Efficient Parallel Simulations of Wireless Signal Wave Propagation

Dorin-Marian Ionita
Computer Science Department
University Politehnica of Bucharest
Bucharest, Romania
dorin.marian.ionita@gmail.com

Filip-George Manole
Computer Science Department
University Politehnica of Bucharest
Bucharest, Romania
filip.manole@gmail.com

Emil-Ioan Slusanschi
Computer Science Department
University Politehnica of Bucharest
Bucharest, Romania
emil.slusanschi@cs.pub.ro

***Abstract*—A common pattern in high performance scientific computing is the structured grid pattern in which one or more elements of a matrix are computed as a stencil operation of other matrix neighbouring elements. Since there are multiple options to efficiently implement this pattern on modern computing architectures, we provide a comparison of the performance of a number of parallel implementations on a multi-core system with GPU capabilities and also on a FPGA embedded inside a SoC. The application used for this case study implements the propagation of wireless signals in a bi-dimensional environment, considering reflections and signal attenuation. The parallel programming paradigms examined in this paper include CUDA, TBB, Rust, OpenMP, and HLS as hardware description paradigm, with CUDA proving to be the fastest implementation.**

***Index Terms*—numerical simulation, PDE, structured grid, CUDA, TBB, OpenMP, Rust, GP-GPU, FPGA, HLS, wave propagation.**

I. Introduction

Out of the numerous types of parallel problems, among the most common are data parallel problems. For this type of problems the data is split in parts and various similar computations are performed in parallel on the data chunks. Such problems are commonly seen in high performance computing (HPC), examples being the patterns of processing on structured or unstructured grids [1]. This kind of parallelization can be implemented using various frameworks, paradigms or specialized programming languages. Deciding which is the most appropriate approach for a given problem, is not straightforward. This paper provides a model for wireless signal wave propagation, taking into account both reflections and attenuation and then implements it in a numerical simulation. We then use this use-case to compare the performance of OpenMP [2] (Open Multi-Processing), Threading Building Blocks [3] (TBB), Rust [4], NVidia CUDA [5] (Compute Unified Device Architecture), and Vivado High Level Synthesis [6] (HLS).

The background information from which we began our work, namely the wave equation and the free path loss formula for wireless signal, is presented in Section II. More details on the numerical solution for the partial differential equation of wave propagation and also the reflection and the damping model of the amplitude of the wave based on the free path power loss model are offered in Section III. Section IV introduces a pseudo-code of the algorithm and outlines the implementation details in the considered parallel programming paradigms and logic synthesis technique. Next, in Section V, we graphically show the results of our model, with and without amplitude attenuation, and we also provide performance measurements for the different implementations. We show that Rust and OpenMP offer similar performance, with TBB clearly outperforming the previous two. We also show that even though CUDA is more efficient than TBB, the performance penalty for using the later over the former is limited. Finally, we highlight that for the high performance computing pattern of structured grid, GPUs are a better hardware choice than FPGAs, mainly due to the in-device memory limitations. Section VI concludes the paper and identifies future research directions.

II. Related Work

We start from the partial differential equation of wave propagation:

$$\frac{\partial^2 A}{\partial x^2} + \frac{\partial^2 A}{\partial y^2} + \frac{\partial^2 A}{\partial z^2} = \frac{1}{c^2}\frac{\partial^2 A}{\partial t^2} \tag{1}$$

Solving this PDE leads us to a function which describes the amplitude of the wave in any given points of space and at any time moment, provided the fact that we also have the initial conditions for it, i.e. the points at which the perturbation first occurs, or, in simpler words, the position and time moment where the signal source is placed. This is precisely what our simulation does: iterates over space and time and computes the amplitude in each point of the 4D space.

$$A = f(x, y, z, t), \forall x, y, z, \tau \in D \tag{2}$$

The discretization of this PDE is provided in [14]. The previous equation doesn't take into account reflections. In order for the model to be more accurate we also use the Law of Reflection which is outlined in Figure 1 and obtained from the Dirichlet conditions of the PDE, as explained in [16] and section III-C. To put it simply, the Law of Reflection states that the incidence angle has the same value as the reflection angle.

$$\sphericalangle\theta_i \equiv \sphericalangle\theta_r \tag{3}$$

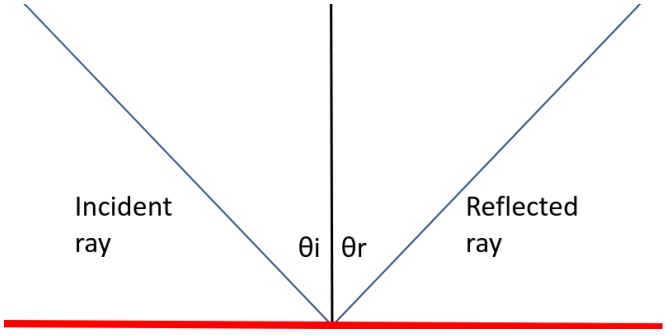

Fig. 1: Law of Reflection Graphically Explained

For a wireless signal this simple description proves to be not enough, because it doesn't take into account the attenuation of the signal. In developing the damping model for the amplitude we started from the free path loss model for power attenuation of signal [15].

$$P_r = \frac{P_t G^2 \sigma \lambda^2}{(4\pi)^3 R^4 L} \tag{4}$$

Expression 4 relates the received power P_r, at a given point in space to the power transmitted P_t, by the signal source, based on the distance R between them. A graphical power variation with space is represented in Figure 2.

