Jupyter notebooks and hubs for scientific computing

Michael P. Lamoureux[1] and Heather K. Hardeman-Vooys[1]

ABSTRACT

The journal **Nature** recently published an article entitled *Why Jupyter is data scientists' computational notebook of choice*. With three years of use under our belts, we discuss our experience with Jupyter notebooks, provide guidelines on how to make the transition to these tools for your own research, and present several useful resources to help you make this transition. As an illustrative example, we present a sample notebook recording our research efforts to achieve a 30x speedup in an implementation of standard finite difference code to numerically simulate acoustic waves in a variable velocity field using WebGL on a GPU-powered video card.

INTRODUCTION

Our goal in this article is to convince the reader to use Jupyter notebooks for their own research, scientific computing, and data analysis. But hey, we got scooped! The journal **Nature** just published an article (Oct. 30, 2018) on *Why Jupyter is data scientists' computational notebook of choice*, Perkel (2018).[2] The article makes a convincing argument, and we think it should be your tool of choice as well. As a brief summary, here is a quote from the Nature article:

> Jupyter is a free, open-source, interactive web tool known as a computational notebook, which researchers can use to combine software code, computational output, explanatory text and multimedia resources in a single document. Computational notebooks have been around for decades, but Jupyter in particular has exploded in popularity over the past couple of years. This rapid uptake has been aided by an enthusiastic community of user-developers and a redesigned architecture that allows the notebook to speak dozens of programming languages – a fact reflected in its name, which was inspired, according to co-founder Fernando Pérez, by the programming languages Julia (Ju), Python (Pyt) and R.
>
> One analysis of the code-sharing site GitHub counted more than 2.5 million public Jupyter notebooks in September 2018, up from 200,000 or so in 2015. In part, says Pérez, that growth is due to improvements in the web software that drives applications such as Gmail and Google Docs; the maturation of scientific Python and data science; and, especially, the ease with which notebooks facilitate access to remote data that might otherwise be impractical to download – such as from the [Large Synoptic Survey Telescope].

We have been using Jupyter notebooks for about three years, for our mathematical research, for our computing needs, and for our teaching, creating lecture notes for classes in

[1]University of Calgary

[2]`https://www.nature.com/articles/d41586-018-07196-1`.

mathematical analysis, numerical methods, and industrial applications. It has completely changed the way we approach key aspects of our research, including our ways of communicating, collaborating, and undertaking computational tasks.

To help convince the reader that it is worth a change, we will present in this article an example of using Jupyter notebooks to explore a new (to us) computational method, namely accessing our computer's GPU and video card to speed up our numerical computations in seismic wave propagation. We are able to achieve a speed-up by a factor of 15 to 30 times, depending on the video card, on a standard finite difference computation for the acoustic wave equation.

It is worth noting that we have been intending to explore the GPU for several years. We obtained a fast GPU card from Nvidia two years ago, upgraded a desktop Windows PC to host the GPU card and access it through Matlab, and did some initial experiments. However, it wasn't until we started the exploration with a Jupyter Notebook that this all came together into working code.

So we begin this article with the example of using a Jupyter Notebook for our GPU numerical work, and discuss how we did it. Then, we go into detail about what a Jupyter notebook is, the free and commercial services that support it, and how you can use it in your own work.

A JUPTYER SAMPLE: 30× SPEEDUP FOR GPU-FD CODE

As a sample project on using Jupyter to develop a research idea, we record the process of implementing finite difference code for a numerical simulation of the acoustic wave equation on a Graphic Processing Unit (GPU). A key goal is to create generic code that would run on any video card that hosts a fast GPU, rather than getting into the intricacies of any particular GPU system. Our target was to speed up the finite difference code so it would be more useful in seismic imaging applications, aiming for a speed improvement with a factor of ten. Our final result showed a speed up by a factor of 30, comparing Matlab to the GPU.

It is worth noting that our initial time tests suggested a speed-up by a factor of 1000 was possible, but more careful study indicated this was overly optimistic. With the Jupyter notebook, we chose to record both the first initial efforts (which were misleading) to the final, correct result. Overall, this was a very successful test, and also demonstrates the process of carefully recording results in a Notebook.

As a quick summary of the hardware, we are working on a 2014-vintage Power Mac that hosts two AMD FirePro D700 video cards. Each video card has a Tahiti-type graphics processors with 2048 cores and 6 Gigabytes of memory. We also have access to an Nvidia Titan Xp video card with 3840 cores and 12 Gigabytes of memory. The key step in this FD code is to update a 1000x1000 grid using a time-stepping algorithm. It takes roughly 10 million floating point operations per update. That is, we have ten FLOPS per grid point, times a million grid points.

In Table 1, we show the results for equivalent code written in Matlab (R2017a), standard

Loop size	Matlab	C code	GPU	Speed up
1	.0045 s	.0097 s	.0001 s	45x
100	.380 s	.588 s	.0005 s	760x
500	1.90 s	2.93 s	.0019 s	1000x
1000	3.80 s	5.83 s	.0038 s	1000x
5000	19.0 s	29.2 s	.0124 s	1500x
10000	38.0 s	58.2 s	.8200 s	46x

Table 1. Initial comparisons of speeds, unfortunately misleading.

C code, and running on the AMD GPU. Except for the first and last entries, it looks like there is a speed up of a factor of 1000 or more for the GPU code. It is worth noting that the C code is running about 30% more slowly than the Matlab code. We did optimize the Matlab code by using its built-in convolution routine to do the discrete Laplacian. We did not do any explicit optimization of the C code in this first test.

The sudden jump in the timing numbers for the GPU as we go from 5,000 to 10,000 loops hints that there might be a problem with the built-in system timing subroutine. We redid the timing, for larger loops, where we could compare the system execution time (as measured by the system subroutine) with timing on a hand-held stop watch. Figure 1 shows the plots, and we note the slope of the two data lines are nearly identical. This suggests the slope would be a good measure of the actual "execution time per iteration," and that we should be measuring loops with sizes in the tens of thousands.

