EE147 FINAL PROJECT REPORT

OPTIMIZING THE VISUALIZATION OF PERIDYNAMIC SIMULATIONS

September 26, 2018

Joel Borja
Stefany Cruz
Brandon Rivera
University of California Riverside

September 26, 2018

Contents

Notes 2
Project Idea / Overview 2
Implementation details 3
How is the GPU used to accelerate the application? 4
Code Documentation 4
Evaluation/Results 5
Problems 5
References 6
Images 7

NOTES

The figures can be found in the **Images** section of this report.

PROJECT IDEA / OVERVIEW

While observing that Dr. Wong gave us the option of choosing to implement a molecular dynamic application for our project, our group thought that it would be brilliant if we could make a visualization of this type of simulation. Our idea was proposed by Stefany after she recalled that her boyfriend, Jonathan Berjikian, was doing research with peridynamics simulations of glass fracture using MATLAB at UCLA. The simulations of the visual representations of peridynamic glass fracture using MATLAB usually takes up two to three hours. Stefany figured it would be an excellent problem a GPU can solve. She proposed the idea and we all agreed to carry on the project. In our simulation we show the glass damage evolution under impact with a high-velocity projectile. In other words, a bullet is hitting a piece of glass. The peridynamic simulation uncovers a fascinating time-evolution of damage and the dynamic interaction between the stress waves, propagating cracks, interfaces, and bending deformations, in 3D. We figured that if we could optimized this simulation using a GPU and were successful, we could document our findings to help others who seek to make the visualizations of their research faster using MATLAB.

IMPLEMENTATION DETAILS

We first began with MATLAB code that used the CPU. In **Figure 1**, the initial **for loop** is set between 300-500 frames.The reason is because in the video, the glass does not begin to fracture until the 300^{th} frame, and stops breaking at the 500^{th} frame. We hypothesized that the block of code highlighted in red in **Figure 1** was going to be sluggish due to the size of the variable atoms, which is 104,000 atoms in size.

In order to get this code onto the GPU, we first had to write a kernel. It can be seen in **Figure 2**. This will later be used in the kernel function for the GPU. Using the command **nvcc-ptx** glass parallel, we created a PTX file as seen in **Figure 3**. This will also be implemented in the kernel function for the GPU.

The final step was creating the MATLAB code that was going to be run on the GPU. The function highlighted in red in **Figure 4** is taking in the kernel and the PTX seen above, to create a new kernel. The code in **Figure 5** is the optimized version of the block of coded highlighted in red in **Figure 1**.

HOW IS THE GPU USED TO ACCELERATE THE APPLICATION?

We took lines 18 through 21 of code as seen in **Figure 6** and parallelize them to optimize their performance seen in **Figure 7**. The red numbers are the total amount of time that line of code took to run, the blue numbers that are not underlined to the left is the total amount of calls for each line. We were able to successfully optimize that part of the code.

CODE DOCUMENTATION

There is a CPU and GPU version of the MATLAB code. For the CPU and GPU section of the code we will navigate to the directory where the following files ***data_new.m, CPU_Sort_Data.m, GPU_Sort_Data.m, GlassKernel.cu, GlassKernel.ptx*** are located.We need to make sure that all of the listed files above are in the directory. We start by loading ***CPU_Sort_Data.m*** by clicking on it in the directory of MATLAB. In the code be sure to set the appropriate directory under the **change directory** comment. For example ***cd('/home/folder/EE147_PROJECT')***. We will do this so we can choose a directory for the output. In line 26 of ***CPU_Sort_Data.m*** the last two numbers of pos_h need to be changed according to the resolution of the operating system. In our case we chose the resolution of the BENDER server. We chose 1362 x 687 resolution. After the code runs, the video produced will be output into the selected directory. ***CPU_Sort_Data.m*** requires ***data_new.mat*** but that is already called in ***CPU_Sort_Data.m***. After hitting run and waiting about three hours there will be video that is output into the directory and in the terminal there will be a return time of how long it took for ***CPU_Sort_Data.m*** to be executed.

In order to run ***GPU_Sort_Data.m***, we start by loading ***GPU_Sort_Data.m*** by clicking on it in the directory of MATLAB. In the code be sure to set the appropriate directory under the **change directory** comment. For example ***cd('/home/folder/EE147_PROJECT')***. We will do this so we can choose a directory for the output.***GPU_Sort_Data.m*** requires ***data_new.mat*** but that is already called in ***GPU_Sort_Data.m***. ***GPU_Sort_Data.m*** we need to make sure that ***GlassKernel.cu, GlassKernel.ptx*** are in the directory becuase they will be needed for the ***parallel.gpu.CUDAKernel()*** function. In our code we use the function as ***K_Kern = parallel.gpu.CUDAKernel('GlassKernel.ptx','GlassKernel.cu');*** We will keep the resolution the same the CPU. After hitting run and waiting about three hours there will be video that is output into the directory and in the terminal there will be a return time of how long it took for ***CPU_Sort_Data.m*** to be executed.

