ECSE 420
Lab 3: Finite Element Music Synthesis

Alex Hale
260672475
Thomas Hillyer
260680811

December 2nd, 2019

Introduction

The objective of this lab was to simulate the result of a drum hit using a grid of finite elements. Three implementations were created: the first in a sequential fashion, the second in a parallelized nature which assigns one GPU thread to each finite element of the grid, and the third in a parallelized nature assigning multiple finite elements to each GPU thread.

All three implementations use Python, and are included alongside this report in a Jupyter notebook entitled “Lab3.ipynb”. The parallel implementations use Numba, a Python package which compiles Python code into a more efficient format, to execute the code using the same CUDA kernel function format that is found in C. The square bracket syntax found in the function calls is analogous to the <<< >>> syntax used for CUDA in C. To execute the code, upload it to Google Colab and run it in a GPU runtime - the Numba setup will be taken care of by the block at the top of the code.

Sequential Implementation

Each finite element in the drum grid is computed one after another, that is to say sequentially, in this implementation. The drum is defined as a matrix at the current time, the previous time, and the second previous time. To simulate the drum hit, a “1” is placed at (2, 2) at the previous instant. It has to be the previous instant as we are simulating time immediately following the hit and the formula for calculating interior elements only takes into account previous timeframes. Constants are also defined for use in the calculations.

```
u = np.zeros([N, N], dtype = float)
u1 = u.copy()
u2 = u.copy()

G = 0.75
rho = 0.5
nu = 0.0002

# hit = "1" at u(2,2)
u1[2][2] = 1
```

Figure 1: sequential parameter definition and drum hit simulation

The interior elements have to be calculated first as the edges and corners (the boundaries are dependent on them). These loops go from 1 to N-2 in both i and j directions because range() is defined as [start, end). This also highlights why the drum hit was simulated at time t-1. The first line of the following image also demonstrates the iteration, all following code occurs inside of the iteration loop. The iteration parameter is passed in to the sequential function.

```
for itera in range(iterations):
    # response 1 to N-2
    for i in range(1, N-1):
        for j in range(1, N-1):
            u[i][j] = ((rho * (u1[i-1][j] + u1[i+1][j] +
                               u1[i][j-1] + u1[i][j+1] -
                               4*u1[i][j])) +
                       (2 * u1[i][j]) - ((1 - nu) * u2[i][j]))
            u[i][j] /= (1 + nu)
```

Figure 2: interior element calculations

The edges are similarly defined by 4 cases. Each edge is a unique case which simply multiplies the interior cell next to it by G (=0.75), simplifying the code.

```
# range 1 to N-2
for x in range(1, N-1):
    ## EDGES
    # left edge = .75 * interior
    u[0][x] = G * u[1][x]
    # right edge
    u[N-1][x] = G * u[N-2][x]
    # top edge
    u[x][0] = G * u[x][1]
    # bottom edge
    u[x][N-1] = G * u[x][N-2]
```

Figure 3: edge element calculations

The corners are not calculated in a loop because there are only 4. They are a multiplication of the edge elements by G (=0.75).

```
## CORNERS
# top left
u[0][0] = G * u[1][0]
# top right
u[N-1][0] = G * u[N-2][0]
# bottom left
u[0][N-1] = G * u[0][N-2]
# bottom right
u[N-1][N-1] = G * u[N-1][N-2]
```

Figure 4: corner element calculations

The final part of the iteration is to “advance” the time by copying the grids to the previous time grids.

```
u2 = u1.copy()
u1 = u.copy()
```

Figure 5: grid copying

The sequential algorithm is simple to implement. Not performing deep copies of the three time slots was an issue that was encountered, this led to all calculations being performed on the same object. It was fixed by using the numpy.copy() function. The other issue encountered was

performing the hit on the current time grid, so that it was never actually included in any calculations and all results in the grid were zero. This was fixed as explained above by performing the hit at time t-1.

The results from this implementation match the provided results from the lab handout, as seen in the following image.

```
0.0
Iteration: 0
Grid:
[[0.         0.         0.37492501 0.28119376]
 [0.         0.         0.49990002 0.37492501]
 [0.37492501 0.49990002 0.         0.        ]
 [0.28119376 0.37492501 0.         0.        ]]

-0.49980001999999923
Iteration: 1
Grid:
[[ 0.28113753  0.37485004  0.28113753  0.21085315]
 [ 0.37485004  0.49980006  0.37485004  0.28113753]
 [ 0.28113753  0.37485004 -0.49980002 -0.37485001]
 [ 0.21085315  0.28113753 -0.37485001 -0.28113751]]

2.9982007234403314e-08
Iteration: 2
Grid:
[[ 4.21621976e-01  5.62162635e-01 -1.63964072e-01 -1.22973054e-01]
 [ 5.62162635e-01  7.49550180e-01 -2.18618763e-01 -1.63964072e-01]
 [-1.63964072e-01 -2.18618763e-01  2.99820072e-08  2.24865054e-08]
 [-1.22973054e-01 -1.63964072e-01  2.24865054e-08  1.68648791e-08]]

0.28102511495301846
Iteration: 3
Grid:
[[-0.08782031 -0.11709375 -0.12294844 -0.09221133]
 [-0.11709375 -0.156125   -0.16393126 -0.12294844]
 [-0.12294844 -0.16393126  0.28102511  0.21076884]
 [-0.09221133 -0.12294844  0.21076884  0.15807663]]
```

