VITIS REPORT

0️⃣

ABSTRACT

The goal of this project is to implement the same code used in the first EUMaster4HPC Challenge into an FPGA environment, investigate its numerous optimizations, achieve adequate performance, and compare to other architectures that have made use of various parallelization approaches.

To do this, we choose to simulate the operation of an FPGA using Vitis, a tool provided accessible by Xilinx, to produce an RTL code that can subsequently be performed on a physical board.

Therefore, the fundamental goal of the project has been to write code that is as scalable, portable, and performs as well as possible while utilizing as many low-level optimizations as the architecture permits.

1️⃣

MATHEMATICAL PROBLEM

The Poisson equation is a partial differential equation that arises in many areas of science and engineering, including electrostatics, heat transfer, fluid mechanics, and image processing. The solution represents the potential, temperature, fluid velocity, or image intensity, depending on the application.

The equation reads

$$\nabla^2 \nu(x,y) = f(x,y) \qquad\qquad in\ \ \Omega \subseteq \mathbb{R}^2$$

since we are in a two-dimensional domain. In this case, we consider $f(x,y) = sin(x+y)$.

We can pursue the discretization of the problem; we use the Finite Difference Method. We remember that $\nabla^2 \nu(x,y) = \nabla \cdot (\nabla \nu(x,y))$ and we introduce a mesh of length h.

So we have

$$\begin{gathered}\nabla \cdot \nu(x,y) = \frac{d}{dx}\nu(x,y) \ + \ \frac{d}{dy}\nu(x,y) = \\ = \frac{1}{h}(\nu(x+h,y) \ - \ \nu(x,y)) \ + \ \frac{1}{h}(\nu(x,y+h) \ - \ \nu(x,y)) = \\ = \frac{\nu(x,y+h) \ - \ 2\nu(x,y) \ + \ \nu(x+h,y)}{h}.\end{gathered}$$

Then the discretization leads to

$$\begin{gathered}\nabla \cdot (\nabla \nu(x,y)) = \frac{d^2}{dx^2}\nu(x,y) \ + \ \frac{d^2}{dy^2}\nu(x,y) = \\ = \frac{\nu'(x,y+h) \ - \ 2\nu'(x,y) \ + \ \nu'(x+h,y)}{h} = \\ = \frac{\nu(x+2h,y+h) \ + \ \nu(x+h,y+2h) \ - \ 4\nu(x+h,y+h) \ + \ \nu(x,y+h) \ + \ \nu(x+h,y)}{h^2}\end{gathered}$$

By renaming the function values and organizing those into a matrix, we obtain

$$\nabla^2 \nu_{i,j} = \frac{1}{h^2}(\nu_{i+1,j} \ + \ \nu_{i,j+1} \ - \ 4\nu_{i,j} \ + \ \nu_{i-1,j} \ + \ \nu_{i,j-1}) = f_{i,j}.$$

This way to write the Poisson 2D problem leads to the linear system

$$Tv = h^2 f,$$

where

$$T = pentadiag(1, 1, -4, 1, 1).$$

This linear system will be solved iteratively with the Jacobi Method.

2️⃣

ALGORITHM AND CODE

The algorithm we applied is the Iterative Jacobi Method.

Starting from an initial guess $v^{(0)}$ we apply the following stencil algorithm:

$$v^{(k+1)} = D^{-1}(h^2 f \ - \ Rv^{(k)}),$$

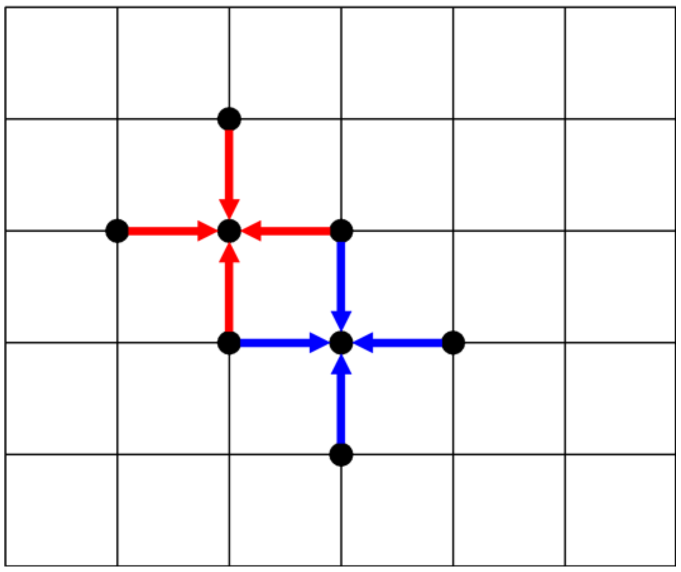
where $D = diag(T)$ and $R = T - D$.

