Performance Improvement of the H.264/AVC Deblocking Filter Using SIMD Instructions

Stephen Warrington, Hassan Shojania, Subramania Sudharsanan, and Wai-Yip Chan
Department of Electrical and Computer Engineering
Queen's University, Kingston, ON, Canada
{warrings, shojania}@ee.queensu.ca
{s.sudharsanan, geoffrey.chan}@queensu.ca

***Abstract*— The H.264/AVC standard defines an in-loop deblocking filter which is used in both the encoder and decoder. This work examines several methods for improving the performance of the H.264/AVC reference software implementation of the deblocking filter. Methods examined include general software optimization, parallelization through standard multimedia SIMD instructions, and augmenting standard SIMD instruction sets with new instructions. Using the above methods, we are are able achieve a large speedup of the deblocking filter computation.**

I. Introduction

The H.264/AVC specification includes a standard integrated loop filter for deblocking in both the encoder and decoder. It is adaptive on several levels, from the slice level down to the pixel level. The deblocking filter forms a significant part of the computational complexity of the codec. In the H.264/AVC case, it is estimated that the deblocking filter forms one third of the computational complexity of the decoder [5], making it the most computationally demanding function of the decoder. Therefore, methods to improve the performance of the deblocking filter will provide significant performance benefits for H.264/AVC implementations.

The high computational requirements of the H.264/AVC deblocking filter have resulted in several dedicated hardware architectures for the operation, such as found in [2] or [3]. Dedicated hardware units for performing the deblocking operation can achieve very good performance, but more flexible alternatives to dedicated hardware are often desired. This paper examines the performance attainable while implementing the H.264/AVC deblocking filter on a programmable processor. This evaluation can be useful to software developers, to gauge the level of speedup achievable over the reference H.264/AVC software implementation. It is also useful to system-on-chip designers implementing H.264/AVC coding, to assist the decision of whether to implement the H.264/AVC deblocking filter in a dedicated hardware unit, or using a programmable processor.

The H.264/AVC reference software implementation of the adaptive deblocking filter is used as a baseline reference. This paper explores a variety of methods to improve the performance of the reference implementation. We first examine performance improvement through general software optimization. We also explore improving the performance by using standard vector single-instruction-mutliple-data (SIMD) instructions, available in current multimedia SIMD instruction sets. Lastly, we look at improvements achievable through adding new instruction extensions, specialized for the deblocking filter application.

II. Background

A. Deblocking Filters

The block-based nature of many forms of video coding produces blocking effects in the output video. H.264/AVC, a block-based video coding standard, is prone to such effects. There are a number of sources for the distortion at block edges, as described in [1]. Typical sources of blocking effects include the block-based discrete cosine transform (DCT) and the edges of blocks used in motion compensated prediction. Because of these blocking effects, it is important to use some form of deblocking filtering on the output video sequence.

There are two main methods used to implement deblocking filters. They may be implemented either as a loop filter or as a post filter, with tradeoffs inherent in either approach [1]. Loop filtering can provide better visual quality and rate distortion performance, but it incurs additional computational requirements. H.264/AVC includes a normative loop filter, which is described in more detail below.

B. H.264/AVC Loop Filter

The H.264/AVC deblocking filter is an adaptive filter. The amount of filtering that is performed along each position on the block edge depends on a number of factors. The adaptivity of the filter can be divided into three levels: slice level, block level, and pixel level.

1) Slice Level: At the slice level, the encoder may vary a threshold offset which determines an overall level of filtering to be used during encoding. This threshold is used during the pixel-level filtering decision, which is described later.

2) Block Level: Filter strength varies at the 4x4 block level. This decision is based on a set of 4x4 block-dependant factors. For each 16-pixel block edge shown in Figure 1a, there may be four distinct strength values, one for each set of four pixels.

3) Pixel Level: At the finest granularity, the filter that is applied can vary at the pixel (pel) level. A decision is made at the pixel level by evaluating the difference between pixels next to the edge. If the difference between these pixels is higher than some quantization level dependant threshold with

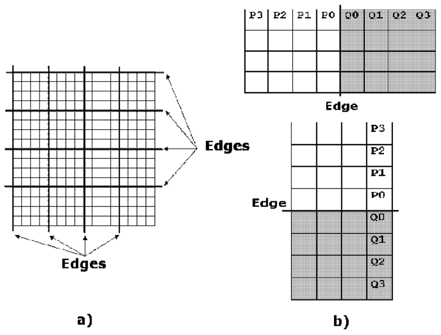

Fig. 1. a) Edges in a macroblock b) Pixel naming on either side of an edge.

encoder-defined offset, then filtering will not be performed. This avoids filtering edges which are integral to image content. Only edges due to coding artifacts should be filtered.

