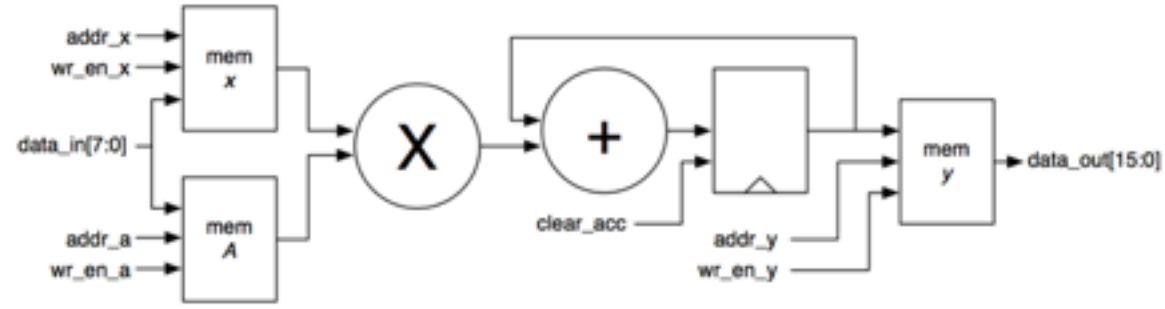
Final Project - Report

Hardware Generator Tool
For
Matrix Vector Multiplier



$$\begin{bmatrix} y_0 \\ y_1 \\ y_2 \end{bmatrix} = \begin{bmatrix} a_0 & a_1 & a_2 \\ a_3 & a_4 & a_5 \\ a_6 & a_7 & a_8 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} a_0x_0 + a_1x_1 + a_2x_2 \\ a_3x_0 + a_4x_1 + a_5x_2 \\ a_6x_0 + a_7x_1 + a_8x_2 \end{bmatrix}$$

Project by
Rishikanth Chandrasekaran & Gunjan Shrivastava
110284413 110615473

Part 1

1. The hardware generator was built by giving great importance to scalability and flexibility. In our implementation the generic aspect or the aspect of making the design scalable and flexible was incorporated into the System Verilog code. The shell script has 4 different Verilog design codes embedded into it to cater to the cases of a. No pipeline or parallelism b. Pipelining c. Parallelism d. Parallelism and Pipelining. Thus the script chooses between designs depending on values of p and g entered. It also computes and appends relevant data constants and masks for the correct functioning and scaling of the test bench and System Verilog design. The scaling parameters k and b are handled directly by the Verilog design. The system Verilog code was architected and crafted using parameterized modules so that, by simply changing the parameters of K, log K and B specified on top of the Verilog code, the design scales for the required specification of k and b. Thus all the script needs to do is to change the values for these parameters and the appropriate design is generated.
   Thus the scaling and flexibility for *k,p,b,g* are handled very efficiently by the sharing the complexity between Verilog and the Script. A script based implementation also allows easy editing of design without any syntactic hassles making it very flexible.

   Limits of *k* and *b*
   The fact that there are no maximum defined values meant that the design as well as the test bench must be able to support the maximum value that can be allowed within scope or support of the respective languages. This way the variance or diversity of the system specification that user can utilize is maximized. The test bench generator or the program which generates the input data files and expected output for verification is written in C and the hardware design description in System Verilog.
   Since both these languages have their own limitations with regard to maximum variable size, the worst of the two is the limiting factor for the parameters.

For *k*:
Since k is an integer the limits of C should have allowed a maximum of 65534. However, upon testing, we observed that the simulation tool is unable to handle designs exceeding 11000 and threw up an error stating the array of memory has exceeded limits. Thus the practical limit for **k** is **11000.** On testing though the simulation and synthesis are extremely slow due to the large value.

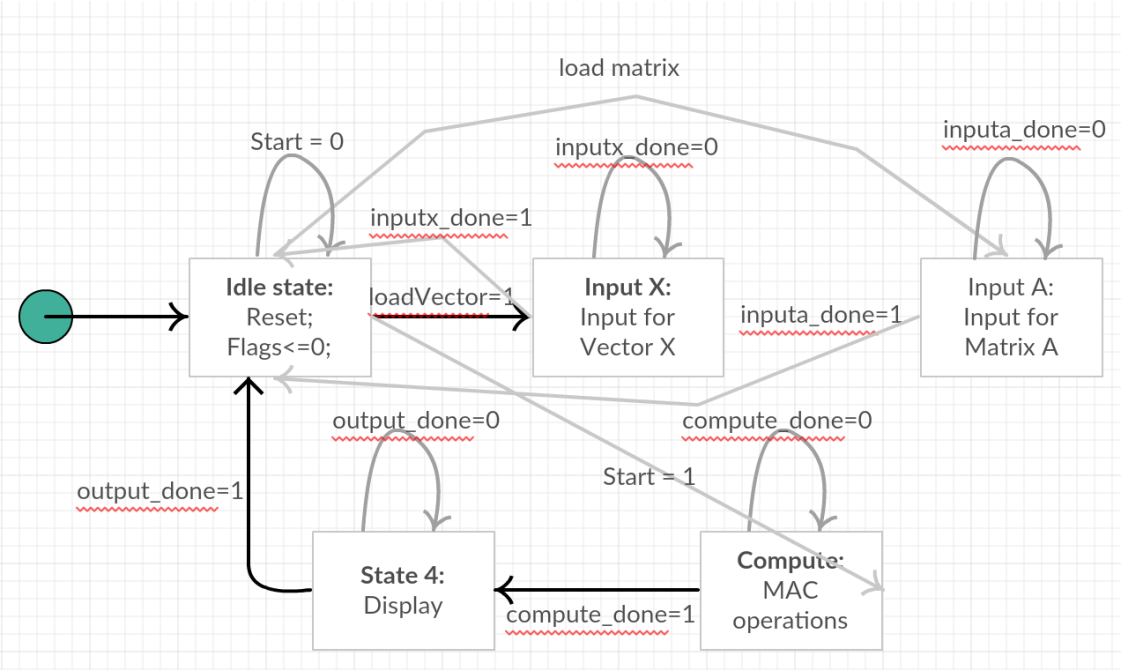
For *b*:
Here the maximum limit is defined by the maximum size variable we can declare in the C code for providing input values and computing expected outputs. Thus this gives a limit of **64 bits**.