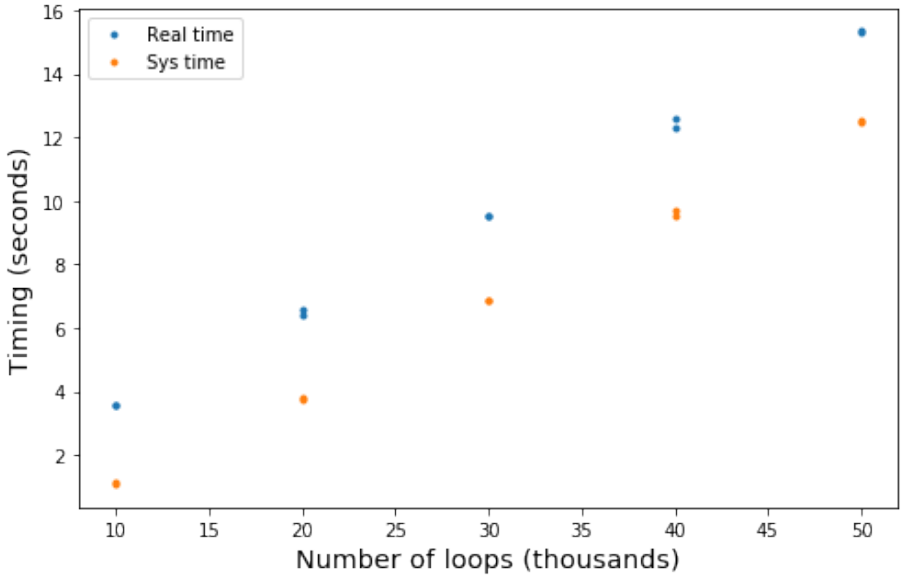

FIG. 1. Plot of execution time versus 1000s of loops. By hand-timing, and system time.

With this information in hand, Table 2 reports the results with larger numbers of iterations, and a check on the slope. Here, we also used a more recent version of Matlab (R2018a) and an optimizing compiler for C to get a more fair evaluation of the Matlab-to-GPU comparison. We also report the times on the AMD GPU as well as a faster NVidia Xp GPU card installed in a Windows PC. Multiple runs were done to test for stability.

Loop size	Matlab - R2018a	C - optimized	AMD	Nvidia	Speed up
20000	68 s	42.69 s	6.259 s	2.263 s	30x
40000	137 s	85.37 s	11.919 s	4.285 s	31x
60000	206 s	129.57 s	17.957 s	6.725 s	31x
80000	275 s	180.82 s	23.844 s	8.873 s	31x
100000	340 s	229.22 s	29.514 s	11.012 s	31x
slope	3.41 ms	2.34 ms	.306 ms	.109 ms	31x

Table 2. Speed comparison, optimized code and timing.

Overall, we see a consistent speed up by a factor of about 30.[3]

The point of this report, though, is to show how we got to these results in a Jupyter notebook. This is covered in the next section.

GPU SPEEDUP - DETAILS

Complete details of the code and data collection has been posted on GitHub[4] in the form of a Jupyter notebook. The interested readers can also launch a demo themselves using the Jupyter "MyBinder" service by clicking on the link in the footnote.[5]

The key idea is to make use of the GPU in an easy-to-implement manner via the software API known as WebGL. GPUs are optimized to work on images and frame buffers which are essentially 2D grids of data points that eventually will be displayed on the screen. (GPUs can also do 3D grids.) These grids are called "textures" or buffers, and as shown in Figure 2, the GPU takes the data in the input buffer, runs it through some code called the "fragment shader," and then sends the result to the output buffer. The key is that the code in the 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 pixels at a time.

Remarkably, the frame buffers or textures look exactly like a spatial grid where we can apply finite difference code to solve a numerical partial differential equation. While the GPU can allow us to use more than one grid as an input to the computation, the output grid has to be a separate, independent grid. To implement our time-stepping finite difference code, we just have to swap buffers from input to output and back again to keep the corresponding grids independent.

Our goal is to use whatever video card and GPU that is available on our personal computer or laptop. Modern web browsers are GPU-aware, so if there is a GPU available, the browser will make use of this. We use this to our advantage by access the WebGL API to request GPU services via the browser where our Jupyter Notebook is running. We also avoid needing to use specialized code such as CUDA to access the GPU in its own language.

[3] A quick first test with the Nvidia video card showed a time of 3.664 seconds for 50,000 iterations, or a rate of .0732 ms per iteration. This is a speed-up by a factor of 46. But we were never able to replicate this.

[4] `https://github.com/mlamoureux/WebGL_sample`

[5] `https://mybinder.org/v2/gh/mlamoureux/WebGL_sample/master?filepath=WebGLwave.ipynb`

