Improving computational efficiency for implementing a sound propagation simulation environment using Python and GPU

Mihai Bucurica
Doctoral School in Electronics, Telecommunications and Information Technology, University “Politehnica” Bucharest, Romania bucurica.mihai@gmail.com

Ioana Dogaru, Radu Dogaru
Natural Computing Laboratory, Dept. of Applied Electronics and Information Eng. University “Politehnica” Bucharest, Romania radu_d@ieee.org, ioana.dogaru@upb.ro

Abstract – This work presents improvements in terms of computational efficiency of a cellular automata based virtual environment for ultra-sound propagation, (previously abbreviated as CANAVI, i.e. Cellular Automata for ultra-sound based robot Navigation). Comparisons with our previous implementations using JAVA indicates good speed-up while using low cost, programming environments based on Python and exploiting GPU’s via NUMBAPRO packages from Continuum. Particularly the “just in time” (jit) compiler was found extremely useful. Several methods of speeding-up the evolution of the cellular automata are proposed and compared herein.

Keywords – ultrasound signal processing, CUDA, GPU, sound propagation simulator, Python, JIT, Cellular Automata

I. INTRODUCTION

Sound propagation modeling and simulation environments are useful tools in a various application range: designing good musical instruments, sound protection from traffic noise of inhabited area, concerto halls design, robot navigation based on ultrasound signals etc. In the past, such a sound-propagation environment called CANAVI [1][2] was designed for the purpose of optimizing robot navigation in an attempt to use the “bat-like” ears of the robot as intelligent sensor capable to discriminate and recognize objects and distances. This initial version for the simulator was written in C++ programming language. Later, an improved version of the simulator was developed in JAVA [3] including a graphical user interface (GUI) and more capabilities allowing to emulate sound wave propagation in a controlled 2D environment and also to perform various virtual experiments such as classification of objects [4] or computing distance to the closest obstacle. Cellular automata are used to model sound propagation and more details on the cellular automata model are given in [1][2].

Since CA are massively parallel, using hardware and programming tools supporting such high degree of fine-grained parallelism are beneficial in speeding-up computations, as reported in [5], [6], [7], [8], [9], thus more diverse and complex virtual experiments can be performed on such a simulator.

A first study on these possibilities was done in [2], indicating that for a good price/speed compromise, utilizing GPUs with adequate software tools is a way to follow. Consequently, Python with JIT facilities and CUDA support was considered recently [5] to implement various cellular platforms. The main reasons behind this choice were the following: i) a large standard library along with vast free packages to use for scientific programming; ii) easy to learn and free to use; iii) the possibility of using CUDA for taking advantage of the parallel computation on GPU. Herein we report a similar approach but used to implement the CANAVI-like sound propagation simulator.

The paper is comparing three implementations of the algorithm using Java, Python with the Numba package which allows us to use the JIT, and Python with the NumbaPro which allows us to use the GPU. The experiments were conducted on a mobile notebook with a CPU (i7-4510U) clocked at 2.0 GHz and turbo at 2.6 GHz and with a GPU (GeForce 840M) with 384 CUDA cores at a speed of 1029 MHz. Python software was used along with the Anaconda distribution package, due to the easy to use and good documentation provided by the Continuum analytics [10] and the Darmstadt University of Technology [11].

II. NUMBA PACKAGE

The Numba package [12], sponsored by the Continuum analytics [10], gives the power to speed up the applications with high performance functions written directly in Python. It is known that pure Python performs poorly, but with a few annotations (e.g. @autojit), array-oriented and math-heavy Python code can be just-in-time compiled to native machine instructions, similar in performance to C or C++, without having to switch languages or Python interpreters.

Numba works by generating optimized machine code using the LLVM compiler infrastructure at import time, runtime, or statically (using the included pycc tool). Numba supports compilation of Python to
run on either CPU or GPU hardware, and is designed
to integrate with the Python scientific software stack.

III. NUMBAPRO PACKAGE

NumbaPro package [13], also produced by the Continuum analytics [10], is an enhanced version of the Numba package which adds premium features for developers to target multicore and GPU architecture with full control of the hardware (thread and block identities). The package offers the user possibility to use ufuncs, generalized ufuncs (gufuns) and also the JIT compiler for CUDA, the last one we used in our implementation. This package is free only for academic users and usage. Compared to a Numba implementation, there is no much difference between the code. One must change the @autojit annotation with @cuda.jit and declare the variables send through the function and also add some management for the memory transfer between the CPU and GPU.

