Intra-Vector SIMD Instructions for Core Specialization

Cor Meenderinck and Ben Juurlink
Computer Engineering Laboratory
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology, Delft, The Netherlands
{cor,benj}@ce.et.tudelft.nl

Abstract— **Research is mainly focussing on exploiting TLP to increase performance. Another avenue, however, for achieving performance scalability is specialization. In this paper we propose a handful of application specific intra-vector instructions for two dimensional signal processing kernels. Utilizing the SIMD capabilities in the row wise operations requires significant data rearrangement overhead. When using the intra-vector instructions for those operations instead, the overhead can be avoided. We have implemented intra-vector instructions in the Cell SPU core and measured speedups up to 2.06, with an average of 1.45.**

I. Introduction

The pursuit for more performance mainly focusses on exploiting TLP using chip multi-processor (CMP) architectures. However, another avenue for achieving performance scalability, orthogonal to the one mentioned above, is specialization of cores for specific application domains. Application specific cores can run the same workload in fewer cycles. Thus, fewer cores or a lower clock frequency can be used. The first option is especially attractive for low-cost embedded systems where less silicon area directly translates to less cost.

The specialization proposed in this paper applies to SIMD units, contained in many architectures to exploit DLP. Normally, a SIMD unit is divided in identical lanes, each performing the same operation on corresponding data elements in the input vectors. In previous work [1], while specializing a core for video decoding, we noticed that often two dimensional kernels cannot fully exploit the DLP offered by the SIMD unit. For vertical processing, the data is aligned in a fashion that fits SIMD processing. Horizontal processing though, requires significant data rearrangement before SIMD processing can be applied. This overhead was reduced by using intra-vector instructions that break with the strict separation of SIMD lanes.

In this paper we analyze the applicability of intra-vector instructions to several two dimensional signal processing kernels. The Cell SPE core was used a baseline architecture and simulation was used to evaluate the performance of the enhanced kernels.

Intra-vector instructions can be found in several processors. Most general purpose SIMD extensions, such as SSE and AltiVec, have some limited form of intra-vector instructions. They range from non-arithmetic operations such as permute, pack, unpack, and merge, to arithmetic operations such as dot product, sum across, and extract minimum. Intra-vector instructions are even more common in the DSP area. NXP's EVP [2], [3] has intra-vector processing capabilities that allows, for example, arbitrary reordering of data within a vector as needed for FFT butterflies, pilot channel removal, and other computations common in communications signal processing. The Sandblaster architecture [4] has several intra-vector reduction operations, such as the multiply and sum operations. Furthermore, there are intra-vector instructions for Viterbi and Turbo decoding. More general intra-vector permutations are provided in the CEVA-XC [5], Lucent's 16210 [6], and SODA [7] architectures. The novelty in our work is twofold. First, we present intra-vector instructions for kernels from the media domain, while the others target the communication domain. Second, as far as we know no one has used intra-vector instructions to solve the data rearrangement problem in two dimensional kernels.

This paper is organized as follows. Section II provides an overview of the Cell SPE architecture. The experimental setup is described in Section III. Next, in Section IV, the application and evaluation of intra-vector instructions in several kernels is presented. Section V discusses the results while Section VI concludes the paper.

II. Overview of the SPE Architecture

The Cell processor consists of one PPE (Power Processing Element) and eight SPEs (Synergistic Processing Elements). The PPE is a general purpose PowerPC that runs the operating system and controls the SPEs. The SPEs function as accelerators. They operate autonomously but are depending on the PPE for receiving tasks to execute. The SPE consists of a Synergistic Processing Unit (SPU), a Local Store (LS), and a Memory Flow Controller (MFC) as depicted in Figure 1. The SPU has direct access to the LS, but global memory can only be accessed through the Memory Flow Controller (MFC) by DMA commands. As the MFC is autonomous, a double buffering strategy can be used to hide the latency of global memory access. While the SPU is processing a task, the MFC is loading the data, needed for the next task, into the LS.