We carry on until we meet the stopping criterion, based on the error

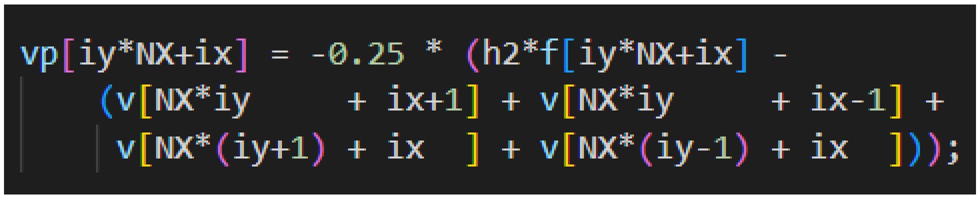
$$e^{(k)} = \frac{\xi^{(k)} \times N_x \times N_y}{\sum_{i=1}^{N_x \times N_y} |v_i^{(k)}|},$$

where $\xi_i^{(k+1)} = x_i^{(k+1)} - x_i^{(k)}$ is the distance between two consecutive iteratons and $\xi^{(k)} = \max_i |\xi_i^{(k)}|$ is the max between all the components of the distance vector; so the exit condition will be $e < \mathrm{E}$, where E can be chosen at will.

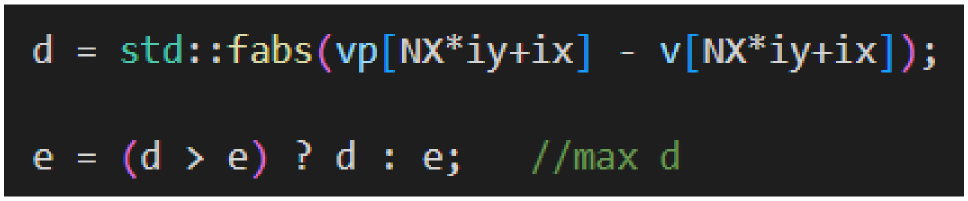
Basically, what we are doing is using the four nearest neighbors in terms of the increment h of every partition element of the domain.



We considered only the matrix, organized by rows within a pointer, to compute solution values at each iteration.



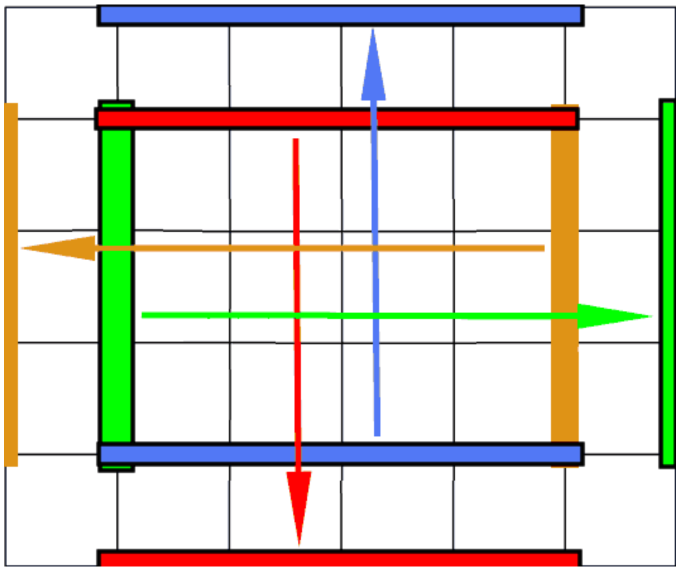
```
vp[iy*NX+ix] = -0.25 * (h2*f[iy*NX+ix] -
    (v[NX*iy     + ix+1] + v[NX*iy     + ix-1] +
     v[NX*(iy+1) + ix  ] + v[NX*(iy-1) + ix  ]));
```



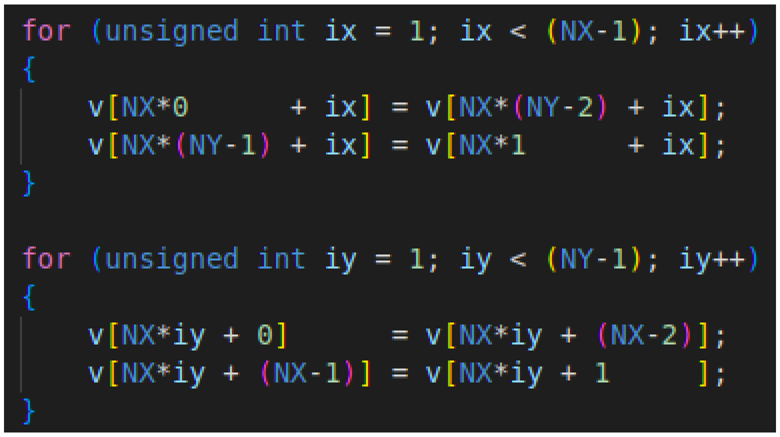
```
d = std::fabs(vp[NX*iy+ix] - v[NX*iy+ix]);

e = (d > e) ? d : e;    //max d
```

As we can see from the last picture, the error is updated at each iteration for every vector value, and at the end the “worst” coordinate distance is stored on the e variable.