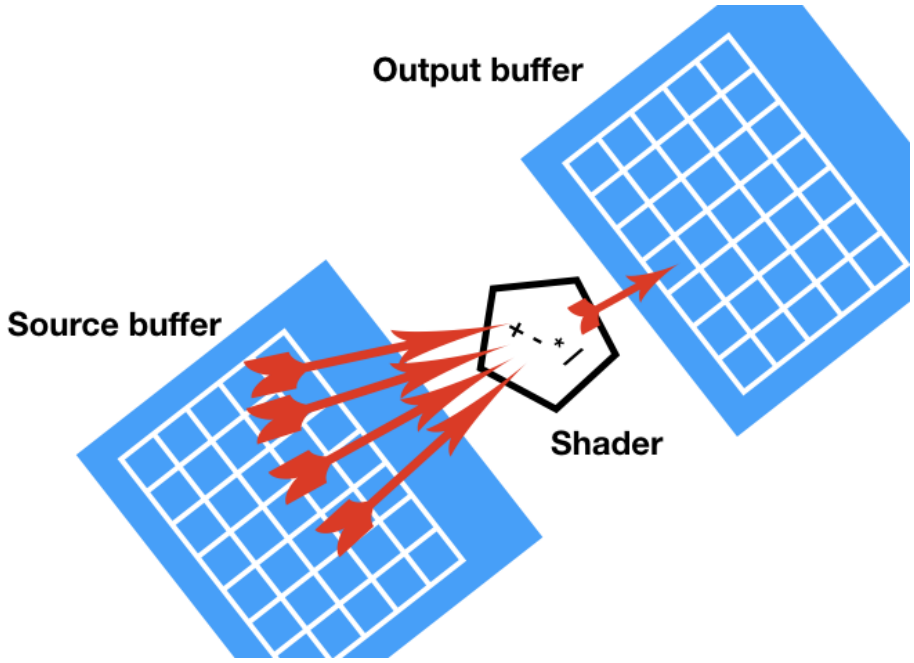

FIG. 2. Result of finite difference code in WebGL.

```
void main() {
   float  psi;
   psi     = texture2D(waveFunction, vTextureCoord).r;
   gl_FragColor = max(0., psi)*vec4(0.,0.,1.,1.)
      + max(0.,-psi)*vec4(1.,0.,0.,1.);
}
```

FIG. 3. GPU shader code to display wavefield.

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 – so in this example, we use some pre-made utilities developed by Vizit Solutions.[6] The main piece of code we use from them is a Javascript file called `GPGPUtilities.js` as a collection of General Purpose Graphics Processing Utilities. This code sets up grids of appropriate sizes, allows us to stuff them with data, define the shader code, and run the code as appropriate.

We also have to write two files with Javascript: one to do the finite difference code and one to display the results.

The display code is straightforward. As shown in Figure 3, it grabs the data in the 2D `waveFunction` at a grid point given by `vTextureCoord` and stores it in the variable `psi`. If `psi` is positive, the output pixel (`gl_FragColor`) is set to the 4-vector (0,0,1,1), which is Blue, scaled by the value in `psi`. If `psi` is negative, the output pixel is set to the 4-vector (1,0,0,1), which is Red. This is very simple code, but fast because any one of the thousands of GPU processors can run it on any output pixel.

[6] `http://www.vizitsolutions.com`

```
void main( ) { ... variable def's omitted ...
   psi = texture2D(waveFunction, vTextureCoord);
   gl_FragColor.r = 2.0*psi.r
      - texture2D(oldWaveFunction,vTextureCoord).r
      + (psi.g)*dt*dt*/(dx*dx)*(
      +  texture2D(waveFunction, vTextureCoord+dss).r
      +  texture2D(waveFunction, vTextureCoord-dss).r
      +  texture2D(waveFunction, vTextureCoord+dtt).r
      +  texture2D(waveFunction, vTextureCoord-dtt).r
      -  4.0*value.r);
   gl_FragColor.g = psi.g;
}
```

FIG. 4. GPU shader code for FD time-stepping waveforms.

The finite difference code is a bit more complicated and is shown in Figure 4. The key idea is there are two input textures corresponding to the waveform values at current time t (`waveFunction`) and at the previous time step $t - \Delta t$ (`oldWavefunction`). The variable `psi` recovers the current waveform values, and then the output (`gl_FragColor`) is a linear combination of the current waveform and the past waveform plus the discrete Laplacian of the current waveform. This Laplacian is computed by evaluating the waveform at shifted places on the texture coordinates (`vTextureCoord` $\pm$ `dds`, $\pm$ `ddt`).

One important point in this shader code is that the output of `texture2D` is a 4-vector, corresponding to the three colours Red, Green, Blue and the alpha channel. In the red channel is the amplitude of the waveform (referenced as `psi.r`) and in the green channel is the variable velocity field (`psi.g`). This velocity information is used in the time stepping, so the simulation can handle non-uniform velocities. Although the data in these frame buffers seem to be associated with colours, they are just floating point memory cells that we can freely use in our computations.

These two pieces of code, and the supporting functions, are stored in files `WavePlot.js` and `WaveEqn.js`. To access them in a Jupyter notebook, we write some simple HTML code in a cell that loads in these files as well as additional Javascript code to set up the display in the output cell. The start of the HTML code is shown in Figure 5. At the top of this code, we see the three Javascript files are loaded, `GPGPUtilities.js, WaveEqn.js, WavePlot.js`. Near the bottom of this figure is the `<script>` which is the start of some more Javascript code that calls up the appropriate pieces to launch our GPU code. The code is this cell in written to a file under the exciting name `myFile1.html`.

Finally, we can run the code in a Jupyter cell with the following HTML code:

```
<iframe src="myFile1.html" width=1000 height=1000>
```

This starts an animation that runs very smoothly and quickly, thanks to the GPU. A sample frame of the output is shown in Figure 6. This shows the result of an initial bump function propagating as a circle until it hits a horizontal layer of a different velocity (slower medium) where a refraction and reflection occur. Note the change in polarity of the reflection.

In Figure 7 we see the result of a different initial condition where we have nearly planar

```
In [12]: %%writefile myFile1.html
<!DOCTYPE html>
<html><head id="Barebones">
        <meta charset="UTF-8" />
        <meta name="viewport" content="width=device-width, initial-scale=1.0" />
        <script src="GPGPUtility.js"></script>
        <script src="WaveLeapfrog.js"></script>
        <script src="WavePlot.js"></script>
        <title>WebGLwave example for CREWES</title>
        </head>
        <body>
        <div class="content">
            <h1>WebGLwave example for CREWES</h1>


            <figure class="center" id="results">
                <figcaption>
                    You should see a waveform traveling here.
                </figcaption>
            </figure>
        </div>

        <br class="clear" />
        <script>
            "use strict";

            /** The canvas onto which we render the wave function & potential */
            var canvas;
```

FIG. 5. HTML code in a Jupyter cell, to start the FD code. .

waves propagating and reflecting off the interfaces.

TIMING ISSUES

We use the Javascript console to record timing information about our GPU usage. In Figure 8, we step through 1000 calls to the time stepper algorithm `wave.timestep()` on the GPU and output the resulting time usage with a call to the Javascript console. By changing the number of steps in the loop, we can capture the GPU usage for various numbers of trials. Note that each time step computes the results on a 1000x1000 grid.

