A multi-GPU finite difference code for calculating elastic wave field

Kangchen Bai

User's guide:

To use the code for calculating wave field, first edit the Makefile. Make sure that the CC variable is directed to the path of your c compiler and NVCC variable is directed to the path of your CUDA compiler. Cuda7.5 and gcc 4.8 or higher version is recommended or otherwise you may need to change some of the compiling flags.

After editing the Makefile, you can install the code simply by typing make under the command line.

There is a demo inside the program, which is to calculate a wave-field generated by a point source at the center of a rectangular domain. You may run the compiled file by typing. /main. I will show the results of that simple demo in later discussions.

After running the program, there will be wavefield snapshot data generated in distributed form with each GPU output its own chunk of data. We use a python script plot_snapshot.py to gather all the pieces to make a whole wave-field snapshot.

Problem description:

This is a finite difference code that solves the 2 dimensional elastic wave equations. We use five components to represent an elastic wave field. Three of them are stress component $\sigma_{xx}, \sigma_{xz}, \sigma_{zz}$ and the other two are velocity component v_x and v_z.
The mathematical relation is:

$$\rho \frac{dv_x}{dt} = \frac{d\sigma_{xx}}{dx} + \frac{d\sigma_{xz}}{dz}$$

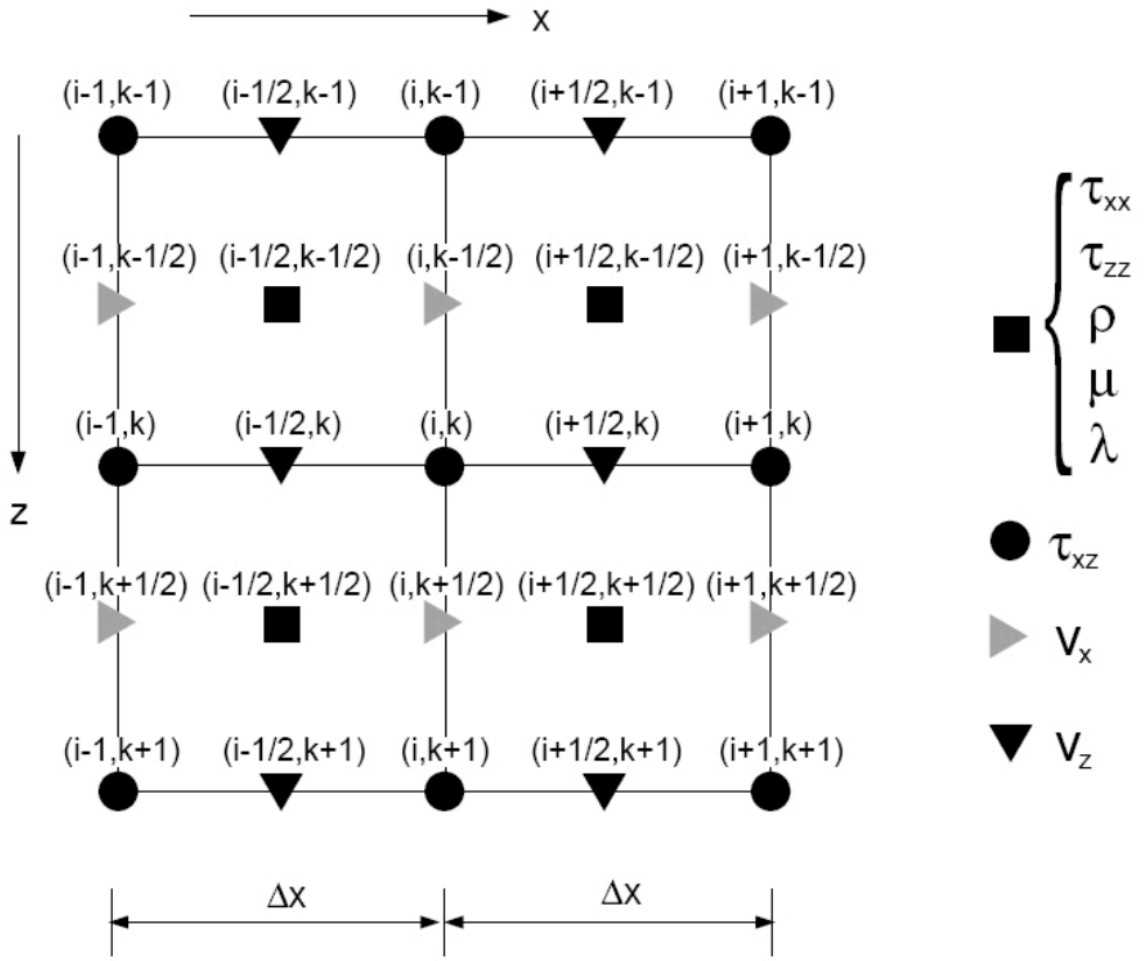
$$\rho \frac{dv_z}{dt} = \frac{d\sigma_{xz}}{dx} + \frac{d\sigma_{zz}}{dz}$$