2. The control module is implemented using a Finite State Machine (FSM). The FSM consists of 5 discrete states, with each state performing specific functions. The transition from one state to other is triggered by flag signals which indicate the completion of the previous states.
The 5 states are listed down below:

Idle State : all counters and flags are set to default values – waits in this state till start signal – Comes to this state when reset is 1
Input X state : The values for vector X are written into memory
Input A State : Input state (A) – The values for Matrix A are written into memory
Compute State : The matrix and vector are fed into the MAC and the results are written into the output memory Y
Output State : The output values are sent out on data_out from the memory Y. The done signal is also generated before the values are sent.

The state flow diagram of the Finite State Machine is shown by the figure below.

Figure 1: State Transition Diagram of FSM



The control path module makes use of counters and flag signals to generate the addresses for the read and write operations of the memories of A, X and Y. It also synchronizes the write enable signals for the memories. It effectively controls and directs the data path module.

In **Idle State**, it waits for the start / loadVector/ loadMatrix signal.
In **Input X State**, the system transitions from Idle it generates addresses and write enable signals for memory X to store the Vector X in constant sync with the clock edge. Once the input for the vector has been stored, it **raises** the **inputx_done** flag indicating the end of this state, and goes back to **Idle State**.
In **Input A State**, it generates addresses and write enable signals for memory A to store the Matrix A in constant sync with the clock edge. Once the input for the vector has been stored, it **raises** the **inputa_done** flag indicating the end of this state. It then transitions to the **Idle State**.
In **Compute State**, it feeds the row wise the matrix and vector elements into the MAC, and after the end of calculation for one row, the answer is written to memory Y, for which addresses and write enable signals are generated at the correct instant. After all calculations are over, it **raises** the **compute_done** flag which signals the end of this state. After this it transitions to **Output state**
In **Output State**, the **done** signal is **raised** indicating that the results are ready to be displayed. The clock cycle immediately after done is raised the outputs

stored in memory Y are pushed out on the data_out line. After all values are displayed, the **output_done** flag is **raised** which brings the FSM back to **Idle State**.

At any instant of time the control module knows precisely in which state the system is operating in by monitoring the flag signals and the previous states. Since each flag signal uniquely represents the operation of each state, the control module is able to keep track of the states.

The FSM structure doesn't change with k or as the scaling of the system with change in k doesn't depend on the State Machine. The design scalability is handled by the effective architecture or implementation of the design as a whole rather than the FSM.

3. Parallel Design

For parallel design (p=k) the generator implements k macs (multiply accumulate modules) to parallel compute the products. Thus this reduces the number of clock cycles required for computation to k cycles. The structure involves implementing k single memory units for storing the vector values so that they can feed 4 macs simultaneously. The matrix memory is split row wise there by giving k memories of size k. Each row out of the k rows are fed simultaneously into the mac along with each of the vector values. Thus in k cycles the mac units compute the products and produce the k final answers simultaneously. The outputs are temporarily stored in k temp memory units and are fed to the output lines using an indexed memory unit which operates like a mux.
This optimization means, theoretically the same overall number of memories are used as that of a non-parallel implementation but are however split into different sizes to facilitate the operation of the system.

4. Test bench Strategy

The generator implements 3 different test benches to check various cases. These test benches are supported by a C program that generates the required input data and expected output data for verification.
The C program generates random inputs (both positive and negative) of

required bit length and number(k) and dumps them to 2 files aData and xData for matrix and vector respectively. It also computes the product of the matrix and vector taking into account overflow cases similar to that taking place in the Verilog design and dumps the results to a file expectedOutput. The C program has a parameterized value for the number of test cases or sets to generate to test the system. (implemented as a #define)
The 3 testbenches read the files aData and xData for loading the memories with values and the expectedOutput file for verifying the computed values.

Each of the testbench tests the following aspects:

tbench1 : Loads Vector first and then Matrix and gives start signal and verifies outputs

tbench2 : Loads Matrix first and then loads Vector and gives start signal and verifies outputs

tbench3 : This resets the system in the middle of operation to verify reset functionality.