In particular, for the algorithm considered herein, we experienced some problems with the speed boost due to large times for the memory transfer between the CPU and GPU. The speed gained from the parallelization on the graphical processor unit was lost at the memory transfer. After a lot of research, we propose in IV a method to decrease that transfer time and make the algorithm faster.

IV. ALGORITHM’S IMPLEMENTATIONS

In this chapter we present the algorithm’s implementations used and each adjustment. While a detailed description of the algorithm is given in [2], herein we briefly remind that for each cell in the CA two state variables (SV and P) must be updated synchronously using the rules (1) (2) depicted in the next figure (the two layers of each cell being represented there, while indices U(up), D(down), L(left), R(right) refer to neighbor cells. S is a pressure from exterior (only for cells of the space occupied by sound transmitters), as we can see in Fig. 1.

Figure 1. Initialize arrays in Python using Numpy

A. Java implementation

In order to increase speed, several modifications were considered in this new version:

- The stimulation pressure signal (S) computation was moved outside the CA simulator (it will receive it pre-computed in an array)
- The receiving pressure vector (recorded at a location indicated by a chosen cell within the 2D space) is now stored as an array into the memory instead of being wrote to a txt file.

B. Numba (Python using CPU) implementation

The same modifications described above were considered while the algorithm was fully rewritten in Python using the Numpy scientific package [14]. This package contains among other things: a powerful N-dimensional array object, useful linear algebra and fast array copy and initialization functions. Using this package we replaced all Python arrays with corresponding Numpy ones as follows:

```python
P_old = np.zeros([height, width], dtype=np.float64)
P_new = np.zeros([height, width], dtype=np.float64)
SV_old = np.zeros([height, width], dtype=np.float64)
SV_new = np.zeros([height, width], dtype=np.float64)
pressure_result = np.zeros([iterations], dtype=np.float64)
time_per_iteration = np.zeros([iterations], dtype=np.float64)
```

The transfer between two consecutive CA states (to implement the synchronous update) can now be made faster and cleaner using the Numpy syntax:

```python
np.copyto(P_old, P_new)
np.copyto(SV_old, SV_new)
```

The function implementing one step dynamics of the CA is annotated with @autojit to invoke the compiler. Herein the “matrix” is a Numpy array containing the sound transparencies (a map of obstacles) of the considered scenario.

Figure 2. Initialize arrays in Python using Numpy

C. Numbapro (Python using GPU with CUDA JIT)

This algorithm approach requires a high level of parallelism and the solution presented in this paper was found after long time developing cycles and numerous versions. Examples in [15] were found useful in the development process. The annotation @autojit was replaced with the corresponding @cuda.jit. This means that the code is executed by the GPU instead of the CPU. One must also declare the block and grid dimensions (e.g. as indicated in [11]), before calling the function. The choice of these two parameters is essential for getting highest speed-up and has to be carefully tuned for each type of problem. For example, herein the following choice was found to be optimal:

```python
@autojit
def loop_matrix(signal, matrix, P_new, P_old, SV_new, SV_old,
```

In designing the Numbapro implement, it is first started from the Numba implementation where the one-step CA advance is now replaced to consider the GPU as target processor. We discovered that a lot of time was lost in data transfer necessary at each iteration (here the state matrixes P_new, P_old,
SV_new, SV_old) from GPU and CPU (in order to emulate the synchronous CA model). In order to overcome this, our solution is to exclude CPU by designing a second GPU function ensuring that data is copied only within global memory on the GPU board. This function runs on the GPU as shown in Fig. 6. This function is called similarly to np.copy in Fig. 3 after each running of the one-step advance of the CA simulator model (call of “loop_matrix” function).

Figure 6. The function used to transfer data between the matrices

```
cudajit((argtypes=[float[:,:], float[:,:], float[:,:], float[:,:], float[:,:],[float[:,:],[int, [int] ]]),
def copy_matrix(P_new, P_old, SV_new, SV_old):
```

Using arbitrary large scenario dimensions (height, width), the implementation required some improvements (due to block and grid dimensions constraints [16]) that we successfully performed. Our solution (Fig. 7) embedded in the “loop_matrix” (one step CA advance) function is inspired from one of the examples in [17].

