Digital Systems Design coursework 2

TszHang Wong CID:01864937 Email: thw20@ic.ac.uk , Bryan Tan CID:01861592 Email: bet20@ic.ac.uk

February 17, 2023

1 Introduction

In this report the performance of different configurations of a softcore NIOSII processor were evaluated. The target applications are the computation of the mathematical functions described in Eq.1 and Eq.2. Evaluation was conducted on the configuration of the cache, the usage of embedded multiplication blocks, and external memory. The aim is to obtain a baseline model that will serve as a fair comparison to future customised hardware accelerator solutions of the aforementioned functions.

$$f(\vec{x}) = \sum_{n=1}^{N} x_i + x_i^2 \tag{1}$$

$$f(\vec{x}) = \sum_{n=1}^{N} (0.5 \times x_i + x_i^2 \times \cos{(x_i - 128)}/128) \tag{2}$$

In addition, various code optimization techniques were explored in order to further improve the performance of the software application. In order to explore the trade-off between accuracy and latency, we discussed the implementation of the cosine function using different methods, such as the Iterative Taylor Series and Look-up table. The insights gained from the findings can be used to inform the selection of an appropriate implementation method based on the specific requirements of a given application.

The results show that tuning the hardware and software of a softcore CPU can have a significant impact on performance.

2 Storing the Program and Data on External Memories (Task 3)

Off-chip memory refers to a memory location that is not instantiated on the FPGA fabric and belongs to external components on the development board. It typically has a larger capacity than on-chip memory but comes with higher memory access latency. The on-chip memory, instantiated by BRAM cells, has a maximum size that is limited by the number of memory bits available on the FPGA. In the case of DE1-SoC board, the maximum on-chip memory is 500KB. However, this is not sufficient for executing testcase 3, as the 261121 floating point input vector alone requires at least 261121*4 bytes of memory for storage in the stack. Thus, an external memory is required to accommodate the larger memory demands of the test case.

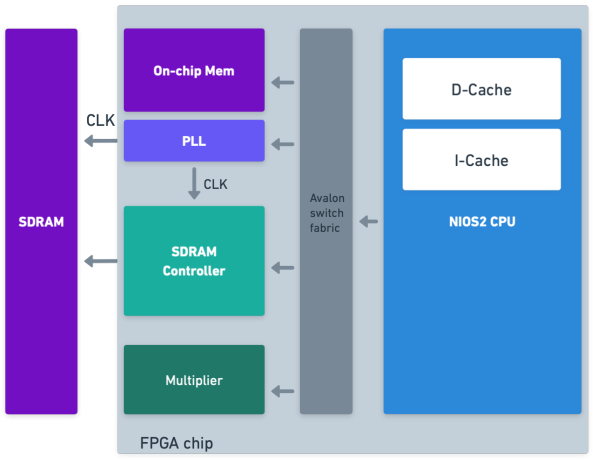


Figure 1: High level overview of NIOS II architecture on FPGAs

The DE1-SoC board includes various types of external memories, with the Synchronous Dynamic Random-Access Memory (SDRAM) being the one used for evaluation purposes. SDRAM is a type of DRAM that has a clocked input/output interface, and since each memory cell contains only one capacitor and one transistor, it is much cheaper to manufacture on a large scale compared to on-chip memory. However, SDRAM has two main drawbacks: firstly, the capacitors experience static leakage so periodic refresh is required to maintain the memory, during which memory access is prohibited. Secondly, unlike on-chip memory, SDRAM requires a controller to communicate with the CPU. The controller generates a busy output which the CPU can use as handshake control, avoiding the need of fixed wait states.[1] However, it also results in a performance penalty.

Since the CPU and SDRAM are located in different areas, clock propagation delay between them must be taken into account to ensure timing is met and to prevent data loss or corruption during data transfers. To address this, a Phase-Locked Loop (PLL) shown in Fig.1 is used to synchronize the clock signals between the CPU and SDRAM, ensuring that they are phase-aligned. It generates a clock signal that is locked to the incoming reference clock signal, thus ensuring that the two clocks are synchronized. By using a PLL, data integrity can be maintained during data transfers, despite the delay caused by clock propagation between the CPU and SDRAM.