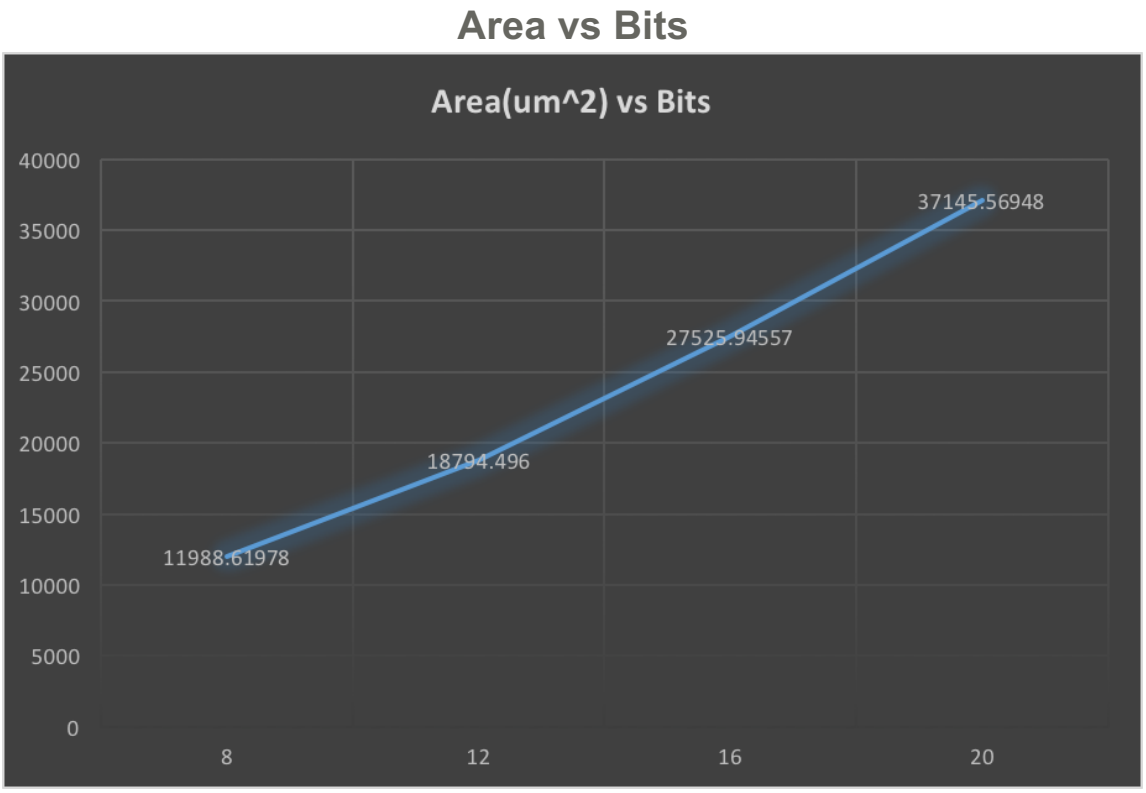
5. These tradeoffs can be explained in terms of Power Performance and Area.

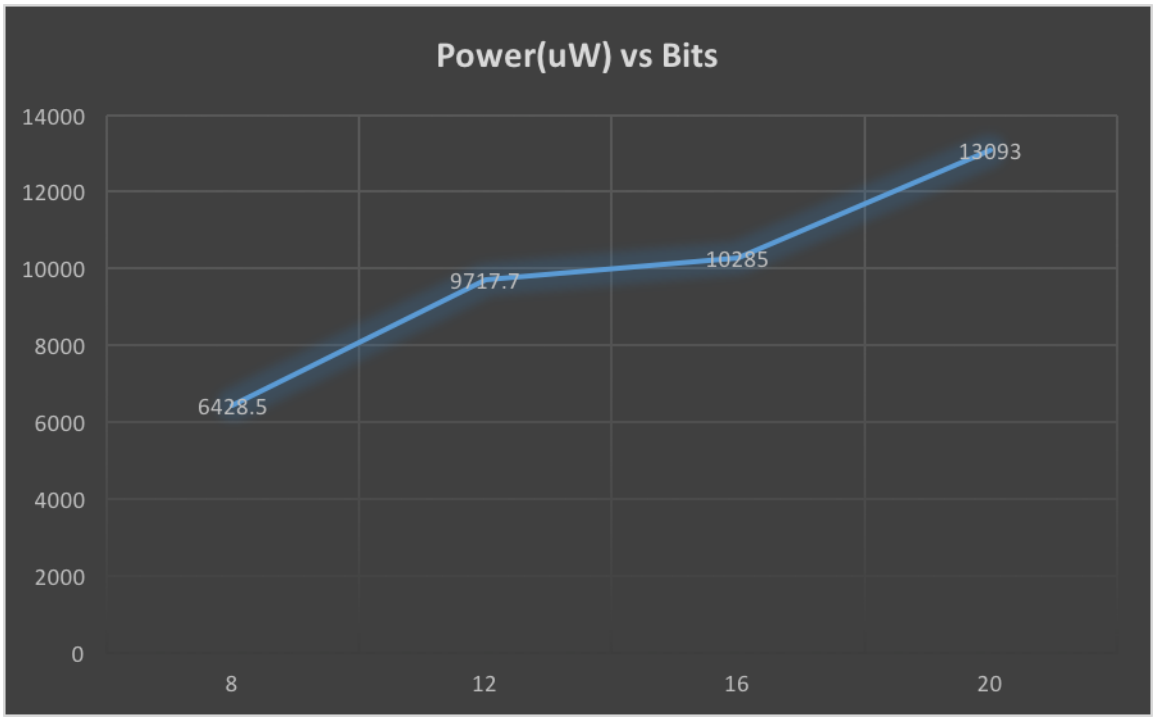
Power: As far as power increases is concerned as K increases, power increases. As K increases more flipflops and registers are used which increases power consumption.
When pipeling is used the power consumption increases slightly due to the one extra flip flop added. In case of parallelism multiple such flipflops are used which can make power increase significantly for large values of K. As b is increased larger memories and flipflops will be required with increased fanouts. This again causes power to increase.

Area : Area also increases with that of K, P, B, and G. This is straightforward. As K and B increase the number and size of components and electronic units increases causing area to increase. In case of pipelining extra flipflops are used which again causes area to increase. When a parallel design is adopted the area increase is significant. This is because a parallel implementation comes with K times the circuitry of that of a non parallel design. Even if total memory doesn't increase there are far too many components which causes a huge increase often 2x in area.

Performance : As K increases performance decreases. This is because more flipflops are used causing the frequency to decrease. Thus performance gradually decreases. When pipelining is used the net throughput increases with a small overall delay allowing the system to be clocked at a faster rate.
Parallelism gives the best performance. When a parallel design is implemented since all computations are done simultaneously the speed is very high.
Increasing bit size will cause the delay to increase thereby reducing performance. This is because fan out increases and the ability of the sources to drive all bits will also be impacted, causing performance to take a slight drip.