The values on the border are copied as shown in the following images:



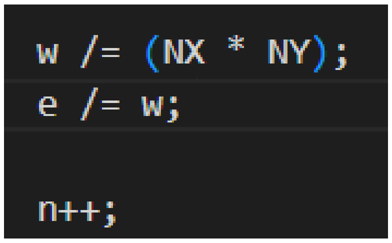
The boundary conditions are updated at each iteration. Please note that the corners were left equal to 0 from the start.



```
for (unsigned int ix = 1; ix < (NX-1); ix++)
{
    v[NX*0      + ix] = v[NX*(NY-2) + ix];
    v[NX*(NY-1) + ix] = v[NX*1      + ix];
}

for (unsigned int iy = 1; iy < (NY-1); iy++)
{
    v[NX*iy + 0]      = v[NX*iy + (NX-2)];
    v[NX*iy + (NX-1)] = v[NX*iy + 1     ];
}
```

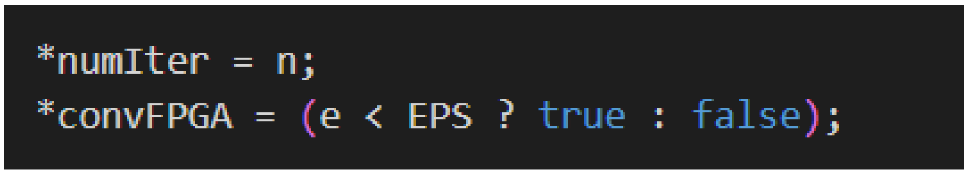
Then we created a variable w where accumulate the sum of the values norm and afterwards we divided that for the number of elements.

Finally, we divided the error for the variable w.



```
w /= (NX * NY);
e /= w;

n++;
```

Lastly, the exit condition is checked at the end of the external loop.



```
*numIter = n;
*convFPGA = (e < EPS ? true : false);
```

Information about the convergence of the method is stored through these pointers.

This is the original code, and then we carried on with some interesting optimizations.

3️⃣

VITIS

3.1 Introduction and workflow

High-Level Synthesis (HLS) is an automated design process that takes an abstract behavioral specification of a digital system and generates a register-transfer-level structure that realizes the given behavior.

The Vitis HLS tool synthesizes a C or C++ function into RTL code for implementation in the programmable logic (PL) region of a Versal Adaptive SoC, Zynq MPSoC, or AMD FPGA device.

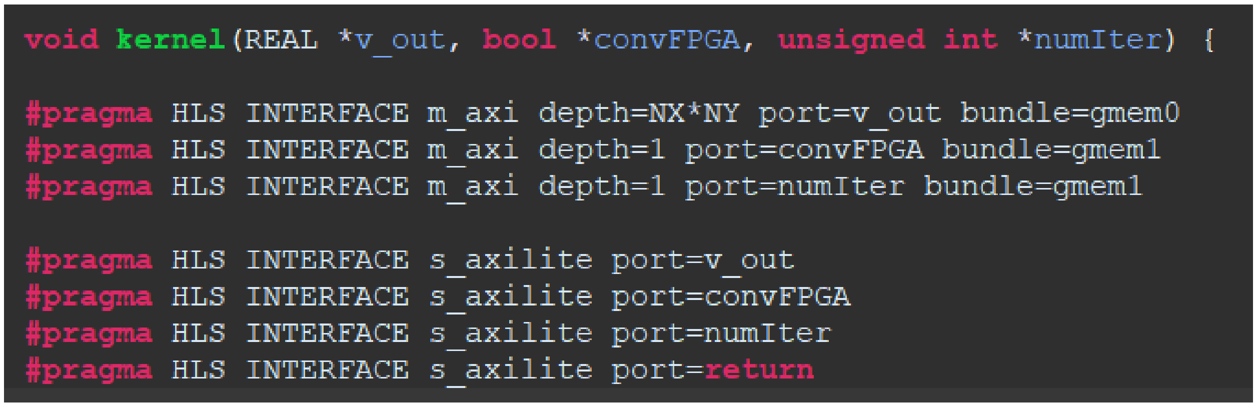
We are going to use the Zynq Ultrascale+ ZCU106 Evaluation Platform board.

The flow using HLS had the following steps:

1. Write the algorithm at a high abstraction level using C
2. Verify the correctness of the program doing a C-Simulation
3. Generate the RTL code doing a C-Synthesis
4. Check the correctness of RTL through a Co-Simulation
5. Optimize the code until performance goals are met

We tested a lot using a 16x16 matrix.

