

# Lecture 5: Libraries and tools

Prof Wes Armour

[wes.armour@eng.ox.ac.uk](mailto:wes.armour@eng.ox.ac.uk)

Prof Mike Giles

[mike.giles@maths.ox.ac.uk](mailto:mike.giles@maths.ox.ac.uk)

Oxford e-Research Centre

Department of Engineering Science

# Learning outcomes

In this fifth lecture we will learn about GPU libraries and tools for GPU programming.

Specifically:

- NVIDIA GPU libraries and their usefulness in scientific computing.
- Third party libraries.
- Directives based approaches to GPU computation.
- Tools for GPU programming.

# Software overview

NVIDIA provides a **rich ecosystem of software tools** that allow you to easily utilise GPUs in your project.

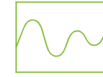
During this lecture we will focus on a range of **libraries and software tools that will make your life easier** when either writing CUDA code or when utilising GPUs in your projects.



## cuBLAS

GPU-accelerated basic linear algebra (BLAS) library.

[Learn More >](#)



## cuFFT

GPU-accelerated library for Fast Fourier Transform implementations.

[Learn More >](#)



## cuRAND

GPU-accelerated random number generation.

[Learn More >](#)



## cuSOLVER

GPU-accelerated dense and sparse direct solvers.

[Learn More >](#)



## cuSPARSE

GPU-accelerated BLAS for sparse matrices.

[Learn More >](#)



## cuTENSOR

GPU-accelerated tensor linear algebra library.

[Learn More >](#)



## cuDSS

GPU-accelerated direct sparse solver library.

[Learn More >](#)



## CUDA Math API

GPU-accelerated standard mathematical function APIs.

[Learn More >](#)



## AmgX

GPU-accelerated linear solvers for simulations and implicit unstructured methods.

[Learn More >](#)

<https://developer.nvidia.com/gpu-accelerated-libraries>

# Dependencies – Advantages / Disadvantages

## Some advantages:

- ***Simple to use*** – you don't need to write your own complex code to perform a specific task.
- ***Well maintained*** – always benefit from the latest optimisations and improvements.
- ***Easier(?)*** to move from CPU code to GPU code (for example see cuFFTW).

## Some disadvantages:

- ***Reduces portability*** - can make installing your code on another system harder (the user also needs to have the dependency installed).
- ***Reliance on a third party*** - If the dependency isn't maintained it could break your code as other things (e.g. compiler) are updated.
- ***Inheritance*** - If your code is very dependent on it and the developers stop supporting it – you become the owner (not a great position to be in).

# CUDA math library

The CUDA Math Library contains all of the typical mathematical functions that you will need for your projects. *It is very similar to Intel's MKL library.*

- various **exponential** and **log** functions
- **trigonometric** functions and their inverses
- **hyperbolic** functions and their inverses
- **error functions** and their inverses
- **Bessel** and **Gamma** functions
- **vector norms** and reciprocals (esp. for graphics)
- **To use** - “`#include math.h`”

The library supports standard int, float and double types, but in recent years has also added support\* for fp4, fp6, fp8, fp16 and bfloat16.

Typecasting and SIMD intrinsics are also included in this library.

\* *To use these lower precision types, you must include the appropriate header, e.g. `cuda_fp4.h`*



## CUDA Math Library

GPU-accelerated standard mathematical  
function library

# cuBLAS Library

The cuBLAS (CUDA Basic Linear Algebra Subprograms) library provides CUDA accelerated standard BLAS APIs (for 152 different routines) for dense matrices.

- includes **matrix-vector** and **matrix-matrix** product.
- it is possible to call cuBLAS routines from user kernels (via a device API).
- some support for a single routine call to do a “batch” of smaller matrix-matrix multiplications.
- also support for using CUDA streams to do a large number of small tasks concurrently.
- has support for **multi-GPU operation** (cuBLASTxt or cuBLASmg).
- has **mixed / low precision** implementations.



**cuBLAS**

GPU-accelerated basic linear algebra  
(BLAS) library

# cuBLAS Library

To use cuBLAS in your codes, a set of routines are called from your host code. These come in two forms, helper routines and compute routines.

**Helper** routines for:

- memory allocation
- data copying from CPU to GPU, and vice versa
- error reporting

**Compute** routines for:

- e.g. matrix-matrix and matrix-vector product
- **Warning!** Some calls are asynchronous, i.e. the call starts the operation but the host code then continues before it has completed!!



**cuBLAS**

GPU-accelerated basic linear algebra  
(BLAS) library

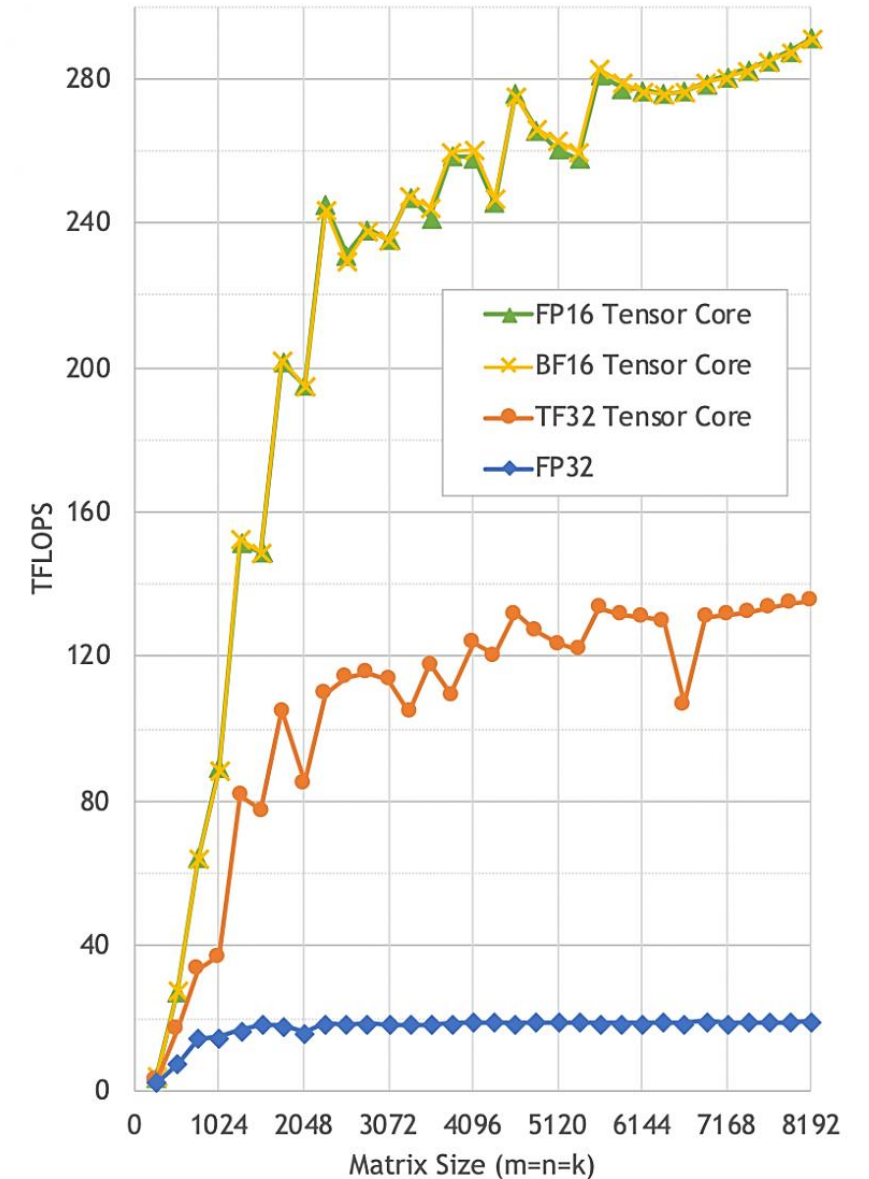
# cuBLAS Library

cuBLAS is one of three libraries that utilize '**tensor cores**'. Tensor cores differ from standard processing cores in that they are designed to perform very specific operations, and these operations are executed on mixed-precision data.

By reducing the precision of your matrix/vector operations in cuBLAS, you can achieve significant acceleration.

