Disparity Algorithm Implementation on the TMS320C6678 DSP

Naimul Hoque
nh15775@my.bristol.ac.uk

Goce Dimitrov
gd14470@my.bristol.ac.uk

I. Introduction

Stereo vision relies on using two separate cameras to generate depth information. These cameras are spaced apart and angled such that they mimic how the eyes perceive depth. This difference in spacing and angle creates what is called *ocular disparity*. To determine this disparity then, the following steps are required:

1. For each epipolar line
2. For each pixel in the left image
   2.1. Compare with every pixel on same epipolar line in right image
   2.2. Pick pixel with minimum match cost

The steps outlined above are used to perform the very basic disparity calculation. There are many more advancements and improvements to this. One is called block-matching or window-matching. In this method a small region of pixels is taken from the right image, and the closest matching region of pixels is searched for in the left image. Some of those similarity metrics that can be used for finding the closest matching block are:

1) Sum of absolute differences (SAD)
2) Zero-mean sum of absolute differences (ZSAD)
3) Sum of squared differences (SSD)
4) Zero-mean sum of squared differences (ZSSD)
5) Normalised cross correlation (NCC)

There are two important factors to consider in a disparity calculation - what similarity measure will be used between the left and right frames, and how is memory accessed between the two frames.

II. DSP Details

For this project the disparity calculation was implemented on a high-performance fixed/floating-point Digital Signal Processor that is based on the TI's KeyStone multicore architecture. Namely, the TMS320C6678 Digital Signal Processor (DSP) from Texas Instruments was used.

The C6678 is a 32-bit architecture, comprising of 64 32-bit General Purpose Registers split between two register files, A and B.

III. Disparity Calculation in C

A. Overview

The disparity calculation is performed using the SAD similarity measure. The default implementation provided relies on using five nested loops to carry out the calculation.

```
#define Height 223
#define Width 280

void stereo_vision_c(unsigned char *L, unsigned char *R,
                     unsigned char *restrict
                       ↪  Disparity_Map,
                     int Search_Range, int Radius)
{
    int i, j, k;
    int ii, jj;
    int Sum;
    for (i = (Height - 1) - Radius; i >= 0 + Radius; i--)
    {
        for (j = (Width - 1) - Radius; j >= 0 + Radius;
          ↪  j--)
        {
            int Distance = 0;
            int Minimize = 100000;
            for (k = 0; k < Search_Range; k++)
            {
                Sum = 0;
                if (j - Radius - k >= 0)
                {
                    for (ii = -Radius; ii <= +Radius;
                      ↪  ii++)
                    {
                        for (jj = -Radius; jj <= +Radius;
                          ↪  jj++)
                        {
                            Sum += abs(L[(i + ii) * Width
                              ↪  + (j + jj)]
                                - R[(i + ii) * Width + (j
                                  ↪  - k + jj)]);
                        }
                    }
                    if (Sum < Minimize)
                    {
                        Minimize = Sum;
                        Distance = k;
                    }
                }
            }
            Disparity_Map[i * Width + j] = Distance;
        }
    }
}
```

Source Code 1: Original C code for Disparity calculation

Having all these loops is not optimal. This is due to the lack of exploiting instruction-level parallelism or more widely known as Single instruction, multiple data (SIMD), as well as constantly recalculating certain values. Beyond this though, there also exist conditions within the loops, which makes pipelining impossible due to the compiler being unable to determine exactly when a branch will occur during all iterations.

B. Optimisation

The optimisation performed was split into two stages:

- Optimising loop usage to exploit instruction parallelism.
- Eliminating conditions to enable software pipelining.

To optimise loop usage, the first thing to note is how the windowing procedure works, and then taking advantage of the C6000 intrinsics to handle the windows better.

The size of a window is 5x5 pixels. The current method accesses one pixel at a time which is not optimal. Instead, loading all 5 pixels per row would improve performance significantly by not having to perform unnecessary loads. There is no clean way to load 5 pixels at the same time as each pixel is a byte, and since the C6678 DSP uses 32-bit registers, the minimum number of registers to store 5 pixels is 2.

