Accelerating the execution of a program to solve the partition problem using an FPGA

Pratik Shrestha
Grand Valley State University

Accelerating the execution of a program to solve the partition problem using an FPGA

Pratik Shrestha, Chirag Parikh and Christian Trefftz
Grand Valley State University
1 Campus Drive
Allendale MI 49401 USA

Abstract

Solving the partition problem is a time consuming endeavor. The execution time required to find a solution to instances of the partition problem is reduced using a Field Programmable Gate Array (FPGA). We used the PYNQ board from Xilinx and created an overlay to accelerate the execution of a function that evaluates if a partition is a solution to an instance of the partition problem.

1 Introduction

Parallel processing has become an important topic in Computing. Microprocessors come with several cores and many computers now have Graphical Processing Units (GPUs). Field Programmable Gate Arrays (FPGAs) can be used as accelerators and they can be very efficient in energy consumption [1].

Xilinx has created a board for pedagogical purposes called PYNQ that can be easily programmed using Python. The board comes with a Xilinx FPGA device that has a built-in microprocessor. The FPGA also has programmable fabric.

PYNQ boards can be connected to a host computer via an USB cable and, optionally, an Ethernet cable. The board is, basically, a small independent computer that runs a custom version of Linux. It comes with a C compiler (gcc) and a python interpreter as well as a web server. Jupyter notebooks on the PYNQ board can be accessed from the host computer, simplifying the interaction with the board.

"Overlays" can be created with the Vivado environment from Xilinx to take advantage of the programmable portion of the array. An overlay can be loaded, dynamically, into the FPGA from a python script. Xilinx and several research groups have made available several overlays that highlight the use of FPGAs as accelerators.

In this paper, we describe the process of creating an overlay to accelerate the execution of a python program that finds a solution to the partition problem, a problem that belongs to the "NP-complete" category of problems. Algorithms to find exact solutions to problems in this category are very time consuming. As the size of the problems grows, the time required to solve these problems precisely grows exponentially. We present the results of timing the execution of several different instances of the problem.

Our main goal is to illustrate how an FPGA can be used to accelerate the execution of an algorithm that solves exactly an NP-complete problem. This solution should be of interest in academic settings where PYNQ boards are used.

The rest of this paper is structured as follows: The partition problem is described in section 2. A "brute force" approach to solve the Partition problem is outlined in section 3. The process of creating the overlay is described in section 4. Experimental results are included in section 5. The conclusions are in section 6.

2 The Partition Problem

Given a multi-set of positive integers S, can it be partitioned into two sub multi-sets S_1 and S_2 such that the sum of the elements is S_1 is equal to the sum of the elements in S_2?

Consider the following example: Let S be the multi-set {4,5,9}. In this particular case it is evident that the answer to the problem is yes: We partition the multi-set into two sub multi-sets S_1 : {4,5} and S_2 : {9}.

The partition problem is one of the simplest NP-complete problems. NP-complete problems are very interesting for several reasons. Many real-life applications can be modeled as NP-complete problems and it is important for software developers to understand the limitations of existing algorithms that can solve those problems. Exact algorithms can find solutions, in a reasonable amount of time, only for small instances of these problems. Large instances of NP-hard problems will take so long to solve with exact algorithms, that for practical purposes those large instances should be considered intractable. Other alternatives are available (heuristics, approximation algorithms) but the solutions produced by these alternatives are likely to be sub-optimal.

3 An Exact Algorithm to Solve the Partition Problem

Woeginger [3] (page 4) has observed that there is a subset of NP-complete problems that can be solved by brute-force by enumerating exhaustively all the possible subsets (the power set) of a particular set of elements.

If we wanted to explore the power set of the multi-set S, we could do it by observing that the binary representation of the integers between 1 and $2^{n-1} - 1$ encode the possible subsets of interest. Notice that the other values between 2^{n-1} and $2^n - 2$ are symmetrical to the values considered.

Table 1 illustrates the values for the example in the previous section: $S = \{4,5,9\}$. The indices for the different encodings of the subsets are listed on the first column, Index, on Table 1. The binary encoding is listed on the second column. The rightmost digit encodes to the subset to which element 1 belongs, the middle digit encodes the subset to which element 2 belongs and the leftmost digit encodes the subset where node 3 belongs. Take the entry that corresponds to 3: 011. This is interpreted as subset 1 (encoded by 0) containing element 3 and subset 2 (encoded by 1) containing elements 1 and 2. The table contains all the integers between 0 and $7(2^3 - 1)$, but it is not necessary to consider the value 0, nor the value 7. Observe that the values between 0 and 3 are symmetrical to the values between 4 and 7; the values are each other's complements, 1 (001) is the complement of 6 (110), 2 (010) is the complement of 5 (101), and 3 (011) is the complement of 4 (100).