After the github repository was created (it is available here https://github.com/AndreaOrtenzi/MdP-Poisson2DJacobi), we proceeded with some pragma directives in order to establish a connection between the Vitis kernel and physical FPGA ports.

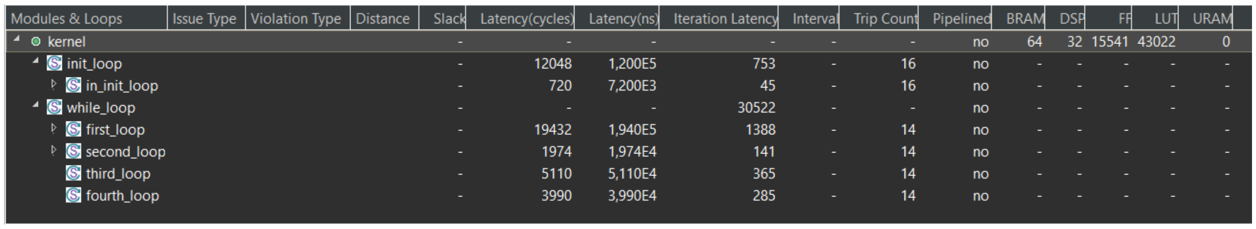


```
void kernel(REAL *v_out, bool *convFPGA, unsigned int *numIter) {

#pragma HLS INTERFACE m_axi depth=NX*NY port=v_out bundle=gmem0
#pragma HLS INTERFACE m_axi depth=1 port=convFPGA bundle=gmem1
#pragma HLS INTERFACE m_axi depth=1 port=numIter bundle=gmem1

#pragma HLS INTERFACE s_axilite port=v_out
#pragma HLS INTERFACE s_axilite port=convFPGA
#pragma HLS INTERFACE s_axilite port=numIter
#pragma HLS INTERFACE s_axilite port=return
```

The Vitis kernel is very easy to compile both in C-Simulation and in C-Synthesis, and it offers a comfortable interface to visualize results.

C-Simulation outputs are simple to read and understand; they are exactly the same as the original C code. Conversely, C-Synthesis outputs must be analyzed more carefully; in fact, C-Synthesis data are organized in some tables, like that shown in the picture below, where we are analyzing the original code performances.

Here we have all the information about our synthesis, like latency in cycles, interval, trip count, floating point operations, and this is for all the loops we are going to select and optimize.

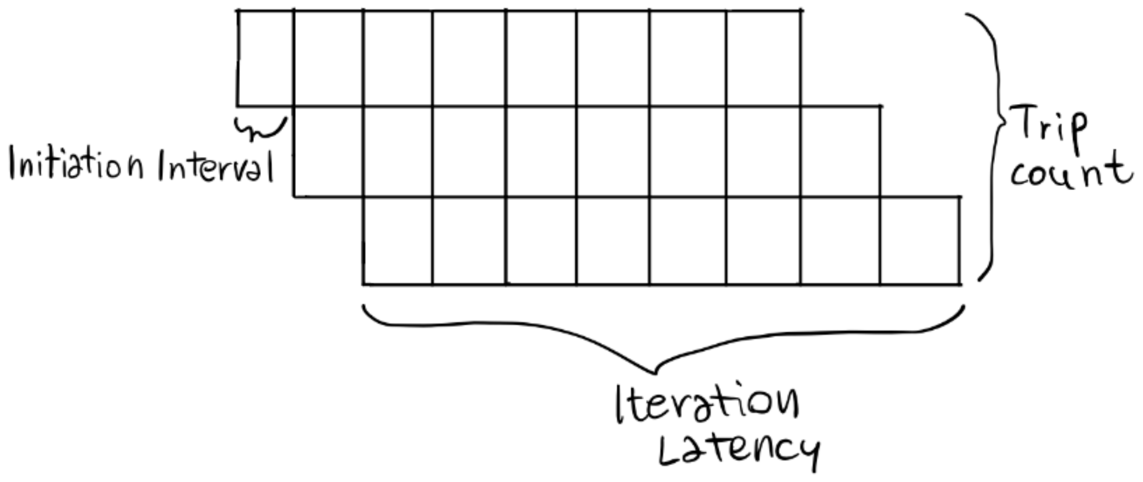


Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
kernel				-	-	-	-	-	-	no	64	32	15541	43022	0
init_loop				-	12048	1,200E5	753	-	16	no	-	-	-	-	-
in_init_loop				-	720	7,200E3	45	-	16	no	-	-	-	-	-
while_loop				-	-	-	30522	-	-	no	-	-	-	-	-
first_loop				-	19432	1,940E5	1388	-	14	no	-	-	-	-	-
second_loop				-	1974	1,974E4	141	-	14	no	-	-	-	-	-
third_loop				-	5110	5,110E4	365	-	14	no	-	-	-	-	-
fourth_loop				-	3990	3,990E4	285	-	14	no	-	-	-	-	-

We tried to deactivate all automatic optimizations with the command HLS PIPELINE off. We can see the results in the above picture, where it is clear that the software is not doing pipelines in any loops.