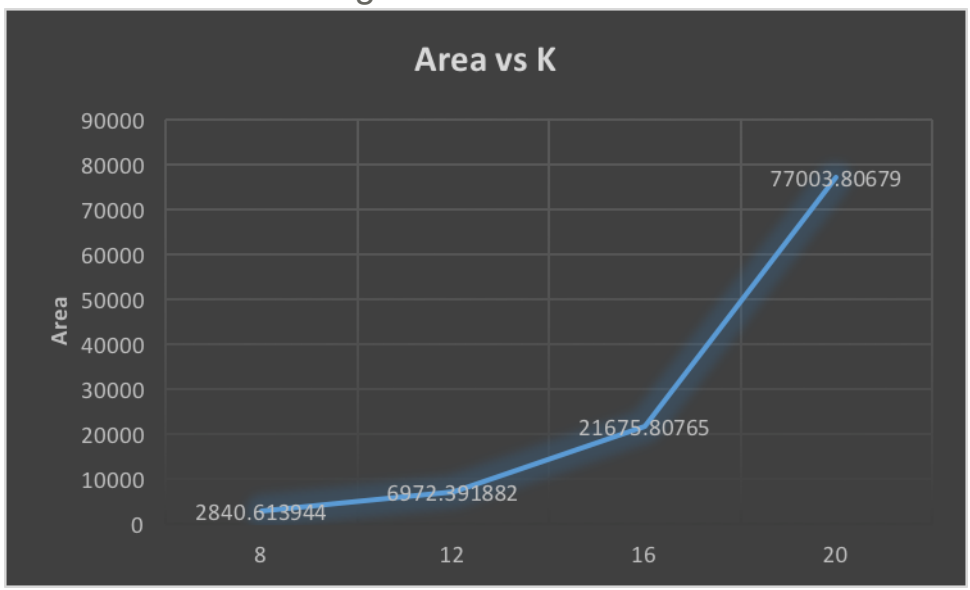
6. Area and Power vs Precision

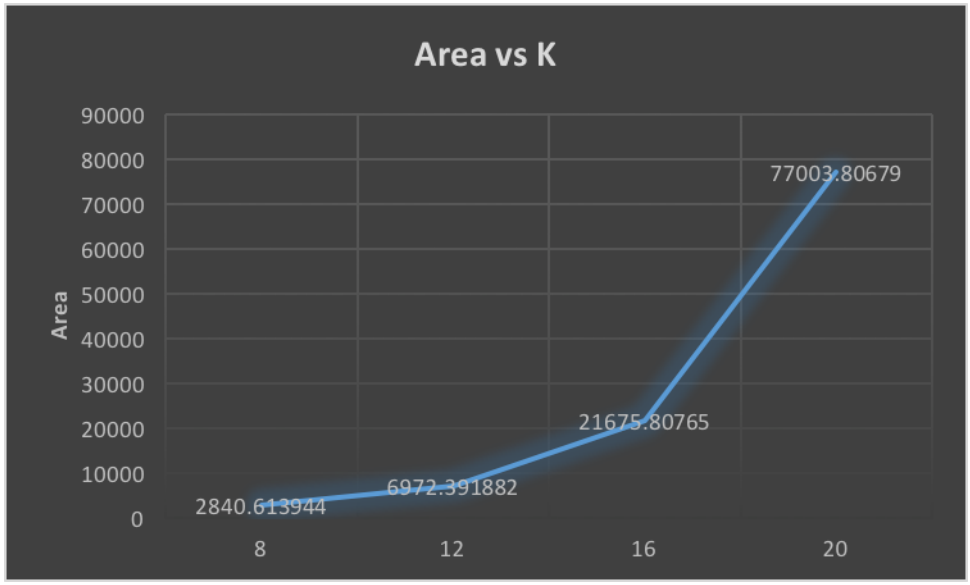




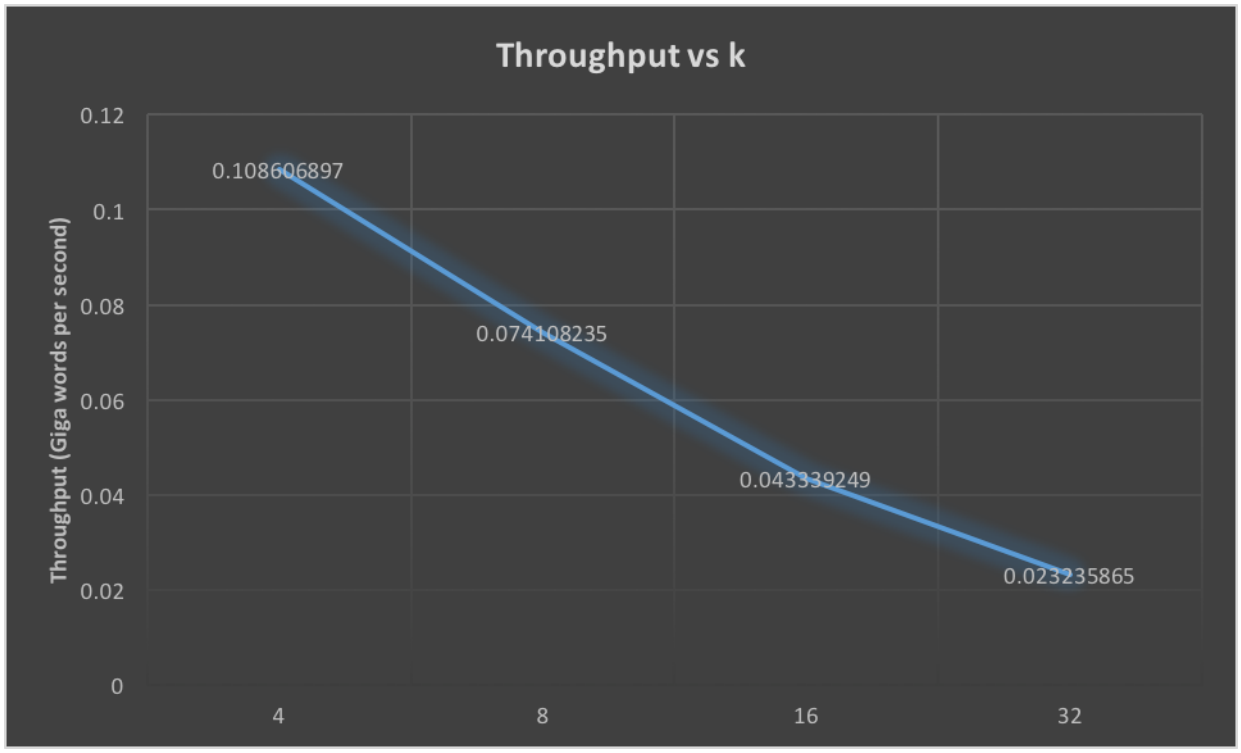
Design (k, p, b, g)	Critical Path
8, 1, 8, 1	Matrix memory to pipeline register
8, 1, 12, 1	Pipeline register to output of mac
8, 1, 16, 1	Matrix memory to pipeline register
8, 1, 20, 1	Pipeline register to output of mac

