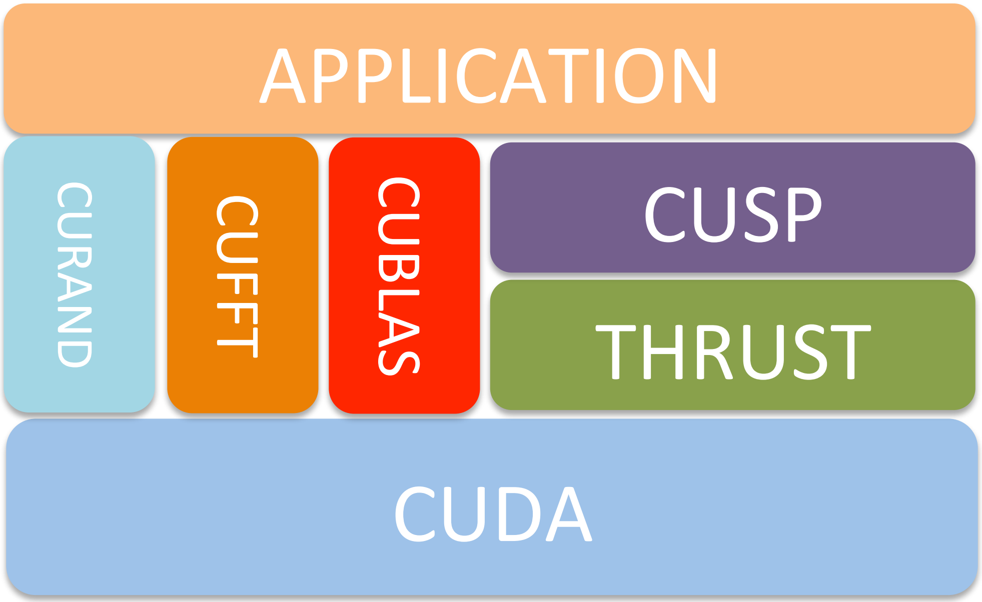
CUDA Libraries

Ezio Bartocci
Vienna University of Technology

Overview of the CUDA libraries



CURAND Library

- This library provides the facilities that focus on the simple and efficient generation of high-quality *pseudorandom* and *quasirandom* numbers.

- A *pseudorandom* sequence of numbers satisfies most of the statistical properties of a truly random sequence but is generated by a deterministic algorithm.

- A *quasirandom* sequence of n-dimensional points is generated by a deterministic algorithm designed to fill n-dimensional space evenly.

CURAND

- Host (CPU) side library:
  - User should include the header file, /include/curand.h to get function declarations and then link against the library.
  - Random numbers can be generated on the device or on the host CPU. For the device generation, calls to the library happen on the host, but the random numbers are stored in global memory on the device.
  - Users can then call their own kernels to use random numbers, or they can copy the random numbers back to the host for further processing
- Device (GPU) header file:
  - User should include the header file, /include/curand_kernel.h. This file define device functions defined in the header file, and the users can access them directly in the kernels.

Host API Overview

- Random numbers are produced by generators. A generator in CURAND encapsulates all the internal state necessary to produce a sequence of *pseudorandom* or *quasirandom* numbers.
- The sequence of necessary operation are:
  1. Create a new generator (calling **curandCreateGenerator()**)
  2. Set the generator options (use **curandSetPseudoRandomGeneratorSeed()**)
  3. Allocating memory on the device (use **cudaMalloc()**)
  4. Generate random numbers (i.e. using **curandGenerate()**)
  5. Use the results
  6. If necessary generate again random numbers by repeating step 4.
  7. Clean up with **curandDestroyGenerator()**.

Generators Types

- Random numbers generators are created by passing a type to **curandCreateGenerator()**. There are two types of random generators in CURAND:

1. Type CURAND_RNG_XORWOW is a *pseudorandom* number generator implemented the XORWOW algorithm.

2. Type CURAND_RNG_SOBOL32 is a *quasirandom* number generator type.

Generator Options

- **Seed**
  - The seed parameter is a 64-bit integer that initializes the starting state of a pseudorandom number generator. The same seed always produces the same sequence of results.