<https://developer.nvidia.com/cublas>

Mixed Precision Matrix Multiply on A100



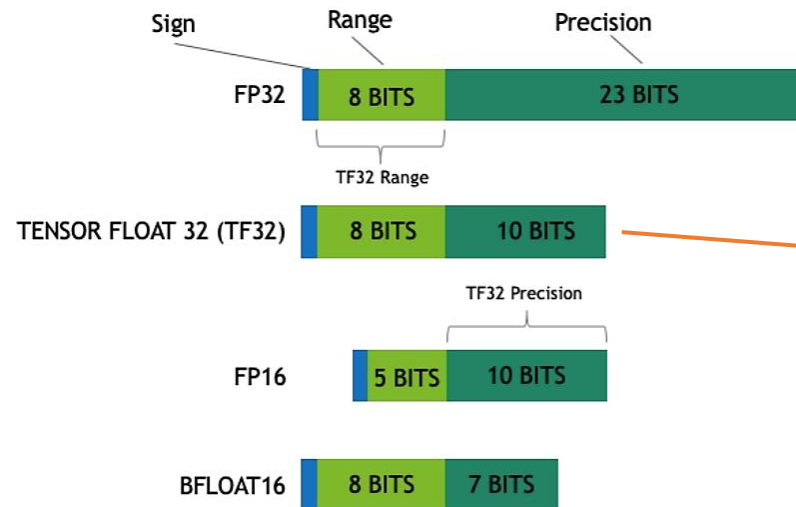


# cuBLAS Library

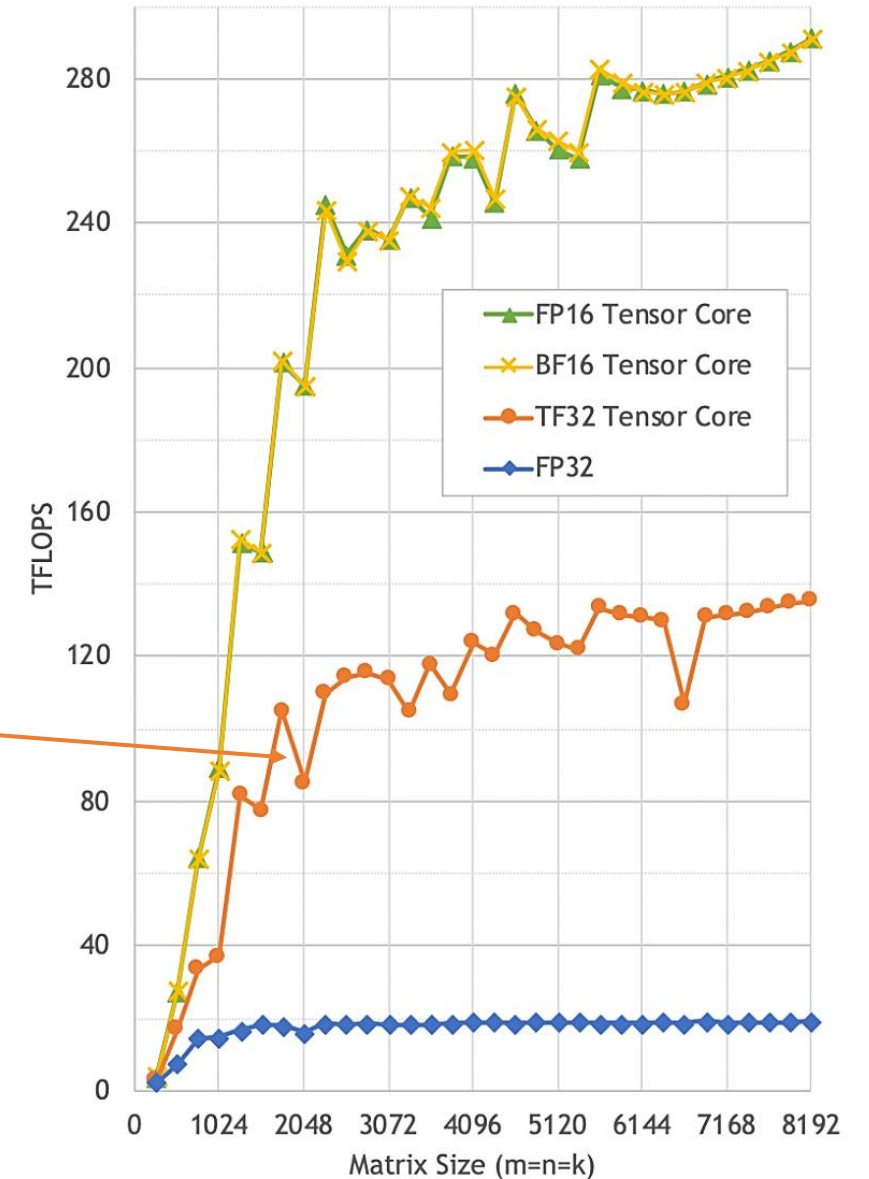
Since the introduction of Ampere, there has been a '**TF32**' variable that serves as a compromise between FP32 and BFLOAT16. It allows for a balance between lower precision and FP32 matrix-matrix operations.

Useful for:

- AI Training
- Linear solvers



Mixed Precision Matrix Multiply on A100



# cuBLAS Example

Here is an example of using cuBLAS to perform the SAXPY operation.

SAXPY stands for *Single-Precision A·X Plus Y*

$$y = \alpha x + y$$

Note the use of cuBLAS\_v2.h, cuBLAS.h contains legacy interfaces so you should pick which best fits with your needs.

<https://docs.nvidia.com/cuda/archive/11.4.4/cublas/index.html>

```
#include <stdio.h>
#include <stdlib.h>
#include <cuda_runtime.h>
#include <cublas_v2.h>

#define N 8

int main(void) {
    float *x, *y;
    cudaMallocManaged(&x, N * sizeof(float));
    cudaMallocManaged(&y, N * sizeof(float));

    // Initialize x and y arrays here
    for (int i = 0; i < N; i++) {
        x[i] = -1.0f;
        y[i] = (float) (i + 3);
    }

    cublasHandle_t handle;
    cublasCreate(&handle);

    const float alpha = 2.0f;
    cublasSaxpy(handle, N, &alpha, x, 1, y, 1);

    // Synchronize device and check for errors
    cudaDeviceSynchronize();

    // Check the result
    float maxError = 0.0f;
    for (int i = 0; i < N; i++) {
        maxError = fmax(maxError, fabs(y[i] - (float)(i + 1)));
        printf("y[%d]:\t%f\n", i, y[i]);
    }
    printf("Max error: %f\n", maxError);

    cudaFree(x);
    cudaFree(y);

    cublasDestroy(handle);

    return 0;
}
```

# cuBLAS Example

Compile with:

```
$ nvcc cuBLAS_example.cu -o test -lcublas
```

Results are as we would expect.

We have initialised  $x$  to be -1.0, we have multiplied it by the scalar  $\alpha$  which we set equal to 2.0.

We then add the output to  $y$ , which was initialised from 0(+3) to 7(+3).

```
y[0]: 1.000000  
y[1]: 2.000000  
y[2]: 3.000000  
y[3]: 4.000000  
y[4]: 5.000000  
y[5]: 6.000000  
y[6]: 7.000000  
y[7]: 8.000000  
Max error: 0.000000
```

<https://developer.nvidia.com/blog/six-ways-saxpy/>

# cuTENSOR

Tensor cores, originally introduced on Volta in 2017, provided hardware-enabled acceleration for matrix-matrix multiplies.

- These cores originally performed a 4x4 matrix multiply-accumulate (**think of it as a FMA for matrices**) using the **wmma::instruction**.
- Matrices **A** and **B** would be lower precision and the **accumulators** would be the same or higher precision.
- With Ampere and Hopper, some of these restrictions have been relaxed.

$$D = A * B + C$$

The diagram shows the operation  $D = A * B + C$  with 4x4 matrices. Matrix A is teal, Matrix B is purple, and Matrix C is light green. Matrix D is the result of the operation. Below the matrices, the precision requirements are listed:

Matrix	Operation	Precision
A	HMMA	FP16 or FP32
B	IMMA	INT32
C		FP16 or FP32
D		INT8 or UINT8

