Optimisation of Disparity Algorithm on Texas Instruments C6678 DSP

Elliott Worsey
Computer Science and Electronics
Department of Electrical and Electronics Engineering
ew15847@bristol.ac.uk
Candidate Number: 27455

Abdu Elturki
Computer Science and Electronics
Department of Electrical and Electronics Engineering
ae15920@bristol.ac.uk
Candidate Number: 27801

I. Introduction

This report discusses the optimisation of disparity algorithm written in C and linear assembly for Texas Instruments C6678 DSP. The compiler version used for this assignment is `TI v8.2.5` with XDC Tools version `3.51.1.18_core` and PDK version `2.0.12`. After reviewing all implementations code lines, they are then examined and compared to each other in terms of run time.

II. C Code Optimisation

A. Original Code Analysis

In the original C code, there are many lines that show why this code is not optimised. For example, the original program has five level nested loops and does not run in parallel. Furthermore, there are no SIMD operations being utilised that can reduce the number of loops used. The Branch statements placements in this code are causing either blocking in the DSP core pipeline or race condition if parallelism is attempted. Therefore, this code can be optimised greatly and perform this algorithm at much lower cycles by reducing the number of nest loop, using SIMD instructions, and preventing blocking from branch statements.

B. Loop Reduction and SIMD

a) Loading Data as Vectors: In the original code, the 2 most inner nested loop with indices `ii` and `jj` calculates the absolute difference of left and right windows. This approach is slow since it produces this difference by subtracting element by element instead of using multiple elements in parallel. The parallelism can be achieved using **"single instruction, multiple data (SIMD)"** and calculating the absolute difference by subtracting row by row of both windows. First, we defined 2 buffers to load current windows of left and right images, this is so it can let us remove the 2 most inner for loops. These buffers are arrays of size 5 and type of `long long`, where they load each window row using _`mem8_const` intrinsic, an example of the left buffer loading is illustrated in the following code lines.

```
left_buf[0] = _mem8_const(&L[(i-2)*Width+j-Radius
    ]) & mask;
left_buf[1] = _mem8_const(&L[(i-1)*Width+j-Radius
    ]) & mask;
left_buf[2] = _mem8_const(&L[(i)*Width+j-Radius])
    & mask;
left_buf[3] = _mem8_const(&L[(i+1)*Width+j-Radius
    ]) & mask;
left_buf[4] = _mem8_const(&L[(i+2)*Width+j-Radius
    ]) & mask;
```

Code 1. Loading Left Window Row into a Buffer

Since _`mem8_const` loads 8 bytes of the start of the window's rows and we are only interested in the least significant bytes, we perform the and operation of the window row and a mask that has a value of 0xFFFFFFFFFF to obtain those 5 bytes of the window, an example of this is illustrated in figure 1.

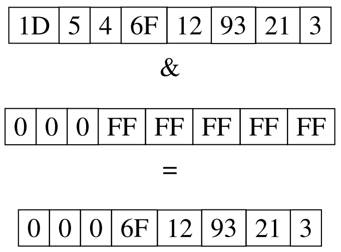

Fig. 1. Example of Obtaining 5 Least Significant Bytes

This part of the optimisation can be implemented with a for loop that iterates from 0 to 4 and optimised with `#pragma MUST_ITERATE(5,,5)` and `#pragma UNROLL(5)` instead of manually unrolling like above. However, manual unrolling gives a slight speed advantage as it removes instructions such as comparison and jump that is introduced by the for loop, which is noticeable for small loops, hence loading into the buffer is done manually.

b) Calculating The Absolute Difference: The calculation of the absolute difference is calculated by the intrinsic `_subabs4` and `_dotpu4`, and since these intrinsic operate only on pairs of packed 8-bit numbers the use of `_hill` and `_loll` is required.

```
55 Sum = _dotpu4(_subabs4(_hill(left_buf[0]),_hill(
       right_buf[0])),ones);
56 Sum += _dotpu4(_subabs4(_loll(left_buf[0]),_loll(
       right_buf[0])),ones);
57 Sum += _dotpu4(_subabs4(_hill(left_buf[1]),_hill(
       right_buf[1])),ones);
                                         ⋮
65 Sum += _dotpu4(_subabs4(_loll(left_buf[4]),_loll(
       right_buf[4])),ones);
```