Figure 7. Correct iteration to get the correct cell of scenario

```
height = matrix.shape[0]
width = matrix.shape[1]
startX, startY = cuda.grid(2)
gridx = cuda.gridDim.x * cuda.blockDim.x
cuda.blockDim.y;
for j in range(startX + 1, width - 1, gridx):
    for i in range(startY + 1, height - 1, gridx):
        cell = matrix[i, j] # simulator point
```

Without the improvements presented above, the NUMBAPRO implementation gave certain errors compared to the JAVA or NUMBA implements. Such errors, sometimes minor, were dependent on the GPU unit as well. After doing the improvements a perfect match of the simulation dynamics is observed on various computing platforms.

V. EXPERIMENTAL RESULTS

To correctly report an improved algorithm, one must compare the work with the original one. In the previous work we used a slower computer. We re-run the tests on the newest one described in part I and reported the results in Table 1.

The virtual scenario dimensions used was 200x200 and the emitter and receiver were at the exact same point. The speed increase of 3.37 times is solely the effect of changing the computing platform. The following efficiency descriptors were considered in [3] and computed as:

\[ E_{tpc} = E_t / (width \times height \times iterations) \]  \hspace{1cm} (1)

Where \( E_{tpc} \) denotes “Execution time per cell” and \( E_t \) denotes “Execution time”. Therefore \( E_{tpc} \) includes, besides the real execution time of iteration, data transfer and other preparing or displaying functions. The formula behind this is:

\[ E_{tpc} = I_{tpc} + \Delta t \]  \hspace{1cm} (2)

Where \( I_{tpc} \) denotes “Iteration time per cell” and \( \Delta t \) denotes “data transfer time”. In most papers the most relevant time is given as \( I_{tpc} \) expressed in ns/cit (nanoseconds / cell iteration) or Mcells/s (Mega cells / second). Consequently herein we consider \( I_{tpc} \) values as well as displayed in Table II.

The reference on the chosen computing platform (old Java implementation) has a performance of 17 ns/cit or 58.82 Mcells/s to be next compared with the improved implementations. The experiments were conducted on the same virtual environment using the same obstacle scenario. Several scenario sizes (200 x 200) up to (1024 x 1024) where considered and the number of CA iterations is 1200 (same as in [2], and sufficient to receive relevant echoes).

Ten simulations under same conditions were performed in order to report the performance as an average over all experiments. The simulation time (Time per simulation – Tps) expressed in milliseconds was also recorded. Next, tables III, IV, V and VI resume the results for various scenario sizes allowing comparisons of all implementation technologies considered herein: Java, Python (JIT from NUMBA), Python (GPU using NUMBAPRO).

For the experiments we considered the scenario sizes (200x200) as well as 256x256 and 512x512. The results for these tests are presented in Tables II to VI.

TABLE I. COMPARING RESULTS BETWEEN COMPUTERS

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Computer</th>
<th>Time per simulation (E(_t))</th>
<th>Time per cell (E(_{tpc}))</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Intel Pentium 8400</td>
<td>7.242 s</td>
<td>167 ns</td>
</tr>
<tr>
<td>2</td>
<td>Intel i7-5410U</td>
<td>2.147 s</td>
<td>49 ns</td>
</tr>
</tbody>
</table>

The virtual scenario dimensions used was 200x200 and the emitter and receiver were at the exact same point. The speed increase of 3.37 times is solely the effect of changing the computing platform. The following efficiency descriptors were considered in [3] and computed as:

\[ E_{tpc} = E_t / (width \times height \times iterations) \]  \hspace{1cm} (1)

Where \( E_{tpc} \) denotes “Execution time per cell” and \( E_t \) denotes “Execution time”. Therefore \( E_{tpc} \) includes, besides the real execution time of iteration, data transfer and other preparing or displaying functions. The formula behind this is:

\[ E_{tpc} = I_{tpc} + \Delta t \]  \hspace{1cm} (2)

Where \( I_{tpc} \) denotes “Iteration time per cell” and \( \Delta t \) denotes “data transfer time”. In most papers the most relevant time is given as \( I_{tpc} \) expressed in ns/cit (nanoseconds / cell iteration) or Mcells/s (Mega cells / second). Consequently herein we consider \( I_{tpc} \) values as well as displayed in Table II.

The reference on the chosen computing platform (old Java implementation) has a performance of 17 ns/cit or 58.82 Mcells/s to be next compared with the improved implementations. The experiments were conducted on the same virtual environment using the same obstacle scenario. Several scenario sizes (200 x 200) up to (1024 x 1024) where considered and the number of CA iterations is 1200 (same as in [2], and sufficient to receive relevant echoes).