C. Filtering Procedure

Filtering must be done in a specific order to ensure that identical results are created by the encoder and decoder. Vertical edges are filtered first, followed by horizontal edges. The filtering inputs up to four pixel values on either side perpendicular to the edge, and can change up to three pixel values on either side perpendicular the edge. Looking at Figure 1b, pixels from P2 to Q2 may be updated, while pixels from P3 to Q3 may be read. The actual filter calculation used to generate filtered pixel values are given in [1], and are omitted here. A high level view of the steps involved in the filtering is shown below.

1) If strength for this position is zero, skip rest of the steps.
2) Load pixel values for the four pixels left/up from the edge, and the four pixels right/down from the edge (P0 to P3, and Q0 to Q3).
3) Calculate filter output values along the edge using conditional checks on strength and the loaded pixel values.
4) Store updated pixel values. Up to three different pixel values may be modified per position.

D. SIMD

Many signal processing applications can be decomposed into a set of vector operations. As a means of speeding these applications, most major processor families include some form of multimedia instruction set, e.g. MMX and SSE1 through SSE3 for Intel [4], among others. These instructions allow for sub-word parallelism by SIMD: one instruction triggers simultaneous execution on a vector of data, which can result in large application speedup. We examine the use of SIMD for the H.264/AVC deblocking filter.

A fundamental requirement of employing SIMD instructions in an application is that it is possible to arrange operations and data such that vector operations can be used. The pixel-level and block-level adaptivity of the H.264/AVC deblocking filter causes some problems here. In [1], the authors state that the conditional branching required consumes a large portion of the filtering time, and that the structure of the filter makes it ill-suited to SIMD implementation. A major impediment to SIMD execution results from the different types of filter that may be required for adjacent positions along an edge. The authors of [5] performed a complexity analysis of the H.264/AVC decoder, where they determined that the best method of implementing the decoder on a general purpose processor is to use standard scalar conditional branching, without making use of SIMD instructions. However, the examination in [5] was performed on a system with a Pentium III processor, where the extended integer SIMD found in SSE2 is not available. In our initial examination of SIMD, we use a system with a Pentium IV processor, where we can make use of the extended functionality of the SSE2 set. This paper demonstrates that it is possible to obtain significant performance gains on a general purpose processor implementation by reorganizing the implementation of the H.264/AVC deblocking filter to allow for SIMD execution.

III. Code Optimization

A. General Software Optimization

An initial step in optimizing the reference implementation of the deblocking filter was to investigate the reference coder, JM version 8.2. From our investigation, we were able to find many opportunities for general software optimization. These optimization steps included the removal of redundant operations, and the reordering of certain operations inside the code. This software optimization provided a large amount of speedup over the reference implementation.

B. SIMD Parallelization

A key modification that was made to the reference software was to alter the code to make use of SIMD operations. The original code performed the deblocking operation on a macroblock-by-macroblock basis. For each macroblock, the reference software first performs the filtering operation on every vertical edge, then on every horizontal edge. There are 16 different positions along each macroblock edge. The approach we used to parallelize the code was to perform simultaneous filtering calculations for multiple positions along an edge. In order to perform parallel computations, we make use of SIMD vector operations. However, including SIMD into the reference deblocking code required considerable changes to the processing flow.

The deblocking operation involves a large number of conditional checks to provide adaptivity. These conditional checks determine the type of filtering applied, and which pixels are to be affected by the filtering. The same operations may not be performed for every position along the edge. In the original source code, conditional checks are performed on a pixel-by-pixel basis to determine what filtering should be applied. It is not possible to design efficient SIMD code while preserving these conditional checks. Instead, we used a “predicated” approach for the filtering operations to allow for SIMD calculation of eight positions simultaneously. The results for all possible branches are calculated. Then, after filtering half a macroblock edge (eight positions), we select results from the different registers based on a set of conditionals held in separate registers. This approach allows for speedup in this case due to the fact that the gains from performing filtering operations on eight positions in parallel offset the loss due to having to calculate multiple possible filter outputs per pixel. The specific speedup that we were able to observe using this approach will be discussed later in the paper.

C. Modification Details

The basic flow of execution in the program was described earlier in the paper. In order to introduce SIMD parallelism to the code, we modified the code so that the last step, the calculation of filtered pixel values, can be performed in a SIMD parallel manner. To implement our changes, we used Intel's SSE2 multimedia extension set on a Pentium IV workstation. SSE2 uses 128-bit integer registers. The arithmetic operations that are required operate on 8-bit values. To prevent clipping and overflow problems, we store the values as 16-bit. This allows us to pack eight 16-bit values per register, and perform operations on eight pixel positions in parallel. Here, pixel position means a single pixel location along the edge; filtering may affect up to six pixels per position, three to each side of the edge.

