GPU Computing with CUDA
Lecture 8 - CUDA Libraries - CUFFT, PyCUDA

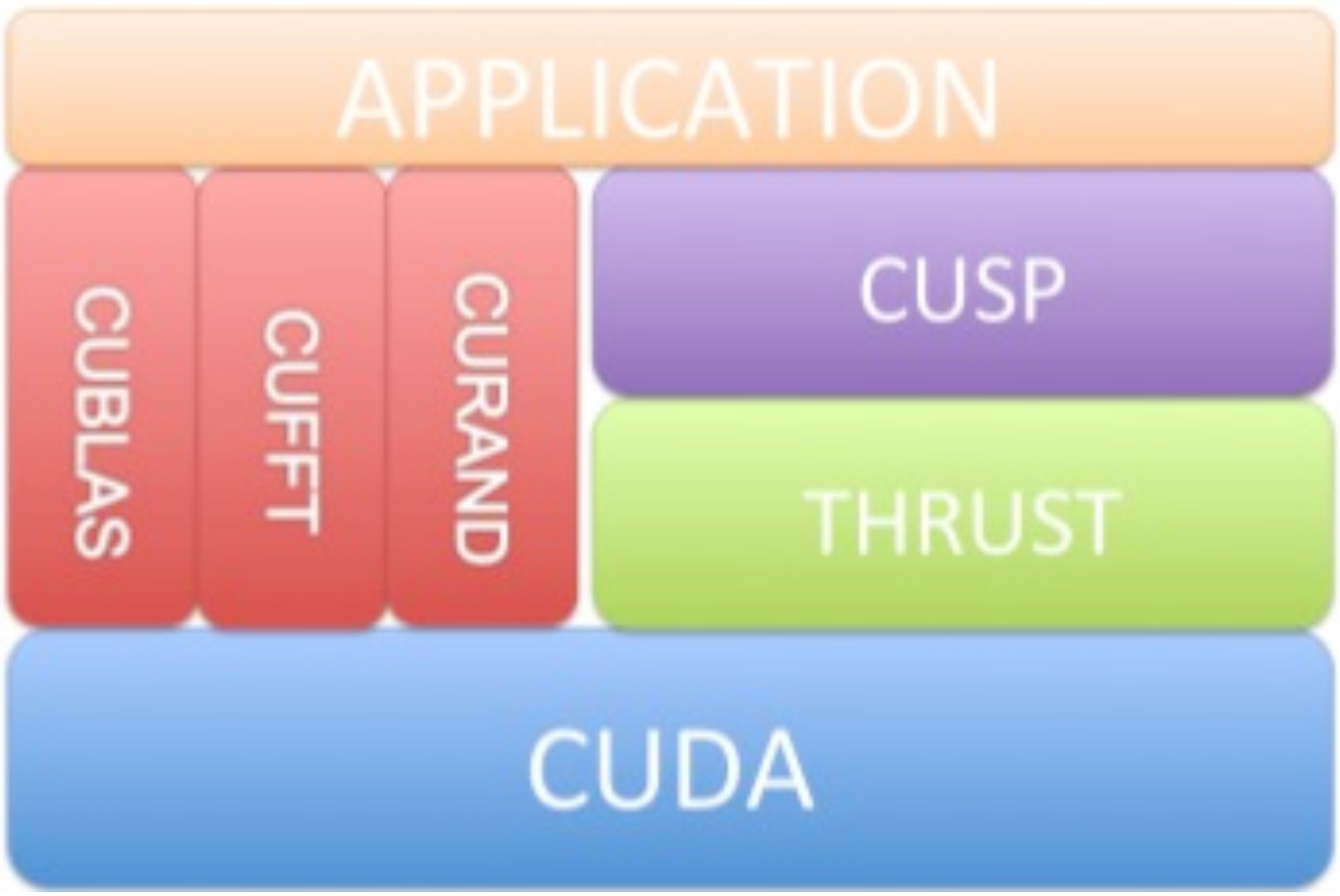
Christopher Cooper
Boston University

August, 2011
UTFSM, Valparaíso, Chile

Outline of lecture

- Overview:
  - Discrete Fourier Transform (DFT)
  - Fast Fourier Transform (FFT)
    - Algorithm
    - Motivation, examples
- CUFFT: A CUDA based FFT library
- PyCUDA: GPU computing using scripting languages