$$\frac{d\sigma_{xx}}{dt} = (\lambda + 2\mu) \frac{dv_x}{dx} + \lambda \frac{dv_z}{dz}$$

$$\frac{d\sigma_{zz}}{dt} = (\lambda + 2\mu) \frac{dv_z}{dz} + \lambda \frac{dv_x}{dx}$$

$$\frac{d\sigma_{xz}}{dt} = \mu\left(\frac{dv_x}{dz} + \frac{dv_z}{dx}\right)$$

Here I am using a staggered grid finite difference scheme meaning that the stress and velocity are defined at different spatial and temporal grid. Stress are defined at NΔt while velocity are defined at (N+1/2)Δt. Stress are defined at (N+1/2)Δx or (N+1/2)Δz while velocity are defined at integer number of steps.



So the finite difference stencil can be formulated as such:

$$\rho \frac{vx_{n,m}^{k+1/2} - vx_{n,m}^{k-1/2}}{\Delta t} = \frac{\sigma xx_{n+1/2,m}^{k} - \sigma xx_{n-1/2,m}^{k}}{\Delta x} + \frac{\sigma xz_{n,m+1/2}^{k} - \sigma xz_{n,m-1/2}^{k}}{\Delta z}$$

$$\rho \frac{vz_{n+1/2,m+1/2}^{k+1/2} - vz_{n+1/2,m+1/2}^{k-1/2}}{\Delta t} = \frac{\sigma zz_{n+1/2,m+1}^{k} - \sigma zz_{n+1/2,m}^{k}}{\Delta z} + \frac{\sigma xz_{n+1,m+1/2}^{k} - \sigma xz_{n,m+1/2}^{k}}{\Delta x}$$

$$\frac{\sigma xx_{n+1/2,m}^{k} - \sigma xx_{n+1/2,m}^{k-1}}{\Delta t} = \frac{vx_{n+1,m}^{k-1/2} - vz_{n,m}^{k-1/2}}{\Delta x} + \frac{vz_{n+1/2,m+1/2}^{k} - vz_{n+1/2,m-1/2}^{k}}{\Delta z}$$

GPU implementation:

To implement the scheme, I have implemented two major GPU kernels. The first kernel calculates the spatial derivative of a given field, either being a spatial derivative in x direction or in z direction. The calculated spatial derivatives are first stored in intermediate variables. The second

kernel is one that updates the five components using the spatial derivative calculated by previous kernel.

The solver is made into a class. The class has member variables that stores all the five components and intermediate derivatives. The class has two methods that call the previously mentioned GPU kernels. When running on multiple GPUs, multiple class objects are created with one for each GPU. There is a method that communicates with object belonging to neighboring GPU for exchanging boundary values.

The simulation domain can be divided in one direction into multiple chunks. In the demo example, the 500*1000 sized domain is divided into 10 500*100 domain. Each GPU is calculating one chunk. Since the computer in my lab only had 3 GPU cards, we are using 10 separate thread to mimic 10 GPUs. But infact , all the calculations are on only 3 GPU(with each GPU taking 3 to 4 threads). The ten threads need to communicate with neighboring thread in order to update their boundary value.

Performance:

The GPU code certainly beats the CPU code. The speed up factor can be about 10-100 based upon analysis of the demo example. The GPU code makes 10000 iterations within less than 1 minute. As for multi GPU cases, since the GPU in my cluster do not support peer to peer access, the performance is penalized by communication cost. The attached figure is a wave field snapshot that is composed of 10 pieces contributed by 10 threads.