Index	Binary Encoding	Solution?
0	000	No
1	001	No
2	010	No
3	011	Yes
4	100	Yes
5	101	No
6	110	No
7	111	No

Table 1: Indices, subsets, and solutions for an instance of the partition problem.

Notice that the set of possible subsets of interest is encoded by the set of integers in the range between 1 and $2^{n-1} - 1$. As soon as an algorithm finds a possible partition of the multiset, the algorithm can stop and the answer for this particular instance of the problem is yes. If all possible partitions are considered and no possible satisfying partition is found, the answer for this particular instance of the problem is no.

The outline of the main algorithm is shown in 1.

Algorithm 1: Algorithm to solve instances of the Partition problem

```
Input: n is the size of the problem, array contains the values
Output: true or false
indexOfPossiblePartition=1;
while indexOfPossiblePartition ≤ $2^{n\ \ 1} - 1$ do
    if evaluatePossiblePartition(indexOfPossiblePartition,n,array) then
        Return true;
    else
        indexOfPossiblePartition++;

Return false;
```

The outline of the function *evaluatePossiblePartition* is presented in Algorithm 2.

Algorithm 2: Algorithm to solve instances of the Partition problem

```
Input: n, array that contains the values, indexOfPossiblePartition
Output: true or false
sumOfValuesInPartition0 = 0;
sumOfValuesInPartition1 = 0;
index=0;
while index<n do
    if $bit_{index}$ In the binary representation of indexOfPossiblePartition Is 0 then
        sumOfValuesInPartiton0+=array[index]
    else
        sumOfValuesInPartiton1+=array[index]

if sumOfValuesInPartition0 = sumValuesInPartition1 then
    Return true;
else
    Return false;
```

As the size of the problem grows, in other words, as the value of n grows, the time required to find an exact solution grows exponentially. The order of this algorithm is $O(2^n)$.

Most of the execution time of the program is spent in the function that evaluates if a particular partition is a solution for the problem. In the work described here, we use the programmable fabric of the FPGA to accelerate the execution of that function.

Notice that the evaluation of each possible partition can be carried out independently from the evaluation of the other possible partitions. On computing platforms with several processors, every processor can evaluate a possible partition in parallel with other processors evaluating other possible partitions [2].

4 Creating an Overlay

Overlays, also known as Hardware libraries, are programmable/configurable FPGA designs that extend the user application from the Processing System into the programmable logic [4]. They are extremely useful to accelerate a piece of software using a hardware platform for a particular application. The software programmer can use an overlay in a similar way to a software library to run some of the applications on an FPGA as overlays can be loaded into the FPGA dynamically. This allows software programmers to take advantage of FPGA capabilities without having detailed knowledge about the low-level hardware design. All they must worry about is the top-level program.

Creating an IP using High Level Synthesis (HLS) is the very first step required to create a custom overlay. For the HLS of this project, Xilinx's Vivado HLS was used. Different pragmas were inserted in a C program to boost the efficiency. After the successful creation of Intellectual Property (IP) core, the IP component is imported into the Vivado Suite. In the block diagram, the Zynq processor is connected to the custom IP as shown in figure 1 below. For this work, the High-performance AXI bus is chosen explicitly to boost up the execution. After successful synthesis of the overlay, the bitstream is then generated. This step produces .BIT and .HWH files which are then stored in the working directory inside the PYNQ board.

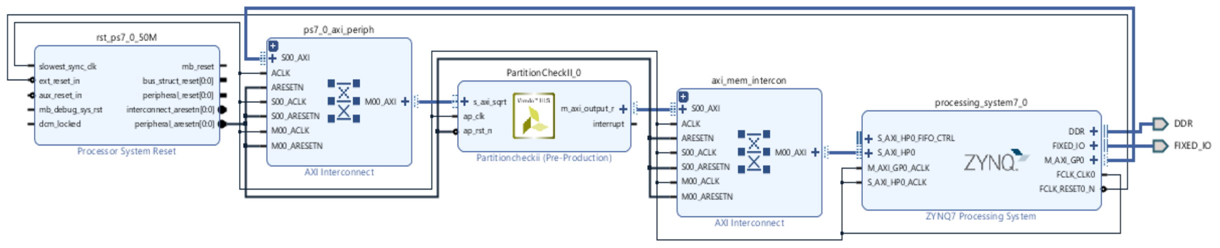
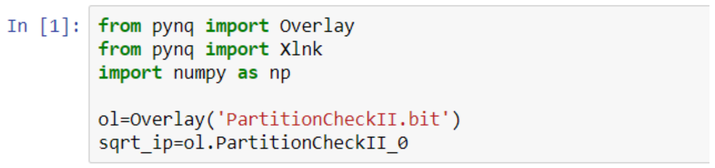


