GPU tools for seismic wave modelling

Michael P. Lamoureux[1], Da Li [1] and RJ Vestrum[1]

ABSTRACT

Current computer desktops and laptops host dedicated graphics processing units (GPUs) which are typically used to speed up computations related to video display on the device. We can take advantage of the integrated GPU to speed up numerical calculations, making effective of parallel processing form of this architecture for raw code acceleration. This article summaries our experience in creating a teaching tool that demonstrates the use of the GPU to model seismic wave propagation in two dimensions, which runs sufficiently fast in real time to make it an enjoyable instructional device. The code is written in a Jupyter notebook, which is a computation tool that has been described in the journal **Nature** as the data scientists' computational notebook of choice (Perkel (2018)). Links are provide to run the demo on your own machine, via a virtual machine hosted online.

THE SNELL'S LAW DEMO

Our colleague Dr. Matt Yedlin at the Unviersity of British Columbia was intrigued by some of our earlier work on Jupyter notebooks (Lamoureux and Hardeman-Vooys (2018)) and suggested we try making a demo of Snell's law for presentation in his course on electromagnetics waves. Snell's law, of course, describes the refraction of a ray of light travelling through media with differing propagation velocities. As shown in Figure 1, the relation between the angles on incidence and refraction can be explained geometrically by aligning peaks and troughs in periodic plane waves traversing the material interface.

The question was then raised: can this be demonstrated in a numerical simulation of wave propagation? In particular, the goal was to see if we could simulate the propagation of a plane wave through a refracting media, and observe the change in direciton of the wave as it undergoes a velocity change. The answer is yes: Figure 2 shows a snapshot of a wave packet travelling through a two layer media, using a finite difference code to simulate the propagation of the wave. Becasue of the challenges in representing an infinite plane wave on a finite computational grid, we simplified the problem to a truncated plane wave – a wavet packet. This gives rise to a number of interesting physical phenomena that are echoed in the numerical simulation. First, the initially parallel wave fronts in the wave packet start to curve, as seen in Figure 2. Second, we see the wave peaks and troughs widen as the wave packet enters the lower region in the medium, the region of higher velocity, as seen in Figure 3. Third, there is a lot of interesting activity right at the refracting interface, such as the development of a head wave, as seen in Figure 4.

As this is a numerical simulation, there are interesting questions about what artifacts might have been introduced from numerical errors in the computation, or conceptual errors in the mathematical modelling of the physics. For instance, in this demonstration notebook, the boundaries of the computational region act as hard reflectors, so the wave packet will

[1]University of Calgary

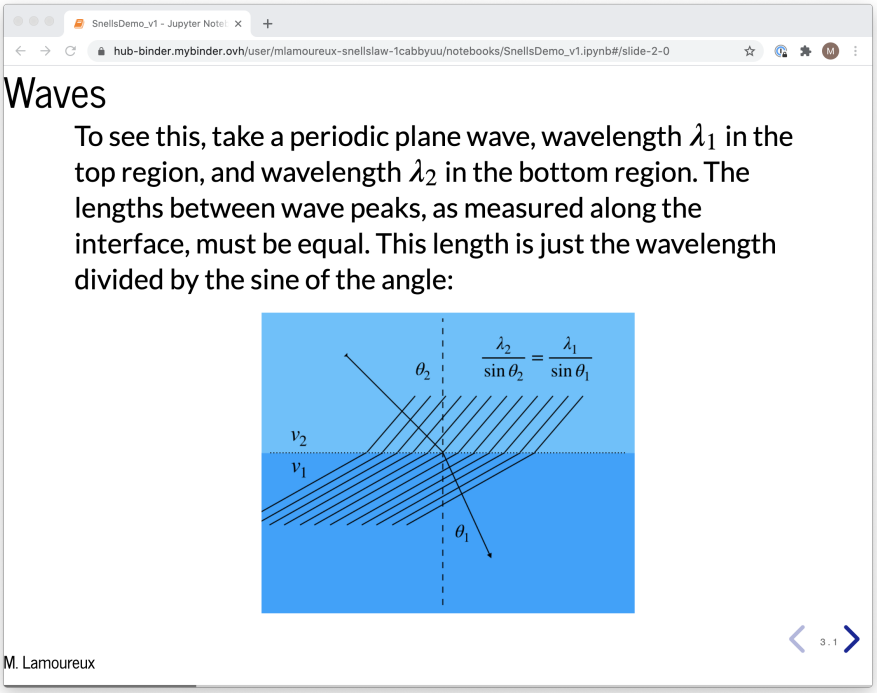

FIG. 1. Snapshot of Snell's law slideshow.

simply reflect at the boundary. While not shown here, it is easily seen when running the notebook in demonstration mode.