- **Offset**
  - The offset parameter is used to skip ahead in the sequence. If offset = 100, the first random number generated will be the 100th in the sequence. This allows multiple runs of the same program continue generating results from the same sequence without overlap.

CUFFT Library

Fourier series:

$$T_0 = \frac{1}{\omega_0}$$

$$x[t] = \frac{a_0}{2} + \sum_{k=0}^{+\infty} a_k \cos(2\pi\omega_k t) + b_k \sin(2\pi\omega_k t)$$

with $\omega_k = k\omega_0$

Euler's formula: $e^{ix} = \cos(x) + i\sin(x)$

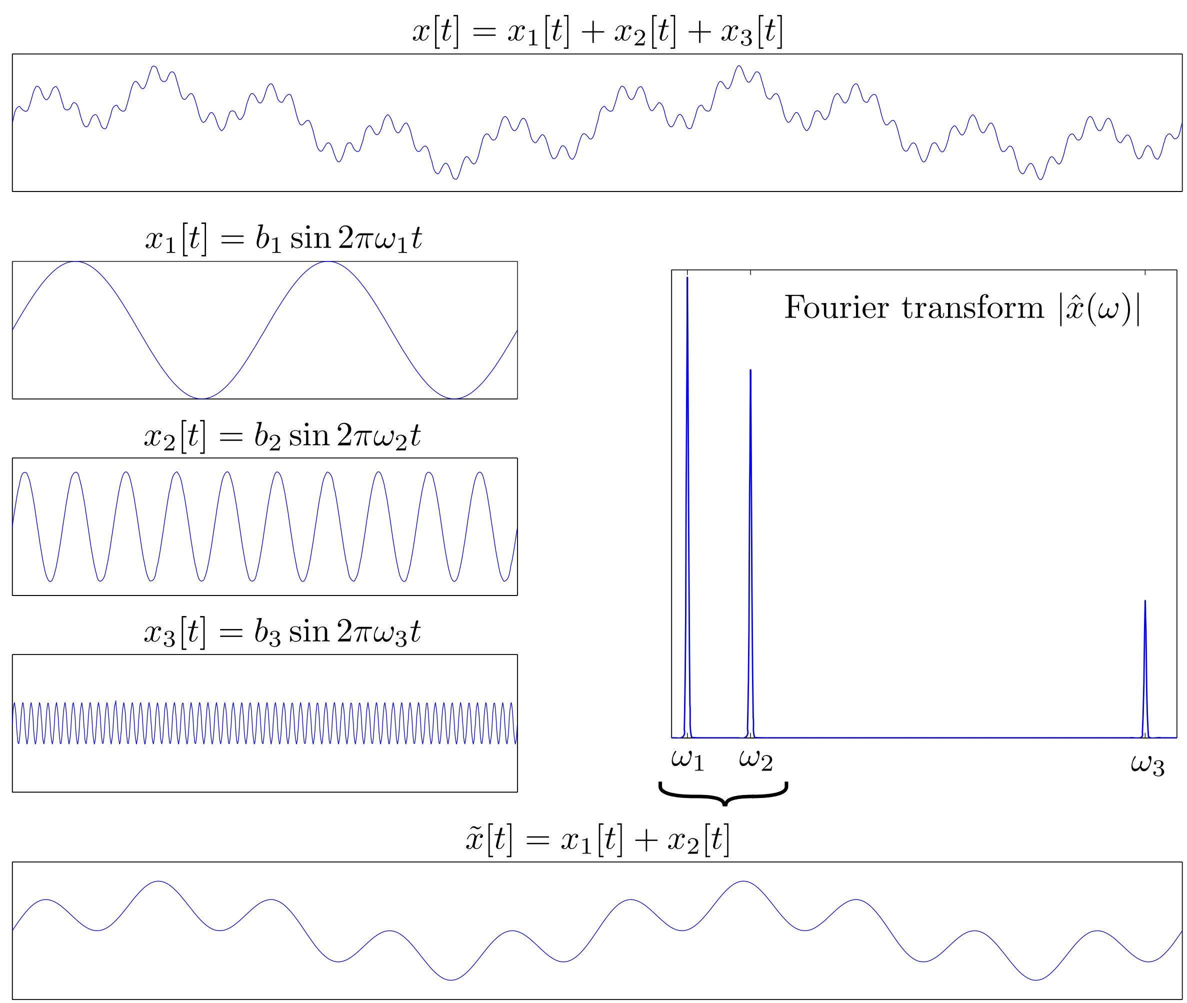
$$x[t] = \sum_{k=-\infty}^{+\infty} c_k e^{2i\pi k\omega_0 t}$$

