EPiC Series in Computing

Volume 97, 2024, Pages 62–71

Proceedings of 36th International Conference on Computer Applications in Industry and Engineering

Accelerating the execution of the Partition Problem on PYNQ FPGA platform

Pratik Shrestha, Chirag Parikh and Christian Trefftz

Grand Valley State University, Grand Rapids, Michigan, USA.
shrestpr@mail.gvsu.edu, parikhc@gvsu.edu and trefftzc@gvsu.edu

Abstract

Exponential-time algorithms for solving intractable problems are inefficient compared to polynomial-time algorithms for solving tractable problems as execution time for former grows rapidly as problem size increases. A problem is NP-complete when a problem is non-deterministic polynomial (NP) and all other NP-problems are polynomial-time reducible to it. The partition problem is one of the simplest NP-complete problems. 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. Solving the partition problem is a time consuming endeavor. 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. The execution time required to find a solution to instances of the partition problem is greatly reduced using a Field Programmable Gate Array (FPGA). In this paper, we talk about the use of the PYNQ board in conjunction with an overlay to accelerate the execution of a function that evaluates if a partition is a solution to an instance of the partition problem. In order to assist with the evaluation, four different overlays are created and performance comparison among them using native python is then presented in the paper.

1 Introduction

Parallel processing is a topic of growing importance in the computing world. The exponential growth of processing and network speeds means that parallel architecture is not just a good idea but now a necessity. Many problems require enormous amount of time to be solved. For e.g., exponential-time algorithms take longer for solving intractable problems in comparison to their polynomial-time algorithm counterpart for large problem sizes. In addition, parallel systems have proven to be the only

alternative to obtain solutions in a reasonable amount of time. Hence, there is lot of recommendations for curricula of computer science undergraduate degrees to emphasize on topic of parallel processing.

Introductory courses in parallel processing include surveys of different computer architectures: Shared memory machines with microprocessors comprising of several cores, Graphics Processing Units (GPUs) and clusters of computers, among others. Field Programmable Gate Arrays (FPGAs) on the other hand have proven to be efficient accelerators for the execution of many different applications [1]. Hence, it is of benefit to have the topic of FPGAs be included in a course in parallel processing. The challenge faced by an instructor who wants to cover FPGAs in a parallel processing course is that programming FPGAs requires a very strong background and skills in hardware design that most computer science students lack. To assist with this, Xilinx has created a board called PYNQ [2] for pedagogical purposes that can be easily programmed using Python without the need of being proficient in hardware design. Figure 1 shows the PYNQ board.

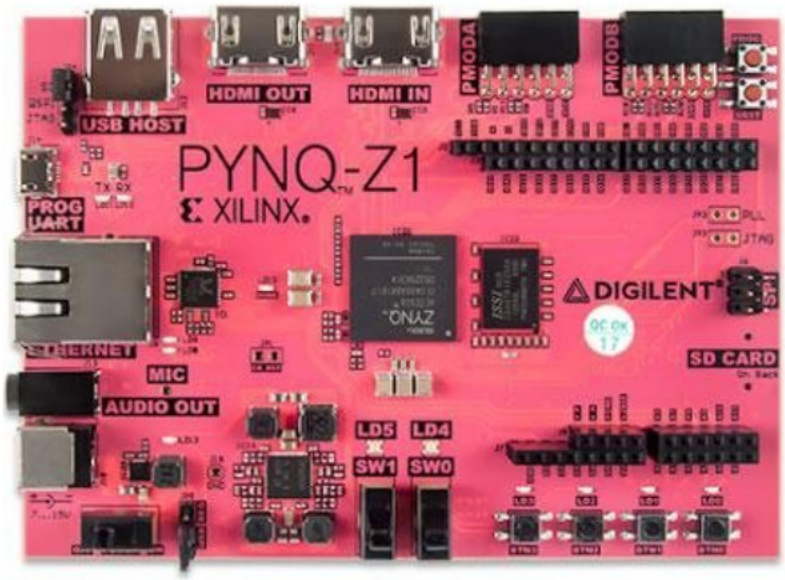

Figure 1: PYNQ Development kit from Xilinx

PYNQ board contains an FPGA device with a built-in Arm microprocessor that has two cores and a programmable fabric. PYNQ board runs a custom version of Linux and are therefore considered as a stand-alone computer. A PYNQ board can connect to a traditional computer through an Ethernet cable and an USB cable. Xilinx has chosen Jupyter notebooks to provide a very convenient way of interacting with a PYNQ board,. The PYNQ board can run a web server that interacts with a python interpreter. The user can start a browser on his/her computer and access web pages on the server running on the PYNQ board. Those web pages may contain python code that will execute on the PYNQ board. The Python interpreter on the PYNQ board can interact with *overlays,* which are configurations of the programmable fabric of the FPGA that can execute specific functions.