The SPU has 128 registers, each 128 bits wide. All data transfers between the SPU and the LS are 128-bit wide. Also the LS accesses are 128-bit aligned. The ISA of the SPU is completely SIMD and the 128-bit vectors can be treated as one quadword (128-bit), two doublewords (64-bit), four words (32-bit), eight halfwords (16-bit), or 16 bytes.

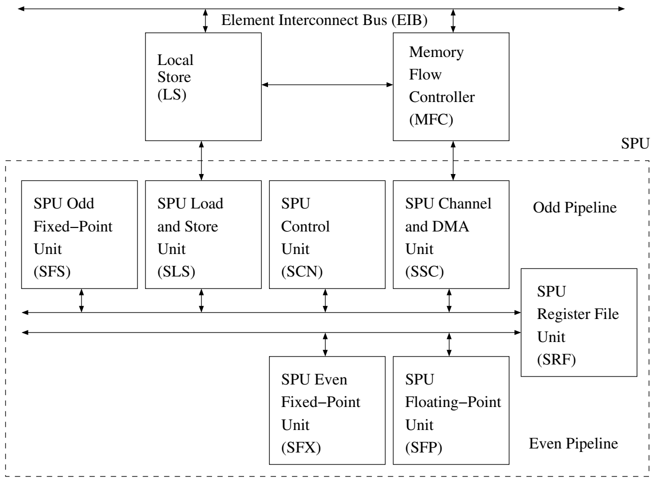
Element Interconnect Bus (EIB)
Local Store (LS)
Memory Flow Controller (MFC)
SPU
SPU Odd Fixed−Point Unit (SFS)
SPU Load and Store Unit (SLS)
SPU Control Unit (SCN)
SPU Channel and DMA Unit (SSC)
Odd Pipeline
SPU Register File Unit (SRF)
SPU Even Fixed−Point Unit (SFX)
SPU Floating−Point Unit (SFP)
Even Pipeline

Fig. 1. Overview of the SPE architecture (based on [8]).

There is no scalar datapath in the SPU, but scalar operations are performed using the SIMD registers and pipelines. For each data type, a preferred slot is defined in which the scalar value is maintained. Most of the issues of scalar computations are handled by the compiler. For more details on the SPU architecture, the reader is referred to [9].

We chose the SPE as the baseline architecture to add the intra-vector instructions to. The availability of a simulator and compiler, as well as our familiarity with those tools and the architecture were reasons for this choice. Any other SIMD unit could be extended with intra-vector instructions as well, and thus our main conclusions are general.

III. Experimental Setup

CellSim [10] was used as the simulation platform. It is not cycle accurate, but it's parameters allows to adjust the execution to match the real processor. We compared the execution times measured in CellSim with those obtained with the IBM full system simulator SystemSim. We did not use the real processor as SystemSim provides more statistics and is accurate as well. Those additional statistics allowed us to check the correctness of CellSim on more aspects than just execution time. The validation showed that the difference in execution time is at most 3.2%. Comparison of other statistics generated by the two simulators also showed that CellSim is well suitable for its purpose.

We used six different kernels to demonstrate the effectiveness of intra-vector instructions. The kernels we used were either develop in-house before or were taken from CellBench [11], which is a collection of publicly available Cell kernels. We examined all kernels available, searching for opportunities to apply intra-vector instructions. That is, we looked at inefficiencies of the SIMDimization and inefficiencies in the implementation of the algorithm. Devising intra-vector instructions for a kernel requires a thorough understanding of the algorithm the kernel implements. Six kernels seemed amenable to the intra-vector paradigm, but we expect experts to be able to implement intra-vector instructions in many more kernels. The kernels used throughout this work are IDCT8, IDCT4, IPol (InterPolation), and DF (Deblocking Filter), originating from the FFmpeg H.264 video decoder [12]. Furthermore, we used the matrix multiply (MatMul) kernel from the Cell SDK. The last kernel is DWT (Discrete Wavelet Transform), taken from JasPer JPEG-2000 for Cell [13].

