Analysis of Video Filtering on the Cell Processor

Arnaldo Azevedo, Cor Meenderinck, Ben Juurlink
Delft University of Technology
Delft, the Netherlands
Email: {Azevedo, Cor, Benj}@ce.et.tudelft.nl

Mauricio Alvarez †, Alex Ramirez † ‡
† Universitat Politecnica de Catalunya (UPC)
‡ Barcelona Supercomputing Center (BSC)
Barcelona, Spain
Email: alvarez@ac.upc.edu, alex.ramirez@bsc.es

***Abstract*— In this paper an analysis of bi-dimensional video filtering on the Cell Broadband Engine Processor is presented. To evaluate the processor, a highly adaptive filtering algorithm was chosen: the Deblocking Filter of the H.264 video compression standard. The baseline version is a scalar implementation extracted from the FFMPEG H.264 decoder. The scalar version was vectorized using the SIMD instructions of the Cell Synergistic Processing Element (SPE) and with AltiVec instructions for the Power Processor Element. Results show that approximately one third of the processing time of the SPE SIMD version is used for transposition and data packing and unpacking. Despite the required SIMD overhead and the high adaptivity of the kernel, the SIMD version of the kernel is 2.6 times faster than the scalar versions.**

I. Introduction

Video processing coders/decoders (codecs) are increasing in complexity to obtain better compression rates and improve the picture quality. The H.264 standard [1] is an example of this trend. H.264 generates a bitstream which is, on average, twice as small as the bitstream generated by the MPEG-2 and MPEG-4 standards, for the same picture quality. However, this improvement comes at the costs of increased computational complexity.

Many processors feature Single Instruction Multiple Data (SIMD) units to accelerate multimedia processing. IBM Cell Broadband Engine processor [2] is an example of a multimedia tailored processor with eight cores that work exclusively in a SIMD fashion. The increasing complexity of multimedia kernels, however, can have a negative impact on the efficiency of SIMD processing, due to the fact that addition of complex control structures decreases data-level parallelism.

This work analysis the suitability of the Cell Broadband Engine for bi-dimensional video filtering. To evaluate the processor a highly adaptive filtering algorithm was chosen: the Deblocking Filter (DF) of the H.264 video compression standard. The DF is not the largest kernel of H.264, but due to Amdahl's law might take up to 49% of the processing time if not SIMDmized along the other kernels [3].

A SIMD implementation of the DF for the Cell processor is presented and analyzed in this paper. SIMD implementations for SSE2 have been presented in [4], [5] which report speedups of 1.13 and 1.49 respectively. However, the Cell processor has some distinct features, like DMA access and scratchpad memory, that significantly impacts performance.

This paper is organized as follow. Section II presents a brief overview of the Cell Broadband Engine. The Deblocking Filter is described in Section III and its implementation on the Cell processor is given in Section IV. Section V presents experimental results and comparisons, and Section VI concludes the paper.

II. Cell Broadband Engine

In this section a brief overview of the Cell Broadband Engine is provided, which focuses on the characteristics relevant to the implementation of the DF. More details about the processor can be found in [2], [6].

The Cell Broadband Engine is a heterogeneous multi-core processor consisting of one Power Processor Element (PPE) and eight Synergistic Processing Elements (SPEs) connected by four 16B-wide data rings [6]. The PPE is a dual-threaded, two-way in-order PowerPC with AltiVec extension. The SPEs are in-order 2-way RISC-like cores with a local store (scratchpad memory), and a DMA unit.

The SPEs are tailored for multimedia processing and have 128 registers of 128 bits wide. All instructions are SIMD and they operate on 128-bit data with varying element width, i.e. 2×64-bit, 4×32-bit, 8×16-bit, 16×8-bit, and 128×1-bit. The SPE compiler manages scalar operations by first moving both operands to the preferred slot of a register, next performing the operation, moving the result to the destination slot, and finally writing it back.

The SPE can only access data and code stored in its 256 KB Local Store (LS). The LS is mapped onto the main memory address space to allow LS-to-LS communication, but this memory (if cached) is not coherent in the system. Data and instructions are transferred, in packets of maximally 16 KB, between LS and main memory by explicit DMA commands, executed by the SPEs DMA unit. Data communication can be performed in parallel with computation. Thus a double buffering strategy can be employed to hide DMA transfer latency.