Here the total number of clock cycles is shown, depending on the architecture and the clock period; the iteration latency, i.e., how many cycles a single iteration has; and trip count reports the number of pipeline rows our RTL code has.

Interval stands for initiation interval, and it represents the number of operations the program will wait to start the execution of the next pipeline.



We know that the software Vitis automatically does the optimizations it is able to do. If it is not possible, Vitis will generate a warning indicating what type of problem it has found.

3.2 State-of-the-art

The state-of-the-art of the Poisson 2D problem solved with FPGA architecture is not very developed. In fact, GPU and CPU parallel purposes are often more efficient.

However, there are papers about hardware implementation (Vivado or VHDL) that exploit the memory locality and compare the power consumption between different architectures.

There is also an article about the V-cycle Multigrid method, where it is shown the robustness of this solver compared to the Jacobi and SOR methods.

That is interesting, even if we know that Jacobi is not so efficient, mostly when considering big problems, because solutions are not updated following the residual.

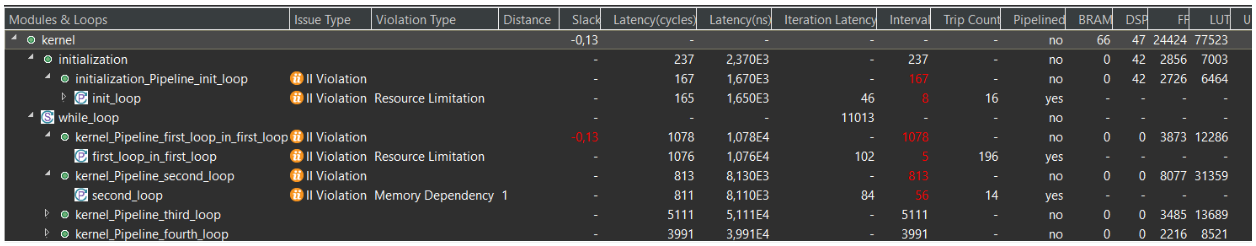
We decided to use Jacobi because of the easier implementation and the reduced matrix size we are going to use.

3.3 Optimization

We implemented three types of optimization

- pragma pipeline and loop unrolling
- code and memory optimization
- fixed-point variables

If we try to activate all the optimizations in the while_loop, we get this result from the synthesis.

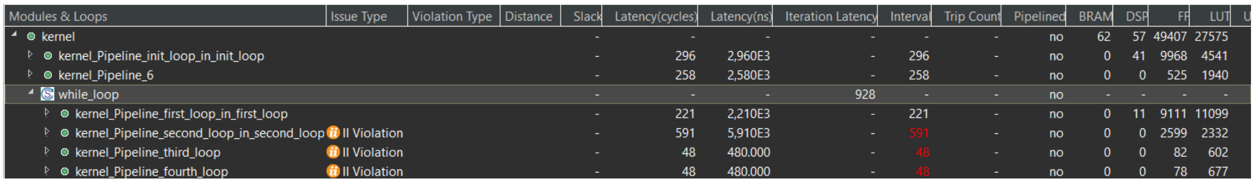


Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	U
kernel				-0,13	-	-	-	-	-	no	66	47	24424	77523	
initialization				-	237	2,370E3	-	237	-	no	0	42	2856	7003	
initialization_Pipeline_init_loop	II Violation			-	167	1,670E3	-	167	-	no	0	42	2726	6464	
init_loop	II Violation	Resource Limitation		-	165	1,650E3	46	8	16	yes	-	-	-	-	
while_loop				-	-	-	11013	-	-	no	-	-	-	-	
kernel_Pipeline_first_loop_in_first_loop	II Violation			-0,13	1078	1,078E4	-	1078	-	no	0	0	3873	12286	
first_loop_in_first_loop	II Violation	Resource Limitation		-	1076	1,076E4	102	5	196	yes	-	-	-	-	
kernel_Pipeline_second_loop	II Violation			-	813	8,130E3	-	813	-	no	0	0	8077	31359	
second_loop	II Violation	Memory Dependency	1	-	811	8,110E3	84	56	14	yes	-	-	-	-	
kernel_Pipeline_third_loop				-	5111	5,111E4	-	5111	-	no	0	0	3485	13689	
kernel_Pipeline_fourth_loop				-	3991	3,991E4	-	3991	-	no	0	0	2216	8521	

There has been a significant improvement in the performances, but we have some warnings and memory dependencies those can make our program slow or can cause errors in Co-Simulation phase.

In order to exploit some properties of the code and remove the data dependencies, we decided to partition the arrays into single variables with the directive $\mathrm{HLS\ ARRAY\_PARTITION\ variable{=}{<}namevar{>}\ type{=}complete\ dim{=}1}$.