To make this a more useful demonstration tool, the code was finally implemented to run on the graphics processing unit, using the WebGL interface to access the GPU hardware vi the GPGPU tools created by Vizit Solutions (Klug (2015)). With a 1000×1000 grid size, we are able to get a real-time animation to run at 5 time steps per video frame, making a very pleasant and informative instructional presentation. As it is a live demo, there are controls to adjust parameters in the simulation, including wave velocities in the upper and lower layes of the medium, and the angle of incidence of the wave packet, as seen in Figure 2.

RUNNING THE DEMO YOURSELF

Since the Snell's law demo is a Jupyer notebook, it is a simple matter to share this material online using standard Jupyter resources. For instance, the MyBinder online tool takes any Jupyter code on a GitHub repository and runs it in a virtual machine in the cloud (typically on Google Cloud, for free) that you can use and experiment with. Clicking on the following link:

`https://mybinder.org/v2/gh/mlamoureux/SnellsLaw/main?filepath=SnellsDemo_v1.ipynb`

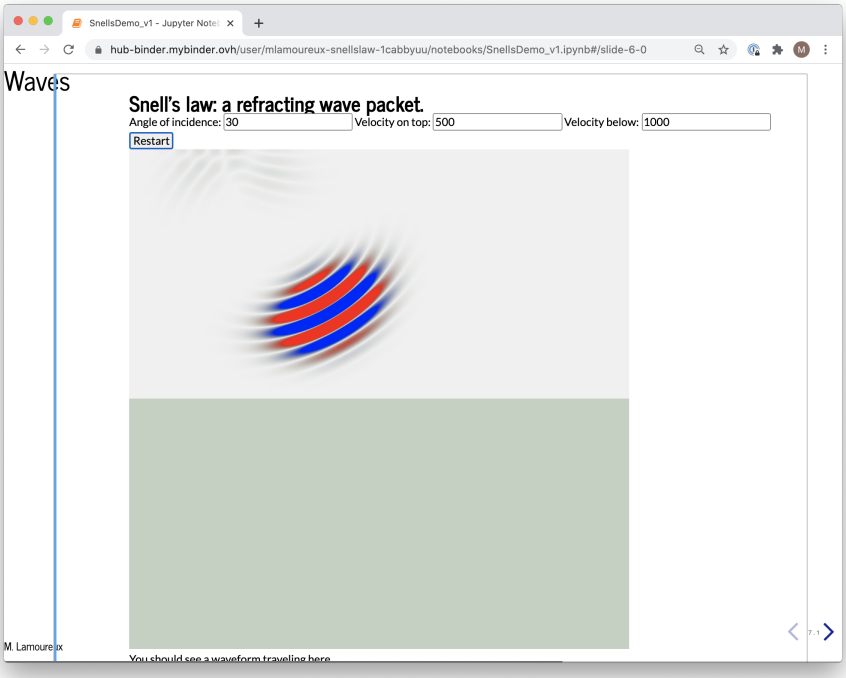

FIG. 2. Wave packet on initial trajectory.

will launch a copy of the notebook on a virutal machine in the cloud.[2] Follow the instructions shown in your browser to run the code and display the notebook as a slide show, which is similar to a PowerPoint deck of slides. The animation will be live, just jump to the appropriate slide.

Try it!

A second version of the notebook is available here:

`https://mybinder.org/v2/gh/mlamoureux/SnellsLaw/main?filepath=SnellsDemo_v2.ipynb`

This second version has a few more parameters that can be adjusted, related to the size and shape of the wave packet, and the wavelength of the enclosed waves. With more parameters, there is more "freedom" to mess up the simulation, but still, it works.

All the code is available for inspection at this github repository: `https://github.com/mlamoureux/SnellsLaw`

[2]It may take a few seconds to a minute, for the VM to initialize and launch.

