The execution of the Partition Problem: A Comparative Study of Various Techniques for Efficient Computation

Pratik Shrestha *
Grand Valley State University, Grand Rapids, Michigan, USA.
Chirag Parikh †
Grand Valley State University,Grand Rapids, Michigan, USA.
Christian Trefftz ‡
Grand Valley State University, Grand Rapids, Michigan, USA.

June 20, 2024

Abstract

Exponential-time algorithms for solving intractable problems are considered inefficient when compared to polynomial-time algorithms for solving tractable problems. The reason being that the execution time for former grows rapidly as problem size increases. 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. 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 by using parallel processing counterparts such as Graphics Processing Unit (GPU) and Field Programmable Gate Array (FPGA). In this paper, we talk about the use of the NVIDIA T4 GPU and PYNQ FPGA 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. To assist with the evaluation, four different overlays are created

for FPGA and performance comparison among them with GPU using python and the numba/cuda library 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 an 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 to be proficient in hardware design. Figure 1 shows the PYNQ board

*Reasearch Assistant Email: shrestpr@mail.gvsu.edu.
†Professor and Chair of Computer Engineering program at Grand Valley State University. Email: parikhc@gvsu.edu.
‡Professor at the College of Computing at Grand Valley State University . Email: trefftzc@gvsu.edu.

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 a 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 be executed 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. On the other hand, a GPU is a specialized electronic circuit designed to accelerate graphics rendering, primarily used in rendering images and videos for computer displays [7]. Over time, GPUs have evolved beyond their original graphics-centric purpose, finding significant applications in parallel processing tasks. Due to their parallel

architecture and capability of high- performance computing, GPUs excel in handling large amounts of data simultaneously, making them highly efficient for parallelizable tasks such as scientific simulations and artificial intelligence (AI) computations. The environment used to implement the version of the program that uses a GPU was Google COLAB. COLAB executes on a dedicated machine with a Xeon microprocessor and a NVIDIA T4 GPU. [8]. A T4 GPU has 2560 cores. It has 16 gigabytes of main memory, and it is connected to the host computer using a PCI Express bus (version 3.0 x16). In this work, the GPU is utilized to solve the NP-Complete problem through parallelization. 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 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. process of creating the overlay is described in section 5 followed by experimental results and conclusions in sections 6 and 7 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, if it can 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. NP-complete problems are yes/no questions. The letters NP stand for Non-deterministically Polynomial. These problems have the characteristic that it is possible to write an algorithm with Polynomial execution time that will be able to determine if a candidate solution is indeed a solution to the problem or not. The challenge for this family of problems

Implementation of the partition problem using NVIDIA T4 GPU is described in section 4. The

is to generate the appropriate candidate solution. To this day, the exact time complexity of algorithms that solve NP- complete problems is not known. So far, the only algorithms that generate the proper candidates for a problem have exponential time complexity. The consensus among most practitioners is that it

is not feasible to find algorithms with better time complexity to generate the proper candidates. NP-complete problems have another interesting property: Algorithms to transform the input of one NP-complete problem to other NP-complete problems exist. Those algorithms are called “reductions” and they have polynomial complexity. When a researcher comes across a new problem and wants to show that this is an NP-complete problem, the proof is the description of a “reduction” to an existing NP-complete problem. Thus, if anybody were to write an exact algorithm with polynomial time complexity to solve an NP-complete problem, all the problems in the class could be solved in polynomial time as well, thanks to the existing reduction algorithms. It is important for software developers to be aware of the existence of NP-complete problems. Some problems from real life are in this category. It is important to be able to tell the users that only small instances of problems in this class can be solved exactly in reasonable amounts of time. Large instances of these problems are intractable, and it becomes necessary to use other alternatives, to use approximation algorithms or to use procedures that may not produce optimal solutions. 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 2n-1 - 1 encode the possible subsets of interest. Notice that the other values between 2n-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(23 - 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 2n-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(2n), exponential

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 NVIDIA T4 GPU to speed-up the execution of the function.

