SENG440 Embedded Systems

– Lesson 107: Motion Estimation –

Mihai SIMA

msima@ece.uvic.ca

Academic Course

Copyright © 2019 Mihai SIMA

Disclaimer

The purpose of this course is to present general techniques and concepts for the analysis, design, and utilization of embedded systems. The requirements of any real embedded system can be intimately connected with the environment in which the embedded system is deployed. The presented design examples should not be used as the full design for any real embedded system.

Lesson 107: Motion Estimation

Introduction

- In MPEG a picture can be of one of three types: **I**, **P**, and **B**
  - **I** (intra-coded) pictures are encoded independently
  - **P** (predictive-coded) pictures are encoded using motion predictions from past **I**- or **P**-pictures
  - **B** (bidirectionally predictive-coded) pictures are encoded using motion predictions from the nearest past and / or future **I**- or **P**-pictures
- Motion prediction exploits the fact that pictures are very similar except for changes generated by moving objects
- **Motion estimation** = the process of determining the motion vectors of different areas in the frame being encoded
- Motion estimation is very computationally demanding, so it is the bottleneck in MPEG video encoding

Motion estimation theory

- Similarities between video frames are exploited to achieve high compression rate
- Instead to code (and thus transmit) a new frame, code only the relative movement of the current frame with respect to the previous one
- **Motion estimation** algorithm captures such movement by finding the **best match** of an $N \times N$ block in a reference frame
- Commonly used metric – **Sum-of-Absolute-Differences** (SAD)
- Motion estimation is performed typically on a block of pixels
- SAD operation is usually considered for 16×16-pixel blocks
- The search area could involve a large number of blocks
- SAD operation can be time consuming

Motion estimation theory

- Motion estimation is performed on a set of pixels
- Each frame is divided into blocks of equal size
- For each block in the current frame a search is performed in the reference frame to find the block resembling the current block the most
- The search is limited to a rather small area
    - A search performed over the whole reference frame for each block in the current frame is computationally intensive
    - Movements in video sequences are usually small
- After finding the best match for the current block in the current frame, two elements are stored:
    - A motion vector (displacement relative to the current block)
    - Difference between the two blocks

Motion estimation theory

- (x, y) is the position of the current block
- (r, s) is the motion vector (the displacement of the current block A relative to the reference block B)
- The computation per each block pair

$$SAD(x, y, r, s) = \sum_{i=0}^{15} \sum_{j=0}^{15} |A(x+i, y+j) - B((x+r)+i, (y+s)+j)|$$

- A very large number of block pairs per frame are analyzed!
  - Motion estimation is the bottleneck in video encoding

Motion estimation – software solution I

- (x, y) is the position of the current block
- (r, s) is the motion vector (the displacement of the current block A relative to the reference block B)
- Lines 07-09: 1 comparison, 1 branch, 0.5 subtraction, 1 addition

```
01   int A[16][16], B[16][16], diff, sad = 0;
02   int i, j;
03
04   for( i=0; i<16; i++)
05   for( j=0; j<16; j++) {
06     diff = A[x+i][y+j] - B[(x+r)+i][(y+s)+j];
07     if( diff < 0)      /* takes the absolute value */
08       diff -= diff;
09     sad += diff;
10   }
```

Motion estimation – software solution II

- The explicit emulation of the absolute operation requires 3.5 operations
- The following code requires only 3 operations
- Penalty: the code size is 1 instruction larger (good trade-off)

```
int A[16][16], B[16][16], diff, sad = 0;
int i, j;

for( i=0; i<16; i++)
for( j=0; j<16; j++) {
    diff = A[x+i][y+j] - B[(x+r)+i][(y+s)+j];
    if( diff < 0)
        sad -= diff;
    else
        sad += diff;
}
```

Motion estimation – software solution III

- Operation count per 16 $\times$ 16 block
  - 256 subtractions
  - 256 if-then-else to calculate the absolute value
  - 256 additions/subtractions
  - 256 comparisons
  - 256 incrementations
  - 256 branch operations
- Large number of operations per block pair
- Large number of true dependencies – the code is sequential
  - Would software pipelining, loop unrolling, etc. help?
- **Hardware** support is needed
- The code is sequential $\longrightarrow$ firmware is not likely to provide improvement

Improving the software solution – loop unrolling I

```
int A[16][16], B[16][16], diff1, diff2, sad = 0;
int i, j;

for( i=0; i<16; i++)
for( j=0; j<16; j+=2) {
   diff1 = A[x+i][y+j] – B[(x+r)+i][(y+s)+j];
   diff2 = A[x+i][y+j+1] – B[(x+r)+i][(y+s)+j+1];
   if( diff1 < 0)
     sad –= diff1;
   else
     sad += diff1;
   if( diff2 < 0)
     sad –= diff2;
   else
     sad += diff2;
}
```