As a comparison, we can show our C code and Matlab code that we run for comparable timing. Figure 9 is the C code and Figure 10 is the Matlab code. The timing results are as reported earlier in this paper.

JUPYTER NOTEBOOK

A Jupyter notebook is a computer document that can host and display formated text, graphics, data and live code in a single computer file. This allows users to assemble all the key parts of their research into a single document that explains the work (in text and formulas), does the data analysis and computation (with the actual data and code), and display the results in a professional manner (with formated text, and advanced graphical displays).

Each notebook is a collection of editable cells that can contain either formated text, graphics commands, or live code. Because Jupyter notebooks use Markdown as its standard typesetting language, it is easy and fast to create formated text with headers, sections, enumerated lists, and even rich mathematical formulas using Latex commands. As Jupyter is based on modern web standards, each cell can be as complex as a full web page – indeed, HTML coding is possible in any cell, to produce elegant-looking sections that are informative and attractive.

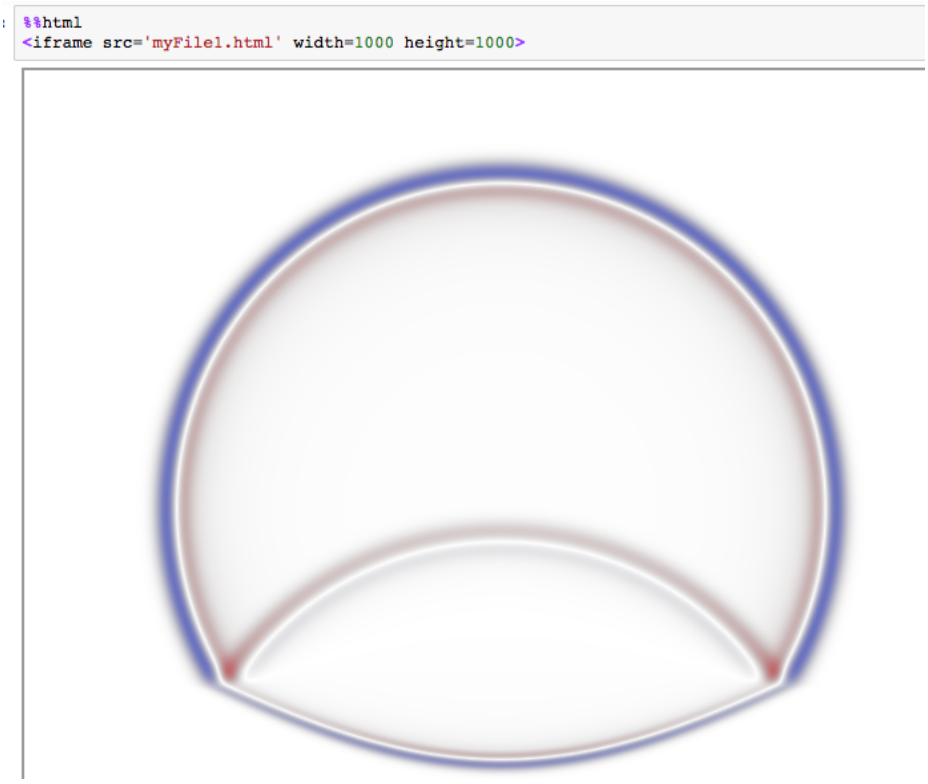

FIG. 6. Result of finite difference code in WebGL.

Jupyter notebooks are typically displayed in a browser such as Chrome, Firefox, Safari, or Internet Explorer. All the edits and code operations are initiated from within the browser, giving a familiar feel to the manipulation of the Notebook document. Interestingly, the Notebook itself may be hosted either on the user's local computer or out somewhere on the cloud. Indeed, the Notebook may be hosted from many different places, making it easy to access and share in many places, with many people.

A sample Notebook is shown in Figure 11. At the top of the Figure, we see the browser interface with tabs and an address bar. Just below is a menu bar of controls for the Notebook, followed by a large display of the active cells. The top cell includes a header (**Math formulaton of FD wave equation**) with some text and mathematical formulas. The bottom cell contains three lines of Python code.

In Figure 12, we see a cell with Python code and the resulting graphical output. In this case, the image is the result of an ODE calculation (as described in Stewart (2014)) which computes the trajectory of the Lorenz attractor, famously known as the "butterfly effect" in meterology.

JUPYTER SERVERS AND HUBS

A Jupyter Notebook is just a file – it has to run on a software service known as a Jupyter Hub. This Hub or server can be hosted on your own personal computer or can be hosted on a server on the cloud.

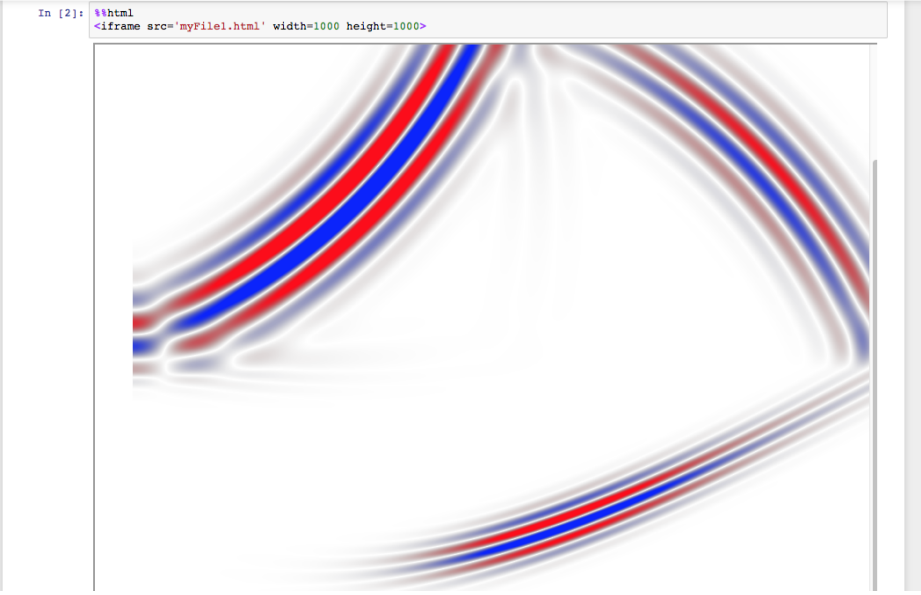

FIG. 7. Result of finite difference code in WebGL.

```
startGPU = window.performance.now();
for (var i = 0; i < 1000; ++i) {
result = wave.timestep();
}
endGPU = window.performance.now();
gpuTime = endGPU - startGPU;
console.log(‘GPU - 1000 steps: ${gpuTime}ms‘);
```