IV. Intra-Vector Instructions

The principle of using intra-vector instructions to enhance two dimensional kernels is well explained by the simple example of Figures 2 and 3. In both figures the elements of a matrix are processed. The matrix is stored in row-major order and, in this case but not necessarily, one row fits in one SIMD vector. Figure 2(a) shows a vertical processing, i.e., the elements of each column are added. Such a computation can efficiently be performed by traditional SIMD instructions, as the `vector-add` depicted in Figure 2(b).

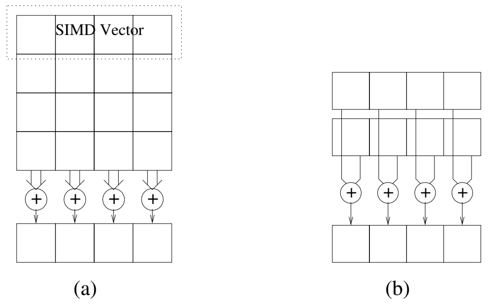

Fig. 2. Example of vertical processing of a matrix. Adding the elements of each column (a) can efficiently be performed with traditional SIMD instructions, such as the `vector-add` instruction (b).

In Figure 3(a), the matrix is processed in horizontal mode, i.e., the elements of each row are added. To perform this computation using traditional SIMD instructions, the data has to be rearranged, for example by a matrix transpose. An intra-vector instruction, such as the AltiVec `sum-across` instruction depicted in Figure 3(b), can be used to avoid this overhead.

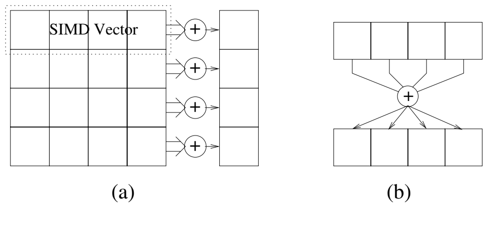

Fig. 3. Example of horizontal processing of a matrix. Adding the elements of each row (a) cannot efficiently be performed with traditional SIMD instructions. Intra-vector instructions, such as the AltiVec `sum-across` instruction (b) do allow efficient computation.

The benefit of intra-vector instructions is not only avoiding rearrangement overhead as we will show in this section. Also collapsing multiple operations in a single instruction and avoiding pack and unpack adds to the benefits of this type of instruction.

For six kernels intra-vector instructions were developed and evaluated. Due to space limitations we do not report on all six kernels extensively. Instead we discuss a few and highlight the main points of interest.

A. Interpolation Kernel

The IPol (InterPolation) kernel is the most time consuming part of H.264 motion compensation. It generates new pixels by interpolating the surrounding pixels and is applied to blocks ranging from 4×4 to 16×16 pixels. This interpolation is applied horizontally, vertically, or in both directions and consists of a 6-tap FIR filter. The horizontal FIR filter, computing one new pixel x'_0, is defined as follows:

$$x'_0 = (20(x_0 + x_1) - 5(x_{-1} + x_2) + x_{-2} + x_3 + 16) \gg 5. \qquad (1)$$

The result of the filter is clipped between 0 and 255.

The vertical mode of this kernel can efficiently be SIMDimized. For the horizontal mode we developed the `ipol` intra-vector instruction of which the computational structure is depicted in Figure 4. Pixel data are eight bit wide and thus 16 pixels fit in one SIMD vector. The input vector contains pixels x_{-2} through x_{10}. The `ipol` instruction computes pixels x'_0 trough x'_7 by applying eight FIR filters. Note that loading the input vector starting at x_{-2} does not introduce additional alignment overhead compared to a vector starting at x_0. Motion vectors can point to any pixel and thus both x_0 and x_{-2} have equal probability of being stored at an aligned memory address. Actually, using the intra-vector instructions only one vector has to be aligned, while in the baseline kernel this had to be done for six vectors.