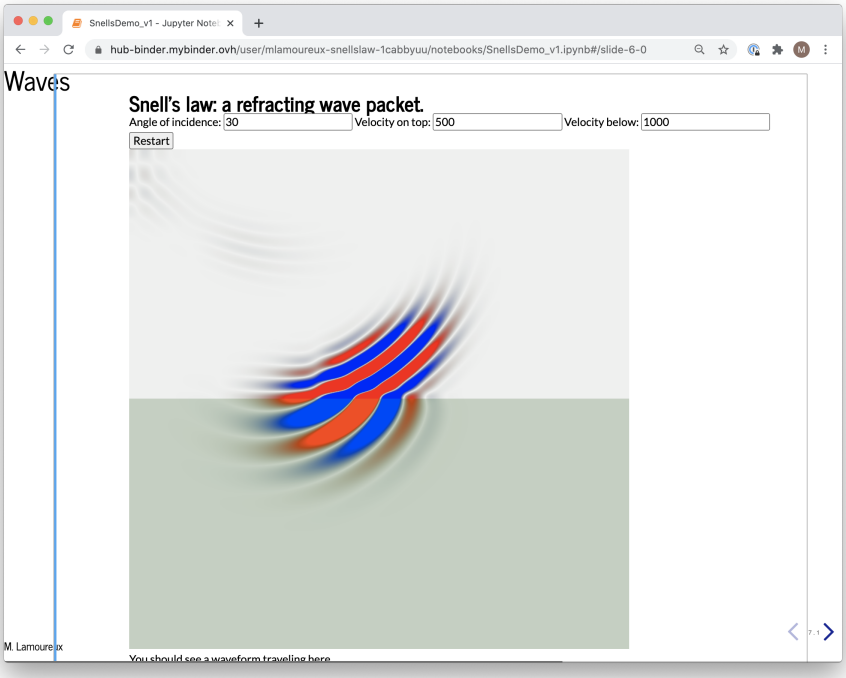

FIG. 3. Wave packet refracting at boundary.

THE GPU AS A PARALLEL PROCEESOR.

Graphical Processor Units are optimized to work on images and frame buffers which are essentially clusters of data points that eventually will be displayed on the screen. As a piece of hardware, the GPU consists of highly interconnected pieces of fast memory and hundreds to thousands of simple processors working on that memomry. The memory can be organizers as 2D or 3D grids, often optimized for 2D operations that ultimately end up on a two dimentional screen. These grids are essentially structured memory buffers, typically called “textures” in the WebGL language that is used to program them.

For numerical work, we can treat a 2D texture as a 2D computational grid. The GPU then assigns grid points to processors, where each processor runs a piece of code called a “fragment shader” that acts on that grid point, and stores the result in a new texture. Figure 5 shows an overview of the computational process, where the input texture holds a 2D grid of data, the GPU runs code that acts on the data in the input buffer, using the fragment shader, and then sends the result to the output buffer. The key to the acceleration is that the code in the fragment shader can be run by any one of the thousands of processing units on the GPU. With a thousand processors, the GPU can compute the outputs for a thousand data points on the grid at a time.

Notably, the textures in this structure are analogous to the spatial grid that one uses in finite difference code to solve a numerical partial differential equation. The GPU can allow the use of more than one grid as an input to the computation, but the the output grid has to be a separate, independent grid. In particular, one could use one texture/grid to store

