Embedded Computer Architecture Assignment 1

Siddhant Rajashekar Honnalli
Department of Mathematics and Computer Science
Student number: *1328492*
s.r.honnalli@student.tue.nl

I. Introduction

In this report the optimisations after the bypass explained in the assignment exercise are documented. The method of optimising each part of the code from the beginning (i.e. local store of arrays a and b) to the end (i.e for loop involving variable yy) is followed.

II. Optimization of storing static global arrays in local memory

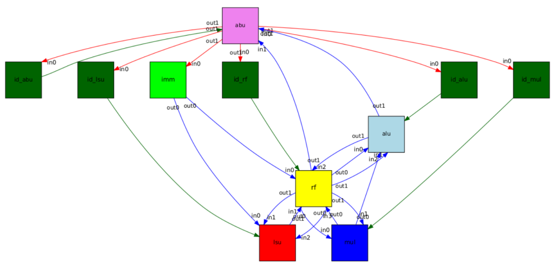


Fig. 1. Improved architecture

In the original assembly code, the arrays a and b are stored in the local memory by explicitly incrementing the addresses of a and b, storing them into register file and then later storing the values in the memory at the computed locations. This method has a lot of overhead and dependencies on ALU, RF, imm and LSU functional units.

This dependency can be reduced by using implicit store method provided by ISA of LSU in CGRA. Although, this is not the dominating part of the code and does not improve the efficiency of computation a lot, this method reduces the dependencies to just between LSU and imm units and is used in further sections of the code. In this method, the start address and stride is stored in the configuration registers r4 and r5. Instruction, sli WORD, inA stores the inA value in memory and increments the address implicitly by the value stored in r5. Figure 2 shows a code snippet of the optimization. The same optimization is performed for storing local static arrays x and y in local memory. The improved efficiency can be observed in figure 3.

	imm 0 ; coeff start address
srm r4, in0	imm 4 ; stride
srm r5, in0	imm 65336 ; a[0]
sli WORD, in0	imm 0 ; a[1]
sli WORD, in0	imm -64572 ; a[2]

Fig. 2. Implicit store instructions

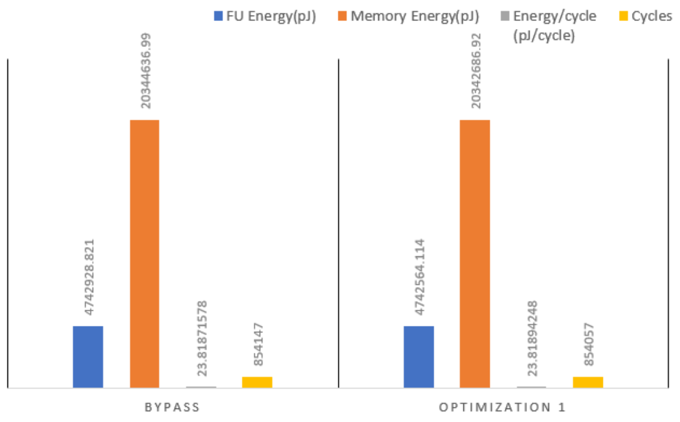
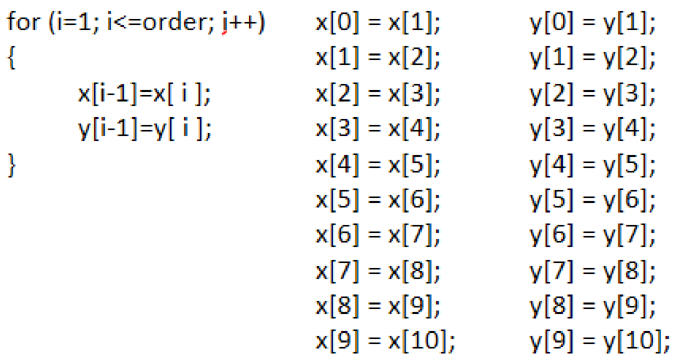