2.1 SDRAM resources evaluation

When evaluating the hardware implementations, our resource metric is given by Eq.3. We desire an implementation that reaches a competent level of accuracy and latency with minimal resources.

$$ResourceUsage = \frac{1}{3}\left(\frac{LE}{TotalLEondevice} + \frac{EM}{TotalEMondevice} + \frac{MB}{TotalMBondevice}\right) \quad (3)$$

Table.1 illustrates the FPGA resource utilization for designs that use on-chip memory and SDRAM. The Table.shows a significant decrease in memory bit utilization for the SDRAM-based design, as SDRAM is a directly embedded memory block on the board and does not rely on memory bits like on-chip memory. However, it is worth noting that the SDRAM's controller requires extra logic elements, resulting in a slight increase in ALM usage. The Table.also indicates the usage of the PLL, which is used to ensure that the CPU and SDRAM clocks are synchronized. Neither design uses any embedded multipliers. Overall the SRAM design achieves a lower resource utilization (Table.2).

	On-chip Memory	SDRAM
Logic utilization (in ALMs)	1,341 / 32,070 (5%)	1,566 / 32,070 (5%)
Total block memory bits	23%	1%
Total DSP Blocks	0 / 87	0 / 87
Total PLLs	0 / 6	1 / 6

Table 1: Resource Utilisation Report: 2KB I-cache, 2KB D-cache, 50KB On-chip Mem vs 2KB I-cache, 2KB D-cache, SDRAM

	On-chip Memory	SDRAM
Resource Utilization	0.093	0.02

Table 2: Resource utilization score: On-chip memory vs SDRAM

2.2 SDRAM Software Performance

After obtaining a hardware design that was able to successfully execute all the testcases, we proceeded to evaluate the performance of the design across all the testcases used in Task2 as shown in Table.3. Since changing hardware design has no effect on software storage, the application and program size of all testcases remains unchanged compared to ones used for evaluating on-chip memory in Task2.

The design utilizing on-chip memory consistently outperformed designs that used SDRAM. This was particularly noticeable as the test cases with longer vector sizes (and therefore more memory accesses), with speedups of up to 41.4% observed at testcase 2 as depicted in Table.4. The performance difference between the two designs was most significant when the memory access was more frequent. The increase in latency observed in the SDRAM design aligns with the SDRAM characteristics we highlighted in Section 2.

While the use of external memory is crucial to enable the execution of large programs, we have decided to use this design as our baseline model for any further evaluation or improvement on the design. This means that without any specific requirements or specifications, any modifications or improvements to the design will be based on this baseline model.

Testcase	1	2	3
Application Size	762,102	762,118	762,406
Input Vector Element	52	2041	261121
Stack storage needed (KB)	52	2041	261121

Table 3: Testcases used for evaluating designs with SDRAM

Testcase	On-chip Latency(ms)	SDRAM Latency(ms)	Latency speed up (%)
1	1	1	0
2	30	51.2	41.4
3	X	7117.5	X

Table 4: Latency comparison across all test cases: On-chip vs SDRAM, 2KB I-cache, 2KB D-cache, Compiler Flag -O3

2.3 Cache Configuration

In the previous report, we observed that modifying the size of the instruction cache (I-cache) and data cache (D-cache) of the NIOS II processor has an impact on the resource utilization of the system. This is because cache is instantiated on-chip using memory bits.

Cache size also has an effect on latency. In C program implementation of (1), there is spacial locality as it loops over an array that is stored as a contiguous sequence in memory and performs one operation per element. Based on the data presented in Figure 2, it can be inferred that a 4KB D-cache is sufficient to capture most of the spacial locality, and increasing the cache size further only has a marginal effect on latency.

Additionally, the program exhibits temporal locality as most of the code spends its time within a loop. This means that the same instructions are repeatedly accessed and executed. It is shown that a 4KB I-cache is sufficient to contain all the instructions in this hot loop in Fig.2 as a further increase show no improvement in latency.

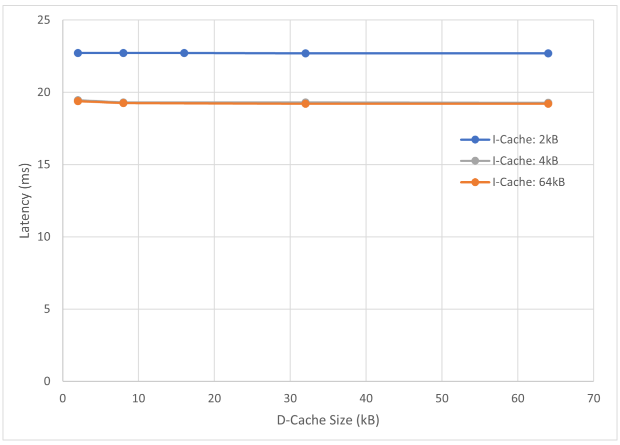
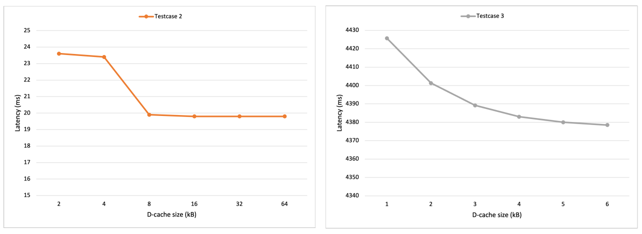


Figure 2: Latency for different cache configurations, 64KB I-Cache line overlapped with 4KB I-Cache line.