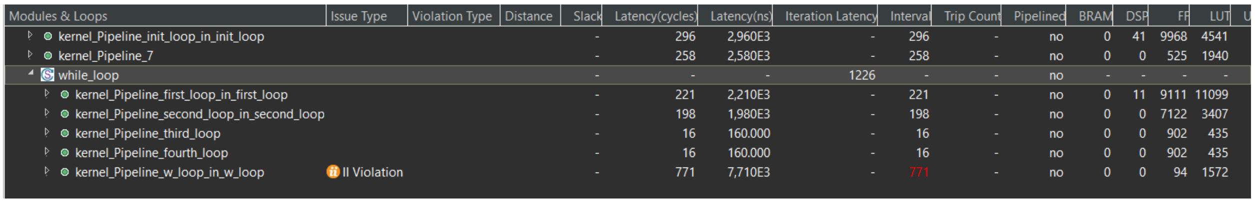
To correctly partition the array and consider variables, we need to copy the solution into a new array in the last codeline, and so change the first kernel input.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	U
kernel				-	-	-	-	-	-	no	62	57	49407	27575	
kernel_Pipeline_init_loop_in_init_loop				-	296	2,960E3	-	296	-	no	0	41	9968	4541	
kernel_Pipeline_6				-	258	2,580E3	-	258	-	no	0	0	525	1940	
while_loop				-	-	-	928	-	-	no	-	-	-	-	
kernel_Pipeline_first_loop_in_first_loop				-	221	2,210E3	-	221	-	no	0	11	9111	11099	
kernel_Pipeline_second_loop_in_second_loop	II Violation			-	591	5,910E3	-	591	-	no	0	0	2599	2332	
kernel_Pipeline_third_loop	II Violation			-	48	480.000	-	48	-	no	0	0	82	602	
kernel_Pipeline_fourth_loop	II Violation			-	48	480.000	-	48	-	no	0	0	78	677	

Instead of calculating w in the same loops with the array v, we decided to compute that in a separate loop.

In this way, Vitis will be free to optimize all the loops containing the array v without data dependencies.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	U
kernel_Pipeline_init_loop_in_init_loop				-	296	2,960E3	-	296	-	no	0	41	9968	4541	
kernel_Pipeline_7				-	258	2,580E3	-	258	-	no	0	0	525	1940	
while_loop				-	-	-	1226	-	-	no	-	-	-	-	
kernel_Pipeline_first_loop_in_first_loop				-	221	2,210E3	-	221	-	no	0	11	9111	11099	
kernel_Pipeline_second_loop_in_second_loop				-	198	1,980E3	-	198	-	no	0	0	7122	3407	
kernel_Pipeline_third_loop				-	16	160.000	-	16	-	no	0	0	902	435	
kernel_Pipeline_fourth_loop				-	16	160.000	-	16	-	no	0	0	902	435	
kernel_Pipeline_w_loop_in_w_loop	II Violation			-	771	7,710E3	-	771	-	no	0	0	94	1572	