III. Deblocking Filter

The discrete cosine transform applied in video and image compression can produce an artifact known as blocking, because of visible square areas in the picture. The aim of the DF is to improve the appearance of the decoded pictures by smoothing the block edges. In H.264 this filter is mandatory.

978-1-4244-1684-4/08/$25.00 ©2008 IEEE

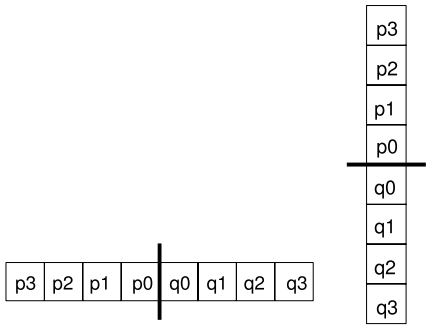

Fig. 1. Line of pixels used for the vertical (a) and the horizontal (b) edges.

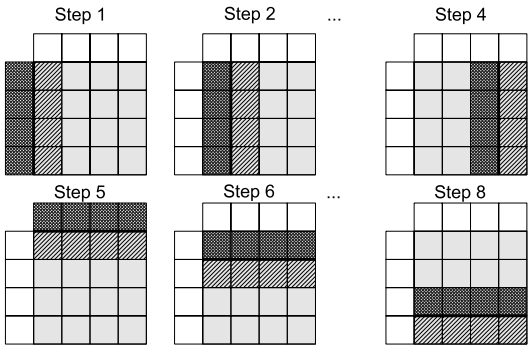

Fig. 2. The filtering process of one macroblock.

The DF is highly adaptive and has different filter strengths depending on the block types, e.g., intra- and inter-predicted types.

The filters employed in the DF are one-dimensional. The bi-dimensional behavior is obtained by applying the filter to both the vertical and the horizontal edges of all 4×4 luma or chroma blocks. The DF is applied to a line of pixels orthogonal to the block edge (see Figure 1). Let qi $(0 \leq i \leq 3)$ denote the pixels of the current block and let pi denote the pixels of the neighboring blocks. Depending on the filter strength, the values of pixels $p2$ to $p0$ and $q0$ to $q2$ are modified, $p3$ and $q3$ remain unaltered.

The DF process first filters the left edges of the macroblock (MB) and then filters the vertical internal edges. This process is repeated for the horizontal edges. This process is illustrated in Figure 2, where each box represents a 4×4 luma block. The gray area represents the current MB, darkly-hatched the p blocks, and the q blocks are lightly-hatched.

The strength of the filter is determined dynamically and depends on the current quantizer, the coding of the neighboring blocks, and the gradient of the image samples across the boundary [7]. There are five Boundary Strengths (BS) which the filter can apply, ranging from 0 (no filtering) to 4 (strongest one). Boundary strength 4 is applied between edges of two Intra Prediction blocks, when one of them is a MB boundary. Boundary strength 3 is applied between edges of an Intra Prediction blocks. The other are used to edges between Inter prediction blocks.

```
if (p0 - q0  < (alpha << 2) + 2){
    if (p2 - p0 < beta){
        p0' = (p2 + 2*p1 + 2*p0 + 2*q0 + q1 + 4) >> 3;
        p1' = (p2 + p1 + p0 + q0 + 2) >> 2;
        p2' = (2*p3 + 3*p2 + p1 + p0 + q0 + 4) >> 3;
    }
    else
        p0' = (2*p1 + p0 + q1 + 2) >> 2;
} else
    p0' = (2*p1 + p0 + q1 + 2) >> 2;
```