CUDA Libraries

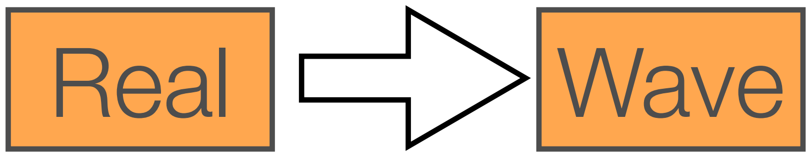


Bell, Dalton, Olson. Towards AMG on GPU

Fourier Transform

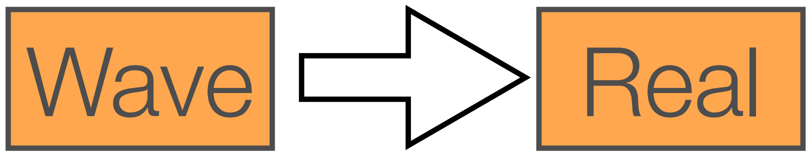
- Fourier Transform

$$\hat{u}(k) = \int_{-\infty}^{\infty} e^{-ikx} u(x) dx$$



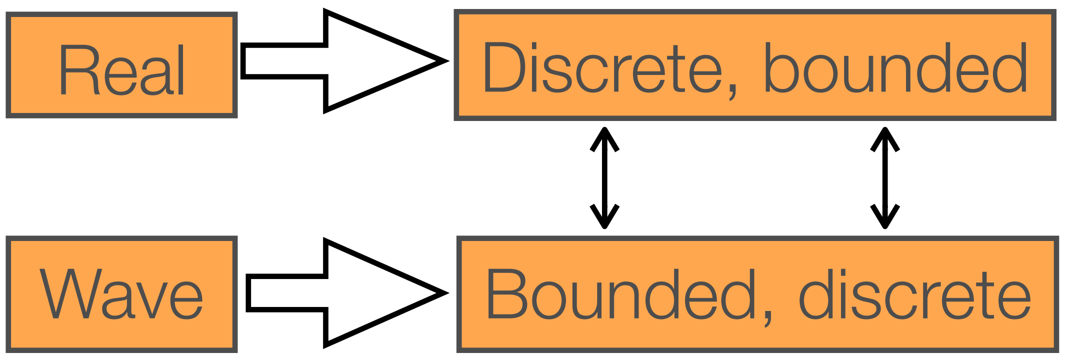
- Inverse Fourier Transform

$$u(x) = \int_{-\infty}^{\infty} e^{ikx} \hat{u}(k) dk$$

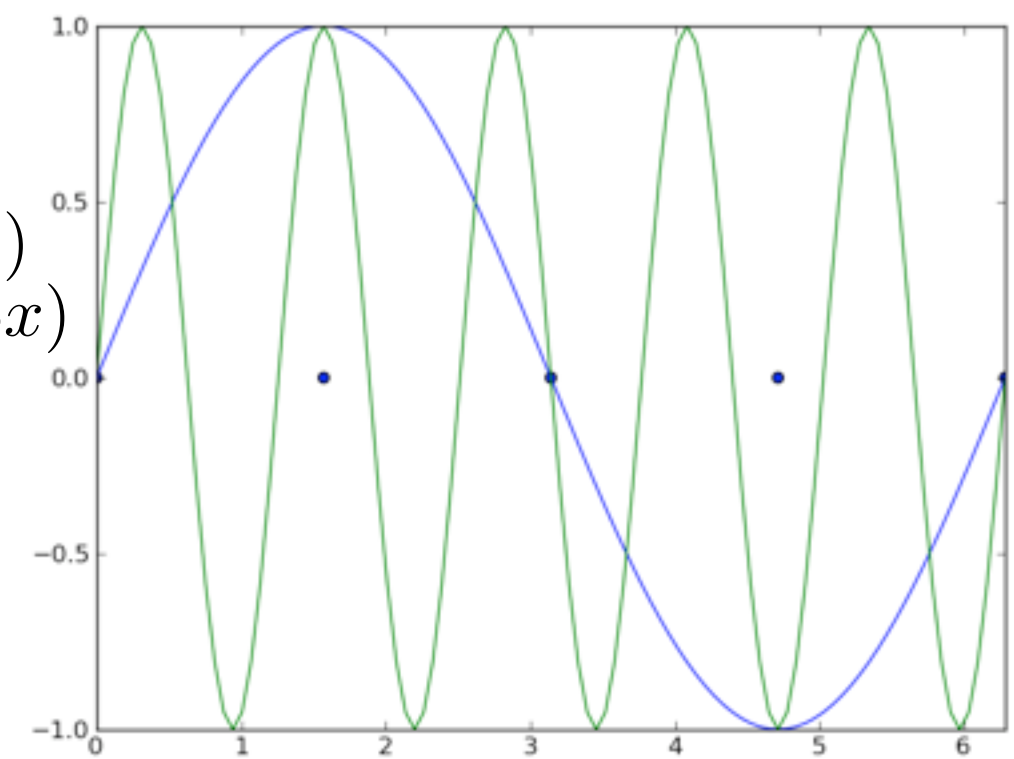


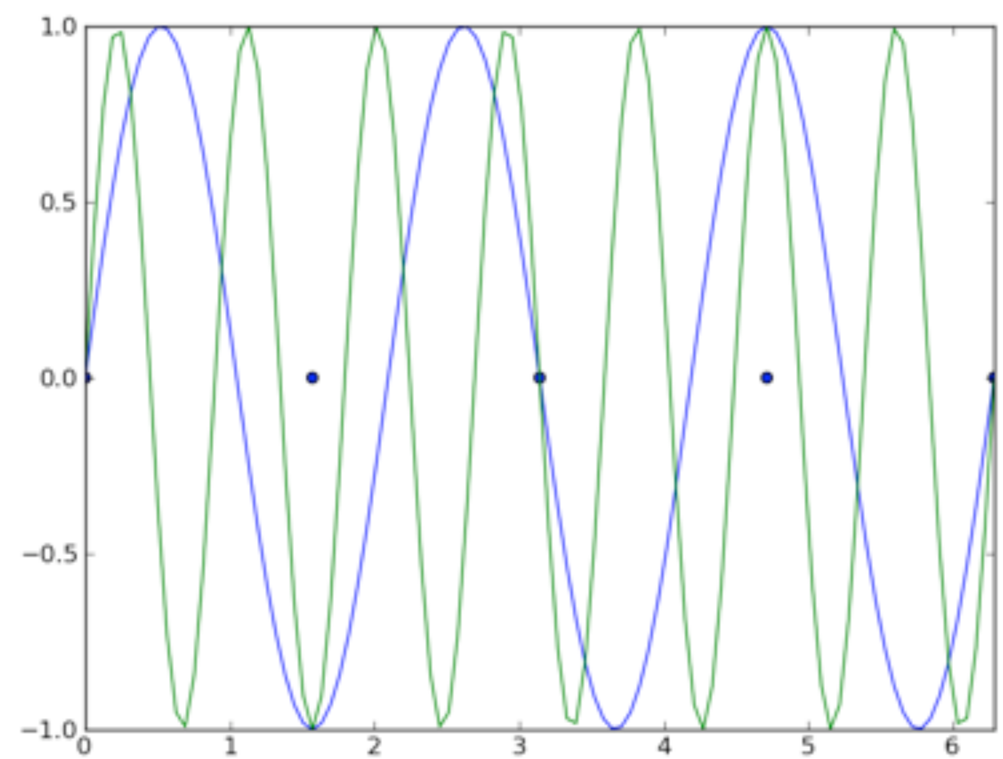
Discrete Fourier Transform (DFT)