7. Evaluation as K changes





K	Throughput(giga words per second)	c	Clock Period	Frequency	Frequency*k
4	0.108606897	29	1.27	0.7874	3.1496
8	0.074108235	85	1.27	0.7874	6.2992
16	0.043339249	293	1.26	0.79365	12.6984
32	0.023235865	1093	1.26	0.79365	25.3968



8. Comparison of designs

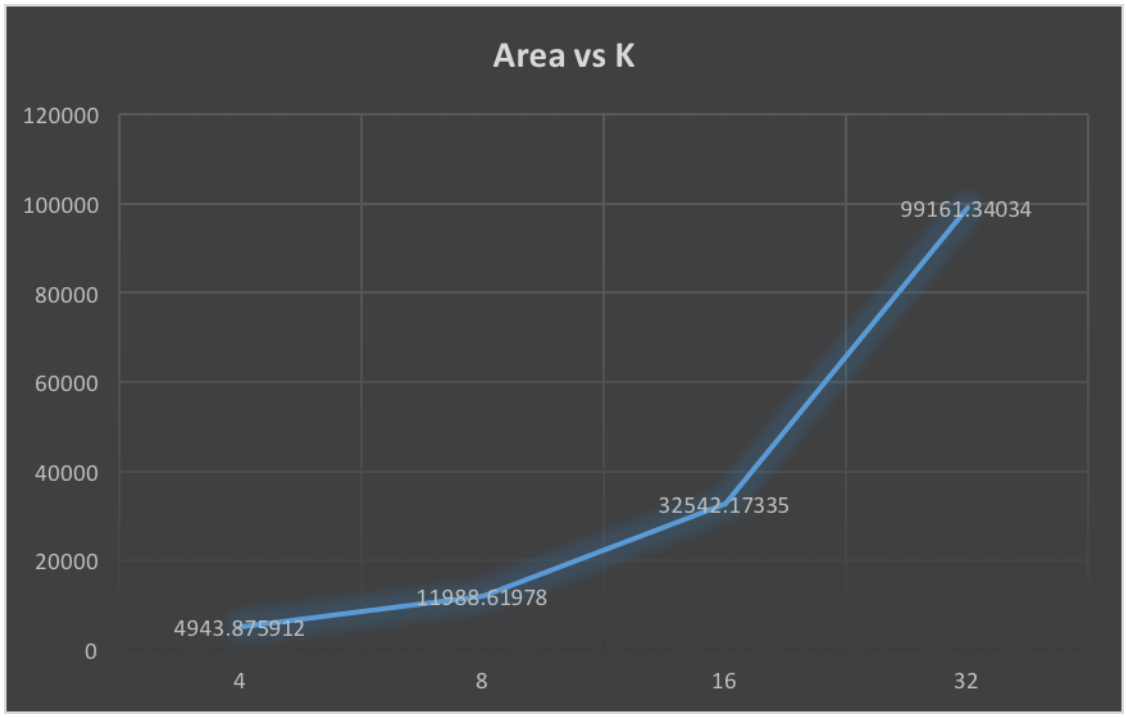
Clock	Design	Area	Power(u)	PC factor
1.27	4 1 8 1	2840.613944	1581.1	1.753169301
1.26	32 1 8 1	77003.80679	51842	0.00198796
1.53	4 1 8 0	2786.881947	1271.1	1.844791551
1.53	32 1 8 0	76937.57279	42914	0.001979284

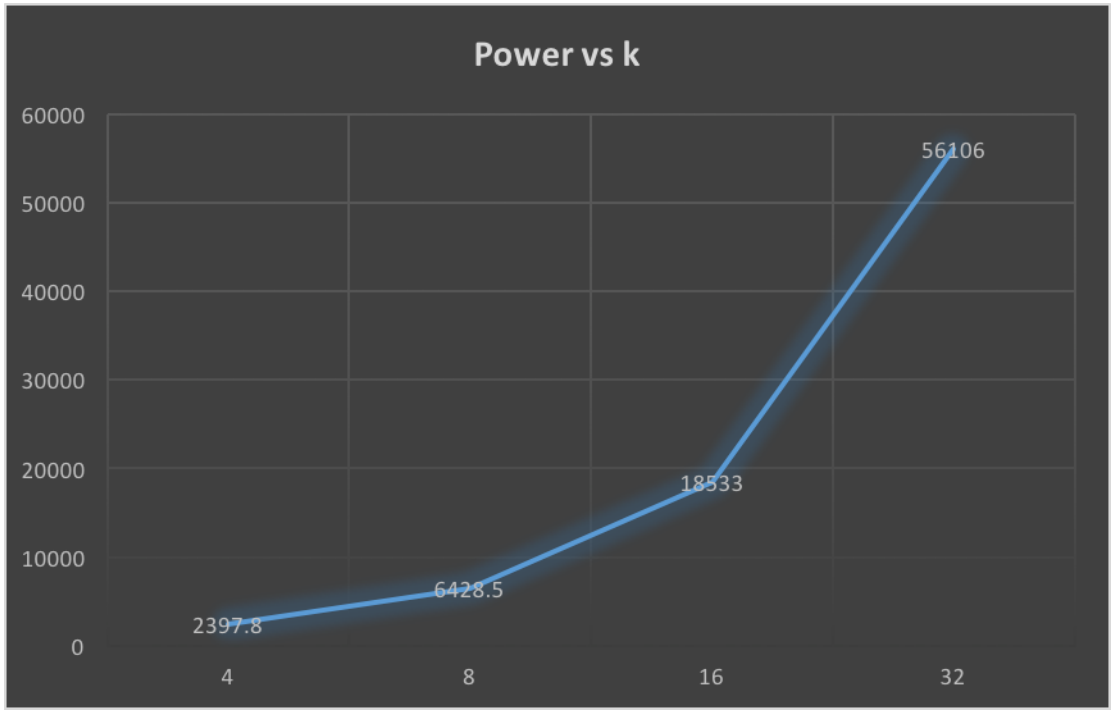
PC Factor = Performance cost factor = (f/(area*power))