The initial goal was to use #PRAGMA UNROLL(n) to automatically unroll the loops, believing that this would improve performance by enabling the compiler to detect instruction-level parallelism. Sadly this was not the case as performance did not improve whatsoever in doing this.

Thankfully, manually unrolling the loops was simple thanks to the usage of the _mem8 intrinsic, which allowed loading the 8 bytes into a 64-bit value. Performing bit-wise AND over this with a particular constant removes the extra 3 bytes at the end, which allowed us to perform the necessary calculations. With the help of additional intrinsics, namely the _hill and _loll intrinsics were used to split the high and low bytes from the registers. Which then enabled us to use the _subabs4 intrinsic which processes four 8-bit pixels and takes the absolute difference between the input data and reference values. The next step was to use the _dotpu4 intrinsic. The masks in the _dotpu4 operations are preloaded with each byte value containing the value +1. Therefore, multiplying each byte value with one and adding them together allows us to sum the four values in the 32-bit register.

```
for (ii=-Radius;ii<=+Radius;ii++)
{
    for (jj=-Radius;jj<=+Radius;jj++)
    {
        Sum += abs(L[(i+ii)*Width+(j+jj)]
              -R[(i+ii)*Width+(j-k+jj)]);
    }
}
```

Source Code 2: Original calculation

```
// At the start of the file there is a typedef
#typedef const long long cll

cll lb0=_mem8_const(&L[(i)*Width+(j)]) & 0xFFFFFFFFFF;
cll lb1=_mem8_const(&L[(i+1)*Width+(j)]) & 0xFFFFFFFFFF;
cll lb2=_mem8_const(&L[(i+2)*Width+(j)]) & 0xFFFFFFFFFF;
cll lb3=_mem8_const(&L[(i+3)*Width+(j)]) & 0xFFFFFFFFFF;
cll lb4=_mem8_const(&L[(i+4)*Width+(j)]) & 0xFFFFFFFFFF;

cll rb0=_mem8_const(&R[(i)*Width+(j-k)]) & 0xFFFFFFFFFF;
cll rb1=_mem8_const(&R[(i+1)*Width+(j-k)]) & 0xFFFFFFFFFF;
cll rb2=_mem8_const(&R[(i+2)*Width+(j-k)]) & 0xFFFFFFFFFF;
cll rb3=_mem8_const(&R[(i+3)*Width+(j-k)]) & 0xFFFFFFFFFF;
cll rb4=_mem8_const(&R[(i+4)*Width+(j-k)]) & 0xFFFFFFFFFF;

Sum = _abs(_hill(lb0) - _hill(rb0)) +
      _dotpu4(_subabs4(_loll(lb0),
              _loll(rb0)), 0x01010101);
Sum += _abs(_hill(lb1) - _hill(rb1)) +
       _dotpu4(_subabs4(_loll(lb1),
               _loll(rb1)), 0x01010101);
Sum += _abs(_hill(lb2) - _hill(rb2)) +
       _dotpu4(_subabs4(_loll(lb2),
               _loll(rb2)), 0x01010101);
Sum += _abs(_hill(lb3) - _hill(rb3)) +
       _dotpu4(_subabs4(_loll(lb3),
               _loll(rb3)), 0x01010101);
Sum += _abs(_hill(lb4) - _hill(rb4)) +
       _dotpu4(_subabs4(_loll(lb4),
               _loll(rb4)), 0x01010101);
```

Source Code 3: Manually unrolled calculation

In dealing with the loop conditions, converting them from if statements to ternary operators (x = c ? y : z) removes branching from the code and thus enabling proper pipelining and reducing the time necessary to do the disparity calculation by a lot.

```
if (j-Radius-k>=0)
{
/*
 * Original loop code here...
 */
    if (Sum<Minimize)
    {
        Minimize=Sum;
        Distance=k;
    }
}
```

Source Code 4: Original conditional code

```
/*
 * Manually unrolled code here...
 */
int sumCheck = !(j < k) && (Sum < Minimize);
Minimize = (sumCheck) ? Sum : Minimize;
Distance = (sumCheck) ? k : Distance;
```

Source Code 5: Branchless conditional code

Both the optimised and non-optimised version of that particular part of the code can be seen in Source Codes 2 and 3. The full source code is present in the zip file under the name “stereo_vision_c.c”. This code was run using the -O3 optimisation compiler flag.