- The case of discrete $u(x)$ we have aliasing



$\sin(x)$
$\sin(5x)$





$\sin(3x)$
$\sin(7x)$

Values at sample points repeat at $k_2 = k_1 + N$

Discrete Fourier Transform (DFT)

- DFT

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

- Inverse DFT

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

Fast Fourier Transform (FFT)

- Fast method to calculate the DFT
- Computations drop from $O(N^2)$ to $O(N\log(N))$
  - N = 10^4:
    - Naive: 10^8 computations
    - FFT: $4*10^4$ computations

Huge reduction!

- Many algorithms, let's look at Cooley-Tukey radix-2

Fast Fourier Transform (FFT)

- Cooley-Tukey radix 2
  - Assume N being a power of 2

$$\hat{u}_k = \sum_{j=0}^{N-1} u_j e^{-\frac{2\pi i}{N}kj}$$

Divide sum into even and odd parts

$$\hat{u}_k = \sum_{j=0}^{N/2-1} u_{2j} e^{-\frac{2\pi i}{N}k(2j)} + \sum_{j=0}^{N/2-1} u_{2j+1} e^{-\frac{2\pi i}{N}k(2j+1)}$$

Fast Fourier Transform (FFT)

- Cooley-Tukey radix 2
  - Assume N being a power of 2

$$\hat{u}_k = \sum_{j=0}^{N-1} u_j e^{-\frac{2\pi i}{N} kj}$$

Divide sum into even and odd parts

$$\hat{u}_k = \boxed{\sum_{j=0}^{N/2-1} u_{2j} e^{-\frac{2\pi i}{N} k(2j)}} + \boxed{\sum_{j=0}^{N/2-1} u_{2j+1} e^{-\frac{2\pi i}{N} k(2j+1)}}$$

Even Odd

Fast Fourier Transform (FFT)

$$\hat{u}_k = \sum_{j=0}^{N/2-1} u_{2j} e^{-\frac{2\pi i}{N/2}kj} + e^{-\frac{2\pi i}{N}k} \sum_{j=0}^{N/2-1} u_{2j+1} e^{-\frac{2\pi i}{N/2}kj}$$

- By doing this recursively until there is no sum, you get log(N) levels
- Sum is decomposed and redundant operations appear
- 4 point transform

$$\hat{u}_k = u_0 + u_1 e^{-\frac{2\pi}{4}ik} + u_2 e^{-\frac{2\pi}{4}i2k} + u_3 e^{-\frac{2\pi}{4}i3k}$$

$$\hat{u}_k = u_0 + u_2 e^{-\frac{2\pi}{4}i2k} + e^{-\frac{2\pi}{4}ik}\left(u_1 + u_3 e^{-\frac{2\pi}{4}i2k}\right)$$