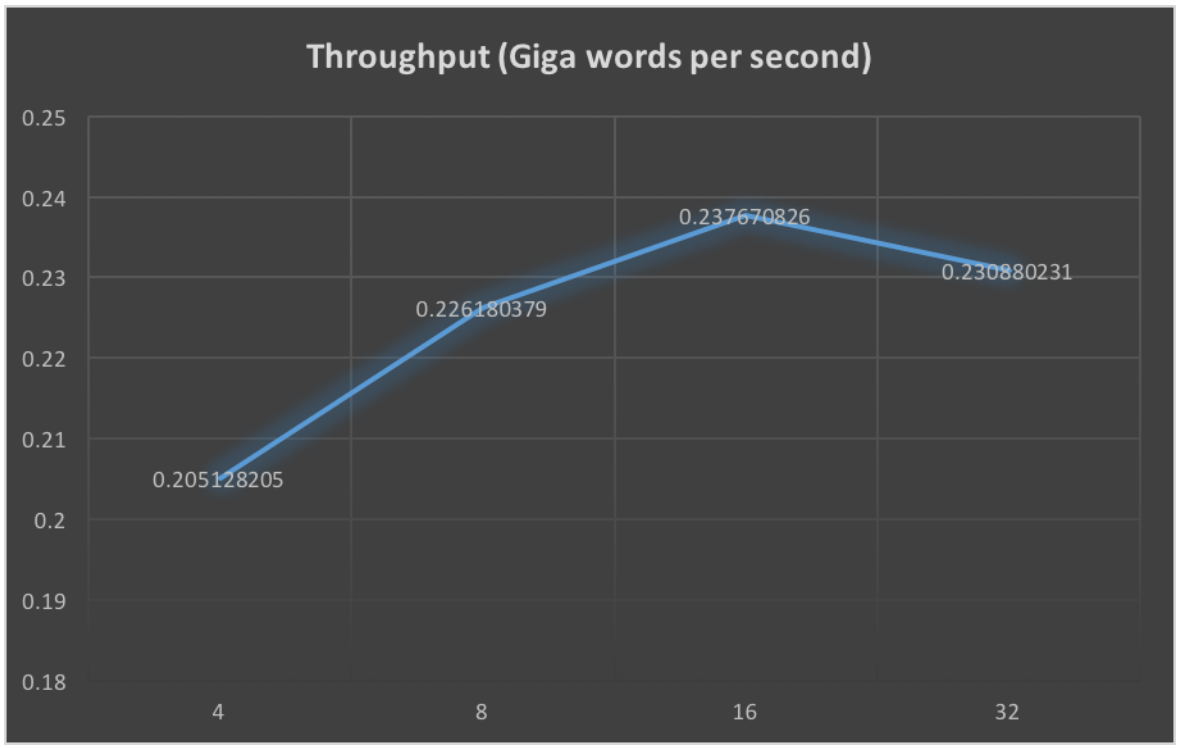
Design	Critical Path
4 1 8 1	Vector to pipeline
4 1 8 0	Vector memory to Mac
32 1 8 1	Vector to pipeline
32 1 8 0	Vector memory to Mac

The usage of pipelining breaks and reduces the critical path thereby enabling better throughput and frequency.

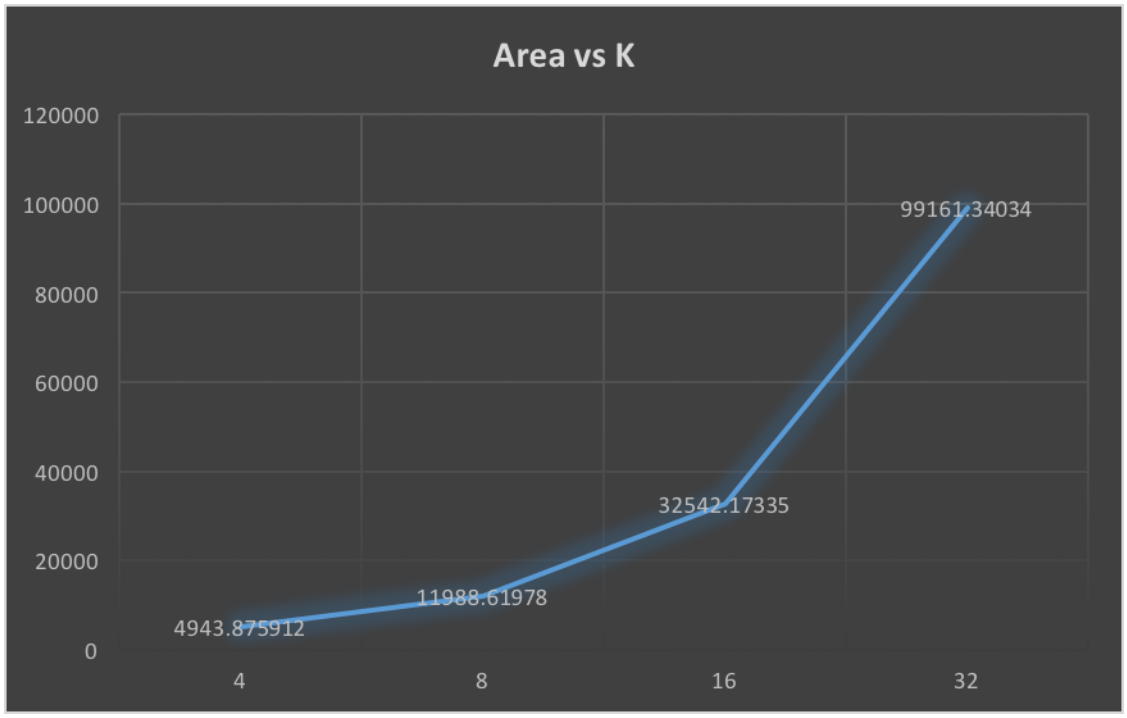
9.

