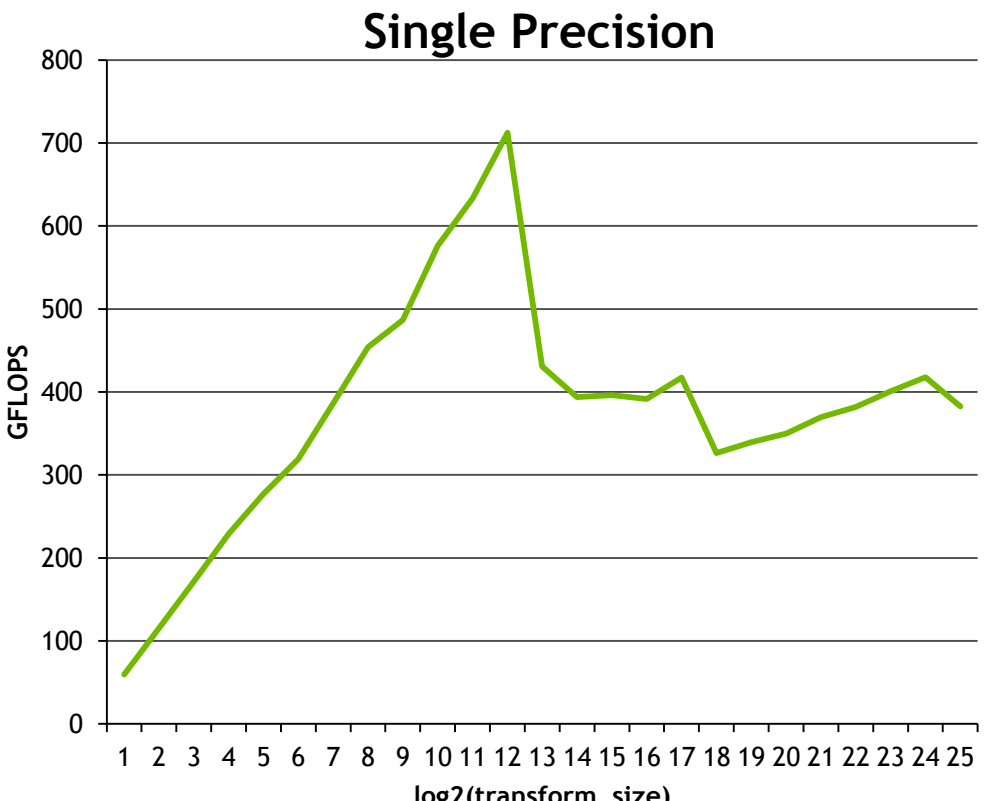
cuFFT

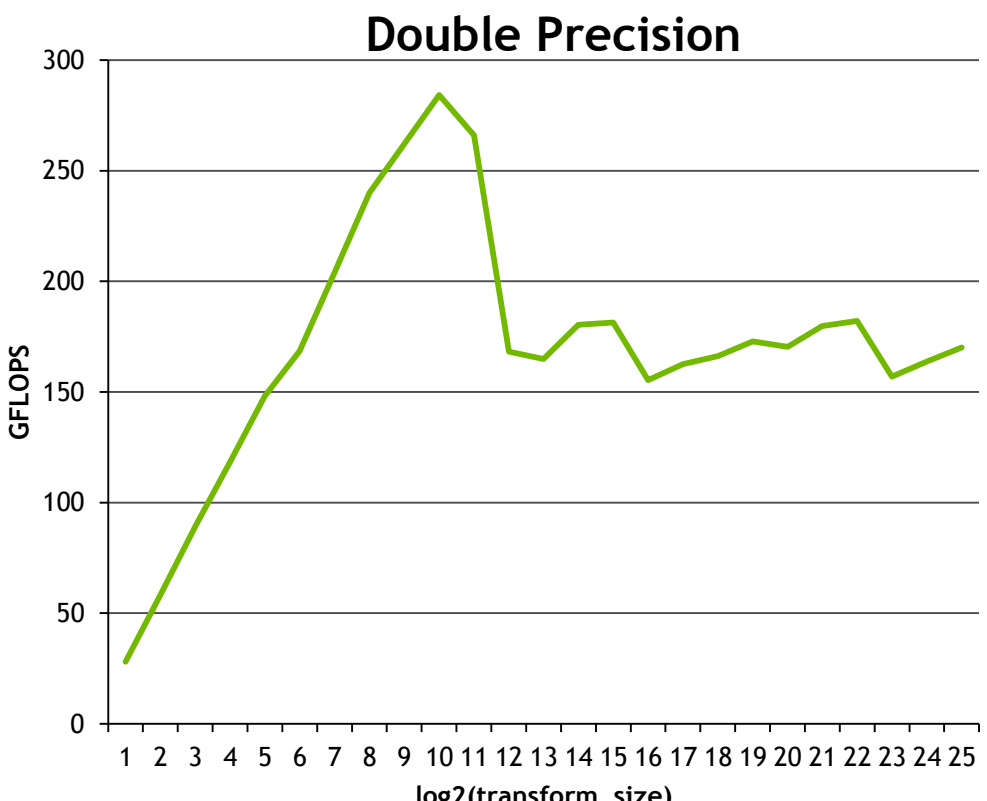
Multi-dimensional FFTs

- Real and complex
- Single- and double-precision data types
- 1D, 2D and 3D batched transforms
- Flexible input and output data layouts