Filtering is applied over if the conditions $(p0-q0) < alpha$, $(p1-p0) < beta$ and $(q1-q0) < beta$ are met. The thresholds $alpha$ and $beta$ depend on the encoder average quantization parameter over the edge. There are two filter functions. When the BS value is 4, the function above is applied, where pi' is the new value for pixel pi. The function below is applied for all other BS values. The code presented filters the p pixels. The equations for the q pixels are symmetrical.

```
clip(x, y, z){
  return x < y ? y : ( x > z ? z : x)
}
clip(x) {return clip(x, 0, 255)}

if( p2 - p0 < beta){
  p1'' = ((p2 + ((p0 + q0 + 1) >> 1)) >> 1) - p1;
  p1' = p1 + clip(p1'', -tc0, tc0);
}

delta' =  (((q0 - p0 ) << 2) + (p1 - q1) + 4) >>3;
delta  =  clip(delta',-tc, tc);
p0'    =  clip( p0 + delta );
```

IV. Implementation

In this section the implementation of the DF on the Cell Broadband Engine is detailed. The implementation will be described in a top-down fashion. The description starts with the main loop of the kernel and gradually goes down to the inner parts of the implementation. The focus of this analysis is the computational part of the DF of the FFMPEG H.264/AVC decoder.

The baseline version is a scalar implementation extracted from the FFMPEG H.264 decoder [8]. The extracted code does not include the parameter calculation of the DF. The analysis focuses on the sample filtering of the code. This scalar version was then vectorized using the SIMD intrinsics for the SPE and AltiVec intrinsics for the PPE. Because the AltiVec implementation is very similar to the SPE version only the second one will be presented.

In the SPE implementations the PPE is used only to read the parameters from the input files and to store them in main memory. After storing the parameters, the SPE threads are spawned. Thereafter, the PPE thread sends a signal to all SPEs to start the computation.

Each SPE thread processes one frame. This approach avoids data movements between SPEs and/or between SPEs and main memory as all data dependencies are between instructions executed on the same SPE. The processing starts by reading

the input pointers for the samples and parameters from the main memory.

Each frame is divided into MB lines (MBs from the same row), to use the SPEs ability of performing computation and data communication in parallel. This partition is based on several factors such as the latency, maximum DMA transmission package size, number of DMA transfers, and organization of the data in the memory. The partition size is proportional to the start-up latency but inversely proportional to the number of DMA requests. The pixel components Y, Cb, and Cr are stored in separate arrays. Partitioning into complete lines of MBs allows to load the pixel samples of one partition with three DMA transfers. Using smaller partitions would require to DMA each line separately.

For every MB line there are four DMA transfers from memory to the LS. One DMA transfer is necessary for 16 lines of luma samples, two for 8 lines of each set of chroma samples, and another one for the DF parameters of the MB line. After the data is available in the LS, the processing of the MB line is performed and the results are transmitted back to the main memory.

The processing of the MB lines is performed as a software pipeline and uses a double buffering strategy. First, the data for the first MB line is requested, followed by the request of the data for the second MB line. After the data of the first MB line is available in the LS it is filtered. This way the processing of MB line 0 is performed in parallel with the data transmission of MB line 1. The pseudo-code below illustrates the process:

```
Request (MB_Line[0]);

Request (MB_Line[1]);
Wait    (MB_Line[0]);
Process (MB_Line[0]);

FOR x = 2 TO frame_height_in_MB -1
{
    Request (MB_Line[x]);
    Wait    (MB_Line[x-1]);
    Process (MB_Line[x-1]);
    Save    (MB_Line[x-2]);
}

Process (MB_Line[x-1]);
Save    (MB_Line[x-2]);

Save    (MB_Line[x-1]);
```

The MB line cannot be immediately transmitted to memory. As can be seen in Figure 2, the processing of the next MB line changes the values of the current bottom edge samples.

The filter process is performed per MB. As described in section III, there are 8 edges, four vertical and four horizontal. First the four vertical edges are filtered and then the four horizontal. The filtering process is divided into the following steps: (1) unpack the 8-bit (8b) samples of the current and left MBs to signed 16b, (2) transpose the current and left MBs, (3) filter the vertical edges, (4) transpose the result, (5) pack back the left MB result to 8b, (6) unpack the last 4 lines of the top MB to 16b, (7) filter the horizontal edges, and finally, (8) pack back the MB result to 8b.