<https://arxiv.org/pdf/2206.02874.pdf>

<https://developer.nvidia.com/blog/programming-tensor-cores-cuda-9/>

<https://developer.nvidia.com/blog/nvidia-automatic-mixed-precision-tensorflow/>

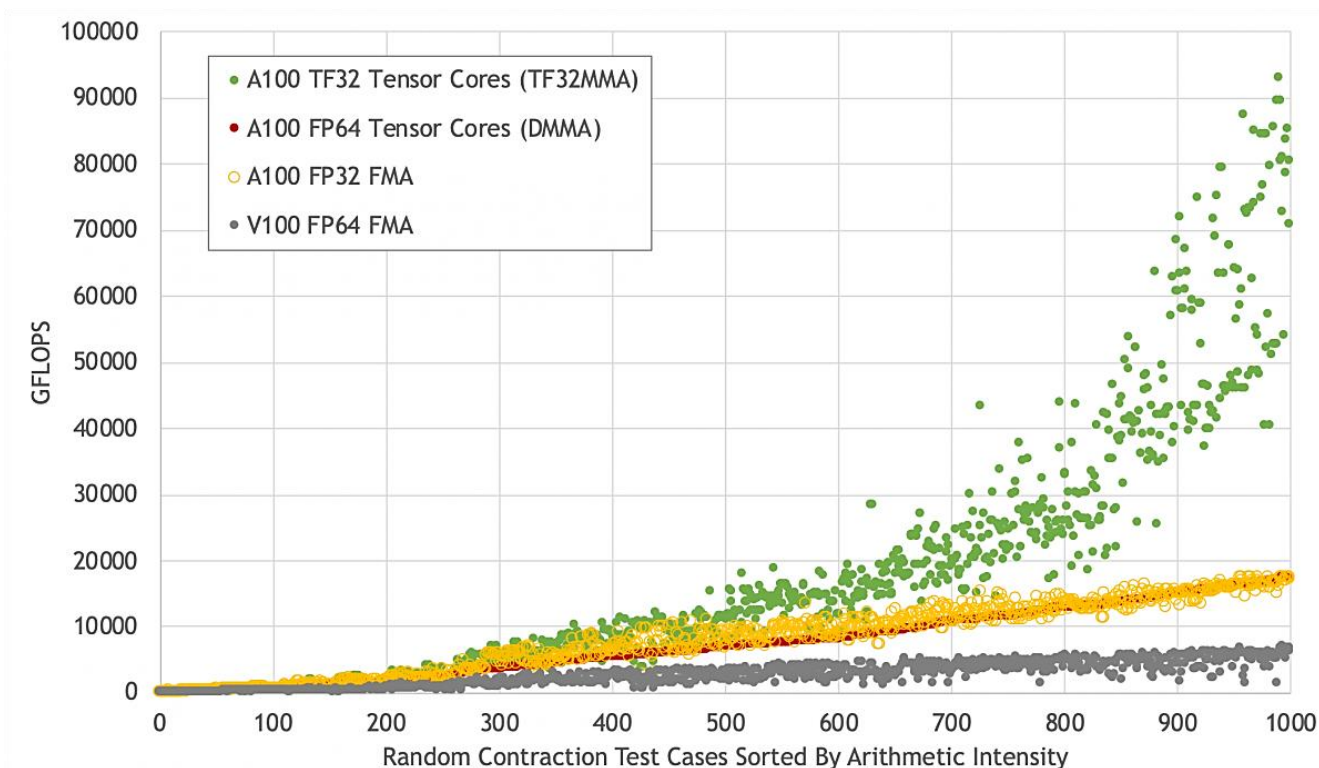
<https://developer.download.nvidia.com/video/gputechconf/gtc/2019/presentation/s9593-cutensor-high-performance-tensor-operations-in-cuda-v2.pdf>

# cuTENSOR

Current tensor cores, such as those in Hopper, can use various precisions and exploit sparsity to achieve further acceleration.

- FP64 inputs with FP32 compute (DMMA).
- FP32 inputs with FP16, BF16, or TF32 compute.
- Complex-times-real operations.
- Conjugate support (without the need for transposition).
- Support for tensors with up to 64-dimensions.

<https://developer.nvidia.com/blog/nvidia-hopper-architecture-in-depth/>



<https://developer.nvidia.com/cutensor>

<https://docs.nvidia.com/cuda/cutensor/index.html>

<https://github.com/NVIDIA/CUDALibrarySamples/blob/master/cuTENSOR/reduction.cu>

# cuTENSOR

A useful description of the low-level tensor operations can be found here:

<https://docs.nvidia.com/cuda/ampere-tuning-guide/index.html#improved-tensor-core-operations>

(note this is for Ampere)

Here you can find code for a double precision GEMM computation using the Double precision Warp Matrix Multiply and Accumulate (DMMA) here:

[https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3\\_CUDA\\_Features/dmmaTensorCoreGemm](https://github.com/NVIDIA/cuda-samples/tree/master/Samples/3_CUDA_Features/dmmaTensorCoreGemm)

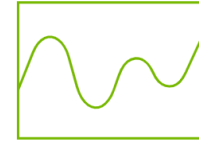
***Note the differences between generations***

Instruction	GPU Architecture	Input Matrix format	Output Accumulator format	Matrix Instruction Size (MxNxK)
HMMA (16-bit precision)	NVIDIA Volta Architecture	FP16	FP16 / FP32	8x8x4
	NVIDIA Turing Architecture	FP16	FP16 / FP32	8x8x4 / 16x8x8 / 16x8x16
	NVIDIA Ampere Architecture	FP16 / BFloat16	FP16 / FP32 (BFloat16 only supports FP32 as accumulator)	16x8x8 / 16x8x16
HMMA (19-bit precision)	NVIDIA Volta Architecture	N/A	N/A	N/A
	NVIDIA Turing Architecture	N/A	N/A	N/A
	NVIDIA Ampere Architecture	TF32 (19-bits)	FP32	16x8x4
IMMA (Integer MMA)	NVIDIA Volta Architecture	N/A	N/A	N/A
	NVIDIA Turing Architecture	unsigned char/signed char (8-bit precision)	int32	8x8x16
	NVIDIA Ampere Architecture	unsigned char/signed char (8-bit precision)	int32	8x8x16 / 16x8x16 / 16x8x32
IMMA (Integer sub-byte MMA)	NVIDIA Volta Architecture	N/A	N/A	N/A
	NVIDIA Turing Architecture	unsigned u4/signed u4 (4-bit precision)	int32	8x8x32
	NVIDIA Ampere Architecture	unsigned u4/signed u4 (4-bit precision)	int32	8x8x32 / 16x8x32 / 16x8x64
BMMA (Binary MMA)	NVIDIA Volta Architecture	N/A	N/A	N/A
	NVIDIA Turing Architecture	single bit	int32	8x8x128
	NVIDIA Ampere Architecture	single bit	int32	8x8x128 / 16x8x128 / 16x8x256
DMMA (64-bit precision)	NVIDIA Volta Architecture	N/A	N/A	N/A
	NVIDIA Turing Architecture	N/A	N/A	N/A
	NVIDIA Ampere Architecture	FP64	FP64	8x8x4

# cuFFT Library

The cuFFT library is a GPU accelerated library that provides Fast Fourier Transforms (FFTs).

- It provides **1D, 2D** and **3D FFTs**.
- It encompasses almost all of the variations found in FFTW and other CPU libraries.
- It **includes the cuFFTW library**, a porting tool, to enable users of FFTW to start using GPUs with minimal effort.
- **Provides some device level functionality** (*If this is something of interest, ask me, we have already produced shared memory device level FFTs for some of our projects*).



**cuFFT**

GPU-accelerated library for Fast Fourier  
Transforms

# cuFFT Library

cuFFT is used exactly like cuBLAS - it has a set of routines called by host code:

Helper routines include “plan” construction.

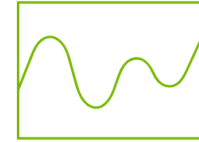
Compute routines perform 1D, 2D, 3D FFTs:

- `cufftExecC2C()` - complex-to-complex.
- `cufftExecR2C()` - real-to-complex.
- `cufftExecC2R()` - complex-to-real.

(double precision routines have different function calls, e.g. `cufftExecZ2Z()`)

It supports doing a “batch” of independent transforms, e.g. applying 1D transform to a 3D dataset.

The `simpleCUFFT` example in SDK is a good starting point.



**cuFFT**

GPU-accelerated library for Fast Fourier  
Transforms

<https://docs.nvidia.com/cuda/cufft/index.html#introduction>



# cuFFT Example

Sample time = 1 millisecond

Number of samples (NX) = 1000

So signal length = 1 second

Create a sinusoid with 10 cycles over the duration of the signal.

Hence frequency of sinusoid is 10 Hz.

**Question: Anything wrong with this code?**

```
#include <cuFFT.h>
#include <stdio.h>
#include <math.h>

#define NX 1000
#define BATCH 1
#define PI 3.14159265358979323846

#define SAMPLE_TIME 0.001 // Sample time in seconds
#define TOTAL_TIME (NX * SAMPLE_TIME) // Total time of the signal

int main()
{
    cufftHandle plan;
    cufftComplex *data;
    cudaMallocManaged(&data, NX * BATCH * sizeof(cufftComplex));

    // Initialize the input data with a sine wave
    for (int i = 0; i < NX; i++) {
        float t = i * SAMPLE_TIME; // Time of the current sample
        data[i].x = sin(2 * PI * 10 * t / TOTAL_TIME); // 10 cycles over the total time
        data[i].y = 0.0f;
    }

    // Create a 1D FFT plan
    cufftPlan1d(&plan, NX, CUFFT_C2C, BATCH);

    // Use the CUFFT plan to transform the signal in place
    cufftExecC2C(plan, data, data, CUFFT_FORWARD);

    // Synchronize the device
    cudaDeviceSynchronize();

    // Print the transformed data (power and frequency)
    for (int i = 0; i < NX; i++) {
        float freq = (float)i / TOTAL_TIME; // Frequency in Hz
        float power = sqrt(data[i].x * data[i].x + data[i].y * data[i].y); // Power
        printf("Frequency: %f Hz, Power: %f\n", freq, power);
    }

    // Destroy the cuFFT plan
    cufftDestroy(plan);

    // Free device memory
    cudaFree(data);

    return 0;
}
```

# cuFFT Example

Compile with:

```
$ nvcc cufft_example.cu -o test -lcufft
```

Frequency resolution = sampling rate / FFT length

So, given a signal of length 1 second, having 10 cycles in it, we expect a 10Hz response.