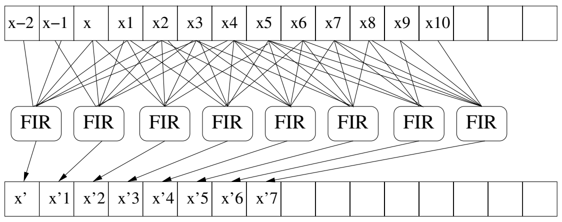

Fig. 4. The `ipol` instruction computes eight FIR filters in parallel.

Each FIR filter consists of eight additions, one saturate, and a few shifts. The longest path consists of five fixed point operations. As those take two cycles in the SPE architecture we set the latency of this instruction to ten cycles. This approach is rather pessimistic but on the safe side. The latencies of all intra-vector instructions were determined in this way.

The speedup and instruction count reduction for the IPol kernel (as well as for all other kernels) is depicted in Figure 5. A speedup of 1.57 is achieved, on the total two dimensional kernel. This speedup is mainly achieved by reducing the number of instructions by 1.76 times. Each `ipol` instruction replaces 67 instructions from the baseline kernel, including the following:

- Creation of auxiliary variables, such as vector of constants and masks for shuffling.

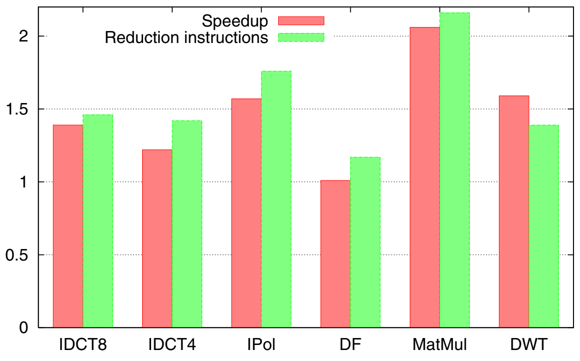

Fig. 5. Speedup and instruction count reduction achieved using intra-vector instructions.

- Aligning 6 input vectors (requiring some branches), instead of one.
- Unpack from bytes to halfwords and pack vice versa.
- Computation: 8 multiply-adds (each requiring a pack from words to halfwords), 8 additions, 2 subtractions, 2 shifts, and 8 instructions for saturation.

Note that the `ipol` instruction is only applied in the horizontal mode, and therefore replacing 67 instructions with one results in an instruction count reduction of 1.76 and not higher.

The speedup of the IPol kernel is lower than the instruction count reduction, which is the case for all but one kernel. The reason is the following. Multiple instructions are replaced by one. This decreases the code size but also the average distance between dependent instructions. Thus, there is less opportunity to schedule two instructions in the same cycle, or dependency stalls might even occur. This is reflected in Figure 6 which shows a breakdown of the execution cycles. It shows that the enhanced IPol kernel has much less dual issue cycles then the baseline.

The IDCT8 and IDCT4 kernels are very similar to the IPol kernel in the way intra-vector instructions were applied. The speedups achieved are 1.39 and 1.22, respectively. The speedup of the IDCT4 kernel was limited by dependencies. The inner loop of the kernel is small and by replacing multiple instructions with the `idct4` intra-vector instruction, the few instructions left were highly depending. Figure 6 shows that the dependency stall cycles nearly doubled.

These three kernels, IDCT8, IDCT4, and IPol, show that large speedups (1.39, 1.22, and 1.57 respectively) can be achieved on kernels by adding one intra-vector instructions per kernel to the ISA.

B. Deblocking Filter Kernel

The Deblocking Filter (DF) kernel [14] is applied to remove blocking artifacts introduced by the IDCT. The DF kernel smoothes the block edges and thereby improves the appearance of the decoded picture. As with the previous kernel, the filtering of the DF kernel is applied both horizontally and vertically. The horizontal filtering was implemented by two matrix transposes and a vertical filter in between. Thus,