cuFFT: up to 700 GFLOPS

1D Complex, Batched FFTs
Used in Audio Processing and as a Foundation for 2D and 3D FFTs





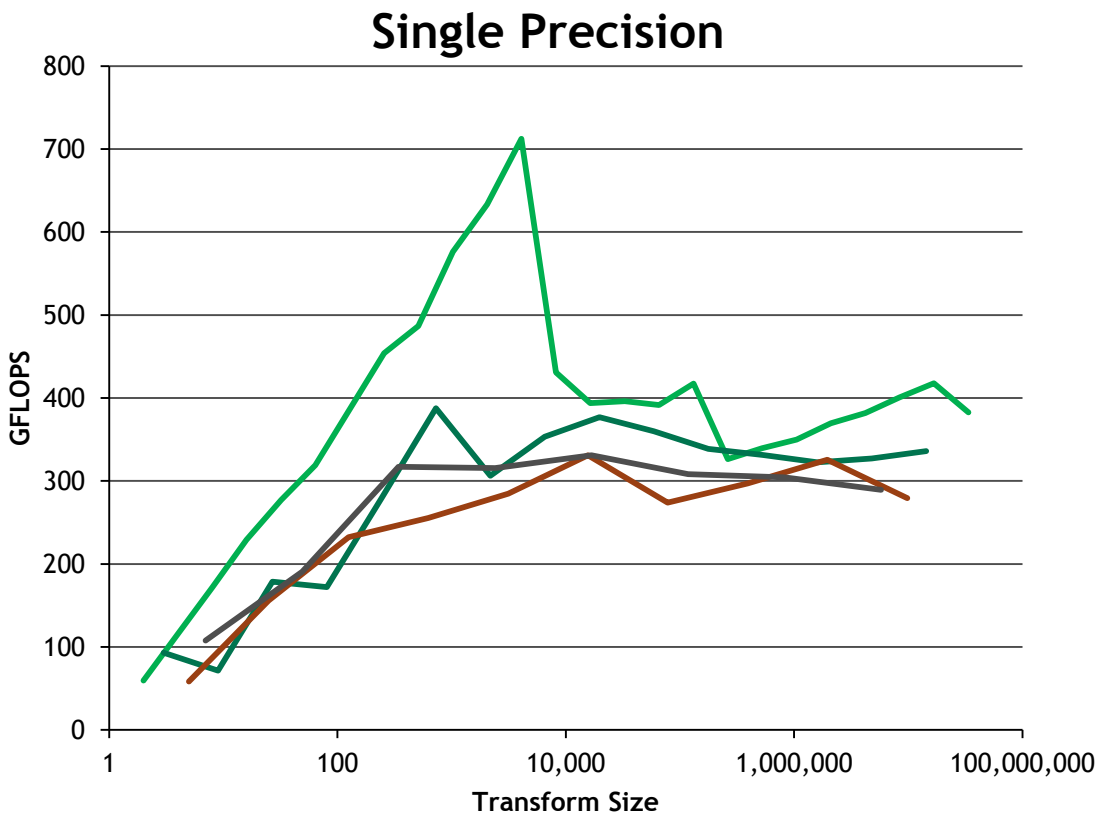
Performance may vary based on OS version and motherboard configuration