Fig. 3. Optimization 1

III. Optimization of x[i-1]= x[i] using implicit load/store and loop unrolling

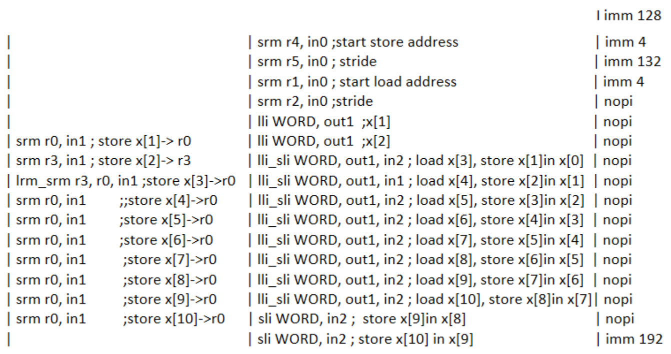
Loop unrolling is a loop transformation technique that helps to optimize the execution time of program. Since in the given program there are fixed number of iterations, the loop can be completely unrolled to reduce the loop overhead. This eliminates the dependency on ABU unit and increases efficiency. Also, unrolling along with implicit load/store instructions increases the efficiency together by almost 1.5x times as this a dominant part of code which loops over 2000 times in the while loop.



```
for (i=1; i<=order; i++)          x[0] = x[1];        y[0] = y[1];
{                                 x[1] = x[2];        y[1] = y[2];
        x[i-1]=x[ i ];            x[2] = x[3];        y[2] = y[3];
        y[i-1]=y[ i ];            x[3] = x[4];        y[3] = y[4];
}                                 x[4] = x[5];        y[4] = y[5];
                                  x[5] = x[6];        y[5] = y[6];
                                  x[6] = x[7];        y[6] = y[7];
                                  x[7] = x[8];        y[7] = y[8];
                                  x[8] = x[9];        y[8] = y[9];
                                  x[9] = x[10];       y[9] = y[10];
```

Fig. 4. Loop unrolling

In the CGRA assembly code, the code structure used in fig 5 implements the above mention method for array x. The same instructions apply for array y, except for changing the local start load/store addresses. The stride, local start store and local addresses are stored in the configuration registers r4, r5, r2 and r3. Initially, the values x[1] and x[2] are loaded by using lli instruction and stored in registers r0 and r3 in register file(RF) unit. The first lli_sli instruction loads x[3] onto data path and stores x[1] at address location 128(address of x[0]). In the next cycle, r0 stores x[3] value and x[2] value is stored at loaction 132(address of x[1]). This is repeated until value of x[10] is stored in address location of x[9]. The advantage of using register r0 in RF is that once the value is stored in that register, the value is available implicitly on the data path in the next cycle. This eliminates the need of explicitly using lrm instruction to load data.



```
                                                                                    I imm 128
|                                    | srm r4, in0 ;start store address              | imm 4
|                                    | srm r5, in0 ; stride                          | imm 132
|                                    | srm r1, in0 ; start load address              | imm 4
|                                    | srm r2, in0 ;stride                           | nopi
|                                    | lli WORD, out1  ;x[1]                         | nopi
| srm r0, in1 ; store x[1]-> r0      | lli WORD, out1  ;x[2]                         | nopi
| srm r3, in1 ; store x[2]-> r3      | lli_sli WORD, out1, in2 ; load x[3], store x[1]in x[0]  | nopi
| lrm_srm r3, r0, in1 ;store x[3]->r0 | lli_sli WORD, out1, in1 ; load x[4], store x[2]in x[1]  | nopi
| srm r0, in1       ;;store x[4]->r0 | lli_sli WORD, out1, in2 ; load x[5], store x[3]in x[2]  | nopi
| srm r0, in1       ;store x[5]->r0  | lli_sli WORD, out1, in2 ; load x[6], store x[4]in x[3]  | nopi
| srm r0, in1       ;store x[6]->r0  | lli_sli WORD, out1, in2 ; load x[7], store x[5]in x[4]  | nopi
| srm r0, in1        ;store x[7]->r0 | lli_sli WORD, out1, in2 ; load x[8], store x[6]in x[5]  | nopi
| srm r0, in1        ;store x[8]->r0 | lli_sli WORD, out1, in2 ; load x[9], store x[7]in x[6]  | nopi
| srm r0, in1        ;store x[9]->r0 | lli_sli WORD, out1, in2 ; load x[10], store x[8]in x[7]| nopi
| srm r0, in1        ;store x[10]->r0 | sli WORD, in2 ;  store x[9]in x[8]            | nopi
|                                    | sli WORD, in2 ; store x[10] in x[9]           | imm 192
```