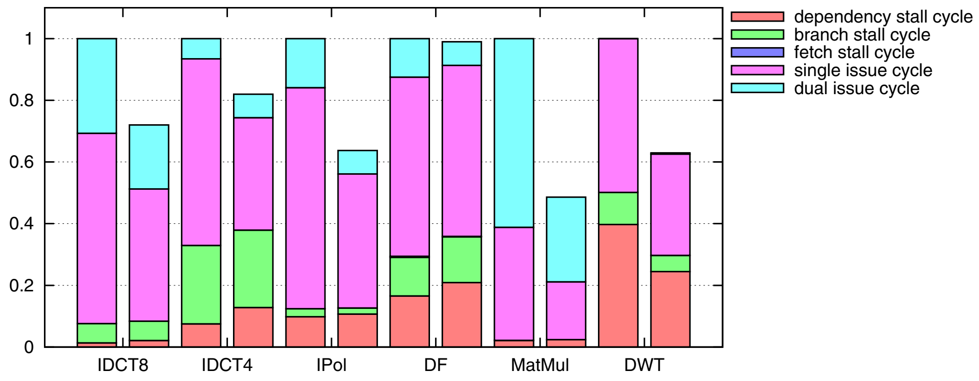

Fig. 6. Breakdown of the execution cycles normalized to the baseline kernel. For each kernel, the left is the baseline while the right is the enhanced version using intra-vector instructions.

at first glance this kernel seems a good candidate to speed up with intra-vector instructions.

The actual filter applied in the DF kernel is depending on several parameters. Therefore, several intra-vector instructions were developed, but all with the computational structure of Figure 7. The input values are 16-bit in this case and thus eight samples fit in one SIMD vector. The samples p_3 through p_0 are the pixels components on the left of a block boundary, while the q samples are those on the right. The values $alpha$ and $beta$ are parameters passed on to the filters (F).

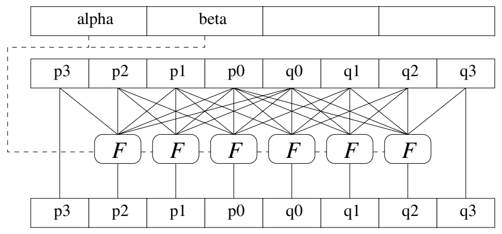

Fig. 7. Computational structure of the DF intra-vector instructions.

If the filter is applied to 4×4 blocks, in half the cases the vector $\{p_3,...,q_3\}$ is located in an unaligned position. For 8×8 blocks it is always located in an unaligned position. Therefore, rearrangement overhead is required to align the data. This was also the case with the IPol kernel, but there is a major difference. In the latter, the enhanced kernel had less alignment overhead compared to the baseline kernel. In the DF kernel, the baseline implementation had no alignment overhead, so the intra-vector instructions bring along additional overhead costs, which is the first negative effect on the performance. This is well reflected in Figure 5. Each intra-vector instruction replaces many, but the instruction count is reduced only little (1.17), because of the additional overhead instructions.

The second negative effect on the performance are the dependencies among the instructions in the inner loop, despite applying loop unrolling. Figure 6 shows an increase in dependency stalls for that reason. It also shows the third negative effect, an increase in branch stall cycles caused by an additional if statement in the inner loop of the kernel. The last negative effect is a low amount of DLP in the intra-vector instructions. Some of those compute only two outputs, while the original SIMD version always computed eight outputs. In total, the latency of the inner loop increased from 77 to 275 cycles. Avoiding the two matrix transposes counterbalances this increase in latency, such that the total speedup is 1.01.

Although the speedup is negligible, the results are very interesting as they show some conditions for efficient utilization of intra-vector instructions.

C. Matrix Multiply Kernel

Matrix multiplication (in short MatMul) is a two dimensional kernel, but in contrast to all previous kernels, it does not have a separate horizontal and vertical mode. Instead, it processes one input matrix along the rows and the other along the columns. Also in this case, intra-vector instructions are beneficial.

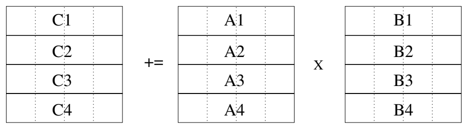

Fig. 8. Data layout of the baseline 4×4 matrix multiply. The vectors of matrix A have to be broken down in order to compute the matrix multiply.