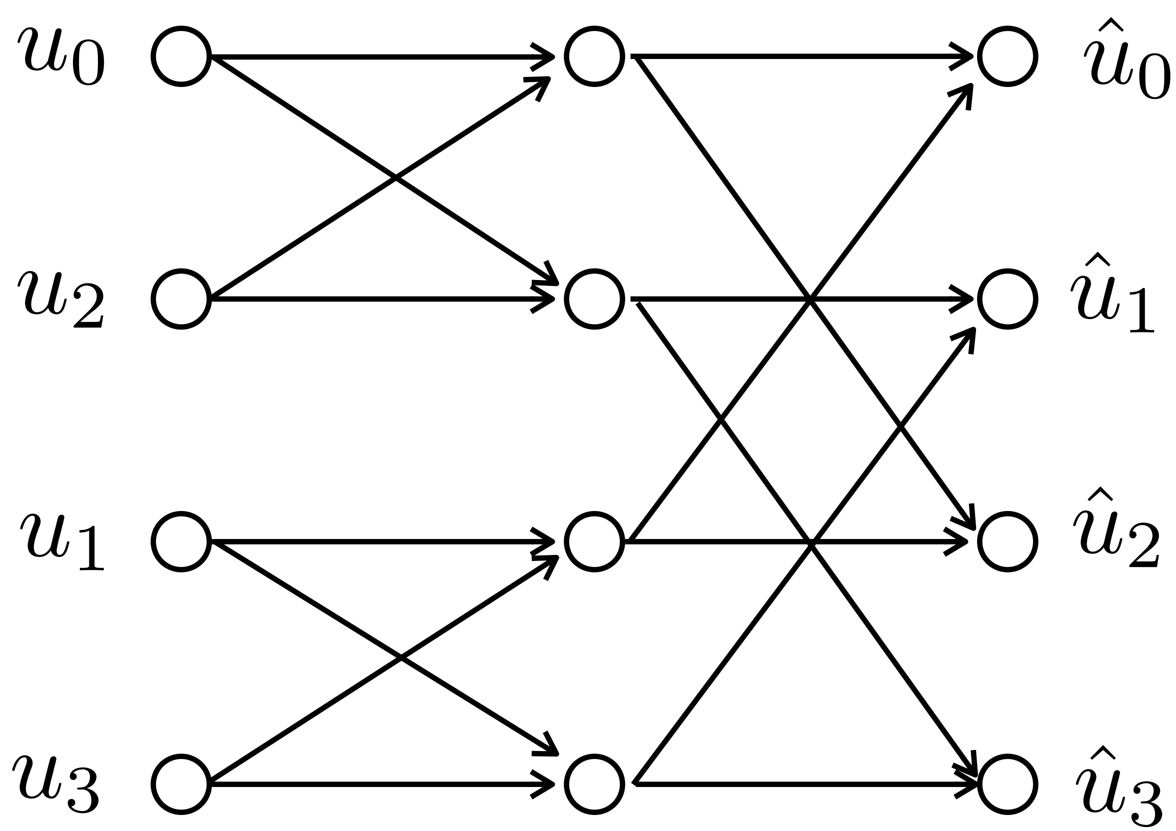
$$\hat{u}_k = u_0 + u_2 e^{-\pi ik} + e^{-\frac{\pi}{2}ik}\left(u_1 + u_3 e^{-\pi ik}\right)$$

$$k = 0, 1, 2, 3$$

Fast Fourier Transform (FFT)

$$\hat{u}_0 = u_0 + u_2 e^0 + e^0 (u_1 + u_3 e^0)$$
$$\hat{u}_1 = u_0 + u_2 e^{-\pi i} + e^{-\frac{\pi}{2} i} (u_1 + u_3 e^{-\pi i})$$
$$\hat{u}_2 = u_0 + u_2 e^{-2\pi i} + e^{-\pi i} (u_1 + u_3 e^{-2\pi i})$$
$$\hat{u}_3 = u_0 + u_2 e^{-3\pi i} + e^{-3\frac{\pi}{2} i} (u_1 + u_3 e^{-3\pi i})$$

$$\text{periodicity} \Rightarrow e^0 = e^{-2\pi i} = 1, \quad e^{-\pi i} = e^{-3\pi i} = -1$$



FFT - Motivation

- Signal processing
  - Signal comes in time domain, but want the frequency spectrum
- Convolution, filters
  - Signals can be filtered with convolutions

$$\int_0^t f(s)g(t-s)\,ds, \qquad 0 \leq t < \infty$$

$$\mathcal{F}\{f * g\} = \mathcal{F}\{f\} \cdot \mathcal{F}\{g\}$$

FFT - Motivation

- Partial Differential Equations (PDEs) - Spectral methods
  - Use DFTs to calculate derivatives

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

For evenly spaced grid

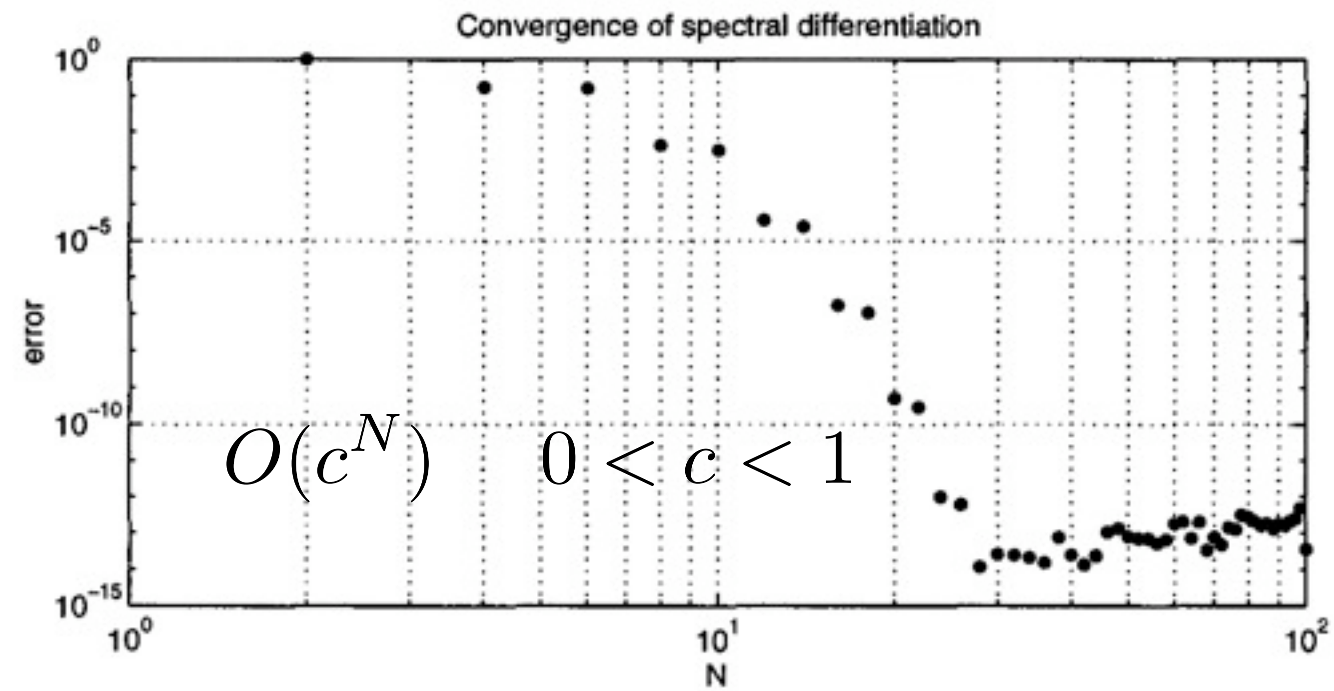
$$u_j = \sum_{k=0}^{N-1} \hat{u}_k e^{ikx_j}$$