The same idea applies to the design using SDRAM. In order to find an efficient cache configuration for it, one approach is to perform an iterative search to determine the optimal D-cache size, while keeping the I-cache size at its maximum value. This helps to avoid potential bottlenecks and ensures that the I-cache can accommodate the maximum number of instructions. Fig.4 indicates a sudden drop in latency at 8KB. This implies that when there is insufficient cache to store data, the processor has to make frequent accesses to SDRAM to resolve any cache miss, which is characterized by higher delays than on-chip memory. As a result, the performance of the system is negatively impacted.

However, once the D-cache size reaches the 8KB threshold, a competent level of latency is achieved for both test cases. A further increase in size only gives no improvement in test case 2 and small improvement in test case 3. This is because 8KB D-cache can store most of the frequently accessed data, thereby reducing the number of accesses to SDRAM. Consequently, the performance difference between on-chip memory and SDRAM becomes less significant.



(a) test case 2 (b) test case 3

Figure 3: Latency for differenct D-cache under 64 I-cache for different test cases

The 8KB D-cache was used to find an efficient I-cache size through iterative testing. Figure 4 shows that the stagnation point occurs at 8KB, where the cache is sufficient to store all the repeatedly used instructions for both test cases. Therefore, an 8KB I-cache and 8KB D-cache are found to be the efficient configuration.

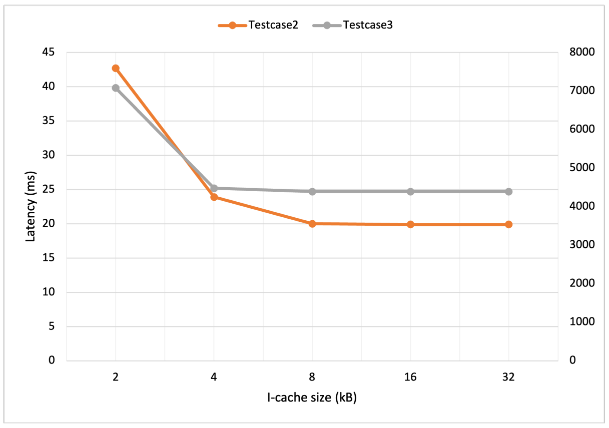


Figure 4: Latency for differenct I-cache under 8KB D-cache

3 Computing a Complex Mathematical Expression (Task 4)

In this task we explore different software implementations of Eq.2 and analyse the latency/accuracy trade-off between the designs. All code was compiled with full compiler optimization (-O3). Four testcases were used: testcases 1 to 3 uniformly span the input range of [0-255] with different step sizes. Testcase 4 randomly samples the input range.

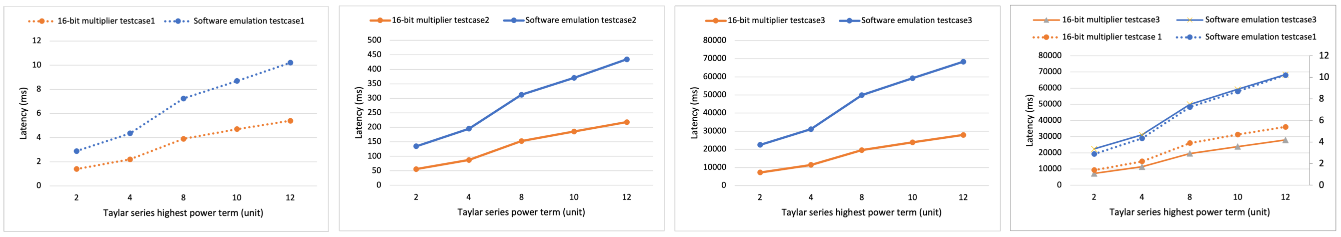
3.1 In-built cos functions

The first implementation is a naive translation of Eq.2 into C code. We use the `double cos(double)` function (double precision input and output) from the math.h library. The implementation of math.h is platform specific, and we were not able to find the documentation for NIOS2.

```c
float task4_function(float x[], int M) {
    int i; float result = 0;
    for (i=0; i<M; i++)
        result += 0.5 * x[i] + x[i] * x[i]
            * cos((x[i]-128)/128);
    return result;
}
```

Listing 1: First Implementation

The output of the function (List.1) was compared against both a Python single precision and double precision implementation, shown in table. 5. The NIOS2 output and Python single precision were identical - implying that the NIOS2 compiler can accurately emulate floating point operations in software. The error between the double and single precision results increases with problem size - this makes sense as each single precision operation introduces error, larger vectors (which require more operations) accumulate more error.