The computational core of the DF is the edge filtering. There are four functions required to implement the edge filtering. Luma and chroma samples require two functions each: one for Intra MB external edges blocks and another one for the other cases. These functions exhibit data-level parallelism and have been optimized with SIMD instructions of the SPE, such that the edge filtering computes 8 pixels simultaneously.

Despite their adaptivity, the filtering functions have been implemented without branches, except for one to select between the filter processes listed above. To perform the filtering without branches, all equations of the function are computed. All branches are replaced by comparisons that result in a mask. These masks are used to select the positions of the result vectors that will be saved in memory.

V. Experimental Results

In this section the experimental results are presented. First the experiment input and methodology are described followed by the analysis of the results. Based on the analysis the conclusions are drawn.

As input, the first eight frames of the Lake Wave video sequence, in the QVGA (320×240 pixels) resolution. The results were obtained using the hardware counters of a 2.4 GHz Cell processor. The presented results are the average of three runs with eight frames each.

Figure 3 depicts the time in milliseconds required to filter a frame for each version of the kernel: the scalar version running on the PPE (PPE - Scalar), the AltiVec version on the PPE (PPE - AltiVec), the scalar version on the SPE (SPE - Scalar), the SIMD version on the SPE without double buffering (SPE - SIMD no D.B.), and the SIMD version on the SPE using double buffering (SPE - SIMD D.B.). For the scalar versions, filtering a QVGA frame takes on average 2.79 and 2.72 ms on the PPE and SPE, respectively. The PPE AltiVec implementation has an average run-time of 2.15 ms, which is 30% faster than the PPE scalar version. The SPE SIMD version takes 1.14 ms per frame, while using a double buffering technique reduces the average runtime to 1.05 ms. The latter correspond to a speedup of 2.6 compared to the SPE scalar version.

As the AltiVec and SPE SIMD versions are almost identical the difference in speedup obtained by SIMDmizing is interesting. A critical difference between the PPE Altivec unit and the SPE is that the first one has only 32 vector registers while the second has 128. Profiling the SPE-SIMD version using the IBM Full-System Simulator [9] showed that all 128 registers are used, with an average of 120 alive registers. Because the MB data set is much larger than the PPE Altivec register file, additional load/stores are required decreasing performance. Moreover, the SPE versions benefit from the local memory and the direct access to main memory through the DMAs, as the kernel has a predefined memory access pattern.

The performance difference between the double buffering implementation with the non-double buffering is only 8%. This is because the DF kernel spends much more time operating than acquiring the data.

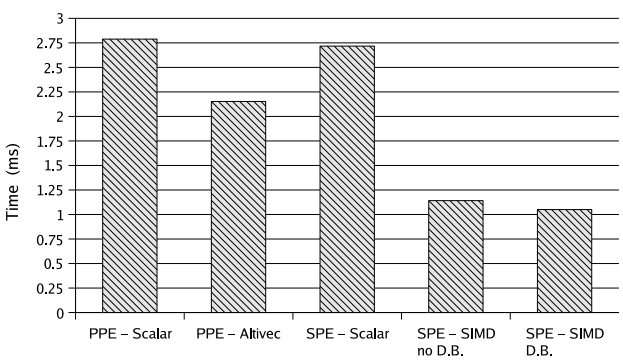

Fig. 3. Average performance of the deblocking filter implementations for one QVGA frame.

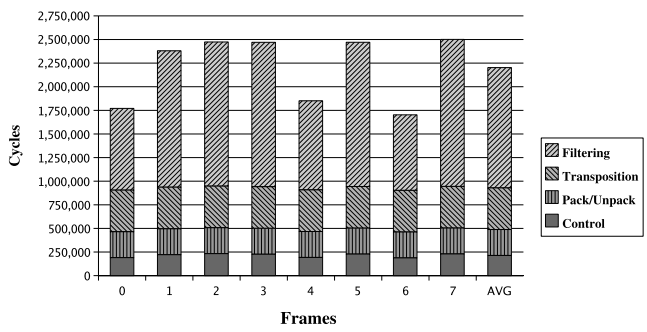

Fig. 4. Deblocking filter performance on the SPE for 8 QVGA frames.