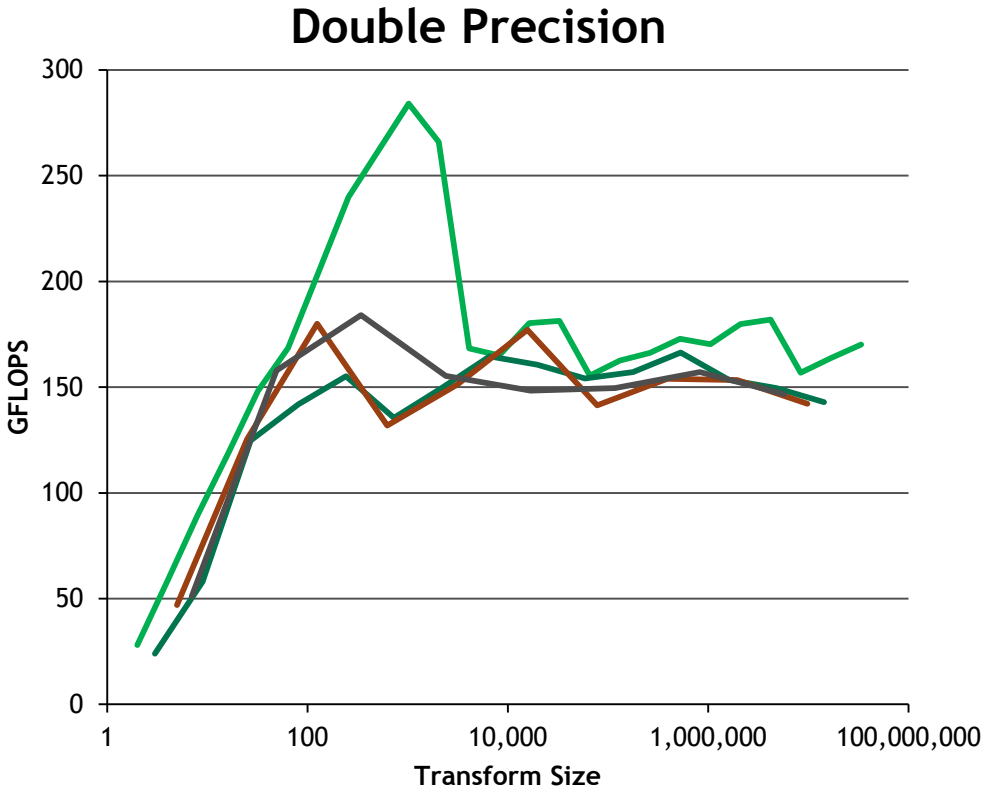
• cuFFT 6.0 on K40c, 32M elements, input and output data on device

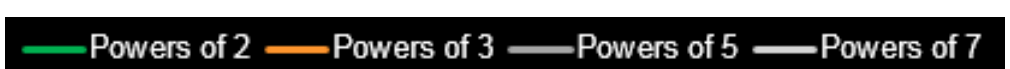
cuFFT: Consistently High Performance

1D Complex, Batched FFTs

Used in Audio Processing and as a Foundation for 2D and 3D FFTs







Performance may vary based on OS version and motherboard configuration • cuFFT 6.0 on K40c, 28M-33M elements, input and output data on device

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;

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

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


  /* initialize idata  */
  …
```

```
  /* 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

cuSPARSE:

Sparse linear algebra routines

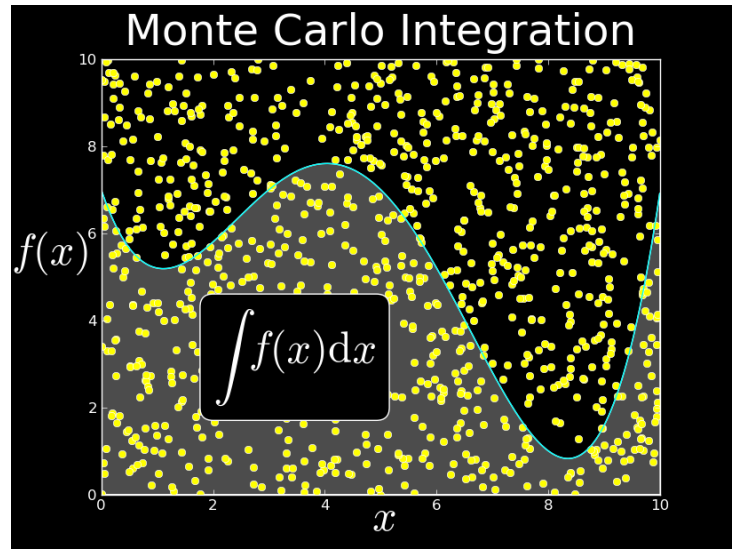
- Optimized sparse linear algebra BLAS routines – matrix-vector, matrix-matrix, triangular solve
- Support for variety of formats (CSR, COO, block variants)
- For example spmv:

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

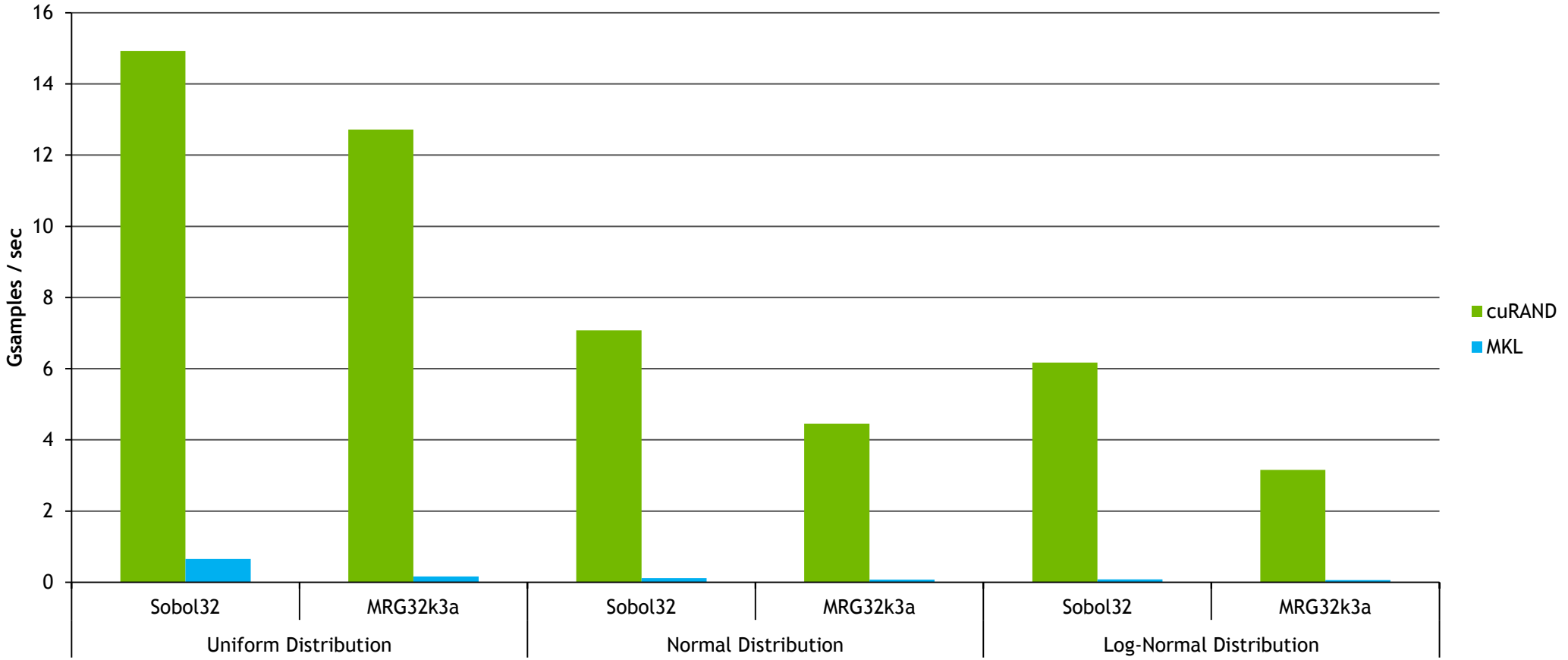
cuRAND

Random Number Generation

- Generating high quality random numbers in parallel is hard
  - Don’t do it yourself, use a library!
- Pseudo- and Quasi-RNGs
- Supports several output distributions
- Statistical test results in documentation