Code 2. Calculating the Absolute Difference

the result of `_subabs4` in this case is a 4 byte `int` where each byte is the absolute difference of the corresponding byte from each window. These bytes gets added up together using dot product intrinsic `_dotpu4` with a value 0x01010101 resulting in the absolute difference from left and right window row.

$$\boxed{12}\boxed{93}\boxed{21}\boxed{3}$$

$$\cdot$$

$$\boxed{01}\boxed{01}\boxed{01}\boxed{01}$$

$$=$$

$$12 + 93 + 21 + 3$$

Fig. 2. Example of Adding Values of 4 Bytes Together

C. Branch Optimisation

If statements can affect the speed performance of a code negatively, so if the condition of the branch is difficult for the speculative execution unit to predict, it causes the DSP to perform additional cycles to correct itself. Thus, the overall cycle count can be reduced when the condition of the If the statement is easier to for the speculative execution unit to predict. Furthermore, mispredictions are inevitable to occur, and so it is best to reduce the number of if statements in a code to improve speculation accuracy of the DSP. For example, in the original non-optimised `stereo_vision_c.c`, the if statement in line 22 causes the speculative execution unit to mispredict often. Since the are many instances where the value of `j-Radius-k>=0` would be consecutively gives `True` at the beginning of the loop, and then suddenly `False` for the remainder of it. Not only this would make it difficult for the branch predictor to speculate, the loop iterates even when $k >$ Radius $- j$ causing unnecessary clock cyles of evaluating the condition of the branch.

```
19 for (k=0;k<Search_Range;k++)
20 {
21     Sum=0;
22     if (j-Radius-k>=0)
23     {
```

Code 3. Original Branching

We can improve this by calculating the necessary number of iterations for the loop with the index `k`, which is given as:

```
int search = Search_Range;
if (j-Radius-Search_Range < 0)
{
    search = j-Radius+1
}
for (k=0;k<search;k++)
{
    Sum=0;
```

Code 4. Non-looping Branch Statement

While this gives much faster results, branch instructions cannot run in parallel or pipeline with other instructions. We can eliminate the previous if statement as following:

```
39 flag = (j-Radius-Search_Range < 0);
40 int search = !flag * Search_Range + flag * (j-
       Radius+1);
```

Code 5. Optimised Non-looping Branch Statement

Now the value `search` can be calculated while the DSP is processing the previous code lines. Similarly, this can be done to line 34 of the non-optimised code, where it can be transformed as follows:

```
flag = (Sum<Minimize);
Distance = !flag * Distance + flag * s;
Minimize = !flag * Minimize + flag * Sum;
```

Code 6. Optimised Branch Statement used for `Distance` and `Minimize`

D. Race Condition

Implementing the previous optimisation would cause in inaccurate output result of `stereo_vision_c.c`. This is due to a race condition occuring when evaluating `Sum<Minimize` and `Sum` in parallel causing to incorrect assignment of values `Distance` and `Minimize` that affect the final disparity map. An obvious solution is to add `NOP` using `__asm__("nop")`, but this would increase the overall clock cycle. Another solution which we used, is updating `Minimize` and `Distance` in the following iteration like so:

```
49     right_buf[4] = _mem8_const(&R[(i+2)*Width+j-
           Radius-s]) & mask;
50
51     flag = (Sum<Minimize);
52     Distance = ((!(flag)) * Distance) + (flag * (s
           -1));
53     Minimize = ((!(flag)) * Minimize) + (flag *
           Sum);
54
55     Sum = _dotpu4(_subabs4(_hill(left_buf[0]),
           _hill(right_buf[0])),ones);
                                         ⋮
65     Sum += _dotpu4(_subabs4(_loll(left_buf[4]),
           _loll(right_buf[4])),ones);
66
67 }
68 flag = (Sum<Minimize);
69 Distance = ((!(flag)) * Distance) + (flag * (
       search-1));
70 Minimize = ((!(flag)) * Minimize) + (flag * Sum);
```

Code 7. Race Condition Elimination

This eliminates the race condition and also ensures it runs in parallel with surrounding code lines.