(a) Testcase 1 (b) Testcase 2 (c) Testcase 3 (d) Testcase 3, 1 comparison

Figure 5: Latency between 16-bit multiplier vs software emulation across testcase

Testcase	1	2	3	4
Problem Size	52	2041	261121	2323
Result single precision	920413.500	36123104.000	4621531136.000	41571204.000
Result double precision	920413.627	36123085.552	4621489017.889	41571244.486
Error (%)	1.37601E-05	5.10699E-05	0.000911354	9.73902E-05
Latency (ms)	20.1	831	119387	1139

Table 5: Accuracy comparison between implementation result(single precision) and Python (double precision) with latency achieved.

Several code optimizations can be applied to List.1 to improve the latency for each testcase. Firstly, it is in fact advantageous to store the result of the computation as a double rather than a float (by changing the type of variable **result** on line 2). In C, a mathematical expression is evaluated at the highest precision of its terms (before the computation all terms with lower precision are cast to the highest precision). Therefore, because cos() returns a double, the computation in lines 4 and 5 is evaluated in double precision, and the result is only cast to single precision at the end of each iteration of the loop. Having **result** as a double means the casting between double and float is avoided, and there is a small performance increase across all testcases (table. 6). The result of the function is also accurate to the Python double precision result.

Secondly, using the `float cosf(float)` function (I/O are both floats) from the math.h library ensures all operations will be done at single precision; this achieves a significant latency decrease of 57.5% averaged across all testcases (Table 6).

Finally, factorizing `x[i]` out of the computation (as in List.2) means there is one less multiplication operation performed per iteration. Using this optimization, a further 1% decrease in latency is achieved.

```
result += x[i] *
    (0.5 + x[i] * cosf((x[i]-128)/128));
```

Listing 2: Factorizing the expression

In the following sections, we investigate alternative methods of implementing the cosine function in software. We will use the result of the function in List.2 and its achieved latency as the benchmark for comparison.

3.2 Taylor Series Implementation

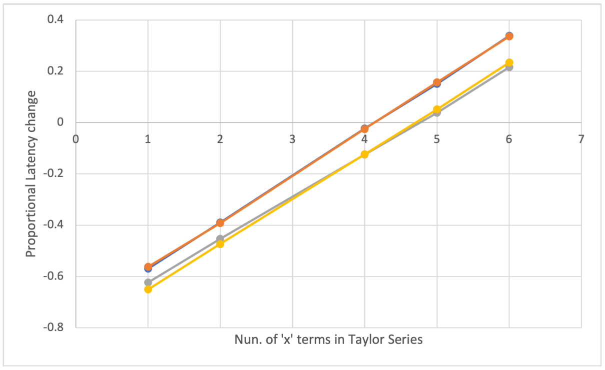
The cosine function can be expressed as an infinite polynomial series using the Taylor series, as demonstrated by Eq. 4.

$$\cos x = 1 - \frac{x^2}{2} + \frac{x^4}{4} - \frac{x^6}{6} + \frac{x^8}{8}... \quad (4)$$

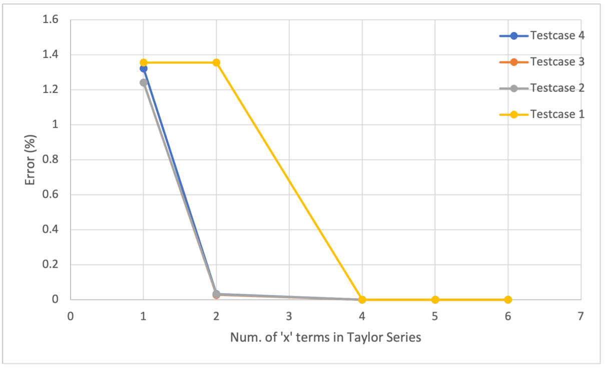
Since computing the infinite summation is impossible, approximation achieved by neglecting the high order terms. A naive implementation of cosine using a Taylor series with three `x` terms is shown in List.3. The elements of the input vector `x` is bounded between [0-255], and therefore analysis of Eq. 2 shows that the input to the cosine function is bounded between [-1, 1]. This means the Taylor series as expressed in List.2 is guaranteed to converge without needing additional code to account for periodicity.

```
float cos6(float x) {
    float y = 1 - x*x/2 + x*x*x*x / 24
    - x*x*x*x*x*x / 720;
    return y;}
```

Listing 3: Cosine as a Taylor Series using 3 terms



(a) Proportional Latency Change vs the numbers of terms



(b) Error vs the numbers of terms

Figure 6: Performance difference vs the numbers of terms in native Taylor series across difference test cases

We investigated the latency and accuracy of the Taylor series implementation with different numbers of `x` terms.