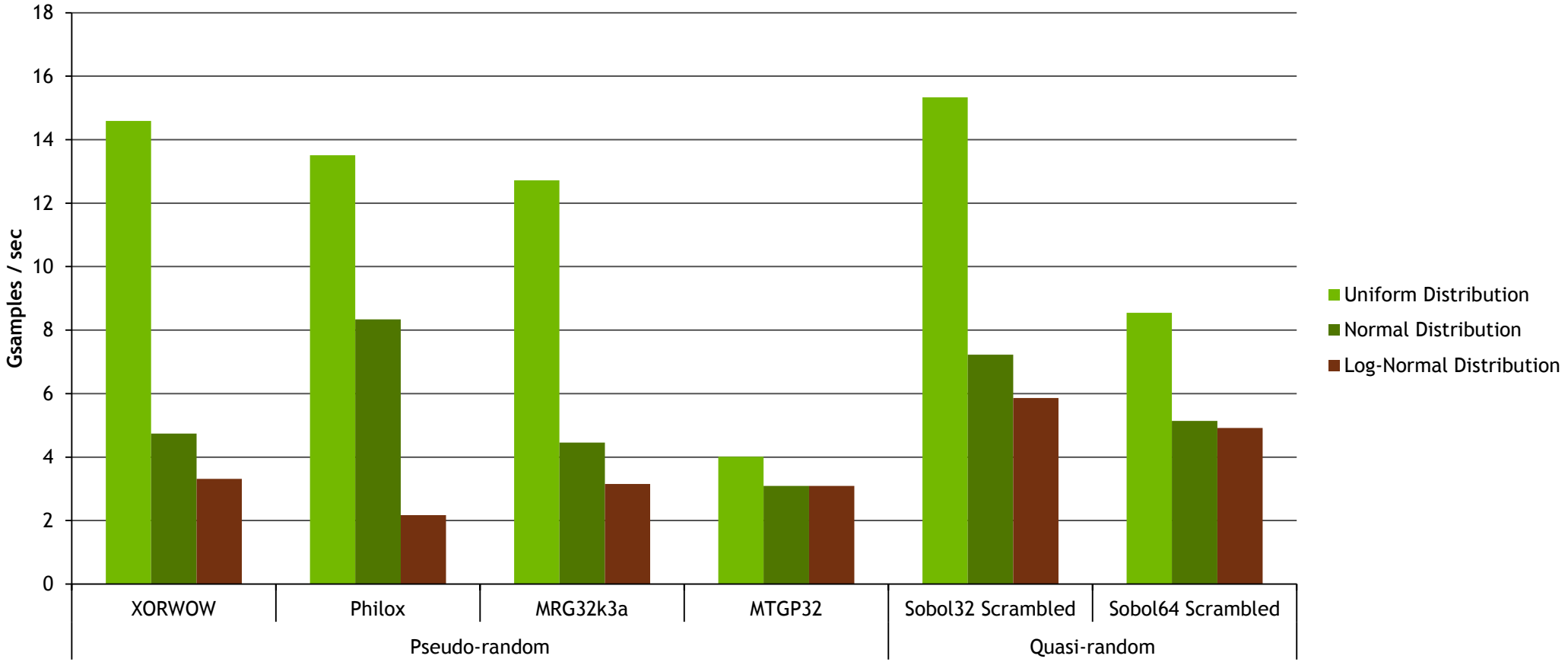
cuRAND up to 75x Faster vs. Intel MKL



- cuRAND 6.0 on K40c, double-precision input and output data on device
- MKL 11.0.1 on Intel SandyBridge 6-core E5-2620 @ 2.0 GHz

Performance may vary based on OS version and motherboard configuration

cuRAND: High Performance RNGs



- cuRAND 6.0 on K40c, double precision input and output data on device
- MKL 11.0.1 on Intel SandyBridge 6-core E5-2620 @ 2.0 GHz

Performance may vary based on OS version and motherboard configuration

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 software

Three Ways to Accelerate Applications