An efficient implementation using the multicore and SIMD capabilities of the Cell processor is provided with the SDK 3.0. We used the optimized SIMD version of that code. The entire matrix multiply is broken down in multiplications of 4×4 submatrices. The SIMD width of the SPU is four slots of single precision floats, and thus one row of the 4×4 matrix is contained within one SIMD vector as depicted in Figure 8. To compute the first row of C ($C1 += a1\_1*B1+a1\_2*B2+a1\_3*B3+a1\_4*B4$), the elements of vector A1 (called $a1\_x$) have to be extracted and splatted. In total computing one row takes 12 instructions.

Using intra-vector instructions, the matrix multiplication could be broken down into blocks of 2×2 . A 2×2 block contains four floats which fit in one SIMD vector. Thus a 2×2 matrix multiply can be done in one instruction. We implemented the `fmma` (Floating Matrix Multiply Add) instruction. Assuming the data of a 4×4 matrix is rearranged such that the four 2×2 blocks are located within SIMD vectors (as indicated in Figure 9), one block of the output can be computed with two `fmma` instructions only (see Listing 1), e.g., $C1 \mathrel{+}= A1 * B1 + A2 * B3$. Usually the input data is not arranged in such a way, and therefore additional conversion overhead is required. However, this rearrangement has time complexity of $O(M^2)$, and thus relatively the overhead decreases for increasing matrix size. For a matrix of 1024×1024 the overhead is only 1.2%.

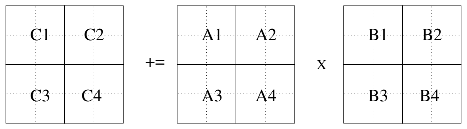

Fig. 9. Data layout of the enhanced 4×4 matrix multiply using 2×2 sub blocks. Each 2×2 block is located within one SIMD vector.

```
C1 = spu_fmma(A1, B1, C1);
C1 = spu_fmma(A2, B3, C1);
```

Listing 1. Computation of block C1 from Figure 9 takes only two `fmma` instructions, while that of row C1 from Figure 8 takes 12 instructions.

Figure 5 shows that a huge speedup of 2.06 is achieved. This is without the rearrangement overhead, which is depending on the input size, but insignificant for normal sizes. The speedup is much larger than that of the previous kernels because the intra-vector instructions affected the entire kernel, while for the previous kernels it affected the horizontal mode only.

D. Discrete Wavelet Transform Kernel

The Discrete Wavelet Transform (DWT) allows to generate a compact representation of a signal taking advantage of data correlation in space and frequency [15]. It is used in different fields, ranging from signal processing, image compression, data analysis, and partial differential equations, to denoising. We used the 2D DWT from the Cell implementation of JPEG-2000 [16], which uses a lifting scheme.

The DWT kernel applies five levels of decomposition, each consisting of two lifting steps and one combine step. In the baseline kernel, these steps are performed sequentially, but using intra-vector instructions they can be combined. Figure 10 depicts the computational structure of the `idwt` intra-vector instruction. The input array contains a low-pass band and a high-pass band. One SIMD vector from both (data elements are 32-bit wide) contain all inputs necessary to compute one output vector. The entire array can be processed by shifting the input and output vectors by two and four positions, respectively. The edges are processed slightly different and are not depicted.

Figure 5 shows that a significant speedup of 1.59 is obtained. Interestingly, the speedup is larger than the instruction

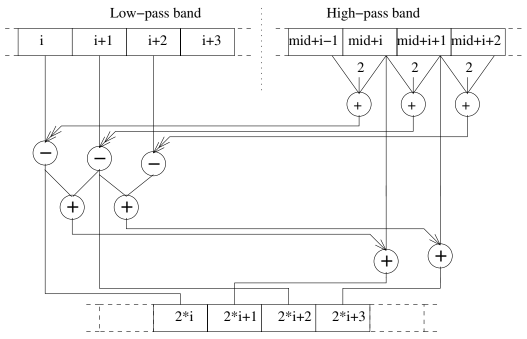

Fig. 10. The computational structure of the `idwt` intra-vector instruction. An arrow indicates a shift right by one bit.

count reduction. Figure 6 shows that the baseline kernel suffered severely from dependency and branch stalls. The new method of processing the array, enabled by intra-vector instructions, proves to be more efficient as both types of stall cycles are substantially reduced. The DWT can also be applied to a one dimensional data array. In that case the speedup achieved will even be larger. We obtained a speedup of 2.53 on an array of 512 pixels.