Figure 6: sequential implementation results

Parallel Implementations

1:1 element-to-thread ratio

In this implementation, the 4x4 grid from the sequential implementation is converted to a parallel approach. First, the constant parameters are defined, the drum grid is created (as well as storage for previous states of the drum grid), and the drum hit is simulated at time t - 1. The drum hit is simulated in grid_back1 (i.e. u1) at location [2, 2] as specified in the lab instructions.

```
# constant parameters
G = 0.75
n = 0.0002
p = 0.5
gridsize = 4

# grid creation
grid = np.zeros((gridsize, gridsize))
grid_back1 = grid.copy()
grid_back2 = grid.copy()
grid_back1[2][2] = 1   # simulate drum hit at time t-1
```

Figure 7: parameter definition and drum hit simulation

Next, the user has the opportunity to change the number of threads and blocks to be used on the GPU, as well as the number of iterations to simulate (T). Since this is such a small grid, the number of threads and blocks used has little effect on performance - in fact, a smaller number of threads and blocks will introduce less overhead, and therefore better performance. Any positive number of blocks may be used, while threads used must be between 1 and 1024 (inclusive).

Next, the number of iterations specified in the T parameter are performed. During each iteration, the CUDA kernel functions are called three times: the first call is for interior nodes, the second call is for edges, and the third call is for corner nodes. The node formulas in the lab instructions require the kernels to be called in this order - if the grid updates were performed in another order, the results could be inconsistent with the lab requirements.

```
# simulate time progression
for t in range(T):
  # function call (numba performs host -> device -> host memory transfers)
    # required order: interior -> edges -> corners
  drum_interior[numBlocks, threadsPerBlock](grid, grid_back1, grid_back2, n, p)
  drum_edges[numBlocks, threadsPerBlock](grid, G)
  drum_corners[numBlocks, threadsPerBlock](grid, G)
```

Figure 8: Function calls to CUDA kernels

Within each CUDA kernel function, the format is largely similar. First, the unique identifier of each thread is calculated. Since this implementation calls for one thread per grid element, if the unique identifier is higher than 15 (i.e. 4x4 grid elements), no further action is taken on that thread. Otherwise, the unique (i, j) assignment of this thread to one grid element is calculated. Next, a conditional statement is used to determine whether the assigned (i, j) position of this thread is an interior node, an edge, or a corner. Finally, the appropriate formula from the laboratory description is applied to perform the update of the grid element. There is no need to return the updated grid at the end of the function, because Numba handles the device-to-host memory de-allocation and data transfer. As an example of the structure, the interior node update kernel function is displayed in Figure 9.

```
@cuda.jit
def drum_interior(grid, grid_back1, grid_back2, n, p):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < 16:
    j = idx // 4
    i = idx % 4

    if i not in [0, 3] and j not in [0, 3]:
      # interior node
      grid[i][j] = ((p * (grid_back1[i-1][j] + grid_back1[i+1][j] +
                          grid_back1[i][j-1] + grid_back1[i][j+1] -
                          4 * grid_back1[i][j])) +
                    (2 * grid_back1[i][j]) - ((1 - n) * grid_back2[i][j]))
      grid[i][j] /= (1 + n)
```