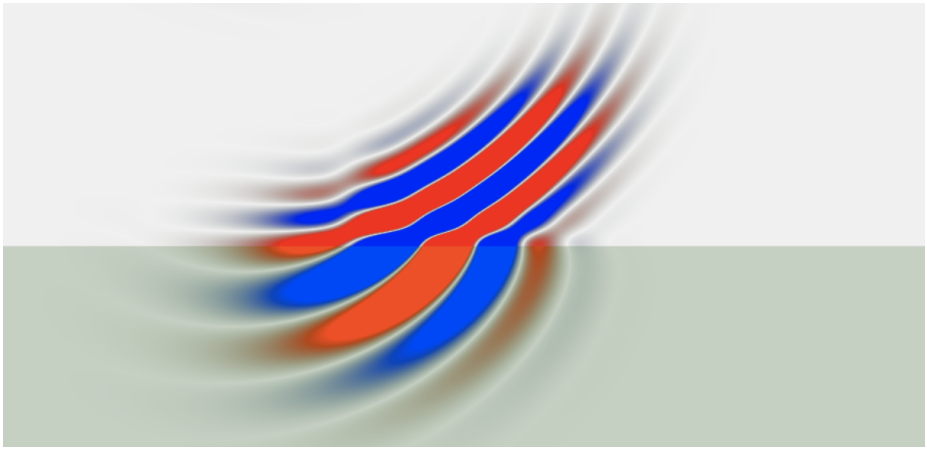

FIG. 4. Closeup of the wave refraction at boundary.

wavefield amplitudes and another texture/grid to velocity information. For time stepping finite difference, one can use a sequence of textures/grids, or to swap buffers from input to output and back again to keep the corresponding grids independent.

The Snell's Law demo mentioned above makes use of whatever video card and GPU that is available on a given destop computer or laptop. Modern web browsers are GPU-aware, so when there is a GPU available, the browser will make use of this. By using the WebGL API to request GPU services via the browser, we also avoid writing specialized GPU code[3] to access the GPU in its own language.

The WebGL API provides links through Javascript to set up these textures and buffers, define the shader code to do the computations, run the process, and swap memory as needed. This API is rather complex and difficult to learn on its own, so we use some pre-made utilities developed by Vizit Solutions (Klug (2015)). The central piece of code is Javascript file `GPGPUtilities.js`, a collection of General Purpose Graphics Processing Utilities, which sets up grids of appropriate sizes, fills them with data, defines the shader code, and rusn the GPU code to come each time step. We create two JavaScript files, one to do the finite difference code for the numerical wave modelling, and one to display the results.

THE TIME STEPPING CODE ON A GPU

We refer the reader to some references on finite differencing modelling, such as Myint-U and Debnath (2007), Lapidus and Pinder (1999), Zauderer (1989), Press et al. (2007).

Recall the time stepping algorithm for modelling acoustic wave propagation begins with the partial differential equation

$$\frac{\partial^2 u}{\partial t^2} = c^2 \nabla^2 u(x, y). \tag{1}$$

Discretizing in time and space, with the standard version of the discrete Laplacian, we

[3]such as CUDA

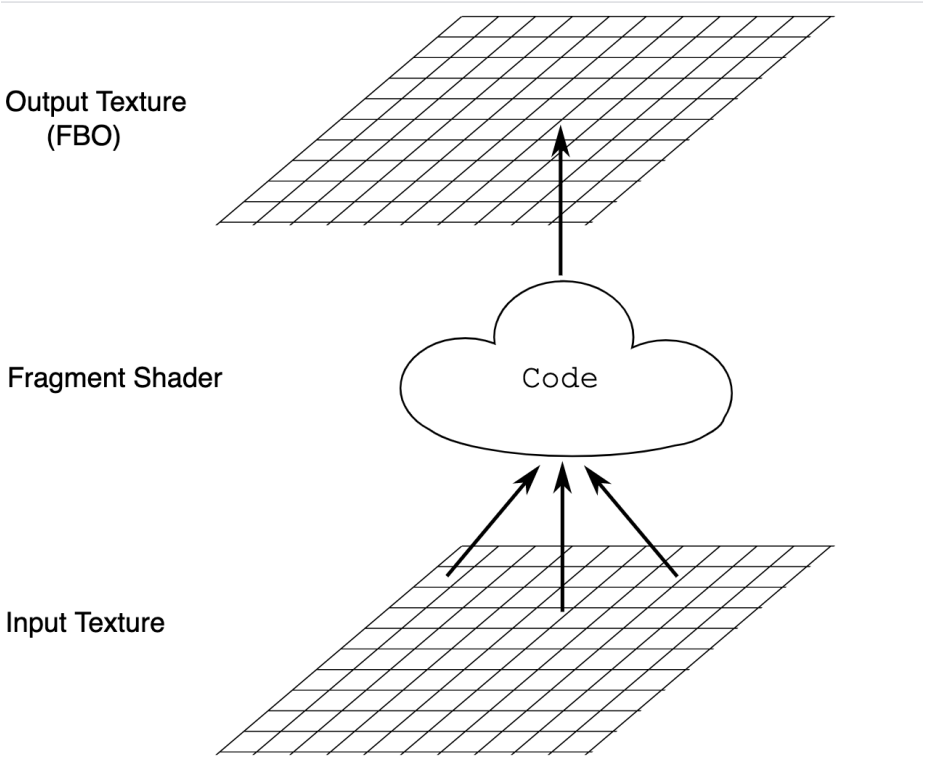

FIG. 5. Structure of the GPU processing. Image from Klug (2015)