Fig. 6a shows the proportional latency difference of the Taylor series implementation compared to the function in List.2. There is a positive linear relationship between the number of terms used in the Taylor series and latency. This implies that the time to compute each term in the Taylor series is constant. This is interesting, as we expected higher powers of `x` to take longer to calculate. Consecutive terms increase by a factor of `x*x`, requiring two more multiplications to compute. By this logic, we expected a quadratic relationship between the number of terms used and latency.

Therefore, it is likely that the compiler has optimized the code so that consecutive terms take nearly the same time to compute. One way of doing this is by reusing intermediate results. For example, in List.3, `x^6` may be calculated as `x^2 * x^4` which have already been computed, decreasing 5 required multiplications to just 1.

Fig. 6b shows the percentage absolute error of the Taylor series result compared against the ground truth single precision result in Table 5. As more terms are used in the Taylor series, there is a large error decrease initially, but then it takes many more terms for the result to converge to correct answer. Five terms are required to achieve full accuracy on all test cases (Table 7), however the latency of this function is marginally greater than the function in List.2.

	Mean change in latency (%)
Casting to Double	-0.007717928
Using Cosf	-0.57578393
Cosf + Factorising x	-0.581379748

Table 6: The difference in latency between each implementation method and the built-in cos function, averaged across all 4 testcases

Testcase	1	2	3	4
Terms needed for full accuracy	2	5	5	2

Table 7: The numbers of terms in Taylor series needed to achieve full accuracy for eachtest case

3.3 Alternative Taylor Series Implementations

A key bottleneck in the Taylor series calculation is the exponentiation of x. The compiler is able to optimize the calculation by reusing intermediate results. We investigate two ways of explicitly doing this in code.

Each term in the Taylor series of the cosine function can be expressed recursively as in Eq. 5. Each successive term therefore requires only 3 more multiplication operations to compute.

$$a_{i+1} = a_i * -1 * x * x/2i \text{ with } a_0 = 1 \tag{5}$$

The code in List.4 takes advantage of this idea. It calculates each term of the Taylor series in a loop and adds it to a running sum.

```
float cos_running(float x, int y){
    float result = 1.0;
    float inter = 1.0;
    float num = x * x;
    for (int i = 1; i <= y; i++)
    {
        float comp = 2.0 * i;
        float den = comp * (comp - 1.0);
        inter *= num / den;
        if (i % 2 == 0)
            result += inter;
        else
            result -= inter;
    }
    return result;}
```

Listing 4: Iterative Taylor Series: running sum

The code in List.4 can be further improved by unrolling the for loop and hard-coding intermediate sharing of terms. Loop unrolling gets rid of the instructions used to keep track of the loop iteration, improving latency. List.5 shows an example of this for the Taylor series with four `x` terms.

```
float cos8_f(float x){
    float xx = x*x;
    float xxxx = xx*xx;
    float y = 1 - xx/2 + xxxx / 24
    - xxxx*xx / 720 + xxxx*xxxx /40320;
    return y;}
```

Listing 5: Efficient Taylor Series: unrolling the iterative loop with 4 terms

Both the iterative method (List.4) and its unrolled version (List.5) generate the same output as the naive Taylor implementation (List.3). Fig. 7 and 8 display the percentage reduction in latency of the iterative and unrolled methods compared to the naive Taylor implementation respectively. As the number of terms in the Taylor series increases, the reduction in latency becomes more significant, with the unrolled method achieving a 20% improvement in latency for a Taylor series with 5 `x` terms. Comparing the two methods, the unrolled version provides slightly better latency performance than the iterative method.

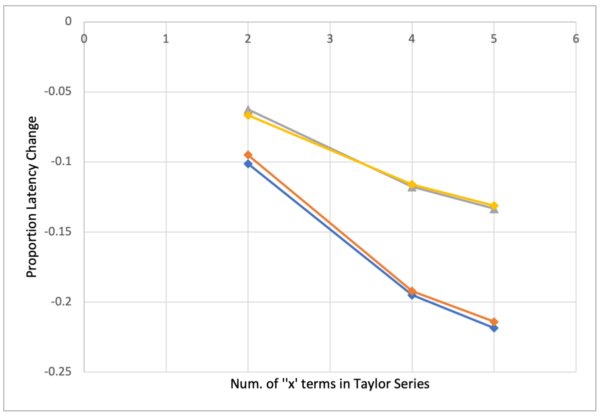


Figure 7: Iterative Taylor series: Proportional Latency Change compared to Naive Taylor