A problem is considered NP-complete when a problem is non-deterministic polynomial (NP) and all other NP-problems are polynomial-time reducible to it. 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. 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 followed by a "brute force" approach to solve the Partition problem outlined in section 3. The process of creating the overlay is described in section 4 followed by experimental results and conclusions are in sections 5 and 6 respectively.

2 The Partition Problem

In the world of computer science, partition problem or sometimes called as number partitioning [3] is the task of deciding when given a multi-set of positive integers S, can it be partitioned into two sub multi-sets $S1$ and $S2$ such that the sum of the elements is $S1$ is equal to the sum of the elements in $S2$?

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 $S1 : \{4,5\}$ and $S2 : \{9\}$.

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. The next section talks about the exact algorithm to solve the partition problem.

3 An Exact algorithm to solve Partition problem

Woeginger [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. For each of those possible subsets, one uses a function that evaluates if that subset is a solution to the problem of interest. One then proceeds to choose, among the subsets that are possible solutions, the one that works best. Other NP-hard algorithms that can be solved using the same brute-force approach include the maximum-clique problem, the maximum independent set problem and the minimum dominating set problem.

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 $2n - 2$ 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 Figure 2. As can be observed, the complexity of the algorithm is $O(2^n)$, exponential.

```
Main Algorithm: Algorithm to solve instances of the Partition problem

input : n size of the problem array: values in the multiset
output: true or false

indexOfPossiblePartition = 1
while indexOfPossiblePartition < 2^(n-1) - 1 do
    if evaluatePossiblePartition (indexOfPossiblePartition, n, array)
      then
        return true
    end
    else
        indexOfPossiblePartition++
    end
end
return false
```

Figure 2: Main algorithm to solve instances of Partition problem

The outline of the function *evaluatePossiblePartition()* is presented as Algorithm 2 in Figure 3. The complexity of this method is $O(n)$. Recall that n is the cardinality (the number of elements) in the multiset.

Algorithm 2: Algorithm to evaluate if partition is the solution.

Input: n, array that contains the values, indexOfPossiblePartition
Output: true or false

```
sumOfValuesInPartition0 = 0;
sumOfValuesInPartition1 = 0;
index = 0

while index<n     do
    if bitindex    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;
```

Figure 3: Outline of the function *evaluatePossiblePartition()*

This algorithm can be easily parallelized using environments like OpenMP, for shared memory machines, Thrust, for GPUs, or MPI for clusters or computers. 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 [5].

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 next section, we use the programmable fabric of the FPGA to accelerate the execution of that function.

4 Implementation on an FPGA using 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 [6]. 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 have to worry about is the top-level program.

Creating an Intellectual Property (IP) core using High Level Synthesis (HLS) is the very first step required to create a custom overlay. For the HLS portion of this design, Xilinx's Vivado HLS was used. Different pragmas were inserted in a C program to boost the efficiency. After the successful creation of the IP core, the IP component is imported into the Vivado Suite. In the block diagram shown in Figure 4, the Zynq processor is connected to the custom IP. 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.

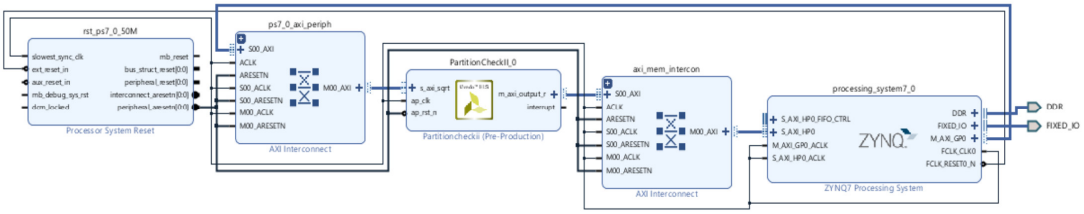

Figure 4: Block design of the overlay

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 6 below using the Python code. 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
```

Figure 5: Import Overlay

This overlay can be thought as a block, as shown in Figure 6, 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.

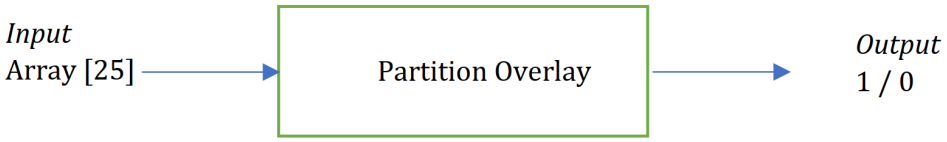

Figure 6: Overlay block

As can be seen clearly from the overlay block in Figure 6, there are two ports in total. Each of them has their own physical memory address used as Memory Mapped Input Output (MMIO) for I/O operation. In the code snippet shown in Figure 7 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.