obtain the discrete version

$$\frac{1}{\Delta t^2}(u^{k+1}-2u^k+u^{k-1}) = \frac{c^2}{\Delta x^2}[u^k(x\pm\Delta x,y)+u^k(x,y\pm\Delta y)-4u^k(x,y)], \tag{2}$$

where we are using the superscript notation $u^k(x,y) = u(x,y,k\Delta t)$ for simplicity, and assuming $\Delta x = \Delta y$. This results in the time stepping algorithm

$$u^{k+1} = 2u^k-u^{k-1}+c^2\frac{\Delta t^2}{\Delta x^2}[u^k(x\pm\Delta x,y)+u^k(x,y\pm\Delta y)-4u^k(x,y)]. \tag{3}$$

In Matlab (MATLAB (2020)) this algorithm is represented as three nested **FOR** loops, for each of the three variables x, y, t, as shown in Figure 6.

In contrast, the GPU fragment shader code is shown in Figure 7. The code is in the section called `main()`. Notice first that there are no loops in this program. This is because the GPU handles the x,y loop by assigning individual processors to each grid point, as indexed by the 2D vector `vec2 pt`. The time stepping is handled in a separate piece of code that does the rendering. The wavefield data that we are simulating is stored in the textures/grids identified as `sampler2D u, u1`, where `u` is the current time step and `u1` is the previous time step. In fact, each grid point actually holds four floating point values, corresponding to the Red, Green, Blue and Alpha channels that are used in video. For convenience, we store the wavefield amplitude in the Red channel (`texture2D(u,pt).r`) while the

```
// CODE SAMPLE - main timing loop (MATLAB)
nt = 10;
nx = 1000;
ny = 1000;
vel = zeros(nx,ny);
u = zeros(nt,nx,ny);
wt = (dt/dx)^2;
for k=2:nt-1
  for x = 2:nx-1
    for y = 2:nx-1
      u(k+1,x,y) = 2*u(k,x,y) - u(k-1,x,y) + ...
      (vel(x,y)^2)*wt*(u(k,x-1,y) + u(k,x+1,y) + u(k,x,y-1) + u(k,x,y+1) - 4*u(k,x,y));
    end
  end
end
```

FIG. 6. Main loop in MATLAB to compute time step.

```
// CODE SAMPLE -  GPU fragment shader
uniform sampler2D u,u1;
uniform vec2 dx,dy;
uniform float wt;
varying vec2 pt;
void main() {
  gl_FragColor.g = texture2D(u,pt).g;
  gl_FragColor.r = 2.0*texture2D(u,pt).r  - texture2D(u1,pt).r
    + gl_FragColor.g*wt*(
      - 4.0*texture2D(u,pt).r)
      + texture2D(u,pt+dx).r + texture2D(u,pt-dx).r)
      + texture2D(u,pt+dy).r + texture2D(u,pt-dy).r)
    );
}
```

FIG. 7. Main GPU fragment shader, to compute time step.

velocity (squared) information is stored in the Green channel (`texture2D(u,pt).g`). Notice the velocity information is used in the time stepping formula shown in the program `main()`.

Of course, this fragment shader code should be simple, in order for the GPU to work quickly on each grid point. For instance this is why our code avoids squaring the velocity at each time step; instead it is precalculated before storing in the texture/grid that needs this information.