$$\frac{\partial u_j}{\partial x} = \sum_{k=0}^{N-1} ik\hat{u}_k e^{ikx_j} \qquad \widehat{\frac{\partial u}{\partial x}} = ik\hat{u}$$

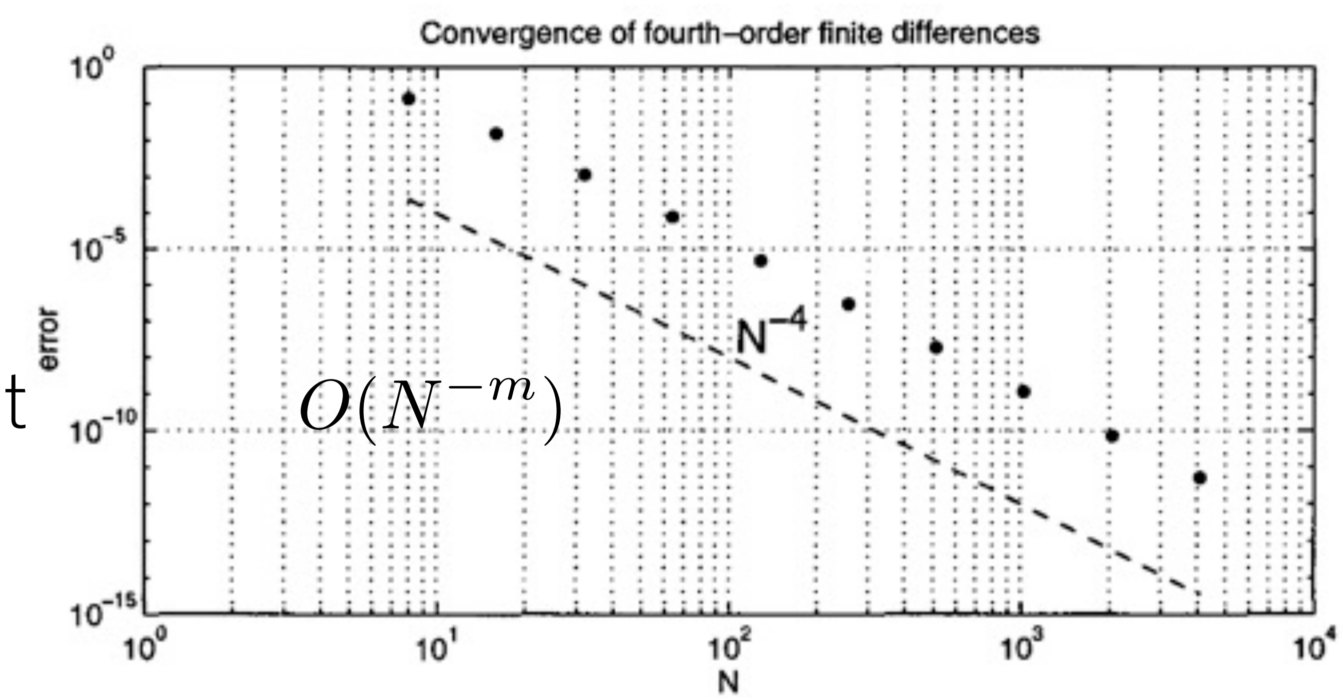
$$\frac{\partial^2 u_j}{\partial x^2} = \sum_{k=0}^{N-1} -k^2 \hat{u}_k e^{ikx_j}$$

FFT - Motivation

- Advantages
  - Spectral accuracy



- Limitations
  - Grid constraints
  - Boundary condition constraint



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)
  - In-place or out-of-place transforms
  - Double precision on GT200 or later
  - Allows streamed execution: simultaneous computation and data movement