The calculation of filter strengths is left as a serial operation. Since there are a maximum of four different strength values along a macroblock edge, the code calculates four different strength parameters. Some general software optimization was performed on the filter strength calculation to remove redundant calculations from the deblocking kernel. The code could be restructured to allow for speedup on a superscalar processor through block pipelining, but that was not pursued for this paper.

Memory accesses are performed partially as serial operations, and partially using vector SIMD. For horizontal edges, we use aligned SSE2 vector loads and stores. For vertical edges, due to memory alignment, we use serial loading into vector registers. Similarly, memory stores are performed partially with vector SIMD and partially with serial instructions.

In order to allow for parallel operation, we store the results of the conditional checks employed in the filtering in Boolean mask registers; each register holds eight 16-bit values, each of which is either 0xFFFF or 0x0000. Filter calculations are performed using SSE2's parallel arithmetic operations. After filter computations are complete, the different possible output registers are selected using the conditional registers. A code sample using SSE2 intrinsic functions and a conditional register is given below.

```
Q0 = _mm_or_si128(
    _mm_and_si128(aq2, Q0_path1),
    _mm_andnot_si128(aq2, Q0_path2) );
```

In the example given, the value of Q0 is set based on a combination of values from Q0_path1 and Q0_path2. The selection of elements is based on the conditional register aq2. This type of operation happens regularly in the code, which led us to believe a new instruction to speed the operation above would be a good addition to the vector instruction set. The investigation into possible new instructions is described later in the paper.

TABLE I
DEBLOCKING KERNEL SPEEDUP THROUGH SSE2 SIMD

SSE2	S/W Opt	Time (s)	Speedup
No	No	5.405	1.000
Yes	No	3.613	1.496
No	Yes	3.346	1.615
Yes	Yes	1.229	4.398

D. Results

This section describes the relative performance of the new SSE2 implementation of the deblocking filter kernel to the original deblocking filter kernel. Tests were performed on 300 frames of the Foreman CIF test sequence. The test machine was a Pentium IV 2.8 GHz system with 2 GB of RAM. Code was optimized for speed during compilation (O3). The code was compiled using the Microsoft C++ compiler. Results are shown in Table I. As shown in the table, both software optimization (S/W Opt) and the inclusion of SSE2 produced similar levels of speedup. Using the original scalar code with some software optimization generated a speedup of 1.615, while rewriting the code to make use of SSE2 instructions produced a speedup of 1.496. When both of these were used in combination, we achieved a speedup of 4.398. This dramatic increase in speedup comes in part because the strength calculation part of the code, which is performed serially, benefited the most from the software optimization. Minimizing the serial portion allows for greater speedup from the SIMD operations; thus a large speedup is obtained when both S/W Opt and SSE2 are applied. These results clearly show that it is possible to get very large speedup over a serial deblocking filter implementation by using standard multimedia extensions such as SSE2.

IV. INSTRUCTION EXTENSIONS

A. Introducing New Instructions

Although significant performance benefits were observed from employing SSE2's SIMD instructions and from general software optimization, we also examined the possibility of improving the speedup through the addition of new SIMD instructions. To test and implement the new instructions, we employed the Xtensa tool suite from Tensilica [6]. The Xtensa tool set allows the user to customize a base Xtensa processor with new instructions. An application can be built with and without instruction extensions to measure the performance difference resulting from the new instructions.

Tensilica includes a base SIMD engine which can be built with an Xtensa processor, the Vectra LX DSP engine. Similar to SSE2, Vectra provides SIMD operations which are able to work on a set of eight packed 16-bit integers. The new instructions were created using Tensilica Instruction Extensions (TIE) on top of a Vectra base. The next sections will describe different instructions that we implemented in TIE to improve the performance of the deblocking filter.

B. New Instructions

1) Reproduced SSE2 Instructions: SSE2 provides some general purpose SIMD instructions that are missing in the Vectra SIMD set. An important set of instructions missing from

Vectra were SSE2's mask-generating comparison instructions. These and other instructions were useful for the deblocking application, so they were reproduced in TIE. With the addition of these instructions we have a "baseline" SIMD set which we could compare an extended instruction set against.

2) One-to-Many Set Instructions: One instruction that is missing in the SSE2 instruction set (and the Vectra set extended to emulate SSE2) is a one-to-many set instruction. The instruction takes a single scalar value and distributes it among all the elements of a vector. This instruction is useful in the initialization phase of the deblocking filter, where a single constant needs to be compared against a vector in a SIMD manner. This instruction is missing in the SSE2 SIMD instruction set, so it was added through a TIE extension. This instruction is functionally equivalent to the "splat"-type instructions found in the Altivec instruction set.