Figure 9: interior node update kernel function

Once all three kernel functions have been called, the grid is completely updated. The value in the grid at position [N/2, N/2] (in this case position [2, 2]) is printed to the output, then the grid history is updated for the next iteration. The results in Figure 10 match those of the sequential implementation from Figure 6. The first three iterations of the complete grid are also printed to the output so that they can be verified as identical to the output in the lab description. The output is displayed below in Figure 10. Please note that very small values are displayed in exponential form in Figure 10, whereas the lab description rounds them to zeros - they are in the same result.

```
u[2, 2] at iteration 1: 0.0
[[0.         0.         0.37492501 0.28119376]
 [0.         0.         0.49990002 0.37492501]
 [0.37492501 0.49990002 0.         0.        ]
 [0.28119376 0.37492501 0.         0.        ]]

u[2, 2] at iteration 2: -0.49980001999999923
[[ 0.28113753  0.37485004  0.28113753  0.21085315]
 [ 0.37485004  0.49980006  0.37485004  0.28113753]
 [ 0.28113753  0.37485004 -0.49980002 -0.37485001]
 [ 0.21085315  0.28113753 -0.37485001 -0.28113751]]

u[2, 2] at iteration 3: 2.9982007234403314e-08
[[ 4.21621976e-01  5.62162635e-01 -1.63964072e-01 -1.22973054e-01]
 [ 5.62162635e-01  7.49550180e-01 -2.18618763e-01 -1.63964072e-01]
 [-1.63964072e-01 -2.18618763e-01  2.99820072e-08  2.24865054e-08]
 [-1.22973054e-01 -1.63964072e-01  2.24865054e-08  1.68648791e-08]]

u[2, 2] at iteration 4: 0.2810251149530184
u[2, 2] at iteration 5: 0.04682818495502672
u[2, 2] at iteration 6: -0.08778516995465212
u[2, 2] at iteration 7: -0.32181476130245723
u[2, 2] at iteration 8: -0.74136664956306
u[2, 2] at iteration 9: -0.38839948920730266
u[2, 2] at iteration 10: 0.6652254330040654
u[2, 2] at iteration 11: 0.7787261426358246
u[2, 2] at iteration 12: -0.22371278398803726
u[2, 2] at iteration 13: -0.5139857688941788
u[2, 2] at iteration 14: 0.3315687356910853
u[2, 2] at iteration 15: 0.7216416509417185
u[2, 2] at iteration 16: 0.16843621887136329
u[2, 2] at iteration 17: -0.28548607266696807
u[2, 2] at iteration 18: -0.4078320015272444
u[2, 2] at iteration 19: -0.563128497465155
u[2, 2] at iteration 20: -0.2835166141928366
```

```
u[2, 2] at iteration 21: 0.4472320260386376
u[2, 2] at iteration 22: 0.39596970515004054
u[2, 2] at iteration 23: -0.4965266956831 86
u[2, 2] at iteration 24: -0.5256373939774961
u[2, 2] at iteration 25: 0.5760366497416338
u[2, 2] at iteration 26: 0.9926034468428319
u[2, 2] at iteration 27: 0.1908710314349021
u[2, 2] at iteration 28: -0.4025014078396819
u[2, 2] at iteration 29: -0.27847620339368123
u[2, 2] at iteration 30: -0.1845074875605142
u[2, 2] at iteration 31: -0.11426122887022694
u[2, 2] at iteration 32: 0.18264642990383687
u[2, 2] at iteration 33: -0.006543707448174733
u[2, 2] at iteration 34: -0.7084566412480809
u[2, 2] at iteration 35: -0.5236507316698267
u[2, 2] at iteration 36: 0.6606126905303338
u[2, 2] at iteration 37: 1.011229967369607
u[2, 2] at iteration 38: 0.06969105659603876
u[2, 2] at iteration 39: -0.47278567809030636
u[2, 2] at iteration 40: -0.007734276427380817
u[2, 2] at iteration 41: 0.29785640628725146
u[2, 2] at iteration 42: 0.08372459141656625
u[2, 2] at iteration 43: -0.05317669912526922
u[2, 2] at iteration 44: -0.29017494547559003
u[2, 2] at iteration 45: -0.7503066244101299
u[2, 2] at iteration 46: -0.4712290728134359
u[2, 2] at iteration 47: 0.5823734455406359
u[2, 2] at iteration 48: 0.7847448542273362
u[2, 2] at iteration 49: -0.18063500872074204
u[2, 2] at iteration 50: -0.5312129608171092
```

Figure 10: console output for first 50 time-steps of the simulation

n:1 Element-to-Thread Ratio

In this implementation, the parallel implementation from the previous approach is extended to accommodate variable numbers of grid elements being assigned to an individual thread. This implementation improves upon the previous one because in many cases, the penalties of GPU overhead and communication will be too high to warrant assigning one GPU thread to each finite element. This approach also covers the situation where the grid is larger than the number of threads available.

The structure of the implementation is similar to the previous parallel implementation. The constant parameters are identical, except the grid size, which is increased to 512x512. The initial drum hit is therefore registered at location [256, 256] at time t-1. The variable parameters are the same, with the addition of an elementsPerThread parameter, which the user can vary between 1 and *gridsize*^2 (i.e. between one thread per element and one thread for the whole grid, inclusive). Before starting to iterate, the number of threads required to cover the whole grid is calculated as shown in Figure 11.

```
threadsRequired = math.ceil((gridsize * gridsize) / elementsPerThread)
```

Figure 11: calculation of number of threads required to cover the grid

Once again, the three separate kernel calls are made for the interior nodes, edge nodes, and corner nodes, in that order. Inside the kernel function, the same process is followed as in the previous parallel implementation, with some modifications to handle the variability in thread-to-element assignment. The unique thread index is compared to the number of threads required (rather than the size of the grid) in order to determine whether any further computation is required for that thread. If the thread index is small enough, iteration is performed between 0 and the number of elements per thread specified by the user. Inside each iteration, the “real” index of the thread is calculated (including the offset from the iteration), and if the resulting real index is within the grid size, the same (i, j) calculation and update operations take place as in the previous implementation. As an example, the kernel function for the internal node update is displayed in Figure 12.

```
@cuda.jit
def drum_interiorMult(grid, grid_back1, grid_back2, n, p, gridsize, elementsPerThread, threadsRequired):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < threadsRequired:
    for iteration in range(elementsPerThread):
      # updating on adjacent nodes isn't a problem because we're using the history grids only
      realIdx = idx * elementsPerThread + iteration

      if realIdx < gridsize * gridsize:
        j = realIdx // gridsize
        i = realIdx % gridsize

        if i != 0 and j != 0 and i != gridsize - 1 and j != gridsize - 1:
          # interior node
          grid[i][j] = ((p * (grid_back1[i-1][j] + grid_back1[i+1][j] +
                              grid_back1[i][j-1] + grid_back1[i][j+1] -
                              4 * grid_back1[i][j])) +
                        (2 * grid_back1[i][j]) - ((1 - n) * grid_back2[i][j]))
          grid[i][j] /= (1 + n)
```