```
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
```

Figure 7: Implementation of the overlay

Execution time is another important aspect of this 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 7 above, the array "*numbers*" include 26 elements where the first element, 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.

Along with the above-mentioned overlay, three additional overlays were created for this work bringing the total to four overlays. The next section discusses the remaining overlays in brief along with their implementation results.

5 Experiments and Results

Altogether four different versions of overlays were created for this work, every overlay had slight modifications. Thus, the experiments were conducted with four different methods. For reference purposes, 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 up to '*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 that 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 results obtained with the methods discussed above. Similarly, Table 3 illustrates the execution time achieved using various methods for different values of '*n*'.

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 2. Results obtained for n=25

	Python	Method 1	Method 2	Method 3	Method 4
n = 10	0.01	0.218	0.038	0.0059	0.0047
n = 15	0.62	9.58	0.019	0.022	0.02
n = 20	25.83	389.21	0.25	0.39	0.38
n = 25	1002.59	n.a.	8.05	15.03	15.03

Table 3. Execution time for various values of n in seconds

It can be deduced from Table 4 shown below 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, once can see that method 2 is 124 times faster than pure python code when n= 25.

	n = 10	n = 15	n = 20	n = 25
Method 1	x 0.045	x 0.06	x 0.066	Too long
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

Table 4. Execution time speed factor versus the pure python code

Figure 8 below shows 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 Table 4 and Figure 8 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.

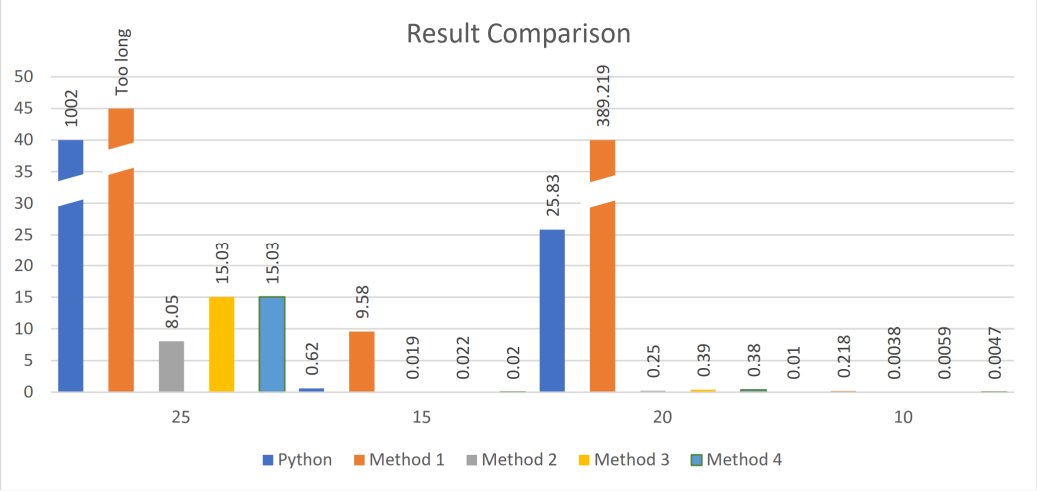

Figure 8: Graphical illustration of Table 3

6 Conclusion

Exact algorithms can find solutions for small instances of NP-hard problems in a reasonable amount of time. However, for large instances of NP-hard problems, the execution time required to find a solution to instances of the partition problem is greatly reduced using a Field Programmable Gate Array. This paper illustrates the ability of FPGAs to accelerate the execution of code. In this paper, we have described how creating an overlay for a PYNQ FPGA board allows the acceleration of the execution of a python program that finds exact solutions to small instances of the partition problem.

The work proposed in the paper can also be used in a course in two different ways:

- Students with little background on FPGAs and hardware design, can observe the speedups that can be obtained by using the fabric of an FPGA to accelerate the execution of code. Jupyter notebooks were created as part of this work that make it simple to observe the speedups.
- Students with experience using FPGAs can use this work as the basis to accelerate other algorithms

A github repository has been setup to assist interested audience with all the resources necessary to use the software: *https://github.com/pratikstha/PartitionProblemUsingFPGA*

References

[1] M. Gokhale and P. Graham, "Reconfigurable computing: Accelerating computation with field programmable gate arrays." 2006. [Online]. Available: Springer Science and Business Media.

[2] "XUP PYNQ," [Online]. Available: https://www.xilinx.com/support/university/boards-portfolio/xupboards/XUPPYNQ.html.

[3] "Partition problem - Wikipedia" [Online]. Available: https://en.wikipedia.org/wiki/Parition_problem

[4] G. Woeginger, "Exact algorithms for np-hard problems: A survey," in *Combinatorial optimization-eureka, you shrink!,* Springer, 2003, pp. 185-207.

[5] 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 algoriths course," Journal of Computing Sciences in Colleges, 2017, 33(1):11-23.

[6] "Introduction to Overlays - Pyhton Productivity For Zynq (Pynq) V1.0," Pynq.readthedocs.io 2020 [Online]. Available: https://pynq.readthedocs.io/en/v1.4/6_overlays.html