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

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 bit $index$ in the binary representation of $indexOfPossiblePartition$ is 0 **then**
    $sumOfValuesInPartition0 += array[index]$
  else
    $sumOfValuesInPartition1 += array[index]$
  end if
  $index++$
end while
if $sumOfValuesInPartition0 = sumOfValuesInPartition1$ **then**
  return true
else
  return false
end if

4 Implementation using NVIDIA T4 GPU

The NVIDIA T4 Tensor Core GPU is a powerful accelerator designed for various cloud workloads, including deep learning, machine learning, data analytics, and video transcoding. Even though the NVIDIA T4 Tensor Core GPU is not specifically designed to solve NP-Complete problems directly, however, it can significantly accelerate certain aspects of solving such problems due to its parallel processing capabilities and high throughput. In terms of performance, the T4 delivers up to 40 times higher performance than CPUs [9]. This implementation was performed in Google Colab using Nvidia T4 GPU as shown in Figure 4 below.

Python lacks native support for GPU programming, but developers can leverage the NUMBA/CUDA library to write code tailored for NVIDIA GPUs. This library harnesses the

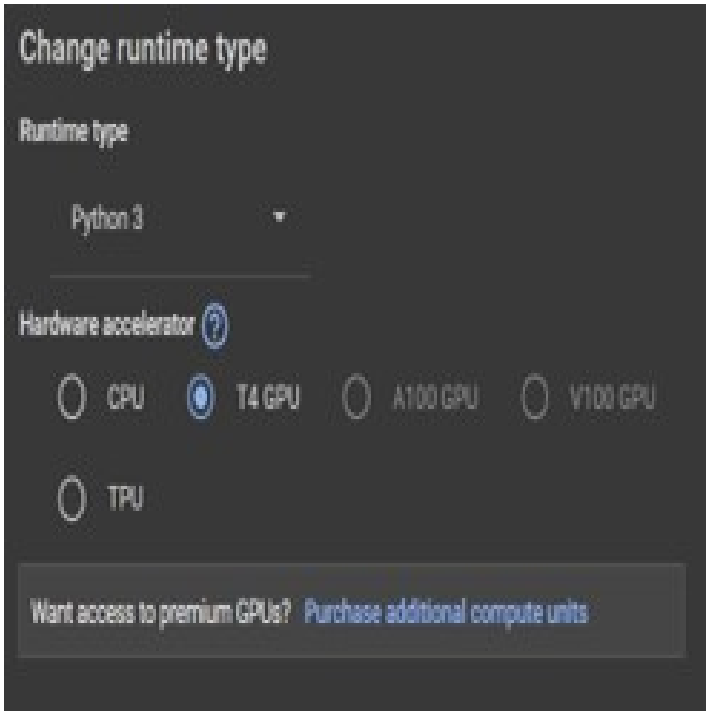

Figure 1: Selection of Runtime type

LLVM compiler infrastructure to generate binary executable code. Through the NUMBA library, specific functions within a Python program can be compiled into native binaries. Decorators are applied to signal which functions should undergo compilation, guiding the library towards optimization. Upon successful compilation, function calls execute as binary code, resulting in accelerated performance compared to the interpretation process of standard Python code. The NUMBA/CUDA library, a subset of NUMBA, specializes in producing GPU- executable code. Decorators are utilized just before functions intended for GPU execution. The evaluate partition has been decorated as shown below:

Figure 2: Decorator used in the implementation

In addition to utilizing decorators, developers must employ additional functions when working with GPUs. GPUs function as distinct computing units with their own dedicated memory. Hence, it is essential to transfer variables, typically arrays, from the host memory to the GPU's memory. Once the relevant variables have been transferred to the GPU's

memory, the code designated for GPU execution is invoked. This requires calling a function that has been compiled with the necessary decorators. Developers must specify the size of the arrays that the functions will operate on. As previously mentioned, the code executing on the GPU corresponds to

a function marked with the apropriate decorator. Upon completion of the GPU code, the program retrieves the desired results and transfers them back to the host's memory. figure below highlights the most important actions in the code to interact with the GPU:

- Moving arrays to the GPU memory
- Executing the code on the GPU
- Finally copying the results from the GPU memory to the host memory

```
# Copy variables to the GPU memory
arrayGPU = cuda.to_device(array)
resultGPU = cuda.to_device(result)
# Execute the function in the GPU
evaluatePartition.forall(nPartitions)( arrayGPU,resultGPU, n)
# Copy the result array back to the CPU
resultGPU.copy_to_host(result)
```

Figure 3: Important steps while interacting with GPU

As discussed in the previous section, the essence of the algorithm is to execute a function that evaluates if a particular integer value is an encoding of a partition that solves the instance of the problem. All the integers that encode subsets (elements) from the power set can be evaluated in parallel. If the number of elements to be evaluated is larger than the number of cores available in the GPU, the GPU operating system takes care of executing the code the necessary number of times so that the function is executed on all the elements. On the COLAB notebook mentioned before, an instance of size 25 took 1.024 seconds to execute. In the next section, we use the programmable fabric of the FPGA to accelerate the execution of that function

5 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.

