FPGA101 project report

Andrea Oggioni

January 2025

Abstract

This report is about the implementation of a *Sparse Matrix-Vector Multiplication* kernel (from now on, *SpMV*) packaged into an IP, using the *Vitis HLS* software from Xilinx [6].

The IP will be instantiated into the *PL fabric* of a Pynq-Z2 board [4] and the kernel will be called through python using the *pynq* library [5].

This report covers the final assignment for the **FPGA101: from reconfigurable to domain-specific systems**[3] course, held by Davide Conficconi and Giuseppe Sorrentino at NecstLAB [7].

Considerations about the various issues encountered during the development of the project will also be included at the end of the document.

1 Algorithm

The SpMV algorithm is an alternative implementation of the conventional matrix-vector multiplication that avoids unnecessary computations by skipping all the products and sums associated with the zeroes in the matrix [2].

The algorithm speed is proportional to the sparsity of the matrix.

The SpMV algorithm requires that the matrix is encoded in *Compressed Row Storage* (from now on, *CRS*) format. This type of encoding consists in keeping only the non-zero elements of the matrix in memory, reducing the amount of memory needed.

The memory footprint is inversely proportional to the sparsity of the matrix.

The CRS encoding of a matrix consists of three vectors a, c and r.

- a is the vector that contains all the non-zero element of the matrix;
- c is the vector that, for each non-zero element of the matrix, stores its column;
- r is the vector that stores the indexes of the previous two vectors where the data for each row begins.

As an example, let a, c and r be three 0-indexed vectors such that

$$a = \begin{pmatrix} 3 & 4 & 5 & 9 & 2 & 3 & 1 & 4 & 6 \end{pmatrix}$$

$$c = \begin{pmatrix} 0 & 1 & 1 & 2 & 0 & 2 & 3 & 1 & 3 \end{pmatrix}$$

$$r = \begin{pmatrix} 0 & 2 & 4 & 7 & 9 \end{pmatrix}$$

.

To reconstruct the original matrix, one has to proceed row by row. To reconstruct the i-th row (0-indexed) one has to slice the a and c vector as follows

$$\bar{a} = [a_k]_{r_i \leq k < r_{i+1}}$$

$$\bar{c} = [c_k]_{r_i \leq k < r_{i+1}}$$

.

The row is an horizontal vector where each element is defined as

$$row_j = \begin{cases} \bar{a}_t & j \in \bar{c}, t = \text{index of } j \text{ in } \bar{c} \\ 0 & \text{otherwise} \end{cases}$$

.

Repeating the previous procedure for each row gives the following matrix:

$$\begin{bmatrix} 3 & 4 & 0 & 0 \\ 0 & 5 & 9 & 0 \\ 2 & 0 & 3 & 1 \\ 0 & 4 & 0 & 6 \end{bmatrix}$$

.

Let a, c and r be three vectors that encode an $n \times m$ matrix A and x another vector, the product $y = Ax$ is defined as

$$y_i = \sum_{k=r_i}^{r_{i+1}} a_k \cdot x_{c_k} \qquad \forall i \in 0, \ldots, n-1$$

.

The implemented kernel is an implementation of the following pseudocode procedure:

Algorithm 1 SpMV algorithm

```
procedure SpMV(a, c, r, n) ▷ a, c and r encode a matrix in CRS format, n is the matrix width.
    for i ← 0 to n − 1 do
        y_i ← 0
        for k ← r_i to r_{i+1} do
            y_i ← y_i + a_k · x_{c_i}
        end for
    end for
    return y
end procedure
```

2 Setup

All the work was done on a Windows 10 machine running *Xilinx Design Tools Vitis Unified Software Platform* version **2024.1**.

Vitis HLS will be used to create the HLS component that will be included in the block diagram in Vivado.

Both the Vivado project and the Vitis project will target the *ZYNQ XC7Z020-1CLG400C* SoC.

The following flags were added to the compiler options: `-Wall -Wextra`. The *Address sanitizer* and *Undefined sanitizer* were enabled.

3 Procedure

3.1 High Level Synthesis

The High Level Synthesis phase consists of 5 steps, of which only the first 4 were needed during development.

The *C Simulation* phase consisted of the implementation and testing of the SpMV kernel.

More details about the testing of the kernel in the next section.