E. Pragma Optimisation

We have added `#pragma MUST_ITERATE` to the three loops that are used in our code, so the compiler is now aware of the minimum number of iterations of the loops. Those pragma also informs the compiler that the loop will always iterate in a factor of 2 and the loop can be unrolled twice. This is so the DSP units of sides A and B are utilised equally, and the and the compiled build does not favour one side more than the other.

```
#pragma MUST_ITERATE(Height-5,,2)
for (i=(Height-1)-Radius;i>=Radius;i--)
{
    #pragma MUST_ITERATE(Width-5,,2)
    for (j=(Width-1)-Radius;j>=Radius;j--)
    {
                            ⋮
            #pragma MUST_ITERATE(2,,2)
            for (s=0; s<search; s++)
{
```

Code 8. `#pragma MUST_ITERATE` Used in Optimised Code

Increasing the factor to any other even number does not lead to increased speed up since that always leads to the compiler splitting the instructions evenly to DSP core units. Hence why the optimised the code only uses a factor of 2 for those pragma.

F. Alignment and Loop Carried-Path

The function `stereo_vision_c` have independent arguments, but the compiler will always believe that there is loop carried-path in any function arguments, which is preventing pipelining in this case. The compiler can be informed that loop carried-path does not exist by adding restrict to arguments `L`, `R` and `Disparity_Map`.

```
void stereo_vision_c(unsigned char * restrict L,
    unsigned char * restrict R, unsigned char *
    restrict Disparity_Map, int Search_Range, int
    Radius)
{
```

Code 9. Loop Carried-Path Elimination

To improve loading performance, we inform the compiler that pointers `L`, `R` and `Disparity_Map` are aligned by 128 bytes at `main.c`. This is done using the `_nassert` intrinsic, which is used in the optimised code as following:

```
_nassert(((int)L & 0x7F) == 0);
_nassert(((int)R & 0x7F) == 0);
_nassert(((int)Disparity_Map & 0x7F) == 0);
```

Code 10. Informing The Compiler of Data Alignment

III. Serial Assembly Implementation

A. An Overview of The Implementation

The implementation of this algorithm in serial assembly is very similar to the previous C optimisation, where it contains three level nested loops.

a) Outer Loop: The outermost loop uses the label `loop_height` with the index i, where it iterates starting from height length - radius down to 0. This loop is mainly used for loading the (left) window from the left image, which is explained in III-B.

b) Middle Loop: The middle loop uses the label `loop_width`, and its index is j that iterates from width length - radius down to 0. At the end of each iteration, the result elements of the disparity map are stored as described in III-D.

c) Most Inner Loop: The most inner loop uses the label `loop_search`, and its index s iterates from 0 up to the required search range to calculate the absolute differences and find the elements of the disparity map, that are described in III-C and III-D.

B. Data Loading

The strategy of this implementation is to load each row of both windows in their own pairs of high and low registers, so it can store the 5 bytes of information. The instruction `ldndw` was used since we are loading data in a non-aligned way.

```
ldndw   *L(offset),L1h:L1l

add offset,width,offset
ldndw   *L(offset),L2h:L2l

add offset,width,offset
ldndw   *L(offset),L3h:L3l

add offset,width,offset
ldndw   *L(offset),L4h:L4l

add offset,width,offset
ldndw *L(offset),L5h:L5l
```

Code 11. Loading The Left Window Into Registers

The starting offset is calculated using a simple equations that is same one used in the C code, which is given by:

$$\text{offset} = (i - \text{Radius}) \times \text{Width} + j - \text{Radius} \quad (1)$$

The index i is used for the outermost loop of the algorithm, while index j is used in for the first inner loop. After loading the entire row, the length of width is added to the offset in order to load the following row(s). Obviously, we are only interested in the last 5 bytes of the loaded data[1] and the first 3 bytes can be omitted. To achieve this, the **and** operator and is applied on high registers and value `0x000000FF` which is stored in register named `mask`. This part can be done in parallel since there are no dependencies.

```
        ;mask out the unwanted data
        and mask,L1h,L1h
||      and mask,L2h,L2h
||      and mask,L3h,L3h
||      and mask,L4h,L4h
||      and mask,L5h,L5h
```

Code 12. Removing The Unwanted Bytes of The Left Window

[1]The reason we are interested in the last 5 bytes and not in the first 5 is because the data are loaded in little-endian sequential order.