FIG. 8. Javascript timing loop.

To install the Jupyter Hub on your computer, follow the instructions at `http://jupyter.org/`. A very successful method is to install the Jupyter package from Anaconda, located at `https://www.anaconda.com/download/`.

However, it is often very convenient to use an online service. The Pacific Institute for the Mathematical Sciences (PIMS), Cybera, and Compute Canada provide a Jupyter service at `https://pims.syzygy.ca`. All you need to access this service is a Google account – it is free, and there is no software to install. The University of Calgary and many other universities in Canada also provide access to this service through a university login such as `https://ucalgary.syzygy.ca`.

To learn more about the Syzygy service and how to get started using Juptyer notebooks, access the eBook `http://intro.syzygy.ca`, which was written by the first author (Lamoureux (2016)).

The Syzygy server resides in the cloud and gives each user access to a fast, virtual two-core CPU and about a gigabyte of RAM.

Commercial servers are also available: AWS, Azure, and Google Cloud all provide (paid) services that allow users to run a Jupyter server on their cloud computing resources. This is particularly useful when the users' computational needs expand beyond what can

```
int main() {
float a[1000][1000];
float b[1000][1000];
int numLoops = 100;

timestamp_t t0 = get_timestamp();
for (int k=0; k<numLoops; k++)
for (int i = 1; i<999; i++)
for (int j=1; j<999; j++)
b[i][j] = a[i][j] + .1*(a[i-1][j]+a[i+1][j]+
a[i][j-1]+a[i][j+1]-4.0*a[i][j]) ;
timestamp_t t1 = get_timestamp();
double secs = (t1 - t0) / 1000000.0L;
printf("NumLoops is %d, Elapsed time is %f \n",numLoops,secs);
return 0;
}
```

FIG. 9. C-code timing loop.

```
u = zeros(1000, 1000);
v = zeros(1000, 1000);
lapp = [0,1,0;1,-4,1;0,1,0];
for j = [1,10,100,500,1000,5000,10000]
tic
for i = 1:j
v = u + 2*u + conv2(u,lapp,'same');
end
x = toc;
[j,x]
end
```

FIG. 10. Matlab timing loop.

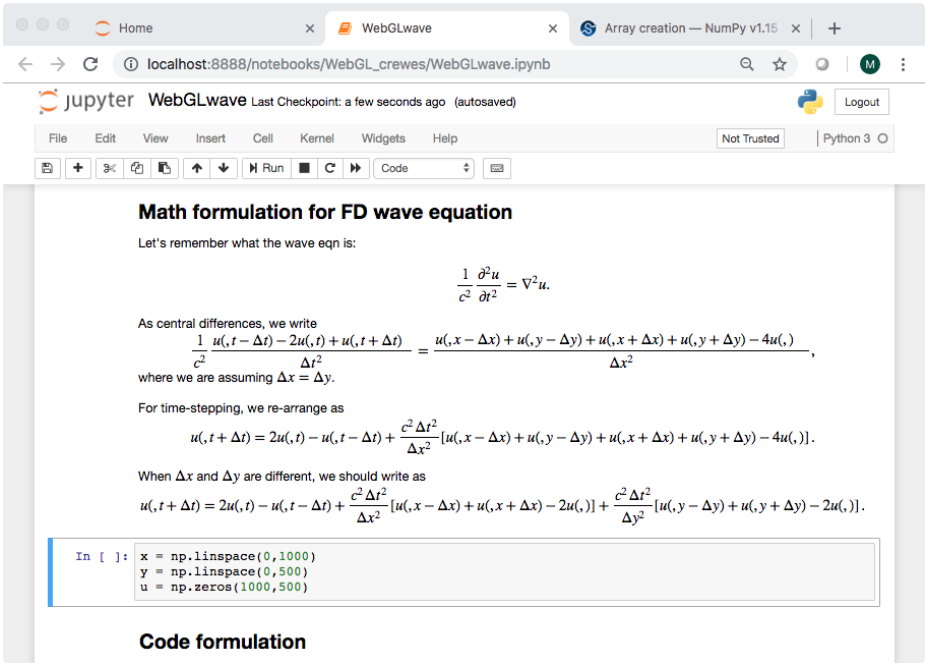

FIG. 11. A sample Jupyter notebook.

be done on their personal computer or on the free services available through other sources. By paying more, the user has easy access to more hardware resources in the cloud.

It is worth noting that some of these commercial services offer free Jupyter servers to get one started. For instance, Microsoft Azure has a free Jupyter notebook server here: `https://notebooks.azure.com/`.

CODING IN A JUPYTER NOTEBOOK

A Jupyter notebook is very flexible in terms of what computing languages can be used in the notebook. This flexibility is possible because Jupyter supports a variety of kernels, which are interfaces between the notebook and the computing service on the Jupyter Hub. Typically, Jupyter Hubs support three standard languages: Julia, Python and R (a statistical programming language), and others may be installed by system administrators or the user.

In Figure 12, a sample of Python code with output is given. In this case, we are using standard packages including Numerical Python, Scientific Python and MatPlotLib to work with numerical arrays (np.linspace), solve ordinary differential equations (odeint), and plot the results (ax.plot). Within the Jupyter notebook, the user has access to all the usual Python tools as well as tools for other supported language.