Exactly what we see.

```
Frequency: 1.000000 Hz, Power: 0.000004  
Frequency: 2.000000 Hz, Power: 0.000006  
Frequency: 3.000000 Hz, Power: 0.000001  
Frequency: 4.000000 Hz, Power: 0.000002  
Frequency: 5.000000 Hz, Power: 0.000001  
Frequency: 6.000000 Hz, Power: 0.000003  
Frequency: 7.000000 Hz, Power: 0.000006  
Frequency: 8.000000 Hz, Power: 0.000004  
Frequency: 9.000000 Hz, Power: 0.000004  
Frequency: 10.000000 Hz, Power: 500.000000  
Frequency: 11.000000 Hz, Power: 0.000004  
Frequency: 12.000000 Hz, Power: 0.000005  
Frequency: 13.000000 Hz, Power: 0.000013  
Frequency: 14.000000 Hz, Power: 0.000005  
Frequency: 15.000000 Hz, Power: 0.000010  
Frequency: 16.000000 Hz, Power: 0.000002  
Frequency: 17.000000 Hz, Power: 0.000001  
Frequency: 18.000000 Hz, Power: 0.000006  
Frequency: 19.000000 Hz, Power: 0.000003
```

# cuSPARSE Library

cuSPARSE is a GPU accelerated library that provides various routines to work with sparse matrices.

- Includes **sparse matrix-vector** and **matrix-matrix products**.
- Can be used for iterative solution (but see cuSOLVER for an easy life).
- Also has solution of sparse triangular system
- Note: batched tridiagonal solver is in cuBLAS not cuSPARSE



**cuSPARSE**

GPU-accelerated BLAS for sparse  
matrices

# cuRAND Library

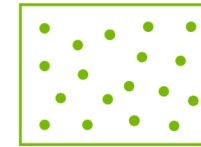
The cuRAND library is a GPU accelerated library for random number generation.

It has many different algorithms for pseudorandom and quasi-random number generation.

Pseudo: XORWOW, mrg32k3a, Mersenne Twister and Philox 4x32\_10  
Quasi: SOBOL and Scrambled SOBOL

Uniform, Normal, log-Normal and Poisson outputs

This library also includes device level routines for RNG within user kernels.



**cuRAND**

GPU-accelerated random number  
generation (RNG)

# cuRAND Example

Here is an example of using cuRAND.

This code generates a normal distribution with a mean of zero, standard deviation equal to one.

It prints out the generated numbers to a file.

To compile:

```
nvcc cuRAND_example.cu -o test -lcublas
```

```
#include <stdio.h>
#include <curand.h>

#define N 1000000

int main(int argc, char *argv[]) {

    curandGenerator_t gen;
    float *data;

    /* Allocate Unified Memory - accessible from CPU or GPU */
    cudaMallocManaged(&data, N*sizeof(float));

    /* Create pseudo-random number generator */
    curandCreateGenerator(&gen, CURAND_RNG_PSEUDO_DEFAULT);

    /* Set seed */
    curandSetPseudoRandomGeneratorSeed(gen, 1234ULL);

    /* Generate N floats on device */
    curandGenerateNormal(gen, data, N, 0.0f, 1.0f);

    /* Wait for GPU to finish before accessing on host */
    cudaDeviceSynchronize();

    /* Open a file for writing */
    FILE *file = fopen("output.txt", "w");
    if (file == NULL) {
        printf("Error opening file!\n");
        return 1;
    }

    /* Write the data to the file */
    for(int i = 0; i < N; i++) {
        fprintf(file, "%1.4f\n", data[i]);
    }

    /* Close the file */
    fclose(file);

    /* Cleanup */
    curandDestroyGenerator(gen);
    cudaFree(data);

    return 0;
}
```

# cuRAND Example

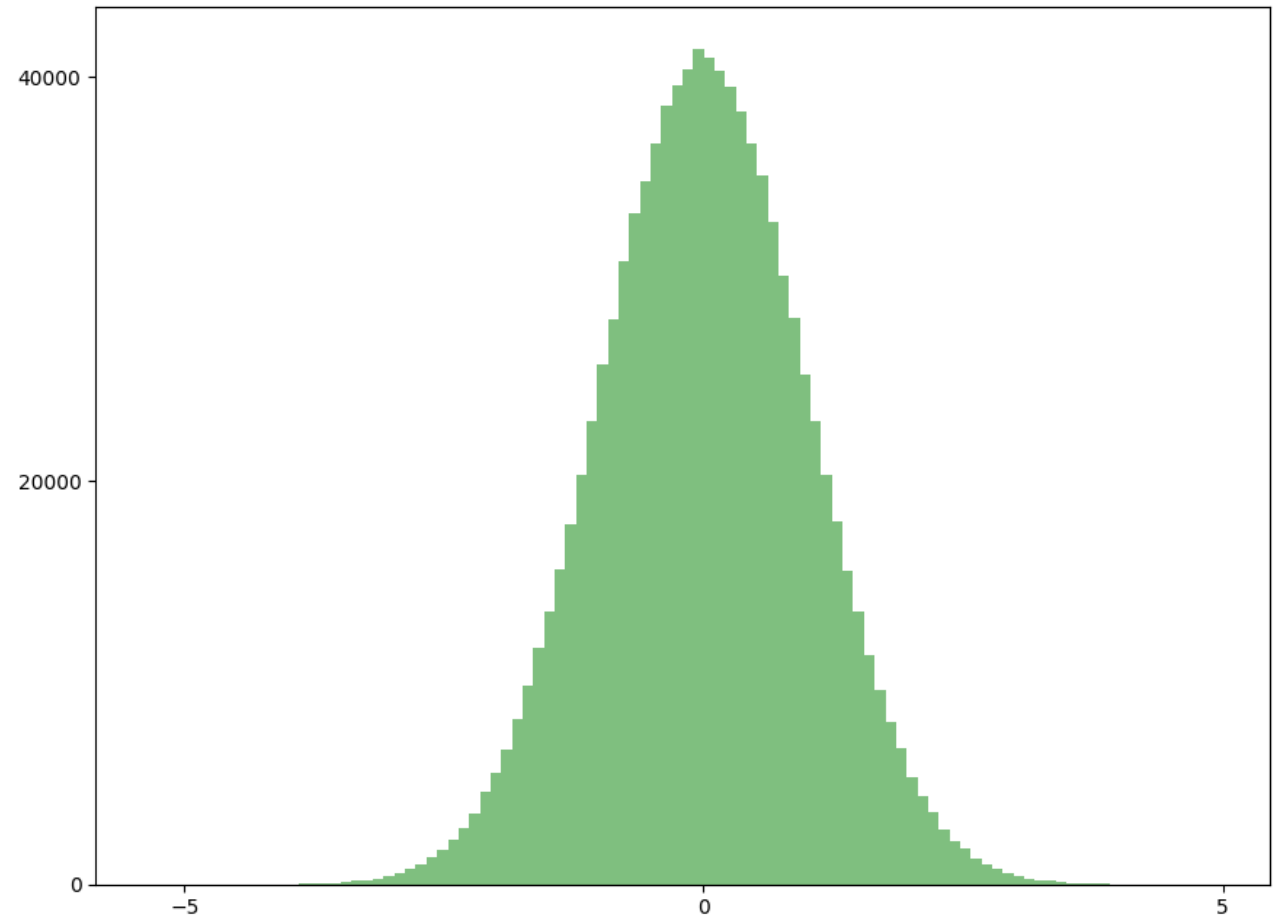
Using matplotlib:

```
import matplotlib.pyplot as plt

# Read the data from the file
with open('output.txt', 'r') as f:
    data = [float(line) for line in f]

# Create a histogram
plt.hist(data, bins=50, alpha=0.5, color='g')

# Show the plot
plt.show()
```



# cuSOLVER

cuSOLVER brings together cuSPARSE and cuBLAS.

Has solvers for dense and sparse systems.

Key LAPACK dense solvers, 3 – 6x faster than MKL.

Sparse direct solvers, 2–14x faster than CPU equivalents.