Fig 1: Block Design of the overlay

To interact with the IP, first the overlay must be loaded into the Jupyter notebook which contains the IP. The PYNQ board must be physically connected to the PC for this step as all the rest of the process will be done in PYNQ board. This step has been depicted in figure 2 below. Here, the overlay "PartitionCheckII" has been imported. Then the next line indicates that the overlay consists of an IP "PartitionCheckII_0" which is the IP of interest here.



```
In [1]: from pynq import Overlay
        from pynq import Xlnk
        import numpy as np

        ol=Overlay('PartitionCheckII.bit')
        sqrt_ip=ol.PartitionCheckII_0
```

Fig 2: Import Overlay

This overlay can be thought as a block, as shown in figure 3, which takes an array as the input and produces single output, 1 or 0, indicating if the given numbers can be partitioned or not. The very first element of the array indicates the total numbers present in the array.

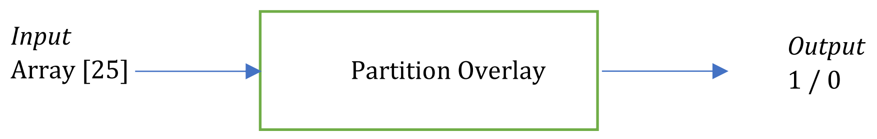
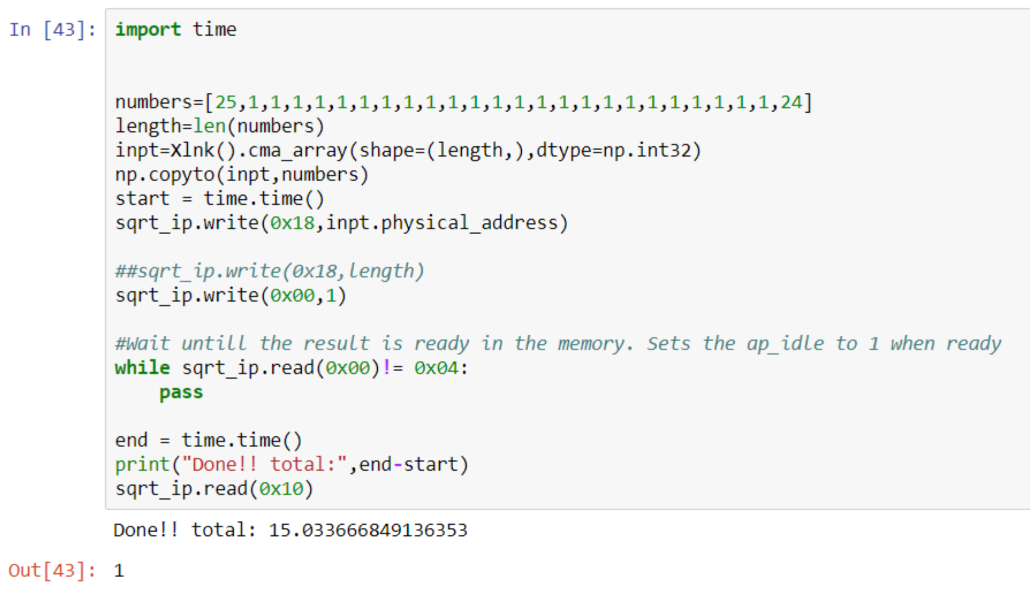


Fig 3: Overlay Block

As can be seen clearly from the overlay block in figure 3, there are two ports in total. Each of them has their own physical memory address as MMIO (Memory Mapped Input Output) is used here for I/O operation. In the code snipped shown in figure 4 below, the address 0x18 is used as the input address for the array and 0x10 is used as output address. The bit value 1 in the address 0x00 indicates beginning of the process.