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

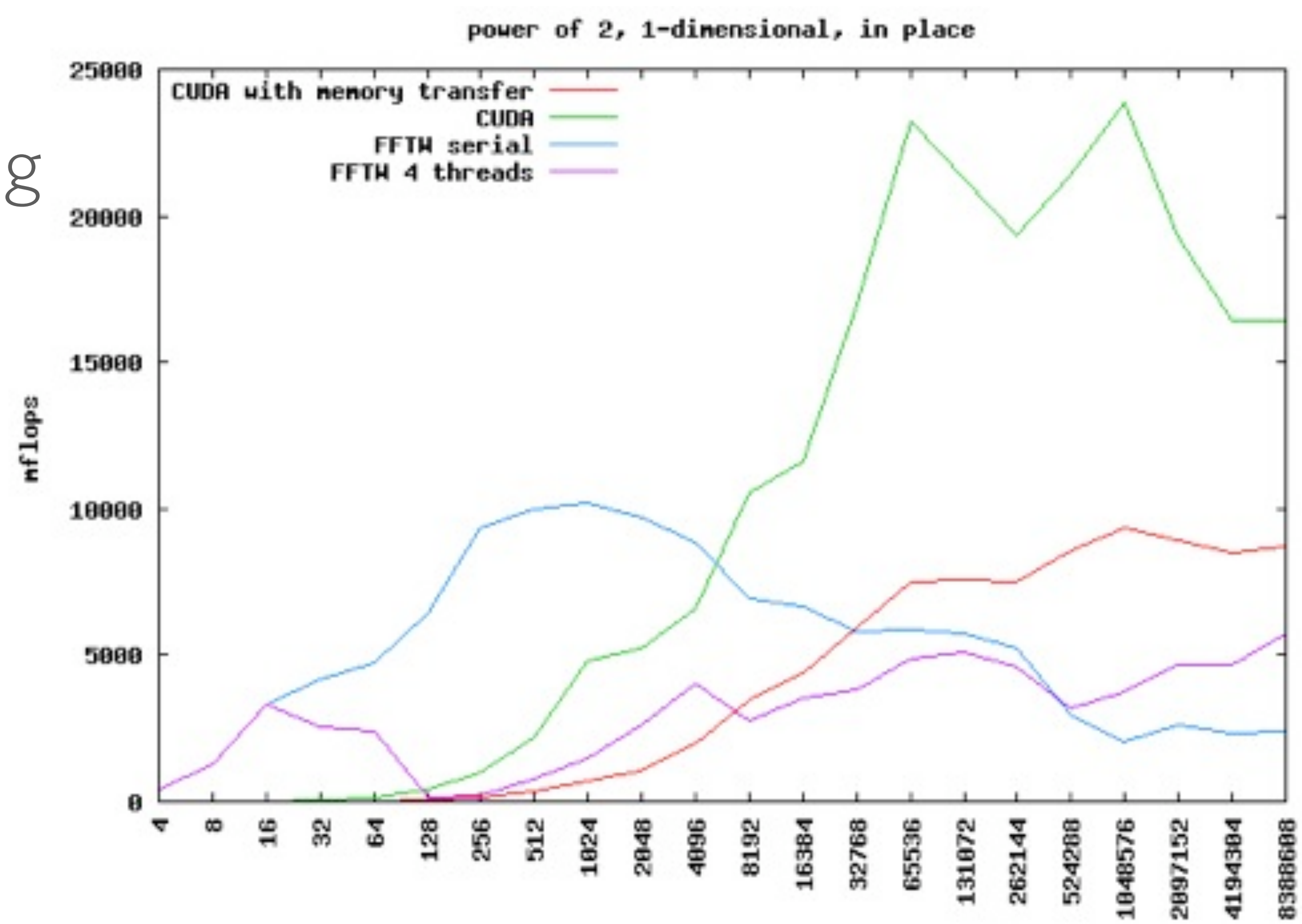
- cufftDestroy
  - Free GPU resources
- cufftExecC2C, R2C, C2R, Z2Z, D2Z, Z2D
  - Performs the specified FFT
- For more details see the CUFFT Library documentation available in the NVIDIA website!

CUFFT - Performance considerations

- Several algorithms for different sizes
- Performance recommendations
  - Restrict size to be a multiple of 2, 3, 5 or 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 X-dimension of single precision transforms to be strictly a power of two between 2(2) and 2048(8192) for Tesla (Fermi) GPUs

CUFFT - Performance considerations

- CUFFT vs FFTW
  - CUFFT is good for larger, power of two sized FFTs
  - CUFFT is not good for small sized FFTs
    - CPU can store all data in cache
    - GPU data transfers take too long



http://www.sharcnet.ca/~merz/CUDA_benchFFT/

CUFFT - Example

```
#include <cufft.h>
#define NX 256
#define BATCH 10

cufftHandle plan;
cufftComplex *data;
cudaMalloc((void**)&data, sizeof(cufftComplex)*NX*BATCH);

/* 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);

/* Destroy the CUFFT plan. */
cufftDestroy(plan);
cudaFree(data);
```