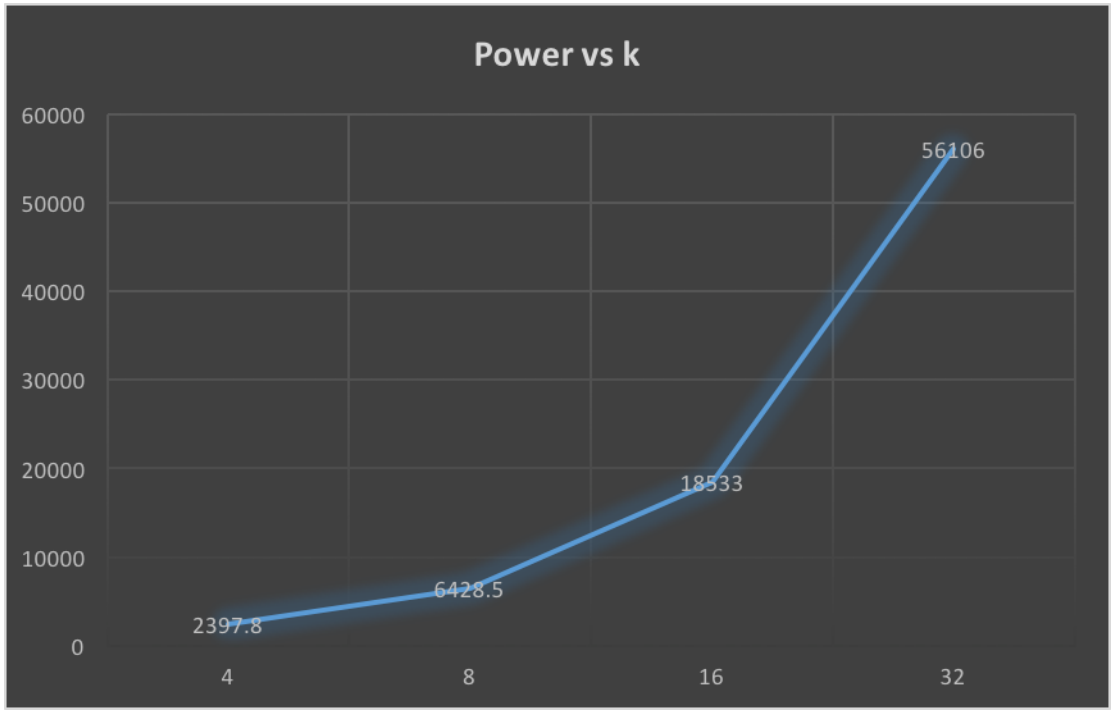

Area vs K
120000
100000
80000
60000
40000
20000
0
4
8
16
32
4943.875912
11988.61978
32542.17335
99161.34034

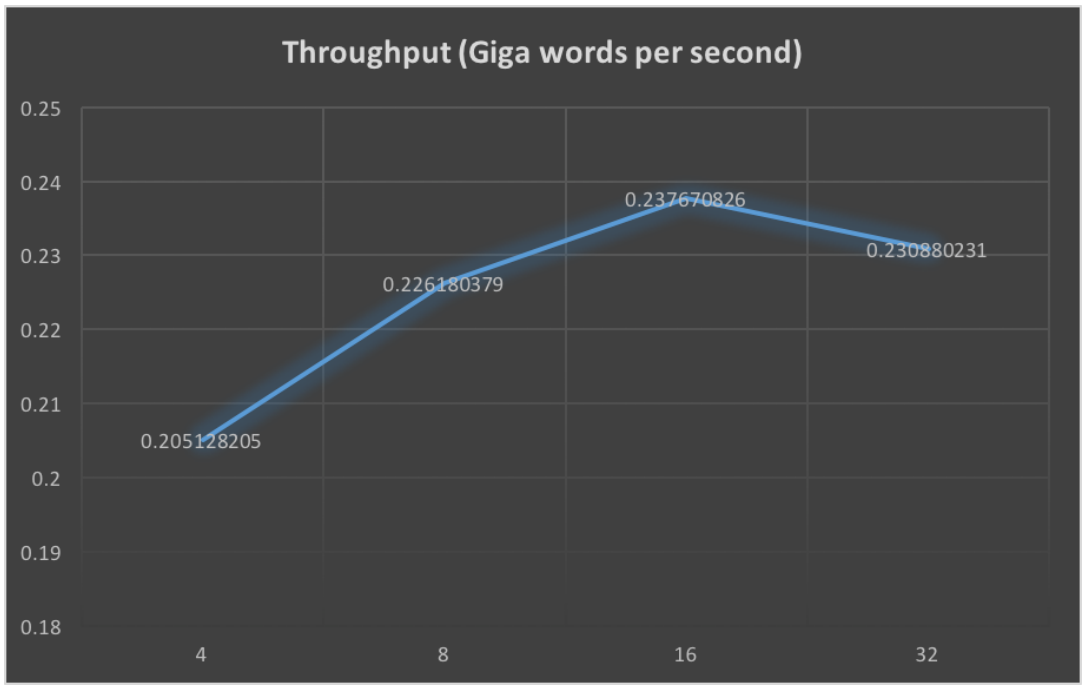

Power vs k
60000
50000
40000
30000
20000
10000
0
4
8
16
32
2397.8
6428.5
18533
56106