Execution time is another key point for the work as the main goal is to accelerate the partition problem using FPGA. To measure the execution time, the ***"time"*** module is imported and used. In figure 4, the variable "numbers" include 26 numbers of which the first number, 25, denotes that there are total 25 numbers which are to be partitioned. The output "Done" indicates the execution has been completed with the time consumption of 15.03 seconds.



```
In [43]: import time


         numbers=[25,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,24]
         length=len(numbers)
         inpt=Xlnk().cma_array(shape=(length,),dtype=np.int32)
         np.copyto(inpt,numbers)
         start = time.time()
         sqrt_ip.write(0x18,inpt.physical_address)

         ##sqrt_ip.write(0x18,length)
         sqrt_ip.write(0x00,1)

         #Wait untill the result is ready in the memory. Sets the ap_idle to 1 when ready
         while sqrt_ip.read(0x00)!= 0x04:
             pass

         end = time.time()
         print("Done!! total:",end-start)
         sqrt_ip.read(0x10)

         Done!! total: 15.033666849136353

Out[43]: 1
```

Fig 4: Implementation of Overlay

Similarly, four different overlays were created in this work. The above-mentioned overlay is just one of those overlays. The details about the rest of the overlays and the results obtained with them are discussed more in detail in Section 5 below.

5 Experiments

Altogether four different versions of overlays were written in this project. Every overlay had slight modifications. Thus, the experiments were conducted with four different methods. For the reference, a partition program was created in native python to compare with the obtained result. The followings are the details about methods used in the project:

1. Method 1: In this method, an overlay was created with inputs 'n' and 'array []'. The S_AXILITE Bus was used instead of the High-Performance AXI bus. A loop was used to assign every array element to every memory location which made this overlay significantly slower. This method performed well with upto 'n=20' but with 'n= 25' the execution time took so long that the execution had to be stopped forcefully.

2. Method 2: In this method, the overlay was created with 'n = 25' defined (hardcoded) inside the overlay. Thus, the only input was 'array []'. Instead of assigning every array element one by one into the memory addresses in the FPGA, it uses the AXI Burst method (High performance AXI Bus) which improves the execution time drastically. As this method explicitly uses 'n=25' inside the overlay, this overlay performs efficiently only for 'n = 25'. The numbers in array can be changed. This method produces result efficiently for instances of the problem of this specific size, but the user does not want a software implementation with this restriction.

3. Method 3: in this method, the overlay was created with inputs 'n' and 'array []'. This overlay is similar to the one created in method 1 but this time the bus used is High Performance AXI bus instead of S_AXILITE bus. Also, this overlay uses AXI burst method to transfer array numbers into the memory addresses instead of using a loop and transferring data one by one. This method is also significantly faster than method 1 but not as much as method 2 for 'n=25'. The good thing with this method is, it provides the user flexibility to change the value of 'n' unlike method 2 which only works efficiently with 'n=25'.

4. Method 4: In this method, the overlay uses 'array []' as the only input. The very first element of the array denotes the value of 'n' in this method. Then rest of the elements denotes the numbers that needs to be partitioned. If the first element in the array is 25, which means 'n=25' and there are 25 numbers after the first element in the array. This method uses High Performance Bus for data transfer and uses AXI Burst method instead of transferring one data at a time. This method produced the same results as method 3.

Table 2 below summarizes the result obtained with the methods discussed above. Similarly, table 3 illustrates the execution time achieved using various methods for various values of 'n'.

Table 2: Results obtained for n=25

Methods	Data Transfer method	Inputs	Execution Time
Method 1	Loop	n, Array ()/	-
Method 2	AXI Burst	Array () with n=25 fixed	8.05 sec
Method 3	AXI Burst	n, Array ()	15.03 sec
Method 4	AXI Burst	Array () with first element as 'n'	15.03 sec

Table 3: Execution time for various values of n (seconds)

	Python	Method 2	Method 1	Method 3	Method 4
n= 25	1002.59	8.05	-	15.03	15.03
15	0.62	0.019	9.58	0.022	0.02
20	25.83	0.25	389.2192	0.39	0.38
10	0.01	0.0038	0.218	0.0059	0.0047

Table 4: Execution time speed factor vs. the pure python code

	N=10	15	20	25
Method 1	x 0.045	x 0.06	x 0.066	-
Method 2	x 2.63	x 32.63	x 103.32	x 124.47
Method 3	X 1.69	x 28.18	x 66.23	x 66.66
Method 4	x 2.12	x 31	x 67.97	x 66.66

It can be deduced from table 4 that as the value of 'n' increases the speed factor increases. Hence, computing this problem in FPGA is much more efficient if the instance of the partition problem is larger. If the size of the instance problem is smaller, then it might not be significantly faster. From the table 4, method 2 is 124 times faster than pure python code when n= 25.