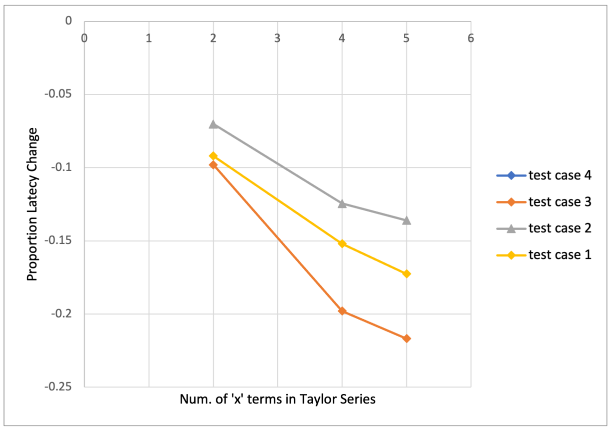


Figure 8: Efficient Taylor series: Proportional Latency Change compared to Naive Taylor

3.4 Lookup Table Implementations

An alternative method for implementing the cosine function is to use a lookup table Instead of computing the result at runtime, the lookup table method uniformly discretizes the input range, pre-computes the results for the selected input values, and stores them in an array.

When the function is called, the input is first rounded to the nearest table entry, then the result is simply obtained by performing a lookup in the table (accessing the array). As there is no computation being done this should significantly reduce the latency.

List.6 shows the Python script used for generating lookup tables. Given that the input range to the cosine function in the application is [-1, 1], and cosine is an even function, our lookup tables only need to span an input range of [0, -1]. The `main_wrapper(n)` function generates a lookup table for cosine with inputs evenly separated by `1/n`.

```python
from math import cos, pi

def main(f, PRECISION, NAME):
    f.write("float %s[] = {\n" % NAME)
    j = 0
    p = 0.0
    while True:
        f.write("{:.20f}, ".format(cos(p)))
        j += 1
        p += PRECISION
        if p > 1:
            break
    f.write("};\n")
    f.write("const int %s_size = %d;\n"
        % (NAME, j+1))
    f.write("float tb_step = %f;\n"
        % (1 / PRECISION))

def main_wrapper(n):
    main(open(f"costable_1_{n}.h", "w"), (1/n), f
    "costable_1_{n}")
```

Listing 6: LUT generation script in Python

List.7 shows an example of a generated lookup table in C. The lookup table itself is stored as a static float array `costable_1_64`. `costable_1_64_size` and `tb_step` indicate the size of the table and the reciprocal of the step size between inputs respectively, are used by the function in List.8 to calculate the exact index of the table that corresponds to the input value.

```c
#define COSTBL = costable_1_64;
float costable_1_64[] = {1.00000000000000000000,
    0.99987793217100662257, ... };
const int costable_1_64_size = 66;
float tb_step = 64.000000;
```

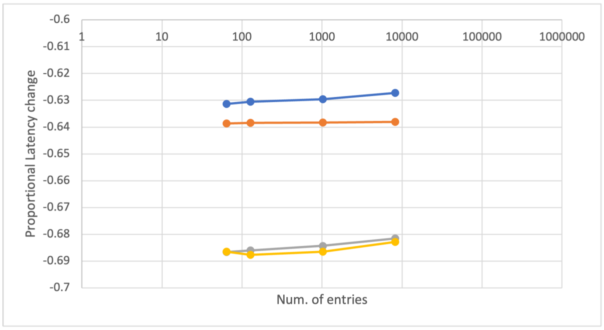
Listing 7: Example lookup table in C

```c
float cos_table(float x){
    x = fabs(x);
    return COSTBL[(int)(x * tb_step + 0.5)];
}
```

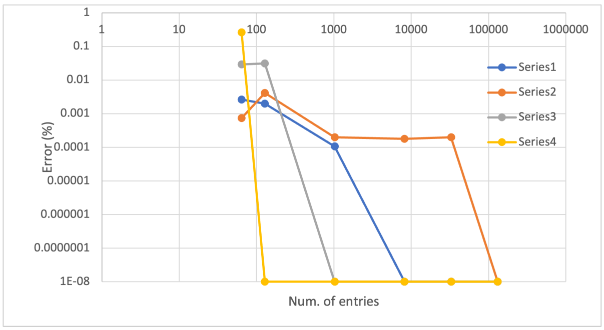
Listing 8: Function to access LUT

In software simulation, we determined that the smallest lookup table size required to achieve full accuracy on all test cases is 131,072 entries. However, due to a memory addressing issue in the NIOS2 processor, the maximum lookup table size that could be used is 8,192. Fig. 9b shows that the error in the computation decreases with the number of entries used.