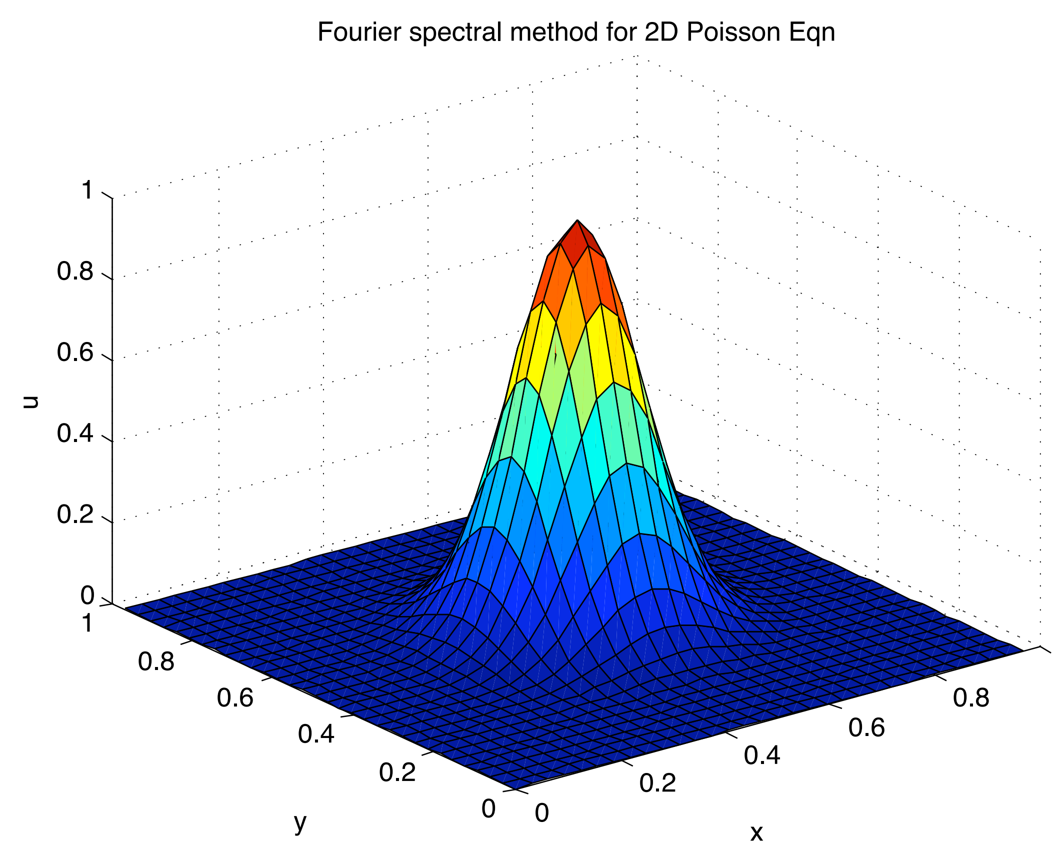
CUFFT - Example

- Solve Poisson equation using FFT

$$\nabla^2 u = \frac{r^2 - 2\sigma^2}{\sigma^4} e^{-\frac{r^2}{2\sigma^2}}$$

$$u_{an} = e^{-\frac{r^2}{2\sigma^2}} \qquad r = \sqrt{(x-0.5)^2 + (y-0.5)^2}$$

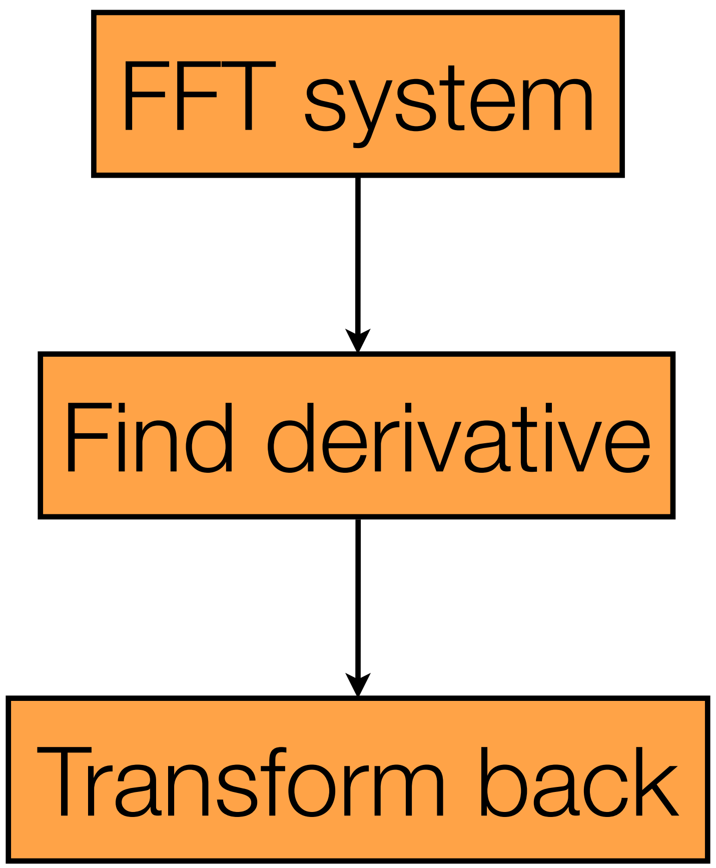
- Consider periodic boundary conditions



CUFFT - Example

- Steps

$$\nabla^2 u = f$$



$$-k^2 \hat{u} = \hat{f}$$

$$\hat{u} = -\frac{\hat{f}}{k^2}$$

$$u = \text{ifft}\left(-\frac{\hat{f}}{k^2}\right)$$

$$k^2 = k_x^2 + k_y^2$$

CUFFT - Example

```
int main()
{
        int N = 64;
        float   xmax  = 1.0f, xmin  = 0.0f, ymin  = 0.0f,
                h     = (xmax-xmin)/((float)N), s     = 0.1, s2    = s*s;

        float   *x = new float[N*N], *y = new float[N*N], *u = new float[N*N],
                *f = new float[N*N], *u_a = new float[N*N], *err = new float[N*N];

        float r2;
        for (int j=0; j<N; j++)
                for (int i=0; i<N; i++)
                {       x[N*j+i] = xmin + i*h;
                        y[N*j+i] = ymin + j*h;

                        r2 = (x[N*j+i]-0.5)*(x[N*j+i]-0.5) + (y[N*j+i]-0.5)*(y[N*j+i]-0.5);
                        f[N*j+i] = (r2-2*s2)/(s2*s2)*exp(-r2/(2*s2));
                        u_a[N*j+i] = exp(-r2/(2*s2)); // analytical solution
                }

        float   *k = new float[N];
        for (int i=0; i<=N/2; i++)
        {       k[i] = i * 2*M_PI;
        }
        for (int i=N/2+1; i<N; i++)
        {       k[i] = (i - N) * 2*M_PI;
        }
```