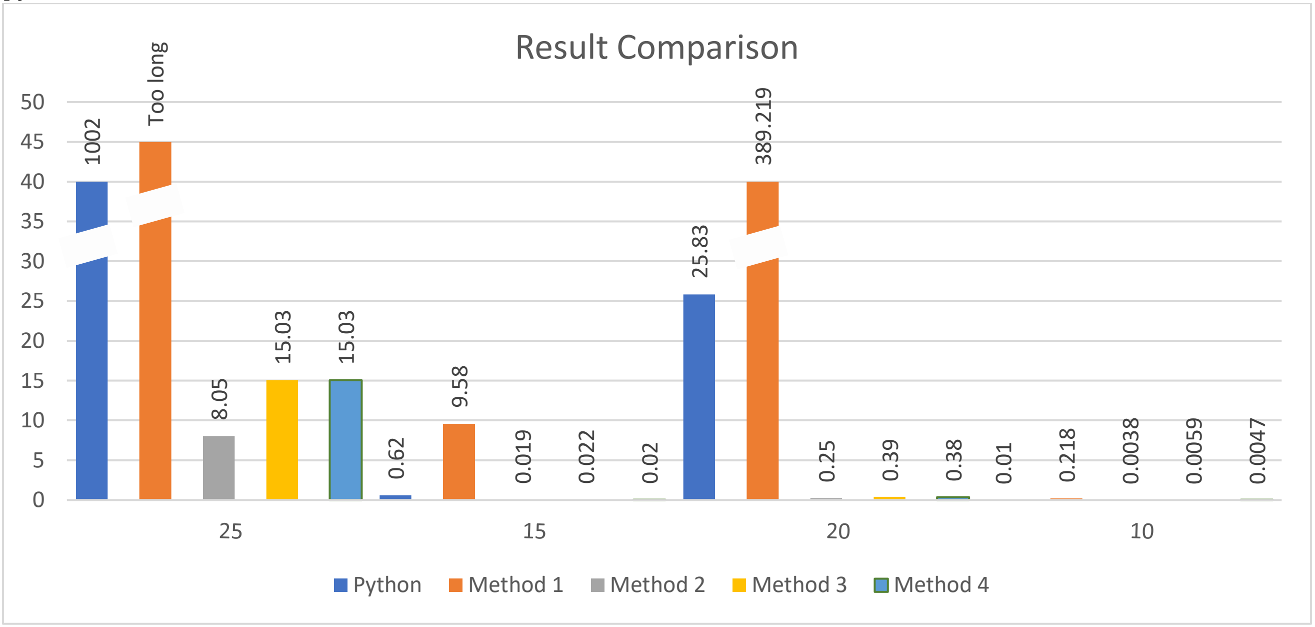


Fig 5: Graphical illustration of table 2.

Figure 5 above depicts the graphical representation of the results achieved using various methods. From the above result, method 1 is the slowest among all the methods as it uses loop technique to transfer the array values into the memory address into the overlay. In method 1, the execution for 'n=25' took too long so eventually the process had to be stopped. Thus, there is no data for that particular size. Also, it can be concluded from the graph and table above that the methods 2,3 and 4, which use HLS as well as AXI Burst technique for data transfer, are faster in execution than compared to pure python code without HLS.

6 Conclusions

We have described how creating an overlay for a PYNQ board allows the acceleration of the execution of a python program that finds exact solutions to small instances of the partition problem.

References

[1] M. B. Gokhale and P. S. Graham. *Reconfigurable computing: Accelerating computation with field programmable gate arrays*. Springer Science & Business Media, 2006.

[2] C. Trefftz, J. Scripps, and Z. Kurmas. An introduction to elements of parallel programming with java streams and/or thrust in a data structures and algorithms course. *Journal of Computing Sciences in Colleges*, 33(1):11–23, 2017.

[3] G. J. Woeginger. Exact algorithms for NP-hard problems: A survey. In *Combinatorial Optimization Eureka, You Shrink!*, pages 185–207. Springer, 2003.

[4] Pynq.readthedocs.io. 2020. *Introduction To Overlays — Python Productivity For Zynq (Pynq) V1.0*. [online] Available at: <https://pynq.readthedocs.io/en/v1.4/6_overlays.html> [Accessed 24 November 2020].