Fig. 9a shows the latency change of using the lookup table method over the naive Taylor series implementation in List.2 - it is around a 60% improvement for all testcases. Latency increases very slightly for larger table sizes. In tables with more precision, there are more entries between desired inputs, so there is less spatial locality in the program, and more misses when accessing the data cache which stores the lookup table.



(a) Proportional latency change relative to cosf implementation



(b) Error vs numbers of entries. Note: Points on the graph achieving error of 1e-08% in reality achieve 0% error (fully accurate to the single precision result). These points have been given a non-zero error in order to plot them on the log scale.

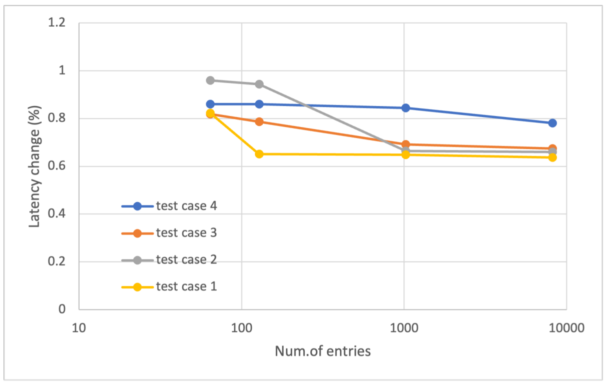
Figure 9: Latency and accuracy of the LUT for different sizes

We are not able to instantiate a lookup table large enough to achieve full accuracy for all testcases on the

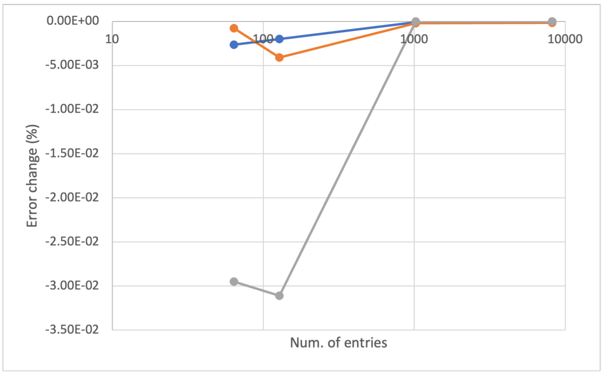
NIOS2 processor. One method of increasing the accuracy of the lookup table method is using linear interpolation (LERP). In essence, LERP approximates the target function as step-wise linear. Any input to the function falls between two entries of the lookup table The function in List.8 takes the value of the nearest entry as output, while LERP will take an average of the value of two entries, weighted proportionally by the distance of the input to each of the entries. List.9 implements an access to the lookup table with LERP.

```
#define interp(w, v1, v2)
    (1.0 - (w)) * (v1) + (w) * (v2)
float cos_table_interp(float x){
    x = fabs(x);
    double i = x * tb_step;
    int index = (int)i;
    return interp(i - index,  // weight
        COSTBL[index],        // lower value
        COSTBL[index + 1]     // upper value
        );
}
```

Listing 9: LERP Implementation



(a) Proportion latency change vs numbers of entries



(b) Error vs numbers of entries

Figure 10: Latency and accuracy of the LERP method for different table sizes

Fig. 10b plots the difference in error of the result when using a lookup table with LERP compared to without. There is a large improvement in error for small table sizes, for larger table. sizes the entries are closer together and the linear approximation between entries becomes less useful.

Fig. 10a plots the latency change in using a lookup table with LERP compared to without. In general, there is a mean 80% increase in latency, due to the extra time required to calculate the weighted average of the two entries.

3.5 Final comparison of implementations

Each of the explored implementations offers a latency to accuracy trade-off. Fig. 11 plots the latency of each implementation against the percentage error of the result for testcase 3 (the largest testcase with the worst accuracy for different parameterizations of each implementation).

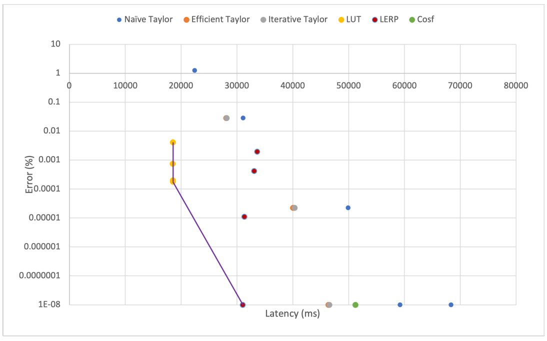


Figure 11: Testcase 3: Latency of each implementation vs the percentage error of the result. Note: as in Fig. 9, points on the graph achieving error of 1e-08% in reality achieve 0% error

The purple line indicates the Pareto front. The five designs on the front are the four different lookup table sizes tested, and also the lookup table of size 8,192 accessed using LERP. The 8,192 entry lookup with LERP achieves full accuracy on all testcases, with a 40% latency decrease on testcase 3.