3) Arithmetic Instructions: Two common patterns of arithmetic operations are found in all the H.264 filter calculations. The patterns are shown below.

```
value = [(A >> 1) + B + C + 2] >> 2
value = [(A >> 1) + B + C + 4] >> 3
```

These patterns are used in most of the possible filter branches. Each pattern requires five standard SIMD instructions. We create one instruction for each of the two patterns. The shifts and correction factors are hard-coded into the instructions. The instructions perform the above operations on a set of eight packed 16-bit values in a SIMD manner.

4) Parallel Absolute Comparison: The pixel-level adaptivity of the H.264/AVC deblocking filter arises from comparisons made to the absolute difference between two pixel values (luma or chroma). To speed these operations, we introduced a SIMD mask-generating function which takes three arguments: value A, value B, and a threshold. The entries of the output register are formed by SIMD comparison to see if the absolute difference between vectors A and B is below the threshold vector. The result is a binary vector comprising fields of ones or zeroes, to be used as a data selection mask.

5) Parallel MUX instruction: Another instruction that we have introduced using TIE is *dblk_select*. This is a parallel MUX-type operation with three vector input registers. One of the registers is the selection register, the other two are the data source registers. Written in C code, the scalar form of the operation is described as follows:

```
for (i = 0; i < VEC_SIZE; i++)
   output[i] = select[i]? a[i] : b[i];
```

This instruction is useful in the H.264/AVC deblocking filter. More generally, it can be used in applications where a SIMD register needs to be filled using a combination of two or more sets of results. The advantage of this approach is that the selection mask is easily generated using a SIMD conditional instruction, as found in SSE2 and our extended SIMD set. With *dblk_select*, we reduce the number of instructions required per conditional selection from three (one parallel OR and two parallel ANDs) to one.

TABLE II
DEBLOCKING KERNEL SPEEDUP THROUGH INSTRUCTION EXTENSIONS

Implementation	Cycle Count	Speedup
Reference	257M	1.00
Standard SIMD	45.9M	5.62
Extended SIMD	44.5M	5.79

C. Performance Evaluation

The results that we obtained are summarized in Table II. We performed our tests using four frames of the Foreman CIF sequence. We test with three different implementations of the deblocking filter: the original reference implementation, a baseline implementation with software optimization and SIMD replicating that available in SSE2, and lastly our full SIMD implementation including extended TIE. The results were obtained using the Xtensa cycle-accurate profiling tool. The cycle counts given here are produced with the assumption that there are no stalls due to memory accesses. The custom SIMD instructions were created as single cycle instructions. As shown in Table II, the custom SIMD provides moderate speedup over the standard SIMD. The bulk of the speedup has been obtained through introducing SIMD parallelism. The custom instructions provide an incremental improvement of this parallelism. With SIMD implemented for the filtering kernel, serial portions such as the strength calculations become a more dominant part of the execution time.

V. CONCLUSION

In this paper, we have shown that a SIMD approach can be used to improve the performance of the H.264/AVC deblocking filter. We demonstrate that speedup is possible using SIMD, even in highly data-adaptive processing algorithms such as the H.264/AVC deblocking filter. Using standard SSE2 SIMD and a predicated approach, we have achieved a large speedup over the reference implementation. This paper also shows that the inclusion of some specific new instructions into existing SIMD instruction sets can provide additional speedup for the deblocking filter. Using a combination of the approaches described in this paper, it is possible to greatly speed up the H.264/AVC reference deblocking filter implementation.

REFERENCES

[1] P. List, A. Joch, J. Lainema, G. Bjontegaard, and M. Karczewicz, "Adaptive deblocking filter," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, pp. 614–619, July 2003.

[2] P. P. Dang, "An efficient implementation of in-loop deblocking filters for H.264 using VLIW architecture and predication," in *2005 International Conference on Consumer Electronics (ICCE) Digest of Technical Papers*.

[3] M. Sima, Y. Zhou, and W. Zhang, "An efficient architecture for adaptive deblocking filter of H.264/AVC video coding," *IEEE Trans. Consumer Electron.*, vol. 50, pp. 292–296, Feb. 2004.

[4] Intel Corporation. IA-32 Intel architecture optimization reference manual. [Online]. Available: ftp://download.intel.com/design/Pentium4/manuals/24896612.pdf

[5] M. Horowitz, A. Joch, F. Kossentini, and A. Hallapuro, "H.264/AVC baseline profile decoder complexity analysis," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, pp. 704–716, 2003.

[6] Tensilica. Xtensa LX. [Online]. Available: http://www.tensilica.com/products/xtensa_LX.htm