Figure 12: grid update kernel function for interior nodes

The remainder of the operation, involving printing the output for position [256, 256] and updating the grid history, functions the same as the previous parallel implementation. This implementation was proven to function correctly by changing the grid size to 4x4 and verifying that the resulting output matched that of the previous two implementations. Additionally, the console output for the 512x512 grid (Figure 13) matches that of the example posted on MyCourses.

```
Iteration 1: 0.0
Iteration 2: 3.998400479824981e-08
Iteration 3: 0.0
Iteration 4: 0.24980013992803154
Iteration 5: 0.0
Iteration 6: 8.9892080370116e-08
Iteration 7: 0.0
Iteration 8: 0.14040029222121936
Iteration 9: 0.0
Iteration 10: 1.403440891682952e-07
Iteration 11: 0.0
Iteration 12: 0.09742231969133452
Iteration 13: 0.0
Iteration 14: 1.9087120709162718e-07
Iteration 15: 0.0
Iteration 16: 0.0745294056393437
Iteration 17: 0.0
Iteration 18: 2.413783011458448e-07
Iteration 19: 0.0
Iteration 20: 0.060320634512427584
Iteration 21: 0.0
Iteration 22: 2.918343471751079e-07
Iteration 23: 0.0
Iteration 24: 0.050645649253699805
Iteration 25: 0.0
Iteration 26: 3.4222637009984303e-07
Iteration 27: 0.0
Iteration 28: 0.0436341257299501
Iteration 29: 0.0
Iteration 30: 3.925480441346873e-07
Iteration 31: 0.0
Iteration 32: 0.03831973287361857
Iteration 33: 0.0
Iteration 34: 4.4279595196121775e-07
Iteration 35: 0.0
Iteration 36: 0.03415301791490642
Iteration 37: 0.0
Iteration 38: 4.929681108493421e-07
Iteration 39: 0.0
Iteration 40: 0.030798546140529577
Iteration 41: 0.0
Iteration 42: 5.430633118003722e-07
Iteration 43: 0.0
Iteration 44: 0.028039967254005736
Iteration 45: 0.0
Iteration 46: 5.930807933462375e-07
Iteration 47: 0.0
Iteration 48: 0.02573148721548688
Iteration 49: 0.0
Iteration 50: 6.43020065892496e-07
```

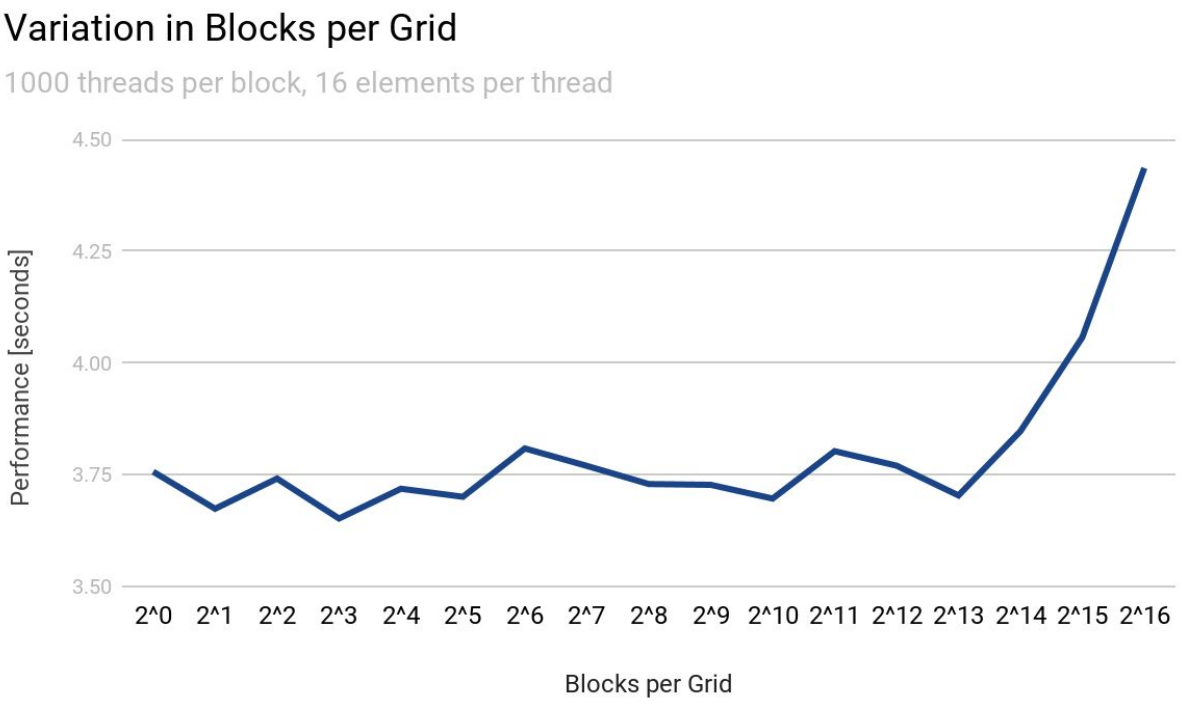
Figure 13: n:1 element-to-thread implementation console output