The data suggests that lookup table methods are the most effective for our application in terms of achieving a flexible latency-accuracy trade-off. However, it is worth noting the possibility of systemic bias our experimental method. Each implementation was tested by running the function on a given input in a loop multiple times and recording the total latency. Between iterations, implementations using the lookup table will have already loaded it to the cache. This may not be the case if the function was used in a real world application. To get a better understanding of the performance of the lookup table implementations, it is necessary to test the implementation of the function in the context of a real world application.

table. 8 illustrates the varying application sizes achieved through different implementation methods. The built-in cos and cosf functions, as well as the factorizing method, were found to occupy the same amount of memory footprint. This is because all three methods require the inclusion of the math.h library, which takes up a significant amount of memory. In contrast, the Taylor series-based methods were found to have a smaller memory footprint as they do not require the use of math.h. However, a slight increase in application size was observed when the number of terms in the Taylor series was increased. This is primarily due the increase in code size. In addition, the look-up table. implementation displayed a significant variance in memory footprint occupation. This was not only due to an increase in code size but also because the initialized data consumed a considerable amount of space.

	Application size (KB)
Build-in Cos	83
Build-in Cosf	83
Factorising x	83
Taylor series Difference Number of term	75-78
Taylor series iterative version	76
Lookup Table (Difference table. size)	76-102

Table 8: Application size for each implementation.

4 Embedded Multiplier Block (Task 5)

The current design does not have hardware support for fixed point multiplication or any floating point operations, so these operations are done through processor's ALU with software emulation. In more detail, the emulation of fixed point multiplication is achieved by utilizing fixed point addition and subtraction, while floating point multiplication is accomplished through a combination of shifting and fixed point operations.

Hence, one solution to improve performance would be adding dedicated hardware that can perform multiplication more efficiently. The DE1-SoC board has 87 embedded multiplier blocks available. Since this multipliers come with dedicated hardware and optimized architecture, they are generally faster in performing multiplication than software emulation method. By explicitly specifying them in Platform Designer, NIOS II can utilize a maximum of 3 16-bit multipliers. After adding these multipliers to NIOS II, the resource utilization report (Table 9) indicates that the only difference compared to Task4 is the usage of floating point multipliers while all other parameters remain unchanged.

Logic utilization (in ALMs)	Total register
1,634 / 32,070 (5%)	2678
Total block memory bits	Total pins
28%	10%
Total DSP Blocks	Total PLLs
3 / 87 (3%)	1 / 6 (17%)

Table 9: Resource Utilisation Report: 64KB I-cache, 64KB D-cache, SDRAM, 3 16-bits multiplier blocks

Figure 5 compares the performance of the design using hardware multiplier and the design using software emulation. The test case is s. As the number of terms used in the Taylor series increases, the number of required multiplications also increases, causing multiplication to become a more significant bottleneck in the system. Moreover, since the size of the input vectors also increases across the test cases, test case 3, which has the largest size, has the highest floating point multiplication weight among all test cases.

In such cases, utilizing hardware support for multiplication can lead to a significant improvement in performance. The result shows that design with 16-bit multiplier always outperforms software emulation in all test cases, with 59.1% speed up achieved in test case 3.

The accuracy achieved by the hardware multipliers is identical to that achieved by NIOS II software emulation. Both implement the IEE 754 floating point specification faithfully.

Overall, the design choice of hardware or software-based system depends on many factors such as hardware resource availability, targeted application and size of the data. To achieve the best possible performance in a given system, it is crucial to evaluate these factors and consider the trade-off between solutions.

	16-bit multiplier	software-based
Resource Utilization	0.103	0.093

Table 10: Resource utilization score: 16-bit multiplier design vs software-based design

5 Conclusion

In conclusion, optimization on both software and hardware side has significantly improved the system performance. In term of hardware, the dedicated 16-bit multiplier allowed the system to perform multiplication operations more efficiently, reducing the computational burden on the processor, resulting in a faster system that can handle larger and more complex calculations more efficiently. On the software side, code optimization that reduce the overall computation complexity was conducted and the trade-off between accuracy and latency was carefully considered. Overall, the combination of software and hardware optimization has significantly enhanced the system performance on the given application.