But there is also a lot of flexibility in the fragment shader code. For instance, we can use a nine point template for the Laplacian, rather than the standard five point template. In Figure 8 we show the code for such a nine point template, where the weights `wt0, wtx, wty, wtd` are the values needed to compute the nine point stencil. Again for efficiency, these values are computed once, elsewhere in the code, then stored in the fragment shader

```
// CODE SAMPLE - GPU fragment shader, 9 point Laplacian
uniform sampler2D u,u1;
uniform vec2 dx,dy,dd,da;
uniform float wt0,wtx,wty,wtd;
varying vec2 pt;
void main() {
  gl_FragColor.g = texture2D(u, pt).g;
  gl_FragColor.r = 2.0*texture2D(u, pt).r - texture2D(u1,pt).r
    + gl_FragColor.g*(
        wt0 * (texture2D(u, pt).r)"
      + wtx * (texture2D(u,pt+dx).r + texture2D(u,pt-dx).r)
      + wty * (texture2D(u,pt+dy).r + texture2D(u,pt-dy).r)
      + wtd * (texture2D(u,pt+dd).r + texture2D(u,pt-dd).r )
      + wtd * (texture2D(u,pt+da).r + texture2D(u,pt-da).r )
      );"
}
```

FIG. 8. Main GPU fragment shader, 9 point Laplacian stencil.

```
// CODE SAMPLE - Fragment shader to display wavefield on screen
uniform sampler2D u;
varying vec2 pt;
void main() {
   float  amp;
   amp  = texture2D(u, pt).r;
   gl_FragColor = max(0., amp)*vec4(0.,0.,1.,1.)
    + max(0.,-amp)*vec4(1.,0.,0.,1.);
}
```

FIG. 9. Fragment shader to display the propagating wavefield.

as a non-varying float value.[4]

Finally, we need to display the waveform on the screen. This requires a fragment shader that takes the waveform data, and then writes it onto an active screen buffer on the computer. The code for this is shown in Figure 9. Again, the main program is a simple one line function that assigns blue pixels for positive values of the waveform, and red pixels for the negative values of the waveform, suitably weighted. Another line can be added to display velocity information, but we omit that in this discussion.

THE JUPYTER NOTEBOOK HOSTING THE GPU CODE

The Snell's Law demo was written in a Jupyter notebook, in order to make it easy to distribute to other researchers and students. The code is available here on github:

`https://github.com/mlamoureux/SnellsLaw`

The code for the numerical simulation is written in JavaScript, in order to access the local GPU (or video card) via the WebGL API. JavaScript is not our favourite language,

[4]In our Snell's law demo, we chose a weighted nine point stencil that gives a maximally circularly symmetrix Laplacian.

but it seems to be well suited for this task. The main tasks of the JS code is to

- set up a user interface for the demo, to set initial parameters;
- initialize the GPU environment;
- initialize three textures/grids as frame buffers, to circulate through three time steps u^{k+1}, u^k, u^{k-1};
- set up initial values for the wavefield – essential the initial wave packet;
- start and stop the numerical simulation;
- render the time steps onto the screen.

There is nothing too deep in this outside code – all the neat stuff is in the fragment shaders (for the mathematics) and in GPUtilities (for accessing the WebGL API). We refer you to the GitHub repo for details.

MARMOUSI DEMOS

With this time-stepping animation successfully working, it was time to test it out on a more complicated model. Fortunately, we have the Marmousi model easily at hand. With 10m spacing of the velocity points in the model, one obtains a grid size of about 900×300 points, which easily fits into our code above with 1000×1000 points.

Using Python in a Jupyter notebook, we converted a Matlab data file of Marmousi velocities into a JSON file, then used our JavaScript code to read that into the GPU. The rest is pretty straightforward, giving a demo that you can run yourself here:

`https://mybinder.org/v2/gh/mlamoureux/Marmousi/main?filepath=Marmousi_1.ipynb`

Code for the demo is on our github repo, at `https://github.com/mlamoureux/Marmousi`.

Results are shown in Figures 10 through 12, where we see an initial wave packet approaching the top of the model, then propagating through the various layers, and accelerating as appropriate through the faster layers. Of course, we can see that the code does not try to deal with the computational boundaries in a very elegant way, and we have not verified the accuracy of this method. We do observe that the numerical simulation appears to be stable.

For a demonstration of wave propagation in a complex media, this GPU implementation appears to be working.