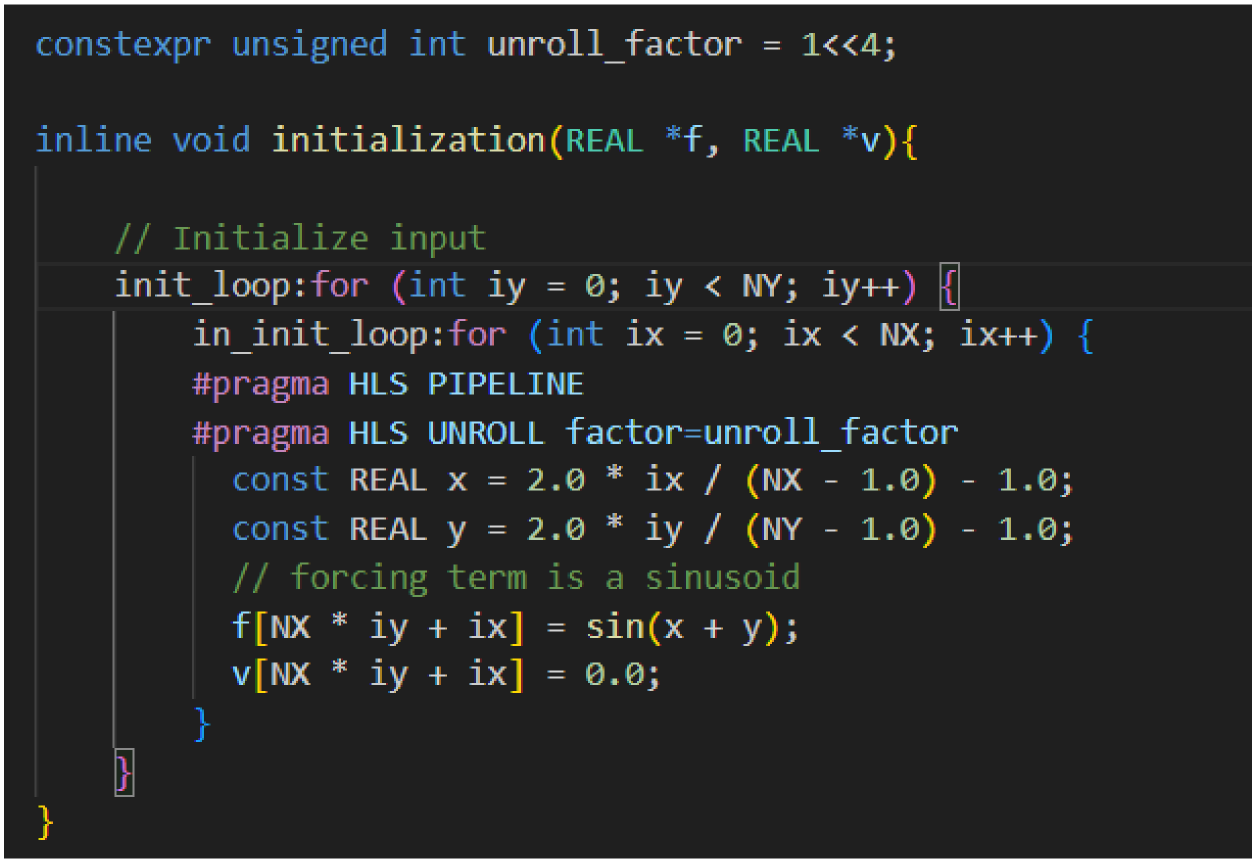
To optimize also the loop on w, we decided to accumulate partial sums of the rows in a new variable w_local using a completely unrolled loop and then assemble w in the more external loop.

```
REAL w = 0.0;
w_second_loop: for (unsigned int iy = 0; iy < (NY*NX / unroll_factor); iy++)
{
    REAL w_local = 0.0;
    w_in_second_loop:for (unsigned int ix = 0; ix < unroll_factor; ix++)
    {
    #pragma HLS UNROLL
        w_local += std::fabs(v[unroll_factor*iy+ix]);
    }
    w += w_local;
}
```

Watching the results, we noticed that the error calculation required a lot of work.

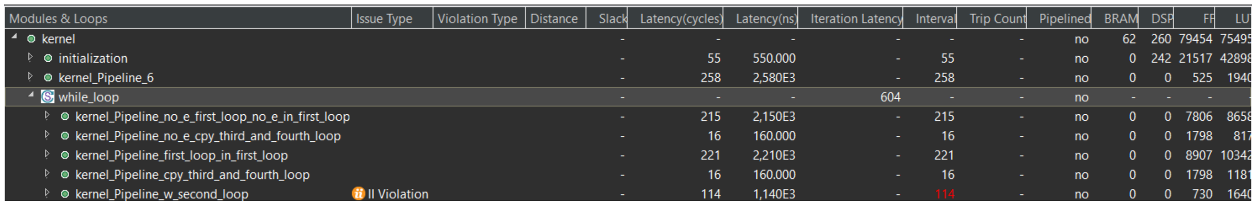
We decided to compute the error once every two iterations, and this allowed us to use a ping-pong buffer instead of copying the full array at each iteration.

At this point, the last part that we could optimize with pragmas was the initialization. To exploit the board space we've also unrolled that loop.



```
constexpr unsigned int unroll_factor = 1<<4;

inline void initialization(REAL *f, REAL *v){

    // Initialize input
    init_loop:for (int iy = 0; iy < NY; iy++) {
        in_init_loop:for (int ix = 0; ix < NX; ix++) {
        #pragma HLS PIPELINE
        #pragma HLS UNROLL factor=unroll_factor
            const REAL x = 2.0 * ix / (NX - 1.0) - 1.0;
            const REAL y = 2.0 * iy / (NY - 1.0) - 1.0;
            // forcing term is a sinusoid
            f[NX * iy + ix] = sin(x + y);
            v[NX * iy + ix] = 0.0;
        }
    }
}
```