Improving the software solution – loop unrolling II

- Generally it is not possible to execute two branch operations in parallel
- Do guarded operations help executing two branch operations in parallel?
- Parallelism: Lines 07 and 08
- Load and store operations can be overlapped to some extend
- The ’for’ loops generate only $16 \times 8 = 128$ branch operations
  - This is half of the initial branch count
  - Penalty: double code size that might induce instruction cache misses
- Conclusion: loop unrolling does not provide significant improvement
- Homework: analyze software pipelining

Motion estimation – hardware solution

- **Which operations to implement in hardware?**
  - A single absolute-difference operation and do the accumulation of the result in software?
  - A sum of 256 absolute-differences (that is, everything in hardware)?
  - A partial sum of absolute-differences and perform the rest of the accumulation in software?
- Host processor architectural constraints – ARM example:
  - Two 32-bit arguments per instruction call
  - One 32-bit result per instruction call
  - Are non-reentrant instructions allowed?
- Consider for the sake of presentation that
  - Each pixel is represented on an 8-bit signed integer
  - Four pixels fit into a 32-bit register (argument in our case)
- The hardware will calculate the SAD's for four pixel pairs

Motion estimation – reentrant or non-reentrant unit?

- The new unit is called Sum-of-Absolute-Differences (SAD)
- **Reentrant SAD**
  - The unit does not have state
  - The output depends only on the inputs
  - The function to be implemented is of zeroth order
    - SAD can be a combinational circuit
    - A higher order circuit can be used, but the function to be implemented is still of zeroth order
  - **The accumulation is done in software**
- **Non-reentrant SAD**
  - The unit has state
  - The output depends on the inputs and on the previous state
  - The function to be implemented is of order greater than zero
  - **The accumulation is done in hardware**
- How is this problem solved in a Multiply-and-ACumulate unit?

Motion estimation – hardware solution

- Sum-of-Absolute-Differences (SAD) instruction

  SAD Rt, Rs1, Rs2

  where

  - Rs1 (source register 1) contains four pixels from the current frame
  - Rs2 (source register 2) contains four pixels from the reference frame
  - Rt (target register) contains the SAD's for these four pixel pairs

- The software routine will be rewritten using the new instruction
- Four pixels are assumed to be packed into one 32-bit integer (the column index ranges $[0 \ldots 3]$)

  ```
  int A[16][4], B[16][4];
  ```

  (If the pixels are not already packed, then they must be packed before the SAD instruction is called)

Motion estimation – reentrant SAD

- The routine using the new SAD instruction is presented below
- The SAD instruction is called 64 times per 16 $\times$ 16 block
- SAD latency is needed to determine the cycle count of the code below

```
int A[16][4], B[16][4], sad = 0;
register int Rs1, Rs2, Rt, i, j;

for( i=0; i<16; i++)
for( j=0; j<4; j++) {
  Rs1 = A[i][j];
  Rs2 = B[i][j];
  __asm__( "SAD␣%1,␣%2,␣%0" : "=r" (Rt) : "r" (Rs1), "r" (Rs2));
  sad += Rt;
}
```

Motion estimation – non-reentrant SAD

- The accumulation is done inside the SAD unit
- A reset instruction for SAD needed
- The interrupts should be disabled during the 'for' loops execution

```
int A[16][4], B[16][4], sad = 0;
register int Rs1, Rs2, Rt, i, j;

__asm__( "RESET_SAD␣%1,␣%2,␣%0" : "=r" (Rt):"r" (Rs1),"r" (Rs2));
for( i=0; i<16; i++)
for( j=0; j<4; j++) {
  Rs1 = A[i][j];
  Rs2 = B[i][j];
  __asm__( "SAD␣%1,␣%2,␣%0" : "=r" (Rt):"r" (Rs1),"r" (Rs2));
}
sad = Rt;
```

Motion estimation – project requirements

- Build the testbench including
  - Two 320 $\times$ 240 images representing the current and reference pictures
- Determine the performance of the pure software solution
- Build a 4-pixel-pair SAD unit in hardware and determine its latency
- Try also a non-reentrant hardware unit and compare it against the reentrant counterpart
- Rewrite the high-level code and instantiate the new instruction
  - Use assembly inlining
- Determine the performance of the hardware-based solution
- Determine the penalty of the hardware-based solution
  - The number of gates needed to implement the SAD instruction

References

- Stamatis Vassiliadis et al., *The Sum-Absolute-Difference Motion Estimation Accelerator*, in Proceedings of the 24th Euromicro Conference, Vasteras, Sweden, August 1998, pp. 559–566.
- Dzung Tien Hoang and Jeffrey Scott Vittler, *Efficient Algorithms for MPEG Video Compression*, Wiley, 2002.
- Barry G. Haskell, Atul Puri, and Arun N. Netravali, *Digital Video: An Introduction to MPEG-2*, Springer, 2002.
- Chad Fogg, Didier J. LeGall, Joan L. Mitchell, and William B. Pennebaker, *MPEG Video Compression Standard*, Springer, 1996.

Questions, feedback

?

Notes I

Notes II

Notes III

Notes IV

Project Specification Sheet

- **Student name:** . . . .
- **Student ID:** . . . .
- **Function to be implemented:** . . . .
- **Argument range:** . . . .
- **Wordlength:** . . . .