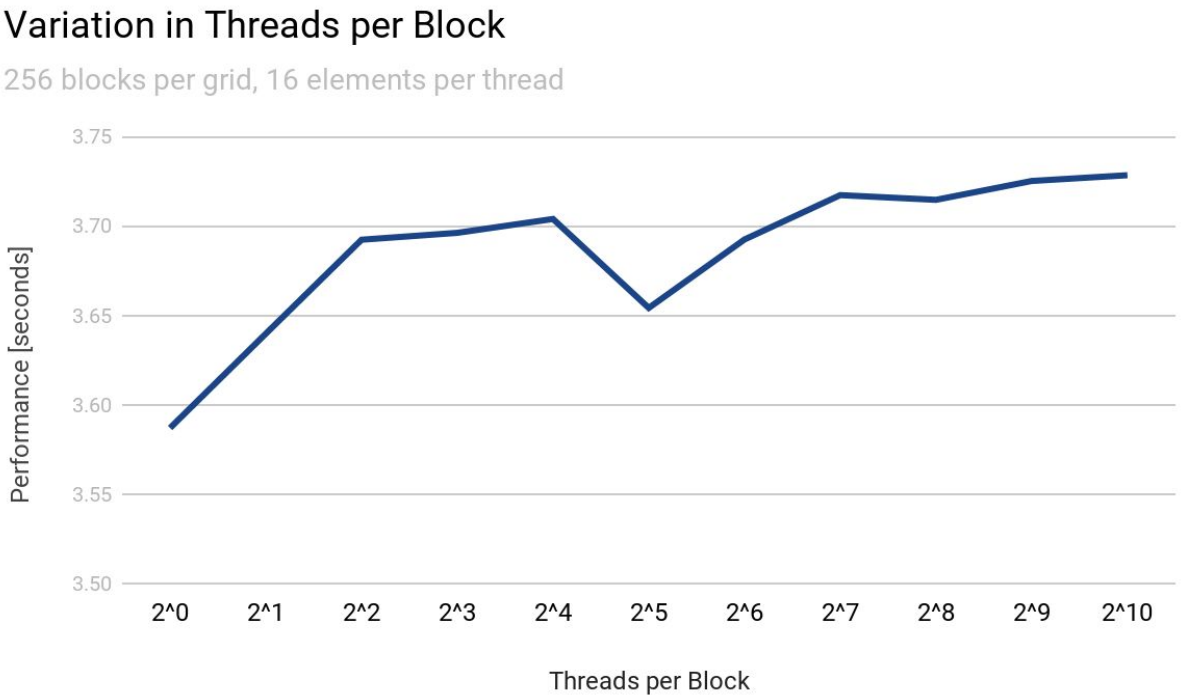
Performance Analysis

To test the implementation's performance, the 512x512 grid was simulated for 500 iterations while using various numbers of blocks per grid, threads per block, and elements per thread. The two types of variations which had an effect on performance were the number of blocks and threads, and the number of elements per thread.

Number of Blocks and Threads

Varying the number of blocks per grid used or the number of threads per block used have the same effect on performance, because these two variations have the same end result: a variation in the number of threads available for parallel computing. Increasing the number of blocks per grid or threads per block results in faster performance due to increased potential for parallelization. Once the number of threads approaches the maximum possible number of threads (i.e. number of elements in the grid), performance begins to degrade as the overhead of multithreading overwhelms the benefits of parallelization.