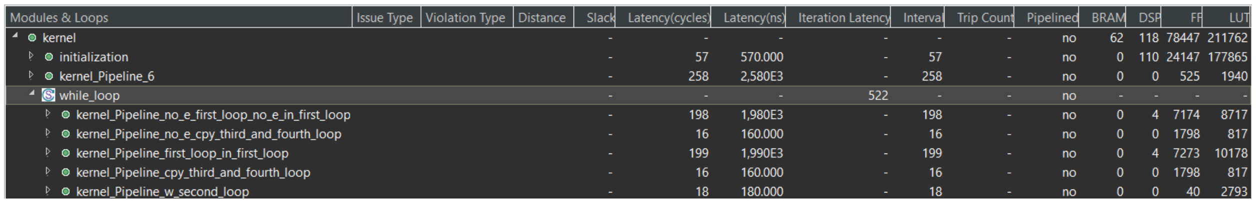
Adding these directives, we can see immediately some improvement in the initialization function.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT
kernel				-	-	-	-	-	-	no	62	260	79454	75495
initialization				-	55	550.000	-	55	-	no	0	242	21517	42898
kernel_Pipeline_6				-	258	2,580E3	-	258	-	no	0	0	525	1940
while_loop				-	-	-	604	-	-	no	-	-	-	-
kernel_Pipeline_no_e_first_loop_no_e_in_first_loop				-	215	2,150E3	-	215	-	no	0	0	7806	8658
kernel_Pipeline_no_e_cpy_third_and_fourth_loop				-	16	160.000	-	16	-	no	0	0	1798	817
kernel_Pipeline_first_loop_in_first_loop				-	221	2,210E3	-	221	-	no	0	0	8907	10342
kernel_Pipeline_cpy_third_and_fourth_loop				-	16	160.000	-	16	-	no	0	0	1798	1181
kernel_Pipeline_w_second_loop	II Violation			-	114	1,140E3	-	114	-	no	0	0	730	1640

As we can see, both latency in cycles and iteration latency gained one order of magnitude.

The last optimization technique we tested is to use fixed-point variables and fixed-point arithmetic operations. Knowing that fixed-point have a limited precision we decided to create an auxiliary method to compute the error between the solution obtained using double and that one using fixed-point. We implemented the fixed-point representation with its enclosed functions, and at the end the performance results are shown in the picture below.



Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT
kernel				-	-	-	-	-	-	no	62	118	78447	211762
initialization				-	57	570.000	-	57	-	no	0	110	24147	177865
kernel_Pipeline_6				-	258	2,580E3	-	258	-	no	0	0	525	1940
while_loop				-	-	-	522	-	-	no	-	-	-	-
kernel_Pipeline_no_e_first_loop_no_e_in_first_loop				-	198	1,980E3	-	198	-	no	0	4	7174	8717
kernel_Pipeline_no_e_cpy_third_and_fourth_loop				-	16	160.000	-	16	-	no	0	0	1798	817
kernel_Pipeline_first_loop_in_first_loop				-	199	1,990E3	-	199	-	no	0	4	7273	10178
kernel_Pipeline_cpy_third_and_fourth_loop				-	16	160.000	-	16	-	no	0	0	1798	817
kernel_Pipeline_w_second_loop				-	18	180.000	-	18	-	no	0	0	40	2793

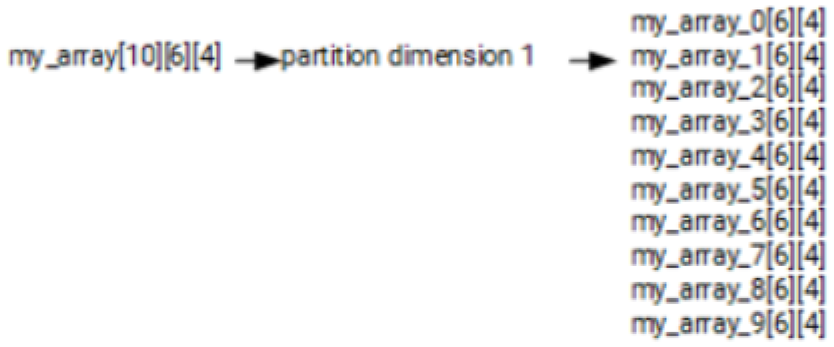
As excepted, this is the best results we have obtained.

4️⃣

FINAL RESULTS

Starting from the original code without any optimization, we gained two orders of magnitude.

The most visible improvements were found when applying the array partition; this technique optimizes the memory accesses and leads to faster usage of the FPGA on the array values. The design trade-off is between performance and the number of RAMs or registers required to achieve it.



Another important improvement has been made using fixed-point variables.

In this case, we gained almost 17% in performance because of the hardware resource savings and the easier computation. Using variable type $\mathrm{ap\_fixed}{<}32{,}5{>}$, i.e. numbers with 32 total digits and 5 integer digits, the iterative method is going to do approximately 1% more iterations, but it is also going to compute faster.

The program, using fixed-point representation, seems to be able to correctly manage the data dependency in the loop for the calculation of w.

Unfortunately, we were not able to synthesize the code with a matrix size of 32x32 or more, so we are not sure about the RTL behavior in a larger dimension.

Some eventual extensions of our problem should be executing the RTL code on a physical board; Amazon Web Services offers some FPGAs under payment, which should make it interesting to analyze the scalability and portability of the code.

A comparison between FPGA and GPU, for example, should be less interesting because the GPU is definitely faster when you work in parallel. Conversely, FPGAs are used when you want to introduce redundancy to the data and where energy consumption is more important.