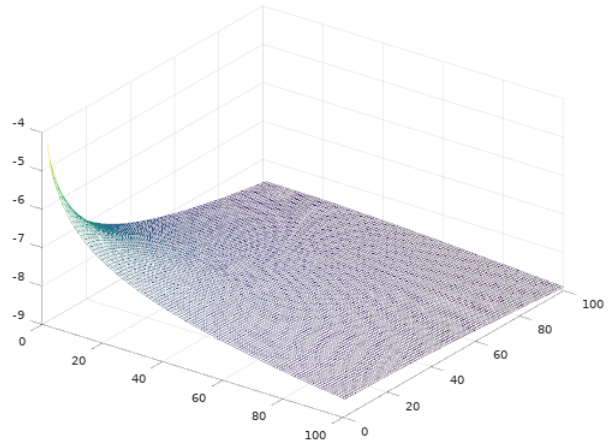

Fig. 2: Log-Representation of Power Loss with Space

In order to relate the amplitude to the power of the wave, the following formula [13] proved to be useful:

$$P = \frac{1}{2}\mu\omega^2 v A^2 \tag{5}$$

III. The Damping Model

First of all, it is worth noting that while in our work we interpreted our model as describing the propagation of a wireless signal, the actual mathematical description used is actually more general. This means that our model can be used in sound engineering (since mechanical and electromagnetic waves share the same mathematical description), earthquake analysis, radar localization, just as well as it can be used for wireless signal description.

This section explains how we augmented the wave equation model to also include damping wave effects, before diving into the actual numerical solution to our model, which also integrates reflections.

A. Damping Model for the Wireless Signal

A common challenge for wireless communications is that the radio signal quickly vanishes into background noise even for relatively short propagation distances. For this reason, the effect can not be neglected by our model and our simulation.

While the wave equation offers information about the amplitude of the wave, the free path power loss formula offers information about the power of the signal. In order to obtain a model for the amplitude of the damped propagating wave, one needs to relate the two formulae by transforming the power loss into amplitude loss over space.

In equation 5 we make the following notation:

$$\frac{1}{2}\mu\omega^2 v = \psi \tag{6}$$

which leads us to the expression for the power of the signal at any point in space and time

$$P(x, y, z, t) = \psi A^2(x, y, z, t), \forall x, y, z, t \tag{7}$$

Let the coordinates be (x_s, y_s, z_s) for the signal source and (x_r, y_r, z_r) for a receiver on the grid.

In order to maintain the generality of our model, we need to eliminate the proportionality factor which particularizes the model for a given frequency and propagation velocity. We do this by dividing the signal power at transmission against the power at reception. For isotropic environments, it follows from expressions 5 and 6 that:

$$\frac{P(x_s, y_s, z_s, t)}{P(x_r, y_r, z_r, t)} = \frac{\psi A^2(x_s, y_s, z_s, t)}{\psi A^2(x_r, y_r, z_r, t)} =$$

$$= \frac{A^2(x_s, y_s, z_s, t)}{A^2(x_r, y_r, z_r, t)}, \forall t \tag{8}$$

This allows us to relate the amplitude of the wave at transmission source against the amplitude at receiver, using a proportionality factor χ, specific to each space coordinate and time moment.

$$\frac{A(x_s, y_s, z_s, t)}{A(x_r, y_r, z_r, t)} = \sqrt{\frac{P(x_s, y_s, z_s, t)}{P(x_r, y_r, z_r, t)}} =$$

$$= \chi(x_r, y_r, z_r, t), \forall t \tag{9}$$

In other words, we use the free path loss formula for power to obtain the proportionality factor at any space coordinates, and considering the initial amplitude of the wave we can use expression 10 to compute the amplitude of the damped wave anywhere in space and at any moment of time.

$$A(x_r, y_r, z_r, t) = \frac{A(x_s, y_s, z_s, t)}{\chi(x_r, y_r, z_r, t)}, \forall t \tag{10}$$

B. Numerical Solution to the Wave Equation

In order to use our model in a simulation, we need to discretize it. One simple way to do this is to consider finite differences instead of derivatives in the wave equation. The bi-dimensional continuous space becomes a bi-dimensional grid by considering a finitely small space discretization step Δx. We also use a smallest time duration possible Δt (the discretization step in time). An example of such a precomputed discretization is provided in [14] and briefly adapted to the use-case outlined in Section V.