Throughput (Giga words per second)
0.25
0.24
0.23
0.22
0.21
0.2
0.19
0.18
0.205128205
0.226180379
0.237670826
0.230880231
4
8
16
32

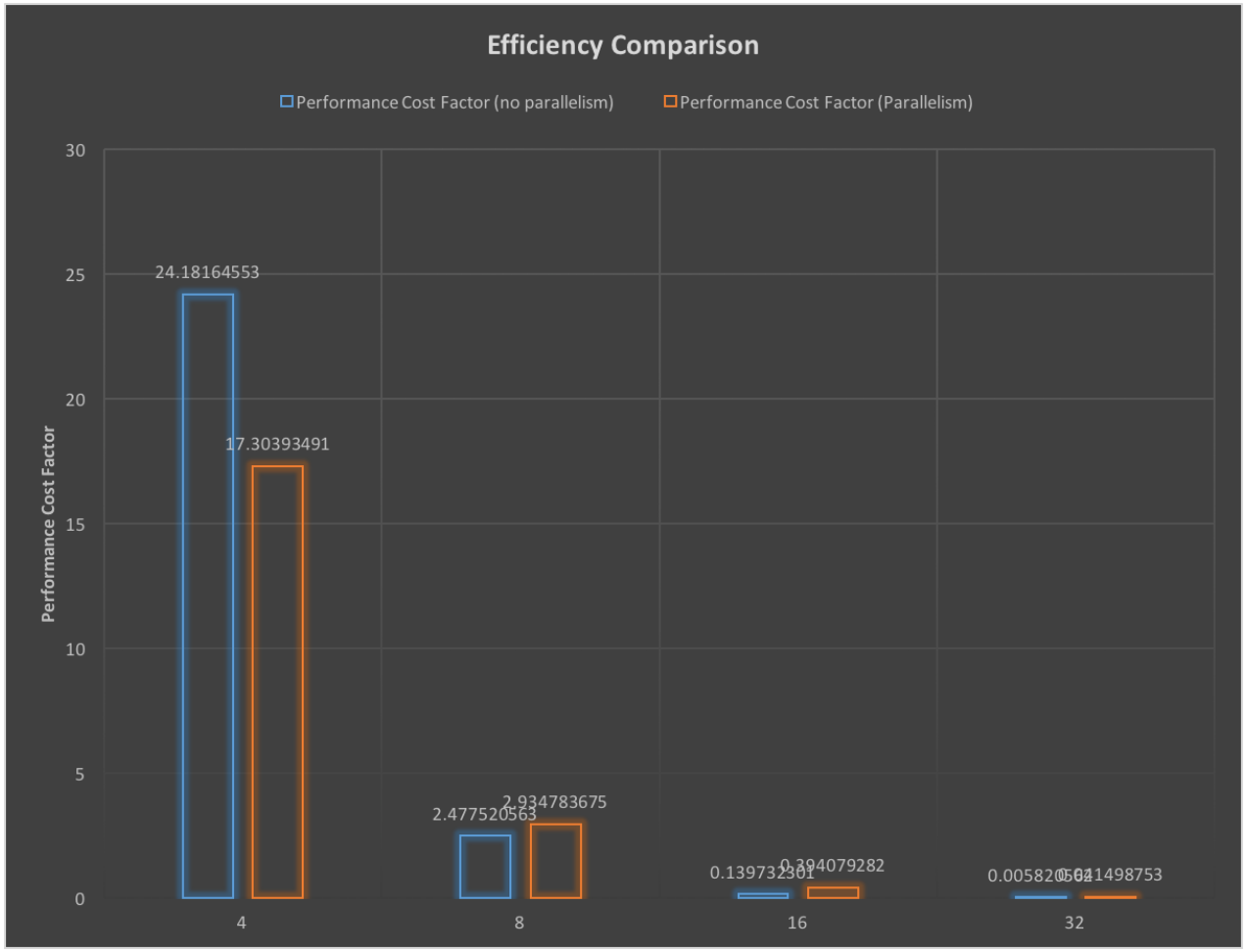
10. Evaluation of Parallelsim






Throughput (Giga words per second)
0.25
0.24
0.23
0.22
0.21
0.2
0.19
0.18
0.205128205
0.226180379
0.237670826
0.230880231
4
8
16
32

Which is more efficient??



The process of choosing one over the other really depends on the requirement of the user. A proper call can be made only based on which design constraint (area, power, speed) is more important for the designer.

Here I attempt to make a design choice for best overall efficiency. I define a parameter or comparison factor called **Performance Cost Factor** which is defined by Throughput / (Area * Power). The higher this factor the better the overall design choice. Ideally for this value to be high throughput should be high and Area and Power should be minimum.

I compute this factor for the designs with and without parallelism and plot them as a graph to compare. It is observed that for a lower k value the the design without parallelism seems to a better choice by a large margin. However, as

the K value increases this margin decreases and the parallel design seems to be the better choice by a small difference.
Whether this difference will go in favor of parallel design as K further increases is speculation as we don't have data for them.

For the given k values 4,8,16,32 it is clear that unless performance and speed is very very important to the designer the parallel design isn't a good choice.
The design without parallelism provides better overall tradeoff for these k values.