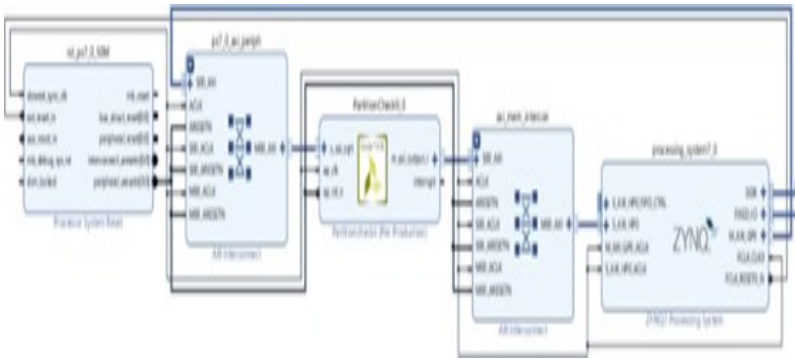

Figure 4: Block design of the overlay

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 8 below using the Python code. Here, the overlay "PartitionCheckII" has been imported. Then the next line indicates that the overlay consists of an IP PartitionCheckII0 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.

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

Figure 6: Overlay block

Output (MMIO) for I/O operation. In the code snippet shown in Figure 10 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 10 above, the array "numbers" include 26 elements where the first element, 25, denotes that there are a 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.

6 Experiments, Results and Analysis

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'.

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 and NVIDIA GPU is much more efficient if the instance of the partition problem is larger. Particularly for n=25, NVIDIA T4 GPU turned out to be the fasted among all, which is 979 times faster than pure python executed in CPU. If

Methods	Data Transfer method	Inputs	Execution Time
Method 1	Loop	n, Array () /	-
Method 2	AXI Burst	Array () n 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 (CPU)	Method 1 (FPGA)	Method 2 (FPGA)	Method 3 (FPGA)	Method 4 (FPGA)	NVIDIA T4 (GPU)
n = 10	0.01	0.218	0.038	0.0059	0.0047	0.338
n = 15	0.62	9.58	0.019	0.022	0.02	0.355
n = 20	25.83	389.21	0.25	0.39	0.38	0.39
n = 25	1002.59	n.a	8.05	15.03	15.03	1.024

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

the size of the instance problem is smaller, then it might not be significantly faster. From Table 4, one can see that method 2 is 124 times faster than pure python code when n= 25. Table 4. Execution time speed factor versus the pure python code

	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
NVIDIA T4	x 0.029	x 1.74	x 66.23	x 979.09

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. Additionally, particularly for n=25, NVIDIA T4 GPU is the fastest among which has the execution time of 1.024 sec.

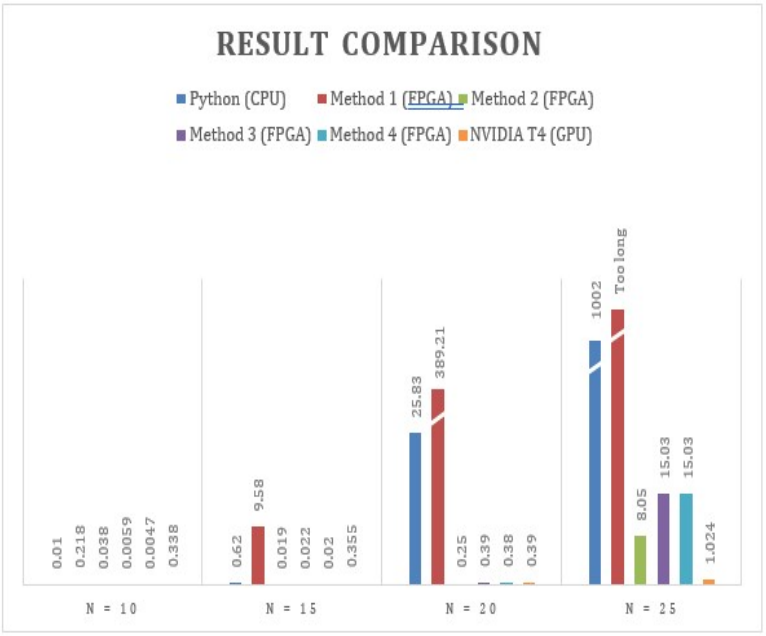

Figure 8: Graphical illustration of Table 3

From Figure 8, it can be seen that FPGA outperforms GPU for all cases except n=25. We think this behavior was observed due to the following factors:

• Architecture and Design o FPGAs and GPUs have fundamentally different architectures. FPGAs are highly customizable hardware that can be tailored to specific tasks, but their design and optimization can be complex. GPUs, on the other hand, are designed for parallel processing and may have better-suited architectures for certain types of algorithms. It is possible that the algorithm we used for this NP- hard problem is better suitable for GPU than FPGA, specifically the FPGA we used. • Optimization and Compiler Efficiency o GPUs have more mature and well-optimized compilers, and their ecosystems, such as CUDA, have extensive community support. This contributes to better compiler optimization and overall efficiency, potentially resulting in improved performance. Whereas in FPGA, we used pragmas manually for optimization. Thus, it is possible that the built-in optimizer of T4 GPU compiler provided by Google outperformed our manual optimization.

7 Conclusion

In this paper, a comprehensive comparative analysis was conducted on various techniques aimed at accelerating the execution of the Partition Problem, a well- known NP-complete problem. The potential of parallel processing architectures, specifically GPUs and FPGAs, was explored to enhance computational efficiency. The effectiveness of GPU and FPGA implementations in accelerating the execution of the Partition Problem was demonstrated through experimental evaluation. Significant reductions in execution time were observed, particularly for larger problem instances, such as n=25, where the GPU implementation on the NVIDIA T4 GPU outperformed FPGA implementations and traditional CPU-based approaches. Insights into design considerations and optimization strategies pertinent to both GPU and FPGA implementations were provided. While GPUs offer mature compilers and extensive community support, FPGAs boast highly customizable hardware tailored to specific tasks. Thus, the choice of which technology to use depends on various aspects of the problem. Overall, this study contributes to advancing the understanding of efficient computation techniques for NP-hard problems. It serves as a valuable resource for researchers and practitioners interested in leveraging parallel processing architectures for computational acceleration. A github repository has been setup to assist interested audience with all the resources necessary to use the software: https://github.com/pratikstha/PartitionProblemUsingFPGA article

graphicx verbatim

Your Title Your Name June 20, 2024

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`.
   (a) "Partition problem" Wikipedia [Online]. Available: `https://en.wikipedia.org/wiki/Partition_problem`.
3. G. Woeginger, "Exact algorithms for np-hard problems: A survey," in *Combinatorial optimization-eureka, you shrink!*, Springer, 2003, pp. 185-207.
4. 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.
5. "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`.
6. "What is a Graphics Processing Unit (GPU)? Definition and Examples," [Online]. Available: `https://www.investopedia.com/terms/g/graphics-processing-unit-gpu.asp`.
7. "NVIDIA T4," [Online]. Available: `https://www.nvidia.com/en-us/data-center/tesla-t4`.
8. "NVIDIA Turing GPU Architecture," [Online]. Available: `https://images.nvidia.com/aem-dam/en-zz/Solutions/design-visualization/technologies/turing- architecture/NVIDIA-Turing-Architecture-Whitepaper.pdf`.

BIOGRAPHY / BIOGRAPHIES

Dr. Chirag Parikh earned his Master's and Doctoral degrees from University of Texas at San Antonio in 2003 and 2007 respectively. Currently, he is a Professor and Chair of Computer Engineering program at Grand Valley State University. His research interests are embedded system design, cryptography and FPGA-based system design.

Dr. Christian Trefftz earned a master's degree in Computer Science from Western Michigan University in 1989 and Ph.D. Degree in Computer Science from Michigan State University in 1994. He is a Professor at the College of Computing at Grand Valley State University. His research interest is in the area of parallel processing.

Pratik Shrestha holds a Master's degree in Computer Science (2020) and a Master's degree in Electrical and Computer Engineering (2019) from Grand Valley State University. Currently, he is a Software Engineer at CTDI. His areas of expertise and interest include machine learning, computer vision, embedded systems, and software development. Pratik has developed a range of software applications and machine learning models, and he is passionate about using his technical skills to drive innovation and contribute to advanced technological solutions.