CUFFT - Example

```
// Allocate arrays on the device
float *k_d, *f_d, *u_d;
cudaMalloc ((void**)&k_d, sizeof(float)*N);
cudaMalloc ((void**)&f_d, sizeof(float)*N*N);
cudaMalloc ((void**)&u_d, sizeof(float)*N*N);

cudaMemcpy(k_d, k, sizeof(float)*N, cudaMemcpyHostToDevice);
cudaMemcpy(f_d, f, sizeof(float)*N*N, cudaMemcpyHostToDevice);

cufftComplex *ft_d, *f_dc, *ft_d_k, *u_dc;
cudaMalloc ((void**)&ft_d, sizeof(cufftComplex)*N*N);
cudaMalloc ((void**)&ft_d_k, sizeof(cufftComplex)*N*N);
cudaMalloc ((void**)&f_dc, sizeof(cufftComplex)*N*N);
cudaMalloc ((void**)&u_dc, sizeof(cufftComplex)*N*N);

dim3 dimGrid  (int((N-0.5)/BSZ) + 1, int((N-0.5)/BSZ) + 1);
dim3 dimBlock (BSZ, BSZ);
real2complex<<<dimGrid, dimBlock>>>(f_d, f_dc, N);

cufftHandle plan;
cufftPlan2d(&plan, N, N, CUFFT_C2C);
```

CUFFT - Example

```
cufftExecC2C(plan, f_dc, ft_d, CUFFT_FORWARD);

solve_poisson<<<dimGrid, dimBlock>>>(ft_d, ft_d_k, k_d, N);

cufftExecC2C(plan, ft_d_k, u_dc, CUFFT_INVERSE);

complex2real<<<dimGrid, dimBlock>>>(u_dc, u_d, N);

cudaMemcpy(u, u_d, sizeof(float)*N*N, cudaMemcpyDeviceToHost);

float constant = u[0];
for (int i=0; i<N*N; i++)
{       u[i] -= constant; //substract u[0] to force the arbitrary constant to be 0
}
```

CUFFT - Example

```
__global__ void solve_poisson(cufftComplex *ft, cufftComplex *ft_k, float *k, int N)
{
        int i = threadIdx.x + blockIdx.x*BSZ;
        int j = threadIdx.y + blockIdx.y*BSZ;
        int index = j*N+i;

        if (i<N && j<N)
        {
                float k2 = k[i]*k[i]+k[j]*k[j];
                if (i==0 && j==0) {k2 = 1.0f;}
                ft_k[index].x = -ft[index].x/k2;
                ft_k[index].y = -ft[index].y/k2;
        }
}
```

CUFFT - Example

```
__global__ void real2complex(float *f, cufftComplex *fc, int N)
{
        int i = threadIdx.x + blockIdx.x*blockDim.x;
        int j = threadIdx.y + blockIdx.y*blockDim.y;
        int index = j*N+i;

        if (i<N && j<N)
        {       fc[index].x = f[index];
                fc[index].y = 0.0f;
        }
}

__global__ void complex2real(cufftComplex *fc, float *f, int N)
{
        int i = threadIdx.x + blockIdx.x*BSZ;
        int j = threadIdx.y + blockIdx.y*BSZ;
        int index = j*N+i;

        if (i<N && j<N)
        {
                f[index] = fc[index].x/((float)N*(float)N);
                //divide by number of elements to recover value
        }
}
```

PyCUDA

- Python + CUDA = PyCUDA
- Python: scripting language ⟶ easy to code, but slow
- CUDA ⟶ difficult to code, but fast!
- PyCUDA wants to take the best of both worlds
- http://mathema.tician.de/software/pycuda

PyCUDA

- Scripting language
  - High level programming language that is interpret by another program at runtime rather than compiled
  - Advantages: ease on programmer
  - Disadvantages: slow (specially inner loops)
- PyCUDA
  - CUDA codes does not need to be a constant at compile time
  - Machine generated code: automatic manage of resources

PyCUDA

```
import pycuda.autoinit
import pycuda.driver as drv
import numpy

from pycuda.compiler import SourceModule
mod = SourceModule("""
__global__ void multiply_them(float *dest, float *a, float *b)
{
  const int i = threadIdx.x;
  dest[i] = a[i] * b[i];
}
""")

multiply_them = mod.get_function("multiply_them")

a = numpy.random.randn(400).astype(numpy.float32)
b = numpy.random.randn(400).astype(numpy.float32)

dest = numpy.zeros_like(a)
multiply_them(
        drv.Out(dest), drv.In(a), drv.In(b),
        block=(400,1,1), grid=(1,1))

print dest-a*b
```

PyCUDA

- Transferring data

```
import numpy
a = a.astype(numpy.float32)
a_gpu = cuda.mem_alloc(a.nbytes)

cuda.memcpy_htod(a_gpu, a)
```

- Executing a kernel

```
from pycuda.compiler import SourceModule
mod = SourceModule("""
  __global__ void doublify(float *a)
  {
    int idx = threadIdx.x + threadIdx.y*4;
    a[idx] *= 2;
  }
  """)
... # Allocate, generate and transfer
func = mod.get_function("doublify")
func(a_gpu, block=(4,4,1))

a_doubled = numpy.empty_like(a)
cuda.memcpy_dtoh(a_doubled, a_gpu)
print a_doubled
print a
```