Whereas the inner loop of all other kernels operate on small sub-blocks, in the DWT kernel entire rows and columns are processed. Therefore, the speedup of the kernel is depending on the input size. So far, an input image of 512×512 pixels was assumed. Figure 11 shows that for smaller input sizes, the speedup and instruction count reductions are actually larger. For smaller input sizes, the startup overhead (processing the edges, initializing variables, etc.) dominates. This overhead is larger in the baseline kernel than is the case in the enhanced kernel. Therefore, the speedup is larger for small input sizes and flattens out for larger sizes.

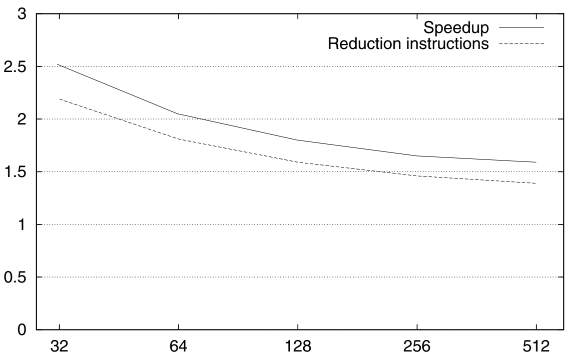

Fig. 11. Speedup and instruction count reduction obtained for the DWT kernel with different input sizes.

V. Discussion

The results obtained are good, but some questions arise about implementability and cost. In this section we will discuss those issues.

The first question is what the influence of the new instructions is on the latency of the existing instructions. Adding the intra-vector instructions to the existing functional units will increase their complexity and latency. It seems best, though, to put all intra-vector instructions in one new functional unit, consisting of a series of ALUs and interconnect such that each intra-vector instruction can be mapped to it. The latency of the total pipeline will be increased, meaning that the write-back is delayed. However, due to the existence of the forwarding network, the other instructions will not really be affected. The forwarding network will become larger, so there is some additional area cost here. Future work is to investigate a smart implementation of the intra-vector unit.

A second issue is the area cost of such a intra-vector unit. Without an actual hardware implementation, some back of the envelope calculations are the best we can do. The intra-vector unit needs five levels of fixed point operations and interconnect in between. From the floor plan of the SPE we determined that the area of the fixed point unit is 3% of the total SPE. This unit includes operations as multiplication, while the ALUs in the intra-vector unit only require addition like operations, even on small data types only. Thus the area of the ALUs of the intra-vector unit will be much smaller. It does need some interconnect though, thus to be on the safe size we assume that the intra-vector unit area will be five times 3%. Therefore, adding the intra-vector unit requires approximately an additional 15% of area.

Another question might be whether the latencies of the instructions are correct. Discussion with experts showed us that our methodology of determining the latencies is on the safe side. However, we also analyzed the speedups for larger latencies. Figure 12 shows that the average speedup only drops from 1.45 to 1.31 in the unlikely case when the latencies would be twice as large.

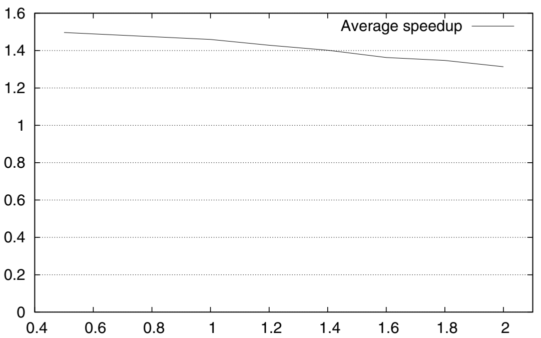

Fig. 12. The average speedup as a function of the normalized instruction latencies.