To speed up the computation, each element of the resulting vector is computed in parallel (see Figure 1) by unrolling the loop in 10 independent *units*. Let n be the width of the matrix to multiply (and also the height of the resulting vector), only the first n units are active during each kernel call.

```
void SpMV(
    int values[MAX_MATRIX_ELEMENTS],
    ColumnIndex columnIndexes[MAX_MATRIX_ELEMENTS],
    RowPointer rowPointers[MAX_MATRIX_SIDE_SIZE + 1],
    VectorSize numOfRows,
    int vector[MAX_MATRIX_SIDE_SIZE],
    int output[MAX_MATRIX_SIDE_SIZE]
) {
    spmv_loop_external:
        for(VectorSize i = 0; i < MAX_MATRIX_SIDE_SIZE; i++) {
      #pragma HLS UNROLL
      int sum = 0;
      if(i < numOfRows) {
        spmv_loop_internal:
            for(
                ValuesSize j = rowPointers[i];
                j < rowPointers[i + 1];
                j++
            ) {
          #pragma HLS PIPELINE II=1
          int matrix_value = values[j];
          ColumnIndex column_index = columnIndexes[j];
          int vector_value = vector[column_index];
          int temp = matrix_value * vector_value;
          sum += temp;
        }
      }
      output[i] = sum;
    }
}
```

Please note that the `#pragma HLS interface` macros were omitted from the listing above.

Figure 1: The callgraph

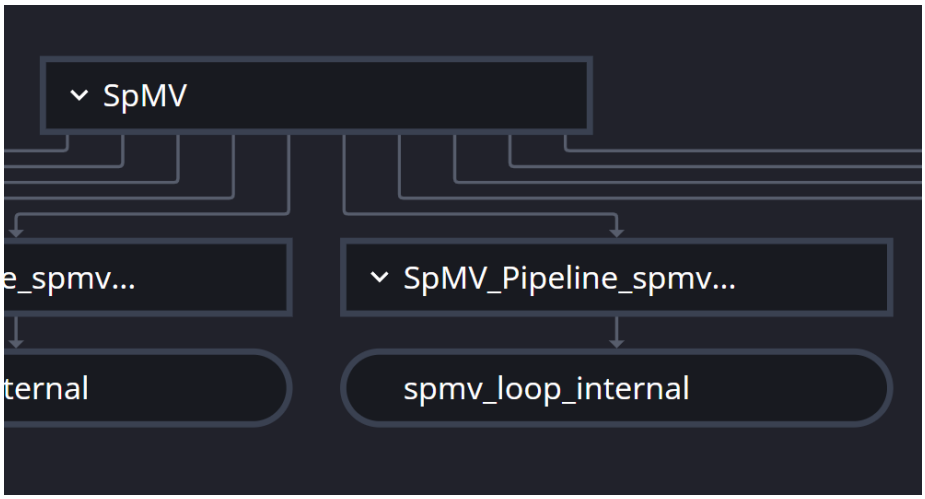

Figure 2: Zoom on a single *unit* in the callgraph

To speed up even more the computation, the inner loop is pipelined.

A local variable SUM is used as an accumulator to avoid unnecessary reads and writes to and from the memory.

All the code is available on Github[8].

After the *C Simulation* phase, there is the *C Synthesis* phase, in which the code written in the previous phase gets synthesized. After completing this phase successfully, it's possible to take a look at the callgraph, and all the interfaces and registers created and at how the loops were pipelined and/or optimized.

After the *C Synthesis* phase, there is the *C/RTL Cosimulation* phase, in which the C code and the results of the synthesis are run together side by side to check that the kernel would work on the target FPGA.

The final phase is the *Package* phase, in which the IP is packaged with all the necessary files so that it's ready to be imported into Vivado.

3.2 Testbench

To ensure the correct functionality of the IP, testing automation was implemented.

The testbench is composed by multiple testcases that are loaded from a text file.

The file containing the testbench is formatted as follows:

```
[Number of testcases in file]
[Testcase 1]
[Testcase 2]
...
```

Each testcase is formatted as follows:

```
[Testcase name]
[Matrix width] [Matrix height]
[Length of values vector]
[Vector of values]
[Vector of column indexes]
[Vector of row pointers]
[Expected product]
```

Each testcase is loaded and run sequentially. The outcome of each testcase are printed on the standard output.

A testcase generator written in Python is also available in the repository.