EVALUATION/RESULTS

Our initial hypothesis was if we optimized a block of code that utilized the most data then we would be able to reduce the amount of time taken to produce the video. We see in **Figure 6** from line 18 through line 22 the lines individually were called about 18 to 20 million times. We see the optimized code in **Figure 6** that lines 31 through 33 were each called 201 times. That is a substantial reduction of calls made.

Through our analysis we determined that CPU executed and created the video faster than the GPU. The reason being is that the GPU requires data to be sent from the host to the device and then processed data is sent back to the host. This transaction requires an ample amount of time. We were able to show this by timing ***CPU_Sort_Data.m*** and ***GPU_Sort_Data.m*** using the tic toc function. For the CPU it took 2 hours 59 min 6.3659 sec to run and for the GPU it took 3 hours 4 min 15.6324 sec.

We found that the culprit that made both of the codes dawdle was the function ***getframe()***. We can see in **Figure 8** that the **getframe()** function is highlighted in red for the CPU and the GPU. We hypothesize that if we were to also optimize this function then maybe we could reduce the amount of time taken to make the video with the GPU. We can see the end result in **figure 9**.

PROBLEMS

One of the problems with this project is that Peridynamic Systems usually takes around 2-3 hours to run. We initially ran the simulation on each of our computers, however each of our computers had different processing power so we were getting different results. We found it best to run it on BENDER to get an consistent results. There was one day where the GPU was down so we lost a day of work. Another problem we ran into is that we had to increase the quota on our engineering accounts from 2GB to 5GB.

REFERENCES

Run CUDA or PTX Code on GPU. (n.d.). Retrieved May/June, 2018, from https://www.mathworks.com/help/distcomp/run-cuda-or-ptx-code-on-gpu.html

Illustrating Three Approaches to GPU Computing: The Mandelbrot Set. (n.d.). Retrieved May/June, 2018, from https://www.mathworks.com/help/distcomp/examples/illustrating-three-approaches-to-gpu-computing-the-mandelbrot-set.html

Ellid, E. (Ed.). (n.d.). *PTX kernel time to run.* Retrieved May/June, 2018, from https://www.mathworks.com/matlabcentral/answers/7511-ptx-kernel-time-to-run

IMAGES

Figure 1: Snippet of code that will be optimized

```
for i = 300:500
    color = damage(:,i);    %damage is the percentenge of the number of bonds broken
for j = 1:atoms
       if color(j,1) < .45      %45 percent is standard accepted value of broken glass
           color(j) = .000001;  %close to zero
       end
end
```

Figure 2: Optimized version of snippet in CUDA code

```
1   #include <stdio.h>
2   #include <stdlib.h>
3
4
5   __global__ void VecAdd(float *color, unsigned int atoms)
6    {
7        int j = threadIdx.x +blockDim.x *blockIdx.x;
8
9       if(j < atoms) //if index is less than 104014
10           {
11              if(color[j] < 0.45)  //if Array is less 45%
12                   {
13                     color[j] = .000001; //then the glass is all shattered
14                   }
15            }
16  }
17
18
```

Figure 3: Snippet of PTX code

```
.version 6.2
.target sm_30
.address_size 64

    // .globl   _Z6VecAddPfj

.visible .entry _Z6VecAddPfj(
    .param .u64 _Z6VecAddPfj_param_0,
    .param .u32 _Z6VecAddPfj_param_1
)
{
    .reg .pred  %p<3>;
    .reg .f32   %f<2>;
    .reg .b32   %r<7>;
    .reg .f64   %fd<2>;
    .reg .b64   %rd<5>;
```

Figure 4: CUDAKernel function

```
%Create the KERNEL
K_Kern = parallel.gpu.CUDAKernel('GlassKernel.ptx','GlassKernel.cu');

%Specify the number of threads
K_Kern.ThreadBlockSize = [K_Kern.MaxThreadsPerBlock, 1, 1];

%Specify the size fo the grid
GridsTotal = ceil(atoms/K_Kern.MaxThreadsPerBlock)+1;
K_Kern.GridSize = [GridsTotal, 1];
```

Figure 5: Calling the GPU array

```
%Call GPU ArrAY
G1 = gpuArray(single(color));
G2 = feval(K_Kern,G1,atoms);
color = double(gather(G2));
```