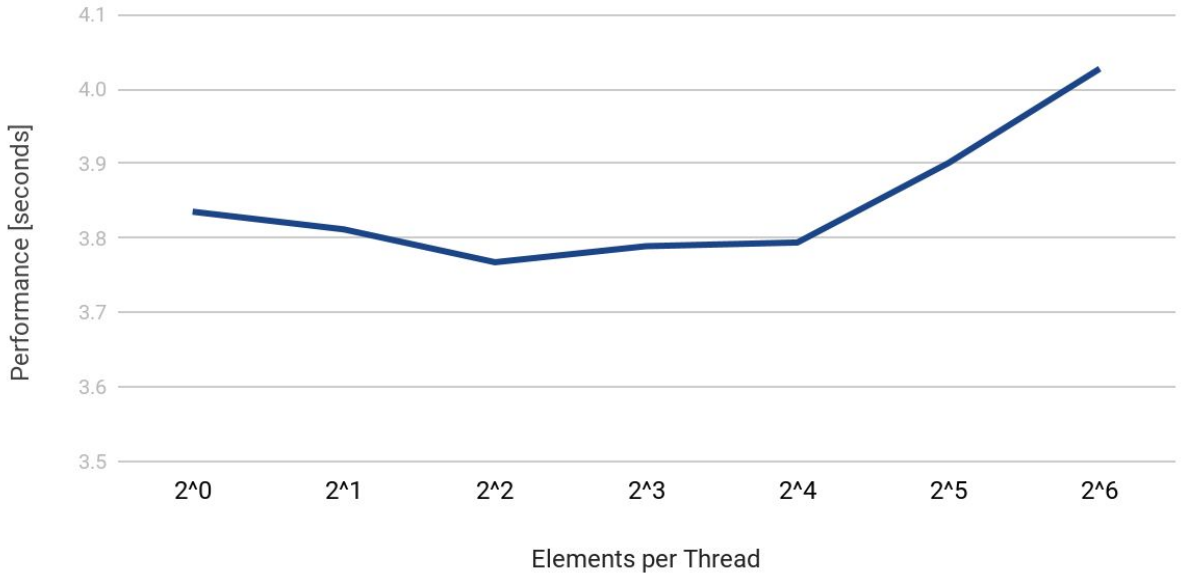


Number of Elements per Thread

Increasing the number of elements per thread initially improves performance, because it reduces the overhead of multithreading caused by assigning each thread to a small number of elements. However, as the number of elements per thread begins to approach the number of elements in the grid, the operation becomes increasingly sequential, decreasing the performance benefits of adding elements to each thread.

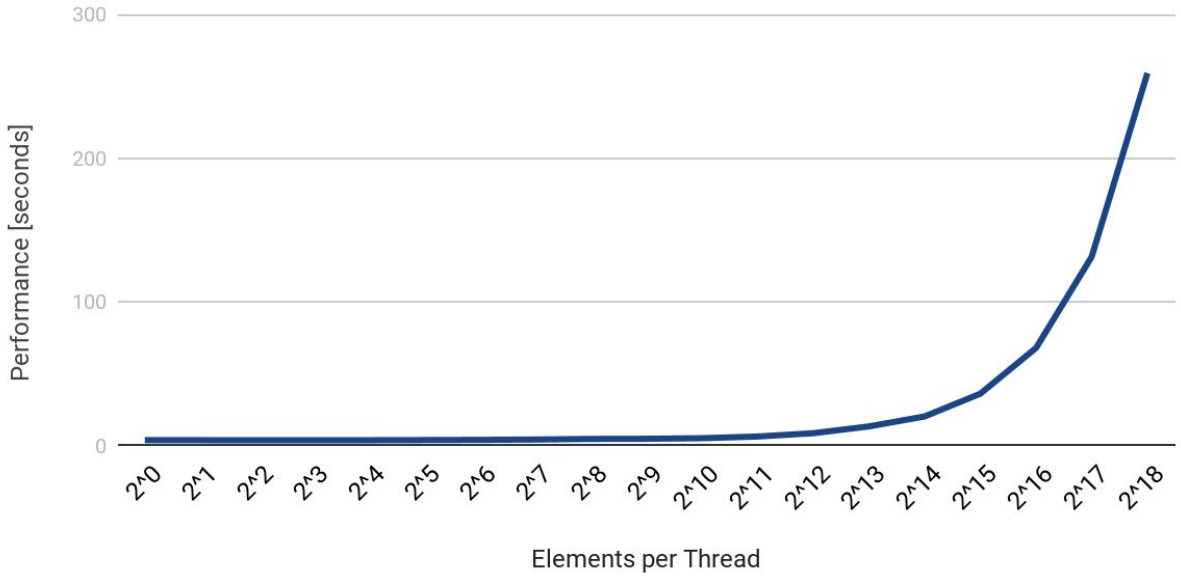
Variation in Elements per Thread - Low Range