This loading process is done similarly for the right window, which is illustrated is below:

```
ldndw   *R(offset),R1h:R1l
and mask,R1h,R1h
add offset,width,offset
ldndw   *R(offset),R2h:R2l
and mask,R2h,R2h
add offset,width,offset
ldndw   *R(offset),R3h:R3l
and mask,R3h,R3h
add offset,width,offset
ldndw   *R(offset),R4h:R4l
and mask,R4h,R4h
add offset,width,offset
ldndw   *R(offset),R5h:R5l
and mask,R5h,R5h
```

Code 13. Loading The Right Window Into Registers

The beginning the offset value where the k is an index of the third inner loop is given by:

$$\text{offset} = (i - \text{Radius}) \times \text{Width} + j - \text{Radius} - s \qquad (2)$$

C. Calculating The Absolute Differences

The calculation of the absolute difference is done mostly using `subabs4` instruction, which works the same way as `_subabs4` intrinsic. The packed 4 bytes created by `subabs4` can be added together using `dotpu4` instruction with the value `0x01010101` stored in register named `ones`, this is the same method described in II-B0a. To achieve parallelism, the results of the higher registers is stored in `parthigher` register then added to `sumhigher`, while the `partlower` stores the lower registers results which are added to `sumlower`.

```
    subabs4 R1h,L1h,parthigher
||  subabs4 R1l,L1l,partlower

    dotpu4 parthigher,ones,parthigher
||  dotpu4 partlower,ones,partlower

    add 0,parthigher,sumhigher
||  add 0,partlower,sumlower
                             ⋮
    dotpu4 parthigher,ones,parthigher
||  dotpu4 partlower,ones,partlower

    add sumhigher,parthigher,sumhigher
||  add sumlower,partlower,sumlower

    add sumhigher,sumlower,sum
```

Code 14. Loading The Right Window Into Registers

D. Storing The Disparity Map

To find the elements of the disparity map it would require finding the lowest sums that are calculated in the `loop_search`. To implement this, the lowest sum is stored in `bestsum` register and its corresponding distance is stored in `dist` register. At the end of the loop, the value stored in `sum` is compared to `bestsum` and if it is less than it, then its value is moved to `bestsum` with its distance/loop iteration.

```
        ;see if the new sum is lower
        cmplt sum,bestsum,check
        ;store the current best sum distance
[check] mv sum,bestsum
[check] mv s,dist
```

Code 15. Loading The Right Window Into Registers

At the end of every iteration of `loop_width`, the value in register `dist` is stored at memory location of `a_3 +` ($i\times$ width).

```
;update the disparity map with the best distance
mpy i,width,offset
add offset,j,offset
stb dist,*a_3[offset]
```

Code 16. Loading The Right Window Into Registers

IV. Results and Discussion

TABLE I
Run Time of The Algorithm

Method	Time in Seconds
Non-optimised C	0.58
Optimised C	0.0278
Optimised Linear Assembly	0.0374

TABLE II
Optimised C Software Pipeline Information

	A-Side	B-Side
.L units	6	6
.S units	0	1
.D units	3	2
.M units	7	7

TABLE III
Optimised Linear Assembly Software Pipeline Information

	A-Side	B-Side
.L units	6	5
.S units	0	0
.D units	0	5
.M units	6	10

As shown in Table I, the run time of the optimised C is 20 times faster than non-optimised code while the optimised linear assembly is 15 times faster. While we did achieve faster than required timing (0.04 seconds) for the optimised codes, the optimised C is faster than the linear assembly code, and this is because of the balancing of the DSP units. The linear assembly code (Table III) have poor balance in `.D` and `.M` units compared to optimised C code (Table II). The balance of the optimised C is achieved by using `#pragma MUST_ITERATE` at line `24`, where if implementing this in a linear assembly would require writing many lines to achieve this balance.

V. Conclusion

We were able to optimise the existing C code for disparity algorithm and make it 20 times faster by having less loops and using SIMD operations. Linear assembly implementation achieved faster run time than non-optimised C code, but it was difficult to balance the DSP core unit usage without the aid of the compiler.