Kernels are available for many other languages, including Matlab™, Mathematica™, Octave, C, Fortran, Haskell, Java, and many others. However, it is necessary to install these kernels on the Jupyter Hub service which is either on the user's personal computer or in the cloud service – and respect the necessary licensing protocols. A complete list of available kernels is given here: `https://github.com/jupyter/jupyter/wiki/Jupyter-kernels`.

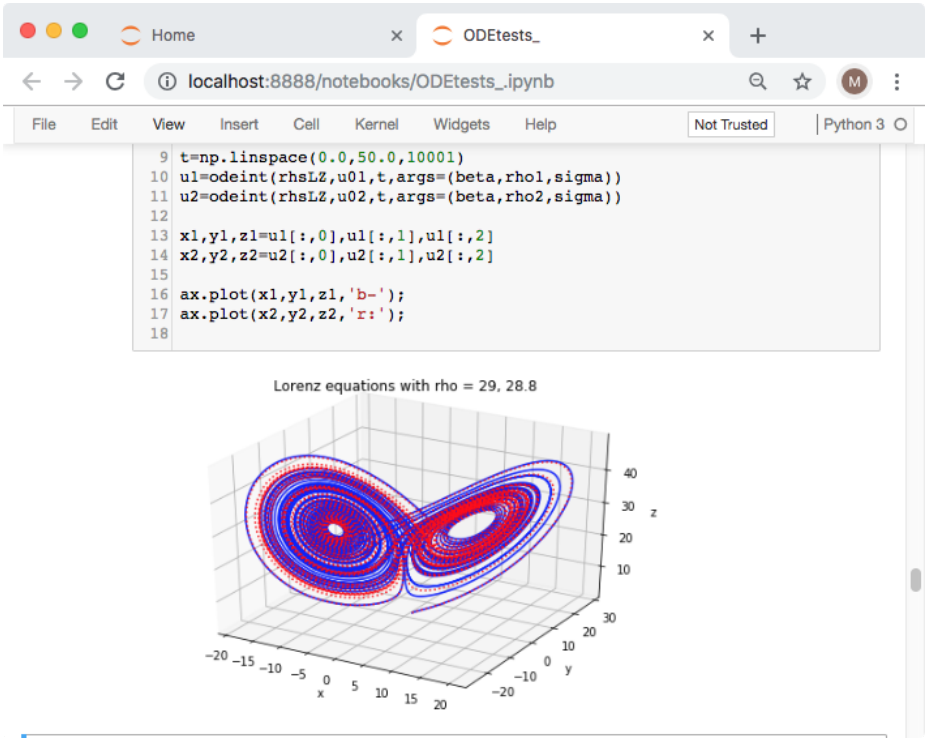

FIG. 12. A sample Jupyter notebook.

Interestingly, because Jupyter is built on a UNIX environment, one has access to system commands including C compilers (and other installed system tools) using Shell/Bash commands within the Jupyter environment. For instance. Figure 13 shows a 3-cell method to write, compile, and run the “Hello, World" example in C. This is particularly useful if the user needs to compile existing code and run it within the Jupyter environment.

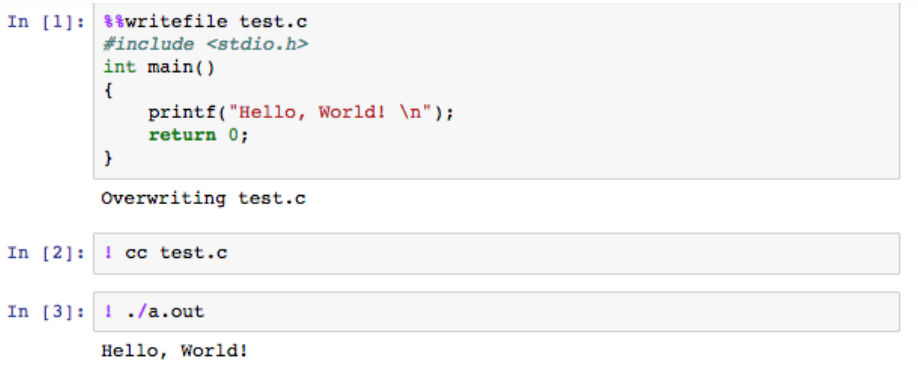

FIG. 13. Three cells to write, compile, and run C code in a Notebook.

Also available within every Jupyter notebook are “magic” commands that allow the use of HTML, Javascript, Perl, Ruby, or SVG code directly in a Notebook cell without installing any additional kernels. This is particularly useful when creating complex documents that have many graphical elements, or require animation and interaction with the user.

For instance, D3 (`https://d3js.org/`) is a very useful Javascript package that al-

lows the display of complex graphical data. The abbreviation D3 stands for Data Driven Documents, and the following description from its webpage summarizes its utility:

> D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG, and CSS. D3's emphasis on web standards gives you the full capabilities of modern browsers without tying yourself to a proprietary framework, combining powerful visualization components and a data-driven approach to DOM manipulation.

D3 is often used in commercial publications, including newspapers such as the New York Times, and is also the basis for dynamic, interactive animations. Again, note that a Jypyter Notebook gives the user full access to these tools via HTML and Javascript code that the user can create and develop.