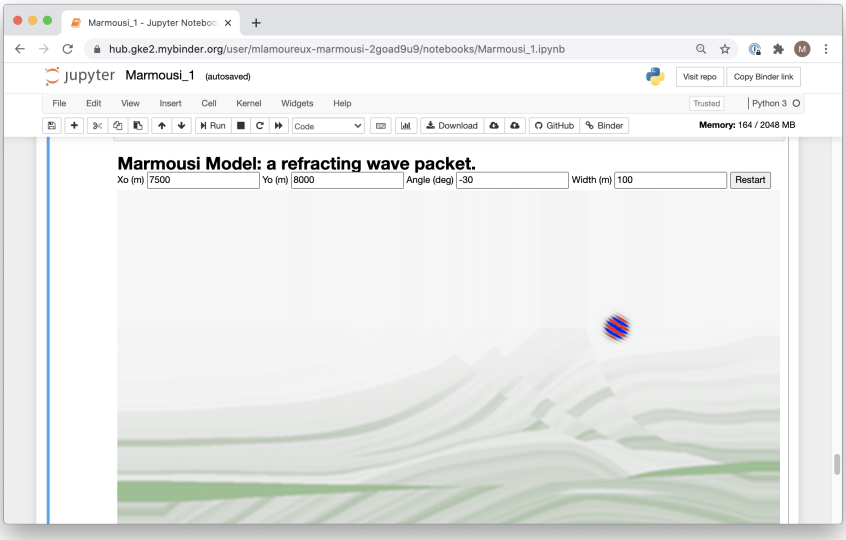

FIG. 10. Wave packet on initial trajectory, velocities in green.

FUTURE WORK

The Snell's Law and Marmousi models have been tested on a variety of desktop and laptop computers, in the Windows, Mac OS and Linux environments, all with good results when the machine has a good video card installed. We have also tested it on a Raspberry Pi with some success – the code runs, but very slowly. For future work, we would like to make the code more robust on a variety of platforms, so the demos can be more widely used.

The WebGL API also allows for 3D textures, which suggests a route to creating 3D models of the wave equation, for demonstration purposes.

It is interesting to note that the GPU textures/grids are set up to store four floating point values per grid point, and thus could store the three components of an elastic, vector-valued wavefield. Thus this would be suitable for modelling elastic wave equations in 3D, and we expect some useful demonstrations could be built from this.

The second author has developed advanced modelling code for dealing with computational boundaries (see Li et al. (2020)); the GPU method above is sufficiently flexible that the PML methods can be implemented in the demos as well. The third author has been developing computational methods based on grid algebras (see Lamoureux and Hardeman-Vooys (2016)) which will be used to develop stable, implicit algorithms for accurately solving the wave equation; we expect that these GPU methods will work here as well. Multigrid methods based on the GPU structure also looks like a promising project that we will pursue.

It is interesting to think of how the WebGL interface could be used for more than just demos – or perhaps it is better to bite the bullet and just start programming the GPU directly.

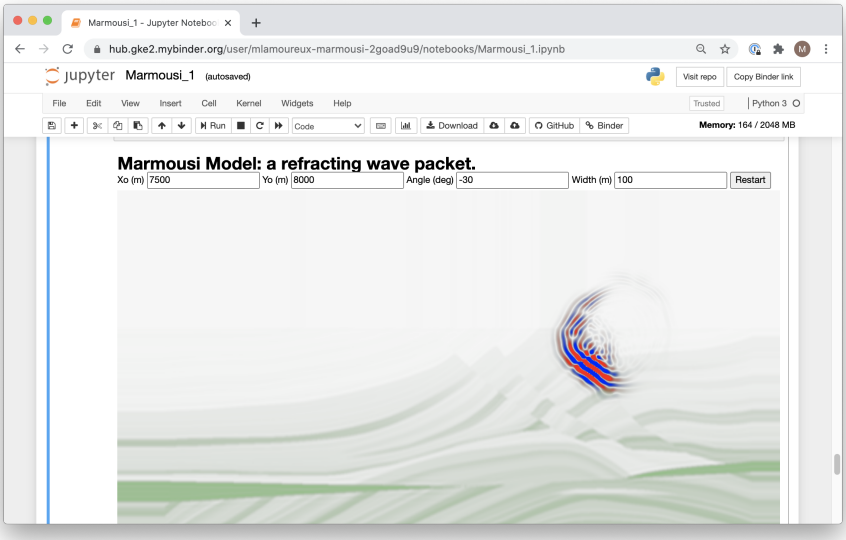

FIG. 11. Wave packet entering the inhomgeneous velocities.