Ten simulations under same conditions were performed in order to report the performance as an average over all experiments. The simulation time (Time per simulation – Tps) expressed in milliseconds was also recorded. Next, tables III, IV, V and VI resume the results for various scenario sizes allowing comparisons of all implementation technologies considered herein: Java, Python (JIT from NUMBA), Python (GPU using NUMBAPRO).

TABLE II. THE RESULTS FOR THE DATASETS USED

<table>
<thead>
<tr>
<th>Nr.</th>
<th>Dataset</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>E(_{tpc})</td>
<td>49</td>
</tr>
<tr>
<td>2</td>
<td>Ns/cit (I(_{tpc}))</td>
<td>17</td>
</tr>
<tr>
<td>3</td>
<td>Mcells/s (Speed = 1/I(_{tpc}))</td>
<td>58.82</td>
</tr>
</tbody>
</table>

TABLE III. RESULTS FOR 200x200 SCENARIO

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Tps(s)</th>
<th>Ns/cit</th>
<th>Mcells/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA old algorithm</td>
<td>2.26</td>
<td>17</td>
<td>59</td>
</tr>
<tr>
<td>JAVA new algorithm</td>
<td>0.554</td>
<td>2.68</td>
<td>372</td>
</tr>
<tr>
<td>Python Numba</td>
<td>0.219</td>
<td>3.09</td>
<td>322</td>
</tr>
<tr>
<td>Python NumbaPRO</td>
<td>0.429</td>
<td>4.75</td>
<td>210</td>
</tr>
</tbody>
</table>

TABLE IV. RESULTS FOR 256x256 SCENARIO

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Tps(s)</th>
<th>Ns/cit</th>
<th>Mcells/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA old algorithm</td>
<td>3.123</td>
<td>14</td>
<td>68</td>
</tr>
<tr>
<td>JAVA new algorithm</td>
<td>0.888</td>
<td>2.58</td>
<td>386</td>
</tr>
<tr>
<td>Python Numba</td>
<td>0.364</td>
<td>3.10</td>
<td>327</td>
</tr>
<tr>
<td>Python NumbaPRO</td>
<td>0.584</td>
<td>2.90</td>
<td>344</td>
</tr>
</tbody>
</table>

TABLE V. RESULTS FOR 512x512 SCENARIO

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Tps(s)</th>
<th>Ns/cit</th>
<th>Mcells/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA old algorithm</td>
<td>9.092</td>
<td>15</td>
<td>66</td>
</tr>
<tr>
<td>JAVA new algorithm</td>
<td>3.764</td>
<td>3.55</td>
<td>281</td>
</tr>
<tr>
<td>Python Numba</td>
<td>2.043</td>
<td>4.09</td>
<td>244</td>
</tr>
<tr>
<td>Python NumbaPRO</td>
<td>2.154</td>
<td>2.18</td>
<td>457</td>
</tr>
</tbody>
</table>
TABLE VI. RESULTS FOR 1024x1024 SCENARIO

<table>
<thead>
<tr>
<th>Scenario</th>
<th>Tps(s)</th>
<th>Ns/cit</th>
<th>Mcells/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAVA old algorithm</td>
<td>31.44</td>
<td>16</td>
<td>63</td>
</tr>
<tr>
<td>JAVA new algorithm</td>
<td>14.276</td>
<td>3.88</td>
<td>257</td>
</tr>
<tr>
<td>Python Numba</td>
<td>8.343</td>
<td>4.22</td>
<td>236</td>
</tr>
<tr>
<td>Python NumbaPRO</td>
<td>10.320</td>
<td>2.51</td>
<td>397</td>
</tr>
</tbody>
</table>

As seen in Fig. 8 and Fig. 9, the improved Java implementation performs slightly better than NUMBA-based Python. For larger scenario sizes, the GPU proves efficient NUMBA-Pro implementation providing best speed (up to 500 Mcells/second, comparable with other CA and CNN systems implemented in the same technology [5]).

VI. CONCLUSIONS

Several new implementations of a cellular-automata based sound propagation simulator are reported herein. Particularly, it was found that using Python is particularly useful since it supports a high productivity approach to use CUDA and GPU when available. Using JIT compilers embedded in NUMBA and NUMBAPRO packages ensures very good acceleration is expected when using better GPU, the one used here is a modest one. The total processing time Tps in NUMBAPRO is however slower than in Numba, possibly due to the second function added to the algorithm (copy_matrix). Overall Python implementations with JIT are better than what is achieved in Java. Further research will consider improving the copy_matrix function, in order to reduce the transfer times influencing the NUMBAPRO implementation.

REFERENCES


[10] https://www.continuum.io/