C. Results

Both optimisations yielded in a high speedup. The first optimisation, which was to remove the two innermost loops and utilise SIMD instructions more than halved the time it takes for calculating the disparity and the second optimisation that pipelined those SIMD instructions resulted in a speedup of 21x. The results have been outlined in table I.

Type	Time/s	Speedup Factor
Original	0.58s	N/A
Unrolled	0.20s	1.9
Unrolled w/Zero Branching	0.0277s	19.94

TABLE I: Results for different optimisation strategies

As it can be seen in Table I, manually unrolling the summation loop and utilising SIMD instructions provided a large boost to performance, though not as significant as removing the conditions of the code. Whilst SIMD is beneficial, pipelining is a far greater provider for performance as it allows instructions to flow as close to continuously as possible.

```
;*   SOFTWARE PIPELINE INFORMATION
;*                                  A-side    B-side
;*.L units                            6         6
;*.S units                            0         0
;*.D units                            5         0
;*.M units                            0         5
;*.X cross paths                      0         0
;*.T address paths                    5         5
;*Logical  ops (.LS)                  0         5   (.L or .S unit)
;*Addition ops (.LSD)                 9        10   (.L .S .D unit)
;*Bound(.L .S .LS)                    3         6
;*Bound(.L .S .D .LS .LSD)            7*        7*
;*
;*      Searching for software pipeline schedule at ...
;*         ii = 7  Schedule found with 4 iterations in
 ↪  parallel
```

Source Code 6: Software pipeline information from optimised C code.

IV. Disparity Calculation in Linear Assembly

A. Overview

Linear assembly is a simplified form of standard C6000 assembly. Simplified in the sense that:

- Instructions do not need to be explicitly designated as parallelised
- Functional units do not need to be stated explicitly.
- NOPs (no operations) don't need to appear.

This makes writing assembly code much easier for the programmer as they can focus purely on the flow of the program without **initially** worrying about timings and what can be parallelised and how best to use the units.

With no basis provided, the first goal was to write out the disparity calculation from scratch in linear assembly. Doing this was simply a matter of translating our optimised C code with all its intrinsics into its assembly equivalent.

This resulted in a speed of 0.0497 seconds for the disparity calculation in linear assembly, which was not less than its C counterpart and was an indication that the C compiler has been doing more optimisations to the code and that we need to do the same as well if we wanted to reach closer speeds between the two.

B. Optimisation

At this stage, re-introducing the functional units and parallel specifications is important as this is where the optimisations occur. The compiler will do a good job with what has been provided to it, but one major issue that occurred was utilisation of resources. Specifically, how unbalanced the utilisation was between the A and B sides. The issue is that if one side carries a greater load than the other, the whole program has to match the speed of the slowest side.

The main issue stemmed from the D units, which are responsible for loading and storing data. The initial code resulted in all used D units appearing on the A side, which was not ideal as it affected the pipeline iteration count.