512 blocks per grid, 1000 threads per block



Variation in Elements per Thread - Full Range

512 blocks per grid, 1000 threads per block



Appendix

The code is pasted below. Additionally, the code and relevant input and output files are attached alongside this report. Since the following was written in a python notebook it may not

```
## ECSE 420 - Lab 3
# Drum hit simulation using a grid of finite elements.

import os
dev_lib_path = !find / -iname 'libdevice'
nvvm_lib_path = !find / -iname 'libnvvm.so'
assert len(dev_lib_path)>0, "Device Lib Missing"
assert len(nvvm_lib_path)>0, "NVVM Missing"
os.environ['NUMBAPRO_LIBDEVICE'] = dev_lib_path[0]
os.environ['NUMBAPRO_NVVM'] = nvvm_lib_path[0]

from numba import cuda
import numpy as np
import cv2
import matplotlib.pyplot as plt
import math

# Question 1 - Sequential Implementation

def sequential(N=4, iterations=4):
      u = np.zeros([N, N], dtype = float)
      u1 = u.copy()
      u2 = u.copy()

      G = 0.75
      rho = 0.5
      nu = 0.0002

      # hit = "1" at u(2,2)
      u1[2][2] = 1

      for itera in range(iterations):
      # response 1 to N-2
      for i in range(1, N-1):
            for j in range(1, N-1):
```

```
                u[i][j] = ((rho * (u1[i-1][j] + u1[i+1][j] +
                                   u1[i][j-1] + u1[i][j+1] -
                                   4*u1[i][j])) +
                      (2 * u1[i][j]) - ((1 - nu) * u2[i][j]))
                u[i][j] /= (1 + nu)

        # range 1 to N-2
        for x in range(1, N-1):
              ## EDGES
              # left edge = .75 * interior
              u[0][x] = G * u[1][x]
              # right edge
              u[N-1][x] = G * u[N-2][x]
              # top edge
              u[x][0] = G * u[x][1]
              # bottom edge
              u[x][N-1] = G * u[x][N-2]

        ## CORNERS
        # top left
        u[0][0] = G * u[1][0]
        # top right
        u[N-1][0] = G * u[N-2][0]
        # bottom left
        u[0][N-1] = G * u[0][N-2]
        # bottom right
        u[N-1][N-1] = G * u[N-1][N-2]

        u2 = u1.copy()
        u1 = u.copy()

        print(u[2][2])
        print("Iteration: {} \nGrid:\n{}\n".format(itera,u))


sequential(4, 4)

## Question 2 - Parallel Implementation #1
# One thread per finite element of the grid

@cuda.jit
def drum_interior(grid, grid_back1, grid_back2, n, p):
```

```
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < 16:
      j = idx // 4
      i = idx % 4

      if i not in [0, 3] and j not in [0, 3]:
      # interior node
      grid[i][j] = ((p * (grid_back1[i-1][j] + grid_back1[i+1][j] +
                       grid_back1[i][j-1] + grid_back1[i][j+1] -
                       4 * grid_back1[i][j])) +
                  (2 * grid_back1[i][j]) - ((1 - n) * grid_back2[i][j]))
      grid[i][j] /= (1 + n)

@cuda.jit
def drum_edges(grid, G):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < 16:
      j = idx // 4
      i = idx % 4

      if i == 0 and j in [1, 2]:
      # left edge
      grid[i][j] = G * grid[i + 1][j]
      elif i == 3 and j in [1, 2]:
      # right edge
      grid[i][j] = G * grid[i - 1][j]
      elif j == 0 and i in [1, 2]:
      # top edge
      grid[i][j] = G * grid[i][j + 1]
      elif j == 3 and i in [1, 2]:
      # bottom edge
      grid[i][j] = G * grid[i][j - 1]

@cuda.jit
def drum_corners(grid, G):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < 16:
      j = idx // 4
      i = idx % 4
```

```
        if i == 0 and j == 0:
        # top left corner
        grid[i][j] = G * grid[i + 1][j]
        elif i == 0 and j == 3:
        # bottom left corner
        grid[i][j] = G * grid[i][j - 1]
        elif i == 3 and j == 0:
        # top right corner
        grid[i][j] = G * grid[i][j + 1]
        elif i == 3 and j == 3:
        # bottom right corner
        grid[i][j] = G * grid[i][j - 1]

# constant parameters
G = 0.75
n = 0.0002
p = 0.5
gridsize = 4

# grid creation
grid = np.zeros((gridsize, gridsize))
grid_back1 = grid.copy()
grid_back2 = grid.copy()
grid_back1[2][2] = 1   # simulate drum hit at time t-1

# variable parameters
numBlocks = 978
threadsPerBlock = 1000
T = 50

# simulate time progression
for t in range(T):
  # function call (numba performs host -> device -> host memory transfers)
      # required order: interior -> edges -> corners
  drum_interior[numBlocks, threadsPerBlock](grid, grid_back1, grid_back2,
n, p)
  drum_edges[numBlocks, threadsPerBlock](grid, G)
  drum_corners[numBlocks, threadsPerBlock](grid, G)

  print("u[2, 2] at iteration {}: {}".format(t + 1, grid[2][2]))
```

```
  if (t < 3):
      print(grid)
      print()

  grid_back2 = grid_back1.copy()
  grid_back1 = grid.copy()

## Question 3 - Parallel Implementation #3
# Variable number of finite elements per thread. Output at [N/2, N/2] for
50 iterations is shown below (for comparison with posted file).

@cuda.jit
def drum_interiorMult(grid, grid_back1, grid_back2, n, p, gridsize,
elementsPerThread, threadsRequired):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < threadsRequired:
      for iteration in range(elementsPerThread):
      # updating on adjacent nodes isn't a problem because we're using the
history grids only
      realIdx = idx * elementsPerThread + iteration

      if realIdx < gridsize * gridsize:
      j = realIdx // gridsize
      i = realIdx % gridsize

      if i != 0 and j != 0 and i != gridsize - 1 and j != gridsize - 1:
            # interior node
            grid[i][j] = ((p * (grid_back1[i-1][j] + grid_back1[i+1][j] +
                              grid_back1[i][j-1] + grid_back1[i][j+1] -