**cuSOLVER**

GPU-accelerated dense and sparse direct  
solvers

# Other notable libraries

CUB (CUDA Unbound): <https://nvidia.github.io/cccl/cub/>

- Provides a collection of basic building blocks at three levels: device, thread block, warp.
- Functions include sort, scan and reduction.
- Thrust uses CUB for CUDA versions of key algorithms.
- API Reference: <https://docs.nvidia.com/cuda/cub/index.html>

AmgX (originally named NVAMG): <http://developer.nvidia.com/amgx>

- Library for algebraic multigrid.
- Well suited for implicit unstructured methods.



## AmgX

GPU-accelerated linear solvers for  
simulations and implicit unstructured  
methods



# Other notable libraries

## cuDNN

- Library for Deep Neural Networks
- Some parts developed by Jeremy Appleyard (NVIDIA) when working in Oxford

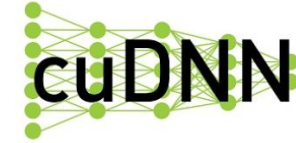
## nvGraph

- Page Rank, Single Source Shortest Path, Single Source Widest Path
- <https://developer.nvidia.com/nvgraph>

## NPP (NVIDIA Performance Primitives)

- Library for imaging and video processing
- Includes functions for filtering, JPEG decoding, etc.

## CUDA Video Decoder API...



GPU-accelerated library of primitives for deep neural networks



nvGRAPH

GPU-accelerated library for graph analytics



NVIDIA Performance Primitives

GPU-accelerated library for image and signal processing

# NCCL

The NVIDIA Collective Communication Library (NCCL) is a low level library that allows GPUs to communicate within a node and across nodes. It provides functionality such as:

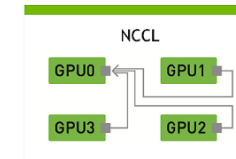
- all-gather,
- all-reduce,
- broadcast,
- point-to-point.

Optimised for PCIe and NVLink (within a node) and over NVIDIA Mellanox Network between nodes.

For ML people – use `torch.distributed`

<https://developer.nvidia.com/nccl>

<https://pytorch.org/docs/stable/distributed.html>



NCCL

Collective Communications Library for scaling apps  
across multiple GPUs and nodes

# MAGMA



MAGMA (Matrix Algebra on GPU and Multicore Architectures) has been available for a few years (See nice SC17 handout: <http://www.icl.utk.edu/files/print/2017/magma-sc17.pdf> )

- LAPACK for GPUs – higher level numerical linear algebra, layered on top of cuBLAS.
- It supports interfaces to current LA packages so porting from a previous LAPACK code is easy.
- MAGMA allows applications to exploit heterogeneous systems consisting of multicore CPUs and multi-GPUs.

<https://icl.utk.edu/magma/>  
<https://developer.nvidia.com/magma>

# ArrayFire

Originally a commercial software (from Acclereyes), but is now open source.


- Supports both CUDA, OpenCL and OneAPI execution.
- C, C++ and Python interfaces.
- Supports NVIDIA and AMD GPUs/APUs, Intel processors and mobile devices from ARM, Qualcomm...
- Wide range of functionality including linear algebra, image and signal processing, random number generation, sorting...
- Actively developed.

<https://arrayfire.com/>  
<https://github.com/arrayfire/arrayfire>




Easy-to-use API, like these examples

```
python
import arrayfire as af
af.set_backend('cuda')      # choose cuda, opencl, or cpu
A = af.randu(2**15, 2**15)  # create GPU data
A2 = af.matmul(A, A)        # fast function calls
B = af.fft2(A2)             # ...
```

 [Open in Colab \(Python\)](#)

```
c++
#include <arrayfire.h>
auto A = af::randu(2<<15, 2<<15); // create GPU data
auto A2 = af::matmul(A, A);        // fast function calls
auto B = af::fft2(A2);             // ...
```

 [Open in Codespace \(C++\)](#)

# Thrust

Thrust is a high-level C++ template library with an interface based on the C++ Standard Template Library (STL).

Thrust has a very different philosophy to other libraries - *users write standard C++ code (no CUDA) but get the benefits of GPU* acceleration.

Thrust relies on C++ object-oriented programming – certain objects exist on the GPU, and operations involving them are implicitly performed on the GPU.

It has lots of built-in functions for operations like sort and scan.

It also simplifies memory management and data movement.

<https://thrust.github.io/>



Thrust

GPU-accelerated library of parallel algorithms and data structures

# Kokkos

**Kokkos is another high-level C++ template library, similar to Thrust.**

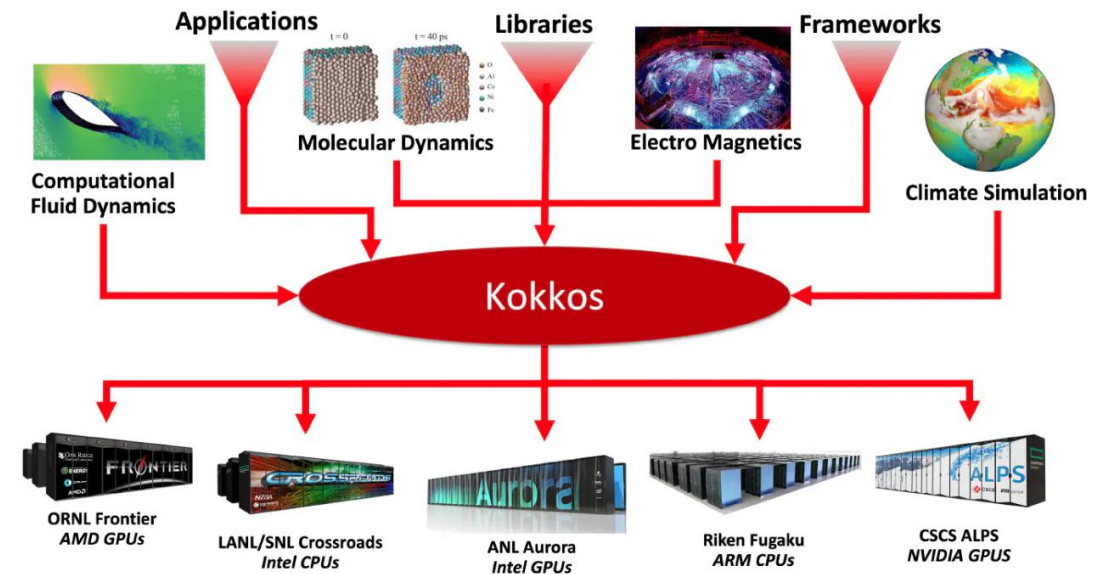
It has been developed in the US DoE Labs, so there is considerable investment in both capabilities and on-going software maintenance.

Could be worth investigating if you are considering using Thrust in your projects.

For more information see

<https://kokkos.github.io/kokkos-core-wiki/>

<https://kokkos.org/about/>



# A final word on libraries

NVIDIA maintains webpages with links to a variety of CUDA libraries:

[www.developer.nvidia.com/gpu-accelerated-libraries](http://www.developer.nvidia.com/gpu-accelerated-libraries)

and other tools:

[www.developer.nvidia.com/tools-ecosystem](http://www.developer.nvidia.com/tools-ecosystem)



[https://en.wikipedia.org/wiki/Duke\\_Humfrey%27s\\_Library](https://en.wikipedia.org/wiki/Duke_Humfrey%27s_Library)

# A note on directive based approaches and other languages

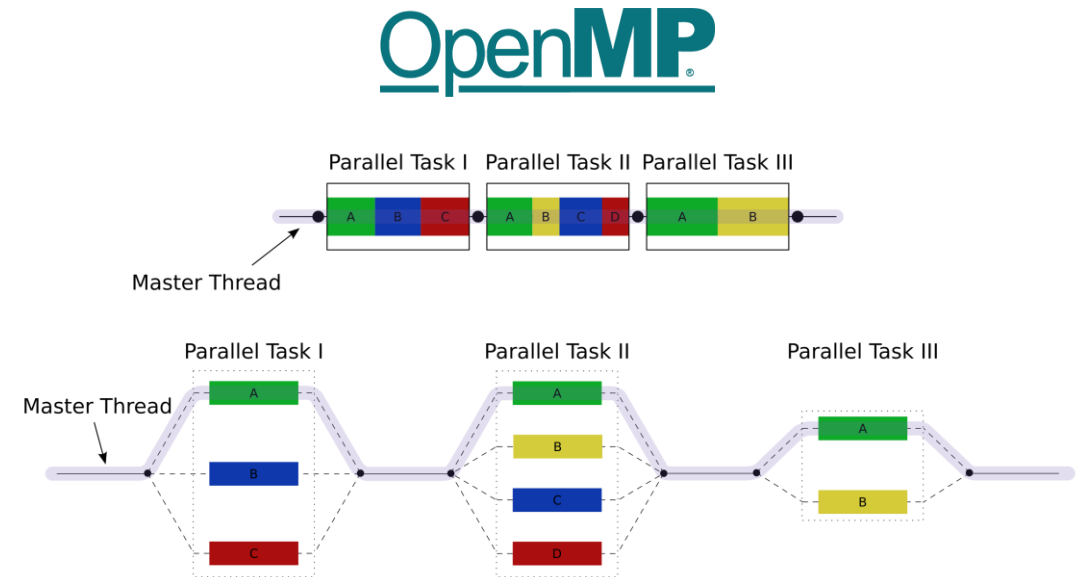


# OpenMP

**OpenMP 5.0** is a directive based approach to parallelisation.

- It uses a fork-join model.
- Can be used in C / C++ and FORTRAN codes.
- It supports both CPU and GPU hardware.