The general form of the amplitude gradients on the grid is:

$$\frac{\partial A}{\partial x}_{i+\frac{1}{2},j} \approx \frac{A_{i+1,j} - A_{i,j}}{\Delta x} \tag{11}$$

$$\frac{\partial A}{\partial x}_{i-\frac{1}{2},j} \approx \frac{A_{i,j} - A_{i-1,j}}{\Delta x} \tag{12}$$

$$\frac{\partial A}{\partial x}_{i,j+\frac{1}{2}} \approx \frac{A_{i,j+1} - A_{i,j}}{\Delta x} \tag{13}$$

$$\frac{\partial A}{\partial x}_{i,j-\frac{1}{2}} \approx \frac{A_{i,j} - A_{i,j-1}}{\Delta x} \tag{14}$$

By differentiating an expression similar to expressions 11–14 one more time, we obtain the discrete expression 15 for the second derivative with respect to time, in which $A^t_{i,j}$ denotes the amplitude at point (i, j) of the mesh and at moment t.

$$\frac{\partial^2 A}{\partial t^2} \approx \frac{A^{t+1} - 2A^t + A^{t-1}}{\Delta t^2} \tag{15}$$

Finally, by replacing the formulae in the wave equation we obtain the discrete form, as follows:

$$\frac{A^{t+1}_{i,j} - 2A^t_{i,j} + A^{t-1}_{i,j}}{\Delta t^2} =$$

$$\frac{A^t_{i+1,j} + 2A^t_{i,j} + A^t_{i-1,j}}{\Delta x^2} + \frac{A^t_{i,j+1} - 2A^t_{i,j} + A^t_{i,j-1}}{\Delta x^2} \tag{16}$$

Consequently, the recursive expression for computing the amplitude becomes:

$$A^{t+1}_{i,j} = 2A^t_{i,j} - A^{t-1}_{i,j} +$$

$$\frac{1}{2}(A^t_{i+1,j} - 2A^t_{i,j} + A^t_{i-1,j} + A^t_{i,j+1} - 2A^t_{i,j} + A^t_{i,j-1}) \tag{17}$$

We notice that this is a formula which computes the amplitude at any given point in space (discretized in a bidimensional mesh) based on the amplitude of the neighbouring points at two previous time steps. In fact, the formula can be interpreted as a stencil operation which combines points on the mesh from 2 different time stamps to obtain the mesh at a new time moment. This implies that the points at the same time step can be computed in parallel, but have to be serial as time increases. In other words, the stencil kernel can be applied in parallel to generate the new mesh values, but it can't generate multiple new meshes at the same time - this has to happen serially. This influences the design of our algorithms, as it is presented in section IV-A.

C. Integrating Reflections in the Numerical Solution

In order to integrate reflections of wireless signal into our model we need additional information, apart from expression 1 and its initial conditions (generation of signal by a source point). Namely, we need to add information about the reflectors on the mesh. In the theory of PDEs this is achieved by fixing the so-called boundary conditions. Out of the many possible behaviours that can be exhibited at boundary, producing precisely the effect of reflection is achieved by using the so called Dirichlet boundary condition [16].

Let D be the definition domain for the (x, y, z) tuples in expression 1 and $\partial D \subset D$ its boundary (the points on the edge of the reflectors), then the Dirichlet boundary conditions can be stated as:

$$A(x, y, z, t) = 0, \forall (x, y, z) \in \partial D, \forall t \tag{18}$$

What the Dirichlet boundary conditions mean for our numerical solution of the PDE is that expression 17 is modified such that in computing the value of $A^{t+1}_{i,j}$ any terms for which points (i, j) fall inside the boundary of the reflectors are replaced with 0. Figure 3 graphically explains the stencil operation. The reflector is colored in green. For a fixed point (in black) on the boundary, the squares in black, blue and red are the points involved in the stencil operation. The blue points are the points outside of the reflector and the red ones are inside the reflector. Therefore, the red points overwrite the corresponding terms of expression 17 with 0 while the blue points don't. The black point is also overwriting the amplitude value with 0.

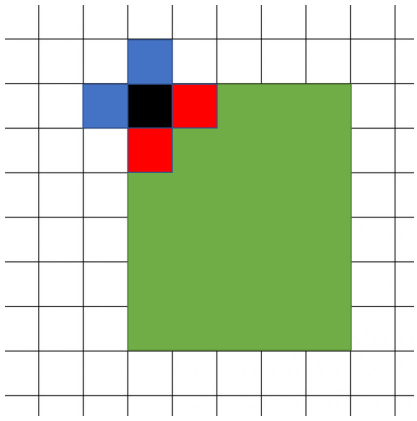

Fig. 3: Boundary Reflections using Modified Stencil Kernel

For example, for Figure 3 the resulting stencil operation is reduced to:

$$A^{t+1}_{i,j} = \frac{1}{2}(A^t_{i-1,j} + A^t_{i,j-1}) \tag{19}$$

IV. Model Implementation

In order to integrate our propagation model in a simulation, we have first developed a serial algorithm. The serial algorithm serves as a base for both the software parallel implementation, as introduced in the first part of this section, and the hardware description, as explained in the second part.