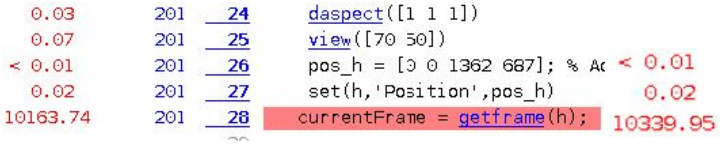
Figure 6: Amount of calls from the CPU code

```
< 0.01         1   16 for i = 300:500
  0.07       201   17     color = damage(:,i);   %damage is the percentenge of the number of bonds broken
< 0.01       201   18  for j = 1:atoms
  1.09  20906814   19         if color(j,1) < .45      %45 percent is standard accepted value of broken glass
  0.89  18026282   20             color(j) = .000001;  %close to zero
  0.83  18026282   21         end
  1.05  20906814   22  end
```

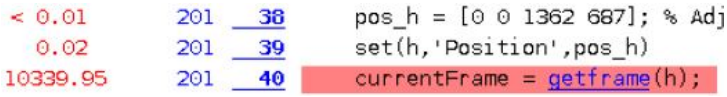
Figure 7: Amount of calls from the GPU code

```
                  30     %Call GPU ArrAY
0.17       201    31     G1 = gpuArray(single(color));
0.03       201    32     G2 = feval(K_Kern,G1,atoms);
0.07       201    33     color = double(gather(G2));
```

Figure 8: getframe() function for CPU and GPU timing in red

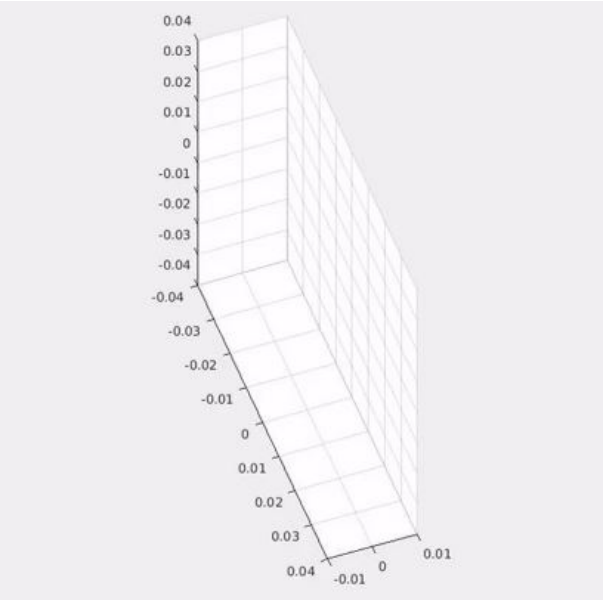


(a) CPU getframe

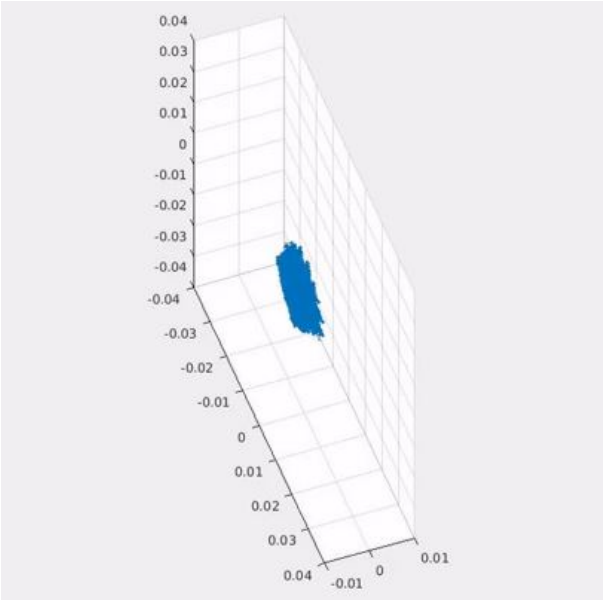


(b) GPU getframe

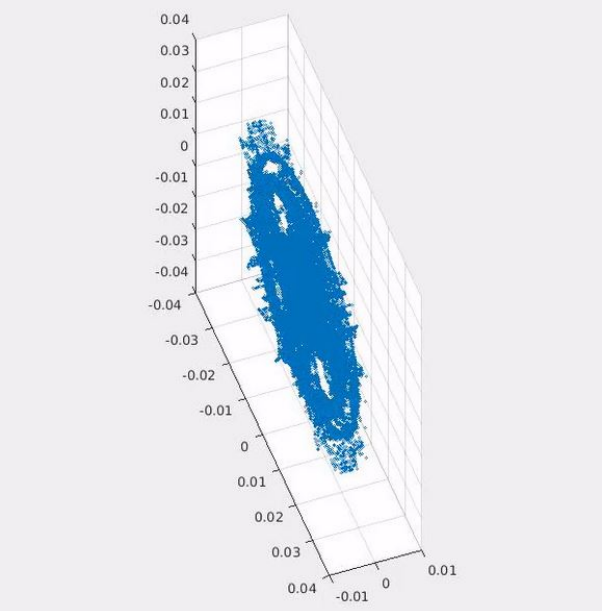
Figure 9: Frames from the 3D video



(a) Initial



(b) Halfway



(c) Final