The required SIMD overhead was also measured using the IBM Cell simulator. For profiling purposes the kernel was divided into four parts: Transposition, Pack/Unpack, Filtering, and Control. We call Control all parts of the kernel other than the other parts. Figure 4 depicts the number of cycles spent in each part of the kernel. Filtering consumes 47% to 62% of the cycles required to process a frame, with an average of 58%. Approximately one third of the time is spent on the Transposition and Pack/Unpack parts, they consume on average 20% and 12%, respectively. The remaining 10% is used by the control structures of the kernel, e.g., data requests and function calls.

The profiling results also show that the double buffering strategy hides the communication latency. The total number of cycles that the cores were stalled waiting for data from memory accounts for only 0.4% of the total running time, on average. As subsequent frames can be overlapped, the data communication latency influences only in the start-up of the process.

This experiment shows that for video processing the SIMD implementation on the SPE was a significant performance improvement over the scalar implementations. The overall speedup is considerable and pays off the extra effort of SIMDmizing the code. The speedup of 2.6 can be considered a good result, considering the high adaptivity of the filtering process and the high overhead required by the SIMD processing, such as transposing and data packing and unpacking. DMA access and the large register file are the main contributors to the speedup.

VI. Conclusions

This work presented the H.264 Deblocking Filter as a case study of the video filtering on the Cell Broadband Engine processor. An overview of the Cell processor and of the DF kernel of the video standard were presented. The DF was SIMDmized and ported to the SPEs of the Cell processor. Profiling was performed using the IBM Cell Simulator, while performance measurements were obtained using hardware counters. Results show that approximately one third of the processing time of the SPE SIMD version is spend on transposition and data packing and unpacking. Despite this SIMD overhead and the high adaptivity of the kernel, the SIMD version of the kernel is 2.6 times faster than the scalar version, on both the SPE and PPE. This experiment shows that for video processing the SIMD implementation on the SPE provides a significant performance improvement over the scalar implementations.

Currently we are replacing the DF of the H.264 decoder of FFMPEG by our modified version and porting it to the Cell processor. These activities is part of our plan to port and optimize the complete H.264 decoder to the Cell processor and also implement a macroblock-level parallelization.

Acknowledgment

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6), the HiPEAC Network of Excellence (IST-44608), and the Ministry of Education and Science of Spain under contract TIN2007-60625.

References

[1] T. Wiegand, G. Sullivan, G. Bjntegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 560–576, 2003.

[2] J. Kahle, M. Day, H. Hofstee, C. Johns, T. Maeurer, and D. Shippy, "Introduction to the Cell Multiprocessor," *IBM Journal of Research and Development*, vol. 49, no. 4, pp. 589–604, 2005.

[3] M. Alvarez, E. Salami, A. Ramirez, and M. Valero, "A Performance Characterization of High Definition Digital Video Decoding Using H.264/AVC," in *Workload Characterization Symposium. Proceedings of the IEEE International*, 2005, pp. 24–33.

[4] J. Lee, S. Moon, and W. Sung, "H.264 Decoder Optimization Exploiting SIMD Instructions," in *Circuits and Systems. Proceedings. The IEEE Asia-Pacific Conference on*, vol. 2, 2004.

[5] S. Warrington, H. Shojania, and S. Sudharsanan, "Performance improvement of the H. 264/AVC deblocking filter using SIMD instructions," in *Circuits and Systems, 2006. ISCAS. Proceedings of IEEE International Symposium on*, 2006, p. 4.

[6] M. Gschwind, H. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki, "Synergistic Processing in Cell's Multicore Architecture," *IEEE Micro*, vol. 26, no. 2, pp. 10–24, 2006.

[7] P. List, A. Joch, J. Lainema, G. Bjntegaard, and M. Karczewicz, "Adaptive Deblocking Filter," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 614–619, 2003.

[8] "The FFmpeg Libavcoded." [Online]. Available: http://ffmpeg.mplayerhq.hu/

[9] "IBM Full-System Simulator for the Cell Broadband Engine Processor." [Online]. Available: http://www.alphaworks.ibm.com/tech/cellsystemsim