A. The Parallel Propagation Algorithm

Before diving into the parallel algorithm, we first explain how we deduced a serial algorithm for simulating the propagation model. To this aim, we iterate over each discrete time step in the time interval during which the wave propagates, and for each such time step, at every point of the discrete space (the grid) we compute the amplitude of the signal as a numerical solution to the wave equation, i.e. by using the Formula 17.

Algorithm 1 Simulate Wave Propagation - Serial

```
INPUT: source position and amplitudes in time, reflectors
for jiffy in discrete_time_interval do
    Compute current transmitted power P_t based on expr. 5
    for line in grid_lines do
        for col in grid_columns do
            Compute amplitude based on expression 17
            Adjust amplitude for reflection based on fig. 3
            Compute received power P_r based on expression 4
            Compute χ based on P_r according to expression 9
            Adjust amplitude using χ factor as in expression 10
        end for
    end for
end for
```

Algorithm 2 Simulate Wave Propagation - Parallel

```
INPUT: source position and amplitudes in time, reflectors
for jiffy in discrete_time_interval do
    Execute the loop serially
    Compute current transmitted power P_t based on expr. 5
    for line in grid_lines do
        Execute the loop in parallel
        for col in grid_columns do
            Execute the loop in parallel
            Compute amplitude based on expr. 17
            Adjust amplitude for reflection based on fig. 3
            Compute received power P_r based on expr. 4
            Compute χ based on P_r according to expr. 9
            Adjust amplitude using χ factor as in expr. 10
        end for
    end for
end for
```

Since equation 1 doesn't offer information about reflection, we use the boundary conditions to adjust the amplitude of the point where appropriate. Up to now, the algorithm simulates the propagation of the signal considering reflections, but not damping. In order to obtain the damping effect, we compute the χ proportionality factor and we use it to normalize the previously obtained amplitude of the signal, as shown in Algorithm 1.

In order to parallelize this algorithm, we notice that, for a given jiffy t, each point of the grid can be computed in parallel, being independent from one another, as they are based only on the past values of the amplitude, according to formula 17. What this means is that considering a fixed jiffy, the computation of every point of the mesh is independent of the value of the other points at the same time moment.

On the other hand, we need to run our algorithm in a serial manner throughout each individual time step. That happens because formula 17 shows that the amplitudes at each jiffy depend on the amplitudes at the previously two jiffies. This results in a simulation which runs in parallel in space, but serially in time. This leads to the parallel algorithm outlined in Algorithm 2.

B. System Architecture for the FPGA and the SoC

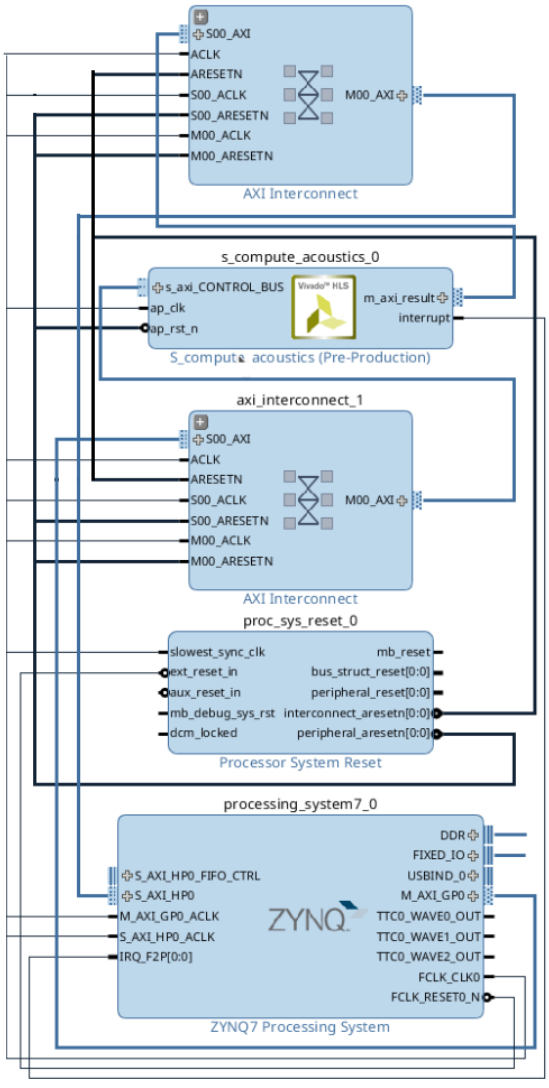

Fig. 4: FPGA - CPU Integration

In order to describe the hardware that implements our model, the three nested loops in Algorithm 1 were implemented in the C allowing for the translation of Vivado HLS to VHDL [7]. The pipelining and hardware loop unrolling behaviour is emplying default settings, as we provided no manual directives to the compiler. The I/O ports allowing intercommunication between the IP implemented in FPGA and the CPU inside the SoC (System on Chip) were specified. The resulting IP was integrated with the CPU inside the SoC using an approach specific to AMBA [8], as shown in Fig. 4.