The speedups achieved seems to be worth the area cost. The average speedup is obtained on kernels only, though. The speedup on complete applications will be lower, although in the signal processing domain generally the kernels dominate the execution time. Application specific processors targeted at signal processing therefore seem to be most suitable for specialized intra-vector instructions as we presented them.

VI. Conclusions

In this paper we have shown that intra-vector SIMD instructions are an effective technique for the specialization of cores, especially in the signal processing domain. For six kernels intra-vector instructions were developed and simulations showed that a speedup of up to 2.06 was achieved, with an average of 1.45.

We showed how intra-vector instruction can reduce the data rearrangement overhead in two dimensional kernels. The efficiency depends on several factors. The ability to collapse a large number of instructions increases the efficiency, while it is limited if a lot of rearrangement overhead is required for intra-vector processing, many dependency stalls occur because the distance between dependent instructions has decreased, or if there is a low amount of DLP in intra-vector instructions.

The evaluation was performed with the Cell SPE as baseline architecture. The results on different architectures will be slightly different. The general conclusion of this paper, however, remains.

Acknowledgements

This work was supported by the European Commission in the context of the SARC integrated project #27648 (FP6).

References

[1] C. Meenderinck and B. Juurlink, "Specialization of the Cell SPE for Media Applications," in *Proc. Int. Conf. on Application-Specific Systems, Architectures and Processors*, 2009.

[2] K. van Berkel, P. Meuwissen, N. Engin, and S. Balakrishnan, "CVP: A Programmable Co Vector Processor for 3G Mobile Baseband Processing," in *Proc. World Wireless Congress*, 2003.

[3] K. Moerman, " NXP's Embedded Vector Processor can Tune into Software-Defined Radio," 2007. [Online]. Available: http://www.eetimes.eu/germany/202403704

[4] M. Moudgill, J. Glossner, S. Agrawal, and G. Nacer, "The Sandblaster 2.0 Architecture and SB3500 Implementation," in *Proc. Software Defined Radio Technical Forum*, 2008.

[5] "CEVA-XC Communication Processor." [Online]. Available: www.ceva-dsp.com/products/cores/ceva-xc.php

[6] I. Verbauwhede and C. Nicol, "Low Power DSP's for Wireless Communications," in *Proc. Int. Symp. on Low Power Electronics and Design*, 2000.

[7] Y. Lin *et al.*, "SODA: A High-Performance DSP Architecture for Software-Defined Radio," *IEEE Micro*, vol. 27, no. 1, 2007.

[8] *Cell Broadband Engine Programming Handbook*, IBM, 2006. [Online]. Available: http://www.bsc.es/plantillaH.php?cat_id=326

[9] B. Flachs *et al.*, "The Microarchitecture of the Synergistic Processor for a CELL Processor," in *Proc. Int. Solid-State Circuits Conf.*, 2005.

[10] "CellSim: Modular Simulator for Heterogeneous Multiprocessor Architectures." [Online]. Available: http://pcsostres.ac.upc.edu/cellsim/doku.php/

[11] "CellBench." [Online]. Available: https://gso.ac.upc.edu/projects/cellbench/

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

[13] "CellBuzz: Georgia Tech Cell BE Software." [Online]. Available: http://sourceforge.net/projects/cellbuzz

[14] A. Azevedo, C. Meenderinck, B. Juurlink, M. Alvarez, and A. Ramirez, "Analysis of Video Filtering on the Cell Processor," in *Proc. Int. Symp. on Circuits and Systems*, 2008.

[15] I. Daubechies and W. Sweldens, "Factoring Wavelet Transforms into Lifting Steps," *Journal of Fourier Analysis and Applications*, vol. 4, no. 3, 1998.

[16] S. Kang and D. Bader, "Optimizing JPEG2000 Still Image Encoding on the Cell Broadband Engine," in *Int. Conf. on Parallel Processing*, 2008, pp. 83–90.