Fig. 5. Loop unrolling and implicit load/store

The effect of this implementation is observed in the graph shown in fig 6. There is an improvement of 35.8% of total cycles compared to bypass optimization and 38.8% compared to naive implementation. The decrease in energy consumption by functional units is 26.7% and by memory unit is 32.73% when compared with optimization 1.

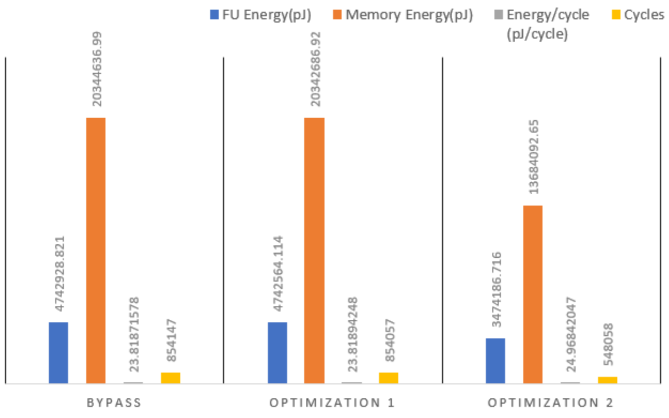


Fig. 6. Optimization 2

IV. Optimization of the last two for loops

The last two for loops involves ABU, RF, imm, ALU and MUL functional units. It is part of the of the while loop and hence, another dominating part of the code. In the naive implementation the two for loops run for 10 and 11 iterations. However, there is no need for performing all the iterations because certain values of a[i] and b[i] are zero. We know that performing a multiplication of any number with zero results with an output of zero and addition with zero does not change the output value. Hence, the computations involving zero can be skipped. This optimization method implements jumping of for loops over a stride with 8, instead of 4. Address increments of 8 skips the zero a[i] and b[i] values, thereby reducing the loop iterations to 6 for 1st for loop and 5 for 2nd for loop.

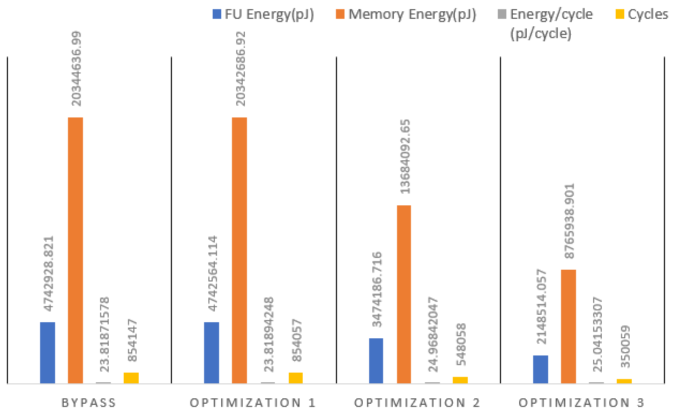


Fig. 7. Optimization 3

By using this technique, there is an decrease in energy consumption by functional units is 38.15% and by memory units is 35.94% compared with optimization 2.

V. Conclusion

In this assignment, optimizations were performed which involved implicit load/store, loop unrolling and loop jumping. Together with all the above optimizations there is an overall average improvement of 58%. Overall decrease in energy consumption by functional units is 56.13%, 58.73% by memory units and 60.94% decrease of active cycles when compared to naive implementation. No architecture improvements were except the bypass of MUL to ALU which was explained in the exercise of the assignment.

VI. Acknowledgements

For this assignment, I collaborated with Elena Galetti. However, I would also like to thank Alexandru Neculai, Cody Arets and Shuaib Ichaairne who helped me during the assignment for ideas and approach methods. Without their contribution, this assignment would have been incomplete.