SUMMARY

We have demonstrated the numerical simulation of seismic wave propagation in complex media through the use of a WebGL API software interface to directly access the computing resources of the GPU hardware available in most modern laptops and desktops. The result is a collection of live demos that effectively show wave propagation in real time, for use in informational demonstrations in class and other learning venues. The coding paradym for accessing GPU resoures is flexible and easy to learn, showing potential for wider using in a variety of simulation demos. We also demonstrate the utility of speeding up numerical simulations through the use of the GPU.

Links are provided to the live code demos that the interested reader can access, to run the seismic wave simulations on a cloud server.

ACKNOWLEDGEMENTS

We thank the sponsors of CREWES for their support. This work was funded by CREWES industrial sponsors, NSERC (Natural Science and Engineering Research Council of Canada) through the grants CRDPJ 461179-13 and CRDPJ 543578-19, and Discovery Grant RGPIN-2020-04561 of the first author. The second author thanks the China Scholarship Council (CSC) for supporting the research.We gratefully acknowledge the support of NVIDIA Corporation with the donation of a Titan Xp GPU used in earlier efforts in this research.

REFERENCES

Klug, A., 2015, GPGPUtility: Vizit Solutions, vizitsolutions.com.

Lamoureux, M. P., and Hardeman-Vooys, H. K., 2016, Grid algebra in finite difference schemes: CREWES Research Report, **28**.

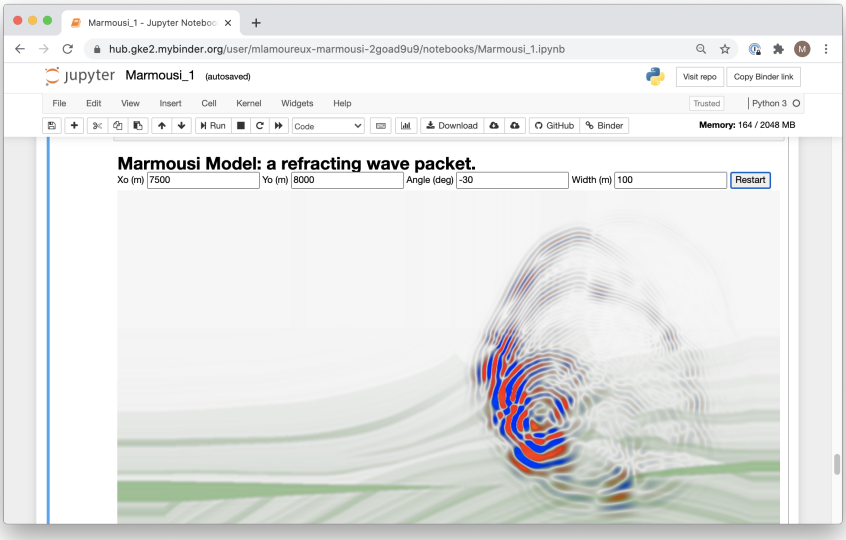

FIG. 12. Wave packet propagating further.

Lamoureux, M. P., and Hardeman-Vooys, H. K., 2018, Jupyter notebooks and hubs for scientific computing: CREWES Research Report, **30**.

Lapidus, L., and Pinder, G. F., 1999, Numerical Solution of Parital Diffential Euqations in Science and Engineering: John Wiley and Sons.

Li, D., Li, K., and Liao, W., 2020, Efficient and stable finite difference modelling of acoustic wave propagation in variable-density media, `arXiv/math.NA/2003.09812`.

MATLAB, 2020, version 9.9.0 (R2020b): The MathWorks Inc., Natick, Massachusetts.

Myint-U, T., and Debnath, L., 2007, Linear Partial Differental Equations for Scientists and Engineers: Birkhäuser.

Perkel, J. M., 2018, Why Jupyter is data scientists' computational notebook of choice: Nature, **563**, 145–146.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P., 2007, Numerical Recipes: The Art of Scientific Computing: Cambridge University Press.

Zauderer, E., 1989, Partial Differential Equations of Applied Mathematics: Wiley Interscience.