Continuos Fourier Transform:

$$c_\omega = \hat{x}(\omega) = \int_{-\infty}^{+\infty} x[t] e^{-2i\pi\omega t}\, dt$$

Inverse Continuos Fourier Transform:

$$x[t] = \int_{-\infty}^{+\infty} c_\omega e^{2i\pi\omega t}\, dt$$



CUFFT Library

Fourier series:

$$T_0 = \frac{1}{\omega_0}$$

$$x[t] = \frac{a_0}{2} + \sum_{k=0}^{+\infty} a_k \cos(2\pi\omega_k t) + b_k \sin(2\pi\omega_k t)$$

with $\omega_k = k\omega_0$

Euler's formula: $e^{ix} = \cos(x) + i\sin(x)$

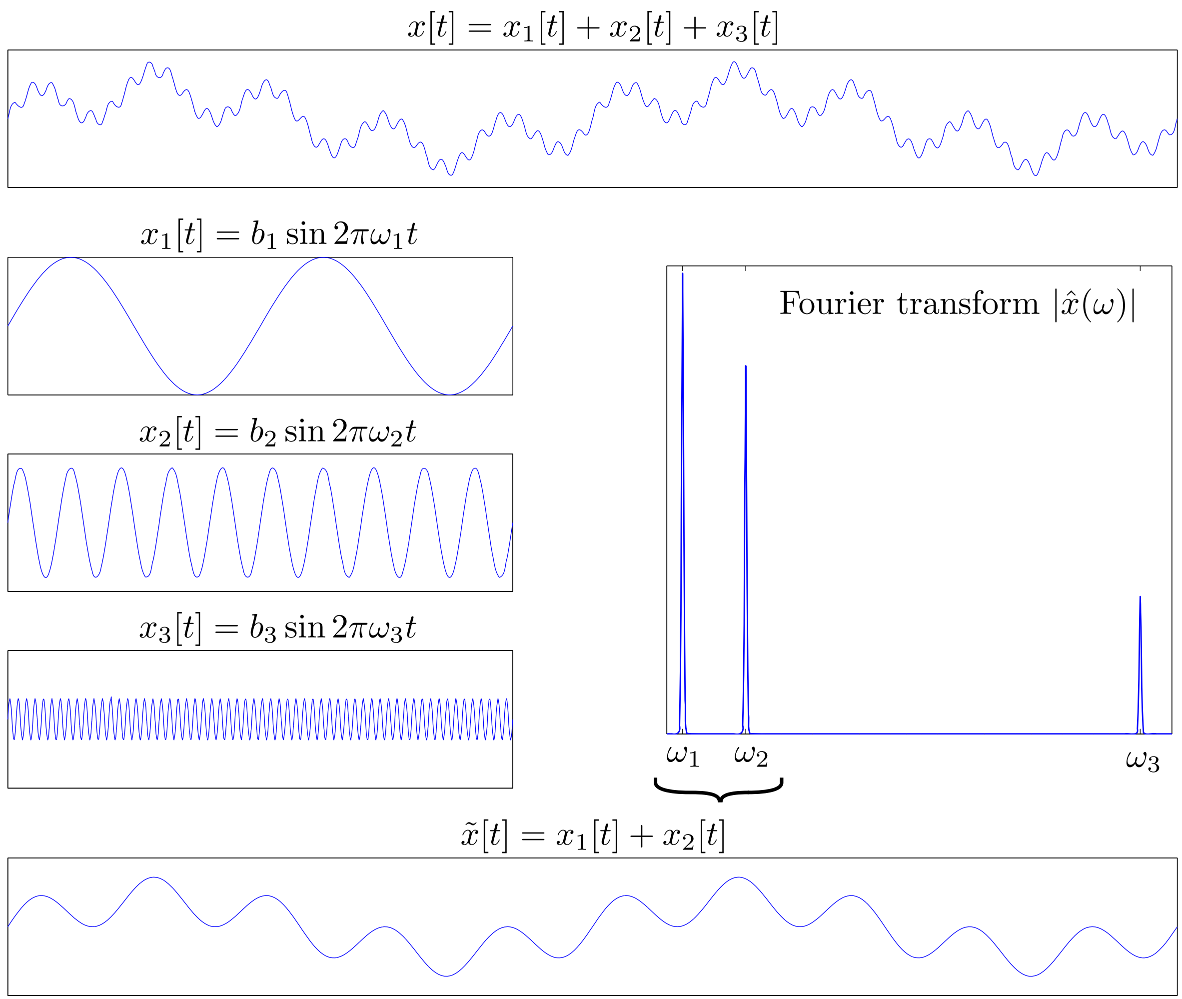
$$x[t] = \sum_{k=-\infty}^{+\infty} c_k e^{2i\pi k\omega_0 t}$$