The Processing System and the Programmable Logic are connected via a double master-slave interface using AXI [8] Interconnect. Namely, the result port of the IP (Intellectual Property) plays the role of a slave to the processor, which, in this case, is the master. On the other hand, the control bus is a master to the Processing System. Furthermore, the interrupt line of the generated IP is connected to the interrupt line of the CPU.

For reset-control capabilities, a Processor System Reset IP from Vivado is used, all the components being connected in a single-clock domain infrastructure, provided by the Processing System and set to a frequency that allows for timing closure, as described in section V-A.

V. Results and Discussion

With respect to the model of wave propagation, a graphical representation without considering damping is shown in Figure 5. The graphical results of the simulation which takes into account the damping of the wave are shown in the Figure 6. Each simulation was run for 30 seconds physical time, with a time-step of 0.05 seconds, totaling 600 iterations.

With respect to parallel implementations and comparisons, we implemented the previous parallel algorithm in a number of different parallel programming paradigms, namely OpenMP, TBB, Rust and NVidia CUDA. The first technology which has been employed was OpenMP due to its increased productivity obtained by abstracting many low-level details from the programmer.

TBB was chosen to allow for more control than OpenMP. During our experiments we noticed that the productivity overhead of TBB is rather small when compared to the productivity of OpenMP. More to the point, about the same amount of time was invested in the TBB and OpenMP implementations.

Our third choice was Rust, as it emerged as a new programming language with the promise of easy and safe parallelization of code [17]. Rust offers two paradigms for parallelization: one for shared memory and one for message passing, based on channels. The channels connect threads on the same machine so no special packing is required and therefore no additional overhead is incurred.

Finally, GP-GPU architectures have established themselves as one of the most important technologies for data parallel applications. Since NVidia CUDA [18] is one of the most efficient and wide-spread libraries, we chose to port our code to this technology as well.

Fig. 5: Wave Propagation with no Amplitude Damping

Fig. 6: Wave Propagation with Amplitude Damping

Apart from using parallel programming paradigms to implement our algorithm, we also obtained a hardware description using the HLS mechanism which we then used to program our FPGA. The motivation behind this choice is that data parallel problems, such as the structured grid problem we used as example, are known to be well-suited for running on GPUs, but on the other hand FPGAs allow for close coupling of the hardware with the problem it solves. As a result of this, the abstractions stack on top of an FPGA are more shallow than on general computing devices such as CPUs and GPUs, resulting in less overhead. Therefore, both GPUs and FPGAs promise good result, but more clarity was needed in order to decide the trade-offs between these hardware architectures for the structured grid pattern.

The propagation model without damping was implemented

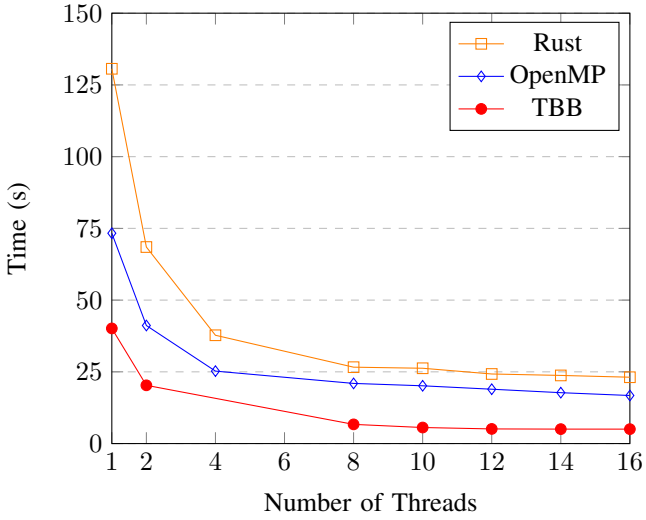

Fig. 7: Rust vs. OpenMP vs. TBB Execution Time without Damping

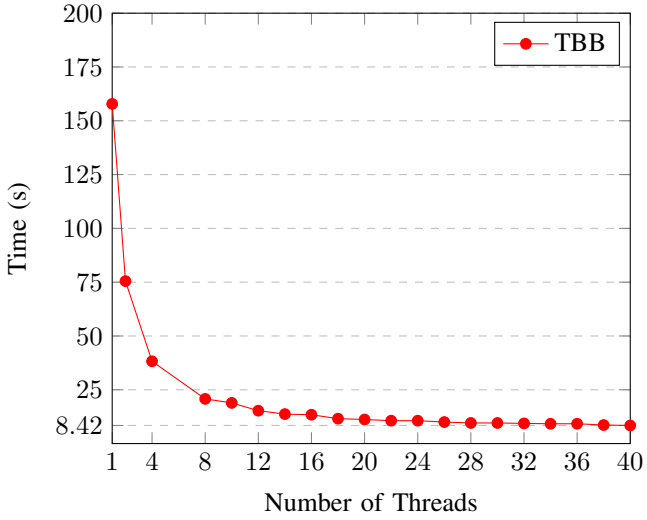

Fig. 8: TBB Execution Time with Damping

in OpenMP, TBB and Rust, while the model including damping was implemented in TBB, CUDA and HLS.