3.3 Block diagram

After importing the IP into Vivado, it's necessary to create a block diagram to describe how the IP will be connected to the hard core of the Pynq-Z2.

The block diagram for the current project is available in Figure 3 on page 8).

The *ZYNQ7 Processing System* block (that represents the CPU of the board) is connected to the *SpMV_0* block (that represent the implemented IP) through two *AXI Interconnect* block.

The left *AXI Interconnect* block is used to manage the IP (starting, stopping, polling status and passing parameters over *axilite*) while the right one is used by the IP to access the data stored in the DDR RAM.

The IP can access the DDR RAM through an *High Performance* port.

3.4 Upload to ZYNQ

Before proceeding to upload the bitstream file and the hardware wrapper file, it was necessary to configure the networking on the Pynq board. In order to do so, the board was connected to the computer over USB. After opening a serial connection and getting access to a terminal, all the lines in the `/etc/network/interfaces.d/eth0` file were commented out and the `eth0` interface was configured as follows:

```
auto eth0
iface eth0 inet static
address 192.168.1.80
netmask 255.255.255.0
gateway 192.168.1.1
dns-nameservers 1.1.1.1 1.0.0.1
```

After the modification, the new configuration was applied with the following command: `/etc/init.d/networking restart`[9].

After the application of the modification, it is possible to reach the Jupyter Notebook running on the board by opening `192.168.1.80` with a browser of choice.

Calling the kernel from Python is quite straightforward: first the buffers containing the vectors of values accessed by the fpga are allocated and populated with a combination of `np.array` and `pynq.allocate`, then the register map was configured to assign the correct values to the parameter of the kernel.

Making the kernel run consists in setting `spmv.register_map.CTRL.AP_START = 1` where `spmv` is the instance of `pynq.overlay.DefaultIP` used to call the kernel.

After starting the kernel, it's necessary to poll `spmv.register_map.CTRL.AP_DONE` until it gets the value 1: this means that the kernel has finished its execution.

After the kernel has finished, the output is read from memory, put into a `np.array` and compared with the expected result to show that the calculation was correct.

4 Problems encountered

In this section there will be the description of the various issues encountered over the development of this project and their respective solutions.

Please note that most of the code that was causing issues was replaced with entirely different logic in later revisions of the project.

4.1 C Synthesis takes a really long time

Cause: unroll of a 2^{18} iterations for-loop.

Fix: remove loop or pipeline it.

Comment: unrolling a loop this big needs a lot of resources both from the FPGA and from the machine synthesizing the project.

4.2 Cannot read from same memory port on two different dataflow processes

Cause: reading two variables at the same time from the same memory port.

Fix: use a different memory port for each variable.

4.3 No templated top function

Cause: use of template on the top function.

Fix: make the top function signature not generic.

Comment: the signature of the top function describes the physical connections between the IP and the rest of the PL fabric so it cannot be generalized. Also, the synthesis process requires a specialized top function, not a generic one.

4.4 stdout skipping lines, IDE lagging, keyboard keys not working

Cause: unknown.

Fix: unknown.

Comment: the Vitis IDE randomly starts lagging so much that arrow keys wouldn't work anymore and the tab key would work with a long delay (tens of seconds). The only way to read the full output of the simulation is to pipe stdout into a file.

4.5 Git integration won't work

Cause: unknown.

Fix: initialize the repository manually from outside Vitis.

Comment: other users have reported having the same issue but no permanent fix was found at the time of writing this [1].

4.6 Never before seen files not found

Cause: renaming source files from Vitis doesn't update project settings.

Fix: remove non-existing files from project settings.

4.7 Address sanitizer crashing

Cause: unknown, possibly an `ap_uint` overflow in *C Simulation*.

Fix: prevent variable from overflowing (for example by increasing the number of bits if it's a suitable solution).

4.8 TB post transaction check failed

Cause: macro used as the `depth` parameter in `#pragma HLS INTERFACE` instead of a `const int`.

Fix: use `const int` instead of macro.

Comment: this was the error that took me the most time to fix because I didn't understand why it was happening. I assume that a variable must be used, and not a macro, because in this way, it's *wired* into the PL and can be used during the transfers. It's interesting to note that this using a macro does not make the synthesis fail or cause any warn/error emission.

4.9 Unused parameters makes cosimulation fail

Cause: unused parameter in top function signature.