                              4 * grid_back1[i][j])) +
                        (2 * grid_back1[i][j]) - ((1 - n) *
grid_back2[i][j]))
            grid[i][j] /= (1 + n)

@cuda.jit
def drum_edgesMult(grid, G, gridsize, elementsPerThread, threadsRequired):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < threadsRequired:
      for iteration in range(elementsPerThread):
      # updating on adjacent nodes isn't a problem because we're using the
```

```
history grids only
      realIdx = idx * elementsPerThread + iteration

      if realIdx < gridsize * gridsize:
      j = realIdx // gridsize
      i = realIdx % gridsize

      if i == 0 and j in [1, 2]:
            # left edge
            grid[i][j] = G * grid[i + 1][j]
      elif i == 3 and j in [1, 2]:
            # right edge
            grid[i][j] = G * grid[i - 1][j]
      elif j == 0 and i in [1, 2]:
            # top edge
            grid[i][j] = G * grid[i][j + 1]
      elif j == 3 and i in [1, 2]:
            # bottom edge
            grid[i][j] = G * grid[i][j - 1]

@cuda.jit
def drum_cornersMult(grid, G, gridsize, elementsPerThread,
threadsRequired):
  idx = cuda.threadIdx.x + cuda.blockIdx.x * cuda.blockDim.x

  if idx < threadsRequired:
      for iteration in range(elementsPerThread):
      # updating on adjacent nodes isn't a problem because we're using the
history grids only
      realIdx = idx * elementsPerThread + iteration

      if realIdx < gridsize * gridsize:
      j = realIdx // gridsize
      i = realIdx % gridsize

      if i == 0 and j == 0:
            # top left corner
            grid[i][j] = G * grid[i + 1][j]
      elif i == 0 and j == 3:
            # bottom left corner
            grid[i][j] = G * grid[i][j - 1]
      elif i == 3 and j == 0:
```

```
            # top right corner
            grid[i][j] = G * grid[i][j + 1]
      elif i == 3 and j == 3:
            # bottom right corner
            grid[i][j] = G * grid[i][j - 1]

# constant parameters
G = 0.75
n = 0.0002
p = 0.5
gridsize = 512

# grid creation
grid = np.zeros((gridsize, gridsize))
grid_back1 = grid.copy()
grid_back2 = grid.copy()
grid_back1[gridsize // 2][gridsize // 2] = 1   # simulate drum hit at time
t-1

# variable parameters
numBlocks = 978
threadsPerBlock = 1000
elementsPerThread = 16
T = 50

threadsRequired = math.ceil((gridsize * gridsize) / elementsPerThread)

# simulate time progression
for t in range(T):
  # function call (numba performs host -> device -> host memory transfers)
      # required order: interior -> edges -> corners
  drum_interiorMult[numBlocks, threadsPerBlock](grid, grid_back1,
grid_back2, n, p, gridsize, elementsPerThread, threadsRequired)
  drum_edgesMult[numBlocks, threadsPerBlock](grid, G, gridsize,
elementsPerThread, threadsRequired)
  drum_cornersMult[numBlocks, threadsPerBlock](grid, G, gridsize,
elementsPerThread, threadsRequired)

  print("Iteration {}: {}".format(t + 1, grid[gridsize // 2][gridsize //
2]))

  grid_back2 = grid_back1.copy()
```

```
  grid_back1 = grid.copy()

### Comparison with Previous Implementations
# Compare implementation #3 with implementations #1 and #2 to ensure that
results match.

# constant parameters
G = 0.75
n = 0.0002
p = 0.5
gridsize = 4

# grid creation
grid = np.zeros((gridsize, gridsize))
grid_back1 = grid.copy()
grid_back2 = grid.copy()
grid_back1[gridsize // 2][gridsize // 2] = 1   # simulate drum hit at time
t-1

# variable parameters
numBlocks = 978
threadsPerBlock = 1000
elementsPerThread = 16
T = 50

threadsRequired = math.ceil((gridsize * gridsize) / elementsPerThread)

# simulate time progression
for t in range(T):
  # function call (numba performs host -> device -> host memory transfers)
      # required order: interior -> edges -> corners
  drum_interiorMult[numBlocks, threadsPerBlock](grid, grid_back1,
grid_back2, n, p, gridsize, elementsPerThread, threadsRequired)
  drum_edgesMult[numBlocks, threadsPerBlock](grid, G, gridsize,
elementsPerThread, threadsRequired)
  drum_cornersMult[numBlocks, threadsPerBlock](grid, G, gridsize,
elementsPerThread, threadsRequired)

  print("Iteration {}: {}".format(t + 1, grid[gridsize // 2][gridsize //
2]))

  if (t < 3):
```

```
      print(grid)

  grid_back2 = grid_back1.copy()
  grid_back1 = grid.copy()

### Performance Measurements

import time

# constant parameters
G = 0.75
n = 0.0002
p = 0.5
gridsize = 512
T = 500

# grid creation
grid = np.zeros((gridsize, gridsize))
grid_back1 = grid.copy()
grid_back2 = grid.copy()
grid_back1[gridsize // 2][gridsize // 2] = 1   # simulate drum hit at time
t-1

for numBlocks in [1, 4, 8, 32, 128, 512, 2048]:
  for threadsPerBlock in [1, 4, 16, 64, 256, 1024]:
      for elementsPerThread in [1, 4, 16, 64, 256]:
      start = time.time()
      threadsRequired = math.ceil((gridsize * gridsize) /
elementsPerThread)

      # simulate time progression
      for t in range(T):
      # function call (numba performs host -> device -> host memory
transfers)
            # required order: interior -> edges -> corners
      drum_interiorMult[numBlocks, threadsPerBlock](grid, grid_back1,
grid_back2, n, p, gridsize, elementsPerThread, threadsRequired)
      drum_edgesMult[numBlocks, threadsPerBlock](grid, G, gridsize,
elementsPerThread, threadsRequired)
      drum_cornersMult[numBlocks, threadsPerBlock](grid, G, gridsize,
elementsPerThread, threadsRequired)
```

```
        grid_back2 = grid_back1.copy()
        grid_back1 = grid.copy()

        print("Runtime for {} blocks, {} threads, {} elements:
{}".format(numBlocks, threadsPerBlock, elementsPerThread, time.time() -
start))
```