A. Execution Environment

The used experimental setup included two execution queues on the University Politehnica of Bucharest's cluster. The model without damping was run on a queue with 12 Intel Xeon cores running 24 threads at 2.67GHz and 32G RAM, while the model which includes the damping factor was run on a queue with 20 Intel Xeon cores with 40 threads running at 2.5GHz, 62G RAM. The CUDA version was tested on a Tesla K40M NVidia GPU, using API version 9.

For our embedded FPGA experiments we used Zedboard from Xilinx, which incorporate a Zynq 7000 SoC [9]. As part of the SoC, an ARM A9 CPU runs the bare metal drivers for the FPGA design, starting the execution and receiving the results. The base clock of the FPGA runs at 100MHz, but we had to lower it to 95MHz in order to achieve timing closure. Worth noting is that the FPGA of Zynq 7000 benefits from 36Kb internal BRAM [10].

B. Execution Time

Figure 7 presents the execution time for implementations which do not consider the damping of the wave. When it comes to the parallelization, TBB produces the binary with the shortest execution time, Rust and OpenMP being almost twice as slow. We noticed that while Rust seems slower than OpenMP and TBB in the serial versions, the difference between Rust and OpenMP shrinks significantly when multiple threads are used. Combining this with the deadlock and race conditions protection provided by Rust's type system, we consider that for data parallel problems one might easily choose Rust over OpenMP. This is particularly relevant when requiring better safety guarantees. TBB provides a significant improvement over OpenMP in terms of execution time. However, this incurs a small productivity penalty.

Grid \ Block	4	8	16
4	103.12	7.79	41.14
8	7.44	31.45	135.84
16	27.21	149.22	592.19

TABLE I: CUDA Execution Time with Damping (s)

Figure 8 presents the execution time for the TBB implementation when amplitude damping is taken intro consideration. The dimension of the grid in this case is $2048x2048$, the same as in the CUDA case. The execution speed for 24 threads is $8.418s$.

Table I shows the execution time for the CUDA implementation, when the input is the same as for Figure 8. The performance of CUDA seems to vary greatly with the dimension of the kernel, with the best choice for dimension not being straightforward. In fact, what we did was to iterate through the space of hyper-parameters (possible kernel dimensions) to find the best fit for our simulation. All the kernels use bi-dimensional square grids and bi-dimensional square blocks. We can see that the best execution time is is $7.44s$, so it is about 11% better than TBB when it comes to execution speed. Also, Table I outlines that symmetric sizes of kernels (i.e. obtained by swapping the grid size with the block size) exhibit similar execution times.

C. Speed-up

In the case of the model without amplitude damping, the speed-up outlined in Figure 9 shows that TBB is better than both OpenMP and Rust, with Rust offering better speed-up than OpenMP. One should note that for all considered programming paradigms the speed-up is flat-lining at 12 threads, which is to be expected on a system with 12 CPU physical cores running a CPU-intensive numerical simulation. Furthermore, Figure 10 and Table II outline that the maximum