RUNNING MATLAB ON JUPYTER

As many CREWES researchers use Matlab, it is useful to know that you can run Matlab within the Jupyter environment on your own PC. There are basically three steps needed to make this happen:

1. Install Jupyter on your PC. See this: `http://jupyter.org/install`
2. Install the Matlab-to-Python Engine on your PC. See this: `https://www.mathworks.com/help/matlab/matlab_external/install-the-matlab-engine-for-python.html`
3. Install the Matlab kernel into your operating system. See this: `https://github.com/calysto/matlab_kernel`

This may sound like a lot. But it is actually very simple. On the Mac OS, open a Terminal and enter the following sets of commands, corresponding to the installation steps above:

1. ```
   python3 -m pip install --upgrade pip
   python3 -m pip install jupyter
   ```
2. ```
   cd /Applications/MATLAB_R2018a.app/extern/engines/python
   python setup.py install
   ```
3. ```
   pip install matlab_kernel
   ```

You might also prefer to use the Anaconda distribution[7] to install Python3 with Jupyter.

[7] `https://www.anaconda.com/download`
   ```

For a Windows or Unix system, the steps are similar. Just read the installation instructions as described in the weblinks above. In Figure 14, we see a Matlab plot, using code from the second author, executed in a Jupyter notebook.[8] It is really that simple. Notice the little "Matlab" label in the top right corner of the figure – this tells you the Notebook is indeed running Matlab.

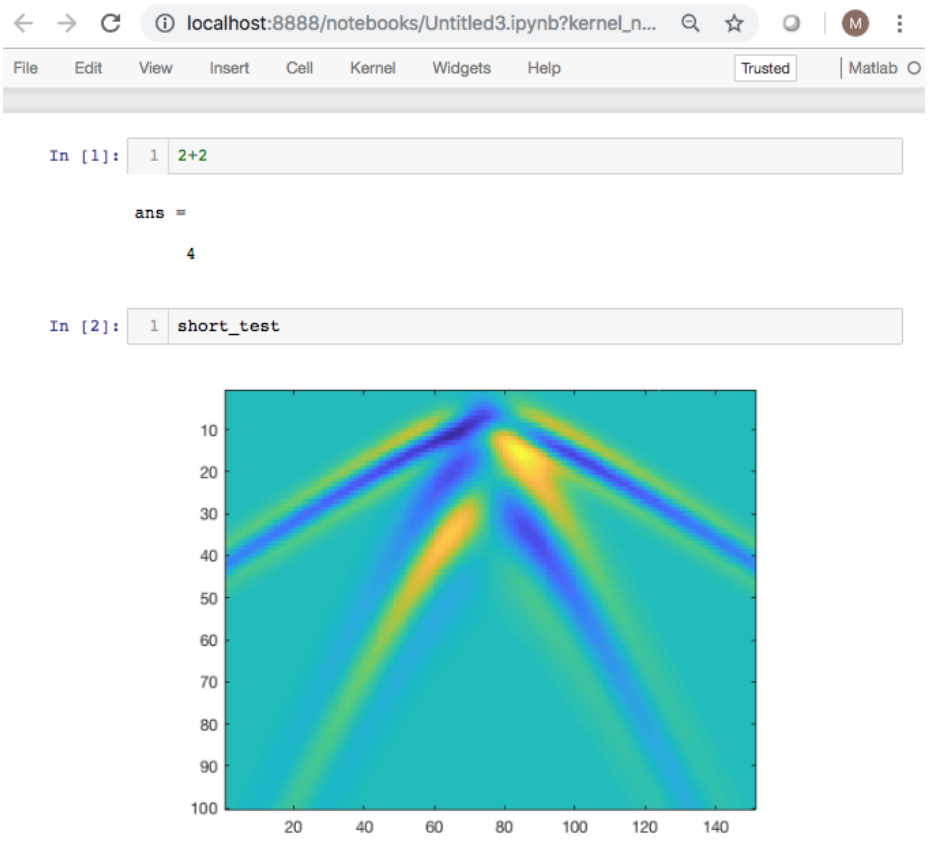

FIG. 14. Matlab running in a Jupyter notebook.

ONLINE SERVICES

As mentioned above, there are many online services available to run your Jupyter notebooks on a Jupyter server. Some free ones that are available to the CREWES community include:

- `https://ucalgary.syzygy.ca` - a free service available to U of C staff and students
- `https://pims.syzygy.ca` - a free service, to anyone with a Google account
- `http://jupyter.org/try` - good for quick tests, demos
- `https://notebooks.azure.com` - a free, commercial service from Microsoft

[8]See the article on modeling DAS in this CREWES report.

The University of Calgary and PIMS options are supported by a partnership between the Pacific Institute for the Mathematical Sciences, Compute Canada, and Cybera (Alberta). These are full service options that allow users to create and store their own research documents on Canadian-hosted cloud computing services.

SHARING RESEARCH VIA JUPYTER NOTEBOOKS

A Jupyter notebook is just a computer file, so it is easy to email to any colleague who can then run the file on their own Jupyter server.

Of course, most researchers share collections of files. An easy way to do this is to post one's Jupyter notebooks (and related data and source files) on the GitHub server `https://github.com`. This is a online-accessible repository for code, data, and more, that is universally used by many programmers and researchers. The first author's GitHub account is available here: `https://github.com/mlamoureux`. On it, you will find complete lecture notes for courses, files for eBooks, research results, and of course research code in Jupyter notebook format.

The Notebook Git Puller service automates the sharing of notebooks. This is a Jupyter extension that allows one to send a link to a colleague which will automatically clone a copy of the relevant Git repo and launch the Notebook in the colleague's own personal Jupyter server. The colleague can then use the notebook freely, making edits and changes all as part of a collaboration or as a new work.

Information on using the Puller is here: `http://intro.syzygy.ca/teaching/`. You can also read more about the extension here:
`https://github.com/jupyterhub/nbgitpuller`

The Notebook Binder service is a free service that launches a live copy of your Notebook for your colleagues to use themselves on their own browser. MyBinder (`https://mybinder.org/`) launches its own virtual Jupyter server to host the notebook, so your colleague does not even need an account to get the notebook up and running. It is a great way to share and display results. However, MyBinder does not allow the new user to save changes to the notebook, so it is not a full-service Jupyter Hub.

OPEN SOURCE SOFTWARE USE

There is a large community of scientists, researchers, and programmers who contribute algorithms, methods, and codes for free use by anyone. Unlike some classic references such as "Numerical Recipes in C" by Press et al. (2007), the open source community provides unrestricted licenses for use of their code by any purpose, by anyone. An interesting example of open source efforts is the GNU Scientific Library "which provides a wide range of mathematical routines such as random number generators, special functions, and least-squares fitting."[9] We make good use of the sources discussed in the text by Stewart (2014), which is particularly useful for researchers programming in Python.