Is now becoming the industry standard for in node CPU parallelisation.



# SYCL

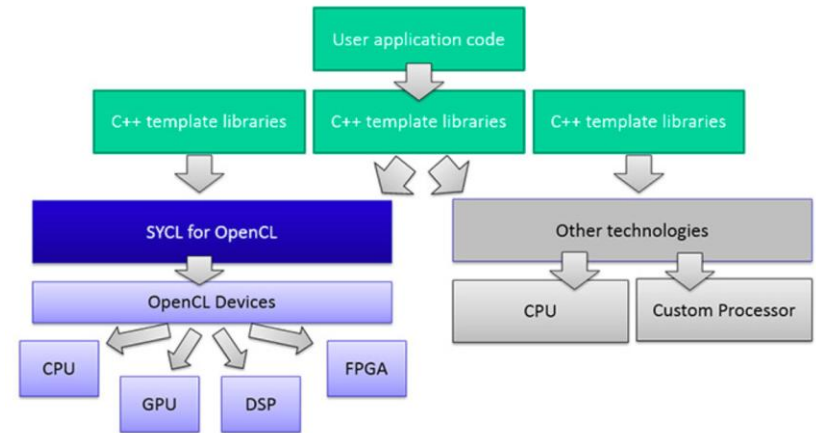
SYCL (“sickle”) - C++ Single-source Heterogeneous Programming for OpenCL.

From KHRONOS Group (responsible for OpenCL).

**Provides an abstraction layer that builds on OpenCL.**

It enables **code for heterogeneous processors** to be written in a “**single-source**” style using completely standard C++.

Supported by Intel, NVIDIA and AMD.



# Other Languages

**FORTRAN:** CUDA FORTRAN compiler with natural FORTRAN equivalent to CUDA C.

**MATLAB:** can call kernels directly, or use OOP like Thrust to define MATLAB objects which live on the GPU  
<https://uk.mathworks.com/help/parallel-computing/run-matlab-functions-on-a-gpu.html>

**Mathematica:** similar to MATLAB?

**Python:** CuPy (compatible with NumPy – acceleration for array computations), Numba and CUDA python  
<http://mathematician.de/software/pycuda>  
<https://store.continuum.io/cshop/accelerate/>  
<https://developer.nvidia.com/cuda-python>  
<https://developer.nvidia.com/how-to-cuda-python>  
<https://nvidia.github.io/cuda-python/overview.html>

# Python - CuPy

CuPy is an open-source library that provides a NumPy/SciPy-compatible array API for GPU computing within python.

You can replace:

```
import numpy as np
```

with

```
import cupy as cp
```

and run your existing code on the GPU with minimal changes.

```
import numpy as np
import cupy as cp
import time

# Define the size of the square matrices
size = 10000

## NumPy
print("Running on CPU with NumPy...")
# Create two large random matrices on the CPU
a_cpu = np.random.rand(size, size).astype(np.float32)
b_cpu = np.random.rand(size, size).astype(np.float32)

# Time the matrix multiplication on the CPU
start_cpu = time.time()
result_cpu = np.dot(a_cpu, b_cpu)
end_cpu = time.time()

print(f"NumPy (CPU) time: {end_cpu - start_cpu:.4f} seconds")

print("-" * 30)

## CuPy
print("Running on GPU with CuPy...")
# Create two large random matrices on the GPU
a_gpu = cp.random.rand(size, size).astype(cp.float32)
b_gpu = cp.random.rand(size, size).astype(cp.float32)

# Time the matrix multiplication on the GPU
start_gpu = time.time()
result_gpu = cp.dot(a_gpu, b_gpu)
cp.cuda.Stream.null.synchronize() # Wait for the GPU to finish its work
end_gpu = time.time()

print(f"CuPy (GPU) time: {end_gpu - start_gpu:.4f} seconds")
```

```
Running on CPU with NumPy...
NumPy (CPU) time: 5.7358 seconds
-----
Running on GPU with CuPy...
CuPy (GPU) time: 1.6428 seconds
```

# Python - CuPy

CuPy can be installed as follows:

```
w @ ~$ python3 -m venv venv
w @ ~$ source venv/bin/activate
(venv) w @ ~$ pip install numpy
Collecting numpy
  Downloading numpy-2.3.1-cp312-cp312-manylinux_2_28_x86_64.whl.metadata (62 kB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 62.1/62.1 kB 3.2 MB/s eta 0:00:00
  Downloading numpy-2.3.1-cp312-cp312-manylinux_2_28_x86_64.whl (16.6 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 16.6/16.6 MB 11.2 MB/s eta 0:00:00
Installing collected packages: numpy
Successfully installed numpy-2.3.1
(venv) w @ ~$ nvcc --version
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2023 NVIDIA Corporation
Built on Fri_Jan__6_16:45:21_PST_2023
Cuda compilation tools, release 12.0, V12.0.140
Build cuda_12.0.r12.0/compiler.32267302_0
(venv) w @ ~$ pip install cupy-cuda12x
Collecting cupy-cuda12x
  Downloading cupy_cuda12x-13.5.1-cp312-cp312-manylinux2014_x86_64.whl.metadata (2.4 kB)
Requirement already satisfied: numpy<2.6,>=1.22 in ./venv/lib/python3.12/site-packages (from cupy-cuda12x) (2.3.1)
Collecting fastrlock>=0.5 (from cupy-cuda12x)
  Using cached fastrlock-0.8.3-cp312-cp312-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_28_x86_64.whl.metadata (7.7 kB)
  Downloading cupy_cuda12x-13.5.1-cp312-cp312-manylinux2014_x86_64.whl (113.1 MB)
    ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 113.1/113.1 MB 9.9 MB/s eta 0:00:00
  Using cached fastrlock-0.8.3-cp312-cp312-manylinux_2_5_x86_64.manylinux1_x86_64.manylinux_2_28_x86_64.whl (53 kB)
Installing collected packages: fastrlock, cupy-cuda12x
Successfully installed cupy-cuda12x-13.5.1 fastrlock-0.8.3
```



CuPy

NumPy/SciPy-compatible Array Library for GPU-accelerated Computing with Python

<https://cupy.dev/>

<https://docs.cupy.dev/en/stable/>

# Python - Numba

Numba is a just-in-time (JIT) **compiler** for Python that translates Python functions into optimized machine code at runtime. It uses decorators to direct the compiler.

The process is as follows:

- You write a standard Python function that contains slower python code, like a for loop.
- You add a Numba decorator (e.g., `@jit`) on the line above your function definition.
- The first time your function is called, Numba's compiler acts. It compiles a highly optimized, type-specific version of your function, then all subsequent calls to that function use the fast, compiled version directly (rather than the slower Python interpreter).



Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

# Python - Numba

Numba standard decorators are:

- `@jit`: This is the main decorator, it will speed up your functions on the CPU.
- `@vectorize`: This allows you to create NumPy "universal functions" (ufuncs) out of a simple Python function.
- `@cuda.jit`: This compiles your Python function into a CUDA kernel to be run on an NVIDIA GPU.



Numba makes Python code fast

Numba is an open source JIT compiler that translates a subset of Python and NumPy code into fast machine code.

```
import numpy as np
import numba
from numba import jit, cuda
import time

@jit
def vector_add_cpu(a, b, c):
    for i in range(a.shape[0]):
        c[i] = a[i] + b[i]

@cuda.jit
def vector_add_gpu_kernel(a, b, c):
    idx = cuda.grid(1) # Provides a unique threadId
    c[idx] = a[idx] + b[idx]
```

# Python - PyCUDA

PyCUDA is far closer to the familiar CUDA that we have been teaching you this week.

It works by taking C/CUDA code as a python string and compiling it on-the-fly and returning a python object.