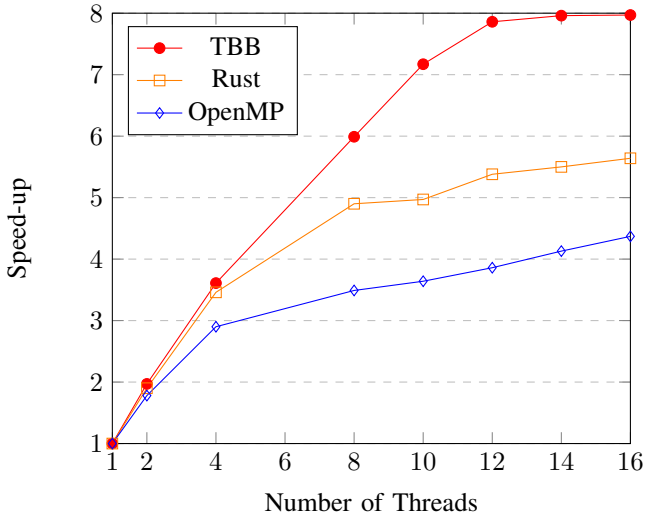

Fig. 9: Rust vs. OpenMP vs. TBB Speed-up without Damping

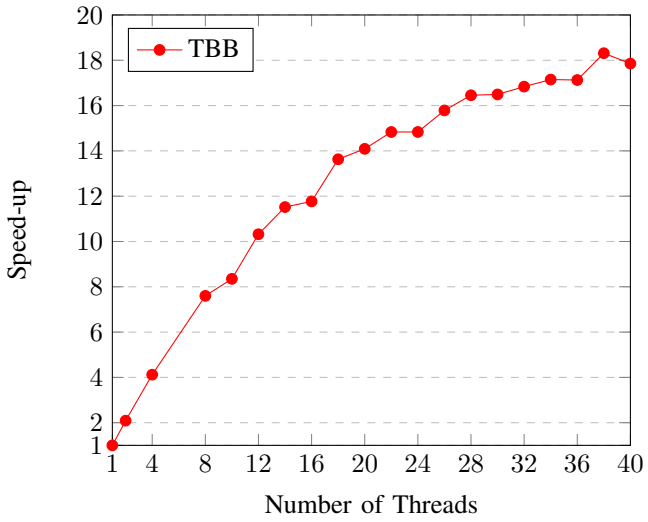

Fig. 10: Speed-up TBB with Damping

speed-up achieved using a K40m with 2880 NVidia CUDA cores is only slightly better than using TBB on a 20 core Intel Xeon machine, i.e. 19.90 vs. 18.31.

D. Efficiency

When it comes to computing efficiency, Figure 11 shows the TBB performs best, with Rust and OpenMP performing significantly worse, in this domain.

Grid \ Block	4	8	16
4	1.43	19.01	4.10
8	19.90	4.70	1.09
16	5.44	0.99	0.25

TABLE II: CUDA Speed-up

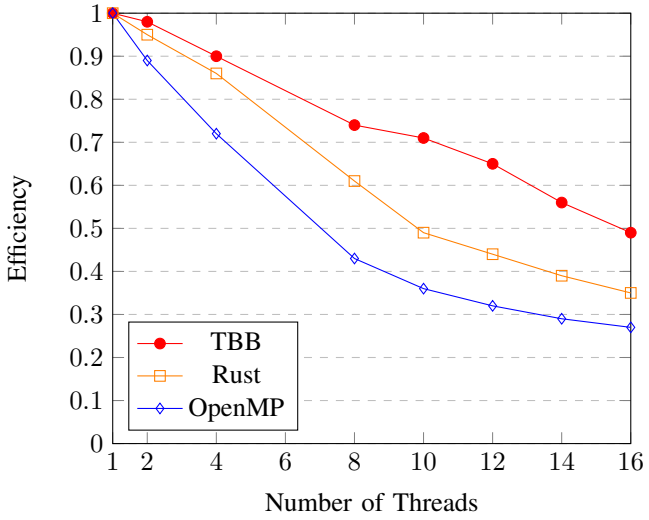

Fig. 11: Efficiency of Rust, TBB and OpenMP without Damping

Resource	Utilization	Available	Utilization %
LUT	21710	53200	40.81
LUTRAM	764	17400	4.39
FF	16457	106400	15.47
BRAM	99	140	70.71
DSP	70	220	31.82
BUFG	1	32	3.13

TABLE III: FPGA Consumed Resources

E. FPGA Results

First thing we noticed is that memorising three bi-dimensional meshes inside the FPGA is the main bottleneck. This happens because the three matrices, representing the meshes, are bound to the BRAM cells. Since the BRAM size is limited, it is quickly exhausted. This means that we had to reduce the meshes to $100x100$. Consequently, required hardware resources are given in Table III.

In fact, we consider that the tiny size of the internal memory of the FPGA is the main limitation against efficiently implementing our algorithm on this platform, since in order to execute the algorithm on larger meshes we need to run data through the FPGA multiple times, in a serial fashion. We consider that this is also the main advantage of the GPU, when dealing with data parallel problems: the fact that they have a large amount of internal memory. For example, the used GPU has $12GB$.

When comparing to the C serial version (which was modified to work with HLS), we notice that for the software implementation the execution time is $1.9ms$, while for the hardware one it is $466.71\mu s$. This translates to a speed-up of 4.07.

With respect to power consumption, we notice Xilinx Vivado estimates it to be $2.269W$, which is only 0.96% of the

$235W$ consumed by Nvidia Kepler K40M, considering that our design runs at 12% the frequency of the GPU ($95MHz$ instead of $745MHz$ [11]).

F. Discussion

To sum up, considering the technologies used in our experiments, there is always a trade-off between productivity and performance. While CUDA is more efficient than TBB, TBB doesn't require the understanding of the GP-GPU paradigm or even the existence of a GPU. On the other hand, when choosing between Rust and OpenMP the trade-off becomes more complex. What is actually traded is the productivity of OpenMP over the safety of Rust. It is significantly harder to reach peak performance with Rust, but there are better chances that the program is correct, giving the fact that deadlocks and race conditions are eliminated by the strong typing rules and the unique ownership system. We encountered very few bugs when setting up our experiments with Rust, compared to our CUDA implementation. Furthermore, while FPGAs show some advantages for the particular type of problem we solved, especially considering power consumption, they are heavily limited by the amount of internal memory and that differentiates GPUs (and CUDA) in a better choice for the problems falling into the structured grid pattern.