To balance this out, the load instructions had the D units specified.

```
; Line 34-57
ADD i, 1, offsetL2
ADD i, 2, offsetL3
ADD i, 3, offsetL4
ADD i, 4, offsetL5
MPY i, width, offsetL1
MPY width, offsetL2, offsetL2
MPY width, offsetL3, offsetL3
MPY width, offsetL4, offsetL4
MPY width, offsetL5, offsetL5
ADD j, offsetL1, offsetL1
ADD j, offsetL2, offsetL2
ADD j, offsetL3, offsetL3
ADD j, offsetL4, offsetL4
ADD j, offsetL5, offsetL5

LDNDW .D1 *a_1(offsetL1), w1:w2
LDNDW .D1 *a_1(offsetL2), w3:w4
LDNDW .D1 *a_1(offsetL3), w5:w6
LDNDW .D1 *a_1(offsetL4), w7:w8
LDNDW .D1 *a_1(offsetL5), w9:w10
; and_one := 0xFF
AND w1, and_one, w1
AND w3, and_one, w3
AND w5, and_one, w5
AND w7, and_one, w7
AND w9, and_one, w9
```

Source Code 7: Data loading snippet from linear assembly

Each offset is stored in a separate register rather than reusing one register. This is because the right side data loads uses the left sides offsets, shifted by the loop variable k to represent the window shifting leftwards. The bitwise AND applies only to one register per load, as those words represent the upper bytes of the 8-byte load.

The right side loads didn't need D units specified, as the compiler had optimised and balanced usage automatically.

Another issue was that we had two conditional comparisons and the compiler was having difficulties assigning proper scheduling because of them. This lead to the idea that since both conditions needed to be true, one comparison can be avoided when the other one is false, which indeed sped up our code. The snippet of that particular part of the code can be seen below.

```
; Line 113-117
                 CMPLT j, k, radius_check;
[!radius_check] CMPLT sum, minimize, sum_check
[radius_check]  ZERO sum_check
[sum_check]     MV sum, minimize
[sum_check]     MV k, distance
```

Source Code 8: Modified conditions

C. Results

The initial speed we got from the linear assembly implementation was not as expected. The expectation was that the linear assembly would achieve equal or near-equal times to the C code, but the reality was the linear assembly was around “2x” slower. This is because the C compiler was performing more aggressive optimisations over the assembly compiler. Of note was the pipelining iteration interval, as this was lower on the C side with ii = 7, whereas the assembly side produced ii=10.

```
;*SOFTWARE PIPELINE INFORMATION
;*
;*Resource Partition:
;*                              A-side   B-side
;*.L units                        6        9
;*.S units                        0        0
;*.D units                        3        2
;*.M units                        4        1
;*.X cross paths                  0        0
;*.T address paths                5        5
;*Logical  ops (.LS)              0        0 (.L or .S unit)
;*Addition ops (.LSD)            19       14 (.L .S .D unit)
;*Bound(.L .S .LS)                3        5
;*Bound(.L .S .D .LS .LSD)       10*       9

;*Searching for software pipeline schedule at ...
;*ii = 10 Schedule found with 3 iterations in parallel
```

Source Code 9: Software pipeline information from optimised assembly code.

Upon optimising D unit usage, times achieved were below the 0.04s threshold, but still higher than the best C code time as seen back in Table I.

Type	Time/s
Initial Assembly	0.0497
Assembly w/ Specified Functional Units	0.0387

TABLE II: Results showing progression of optimisation strategies

The full source code is present in the zip file under the name “stereo_vision_sa.sa”. This code was run using the -O3 optimisation compiler flag.

V. Conclusion

Optimising for the DSP relies on knowing what the hardware has, what it’s capable of and making extensive and effective usage of it. Both the C and Linear Assembly achieved times below the 0.04s threshold, but the Linear Assembly was found to be slower than the C code.

It is clear that a lot more optimisations can be done to the assembly code, but due to time constraints and the fact that for both versions the previously mentioned threshold was reached, further optimisations were not performed.

A final point to be made is that conditionals still produce the greatest bottleneck to performance. Removing those would yield far better results. Tentatively, times below 0.01s could be produced if all conditionals were removed from the critical loop, but in the end those conditions had to stay to prevent calculation errors. Given more time though, it is possible to remove those calculation errors and achieve the aforementioned timing.

Method	Time/s
Non-optimised C	0.58
Optimised C	0.0277
Optimised Linear Assembly	0.0387

TABLE III: Final results