- SourceModule: This is a class provided by PyCUDA. It takes a string containing source code for a CUDA kernel.
- Compilation: When you create a SourceModule object, PyCUDA takes that string and passes it to nvcc which compiles to machine code for execution.
- The variable mod holds the result of this compilation, a compiled "module." You can think of this module as a container for all the functions you defined in your C/C++ string.

```
import numpy as np
import pycuda.autoinit
import pycuda.driver as cuda
from pycuda.compiler import SourceModule

# C/CUDA code as a python string
mod = SourceModule("""
__global__ void add_vectors(float *dest, float *a, float *b, int n)
{
    // Calculate the unique global index for this thread
    const int i = blockIdx.x * blockDim.x + threadIdx.x;

    // Ensure the thread index is within the bounds of the array
    if (i < n)
    {
        // Perform the vector addition for the element at this index
        dest[i] = a[i] + b[i];
    }
}
""")

# Define the size of the vectors
n = 10000000

# --- Host (CPU) Code ---
print(f"Performing vector addition for {n} elements.")

# Create random input vectors and an empty destination vector on the CPU
a_cpu = np.random.randn(n).astype(np.float32)
b_cpu = np.random.randn(n).astype(np.float32)
dest_cpu = np.empty_like(a_cpu)

# Get the kernel function from the compiled module
add_vectors_kernel = mod.get_function("add_vectors")

# Define the grid and block dimensions
block_size = 256
grid_size = (n + block_size - 1) // block_size

print(f"Launching kernel with grid size {grid_size} and block size {block_size}...")

# Call the kernel on the GPU, note kernel call returns after completion
# because PyCUDA handles memory allocation and data transfer automatically.
add_vectors_kernel(
    cuda.Out(dest_cpu), cuda.In(a_cpu), cuda.In(b_cpu), np.int32(n),
    block=(block_size, 1, 1),
    grid=(grid_size, 1)
)

print("Verifying the result...")
if np.allclose(dest_cpu, a_cpu + b_cpu):
    print("Success! The GPU computation is correct.")
else:
    print("Error! The GPU computation is incorrect.")
```



# Other useful things...



## CUDA Toolkit

Provides a comprehensive environment for C/C++ developers building GPU-accelerated applications.



## CUDA FORTRAN

Enjoy GPU acceleration directly from your Fortran program using CUDA Fortran from The Portland Group.



## OpenCL™

OpenCL is a low-level API for GPU computing that can run on CUDA-powered GPUs.

## OpenACC

### OpenACC

Directives for parallel computing, is a new open parallel programming standard designed to enable all scientific and technical programmers.



## PyCUDA

Gives you access to CUDA functionality from your Python code.



## Alea GPU

This is a novel approach to develop GPU applications on .NET, combining the CUDA with Microsoft's F#.

<https://developer.nvidia.com/tools-ecosystem>  
<https://developer.nvidia.com/language-solutions>  
<https://developer.nvidia.com/hpc-sdk>  
<https://developer.nvidia.com/hpc-compilers>

Which library should I use for my problem?

# The seven dwarfs

Phil Colella a senior researcher at Lawrence Berkeley National Laboratory, talked about “7 dwarfs” of numerical computation in 2004.

Expanded to 13 by a group of UC Berkeley professors in a 2006 report: “A View from Berkeley”

[www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf](http://www.eecs.berkeley.edu/Pubs/TechRpts/2006/EECS-2006-183.pdf)



# The seven dwarfs

These 13 dwarfs define key algorithmic kernels in many scientific computing applications.

They have been **very helpful to focus attention on HPC challenges and development of libraries and problem-solving environments/frameworks.**



# The seven dwarfs

1. Dense linear algebra
2. Sparse linear algebra
3. Spectral methods
4. N-body methods
5. Structured grids
6. Unstructured grids
7. Monte Carlo



# 1. Dense Linear Algebra

Many tools available, some from NVIDIA, some third party:

- cuBLAS
- cuSOLVER
- MAGMA
- ArrayFire

CUTLASS, an NVIDIA tool for Fast Linear Algebra in CUDA C++ might also be worth a look if you can't use the above libraries for any reason.

<https://devblogs.nvidia.com/cutlass-linear-algebra-cuda/>



**cuBLAS**

GPU-accelerated basic linear algebra  
(BLAS) library



**cuSOLVER**

GPU-accelerated dense and sparse direct  
solvers



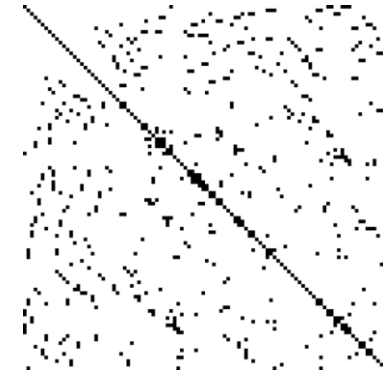
## 2. Sparse Linear Algebra

### Iterative solvers

- Some available in Petsc (Portable, Extensible Toolkit for Scientific Computation, for solving PDEs) - <https://petsc.org/release/overview/nutshell/>
- Others can be implemented using sparse matrix-vector multiplication from cuSPARSE (is also now in PETSc).
- NVIDIA has AmgX, an algebraic multigrid library.

### Direct solvers

- NVIDIA's cuSOLVER.
- SuperLU project (Gaussian elimination with partial pivoting) <https://portal.nersc.gov/project/sparse/superlu/>
- STRUMPACK (ask Mike) <https://portal.nersc.gov/project/sparse/strumpack//>



**cuSOLVER**

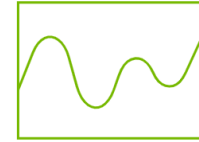
GPU-accelerated dense and sparse direct solvers

# 3. Spectral methods

cuFFT /cuFFTW

Library provided / maintained by NVIDIA

For those interested in FFTs on GPUs – ask me...



**cuFFT**

GPU-accelerated library for Fast Fourier  
Transforms



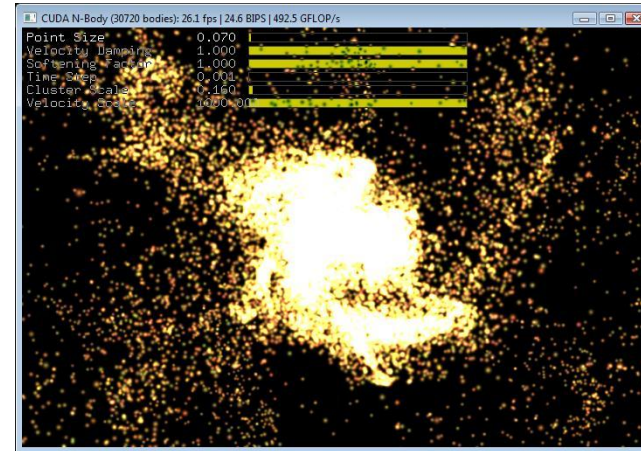
# 4. N-Body methods

OpenMM:

- <http://openmm.org/>  
open source package to support molecular modelling, developed at Stanford.

Fast multipole methods:

- ExaFMM by Yokota and Barba:  
<http://www.bu.edu/exafmm/>
- FMM2D by Holm, Engblom, Goude, Holmgren: <http://user.it.uu.se/~stefane/freeware>  
<https://lorenabarba.com/figshare/exafmm-10-years-7-re-writes-the-tortuous-progress-of-computational-research/>
- Software by Takahashi, Cecka, Fong, Darve:  
<http://onlinelibrary.wiley.com/doi/10.1002/nme.3240/pdf>



<https://docs.nvidia.com/cuda/cuda-samples/index.html#cuda-n-body-simulation>

[https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86\\_website/projects/nbody/doc/nbody\\_gems3\\_ch31.pdf](https://developer.download.nvidia.com/compute/cuda/1.1-Beta/x86_website/projects/nbody/doc/nbody_gems3_ch31.pdf)

# 5. Structured grids

Lots of people have developed one-off applications.

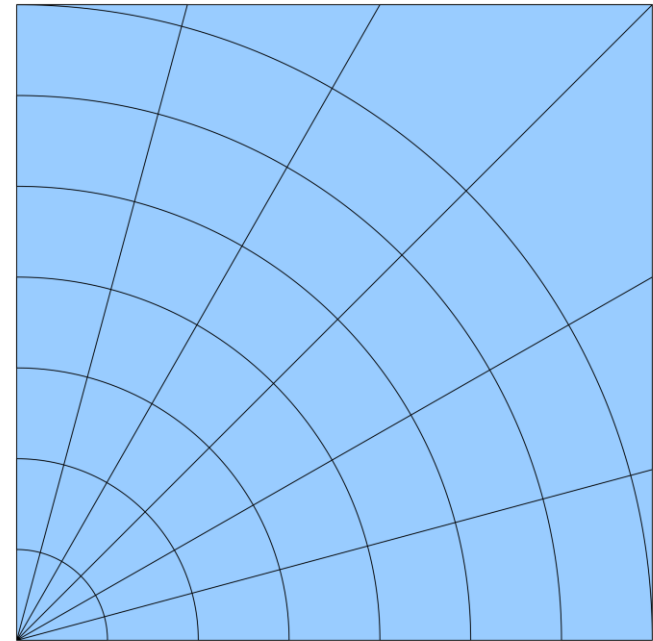
No great need for a library for single block codes (though possible improvements from “tiling”?).