VI. Conclusion and Future Work

To conclude, we provide a numerical model for wireless signal propagation taking into account reflection and attenuation. The considered model is general enough to describe earthquake and acoustic wave propagation, as well as radar.

We consider the simulation of the wave propagation an example of the structured grid pattern in scientific computing and we compare parallel implementations in OpenMP, Rust, CUDA, and TBB, showing that OpenMP and Rust exhibit similar performance and that the best performance overall is achieved by the CUDA implementation. We also compared the performance of a hardware implementation using HLS against the serial version in C and against CUDA, thus outlining the trade-offs between the GPU and FPGA platforms for this use-case. Each tested paradigm has certain particularities which makes them better suited for particular use-cases, such as safety concerns, rapid development needs, performance, power efficiency, etc. The purpose of this work was to provide information for users and application developers, helping them decide which parallel programming paradigm to use when deploying a particular numerical algorithm, and having specific performance or resource requirements.

The considered signal propagation model can be further augmented to better match reality by taking into account a heterogeneous propagation environment, modifying the speed of propagation of the wave and therefore the ψ factor. To date, our model considers only the damping of electromagnetic waves due to the permittivity of the environment. In reality, when hitting a wall, the wave doesn't only reflect, but it is also partially absorbed by the wall, as the collision is not perfectly elastic. This phenomena affects both the amplitude and the power of the reflected signal.

Furthermore, with respect to numerical error, one can extend the model to have the amplitude of the source a significantly nonlinear function and then use automatic differentiation [12] instead of finite differences to compute the derivatives within machine-precision.

At least two possible research directions can be pursued starting from this work. The first is combining multiple technologies, perhaps involving heterogeneous computing techniques, e.g. using MPI for distributed data processing, with each node computing the data using CUDA. The second is to hand-optimize our FPGA version so that the speed-up is increased, as shown possible in other research results involving HLS for FPGA [19] and use a specialized language for HLS, such as OpenCL [20].

Acknowledgements

The authors would like to thank Mihai Trascau for the original version of the numerical simulation of wave propagation.

References

[1] Jiyuan Tu and Guan-Heng Yeoh and Chaoqun Liu, "Chapter 6 - Practical Guidelines for CFD Simulation and Analysis," in Computational Fluid Dynamics (Second Edition), pp. 219–273, 2013, doi: 10.1016/B978-0-08-098243-4.00006-8.

[2] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," in IEEE Computational Science and Engineering, vol. 5, no. 1, pp. 46–55, Jan.-March 1998, doi: 10.1109/99.660313.

[3] Reinders, James. Intel Threading Building Blocks: outfitting C++ for multi-core processor parallelism. " O'Reilly Media, Inc.", 2007.

[4] Saligrama, Aditya and Shen, Andrew and Gjengset Jon, "A Practical Analysis of Rust's Concurrency Story" in CoRR, 2019.

[5] Kirk, David. (2007). NVIDIA CUDA software and GPU parallel computing architecture. 7. 103–104. 10.1145/1296907.1296909.

[6] Xilinx Inc., Vivado Design Suite User Guide, pp. 7–19, 2012.

[7] Xilinx Inc., Vivado Design Suite User guide, pp. 198-388, 2020.

[8] ARM Inc., AMBA AXI and ACE Protocol Specification, pp. 119-130, 2011

[9] Digilent Inc, Zedboard Hardware User Guide, pp. 3, 2014.

[10] Xilinx Inc, Zynq-7000 SoC Data Sheet Overview, pp. 1, 2018.

[11] Nvidia Inc, Nvidia Tesla K40M Data Sheet, pp. 1.

[12] Atilim Gunes Baydin, Barak A. Pearlmutter, Alexey Andreyevich Radul, Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. The Journal of Machine Learning Research, 18(153):1–43, 2018

[13] Lane, William and Kovacs, J. and McHarris, O., "Intensity and Energy in Sound Waves" in Physnet MISN 0-203, 2002.

[14] Trascau, Mihai. Ecuatia undelor pentru acustica 2D, pp. 23–26, Diploma Thesis. University Politehnica of Bucharest, June 2009.

[15] Seybold, John. Introduction to RF Propagation, pp. 107-108, Wiley, 2005.

[16] Langtangen, Hans Peter. On the impact of boundary conditionsin a wave equation, pp. 3, Center for Biomedical Computing, Simula Research Laboratory, September 2014.

[17] Klabnik, Steve and Nichols, Carlos. The Rust Programming Language, pp. 421–422, No Starch Press, 2019.

[18] Garland, Michael. (2010). Parallel computing with CUDA. 1 - 1. 10.1109/IPDPS.2010.5470378.

[19] Ionita, Dorin-Marian and Deaconescu, Razvan. Hardware Accelerator for Python Bytecode, pp. 31, University Politehnica of Bucharest, June 2019.

[20] Stone, John and Gohara David and Shi Guochun. OpenCL: A Parallel Programming Standard for Heterogeneous Computing Systems. 2010.