[9] `http://www.gnu.org/software/gsl/`

With Jupyter notebooks, one quickly gets used to borrowing open source code and ideas that are floating around on GitHub and other repositories. Many of the foundational computational tools have been encapsulated into formally-supported packages such as Numerical Python (Numpy) and Scientific Python (SciPy). The code in these packages is of the highest quality. The numerical algorithms in Scientific Python use the BLAS, LINPACK, and LAPACK software (from `http:\netlib.or`) to provide state of the art numerical methods for linear algebra and other numerical methods. Our earlier example of the Lorenz attractor used the SciPy function `odeint` to compute this solution – this is one of the best pieces of ODE code available which originated at the Lawrence Livermore Labs in California.

It is possible to contribute to this community as well by creating tool boxes of open source code that is available publicly. The works released by the Seismic Laboratory for Imaging and Modelling[10] at the University of British Columbia is an excellent example of such open source contributions in our research community. The CREWES[11] toolbox from the University of Calgary is another example, using Matlab as its engine. The Signal Analysis and Imaging Group[12] at the University of Alberta uses Julia tools for Jupyter notebooks.

CAUTIONS ON OPEN SOURCE CODE

Some caution with open source resources is advised, because not all of them are easy-to-use or reliable. For instance, our initial GPU experiments on FD code for the wave equation were not successful because we were using code that was not suitable to the task – or perhaps just poorly documented for our purposes. In particular, we spent a lot of effort trying to understand and use the WebGL utilities developed at `http://gpu.rocks/`. We had some success using their GPU toolbox `gpu.js`, in the sense that our developed code would run and kind-of looked like a wave propagating, but with huge errors.

For instance, Figure 15 shows a frame snapshot of what is supposed to be a circular wave propagating outwards. The initial central circle is there, but there are weird "ghosts" appearing that contaminate the image. These ghosts then propagate quickly like a wave which makes one think the finite difference code is sort-of working, but not quite. After spending many hours trying to debug the code, we threw in the towel and went to other code sources. Our successful project is described in the section above, using the WebGL utilities from another source.

SUMMARY

A Jupyter notebook is a valuable means for advancing our research, providing a tool for computation, communication, and collaboration. In this article, we discussed our experience in using the notebooks for teaching and research with a particular example using a Jupyter notebook to access a video card GPU for a 30x speedup of our acoustic wave simulation. We presented information on resources available to the reader to begin making

[10] `https://www.slim.eos.ubc.ca/software-documentation`
[11] `https://www.crewes.org/ResearchLinks/FreeSoftware`
[12] `https://github.com/SeismicJulia/Seismic.jl`

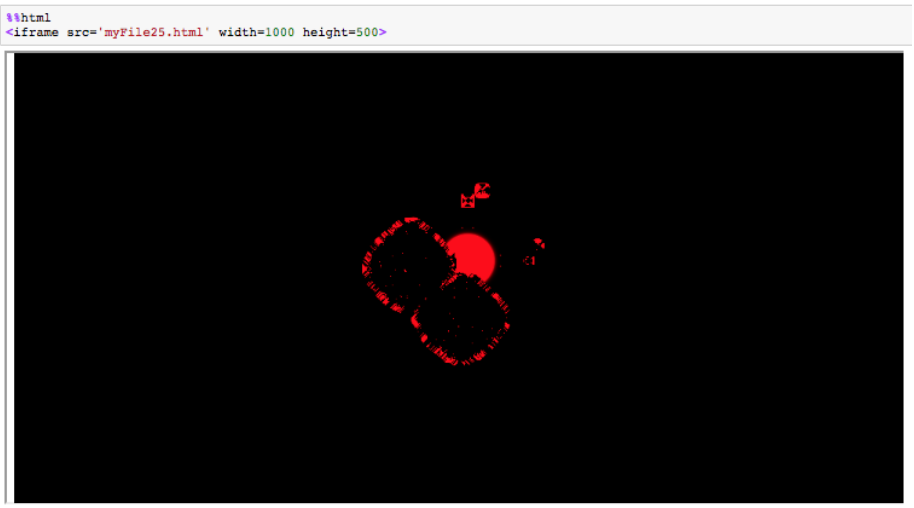

FIG. 15. Breakdown in code implementation.

the transition to Juptyer notebooks for their own research.

FUTURE WORK

First steps are to implement code that transfers data from a Python or Matlab environment in a Jupyter notebook into the Javascript environment that run the GPU code. Currently, all our code is written in Javascript which is not the friendliest language to use. We want to seamlessly move our usual Python or Matlab data onto the GPU environment, bypassing the complexities of Javascript.

Next, we will implement absorbing boundary conditions and perfectly matched layers to make acoustic simulation more suitable for our seismic imaging applications.

Moving to three dimensions and elastic wave equations are the obvious next steps that would be suitable for the GPU speedup.

Generally speaking, we would like to provide the basic code APIs that allows the seismic researcher to access any GPU resources available on a system's video card without the user having to worry about a lot of technical details. So, future steps include creating and testing these API in real seismic explorations.

ACKNOWLEDGMENTS

We thank the sponsors of CREWES for their support. We gratefully acknowledge the support of NVIDIA Corporation with the donation of the Titan Xp GPU used in this research. We also gratefully acknowledge support from NSERC through the grant CRDPJ 461179-13 and a Discovery Grant for the first author.

REFERENCES

Lamoureux, M. P., 2016, Introduction to Syzygy: Pacific Institute for the Mathematical Sciences.

MathWorks, 2018, Matlab™ programming language.

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.

Shreiner, D., Woo, M., Neider, J., and Davis, T., 2004, OpenGL Programming Guide, Fourth Edition: Addison-Wesley.

Stewart, J. M., 2014, Python for Scientists: Cambridge University Press.