Multi-block codes could benefit from a general-purpose library, mainly for MPI communication.

Oxford OPS project has developed a high-level open-source framework for multi-block codes, using GPUs for code execution and MPI for distributed-memory message-passing.

All implementation details are hidden from “users”, so they don’t have to know about GPU/MPI programming.

For those interested – ask Mike...

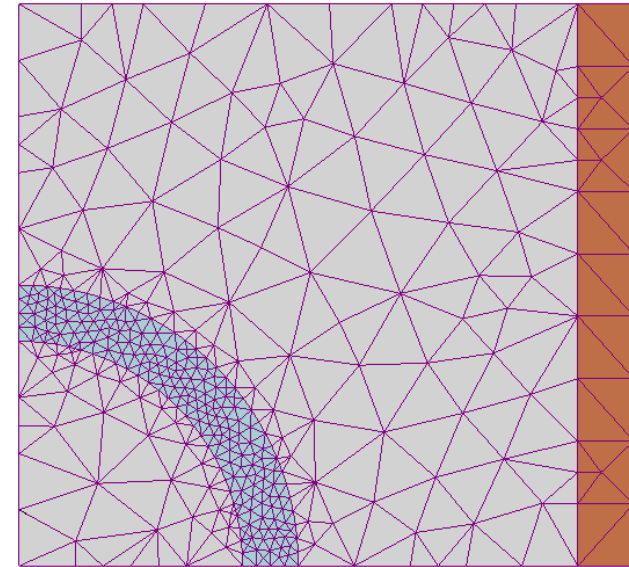


# 6. Unstructured grids

In addition to GPU implementations of specific codes there are projects to create high-level solutions which others can use for their application codes:

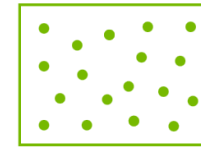
- Alonso, Darve and others (Stanford).
- Oxford / Imperial College / Warwick project developed OP2, a general-purpose open-source framework based on a previous framework built on MPI.
- If there's interest Mike could talk about OP2 and OPS in lecture 8/9.

See <https://op-dsl.github.io/> for both OPS and OP2



# 7. Monte Carlo methods

- NVIDIA cuRAND library.
- ArrayFire library.
- Some examples in CUDA SDK distribution.
- Nothing else needed except for more output distributions?



**cuRAND**

GPU-accelerated random number  
generation (RNG)



# Useful tools

# Tools - Debugging

```
compute-sanitizer -tool memcheck
```

A command line tool that detects array out-of-bounds errors, and mis-aligned device memory accesses – very useful because such errors can be tough to track down otherwise.

```
compute-sanitizer --tool racecheck
```

This checks for shared memory race conditions:

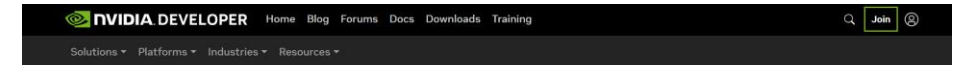
- Write-After-Write (WAW): two threads write data to the same memory location, but the order is uncertain.
- Read-After-Write (RAW) and Write-After-Read (WAR): one thread writes and another reads, but the order is uncertain.

```
compute-sanitizer --tool initcheck
```

This detects the reading of uninitialised device memory.

```
compute-sanitizer --tool synccheck
```

This detects incorrect use of `__syncthreads()` and related intrinsics.



Home / GameWorks / Tools / NVIDIA Compute Sanitizer

## NVIDIA Compute Sanitizer

### Compute Sanitizer Tools & API

Compute Sanitizer is a functional correctness checking suite. This suite contains multiple tools that can perform different type of checks. Tool features are described below.

The Compute Sanitizer API enables the creation of sanitizing and tracing tools that target CUDA applications. Examples of such tools are memory and race condition checkers. The Compute Sanitizer API is composed of three APIs: the callback API, the patching API and the memory API. It is delivered as a dynamic library on supported platforms.

#### Memcheck

The memcheck tool is a run time error detection tool for CUDA applications. The tool can precisely detect and report out of bounds and misaligned memory accesses to global, local and shared memory in CUDA applications. It can also detect and report hardware reported error information. In addition, the memcheck tool can detect and report memory leaks in the user application.

#### Racecheck

The racecheck tool is a run time shared memory data access hazard detector. The primary use of this tool is to help identify memory access race conditions in CUDA applications that use shared memory. In CUDA applications, storage declared with the `__shared__` qualifier is placed on chip shared memory. All threads in a thread block can access this per block shared memory. Shared memory goes out of scope when the thread block completes execution. As shared memory is on chip, it is frequently used for inter-thread communication and as a temporary buffer to hold data being processed. As this data is being accessed by

#### Initcheck

The initcheck tool is a run time uninitialized device global memory access detector. This tool can identify when device global memory is accessed without it being initialized via device side writes, or via CUDA memcpy and memset API calls. Currently, this tool only supports detecting accesses to device global memory.

#### Synccheck

The synccheck tool is a runtime tool that can identify whether a CUDA application is correctly using synchronization primitives, specifically `__syncthreads()` and `__syncthreads_block()` and their Cooperative Groups API counterparts.

<https://developer.nvidia.com/compute-sanitizer>

```
user@machine:~$ compute-sanitizer --tool memcheck ./cuRAND_test
===== COMPUTE-SANITIZER
===== ERROR SUMMARY: 0 errors
user@machine:~$ compute-sanitizer --tool racecheck ./cuRAND_test
===== COMPUTE-SANITIZER
===== RACECHECK SUMMARY: 0 hazards displayed (0 errors, 0 warnings)
user@machine:~$ compute-sanitizer --tool initcheck ./cuRAND_test
===== COMPUTE-SANITIZER
===== ERROR SUMMARY: 0 errors
```

# Tools – CUDA-GDB

For those familiar with the GNU debugger – GDB, this is an extension of GDB that allows users to debug both GPU and CPU code.

All existing GDB debugging features are included for debugging host code and then further functionality allows the user to debug device code.

Supports C/C++ and Fortran (that includes CUDA code).



## CUDA-GDB

Delivers a seamless debugging experience that allows you to debug both the CPU and GPU portions of your application simultaneously. Use CUDA-GDB on Linux or MacOS, from the command line, DDD or EMACS.

# Tools - IDEs

Integrated Development Environments (IDE):

Nsight Systems – Unified IDE for Windows/Linux/Mac/Jetson:

<https://developer.nvidia.com/nsight-systems>

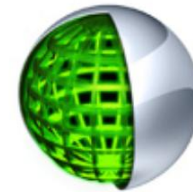
Nsight Visual Studio edition – NVIDIA plugin for Microsoft Visual Studio

<http://developer.nvidia.com/nvidia-nsight-visual-studio-edition>

Nsight Eclipse plugins

<https://docs.nvidia.com/cuda/nsight-eclipse-plugins-guide/index.html>

these come with editor, debugger, profiler integration



**NVIDIA® Nsight™**

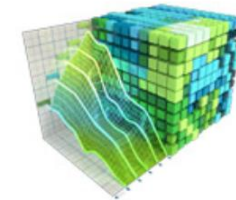
The ultimate development platform for heterogeneous computing. Work with powerful debugging and profiling tools, optimize the performance of your CPU and GPU code. Find out about the Eclipse Edition and the graphics debugging enabled Visual Studio Edition.



# Tools - Profiling

NVIDIA Profiler `ncu` or `ncu-ui` for a graphical interface.

- This is a standalone piece of software for Linux and Windows systems.
- It uses hardware counters to collect a lot of useful information.
- Lots of things can be measured, but the limited number of counters means that, for some larger applications, it runs the application multiple to gather necessary information.
- The `ncu` CLI can be useful if you want to profile on a machine that you don't have a graphical interface to.
- Do `ncu --help` for more info on different options.



## NVIDIA Visual Profiler

This is a cross-platform performance profiling tool that delivers developers vital feedback for optimizing CUDA C/C++ applications. First introduced in 2008, Visual Profiler supports all CUDA capable NVIDIA GPUs shipped since 2006 on Linux, Mac OS X, and Windows.

<https://docs.nvidia.com/nsight-compute/NsightCompute/index.html>

<https://docs.nvidia.com/cuda/profiler-users-guide/index.html>

# What have we learnt?

In this lecture we've looked at the **wide software ecosystem that now surrounds GPU computing** and how that can be used to make your life as a programmer easier.

We've looked at **directives-based approaches** and how these are useful.

Finally, we've looked at **tools** that allow us to develop CUDA code in an easy and maintainable way.