Continuos Fourier Transform:

$$c_\omega = \hat{x}(\omega) = \int_{-\infty}^{+\infty} x[t] e^{-2i\pi\omega t}\, dt$$

Inverse Continuos Fourier Transform:

$$x[t] = \int_{-\infty}^{+\infty} c_\omega e^{2i\pi\omega t}\, dt$$



Discrete Fourier Transform (DFT)

- **DFT**

$$\widehat{u}_k = \sum_{j=0}^{N-1} u_j e^{-\frac{2\pi i}{N}kj} \qquad k=0,1,\ldots,N-1$$

- **Inverse DFT**

$$u_j = \sum_{k=0}^{N-1} u_k e^{\frac{2\pi i}{N}kj} \qquad j=0,1,\ldots,N-1$$

Discrete Fourier Transform (DFT)

- Cooley-Tukey introduced a Fast method to calculate the DFT
- Computations can drop from $O(N^2)$ to $O(N \log N)$
  - If $N = 10^5$ the computation goes from 10^{10} to $5 * 10^5$
- Application:
  - Signal processing
  - Convolution, filters
  - Calculating derivatives Partial Differential Equations

CUFFT

- CUFFT: CUDA library for FFTs on the GPU
- Supported by NVIDIA
- Features:
  - 1D,2D, 3D transforms for complex and real data
  - Batch execution for multiple transforms
  - Up to 128 million elements (limited by memory)
  - Double precision supported
  - Data streamed execution

CUFFT - Types

- cufftHandle
  - Handle type to store CUFFT plans
- cufftResult
  - Return values, like CUFFT_SUCCESS, CUFFT_INVALID_PLAN, CUFFT_ALLOC_FAILED, CUFFT_INVALID_TYPE, etc.
- cufftReal
- cufftDoubleReal
- cufftComplex
- cufftDoubleComplex

CUFFT – Transform types

- R2C: real to complex
- C2R: Complex to real
- C2C: complex to complex
- D2Z: double to double complex
- Z2D: double complex to double
- Z2Z: double complex to double complex

CUFFT – Plans

- cufftPlan1d
- cufftPlan2d
- cufftPlan3d
- cufftPlanMany

CUFFT – Functions

- cufftDestroy
  - Free GPU resources
- cufftExecC2C, R2C, C2R, Z2Z, D2Z, Z2D
  - Performs the specified FFT

CUFFT – Performance considerations

- Several algorithms for different size
- Performance recommendations
  - Restrict size to be a multiple of 2,3, 5, 7
  - Restrict the power-of-two factorization term of the X-dimension to be at least a multiple of 16 for single and 8 for double
  - Restrict the power-of-two factorization term of the X-dimension to be a multiple of 256 for single and 64 for double
- CUFFT is good for larger, power-of-two sized FFT
- CUFFT is not good for small sized FFTs
  - CPU can store all data in cache
  - GPU data transfers take too long