Fix: remove unused parameter.

Comment: while removing the unused argument from the signature fixed the issue, I couldn't figure out the correlation between the non-usage of a parameter and the cosimulation fail (maybe some optimization by the compiler on the C side?).

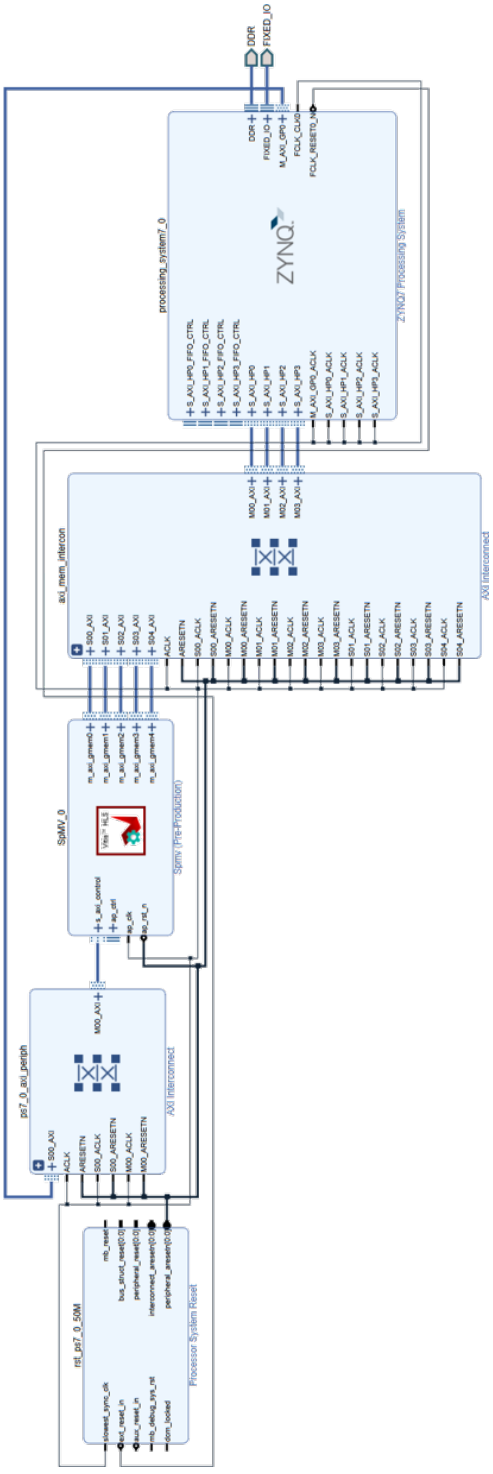

Figure 3: The block diagram with the various blocks and the connections between them.

References

[1] 263580ergzdezde. Problems when i try to use git integration in vitis unified ide. version: Vitis 2023.2 on ubuntu 22.04. report error: "sign in failed: Error: Invalid scheme 'theia'". https://adaptivesupport.amd.com/s/question/0D54U0000080lRYhSAM/problems-when-i-try-to-use-git-integration-in-vitis-unified-ide-version-vitis-20232-on-ubuntu-2204-report-error-sign-in-failed-error-invalid-scheme-theia?language=en_US.

[2] Rob H. Bisseling. Sequential sparse matrix vector multiplication. https://www.youtube.com/watch?v=0ig6rRc5ewY.

[3] Davide Conficconi and Giuseppe Sorrentino. Fpga101: From reconfigurable to domain-specific systems. https://www.polimi.it/formazione/passion-in-action/dettaglio/fpga101-from-reconfigurable-to-domain-specific-systems.

[4] Xilinx Inc. Aup pynq-z2. https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html.

[5] Xilinx Inc. Pynq: Python productivity for adaptive computing platforms. https://pynq.readthedocs.io/en/latest/index.html.

[6] Xilinx Inc. Vitis hls. https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[7] NECSTLab. Necstlab: Advanced projects in computing systems. https://necst.it/.

[8] Andrea Oggioni. fpga spmv. https://github.com/etabeta1/fpga_SpMV.

[9] tim11g and Dave. How do i add an additional ip address to an interface in ubuntu 14. https://askubuntu.com/questions/585468/how-do-i-add-an-additional-ip-address-to-an-interface-in-ubuntu-14.