# GPU-minicourse 2012

# Assignment 3: Spectral derivatives

## 1 Background: Spectral differentiation

Spectral methods expand spatially dependent data in terms of a basis-series,

$$u(x) = \sum_{k=0}^{N} \tilde{u}_k T_k(x). \tag{1}$$

Here, the $T_k(x)$ are the known basis-functions. For a Fourier-series, these are sines and cosines. For an interval with boundaries, the $T_k(x)$ are usually Chebyshev polynomials,

$$T_k(x) = \cos(k \arccos(x)). \tag{2}$$

Associated with each set of basis-functions are carefully chosen grid-points, which are called collocation points. For Chebyshev,

$$x_k = \cos(\pi k/N), \qquad k = 0, \ldots, N. \tag{3}$$

The function values at the collocation points are denoted by $u_i \equiv u(x_i)$.

You might ask: Why bother? The fundamental reason is that the accuracy increases *exponentially* with the number $N$ of basis-functions. Therefore, spectral methods tend to be far more accurate and efficient than finite-difference methods. For the supplied example `1D_Example.cpp` a mere 64 collocation points give an accuracy of $10^{-9}$, despite the test-function being quite nasty.

Derivatives can be represented as matrix-multiplications. For instance, to compute the x-derivative at the i-th collocation point:

$$\frac{\partial u}{\partial x}(x_i) = \sum_{j=0}^{N} D_{ij} u_j. \tag{4}$$

The matrix $D_{ij}$ can be computed quite easily.

Equation (4) is a simple matrix-multiplication, which we have addressed in GPU-class.

### 1.1 Dimension $d \geq 2$

In higher dimensions, one generally uses a product of 1-D basis-functions, e.g.

$$u(x, y) = \sum_{k=0}^{N_x} \sum_{l=0}^{N_y} \tilde{u}_{kl} T_k(x) T_l(y). \tag{5}$$

Data is now represented on a 2-D grid:

$$u(x_i, y_j), \quad i = 0, \ldots, N_x, \;\; j = 0, \ldots, N_y, \tag{6}$$

where $x_i$ and $y_j$ are the collocation points of the 1-D Chebyshev series.

Now comes the fun part. Derivatives are taken by matrix-multiplication over *one* of the two indices:

$$\frac{\partial u}{\partial x}(x_i, y_j) = \sum_{k=0}^{N_x} D^x_{ik} u(x_k, y_j), \tag{7}$$

$$\frac{\partial u}{\partial y}(x_i, y_j) = \sum_{k=0}^{N_y} D^y_{jk} u(x_i, y_k). \tag{8}$$

The matrices $D^x_{ij}$ and $D^y_{ij}$ are both square, but have different dimension when $N_x \neq N_y$.

As far as numerical implementation is concerned, Eqns. (**??**) and (8) differ from the 1-D case above in two crucial aspects:

1. Instead of 1 matrix-multiplication, we have $N_y$ separate multiplications by $D^x$ in Eq. 7, and $N_x$ separate multiplications by $D^y$ in Eq. (8. Thus, we have a moderate number of moderate-sized matrix-multiplications.

2. Second, a 2-D grid is usually mapped to 1-D memory via

$$u(x_i, y_j) = u[i\, N_y + j]. \tag{9}$$

Therefore, in one of Eqs. (7), (8), the matrix-multiplication will act on *contiguous* memory location, whereas in the other, it will act on *strided* memory.

# 2 Homework assignments

1. In class, I described BLAS's matrix-multiply, and gave two year old benchmark numbers.
   (a) Code BLAS matrix multiply on CPU and with CUDA in double precision (*hint*: BLAS `dgemm`).
   (b) On the ARC cluster, collect benchmark information. Sample matrix-sizes $N$ which are powers of 2, and which are not powers of 2.
   (c) Make simple estimates of the number of FLOPS performed, and of the amount of data moved between GPU-RAM and GPU-cores. Compute percentage of peak FLOPS and peak memory bandwidth.

2. The supplied programs `1D_Example.cpp` and `2D_Example.cpp` implement spectral derivatives in 1-D and 2-D, respectively.
   (a) Implement the 2-D matrix multiply, Eqs. (7) and (8) on GPUs, by modifying `2D_Example.cpp` appropriately. (*hint*: BLAS `dgemm`). It suffices to port only the matrix-multiply, and not the surrounding operations.
   (b) Consider the case $N_x = N_y \equiv N$. Repeat (1b) and (1c) for various problem sizes $N$. Perform timings separately for Eq. (7) and (8). Do the timings differ? What is the reason?
   (c) Because spectral methods are so accurate, real-world applications use fairly low $N$, $N \sim 30$. This limits the amount of parallelization opportunities. However, the real world is 3-dimensional. For the x-derivatives one has then $N_y N_z$ independent matrix multiplies by $D^x_{ij}$. This means, parallelization opportunities are increased by a factor $N_z$.
   To mimik 3-D spectral transforms, use your 2-D code, but set $N_x = N$, $N_y = N^2$, as well as $N_x = N^2$, $N_y = N$. Repeat (1b) and (1c) for various problem sizes $N$.